ProgramKitsForEveryone

Kit 1: Design and Build a Platformer Game

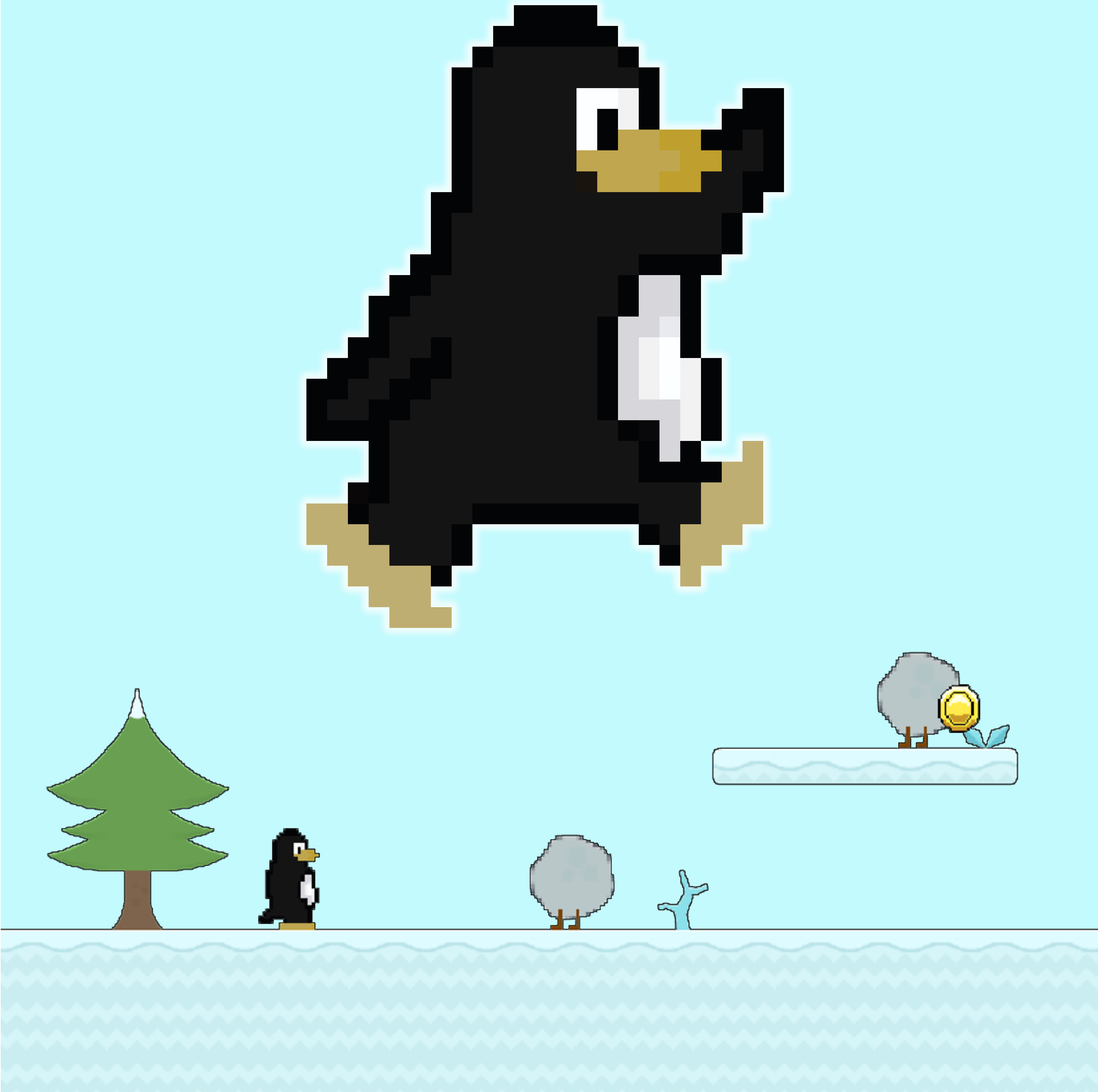


Table of contents

Introduction 3
The idea behind this project 3
What this project is 3
What this project is not 4
Why this project was created 4
Chapter One 5
What tutorials get wrong 5
How do you follow these 5
nebulous “coding styles”? 5
The importance of writing 6
intelligible code 6
Guidelines for writing good code 7
The importance of reading code 8
Conclusion for chapter one 8
Bad code example 9
Chapter Two 10
Why I chose the Godot game engine for the first kit 10
After choosing a game engine 11
Making the art 12
Map and level design 13
Chapter Three 14
The game loop 14
Programming the game 14
How the example code is structured 17
The most important part of a platformer 17
Move_and_collide() vs. move_and_slide() 18
Jumping and gravity 19
The animations 19
The enemies 20
Coins, collectibles and the score 20
The first level 21
The camera 22
Extension challenges 23
After all has been said and done... what now? 23
Credits 24
Curated list of Godot learning resources that I have used 24

Introduction

The idea behind this project

The intention of this project is to be a way to teach how to design and build larger applications and programs. In programming, many beginners lack the experience of creating medium and large scale programs. In tutorials what is most often taught is how to do something, and not why to do something. The why is very important as code that might function on its own, isolated, might not function in a massive program with a large number of things happening simultaneously.

For example, in beginner programs I've noticed the use of sleep and delay functions. These functions work well when creating a program with a single purpose, but in a large program, they will lock up the entire program. That kind of learning is dangerous as beginners can form bad habits. I've seen it personally, working on a robotics competition. My teammate implemented a delay function which locked up my time-critical code. Debugging that was a mess!

What this project is

This project is the culmination of my effort to teach other people responsible programming, both in the context of my school and beyond. This part of the project is called a "programming kit." It contains a program which serves as the foundation of a bigger project. I will explain how the program works on a higher level, and it is up to the reader to either follow "extension exercises" or do whatever they want. I went through many revisions of this project, but I decided that this last revision was the best because it was what I would have wanted when I was starting programming.

In the beginning, for me, despite knowing all the C++ tricks needed for competitive programming, I had never created anything bigger than 100 lines of code. When I finally wanted to create my first program I struggled immensely. I had been so invested in this specific field of programming where speed of writing and execution of code were more important than quality and readability, that when it came time to create something I just stayed with my habits. That resulted in hundreds of lines of code with variables named "a1", or "num1." When I had to leave my project for a few days, when I came back I couldn't understand anything. Looking at the code now, even if I named my variables right, the code was fundamentally flawed as it was all in the main function, uncommented and following no code style except for being written as fast as possible.

Though I left competitive programming a year ago, together with the tutorials I had watched I had to unlearn my terrible habits. Seeing how most of the programmer friends I know also participated first in competitive programming, I fear that our skill might even be a disadvantage when making actual software!

In the end, this project aims to teach beginners how to not fall into the trap I fell in and focus instead on the real world, and how programming works in the real world. Had I been taught to read code before I wrote it, I wouldn't have had to unlearn bad habits for a year. If you go to my Github, you'll still see some of those bad habits present in the little code I have available to the public! I hope that everyone who reads this at least gets some tips out of it or a reminder on how important following design standards and principles is when designing and creating software.

What this project is not

This project will not teach you the tools used to create the program, namely the Godot game engine in this case. I chose Godot for this programming kit as it is very easy to learn and forces its users to follow many of the design philosophies I will talk about later. You will have to learn Godot on your own, which is a good thing in my opinion! I will link some tutorials that really helped me learn the Godot game engine, and you can see that it's a very simple, but deep, game engine. This project will also not teach you programming. It can serve as a supplement when learning programming, to learn some of the standards and hidden rules of programming as you go along, but it will not teach you programming by itself. Had I had to teach programming, this would have been way longer and, in the end, redundant as I would be rehashing things people have said before me for decades.

