Game Development with GameMaker Studio 2

Make Your Own Games with GameMaker Language

Sebastiano M. Cossu

Apress®

Game Development with GameMaker Studio 2

Make Your Own Games with GameMaker Language

Sebastiano M. Cossu

Apress®

Game Development with GameMaker Studio 2

Sebastiano M. Cossu
London, UK

ISBN-13 (pbk): 978-1-4842-5009-9 ISBN-13 (electronic): 978-1-4842-5010-5
https://doi.org/10.1007/978-1-4842-5010-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: Laura Berendson
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5009-9. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

For my family, who taught me to dream and work hard.

Table of Contents

About the Author xv

About the Technical Reviewer xvii

Acknowledgments xix

Introduction xxi

Chapter 1: Overview 1

- The right tool for the job 2
- What is GameMaker Studio 2? 3
- About Game Design 4
- About coding 5
- How to use this book 6
- Additional content 9
- Pricing 10
- Installing GameMaker Studio 2 12
  - Installing from YoYo's web site 13
  - Installing from Steam 17

Chapter 2: Hello, World! 23

- Right Sidebar 25
- Sprites 26
- Objects 28
  - Events 29
  - Code 31

Tile sets 32

Fonts 32

Rooms 32

Hello, GML! 34

Create event 35

Left Pressed (Mouse) event 39

Draw 40

Chapter 3: Card Game (Part 1) 45

The design 45

A Game Design Document primer 46

Memory GDD 47

Assets 50

From GDD to development 51

Cards 52

Implementation 53

Deck 63

Fantastic data structures and where to find them 65

Designing decks 71

Code loops 72

Making decks 75

Every day I'm shuffling 78

Chapter 4: Card Game (Part 2) 87

Finite-state Machines (FSMs) 87

From State machine to code 93

A matter of time 101

Play to win! 107

Chapter 5: Fixed Shooter 117
History of the genre 117
Space Gala (GDD) 119
Story and setting 120
Gameplay 120
Victory conditions 120
Controls 120
Menu 121
Pacing 121
Enemies 121
Game modes 121
Level 1 122
Similar games and influences 122
Target audience 122
From GDD to the game 122
Assets 123
Making features, not objects 129
Shooting 138
Designing rm_level_1 148
Game states 149
Making HUDs 157
What about victory? 160
Menu 161
Chapter 6: Shoot 'Em Up! 171
Fixed vs. scrolling shoot 'em up! 172
Space Gala v.2.0 (GDD) 173
Story and setting 173
Gameplay 173

- Victory conditions 174
- Controls 175
- Menu 175
- Pacing 175
- Enemies 176
- Game modes 177
- Level 1 177
- Level 2 177
- Similar games and influences 177
- Target audience 178
- Assets 178
- Cameras and viewports 183

Designing color-switching 194

Inheritance 198

Color shooting 203

More enemies 206

- Ain't nothing but the blues 207
- Walkers on paths 209
- Unidentified Flying…Instance! 213

Super-attack 216

How to design a good shmup level 222

Boss fighting 223

Conclusion 228

Chapter 7: Designing Bosses 231

Teaching and experimenting 234

Motivation! 236

How can we use this? 238

Chapter 8: Single-Screen Platformer....................................241
Cherry Caves....................................244
Story and setting....................................245
Gameplay....................................245
Victory condition....................................245
Controls....................................246
Enemies....................................246
Assets....................................247
Sprites....................................247
Fonts....................................250
Sounds....................................251
How to create a hero....................................251
Setting the boundaries....................................255
Everything that goes up comes down....................................259
Get a jump on!....................................262
Climbing the ladder....................................263
Controlling the game flow....................................269
HUD....................................279
How to die....................................282
Cherry-picking....................................285
Through cherries, to the star....................................286
Level design: The art of creating worlds....................................288
Check the jumps....................................289
Hard is good, too hard is not....................................289
Make it nice....................................289
Don't make it too easy!....................................289
Designing caves....................................290
Level 2!....................................291

Chapter 9: Scrolling Platformer 297
Story and setting 298
Gameplay 298
Victory condition 299
Items 299
Controls 300
Enemies 301
  Attack 301
Miscellaneous 302
Similar games 303
Assets 303
  spr_land 303
  spr_skybg 304
  spr_platform_falling 305
  spr_platform_trampoline 305
  spr_platform_moving 305
  spr_octopus_green 306
  spr_octopus_purple 306
  spr_titlescreen 307
  spr_coin 307
  spr_terrain 308
Fonts 308
  fnt_title 308
Sounds 308
The more you do it … 309
Title screen 310
Tiles and level design 318

Scrolling camera 327
Fixing and re-adapting 330
Different ways to move 338
Gotta squash 'em all! 348
Items and power-ups 354
Coins 354
Cherries 356
Creating the first level 359

Chapter 10: Designing Platformers 367
Controls are key 367
It's my fault! 369
Keep it simple! 371
Power-ups, items, and gear 373
Interesting collections 373
World makers 376
Conclusion 380

Chapter 11: Metroidvania (Part 1) 381
History 382
Isolation (Game Design Document) 383
Story and setting 384
Gameplay 384
Victory condition 384
Controls 384
Enemies 386
Attack 386
Skills 386
Maps 387

Inventory....387
Similar games....388
Assets....388
Fonts....393
Sounds....394
Creating the platforming base....395
Gamepad support!....399
Gravity, no escaping!....401
Making the leap....404
Another kick in the wall....406
Moving forward with a dash....413
The game flow....420
Warped!....429
Conclusion....434

Chapter 12: Metroidvania (Part 2)....437
About maps....438
Map makers, grids, and semaphores....440
Items and inventory....459
Creating the combat system....474
Old enemies....478
Saving Maria....485
Conclusion....496

Chapter 13: Designing Fun Games....499
Document your design!....499
Respect your game....501
Keep your player immersed....502

Autonomy 502
Competence 504
Relatedness 505
Having fun means learning 506
Conclusion 508
Chapter 14: What's Next? 511
ITCH.IO 511
GOG 515
Humble Store 517
Steam 518
End game 519
Index 521

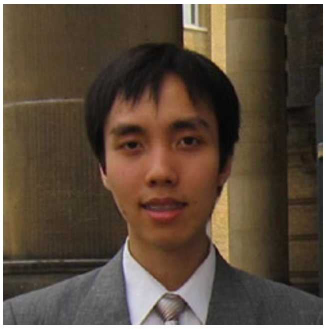
About the Author

Sebastiano M. Cossu is a software engineer and game developer.

Video games have always been his greatest passion, and he began studying game development at an early age.

Sebastiano started working with GameMaker in 2002 and has worked with every version of the software since then.

About the Technical Reviewer



Dickson Law is a GameMaker hobbyist, commentator, and extension developer with 9 years of community experience. In his spare time, he enjoys writing general-purpose libraries, tools, and articles covering basic techniques for GameMaker Studio. As a web programmer by day, his main areas of interest include integration with server-side scripting and API design. He lives in Toronto, Canada.

Acknowledgments

This book would not have happened without many aware and unaware contributors.

My warmest thanks go to my mother, the best woman I know. You taught me to write, and play games. I couldn't have made this book without you.

Thanks to my father, who introduced me to computers and video games and taught me to never give up.

Huge thanks to my sister, who was the best playfellow of my childhood – always the player one, always living her own way, the bravest person I know.

Thanks to the rest of my big family and especially my grandparents, who built it and sustained it with great sacrifice and effort overcoming the horrors of World War 2. You taught me to always smile and to hope in the future, because when the night is darkest the dawn is nearest.

Grossly incandescent thanks to my best friend, who is always there to talk and play some random Dark Souls.

I'm deeply indebted to Spandana, who believed in me and this book from day one and supported me with great humanity and professionalism. I cannot thank you enough!

I am truly grateful to all the Apress team and especially Divya and Laura, who were always available, helpful, and kind. Thank you both for all your good ideas and advices.

Special thanks to all the game designers and developers who inspired me with their works and put me on the path to game development. I owe all my knowledge and passion for video games to you and your works. I'm standing on the shoulders of giants.

Thanks, in particular, to Hideo Yoshizawa, who created Klonoa, which is the reason why I decided to become a game developer.

Finally, thanks to you, who bought this book. I hope you will find the answers to your questions between these pages!

Introduction

When I was a kid, the game industry was rapidly growing, getting bigger and shinier. There was no space for homemade games or one-man companies, and creating games was more of an elite activity. That was when I discovered GameMaker, a very accessible game engine that allowed everyone to create simple 2D games with Drag and Drop (DnD) and a basic scripting language.

In recent years, game engines for nonprofessionals became so powerful that it was possible to create complex games that started to be comparable to what the industry offered. Having the same possibilities of big companies, independent developers started to rise and publish their own games. They were different from the industry standards: there was a lot of experimental gameplay, art, deep narrative, and sometimes even controversial topics. A brave new world of video games was born, and the industry changed forever.

Today, the indie scene is one of the most prolific and successful of the entire game industry. Indie games are showcased at E3, and sometimes they even make more money than AAA games. Independent game developers are the true vanguard of the industry, and anyone can join the revolution of digital world creators thanks to those incredibly easy-to-use tools. In particular, GameMaker was used to create some of the most revolutionary games of the indie scene, like Undertale, Spelunky, Hotline Miami, Gods Will Be Watching, Risk of Rain, and Hyper Light Drifter.

Even if it's very accessible, GameMaker requires some knowledge. This book will guide you through the process of creating games with GameMaker Studio 2 (GMS2) from design to release while exploring the history of some of the most important game genres and their evolution, trying to understand the secrets behind their success.

We will start studying card games, how they streamline the concept of game to the essential, formalizing rules, and defining the game flow.

Then we will depart to a dangerous and exciting travel into the deep space, fighting aliens while exploring the evolution of the shoot 'em up (shmup) genre creating a fixed and a scrolling shooter.

We will make our journey back to Earth, to explore the success and golden rules of 2D platforming, developing a classic single-screen platformer (SSP) focusing on the basics and then evolving it to a modern scrolling platformer, introducing level design, new features, and challenges.

Finally, we will gather all the experience and knowledge gained, to design and develop our last game belonging to one of the most exciting, complex, and popular genres around: metroidvania!

As this book is not printed in color, I have placed color figures in the source code download file for your reference.

At the end of this long journey, you will have everything you need to start creating games on your own, from the idea to publishing.

I wish you good luck with your game career, and I hope you will create great games and contribute to this wonderful creative revolution!

CHAPTER 1

Overview

"How can I make video games?" This is a question I asked many times to a lot of people (and mostly to Google) when I was a kid. The desire to create games is something that nearly every true gamer happens to have at a certain point. It's something that is common between all media consumers, from books to movies to video games: we try to create the things that make us feel good. We dedicate a lot of time to video games, and they give us strong emotions and wonderful stories in return. Sometimes they help us in hard times – like if a piece of software can understand us better than a person – and sometimes they just entertain us when we are bored or we need some instant fun. We give them time, and they give us emotions and wellness in return.

Fascinated by the power of video games, some gamers turn themselves into game developers. Those kinds of developers are probably the most passionate you can find. Their mission is not only to make software but to share emotions and create worlds.

In this chapter, I will introduce you to the tools we are going to use and the topics we are going to cover in this book. I will give you a short overview of the next chapters and show you how to install GameMaker: Studio 2 on your PC or Mac so that we can start making games with no further ado!

© Sebastiano M. Cossu 2019
S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_1

The right tool for the job

Video games are a very special kind of media. They can be just fun pastimes or very trying experiences. They can teach us concepts, train us on activities, and stimulate our creativity and problem-solving. They can also tell us stories and entertain us. They use graphics, music, gameplay, and technology to do all this and much more. This makes games one of the most complex media around.

There was a time when if you wanted to make even just text-based video games, you were supposed to code in Assembly (a very low-level programming language that was different from processor to processor). Some of the simplest retro games you can think of, like *Rogue*, *Pitfall*, *Super Mario Bros. (SMB),* or *Wolfenstein 3D*, are made fully or partly in Assembly; and even if they look so simple, every one of them represents a major improvement to the media that forever changed the rules of video games.

Fortunately, we live in a time in which you don't need to learn how to code in Assembly to make a game. You can rely on game engines: software specialized in making games. They offer some very useful technical features that really simplify the process of making games, like the possibility to show an image, play a sound, or get keyboard input.

In this book – as you probably guessed by the title – we are going to use *GameMaker Studio 2*, a professional yet easy-to-use game engine that is both capable of managing 2D and 3D (we are concentrating on 2D, though). We will work in GML (*GameMaker Language),* to reach the full potential of GameMaker: Studio 2 and introduce some important coding principles. But don't worry. Our focus is about making games. Good game design and delivering fun gaming experiences are our main goals. You will learn how games are made using real-world video games as study cases.

What is GameMaker Studio 2?

GameMaker Studio 2 (*GMS2*) is a game engine that's perfect for both beginners and professionals. It supports 2D and 3D game development and allows you to create games with both *Drag and Drop* and *GML* coding. Let's take a closer look at those options.

Drag and Drop (*DnD*) is a system that allows you to structure algorithms dragging and dropping blocks that represent pieces of code. I suggest you to use DnD if you're not a coder and you're not interested in programming, but only in creating games. There is nothing bad in using DnD. Don't feel less cool if you don't mind learning how to code. Other professional tools like Unreal Engine have a visual coding solution (Blueprint) to make coding easier and faster.

In Figure 1-1, you can see an example of DnD programming in GameMaker Studio 2. From left to right, we have the object's properties, the list of the object's event triggers, and the DnD code for each event trigger.

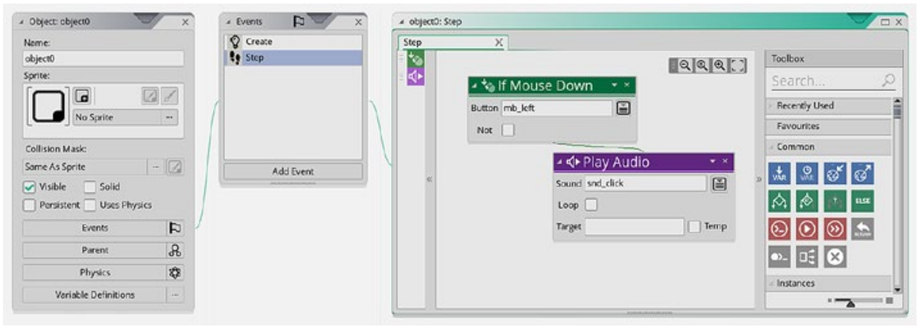


Figure 1-1. *An example of DnD programming with GameMaker Studio 2*

GML (*GameMaker Language*) is GameMaker's own programming language. It's very easy to both use and learn, and it has everything you may need to create games.

GML is a very specialized scripting language and will allow you to achieve everything you will need with little effort.GML, as a game development tool, is very similar to some content development and modding tools but allows you to create complete games from scratch with very little effort, just like more complex game development tools used in the gaming industry.

GameMaker Studio 2 is so complete and easy to use that a lot of indie developers adopted it to create some of the bestselling games of the last years like *Undertale*, *Hyper Light Drifter*, *Spelunky*, *Hotline Miami*, *Gunpoint*, *Nidhogg 2*, *Risk of Rain*, and so on.

GameMaker: Studio 2 is an *IDE* (Integrated Development Environment) – a software that contains all the tools you need to follow a certain development process. Indeed, GameMaker Studio 2 will give you direct access to a file browser to manage your resources, a text editor to write your GML code, a 2D graphic editor (similar to Microsoft Paint) to create and edit your sprites, an animation editor to make your sprite animations, a compiler (YoYo Compiler, aka YYC) to export your games to the right platform, an interpreter to run and debug your games using the GameMaker virtual machine, and everything else you may need.

About Game Design

Gaming is something that I always did with an inquisitive mind. I always played games (and I still do it now) asking myself questions like *"How is this made?" "How does it work?" "Is this fun? Why?" "Why is it not fun?"* and most importantly *"How can this be funnier?"* Only much later I realized that what I was doing all my life they called it **Game Design**.

Game Design is the process of imagining games, planning and defining all their main features. You usually start with an idea that is either a fun game mechanic or a cool story idea and then you build a world around it.

Game designers are very eclectic. They have the vision about the whole project, and so they need to understand what the best choices for every aspect of the game are, from art to music, from the game system to the level design.

Often, in small teams, game designers are also programmers or artists, but their main focus is always the design of the story, game mechanics, and levels (or game world).

To be a game designer is not only to be a developer but someone who is capable to ask the right questions and come up with the right answers. It's a continuous learning process on how to make amazing game worlds, memorable stories, and fun interactions. It's the process to make good video games.

In this book, we will be both game designers and coders, with a focus on the former. To achieve our game development objectives, we will use GameMaker Studio 2 with GML coding.

About coding

Coding is a huge part of game (and software) development and probably the hardest to master. It's based on the process of writing a list of instructions in a language (programming language) that tells your computer what to do. This list of instructions is translated by a compiler in machine code, the language of your computer, and the result is a binary file (a file made of ones and zeroes). The binary file is your game (or software). In Windows, binary (executable) files are called *EXE* files.

I am a software engineer and, making software all my life, I learned a good number of tricks and patterns that are very useful to make your code look better and do things more efficiently. I will try to teach you everything you may need about software design; but remember that this is not a programming book, so the focus is not on that, but it's on game design. We want to deliver a good game that is possibly also good software, but efficiency and software engineering is not our priority.

We will use GML to make the projects in this book, and I will do my best to cover everything you may need to know about coding and GML itself. But if for any reason you need to know more, just head to **YoYo Games** (the company that makes GameMaker) web site. They have a very comprehensive user manual about the language, with detailed description about every keyword, function, and language feature (link in the "Additional content" section).

How to use this book

Personally, I never liked much manuals that wanted you to read them from cover to cover. I am a big fan of reference books, the kind of books that allows you to read specific chapters if you are advanced in that skill that they're trying to teach you.

So, if you're a beginner with both game development and GameMaker, just read it cover to cover; if you're a more advanced user of GameMaker or a developer who likes to do things without being guided too much, just jump on the interesting chapters.

Mostly every chapter is about a specific project that will introduce you to a specific topic of game development.

Here's a little overview on the chapters' topics:

1. **Overview**: This is the chapter that you're reading, and it's just an introduction to game development and the book.
2. **Hello, World!**: In this chapter, you will create your first project with GameMaker: Studio 2 learning some basics about the software.
3. **Card Game (Part 1)**: In this chapter, you will design and develop the first version of a card game called Memory that will be about pairing

matching cards. Card games are a classic starting point of game development. Easy to design and very good to start learning basics such as sprite management, Graphical User Interface (GUI) design, implementing rules, and checking victory conditions.

4. **Card Game (Part 2)**: This chapter will conclude the development of Memory, our card game about pairing matching cards that we started in Chapter 3. At the end of the chapter, you will have completed your first game!

5. **Fixed Shooter**: This chapter will be dedicated to the creation of a fixed shooter called Space Gala. The game will be a mix of Space Invaders. And here we will discuss about one of the most important games of the golden era that defined the top-down shooter genre. Introducing vertical scrolling, bullets (of course!), and enemies. Lots of fun!

6. **Shoot 'Em Up!**: In this chapter, we will extend Space Gala transforming it into a shoot 'em up taking inspiration from classics of the genre like Ikaruga, R-Type, and Tyrian. We will introduce some more iconic features like skills, level design, and boss fights.

7. **Designing Bosses**: This chapter covers some interesting in-depth analysis of boss fights design taking as examples real-world video games that made boss fighting good.

8. **Single-Screen Platformer**: In this chapter, we will explore the design and implementation of platformer games by creating a single-screen platformer game called Cherry Caves. You will learn how to create the basic platforming system, design levels and enemies, and create your second game from start to finish.

9. **Scrolling Platformer**: In this chapter, we will design and develop a scrolling platformer, one of the most famous and long-lasting game genres. We will use this genre to introduce the player to conscious level design and to learn how to create some interesting gameplay features like power-ups, different kinds of enemies, a simple combat system, and different types of platforms to create interesting platforming sections in your levels.

10. **Designing Platformers**: In this chapter, we analyze the history of platformers and how they evolved in the years. Considerations about how to make a platformer fun and challenging are the main topic of the chapter. There is an in-depth analysis of masterpieces of the genre like Super Mario games that will help us understand the golden rules for a good platformer.

11. **Metroidvania (Part 1)**: In this chapter, we cover the design and implementation of the first part of a *metroidvania* game called Isolation. Metroidvania is a genre that is becoming more and more popular thanks to the indie market that greatly enlarged the list of games under that label. Main features of the genre are exploration, platforming, and combat. We

will start creating Isolation by using the concepts studied in the previous chapters and introducing new concepts like exploration skills (dash and wall jump).

12. **Metroidvania (Part 2)**: This chapter concludes the Isolation project. You will learn how to implement all the defining features of a metroidvania including maps, a checkpoint system, a shooting system, and an inventory. After this project, you will have every knowledge you may need to start making games by your own.
13. **Designing Good Games**: In this chapter, we leverage on game design and psychology to understand how to create fun games. We will analyze famous and successful games to understand what they did good, why they are considered masterpieces, and how we can use this knowledge to design good games.
14. **What's Next?**: Our journey ends with a little guide on how to go forward in your game development career. I will talk to you about the most convenient options to sell or distribute your game as an indie developer on the most popular digital games stores like Itch, GOG, Humble Bundle, and Steam.

Additional content

This book is heavily based on the use of GameMaker Studio 2 and revolves around the projects proposed in every chapter. So, if you're having some problem following the instructions or you just want to see the working project before you start, you can take a look at the source code on GitHub (via the book's product page located at www.apress.com/9781484250099. For more detailed information, please visit www.apress.com/source-code.) Don't worry. You will find all of them exactly as you see them in this book.

Anyway, I strongly suggest you to try and solve problems by yourself and think about the solutions, before checking for the answers in my code (or in the book). It's a good practice to strengthen your knowledge and gain experience.

If you need additional information on the tools used or the language, don't forget that YoYo Games' reference (http://docs2.yoyogames.com/) is your best friend to understand all the secrets of this wonderful software.

Pricing

GameMaker Studio 2 comes in different flavors depending on your needs and if you are a professional or amateur developer. In the following, you can find a useful table that can help you out making your decision based on your own needs.

The free *Trial* license is very good to play around with the IDE and start learning and making small games, but if you're serious about game development and you feel GameMaker Studio 2 is your tool, go ahead and choose the license that better fits your game developer needs. Remember that with the free Trial version, you won't be able to export you game; in fact, you may run it only inside GameMaker Studio 2.

I personally suggest you the Desktop license since it's the most convenient. In fact, it allows you to fully concentrate on PC/Mac game development and allows you to export on all the operating systems, including Ubuntu Linux (see Table 1-1).

Table 1-1. *GameMaker Studio 2 comes in many different flavours to please every budget and every need. This table shows a list of all the possible licenses that GameMaker Studio 2 offers*

License	Price	No limitations	Exports to …
Trial	Free!	Some functions are not available, and projects must be small	✗
Windows	35$ (12-month license)	✓	Windows
Mac	35$ (12-month license)	✓	Mac
Desktop	99$ (permanent license)	✓	Windows, Mac, Ubuntu
Web	149$ (permanent license)	✓	HTML5 (web platforms like Facebook and Instant Games)
Mobile	399$ (permanent license)	✓	Android, Amazon, and iOS App Store
UWP	399$ (permanent license)	✓	Xbox One Creators program and all Windows 10 devices via the Universal Windows Platform
PlayStation 4	799$ (12-month license)	✓	PlayStation Store
Xbox One	799$ (12-month license)	✓	Xbox Store

(*continued*)

Table 1-1. (*continued*)

License	Price	No limitations	Exports to …
Nintendo Switch	799$ (12-month license)	✓	Nintendo Store
Amazon Fire	149$ (permanent license)	✓	Amazon Fire Store
Ultimate	1500$ (12-month license)	✓	All

Installing GameMaker Studio 2

Before we can start messing around with code, sprites, and game objects, we need to install the engine. To do it, just go to the official web site (`www.yoyogames.com/gamemaker`) or the GameMaker Studio 2 Steam page and download/purchase the version you prefer (you may find useful the previous paragraph to decide).

There are some prerequisites you need to meet to run GameMaker Studio 2. Here they are:

- **Windows**: You need at least Windows 7 (64-bit version) and DirectX 11 or later. You also need a 64-bit Intel-compatible dual-core CPU and a screen resolution of at least 1024 × 768, 2 GB of RAM, and 3 GB of available space.
- **Mac OS**: You need a Mac featuring a 64-bit Intel-compatible CPU, at least 4 GB of RAM, and 3 GB of available space, running at least Sierra (10.12) or later and Xcode. It's suggested to always update to the latest versions of both Mac OS and Xcode.

I suggest you to get GameMaker Studio 2 via Steam, since it has a direct and easier support to export the game on Steam (check `https://help.yoyogames.com/hc/en-us` for more info about this). In the next section, we will look more closely on how to properly set up GameMaker Studio 2 on any supported platform.

Installing from YoYo's web site

To download and install GameMaker Studio 2 from YoYo's web site (not using Steam), you need to head to YoYo Games' official web site (`www.yoyogames.com`) and create a YoYo account. After that, you can download the Windows or Mac client via the *Download* section in the *Account Dashboard* page (as you can see in Figure 1-2).

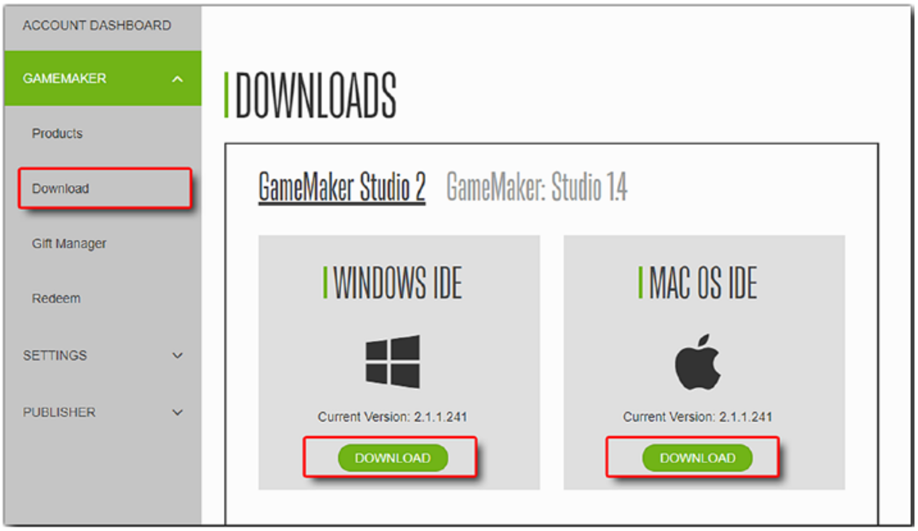


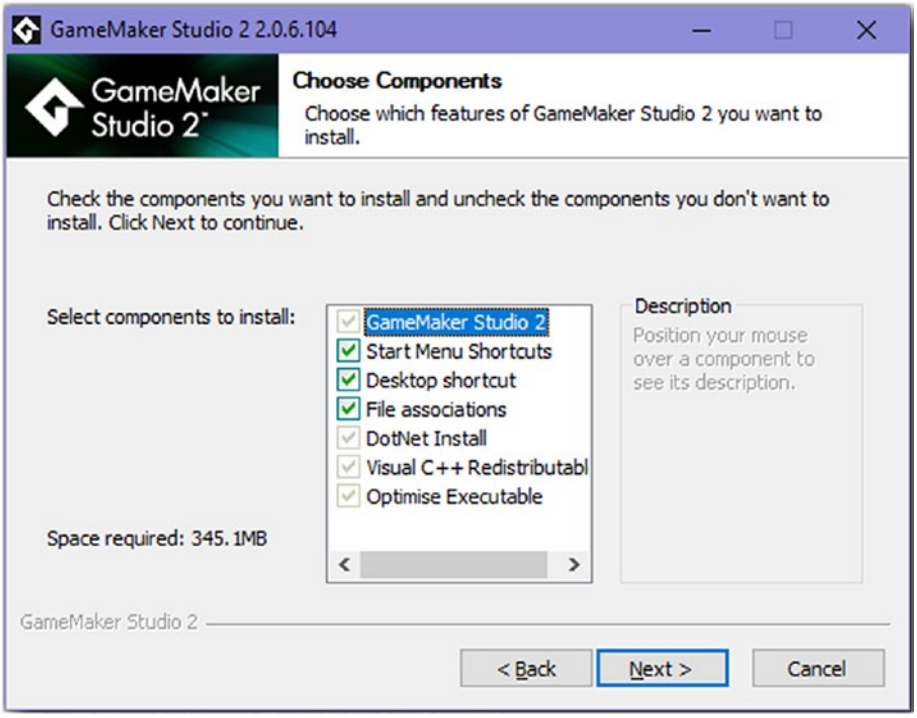
Figure 1-2. *GameMaker: Studio 2 download page*

After downloading the client, just double-click it to start the installation process.

Windows

When you execute the installer client, you will be prompted to a license agreement (that you need to accept to use the software). Note that if you're updating to a new version, you will be asked if you want to delete the previous version before continuing the installation process.

After that, you will be taken to the Choose Components screen (Figure 1-3) where you can check the additional components to install and some other options like creating Start Menu shortcuts, Desktop shortcuts, or file associations (some special extensions like *YYP* and *GML* will be associated to GameMaker: Studio 2). If you're in doubt, default choice is the safe one.



GameMaker Studio 2 2.0.6.104

Choose Components

Choose which features of GameMaker Studio 2 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- ✓ GameMaker Studio 2
- ✓ Start Menu Shortcuts
- ✓ Desktop shortcut
- ✓ File associations
- ✓ DotNet Install
- ✓ Visual C++ Redistributabl
- ✓ Optimise Executable

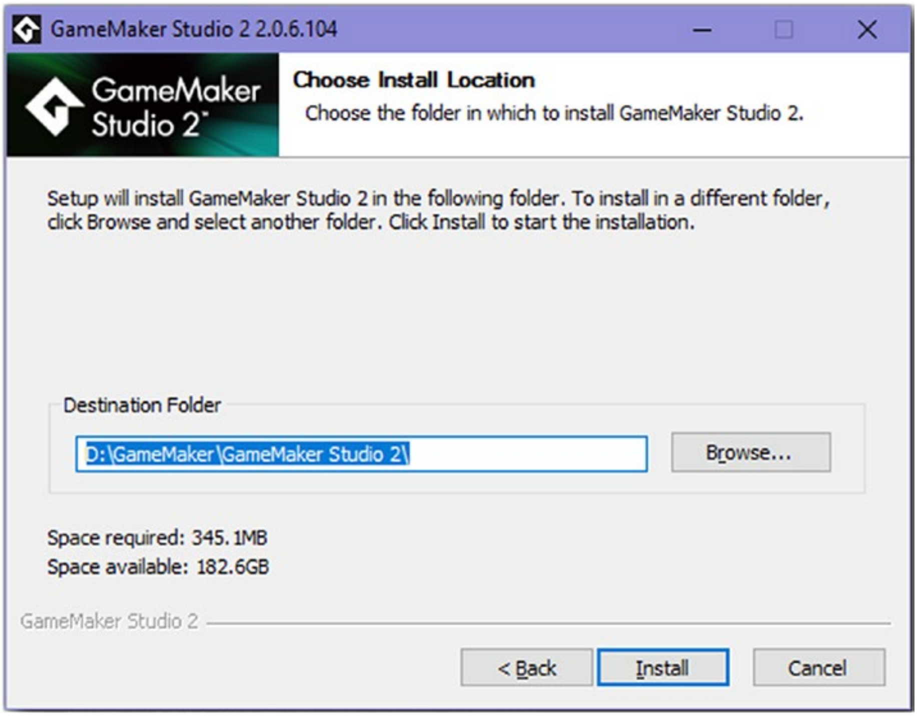
Description

Position your mouse over a component to see its description.

Space required: 345.1MB

Figure 1-3. *Windows installer's Choose Components screen*

Clicking Next, you will be asked to choose the destination of the installation (Figure 1-4). If you don't know what to do, just stick with the default choice. It will install GameMaker: Studio 2 in your main disk.



Choose Install Location
Choose the folder in which to install GameMaker Studio 2.

Setup will install GameMaker Studio 2 in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder
D:\GameMaker\GameMaker Studio 2\

Space required: 345.1MB
Space available: 182.6GB

GameMaker Studio 2

Figure 1-4. *Windows installer's Choose Install Location screen*

Note that even if you specify a destination folder, some components will be installed in the *%programdata%*, *%Localappdata%,* and *%appdata%* folders. You can change this after the installation heading to *GMS2*'s *File* ➤ *Preferences* menu.

Clicking the *Install* button will start the installation process, at the end of which you can click the *Finish* button to close the installer and start *GameMaker Studio 2*.

Additional updates may be needed by *GameMaker Studio 2* to properly work, but they will be managed by the IDE. In fact, every time you start up the software, if you are logged into your YoYo account, GameMaker Studio 2 will check for updates and automatically install them.

Mac

Choosing the Mac version, you will download a *PKG* file that you can double-click to start the installation (Figure 1-5).

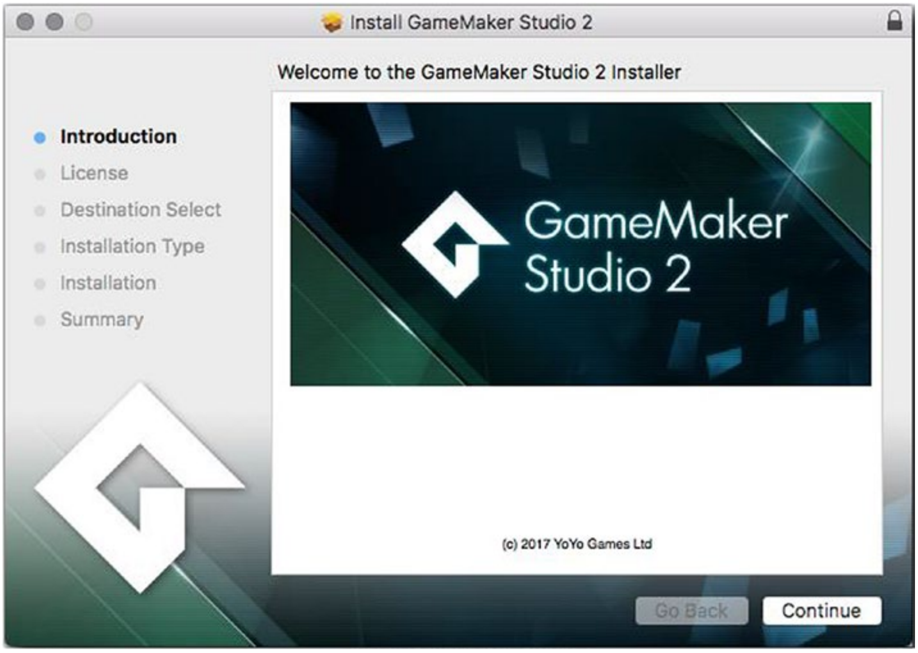
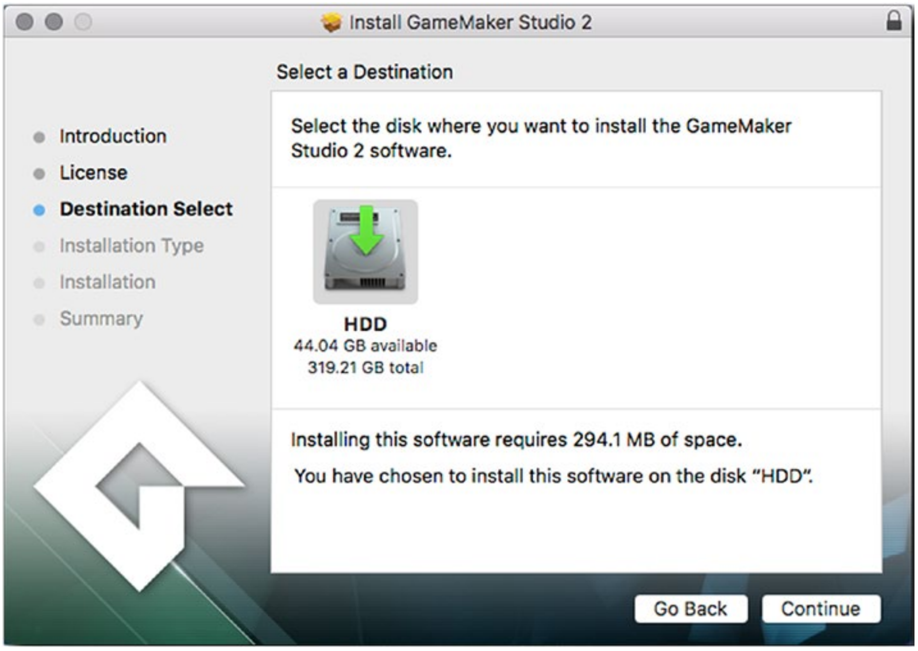


Figure 1-5. *GameMaker: Studio 2's Mac installer*

Clicking Continue on the Introduction screen, you will be prompted to accept the license agreement and then asked where you want to install GameMaker Studio 2 (Figure 1-6). A main difference from Windows is that you can't decide the folder, but only the disk you wish to use. Sticking with the default choice is the right decision if you're in doubt.



Select a Destination

Select the disk where you want to install the GameMaker Studio 2 software.

44.04 GB available
319.21 GB total

Installing this software requires 294.1 MB of space.

You have chosen to install this software on the disk "HDD".

Figure 1-6. *Mac installer's Select a Destination screen*

Just go ahead and click Continue to choose the default setup until you reach the Summary. Then click Close to finish the installation process. You will be asked if you want to move the installer to the trash folder.

You will find GameMaker Studio 2 in your Application folder, as it's the standard on Mac OS.

Installing from Steam

Installing GameMaker Studio 2 using Steam is a straightforward process. Just open your Steam client and search for "GameMaker Studio 2" in the search bar as in Figure 1-7.

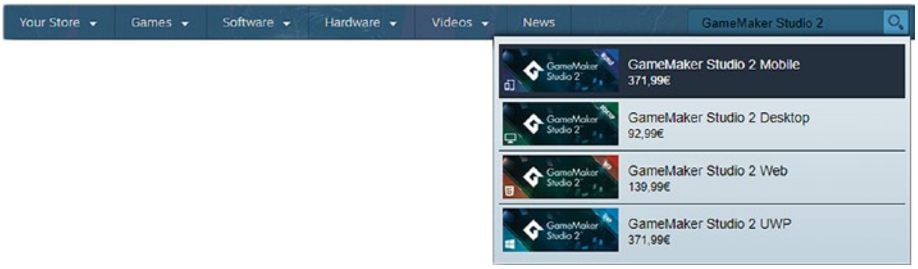


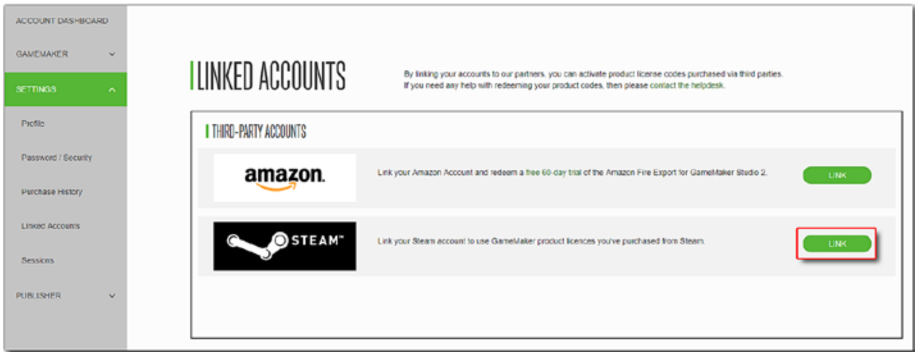
Figure 1-7. *Searching for GameMaker Studio 2 on Steam's search bar, will prompt you all the different versions of the software*

Now you just need to choose your GameMaker flavor (use the table in the "Pricing" section as a reference) and purchase/download it as any other software on Steam.

The IDE will be installed in the Steam folder as any other game, and you may access it from the Steam library or double-clicking the executable file.

Note that you will need a YoYo account to use the software, just as if you downloaded it from the official web site.

If you purchased GameMaker Studio 2 from YoYo Games' web site and you want to link it to your Steam client, just go to your YoYo account's Dashboard and open the *Settings* ➤ *Linked Accounts* menu. Now click the "Link" button next to the Steam icon as shown in Figure 1-8.



By linking your accounts to our partners, you can activate product license codes purchased via third parties.
If you need any help with redeeming your product codes, then please contact the helpdesk.

Link your Amazon Account and redeem a free 60-day trial of the Amazon Fire Export for GameMaker Studio 2.

Link your Steam account to use GameMaker product licences you've purchased from Steam.

Figure 1-8. *It is possible to link GameMaker Studio 2 to Steam even if you purchased it from YoYo's store*

Ubuntu

You cannot run GameMaker Studio 2 on Linux, but you can export games to Ubuntu Linux. To do so, you need to connect your GameMaker Studio 2 client to an Ubuntu (virtual or physical) machine.

To connect an Ubuntu PC to your GameMaker Studio 2 client, you need to install some additional software on your Ubuntu system:

- **OpenSSH**: A suite of secure networking utilities based on the Secure Shell (SSH) protocol
- **OpenAL**: A cross-platform audio application programming interface (API)
- **Clang**: A C/C++ compiler that uses the LLVM compiler infrastructure
- **Fakeroot**: A tool used to create Ubuntu packages

To install them (if you don't already have them), you just need to open Terminal, which you can find searching "*Terminal*" in the Dashboard.

Opening Terminal, you will be presented a command prompt in which you should run the following commands one by one:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential openssh-server clang
libssl-dev libxrandr-dev libxxf86vm-dev libopenal-dev libgl1-
mesa-dev libglu1-mesa-dev zlib1g-dev libcurl4-openssl-dev
```

Now, getting back to GameMaker: Studio 2, you should click the small pencil icon in the *Device* section of the *Target Manager* (Figure 1-9).

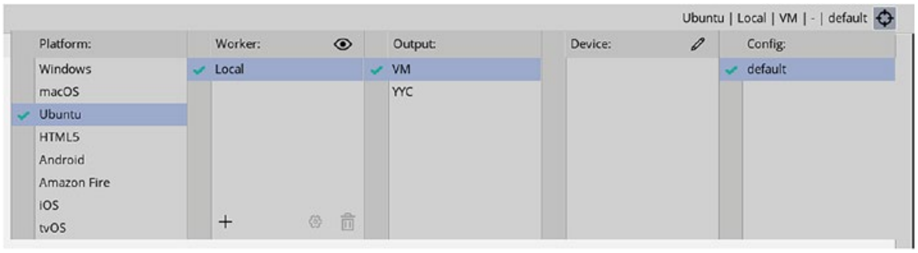


Figure 1-9. *The Target Section allows for the setup of many different target platforms, depending on which license was purchased*

This will open the *Device Editor* (Figure 1-10) that you can use to create new devices. From the left panel, select Ubuntu, and then click *Add New Device*. Now you should fill the fields to connect to your Ubuntu machine.

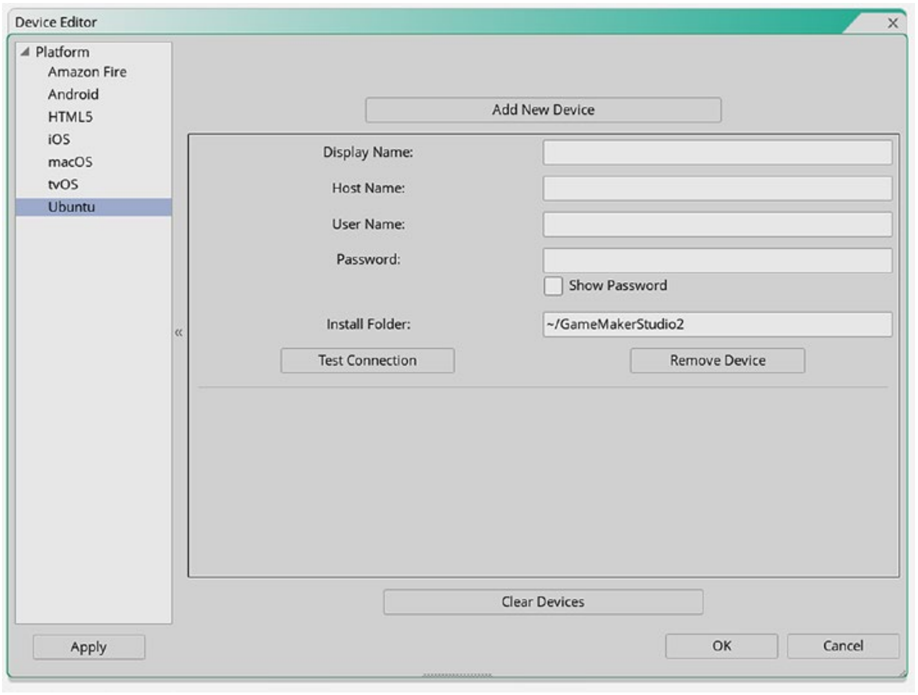


Figure 1-10. *Ubuntu Device Editor*

If you're having some problems understanding what each field means, here's a little explanation:

- **Display Name**: The name that will be displayed in the devices list.
- **Host Name**: Your Ubuntu Linux machine's local IP address (you can check it on the Settings menu in the Network icon in the Ubuntu traybar or by typing ifconfig in Terminal).
- **User Name**: Your username on the Ubuntu PC.
- **Password**: Your username password.
- **Install Folder**: It's a folder you need to specify so that GameMaker Studio 2 can install required components to run your game.

Now just click the *Test Connection* button to make sure everything is set up properly.

Ok, everything is in place! You're ready to build your game for Ubuntu Linux! To do so, select "Linux" as a target in GameMaker Studio 2 and click *Create Executable,* and the game will be exported to your Ubuntu PC in the specified folder.

All done! You're set up! We can finally start our journey into the magic world of game development!

CHAPTER 2

Hello, World!

Welcome, fellow developers! This is the very beginning of our journey into game development!

There is a tradition, in developers' culture. Every time you're going to learn a new language, framework, or library, you start with a program that just displays the message "Hello, World!"

This tradition started in the 1960s, during the writing of the manual of BCPL programming language by Prof. Brian Kernighan who wrote a program that displayed the string "hello, world" to show how I/O worked in BCPL. This example program was later used by Kernighan in his own tutorial to the C programming language (1972), and then it was used again in *The C Programming Language* book by Brian Kernighan and Dennis Ritchie (often referred as K&R).

The Hello World program was also used as the first test program for the C++ compiler by Bjarne Stroustrup (the creator of the C++ programming language).

The Hello World test program became the standard to teach and test new programming languages (but also frameworks and libraries), and we are not going to break this tradition!

In this chapter, we are going to build our own Hello World program with GameMaker Studio 2.

We will use this opportunity to explore the IDE and the elements that compose a game. We are also going to introduce GML programming writing our own Hello World program.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_2

GameMaker Studio 2 is an IDE that contains every tool you may need to build your game from scratch. Let's analyze every important part of this software! Let's open GameMaker Studio 2 (Figure 2-1)!

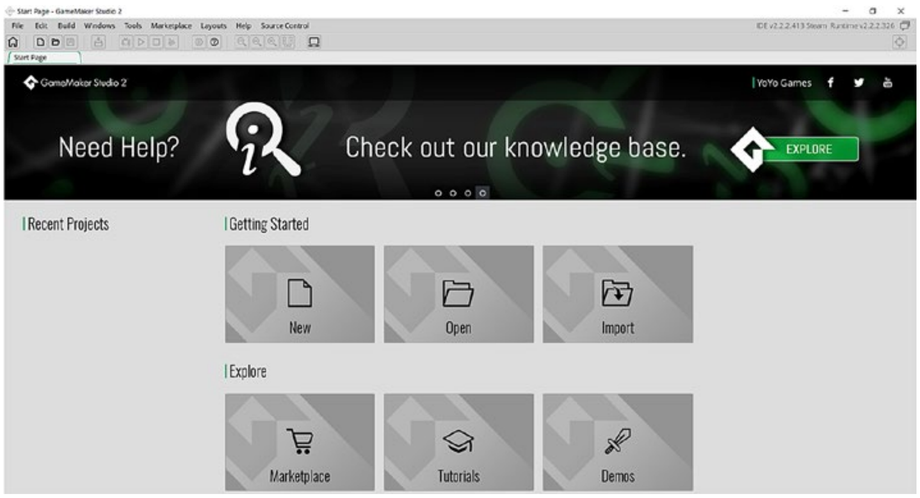


Figure 2-1. *GameMaker: Studio 2 Start Page*

You can see on the top the toolbar that allows you to do the most important actions like creating a new project or opening an existing one, saving your project, compiling and running your game, and so on. We will discuss and further analyze those actions when we will need them.

Right under the toolbar, there are the tabs. GameMaker Studio 2 makes an extensive use of tabs. They are used to manage every different view of the IDE, and they are also detachable, so that you can arrange them in any fashion you like and create a proper comfortable work environment. The current tab is colored in green.

Right under the tabs, there is the main window that shows the content of the currently selected tab – in this case the Start Page tab.

In the Start Page tab, you can find some quick links to the most common activities.

On the left, you have a list of the most recently opened projects, to quickly resume your work (just click the project you want to open).

On the right, you will find the Getting Started section, with buttons to create a new project, open an existing project, or import a project that you downloaded from somewhere else; and finally, the Explore section contains some quick links to YoYo Games' official Marketplace to buy new plugins and assets for GameMaker Studio 2 (just like Unity Asset Store), tutorials, and demos.

To start off, let's click New, in the Getting Started section of the Start Page tab, and let's create a new GML project called "HelloWorld."

Right Sidebar

Creating a new project, the interface will change a bit, introducing the Workspace tab, which will contain all the windows related to your game's objects.

On the right of the screen, there is now the Right Sidebar.

The Right Sidebar is one of the most important parts of the UI (User Interface). It contains all the elements that compose your game.

On the top of the right sidebar, you can find the target OS (Operating System) you're going to develop for (Windows, in this case); the target machine, worker, and configuration (in this case Local, default, default); and the compiler you're going to use (VM or YoYo Compiler aka YYC).

In the Resources tab (Figure 2-2), you will find a list of all the resources you have included into your project. Sprites, tile sets, sounds, scripts, fonts, levels, everything is in there.

Other than that, the Resources tab also includes the main options and configurations about your target OS and the compiler you're going to use.

Don't worry if it looks like a lot of stuff going around, we will give it a closer look later on. For now, just keep in mind that every resource you're going to use is in the Resources tab and here is where you have to create and manage.

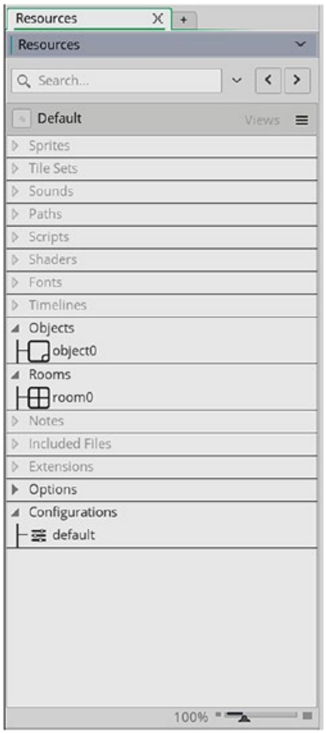
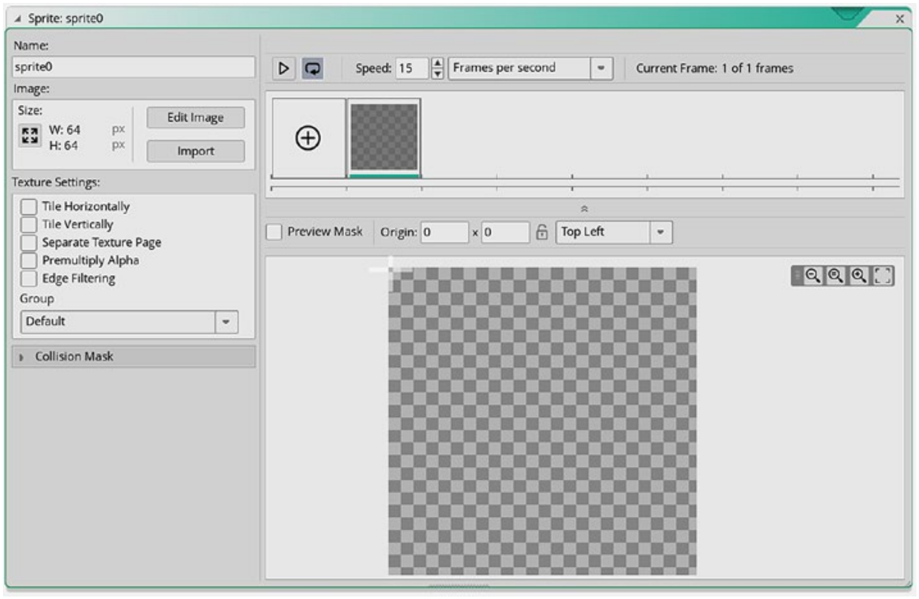


Figure 2-2. *The Resources tab lists all the assets of the game*

Sprites

Sprites are basically images drawn on your screen. They are graphical elements that can represent everything in your game from UI elements to NPCs to the player-controlled character.

To create a new sprite, right-click the Sprites category in the Resources tab and select Create Sprite.


Sprite: sprite0

Name: sprite0

Size: W: 64 px H: 64 px

Speed: 15 Frames per second

Current Frame: 1 of 1 frames

Group: Default

Origin: 0 x 0 Top Left

Figure 2-3. *This is the sprite creation window, which can be used to easily create and edit sprites*

Creating a new sprite, you will be displayed a new window in the Workspace tab (Figure 2-3). This new sprite window will show every property about that particular sprite. In this window, you can edit everything about a sprite from its name to its size and even its appearance.

The sprite window is also the place where you can define if that sprite represents an animation concatenating multiple images, how the game should consider this sprite for collisions with other sprites, and where the pivot point in this sprite is.

Sprites can be made in any graphics software, like Gimp, Photoshop, and even Paint. To make the development process faster, GameMaker Studio 2 includes a basic sprite editor. By clicking the Edit Image button in the sprite window, you can edit the currently selected image. This allows you to make your own images inside GameMaker Studio 2.

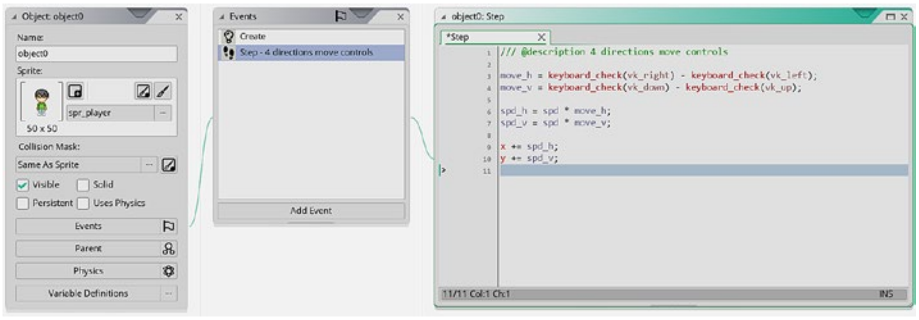
Note that a sprite can be made of more than just one image. To add a new image to the sprite, just click the big circled plus button just under the animation speed settings. You can add as many images as you like to make your sprite.

Objects

Objects are the basic blocks that build games in GameMaker Studio 2. They can be programmed to respond to certain events with predefined actions.

You can create a new object by right-clicking the "Objects" category in the Sidebar and selecting "Create Object" in the pop-up menu that follows.

Creating a new object, just like creating a new sprite, will display the object window (Figure 2-4) that will allow you to define the object's properties and to program it to respond with predefined actions to certain events.



Name: object0
Sprite: spr_player
50 x 50
Collision Mask: Same As Sprite
[✓ Visible] Solid
Persistent Uses Physics

```
/// @description 4 directions move controls

move_h = keyboard_check(vk_right) - keyboard_check(vk_left);
move_v = keyboard_check(vk_down) - keyboard_check(vk_up);

spd_h = spd * move_h;
spd_v = spd * move_v;

x += spd_h;
y += spd_v;
```

Figure 2-4. *The Object Editor allows for the creation and edit of objects. It is also the place where you can program the behavior of the object*

Objects can have a sprite associated, so that they are visible on screen. You can also define a collision mask that is different from the sprite defined.

The object window is also the place where you can define things like if you want to use the predefined physics engine, objects hierarchy, and of course events.

You can use objects to create instances. Instances are copies of objects that live inside the game. They are independent of one another, and they all work with the rules defined in the object from which they generate.

Think of an object as a blueprint of a house. From that blueprint, you can create two identical houses. Then you can put different people and furniture in them, and you can paint them differently and demolish one of them (or both). Whatever thing you do to the houses created from the blueprint, it's not going to affect the blueprint itself, which can still create more houses.

Events

When you start a game, it starts to check for a series of things and react to changes. This repetition of actions and checks is called game loop.

For example, a clicker game constantly checks if the player is clicking the left mouse button and acts accordingly.

You can direct the logic of a game, by controlling the game loop, and you do that by programming events.

An event is a discreet moment inside the game loop.

GameMaker Studio 2 offers a set of predefined events:

- **Create**: It occurs when the object's instance is created inside the game.
- **Destroy**: It occurs just before the object's instance is destroyed.
- **Clean Up**: It occurs when the object's instance is destroyed, the room changes, or the game is closed.
- **Step**: It occurs every frame, so its rate of recurrence depends on the number of frame per second (FPS) your game runs.

- **Alarm**: It occurs every time a certain timer runs out. In GameMaker, you can set up timers (called alarms) to count down from a certain value to zero. When a timer finishes counting, it triggers the Alarm event.
- **Draw**: This event runs once per frame per view and is the one that governs what you see on the screen when you run your game and is split into various separate "sub-events" that allows you to better organize the drawing of the graphical elements on the game screen.
- **Draw GUI**: This event runs once per frame and is specialized in managing Heads-Up Display (HUD) drawing. When you're drawing in this event, keep in mind that coordinates won't change: 0,0 will always refer to the top-right corner of the screen, not the room.
- **Mouse**: This allows you to wait for a mouse-related event (right/left clicking, hovering, etc.) and execute actions when that event occurs.
- **Mouse Global**: This occurs every time a mouse event is triggered, regardless of the instance. For example, if you want to execute an action when you click, regardless of where the mouse pointer is positioned, you should use Mouse Global, instead of Mouse.
- **Key Down**: This event triggers for all the time the defined key is held down.
- **Key Pressed**: This event triggers every time the key is pressed.
- **Key Up**: This event triggers every time the key is released.
- **Gesture**: This event allows you to specify a gesture that you want to use as a trigger.

- **Collision**: This event occurs when an instance collides with another instance.
- **Other**: This allows you to specify other events related to GameMaker: Studio 2 features like starting/ending the game, entering/exiting rooms, and so on.
- **Asynchronous**: This allows you to select an asynchronous function as a trigger, like HTTP requests, In-App purchases, and so on.

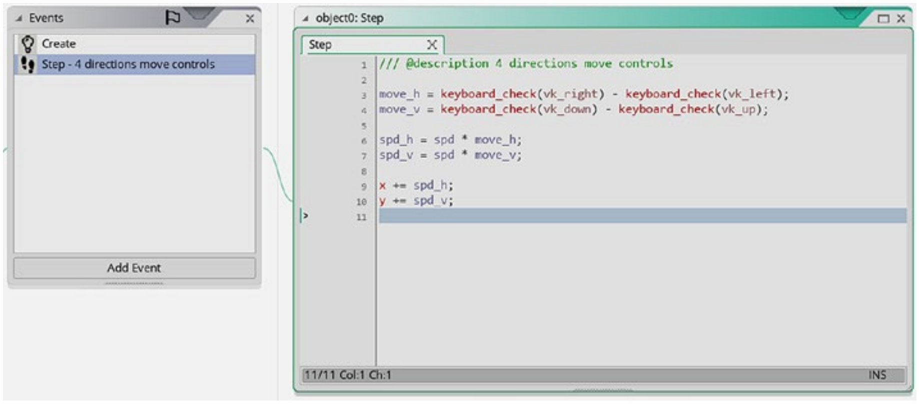
For more advanced or customized events, you will want to use a combination of events and actions.

Code

When you create a new event in a GML project, the IDE will open a window related to that event where you can write your GML code (Figure 2-5).

Think about GML code as a list of instructions or actions that you want your game to execute when that specific event triggers.

If you are confused about what to write and how, don't worry! We will cover this exhaustively in this book!



```
/// @description 4 directions move controls

move_h = keyboard_check(vk_right) - keyboard_check(vk_left);
move_v = keyboard_check(vk_down) - keyboard_check(vk_up);

spd_h = spd * move_h;
spd_v = spd * move_v;

x += spd_h;
y += spd_v;
```

Figure 2-5. *An example of GML code associated to a Draw event*

Tile sets

A tile set is basically an image that represents a collection of tiles with which you can build up your level. Tiles are just squares of fixed size that represent the fundamental graphical element of a level.

Like tiles on a floor, the same concept is applicable to tile sets and levels in a game.

To create a tile set, just right-click the Tile Set category in the right sidebar and select Create Tile Set.

Fonts

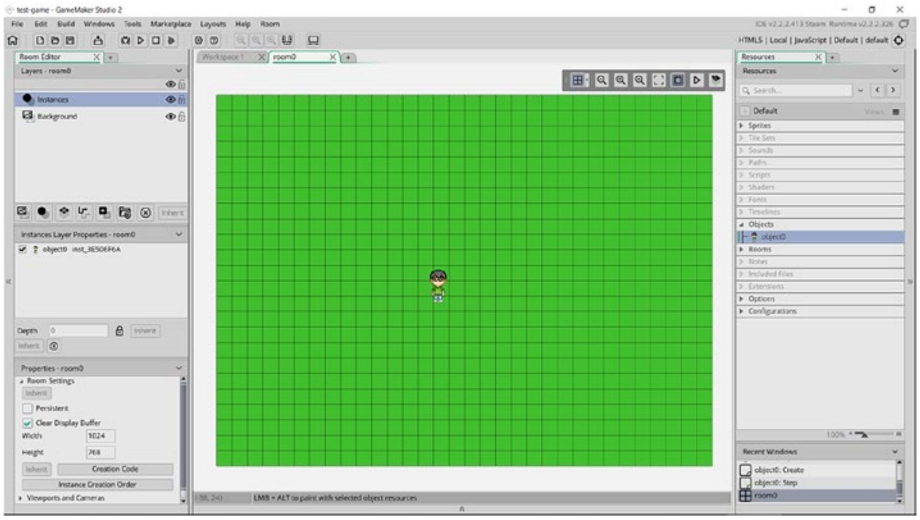
Fonts are just like the ones you use in word processing software like Microsoft Word or similar. They are used to create texts with customizable aesthetics. You can use the ones installed into your system or you can download them from the Internet and import them into your project.

Rooms

A game in GameMaker: Studio 2 is organized in rooms. Rooms are levels for your games. You start your game in the first room listed in the right sidebar. All the rooms should be connected so that you can pass from one to another.

You can use rooms to display game levels, menu screens, or whatever you like.

To create a room, right-click the Room category of the Sidebar and select Create Room.



object0 inst_3E5DEF6A
Depth 0
Persistent
Clear Display Buffer ✓
Width 1024
Height 768

Figure 2-6. *An example of a generic game room in a test game*

Opening or creating a new room will open up a left sidebar with all the room properties listed (Figure 2-6). Here you can edit things like the following:

- **Layers**: They allow you to manage objects' drawing order, in the room.
- **Room Settings**: Properties of the room, like size and resolution.
- **Viewport and Cameras**: Useful to manage how the game is viewed by the player (window size and others).
- **Room Physics**: Allows you to set up some settings for the physics engine.

The room in Figure 2-6 shows a room with a character. You can reshape a room dragging and dropping objects into it and editing its appearance with tile sets.

Remember that the first room in the Rooms list is always the first to be shown when the game starts.

Hello, GML!

Now that we have our empty project and we saw a bit of the interface, let's do what we're here for: let's create our Hello World program! This will be a program that displays an image and a "Hello, World!" message.

First things first, we need to create a new sprite. To do it, right-click the Sprites category, in the right sidebar, and select *New Sprite*.

You can both create the image for the sprite from scratch clicking the Edit Image or import it selecting the Import button. I will use a handmade strange guy as a sprite image. Don't forget that you can find any resource used in this book on GitHub.

Give the sprite a name that makes sense, like `spr_funnyguy`.

It is important to stick with a naming convention in your project. It's very common in GameMaker to use as a prefix for any resource two or three letters that represent the type of that resource. In this case, "spr" stands for sprite.

Now right-click the Objects category in the right sidebar and create a new object. Let's call it something similar to the sprite, in my case obj_funnyguy.

We want our object to be visible in the room, so we need to associate a sprite to it. Click the *"No Sprite"* text right into the object's properties section and select the sprite you just created for this object.

Let's stick with the default collision mask (same as sprite) and move right to event programming.

We want this object to display a text on a random position inside the game room every time you click it.

For this purpose, we need to use the Create event to set up some initializations and the Mouse (Left Pressed) event to move and show the text every time the user clicks the object.

Create event

Click the Events button and select **Create** event. You will be displayed an Events window with the Create Event tab opened and a special sidebar called Toolbox from which you can drag actions and drop them into the Create Event tab.

We need to randomize the horizontal and vertical position of the text. To do that, first we need to create two containers for those two values.

Introducing variables

Variables are containers that we use to store values that we need for calculations or other actions in our games.

You probably already met something like variables in your life. Just think about pi; it's a label associated to the value 3.14. That's exactly what a variable is: a label associated to a value. To be fair, pi is a constant; that means that it will always be associated to 3.14, no matter what. The cool thing about variables, instead, is that you can change the value associated to that label as many times as you want.

Variables can contain different data types. The most common are undefined, strings, numbers, and Booleans.

Undefined is a special data type that means *no-data*. In other programming languages, it's called NULL and represents the concept of void. You can check whether a variable contains the type undefined by using the built-in function *is_undefined*. Check the official documentation for more information.

Strings are simple text expressed between quotation marks. You mostly use them to write messages to show on screen. Strings can be manipulated using some specific built-in functions and can be concatenated by using the plus sign (+). Check the official documentation for more information about that.

You can declare a string like this:

```
var my_string = "Hello, World!";
```

Numbers are just real numbers stored as 32-bit floating point. You can operate on numbers using the most common mathematical operators: plus to add (+), minus to subtract (-), slash to divide (/), and asterisk to multiply (∗). There are also some more advanced functions offered by GameMaker. We will check some of them further on.

You can assign a number to a variable like this:

```
var my_number = 24.4;
```

Booleans are a special type of data that can either be true (1) or false (0). You can operate on Booleans using logical operators like *AND*, *OR*, and *NOT*. The result of operations on Boolean values is another Boolean value. The combination of multiple Boolean values concatenated by logical operators is called Boolean expression.

AND is a binary operator expressed with *and* or *&&*. It can be used to compare two Boolean values and returns true when the two values are both true; otherwise, it returns false.

Eg.

```
var my_true = true;
var my_false = false;
var result = my_true and my_false; // FALSE
result = my_true and my_true; // TRUE
result = my_false and my_false; // FALSE
```

OR is a binary operator expressed with *or* or ||. It can be used to compare two Boolean values and returns true when at least one of the two values is true; otherwise, it returns false.

Eg.

```
var my_true = true;
var my_false = false;
var result = my_true or my_false; // TRUE
result = my_true or my_true; // TRUE
result = my_false or my_false; // FALSE
```

NOT is a unary operator expressed with *not* or *!*. It can be used to negate the value of a Boolean value: if the original value is true, it returns false; otherwise, it returns false.

Eg.

```
var my_true = true;
var my_false = false;
var return = not my_true; // FALSE
return = not my_false; // TRUE
```

In GML, we can declare a variable in many ways, depending on how long we want it to live on. In fact, variables have an area of reach, which is where they operate and can be accessed. This is called **scope**. Scope determines the lifespan of a variable and defines if a variable is local, global, or instance related.

In GML, variables can be as follows:

- **Instance Related**: The most common variables. They live inside the instance for all its life cycle. They are destroyed when the instance is. Instance variables can be accessed by other instances using the dot notation (instance.variable).
- **Local**: Local variables are declared using the **var** keyword. They live only in the block of code in which they are declared, and they're destroyed when the game exits that code block.

- **Global**: They live on for all the duration of the game. They belong to the game itself and not to an instance. You must declare them using the **global** keyword. Global variables can be accessed by all the instances in the game.
- **Built-In**: Those are variables built-in in some elements of GameMaker: Studio 2 (like objects, rooms, etc.). You can access them by using the dot notation (object. variable). Note that built-in variables are never local, but they can be instance related or global.

The way to define those three kinds of variables is shown in the following code:

```
foo = 10; // this is an instance variable
var bar = 100; // this is a local variable
global.foobar = 1000; // this is a global variable
```

Note that the // prefix tell GameMaker that everything that follows until the end of line is a comment, so it shouldn't be treated as an instruction (or action).

The var keyword is used to tell GameMaker that we are going to define a local variable, while we don't need any special prefix to define an instance variable. The var keyword is followed by the variable name and the value we want to assign to it. The assignment operator is the equal character (=).

To define a global variable, you need to use this strange syntax (as also shown earlier):

```
global.<variable_name> = value;
```

The semicolon, in GML (and many other languages), indicates the end of the instruction. You must terminate every instruction with a semicolon (with some exception that we will explain later).

Now that we know what a variable is, we can declare a couple of them to store X- and Y-coordinates for the text to be displayed. So just put those lines inside the **Create** event code:

```
txt_x = 0;
txt_y = 0;
randomize();
```

We can easily figure what the code that we wrote in the Create event means:

```
Create an instance variable called txt_x and give it value 0.
Create an instance variable called txt_y and give it value 0.
```

```
The third line is a bit trickier.
```

randomize() calls the randomize function (explained later) that sets the RNG (Random Number Generator) seed to a random number.

A function is a piece of code that contains a list of instructions to execute.

Simply put, a function is a piece of code that given an input (that can be empty) executes some actions and returns a result (that can be empty too).

In this case, randomize() takes no input (actually an empty input), executes the action of randomizing the RNG seed that will provide us the random numbers, and returns no output (an empty output).

Easy, isn't it?

Left Pressed (Mouse) event

Now that we have our variables set up, we need to randomize them based on the room dimensions.

We will use an 800 × 600 room, so our text's X- and Y-coordinates shall not grow bigger than those values.

To do that, we will generate the value of txt_x in a 0–800 range and the value of txt_y in a 0–600 range.

So create a new event and choose the Mouse option. You will see a number of different events related to the mouse device. Just click the Left Pressed event.

Now a Left Pressed Event tab will be displayed into the Event window so that we can start dropping actions inside it.

When the Left Pressed event occurs, we want to generate a couple of random numbers and assign them to our txt_x and txt_y variables.

To do that, we are going to use this code:

```
txt_x = random(800);
txt_y = random(600);
```

The random(max) function does take an input (max – that must be a number) and returns a random number between 0 and max-1 as a result of its actions.

So, in the preceding code, we are generating a random number between 0 and 800 and assigning it to txt_x; and we are generating a number between 0 and 600 and assigning it to txt_y.

Note that if you want to define the minimum value of the random range, you just need to add that value to the result of the random function like this:

```
var adult = 18 + random(100);
```

Or you can use random_range(min, max), where *min* is your minimum number, while *max* is the maximum, so it will return a number between *min* and *max-1* like this:

```
var teenager = random_range(13, 19);
```

Draw

Now you just need to display the message at the randomized coordinates.

But before this, we have to create a new font to use!

Right-click the Fonts section in the right sidebar and select Create Font.

A new font will be created and displayed with a dedicated window in the Workspace. There is really nothing much to say at this stage about fonts, since we are going to use a standard one.

Click Select Font and choose Arial. Now give this font a reasonable name like fnt_arial. That's it!

Go back to our obj_funnyguy and create a new Draw event.

The Draw event is triggered every time the room is being drawn on screen, meaning every time the room shows you all its graphical objects.

The first thing to do in a Draw event is to draw the object itself using the **draw_self()** function, or its sprite won't be shown during the room drawing.

Note that you don't need to do it if you're not using the Draw event. This is because the room will use a default Draw event containing the **draw_self()** action for every object that doesn't specify a custom Draw event.

We also have to set the font we just created to be active, set its color, and then define the text to show.

To do that, we will use **draw_set_font, draw_set_color,** and **draw_text** functions.

Let's write some code to better understand.

Create a Draw event into the obj_funnyguy object and put inside this code:

```
1   draw_self();
2   draw_set_font(fnt_arial);
3   draw_set_color(c_black);
4   draw_text(txt_x, txt_y, "Hello, World!");
```

Line 1: The draw_self() function draws the object's own sprite on screen.

Line 2: The draw_set_font(fnt) function defines fnt as the font that is going to be used to display the text (in this case fnt_arial).

Line 3: The draw_set_color(col) function defines col as the color that is going to be used to display the text (c_white corresponds to the white color).

Line 4: The draw_text(x, y, msg) function displays the message msg (can be a string or a variable name) at the coordinates x and y, inside the current room.

That's it. Now click the Run button in the toolbar or press F5 to compile and start your first GameMaker Studio 2 project!

If you followed all the steps, the room will be displayed along with our funny guy and the "Hello, World!" message in the top-left corner of the room (Figure 2-7). Every time you will click the funny guy, the message will change its position to a random one inside the room.

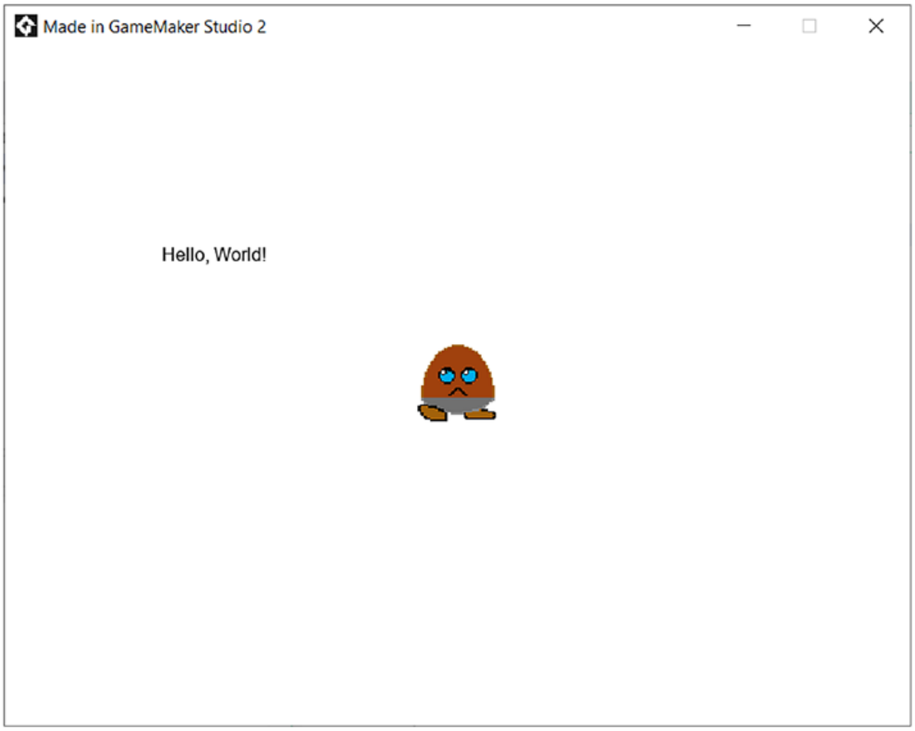


Figure 2-7.

Congratulations! You made the first step toward video games development! Cheers to that!

In the next chapter, we will go further, starting to design and create real video games. You will learn how to create a Memory card game with GML and GameMaker Studio 2.

TEST YOUR KNOWLEDGE!

1. How can you access the resources of your game in GameMaker Studio 2?
2. What is a sprite?
3. Can you create the image of a sprite inside GameMaker Studio 2?
4. What is an object?
5. What's the difference between objects and instances?
6. What is the game loop?
7. What is an event?
8. Can you give an example of an event to control mouse input?
9. Can you give an example of an event that occurs when an instance is destroyed?
10. What is GML?
11. What is a tile set?
12. How can you create customizable texts inside your game in GameMaker Studio 2?
13. What is a room?
14. What is a variable?
15. What is the scope of a variable?

16. How many different variable scopes exist in GMS2?
17. What is a global variable?
18. How can you randomize a value?
19. How can you draw a text in your game?

CHAPTER 3

Card Game (Part 1)

Card games are always one of the best starting points to learn game development. They feature all the characteristics that a game needs, but they often have very streamlined graphics and aesthetics, which is good and makes them an easy pick for game development's beginners.

In particular, we are going to develop a **Memory** card game (also known as Concentration or Pairs).

The design

Memory rules are pretty straightforward: a deck of cards made of pairs is shuffled, and the cards are placed facedown. Each turn, the player picks two cards; and if they match, the player wins the two cards; if they don't match, the cards should be put back in their place facedown.

Rules are pretty clear, but how can we apply them to a computer game? We have to think about how to translate every real-life action in a video game action. To do that, we need something like a blueprint of what we are going to create. In game development, we call this blueprint a **Game Design Document** (GDD).

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_3

A Game Design Document primer

A Game Design Document (GDD) is a design specification of a video game. It describes the game rules, its goals and mechanics, the ways the game can be played, and everything else concerning the way it should work. In cases of games with a story, the GDD also includes story background, character descriptions, and so on.

The GDD is a vital piece of documentation for your project, even if you are an indie developer or a total beginner. It's very important that you write your GDD with a simple language making it easy to read, since you and your team will probably need to read it multiple times. You should also keep in mind that a GDD is a living document. Things will change during development: new ideas may come in mind, and things that worked in the first draft may not work anymore once implemented. A game design document evolves with the development and should not be written in stone, because its role is just to fix the ideas on a piece of paper to have a path to follow.

To help you keep the GDD simple, you may (and should) use images, graphs, and other graphical elements to better give the idea of what you mean. Mockups and concept art are very welcome and help a lot with the general comprehension of the project and its goal both from a technical and a narrative point of view.

Now, without any further ado, let's write our very first GDD. Just be aware that our first GDD will be littered with comments and explanations of what we are doing (things that you don't want to keep on a GDD, but are crucial in this book for you to understand).

Note Writing a design document is crucial to better understand how to develop your project!

A common mistake is to think that a project is too simple to have a design document and that making one would be a waste of time. Keep in mind that a design document is never a waste of time! It's the opposite: a design document is going to save you a lot of time, because it lets you face development difficulties and implications before they happen. This allows you to better organize your work and do a brilliant job!

Memory GDD

Memory is a single-player card game in which the player has to find all the matching pairs of cards in a deck of covered cards. To be more challenging, the game should be time-based.

Rules

- The game is time-based (the player has to reach the goal before the time expires).
- The player can flip (and compare) only two cards per turn.
- The cards match if and only if they present the same graphics.
- If the player finds all the matching pairs, the game is won.
- If the time runs out, the game is lost.
- The player can reset the game anytime losing their progress and resetting the timer.

Tip Rules are one of the most important aspects of the GDD. They define the laws that your game should never break and are the core of its **fun factor**. Never forget: your game, no matter the genre, should always be fun. Writing good rules for your game is half the work done!

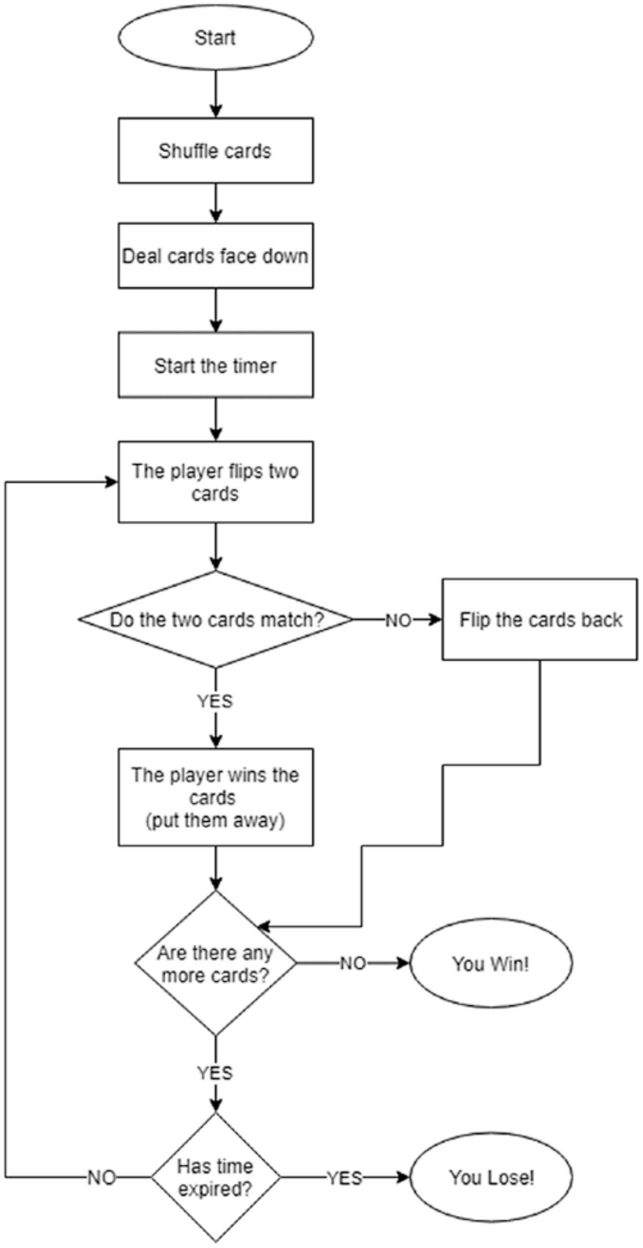


Figure 3-1. *Memory card game logic flow*

Game flow

When the game starts, the player is taken to the game screen so that they can immediately start to play.

The gameplay follows the rules we just defined in the previous section, as you can see in the flow chart in Figure 3-1.

Note In this case, it is very convenient to use a flow chart. Indeed, the flow chart makes very easy to see and understand the game loop and the flow that the whole application should follow.

Tip Having a good understanding of how your game should act allows you to better understand how to implement it. That is why a design document is extremely useful in software development.

Similar games

There is a huge quantity of Memory card games for PC. Probably one of the most relevant is Mickey's Memory Challenge (1990) by Walt Disney Computer Software (now Disney Interactive).

Game modes

Time-based mode is the only mode we want to support. We want to keep the project simple, since it's our first project.

Note In the future, you may want to extend the game with some more modes, for example, a free mode, in which you can play without worrying about the time, or a 3D mode like Mahjong, in which you can only pick cards that are on the top.

Keep in mind that one of the best things about Game Design is that it gives you endless possibilities to improve the fun in a game. Even if you have to deal with classics, like we are doing, you can experiment and borrow ideas from other games (or other fields) to make it better and funnier. After all, complex games like Total War or Pokémon game series are just evolutions of Rock-Paper-Scissors.

Target audience

This is a game that suits every age.

Target system

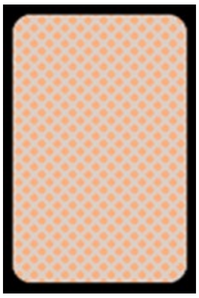
The game is designed to be playable on a PC using a mouse.

Assets

For this game, you will need a set of ten sprites for the cards: nine sprites for the different types of cards and one sprite for the back of the cards.

You can take inspiration from any kind of existing cards. In the following, there is an example of what those sprites should look like.

spr_cardback



Pivot Point: Middle-center

Size: 100 × 150

spr_rain

Pivot Point: Middle-center

Size: 100 × 150

From GDD to development

The GDD I just presented you is pretty basic, but still has anything we need to start creating our game.

From the rules of the game and the game flow, we know that we will need to implement some concepts:

- **Card**: The basic element of the game. A card can be flipped and has two faces and a type.
- **Deck**: A collection of cards. It can be shuffled and dealt.
- **Timer**: Counts the time passed. It can be reset to the initial state anytime with the enter key.
- **Game Controller**: An object that checks on rules and victory conditions.

Those four concepts are everything we need to make and play our Memory game. But how can we make them in GameMaker Studio 2? In the next sections, we will cover the complete design and implementation of the first half of the game, and we will complete it in the next chapter.

Let's start from the cards!

Cards

The card object is the basic brick to build and play our game. We need to design it thinking about what properties we want it to have and what actions we want to use it for.

Let's do this using our experience with card games in real life. Thinking about properties for our cards, for example, we know that a card object into our game, to be like a real-world card, should have two faces (back and front), one type/value (represented by its graphics, just like the combination of color and rank in poker cards defines a unique card and its value in the deck), and an index that we can use to know its position inside the deck.

Summarizing what we just said, our card should have those properties:

- **Type**: The figure represented on the card
- **Face**: Tells us if the card is faced down or faced up
- **Index**: Represents the position of the card inside the deck

The next questions are: *How do we want to use our card into the game? What actions can we apply to the card object?*

From the rules, we know that the player should be able to flip two cards to see their type and then check if they match. So we need an action to change the face of a card from front to back and another action to store in memory the type of a card to do a later check. We can call those actions flip and select:

- **Flip**: Changes the visible face of the card. If the card is faced up, it turns it facedown and vice versa.
- **Select**: Stores the current card in memory for a future check.

GameMaker: Studio 2 is built around the **event-driven programming** paradigm. So, to implement the Flip and Select actions, we need to understand which event should trigger them.

Since we are playing on a PC, we want to use the click of the left mouse button to select and flip our cards. So, when the player clicks on a card, the card is flipped and then stored in memory to be checked with a second card. The selected card will remain selected and faceup until a second card is clicked. When two cards are selected, we need to check if they match and then clear the selection.

Translating all that we just said about the card object in GML concepts, we need to create an **obj_card** game object and set those two events:

- **Create**: This will include the initialization of the card's properties (type, face, index).
- **Left Mouse Button Pressed**: This will contain the Flip and Select actions code.

We defined everything we need for the card object design. We can go ahead and implement it in GameMaker: Studio 2.

Implementation

First of all, we need to create a number of sprites for our different cards, plus one for the back of the cards. Let's say eight different designs for the front and one for the back (total: nine sprites). Feel free to use my cards; you can find them in the official GitHub repository of this book.

To represent the different kinds of cards in the code, we will use a variable to point to a number (from 0 to 7) that represents the sprite we want to assign to that card. We now need a way to store in memory all the different cards' sprites so that we can access them via code. We can do this by using an **array**.

Array

An array is a collection of elements ordered sequentially (Figure 3-2). The most basic kind of array is the one-dimensional array: a list of elements linked to one another and accessed using a sequential number starting from 0 (that tells us that item's position in the array).

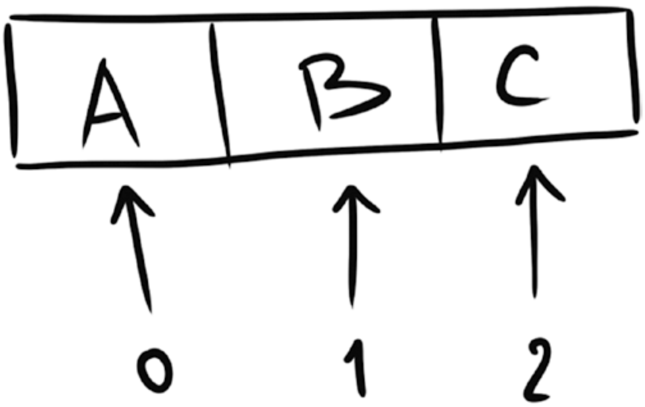


Figure 3-2. *An array made of three elements*

A two-dimensional array is a table (or grid) made of two arrays (Figure 3-3), as if they were the rows of the table. To access the elements in a two-dimensional array, you need two indexes: one to refer to the row and the other to refer to the column.

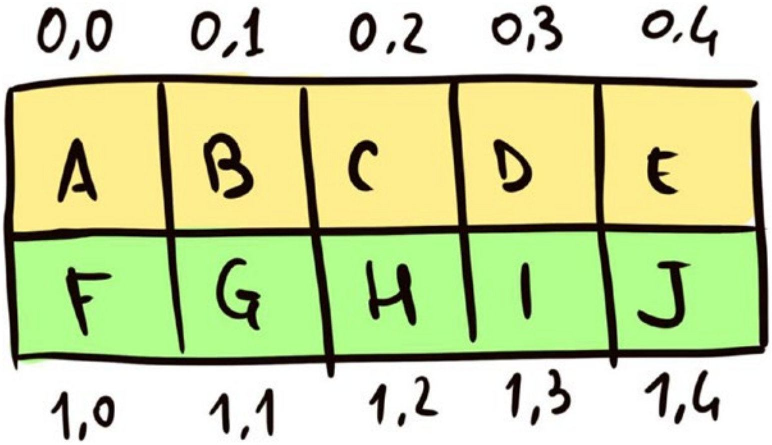


Figure 3-3. *A two-dimensional array*

Arrays are very useful to organize information in a schematic form so that you can easily access them. Here are some real-world applications of arrays in games:

- Menus
- RPG stats
- Inventory
- Game statistics
- Others

To create an array, you just need to initialize it:

```
my_array = [ 10, 20, 30 ];
```

You can access the single elements by using their index like this:

```
my_array[0] = 10;
```

The preceding line initializes the first element of the array named my_array (element number 0) to a value equal to 10.

To add more items to the array or change their value, you can do the same thing for every other item:

```
my_array[1] = 20;
my_array[2] = 30;
```

The preceding code initializes (if they don't exist) or assigns new values to the second and third elements in the array **my_array**.

To access the value from an item in an array, we just call the array with the right index, like this:

```
var third_element = my_array[3];
```

Caution Unlike other programming languages, like C/C++ and Java, GML doesn't ask you to declare the size of the array when you create it. You can freely extend the array after the creation.

An array of sprites

In our game, an array can be really useful to store all the cards' sprites so that we can use them in our card object. We will do this inside our card object.

So, without further ado, let's create a new object called **obj_card**. This will describe the behavior and characteristics of every card in the deck.

In the **obj_card** object, make a new **Create** event. This will be the place in which we are going to define the card's properties. To recap, we want our card to have a variable that determines if the card is showing the front or back. We also want a variable that tells us which kind of card we are dealing with and its place in the deck.

Ideally, to indicate the type of the card, we want to use a number (from 0 to 7, since we have eight different sprites for our cards). This number will be stored in a variable called **type** and will be later used to access the right sprite in the array **cardtype** that contains all the cards' sprites.

We can safely initialize the value of **type** to 0, to refer to the first card (so that we can already start testing what we are doing).

