MonoGame Mastery

Build a Multi-Platform 2D Game and Reusable Game Engine

—

Jarred Capellman
Louis Salin

Apress®

MonoGame Mastery

Build a Multi-Platform 2D Game and Reusable Game Engine

Jarred Capellman
Louis Salin

Apress®

MonoGame Mastery

Jarred Capellman
Cedar Park, TX, USA

Louis Salin
Cedar Park, TX, USA

ISBN-13 (pbk): 978-1-4842-6308-2
ISBN-13 (electronic): 978-1-4842-6309-9
https://doi.org/10.1007/978-1-4842-6309-9

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

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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/978-1-4842-6308-2. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my wife, Amy, for always supporting me through thick and thin.
—Jarred Capellman

To my kids, in the hope that they pursue their dreams.
—Louis Salin

Table of Contents

About the Authors

Jarred Capellman has been professionally developing software for over 14 years and is Director of Engineering at SparkCognition in Austin, Texas. He started making QBasic text–based games when he was 9 years old. He learned C++ a few years later before studying OpenGL with the eventual goal of entering the gaming industry. Though his goal of professionally developing games didn't come to fruition, he continued deep diving into frameworks such as MonoGame, Vulkan, and DirectX as an important part of his free time. When not programming, he enjoys writing music and is working on his DSc in Cybersecurity, focusing on applying machine learning to security threats.

Louis Salin has been a developer for more than 15 years in a wide variety of fields, developing on Windows in the early days in C, C++, and eventually C# before working as a developer on Linux-based web applications using different scripting languages, such as Ruby and Python. His early love for coding comes from all the time he spent as a kid copying video games written in Basic from books borrowed from the library. He wrote his first game in high school and took many classes in computer graphics.

About the Technical Reviewer

Simon Jackson is a long-time software engineer and architect with many years of Unity game development experience, as well as an author of several Unity game development titles. He loves to both create Unity projects and lend a hand to help educate others, whether it's via a blog, vlog, user group, or major speaking event.

His primary focus at the moment is with the XRTK (Mixed Reality Toolkit) project; this is aimed at building a cross-platform Mixed Reality framework to enable both VR and AR developers to build efficient solutions in Unity and then build/distribute them to as many platforms as possible.

Acknowledgments

There were two big drivers for bringing me to focus my career on programming. The first being my father who handed me a QBasic book when I was 9 years old. He supported my passion for programming throughout my childhood, buying books and updated versions of Visual Basic and Visual C++ every release. The other was John Carmack. When I first played *Wolfenstein 3D* in 1992, I was mesmerized at how immersive the game was. A few years later seeing John Carmack on the cover of the *Wired* magazine, reading how he and John Romero had created and transformed the first-person-shooter genre, I knew I wanted to achieve that level of impact in my career.

—Jarred Capellman

Many events in my life brought me to this point, where I get to thank the people who have helped me and believed in me. My father bought me my first computer and enrolled me in a Basic class in sixth grade, where I learned how to draw lines and circles on the screen.

Years later, Richard Egli, my computer graphics professor in college, brought me to my first SIGGRAPH conference where I learned how deep the rabbit hole goes. Thank you for believing in me.

Finally, I'd like to thank Jarred, my coauthor, for giving me a chance to help him write this very book.

—Louis Salin

Introduction

Building video games has an undeniable appeal in the imagination of many people coming from various backgrounds. Many children want to be game designers when they grow up and many programmers have learned the art of writing code thinking that they would, one day, create their own game.

Creating a video game is both an expressive art form and a series of logical challenges that must be solved. A game programmer needs to be creative while materializing the content of their imagination on a computer monitor, and at the same time, they must constantly solve the many physical constraints placed upon them as they shape their game. For those of us that enjoy solving problems and have a penchant for art, this is a dream field, whether as a hobby or, professionally, as a full-time job.