Why this project was created

As I touched on before, this project had been something I had wanted to do for a long time. Now with both the support of the GNOME project and my school, I can finally justify doing such a large project. I have always loved teaching and helping people, and this is enabling me to do it on a global scale! Thanks to both the GNOME project and all of my teachers who helped me along the way. The open source community is especially a big motivator for me. I even started learning about coding standards and design principles a year ago because I wanted to start seriously contributing to open source projects. Currently almost all of the software I use is not only open source, but sometimes software I've contributed to myself. That's why I encourage anyone who uses this or any other programming kit to extend their software and make it open source. There are no downsides to open sourcing your personal projects!

Books like this one partly inspired the creation of this project. I admire the DIY culture of the 80s and how it was necessary for everyone to be somewhat computer literate to use their computers. The Commodore 64, in my opinion, is the paragon of retro computers. Image attribution - Lemon Retro Store

Chapter One

What tutorials get wrong

I believe that online tutorials, both written and in video form, are one of the best things to happen to programming, ever. Coupled with the fact that more and more tools are becoming free and open source, it has never been easier to learn programming. Whenever I'm learning a new programming language or environment, there always comes a time when I need a tutorial to do something specific.

But due to the limited time tutorials have, coupled with the fact that they are usually beginner oriented, they neglect talking about the implications of what they are teaching in the context of a large program. Like in the example with the delay stated previously, my teammate most likely learned that from an online tutorial, as the documentation we were provided was of very poor quality. If our robot code fell apart due to a delay, even though it was a few hundred lines of code and done by only 2 people, imagine if he had done that in a bigger project.

It's much easier to learn something than to unlearn it, especially in such a field where we rely on habits a ton. The best way to avoid bad habits is to follow the programming language's style guide and to also read others' code, but we will get to that a bit later.

Essentially, what tutorials miss is the fact that often you are not the only person working on a project. Writing all of your code in the main function, uncommented and following your personal style is terrible. Yet, many times when you look up "how to do x", the solution is exactly that, a chunk of uncommented code in a main function following the author's coding style. Having to work with classmates for a programming project in school showed me exactly how people who learn from those tutorials write code. Imagine a project where 3 people have separate coding styles and skill levels. Like with the robot previously, it obviously broke quickly.

How do you follow these nebulous "coding styles"?

What beginners don't usually learn is the fact that practically every programming language, game engine and framework has its own coding style. Sometimes there are multiple standards, but even those share a lot in common. You can find out the coding standards by just looking up "language/game engine/framework style guide" and looking on the official web site first, and if there isn't any, looking on community sites. If nothing turns up, it's never bad to ask someone from the community!

The Godot game engine we will be using has a very good page on the Godot style guide. Everything is explained, from the order of writing variables and functions to the way comments and variables should be named is written down with examples for everything. It clearly states that, for example, snake_case should be used for most types of values, and also states when to not use snake_case. It should be said, as it is stated in the code style itself, these are not hard rules.

Nobody will punish you for using your own code style, but having different code styles on a single project does not benefit anyone – you'll just get confused when reading someone else's code. For example, in the Godot code style it is stated that variables should be written in all lowercase snake_case, while constants should be in uppercase.

With that in mind, if I was working with a team on a project and someone named a variable in all uppercase, I would recognize it as a constant, and could make a simple mistake while treating it as a constant. Because in programming we rely on habits a lot, I would just assume it's a constant and treat it as such.

Following a coding style not only helps if you are collaborating with multiple people, but also when you leave a project and come back to it some time later. A consistent coding style means that your code is immediately readable to both you and anyone who collaborates with you. Being readable is one thing, but what about understanding what the code does?

The importance of writing intelligible code

When working on a larger project, the bigger the project is, the more time you will have to spend reading code. Depending on the project, that can either be a blessing or a curse. A project with good and intelligible source code can inspire and lead to the creation of many others that are based upon it. One of the most famous examples I know of would be the video game Doom.

Doom was made open source some time after its release, which created a flood of ports of Doom to other computers and video game consoles and also modifications of Doom. The same thing happened with the later game Quake which was based on Doom. The fact that these two games were open source and had good, intelligible code was the reason the video game genre of first person shooter exists today. Some of the most famous first person shooters that are still played to this day are direct descendants of Quake, like Half Life, Counter Strike and Team Fortress.

If they didn't open source the code, or if the code was written in uncommented assembly (as some games at the time still were), the genre of the first person shooter would most likely either be not as popular, or replaced by another. The fact that a few thousand lines of good, intelligible code can define the biggest entertainment medium in the world for decades should be enough proof of how important good code is.

Another thing to keep in mind is that bad code decreases productivity as a project grows. A badly designed function in the beginning can have massive ramifications later on. I personally saw that with one of my projects, where a small app I was developing had a rushed main function, which was impossible to rewrite as I had already implemented quick and dirty fixes to interface with the main function. In the end, all of the code was completely unusable and I had to rewrite the app from scratch. It took me longer to design and write the main function, but the time I spent writing all the other functions turned out to be much shorter. In the end, because I didn't have to implement "kludges[1]", the code ended up being much more elegant. All it took was just some more time to design and think about all of the interactions between different parts of the code, but more on that a bit later.

[1] a kludge is something, especially computer hardware or software, that has been put together from whatever is available, especially when it does not work very well - Cambridge defnition

Guidelines for writing good code

We talked about the importance of good code, but what is good code? Unfortunately there is no single list of good coding principles that exists, but there is a general consensus on what good code looks like. Many of the older programmers I know swear by the book Clean Code, and after reading it, I tend to agree. This part of the kit will essentially be a summary of the more important takeaways from Clean Code, but with my own opinions and experiences added. Just make sure to not consider these rules to be completely perfect and objective. Sometimes you need to break the rules to make something work.

What I consider to be the two most important rules of programming are brevity and structuring. For brevity, it is important to consider the Unix philosophy, which is what lead to the creation of much of the software we use today. Good code should follow the Unix philosophy, most importantly the first 2 rules:

1. Have your functions do one thing, and do it well
2. Write functions and programs that can be chained together

With that in mind, we can create some simple rules that you can follow when writing code, like:

1. Make it simple and direct so that anyone can skim it and understand (bit manipulation is an example of something that is hard to understand if just skimmed through)
2. Well named variables and functions that show meaning and intent (name your variable playerName and your function getPlayerName(), not pn and ret_pn(), and should also have verb names, like do_something())
3. Have short functions – around 10-20 lines is good enough
4. Make your code modular and portable
5. Use abstractions – they were invented for a reason!
6. Follow the rules, but know when to break them too (fastinvsqrt() is a function used in the video game Quake III which was notorious for exploiting undefined behavior, using undescriptive variable names and most famously the comment on the code about how not even the author understands what the function does, but because the function was very fast, it had to be used)

Comments are often regarded as the savior of bad code. It does not matter if the code makes no sense, when the comments explain everything, right? Well, comments are just a hack used to justify writing bad code. It is important to have comments explain the function of longer chunks of code, but commenting everything just means that the code is not good enough to be self explanatory. Redundant comments are especially unnecessary, and pointless comments just clutter up the screen and make it harder to focus on the actual code. In some cases comments are absolutely necessary, especially when writing in programming languages that are hard to understand by themselves, like assembly programming languages.

Generally, you get a feel of writing good code over time and as you gain more and more experience. Following rules and standards can only help you and the worst it can do is wasting a couple of extra milliseconds to type a longer more descriptive name, or taking some time to plan out how your functions should be structured. But writing code is not the only way to learn how to write good code.

The importance of reading code

Imagine a writer who does not read other books or a composer who does not listen to music. Their work would obviously reflect that in multiple ways, so artists are generally supposed to consume more art than they create. If programming is an art, then why is reading other people's code not emphasized in education? Many of the people I asked to help me with getting information for writing this, did not have a habit of reading other people's code. When working on a project, reading code should be more important than writing it, because your code will affect the whole program. Having a rough idea on how the entire program works is important.