```
1   // Initializing the main properties for a card object
2   index = noone; // the position of the card in the deck
3   type = 0; // associates the card with its sprite (0-7)
4   face = 1; // the face the card is showing (0 = back,
    1 = front)
```

In the preceding code, we are initializing the card properties (lines 2–4). The index of the card is the position of the card in the deck, the type of the card is used to associate a card with a sprite, and the face variable tells us if the card is showing the back or the front.

Now we need to write the code to tell the card to flip when the player clicks on it with the mouse. This is very simple: we only have to tell the game that whenever the player clicks with the mouse on the card, we should change the variable **face** from 1 (front) to 0 (back) or vice versa.

To be able to do that, we need another piece of programming knowledge: we have to understand how to decide things if GML.

In programming, you can check for certain conditions and act accordingly using conditional statements like if-then-else and switch.

if-then-else

if-then-else is a conditional statement that allows you to check for a condition and execute a block of code if that condition is verified. You can also execute a specific block of code in case that condition is not satisfied.

if-then-else logic is very simple: if this condition is true, then do this; or else do that.

For example:

```
if a == b
{
    // do this
}
```

```
else
{
    // do that
}
```

You can specify multiple options by concatenating if and else like this:

```
if a == b
{
    // do this
}
else if a < b
{
    // do that
}
else
{
    // do other things
}
```

switch

switch works similarly to if-then-else, but it's specialized in checking which value a variable is assuming parsing a list of possible values.

For example:

```
switch ( a )
{
    case 0:
        // code for the case that a == 0
        break;
    case 1:
        // code for the case that a == 1
        break;
    default:
        // code for all the other cases where a is neither
        equal to 0 or 1
        break;
}
```

The keyword case is used to ask GameMaker to check if the variable we are testing is equal to a specific value. The code to execute if that condition is true is included between case and break.

The default case holds the code to execute when none of the previous cases occurred.

Let's immediately use those new tools! Create a new event in the **obj_card** object **Mouse ➤ Left Pressed** and write the following code inside of it:

```
if (face == 0)
{
    face = 1;
}
else
{
    face = 0;
}
```

The preceding code states that every time we left-click on the **obj_card** object, if the card is showing its back (**face** is equal to 0), we flip it changing the value of **face** from 0 to 1 and vice versa.

Caution The *double equal sign* (==) is not to be confused with the *single equal sign* (=)!

The former is used to compare two expressions; the latter is used to assign the rightmost value to the leftmost variable.

Note You can compare numbers using comparison operators, which are <, <=, ==, !=, >, and >=. They are binary operators that allow you to compare two expressions or variables returning a Boolean value (true or false) as the result of the comparison.

a == b returns true if a and b are equal; otherwise, it returns false.

```
a != b returns true if a and b are different;
otherwise, it returns false.

a > b returns true if a is greater than b and
false otherwise.

a >= b returns true if a is greater than or equal
to b and false otherwise.

a < b returns true if a is less than b and false
otherwise.

a <= b returns true if a is less than or equal to b
and false otherwise.
```

Ok, now we have the functionality to flip the cards, but we won't see anything yet, because we don't have a way to update the visual style of the card when we click. To do this, we need an event that can constantly check if the card is showing the front or the back.

Introducing the **Step** event. **Step** is an event that is recurring once per frame (so it depends on how many frame per second your game runs). We will use this event to constantly check which face the card is showing, so that the game can draw the right sprite on screen. So let's create a new **Step** event for our **obj_card** and put this code inside:

```
/// @description Show the correct sprite
if ( face == 0 )
{
    sprite_index = spr_cardback;
}
else
{
    switch(type)
    {
        case 0:
            sprite_index = spr_fire;
            break;
        case 1:
            sprite_index = spr_mountain;
            break;
        case 2:
            sprite_index = spr_rain;
            break;
        case 3:
            sprite_index = spr_sun;
            break;
        case 4:
            sprite_index = spr_river;
            break;
        case 5:
            sprite_index = spr_moon;
            break;
        case 6:
            sprite_index = spr_morning;
            break;
        case 7:
```

```
32                  sprite_index = spr_afternoon;
33                  break;
34              default:
35                  break;
36          }
37      }
```

Lines 1–4: We check if **face** is equal to 0. If it is, it means that we want to show the back of the card. So we assign the back sprite (spr_cardback) to the property **sprite_index**.

Lines 5–8: When the value of face is 1, we check the type of the card and assign the right sprite to the object. Each sprite represents a type of card.

Caution **sprite_index** is a reserved word. It is a property that any GameMaker object has and represents the sprite that the object is showing. If you change its value, you change the sprite the object shows. Be careful, though, because **sprite_index** only accepts sprites as values or -1 to indicate that there is no sprite.

Now we have everything we need to flip cards. We need to test if we did everything alright! So let's open the main room and Drag and Drop the **obj_card** in the middle of it. Since we are here, let's also change the size of the window. You can do that by changing **Width** and **Height** in the room's **Property** tab. Let's make it 600 × 800.

Now run the game, and you will see your card standing there, showing its back. If you click on it, the card will flip, showing the front (Figure 3-4); if you click a second time, the card will flip back.

Good job!

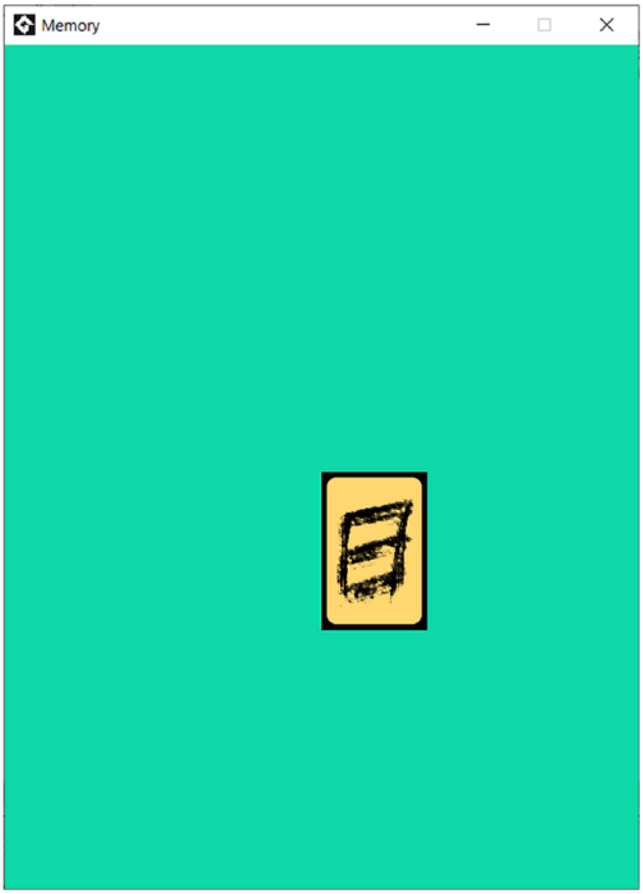


Figure 3-4. *Our first flipping card!*

Now that we have our card working and flipping, we can start thinking about the deck and the shuffling feature.

Deck

Now that we successfully coded the cards mechanics, we need to take care of how to create a deck of cards for our game. To do that, we have to ask ourselves a question: *What is a deck of cards?*

A deck of cards is a collection of cards that can be ordered, shuffled, and accessed (e.g., searching for a single card inside the deck, picking the top card, etc.). We have already seen one type of collection of items that can be used to create a deck of cards in the previous section: arrays. Anyway, it’s convenient to talk about some other possibilities that we can choose from, to better design our deck of cards.

In computer science, there are some concepts that are used to represent different types of collection of items: they are called **data structures**.

Each data structure can be sorted and accessed to add a new item or search for an existing item to move it or delete it. Depending on what we need to do, we can use a specific data structure to get the job done, since some data structures are better at some tasks than others.

GameMaker Studio 2 supports some commonly used data structures. The good thing is that the choice you have is wide enough to let you make whatever you need. The bad thing is that GML is not flexible enough to let you create new data structures easily. Anyway, you will hardly need to.

GameMaker Studio 2 offers six different types of data structures:

- Stack
- Queue
- List
- Map
- Priority Queue
- Grid

In the next section, we are going to briefly describe each one of these options so that we can choose the right one for our deck design. If you are already aware of data structures and you are only interested in our final choice, feel free to skip the next section.

Fantastic data structures and where to find them

In this section, I will give you an overview on the most used data structures in Computer Science and provide you with the corresponding concept in GML.

Data structures are fundamental concepts that are at the base of problem- solving and software engineering. Bear with me, while we explore one of the most powerful tools of Computer Science!

Stack

Just like a pile of dishes, a stack is organized as a pile of elements in which the element on the bottom is the oldest, while the one on the top is the most recent. Going ahead with the pile of dishes analogy, stacks offer two main actions: push and pop (Figure 3-5).

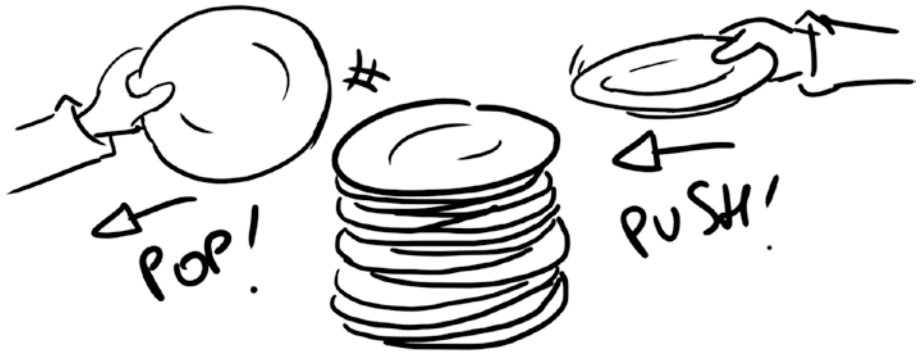


Figure 3-5. *A stack is just like a pile of dishes*

If you push an item in a stack, you will place it on the top of it, so that it will become the new first item of the structure, while the pop action deletes the topmost element of the stack.

As you can easily guess, stacks are very good for lists of elements that need to be accessed always from the most recent to the oldest. Indeed, stacks' performance is optimal with push and pop actions and suboptimal if you need to search an item in a specific position. For example, if you need to access the fifth item in the stack, you have to start from the topmost (the first item) and go down through the items one by one until you reach the fifth item.

Better not to use a stack to represent our deck of cards. Just try to sort a pile of dishes in chromatic order… not so easy, right?

In GML, stacks are called **DS Stacks**.

Queue

A queue is very similar to a stack, but it differs in some crucial details. In fact, queues follow a first-in-first-out (FIFO) policy. That means that the oldest element of the queue is always the first to be accessed.

Just think about a queue at a post office (Figure 3-6). The first people to be served are always the ones who are waiting the longest, and every new person who needs to access the post office goes on the end of the queue and waits for their turn.

This is exactly how a queue in coding works. Each new element added to the queue is put in the end of the queue, and the elements of the queue are consumed from the oldest to the newest.

Let's think, for instance, that we need to instruct a robot on how to get from point A to point B in a maze. We will say things like "Go forward for 1 meter," "Turn 90 degrees," "Go forward for half a meter," and so on. We want the robot to follow precisely the instructions we are giving, in exactly that order: from the oldest to the newest. We don't want him to follow the instructions randomly or in reverse order. This is a perfect situation in which we can use a queue! We just add each instruction in a queue, then we pass the queue to the robot, and it will follow the instructions in the right order.

Queues are very good to manage cases like that! They're very efficient in queueing and dequeueing elements in a FIFO fashion, but just like stacks, they're not good for random access.

We can't use queues for our deck of cards, since, just like a stack, they are not made to be easily shuffled, so that would be a bad design choice.

In GML, queues are called **DS Queues**.

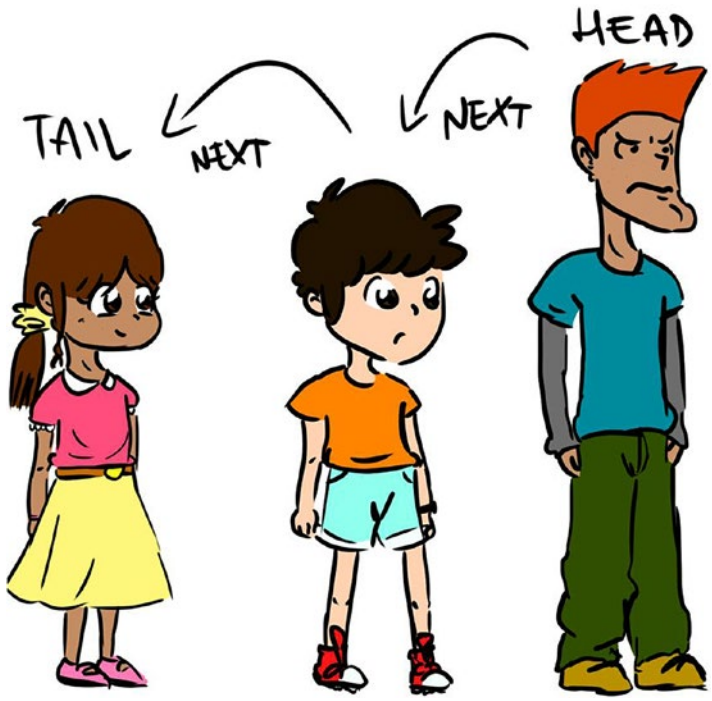


Figure 3-6. *A queue is like a bunch of people standing in a line. The head of the line is the first item of the queue, while the last one is called tail.*

List

A list is a data structure that organizes elements sequentially. In a list, each element is associated to an integer value called index, so that you can immediately access an element in any position of the list.

A list can be sorted in ascending or descending order or shuffled (randomized). They are very flexible and can be modified in length with ease (unlike arrays).

Lists are very good to represent collections of items that can vary in size, that need to be sorted, and that require quick random access (accessing elements in any position). This is exactly what we need for our deck of cards! We found our data structure!

In GameMaker: Studio 2, lists are called **DS Lists**.

Map

A map is a data structure that stores key and value pairs (Figure 3-7). Both key and value can be of any type. You can quickly insert a new pair in a map or pick an existing value if you know the associated key. Beware, though, that maps are not sorted; so if you don't know the key associated to a specific value, you will have to iterate through all the existing pairs, before you find the right one; and that's very slow! Be aware that you can't assign more than one value to a key and that keys are unique (you cannot have doubles).

Maps are very useful in all those situations in which you need to assign a value to a concept. For instance, let's think about an RPG: your character will likely have an inventory containing all their items. They may have two health potions, an apple, and three keys. You can easily represent the inventory with a map associating to each item, the value representing its quantity:

- Health Potion, 2
- Apple, 1
- Keys, 3

That's it! Whenever your character uses an item, you just have to access it searching for the right key (e.g., "Keys") and modify its quantity, if needed.

Even if we could represent a deck of cards with a map, it's very inconvenient, since as we just said, it's an unsorted data structure and being unsorted also means to be non-randomizable (so we won't be able to shuffle a deck made with a map data structure).

In GameMaker Studio 2, maps are called **DS Maps**.

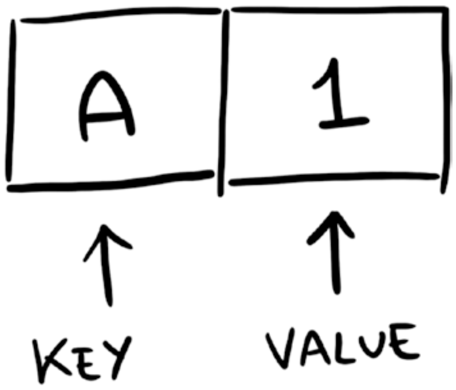


Figure 3-7. *In a map, each element is a couple made of a key and a value*

Priority Queue

A priority queue is very similar to a queue, with the only difference that it's ordered by a priority value.

Let's make an example to better understand the principle. Think, for instance, about the queue in an emergency room. People don't just queue in the order they arrive. There is a priority value that is the actual gravity of the injury. The more the injury is serious, the more the patient is high in the queue.

A priority queue has a numeric (real) value – called weight – that expresses the order in the queue for every element.

Priority Queues are very useful in situations in which you need a data structure that should be sorted all the time. For example, if you need to make a leaderboard, you don't want to sort it every time you add an item to the queue; it would be very time consuming! In this and many other cases, a priority queue is just what you need to get the job done!

Another interesting and common usage for priority queue is the management of processes to execute in an operating system. In fact, some operating systems use process scheduling algorithms based on priority queues. They just give ratings to tasks based on how critical they are for the system and the user experience and then put the tasks into the queue to be executed.

For our case of building a deck of card, we don't need a priority queue, since we do want the deck to be randomly sorted and it wouldn't be possible using a data structure that remains ordered all the time based on a specific rating.

In GameMaker Studio 2, priority queues are called **DS Priority Queues**.

Grids

Grids are basically two-dimensional arrays. Just think about a table: you define the number of rows and columns, and then you can access items with *x,y* pairs.

A very common application for grids is representing maps and playgrounds. For instance, think about *Battleship* (or *Sea Battle*): you have a map in which you can place your ships, and to identify a single spot on the map, you call a number that represents the row and a letter that represents the column. Grids are based on the same principle. You have a grid, and you can access every item by using the number of the row and the number of the column.

We can totally represent a deck using a grid, but it is effective only if you're designing a game with a deck of cards made of more than one card suit. Since we don't have card suits, it would be overkill. Also, we need the deck to be ordered randomly, and it would be difficult using a grid (it could increase time complexity). So we are not using grids.

In GameMaker Studio 2, grids are called **DS Grids**.

Designing decks

All in all, our best option to design a deck of cards for our Memory card game is a **DS List**. In fact, as we already saw, DS Lists are very flexible: they can be used to represent data that need to be sorted (or randomized), and they also feature good random access time performances. That's exactly what we need!

A DS List is a bit different to use than an array. To create a DS List, you need to call a function. We will explain later what a function is. Right now, let's just use it. You can create a list and associate it to a variable name like this:

```
var mylist = ds_list_create();
```

To add an item to a DS List

```
ds_list_add(mylist, 10);
```

The preceding code creates a new element in the list **mylist** with the value 10. The new element is positioned in the last position.

To access the value of an item at position **i,** you can either do this

```
var item_i = ds_list_find_value(mylist, i);
```

or use this equivalent notation:

```
var item_i = mylist[| i];
```

There are other interesting and useful functions that you can use with your DS Lists. Feel free to check them out in the official GML documentation.

Since the deck is only a collection of cards represented by a DS List, we don't need to create an object for it. So we will include the code for the deck into another object: the **game controller**.

Indeed, the game controller is the object that rules upon our game, checking that the game is working properly and managing every object involved.

Because of its role, the game controller is perfect to take care of the creation and management of our deck.

Note To merge and combine two concepts is a common thing in software design. As designers, we have to understand when it's convenient to keep two concepts separated or when it's better to merge them into one. In this case, since we don't need a deck object, it's better to simplify our project avoiding it and delegating the creation and management of the deck of cards to the game controller.

Create a new object called **obj_controller**. This will be the object that will rule upon our game checking that everything is working properly.

First of all, we need a *Create* event in which we will create the deck and fill it with 16 cards, each of which should be initialized properly. Even if we can do this one card at a time, it would be very inefficient and dull to do it that way, so we are going to introduce a new programming concept: **loops**.

Code loops

A **loop** is a flow control statement that allows a block of code to be repeated until a certain condition is met.

There are four loop control statements in GML:

- Repeat
- While
- Do
- For

Let's see how and when to use them.

Repeat

Repeat has the form

```
repeat(<expression>)
{
      <statement>
}
```

Repeat will execute <statement> a number of times equal to the result of <expression>.

Repeat is very useful when you have to execute some actions a predetermined number of times.

While

While has the form

```
While(<expression>)
{
      <statement>
}
```

While doesn't just repeat <statement> a number of times expressed by <expression>. In fact, here, <expression> is not just a number, but a Boolean expression. This means that **while** will constantly evaluate <expression> and if its logic value is **true**, the code inside the curly brackets (<statement> in this case) will be executed.

While is very useful if you have to repeat some action as long as a certain condition is met.

Do-until

Do has the form

```
do
{
      <statement>
}
until(<expression>)
```

Do is a bit different from the other loops. It executes the code inside the curly brackets at least once and then starts to evaluate <expression> for all the next iterations. **Do** executes <statement> until <expression> is **true**.

Do is very useful if you need to execute the actions inside your loop at least once.

For

For has the form

```
for(<statement1>; <expression>; <statement2>)
{
      <statement3>
}
```

For executes <statement3> as long as <expression> is **true**, just like **while**. The difference is that **for** allows you to execute some actions at the beginning of the loop and after each iteration. In fact, <statement1> is executed when the loop starts the first time, while <statement2> is executed after every iteration of the loop.

For is used to count up (or down) to a certain value. Let's say that you want to calculate the factorial of 10. You need a counter variable that starts counting from 1 to 10 multiplying all the numbers by one another.

You can do something like that using **for**:

```
1   var result = 1;
2
3   for(var i = 1; i <= 10; i += 1)
4   {
5       result *= i;
6   }
```

At line 3, inside the brackets, we are declaring the **i** variable and assigning a starting value of 1 (i = 1).

For every iteration of the **for** loop, we are checking if **i** is lesser than 10 (i < 10). If it is, we multiply **result** by the value of **i** (result *= i). After this, we increase the value of **i** by 1 (i += 1) so that we can pass to the next number.

For is very useful when you have to loop a piece of code until a condition is met, and that condition is being ruled by the value of a counter variable.

Caution Loops are a very powerful programming concept that can be the cause of many bugs in your code, if not used properly.

Be sure that the conditions of your loops can be met, or you will experience endless loops that will continue to run until your game crashes or your memory runs out.

Making decks

Let's get back to our deck! We now know how to repeat a set of instructions for a number of times or until a condition is met. We can use this concept to automate our deck creation and the setup of our cards.

Let's make a **Create** event for our new **obj_controller** and put this code into it:

```
1   cards_number = 8;
2   deck = ds_list_create();
3   var deck_size = cards_number * 2;
4
5   for(var i = 0; i < deck_size; i+=1)
6   {
7       ds_list_add(deck, instance_create_layer(0, 0,
        "Instances", obj_card));
8   }
9
10  // assign card types to card objects and set up cards
11  for(var i = 0; i < deck_size; i+=1)
12  {
13      deck[| i].type = i % cards_number;
14      deck[| i].face = 0;
15      deck[| i].index = i;
16      deck[| i].visible = false;
17  }
18
19  // shuffle cards
```

Lines 1–3: We define variables to keep track of the number of available cards, the deck's size, and the number of different cards that we have (the deck will be of size *cards_number* ∗ 2).

Lines 5–8: We fill the DS List with 16 cards (as we already said, *deck_size* = cards_number ∗ *2* = *16*).

Lines 11–17: We assign a *type, face,* and *index* to every card in the deck. We need eight couples of cards. Each couple is made of two copies of the same card.

To access cards properties like type, face, and index, we are using the **dot notation**. Dot notation allows us to access variables declared inside objects.

We are also turning off the visible property for our cards, so that the game will not show them (the sprite won't be drawn on the screen).

The visible property is present in every object in GameMaker Studio 2 and tells the game if an object should be drawn or not to the screen. We are turning this off because we want to show the cards only after we shuffled them. We will turn this back on right after the shuffling and dealing of our deck of cards. Speaking of shuffling… how can we do it?

Note To create a new object in GML, you can use the instance_create_layer function that allows you to create a new object into a specific layer of the room.

For example:

```
instance_create_layer(0, 0, "Instances", obj_card);
```

This will create a new instance of obj_card in the Instances layer (the default one) at position 0,0 in the current room.

Caution The **dot notation** is very useful to access object's variables from external scopes, but you can't access local variables declared with the keyword **var** using the dot notation. Variables declared with the **var** keyword are bond to the scope in which they are declared.

Every day I'm shuffling

Shuffling is a very important feature in our game, and we need to repeat this action more than once. More precisely, we need to shuffle cards when the player

- Starts the game
- Presses the **enter** key
- Restarts the game (e.g., after a game over)

In software design, to repeat code is a very bad practice. In fact, code duplication increases the possibility to experience bugs. Just think about the shuffling algorithm we want to write: repeating the same code three times will force us to modify three pieces of code every time we want to change something in that algorithm. This means that we could forget to modify something and our code will become inconsistent and buggy.

Anyway, we do need to execute this code three times! So how can we do that? Here come functions, saving the day!

How do functions function?

Functions are blocks of code associated to a label. You can call a function from anywhere in your code.

When you call a function, you can pass it some parameters that can change its behavior.

A function can return a value – called output – that often represents the result of an evaluation or the outcome of the function (e.g., if everything's gone well or not). In nearly every programming language, functions are called like this:

```
my_output = my_function(my_input)
```

Functions are directly taken from the concept of mathematical function. In fact, a mathematical function *f* is an algorithm that can be applied to a (or more than one) variable *x* and returns a result *y*. The following notation represents this concept, in math:

$$y = f(x)$$

Mathematical functions are very important and are widely used for a plethora of evaluations. Just think about Trigonometry. In trigonometry, you use sin and cos functions to get the value associated (in the range [0,1]) to a given angle expressed in degrees or radiants. And you do it this way:

$$y = sin(x)$$

That means *y* is the value associated to the sin function applied to the angle *x*.

You already used some functions in GML, like **array_length_1d(cardtypes)** to get the size of the cardtype array or **ds_list_create()** to create a new DS List.

The former required you to pass it the name of the array of which you wanted to know the size; the latter didn't require any other information, because all that it did was to allocate some space in memory to create your DS List.

In both cases, you got a result: **array_length_1d** returned a number (the length of the array), while **ds_list_create** returned a DS List that you could associate to a label (in our case **deck**).

GML functions

To create a function, in GML, you need to create a new script object. In fact, in GML, functions are literally pieces of code stored in your assets list that you can call just like you do with any other object listed in the **Resources** tab.

So let's create a new script by right-clicking **Scripts** in the **Resources** tab. Let's call it **shuffle_cards**.

You will be presented a blank text box. Here is where you can write the code of your function.

To shuffle a deck of cards, we need to get access to the deck, so this is the parameter that we want to receive when the **shuffle_cards** function is called. This means that to call this function, we will do something like this:

```
shuffle_cards(my_deck);
```

To retrieve the deck inside the function code, we have to use a reserved keyword: **argument**. Every parameter passed to the function is an argument numbered starting from 0. So my_deck is the argument 0, and we can access it like this:

```
var gamedeck = argument0;
```

Pretty easy, isn't it? Now gamedeck is associated with my_deck, and we can use it inside our code.

Now we need to randomize our deck of cards. To shuffle a DS List, we can make use of a GML function: **ds_list_shuffle**. Remember that, as we saw in the previous chapter, to randomize the RNG (Random Number Generator), we have to call the **randomize** function. So let's add those two lines to our function:

```
randomize();
ds_list_shuffle(deck);
```

Shuffle cards code

Now that we faced every concept, we needed to shuffle our cards, we can rewrite our **shuffle_cards** function, so that we can include the ability to visualize our cards.

```
var gamedeck = argument0;

var cards_x = 130;
var cards_y = 160;

randomize();
ds_list_shuffle(deck);

var deck_size = ds_list_size(gamedeck);

// position cards on the table
var cards = 1;
for(var i = 0; i < deck_size; i += 1)
{
    deck[| i].x = cards_x;
    deck[| i].y = cards_y;
    deck[| i].index = i;
    deck[| i].visible = true;

    if(cards % 4 == 0)
    {
       cards_x = 130;
       cards_y += 160;
    }
    else
    {
       cards_x += 110;
    }
    cards += 1;
}
```

Woah! That's some long piece of code! Let's break it and analyze it!

Lines 1–7: As we did before, we're importing the deck and shuffling it. The only new thing is that we now have two variables that will help us in positioning the cards in the room:

- **cards_x**: The horizontal starting position of the cards (the x value of the first card that we will place in the room)
- **cards_y**: The vertical starting position of the cards (the y value of the first card that we will place in the room)

Line 12: Here we are declaring a new variable called **cards**. This variable will keep track of how many cards we have placed in the room.

Line 13: We start a for loop that will repeat for every card, counting from 0 to 16 (**deck_size** value).

Lines 15–18: We set the coordinates of the i-th card and its index value, and we make it visible, so that it can be drawn in the room.

Lines 20–28: We want to place our cards in a 4 × 4 grid (since they are 16), so every time we place a card, we do a right shift on the X-axis to place the next one. When we place four cards in a row, we reset our X-coordinate to the starting value and increase the Y-coordinate so that we can start a new row.

To properly count how many cards we are placing in a row, we are using a mathematical trick called modulo function. Modulo function, in GML, is represented by the percent operator (%) and tells you what the remainder of the division between two numbers is. If the remainder of a division between A and B is 0, it means that you can divide an A number of elements in a number of groups of exactly B elements. So we can know if we have placed four cards in a row with the following line of code:

```
(cards % 4) == 0
```

That way we can decide to start over from a new row. In fact, cards % 4 equals 0 only when **cards** is equal to a multiple of 4 (4, 8, 12, 16, etc.). This means that line 20 will be true only in that case.

Line 29: At the end of the loop, we increase the value of the **cards** variable to keep track of the number of cards we placed.

This is all we need to shuffle our cards and place them in the right position.

As we stated before, we want to be able to shuffle cards every time we press the enter key, so that we can reset the game. Let's do this right now!

Create a new **Key Press ➤ Enter** event and add this single line in it:

```
shuffle_cards(deck);
```

Now, every time we press the enter key, the deck is going to be shuffled!

We can now test what we did so far! Let's open the main room of our game and get rid of the **obj_card** we put in there last time. Now Drag and Drop an instance of **obj_controller** in the room. Save and run (Figure 3-8)!

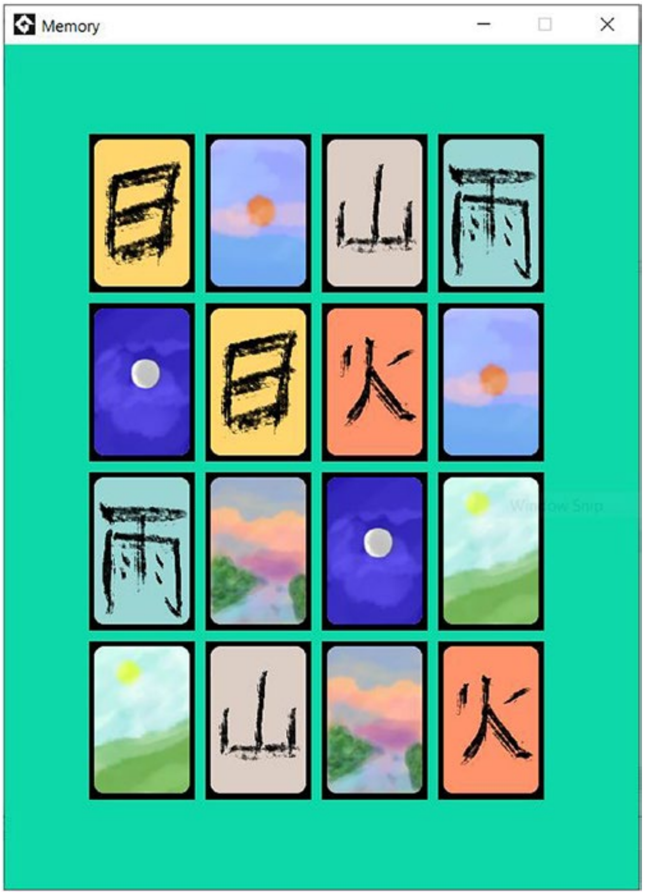


Figure 3-8. *The first version of the game, allows for card shuffling and flipping*

Running the game, you will see a grid of cards showing their back. If you click them, they will flip. Cards are shuffled every time you run the game, and you can also shuffle them by pressing the enter key (Figure 3-8).

That's great! We are half the way to complete our Memory card game!

In the next chapter, we will complete our Memory card game, and we will see how to make it enjoyable by adding some new and fun features like the following:

- The constraint to flip no more than two cards per turn
- Some checks to see if the cards flipped are matching
- Victory checks to actually win the game
- A timer (when the time runs out, we lose the game)

TEST YOUR KNOWLEDGE!

1. What is a Game Design Document (GDD)?
2. What are the most important characteristics of a GDD?
3. Why is it important to prepare a design of your game before you start coding?
4. What is an array? Why is it useful?
5. How can you access an element of an array?
6. What is a conditional statement?
7. Which conditional statements can we use in GML? How do they work?
8. What is a data structure?
9. What data structures does GameMaker Studio 2 offer?
10. Can you tell the difference between a stack and a queue?
11. What is a function? How can you make one in GameMaker Studio 2?
12. What is a loop?
13. What is the difference between a while loop and a do-until loop?
14. Why do we use lists to represent a deck of cards for our game?

CHAPTER 4

Card Game (Part 2)

The main feature of a game is to be playable, and our card game is not, really – unless you do enjoy flipping things endlessly for no reason.

So let's add some meat to our Memory game!

In this chapter, we will define the game rules we designed in Memory GDD (in the previous chapter) to make it playable and enjoyable. We will implement the constraint to flip at most two cards per turn, and then we will add the functionality to check whether the cards are matching or not; we will also define some victory conditions and create a countdown to add some time-based gameplay that will make our game more challenging and fun.

So get ready! We are going to learn a ton of new concepts and make our very first video game with GameMaker Studio 2!

Finite-state Machines (FSMs)

A video game, just like any other software, is an interactive application. It takes inputs from the player and returns outputs generated from an elaboration. To get from inputs to outputs, the game (or the application) passes through a series of states.

Let's make an example: we want to buy a drink from a vending machine. We see from the machine's display that it's ready to take our money, so we insert cash inside it. After receiving our money, the machine's display tells us exactly how much money we put inside, and we know that now we can choose what we want to buy. If we select a more

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_4

expensive drink, the machine just says that the drink is too expensive and asks you to add the right amount of money to get that drink; if we choose a drink that's right for the amount of money we inserted, the machine releases the drink – so that we can take it – and it stores the money. The displayed import will be now the money we inserted minus the price of the drink.During the process of buying a drink, the vending machine passes through a number of states to get from input (money inserted) to the output (drink received).

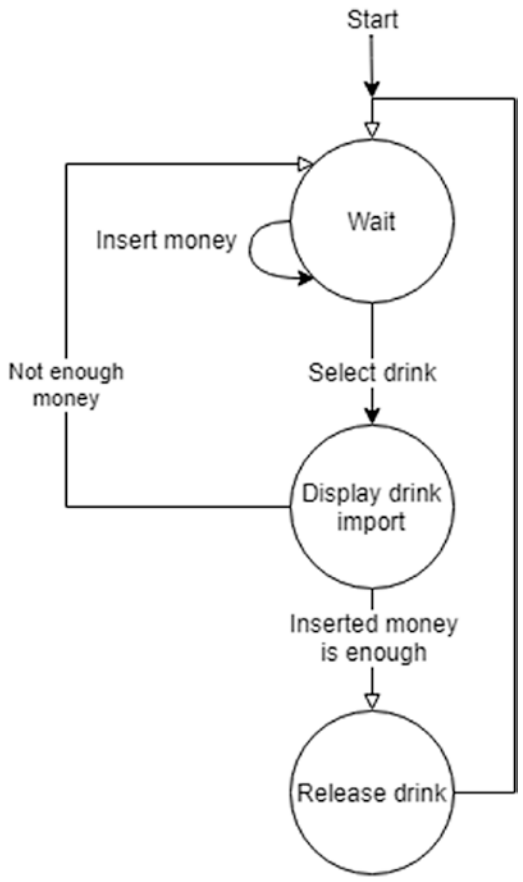


Figure 4-1. *FSM scheme of a vending machine*

Figure 4-1 represents the different states of the process of buying a drink from a vending machine. Black-headed arrows are user inputs, while white-headed arrows are consequences to the machine's elaborations.

When we approach the machine, it is in a waiting state (`Wait`), and it's ready to get an input. We can interact with the machine only in two ways (if you don't count kicking it when it doesn't work as expected): inserting money and selecting a drink.

If we insert money, the machine displays the import and remains in the `Wait` state.

If we select a drink, there are two possible cases:

- If we inserted enough money, the machine passes in the `Release drink` state giving us the drink we paid for (and possibly the change), and then it returns in the `Wait` state to start over again.
- If we didn't insert enough money, the machine just displays the amount of money needed and goes back to the Wait state expecting you to insert the right amount of money.

As you can see, to schematize a process in that way can be really useful to have a clear idea of what the process is all about. You can get rid of all the technical details in the process (like counting money, recognizing coins, etc.) and concentrate on the important bits of the flow, like the interactions and the responses of the application (inputs and outputs).

That way of organizing flows is called **Finite-state Machine** (or just State Machine).

A Finite-state Machine (FSM) is an abstract machine that can be in just one state at any given time. The state of that machine can change according to inputs; that change is called *transition*.

FSM is a huge topic that we don't need to explore in its entirety here. There are very interesting university courses about it that can give you a deep understanding of the power of this mathematical modeling tool widely used in computer science. Don't worry, I will give you all the information needed to use this tool for the purpose of this book.

A FSM has an entry point, a finite number of states, and a *transition function* that allows us to pass from a state to another.

But enough with that academic gibberish! How can a FSM help us with our game?

FSM can be really useful in design and development of video games because it can effectively represent the flow of a game. In fact, games constantly pass from a state to the other during their execution, and to be able to clearly represent this flow is crucial to good design and optimized implementation of any kind of software.

Just think about the flow that we want to implement in our card game.

We want the game to wait for the player input in a wait state; and then, when the player flips two cards, we want it to pass to a check state and check whether the cards are matching or not. After that, we want the game to check if there are any cards left on the table: if that's the case, the game goes back to the wait state; if there are no more cards, the victory condition is checked, and the game passes to a win or lose state.

Organizing our game flow like this, we can control our gameplay and our code more clearly, dividing it in separate moments.

So we know that our game will have four states:

- **Wait**: We are playing, and the game is waiting for the player's inputs.
- **Check**: No inputs permitted. The player must wait until the end of the elaboration.
- **Victory**: No more cards on the table! The player won the game!
- **Loss**: Time's up, and there are still cards on the table. The player lost the game.

Now that we decided the states we need for the game, we can draw our FSM scheme (Figure 4-2).

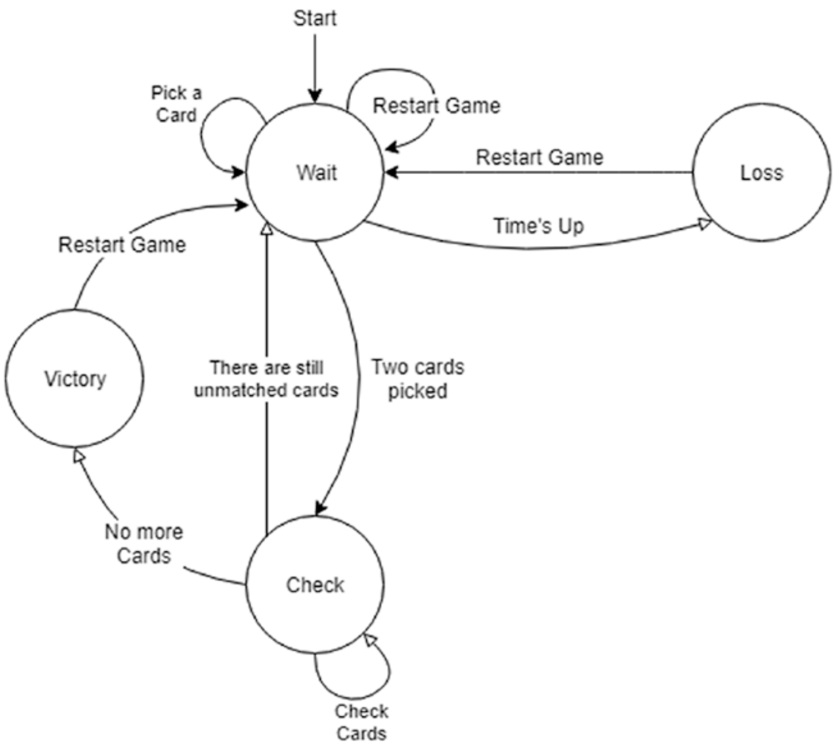


Figure 4-2. *FSM scheme for our Memory card game*

The FSM scheme makes clear the flow of the game and gives us a clear division between the separated moments of our gameplay and also gives us an idea of what we need to develop. Let's make a list of all the features we are missing for each state:

- **Wait**
  - Flip cards: We already have it!
  - Pick cards: We need a system to store in memory the cards we decide to flip.

    - Constraint of two cards: We need to check that the player cannot pick/flip more than two cards.
    - Restart game: We give the player the possibility to restart the game anytime.
- **Check**
    - Confront cards: We need the game to check if the flipped cards match and then act consequently (take them out or flip them back).
    - Check victory: We want the game to check if we won the game (no more cards to pick) or not and act consequently.
- **Victory**
    - Victory message: We just show a victory message.
    - Restart game: We give the player the possibility to restart the game and play again.
- **Loss**
    - Game over message: We just show a message to tell the player he just lost the game.
    - Restart game: We give the player the possibility to restart the game and play again.

Structuring the tasks like that is very useful to keep the development clear and easy to follow.

This is something I want you to do anytime you have to code: spend time designing! The more time you spend thinking about what you're going to do and organizing it, the less time you spend coding and fixing bugs. Having a well-designed project and a clear set of features to implement, it's priceless and makes the coding way easier and faster.

Now that we have our nice plan to introduce new features in the game, the only thing remaining is to start coding!

Tip Spending time designing a game covering its flow and features before diving into the code is very convenient and allows you to create a project that is not just a good game, but also a good software. To make a good software is not just a style exercise, but it's very important to make the game easy to modify and update from anyone. Remember that if you decide to become a game developer, you will likely have to work with other people. That's why it's important to have a well-designed game and tidy code.

You may want to use FSM every time you can. It is a powerful tool to analyze the flow of a software, and it can be very useful to optimize it. Writing down a FSM can highlight when a game/software flow is too contorted and can be shortened, for example. It's a very good practice that you should learn to use for the best. FMS is also very useful to prove mathematically that your code works, no matter what.

From State machine to code

We will code our game following the design we just created with our FSM.

First things first, we have to decide how to implement the concept of game state. The game state can be seen as a checkpoint, an information that tells you where you are at a certain moment in the execution of the game. This means that a game state is no more than a label we associate to a certain moment of the application. A common way to create those labels is by declaring a so-called enumerator. An enumerator is a data type that allows you to create a list of objects with unique values associated. Using an enumerator, you can create labels with a human-readable name. In this case, we can use an enumerator to label game states.

You can declare an enumerator in GML like this:

```
enum myenum {
    item1,
    item2,
    item3
};
```

The game state will change according to what the player does, and this will direct the game flow and make things happen in the right moment:

- Enumerators are implicitly declared as global scope variables, so we can access them from anywhere in the game.
- Other than the enumerator itself, we have to declare a global variable that will represent the current state of the game.
- Go ahead and declare the game state enumerator and the current state global variable in `obj_controller`'s `Create` event:

  ```
  enum states {
      paused,
      playing,
      won,
      lost
  };
  global.game_state = states.paused;
  ```

The next problem we have to solve is how to pick and remember the cards that the player clicks. We know how to flip them, but nothing more than that.

To pick a card and remember it means that we should have some variable or data structure to store the information related to our cards. To do that, we can use a simple array of two elements. We will store the card objects into that array, and because we need to access it from anywhere in the game, we want it to be global.

Open up obj_controller's Create event and add this line just under the declaration of the game_state global variable:

```
global.cards = [ noone, noone ];
```

In the preceding code, we initialize the two elements of the cards array to *noone*. *Noone* is a special value that represents the lack of a value. We will use that value to represent the fact that the player has not selected any card.

When the player clicks a card, we want the game to flip that card and check if the player already selected a card before; and if they have, we want the game to check if the two selected cards match. If they do, we put them away; if they don't, we flip them back.

We can use the global array *cards* to store the cards as soon as the player selects them. When we see that the player is picking a second card, we can run the checks on the cards to see if it's a pair or not.

To know if the player picked one or two cards, we can rely on the order in which we access the array global.cards. In fact, if the code is consistent with the data access order and always fills the array from the first to the last element of the array, we can be sure that if the first element is empty, also the second is and if the first is not empty, the second is. As soon as both the elements of global.cards are filled with data, we run the checks, and then we empty the array in its entirety.

We just need to check one thing: is the first element of global.cards empty? If it is, this is the first card the player selects, and we save it in global.cards[0]; if it's not, we put the card's data in global.cards[1], we run the checks, and we empty global.cards.

Now that we designed it, let's code it! Open up obj_card's Left Pressed event and replace the existing code with the following:

```
if ( visible and global.game_state == states.wait )
{
    face = 1;

    if ( global.cards[0] == noone ) // if no cards are picked
    {
        global.cards[0] = self;
    }
    else
    {
        global.cards[1] = self;

        if ( global.cards[0].index == global.cards[1].index )
        {
            face = 0;
            global.cards[0] = noone;
            global.cards[1] = noone;
        }
        else
        {
            if ( global.cards[0].type == global.cards[1].type )
            {
                global.cards[0].visible = false;
                global.cards[1].visible = false;
            }
        }
27          global.cards[0].face = 0;
28          global.cards[1].face = 0;
29          global.cards[0] = noone;
30          global.cards[1] = noone;
31      }
32  }
```

Let's analyze this code!

Line 1: Since we are writing the functionality to pick cards and we know that we can allow the player to pick cards only in the Wait state, here we check that we actually are in such a state. We also check if the card is visible because even if not visible, the card would be clickable anyway. In fact, the only effect that the visible effect has on a property is to avoid its own draw event to be executed.

Line 3: We are just flipping the card, as we did before. Easy!

Lines 5–8: We are going to use the two-element array (global.cards) in a precise order. We will always fill the first element first (element 0) and then the second element (element 1). Following that order, we know in any moment that, if the first element is empty, also the second is empty. That's important so that we can easily tell if the one picked by the player is the first or second card.

So, in those lines, we check if the first element of global.cards is empty. If it is, that means that this is the first card the player is picking and we need to save it inside global.card's first element.

Line 9: If global.cards[0] is not empty, that means that the player is picking the second card, so here we cover the case in which we save the second card's information and we confront the two cards picked.

Line 11: In line 9, we make sure that this is the second card that the player is picking, so we are going to save this into the second element of global.cards (global.cards[1]).

Lines 14–19: Here we check if the player has clicked twice on the same card. If they did, we just flip back the card and empty our global.cards array.

We do the check by confronting the indexes of the two cards, since we know that the index is a unique number associated to each card in the deck.

Line 20: If the two cards picked are not the same cards, we have to check if they match, and that's what we do in this else block.

Lines 22–26: In those lines, we check if the two cards match and are different. We already know that they're not the same cards, thanks to the check at line 14. What we need to do now is to make sure their type (the sprite shown by the card) is the same. That means they match and we can get rid of them by setting their visible property to false. That will make them vanish (not really, they will just become invisible).

Note Every instance in GameMaker: Studio 2 has the `visible` property. It's a very trivial property that can be set as `true` or `false` and decides whether the instance is visible or not.

When an instance is not visible, its Draw event won't be drawn. That means that all the events bond to its sprite, like mouse clicks or collisions with other sprites, will trigger anyway, but the sprite won't be visible.

Lines 27–30: We finished to check the cards, and we can freely empty the global.cards array and flip back the cards (even if they are not visible).

Running the game, you may notice that everything works fine: we can pick cards, and if we pick two matching cards, they disappear (Figure 4-3). That's perfectly good! Our game is working as expected!

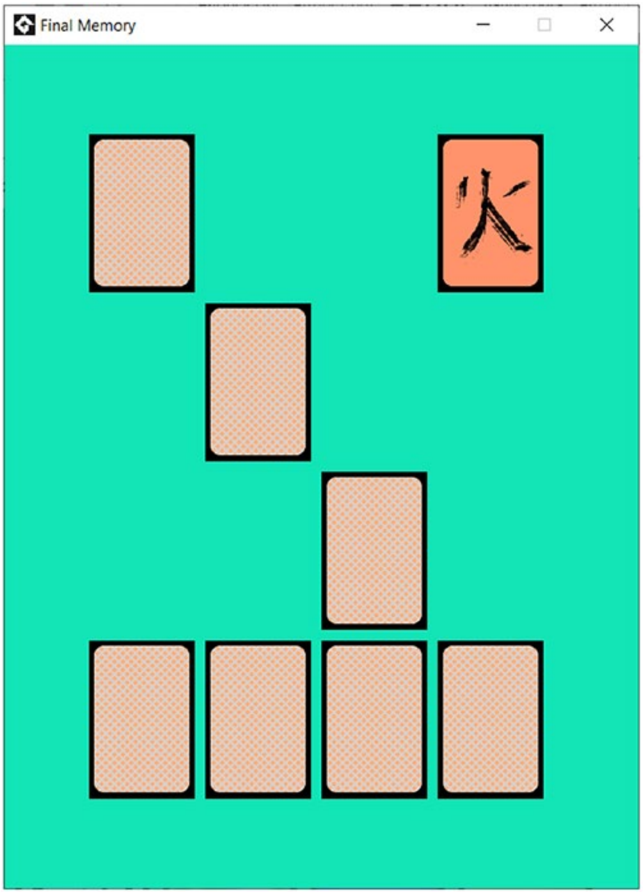


Figure 4-3. *The first working version of the game*

There's just one weird thing: we don't really see the second card we flip, even if we tell the game to flip it, in obj_card's Left Mouse Button Pressed event (of the preceding code). This is because the calculation made by the game, to check the two cards, is too fast and it's completed before the game can even show us the card flipped, so it flips the card back before we can see it.

This may look as not a big deal, but it actually is! In fact, without the player being able to see the second card, it's very difficult for them to have a clear idea of which card is being picked; also, the progression of the game is way slower than expected, because the player is confused and the game doesn't play natural.

Having a confusing feedback from a game, it's not fun, because it looks like it's not right or it's broken. The player may even start thinking that the game is tricking them. Overall, an unclear feedback from a game gives a sensation of shallowness and prevents the player to concentrate on the game and immerse in it. Immersion is a crucial thing in games, because it's the characteristic that determines how much time the game is being played and with how much enjoyment. We need to boost immersion in our game and make sure that the only thing that our player thinks while they play our game is the game itself, not the bugs or the imperfections, but just the gameplay. We want them to fully immerse themselves into the gameplay and just enjoy the challenge or the chill of our game. We want the player to feel and enjoy the rhythm of our game.

Those are things to always keep in mind when designing a game. A good gameplay always keeps the player happy and amused by giving them a good challenge or a relaxing gameplay. The game must be fun and clear and the experience enjoyable and natural. Long story short, we need to fix our game!

Tip When you create your game, always pay attention to the *fun factor*.

The fun factor is that thing that makes your game enjoyable. You need your game to be fun, if you want it to be played. So spend time thinking how your game can be more enjoyable. Maybe you can boost your gameplay by adding a new game mode or a new challenge? The possibilities are endless!

The idea is to slow down the process of checking the cards, so that we can see for some moments the two cards selected and then the game will make them disappear or flip them back.

To do that, we need to introduce a concept: timers.

A matter of time

A timer is a common concept in programming and represents an object that counts the passing time and triggers one or more events when a certain amount of time has passed.

Somehow a timer is not very different from the concept of kitchen timer. A kitchen timer is a tool that can be set to a certain value. The value you set is the number of seconds you want the timer to wait before it rings. After you set the time, the timer starts counting down from that value to zero; and when it finishes, it rings.

Timers are very useful in a plethora of applications. Game development is probably the one kind of application that uses that concept the most.

In GameMaker: Studio 2, timers are implemented as alarms. An alarm is just a variable that can be set to a value and starts to decrease constantly until it reaches zero. When it does, it triggers the event related to it and executes some code.

Actually, what an alarm does is not counting time, but frames. You set the number of frames you want the alarm to wait, and after that amount of frames is rendered, the code attached to the alarm is executed.

Counting frames, instead of time, may be confusing at first; but it's the only thing that really makes sense in a video game. In fact, games – as you may have noticed playing them – run at a certain amount of frames per second. The number of frames per second (FPSs) determines how smooth the game is. Our game is running at 60 frames per second, which means that if we want to set an alarm that triggers after 1 second, we need to set it so that it waits 60 frames; if we need it to trigger after 2 seconds, we set it at 120 frames and so on. In general, the rule is

```
Time to wait = game_FPS * seconds
```

Tip In GameMaker Studio 2, every room has its own FPS value. So you can make different levels or sections in your games that run at different FPS rates.

To change a room's FPS rate, open up your room; and in the Properties section on the left, search for the room_speed field. You can access the FPS of a room anytime by code by accessing the variable room_speed. That variable will tell you the FPS of the current room (the room in which the instance is).

Alarms, in GameMaker, are organized in an array. Each element of the array represents an alarm that you can use. You can set 12 alarms for each instance in your game.

An alarm is inactive when it's set at value -1. If you set an alarm to a positive value, it will count down that specified number of frames and then trigger the code that you specified in its event.

To set an alarm, in GML, you just need to assign a numeric value to the right element of the alarm array, like this:

```
alarm[0] = 60;
```

The preceding code will set alarm 0's value to 60, meaning that the alarm will count down a time value of 60/room_speed seconds (remember that room_speed is the FPS value of the current room). So, if your game runs at 30 FPSs, the alarm will wait 2 seconds (60/30 = 2); while if it runs at 60 FPSs, it will wait 1 second (60/60 = 1).

To write a piece of code for alarm[0], you have to add the event handler for that alarm in your object by selecting in the Events window: Add Event

➤ Alarm ➤ Alarm 0 (Figure 4-4). Inside that event, you can write the code that you want the game to execute when the alarm runs out.

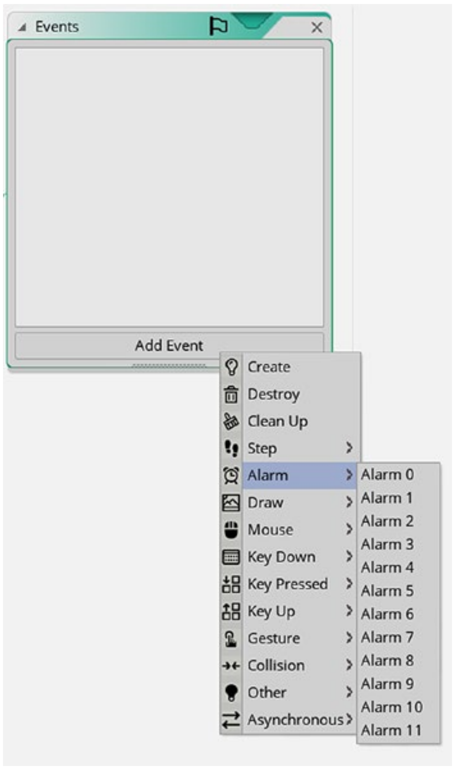


Figure 4-4. *Creating an Alarm event*

To make our little modification to the gameplay, we will set alarm[0] to, let's say, 0.5 seconds. Then we will attach the code to check the cards to the alarm[0] event.

Doing so, when the player picks the second card, the game will flip the card, wait half a second, and then do the checks. That way, the player would be able to see both the cards having a clearer visual feedback and enhancing the gameplay.

Go ahead and open up `obj_card's Left Pressed` event and modify the code so that it will look like this:

```
if ( visible and global.game_state == states.paused )
{
    face = 1;

    if ( global.cards[0] == noone )
    {
        global.cards[0] = id;
    }
    else
    {
        global.cards[1] = id;

        if( global.cards[0].index == global.cards[1].index )
        {
            face = 0;
            global.cards[0] = noone;
            global.cards[1] = noone;
        }
        else
        {
            global.game_state = states.playing;
            alarm[0] = 0.5 * room_speed;
        }
    }
}
```

We are moving the cards' check from Left Pressed event to put it inside the Alarm event.

Lines 21–22: Substitute the check code, change the game's state to Check (1), and set alarm[0] to a value of half a second. This is made by setting alarm[0] to 0.5 times the room's speed. The speed of a room represents the number of frames per second that room is rendering. So if our room is set to 60 FPSs (room_speed = 60), what we are doing is setting alarm[0] to 60 * 0.5.

Now create an event for alarm[0] in obj_card, by selecting Add Event ➤ Alarm ➤ Alarm 0 (as in Figure 4-3), and put this code inside:

```
1   if ( global.cards[0].type == global.cards[1].type )
2   {
3       global.cards[0].visible = false;
4       global.cards[1].visible = false;
5   }
6
7   global.cards[0].face = 0;
8   global.cards[1].face = 0;
9
10  global.cards[0] = noone;
11  global.cards[1] = noone;
12
13  global.game_state = states.paused;
```

This is exactly the same code we were using before to check the cards, except for line 13, where we're setting the game's state back to Wait (0).

Go ahead and run the game. Now everything works fine (Figure 4-5)!

When you pick two cards, the game shows them to you for half a second and then decides if they match or not. If they do, they disappear; if they don't, they are flipped back. In any moment, when you press the enter key, the cards are reset and shuffled.

Awesome!

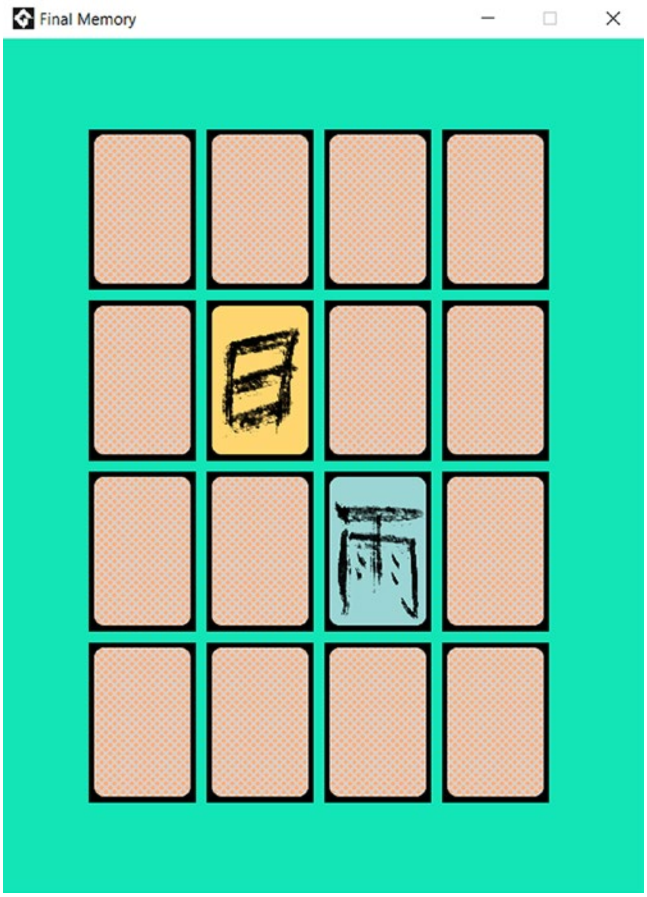


Figure 4-5. *Now we can see both the cards for some seconds before the game decides if they match or not*

We have this smooth gameplay, but we still lack the thrill of winning and losing. It's time for us to implement some fun victory condition!

Play to win!

Everyone likes to win. That's a fact! But whether you win or lose, that's not the main reason why you play. In fact, people play a game either because the game is fun or to get a reward. Since we are offering no rewards, we have to stick with the fun gameplay!

To create a fun gameplay, we have to think about how we can make the game interesting and worth playing. One of the first things that come in mind is to add a challenge that can be failed or a score that can be used to confront all the players who played that game (or both).

Since the first games ever made in the first generations of the gaming industry, lives, score, and time-based gameplay were present as elements to determine how the player could win or lose. Those three gameplay elements were the only way a game could be lost or won, back in the days. Let's think about masterpieces like Pac-Man, Arkanoid, Space Invaders, Tetris, and Super Mario Bros. They all have lives or score or time as a victory or losing condition.

For example, in games like Pac-Man and Tetris, you would go ahead completing the levels one by one until you die (in game, possibly); and when you do, your name would be inserted into a rank and compared to other players. The thing that glues the players to those games is the fact that they could have the highest position in that rank and feel like they are the best.

Memory is a kind of game that can be enjoyable for the sake of it. Just like Mahjong and other similar games, Memory is a chill game. You play it because it's relaxing, and you don't need to have a rank to climb to be content with the experience, meaning we can skip the score thing and concentrate only on the losing condition. We don't want the game to be too chilly so that's pointless to play it, so we are adding a bit of a spicy rule to our game: you have to pair all the cards in a specified amount of time; if you take too much time to do it, you lose.

In our GDD, we designed the game as time-based. That means that we lose if after a certain amount of time, we still haven't matched all the cards on the table.

To implement this, in our game, we can use the concept that we just introduced in the previous section: alarms.

We will set the alarm every time we start or restart the game at a fixed value of – let's say – 60 seconds. When the game starts, the time decreases until it reaches 0. When this happens, the game checks if the game is lost and acts consequently.

To take track of the passing time, we need a couple of variables: one that represents the total amount of time we have (we will use it to set and reset the alarm) and one that represents the current time (it decreases while we play, and we use it to display the remaining time).

Let's do this!

Open up your obj_controller. Inside the Create event, put these two lines on the top of the code:

```
play_time = 60;
cur_time = 60;
```

As we mentioned earlier, we will use those two variables to store the time in seconds.

`play_time` is the total time that we can play the game. We will use this variable anytime we want to start/restart the game to set the alarm to the right value.

`cur_time` is a variable that will store the time at any moment and will be used to be displayed on screen for the player to see. What we are going to do with `cur_time` is to set it equal to `alarm[0]`, but we can't use alarm[0] directly because when it will run out, it will go back to its natural status of -1 and we don't want our displayed time to be `-1`. So we will use `cur_time` as our time variable so that we can manually set it to 0.

We have to check the victory conditions when `obj_controller`'s `alarm[0]` runs out. When it does, we want to check if the victory condition is satisfied or not and change the game state accordingly, so that we know that we are in the Victory or Loss state.

So let's create a new Alarm 0 event for `obj_controller` and add this code to it:

```
1   cur_time = 0;
2
3   if ( all_cards_matched( deck ) )
4   {
5       global.game_state = states.won;
6   }
7   else
8   {
9       global.game_state = states.lost;
10  }
```

Line 1: Time's up! We're setting cur_time to 0.

Lines 3–10: all_cards_paired() is a function we still have to write. It will check if the player paired all the cards or not returning true or false.

If all the cards are paired, the game goes in state 2 (Victory); if they're not all paired, the game goes in state 3 (Loss).

Now let's write `all_cards_matched()`! Similarly to `shuffle_cards()`, this is a function that takes one argument in input. We are going to check all the cards in the deck and see if there is at least one card that is visible. In that case, we return false because if there's at least one card visible, that means that not all the cards in the deck were paired.

Let's create a new script called `all_cards_paired` and put this code in it:

```
var deck = argument0;
var deck_size = ds_list_size(deck);

for ( var i = 0; i < deck_size; i += 1 )
{
```

```
6       if ( deck[| i].visible == true ) // there is at least
        one card visible
7       {
8           return false;
9       }
10  }
11
12  return true;
```

Lines 1–2: We take the argument into the variable deck, and we define a variable called deck_size containing the size of the deck.

Lines 4–10: We loop the cards in the deck and check if there's at least one of them with the `visible` property set to true. If there is, we return false.

Line 12: If the game executes this line, it means that no visible card was found in the deck, so we can safely return true.

Now that we have a function to check our victory condition, we should call it also inside `obj_card`'s `Alarm 0` event, so that every time the player picks two cards and they match, we can check if they won the game (they paired all the cards) or not.

Open up `obj_card`'s `Alarm 0` event and modify the code so that it looks like this:

```
if ( global.cards[0].type == global.cards[1].type )
{
    global.cards[0].visible = false;
    global.cards[1].visible = false;

    if ( all_cards_paired(deck) )
    {
        global.game_state = states.won;
    }
}

```

```
12  global.cards[0].face = 0;
13  global.cards[1].face = 0;
14
15  global.cards[0] = noone;
16  global.cards[1] = noone;
17
18  if ( global.game_state not (states.won or states.lost) )
    // if state isn't Victory or Loss
19  {
20      global.game_state = states.paused;
21  }
```

Lines 6–9: When two cards match, we check if there are any left. If there aren't any more cards, we set the game state to 2 (Victory).

Lines 18–21: If the game is not in the state of Victory or Loss (2 or 3), it means that there are still cards to pair, so we can safely set back the game state to Wait (0).

Now nearly everything is set. We only have to visualize the time remaining, so that the player knows and sees a message when they win or lose.

To show the time, we have to create some sort of HUD. A HUD (Heads-Up Display), also known as Status Bar, is a part of the Graphical User Interface (GUI) which is commonly used to display useful data about the status of the activity the user is doing. We can use a HUD to show the player those information related to the game while it is running.

GameMaker Studio 2 has a special event to handle GUI drawing, which is called Draw GUI event.

Draw GUI is a sub-event of Draw specialized to GUI element drawing. The difference between the two is that in the Draw GUI event, the coordinates of the screen are fixed and are not bond to the room, but to the game window. That means that if you draw a picture or a text at coordinates x:0, y:0, it will always be drawn in the upper-left corner of the screen, even if it's not correspondent to the upper-left corner of the room.

What we need to do inside of Draw GUI event is to check whether the state of the game is set to Victory or Loss and draw a victory or loss text. If the game state is neither in one or the other state, we just update the value of `cur_time` and draw it on the screen.

To write a message, we need a font in our assets. So let's create one. Right-click Fonts in the Resources panel and select Create Font. You can use the font you like. Just make sure you create two fonts, one called fnt_message of size 20 and one called fnt_timer of size 12.

So let's create a new Draw GUI event for obj_controller and put this code inside:

```
if ( global.game_state == states.won )
{
    alarm[0] = -1;
    draw_set_font(fnt_message);
    draw_set_color(c_red);
    draw_text(room_width/2 - 100, room_height/2 - 100, "YOU
    WON");
}
else if ( global.game_state == states.lost )
{
    alarm[0] = -1;
    draw_set_font(fnt_message);
    draw_set_color(c_red);
    draw_text(room_width/2 - 100, room_height/2 - 100, "YOU
    LOST");
}
else
{
    cur_time = ceil( alarm[0] / room_speed );
}

```

```
20  draw_set_font(fnt_timer);
21  draw_set_color(c_white);
22  draw_text(90, 0, "Time left: " + string( cur_time ));
```

Lines 1–7: We check if the state of the game is set to Victory. If it is, we deactivate the timer forcing it to `-1`, and then we write a victory message in the middle of the screen.

Note draw_set_font and draw_set_color must be called every time you want to write a text calling draw_text. In fact, those two functions allow you to set the font and the color of the text you're going to write on screen.

Lines 8–14: If a game's state is set to Loss (3), we draw on screen a loss message, just like we did in lines 1–7 for the victory message.

Lines 15–18: If the player has not won or lost the game yet, we just update `cur_time` variable by setting it to the value of alarm[0] divided by the room_speed. That will give us the time remaining in seconds. We use the ceil function on the result of the division, to get the value as an integer (or we would have a decimal number).

Lines 20–22: We draw the time on screen making use of the cur_time function.

As a finishing touch, we want the time to be reset also when we reshuffle the cards pressing the enter key. We also want to make sure that the game state is set again to Wait (0).

So let's open up obj_controller's Key Down ➤ Enter event and add these two lines at the end of the code:

```
alarm[0] = play_time * room_speed;
global.game_state = states.paused;
```

Those two lines will make sure that when you press enter, the game timer will be reset to 60 seconds and that the game state will be set back to

Wait (0), so that the player can play again and the victory/loss message will not be prompted anymore.

Let's run the game and see if everything's alright (Figure 4-6)!

If you followed and coded along, the game should play as expected, and now you have a complete game with victory and loss conditions that's replayable and fun!

Good job to you!

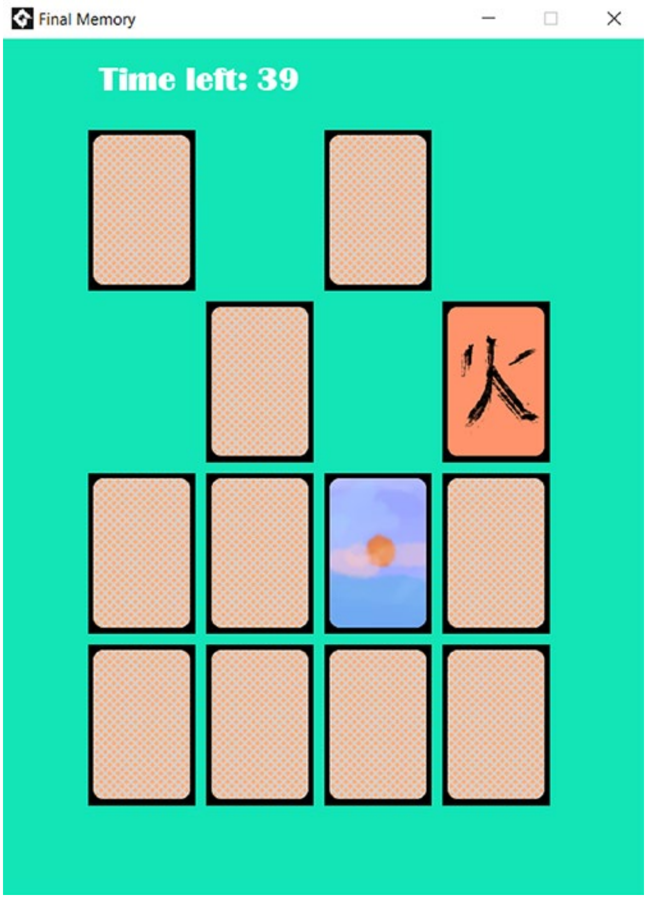


Figure 4-6. *Our complete memory game with a time-based gameplay!*

You have your very first and complete video game made with GML and GameMaker: Studio 2! Congratulations!

Take your time to enjoy this exciting moment and feel great, because you achieved something impressive! Just think about it: before starting to read Chapter 3, you probably never coded a game in your life; and now you have a full and working card game with time mode, nice graphics, and a cool gameplay!

Never forget to recognize and cheer on your achievements, because this helps you to not giving up and not feeling like you're wasting time.

Now that you have Memory, you can free your imagination and try to improve it by adding some features, some game modes, or anything else you might think about.

In the next chapter, we are going to face another cool and exciting project. We will code a top-down shoot 'em up game introducing a lot of new features and concepts like enemies, bullets, score, and many more!

TEST YOUR KNOWLEDGE!

1. What is a Finite-State Machine?
2. How can a FSM help you creating your game?
3. Can you design a FSM of a vending machine?
4. What are game states?
5. Can you describe the meaning of the Wait state in the game flow?
6. Why is the design phase so important?
7. What are enumerators?
8. How can you declare and access an enumerator in GML?
9. What is the scope of enumerators?

10. Can you describe the visible property of an object?
11. Can you interact with an object while it's not visible (visible == false)?
12. What's an HUD?
13. What's the difference between Draw and Draw GUI?
14. What do we mean when we talk about the fun factor?
15. Why is immersion important in your game?
16. Can you think of a new feature you can add to our Memory game to make it more fun? Can you design it?
17. What is an alarm in GameMaker, and how does it work?
18. If I set my alarm to the value of 120 and my room runs at 30 FPS, after how much time will the alarm be triggered?

CHAPTER 5

Fixed Shooter

Making your first video game, you made the very first step into the world of game development! Cheers! But don't settle on that! The job of a game developer is to constantly study, play, and make games. So let's start a new project!

In this and the next chapter, we are going to create a game belonging to one of the most popular and important genres of the video games history: shoot 'em up!

A shoot 'em up (or shmup) game consists in the player facing multiple enemies shooting 'em up (you don't say?) and dodging their bullets.

It's still debated which design elements are canonical in a shmup game. Anyway, we are happy with the recognized basic definition: a game in which the player has to face multiple enemies shooting them and dodging their bullets. That's exactly what we are going to create in this chapter!

Anyway, it's important to do a little introduction to the genre, so that we can understand better what we are talking about.

History of the genre

Shoot 'em up (aka shmup or STG) is one of the most enduring and purest game genres in the history of video games. Born in 1962 with Spacewar!, the genre had its golden age in the years between 1980 and 1995. This majestic era for the genre brought to us some of the best shmups of all

© Sebastiano M. Cossu 2019
S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_5

time; legends like Space Invaders, Galaga, 1940 series, Darius, Ikaruga, R-Type, Raiden, and DonPachi – but also the entire run-and-gun sub-genre (e.g., Contra and Metal Slug) – saw the light in those years.

If the 1980s were the years of the definition of the shoot ‘em up genre, where games perfected those that are now the standards in the genre (like scoring mechanics, waves of enemies, bullet patterns, etc.), the 1990s were the years in which those concepts were pushed to their limits: enemies greatly increased in number; and bullets started to fill the screen like never before, with a lot of colors, patterns, and flashes. Some shmups started to become something more than improvements on Space Invaders and Galaga; they became precursors of a new extreme sub-genre: bullet hell.

Bullet hell (aka manic shooter or danmaku) is a sub-genre of shoot ‘em up and focuses on the player dodging complex patterns of enemies and bullets while scoring points by killing enemies. Bullet hell games often hide some interesting mechanics like DonPachi’s combo mechanic in which you have to keep the combo chain to maximize your score or Bangai-O’s grazing mechanic in which you have to graze bullets without taking damage to greatly increase your counter-attack’s power.

The possibilities with shoot ‘em ups and their sub-genres are endless, and they surely can teach you a lot about game design and gameplay mechanics. Think about it: they are improving on the same concept of killing a lot of enemies shooting at them since 1960s, and they still are popular and one of the world’s favorite genres.

It’s important and very interesting to note how pure this genre remained in the years. In fact, some of the core mechanics are still there, untouched. For example, in a STG, it is very important that controls are precise and smooth. Another interesting thing to note is that background story is very marginal, because gameplay is the main focus of the genre. STGs’ players want to deal with a huge number of enemies and test their reflexes and dodging skills (often in fast-paced levels) and don’t really care about the reason why they’re doing it.

That's why it's a very interesting case of study to us! We can concentrate on game mechanics and gameplay and study some of the most interesting features that a game can have, like bullets, enemies' movements, camera scrolling, power-ups, and of course energy and ammunition management.

In this chapter, we are going to create a fixed shooter, that is, a shoot 'em up game with fixed screen (non-scrolling level) and some limitations, like the fact that enemies are lined up attacking or advancing at regular intervals of time and the player can move only left and right and has just one type of attack. Some famous fixed shooters from which we are taking the inspiration are Space Invaders (Taito, 1978) and Galaga (Namco, 1981).

The next chapter (Chapter 6) is going to build upon Chapter 5. We will create a top-down STG (like Ikaruga, Star Fox, DonPachi, etc.) using assets and code from our fixed shooter and improving on it adding new features like power-ups, vertical scrolling, enemies following patterns, and other exciting features like boss fights! We are also dedicating a special chapter (Chapter 7) to boss fights, which are a super-important aspect of video game design. We are going to design and implement our very first boss fight (Chapter 6) and think about how to make it an interesting and challenging experience for the player (Chapter 7).

So let's go ahead and design our shoot 'em up game starting – as usual – with a clear and well-written game design document!

Space Gala (GDD)

Space Gala is a single-player shoot 'em up game – specifically a fixed shooter – in which the player has to eliminate all the enemies shooting at them and survive dodging all their attacks before they get too near the home base.

Story and setting

You are colonel Jonathan Spacepants, and you are the last hope for mankind. Your mission is to destroy the alien fleet before they reach our space station.

Gameplay

Space Gala revolves around dodging and shooting. It's very important to keep the player focused on those two activities maintaining a fast pace but also giving the player a good amount of satisfaction and motivation.

Satisfaction should be reached by making smooth controls and a nice-to-use weapon.

Motivation can be reached by giving the player a sense of progression. Progression can be achievable by increasing the difficulty of the game from level to level.

Victory conditions

The game can be won by eliminating all the enemies in a level.

You can lose both by dying and allowing the enemies to advance reaching the bottom of the level.

Controls

The player can control the spaceship by using the **arrow keys** to move left and right only and the **spacebar** to attack.

It's very important, for the genre, to have precise and smooth controls. We don't want to add any friction in the player's movements.

Right Arrow: Move right.

Left Arrow: Move left.

Spacebar: Attack – a single bullet dealing standard damage.

Esc: Open the menu.

Menu

You can open/close the menu by using the Esc key. Via the menu, you can close the game, restart it, or resume the paused game.

A smaller version of the menu should be shown when the game is over to allow the player to restart or close the game.

Pacing

The sense of urgency should be the preponderant feeling in Space Gala. You need to wipe out a fleet of aliens before they reach the base and/or kill you. The aliens are continuously moving, and you need to be a fast and precise shooter to deal with them quickly.

Enemies

There is just one type of enemy:

- **Reds**: Basic enemies that move left and right and advance while randomly shooting.
    - HPs (Health Points): 1.
    - ATK (How much damage they inflict): 1.
    - Movements: They move left and right and regularly jump down by X pixels.

Game modes

There is just one arcade game mode. The player must kill all the enemies to win the game.

Level 1

Level 1 is pretty simple. The player has to face a fleet of nasty red aliens that want to approach the space station.

The aliens will dodge the bullets by continuously moving left and right while they recharge their FTL engines to jump toward the space station. They can jump no more than 30 pixels forward, and they need to wait approximately 5 seconds before the FTL engine recharges and they can jump again.

The aliens will shoot randomly in front of them (easy to dodge).

The level is made of one group of 16 aliens.

Similar games and influences

Space Gala is obviously inspired by Space Invaders and Galaga.

The gameplay is more like the Space Invaders experience, with the aliens descending gradually and sporadically shooting.

Other notable games of the same genre are Centipede, Galaxian, and Moon Cresta.

Target audience

Fixed shooter today is a niche sub-genre of STG. The audience is not very wide, but it's super passionate and cares a lot about the purity of the genre.

From GDD to the game

Let's start by creating a new project called Space Gala. To do that, open GameMaker: Studio 2 and select File ➤ New Project from the menu bar at the top of the window.

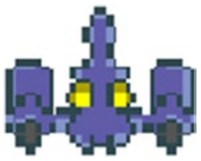
Now that we have our new project, we need to create all the assets to get started and build our game!

Assets

We will need some sprites and other assets for our game. You can either make them yourself or download them from the companion GitHub repository of this book.

Whether you download them or made them yourself, the following is a complete list of all the assets needed in this game. Make sure that your custom assets are compatible with those or that you make the right changes to avoid any incompatibility.

spr_player

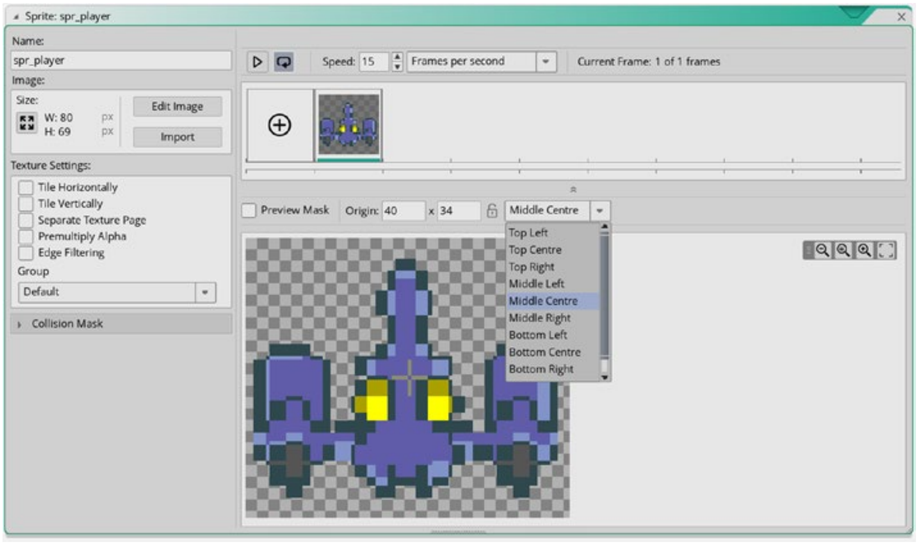


This is the sprite representing the player's spaceship.

This sprite is of dimensions 50 × 43 pixels and has the pivot point in the middle-center.

The **pivot point** is the reference point that will be used to calculate the position of the sprite. For example, if we set the pivot in the middle-center of the image, when we will move an instance using that sprite to coordinates 0,0, the center of the sprite will be exactly at coordinates 0,0, so we will be able to see only the bottom-right corner of the sprite.

To change the pivot point of a sprite, open up the sprite and head to the combo box just above the sprite preview (Figure 5-1). There you can select the point in which you want your pivot point to be. Alternatively, you can select *Custom* from the combo box and select the point yourself by clicking directly on the image in the point of the sprite you want your pivot to be.



Sprite: spr_player

Name: spr_player

Image:

Size: W: 80 px H: 69 px

Speed: 15 Frames per second

Current Frame: 1 of 1 frames

Texture Settings:

- Tile Horizontally
- Tile Vertically
- Separate Texture Page
- Premultiply Alpha
- Edge Filtering

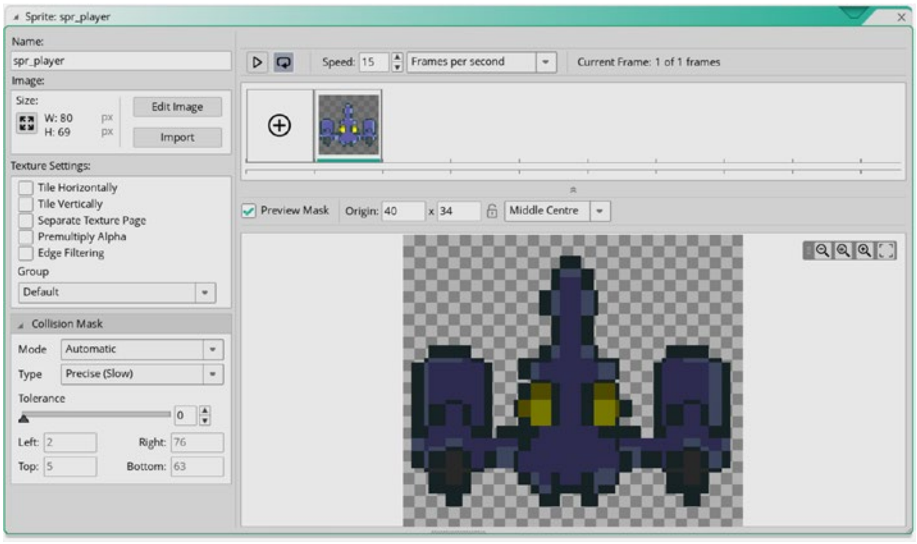
Group: Default

Preview Mask Origin: 40 x 34 Middle Centre

Figure 5-1. *Changing the pivot point for a sprite*

Because we want out spaceship to be hittable by enemies' bullets, we need to tell GameMaker that this specific sprite can collide with other objects. How do we do that? Using collision masks!

A sprite's collision mask is the area used to calculate if that sprite is colliding with any other sprite or not – if it does, a collision event for the object associated with that sprite is triggered.



Sprite: spr_player

Name: spr_player

Image:
Size: W: 80 px H: 69 px
Edit Image
Import

Speed: 15 Frames per second
Current Frame: 1 of 1 frames

Texture Settings:
- [ ] Tile Horizontally
- [ ] Tile Vertically
- [ ] Separate Texture Page
- [ ] Premultiply Alpha
- [ ] Edge Filtering

Group: Default

Collision Mask
Mode: Automatic
Type: Precise (Slow)
Tolerance: 0
Left: 2 Right: 76
Top: 5 Bottom: 63

✓ Preview Mask Origin: 40 x 34 Middle Centre

***Figure 5-2.** Applying a collision mask to a sprite*

To create a collision mask for spr_player, just click the Collision Mask button to open up a new section that allows you to set up the collision mask you want (Figure 5-2). There are various options. Let's check them out:

- **Mode**: The mode decides how the mask is positioned on the image and can be of three types:
    - Automatic: GameMaker calculates by itself where to put the mask based on the image (it basically tries to fit the colored parts and ignore the transparent pixels).
    - Full Image: The mask is applied on the entirety of the image ignoring transparency.
    - Manual: You have to position the mask on the image by yourself.

- **Type**: The type of the mask decides the shape and nature of the mask. It's the most important thing, since it's the one setting that can make your game run slower.
    - Rectangle: The mask consists in a rectangle. If any sprite enters the coordinates of this rectangle, the collision event is triggered.
    - Rectangle with rotation: It's a rectangle collision mask that is able to rotate with the sprite to keep the right collision area.
    - Ellipse (slow): The mask consists in an ellipse.
    - Diamond (slow): A diamond-shaped collision mask – very useful to approximate plane-like objects like our spaceship!
    - Precise (slow): The mask precisely mimics the image shape (works only with Automatic and Full Image modes).
    - Precise per frame (slow): The mask precisely mimics the image shape and is recalculated once per frame (in case you rotate it or resize it).

For our spaceship, an automatic rectangle-shaped mask is more than sufficient; but if you want, you can alternatively use a diamond or precise mask; since it's just one object, it won't affect much the performance of the game.

spr_bullet_player

This is the sprite we are going to use for the player's bullets.

It's a 16 × 16-pixel sprite representing a yellow pellet.

Its pivot point is in the middle-center.

You can safely select an automatic rectangle collision mask, for this one.

spr_life

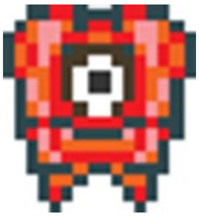
This is the sprite we are going to use to represent the player's HP.

It's a 16 × 16-pixel heart-shaped sprite.

Its pivot point is in the middle-center.

This one doesn't need a collision mask, as you might have guessed.

spr_enemy_red

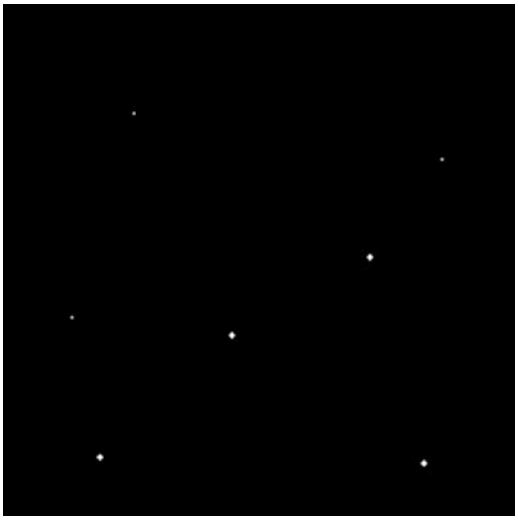


This is the sprite that we will use for red alien spaceships.

It's a 16 × 18-pixel sprite with pivot point set at the bottom-center.

You can safely choose an automatic/manual rectangle collision mask; but if you feel like it, you can choose a more accurate collision mask, like ellipse or precise.

spr_background



Of course, we need a background image to give the player the idea that we are in deep space!

This is a 256 × 256-pixel image representing the space that we are going to repeat covering all of our room's surface.

Other than sprites, we will also need some fonts to manage the aesthetic of the information we are going to show on the screen (score, HP, menu, etc.).

It's up to you to choose the font type, but here you can find some interesting characteristics of those fonts.

fnt_score

The font that we will use to show information on screen like HPs (Health Points) and score.

Font Type: Arial

Style: Regular

Size: 14

Leave the rest at their default value.

fnt_messages

This is a font we will use to show messages like the current game state or the menu options.

Font: Arial

Style: Black

Size: 14

Leave the rest at their default value.

rm_level_1

Right-click the default room room0 and rename it rm_level_1. Double-click it, and the room will open revealing the Room Editor as a left sidebar.

In the Room Editor, head to Properties ➤ Room Settings and make sure that width is 1024 and height is 768.

Now go in Layers and click Background layer; a new section will appear just below the layers list named Background Layer Properties. Go ahead and click No Sprite to select spr_background. Now check both Horizontal Tile and Vertical Tile, right below it. That's it. The room is properly set up and should show a preview of the spr_background sprite repeating itself for the entire surface of the room, making the impression that we're in front of the vastness of the universe. Cool, isn't it?

Making features, not objects

Now that we defined our assets, let's think about serious stuff and create some cool objects!

We are going to create a total of five objects, for our game: the player, the game controller, one enemy, and two bullets. We will need those objects to manage every aspect of the logics of our game – from movements to shooting to the menu management. We won't cover the objects' creation by creating them one by one, but we will code following the concepts and gameplay elements we need to implement.

It's important to understand that your work is not to assemble pieces of something pre-built by someone else. You're not just following instructions; you are understanding how to make a game by yourself. I want you to be completely independent at the end of this book, so that you can make games and learn new things by yourself, without having to ask someone else to solve a problem for you.

Your work is, and should always be, to add features to your game. So we will cover one by one each feature and gameplay element that we need to implement for our game, by moving back and forth between our assets. I hope this will help you to understand the connection between each and every element that composes your game.

Let's start from the movements. In the next section, we will understand and implement the functionalities to move our spaceships with the arrow keys and to make the enemies move left and right and jump down every five seconds.

Movements

We will start from the player's avatar. Right-click Objects in the Resources sidebar, select Create Object, and name it obj_player.

Make a new Create event for obj_player by clicking Add Event in the object's Events window and selecting Create. Now add to that event the following code:

