[Document title]

[Document subtitle]

Gournay, Raphaël

Table of Contents

1 Analysis 3

1.1 Background and Objective of the Investigation 3

1.2 Research overview and process 3

1.3 Research Findings 4

1.3.1 End user requirement 4

1.3.2 Analysis of previous successful games 6

1.3.2.1 Key Findings 7

1.3.2.2 The Legend of Zelda (1986) 7

1.3.2.3 Shovel Knight Treasure trove (2014) 8

1.3.2.4 Metroid (1986) 8

1.3.2.5 Pokémon D/P/Pt (2006) 9

1.4 Specification: 10

1.5 Specification justification 12

2 Design 16

2.1 Program development and objectives 16

2.2 Language and graphics APIs/Libraries 17

2.2.1 Direct X 17

2.2.2 OpenGL-SharpGL 17

2.2.3 Metal 17

2.2.4 Monogame 17

2.2.5 SpriteKit 18

2.2.6 Choosing 18

2.3 Classes and game setup 19

2.3.1 Sprite class 19

2.3.1.1 Constants 19

2.3.1.2 Variables 20

2.3.1.3 Entity and Block derived classes 21

2.3.1.4 Database 25

2.3.1.5 Constructors 26

2.3.1.6 Maps-structure 27

2.3.1.7 Use inside the game 29

2.3.2 Fetch\_Data 30

2.3.2.1 Get\_path\_g 30

2.3.2.2 Get\_path\_map 30

2.3.2.3 Fetch 31

2.3.2.4 fetch\_Map 31

2.3.2.5 Add\_map 31

2.3.2.6 Get\_map 31

2.4 Setting up the objects 32

2.4.1 p1\_setup 32

2.4.2 block\_setup 32

2.5 Movement 33

2.6 SpriteKit project bundle 33

2.6.1 Project Directory 33

2.6.2 Art assets and the GameEngine.projitems file 35

2.6.3 GameScene 35

3 Index 36

3.1 File Directory 36

4 Bibliography 41

# Analysis

## Background and Objective of the Investigation

* Decided to build the game mechanics for a 2D AA VG

The video game industry is a major player in the entertainment sector, reaching a global net worth of $135bn in 2018, and 2.3bn active gamers worldwide. (Wijman 2018). It is estimated to reach $180bn in 2021 after having experienced exponential growth in the past 6 years. As a comparison the TV and video industry is currently worth $136bn as of 2018. (Wikipedia 2019).

* Wide Variety of genres

The sector on which I focussed my research was 2D Action-Adventure games,

* Examples

specifically the movement mechanics of the objects inside the game. My aim for this project is to determine which fundamental mechanics of a game play into its success

* by coding a sufficiently complex and efficient game, since optimising time and investment is paramount in video game development, being an industry where smaller Independent studios (Chucklefish, Yacht club games, Team Cherry, etc) have to compete against a mix of large international corporations (Microsoft, Nintendo, Ubisoft among others).

## Research overview and process

* In order to decide how to build the program, I first analysed the elements that made a game attractive and successful.
* Started by defining key success factors for this type of game.
  + Interviewed gamers to understand what they looked for in a game.
  + Focused on factors I could drive through the programming and coding, recognising there were other factors I could not address (music, story).
* Looking at past games and analysed the appeal of certain recurring and inventive mechanics and how they were implemented.
  + Key focus on movement fluidity and collision management
  + Also allowed me to find practical solutions as to how to implement these mechanics (tiles, database, etc).
* This allowed me to define what the game should be able to achieve, how to design it and define precise specifications.

## Research Findings

### End user requirement

Another part of my research to find out what core mechanics would be successful was asking and interview people for what kind of things they looked for in top-down 2D video games. So I interviewed a multitude of people, asking them questions as to what really attracted them towards video game titles, as well as what kinds of attributes they want to see and use to discern what kinds of games they want to buy.

|  |  |
| --- | --- |
| Attribute | Reason |
| Intuitive | The game has to be easy to pick up and easy to control, you have to be able to tell what the controls do, instead of an unnecessarily complicated system that makes the game hard to pick up and play for new players. |
| Mechanics | When looking at a game, the basic movement and control mechanics have to usually be present, but what really helps to set a game apart are the unique movement mechanics introduced to the player in the game. What's important though, is that these mechanics don't feel shoehorned into the game, but the game and level design is built around these mechanics, forcing you to use and exploit them to your advantage to do well. Making these mechanics satisfying to pull off and use also makes the game more enjoyable, an example I was given here were the Shovel Knight: Treasure trove Shovel Knight, Spectre Knight and Plague Knight Campaigns where the levels of the game all feature the same areas and enemies, but the level design is strongly changed, as are the behaviours of the bosses to make each of the game modes play around the characters and their abilities completely. Another game mentioned was "Celeste", a platformer featuring strong Physics elements, that are used consistently throughout the game. |
| Not having fake 3D and 2.5D elements | Making the game in a fake 3 dimensional element, or 2.5D when it doesn't need it is unnecessary and distracting, it doesn't add anything interesting to the game in itself. However, detailed dynamic backgrounds to give a sense of a larger world like in games such as: "Shovel Knight or "Starbound" work, because they give a sense of dynamic movement otherwise lacking in some 2D platformers. |
| Complexity[[1]](#footnote-1) | People like complexity in games where it develops over time, forcing the player to become better, to think more about their actions, without ruining its original intuitiveness. In other words, featuring a difficulty curve to spice up gameplay and push the player to become better. An example for this I was given was the hit game "Super Hot", where the game starts of easy, and with each progressive level becomes harder, taking the player up a learning curve, leading to very difficult and complex gameplay later in the game.  On the other hand, it is important that the game be not swamped in obsolete mechanics such that they overshadow the actual gameplay itself and push the player away from it. |
| Level Design[[2]](#footnote-2) | Much like described in the Mechanics section, the levels have to be streamlined, designed around the mechanics of the game, to make the gameplay more interesting, a good starting level is one that acts as a sort of tutorial without feeling as one. Most players do not want their hands held through what seems to be the umpteenth platformer they've played, or suffer through multiple hours of expository dialogue on how to perform a single mechanic which will only end up helping them once throughout the game.  Of the 2D games that are most enjoyed and positively viewed for their level design are the ones where no part of the level is obsolete or added just to make the game look good. Every part of the level design has a purpose, to enhance the experience provided by the gameplay.  It is also important that the level give options to the player as to what they are and aren't able to do, give them multiple paths for tackling a problem and reward the player for their creativity and skill.  The games I saw most praised for this were: Shovel Knight Treasure trove and Hollow Knight. |
| Story | Whilst most games have a story, it is important that the story take the backseat, it should itself be based around the gameplay and mechanics, and work in conjunction with the gameplay, not against it. Having the story be completely unrelated to the game serves no purpose whatsoever and often ruins immersion and experience. |
| Good music | In general, good music helps set a theme in a game, as well as emphasising the in-game atmosphere to help immerse the player into the world. Multiple people told me that music was a large factor when they were deciding whether to buy a game. |