Reading code helps to destroy bad habits both before and after they set in. This also relates in a way to following the style guide mentioned previously, but bad habits are formed when there is nobody to tell you that they are bad! For example, I still see some C tutorials and code bases that use the gets() function, which is a very dangerous function to be used in an actual program. If you read others' code you will see that gets() is not used anymore, and even most C compilers will tell you to not use gets(). If someone forms a habit of using gets() and continues to use it because there is nobody to tell them to stop, then their code is vulnerable to all kinds of easily exploitable buffer overflows. If there was no compiler to tell them to stop, they might continue using it. Luckily for us, humans are creatures that require social approval, whether that's by friends or compilers, and I believe that most people would listen to a compiler when it says that the function they are using is dangerous.

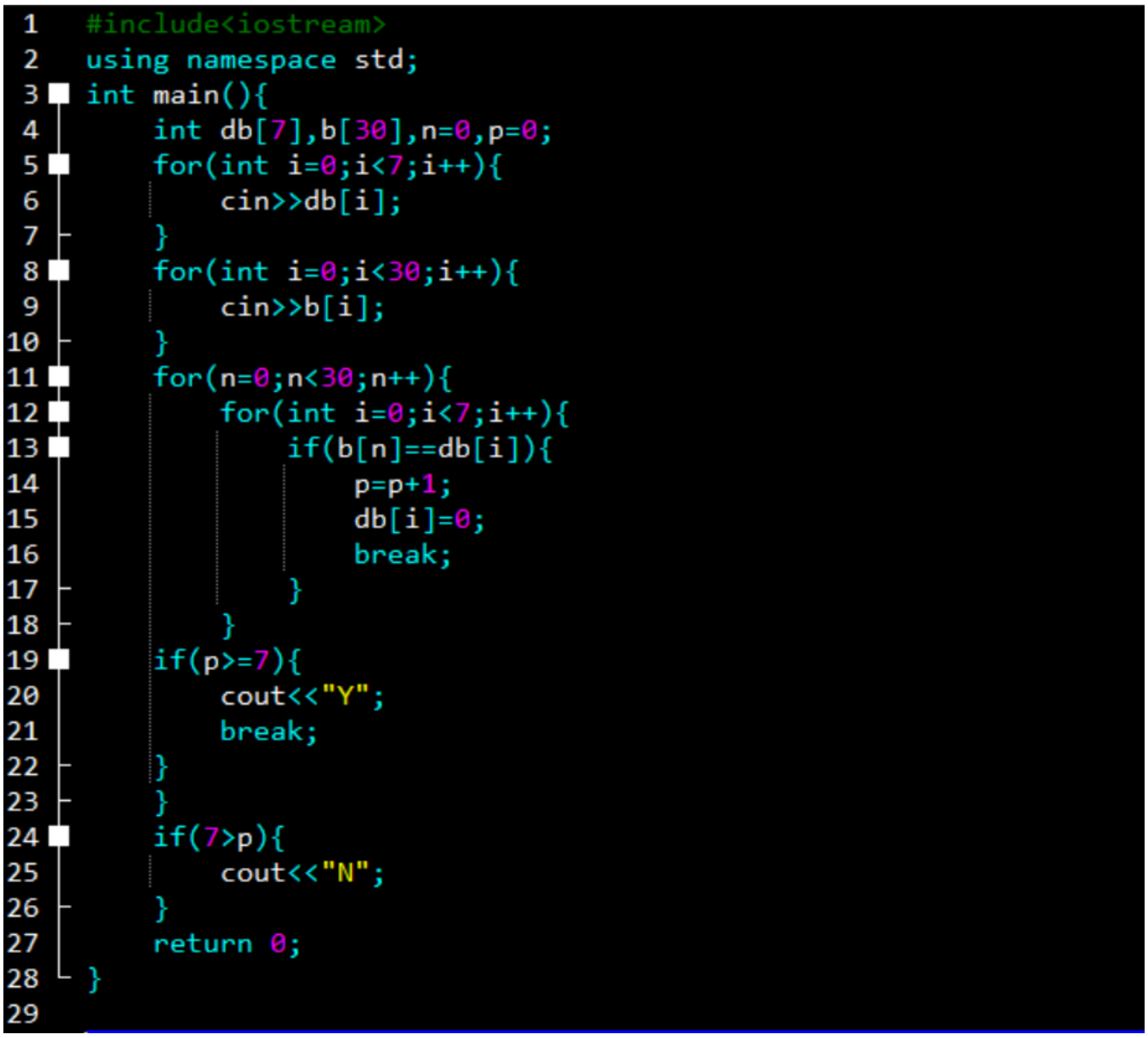
All this could be avoided by just reading code, but also comparing yours to someone else's. A good exercise I found is to take your program and compare it to someone else's that does the exact same function. Usually, you can see all kinds of unique approaches to the same problem, and oftentimes you will find out that someone's made your program better than you. Even better, sometimes you can combine multiple different approaches to solving a problem into one, to make your code even better.

The most important skill learned when reading code, like when reading a book, is being able to read other code even better. It is an exponential growth of knowledge which only benefits you more and more. You start noticing underlying patterns in how people write and structure things. Learning to read code is much like learning to read complex philosophical books. In the beginning, you will not understand most things. The fact that most people also disregard reading code as an important skill does not help you! But as you read more and more, you become better at both reading, but also writing. And when you work on a larger project, you will read a lot more code than you write.

Conclusion for chapter one

The importance of good, clean code cannot be understated! In the beginning it might seem silly to follow rules when writing code that only you or a small team is meant to read, but forming good habits is good for both you, your teammates and your projects. Personally, these past couple of months after preparing for writing this and reading some of the Godot engine, I started to appreciate programming as a legitimate art form. If programming is to be treated as an art, we must have some standards to follow. Sometimes rules have to be broken, but make sure you try to make your program as easy as it can be to read and understand. Your future self and everyone who collaborates with you will thank you for it.

Bad code example



```
#include<iostream>
using namespace std;
int main(){
    int db[7],b[30],n=0,p=0;
    for(int i=0;i<7;i++){
        cin>>db[i];
    }
    for(int i=0;i<30;i++){
        cin>>b[i];
    }
    for(n=0;n<30;n++){
        for(int i=0;i<7;i++){
            if(b[n]==db[i]){
                p=p+1;
                db[i]=0;
                break;
            }
        }
    if(p>=7){
        cout<<"Y";
        break;
    }
    }
    if(7>p){
        cout<<"N";
    }
    return 0;
}
```

An example of bad code from my competitive programming days. This particular piece of code is only three years old. You can see me breaking every single rule I talked about previously. In this case, that did not matter as the program was supposed to be written in a short time frame, so development speed was more important than everything else. After revisiting this code, nothing makes sense. What are db[7], b[30], n and p? What is happening here? There is some strange loop that checks every b with every db, and then adds one to p if they are the same, then the corresponding db is set to zero. Even for such a small program, I have no idea what is happening here. Listing all of the things that are wrong with this code would take one page. Can you spot all the bad practices here?

Also, I used to think that setting my IDE to this color scheme made me look cool. Ah, Dev C++, why did you let me do this?!

Chapter Two

Why I chose the Godot game engine for the first kit

I have followed and even slightly contributed to the Godot engine for a while. The game engine has grown to become a professional game engine with many parts of it that follow my philosophy of making software. The small team behind Godot engine have made a free and open source game engine that can rival the industry giants. But I believe that Godot's killer feature is the fact that it allows for the development of all kinds of programs, and on every platform. The Godot engine application is itself made in Godot engine, which shows the power of the game engine for not only video games, but GUI apps. Because of Godot's strong GUI features, I believe that many people can use it for building actual, high performance desktop apps as well as high quality games. It is essentially a platform that provides development of most kinds of desktop apps.

Godot is also highly extensible and can be interfaced with through any programming language, though that feature is not documented that well. I have used it to make desktop apps as well as games, and for both it works wonderfully. The area of desktop apps is currently dominated by Electron, which essentially means that Electron-based desktop applications are just browser windows. That brings in a ton of vulnerabilities and performance problems, and trades those off with development time. For both speed and performance of desktop apps, Godot beats Electron. I believe that if the Godot team markets the game engine to the GUI crowd, there is potential for the game engine to become more than that.

