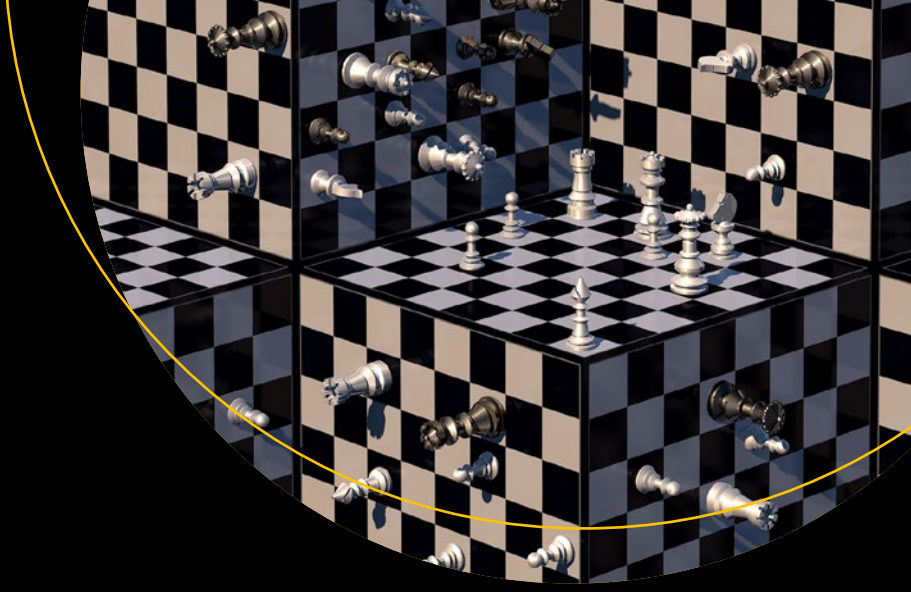

How to Make a Game

Go From Idea to Publication Avoiding the Common Pitfalls Along the Way

—

Minhaz-Us-Salakeen Fahme
Tanimul Haque Khan

Apress®

How to Make a Game

Go From Idea to Publication Avoiding the Common Pitfalls Along the Way

Minhaz-Us-Salakeen Fahme
Tanimul Haque Khan

Apress®

How to Make a Game: Go From Idea to Publication Avoiding the Common Pitfalls Along the Way

Minhaz-Us-Salakeen Fahme
Narayanganj, Bangladesh

Tanimul Haque Khan
Dhaka, Bangladesh

ISBN-13 (pbk): 978-1-4842-6916-9
ISBN-13 (electronic): 978-1-4842-6917-6
https://doi.org/10.1007/978-1-4842-6917-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: Matthew Moodie
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 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/978-1-4842-6916-9. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To our parents and siblings who took care of homes while we played and made games

and

to those who laughed at us for loving games

Table of Contents

About the Authors

Minhaz-Us-Salakeen Fahme is the co-founder and CEO of Battery Low Interactive Ltd, a company that makes wishes come true. It started as a game studio in 2015 and now has several wings covering serious games; web and app development; business and marketing solutions using augmented reality and virtual reality; digital marketing; and small-scale indie games. With its outreach campaigns, Battery Low has reached a half-million kids with the experience of immersive technology for the first time. Fahme also conducts sessions on AR, VR, MR, game design, entrepreneurship, careers, and leadership for youth and entrepreneurs in educational institutions and national and international events. He is a blogger and game designer/writer by passion while cats, travel, live concerts, and epic fantasies keep him running.

Tanimul Haque Khan is the head of the Unity department of Brain Station 23, a service-based company. He and his team have been providing AR/VR enterprise solutions since 2018. He has been working with Unity technologies since 2015. He has played an uncountable number of games across multiple platforms, mostly on PCs and consoles. He is one of the most well-recognized Unity developers in Bangladesh. Outside of the tech industry, he cofounded the very first cat cafe in Bangladesh known as Capawcino.

About the Technical Reviewers

Joshua Willman is an instructor, a Python developer, and an author. Normally he creates courses or software for various fields, including game development, AI, and machine learning. In his free time, he pursues other ventures, such as becoming a solo indie game developer and building up the site `redhuli.io` to explore utilizing programming for creativity. He is also the author *of Beginning PyQt: A Hands-on Approach to GUI Programming* and *Modern PyQt: Create GUI Applications for Project Management, Computer Vision, and Data Analysis*.

An immersive technology entrepreneur, **Abhiram A** is also the cofounder of Odyn Reality, an XR Tech startup. He is a Unity3D Ambassador for Unity India and doubles as an XR Coach for Pupilfirst Facebook School of Innovation. He was also part of Future Technologies Lab, Kochi as a Research Fellow in VR. He is a Udacity VR Nanodegree graduate and a Unity Certified Developer.

Acknowledgments

Thanks to Professor Josiah Lebowitz, founder, lead designer, and writer at *Pen and Sword Games* and Professor of Game Writing at George Mason University for your mentorship, guidance, and inspiration. This book has been enriched a lot by your review on such short notice.

Thanks to Istiaque Ahmed and Minhaj Mimo for continuing together against all odds and the superheroes of Battery Low for making wishes come true as we kept making games.

Introduction

This book is like a journey that begins as wishful thinking regarding game development and ends in publishing and earning from it. It starts with your strong affection towards games and paves the way for you to move from a hobbyist to a professional. It is directed at beginner to intermediate-level game developers. If you want to form your own indie game studio, you can use this book as a guide since we cover the technical and business aspects of the journey. This book will also point you towards future improvement scopes at places. If you just want to be a game industry professional, this book will serve you nonetheless.

We delve into the pitfalls and common mistakes made in real-life cases of game development. In fact, we dissect games and their development stories to understand the secrets of making a good game. Besides sharing the dangers, we also discuss ways you can try to avoid them.

We start the book with the core motivation behind game development. We show how to analyze games, benefit from them, and learn about game engines and how to choose among them. We explore the game design document (GDD), version control, game design, intermediate coding practices, testing, where and how to publish, game marketing, reality checks, career cautions, and attaining success. Before you take on the chapters that include coding, know that we expect you already know the basics of programming; the chapters will just elevate you to the next level in game programming.

If you are a beginner-level game developer, this book can be the ladder for you to reach the intermediate level. If you fall down in between, the restart mechanism is always there. While you read this book, perhaps don't use it as a one-time manual. Think about the topics as you read them, and take time to review the topics as required. This book can serve you for a long time as a cheat sheet or a handbook which you will come back to in many intervals of your career.

PRESS ANY KEY TO CONTINUE...

CHAPTER 1

Don't

Do you think everyone in this world gets what they deserve? Mathematically, there's a 50-50 chance that you're laughing out loud right now (yes, I know the abbreviation, but we're trying to keep things a bit formal here). But my heart says, four out of five good people out there reading this sentence right now are laughing sarcastically or sadly. Because you most certainly believe that I am joking, which I can't certify because no judgments shall be done here, and one out of the five folks out there believe it all makes sense in this world. You don't believe me? Oh yes, you do and you're laughing about them too, now. Have it your way, because a good laugh never hurts.

That's a hefty debate to start this book. But it is important that we discuss perspective. The night ahead is long and dark, and you'd need more than one miracle to get through this. So, we are here to do that. Let's call it a reckoning. A summoning of miracles.

Back to the debate. I say, all five people are right. Yes, often people don't get what they deserve, but sometimes they do. Let's say you have made a great game. It's so perfect that you can't stop playing it. It's really good, but millions of people are not paying money for it. Now, it's impossible that the perfect game you made will be recognized worldwide. Because the world isn't fair, I know. We all know the legendary game studio that was at the forefront of the mobile game revolution. It didn't work out for them for what—51 times? But it did work out for them on the 52nd time. The game they released—after 51 games that no one loved furiously—was Angry Birds. So, the universe didn't give them what

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_1

they deserved at first, but eventually, they got it all back and more. This doesn't mean that we all need to experience 51 titles of commercial heartbreaks before we hit our big break. It can be the fifth title you ship, or even the first one! There's no pattern to it. But hear me out: if you can just pay the price for success, it increases the probability of being successful quite a lot. That's what we can do most: increase the likelihood of success. Just a point to be noted, when it comes to making games, the payment for success is quite high. It's expensive, but not impossible. Enough sparring. Let's get down to business now.

Defining a Game

You probably have read many definitions of "game" by a lot of veterans in the industry. I know some of them by heart too. We have our own explanation of the term "game" as well. You probably have one too. If you do, that's great! If you don't, you can always believe in someone else's, but make sure you believe it from the heart. Because, friend, that's the central pillar of our world: belief. Make sure you have a solid foundation. The one who doesn't might have to start over someday from the beginning. After all, anything without a proper basement crumbles, be it today or tomorrow. Sometimes a storm comes by, and sometimes your faulty basement can't withstand the new weight. Either way, it happens, and let's not head in that direction. Let's have a raincheck on our basics.

"Play" can be the most meaningless yet the most meaningful thing ever created. Now, why would we say that? Two human beings kicking a ball or running around trying to catch each other without absolutely any reason at all—does this make sense? Maybe tapping on a screen to see a bird jump is meaningful. Knocking around a few stones, gunning down a few weird-looking aliens, matching candies. You're laughing? So is the full-grown man who has been playing the game for hours. He doesn't know exactly why he is laughing; in fact, he doesn't bother at all. He is having fun.

No matter how meaningless it may seem to the naked eye, playing has always been an integral part of our existence. It is about fun without any dependency on reason and logic, and no human being can survive properly without it. Even animals play, right? It is the mysterious activity that creates the emotion of fun and happiness among us all. In a game, we create "play." Well, some of the play can have a definite purpose, such as for education, training, or getting closure. These games are called "serious games." But that's just a category of games. The other type (the one with the majority) focuses on entertainment, primarily.

Any kind of play happens in a world of its own. To us, making games is like creating a world. You make some elements, patch them together in an environment you design, set some rules, and then set things in motion. Sounds just like the life we are living out here in reality, doesn't it? Well, maybe some simulation games like the Sims or choice-based narratives, you're thinking. But we have to ask you to reconsider, if so. It's not confined to those genres only.

Let's take the example of chess. You have elements like pawns, knights, bishops, and kings. They lie in an environment that has been set. The game has its own set of rules, and the players must follow them. You can break the rules, but then you must face penalties if the referee sees you. If you're playing a digital version, and you use a cheat code or hack it, the original creator should be able to identify it.

As a game creator, you to make your own game. Whether you cater to the popular public choices or explicitly your own ideas is totally up to you. But if they have things in common, then congratulations! You just achieved a big milestone, and you are a force to be reckoned with, so well played! Either way, making a game means creating a world, and this is no easy job. It's also not a small job.

Have you played any of those real-time strategy games, such as *The Age of Empires*? It's fun, and people spend hours building bases and civilizations. I still remember playing the last mission of *Age of Empires 3*; in it, they had a fight around a spring/lake which was stunning! Now, the

computer I had back then didn't have the best configuration and I had to wait two or three seconds after every single click to see the action start. Yes, I finished the level. Imagine the level of fun if even that insane lag couldn't stop me. Countless people around the world created farms, blacksmiths, markets, prayer halls, laboratories, and then upgraded them for centuries in the game. It's a lot of work! Of course, the game didn't cover all parts of civilization. No one is interested in building places like dirty public toilets, so naturally, you skip them and a lot more. Also, it's not feasible that you have to spend 35 real days after you start constructing a building. So imagine how difficult it was in reality to make this world. If you want to make a world of your own, it is going to be a lot of work as well. The smaller the world is, or the less complex the rules are, the less time and resources it's going to cost. It's just proportionate, that's all. But no matter how small it is, it still is a huge task. There's also another factor you have to consider.

Basic Components

So, what are the basic components you need in a game? You must have some sort of environment. It's very logical and common to start with the environment. Is it like the real world? Then it has to look close to reality. Is the sky too blue? Probably you should try a different skybox. The water shader looks too cartoonish. The houses look like dollhouses. The wire patterns are too perfect; that's not how it happens in the real world. Oh, are you forgetting something? You forgot the birds. There are thousands of things to pick from and you must make sure you don't miss out on the important ones. Also, they have to look realistic.

Let's say you go for an imaginative environment. This has its own issues. What kind of world are you thinking of? Is it a rectangular box? A circle? A dungeon? Have you checked and implemented the laws of physics? What if something falls off the boundary? Is that condition checked? What happens if something hits with a huge force? Is the background distinct? Maybe the gamers will be confused about the UI,

which is clearly blending with the BG color. Again, the artwork must match your imagination. The sound has to resonate. And you must check if it is becoming similar to an existing creation by someone. It's not unusual, of course; there are billions of people, so the odds say that it could match.

Let's think of the characters now. There are two kinds of characters in a game: non-player characters (NPC) and player characters (PC). The characters who stand by and are controlled by artificial intelligence or your programming are non-player characters. The ones who are controlled by the player are player characters. You have to make sure you have made all of them properly and given them enough background and context so that no one seems like a loose end. Oftentimes we neglect the side characters among the NPCs. But they can turn out to be some loose ends in your masterpiece and ruin the whole experience eventually. The PC is the most important, undoubtedly. But the tricky part is, you have to think about the actual players who will be playing the game. After all, they will be controlling the character, right? So, have you spoken to them? Are you sure you know what they want? If not, you can't just "assume" their thoughts. If you do, you are just praying that it will work out. It's like a lottery—well, a calculated lottery—since you definitely have some knowledge about it because you are one of the players. But if you don't know about the hundreds and thousands of players and their feedback, their choices, and their philosophy, then you are not prepared at all.

You are not paying the price for the treasure. You are not hopping onto the adventure. The adventure has its pirates, sea monsters, and jungle terrors. If you sit in your home and pray that you will have that treasure of Jim Hawkins or Indiana Jones delivered by Amazon/FedEx to your door, well, you know the probability of that. So yeah, you have to know what your players want. No matter how you "feel" about a feature, you must test it with the players to check the reality, at least to some extent.

We haven't even gotten into programming yet. There are countless ways it can go south on that front. You can simply forget an `elseif` condition. You can forget to reset some counter. It's to be expected that

your users will keep discovering bugs and report them to you if they like your game. If not, they will just uninstall it and move on. That's a whole different topic, though; we are going there soon.

You have made a game, but it is silent as of now. Now, what voice do you summon? Have you checked if you had faults at the end of the loop and there's a tiny bit of imbalance at the beginning every time after it first plays? Is the sound effect too funky? Will people want to mute the tracks or listen to them even when they are not playing the game? You have some good sound, but does it match well with the context of the game? Okay, let's say you hear it loud and clear.

Now is the time for the database, high scores, and the marketplace. You can't afford any mistakes there; it's sensitive. Have you tested everything before rolling it out? If your team can't find the bugs here and a player does, then you might be as well be finished. Why do we keep saying that?

Making Your Game

We are guessing that you have some ideas about the competition out there. If you don't, then you should finish this chapter and immediately (not tomorrow, not next week) do some research. It doesn't matter where you want to release your game. Every platform has competition that is growing at an insane rate. We don't live in an era where digital games are a thing for kids anymore. Even before COVID-19, one in three people played games. After the lockdown of months, well, we can safely imagine the number has gone much, much higher. People took shelter in games, and game makers welcomed them with open arms. The games that had 10-15 million downloads before the lockdown have crossed 50 million as of the last quarter of 2020 (when we are writing this).

People know that games are the present and the future. The industry is booming and so are its components. Even if you do very well in terms of making, there isn't any certainty that your game will be a massive hit.

Maybe you couldn't fine-tune the matching mechanism of online players because you were short-staffed, or maybe you have some silly random issues with the scoring. Even though your game is very good, people will still uninstall it and move to your competitor because they can afford it. You have to be perfect to stand out. Well, even if you are perfect, there's no certainty of the limelight. But you have a good probability. You can try those big publishers. But they are flooded with applications. So if your game isn't really killing it, they can't afford to spend their time on your game. It's going to take a sizable amount of time and resources to make something perfect. So, tighten your seatbelts for a long and bumpy ride.

Above all, you must know your core motivation behind making a game. What is it? Do you love playing games? Do you love it when people play your games? Or is it the money that ticks for you? Whatever it is, find it properly and get ready accordingly. If it's for your own pleasure, then you can do anything you wish with it, bend every single rule, and still be happy about it even if nobody else plays your game. If it's about making other people happy, then you should be clear on what makes them happy; you can't just sit back and assume things. You have to know facts and figures. If it's for both, then it's challenging, but it means more fun at the same time. If it's only for the money, you should probably do something else. There are lots of things to do which are not this much sophisticated.

The game industry is a giant one and people are making billions out of it. But you should definitely look past those fancy numbers and ask yourself what your goal is. How many people make games and how many become successful? If money is the only motivation, it's better to play it safe and just buy some successful game franchise and earn from its future revenues. That's easier if you have the money. It's an investment, rather than a creation. The journey of making anything is different. If you are going to create something from the ground up, you need miracles. You need people who can create magic because that's what it takes to make a successful game. It's **magic**. Well, it has a fair bit of science in it, but to put it to proper use, you need wizards. And, as they show in the Disney

movies, magic isn't something you buy in the marketplace or order online. You have to believe it from the core of your heart and dedicate your life, at least a sizable portion, to it. There's always a price. You pay it; you make yourself worthy. Also, you must know why other people failed while trying to master the art. What makes a failed game? Why is it a failed game? Who failed and why? You must know them as well as the success stories.

The Don'ts

It's a tough job, so don't do it if you don't like an adventure.

It's going to be a long journey, so don't forget that.

Don't be naïve: know what you're getting yourself into because the treasure lies at the end, and the path is perilous. Don't come if you don't have it in your core.

Don't worry. The journey can be magical if you are true to your heart.

Now, let's find out how it all begins.

CHAPTER 2

The Fault in Our Stars

Do you remember the first game you played? You probably spent days, if not weeks and months, playing it. The excitement, the feeling of discovery, and the fun of it can't be expressed in words. It's almost sacred. The very thought of it makes your eyes close as you get lost in euphoria. What was it? Maybe...*DXBall*? I loved *DXBall*. Getting the shooting power button was more important than passing the levels for me. Or was it a racing game? Like *Road Rash*, where beating other racers with sticks was the most fun part? Maybe you are even more vintage, and you played the first *Pong* in the arcades. Perhaps it was *Mario,* and you were, let's face it, not good at it at all. You kept jabbing at it until, after hours of struggle, you finally reached the end goal. Did Mario find his Princess? Did you get what you were looking for? If you are buried deep in memories, then you are an old-timer, like us. Well, you might not be and that's also okay. There's no proper age to fall in love. It just happens whenever it happens. Just humor me; could you squint a bit and try to remember the initial encounters with games and your thoughts regarding them? Did you feel like changing something in those games back then? Would you want to change anything now, given the opportunity?

We received different responses to this question when we asked our game developer friends around the world. Some say that those games were perfect just the way they were. Others had things they would have dearly loved to see in some of the games they played initially. But they could not, of course. Then the question came into their minds: what if I could make them as well? Maybe I could get this feature in! That's definitely a childish or a romantic thought. But to create art, doesn't romanticism help?

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_2

Like, a lot? For us, there were some games we loved so much! But, despite worshipping them, some aspects seemed lacking to us. Those little issues made us think about learning to make games in the first place. So, that's our motivation. What's yours? Why do you want to make games? Find out the answer to this question honestly and hold it dear. Always remember this. In this chapter, you will try to discover it. You'll also find out how to make the most out of playing games and how to analyze them.

Let's Go Back to the Start

I have yet to find a beginning game developer who isn't fascinated by the games they play. Yes, we have seen some cases here and there where developers grow some sort of resistance towards playing games after they have been at it for a really long time. Speaking from the experience of watching them with my own eyes, I can safely say that they have had their fair share of gaming in life already. Maybe it has become too much after so many encounters. Some people spend so much time in the game world that they probably think about them even when they are not playing or creating games. Continuing this for years can have some impact on some of us. It's more of a psychological issue, and it doesn't happen to everyone. Even if we look at industry people who don't play that much anymore, they were all crazy about playing games for a sizable portion of their lives. The rest of us, well, I guess we are here primarily because of our love for games. But how did we become makers from just a consumer? How did the transformation take place? Why did we ascend?

Both of us love myths, thanks to the habit of reading books with earnest interest. One of us got into mythology at a very early stage of his life. Not long after, he found himself playing a video game with one of the characters he loved, Hercules. It was a 2D platformer but full of challenges and fun gameplay. To this date, the gameplay rings loud bells, a lot of them, in his mind palace. There were, of course, racing games like *Road*

Rash, which was very relaxing to play, especially while using the feature to kick your rival racer or police in the game. There were shooting games like *Virtual Cop 2* and *House of the Dead*, which you could not get tired of playing, no matter how many times you finish the campaigns. But then, one day, he stumbled across something he didn't know existed.

It was a Harry Potter game. *Harry Potter and the Prisoner of Azkaban*, based on the third book of the phenomenal series by J. K. Rowling. The experience was nothing short of a dream. After that, he tried every game in this series, hoping they would give him that feeling again. Well, most of them were really great! But not… *perfect*. "It's possible to make a perfect game out of this book. It's possible that it would be the best game in the genre," he thought to himself after finishing the final book of the series, *Harry Potter and the Deathly Hallows*, just one week after its global release. The game came out later. Well, not *a* game; there were a couple of games. Both of them were really good to play. The gameplay was great, they followed the story in most parts, and the graphics quality was terrific. But to him, something was still *lacking*. He wasn't sure…probably the ensuing tension that was supposed to be there when the players play the game. Maybe it should have been more challenging? Not difficulty-wise, but there was an absence of the monumental tension that was felt for real in the original story. The kind of tension you feel when you are playing *The Last of Us*, *Uncharted 2*, or *Shadow of Mordor*. To summarize, the games weren't *perfect*. They weren't the best game in that genre. "But they *had to* be!" he thought. "Someday I'll make sure mankind does proper justice to their creations and produces the best output where it deserves." Thus, a seed was planted, which grew over the years, into writing this book today. But, it's far from over. The journey has just begun, but it will not stop until he discovers the formula to create his *perfect* game.

The other author played his first game at a friend's house, and it was named *Cadillacs and Dinosaurs*. After that, he sort of got addicted to playing games. He's a fan of old-gen games such as *Mario*, *Contra*, *Loony Toons Adventure*, and *Chip n' Dale*. But it wasn't until 2008 when

he thought of developing games. By this time, he had played over 250+ unique games on consoles and PCs. But all of them were single-player or split-screen multiplayer at best. Then one day, he tried a game called *Twisted Metal 2*. It was ported to PC. It was a car death battle game. The game was simple: you destroy every other car with the powers you find in the match. The game mode was last man standing, currently known as the battle royale or king of the hill, etc. This type has been on the market forever, just in a smaller scope. And then he thought, "I want to play this with my friends." But the game didn't offer much of a scope to play over the Internet. And then he thought, "Imma gonna make a game myself that I can play with my friends." Since then, he has done his best to learn about game development little by little so that one day he can make something close to that game. He still has a long way to go. All he desires is to play games and have fun with friends.

Thoughtful Playing

Let's come back to the present. A study by Facebook Gaming states that the game genre is the top deciding factor for mobile gamers when trying out a new game. Each of us has a favorite game genre. Some love action/adventure games; many of us are shooters. A lot of us spend a lot of time playing our favorite sports game. Maybe you are addicted to battle royales or games with good narratives. Many of us have played hundreds of hyper casuals already, if not thousands. We believe that to be a good professional in game development, it helps if you play games. It might not be mandatory, yeah. I mean, if you have a role where it doesn't require in-depth knowledge about games, maybe you can make do. But even then, this will only take you so far. Games are a unique kind of matter. They require a perfect blend of science and art like no other. We cannot compare it with anything else. Hence, the understanding of the matter is of utmost importance.

There is an infinite number of things that can go wrong; after all, you are building a world. As a creator, the more you know about the world and its elements, the better for you. Now don't get too much ahead of yourself after reading this. If you keep playing games without thinking, maybe it's not the most optimal way forward. Many of us make this mistake. We keep playing without thinking and assume that it is making us better. Well, I won't disagree. But to what end? If you are doing something, why not make the most out of it? That's what an intelligent human would do, right? Answer this: would it hurt to think about the game in a way where you assume yourself as the developer? I mean, you already know the issues the game has, considering that you are an attentive gamer. Now you just have to think about them from a developer's perspective by putting yourself in the creators' shoes. Let's discuss this with an example.

Maybe you are a big fan of racing games. There's an exclusive title out in the market, and you embraced it immediately. It is fun to play and has different modes inside. The most exciting one is, of course, the competitive one. You burst into the lobby looking for a random opponent. You are probably a medium-tier player. You find an opponent, and the race begins. A minute into the contest, you are just overpowered by him. Losing miserably, you go for it once again. In the next race, history repeats itself. You get dismantled. The third time's a charm, you think, but you suffer the same fate once again. By now, you are probably complaining in the social media chat group you have with buddies who play this in your community. Apart from all the swearing regarding the makers, could you stop for one second and think about it from a game developer's angle? I mean, let's calm down for a second after you are done with all the heat. You think there's an issue with the game. Ask yourself, what is the issue? You are not matching with players with a similar level, right? If you do not know, then ask around or look for it. There's a technical term for it; it's called "matchmaking."

Now you could simply stop after the complaint spree with your peers, or you could think about the problem methodically. If you did, you would discover this term and more. You could specifically read about the

matchmaking system that's in place in this game (according to the developer) and try to match it with your experience. If there's a contradiction, you could perhaps open up a dialogue with the developer. If nothing of that sort happens, still you'll learn, and that's probably the most crucial aspect of all. You need to learn continuously. The superpower that all game developers initially have is that they love what they do and love what they make. Think of it like cooking. If you're not only a great admirer of food but also a good cook, you are a winner big time. It's the same analogy here. You get to play good games yourself; you just have to make them. While you are at it, you must play lots of other titles as well, which contribute heavily to building up your love for games. Playing games takes up a lot of time, and it should. Add a bit more time as an add-on to think about them, and you will get yourself a very handy practice!

Now comes the inevitable question. You probably are not a very skillful creator yet. So how can you decipher the blockbuster titles? Guess what, nobody is judging you! Assume incorrectly, make mistakes. Making games is a journey that unspools along the way. Once you are on the path, you just have to keep at it. Yeah, it's probably an unknown one, but so were the journeys in your favorite tales. The heroes in your favorite fables get hit with many problems, but they keep going regardless. The mission seems foolish, and impossible in the beginning and setbacks happen. But they are just part of the grand tale, and they must happen. You can't go around them; instead, you have to go through them! You will hit obstacles, meet dead ends, but everything will contribute to something. Every game you play and analyze shapes you.

You can draw conclusions about how a particular segment of the game was made. Let's start with the things you find the most fascinating. Is it the artwork? Maybe it's so beautiful that you want to live there for real, like those islands in the *Uncharted* series, or the vast nature in an *Assassin's Creed* game, or the record-breaking *Ghosts of Tsushima*. Okay, let's see what information is available on the Internet on the making of those art pieces. Is there any way to get in touch with one of the creators who made this? If yes, what questions would you ask them?

Or maybe it's the gameplay that you found intriguing. Maybe some superhero games where flying and shooting mechanisms give you the ultimate pleasure. Is it the first of its kind, or are there more games with similar kinds of pleasing experiences due to the gameplay? Maybe it's a puzzle game on your phone that keeps you hooked. Dive deeper into it. What's the secret of making the game? Has somebody given an interview? Maybe they have written blogs, tutorials, or even a press release. Dig them out and get lost in them. Don't limit yourself while diving deep into analyzing them. You need to let go and embrace openness. Starting with your favorite items always helps. After you have dug into things you like about the game, move on to things you are indifferent about and things you disliked. After all this, you might even think of some new things they could do!

Diving Deep

Regarding analyzing a game, we can suggest some definite steps to keep it structured. But please note that it's only a standard you could follow. You can always make one of your own and follow it if it suits you, as long as you are getting and using all the possible information that can be salvaged.

We can break it down into these few steps:

1. Play
2. Write
3. Modulation
4. Hypothesis
5. Study and reality check
6. Enlightenment

The first step is the most straightforward one: playing games. There is no telling what games you "should" play in general. Start by discovering which genre you like. If you are an explorer, then you should be able to find enough games that you like on your journey. When you play a game, it's always better if you give it at least some time. Some games take time to settle into your heart and provide the punch. They are just designed this way. Also, in this era, you can easily find reviews for any game that has even a minimum bit of traction. Going through reviews can save you some time, but do make sure not to get spoiled too much, which is a common risk when you are browsing for feedback from people who have already played the game. While playing, enjoy yourself. When you feel like you have a grip on the game, start thinking critically. Why is this happening in the game? Why did they write the story this way? How accurate is the fighting mechanism? How realistic are the game mechanics? How did they implement this feature?

Write down your thoughts. You can write randomly if you feel like it. Positive or negative, just write them. You can just write, "the music that plays after the player's friend dies is heartbreaking" or "the shooting mechanism seems unstable." Don't leave out anything; jot down whatever thoughts crossed your mind while playing.

After a while, see if you can segregate the elements you are thinking about. There should be some specific topics you can identify among all the noise out there. It may look messy until you dive in deep, but as soon as you do that, it should be much easier to sort out. This part is the modulation. Again, there are no hard-and-fast rules about which modules you should sort out. Just go ahead based on your understanding. For instance, you could just put some feedback for game mechanics, a few regarding game art, some in music, UI, scoring, and so on. But this is not the only approach. You can do it differently. Just make sure that you separate the modules well so that you can work on them individually later on.

You probably have some interesting responses by now. It's time to take this to the next level by forming some theories of your own! Take a module and find an issue in it. For example, let's say you have found out that you can jump over any rock except one in an area. This issue falls under "game mechanics" in your list of modules. Why do you think you can jump over any rock except a particular one? The first thought that comes to your mind is, "maybe the developers didn't keep anything beyond the rock, so they didn't want me to jump over it in case I fall into nothingness." My friend, you have just made a hypothesis. Congratulations! Hypotheses are thoughts you generate based on your understanding. Now for each issue, good or bad, try to form a list of hypotheses. You will work on them afterward.

Here comes the most lengthy part. You have these issues jotted down. Now let's see what information is available regarding them. While searching on the Internet, keep in mind that you can find feedback from the makers of the game on various channels. Maybe you don't need to go that far. Most of the issues get addressed in various blogs, forums, or YouTube videos. Cross-check your findings with the hypotheses now. Were they correct? If not, then what was the issue? Keep on doing this exercise for all of the issues. Now, if you're really thorough and if you're lucky, you might actually find some issues that haven't been discussed or solved anywhere. There's a probability, albeit small, yes. Well, in that case, just go ahead and see if you can get in touch with the people who made the game. See what they use to communicate with the public. Is it social media where you can message them, or is it an e-mail address? Whatever it is, don't feel shy. As soon as you explain your raw thoughts, you are done. If they don't get back to you, well, no harm is done. What do you have to lose here? Nothing! However, if they do respond, then it's a big step. It may feel like a miracle when it first happens. Enjoy the feeling to the fullest and go on dissecting the issue until it is resolved, as long as you have any leads remaining.

I guess you are waiting for the final step now. Guess what; it's done already! By the time you have done these five steps, you have dissected a game successfully. You have met many characters in the journey and

encountered much information that was previously unknown to you, and these things have shaped you into a person who knows and understands games more deeply. The journey itself is an adventure, and you have been shaped by it. Now, when you make games, some portion of these practices will come in handy, and then you will be able to relate. It may be while making your first game, maybe the tenth one, we don't know. But it is bound to happen, that's for sure. This exercise will make you a much more competent person when creating games!

Making games is an incredible journey. It has its own origin story. We shared ours. What about you? What's your origin story? The expedition also requires preparation. In the beginning, you have to watch others closely and with attention. In our case, we played the games and dissected them in our minds. We showed you some ways regarding that as well. We hope that you have found your purpose in the journey. If that's the case…

Let's start making some games, shall we?

CHAPTER 3

Don’t Reinvent the Wheel

Now that you have decided to become a game maker, let's talk about the next steps. The baby steps first. In this chapter, we will talk about the following:

- Where to start
- Common mistakes
- Misconceptions
- Tools to use
- Code language to use

Let’s Jog The Memory

You have probably heard of *Super Mario Bros*. Thinking about the third-generation video game legend probably gives you goosebumps. That’s okay and very normal. But we see some people taking the inspiration too literally. We know that the size of that game was 32 kilobytes. 32KB can store roughly 2 seconds of MP3 audio. But somehow the game developers were able to squeeze the whole game into that 32KB of storage. However, if you download a *Super Mario* image from the Internet, you will notice that its size is more than 32KB. How did they make it happen back then?

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_3

First-few generation of games were developed using assembly language, which talks with registers and memories directly. That's why it's much more efficient than other methods. But assembly language is not very human-readable. Did you ever ask yourself why they chose assembly language back then? It was used back then because there were no alternatives to the hardware limitations. Back then a gigabyte was probably a myth. So, they **had to** use assembly language to make a game. There was no other way!

Now we have more memory at our disposal and we have human-friendly languages like C, C++, C#, Java, etc. Even though assembly will always be significantly more performant for game development, it's not a wise decision now. The reason is simple. Developing a game with assembly language is not cost-effective anymore because we want to develop a game and we want it now. To do that, we must use all the tools we have now. Over the last few decades, game developers all over the world have created lots of tools to make game development easier. One of the tools is the programming language itself. Assembly may give you superb control over memory allocation but to do that you need to invest a lot of time. And time is money. If you can save a significant amount of time (20x-30x) by using a human-friendly language over a machine-friendly language, why shouldn't you do so? And the performance you are worried about would change about only 1-2%. Moreover, it's always better to use the latest technology. You can certainly water your garden with a bucket, but a hose will save you a lot of time and pain. That's why they invented the hose in the first place!

Since we don't have the memory limitation anymore, we can focus on adding new features to the games instead of looking for coding tricks to fit the game into a small size. Ultimately all of the code you write will be converted to assembly and then machine code anyway. Machines only understand bits 0 and 1. They don't understand assembly language, let alone human-friendly languages. To make machines understand our code, there's something called the compiler. The job of the compiler is to convert our code into bits and bytes. Each language has its own compiler.

They all do more or less the same thing, which is to strip it into assembly language and then machine code. However, some languages only convert themselves into ILs (intermediate languages) and then run it on some kind of virtual machine. You may wonder what a virtual machine is. It is basically an emulation of a computer system. We won't go into details about how it works but it's like a small contained computer inside your computer which exists virtually only. It does use the resources from your hardware, though.

Game-Making Tools

Thanks to the inventions, we now have lots of tools to make our lives easier. Over the last few decades, programmers all around the world have developed lots of tools for the sake of game development. Some common tools are the following:

- Graphic engines
- Physics engines
- Fluid simulation
- Particle simulation
- Path finding
- Auto-rigging and animations

These are some of the tools that every game developer needs. And instead of having them scattered as different tools, they are combined into a single one called a **game engine.** All game engines offer more or less the same set of features, just with different implementations and enhancements. Game engines aren't limited to only the specific features mentioned above. A game engine is a game developer's primary tool to develop games. But the developer may need help from other external tools nonetheless.

We have seen that newcomers often like to take on a lot of challenges. They like to solve a lot of problems. Sometimes they actually start developing tools that already exist in the market because it's fun. We do agree that it's fun and it has a lot of educational value. It does help you understand how a game engine works behind the scenes. But it's wise to keep yourself in check. Often we waste too much time on developing a simple system that is already available, such as the input system or movement system. Yeah, somebody ought to do it, but does it have to be you? What's your purpose? If it's making a game, it's best we stuck to it. To be a professional game developer, it is very important to learn when to stop—just like hitting the breaks in a racing game.

Graphics Engine

A graphic engine is the very first component of a game engine. It's almost pointless to have a game engine that doesn't come with a graphics engine. It's the part where we tell the screen to render pixels.

There are a few common libraries that are used to do this. The common and standard ones are the following:

- OpenGL
- DirectX
- Vulcan

All three offer the functionality required for drawing pixels on the screen. It's possible to use these libraries directly and write your own graphic drawing code. But it's very tedious. Instead, it's wise to use the graphic engine provided by the game engine itself and extend it if necessary.

Physics Engine

Almost every game needs a physics engine. What does it do? In short, it does the physics calculations. It handles a lot of features but we'll focus on the most easy-to-understand ones:

- Collision detection
- Velocity calculation
- Force calculation
- Buoyancy effect

Let's give an example. You want to develop a golf game. A golf ball is placed on a surface. Here you need collision detection; otherwise, the ball would go through the surface due to gravity. Speaking of gravity, it's also a part of physics calculation. Just how much force does a golf ball put on the ground? What happens when the player hits the ball? How much force does it apply? At what angle? How far will the ball rise up and then go down? All these motions are based on the laws of physics. And yes, math.

If you were to do all the math necessary every time you wanted to develop a game, you wouldn't get very far. Because you would have to learn about the laws of physics and the equations. And you would have to validate that it works. All of this before even implementing them in the game! It's very easy to get sidetracked while working with fun mechanics. You need to be very careful about your objectives.

Pathfinding

Another common example is the pathfinding system. Almost all games need A.I.

Any game that has movement needs pathfinding for its A.I. So what is pathfinding? We'll try to explain this as easily as possible. Take a look at Figure 3-1.

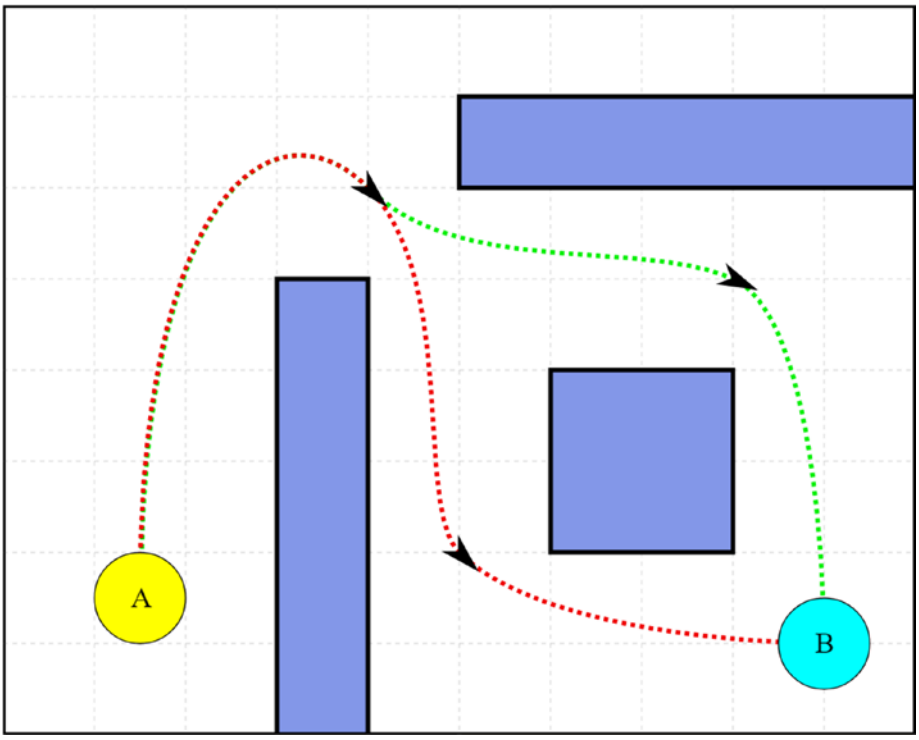

Figure 3-1. *Pathfinding*

So we want to go from point **A** to **B**. Sounds simple, right? Not really. We don't just want to go from point A to B; we want to avoid any obstacles on our way, too. Not only that, but we want to find the shortest path as well. So how do we do that? What is an obstacle? How does our A.I. even know what obstacles are? This is what pathfinding does for us. We just have to mark the obstacles and the system will compute one or more logical paths that lead from point A to B. Then we can tell our AI character to follow that path.

Why is this necessary? Remember those point-and-click games where you tap on a location and your character(s) walk to that position? They don't walk over obstacles, do they? That's an implementation of pathfinding. The AI enemies that chase you in-game also have a form of

pathfinding. There are actually a lot of algorithms developed to do that for you. It's wise to use them instead of designing new ones from scratch.

It's sort of an advanced feature. More or less every good game engine has this feature built in or there's a plugin you can use. Sometimes you don't know that it exists or sometimes you just want to develop your own system because you feel like it. It's not wise to develop such a complicated system on your own if you plan to commercially launch your game. It's very likely that the first time you develop such a system you will have lots of bugs. To fix them, you will have to invest more time, not to mention the amount of time you will need to study to understand how it works and how to make it and find which algorithm is more optimized and when. Why not use an existing system that's been tested by developers around the world?

It's always a good practice to start working with tools that exist. Most of the time existing tools will be more than enough to suit your needs. If you really want to develop your own system, you should start with graph theories. And here's a small list of algorithms that will very good learning material:

- Dijkstra's Algorithm
- A* Search Algorithm
- D* Algorithm

Auto-Rigging

Now comes the auto-rigging and animations part. What is rigging? It is adding a skeleton to the model. Why do we need a skeleton? We need a skeleton to make it animation easy. Animating a model is no easy feat. A 3D character model has a lot of parts. For example, if we take a humanoid character model, what do we get?

- Head
- Shoulder

- Torso
- Arms
- Legs

For simplicity, let's not break down components such as hands and fingers.

Okay, now let's list the basic animations almost every character will have:

- Idle
- Walk
- Run
- Jump
- Punch

These animations will look the same for almost every character. So should we animate every character every time? No, that would be a significant waste of time. Hence we make a skeleton and animate it and attach a 3D model mesh to that skeleton. This way we can animate without even worrying about the 3D model.