Focussing on the attributes I could control, I translated them into clear, well defined objectives:

* Intuitive
  + Simple to move.
  + Movement is free.
  + Character reflects status.
* Mechanics
  + Movement mechanics are well defined
  + Collisions are strongly and coherently managed.
  + The objects of the game obey set physical laws.
* Realistic 2D
  + The object should implement complex movement mechanics in all dimensions without forcing it upon the user using 2.5D or 3D.
* The complexity of the game should be variable.
  + It should be possible to increase the complexity of the game by adding, removing and altering objects.
* Level Design and re-usability
  + Reusable templates for sprites to construct larger levels.
  + Ability to move from area to area on a larger map.
  + Make design efficient and don't force mechanics onto the user.

### Analysis of previous successful games

For my investigation, I decided to look at simple 2D platformers and overhead perspective games such as the original Legend of Zelda and Metroid games, that are known classics that defined their genres and the video game industry.

I looked what made them work so well and how they integrated their well-polished mechanics with their gameplay. However, since the game industry has progressed much since then, I also needed to look at some more recent titles to see if the kind of gameplay offered by these games was still successful.

To this end, I decided to also take a look at more recent games and prototypes like the indie hit game Shovel knight. Another thing I wanted to look at was the way the overworld was modelled in main series rpg titles such as Pokémon, since despite the main intrigue being the combat system, the puzzles and level design are quite interesting, being rooted heavily in exploration, and there have been innovative Physics ideas in some titles, incorporating Inertia and momentum with bikes in generation 4 (Pokémon Diamond, Pearl and Platinum), and having collisions and Friction being a large part of some puzzles, as well as exaggerated forces due to winds and dealing with sinking into holes and climbing out of them, as well as breakable objects.

#### Key Findings

* Fluidity of movement
* Collision management
* Use of tiling system.->positioning
* Reusability of sprites

#### The Legend of Zelda (1986)

The original Legend of Zelda game is one of the most famous games to have ever been created, and revolutionised the gaming industry, selling over 9.03 million copies (Fandom 2019). It involved non-linear gameplay and forced players to think about what to do next, as well as encouraging sharing and communication between players to allow them to unlock the different secrets stashed within the game, as opposed to the multitude of linear games that existed at the time. It was also modelled as a coming of age story and featured a progression system whereby the player and Link would grow stronger together.

The game simulated a sort of forced fake three dimensional perspective on the player to make it seem as if they were looking diagonally down onto a field, instead of the world seeming completely flat, a method still widely used in video games today.

The character is only able to move in the cardinal directions and with a highly accurate collision detection system. Link's (the name of the primary protagonist) movements and sprite animations are fluid and smooth giving the illusion of fully analogous movement, giving a sense of fluidity to the controls. Seeing as how this level of fluid control has become a staple of mainline Action adventure games and future two dimensional overhead view Zelda games, it is safe to say that this is a key factor in the success of these games.

While it seems that some obstacles are themselves pixel accurate on their hitboxes or the bounds where entities such as the player can touch them, the majority of the various obstacles of the game such as rocks and trees are not. These obstacles occupy a space which is around the same size as the player, and have their sprites reused throughout the game, the same can be said of the basic enemies inside the game and their spawn animations, who themselves are all no larger than the player, and seem to appear in the same square positioning grid as the obstacles, possibly indicating that these spawn positions and block positions are stored in a file which indicates their co-ordinates. Something which could definitively be a possibility, enhanced by the fact that the game was built to be able to run on the NES's limited hardware restrictions, whilst trying to make sure its assets took up the least amount of memory in the cartridge, to increase the amount of things that could be fit inside the game. This could mean that to save space and/or to reduce the complication of placement calculations, Nintendo used a different system than representing each pixel as a single x/y unit. This is also visible from the paths and gaps in the walls and the size of large objects such as lakes and rivers having their dimensions being exact multiples of the player's size. This is made even more apparent from the fact that the walls of the early stages of the game are formed from blocks of the same sprite, which are again the same size as the player. Conversely, this could have been done to more easily build the world. However, since the entities in the game seem to also be able to move to some extent inside a small area (as large as the player indicating that the possibility of movement of the and collision with solid objects of the entities is not calculated from the same hitbox as the one used to judge projectile and attack collisions. The level of accuracy of these collisions must also mean that the size of the hitbox and its current bounds as well as the projectile or attack's current location and hitbox are used in calculations to find if an object hits. Having files to store levels as images, or as maps and creating levels out of smaller pieces seems to still be a mechanic used today in some game creations, with heavy use in 2D platformers and top down Roguelike RPGs where the maps can be generated randomly from modular rooms.