Godot is also the fastest growing game engine currently, but the community is still very friendly and helpful. The programming language Godot uses, GDScript is very beginner friendly, but also made specifically to suit the needs of Godot and programming it. GDScript is in a way a mix of the programming languages Python and C++, which makes it simple to read and understand, but because it was built to work in Godot, it has a very elegant way of interfacing with Godot. Anyone from a Python or C++ background can learn GDScript in a day. Though GDScript cannot be used outside of Godot, learning GDScript can help with learning the programming languages it is based on.

Finally, the way programs in Godot are made follows the philosophy of modularity I mentioned previously. Everything in Godot is organized in scenes, which are just groups of nodes. Nodes make up everything in Godot, and a node can be a sprite, a button, a collision area and everything else. It is not only possible, but encouraged to split your Godot games into many scenes, and then add them in when necessary. For example, if all levels are just scenes, you could add 2 levels together very easily, and nothing in the game would break. More importantly, this allows programmers to very easily reuse scenes in multiple projects. You could have the scene for your character be in multiple games that feature that character. When your games get more complicated, it is possible to just nest scenes in other scenes. That way, the whole game is modular. That also opens the door to modding, and mods to games sometimes turn into actual games themselves. Many of the most popular games today started off as mods, and many old games are kept alive only because of modders.

Godot is also very accommodating to all kinds of developers and platforms. Because of its open source and modularity, you can program in many programming languages – both the 3 languages supported by default (C++, C# and GDScript) and any programming languages that you can make it interface with. GDScript especially has a lot of Godot-specific abstractions and functions that can make code extremely short compared to equivalent game engines. All in all, the Godot game engine is an excellent choice for any developer to make any kind of game or desktop application.

After choosing a game engine

When making a game, you usually need to design the game. Most of the choices should be made at this stage of game development, such as which game engine and art style to use. Since we already chose the Godot game engine, it is time to make some of the other choices. At this stage, all the choices are easily reversible, since we are not building anything yet. I chose that this game will be a platformer, something like SuperTux/Super Mario.

Choosing the art style in the beginning is very important to crafting the identity of your game. Different art styles evoke different feelings, and it is important to have your art style follow the feelings you want to evoke with your game. Art can make or break a game. For this kit, I chose the art style to be pixel art since I want to evoke the feelings of friendliness and simplicity. Pixel art is a very common choice for developing games with a small team since it is easier to make compared to other art styles. Usually when developing a game without a dedicated artist, it is very hard to make art that is as good as what an artist can do. That is why being an artist is a legitimate job – it is hard to do and takes a ton of experience. Pixel art is comparatively simple and does not require you to reinvent the wheel. There are many tutorials and courses that can teach you pixel art. Since pixel art has been used for decades to make all kinds of games, it is easy to draw inspiration from what other people have done.

Pixel art also does not require any specialized equipment. Artists usually need expensive drawing tablets if they are doing digital art, and then they have to get a specialized program for drawing that usually costs a lot (Krita is an amazing exception to this!). For a solo game developer or small indie studio, getting a dedicated artist is a big investment and sometimes not feasible. So, when choosing an art style, think of the real world logistics as well as how your game looks. Motion captured real life actors might seem like a good idea, but that is completely infeasible to do for an indie studio or a solo developer. Sometimes the game we imagine has to look different from the game we make, but that should not discourage you from making games, art styles like pixel art have specifically made game development feasible for small teams!

I discussed pixel art, which is used in 2D games, but what about 3D? 3D games are often much more complicated to make than 2D games, and their art styles reflect that. In a 2D game you only see one side of a character or a sprite, but in 3D games you have to make characters visible from all sides. That is why most games made by small teams are 2D. 3D also requires software that is much harder to learn than 2D. In 2D games, you make your character and just put the sprite in the game, maybe animate it a bit and it works. Compared to that, 3D games require multiple steps to make a character. You have to have a mesh, geometry, rigging and more. Thankfully, the free and open source software Blender makes that easier than ever, and more and more games are starting to come out that use a 3D art style and are developed by a small team.

After choosing an art style and an overall mood of your game, it is important to choose a color palette. Colors also indicate the mood and tone of your game and are as important as your art style. To find a color palette, I usually look at other games with a similar feeling I want to evoke. Horror games usually use black and red, and all other colors are made darker to strengthen the feeling of fear in the game. On the other hand, platformers like the one in the kit and the ones it is based on are positive and happy. Super Mario's color palette is bright and contains a mix of a few basic colors used in all the games – yellow, red and green. In the Mario games there is a distinct lack of blue (except for the sky/background, but even that is light blue) compared to all the other colors, and that could be because blue is relatively rare in nature compared to yellow, red and green. Super Mario is famous for its cheerful and relaxing atmosphere, and the color palette is an important reason why! The main colors in SuperTux are light blue and white because of the snowy/icy theme of the game. SuperTux feels colder and more rigid, but because all the colors are light it retains an atmosphere of relaxation and never gets too serious.

The Tux I made!

Making the art

If you are part of a small team with a low budget (or, sometimes, no budget), then making art can be a bit more challenging. Luckily, with the rise of free and open source tools, you no longer need to pay an expensive subscription to even use drawing software. Without an artist, I would recommend taking some sort of free art course or tutorial and using pixel art for 2D. The application I will be using for this kit is Piskel. Piskel is a browser-based pixel art application with rudimentary tools, but for something as simple as this, it is more than enough. For more serious pixel art, I recommend Aseprite. Even though it is paid software, it is still open source and if you know how to compile it you can get it for free, though I recommend paying for it to support the small developer team. If there is a free and open source application you use all the time and appreciate the developers making it, you should consider helping them in some way, either through donations or contributing code and documentation.

Before I started drawing, I played SuperTux to get a feel of how the sprites work and how the animations should feel. When drawing inspiration for something, make sure to know the source material well. How the animation and movement feels is also very important to the art style. I knew that in SuperTux, there is a lot of sliding on ice, but I decided to make my version feel more like Super Mario because the movement is much easier in Super Mario. Sliding on ice is a mechanic that takes some time to master when combined with a platformer, and if done incorrectly it can be a source of frustration for new players.

I first drew the character and the enemies – Tux and a walking snowball. I decided to have only one enemy at first because multiple enemies would need a lot more coding and interaction, and I did not want to just copy all the enemies from either Super Mario or SuperTux. I took a screenshot of SuperTux and drew Tux them with the screenshot on one part of the screen and Piskel on the other. The snowball was based on a sprite from the tile set I used for this project, but more on that a bit later. I did not want them to look exactly the same, so I changed them up and drew them in my own style. It is important to get the characters and entities to look right because the player will be focusing on them for most of the play time.

At first I did not have animations, but the game seemed very rigid and gameplay did not feel as fluid as I had hoped, so I added animations. Animations in Godot are easy to do, but more on that later. The animations were just simple walking and jumping animations, for this type of game there is no need for anything more complicated than that. After doing the entities, it is time for some level design!

Map and level design

Because drawing a sprite set for a level that looked good would take a very long time, I decided to use a Kenney's set. The Kenney game studio creates lots of good looking assets that are free to use in games. Many indie games use them for that reason, and if I were seriously making an indie game, Kenney's assets would most likely be a part of my game.

To design the levels, I decided to first draw them on a piece of long horizontal paper. For platformers level design is important, as most of the difficulty of the game comes from doing the challenges each level provides. Essentially, in platformers, levels are just multiple puzzles connected with each other. Good level design should include multiple different types of puzzles, and make the transition between them as fluid as possible. In games like Super Mario, a level can have a theme all puzzles follow. For example, there are levels where all the puzzles and challenges are related to the bullet type of enemy. In the later levels, rotating obstacles are used a lot. You should think of a theme for each level to make every level unique. Every level should also have a gimmick, whether that is an enemy that flies above the player and throws something at them, or having a secret path that can lead you to the end of the level.