So what's the common mistake here? This is mostly for animators. They tend to rig their models on their own and animate their generic characters from scratch. But there are tools that make things much simpler. For example, there's something called an auto-rigger that can rig the model by itself. It's not perfect, but it tries to do its best. If it's not up to your mark, you can always fix the issues and start from where the system couldn't fix your problems. This way you can save a lot of time. For animations, it's even easier: once the model is rigged, you can use already existing animations, paid or unpaid. If you don't like some parts of the animation, you can most certainly change them manually. The common misconception about using these tools is they take away

creativity. But using tools doesn't hamper your creativity. Rather it assists you in streamlining generic workflows. It saves you from wasting time on repetitive work so that you can focus on new things that the tools cannot provide.

At some point in the future, you may need to develop your own game engine, but as a beginner, you need to learn to use existing game engines. As a programmer, sometimes it's very tempting to code everything from scratch. We know it feels great to see your game objects fall down adhering to gravity and collide with the floor. But that could mean committing to thousands of hours of coding to develop a system that has been developed by hundreds of developers over the last few decades. Even that isn't perfect, by the way. It is important that we understand our limitations and accept them. Because we are all human. There's no such thing as a perfect tool. More and more features are being developed each day and they get better each day. We all respect vintage tools. But just because you respect or worship something, it doesn't mean you should use it blindly. They have their own place. You have to adapt to the new world and the new requirements it holds. It's your choice if you still choose assembly language to develop a game, but it doesn't have any advantage over anything now. Some people also develop console games in C. It's fun to play with, and it will probably earn you an A in the lab, but commercially it doesn't have any value anymore.

Everybody wants to plug and play now, and to satisfy the hunger of the large game community, game engines are here. The expectation levels of gamers are just too high and complicated to be fulfilled by those old tools that were used 40 years ago. So let's leave the gods in their place and respect them. Their magic ruled the world once. Let's tell and cherish those tales. But it's time for something else now.

Choosing a Programming Language

Now that you have decided to use a game engine, let's get back to choosing a programming language. There are two ways you can go about it.

1. Pick a game engine and learn the language it supports or recommends.
2. Check which game engines support the language you prefer the most.

Both ways have pros and cons. Choose which one suits you the most. If you want to get a job, the first way is better because most companies look for developers with certain game engine skills. So you need to focus more on a certain game engine than the language itself. For example, companies looking for Unity 3D (one of the most popular game engines right now) skills need C# programmers and companies looking for Unreal Engine skills look for C++ skills. The cons are that you may need to learn a totally different language for onboarding. Also, you may not be able to switch your work stack easily. For example, if you want to switch to web development at some point in the future, you can't do that if you only know C++. But for C#, you can do that easily.

The second way is preferable for those who want to get into game development as quickly as possible. If you wish to develop a hobby project and want to do something with what you already know, this would be more suitable for you. For example, if you are a JavaScript programmer, you can pick a game engine that supports JS. This way you can quickly start working on something. But you are risking your job security a bit because most of the time recruiters are looking for specific game engine developers to suit their needs.

If you are still in doubt, our recommendation is to learn C# because most game engines support this language. It's a cross-platform language and there are other development stacks for it. You can switch your work stack easily if the day ever arises. Having an exit plan is a good idea.

Let's not forget those drag-and-drop game engines. There are quite a few that are extremely popular like Gamemaker, Construct 2 and 3, and Buildbox. You might have seen their ad ("Make a game! No programming languages needed!") and dismissed them because you are a great programmer. Why would you work with these simple tools? Well, friend, I think you might want to reconsider. A game is a mixture of art, mechanics, sound, story, and many other things. As long as people have a great experience and everything is working perfectly, no one cares which engine or language was used. Many gamers will die after decades of gaming before ever learning the name of a single game-making tool, and that's only normal. Programming languages, tools, and software are just tools used to make games. We use them because the game needs some help in some segments and they make the job easier. They are secondary. They depend on the game; the game shouldn't depend on them. Let go of the stereotypes. Think of the game, what it needs, and the most efficient way to feed these needs. Go for it. Focus on the core game, not the tools. A generic tool that offers some template code libraries to generate template games can hardly match your creativity. You will eventually hit a wall where you need to code your feature. Congratulations when you do that; you have basically leveled up to the next level. It is important to understand the need for a tool before using it. Otherwise, you may take its features for granted. Observing template code may actually help you learn new programming styles. There's always something you can learn.

Summary

Before you do anything, always search to see if the solution already exists. Then see if the solution solves all your needs. If not, can you adjust your needs? Sometimes small compromises will be necessary. You may have to pivot your idea slightly (1-2%). Then use that tool. But if you find that you

have to pivot too much, then you have reached the point where you have to develop your own tools for your game. It's wise to let existing tools do the heavy lifting so that you can focus on the unique parts of your game that cannot be managed by existing tools.

Now that you understand which path to choose, it's time to pick a game engine for yourself. It will be a very decisive decision for you. We are going to talk about this in detail in the next chapter.

CHAPTER 4

Choose Your Arsenal

What's the last role-playing game (RPG) you played? Was it the recent *Final Fantasy* remake or maybe the *Witcher*? You're probably thinking about *Mass Effect*. Well, if you're swelled by memories, you must remember all those character upgrades you took. What are you good at? Are you a sharpshooter? If so, you must love a bow and an arrow. Are you a warrior whose fantasies feature swords? Maybe you are intimidated by magic and hence you always choose the magical upgrades so that you can take your enemies down with a spell that brings the thunder. Each one of us has our own type, right? Still, we all end up finishing the game, as long as we keep playing well in our own way. The journey of a game maker is no different. There are different tools out there; in this context, they're game engines. Each one is for a certain category of people at a certain stage.

By now you have already decided to use a game engine. In this chapter, we will talk about the following:

- The first mistake
- Existing game engines
- Their pros and cons

The First Mistake

Choosing a game engine is one of the very first steps you will take in your game development career. And this is exactly why this is the very first possible mistake you will make. Every task requires proper tools. Each tool

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_4

has its own usages. You can't use a wrench where you need a hammer. You can certainly whack the nail with anything you can find and get your job done, but that's not efficient. You need to know what you are going to do and the tools to use to achieve your goal.

For game development, there are a lot of tools you can use. Their base is the same yet they differ slightly from each other. And that small difference can make a major difference in reaching your goal.

Before we start talking about the details of specific game engines, let's think about the mistakes we can make on a personal level when choosing them.

Case 1: You are new to game development. You picked a random game engine and started to learn how it works. After developing your first game, you realize you have to pay a licensing fee if you wish to make your game public.

Case 2: You want to develop a photorealistic 3D game. After fiddling with a game engine for a few days, you realize that this engine was not targeted for a photorealistic 3D game.

Case 3: After getting your game to the half-way point, you realized that the game engine itself isn't complete. It's lacking a lot of features.

Case 4: After developing much of your game, you realized that the community of this engine is very small. As a result, you can't find a lot of resources.

Case 5: The engine is only good for developing a fixed type of game. There's no extensibility. You get stuck with a few generic ideas.

All of these cases can be avoided if proper research is done before choosing the engine. To kickstart your research, we are going to talk about the most popular game engines here. Here's the list of game engines we will talk about:

- Unity 3D
- Unreal Engine
- Godot

Unity

Unity is the most popular choice for any indie developer. One of the reasons is because it's one of the first free game engines available in the market. Most good game engines had to be purchased for a hefty amount back then. As a result, Unity grew in popularity very quickly. The current rise of indie games on the market has a lot to do with Unity 3D.

Unity's true power lies in its community. The asset store is filled with game development tools that you will most likely need at some point in your game development career.

In terms of capability, Unity is a very powerful game engine. It also accommodates a wide range of categories in terms of creators. You can develop games starting from low-end mobile to AAA games. It's friendly for newcomers and featured enough for the enterprise. However, for the enterprise, sometimes it may not be the out-of-the-box solution, because you must configure quite a lot of features to support the quality you desire. Once configured, it's good to go. Also, you may have to buy some premium plugins or assets, which you might have expected to be built into the engine editor. One of the core reasons it is so popular is its cross-platform support. Unity supports every game platform that exists: Android, iOS, Windows, Mac, Linux, XBOX, PlayStation, Nintendo, you name it. If it exists, Unity most likely will support it. You can just code once and deploy your game to any platform with minimal to no code change at all.

Unity has a generous licensing option for developers. It doesn't ask for royalties from the games you develop using Unity. You fully own the game. If your company is making under a threshold, you don't have to pay a dime. If you earn more than the threshold, then you have to pay a monthly fee. Depending on how much you earn, you may have to pay less or more, which sounds fair, right? For someone who's just getting into game development, this is a great starting point for free.

Here's a summary of all the pros and cons of Unity.

Pros:

- Massive community support
- Different render pipelines for different platforms
- Free license
- Full ownership of the final game product

Cons:

- AAA games need a lot of initial setup time.
- You may need to buy some tools from the asset store.

Games Made with Unity

The next sections cover some games made with Unity and short notes about why Unity was used.

Ori and the Blind Forest

This game was a major hit of 2015. It's a 2D action platformer that reminded everyone why games are the best kind of artwork. This game was developed by an indie studio called Moon Studios (later acquired by Microsoft Studios). For developing a 2D game at that time (and even now), Unity had almost no rivals in terms of the features it offered. The developers were able to port and publish the game onto multiple stores quickly because of Unity's build support.

Cuphead

2017's talk of the town, *Cuphead* was made with Unity by another indie game studio called Studio MDHR. It was very well known because of its aesthetics, which were possible because of the sprite sheet animation support in Unity. The whole game was made using hand-drawn animations.

HearthStone

This is a game from one of the giants, Blizzard. They usually develop their own game engine for their games. But for Hearthstone they used Unity. They admitted that the Unity game engine made their work fast. They didn't have to hire a lot of employees for the project. They only had to use a small team of 15 people for the game. Usually, their teams were two to three times larger than that.

Call of Duty: Mobile

2019's mobile multiplayer hit was from another giant, Activision. They chose to develop their game using Unity for its lower-end device support since they were aiming for mass users. They too use their own engines when they develop games but they chose to use Unity for its convenient features.

Comments

Unity can be your starter pack for game development. It is one of the best game engines if you are targeting 2D. Even well-known publishers choose Unity for a lot of games. If in doubt, get started with Unity.

Unreal Engine

Unreal should be your first choice if you are planning to develop a high-budget AAA title. If you are targeting photorealistic graphics, this is your out-of-the-box solution. It has all the features you will need to develop a game from scratch without buying a third-party library to support your development.

C++ is used as the scripting language for Unreal Engine. It is the most powerful, human-friendly language you can use to develop a game. In terms of

memory management, the only rival for C++ is assembly, which isn't human friendly at all. Even if there are other languages such as C# or Java that are much more human-friendly, they lack the control of memory management, which is very crucial for game development. So choosing C++ over C# as Unreal Engine's primary development language is understandable.

Unreal Engine has gone a step forward and developed a very powerful visual scripting tool called The Blueprint. Even without knowing a programming language, you can dive into game development with its help. It's very useful for non-coder people to dive into game development. And if you do want to write your own code (and at some point you will definitely want to), you can use the raw C++.

Like all other prominent game engines, it supports building on multiple platforms like mobile, web, desktop, PlayStation, XBOX, and Nintendo. However, it must be pointed out that, unlike other game engines, it only targets high-end mobile devices. If you are planning to develop games for everyone, this is probably not a good choice.

Another point to be noted is that to start developing with Unreal Engine 4, you will need very decent hardware with a decent GPU. This could get expensive for educational purposes.

Licensing is a bit complicated. If you develop a free game, you don't have to pay for anything. But if you plan to publish your game, you must pay a percentage of royalties (5%) if your gross revenue exceeds a certain threshold per quarter. These charges depend on your product's success and may become unacceptable later.

Here's a summary of all the pros and cons of Unreal Engine 4.

Pros:

- Free projects are totally free.
- A royalty-based licensing for monetized games
- Off-the-shelf, state-of-the-art graphics

- All the tools you can think of are built in.
- Powerful visual scripting tools

Cons:

- Doesn't support low-end devices
- Community isn't as big as Unity's
- Supports only 64 bits for development
- Requires expensive hardware for development

Games Made with Unreal Engine

Here's a list of games made with Unreal and short notes about why Unreal was used.

Fortnite

Unless you have been living under a rock, you have heard of *Fortnite*. Even though Unreal Engine's core power lies in its photorealistic graphics, they chose to develop a low-poly battle royale game. This was mostly to show off that Unreal Engine can be used for low-poly games too. And then it clicked in the market. The game you know now was just a spin-off mod from their original *Survival* PVE game.

Final Fantasy VII Remake

In 2020, a lot of games were remade. But none of them were as desired as *FFVII*. The original game was first released on PlayStation. That was the dawn of a new era. When Square Enix decided to remake the game for PS4 in a new flavor, they chose to use Unreal Engine 4 for its stunning graphics. And they did win back the hearts of millions. They left everyone in awe in 2020, just like they did in 1997 with the PlayStation 1 original.

Comments

Unreal Engine is your enterprise solution. It's made for heavy lifting. It's reliable for any AAA game. If you want to captivate your players with graphics, this is the way to go. However, if you are planning to develop mobile games, this is not your tool. Using Unreal Engine to develop mobile games is like trying to light a cigarette with a flame-thrower.

Godot

Godot is very similar to Unity, but it's open source. It is a new game engine that has the capabilities to rival Unity Engine in terms of features. Maybe in two to three years it will be a solid alternative to Unity. But the main reason you should choose it is for its open-source licensing. The whole engine code is open. You don't have to pay anyone anything for what you develop with it. Period.

Language support is the most interesting part of this engine. Unlike other engines, it supports both C++ and C# as well as its own native scripting language officially. Unofficially you can use Rust, D, Python, and even Go.

Since it's new to the market, it uses bleeding edge technologies. As a result, there are already some features available in Godot that are currently in development for Unity.

Sadly, no well-known games have yet been developed using Godot. It's still being used in a hobbyist state. But maybe after a few years of polishing, it will become the entry point for students and newcomers. At the moment, the UI isn't very charming. And according to a lot of users, the UX is a bit complicated for newcomers. It's not very friendly for those who don't know about game development at all.

Here's a summary of the pros and cons of Godot.

Pros:

- Free license no matter what you do
- Lots of language support

Cons:

- Complex UX for newcomers
- Doesn't look good out of the box

Comments

Godot is an aspiring alternative to Unity. But it's not yet recommended for newcomers. If you really have issues with the licensing of other engines, check out this option.

Verdict

Among the three engines we talked about, Unity is the most recommended general-purpose solution. If you need more graphical quality for your games, you can choose Unreal but it's still possible with Unity. And if you want something completely open-source, Godot is the way to go. Remember, in the RPG, if you get the stealth options, you probably can't get strength options and vice versa. Also, there's no point choosing bits from every path whenever you can; rather, it's better to be the master of one path. Being 20% ninja, 10% brute, 30% swordsman, 20% mage, 20% shooter will not help you much in the big boss battles that come in the later portion of the game. Similarly, knowing bits from a number of game engines won't be the wise way to start your journey. Instead, if you just be efficient at one, you can do wonders. You choose one engine, you utilize its

potential, and you can go a long way. Choose what suits you. If you can't decide, just pick one. You are welcome to take our suggestions. Just get the ball rolling and focus on the game. You are on a path. Don't look back every time you hit a hurdle. Find a way to deal with them. Because that's what we do: we solve problems.

CHAPTER 5

It's All in My Head: Writing a Game Design Document

Our brain is a fascinating substance. We can store so much information at a time into it and process so many problems. It's like a massive supercomputer in terms of capability. So it's pretty reasonable that we put our faith in it in terms of storing our thoughts as we think about magnificent creations in our heads.

Except it isn't.

Our brain may hold the unique power we have been blessed with, but it too has limitations. We all get it; trust us. It's the first reflex we naturally have as soon as we have something cooking in that beautiful brain of ours. "If it can think of something this brilliant, it can, of course, keep it safe. I will extract it when necessary," the voice tells you. That, my friend, is the trap you have to learn to avoid. It's tempting; it's subtle; it's manipulative. Sadly, the harder it is to sense an enemy, the more challenging it is to beat that enemy. The only way you can break this particular anomaly is by consistency. You have to make a habit of documenting anything you see. It's for your own protection. In this chapter, you will discover what documents we are talking about and how to avoid the clever pitfalls you may face on the journey while making them.

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_5

The breeding process of a game consists more of creativity and less of everything else. There are so many activities that are exciting in nature. Designing the beautiful game world, character buildup, writing a compelling narrative, mixing some music that soothes both your ears and heart, coding a cool new mechanism that people have never seen before; you can get lost in the long list of fun work that happens during the birth of a game. So, it's kind of tempting to set aside the comparatively tedious tasks like documenting your creative thoughts and keep yourself more engaged in the fun activities.

Let's skip the theories for a bit and talk about one of your favorite games. What was your favorite game of 2018? Was it the best game of the year, *God of War*? Mind-blowing graphics, intense story, and fantastic gameplay; it had everything, right? It felt so good fighting gods and monsters with Kratos and Arteus in all the realms. The combos in the fight were so pleasing to perform, right? Let's assume that you last played the game two years ago. If you were to play it again and face one of the most challenging foes (say, a corrupt Valkyrie) in the game, it would be a piece of cake for you, right? Indeed, you have every combo in your head, and you can take it head-on. Oh wait, are you hesitating a bit? Come on, the blades of chaos are a powerful weapon, and you can definitely slay with them. Oh, I see. Just slaying randomly won't get you past the advanced foes, let alone the tough bosses.

You need combos to win against them, and a lot of other legendary items. Items, combos… seems like you don't remember them as well as you did two/three years back, do you? No worries; you can always look at the guide from the menu and find the combo combinations and the necessary items you ought to use after looking up the detailed weakness about the enemy from the Beastiary (inventory of enemies). Well, that's weird. A few moments ago, you thought that you could put every tiny detail about a game that did not exist into your head, and now you can't remember some basic details of a game that you played for over 50 hours not so long ago. So much for the almighty brain, eh?

Maybe our brain cannot always store everything for us in a way that is usable all the time. A game can contain so much information that the need to store them systematically is almost intuitive. This is where the game design document (GDD) comes in.

Introducing the GDD

The GDD contains all of the ideas a game has and every stage of the development in detail. The lifecycle of this thing should be the same as the game itself. It starts with chalking out the initial concepts and goes into a detailed list of features and then development cycles. There's no concrete rule about which exact components it should have, but the idea is to convey the concept of the game and regular updates in it, which can be used as a reference for all parties involved. Now, the definition of all parties can be tricky as well. A complete GDD can serve all the people involved in the development process. It's not limited to developers only; the other parties of the process (investors, partners) can access it, too. But that's not something that happens regularly. The developers, artists, designers—all of them will contribute to the making of it. It is a live document and a map of the entire game, and it can get everyone onboard with a unified vision. We will mention as many components of the GDD as we can. You can try different combinations of them or use all of them.

- **Philosophy**: Discuss the core of the game. What is it about? How did you come across it? What do you want to say? What does the game achieve?
- **Genre**: What kind of a game is it? Is it a board game, card game, sports, adventure, puzzle, or something else?

- **Targeted audience**: Who will play your game? What is the age group that you're targeting? What do they do? What kind of gamers are they, and what is their playing pattern?
- **Targeted system/platform**: Which platform do you want to release it on? What is the release plan? Which systems will be able to run this game? Will it run on older systems as well? Are you planning for some new versions of systems?
- **Plot and story**: Discuss the plot briefly here, include your inspirations. Write basic story outlines as well. As you progress, write out the detailed story or attach the supporting story documents.
- **Characters**: Who are the characters in the game? Some are PCs (playable characters), those whom you can control. The others are NPCs (non-playable characters), whom the gamer can't control.
- **Feature list**: Go into details about the features of your game. What are you playing? How are you playing it? What are the things that happen when you play? Describe the basic and advanced features of the game here.
- **Gameworld**: Where does the game take place? What does it look like? What are the elements in the environment? What are the weather conditions? Which components are interactable, and how can you use them?

- **Development ideas**: How can you create a game out of these plans? Are you using a game engine? If yes, which one and why? If you have chosen one, how do you think the features will be implemented? Will you be using any readymade templates, or are you starting from scratch? State your initial plans and keep developing them gradually.
- **Visual examples**: This portion is significant because it lets the readers properly visualize the game you have in your mind. Begin with concept art, and go along adding assets to describe your features, mechanics, and game world.
- **Overall aesthetics**: This deals with the general tone of the game. What sort of impression do you want the gamer to have about the overall environment and ambiance? In this portion, describe the nature of the whole experience in terms of all the assets, sounds, and mechanics.
- **Gameplay mechanics/systems**: Describe your game mechanics. What are the controls, and what do you do with them? Is it a level-based game? If yes, then how do you move up the levels? How do you overcome the challenges, and what rewards await you upon success? Discuss the unique experiences the player will encounter and how the overall system will support them.

- **Sound**: Some prefer this to be a part of the asset segment, but many like this to be a separate segment where you talk about SFX and game music. You can start by listing the items that are needed, the ideas behind them, and how you want them to be. Then you can document ideas about sourcing them. Update the document as soon as you make any progress, as usual.
- **Team**: List the team members. Some people even include their contact details in the GDD. As soon as any change happens in the lineup, update the list. Please make sure you have listed their designation and which team they belong to as well.
- **Funding, marketing, and sales**: This is an optional segment. If you have any ideas regarding the financing of your project and how to market it after publishing, you can keep a tab of them here. Gradually you can verify your previous estimations, learn from them, and make better decisions later on. Many don't include this segment in the GDD, but many indie teams do.

You can use all of these segments or use some of these points to make your own. We'll share a demo GDD here, consisting of a few elements we mentioned.

Game Design: Mudwash

Philosophy

I love rain and thunder. I tend to leave whatever I am doing and watch it all as soon as the thunder rumbles and the clouds close in. I am a big fan of Norse mythology as well. Hence, I adore the god of thunder, Thor,

and his heroics. In the recent movies by MCU (*Thor Ragnarok*, *Avengers: Infinity War*, and *Avengers: Endgame*), the heroics by him have been quite stunning. The scenes where he takes out minion hordes are so pleasing. This inspired me to think of a hero who does a similar kind of action. I choose to create a descendant of the Norse god.

The only thing I hate about rain is that it creates mud, which feels depressing to me. Also, in many of the myths I grew up reading, there are monsters in the creeks. So I made the only thing I hate about rain (mud) into the antagonist. The game takes place in my favorite setting: an atmosphere of rain, clouds, and storms.

Mud creeks have always been sources of great myths. The mud monsters were said to be buried away in their hives until an earthquake makes them resurface. They come out of their hives and try to kill the living beings by suffocating them in the mud and hence grow the muddy environment. The player, being the descendant of the god of thunder, arrives to protect humanity and nature from this invasion. They use their power of bringing the thunder to destroy the monsters. They also have the superpower/special ability to bring rain. Rain can dissolve many monsters at a time and damage the hives. The goal of the player is to get rid of the small hives and reveal the mother hive. As soon as they destroy the mother hive, the level is passed because the area is free from monsters. They have to finish the levels one by one and free Uptown from these mud monsters once and for all.

Feature List

Uptown is infested with clay monsters. With the power of rain and thunder, you as the player must vanquish the hordes of minions made with clay. There will be many areas to clear one by one as you level up. The game takes place in the different lakes of Uptown. The long-asleep clay monsters have awoken, and you must take them out one by one and clear the mother hive in each level, which is highly guarded and only becomes

visible after you kill a certain number of clay monsters. You, being the descendant of the god of thunder, can attack them using your power of thunder and rain. You can grow your energy by having a kill streak to make yourself capable of using special abilities like a thunder strike or super rain. But you risk harming civilians while using these extraordinary powers, so you have to be careful about the timing and placement while using them. The monsters will try to cover you with mud and drown you, so you can't let many of them touch you for long, or else you will die.

Gameworld

The action takes place around lakes of different shapes, where monsters created of mud get born out of invisible hives. In a lake, many hives are floating around. The monsters are born from these hives and rush towards you, the hero. You move around the lake, destroying them one by one while saving yourself and the people around you from their attack. The weather is gloomy, and the roads are all awash with mud and water. Nearby structures have a grim shade over them, as if the mud has infected them as well. The only birds visible at times are crows. You can bring thunder and rain as your superpower.

You are a descendant of the god of thunder. Hence you have the power of thunder. You can bring the thunder with your bare hands and call for huge bolts of thunder in different places as your special ability. Additionally, you can bring rain as a special ability. You had a rough past of fighting with monsters and aliens, making you a true warrior. You have no extra emotion when you are on a mission. You stay focused on your duty of protecting the world of the living. You must be careful that you don't end up hurting innocent people while you are on a mission because your powers are divine and deadly. You stay careful so that there's no collateral damage. If any innocent lives are affected by your powers, your ability to use them gets hampered. You are ruthless against your enemies.

Your antagonists are the mud monsters, who are pretty much brainless (like zombies). They come in hordes and try to decimate everything in their way. Their only approach is to cover living creatures with mud and kill them to spread the reign of mud everywhere. They come in all shapes, and most of them have multiple eyes.

The other characters are pretty much standby. The people are mainly defenseless and depend on you, the protagonist, to save them from the mud monsters. They seem to move randomly on the map, appearing only to make the game more challenging for you. The birds and other animals are bystanders as well.

Development Ideas

As this is a level-based game, you have to go level by level. Once you code one level, the rest should be easy to replicate because the main functions will be the same. The enemies will get more challenging to handle, and they will grow in number as well. Unity 3D can be used as the game engine because it is simple enough considering the human resources you will work with, and it should serve the purpose quite well.

Overall Aesthetics

The game has a classic feel of light vs. dark. The hero is the bringer of light and lightens everything up with his power of thunder. The mud monsters are all about darkness, starting with their color and their activities. The game has a setting of a cloudy environment; hence it will be pretty dark out there. The audio will be close to reality as it will resonate more this way. The sound of mud thumping when something falls in it will be heard as they try to drag elements of the game inside. The sound of rain and thunder has to be the original to initiate the proper feeling. The visuals of the hero in action should be robust. It has to have the sense of power and dominance of a god in it.

Gameplay Mechanics and Systems

The game is a level-based saga of missions that get tougher as the game progress. The mud hives are all over the lake. In the beginning, there are only a few in a level, but as the player moves to upper levels, their numbers grow, as well as the spawn rate of the monsters. The location of the mother hive in a level is not revealed until the player kills 80% of the hives in that. The mother hive generates small hives once in a while, and she can increase her reproduction rate after the player destroy 90% of the hives. So the player has to act fast while finishing it off. The player will move along the paths and sides of the lake to kill off the monsters and their hives. The player's primary weapon is the bolts of thunder generated from his bare hands and body. After he has achieved a certain kill streak, the superpower abilities will be ready. They can bring down vast bolts of thunder from the sky to destroy monsters and hives around the map. But thunderbolts can also affect other living creatures, and killing them will demoralize your hero. They can also bring down rain, which can destroy hordes of monsters at a time but not kill innocent people. But to achieve this power, they must play well consistently.

There are 10 levels in total. As the player moves up the levels, the spawn rate of monsters and hives increases. The time duration of people staying on the screen also increases. In the final level, there's more than one mother hive, making it challenging to finish the game because one mother hive supports the other.

Encountered by too many monsters at a time, the protagonist can be taken down by them, so the player has to make sure that they can't come close. There's a health bar that will start depleting as the monsters touch the player and try to root them to the spot and cover them with mud.

The game offers gloomy and dangerous weather and dark enemies. They may be dumb, but their production system will give the protagonist a run for it.

Working with GDDs

Now that you have read a GDD, what do you think of it? That wasn't difficult, was it? There are misconceptions regarding the task of making a game design document. Many people spend hours and days looking for the perfect template. Sadly, there isn't one. The more detailed you can be, the better it is. But there's no hard-and-fast rule. Also, we often see many first-timers thinking they'll make the GDD first and then build the game. Well, it's a two-way process. You should update your document when there's a change or update in the plan. The GDD should have continuous updates tagged along with the development progress. The design document will have many versions, and with each iteration, you will have an updated picture of the whole map of the game. Now that you have read the sample document we have given here, what is your first impression?

It should be a very early iteration, right? This is just the beginning. As soon as you make progress in the making of the game, you should update the game design document. For the initial iteration, if you want to detail things out a bit more, you can write a high concept note where you discuss the game concept mainly. You can write separate documents for game plots and development plans initially. But eventually, it's good to have the central GDD updated all the time. The supporting documents can be more detailed. Also, another shortcoming of this sample GDD is that there were no visual references included, making it hard for the readers to relate. Enrich your document with sketches. Be it raw, unedited, or simple; it doesn't matter. You should have as many sketches as possible. The character sketches, environment design, UI; you can do absolutely anything here. It's not mandatory that the sketches you make now are used in the game. If things turn thus, consider it as an added benefit. But it's totally fine if that doesn't happen. The primary purpose of these sketches is to get you started on visualizing the game.

There is another issue we have seen happen. After a few iterations, when things start detailing out, it can be challenging to document everything. To address this, we suggest making it a habit to document whenever progress happens. You should have dedicated people working on the GDD if possible. If not, at least make it a rule that whenever some update/progress happens, it can't be approved without a written document. This is one way of enforcing the habit of updating regularly. But like all other things, it's not mandatory. You can find your own solution.

Another misconception is that the GDD is updated by game writers/designers. You should give the ability to change the document to whoever is involved in the progress of it. It is recommended to have people from each team contributing collaboratively. Otherwise, it's difficult to move at a reasonable speed. You can have your expert writer edit the document once in a while to flesh things out a bit more or make it look better, but editing the GDD should be a collaborative exercise in the first place.

Let's have a look at another game design document now. It's just another sample.

Game Design Document: Endora- Relics of The Ancients

This game design document describes the details for an open-world, third-person action RPG game with solid mechanics and an original story and characters. This game will be based on the PC platform.

Characters

The main character of this game is **Endora**, a cheerful, charismatic girl who has set off on a journey to become the strongest wizard in the world. With her might, she wants to uproot all evil powers from the world. To truly experience how it is to live among the common folks, she takes an alias (Player Name) in order to hide her noble ancestry.

In her journey, she is accompanied by her childhood friend **Theo**, a joyous little puppy who passed away when Endora was still a child. Later he returned to Endora as a spirit after becoming the king of the puppy realm, promising that he'll stay by her side for eternity. As the king of one of the divine realms, Theo possesses all the knowledge to guide Endora on her journey. Also, he hates cats. (Through Theo, players will be walked through interactive tutorials.)

Story

As a princess of the **Lee** family, a powerful wizard family that rules over the city of **Sunridge**, Endora has been able to observe the oppressive power that cripples the society from a close distance. She has always been skeptical of her father's inability to deal with those who work from the shadows. She believes that only absolute power can put an end to this evil power. Thus she embarks on a journey to become the mightiest of the wizards, accompanied by her best friend, Theo. To become the strongest, she'll need to find out the five Relics of the Ancients.

Theme and Story Progression

This game will have a fun and jolly mood, reinforced by its music and interaction with the NPCs. This game is also about Endora's quest to become the liberator of her people. On her journey to become the strongest, she'll come in contact with people from different walks of life. She'll come to a realization that brute power can't solve all problems; some problems need to be solved with other approaches. Thus she'll start questioning the purpose of her own journey. Throughout her journey, players will see the naive, cheerful Endora grow into a more mature and self-contained version of herself.

Gameplay

Goals

Overall (long-term) goal: Search for the five ancient artifacts, known as the Relics of the Ancients. These relics are supposed to contain a large amount of elemental energy, which will help Endora become stronger.

Gameplay (short-term) goal: Become stronger by leveling up your character, finding better weapons and leveling them up, and finding better rings to get boosts. Story quests and side quests include one or more of the following activities:

- **Clear dungeons:** Defeat the normal enemies. Advance to the bosses. Defeat the bosses to get leads on the search for the relics.
- **Help other NPC:** Carry out tasks given by NPCs.
- **Level Up:** Level up your character, weapons, and rings to become stronger.

Skills

The player can develop two kinds of skills.

- **Base skill:** As the player progresses through the game, they'll get experience points (XP). XP can be used to upgrade their base stats. The base stats include
    - **Base level:** Player level. Other stats can be maxed out to the base level only. If the player wants to increase other stats, they'll have to increase the base level first.
    - **HP:** Maximum health.

    - **Crit rate:** The percentage of critical hits. Critical hits deal extra damage.
    - **Crit DMG:** The amount of extra damage critical hits deal.
    - **Speed:** The speed at which players can run and dash.
    - **Stamina:** Running and dashing consume stamina.
    - **Strength:** Every weapon requires a specific strength requirement.
    - **Intelligence:** Every weapon requires a particular intelligence requirement.
    - **Heroism:** Every weapon requires a specific heroism requirement.
- **Elemental skill:** XP points can also be used for elemental skills. Elemental skills are magical powers gained from elements. There are five elements: Fire, Water, Air, Electro, and Earth.
    - There is a skill tree with five branches. Each branch represents one of the five elements.
    - The player can use XP to unlock skill points and progress through the tree.
    - Regardless of how many elemental skills have been unlocked, the player can equip only three at a time. (In combat, those three skills can be used. Players can swap elemental skills outside combat.)

Items

All items in this game can be categorized into two types: rare items and common items. Rare items have a rarity rating ranging from 1 star to 5 stars. Items with low rarity can be easily found, whereas items with higher rarity are extremely difficult to find. The **rare items** in this game are

- **Weaponry**: Each weapon has a stat requirement (Strength, Intelligence, Heroism). Weapons have a base attack stat and other substats, which can be increased by upgrading weapons. Crafting and upgrading weapons require (1) money, (2) minerals that can be mined from the open world, and (3) diagrams that can be bought or acquired from chests.
    - **Weapons** (sword, greatsword, bow, spear, dagger, axe, and other types of weapons): Weapons have a base attack stat and other substats, which can be increased by upgrading weapons. Weapons can be acquired by opening chests, doing quests, and crafting.
    - **Weapon diagram**: It is used to craft weapons.
    - **Weapon craft/upgrade material**: Used to craft and upgrade weapons.
    - **Runes**: A weapon can be equipped with a rune that can boost the weapon. Runes can be acquired from quests and chests.

- **Rings**: Rings give the player various kinds of boosts and powerups (e.g., increased attack, increased HP, increased elemental power, and so on). Rings can be acquired from chests and by doing quests.
    - A player can equip a maximum of 5 rings at a time.
    - Rings can be upgraded by infusing them with other rings. This requires money.
- **Outfit**: The player can choose different outfits to wear. Outfits can be bought from shops and acquired from chests.
- **Chests**: There are three kinds of chests: common chests, epic chests, and legendary chests.
    - Common and epic chests can be found throughout the game in the open world. These chests can also be acquired by doing quests. Common chests drop 1-3 star items, whereas epic chests can drop 4-star items as well.
    - Legendary chests drop rare 5-star items. Legendary chests can be bought with gems. Gems are to be bought with real-world currency.
- **Gems**: Bought with real-world currency. Used to buy legendary chests.

Common items, on the other hand, do not have a rarity rating. They can be found in the game by doing quests and acquiring chests. The common items in this game are

- **Money**: Money is used for buying items from the shop, crafting and upgrading weapons, making potions, and buying outfits. Money can be acquired by doing quests and opening chests.

- **Herbs**: Herbs are used to make potions. Herbs can be found throughout the open world.
- **Minerals**: Minerals are used to craft and upgrade weapons. They can be mined from the world.

Level Design

Since this is an open-world game, the game will have memorable landmarks and eye-catching populated cities. There will be secret passageways that the player will have to discover. Finding out secret locations and going to difficult places will reward the player with chests and quests, therefore encouraging the player to explore the world.

Controls

The player controls the main character. The player can do the following things:

- **Normal attack**: Left-mouse click
- **Charged attack**: Left-mouse hold
- **Lock enemy**: Right-mouse click
- **Jump**: Spacebar
- **Dash**: Left-Shift press
- **Run**: Left-Shift hold
- **Elemental skill 01**: Press E
- **Elemental skill 02**: Press Q
- **Elemental skill 03**: Press X

There will be key bindings for things like opening a map, opening inventory, and opening a quest menu. Players can also change their control settings.

Progression and Challenge

Here is the list of progression and challenge:

- As the story progresses, the enemies will become stronger.
- To deal with stronger enemies in quests, players will have to level up their character and gear.
- As the player levels up, the enemies in the game level up as well. In other words, the *enemies' level scales with the player's level*. (This is done in order to keep the game challenging.)
- ***If the player dies, 50% of the unused XP will be gone.*** This feature will make the players think more about survivability, thus making them use potions and upgrade their character as well as gears. This will make the game challenging enough to keep the player engaged.

Achievements

When a player does a certain task, they'll unlock an achievement. This system exists to encourage the players to unlock all achievements. Unlocking achievements provides the player with gems, which are used to buy legendary weapons. *(N.B. Unlocking achievements is the only way to earn gems, except for buying gems with actual money.)*

Art Style

The art style will resemble that of *The Legend of Zelda: Breath of the Wild*, with anime-ish character designs. Everything should be very colorful and feel alive, with highly animated scenarios and cinematics.

Music and Sounds

Now for music and sounds:

- For in-game music, we'll use a more relaxed approach. When exploring the world, calm and soothing music will run in the background. Different locations can have music of their own.
- If the player is approaching a dangerous area, the music will become tense.
- When low on health, the music should go up on tempo.
- Sound effects must praise the user when they do something good. There should be immediate and positive feedback.

Technical Description

Initially, the game will be released on the **PC platform**. Then it can be expanded to the console platform if it seems viable.

It could also be released on the mobile platform, although porting such a massive game to mobile would be a daunting task since there are hardware limitations as well as different control quirks.

The following game engines should be considered:
Unreal Engine, Unity.

Monetization

The game will follow a freemium model. The game is free to play. Interested players will be able to **purchase gems with real-world currency**, which can be used to buy legendary chests. (Legendary chests contain rare 5-star items like powerful weapons, rare artifacts, and exotic outfits.)

Again, this GDD is a basic version of GDD. It should be updated as the development goes on in parallel. (This game 'Endora' and its GDD belongs to Mr. Tahsin Tunan from Bangladesh. We thank him wholeheartedly for letting us use his piece as a reference to follow.)

Fragile Memory, Process, and Future

The shortcomings of human memory are arguably not the primary motivation behind documenting a game. Instead, the nature of "game" itself is the main one, we think. When we developed the first game we published in Google Play Store with Battery Low Games, we didn't make a game design document. It was a very simple game, a hyper-casual involving a bubble and some spikes. Even during publishing, there were some leftover bugs. The game was a failed case because we had to take it down due to the bugs. Back then, we didn't have the required expertise to address those. There were four of us and all of us knew pretty much every corner of the game. Naturally, we didn't bother with a design document. "Why would we need one?" we thought. We had the whole game printed in our brain perfectly. It came out of the continuous brainstorming sessions, after all. That kind of thing is hard to forget.

Years later, we had the expertise to tackle the issues that took the game down initially. Sadly, by this time, we couldn't remember what those issues were and what we were trying to achieve. The code was commented on properly, but the code can't tell you about your ambition and plan with the game. We had to guess a lot. There were graphical pieces, but even

with them, the new developer team was pretty much clueless. We were back at square one. See, the mistake we made was not the only one. We thought that we could remember everything, which was foolish. We also considered in the lifetime of the game we would be the only ones to work on the project. This particular thought was downright stupid. Even if it is a failed project, there's always a chance that new people will try to revive it, and they will need a proper guideline.

In this world where everything is fragile (not the one from Hideo Kojima's *Death Stranding*), you can't rely on some particular individual to always serve your art, which is not mortal, by the way! You must establish a process and trust it. Even if you consider the best of the hypothetical scenarios, if a team makes a perfect game in their first release, they will still need to roll out future versions eventually. So, even in the utopian scenario, new people will eventually need a guideline to work on in your game.

While you are working on the game, it can be extremely difficult to keep everyone on the same page. A game design document works as the ultimate reference book for your team. It can reduce a lot of miscommunication and time-wasting. If your game is a religion, it's your holy book. Every religion has one. Your game should have its GDD as well. You should make and maintain it with care. It should be the center of your development and guide for the future rock stars serving under your game.

CHAPTER 6

A Stitch in Time Saves Nine

Now that we are going to move into the development phase, let's talk about some precautions you should take. They may look unimportant at first but, as the project grows, everyone regrets not doing them right from the start. This chapter's takeaway is our greatest mistakes and realizations over the years.

In this chapter, we will talk about the following:

- Project structuring
- Code patterns
- Conventions
- Asset optimization

Note In this chapter, we assume you have some sort of idea regarding coding, not a huge amount of experience. In other words, we provide some intermediate coding concepts assuming you know the basics already.

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_6

Project Structure

Let's start with the first step, which is the structure of the project. More specifically, we mean the folder structure of the project. If you are familiar with other development tools, you may have noticed that other technologies enforce their own project structures. But Unity does not force you to follow any specific pattern. In a way, this is a very handy tool because you can structure the project in whatever manner you wish. At the same time, it does get messy if you don't know what to do. And for a newcomer, it's hard to organize everything properly. Unlike other technologies, you are dealing with multiple types of assets in the project, not just code. For example, any Unity project will more or less have these assets:

- Code
- Image assets
  - Icons
  - Sprite sheets
- 3D assets
  - Character models
  - Props
  - Environment models
- Animation data
- NavMesh data
- Lightning data
- Audio files
- Native plugins

- JAR
- DLL
- Video files

In short, a game project consists of lots of different types of files. You must know which files are being kept where or you will lose track. And for that, you need to maintain your project folder structure. Just so you know, there's no fixed rule for this. You can maintain your own structure. Our recommendation is to take our structure as a suggestion for building your own.