The game also has the ability to manage multiple enemy projectiles at the same time. The different projectiles of the game also have different properties, with some being able to go over obstacles, whilst others would not, prime examples of the former being Link's beam sword and the Moblin's arrows and an example latter being the Octoroc's rocks. This could mean that for possible collisions, the game looks up the projectile's values to see if it is stopped or it goes through the object it is currently touching. One thing to note about the projectiles is that most are generated behind the sprite of the current object emitting it, to make it seem as though they are being shot by that entity. An exception to this is Link's beam sword which is generated under the sword of Link's attack animation. The projectiles also seem to have a bigger z-display position than the obstacles in the game, since they appear to pass over them. An interesting mechanic that is not obvious at first is Link's ability to block projectiles with his shield. If Link is not currently attacking and is facing the projectile coming towards him, it will bounce off diagonally.

Although the game is filled to the brim with enemies, a lot of them are variations and re-colorations of each other. These species of enemies all have the same behaviour, with the differing colorations indicating maximum health. This allows the player to feel a sense of progress as they might struggle at first with weaker enemies, but as they and Link get stronger from overcoming trials and adversity, this allows them to be able to take down stronger enemies.

#### Shovel Knight Treasure trove (2014)

#### Metroid (1986)

The original Metroid sold 2.73 million copies for the NES (Fandom 2019). It was strongly celebrated for its non-linear exploration and discovery system much like the Legend of Zelda. What set it apart from its counterpart however was its vertical platforming element and darker atmosphere. Along with Castlevania, it helped create its own genre of games (Metroidvania), and is no doubt one of the most influential games of all times.

Much like Zelda there is a strength of progression along the game, as the player must unlock permanent power ups in order to be able to move forward. The game is also heavy in backtracking, as once the player has unlocked a new power up, areas that were previously blocked off might now become available.

Also visible is the way the levels are designed with the rooms of the game being comprised of small blocks which are half the size of Samus, which again displays the methodical approach to designing the world or map to be compact and easier to store, like described above in the original Legend of Zelda section[[3]](#footnote-3).

The game also includes a feature which allows you to save your progress through the use of a password system, that you have the ability to input into the game to access your progress on another machine.

#### Pokémon D/P/Pt (2006)

For my research, I decided to look at one of the most best-selling Pokémon games: Diamond and Pearl, having sold over 17.67 million copies worldwide (Nintendo 2018). However, the main selling point was the combat system, although throughout the years, Pokémon has added some staple movement mechanics to its games, with this title being no exception. The game features a complex non-linear overworld, which much like Metroid talked about above requires the player to backtrack before advancing, and gives players a sense of mystical mystery with the atmosphere set by the music and graphics in the sparse secret areas scattered throughout the game. Despite the players having to backtrack through areas they have already visited, the game still manages to push for a sense of exploration through the multitude of various intertwining passages in Mount Coronet's vast cave system, through the use of height differences and platforms, as players will find themselves going back through rooms they have already visited albeit in a different part.

Another interesting factor in the games is the Bike's gearing system. The Bike gives the player the ability to switch gears between third and fourth gear, where fourth gear has some acceleration mechanics and has the ability to travel faster, at the cost of mobility. In some puzzles, the player will have to accelerate the Bike over a distance in fourth gear to be able to climb some muddy slopes to get to new areas, or have to methodically choose which gear to use when having to clear ramps, such that they are able to land in the correct spot to be able to progress. One notable example of such puzzles is Wayward cave, where there are four ramps where the first three jumps have to be performed in fourth gear and the last performed in third. One thing to note is that it is possible to fail to climb the slopes or ramps if the bike does not have enough momentum, with the game having the ability to calculate the height to which the player will climb when they attempt to ride up the slope, and if they do not have enough of it, to make them slide down and away. This acceleration and momentum mechanic for the Bike's fourth gear is also present in the way that when moving fast in fourth gear, the player will experience more difficulty stopping the bike and turning it.

Yet again, looking at the overworld, we see the modular design mentioned so much in Metroid and Zelda, however, in this case, it also uses some of the DS's 3D model capabilities, as visible from the way the buildings are rendered and the design of the walls inside the game's caves. The maps of the D/P/Pt games are broken up into squares of size 32 by 32 tiles. To save memory and processing power, only four of these squares are loaded at once, and in such a way that the world appears uniform and seamless (Bulbapedia 2018). What is also apparent is that the ground textures used in the game for certain areas are also only the size of a tile. Unlike the other games mentioned above however, the processing and data collection is much less obvious due to the fact that the 3D models and sprite designs make it such that the objects of the game blend much better together, as well as the presence of larger objects that take up multiple tiles.

In contrast to the more variable movement of the above titles, the player only has the ability to move by a single tile in the cardinal directions inside the game, albeit the speed at which it happens can vary in some cases[[4]](#footnote-4). Whereas this mechanic would be seen as a hindrance and a downside to the game, it allows the game to be able to generate its random encounters on the different possible tiles of the game, randomly deciding using the encounter probability whether the next tile the player steps on will have a random encounter and if so, what Pokémon it will be. This mechanic remained a staple feature of the franchise itself until Pokémon made the jump to 3D, and even then, in 3D game for the overworld, caves, and most areas, the same type of tile system can be seen.

Some tiles in the game will also have different properties, with slippery ice tiles preventing the player from changing their initial direction of motion, and making the players sprite slide along until they either land on another tile, or collide into an object, allowing for some complex puzzles. Other tiles such as Marsh tiles and Snow tiles can restrict the player's movement by trapping them in a hole temporarily, forcing the player to think about what they do next to make the most efficient path.

## Specification:

Having taken in the most common and recurring aspects of the above games, I settled on making a 2D top-down game/simulation.