Level design should also introduce new concepts in a safe environment where the player can only learn and not be threatened by anything. The beginning of the original Super Mario is a perfect example of this. When you start, you walk and see the yellow blocks. If you hit the yellow blocks, you get a mushroom that makes you grow. In that space, there is nothing else happening except you and the mushroom. After that, you are shown your first enemy. The first enemy is very weak and is contained in an area where it cannot harm you easily. The first level continues like that until it teaches you all of the basics of the game. You do not need a tutorial – make the first level a tutorial!

Since I assume that most of the audience is familiar with platformer games, the first level starts off with action immediately. I focused on having more enemies and easier puzzles, but that is up to personal preference. I designed one level, but made it so that any other games can be designed very easily, but I will talk more about that later. The first level was designed to be simple and to act more as a playground for ideas for anyone who wants to extend the project later.

Chapter Three

The game loop

When making games, you need to have a goal that the game needs to achieve from the beginning. Whether that is making a fun and entertaining game, or a game that requires solving puzzles, or a game that tells a story. If the core of your game does not fulfill that goal, then all the polish you add will not make much of a difference. Polish is not as important as the core of the game! Say I wanted to make a story driven game. If the story does not evoke the feelings I am trying to evoke, then why even make a game around it? It is much easier to make the base of the game work well at first, because it is really hard to make changes to the core of the game afterwards without breaking something. Make sure the first prototype, the pre-alpha test, works well! Ask others about it, test it with friends, but make sure that the base mechanics are what you want them to be.

Even before starting to make your game, think of what the most fun part of the game will be. Is it the platforming, the collecting of coins or fighting enemies? Focus on the most fun part of the game, or whatever part you want to make prominent. Sometimes players will find other mechanics more entertaining than what you intended, but as long as the main part of the game is good, then your game should not have a problem. Having too many fun mechanics cannot hurt your game!

For example, the main point of the game Counter Strike is to shoot and outsmart opponents. Yet, there is a massive part of the community who make their own levels and use a movement bug called bunnyhopping to traverse them. These players found a bug that makes the player move faster when jumping in a specific pattern, and created platforming levels around it. Though that is not even remotely the point of the game as the developers intended it, it is still one of the defining parts of the game and a sizable chunk of the community participates in these maps. Counter Strike is unique in the fact that it actually has multiple of these bugs that have spawned sub-communities based on using a bug as a main feature of the game. Sometimes, while creating your game you can accidentally create a feature that is more entertaining than what you intended to be the main part of the game. That is fine, and you can use that as one of the main features of the game!

Programming the game

After designing the game and figuring how the art and main loop will look like, it is time to combine all that together with programming. Before starting to code, you should make a diagram of all the most important parts of the code. For example, if I were creating a Doom-style game in C/C++, here is what the diagram for all the subsystems and important parts would look like.

All of the subsystems for the most important parts have their own subsystems, and if I were to draw all of the functions, it would look like a tree with functions splitting into other functions. Based on the principles of good code we established a few pages ago, all of these functions should be 5-10 lines and should be split into their own functions. When done like that, it makes debugging much easier because you can put breakpoints and find out exactly which function is causing a problem. Also, when you divide your functions to make a tree structure, you can reuse large parts of the program in other projects. Godot makes this much easier and it is an intended feature of Godot to reuse scenes in multiple projects, just make sure to also bring the whole “branch” with it (all the dependencies).

When splitting a game into a tree structure like this, it also becomes easier to parallelize different branches, which in turn increases performance of the game. As your games get bigger and more complex, the need for parallelization increases. Especially in the modern world, when more and more processors are focusing on the number of CPU cores rather than raw power. If your game is not parallelized, you could only end up using a tiny fraction of the power of the CPU. With modern CPUs with 8 threads and more, you could end up using less than 10% of the processing power available. Design your code to be able to use as much of the CPU as you can!

Luckily, Godot already handles a lot of the parallelization by itself, using different CPU cores for different parts of your game. Still, remember to think about how to use the CPU as efficiently as you can. Modern computers have a ton of processing power, but it is up to you to use it.

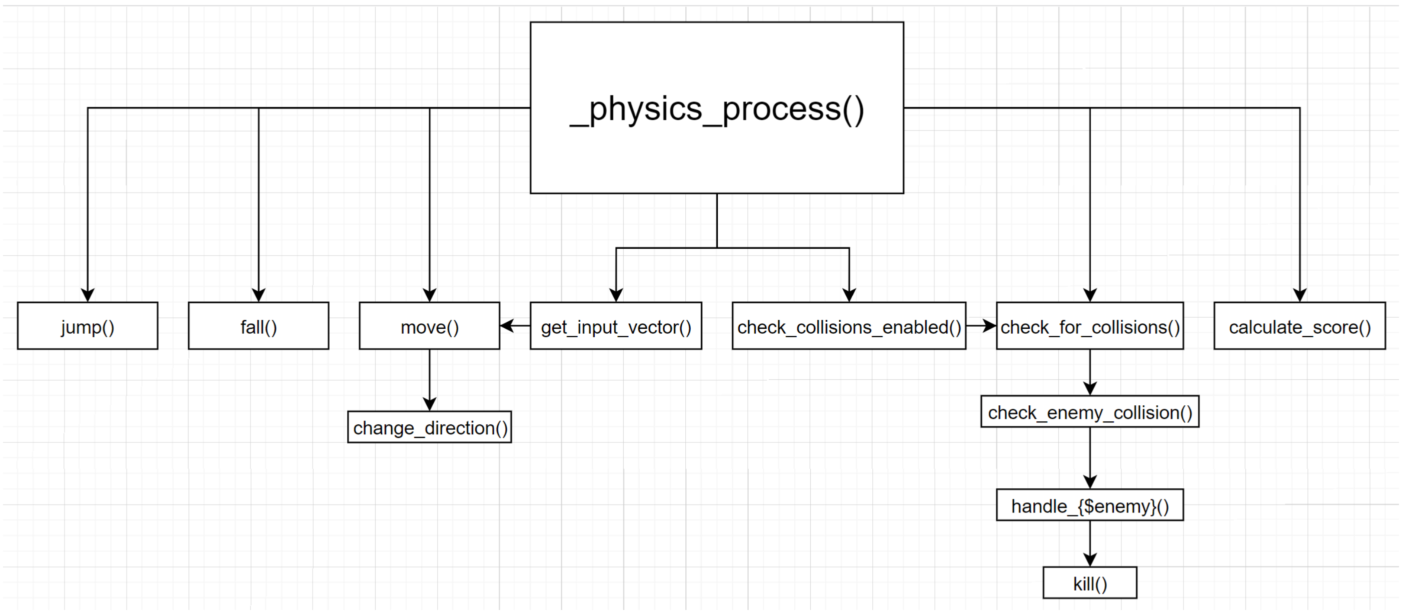
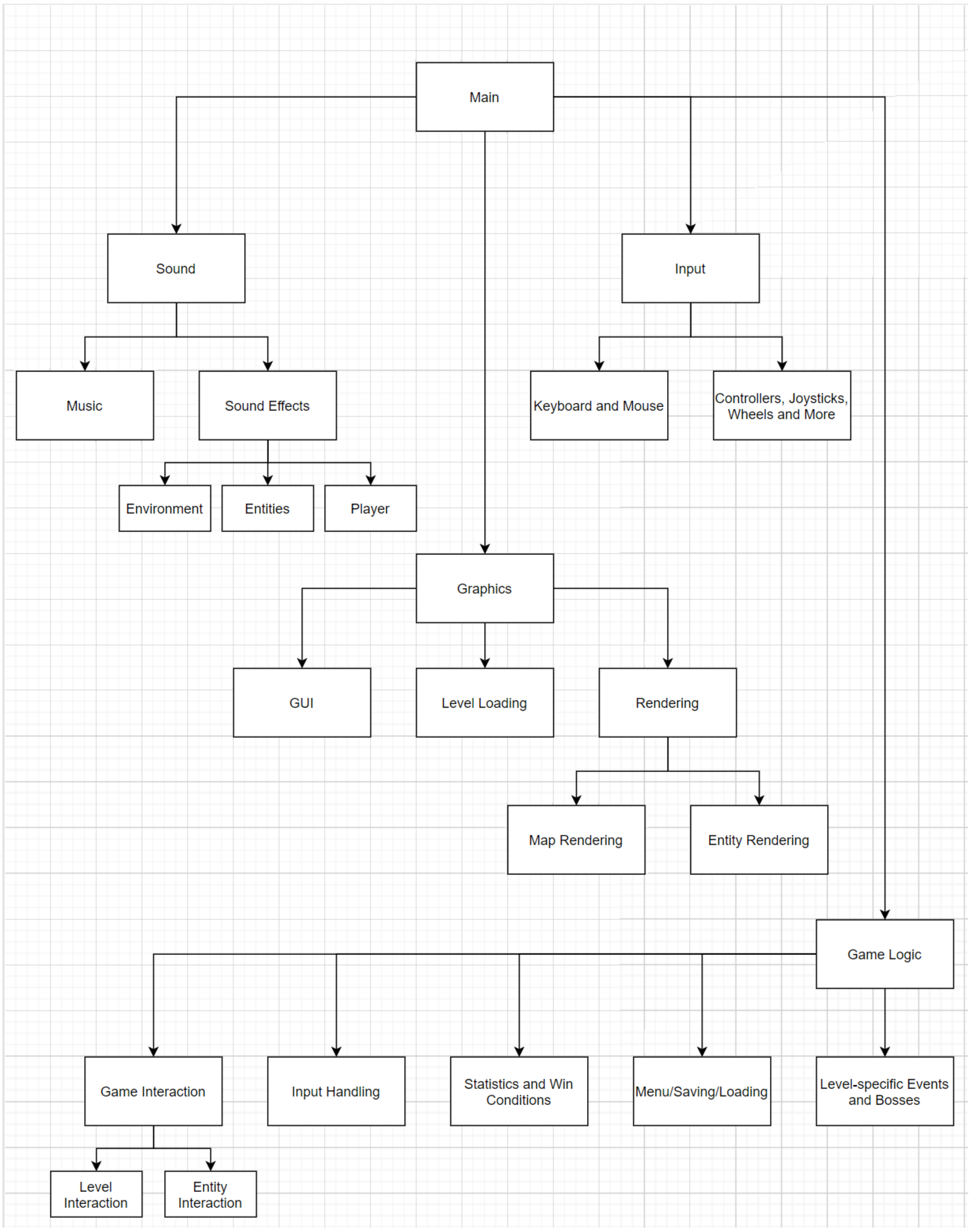


