Final Report - Game Project

University of Royal Holloway

Supervised by - Riquelme Granada, Nery

Krzysztof Tos

Contents

1 Introduction

1.1 The project

The reason why I chose a game project is because I want to become a game developer. With the project I want to learn the basics of programming various methods and classes for a video game. I want my end goal to be a playable demo of an RPG game where the player can perform multiple tasks and play through a limited amount of quests. On top of that I want to develop a video game without advanced tools newer engines provide so that I deepen my understanding on how certain features of a game are developed. In this report I will be discussing and explaining different parts of my project. Firstly, I will be talking about different programming patterns. The patterns include things like:
- Design that explains the usage of singletons, observers and states, sequencing and optimization.
- Sequencing that covers the usage of a game loop, update and draw methods.
- Optimization that talks about overall game performance by using object pools and spatial partitions.

Secondly, I will go over different game engines and frameworks that I could or have chosen for my project. Today there are multiple choices to go with when choosing an engine but not all of the will fit the game so it is important to go over these sections as to understand what engine is best to build my game on and what will give me most experience in the process. Thirdly and lastly I will go over different parts of my code. I will explain the functionality of the code, why I included it in the game and how I went about in developing the idea. This sections will go over things like animation, collision, quest generation, sorting algorithms and the like.

Of course, I will also go over different professional issues that can arise when my project is released into public. Things like copyright issues when using third-party assets, plagiarism of my project by different developers and using them for their own games or ideas, things like bugs may also be an issue when the game is released and lastly the security issues of the game.

The project itself could not be completed without external help by using books or forums so I will dedicate a chapter for the sources I have read when I was developing my project.

1.2 What is game development?

Game development is an artistic way of creating entertainment in a form of a video games. Game developer is a broad title that usually includes programmers, artists, voice actors or even mathematicians and other scientists. Development of a game follows similar principals to developing a software but is proven to be longer and harder as games require a lot of moving parts therefore the game needs to be flexible and preferably with parts that can be reusable by different objects, there is planning, development, testing and release. As an solo developer I will have to cover every task myself, but mostly I was focused on the technical part of the game which involved planning and developing parts of the engine.

1.3 What game do I want to develop?

I was always a fan of 2D RPG games therefore for this project I wanted to start developing basic engine where I can move the player, do few quests, have an inventory and building system, therefore I decided to design a farming sim where the player can plant few plants and craft resources. I also didn't want to create a linear game as I wanted the player to decide what to do at their own pace so I got inspired by a title known as '**Stardew Valley**' where the player is given a piece of land, tools and seeds and is given full freedom on their play style. Artistically, I wanted to create everything in pixel art, and for this purpose I used a program 'Marmoset Hexels' to create sprites.

2 Project Specification

2.1 How to start the game, controls and how to play the game

To play the demo go to: GameOnly/GameProject.exe

Move up - W
Move down - S
Move Right - D
Move Left - A
Open/Close Inventory - I
Open journal - J
Interact - LMB
Place object on map - RMB
Add Health - P
Subtract Health - O
Add Mana - L
Subtract Mana - K
Add apple - M
Add Tool - N
Close ui/Open Main menu - ESC

At the beginning the player is spawned with just few starting items. Two most important items are the plant pot and a kettle. Place them down and wait few seconds for the plant to grow (for the demo I shortened the time needed for plants to grow), once the plant fully grows press on it and gain green tea leaves. There are four NPCs in a line. Each one of them gives a quest. Accept all quests and if the player already has enough green leaves in inventory, right click on the npc that wants green tea leaves.
The player will be given apples that can be used to craft apple tea by right clicking on the kettle and choosing apple tea. Follow the same instruction for green tea quest to finish all quests.
At any time the player can open up their inventory, split items, organize or throw away items. The player can open up journal to see all the quests and by left mouse pressing on one of them it will display all details about the quest and by pressing 'esc' the player can pause the game.

There are few features that are in the game but have no particular purpose. The clock and health/mana bars are these features. The health can be added and removed by pressing 'O' and 'P' keys and mana can also be manipulated by pressing 'L' and 'K' this process is animated and could be used in the future.

My aims and objectives and what I want to discover for this project are as follows:

2.2 Aims

Aims:
- Learning about usage of vectors in video games by drawing sprites, calculating distances and creating movement of either NPC's or player's.
- Learning about creating, loading and saving maps using textures. Some engines provide GUI that gives visual scene of the map creating in real-time, it also provides an option to save or load a scene. I want to discover how to achieve that by writing map classes and creating save files myself.
- Learn about NPC behaviour and how to program a simple AI by creating enemies, allies and neutral NPCS.
- Discover how to optimize the game by writing efficient code and using threading system.
- Learn about customization of certain aspects of the game. I want the player to be able to change colours, clothing, hair and the like. I want to achieve that by implementing a character customization .
- Learning on how to implement quests and combat by creating tasks and system where player can fight with certain NPCS.
- Learn how to use sounds in a game and how to best use them by creating various sound classes that are played in different situations.

2.3 Objectives

Objectives:

- Create a playable and scalable scene that will work on different environments and also a camera that will always follow the player.
- Create a player that is able to move around and performs certain tasks.
- Make a demo map, with quests and NPCS.
- Implement conversation system with NPCS and the player.
- Create a player build area, witch can be edited by the player and saved in the game files.
- Implement inventory system, currency, and equipment system.
- Implement some customization that can be saved and loaded.
- Make a simple menu with start, load, options and exit tabs.

2.4 Future extensions

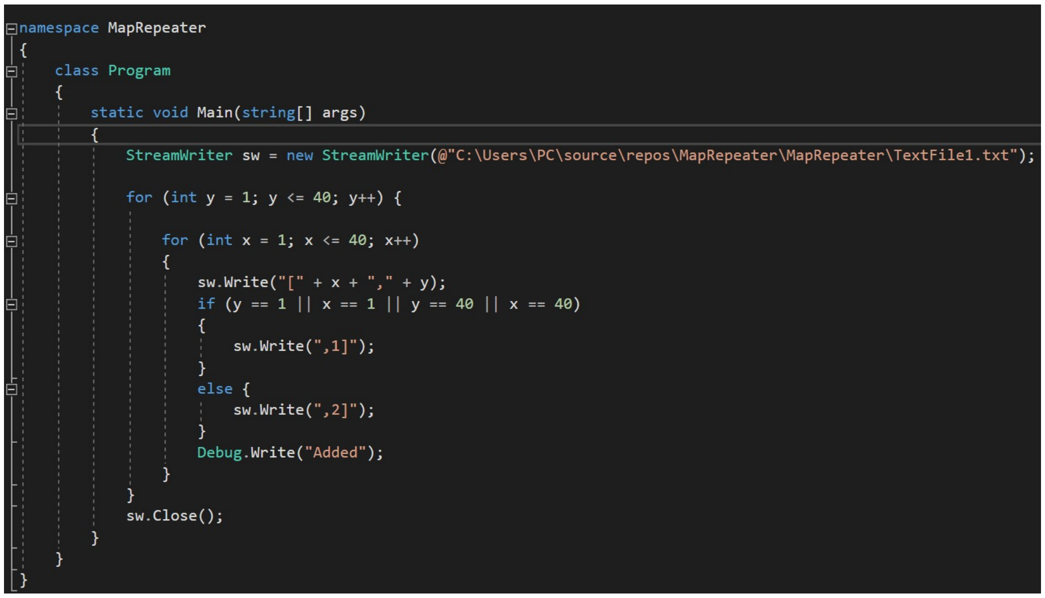
The project is just a demonstration of what the game can be and **in the future** I want to include different things into my project. These things include:

Usage of tools - At the present there is no functionality of tools. The tools are important in a video game because they limit player actions and force the more interaction and planning with the UI and inventory system. Different tools may also have different bonuses when doing their specific actions.
NPC path-finding - At the present NPCs do not move and act as a part of the static scenery. In the future I want to have NPCs that move around, do certain actions like combat and overall be more dynamic.
Object rotation and pixel based collisions - When building the player doesn't have any ability to rotate objects, therefore in the future I want to allow the player to rotate certain items. On top of that I want to have a way of precisely detecting collision by using the program to check what pixels were pressed.
Combat, enemies and consumable items - As an RPG game there almost always is a need for enemies and combat as it extends playability of the game. I will want to create a simple combat where the player dodges, blocks and hits back enemy sprites. I already created usable health and mana bars so they can be used for the purpose of combat. On top of that I would also want to have certain items that when used will have different effects on the player like adding health when eating wheel of cheese.
Customization - Part of the modern games is the ability of changing clothes, colours and shapes of items and characters. This feature would add more depth to the game and have the player add more of their own identity to the game. As the starters I could create a character creation screen when the player first starts a new game.
Map creator - For this project I have been manually creating a text file that holds tile information. In the future or in the next project I would most likely create another program that would help me create maps. I would make a simple map editor with ui that creates text file with all the tile data in it and then use it for my game. While making a demo presentation of my project I decided to quickly write a scrip that creates and writes a map to a text file.

2.5 Software Engineering in game development

Software Engineering in game development

Software engineering translates pretty well into game development. For starters software engineer will apply standards into the code, it means that the developer will keep constant way of programming to make the code readable. This is very essential because bad code will result in long amount of time spent trying to understand certain parts of the code. Standards may include singletons, splitting



```
namespace MapRepeater
{
    class Program
    {
        static void Main(string[] args)
        {
            StreamWriter sw = new StreamWriter(@"C:\Users\PC\source\repos\MapRepeater\MapRepeater\TextFile1.txt");

            for (int y = 1; y <= 40; y++) {

                for (int x = 1; x <= 40; x++)
                {
                    sw.Write("[" + x + "," + y);
                    if (y == 1 || x == 1 || y == 40 || x == 40)
                    {
                        sw.Write(",1]");
                    }
                    else {
                        sw.Write(",2]");
                    }
                    Debug.Write("Added");
                }
            }
            sw.Close();
        }
    }
}
```

Figure 1: Another simple program I wrote to help me create a map

code into separate classes, code directories and most importantly comments. Another thing software engineer may introduce into a game is implementing computer science principles into a game. This may include things like creating a well documented deign for the project or applying algorithms to optimize the code and achieve certain outcome.

For my project I have decided to stick to modularity the most. Most of my classes can be reused by multiple objects. These classes include 'Sprites', 'items', 'quests' and the like. I also created few singletons for the purpose of keeping only once instance of them since the data created by them was used universally by all classes. My code consists of 'Input', 'Mouse Observer' and 'Quest Manager'. I have used tree nodes for the dialogue in quests and also used selection sort algorithm for my depth draw class. These two things improves my game development and optimization respectively. Lastly I also used params for classes that are complicated and need explaining.

2.6 Timeline - Term 1

Timeline - Term 1:

05/10/2020 - 18/10/2020
During that time, I plan starting the development the game. I'll set up an SVN repository to save all my future progress, start working on my diary and creating basics for my game. I plan on creating a simple player and sprite class so

I can draw my character on screen then I'll add movement and input classes to the game and make my character move. Lastly, I'll implement basic animations that will change the sprite based on what direction key was last pressed.

19/10/2020 - 01/11/2020
During this period of time I'll be working more on animations. I want the sprites to be animated such that they constantly change when the player is moving in certain direction or idle. I'll also work on adding a following pet, that goes everywhere after the player. I expect animations to take a bit longer to implement so it will be all I'll be doing this time around.

02/11/2020 - 15/11/2020
Now that I have working animations and movable player, I'll be adding a camera that stays attached to the player at all times. I'll be also creating a test map that contains walk-through sprites like grass and impassable sprites like water or hills. I plan on adding an idle NPC that will show a speech bubble once the character gets close to it.

16/11/2020 - 29/11/2020
This time around I'll be implementing an in-game inventory system with few items to equip and use. I'll be adding health, experience and energy bar on the top left corner of the game and on the bottom a tool bar that resembles the first row of inventory and the items that are placed there. Lastly, I will be developing a clock at the top-right side of the screen.

2.7 Timeline - Term 2

Timeline - Term 2:

30/11/2020 - 13/12/2020
After I've completed a user UI, I will be working on the farming part of the game. I'll create a system where the players have to dig the soil, place the seed in the soil, water it from time to time with a water can and once the plant grows to the max the player will be able to gather the leaves. There's no item drop system implemented yet so I'll be just adding items directly to the inventory.

14/12/2020 - 27/12/2020
Around this time, I want to work on an item drop system. When the item drops, I want to implement a little bit of bounciness to it so it catches the eye of the user. The next thing I want to implement during this time would be particles. I want to create a particle system that can be deployed anywhere I want. The particles will have burst effect and continuous effect modes. This stage will also be last for the first term.

04/01/2020 – 17/01/2021
During that time, I want to add some more sprites and work more on the map of the game. I'll start implementing a way of loading the map from a file. The process will take place by calling a method that will read from txt then it will pass the details to the writer method and finally a full map will be loaded. First map will be simple.

18/01/2021 - 31/01/2021
This time I want to work on user statistics, I will add health, energy, speed, gathering, brewing and luck statistics for the player, once that is done, I want to add modifiers that will change based on what items the player is wearing. Each item will be given statistics that will change the player modifiers.

01/02/2021 - 14/02/2021
This time around, I plan on adding a working NPCs quest. First, I'll add a journal that will be displayed once the player pressed 'J', it will include all the current quests. Secondly, I'll add an NPC that once interacted with it will display a text box on the bottom of the screen and a text will be displayed. The game will read from the text file and display its contents to the box. Lastly, at the end of the conversation the player will be able to accept the quest or deny it. If accepted the quest will be added to the journal.

15/02/2021 - 28/02/2021
Sound is one of the most important features for the game. I plan on adding a class that will take a sound file and play it. I plan on adding music, ambient sounds and walking sounds therefore there will be few instances that will use the sound class. Music and ambient sounds will be played automatically by a

sound controller and the walking sound will be played when the player is pressing the movement keys. Movement will have few walking clips and they will be randomised.

01/03/2021 - 21/03/2021
Now that most of the features are in the game, I'll be focusing on the aesthetics of the game and I'll be designing a map, creating a player area, farming area, a town, adding NPCs, few quests and few buildings.

22/03/2021 - 04/04/2021
Now the project will be almost over therefore I'll want to try and implement a simple character customization screen that will be displayed when the player interacts with a block in a shop building, here the player will be able to change the look the character and the changes will be saved in the save file. The player will be divided into few sprites at this point.