1. The game should have a clear user interface.
   1. The user should be able to have a clear understanding of what is going on where and when, ie what objects are positioned where and how they are moving.
   2. The user should have a clear idea of the controls and how they work (ie the keys used to control the character).
   3. The graphics implemented should be smooth and fluid, not slow and choppy.
2. Entities and moveable objects should have fluid multidirectional movement bound by set rules
   1. The players and entities should have the ability to move in the Cardinal directions.
      1. The movement of the entities should be well defined and the player should feel in control at all times
      2. The player should also have the ability to climb onto objects in order to overcome obstacles throughout the game, to create a sense of vertical movement.
      3. The sprites of objects should be altered in real time to show their current status, ie direction.
   2. The object should not be able to move into objects, or outside the boundaries of the area, in order to implement this, collision detection will be added.
      1. The collisions of the entities will be dictated by their in-game size, and will be strongly implemented to prevent an appearance of moving into objects.
   3. The entities of the game should be subject to set Physical laws dictated by their attributes.
      1. The entities of the game will be subject to laws of conservation of momentum and Inertia and Impulse, such that they will experience rebound when colliding into something, and have to decelerate over a distance when changing direction. These attributes will be small however, to make sure that the player does not feel their character is out of control.
      2. If an entity keeps moving in the same direction for some time, they will accelerate to a maximum threshold value instead of moving at a constant speed.
3. The in-game objects should have multiple properties that dictate how the player can interact with them. These should be defined outside the program and fetched from a database.
   1. These properties should be defined by their type.
   2. The objects will be defined in a general database which instructs what kind of object they are, so that the program is able to refer to the database for those types of objects to use parameters specific to that kind of object.
      1. The primary database for the objects should refer to other smaller databases specific to each type of object, through an ID system.
      2. Each different object in the game should have a unique ID to identify it.
   3. Objects should have properties that allow their states to be altered, so that other objects and the player can interact with them.
4. The game should be able to build levels from pre-made maps, that identify the positioning of certain objects.
   1. These maps should refer to the different types of objects in the game and their positioning using their ID, in order to initiate the game.
   2. The maps should constitute of a set size with a set amount of tiles contained within each map.
      1. Each object added to the map should only take up one tile.
   3. The player should eventually be able to move between areas on a larger (in-game) world map.
      1. This movement should be signalled by a screen animation visibly displaying this shift in area while the area is being loaded in.
      2. The game should be made such that it only displays one map at a time and clears the screen from objects whenever the area changes.
5. The game should be able to fetch the data stored in text and database file no matter where the project file is, on any computer, such that file location errors do not occur.