Diagram of how the main function of the player looks in the example program - this would be all the functions necessary for a simple platformer game.
On the next page there is another example diagram.



This is an example of an simple diagram for a first person shooter style game. In a game, usually, most functions are for dealing with the game logic, but the most demanding are the ones that deal with graphics and entities. A game is the sum of tens of thousands lines of code, or millions of lines of code as is the case with modern games.
Also, yes, you can play Doom with a racing wheel - https://www.youtube.com/watch?v=RRX-niBSyH2U

How the example code is structured

As part of this kit, a foundation for a platformer game is included. That foundation includes a fully fledged first level – complete with coins, enemies and a victory. There is a score counter and the player gets 3 lives. The code breaks one of the rules stated previously – as little comments as needed, but I believe that for people that are just starting out or need a refresher on Godot, these comments can help so they will stay. Most of the code is located in the player script (called Tux.gd), and a few of the other scripts interface with the player through the functions found in the bottom part of the script. Godot makes it really easy to interface different scripts with one another, and that function is used a lot in the example. Special attention has been paid on performance and not wasting CPU cycles, but all the specific optimizations will be explained in subsequent analyses of all the parts that make up the code.

The most important part of a platformer

When programming a game I first start out with the most important part of most games – the movement. Moving around in an empty world is more fun than standing in a finished level, so in a platformer movement is especially important. But first, as touched on previously, it is important to make the movement reflect your audience. If your intended audience is very young, then having complex movement would be hard for them and they would not like your game. Platformers like Celeste have many dashes and jumps which makes them harder to learn, but more rewarding. On the other hand, in Super Mario movement is very simple, which is most likely also based on the fact that Super Mario was on the NES game console which only had 2 inputs that were not directional, which did not allow any complex ways of moving even if the developers had wanted to. Since on modern computers there is a large selection of ways to input movement, we do not need to worry about any limitations on inputting movement, so movement can be as complex as you want.

To move a character, you need an input to tell the game that you wish to move it. For the example project the only type of movement I added was horizontal and vertical movement using the arrow keys. Godot makes it really easy to interface all kinds of controllers and methods of movement, but that is up to you to research it. If you plan on making the game cross-platform and including mobile devices, make sure you design your levels around that. Touchscreens allow for using many fingers simultaneously, but don't allow the speed of switching keys that a keyboard provides. Good cross-platform level design should accommodate all kinds of input devices. If your audience is also younger, remember to make everything slower paced, as kids do not have the trained reflexes of teenagers and older. The same goes for older people or people with motor issues. Remember to accommodate such people with either a setting that makes the game easier, or even a speed setting as found on older games.

When you have the input devices sorted out, it is time to think about the feel of moving the character. Do you want fast paced movement based on reflexes like in Sonic, or slower paced and more planned movement like in Super Mario and SuperTux? Movement defines the feel of your game, as the player is viewing the game through the eyes of the camera following the character. The camera also plays a large role in this, but that will be discussed later. For the example game, I chose to make the movement more slow paced like in Super Mario and SuperTux. You should test out multiple ways of movement and find out which one you feel like suits the game best.

Remember that movement on your particular computer setup is not the same as other people would experience it. Even when you find movement settings that feel good to you, test it out on different devices. As stated previously, movement on a mobile game should be a bit slower and accommodate a smaller screen, but more important than that is the feel of the game on lower-end machines. Many people around the world rely on older and slower computers, and programmers often neglect making their game accessible for those people. In the movement function (move()), the function that tells the game engine to move the character is move_and_slide(). That function internally multiplies the velocity by delta, which means that even if the game is lagging, the character will move as they are supposed to.

In Godot, multiplying movement by delta is important to keep the pace of the game flowing even if the computer cannot catch up. Multiplying all movement by delta is also extremely important if you want to change the frame rate of the game, or if you want to provide an option for the user to change the game. If all movement is synchronized with delta, then any change in the frame rate provides the exact same experience to the player. If we had not multiplied by delta, then by increasing the frame rate, you would increase the movement speed of everything, which would also increase difficulty. But wait, in the Snowball.gd script, we multiply velocity by delta, but not in the player's. Why is that?

Move_and_collide() vs. move_and_slide()

Godot provides two functions that are almost always used for movement, but there are slight differences between these functions that become important as your game increases in complexity. The first function is best used for objects and entities, because when colliding with another object it stops moving if the object is blocking it. So, for example, you could detect if the Snowball enemy is colliding with a wall and then tell it to reverse. The reason the second function is better for the player is the fact that when move_and_slide() hits a diagonal, the function would do as it says and slide along it. This allows for smoother physics and less code. For move_and_collide(), you have to provide your own programmed response for when an object collides with another.

One of the most important reasons move_and_slide() is used is because of something called the floor normal. The floor normal tells the physics engine where the floor is located. With that, you can very easily detect if your character is touching a floor, touching a ceiling or touching a wall and create specific responses for all three types of collisions with the world. But you first need to tell the physics engine where the floor is, which is why we provide a Vector2 as the second argument for the move_and_slide() function. Remember, only move and slide needs a Vector2 argument. In Godot you can immediately read the documentation of a function by holding CTRL and left clicking the function whose definition you want to see. You can see that for move_and_slide() the second argument is up_direction, so we need to tell it which way is up. In Godot and most 2D game engines, up is negative, so we provide Vector2 with (0,-1) coordinates to tell it which way is up. This way it is really easy to make a game where you reverse gravity and do similar tricks. You only need to change the vector and velocity, and you can run on walls and on the ceiling!