05/04/2021 - onwards
Now that the key features are in the game, I reserved some time to continue on code that might have taken a bit longer to complete, adding small combat system, implementing minor features that could be thought of during the development of the game and finally bug testing.

2.8 Diary

Diary:

28/09/2020 For starters, I had a meeting with my supervisor and we talked about my project plan. I've been told to restructure my table of contents, add a section on why I'm doing the game and the conclusion at the end. This overall concluded my first project meetings. 05/10/2020 I've set up a project, connected it to the SVN using visualSVN plugin for visual studio, I've created basic repository structure (trunk, tags and branches). I've encountered a little issues with the plugins as they were giving me errors and I wasn't able to branch my project but I fixed it by installing tortoiseSVN and branched the project by hand. 08/10/2020 I created an input, player input manager and draw classes. I've also played with the game window resolution, made it open in full screen and also printed a sprite onto the screen. I also thought a little about how to structure classes responsible for the player and his input and decided to use player input manager class as the centre of operations for now. *see graph included* First page of my diary I found the 1MB limit too small even after compression for my pictures so I uploaded the image from external link.

12/10/2020 – 15/10/2020 Changed the way player sprite is drawn and opted for the sprite to be drawn in the player class. Normalised movement vector so that when two movement keys are pressed (e.g. W and D) the sprite doesn't move twice as fast and has a constant speed. Capped the movement to game seconds instead of FPS to keep the movement constant, started working on animation class, also started drawing an original character sheet.

20/10/2020 – 23/10/2020 I've added getters and setters for animation class, I then created animationManager that holds a single animation class and plays it. I plan on dividing the textures with animations that only correspond to single direction movement, e.g. a texture 'PlayerMoveLeft' will hold animation for movement to the left. In the player class I initialised animationManager and animation dictionary and lastly I've been working on a player sprite.

24/10/2020 – 30/10/2020 I've completed animation and animation manager classes. Created a sprite sheet with simple idle animation and proceeded to animate it. I've also started writing code for the pet class that I'll be implementing.

02/11/2020 – 08/11/2020 Wrote a design report, finished a pet class.

09/11/2020 – 15/11/2020 This week I've created a camera class that make the screen follow the player, I've also created basic testing map that I used to see the camera changes when I move the player.

21/11/2020 – 25/11/2020 This time around I've implemented a health and mana bars. The game draws 255 pixels for health, 207 pixles for mana and they all correspond to 1 health point or mana each. The health class is like an observer

that checks the difference of health each loop and once the change was detected, the pixels are animated. To test the bars I programmed buttons that check add or remove points from health or mana. I made some minor changes to the way game detects what keys are pressed each loop and deleted some unused methods from input. I have started working on inventory and item classes, they are about half done, and lastly I've worked on my second report about Monogame and other game engines.
26/11/2020 – 30/11/2020 I've worked on project report. Added small bits of code to inventory class, it is still not completed. Planning to finish it before 4th of December.
01/12/2020 – 03/12/2020 Finished Inventory class. Player is able to pick item from the slot and place it in another slot. If slot is already taken it replaces held item, on top of that I changed the health and mana classes so they look more presentable.
04/12/2020 – 07/12/2020 Firstly I added Toolbar to the bottom of the screen, I had to change the inventory class a little to accommodate it, but it now reflects first 9 item slots in the inventory. Secondly, I've added drop button, when the user is holding an item and presses on the button it will remove the item the player is holding. Lastly, I've added a clock into a game, where 2 seconds in real time will reflect 1 minute in the game time.
25/01/2021 – 03/02/2021 I've been implementing collider function from scratch and it's been quite challenging process. In the process of making the collider I met with many bugs and I had to rewrite my code three times to achieve what I wanted. Each tile that has a collider will now check if there's collision by taking future position, check where the centre of both colliders and based on X and Y move the player away from the collider. The most notable bug I have encountered was when moving up and to the side the player would get stuck between colliders. The cause of the bug was that tile collider1 would move the plater up and then tile collider2 would move the player down again. The solution to that was to splice the colliders into four parts and each part would move the player in different direction. Another feature I've implemented was the selection of toolbar item by pressing 1-9.
04/02/2021 – 07/02/2021 Modified collision manager. Because I was checking for both X and Y collisions at the same time I encountered multiple bugs so I changed the code to move the sprite X first and then Y second, this eliminated the problem and the sprite can slide alongside the colliders without problems.
10/02/2021 – 14/02/2021 Created depth class that holds sprites in the list and sorts them based on their Y value, it then draws the items on screen. The method was created to imitate depth of the sprites, in other words if the sprite1 is higher then sprite2 then sprite1 will always be behind sprite2. I also created sprite class so that any new sprite will use DepthDraw Class.
15/02/2021 – 19/02/2021 I created new NPC and NPC manager classes and added an NPC that I will use to implement the first quest in the game. I also started working on quest system, the way I'm implementing it is that I'll read the contents of the .txt file that holds all the quests. I'll then use Quest class to hold different nodes of text, essentially creating a tree holding NPC dialogue

and then add it to the said NPC.
19/02/2021 – 09/03/2021 Finished dialogue system, when the player presses on an npc that holds a quest, it will open up a dialogue window. The npc will then pass tree nodes to the dialogue class, which will change text after player input. At the end of each node if a player response exists, the game will generate responses that will lead to further text nodes. Another thing I have been working on is a quest system which is almost done, I only need to make code that will take items from player inventory and give rewards.
crafting system where a player places an item on the floor, presses on it and it will display ui for the crafting. In the demo I have included two items to be crafted. Green tea and an Apple tea. They both remove something from the inventory and give new item. Second thing I made was a simple main menu where I can start a new game, pause while in playing and exit the game and lastly I made few more NPCs and quests to accept.

2.9 Issues during the development

2.10 SVN - missing Monogame contents

Using SVN was very useful during the development of the game project.I could change versions, switch branches and merge changes into the trunk. My experience with SVN was great but it did have few shortcomings. Monogame uses a content pipeline tool, it allows me to import textures, sounds and the like to the game, when the files are imported I can then easily load files from the content manager. As great as it was it had an issue with SVN, sometimes when I merged a branch into a trunk the contents of the pipeline tool would not be transferred and thus creating missing files errors and on top of that these files could not be downloaded back by switching back to the branch. Thankfully I held a copy of textures in separate folder and could always import them back into the project but it always took a lot of time to restore the file structures.

2.11 Collision

Colliders took me a lot of time to implement. I researched collision detection in Monogame and had basic idea of how to implement them. At the beginning there were many bugs, collision moving the player to the wrong side of the tile, the player getting permanently stuck after collision, collider not appearing in the right position and colliders teleporting the player few thousand meters away from the intersection. Through first week I've been creating theories how to handle colliders, refining and rewriting the code. In total I rewrote the code five times entirely and with each implementation I removed few bugs. At the end there was one notorious bug that I couldn't get rid of. When moving along the wall and hitting the edges of two colliders they would negate player's movements. This bug only occurred on two sides of the tiles (left and bottom, right and top or top and bottom) and when I changed the math of the collisions, this bug would appear on different two sides of the tiles, usually on the opposite sides. The reason for this bug was because I was moving the player in X and Y axis at the same time, I changed the code so that firstly the game moves the sprite and collision in X axis, then repeat the process in Y direction. Overall this was the biggest challenge I encountered during the project implementation.

2.12 Experience

My experience of developing video games at the start of the project was very shallow. Although I did a lot of research for the project I still lacked the on the ground experience of developing a game thus resulting with me rewriting already existing code on multiple occasions. At the beginning, few portions of my code were linear, not applicable to other classes with similar functions. For example I lacked the draw class and instead insisted of creating separate draw methods inside multiple classes. A problem occurred when I wanted to create layering in my project, there was no centralised class that calculated what layer should sprites appear on and thus I had to create one and rewrite classes with their own draw method.

Another smaller issue was that the player class held it's own position data, and when I created a modular sprite class I had to update different classes all across my program to use data from the sprite class instead of the player class.

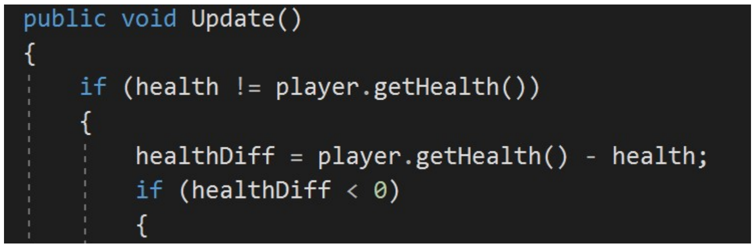
With the experience from this issue I've learned that planning a lot of modular classes in the future will reduce the implementation time.

3 Design

There are few things in design patterns that might be useful but most of the time these three were the most useful when coding:

3.1 Observer

Observer - An observer is a class that, in a manner of speaking, 'observes' the code. Let's say a player activated a certain item that damages every spawned enemy, in order for the enemies to register the damage there has to be an observer that will constantly check what items, skills, e.t.c. the user activates and send a message 'X have been activated' across the network. All other classes will check if they have to do anything with X and do something if so. See figure 1.

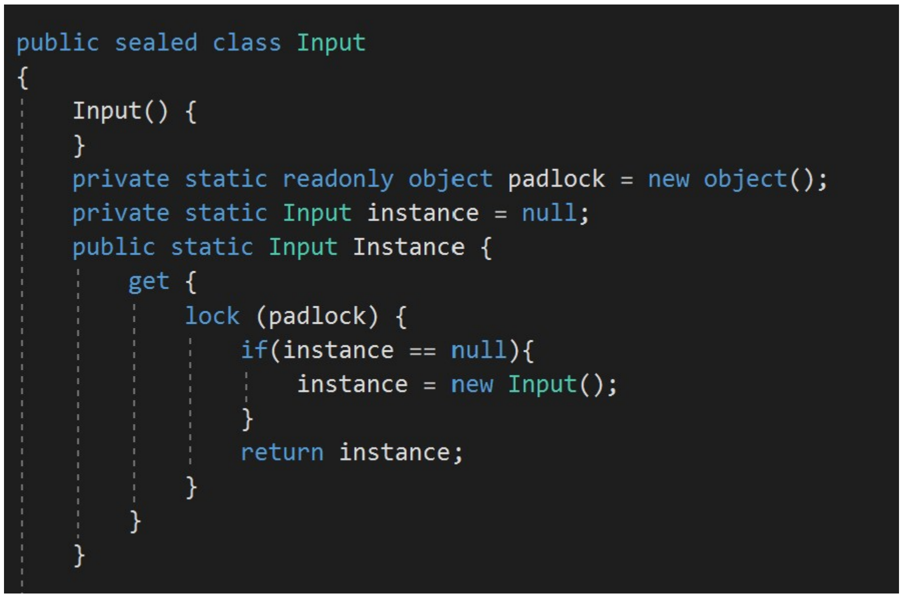


```
public void Update()
{
    if (health != player.getHealth())
    {
        healthDiff = player.getHealth() - health;
        if (healthDiff < 0)
        {
```

Figure 2: Image here the method observers the health state of the player every loop.

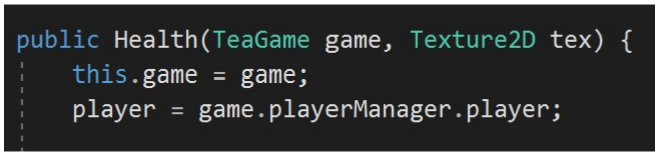
3.2 Singleton

Singleton - It's normal for the programmer to declare a new class whenever something needs referencing, but let's say we have a 'PlayerManager' that initialised a 'player' class. This player class functions normally but another class 'PlayerEquipment' is created and it also initialised a 'player' class. Now there are two player classes and it creates an issue. To combat that we will have to create a singleton which is essentially a class that once created, holds itself and if another class wants to initialise it anew, it will pass itself instead. See figure 2. Another way of avoiding multiple classes is to pass the main game class into classes that were created within main code. See figure 2 and 3.



```
public sealed class Input
{
    Input() {
    }
    private static readonly object padlock = new object();
    private static Input instance = null;
    public static Input Instance {
        get {
            lock (padlock) {
                if(instance == null){
                    instance = new Input();
                }
                return instance;
            }
        }
    }
```

Figure 3: Figure shows input class that's a singleton

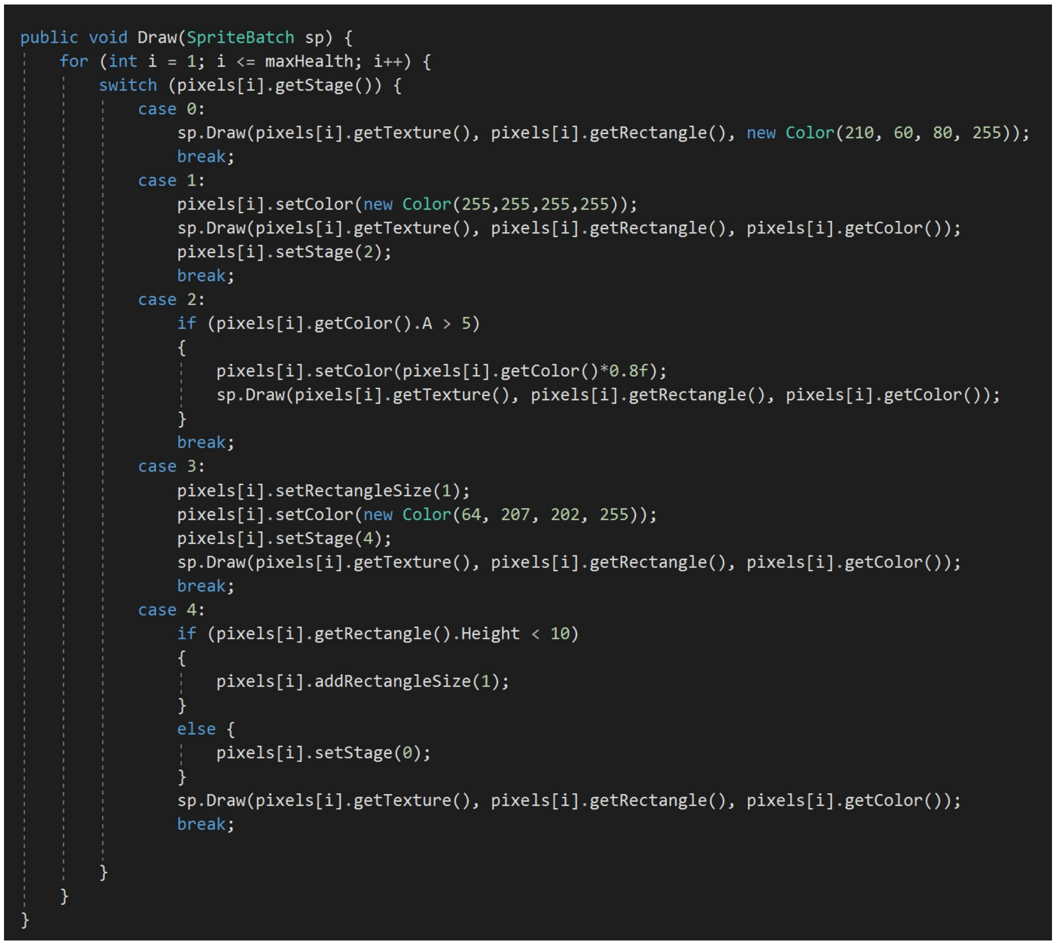


```
public Health(TeaGame game, Texture2D tex) {
    this.game = game;
    player = game.playerManager.player;
```

Figure 4: Figure shows me passing main game class to newly constructed sub-class.