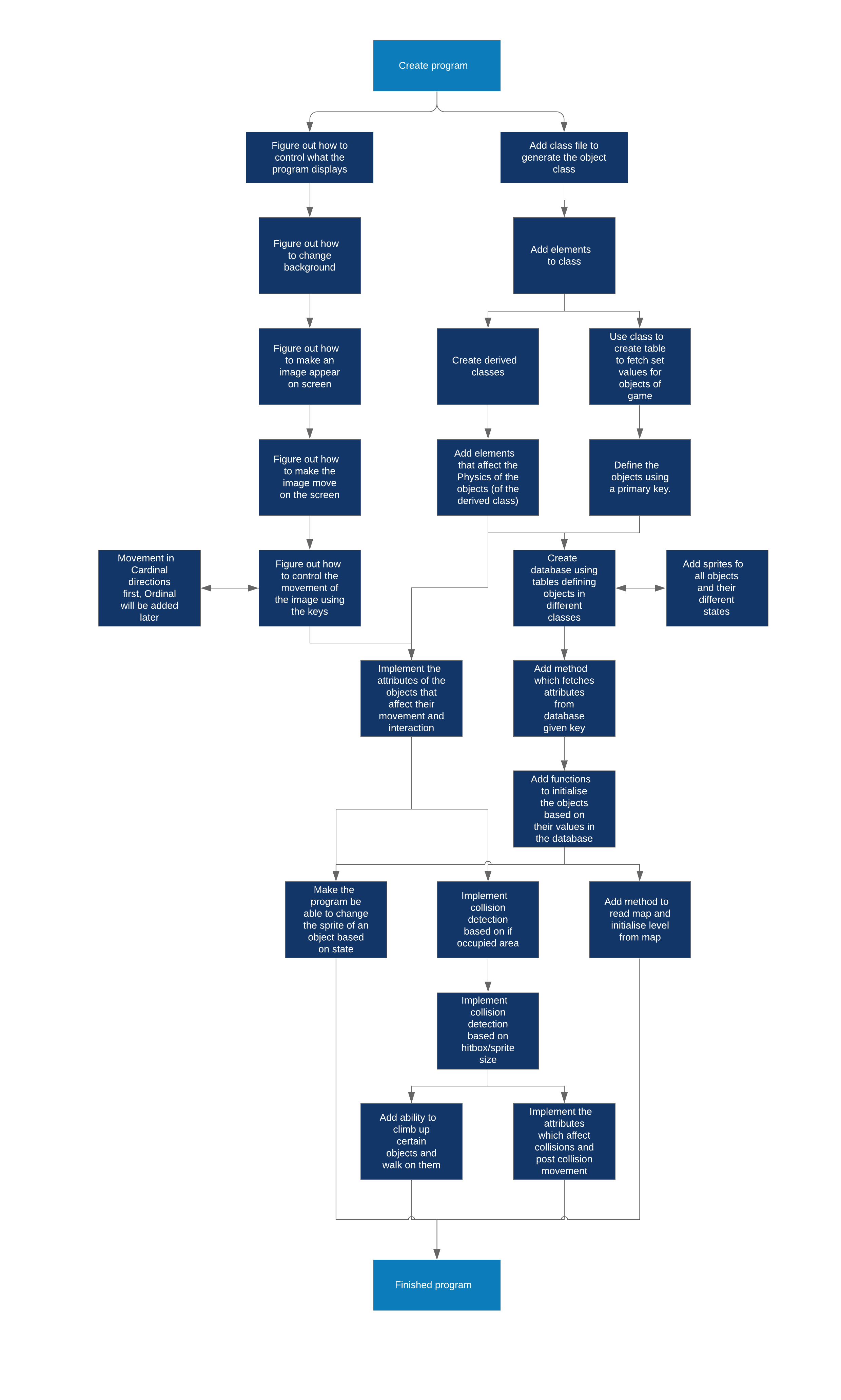
## Specification justification

1. When playing a game, it is of the utmost importance that there is no ambiguity to the player as to what is happening on screen and what they are doing. What is clear in the most enjoyable games is that from the get go, you know what you are and aren't able to affect on screen, so that players don't get lost.
   1. When the player does something, or an object moves on screen, it is important that the game effectively feeds back to the player the output of what they just did.
   2. The controls of the game should be made evident and clear to understand, the game should not be flooded with useless controls and button checks for very specific action, and should have a clear idea of what they can do and how they control the game.
   3. The input and controls of the game are important to how much the player is able to enjoy the game. Looking at the above games researched, none of them were successful because they had bad gameplay, conversely, what sold most of them to audiences so well was because the gameplay was smooth and fluid and responded extraordinarily well to user input.
2. For the game to feel more interactive and structured, adding well defined motion and movement mechanics is important such that the player is able to feel a complete sense of control over their character.
   1. Only allowing movement in the Cardinal and Ordinal directions was chosen due to the simplicity of the design and input requirements, since making it possible to control the character in 360° would be far too complicated to control and implement, not to mention unnecessary, seeing as most games owe their success to making the controls simple and easy to use.
      1. Making sure that the player is happy with the way they are able to control the character is one of the primary aspects of making sure that they want to play the game.
      2. Using sprites to show their current stateeffectively communicates to the player what they are doing, where they are heading, and is a feature that is present in the Legend of Zelda, Pokémon D/P/Pt, Shovel knight.
   2. Making the game bound by Physical laws adds a level of depth and complexity to the game, instead of making it fairly linear and boring, adding some attributes that make the game feel more alive and responsive, as well as allowing some situations that would force the players to use or manipulate these mechanics to their advantage to solve puzzles.

      2. Acceleration is an interesting mechanic because it allows the world inside the game to feel more real and concrete. As described above in the description of Bikes in the Pokémon diamond and pearl section, it allows for some interesting puzzle ideas as well. Although pushing it to too much of an extent reduces the control the player has over a game, and prevents makes manoeuvring more difficult and dissatisfying for the player.
      3. Implementing collisions and impulse such that momentum is conserved, as it forces the user to think about how they are going to pilot their character, without recklessly running into things as this could undermine their progress, for example implementing sliding tiles and a slippery surface at some area with obstacles to manoeuver around carefully.
      4. Dictating collisions by how large the sprites are and how much they overlap instead of hidden values, again increases the communication the game has with the player and lets them know that when they go into an area, they will collide with that object, or how exactly they need to move in order to not get hit by an enemy or a projectile.
3. The addition of in-game properties defined by the class of the user allows control over their mechanics of interaction, as well as a definition to be able to integrate them into the game and have them walk around and interact with the player.
   1. Implementing different classes for the objects allows the formation of derived classes, where a derived class could be defined for a very specific type of object that acts differently, as well as allowing the game to compare the objects to each other more easily, by being able to fetch and set its values. It also allows the creation of functions of the derived classes that are able to take in inputs from a member of the derived class when fetching its values from files. It also means that instead of having one type of uniform structure and table for all of the objects of the game, where a large part of some values might not end up being used for objects resulting in a waste of space and potentially resulting in errors, the game can fetch the specific values from independent tables for each type of object. Using a class to derive classes for all objects also allows data structures containing multiple different kinds of objects to be formed (ie a map of the objects and their positions).
   2. Using a database to define the objects allows their values to be flexible and altered during development to balance and correct the game so that the objects are the game can function coherently with their design, as opposed to hard coding them into the game, where it would be necessary to navigate through the program and needlessly create a different case for what values to generate/attribute depending on the object being created.
      1. Referring to the other databases allows identification and normalisation, of the database to ease manipulating the data and the time taken to search through the document, as well as making the attributes specific to each kind of object easier to manage, change and add.
      2. Referring to an ID instead of a name prevents collisions between objects which might have similar names, and makes it easier when creating maps of the environment, as well as making labelling sprite images easier.
   3. Allowing the objects to have changeable states such as health speed, breakability, movability and so on, allows the game to become more complex, as well as allowing the addition of enemies the player can kill, such factors are present in all the games Researched, and is clearly a core mechanic when it comes to such games.
4. Building levels from pre-made maps and fetching them from files is also a core and instrumental mechanic that is visible in all the above games, as it allows level design, and the linking of separate areas of the game; giving the player the ability to move from one 'room' to another seamlessly.
   1. Referring the different types of objects by their ID allows the program to uniquely identify the objects and create a new instance of them at that position when generating the map.
   2. The tile system, is something we see in Zelda, Metroid and Pokémon, all generating the base map for the world using a set tile system, where large walls or structures are made from the combination of many smaller tiles.
      1. Making it so that each object of the map only takes up one tile prevents overlapping when loading the game, as well as making sure that all the maps have one uniform standard for reading them and generating the objects from their data.
   3. Allowing the player to move between areas of a larger world map adds a sense of openness and largeness to the world making it seem larger and more interesting, making it accessible to move between areas, instead of making the game one giant also makes the world seem bigger, and makes it easier to manage the objects loaded into the memory rather than allowing scrolling.
      1. Visibly displaying a movement from one area to the next using an overlaid animation both gives time to the program to load in the next map, and gives the player time to figure out where they have ended up due to moving to an adjacent map.
      2. Making sure that the game clears the screen from objects and sprites before moving to the next is highly important to make sure that the game does not appear to bug and ruin the gameplay and immersion experience.
5. Making it such that the game is able to fetch the files reliably is necessary to keep the program functioning to prevent any issues or bugs, so that the game can be reliably shared and tested between devices, independent of user.