In the move_and_slide() definition you can also see the default values for all the other optional arguments, which can come useful in many cases. Godot makes reading documentation very easy, and the documentation itself is amazing. Most functions also have an explanation on how the function itself works. The documentation also says all the facts you need to know about the function, and explains how the arguments work as well. Always read the documentation for a function, even when you know what it does. You can learn so much about both how the function works and how the game engine works under the hood by just reading. The documentation also usually includes a link to online tutorials and examples that show how one or more functions work. Some functions are not well documented, though, but usually they have been talked about in an online forum somewhere. As stated before, if you do not know something, the Godot community is very helpful and there is always someone that knows something about the questions you ask.

Jumping and gravity

The node we use for our character does not provide functions for applying gravity, but it is really simple to do if you know any middle school physics. Gravity provides a constant downward acceleration to all objects on earth. Specifically, for a 2D game, gravity pulls everything down. To apply gravity, we just need to apply a constant acceleration of 10 to the movement vector. Interestingly in the real world gravity is ~9.81m/s^2, and plugging in the same value for the Godot engine gives us gravity that feels very nice, just like it is on earth. You can see that in the fall() function, if the player is on the ground, the gravity does not increase, so that if the player jumps, the cumulative force of gravity will not prevent them from jumping.

A very helpful side effect of applying gravity like that is the fact that the player sticks to the ground and collides with the ground. If gravity is not constantly applied, then the player might slightly drift away from the ground, and the player would not be on the floor, which would make it impossible for our jump function to work. You can also see in the fall() function which handles gravity that if the player hits a platform above them, then they lose their momentum.

The jump() function is very simple – if the player is on the ground, apply a specific instant force that makes the player go up. Because of how gravity is applied, when jumping forward the jump forms a parabola, which feels realistic. You can see that both in the jump() and move() function there are speed and jump multipliers. With this it is possible to add effects that increase or decrease movement speed, while still having reference values. If the project was larger and the speed was constantly modified, it would be possible to modify the speed and have no way to bring it back to its original speed. Having multipliers that are equal to one means that all effects can be easily removed by resetting the variables back to one. With the movement of the player out of the way, what about the other mechanics of the game?

The animations

When making a game without a game engine, animations are really hard to get right. They require precise timing and a good state machine to track everything the player does and play a specific animation. Luckily, Godot makes animations really easy to implement with the AnimationPlayer node. All it takes is an image with all the animations, and you can chop up the image in the AnimationPlayer very easily.

All of the animations in the kit were done within the Piskel app, which has support for rudimentary animation. Both Tux and the Snowball enemy are animated, with Tux having animations for jumping, moving and idling. The timing of the animations is very important, and if you add speed boosts you should also speed up the animations, otherwise it will look clunky. Snowball only has one animation, which is moving. A fun fact is that I based the Snowball animation on a video of a chicken walking that I found on the internet. Now, let us meet Snowball!

The enemies

As I wrote a while back, I decided to only add one enemy to keep the game simple and easy to modify. The enemy I added is called Snowball, based on the snowball enemy in SuperTux, which itself is based on the Goomba enemy in Super Mario. This type of enemy is simple – it just patrols its surroundings and in a collision with the player, it damages the player. But if the player jumps on top of the Snowball, the snowball dies. Since the collision shape is a circle, make sure to jump right on top of the Snowball, or it will hurt you!

The Snowball patrols a specific area based on a tether location. Basically, when spawning the snowball remembers its initial position and when it gets a certain distance away from that original position, it turns around and goes the other way until it reaches the same distance from the original position, turns around again and does that forever.

The tether position is determined in the beginning of the scene, and the only thing the Snowball can do is move around its tether. Realistically, the Snowball is extremely simple, except for a small bug that happened when turning the Snowball around. It turns out that the usual way of flipping around a node, by changing its scale, does not work very well when referenced by itself. So, to turn around the Snowball it is necessary to both turn around the sprite, but also change the movement to go in the opposite direction.

The Snowball also has something that the Goomba in Super Mario lacks – the power to not fall from cliffs! With a simple raycast that goes in front of the Snowball, the Snowball can detect whether moving forward will cause it to fall off a cliff. This means that Snowballs can be put on small platforms without them falling off.

Finally, the snowball has a kill() function that can be called from any other script. This makes it really easy to have two players at the same time, for example. The kill() function can also be called by the raycast below the map that detects if an object is falling below the map and kills it because it will only waste resources if it falls endlessly.

Coins, collectibles and the score

No platformer is fun if you are not chasing after coins! In platformer game design, coins are usually used as a small reward to the player for doing something harder than just rushing through the level. Coins should be rewards for completing the hardest puzzles or defeating a pack of enemies instead of running away from them. It should be noted that collecting coins should not be made mandatory. In the kit, the coins have a script that checks if the body that collides them has a function that is called collect_coin(). This makes it possible to have multiple players or even enemies that destroy coins. Coins serve purely to motivate players to do more than rush through, and not to force them to do the hardest challenges or to play the game as you intended it to be played.

There is a whole community around beating games as fast as possible – speedrunners – whose main goal is to beat the game as fast as possible. The most common games for speedrunning are platformers, as they usually only have one goal – to run through the level as fast as possible. That community is very dedicated and can prolong the life of a game way beyond what its creators intended. Super Mario still has a dedicated speedrunning community almost four decades after it was released. You should try to think about those players and how to make the most dedicated fans still continue to have fun. That is why there is a score as well as all the coins collected.

The score, at the moment, is just the number of coins collected and enemies killed multiplied by specific values and when you win you also get some coins. In the past, when programmers did not have the storage space to put many levels in a game, the score was necessary so that the game could have replay value.

The score also adds a social factor to the game – players can compare their scores and try to see who can get the highest score. Usually, games also take the time the player takes to complete the level into account when calculating the score. The faster the player completes the level, the higher their score would be, but that means that they would also have to miss some coins and enemies. This requires a lot of skill and practice to figure out the best combination of moves to get as high of a score as possible, which adds a lot of replay value.

Adding collectibles is also always a good idea. Having collectible items that are located in hidden areas, or require doing some challenge to collect, force dedicated players to find them, thereby playing more of your game. They can also create guides online on how to find those collectibles, which makes your game have an online presence. Everyone wins!

Finally, another big way to make your players continue playing is by adding multiple difficulty settings. Usually three – easy, medium and hard – are used, but you could also add more, maybe even some difficulties where a significant game mechanic is changed. If you want, you could name your difficulties like in Wolfenstein 3D, where the game mocks you in a friendly way if you select the easier difficulties. Try to make your game stand out like that, and make it as replayable as you can. So we talked about the player, the enemies and the score, but what about the map?

The first level

The first level was created as a demonstration of all the components that make up the game. It is made using the TileMap node in Godot, which is a very powerful type of node that allows you to essentially draw a map with a mouse very quickly. Just choose an area of a group of sprites, set the size of the area and you can freely draw your level and sketch out a level in no time!

For the first level I used the Kenney tile set, and it turned out to fit the art style of the game very well. The colors of the tiles matched the colors of Tux and the Snowballs very well, and the small decorations in the tile set livened up the map a bit. Unfortunately I did not find a background that suited the style of the game that had a permissive licence. So, the level is stuck with having a dull gray background.

The level was designed with demonstration purposes in mind. All of the features of the game are displayed, and all of the design principles are followed. There are intermittent patches of the ground and platforms. Like in other platformers, there are challenging parts, then parts that contain rewards for crossing those challenging parts. When Tux reaches the igloo in the end, a short animation is played, the screen is blurred and a button to restart the level appears. If you want to add multiple levels, you would have to make a menu and change the button to go to the next level.