First, you need to know a few things about Unity's file system. The project folder you see within Unity is called the `Assets` folder. There are more folders above it. But you aren't going to touch them. Leave those folders as is because they are crucial for Unity to understand your project. So you are only going to structure the `Assets` folder (Figure 6-1).

Figure 6-1. *Assets folder structure*

The next thing is about adding or removing files from Unity. Always do so from Unity. Never try to add or remove files from the File Explorer. It is certainly possible, but please don't do it. Whenever you add or remove files from the Unity project, Unity creates or removes metafiles for them. These metafiles are not visible from the editor but are visible from the File Explorer (and don't remove them either). Now that you know the things you shouldn't remove, let's move on to adding stuff.

A blank Unity project only contains a default scene. You can delete it or keep it for preparing the demo. It's up to you. You need a `scene` folder to keep all the scenes, which goes inside the `v1.0` folder. We are going to propose two ways to structure your project because they're very common.

The First Way

If you are going to develop a game with different levels that contain lots of different graphical assets that aren't reused in other scenes, this way is best. This is how we organize our `Assets` folder (also shown in Figure 6-2):

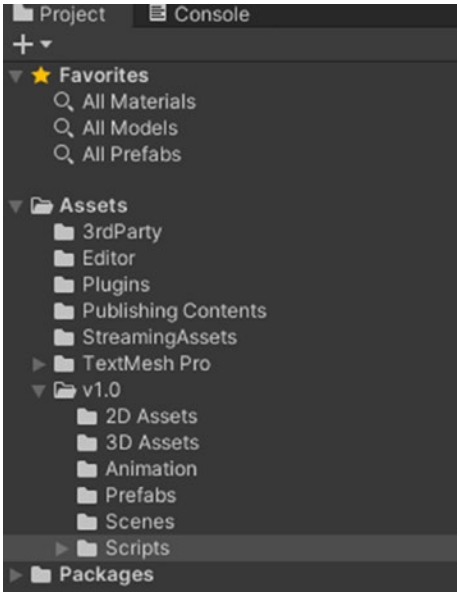

Figure 6-2. *Project structure tree view*

1. `Third-party assets`
2. `Plugins` [this is created by Unity]
3. `Editor` [this is created by Unity]
4. `Streaming assets` [this is created by Unity]
5. `v1.0`
    a. `Scripts` [reusable]
    b. `2D assets`

    c. 3D assets
    d. Animations
    e. Prefabs
    f. Scenes
        i. Maps
            1. Scene1
            2. Scene2
            3. Scene3
        ii. UI
            1. HUD
            2. MainMenu
            3. PauseMenu
6. Publishing content

Let's explain what each folder contains.

Third-Party Assets

Eventually, you will need to use some third-party assets/libraries. They will go here. Inside this folder, each asset will have its own folder like this:

1. Third-party assets
    a. Asset1
    b. Asset2
    c. Asset3

This way whenever you need to change anything from a vendor, you know where to look for it. Here are a few common third-party assets you will be using throughout most of your projects.

- DOTween (tween animations)
- JSON.NET (serialize and deserialize JSON files)
- Odin Inspector (custom editor)

Plugins

What is a plugin? It's more like an extension to the tools you are already using. For your case, it's an extension of the Unity editor. The Unity editor comes with a basic set of tools. Yet there's always more that you need. Most often these plugins are `.dll` or `.jar` files. For example, if you need to show an Android webview with Unity, you use an Android plugin that makes the native Android calls to do that for you.

These files need to be handled specially, so this folder will be auto-generated by Unity if you are using any plugins. Do not rename this folder or the plugins would not work. This folder will automatically organize itself when you import any new plugin. The insides of this folder look the same as the third-party assets folder:

1. `Plugins`
    a. `Plugin1`
    b. `Plugin2`
    c. `Plugin3`

Here are a few common plugins that you might use in the future:

- UniWebView (cross-platform webview)
- Cross-Platform Native Plugins (Android and iOS native calls)
- SteamVR Plugin (virtual reality integration)

Editor

This is a very special folder. Files in this folder will never be included in your build. Sometimes you will install some editor-specific tools mostly for debugging or enhancing the development speed. All that content will go here. You must remember this always: any content in this folder will not be available in builds so do not put any reference content for the game in this folder.

Streaming Assets

This is also a special folder. But this is something that will be included in your build. This folder is used mostly to keep *asset bundles*. We will talk in more detail about this folder in a separate chapter.

v1.0

This is the folder that holds all your project-related assets and for us, this is the root of the project (see Figure 6-3). Why name it `v1.0`? Because later on if you ever want to make a drastic change on the project, you should create a new folder called `v2.0` and make all the drastic changes there. Please note that by drastic changes, we mean a drastic change in assets. For code change, there's a different tool called Git, which we will talk about in detail later.

Figure 6-3. *v1.0 folder structure*

Scripts

Under the `v1.0` folder you will have a `Scripts` folder. In this folder, you will keep all the code that you are going to use in multiple scenes. This folder may need to be broken into more sections like the following (Figure 6-4):

1. `Scripts`
    a. `Character controllers`
    b. `Managers`
    c. `UI`
    d. `AI`
    e. `Audio`

But do remember that this folder will contain the scripts that are reused over multiple scenes. We will talk about scripts that are specifically written for specific scenes later.

Figure 6-4. *Scripts folder structure*

2D Assets

Next comes the `2D assets` folder. Even if you are developing a 3D game you will need this because UIs are 2D and you will need them in your game. All the UI assets will go here. So if you want to break it up even further, you can do this:

1. 2D assets
    a. UI
    b. Game
        i. Character
        ii. Environment
        iii. Props

Now here's a choice you have to make. You can keep the character/environment/props/animations inside this folder or you can make subfolders for them. It depends on how you want to manage everything. Choose whichever style makes sense to you. It will probably depend on what kind of project you are making.

3D Assets

Then comes the 3D assets folder. If you are developing a 2D game, this folder is not required. But for 3D it will most likely look the same as the 2D assets folder but will have some additional subfolders:

1. 3D assets
    a. Character
        i. Model
        ii. Texture
        iii. Material
    b. Environment
        i. Model
        ii. Texture
        iii. Material

c. Props

i. Model

ii. Texture

iii. Material

For animations, you can add one additional subfolder called `animations` for character/environment/props. Alternatively, you could have them separately defined in a different folder. It will depend on your project. For example, if you have only one character animation, then you can put it in there. But if you have multiple characters, you may wish to make one generic animation and set it for each character.

Prefabs

Then there's the `Prefabs` folder. This is one of the most important parts of Unity's workflow. Before we go back to talking about folder structure, let's talk a little about prefabs. What are they? Prefabs are like templates/copies/images of a game object you modified in the inspector. Most often you use the same game object over and over again. So instead of creating it every time or duplicating it manually, you can save a template of it and then use it. And the best thing about prefabs is you can change the template and it will show up in the scene you are working on, which wouldn't be possible if you just duplicated the game object. So if you are using the same object multiple times, always make a prefab out of it. It helps you follow the DRY (Don't Repeat Yourself) principle. And it's not like all the instances of a prefab are identical. You can override if necessary. To learn more about prefabs, go to https://docs.unity3d.com/Manual/Prefabs.html.

Scene

The Scene folder (Figure 6-5) is basically the root of all the categorized scenes. For all your game projects you will have at least two different types of scenes.

- Maps: The game scenes
- UI: The scenes that manage the UI elements

Figure 6-5. *Scenes folder*

Scenes/Maps

The Scenes/Maps folder comes next (Figure 6-6). This is where all your scenes reside. Each scene will have its own folder and each folder will have Scripts, 2D Assets, 3D Assets, and Prefabs folders again, like this:

1. Scenes/Maps
    a. Scene-1
        i. Scripts
        ii. 2D Assets
        iii. 3D Assets
        iv. Prefabs

b. Scene-2

i. Scripts

ii. 2D Assets

iii. 3D Assets

iv. Prefabs

Why? Because now you will keep scene-specific content in these folders. Remember not to include any files inside these folders that are used in multiple scenes.

Figure 6-6. *Maps folder structure*

Scenes/UI

The Scene/UI folder (Figure 6-7) will contain the scenes that manage the menu that guides your game. The common scenes in this folder are the following:

- Hud
- MainMenu
- PauseMenu

Figure 6-7. *UI folder structure*

Now, which assets are usually reused?

It's quite easy. Your UI elements should be consistent throughout the game no matter how many levels there are. So never put UI elements in a scene-specific folder. Your player character or enemy character will most likely be reused in every scene. So don't put them there either. However, if you may have unique enemies for each scene, then you should put them in there. But this is rarely the case. The most common scene-specific character is the enemy boss. Usually, a boss character does not come back in the future. Well, that's not always the case but usually they don't. And each boss usually has its own custom logic scripts that should go to these folders too. Some props will be scene-specific and they can live inside these folders. Some environmental animation and NavMesh data should be in these folders. NavMesh is what you use for pathfinding inside Unity (which we talked about in Chapter 3). As it's an advanced topic, we won't be covering this in the book but you can find out more about it at `https://docs.unity3d.com/Manual/Navigation.html`.

Let's summarize the common things that should be in scene-specific subfolders:

- Props
- Boss model
- Boss scripts
- NavMesh
- Level background music

Please note that this is not an exhaustive list. More items can reside here. But most platformer games will fall into this structure.

Publishing Content

This folder contains the company logo, intro video/animation, company information, legal documents, etc. Usually this content will persist across multiple projects, except for the game icon.

The Second Way

If your game is rather a small one and it reuses the same code over and over again, then this simplistic structure (also shown in Figure 6-8) will suit you more:

1. `Third-party assets`
2. `Plugins` [this will be created by Unity]
3. `Editor [this will be created by Unity]`
4. `Streaming assets` [this will be created by Unity]
5. `Scripts`
    a. `Character controllers`
    b. `Managers`
    c. `UI`
    d. `AI`
    e. `Audio`
6. `2D assets`
    a. `UI`

    b. Game

        i. Character

        ii. Environment

        iii. Props

7. 3D Assets

    a. Character

    b. Environment

    c. Props

8. Animations

9. Prefabs

10. Scenes

11. Publishing content

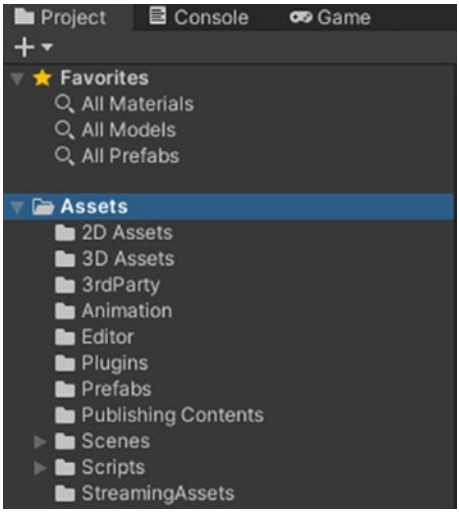

Figure 6-8. *Simplified project structure*

Yes, this is the same structure only more simplified. You don't need to make scene-specific folders; one scene folder will contain all the scenes. All the assets are assumed to be reused. Maybe some assets will be scene-specific but as their numbers are low in this kind of project you can ignore organizing them separately. Most mobile puzzle games fall into this category.

Code Patterns

In Unity, you can code any way you want. You can write a script anywhere and reference it from anywhere. Unlike other code projects, you don't have to worry about assembly or namespaces, although it's possible to have an assembly definition or namespaces to organize your code inside Unity. But just because Unity lets you code however you want, you should not. If you are trying out new styles or features, this can be very powerful as you can test out new things without worrying about these things. But the coding patterns are there for a reason: to keep things organized.

There are a lot of coding patterns that you can work with to keep your code organized. We are going to talk about two architectural patterns:

- MVC
- Single responsibility

MVC

Model-view-controller (Figure 6-9) is a very common design pattern for most software development paradigms. How does this fall into game development? Because in games we don't show data; we have characters moving around or other graphical content moving around or background music being played. Where's the model or the controller? In the case of

game development, it becomes a little fuzzy. But it is still a very good design pattern to follow. Let's try to explain what the model, view, and controller are in game development.

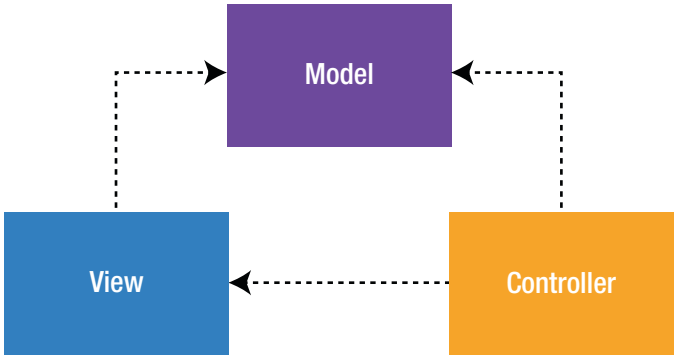

Figure 6-9. *Simplified MVC architecture*

View

Let's start with the view as this is the only thing the players will actually see. Whatever we see is called the view, or at least that's how it's supposed to be in the MVC pattern. But for games, it's a little more to it. Games are not just something we see. They're something we experience, so we can safely say anything that the player can experience is the view. For example, not just the visuals but also audio and haptic feedback sent through the vibration of controllers. There should be no doubt that everything in the game window we can see can be called the view. But we should also address the sound and other haptic sources as a part of the view for game development.

Model

Before we explain what a model is, let's try to fix one misconception. You might think that there is no data in games. However, everything we see on our machines is data, manipulated in one form or another. By data, we don't always mean databases or Excel sheets.

We'll make things easy. Where is the model? What kind of data are in games? In games, the most common data is the coordinate geometry data. Each item on the scene must have these three data points:

- Position(x,y,z)
- Rotation(x,y,z)
- Scale(x,y,z)

Whenever something moves, it's done by manipulating those three data points. No game object can exist without them.

Game objects need more data to explain themselves. For example, if you are designing a 2D character or a model, you will need image data for that game object. For 3D, you need to give it some mesh data. For animation, you need to give it some keyframe data so that it knows which frames change into which frames.

Now you shouldn't have any doubt about models existing in game development too. Models are data containers for you to manipulate the game.

Controller

Controllers take input from the players and then process the model and update the view. They're more like the central management system. The controller does not necessarily have to take input from the keyboard or mouse only. Even button events from the UI fall into this category.

To understand the analogy better, take a look at Figure 6-10.

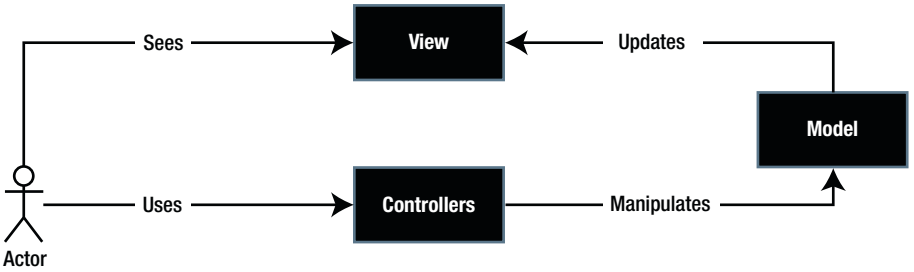

Figure 6-10. *MVC for games*

Now that you know the terms, let's talk about the benefits of using this pattern. Why take all these extra steps to develop game code?

As you develop more and more games, you will realize sometimes you are rewriting the same type of code again again again. But if you follow the rules, you can abstract your code and make it reusable in many other scripts as well as other games.

You should always strive to write decoupled code. If you are the only game developer on your project, you might find it unnecessary to organize the code in this way because you already know what is where. *This* is where you make the biggest mistake. A game is never developed by one person. A game is always developed and maintained by at least two developers. Both developers are you, but one is *you today* and the other will be *you tomorrow*. You today will know what is where, but you tomorrow will very likely forget what is where. And that's when you get in trouble. But keeping things organized will save you in the future. After 5-6 months you will forget what was where because you are likely doing something new at this point. So if you need to support your own project, you're in trouble. And if you need someone else to help with the project, this is where the project pattern will shine the most. Following a design pattern will make it easier to communicate with team members.

Single Responsibility

If your system needs new features, you will have to add or remove some code. And that may very likely force you to make changes to other files too. But what if I told you that you can add or modify features without touching anything or breaking anything else? The goal is to make a decoupled system that works when files are combined but the files don't rely on each other to work. You will need something at the top of the hierarchy that ties everything together.

Single responsibility is another widely used design pattern for game development. It's quite self-explanatory. One class should be responsible for only one thing. It's as simple as that. The benefit of using this pattern can be explained very easily with the diagram shown in Figure 6-11.

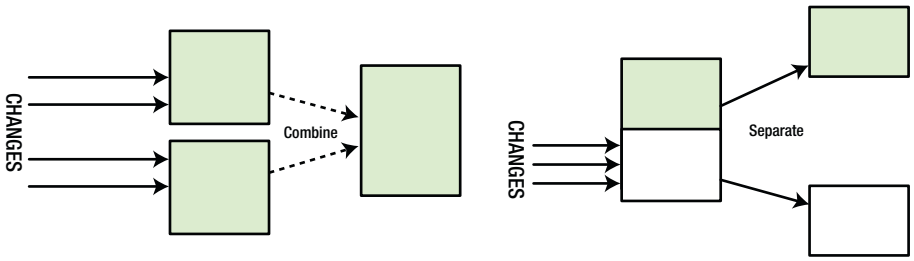

Figure 6-11. *Adding/removing features with single responsibility*

Need to add new features? Just combine them with previous ones. Need to remove a feature? Just pull it out of the system. It's as simple as that. So how do we make such a system? Let's design a small system for that.

You are going to develop a **movement system** for a 2D character and break it down into this pattern. But first, let's explain briefly what is expected of the movement system:

- Should take inputs
- The character should move according to the inputs.

- Proper animations should be played.
- Manage collisions during movement

It's an oversimplified version of a movement system but it will serve as a good example of this pattern.

By definition of this pattern, one class must be responsible for doing one and only one task. It shouldn't have more than one public function. But that's not always the case. In reality, what you should do is create one class that focuses on one particular domain such as the movement system. It shouldn't go beyond doing anything other than the movement-related tasks.

You are going to write four classes to handle these four features and one manager class to handle the features.

Input Handler

This class will only receive inputs. It doesn't know about anything else. Once it gets an input, it will fire an event. It doesn't know who will use the events and how. It will have these public events:

- `OnMove`
- `OnJump`
- `OnCrouch`
- `OnAttack`

Motion Handler

This class is responsible for moving the character. It doesn't know anything about inputs. It only knows about the game object that needs to be moved. It will have these functions:

- `Move`
- `Jump`

- `Crouch`
- `Attack`

Animation Handler

This class will have one private property called the `animator`. An animator is a state machine. It manages these states of animation:

- `Move`
- `Jump`
- `Crouch`
- `Attack`
- `Take Damage`
- `Death`

Each one of these is a function that changes the state of the animator. This class doesn't manage any input. It doesn't even know anything about inputs. All it knows is there are an animator and some states that can be manipulated publicly with those functions.

Collision Handler

The purpose of this class is to track a game object and fire events whenever it collides with something. It has these public events:

- `OnCollisionEnter`
- `OnCollisionExit`
- `OnTriggerEnter`
- `OnTriggerExit`

Putting the Pieces Together

The four classes above don't know anything about each other. And they don't rely on each other either. Why did you go through all these extra steps? Decoupling. You removed dependencies on each feature. That way if you need to make any changes to the animation, you don't have to worry about breaking any other feature. Maybe you want to manage the collision differently now. You can change it without breaking anything else. The same goes for the input handler.

Now you need to tie all of these classes together to make the movement system work. You need a new class. Let's call it, `movement system manager`. In this class, you will have a reference to the four classes above. This class will know how each of them works.

Now when the input handler fires the `OnMove` event, the manager will tell the motion handler and animation handler to execute their move function. The motion handler will manage the actual movement of the game object and the animation handler will animate the movement. And all three classes will work together without even knowing each other.

The collision handler will let the movement system manager know if the game object should be allowed to move forward. Depending on that, the manager can decide how to handle the motion and the animation.

This way you manage to write a decoupled movement system.

If you want to make changes to the animation system, just open the animation handler and make the necessary changes. The system will work without breaking anything.

Drawbacks

This pattern has one serious drawback: it creates lots of classes and lots of files. If you don't follow the proper folder structure, the increasing file size will get overwhelming. But that's exactly why we talked about folder

structure at the beginning of the chapter. If you have been following the steps, the overwhelming number shouldn't be an issue, since everything has already been organized.

Conventions

Now that you know about structuring your code, let's go in-depth about what to write and whatnot. We are going to cover conventions.

You can certainly write code in any way you want to. You can name your variables `ABC, asd, aaaa,` or whatever you wish. But should you? Maybe you need to write a quick function that will only be used within the class. You might feel lazy and name it `QuickFunc`. For the moment it sounds fine. But after a week (when you've totally forgotten about it), you have to debug something in that very class. Now you are in trouble because you need to understand what the `QuickFunc` does by reading it line by line. Worse, someone else has to do it. So what if you named it properly?

So how do you name a function properly?

Here are a few tips from us. Ask yourself these questions:

- Who's going to call it?
- What does it do?
- What's the output?
- What's the input?

Once you have the answers, you can decide the signature of your function.

Who's Going to Call It?

If the function is being used only by its class, you better put a private or protected access modifier. If you want your class to be inherited, make it protected; otherwise, most of the time you can make it private. If you

want to call the function outside of the class, then you need to put a public modifier.

Now you may say, oh it's common sense to do that. But does it matter? Yes, because making everything public saves you the trouble of thinking too much. Actually, this is done to save you time later. Because once you see that the function is named private, you know this function is only used within the class. During debugging, this will greatly help. It has another benefit. Since it's private, you won't get this method as a suggestion when using the class from outside. This way you will have a clear idea of functions that are available to you. This is very helpful when others are using your classes. They don't need to know the small details about your class.

What Does It Do?

This is where the naming happens. It's wise to name a function according to what it does. For example, if you want your function to move your character, you should probably name it `Move()`. If the function moves the character to the left, you should name it `MoveLeft()`. The same goes for the right: `MoveRight()`.

How does it help? Without reading a single line of code, you already know the purpose of the function. This is extremely important for debugging or feature enhancement. One class may have 4-10 functions. What if you need to fix the code for movement? You can instantly find the lines of code by looking at the name of the function. If you named the functions `ABC()` or `ML()`[move left, in short], you would have to spend some time reading the whole class whenever you needed to do any modifications.

As for the naming conventions in C#, it's recommended that you use **CamelCase**. But you can indeed follow your own convention. However, always remember whatever you do, follow one convention throughout the project, and let your team know what it is. This will save a lot of time.

What's the Output?

Every function must give you something back. Functions are like machines you put something into. It gets processed and it gives you something back. Ninety percent of the time your function will have something to give you back. Always try to return something, even if it's just a boolean. If the function doesn't have anything to return, try to return true or false if the action was successful. Think twice before you write a void function. If you make a function void, it kind of says that it's an action. This is a great way to communicate with yourself and the other members of the team.

What's the Input?

This is what decides if your function needs one or more parameters. Think about the things it needs to process. Obviously, you are working with some form of data.

A function should almost always take input. Public functions are usually made to process data from a different place, which may not be available to the class already. Usually public methods are the main entry point for sending data to a class to process.

Private functions are usually used to assist the public functions. Most of the time, private functions are written to make sure the public functions aren't too bloated with code. A private function is usually called from within the class, or more specifically, a public function within the class. As a result, most of the data you need to process in a private function should already be available via some public function, which can now easily be used from private variables. But it's always wise to have parameters in all functions that compute with one or more variables. This way it is always reusable.

The Benefit

Let's combine this together. Here's a function named

```
public bool SaveFile(string filepath, string data)
```

Without reading the body of the function, you can understand what it does.

- From the name, you can tell it saves a file.
- It's a public function, so it's expected to be called from a class object instance from outside.
- The input parameter is a string called `filepath` and a string called `data`. So you can safely assume it saves the data to the file path.
- It returns a bool. So it's very likely that the method will return false if it fails to Save.

You got all this information from the signature of the function. You didn't even have to worry about how the function executes.

Changing even one factor here will send a different kind of information. For example, if you remove both parameters, the function will look like this:

```
public bool SaveFile()
```

But now you don't what it is saving and where. What is the message you get from here?

You aren't supposed to worry about where the file is saved and what is being saved. The class will handle itself.

This way team members can communicate with each other without ever talking.

Asset Optimization

We already explained the kind of assets/files that are required for a game project. We are going to explain how to optimize these files.

Text Files

These files contain the configuration details such as

- Audio/video settings
- Button mapping
- Preferences
- Item details
- Dialogues
- Localization data
- Local leaderboards

You must be thinking that you should have a database for this. But a database for most games would be overkill. And slow too. You need to access this data as fast as possible. You may even want to tweak some of these files outside of the game if necessary, such as the *audio/video settings* or the *button mapping*.

And some files you don't want to be tweaked by users because that could break the game. For example, if someone tweaks the values in the item details, it may make the game unbalanced. For those types of files, you will very much want them to be encrypted.

So how should you save these files?

1. Unstructured
    a. Plain text

2. Structured
    a. Excel
    b. CSV
    c. XML
    d. JSON

Plain text will save you the most space because it will only contain the data you need. But it's going to be hard to maintain it since it's not structured in any way. However, reading speed is very fast.

Excel is very easy to modify, but it adds a lot of extra information that will increase the size of the file unnecessarily. Reading speed is very slow.

XML is also easy to modify as it can be opened with Excel, but it doesn't contain any extra information other than the tags. Reading speed is slow.

CSV (comma-separated values) is probably the best for managing both speed and external readability. It's much better than XML because it doesn't have all those tags. Instead, it organizes data with commas. It can also be opened with Excel and can be easily read by the parsers. The reading speed for this is the second best.

JSON is a structured file system that's like plain text. It isn't as easy as Excel to modify but it is possible. Reading speed is very fast.

What to use and when?

Type	Read/Write Speed	Outside Modification	Formatted	File Size
Plain text	Very fast	Very complicated	No	Smallest
Excel	Very slow	Very easy	Yes	Large
XML	Slow	Easy	Yes	Moderate
CSV	Fast	Easy	Yes	Small
JSON	Very fast	Moderate	Yes	Small

From the chart, you can easily decide which structure to use when.

You shouldn't use plain text almost ever, unless you want to store some temporary data that doesn't need formatting but needs to be read and written very quickly. However, if you are a hardcore programmer and want to parse your own data, sure, go ahead.

You also shouldn't use Excel because it's slow and big in size. It's hardly ever needed for any game. The only feature it provides here that you can click once and it will open with a conveniently installed editor.

If you really want an outsider to modify a file and make their life a bit easier, use XML so that they can open the file with some form of XML reader and modify it. You probably want the configuration files in this format. You won't read them frequently from the game anyway, but users may need to modify them from outside. For example, they might need to change the resolution or keymapping if they messed up very badly.

JSON is the format you should almost always use. It is the most suitable for data storage. It's as fast as plain text. It's formatted. It doesn't waste space like XML tags. It's a well-structured minimalistic data container.

If in doubt, always save in JSON.

Audio Files

This is where most new developers will make a mistake. There are a few mistakes that will happen usually:

- Compression
- Long tail
- Repeating loop

Compression

There are two kinds of compression for audio files:

- Lossy compression
- Lossless compression

Lossy compression removes unnecessary data such as sounds that you can't hear. Once this is done, this information cannot be recovered. For optimal file size, this should be used.

By default, we get the lossless compression, which isn't as good as the lossy one in terms of file size. Also, it's heavy on the CPU because the audio file must be decompressed when it is to be played.

Unless your game focuses a lot on sound, go for the lossy compression.

Long Tail

Some audio files have a long ending tail. Most of the time they are unnecessary for the game. It's always wise to chop out those long ends if not needed.

Repeating Loop

Most of the time you want repeating audio for the background music. Sometimes some audio files contain a 5-10 seconds of long music which is looped for 1-2 min. The actual music is very likely 5-10 sec but the file contains the same data for the rest of the file. You can easily loop the music from the inspector or code if necessary so you can strip out that data. Actually, you should always do this because it will give you more control over the audio loop. To loop your audio clip, just do one simple thing: enable the loop check box as shown in Figure 6-12.

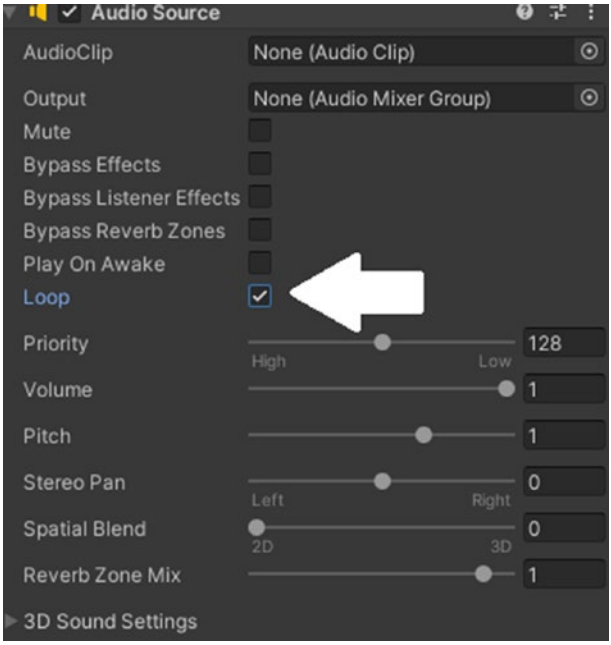

Figure 6-12. *Enabling a loop from the inspector*

Unity Settings

Unity provides a few options for optimization. Let's take a look at the settings available in Unity. Figure 6-13 is the reference.

- **Force to mono:** This makes the audio have only one channel. Unless you want your sound to have different effects on different speakers, check this and the file size will go down.
- **Load in background:** It's pretty self-explanatory. Check this if you have a very big audio file for whatever reason, like dialogue.
- **Ambisonic:** This is a decoder. If your audio files are encoded, check this.

- **Default:** Default settings for all platforms
  - **Load type**: This will decide the type of decompression you wish to have.
  - **Preload audio data:** If enabled, the audio clip will be preloaded when the scene is loaded.
  - **Compression format:** Select the type of compression you want.
  - **Sample Rate:** You can modify the sample rate here.

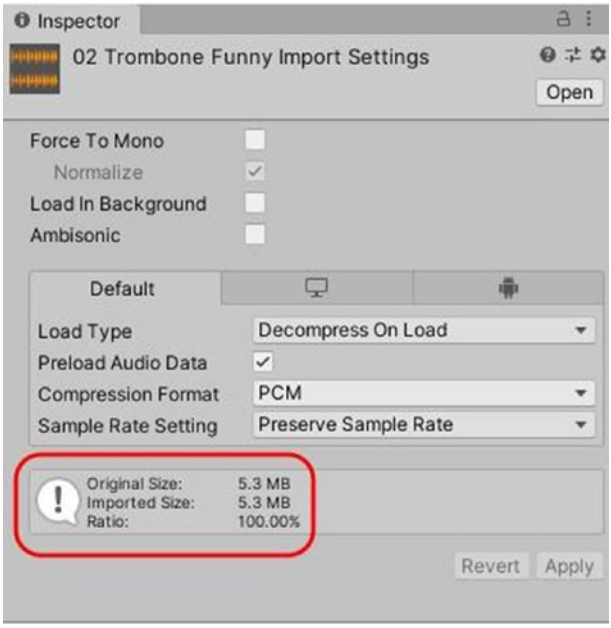

Figure 6-13. *Audio compression settings*

Here you can see by default the size is unchanged. The original size and imported size are the same. You can play around with the configuration to see how the size changes.

For more details, go to https://docs.unity3d.com/Manual/class-AudioClip.html.

Image Files

Imagine a game without image files. Pretty boring, right? Yes, images are the most important file type in a game. They are almost always the main assets and they occupy most of the space. Even if you are making a 3D game, the textures are the ones that will take more space than the actual 3D mesh data.

Common Mistakes

Below you will find a few common mistakes most new game developers will make and some good tips on how to avoid making them.

Resolution

One of the most common mistakes for image files is to use very high-resolution textures. New developers tend to think higher is better and try to get a 4k texture for a mouse cursor or something similar. That's a very big waste of space. You need to know what you need and when. First of all, if you are using anything higher than *1920x1080* pixels, you need to do some quick checks. If it's for background, see if your texture can be tiled. If you are making a 2D game, your characters should almost never be higher than *512x512* pixels. For props, it's probably even lower.

Try to use vectors for your image assets, SVG files. This will save you a lot of headaches. Because then you don't need to worry about higher pixels and file size. No matter how big your screen size, the file size will remain the same. Choose this file format if you are planning to support a very wide array of screen sizes from small to huge. But there's a drawback to this system. Since it's a vector, you may get stuck with a cartoonish art style because it's still not mature enough to give you the details of the other available systems.

Compression

We almost always forget to compress image files. By default, most images are imported inside Unity with a 2k resolution. You may have downloaded icons that are *512x512* pixels but you will most likely need them to be *64x64*. It's wise to compress the image before loading it into Unity. Although Unity can compress it for you, it's better to do it yourself to save some memory.

Unity Settings

Unity offers a few built-in optimizations. You can actually compress your textures from here, as shown in Figure 6-14, instead of doing it with an external tool from outside. We marked the points of interest in the figure that can be tweaked for optimization. This has some benefits. For example, the original image is stored. You only load the amount you need in the game. If you need to dynamically increase sizes, you can decompress from code, which gives you more control over the optimization. But if you don't need to dynamically decompress images, try to avoid doing this because Unity will always keep the original file size in the project.

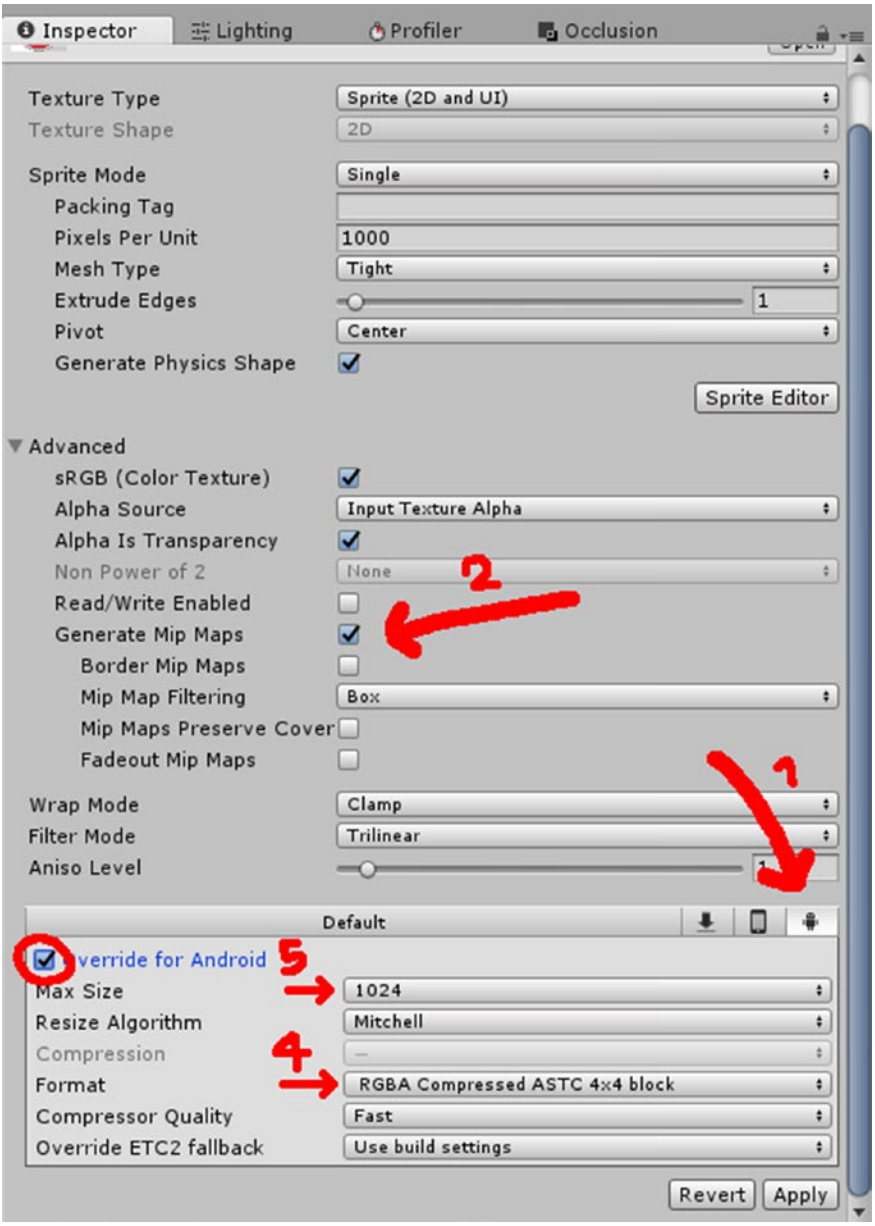

Figure 6-14. *Texture import settings*

Unity offers very powerful tooling for sprite management. You can have one image that contains all the sprites. For a 2D character, this is very useful. You can have all the animations in one file and load them up. To do so, you must select the sprite mode as multiple and open the sprite editor to slice up the sprites.

For more details go to https://docs.unity3d.com/Manual/class-TextureImporter.html.

A Disorganized Army Is As Good As A Dead One

Everything we mentioned above may look like extra work. But in reality, if these steps aren't taken as early as possible, the project will end up getting messy. It's always very hard to organize a mess because it's a mess. It's wise to keep everything as well sorted as you possibly can. Have you ever played RTS (real-time strategy) games like Age of Empires, Command and Conquer, or The Battle for Middle Earth? They take hours to play, especially the ones where you have to build the whole civilization while struggling to fight and survive at the same time. You have to do so many things here. Building a civilization/army from zero needs continuous resources. You have to build up an army simultaneously because otherwise your opponent's scouts might discover your lack of manpower and decide to invade early to get a quick and early win. While you are producing the resources and manpower, you have to manage the defenses as well. In the meantime, you also need to prepare yourself to attack. There are probably barracks to train soldiers and farms to cultivate crops as well. There are some fascinating heroic statues too, providing magical powers or civilization upgrades. Apart from these intriguing structures, there are also some strategic structures. They don't produce anything. They help with some central boosts, for example, 2x resources, increasing kill points. Boring as they may seem, these little things make a lot of difference.

If you check the after-game statistics, you might see that the players using these resources had more points than you. Had they been decent players, you might have been defeated by them. See? Small, boring things can make a difference. Also, in these games, there are options/shortcuts to organize your troops in battle. For example, you are busy defending the northern boundary where heavy artillery is raining down on your game. Suddenly, you hear an infiltration alert. Some enemy troops have used stealth and sneaked into the heart of your base. You knew that this could happen, so you did keep a small but fast troop to handle these sorts of incidents. You check the infiltration location and try to find your agile soldiers to take care of the enemies there. But wait. Where did you keep your agile battalion? The map is filled with little dots depicting your soldiers, especially in your area, but the map is huge and you can't find them now. You forgot where exactly you kept them. There is no way anyone can remember the location of every troop in their huge army. After scanning the entire region for a whole minute, you find them, but by then your important structure has been demolished and you can't make any new upgrades to your troops, eventually buying your opponent a window of a few minutes to try and get to your throat.

This is exactly what you wanted to avoid.

You should have used the shortcuts and tactics to keep your armies at your fingertips. You should have kept them organized so that you could find them in time. After all, what good is an army if you can't use it when you need it? There's not much use for good code/great assets if you can't organize them in your project. You may lose a piece of the puzzle and have to do the whole thing over again, which is equivalent to not only losing the game but also throwing it away in frustration. Let's not go that way. Let's try to do what's necessary when it is due. This extra effort is worth it.

CHAPTER 7

Git Good

In this chapter, we are going to talk about one of the most important tools in the history of software engineering: version control. You will learn about the following:

- What is version control?
- Do we really need it?
- Version control with game development
- What is GitHub?
- What is CI/CD?

What Is Version Control?

Let's start with a scenario. Imagine you are working on a game. After a few days, you install some plugins and change a few lines of code here and there. And now your game stops working altogether. What do you do? Uninstall the plugin and remove all the changes you made. But do you remember all the changes you made? Even if you did, don't you think it's quite a hassle to do all these steps manually? Is there an undo button somewhere?

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_7

So how do you fix this? You need version control. And how do you get it? Well, of course you can do it manually. You could have made a backup of the code before making significant changes and, if you do not like the change, you can just copy the backup back and start working from there. However, creating backups manually is time-consuming and it wastes storage. But it's certainly a solution.

However, there's something better. Something that can keep track of changes. And it doesn't copy everything. It just keeps track of things that have been changed. So if you ever need to revert your changes, you can ask that tool to do so. And that tool is called Git. It will keep track of every character change in your files.

To find out more about Git, go to `https://git-scm.com/`. To install Git, go to the installation page at `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`.

Is that all?

No. Now that Git is keeping track of your file changes, you can use it for a lot of things. For example, what if you are working with two other people on the same project and all of you need to add your code into the same file? How do you collaborate? Since Git can keep track of changes in files, it can be used to make merges from multiple changes. If the changes do not conflict with each other, Git can merge them together without any hassle. If there's a conflict, you can always manually resolve the conflict and let Git know how it should handle that.