# Design

## Program development and objectives

Before I could begin making my program, I had to plan out how I was going to code it and what objectives I would aim for as the code went along to allow me to meet the planned specification. To this end I developed the Following Route map:   
As visible above, the entirety of the design process was not one single linear process, in order to be able to streamline my work, I needed to work advance on different parts of the project simultaneously, in modular implements so that I could have a clear idea of what I was doing and where I was going at any time. I could also have the ability to test and polish each individual part of the project to prevent errors from occurring later in development, where they would be more troublesome and harder to find, as opposed to coding larger parts of the program all at once. The design for my process was also itself quite flexible, I could easily switch between branches of the of the map as I advance, which would allow me to move forward if I got stuck on something to give me the time to come back to it later, since the project begins by working on the different aspects of the design and eventually bringing it all together in the final few steps.

## Language and graphics APIs/Libraries

When making the game, I decided to code it in C#, since it was the language I was most familiar with, and since the machine I was using to code my project was a mac which did not have Wine or Parallels Desktop, or any other software that would let me run Windows on Mac, I had to find an external graphics library, frameworks and/or APIs which would be compatible with Visual Studio Mac, and which would let me be able to have an output for the program. Through looking at online forums and what developers of recent games that I knew were compatible with OSX, I ended up with a list of possible solutions, following the most recommended and popular ones:

### Direct X

I decided to take a look at Direct X on windows since Yacht club games had used it for Shovel knight to build their own game engine. Direct X is a collection of APIs, however, since it is Windows exclusively, it was not relevant to my project.

### OpenGL-SharpGL

I decided to take a look at OpenGL for OSX and Linux since it was used by Yacht Club games for the development of shovel knight, specifically looking into the SharpGL library which made it compatible with C#, since OpenGL is coded in C. However, I learned that "OpenGL was deprecated in MacOS 10.14" whilst looking at the apple developer tutorials and instructions on the API. Since I use OSX 10.14, and Apple had deprecated OpenGL in favour of their own API: Metal. I had no choice but to look elsewhere.

### Metal

Seeing as Apple had deprecated OpenGL in favour of Metal, I decided to give a look at the API, but gave up after finding out it wasn't compatible with C#, working only in objective-C and swift, both languages in which I am unfamiliar.

### Monogame

Monogame was a strong contester and framework which I saw being recommended in multiple online forums, being cross platform and compatible with Visual studio, however I did not end up using it, because of the learning Resources that Spritekit offered and how much easier it was to use.

### SpriteKit

Since the Spritekit library came packaged with Xamarin and Visual Studio community 2017, and was very strong with a lot of online support, it was ideal for me to use, since was easily accessible and had all the features I was looking for, as well as the fact that it was compatible with my current OSX version meant it was by far the best choice to use. Although most online support is in Swift, as are most tutorials, due to the similarities between C# and Swift, it was easy to bridge the gap.

### Choosing

In the end, I decided to choose SpriteKit because of how much more available it was in comparison to the other libraries, as well as the fact that it had a lot of online support from the Xamarin website, and a multitude of helpful and useful tutorials (in swift). The fact that it was built by Apple to work using their own integrated systems in OSX and iOS, made it the obvious option, Monogame, being a close second due to its viability and cross-platform abilities.

After having decided to use SpriteKit, I had to make sure that it would be compatible with the built-in libraries I needed to detect key presses so, I changed the project framework from Xamarin.mac, to .NET 4.7. In order to learn how the library worked and its functions I turned to online swift tutorials, due to the lack of C# tutorials, since the languages were so similar.

## Classes and game setup

### Sprite class

Taking inspiration from the tile systems present in the original Zelda and Metroid games, as well as the tile and height system heavily present in Pokémon, I built a class that would be able to represent the objects inside the game.

The Sprite class stores all the necessary attributes of the objects that fill out the world, such as the player, enemies, and blocks/obstacles. Originally, I was going to build it as a structure, however, due to the lack of flexibility and relationships between the structures, I opted to use a class instead.

The base class is made to act as an abstract class to build its derived instances. It stores the main values that allow the program to initiate and display the objects and their sprites onto the screen. In fact, the class is split up into two groups of data: The constant data that is initiated when the program is launched, or when the object is generated, and the variable data that can change as the program runs.

#### Constants

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| ID | String | The ID allows the specific type of object to be uniquely identified such that its values can be fetched from the table. It also is used to uniquely identify the sprite names. |
| Name | String | The Name helps identify what kind of object it is when developing, such that it was easier for me to classify each object and see recall what they were built to do. |
| Solid | Boolean | The Solid attribute determines whether other moveable objects are able to pass through them when they collide. |
| Type | String | The type field states the kind of object we are initiating, and so what kind of derived class/ child class to initiate, as well as which table to fetch the data from. |
| defaultZ | Integer | The defaultZ field is used to initiate what kind of z display position the object is should be given to prevent certain objects from appearing to overlap on top of each other, as well as preventing objects from being placed behind another when making the object move around the map. |
| path | String | The path field is the string used to identify the path to the location of the object's sprites inside the project folder such that they can be found such that no matter the computer or user loading the project, if the project folder is complete it will be able to successfully find the path to its sprites. |
| yShift | Integer | yShift is like defaultZ a display correction field added to give a bigger sense of realism to the display such that some characters appear to be standing inside a square instead of occupying it, to make the game appear to have a more angular view to make it easier for the player to visualise the platforms and blocks of the game. |
| spriteh | Integer | Since the width of all sprites is fixed such that they occupy only a single tile, spriteh gives the height of the object's sprites such that it can be initiated correctly, for example, for larger objects such as taller blocks, their sprites will be taller so that it appears that the object placed on top is higher. |