3.3 State

State - A state is a way of changing the behaviour of the class. For example if I press a button different part of the class will play e.g. 'Movement'. In my 'playerManager' class I can press WASD keys and expect different result for each key press. It allows me to manipulate what code is running, and create adequate interactions with the player. See figure 4.



```
public void Draw(SpriteBatch sp) {
    for (int i = 1; i <= maxHealth; i++) {
        switch (pixels[i].getStage()) {
            case 0:
                sp.Draw(pixels[i].getTexture(), pixels[i].getRectangle(), new Color(210, 60, 80, 255));
                break;
            case 1:
                pixels[i].setColor(new Color(255,255,255,255));
                sp.Draw(pixels[i].getTexture(), pixels[i].getRectangle(), pixels[i].getColor());
                pixels[i].setStage(2);
                break;
            case 2:
                if (pixels[i].getColor().A > 5)
                {
                    pixels[i].setColor(pixels[i].getColor()*0.8f);
                    sp.Draw(pixels[i].getTexture(), pixels[i].getRectangle(), pixels[i].getColor());
                }
                break;
            case 3:
                pixels[i].setRectangleSize(1);
                pixels[i].setColor(new Color(64, 207, 202, 255));
                pixels[i].setStage(4);
                sp.Draw(pixels[i].getTexture(), pixels[i].getRectangle(), pixels[i].getColor());
                break;
            case 4:
                if (pixels[i].getRectangle().Height < 10)
                {
                    pixels[i].addRectangleSize(1);
                }
                else {
                    pixels[i].setStage(0);
                }
                sp.Draw(pixels[i].getTexture(), pixels[i].getRectangle(), pixels[i].getColor());
                break;

        }
    }
}
```

Figure 5: In this example I have included state changing of health pixels. Different state will yield different results.

4 Sequencing

The sequencing patterns are mainly the core of the code. Sequencing patterns include things like game loop, update and method.

4.1 Game Loop

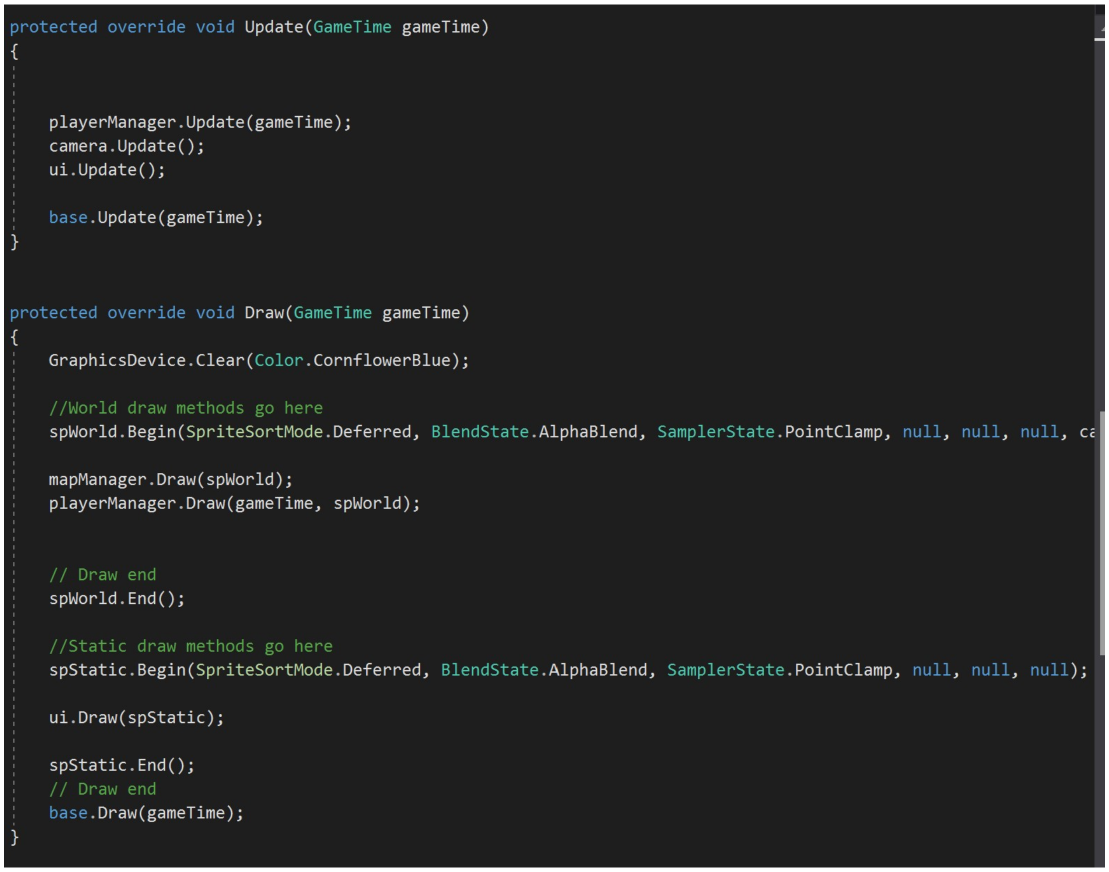
Game loop is a core method where the entire game runs. The idea of the game loop is to keep the program working and constantly check for inputs, changes and write outputs. An example of what might be located in the game loop would be an input method that checks if any keys were pressed by the player.

4.2 Update

Update method is a method that is called in the game loop. This method is always called by a main game loop. Almost every class will have the update method implemented so it can be called from the game loop.

4.3 Draw

Draw method will be implemented by sprites. These methods will take the animations and draw them on screen. These methods are called just like the update method except they are more specialized, created to just draw the textures.



```
protected override void Update(GameTime gameTime)
{


    playerManager.Update(gameTime);
    camera.Update();
    ui.Update();

    base.Update(gameTime);
}


protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    //World draw methods go here
    spWorld.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointClamp, null, null, null, ca

    mapManager.Draw(spWorld);
    playerManager.Draw(gameTime, spWorld);


    // Draw end
    spWorld.End();

    //Static draw methods go here
    spStatic.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointClamp, null, null, null);

    ui.Draw(spStatic);

    spStatic.End();
    // Draw end
    base.Draw(gameTime);
}
```

Figure 6: Image shows main game class where the game loop takes place.

5 Optimization

Optimization is very important in video games. Every time something is created it will take space in the machine. There are different tricks to prevent the drop in optimization. I have found two patterns that might be helpful: Object pool and spatial partition.

5.1 Object Pool

Object pool is a way of storing object in one place and reusing them as the game plays. The reason why I want to use object pool in my game is because if I keep creating new items or load the same textures for each enemy over and over again it will affect the game performance. To combat that I created a list of objects that will be referenced from other classes and reused as they see fit. The main example of this are the textures. When the game loads images, it stores them in a list and passes the list around. The game then creates objects using these textures. Therefore instead of constantly loading in objects, the game uses references to the same single file all the time.

5.2 Spatial Partition

Spatial partition is a way of determining the general location of objects. When a class keeps on checking the exact location of the objects and if they're in range every game tick, it will affect the performance of the game itself. This pattern creates a list of 'near' and 'far' objects in regards to the class, when the class checks the range and location it will pick from the 'near' list. After I created building system I noticed a noticeable drop in performance that cause the game to be almost unplayable. What was essentially happening was that the game was looping thought every possible tile, checking it it's a position where the player pressed on. To combat that I have used chunks that hold positions of different tiles. They way it works is that when the player presses on screen, the game loops through the chunks and checks if they contain the mouse position. Once the match was found it then takes the tiles, loops through them and check the exact position of the mouse press. This method reduced the number of looping items from 500 to 16 on each iteration.

6 Engines

In order to create a game first there has to be an engine where the program can be written in. It is essentially a building block that make the whole video game. Engines consist of few elements like: Graphics Device, Audio Device, Physics and Game Logic. On top of that most game engines provide certain tools like map creator or animation tools, that helps speed up creating of video games. There were several programs I've been investigating for my project and ultimately the engine I have chosen for the project is called Monogame.

6.1 Monogame

Monogame is not necessarily an engine as it doesn't provide any sort of advanced tools nor does it have any UI, it is a framework that entirely consists of lines of code and if the developer wishes to apply collision detection into the game they will have to implement it them self, it is also worth noting that the framework allows for making the game cross platform with OpenGL. Monogame implements XNA, which is a Microsoft environment that helps in creating video games. XNA provides basic tools that helps create game windows, inputs, draw/update methods, networking, etc.

The main reason why I chose Monogame was because I want to experience creating methods from the beginning, by creating animation, collision[1] and other classes, that would be otherwise included with other engines, I'm learning how they work and experience challenge. On top of that Monogame is a plugin for Visual Studio[2] and it also uses c, I'm familiar with both the software and language so I didn't have to learn it before I started the project.
I mentioned that Monogame does in fact have few tools, this is the list of the most useful tools implemented in Monogame:

- PipelineTool is a software that loads any textures, sounds, or any other content and encrypts it. This content can be in game with a SpriteBatch.

- SpriteBatch a class that draws sprites on screen, when called I can give it various effects like if the game accepts alpha, camera position etc...

- Vectors are a nice addiction to the framework. Vectors are used everywhere in the video games and it would complicate things if they weren't included in the engine.

- KeyboardState is a class that detects player input, I used this class to determine what to do witch each player input.

- Template start with what little Monogame provides, it does give the developers the template at the start. It's a simple code that consists of a constructor, initialization, load, unload, draw and update methods.

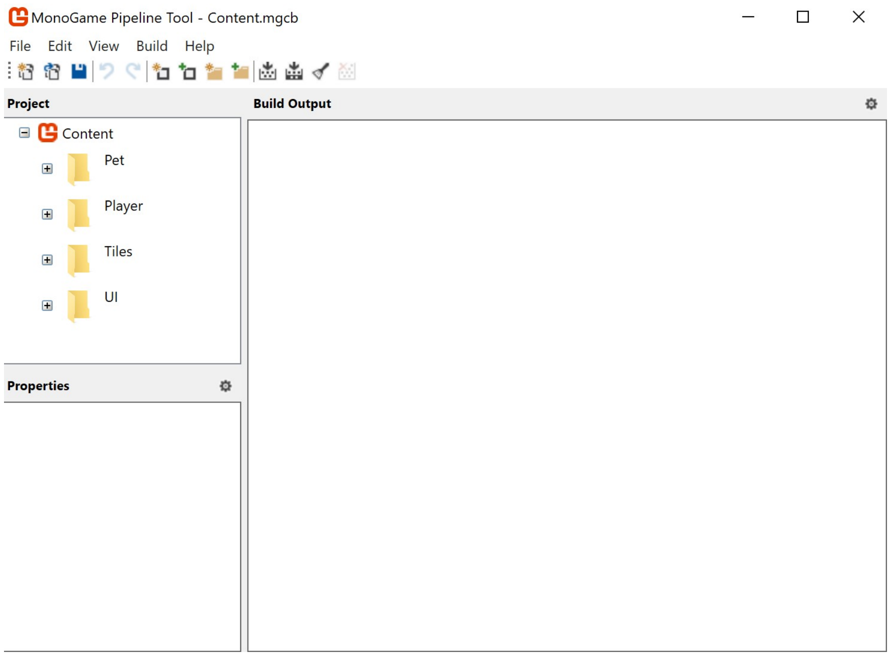
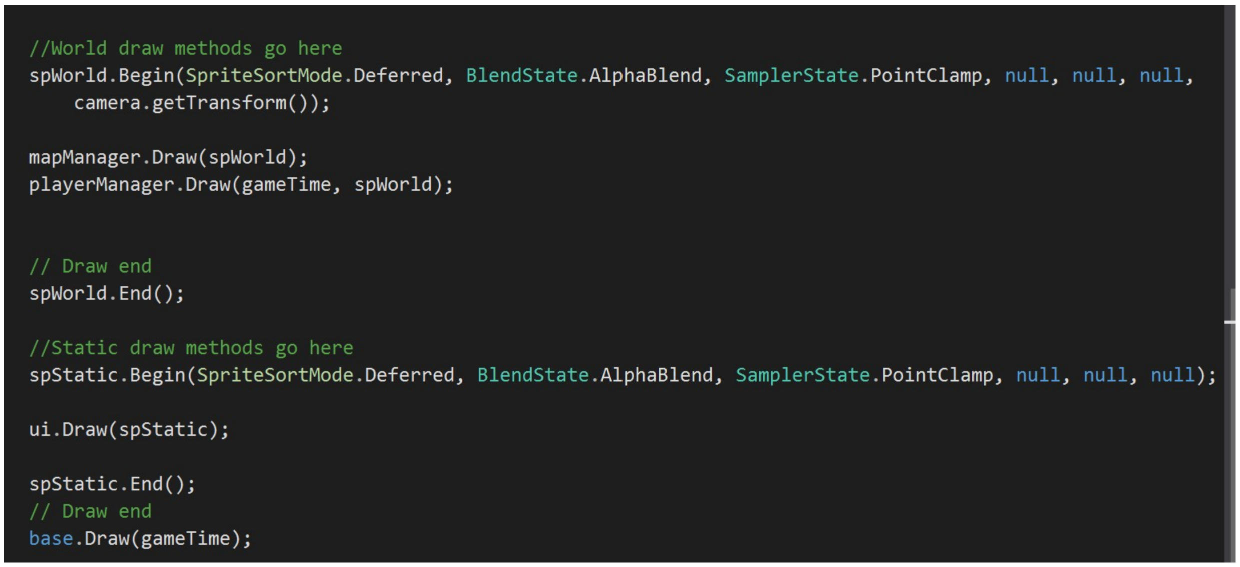


Figure 7: Monogame pipeline tool software.