But how do you design unique levels? To

find out, I spoke to some of my friends who are indie game developers, and I got this list of steps for designing platformer levels:

1. Think of what makes this level unique – like the gimmick-based design mentioned above, why should this level be made. You should not make forgettable levels! I played Super Mario a long time ago, and yet I can recall the themes and gimmicks for most of the levels. This is what effective game design looks like.
2. After you have your idea, think about the impact of the level in the game's progression. Does it matter to the story in any way? Is anything new introduced? Going by the Super Mario design philosophy, every level should introduce a new enemy, a new theme, a new powerup and more. Levels need to justify their existence in the game.
3. Make a story of the level in particular. Is there a specific cutscene that needs to be played, or a twist in the story? The color palette should not be static as the story changes. To use another example from Super Mario, the later levels replace the natural and cheerful colors with red and black. The later levels become harder and the story becomes darker, so the colors change to accommodate that. Interestingly, the later levels become more claustrophobic. You are stuck under water or in a castle with walls on the top and bottom. The levels manage to tell the story without the need of cutscenes and walls of text. Remember one of the most important rules in art: show, don't tell.
4. After deciding on the theme, draw a sketch of how the level should look. Remember to make your levels diverse – the player should go in multiple directions, jump around and most importantly, have multiple ways to win. That adds replayability to the game and allows people who have different play styles to be able to finish the level without struggling too much. Allow choices – risky routes with rewards or safe routes without rewards, lots of enemies or lots of platforming. Let players choose how they want to play your game.
5. When you consider the level to be done, add it to the game and play it. Try to get people who are inexperienced at your game to play through the level on their own and see when or if they get stuck. Then, try to get people who are experienced and see if they get bored or breeze through the level. Try to ask for honest feedback. It is more important to learn that a level does not work than to pretend that everything is fine. You should also change the level as you see fit. You do not need to follow your sketch exactly – know when to break the rules, even if you made them for yourself!
6. Decorate the map. Like in the one level provided, add trees, bushes or anything that fits your mood and art style. Use all the tools you have on your disposal to create a level that evokes a feeling that you want to evoke. Tell a story through all parts of your game, even the ones that a player only glances at while they are running away from an enemy or jumping to get a coin.

The camera

Finally on our tour of the parts of the game comes our camera. I did not find the default Godot camera settings to my liking, so I decided to take the default camera and add my own functions to move it. Since I wanted an old-school platformer camera, I only wanted it to move horizontally and not vertically. Of course, if I were to design more complex levels that have different vertical paths and challenges, I would change that, but Godot's modularity makes modifying the camera for each scene pretty easy. The camera works in a relatively simple way – it detects if Tux changes position, and moves accordingly following some rules.

The camera also contains the ending "animation", which is just an overlay over the screen that makes the screen blurry using a shader. Shaders are way beyond the scope of this tutorial, and even most online tutorials, but you can always find ready-made shaders for just about everything. Many good people give away lots of code for free – use that, but do not forget to credit them somewhere. Whether that is in a text file provided with the game or as part of the credits sequence. The open source community is amazing, but we should not abuse the generosity of all the altruistic people who make this community what it is.

Extension challenges

If after all this typing you do not know what to add to this foundation, do not fear! Here I will provide some extension challenges which you can try to implement in your game. They are split into three categories, based on difficulty.
Easy:

- Add one more enemy
- Add a timer to the level
- Add a background
- Add a reward block that gives simple power-ups when hit from below (Godot has a function for this!)
- Make the camera be able to move vertically
- Add a menu
- Add one more level

Medium:

- Add sound effects and music
- Add saving and loading using JSON
- Add powerups that can give other modes of movement (flying, riding on an animal)
- Add environmental obstacles (falling ice shards, ice that breaks when you walk on it)
- Add moving platforms
- Add highest score counter for all levels that stores data in JSON
- Make the controls mobile-friendly

Hard:

- Add a world map to see which levels the player has passed in graphic form
- Add local co-op play with two or more players
- Add bosses to the end of every level
- Add cutscenes

Extra hard:
Make a website where players can upload their scores and compare with other people

After all has been said and done... what now?

If you, dear reader, have arrived this far in the kit, congratulations! You have listened to me talk about hundreds of hours spent on talking to people, reading books and watching videos, and all of that knowledge I have distilled in a dozen pages and written in detail so that you can work hard and smart to make the best platformer there is. The reason why I have provided a simple example of a video game to this kit is because I want you to take it, modify it and turn it into something great! When I was younger, I always wanted to be able to take a game someone else made and tweak it. I wanted to add something, remove something and be able to call it a game that truly fits my taste. I satisfied that need to an extent with modding – I participated in the modding scene of a game made in 2003, because it was probably the most demanding game my computer could run! So I want you to take this kit and do the same thing I did, but on a grander scale. Many people are afraid to start a project, and do not know how to organize the creation of a project. I want to tell you that if you got this far in the book, if you build upon the foundation I have provided you can make something really interesting. If you do not have any ideas on what to add to this project – do not fear! I will provide extension challenges that tell you everything you can and should add to turn this small foundation into a great game. I have also unintentionally included some bugs in the game due to my wish for the game code to stay as short as needed, so I implore you to find those bugs and fix them. Again, if you have come this far, thank you so much for letting me explain to you some of the things that have made the greatest games, well, the greatest games. But this does not apply to games only, programmers of all types of software can learn from these tips. Though this might have, at parts, seemed like rambling, it is the culmination of months of studying and also most likely the largest project I have undertaken in my life. Stay awesome!

Credits

Thank you to the team at GNOME for making this possible. I have been using your desktop environment even before I knew what "a Linux" was, and the GNOME community as well is awesome.
Thank you to my computer science teacher who helped a lot by teaching me some of these things and unintenionally giving me the initial idea to create this project.
Thank you to all my friends who let me talk to them about game development and software development. Without you this project would probably not exist.
Thank you to all my friends who tested the project and all of my iterations. I hope you had fun and learned something.
Finally, thank you to the amazing altruistic people who created some of the software and assets used in this project. List of all the software and assets used to create everything:

- The Godot game engine
- The Piskel app
- Libreoffice Writer
- Platformer tileset by Kenney – https://www.kenney.nl/assets/platformer-art-winter
- Blur shader by KidsCanCode – https://kidscancode.org/godot_recipes/shaders/blur/
- Coin sprite by madcowj on OpenGameArt.org – https://opengameart.org/content/spinning-coin-8-bit-sprite
- Adobe Indesign

Curated list of Godot learning resources that I have used

- Godot Engine Game Development in 24 Hours – The Official Guide To Godot 3.0 by Ariel Manzur (one of the creators of Godot!) and George Marques
- KidsCanCode.org – https://kidscancode.org/blog/tags/godot/
- Heartbeast – https://www.youtube.com/user/uheartbeast
- GDQuest – https://www.youtube.com/channel/UCxboW7x0jZqFdvMdCFKTMsQ
- UmaiPixel – https://www.youtube.com/channel/UCla6BhPwo5zGal6vR5le4YA
- Miziziziz – https://www.youtube.com/channel/UCaoqVlqPTH78_xjTjTOMcmQ
- The entire Godot subreddit – https://www.reddit.com/r/godot/
- And most importantly, the Godot documentation – https://docs.godotengine.org/en/stable/

And many more!

As I touched on before, this project had been something I had wanted to do for a long time. Now with both the support of the GNOME project and my school, I can finally justify doing such a large project. I have always loved teaching and helping people, and this is enabling me to do it on a global scale! Thanks to both the GNOME project and all of my teachers who helped me along the way. The open source community is especially a big motivator for me. I even started learning about coding standards and design principles a year ago because I wanted to start seriously contributing to open source projects. Currently almost all of the software I use is not only open source, but sometimes software I've contributed to myself. That's why I encourage anyone who uses this or any other programming kit to extend their software and make it open source. There are no downsides to open sourcing your personal projects!

Made by: Stefan Nikolaj
Contact me:

- Email: stefannikolaj@gmail.com
- Website: nikolaj.club
- GitHub: github.com/snikolaj
- YouTube channel: https://www.youtube.com/channel/UCJxzNR_M2Urx6u30l7i-8uQ