#### Variables

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| spriteNode | SKSpriteNode | The spriteNode class deals with all the display and positioning and moving of the object, it is a complex class containing the sprite's exact position (stored as a CGPoint), the sprite's current texture, and its size. This class is instrumental in allowing the code to used to calculate and return the objects movements and positioning to communicate with the SpriteKit library to output the results in an Application window. |
| xPos | Integer | The xPos field is used to represent the nth tile in which the object is currently in in the x-direction, which allows the program to initiate the objects when fetching a map, as well as defining the boundaries of the map, and helping to look for collision detection. This is used in conjunction with a 3-dimensional array of the sprite class to |
| yPos | Integer | The yPos field is the same as the xPos field, but in the y-direction. |
| zPos | Integer | The zPos field represents the current height the object is at, this implementation allows to detect when the object should fall off a platform, and is used in conjunction with the three dimensional array of sprites to do so. |
| actualX | Double | The actualX and actualY attributes are like the xPos and yPos attributes but different in the way that they will be used in order to model the movement to make it fluid, more pixel by pixel instead of making it tile by tile, as would be necessary if using only xPos and yPos. Using the knowledge that each sprite's hitbox is one tile big, this will also allow to use collision detection based on using these 'actual' values as a base for our calculations. |
| actualY | Double |

#### Entity and Block derived classes

When making the sprites class I had to consider the types of objects that would be in the game, in the end, taking inspiration from the fact that in video games there were only really two kinds of objects; moveable entities and obstacles/fixed objects which the player had to overcome. Since in this game/simulation both kinds would be present I created the Entity and Block derived classes.

The Entity class contains the attributes for the self-moveable entities inside the game. These entities range from enemies to the player and all required variables that defined their movement and attributes as to how they would interact with each other. Since the game would be based around movement I added attributes that wold define their movement to meet the specification such as a Base speed, a maximum speed, a mass and a value for its acceleration to meet part 2.b.i , 2.b.ii, 2.b.iii, as well a attributes such as Strenght and Health to meet part 3. of our specification. And it should be able to reflect its movements in the output to meet part 1. of our specification. To This end, I came up with the following attributes

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| Speed | Double | The speed property describes the speed at which the entity is able to perform an action, such as move, the more an entity accelerates, the faster it moves. |
| last\_direction | string | The last direction field describes the last direction in which the object moved, such that if the new direction is equal to the last, the object will accelerate towards its maximum speed, otherwise, its speed will return to its base value. |
| destination | CGPoint | The destination CGPoint is a copy of the point towards which the entity is moving, this allows the entity to perform checks whether they have finished their move to see whether they can begin another. This also allows the entity to set themselves a long term goal to move towards, for example, so that an entity can follow another. |
| Base\_Speed | Integer | The Base speed attribute is used to reset the speed once an object begins moving or changes direction. |
| Max\_Speed | Integer | Max\_Speed dictates the maximum boundary for the entity's speed so that it does not keep accelerating in a straight line ad infinitum. |
| Acceleration | Integer | The acceleration of an entity defines by how much its speed will increase if it keeps moving in the same direction. |
| Health | Integer | This field stores the current health of the entity, such that once their health reaches 0 or below, they will die. |
| Strength | Integer | This field stores the current strength of an entity, this amount defines the amount of damage they inflict upon other entities, such that once damage is inflicted by an attack, the strength is used to calculate how much health is lost. |
| spritef | String | These 4 strings indicate the paths to the directional sprites of the entity such that the program is able to fetch them to reflect the direction the entity is moving in. |
| spriteb | String |
| spritel | String |
| spriter | String |

Since the different entities will all be using different files for their directions, instead of having a function to determine which sprite to fetch or using multiple lines calling a file using the path field being called or used every time the entity moves, I instead decided to make the program define the paths to the fields when generating the entity.

The block derived class contained all the fields for the objects/obstacles inside the game that would be unable to move by themselves. Particularly to see what kind of interactions I had to build, I looked at part 3. of the specification and the interactions present in Zelda, Shovel Knight and Pokémon D/P/Pt, specifically D/P/Pt, where blocks could be interacted in various ways. After this, I came up with the following fields:

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| Climbable | Boolean | This attribute describes whether a block is climbable inside the game, so that if an entity tries to walk into it, they will be able to go on top of it, and walk onto other non-climbable blocks of the same height. The addition of this attribute will help add a level of complexity and interest to the map, as well as the addition of multi-level maps. This attribute was inspired by the cave and overworld puzzles featured in D/P/Pt. |
| height | Integer | The height attribute describes how high a block is inside the game, which along with the Climbable field will dictate whether an entity can climb a certain block, this addition paired with climbability allows the entities to interact with the blocks to solve large complex multi-level mazes. |
| Traversable | Boolean | This field dictates whether a block can be walked on. This allows the addition of non-platform objects which can act as obstacles to the player. |
| Breakable | Boolean | The Breakable attribute controls whether a block can be broken by attacking it, in a way that if a block is broken it loses its solidity and can now be passed through. |
| Moveable | Boolean | Moveable blocks have the ability to be pushed around by the player to solve puzzles. In conjunction with the traversable property, this allows the construction of moveable platforms. |

Since all the attributes above had to be fetched whenever a sprite collided with a block it had to check these attributes, I decided to implement hollow functions in the main class which I overrode in the derived class to make sure no errors were thrown, when trying to fetch the field.

Since I also needed a class that would allow me to construct instances of the other two, I added the Basic class as a way of temporarily storing values that would create an instance of the other two main derived classes.