6.2 Unity and Unreal Engine

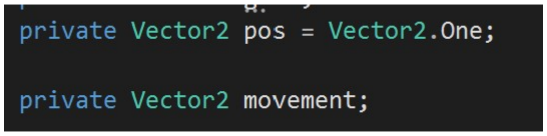
Unity and Unreal Engine are one of the most popular game engines, their UI is very useful, it contains multiple tools to use and there's a massive market for developers to buy and sell assets. Before I chose Monogame as my game engine I was considering Unity for how easy it is to create 2D/3D games. This engine can use different languages in a single program.

Unity and UE has a wide range of tools and on top of that some tools created by other developers can be purchased on their market. Here are notable default tools that I consider very useful:
- Animation tool allows for quick and easy creation of animated sprites. It also allows for splitting a single texture into parts.



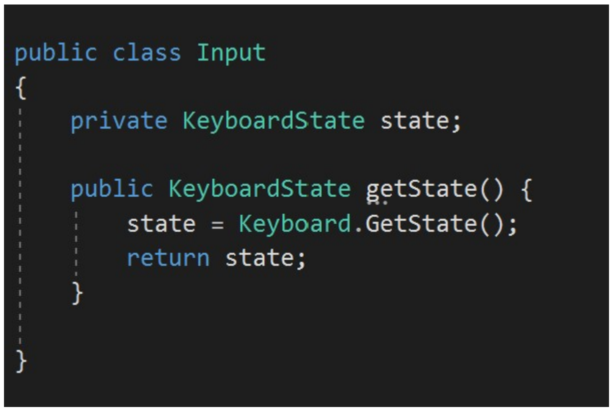
```
//World draw methods go here
spWorld.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointClamp, null, null, null,
    camera.getTransform());

mapManager.Draw(spWorld);
playerManager.Draw(gameTime, spWorld);


// Draw end
spWorld.End();

//Static draw methods go here
spStatic.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointClamp, null, null, null);

ui.Draw(spStatic);

spStatic.End();
// Draw end
base.Draw(gameTime);
```

Figure 8: SpriteBatch example code



```
private Vector2 pos = Vector2.One;

private Vector2 movement;
```

Figure 9: Example of vector code.



```
public class Input
{
    private KeyboardState state;

    public KeyboardState getState() {
        state = Keyboard.GetState();
        return state;
    }

}
```

Figure 10: Example of keyboardState.

- Shaders are a nice addiction in unity, they completely change the way materials look and can be dynamically changed during the game play.

- Hierarchy and scene help a lot, it allows quick access to an object and a scene can be arranged.

6.3 Game Maker

Game Maker is not powerful for 3D game but it's a very good engine for any 2D games. Game Maker uses it's own language that's similar to java and just like Unity and UE it hosts it's own marketplace. Game Maker strikes as the most easiest engine to learn and code in as animation, layers[3], making maps is as simplified as possible. It is also important to note that a lot of developers without prior knowledge learned how to code using this engine in a short amount of time.

7 Technical Concepts

7.1 Animation design

Since Monogame doesn't have animations tools I had to create animation class myself. I wanted to create a class that's modular and can be used by every texture atlas without further coding. In the end my animation of a sprite consists of three stages:
Stage 1: Load a texture and assign it to Animation class
Stage 2: Assign the Animation class to animation manager
Stage 3: Update the sprite and draw it every update

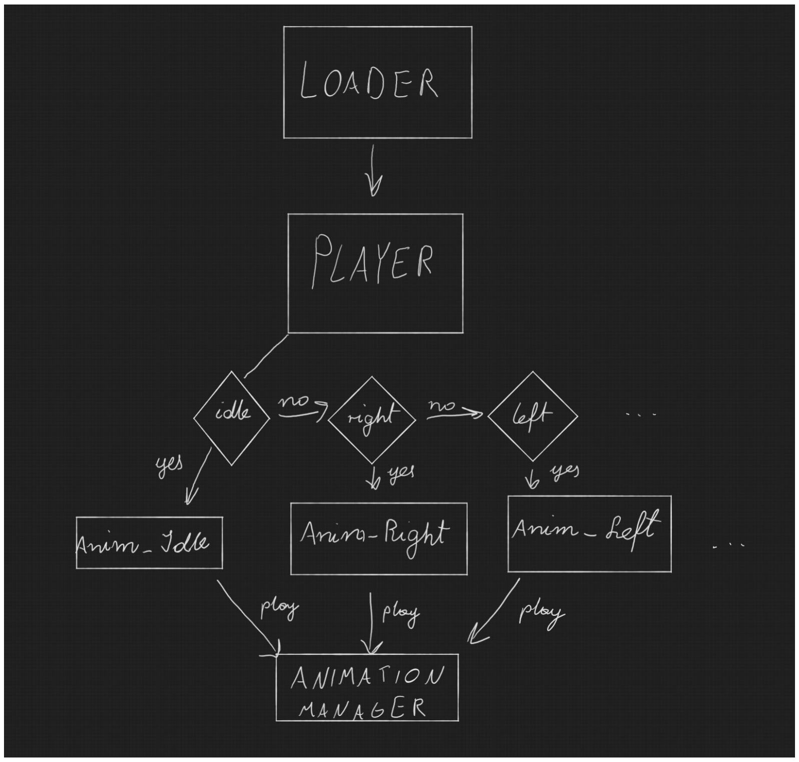
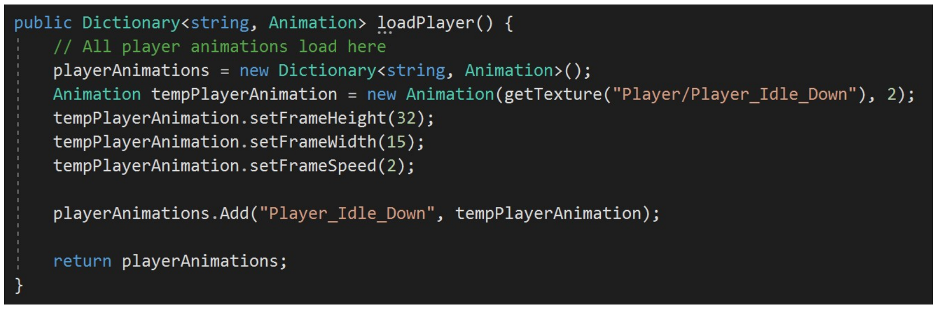


Figure 11: Part of the code from ContentLoader class. Method shows loading of the player animation.

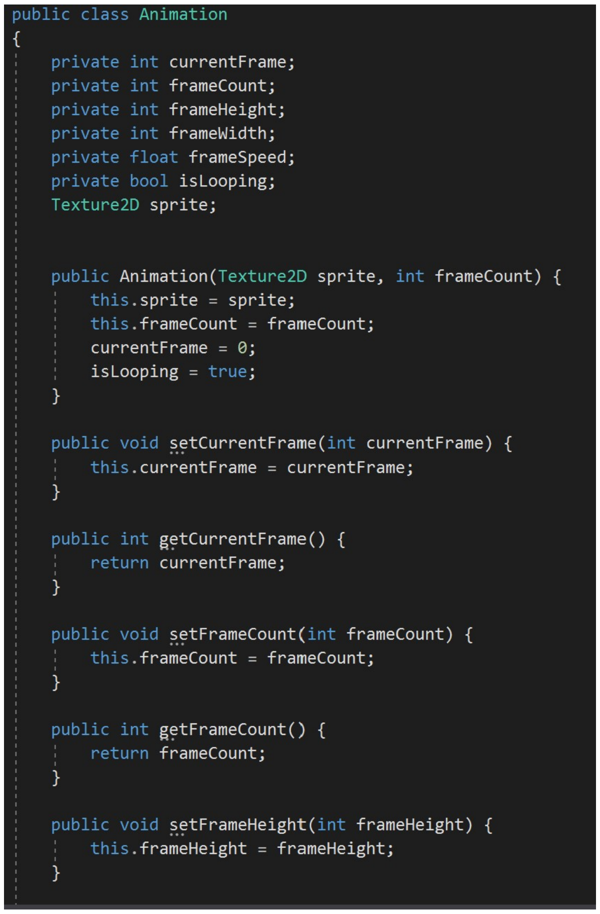
This figure shows the loader class. As the name suggests all content is loaded inside this particular class. In this example I am loading a player atlas texture and assigning it to animation class, I then assign values for the animation class, which will be discussed later. In the end animation is assigned to a dictionary with a string as a key.



```
public Dictionary<string, Animation> loadPlayer() {
    // All player animations load here
    playerAnimations = new Dictionary<string, Animation>();
    Animation tempPlayerAnimation = new Animation(getTexture("Player/Player_Idle_Down"), 2);
    tempPlayerAnimation.setFrameHeight(32);
    tempPlayerAnimation.setFrameWidth(15);
    tempPlayerAnimation.setFrameSpeed(2);

    playerAnimations.Add("Player_Idle_Down", tempPlayerAnimation);

    return playerAnimations;
}
```

Figure 12: Part of the code from ContentLoader class. Method shows loading of the player animation.

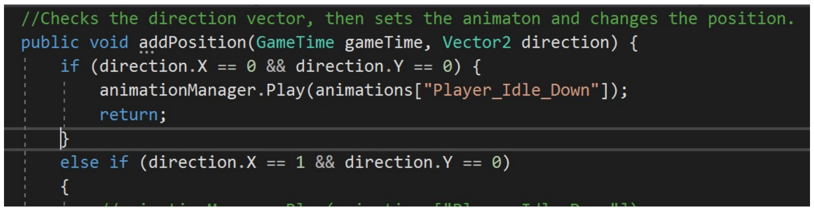
Each atlas[4] is assigned an animation class. The class holds values about the texture, the information here is crucial and a mistake will completely give different results. Frame variables tell the game what frame it currently showing, information about looping, speed of animation and what's the maximum frame count, it also holds information about the height and width of each frame. This class holds no complicated code as it is only about holding data.



```
public class Animation
{
    private int currentFrame;
    private int frameCount;
    private int frameHeight;
    private int frameWidth;
    private float frameSpeed;
    private bool isLooping;
    Texture2D sprite;


    public Animation(Texture2D sprite, int frameCount) {
        this.sprite = sprite;
        this.frameCount = frameCount;
        currentFrame = 0;
        isLooping = true;
    }

    public void setCurrentFrame(int currentFrame) {
        this.currentFrame = currentFrame;
    }

    public int getCurrentFrame() {
        return currentFrame;
    }

    public void setFrameCount(int frameCount) {
        this.frameCount = frameCount;
    }

    public int getFrameCount() {
        return frameCount;
    }

    public void setFrameHeight(int frameHeight) {
        this.frameHeight = frameHeight;
    }
```

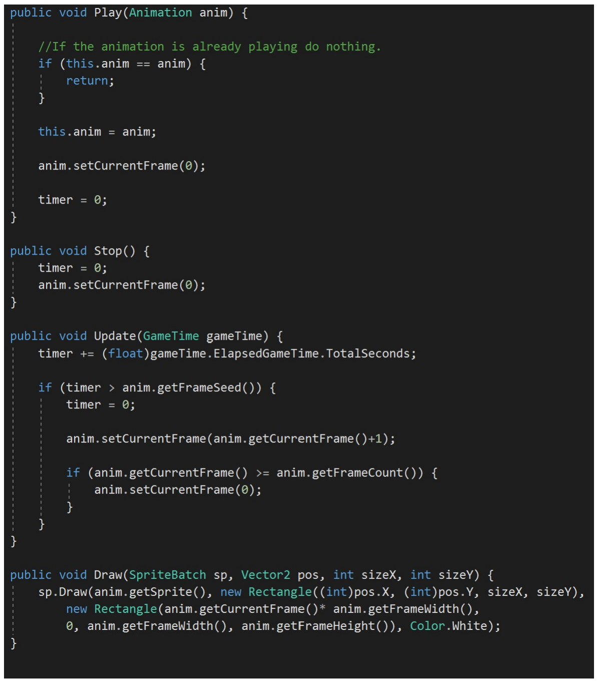
Figure 13: Image shows part of the Animation class.

Player class gets the dictionary with animations and then decides what animation to play. If the player class decides that idle animation should be played then it takes the animation from the list and passes it along to the animation manager. Animation manager now determines if the same animation already plays, then it updates the frame and lastly it draws the sprite on screen.



```
//Checks the direction vector, then sets the animaton and changes the position.
public void addPosition(GameTime gameTime, Vector2 direction) {
    if (direction.X == 0 && direction.Y == 0) {
        animationManager.Play(animations["Player_Idle_Down"]);
        return;
    }
    else if (direction.X == 1 && direction.Y == 0)
    {
```

Figure 14: Image shows part of the player class. This code is part of the movement method.



```
public void Play(Animation anim) {

    //If the animation is already playing do nothing.
    if (this.anim == anim) {
        return;
    }

    this.anim = anim;

    anim.setCurrentFrame(0);

    timer = 0;
}

public void Stop() {
    timer = 0;
    anim.setCurrentFrame(0);
}

public void Update(GameTime gameTime) {
    timer += (float)gameTime.ElapsedGameTime.TotalSeconds;

    if (timer > anim.getFrameSeed()) {
        timer = 0;

        anim.setCurrentFrame(anim.getCurrentFrame()+1);

        if (anim.getCurrentFrame() >= anim.getFrameCount()) {
            anim.setCurrentFrame(0);
        }
    }
}

public void Draw(SpriteBatch sp, Vector2 pos, int sizeX, int sizeY) {
    sp.Draw(anim.getSprite(), new Rectangle((int)pos.X, (int)pos.Y, sizeX, sizeY),
        new Rectangle(anim.getCurrentFrame()* anim.getFrameWidth(),
        0, anim.getFrameWidth(), anim.getFrameHeight()), Color.White);
}
```

Figure 15: Image is showing AnimationManager class.