Wait! There's more. Since Git can track the changes in files, it can be used to do code review. For example, when you are working together on a project, you may want to do some kind of review before you decide to merge each other's code into the main build. It's possible to see all the changes with the help of Git and this way you can quickly see what's being added and what's being removed in someone else's code. If you do not like the changes, you can ask them to adjust their code and vice-versa.

Do We Really Need It?

There are only two types of people in the world:

- Those who use version control in their project from the start
- Those who will start using version control after they make a mess of their project

In short, version control is not optional. It's only a matter of time before you start using it.

Version Control with Game Development

Now we are going to talk about the technical bits. Version control tools are designed to keep track of file changes. Most changes are in code. But games aren't just made up of code. Games have a lot of other types of files that do not require change tracking because it would be really complicated and mostly unnecessary, at the very least on the file level.

For example, the engine generates metafiles and a huge library folder, which is not necessary for any version control. They are needed by the engine and will be generated automatically by the engine. You won't change them manually, but they are still files. So the version control tool will try to analyze them and give you a list of changes happening in them too. Since you do not want to keep track of these changes, there's something called `.gitignore`. It's a file that instructs Git to ignore files with specific patterns.

Let's see an example of how this is done. In your code repository, create a file called `.gitignore`. Inside the file, you can add patterns that should be ignored. What kind of pattern? It looks something like this:

```
*.csproj
*.unityproj
*.sln
*.tmp
```

You can filter out files by extensions, like this. You can even filter out folders if you wish. For that, you just need to name the folders. They are directed from the root of the project. If you want to ensure the case-sensitivity of your folders, you can define that too. For example, if you are sure you have a Library folder but unsure if it's Library (upper case L) or library (lower case l) you can just write it like this: [Ll]ibrary. Here's an example of folders to be ignored:

```
/[Ll]ibrary/
/[Tt]emp/
/[Oo]bj/
/[Bb]uild/
/[Bb]uilds/
/[Ll]ogs/
/[Uu]ser[Ss]ettings/
```

You can find more details in the documentation at https://git-scm.com/docs/gitignore.

For game development, this is very important because the game engine will very likely generate a lot of unnecessary metafiles that are not needed in the source but for the engine itself. You do not need to keep track of those files either. Now you must be wondering if it's going to be a pain to add those file types to `.gitignore` manually. Fear not; almost every programming tool/game engine has a `.gitignore` template. Here's the repository that contains more or less all of the templates: https://github.com/github/gitignore.

For Unity, you can download the template from https://raw.githubusercontent.com/github/gitignore/master/Unity.gitignore. Also, you can always look for resources/manuals online for the tool/engine.

But now you have a different problem. You may not want to track all the files but you definitely want to add the resources to your version control. You can't just mark them to be ignored. They aren't just meta files

that are generated from the engine. The game cannot be built without resource files. At the same time, these files are too big to be version controlled. Even a few years ago Git was unable to handle such files. But now there's something called LFS (large file storage). Resources such as video files (`.mp4`), audio files (`.mp3`/`.ogg`), or images(`.png`/`.jpg`) should be tracked with Git LFS. With it, Git can now add big files to the source and update them as intended. For this to work, you just need to run a few commands.

Step 1: Install the tool.

```
git lfs install
```

Step 2: Track the files you want to be tracked by LFS.

```
git lfs track "*.psd"
```

Step 3: Ensure the newly made `.gitattribute` file is tracked if not already.

```
git add .gitattributes
```

Step 4: There's no Step 4. You can commit and push your code normally as you would without Git LFS.

To keep this book on track, we will not discuss how the LFS (https://git-lfs.github.com/) works any further. But feel free to look it up if you are curious. Just know that without LFS, version control for games would have been incomplete if not impossible.

What Is GitHub?

Great, so now you have version control on your machine but how do you collaborate? Isn't that one of the primary reasons you are thinking about using Git? What about backup? What if your hard drive crashes? No version control will save you from a corrupted hard disk. Whether today

or tomorrow, you will crash your hard drive; it's hardware, after all, and will break down like all other hardware. And when that happens, you will lose most of your data. Maybe you've been working on a game project for two years and your hard drive is full of backups of the project. But now the hard drive itself is dead. What do you do? You could get an engineer to try to recover your project or you could do what everyone does: access your backup in the cloud. Here's where GitHub (`https://github.com/`) comes into play.

Git is the version of your sources locally on your machine. But ideally, you want your source code to be stored on a server. There are quite a few platforms that provide the service, such as GitLab, BitBucket, GitHub, and many more.

We are going to talk about GitHub only because it's the freest platform as it gets. But other platforms offer more or less the same features. Currently, the owner of GitHub is Microsoft. And it made things better than before.

You can publish your repository on GitHub and share it with other collaborators. You can add collaborators from the repository settings and they can contribute to your code. If you make your code public, people all around the world may contribute to your codebase if it's interesting enough.

A few years ago it wasn't feasible for game developers to use GitHub for source control because there was no way to manage all those large files required in the source. But now GitHub, like all other platforms, supports LFS and you can keep your source code on GitHub. Once it's on GitHub, you will no longer have to worry about losing your code. It will always be there.

What Is CI/CD?

Since you are using Unity, the cross-platform game engine, you very likely to want your game to be built for multiple platforms. More specifically, all the available platforms you can support. And Unity does a great job of doing the heavy lifting for you. That being said, it still takes a while to build

your code into binaries, and each time you switch platforms (for the build) there's a significant time waste for asset reimport. Also, while the game is being built, you literally can't do anything on Unity. Sometimes you can't do anything at all while the games are being built. And that can take hours. Let's try to work around this.

What if you had another PC that you used only for game builds?

But getting another PC for just game builds sounds like overkill, right? And expensive too. For an indie, this could be a lot of money. But don't worry. Because cloud computing is here to save you! You can rent out a cloud computer and do the build in there. Azure, AWS, App Engine, and other cloud services provide these services. They are not free, but they aren't that expensive either.

Don't want to pay for that? Alright, GitHub has you covered. GitHub offers free, headless cloud services where you can run any Docker image and get your workflow running. This is a very advanced feature so we aren't going to explain how it works, but you can look up more about Docker from this link: `www.docker.com/why-docker`.

Instead, we are going to do something better. We are going to tell you what happens and what you get out of it.

In short, when you push your code to GitHub and have a build workflow ready, GitHub will build the code for you and give you the binaries such as APK, IPA, EXE, Deb, DMG, etc. as artifacts. And you won't have to build anything on your machine. What's more, all these tasks will run in parallel in multiple instances on the GitHub server. So you will get a faster build time. Also, if you set it up, you can push your builds to the stores from GitHub. Sounds interesting and fun, right? Google it. Here are some keywords that will help in your search: automate, workflow, CI/CD, cloud build.

Here's an example of a GitHub actions (`https://github.com/features/actions`) workflow. Please note that this is a very advanced feature so we aren't explaining every step. This part is added as a pointer for those who are interested in automation.

For this, you need to write a `.yml` file in the root of your repository where you can tell the server what to do.

Step 1: Check out the repository:

```
# Checkout
      - name: Checkout repository
        uses: actions/checkout@v2
        with:
          lfs: true
```

In this step, you tell the server to download the repository with LFS. As almost always, Unity projects have a large file dependency.

Step 2: Cache the library for the future.

```
# Cache
      - name: Cache
        uses: actions/cache@v2
        with:
          path: Library
          key: Library
```

In this step, you tell the server that it should cache the library folder for future use. Generating the library folder takes some time. This step will save you time on future builds.

Step 3: Build the project in Unity.

```
   # Build
      - name: Build project
        uses: game-ci/unity-builder@v2
        with:
          unityVersion: 2019.4.9f1
          targetPlatform: Android
          buildName: myapp
          androidKeystoreName: myapp.keystore
```

```
        androidKeystoreBase64: ${{ secrets.ANDROID_KEYSTORE_
        BASE64 }}
        androidKeystorePass: ${{ secrets.ANDROID_KEYSTORE_
        PASS }}
        androidKeyaliasName: ${{ secrets.ANDROID_KEYALIAS_
        NAME }}
        androidKeyaliasPass: ${{ secrets.ANDROID_KEYALIAS_
        PASS }}
        androidAppBundle: true
```

In this step, you tell the server to build the Unity project. How is this build done? Now that's a topic for an entirely different book. But for now, there's an open-source project called Unity Builder (https://github.com/game-ci/unity-builder) that does the heavy lifting for you. You let your server know about it and send the parameters it needs to build your apps such as the name of the build, the version of Unity to be used, and the target platform. You can also sign the app with this if you set up GitHub Secrets. GitHub Secrets? It's another thing you will find out in detail when you study GitHub actions.

Step 4: Upload to the Google Play store.

```
- name: Upload artifact to Internal Sharing
        uses: r0adkll/upload-google-play@v1
        id: gplayUpload
        with:
          serviceAccountJson: ./ServiceAccount.json
          packageName: com.myapp.mycompany
          releaseFiles: ./build/Android/myapp.aab
          track: internal
          whatsNewDirectory: ./whatsNew
```

In this step, you tell the server to upload the file to the Google Play store. Here you will use another open-source actions project (`https://github.com/r0adkll/upload-google-play`) to upload the `.abb` file you got from your last job. And you are done.

You can see the output in the Actions console in Figure 7-1.

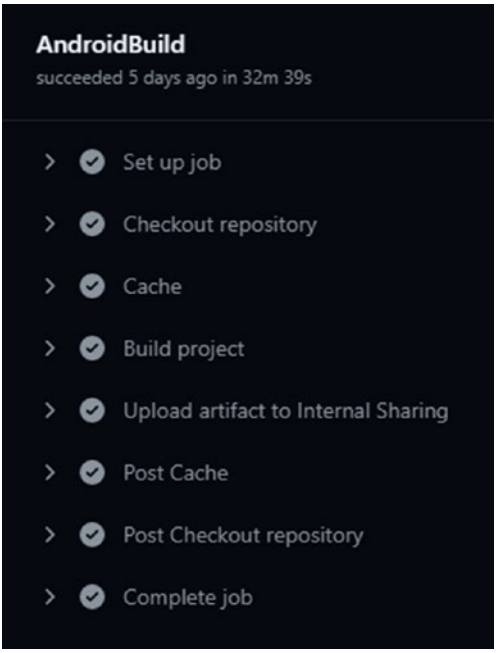

Figure 7-1. *Github actions CI build for Android*

When the server is run, it sets up the jobs. Then it runs them in order. First, it checks out the repository, then it caches it, then it builds the application, and then it uploaded the build to the Google Play store. Before closing, it runs a few post jobs.

As mentioned, this step is actually very advanced so don't get disappointed if you don't get it right on the first try. You'll get there eventually.

Let's summarize the key information in this chapter:

- Use version control from the start of your project.
- Push your code to remote services such as GitHub, BitBucket, or GitLab.
- Automate your build pipeline.
- Automate your store publishing.

Just push your code to the server and let the server do the rest for you.

CHAPTER 8

Get Smart: Good Programming Practices

Now that you know how to set up the project, let's talk about how you can avoid making mistakes in the actual code. We are assuming that you already know how to code, or at least you know the basics. No matter what skill level you are currently at for programming, this chapter will definitely be beneficial to you. Making mistakes in coding is okay because it is strictly proportional to your experience in the field. To jump-start your coding skills, let's talk about some common mistakes in coding.

Here's what we are going to talk about in this chapter:

1. Hard code
2. Loops
3. Booleans
4. Returns
5. Try-catch
6. Arrays vs. lists

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_8

7. Text vs TextMeshPro (Unity only)
8. Unity-specific conventions
9. General best practices

Hard Code

Hard code doesn't mean difficult (!) code. It means coding in static information. What is static information? Information that doesn't change. For example, the value of PI is always 3.1416. So what do we do when we need to use the value of PI? Since it's never going to change, we put that in our code directly. This is a very common mistake; even the pros tend to make it due to laziness. It's one of the biggest malpractices in software engineering. You should never hard-code information. Never!

Let's see a few examples of hard code first and then let's talk about how to fix them. The most common mistake is to hard-code strings. In any game, there will be text that must be shown to the player. Where do you store the text? Do you do it like this?

```
MessageBox.Show("My message");
```

This code has several problems. First problem: What do you do if you want to change the text? You have to open up your code and change the text and recompile the code to see any update. No matter how many times you tell yourself that you won't have to change the string, you will find yourself doing this over and over again.

Let's do a quick fix. Instead of directly passing the string, let's store the string in a variable that can be changed from the inspector:

```
public string message;
public void ShowMessage(){
       MessageBox.Show(message);
}
```

This will achieve the same thing but now you can change the message from Unity's inspector window. So anytime you want to change your string, you can do it from the inspector. The main benefit is you don't have to compile your code each time you do so.

This doesn't end here. Have you heard of localization? Remember the games that support multiple languages, where you can change a language from the settings and suddenly all the UI elements now show a different language? It's called localization. Localization is a process where you localize your game to have localized content. Since different countries have different languages, we need to account for that. This example will deal strictly with text localization.

How do you localize text? One way is to store multiple variables that adhere to the language selected, like this:

```
public string SelectedLanguage="en";
public string messageEN;
public string messageRU;
public string messageJA;

public void ShowMessage(){
      if(SelectedLanguage=="en") MessageBox.Show(messageEN);
      if(SelectedLanguage=="ru")MessageBox.Show(messageRU);
      if(SelectedLanguage=="ja")MessageBox.Show(messageJA);
}
```

This is not a great way to manage localization in the long run because you don't want to type in all of the supported languages each time you want to update. What do you do if you need to support French in this way? Do you add a new variable and manually update the strings from the inspector all over again?

No, you are going to use the old code. The simple code:

```
public string message;

public void ShowMessage(){
       MessageBox.Show(message);
}
```

The greatest problems have the simplest solution. What if you had a way to update the string message before it was passed down to the function? Instead of doing it from the Unity inspector, what if something else did it for you? And by something else, we mean different code.

Now let's talk about how localization is done in reality. What you should do for strings like these is to convert them into resources. By resources, we mean you want them to be available outside of the binary files. They should be loaded as needed. All the strings of a game should have separate dictionary files. It's just a key-value pair. Whenever you need to show a string, you open a file and look up a fixed key. The key you should know about but you don't need to know about the value. You show whatever value is assigned to that key. Figure 8-1 shows an example.

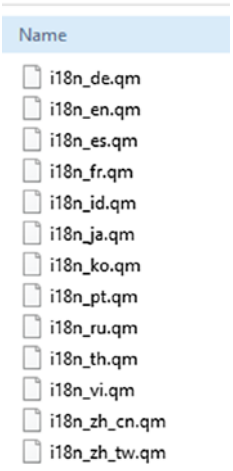

Figure 8-1. *A localized content folder*

`.qm` file extensions are files generally classified as data files that contain source text and translations into another language and are stored in a compact binary format. But for your needs, it's not necessary that you use this specific file format. Also, you don't want your players to mess with the localized files so you probably have some encryption in place and your own file format instead of easy-to-read files like `.txt/.json/.xml/.xls`, which we covered in Chapter 6. Note that it's not exactly necessary to do it as this won't break the gameplay but it's best to do it properly. We aren't going to talk about encryption here; you're encouraged to look it up.

Since these files are encrypted, we can't exactly open them up, but we can show the contents of the file. The English file would have something like Figure 8-2.

	A	B
1	Key	Value
2	Yes	Yes
3	No	No
4	Close	Close
5	WarningMessage	Are You sure?

Figure 8-2. *English (default) localization*

In this case (English), keys and values are more or less the same. But they don't necessarily have to be the same. Keys are there just to look up. The code will search for keys and the values will be shown in-game.

For other languages, this will change. Let's see how they look (Figures 8-3 through 8-5).

	A	B
1	Key	Value
2	Yes	si
3	No	No
4	Close	Cerrar
5	WarningMessage	¿Estás seguro?

Figure 8-3. *Spanish localization*

	A	B
1	Key	Value
2	Yes	はい
3	No	番号
4	Close	閉じる
5	WarningMessage	本気ですか？

Figure 8-4. *Japanese localization*

	A	B
1	Key	Value
2	Yes	Ja
3	No	Nein
4	Close	Schließen
5	WarningMessage	Bist du sicher?

Figure 8-5. *German localization*

Each file should have exactly the same keys, but the values will differ depending on their language.

The next part is easy. In your code, you load up the localized file depending on the language your player has selected. Instead of showing the hardcoded strings, you look up keys in the files you loaded and show whatever value is in there.

This has multiple benefits.

The first one is you can now change your values without touching any game code. So if your tester finds a typo in your game, all you have to do is open up the localized file and update the value there and the game will get updated without any modification to the game code.

The second one is you don't have to do all the translation manually. Someone else can do it. Now that you have decoupled the strings from the game, you can just give an Excel sheet to your translator. Once they provide you with the updated file, you just need to drop it in your project output.

You couldn't possibly add this feature without removing the hardcoded strings from your project.

There are other easy-to-find hard code examples. Whenever you find yourself writing some value in your code, you are doing it wrong. You should never have to write a value in your code. If you absolutely have to do it, declare a variable for it and use that variable everywhere.

Let's give an integer example.

You are making a difficulty setting for the game where the health of the enemies increases depending on the difficulty. Here's a sample code for that:

```
public void healthModifier(int difficulty){
  if(difficulty==1)enemy.health*=1;
  if(difficulty==2)enemy.health*=1.5;
  if(difficulty==3)enemy.health*=2;
}
```

Here's the problem. What if later on you realize that these values need to be changed because the enemies are too hard or too easy?

You have at least five different enemies and maybe they need different types of tweaking. Hard-coding the difficulty multiplier isn't a great idea, is it?

Let's see a quick solution to this problem. You define three different variables so that you can modify them from the inspector:

```
public float easyMultiplier = 1f ;
public float mediumMultiplier = 1.5f ;
public float difficultMultiplier = 2f ;

public void healthModifier(int difficulty){
  if(difficulty==1)enemy.health*=easyMultiplier ;
  if(difficulty==2)enemy.health*=mediumMultiplier ;
  if(difficulty==3)enemy.health*=difficultMultiplier ;
}
```

Now you have more control over how to modify the health of each enemy. Since you have them stored in variables, you can even tweak different enemies differently by storing different values from the inspector.

But wait. This is not ideal code. It can be improved a lot. Let's do it with a dictionary:

```
Dictionary <int, int> difficultyMultiplier = new
Dictionary<int, int>(){
      {1, 1f},
      {2, 1.5f},
      {3, 2f}
};

public void healthModifier(int difficulty){
  enemy.health*=difficultyMultiplier[difficulty];
}
```

You can modify the dictionary values from the inspector (to edit dictionaries from inspector you will need to use something like Odin Inspector) and the health modifier function now gets decoupled. It no longer needs to know how many difficulty settings there are. Maybe you want to add one more difficulty setting. Now you can do it without making changes in code.

Let's talk a little more about the dictionary. It is a `DataStructure` that stores data in key-value pairs. The generic dictionary syntax is Dictionary<TKey, TValue>. Here the TKey can be any type you want it to be and TValue can be any type you wish it to store. But be warned if you pass class in the TKey, the dictionary will behave a bit differently. It's wise to use value types such as int, string, float, or structs. If you really must use classes such as TKey, you will need to override methods `GetHashCode` and `Equals`.

```
//Source : https://stackoverflow.com/questions/46023726/
dictionary-with-class-as-key
class Foo
```

```
{
    public string Name { get; set;}
    public int FooID {get; set;}
    public override int GetHashCode()
    {
           return FooID;
    }
     public override bool Equals(object obj)
    {
             return Equals(obj as Foo);
    }

    public bool Equals(Foo obj)
     {
          return obj != null && obj.FooID == this.FooID;
     }
}
```

By now you should realize the goal is to make sure the code is decoupled of any fixed values. Values change and will change continuously through development. Do your best to make sure you don't have any hard-coded values in your codebase.

Loops

Loops can be a programmer's best friend, but for a game developer, they are one scary tool. It's rather easy to mess them up. One wrong loop and your editor will crash and you will lose any unsaved data.

Whenever you feel like writing a loop in your game, ask yourself multiple times how you can avoid writing it. If you face a situation where

you have no other way but to use a loop, always make sure the loop exits after certain iterations, like this:

```
public List<string> players;

void DoSomethingInLoop(){

 int MaxIteration=20;

 for (int i=0;i<players.count;i++){
   MaxIteration--;
   if(MaxIteration<=0)break;
   //do whatever you want
}
```

In this code, the `MaxIteration` variable has no value to the game, but it ensures that the loop never runs more than 20 times. When the function runs, it sets the `MaxIteration` to 20, and after the execution reaches the loop, after every iteration the count goes down by 1. After 20 times, it goes down to zero and at that point the loop breaks.

It doesn't matter how safe you believe your loop to be. This one of the primary reasons your game will crash/stop working. Players hate it when games crash. Bugs are treated much more generously than crashes.

So the rule of thumb for writing any loop in a game is to make sure the loop always exits no matter what after a fixed set of iterations. It's best if you can avoid loops as much as possible.

Booleans

Booleans are one of the best ways to beautify code, yet we tend to make really bad code with them. It's almost hilarious that we tend to make things complicated when it's actually really, really simple. Let's see an example

code of how new programmers might write one and how it should actually be written. It's a function that turns a game object active and inactive:

```
public void Activate(){
      gameObject.SetActive(true);
}

public void Deactivate(){
      gameObject.SetActive(false);
}
```

The `Activate` function basically makes the game object visible by setting a boolean to true; `Deactivate` does the opposite.

But at the same time, it breaks the DRY principle. DRY means Don't Repeat Yourself. These two functions ideally do the same thing with hardcoded parameters. Instead, what should happen is something like the following code:

```
public void ChangeState(bool state){
  if(state){
     gameObject.SetActive(true);
    }
  else{
     gameObject.SetActive(false);
    }
}
```

Now, this gets the work done, but this is the part we were warning you about. This is the common mistake in boolean code. It is rather funny how we wrote this code. We could just set the state directly in the `SetActive` function without checking if it's true or false because it really doesn't matter. We are telling something to set true if true and set false if false. It is doing unnecessary checks that we do not actually need. However little it

may be, each check does need some execution time. Here's how we should actually write this:

```
public void ChangeState(bool state){
      gameObject.SetActive(state);
}
```

Here we removed the unnecessary checks. As a result, the machine no longer has to perform an unnecessary conditional check. It can just directly assign something. Less conditions mean better performance. Although we don't usually notice the performance improvement for a single-line execution, this will have a significant performance impact on loops. It improves the readability of the code as well.

Let's go a step further and replace this with an expression lambda:

```
public void ChangeState(bool state)=> gameObject.
SetActive(state);
```

What's the benefit of using an expression lambda? An expression lambda isn't like a function; instead it generates expression trees (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees). It implements closures that can allow you to pass local state to the function without adding parameters to the function or creating one-time-use objects. Expressions are useful for methods that don't benefit from the extra work required to create a full method. So it's wise to replace functions like these with expression lambdas whenever possible because it has a performance impact.

So what if you want to do the opposite? What if the state is true and you want the game object to be hidden? You can just add one ! (Not Operator)

```
public void ChangeState(bool state){
      gameObject.SetActive(!state);
}
```

Return

Most people tend to ignore the potential of the `return` keyword. It is there for a reason. And it can be used to significantly improve your code performance as well as structure. For now, we'll focus on the structural part.

You use conditional statements in almost every function. Usually what you do is check if something is not null and you do this or that. Something like the following:

```
public bool DoAwesomeThing(Gameobject myObject){
          if(myObject!=null){
          //Do this
          //Do That
          //also that
          return true;
       }else{
        return false;
       }

}
```

Now, this code is okay but what if you made some improvements? Maybe instead of doing something if it's not null, you tell the code to do something if it finds null. And it will simplify how the code looks:

```
public bool DoAwesomeThing(Gameobject myObject){
          if(myObject==null)return false;
          //Do this
          //Do That
          //also that
          return true;
}
```

You just changed how the conditional statement acts, and the code now looks much cleaner. You no longer need that `else` statement.

You could also use the `return` to break from loops. For example, take a look at this search function:

```
public player SearchName(list<player> players,string keyword){
      foreach(var player in players){
              if(player.name==keyword) return player;
      }
      return null;
}
```

In this loop, you check for player names. Once you find the player you want, you can immediately return that player object. And if you don't find any player by the end of the loop, you can just return null. Note that `foreach` loops are safe to write because they always exit by the end of iterating over their iterator.

Avoiding Try-Catch

Yes, it may sound weird that we are promoting code that will most likely fail. When we get code that keeps failing, usually for some null value, we almost always use exception handling. There are multiple reasons you should always avoid that.

Know this: Prevention is better than a cure. You may be thinking with try-catch that you're preventing the crash. But in reality, what you are doing is allowing a bug to run. You are making an exception for some problems you didn't foresee. It's better to let your code crash while you are developing it than have it crash at the player's hand or have some feature break because you let a bug run loose.

The reason we discourage writing try-catch is that it won't throw exceptions that you can debug. If you don't write any try-catch, your game will stop running and you will know what exactly you did wrong.

There's another reason we don't suggest try-catch. This is a very advanced feature for high-level languages. So what's the problem with that? C# is a high-level language, after all. Most of the time you will be building your game for mobile devices, namely Android and iOS. These devices now require you to build a new arm64 structure. And for that, you need to use the IL2CPP (`https://docs.unity3d.com/2019.4/Documentation/Manual/IL2CPP.html`) compilation. It converts intermediate languages like C# to C++. C++ does support try-catch; however, it doesn't like it very much. Most often the try-catch block is ignored. Once that happens, you will notice a lot of bugs that you didn't face during game development and thus didn't fix. In IL2CPP builds, you can select two modes:

- Fast but no exceptions
- Slow and safe

The "fast but no exceptions" option means that any managed exceptions that are not caught in user script code and escape to the engine code will cause an immediate crash. With the "slow and safe" option, those exceptions will simply be logged by the engine code, meaning that user script code will be left in an invalid state. So it's basically a good idea to avoid try-catch at all costs to be safe.

Also, exception handling is rather expensive, especially in game development. Normal software is usually event-driven, so exceptions only get called once. But in game development, since almost everything is running on a thread, it's highly likely the same exception will keep throwing until it is resolved. Each time your game throws an exception, it takes up some memory space, which needs to be cleared by the garbage collector, which in turn affects devices with low memory.

Arrays vs. Lists

Arrays and lists serve the same purpose, yet it's best to use lists more often than arrays. It is very easy to mess up arrays because in arrays you need to manage how the data is stored and deleted, whereas lists are more self-contained and have battle-tested insertion and deletion. When we deal with lists or arrays, we almost always do some form of insertion and deletion. Let's see why it's a little complicated for arrays first. If you have an array of game objects and want to remove something from the middle, what do you do?

1. You can go to that index and initialize it with default or null:

   ```
   somedata[index]=null;
   ```

   Here you just go to the specific index and remove it. But this is prone to null pointer exceptions. When you write a list or array, you almost always want to iterate over it and expect to get some data out of it. If you do set some indexes value to null, you will get null data at some point, which isn't very idealistic.

2. Left/right shift all items to that index manually:

   ```
   public static T[] RemoveAt<T>(this T[] source, int
   index) {
     T[] dest = new T[source.Length - 1];
     if( index > 0 ) Array.Copy(source, 0, dest, 0, index);
     if( index < source.Length - 1 )
        Array.Copy(source, index + 1, dest, index, source.
        Length - index - 1);
     return dest;
   }
   ```

In this case, you try to improve your previous issue. You don't just set one value to null. You essentially create a whole new array and copy everything before the index to be deleted and after the index to be deleted. This caused a lot of unnecessary overhead.

Arrays are a bad choice when you want to add and remove data continuously. Instead, you can just use lists to handle the code more simply. You can add remove or insert at indexes as you wish:

```
var numbers = new List<int>(){ 10, 20, 30, 40 };
// inserts 11 at 1st index: after 10.
numbers.Insert(1, 11);
 // removes the first 10 from a list
numbers.Remove(10);
//removes the 3rd element (index starts from 0)
numbers.RemoveAt(2);
```

It's as simple as that. Moreover, you can use the `foreach` loop on lists, whereas you need to use a loop for arrays.

Lists are more maintainable over arrays. So a rule of thumb is to use lists by default unless there's a very specific reason not to. Lists are, after all, a wrapper over an array.

Use TextMeshPro over Text

Unity has two text components: Text and TextMeshPro. TextMeshPro (`https://docs.unity3d.com/Manual/com.unity.textmeshpro.html`) is fairly new, but you should always use it in your new projects. Why? See Figure 8-6.

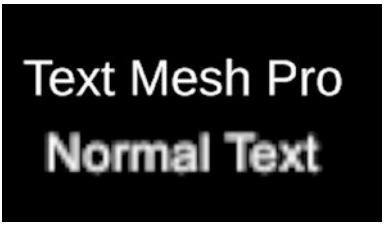

Figure 8-6. *Text vs. TextMeshPro*

In Figure 8-6, we used a font size of 12 and scaled it up to 3x. The scaling effects are easily visible in the normal text.

You want your text to look sharp regardless of the platform or screen. The text component isn't that extensive. You will struggle to get it to look crisp on multiple screen sizes. It doesn't offer nearly half the features of TextMeshPro.

The things you can do with the Text component are pretty basic (Figure 8-7). You can set basic alignments, enable rich text, and set font size. Yeah, that's about it.

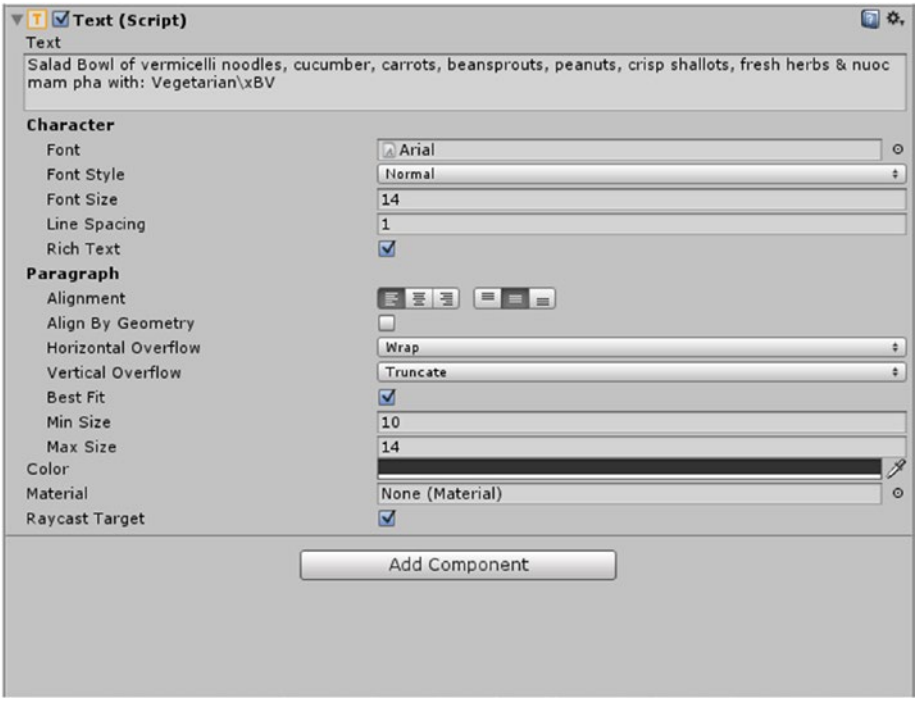

Figure 8-7. *Text component configurations*

TextMeshPro allows you to do a lot more (Figure 8-8). You can enable/disable right-to-left/left-to-right with just a flip of a switch. Unlike Text components, you can use gradient colors. Even better, you can actually add materials to it for more dynamics. Other than just basic bold, italic, underline font style, TextMeshPro offers force uppercase/lowercase features as well. You can even change the spacing between words. Adding extra margins or padding is also possible in TextMeshPro.

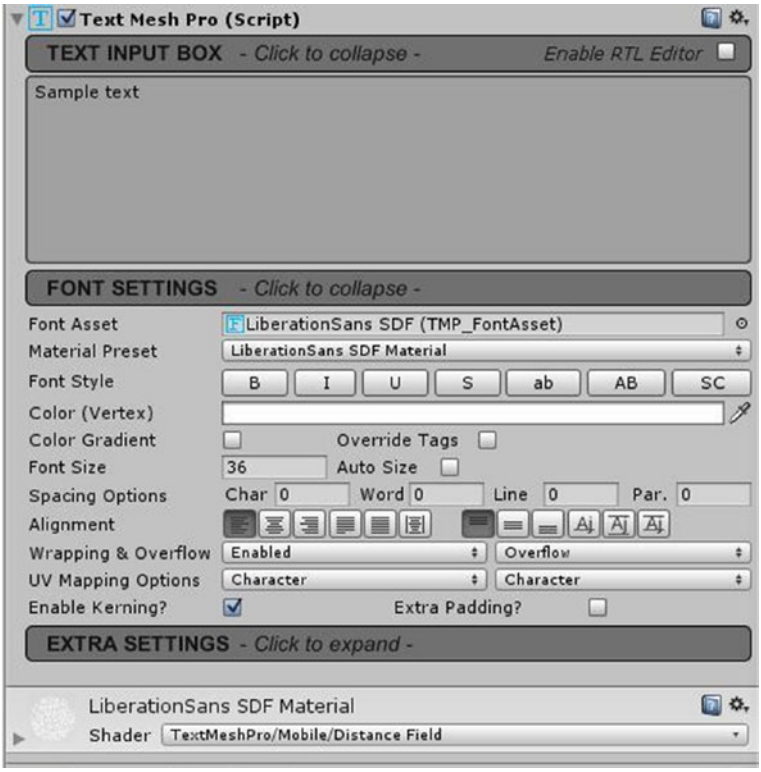

Figure 8-8. *TextMeshPro component configurations*

If you are in doubt about which component to use, 95% of the time you should use TextMeshPro. Use Text over TextMeshPro only if you are making a game that has concerns about build size. That being said, the size difference in a small game for the text component is almost insignificant.

Unity-Specific Conventions

There are some conventions directly regarding Unity. Here are a few examples:

- Private fields start with an underscore and then lowercase.
- Public fields start with no underscore and uppercase.
- Variables start with no underscore and lowercase.
- Private variables cannot be seen from the inspector by default; however, you can add `[SerializeField]` before a private variable to make it serialized and visible from the inspector.
- Public variables can be seen from the inspector by default; however, you can add `[HideInInspector]` before a public variable to make it invisible from the inspector, but other scripts need this variable.
- If any serialized field is initialized from the script, it will start with that as a default value but will get overridden in the editor.

Here are some code examples.

```
[SerializeField] private float _myPrivateEditorField = 1f;
[HideInInspector] public float MyPublicHiddenField = 2f;
private float _myPrivateField = 3f;
public float MyPublicField = 4f;
```

More Good Practices

Other than the above-mentioned topics, there are many other good practices to be followed. They are nice to have but aren't as crucial as the ones mentioned. Here's a list.

1. Always name your variables properly. Example: Instead of `btn` use `Button`.
2. Do not concatenate strings with +. Use the `String.Concat()` instead because it will save memory.
3. Turn off auto light baking while designing big 3D scenes because baking everything while you continue to edit the scene is pointless and expensive.
4. If you're instantiating/destroying a lot of objects, try using an object pool. The principle is simple.
    a. Have a set of game objects instantiated beforehand and set them as inactive.
    b. Instead of instantiating, take an object from the pool and set it as active.
    c. Instead of destroying, just set it as inactive and return it to the pool.

CHAPTER 9

Game Design - The Three Musketeers!

In this chapter, we will talk about good game design and explain the basic formula behind it. We will cover the following in this chapter:

- What makes a game good
- Dissection of successful games
- Game Design: Gameplay
- Game Design: Story
- Game Design: Music
- Game Design: Extras

Dissecting Games

Well, if you have come this far, it's about time we start talking about game design. To understand good game design, we will dissect successful games and think about why they are successful. By the end of this chapter, we will have dissected a good number of games and found a few key features that make a game good. Along the way, we will criticize some wrong notions regarding priorities and design planning. For example, oftentimes we see people focusing so much on graphics these days that they prioritize this

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_9

over all other aspects of the game. Also, people tend to follow success stories rather blindly. As soon as *PUBG* and *Fortnite* made a fortune, people jumped into making a lot of battle royale games, thinking they *should* make battle royale games since they were popular. Similarly, after *Flappy Bird* shot to success within a very short time, people made a lot of clones, only to see them achieving close to zero success compared to the original one.

To understand how to design a good game, we need to look at games that are already proven good and try to reverse engineer the facts of them being good. After doing so, we can decide upon some sort of a formula for a good game.

Let's start with something timeless. Let's go back to the roots. Remember Nintendo's *Super Mario Brothers*? It was released around 35 years ago. And the game is still relevant. Don't believe me? Search on YouTube and you will find at least one player streaming *Super Mario Speed Run*. It doesn't matter what year it is. No matter when you read this book, this game will still be relevant. At the time of writing this chapter, I experienced a battle royal version of the game. Anyway, so what is it that made this game such a hit? Not just a hit, an enduring hit since 1985. It is solid proof that a good game does not necessarily need good graphics. So what are the fundamentals? Let's take a look at a few more examples.

What about *NFS: Most Wanted*? Yes, you will very likely still find live streamers of this game even at this point in time.

Observe something interesting? *Super Mario* and *NFS: MW* are games of different genres and platforms. Yet they are as successful as they can be. So we can clearly say the genre of a game has nothing to do with its success. Nor does a well-established platform. Please stop jumping to a particular genre or platform due to its popularity, before having logical reasons and plans.

Okay, now let me ask you a question. Think of thunder, or water flowing, or dry leaves scratching on the floor. When you think of games in general, does any sound come into your mind? Let me help you out a

bit. Do you remember the background music of *Super Mario World 1-1*? I believe you can already hear it in your head right now if you have played the game. You can probably find it as a ringtone on someone's phone. That was and probably still is "the" sound of games to many people. What about *NFS*? You probably can still remember the menu music because you spent a good amount of time in the menu customizing the cars. But can you remember the background music of *Flappy Bird*? Well, that was a trick question. *Flappy Bird* didn't have background music. What about *Clash of Clans* or *Fortnite*? It's getting harder, isn't it?

One of the core reasons a game leaves a mark in our hearts is the sounds. Not just background music or the sound effects. Can you remember the jump sound in *Mario*? The coin? What about the mushroom? The bullets? If you can remember them without any hassle, you can understand why this game is still relevant even at this point in time.

We know music has its charms. But is it all about music? Do you remember why you even played games? I wonder how many of you knew the story/plot behind *Super Mario*. What was it that kept you running from a castle to another castle? Who even was the princess? And why did she matter to Mario? That we never knew at the time. Yet we played until the game finally told us our adventure was over. Yes, indeed the adventure was over. But we chose to reset the console, and begin the game from start over, and over and over again. What was it that kept us coming back to the same old game? It wasn't even an online multiplayer. Yes, it had an alternating two-player mode but not a traditional two-player game as we experience now. It offered nothing new. At some point, you probably memorized the layouts. If you played long enough, you probably even figured out the secret shortcuts in the game. Yet after all these years we play this very game.

The game was pretty simple: you moved left and right, and you jumped and landed on enemies to kill them. And you shot when you had the flower. What the game was offering at the time was gameplay. It was fun to play with Mario. Just that. All he had to do was jump. Nothing fancy.

Imagine a Mario world without any enemies, just the platforming, the empty space, the breakable blocks. Is it still fun to play? Yes, it definitely is. And will be. The enemies were just juice to the game. Now we have a genre called platformers. And Mario is the father of that. I'd rather not call it the mother as there were certainly other platformer games at the time.

Okay, what else could be the core deciding factors of a game being successful? Have you heard of the game named *Undertale*? If you haven't, I recommend you pause for a moment and take a quick look on Google.

It looks like a game from 1980. Yet this game is one of the best sellers of 2015 over steam. As we repeatedly see games being successful even without astonishing graphics over different decades, we can certainly rule out the graphics being the core factor of a game being popular.

But what about gameplay? The game can easily be related to the Pokemon games from the Gameboy era. Your main character walks around the world and talks to people. And whenever you have an encounter with an enemy, you get a new battle scene where you fight it out. In *Undertale*, the concept is very similar. But there is more versatility in the combat section. You would know if you play it; we are not going to ruin the game for you. But if you are curious, do take a look on the Internet.

Does it look appealing visually at first glance? Most likely not. We are really into graphics these days. Yet without any graphical enhancement, this game was a major hit.

I'll tell you why it was a major hit. It's because of the story. And yes, of course, Sans (in game character). Now I'll get back to Sans later. For now, let's talk about why the story matters. Even without Sans, the game would still be one of the best games because of its story.

The game starts like every other cliché game intro. The main character falls into some cave without any significant backstory. The game shocks you in the intro when the flower tries to kill you in the tutorial. If you are playing the game for the first time, you will be shocked. But what happens if you play the game a second time and you try to avoid getting killed? Well, you will be shocked again. The game breaks the fourth wall. Well, I'm not

going to spoil it any further now. If you are planning to be a game dev, you better play this game or watch someone play it.

Anyway, these surprising parts are part of the story. And this is why it shocks you. Unlike every other game, this game did something uncommon. Maybe not exactly new, but not very common either. The story wouldn't stay inside the game. It dragged you in. How? We will get into details later in this chapter but be warned that this will spoil your experience.