There has never been a better time for regular people than today to write video games! While hobbyists around the world have built games since the 1970s, the amount of deep technical knowledge required has diminished and the barrier of entry has dropped much lower in the last few years. Game tooling and game engines now abstract away the complexities of getting something drawn on a screen, while computers have gotten so powerful that programmers do not have to be so precise anymore in the way they handle memory management and the game performance. Furthermore, getting a video game published on gaming consoles and computers has become much more accessible today to anyone with the perseverance to bring their game to completion, as can be seen with the sheer number of indie games found on the market.

Software developers today have a wide array of technologies to choose from when building their game. One of these choices is MonoGame, a framework for creating powerful cross-platform games.

In this book, we aim to take experienced C# programmers through a journey as we explain game development basics and build a small two-dimensional vertical shooter video game from scratch, using MonoGame as our framework. By the end of this book, our readers will not only have built a reusable game engine that they will be able to use in their future games, but they will also have gained valuable knowledge to give them a leg up in their future projects, whatever framework or engine they decide to use.

CHAPTER 1

Introduction

Chances are by reading this opening line you are at the very least intrigued to start learning about developing a game from scratch. This book was written to take you from this thought to fruition utilizing the MonoGame Framework. We will start by providing you, the reader, a strong foundation in the MonoGame architecture and continuing through with sprites, sound, and collision detection before wrapping up with separation of concerns preparing you for future developments.

Unlike other books on game development, this book will evolve with each chapter building on the last with a project-based approach as opposed to snippets of code here and there. For this book we will start from scratch on a vertical shooter akin to those of the late 1980s and early 1990s. The vertical shooter game type is a great starting point for aspiring game developers as it contains all the elements found in modern games:

- Multilayered scrolling backgrounds
- Collision detection of projectiles and enemies
- Computer-controlled enemies
- Sprites
- Player input
- Event-based sound effects
- Level structure

J. Capellman and L. Salin, *MonoGame Mastery*,
https://doi.org/10.1007/978-1-4842-6309-9_1

In addition, this book will dive into proper engine design and game tooling that are arguably overlooked in many game development books.

In this chapter, you will learn about

- MonoGame at a high level
- Difference between MonoGame and game engines
- Game types suited to MonoGame
- What to expect from the book and previews of the chapters to come

Who This Book Is For?

This book is targeting the aspiring game developer who wants to make a 2D game. Royalty-free game assets for sound, music, textures, and sprites will be provided (all created by yours truly), thus allowing the book to focus on the programming and architecture components of game development without worrying about cranking out game assets.

What This Book Is Not

While we will review game types as it relates to what pairs well with MonoGame, this book will not go over game design principles, asset creation, or the game development life cycle. There are numerous resources available including entire books devoted to these individual components and are outside the scope of this book.

Reader Assumptions

While no game development experience is required, there is an expectation that you are a seasoned C# programmer. While MonoGame is easy to get started with due to the architecture and the simple design, the framework is written in C#. In addition, the project we will be iterating on throughout this book utilizes many core aspects of the C# programming language such as inheritance and reflection. If you find yourself reviewing the accompanied source code and are struggling, I suggest picking up *C# Programming for Absolute Beginners* also from Apress to close the gaps.

From a development machine standpoint, this book will review how to configure a MonoGame development environment on both macOS and Windows with Visual Studio. Linux can also be used as a development environment with Visual Studio Code; however, Windows will be the preferred environment for the scope of this book due to the tooling Visual Studio for Windows offers.

With all of the assets provided with this book in formats MonoGame's pipeline can natively read (more on this feature in Chapter 5), no other tools are required. Experience with tools such as Photoshop, 3ds Max, and Audition will come in handy for your future development efforts even if it is simply a beginner skill level.

At the time of this writing, version 3.8 of MonoGame is the latest production version available, which was released on August 10, 2020. This version will be used for all code samples and snippets throughout this book. Versions 3.8.x or later may be available by the time you are reading this; however, based on the road map, samples should continue to work without issue.