7.2 What are colliders

Collision detection centers around shapes, known as colliders, interacting with each other. Collider in video games is a shape that detects collisions with other shapes by using mathematical equations. Colliders are widely used in almost all titles as they showcase physics of objects interacting with the world. Each video game come with it's own unique collision detection as there are different factors that can be used when calculating object interaction, such factors include: 2D or 3D environments, speed, angles, gravity and shapes. Usually colliders are used to hamper movement between objects such as a wall and a ball, but they can be also used to trigger actions like opening a door when player steps inside a collider.
In video game industry it is important to have an optimized methods that handle collisions as they can easily slow down the game. Game that check for 64 colliders around it will run smother than a game that constantly checks an entire map for collisions. Generally there is no best way to code collisions, some titles use chunks to check for collisions and others sophisticated trees that detect nearest objects (look at spatial partition section).

7.3 Colliders in Unity and Unreal Engine

Unity and Unreal Engine use already build in collision detection. These colliders are highly customizable. It is possible to change shape of a collider from default shapes (circle, rectangle, capsule), have their dimensions changed, change how they interact with environment, select different layers they can interact with and on top of that they have a feature that allows creation of colliders based on the object mesh (will mimic the shape of an object). It's a very easy tool to use and has small learning curve but it's very efficient.
When it comes to optimization of the collisions Unity and UE take static objects (objects that are part of the map like walls, ground or chairs) and combine their colliders together. It eliminates the need to cycle through hundreds of colliders each iteration, and thus it greatly speeds up collision process. There is also a large market in both engines where developers can buy pre-existing classes that handle advanced collisions and physics.

7.4 Colliders in Monogame

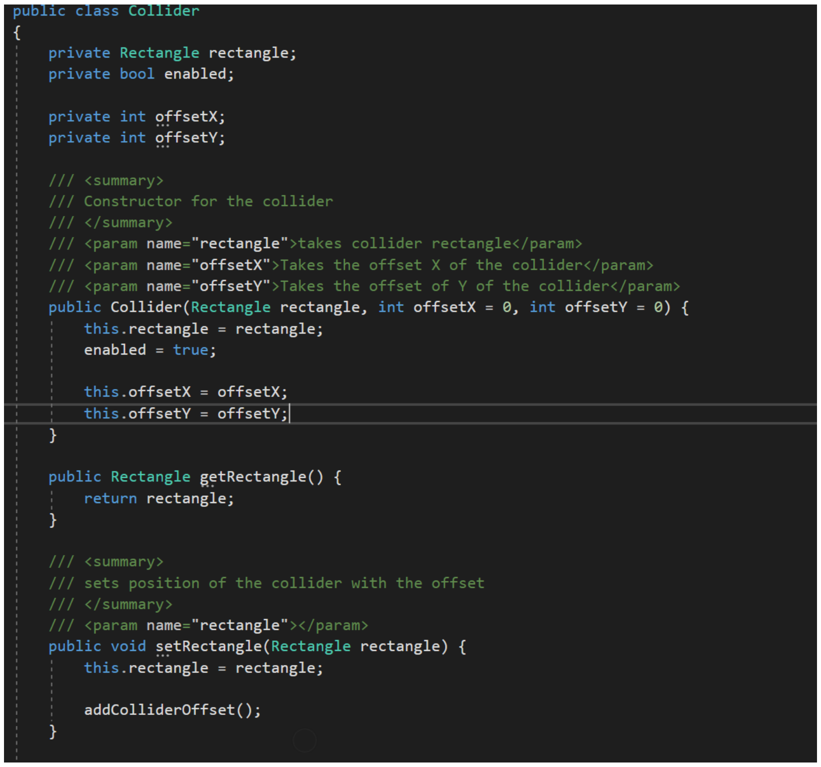
Monogame doesn't have build in tools that help with collision, but it comes with a simple method that allow for creation of rectangle collision. Monogame uses 'Rectangle1.Intersects(Rectangle2)' code to detect if there's a collision but it does not provide information on where the collision occurred and it's depth, the developer has to write code that check what side the collision happened,

where to stop the object hitting the collider and how to handle multiple collisions. It is possible to create collisions between different shapes but they have to be manually written.
It's also worth noting that Monogame doesn't have any optimization tools so each step has to be written by the developer. Although there is no support in regards to many features that come with game engines, there is a large community centered around Monogame that provide useful tips on how to handle collision optimization.

7.5 My implementation of collision

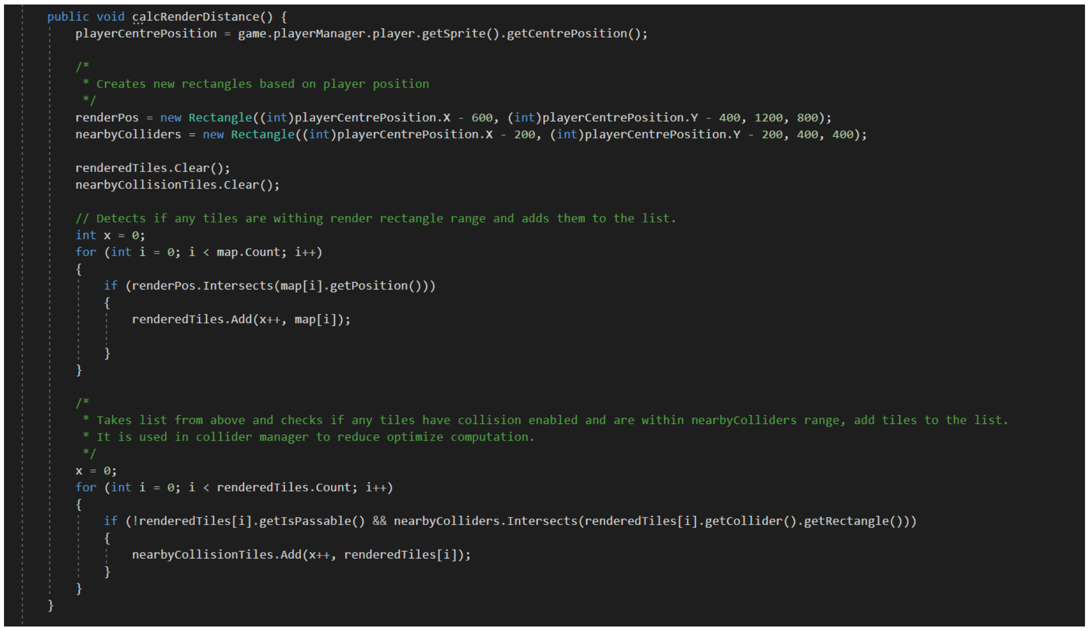
Implementing collision physics is always tricky, there were multiple factors that needed to be taken into consideration. Firstly, I had to create a collider class that had to be customizable. Each object might need different collider and it would get confusing if there were multiple versions of them. Secondly there had to be a manager that checks for collisions by taking sprites from different classes, something like centralized office that holds information about different colliders. Thirdly, I needed to implement spatial partition that takes localized colliders to optimize the game. If I were to check for every tile on the map it would impact the playability of my title. Fourthly and lastly, I required logic that handled the collisions themselves.

Collider class is straight forward, it's a simple class that holds variables for each sprite. The main features of the class is the Rectangle it holds and the X and Y offsets. When the sprite moves, it gives it's new position to the collider, witch then adds offsets to it. The offset exists because some sprites don't have the same sized colliders, an example of this would be player sprite; It's collider is placed on their feet so the top of the sprite can move onto the collider giving it the 3D illusion.



```
public class Collider
{
    private Rectangle rectangle;
    private bool enabled;

    private int offsetX;
    private int offsetY;

    /// <summary>
    /// Constructor for the collider
    /// </summary>
    /// <param name="rectangle">takes collider rectangle</param>
    /// <param name="offsetX">Takes the offset X of the collider</param>
    /// <param name="offsetY">Takes the offset of Y of the collider</param>
    public Collider(Rectangle rectangle, int offsetX = 0, int offsetY = 0) {
        this.rectangle = rectangle;
        enabled = true;

        this.offsetX = offsetX;
        this.offsetY = offsetY;
    }

    public Rectangle getRectangle() {
        return rectangle;
    }

    /// <summary>
    /// sets position of the collider with the offset
    /// </summary>
    /// <param name="rectangle"></param>
    public void setRectangle(Rectangle rectangle) {
        this.rectangle = rectangle;

        addColliderOffset();
    }
```

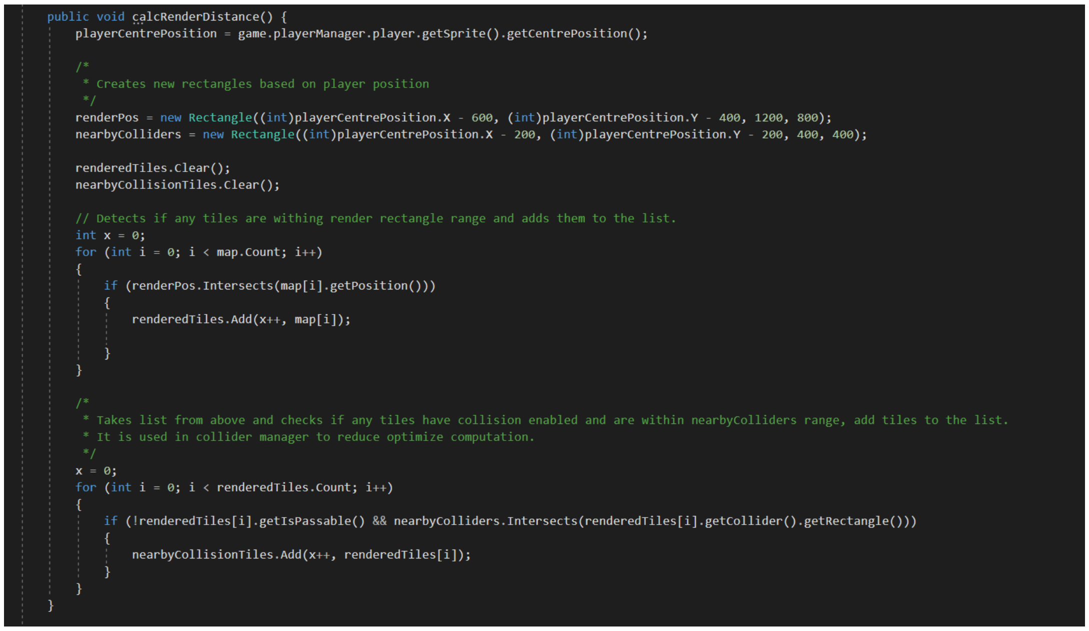
Figure 16: Code showing my implementation of collider class



```
public void calcRenderDistance() {
    playerCentrePosition = game.playerManager.player.getSprite().getCentrePosition();

    /*
     * Creates new rectangles based on player position
     */
    renderPos = new Rectangle((int)playerCentrePosition.X - 600, (int)playerCentrePosition.Y - 400, 1200, 800);
    nearbyColliders = new Rectangle((int)playerCentrePosition.X - 200, (int)playerCentrePosition.Y - 200, 400, 400);

    renderedTiles.Clear();
    nearbyCollisionTiles.Clear();

    // Detects if any tiles are withing render rectangle range and adds them to the list.
    int x = 0;
    for (int i = 0; i < map.Count; i++)
    {
        if (renderPos.Intersects(map[i].getPosition()))
        {
            renderedTiles.Add(x++, map[i]);

        }
    }

    /*
     * Takes list from above and checks if any tiles have collision enabled and are within nearbyColliders range, add tiles to the list.
     * It is used in collider manager to reduce optimize computation.
     */
    x = 0;
    for (int i = 0; i < renderedTiles.Count; i++)
    {
        if (!renderedTiles[i].getIsPassable() && nearbyColliders.Intersects(renderedTiles[i].getCollider().getRectangle()))
        {
            nearbyCollisionTiles.Add(x++, renderedTiles[i]);
        }
    }
}
```

Figure 17: Code showing second part of the collider class

Spatial partition exists to optimize the game. The main premise of it, is to keep track of nearby colliders in a list and use them to calculate collisions. My implementation of spatial partition works by creating a large rectangle witch then checks if any tiles with collision are within it's boundaries and adds it to the list, which is then passed over to the collider manager.



```
public void calcRenderDistance() {
    playerCentrePosition = game.playerManager.player.getSprite().getCentrePosition();

    /*
     * Creates new rectangles based on player position
     */
    renderPos = new Rectangle((int)playerCentrePosition.X - 600, (int)playerCentrePosition.Y - 400, 1200, 800);
    nearbyColliders = new Rectangle((int)playerCentrePosition.X - 200, (int)playerCentrePosition.Y - 200, 400, 400);

    renderedTiles.Clear();
    nearbyCollisionTiles.Clear();

    // Detects if any tiles are withing render rectangle range and adds them to the list.
    int x = 0;
    for (int i = 0; i < map.Count; i++)
    {
        if (renderPos.Intersects(map[i].getPosition()))
        {
            renderedTiles.Add(x++, map[i]);

        }
    }

    /*
     * Takes list from above and checks if any tiles have collision enabled and are within nearbyColliders range, add tiles to the list.
     * It is used in collider manager to reduce optimize computation.
     */
    x = 0;
    for (int i = 0; i < renderedTiles.Count; i++)
    {
        if (!renderedTiles[i].getIsPassable() && nearbyColliders.Intersects(renderedTiles[i].getCollider().getRectangle()))
        {
            nearbyCollisionTiles.Add(x++, renderedTiles[i]);
        }
    }
}
```

Figure 18: Code showing my implementation of spatial partition. Firstly rectangle1 checks for tiles within rendering distance, secondly rectangle2 checks if any tiles have colliders

Collider Manager and collision logic is the main focus of the collision detection. When the player moves it passes the sprite's new and old position, movement vector and the collider to the collider manager. Firstly, the manager moves the player and checks if movement occurred on the X axis. Secondly, it loops through the list of nearby colliders to check if any colliders intersected with each other. Thirdly, it determines what side of the tile collision occurred by using collider centre position and lastly, it calculates the depth of the collision and applies the difference to the revised position (adds or subtracts difference from the position). This process is then repeated for Y axis. If a collision is detected on X but not Y axis, the player will still move up or down while maintaining the same X position. Revised position is then returned back to the player class.

depth of the collision is calculated based on the side. For example, if the player hits the tile from the left, the game will take it's right most point (X + width) and subtracts it with player collider X point. Another example where the player hits the tile from the right (X + width), the game will take player collider right most point and subtract it with tile collider X point.