This game has one of the most unique stories in the game industry. Games are one of the best mediums for storytelling and this game masterfully tells the story. No matter how many times you read a book or watch a movie, the experience remains more or less the same. There are books with multiple endings and there is interactive fiction, yes. During the 1980s, choose-your-own-adventure books were popular, and afterward there were waves in the film industry for a while as well. Although its popularity increased most after *Black Mirror: Bandersnatch* (Netflix) was released in this era, it can easily be called a game. We do have similar games, after all, such as *Erica* and *Detroit: Become Human*. In many games, there's a possibility of having multiple plots in one story and multiple endings. And you can decide how you want your story to be told. You are no longer the reader in a choice-based storytelling game. You are the character that's being experienced. Or you are the one who's deciding your character's fate. This gives a sense of control to the players. And we all like to be in control. A game with a good story has more chance of being successful than a game that doesn't have a good story.

Choice-based games rely heavily on their storytelling. Graphics are a plus, but they have a much less significant role for this particular genre. Big-scale AAA games now have multiple endings. The new editions of *Assassin's Creed* have multiple endings. *Farcry* is another example.

Now let's talk details about the core deciding factors of a game, according to us. We are gonna call them the Three Musketeers**!**

- Gameplay
- Story
- Music

We will talk about these three aspects of the game in detail.

Gameplay

So how do we define gameplay? Everything we can do in a game is the gameplay. It can't be a game without any gameplay. That being said, we are not interested in just gameplay. What we want is good gameplay**.** Now, what is good gameplay? That's a harder question to answer.

An easy answer would be, something new. Something fresh!

- For Mario**,** it was the jump to kill. Mario was the first game that introduced this mechanic.
- Remember *Max Payne*? Yes, the slow-mo dive attack was the unique gameplay for this game. Other than that, this game was nothing more than another third-person shooter.
- *Gears of War* made the cover shooting genre popular.
- *GTA 3* made the open-world objective-based game a thing.

Each one of the named games had something new to offer in-game. There were old mechanics but on top there was something new. It wasn't just another clone of a successful game.

For example, when *Candy Crush* was a hit in the Android market, the store was soon flooded with copycat games that offered almost zero new

ideas. After all these years, we remember the name of *Candy Crush* but not its clones. Why? Because *Candy Crush* was a new game. It brought a new gameplay mechanic but the others didn't. The same goes for *Temple Run*.

It's not always necessary to introduce new mechanics to have good gameplay. Sometimes just polishing the existing mechanics will have a great effect, although doing such a thing is quite challenging.

Knowing when to stop is very important in game development. We tend to get obsessed with adding more and more features in the game than we originally intended. Sometimes it's wise to stop adding more sweets on top of the cake. A prime example is the *Assassin's Creed* series. The core mechanics have always been the same: walk up to the enemy and stab them with a hidden knife. The developers knew it was already a fun gameplay mechanic, so instead of changing how this hidden knife worked, they started adding more ways to execute this mechanic.

For example, in *Assassin's Creed 2*, they added two knives instead of one. In the later phases, they even added a miniature gun with the hidden blade, which barely anyone used. It seemed stabbing up close with a hidden blade was a lot more fun than just shooting with a loud noise in a stealth-focused game. That was definitely an overdone feature. But Ubisoft understood it quickly and made interesting changes. You have to understand that the demand was increasing regarding the franchise. After devouring the poetic experience of the first two games, people wanted more.

To me, the *Assassin's Creed* series bounced back in style in *Assassin's Creed 4: Black Flag*. What was the unique addition to that game? The ship battles, of course! All the fans were delighted after finding this new and interesting feature. Let's face it…it was well made. The ship upgrades were worth pursuing while facing legendary enemies and the rewards were attractive as well. You fight a ship, win and collect the loot, upgrade your ship, and then go after bigger dogs. The cycle continues. It doesn't stop in a while because they put the premium vibe in the upper-level ships. Also, the gameplay was fun!

The *Assassin's Creed* series understood when to stop mutating with the hidden blade. This proves that you shouldn't abuse a feature once it becomes popular. Rather, try a different angle, by making it premium feature. The *Assassin's Creed* series evolved a lot in the later years. Now it has become a full-fledged open-world RPG. Even so, the hidden blade is still considered a premium feature and assassination is at the heart of the core game mechanics. There's another lesson there. You shouldn't change the core of your franchise. They tried to be loud in *Assassin's Creed Syndicate*, where you didn't have to use stealth too much as your teammates could aid you heavily in battle. That got a mixed reaction from the fans. The core of the *Assassin's Creed* series is the assassination. No matter how you expand or explore, you shouldn't change the core.

In the case of multiplayer games, you can clearly see that the gameplay is the single thing that matters. You can play with mediocre graphics, at times you can play without sound as well, but the experience of multiplayer gaming solely depends on the gameplay and how the players experience it. A healthy connection matters most while playing, so you should probably optimize everything fully to ensure that the gameplay experience is on point. To this day, people still play old editions of *Age of Empires* and *Command and Conquer* online for hours every day. Their graphics were good at the time when they were released, but even in 2020 when we have much better games (in terms of graphics) and the financial statistics has tipped in the favor of different genres (for example, battle royale games), these old dogs are still kicking on multiplayer. This tells you the whole story. The best example to back this is *Counter-Strike: Source*. No matter what background you come from, you must have someone in your circle who still plays *Counter-Strike* with their buddies online.

Now let's point out a few games that are purely based on gameplay. Have you heard of *Dark Souls*? I'm quite sure you have. Although the game has some lore and stories in play, I doubt any casual gamer actually focused on them. *Dark Souls* is widely famous for its difficult gameplay. The gameplay is so famous that we now have a Souls-like genre. Another

prime example is *Celeste*. *Celeste* does have a story but it's famous for the gameplay it offers. The point to be noted is if you want to create a new genre, you must come up with new game mechanics.

What we want to point out here is that the current trend of reskinning and cloning games is a bad idea to make your game successful. To make a good game, you should probably focus on something that you can truly claim as your own.

Story

The story is the second decisive point of a good game. Right after gameplay, most games are famous for the story they tell. The story doesn't only mean the narrative or lore. It includes characters as well. It's the characters that tell the story of the game.

It's hard to remember games that didn't have a proper story/character in the game. Do you know who the first game character was? It was the 1980's arcade game *Pac-Man*. The game itself is pretty simple. It's basically a collect-a-thon, although just like Mario, it has aged well. Even in 2020 playing *Pac-Man* is still fun. Although there's almost zero story element in *Pac-Man*, it introduced a named character.

Now, what does a character represent? A character represents the player. Once the player can relate to the character they control, they will be more immersed in the game. Even car games nowadays have this approach. Instead of driving an empty car, they let you role-play the driver with actual names and personalities.

Let's talk about Sans from *Undertale*. Sans is an NPC character in *Undertale* and he looks very harmless. Throughout the game, you talk with Sans and become friends with him. During your journey, depending on your actions you will meet a very different version of him. If you don't know about him yet, I recommend you don't look him up directly because it will spoil the game. So before you look him up or read here any further, this is the final warning to turn back and play the game.

SPOILER ALERT!

To explain why Sans is important, we must spoil the game. So if you want the best experience of the game, you should probably close the book and take some time to enjoy what the game actually offers. It will be a good exercise as well.

Reading from this point will spoil the game. This is your last chance to turn back.

Okay then. Let's begin, shall we?

The version of Sans we are talking about is found in the genocide route. In this path, you have to kill every single game character you ever meet. Every single one. You need to take time and find enemies hidden in places until you get notified that you killed everyone.

If you did kill everyone, after this, you are going to have a very bad time. By bad I mean really, really bad. And that's probably what makes this game so memorable.

He'll warn you not to take a step forward. If you do, then this will happen:

Sans: It's a beautiful day outside.

Sans: Birds are singing. Flowers are blooming…

Sans: On days like these, kids like you…

Sans: Should be burning in hell…

Well, what happens after this is a sight to behold. Your world will get turned upside down. Practically. Now we are back to gameplay mechanics so we aren't going to talk about them again in this section. What will happen is you are going to die. You will die for sure no matter how hardcore a gamer you are. But, as mentioned, this is no normal game,

and we are talking about character. What makes Sans so interesting is he explains that he knows about this being a sort of game and all the resets and all. He breaks the fourth wall and explains that in some timeline they were friends and the character was a nice little kid. The player can still choose to reset the game and play the game nicely without killing anyone, at least not unnecessarily.

By this time almost every player will have their mind blown. Because the game is now communicating with the player (the real player) instead of the character that they control. This is something you don't find every day.

Even though *Undertale*'s intended route is to play the game nicely, what gives the game perfection is Sans, the guardian of the genocide route. The game has a lot of aspects that can be praised and a lot of mind-blowing elements. Yet Sans single-handedly makes the game one of the best games ever made. This is the power of a character in a game.

Let's take another look at storytelling. This game is one of the most favorite games for both of the writers. We are going to talk about *Life Is Strange*.

Life Is Strange is a pretty casual game, something everyone can enjoy. It doesn't have any complicated mechanics. Yet it's a very well-received title. This is one of the prime examples of a game being famous just because of its story.

SPOILER ALERT!

Now from this point onward, we will spoil the game. So you have been warned!

Life Is Strange allows you to seize control of a character called Max. She was out of town. A few minutes into the game and you will find yourself with the superpower of rewinding time. And that's all the additional twist this game offers from other story-based games.

This genre has a better name: adventure game. An adventure game structure is the following:

- A complete story that draws you in
- A lot of dialogue
- Puzzles that block your path forward
- Conclusion

The core of the adventure games is its story (*Undertale is also an adventure game*). The second most important part of an adventure game is the dialog. The story is usually formed via the conversations the players have with other characters. Puzzles aren't necessary but it's become a tradition to have some puzzle elements in any adventure game. Come on. What's an adventure without puzzle-solving? Finally, the conclusion: an adventure game must end. Good or bad, it must draw to a conclusion, unless you want your game to hint at a follow-up game.

In Life Is Strange, most of the puzzles are based on the core mechanic of the game, which is time rewind. Even if the time is rewound, you retain memories of what has happened. Now you can make better decisions that will change your outcome. Maybe you'll answer a question that you didn't know before or save someone from getting hit by a baseball bat or manipulate things to drop a bucket of paint on some poor girl.

And that's it. That's all the gameplay it offers. Yet this game has a very long-lasting impression. Why? Because of the story. If you cut out the whole time reverse mechanic of the game, is it still worth playing? Well, at the end of the game you are very likely to find it out the hard way.

The buildup of the story is solid, and you will start getting into the character by doing small but interesting things. Also, they keep giving you this sense of mystery once in a while, which unravels slowly but keeps you going. Eventually, you unlock dark secrets and make choices that make/break lives. All of this is combined with small things you keep doing. This game makes you experience a story that could have really been yours!

Even though it has time rewind in it, it feels so real and you can empathize with the characters almost all the time! If you are past your teenage years, you might even be able to relate to many things that happen in the game. In the end, the game makes you emotional. That's not an easy feat to achieve. Actors in movies can use the power of quality acting, and a good writer can use a plethora of compelling words to make you feel captivated. But a game doesn't have actors and most of the time it doesn't have the luxury of using many words. So if you manage to tell a captivating story through a game, it really makes an everlasting impact on the player. It can change lives. For us, it did.

To end this particular topic, we will summon the zombies. What comes into your mind when you think of zombies? Stealth attacks, huge blasts, shotgun fire, right? There's nothing much to think about other than slaughter and survival.

But wait.

The Last of Us, mainly a zombie apocalypse game, gave us one of the most compelling stories ever told! Although the game was released in 2013, the hype of the first game still persists like crazy in 2021, when we have part 2 in our hands as well. This is just quality storytelling and good game design. A zombie story focusing on two people can be better than most other games you have played in your life? I didn't believe it was possible. Then I played it and my life was never the same again. By the time you read this book, *The Last of US 2* has overtaken *The Witcher 3: Wild Hunt* as the game with most awards. That's not the only news wthough. Well, there's huge backlash regarding the story this time. But it didn't stop the game from being successful, rather made it the center of attention as it bagged the most awards in the Game of The Year ceremony. If you browse through Reddit, you will find a large number of people still rooting (Not all of them are subtle, to be honest) for changing the storyline in this Part 2. This is how much the story and the characters mean to people. All of us who have played the saga are in love with the characters and the story. Not to take

away any credit from the fantastic gameplay this series has, the story has equal say in the phenomenal success of this, if not more.

We can certainly mark the story as one of the fundamentals of a good game.

Sound

Sound is the third fundamental feature of a good game. We don't only mean background sound. By sound, we mean everything audible in a game. The use of proper sound can affect how everyone will feel about the game. Sound has always been an element of mind control. Music makes your body move automatically. There are even war songs. People used to sing those during wars. It's only natural that music is used to control your feelings of a game.

Music can easily draw out emotions from people. If we want to classify music properly, the list might get really complex, so let's not head towards those theories. We can label music in a super simple way like this:

- Happy music
- Sad music
- Curious music
- Horror music

What does it mean? It means music can be used to invoke/manipulate your feelings. If the game is trying to scare you, it will use horror music. If the game is trying to make you feel sad, it will use sad music.

Some games are just based on music, such as *Piano Tiles* and *Beat Saber*. By now you can start to see a pattern. *Piano Tiles* and *Beat Saber* are good games but they do not have any story. They have gameplay and music. It can still be a good game if there's no story, but it's very hard to make a good game without any sound. Try thinking about the simplest

game you ever played. It had sound. *Snake* on Nokia phones? Yeah, it certainly did. Music is used for more than manipulating your emotions. It's one of the means of *feedback*. Feedback is something we will cover later.

We already know that the story is fundamental to a good game. So what if we add that as well?

That brings us to another masterpiece of 2015: *Ori and the Blind Forest*. This game is a prime example of gameplay+story+music-based games. You will notice that the game's story is told by a narrator and his voice will have an impact on your body. Yes, that's what music does. The game will make you sad about music. The game will make you scared when you are powerless with music. And the game will put you on a battle trans when you are trying to escape from a deadly environmental hazard. You feel the game with your ears. Every time you strike your enemies, they make a sound that's also a part of making you feel immersed in the game. We will cover this in more detail in the next chapter.

But, But...What About Graphics?

Now that we have the frame ready, we can add more stuff to finalize our game. Graphics is the skin. It's the final representation of the game. It's certainly important but not as important as the other ones.

When we were writing this book, we couldn't meet the original timeline to publish. Not because there was an imposter *among us*, but because we spent many nights staying awake in a mundane spaceship with very little graphical work, fighting over who was the villain in a 10-player multiplayer game named *Among Us*. *Among Us* was released in 2018 and it didn't get any traction. Two years later, when the world was sitting at home in a pandemic, everyone jumped into a game with an average or below-average graphical merit. The only good thing about its graphics is that the characters are cute. Discord servers are full of channels where friends spend the whole weekend in a game that uses basic guessing and

observation mechanisms. This shows us that simple and basic things done right can achieve wonders!

Fall Guys was released in 2020 and became an instant hit. It uses a battle royale mechanism in a very basic and cute way. The graphical work is not that advanced, but very adorable indeed. The excellence is in the gameplay and music as the sound effects are really fun! You find a lot of players playing with the default character, but you will find it hard to find anyone playing it while keeping the machine mute.

We often see games that are too focused on graphical content over the fundamentals. This is what you need to avoid while thinking about your next good game. EA Sports releases their game *FIFA* every year. Now, just graphical updates won't please the fans of this franchise. They won't pay a premium fee each year just to get a graphical update. So the developers have to upgrade the gameplay every year. But graphics do have a place in the matrix… just not as prime as the three musketeers, but close. For a better understanding, please head towards the next section.

No matter what you do, never forget your core. Don't ruin the basics or overdo things, and you'll be good.

The Fourth Musketeer

Plot twist! In a story about three musketeers, there is after all a fourth. There is always more you can add to the game to add more value. They are addons but they do enrich the game nonetheless. They're like sesame seeds on top of burgers, or maybe an extra patty or cheese. They can make your game stand out from many of the other good games if utilized properly. Among these contenders of the Fourth Musketeer, the first one is the fan favorite: graphics!

We can cite a number of games that are famous for graphical work. We are writing this chapter as *Ghost of Tsushima* has amazed the world with beautiful vast landscapes, breaking records while winning hearts. Not so long ago *Uncharted 4* made us wonder if we were seeing real footage of a hill/spring while pausing the game every few minutes to enter photo mode.

Often a gamer is found on top of a large statue/natural monument, observing miles of the scenery of ancient Greece/Rome/medieval cities, before he goes on to perform the Leap of Faith (the infamous feature of jumping from high altitudes in the *Assassin's Creed* series). All the *Assassin's Creed* games in their respective eras were famous for their visuals.

Spider-Man PS4 did a remarkable job in portraying New York City. Even after finishing the game, I have spent countless hours moving around the city skyscrapers and Central Park. To me, this particular game has the best and most beautiful portrayal of New York City. Graphics can have a fascinating impact on a gamer's mind. I remember playing the last mission of the *Age of Empires 3*. When I played it more than a decade ago, the graphical work was really attractive to me. But sadly, my computer wasn't fully optimized for the game and it was running low on memory at that specific time. It started lagging and I had to play the whole mission in a very weird way as I could see the frames of animations as they were happening. The button delay was horrible as well. But the visuals were just too good for me to ignore. But wait.

Was that the only reason?

Did I play the whole campaign to reach the final mission only to leave it unplayed? It was out of the cards! Curiosity won. The bad experience was overlooked by me at that time, and graphics did play a vital role in it. If you look closely, graphics can hardly be the single deciding factor for persuading a player to play. But it can be a big plus overall. Let's try to validate this concept with two examples.

The first one is the experience of moving around and discovering different parts of New York in *SpiderMan PS4*. Now, surely, the graphics were breathtaking. It felt so real. I am not a New Yorker, so I don't have any emotional ties to the city. Although I appreciate the work from a game designer's point of view and love the experience as a gamer, it would have needed something more to get me hooked properly. Guess what? They did that.

One night I was in the game, discovering the city when from the large window of a Fisk Tower, I recognized something that I know and *care about*. The Avengers Tower was standing tall among all other skyscrapers in the distance.

The interesting thing is, the game didn't force me to find it at that moment. I happened to walk near the window and look in the scene beyond to discover it. This particular experience achieved three things:

- *Emotional attachment*: I am a big Marvel fan, like most other players of this game, so I couldn't control myself from rushing towards it immediately after discovery. They even added some witty dialogues regarding the Avengers as you scale the tall tower. This felt so fulfilling.
- *Surprise*: They surprised me with the discovery. I didn't expect to find the Avengers Tower in the game in the first place.
- *Serendipity*: The way the discovery took place was candid. It didn't feel like anything was forced upon me. I easily could have missed it if I wasn't paying attention. Now that implanted an interesting idea in my mind. What if there's more like this?

So, out of curiosity and hunger for more discoveries like this, I kept on making random runs across the city at times. Sometimes I even did some focused attempts to find other significant places (like the Wakanda Embassy) as well. They have a term for these things. They are called Easter eggs. I loved it more because the graphics were so good, but even if they weren't that good, I would have played anyway for the Easter Eggs and the experience!

Let's discuss a famous scene from the game *The Last of Us*. We are talking about one of the most discussed scenes of the first game, of course: the giraffe scene. In the middle of all the death and zombies, Ellie (the protagonist woman) and Joel (the protagonist man) find a number of giraffes moving around in a demolished city. Actually, they are grazing on a place that had been a stadium when the world was alright. Eventually, Ellie gets to pet one for a while, after gazing over them for a while. It's beautiful.

Now, many can argue that the beauty of the scene is so compelling that it became the iconic and most sought-after scene. But what did the narrative designer of Naughty Dog (the studio that made this game) think when designing this particular buildup? Well, turns out, there was more than one factor working here. Ellie and Joel are in a world that was stripped of its former glory. The buildings are abandoned, and the other structures are either half or fully demolished. The fight for survival is ugly. Before having this encounter, they had to fight their way through humans and zombie hordes. These battles were gruesome and the journey finally begins to take a toll on Ellie just before they meet the giraffes. So, the timing is perfect. It portrays life in the middle of all the death and distraction. When Ellie and Joel spot them, the whole scene keeps rotating very slowly and the game doesn't do anything unless you press a button. The narrative designer disclosed later on that they wanted the player to rest and contemplate for a moment. So, everyone just keeps resting and looking at this calm and peaceful view. As soon as you are ready to go again, all you need to do is press any button. The buildup to this particular scene was solid, the execution was on point, and the visuals were fantastic. All of this combined to became one of the most iconic scenes of the game.

These two examples show us that even a game that's graphically enriched has to rely on some other factors heavily to be in the hall of fame. It's extremely difficult to be timeless with a trait like graphics, which upgrades with time automatically in the industry. By now you know that you shouldn't take a leap of faith until you're certain the game is *Assassin's Creed*. What we mean is when you are distributing your focus, you shouldn't solely emphasize the fourth musketeer. Because fourth can be any feature that enhances the game after the first three. The better approach is to start with gameplay and add one or two of the first three. After that, may the fourth be with you!

CHAPTER 10

Game Feels and Effects

This chapter is about how to make a game feel alive. Can you tell what makes a game feel alive? Okay, let's make it easier. What makes anything feel alive? How do we know what's alive and what's dead? The answer is feedback. Feedback or response can be in any form; it can be auditory or visual or both. But it has to exist and we need to feel it. For example, if we see something moving, we can assume it's alive. If it's making a sound, we can assume it's alive. For games, it is the same. The catch is in philosophy. Do you consider your game as a living thing? Do you want your game to be a living thing? In many cases, we treat our games as if they are just things, and we easily forget to breathe life into them. In this chapter, we will be talking about **game feels**.

Instead of telling you what makes a game feel alive, we are going to remove elements from known games so that you can think about the little things that change the whole experience. They may sound less important, but as soon as you remove them, you realize how important they are for the game to be good.

Let's start with something we all know. Yes, *Super Mario Bros* again. You can imagine Mario jumping. But is that all? I'm sure most of you can actually **hear** the *boing* sound he makes while he jumps. Now imagine if Mario never shipped with that sound. Would the game experience be the same?

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_10

Let's try another one. Whenever Mario hits something that he can interact with it, such as the bricks, they either break or make a coin sound. Imagine if the game never shipped with these sounds. Would the game still feel the same to you?

The answer is most likely no. But they are just small features of a game, right? It's like salt. Salt itself isn't really tasty. But without salt, your dish just tastes plain. Did you know even sweets have some salt in them?

So for games, there's salt too. When it exists, the game feels alive and well. Without it, the game feels lifeless. And the salt in games is known as feedback.

In real life, when we break something, it makes a sound. How would you feel if you dropped a glass and it shattered without making the appropriate noise? You would very likely freak out. For games, we don't really expect them to provide all the feedback properly, but when this does happen, the game feels more alive. There are multiple forms of feedback. The simplest form of feedback is auditory. Let's talk about auditory feedback first.

Sound

Auditory feedback is very easy to add and understand as well. Think about how many games actually implement footstep noise. It is one of the most important immersions a game developer can provide. Yes, it's also pretty easy to overdo the feedback. Not all games need footsteps. For example, a fast-paced shooting game like *Doom* doesn't need footstep noise. It needs a different kind of feedback.

Let's talk about games that do benefit from footstep sounds. Stealth games benefit from footstep sounds. Horror games can make the most out of them; just adding some footsteps can actually make things scarier. For example, if you suddenly hear footsteps other than your own in a horror game, how would you feel? The devs don't even need to add any graphical content to make you scared. That's the beauty of a sound effect.

It's also kind of easy to make a mistake when making footstep sounds. Footsteps shouldn't be repetitive. Also, different surfaces should make different sounds. Walking on wood and walking on grass should have completely different SFX. Otherwise, the illusion of immersion is broken.

The point to be noted is that sounds don't always have to be realistic to make a game feel alive. Here's an interesting fact about movies. Remember Star Wars or any battle that was fought in space? In a vacuum, there's no medium in which to pass sound, so in reality no matter how big an explosion is, you shouldn't hear anything at all. But that would be too boring, wouldn't it? That's why movies cheat and put in sound effects even if it's not realistic. Try to imagine Star Wars without sound and you should realize why it is necessary. Even in cartoons, we hear "swoosh" SFX when the wind passes, but do we really hear such a thing in reality?

Let's talk about *Fall Guys*. The game is full of cute sound effects, which complement the visuals of the game. They didn't stop after making the game adorable; they backed it up with proper sound effects. Any game that you feel you shouldn't play while the sound is muted is certain to have used the feel of sound really well.

Sometimes sound effects become a part of the gameplay. There's a new game on the market called *Valorant*. It's a 5v5 multiplayer tactical shooter game. In this game, if the players run, they make footstep noises. If the opponent has good headphones or a surround system, they can hear which direction the player is coming from. This gives the players an edge in the game. Most multiplayer FPS games have this feature in place. At the same time, if the players walk slowly, they don't make any noise and they can sneak up behind others for the kill. Sound feedback isn't just about making a game feel alive; it's becoming a part of game mechanics.

Controls

Game controls are a part of the game feel as well. Have you seen players lean in real life to see what's around the corner in-game? Or move their body as they try hard to turn their car as if they are actually in it? Game feel is a complete psychological experience. It transcends the realm of virtualizations.

The easiest way to test if your game controls feel right is to follow the development process of *Super Mario 64*. They asked themselves one question during the game development: How does the game feel when we remove the story, props, music, score, obstacles, even enemies? Is the gameplay any fun?

We talked about gameplay in detail previously; now this is what should govern your game development. The first few months of *Super Mario 64*'s development was just about polishing the movement controls. It's a platformer game so the core game mechanic is movement. Once they realized that even without all the additional bulk the game is still fun, they started adding toppings to the game. The game feels for *Super Mario 64* are friction, momentum, and weight of Mario. The level design for *Super Mario 64* is there just to enhance the game feels behind these systems.

Haptics

Haptic feedback is a way to make game interactions physical. Most gamepads come with rumble motors. It's up to the game devs to use them. For example, if you are making a shooter game, every time the player shoots with their gun you can shake their gamepad to make the illusion of gun recoil.

It's pretty easy to overdo this. Some games shake your controller on almost every action. Too much of anything is always bad. And it loses value as well. Strike a balance here.

Haptic feedback makes more sense in virtual reality games. Inside VR your brain can get confused. Most people get motion sickness when they are in VR because their brains can't differentiate reality from virtual reality. At the same time, virtual reality lacks a lot of the rules of real life. For example, when we touch something, we are supposed to feel the touch, however it may feel. When we pick something up, we are supposed to feel the weight. But in VR, we can see that we touched or grabbed something but we do not feel anything. This confuses our brains and makes us feel sick.

This can be optimized by making the VR scenes more realistic or not realistic at all so that the brain knows the difference. Haptic feedback can help you make the games feel more realistic.

Whenever you touch something in VR, you can make the controllers vibrate, allowing your brain to make sense of it. Let's take a VR game for example. *Beat Saber* is a game where you beat the beat with lightsabers. Yes, it sounds weird but it's a fun game. While playing the game, you notice that you can cross your lightsabers together. Now, what happens if you cross two swords in real life? Do they go through each other? No, because of the laws of physics. In VR, you can move your controllers any way you want, but then the lightsabers would go through each other without any form of feedback. This will not make any sense to your brain. So instead what *Beat Saber* does is very smart. Whenever you collide the lightsabers with each other, they make the controllers vibrate in a way that makes you feel like you are pushing two sabers with each other in real life. And your brain sees this too and then it tricks you into doing something really funny. You know that there is nothing, in reality, to hold you from forcing the two lightsabers to go through each other, yet you don't. Or more like you can't. Your brain makes you feel like the lightsabers are real and they are colliding with each other.

At this point, you should already realize how important feedback is for a game.

Visual Effects

This is also known as "juice." Visual effects make a game feel more alive. Visual effects do provide feedback, like the previously mentioned effects, but they do a little more than just provide feedback. They offer almost zero gameplay value but without them, the game can feel dull. Have you ever wondered why mobile games are so addictive? Because One of the prime reasons is that they are filled with juice. Every time you do something, the game rewards you with some form of juicy visual effects that pleases the eyes, which everyone wants to keep coming back to see. Let's talk about a few common juices.

Screen Shake

This works just like the rumble of your controller, only more visually. What do you expect when you hear a tremor? Or when a building is blowing up? It's supposed to shake, right? Screen shake is a very easy way to compensate for the lack of rumble motors in devices that don't have them.

Where should you use screen shake? Any explosion should have some extent of shake. Depending on how close the player is to the explosion, the screen should shake. Also whenever the player falls down on the floor for whatever reason, it's nice to add a small shake there.

It's not necessary to be realistic with screen shakes. You can use them to add more juice to your gameplay. Have you noticed every time you match candies in *Candy Crush*, the screen shakes a little? It's there to add juice to the moment of scoring. There are lots of games that don't do anything special while a score is being updated. A vital part of a good game is rewarding the player. While you do it, you should do it in style. If you don't, your player will lose interest.

How about we show some code? Since this is a very generic feature, most often you can just Google and find code that works as-is. For different game engines, you can find different source codes. For example, we will be

using one open source code to explain how it works. You can find this code at https://gist.github.com/ftvs/5822103:

```
using UnityEngine;
using System.Collections;

public class CameraShake : MonoBehaviour
{
     public Transform camTransform;
     public float ShakeDuration = 5f;
     public float shakeAmount = 0.7f;
     public float decreaseFactor = 1.0f;

      private float shakeDuration = 0f;

     Vector3 originalPos;

     void Awake()
     {
          if (camTransform == null)
          {
               camTransform = GetComponent(typeof(Transform))
                  as Transform;
          }
     }

     void OnEnable()
     {
            shakeDuration=ShakeDuration;
          originalPos = camTransform.localPosition;
     }
```

```
    void Update()
    {
        if (shakeDuration > 0)
        {
            camTransform.localPosition = originalPos +
            Random.insideUnitSphere * shakeAmount;

            shakeDuration -= Time.deltaTime *
            decreaseFactor;
        }
        else
        {
            shakeDuration = 0f;
            camTransform.localPosition = originalPos;
        }
    }
}
```

How do you use it? You just attach this script to the camera game object and hit the Play button. This will start shaking your camera as soon as you hit Play. You can play with the public properties to fine-tune the effect to your needs.

How does it work?

Let's take a look at the variables at hand. First is the `cameraTransform`. You shake the `cameraTransform`. The three float parameters are used to manipulate how the shakes happen. They are made public because you will want to tweak them to fit your game. And finally, there is the original position vector. This is to store the initial value so that you can get your camera back to its original position after shaking it.

The private `shakeDuration` variable temporarily stores the value for using in the `update` method.

The Awake method populates the camera variable and updates the private shakeDuration value with the actual public variable. Every time you enable the gameobject, it sets its current position as its initial original position.

The update method is where all the magic happens. The way this script is written is whenever you enable this game object, the camera will shake. When the shake duration is greater than zero, the function will randomly set the camera's position with the shake multiplier. You can tweak this value publicly to play around with the shake effect.

At the same time, you decrease the value of the private shakeDuration value towards zero. Once it reaches zero it stops shaking. When you want to start shaking again, you just reset the value with the public ShakeDuration variable. This is why you need one extra variable.

How do you use it for your games?

For that, you must change some code. Firstly, you don't just want to start shaking the camera as soon as you hit Play. You can fix this easily. Just add a new boolean to control the shake on and off. Let's add a new boolean:

```
public bool isOn = false;
```

Then you can use this flag to control your update method:

```
  void Update()
  {
         if(!isOn)return;
       if (shakeDuration > 0)
       {
            camTransform.localPosition = originalPos +
            Random.insideUnitSphere * shakeAmount;

            shakeDuration -= Time.deltaTime *
            decreaseFactor;
       }
```

```
        else
        {
            shakeDuration = 0f;
            camTransform.localPosition = originalPos;
        }
    }
```

Now the code won't run unless you toggle the `isOn` boolean.

You can have a public method to set this as enabled or disabled so that you can change it from other scripts as well:

```
public void SetScreenShakeEnabled(bool state)=>{
    shakeDuration=ShakeDuration;
    isOn=state;
 }
```

Now, whenever you call this method, you can enable the screenshake or disable it. Please note that this is actually an oversimplified version of screenshake. What you want is a method that actually takes those public properties as parameters so that you can have more control over it via code.

Although this specific code was designed to work in a specific way, it may not fit your game logic as-is. It's recommended that you study the code to understand how it works and then implement your own. It's never wise to just copy code blindly.

Actually, Unity offers a camera package called Cinemachine that has a built-in screen-shake feature known as Impulse (https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineImpulse.html). But this package is quite heavy if you just need the screen-shake feature. Depending on your needs, you can either code the feature or use Unity's Cinemachine package to achieve this effect.

Particle Effects

Just making a red barrel explode isn't really fun. It needs to have fire and smoke. Shooting a gun isn't any fun unless you can see it making a dent in the surface it collides with. In mobile games, you see the most abuse of these effects. Almost everything you do is bombarded with a lot of particle effects. And yes, they are fun. But many people do only half the job. You have to follow up your action with a properly visible outcome. Particle effects can help you do this, but you need to figure out where to use them smartly.

Physics

This is probably the greatest juice of all. What's more realistic, a tree stopping a speeding tank in *GTA: Vice City* or literally every building breaking down in the *Just Cause* series?

In this age, physics-based juice isn't just an enhancement; it's also expected to some great extent. We always wonder why the trees in *GTA: Vice City* is invulnerable. To be honest, it's kind of a mood killer. Busting down everything on your road except for the trees is an apple of discord to many of the players.

So, how can you implement breakable game object physics? There's an easy way to do so. The easy way isn't very dynamic and thus is kind of static. But it's barely noticeable if you can do it correctly with some randomness. The shortcut way to implement a breakable game object in 2D or 3D for any game engine is the same. You need to actually design the broken object and then assemble it like it's not broken. And then you hide it behind the actual gameobject. Once the player hits the object or whenever you want to show the breaking physics, just disable the actual game object and enable the broken pieces with a physics component(s)

attached to it. For Unity, this is the rigidbody and some collider. And you can even add some force to the broken objects. It will create the illusion of actually breaking it. Here's a community tutorial link that does exactly this:

`www.youtube.com/watch?v=EgNVOPWVaS8&ab_channel=Brackeys`.

This channel (Brackeys)is full of great resources that will help you in your development career if you are a beginner.

Now, what are the drawbacks of this method? Firstly, it should be obvious that you need to manually break the mesh and have them set up in the game. In a small-scale game, this could be feasible. But in a large-scale game, this is a bad idea. You can't possibly do all this manual work for each game object you want to be breakable. Some games actually address this by giving up some physics realism by replacing the breaking effect with VFX and particle effects. For example, when a red barrel explodes, it just creates big flames and explosion effects instead of actual barrel pieces blown around. And the most logical way is the most performance-heavy one: you can split the mesh in runtime and apply physics to that. There's an open-source project that does it for you (examples of which are shown in Figures 10-1 and 10-2). Here's the link: `https://github.com/hugoscurti/mesh-cutter`.

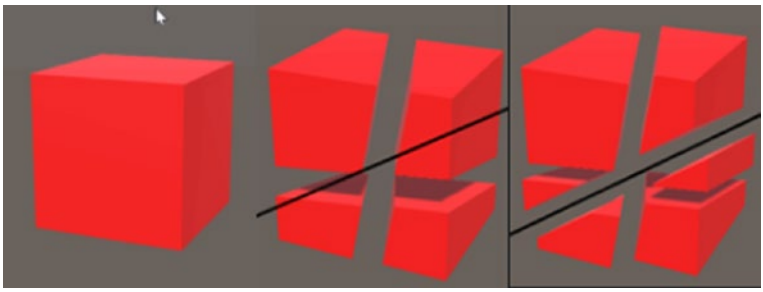

Figure 10-1. *Dynamic mesh (cube) slicing*

Figure 10-2. *Dynamic complex mesh slicing*

But this is a very process-intensive work. So test it out before you decide to chop your meshes at runtime. This directly affects the platform you are targeting. For example, if it's a mobile game, this is very likely a bad idea. For mobile, you might want to stick to the easy way. For PC or console, this is something to be considered after understanding the drawbacks.

Since this is a very advanced topic, we won't be discussing the details on how it works. If you wish to understand how mesh slicing works in-depth, go to `www.youtube.com/watch?v=1UsuZsaUUng&ab_channel=itsKristinSrsly for a tutorial`.

You could actually make games that offer nothing but physics simulation and be successful. A good example is *Goat Simulator*. The only feature this game offers is physics. It's as much of a physics simulation as a game can possibly offer. Nothing else thrills the player more than seeing an actual physics simulation in a game. The most pleasing physics simulation is watching fluid simulations. Games that offer fluid simulation tend to attract more players.

That being said, it's not always a good idea to go wild with physics simulations. It may be the best visual effect but it's costly as well, especially if you are planning on simulating water particles or cloth. They are way too expensive for general-purpose processors to handle. So it's a wise idea to

know where to stop and know how to cheat around the expensive physics simulations with animations. For example, you might notice there are a lot of games where you are placed in an ocean with water. But as we said, water simulation is expensive, so how is it done?

Simple. They cheat. It's not an actual physics simulation. Instead, they just put some randomness to the waves and it feels like the simulation is accurate. If you look closely, you will see the inaccuracies but most players won't focus on them since it's not an important part of their gameplay. Another example is the door animation. You can use physics force to calculate how the door should react to a push. But you can just replace it with a few fixed animations as well. If they are good enough, everyone is happy. Whenever you can replace expensive physics simulations with good enough animations, you should consider doing so.

Things That Can Go Wrong

The mistakes can be divided into two categories in this chapter. One, people forget to inject life into their games. Second, occasionally they overdo it by pushing it too much. By overdo, we mean that you can rarely become successful by making a game based on juice. It's rare but not impossible. Look at *Goat Simulator*. Maybe you still play it after so many years!

The case varies from game to game, so the best way is to judge each one of them properly. You need life and juice in your game in proper places. The reason behind the ignorance is largely due to the fact that we don't give them enough priority. We consider these things as good-to-have things. Well, turns out they are much more than that. Small things can be magic. Small things *are* magic.

CHAPTER 11

Help!

In this chapter, we are going to talk about some specific mechanics and even share some pseudo code on how to implement them. In this chapter, you will learn about the following:

- Difficulty settings
- Rewards vs. punishment
- Game mechanics: Jumping
- Game mechanics: Health
- Game mechanics: Movement

This chapter is dedicated to a few game mechanics that can significantly improve or ruin your game experience. A good game should help the player to play the game. Enemies in-game are but obstacles for players to overcome. Almost all games have multiple paths for players to choose from. The main story is called the critical path. This is something that should be the easiest feat for the players. When you make a game, it needs to target a huge number of audiences. Targeting a niche audience for your game isn't always a great idea. It's not easy to please a niche audience. Then again, you want to empower the niche audience to some extent.

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_11

Player Types

There are a lot of types of players in the community. Here's a list so you can understand the situation better.

Type	Characteristics
The novice	Plays very few games
Story hunter	Cares more about the story than the gameplay
The casuals	Plays a few hours per week
Achievement hunter	Cares more about finding secrets over actual game completion
The hardcore	Plays games daily and has a schedule
The magician	Can play at least one game with their eyes closed
The nerd	Eat ➤ Play ➤ Sleep ➤ Repeat

The lower you get down the chart, the harder it gets to please those players. Your ideal player base is the top four types of players. They are your major audience. But the players who will truly appreciate your games are the latter three types.

Now you have a problem. You can't possibly make a game that will please all of players mentioned in the chart. Their tastes are different. Their expectations of games are different. A niche player base that is your actual appraiser wants a game that offers challenges while the majority want a game that's a fun way to pass time.

This brings us to our first topic: difficulty settings**.** How do you implement good difficulty settings in your game?

Difficulty Settings

When we start a game, we are usually faced with a choice. We need to choose the difficulty: easy, medium, or hard. As a first-time player, nobody knows what each setting will offer. But this is one of the deciding factors of the game experience the player is going to have. Every player wants to have a fun time, but the definition of a fun time isn't the same for every player. Let's take *Resident Evil 4* as an example.

It is the fourth installment of the franchise, and the game already has veteran players (hardcore category). By veterans, we mean the players who played previous installments and are familiar with the gameplay. If the game is too easy for them, they will lose interest. And if the newcomers find it too difficult, they will abandon the game.

Just stamping an easy, medium, or hard difficulty setting at the start of the game is not a good choice because it doesn't explain or adjust to the actual skill level of the player. How does a player know which mode is the best experience for them? A lot of games briefly explain what each mode offers. This has a lot of other problems. For example, this can be a spoiler alert because you are potentially telling how the game will go on, so there's less surprise in the game. Aside from that, this still brings us to the first problem. How does a new player relate to "enemies have low health?" How low is actually low? Nobody can actually know without playing the game first. A fixed difficulty scale is a really bad UX.

There's another problem with fixed difficulty. Let's assume the players selected the easy mode and they're actually a new player. Over time, their skill in the game will get better. At some point, they will no longer face any challenge because the difficulty is not increasing.

Consider it the other way around too. Let's assume the player has selected the hard mode from the start and is having a very difficult time playing the game and is getting frustrated. But over time their skill will increase. And at some point, they will reach the point where the game is no

longer difficult for them anymore. Hopefully, the game will be over by the time the player reaches this point. But what if they reach this point before the game ends?

What you want is a dynamic difficulty setting, a difficulty setting that adjusts itself according to the player's skill. If the player does well, the difficulty rises. If the player does badly, the difficulty lowers. But it never rises too high to frustrate the player and never goes too low to make it boring. It's called the flow state; see Figure 11-1.

Resident Evil 4 is a prime example of this dynamic difficulty implementation. This game had a hidden difficulty setting that adjusts how the AI works depending on how the player is progressing. This is something the developers can find out by analyzing the performance of a player. For example, if the accuracy of the player is too low, maybe make the zombies move slowly so that players can shoot easily. And if the player's accuracy is too high, make them move faster. This way this game is not too frustrating for new players and not too boring for skilled players because the game adjusts its skill cap. This is explained in Figure 11-1.