What Is MonoGame

MonoGame at the highest level is a C# Framework that provides the developer a canvas to quickly create the game of their dreams. MonoGame is open source (Microsoft Public License) and royalty-free (over 1000 games have been published to various stores). While MonoGame does offer 3D support, the community by and large uses its powerful 2D support almost exclusively, and that will be the focus in this book (for 3D games the use of Unity or Unreal Engine is recommended). MonoGame's source code is available on GitHub (https://github.com/MonoGame/MonoGame).

Like many frameworks and engines available today, MonoGame like C# is cross-platform. MonoGame currently runs on

- Windows Desktop (7/8.x/10)
- Universal Windows Platform
- MacOS
- Linux
- PlayStation 4
- Xbox One
- Nintendo Switch
- Android (4.2 or later)
- iOS

For PlayStation 4, Xbox One, and Nintendo Switch, it should be noted that additional developer agreements are required before publishing to the respective stores.

Given MonoGame's underlying usage of C#, as new platforms become supported by C#, MonoGame should not be far behind.

Throughout the book, we will review any platform-specific considerations such as resolution and input methods (touch vs. keyboard, for instance). Fortunately, designing around a cross-platform game does not require much upfront effort with MonoGame.

MonoGame at a high level provides

- The Main Game Loop
  - Handling Updates
  - Rendering Method
- Content Manager
- Content Pipeline
- Support for OpenGL and DirectX

One of the best features of MonoGame's design is this simplicity, unlike other frameworks that have an extremely difficult learning curve to even get the first pixel rendered. Over the course of this book, we will extend this structure to support more complex scenarios and provide a rich expandable engine to not only build on with each chapter but also provide a framework to build your own game. In Chapter 3, we will deep dive into this architecture.

Seasoned developers at this point may be wondering what the relationship between MonoGame and Microsoft's XNA framework is. At a high level, there isn't a direct relationship. The underlying structure bulleted earlier is retained and the use of C# as the language is where the correlations end. MonoGame grew out of a desire from Jose Antonio Leal de Farias in 2009 to create XNA Touch. Similar to the effort on Mono Touch to bring Mono to iOS, the goal was to bring XNA to iOS. By that point, XNA was stagnating with the release of 4.0 in 2010 (which would be the last version released) and an official statement ending support in 2013. From there XNA Touch was renamed to MonoGame with support coming to Android, Mac, and Linux shortly thereafter. MonoGame eventually made it to GitHub and at the time of this writing has over 2200 forks with 267 contributors.

MonoGame Compared to Engines

MonoGame as mentioned is a pure framework. From the beginning of development, the goals of MonoGame were to create a flexible, simple, but powerful framework. The main design reason for this was to allow MonoGame to be used in a wide range of genres and game types as opposed to an engine that more often than not is tailored to a specific genre (generally the genre that the game driving the engine's development was such as the *Quake* series).

An engine conversely like that of Unity, Unreal Engine, or id Tech, to name a few, provides an end-to-end engine and editor with all of the various components that make up a game engine such as rendering, physics, level editors, and content pipelines with integrations into modeling programs. Depending on the level of deviation from the engine's core, there may be very little for an implementer to have to extend on their own. The engine approach allows a team of artists and designers a canvas ready to start implementing the game as opposed to waiting for the programmers to create the engine from scratch or build on top of a framework such as MonoGame. Learning curves and licensing fees of the aforementioned engines also should be taken into consideration.

If you're reading this book, chances are you wish to dive a bit lower level with a quick learning curve – this book should achieve that.

Game Types Best Suited for MonoGame

As mentioned previously, MonoGame is best suited for 2D games. With the revival of classics from the 1980s and 1990s in addition to a return to simple but fun games like *Castle Crashers*, this isn't a hindrance, if anything a benefit as the framework is set up for these game types.