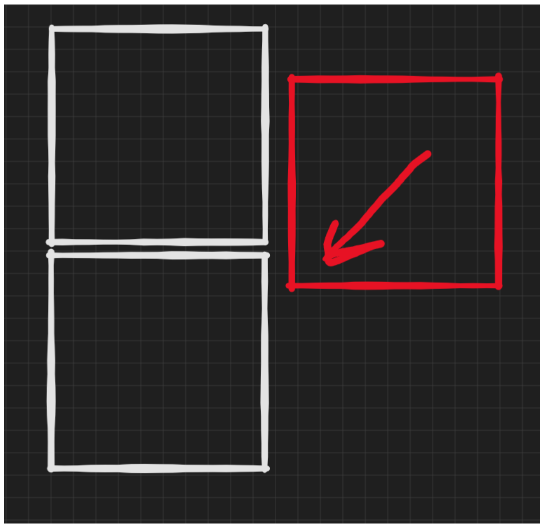


Figure 19: Representation of movement, red collider moves diagonally towards two colliders

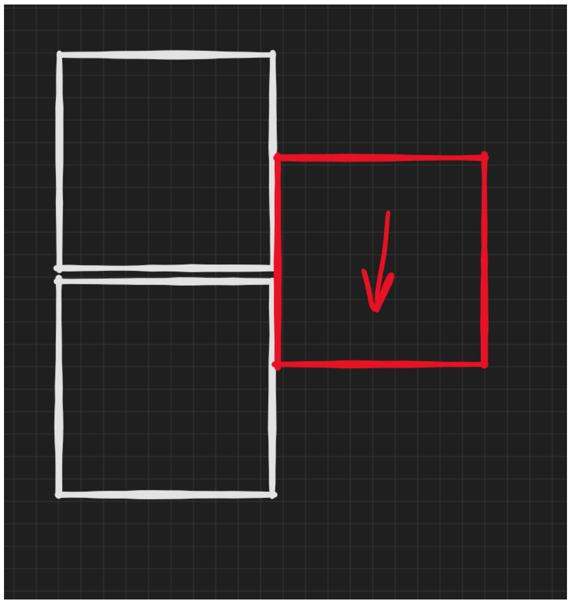
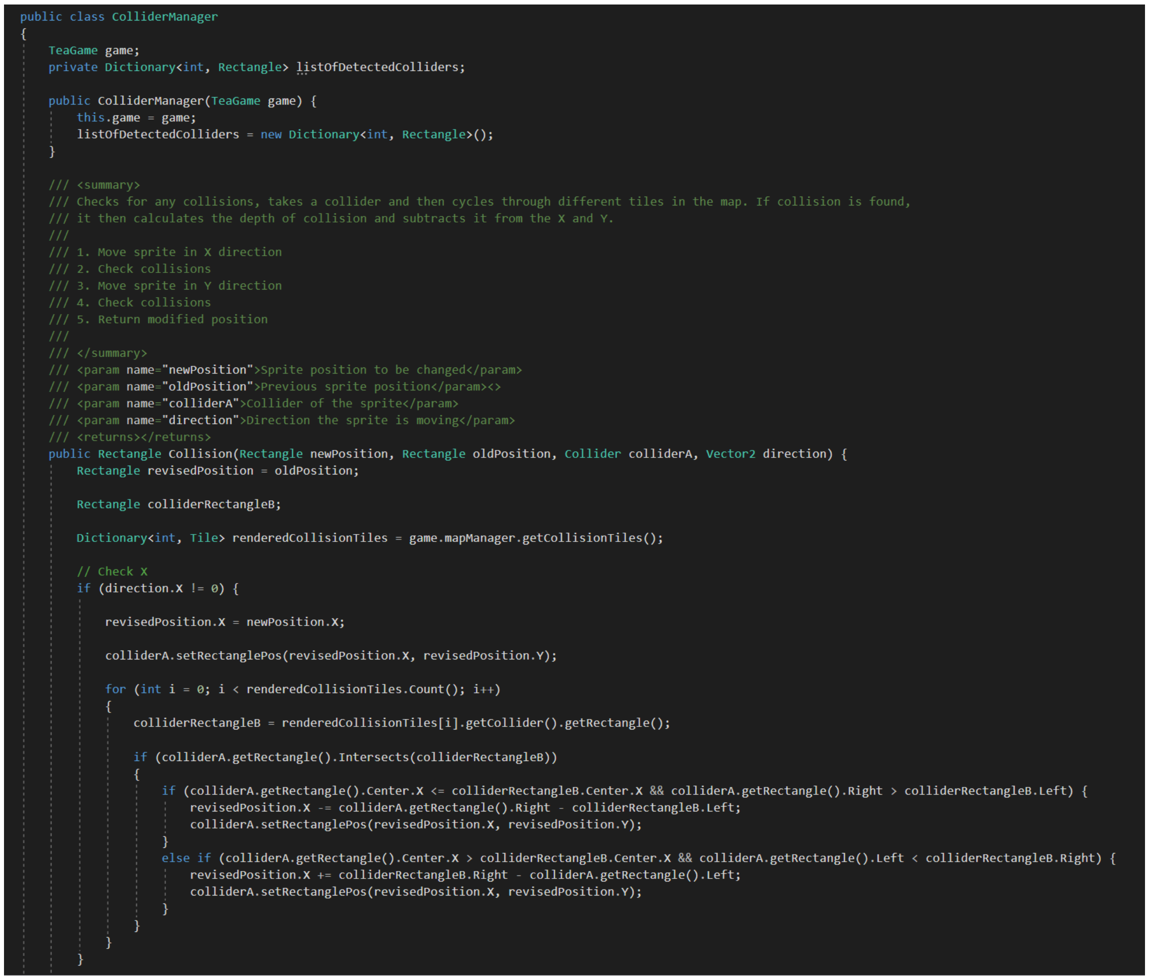
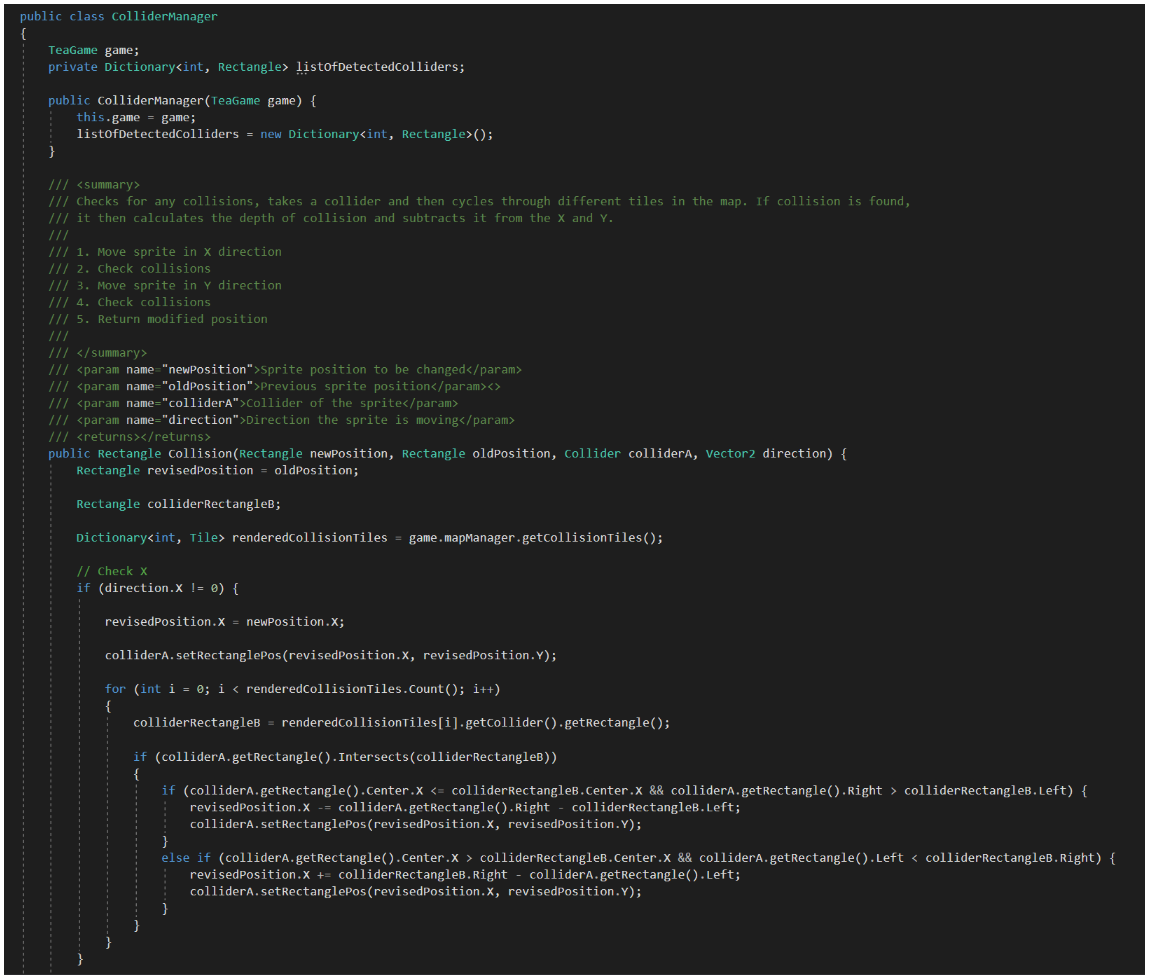


Figure 20: Representation of collision, red collider cannot move to the side but it can still maintain it's Y movement



```
public class ColliderManager
{
    TeaGame game;
    private Dictionary<int, Rectangle> listOfDetectedColliders;

    public ColliderManager(TeaGame game) {
        this.game = game;
        listOfDetectedColliders = new Dictionary<int, Rectangle>();
    }

    /// <summary>
    /// Checks for any collisions, takes a collider and then cycles through different tiles in the map. If collision is found,
    /// it then calculates the depth of collision and subtracts it from the X and Y.
    ///
    /// 1. Move sprite in X direction
    /// 2. Check collisions
    /// 3. Move sprite in Y direction
    /// 4. Check collisions
    /// 5. Return modified position
    ///
    /// </summary>
    /// <param name="newPosition">Sprite position to be changed</param>
    /// <param name="oldPosition">Previous sprite position</param><>
    /// <param name="colliderA">Collider of the sprite</param>
    /// <param name="direction">Direction the sprite is moving</param>
    /// <returns></returns>
    public Rectangle Collision(Rectangle newPosition, Rectangle oldPosition, Collider colliderA, Vector2 direction) {
        Rectangle revisedPosition = oldPosition;

        Rectangle colliderRectangleB;

        Dictionary<int, Tile> renderedCollisionTiles = game.mapManager.getCollisionTiles();

        // Check X
        if (direction.X != 0) {

            revisedPosition.X = newPosition.X;

            colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);

            for (int i = 0; i < renderedCollisionTiles.Count(); i++)
            {
                colliderRectangleB = renderedCollisionTiles[i].getCollider().getRectangle();

                if (colliderA.getRectangle().Intersects(colliderRectangleB))
                {
                    if (colliderA.getRectangle().Center.X <= colliderRectangleB.Center.X && colliderA.getRectangle().Right > colliderRectangleB.Left) {
                        revisedPosition.X -= colliderA.getRectangle().Right - colliderRectangleB.Left;
                        colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);
                    }
                    else if (colliderA.getRectangle().Center.X > colliderRectangleB.Center.X && colliderA.getRectangle().Left < colliderRectangleB.Right) {
                        revisedPosition.X += colliderRectangleB.Right - colliderA.getRectangle().Left;
                        colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);
                    }
                }
            }
        }
```

Figure 21: Code showing my implementation of collider manager class



```
public class ColliderManager
{
    TeaGame game;
    private Dictionary<int, Rectangle> listOfDetectedColliders;

    public ColliderManager(TeaGame game) {
        this.game = game;
        listOfDetectedColliders = new Dictionary<int, Rectangle>();
    }

    /// <summary>
    /// Checks for any collisions, takes a collider and then cycles through different tiles in the map. If collision is found,
    /// it then calculates the depth of collision and subtracts it from the X and Y.
    ///
    /// 1. Move sprite in X direction
    /// 2. Check collisions
    /// 3. Move sprite in Y direction
    /// 4. Check collisions
    /// 5. Return modified position
    ///
    /// </summary>
    /// <param name="newPosition">Sprite position to be changed</param>
    /// <param name="oldPosition">Previous sprite position</param><>
    /// <param name="colliderA">Collider of the sprite</param>
    /// <param name="direction">Direction the sprite is moving</param>
    /// <returns></returns>
    public Rectangle Collision(Rectangle newPosition, Rectangle oldPosition, Collider colliderA, Vector2 direction) {
        Rectangle revisedPosition = oldPosition;

        Rectangle colliderRectangleB;

        Dictionary<int, Tile> renderedCollisionTiles = game.mapManager.getCollisionTiles();

        // Check X
        if (direction.X != 0) {

            revisedPosition.X = newPosition.X;

            colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);

            for (int i = 0; i < renderedCollisionTiles.Count(); i++)
            {
                colliderRectangleB = renderedCollisionTiles[i].getCollider().getRectangle();

                if (colliderA.getRectangle().Intersects(colliderRectangleB))
                {
                    if (colliderA.getRectangle().Center.X <= colliderRectangleB.Center.X && colliderA.getRectangle().Right > colliderRectangleB.Left) {
                        revisedPosition.X -= colliderA.getRectangle().Right - colliderRectangleB.Left;
                        colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);
                    }
                    else if (colliderA.getRectangle().Center.X > colliderRectangleB.Center.X && colliderA.getRectangle().Left < colliderRectangleB.Right) {
                        revisedPosition.X += colliderRectangleB.Right - colliderA.getRectangle().Left;
                        colliderA.setRectanglePos(revisedPosition.X, revisedPosition.Y);
                    }
                }
            }
        }
```

Figure 22: second part of collider manager code

7.6 Depth Draw

In the early stages of my project all sprites had fixed drawing order witch was making sprites permanently above or below certain sprites. Monogame has a feature that involves layers but it's limited as new objects are constantly created in my game and the layer number has to be updated each loop for every existing object, so therefore it wasn't a perfect solution for my game.