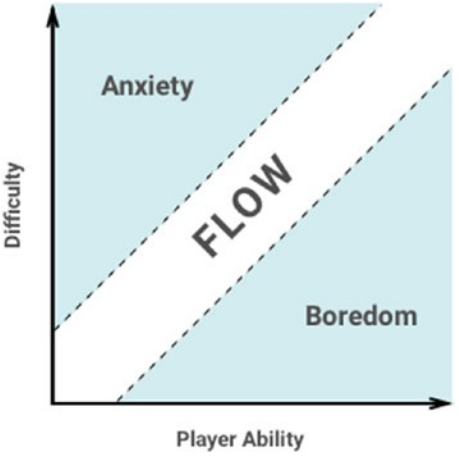

Figure 11-1. *Flow state chart*

On the x-axis is the player ability, which rises over time, and on the y-axis is the difficulty. If you have a constant difficulty setting on the y-axis, you will have a problem. Over time the skill of the player will increase. As the player becomes more skilled and the difficulty stays fixed, the players will enter the boredom state because there's no longer a challenge.

And there's the other state, anxiety. If the difficulty is already too high, the players will enter this state if their skill level does not match.

What you want is to be in the flow state. The flow state is the dynamic difficulty settings because the flow state will change for each player. What this state does is expand the horizon just enough to keep the player out of their comfort zone. And it keeps expanding according to their speed of improvement. Being in the comfort zone is what you want to avoid. This is where the players are struck with boredom. Also, expanding too much brings anxiety.

Ideally, the flow state is what you want to achieve in modern games that tend to target a vast range of audiences.

That being said, some games target a niche audience with this artificial anxiety. Some players even enjoy this; there are games specifically designed to keep gamers in an anxiety state.

The most common example is *Flappy Bird*. It doesn't have much to offer other than frustration. The only reason you probably kept playing the game was to show off on social media how much you can endure the punishment. Yes, gamers really like to flex. This is a form of psychological manipulation. Making the game unusually hard is a way to attract experienced gamers who are very hard to satisfy.

Frustration is a very sharp double-edged sword. You should handle it with care. We can cite Bennett Foddy, who is mostly famous for his frustrating platformer *Getting Over It*. Now, he has made an art out of it. There's even a blog post by him where he describes different flavors of frustration and why game makers should embrace them. If you can play

well with a double-edged sword, it is definitely a very handy tool in a battle. However, it is equally dangerous as well since you might end up hurting yourself. Developers tend to make this mistake a lot.

Another good example of a difficult game that is a measurement of flex in the game community is *Dark Souls*. It's very well known for being very difficult from the start. Being able to finish this game in any difficulty is a crown for a gamer to wear.

Some games do this the wrong way. They develop a broken game and make it unplayable to achieve this state. But that's not how this state is achieved. The game needs to be playable. This is something that we can learn by following what Konami did with *Castlevania*, another genre-defining title. They too were developing a game that challenged the players. But to ensure that the difficulty wasn't something impossible, they had the developers play the game. If the programmer who scripted the boss fight couldn't beat the boss without getting a single hit, they had to fine-tune the script until they could. This way they knew the game was beatable.

Both *Flappy Bird* and *Dark Souls* do this right. The mechanics aren't broken. The game isn't unfair. There are players who have finished the game. So you just need to "git gud." But there's a problem in making a game hard. As shown in Figure 11-1, as time passes, the skill of the player will increase. No matter how high the initial difficult cap was, players will eventually get to that point where their skills are way above the difficulty. And they will fall into the boring state. But as a result, we see new self-imposed difficulty increase in games now.

Yes, players themselves can increase the difficulty of the game manually. For example, *Dark Souls* offers armor for survivability and even health potions. Some players abandon them from the start and try to finish the game so that they can get that thrill of artificial anxiety state the game can no longer deliver normally. There are players who have finished the whole game without being hit a single time even though they used a

wooden stick instead of a sword to beat the game. This is major flexing but we can learn something from it. There's a crowd out there who yearn for games that keep players in an anxiety state rather than the flow.

But this is a niche audience. The recommended difficulty setting design is the dynamic difficulty setting instead of the classic easy/medium/hard choices.

Note that *Resident Evil 4* wasn't the first game to implement dynamic difficulty. In reality, this is an age-old concept. It has existed since the 1980s shoot 'em up games. Also, *Max Payne* had this setup.

There's always the option to mix both of the approaches. The recent versions of EA SPORTS *FIFA* offer the option to choose a difficulty when you begin your first game from the menu option "Kick-Off." This portion is the predetermined approach. After that, based on your performance in the match, the game suggests some changes in the difficulty level. It asks whether you want to play with a higher or lower difficulty before you begin the next match, although this suggestion might not be very effective as most players have too much ego to lower the difficulty after the game says to do so.

So what's the drawback? This game keeps the player in the flow state for the best experience. But we as humans tend to want to be in the safe zone. If the players know that the game adjusts itself when they start playing badly, it can be misused. For example, players can intentionally play badly in some sections to avoid difficult fights so that the game always thinks the player to be on a lower skill cap than they are. Although the players have every right to play the game however they want it to, it may not be the best experience you as a game developer want them to have.

We talked a lot about ideas on how to design difficulty code. Let's now code a simple dynamic difficulty. By no means is this an easy system to develop. But for the sake of simplicity, we will take a game that we already know, and develop a dynamic difficulty setting around it.

We are assuming all of you are familiar with the *House of the Dead* series. It's an arcade game. Arcade games are where the dynamic difficulty shines the most because the business policy behind arcade games to keep getting coins out of you without frustrating you. So if the game is too easy, you can finish the game in one try, which is bad for business. At the same time, if the game is too hard, you won't want to try it since it'll take too much money, which is also bad for business. So a constant flow state is required where the game adjusts itself to its player's performance and makes sure they have to drop a few coins to end the game.

So how do you add dynamic difficulty in *House of the Dead*? Before you dive into coding, let's analyze how to do this. First of all, let's mark the properties that you will need to modify for difficulty change. First, you will see how the conventional difficulty setting is done, and then you will update it to be dynamic.

- The health of enemies (number of bullets needed to be killed)
- Speed of enemies (the faster the enemies will attack you)

For simplicity, let's just keep track of these two properties. First, let's see how to make conventional difficulty settings for these two properties. You can make a small preset table for the enemies that you want to modify for difficulty changes.

Enemy Type	Health	Speed
Boss 1	100	10
Boss 2	150	15
Boss 3	300	18
Boss 4	500	20

Let's assume three modes: easy, normal, and hard. And the chart here shows the default values for the enemy health and speed.

These are the default values (or you can call it the Easy Mode). What do you do to make things harder for the player? If the health value is high, it will be harder to kill the enemies, which in turn, will make things harder. How do you increase enemy health? Or the speed? You already did that previously, in Chapter 8. Let's take a look at the code again but the final version this time:

```
public class Boss1{
 Dictionary <int, int> difficultyMultiplier = new
Dictionary<int, int>(){
     {1, 1f},
     {2, 1.5f},
     {3, 2f}
 };

 public int health=100;
 public int speed=10;
 public void initialize(int difficulty){
  healthModifier(difficulty);
  speedModifier(int difficulty);
 }

 public void healthModifier(int difficulty){
  health*=difficultyMultiplier[difficulty];
 }

 public void speedModifier(int difficulty){
  speed*=difficultyMultiplier[difficulty];
 }
}
```

This code simply updates the health values according to the difficulty that was set before. Depending on the mode the player selected initially, the health and speed will be set for that game.

So how do you make it dynamic?

For that, you need to get some kind of judging criteria first. Remember how every time you died in *House of the Dead* you were given a rank? So there was some kind of judging criteria to rank how well the player was doing. It could be based on several things:

- Shooting accuracy
- Remaining health

There can be more, but for simplicity, let's take these two parameters to judge a player. You can make a table for the player too but that means you are fixing the player to some fixed preset values. You want to avoid this. So instead you make a function to understand floating points instead of whole numbers so that you can make little changes as needed. And you make a new function to place in some kind of rating system to judge the player and get a number out of it.

```
public class PlayerSkill{

 public float accuracy=0;

 public int maxHealth=10;
 public int remainingHealth=10;

 public float GetCurrentSkill(){
    float skill=accuracy+(remainingHealth/maxHealth);
    skill=Mathf.Clamp(skill, 0.5f, 2f);
    return skill;
 }
}
```

In this player skill class, when you call the `GetCurrentSkill` function you get a floating-point value that is clamped between 0.5 and 2. Why? Because you don't want to go any lower than a fixed value. Otherwise, the game will become very boring. For example, you don't want any boss to die with just one bullet no matter what, right? And with 2 set as the upper value, the player can't go anywhere above that because it might make the game unfair. Please note that these values will be different for each game. 0.5 and 2 have no direct relation to this topic; they're used as an example only.

So what does this code do? When you get the `GetCurrentSkill` for the first time, it will return 1 because initially the accuracy is zero and the health calculation also returns 1 as the remaining health and max health are equal. (Please note that you can define your own formula for ranking players.)

As the game progresses, the player will get an accuracy value. It can be anything ranging from 0.1 to 1. As the player gets hit by the enemies, they lose health and start adding less and less points to the skill function. So you have two properties that fluctuate. If the accuracy is good, the skill is good. The more health the player has, the better the skill.

Now you have a skill point. What do you do with it? You need to modify the previous code a little:

```
public class Boss1{

 public float health=100;
 public float speed=10;

 public void initialize(float skill){
  healthModifier(difficulty);
  speedModifier(int difficulty);
 }

 public void healthModifier(float skill){
  health*=skill;
 }
```

```
 public void speedModifier(float skill){
  speed*=skill;
 }
}
```

Firstly, you no longer need a dictionary to hold the preset data. Secondly, you change the data to floating points, which we explained previously. Thirdly, instead of the fixed difficulty code, you pass player skill.

What happens now is you get the current skill of the player at the point before you initialize the boss and change its health and speed values. If the player has good skill values now, the boss will have more health and more speed. If the player has bad skills, the boss will have less health and less speed.

This is an oversimplified version of a dynamic difficulty code. You don't necessarily have to follow these parameters by the book. Even the equation to calculate player skill can be different. There can be more parameters. It will depend on the type of game you are developing. You can use the skill parameter for other components as well. Depending on the skill level, you can give the player some kind of reward. We will talk about rewarding a player in the next section.

Rewards vs. Punishment

There are two ways to guide the player on how the game should be played. The most common one is punishing the player for not playing the way you think they should play. Let's take *XCOM2* as an example of this.

XCOM is a turn-based tactical game. The developers intended the players to take risks in the game and get the best out of it. Instead, the players almost always take the safest route and grind to optimize the difficulty. It's in human nature to play safe. To counter this in their next game, *XCOM2*, the developers put a new system which they believed to be something to guide the players to get the best experience.

What Did They Do?

They put in a timer system. In *XCOM2*, they added a timer system that forces the players to end the mission in a fixed set of turns, otherwise it's game over. The developers had the best interest of the players but this new timer system met with controversy. The players hated to be forced to play in a way they didn't want to. The players didn't stop whining; they even made a mod of the game that removed the timer settings.

What Went Wrong?

Nobody likes being bossed around. The players won't want to be forced to play in a way that you as a game developer may think is the best way to play. As a game developer, you must respect the players' will to play the game as they want as well.

So How Do We Fix It?

The solution to this problem isn't new. To be honest, it's very old. Let's look at some old games. *World of Warcraft* is an age-old MMO. MMO players tend to get addicted to playing a game. They lose a sense of time and sometimes they get sick. You as a game developer don't want your players to be sick because of your game. One, you'll get a bad reputation and two, they can't play your game if they are sick. So the developers thought about this problem and punished the players for playing too long. The reason players play MMO for too long is the games are based on grinding. They have to grind for XP to make progress. The more they grind, the more they get powerful in-game. So the developers added a penalty on XP when the players played the game for too long. The longer they played, the higher the penalty. The community didn't like this at all. It too faced controversy because it was clearly a punishment mechanic even thought it was good for the players.

So the developers came up with something smart. Instead of punishing players for playing too long, what about rewarding players who play less? So they tweaked the XP penalty mechanics into XP boost mechanics. Whenever the players took a break, their rest meter went up. The longer they rested, the more the boost was. So once they came back to play, they gained more XP while the players who didn't rest had the same XP they did before. The numbers were the same yet the goal was achieved without forcing anyone.

Another brilliant example is by EA Sports. Its *FIFA* has been the market leader among all football video games for as long as we can remember. We are going to observe a reward related incident here and talk about our assumption regarding what might have happened there.

In the past few years, arguably the most popular game mode has been the *Ultimate Team*. Here you make your own team and play in competitive modes regularly. There are weekly competitions where you fight to secure a better position to get lucrative rewards. You keep grinding to make an overpowered team dominate the field. As long as you keep playing, they reward you with coins, reward packs filled with players, and other items. Every year, a new *FIFA* gets released around October. People start building their squad immediately. Some people even pay extra to get access a week before the official release of the game.

Since *FUT (FIFA Ultimate Team)* keeps rewarding the player as they keep playing, it's pretty much expected that somebody who starts in October will have a much higher rated squad after playing six months, right? Well, technically, yes. So, what happens to the player who starts their journey in *FUT* in March of the next year? They will start with a basic team whereas all their opponents will be in much better shape, right? The imbalance should be significant. But before you entertain that idea, let me tell you what happened to me.

I started playing *FUT 20* in March 2020, whereas most of the players started six months prior. The division rivals mode (which you can play all week) was okay since it's based on skill level. So, all the new players are put

into Division 9, which is just one step ahead of the lowest division (Division 10). This is suitable to start the journey, but the issue begins when you start playing the Weekend League. The FUT Champs or Weekend League (WL) is like the weekly World Cup to the *FUT* gamers. Not all players can qualify here. You only get 30 matches to play within 3 days. This tournament gives away the most lucrative awards of the game. The matches are really intense and full of twists. The matchmaking happens mainly on how many matches you have won so far on that particular weekend. So, it's very likely to match with a real pro opponent with a powerful squad. I faced this fate with my basic squad in my few WLs. It was painful.

Now, let's consider *FIFA* packs. You earn them by completing objectives, doing well in tournaments, or purchasing them directly. They contain players and items for your club. Pack luck is one of the most discussed things in *FUT*. In any random *FIFA* group, you will always see people complaining about how bad their luck is, whereas very few people come flexing about how they got a super card out of a very ordinary pack. You can get a very low-rated player out of a costly pack and sometimes you can get the best player in the game from a very ordinary pack. I thought that pack luck was a myth because mostly I saw people getting very low-rated players from packs.

So imagine my surprise when I got an actual high-rated player in a pack one day! A few days later, I ended up packing the most recent card of Lionel Messi (99 rated card). Now that was a shock beyond imagination because not only did he fit in my team perfectly (I just had to replace his basic card, which became cheap at that time) but also he is one of the most sought out players in the game. Seeing me packing him, people offered to buy the FUT account from me!

Surprisingly, that was not the end. I kept getting great players from packs once in a while, and eventually my team became one of the most highly rated teams among most of the players. Meanwhile, my friends, who had been playing the game since last October, became furious about

my pack luck because even after playing much more than me, their team was much below standard than mine. It felt like just dumb luck at that time. But when I look back, I remembered that before packing Messi, I was just about to give up playing FUT because of the constant bullying I used to face in the Weekend League with my basic team. I played *FIFA 20* wholeheartedly until the release of *FIFA 21*.

Maybe I didn't deserve that overpowered team but it was necessary to keep me going.

Looking back, it all feels connected. I had help, maybe.
Disclaimer: We don't claim to have known this for certain. Just entertain this as an outcome of logical deduction, nothing official.

Let's Fix XCOM2

Now let's do a little brainstorming. Close the book and think about how they could have done the same thing in *XCOM2* without making controversy?

So here's our suggestion on how they could have done it. Instead of adding a timer that ends the game, they could have made it into a challenge that rewards the players greatly. For example, if the players can finish the game on X turns, they will gain a very powerful weapon or character.

Mobile games propose a good solution to this problem. Most mobile games have this concept of stars. Each stage is rated with stars. If you can finish the stage in the most skillful way, you get the highest stars. And in some games, stars are totally optional. You can still finish the game even if you don't get the stars.

This fixes a major problem from our last topic as well. How do we balance the best difficulty experience? Instead of blocking players from moving forward unless they match a certain skill threshold, maybe reward them if they play well enough. Give them a badge or something to show off that they played well. And let the casuals still enjoy the game.

A prime example of this are the games made by CAPCOM. Everyone knows about *Devil May Cry*. This series emphasizes stylish gameplay with

combos. However, mastering the combos is no easy feat. You can finish the game without even realizing the game has such good combos to offer if you can master them. This opened up the game to a larger player base. Newcomers can just play the game in easy mode and not worry about the combos at all. But the veterans can use their skills to enjoy the game to its full potential and feel rewarded by the game with its ranked scoring system. The ranking system itself has zero impact on the game's critical path, so it doesn't block the players from progressing. But often the game hides some special reward behind the skill cap. Some challenges are made for skilled players to feel rewarded. These are hidden rewards that players have to find themselves. They might even miss them entirely. As a result, neither the veterans feel left out nor do the freshers feel overwhelmed.

This isn't the only example of punishment. Remember those stealth games where the game just forces you to replay if you fail to play stealthily? It's a bad game design. It means you as the developer just didn't account for other playstyles. If you want your players to play a game in stealth mode, reward them for doing so. But do allow them to just go Rambo if they want. *Hitman*, for instance, does this very well. You can always just kill everyone in the mission, but to rank on the leaderboard you will have to score well. And to score well, you must play the game in a very stealthy way.

There are some good examples of punishment as well. Punishing players for not playing the way you want the players to play can also be a great experience. It's something that shouldn't be overdone. We talked about *Undertale* before and the fight with Sans. This can be interpreted as a punishment. This battle only happens if you take your sweet time to kill every single character you ever encounter. You must even find NPCs that are not on the critical path. A game that promotes compassion as its core message would like to encourage you not to do so. The fight with Sans is clearly an off-the-charts difficult boss fight. It's not intended to be won. Not that it's impossible; it's just a big wall that discourages you to be a bad guy. Sans keeps telling you to go back. Go back and be a nice kid. Some

players may get frustrated and go back and be nice to finish the game. That's very likely the intention of Toby Fox, creator of *Undertale*.

The classic example of punishment can be found in a game franchise that has probably been the most popular one for quite some time. We are talking about *Grand Theft Auto (GTA)*! You can be a gangster with the capability to do almost anything in the game. This has been the synonym of freedom for an entire generation, if not more. The story modes have always been fun, but the most enjoyable part has always been roaming around the city and, yes, destroying things.

Now the game does allow you to vandalize/kill at will, but it comes with a price. The police will look for you and the wanted stars will only go up as you keep at these crimes. Higher wanted stars mean more superior law enforcement will come for you. It's your choice. You can be a bad boy in the game, but you will be punished. Most of the pro players enjoy this a lot. Oftentimes, we see them trying to get the highest wanted stars just to have some sense of challenge. In my case, in *GTA 4 Vice City*, it is all about getting to the three stars wanted level so that they send in helicopters and I can shoot them down with a rocket launcher. But I had to run away once they sent the SWAT team to get me. In six stars, they used to send in tanks, making things impossible to handle. These memories are very dear to all of us, right? Look how they have implemented punishment in this mechanism and made the best possible outcome out of it. Years and decades later, we still love the game.

Game Mechanics: Jumping

A game designer's most difficult task is to protect gamers from themselves. Yes, you read that right. Gamers are their own worst enemy in a game. Sometimes gamers have unexpected expectations from games. One common example is in platformer games where the core of the game is jumping around. Most players want to jump off the edge of the platform. They run

until they reach the end of the platform and jump just after they step off the platform. Now, in normal game logic the game character is mostly not even grounded anymore. So logically, it can't jump. But the players won't like that. They'd say the game has bad controls. Just this one jump mechanic could affect how a player feels about this game. This brings us to our next topic.

You must be wondering, what could go wrong in implementing such a simple thing? Well, we hate to break it to you but even a simple mechanic such as a jump can go wrong. Actually, most new developers do get it wrong, thinking it is just a simple mechanic. It is not.

The problem we are talking about is the edge jumping problem. There's a solution to it. And the solution comes from a cartoon. Remember Will E. Coyote from *Roadrunner - A cartoon show by Warner Bros*? It seemed like the laws of physics didn't work for him properly. He could hang off the edge of the cliffs until he looked down. This is why we are going to use the term called **coyote timer**.

This timer is simply a grace period where the game ignores the fact that the player just lost their footing. Usually, this grace period is like 0.1-0.3 seconds. If the player tries to jump during this grace period, the player can still jump even if they have no footing and are supposed to be falling.

Here's a little pseudo-code so you can get a better idea of how to implement this. The most common way to implement a jump is to check if the player is on the ground and the jump button is pressed. If yes, then do the jump; otherwise, ignore.

Code: Pseudo Code for Jump

```
if (key_jump_pressed && onground)
{
    //jump
}
```

But this way the player will not be able to jump off the edge of the platform since they are no longer grounded. So instead of doing this, you do some additional calculations. You need to start a countdown as soon as the

player is no longer grounded. Add this variable called `jump_grace_timer`. The countdown timer is the grace period going down to zero. Whenever the player is grounded, you set the timer back to its maximum.

Code: Pseudo Code for Coyote Timer Ticking Down

```
if (!onground)
{
    jump_grace_timer -= 0.1f
}
```

Code: Pseudo Code for Coyote Timer Going Back to Maximum

```
if (onground)
{
    jump_grace_timer = 0.3f
}
```

If the player presses the jump button while it's greater than zero, you allow the player to jump whether the character is grounded or not. Let's see how the code for the jump looks now.

Code: Pseudo Code for Coyote Jump

```
if ( key_jump_pressed && (onground || jump_grace_timer > 0) )
{
    //jump
}
```

There's more to jumping mechanics than jumping off the cliff. Notice how you can jump only after you have touched the ground? It's a split second but it could make the player feel laggy input behavior. Players may want to jump as soon as they hit the ground. But this code only works after the players hit the ground. To allow this other behavior, you need to

do something called input buffering. If you register the jump button to be pressed while the player is still in mid-air, you need to keep the event buffering. What do we mean by buffering? It's waiting another grace period before you decide the input was invalid; otherwise you make the character jump once the requirements are fulfilled.

In short, you want the character to be able to jump if the player hits the jump button just before (0.1-0.2 sec) they should actually be able to. Let's see how to code it. Let's call it jump-buffer.

First, you check if the player is on the ground or not. If they are, add some time to the jump buffer.

Code: Pseudo Code for Jump Buffer

```
if ( key_jump_pressed && !onground )
{
    JumpBuffer=0.3f;
}
```

Second, you have the `Update` method that runs every frame. For Unity, `Update()` is a very special method. This function gets called with every frame change in game. This is the method where you want to handle your input code. Also, since this is called after every frame change, you need to consider the delay for each frame since all devices don't have the same power to render each frame at the same time. Unity does something to help. A property called `Time.DeltaTime` stores the time difference since the last update call was made. You can use it to scale your calls.

So in your case, you check each frame if the jump buffer has turned to zero. If not, you subtract a small amount from it and check it again on the next frame. From what you coded in the pseudo-code, this code will run four times/frames before the buffer is below zero.

Code: Pseudo Code for Jump Buffer

```
void Update()
{
    if(JumpBuffer <0)return;
    JumpBuffer -= 0.1f;
}
```

Finally, in your onground method, you just need to add if the jump buffer is greater than zero when the character is grounded to do the jump.

Code: Pseudo Code for Jump Buffer

```
if (onground || JumpBuffer>0)
{
    //jump
}
```

This will make the jump button feel a lot smoother. Players can even hold the button to get a bouncy feel.

This isn't the end of your jumping saga. There's more. Players sometimes expect the character to jump higher if they press the button longer/harder. Pressing the button harder shouldn't really do anything extra but you can comply with the press-for-longer-and-get-a-higher-jump desire.

So how do you do that? Let's check out the C# code for Unity. You have two variables to manage. One is the charger that stores how long the button is pressed and a discharge boolean that is true as soon as the button is released:

```
private float charger = 0f;
private bool discharge = true;
public Rigidbody rb;
// Update will receive the input data
```

```
void Update () {

        // If pressing Space, charge the variable charger using
           the Time it's being pressed.
        if (Input.GetKey(KeyCode.Space))
        {
            charger += Time.deltaTime;
        }

        // On release, set the boolean 'discharge' to true.
        if (Input.GetKeyUp(KeyCode.Space))
        {
            discharge = true;
        }
}
```

Reference: https://answers.unity.com/questions/1459064/long-press-for-charged-jump.html

Now you are going to make the character jump or fall down depending on those two variables.

While the player holds down the button, you want the character to get increasing jump force. So you continuously increase its velocity to its rigid body component. (Note: This must be assigned from the inspector or from the `start` method.) Continuously increasing its velocity ensures that the character jumps higher as the player holds down the button.

As soon as the player releases the jump button, you want to stop adding force to it.

For this, you can use `FixedUpdate` instead of the `Update` method. Why? `FixedUpdate` is also a special method given to us by Unity. It is very similar to the `Update` method but instead of calling it every frame, it is called on fixed intervals in real time. So if you wish to update something on a fixed interval, this is the method you want to use. Also, `FixedUpdate` is basically called by the Physics Engine. So any physics-related code goes here since

all the physics updates are done before this is called. The code you are going to do next needs access to some of the physics-related features, so you use `FixedUpdate`.

```
// Using fixed update since we are dealing with Unity's Physics
private void FixedUpdate()
{
    // On (discharge == true)
    if (discharge)
    {
        // Set jump force and give new velocity
        jumpForce = 10 * charger;
        rb.velocity = new Vector2(rb.velocity.x, jumpForce);

        // Reset discharge and charger values
        discharge = false;
        charger = 0f;
    }

}
```

Reference: https://answers.unity.com/questions/1459064/long-press-for-charged-jump.html

Now, this code has a little bit of a problem. It will keep adding velocity as long as the player holds down the button, which is bad. You don't want to increase the velocity indefinitely. There should be maximum velocity that can be reached. So you need to add some sort of clamping code. Here's how you can do it in Unity:

```
float jumpForce = Mathf.Clamp(10 * charger, minforce,
maxForce);
```

`Mathf.Clamp` is a Unity library. It clamps a value between its min and max value. As a result, you will never go below or above certain thresholds.

How does `Clamp` work?

It clamps the given value between the given minimum float and maximum float values and returns the given value if it is within the min and max range:

```
public static float Clamp(float  value, float min,,float max);
```

This concludes your saga of how to not mess up the jump mechanics. Combining these solutions in one code is a good exercise for you. We strongly encourage you to code in a small character controller with these jump features in mind.

Game Mechanics: Health

You must be wondering how anyone can mess up a health mechanic. It's because most developers don't mess up this mechanic. As a result, the game loses some of its thrills. How come? Glad you asked.

Imagine you are playing this game for hours. You are almost at the checkpoint. And you are low on health. And suddenly some small fry enemy lands a blow on you and you are dead. Wasn't very fun, was it? Or how about this: you almost kill a callous boss, but just before you place your final blow you get hit and die.

It's not really fun to die, is it? You as a developer aren't here to make the player's life miserable; you are here to give them a good experience. And for that sometimes you need to cheat.

Let's tell the same story again but with different outcomes. You are low on health. You have almost reached the checkpoint. The wild enemy appears. You beat him and reach the checkpoint with low health. How do you feel? A sense of achievement, right? What about the boss fight? How do you feel when you defeat a boss with almost no health? Yeah, you don't have to say it out loud. It's thrilling. And it's fun.

But most players aren't that good. They die a lot. And the chance of a player being able to clear the boss with low health is pretty low, unless you step in. There are a lot of ways you can assist the player in overcoming this. But if the player knows you are assisting them or at least if it's too clear, they may not feel a sense of achievement at all. For this reason, you need to hide some cheating in-game code that's there to assist the player.

Let's take a look at normal damage code in a game.

Basic Take Damage Script

```
float maxHP;
float currentHP;

void ApplyDamage(float damageToApply)
{
  currentHP-=damageToApply;
  if(currentHP<=0){
   //Player dying code
  }
}
```

This is the simplest form of managing how to kill the player. The function takes a damage value and negates it with the current HP and checks if the HP has gone below zero. If yes, then it kills the player. Now the question is, should you just kill the player that easily? How do you make the player be on edge?

The concept is pretty simple. You don't always apply the full damage. You cheat here. One way is to scale the damage depending on how much health the player has. You can just multiply the damage with the remaining health percentage, something like the following code.

Improved Take Damage Code

```
float maxHP;
float currentHP;

void ApplyDamage(float damageToApply)
{
  currentHP-= (damageToApply * (currentHP/maxHP));
  if(currentHP<=0){
   //Player dying code
  }
}
```

This way the lower the health points of the character, the lower the damage. But hold it there. This is actually a bad idea because at 1% health, the damage will be almost close to zero after scaling it down that low.

You are certainly on the right track but something is wrong. Remember that clamping method from the last topic? Yeah, you need to add it here. You need to ensure that there's minimum damage even after doing the scaling. You can change that value depending on your game. Let's see how the code looks now.

Even Further Improved Take Damage Code

```
float maxHP;
float currentHP;
float minDamage=10;

void ApplyDamage(float damageToApply)
{
  damageToApply = Mathf.Clamp(damageToApply * (currentHP/
  maxHP), minDamage,damageToApply);
  currentHP-= damageToApply;
```

```
  if(currentHP<=0){
   //Player dying code
  }
}
```

In the `clamp` method, you first pass the damage to be dealt with after scaling it with the remaining HP. Secondly, you pass a minimum damage value. As the health goes low, the damage will start to go very low. You want to avoid this as well. And finally, the damage to apply is sent as the max damage since the highest damage that should be done is the exact same value without scaling.

Now, this code will ensure the better survivability of the character. Also, this doesn't expose that you as the game developer are assisting the player. It could hurt their pride if they ever knew.

No, this is not the end. You can further improve this. How about offering one last chance? How about you make sure the player always gets one last chance before dying? You just need to make sure if a hit is supposed to kill a player, it should instead lower the HP to the minimum value without killing them. For one time. Let's see how the code looks for this:

```
float maxHP;
float currentHP;
float minDamage=10;
bool dyingBreath;
void ApplyDamage(float damageToApply)
{
  damageToApply = Mathf.Clamp(damageToApply * (currentHP/
  maxHP),minDamage,
 damageToApply);

  if(currentHP-damageToApply<=0 && !dyingBreath){
     currentHP=1;
```

```
      dyingBreath=true;
     }else{
      currentHP-= damageToApply;
      dyingBreath=false;
    }

  if(currentHP<=0 && ! dyingBreath){
   //Player dying code
  }
}
```

This puts an additional check before applying damage. If the applied damage would kill the player, it just makes the player's HP as 1 instead of applying the damage it otherwise would. And the flag `dyingBreath` is a check to trigger this feature. Once `dyingBreath` is true, it will always deal with the damage next time.

You can even go further by allowing the player to take more than one hit. But it's a bad idea to allow more than three hits because the goal is to assist the player without ever letting them find out. There are other examples of assisting players when they are low on health.

In *Spider-Man PS4*, you can acquire an ability that slows time when your health bar is depleted up to a certain level. It's a predetermined features to make sure you have a balanced health system.

Borderlands has a mechanic called second wind: whenever you die, you get to shoot enemies from a crawling state. And if you can kill any enemy from that state, you get a health refund. This resets every time you successfully come back from death.

Some games take it another step further. They provide this last chance as some kind of power that you can acquire. You can either acquire it as a reward by doing some in-game objective/mission or you can purchase it with some coins/XP you acquired in the game. Sometimes this power comes with a particular tool. If you avail that tool, you get this last chance

ability as an associated attribute. *God of War* has different kinds of resurrection stones. Each replenishes health by its own capacity as soon as you die, so you can resume the battle from there without having to start over.

Game Mechanics: Movement

At this point, you should understand we aren't kidding. There are a lot of ways to mess with every simple mechanic possible. But just like jump, this is mostly the fault of gamers. You are going to add features to movements that aren't supposed to be there, but they need to be there to help players to get a better experience.

Almost all 2D platformer-like games have visual obstacles in games that weren't intended to be a part of the gameplay. But often developers make the mistake of adding physics there. Even if it's almost ignorable, this will ruin the flow of the player. Let's take an example from *Dead Cells*. They did something very interesting to help the players.

Figure 11-2. *Small environmental collision*

See that small bump on the edge in Figure 11-2? This part was never meant to force the player to jump. But if you don't have any kind of helping mechanics in place, this is what will happen: see Figure 11-3.

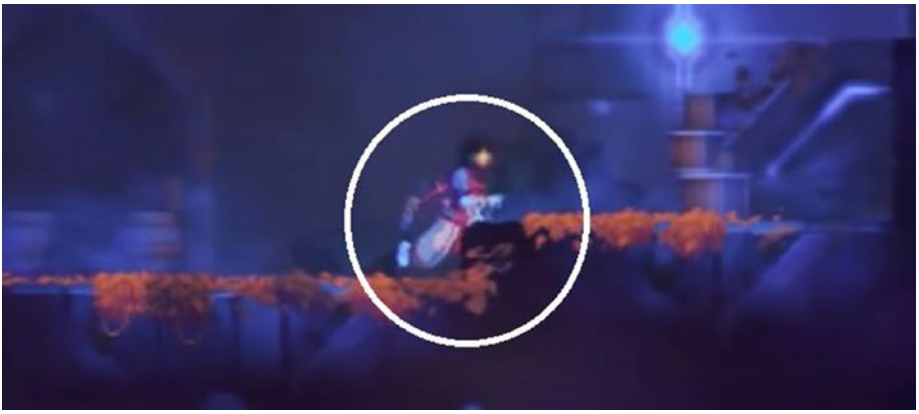

Figure 11-3. *Player colliding with the environment*

Yes, the player is stuck and they have to jump to actually move forward, as you can see in Figure 11-3. Instead, what you should do is push the player forward if they are colliding with something small in height. That's actually how it works in Dead Cells. How did the devs it? We do not know but we can certainly try. Lets try that now.

If you are planning to do this in Unity, it can get a little complicated as we don't usually handle physics collisions like this. But you can certainly cheat to make this happen. The problem is you do not want the player to get stuck here. Let's see the fundamental reason behind this problem. By default we will use box colliders because they make more sense. And it would sort of look like Figure 11-4.

Figure 11-4. *Generic collider design*

Now the character itself has a collider. And by default, it does what it's supposed to do: collides with each another. And without any other helping mechanism in place, the character collides with the small edge shown right now.

Here's a quick fix for the problem. What if you didn't make such a pointy edge? What if it was a slope? No, I don't mean changing the game design at all. Players can't see the colliders anyway so what if you did it as shown in Figure 11-5?

Figure 11-5. *Better environmental collider design*

No player will notice this, but it bypasses your problem and gives the player a better experience here. The goal of this chapter was to help players without letting them know that you are lending them a hand.

But this isn't a very good way to solve this problem. The reason you had this problem in the first place was you were using default colliders. And in most cases for the level building, you would. It'd be bad to leave these crucial helping mechanics on the level designer's hand. Because we are human and we make mistakes. Although this is a quick fix to the problem, this is not a great fix. What you should do is handle this with code. And the code looks somewhat like this:

```
//NOTE: THIS IS JUST A PSEUDOCODE
void OnCollision(collider coll){
     if(coll.rect.y<=someThreshold){
         //The player should be able to move forward
      }
}
```

In Unity, just like `Update` methods, there's another special method called `OnCollision`, which gets called whenever a game object collides with another game object. Other game engines most likely have something very similar. In Unity, the `OnCollision` method brings in some data where you can get the size of the collision:

```
coll.rect.y
```

This represents the rectangular height of the collider. Since your goal is to let the player move forward if the collision height is a very small threshold, you are going to check if it's below the threshold or not. If it is, you can write code that allows the player to move forward from there.

This pseudocode is here just to give you an idea about how to implement the feature. For each game movement, the logic will be different. So we are leaving that space empty for you to fill up as you develop your own awesome game.

The Cavalry Will Arrive

By now it's pretty clear that almost all of the fundamental mechanics in a game are vulnerable. Rewards and punishments both are necessary, but you must not overdo either. If you try to overdo rewards, the gamers will lose interest since there's a very little challenge. That's very basic. Also, you shouldn't just piss someone off in a way that they get rid of your game and never come back. You should always maintain the flow. At times, gamers may not be as good as they think. Most of the time developers help them play the game. And that's fine. As a game developer, that's your job. The goal of a game is to deliver a fun experience. If a little help from the developers makes the experience better, why not do it?

CHAPTER 12

Input Matters

Now that you understand feedback, which we talked about in detail in Chapter 10, you need to focus on something that most game developers are starting to forget. By the end of this chapter, we will have talked about the following things:

- Cognitive mapping
- Input devices
- Why do we care?

We can see that the new game developers and big companies are suddenly shifting a lot towards mobile games. Let's not go into the argument on whether this is good or bad. Let's talk about why it's not a good idea to have everything developed for mobile by default and why the input matters in this case.

For this, you need to know about cognitive mapping. It's a term used in HCD (human-centered development) or UX (user experience). So why do we need it here? What is a game but an experience? Of course user experience matters here. Have you played a game that had confusing controls on a phone? Or a game that made you feel like you need additional attachments to get a better experience?

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_12

What Is Cognitive Mapping?

This is a vast topic. Also, the definition itself might be complicated to understand out of the blue. Instead, we are going to give examples so you can relate and understand what it is.

Imagine you are in a room (inside a game) and you are asked to get out of the room. What do you do?

First, you look for a door. If you find a door, you look for a knob or a handle or something similar that will open the door so you can exit. If you don't find any handles or knobs, what will you look for? A keypad, maybe? Or scanners or sensors?

Now let's assume the room doesn't have a door. What do you do then? You look for a window. If you do find the window, you look for a handle or knob to open it and thus exit.

What if there's no window? Now maybe you look for air vents. Let's assume you can't find anything at all. It's just an empty white room like in sci-fi movies. What do you do? Maybe try to look for a hollow wall or floor?

Now comes the most interesting part. Even if we didn't guide your thought process, you were probably going to think in a similar way (maybe in a different order or with more insights, but this is a small summary of what you could have thought about).

Now let's go back to the instructions. You were asked to exit the room. No other instructions were provided. So how did you know to look for a door or the doorknob or anything that comes after that? You were not instructed to do anything yet you already knew what you had to do.

Over the years we have learned that to exit a room naturally we need a door. And a door usually has a knob or handle. If it's a prison or something, maybe it has keypads or things like that. Usually, a room will have some windows, which could also open up opportunities for exit. This is knowledge we have gained over our lifetime.

So if the designer of the room wants the people inside of the room to get out easily, they just make a door with a knob. Why? Because this is the most common knowledge about exiting a room. This is called cognitive mapping.

Cognitive mapping doesn't always have to be something that makes things easily understandable. It can be used in other ways, such as to hide things. If you want a hidden door, what do you do? Well, there are a lot of ways you can do it. The classic hidden door is a bookshelf that opens up after picking up a specific book. Or a hidden switch somewhere opens up a secret door. Imagination is the limit here. Let's take a look at the switch example with a game.

Among Us took the Internet by storm. It is a multiplayer game where 7-10 players meet in a spaceship. One of them is marked as the imposter and everyone else is just a crewmate. The crewmates do not know who the imposter actually is. The goal of the game is for the crewmates to fix the engine and the imposter will try to kill them one by one. If the crewmates can fix all the problems before the imposter can kill them, the crewmates win. If the imposter can kill the crewmates before they fix the engine, the imposter wins. But it isn't as simple as it sounds. For starters, the imposter can't go on a killing rampage. He needs to pass a cooldown period after each kill. And in that time the crewmates can find the dead body and start a meeting in which everyone talks about who might be the imposter. After the meeting, they can vote to throw one person out of the spaceship. If the crewmates can successfully deduce who the imposter is and kick them out of the ship, they win. Now, since it's mostly one imposter vs. all of the other crewmates, the imposter gets a few additional powers. They can lock the doors of each room and sabotage some key points remotely. The main map of the game has 13 rooms. Some of them have doors that can be locked down. So let's make a chart that lists all the buttons and their respective names.

Number	Room Name	Door Switch	Sabotage
1	Cafeteria	Yes	No
2	Weapons	No	No
3	Navigation	No	No
4	Shields	No	No
5	Admin	No	No
6	Communications	No	Yes
7	Storage	Yes	No
8	Electrical	Yes	Yes
9	Med bay	Yes	No
10	Security	Yes	No
11	Lower engine	Yes	No
12	Upper engine	Yes	No
13	Reactor	No	Yes

So how do you make a UI from this chart? Well, this chart itself can be converted to a generic UI. Cells that have yes can be replaced with a button. The cells with no don't need anything. But this is kind of a bad design because when you want to close a door you need to skim through the names first and then flip the switch. And it's not easy to find the name either. Try to find "Med bay" from this list and see if you can sabotage it. Take note of how much time you needed to find it.

Let's look at a better design. This time you categorize the buttons into two sets. In Figure 12-1 you can see there are two rows: one for doors and one for sabotage. You also removed any button that doesn't have doors or sabotage buttons from its respective rows. This makes it easy to find what is where. Try to find O2 for sabotaging from Figure 12-1 and take note of your time to find it.