MonoGame can be used in a wide range of game types; the following are a few examples of types that work best. In addition, for each game type, the pros and cons in comparison to the other types will be reviewed. When planning a game, weighing all of the pros/cons of a particular type should be a major part of your development efforts. For your first game after completion of this book, choosing an easier to implement game type is strongly suggested.

Vertical Shooters

Popularized by Capcom's *1942* and enhanced into the 1990s as graphics and gameplay advancements were made, vertical shooters can range from more science-fiction ala *Major Stryker* or more grounded like that of *Raptor*. As mentioned earlier in this chapter, for this book we will be building a vertical shooter from the ground up; a screenshot of the game from Chapter 4 is depicted in Figure 1-1.

Figure 1-1. *Our 2D game from Chapter 4*

There are some advantages and disadvantages to developing vertical shooters:

Pros

- Easy to dive into.
- Controls are basic.
- Graphics are easy to implement.
- Level generation and tooling is simple.
- AI is easy to implement.

Cons

- Tired genre
  - Need to generate some unique gameplay to differentiate from *Raptor* and other well-known vertical shooters.

Horizontal Shooters

Made popular by games like *Einhander* in the 1990s, similar to a vertical shooter, but affords more variety in the gameplay. A great MonoGame example of this is Pumpkin Games' *Paladin* in Figure 1-2.

Figure 1-2. *Pumpkin Games' Paladin*

There are some advantages and disadvantages to developing horizontal shooters:

Pros

- Easy to dive into.
- Controls are basic.
- Level generation and tooling is simple.
- AI is easy to implement.

Cons

- Graphics fidelity in this genre is required to be high due to competition.
- Tired genre
    - Need to generate some unique gameplay to differentiate from other games.

Side Scrollers

Side scrollers are a genre that took off in the late 1980s and continues to this day, offering a wide range of adventure and action games from the horizontal perspective. MonoGame's native support for sprites and hardware-accelerated 2D graphics have made this an easy choice to develop for.

Krome Studios' *Tasmanian Tiger 4* is a great example of fluid animation and fast action using MonoGame as shown in Figure 1-3.

Figure 1-3. *Krome Studios' Tasmanian Tiger 4*

There are some advantages and disadvantages to developing side scrollers:

Pros

- Diverse Gameplay is achievable.

Cons

- Graphics can be tricky to implement depending on the gameplay.

- AI can also be tricky depending on the gameplay.
- Tooling can also be cumbersome to develop for.

Role Playing

Made popular by the *Final Fantasy* series on Super Nintendo, the 2D isometric view has been used ever since for 2D role-playing games. A popular example of this game type with MonoGame is ConcernedApe's *Stardew Valley* as shown in Figure 1-4.

Figure 1-4. *ConcernedApe's Stardew Valley*

Pros

- Diverse Gameplay is achievable.
- AI can be easy to implement (depending on the level of NPC interactions).

Cons

- Graphics handling of the tiles and sprites can be cumbersome.
- Tooling can also be cumbersome to develop for.

Puzzle

Puzzle games especially on mobile given the popularity of *Angry Birds* and *Bejeweled* among others in recent years coupled with MonoGame's ease of use are a perfect fit. An example of this game type using MonoGame is Endi Milojkoski's *Raining Blobs* as shown in Figure 1-5.

Figure 1-5. *Endi Milojkoski's Raining Blobs*

Pros

- Diverse Gameplay is achievable.
- Graphics can be easy to implement.

- AI can be easy to implement.
- Tooling can also be easy to implement.

Cons

- Achieving a unique and/or fun gameplay in the crowded market can be extremely challenging.

Strategy

Strategy games commonly range between turn-based, real-time, and strategy/role-playing game hybrids. While much more complex to design and implement, they can provide a unique experience for gamers. Reason Generator Inc's *Wayward Terran Frontier* is a good example of utilizing MonoGame to its fullest in Figure 1-6.