7.7 My implementation of 3D space in 2D space

I thought a little bit how to handle this problem and ultimately I have created a class that takes all dynamic sprites like NPCs, items and the player, calculates their lowest Y positions and based on that it sorts sprites in the list to create an order in which the game drawer will print the sprites on screen. So I have created a layering system or a 3D plane in a 2D game. So as an example if a tree has a Y coordinate of 10 and the player 11, the tree will be drawn first on screen and then the player second giving the impression of 3D space.

7.8 Optimization of depth draw

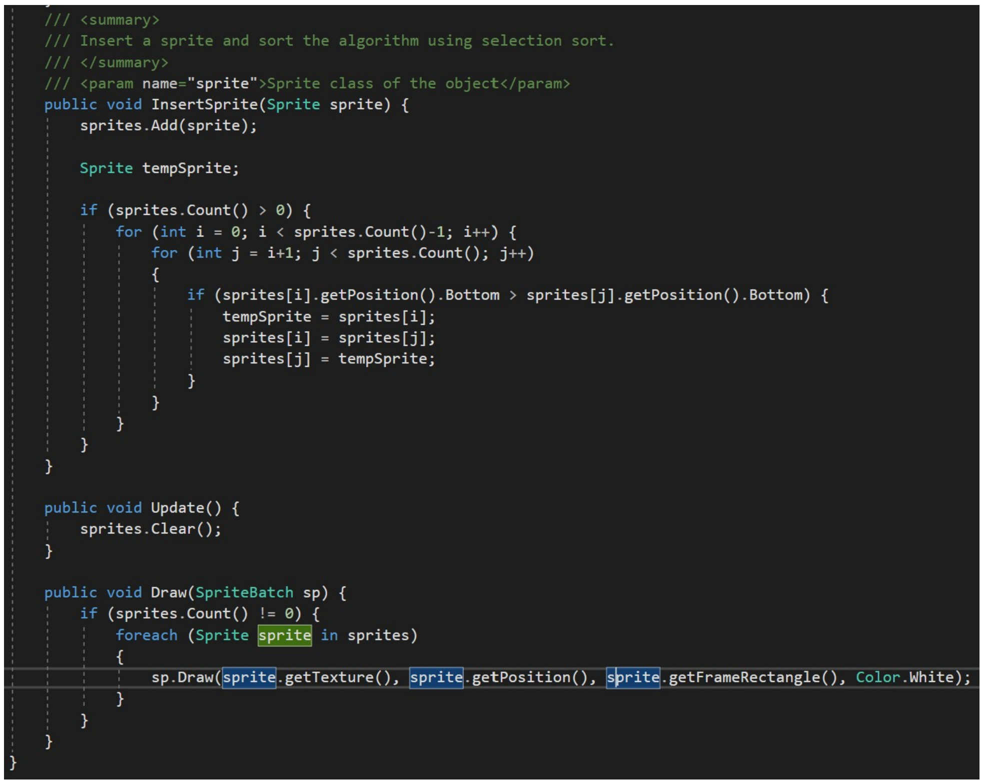
The issue of depth draw is that if mismanaged it can become a big burden on the CPU if there are too many calculations. Thankfully I already had a script that takes camera position and creates a list of rendered object within the screen that I used for collision detection, so I reused that so get only a specific number of sprites and run calculations on them.

In the image above I have shown a simple sorting algorithm that I wrote. Once a class inserts a sprite into the method, it runs from the beginning and compares Y's of sprites, and ultimately inserts the sprite into a correct location. What's left for the code is to draw the list since it's already ordered.

I was thinking about using different sorting algorithms like 'selection sort' or 'merge sort' but 'insertion sort' was proven to be quick enough for the amount of items in the scene therefore I left it as it is.

7.9 Quests and tree nodes

Quests are the most important part of RPG games. They are usually given by NPCs or given by the game itself. When players think of quests they imagine rewards and playable content, therefore I decided to include them in my project.



```
    /// <summary>
    /// Insert a sprite and sort the algorithm using selection sort.
    /// </summary>
    /// <param name="sprite">Sprite class of the object</param>
    public void InsertSprite(Sprite sprite) {
        sprites.Add(sprite);

        Sprite tempSprite;

        if (sprites.Count() > 0) {
            for (int i = 0; i < sprites.Count()-1; i++) {
                for (int j = i+1; j < sprites.Count(); j++)
                {
                    if (sprites[i].getPosition().Bottom > sprites[j].getPosition().Bottom) {
                        tempSprite = sprites[i];
                        sprites[i] = sprites[j];
                        sprites[j] = tempSprite;
                    }
                }
            }
        }
    }

    public void Update() {
        sprites.Clear();
    }

    public void Draw(SpriteBatch sp) {
        if (sprites.Count() != 0) {
            foreach (Sprite sprite in sprites)
            {
                sp.Draw(sprite.getTexture(), sprite.getPosition(), sprite.getFrameRectangle(), Color.White);
            }
        }
    }
}
```

Figure 23: Algorithm code for the depth draw of the sprites

First of I wanted to create a simple way for assigning quests to the NPCs. When I started development in the quest class I immediately thought of using tree nodes for my advantage. My first iteration of quest system consisted of a single node that was used both for the player and NPC dialogue. It turned out to be over-complicated so I switched it around and created two nodes, one for the player and the other one for the NPC. NPC node hods a list of player responses nodes, and then those player nodes will go to ai node again.

Because it would take a long time to script each quest I decided to create a file loader that would take data in the form of strings, create new quest and assign it to the NPC. When writing a new quest in a text file there are several rules that need to be followed but ultimately the process is faster than manually writing the code. I use ':' to separate a command from the content so for example 'QUEST NAME:Apples' will be read as an instruction to assign new name to the quest, using numbers '0.1.1...' I create new tree routes and I can add requirements and rewards to the quest as well. It is very useful system and

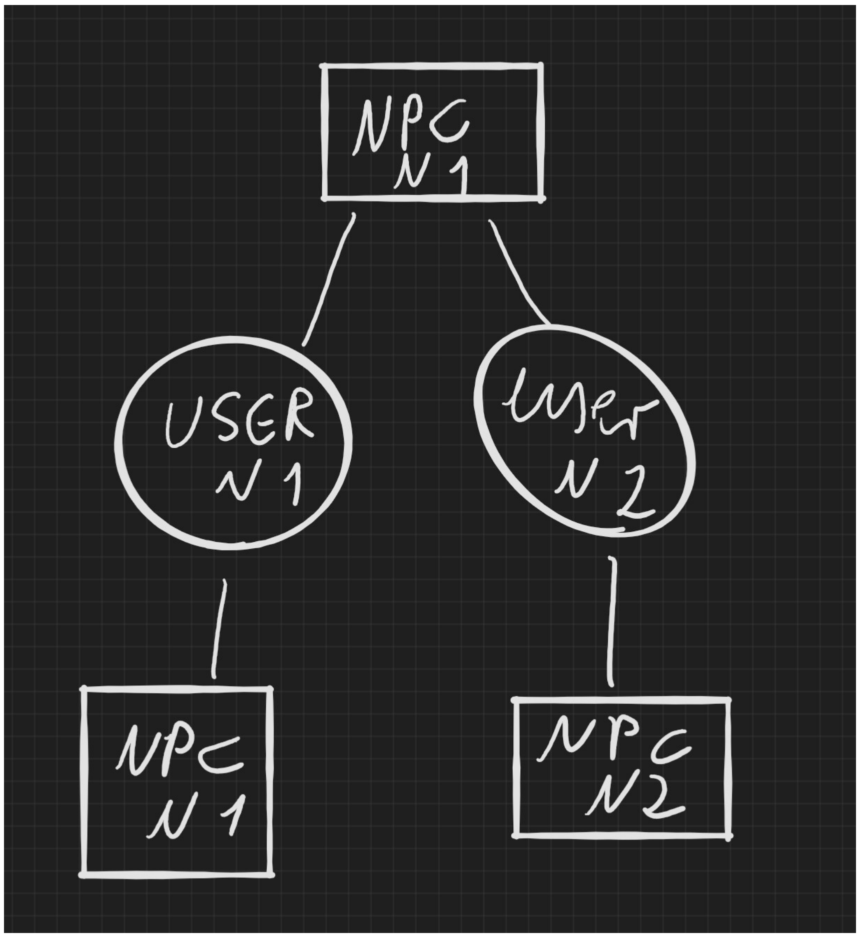
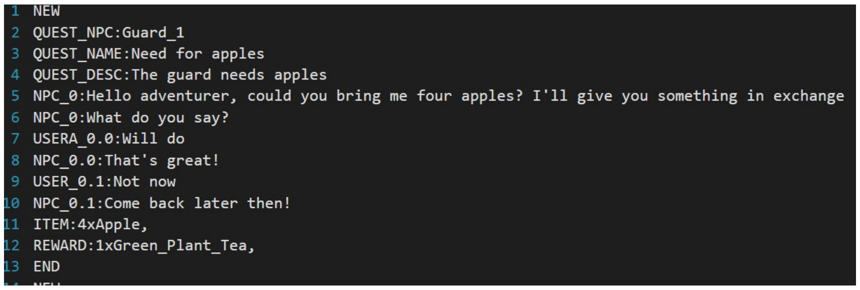


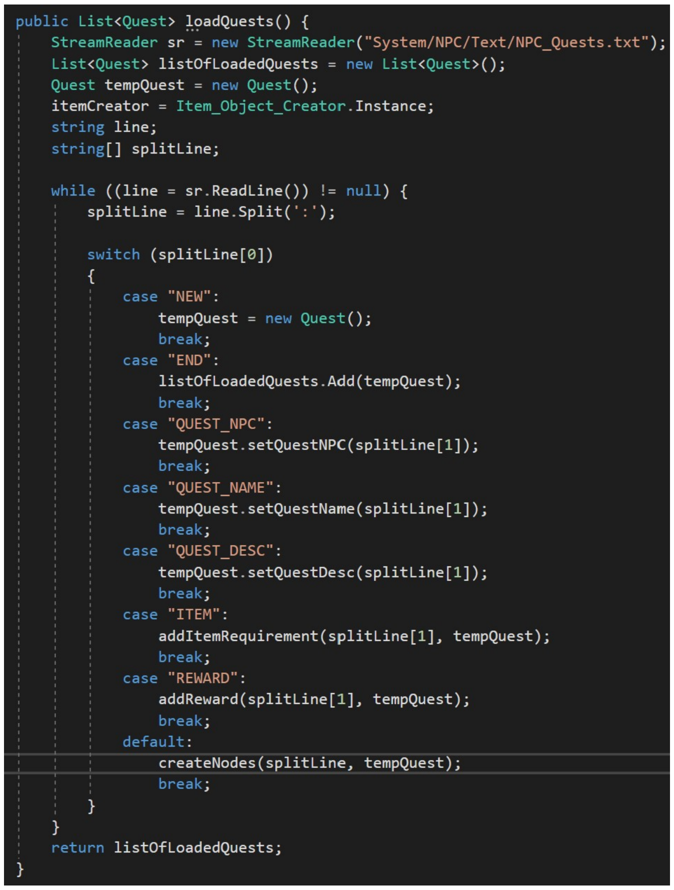
Figure 24: Structure of nodes in the last update

I'm happy how it turned out.



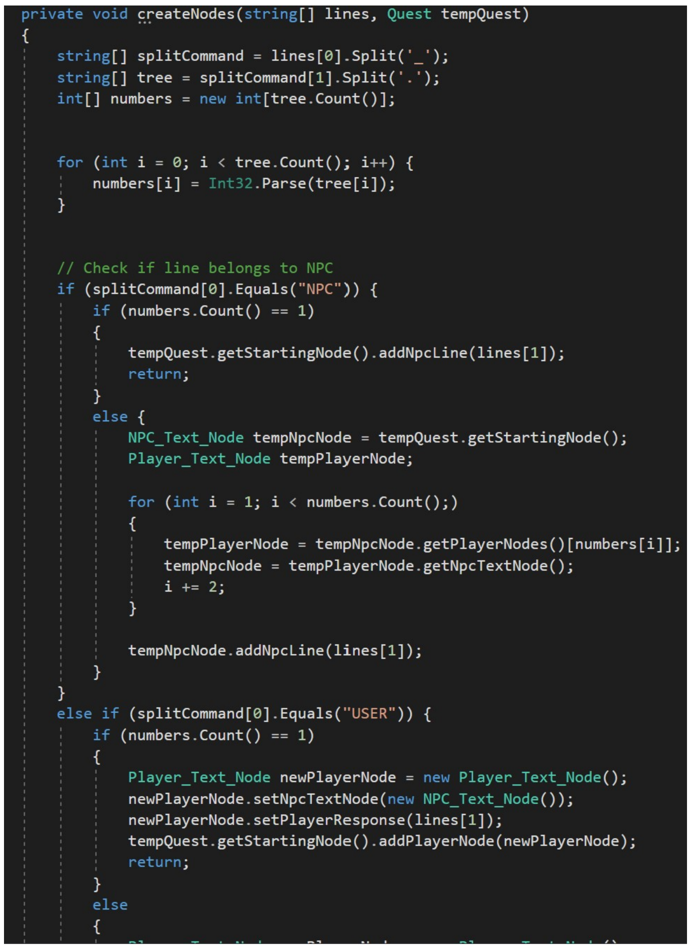
```
NEW
QUEST_NPC:Guard_1
QUEST_NAME:Need for apples
QUEST_DESC:The guard needs apples
NPC_0:Hello adventurer, could you bring me four apples? I'll give you something in exchange
NPC_0:What do you say?
USERA_0.0:Will do
NPC_0.0:That's great!
USER_0.1:Not now
NPC_0.1:Come back later then!
ITEM:4xApple,
REWARD:1xGreen_Plant_Tea,
END
```

Figure 25: Text folder that is used to create quests. ’NEW’ and ’END’ start and end creation of a quest