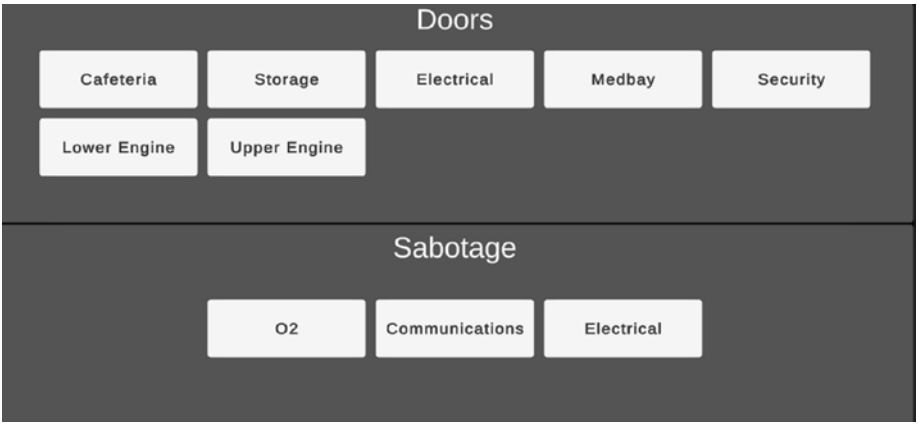

Figure 12-1. *UI design*

You should have noticed how categorizing made your search really easy. And surely the second design helped make your search for the button faster. But this can be further improved. You still need to read the text of the buttons. What if you didn't have to do this?

We can search through images faster than we can through text, so let's replace the text with icons. Although icons make the search much faster, it's not the fastest because we might not always get a well-known icon to replace text. To fix that, we need something better. Something we can relate to our subconscious mind. Something we can relate to very easily. This is where cognitive mapping comes into play. Let's see how to do this.

The game is based on a spaceship and it has connected rooms. To play the game, you need to know the layout. Eventually, you will automatically generate a mind map in your head. For example, you will know where you can go from the room you are currently in, just like you can traverse your house. Close your eyes and see if you can see a top-down view of your home. That's crucial information of your life that is mapped to your mind for many purposes. Maybe you can use that. And that's exactly what they did in *Among Us*.

The sabotage buttons were placed in their specific rooms and each door that could be closed was placed exactly in the room that it closes. But this is something you can appreciate only if you have the map in your mind, which will be after a few games anyway. To be able to quickly relate to this, you can create something similar for your home. Make a table UI, a categorized UI, and a map UI. Then try to find your rooms quickly and you will see that the fastest result was on the third one. Not just fast; you could find your buttons as soon as you thought about it because you were tapping into your mindmap.

So What's the Problem?

Here's something we end up forgetting: available inputs. Each platform has its own input hardware. Here's a small list.

Platform	Input Device
Smartphone	Touch
PlayStation	Controller
Xbox	Controller
VR headsets	HMD + controllers
PC	Keyboard + mouse

It's important that we acknowledge the weaknesses of each platform. For example, if you want to make a game that's based on sensors, you probably want to go for VR headsets because they can take movement inputs.

If you want to make a multiplayer shooting game, you should make it on a PC because in this case you need precision, which can be attained via a keyboard and mouse. It's certainly possible to just mimic the gameplay over controllers or touch but ask yourself, is that cognitively mapped?

It isn't. A smartphone's cognitive inputs are limited. Drag, tap, swipe. That's it. But for an action game you need to move your characters in four directions at least: up, down, left, and right. At least one action button needs to so something like jump/shoot/interact. A smartphone has no buttons. So what did the developers do? They made virtual buttons that change from game to game, so each game has a learning curve. But for games that are available in standard controllers like PC/consoles, players barely need to know how to move. A PC gamer knows to use

- *WASD* for movement
- *Space* to jump
- *Mouse* to move the camera
- *Ctrl/c* to crouch
- *Left mouse button* to shoot
- *Right mouse button* to zoom

All of this information has been cognitively mapped to our brains. It's the same for controllers:

- *Left stick* to move
- *Right stick* to move the camera
- A(Xbox)/X(PlayStation) to jump
- LT(Xbox)/L2(PlayStation) to shoot

It's crucial that you as a developer understand just because it can be done doesn't mean it should be done. Games are a form of art. And a good game will always reach its target audience. Forcing a game to be on a platform it shouldn't be on is a way to kill it.

Let's take a look at the most successful games on mobile devices and their mapping.

Game	Actions
Candy Crush	Swipe
Angry Bird	Tap + swipe
Temple Run	Swipe + motion
Clash of Clans	Tap

Notice that all of these games are doing their best to use the available inputs for their game instead of forcing virtual controls. If your game needs virtual controls, you may want to rethink what you are making.

Let's take a popular game on Android for example. Do you think PlayersUnknown's *BattleGround* players from mobile will stand a chance against PC players if cross-play is enabled? Even console players with controllers will be at a severe disadvantage against PC players due to the precision of the mouse. And the APM (actions per minute) will be significantly higher on keyboards than controllers. We can't even bring virtual touchpads into comparison here.

But is it always that bad? Well, it's not like PC gamers always have an edge. Fighting games work very well on joysticks. As for keyboards, you need mechanical ones to actually get the same latency, so cross-play between a PC and a console for fighting games would be unfair for PC gamers, although PC gamers can just get another joystick for their system.

Games that rely heavily on motion do much better on motion-sensitive console controls such as the ones with VR headsets or Nintendo's Wii but for simplicity let's talk about mobile devices. For example, how would you tilt your screen left or right on *Temple Run* if it was on PC? Would the experience even be the same if it was done with keys?

Then again there are some games that can work on any input system without being unfair, such as chess.

By now you can clearly notice how available inputs affect gameplay. A well-mapped input could be the sole reason whether the players will continue to play the game or not.

So what are we trying to say here? Don't develop mobile games? No. Instead, we are encouraging you to develop mobile games thinking about their native inputs. The games are being forced because you are trying to map keyboard/gamepad control on the touchpad, which is not what cognitive mapping should do.

Adding virtual controls shows a lack of effort and love towards your game. You didn't spend any time even thinking about how to make this game more accessible. Yes, you can still make it work by putting out extra efforts of tweaking the accessibility settings but this is not the default way of doing things.

A good game should be a fun experience without any addons. If your game needs the players to purchase addons to experience the game fully, then it's probably not mapped properly. Or it is on the wrong platform.

If you wish to learn more about cognitive mapping, you may want to read *The Design of Everyday Things* by Don Norman.

CHAPTER 13

Choosing a Platform

In an era of cross-platform build facilities across most of the prominent game engines, this chapter might feel a bit less significant than the others. I mean, it's pretty much straightforward, isn't it? You can simply deploy the project on as many platforms as the engine supports and get them running! Technically, you can, but should you? This chapter will seek the answer with proper justification and explain which questions to ask when choosing the platform. There are a few filters you should use to adjudicate your initial notions regarding the platform of choice. A number of variables should dictate which platform your game should be deployed to or should be deployed to first. They can include

1. Your gameplay mechanism
2. Market fit
3. Investment capacity
4. Platform fit
5. Technical issues

Input Matters: The Return and The Overnight Success

The first point to address is basically the continuation of the previous chapter, where we talked about how input mechanisms play an important part in deciding where to deploy your game. Without analyzing which kind

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_13

of controls suit your game, you shouldn't make the decision regarding the deployment platform.

If you are not very experienced yet, it can be tempting to deploy the prototype of your new concept into the gold mine of mobile gaming. It's understandable but might be unwise. Why? Let's dive deeper.

Mobile games are by far the fastest-growing segment of the industry. In 2020 alone, almost one-third of people in this world played mobile games. The industry was valued at 77.2 billion USD in 2020. New casual gamers have come onboard due to the global pandemic. In the middle of the year 2020, the World Health Organization even suggested people play games at home to remain mentally sound during this unprecedented global crisis and lockdown. So if you looked around now, you would see gamers tapping furiously on their mobile device on a hyper-casual or shouting over headphones in a battle royale. People are shouting even at 4 a.m. in their room as they find out the imposter among them while playing *Among Us* with their friends online. You don't need to look at statistics to understand that exactly one-third of downloads in Google Play and Apple stores are related to mobile games. Mobile gaming will soon surpass the combined worth of both PC and console gaming.

So it should be really lucrative, even logical, to start working in mobile games. At the same time, for a beginner, maybe starting with a mobile game feels the easier way. You can build simple 2D games with minimum investment and still be an overnight phenomenon. We wish it were really this simple. We wish somebody told us the story of John Hanke, who built *Pokemon Go* under Niantic. It took him 20 years to make an overnight success. And then there's the Angry Birds game. Prior to it, Rovio shipped 51 games in the market without noticeable success.

You have surely heard the story of John Hanke, no? If not, you can hear it now.

The Pokémon Go game broke all records. It got 10 million+ downloads in the first week, exceeding Twitter in daily active users, and with a higher average user time than Facebook, Snapchat, Instagram, and WhatsApp.

Let's track the creator's journey from 1996 to 2016, when the game got released.

While John was still a student in 1996, he co-created the very first MMO (massively multiplayer online) game called *Meridian 59*. He eventually moved on to world mapping. The breakthrough came in 2000, when John launched Keyhole, the first online, GPS-linked 3D aerial map of the world after he found a way to link maps with aerial photographs. In 2004, Google bought Keyhole and John helped turn Keyhole into today's Google Earth. John created GPS-based games afterwards. He ran the Google Geo team from 2004 to 2010, creating Google Maps and Google Street View. These six years also built the foundation of the team that would later create *Pokémon Go*.

John launched Niantic Labs in 2010. It was a start-up funded by Google to create a game layer on maps. John explained why he called it Niantic.

"The *Niantic* is the name of a whaling ship that came up during the gold rush and through a variety of circumstances got dragged onshore. This happened with other ships, too. Over the years, San Francisco was basically just built over these ships. You could stand on top of them now, and you wouldn't know it. So it's this idea that there's stuff about the world that's really cool but even though it's on the Internet, it's hard to know when you're actually there."

Do you know the predecessor of *Pokemon Go*? It was Niantic's first geo-based MMO, *Ingress*, which got released in 2012.

John explains: "In the case of *Ingress*, the activity is layered on top of the real world and on your phone. The inspiration was that it was something that I always used to daydream about while I was commuting back and forth from home to Google."

"I always thought you could make an awesome game using all the geo data that we have. I watched phones become more and more powerful and I thought the time would come that you could do a really awesome real-world adventure-based game."

In 2014, we got a surprise on April Fool's Day when we found Pokémon creatures on Google Maps. Google and the Pokémon Company made

this together only to find out that it had become a viral hit. So John took inspiration once again.

John finally decided to build *Pokémon Go* on the user-generated meeting points created by players of *Ingress*, and the most popular became the Pokéstops and gyms in *Pokémon Go*.

As John says, "The Pokéstops are submitted by users, so obviously they're based on places people go. We had essentially two and a half years of people going to all the places where they thought they should be able to play *Ingress*, so it's some pretty remote places. There are portals in Antarctica and the North Pole, and most points in between."

Between December 2015 and February 2016, John raised $25 million from Google, Nintendo, the Pokémon Company, and other investors and put up a team of 40+ to launch *Pokémon Go*. Finally, on July 6, 2016, *Pokémon Go* was released in the USA, Australia, and New Zealand. The overnight success of *Pokémon Go* took John Hanke 20 years to create. (This narrative is inspired by a Facebook post by Roger James Hamilton.)

The Budget

Now that we have established that overnight success is a myth let's take care of other pressing issues. At this point, you are probably saying that you are not aiming for such a huge, complicated game right away. You just want to make a 2D platformer or a simple infinite runner. Do you know how many developers upload games on Google Play Store and Apple Store? Around 110k! How many of them make successful games? Can you even name 110? That's right. You probably can't. The number of successful developers is very small (Disclaimer: we are not saying that the number of successful developers is less than 110). There's a common misconception where our optimistic mind tells us that we can easily make a better game than those hundreds of games in a genre available for free in the Google Play store. It is a rather emotional idea. Again, it's understandable but unnecessary. In the Google Play store, any game at the top in one genre (for example, a particular card game) takes all the honey.

The one in the second position also earns a lot, but usually, the difference from the topmost one is noticeable. From the third/fourth position, it is usually a steep downhill. If you search for flappy bird games on the Google Play store, five or six will have over a million downloads. After that, the rest have only 10,000. If you don't make it to the front page, you probably won't make that much money. Apparently, it's not that simple as it seems initially. We tend to ignore basics in the face of temptation. You should always assess the business aspect, but it is necessary to check on the technical feasibility at the same time. As discussed in the last chapter, check the control mechanism first and cross-match it with intuitive input systems of different platforms. We are not saying that you should not deploy a game on different platforms. If you have logical reasons, you should! Then again, even if you deploy it across platforms, there's always the question of priority. Which one do you prioritize more? In addition to the other factors that we have been discussing, this is also directly connected to your investment.

Different platforms require different kinds of investments. For example, while you can obtain a lifetime license for developing and publishing for only 25 USD in the Google Play store, you have to pay 99 USD/year in the Apple Store. If you are developing on Xbox as an individual, it's only 19 USD, but it is 99 USD for corporations. PlayStation still has a hefty 2500 USD price tag for its development kit, and Nintendo charges 450 USD. Also, if you are developing for consoles, you might need a greater team and thus a bigger financial capability. The financial capability should include platform fees, development costs, and marketing costs.

So, your budget can play some role in deciding which platform you want your game on, or at least where you want to focus first. But please don't let this factor overwhelm you. Always think about your game mechanics and fanbase. If the control system of your game is perfect for the PC, then launch it on the PC. You don't want a dev kit, perhaps? Then launch it on `itch.io` or any other platforms like it. Of course, you can always choose steam if you can afford a 100 USD fee on top of each game you upload on it.

Your Audience

Also, ask yourself, who will play your game? Can they be found on this platform? How much will it cost you to reach them? The marketing costs are an integral part of your budgeting. Oftentimes we see a fundamental mistake of the budget estimate only including development costs. This is common among first-timers. They have a utopian concept where they think the game is going to market itself. Well, if your game is good, it will, of course, impress people. But the number of games that have been successful without any proper marketing budget is really few on crowded platforms. Whenever you want to stand out among your competitors, marketing is necessary. As consoles receive fewer game submissions, you might have better odds there with little marketing activity in comparison. Still, as soon as you move into PCs, the necessity for extensive marketing increases a lot. On mobile platforms, it is usually a madhouse. There are costs to acquire users. You have to invest in getting more customers, like putting ads in other games through ad platforms. Do a thorough study of user acquisition costs and other activities to market your game properly. If you want a head start on marketing tips and hacks, you will find them in the next chapter.

Technical Issues

Also, if you don't have any issue with investment, it might seem logical to buy the fastest machine to try your games out. If you are deploying on PlayStation, you might be thinking of testing on a PS4 Pro or PS5; if you are on Xbox, the choice might be Xbox One or Xbox Series X. You want to see how amazing your game looks and how fast it performs on the machines of highest configuration. There's no harm in it, but you must test your game on the lower-end configurations as well because you need to make sure that there are no performance issues on them. Think about the gamers

who will play your game. A lot of them will still be using base devices. So, whatever you do, don't forget to test your games on lower-end machines!

If you are making a VR game, your best bet is console or PC. Yes, you can always do a mobile VR game. But best of luck finding a successful mobile VR game that surpasses *Beat Saber VR*. Virtual reality is all about the amazing graphical world you can create in a smooth experience. It is challenging to do so on a device that has significantly lower processing power. Even if your game reaches a niche audience, if you release it on PC VR or console VR, it has the potential to give you the best success since the mobile VR platform is still very random and immature. You can only play a VR game for so long on your mobile.

If you are making your first game, you should focus more on the game itself, not the additional work. Mobile devices have limited processing power compared to consoles and PCs. Mobile devices vary among themselves a lot as well, from low-budget Xiaomi phones to Google Pixel or Samsung's flagship models. When you upload your game to the Google Play store, you are targeting users from both categories. This can be tricky at times. Have you tested your build in the lowest configured phone in the market? There's a high chance that your game will have performance issues because you haven't done the optimization yet. Oftentimes we upload our game after testing it on our mediocre handsets, which clearly don't represent the lowest segment. If you are really going for mobile games, you should get the most inferior device on the market and test the game on it. If the performance isn't satisfactory, you will have to optimize.

When you are done optimizing, the game isn't over yet. There's always another secret. In this case, test it on the higher-end devices as well. You don't want your game to look substandard on them. So, balancing might be required. You can't afford any lag/performance issues on lower-end devices (there are many, considering the wide range of phones available), and at the same time, you can't lose the interest of the gamers with higher-end devices. The balancing can take time to master, and it can be hectic work. So, the question you need to ask yourself is, do you really

wish your first game to be on mobile? It has a significantly higher user base, and the promises of returns might be higher in cases, but what are the odds that you will be able to tap into that on your first go? Of course, you could always choose another platform that doesn't require as much optimization.

All we are saying is, when choosing your platform, don't rush or take anything for granted. Dig the facts out and match them with your situation. Then make the decision. There's no right or wrong here. The royalty percentages also vary from platform to platform. If your game is good, you can be a winner on any of the platforms. There's an ongoing debate regarding royalty percentage. Some people want to dictate platform choice based on the percentage of royalty the platform offers. That's another weird way of thinking. If your game is good and you have done proper marketing, it has a high probability of making you money. It's not guaranteed, of course, but it increases the odds in your favor by a significant margin, and we should take whatever we can salvage here. All your game needs to do is be a proper fit, so focus on that. If your game is a better fit for a particular platform than others, you should focus on it first. 10% of one million is always better than 50% of 100k. Your game can't do much if it's not a good fit for a platform, no matter how much share/royalty you own.

CHAPTER 14

Game Testing and Publishing - The Great Filter

You have come this far. Only the final chapter remains after this. You may think that the hard part is over. In that case, we ask you to look into some recent big titles you have played. The common and expected style right now is to use the last chapter to either give some sort of closure to the player or throw in some hints regarding the sequel. The final boss fights are usually put in just before that. Hence, the chapter before the last one is usually... tricky. Okay now, don't be scared. This is more like a manual, rather than a game. We won't drop a bombshell on you suddenly. But trust us when we say that game testing and publishing is one of the biggest concerns you should have in mind. So far, we have covered the part of the development journey. Testing and publishing are integral parts of it. So, in this chapter, you will discover these parts of the map. By the time you have finished this, you will have learned about the following:

- The paradox and probability estimation of making games successful
- The position of game testing and publishing in the whole lifecycle

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_14

- Good and bad practices regarding game testing
- How to approach the publishing of your game
- Things you should keep in mind while planning game marketing

Where Are They?

You have been rocking it so far. You have the intent on point, you've analyzed games in the genre you're working on, and you've carefully and rigorously designed and developed your game while being cautious about the pitfalls. Let's say you have done it all on a perfect note, just like some YouTubers who try to finish a game without taking any damage and brag about it later on as you watch the video in wonder. Now what?

Your game is supposed to break all the records, right? Theoretically, yes. But then you look into all the numbers on the other side of the coin. You become astonished to find out that statistically, almost no game makes it through. The success ratio is so bad that, even if you make 10 or more games, only 1 of them might click. The rest probably won't. In fact, most likely, none of your games will make it. In the previous chapter, we discussed that the success ratio varies from platform to platform. Console games with physical releases have a better success rate, but they are also difficult to make, whereas mobile and PC games have astonishing odds when it comes to percentile. Regardless, the question arises...what makes a successful game? What do they do so differently? Where are they?

Where is everybody?

That is the question Enrico Fermi asked while he was wondering if there was any intelligent civilization somewhere in space. If there is, why haven't they communicated with us yet? Fermi was a famous physicist. These questions became the famous concept called the Fermi Paradox!

People have tried to answer this from different angles. Some suggest that we are just animals in a zoo observed by one or many advanced races. Just like characters in a game who can't interact with the people playing the game, we can't communicate with our superiors. Some say that we are just living in a simulation or a dream. These were just mere tales for science fiction. But nobody could give a satisfactory explanation until Robin Hanson came up with a fascinating concept. This is called The Great Filter. We will get into that soon.

Like the game development process, our society also has some definite steps of evolution. It all starts with a singular cell being formed. It eventually builds and combines with different cell and chemical structures to make living organisms. As soon a multi-cell organism forms, possibilities open up for advanced living beings. In some period of time, there are intelligent creatures that can build things and communicate with each other in multiple manners. Among those intelligent creatures immerges a leader. The leading creature/species dominates the whole planet and dictates how things will roll out. On Earth, we humans have reached this step.

Now the question is, where do we go from here? The Earth is done and dusted, so we need to expand our horizon into space and interact with other civilizations. That's what we have been trying for the last 60 years. The universe is billions of years old. The scientific calculations suggest that there are hundreds of millions of Earth-like planets in the Milky Way alone (Figure 14-1)! But as of yet, we have not reached other species, nor have they reached out to us. Our sun is relatively young, but other solar systems are relatively older. Why is it that they have not reached the stage where they can communicate with us? *Where are the aliens?*

Figure 14-1. *In our Milky Way galaxy, there are 300 million possibly habitable planets at least. (Image courtesy: Kurzgesagt)*

You have completed 13 chapters of this book. You have thought about a game and made it into reality. You need to test the game and release it on different platforms. Let's say you release it on Steam. You want to set a high price tag because you have invested a lot and want to get your money back. Mainly, you believe that your game deserves it. It's alright and understandable. The average indie game doesn't even cost 10 USD on Steam. So, it's a risky strategy if you want to charge a premium price for your game while going out of the industry-standard range. Thinking deeply, maybe you want to come back to a more lenient price. Maybe not. It is a secondary concern. The game is a unique product. The pricing matters, yes. But you are selling a product that requires time investment from the buyer as well. I can afford to buy a hundred games this week but can hardly finish one.

So, the first question the buyer asks in their subconscious mind is, "Will I have time to play this?" If your game can't pass this filter, it's highly unlikely that they will buy it. (If it is free, they might keep it in their library,

but even then, they most likely won't ever play it). This is the reason why it is so difficult to sell a game. You have to convince the buyer that it will be worth their time and money both. There can be 100 other games people can choose to play in this genre. Why would they buy yours? How do you make a game popular and successful?

Well, you're thinking, if only I had a brand name. But let us stop you right there. What comes to your mind when you hear about any *Call of Duty* game? This famous series by Activision is one of the top names in the history of the game industry. But when they released the trailer for *Call of Duty: Infinite Warfare*, it was hit with a huge backlash from the community. In fact, it was disliked in a record number on YouTube. Nobody liked the futuristic galaxy setting they were trying to pull off. Activision came up with some strategies (releasing it with *Call of Duty 4: Modern Warfare remastered*) to control the damage. Still, it got only half in sales compared to its predecessor, *Call of Duty: Black Ops III***,** and Activision acknowledged that *Infinite Warfare* failed to fulfill the sales expectations. As you can see, there's no guaranteed success! Yes, a brand name can work as a boost, but you have to deliver as well. Also, you can't have a brand name with your first game, right? You have to have some success first to build your brand identity. It feels like you are in a paradox. How do you make a successful game that everyone likes?

The answer to the Fermi Paradox varies from person to person. People like to interpret it in their respective ways. As game developers, we keep wondering about how a game becomes phenomenal. You could also ask some legendary game makers. All of them will have their own answers. Don't get disheartened. It's a paradox, after all! It won't be easy to solve.

Some of us could argue that no game has yet reached its full potential. There's no concrete and 100% proper answer to the Fermi Paradox. There are only possible explanations. Some people answer this with The Great Filter. Let's talk about this now.

The Great Filter

The Great Filter is basically a probability barrier. It denotes that there might be some inevitable phenomenon or level that prevents life forms from developing to the point where they can communicate or travel across interstellar distances. Think of it as a stage that every civilization/species encounters eventually, but it is very difficult or impossible to pass. Maybe no species has made it past the Great Filter, and that is why we don't have any verified alien sightings or communications yet (Figure 14-2).

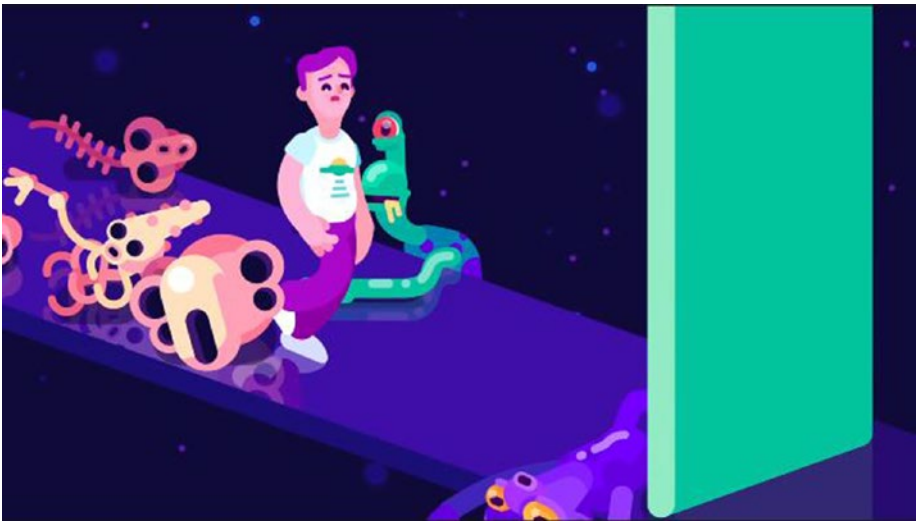

Figure 14-2. *The Great Filter, behind or ahead of us? (Image courtesy: Kurzgesagt)*

In your case, you can apply the same concept. You may have passed all the previous steps very well. But since you still can't access success, there might be something that's missing in the whole process or something that's blocking your journey to the next level. What is it? We can post a

hypothesis regarding this in game development. To dig this secret out, let's ask you a question: What is the goal you're trying to achieve here? The most likely answer is, "I want people to like my game and maybe pay for it or spend time in it." You have played your game so many times, and you like it. That's alright. Now you want other people to like your game. You have done the experiments yourself; understandable. But what about the other people whom you want so desperately to come into your kingdom?

Have you asked them and taken their suggestions? Maybe one. Two. Ten. That's probably it. Ten people have given you their feedback, and based on this information, you have made adjustments in your game.

But you want 10 million people to like your game. Do you think these 10 people represent the likelihood and choices of the 10 million you are aiming to get? If yes, then congratulations. But most likely, it will be a no. Hypothetically, think of this. What if you could let all these 10 million people try out your game, give feedback, and then you could magically accommodate all their suggestions in your game? Practically, it's not possible to adjust so many variants in a game and still keep it commercially viable. It may be technically impossible as well. Regardless, theoretically, if you could do so, what would be the result?

The perfect game for all those 10 million users.

The Great Filter in game development is game testing and publishing. If you have tested your game well enough and acted accordingly, then go on to publish your game. You can pass the filter and go for glory. (Well, most of the time.)

Basic Steps of Game Testing

Just like any other software development, game testing has some definite steps. We will only cover some basic ones in this book to get you started.

Unit Testing

Unit testing is the test run by the developers to check whether their program/code/game is running properly and doing what it was initially supposed to do. It comes before anything else, especially the dedicated testers laying their hands on your game. It is called unit testing because the developers test to see if the "units" or features are working properly. It is basically checking whether your code is doing what it is supposed to do.

Alpha Testing

Alpha testing is conducted by the internal employees of an organization. In our case, the game is tested for user experience, but by the testers of the studio. There are a couple of testing techniques here: White Box and Black Box. White Box testing includes knowledge about the internal structure of the program, and the testers do the testing while having this knowledge. In Black Box testing, however, the tester has no idea about the internal structure of the product. They only test the user experience as a common user. Alpha testing occurs on-site within the lab, before rolling it out to the common users for beta testing.

Beta Testing

Beta testing is done by external users who play the game for real. They don't have any idea about the internal structure of the game, so the Black Box testing technique is used. It consists of real users and their honest feedback.

These are the basic steps only. There are other detailed approaches regarding game testing. You can study more if you wish in any software development resource that is available to you. Any books or videos that you can get your hands on regarding software testing will help you gain more detailed knowledge about testing.

While doing the testing, make sure you do it in a structured format. The last thing you should receive as a developer is an unstructured bunch of feedback in the form of a huge essay that has individual formats. It will just demotivate the developers and harm their productivity. There should be a list of points and questions that are answered through the tests. The list will depend on the genre of the game. Incorporate as many playtest as possible and be as detailed as you can. There should be many iterations to make sure there aren't any hidden bugs waiting to happen after you do something unorthodox in the game or repeat an action a certain number of times. The testers should provide recommendations along with their observations whenever they can as well. For example, "The player HP remains too low at the beginning of this boss level. Could you consider offering some health items to pick up at the beginning?" Maybe you won't take the suggestion, but it will help you understand the problem better. Perhaps you can implement the suggestion as well!

Hidden Traps Regarding Game Testing

Now that we have established the fact that you need to test your games rigorously to hope for success, let's see what can stop you along the way.

The Lazy Developer

You are doing the noble and tiresome work of making a game. But the bugs are restless. You keep getting some in every build. Finally, after renaming the output as the nth time, you get a version you like and test it yourself. "Ah, finally!" you shout out because you can't find any major bugs anymore. It's time to sleep, finally. Also, it was a mammoth task to pull off the project. Who has the time to do testing again, right?

That's the first mistake you make as a beginner-level game developer. We understand the struggle, but in all honesty, game testing is equally important as the development of the game. You should make time for

testing it, over and over. If you are the developer, your testing is essential. You will know many hidden bugs that might exist, which a general user might not find so easily. After development, you should make time to test your game rigorously. If you have trouble noting things down, get help from someone to do so. But do test your game over and over again even while you are lying on the couch and relaxing.

It's Working Fine Here

You have the latest device in your hand, and your game is running smoothly on it. What else do you need? You do the basic testing and go for publishing. Wrong…again. The players who will play your game may not have this device at their disposal. Their device might be backdated, but they do have access to the store you're going to list your game in, hence making them a potential customer. Have you checked the market trend? What percentile of people owns which phone, and have you tested your game on the configurations that most people have right now? If not, stop everything and do it.

Performance issues are the last thing you want to have in your game. Most people won't think twice before deleting your game if there are any performance issues. You should get as many devices as possible in multiple configurations to make sure that your game runs smoothly on them. If your game requires some special sensors to take the input, then you have to plan accordingly. Planning accordingly can mean a couple of things:

- First, your target audience might be very niche. That's okay.
- Second, you can tweak the accessibility setting in a way that the players who don't have those sensors in their devices can perform the same tasks by some other available inputs.

The last thing you should do is test the game on the available devices in the office and get a verdict that it's working. Just because it's working on the available devices only doesn't mean it will run smoothly everywhere. You have to test it for real. In the beginning, you might not have that sort of luxury to have this wide range of devices at your disposal. So what do you do? You can look for testers who can give you results on the particular devices that are missing. They might not be your team member. But you can always seek help from people like this here. Get your hands on as many devices as you can (Figure 14-3). Borrow from your friends and family as long as you can keep them safe. Test out your game on every variation of configuration involved.

Figure 14-3. *Different types of devices with different configurations*

The Developer Bias

Since you are the architect of your game, it can get a bit challenging at times to find certain bugs. Maybe they are hiding in some mechanism that you developed so well. You are so confident that even before testing, you know that everything should be working fine. I mean, the logic works that way!

It is impossible to have a bug in here, you think. Well, no offense, but blind spots are never good to have. Your biases towards your game can blind your vision and capability to find the tricky bugs that hide easily in your system. Of course, the developer should test the game. But they shouldn't be the only ones. The bias is almost intuitive and very hard to detect.

Prevention is better than a cure, and in terms of the developer bias, you must try to act in advance and prevent things from happening in the first place. The idea is to make a structured approach to in-game testing and fine-tuning. List the features before you do the unit testing. If you don't leave out any feature/function while documenting the feature list, you should be fine. Because even if you fall prey to the developer bias in unit testing, it will be checked in the later stages, like alpha and beta testing. As long as all the features and functions are documented, you should be okay.

The Nice Guy

You give the game to a friend, and immediately they comment that it is a very good game. You are pleased. That's okay. It's good to have positivity in life. But in this case… at what cost? When you are testing your game, you need honest and thorough feedback. The last thing you need here is generic feedback that tells you that the game is good/nice. You should look for specific feedback that you can work on. It's good to have nice people around, but they can't do you any good when it comes down to game testing. No matter what you throw on their face, they will always say, "I think it's good." This is just bad closure.

If you are looking for proper beta testers, there are many online communities and forums where you can seek help. GameFAQs, IGN, GameSpot, Discord, Steam, and Reddit are a few leading platforms. There are others as well. When choosing communities and testers from them, look for the ones that have the type of gamers that match with your target group for the game. They can give you honest feedback. Yes, they can and will be harsh on you at times regarding feedback. But that's necessary. By

being blunt, they are giving you honest feedback, which will only help you. So, put on a thick skin and use the best out of that feedback. We see popular games getting roasted all the time by gamers in these communities. Many platforms are even full of memes. Underneath all the humor, you can find valuable feedback, if you know where and how to look.

Jealousy

Game-making is not everyone's cup of tea. It's okay to have people be jealous of you. The best way to handle this is to enjoy it. Regardless, don't get too consumed with this concept. Learn to take criticism. A particular person might not give you proper feedback because they are jealous. This is harmful. First of all, jealousy is often assumed when it doesn't exist for real. It is mostly our imagination. For the sake of argument, let's say people are jealous of you. In that case, that's good news! People do stupid things when they are jealous. If they give you a huge list of issues, be thankful to them. They just became a great contributor to the game. They will do everything to find an issue in your game. Don't ignore them. Instead, keep them close.

The Bug Is Small and Hard to Catch

You have done rigorous testing of your game and solved most of the issues that have been found. You are cruising along and preparing for the release. All of a sudden, you discover a small experience issue. It's nothing serious, but it exists. Maybe you found out that the high jump is not that smooth, or the character rotation looks clunky. You try to solve it, but somehow it's not working. After wrestling with it for quite some time, you discover that the issue is very fundamental, and you need to build the game from the beginning once again to solve this problem. This is troublesome. You have probably spent days, if not weeks already, to reach this stage. Now you have to throw it all away?

Maybe you could ignore this tiny issue and go forward with the release? This is the mistake many of us make in our life. A game can have a huge number of issues. It's a fragile thing like a baby. Even if you test your game with groups of professional game testers properly, there might be some issues that you couldn't find. It's because the possibilities are limitless. In this scenario, how can you find a bug and decide not to address it? The users will surely discover it, and all the hard work you have done will be undone. We have been there. It's difficult. But you have to buckle up and break the game. Start anew if needed. But never undermine a bug/issue. No matter how minor it is, it can and will ruin your game. Then again, many studios don't have enough resources or capabilities to solve every single issue. In that case, they have to prioritize. Rather than prioritizing the issues that you can solve easily, give priority to bugs that can hurt your game the most. Then again, beware of the small and nonsignificant bugs. They can and will ruin your game. If you don't have the resources to address all of them, maybe you should think of ways to get the resources. You can try to raise more funds. No funding is enough, but having more funds can give you better odds all the time.

The Boring Job

Most probably, you have come into the domain of game development due to the severe passion you feel for games. You may be an artist, designer, writer, developer, or tester. Whatever technical job you do, we are pretty sure that you love it! You enjoy the journey of making a game. After a happening journey, you have finally made a game that you think has great potential. Well, what happens now? You need to get your game out there, right? Let's say you have gone ahead and published in suitable platforms (if you are looking for some suggestions, go back to Chapter 13). Now all

you have to do is make sure it gets the proper audience. We are talking about game marketing, distribution, etc. These are non-technical tasks. As you are a technical person, these things may bother you, even scare you. It can get boring. Getting out of your comfort zone is not an easy feat, but for those who can do it, great rewards await them. We are going to assist you a bit in choosing the right options regarding this.

It's Going to Work Out by Itself

There's a popular myth in the market that if your game is good enough, it will be an overnight phenomenon. All you have to do is get the magical baby into the world, and it will conquer the world on its own. This a romantic tale we keep telling ourselves again and again. And it's a tale only. In the competitive industry of today, you can't afford to release your game and just sit back to watch it to conquer the world all by itself. I tried doing that in 2015. I made a game, tested it, and put it up on the Google Play store in the hopes that it would be downloaded by thousands, if not millions of people. From Battery Low Games, the first game to get listed on the Play store was a 2D infinite runner with unique graphics. The gameplay was okay. We thought that people would download our game because it was free and it looked beautiful. We published it on the eve of Valentine's Day and named the game *Valen-Tine Run*, hoping to catch the hype. Even after sharing in all the social media groups, the download count merely reached a thousand. We were heartbroken. We didn't spend any money marketing the game, but we had to spend resources on our influencer friends. Despite their best efforts, the game couldn't hit any numbers at all. Clearly, the concept of working out all by itself was not working at all. Also, even if you release something on the eve of a special day, it won't work out unless you have the proper effort behind it.

Make Up Your Mind

Now that we have established the fact that you can only do so much by hoping for a miracle, let's talk about other options than luck. There are two definite ways you can go about publishing. One is you can do all the tasks by yourself and the second one is you can partner with a publisher.

The common dilemma that you face is which way you should go about it. If you choose to go all in all by yourself and publish your game as an indie game without any publisher, just bear in mind that you need to take care of all the marketing of the game. You need to spend money on user acquisition, and then go ahead and spend again on retaining those users. The upside? You get to keep it all to yourself: the rights, decision-making power, the revenue (well, not ALL of it since the platform that you're publishing on it will most likely take their cut). The downside is that it is hard to do it, and you have to manage the funds to market your game yourself.

The upside of choosing to partner up with a publisher is the offloading of pressure. You can solely focus on making a better game and let them worry about the rest. Also, you can worry less about the success of the marketing activities since the publishers are usually very good at it. The downsides. It is not an easy feat to strike a deal with a publisher. They receive a lot of good submissions every day. They have to say no to almost all of them. Only a handful of games remain.

It's a very easy process to submit your game to the publishers. You just submit a working version of your game and maybe fill out a form containing basic information about you/your organization and the game itself. Sometimes some publishers also allow or want a screenplay video of the game as well to judge. This part is rather simple. What's hard is to get selected. Well, this works when you are submitting an already developed game. There's another way of working with publishers.

You can strike a deal with publishers to fund your development costs. In this case, they take a bigger chunk of the money earned. But that's not the downside here. Since they decide how the money will be spent,

they intend to spend more on marketing purposes, and oftentimes the development budget is undercut to accommodate the ever-growing marketing budget. Also, sometimes the publishers try to change the game idea forcefully to get in the market. For example, they want to make a solo game into multiplayer for no reason or incorporate some other feature that the devs didn't want to include in the first place. If you are working with a publisher, get ready to experience this and be okay with these issues if you want to keep working with them.

Not all that glitters is gold. You shouldn't jump into signing a deal without understanding the term sheet and doing your own check on the publisher. Analyze their track record thoroughly. Dive inside the sales numbers. Have a look at their interaction with followers. If they have a huge number of followers but very little interaction in their social media activities, that might be a bad sign. When reading the term sheet, don't hold back in your negotiations. If you have doubts about royalty share duration, the right to sequels, free key distribution plans, platform confinement, and any other terms, clear them up and name your terms if you don't like the terms they have proposed.

We see many developers with a stubborn attitude, which is understandable. They want to do it all by themselves. You can always opt to go this way, but just remember that you might have to compete with big publishers, their good games, and insane marketing investments. Maybe it's not the worst idea to begin your journey with a publisher first, give yourself some ground, and then maybe shift and go solo if you must. Also, the bigger and more expensive the game is to develop, the more likely you need a publisher. In the case of physical console releases, you most likely need a publisher, whereas in digital, it might not be 100% necessary in most cases.

Whatever your decision, the last thing you should do is procrastinate behind the decision. Think about your game, do your research, and decide what you want. Do it now.

Restarting Every Time from Zero

There's a game mode in *God of War 4* and in *The Last of Us 2* called New Game Plus. It gets activated after you finish the game for the first time.

In this New Game Plus mode, you get to keep all your weapons and the upgrades you earned by finishing the campaign earlier. You have already played around 50 hours or more to collect these weapons and upgrades. Why spend time acquiring these same items once again? You just want to explore the story once again, with a different difficulty as well, maybe. This sounds logical, right?

In case you were wondering how this connects with this chapter, let's talk about that now. If you have decided to self-publish your game, then you know that there are many things you need to do to submit your game to any platform (Google Play, iTunes, Steam, etc.). They ask about game materials, descriptions, screenshots, videos, etc. Well, usually, they don't change these demands that much. Sometimes the privacy policy updates, and once in a blue moon, they ask for one or two new things. But usually, it's the same deal, more or less.

Since you will keep submitting games to these places, it doesn't make any sense to encounter the requirements anew every time and satisfy the queries individually. I wasted a lot of my most precious (when you are on edge after a huge amount of hard work) time by checking out the developer console to find out which dimension of the screenshot I need to take for the game and which graphics I need to make. The easier way is to create a simple template when you submit your first game. Then every time you want to publish a game, you just need to follow the template. It's gonna save you a lot of your precious time during store listings. For example, let's say you are making games for the Google Play store. You have paid the developer license fee (a one-time fee of 25 USD) to access the Play console. Now, what are the things you need to do to get your game listed?