```
1   hp = 10;
2   spd = 3;
```

Line 1: Sets the player's health to 10.

Line 2: Sets the movement speed for the player to 3 pixels. Since we are going to manage the movement in the Step event and it occurs once per frame, this means that the player will move 3 pixels per frame (3 pixels × 60 frames = 180 pixels per second).

According to our GDD, the player object should be able to move left and right smoothly without any acceleration or friction.

To do that, in the obj_player's Event window, click Add Event and select Key Pressed ➤ Left, to create a Key Press ➤ Left key event for obj_player and put inside it the following code:

```
if ( x > 0 + sprite_width/2 )
{
    x -= spd;
}
```

What this code does is pretty straightforward. It just checks if the player is still inside the left margin of the screen, and if it is, it moves the object by decreasing its X-coordinate by `spd` (that we initialized to 3). This happens once per frame, so – as we just said before – it will be a movement of `spd` ∗ 60 = 180 pixels per second. We need to add half the width of the sprite to 0 (that is the x margin of the room) because the pivot point of our sprite is in the middle-center – if we don't do this, only half of our sprite would be visible, since the coordinates are calculated for the pivot of the sprite and so it would be allowed to move left until the pivot reaches x = 0.

Obviously, we could have safely written x > sprite_width/2 instead of x > 0 + sprite_width/2; but to write that 0 can greatly increase the readability of the code, because it makes clear that we are referring to the room's left margin + half the size of the sprite. Sometimes it's better to be less concise, but make your code clear to understand, especially if the things you're adding don't really make the difference in performance – like in this case.

Let's do something very similar for the right arrow key. Click Add Event again and select Key Pressed ➤ Right to create a Key Press ➤ Right key event and put inside of it the following code:

```
if ( x < room_width - sprite_width/2 )
{
    x += spd;
}
```

The preceding code checks if the object is still inside the margins defined by the width of the room minus half the width of the sprite. We need to subtract half the width of the sprite to the width of the room because the pivot point of our sprite is in the middle-center – not doing this means that we would see only half of the sprite for the same reason we had to add that value to 0 in the Key Press ➤ Left event.

Now, just to test if we're doing this good, open up rm_level_1 and drag our newly created object in the middle of it. Press F5 or click the Run icon in the toolbar to get the game compiled and started (Figure 5-3).

If you followed the instructions, you will be able to move left and right using the left and right arrow keys, and your character will stop moving reaching the left and right margin. Cool! You made your first control movements! That's an exciting goal! Controls are very smooth and arcade-styled. You move without frictions or accelerations, and that's perfect as it is! No need to tweak it further! Now close it up and go back to GameMaker; we are going to add some more meat!

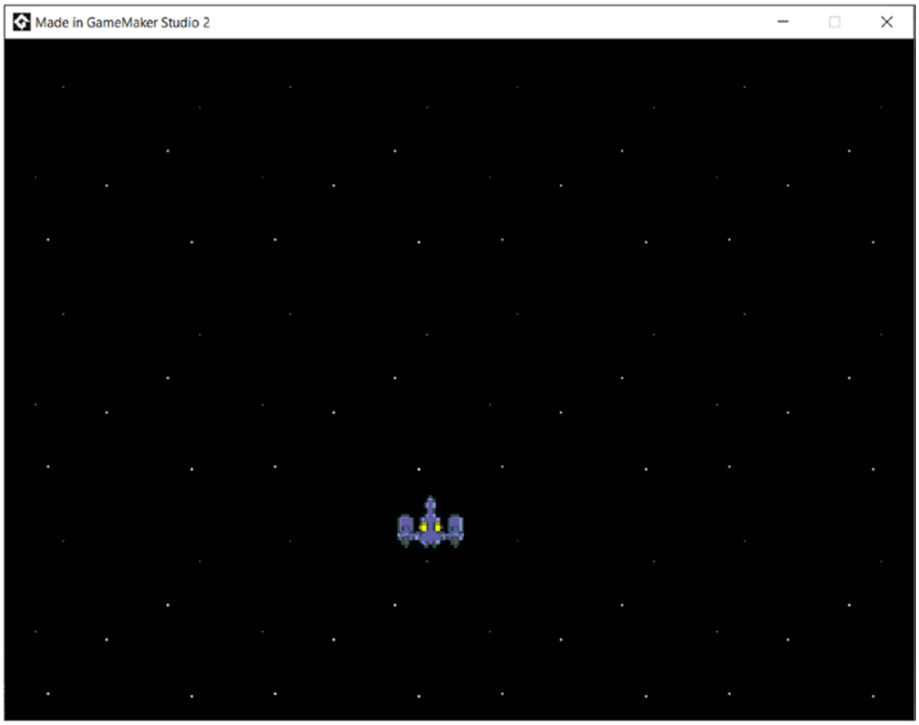


Figure 5-3. *Running the game for the first time*

Now that we have our player's avatar working, we need to take care of the enemies!

Right-click Objects in the Resources sidebar and select Create Object to create a new object called obj_enemy_red. This will be our first enemy! According to the GDD, it's a very weak enemy with just one HP and can deal only one damage. It moves left and right and sporadically shoots a bullet toward us. That's a lot of new things! Let's start step by step! Divide et impera – as the Romans (and coders) say!

We want obj_enemy_red to recreate a swinging movement like the one in Space Invaders and Galaga…well, maybe a little bit smoother! Our red aliens will swing left and right so that they are harder to hit and can dodge some of the player's bullets and they will slowly advance toward the player direction.

This is a gameplay trick to make the enemies harder but without writing an AI algorithm, which requires way more effort and knowledge. Also, we don't need a very advanced AI for our enemies because they are a lot! We can be happy with the fact that they move left and right forcing the player to shoot with precision. The fact that there are a lot of enemies to kill and that they slowly advance creates pressure – that's more than sufficient to make the game challenging.

So the aliens will move from left to right and then go back from right to left and repeat. To do this, we can calculate a complex oscillation function using trigonometry notions, or we can just move toward a direction until we reach a boundary and then invert the direction until we reach the other boundary. I think the second idea is way better, because it's easier to implement and understand and I am a big fan of the KISS (Keep It Simple, Stupid) principle.

That being said, open up the newly created obj_enemy_red and click Add Event ➤ Create to add a Create event. Add to that event the following code:

```
1   hp = 1;
2   atk = 1;
3   spd = 1;
4
5   dir = 1;
6   start_x = x - 25;
7   end_x = x + 25;
8
9   move_down_speed = room_speed * 5;
10  alarm[0] = move_down_speed;
```

Lines 1–2: We are creating an hp variable to keep track of the resilience of this enemy and an atk variable to know how much damage it will deal. In this particular case, the enemy has one HP and deals one damage point, so it would be safe to omit those two variables and just make it die when hit or deal a single damage when they hit something; but – as I'm always repeating – it's important to structure your code so that it's manageable and understandable by you and other people. It's a very good habit to write generic code that can be reused for other similar tasks (in this case, it can be used to create different enemies) and that's easy to understand even by someone who never worked on it.

Line 3: This is the speed variable. Just like for obj_player, we are defining a variable that tells us the speed at which the alien should move. You can set it whatever value you like, but it's fine for now to keep it at 1.

Line 5: This is the variable that decides the direction toward which the alien is moving. In fact, this variable will be multiplied to spd so that we can change its value to a positive value or negative value depending if dir equals to 1 or -1. As you will probably remember, moving right means increasing the x value, and moving left means decreasing it.

Line 6: This is our left limit (25 pixels on the left of the alien's original position). The alien will move left until they reach or pass the X-coordinate; then they will invert their direction.

Line 7: This is the right limit (25 pixels on the right of the alien's original position). The alien will move right until they reach or pass the X-coordinate; then they will invert their direction.

Lines 9–10: Here we declare the variable move_down_speed to 5 seconds and use it to set alarm 0 to that value. We will use this alarm to make the alien ship move down every 5 seconds. You can safely change move_down_speed's value to whichever value you prefer, but I think that 5 seconds is a good amount of time for a first level.

Now that we have our variables set in the Create event, we need to make good use of them to manage the movements of the alien.

Click the Add Event button and select a Step event and write up this code in it:

```
1  if ( x <= start_x or x >= end_x )
2  {
3      dir *= -1;
4  }
5
6  x += spd * dir;
```

Line 1: This line checks if the object is going out of bounds relatively to the start_x and end_x variables we defined in the Create event.

Line 3: If we are going out of bounds (so the condition checked at line 1 is true), we invert the object direction by multiplying dir by -1.

Line 6: Just after we did our checks, we can safely modify the x value by adding to it spd times dir. If dir is positive, we will move right; if it's negative, we will move left.

Note Multiplying a number by -1 always gets you the opposite value because of the rules of arithmetic. When you multiply a positive number by a negative number, you get a negative number; when you multiply a negative number by a negative number, you get a positive number. For more information, check any arithmetic book.

Go ahead and create an Alarm 0 event by clicking the Add Event button and selecting Alarm ➤ Alarm 0. Now write the following code inside of the event:

```
y += 30;
alarm[0] = move_down_speed;
```

This code is pretty straightforward. We increase the y value by 30 so that the alien ship jumps down by 30 pixels (that's a really cool out-of-the-box warp effect, by the way), and then we reset the alarm to the same value so that after 5 more seconds (or whichever value you choose), the ship will jump down by another 30 pixels.

Let's test if we did everything good by dragging obj_enemy_red in the middle of room0, and press F5 or click the Run button in the toolbar to compile and execute the game (Figure 5-4).

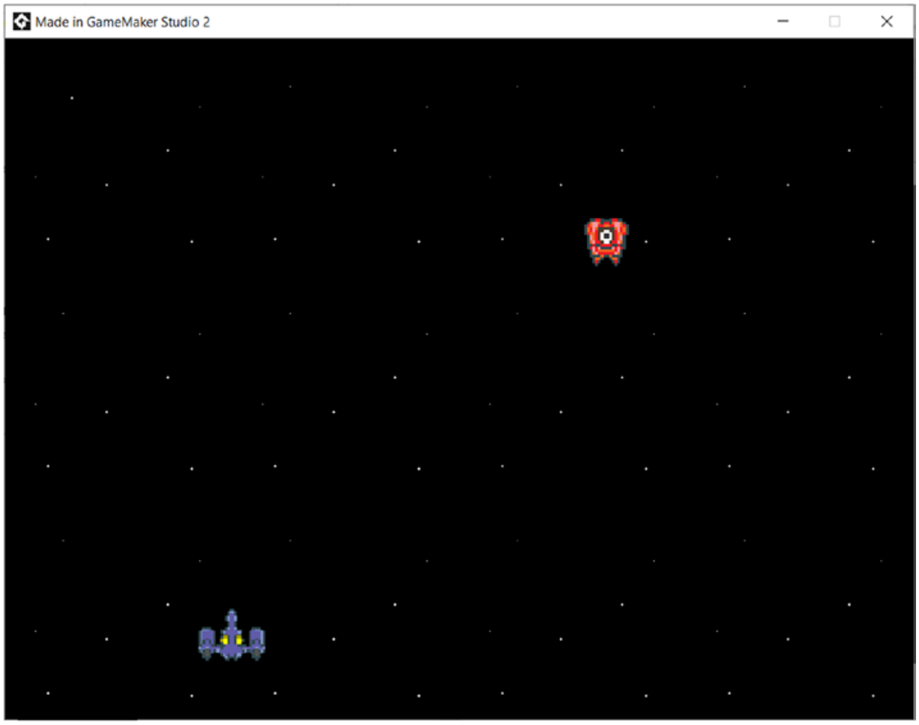


Figure 5-4. *We added to the game the swinging enemy!*

Great! Our red alien ship moves left and right flawlessly, and it jumps down by 30 pixels every 5 seconds! We did all this with just a bunch of lines of code. Amazing!

Now close the game and go back to GameMaker; we are going to work on the shooting (about time)!

Shooting

The concept of shooting we are going to create is based on a variable that both the player and the aliens have: atk.

atk decides the amount of damage that an object inflicts on others. To make this work, we need to pass that value to the bullet object, so that when the bullet collides with an instance of another object, it inflicts the right damage.

Let's start by creating a bullet object.

Right-click Objects in the Resources sidebar and select Create Object to create a new object and name it obj_bullet_player.

This will be the bullet shot by the player when they press the spacebar.

Assign to this object spr_bullet_player and click Add Event ➤ Create and add this one line of code:

```
1   atk = 1;
2   spd = 10;
```

Line 1: We are assigning 1 to the bullet's atk variable so that it deals one damage. We will use this value to decrease the hp of the instance colliding with the bullet.

Line 2: We assign 10 to the bullet's spd variable so that we can later set its velocity by assigning this value to the speed built-in variable.

Note speed is a GameMaker's built-in variable that allows an instance to travel by moving by *speed* pixels per frame in the direction faced by the instance. An instance's direction is decided by the value of the built-in *direction* variable.

Note direction is a GameMaker's built-in variable that represents the direction faced by an instance, and it's expressed in degrees:

- **Right**: 0 degrees

- **Up**: 90 degrees

- **Left**: 180 degrees

- **Down**: 270 degrees

You can implement a rotating system, for example, by gradually changing the value of *direction* by pressing a key. That could be a good way to simulate how a car steering works.

The job of a good bullet is to damage when it collides, so click Add Event in the Events section of obj_bullet_player and select Collision ➤ obj_enemy_red and put this code inside it:

```
1   other.hp -= atk;
2   instance_destroy(id, true);
```

Line 1: other is a reserved keyword that you can use in a collision event to refer to the instance with which the collision is happening – in that case obj_enemy_red.In this line, we are subtracting atk to the colliding obj_enemy_red instance's hp variable, dealing exactly atk damage points. Since our plan is to use the shooter object's atk value, that means that the colliding obj_enemy_red instance will take a quantity of damage depending on the atk value of the shooter instance.

Line 2: After the bullet collides with the enemy instance, it gets destroyed. destroy_instance(id, val) destroys the instance referred by id and triggers the Destroy event if val is true. We want to trigger the Destroy event so that we can make the bullet explode, just after it's destroyed.

Note destroy_instance(inst, val) is a function that destroys the instance referred by inst and triggers the Destroy event for that instance if val is true.

When you call destroy_instance without specifying the arguments like this

```
destroy_instance();
```

by default, the function destroys the caller instance and triggers its Destroy event.

As we just said, we want the bullet to explode when it's destroyed. To do that, click Add Event in the Events section of obj_bullet_player and select Destroy to create a destroy event and add this line in it:

```
effect_create_above(ef_firework, x, y, 0.1, c_yellow);
```

effect_create_above allows us to use the particle system embedded in GameMaker Studio 2 to create a cool particle effect above all the instances. In the note below, you can read a complete explanation of the function.

There is a good amount of pre-made particle effects that you can use; ef_firework makes a cool fireworks-like explosion that we can use to simulate the explosion of our bullets onto the alien ships.

In this case, effect_create_above will make a fireworks particle effect in the exact position in which the bullet collided of size 1/10 of the original size of the effect and yellow color (since yellow is our bullet's color – of course, you can play with that value and change it accordingly to your own taste).

Note You can easily create particle effects with GameMaker by using the effect_create functions.

There are two functions:

`effect_create_above` which creates the effect above all the instances.
`effect_create_below` which creates the effect below all the instances.

The arguments you can pass to the two functions are the same:

```
effect_create_above(kind, x, y, size, color);
effect_create_below(kind, y, y, size, color);
```

where kind is the type of particle effect you want to create (you can find a list on the GML official documentation: `https://docs.yoyogames.com/source/dadiospice/002_reference/particles/simple%20effects/index.html`), x and y are the coordinates at which the effect will be created, size is the scale value of the effect (1 = full size, 2 = double size, 0.5 = half size), and color is the RGB value of the color you want the effect to be drawn.

So now we have our ready-to-use bullet. The only thing missing is the possibility to shoot it. Let's fix this issue!

Double-click obj_player to show the Object Editor and head on the Create event that we made in the previous section.

To be able to shoot and deal damage, we need to add an atk variable to the player object – as we just said. We also want to set a sort of shoot delay, so that we can decide the rate at which we are shooting the bullets – or it will shoot 60 times in a frame, which is suboptimal!

So let's edit the Create event for obj_player so that it looks like this:

```
hp = 10;
spd = 3;
atk = 1;

```

```
5   can_shoot = true;
6   shoot_delay = room_speed * 0.2;
```

Line 3: We added the atk variable, so that we can control the amount of damage that our spaceship inflicts to the enemies.

Line 5: can_shoot is a Boolean variable that we will use to regulate when we can shoot and when we cannot. It will be set to false when pressing the space key and reset to true after shoot_delay steps, so that we can shoot a bullet every shoot_delay steps.

Line 6: shoot_delay is a variable that represents the delay we want to add to our shooting. In this case, we are setting a delay of 0.2 seconds, so that our spaceship's gun will shoot five bullets per second.

In the Events section of obj_player's Object Editor, click *Add Event* ➤ *Key Down* ➤ *Space* to create an event that will trigger when the spacebar is held down and put this code in it:

```
1   if ( can_shoot )
2   {
3       can_shoot = false;
4       var bullet = instance_create_layer(x, y, "Instances",
        obj_bullet_player);
5       bullet.atk = atk;
6       bullet.direction = point_direction(x, y, x, y-1);
7       bullet.speed = bullet.spd;
8
9       alarm[0] = shoot_delay;
10  }
```

Lines 1–3: We shoot bullets only if can_shoot is true. When it is, we change its value to false, so that we can implement that shoot delay we just talked about.

Line 4: We create a local variable to store the obj_bullet_player instance we are creating calling instance_create_layer at coordinates x,y in the Instances layer.

Lines 5–7: We then assign obj_player's atk value to the bullet's atk variable. We set the bullet's direction to point straight up at a speed of spd (which is 10) pixels per second.

Line 9: Lastly we set alarm[0] to a value equal to shoot_delay. Alarm 0 has the job to set back can_shoot variable to true, so that we can attack again after shoot_delay steps (or shoot_delay/room_speed seconds).

Note point_direction(x1, y1, x2, y2) is a function that can be used to change the orientation of an object to face an ideal line drawn between points x1,y1 and x2,y2. The function returns the value in degrees of a vector comprised between the points x1,y1 and x2,y2.

A common usage of this function is

```
direction = point_direction(x1, y1, x2, y2);
```

Last, but not least, let's create the event linked to alarm[0]. Once again, in the Events section of obj_player's Object Editor, click Add Event ➤ Alarm ➤ Alarm 0 and add this line to it:

```
1   can_shoot = true;
```

That's it! We don't need anything else. Just set the variable to true, so that we can shoot again after the right amount of time.

Now let's check that everything is working good by running the game (press F5 or click the Run icon in the toolbar).

Great! Pressing the spacebar we can shoot yellow pellets, and they collide with the enemy making a nice particle effect (Figure 5-5)!

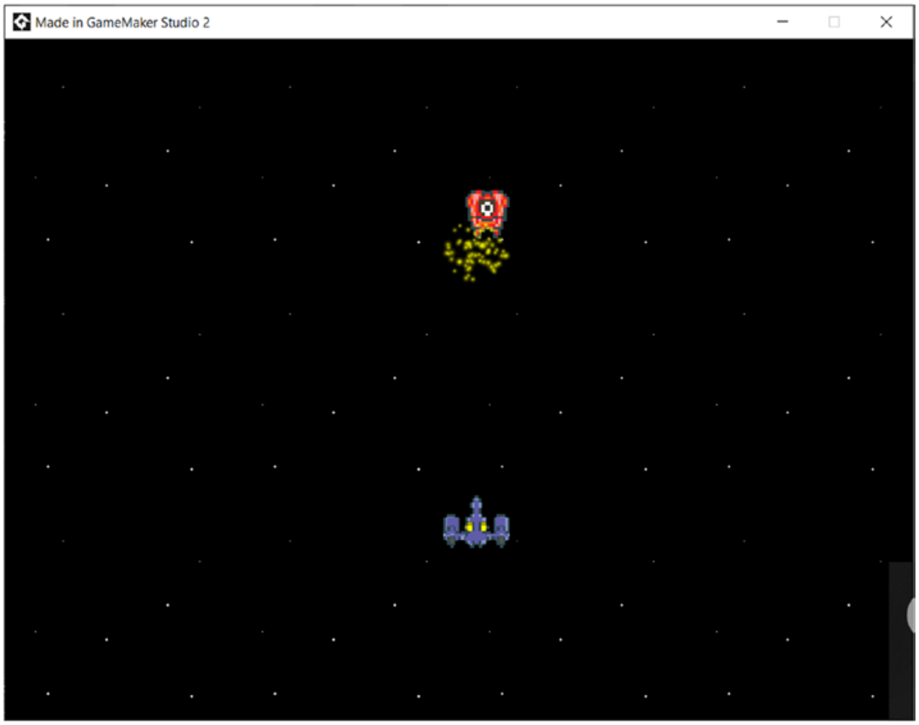


Figure 5-5. *We can now shoot bullets that explode when they collide with alien ships!*

That's good, but not great. A good weapon is such only if it actually kills; and it's not the case! We have to make the enemy mortal.

The bullet does actually deal damage to the enemy, because of that other.hp -= atk; line that we inserted into the bullet's collision event with the alien spaceship, but the enemy is not dying. To make it die, we need to continuously check if enemy's HP reaches zero, because if it does, it must die.

The best way to do it is to modify the step event of obj_enemy_red adding this code at the end of it:

```
if ( hp <= 0 )
{
    instance_destroy();
}
```

The preceding code checks if obj_enemy_red's hp reaches 0; if it does, it calls instance_destroy and triggers the Destroy event. Remember that the default behavior of instance_destroy (that means when you call the function without arguments) destroys the current instance and triggers the Destroy event.

So let's create it, this Destroy event! Click Add Event ➤ Destroy and add this code to it:

```
effect_create_above(ef_explosion, x, y, 1, c_dkgray);
```

This one line creates a cool particle effect explosion of size 1 and color dark gray at enemy's current coordinates.

That looks like it's all set up! Let's check again! Press F5 or click the Run button in the toolbar to compile and run the game.

Okay, that's way better! Now we can shoot at the alien ship and make it blow in a cool smokey explosion! That looks more like it!

Now that we dealt with the player's shooting, we should make it so that even the enemies can shoot!

According to the GDD, the enemies should shoot after a random amount of time. As we learned in Chapters 3 and 4, to generate random numbers, we have to initialize the random seed by calling the randomize function. We don't want to do it inside of obj_enemy_red because that means that the function will be called as many times as many enemies we put in the level. What to do? Of course, we are going to do it inside a game controller object!

Right-click Objects in the Resources sidebar, select Create Object, and name it obj_controller. This will be our game controller object.

In the Events section of obj_controller, click Add Event ➤ Create and add the randomize function call to it:

```
randomize();
```

Now we need to create the bullet object for the enemies. We can use the template of obj_bullet_player by right-clicking it and selecting Duplicate. Rename that copy as obj_bullet_enemy and double-click it to open up the Object Editor.

We just need to change a couple of things in this object. Let's do it!

Head to the Events section of obj_bullet_player, right-click the collision event, choose Change Event, and select Collision ➤ obj_player.

Now we're all set, and the bullet is ready to be shot by the alien ships. We just need to make it so.

Open up the obj_enemy_red's Object Editor and select the Create event. We must make some modification to the code in this event, so that we can properly set up the random attack. Let's add this one line to the bottom of the code:

```
alarm[1] = room_speed * random_range(0.5, 5);
```

We set alarm[1] at a random time value, so that it will trigger between half a second and 5 seconds since when the instance is created.

Now click Add Event ➤ Alarm ➤ Alarm 1 and put the following code inside the event:

```
1   var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet_enemy);
2   bullet.atk = atk;
3   bullet.direction = point_direction(x, y, x, y+1);
4   bullet.speed = bullet.spd;
5
6   alarm[1] = room_speed * random_range(0.5, 5);
```

Lines 1–4: Just like for obj_player, we are creating a bullet, setting its direction (pointing down), speed, and attack power.

Line 6: We reset the timer to a new random value between half a second and 5 seconds.

Now we are all set for the enemy to shoot bullets at random time, but we still cannot die. Let's double-click obj_player and create a new step event by clicking Add Event in the Events section of the Object Editor and select Step ➤ Step.

Inside the Step event, add this code:

```
1   if ( hp <= 0 )
2   {
3       instance_destroy();
4   }
```

Lines 1–4: Just like we did with obj_enemy_red, we are checking every step if hp reached 0 (or less); and if it does, we destroy the instance and trigger the Destroy event.

Now let's add the Destroy event by clicking Add Event ➤ Destroy and add this one line to it:

```
effect_create_above(ef_explosion, x, y, 1, c_dkgray);
```

It's the same line that we used to destroy the alien spaceship. It's more than enough to create a cool effect when our spaceship is destroyed.

Now it should be all set. We can verify it by pressing F5 (or clicking the Run button in the toolbar) and executing the game.

It works! The alien ship is shooting at random intervals of time toward the bottom of the room, and if it hits the player three times, the player dies! That's all we need to play! We have all the elements for our shoot 'em up gameplay!

Now the only things we still miss are a HUD to visualize some information, the menu, and some level design! Let's deal with those features!

Designing rm_level_1

Here we are! We have our enemy prototype and our player's avatar and an empty room. We have to use those elements to make a level. The art of level design is all about creating levels that can be beautiful to look at, fun to play, and possibly narratively interesting.

In this version of the game, being that a fixed shooter, there's not much that we can do with our levels; but we can try to copy the two big names of this genre: Space Invaders and Galaga. Those two games just put a fleet of aliens at the top of the level in an ordered formation and let the battle be consumed in that limited space.

Open up rm_level_1 and Drag and Drop obj_enemy_red instances so that it looks like Figure 5-6.

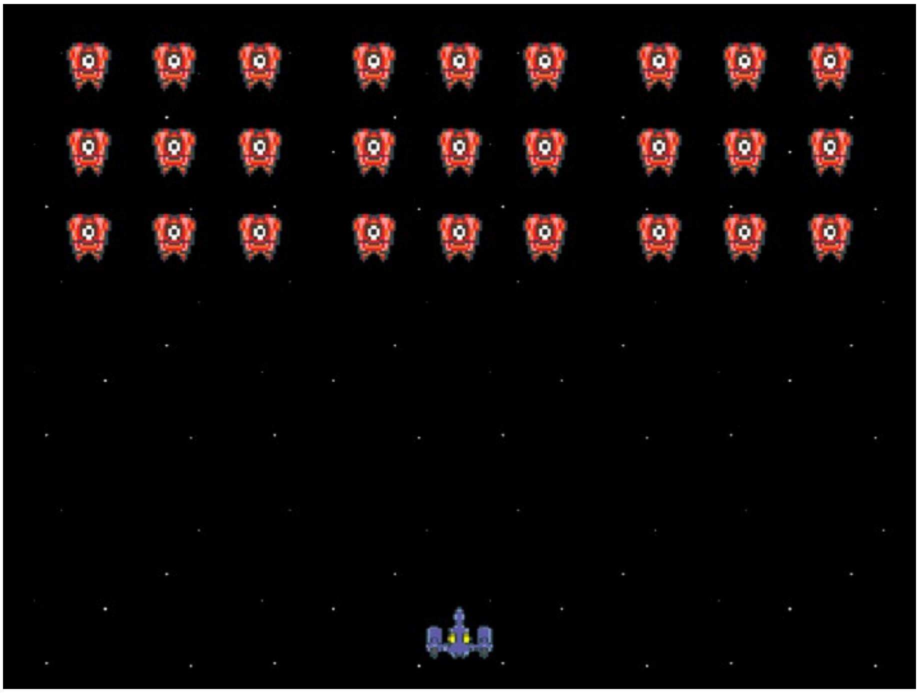


Figure 5-6. *Building rm_level_1 in GameMaker's Room Editor*

Running the game, you will notice how that kind of design is more than enough to make the level challenging. In fact, depending on how often the enemies shoot, it's not so easy to dodge all those bullets and eliminate 27 enemies with just three HPs. I think we reached a good balance for that level.

We now need to think about how we can win or lose the game.

Game states

You probably noted that in Chapters 3 and 4, when we built the Memory card game, we started from designing the game flow and then we implemented the functionalities, while now we are going the other way around. Those are the two most common creative processes to make games.

When you have a clear idea of what you are going to do and you have a clear idea of the game flow, you should start by this and prepare the whole structure of the game from the ground up.

When you know you want to use some specific functionalities in your game (like in this case we knew we wanted some basic shmup functionalities like shooting and moving), but the game flow is still not clear, you should start by prototyping/studying the various features you want to implement in your game – like we did here by building the moving and shooting features before thinking about the game flow.

Both those two ways to create games are viable and good. Sometimes you just need to play around with features to have an idea of what you can make; sometimes you don't need anything but precision and planning from the beginning.

I think starting from the features it's a good way to think about what you're building and come up with interesting gameplay ideas and prototypes, but this can be achievable also by preparing a basic system with all the useful functionalities already implemented and then building upon it. You just have to find out which mode fits you better.

That being said, since we decided we want to build a game state system to control the flow of the game, we have to create a global variable called game_state whose value can be one of the states defined by an enum representing the various states.

Space Gala's states flow is really simple, as you can see in Figure 5-7. When we start up the game, we can immediately start playing. We can pause the game by pressing the Esc key; and through that menu, we can restart the game, resume it, or quit it. We can access the restart and quit functions also by losing or winning the game.

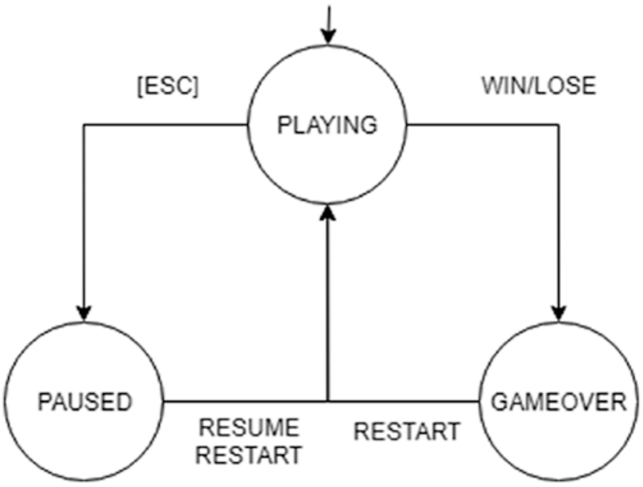


Figure 5-7. *Space Gala's states flow*

Double-click obj_controller to open up the Object Editor and select the Create event we already added before and modify it so that it looks like this:

```
enum states {
    playing,
    paused,
    gameover
};
```

```
6
7   global.game_state = states.playing;
8
9   randomize();
```

Lines 1–5: Here we define the state enum that represents all the various states our game can be in.

Line 7: When the game begins, the game is in playing state.

Line 9: As we did in the last section, we are initializing the random seed by calling randomize.

Now that we have our game state global variable, we can start using it to put the game in pause or to make the game end.

Create a new Key Pressed event for obj_controller by clicking *Add Event* ➤ *Key Pressed* ➤ *Other* ➤ *Esc* and add the following code:

```
if ( global.game_state == states.playing )
{
    global.game_state = states.paused;
    show_debug_message("PAUSED - " + string(global.game_
    state));
}
else if ( global.game_state == states.paused )
{
    global.game_state = states.playing;
    show_debug_message("PLAYING - " + string(global.game_
    state));
}
```

This code will change the state of the game from playing to paused and from paused to playing when we press the Esc key.

Now we have to change the game state also when our ship is destroyed, so that we know that the game ends.

Double-click obj_player and open up the Destroy event and add this one line at the bottom of the code:

```
global.game_state = states.gameover;
show_debug_message("GAMEOVER - " + string(global.game_
state));
```

Every time an instance of obj_player gets destroyed, the game state changes to states.gameover allowing us to understand that the game ended and the player shouldn't be able to play anymore.

Obviously, the game should also end when every alien gets killed by the player.

That's simpler than it seems! Just open up obj_controller by double-clicking it in the Resources sidebar and create a Step event by clicking Add Event ➤ Step ➤ Step and write this code in it:

```
if ( !instance_exists(obj_enemy_red) )
{
    global.game_state = states.gameover;
}
show_debug_message("STATE CODE = " + string(global.game_
state));
```

In the preceding code, we check if there is any instance of obj_enemy_red in the room; if there isn't, the game state is changed to states.gameover.

Doing this check in the Step event overrides the change of state we do when we press the Esc key, and that's just perfect.

Also, anytime the Step event is called, we show in a debug message (shown in GameMaker's console) the code of the current game state, so that we can verify the change of the state when we run the game.

Note By default, when we declare an enum, GameMaker assigns to each of its elements a value starting from 0 to n-1, with n being the number of the elements of the enum.

In our case

```
- states.playing = 0
- states.paused = 1
- states.gameover = 2
```

Executing the game by pressing F5, we can see that when we press the Esc key, GameMaker's console shows a text telling us whether the game is in pause or not adding the value of global.game_state (which is 0 in playing mode and 1 in pause mode). Also when we die or we kill every enemy in the room, we can see the change of the state to states.gameover.

Perfect! It's working! You can close the game and get rid of those show_debug_message lines; we don't need them anymore.

We have a working game state system. Now we just need to use it as a semaphore to properly regulate the flow of our game. For example, it would be great to implement a pause/resume functionality that allows both player's and enemies' spaceships to move and shoot only when the state is playing, and we also want the bullets to freeze when the state is not playing, so that we can put the game in pause or stop the action when the game is over. It would also be great to notify the player about the current game state by showing an appropriate text on the screen.

Go ahead and open up obj_player's Object Editor and modify the code inside the Key Down ➤ Left event so that it looks like this:

```
if ( global.game_state == states.playing and x > 0 +
sprite_width/2 )
{
    x -= spd;
}
```

We just added a condition to check whether the game is in the playing state; if it is, we move left the spaceship.

Now do the same for Key Down ➤ Right:

```
if ( global.game_state == states.playing and x < room_
width - sprite_width / 2 )
{
    x += spd;
}
```

And of course we have to do the same also for our shooting key, by modifying the Key Down ➤ Space event like this:

```
if ( global.game_state == states.playing and can_shoot )
{
    can_shoot = false;

    var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet_player);
    bullet.atk = atk;
    bullet.direction = point_direction(x, y, x, y-1);
    bullet.speed = 10;

    alarm[0] = shoot_delay;
}
```

Now, to be able to shoot, we don't need only to have can_shoot to be true. We also need that the game state is in playing mode.

This only blocks our capacity to move left and right and to shoot. Our bullets will still travel forward once shot, even if we pause the game. Let's fix this!

Thinking about what we did with the bullets, we are not changing their position by increasing/decreasing the Y-coordinate like we do with the enemies' and the player's spaceships. We make them move by changing their direction and changing their speed. That means that they travel at a fixed speed – so to make them stop, we just have to change speed value to 0.

Double-click obj_bullet_player and create a new Step event by clicking Add Event ➤ Step ➤ Step and add the following code:

```
if ( global.game_state == states.playing )
{
    speed = spd;
}
else
{
    speed = 0;
}
```

The Step event will now continuously check whether the game state is in playing state or not; and if it is, it will change the speed to spd; if it's not, it will change it to 0 making the bullet stop.

Actually, since we are now changing constantly the bullet's speed, we don't even need obj_player to do it anymore. Let's get rid of that line in obj_player's Key Down ➤ Space event and modify the code so that it looks like this:

```
if (can_shoot and global.game_state == states.playing )
{
    can_shoot = false;

    var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet_player);
    bullet.atk = atk;
    bullet.direction = point_direction(x, y, x, y-1);

    alarm[0] = shoot_delay;
}
```

We deleted the line that modified the speed of the bullet from there, since we are doing it from obj_bullet_player's step event.

Running the game (pressing F5 or clicking the Run button in the toolbar), you will now notice that the player's spaceship and its bullets will freeze where they stand when you pause the game, but the enemies and their bullets won't!

Well, not bad! We achieved what we wanted, and we made sure it's working. Now we need to apply that same mechanism to the enemies and their bullets.

Double-click obj_enemy_red's and head to its Alarm 1 event and get rid of the line that modifies the speed of the bullet, so that it looks like this:

```
if ( global.game_state == states.playing )
{
    var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet_enemy);
    bullet.atk = atk;
    bullet.direction = point_direction(x, y, x, y+1);
}
alarm[1] = room_speed * random_range(0.5, 5);
```

Now double-click obj_bullet_enemy in the Resources sidebar and substitute the content of the Step event with this code:

```
if ( global.game_state == states.playing )
{
    speed = spd;
}
else
{
    speed = 0;
}
```

Well! Looks like everything is in order! Let's double-check by running the game!

The player and the enemies can shoot at each other while the game is not in pause (Figure 5-8), and they freeze when it is. Great!

Welp! That was a long run, but we didn't finish yet! We need to take care of two last things: how to open up the menu and drawing info in the HUD.

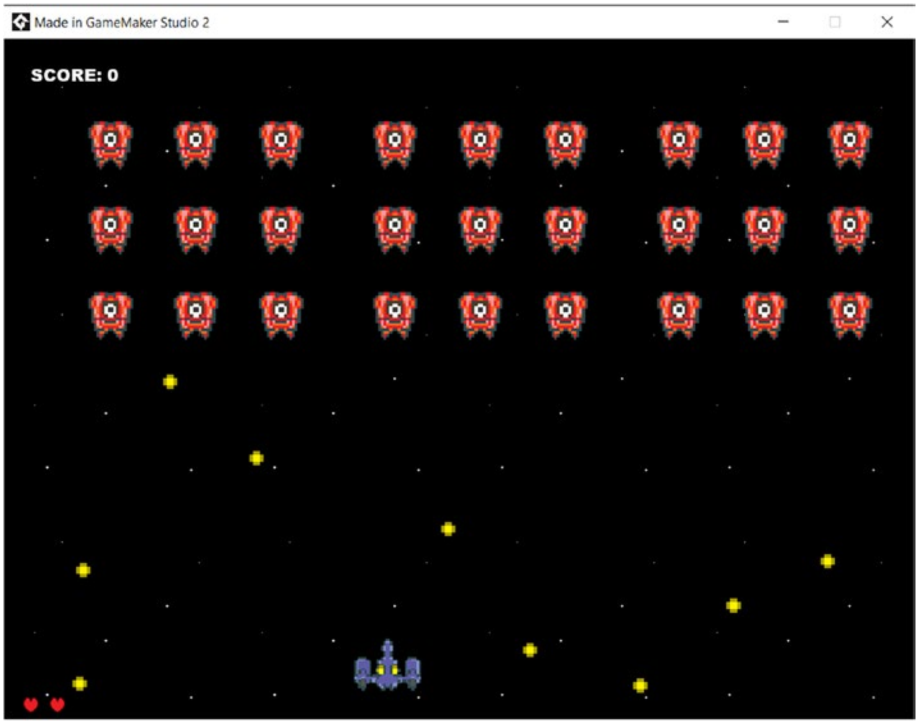


Figure 5-8. *Shooting implemented for both the player and the enemies*

Making HUDs

Our HUD will be very simple. It will consist only in a couple of texts drawn on screen showing the player's score and the status of the game (whether paused or not) and some icons to show the player's HP.

We will do this – as always – using obj_controller. So let's open it up by double-clicking it in the Resources sidebar and create a new Draw GUI event by clicking Add Event ➤ Draw ➤ Draw GUI.

In the Draw GUI event, add the following code:

```
1   draw_set_font(fnt_score);
2   draw_set_color(c_white);
3   draw_text(30, 30, "SCORE: " + string(score));
4
5   switch ( global.game_state )
6   {
7       case states.paused:
8           draw_text( 900, 30, "PAUSE" );
9           break;
10      case states.gameover:
11          draw_text( 850, 30, "GAME OVER" );
12          break;
13  }
14
15  if ( instance_exists(obj_player) )
16  {
17      var xhp = 30;
18      repeat( obj_player.hp )
19      {
20          draw_sprite( spr_life, 0, xhp, 750 );
21          xhp += 30;
22      }
23  }
```

Lines 1–3: As usual, we set up the game to draw a text showing our current score represented by the score built-in variable. We never change this value in the game yet, so it will remain at 0 for all the time, but we are going to fix this very soon!

Lines 5–13: In this part, we check the current state of the game; and if we are in a pause or game over state, we draw the right text on the screen to let the player know.

Line 15: It checks if an instance of obj_player exists. We need this check because we are accessing a variable belonging to obj_player. If we access a variable of an object without doing this check, there is the risk that our game can crash. In fact, accessing a variable of a nonexisting instance – for example, an instance that has been recently destroyed – will cause a fatal error in the game and a crash.

Lines 16–22: Here we are drawing a spr_life sprite on screen as many times as many HPs we have. We draw them at 30 pixels intervals from one another at the bottom of the screen. We are using our local variable xhp as the X-coordinate so that we can easily increase that value by 30 pixels without using fixed coordinates for every single icon drawn.

Now that we are dealing with the score, we should increase it every time we destroy an alien spaceship. Let's do this by editing the obj_enemy_red Destroy event and adding this line in it:

```
score += 100;
```

Now, when we destroy an enemy ship, our score will be increased by 100.

That's it! Run the game to check that everything is working!

Running the game, you may see that the player's HPs are shown as hearts in the bottom-left corner of the screen and the score is visible in the top-left corner of the game screen. Also, when you press the Esc key to pause the game, everything freezes, and the text PAUSE is shown in the top-right corner of the screen.

Great! That looks a lot more like a game!

What about victory?

According to the GDD, the player wins the game when they destroy every alien spaceship in the room and loses it when they are killed or the enemies reach the bottom of the room – meaning they reach the base.

We already added the code to change the state when we die or all the enemies are wiped out. We only need to change the state also when they reach the bottom of the room. Nothing simpler!

Open up obj_controller and head to the Step event and change it so it looks like this:

```
1   if ( !instance_exists(obj_enemy_red) )
2   {
3       global.game_state = states.gameover;
4   }
5   else
6   {
7       for ( var i = 0; i < instance_number(obj_enemy_red); i++ )
8       {
9           var enemy = instance_find(obj_enemy_red, i);
10          if ( enemy.y >= room_height )
11          {
12              global.game_state = states.gameover;
13          }
14      }
15  }
```

Lines 5–15: If there's still at least one instance of obj_enemy_red, we cycle through all the instances of obj_enemy_red in the room and check if at least one of them has the Y-coordinate greater than or equal to the height of the room; if exists such an instance, we change the game state to states.gameover.

Note instance_number is a built-in function that counts how many instances of an object exist in the current room and returns that number.

For example:

```
var monsters_number = instance_number(obj_monster)
```

The preceding code counts how many instances of obj_monster there are in the current room and returns the value into monsters_number.

instance_find(obj, i) is a built-in function that searches for the i-th instance of the object obj looping through all the instances in the current room.

For example:

```
var second_monster = instance_find(obj_monster, 2);
```

Returns into the second_monster variable, the id of the second instance of obj_monster after searching all the instances in the room.

Running the game by pressing F5, you will see that everything is working as expected. The game over is triggered when we kill every alien, when we die, or when an alien reaches the bottom of the room. Great! But what if we want to play again? Or quit the game? Here comes the menu!

Menu

We will create a very simple menu made of texts that we can navigate with the up and down arrow keys.

Our menu should offer the functionalities to resume the paused game, restart the game, and quit it.

Double-click obj_controller in the Resources sidebar to open up the Object Editor and add these three lines of code at the bottom:

```
options = [ "RESUME", "RESTART", "QUIT" ];
opt_number = array_length_1d(options);
menu_min = 0;
menu_index = 0;
```

In the preceding code, we create an array called options that contains the various labels for our menu options, then we store the size of the array in the opt_number variable, and we define menu_index that we will use to keep track of our position in the menu and menu_min that represents the first element of the menu. We need menu_min because we want to show RESUME as an option when we press the Esc key while playing, but we don't need it when we are showing the menu after the game is over, because there is not a game to resume. So, when pausing during the game, menu_min will be 0; but when the game is over, menu_min will be 1 – so that we can avoid the RESUME option.

The logic to move the cursor in the menu will be part of the Step event. We want the game to allow us to move in the menu only if the game is in pause or the game is over – meaning when the game is not in the playing state. If that's the case, we want to change the value of menu_index by pressing the up and down arrow keys.

Open up the code related to obj_controller's Step event and add this code at the bottom:

```
if ( global.game_state != states.playing )
{
    if ( global.game_state == states.paused )
    {
        menu_min = 0;
    }
    else
8          {
9              menu_min = 1;
10         }
11
12         var move = keyboard_check_pressed(vk_down) - keyboard_
           check_pressed(vk_up);
13         menu_index += menu_move;
14         if ( menu_index < menu_min )
15         {
16             menu_index = opt_number - 1;
17         }
18         else if ( menu_index > opt_number - 1 )
19         {
20             menu_index = menu_min;
21         }
22     }
```

Line 1: This code is executed only if the game is paused.

Lines 3–10: Here we change the value of menu_min to 0 or 1 (getting rid of the RESUME option in the second case) according to the value of the game state global variable.

Line 12–13: move represents the movement we make in the menu by pressing the up or down key. keyboard_check_pressed(k) returns 1 when the key k is pressed. So if we both press the up and down arrow keys, the value of move is 1 - 1 = 0; if we press only the down arrow key, its value is 1; and if we press only the up arrow key, it is -1. We can directly sum this value to menu_index (which we do in line 4) to properly change our position in the menu according to the array options.

Lines 14–21: This is pretty straightforward. We check whether we are going out of bounds and, if we do, fix it. If we press up when we are at the topmost option in the menu, we are brought to the last one and vice versa. It's very important to check the bounds of your arrays, because not doing this means your game will crash when you try to access an array element that does not exist.

Perfect! We only have to draw that on screen, so that we can see it!

To achieve this, we can just add some lines of code in the Step event's game state check, so that when the game is in pause, we can draw our menu on the screen.

Open up obj_controller's Draw GUI event and modify the code so that it looks like this:

```
draw_set_font(fnt_messages);
draw_set_color(c_white);
draw_text(30, 30, "SCORE: " + string(score))

if ( global.game_state != states.playing )
{
    if ( global.game_state == states.paused )
    {
        draw_text( 900, 30, "PAUSE" );
    }
    else
    {
        draw_text( 859, 30, "GAME OVER" );
    }

    for( var i = menu_min; i < opt_number; i++ )
    {
        if ( menu_index == i )
        {
            draw_set_color(c_white);
        }
        else
        {
```

```
24              draw_set_color(c_dkgray);
25          }
26          draw_text( 850, 600 + 30 * i, options[i] );
27      }
28  }
29
30  if ( instance_exists( obj_player ) )
31  {
32      var xhp = 30;
33      repeat(obj_player.hp)
34      {
35          draw_sprite( spr_life, 0, xhp, 750 );
36          xhp += 30;
37      }
38  }
```

We totally revolutionized this code. Now we are not checking anymore only if the game state is set to paused or game over, we are firstly checking if it's not in playing mode (line 5), then we check whether the game is paused or is over, and we draw the right text in the top-right corner of the screen. At lines 16–27, we loop between the menu options, and we draw them one by one in the bottom-right corner of the screen white-coloring the option with the same index of menu_index – that means that the option we are currently pointing is the white one, while the others are grayed out. This helps the player to quickly see the selected item which is the brighter one in the list.

The last thing remaining is to actually do something when we select an option from the menu. We do this by creating an event triggered when we press the enter key. We will check if we are in pause mode and where are we in the menu, so that we can execute the right action.

Click Add Event ➤ Key Pressed ➤ Enter and add this code to it:

```
1   if ( global.game_state != states.playing )
2   {
3       switch( menu_index )
4       {
5           case 0:
6               global.game_state = states.playing;
7               break;
8           case 1:
9               game_restart();
10              break;
11          case 2:
12              game_end();
13              break;
14      }
15  }
```

In the preceding code, we check if we are in a status different from states.playing; if that's the case, we check the value of menu_index and execute the right action according to it.

Lines 5–7: If menu_index equals 0, we know that we are pointing to the first element of the array, which is Resume game. In that case, all we want is to go back to the game, and we only need to reset global.game_state to states.playing to do that.

Lines 8–10: If menu_index equals 1, we are pointing to Restart game. To restart the game, the only thing we need to do is to call the game_restart() function. It will reinitialize everything and restart the game from the beginning.

Lines 11–13: If menu_index equals 2, we are pointing to Quit game. To close the game, we use game_end() built-in function that closes the game application.

Done! Let's check that everything works by pressing the F5 key.

Running the game, you can verify that pressing the Esc key you can access the full version of the menu with the RESUME, RESTART, and QUIT options and check that they work properly (Figure 5-9). When you both win or lose the game, you can access a menu with only the RESTART and QUIT options – just what we wanted!

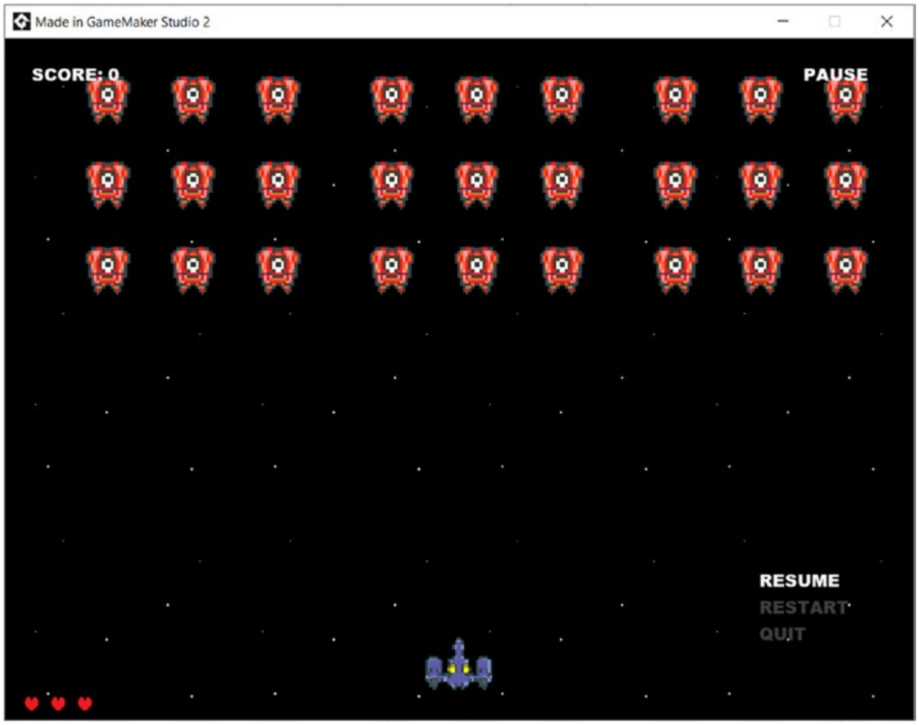


Figure 5-9. *The complete Space Gala game*

Whoah! That was a long one! But we did it! We made a complete shoot 'em up game from scratch! Can you believe it?

Enjoy the game and play with it trying to add new features! In the exercise section, you can find some interesting coding challenge that you can use to add features to the game you just created.

In the next chapter, we will extend this version of the game we just created to make it look more like a 1990s STG. We will convert the game into a scrolling shooter and add more enemies, and we will also create our first boss fight! But, coolest of all, we will take inspiration from the masterpiece of the genre, Ikaruga; and we will design and implement its iconic polarization system.

Fasten your seatbelt, we're getting deeper into the outer space!

TEST YOUR KNOWLEDGE!

1. What is a pivot point?
2. What is a collision mask?
3. What happens when the collision masks of two instances collide?
4. Can you tell the difference between an Automatic Rectangle and a Full Image Rectangle collision mask?
5. Do you think it's good to always use precise masks for your sprites?
6. How does the built-in variable speed work?
7. How does the built-in variable direction work?
8. To what value should you set the direction built-in variable to make an instance face left?
9. Can you use another way to modify the value of direction?
10. How does point_direction work?
11. How does destroy_instance(inst, val) work? What is the default action when you call it without argument?
12. How can you create particle effects in GameMaker?

13. How can you count how many instances of an object are present in the room?

14. How does instance_find work?

15. Currently, the player loses the game when the enemies reach the bottom of the room. This can feel unfair to the player, since there is a moment in which they cannot fight back the enemies if not crashing into them. The limit should be raised to make the game feel less weird.

    a. At which coordinate on the Y-axis do you think the game should trigger the game over?

    b. Can you modify the game to change this feature?

16. Play some classics of the shmup genre (Space Invaders, Galaga, Centipede, 1940, etc.) and write down the best features of any one of them.

    a. Compare how these games play to how Space Gala plays. What do you think can be improved in Space Gala, after doing this comparison?

    b. Can you implement one or more of those changes?

CHAPTER 6

Shoot ‘Em Up!

In the previous chapter, we began our journey into the world of shooter games starting from the classics of the fixed-shooting genre, making a game inspired by Space Invaders and Galaga.

In this chapter, we will move toward the classics of the 1990s, like R-Type and especially Ikaruga.

We will adapt the code of Space Gala so that it can implement some new features, like scrolling camera, enemies moving on patterns, enemies tracking the player and aiming while shooting, eight-direction movement, and boss fights.

Moreover, we will borrow an interesting idea from one of the most important shoot ‘em ups ever made: Ikaruga. We will create a gameplay mechanic inspired by Ikaruga’s polarization system.

Ikaruga had this innovative gameplay element that allowed the player to change the color of its spaceship from white to black to match the color of the enemies’ bullets. In fact, in Ikaruga, your ship wouldn’t get damage if hit by a bullet of its same color. Instead, the ship would absorb the bullet to charge an attack bar for its secondary attack. This is a very original gameplay that made Ikaruga a recognizable and unique game that even now is acknowledged as one of the best shoot ‘em ups around. Also, when destroying three enemy ships of the same color, the player would get bonus points.

We will borrow this color-switching idea to implement an additional gameplay element to make the experience more fun. The player will be able to switch between red and blue, and also their bullets will change

© Sebastiano M. Cossu 2019
S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_6

color. When the player gets hit by a bullet of the same color, it will charge a bar that once full will allow the player to launch a super strong attack that kills every visible enemy. Also, when shooting enemies with bullets of the same color, the player will score bonus points.

This gameplay mechanics motivates the player to get hit and master the color-switching function, but it will also raise the risk to be damaged, and so the challenge is raised too! This will make the game more fun and enjoyable.

There's a lot of stuff to cover indeed, but first let's spend some words on the design and gameplay considerations, so that we can better face the coding.

Fixed vs. scrolling shoot 'em up!

The first big difference between fixed and scrolling shooters is, of course, the scrolling feature. In a scrolling shooter, the level advances at a constant speed, and the player is forced to dodge and attack at a sustained pace.

Traditionally, scrolling shooters were just vertically scrolling all the levels until they reached the end. Going forward, while the genre evolved, level design became more complex; and some scrolling shooters (and not only them) became shooters on rails, where the camera followed some nonlinear path throughout a complex level. There are some interesting examples of level design, in the scrolling shooter panorama, which are very interesting to explore (e.g., R-Type or Ikaruga); but in this book, we will just cover vertical scrolling shooters, which are the most important and iconic.

Since we are going to make a lot of changes to our game, it's appropriate that we update Space Gala's game design document. It's very important to regularly update the GDD with new features and modifications so that it reflects and tracks the changes in the game.

The first thing we need to change is the genre. Space Gala is not just a fixed shooter anymore, but it's a hybrid shoot 'em up with every level implementing a different game mechanic.

Then, it's important to talk about the new level that we are going to include in the game.

Let's edit the GDD so that it includes the new features we want to implement. We will use it as a guide to follow the upgrade of Space Gala.

Tip Remember to always keep your game design document updated.

Every change you make on the project should be registered in the GDD to keep the whole team on the same page.

Space Gala v.2.0 (GDD)

Space Gala is a single-player hybrid shoot 'em up game – meaning that depending on the level, it may be a fixed shooter or scrolling shooter.

The objective of the player is to eliminate all the aliens shooting at them while trying to survive dodging all their attacks.

Story and setting

You are colonel Jonathan Spacepants, and you are the last hope for mankind. Your mission is to destroy the alien fleet before they reach our space station. After that, you shall advance and hunt the remaining aliens and kill their leader.

Gameplay

Space Gala revolves around dodging and shooting. It's very important to keep the player focused on those two activities maintaining a fast pace but also giving the player a good amount of satisfaction and motivation.

Satisfaction should be reached by making smooth controls and a nice-to-use weapon.

Motivation can be reached by giving the player a sense of progression. Progression can be achievable by increasing the difficulty of the game from level to level.

Color-switching

The game implements a color-switching system similar to Ikaruga's polarization system to enhance the gameplay and offer an additional challenge.

The player can change the color of their ship from red to blue and vice versa using the X key.

Color-switching allows the player to get hit by bullets of the same color to recharge the super-attack, **X-bomb**, a bomb that will wipe out every enemy in the visible area.

Color-switching is also useful to raise the player's score. In fact, killing an enemy with a bullet of the same color will raise the score by 100% and double the damage inflicted.

X-bomb charge

The player can charge the X-bomb by being hit by bullets of the same color. Once the bar is full, pressing the spacebar, the X-bomb will be released.

The X-bomb will kill every visible enemy.

Victory conditions

The game can be won by eliminating all the enemies in level 1 or by killing the final boss in level 2.

You can lose either by dying or failing the level's mission.

Controls

The player can control the spaceship by using the **arrow keys** to move and the **Z** key to attack and **X** key to activate polarization.

It's very important, for the genre, to have precise and smooth controls. We don't want to add any friction in the player's movements.

Right Arrow: Move right.

Left Arrow: Move left.

Up Arrow: Move up.

Down Arrow: Move down.

Z: Attack – a single bullet dealing standard damage.

X: Switch color.

Spacebar: Release X-bomb.

Esc: Open/close menu.

Menu

You can open/close the menu by using the Esc key. Via the menu, you can close the game, restart it, or resume the paused game.

A smaller version of the menu should be shown when the game is over to allow the player to restart or close the game.

Pacing

The sense of urgency should be the preponderant feeling in Space Gala. You need to wipe out a fleet of aliens without being killed. The aliens are continuously moving, and you need to be a fast and precise shooter to deal with them quickly.

The introduction of the color-switch system will make the pace even faster.

Enemies

There are three types of enemies:

- **Reds/Blues**: Basic enemies that move left and right and advance while randomly shooting – they are present only in level 1.
    - HP (Health Points): 1.
    - ATK (How much damage they inflict): 1.
    - Movements: They move left and right and regularly jump down by X pixels.
- **Red/Blue Walkers**: Advanced one-eyed enemies. They follow complex paths around the map and attack randomly the player.
    - HP: 2.
    - ATK: 1.
    - Movements: They follow complex paths around the level.
- **Red/Blue UFOs**: Turret-like enemies. They continuously aim at the player and regularly shoot them.
    - HP: 2.
    - ATK: 1.
    - Movements: They remain still and track the player's movements shooting at them.

Game modes

There is just one arcade game mode. The player must kill all the enemies to win the game.

Level 1

Level 1 is pretty simple. The player has to face a fleet of nasty red aliens that want to approach the space station.

The aliens will dodge the bullets by continuously moving left and right while they recharge their FTL engines to jump toward the space station. They can jump no more than 30 pixels forward, and they need to wait approximately 5 seconds before the FTL engine recharges and they can jump again.

The aliens will shoot randomly in front of them (easy to dodge).

The level is made of one group of 16 aliens.

Level 2

Level 2 is a classic vertical scrolling level. The player travels all the length of the level dodging/absorbing bullets and dodging/killing the enemies until they reach the final boss.

The level is filled with three types of enemies: reds/blues, red/blue walkers, and red/blue UFOs.

Similar games and influences

Space Gala is obviously inspired by Space Invaders, Galaga, and Ikaruga.

The gameplay of level 1 is more like the Space Invaders experience, with the aliens descending gradually and sporadically shooting, while level 2 resembles more like Ikaruga.

Other notable games of the same genre are Centipede, Galaxian, and Moon Cresta.

Target audience

The audience is not very wide, but it's super passionate and cares a lot about the purity of the genre.

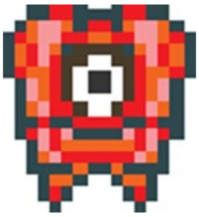
Assets

All those changes come with some additions to the assets.

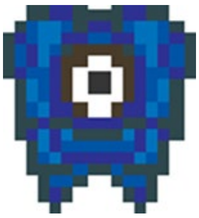
Other than spr_enemy_red, we now want to add a collection of red and blue enemies so that we can implement the color-switching system and a couple more types of enemies.

We also need an original sprite for the boss and two recoloring of the bullet sprite.

spr_enemy_red



Pivot Point: Middle-center
Collision Mask: Automatic, rectangle
Size: 50 × 57
spr_enemy_blue



Pivot Point: Middle-center

Collision Mask: Automatic, rectangle

Size: 50 × 57

spr_enemy_ufo_red

Pivot Point: Middle-center

Collision Mask: Automatic, rectangle

Size: 64 × 64

spr_enemy_ufo_blue

Pivot Point: Middle-center

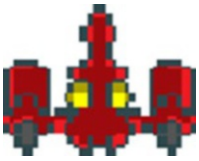
Collision Mask: Automatic, rectangle

Size: 64 × 64

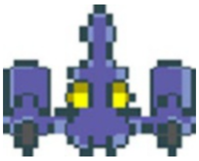
spr_bullet_red

Pivot Point: Middle-center
Collision Mask: Automatic, rectangle
Size: 16 × 16
spr_bullet_blue

Pivot Point: Middle-center
Collision Mask: Automatic, rectangle
Size: 16 × 16
spr_player_red



Pivot Point: Middle-center
Collision Mask: Automatic, precise
Size: 80 × 69
spr_player_blue



Pivot Point: Middle-center
Collision Mask: Automatic, precise

Size: 80 × 69

spr_boss

Pivot Point: Middle-center

Collision Mask: Automatic, precise

Size: 120 × 135

Sounds

Sounds are very important in a game. They can change the game's pace and make everything more enjoyable. Sounds can make a situation more realistic or atmospheric. We are creating a great game and adding a lot of nice features, but it's totally silent. In this chapter, you will learn how to add sounds to the game.

To create a new sound effect to use in your game, right-click Sounds in the Resources sidebar and select Create Sound, name it, and then click the label *No Sound* to select a new audio file from your computer.

You can play sounds in your game by using the audio_play_sound function; its signature is

```
audio_play_sound(soundid, priority, loops)
```

where

- *soundid* is the name of the sound asset that you want to play.
- *priority* is the channel priority of the sound.
- *loop* is a Boolean that tells GameMaker if the sound should be played in loop or not.

To stop playing a sound that you set to loop, you can use the function audio_stop_sound. Its signature is

```
audio_stop_sound( index )
```

where *index* is the name of the sound you want to stop.

Note You should use audio_play_sound and audio_stop_sound both for sound effects and to play soundtracks. The only difference between the two, from GMS2's point of view, is that sound effects are played once, while soundtracks are looped.

For this version of the game, we will introduce sound effects. As usual, you can use my sounds (that you can find in the web site of this book), or you can make them yourself.

We will need these sound assets:

snd_menu: This will be played when moving the cursor in the menu.

snd_shoot: This sound effect will be played every time the player shoots.

snd_damage: This will be played when the player gets hit or the enemy explodes.

snd_color-switch: This is the sound effect related to the new color-switch feature.

snd_esc: This sound effect will be used to open/close the menu.

snd_gameover: This one will be played when the player dies.

This will also be the chapter where we introduce cameras and views, so that we can decide which part of a room to show. In Space Gala, we are going to use cameras to manage the vertical scrolling of the game, so that we can let the player explore the entirety of the new level 2 and travel for all its length toward the final boss.

Cameras and viewports

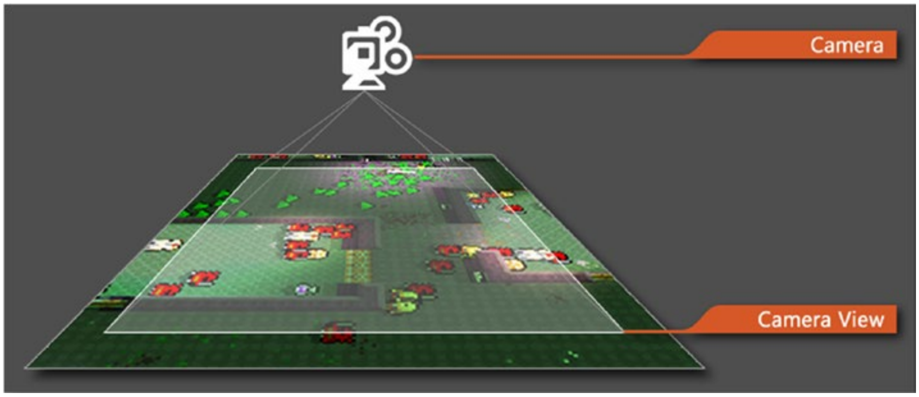


Figure 6-1. *Here's a visual explanation of the difference between Camera and Camera View*

By default, rooms in GameMaker are shown entirely into the viewport. A viewport is basically a window in your game world. You can decide which portion of the room you want to show to the player using cameras.

Cameras (Figure 6-1), in GMS2, allow you to decide which portion of the room you want to show to the player. Like real cameras, you can change their angle, move them, and use them for a lot of tasks, for example, draw HUD elements (like Minimaps or zoom-in/zoom-out sections) or create split screens or Cutscenes.

When you create a camera, you can define the view, which is the area that is visible from the camera.

Space Gala's level 2 will be a way bigger room than the one in level 1, and we want to travel through its entirety using a scrolling camera, so we need to activate viewports and cameras.

Create a new level by right-clicking Rooms in the Resources sidebar and selecting Create Room. Call the new room rm_level_2 and modify its property so that it has a width of 1024 and a height of 10000 pixels.

In the properties section of the Room Editor, tick *Enable Viewports* and *Clear Viewport Background*. Below, in Camera Properties, edit the fields so that the values are as follows:

- X Pos = 0
- Y Pos = 9230
- Width = 1024
- Height = 768

In Viewport Properties, change the values in the fields like this:

- X Pos = 0
- Y Pos = 0
- Width = 1024
- Height = 768

We just set the viewport and the camera. Now we can use them to show our game world. The camera will be placed at coordinates 0,9230 – meaning the bottom of the very long room we created.

We want the camera to scroll vertically from the bottom to the top of the level. To do that, we will create a camera object. This object will regulate and check all the things related to the camera management and will travel from the bottom of the room to its top, while the real camera will follow this object. Confused? Well, it's really easier to show it than explain it.

Create a new object by right-clicking in the object section of the Resources sidebar and selecting Create Object.

Call the new object obj_camera and add a new Create event by clicking Add Event ➤ Create in the Object Editor. Add this code in it:

```
1   cam = view_camera[0];
2   x = room_width /2;
3   direction = point_direction(x, y, x, y-1);
4   spd = 2;
```

Line 1: We assign the camera to a variable named cam. We are doing this to make it simple to manage the camera.

Line 2: We move obj_camera to the center of the room.

Lines 3–4: We set the direction of obj_camera to face up, and then we set the spd variable to 2. We will use this variable later to update the speed of the object, so that it will start moving up.

Note view_camera is an array that contains the eight active cameras that you can have in your game.

In fact, you can have up to eight views in your game; and for each of those views, you can only have one active camera. This means that you can only have eight active cameras at any time.

view_camera contains the cameras associated to the eight views of the game like this:

```
view_camera[0] contains the id of the active camera
for viewport 0

view_camera[1] contains the id of the active camera
for viewport 1

...

view_camera[7] contains the id of the active camera
for viewport 7
```

We need to stop camera scrolling every time the game is not in playing state and to resume when it goes back to playing mode.

To do that, create a Step event for obj_camera by clicking Add Event ➤ Step ➤ Step and put this code in it:

```
if ( global.game_state == states.playing )
{
    if ( instance_exists(obj_player) )
    {
        var cam_x = camera_get_view_x(cam);
        var cam_y = camera_get_view_y(cam);
        var cam_w = camera_get_view_w(cam);
        var cam_h = camera_get_view_h(cam);

        if ( obj_player.x - obj_player.sprite_width/2 <= cam_x )
        {
            obj_player.x = cam_x + obj_player.sprite_width/2;
        }
        if ( obj_player.x + obj_player.sprite_width/2 >= cam_x + cam_w )
        {
            obj_player.x = cam_x + cam_w - obj_player.sprite_width/2;
        }
        if ( obj_player.y + obj_player.sprite_height/2 >= cam_y + cam_h )
        {
            obj_player.y = cam_y + cam_h - obj_player.sprite_height/2;
        }
22          if( obj_player.y - obj_player.sprite_height/2 <=
            cam_y )
23          {
24              obj_player.y = cam_y + obj_player.sprite_
                height/2;
25          }
26
27          speed = spd;
28          obj_player.speed = spd;
29      }
30  }
31  else
32  {
33      if ( instance_exists(obj_player) )
34      {
35          speed = 0;
36          obj_player.speed = 0;
37      }
38  }
```

The purpose of the preceding code is to check if the game is paused or not. If it is, the speed of the object and the player ship gets zeroed (lines 35–36); if it's not, the speed of both the camera and the player is reset to the value of the spd variable (lines 27–29). Also, when the game is not in pause, we constantly check if the ship is inside the camera view coordinates; if it's not, we put them back in. This creates some sort of bounds, so that the player cannot escape the game screen.

We need to change also the speed of the player's spaceship to the same velocity value because otherwise, eventually the player's spaceship will be trained by the screen (since it cannot go out of bounds) and so the player movements will be influenced by some sort of friction that we must absolutely avoid.

Ok, it's time to test that everything is in the right place! But before we do, we should make some modifications to both obj_player and obj_controller.

In level 1, obj_player was prevented from exiting the bounds of the screen with a simple check on the width of the room. This is not appropriate anymore, since now that we have a camera, we want the player to rely on the camera's coordinates, so that we are free to move and tilt the camera and change the room's dimensions without messing with this feature. To do that, we will change this control to the default camera view coordinates (that in level 1 coincide with the room itself, so it's safe).

Open up obj_player and select the Key Down ➤ Right event and change its code so that it looks like this:

```
var cam = view_camera[0];
var cam_w = camera_get_view_width(cam);
if ( global.game_state == states.playing and x + sprite_
width/2 < cam_w )
{
    x += spd;
}
```

This will check whether the player exited the bounds of the camera and in that case prevents the player from moving the spaceship.

Do the same also for Key Down ➤ Left event:

```
var cam = view_camera[0];
var cam_x = camera_get_view_x(cam);
if ( global.game_state == states.playing and x - sprite_
width/2 > cam_x )
```

Now we want to add the possibility to move up and down, but only when we are in level 2, since level 1 is a fixed shooter level. To do that, we will use the built-in variable *room* that tells us the name of the room.

In the **obj_player**'s Object Editor, click *Add Event* ➤ *Key Down* ➤ *Up* and write this code in it:

```
if ( room != rm_level_1 and global.game_state == states.
playing )
{
    y -= spd;
}
```

The preceding code allows the use of a key to move up, only if the player is not in level 1.

Let's do the same for the down arrow key clicking Add Event ➤ Key Down ➤ Down and adding this code to the newly created event:

```
if ( room != rm_level_1 and global.game_state == states.
playing )
{
    y += spd;
}
```

Thinking about boundaries, right now the enemies shoot at the player no matter where they are in the room. This is not optimal, because it means that even if the player can't see the enemy, the enemy will shoot at them. This is unpredictable and unmanageable for the player, meaning not fun. Every time a feature in our game risks to be or become not fun, we have to stop and rethink. In this case, we can fix this issue by letting the enemies shoot only when they are in the camera boundaries.

Open up obj_enemy_red's Alarm 0 event and modify it to do these checks like in the following code:

```
var cam = view_camera[0];
var cam_x = camera_get_view_x(cam);
var cam_y = camera_get_view_y(cam);
var cam_w = camera_get_view_width(cam);
```

```
5   var cam_h = camera_get_view_height(cam);
6
7   if ( global.game_state == states.playing )
8   {
9       if ( x > cam_x and x < (cam_x + cam_w) and y > cam_y
        and y < (cam_y + cam_h) )
10      {
11          var bullet = instance_create_layer(x, y,
            "Instances", obj_bullet_enemy);
12          var bullet.atk = atk;
13          var bullet.direction = direction;
14      }
15  }
16  alarm[0] = room_speed * random_range(0.5, 5);
```

Lines 1–5: We calculate the camera's coordinates, like we did before.

Line 9: Here we check that the enemy is within the camera visible surface; we only want to execute that code if it is.

There is a similar issue with the player's bullets. In fact, right now the player can shoot, and the bullet will travel through all the map hitting enemies out of sight. This is not good, because the player can manage to clear the area killing all the enemies while they can't even shoot (because they're out of the screen, as we just defined in the previous code block).

What we want to do is to make sure that when the bullet gets out of the camera coordinates, it gets destroyed. We can implement this by using the point_in_rectangle function.

Note point_in_rectangle(px, py, x1, y1, x2, y2) is a built-in function that allows you to check whether a given point px,py falls in the rectangular area defined by x1,y1 and x2,y2.

We can use this function, for example, to check if the current instance is inside the camera's bounds like this:

```
point_in_rectangle( x, y, cam_x, cam_y,
cam_x+cam_w, cam_y+cam_h)
```

To implement the check on the bullet's position, head to obj_bullet_player's Step event and append the following code:

```
1   var cam = view_camera[0];
2   var cam_x = camera_get_view_x(cam);
3   var cam_y = camera_get_view_y(cam);
4   var cam_w = camera_get_view_width(cam);
5   var cam_h = camera_get_view_height(cam);
6   if ( not point_in_rectangle(x, y, cam_x, cam_y, cam_x +
    cam_w, cam_y + cam_h) )
7   {
8       instance_destroy(id, false);
9   }
```

Lines 1–5: As we already did in other situations, here we define the camera variables, so that we can access its properties.

Lines 6–9: In these lines, we use point_in_rectangle to check whether the current instance (the bullet) is inside the camera's boundaries (defined in lines 1–5). If the bullet gets out of bounds, we destroy the instance not triggering the Destroy event, because we don't want to play the particle effect when this happens (playing it would give the player a wrong feedback on what's happening in the game).

We have to make some modifications also to obj_controller. In fact, right now, victory conditions of level 1 are applied also to level 2; and that's a problem, since level 2 has different rules and gameplay mechanics. The idea is that you win level 1 by killing all the enemies and level 2 by reaching the end of the level and beating the final boss. So we must divide the two things.

Open up obj_controller's Step event and change the code so that it looks like this:

```
if ( global.game_state != states.playing ) // menu managing
{
    if (global.game_state == states.paused)
    {
        menu_min = 0;
    }
    else
    {
        menu_min = 1;
    }

    var move = keyboard_check_pressed(vk_down) - keyboard_
    check_pressed(vk_up);
    menu_index += move;
    if ( menu_index < menu_min )
    {
        menu_index = opt_number - 1;
    }
    else if ( menu_index > opt_number - 1 )
    {
        menu_index = menu_min;
    }
}

if ( room == rm_level_1 ) // check victory condition for
level 1
{
    if ( !instance_exists(obj_enemy_red) )
27      {
28          global.game_state = states.gameover;
29      }
30      else
31      {
32          for ( var i = 0; i < instance_number(obj_enemy_
            red); i++ )
33          {
34              var enemy = instance_find(obj_enemy_red, i);
35              if ( enemy.y >= room_height )
36              {
37                  global.game_state = states.gameover;
38              }
39          }
40      }
41
42      if ( global.game_state == states.gameover and
        !instance_exists(obj_enemy_red) )
43      {
44          if (room_exists(room_next(room)))
45          {
46              room_goto_next();
47          }
48      }
49  }
```

Lines 1–22: This is our old code to manage the menu. It's still valid.

Lines 24–49: We moved the checks of victory and game over into the if that checks if we are in level 1 or not. In lines 44–47, we also added a check to control if we actually won (we killed every obj_enemy_red); and in that case we warp to the next room, if it exists.

Everything is in place to do our first test for the scrolling feature! We only need to drag the objects inside the room. Open up rm_level_2 and put obj_camera and obj_player at the bottom of the room by dragging them from the Resources sidebar and dropping them in the room. You must also drop obj_controller in there, but please note that while you can place obj_controller anywhere in the room, you don't want to do the same with obj_player and obj_camera. You should put them where you want your game to start since there will be the starting position of the game.

Run the game by pressing F5, and you would notice that the camera correctly scrolls vertically followed by the player's spaceship that can now move in the eight directions. Great! Let's go on!

Designing color-switching

We should now design and implement color-switching, our new gameplay mechanic inspired by Ikaruga.

Color-switching can be activated by pressing the X key that will change the color of the ship from blue to red.

The ship will now shoot bullets of its same color, and hitting an enemy with a bullet of the same color will deal double damage.

When the player gets hit by a bullet of the same color, they won't get damage at all.

To manage color switching, we will use a color variable, so that we can tell whether an instance of an object is blue or red.

We will implement the various colors using enums, to enhance readability.

So let's add this code to obj_controller's Create event:

```
enum colors {
    none,
    red,
    blue
};
```

Now we can use this enum to assign a color to obj_player and change it when the player presses the X button.

In obj_player’s Create event, add this line:

```
1   sprite_index = spr_player_blue;
2   if ( room == rm_level_1 )
3   {
4       color = colors.none;
5   }
6   else
7   {
8           color = colors.blue;
9   }
```

Line 1: We change obj_player’s sprite to blue whether we are in the fixed level (rm_level_1) or in the scrolling level (rm_level_2).

Lines 2–9: Here we check if we are in the fixed level or not. If we are in the fixed level, we want the color variable to have no effect, so we set it to none; if we are in any other level (that’s not rm_level_1), we set the starting color to blue.

Now let’s create a new event by clicking Add Event ➤ Key Down ➤ Letters ➤ X. This will be our color-switching key, so let’s attach this code to the event:

```
1   switch( color )
2   {
3       case colors.red:
4           color = colors.blue;
5           sprite_index = spr_player_blue;
6           break;
7       case colors.blue:
8           color = colors.red;
9           sprite_index = spr_player_red;
10          break;
11  }
```

The preceding code will check for the value of the color variable. If it is red, it changes it to blue (lines 3–6), and it also changes the sprite of obj_player to spr_player_blue; vice versa, if it's blue, it changes the value of color to red (lines 7–10) and obj_player's sprite to spr_player_red.

There is no check for colors.none, because we don't want this option to be available when in the fixed shooter level (rm_level_1).

The next thing to implement is to change the color of the bullets according to the color of the ship.

To do that, we first have to add the color feature to the bullets. So open up obj_bullet_player and add this line at the bottom of the Create event:

```
color = colors.none;
```

Let's do the same also for obj_bullet_enemy's Create event by adding the same line:

```
color = colors.none;
```

According to the GDD, we must also change the binding of the shooting key to Z, since now it's the key for the primary shoot, while space is the key for the secondary shoot.

Right-click Key Down ➤ Space event and select Change Event. Now choose Key Down ➤ Letters ➤ Z. Double-click it to open the code and make these modifications:

```
if (can_shoot and global.game_state == states.playing )
{
    can_shoot = false;

    var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet_player);
6       bullet.atk = atk;
7       bullet.direction = direction;
8       bullet.spd = 10;
9
10      switch(color)
11      {
12          case colors.red:
13              bullet.color = colors.red;
14              bullet.sprite_index = spr_bullet_red;
15              break;
16          case colors.blue:
17              bullet.color = colors.blue;
18              bullet.sprite_index = spr_bullet_blue;
19              break;
20          default:
21              bullet.color = colors.none;
22              bullet.sprite_index = spr_bullet;
23              break;
24      }
25
26      alarm[0] = shoot_delay;
27  }
```

Lines 10–21: Here, after we create the bullet and change its properties, we check the color of the ship (if set) and change the color of the bullet according to it – both changing the sprite and the variable.

The rest of the code is unchanged.

Now run the game by pressing F5. You will be now able to change your ship's and bullet's color by pressing the X key.

Great! But right now, the color switching feature isn't doing much difference. We should implement those mechanics we talked about. Let's start from the player's bullets. We want that when the player kills an enemy with a bullet of the same color, they get additional score. Let’s make a new type of enemy for level 2, so that we can keep obj_enemy_red bound to level 1’s gameplay.

Our new enemy will have some features in common with obj_enemy_red, but will be different in other things. It would be great if we could define a generic enemy object on which all the enemies will be based. Fortunately, we can do this in GameMaker by using the concept of inheritance.

Inheritance

Inheritance is an important concept of the Object-Oriented Programming paradigm. It is the mechanism of basing an object construction upon another object. Thanks to this principle, we can define a general object that acts as a blueprint to other objects derived from it. There is a specific terminology to express the hierarchy that lies at the heart of inheritance: the generic object is called the parent, while the objects that derive from it are its children.

For example, in our game, we want to create different kinds of enemies that share the same basic structure; in particular, we want that every enemy has basic stats like HP, speed, and attack power and that they all are vulnerable to the player’s bullets; but we want to differentiate them in how they move and shoot. So we can create a parent object obj_enemy that has the basic features that all the enemies share so that we can extend this template creating children objects that implement other specific types of enemies.

Figure 6-2 shows a representation of that hierarchy concept; Alien-X is the parent object that is extended by children Alien-Y and Alien-Z.

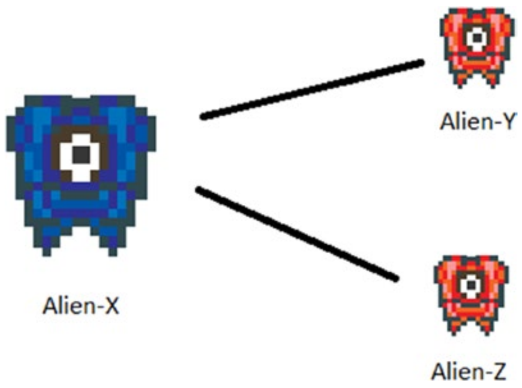


Figure 6-2. *Alien-X is the parent object that is extended by children Alien-Y and Alien-Z. Alien-Y and Alien-Z can both inherit or override Alien-X properties and events*

So let's create our enemy parent by right-clicking Objects in the Resources sidebar and selecting Create Object. Call this new object obj_enemy. There's no need to assign a sprite to this object.

For this new object, we will create some events that implement common things that we want all our enemies to have. Let's start with the create event.

In obj_enemy, click Add Event ➤ Create and put this code in it:

```
hp = 1;
atk = 1;
spd = 1;
color = colors.none;
```

We are assigning by default to every enemy some stats like HP, attack strength, speed, and color (but the color is none by default). We can still customize those stats for every single enemy by overriding them. We will see how very shortly.

Now create a Step event by clicking Add Event ➤ Step ➤ Step. We will use this event to manage the HP level of the enemies, since we want every enemy to die by default when HP drops to zero. So let's add this code to the Step event:

```
if ( hp <= 0 )
{
    instance_destroy();
}
```

We also want that after they died, every enemy explodes giving a base score of 100 points. So create a new Destroy event for obj_enemy and add this lines in it:

```
effect_create_above(ef_explosion, x, y, 1, c_dkgray);
score += 100;
```

Finally, we want to make standard the fact that Alarm 0 is bond to shooting, so that we don't have to rewrite every time that alarm if we don't want to implement a more advanced shooting for our enemies.

So click Add Event ➤ Alarm ➤ Alarm 0 and write this code in the event:

```
var cam = view_camera[0];
var cam_x = camera_get_view_x(cam);
var cam_y = camera_get_view_y(cam);
var cam_w = camera_get_view_width(cam);
var cam_h = camera_get_view_height(cam);

if ( global.game_state == states.playing )
{
    if ( x > cam_x and x < (cam_x + cam_w) and y > cam_y
    and y < (cam_y + cam_h) )
    {
            bullet = instance_create_layer(x, y, "Instances",
            obj_bullet_enemy);
            bullet.atk = atk;
            bullet.direction = direction;
            bullet.color = color;
        }
}
alarm[0] = room_speed * random_range(0.5, 5);
```

Now, every enemy inheriting from obj_enemy will have by default the shooting alarm to be Alarm 0, unless differently specified.

Since we are restructuring the enemies and we have our player moving in the eight directions, it's good to cover the case in which the player collides with an enemy. When this happens, we want the enemy ship to explode and the player to get standard damage. To do that, create a collision event with the player in obj_enemy, by clicking Add Event ➤ Collision ➤ obj_player, and put these two lines of code in it:

```
other.hp -= atk;
instance_destroy();
```

Now, every time an obj_enemy collides with the player, it will deal its damage to the player and then destroy itself.

Now that we have the parent object, we want to align obj_enemy_red to it, so that we can make use of obj_enemy's blueprint to implement some main features.

Firstly, let's create the parent-child relationship between obj_enemy and obj_enemy_red. Open up obj_enemy_red's Object Editor and click Parent to open the Parent view and click *No Object* to select obj_enemy so that it will become obj_enemy_red's parent.

Now, open obj_enemy_red's Create event and substitute the code with this:

```
1   event_inherited();
2
3   color = colors.red;
4
5   dir = 1;
6   start_x = x - 25;
7   end_x = x + 25;
8
9   move_down_speed = room_speed * 5;
10
11  alarm[0] = room_speed * random_range(0.5, 5);
12  alarm[1] = move_down_speed;
```

Line 1: Here we are telling GameMaker that we want this object to inherit completely all the code included in its parent's Create event. So it means that to have this line, it's like having copy-pasted all the code in obj_enemy's Create event.

We have to modify also obj_enemy_red's Step event so that it will inherit obj_enemy's behavior and implement its own movement policy. So open up obj_enemy_red's Step event and substitute the code managing the HP dropping with event_inherited. The result will look like this:

```
1   event_inherited();
2
3   if ( global.game_state == states.playing )
4   {
5       if ( x <= start_x or x >= end_x )
6       {
7            dir *= -1;
8       }
```

```

    x += spd * dir;
}
```

Lastly, let's get rid of Destroy and Alarm 0 events, since we want obj_enemy_red to only use the code in obj_enemy without further modifications.

Now that we have obj_enemy acting as a blueprint for all the enemies, we should use it also to manage collisions with bullets, so that we don't have to rewrite the same code for every enemy we make.

Double-click obj_bullet_player and right-click the collision event with obj_enemy_red, choose Change Event, and choose Collision ➤ obj_enemy.

All done! Now the bullets will damage and kill every object inheriting from obj_enemy. To double-check, feel free to press F5 and test your new code.

Color shooting

To add the possibility of enemies to shoot bullets of their own color, it's convenient to add this functionality to every enemy, so that we don't have to rewrite the same code again and again.

Tip It's a very recommended thing to try not to write the same code multiple times. It's one of the principles of good programming, and it's commonly referred to as DRY (Don't Repeat Yourself). If you know that a piece of code should be used multiple times, try to think of how you can avoid it and write it once and use it everywhere. Some of the ways you can do this are both by using OOP principles or functions.