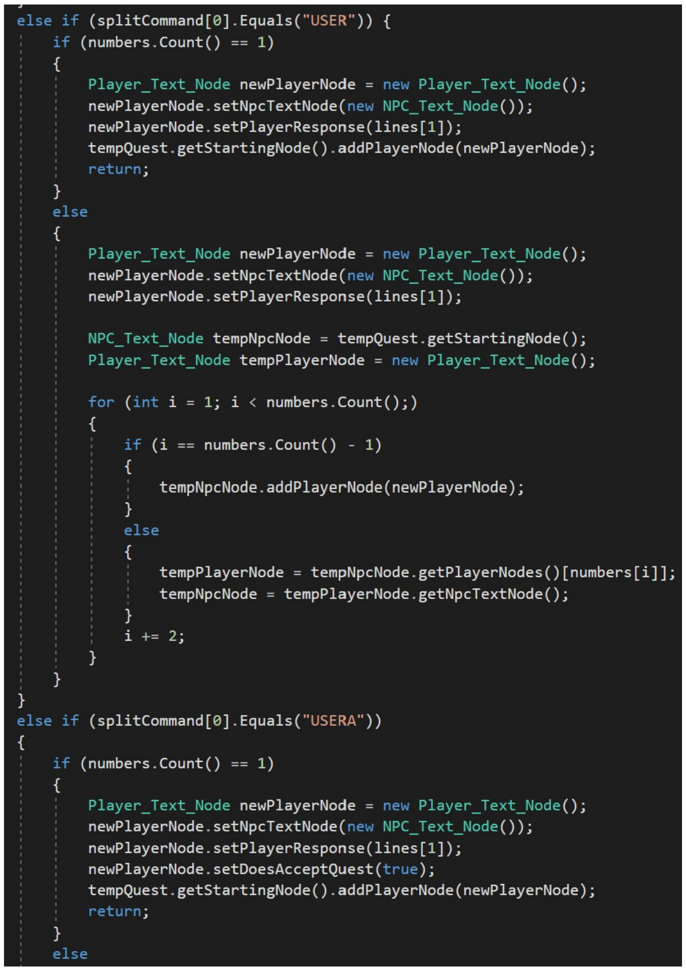
```
public List<Quest> loadQuests() {
    StreamReader sr = new StreamReader("System/NPC/Text/NPC_Quests.txt");
    List<Quest> listOfLoadedQuests = new List<Quest>();
    Quest tempQuest = new Quest();
    itemCreator = Item_Object_Creator.Instance;
    string line;
    string[] splitLine;

    while ((line = sr.ReadLine()) != null) {
        splitLine = line.Split(':');

        switch (splitLine[0])
        {
            case "NEW":
                tempQuest = new Quest();
                break;
            case "END":
                listOfLoadedQuests.Add(tempQuest);
                break;
            case "QUEST_NPC":
                tempQuest.setQuestNPC(splitLine[1]);
                break;
            case "QUEST_NAME":
                tempQuest.setQuestName(splitLine[1]);
                break;
            case "QUEST_DESC":
                tempQuest.setQuestDesc(splitLine[1]);
                break;
            case "ITEM":
                addItemRequirement(splitLine[1], tempQuest);
                break;
            case "REWARD":
                addReward(splitLine[1], tempQuest);
                break;
            default:
                createNodes(splitLine, tempQuest);
                break;
        }
    }
    return listOfLoadedQuests;
}
```

Figure 26: First part of the quest reader script



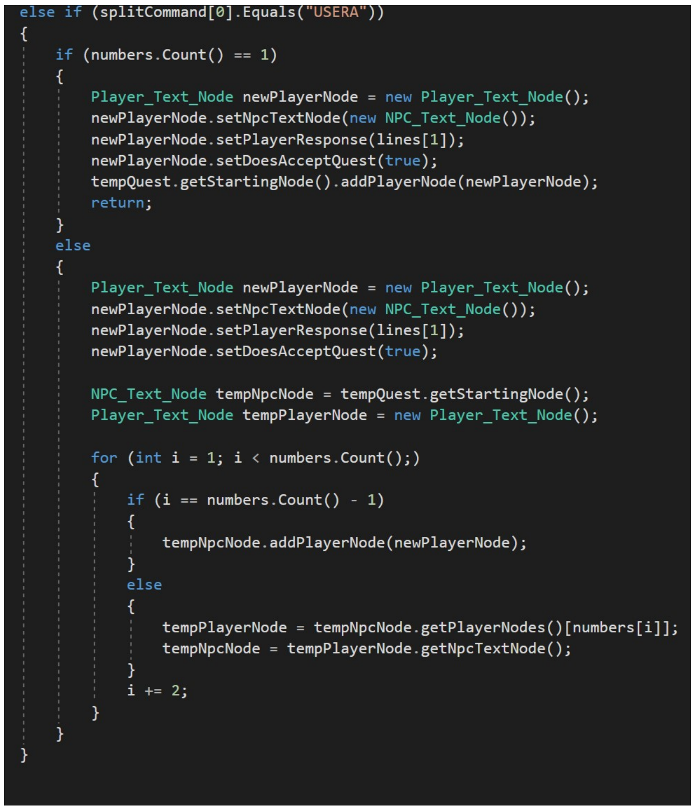
```
private void createNodes(string[] lines, Quest tempQuest)
{
    string[] splitCommand = lines[0].Split('_');
    string[] tree = splitCommand[1].Split('.');
    int[] numbers = new int[tree.Count()];


    for (int i = 0; i < tree.Count(); i++) {
        numbers[i] = Int32.Parse(tree[i]);
    }


    // Check if line belongs to NPC
    if (splitCommand[0].Equals("NPC")) {
        if (numbers.Count() == 1)
        {
            tempQuest.getStartingNode().addNpcLine(lines[1]);
            return;
        }
        else {
            NPC_Text_Node tempNpcNode = tempQuest.getStartingNode();
            Player_Text_Node tempPlayerNode;

            for (int i = 1; i < numbers.Count();)
            {
                tempPlayerNode = tempNpcNode.getPlayerNodes()[numbers[i]];
                tempNpcNode = tempPlayerNode.getNpcTextNode();
                i += 2;
            }

            tempNpcNode.addNpcLine(lines[1]);
        }
    }
    else if (splitCommand[0].Equals("USER")) {
        if (numbers.Count() == 1)
        {
            Player_Text_Node newPlayerNode = new Player_Text_Node();
            newPlayerNode.setNpcTextNode(new NPC_Text_Node());
            newPlayerNode.setPlayerResponse(lines[1]);
            tempQuest.getStartingNode().addPlayerNode(newPlayerNode);
            return;
        }
        else
        {
```

Figure 27: Second part of the quest reader script



```
else if (splitCommand[0].Equals("USER")) {
    if (numbers.Count() == 1)
    {
        Player_Text_Node newPlayerNode = new Player_Text_Node();
        newPlayerNode.setNpcTextNode(new NPC_Text_Node());
        newPlayerNode.setPlayerResponse(lines[1]);
        tempQuest.getStartingNode().addPlayerNode(newPlayerNode);
        return;
    }
    else
    {
        Player_Text_Node newPlayerNode = new Player_Text_Node();
        newPlayerNode.setNpcTextNode(new NPC_Text_Node());
        newPlayerNode.setPlayerResponse(lines[1]);

        NPC_Text_Node tempNpcNode = tempQuest.getStartingNode();
        Player_Text_Node tempPlayerNode = new Player_Text_Node();

        for (int i = 1; i < numbers.Count();)
        {
            if (i == numbers.Count() - 1)
            {
                tempNpcNode.addPlayerNode(newPlayerNode);
            }
            else
            {
                tempPlayerNode = tempNpcNode.getPlayerNodes()[numbers[i]];
                tempNpcNode = tempPlayerNode.getNpcTextNode();
            }
            i += 2;
        }
    }
}
else if (splitCommand[0].Equals("USERA"))
{
    if (numbers.Count() == 1)
    {
        Player_Text_Node newPlayerNode = new Player_Text_Node();
        newPlayerNode.setNpcTextNode(new NPC_Text_Node());
        newPlayerNode.setPlayerResponse(lines[1]);
        newPlayerNode.setDoesAcceptQuest(true);
        tempQuest.getStartingNode().addPlayerNode(newPlayerNode);
        return;
    }
    else
```

Figure 28: Third part of the quest reader script



```
else if (splitCommand[0].Equals("USERA"))
{
    if (numbers.Count() == 1)
    {
        Player_Text_Node newPlayerNode = new Player_Text_Node();
        newPlayerNode.setNpcTextNode(new NPC_Text_Node());
        newPlayerNode.setPlayerResponse(lines[1]);
        newPlayerNode.setDoesAcceptQuest(true);
        tempQuest.getStartingNode().addPlayerNode(newPlayerNode);
        return;
    }
    else
    {
        Player_Text_Node newPlayerNode = new Player_Text_Node();
        newPlayerNode.setNpcTextNode(new NPC_Text_Node());
        newPlayerNode.setPlayerResponse(lines[1]);
        newPlayerNode.setDoesAcceptQuest(true);

        NPC_Text_Node tempNpcNode = tempQuest.getStartingNode();
        Player_Text_Node tempPlayerNode = new Player_Text_Node();

        for (int i = 1; i < numbers.Count();)
        {
            if (i == numbers.Count() - 1)
            {
                tempNpcNode.addPlayerNode(newPlayerNode);
            }
            else
            {
                tempPlayerNode = tempNpcNode.getPlayerNodes()[numbers[i]];
                tempNpcNode = tempPlayerNode.getNpcTextNode();
            }
            i += 2;
        }
    }
}
```

Figure 29: Fourth part of the quest reader script

8 Professional issues

After delivering the game out to the public, there may be different issues that can come up in the future. this section will cover possible problems and a way of dealing with it.

8.1 Copyright issues

Copyright can be a real issue as it can be the main reason the project died. If I were to publish a video game and at some point I have used an external assets in the game like a sound or an image without permission, it can bring legal issues, therefore when using other people's work I need to get permission first, buy it from the owner or alternatively not use external sources at all which will require more work. When buying a license for an asset to the game it might be possible that it will require monthly payments, and when the payment is cancelled the functionality of the project could come to an end. Another important thing worth mentioning is to include my own logo/signature on my own project to protect it with the copyright.

8.2 Plagiarism

When publishing video games it is important to include a signature of some sorts withing the game. There was a case were a developer gave out a game project for non-profit purposes and later on the same game was found on the Nintendo switch market, or a case where a video game was released with it's own source code which was disastrous as many hackers used that opportunity to mess with the game. The reason for this is because other people could simple take the entire project and claim it as their own. Protecting the files is also important as the assets can be stripped and reused somewhere else. Quite a lot of video game developers decide to encrypt certain files to stop plagiarism. When releasing the game it is important to not include source code of the video game as it can also be taken and reused somewhere else or used to create hacks.

8.3 Bugs

Video games have a lot of 'moving parts' meaning some code may break and cause unexpected errors or bugs. It is important to test the game before the release and check as many possible actions as possible to debug the code. In the modern days there are always bugs on release, especially if the game development was rushed and had little time to debug the code. To combat the bugs during release I could create different patches/updates if any sort of bug is found in the game.

8.4 Security issues

Security is a serious issue, as a software engineer it is one of the most important tasks to make the software as secure as possible. Although there aren't any serious security risks in video games it could be possible that if a game has any way of accessing the internet a third-party could do something harmful and if a game has a store mechanic that hold private information it could be disastrously dangerous to not have any sort of security in place. Therefore it is important to restrict access to the game from scripts that deal with internet connections.

9 Literature review

9.1 Game programming patterns - by Robert Nystrom

Game programming patterns - by Robert Nystrom
This book helped me understand how to structure my code. I have used this book early in the project as a reference for how to pass variables by only referencing them, create singletons, observers, managers and many more. The book also includes several interesting patterns that I might use for my future code.

9.2 THE C PLAYER'S GUIDE - by RB Whitaker

THE C PLAYER'S GUIDE - by RB Whitaker
Very good book that included example code for various topics. So far the book was mostly useful when I was developing Animation in my project.

9.3 Make your own pixel art - by Jennifer Dawe and Matthew Humphries

Make your own pixel art - by Jennifer Dawe and Matthew Humphries
A honorable mention as I used this book multiple times when creating textures and atlases. The book is not related to programming but it does provide strong advice how to draw sprites.

1 - Collision is created when boxes, also known as colliders, interact. The game usually determines if Vectors (a,b,c,d) are touching vectors (e,f,g,h)
2 - Visual Studio is a developing environment used for computer programs, websites, mobile apps and many more.
3 - a layer in video game is a way of picking what object gets drawn first. Objects drawn first will always be at the back of other sprites.
4 - Texture atlas is a way of combining multiple textures into one and then using software to split the image into parts again.

10 Conclusion

Since the beginning my goal was to learn the basics of game and engine development, for this purpose I have starting working on this project. After I started development of the game I had my ups and downs, there was some content I wasn't able to develop due to time constrains and other factors but I believe I gained a lot from it. I have learned how colliders behave, what parts of the game need optimization, where to use algorithms or how to use trees in dialogue between NPCs and the player and most importantly I have gained a lot of first hand experience on game development.

Colliders were the hardest to develop. At the beginning I allocated a week to code them as I believed they would be easy. Unfortunately, they proved to be a real challenge. I have spent a significant amount of time on them, wrote multiple versions of colliders as I was trying to figure out different methods and algorithms. The most 'popular' bug seemed to be when a player touched two edges of colliders and they would negate each other's movements. Usually one change would create the same bugs in different locations. Ultimately I rewrote the code for the last time and fortunately managed to make it work. In all the knowledge I gained from this particular issue was very valuable and the next time I write a collision class I won't have the same issues.

One thing I found very interesting was the usage of layers. Usually Monogame gives an option to select a layer for each item but it's not possible to create a layers for a dynamic world, where new items, tiles and NPCs are constantly created and moved around. With that predicament I remembered how the Monogame draws sprites; first sprite to be drawn will always be behind, which is something like a back layer. I figured I can take advantage of it and decided to use one of the sorting algorithms I have learned in University to sort sprites based on their Y position.

When I was developing the game I always prioritized modularity. I always created classes that can be used by multiple objects. That also included quests, so I took upon myself to create a system where I can easily write a quest in a text document and the game will create everything itself and assign the quest to the NPC. That was very interesting as I used tree branching nodes to achieve this goal, which is another useful method I learned from university. In the end I achieved this goal and I can write few lines in my text document to create a quest which would take a lot of time if I were to do this manually.

In the end I am very happy with what I got. This project pushed my limits and I managed to gain a lot from it. After multiple mistakes, issues and the information I gained from this project I believe my next one will go smoother and faster.