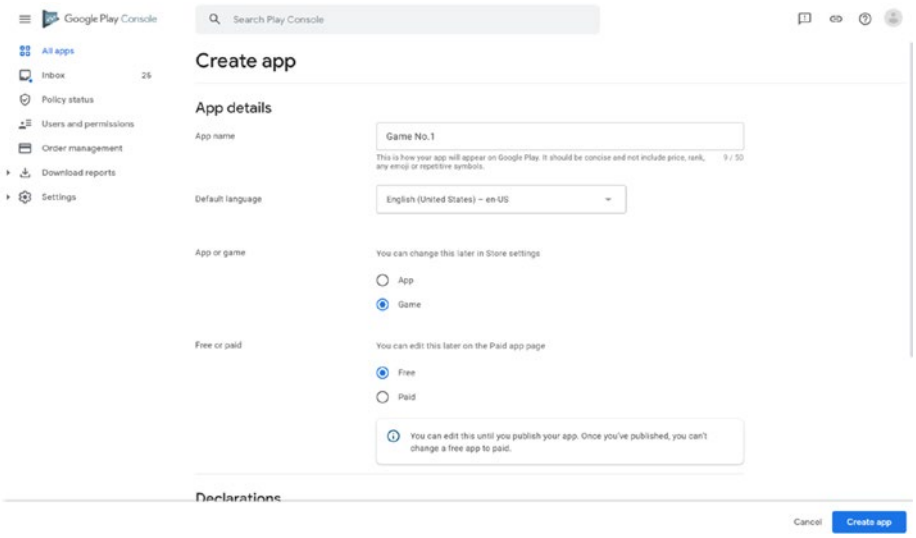

Figure 14-4. *Snapshot from the Google Play console*

Table 1. From the Play console, you need to select that you are submitting a game and note whether it is free or paid while giving it a name (Figure 14-1). After that, it will ask you to set up the app (Figure 14-5).

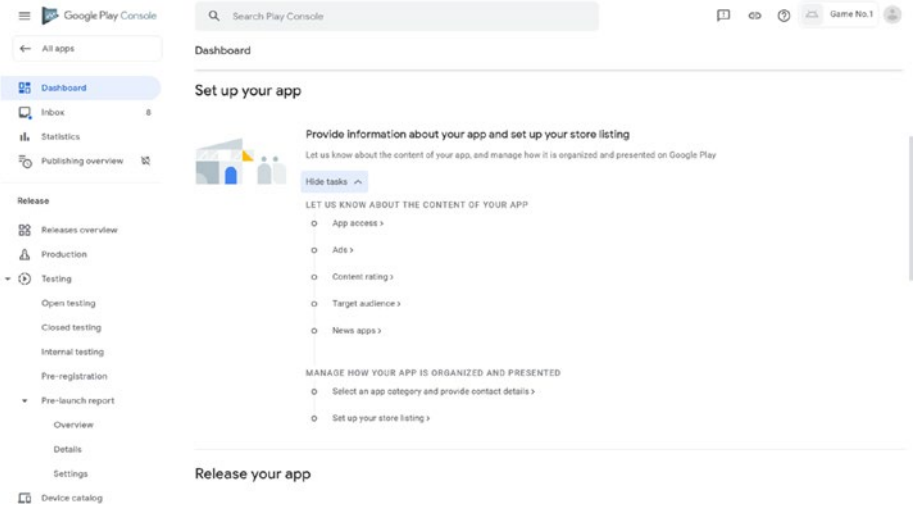

Figure 14-5. *Snapshot from Google Play console after setting up the name*

Your concern is the store listing at first. You can make a template for the store listing part. For the in-store listing, these are the things that you will need to provide:

1. App Name *

   (This is how your game will appear on Google Play. It should be concise and not include price, rank, emoji, or repetitive symbols.)

   50 characters limit

2. Short description *

   (A short description for your game. Users can expand to view your full description.)

   80 characters limit

3. Full description *

   (Full description about the game)

   4000 characters limit

Graphics

1. App icon *

   Your app icon is shown on Google Play. Don't use badges or text that suggest store ranking, price, or Google Play categories (such as "top," "new," or "sale").

   JPEG or 32-bit PNG

   512 px by 512 px

   Up to 1 MB

2. Feature graphic *

   This will help promote your app in different places on Google Play. In the case of cropping, avoid placing text near the edges, and center key visuals.

   JPEG or 24-bit PNG (not transparent)

   1,024 px by 500 px

   Up to 1 MB

3. Phone screenshots *

   Upload 2–8 phone screenshots. Your game is more likely to be promoted on Google Play if you include 3 landscape screenshots of the in-game experience.

   JPEG or 24-bit PNG (not transparent)

   Between 320 px and 3,840 px

   16:9 aspect ratio (for landscape screenshots)

   Up to 8 MB

4. 7-inch tablet screenshots *

   Upload up to 8 7-inch tablet screenshots

   JPEG or 24-bit PNG (not transparent)

   Between 320 px and 3,840 px

   16:9 aspect ratio (for landscape screenshots)

   Up to 8 MB

5. 10-inch tablet screenshots *

   Upload up to 8 10-inch tablet screenshots

   JPEG or 24-bit PNG (not transparent)

Between 320 px and 3,840 px

16:9 aspect ratio (for landscape screenshots)

Up to 8 MB

6. Video

   https://www.youtube.com/watch?v=

Add a video by entering a YouTube URL. Your game is more likely to be promoted on Google Play if you include a landscape video showing an in-game experience. The video must be public or unlisted, and ads must be disabled.

Apart from the video, everything else mentioned here is mandatory. You can provide this template to the person in charge of your content. Whenever you release a game, they need to provide you with these materials, so you won't have to tell your graphics designer the icon size during every single release.

Also, please focus on making the visuals great. Oftentimes we make the mistake of not prioritizing these graphics that much. Please keep in mind that people will judge whether to get your game primarily by these marketing materials. You should take good care of them. Even mediocre games can have good download counts if the feature graphics and screenshots are great. (We can't say the same about the retention rate, though.) I mean, we humans have this unfair tendency of favoring things that please our eyes. We tend to check on the quality later on. A video is a big plus as well. Many gamers download games primarily based on their impression after seeing the video available in the store listing.

If you want to know how to list your game on Steam, here's a blog post: https://xsolla.com/blog/monetization/2206/self-publishing-on-steam-the-ultimate-guide.

Stop Waiting

You have a dream publisher. You take your time and submit your game properly to them. What do you do after that? Well, some people decide to wait until the publisher gives feedback on it. What you have to understand is that these decisions can take some time to happen. Sometimes they can take a month or more. In the meantime, it's a waste to sit around and wait. You should keep submitting your games to a lot of publishers. You never know which one will click, do you? If you are making games for mobile, there's a good chance that advertisement revenues are one of the major concerns for you. In that case, study different ad networks and incorporate their controls into your code. Constantly feel like you picked the wrong ad provider and reiterate. Don't just take any ad network for granted. Continuously observe the performance of your game. Dive into the battle of retention and try to retain the users as much as possible. If you are in mobile games, most likely, your development cycle will just continue like this. Even after trying your best, if you don't succeed, don't worry. We are not going to reiterate the story of Robert Bruce. You already know that failure is the pillar of success. Each failed game is one step closer to your success. But most importantly, you should keep working. Do not wait for people. There's always something you can do with your game. Please don't procrastinate over others' backlog too long until it has clearly visible dependencies.

Influencers vs. Game Streamers

Many of you have good friends who are influential in their respective social media circles. They have many followers, and you can reach a good amount of people if they post about your game from their profiles. The intuition is alright. The issue is with the quality of the audience. If they

reach 100k people with a post, how many of those people are your target customer? Usually, public celebrities have a very diverse audience. It's very possible that most of their audience might not be the type of gamer you want to target at all. In my experience, they don't have a significant impact (unless you combine Keanu Reeves and a Cyberpunk game that people have wanted for 8 years, of course).

Game streamers, however, are the more logical way to go. The number of streamers is increasing daily. Thanks to Twitch and other streaming platforms, people spend a lot of time watching these videos. The viewers like to watch the streamer's reaction to in-game situations, which contributes to engagement and activity in chats. So, if you can get popular gamers to stream your game, it can be useful. Well, if you can do it superbly, it can work wonders. *Fall Guys* by Mediatonic has become the highest downloaded game on PS Plus. On July 24, 2020, Mediatonic sent out the code to popular streamers, a day before the open beta. At the start of the beta, streamers sent beta code to viewers. This shareable system worked like magic. *Fall Guys* became the second most popular game on Twitch and YouTube in the first month after release, with a total of 113 million hours viewed.

You all know what happened afterward. It has been a hit on both console and PC, has won hearts, and Season 2 was released recently. This shows how significant streamers can be!

Set your priorities straight. Don't do what you can do easily. Instead, do what needs to be done. Let's discuss a few things that need to be done while you market your game.

Game Marketing Strategies

This subject itself is big enough to write a book on. We are not going into details, but we will give you a head start into the vast domain and some hacks to help you steer around some traps.

A Few Things to Make/Have

To market your game, regardless of whatever partners you have, you will always need these basic things:

- *Trailer*: The game trailer is your wild card among all the noise out there. By now, you know how competitive it is out there. But a compelling trailer can simply ignore all the hurdles and get you a jump start. People judge your game based on the trailer you make. So, if you can make one that stands out, this will get you valuable eyeballs. A good trailer is not just some random clips of your gameplay. Instead, it must feel interesting to watch. Great visuals will indeed help, and a short but compelling story can be of use as well. Whatever you do, please don't forget to include the best thing about your game in your trailer. There's a popular talk by Derek Lieu in GDC 2019 on how to make a great game trailer (`www.youtube.com/watch?v=QWd7FOz1W_Y`). To learn more about how to do it well, you can follow him, Kert Gartner, or any other professional in the industry who does it well.
- *Wishlist*: As soon as you get the word out that you are making a game, you should start getting people to wishlist your game. Wishlisting your game means they will show their interest officially by providing their email address so that you can send them important updates regarding the game (release date announcement, new trailer, preorder announcement, etc.).
- *Landing page*: The landing page is the website for your game. It has to contain only the important information and content like your game logo, assets, trailer, release

dates, links to your social media, etc. Getting people to wishlist your game from your landing page is exceedingly important. Please make sure that you have given the sign-up option grand visibility. Also, they should be guided properly to join the community of your game on different channels and platforms.

- *Devlog*: Devlog is a blog/forum where you log your progress in making the game and share it with your community. You have to update it regularly and be as original as you can here. Don't hesitate to upload raw concept art or unstable updates as long as they are interesting. Gamedevs will look out for this content, and so will your community. If you are publishing your game on `itch.io`, then you can use their devlog section. Or you can try gamedevs.net, TIGSource Forums, or just host your own.
- *Mailing list and press*: Developing a mailing list will be of great help in the initial stages. This can be your primary source for getting an enthusiastic community for your game. You have to develop at least two separate mailing lists. One is people who will play your game and will be the early adopters. The other one is the addresses of journalists and bloggers mainly. The latter one is the one you will need to send your press kit to in the hopes that they will cover your game. If you don't have a huge budget regarding paid promotion, this is where you have a chance to shine. Many of us subscribe to different game-related blogs and sites. We see interesting articles regarding a game, and we add them to our wish list. Find as many email addresses as you can and send them your press kit. If you are

following a list that is a few years old, many email ids might be obsolete. Also, not every writer will be willing to write about your game. To increase your odds, narrow it down to the writers who already have some traction and, most importantly, have a track record of writing about games of a similar genre. You can look for their current email address on their website or social media handles. It's going to take some time to find the latest email ids of these filtered-out writers, but the time investment can be worth it. If you can't find their email id, just find them on social media. Send a message there and hope for the best. Most won't respond to you, but a few will, certainly, and with a bit of luck, that can just be enough. You can use any prominent email service like MailChimp to send out and manage the email.

- *Press kit*: When you send out emails or messages to the writers, make sure to include everything they need to write about your game in your email/message. Include your game logo, short description, important information regarding the game, screenshots, and promotional images from which they can choose when they post their piece.
- *Discord*: Don't forget to open up a channel in Discord and build up a community there. They will not only act as initial customers and fans but also can turn out to be a great community for conducting some playtests.
- *Reddit*: Go ahead and open up discussions regarding your game on Reddit. Don't post promotional stuff there or anything formal. Rather, if you have a personal story around the game or your journey, that can be great

content on Reddit. People also love small animated GIFs there. Be sure to wear thick skin because it is filled with people of all sorts, and not all of them are kind.

- *Social media handles*: It's always good to build a community on Twitter, Instagram, and even Facebook. If you have the resources to maintain them, start as soon as possible. These days you can get interns to take care of these things. They have fresh ideas and can be a bit budget-friendly as well. But do make sure you do proper justice to them and make sure to guide them.

Getting the Best Out of Your Platform

Every platform has its own ways of helping developers out. We see different sections like Popular Games, Top Games, and Featured Games all the time. Talk to your platform and find out how you can get your game featured in those lists. They may run some programs to promote indie game devs as well! For example, Nintendo has the Indie World Initiative and the Nintendo Power Podcast. They also promote many indie game trailers on their official YouTube channel. Xbox has a program called ID@Xbox to promote indies on its platform, offering a range of promotional options. Some platforms even have built-in marketing tools. For example, Steam has Curator Connect and Visibility Rounds. We often see games being sold at a better rate during any sales promotion. You should actively participate in these options with a proper strategy regarding the timing of your sales campaigns. You can always participate in the common major sales campaigns that they have. There are Winter Sales, Halloween, Black Friday, etc. While everyone tries to get their game on a major sales campaign, only a limited number of games can participate in week-long sales deals. Make the best out of them. If you get lost in the huge content on the websites of the platforms, ask them for specific details regarding how you can utilize

their tools and programs to promote your game. It's usually recommended to visit the sites and read all about those policies. Don't leave out any details, regardless of everything.

Game Engines and Dev Communities

The engine that you are working on can provide fantastic opportunities to get your name out there. If you can show them that you have a compelling game coming up, they will talk about your game on their official channels, and even feature it on their shows/events. Wouldn't it be awesome to get a place in their booth in a global event? The best-case scenario would be to receive an award.

The indie dev community in this world is a blessing. You can connect rather easily. Simply connect with them via Twitter, Discord, or any other platform and try to be friends. You can promote each other's games intuitively. This won't cost you money, but having a healthy amount of connections in the indie game dev community will surely help you in some great promotions without costing a dime. Go to different indie game dev events and keep expanding your network. They can come in handy in many other ways as well. Don't hold back in sharing your learnings with them, as you will also be enhanced by the knowledge they share from their experiences. It's a totally win-win partnership for both. Also, it gives you someone to talk to about your game-making life, someone who understands your pain points and helps you feel better.

Communication Hacks

While you're trying to get someone to know about your game, try to remain as original as you can. Simple "Please play my game, here is it" might not work on most people, especially to those who deal with many requests like this regularly. An exception might be a cool trailer. If your trailer stands

out, you might get away with this at times. If you want to get people really hooked, tell them something interesting that they wouldn't expect. For example, you could talk about a unique feature of your game that could attract them. If you are talking to a reviewer/professional, you could bring up any possible synergy they might have with any segment of your game. For example, say somebody did a few articles on slash weapons in mobile action games. If your game features a slashing weapon, you can ask him, "Hey, check out the trailer of my game. I would love to know what you think about this cool slash weapon it has." These kinds of emails stand out more.

Communicating with streamers might get a bit tricky at times. Twitch is a great option to find out proper streamers who can help your cause. Facebook and YouTube gaming are also great options. Whatever platform you choose, look at the audience engagement performance of the streamers. A streamer can have many followers on Twitch but very low user engagement. In that case, it would not be a wise choice to make a partnership here. While connecting with influencers/streamers, you can just go ahead and send your offers to them via the contact channels they use. But yes, it can be helpful if you can go via someone they know. They receive many promotional requests. If you are a first-timer, a referral from someone they know can help you get noticed a bit faster than usual, and that timespan might be crucial for you.

There's another way of getting their notice. You can invite them to participate in your game in some way, like voice acting or some other way. This unique approach has a better chance of grabbing their attention. They might end up promoting your game intuitively if they get involved eventually.

While talking to professionals like reviewers, streamers, and journalists, always keep your material ready. Do send out the materials they need to promote your game (game name, basic and important info, game assets, trailer link, release date, etc.) when they show interest so that if they chose to do a feature, they don't have to ask for materials. This saves

them time and forms a good impression about you in their mind. Also, don't forget to follow up with them later on with gratitude and information about the next major updates regarding your game and your studio.

Most importantly, don't be afraid to speak up. Reach out to people. Talk about your game boldly. Don't worry about your game being a small or insignificant one. Speak highly of it, and show that you believe in it. When someone is streaming your game, join the stream. Stay visible. Be original. Your confidence will help your game.

Marketing Through an Agency

There is another way to get your marketing done. You can do it through a marketing/PR agency. However, you must not jump into any deal without doing a proper check on the agency. You should always thoroughly go through their case studies. The first thing we check is their outcome, so we tend to study their performance with previous clients. That's okay, but that shouldn't be the only judgment criteria. You should see their outlining of the problem, approach to the solution, their learning from the campaigns, and most of all, validate the numbers if you can. Every agency wants to showcase themselves in their portfolio and, in the process, may end up exaggerating too much. That can be hurtful for you. Also, be clear about your communication. Be clear about all the terms and conditions, and be aware of any hidden clauses that you don't understand. You might end up spending a lot of extra money on them later on. Lastly, do check on their failure cases. Every agency will showcase its success cases in a pretty visible manner. Apart from validating them, you should find out which games they failed and ask them the reasons. Look at their recent activities. How did the last few clients do? If those games didn't sell well or the social media responses were bad, there's a chance that the agency is unable to recreate the success they had on previous titles. Don't just look into their media list, get impressed by it, and sign the deal. If their recent games are

not doing well, that means they might just be spamming their media list without taking custom care of their new games. This can be disastrous for you as a client.

Pricing

It's totally your call if you want to give away your game for free or as the most expensive one. We won't judge that. But please make sure you read about the case studies regarding free/paid/freemium models. Also, why try to sell a game for a million dollars when you can sell a million copies for one dollar? There's another concept in the market which states that more people will get your game if it's cheap. We see a lot of developers making their game free by default and relying on ad revenue. This is a proven method, yes. You can make significant money out of ads if you have many users. But games of certain types like hyper casuals and card games usually do better in terms of ad revenues. Also, the geography of your user base is important. In western countries, you usually get better ad revenues than in Asian regions. But only focusing on ads should not be the default option here. Sometimes people end up doing weird ad placements that only irritate players. Also, some games are better off remaining paid. You can always go for a mix. For example, you can ask for some payment if someone wants an ad-free version. Whatever you do, please don't do anything without analyzing your own game and what its revenue matrix should be. Some games are better at being a bit pricey, whereas you need to know where ads are relevant and where you can incorporate them. We also discourage flooding the user with ads. It might earn you revenues to some extent. But unless it's highly addictive, people will move away from your game just too soon. Every game has its own suitable price matrix. It's like a unique sweet spot. You've just got to find it.

Right now, blockbuster titles usually cost around 60 USD. This is probably the highest in the market in terms of mainstream games. In terms

of other games, it largely depends on the market trend and the platform. If you are developing an action game for mobile, you have to check the standard market prices for similar ones on the App Store and the Play store. Compare the games with yours and then decide. The game length and quality are the next adjudicating criteria. The lifecycle of the game is also important. How long will the gamers be able to play it? Will it get repetitive? What is the standard price for a game of this type and length in the market right now? In terms of quality, what's the comparison between those games and yours? These are the questions you need to ask yourself. Also, please study the sale policies of different platforms. We see platforms like Steam, PS Plus, and Nintendo giving offers on games during Black Friday, Christmas, etc. You can go ahead and read the policies regarding those sales offers. Also, do keep in mind that you won't be able to sell your game at the original price forever. It can be a good marketing policy at times if you want to grab new users while offering discounted prices for a campaign. It will depend largely on your plan regarding the whole lifecycle of the game. If you charge too little in the beginning, it limits your opportunities to provide good offers later on.

Go ahead and set your dimensions about The Great Filter. We gave you some heads-up regarding the options on your journey and showed you the doors to unlock. The possibilities are limitless, but the question remains how far you will be able to progress. In the end, will you be able to pass the Great Filter? Only time will tell. But we wish you a fascinating journey towards discovery and the loop that awaits! The core loop and some Greek myths.

CHAPTER 15

Game Over - The Myth of Sisyphus

The human mind expects something meaningful and fascinating at the end of any journey. We love to believe that there's a reward in the end. We romanticize the light at the end of a tunnel. We love happy endings. You have been with us throughout this journey and finally, we are at the epilogue. We won't make you upset. We will talk about the thing you want most: success. You will discover the possible paths you can take to fulfill your journey and the perils that come with them. We will also discuss what happens after the game is over and the process of restarting this journey of yours. This chapter will contain real-life cases regarding what might happen and what you should be prepared for. We will live in real life while taking inspiration from a Greek myth, as promised. In terms of topics, this chapter will discuss the following:

- The nature of "game"
- The importance of money in the process
- Ways to acquire funding
- Career cautions in the industry
- Hacks of team formation and management
- The business aspect of games
- Success in game development

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6_15

The Nature of "Game"

What do you consider a game to be? We believe video games to be one of the most significant art forms. Now, what makes art? What are the characteristics that are present in an art form? Let's say there are at least three traits that are present in any art form, if not more. They are the following:

- There's no concrete formula to make art.
- It's expensive.
- It can break your heart.

There's No Golden Apple

People like to think that there is some magical formula that lets people attain success in games. "There exists a golden apple. I just gotta find it and after that it's heaven!" they keep telling themselves. First of all, the golden apple is a myth. Secondly, even for the sake of argument, if we agree that there's one, what's the pathway towards attaining it? If we asked you how to make a painting that lasts through the ages, what would you do? Unlike potions in Harry Potter, where you can mix different ingredients to make a working liquid, art has no such formula/mapping.

Art is something that directly comes out of the human soul. Every person has a unique experience in life. They feel differently. So, each of us can produce our own output. That's why you can't just tell someone that this is exactly how you do it. You can share your stories of success and failure. You can tell someone which practices are good and which are bad, but you can never give someone a map. All of us have a custom map of our own when it comes down to art. You can take inspiration from the Mona Lisa, but you can never make another Mona Lisa-like painting. You can only create your own. That's why we can't tell you exactly how you can make a game. We can only show you different ways that we have seen working and many wrong paths you should avoid.

In our experience, we have seen that most cases of game development don't end in success. What we can do is learn from these cases so that we can avoid most of the known malpractices that have sabotaged our predecessors. We can tell you how to use a tool and how to navigate your way around the pitfalls people usually fall prey to. But you need to make your own custom journey into this universe.

Costly to Make, Costly to Get

Can you remember the old joke that you or your friends used to make regarding an expensive painting? "Oh, what fools would buy a mere painting with such high prices?" you would laugh with your friends. Maybe you were also making fun of the people who go to the opera after buying such high-priced tickets. Back then, perhaps it didn't make sense to you why paintings or other art forms were so costly. Well, now that you have grown up, maybe you understand. Art is always precious. It has this personal touch of the artist attached to it, making it a premium thing. You have to make a lot of sacrifices usually if you want to make great art. Sometimes these costs go beyond materialistic things and incur your undisputed spiritual devotion as well. Making good games can get really expensive regardless of the size or scale of the games.

It Can Break Your Heart and Also Break You

Art often can make or break a person. It involves a person's passion, memories, and philosophy. Those of us who are in the business of making games can get heavily invested in them. We dream about them all day long and try to work on them whenever we have some time. So, it becomes essential for us to see our sketches come true. Failure to produce the game we have in our heads can have a detrimental effect on our minds.

On top of that, sometimes we crave acceptance. We want people to love what we have created. If that doesn't happen, we get hurt. If you are in the business of creating art, you must always be ready to get hurt. Yes, we can tell you to be professional, and you can be, of course! But still, it will hurt as failure looms large. A game can involve way too much love and emotion during its birth in the minds of the makers. Even if you are a solid professional, still be ready for heartbreaks.

The professionals (artists, developers, directors) from Naughty Dog received death threats after the release of *The Last of Us 2*. The Internet is still filled with hate speech regarding the sequel to one of the best games ever made. Part 2 was released in 2020, and it bagged the Game of the Year title along with six other major awards in the Game Awards 2020. It has a Metascore of 93 in Metacritic, but the user score is only 5.7. This shows that if you do everything well, still you can face extreme opinions in this industry. The *Last of Us 2* didn't have any issues regarding the game quality. Many people hated it just for the plot.

There are cases where there are quality issues. The community can be pretty harsh on you if that happens. *Cyberpunk 2077* was released after eight years of waiting. Due to the massive hype, it scored record numbers in sales on the first day and in preorders. But the gamers found quality issues on Playstation 4, and the backlash was sharp. Luckily, CD Projekt Red was prompt in its response; it refunded people who bought the game on PS4. Eventually, the company had to pull the game off the Playstation Store. Founders of the company lost more than USD 1 Billion of wealth due to the whole scenario.

Imagine the pain of the hundreds of people behind the game. The gaming community had been waiting for this, making it one of the most anticipated games ever. Now the Internet is filled with memes regarding the console screenshots of the game. Even though the game is great in the PC experience, it wasn't enough to calm down the angry mob. If you are in this industry and face scenarios like this, the options are to develop a thick skin or avoid social media. People have great expectations. The bigger you get as a studio/ brand, the greater the burden will be.

Money Matters

It may feel like it requires small money to make a game. Well, that's not entirely false, to be honest. You can make small tiny hyper casuals, some puzzle games, and a few other short ones on a very small budget. But to make a successful/sustainable game, you need more than just a few thousand dollars. Initially, you may feel like you can roll out a game on your own with some dedicated effort for some time, or you can ask for some assistance from some people in their free time. Let's say you deploy it successfully. Then what? We discussed the process and the issues you might find while making your game a success in the previous chapters. Reading them, what is your impression? Do you think you have enough funds to pull it off? Well, how much are enough funds? Sadly, there's no exact answer to this question.

The game industry only tells you the stories of success, but what about the stories of countless failures? Many indie developers who put their life's savings into their dream project, developed a game, and eventually lost all of it without seeing success. Some people leave their day job and start on their own with a few months of funds. Now, leaving your day job is a big step, and congratulations to you if you have taken it. But remember, many of those who quit their regular job and started making games on their own couldn't make it sustainable. After a few months, they were at a loss, because their savings had run out and they needed to buy new development works, and pay marketing expenditures. There are many devs who roll out their game successfully but it doesn't pay them enough to afford health insurance. They can barely maintain their rent and three meals a day. Try to go through these stories as well. They can happen to you.

The opposite can also happen. Your first game can be a hit, and you can become an instant phenomenon as well (that's highly unlikely, though; the best-case scenario is some success after at least a few tries). It would be best if you kept both cases in mind while you are diving.

Another common mistake is to only think about the development budget. The game marketing budget keeps growing as your game keeps doing better. If you can't afford it, then the game will stumble. Today most of the popular platforms have become money-driven. The more you pay for ads, the more visibility you will have. Even if you make a great game, you still have to market it, most likely. Even if it is only until you can get yourself a publishing deal, you have to invest in the business side of the game. There's no way around it. So, what are possible funding sources?

Finding Funding

Taking investors is a usual option, but you need a good and convincing portfolio with a clear vision to convince people to invest in you. In terms of investors, we can divide them into two major categories: venture capital (VC) firms and angel investors. Venture capital firms are large organizations with deep pockets. They usually invest in ventures that are already big or making quite some impressions out there. They also have a very structured funding process. These are financial organizations. They only invest in places that financially make sense to them. They are not driven by emotion. Also, they don't invest in small numbers. It's either big or zero. So, if you are just starting out, VC firms are not the more likely option among the available ones.

The other option is angel investors, who are usually more lenient towards the process and funding range. An angel investor is a person who invests in you based on their own judgment. The ticket size differs from one investor to another. Ticket size denotes the range of money an investor typically invests in one venture. You should first figure out what amount you need. Then you can match that amount with the ticket size of the investors you have your eyes on. Also, it's good to have someone who has some portfolio in investing in this industry already. But if you are only starting out, it's going to be more challenging to get the best people

on board as investors. If you can't convince them regarding the ideas/plans you have for your game, you must at least convince them about your capability. Investors tend to invest in people rather than a venture, in many cases.

If you are having a hard time convincing typical and professional investors, you can opt for someone in your close circle, within friends and family first. They are more likely to have faith in you. They might have smaller money to inject, but if you think it will keep you afloat until you get ready to hit a home run, or at least secure the next level of funding, then this can be a viable option. But whatever you do, please don't do injustice to the faith of the investors.

When you take someone as an investor, they want to see their money bearing fruit. Often we see people overcommitting to investors to get them in. This is dangerous. You might be desperate for funds, but overcommitting to investors is like a potential suicide. They can hurt you real bad if you fail miserably to live up to your promises. The best way is to be absolutely clear about the expectations from both ends. Also, please make sure the cash flow is happening on time. This is of utmost importance. There's no point in having imaginary money in the bank when you can't pay your bills on time. The terms and conditions of fund disbursement should be clear, and they should be maintained properly. Always make sure to have more money in your bank than you need during the running month.

While we don't recommend taking significant money from investors who don't understand games, the better approach is to apply for some institutional investments. For example, `www.indie-fund.com` can be worth a shot. They have funded more than 50 games already.

If you don't find investors, you can aim for the art-based fundings of different governments or other organizations. Usually, the organizations provide funds based on their requirements, and you have to follow their terms and conditions. It might take some time to learn about all the opportunities and apply to all of them because they might ask

you to provide a lot of documents. Also, you can aim for grants like www.ukgamesfund.com. There's a platform named ko-fi, which is a crowdsourcing grant. Unlike any other crowdsourcing platform, it doesn't take any cut from the grant you raise. But what are the regular crowdfunding options?

There are a number of options if you want to crowdsource the required funding. Kickstarter, Indiegogo, fig, and Patreon top the list. The terms and conditions of Indiegogo and Kickstarter have many similarities, whereas the other two are a bit different. Some platforms will only release funds if your crowdsourcing pledge target is met. It is an all-or-nothing policy. In some cases, you can find options where you get whatever funding has been pledged. Make sure to read all the terms and conditions before you jump in. There are many prominent examples of crowdsourcing for game development. But do remember that you need to stay visible in front of people who might be potential donors. So don't make the mistake of sitting and waiting. Make and run campaigns. Reach out to people. Tell them about your game!

But we must warn you. These days, a successful crowdsourcing campaign has also become really difficult. People tend to give money to those they trust. Having a brand that people already trust, be it an organization or a person, can help a lot in making a crowdsourcing campaign successful. If you don't have either one of them, you have to spend on marketing, which requires money. To have your best shot at a successful crowdsourcing campaign from your end, please try to make the demo as good as possible, talk about your game in a passionate way, and give regular updates to the community that is supporting you.

Asking people for money is a special skill. If you are not born with it, you can learn it. But the journey isn't smooth. People get disheartened after rejections on this front. Hold your ground. You need the money. If you are looking to get better at it, go ahead and learn how to do so. This can be one of your future scopes of self-improvement after finishing this book.

In any case, if you don't want to toil to get money from someone else, you can always self-finance. But, as we have warned you, please be very careful when planning this, because you don't know the exact amount you will need. It can be drastically different than your initial plans. You can maintain a stable cash flow by doing some other business and then invest from the profits you make there. If you don't have a business to back your game up, you can take some other job. In this case, that other job has to pay well so that you have just enough money to invest in your game. But it's really challenging because doing a full-time day job that pays well will require a lot of your attention, merit, energy, and time, leaving you a bit short-handed on the possible efforts you can devote to your game. Many people fund their own games by doing other jobs/contractual work on games. This has some advantages over the other path where you are doing jobs from other domains to sustain yourself. Well, if you don't want the hassle of making your own venture, you can always take the option of working full time in other gaming companies to satisfy your hunger for making games. How about that path? Let's talk about that now.

Surviving Artists

Many people want to be a part of the giant names in the game industry. In most cases, this originates from the core emotion of playing their titles. "The game is so good… the makers are so talented… what if I could work in this company?" This is a common dream of many who are about to begin their journey in game development. This is romantic. Sometimes, we see dreams like this come true and have a happy ending. At the same time, like any other romance, this can have some tragic outcomes. There have been many issues in the gaming industry for some time. Probably the most common complaint is about crunch. To meet the expectation of the market and gamers, the game developers have to pull off insane amounts of work hours to see this through.

This is a heavily debated topic. Look into it before you fantasize about any particular company. There are people who enjoy the crunch. If you fall in that category, still I'd ask you to consider the detrimental effects a crunch can have on your physical and mental health. We see many game developers living a miserable life as they lose their jobs frequently (owing to the fact that companies tend to fire their team or at least a portion of it after the discontinuation of a game) and have to earn a living by freelancing. They lack stability in life. If you look closely, you can find many surviving artists in the game industry. They lead a difficult life. Read about their tales before you jump in.

The Lone Wolf

Maybe you want to be a one-dev army and conquer the game world all by yourself. That's okay. But make sure you know what you are up against. That's what this book has been all about. When you are the only indie working on a game, it all depends on you. If you get sick and can't work, the work of the game is paused. There's no one else to back you up. You have to take care of everything yourself. This, again, can have a hazardous impact on your mental health when things don't go well. If you have people in your life whom you can talk to, it can help you a lot. You get to keep the fruits to yourself, but there's a cost associated with it. What if things get difficult and you can't find a solution from your peers? George R. R. Martin says in his *Game of Thrones* that "when the snows fall and the white winds blow, the lone wolf dies, but the pack survives."

The Pack Survives?

There are lots of real-life stories to cherish regarding a few friends forming a game studio and achieving miracles with it. To be honest, this is more popular among people than any other path. There's a feeling of

forming a gang in it and this entices people like nothing else. When you have partners, you can afford to look after each other. You can watch each other's backs. If you're united in a singular vision, then a tiny team can achieve wonders. This is true, but it's easier said than done. The first problem is, of course, to find the proper partners. If you make a mistake while choosing partners initially, life can get very hard for you eventually. The first mistake is to take partners based on your needs. You are a programmer. You need to partner up with an artist. Of course, that's the way forward; you need each other. True, but your need to mitigate a particular skill gap should not cloud your judgment regarding what to look for and what to be aware of when choosing a business partner. Go ahead and get someone who compliments you as a partner, but not before you thoroughly think if they will be honest and loyal to you, your team, and the cause. Having a team member who has ill intentions is worse than having no teammates because they can destroy all of your efforts someday. Please be aware while choosing partners.

But don't worry too much. Even with your precautions, you are most likely to make a mistake here. Most people do. Some learn from the mistake and bounce back, but others don't. If you make a mess, don't get upset. Instead, think of the solution and learn from the experience. Try not to make the same mistake again.

Well, even if you are lucky enough to have nice teammates (one of us did in Battery Low Interactive), there's the huge task of maintaining the relationship with the partners and keeping your chemistry okay. You might have ups and downs. That is absolutely okay, as long as you don't go below the threshold. As long as you are honest with each other, you have the biggest problem solved. There can be performance issues as well. One teammate can underperform at times. You need to do expectation management very well. Team morale is a tough thing to handle while you have yet to hit the first success. Things can go wrong, but no matter what, you can't let frustration fester in your bonding. Also, while you are making a game, it's always recommended to have a team where all of the

teammates enjoy and play similar kinds of games. If there's a massive mismatch, you will lose a big advantage. Also, clarity is essential. Try to make policies before things happen, not after issues arise. For example, what happens if somebody gets an offer from the best game studio in town? What happens if you choose to quit game development? There should be policies in place. Feeling a bit cold? Maybe the art vibe is fading a bit. You're right. We are into business, almost.

From Art to Business

Since you are about to work with your own studio, you are starting your journey as a business entity. What you are producing is an art, but the organization is a business organization as long as it deals with money. So you need to learn how to run a business. This comes with a good deal of time investment. Make sure to have someone manage the business part of the work. If they are experienced, that's better. If not, use someone who has experience in doing similar work. One of us cofounded Battery Low and sacrificed his career as a game programmer because he was the best fit to take care of the business aspects. It was heartbreaking for him but necessary for the company. You need to learn about taxes and other formalities that you need to do as a company. You need to be good with the accounting books, or you must find someone who's good at it. Make sure that every transaction is logged adequately with evidence. This will build up a trustable culture in your company and help you in the future when you raise investments and face tax inquiries. Don't mix up your personal accounts with your business ones. If possible, use separate credit cards. As a business owner, it's better for you to know about the basic business rules and practices. Don't just ignore them, thinking that all you gotta do is focus on your art. When you are working on a team, you have to make sure that it doesn't fall apart. You must play your part. Ignorance brings problems. Dive deeper into these issues and learn more as you go.

Following Paths (Lack of Originality)

The good thing about the Internet is that all of us can know everything. The bad thing about the Internet is that all of us can know everything. There are many success stories floating around in games. You can't just copy and paste their success. It doesn't work this way. Just because somebody made *Angry Bird* and became famous, you shouldn't try to replicate this immediately. The stores are flooded with copy games. They don't have much originality. That's why they don't make it to the top. Usually, you can only go so far with copied concepts.

Don't be afraid to use your original and authentic concept. Short-term gimmicks will only take you so far. Stop selecting a trendy game and implementing some gimmicks on it to release a new game like that one. This is a bad and boring practice. People want the original you, remember? Art is unique. Make your own game. Do take inspiration from others, but make your own. But yes, don't forget the basics of play. In this book, we touched on the basics of making a good game in the chapters related to game design. If you want to study more on the basics of play, the book *Rules of Play* by Eric Zimmerman and Katie Salen is a great resource. Lastly, part of the job is to convey your message and effort to the audience. While creating your unique piece, don't forget to make it relatable and understandable.

The Winner Takes All

The game industry worships winners. It takes care of them and showcases them over and over again as their success case. If you're successful, you get the money, the fame, everything. If you are not, you get nothing. It is also biased towards the wealthy ones. Let's consider a case. One indie developer studio has a good dev budget, along with the latest devices to work on, and of course, a healthy reserve to spend on marketing if needed.

They can afford to hire top-level industry veterans as well. They will have a better chance of making a game popular than you, of course. There are many talks and events all around the world in the game developer community. Successful people go there to talk about their experiences. The failed developers usually don't get an invitation to share their stories. This industry believes in success towards the successful. When you have a hit game, people look forward to your next game. This helps you intuitively in the outreach. But it also comes with an added burden: expectation. But with any success, the expectation has to come, in all domains. If you are not successful yet, the climb might just be uphill. Once you are there, things will be much easier for you the next time. It doesn't necessarily guarantee you success again but will make the journey much smoother next time. But we didn't answer the most important question yet. What is success anyway? To answer that, we need to seek salvation in some ancient Greek mythology.

The Core Loop and Sisyphus

Sisyphus was the king of Corinth. After his death, like any other mortal, he was taken to the underworld. But he didn't stay dead for long. He tricked Death and chained him there in his place. This caused chaos. Since Death was stuck, people stopped dying. The gods became furious with him. They took drastic action, solved the issue, and got the life of Sisyphus again, only to be deceived by him once again. He tricked the gods one more time and escaped death. Having tricked the gods twice, he drew the attention of the king of the gods, Zeus. This time, after his death, Zeus gave him a unique task. Sisyphus had to push an immense boulder up a mountain. Upon reaching the top, the boulder would roll down again, leaving Sisyphus to start over. He was doomed to push that rock tirelessly for eternity. This is his loop and legend. This has been the topic of discussion in many segments, and many philosophers have depicted their own version of understanding regarding this infinite labor.

Some describe tasks that are both laborious and futile as **Sisyphean.** At times, we can feel the same. The life of a game developer might get caught up in the cycle of making games and not reaching targets. No matter how hard you try, as you seem to reach the ending (in this case, success), you always come short and start over again. Suppose you can break the loop, then good for you. German author Manfred Kopfer suggested a way to break it. According to him, Sisyphus breaks a stone off the mountain every time he reaches the top and carries it to the bottom. If he keeps doing this, someday, the mountain would be leveled, and his burden would end. Thus, Sisyphus' punishment can be interpreted as a chance or test for Sisyphus to prove his worthiness. If he can really move a mountain, he can ascend to the same level as the gods or earn the capabilities a god has. If you keep learning from your endeavors in game development and failed cases, someday you will be able to make a game that everyone loves. Someday you will get out of the loop.

But… What if Kopfer's case doesn't work for you? What if you get stuck in this core loop for eternity?

There's salvation for you in Albert Camus' book called *The Myth of Sisyphus*. Camus imagines a scenario where Sisyphus is smiling while pushing the rock. He thinks that Sisyphus has embraced his situation and lives in the present, rather than dwelling in the past or future. He refuses to surrender to his fate by being persistent. He is unstoppable as he keeps pursuing in the face of imminent failure every single time. He makes it his sole purpose in life and gets defined by it. It is to be noted that in the past, he had done many evil deeds, but he will always be remembered most for his infinite labor. Without his relentless effort, nobody would discuss him like they do now.

Imagine Sisyphus' face the moment before he starts rolling the boulder once again, after just seeing his effort ending up in vain. He knows that once he starts, the cycle will repeat itself once again. He can't hope, but he won't give up either. He simply acknowledges the truth and, in this way,

conquers it. He understands the absurdity of his situation and accepts it with contentment in his heart. Camus concludes that "all is well." He states that "one must imagine Sisyphus happy."

It's not the conclusion or outcome that denotes your success or failure. If you are smiling on your journey, then you're good, my friend. If you are enjoying it, that's enough. Most people don't enjoy what they do in their lives. If you do, then you are already successful! We are driven by passion and our love for games. What else would you rather do than spend time with games? The destination matters less. What's more important is the journey. It's not easy, the destination might be impossible and irrational to reach, but the experience can be fun.

In the end, that's all that matters.

Index

M. Fahme and T. H. Khan, *How to Make a Game*,
https://doi.org/10.1007/978-1-4842-6917-6

D

E

F

G

H

I

J, K

L

M

N, O

P, Q

R

S

T

U

V, W, X

Y, Z