```
1   if ( global.game_state == states.playing )
2   {
3       var bullet = instance_create_layer(x, y, "Instances",
        obj_bullet_enemy);
4       bullet.atk = atk;
5       bullet.direction = point_direction(x, y, x, y+1);
6       bullet.color = color;
7
8       switch( color )
9       {
10          case colors.red:
11              bullet.sprite_index = spr_bullet_red;
12              break;
13          case colors.blue:
14              bullet.sprite_index = spr_bullet_blue;
15              break;
16          default:
17              bullet.sprite_index = spr_bullet;
18              break;
19      }
20  }
21  alarm[1] = room_speed * random_range(0.5, 5);
```

Lines 6–19: We assign obj_enemy's color's value to bullet's color; and then, according to that value, we change the bullet's sprite.

Now that we are applying color-based shooting to every enemy, we want this feature to be active only with enemies that are not in rm_level_1. In fact, it's total nonsense to have that feature there, since the final goal of color-switching is to recharge the super-attack, but that would be overkill in rm_level_1. So we need to make a little modification to obj_enemy_red's Create event. Instead of having this line

```
1   color = colors.red;
```

change it to this:

```
if ( room == rm_level_1 )
{
    color = colors.none;
}
else
{
    color = colors.red;
}
```

Adding this control, obj_enemy_red will shoot red bullets only if it's not in level 1. Perfect! Actually, we have another feature that we want to be active only in level 1, and this is the jumping down action that we implemented using Alarm 0. Open up obj_enemy_red's Alarm 1 event and add the check for level 1 like shown in the code below:

```
1   if ( room == rm_level_1 and global.game_state == states.
    playing )
2   {
3       y += 50;
4   }
5   alarm[0] = move_down_speed;
```

Now this code will be triggered only if we are in level 1 (line 1).

We should do the same modification we did in obj_enemy_red Create event to obj_player. Let's open it up by double-clicking it in the Resources sidebar. Head to its Create event and change this line

```
color = colors.blue;
```

into this:

```
if ( room == rm_level_1 )
{
    color = colors.none;
}
else
{
    color = colors.blue;
}
```

Great! Now we have our color-switching feature active only for levels that are not level 1!

You can test it by running the game in level 1 and verifying that you cannot switch color with your ship and both you and the enemies shoot yellow bullets.

Now that we have this cleared out, let's add more fun to the game, by adding more enemies that we can use to fill level 2.

More enemies

We have color-switching and enemy hierarchy done right. Now we can safely add more enemies to enhance both fun and challenge. In particular, we will add these enemies:

- **Blues**: Just like obj_enemy_red, but blue!
- **Red/Blue Walkers**: This kind of enemy will follow a predesigned path and shoot red/blue bullets.
- **Red/Blue UFOs**: Red/blue UFOs that track the position of the player and shoot bullets of their same color.

Ain't nothing but the blues

Blues are just needed to balance the presence of the reds in level 2, since color switching will be active. In fact, having this feature with only one type of enemies wouldn't make much sense.

They are pretty simple to create: just duplicate obj_enemy_red by right-clicking it in the Resources sidebar and choosing Duplicate.

Rename the duplicated object as obj_enemy_blue and open it up by double-clicking it.

The only things we should change are the sprite that you should change to spr_enemy_blue and the value of the color in the Create event. Opening the Create event, you should change the code that defines the color variable like this:

```
1   if ( room == rm_level_1 )
2   {
3       color = colors.none;
4   }
5   else
6   {
7       color = colors.blue;
8   }
```

The only interesting line, here, is line 7 that we changed according to the enemy's color.

That's it! We have a new enemy that will shoot blue bullets!

Before trying this in level 2, we have to make sure that when the player's ship has the same color of the colliding obj_bullet_enemy instance, they're not getting damage.

Double-click obj_bullet_enemy, open the Collision event with the player, and change the code to this:

```
1   if ( room == rm_level_1 )
2   {
3       other.hp -= atk;
4   }
5   else
6   {
7       if ( color != other.color )
8       {
9           other.hp -= atk;
10      }
11      else // same color, so not getting damage
12      {
13          // TO DO: super attack charge
14      }
15  }
16  instance_destroy();
```

What we did here was to prevent the use of color-based damage for level 1 and add it only on other levels.

Don't worry, w We will take care of line 9 very soon!

Let's create a test level to see that everything is working. Right-click Rooms in the Resources sidebar and select Create Room and call it rm_test.

Open up rm_test and Drag and Drop these instances in it:

- obj_player
- obj_enemy_red
- obj_enemy_blue
- obj_controller

Now press F5! You should see that getting hit by a bullet of your ship's same color is not a threat anymore. You will get your ship damaged only by different colored bullets.

Also, shooting to an enemy with a bullet of its color will get you 100 more points!

Walkers on paths

In your game, you may want, at a certain point, to let one of your objects to follow a predefined path. For example, in a top-down RPG, you may want some characters to walk back and forth on a street or things like that. To do that, there is a pretty nice tool in GameMaker called Paths.

Paths are a series of points in a room connected to one another by a line or a curve, and they represent a way that your instances can walk back and forth.

We will use those useful tools to create interesting patterns that our enemies can follow while the player traverses the level. We will call those enemies walkers.

To create our first walker, right-click Objects in the Resources sidebar and select Create Object. Rename the newly created object obj_enemy_walker_red and set its sprite as spr_enemy_red since it's not important for the player to recognize the difference between a base red enemy and a red walker. The only difference between the two is the pattern they follow moving.

We want this enemy to be child of obj_enemy, so click Parent and select obj_enemy as its parent.

Now, double-click the newly created obj_enemy_walker_red and create a new Create event with this code inside:

```
event_inherited();
color = colors.red;
```

There's no need to check if we are in room 1 or not, since this kind of enemy won't be present there.

We now want to create a path for obj_enemy_walker_red to follow, so let's open up rm_level_2. In the Room Editor, click Create New Path Layer as shown in Figure 6-3.

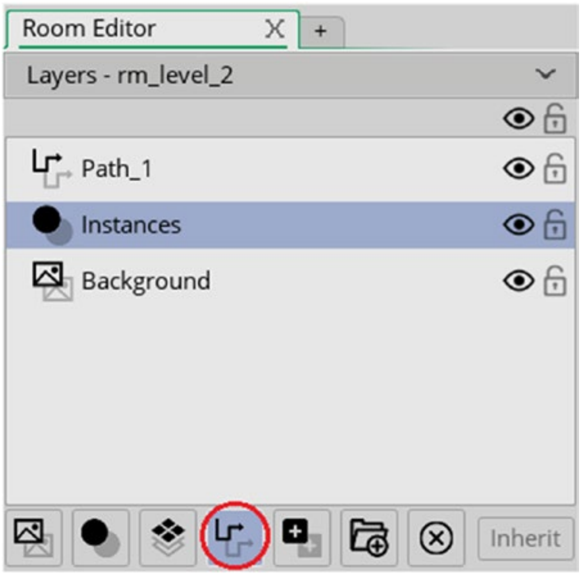


Figure 6-3. *The button circled in red allows you to create a new path layer*

Creating a path layer will open a new section in the Room Editor called Path_1 Layer Properties. From there, click Select Path and click Create New. You're now in creating mode, and you can create a path by clicking points directly in the room.

Create a path (Figure 6-4) of your choice that you want one (or more) of your walkers to follow. When you are done drawing the path, you can change some details like the smoothness of the curve and if it's closed or not.

A closed path is a path where the last point is connected to the first one; GameMaker will automatically draw a line between that path's first and last points.

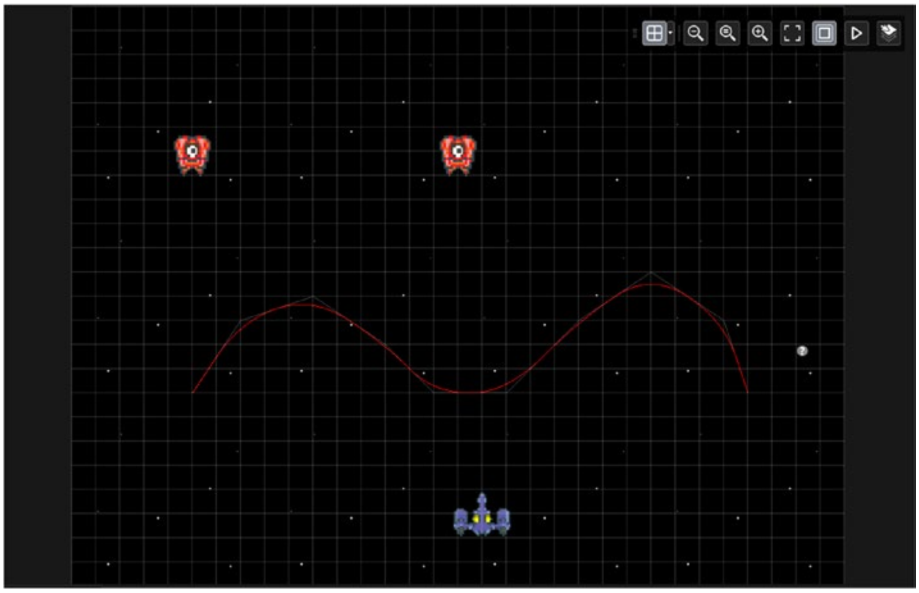


Figure 6-4. *Designing a path in a room*

After you designed your path, give it a name – for example, path0 – by renaming it. Now we only need to assign this path to obj_enemy_walker_red.

Open up obj_enemy_walker_red's Object Editor and add this line of code at the bottom of the Create event:

```
spd = 3;
path_start(path0, spd, path_action_reverse, false);
```

Calling path_start, you can assign a path to an object and decide how it will traverse it.

The signature of path_start is

```
path_start( path_name, obj_speed, path_action, path_absolute )
```

where

- *path_name* is the name of the path.
- *obj_speed* is the speed at which you want the object to traverse the path.
- *path_action* is the action you want to be executed after the object traversed the path – like walking back to the reverse path, starting over again from the first point, and so on. See the documentation for more details.
- *path_absolute* tells GameMaker if you want the object to walk the path following the exact coordinates you specified or if you prefer to walk relatively from the coordinates of the object.

In our case, we want obj_enemy_walker_red to traverse path0 at speed spd (that is equal to 3), we want it to walk back the reversed path after it reaches the last point of the path, and we want it to do it relatively to its starting coordinates.

Place an obj_enemy_walker_red instance in the room and press F5 or select Run from the toolbar to compile and run the game. You will see that obj_enemy_walker_red is traversing the path you designed at the speed you specified in its Create event.

Great! We have a new kind of enemy! Let's make also a blue one!

To make a blue walker, you just need to duplicate obj_enemy_walker_red by right-clicking it and selecting Duplicate.

Remember to change its color to blue both in the sprite by choosing spr_enemy_blue and in the create event by changing the value of color like this:

```
event_inherited();
color = colors.red;
spd = 3;
path_start(path0, spd, path_action_reverse, false);
```

Now we have a blue walker! Easy peasy!

You can have fun making paths and assigning them to different kinds of walkers by duplicating them. Designing nice paths can greatly enhance the game experience and increase the challenge for the player, so free your imagination!

Anyway, if you are not really into that, you can just use the one path we just made and use it relatively with any of your walkers.

Unidentified Flying…Instance!

Our third enemy is a turret. It tracks the movements of the player and shoots at them at regular intervals.

Create a new object by right-clicking Objects and selecting Create Object and call it obj_enemy_ufo_red. Assign to this object obj_enemy as a parent.

Now create a new Create event for obj_enemy_ufo_red and put the following code inside:

```
event_inherited();
spd = 1;
color = colors.red;
alarm[0] = room_speed * spd;
```

We set the Create event so that it will inherit all the stats from obj_enemy, but we are changing some of them. We are setting the color as red and the speed as 1. We will use the speed not to move, but to shoot. We want the turret to shoot every second, so we are setting the alarm with spd.

Since we want the ufo to shoot every second, we have to modify the shooting rate in Alarm 0's event, because the default value is calculated randomly (in obj_enemy).

So create a new Alarm 0 event for obj_enemy_ufo_red and, after inheriting the parent's event, set Alarm 0 to the right value, like this:

```
event_inherited();
alarm[0] = room_speed * spd;

```

Now we want to make sure that the ufo follows the player around aiming at their spaceship. We can do this using the Step event. Once per frame, we will check the position of the enemy ship and set the direction of obj_enemy_ufo_red to point at it.

Let's do it by creating a step event clicking Add Event ➤ Step ➤ Step and adding this code:

```
1   event_inherited();
2
3   if ( instance_exists(obj_player) )
4   {
5       var dir = point_direction(x,y, obj_player.x,
        obj_player.y);
6       direction = dir;
7       image_angle = dir;
8   }
```

After inheriting the event of obj_enemy at line 1, we are checking that the player exists at line 3; and if it does, we set obj_enemy_ufo_red's direction so that it faces obj_player. We also rotate the sprite by using image_angle (line 7) so that also the sprite faces the player's direction.

Let's test our UFO by putting it into the room and running the game pressing F5.

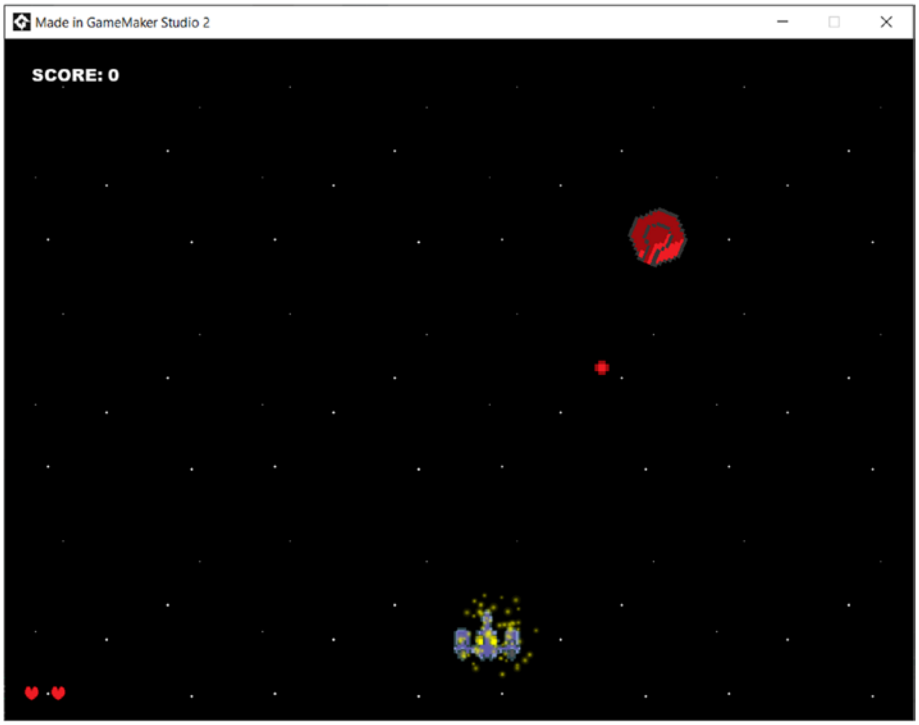


Figure 6-5. *UFOs will track the player's movements and aim to them while shooting*

Great! While the player moves, the UFO tracks their movements and shoots aiming perfectly (Figure 6-5) – just what we wanted.

Now let's create a blue ufo too, by duplicating obj_enemy_ufo_red. You can do that by right-clicking it in the Resources sidebar and choosing Duplicate. Rename it obj_enemy_ufo_blue and open up its Object Editor.

Let's edit its sprite by choosing spr_enemy_ufo_blue and its Create event by changing its color value to blue like this:

```
event_inherited();

spd = 1;
```

```
color = colors.blue;

alarm[0] = room_speed * spd;
```

Cool! Now we have two UFOs ready to hunt down the player's spaceship! We only need a super weapon to put them in their place!

Super-attack

Following the design we made in our GDD, the color-switching feature has the main objective of charging the special attack that will blow up every enemy visible to the camera.

We will assign this attack to the spacebar, and this will be available only when a controller variable gets to 100, representing a 100% charge of the special engine that triggers the bomb.

We need to define a controller variable for the special attack, and we can do this in the Create event of obj_player. Let's open up obj_player's Object Editor and add this line to the bottom of the event:

```
super_attack = 0;
```

Now, click Add Event ➤ Key Pressed ➤ Space to create the event that will trigger the attack and put this code in it:

```
if (super_attack >= 100)
{
    var cam = view_camera[0];
    var cam_x = camera_get_view_x(cam);
    var cam_y = camera_get_view_y(cam);
    var cam_w = camera_get_view_width(cam);
    var cam_h = camera_get_view_height(cam);

9        for (var i = 0; i < instance_number(obj_enemy); i++)
10       {
11           var enemy = instance_find(obj_enemy, i);
12           if point_in_rectangle(x, y, cam_x, cam_x+cam_w,
             cam_y, cam_y+cam_h)
13           {
14               with(enemy) instance_destroy();
15           }
16       }
17
18       super_attack = 0;
19   }
```

Line 1: We want to execute this code only when the attack is fully charged (when it reaches 100).

Lines 3–7: Here we calculate the coordinates of the visible portion of the screen like we did for our obj_camera. We are not using the same coordinates by taking them from that object, because we want that feature to not be dependent on obj_camera.

Lines 9–16: We're looping between all the enemies to find the ones inside the boundaries of the camera to blow them up!

Now that we have the attack, we need to visualize its data and add the functionality to charge it by absorbing enemies' bullets.

To do that, we will use obj_controller's Draw GUI event, since we are using it for every other HUD element.

Open up obj_controller's Draw GUI and edit its code so that it looks like this:

```
draw_set_font(fnt_messages);
draw_set_color(c_white);
draw_text(30, 30, "SCORE: " + string(score));

```

```
if ( global.game_state != states.playing )
{
    if ( global.game_state == states.paused )
    {
        draw_text( 900, 30, "PAUSE" );
    }
    else
    {
        draw_text( 850, 30, "GAME OVER" );
    }

    for( var i = menu_min; i < opt_number; i++ )
    {
        if ( menu_index == i )
        {
            draw_set_color(c_white);
        }
        else
        {
            draw_set_color(c_dkgray);
        }
            draw_text( 850, 600 + 30 * i, options[i] );
    }
}


if ( instance_exists(obj_player) )
{
    var xhp = 30;
    repeat(obj_player.hp)
35      {
36          draw_sprite(spr_life, 0, xhp, camera_get_view_
            height(view_camera[0])-30);
37          xhp += 30;
38      }
39
40      draw_set_font(fnt_messages);
41      draw_set_color(c_white);
42      draw_text(30, 60, "X-BOMB: " + string(obj_player.super_
        attack) + "%");
43  }
```

Lines 40–42: These are the only lines we modified. Nothing really hard. We just set a font and a color for the text, and then we wrote the label X-BOMB followed by the value of super_attack variable. Last thing to do is to charge it up!

We will manage the charging in the collision event between the player and the enemy's bullet. So open up obj_bullet's collision event with obj_player. We only have to add one line where we put that placeholder comment. The new code of the collision event will be this:

```
if ( room == rm_level_1 )
{
    other.hp -= atk;
}
else
{
    if ( color != other.color )
    {
        other.hp -= atk;
    }
11      else
12      {
13          if ( other.super_attack < 100-10 )
14          {
15              other.super_attack += 10;
16          }
17          else
18          {
19              other.super_attack = 100;
20          }
21      }
22  }
23  instance_destroy();
```

Lines 13–20 are the part we changed. For every bullet colliding with the player, if it has the same color of the player's ship, the super-attack gets charged by 10. We do an extra check to see if the variable is going over 100 and fix it if it does.

Ok, now the whole system is working! We just have to double-check if we did everything alright!

Run the game by pressing F5 or clicking the Run button in the toolbar!

Great! When the ship gets hit by bullets of the same color, it charges (Figure 6-6) the special attack; and when it reaches 100%, we can use it by pressing the spacebar wiping out every enemy in the area (Figure 6-7)! Cool!

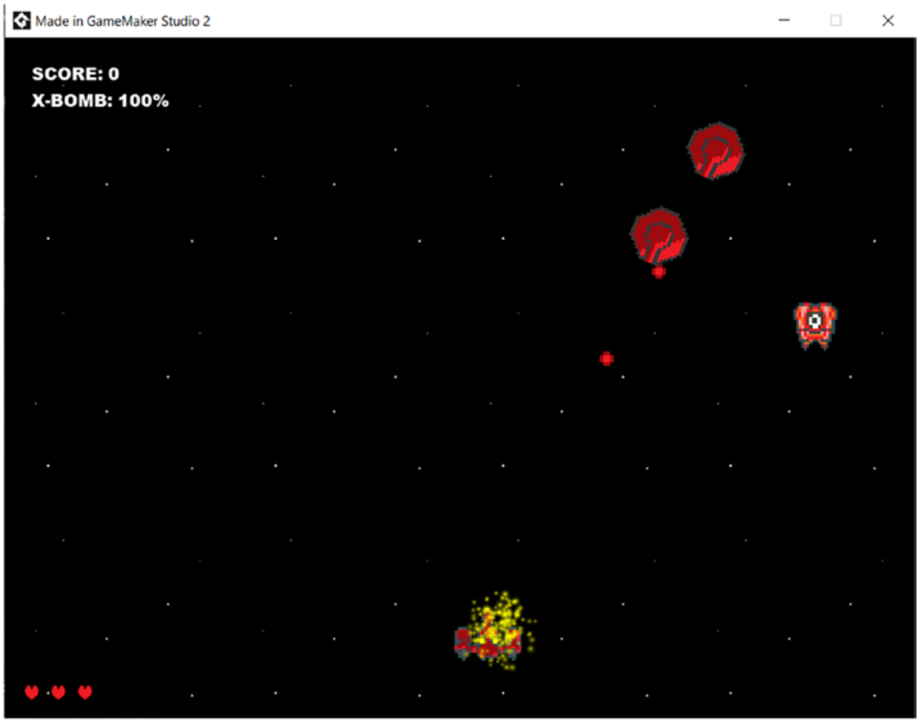


***Figure 6-6.** The X-bomb charge reaches 100%, thanks to the collision with bullets of the player's same color*

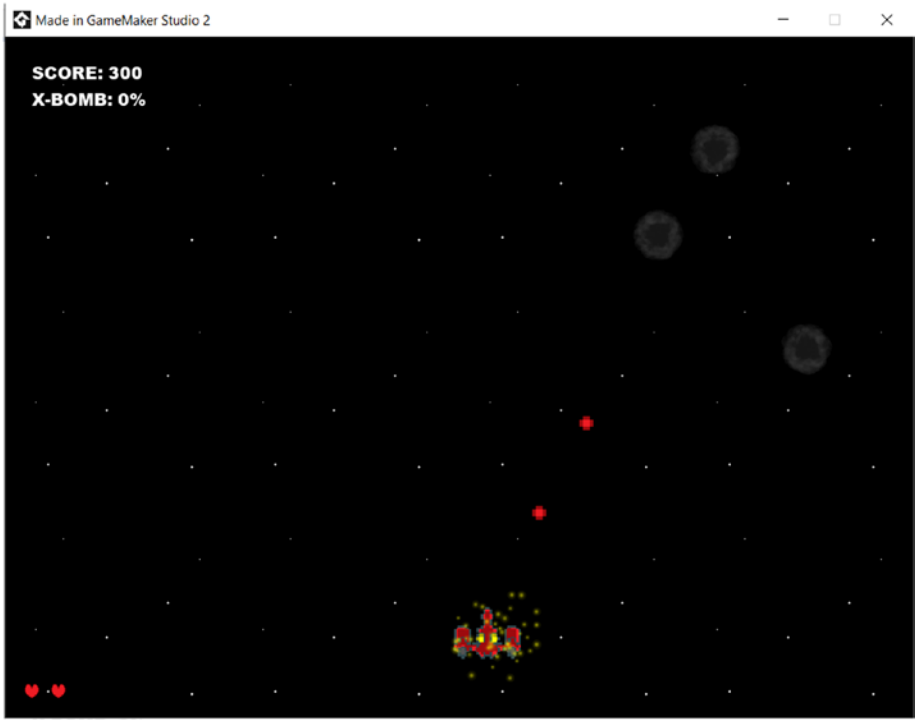


Figure 6-7. *The X-bomb, when triggered, destroys every enemy in the visible area*

How to design a good shmup level

Shoot 'em ups are games very based on gameplay mechanics. The most important thing to do is to find a good gameplay mechanic that can drag forward all your game. Level design, for any of the levels you design, should be based on the concept of highlighting this principal mechanic.

In Space Gala, we should design level 2 to exploit the new powers we are giving to the player. First of all, we should introduce the player to the feature, by making so that the first part of the level alternates blue and red enemies. Then we should start mixing them in small quantities. The end of the level should feature at least a couple of sections with a big number of blue and red enemies following interesting patterns, so that the player can feel the tension of the frantic pace of our game and they will be forced to master their color-switching skills.

Try to design a level following those guidelines. If you have problems coming up with something meaningful, you can check the level I made in the code of this chapter.

Boss fighting

Boss fights are one of the most important and fun elements in a video game. We will talk extensively about them in the next chapter. For now, let's just focus on how to make a boss in our game.

Shoot 'em ups have probably the most interesting boss fights. They are often made of phases, in which the boss changes moveset and/or attacks, and they mix various styles. There are bosses that use a lot of different and artistic bullet patterns, there are bosses that spawn minions, and so on.

Our boss will start by aiming at the player and shooting normal bullets. Then, after reaching half of its HP, it will grow bigger and start spawning turrets.

Let's start by creating a new obj_boss object by right-clicking Objects in the Resources sidebar and choosing Create Object.

Set spr_boss as obj_boss's sprite and select obj_enemy as its parent. Now you should see all the events inherited from obj_enemy in the Events panel of the Object Editor.

Right-click the inherited Create event and choose Inherit Event. Now you can access the code. We will use this event to set some important variables and start the attack timer:

```
event_inherited();

maxhp = 10;
hp = maxhp;
```

```
5   str = 1;
6   atk_delay = 2;
7
8   phase = 0;
9
10  alarm[0] = room_speed * atk_delay;
```

Lines 3–4: Setting the HP for the boss. We will use maxhp in later calculations.

Lines 5–6: Setting the strength of the attack and the attack delay (for the first phase).

Line 8: We will use this variable to keep track of the various phases of the boss.

Line 9: As usual, alarm 0 is the one used to attack.

Inherit also the Step event by right-clicking it and selecting Inherit Event. We want to keep the check on the HP, but we want to also add a tracking feature and the phase management:

```
1   event_inherited();
2
3   if ( instance_exists(obj_player) )
4   {
5       var dir = point_direction(x,y, obj_player.x, obj_
        player.y);
6       direction = dir;
7   }
8
9   if ( hp < maxhp/2 and phase == 0)
10  {
11      phase = 1;
12      alarm[1] = room_speed * 0.1;
13  }
```

Lines 3–7: This code will rotate constantly the direction of the boss to face the player. This won't affect the boss sprite.

Lines 9–13: These lines change the phase of the boss to 1 when the HP drops to half of the maximum HP (maxhp/2). When the boss is in phase 1 (phase == 1), it grows bigger and then starts spawning UFOs.

Now, we have to implement the growth of the boss. Let's do it by creating a new Alarm 1 event by clicking Add Event ➤ Alarm ➤ Alarm 1. This is the alarm that will rule the growth of the boss. Add this code to the event:

```
1   if(sprite_width < 120 * 3)
2   {
3       if ( global.game_state == states.playing )
4       {
5           image_xscale += 0.1;
6           image_yscale += 0.1;
7       }
8       alarm[1] = room_speed * 0.1;
9   }
10  else
11  {
12      atk_delay = 1;
13      alarm[2] = room_speed * 4;
14  }
```

Lines 1–9: This code makes the boss grow bigger until it reaches three times its original width (120 pixels).

Lines 10–14: After the boss finishes growing, it sets its attack delay to 1 (meaning one bullet per second). After this, it sets Alarm 2 so that it starts spawning one UFO every 4 seconds.

Finally, let's create a new Alarm 2 event to manage the UFO spawning by clicking Add Event ➤ Alarm ➤ Alarm 2. Put the following code in the event:

```
1   if ( global.game_state == states.playing )
2   {
3       var cam = view_camera[0];
4       var cam_x = camera_get_view_x(cam);
5       var cam_y = camera_get_view_y(cam);
6       var cam_w = camera_get_view_width(cam);
7       var cam_h = camera_get_view_height(cam);
8
9       var minion_x = random_range( cam_x, cam_x + cam_w );
10      var minion_y = random_range( (cam_y + cam_h)/2, cam_y +
        cam_h );
11      instance_create_layer(minion_x, minion_y, "Instances",
        obj_enemy_ufo_red);
12  }
13
14  alarm[2] = room_speed * 2;
```

Lines 2–7: As usual, we calculate the coordinates of the cam.

Lines 9–11: These lines calculate random coordinates in the lower half of the screen and spawn an UFO in there.

That’s it! Now we just need to do some little adjustment. We want the player to not destroy the boss when colliding with it, and we want the boss to make three explosions and give 1000 points as score, when killed.

Let’s start with the collision. Right-click the collision event with obj_player and choose Override Event and write this code in it:

```
other.hp -= atk;
```

The preceding code will just damage the player when colliding with the boss.

Now, let’s take care of obj_boss’s Destroy event.

Override also the Destroy event and write the code below in it:

```
effect_create_above(ef_explosion, x-30, y, 1, c_fuchsia);
effect_create_above(ef_explosion, x, y, 2, c_purple);
effect_create_above(ef_explosion, x+30, y, 1, c_fuchsia);
score += 1000;
```

Here we are! Put the boss at the end of level 2 and test the game by pressing F5 (Figure 6-8 shows the boss fight in action).

If you followed me until now, everything should work properly!

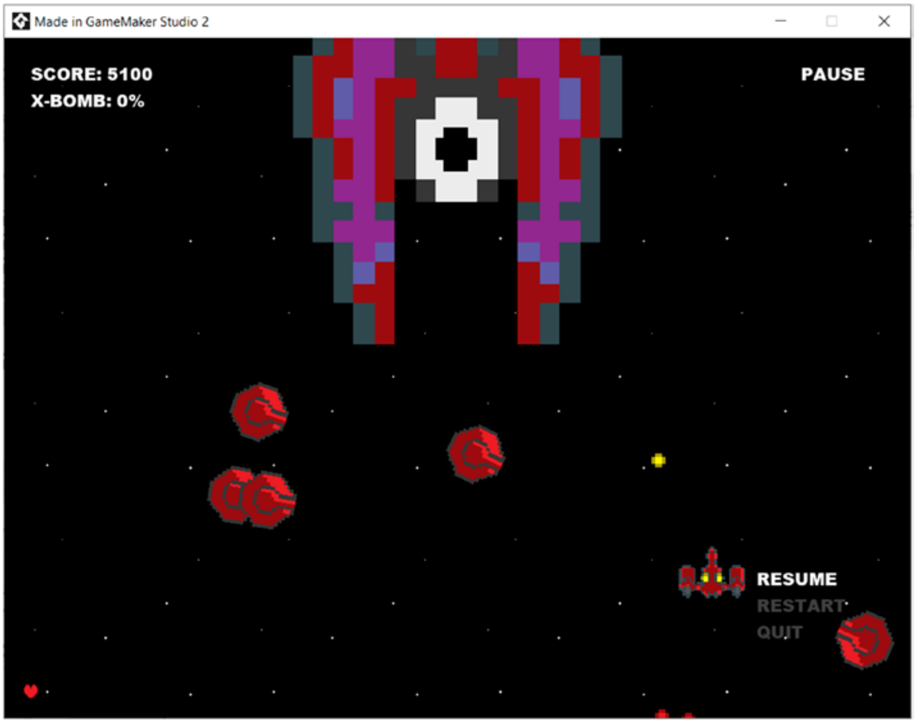


Figure 6-8. *In the middle of phase 1!*

Great! The game is finished! We have an interesting gameplay and a variety of enemies! More importantly, we have a boss!

This game is a template that you can use to play, learn, and experiment! Free your imagination and try to make new levels or even extend the game with brand-new features!

Only by experimenting by yourself you can channel the game designer that's in you!

Conclusion

Space Gala started very small and became a game full of features and with an interesting gameplay. You learned so much in this chapter! We created a scrolling shooter filled with different kinds of enemies; we developed a chargeable special weapon, thanks to a color-switching system that allows the player to absorb bullets of their same color; we learned how to create paths and make NPCs follow them; and finally we made our very first boss fight!

This is amazing, and you should spend some time cheering to your determination and accomplishment!

Before diving into the next chapter, you should experiment with your new game and try to create some new exciting features (don't forget to update your GDD, if you do)!

In the next chapter, we will talk further about boss fights; and we will analyze some of the most important and iconic of the video games history, to better learn how we can do better and how a boss fight can be both challenging and fun!

TEST YOUR KNOWLEDGE!

1. How do you play an audio file in GMS2?
2. What is a viewport? How many of them can you have in GMS2?
3. What is a camera? How many of them can you have in GMS2?

4. What is an active camera? How many of them can you have in GMS2?
5. What is the difference between a viewport and a camera?
6. How can you access active cameras in GML?
7. How can you get the properties (position and size) of a camera?
8. How can you check if an instance of an object exists in a room?
9. How can you check if an instance is inside the boundaries of a specific active camera?
10. Can you improve color-switching by adding a new bonus or malus?
11. Do you think that adding more colors to color switching would be beneficial to gameplay? Why?
12. Can you explain the concept of inheritance?
13. Why is inheritance advantageous?
14. Can bullets benefit from inheritance?
15. Right now we are reusing obj_enemy_red from the previous game. This can be isolated in a separated enemy object not featuring the color property. Can you improve the hierarchy of obj_enemy by creating the colorless object?
16. What is a path in GMS2? How can you create one?
17. How do you assign a path to an instance?
18. Are paths always walked at the coordinates at which they are drawn when created?
19. How can you make an instance track the position of another instance?
20. Can you come up with a new kind of enemy?

21. Enemies are quite weak until now. Make them stronger to increase the challenge. How does this affect the level design?
22. What are the best features of a shmup level?
23. Design a new level for Space Gala trying to make the color-switching feature shine.
24. What are the key features that a boss fight should have?
25. Design and implement a third phase for obj_boss.
26. Analyze DonPachi's combo mechanic, Bangai-O's grazing mechanic, and Ikaruga's polarization system.
    a. Can you tell what makes them so interesting and fun?
27. Play a shmup game (you can pick one of those listed) and write down what you liked and what you didn't like in the gameplay.
    a. How would you fix the things you didn't like?
    b. Can you integrate the things you liked in Space Gala?

CHAPTER 7

Designing Bosses

Video games are probably the most complex media around. They are made of a huge quantity of different components. They tell us stories in many ways: by showing us places using the concept of environmental storytelling or simply by showing us Cutscenes and – of course – by letting the player interact with different characters.

The most interesting and often complex and charming characters are the ones that get in the players' way: the so-called bosses.

Bosses are special characters that often distinguish themselves from the regular enemies both from a narrative and a gameplay point of view. They are often found at the end of a level or an area and are very important for both the progression of the player and the story.

In fact, most of the time, bosses serve as teachers for the player to learn some mechanics or to master some techniques; and the best bosses are always the ones that, while doing this, make it fun or important in terms of storytelling and pathos. But what does it mean to make a boss fight fun?

The meaning of fun is debated in many fields from psychology to game design. One of the most accredited theories about the nature of fun in game design is the one exposed by Raph Koster in his book *A Theory of Fun*. He says that a game is fun as long as it teaches the player something without confusing them with complexity. When a game stops teaching or makes the learning too hard, it is not fun anymore.

This is true for every aspect of gaming, especially boss fights.

But let's talk about this more in detail!

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_7

A good boss fight doesn't feel unfair; it feels challenging and sometimes even intimidating, but also rewarding and fair, meaning that it feels beatable with adequate training and precision.

Boss fights are particularly fun when they teach some gameplay mechanic to the player making them learn new things during all the fight. The thing to learn is usually a technique, like a combo, a new move, or a new item, but sometimes is also some broader concept, like how to outsmart enemies or how to use the environment at your own advantage. The possibility to face a challenge with critical thinking gives the feeling of being in charge of the situation and that the choices made are meaningful.

A good example of a boss fight with exactly those characteristics is the *Draygon* in *Super Metroid* (Nintendo, 1994).

The **Draygon** is an interesting boss because it's a huge beast that can fly and it's hard to dodge, so nearly all of its attack will hit the poor Samus (the player's avatar). The player can defeat it only by mastering the controls or outsmarting the beast. In fact, during the most dangerous moment in the boss fight – when the Draygon grabs Samus and takes her around the map flying – she can set herself free by shooting the grappling hook into a sparking broken machinery and channeling electricity into the Draygon.

Outsmarting such a huge and strong beast makes the player feel proud of themselves and gives them a strong sense of satisfaction that makes the boss fight memorable. The concept of finding a way to turn into easy something really hard leverages on our primordial desire to be the *smartest monkey*, and it always works to make the player feel engaged and motivated to play. Succeeding in something that's really hard by using our intelligence or hard-earned skills gives us pleasure and engagement.

Another good lesson that the Draygon's boss fight teaches us is showing game mechanics. By using the grappling hook trick, a player understands or reinforces the concept that they can use weapons and tools to interact not only with enemies but also with the environment and every tool can have a different and useful effect on the environment, opening new ways to do things and reach places. And that's not only the key to beat the Draygon easily, but it's also the good way to play a metroidvania.

Not only the Draygon boss fight does all this but it also sets a tone on the narrative of the game. In fact, while you fight this monster, you feel the sense of danger and urgency, and you can see that Samus is fighting a beast that is way stronger and bigger than her. After defeating it, you feel like a hero! Samus becomes – in the eyes of the player – the strong and brave heroine that she is in the game box cover. This boss fight enhances the atmosphere of the game and gives it credibility, which keeps the player immersed in the game and focused.

Draygon's boss fight teaches us how to create a challenging boss fight by exploiting the game's core mechanics and using that fight to reiterate on gameplay concepts and set a tone on the narrative of the game. It teaches us that to be memorable, a boss fight should be able to provide the player some sense of autonomy by giving them the possibility to face the challenge in an alternative way, maybe by exploiting some weakness of the boss or by using some tool in combination with the environment or some other creative way to take it down. Autonomy makes the player feel in charge of their own actions and in command of their destiny, so it keeps the player motivated and engaged.

A very similar example of autonomy given by a way to avoid a hard fight is ***Ceaseless Discharge*** in *Dark Souls* (*FromSoftware*, 2009). This is an optional boss, a massive demon made of lava that you trigger by stealing his sister's overpowered robes (one of the best equipment in the whole game) and you have to beat it in an area basically made of corridors. All those corridors, except for some very little spots, are accessible by the demon's attacks. Those attacks deal fire and physical damage, and they track the player's movements – just like every enemy and attack in Dark Souls (even arrows track the player like if they are homing missiles). You can beat this boss by engaging in a very delicate dodge-and-attack fight where you must remain super-focused for a long time because probably if you get caught you will die (depending on your vitality and fire resistance level); or you can trick the demon by forcing him to follow you in the way back to the entrance of the area where there is a pit that he cannot cross. The demon wants so hard to punish you and take back his sister's robes that he rashly jumps into the pit and hangs on the edge of the path unable to move. That's your moment: you can now spam the attack button until the demon loses the grasp and falls in the pit.

Ceaseless Discharge's secret is so well thought in narrative terms that he's become one of the most loved bosses in the game because people empathize with him and appreciate the love and affection he feels for his beloved dead sister. They feel bad for the suffering to which he's doomed forever because of the lava that constantly burns his flesh. For this reason, players either decide to not fight him even if it's an easy way to make souls or decide to kill him to stop his pain. In both cases, it's evident that players feel motivated in their actions and feel that their decisions have an effect and a meaning in the game world they're in, even when it's all players' role play. This makes Ceaseless Discharge a good boss fight and a worthy successor of Super Metroid's Draygon.

Teaching and experimenting

One of the video game series that gets everyone to agree is *The Legend of Zelda*.

Zelda's series is an endless source of game design lessons. One of the most important is around boss fighting. And between all the boss fights in all the Zelda games, probably one of the most interesting is the first boss fight in The Legend of Zelda: A Link to the Past. It consists in six ***Armos Knights*** jumping around the room in various patterns trying to collide with the player damaging them. Because of how Link attacks and how the knights move, it's very difficult to dodge and attack them with the sword. However, just before the boss fight, there is an area with a chest containing a bow. Using the bow allows the player to stay at a distance and eliminating one by one all the knights easily, turning a challenging boss fight into an easy task. This is the basics of teaching how to use new items via boss fighting. You get a new item, you test it on easy foes, and then you get tested on your skills with a boss fight. After you defeated the boss, the game knows that you mastered that technique and can move forward adding complexity to the gameplay.

Learning and being tested is super good and makes a challenge fun. But, as we said before, the possibility to act independently is very important. This is achieved by letting the player experiment and face the challenge in different nonstandard ways. So it's important to teach techniques in boss fights, but it's equally important to not force the player into it and give them the possibility to win the boss fight in an alternative way.

One of the best examples of autonomy and experimentation in a boss fight is ***The End*** in *Metal Gear Solid 3: Snake Eater* (*Konami*, 2004). It consists in a sniper challenge against the veteran sniper nicknamed The End. He's hiding in the forest with full camouflage constantly aiming and shooting at the player. The standard way to take him down is by using the binoculars searching for the glimpse of his sniper rifle's lens or by using the directional microphone to listen to his heavy breath and, after locating him, take him down.

This is super fun and educational, since you get to test every important skill in the game: observing, planning, camouflaging, stealth, and so on. You also learn how to use important tools that can help you out also outside the boss fight. This is fun because you learn and you internalize the reasons why you're doing it. In fact, he's trying to stop you from chasing The Boss (a principal character in the game), a character that you want to reach so hard! This is also a *smartest-monkey* kind of challenge, since you have to outsmart a veteran trying to locate him without being noticed.

There are a couple of tricks to avoid the fight, making this probably the most interesting boss fight in the game. You can avoid facing The End by sniping him in an early point in the game or by saving the game when the fight starts and loading it one week later (one week in the real world, not in the game). With the second method, since The End is very old, he will die from natural causes allowing you to proceed without problems. Genius!

As you can see, good boss fights always give you possibilities, a challenge, motivation, and a lesson to learn.

Motivation!

There are boss fights that violate all the guidelines we talked about until now, but are memorable anyway. Those boss fights rely on the narrative aspect and give the player a strong motivation to fight leveraging on their feelings and their need of social relatedness within the game world and its characters.

One of the most brilliant examples of this is yet again from *Super Metroid*: *Mother Brain*'s boss fight. ***Mother Brain*** is the mother of all the metroids and the root of all the disasters happening in *Super Metroid*. This is a good enough reason to kill her once and for all, but as this were not enough, just before the fight starts, she decides to kill the baby-metroid (now grown up) you adopted in the beginning of the game. Well, this means war! You are equipped with a vaporizing laser, and shooting at her brain shattering it in pieces is one of the most satisfying experiences in video games. It's a very easy boss fight, but it's still super enjoyable because you want to kill her off because of what she did.

This is how to internalize motivations like a boss (pun not intended). You create a narrative setting, create relationships and sympathies between the player and the NPCs, and then make dramatic events happen. The player is engaged because he cares about the characters and knows that his actions have a meaning and make the difference. So they play and don't give up, even if the challenge is hard; and the more the challenge is hard and fun and the motivation is strong, the more the boss fight is memorable.

A contemporary example of a strongly motivating boss fight leveraging on the need of social relatedness can be found in *Undertale* (Toby Fox, 2015).

Undertale is full of charismatic characters, and the relationship between them and the player is the main reason why everyone who plays Undertale does it three times. In fact, Undertale has three different endings and outcomes for every boss fight. The three different endings are unlocked by acting differently to NPCs and bosses; and they are acknowledged as pacifist-run (not killing anyone), genocide-run (killing everyone), and neutral-run (which is in the middle between the pacifist and the genocide).

One of the most important boss fights of the whole game is an optional one that is unlockable only by doing the genocide-run (killing all the monsters, bosses, and characters you find in your way). After you killed nearly anyone in the game, you unlock the fight against ***Sans*** in the so-called *judgment room*. Sans breaks all the good design rules we listed earlier; in fact, he's a cheater, and his boss fight is very unfair. He deals damage over time (DOT), so while you touch his attacks, he constantly damages you until you stop touching them. This is one of the things that usually video games avoid. In fact, most video games allow the enemies to damage the player on contact just once. Right after the damage, the player starts blinking and becomes invulnerable for a second (you can see this in games, e.g., Super Mario Bros.) so that they can escape the situation and avoid further damage. This is a principle totally ignored in Sans' boss fight. He continues to damage the player constantly.

Another thing that makes Sans an unfair boss is the fact that he attacks you without announcing his attacks. Those attacks are super-fast and hard to detect and avoid. Sans also mocks you and suggests you to give up and don't come back again, leveraging on your feelings and your common sense to stop fighting. You need a lot of training and determination (and to remember the attacks by heart) to beat him and win the fight.

This is the point of the whole game: determination. All the story is about this concept. The characters you encounter in Undertale have very interesting and complex stories about friendship, love, loneliness, hope, sadness, fear, and sorrow; but they are all associated by the determination to not give up, and the most determined character in this world is you: the player.

This concept is explained very exhaustively in all the game, and you get to know all the characters and love or hate them. You start to internalize the reasons why you do things, and you play firstly by choosing your own path, then by doing exactly what is asked of you (e.g., completing the three runs) but still living the experience as it was your choice (even if it's not).

The determination mantra is repeated for all the game, and you will eventually start to make it yours and believe that all you need to win everything is determination. So you don't step back, even if this means facing an unfair fight or killing a character that in a previous run was your friend.

Undertale and Sans in particular can teach any game designer a huge quantity of lessons on how to design charismatic and fun bosses, but probably the most important lesson is how to leverage on the player's need of social relatedness to boost their motivation to play and their immersion. Sans does this by trying to deny the freedom to act from the player by playing rough and trying to make vain something for which the player worked so hard (the genocide). This pushes the player to not give up and face the challenge having fun even if it looks like it's impossible. It's challenging, it's fun, and it's memorable even if it's unfair because the player cares about the reasons why he's fighting. This is a golden lesson to keep in mind!

How can we use this?

In Space Gala, we created a fairly simple boss fight that is not very comparable with the Draygon, Sans, or any other boss fight we listed earlier.

Our one-eyed space-monster may be a bit scary (is it?), but it lacks the pathos. The only thing that really does good is to reiterate the two major concepts of Space Gala's gameplay: dodging and shooting. And this is good! Since, as we said, a good boss fight is one that teaches or reinforces gameplay mechanics (remember the Armos Knights in A Link to the Past?). In fact, in order to defeat our one-eyed boss, you have to dodge first the bullets and then the spawning UFOs and to continuously shoot at it. It's not a really memorable boss fight though. Why? And how can we make it better and more enjoyable?

Well, for example, we could introduce a new behavior. Maybe the monster feels fatigued following the player going around the screen, and when it's too tired, it can maybe close its eye and stop both spawning UFOs and shooting so that the player can clear the area destroying all the UFOs before the monster opens the eye again. Or maybe we can play on the narrative side of the game and give that monster a purpose or a meaning; maybe it's the chief of the vanguard of the aliens, and dispatching it, we can delay the invasion!

There are a lot of ways to improve this boss fight. Right now, you have a basic project on which you can build and improve concepts and mechanics. You can create new tools, weapons, mechanics, or narrative expedients to enhance the gameplay and the game experience.

Designing a game experience is the most delicate phase of making games and the one that truly decides if the game is just a pastime or an exciting and memorable experience.

So far, from our dissertation, we can say that a fun boss fight should challenge the player and a good one should teach or reinforce a gameplay mechanic. It should leverage on the player's emotions and psychological needs, like the need to relate with the game world and its characters and the meaningfulness of the player's actions. The player should be able to choose their own style and pursue their objective without feeling forced on narrative or gameplay rails or having to deal with characters without charm that makes everything flat and uninteresting.

To conclude, a good boss fight should always maintain the player immersed in the flow of the gameplay and give them a reason to keep them playing – may this be a narrative motivation or a fun gameplay. If possible (and if necessary), a good gameplay should act as a final test on some skills and mechanics that are needed to be mastered to develop competence in the gameplay.

Tip You should try to think about all the most interesting and memorable boss fights you found in your gaming experience and ask yourself: What makes this boss fight memorable? Asking and answering these questions will help you to develop a sort of awareness in terms of good game design and will allow you to create better game experiences.

Try to use this knowledge to improve Space Gala and its boss fight!

In the next chapter, we will start a new adventure exploring the platformer genre, which is one of the most important. We will create a single-screen platformer that will be extended to a scrolling platformer in Chapter 9 – just like we did with Space Gala. Exploring the huge world of 2D platforming will give us the opportunity to learn some very important concepts of game design and development like gravity, jump, 2D movements, special platforms, power-ups, different kinds of enemies, and so on.

That's going to be a lot of fun!

CHAPTER 8

Single-Screen Platformer

In the previous chapters, we built from scratch a couple of shoot ‘em up games basing our design on the classics of the genre. We got inspiration from giants like Galaga, Space Invaders, and Ikaruga. We explored the genre in its entirety and its evolution through the years learning precious lessons about how to design and develop a good and fun STG. We studied and reproduced important and iconic features that shaped the genre, and we improved on our own design from chapter to chapter. Finally, we talked extensively about boss fight design, and we saw some important cases from the industry.

In this chapter and the next, we are going to follow the same creative process by creating a basic game and improving it following the historical evolution of the 2D platformer game genre.

In this chapter, we will create a single-screen platformer taking inspiration from games like Space Panic (Universal, 1980), Donkey Kong (DK; Nintendo, 1981), Pitfall! (Activision, 1982), Lode Runner (Brøderbund Software, 1983), The Fairyland Story (Taito, 1985), Mario Bros. (Nintendo, 1985), and Bubble Bobble (Taito, 1986). Our game will be a single-screen platformer in which the player has to collect items scattered in the level moving between the platforms by jumping and climbing ladders while avoiding enemies and get to the end of the level.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_8

With this project, we will have the possibility to understand the nature and the evolution of the genre while learning how to create from scratch common game mechanics like jumping, climbing ladders, creating power-ups and items, and managing a side-scrolling camera. We will then focus our attention on the design of the genre, what's important, and how to make it fun. Fasten your seatbelt and get ready, once again, to make games!

When you think about video games, probably the first idea that comes to your mind is platformer games – those games where you control a character that has to walk through the whole level from one side of the screen to the other picking items, crushing enemies, and saving princesses.

Platformers are one of the most important and the oldest game genre of the history of the medium. It's actually a sub-genre of the action game genre; and even if now it's synonym with jumping on platforms (so much that jumping on platforms is actually known as platforming), in the beginnings of the genre, it was mostly about climbing ladders and trekking through small single-screen levels.

We are going to study platformers and not action games in general because it's a singularity in the video games history. In fact, it's the only sub-genre that is constantly on the frontline of the gaming experimentation. Always evolving, platformers defined the various gaming eras by introducing new iconic gameplay elements – a totally opposite approach compared to STG games!

There are a lot of platformer games that revolutionized the video games industry! For example, Super Mario Bros. introduced side-scrolling and metamorphosis; Sonic the Hedgehog added (a kind of) physics to movements and platforming; Tomb Raider created the first worldwide recognized female heroine; Crash Bandicoot mixed traditional platforming with plenty of mini-games; Super Mario 64 set the de facto standard on 3D platforming with cutting-edge camera and movement controls; and Prince of Persia: The Sands of Time revolutionized the genre with its rewind system. Even now, the platformer genre is a vibrant one, full of revolutionary ideas and always dragging the gameplay evolution forward.

It all started with Space Panic (Universal, 1980), an arcade game in which the player had to dig holes in the ground to trap the enemies and then hit them with the shovel to kill them off. The possibility to move between the platforms using ladders made this the first platformer game of the history.

Space Panic didn't feature neither jump nor screen scrolling. At the time, jump and gravity were two complex concepts to add to a game, and screen scrolling was unaffordable in many cases for technological reasons.

One year after the release of Space Panic, Nintendo published the game that started not one but two IPs: Donkey Kong (Nintendo, 1981).

DK was so important in many ways! Firstly, it was the beginning of both Donkey Kong and Mario; secondly, it was the game that everybody recognizes as the true first platformer. In fact, even if Space Panic was the first, the one that everybody remembers is Donkey Kong. This is probably because it implemented all the determining characteristics of the 2D single-screen platformer like jumping to move between platforms, ladders, power-ups, and obstacles. From Donkey Kong, the platformer genre became more about jumping than climbing ladders.

1982 saw the rise of one of the most loved platformers of all times: Pitfall!

Pitfall! was truly a revolution in the gaming industry! It featured a very realistic (for the time) graphics depicting man in the jungle and a very fun and immersive gameplay around exploration and adventuring in the dangers of nature.

Apart from those three singularities, the first half of the 1980s was mostly made of similar-looking single-screen platformers that were about killing all the enemies in the room to move forward or going from point A to point B. The first formula was overabused by Taito that made a huge quantity of similar games with very little differences in gameplay from The Fairyland Story to Bubble Bobble; the second formula was the most used by Nintendo with games like the Donkey Kong saga and Mario Bros. in 1985.

It's interesting to dedicate some words to Mario Bros., though, because it introduced a couple of interesting features. In fact, this first prototype of the adventures of the Italian plumber featured complex interactions with the enemies that could be jumped on making them fall on their back and then could be kicked (a mechanic that became iconic since then) and also a nice cooperative gameplay that became the de facto standard in single-screen platformers.

We now have a clearer idea of what are the main characteristics of the genre, but what is important to a platformer to be fun? What do we want to take from those classics to shape our game?

The game we are going to create will be a single-screen platformer. This choice has a double benefit: it's easy to make (which is good, since you are learning), and it's also very interesting from a game design point of view! Indeed, it's not easy to create an interesting gameplay and level in a space so small! But it's when you have to face hard challenges that you can really learn and test your design skills!

The game will use both the jumping and ladder climbing systems. Since we have a limited space, it's important to give the player more options to move and the level designer more elements to create interesting levels.

The game, as we said, will be about collecting items. The player should collect all the objects in the room to go to the next level.

The player can win the game by collecting all the items for all the levels, and can lose it dying three times.

As this has become our tradition, let's put this on a game design document that we will use while creating the game as a reference!

Cherry Caves

Cherry Caves is a single-player 2D platformer game about collecting items in a set of labyrinthine caves. It's inspired by the classics of the first half of the 1980s – the single-screen platformer era.

Story and setting

In 20XX, the Earth saw the extinction of cherries that now can be found only by extracting them from old snacks and illicit traffics. Cherries are now even more expensive than the most precious metal in the world and the purest diamond.

You heard the voice that cherries are naturally growing in a cave populated by strange alien creatures in the form of big colored balls.

Fearless and with a great desire of tasting cherries and becoming rich by reselling them, you decide to adventure in the deeps of the Cherry Caves.

Gameplay

The goal in Cherry Caves is to collect all the cherries in every level and get to the exit.

While trying to collect all the cherries, the player will be put in danger by strange enemies bouncing around the level.

Victory condition

Each level is completed by reaching the goal represented by a yellow star (Figure 8-1).

Figure 8-1.

The level can be completed only by collecting all the cherries in it (Figure 8-2). Until that moment, the star will remain grayed out (Figure 8-3).

Figure 8-2.

The player can lose the game by getting hit by the enemies three times. Every time the player gets hit, they restart from the starting point of the current level.

Note The player has only one possibility of getting hit before restarting the level, because levels are very small and to have more than one possibility to get hit would have lowered the challenge too much.

Controls

Controls are pretty standard with the arrow keys to move and the spacebar to jump.

Left: Move left.
Right: Move right.
Up: If on the ladder, climb up.
Down: If on the ladder, climb down.
Spacebar: Jump.
Esc: Open/close menu.

Enemies

Enemies, from a coding point of view, are very similar. They don't attack; they just move following patterns. The design of the level is what makes them a threat to the player. Anyway, just to differentiate a bit, there are two different enemies that you can find in the caves:

- **Purple Balls**: These strange balls only bump following a vertical pattern. They are found usually in small spaces blocking the way to some juicy cherry treasure.
- **Green Balls**: These strange green balls follow more complex and longer patterns and can give a serious headache to an unprepared explorer.

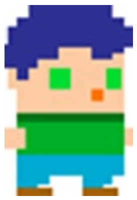
Assets

To create Cherry Caves, first of all, we need some assets. So open up GameMaker Studio 2, and let's create a bunch of new resources that we will use to build our game!

Sprites

To create a new sprite, right-click Objects in the Resources sidebar and select Create Object. Do this for all the following sprites!

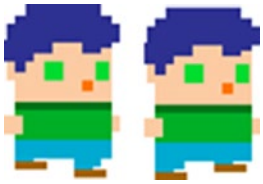
spr_player_idle



Size: 50 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle

spr_player_walk

This is an animated sprite. You can make one by simply creating a single spr_player_walk sprite and adding two images to it using the *Import* key or by creating images individually in the Sprite Editor.



Size: 50 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

Speed: 4

spr_player_climb

Size: 50 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_block_red

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_block_brown

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_ladder

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_ball_purple

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_ball_green

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_cherry

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_goal

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

Fonts

To create a font, right-click Fonts in the Resources sidebar and select Create Font.

fnt_score

For this font, I am using the preinstalled Consolas font with style regular and size 20. If you are not a Windows user or you don't have

Consolas installed on your computer, just pick the font you like better to show the score in your game.

Sounds

snd_menu: This sound effect will be played when opening the menu and moving the cursor.

snd_goal: This one will be played when touching the goal.

snd_cherry: This is played when picking a cherry.

snd_jump: A sound effect played when jumping.

snd_damage: This effect will be played when taking damage.

How to create a hero

Berry will be our first recognizable hero! We already made three games, but none of them featured a recognizable character. Berry, with his blueberry hair and chibi style, aims for that title! Let's help him by creating him!

To create Berry, right-click Objects in the Resources sidebar and select Create Object. Name it obj_player and select spr_player_idle as its default sprite.

A platformer is not such if you can't move! So let's give Berry the ability to walk!

The concepts we are going to use to let the player walk are just the same that we applied to Space Gala. Anyway, this time we will not use separated events for every key, but we will manage everything in the Step event by using GML.

Before we dive into the coding in the Step event, we need to create some useful variable that we will need later.

Create a new Event by clicking Add Event and selecting **Create**.

In this event, we will create the usual spd variable. You already know that one; it will store the value of the speed at which we want the player to move.

So, easy as it sounds, add this line to obj_player's Create event:

```
spd = 4;
```

Ok, now we can concentrate on the main event of this object: the step event.

Create a new step event by clicking Add Event ➤ Step ➤ Step in the Object Editor.

To check whether the player is pressing a key, without using the dedicated events, we must use a family of dedicated functions that I will shortly explain later.

keyboard_check(key_code): This function checks whether the key with code key_code is held down or not.

It takes a keyboard code as input and returns a Boolean value that tells whether the player pressed or not the key.

The key_code is a number that represents a keyboard key for the system. GameMaker provides you with all the key codes you may need. Check the documentation for more details.

keyboard_check_pressed(key_code): This function checks whether the key with code key_code has just been pressed. Just like keyboard_check, it takes as input a key code and returns a Boolean telling the programmer if the key was actually pressed or not.

keyboard_check_released(key_code): This function checks whether the key with code key_code has just been released. It takes as input a key code and returns a Boolean telling the programmer if the key was actually pressed or not.

For our purpose, we will use keyboard_check to check if the player pressed the arrow keys to move, because we want the player to continue moving while holding down the key.

So let's write up some code in that empty Step event:

```
 var keyleft = keyboard_check(vk_left);
 var keyright = keyboard_check(vk_right);

```

```
4   var move = keyright - keyleft;
5   hsp = spd * move;
6
7   if ( move != 0 )
8   {
9       image_xscale = move;
10      sprite_index = spr_player_walk;
11  }
12  else
13  {
14      sprite_index = spr_player_idle;
15  }
16
17  x += hsp;
```

Lines 1–2: We use keyleft and keyright to store the result of the function calls that check if the player pressed the left or right keys.

Lines 4–5: We declared a new temporary variable called move. In this variable, we are storing the difference between the values of keyleft and keyright (note that the numeric value of true is 1 and the value of false is 0) so that if we are pressing right, move is equal to 1 - 0 = 1 and when we are pressing left, move is equal to 0 - 1 = -1. You will see in a bit how this can be useful.

hsp represents the horizontal speed. That means the velocity at which the player is moving on the X-axis. When hsp is 0, the player is not moving left or right; when it's greater than 0, the player is moving right; when it's less than 0, the player is moving left.

Since we are very smart, we are using move to understand the direction toward which the player wants to move by multiplying the value of the speed to move. This will save us an if statement.

Lines 7–16: Here we check that the player is moving by checking that move is different from 0. If the player is moving, we want the sprite to face the right direction. To do that, we are using the image_xscale property which tells GameMaker to draw the sprite normally (when image_xscale equals 1) or flipped (when image_xscale equals -1). We could have accomplished this also by creating two sprites, one for the left and one for the right; and you should still do this if you want your character to have two different looks when looking left and when looking right, but this is not the case.

We also want the player to change the sprite accordingly to the fact that it's moving or not. This is why we change the sprite to spr_player_walking when move is not zero; else, we change it to spr_player_idle.

Line 17: At the end, we add the value of hsp to x, effectively moving obj_player on the X-axis.

Now open up room0 and drag obj_player and drop it in the middle of the level. Run the game, and you will see that Berry moves left and right (Figure 8-3), but it doesn't stop and goes outbound. Let's fix this!

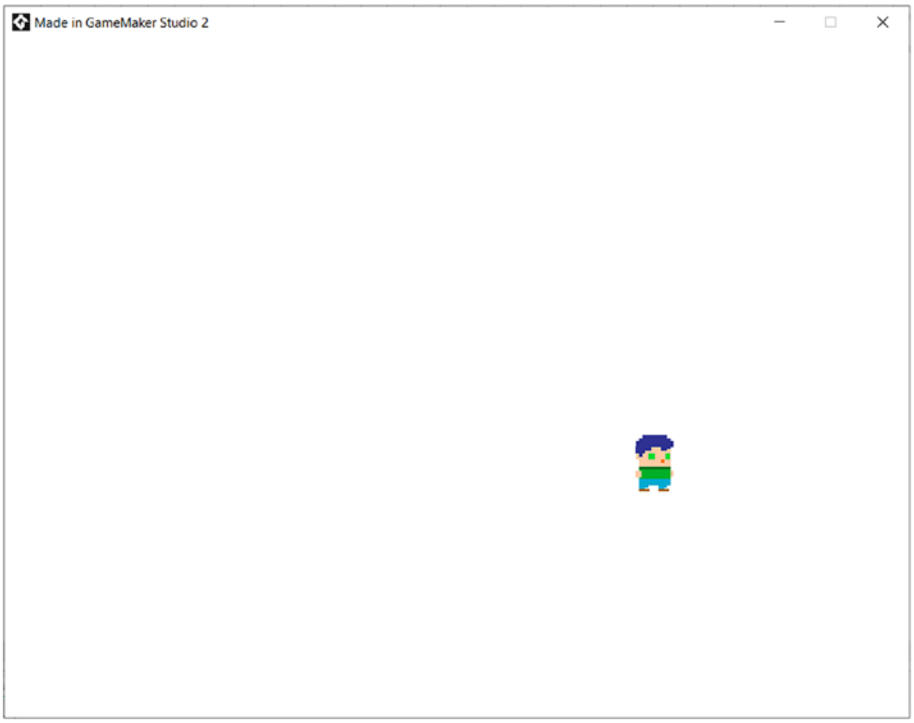


Figure 8-3. *Berry moves left and right, but doesn't stop when the room ends*

Setting the boundaries

In Space Gala, we defined where the player could go and where they could not by setting some boundaries related to the dimension of the camera. We could do this also this time, but instead, we will create an object that we will use to block the way of our player. This block will be used both as a delimiter and as the ground on which the player can walk. Having the blocks to also delimit the screen boundaries allows us to better define the level from an aesthetic point of view and to save us some lines of code (which is always good).

In Cherry Caves, we will use two kinds of blocks that from a coding perspective are the same thing, but they feature two different sprites (spr_block_brown and spr_block_red). This will help us in the task of making levels feel more colorful and different from one another. Often this technique was used in blockbuster games like Taito's Bubble Bobble or even other game genres like Namco's Pac-Man.

To have those two kinds of blocks means that we should create two different objects; but since we want them to behave in the same way, we should make a third object that we will use as the parent, so that we can code all the logic around that one object and have it inherited to the other two.

Figure 8-4 shows the hierarchy that we will use with the three block objects.

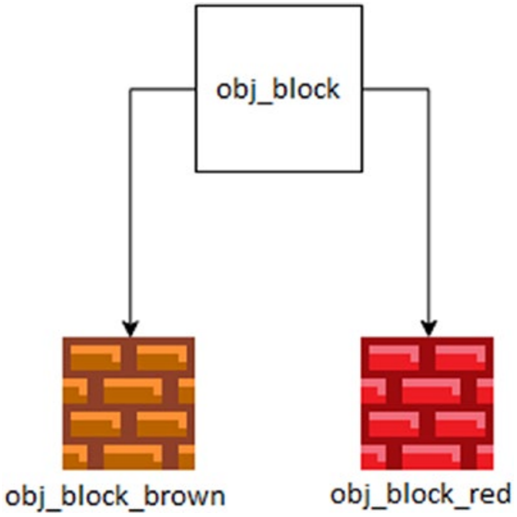


Figure 8-4. *Blocks hierarchy*

Let's create the three objects by right-clicking Objects in the Resources sidebar and selecting Create Object. Name the first obj_block and assign no sprite to it, name the second obj_block_brown and give it the spr_block_brown sprite, and then give the third one the name of obj_block_red and the spr_block_red ground.

Add obj_block as parent for both obj_block_red and obj_block_brown. To do so, open up their Object Editor and click Parent and choose obj_block.

Anyway, we won't use any of those objects to program the logic. We will continue using the obj_player's step event.

So open up obj_player's Object Editor and select the Step event. Add to the bottom of the event these lines:

```
 if ( place_meeting(x+hsp, y, obj_block) )
   {
         while ( not place_meeting(x+sign(hsp), y, obj_
         block) )
```

```
4           {
5                x += sign(hsp);
6           }
7           hsp = 0;
8    }
9    x += hsp;
```

Line 1: place_meeting(x, y, object) moves the current instance to the provided X-,Y-coordinates, then checks if there's a collision with an object, and finally moves the instance back to its original coordinates.

It's a function to check collisions just like the dedicated event we used in Space Gala to check collisions between the enemies, the player, and the bullets.

We are checking collisions in the step event instead of using a dedicated event, because we need to tweak a bit the collision checking. In fact, in line 1, we are not just checking the current coordinates at which obj_player's instance is; we are checking if we will collide with an instance of obj_block (or its children) after advancing *hsp* pixels on the X-axis.

Lines 3–7: If obj_player is going to collide after *hsp* pixels, since we want a pixel-perfect collision, we start a loop in which we add 1 pixel in the right direction (using the sign function, explained in detail later) on the X-axis to obj_player and check if it collides with obj_block (or its children). Until it doesn't, we continue adding 1 pixel on the X-coordinate. When obj_player eventually collides with obj_block (or its children), we stop doing this and set hsp to 0 so that obj_player won't move anymore.

Here we created a pixel-perfect collision system to use obj_block instances as walls that block horizontally the player.

Line 9: We moved the line in which we update x to the end of the code, so that all the modifications we are making with the collision checking have effect on *hsp* and on the value of x.

Note In the previous code, to check for the sign of hsp, we used the sign function.

sign(num) is a function that given a number num returns 1 if num is greater than 0, -1 if it's less than zero, and 0 if it's equal to 0.

Now open up room0 and add two instances of obj_block_brown or obj_block_red to the left and right side of obj_player instance. Running the game, you can verify that obj_player will collide with the blocks being unable to pass over them (Figure 8-5).

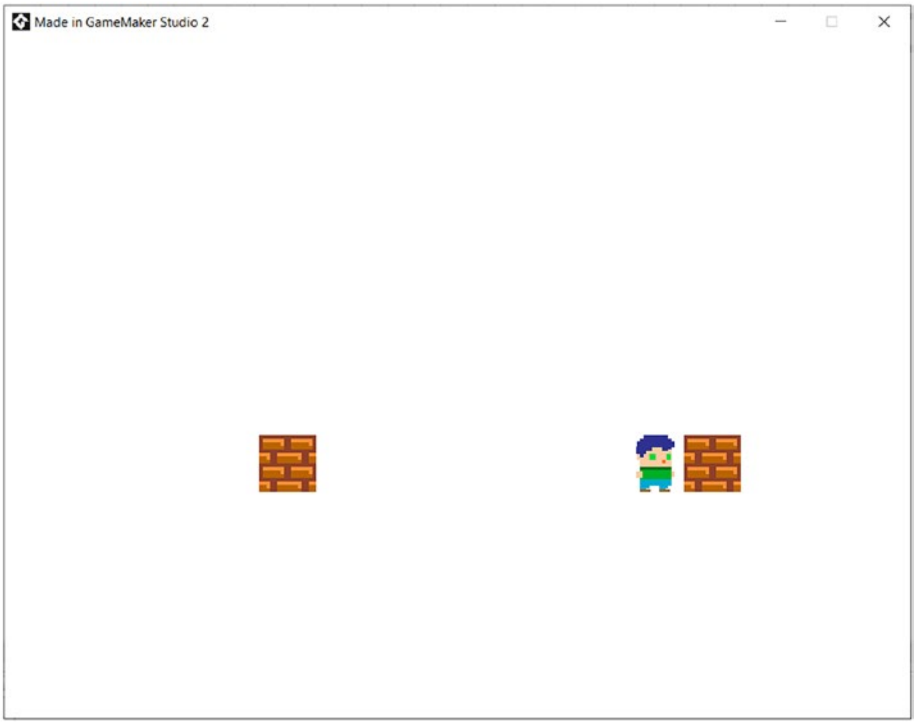


Figure 8-5. *Now we can block Berry using children of obj_block (in this case obj_block_brown)*

Something is not right … Berry shouldn’t be allowed to float in the air; he should be subject to gravity.

GameMaker comes with its own physics system that can simulate a fairly accurate physical system. It has also a gravity system, and it’s very good if you want to make a game like Angry Birds. That’s not our case. In fact, in a platformer, just like in a shmup, it’s very important for the system to have pixel-perfect movements and a very arcade playstyle. Platformer gamers don’t want to slip over a platform because of friction and acceleration, they don’t want to have an accurate physics system when colliding, and so on. What is important in a platformer game is precision, and this can be achieved only by moving things the old way. That’s why we won’t use GameMaker physics system, but instead we are going to learn how to create an arcade gravity system in the old-school way.

Everything that goes up comes down

Gravity, in a game, is often a value that adds constantly to the y-position of the player’s avatar, instead of being a physical force that constantly increases the falling speed of an object. While an object is not on a blocking surface, it falls down, and its position is updated by adding to its Y-coordinate a fixed amount of pixels equal to the gravity value.

We will use a variable to represent gravity and to update the player’s Y-coordinate. We will then apply the same principle we used for the horizontal collision with obj_block, on the Y-axis, so that obj_player will be blocked by obj_block (or one of its children) and it will stop falling down. We will track if the player is grounded or not by using a Boolean variable.

So, first of all, let’s add grv and grounded to the Create event of obj_player:

```
grv = 0.5;
grounded = false;
```

And now let's open up again obj_player's step event and add some code at the bottom of the file:

```
vsp = vsp + grv;
1   if (place_meeting(x, y+vsp, obj_block))
2   {
3       while ( not place_meeting(x, y+sign(vsp), obj_block))
4       {
5           y += sign(vsp);
6       }
7       vsp = 0;
8       grounded = true;
9   }
10  else
11  {
12      grounded = false;
13  }
14  y += vsp;
```

Line 1: Here we are doing the same thing we did with hsp; but instead of adding speed and multiplying it by the direction we want to move to, we just add the gravity value to vsp, which is the vertical speed.

Line 2: Here we check if obj_player will collide with an instance of (or derived from) obj_block after vsp pixels on the Y-axis. This is the same concept we applied to the previous piece of code to calculate horizontal collisions with obj_block.

Lines 4–9: Like in the previous code, when we see that adding vsp pixels to the Y-coordinate of obj_player will result in a collision with obj_block, we update y by adding 1 pixel in the direction of vsp (just like we did with x and hsp). When obj_player finally reaches obj_block, vsp gets set to 0 and grounded to true. In fact, when obj_player perfectly collides with obj_block, this means that Berry touched the ground, so he's grounded.

Line 13: If obj_player is not touching obj_block anymore, grounded gets set to false, meaning that the player is floating and should be affected by gravity (hence, vsp shouldn't be set to 0, like we do in case of collision (line 8).

Line 15: After all these checks, the Y-coordinate gets finally updated by adding to it a vsp number of pixels effectively moving (or not) the player's avatar.

Open up room0 and add some instances of obj_block_brown all around the borders of the room and put obj_player in the middle of it and run the game (Figure 8-6).

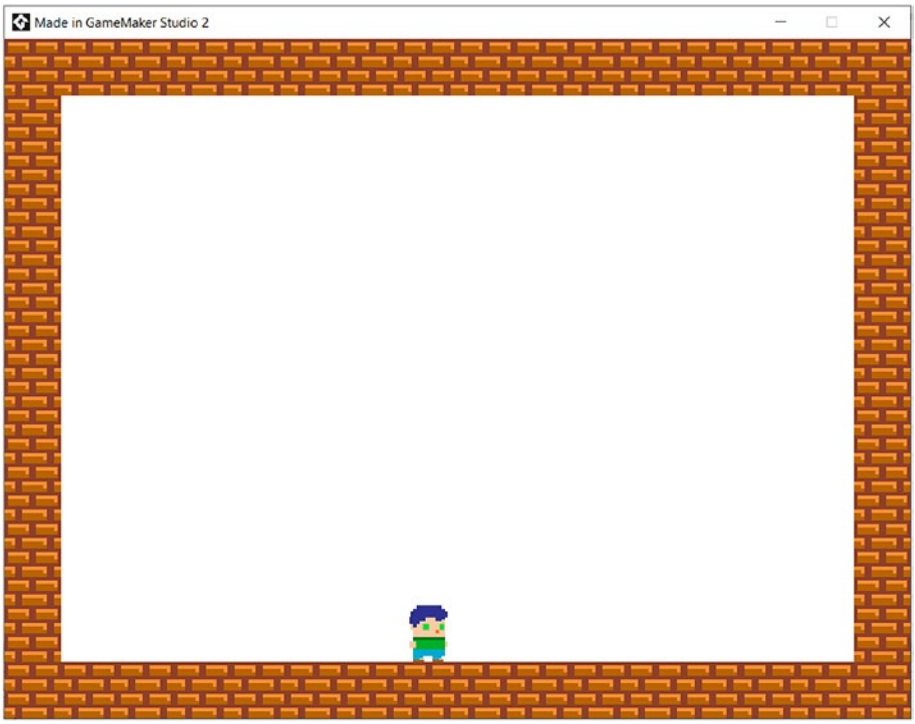


Figure 8-6.

You will see that Berry will fall down to the ground and can move around but can't go over the blocks.

A gravity system is not very useful if you cannot jump! That's the next thing we are going to cover!

Get a jump on!

Jumping is actually way simpler than you could think. In fact, if falling means adding pixels to the Y-coordinate, jumping means the opposite: subtracting pixels from the Y-coordinate. This is what we do when jumping. We will check whether the player pressed the spacebar key, and if they did, we subtract a fixed value to the Y-coordinate.

First of all, we should add to the Create event the jspd variable:

```
jspd = 10;
```

Then open up obj_player's Step event, and let's add the logic we talked about by adding this line at the top of the event

```
var jumping = keyboard_check_pressed(vk_space);
```

and this code at the bottom of the Step event, just before updating the Y-coordinate:

```
if ( grounded and jumping )
{
    vsp = -jspd;
    grounded = false;
    sprite_index = spr_player_idle;
}
```

That's it! Now running the game, you should be able to see that pressing the spacebar, the player will be able to jump and fall back on the ground. That looks more like it (Figure 8-7)!

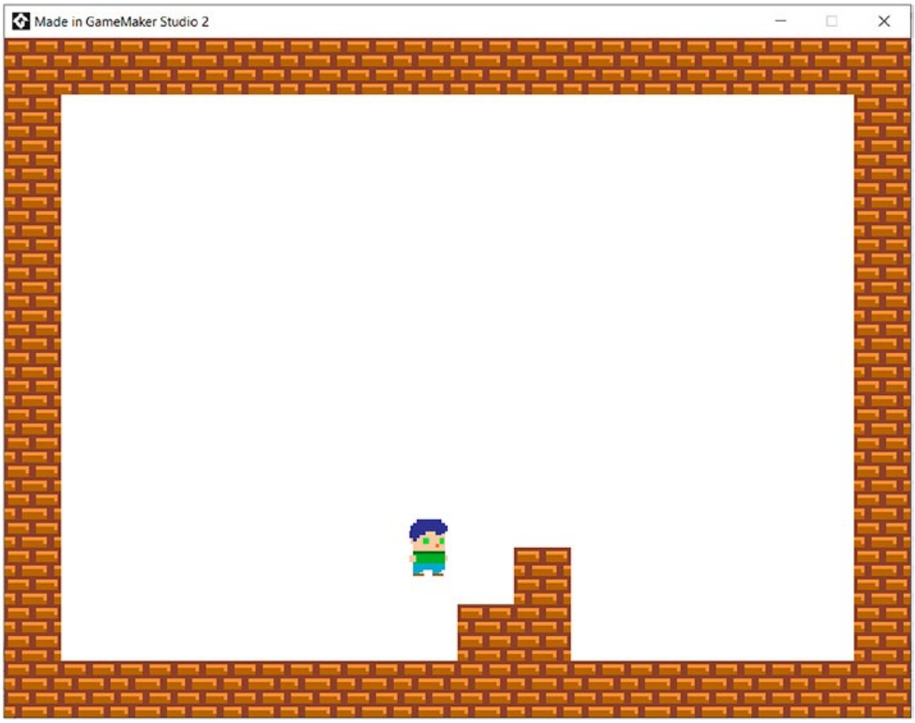


Figure 8-7.

Climbing the ladder

We are halfway from finishing our game. We want to add yet another feature! We want the player to be able to climb ladders which is a very important feature in single-screen platformers.

To manage gravity, we will use a climbing controller variable that will be switched to true when the player is on a ladder and to false when they're not.

This variable will allow us to better manage animations and the gravity.

We also need to check if the player is pressing the up and down arrow keys, because we want the player to climb the ladder only when he's standing in front of it and pressing the up or down button. By doing this, the player will change the status of the climbing variable; and from that moment, pressing the up or down key will mean moving up or down, not being affected by gravity anymore. Gravity will return to affect the player position after they climbed down the ladder.

So, first of all, let's add the climbing variable to the obj_player Create event:

```
climbing = false;
```

We initialize it with the false value because we assume that every level should not start being on a ladder.

We are going to change some bits of code we wrote before and add some more, so let's review the new code and rereview the old one that should be inside the step event:

```
var keyleft = keyboard_check ( vk_left );
var keyright = keyboard_check ( vk_right );
var keyup = keyboard_check ( vk_up );
var keydown = keyboard_check ( vk_down );
var jumping = keyboard_check_pressed ( vk_space );

var move = keyright - keyleft;
var vmove = keydown - keyup;
hsp = move * spd;
vsp = vsp + grv;

 // WALKING
 if (move != 0)
 {
    image_xscale = move;
    if ( grounded )
    {
        sprite_index = spr_player_walk;
    }
}
else
{
    if ( not climbing )
    {
        sprite_index = spr_player_idle;
    }
}

// JUMP
if ( grounded and jumping )
{
    vsp = -jspd;
    grounded = false;
    sprite_index = spr_player_idle;
}

// CLIMBING
if ( place_meeting(x, y+1, obj_ladder) )
{
    if (vmove < 0) or
        (vmove == 0 and climbing) or
        (vmove > 0 and place_meeting(x,y+sprite_height,
        obj_ladder))
    {
        climbing = true;
    }
    else
    {
        climbing = false;
    }
}
else
{
    climbing = false;
}

if ( climbing )
{
    vsp = vmove * spd;
    sprite_index = spr_player_climb;
}

// HORIZONTAL COLLISION WITH BLOCKS
if ( place_meeting(x+hsp, y, obj_block) )
{
    while ( not place_meeting(x+sign(hsp), y, obj_block) )
    {
        x += sign(hsp);
    }
    hsp = 0;
}
x += hsp;

// VERTICAL COLLISION WITH BLOCKS
if ( place_meeting(x, y + vsp, obj_block ) )
{
    if (not ladder)
    {
        while ( not place_meeting(x, y+sign(vsp), obj_
        block) )
```

```
79          {
80              y += sign(vsp);
81          }
82          vsp = 0;
83          grounded = true;
84      }
85  }
86  else
87  {
88      grounded = false;
89  }
90
91  y += vsp;
```

Lines 1–5: Here we check the keyboard inputs using the keyboard_check functions instead of the dedicated events.

Lines 7–8: move and vmove just decide the horizontal and vertical direction toward which the player should move based on the player's input.

Lines 9–10: hsp and vsp are, respectively, the horizontal and vertical speed variables. They count how many pixels the player should move horizontally and vertically. This value will be affected by later calculations, but it's also dependent on the player's input checked at lines 1–5.

Lines 13–20: Here we check if the player is moving left or right. If they are, we flip the avatar in the right direction using image_xscale (note that a value of 1 means that the sprite is left at the original orientation, while -1 flips it in the opposite direction); and then, if the player is grounded, we change the sprite of obj_player to spr_player_walk that shows a walking animation. We change the sprite only if the player is grounded, because we don't want Berry to look like he's walking in the air, when he's jumping or falling.

Lines 21–27: If the player is not moving left or right, we check that he's not climbing, and then we change obj_player's sprite to spr_player_idle that shows Berry idle.

Lines 30–35: This is our jumping code. We check if the player is grounded and jumping (hence pressing the spacebar), and if they are, we make obj_player go up by changing vsp to a negative value of -jspd. Then we set the grounded variable to false, since we are not anymore on the ground; and then we change the sprite of obj_player to spr_player_idle, so that we have the illusion that Berry is jumping feet together.

Lines 38–54: This is new! Here we manage the climbing mechanic. First, we check if obj_player is colliding with an instance of obj_ladder. We do it by moving testing at coordinates x,y+1 because we want to include the case in which the player has the ladder just under their feet.

If this condition is true, it means the player is touching a ladder or has it at their feet, so we must determine the various cases that can trigger the climbing mode of the player's character.

We said that we want obj_player to keep climbing the ladder when

- The player is pressing the up key (vmove < 0) in front of a ladder.
- The player is pressing the down key (vmove > 0), and there is a ladder just under obj_player. In that case, obj_ladder is at coordinates x, y+sprite_height (just one block under the player).
- The player is not moving up or down, but they already are in climbing mode (they were just climbing the ladder and stopped before reaching the ground).

If one of those three conditions (lines 40–42) is true, we set the climbing variable to true; else, we set it to false.

Finally, if the player is not touching the ladder at all (line 51), we set climbing to false.

Lines 56–60: If the player is on the ladder (after the calculation we made in lines 38–54), we don't want to be affected by gravity anymore, so we just set vsp to a value that's equal to the player's movement speed (represented by spd) times the direction in which the player wants to climb (represented by vmove). We also change the sprite of obj_player to spr_player_climb.

Lines 63–71: This is our code to check the collisions with obj_block on the X-axis (moving horizontally). Nothing has changed, so we can move on.

Lines 73–89: This is the code that checks the collisions with obj_block on the Y-axis (moving vertically). Also, this one is unchanged, so there's nothing to add.

Controlling the game flow

We have a working platforming system, but this alone doesn't make a game.

According to the GDD, Cherry Caves is a game involving the collection of cherries for all the levels. Those cherries will unlock the exit of the game. The collection of cherries is made harder by enemies that will block Berry's way. When he gets hit by those enemies, the level will restart. Berry can be hit only three times. After that he will die for good, and the game will be over.

We will manage the game flow using the game state system we used also in Space Gala. Since Cherry Caves' flow is pretty simple, we will borrow most of the states from Space Gala. Figure 8-8 shows the Cherry Caves' game flow as a finite-state machine. Let's describe them:

- **playing**: This state represents the playing phase of the game. The game is ready to get inputs from the player, and every object acts normally.

- **paused**: In paused state, every object gets deactivated; and the player can access the menu to resume, restart, or quit the game.
- **dead**: The player has been hit by an enemy. As a consequence, they lose a life and restart the room from the starting point.
- **gameover**: The player has been hit for the third time. The game is over, and the player can only restart the game or quit.

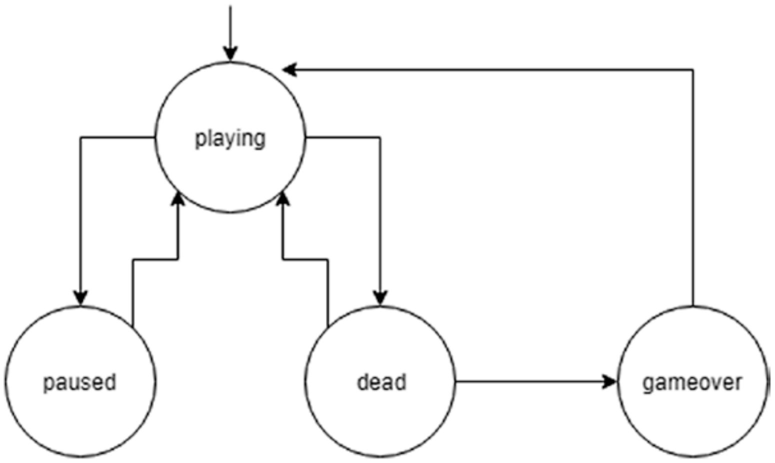


Figure 8-8.

Let's explore the various transitions:

- **From playing to paused**: Triggered by pressing the Esc key. It opens the pause menu.
- **From paused to playing**: Triggered by pressing the Esc key. It closes the pause menu.
- **From playing to dead**: Triggered by being hit by an enemy. It restarts the room and subtracts one life.

- **From dead to playing**: Triggered by being hit by an enemy and not having finished all the lives. The room is restarted, and the status is set to playing again.
- **From dead to gameover**: Triggered by being hit by an enemy and having finished all the lives or by having completed all the levels. The game over screen is shown, and the player can decide to restart or quit the game.
- **From gameover to playing**: Triggered when the player, after a game over, decides to restart the game. It restarts the game from the first room.

Now we have a clearer idea of the game flow, and we can go back to coding.

To implement the game controller and manage all those states, we will use a game controller object, as we did in Space Gala.

Let's create the obj_controller and its Create event. Then add the following code:

```
enum states {
    playing,
    paused,
    dead,
    gameover
};
global.game_state = states.playing;

global.cherries = 0;
global.cherries_max = instance_number(obj_cherry);
global.startx = obj_player.x;
global.starty = obj_player.y;

```

```
14  options = [ "RESUME", "RESTART", "QUIT" ];
15  opt_number = array_length_1d(options);
16  menu_index = 0;
17
18  if ( room == room0 )
19  {
20    lives = 3;
21  }
```

Lines 1–7: We create the states data structure, as we did for Space Gala. Then we set up the starting state as states.playing.

Lines 9–10: cherries is the global variable that counts the number of cherries picked up by the player, while cherries_max counts the number of cherry objects present in the room by using instance_number function.

Note instance_number(obj) takes as input an object and returns how many instances of that object are present in the current room.

Lines 11–12: startx and starty are the starting position of obj_player in the level. We will use them to reset the player to the original position once they die.

We could have used the room_reset function to reset the whole level, but it's not optimal, since some global variables (e.g., score) are kept, while others (e.g., lives) are reset; so since the behavior of room_reset is not consistent, we avoid using it under this circumstance.

Lines 14–16: This is the code to set up the pause menu, as we already saw in Space Gala. The options are the same: resume, restart, and quit. Resume will resume the game, restart will restart it from the first room, and quit will close the application.

Lines 18–21: These lines set the number of lives to three when you start the game from the first room. When lives reach 0, the game is over.

Everything is set up, and the obj_controller is ready to manage all the mechanics we planned to implement.

Firstly, let's deal with the various states. When the game enters in the paused state, we want to stop everything and show the menu. Differently from Space Gala, this time we will manage everything concerning the logic inside the Step event avoiding to rely on keyboard management events. This will allow us to have more control on the code that will not be scattered around. You may prefer the modular style that GameMaker suggests, but it's really a matter of taste and personal preference. There's not a golden rule, for this aspect.

So let's create a Step event for obj_controller and put some code in it. As usual, we will comment the following code:

```
var esc_pressed = keyboard_check_pressed(vk_escape);
var enter_pressed = keyboard_check_pressed(vk_enter);
var move = keyboard_check_pressed(vk_down) - keyboard_
check_pressed(vk_up);

if ( esc_pressed )
{
    if ( global.game_state == states.playing )
    {
        global.game_state = states.paused;
        audio_play_sound(snd_menu, 1, false);
        instance_deactivate_all(true);
    }
    else if ( global.game_state == states.paused )
    {
        global.game_state = states.playing;
        instance_activate_all();
    }
}
```

```

if ( global.game_state == states.paused )
{
    menu_index += move;

    if ( move != 0 )
    {
        audio_play_sound(snd_menu, 1, false);
    }

    if ( menu_index < 0 )
    {
        menu_index = opt_number - 1;
    }
    else if ( menu_index > opt_number - 1 )
    {
        menu_index = 0;
    }

    if ( enter_pressed )
    {
        switch( menu_index )
        {
            case 0:
                global.game_state = states.playing;
                instance_activate_all();
                break;
            case 1:
                game_restart();
                break;
            case 2:
```

```
50                  game_end();
51          }
52      }
53  }
```

Lines 1–3: These lines just make some keyboard checks that we need later. We are using keyboard_check_pressed instead of the dedicated events.

Lines 5–18: When the escape key is pressed, if the active state is playing, we change it to paused, play a sound, and deactivate all instances using instance_deactivate_all; if the active state is paused, we change it to playing and reactivate all the instances in the room.

Note You can activate or deactivate all the instances in the game by using instance_activate_all and instance_deactivate_all.

instance_activate_all() activates all the instances in the room.

instance_deactivate_all(notme) deactivates all the instances in the room. It takes one parameter that says whether we want the current instance to remain active or not. If the parameter is true, the current instance (the one which called the function) will not be deactivated.

Lines 20–53: Here we check if the active state is paused. If it is, we access the functions of the menu that works exactly like the one in Space Gala. The only difference is that we added the keyboard control for the enter key in there using the result of the keyboard_check_pressed function stored in the enter_pressed variable (lines 38–52).

When the player chooses the first option (resume), all the instances get reactivated and the state changed to playing.

If the player chooses the second option (restart), the game gets reset by calling game_restart().

If the player chooses the third option (quit), we call game_end() and quit the application.

This code is the same we used in Space Gala and allows us to manage the simple menu we designed in the previous chapters to access the three main functions (resume, restart, quit).

To visualize the menu, we should now draw the various graphical elements using the Draw GUI event, as we did in Space Gala.

Let's create a Draw GUI event for obj_controller and add this code in it:

```
if ( global.game_state == states.paused )
{
    draw_set_color(c_white);
    draw_set_font(fnt_score);
    draw_text(room_width/2, room_height/2, "PAUSE");

    for( var i = 0; i < opt_number; i++ )
    {
        if ( menu_index == i )
        {
            draw_set_color(c_white);
        }
        else
        {
            draw_set_color(c_dkgray);
        }
        draw_text( 1200, 700 + 30 * i, options[i] );
    }
}
```

In the preceding code, when the active state is paused, we draw a PAUSE writing at the center of the screen, and we show the menu in the bottom-right corner.

We want to do something similar for the game over state. We want the screen to go black and show the menu with just the options to restart and quit the game, so let's open up again obj_controller's Step event and add this code to the bottom:

```
if ( global.game_state == states.gameover )
{
     instance_deactivate_all(1);
     menu_index += move;

     if ( move != 0 )
     {
          audio_play_sound(snd_menu, 1, false);
     }

     if ( menu_index < 1 )
     {
          menu_index = opt_number - 1;
     }
     else if ( menu_index > opt_number - 1 )
     {
          menu_index = 1;
     }

     if ( enter_pressed )
     {
          switch( menu_index )
          {
               case 1:
                    game_restart();
                    break;
               case 2:
```

```
28                          game_end();
29                }
30           }
31      }
```

Similarly to what we did for the paused state, we should add some code to the Draw GUI event to visualize the menu. The only difference is that at lines 6–13, we check if there are still lives: if it's the case, it means that the game is over because the player won, so we write a victory message; else, we write a game over message.

Let's write this code to the bottom of obj_controller's Draw GUI event:

```
1    if ( global.game_state == states.gameover )
2    {
3        draw_set_color(c_white);
4        draw_set_font(fnt_score);
5
6          if ( lives <= 0 )
7          {
8                 draw_text(room_width/2, room_height/2, "GAME
                  OVER");
9          }
10         else
11         {
12                draw_text(room_width/2, room_height/2, "YOU
                  WON!");
13         }
14
15       for( var i = 1; i < opt_number; i++ )
16       {
17           if ( menu_index == i )
18           {
19               draw_set_color(c_white);
```

```
            }
            else
            {
                draw_set_color(c_dkgray);
            }
            draw_text( 1200, 700 + 30 * i, options[i] );
        }
    }
```

HUD

It's important for the player to constantly be updated about some information like the number of lives remaining, the current score, and of course the number of cherries collected so far.

A simple way to manage it is to add all those information in the top of the window, creating a simple HUD with all that info in line.

The number of lives will be represented by a little version of spr_player_idle and a text showing the value of the lives variable; the number of cherries will be shown using a little version of the spr_cherry sprite and using global.cherries and global.cherries_max, while the score, as usual, will be a text showing the value of the score variable.

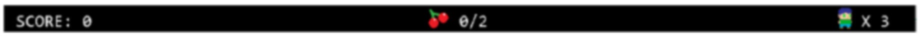


Figure 8-9.

The result will be the status bar in Figure 8-9, which is simple, but effective. All the information is available to the player, and every entry has a clear semantic.

The HUD will be created in obj_controller's Draw GUI event, as every GUI-related element. So let's open up that event and add these lines on the top of the code:

```
1   draw_set_color(c_black);
2   draw_rectangle(0, 0, room_width, 40, false);
3
4   draw_set_color(c_white);
5   draw_set_font(fnt_score);
6   draw_text(20, 10, "SCORE: " + string(score));
7
8   draw_set_color(c_white);
9   draw_sprite_ext(spr_cherry, -1, (room_width/2)-32, 20, 0.5,
    0.5, 0, c_white, 1);
10  draw_text(room_width/2, 10, string(global.cherries) + "/" +
    string(global.cherries_max));
11
12  draw_set_color(c_white);
13  draw_sprite_ext(spr_player_idle, -1, room_width-110, 20,
    0.5, 0.5, 0, c_white, 1);
14  draw_text(room_width-100, 10, " X " + string(lives));
```

Lines 1–2: This is new, but easy. We are drawing a rectangle using the draw_rectangle function.

Note You can draw shapes on the screen by using some specific built-in functions. In the preceding code, we used one of those functions to draw a rectangle.

draw_rectangle(x1, y1, x2, y2, ol) draws the rectangle defined by the points x1,y1 and x2,y2. The rectangle consists just in the outline if ol is true, and it's filled in if ol is false.

Lines 4–6: As we already did in the previous projects, these lines will show the current score in our HUD.

Lines 8–11: This is how the cherries' number is shown. We first draw the sprite by resizing it of 1/2 its dimension, and then we show the numbers next to the icon.

Lines 12–14: Similarly to lines 8–11, we show the number of lives next to a resized idle image of Berry.

You can now run the game to check that the HUD and the pause system both work well. The result will be the one in Figure 8-10.

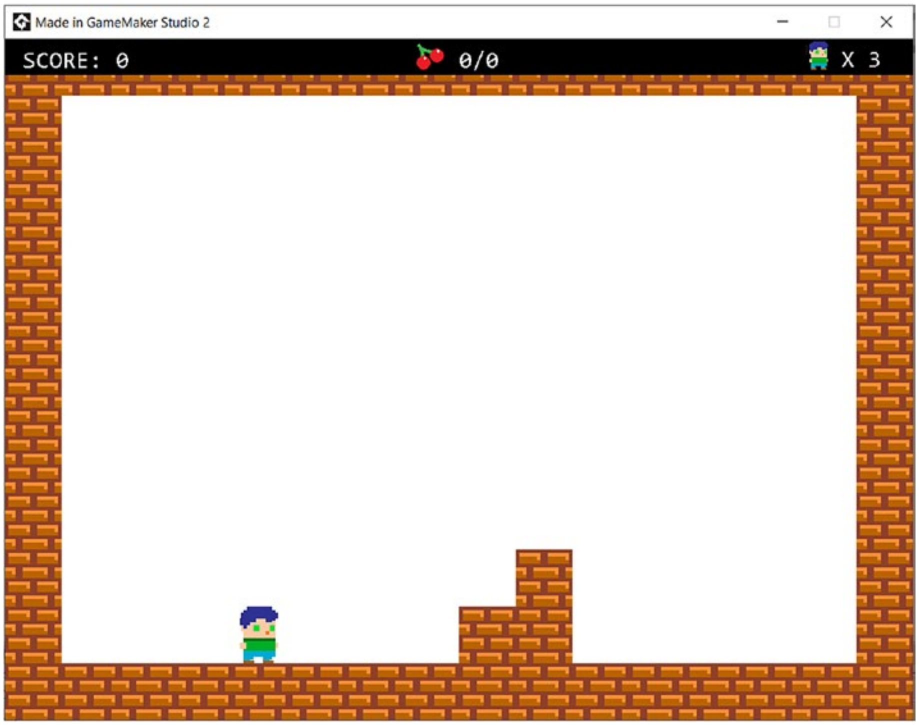


Figure 8-10.

Anyway, we can't check the game over functionality or the collecting feature, since we don't have any enemy or cherry in our game! Let's fix this!

How to die

This will be one of the easiest sections in the whole book. In fact, we structured the game so neatly that enemies and death are just a matter of setting variables and collisions.

First of all, let's talk a bit about the frightful ball-shaped enemies.

Cherry Caves' enemies are bouncing balls that follow some patterns conveniently with the design of the level. For the first level, we will create a basic ball-enemy bouncing on a vertical trajectory. For the second level, we will create a second ball-enemy following a more complex path simulating the bouncing down to various platforms. Since all the enemies have the same effects on the gameplay, we want to manage them using inheritance. We will create a ball-enemy parent with which we will manage all the interactions with the player, so that we can create as many ball-enemies as we like without being forced to rewrite the same mechanics and interactions every time.

Let's create a couple of new objects called obj_ball and obj_ball_purple. Give to the second the sprite spr_ball_purple and set its parent to be obj_ball.

This will be the first enemy type. We will code its bouncing behavior by making it move upward until it finds a wall; then it will invert the direction and start moving downward until it finds another wall forcing it to move upward again and so on.

Add a Create event for obj_ball_purple and initialize those two variables:

```
spd = 4;
dir = -1;
```

The enemy will move at a speed of 4 pixels per step, while the dir variable will track the direction.

Now we need the logic! Create a Step event and add this code:

```
1   if ( global.game_state == states.playing )
2   {
    1   y += spd * dir;
3
4       if ( place_meeting(x, y, obj_block) )
5       {
6           dir *= -1;
7       }
8   }
```

First, we check if the game state is in the playing state (line 1); then we increase the y value by spd * dir (line 3), which means that we are moving the instance by spd pixels in the direction defined by dir (left if dir is less than zero and right otherwise). Finally, we check for a collision with obj_block at line 5: if the collision occurs, we invert the direction of the instance (line 7).

Easy, right? You can test that everything's fine by putting some obj_block_red or obj_block_brown and a couple of obj_ball_purple in a room. You should verify that the instances of obj_ball_purple bounce up and down blocked only by instances of obj_block_red or obj_block_brown.

The only thing that's missing for our enemy is the ability to kill the player. We can manage this by creating an interaction between obj_player and obj_ball based on collisions.

Open up obj_player and add a new collision event by selecting Add Event ➤ Collision ➤ obj_ball.

In this event, we don't need to do much, since we will manage all the logics of death in the obj_controller Step event, so let's just add a status change and destroy the player object:

```
1   global.game_state = states.dead;
2   instance_destroy();
```

In the destroy event, we will just play a death jingle. Create a Destroy event for obj_player and add this one line in it:

```
audio_play_sound(snd_damage, 1, false);
```

When the player gets hit, as we said, we want to make them start again from the initial position (that we saved using global.startx and global.starty) until they run out of lives. Open up obj_controller's Step event and add this code at the bottom to manage the dead state:

```
1   if ( global.game_state == states.dead )
2   {
3       lives--;
4       alarm[0] = room_speed * 1;
5       global.game_state = states.playing;
6
7       if lives <= 0
8       {
9           global.game_state = states.gameover;
10      }
11  }
```

In the preceding code, we check whether the active state is dead. If it is, we decrease by one the global variable lives, change back the state to playing, and set the alarm 0 to 1 second. The alarm 0 event will deal the task to relocate the player on the original coordinates. We also check if the player has still lives (line 7). If they haven't, the active state becomes gameover.

We have to create an Alarm event to manage the respawn of the player. Click Add Event ➤ Alarm ➤ Alarm 0 and add this line of code:

```
instance_create_layer(global.startx,global.starty,
"Instances", obj_player);
```

Great! Now Berry can die! Well, I don't know if this is really a plus for him, but it definitely is for us!

Let's try out the game to check that everything is working great.

The only thing we are missing right now is the possibility to collect cherries. Let's work on it!

Cherry-picking

Collecting cherries, as per GDD, is the main goal in Cherry Caves. In this section, we will create the cherry object that we will use to score and unlock the next level.

Create a new obj_cherry object and give it the spr_cherry sprite. The player can pick up this item by colliding with it, and when they do, the score is updated as well as the number of cherries picked (monitored by global.cherries). So we will just need three events:

- **Create**: In which we will set the pts variable which is the amount of points gained by picking the item
- **Collision with obj_player**: That will destroy the instance, add pts to score, and increase global.cherries
- **Destroy**: That will play a simple particle animation and play a sound

Let's start by adding the Create event for obj_cherry. Inside this event, we just need to add this line:

```
pts = 100;
```

Now add a collision event with obj_player by clicking Add Event ➤ Collision ➤ obj_player and write this code in it:

```
score += pts;
global.cherries++;
instance_destroy();
```

The preceding code adds the amount of points we just set up in the Create event to the score variable, increases the number of picked cherries, and destroys the instance.

Last but not least, the Destroy event will just play a jingle and a particle animation:

```
effect_create_above(ef_firework, x, y, 1, c_red);
audio_play_sound(snd_cherry, 1, false);
```

Great! Now let's try this out by adding the cherries to the room. Because of how we coded obj_controller, it will count the number of cherries in the room and automatically set the maximum number of cherries that should be picked to pass the level. Picking them up, the number of picked cherries in the HUD will update as well as the score.

Anyway, we can't actually leave the room both because we don't have a goal object and because we don't have a second level. Well, we don't either have the first one. Let's fix all those issues one by one.

Through cherries, to the star

To win the game, as we said, the player should collect all the cherries in all the levels, and then they can access the exit of the level.

The exit is represented by a yellow star that will show up only when the player collects all the cherries in the room.

The logic behind the goal object is pretty simple: it's invisible and inactive until global.cherries is equal to global.cherries_max; then it turns active and visible; and when colliding with the player, it will warp them to the next level.

Let's create a new object and call it obj_goal and add a Create event. This event will be responsible to initialize the active and goal_reached variables which are used to manage the status of the instance depending on if the player collected all the cherries. Add this line in obj_goal's Create event:

```
active = false;
goal_reached = false;
```

Create a Step event and add this code:

```
1  if ( global.cherries == global.cherries_max and not goal_reached )
2  {
3      active = true;
4  }
5
6  visible = active;
```

To constantly check the status of the game, we need to check in the step event whether global.cherries has reached global.cherries_max (line 1). If the player collected all the cherries and we haven't yet triggered the goal, we want the instance to be activated (line 3). From the status of the active variable depends the visibility of the object (line 6).

Now, the most important piece of code of all is the collision with obj_player. When the player hits the goal, we want to deactivate the goal instance, play a victory sound and a little particle effect, and after a second, warp the player to the next room.

Add a new collision event with obj_player and put this code in it:

```
if ( active )
{
    alarm[0] = room_speed * 1;
    active = false;
    goal_reached = true;
    effect_create_above(ef_firework, x, y, 1, c_yellow);
    audio_play_sound(snd_goal,1,false);
}
```

The Alarm event will check if there is a room after the current one. If there is, the player gets warped to it; if there's not, the game will end showing the game over screen.

```
if ( room_next(room) != -1 )
{
        room_goto_next();
}
else
{
        global.game_state = states.gameover;
}
```

Great! Now everything is in place! We just need to add a new room to test it!

Create a new room of the same dimensions (1440 × 900) of room0 and call it room1. Add in this new room just an obj_block_red border and ground, the player, and obj_controller.Now run the game; it should start from room0 (if not, check the order of the rooms in the Resources sidebar: room0 should be on top of room1). Collect all the cherries in the room and run for the goal. Once you hit the goal, you should be teleported to room1.

Well, those rooms are a bit shallow, anyway. Let's design some better ones.

Level design: The art of creating worlds

Level design is more a matter of taste, experience, and convenience; but there are some good advices you can use. Especially with genres like single-screen 2D platformers, there are some easy checks you can do when you're designing a level. Let's talk about this!

Check the jumps

A common mistake is to not check jumps for good. Sometimes, if you're not very expert in level design, you can fall victim of the enthusiasm and design a very good level but with way too hard jumps. Maybe you know that particular jump can be done, but it's very hard to do, and you should be super precise. If it's not intentional for the jump to be so hard or if it's in the early phases of the game, avoid it.

Hard is good, too hard is not

The same should be applied to the toughness and density of enemies. A frantic-paced level is good, even a hard one, but pay attention to not make everything too difficult or you will get the opposite effect: instead of challenging the player, you will bore them, and they won't play at all.

Make it nice

One of the most important things of a level is good looking. Levels should be nice to look at. They should be harmonious, have a purpose, and/or tell a story. The industry is full of games that tell stories only by using good level design. Make your levels interesting and worth playing.

Don't make it too easy!

A thing that should always be avoided is to make things too easy for the player. No one wants to waste time playing a game too easy. Gamers want to be challenged and want to feel like they're improving. Try to make levels that reward the player, but making them earn that reward. An easy way to accomplish this is by making an enemy protect a valuable item or by hiding items in fake walls.

Designing caves

Back to our game, we should make something easy for room0 that can teach the basic mechanics without saying a thing. How can we do this?

Figure 8-11 shows an example of room0. This is a pretty good first level because it has some training area like (1) that has the purpose of letting the player take confidence with the controls and (2) that is useful to learn the ladder mechanic without any danger.

(3) is a good place for a first tentative scoring of points by collecting cherries. It's an isolated cherry guarded by a single easy-to-dodge ball. Also, there are two ways to reach that place: the first is by jumping the steps in front of the red ball and the second is by climbing the ladder and jumping from the platform. The conformation of the level in (4) suggests both the possibilities.

After picking the first cherry, the player can go easy to (5) and then reach the goal at (6).

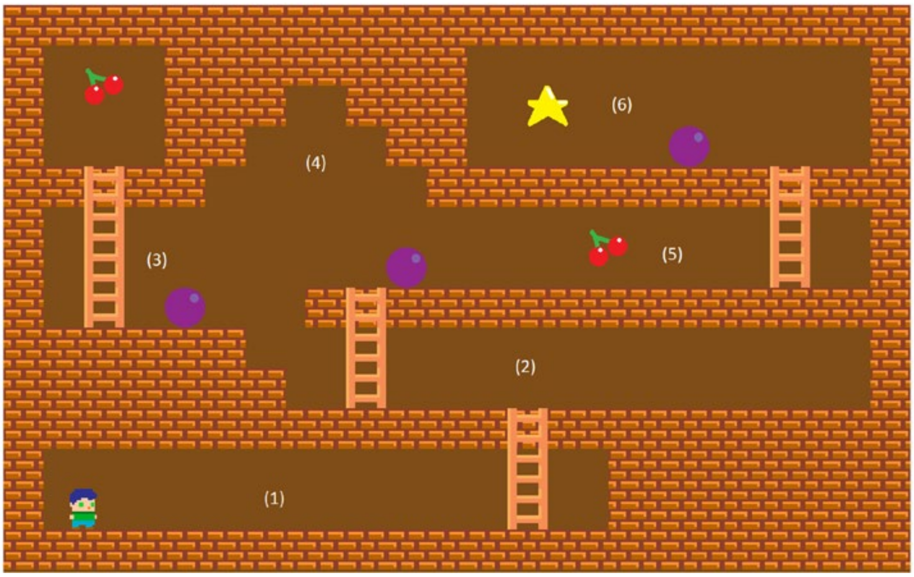


Figure 8-11.

A level designed trying to respect all those rules will help the player understand the basics of the game. A good understanding of the fundamental mechanics is crucial to introduce more interesting challenges in the next levels.

Level 2!

A second level should reinforce the concepts just learned in the first.

For example, you can build something like Figure 8-12. You can see that the starting point (1) is protected, so that we are still giving a safe start to the player; but just a second later, we introduce a pit (2) to start making the player uncomfortable and ready to face dangers. There is an easy-to-score cherry at (3) that has the purpose to motivate the player to pick the other two. A little gift is always a good encouragement.

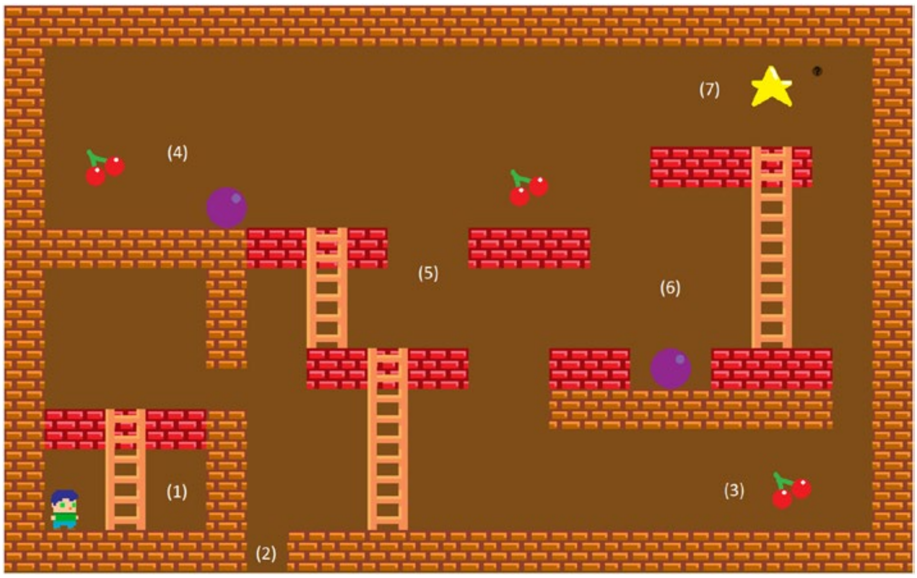


Figure 8-12.

At (4) there is a danger similar to the one the player experienced in room0, while at (5) they need to decide which way is better to use: the upper or the lower. Both look like feasible but dangerous. For the first time, the player can choose how to act. At (6), there is a not so easy passage, and at (7) there is finally the goal.

This level reinforces all the concepts learned in the previous level and introduces some new, like the possibility to choose and pits in which the player can fall off.

Actually, there is a way to make this level even more interesting! We could introduce a ball-enemy that rolls from (7) down to the pit at (2) passing from (5).

That sounds pretty interesting! Let's do it!

Select Path Layer in the Room Editor for room1 and design a path that rolls down the level into the pit, as in Figure 8-13, and call it path_greenball.

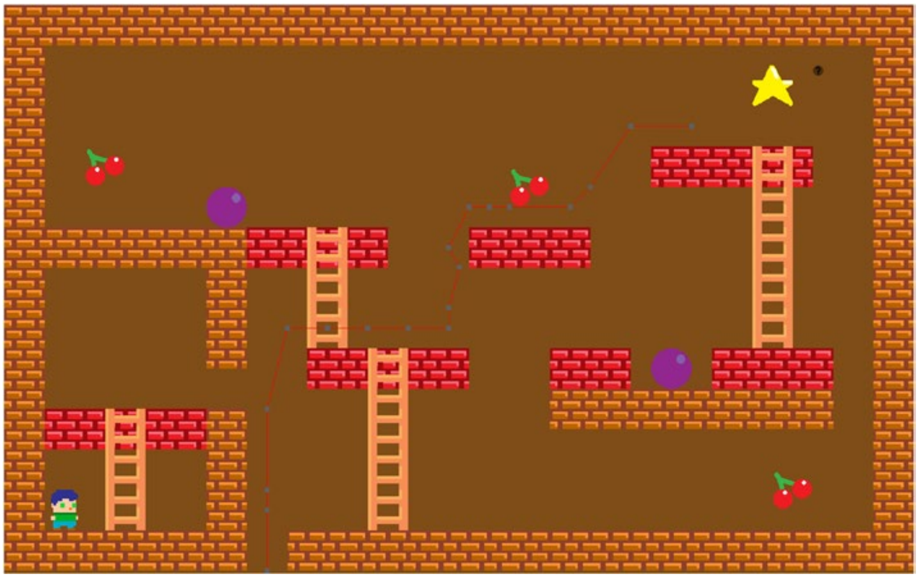


Figure 8-13.

Now create a new object and call it obj_green_ball and set its parent as obj_ball.

Add a Create event for this object and put this line in it:

```
path_start(path_greenball, 3, path_action_stop, true);
```

This line will make the ball start the path you just design as soon as it is created.

We want the object to be destroyed when it goes out of bounds. Since the path ends out of the screen, we can make a check on the Y-coordinates of the object in the Step event.

Create a Step event and add this code in it so that when the ball reaches a y value greater than the room height, it gets destroyed:

```
if ( y >= room_height )
{
        instance_destroy();
}
```

Now we want that every 2 seconds, a new ball is created and follows the same path. To do that, we can use an alarm in obj_controller.

```
Create a new Alarm 1 event for obj_controller and put this code
in it:
```

```
instance_create_layer(x,y, "Instances", obj_ball_green);
alarm[1] = room_speed * 2;
```

Now that we have this event, we want it to be triggered when we enter the level, so let's add this code at the bottom of obj_controller's Create event:

```
if ( room == room1 )
{
       alarm[1] = room_speed * 0.1;
}
```

Great! Now you can run the game and test that everything is working great! The balls are generated once every 2 seconds, and they follow the path down the pit (Figure 8-14).

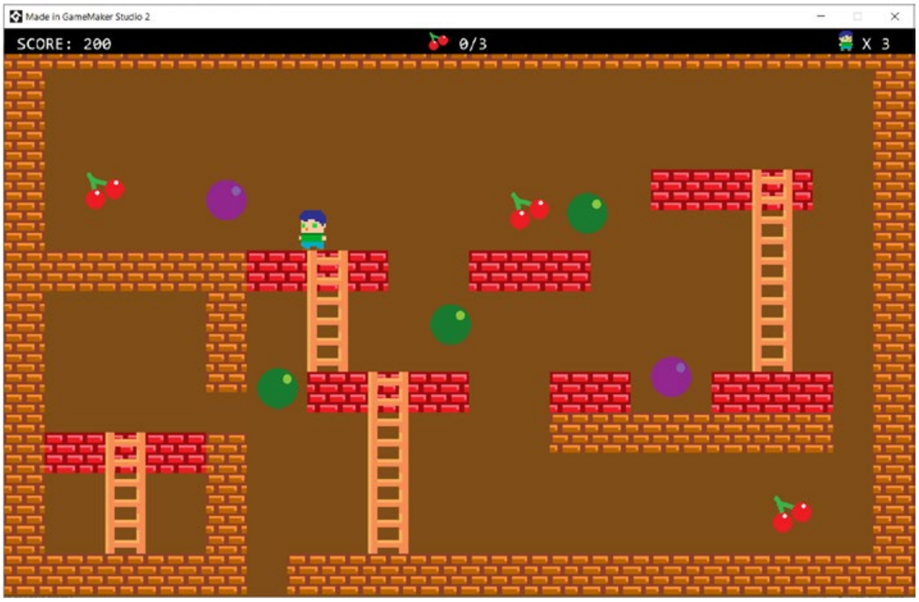


Figure 8-14.

We concluded this exciting chapter by creating a nice and fun single-screen 2D platformer. It's very old-school, indeed, but it has a lot of potential!

In fact, I challenge you to play out with it by designing new levels and enemies and maybe adding new items to collect!

Feel free to explore, create, and play, because this is the spirit to become a great GameMaker!

In the next chapter, we will go further and expand this project to make it become a scrolling platformer in the style of the end of 1980s/early 1990s classics like Super Mario Bros. and Sonic the Hedgehog.

TEST YOUR KNOWLEDGE!

1. What are the most important characteristics of a platformer?
2. How can you check if the player pressed a key using only GML?
3. How can you flip a sprite?
4. How does the place_meeting function work?
5. How does the sign function work?
6. Can you describe how we implemented gravity?
7. Can you modify the code so that you can perform a double jump?
8. Can you describe how we implemented the ability to climb ladders?
9. Describe Cherry Caves' game flow. Why do we need the dead state?
10. How can you activate or deactivate all the instances of the game at once?
11. Try to play around with the HUD trying different layouts. Which one is the best? Why?
12. What are the good level design principles?
13. Can you design a third level following the good level design principles we defined?

CHAPTER 9

Scrolling Platformer

In Chapter 8, we saw how to create a single-screen platformer (SSP) implementing some interesting features and tackling some challenging new problems like gravity management, jumping, climbing on ladders, and picking up collectibles to clear the level. We also talked about level design and how to make interesting and fun the same dodging-and-collecting activity by positioning things in smart ways.

In this chapter, we are going to extend that project by adding some new features to turn Cherry Caves 1 (CC1) into a scrolling platformer in the style of Super Mario Bros. The new project will be called Cherry Caves 2 (CC2) and will be heavily based on CC1 both from a technical and narrative point of view.

We will use this opportunity to compare single-screen platformers to side-scrolling platformers and learn how to implement all those new and juicy features that made the fortune of the genre from SMB on.

At the end of the 1980s, scrolling platformers started to conquer the market becoming one of the most popular types of game. Most of the success of this genre is thanks to Super Mario Bros. (Nintendo, 1985). The game by Nintendo revolutionized the entire gaming industry giving the players a game so good that's still subject of study for game designers. What makes this game so good? Why is the platforming genre so successful? Let's dive into the exciting world of side-scrolling platformers to answer to these questions and to understand how to create good and fun games!

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_9

As usual, let's start from the game design document.

Cherry Caves is a single-player 2D side-scrolling platformer game inspired by classics like Super Mario Bros.

The player will guide Berry through his second adventure. This time the goal is not collecting all the cherries, but just reaching the end of each level.

Story and setting

After his descent to and return from the Cherry Caves, Berry brought cherries back to humanity, and the mankind returned to grow them. Berry was celebrated like a hero.

However, Berry underestimated the dangers that lay in the Cherry Caves; in fact, the strange green and purple balls were the eggs of an octopus-like alien race attracted by the unique taste of cherries. Now that the human being took all the cherries from the caves, the eggs hatched, and the squishy aliens are invading the surface of the Earth!

There's only one person who can exploit the power of cherries to defeat the aliens: Berry!

Gameplay

The goal in Cherry Caves 2 is to reach the end of each level, like what happens in classic 2D side-scrolling platformers. So, differently from the first game, this one is very focused on platforming itself and collecting items is just a plus.

The ball-enemies are gone. In this second game, the enemies are octopus aliens who have similar behavior, but are implemented in a slightly different way.

Victory condition

Each level is completed by reaching its end represented, like in the first game, by a yellow **star** (Figure 9-1).

Every time Berry (the player's character) gets hit, he dies and loses a life; then the level is restarted.

The player can die three times before the game is over.

Figure 9-1. *Star (goal)*

Items

CC2 features **cherries** (Figure 9-2) not as mere collectibles, but as a power-up. In fact, picking cherries will boost Berry making him invulnerable and lethal to all the enemies for a small amount of time.

Figure 9-2. *Cherry (power-up)*

Coins (Figure 9-3) are a new addition to the game. They are just collectible items scattered in the levels. When the player pick a coin up, their score grows.

Figure 9-3. *Coin (collectible)*

Note Like in Super Mario Bros., the player can be hit just once before they die. This is so to try and bring some balance and challenge to the game.

In this project, collectibles are just things that you collect for the sake of it, and their only effect is to grow your score. Anyway, a good thing to do would be to give a meaning to collectibles and to allow the player to unlock or buy something in exchange, as a reward for the effort. We will talk more about this in Chapter 10.

Controls

Controls are the same to the ones in Cherry Caves.

Keyboard

Left: Move left.

Right: Move right.

Up: Move up (ladder).

Down: Move down (ladder).

Spacebar: Jump.

Esc: Open/close menu.

Enter: Confirm (menu).

Enemies

In CC2, you have a new kind of enemies, more detailed from an aesthetic point of view and with different patterns compared to those in CC1.

The new enemies are alien octopuses. There are two kinds of enemies:

- **Green Octopus** (Figure 9-4): These strange green octopus aliens hatched from green balls/eggs, and they just move back and forth. Don't touch them!

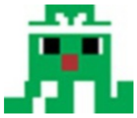


Figure 9-4.

- **Purple Octopus** (Figure 9-5): These purple octopus aliens can jump very high and can cling to the ceiling. Beware of their fall! They hatched from purple balls/eggs.

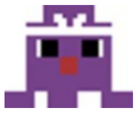


Figure 9-5.

Attack

This time Berry is not so vulnerable. In fact, not only he can destroy octopuses by eating a cherry and becoming invincible but he can also jump on enemies' heads and squash them.

Miscellaneous

Other cool additions in CC2 are the platforms. You can now use special platforms to reach new places. There are three different kinds of platforms:

- **Falling Platforms** (Figure 9-6): Very unstable platforms. They tend to fall down when the player touches them.

Figure 9-6.

- **Moving Platforms** (Figure 9-7): They move back and forth and can be used by the player to access certain hidden places. You can see a lot of them in games like Super Mario Bros. and the original Mega Man series.

Figure 9-7.

- **Trampoline Platforms** (Figure 9-8): They're like real-world trampolines and make the player's character jump constantly. When the player presses the jump button at the right time, the platform makes the player jump even higher to reach very high places.

Figure 9-8.

Similar games

CC2 is a very standard 2D side-scrolling platformer and plays like Super Mario Bros. (Nintendo, 1987).

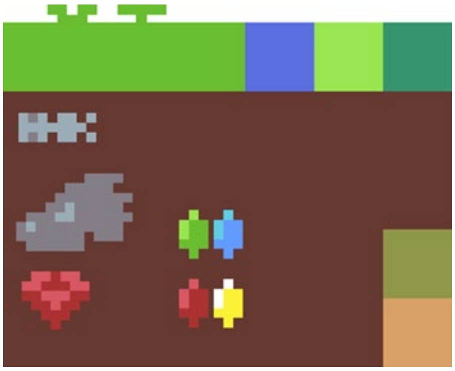
Other similar games are Rayman (Ubisoft, 1995), Alex Kidd in Miracle World (SEGA, 1986), Donkey Kong Country (Nintendo, 1994), Super Mario Bros. 3 (Nintendo, 1993), DuckTales (Capcom, 1989), and Klonoa: Door to Phantomile (Namco, 1997).

Assets

As usual, let's list the assets needed for this chapter. They are basically the same to Cherry Caves, with some new addition. Since the new game will be made starting from the base of Cherry Caves, I will only list the new assets.

spr_land

This is our first tile set. We will learn how to work with tile sets to build our levels faster and make them look more beautiful. Tile sets are also a good way to not fill rooms with objects to design a level, like we did in Cherry Caves, making them a good way to create levels saving computing resources.



Size: 416 × 352
Pivot Point: Top-left (not important)
Collision Mask: Automatic, Rectangle (not important)

spr_skybg

This image is just a blue image that we will use for the sky in our levels.

Size: 400 × 300 (not important, since it will be repeated as a pattern)
Pivot Point: Middle-center (not important)
Collision Mask: Automatic, Rectangle (not important)

spr_platform_falling

This is the sprite we will use for the falling platforms.

Size: 64 × 32

Pivot Point: Top-center

Collision Mask: Automatic, Rectangle

spr_platform_trampoline

This is the sprite we will use for the trampoline platforms.

Size: 64 × 32

Pivot Point: Top-center

Collision Mask: Automatic, Rectangle

spr_platform_moving

This is the sprite we will use for the moving platforms.

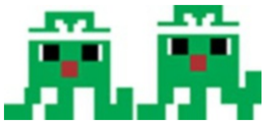
Size: 64 × 32

Pivot Point: Top-center

Collision Mask: Automatic, Rectangle

spr_octopus_green

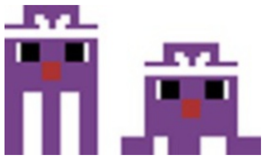
These are the sprites we are going to use for green octopus aliens. Together they make a nice animation that simulates their squishy crawl.



Size: 64 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle
Animation Speed: 4

spr_octopus_purple

These are the sprites we are going to use for the purple octopus aliens, the ones that jump and cling to the ceiling. These two sprites form the jumping animation, which is all we need for them.



Size: 64 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle
Animation Speed: 4

spr_titlescreen

This is the image that we will use in the title screen.

cherry
caves
2

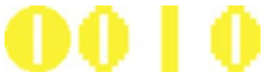
Size: 1280 × 720

Pivot Point: Top-left

Collision Mask: Automatic, Rectangle (not important)

spr_coin

This is the sprite of the coins, the new collectibles of CC2. You should import these four images in the order shown here to create the illusion of a rotating coin



Size: 32 × 32

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

Animation Speed: 8

spr_terrain

This sprite will be used to create new blocks that we will use just to mark the impassable places.

Size: 64 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle

Fonts

We need just one more font. This font will be used to draw "PRESS START" in the title screen.

fnt_title

For this font, I am using the preinstalled Impact font with style regular and size 20.

Sounds

Other than the sound effects we already had in Cherry Caves 1, we will also need the following new assets:

snd_coin: This sound effect will be played when the player picks a coin up.

snd_frenzy: This sound effect will be looped when the player picks a cherry and becomes invincible for a small amount of time.

snd_squash: This sound effect will be played when the player kills an enemy.

The more you do it …

… the better you get! And the more you create games, the more you align to the standards of modern gaming. Cherry Caves was a fun game with a lot of potential; but the ideas that we want to implement in this chapter will put our game closer to standards of modern blockbusters like New Super Mario Bros. 2, Shovel Knight, Celeste, Hollow Knight (HK), and Rayman Legends.

This major update to Cherry Caves will be easier to make, very interesting to discuss, and lots of fun to play! Why easier? Well, because we are starting with a working 2D platformer engine and we just need to implement some new content and quality of life (QoL) features – the hard and boring part is already done! You will be amazed by how easier it is to create a game when you already have a good base. This is why you should always work in a modular fashion! No one likes to create the wheel every time they want to drive a car.

So let's begin!

First things first, duplicate your Cherry Cave project. We will use a copy of that project to build upon it, so that we don't have to make the same things twice.

To duplicate a project in GMS2, just open it and select File ➤ Save As in the menu at the top of the window, to save a copy of the project in another location. Select the location you want, and call the new copy Cherry Caves 2.

Ok, now let's start working on the new copy of the project! The first thing we want to do is to get rid of all the rooms we previously created. We will create new rooms in this chapter, and we don't really need those we created in Chapter 8, since they are made with a style that is more right for a SSP than a scrolling platformer.

A golden rule of game design is to always remember the genre you are working on, when you design new content. A good content for a genre can be extremely wrong for another genre.

Title screen

Title screens are the first thing you see in a game. They are very important, more than you think! In fact, they are the first thing to set the tone of the game. You may have noted that title screens always match the atmosphere of the game you're going to play, so that you can enter in the right mood to enjoy the game at its best. This is a big part of the immersion process, one of the factors that put the player in a cognitive flow status.

The cognitive flow is a status of energized focus that happens when someone is fully immersed into an activity. This status can be influenced by a lot of factors like the atmosphere of the game (boosted by aesthetics and narrative), the gameplay, the difficulty level, and the skills of the player.

A good title screen can totally help the player to enter in the right mood for the game. An appropriate title screen can give the game credibility and help the player to immerse in the experience, facilitating the triggering of the cognitive flow status.

This is a good enough reason to create a title screen for our game!

In particular, the title screen I created (Figure 9-9) tries to communicate the main topic of the game: the invasion of those strange octopuses aiming for the cherries. Let's analyze it!

First of all, we see the sky. It's strange because the title is Cherry Caves 2, but we're not in the caves anymore. Since the game is still called like this, it means that the caves have a role in all this (and they have)!

The point of view of the image is from the bottom: this gives us the impression of being in a disadvantageous position, like if we are dominated.

The octopuses dominate the scene by overlooking us. This gives the impression that they are dominating us.

Another thing to note is that one of them (the green one) watches angered in our direction, meaning that they're not friendly, while the other (the purple one) is trying to pick the cherry from the game title, which tells us that they want the cherries.

Putting it all together, we can tell that those strange octopuses are angered with us because we stole their cherries (in Cherry Caves 1) and now they are here to take them back and punish us.

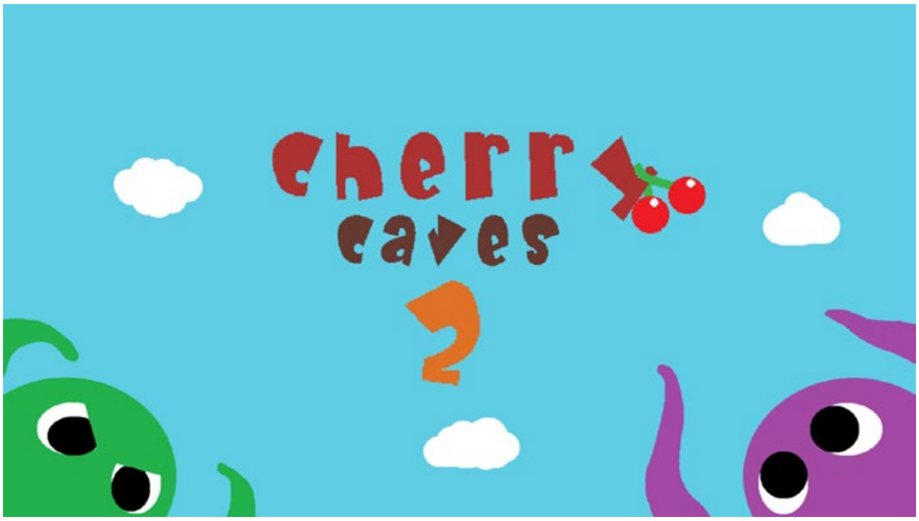


Figure 9-9. *The title screen of Cherry Caves 2 introduces us to the story of the game*

Let's start by creating a new room named *title* by right-clicking Rooms in the Resources sidebar and selecting Create Room.

Double-click the newly created room, and it will open the Room Editor. Head to the Layers panel and select the Background layer as in Figure 9-10.

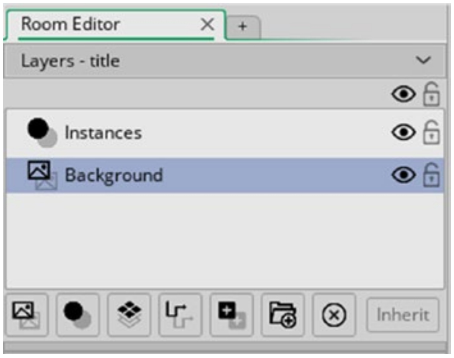
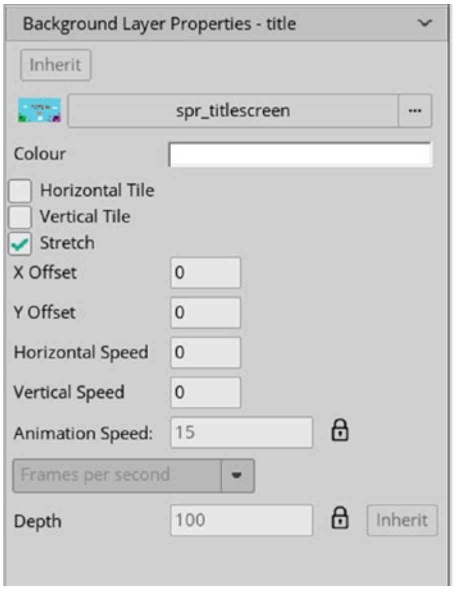


Figure 9-10. Select the Background layer in the Room Editor

Once you selected the Background layer, head to the Background Layer Properties panel (just under the Layers list) and select spr_titlescreen as sprite by clicking the three-dot button. Don't forget to also tick the Stretch box, so that the background image will be displayed also on bigger screens. The result should look like Figure 9-11.


Background Layer Properties - title

Inherit

spr_titlescreen ...

Colour

Horizontal Tile

Vertical Tile

✓ Stretch

X Offset 0

Y Offset 0

Horizontal Speed 0

Vertical Speed 0

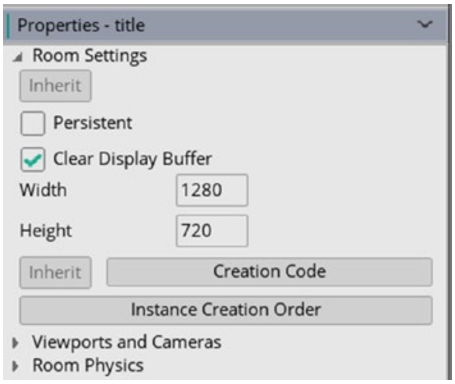
Animation Speed: 15

Frames per second

Depth 100 Inherit

Figure 9-11. Select spr_titlescreen as Background layer's sprite from the Background Layer Properties panel

Finally, go to Properties panel, just under the Background Layer Properties panel, and set the Width of the room to 1280 and its Height to 720 like in Figure 9-12.



Persistent: unchecked
Clear Display Buffer: checked
Width: 1280
Height: 720

Figure 9-12. *Set the room's width and height*

Ok, now we have our title screen, but it doesn't really do anything, and of course it doesn't allow us to play the game.

We can fix this by creating a new controller object – just like obj_controller – specifically for the task of managing this title screen.

The title controller will allow us to create a blinking *PRESS START* label, and it will manage the keys input to start or quit the game, and we will also implement the possibility to go fullscreen pressing F12 or the F key.

Create a new object by right-clicking Objects in the Resources sidebar and selecting *Create Object*. Call the new object obj_controller_title and open up its Object Editor.

Just like obj_controller, this object won't have a sprite.

Let's start from the easy things: keys management.

Create a new Step event for obj_controller_title and write the following code in it.

```
1   if ( keyboard_check_pressed(vk_enter) or keyboard_check_
    pressed(vk_space))
2   {
3       room_goto_next();
4       audio_play_sound(snd_menu, 1, false);
5   }
6
7   if (keyboard_check_pressed(ord("F")) or keyboard_check_
    pressed(vk_f12))
8   {
9       window_set_fullscreen(!window_get_fullscreen());
10  }
11
12  if (keyboard_check_pressed(vk_escape))
13  {
14      game_end();
15  }
```

Lines 1–4: Pressing the enter or the space key will allow the player to go to the next room (that will be the first level of the game). We are using keyboard_check_pressed, but really any function of the keyboard_check family would do, in this case.

Lines 6–9: Here's the new bit! F and F12 are the fullscreen toggle keys. You can activate or deactivate fullscreen (depending on the current status) by pressing one of those keys. To do it, we are using window_set_fullscreen(bool), which is a function that takes a Boolean argument (bool in this case) and activates (if bool is true) or deactivates (if bool is false) the fullscreen mode. To do the toggle trick, we are passing as an argument to this function the negative of the current status of the screen. In fact, windows_get_fullscreen returns true if the game is in fullscreen mode and false if it's not. By negating it with the unary operator NOT (!), we are just saying that we want to set the fullscreen mode if it's not set and we want to deactivate it if it's set.

This is a trick you can reuse anytime you need to implement a toggle: just negate the current status.

Lines 11–14: Finally, by pressing the escape key, the game just ends using the function game_end().

The next thing we want to add is the blinking text. It's a nice addition that makes the screen feel less boring and static and reminds of some classic platformers that did this, like Mega Man (Capcom, 1987) and Power Blade (Taito, 1990).

To create a blinking text, we need a timer that will toggle a Boolean variable. This Boolean variable will be used to decide if the text should be shown or not.

The diagram in Figure 9-13 shows the logics of the algorithm we are going to write.

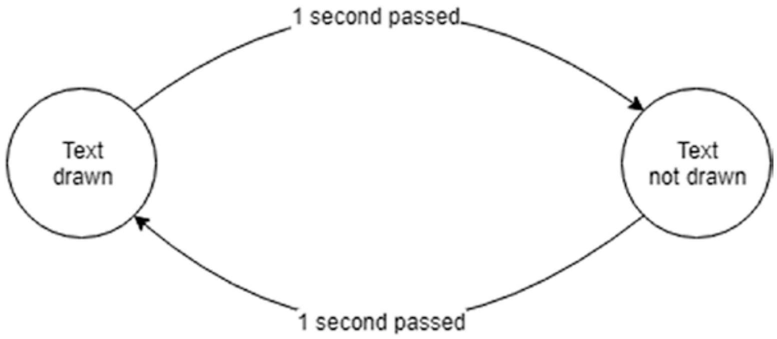


Figure 9-13.

So we need a controller variable, a speed variable (to decide the blinking speed), and an alarm.

Let's start by adding a Create event to obj_controller_title by clicking Add Event in the Events panel of the Object Editor. In this new event, add this code:

```
1   // --- Blinking text
2   blink_speed = 1;
3   press_start = true;
4   alarm[0] = room_speed * blink_speed;
5
6   // --- Display settings
7   var width = 1280;
8   var height = 720;
9   display_set_gui_size(width, height);
10  window_set_fullscreen(true);
```

Line 1: This is our blinking speed variable. The text will blink once per second (you can adjust this following your taste).

Line 2: This is our controller variable. We set it to true because we want to start the game with the text visible.

Line 3: Here we set up the alarm to be triggered after one second.

Lines 7–10: In these lines, we force the resolution of the game to 720 p (line 9), and then we activate the fullscreen mode.

Tip Using display_set_gui_size(width, height) with hardcoded width and height values is very useful when you want to quickly make your game compatible with many screen resolutions. Anyway, if you want some better compatibility avoiding black stripes at the borders of your screen, you will need to add the possibility to set a custom resolution by allowing the player to select from a list the right values of width and height to pass to display_set_gui_size.

The code for the alarm is pretty straightforward. We just negate the control variable and reset the alarm so that it will trigger again in a second.

Create a new Alarm event by clicking Add Event ➤ Alarm ➤ Alarm 0 and write the following code in the event:

```
1   press_start = !press_start;
2   alarm[0] = room_speed * blink_speed;
```

In the preceding code, as we said, we negate the control variable (line 1) by assigning its own negated value to it (just like we did previously with the fullscreen key), and then we reset alarm 0 to trigger in 1 second (line 2).

We're nearly done! We only have to actually draw (or not) the text. As usual, we will do it in the Draw GUI event. Create a Draw GUI event by clicking Add Event ➤ Draw ➤ Draw GUI and write the following code in the event:

```
3   var cam_w = display_get_gui_width();
4   var cam_h = display_get_gui_height();
5   x = (cam_w/2)-50;
6   y = cam_h-100;
7
8   if press_start
9   {
10      draw_set_color(c_red);
11      draw_set_font(fnt_title);
12      draw_text(x, y, "PRESS START");
13  }
```

Lines 1–4: We change the coordinates of the object calculating the position using the width and height of the display. To do that, we use display_get_gui_width and display_get_gui_height.

Lines 6–11: Here we check if the control variable is true (line 6), and if it is, we draw the text at coordinates x,y (the ones at which the object has been repositioned in lines 3 and 4) using the font fnt_title (line 9) of color red (line 8).

Ok, now it's all done. We just have to test it! Drag and Drop obj_controller_title in the room, in the position in which you want the text to be drawn, then save and run the game by pressing F5 or clicking the Run button in the toolbar. It should open up the title screen with the blinking text PRESS START (Figure 9-14). Of course by pressing the enter key, we will get an error, because we don't have a next room to join; but you can test out that the fullscreen toggle is correctly working and that by pressing the escape key you will be able to quit the game. Well done!

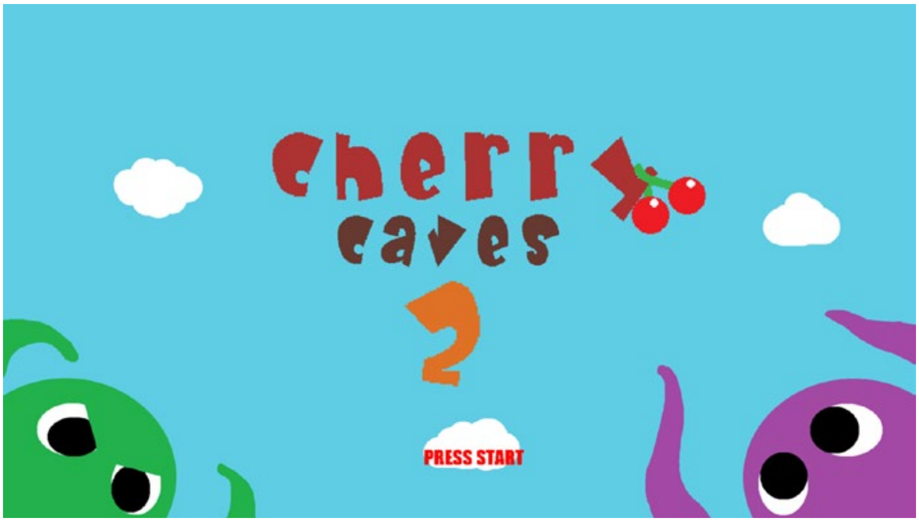


Figure 9-14. *Now the title screen features a nice blinking PRESS START text*

Tiles and level design

In the previous chapter, we created levels just placing blocks and ladders, like in the SSP classics. This was the way to do it back in the days, but with the technology advancements, more memory was available to devs and so more colors and more possibilities to shape new and varied worlds. With all those possibilities, a single image wasn't enough anymore, so tile sets were created.

Tile sets are images that collect various tiles of the same size that can be used to build game worlds. Tile sets revolutionized level design because they reduced the quantity of files to load to build a level but still allowing the designer to create diverse levels. Tile-based levels have the advantage to have the level organized as a grid of N × M tiles of the same size. This makes easier to manage distances and movements and of course have a sense of distance that can be expressed in scale in Minimaps or that can be exploited to recreate realistic locations.

The first tile-based game was Galaxian (Namco, 1979), a fixed shooter that implemented a tiled (using tiles of 8 × 8 pixels) scrolling background (just the background, not the level). This technology became the basis of later blockbusters like Donkey Kong (Nintendo, 1981) and revolutionized the industry. In fact, back in the days, to make games was one of the most difficult things for a developer. Game developers didn't have many tools that we have now like modern game engines and powerful computers; they had to save on every bit of memory and directly program their games for every single piece of hardware. Having a new technology that allowed to save on the execution time needed to load images in memory was a great thing that allowed games to be faster and more colorful and also allowed game designers to express themselves and their ideas better.

Tile sets are a great addition to a GameMaker's toolbox, and, of course, they are available in GameMaker Studio 2. The tile set we are going to use is the one in *Figure 9-15,* and it's made of *32 × 32-pixel* tiles, and the total size of the image is *416 × 352 pixels*.

To use our tile set, we should first of all create a sprite associated to the image representing the tile set, as we said in the *"Assets"* section.

The new sprite, as we already said, is called *spr_land* and will be the base to create the tile set. Change its tile width.

Right-click Tile Sets in the Resources sidebar and select Create Tile Set to open up the Tile Set Editor. Rename the new tile set *ts_land* and assign *spr_land* (Figure 9-15) to it.

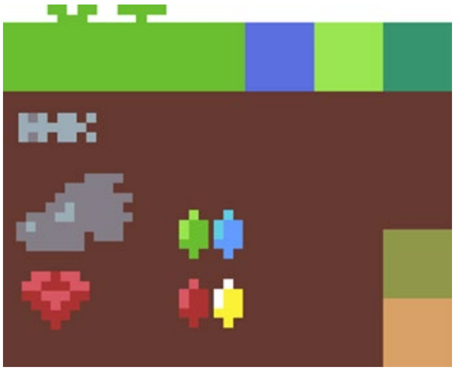
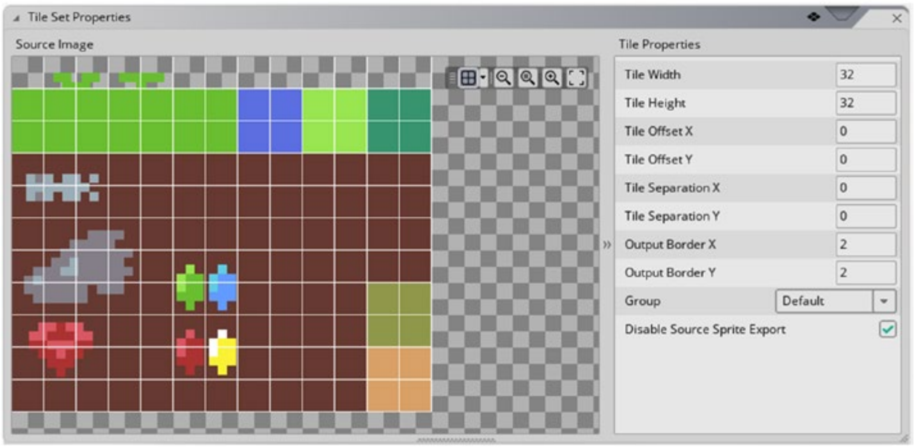


Figure 9-15. *The tile set that we will use in this project (tiles of size 32 × 32 pixels)*



Tile Width	32
Tile Height	32
Tile Offset X	0
Tile Offset Y	0
Tile Separation X	0
Tile Separation Y	0
Output Border X	2
Output Border Y	2
Group	Default
Disable Source Sprite Export	✓

Figure 9-16.

In the Tile Properties section of the Tile Set Properties window, change both the width and height of the tile to 32 pixels, as shown in Figure 9-16. We will work on the level design using tiles of 32 × 32 pixels.

We are now ready to build the first level of the game! Let's create a new room called room0. This new room should feature some more layers, compared to our first project (Chapter 8).

We will need three instance layers: one for enemies, items, and the player, one for the blocks that build the level like ladders and red and brown blocks, and one for the objects that we will use to delimit the impassable parts of the level (this layer will not be visible, once in game).

We will also need, of course, a tile layer that we will use to build the aesthetic of the room.

Let's start from this one!

Create a new tile layer by clicking Create New Tile Layer button in the Layers panel in the Room Editor as shown in Figure 9-17.

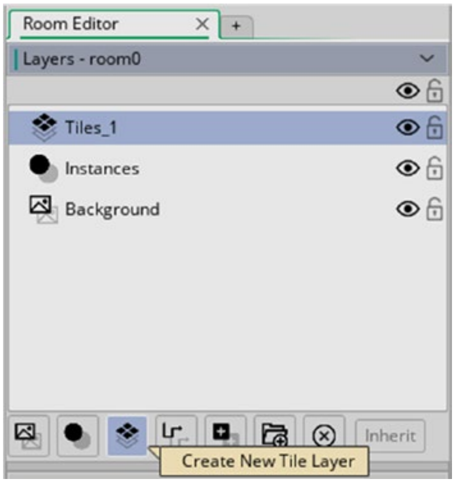
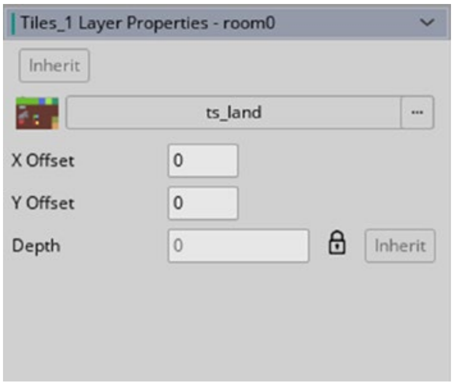


Figure 9-17.


Tiles_1 Layer Properties - room0

Inherit

ts_land

X Offset 0

Y Offset 0

Depth 0 Inherit

Figure 9-18.

In Tiles_1 Layer Properties, add a new tile set by clicking the three-dot button and choosing ts_land, as shown in Figure 9-18.

Now that the tile set is loaded and the tile layer is selected, a new panel labelled Room Editor (Figure 9-19) should open on the right side of GMS2.

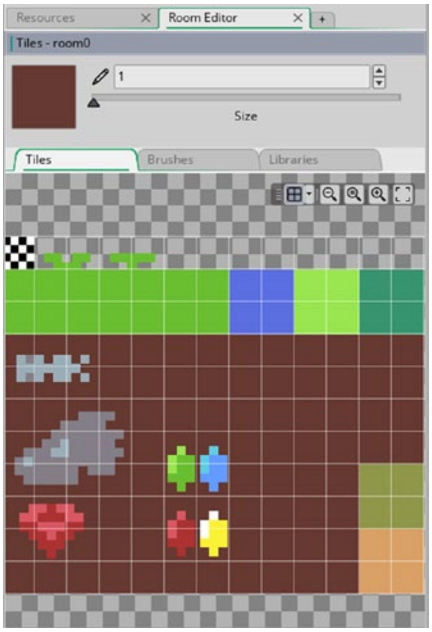


Figure 9-19.

This new panel features three tabs:

- **Tiles**: Where you can select one or more tiles to draw your level
- **Brushes**: Where you can compose brushes to design your level using some specific tile patterns created by selecting some areas from the tile set
- **Libraries**: Which allow you to add to the room premade animated tiles and auto-tiles. Animated tiles are exactly what they sound: tiles that are animated, for example, the water in games like Final Fantasy, The Legend of Zelda, or Pokèmon. Auto-tiles are a tile set that allow you to create tiles that automatically connect to each other when they are placed together. They are widely used to create sand pits, rivers, lakes, and so on.

Note Auto-tiles are widely used in top-down games, like The Legend of Zelda, Final Fantasy, Pokèmon, and basically all the top-down RPGs that you can think of. Anyway, we are not going to cover them in this book since they're not very popular in platformers. If, after reading this book, your objective is to create a top-down RPG, I strongly recommend you to check auto-tiling on GameMaker's official documentation (`https://docs2.yoyogames.com/source/_build/2_interface/1_editors/tilesets.html`). It can add a lot to your game's aesthetic.

To start drawing the level using tiles, the first thing to do is to adjust the grid of the room by opening the grid options in the grid toolbar and setting both Grid X and Grid Y to 32 pixels, as shown in Figure 9-20.

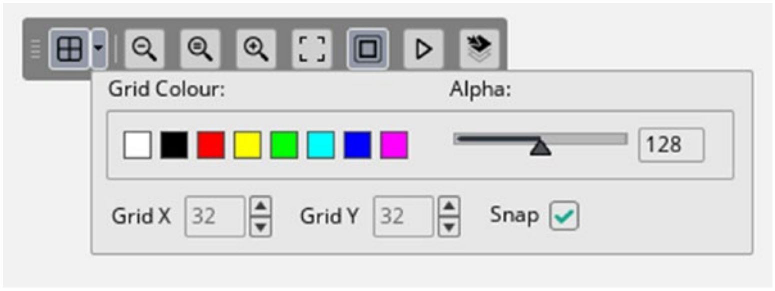

Grid Colour:
Alpha: 128
Grid X 32 Grid Y 32 Snap ✓

Figure 9-20. *The grid toolbar showing various options*

Select the first tab, Tiles, and select the green tile and draw a line of grass. Just under this green line, draw a thick brown area that goes down to the bottom of the room like in Figure 9-21.

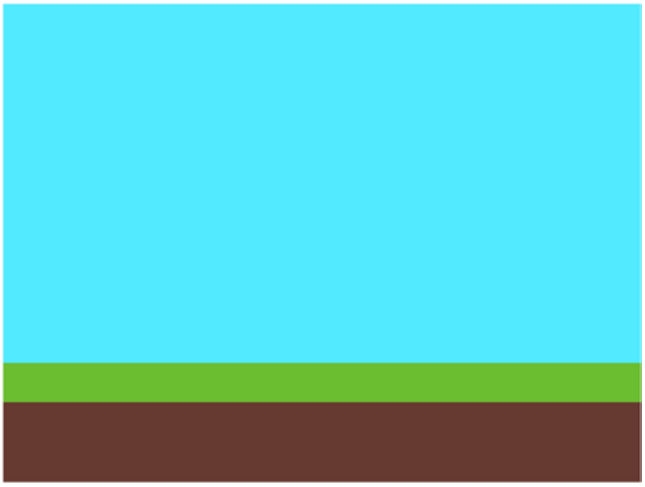


Figure 9-21.

Now that we have something like Figure 9-21, we should delimit the areas in which our hero can move. We will do it using a new object. Let's create it!

Select the Resources sidebar and right-click Objects and select Create Object and call it `obj_terrain`. Assign to this object the `spr_terrain` sprite and set obj_block as its parent by clicking Parent in the Object Editor and selecting obj_block.

Now we have an object that we can use to delimit walkable areas in our levels.

Open up again the Room Editor and create a new Instance Layer and name it Blockers.

With the Blockers layer selected, Drag and Drop an obj_terrain object and stretch it so that it covers completely the grassland (and underlying ground) you created with the tiles as shown in Figure 9-22.

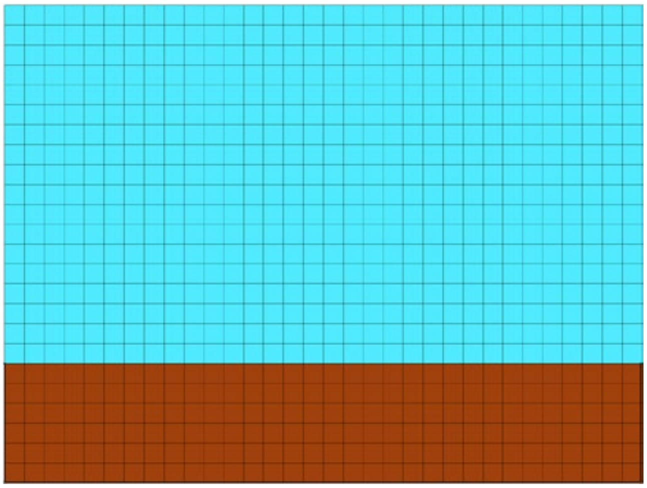


Figure 9-22.

We don't want obj_terrain to be shown on screen; we just want to use its collision mask. To do that, we can make the whole Blockers layer invisible by clicking the open eye icon in the Layers list in the Room Editor (Figure 9-23).

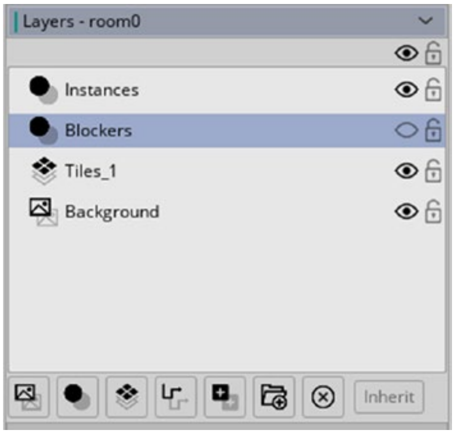


Figure 9-23. *Blockers layer is now invisible (the eye icon is now a closed eye)*

Now that the Blockers layer is invisible, the grassland we have drawn previously is visible once again. Drag and Drop obj_player into the room, just above the grassland, and run the game.

The title screen will appear. Pressing the enter key or the spacebar, the new room (room0) will be loaded, and the player will gently fall on the grass. Great! It works!

We just learned how to design aesthetically complex levels without using a ton of objects to manage collisions, which is a very efficient way to do it! It's a big save of memory and execution time. This will make our games lighter and faster, and it will also allow us to create levels faster.

We can use obj_terrain also to delimit the borders of the screen. In fact, you can place instances also outside the room grid. This is a fast way to delimit the borders of the screen without writing a single line of code at the price of just three instances (because generally you want to leave the player fall into pits and die and you don't need to block this fall).

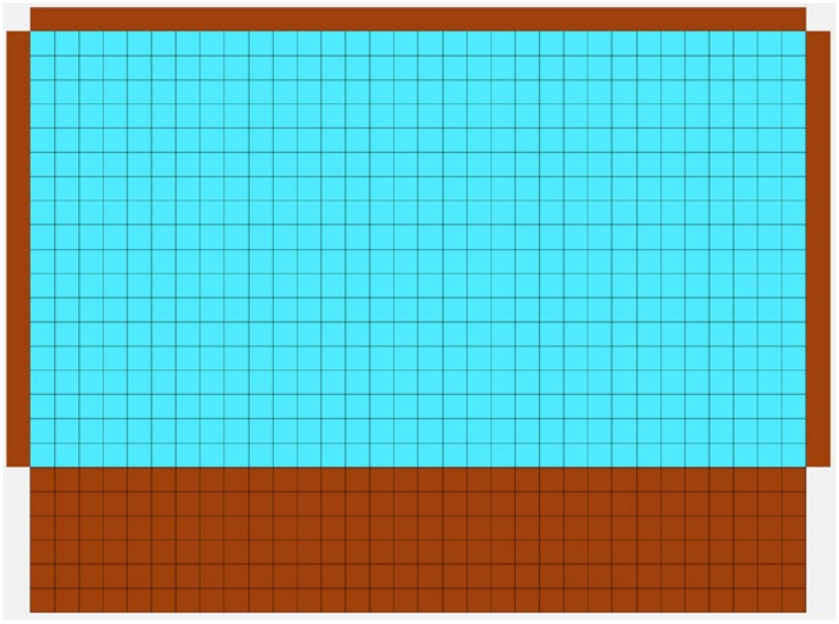


Figure 9-24.

Note You can add instances to a layer only if it is visible. So remember to always make the Blockers layer visible when you want to add new instances of obj_terrain to it.

Scrolling camera

A scrolling platformer is not such if it's not scrolling, and our game is not!

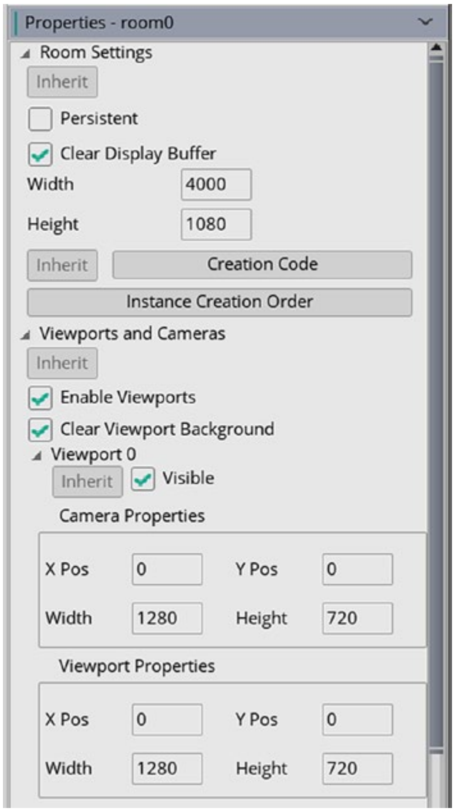
We saw how to create a scrolling level in Chapter 6, when we created the scrolling level for Space Gala. We are going to follow a similar process, but with some little differences.

As we learned during the development of Space Gala, to create a scrolling level, we need to activate viewports and create a camera. The camera will zoom in to a certain position, and this time it won't move by itself, but it will follow the player moving through the level.

Let's start by activating viewports. You can do this by opening up room0 and heading to the Properties section in the Room Editor left sidebar. Once there, tick the Enable Viewports and Clear Viewport Background boxes; then open up the Viewport 0 section by clicking the small black arrow beside the Viewport 0 label and tick the Visible box. Now, in the Viewport 0 section, head to Camera Properties and change the Width to 1280 and Height to 720. Do the same for Viewport Properties.

Figure 9-25 summarizes the settings needed to correctly set up the viewport and camera.

As we said, we want this camera to follow an instance of obj_player, so we will use the built-in function accessible directly from the Room Editor. In fact, in the Viewport 0 section, just under the Viewport Properties section, there is a section called Object Following (9-27). Click the three-dot button, select obj_player, and set Horizontal Border to 512 (which is half the width of the camera) and Vertical Border to 384 (which is half the height of the camera). The horizontal and vertical borders tell GameMaker how much space at least there must be between the instance and the horizontal and vertical borders of the camera. Tweaking with these values allows you to choose how strict the camera should follow the player – if you set those values to 0, the camera will move only when obj_player reaches the borders of the camera.



Properties - room0

Room Settings

Inherit

☐ Persistent

✓ Clear Display Buffer

Width	4000
Height	1080

Inherit | Creation Code

Instance Creation Order

Viewports and Cameras

Inherit

✓ Enable Viewports

✓ Clear Viewport Background

Viewport 0

Inherit | ✓ Visible

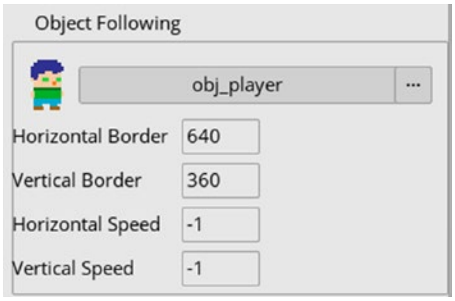
Camera Properties

X Pos	0	Y Pos	0
Width	1280	Height	720

Viewport Properties

X Pos	0	Y Pos	0
Width	1280	Height	720

Figure 9-25.



Object Following

obj_player

Horizontal Border	640
Vertical Border	360
Horizontal Speed	-1
Vertical Speed	-1

Figure 9-26.

Note As many other features, you can turn on the camera following feature also via code using the camera_set_view_target(camera, object).

For example, instead of using the IDE, we could have added this line in the obj_controller's Create event:

```
camera_set_view_target(view_camera[0], obj_player);
```

I'm using the IDE just because it's easier and because – in my opinion – it's always better to not write code, if you can avoid it by using a GUI made just for that job.

Save and run the game! It's easy to notice that the game has zoomed in to Berry and it's following him moving through the level…well back and forth that single platform that we made! But it's working! So this is officially a side-scrolling platformer now, isn't it?

Fixing and re-adapting

Before we go further creating the game, we should make some modifications to the existing code. In fact, there are some objects which are just useless right now and some other that need to be tweaked a bit to work in this new project.

Let's start by deleting obj_ball_red and obj_ball_green. We don't need them anymore; we will create brand-new enemies for this game.

Since balls are not a thing anymore, let's also rename obj_ball to a more appropriate obj_enemy.

Since we deleted the lines related to obj_ball_green, we need to also delete the Alarm 1 event from obj_controller, because it was used to respawn instances of obj_ball_green to the top of the level.

Now head to obj_controller's Create event. Get rid of the lines defining global variables global.cherries, global.cherries_max, global.startx, and global.starty. We won't need those variables anymore since we are drastically changing mechanics.

In fact, to complete a level, we don't want the player to collect all the cherries anymore. We just want them to walk all the way to the end.

We don't need startx and starty global variables either, because when the player dies, we want to reset all the level and we don't want to maintain some sort of memory of the state of the game (collected cherries and others) as we did before. The objective of the game, now, is just to get over the platforming challenges; so, when the player dies, the game should reset any object, enemy killed, and so on.

In the Create event, delete also the lines about room0 and room1. The code for room0 is not good anymore. We will initialize the lives variable in the title screen, so that we can set as the first room whatever room we like and won't have any problem with lives. The code about room1, instead, was intended to work with obj_ball_green that we don't have anymore.

As we stated before and in the GDD, the goal of this game is just to reach the star at the end of the level, so we should make a little modification also to obj_goal, since it's programmed to activate itself only when the player picks every cherry in the level. Let's double-click obj_goal in the Resources sidebar to open it up. Head to the step event and delete these lines:

```
if ( global.cherries == global.cherries_max )
{
    active = true;
}
```

That's it! Now obj_goal is ready to be used in Cherry Caves 2!

Add a new global variable called global.money and assign the value of the global built-in variable score to it. We will use this variable to reset the value of score when the player dies, so that we will set the value of the coins that the player had when starting the level (otherwise, score won't be reset and will continue to basically reward the player for dying by adding more and more points for each death).

When you finished editing, the Create event of obj_controller should look like this:

```
enum states {
    playing,
    paused,
    dead,
    gameover
};
global.game_state = states.playing;

global.money = score;

options = [ "RESUME", "RESTART", "QUIT" ];
opt_number = array_length_1d(options);
menu_index = 0;
```

Now open up the title room and, in the left sidebar in the Properties section, click Creation Code and add this line in it:

```
lives = 3;
global.game_state = states.playing;
```

A room's creation code is a piece of code that is going to be executed when the room is created. It's useful to initialize variables and settings, like we did.

Done! Now, every time we enter the title screen, lives will be initialized to 3.

Now go back to obj_controller and head to the Step event and scroll all the way down to the section in which we check for the game state paused.

In this section, we need to change the effects of the reset and quit options. Edit those lines like this:

```
if ( global.game_state == states.paused )
{
    menu_index += move;

    if ( move != 0 )
    {
        audio_play_sound(snd_menu, 1, false);
    }

    if ( menu_index < 0 )
    {
        menu_index = opt_number - 1;
    }
    else if ( menu_index > opt_number - 1 )
    {
        menu_index = 0;
    }

    if ( enter_pressed )
    {
        switch( menu_index )
        {
            case 0:
                global.game_state = states.playing;
                instance_activate_all();
                break;
            case 1:
                room_restart();
                score = global.money;
```

```
                    break;
                case 2:
                    game_restart();
                    break;
            }
        }
    }
```

Just following this section are the states.dead and states.gameover sections. We need to make great changes to those two. In fact, we don't want anymore the game to show a menu when the player dies, but we just want the text GAME OVER to be shown; and then, if the player presses the spacebar or enter, we want to go back to the title screen.

So modify that section so that it looks like this:

```
if ( global.game_state == states.dead )
{
      global.game_state = states.playing;
      alarm[0] = room_speed * 1;
}

if ( global.game_state == states.gameover )
{
      instance_deactivate_all(1);
      if ( enter_pressed )
      {
             game_restart();
      }
}
```

Now let's move to the Alarm 0 event that we need to change completely. We don't want the game to respawn the player at the starting position as it was before when they die; we want the game to reset the room and the score. So open up Alarm 0 event and get rid of the code inside it and substitute it with this:

```
lives--;
if lives <= 0
{
        global.game_state = states.gameover;
}
else
{
        room_restart();
        global.game_state = states.playing;
        score = global.money;
}
```

Finally, the biggest edit of all, we need to change the code that draws the HUD. In the previous chapter, we used the size of the room to place items on the screen. This can't be done anymore, since now we are using cameras and viewports which are way smaller than our levels.

So, to draw the HUD, we will use now the size of the camera. This means that instead of using room_width we must use display_get_gui_width() and instead of room_height we must use display_get_gui_height().

We need to eliminate the part of the HUD in which we show the number of the cherries collected, and we also want to eliminate the possibility to show the menu when the game state is set on game over, so we need to get rid of that code.

Finally, we want to add a little touch to the pause and game over screens. Since now the level will be made not only by objects but also by tiles, using the instance_deactivate_all() function will make disappear all the objects from the screen and freeze them; but it won't make disappear the tiles, so we would have a bad effect by leaving things like they are right now. To overcome to this problem, we could simply draw a black rectangle to cover the scene, just before starting to draw the menu and status texts.

To draw a rectangle on the screen, we can use the draw_rectangle function that we already used to draw the black band for the HUD.

After those modifications, the code inside the Draw GUI event should look like this:

```
var cam_w = display_get_gui_width();
var cam_h = display_get_gui_height();

draw_set_color(c_black);
draw_rectangle(0, 0, room_width, 40, false);

draw_set_color(c_white);
draw_set_font(fnt_score);
draw_text(20, 10, "SCORE: " + string(score));

draw_set_color(c_white);
draw_sprite_ext(spr_player_idle, -1, cam_w-100, 20, 0.5, 0.5,
0, c_white, 1);
draw_text(cam_w-100, 10, " X " + string(lives));

if ( global.game_state == states.paused )
{
      draw_set_color(c_black);
      draw_rectangle(0, 0, cam_w, cam_h, 0);

      draw_set_color(c_white);
      draw_set_font(fnt_score);
      draw_text(cam_w/2, cam_h/2, "PAUSE");

      for( var i = 0; i < opt_number; i++ )
      {
             if ( menu_index == i )
             {
                   draw_set_color(c_red);
             }
```

```
            else
            {
                  draw_set_color(c_white);
            }
            draw_text(cam_w-200, cam_h-200 + 30 * i,
            options[i] );
      }
}

if ( global.game_state == states.gameover )
{
      draw_set_color(c_black);
      draw_rectangle(0, 0, cam_w, cam_h, 0);

      draw_set_color(c_white);
      draw_set_font(fnt_score);

      if ( lives <= 0 )
      {
            draw_text(cam_w/2, cam_h/2, "GAME OVER");
      }
      else
      {
            draw_text(cam_w/2, cam_h/2, "YOU WON!");
      }
}
```

Ok, now everything's alright. But before we move forward, let's make also a little modification to obj_player. We want to have death pits in this game, so we need to add a small piece of code that allows us to just die when we fall down, so open up obj_player's Step event and add this code to the bottom:

```
if (y > room_height)
{
    global.game_state = states.dead;
    instance_destroy();
}
```

This code will just check if the y variable of obj_player becomes greater than the height of the room; and if it does, it means that the player went down out of the screen, and so it changes the game state to states.dead and destroys the obj_player instance by calling instance_destroy().

You can now Drag and Drop an instance of obj_controller and check that it's now correctly working and showing the HUD and also that you can fall into pits and die!

Ok, now everything is in the right place, and we can continue creating our game!

Different ways to move

One of the funniest things in platformers is that there are different ways to traverse the level. One of those ways is by using different kinds of platforms. There are a whole lot of different platforms like elevators, moving platforms, rotating platforms, trampolines, and so on. In this section, we will implement three kinds of platforms to have some more tools to build fun levels.

The first kind of platform we will make is a trampoline. This special kind of platform, when walked on, bounces the player ceaselessly and allows them to jump even higher by pressing the jump button. One of the first examples of trampolines appeared in Super Mario Bros. (Nintendo, 1987), and it was crucial to reach high places and secrets. Another game that makes a large use of trampolines is Sonic the Hedgehog (SEGA, 1990) that uses them to send Sonic flying through the distance to reach some more advanced point of the level.

The trampoline platform works in an interesting way. Every time the player collides with it, the trampoline triggers the jump action into the player, just like if they pressed the jump button. When the player actually presses the jump button while bouncing on the trampoline, an additional force will be summed to the standard jump force of the player making them jump higher.

Create a new object and call it obj_platform_trampoline and associate to it the sprite spr_platform_trampoline.

Set obj_block as the parent to obj_platform_trampoline, since we want the player to be able to jump on it and not fall through it.

Add a Create event to obj_platform_trampoline by selecting Add Event ➤ Create in the Object Editor and add this one line to it:

```
1   jump_force = 6;
```

This variable represents the additional jump force that we will impress in the player, when they press the jump button.

Now create a Step event and add this code inside it:

```
1   if (place_meeting(x, y-1, obj_player))
2   {
3       obj_player.force_jump = true;
4       if (keyboard_check(vk_space))
5       {
6           obj_player.jspd_bounce= jump_force;
7       }
8   }
```

Lines 1–2: This code checks if the player is on the platform; if they are, a Boolean variable inside obj_player, force_jump, will be set to true. It will trigger the jump function.

Lines 4–7: This code checks if the player is pressing the spacebar. If they are, a variable called jspd_bounce, inside obj_player, is set to the value of jump_force. This new variable, jspd_bounce, will be summed to jspd that decides the height of the jump inside obj_player.

Of course, those two variables (jspd_bounce and force_jump) are not present right now in obj_player. So let's add them, so that we can add the trampoline to the game!

Open up obj_player and add those two lines to its Create event:

```
1   jspd_bounce = 0;
2   force_jump = false;
3
4   Now open up its Step event and modify the jump section so
    that it looks like this:
5   // JUMP
6   if ( grounded and jumping or force_jump)
7   {
8       force_jump = false;
9       vsp = -(jspd + jspd_bounce);
10      jspd_bounce = 0;
11      grounded = false;
12      obj_player.sprite_index = spr_player_idle;
13      audio_play_sound(snd_jump, 1, false);
14  }
```

Line 2: We added as a trigger for the jump the condition that force_jump is true. So now the jump will trigger both if the player is grounded and is pressing the jump button and if the force_jump variable becomes true.

Line 4: When the jump function is triggered, force_jump is set to false (so it won't jump endlessly).

Line 5: The vertical speed of the player is calculated by adding the standard jump force jspd to jspd_bounce that can be changed by the trampoline object.

Line 6: Just after being used, the jspd_bounce must be reset to 0; otherwise, the jump will be higher and higher.

Lines 7–9: These two lines are just as they were before. We just change the sprite to the idle sprite and play the jump sound.

It's all in place! We just have to Drag and Drop an instance of obj_platform_trampoline into the room and test it by running the game!

Figure 9-27. *Berry having fun on his new trampoline!*

The second kind of platform is the falling or disappearing platform. This is a classic of platforming games. You can find it literally in any platformer game ever (even 3D!).

Falling (disappearing) platforms are basically platforms that, when in collision with the player, just fall down after a certain amount of time. This kind of platforms is a very interesting addition to level design, since they both add a challenge and a way to reach distant places. Often, after a session of platforming on those platforms, the player is rewarded with a precious object or a secret.

Falling platforms are actually conceptually easy. They just wait to collide with the player, and when they do, they start to move down until they reach the end of the map; at this point, a new platform is generated in the original position of the one that is falling, and the falling platform is destroyed.

So let's create a new object called obj_platform_falling and assign to it spr_platform_falling.

Add a Create event to the newly created object and write this code in it:

```
triggered = false;
startx = x;
starty = y;
```

The preceding code just initializes a controller variable that we will use to control the generation of the new platform, and it sets the starting coordinates that we will use to generate the new platform in the right place.

Add a Step event to obj_platform_falling and add this code in it:

```
1     if ( place_meeting(x, y-1, obj_player) and !triggered )
2     {
3             alarm[0] = room_speed * 1;
4             triggered = true;
5     }
```

Lines 1–5: This code just checks for the collision with the player (when they are just above the platform) and triggers only if the triggered variable is set to false. If those conditions are satisfied, an alarm is set to 1 second, and the triggered variable is set to true.

We need the alarm to make the platform fall after a second, so that we will give the player the possibility to escape.

Add a new Alarm 0 event to obj_platform_falling and add this code to it:

```
1   move_towards_point(x, y+1, 10);
2   alarm[1] = room_speed * 0.5;
```

Line 1: We move the platform downward at a speed of 10 pixels per frame.

Line 2: We set another alarm to trigger in half a second.

The new alarm 1 will be used to generate the new platform and destroy the falling one. Add a new Alarm 1 event to obj_platform_falling and add these lines to it:

```
1   if not place_meeting(startx, starty, obj_player)
2   {
3       instance_create_layer(startx, starty, "Instances", obj_
        platform_falling);
4       instance_destroy();
5   }
6   else
7   {
8       visible = false;
9       alarm[1] = room_speed * 0.2;
10  }
```

Lines 1–5: When the alarm triggers, we check if the player's character is where the new platform should be created, because we don't want to pin them creating a platform where they stand. If the area is free, a new platform is created at the original coordinates, and the falling platform is destroyed for good; otherwise, the falling platform becomes invisible, and the alarm gets reset, so that we can check again in 0.2 seconds.

You can personalize this platform by adding a particle effect in the Destroy event, so that it looks like the platform crashed to the ground.

Now let's test the new platform by dragging and dropping it in room0, possibly in the pit we created while drawing the level.

Save and run the game! The platform should be there waiting for the player to walk on it. When they do, the platform waits a second and then falls down (Figure 9-27), and a brand-new platform is generated in its place. Awesome!

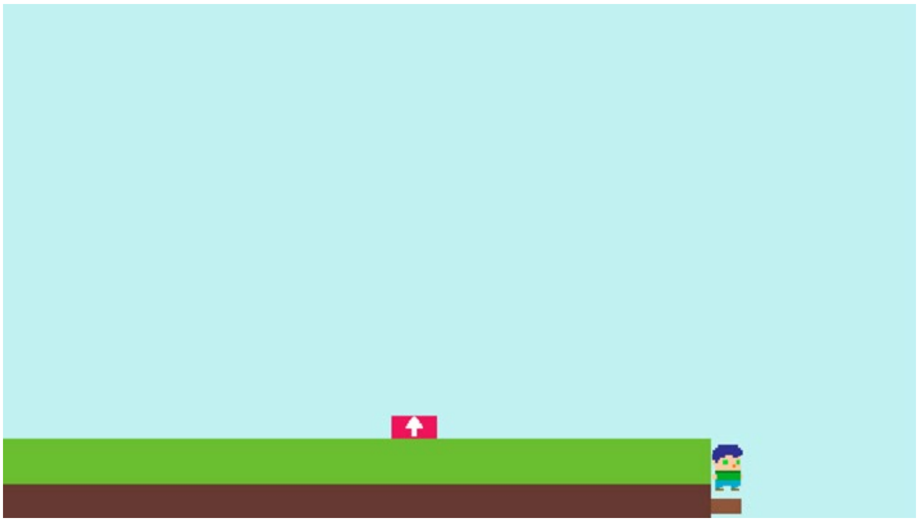


Figure 9-28. *Berry is falling with the falling platform…oops!*

Last, but not least, one of the most useful and fun platforms you can find in a platformer game is the moving platform. It's a super-useful platform that moves back and forth allowing the player to reach distant places giving also a bit of a challenge, because the player is forced to jump on it while it's moving.

The movement of the platform is really easy, but the problem is that just moving the platform won't move also the player who's on it. In fact, we assume they will just because in the real world that's how it works. But actually, our game doesn't work with real-world physics. Don't forget that every single thing moving or falling is programmed to do it and there is not a physics engine that's governing forces. We can put any object in the middle of the map, and it won't fall. Only the player's character will, because we programmed it to do it.

So we need to simulate physics even with moving platforms. How can we do this? Well, we can borrow the idea we used to create trampolines and apply a horizontal movement in the direction in which the platform is moving when the player is on the moving platform.

Let's create this new object, call it obj_platform_moving, and add a Create event to it with this line:

```
1   speed = 2;
```

We change the speed so that the platform starts moving. We will invert the direction in the Step event when needed.

Now add a new Step event to obj_platform_moving. This is the main event of the object and will feature the whole mechanic. So add this code in it, and let's analyze it:

```
1   if (place_meeting(x, y, obj_block))
2   {
3       speed *= -1;
4   }
5
6   if   place_meeting(x, y-1, obj_player) or
7       place_meeting(x-1, y, obj_player) or
8       place_meeting(x+1, y, obj_player)
9   {
10      obj_player.hsp_carry = speed;
11  }
```

Lines 1–4: We are letting the platform move in a direction until it finds an instance of obj_block. When this happens, the platform inverts its direction (line 3).

Lines 6–11: Here we check if the player is on the platform. If they are, the value of the movement of the platform is assigned to obj_player's hsp_ carry variable which sums up to the horizontal movement of the player's character.

There's a little problem with this platform. Right now, it works right only if we put it between two instances of obj_block. This means that we cannot make some interesting platforming sessions where the player has to jump between many moving platforms without having the possibility to rest on a solid instance of obj_block_red or obj_block_brown. We can fix this and gain some more level design possibilities by creating a special object which will work as an invisible marker for our moving platforms.

Create a new object called obj_marker and assign to it a square sprite. It can be whatever sprite you want, since it's going to be invisible. In fact, tick on the box labelled *Invisible* in the object's Object Editor.

Now go back to obj_platform_moving's Step event and modify the code like this:

```
if (place_meeting(x, y, obj_block) or place_meeting(x, y,
obj_marker)
{
    speed *= -1;
}

if   place_meeting(x, y-1, obj_player) or
    place_meeting(x-1, y, obj_player) or
    place_meeting(x+1, y, obj_player)
{
    obj_player.hsp_carry = speed;
}
```

All set, now you can place the moving platform between two instances of obj_marker, and the platform will go back and forth between them.

To make all this work properly, though, we have to create the hsp_carry variable into obj_player, like we did with jspd_bounce; and we have to add this value to the horizontal movement of the object.

Let's open up obj_player and head to its Create event and add this line:

```
hsp_carry = 0;
```

Now open the Step event and add these lines just before the horizontal collision check:

```
hsp += hsp_carry;
hsp_carry = 0;
```

Cool! Looks like it's all ready to test this out!

Add an instance of obj_platform_moving to room0 and test that it's correctly working. You should be able to move back and forth together with the platform.

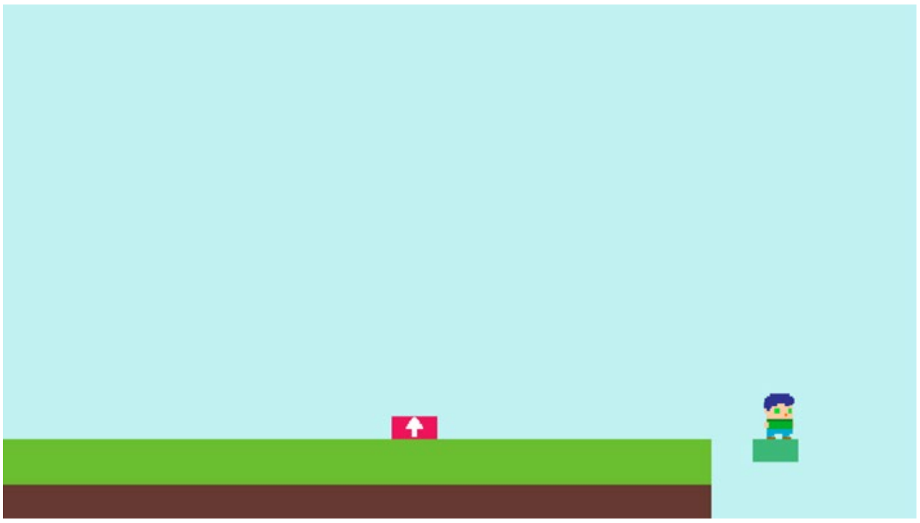


Figure 9-29. *The moving platform is correctly working!*

Cool! We created three new platforms that we can use to design fun and challenging levels!

Now we just need some enemy to fight…or to escape from!

Gotta squash ‘em all!

In Cherry Caves (Chapter 8), we created very simple enemies that just rolled around in the level. You couldn’t possibly fight them, but they could kill you with just a touch. Not fair, right? Let’s create some enemies that can be fun to face.

Our new enemies are the octopus aliens we already talked about in the GDD and, as we already said, will have two simple patterns: moving left and right and jumping up and down. The big difference, anyway, is that they can be killed by jumping on their heads.

Let’s start by creating a new object named obj_octopus_green. Assign to this object spr_octopus_green and set obj_enemy as its parent.

Add a create event to this object and initialize those two variables in it:

```
dir = 1;
spd = 4;
```

`dir` is, as always, the direction in which the enemy will move; and spd is the speed at which it will move.

The enemy we are creating will move in a direction until they find an instance of obj_block; when it finds it, it will change direction and continue moving.

To implement this behavior, we need a Step event. Let’s add it to obj_octopus_green and add this code to it:

```
if ( global.game_state == states.playing )
{
    if place_meeting(x, y, obj_block) or place_meeting(x, y,
    obj_marker)
    {
        dir *= -1;
        image_xscale = image_xscale *-1;
    }
8       x += spd * dir;
9   }
```

The preceding code is pretty self-explanatory. The object checks whether the game is in playing status; and if it is, it moves spd*dir pixels until it finds an instance of obj_block or obj_marker (line 3). When this happens, the object inverts the moving direction by multiplying dir by -1 (line 5) and flips its own sprite using image_xscale.

Finally, as always, add a Destroy event with a particle effect to make it more gore:

```
effect_create_above(ef_firework, x, y, 1, c_purple);
audio_play_sound(snd_squash, 1, false);
```

Ok, the enemy is ready! The only thing left is the interaction with the player. We will code it into obj_player, by using the collision with obj_enemy.

Double-click obj_player and open up the collision event with obj_enemy. In this event, add the following code to create the possibility to squash squids!

```
if ( y + sprite_height/2 ) < other.y
{
    instance_destroy(other);
    force_jump = true;
}
else
{
    global.game_state = states.dead;
    instance_destroy();
}
```

Lines 1–5: In these lines, we check if, at the moment in which the collision happens, the instance of obj_enemy is just below the player. If that's the case, the enemy gets destroyed and the player bounces.

Lines 6–10: If, when the collision happens, the enemy is not below the player, this means that the enemy is touching the player and so the player should die; so the game state gets changed to states.dead, and the player instance is destroyed.

That's it! Open up room0 and add an instance of obj_octopus_green and put it between two blocks (instances of obj_block) or two markers (instances of obj_marker) as shown in Figure 9-30, and test what we did until now!

The enemy will move back and forth between the two blocks or markers (Figure 9-30) and kill the player when touched. If the player jumps on it, the enemy dies in a purple gore explosion.

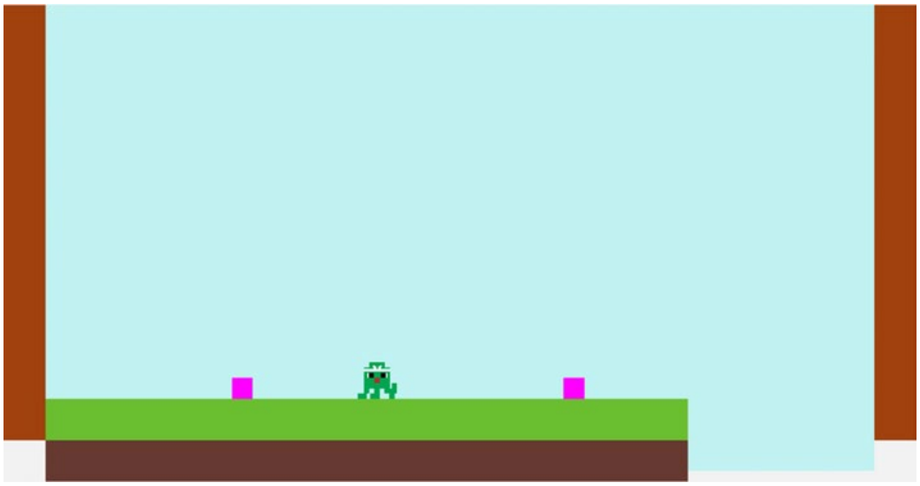


Figure 9-30. *The enemy has been placed between two instances of obj_marker (the magenta squares)*

The second kind of enemy that we want to create is the purple octopus. This is an octopus that jumps very high and clings on the ceiling; then, it slowly falls down to the ground again – and repeat.

To create such a goofy enemy, create a new object called obj_octopus_purple and set as its parent obj_enemy.

Add a create event to this new object and add this code to it:

```
1   dir = -1;
2   spd = 4;
3   wait = false;
```

Lines 1 and 2, as usual, are about direction and speed that we will use to program the up and down patterns, while the wait Boolean variable is used to make the octopus wait a bit before it falls down or jumps up on the ceiling again, so that we can give the impression that it's really jumping, grabbing on the ceiling, and falling down again.

The majority of the logics is, as usual, in the Step event. Let's add one to this object and write up this code in it:

```
if ( global.game_state == states.playing )
{
    y += spd * dir;

    if ( place_meeting(x, y, obj_block) and !wait )
    {
        wait = true;
        spd = 0;

        image_yscale = image_yscale *-1;
        image_index = 0;
        alarm[0] = room_speed * 1;
    }
}
```

Lines 1–3: As usual, if the game is in the playing state, the instance moves spd*dir pixels.

Lines 5–13: We want the instance to stop when it collides with a block and flip vertically its sprite, so that it gives the impression to be attached to the surface, and then we set alarm 0 to 1 second. Alarm 0 will make the octopus move again in the opposite direction and reset the wait variable, so that the octopus can splash on another surface.

Create an Alarm 0 event and add this code in it:

```
spd = 4;
dir *= -1;

image_index = 1;
wait = false;
```

Lines 1–2: After 1 second, the movement speed of the octopus is set back to 4 (the original value) and the direction is inverted, so that it can move in the opposite direction.

Lines 4–5: The image index is set to the appropriate frame, and the wait variable is set to false, so that the condition in the Step event can be triggered again when the instance will find another obj_block instance.

Once again, let's add a Destroy event with a particle animation, just like we did in obj_octopus_green using the same line of code:

```
effect_create_above(ef_firework, x, y, 1, c_purple);
audio_play_sound(snd_squash, 1, false);
```

Ok, it's done! Let's test it by dragging and dropping it in room0 and placing a block some tiles over it. Starting the game, the octopus will start to jump up and down splashing and grappling to the block and falling down again (Figure 9-31). This kind of enemy can be killed when it's on the ground, before it jumps up to grab the ceiling (or the block), just like obj_octopus_green.

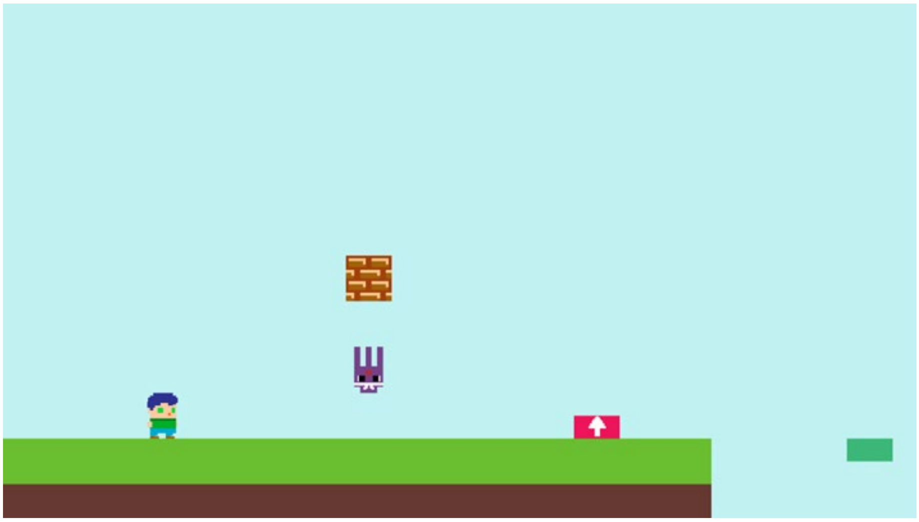


Figure 9-31.

We created a new enemy! It was very easy to make, but it feels original and very different from the other one. Actually, it uses some tricks that make it feel more alive than obj_octopus_green, so it's a very good addition.

This is a very good example of how you can create enemies and interactions with little effort.

Tip Very often, good game design is not about creating complex things, but things that feel credible and appropriate. The objective of a good design is to immerse, entertain, and amuse the player; and you can do it often with simple things.

Items and power-ups

A very important feature of platformer games is the possibility to interact with items and power-ups. Items are usually objects that can be used to do something. The most popular items are collectibles like coins that are often used to buy things from merchants and special items like keys that are used to open special doors and unravel other hidden parts of the game world.

Power-ups are another kind of beast. They allow the player to gain temporary bonuses and powers and so to face levels, enemies, and all the game's challenges in different ways. This adds a layer of variety that makes the player feel in control of their decisions; and this leads them to experiment, play in different ways, and have fun with the game, which is our primary objective as GameMakers, so yay! Let's create some items for our game!

Coins

Coins are the most common collectible items in video games. Started just to track the score, they became a way to mark the path that the player is supposed to follow, like breadcrumbs. Nearly every platformer has some sort of coins used with this scope in mind or as a currency to buy items from in-game merchants or shops.

We are going to implement our own coins too! Our coins will have a nice rotating animation and will jump up when picked up (like in Super Mario Bros.). The only effect that these coins will have will be to grow the score and to mark the path to follow to get to the end of the level.

Let's start by creating a new object named obj_coin. Assign to it the spr_coin sprite and add a Create event. In this event, write up this single line of code:

```
value = 1;
can_score = true;
```

At line 1, we have the value of the coin. When picked up, it will add 1 to the score.

At line 2, we have a controller value that we will use to avoid that a coin can score more than once because of a prolonged collision with the player.

Add also a collision event with obj_player by clicking Add Event ➤ Collision ➤ obj_player and add this code in it:

```
1   if can_score
2   {
3       can_score = false;
4       score += value;
5       image_speed = 0;
6       image_index = 2;
7       image_xscale = 0.5;
8       move_towards_point(x, y-1, 10);
9       alarm[0] = room_speed * 0.1;
10      audio_play_sound(snd_coin, 1, false);
11  }
```

In the preceding code, we check if the coin can score. If it can, we set the controller value to false, so that it can't score more than once. Then we increase the score of the player by the value of the coin. Finally, we play an animation by scaling the sprite and moving it upward (5–7) and finally destroying it thanks to an alarm (line 9).

Create an Alarm 0 event and add a call to instance_destroy() in it:

```
instance_destroy();
```

Coins done! Let's put them in the room and test them (Figure 9-32). By running the game, you should see them rotating; and when picked up, they get pulled up in the air and then vanish.

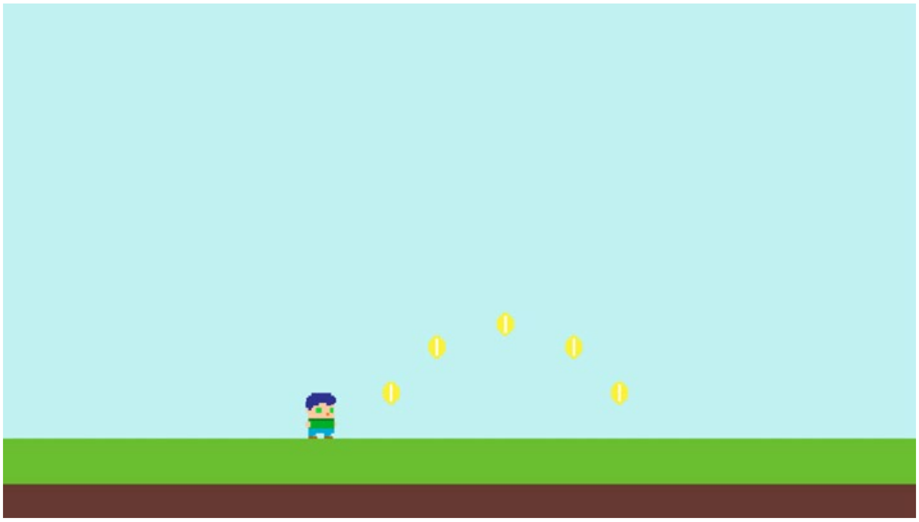


Figure 9-32.

Cherries

Cherries are our power-up for this game.

As per the GDD, they allow the player to become invincible for a small amount of time. During this invincibility phase, Berry can kill every enemy by just touching them.

Creating this power-up is very simple. Firstly, we need to create a powered_up Boolean variable that tells us whether Berry is powered up or not.

Open up obj_player's Create event and add this line at the bottom:

```
powered_up = false;
```

Now, add a collision event with obj_cherry by clicking Add Event ➤ Collision ➤ obj_cherry. When obj_player touches an instance of obj_cherry, we want obj_player to enter in powered-up mode for a small amount of time… Let's say 5 seconds. So we need to set up an alarm that can switch off powered_up and so deactivate the powered-up status. Add these two lines of code in the collision event:

```
powered_up = true;
alarm[0] = room_speed * 5;
audio_play_sound(snd_frenzy, 1, true);
```

Now create an Alarm 0 event and add this line in it:

```
powered_up = false;
audio_stop_sound(snd_frenzy);
```

When the player touches a cherry, the powered-up status is triggered, but we have to add some effects to it. Let's start by creating a fun color-frenzy animation on Berry, just like the effect of the Super Star in Super Mario Bros.

Open up the Step event of obj_player and add this code just below the code that manages the ladder:

```
if ( powered_up )
{
    image_blend = make_color_rgb(random(255), random(255),
    random(255));
}
else
{
    image_blend = -1;
}
```

To create the rainbow color-frenzy effect, we use image blend, which is a property of any sprite that applies a filter to the image. We are applying to it a random color created using the make_color_rgb() function by generating three random values for red, blue, and green using the random() function. This effect is applied once per frame, so the result is a crazy succession of random colors.

When the powered_up variable is false, we set back image_blend to -1, which means no filter gets applied to the image.

Now we only need to add the code to kill every enemy when the player is powered up.

Open up the obj_player's collision event with obj_enemy and change the code like this:

```
1   if ( !powered_up )
2   {
3       if ( y + sprite_height/2 ) < other.y
4       {
5           instance_destroy(other);
6           force_jump = true;
7       }
8       else
9       {
10          global.game_state = states.dead;
11          instance_destroy();
12      }
13  }
14  else
15  {
16      instance_destroy(other);
17  }
```

We added the check to the status of the powered_up variable. If it's true, we instantly kill every enemy (lines 14–17). If the player is not powered up, the same check that we did before is executed.

Everything is in the right place, and you just need to test this out.

Drag and Drop a cherry into room0 and run the game. You will verify that picking up the cherry will put Berry in a frenzy mode for 5 seconds and in this status he can kill every enemy by just touching it (Figure 9-33).

Amazing!

Figure 9-33. *When powered up, Berry can kill enemies by just touching them!*

Creating the first level

Now that we have a complete 2D platformer engine, the only thing left is to create a nice first level.

The first level is always an interesting challenge. It's the first time the player enters the game world you are creating, and so it's the first time your creation embraces them. It's easy to make an overwhelming or underwhelming first level. Many inexperienced level designers tend to put too many or too few elements into the first level or to not place them properly. What should be clear to the level designer is that the first level should be introducing both the atmosphere and the possibilities that the game offers.

In our case, we have to introduce the player to the colorful world of Cherry Caves and its crazy characters, but also to the game mechanics and the various elements we created, like the different platforms and the power-up.

We can actually put everything in the first level, and we will. The important thing is just to place things in a harmonious way, so that the level feels natural and linear and not chaotic.

The first thing we want to introduce to the player is the jump function. So we just put a pile of blocks in front of them (Figure 9-34). They will try the keys and jump through it. This will make the player feel competent because they figured out how to get over that first obstacle without explanations.

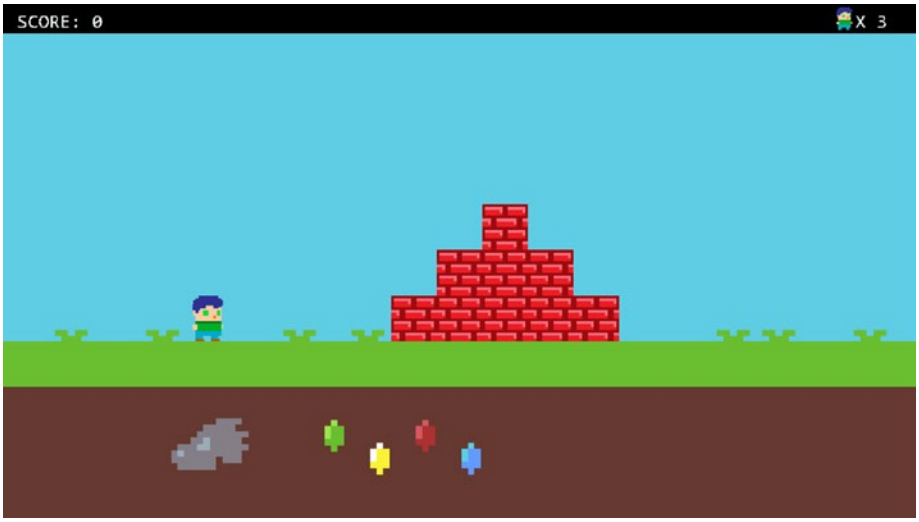


Figure 9-34.

Once the player learned how to jump, they should learn how to climb ladders; but maybe we can contextualize this, by creating a little structure made of bricks that we can climb with the ladder. We can exploit this idea to introduce the concept that risking can be rewarding. So, just under this brick structure, we add an optional cherry that the player can try to pick up by facing both a falling platform and a jumpy octopus.

The fact that this is an optional way is made clear by the line of coins that are just on the top of the structure. The level design screams "climb up there and pick the coins," but it also silently offers the player the possibility to get an object that the player suspects may be of some importance in the game (because, you know, the game has cherry in the title, so cherries must be relevant, somehow).

In this small space, we introduced five concepts of the game, but it doesn't feel confusing. It's very clear, and the player doesn't feel forced to face the octopus to get the cherry. They can avoid getting in there and just climb the ladder, pick up the coins, and kill the easy-to-kill octopus. Because of how the second octopus is positioned, the player will probably kill it by accident by falling on their head and discovering that you can squash enemies by jumping on their heads.

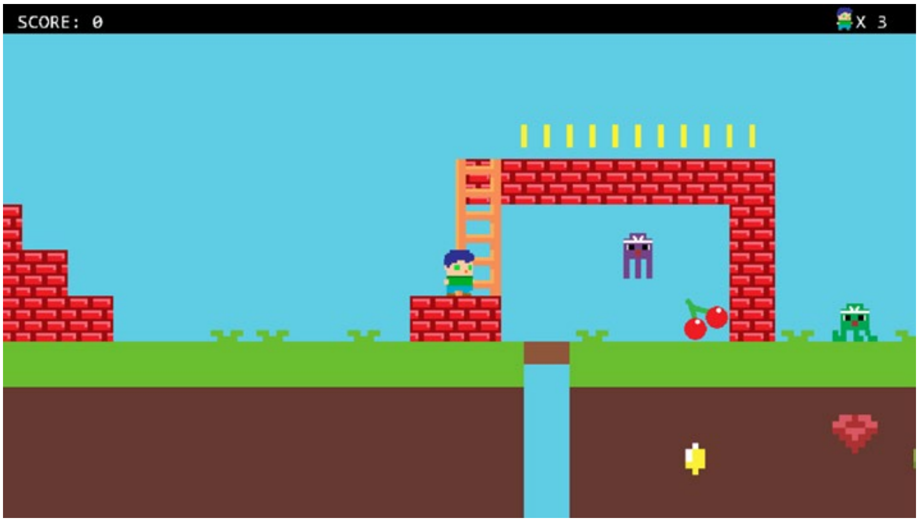


Figure 9-35.

After that condensed learning session, the player would probably enjoy a platforming session without further distractions. So we can, for example, put some pits here and there or some suspended platforms to jump on.

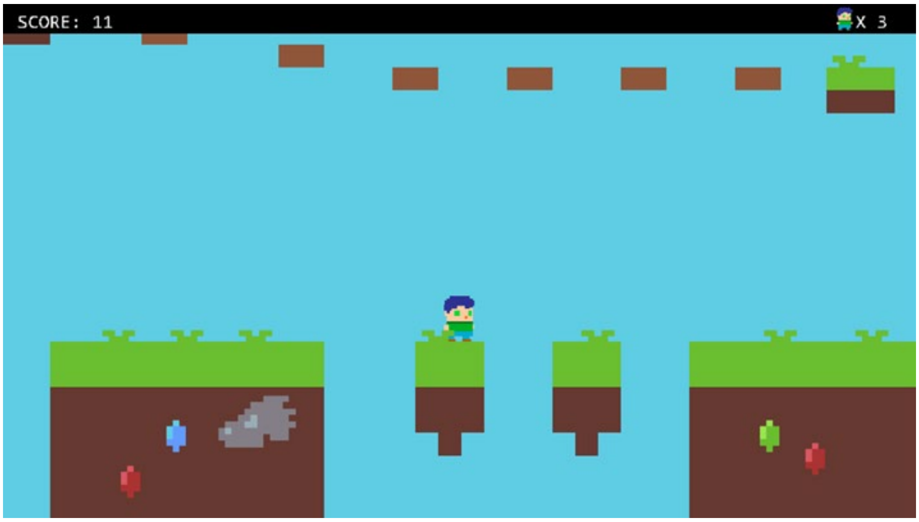


Figure 9-36.

An interesting idea is to create a suspended path above the ground that can be reached only using a trampoline and then present the player a platforming session using falling platforms.

After that demanding platforming session, we can present the player the more comfortable moving platform and then finally the star that allows them to complete the level.

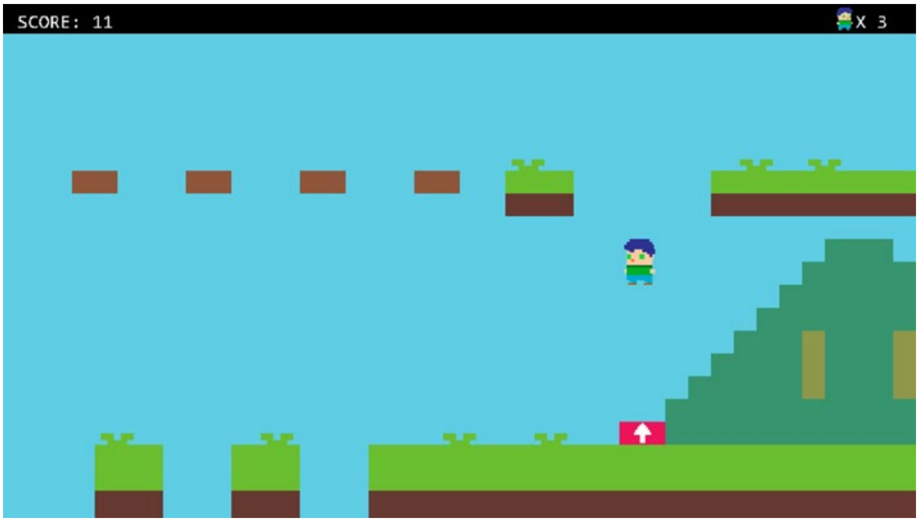


Figure 9-37.

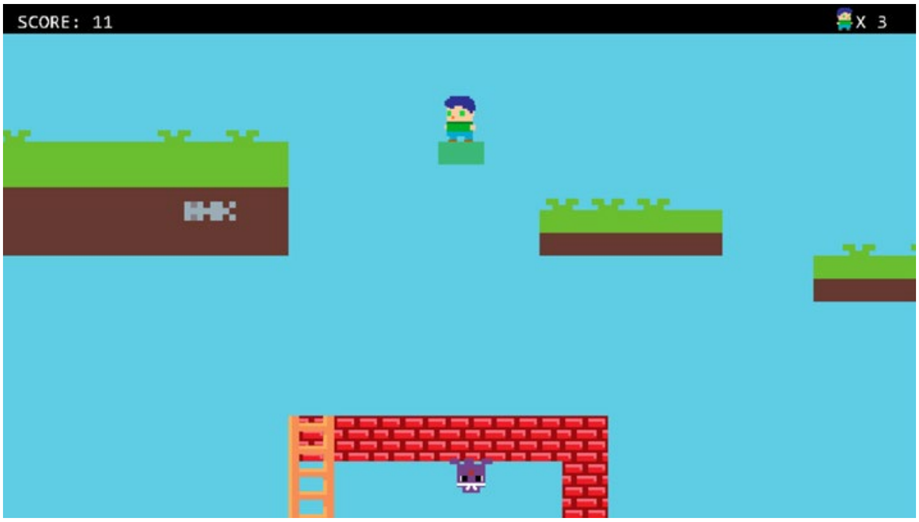


Figure 9-38.

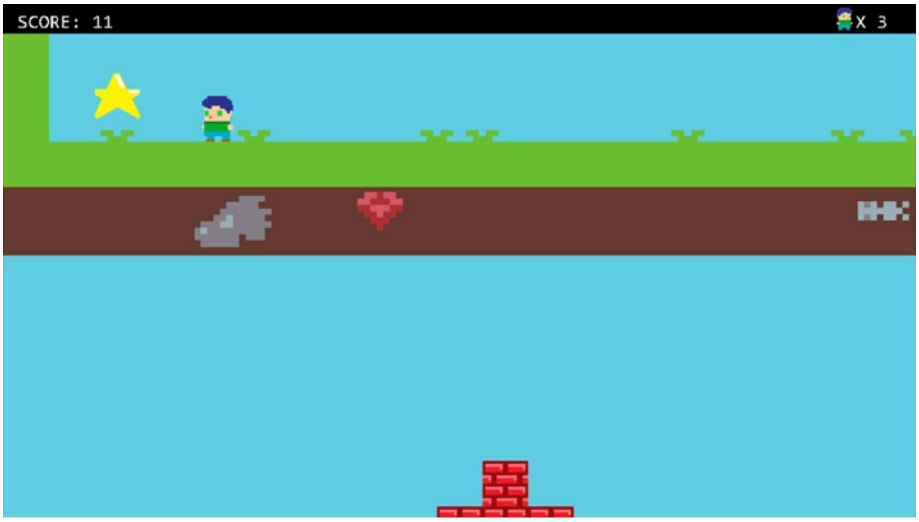


Figure 9-39.

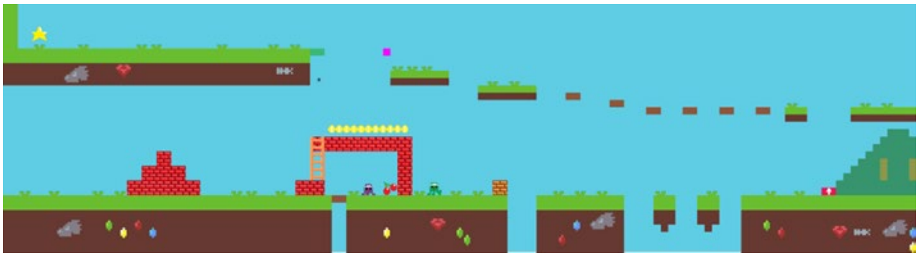


Figure 9-40.

The proposed first level is represented in its entirety in Figure 9-40, and it's a room of size 4000 × 1080.

Enjoy this new game we created building on top of Chapter 8, improving the gameplay and adding new features while learning more things about game development and GML programming.

I invite you to experiment and think about new ways to entertain the player with the gameplay elements you built. Try to create interesting and fun encounters with enemies, puzzles, and platforming sessions using the tools we built in this chapter.

In the next chapter, we will continue our discussion about good platformers' design. We will explore the most popular and important games of the genre, trying to understand which one of them nailed it. We will talk about all the most important aspects of a good platformer, increasing our knowledge on the matter.

Finally, in Chapters 11 and 12, we will use all the knowledge we accumulated to create a metroidvania game, which will borrow a lot from our studies on good platformers' design.

TEST YOUR KNOWLEDGE!

1. Why is the title screen so important to a game?
2. How can you add a background image to a room?
3. How can you set the fullscreen mode in your game?
4. How can you force a specific resolution in your game?
5. What is a tile set?
6. How can you create a tile set in GameMaker Studio 2?
7. Why are tile sets good to create levels?
8. Tile sets cannot be used to manage collisions; so how can you stop your player from falling, if you use them to build your levels?
9. How can you set the camera to follow an object?
10. Can you describe how the trampoline platform works?
11. Elevators are another common type of platform in video games. They move up and down and can be used by the player's character to reach high places. Using the principles explained for the moving platform, can you create an elevator platform?

12. In Cherry Caves 2, you take damage when you touch an enemy, but you can kill them by jumping on their head. Can you explain how this system works from a technical point of view?

13. How does the image_blend property works?

14. What do you think can be improved in the level we created at the end of the chapter?

15. Can you create a second level that reinforces the basic gameplay concepts of the game by offering some more complex challenges?

CHAPTER 10

Designing Platformers

In Chapter 9, we mentioned the importance of the cognitive flow in a game. We identified the cognitive flow in that condition in which the player is totally immersed into the activity of playing. While in this status, the player senses a loss of self-awareness and extreme focus on the task and loses track of the time passing. In this phase, the player is constantly learning, and this gives them amusement.

As game designers, we want the player to enter and remain in the cognitive flow, and we can accomplish this task by leveraging on game design and making use of some interesting tips.

Controls are key

Concerning platformers, the feeling of controls is the most important thing. In fact, controls not only define how you interact with the game, but they're also responsible of your own immersion in it. Games with tight controls let you forget quickly that you are playing a game and put you straight into the cognitive flow. You feel confident and natural while traversing the game world, and you are ready to face challenges more and more difficult. In fact, even when you have to accomplish a hard task, knowing that you can count on your instinct and the fairness of the controls and the game system makes you feel like you can do that and gives you the motivation you need to complete the task.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_10

In particular, the movement of the player should be precise and responsive. The golden rule is to make the character move when the button is pressed and make it stop as soon as the button is released. This will give the player control over the entity of the jump; and so it will make the player feel into the game, moving naturally and instinctively, boosting their capacity to immerse in the game.

The same principle is used to develop the perfect jump. In fact, for most of the gamers, jumping should be dynamic, allowing the player to choose the height of the jump. This translates into the natural instinct of impressing the right amount of force in the jump to accomplish different tasks.

Jumping at different heights gives the player more possibilities in both exploration and movement. In general, this allows the level designer to create levels that push the player to constant problem-solving making the experience fun and challenging. The golden rule is to regulate the height of the jump according to the amount of time the player holds the jump button.

In our platformer games (Cherry Caves and Cherry Caves 2), the jump is very basic: you always jump at the same height, no matter how much you press the button. This is going to change in Chapter 11, where we will create the perfect jump!

Other than choosing the height of the jump, the player should be able to move in the air just like they move on the ground. Adjusting the falling trajectory is crucial for precise platforming. Giving the player the power to move while in midair, they will feel a sense of autonomy and control that will keep them focused on the game and so in that status of cognitive flow that we discussed in previous chapters.

All those characteristics were introduced by Super Mario Bros. (Nintendo, 1985); and, since that moment, they started to be implemented in nearly all the platformers and became the difference between good platformers and bad platformers.

One of the best modern examples of controls made right is Super Meat Boy (Team Meat, 2010). The game became famous for featuring a very challenging platforming and ultraprecise controls. Playing Super Meat Boy is a matter of raw instinct and precision. Every second playing it is a second improving at it. The player develops the skills needed to win the level just with repetition and training, and the controls are so tight and well made that you will never end up blaming the game to be unfair. Every death is entirely your fault. And this takes us to the next important requisite of a good platformer: fairness.

It’s my fault!

When you play a game, it's crucial to feel that you're playing in a fair environment with consistent rules. This is important to take decisions and create your strategy, which can only be done based on the rules of the game. If the player can’t understand the rules, they can’t learn and become better at the game, thus never being able to advance. This makes the game pretty pointless, doesn’t it? That's why it's so important that the player feels like they're playing in a fair environment and that if they lose, it's because they did wrong and not because the game is buggy or glitchy or behaves randomly.

This feeling of unfairness, in platformers, can be found in mainly two activities: moving around and colliding to objects.

As we already saw, a platformer's gamer expects to move in a certain way, with a certain precision. It's not acceptable to slip over a pit because of the imprecision of the controls, and it's not acceptable to have just a standard jump height or to be unable to move in midair. This is the first thing to check out when designing a good platformer game: to give the player the tools to effectively move around the map in freedom.

Collisions are the second important thing to keep under control. A good collision mask is not always the one that best fits the sprite, but it's the one that reflects the perception of the player. As shown in Figure 10-1, a good collision mask covers not the entirety of the sprite, but just the main section of the body, often leaving out the arms and legs.

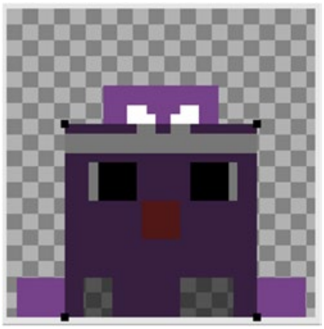


Figure 10-1. *A fair collision mask for our purple octopus!*

Together with controls and collision masks, the camera plays an important role in determining the fairness of the game. The camera should always show you everything you need to see in the game to play and win.

When a game camera fails at its job, it’s likely that the player will pay the consequences. When the player gets punished because of the camera, the player feels cheated and frustrated by the whole game system. It’s a very bad feeling that causes the player to label the game as bad and give it up.

We can observe a bad example of camera management in the first Dark Souls. The camera system in Dark Souls 1 (DS1) is mostly manual, but the problem with it is that in some frantic combat situations, the camera totally glitches and starts to play against you – often causing your death.

I agree. Dark Souls is a 3D game, and it’s not even a platformer; but that’s not so important, in this case, since the concepts that we are talking about are applicable to a wide variety of genres. In fact, regardless of the game genre, a good camera system should always put the player in the condition of being aware of the environment around them so that they can take the appropriate action.

Back in the golden era of platformers, there were games like Mega Man, Double Dragon, Ghost 'n Goblins, and Contra that because of their 4:3 aspect ratio and low resolution (256 × 240 pixels) didn't show much space behind and in front of the character. Because of that technological limit, the player had the problem of having very little time to react to incoming enemies, and that felt unfair. If this wasn't enough, some games also featured an unfair enemy-spawning logic. For example, in Ghost 'n Goblins, the enemies had a chance of spawning under the feet of the player, dealing instant damage. This is a terrible design choice! The player should always be able to avoid damage and win the game. It's very important to think about how to avoid unpleasant situations when dealing with randomized situation, like this one. In this case, it would have been enough to check the position of the player just before spawning the enemy, making sure that the player was not there. We did exactly the same thing in Cherry Caves 2 with the falling platform, to avoid creating it on the player's character. Sometimes even little precautions like that can change the perception of a game from fair to unfair and vice versa.

Keep it simple!

Platformers are often very fast games, and sometimes they even grow in complexity. Just think about the metroidvania sub-genre: it's a platformer in which the player moves by jumping and climbing; but it also features RPG elements like character's statistics, equipment, inventory, and map. Because of the quantity of the information to show and the pace of the game, it's important to have a clean and simple HUD that can display all the information needed without overwhelming the player or confusing them. The player must be able to quickly locate themselves on the map, manage the inventory, understand statistics, and so on. This is crucial in a good game: keep the user interface simple!

We did a good job with this in all the games we created, by always showing information in the simplest way possible.

An example of a simple and clear HUD in a commercial game is the one featured by Diablo III (Blizzard Entertainment, 2012).

The HUD only covers the bottom of the screen and consists in the following:

- A sphere containing blood, which represents the player's health
- A sphere containing the main resources used by the player to attack (e.g., Mana, Fury, Spirit, Arcane Power, etc.)
- A long horizontal bar filling as the player gets experience, which represents the progress toward the next level
- A set of buttons representing the active skills, labelled with the key that you can use to activate them
- A heal button labelled with the key you can use to heal
- A teleport button that you can use to teleport back to the encampment (safe place)
- A set of buttons that open some game screens (e.g., character view, inventory, skills view, quest journal, and main menu)

It's a very simple and intuitive interface; all you need to know to play the game is right in the bottom of the screen. It doesn't feel confusing or tricky to understand; with just a glimpse, you can tell the amount of hits you can take and if you can heal or attack. You can build a strategy in a matter of seconds thanks to the clarity of the interface. This is so crucial, if the game has fast-paced combat and challenging battles, like Diablo III.

Power-ups, items, and gear

Either if your game is a classic platformer like Super Mario Bros. or a complex metroidvania in the style of Castlevania: Symphony of the Night (Konami, 1997), your game needs some items to be enjoyable. Special items like power-ups, gadgets, or even gear and equipment add a lot of variety to the game and allow the player to face the challenges and advance the story in their own style, by choosing the path that better fits them.

Items, gear, and power-ups are extremely important to add diversification to basic tasks of the game; and they help the player to feel in charge and in autonomy, keeping them in the status of cognitive flow.

Super Mario Bros. 3 does a great job at that by offering the player a big number of power-ups. Every power-up gives Mario new powers that he can use to defeat enemies or to overcome some tricky area.

The variety and effectiveness of each power-up give the player a lot of possibilities to take on problems in different ways allowing the player to choose the one that best suits their own playstyle. Power-ups give the player the freedom to experiment and play with the rules, which leads to amusement and fun.

Interesting collections

A thing that too many platformers do wrong are collectibles. Too many games treat them just as things to pick up while traversing a level and don’t understand what collectibles really represent: rewards.

Collectibles are the reward a player is given for exploring a level, for winning a challenge, for killing a hard enemy, or for finding a secret in the level. They are things earned with commitment and skills; they can’t be pointless! They must have a meaning or a purpose. Some interesting applications for collectibles are to unlock a bonus level, gift the player some rare item or skill, or even advance a specific plotline. They can’t be collected just for the sake of it.

A good example is Klonoa: Door to Phantomile, in which by saving all the Phantomilians, you are rewarded with a challenging time-based extra level and a new Cutscene. Completing the extra level, you will unlock Lephise's Jukebox that allows you to play the entire soundtrack song by song – which I personally saw as an awesome reward, back in the days. Apart from my personal opinion, that's a good example of how to reward for collecting items. It adds new content (a whole new level with a brand-new game mode), it gives more details about the background story of one of the characters of the game (the Cutscene), and it even unlocks the whole OST (Original Soundtrack). Other than that, the act of saving all the Phantomilians is not even annoying, because you find them trapped along the way and they are rarely hidden. The only thing that the game asks of you to save them all is using your skills. If you have good double/chain jump skills, you can get to any Phantomilians in the game.

Moreover, the extra level that you receive by saving all the Phantomilians is a very good example of gameplay-based level design. It's built all around chain/double jump skills, and it's a lot of fun to play and replay, since it features a leaderboard with all the completion times registered. The extra piece of story told through this level and the extra Cutscene is just plain fan service, which is probably one of the most appropriate rewards you can deliver to a person who loved the game so much and got to its very end. The game also rewards you with the full soundtrack that you can play using Lephise's own gramophone whenever you want.

Klonoa shows an example of how you can deliver an interesting collectible experience without stressing the player or forcing them to ask for help (like Googling for those guides about how to collect every item), but betting everything on skills and rewarding them fairly for their efforts.

Another example of collectibles done right can be found in Batman: Arkham Asylum (Rocksteady Studios, 2009). In this modern classic of the Dark Knight, you have the optional task to collect the Riddler Trophies. They are scattered all over Gotham, and collecting all of them takes you a step closer to the Riddler himself. Every time the player collects a Riddler

Trophy, they get a new clue about the location of the Riddler; and when they collect all of them, the location of the Riddler's hideout is unveiled. This achievement unlocks the Riddler's boss fight.

This is a wonderful example of how collectibles should be done because collecting them all unlocks new game content and even if the trophies are hidden, they are never impossible to find; in fact, you just have to pay attention to the environment.

The positioning of collectibles is crucial to their positive or negative perception in the eyes of the player. For example, a bad way to do it is to scatter them in huge open world maps. This is bad because open world maps take long to explore, especially if there isn't a good way to fast travel. In fact, exploring those maps can take so long that it has become very common, in open world games, to implement fast travel via some kind of teleport or offering the player mounts to move faster.

If a level is very big, collectibles should be placed conveniently to guide the player through the relevant paths and areas, so that they don't end up getting lost. Moreover, when it's about very big areas, items and collectibles should be abundant.

We can see a great example of that principle put in practice in Super Mario Odyssey, where there are tons of collectibles all over the world and you never feel like there's not much to do. The map never feels empty. Moreover, the game features fast and fun ways to move at different speeds. This allows even huge levels to be traversed in a small amount of time and without feeling stressed or having the impression of wasting time. It's very important, for the player, to be able to move at their own pace.

On the other hand, Yooka-Laylee (Team17, 2017) features very big levels, but even if they are beautiful to watch, they're not very fun to traverse. In fact, Yooka moves very slowly and doesn't have any skill or options to travel faster. This makes the experience feel very slow; and this, combined with a bad collectible positioning and variety, can cause many players to lose interest and motivation.

A famous example of collectibles done wrong can be found in Donkey Kong 64 (DK64; Nintendo, 2010). DK64 features five different collectibles: one for each Kong that you can play with. Every Kong can only pick up its own type of collectible. This means that to collect them all, you are forced to repeat the same level five times. This mechanics is repeated for other kinds of game objects like switches, secret areas, and even boss fights. It's good to give the player variety and allowing for backtracking; but if the level remains aesthetically exactly the same and plays the same and the only thing different is the color of the collectibles, there's something wrong!

Other than that, the positioning of collectibles in DK64 sometimes feels a bit random. It's always a good idea, instead, to place collectibles and items along the right way that the player has to follow. This makes the collection less tedious and also tells the player where they should go.

This takes us to the last topic: level design. How much is important in a platformer? And how should a level be to be enjoyable?

World makers

Level design is a hard task. It's the art of creating credible and recognizable places which are fun to traverse.

In the platformer genres, there are way too many games that feature huge and not memorable levels. The player is forced to explore very big places that totally lack recognizable landmarks or some way to orient yourself while exploring. Some of those games intelligently turn around the problem by implementing a pop-up map or a Minimap in a corner of the screen; some others don't address the problem at all and end up being perceived as dull or frustrating.

A good example of memorable levels filled with landmarks are the levels found in Gex: Deep Cover Gecko (Crystal Dynamics, 1999). They are all built around a theme: there is the pirate level, the one inspired by a Japanese anime, the one set in the ancient Egypt, the Christmas level, and so on. They are all open world levels, but they never feel confusing, both because they're not too big and because they're full of landmarks. The whole game looks like a theme park, where you can go around and play various mini-games interwoven by fun platforming sessions. In fact, every level offers many puzzles and different activities in the form of mini-games that you have to deal with to advance.

A classic example of a game featuring good level design is Super Mario 64 (Nintendo, 1998). In fact, every level has a unique setting filled with recognizable landmarks and, more importantly, a clear purpose. Just take Bob-omb Battlefield as an example; it's built around the boss fight with King Bob-omb that waits for you at the top of a hill. Around this hill, there is the rest of the level that mostly consists in a path that runs from the start of the level to the top of the hill. That can look like a lazy level design, but it's actually perceived as a fun and well-designed level because while you walk that path, you are introduced to a lot of game mechanics that you will need to use in the rest of the game.

The level starts with the classic goombas trying to chase you. You already know them: you jump on their heads, and they're gone. Then, there are the bob-ombs that you can't defeat with a jump, so you try other buttons, and it turns out that you can grab them and throw them away. Going further, you get in contact with coins and question mark blocks that work pretty much like Super Mario Bros. Finally, you find yourself in front of a big bad Chain Chomp,which is a big bomb chained to a wooden pole behaving as a guard dog; and as soon as you get close, it will attack trying to bite Mario. There are two things you can do to overcome this situation: you can try to escape by using a forward jump to quickly surpass it and get to the next area, or you can use the ground-pound move to stick the wooden pole into the ground freeing the Chain Chomp that will eventually run away breaking a jail that contains a collectible. In so little space, Nintendo managed to put a lot of information teaching the player some basics of the game, like how Chain Chomp works and how to solve some environmental puzzles. This is not only just a good learning moment, but it's also a fun memorable moment (and place) in which the player is rewarded with a collectible. While freeing the Chain Chomp, you are not thinking about the fact that you have to get the collectible; you are only thinking about how to get rid of that beast. The collectible is just your reward for solving the puzzle. It's like if the game congratulates with the player telling them that this is how you have to play this game. When level design can communicate so many information so clearly, you understand how powerful even the simplest asset in your game world can be and how big is your responsibility toward the fun factor of the game.

Level design in Super Mario 64 is often built around rewards and environmental puzzles. This makes the level memorable and fun and offers the level designer a good criterion to place collectibles making them an extra reward that doesn't turn the act of collecting into an annoying or repetitive task.

All the information you gathered while climbing the king's hill will be very useful to win the boss fight. In fact, being King Bob-omb a bob-omb himself, you know that you can grab him and throw him away. After some throws, he will surrender and beg you to stop.

Bob-omb Battlefield is a level built not only around a theme, but around a series of concepts that need to be taught to the player. That's why it's so good!

The design lesson to learn, analyzing Bob-omb Battlefield and Gex: Deep Cover Gecko, is that when you design your own levels, you should always try to think of them as real places in the fictional world of your game. Anyway, good and coherent aesthetics alone doesn't make a good game! Other than choosing a theme to give an aesthetic personality to your level, you also need to build the level around one or more gameplay mechanics.

Start creating the main area of the level (e.g., King Bob-omb's hill) and think about what the player needs to know to face that challenge or – more generally – to enjoy that piece of gameplay. Context will naturally grow around it, bringing ideas for decorative elements, paths to follow, puzzles, and platforming challenges.

For example, if you want to introduce the player to a new kind of mechanics – let's say the wall jump – design the piece of level that uses that technique at its best, and then build the rest of the level in function to that special moment of gameplay. As all the learning processes, it starts with a good preparation which can be made by adding some easier and safer wall jumps along the way. Following this process, you will end up with a nice level that teaches a new technique step by step giving the player the time they need to grok the new skill.

At the end of Chapter 9, we built our first level for Cherry Caves 2 by thinking about how to introduce the various gameplay elements we built, and this is a very good way to design games. If you don't have a narrative concept to explain or it's not the primary focus of the game, bet everything on gameplay elements by teaching them to the player one by one and increasing the difficulty step by step.

A sublime example of this level design philosophy can be found in Celeste (Matt Makes Games, 2018), where each and every level builds upon the concepts learned previously.

Celeste teaches you, one level at a time, how to use every skill of the game one by one and how to combine them to accomplish what in the beginning seemed impossible. The game also reinforces this concept of accomplishing hard tasks by mastering the fundamentals with the narrative expedient of the mountain climbing – which you can accomplish by focusing on doing right small simple actions.

All the levels in Celeste are challenges, and they're purely built around gameplay. That makes them not memorable, but extremely fun and challenging. Games like this save a lot on aesthetics and invest even more on gameplay mechanics. In fact, to make the level feel fun and enjoyable, the player must have access to a wide array of skills and abilities to combine one another to solve puzzles and platforming sessions. Momentum, gripping, wall jump, dash – they are all explained one by one in levels that grow more and more challenging and skill-demanding intertwining gameplay concepts together.

Beware, though, because when we talk about games built around pure platforming, it's easy to fall into some bad habits, like repeating ideas or making symmetrical levels or levels that are way too linear. In particular, symmetrical levels are bad because they force you to walk along a path and then repeating it reversed. This can be fun for a level or two, but on the long run, it can become tedious and annoying. Instead, try to design asymmetrical areas where each section of the level focuses on one action, skill, or gameplay moment, not on the shape of the level itself. Symmetry is nice to look at, but it hardly makes a fun level.

Conclusion

Don't worry if this may appear overwhelming or too difficult and don't mind too much if your first level designs are not super good. Level design, like nearly everything in life, is an art that requires time and practice to be mastered. While creating levels, you will constantly learn new things and become better at it. So don't give up and keep on creating games!

In the next chapter, we will go further into the platforming genre by analyzing and implementing a metroidvania game! We will learn how to create even more interesting gameplay elements like wall jump, dash, and a Minimap; and we will address the problem of backtracking in level design, which is one of the defining characteristics of the genre.

CHAPTER 11

Metroidvania (Part 1)

We covered many different genres and sub-genres; and you learned how to create a card game, a fixed shooter, a scrolling shoot 'em up, a single-screen platformer, and a scrolling platformer. Each of these projects taught you an important lesson about game design and development! Let's summarize a bit what we saw in the previous chapters:

- Memory (Chapters 3 and 4) taught you randomization, variables, loops, and data structures.
- Space Gala and Space Gala 2 taught you how to manage shooting, bullets, health, score, enemies, and basic enemies AI and how to create basic boss fights.
- Cherry Caves and Cherry Caves 2 taught you about jumping, ground collisions, gravity, power-ups, menus, and some more about enemies with basic AI. We also started to think about how to design good levels and to foster players' motivation and engagement creating feature-rich games.

In this and the next chapter, we are going to put all this knowledge together to study and implement a project based on one of the most interesting and complex game genres around: metroidvania.

The game will be called *Isolation*. The setting is a labyrinth of tunnels and caves on an alien planet. The player, using a map and some skills like wall jump and dash, will explore those strange places in search for the exit.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_11

The idea behind the gameplay of a metroidvania is pretty simple, but the development is not. In fact, it requires some complex mechanics and features like maps, Minimaps, exploration skills, and also items, equipment, and an inventory menu to manage your items. A metroidvania game also requires a system of checkpoints to save the game and let the player continue the exploration.

History

Metroidvania is a sub-genre of action-adventure video games strongly based on exploration and platforming which often features also complex combat mechanics and RPG elements.

The word metroidvania is a portmanteau of two video game titles: Metroid and Castlevania. In particular, the word refers to Super Metroid (Nintendo, 1994) and Castlevania: Symphony of the Night (Konami, 1997), which are the two fathers of the genre.

The focus of the genre is on the exploration of complex structures, an activity that requires particular skills or items to be completed, thus forcing the player to backtrack. In fact, one peculiarity of metroidvania games is that they allow the player to unlock new areas only by acquiring special items like keys or weapons or by gaining new exploration skills that allow to get to places that were out of reach before.

A classic example of this mechanic can be found in games featuring doors that can be opened only with a particular key or games that feature areas reachable only with particular skills like double jump, wall jump, dash, and so on.

The way the level designer places those key items or the moment in which he chooses to unlock some skills greatly affects the player's perception of the exploration and how much fun and engagement the game will give to the player.

Apart from the classic examples of Super Metroid and Castlevania: Symphony of the Night, some (more modern) very good examples of exploration and backtracking done right are Hollow Knight and Iconoclast.

Hollow Knight does an incredible job in immersing the player in a world with deep lore and wonderful atmosphere dragging them into a huge world of interconnected underground tunnels that tell the story of a great lost kingdom. In HK, backtracking can be felt as a real consequence of finding clues that open new paths in discovering the history of this ancient world.

Iconoclast has a very original and fun way to traverse areas by solving puzzles that make a great use of key objects and skills to unlock paths. It's interesting how classic environmental puzzles make the game so fresh and different from all the other metroidvania games.

In this chapter, we will create the first part of our metroidvania called Isolation. We will implement just a subset of the features of the project; namely, we will create firstly a platforming base for the game (managing gravity, jump, and basic movements) by quickly recreating some features we already saw in Cherry Caves chapters; then we will enhance the platforming adding wall jump and dash. After, we will create the menu system to manage the different states of the game, and finally we will create a couple of rooms and a warping system to travel through them.

Isolation (Game Design Document)

Isolation is a single-player metroidvania game inspired by games like Super Metroid, Castlevania: Symphony of the Night, and Hollow Knight.

The player impersonates Maria, an archaeologist specialized in alien research isolated on an alien planet. Her objective is to escape the labyrinthic underground world in which she's trapped and find her expedition team.

Story and setting

Maria was part of an explorative expedition on an alien world. While she was studying some rocks in the area, something went wrong, the tunnel collapsed, and she was divided from her team. Now she's alone, isolated, trying to find the way back.

Gameplay

The game revolves mainly around exploration and action combat. By using the items found along the way, the player will be able to upgrade Maria's gear and fight the enemies.

Victory condition

As in all the metroidvania games, victory is achieved when the end of the maze is reached or the main enemy is killed.

The version of the game we will develop will be – of course – very short for the standard of a metroidvania. We will only concentrate on creating the game system, and so this time we won't have a victory condition. The game we will create will only be a prototype to show you the main features of the game system.

Controls

Isolation supports both keyboard and gamepads.

Keyboard

Left: Move left.

Right: Move right.

Z: Jump.

X: Dash.

C: Attack.

Esc: Pause/unpause the game.
I: Open the inventory.
Tab: Open the map.
Enter: Confirm (menu).
Gamepad
Left Analog/Direction Pad: Move left.
Right Analog/Direction Pad: Move right.
Face 1 Button: Jump.
Right Shoulder Button: Dash.
Face 3 Button: Attack.
Start: Pause/unpause the game.
Face 4 Button: Open the inventory.
Select: Open the map.
Face 1 Button: Confirm (menu).

Note that the face buttons on a gamepad are the ones that you can find at the right of the pad and are commonly bound to actions. There are two main configurations on modern controllers: one based on Xbox gamepads and one based on PlayStation gamepads. So, depending on which one of the two kinds of gamepads you have, the face buttons map like in Table 11-1.

Table 11-1. *Face buttons*

	Xbox controller	PlayStation controller
Face 1	A	Cross
Face 2	B	Circle
Face 3	X	Square
Face 4	Y	Triangle

Enemies

Isolation will feature one of the enemies from Cherry Caves 2: the green octopus (Figure 11-1) which moves back and forth horizontally.

The focus of this chapter is not on how to code interesting foes – we covered it in Chapters 5, 6, 8, and 9 – but on exploration and platforming features, so this is enough for our purpose.

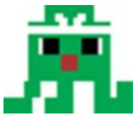


Figure 11-1. *Green Octopus*

Enemies have a passive AI, meaning they won't attack or search for the player, but will just move around being a physical obstacle to the player's exploration.

Attack

Maria has a highly technological pistol, which is the standard equipment of space archeologists.

This special pistol can be upgraded with certain materials that she will find along the way changing the effectiveness of the weapon.

Maria can pick up and equip those upgrades by accessing the inventory system.

Skills

Maria will have two main skills:

- **Dash**, which consists in a quick step forward
- **Wall jump**, which consists in a jump stepping on a wall, allowing her to reach higher places

Maps

Isolation features a map system that will help the player to keep track of their exploring progresses in real time.

The map system is divided in a complete map accessible by pressing the Tab key on a keyboard or Select on a gamepad and a Minimap that is always visible during the game.

While the complete map, as the name suggests, shows the place in its entirety (meaning all the rooms), the Minimap only shows the current room and the adjacent rooms, as shown in Figure 11-2.

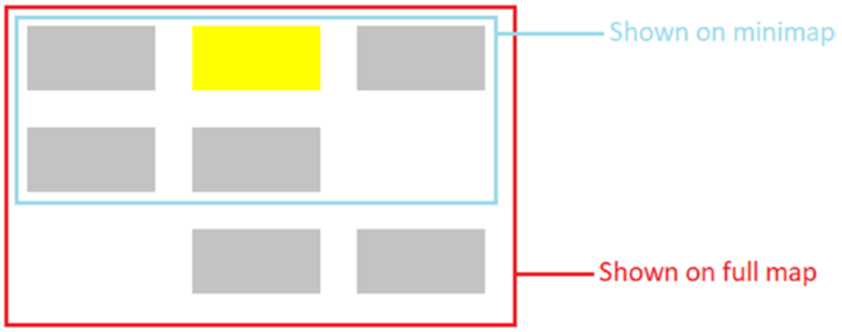


Figure 11-2. *The portion of the area shown on the Minimap and on the full map*

Inventory

Isolation features an inventory system to store and equip all the items that Maria picks up.

We will concentrate more on the technical implementation of the inventory, so the aesthetical aspect will not be central. The inventory will be pretty simple, being just a list of items which updates anytime Maria picks an item.

The player can navigate the list with up and down buttons (both gamepad or keyboard) and equip an item by pressing the confirmation button (Face 1 on gamepad and Z on keyboard).

Similar games

Isolation is based on Super Metroid for both mechanics and atmosphere, but has some inspiration also from Castlevania: Symphony of the Night (concerning the possibility to upgrade the equipment).

Some other similar games are Hollow Knight and Axiom Verge.

Assets

The following is the list of assets we are going to use for this project, plus – as usual – instructions on sprites' characteristics.

spr_player_idle

This represents Maria in idle position (not moving).

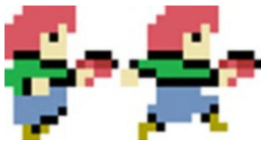
Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_player_walk

This sprite will be used for Maria's walking animation.



Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_player_jump

This will be used for Maria's jump animation (ascending).

Size: 64 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle
spr_player_jump_fall/spr_player_dash

To represent both the fall after a jump and Maria's dashing skill, we will use the same image. You can use a single sprite to do so, but I strongly suggest you to create two different sprites, so that if you want to substitute the image for an action but not the one for the other, you can without touching the code. My code will use two different sprites with the same image.

Size: 64 × 64
Pivot Point: Middle-center
Collision Mask: Automatic, Rectangle
spr_player_wallslide

This sprite represents Maria during a wall slide, which consists in sliding down while touching a wall. It's the starting state to perform a wall jump.

Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_heart

This heart will be used to represent Maria's HP in the HUD.

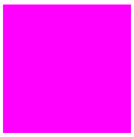
Size: 64 × 64

Pivot Point: Top-left

Collision Mask: Automatic, Rectangle

spr_warp

This sprite will be used to mark the warp zones. We just need its collision box. The sprite won't be visible.



Size: 64 × 64

Pivot Point: Top-left

Collision Mask: Automatic, Rectangle

spr_marker

This sprite will be used to mark the boundaries that the green octopus cannot pass.

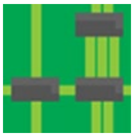
Size: 32 × 32

Pivot Point: Top-left

Collision Mask: Automatic, Rectangle

spr_upgrade

This sprite will be used to represent upgrades for the pistol that can be picked up by Maria and equipped.



Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_cure

This sprite will be used to represent cures which are consumable objects that restore HPs.

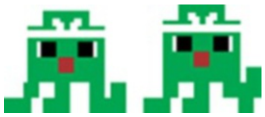
Size: 64 × 64

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_octopus_green

This is the kind of enemy moving back and forth.



Size: 64 × 64

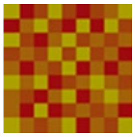
Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

Animation Speed: 4

spr_ground_brown

This sprite will be used to create the ground and wall tiles. Since we are in a cavern, it seems legit to use the same sprite for both.



Size: 64 × 64

Pivot Point: Top-left

Collision Mask: Automatic, Rectangle

spr_checkpoint_inactive

This is the checkpoint platform when not active. The player should walk on it to activate it.

Size: 64 × 64

Pivot Point: Bottom-center

Collision Mask: Automatic, Rectangle

spr_checkpoint_active

This is the checkpoint platform when active (when the player collides with it).

Size: 64 × 64

Pivot Point: Bottom-center

Collision Mask: Automatic, Rectangle

spr_bullet_heavy

This sprite represents a bullet of a heavy weapon.

Size: 16 × 16

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

spr_bullet_light

This sprite will be used to represent a bullet of a light weapon.

Size: 16 × 16

Pivot Point: Middle-center

Collision Mask: Automatic, Rectangle

Fonts

fnt_text

This is the font used to create small standard texts.

Font: Consolas

Style: Regular

Size: 18

fnt_menu_h1

This is the font used for headers in menus.

Font: Consolas

Style: Bold

Size: 32

fnt_menu_h2

This is the font used for smaller headers in menus.

Font: Consolas

Style: Bold

Size: 22

Sounds

snd_item

This is the sound effect we will play when picking up an item. We will use it in Chapter 12.

snd_jump

This is the sound effect we will play when the jump button is pressed.

snd_dash

This is the sound effect we will play when the character's player dashes.

snd_shoot

This sound effect will be played when shooting with the gun. We will use it in Chapter 12.

snd_kill

This sound effect will be played when Maria kills an enemy. We will use it in Chapter 12.

snd_hit

This is the sound effect we will play when Maria gets hit. We will use it in Chapter 12.

snd_menu

This sound effect will be played when navigating through the menu.

Creating the platforming base

Metroidvania, as we already said, is a sub-genre of action-adventure platformers. This means that to create one, we must first build a platforming base upon which we can add features.

Since we already created a platformer in Chapters 8 and 9, I am not going to explain in detail what we are doing. Don't worry, I will show you the code and explain what it does, as always, but I'm going to do it in less detail.

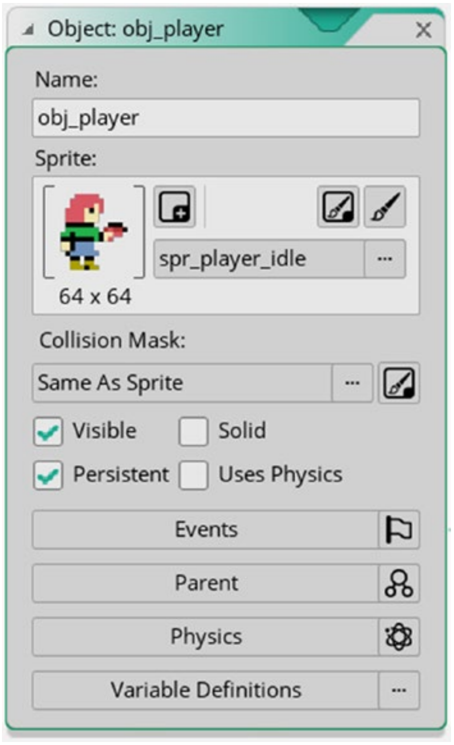
The platforming base of a metroidvania is an interesting one, and it's made mixing the concepts of the two genres we already talked about: single-screen platformers and scrolling platformers.

In fact, since metroidvania games are made of big areas, it's not reasonable to load such big levels into the memory all at once; so they use the trick of combining SSP with scrolling platformers. Basically, areas are divided into sections, so the player walks as a scrolling platformer in one section at a time, allowing the game to load just a little part of the area at a time and still giving the player the idea that they're traversing a very big place.

This is a technique that first came out with Pitfall! (Activision, 1982) which tried to evade the impossibility to build a scrolling platformer (because of technological impediments) creating the illusion of horizontal travelling connecting various rooms to one another. So when the player walked to the right border of the screen, a new section of the area was drawn; and the player teleported to the left border of the screen, giving the illusion of progression.

Let's start working on our new project by creating a new object named `obj_player`.

Associate the new object to spr_player_idle and tick the Persistent option as shown in Figure 11-3.



Object: obj_player

Name: obj_player

Sprite: spr_player_idle

64 x 64

Collision Mask: Same As Sprite

[✓ Visible] Solid

[✓ Persistent] Uses Physics

Figure 11-3.

Every instance, in GameMaker, is created when you enter a room and destroyed when you quit it.

Warning! When you change room, all the instances are destroyed, but their destroy event is not triggered! Use the Room End event (Other Events ➤ Room End) to perform actions when you leave the current room.

Since our game will feature several rooms, we don't want to lose all the information we have on our character, like HPs, items, and so on. So, to prevent this, we make obj_player a persistent object, meaning that it will not be destroyed when changing rooms, but will be preserved until the game is closed or the instance_destroy() function is called.

Add a Create event to obj_player and set some variables:

```
1   // stats
2   spd = 6;
3   hsp = 0;
4   vsp = 0;
5
6   // moving and jumping
7   direction = 0;
8   facing_dir = 1;
```

Line 2: As usual, this is the speed at which the player can move.

Line 5: Here we assign the starting value of the built-in variable direction. We are setting it now for convenience, but we will use it in the next chapter to set the direction of the bullets shot.

The direction variable is set to 0 at the start to be aligned to the starting position of the player (facing right). It's a recognized tradition to start platformers always facing right, because it gives the sense of progression and suggests the player to move forward; but if you plan to make your player start facing left, change this value to 180. If you feel like you're using magic numbers, you can set the value using the point_direction function as we did in Space Gala.

Line 6: This is a variable we are going to use to keep track of the facing direction of the player, since the format of direction is not very comfortable to do some quick math. This variable will be set to 1 when facing right and to -1 when facing left.

Now, create a Step event, and let's start coding the controls of our avatar:

```
// --- CONTROLS HANDLING --- //
var move_left = keyboard_check ( vk_left );
var move_right = keyboard_check ( vk_right );

```

```
5   // set move and speed variables
6   var move = move_right - move_left;
7   hsp = move * spd;
8
9   if move != 0
10  {
11      sprite_index = spr_player_walk;
12      facing_dir = move;
13      image_xscale = facing_dir;
14      direction = point_direction(x, y, x + move, y);
15  }
16  else
17  {
18      sprite_index = spr_player_idle;
19  }
20
21  // horizontal movement
22  x += hsp;
```

Lines 2–3: We use two Boolean variables to keep track of the user's input.

Lines 6–7: We set up the horizontal speed according to the moving direction defined by the user's input.

Lines 9–15: If the player is moving (move is either 1 or -1), we set the direction and facing_dir variables accordingly, change the sprite to the walk sprite, and scale it to face the right direction.

Line 18: If the player is not moving, we change the sprite to the idle sprite.

Line 22: Finally, we set x to the new value defined by the horizontal movement represented by hsp.

You can run the game and verify that the character is correctly moving when pressing the arrow keys.

Gamepad support!

Keyboard controls are not enough! For our games to be actually enjoyable, it's time we start supporting gamepads. GameMaker Studio 2 allows us to do it very easily thanks to some specialized functions. Let's explore them!

Note GameMaker Studio 2 supports both XInput and DirectInput gamepads, but we are going to cover only XInput which is the de facto standard. Xbox pads and PlayStation pads are both XInput based.

GMS2 supports up to four XInput gamepads. The gamepads are indexed from 0 to 3, and you can detect when they are plugged in or out by listening to an asynchronous system event.

Since Isolation is a single-player game that supports both keyboard and gamepads, we have no interest in this. We will always use the gamepad indexed 0.

GMS2 offers a list of functions to handle buttons and analog sticks very similarly to how mouse and keyboards are handled.

For example, to check the status of a button on the gamepad, you can use one of these functions:

- gamepad_button_check(gamepad_id, button_code)
- gamepad_button_check_pressed(gamepad_id, button_code)
- gamepad_button_check_released(gamepad_id, button_code)

As you can see, those functions are very similar to keyboard-related functions. This will make everything easier!

In the functions presented earlier, the gamepad_id is the index of the connected gamepad (from 0 to 3, as we already said), while the button_code is a unique value which identifies a button on the pad.

Figure 11-4 shows a map of the codes of each button and analog that composes a gamepad.

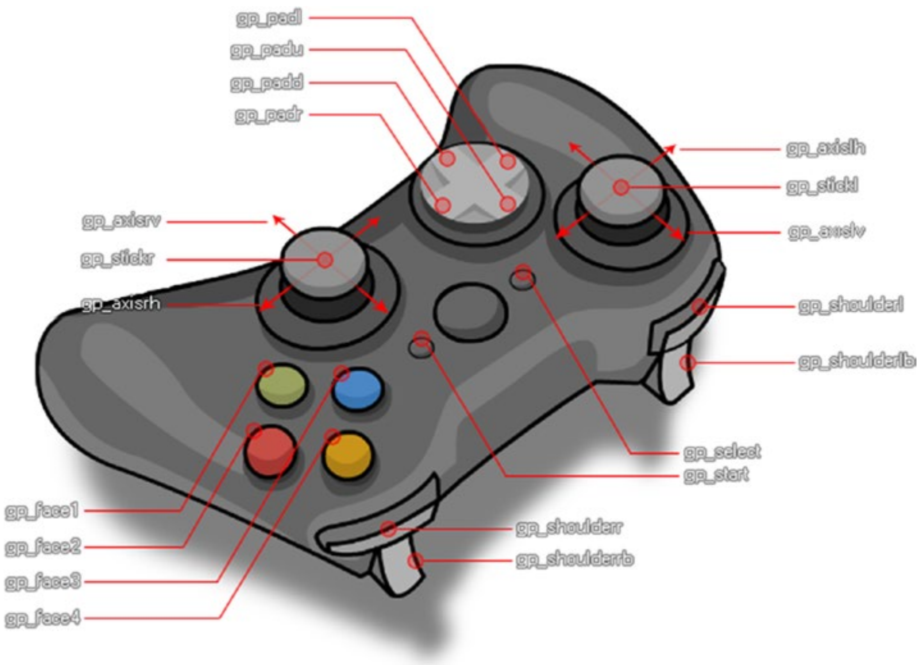


Figure 11-4. *A visual scheme of gamepad mapping on GameMaker Studio 2*

To add gamepad support to our game, we just need firstly to initialize the gamepad analog deadzone. It's important to set the deadzone of an analog stick, if you don't want your game to react to every micro-movement of the analog. Having an excessively sensible analog in a platformer can be troublesome since it could cause the avatar to move when the player doesn't intend to.

We can do this in obj_player's Create event with this single line:

```
gamepad_set_axis_deadzone(0, 0.5);
```

Next, we have to handle the actual input, and we can do it modifying the initial part of the code in obj_player's Step event:

```
1   // --- CONTROLS HANDLING --- //
2   var haxis = gamepad_axis_value(0, gp_axislh);
3
4   var move_left = keyboard_check ( vk_left ) or gamepad_button_check(0, gp_padl) or (haxis < 0);
5   var move_right = keyboard_check ( vk_right ) or gamepad_button_check(0, gp_padr) or (haxis > 0);
```

Line 2: Here we set the haxis variable which keeps track of the position in which we move the analog stick on the horizontal axis (either left or right). To do so, we pass to the gamepad_axis_value function, the code gp_axislh which tells the function to return the position of the left analog on the horizontal axis.

Lines 4–5: Now we set move_left and move_right to true not only according to keyboard input but also checking if D-pad (direction pad) buttons were pressed or if the left analog was used.

Now save and run the game to check that inputs are correctly working!

Take your time to taste the emotion of playing your game with a gamepad. It's a special feeling, isn't it?

Gravity, no escaping!

We definitely need some gravity to our game world, to build an actual platformer. We know all the steps to make thanks to Chapters 8 and 9, so let's do this quickly.

First, let's add a couple of variables to obj_player's Create event:

```
1  grv = 0.8;
2  grounded = false;
```

Line 1: This variable represents the gravity to which our character is affected.

Line 2: This variable tells us if the object is touching the ground or not.

Now we should apply the gravity on our player's movements. Just append this code to obj_player's Step event:

```
1  // apply gravity
2  vsp = vsp + grv;
3
4  // vertical movement
5  y += vsp;
```

At line 2 we apply gravity to the vertical speed of obj_player, while at line 5 we commit the changes to be reflected on the actual position of the object on the Y-axis.

Next step is to create an object that can actually stop obj_player fall when colliding and program obj_player to manage this collision.

Go ahead and create a new object called obj_block and add no sprite to it. Then create another object and call it obj_ground_brown and associate it with spr_ground_brown. Add obj_block as a parent for obj_ground_brown. This is crucial to make obj_ground_brown an effective blocking object for obj_player.

Now let's go back to obj_player's Step event and add this code just before the line that updates the x variable:

```
if ( place_meeting ( x + hsp, y, obj_block ) )
{
    while ( not place_meeting ( x + sign(hsp), y,
    obj_block ) )
    {
        x += sign(hsp);
    }
    hsp = 0;
}
```

The preceding code is borrowed from Cherry Caves and manages the horizontal collision with obj_block instances. I am not going to explain this in detail, since we already saw this code in previous projects. If you need some more explanation, go back to Chapter 8 and find out the section where we first introduced this way of handling collisions with blocks.

Do the same for the vertical collisions and insert this code just before you update the y variable:

```
if ( place_meeting(x, y + vsp, obj_block ) )
{
    while ( not place_meeting ( x, y + sign(vsp), obj_block ) )
    {
        y += sign(vsp);
    }
    vsp = 0;
    grounded = true;
}
else
{
    grounded = false;
}
```

The preceding code handles vertical collisions with obj_block instances and sets the grounded variable to true or false accordingly.

You can now position some instances of obj_block_brown and an instance of obj_player in a room and check that everything is working by running the game.

Making the leap

We have the majority of the platforming system working. We just need to add the jump, so that we can move properly between platforms.

As we saw in Chapters 8 and 9, we need just three variables and a handful of lines of code to accomplish that.

Head to obj_player's Create event and add a new variable declaration at the bottom of the code:

```
jspd = 18;
```

We will use this variable to apply a force that can contrast the gravity and make our avatar jump.

Open up obj_player's Step event and, in the controls handling section, just below the initialization of move_left and move_right variables, add these two lines:

```
var jumping = keyboard_check_pressed ( ord("Z") ) or
gamepad_button_check_pressed(0, gp_face1);
var jump_released = keyboard_check_released( ord("Z") ) or
gamepad_button_check_released(0, gp_face1);
```

Those two lines check whether the jump button has been pressed or released. We will use this information in the next chunk of code to manage the jumping and stop the raising of our avatar when the button is released. This will give us total control on the character's jump allowing us for creation of fun and challenging platforming sessions in our levels.

Now, just above the section that manages collisions with obj_block instances, insert this code to manage jumping:

```
// JUMP
if jumping and grounded
{
    grounded = false;
    vsp = -jspd;
```

```

        sprite_index = spr_player_jump;
        audio_play_sound(snd_jump, 1, false);
    }

    if jump_released
    {
        vsp *= 0.5;
    }
```

The preceding code is also borrowed from Cherry Caves, and it's very straightforward. When the character is grounded and the jump button is pressed, a negative force is applied to its vertical speed. Being the force greater than the gravity applied to the vertical speed, the character will move upward, jumping. While jumping, the grounded variable turns to false, so that the player cannot jump anymore until they reach again the ground.

When the jump button is released, the jumping force gets halved so that the raising stops quickly and the character begins its fall as soon as the vsp variable reaches 0.

Save and run the game to check that everything is in place (Figure 11-5). The character should be able to move and jump on platform correctly with both keyboard and gamepad. Great!

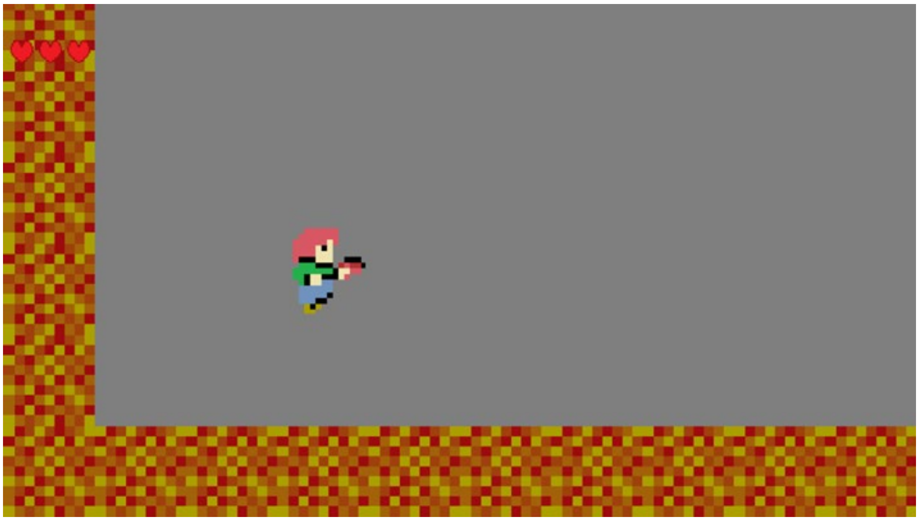


Figure 11-5. *Finally, Maria can happily jump!*

Another kick in the wall

A very loved feature of metroidvania games and action-adventure platformers in general is wall jump (or wall kick). Widely known thanks to the Mega Man series, wall jump allows the player to use walls to gain an additional jump starting from the moment the player's character touches the wall.

This skill is widely used in metroidvania and action-adventure platformers, because it allows the player to have much more freedom and to reach places that were unreachable before by bouncing between walls or directly climbing it with some chained wall jumps.

Wall jump is often connected to another skill: wall slide. This one is a very interesting perk that allows the player's character to grab a wall and gently slide downward. This is a huge help to avoid the leap of faith problem (meaning that the player has to make some jumps without knowing what's below) in level design without making obvious what lies on the bottom of an area.

Some games known as rage games do use level design patterns like leaps of faith to trick the player into taking the wrong decision and die. I personally think that this is bad because it makes the player feel like the whole experience is unfair. Most of the people don't like to play unfair games that don't give you the possibility to control your performance, and this is one of the feelings that breaks the magic of the cognitive flow status.

With wall slide, the player's character will slow down the fall by sliding on the wall allowing them to see what's below and eventually jump to safety with a wall jump.

We are going to implement wall kick by dividing the effort in two phases:

1. We check if the player's character is touching a wall while not grounded. If they are, we check whether the player has pressed the jump button; and, in this case, we set the coordinates at which the character should be rebound horizontally and perform a jump in the usual way.

2. When phase 1 executes, we start moving the player's character toward the rebound goal we set in phase 1. When the coordinates are reached or the avatar touches the ground, we stop moving it and end phase 2.

Let's start by defining the variables we will need. Go ahead and open up the Create event in obj_player and append this code to it:

```
// wall jump/slide
can_wall_jump = true;
wall_jump = false;
wall_slide_friction = 0.5;
wj_goal_x = x;
already_walljumping = false;
```

Let's explain the meaning of those variables one by one:

- can_wall_jump is a controller variable that we will use to check whether the player has unlocked the wall jump skill or not.
- wall_jump tells us if the player's character can wall jump or not. This will be set to true when the player's character touches the wall and to false when they move from it.
- wall_slide_friction is used to store the value of the friction that we apply to the fall of the player's character when touching a wall (wall sliding). You can change this accordingly to your preference.
- wj_goal_x is the coordinate on the X-axis that the avatar should reach after performing a wall jump.
- already_walljumping is the trigger to start phase 2, as we described earlier.

Now, in obj_player's Step event, we want to add the code that actually tells us when the player is not grounded and is touching the wall. To do so, we need to check for the collision with an obj_block instance just a pixel next to the player's character, in the direction they're facing.

So add this code in obj_player's Step event, just above the jump code:

```
// --- WALLJUMP / WALLSLIDE --- //

// Check if touching a wall -> activate wall slide / jump
if not grounded and can_wall_jump and place_meeting ( x +
facing_dir, y, obj_block )
{
    sprite_index = spr_player_wallslide;
    wall_jump = true;
8        if vsp > 0
9        {
10           vsp -= wall_slide_friction;
11       }
12   }
13   else
14   {
15       wall_jump = false;
16   }
```

In the preceding code, we check whether the player's character is not grounded, can perform a wall jump, and is touching the wall using the facing_dir variable to make sure to check the collision in the direction the player's character is facing (line 4). If the condition at line 4 is not satisfied, the player cannot wall jump, so we set the wall_jump variable to false. However, if the condition at line 4 is satisfied, we change the sprite to the appropriate wall slide sprite (line 8) and set the wall_jump variable to true (line 9), so that the player can wall jump; and finally, we check whether we are descending or not, and we apply the right friction to the fall (lines 10–13). In fact, we don't want the avatar to slow down also when jumping; we want them to slow down only when descending while touching the wall. To do that, we have to check whether the value of vsp is positive (line 10), meaning that we are moving downward (remember that the Y-axis has the 0 on the top and moving down means adding a positive value to the Y-coordinate).

Now, to implement the last part of the wall jump, we need to modify the jump code. Go ahead and substitute the block of code that manages the jump in obj_player's Step event with this code:

```
1    // JUMP
2    if jumping and (grounded or wall_jump)
3    {
4        grounded = false;
5        vsp = -jspd;
6
7           // WALL JUMP - PHASE 1
8           if wall_jump
9           {
10              effect_create_below(ef_smoke, x, y, 1, c_white);
11              facing_dir *= -1;
12              image_xscale = facing_dir;
13              direction = point_direction(x, y, x + facing_dir, y);
14              wj_goal_x = x + 80 * facing_dir;
15              already_walljumping = true;
16              wall_jump = false;
17          }
18
19          sprite_index = spr_player_jump;
20          audio_play_sound(snd_jump, 1, false);
21      }
22
23      if jump_released
24      {
25          vsp *= 0.5;
26      }
27
28      if not grounded and not wall_jump and vsp > 1
29      {
30          sprite_index = spr_player_jump_fall;
31      }
```

Line 2: We added to the trigger condition a check upon the value of wall_jump that we potentially set to true in the previous block of code (the one handling the collision with obj_block).

Line 5: We perform the jump. This line of code is executed both when the player jumps grounded and when they perform a wall kick.

Lines 8–17: If the value of wall_jump is set to true, meaning that we are performing a wall jump and not a simple grounded jump, we invert the direction faced by the avatar, to simulate the propulsion in the opposite direction given by the wall kick. Then we set the position on the X-axis to which we want our avatar to move (line 22), we set the trigger variable already_jumping to true (line 23), and finally we switch off the wall_jump variable (line 24). Phase 1 done!

Lines 19–20: Pretty self-explanatory! We assign the jump sprite to obj_player and play a jump sound.

Lines 23–26: We check if the jump button has been released. If that's the case, we stop the ascension.

Lines 28–31: We have a little icing on the cake. We check if the avatar is descending and it's not wall sliding. If that's the case, we change the sprite to spr_player_jump_fall. This gives the avatar some more personality and credibility and makes the game feel a bit more visually pleasing.

Ok, it's time for phase 2! Ready?

As we said, phase 2 is going to constantly move the avatar toward a goal position calculated in phase 1. We also borrow the code to check if we are going to hit a wall while moving. It's actually highly probable that we will hit a wall after a wall jump, since to bounce between walls is its primary purpose.

So, cutting the chitchat, let's add some code below that we just wrote in the Step event of obj_player:

```
// WALL JUMP - PHASE 2
if already_walljumping
{
     wj_move = spd * facing_dir;
          if ( place_meeting ( x + wj_move, y, obj_block ) )
6              {
7                    while ( !place_meeting ( x + sign(wj_
                     move), y, obj_block ) )
8                    {
9                          x += sign(wj_move);
10                   }
11                   wj_move = 0;
12                   already_walljumping = false;
13             }
14             x += wj_move;
15             already_walljumping = already_walljumping and
               ((facing_dir > 0 and wj_goal_x > x) or (facing_
               dir < 0 and wj_goal_x < x));
16      }
17
```

Line 4: We calculate the position on the X-axis that we want the player's character to move to.

Lines 5–13: That's the code we borrowed to check if we are going to hit a wall after the wall kick. As you can see, if that's the case, after we approached the wall, we set wj_move to 0 and already_walljumping to false so that this piece of code won't be executed again.

Line 14: Here we update the X-coordinate using wj_move. This happens constantly until already_walljumping turns to false.

Line 15: After every iteration, we update the value of already_walljumping. If the avatar reached the goal or already_walljumping has been set to false (line 12), this means that we don't need to move the avatar anymore and that we can set the variable to false (or leave it to false).

Done! It was a bit tricky, but we did it!

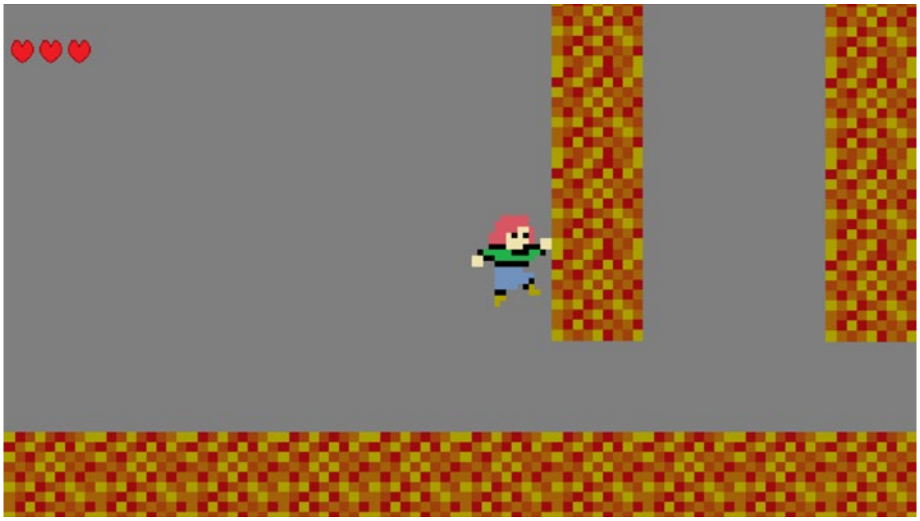


Figure 11-6. *Maria can now wall slide and jump like a modern pink-haired ninja!*

Save and run the game and enjoy your new shiny wall kick skill (Figure 11-6)! This is a great addition to any platformer that aims at exploration and complex platforming sessions!

Now let's talk about another super-important feature that's a must-have for any good metroidvania game: dash!

Moving forward with a dash

Dash is another skill made great by the Mega Man series. It's a feature that firstly appeared in Mega Man X, but there was a similar thing since Mega Man 3: sliding. The only difference between the two skills is that a dash can be performed in midair as well as on the ground, while a slide is only possible when on the ground. Also, a dash is exclusively used to navigate through the level quickly, while the sliding is used mostly to traverse low passages or to avoid some attacks, other than moving quickly.

Since sliding seems to have been abandoned after Crash Bandicoot and dashing has a huge following that makes it a must-have feature in nearly every single metroidvania game, we are going to concentrate only on the latter.

The dashing logic is very similar to the one we used to implement the wall jump. We basically want the player's character to be able to perform a dash every time they are grounded. This means that if they jump, they can perform just a single dash, before they touch the ground again. Touching a wall, even while wall sliding, will grant the player a dash.

The dashing movement is a fast progression forward to cover a certain distance that must be always the same. May the distance covered by a dash change, the player wouldn't be able to use it to precisely move in the level, making it a useless skill, because complex and engaging platforming always puts precision before speed. Well, except for Sonic!

To accomplish this, we divide the handling of dashing in two phases, as we did for wall jump:

1. When the player presses the dash key, we save the current position of the avatar, set the goal – which is the position at which we want to move the avatar with the dash – and set to true a Boolean variable named already_dashing that starts the second phase.

2. Phase 2 is activated when the already_dashing variable is true. In this phase, we borrow the code to manage pixel-perfect collisions with obj_block instances and use it to check whether we are going to hit a wall, just as we did for wall jump. We also want to suspend gravity and every force that can move the player's character vertically. We want the dash to be a pure horizontal quick movement, so we leave the avatar floating for a split second and then restore gravity. To do so, we will use an alarm.

All set? Ok, let's start from the beginning opening up once again obj_player's Create event and declaring those variables:

```
// dash
can_dash = true;
already_dashing = false;
dash_speed = 25;
dash_power = 200;
dash_move = 0;
```

Let's see what those variables do:

- can_dash, just as can_wall_jump, tells us if the player is allowed to use that skill or not. You may want to deactivate it temporarily for gameplay purposes.
- already_dashing is the controller variable we will use to access phase 2.
- dash_speed is the variable that represents the speed at which we are moving while dashing.
- dash_power is the distance in pixel that we can cover with a single dash.
- dash_move is a variable we will use to move horizontally the avatar according to the dash.

Now open up obj_player's Step event and add this line just below the definition of the jump_released variable, in the controls handling section:

```
var dashing = can_dash and (keyboard_check_pressed( ord("X") )
or gamepad_button_check_pressed(0, gp_shoulderrb));
```

Now position the cursor just below the block of code that handles the wall jump.

Append the following code which implements the first phase of dashing:

```
1   // DASH - PHASE 1
2   if dashing and not (already_dashing or dash_recharging)
3   {
4       dash_goal = x + dash_power * facing_dir;
5       already_dashing = true;
6
7       effect_create_below(ef_smoke, x, y, 1, c_white);
8       audio_play_sound(snd_dash, 1, false);
9       sprite_index = spr_player_dash;
10  }
```

In the preceding code, as we said, we check if the player wants to perform a dash (meaning they pressed the dash key); and if they did, we set the dash goal (line 4), set the controller variable to true at line 5 (so that we can access phase 2), create a nice particle effect of a dust cloud to transmit the sense of speed (line 7), and play a sound effect (line 8). Finally, we change the active sprite to the dashing sprite (line 9).

Phase 2 is a little bit longer, but it's very similar to wall jump's phase 2, so you shouldn't have problems understanding what's happening:

```
1   // DASH - PHASE 2
2   if already_dashing and not dash_recharging
3   {
4       // floating after dashing
5       vsp = 0;
6       jspd = 0;
7       grv = 0;
8
9       dash_move = dash_speed * facing_dir;
10      if ( place_meeting ( x+dash_move, y, obj_block ) )
11      {
12          while ( not place_meeting ( x+sign(dash_move), y,
            obj_block ) )
13          {
14              x += sign(dash_move);
15          }
16          dash_move = 0;
17          already_dashing = false;
18          dash_recharging = true;
19      }
20      x += dash_move;
21
22      already_dashing = already_dashing and ((facing_dir >
        0 and dash_goal > x) or (facing_dir < 0 and dash_goal
        < x));
23
24      dash_recharging = not already_dashing;
25
26      if not already_dashing
27      {
28          alarm[0] = room_speed * 0.2; // stop floating in
            0.2 secs
29      }
30  }
```

Line 2: As we already said, we check if we are allowed to dash and the dash skill is not recharging (e.g., we already have performed a dash in midair and touched no wall nor the ground).

Lines 5–7: We reset any force that moves our player vertically, since, while dashing, we want them to move exclusively horizontally.

Line 9: We calculate how much the avatar should move in this iteration by using dash_speed.

Lines 10–19: We check if we are going to hit a wall while dashing or not. If that's the case, we stop dashing by resetting dash_move and already_dashing, and we set to true the dash_recharging variable – which means that to perform another dash, we have to touch the ground.

Line 20: We finally move the avatar of dash_move pixels.

Line 22: We set the controller variable already_dashing accordingly, just like we did with wall jump's phase 2.

Line 24: Here we set dash_recharging to true if we finished dashing – namely, already_dashing was set to false.

Lines 26–29: If we finished dashing, we start the alarm setting it to 0.2 seconds. That alarm just sets gravity and all the variables related to vertical movement to their original value, so that we can fall down 0.2 seconds after the dash.

Now let's create the actual alarm. Add an Alarm 0 event to obj_player and just add these three lines in it:

```
// Restore gravity and jump speed after dashing
grv = 0.8;
jspd = 18;
```

One last thing! Pick the piece of code related to vertical collisions with blocks and change it like this:

```
// VERTICAL COLLISION WITH BLOCKS

if ( place_meeting(x, y + vsp, obj_block ) )
{
    while ( not place_meeting ( x, y+sign(vsp), obj_block ) )
6           {
7               y += sign(vsp);
8           }
9           vsp = 0;
10          grounded = true;
11          dash_recharging = false;
12      }
13      else
14      {
15          grounded = false;
16      }
17
18      y += vsp;
```

We just added a single line (line 11) in which we set dash_recharging to false, when the avatar touches the floor. This line is crucial to make the dash work properly.

Ok, you can now save, dry your sweat, and run the game to check that everything works good!

You should be able to dash your way around and wall jump like a ninja (Figure 11-7)! That's great! We just created a super-fun platforming system that gives us a huge amount of game design possibilities! We're half the way!

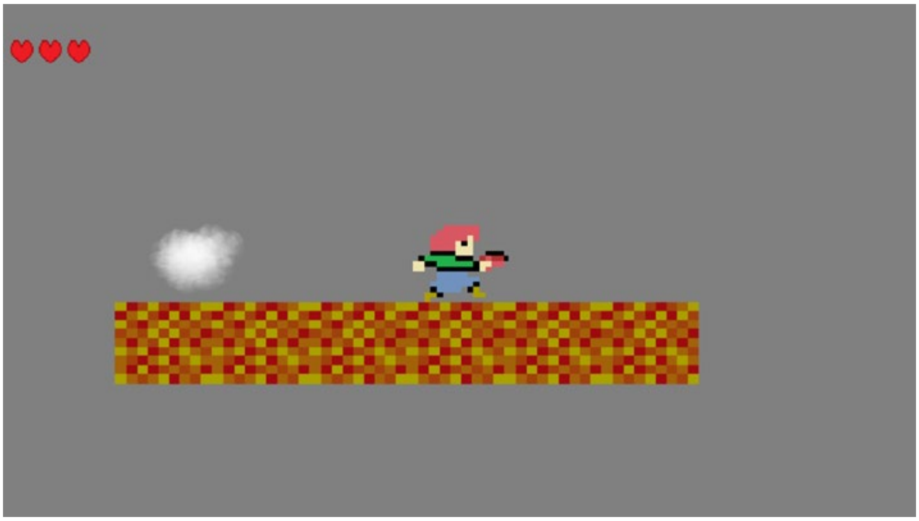


Figure 11-7. *The dash skill, combined with wall jump and wall sliding, adds a plethora of game design possibilities and greatly enhances the fun! Even Maria looks amused!*

The game flow

I know! I know! This is the most boring part! But we need a system on which we can base our code for the map, so…let's start creating a game state system!

We saw this in all our projects, and it's probably the most powerful piece of code we are writing. It's boring just because it never changes a bit – which is also its greatness: a universal piece of code that's good for every situation!

Defining the game flow, we want to regulate the heartbeat of our game by creating statuses that tell us what we are supposed to do and display.

In Isolation, we go back to the three-state structure – just like in Space Gala – since we don't need the lives mechanic that we introduced in Cherry Caves which forced us to create an additional state (states.dead).

In Isolation, we don't want the mechanic we used in previous games where getting a hit meant to die. Maria is a tough girl and can manage to take some hits without giving up. We will introduce the concept of health. Maria will lose health for every hit she takes; and when she dies, she dies for good, possibly respawning at the last checkpoint. So we just need a game over state.

Concerning the states.paused state, we are going to use it for the pause menu, the full map of the area, and the inventory.

The playing state is still the state that we are going to use to actually play the game. Figure 11-8 shows Isolation's game flow as a FSM.

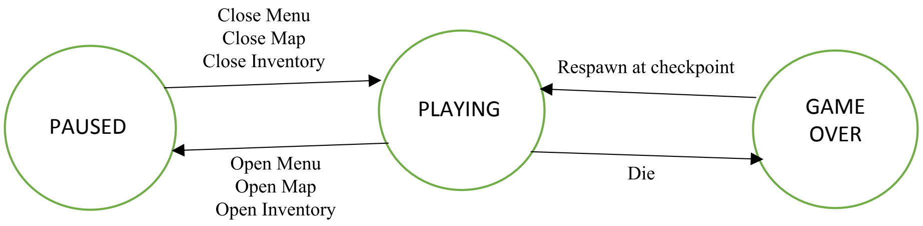


Figure 11-8. *Isolation's game flow*

Ok, let's get started! We will implement the game states shown in Figure 11-8 by reusing some code from the previous projects.

Create a new object called obj_controller, make it persistent, and add a Create event containing this code:

```
// GAME STATES
enum states {
    playing,
    paused,
    gameover
};

global.game_state = states.playing;

```

```
10  // MENU
11  options = [ "RESUME", "RESTART", "QUIT" ];
12  opt_number = array_length_1d(options);
13  menu_index = 0;
14  cur_moved = false;
15  menu_open = false;
16
17  // resolution
18  var width = 1280;
19  var height = 720;
20  display_set_gui_size(width, height);
```

Lines 1–5: Here we define the enum structure containing the various states of our game.

Line 8: We set the initial state to playing.

Lines 10–15: This is our usual code to manage the options in the pause menu. Nothing really new here.

Lines 18–20: Like in previous projects, we are forcing the resolution for compatibility.

Now let's create a Step event for obj_controller and write this code to manage the state switch:

```
1   // --- CAPTURE CONTROLS --- //
2   var vaxis = gamepad_axis_value(0, gp_axislv);
3
4   var esc_pressed = keyboard_check_pressed(vk_escape) or
    gamepad_button_check_pressed(0, gp_start);
5   var enter_pressed = keyboard_check_pressed(vk_enter) or
    gamepad_button_check_pressed(0, gp_face1);
6   var move = (keyboard_check_pressed ( vk_down ) or
    gamepad_button_check_pressed(0, gp_padd) or (vaxis > 0)) -
    (keyboard_check_pressed(vk_up) or gamepad_button_check_
    pressed(0, gp_padu) or (vaxis < 0)) ;
```

```
var f_pressed = keyboard_check_pressed(ord("F")) or
keyboard_check_pressed(vk_f12);

// Cursor move
if move != 0
{
    audio_play_sound(snd_menu, 1, false);
    if cur_moved
    {
        move = 0;
    }
    cur_moved = true;
}
else
{
    cur_moved = false;
}

// Fullscreen setting
if (f_pressed)
{
    window_set_fullscreen(not window_get_fullscreen());
}

// --- MENU --- //
if esc_pressed
{
    audio_play_sound(snd_menu, 1, false);
    if ( global.game_state == states.playing )
    {
        global.game_state = states.paused;
        menu_open = true;
    }
    else if ( global.game_state == states.paused )
    {
        global.game_state = states.playing;
        menu_open = false;
    }
}

if ( menu_open )
{
    menu_index += move;

    if ( move != 0 )
    {
        audio_play_sound(snd_menu, 1, false);
    }

    if ( menu_index < 0 )
    {
        menu_index = opt_number - 1;
    }
    else if ( menu_index > opt_number - 1 )
    {
        menu_index = 0;
    }

    if ( enter_pressed )
    {
        switch( menu_index )
        {
```

```
68              case 0:
69                  global.game_state = states.playing;
70                  instance_activate_all();
71                  break;
72              case 1:
73                  game_restart();
74                  break;
75              case 2:
76                  game_end();
77                  break;
78          }
79      }
80  }
81
82  // -- GAME OVER --- //
83  if ( global.game_state == states.gameover )
84  {
85      instance_deactivate_all(1);
86      if ( enter_pressed )
87      {
88          game_restart();
89      }
90  }
```

We already saw all this many times in our previous projects, so let's not spend too much time on it and let's comment it quickly.

Lines 2–7: Those are the variables storing the information on user inputs. They are used to open and close the menu, to move the cursor through the various options, and to select the desired option.

Lines 10–22: It's a piece of code that regulates the input navigating through the options of the menu. When the cursor is moved the first time, cur_moved is set to true, so that in the next frame, cur_moved is true and move gets set to 0, so it won't move. This goes on until the player releases the button/key. When they do, cur_moved is reset to false, and the cursor can be moved again.

Lines 25–28: This code allows the player to put the game to fullscreen or windowed mode.

Lines 31–44: This is the code that manages the opening and closure of the pause menu and uses the states to understand whether the menu was opened or not.

Lines 46–80: This code manages the menu and the navigation between the available options and their selection (lines 64–79). We already saw it in detail in the previous chapters. If you feel uncertain about what this code does, feel free to pause this chapter to go back and check that piece of code.

Lines 83–90: Finally, this is the piece of code that handles the game over status. When the game is over, there's no choice, you press enter, and the game restarts.

Ok, now we only have to deal with the draw event to actually see the menu.

Let's create a Draw GUI event for obj_controller and add this code in it:

```
// --- SCREEN SETTINGS --- //
var cam_w = display_get_gui_width();
var cam_h = display_get_gui_height();

// --- DRAW LIFE --- //
if instance_exists(obj_player)
{
    for( var i = 0; i < obj_player.hp; i++)
    {
        draw_sprite_ext(spr_heart, -1, 10 + (40 * i), 50,
        2, 2, 0, c_white, 1);
    }
```

```
}

// --- DRAW MENU --- //
if ( menu_open )
{
    draw_set_alpha(0.5);
    draw_set_color(c_black);
    draw_rectangle(0, 0, cam_w, cam_h, 0);

    draw_set_alpha(1);
    draw_set_color(c_white);
    draw_set_font(fnt_text);
    draw_text(cam_w/2, cam_h/2, "PAUSE");

    for( var i = 0; i < opt_number; i++ )
    {
        if ( menu_index == i )
        {
            draw_set_color(c_red);
        }
        else
        {
            draw_set_color(c_white);
        }
        draw_text(cam_w-200, cam_h-200 + 30 * i, options[i] );
    }
}

// --- DRAW GAME OVER SCREEN --- //
if ( global.game_state == states.gameover )
{
```

```
43      draw_set_color(c_black);
44      draw_rectangle(0, 0, cam_w, cam_h, 0);
45
46      draw_set_color(c_white);
47      draw_set_font(fnt_text);
48
49      draw_text(cam_w/2, cam_h/2, "GAME OVER");
50  }
```

As we did with the step event, let’s analyze quickly also this code block.

Lines 2–3: We capture the size of the screen.

Lines 6–12: This code draws on the top-left corner as many hearts as HPs the player’s character has.

Lines 15–38: This code draws the menu with a semi-transparent black cover and the three options, plus the write PAUSE in the middle of the screen.

Lines 41–57: Draws the write GAME OVER in the middle of the screen, when the player loses.

Ok, now nearly everything is done. We just need to add a single line at the top of obj_player’s Step event:

```
if global.game_state == states.paused exit;
```

This line makes sure that if the game is in pause, nothing about our avatar changes, and it will remain frozen where it stands.

Ok, now you can save and run the game and check that the pause menu actually pauses the game.

We have a nice platformer with interesting exploration mechanics, a fully working state system, and a pause menu. The last thing we need to conclude this chapter is some rooms to travel to and a working map system to keep track of the movements of the player.

Warped!

Until that moment, we travelled through rooms without the possibility to go back. The avatar walked through a level, reached the end, and got spawned to the next room – never going back.

This makes sense if you're working on a game in the style of Super Mario Bros.; but if you're creating an exploration-based game, you may want your character to be able to walk back and forth, retraversing the same path multiple times.

This means that we cannot use blindly the room_next() or room_goto() function anymore. We need to warp the character to a certain position in a certain room depending on how we exit a room. For example, if we leave a room from the far-right door, we expect to warp in the next room starting from the far left. But if we decide to move the other way and exit from the far left, we expect to enter the next room from the far right.

The first thing that comes to mind, using our skills and knowledge until now, is to create a unique object for every warp; but this feels wrong, doesn't it? Well, your coder sense is tingling for a good reason: it definitely is wrong!

To achieve this goal, we need to create a single warp object and change some variables instance by instance from the Room Editor.

Note In this chapter, we are not covering level design in detail. We did it in the previous chapter, and I am assuming you're now able to set up the first room featuring a basic layout based on block objects like Cherry Caves 1 or a tiled one, if you prefer like in Cherry Caves 2.

If you didn't already, set up your first room as you like, using the things learned in Chapters 8 and 9 and activate a viewport and a camera for the room. As we said at the beginning of the chapter, every room will be a small side-scrolling room.

Let's start by creating a second room called room1. Create some basic floor and make sure you leave an entrance at both the left and right borders. Those will be our entrances/exits for this room. Edit room0 accordingly to achieve the same goal: having two entrances/exits.

The result should be something similar to that in Figure 11-9.

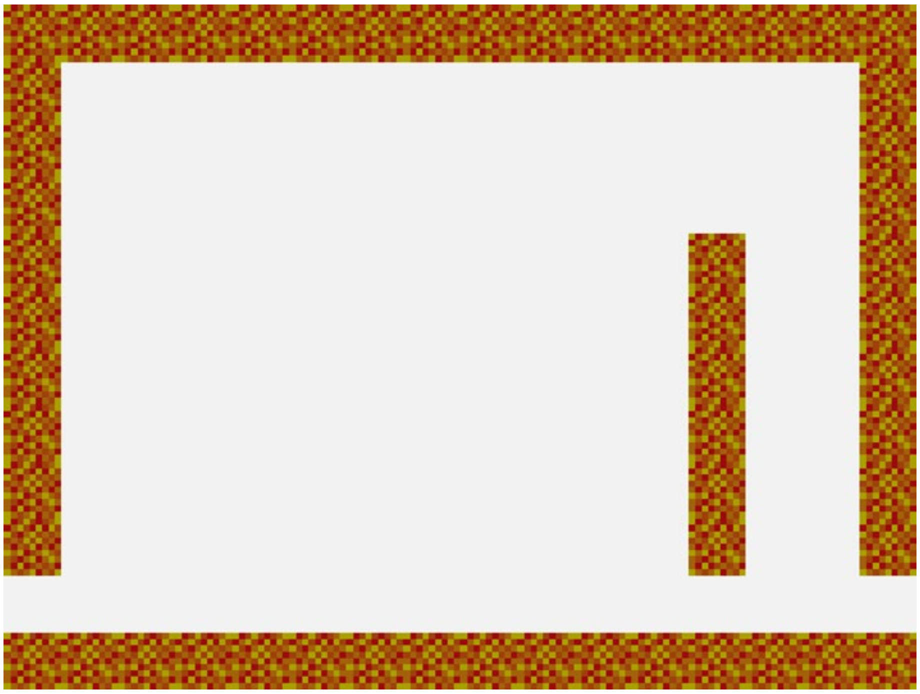


Figure 11-9.

Now let's create a warp object called obj_warp. Assign spr_warp to it and make it not visible by ticking the right box.

The only event we need for this object is the interaction with the player. So click Add Event ➤ Collision ➤ obj_player and insert this code in it:

```
1   if room_exists(target_room)
2   {
3       other.x = target_x;
4       other.y = target_y;
5       other.dashing = false;
6       other.already_dashing = false;
7       room_goto(target_room);
8   }
```

To travel between rooms, we are going to use three variables: target_room which is the room we want to warp the character to and target_x and target_y which are the X- and Y-coordinates of the new position in the new room.

So we check if the target room exists (line 1); and if it does, we change the obj_player's coordinates to the new target coordinates (lines 3–4), and we teleport to the next room (line 7). We also need to stop dashing, in case the player is dashing, because the warp changes the player's position, making the calculation of the dash goal inconsistent.

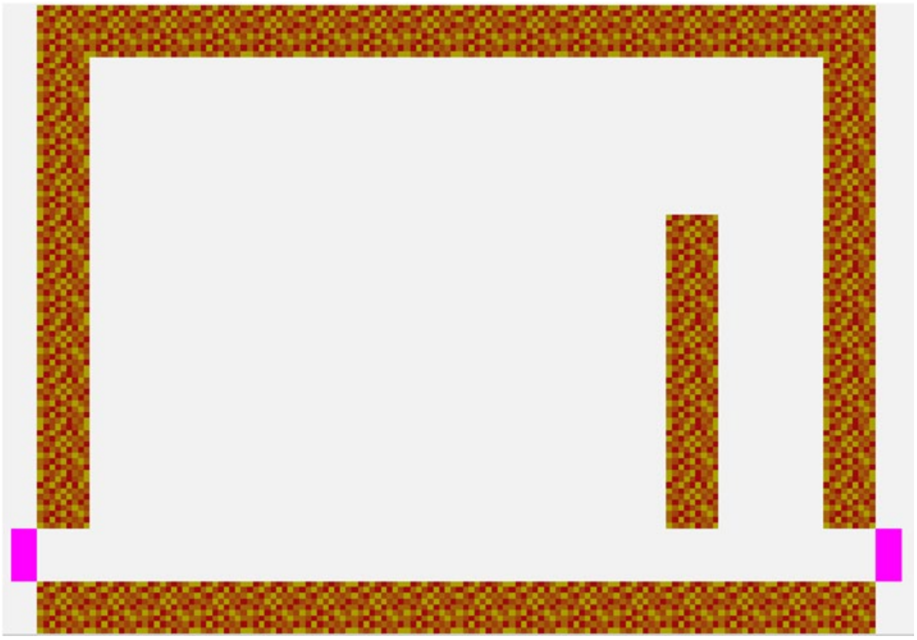


Figure 11-10. *The room with the two obj_warp instances at the two entrances*

Now that we have our obj_warp ready, let's position two instances of it in room0, one for each exit (Figure 11-10).

Now double-click the far-left instance, and a small window will pop up. This is the Instance Editor. You can modify every single instance you place in a room to personalize how they act. This allows you to create a general object with many behaviors and change the behavior accordingly to your needs, right in the Room Editor.

With the Instance Editor open, click Creation Code. The creation code is a piece of code that is executed every time once when that instance is created.

In the creation code, place this code that initializes the variables we are using to warp to the next room:

```
target_room = room1;
target_x = room_width-100;
target_y = 670;
```

Please note that the value of my target_y is 670 as a consequence of the height of my room (768 pixels) and the 64 × 64 floor tiles. If you made a room with a different height or you want to spawn the avatar at a different height, change this accordingly.

Now do the same with the far-left instance and add this to its creation code:

```
target_room = room1;
target_x = 100;
target_y = 670;
```

Before we check that everything is working right, let's add a couple of warps also in room1.

Add them to the room just as we did for room0: one at the left entrance and one at the right entrance.

Double-click the left warp and add this creation code to it:

```
target_room = room1;
target_x = room_width-100;
target_y = 670;
```

Now double-click the right warp and add the following creation code:

```
target_room = room1;
target_x = 100;
target_y = 670;
```

Ok, now we are all set! Save and run the game and enjoy your fully working warping system!

Conclusion

That was fun, huh?! We created a solid foundation for a nice action-adventure platformer. To make it a true metroidvania, we need some more features, like a map, an inventory to carry our items, and maybe some enemies and a cool combat system.

In the next chapter, we will cover all those features; and we will also implement a saving system for our game, so that we can continue playing from the last checkpoint we visited when we die!

TEST YOUR KNOWLEDGE!

1. Which kinds of gamepads are supported by GMS2?
2. How can you handle input from an XInput gamepad?
3. How do you set up the deadzone of the analog stick of a XInput gamepad?
4. What is a wall jump? Why it's so important in an exploration-based game?
5. How does a wall jump work, from a technical point of view?
6. Can you modify the wall jump skill so that after the first wall jump, if the character finds another wall while rebounding, they automatically perform another wall jump?
7. How does wall sliding work?
8. Why is wall sliding important from a level design point of view?
9. What is a dash?

10. How does a dash work, from a technical point of view?
11. Can you modify the dash code to change it into a backstep like the one in Castlevania: Symphony of the Night?
12. How does the warp system work? Why is it better than the methods we used so far to manage room entrances/exits?

CHAPTER 12

Metroidvania (Part 2)

In the previous chapter, we created the base platforming game system on which to build our metroidvania. In this chapter, we are going to finish the job by adding all the fundamental features a metroidvania must have:

- Map screen
- Minimap
- Inventory screen
- Items
- Equipment
- A combat system affected by the equipment
- Enemies
- The possibility to save the game
- Checkpoints

It won't be easy or short, but I promise it's going to be fun!

We will make extensive use of data structures (which were introduced in chapters 3 and 4, building the card game) to create the map and inventory systems, and we will take inspiration from Space Gala and Cherry Caves to create our combat system and enemies.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_12

To implement the saving system and checkpoints, we will talk about file management with GML and the JSON standard, which is a very popular object notation introduced by JavaScript to represent data in an attribute-value fashion.

About maps

A good map can save your journey, both in games and real life. A game made with extensive exploration moments that doesn't offer some kind of map is a game that doesn't understand the player's needs.

From ARPGs to platformers, from strategic games to shooters, nearly any game genre may need a map. Whenever there is exploration and discovery involved, you need a good map. If the player has a choice about where to go or you plan some backtracking in your games, you need a map.

There are many kinds of maps in video games, from realistic maps full of details to basic stripped-down maps. The style of your map can depend on both narrative and gameplay reasons.

For example, in DOOM (id Software, 1993) and DOOM 2 (id Software, 1994), you have a very basic map exclusively made of lines. This serves the purpose of representing a believable map which resides in some kind of portable pocket computer that space marines have in their standard equipment. This is a narrative reason.

When you compare DOOM level maps to DOOM's world map, you get the idea; the latter is a way more detailed map and shows all the areas and the places you visited crossed with a red X and the places you are going to visit. When you see it, you immediately understand that it's another kind of map: it's a map for the player, showing all the progresses they made so far.

In fact, it's clear that DOOM's world map is out of the narrative scope and your space marine cannot see it. When you end a level, the fourth wall falls down; and the game shows you how much time you spent in that level, what was your score, how many collectibles you got, and where you're going to spawn next. It's a pure gameplay moment where the narrative is temporarily put on hold. This is why the game doesn't need to be coherent with the style of the level's map. It's a pause, for the player, from the frantic gameplay tied to level exploration – a moment to relax, release the tension, and get ready for the next challenge. It's a good design technique that allows the levels to be more punishing and challenging, since the player knows that they're going to reach a safe place where they can relax and release the tension. The end of the level is an actual escape from those hellish areas.

Action-adventure games – particularly metroidvania games – make extensive use of maps. Traditionally, they offer two kinds of map: a small map that resides in the HUD and you can examine in any moment and a full map, accessible through a dedicated screen, which shows you all the areas available, so that you can plan your next move.

As in the DOOM example, many offer a kind of map that suits the narrative and fosters immersion, but many more decide to concentrate on readability, clarity, and providing information. In fact, being the map the thing that you check more frequently in an exploration-focused game, it has to be clear and quickly readable, so that the player doesn't get confused or slow down.

An example of a hybrid map that manages to elegantly blend narrative and gameplay is Hollow Knight's map. It's a complex and aesthetically detailed map which features the possibility to add pins while exploring, to mark places of interest like boss fights, NPCs, stores, checkpoints, and so on.

You get access to Hollow Knight's maps by buying them from a map maker whom you can find while travelling, since he's exploring the world too. This gives a narrative reason to the existence of maps, which fosters immersion.

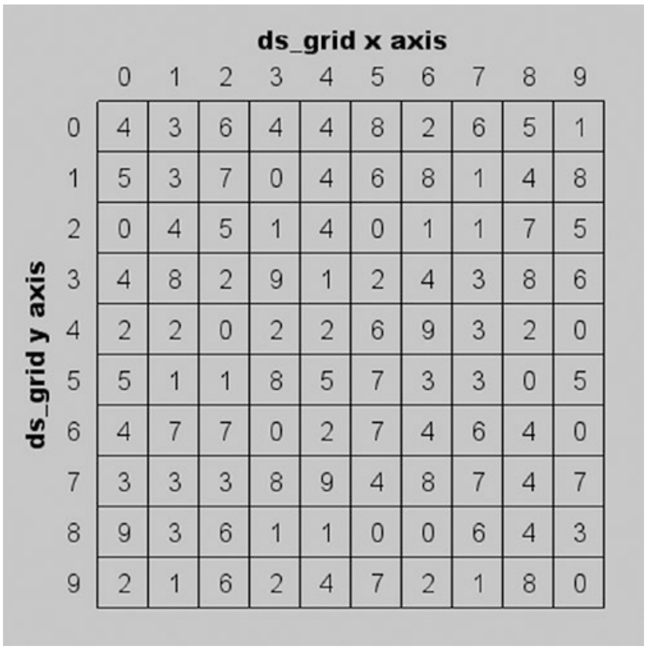
Another interesting thing about Hollow Knight's maps is that when you sit on a bench (the checkpoints of the game), if you have a pen (it's an actual item in the game), you can write on maps adding the details you discovered exploring. This is a huge addition that actually gives importance to exploration and makes the player feel like they're discovering uncharted places that even the cartographer didn't reach or know.

No matter how detailed and beautiful a map can be, underneath there is always a data structure representing rooms' interconnection. In this chapter, we will create a basic map system for our game by using a very important and useful GameMaker's data structure: DS Grid.

Map makers, grids, and semaphores

To create Isolation's map, as we said, we will use a grid.

Grids, or DS Grids, are basically two-dimensional arrays with some more features and dedicated utility functions.


ds_grid x axis
ds_grid y axis

	0	1	2	3	4	5	6	7	8	9
0	4	3	6	4	4	8	2	6	5	1
1	5	3	7	0	4	6	8	1	4	8
2	0	4	5	1	4	0	1	1	7	5
3	4	8	2	9	1	2	4	3	8	6
4	2	2	0	2	2	6	9	3	2	0
5	5	1	1	8	5	7	3	3	0	5
6	4	7	7	0	2	7	4	6	4	0
7	3	3	3	8	9	4	8	7	4	7
8	9	3	6	1	1	0	0	6	4	3
9	2	1	6	2	4	7	2	1	8	0

Figure 12-1. *A basic example of a DS Grid*

In Figure 12-1, you can see what a DS Grid looks like. Think of it as a table working exactly like the coordinates on your screen, with X- and Y-axis. Both the systems (your screen and DS Grids) are Cartesian coordinate systems (CCSs). In a CCS, a point is identified by a couple of coordinates (x,y) where, thinking in a column-per-row fashion, x (the index of the X-axis) is the column index and y (the index of the Y-axis) is the row index.

Some nice features of this data structure vs. a common array are the possibility to perform a fairly quick search by value, resize the grid, sort it, shuffle it, and decide to act only on specific regions of the grid.

To create a new DS Grid, you can use the ds_grid_create function specifying the width and the height of the grid. Note that the width can be seen as the number of columns and the height as the number of rows:

```
var my_grid = ds_grid_create(width, height);
```

This is a function that takes the width (number of columns) and height (number of rows) of the DS Grid you want to create as inputs and returns a real that represents the id of the newly created DS Grid.

You can access a DS Grid's element both by using ds_grid_get and by using the # accessor.

To access a DS Grid via ds_grid_get, you have to specify the id of the grid and the X- and Y-coordinates of the element you want to retrieve in that grid like this:

```
var val = ds_grid_get(my_grid, x,y);
```

To access the same element using the # accessor is like this:

```
var val = my_grid[# x, y];
```

It's up to you which approach to have when accessing your grids. The two methods do exactly the same thing.

You can search for a specific value in a DS Grid and receive its coordinates in return by using ds_grid_value_x and ds_grid_value_y like this:

```
var row = ds_grid_value_x(my_grid, reg_x0, reg_y0, reg_x1,
reg_y1, my_val);
var col = ds_grid_value_y(my_grid, reg_x0, reg_y0, reg_x1,
reg_y1, my_val);
```

In both functions, you must specify the region in the DS Grid in which you want your search to be done. To do it, you have to tell GameMaker the start and end coordinates of the region in the grid. The search will be done only between the elements of that region. An example of a DS Grid region is shown in Figure 12-2.

In the preceding example, the start coordinates are reg_x0 and reg_y0, while the end coordinates are reg_x1 and reg_y1.

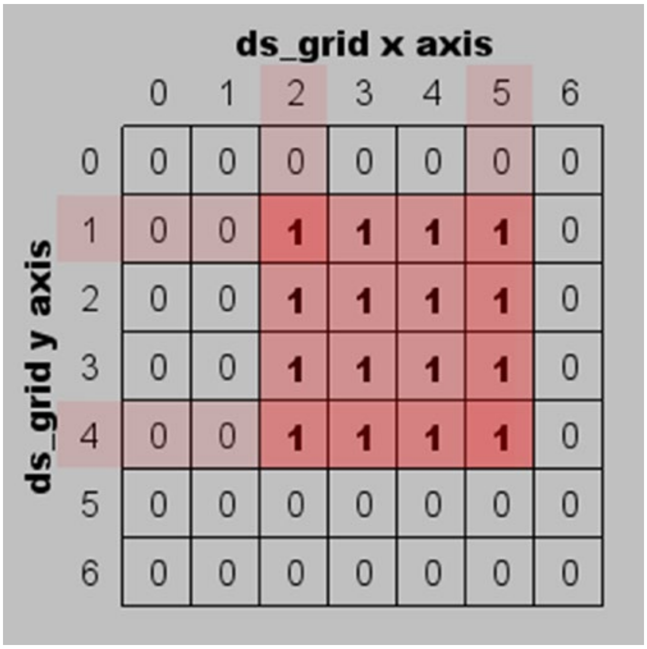


Figure 12-2. *The pink square is a grid region inside a DS Grid. This specific region starts at (1,2) and ends at (4,5).*

We will use the grid to represent the rooms' arrangement and visualize both in the full and Minimaps where the player is located.

Let's start by creating a new object called obj_map and make it persistent, so that it will maintain the information when the player's character changes room.

Add a create event to obj_map. In this event, we will define the DS Grid we are going to use to represent the map in the game:

```
1   map_h = 1;
2   map_w = 3;
3   map = ds_grid_create(map_h, map_w);
4   map[# 1,0] = room_get_name(room0);
5   map[# 1,1] = room_get_name(room1);
6
7   open_map = false;
```

At lines 1 and 2, we define map_w and map_h, which are the width (number of columns) and height (number of rows) that compose the DS Grid map.

We use those values at line 3 to create the DS Grid using the ds_grid_ create function, as we saw earlier.

Then we assign the two rooms we created in the previous chapter to two elements in the 1 × 2 DS Grid we just created, so that the first element at position 0,0 is room0 and the second element at position 0,1 is room1 (Figure 12-3).

Each element that has a value different from zero represents a room. The zeroed elements do not, and so they won't be drawn on screen. This allows us to represent maps with interesting and complex shapes.

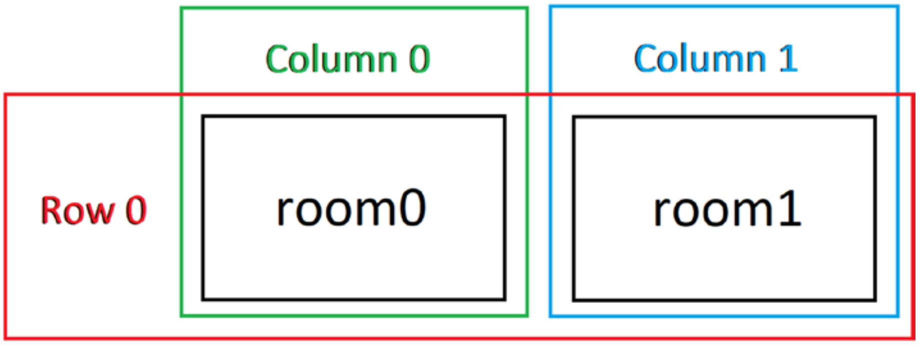


Figure 12-3. *An example of how we represent rooms with a DS Grid. Rooms with the same row value are on the same floor; rooms with the same column value and different row value are stacked on each other.*

Lastly, at line 7, we initialize a controller variable, labelled open_map, to false. We will use this variable to check whether the map is closed or open, just like menu_open in obj_controller.

Just like when we managed the pause menu in obj_controller, we need a Step event for obj_map to handle the player's input to open and close the menu, calculate the player's character position inside the grid, and of course manage the game state and switch on or off the controller variable accordingly.

We have a problem, though. In fact, we already have a menu that shows up taking control of the entire screen, when we press the Esc key: the pause menu in obj_controller. How can we manage this situation?

Well, this is a common problem in Computer Science and programming, and it's called **race condition**. Race condition happens when two actors try to access the same resource trying to modify it affecting the result of each other's calculations. When there is a race condition, there must be a concurrency algorithm to schedule the two concurrent activities.

Two activities are said concurrent when they both try to access the same resource. This can be particularly troublesome when the two (or more) activities try to modify that resource (causing a race condition). Let's make an example!

Say that two people, Bob and Alice, have a shared bank account.

Both Bob and Alice can access the account and withdraw money. When they try to withdraw, the ATM reads how much money there is on the bank account, asks how much they want to withdraw, and finally gives them the right amount of money.

Imagine those two people trying to withdraw money at the same time. Let's say there are $50 on their bank account. Bob and Alice decide to withdraw $40 at the same time. Both the ATMs read the available money on the account; they both see that there is enough money to withdraw $40 and lend the money. Both Bob and Alice receive $40, and they still have $10 on their shared bank account (Figure 12-4). The problem is that the bank just gave them $80! This is an example of an unaddressed race condition.

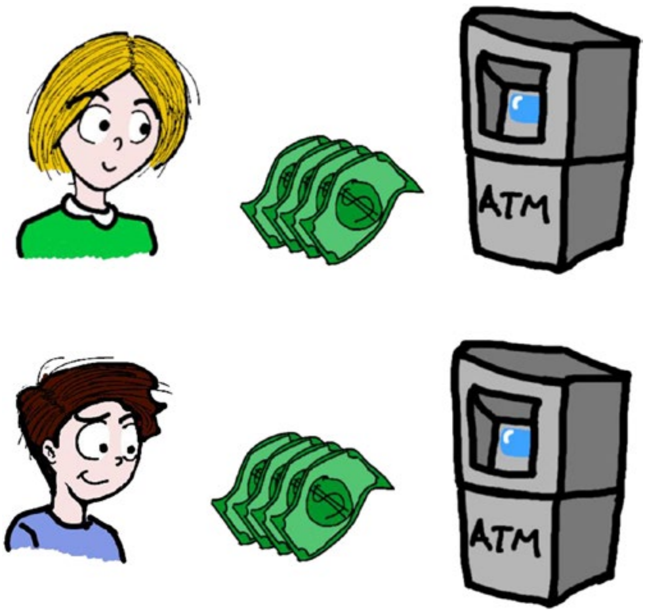


Figure 12-4. *Alice and Bob both receive $40 from the ATM, even if they have just $50 in their shared account. This is because the race condition wasn't addressed by the ATM programmer.*

We can avoid this kind of problem by scheduling the two activities implementing mutual exclusion. Mutual exclusion (also known as mutex) is a programming technique that synchronizes two activities subject to race condition by giving them exclusive access to a resource one at a time.

Mutex is often achieved with a semaphore, which is a variable or an abstract data type that is used to control the access to a resource.

For example, we can use a semaphore to solve the problem of the bank account! All we need is that, when Bob or Alice try to access their account, the ATM sets a flag which states that someone is accessing the account. So, when Bob and Alice try to access the account at the same time, one of the two ATMs will find out that someone else is accessing that same account and will wait for its turn before performing actions on that account.

We can use mutex and semaphore also to address our menu problem. We can set a global variable, global.can_pause, and use it to give exclusive access to the display of the menus. So, as shown in Figure 12-5, when a menu is opened, global.can_pause is set to false, so that no other menu can be opened. When the open menu is closed, global.can_pause gets reset to false.

This assures us that the player can open only one menu at a time.

Going back to our obj_map's Step event, we must handle the opening and closing of the map screen similarly to how we handle the pause menu, but with the introduction of the semaphore variable.

Here's the code to do it:

```
map_key = keyboard_check_pressed(vk_tab) or gamepad_button_
check_pressed(0, gp_select);

if map_key
{
    audio_play_sound(snd_menu, 1, false);
6       if open_map
7       {
8           open_map = false;
9           global.can_pause = true;
10          global.game_state = states.playing;
11      }
12      else
13      {
14          if (not global.can_pause) exit;
15          open_map = true;
16          global.can_pause = false;
17          global.game_state = states.paused;
18      }
19  }
```

Line 1: We check for the player's input. We want the map to open when the player presses the Tab key on the keyboard or the Select button on a gamepad.

Lines 3–18: We manage the opening and closing function of the map screen, just like we did for the pause menu in obj_controller's Step event.

When the map key is pressed, if the map is already open, we close it by changing the value of open_map to false, reset to true the value of the global variable can_pause (so that we can open again the map or the pause menu), and change the state of the game to the playing state, so the action can be resumed (lines 6–11).

When the map key is pressed, if the map is not already open, we check if there is another menu open by checking the value of global.can_pause. If there actually is another menu open, we give up trying to open the map and stop executing the code (line 14). Otherwise, if no other menu screen is being displayed, we can take the control of the screen by setting global.can_pause and open_map to true and change the state of the game to paused (lines 15–17).

We must add the semaphore also to manage the pause menu in obj_controller. So head to obj_controller's Create event and append this line to the code:

```
global.can_pause = true;
```

Now, in obj_controller's Step event, change the code related to the menu so that it looks like this:

```
// --- MENU --- //
if esc_pressed
{
    audio_play_sound(snd_menu, 1, false);
    if ( global.game_state == states.playing and
    global.can_pause )
    {
        global.game_state = states.paused;
        audio_play_sound(snd_menu, 1, false);
        menu_open = true;
        global.can_pause = false;
    }
    else if ( global.game_state == states.paused and
    menu_open )
    {
        global.game_state = states.playing;
        menu_open = false;
        global.can_pause = true;
    }
}

if ( menu_open )
{
    menu_index += move;

    if ( move != 0 )
    {
        audio_play_sound(snd_menu, 1, false);
    }

    if ( menu_index < 0 )
    {
        menu_index = opt_number - 1;
    }
    else if ( menu_index > opt_number - 1 )
    {
        menu_index = 0;
    }

    if ( enter_pressed )
    {
        switch( menu_index )
        {
            case 0:
                global.game_state = states.playing;
                menu_open = false;
                global.can_pause = true;
                break;
            case 1:
                game_restart();
                break;
            case 2:
                game_end();
                break;
        }
    }
}
```

The condition to open the pause menu is that no other menu is already open – meaning global.can_pause is false (lines 5 and 12). Because of this, we change the value of global.can_pause at lines 10 and 16 when we open or close the pause menu.

Following the same idea, we set global.can_pause to true when the player chooses the Resume option in the pause menu (line 45).

Now, open up obj_map and add a Draw GUI event. In this event, we will draw in the HUD both the full map and the minimap.

To draw the full map, we just traverse the map line by line and draw each element which has a value different from zero. It's basically a graphic representation of the grid.

Go ahead and add this code to obj_map's Draw GUI event:

```
var cam_w = display_get_width();
var cam_h = display_get_height();

pos_x = ds_grid_value_x(map, 0, 0, map_w-1, map_h-1, room_get_name(room));
pos_col = ds_grid_value_y(map, 0, map_w-1, 0, map_h-1, room_get_name(room));

if open_map
{
    draw_set_alpha(0.5);
    draw_set_color(c_black);
    draw_rectangle(0, 0, cam_w, cam_h, 0);
    draw_set_alpha(1);
    var cur_x = cam_w/3;
    var cur_y = cam_h/3;
    var box_h = 50;
    var box_w = 100;
    var box_offset = 30;
18      for(var i = 0; i < map_h; i++)
19      {
20          for(var j = 0; j < map_w; j++)
21          {
22              if map[# j, i] != 0
23              {
24                  draw_set_color(c_white);
25                  if map[# j, i] == room_get_name(room)
26                  {
27                      draw_set_color(c_yellow);
28                  }
29                  draw_rectangle(cur_x, cur_y, cur_x + box_w,
                    cur_y + box_h, 0);
30              }
31              cur_x += box_offset + box_w;
32          }
33          cur_x = cam_w/3;
34          cur_y += box_offset + box_h;
35      }
36  }
```

Lines 1–2: We get the width and height of the camera. We need this to properly draw elements on the screen.

Lines 3–4: Calculate the position of the room in which the player's character is in that moment. We search for the X- and Y-coordinates inside the DS Grid map using ds_grid_value_x and ds_grid_value_y. We need this information to correctly draw both the full and Minimaps.

If the map is open (line 7), we draw a semi-transparent black background; and then, from 1/3 of the camera width, we start to draw one by one all the elements in the grid that represent rooms, using rectangles of 100 × 50 pixels (lines 13–36). The rectangles are separated by an offset of 30 pixels.

For each element, we check whether its value is zero or not. If it's zero, we skip it and leave the space blank.

If the value of the element is not zero, we check again whether the value is the name of the current room or not. If it is, we draw a yellow rectangle; if it's not, we draw a white rectangle.

The result is a representation of the grid where the room in which the player is is highlighted in yellow, so that the player knows where they are in that very moment, compared to the vastness of the map (well, in this case there are just two rooms, but you get the point).

The Minimap is something very similar to the full map, aesthetically, but its code is very different. In fact, the Minimap shows only a subsection of the full map. We want it to show only the rooms adjacent to the current room. Also, the Minimap will always be visible during the gameplay (except when the full map is open) in the top-right corner of the screen.

To correctly draw the Minimap, we have to check if the element containing the name of the current room has any elements in the eight directions around it: north, north-east, east, south-east, south, south-west, west, north-west.

For each of these elements, we have to check whether it's a room or not and eventually draw it, if it is. Even if in this case we always draw the current room in the middle, we apply the same coloring rules of the full map: a yellow rectangle for the current room and a white rectangle for all the other rooms.

So, just under the code for the full map, append the following code to draw the Minimap when the full map is not open:

```
else // if the full map is not open
{
    var box_x = cam_w-100;
    var box_y = 80;
    var box_w = 40;
    var box_h = 20;
    var box_offset = 10;

    // draw the current room
    draw_set_color(c_yellow);
    draw_rectangle(box_x, box_y, box_x + box_w, box_y +
    box_h, 0);

    // draw the rooms adjacent to the current room
    draw_set_color(c_white);

    var west_room = pos_x > 0;
    var east_room = pos_x < (map_w-1);
    var south_room = pos_y < (map_h-1);
    var north_room = pos_y > 0;

    if west_room // draw the west room
    {
        if ( map[# pos_x-1, pos_y] != 0 )
        {
            var b_x1 = (box_x) - (box_offset + box_w);
            var b_x2 = (box_x) - (box_offset);
            var b_y1 = box_y;
            var b_y2 = box_y + box_h;
            draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
        }
    }

    if east_room // draw the east room
    {
        if ( map[# pos_x+1, pos_y] != 0 )
        {
            var b_x1 = (box_x + box_w) + (box_offset);
            var b_x2 = (box_x+box_w) + (box_offset + box_w);
            var b_y1 = box_y;
            var b_y2 = box_y + box_h;
            draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
        }
    }

    if north_room
    {
        if map[# pos_x, pos_y-1] != 0 // draw the north room
        {
            var b_x1 = box_x;
            var b_x2 = box_x + box_w;
            var b_y1 = (box_y) - (box_offset + box_h);
            var b_y2 = (box_y) - (box_offset);
            draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
        }

        if west_room // draw the north-west room
        {
            if map[# pos_x-1, pos_y-1] != 0
            {
                var b_x1 = (box_x) - (box_offset + box_w);
                var b_x2 = (box_x) - (box_offset);
                var b_y1 = (box_y) - (box_offset + box_h);
                var b_y2 = (box_y) - (box_offset);
                draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
            }
        }

        if east_room // draw the north-east room
        {
            if map[# pos_x+1, pos_y-1] != 0
            {
                var b_x1 = (box_x + box_w) + (box_offset);
                var b_x2 = (box_x+box_w) + (box_offset + box_w);
                var b_y1 = (box_y) - (box_offset + box_h);
                var b_y2 = (box_y) - (box_offset);
                draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
            }
        }
    }

    if south_room
    {
        if map[# pos_x, pos_y+1] != 0 // draw the south room
        {
            var b_x1 = box_x;
            var b_x2 = box_x + (box_w);
            var b_y1 = (box_y + box_h) + (box_offset);
            var b_y2 = (box_y + box_h) + (box_offset + box_h);
            draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
        }

        if west_room // draw the south-west room
        {
            if map[# pos_x-1, pos_y+1] != 0
```

```
95              {
96                  var b_x1 = (box_x) - (box_offset + box_w);
97                  var b_x2 = (box_x) - (box_offset);
98                  var b_y1 = (box_y + box_h) + (box_offset);
99                  var b_y2 = (box_y + box_h) + (box_offset +
                    box_h);
100                 draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
101             }
102         }
103
104         if east_room // draw the south-east room
105         {
106             if map[# pos_x+1, pos_y+1] != 0
107             {
108                 var b_x1 = (box_x + box_w) + (box_offset);
109                 var b_x2 = (box_x+box_w) + (box_offset +
                    box_w);
110                 var b_y1 = (box_y + box_h) + (box_offset);
111                 var b_y2 = (box_y + box_h) + (box_offset +
                    box_h);
112                 draw_rectangle(b_x1, b_y1, b_x2, b_y2, 0);
113             }
114         }
115     }
116 }
```

The code is pretty much self-explanatory. First, at lines 10–11, we draw the current room; then, we check for every one of the eight directions if there is an element adjacent to the element that contains the current room (lines 16–19).

For each one of the directions, if there is an element, we check if it's a valid element or not (has a non-zero value) and we draw it on the screen in the right position.

The concept to calculate the eight directions inside the grid is pretty straightforward too. An element is at the north of another element if it is one row up, and it is at the south if it's one row below. Similarly, an element is at the east or at the west of another element if it is, respectively, one column ahead or behind the current element.

Basically, for an element at position x,y, we can check the eight directions around it in the grid, like this:

- **North**: One row up, same column (x, y-1)
- **North-East**: One row up, one column ahead (x+1, y-1)
- **East**: Same row, one column ahead (x+1, y)
- **South-East**: One row down, one column ahead (x+1, y+1)
- **South**: One row down, same column (x, y+1)
- **South-West**: One row down, one column behind (x-1, y+1)
- **West**: Same row, one column behind (x-1, y)
- **North-West**: One row up, one column behind (x-1, y-1)

Now, before you save and run the game, make sure that you put the obj_map element in room0. Running the game, you should be able to open and close both the pause and map menus without overlapping them.

The map menu will show the room in which the player's character is at any time.

If you want to create new rooms and make a more complex map, just create the rooms and add them in the DS Grid in the position you like. Don't forget to complete all the rows with zeroed elements.

For example, if you want to add a room just below room1, the map should look like this:

```
map[# 0,0] = room_get_name(room0);
map[# 1,0] = room_get_name(room1);
map[# 0,1] = 0;
map[# 1,1] = room_get_name(room2);
```

Ok, now it's the time to save and run the game to enjoy the simple yet effective map we created! This is a very important achievement! You learned about a lot of things: DS Grids, map design, and even concurrency and race condition! But more importantly, you also managed to create something out of the learning! So congratulate yourself, enjoy your new toy, and get ready for the next section, in which we are going to focus on the next big thing: items and inventory!

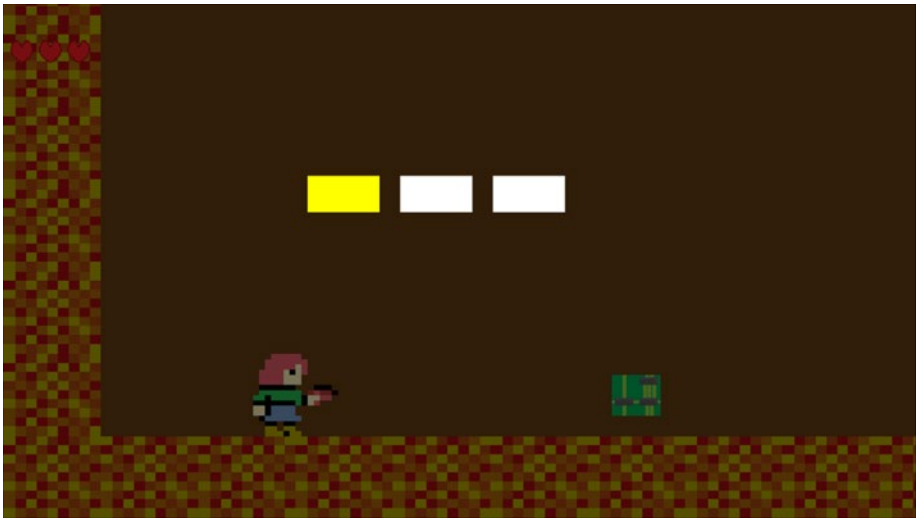


Figure 12-5. *The full map featuring three rooms as shown in the map screen*

Items and inventory

A fundamental feature that any metroidvania and action-adventure game has is the possibility to pick up objects and use them. In particular, a very popular thing among those games is the concept of equipping special items like weapons or armors and making our avatar more powerful. But to have this kind of items, we have to implement the possibility to carry them with us. For this reason, it's not very convenient to represent items as object's instances. We need some more light and manageable kind of data: a data structure.

An important thing to note is that items should carry some information with them:

- **ID**: We want to uniquely identify each item in the game, so that we are able to distinguish between two items of the same kind, like two potions, two identical weapons, and so on. Actually, it's not very smart to use the id word, since it's a reserved word of GML. Probably it would be better to use something different, like key.
- **Name**: We want to access a human-readable name for that item. This is what we will show to the player in the inventory.
- **Description**: Of course, we need a description of the item, so that we can tell the player what's that item and how to use it.
- **Type**: We need a piece of data that can tell us what kind of item is this. Is this a cure? Is this a weapon?
- **Value**: This is a jolly field. Depending on the kind of item we are treating, this can be, for example, the amount of HP cured by a potion or the attack power that gives to the player's character.

We can add as many fields we want to our items, but those are pretty much the basics.

To organize all those fields, we could use one of the data structures offered by GameMaker. In particular, the most fitting for this case is a DS Map.

A DS Map is a data structure that lets you store pairs of keys and values. DS Maps are particularly useful to store mixed-type data (strings, numbers, etc.), and, since they store data organized with keys, they have the fastest access to data. You just need to provide the key to access a specific value.

GameMaker offers a wide variety of functions to manage DS Maps. Let's see some of the most interesting.

ds_map_create()

This function creates an empty DS Map. It returns the empty DS Map.

For example:

```
var my_map = ds_map_create();
```

ds_map_add(id, key, val)

This function adds a key,value pair to the DS Map indicated by the id value. The function fails if the specified key already exists inside the DS Map.

For example:

```
ds_map_add(map_id, "name", "Seb");
```

ds_map_replace(id, key, val)

This function is used to add or replace a key,value pair inside a DS Map. Differently from ds_map_add, this one won't fail if the key already exists inside the DS Map, but it will replace the value.

For example:

```
ds_map_replace(map_id, "name", "Sebastiano");
```

ds_map_empty(id)

This function checks whether the DS Map is empty or not. It returns a Boolean value.

For example:

```
if (ds_map_empty(map_id)) exit;
```

ds_map_find_value(id, key)

This function reads the value associated with a certain key, if it exists. If no such key exists, it will return undefined.

For example:

```
var my_name = ds_map_find_value(map_id, "name");
if is_undefined(my_name) exit;
```

So that's the idea: we can represent items as a collection of keys and values like in Table 12-1.

***Table 12-1.** Keys and values*

Key	Value
ID (or key)	123456…
Name	"Health Potion"
Description	"This can restore your HP!"
Type	item_type.cure
Value	1

Using this format, it will be pretty easy to carry them with us! In fact, it will suffice to create a list to store all the items carried. Picking up an item means copying it into the list.

So we need an inventory, which, as we said, is basically a list of all the items the character owns. We already saw lists in chapters 3 and 4. We used them to manage our deck of cards, shuffle it, sort it, pick cards, and compare them. That's not too different from what we need to do right now with items!

Let’s create the inventory in obj_player as a DS List. We will use it to manage the items that the player decides to bring with them.

Open up obj_player’s Create event and append these lines at the bottom of the code:

```
1   items = ds_list_create();
2   equipped = ds_map_create();
```

At line 1, we define our inventory as a DS List, as we discussed. We will add the items found to that list.

At line 2, we define a variable representing the item that we are currently equipping. The equipped item is an item in our inventory that we want to wear or wield. We cannot equip an item that is not in the inventory, and we can only equip one item at a time. Equipping an item while another item is already equipped will automatically unequip the currently equipped item.

Items, as we said, can be differentiated by their type. In this project, we only have two types of objects: weapons and cures. We will represent them using an enum, so that we can refer to the different types in a human-readable fashion. So head to obj_controller’s Create event and add the following lines of code:

```
enum item_type {
    weapon,
    cure
};
```

With this piece of code, we will be able to distinguish between weapons that we can equip whenever we want and cure items that we can use only one time and have the immediate effect of increasing the player’s HP.

The inventory will be shown in a dedicated screen, just like the pause menu and the full map.

To create and manage an inventory screen, we need a controller variable which tells us when the inventory is open and a cursor variable that can help us keep track of the position of the cursor while navigating in the inventory.

Let's create those two variables by appending the following lines at the bottom of obj_controller's Create event:

```
inv_open = false;
inv_index = 0;
```

To manage the opening and closing actions for the inventory, just as we did for both the pause menu and the map screen, we have to add yet another check for inputs at the top of obj_controller's Step event:

```
var inv_key = keyboard_check_pressed(ord("I")) or gamepad_
button_check_pressed(0, gp_face4);
```

The inventory key is set when the player presses the letter I on the keyboard or the Face 4 button on the gamepad (triangle on PlayStation gamepads or Y on Xbox gamepads).

The first thing to do is to check if the inventory is already open when the inventory key is pressed by the player; and in that case, we close it (lines 4–9 of the following code). If the inventory is not already open, we have to check if the screen is already claimed by another menu (line 12) or if we can open the inventory menu; in that case, we claim the screen by setting global.can_pause to false, we set the controller variable to true, and then we change the game state to paused (lines 13–15).

Let's do this by appending this code at the bottom of obj_controller's Step event:

```
1   if inv_key
2   {
3       audio_play_sound(snd_menu, 1, false);
4       if inv_open
5       {
6           global.game_state = states.playing;
```

```
        global.can_pause = true;
        inv_open = false;
    }
    else
    {
        if not global.can_pause exit;
        global.game_state = states.paused;
        global.can_pause = false;
        inv_open = true;
    }
}
```

Now, if the inventory is open, we want to show each item that's in the items DS List and give the possibility to the player to navigate that list with a cursor.

Moreover, we want to give the possibility to equip one item when the cursor points to an equippable item (a weapon) and to use it if it's a consumable item (a cure).

To achieve this, append the following code just below the code we just wrote in the previous paragraph, in obj_controller's Step event:

```
if inv_open
{
    if instance_exists(obj_player)
    {
        if (move != 0) audio_play_sound(snd_menu, 1, false);
        inv_index += move;
        if inv_index < 0
        {
            inv_index = ds_list_size(obj_player.items) - 1;
        }
        if inv_index >= ds_list_size(obj_player.items)
        {
            inv_index = 0;
        }

        if enter_pressed and ds_list_size(obj_player.items)
        > 0
        {
            switch(ds_map_find_value(obj_player.items[|
            inv_index], "type"))
            {
                case item_type.weapon:
                {

                    if ( ds_map_empty(obj_player.equipped) or
                        ds_map_find_value(obj_player.
                        equipped, "key") != ds_map_find_
                        value(obj_player.items[|inv_index],
                        "key") )
                    {
                        var key = ds_map_find_value(obj_
                        player.items[| inv_index], "key");
                        var name = ds_map_find_value(obj_
                        player.items[| inv_index], "name");
                        var type = ds_map_find_value(obj_
                        player.items[| inv_index], "type");
                        var value = ds_map_find_value(obj_
                        player.items[| inv_index], "value");
                        var desc = ds_map_find_value(obj_
                        player.items[| inv_index], "desc");
                var bullet_type = ds_map_find_value(obj_player.items[| inv_index], "bullet_type");

                ds_map_replace(obj_player.equipped, "key", key);
                ds_map_replace(obj_player.equipped, "name", name);
                ds_map_replace(obj_player.equipped, "type", type);
                ds_map_replace(obj_player.equipped, "value", value);
                ds_map_replace(obj_player.equipped, "desc", desc);
                ds_map_replace(obj_player.equipped, "bullet_type", bullet_type);
            }
            else
            {
                ds_map_destroy(obj_player.equipped);
                obj_player.equipped = ds_map_create();
            }
            break;
        }
        case item_type.cure:
        {
            obj_player.hp += ds_map_find_value(obj_player.items[| inv_index], "value");
            ds_list_delete(obj_player.items, inv_index);
```

```
51                  }
52              }
53          }
54      }
55  }
```

The preceding code handles the cursor movement through the items in obj_player's items list using the same idea we used to create the various options in the pause menu. To know where the cursor is located in the list, we compare its value to the index of each object in the DS List (note that the cursor loops between 0 and ds_list_size(obj_player.items) as you can see in lines 7–14).

When the enter key (or Face 1 button on the gamepad) is pressed, the currently selected item is selected to be equipped or unequipped by obj_ player, if it's a weapon (lines 20–46), or to be consumed if it's a cure item (lines 47–51).

If the item selected to be equipped is already equipped, we unequip it.

To actually equip an item, we replace every field of the currently equipped item with the fields of the item we want to equip (26–38).

To unequip an item, we just destroy the equipped DS Map and reinitialize it (lines 42–43).

Let's see now how we can show that information in the inventory menu.

Head to obj_controller's Draw GUI event and append this code at its bottom:

```
// --- DRAW INVENTORY --- //
if (inv_open)
{
    if instance_exists(obj_player)
    {
        draw_set_alpha(0.5);
        draw_set_color(c_black);
        draw_rectangle(0, 0, cam_w, cam_h, 0);
    draw_set_alpha(1);

    draw_set_font(fnt_menu_h1);
    draw_set_color(c_white);
    draw_text(200, 100, "ITEMS");

    draw_set_font(fnt_menu_h2);
    draw_set_color(c_white);

    var list_x = 200;
    var list_y = 300;
    var text_offset_x = 500;
    var text_offset_y = 50;

    draw_set_color(c_silver);
    draw_text(list_x, list_y - text_offset_y-20, "Name");
    draw_text(list_x + text_offset_x, list_y - text_
    offset_y-20, "Description");
    draw_text(list_x + text_offset_x * 3, list_y -
    text_offset_y-20, "Stat");

    for (var i = 0; i < ds_list_size(obj_player.items);
    i++)
    {
        draw_set_color(c_white);
        if inv_index == i
        {
            draw_rectangle(list_x - 50, list_y +
            (text_offset_y * i), list_x - 30, list_y +
            (text_offset_y*i) + 20, 0);
        }

```

```
36              if not ds_map_empty(obj_player.equipped)
37              {
38                  if ds_map_find_value(obj_player.items[|i],
                    "key") == ds_map_find_value(obj_player.
                    equipped, "key")
39                  {
40                      draw_set_color(c_yellow);
41                  }
42              }
43              var t_name = ds_map_find_value(obj_player.
                items[|i], "name");
44              var t_desc = ds_map_find_value(obj_player.
                items[|i], "desc");
45              var t_value = ds_map_find_value(obj_player.
                items[|i], "value");
46              draw_text(list_x, list_y + (text_offset_y * i),
                t_name);
47              draw_text(list_x + text_offset_x, list_y +
                (text_offset_y * i), t_desc);
48              draw_text(list_x + text_offset_x * 3, list_y +
                (text_offset_y * i), string(t_value));
49          }
50      }
51  }
```

Just like we did with the other two menus, also for the inventory, we check whether it should be visible or not by relying on the controller variable (line 2); then we check if obj_player exists (so that we can access the inventory) and start drawing the inventory menu.

First, we draw the usual black semi-transparent background (lines 7–9); then we draw the title of the menu (lines 12–15) and the label of each column (lines 17–27) so that we can make the player understand what the information shown means.

Lines 28 to 49 are basically a loop between all the items in obj_player's items DS List. We loop through them one by one; we draw the cursor at the right position, comparing the cursor's index to the item's index in the DS List.

Finally, we check whether the item is equipped by comparing the key values (it's equal to the content of the equipped variable); and if it's the case, we draw its information (name, description, and value) in yellow, or else we draw the information in white.

Ok, now that we have a working inventory and a structure to represent items, we only need to create the actual items to pick up.

Since we want our items to be uniquely identifiable, we need a function to generate a unique key. So, before we start creating items as GML objects, let's create a new script by right-clicking Scripts in the Resources sidebar and call the new script generate_key. Open it up and append this code in it:

```
return room_get_name(room) + object_get_name(object_index) +
string(x) + string(y);
```

That's it. This function will generate a unique id by concatenating the name of the room, the name of the instance that called the script, and its coordinates.

This is called a hash function, as we saw earlier, and it's how you identify and access items inside a map data structure.

It's not easy to create good hash functions, but it's fairly easy to create good enough hash functions.

Now that we have our hash function, let's make our first item by creating a new object named obj_item. Assign no sprite to it and just add a Create event with these lines of code in it:

```
1   key = generate_key();
2
3   data = ds_map_create();
4   ds_map_replace(data, "key", key);
```

Line 1: We create a unique key for the item calling the generate_key function.

Lines 2–3: We create the map that represents the item and add the unique key to it using ds_map_replace.

That's it for the generic item. We will add more detailed information like the name, the type, and other things in the children objects of obj_item.

Anyway, we need to manage one more event for obj_item. We want that when obj_player touches an instance derived from obj_item, it can pick it up by basically copying the data map we just defined in the items DS List.

So let's add a collision event between obj_player and obj_item in obj_item and let's write this code in it:

```
1   ds_list_add(obj_player.items, data);
2   ds_list_mark_as_map(obj_player.items, ds_list_size(obj_
    player.items)-1);
3
4   instance_destroy();
```

When obj_player collides with obj_item or an instance derived from it, the data DS Map belonging to the obj_item instance is added to obj_player's DS List (line 1). We also need to mark that newly added element as a DS Map, since GML needs to know that this is not a normal base type of data, but a data structure (line 2). Finally, since we don't need anymore the instance, we destroy it (line 4).

All the items we are going to create will act like that. So, as you can see, even if they are very different, the items in a game can be treated all the same in an inventory (unless you need to access specific information that only a type of item has).

Let's create our first item derived from obj_item, starting from this template.

Create a new object called obj_upgrade_light. Assign spr_upgrade_light to it and select obj_item as its parent object.

Now add a Create event to obj_upgrade_light and insert the following code in it:

```
1  event_inherited();
2
3  ds_map_replace(data, "name", "Better pistol");
4  ds_map_replace(data, "type", item_type.weapon);
5  ds_map_replace(data, "value", 5);
6  ds_map_replace(data, "desc", "Decent fire power");
7  ds_map_replace(data, "bullet_type", spr_bullet_light);
```

At line 1, we inherit the code from obj_item's Create event, so that we have the key generated.

Then, from lines 3 to 7, we add the various fields to the data DS Map we defined in obj_item.

Since this is a weapon upgrade, we also define a field containing the sprite we want to assign to the bullet (line 7). We will need this information when it will come the time to make Maria shoot with her gun.

Now let's create two more items, just to see our inventory grow up a bit! We will create another weapon upgrade and a cure item.

Let's start by creating a new object called obj_weapon_heavy and set its sprite to spr_upgrade and set obj_item as its parent. Then in obj_weapon_heavy's Create event, insert the following code.