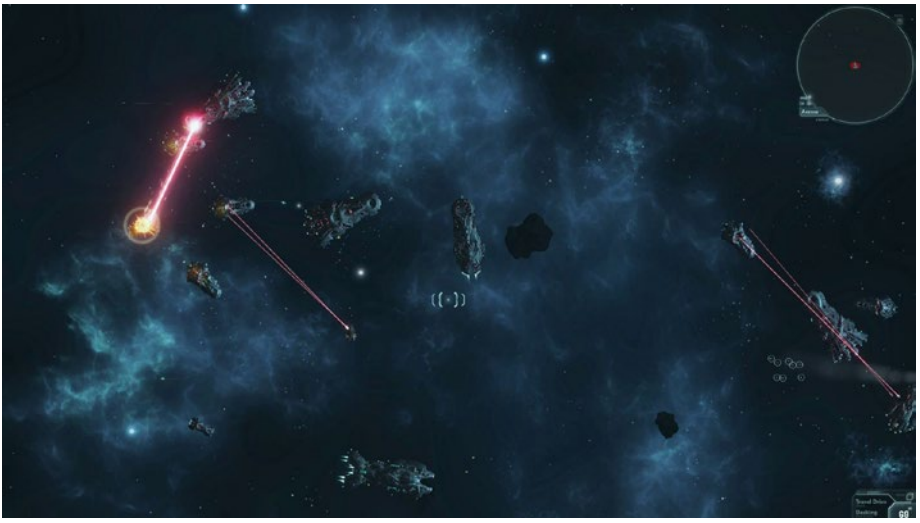

Figure 1-6. *Reason Generator Inc's Wayward Terran Frontier*

Pros

- Diverse Gameplay is achievable.

Cons

- Graphics can be tricky to implement depending on the gameplay.
- AI can also be tricky depending on the gameplay.
- Tooling can also be cumbersome to develop for.

Organization of This Book

As stated at the start of this chapter, this book breaks down each of the topics into manageable and isolated chapters. The following is an overview of the book and the topics we will cover:

Chapter 2 details how to get your development environment setup from start to finish for the remainder of the book. By the end of the chapter, you will be able to run a blank MonoGame project. Both macOS and Windows setup will be covered in detail. Linux will be discussed, but not recommended going forward for the rest of the book.

Chapter 3 deep dives into the MonoGame architecture including going into detail about 2D graphics, the game timer, and input. This chapter should not be overlooked even if you have done game development in the past as it will offer a deep insight into how MonoGame's architecture is set up.

Chapter 4 starts the deep dive into creating an architecture that we will be building off of for the remainder of the book. As with Chapter 3, this chapter should not be skipped as the objects, managers, and Game class changes will be described in detail.

Chapter 5 goes into detail of how the Asset pipeline works in MonoGame. In addition, integration with the ContentManager into the Game States will also be detailed. At the end of the chapter, we will render our first sprite.

Chapter 6 covers the handling of input with both a keyboard and mouse. In addition, platform-specific considerations will be reviewed to handle gamepad and touch screen input.

Chapter 7 goes into how to add audio to our architecture and add audio triggers to our event system. In addition, supporting background music layers will also be discussed.

Chapter 8 deep dives into how to integrate particles into our architecture to handle the bullet fire from both our player object and setup for future enemy objects.

Chapter 9 reviews various methods of collision detection used in games. For our project, we will use box collision and integrate it into our architecture to handle not only player object collisions but also projectile collisions.

Chapter 10 adds animations into our architecture and reviews approaches used throughout the industry. At the end of the chapter, animations of objects are added to the game.

Chapter 11 reviews the importance of level design and goes into detail of how to add level loading to our game engine.

Code Samples

Code samples starting with Chapter 3 will be referenced throughout each section. Outside of the code samples, there is also an Assets archive that contains all of the music, sound effects, sprites, and graphics used throughout the book.