```
event_inherited();

ds_map_replace(data, "name", "Super pistol");
ds_map_replace(data, "type", item_type.weapon);
ds_map_replace(data, "value", 5);
ds_map_replace(data, "desc", "Super fire power");
ds_map_replace(data, "bullet_type", spr_bullet_heavy);
```

Create another object, call it obj_cure_paracetamol, and select obj_item as its parent object.

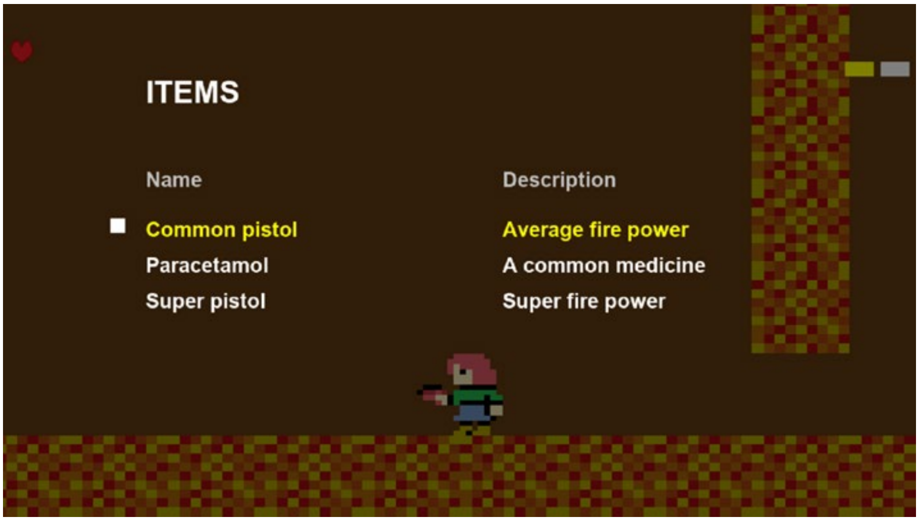
This object is a cure, and it will give the player a bonus HP when used from the inventory. For this object, use the sprite spr_cure and add this code to its Create event:

```
event_inherited();

ds_map_replace(data, "name", "Paracetamol");
ds_map_replace(data, "type", item_type.cure);
ds_map_replace(data, "value", 1);
ds_map_replace(data, "desc", "A common medicine");
```

It's all done! Now just drop an instance for each of those three objects in one of the two rooms and run the game!

You should verify that everything works right. You can see and pick up the objects and see them in your inventory by pressing the inventory key (Figure 12-6). You can even equip them (one at a time) and use the cure to gain a bonus heart! But still, we can't attack or use the equipment yet. Don't worry. We are going to fix this in the next section!


ITEMS

Name	Description
Common pistol	Average fire power
Paracetamol	A common medicine
Super pistol	Super fire power

Figure 12-6. *The inventory screen showing all the items carried by the player. The yellow highlighted item is the equipped item.*

Creating the combat system

We have an inventory that we can fill with cool weapon upgrades, but we can't use them. Let's fix this by implementing the possibility to shoot. We want the player's character to be able to shoot only if they equipped an upgrade for the gun. The upgrade will define the fire power of the gun.

The shooting itself will be very similar to the one we implemented in Space Gala. We will check for the user input and then generate a bullet instance that will start travelling at a certain speed in the direction obj_player is facing.

We will set the bullet's sprite and attack power based on the data retrieved in the item equipped (the value and bullet_type fields in the DS_Map that represents the item, as we saw in the previous section).

Let's start from the bullet object. Create a new object and rename it obj_bullet. Add to this new object a Create event and put the following code in it:

```
1   speed = 30;
2   atk = 0;
3   start_x = x;
4   shoot_range = 200;
```

At line 1, we define the bullet's speed, since we want the bullet to start going forward as soon as it's created.

At line 2, we created the atk variable which serves to the scope of representing the attack power of the bullet. We will update this value when shooting based on the equipped item.

At lines 3 and 4, we save the starting X-coordinate and the maximum range of the bullet. The range of the bullet allows us to make the bullet travel only for a predefined amount of space; then it will be destroyed. We will see it in greater detail in a bit.

Now create obj_bullet's Step event and add the following code in it:

```
1   if abs(start_x - x) > shoot_range
2   {
3       instance_destroy(id, false);
4   }
```

In the preceding code, we calculate the amount of space covered by the bullet; and if it's greater than the maximum range of the gun (line 1), we destroy the bullet.

It's important to give a range to a weapon, to avoid the possibility that the player can shoot and kill enemies from the other side of the level totally eliminating the challenge.

Another solution would have been to destroy the bullet as soon as it reaches the sides of the camera as we did in Space Gala, but in this particular case, the weapon range can be also useful to distinguish the various weapons. In fact, it allows us to think about future improvements where we can define and change the range of a weapon through the equipment.

We want our bullets to be destroyed when they collide with a wall, so let's create a collision event between obj_bullet and obj_event. Add the event to obj_bullet and insert this single line in it:

```
instance_destroy();
```

Now create a Destroy event in obj_bullet and add this single line:

```
effect_create_above(ef_spark, x, y, 0.4, c_yellow);
```

Now, when the bullet hits a wall, it gets destroyed in a particle effect.

The bullet object is complete, for now. Let's now dedicate on the actual shooting.

Open up obj_player's Create event and append these lines at its bottom:

```
// combat
can_attack = true;
atk_speed = 0.3;
```

In the preceding code, we created a couple of variables that we will use to regulate the attack rate. The controller variable can_attack will be switched on and off every 0.3 seconds (the value of atk_speed).

Now, in obj_player's Step event, just under the last input check, add this line to handle the attack key input:

```
var attack = keyboard_check(ord("C")) or gamepad_button_
check(0, gp_face3);
```

To attack, we will use the C key on the keyboard and the Face 3 button on a gamepad (X on an Xbox gamepad or square on a PlayStation gamepad).

Finally, append this code at the bottom of obj_player's Step event:

```
// ATTACK
if attack and can_attack and not ds_map_empty(equipped)
{
    var bullet = instance_create_layer(x, y, "Instances",
    obj_bullet);
5       bullet.direction = direction;
6       bullet.atk = ds_map_find_value(equipped, "value");
7       bullet.sprite_index = ds_map_find_value(equipped,
        "bullet_type");
8
9       can_attack = false;
10      alarm[1] = room_speed * atk_speed;
11
12      audio_play_sound(snd_shoot, 1, false);
13  }
```

The first thing we do (line 2) is to check if the player can attack. The player can attack if they are pressing the attack button, the controller variable can_attack is true (meaning that more than 0.3 seconds are passed after the previous bullet was shot), and the player equipped a weapon item.

Then we create a new instance of obj_bullet (line 4), and we assign the right sprite to it according to the value of the bullet_type field in the equipped item's DS Map (line 7).

We then assign the bullet the same direction of the player's character (which we are updating anytime the character turns left or right) and use the value variable of the equipped item to set the bullet's attack power (lines 5 and 6).

Finally, we set can_attack control variable to false (line 9), and we start the alarm (line 10) which will reset can_attack to true, so that we can attack again after atk_speed seconds (which is 0.3).

We also play a sound effect for the shoot at line 12.

The last step is to create an Alarm 1 event for obj_player and insert this one line:

```
1   can_attack = true;
```

Ok, now it's all ready to be tested! Save and run the game and enjoy the shooting and your new shiny items and inventory system (Figure 12-7).

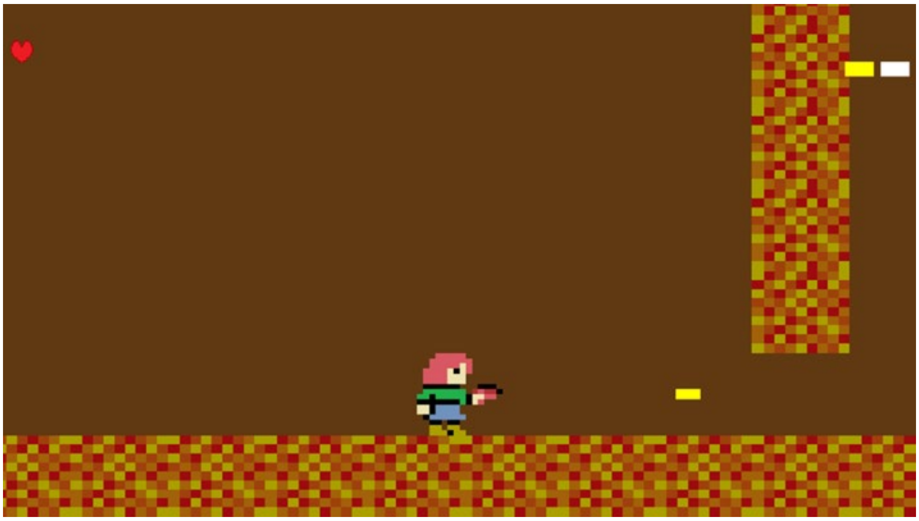


Figure 12-7. *Maria trying her new gun!*

You can now add as many cure and weapon items as you like; and, changing a couple of things, you can also create new kinds of items, for example, armor items giving defense bonuses or items that boost speed or jump height or weapon range extenders as well as new equipment slots. Free your imagination and customize your inventory system as you like!

A combat system is pretty pointless without something to kill. It's now time to add enemies, so that all those bullets don't go wasted!

Old enemies

As we anticipated in the GDD in Chapter 11, in Isolation, we are going to use the same enemy we created for Cherry Caves 2. Why? Well, because creating more interesting enemies is not really the purpose of this chapter, but still we need them to complete our combat system and make our gun be a bit more effective. There's no shame in using our own creations.

In this section, we will create the obj_octopus_green from Cherry Caves 2. We won't cover obj_octopus_purple, since its implementation is pretty straightforward. After creating obj_octopus_green, I want you to try and add it by yourself, as an exercise.

As you probably remember from Chapter 9, obj_octopus_green moves between two markers or blocks. We have the blocks, but we lack the markers. Let's create them!Create a new object called obj_marker, assign spr_marker to it, and uncheck the Visible property from the Object Editor. That was pretty fast, right?

Let's make our enemy object by creating a new object and calling it obj_enemy. This, as we already saw in Chapter 9, will be the parent object that we will use to manage collisions and other events that should affect any enemy.

Leave this object without a sprite and add a Create event containing these two lines of code:

```
hp = 1;
atk = 0;
```

We also need a Step event to check whether the HP of the enemy has dropped to 0. In that case, we want the instance to be destroyed. To do that, append this single line to obj_enemy's Step event:

```
if (hp <= 0) instance_destroy();
```

That's it! Now we just need to create his child: obj_octopus_green.

Create a new object called obj_octopus_green and assign spr_octopus_green to it. Add obj_enemy as the parent of obj_octopus_green.

Add a Create event with this code:

```
hp = 10;
spd = 4;
dir = 1;
```

In the preceding code, we redefine the basic properties of the object by setting the HP, speed, and direction to arbitrary values.

As usual, create the Destroy event and add this single line to it to create a particle effect and play a sound effect when you kill this monster:

```
effect_create_above(ef_firework, x, y, 1, c_purple);
audio_play_sound(snd_kill, 1, false);
```

Finally, add a Step event. We will use this event to manage the movement and the logics of this particular enemy. Add this code in obj_octopus_green's Step event:

```
event_inherited();

if ( global.game_state == states.playing )
{
    if place_meeting(x, y, obj_block) or place_meeting(x,y,
    obj_marker)
    {
        dir *= -1;
        image_xscale = image_xscale *-1;
    }

    x += spd * dir;
}
```

We already saw in Chapter 9 how this enemy works. It walks back and forth between two instances of obj_block or obj_marker. The code is exactly the same.

An enemy is not such if it doesn't hurt you. We have to handle the collision between obj_player and obj_enemy to make this happen.

To handle the damage from obj_enemy, we need to create a couple of variables in obj_player. In fact, we don't just want to take the damage when we touch an enemy; but we also want the player's character to become invincible for a second, just after being hit, and we also want to push them away from the enemy. We achieve the invincibility by using a switch that tells us to deactivate the collision with the enemies, and we jump away by adding a new controller variable that can trigger the jump when switched to true, just like the bouncy platforms in Cherry Caves 2.

So we must define two variables into obj_player's Create event:

```
// damage
invincible = false;
force_jump = false;
```

We will handle the logic in the Step event, as always. So, in obj_player's Step event, add this code:

```
// Flashing when invincible
if invincible
{
    visible = not visible;
}
```

The idea behind the preceding code is that we simulate the blinking of the avatar like in Super Mario Bros., by activating and deactivating the visibility of the sprite constantly until invincible gets set again to true. We will write the logic of when activating and deactivating the variable in the collision event with the obj_enemy (which we are going to cover in a bit).

Now head to obj_player's Step event, in the jump-related section. We want to change this part so that the jump code is executed also when the variable force_jump is true. force_jump is deactivated anytime the jump code is executed, so that we trigger only one jump. So modify that section like this:

```
if force_jump or (jumping and (grounded or wall_jump))
{
    grounded = false;
    vsp = -jspd;
    if wall_jump
    {
        dash_recharging = false;
        effect_create_below(ef_smoke, x, y, 1, c_white);
        facing_dir *= -1;
        image_xscale = facing_dir;
        if (facing_dir > 0) direction = 0;
        else if (facing_dir < 0) direction = 180;
        wj_goal_x = x + 80 * facing_dir;
        already_walljumping = true;
        wall_jump = false;
    }
    obj_player.sprite_index = spr_player_jump;
    audio_play_sound(snd_jump, 1, false);
}
```

Now that we have everything set up, we only need to add the logic we talked about when obj_player collides with obj_enemy. So add a new collision event between obj_player and obj_enemy in obj_player and insert the following code in it:

```
if ( not invincible )
{
    audio_play_sound(snd_hit, 1, false);
    hp--;
    if hp > 0
    {
        force_jump = true;
        invincible = true;
        alarm[2] = room_speed *1;
    }
11      else
12      {
13          audio_play_sound(snd_kill, 1, false);
14          global.game_state = states.gameover;
15          instance_destroy();
16      }
17  }
```

In the code above, we check whether the invincible variable is set or unset (line 1). If it's set, it means that we must ignore the collision with obj_enemy, so we do nothing. If invincible is not set, it means obj_player's instance can be hit; so we play the hit sound (line 3), decrease by one obj_player's HPs (line 4), and check if that value reached zero (line 5). If it did, we play a sound effect to report the death, change the game state to states.gameover, and destroy the instance (lines 13–15); otherwise, if it still has some HP, we force the jump and the invincibility trigger by setting force_jump and invincible both to true, and finally we start a new alarm to stop obj_player to be invincible after 1 second (lines 7–9).

Lastly, we need to create this new Alarm event. Click Add Event ➤ Alarm ➤ Alarm 2 to do it and add these two lines in it:

```
invincible = false;
visible = true;
```

Summarizing, when obj_player hits obj_enemy, obj_player loses one HP; it gets bounced away and starts to blink becoming invincible for 1 second. After that second, it stops blinking and can be hit and damaged again.

Everything is in place for what concerns the damage received, but what about the damage done?

Maria's bullets, even if upgraded, don't actually do much. Let's fix this!

Head to obj_bullet and add a new collision event between obj_bullet and obj_enemy. Inside this event, write these two lines of code:

```
1   other.hp -= atk;
2   instance_destroy();
```

At line 1, we access the enemy's HP, and we decrease the value by the attack power of the bullet (which is increased by the equipment); and at line 2, we destroy the bullet, as usual, when it hits something solid.

Ok, now really everything is in place, and we can finally put an instance of obj_octopus in the room and run the game to test it!

You should be able to take hits by the evil octopus by bouncing into it; and, when you equip a weapon upgrade, you should also be able to shoot it down and kill that abomination once and for all (Figure 12-8)!

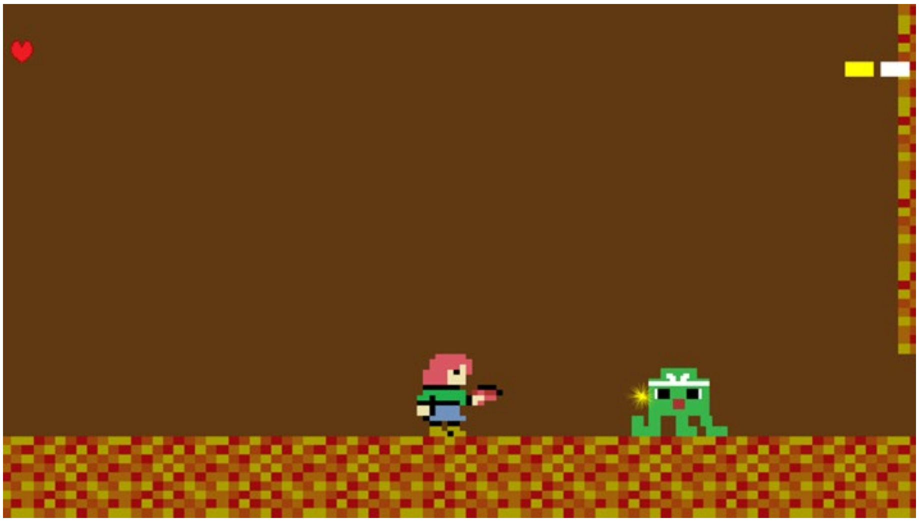


Figure 12-8. *One step away from death (one HP remaining), Maria manages to shoot down the evil octopus and save her life. I guess the shooting is working!*

Saving Maria

I was born in 1990, where games didn't have a save system. To finish a game, you had to play it all at once from start to finish. Those were hard times, and I managed to finish some games only because I was a kid and had a lot of time in my hands.

To surpass the technological limit of the impossibility to save, many games had a lot of secret warp zones that allowed you to jump straight to a more advanced level, so that you could avoid passing 10 hours straight playing.

Times have changed, and now if your game doesn't offer a save feature, no one is going to play it. So let's roll our sleeves up and create our save system.

There are many ways in GameMaker to save the state of the game, but from my experience as a developer, the best way is that which is portable.

GameMaker offers some functions to work with normal text files and a couple of functions to encode and decode data from DS Maps to JSON and from JSON back to DS Maps (namely json_encode and json_decode).

JSON is one of the most popular standards to store and organize data. The name is an acronym for JavaScript Object Notation and was made popular by the JavaScript programming language for being the de facto standard to serialize JavaScript objects on files as an alternative to XML.

JSON is very convenient since it's organized in a key-value fashion, just like our DS Map data structure; and it's just a simple string, so it can be easily parsed from or written to a text file. The simplicity of JSON is its force. In fact, it's the most used and supported data-interchange format, which makes it a very portable solution that can be run and easily parsed on any platform and by any application.

How is this interesting for us? Well, having a save file which is easy to read and parse and is a recognized standard by an endless list of languages means that we can, in the future, create some external software to manipulate the save file, like a modding tool or a level editor, or even manipulate it on the Web using JavaScript. The possibilities are endless; and to use a recognized standard to organize our data, it's always a very good software engineering practice that will surely pay in the long run, since the more a standard is popular, the more it is going to be supported in the future. This means less work for us, less bug hunting, less reengineering.

Moreover, it's pretty easy to work on text files in GMS2. Let's quickly see how to do it!

There are many ways to manage (open, read, write, close) text files in GML, depending on the standard we want to use (e.g., there are dedicated functions for INI files, which are text files following a particular syntax, just like JSON). In our case, we want to use the most generic functions to open plain text files. Let's briefly see some of the most important.

file_text_open_read(fname)

This function opens the text file indicated by fname (which is a string) for reading. It returns a unique id of the opened file so that you can use it to read from the file.

For example:

```
var file = file_text_open_read(filename);
var name = file_text_read_string(file);
file_text_close(file);
```

file_text_open_write(fname)

This function, instead, is used to open the file indicated by the string fname for writing. If the file doesn't exist, GameMaker creates it. If the file does exist, it gets overwritten.

The function returns a unique id of the opened file. Let's see a usage example.

For example:

```
var file = file_text_open_write(filename);
file_text_write_string(file, my_text);
file_text_close(file);
```

file_text_open_append(fname)

Similarly to file_text_open_write, this one is for writing. The difference is that file_text_open_write overwrites the file if it already exists, while this function starts writing from the bottom, appending the data to the content of the file. If the file doesn't exist, it is created.

The function returns the unique id of the opened file to be used to write on the file.

For example:

```
var file = file_text_open_append(filename);
file_text_write_string(file, my_text);
file_text_close(file);
```

file_text_write_string(file_id, my_string)

This function writes a string into a file indicated by the unique identifier file_id.

For example:

```
var file = file_text_open_write(filename);
file_text_write_string(file, "Hello, World!");
file_text_close(file);
```

file_text_close(file_id)

This function closes an open text file indicated by file_id. It's important to always close the files you open, when you finish using them, or else you risk losing information or creating corrupted files.

Now that we made a little tour around some of the most important functions to manage files in GML, let's get back to our game. Don't forget that if you need more information, you may always go online and check the huge GMS2 documentation on the official web site of YoYo Games.

So the idea is to create a DS Map in which we put all the data we need to save (like the status of the inventory, the equipped items, the current room, the player's position, etc.); then, when we want to save the game, we convert the data to JSON and finally write the JSON into a text file. If, instead, we want to load the saved data, we load the content of the save file into a string variable, we convert the JSON string to a DS Map data structure, and we recreate the state of the game based on the content of the saved data.

As you probably guessed, we are going to define that structure in obj_controller. So open it up and head to its Create event and append these two lines:

```
1   game_data = ds_map_create();
2   save_file = "isolation.sav";
```

In the preceding code, we create the data structure that we will use to store the data we want to save (line 1) and define the filename for our save file (line 2). Note that the extension of the file can be whatever you want. I am using .sav just for convenience since it's a commonly used extension to indicate save files.

To load and save data from the save file, we will use some user-defined functions. It's a common task that we may want to do more than once, depending on the saving policy of the game. Some games save every time you enter in a room, some others when you reach a checkpoint, and so on. So, to support all those possibilities without rewriting a lot of code, we define two functions to do the work.

Let's start by creating the save function. Create a new script by right-clicking Scripts in the Resources sidebar and rename the new script save_game. Inside the newly created script, write the following code:

```
ds_map_replace_list(obj_controller.game_data, "player-items", obj_player.items);
ds_map_replace_map(obj_controller.game_data, "player-equipped", obj_player.equipped);
ds_map_replace(obj_controller.game_data, "player-x", obj_player.x);
ds_map_replace(obj_controller.game_data, "player-y", obj_player.y);
```

```
5   ds_map_replace(obj_controller.game_data, "player-hp",
    obj_player.hp);
6   ds_map_replace(obj_controller.game_data, "player-can_dash",
    obj_player.can_dash);
7   ds_map_replace(obj_controller.game_data, "player-can_wall_
    jump", obj_player.can_wall_jump);
8
9   ds_map_replace(obj_controller.game_data, "room", room_get_
    name(room));
10
11  var str_save = json_encode(obj_controller.game_data);
12
13  var file = file_text_open_write(obj_controller.save_file);
14  file_text_write_string(file, str_save);
15  file_text_close(file);
```

Line 1: We save obj_player's items list in the game data grid by using ds_map_replace_list. Since it's not a basic type, we need to specify to GameMaker that we are inserting a data structure inside the DS Map obj_controller.game_data. Not doing it may cause GameMaker to corrupt the data and so make the save file unusable.

Line 2: Similarly to line 1, here we save the equipped item into the DS Map specifying to GameMaker that we are inserting a DS Map into another DS Map. Also here, not specifying this may cause data corruption.

Lines 3–7: We store important variables related to obj_player, like the position in the map, the HPs and if it can wall jump or dash. Nothing really complex here, we are just inserting variables into a DS Map.

Line 9: Here we save the room name into the DS Map. It's important to not use the id, but always the name of the room, since you don't really have the control on the id, while you have it on the name. So it's safer to rely to an information on which you have control.

Line 11: After we updated the DS Map containing the data to save, we convert it to JSON using json_encode and save the resulting string into a variable str_save.

Lines 13–15: Finally, we open the text file using file_text_open_write which takes as argument a string representing the path and filename of the file we want to open, we write the string into the file using file_text_write_string, and lastly we close the file we just opened via file_text_close.

Now let's take care of the load function. Create a new script and rename it load_game; then write the following code:

```
if not file_exists(obj_controller.save_file)
{
    return false;
}

ds_map_destroy(obj_controller.game_data);

var file = file_text_open_read(obj_controller.save_file);
var str_data = file_text_read_string(file);
game_data = json_decode(str_data);
file_text_close(file);

obj_player.items = ds_list_create();
obj_player.equipped = ds_map_create();

obj_player.items = ds_map_find_value(obj_controller.game_data, "player-items");
obj_player.equipped = ds_map_find_value(obj_controller.game_data, "player-equipped");
obj_player.x = ds_map_find_value(obj_controller.game_data, "player-x");
```

```
19  obj_player.y = ds_map_find_value(obj_controller.game_data,
    "player-y");
20  obj_player.hp = ds_map_find_value(obj_controller.game_data,
    "player-hp");
21  obj_player.can_dash = ds_map_find_value(obj_controller.
    game_data, "player-can_dash");
22  obj_player.can_wall_jump = ds_map_find_value(obj_
    controller.game_data, "player-can_wall_jump");
23
24  room_restart();
25  var rm = ds_map_find_value(obj_controller.game_data,
    "room");
26  room_goto(asset_get_index(rm));
```

Lines 1–4: We check if the save file actually exists or not. If it doesn't exist, it means we cannot load data, so we just return false.

Line 6: We prepare to load the data by destroying the actual data. It's a good practice to always clean the data structure before reusing them.

Lines 8–11: As we saw earlier, when I presented you some file management functions, here we open up the save file (line 8), we read its content into a string (line 9), and then we decode that string converting it from JSON to DS Map and assign the result of the conversion to obj_controller.game_data (line 10). Finally, we close the file (line 11).

Lines 13–14: Here we initialize both the inventory (obj_player.items) and the equipped item (obj_player.equipped) so that we can use them again to store the loaded data.

Lines 16–22: Here we load all the player-related data from the save file, and we directly inject them into the appropriate obj_player's properties. Note that while loading we don't need to specify to GameMaker that we are dealing with a DS List and a DS Map at lines 16 and 17. We only have to do it when saving.

Lines 24–26: Finally, we restart the room so that all the instances can read from the new version of the game data; and then we fetch the name of the saved room, and we warp to it.

That's it! Our save and load functions are ready! Now let's use them!

We want to add the possibility to load the game or start a new game from the pause menu. So let's open up obj_controller's Create event and modify the options variable so that it looks like this:

```
options = [ "RESUME", "LOAD GAME", "NEW GAME", "QUIT" ];
```

Now head to the Step event and head to the section in which we check if the pause menu is open and the enter key is pressed and change that section like this:

```
if ( enter_pressed )
{
    switch( menu_index )
    {
        case 0: // resume
            global.game_state = states.playing;
            menu_open = false;
            can_pause = true;
            break;
        case 1: // load game
            load_game();
            break;
        case 2: // new game
            game_restart();
            break;
        case 3: // quit
            game_end();
            break;
    }
}
```

When the player presses the Load Game option, the saved game, if present, is loaded; and when they press the New Game option, the game is restarted and reinitialized.

That's great! Now we just need a checkpoint that saves the game for us when activated.

Create a new object named obj_checkpoint and assign spr_checkpoint_inactive to it.

Add to this new object a Create event with this single line in it:

```
already_saved = false;
```

We will use this variable, already_save, to switch on and off the saving feature. We need this because we want the checkpoint object to save the game when obj_player collides with it, but we don't want it to continuously save while obj_player is colliding with it. So we use already_saved to regulate when it's possible to save and when it's not.

Add a Step event to obj_checkpoint and write this code in it:

```
if place_meeting(x,y, obj_player)
{
    if already_saved exit;
    sprite_index = spr_checkpoint_active;
    already_saved = true;

    save_game();
}
else
{
    sprite_index = spr_checkpoint_inactive;
    already_saved = false;
}
```

At line 1, we check if obj_checkpoint is colliding with obj_player; if it is, we check whether we already saved or not, with the help of the already_saved variable; and if we already saved, we stop executing the code (line 3).

If we haven't saved already, we change the sprite of obj_checkpoint to show the player that we are saving (line 4), then we switch on the already_saved variable (line 5), and finally we save the game (line 7).

If obj_checkpoint is not colliding with the player, its sprite gets set to spr_checkpoint_inactive, and the already_saved variable is switched off so that we can save when we move the avatar upon the checkpoint instance.

Finally, we want that the items picked up won't show up again when we reload the saved game. So we need to save some information about the single items in the game data and check them when the instances are created in the room.

Let's start by creating a key-value pair into the game data when we pick up the object (the collision event between obj_item and obj_player). All we need to do is to add a record with the unique id of the item and a flag that lets us know that the item shouldn't be shown in the room.

So let's open up obj_item's Step event and add this line of code just before the instance_destroy function call:

```
ds_map_replace(obj_controller.game_data, ds_map_find_value(data, "key"), false);
```

Now, we need to read this value, if it exists, when the instance is created so that we can decide if we want to show the instance or destroy it.

Open up obj_item's Create event and append this code at its bottom:

```
var data_item_exists = ds_map_find_value(obj_controller.game_data, key);
if (not is_undefined(data_item_exists))
{
    if (not data_item_exists) instance_destroy(id, false);
}
```

At line 1, we try to load the data about the instance by using the unique key inside the game data. Then we check if the returned value is legit (line 2); and if it is, we check if its value is false; and, in that case, we must destroy the instance at once, so that it won't be available in the room (line 4).

It's that simple! Now drop an instance of obj_checkpoint into one of the rooms and then save and run the game and check that everything works as expected.

You should be able to save the game touching the checkpoint instance and load it via the pause menu keeping all the information on the room, the position of the player, and the items. You should also be able to create a new game via the pause menu (Figure 12-9).

That's great! We have a fully working saving system! Now our game has all the technical features to be a good metroidvania!

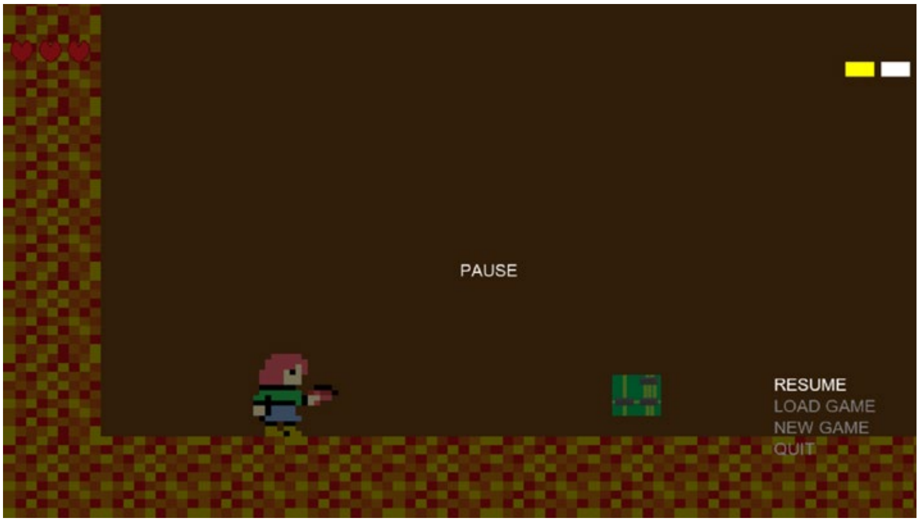


Figure 12-9. *The pause menu now features a Load Game and a New Game option*

Conclusion

It was a long run from Chapters 11 to 12. We managed to create a fully featured metroidvania game system. Our game has all the elements that a good metroidvania needs: tight platforming controls, exploration skills like wall jump and dash, a full and a Minimap, an inventory, equipment, consumables, customizable combat system, and a nice saving system based on checkpoints. From there, the possibilities are endless!

The first thing you need to do to improve Isolation is to design some more nice levels, maybe using the tiling technique we saw in Chapter 9! Then you may want to deactivate some skills only to activate them after a certain point in your game. This is easy to accomplish, thanks to obj_player.can_wall_jump and obj_player.can_dash!

Moreover, you can add more types of items to enhance the customization of the player's character! The possibilities are truly endless, now that you have a powerful game system easily customizable!

If you're confused about how to design a good game or where to start to make this project more than just a prototype, don't worry! In the next chapter, we will take on this topic analyzing great games and discussing how to create a good and fun video game!

TEST YOUR KNOWLEDGE!

1. What is a DS Grid? How can it be used to implement a map?
2. What is a race condition?
3. Can you briefly describe how our full and Minimaps work?
4. Which technique can be used to solve a race condition?
5. Why is it a good idea to manage an inventory with data structures?

6. Can you describe the inventory and item system we developed?
7. Create another weapon item.
8. Create another cure item.
9. Add a new *boost* item category and a new dedicated equipment slot.
10. Create a boost item that increments the moving speed when equipped.
11. Add a new *power-up* consumable item category (similar to the cure item type).
12. Create a power-up item which increases the player's attack power for 2 seconds, when consumed.
13. When the player's character gets hit, it becomes invincible for a while. Why is that a good idea? How did we achieve it?
14. Implement a second type of enemy into Isolation: the purple octopus we created in Chapter 9.
15. How can you write data to a text file?
16. How can you read data from a text file?
17. Why is it important to use a recognized standard to export data?
18. Why are we using JSON to save data in this game?
19. How can you convert DS Map data into JSON data?
20. How can you convert JSON data to DS Map data?
21. Can you describe briefly how our saving system works?
22. Add the auto-save feature when you change room.

CHAPTER 13

Designing Fun Games

In our journey into game development, we explored a small part of video games history and genres.

We created some interesting concepts and prototypes, implemented nice features, and studied game mechanics. We thought about how to make things in the proper way to respect the genre standards and the player's expectations, and you learned some useful game design principles aiming at fun and entertainment. But those were very genre-specific concepts. How can we add the fun to all our games, regardless of the genre? What are the good questions we should ask ourselves? What are the right paths to follow to create a good game design?

Let's talk about this!

Document your design!

The first tool that can help you to ask yourself the right questions is the game design document.

I stressed on the importance of writing a design document for every single game we created for a reason: to write a design document means to think about what your game should be and which direction you should follow.

A design document is a way to keep the project organized and keeps you focused on the things you did and those you should do. A GDD constantly tracks the progresses of your game, and it represents a report

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_13

on the evolution of the design choices. For this reason, I strongly suggest you to use Git or a similar versioning software to keep your GDD updated, so that you can get access to all the previous versions and have a better understanding of the direction your game is taking and what didn't work or worked well.

While you design and describe your idea in the document, features, potential issues, and gameplay elements start to pop up allowing you (and your team) to experiment on them trying to find a balance that will eventually (hopefully) converge to the final version of your game.

Experimenting is a huge content creator mechanism for your games, and it's done by prototyping new features into your game and letting them be tested by a game tester. The elaboration of the feedback you get from the game testing lets you either commit the changes (if the feedback was positive) or trash them and start over (if the feedback was negative).

A negative feedback tells you that your game is not delivering a fun experience to the player for some reason. Your first job is to understand why your game is failing at fun.

The reasons for a bad feedback can be extremely various. Maybe the player felt frustrated because the level design failed at teaching the mechanics or didn't allow enough time for them to train their skills; maybe there are some technical problems, like bugs, glitches, or random crashes or things like badly designed hit boxes (collision masks in GameMaker) that make the experience unfair or inconsistent; maybe the controls are sloppy and the player doesn't get the right feedback related to their input, making them feel like they can't properly play the game; or maybe a challenge is just too difficult and unforgiving ending up in frustration. The possibilities are pretty much endless. To be able to debug your design by understanding what is making the player feel frustrated is a crucial skill to design good games, and it's what makes the difference between a good and bad game design.

Writing a good game design document and taking into account feedbacks from playtesting are great ways to study your game and make your design better, but there are other things you can do to ensure that your game is fun. For example, the way you treat your own game makes a great difference on how your game is perceived by the player.

Respect your game

A fun game is a good game. And a good game is a game that respects its own nature.

The first thing you should point out in your GDD is the purpose of your game. What's your game about? What's its audience?

Every game has a precise purpose and audience, and those two things should be the main pillars to give a direction to your game design. If you fail at delivering the right experience to your audience or reaching the purpose of your game, the whole project will be perceived as a bad game.

Think about Euro Truck Simulator 2 (SCS Software, 2012); it's a game about driving trucks across Europe to deliver goods. You are asked to respect the traffic laws, get gas, sleep when you're tired, and all the things that a truck driver does. Why is it so successful? What's so fun in driving trucks all day respecting the rules like in real life? Well, actually, the fact that it's just like a *real-life* experience makes it fun. The game takes itself seriously, and its main activity is not trying to be something else or a parody of truck-driving. It's exactly and only truck-driving. It's respecting the idea of being a truck-driving simulator, and it's doing it at its best.

Respecting a game means giving depth to the gameplay and dignity to the activity of playing. If a game can't be so brave to respect its fundamental idea and be faithful to its own nature, there's very little chance that someone is going to play it and like it.

That's also why some hybrid games with an interesting concept end up to be a failure. They don't succeed at properly mixing the multiple concepts that compose them because they can't deliver enough dignity to each of those components.

Keep your player immersed

Immersion is that feeling that you experience when you play and forget it's just a game. You feel the importance of every action and fear the consequences, and every need becomes real.

That's one of the most wanted features in a game, but how can we trigger such a condition in our games?

As we saw in Chapters 9 and 10, there are many different game mechanics and elements that can be responsible for the player's engagement and immersion, from game controls to the fairness of the challenges; and they strongly depend on the game genre to which your game belongs. But are there any general rules that we can follow to create engagement and immerse the player in the flow of the game?

The answer is fortunately yes. Thanks to the psychological theory of self-awareness, we know that engagement can be triggered in an actor by satisfying three psychological needs of theirs:

- Autonomy
- Competence
- Relatedness

Autonomy

Autonomy is the perception of being in charge of your own actions. Players need to feel free to act as they want or at least have the illusion that they can decide what to do, how, and in which order. A game in which the player must follow a lead all the time without making a single decision is more like a passive pastime, like watching a movie or reading a book; and, even if those are wonderful ways to spend your time, it's not what a player wants from a game, and so it ends up in a decrease of the player's interest.

An incredibly immersive game that bets everything on autonomy is The Elder Scrolls V: Skyrim (Bethesda, 2011). This game is one of the most immersive games ever made and features a huge fantasy world with so many content that even after hundreds of hours, you keep finding new books, pieces of lore, and side quests.

Skyrim makes the biggest effort in offering a totally custom experience. When you first start the game, you are asked to customize your character, which can be done in great detail. The avatar creation is the first hint to the degree of freedom you get from Skyrim. People spend hours customizing their characters, and that makes them feel happy, because the more detailed is the character builder, the more accurate and believable is their own representation in a virtual world.

Character customization is a particularly strong concept in role-playing games because in this kind of games, the player has to impersonate a role and doing it in a body that you don't resonate with is not so easy. Deciding who you are in a role-playing game from an aesthetical point of view automatically creates in your head a preliminary background of your character. This is very powerful to put the player in the right mood to immerse in the game.

After the character building and a small narrative session which leads to a light non-binding decision, you are thrown in the middle of a huge world with just a direction suggested to you by an NPC and the freedom to decide where to go and what to do. No one ever tells you which quest you are supposed to complete or what kind of behavior you should keep. You're free to do whatever you want in the order you like. Moreover, your actions have a meaning inside the game world, which gives Skyrim credibility; and, even if not technically perfect, that makes it feel like a real place.

Another game that does a great job in implementing autonomy is Dark Souls. It allows the player to play the game the way they want, with their own style by making available a lot of different equipment that can be combined to create many different builds. Moreover, even if the game gives some sort of direction, it allows the player to decide whether to follow that direction or face challenges in another order. That's a perk that is particularly interesting when restarting the game or in the NG+. This kind of freedom gives not only autonomy but also replayability.

To summarize, freedom of action, customization, and knowing that actions have a meaning in the game world are powerful ideas to foster autonomy into your game; and you should take them into account to understand how to implement them into your specific game genre and design idea.

Competence

Competence is the ability of the player to successfully complete an activity. Every player needs to know if they are doing well or not and to have a continuous feedback that allows them to become better at what they are doing and possibly understand how to optimize their actions to get the best outcome with the minimum effort.

To communicate to the player how they are performing, different game genres implement different feedback systems. The player's competence can be measured and shown with many different game design patterns. For example, a lot of platformer games have a score system. Score gives the player a granular feedback, which means that the player earns points for every positive action giving the player an immediate and clear feedback about what's good and what's wrong. The continuous growth of the score tells the player that they are progressing and keeps them from being distracted from the main activity. In fact, the player will always look for a feedback from the game; and if it's hard to understand or detect, the player gets distracted from the game switching their focus to the search for a feedback to their actions.

Some games, like strategy games, have a different feedback system because in that kind of games, actions are more complex and often composed by a number of smaller actions. Just think about how you manage an empire in Victoria 2 (Paradox Interactive, 2010): you must monitor the empire expenses and the income and think how to maximize it, build a great army and a powerful fleet to defend your country without going bankrupt, make sure that your regions are doing well, take care of research and policies, make sure that the people are happy to avoid revolts and keep good diplomatic relations with the other countries, and so on.

To do all those things, a score is not enough anymore. You need graphs, numbers, and a bunch of colored maps that give you political, geographical, economic, and social information on your territory. Every one of these feedbacks represents a single problem that the player should resolve mastering some specific management skill. In particular, graphs are called sustained feedbacks and tell the player how they performed overtime, which is critical to understand the effectiveness of their actions giving them the ability to understand how to improve and become competent.

Other than granular (score) and sustained feedbacks (graphs), there is a third kind called cumulative feedback. In this category fall badges, achievements, leaderboards, and all those feedbacks that give the player a measure of the overall level of competence that the player acquired by comparing them with other players or just showing them how many important things they achieved.

A classic example of a cumulative feedback is Gym Badges in Pokémon games. They can be acquired by defeating a Gym Leader, who is usually specialized in a discipline or a Pokémon type and tests you on a particular concept or game mechanic. Every Gym Badge is earned after learning a lesson and successfully passing the final test, and this gives the player a concrete measure of their overall skills and competence at the game culminating with their eventual victory at the Pokémon League.

Relatedness

The last of the three psychological needs, relatedness, is about the player having the impression to be part of the group and that their action can actually have an impact on NPCs, other players, and the game world. It's a mixture of sense of relevance and bonding with the other characters or just relating with the game world or the lore and story.

Relatedness is a very powerful concept that can be leveraged with a coherent world design and meaningful stories. A meaningful story creates a narrative overlay on some concepts or game mechanics. It's able to deliver a message by using a narrative expedient that keeps the player interested in what's going on in the game world and to the characters that live in it. To leverage on relatedness, it's important that the player can have some impact on other characters' stories and possibly also on the world itself. Making the difference makes you feel involved in the story keeping you glued to the game to try to realize your own happy ending or experimenting new things to see what could have happened if you took a different decision.

A game that does a wonderful job in this is Undertale (Toby Fox, 2015). People related so much to the game world and its characters that nearly everyone who played the game finished it three times to get all the three endings. This is a huge accomplishment that proves the importance and the power of relatedness in fostering engagement and motivation.

Having fun means learning

All those ideas are worthless without the concept of learning. In fact, the secret of a fun game is that it constantly teaches you something, continuing to give you evocative stimuli every time you play it. The object of learning can be a new concept or just a new degree of competence to the game activity; the important thing is to keep the progression and never stale. If your game stops to simulate the player, they will lose interest and drop it.

So the trick is to combine the theory of self-determination with the theory of fun by learning. To create a fun and engaging game, we should design it so that it satisfies the player's psychological needs (autonomy, competence, and relatedness) while keeping the player learning.

Autonomy can be combined with learning so that the freedom of action given can be used to let the player experiment new paths and combine new ideas. A master example of this is Minecraft (Mojang, 2011), which successfully combines freedom of action (which is absolutely meaningful to the world) to continuous learning. In fact, it allows the player to constantly experiment by combining tiles and creating everything the player can think of.

The measure of **competence** can be used to let the player improve and get a better understanding of the game world and rules. So the measure of competence is particularly effective in practicing and testing some knowledge and skills, which is learning how to optimize the actions to get the best outcome.

Finally, **relatedness** can be a huge leverage to player's learning. In fact, the relationship developed with the NPCs, for example, can be a massive motivator to accomplish some particular tasks that teach you something. Moreover, both NPCs and the game world's story can entertain the player with a good amount of narrative and lore which can give the player some deeper knowledge about game rules allowing them to master some concepts or just to learn new lore that can foster immersion and consequently engagement.

As we saw in Cherry Caves 1 and 2, level design can be a very powerful tool to teach mechanics and stories alike. Games like Dark Souls, Hollow Knight, and DOOM do a wonderful job in delivering knowledge about the world just by using environmental storytelling, which is a technique that takes advantage of the environment in which the player is immersed to tell them a story. Just think about how clear is the decadence in Dark Souls' settings. Every area is in ruins, and you can clearly understand what it was like in the past. It's easy to imagine the greatness of Izalith at its apogee or the frantic activity in the Undead Burg streets. It's easy even if you never saw it; you're learning the history of the game world just by looking at it. That's the magic of environmental storytelling.

Environmental storytelling can give a huge boost to relatedness by defining the setting and keeping the player into the right mood. Just think about the constant sense of isolation of Super Metroid's caves or the deep sense of decadence of the areas of Hollow Knight. Those two games successfully manage to transmit the right mood to the player, which, as a consequence, feels deeply immersed in the story.

In the final part of Chapter 9, we saw how to create a first level that can introduce the player to the game mechanics by thinking how they will probably face the challenges and giving them enough room to practice new skills and enough occasions to understand how to move in the world. It's vital, for your game, to have a level design that takes those concepts into account by teaching the player how to play the game little by little without putting too much or too little pressure. Too much pressure leads toward frustration, while too little pressure leads toward boredom. *In medio stat virtus* (virtue stands in the middle), as the Romans used to say. Designing levels that aim to teach skills and game rules to the player allows you to satisfy the need of competence of the player. In fact, the level itself becomes a feedback machine that tells if you are doing well or not with a binary feedback: if you proceed, you are doing things right; if you are stuck, you are doing something wrong.

Creating a good and fun game is not an easy task, and to become a good game designer, you have to make games – a lot of them! Practice is key, but knowing where to go and what to look for can make a huge difference between being stuck in bad design decisions and knowing how to fix your game.

Conclusion

It was a long run, but you learned so many things that now you are able to both design and develop a full game all by yourself! And if it wasn't enough, you also created a bunch of game prototypes and demos for your portfolio. From there, you can also create new projects or extend some of the games we made together and give your own unique contribution to the indie video game scene. But to do that, you should learn how to publish your game to make it available to everyone! And that's what I'm going to teach you in the next chapter! You will learn how to publish your game on some of the most popular stores and platforms to make it available to the world!

CHAPTER 14

What's Next?

This was a wonderful journey, wasn't it? You created a lot of games, studied video games history, and learned the rules of good game design and even some software engineering best practice. You officially stepped into game development, and you're ready to become the next Will Wright or John Romero (depending on if you want to make games about creating or taking lives). How do you do that? How can you bring your games to the people?

In this chapter, we are going to talk about the most interesting game publishing platforms analyzing pros and cons to sell or freely distribute your game to the world and officially enter the video games industry.

ITCH.IO

The first game distribution platform I want to introduce to you is Itch.

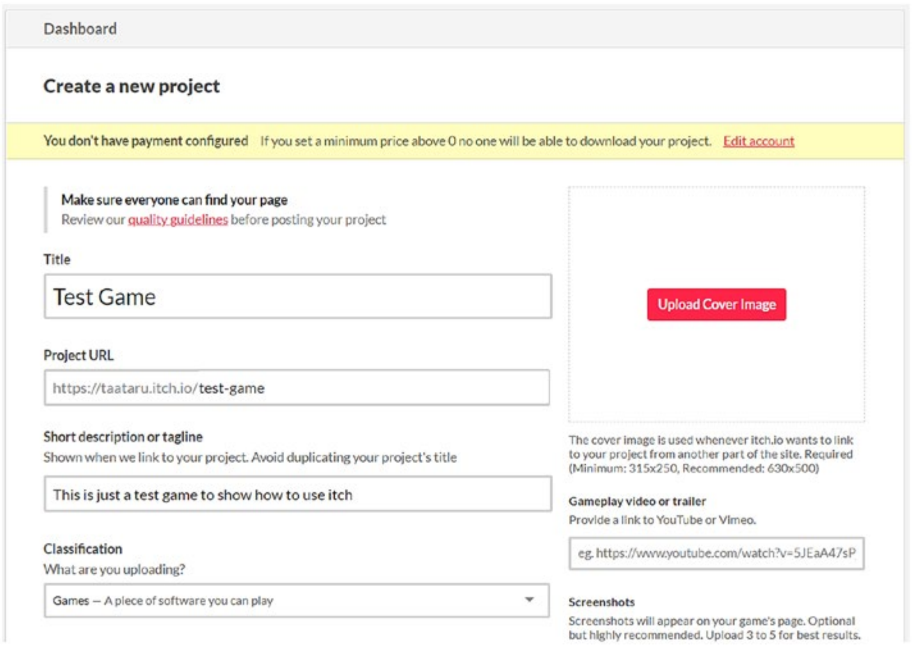
Itch is an open marketplace specifically designed for independent digital creators focusing on indie game developers. It's a very interesting option to sell your game, especially if you're just starting, because there are no requirements to get your game approved. You design your store page, decide your price, and upload it using their dedicated tool. You can also decide to give your game for free or with a pay-what-you-want policy allowing free donations to support your project.

S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5_14

Itch allows you to also decide the revenue split between you (the seller) and them. You can even set it to 0% and get the entire amount paid by your customers, if you like.

To upload your game, you first need to register it as a new project on Itch (Figure 14-1). To do that, you need to log in to your Itch account and click the arrow beside your username in the top-right corner of the web page and select Upload New Project.

This will open up the page to register your game on Itch and create its store page. From here, you can add all the information needed on your project, from the name to the kind of project to the monetization settings or the release status.


Dashboard

Create a new project

You don't have payment configured If you set a minimum price above 0 no one will be able to download your project. Edit account

Make sure everyone can find your page
Review our quality guidelines before posting your project

Title
Test Game

Project URL
https://taataru.itch.io/test-game

Short description or tagline
Shown when we link to your project. Avoid duplicating your project's title
This is just a test game to show how to use itch

Classification
What are you uploading?
Games — A piece of software you can play

Upload Cover Image

The cover image is used whenever itch.io wants to link to your project from another part of the site. Required (Minimum: 315x250, Recommended: 630x500)

Gameplay video or trailer
Provide a link to YouTube or Vimeo.
eg. https://www.youtube.com/watch?v=5JEaA47sP

Screenshots
Screenshots will appear on your game's page. Optional but highly recommended. Upload 3 to 5 for best results.

Figure 14-1. *At this page, you can register your game on Itch and add a lot of useful details*

From this page, you can also upload the game accessing your disk or linking a Dropbox folder with the option "Choose from Dropbox." Optionally, as Itch itself suggests in the documentation, you can upload your game using Butler, which is Itch's own command-line tool to manage your games.

You can download Butler from https://fasterthanlime.itch.io/butler. If you already have the Itch app installed on your PC, you don't need to manually download and install Butler, since it's already included in the app.

The next thing to do is to add Butler to the Windows' Path, so you can use it directly from the command line. To do it, just open up the Advanced System Settings window (press the Windows button on your keyboard or open the Start menu and write "View Advanced System Settings") and click "Environment Variables." Then select the Path string and click Edit. In the new view, click Add to add the path to the butler executable. If you installed it with the Itch app, you can find the butler executable in

```
%appdata%\itch\broth\butler\versions\<itch version>
```

Now that you have Butler set up, you can use it from the terminal (cmd.exe on Windows). The command to upload your game with Butler is

```
butler push your-game-folder your-username/your-game-
name:platform
```

where

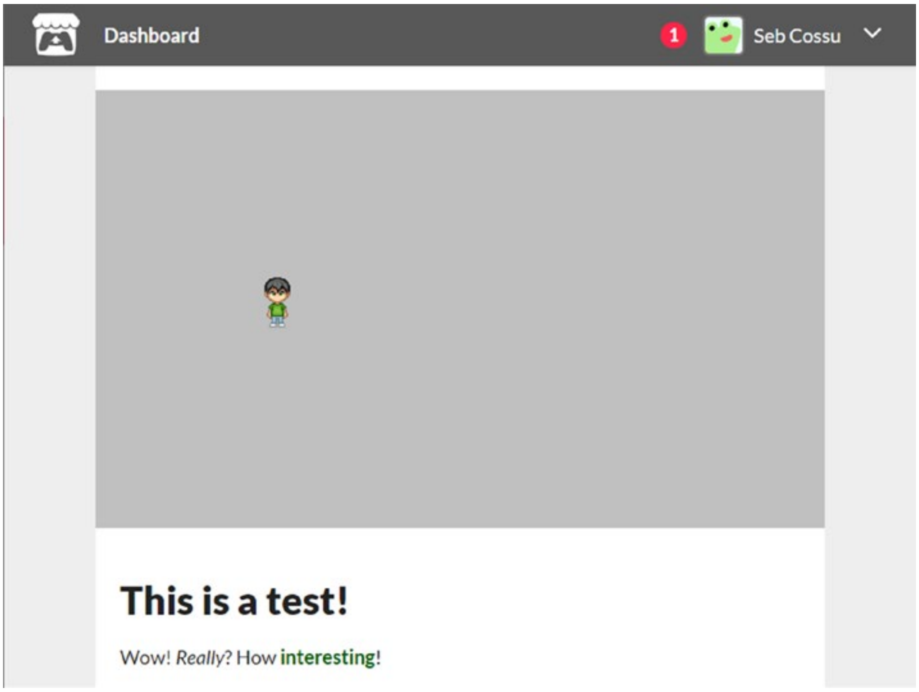
- your-game-folder is the folder where your game package is stored.
- your-username is your username on Itch.
- your-game-name is the name of your game project on Itch.

- platform is the platform compatible with this build of the game (Windows, Linux, Mac, Android). You can specify more than one platform by concatenating them with a dash like this: win-linux-mac. Note that if you use an understandable way to tag a platform (like win or windows for Windows and osx or mac for Mac), Itch will automatically tag the game with the icon of the platform specified.

An interesting feature of Itch is that it's a HTML5-first platform. This means that you can upload your HTML5 game (built with GameMaker Studio 2 Web) on Itch and it will be playable online right from the game's store page (Figure 14-2).

That's all you need to know to start publishing your games on Itch. If you want more information on Itch, there is a very complete documentation covering any topic you may want to deepen at `https://itch.io/docs/creators/getting-started`.

I strongly suggest you to join Itch since it's very easy to get started with it and it features a huge community of indie developers and indie game lovers which is wonderfully supportive to new projects and developers. Itch may be your best first option if you are an indie developer and this is your first experience.



This is a test!

Wow! *Really?* How **interesting!**

Figure 14-2. *You can upload and run HTML5 games on Itch!*

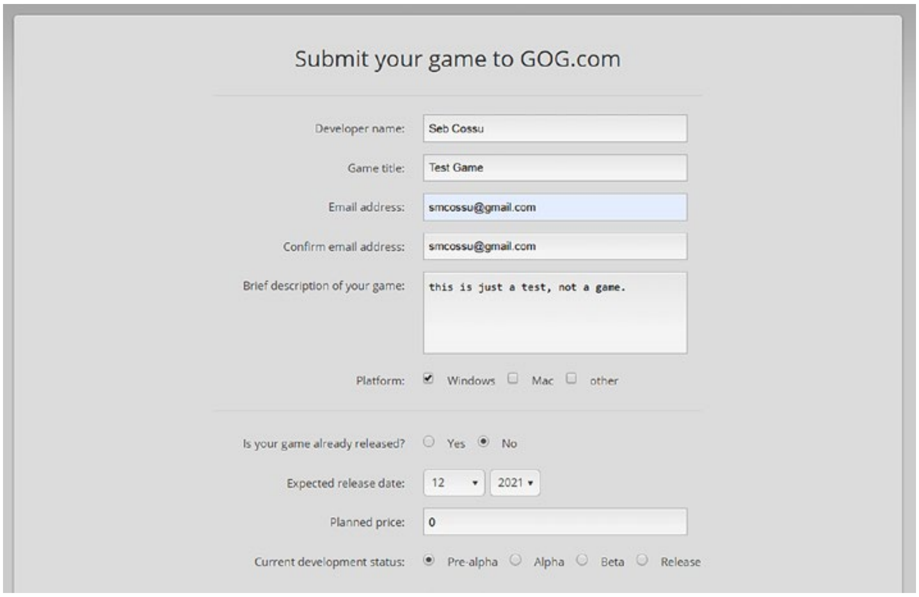
GOG

GOG stands for Good Old Games. It's a platform created by CD Projekt Red, the studio behind the award-winning RPG saga The Witcher. The original purpose of GOG was to bring back old masterpieces from the past and distribute them DRM-free. Before GOG, most of those games were unlicensed and could often be freely downloaded from abandonware web sites (web sites that collected all those old unlicensed games abandoned from their developers) and be run with an emulator.

Thanks to GOG, you can now just buy those games and play them without the hassle of working with emulators; they are ready to be run as soon as you download them from the web site or the GOG Galaxy app.

Today GOG is not just a store selling old games, but it has become a store that only sells DRM-free games with a huge catalogue of old and new games. It's particularly popular between indie game developers because of the love and interest they show to the category, just like Itch. Anyway, very differently from how Itch works, you don't have full control. You have to make a request for your game to be accepted at `www.gog.com/indie` and wait for their reply in which they will give you all the information you need for the next steps. At that page, you can specify all your game's details and the release date you plan for your game (Figure 14-3). You can even specify if you plan to make DLCs or to add microtransactions.

The good and the bad thing about GOG is that everything is made by humans, so expect to wait some time before you get a reply. They are very popular, and they have very little automation in their working pipeline. Nevertheless, it's a very good option especially because they are very supportive of indie game developers.



Submit your game to GOG.com

Developer name:	Seb Cossu
Game title:	Test Game
Email address:	smcossu@gmail.com
Confirm email address:	smcossu@gmail.com
Brief description of your game:	this is just a test, not a game.
Platform:	[✓ Windows] Mac other
Is your game already released?	Yes [✓ No]
Expected release date:	12 2021
Planned price:	0
Current development status:	[✓ Pre-alpha] Alpha Beta Release

Figure 14-3. To upload your game to GOG, you have to register it on that form and wait for their reply

Humble Store

Humble Store was born from the success of Humble Bundle, which is a web site that offers bundles of games (but also software and e-books) with a pay-what-you-want policy. Thanks to its great success, it evolved including a full store called Humble Store, and now it also features a monthly subscription called Humble Monthly which offers a selection of games for a fixed price of $12.

You can register your game to be included in a bundle or in the monthly bundle or just to be published on the store. For each of these requests, there is a specific form to fill, and you can find links and information about that at `https://support.humblebundle.com`. After submitting the form, you have to wait for their response, and they will instruct you about next steps.

In particular, to publish your game on Humble Store, you have to fill the form at `www.humblebundle.com/developer/store/application` and wait for them to reply for your submission. You may provide all the relevant information about your game like the type of product you are selling (Base Game, DLC, Collection), the status of the game, social links, and so on.

The Humble Store submission form requires a very detailed list of your game's characteristics which they need to consider if the product suits their standards.

Humble Store allows you also to add a widget on your web site to allow your customers to directly buy your game through them. You can find more details on that at `www.humblebundle.com/developer/widget`.

Humble Bundle is not just a web site selling bundles and games, but it's also a game producer. You can apply and get your game published by them filling the form at `www.humblebundle.com/publishing`. In the years, Humble Bundle produced a huge number of successful games like *A Hat in Time*, *Cultist Simulator*, *Staxel*, *Wizard of Legend*, and many more. Being published by Humble Bundle, you keep the ownership on your IP, a huge pool of customers, a community of Twitch streamers and YouTubers partnering with Humble Bundle who can help your game to become popular, and a marketing team to help you out marketing your game; and more importantly, you will be paid upfront by Humble Bundle as a support to the development of your game.

This is a very interesting and convenient option for indie developers who want to avoid the financial hassle and work with a flexible publisher that respects the creativity of indie developers!

Steam

Who doesn't know Steam? Valve's video games store is so popular that it has become synonym with PC gaming. For understandable reasons, it's also the greatest ambition for PC game developers. They know it, and in fact they ask you to pay a fee of $100 to publish your game on Steam. Not very indie-friendly, right?

Steam Direct is the heir of Greenlight, and it's way more complex (detailed information can be found at `https://partner.steamgames.com/doc/gettingstarted`). It requires you to create a new Steam account dedicated solely to game publishing and to register it for Steam Direct access. You can apply for Direct access at `https://partner.steamgames.com/steamdirect`, where you will be asked to provide all your details and to pay a join fee of $100. Note that to register to Steam Direct, you need to have a registered company or to be registered as a sole trader.

After being granted the access to Steam Direct, you can download the Steamwork SDK to prepare your game to Steam publishing. You can find detailed information and video tutorials on how to use this SDK at `https://partner.steamgames.com/doc/sdk` and `https://partner.steamgames.com/doc/sdk/uploading`.

This is not the easiest nor the cheaper option to publish your game, but it's surely a very popular store and the most important PC game distribution platform counting 90 million monthly active users worldwide (as Valve itself declared at the end of 2018); and its popularity is always growing, despite the diffusion of alternatives like Humble Store, GOG, Itch, and so on.

Anyway, I suggest you to not dive into Steam publishing at your first project. Maybe it's better to use the indie-friendly alternatives, like Itch, GOG, and Humble Store (especially if you need a publisher) first, and then include also Steam, when your business is more stable.

End game

Being a game developer is not an easy quest. You work an insane number of hours a day to get the work done; and, if you don't have a publisher or a crowdfunding, you don't even see money until release.

Being a game developer is about being a dreamer who loves their own work and fantasizes about bringing emotions and amusement to the world with their creations.

This one can be a very tough road to walk on, and you may want to give up so many times. Your will is going to be tested constantly, and it's very likely that you will face failure in your first experiences. This must not stop you! When times are hard, always remember why you are doing this, why you want to become a game developer, and what you want to bring to the world. Let this be your single motivation to keep on working! Just think about the experience with this book: you came knowing nothing about game development, and you ended up creating six full-featured games! If you did this, you can do amazing things!

I wish you all the best with your career, and I hope that you will create amazing games!

Good luck!

Index

A

Action-adventure games, 439
all_cards_paired() function, 109
Array, 54
audio_play_sound function, 181
audio_stop_sound
    function, 182
Autonomy, 233, 502–504

B

Bosses, designing
  autonomy, 233
  behavior, 239
  Ceaseless
    discharge, 233, 234
  defined, 231
  Draygon, 232, 233
  fun boss fight, 239
  game mechanics, 232
  motivation
    immersion, 238
    Mother brain, 236
    Undertale, 237
  outsmarting, 232
  Space Gala, 239
  teaching and experimenting,
    234, 235
Boss fighting
  boss, growth, 225
  collision, 226
  Destroy event, 226
  Inherit Event, 224
  obj_boss object, 223
  shoot ‘em ups, 223
  spr_boss, 223
  UFO spawn, 225, 226
Butler, 513

C

Cameras and viewports
  boundaries, 189
  camera properties, 184
  move up and down, 188
  obj_camera, 184
  obj_controller, 191, 193
  obj_enemy_red’s Alarm 0
    event, 189
  obj_player, 188
  player’s speed, 187
  point_in_rectangle, 191
  scrolling feature, 194
  Space Gala’s level 2, 183
  stop scrolling, 185, 186
  victory conditions, 191
  viewport properties, 184

© Sebastiano M. Cossu 2019
S. M. Cossu, *Game Development with GameMaker Studio 2*,
https://doi.org/10.1007/978-1-4842-5010-5

Cards
  actions, 52
  implementation
    array, 54–57
    if-then-else, 57, 58
    switch, 58–60, 62, 63
  obj_card game object, 53
  properties, 52
Cartesian coordinate systems (CCSs), 441
Character customization, 503
Cherry Caves game
  controls, 246, 300
  enemies, 246, 301
  fonts, 308
  gameplay, 245, 298
  items, 299, 300
  miscellaneous, 302
  settings, 245
  sounds, 308, 309
  squash 'em all!
    Alarm 0, 352
    collision, 349
    Destroy event, 349, 352
    obj_block, 348
    obj_octopus_green, 348, 349
    obj_octopus_purple, 351
    Step event, 351
  story, 298
  victory condition, 245, 299
Cherry Caves game assets
  spr_coin, 307
  spr_land, 303, 304
  spr_octopus_green, 306
  spr_octopus_purple, 306
  spr_platform_falling, 305
  spr_platform_moving, 305
  spr_platform_trampoline, 305
  spr_skybg, 304
  spr_terrain, 308
  spr_titlescreen, 307
Collectibles, 373
Collision mask, 28, 124
Collisions, 370
Color shooting
  enemy, 204
  jumping down action, 205
  super-attack, 204
  test, 206
Color-switching, 171, 174
  bullets color, 196
  colors.none, 196
  color variable, 194, 195
  enums, 194
  key, 195
  shooting key, 196, 197
  X key, 194
Competence, 504, 505
Cumulative feedback, 505

D

Dark Souls 1 (DS1), 370
Data structures, 64, 65
Deck of cards
  data structures
    grids, 70
    list, 67, 68

map, 68, 69
priority queue, 69
queue, 66, 67
stack, 65
designing
DS List, 71
For loop, 74
obj_controller, 72
repeat loop, 73
while loop, 73
making
Create event, 76
GML functions, 79, 80
shuffle_cards
function, 80, 81, 83, 84
Design document
feedback, 500
GDD, 499
playtesting, 500
destroy_instance(id, val), 139
DOOM, 438
Drag and Drop (DnD), 3
draw_rectangle function, 280, 336
draw_self() function, 41
draw_set_color(col) function, 41
draw_set_font(fnt) function, 41
draw_text(x, y, msg) function, 42
Draygon, 232, 233
ds_grid_create function, 441, 443
DS Grids
ds_grid_create, 441
ds_grid_get, 441
obj_map, 443
search, specific value, 442
step event, 444
two-dimensional arrays, 440
DS Map, 69, 460
ds_map_add(id, key, val), 460
ds_map_create() function, 460
ds_map_empty(id), 460
ds_map_find_value(id, key), 461
ds_map_replace(id, key, val), 460

E

effect_create_above, 140, 141
effect_create_below, 141
Enemies, add
blues
color switching, 207
color variable, 207
obj_bullet_enemy, 208
obj_enemy_blue, 207
rm_test, 208
spr_enemy_blue, 207
turret
image_angle, 214
obj_enemy_ufo_red, 213
spr_enemy_ufo_blue, 215
step event, 214
UFO, test, 214, 215
walkers on paths
closed path, 210
obj_enemy_walker_red, 209
path layer, 210
path_start, 211
speed, spd, 212
spr_enemy_blue, 212

Enumerator, 93, 94
Environmental storytelling, 507, 508
Euro Truck Simulator 2, 501

F

Falling platforms
  Alarm 0, 343
  Alarm 1, 343
  obj_platform_falling, 342
  particle effect, 343
file_text_close(file_id), 487, 490
file_text_open_append(fname), 487
file_text_open_read(fname), 486
file_text_open_write(fname), 486, 487
file_text_write_string(file_id, my_string), 487
Finite-state machines (FSMs)
  to code
    all_cards_paired() function, 109
    create Draw GUI event, 112, 113
    cur_time function, 108, 113, 115
    enumerator, 93
    game flow, 94, 95
    gameplay elements, 107
    global.cards, 95, 97–100
    obj_card's Alarm 0 event, 110
    play_time, 108
  states, 90–92
  timer
    alarm, 102, 103
    Alarm event, 104–106
    counting frames, 101
    obj_card's Left Pressed event, 104
  vending machine scheme, 87, 88
  Wait state, 89
First-in-first-out (FIFO) policy, 66
First level, creation
  climb ladders, 360
  concepts, 361
  jump function, 360
  suspended path, 362
Fixed *vs.* scrolling shooters, 172, 173
Fixing and re-adapting, Cherry Caves game
  Draw GUI, 336, 337
  HUD, 335
  obj_ball_green, 330
  obj_ball_red, 330
  obj_controller, 332–334
  obj_goal, 331
  obj_player, 337, 338
  room creation code, 332
  states.dead, 334, 335
  states.gameover, 334, 335
  variables, 331
Fun game, 501, 506
  autonomy, 507
  competence, 507
  designing levels, 508

environmental storytelling, 507
relatedness, 507

G

Game design document (GDD), 45
assets, 50, 51
card object (*see* Cards)
deck (*see* Deck of cards)
game modes, 49
memory
game flow, 49
rules, 47, 48
game_end(), 166, 276, 315
GameMaker language (GML), 3, 4
GameMaker Studio 2
coding, 5
game design, 4
installation
Mac version, 16, 17
Ubuntu, 19–21
using Steam, 17, 18
windows, 14, 15
overview, 6, 8, 9
prerequisites, 12
pricing, 10, 11
gamepad_axis_value function, 401
Gameplay mechanics, 172, 222
game_restart(), 166, 275
Game states, Space Gala
Alarm 1 event, 156
bullet's speed, 155, 156
Destroy event, 152
obj_controller, 150
playing state, 154
shooting, 157
shooting key, 154
state enum, 151
states flow, 150
states.gameover, 152, 153
Step event, 152, 155
game_state variable, 95, 150
generate_key function, 470, 471
global.can_pause, 446, 447, 463
Golden rule, 310, 368
Good old games (GOG), 515, 516
Grids, 70

H

Hash function, 470, 471
Hello World program
fonts, 32
GML
create event, variables, 35, 37–39
Draw event, 41, 43
Left pressed (Mouse) event, 39, 40
sprite, 34
Right Sidebar, 25
rooms setting, 32, 33
tile set, 32
Hollow Knight's map, 439
HTML5 game, 514, 515
Humble store, 517, 518

I

Ikaruga, 7, 171, 177, 194
Immersion, 100, 502
Inheritance
  collision event, 201
  defined, 198
  Destroy event, 200
  enemy stats, 199
  event_inherited, 202, 203
  hierarchy, 198
  obj_enemy, 198, 201
  parent-child relationship, 201
instance_destroy() function, 397
instance_find(obj, i), 161
instance_number
    function, 161, 272
ITCH.IO
  Butler, 513
  digital creators, 511
  HTML5, 514
  install Butler, 513
  Itch account, log in, 512
  Path, 513
Items and inventory, metroidvania
  collision event, 471
  DS List, 462
  DS Map, 460, 461
  equip item, 464, 465, 467
  GML objects, 470
  inventory, equipped item, 474
  inventory key, 463
  inventory screen, 462
  items information, 459
  keys and values, 461
  obj_upgrade_light, 472
  obj_weapon_heavy, 473
  open and close,
    inventory, 463
  unequip item, 467–469
Items and power-ups, Cherry
    Caves game
  Cherries
    Alarm 0, 357
    collision, 356
    obj_enemy, 358
    obj_player, 357
    powered_up variable,
      356, 358
    rainbow color-frenzy
      effect, 357
  coins
    Alarm 0, 355
    collision, 355
    obj_coin, 354

J

JavaScript Object Notation
    (JSON), 485

K

keyboard_check(key_code)
    function, 252
keyboard_check_pressed
    (key_code) function, 252
keyboard_check_released
    (key_code) function, 252
Klonoa, 374

L

Left Pressed (Mouse) event, 39
Level design, 376, 378

M, N

Metroidvania
  assets
    fonts, 393
    sounds, 394
    sprites' characteristics, 388, 389, 391–393
  definition, 382
  history, 382, 383
  isolation
    attack, 386
    controls, 384, 385
    enemies, 386
    game play, 384
    games, 388
    inventory system, 387
    maps, 387
    setting, 384
    skills, 386
    victory condition, 384
  platforming base
    create Step event, 397, 398
    gamepad support, 399–401
    gravity, 401–403
    instance_destroy() function, 397
    making leap, 404, 405
    spr_player_idle, 395
  wall jump
    action-adventure platformers, 406
    dash, 413–415, 417, 419
    defining variables, 407, 408
    game flow, 420, 421, 423–426, 428
    obj_player's Step event, 408, 409, 411, 413
    phases, 407
    warped, 429–431, 433
Metroidvania features
  combat system
    atk variable, 475
    bullet range, 475
    can_attack, 477
    collision event, 476
    obj_bullet, 474
    shooting, 476
  eight directions, grid, 457
  global.can_pause, 446
  Grids (*see* DS Grids)
  items and inventory (*see* Items and inventory, metroidvania)
  map menu, 457
  maps, 438–440
  Minimap, 452–454, 456
  old enemies (*see* Old enemies, metroidvania)
  save (*see* Save feature, metroidvania)
  semaphore, 446
    Draw GUI event, 450, 451

Metroidvania features (*cont.*)
global.can_pause, 447, 450
obj_controller's Step event, 448, 449
Minimap, 452
Modulo function, 82
Movements feature
Add Event, 131
controls, 132
create Alarm, 136
create event, 135, 136
GDD, 131
hp variable, 134
limit, 135
move_down_speed variable, 135
obj_enemy_red, 133, 134, 137
obj_player, 130
oscillation function, 134
pivot point, 131
rm_level_1, 132
speed variable, 135
Moving platform
hsp_carry, 346
obj_block, 346
obj_marker, 346
obj_platform_moving, 345
Mutual exclusion, 446

O

obj_card object, 53, 56
obj_controller, 72
Objects
code, 31
create instances, 29
events, 29, 31
Old enemies, metroidvania
Alarm event, 483
blinking, 481
collision event, 482, 484
destroy event, 480
force_jump, 481, 482
obj_enemy, 479
obj_octopus, 484
obj_octopus_green, 479
states.gameover, 483
Step event, 480

P

Paths, 209
path_start, 211
Platformer games, 368
aesthetics, 378
basic tasks, 373
camera management, 370
collectibles, 375
collision, 370
environment, 369
game mechanics, 377
gameplay elements, 379
HUD, 372
level design, 376, 378
Phantomilians, 374
platforming sessions, 379
power-ups, 373
Riddler Trophy, 375

RPG elements, 371
technological limit, 371
Player immersion
autonomy
Dark Souls, 503
Skyrim, 503
competence, 504, 505
relatedness, 506
point_direction(x1, y1, x2, y2), 143
point_in_rectangle(px, py, x1, y1, x2, y2), 190

Q

Quality of life (QoL) features, 309
Queue, 66

R

Race condition, 444, 445
Randomize function, 80
Relatedness, 505–506
Riddler trophy, 374
room_reset function, 272

S

Save feature, metroidvania
already_saved, 493, 494
DS map, 487, 489
GML, 486
instance_destroy function, 494
JSON, 485
load_game, 490, 491
Load Game option, 493
load or start game, 492
obj_checkpoint, 493, 495
obj_controller, 488
obj_player's properties, 491
pause menu, 492, 495
save_game, 488
text files, 486
Scrolling camera
features, 330
obj_player, 328
setting, 328
viewports, 328
Semaphore, 446, 448
Shmup level, 222, 223
Shoot 'em up,
history, 117–119
Shooting, Space Gala
alarm[1], 146
Alarm, event, 143
atk, 138, 141
bullet collision, 139, 144
can_shoot, 142
Destroy event, 145, 147
ef_firework, 140
enemies, bullet object, 146
event trigger, 142, 143
obj_bullet_player, 138, 139
obj_controller, 145
obj_player, 141
random attack, 146
shoot_delay, 142
shuffle_cards function, 80, 109
Sign(num) function, 258

Single-screen platformer
assets, 247
climbing ladder, 263–266, 268, 269
controlling Cherry Caves game flow
cherries to the star, 286, 287
cherry picking, 285
create obj_controller, 271, 273, 274
to die, 282–284
finite-state machine, 269, 271
HUD, 279–281
obj_controller GUI event, 276, 278, 279
obj_controller's Step event, 274, 277
create a hero
add Event, 251
check player, 253
keyboard_check(key_code) function, 252
keyboard_check_pressed(key_code), 252
keyboard_check_released(key_code), 252
fonts, 250
gravity system, 259, 260, 262
jumping, 262
level design
check jumps, 289
designing caves, 290, 291
level 2, 291–294
setting boundaries
block, 255
blocks hierarchy, 256
obj_player's step event, 256, 257
verify obj_player, 258, 259
sound, 251
sprites, 247, 249, 250
Space Gala, assets
fnt_messages, 129
fnt_score, 128
rm_level_1, 129
spr_background sprite, 128
spr_bullet_player sprite, 127
spr_enemy_red sprite, 127
spr_life sprite, 127
spr_player sprite (*see* spr_player)
Space Gala game
assets (*see* Space Gala, assets)
F5 key, 166
game flow (*see* Game states, Space Gala)
HUD
Draw GUI event, 158
obj_controller, 158
score, 159
spr_life sprite, 159
movements (*see* Movements feature)
objects, 129
rm_level_1, 148, 149
shooting (*see* Shooting, Space Gala)
victory

GDD, 160
obj_controller, 160
states.gameover, 160
Space Gala game menu
check bounds, 163
Draw GUI event, 164, 165
menu_index, 162, 165
menu_min, 162
option, event trigger, 165
paused game, 163
resume, restart, quit, 161
states.playing, 166
Space Gala (GDD), 119
controls, 120
enemy, 121
gameplay, 120
level 1, 122
menu, 121
pacing, 121
victory conditions, 120
Space Gala v.2.0 (GDD)
add enemies (*see* Enemies, add)
assets
spr_boss, 181
spr_bullet_red, 179
spr_enemy_blue, 178
spr_enemy_red, 178
spr_enemy_ufo_blue, 179
spr_enemy_ufo_red, 179
spr_player_blue, 180
spr_player_red, 180
controls, 175
enemies, 176
gameplay, 173, 174
level 1, 177
level 2, 177
menu, 175
sound assets, 182
victory conditions, 174
Sprites, 26–28
spr_player
collision mask, 124, 125
mask type, 126
mode, 125
pivot point, 123, 124
Stack, 65–66
Steam Direct, 518
Super attack
collision event, 219, 220
color-switching, 216
controller variable, 216
obj_camera, 217
obj_controller's Draw GUI, 217, 219
X-BOMB, 219–222

T

Tiles and level design, Cherry Caves game
blockers layers, 325, 326
grid toolbar, 323, 324
obj_player, 326
obj_terrain, 324
Tile sets
layers, 321
panel features, 323

Tile sets (*cont.*)
  spr_land, 319
  Tiles_1 Layer, 321, 322
Title screen, Cherry Caves game
  alarm, 317
  blinking text, 315
  cognitive flow, 310
  Draw GUI, 317
  obj_controller_title, 313, 316
  Room Editor, 311, 312
  spr_titlescreen, 312
  width and height, 313
Trampoline platform
  obj_platform_trampoline, 339
  obj_player, 340
Transition function, 90

U

Ultraprecise controls, 369

V

view_camera, 185
Viewport, 183

W

window_set_fullscreen
    (bool), 314

X, Y, Z

X-bomb charge, 174, 221