#### Database

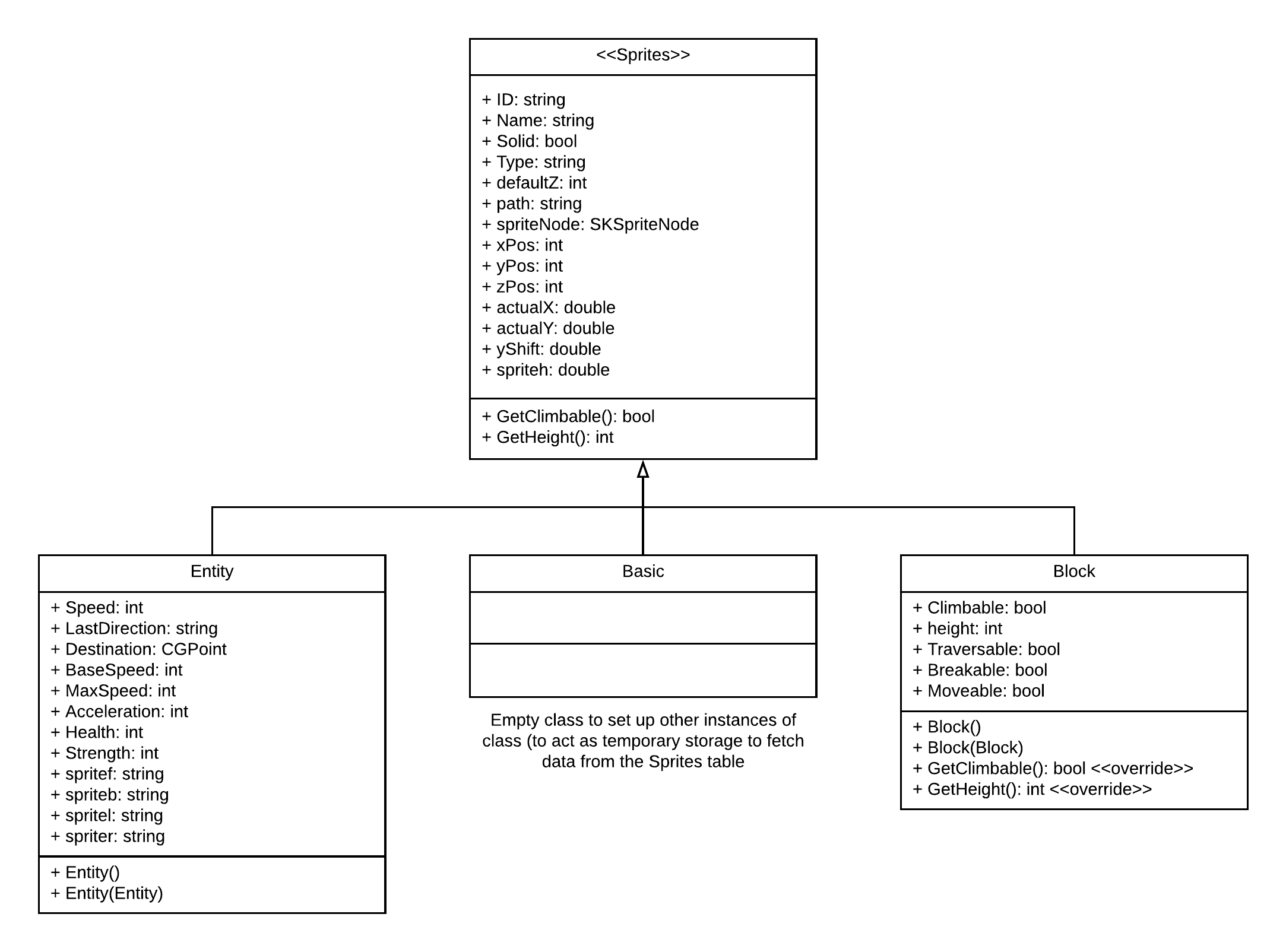


Figure . Sprites class diagram

After building my class structure, I should end up with the above diagram, with the classes being related in the above way. Since I needed the program to fetch the values for the sprites it was using whenever it needed to create a new instance, I added tables:



Figure . Sprites table

Using the ID field (highlighted in bold above in Figure 2.2 Sprites table) as our primary key, this would allow us to uniquely identify the different sprites across all the different tables. The type field, would direct the program as to which table to search into to find all the values to fully construct a derived class. This relates to all the constants in 2.3.1.1, as opposed to our variables, which are defined using other means, when the sprites are instantiated in the game.

Below, in Figure 2.3 Blocks table, we see the table with the constant attributes exclusive to the block class, and in Figure 2.4 Entities table the attributes exclusive to the Entities class.



Figure . Blocks table



Figure . Entities table

Storing these tables as .csv files, in a tables folder in our Resources directory, we could then access them whenever we need using a fetching function (See Fetch\_Data). This would then allow us to initiate an instance of an object in our game as per point 3. of out specification.

This set up is far more practical, meaning that instead of hard-coding our values into the program for every instance of an object, or creating countless instances of derived classes, which would be nightmarish to maintain and navigate. we can use the tables to easily alter attributes and manage them. This setup also allows us to easily create maps which we can then use to instantiate a level for the player to navigate, by having map cells refer to the IDs of the objects within the cells, so that the program may then place a new instance of that object at that place, with the final result for our database being Figure 2.5 Sprites ERD.

#### Constructors

Both the Entity and Block derived classes contain two different constructors, the first that generates an empty class, and the second creating a new instance from another instance of the class. The reason for the implementation of an empty instance instead of just using the second constructor is pretty self-explanatory: it would be impossible to create an instance from another instance if no other instances exist. As for the reason for the second constructor, it allows the copying of an existing object within the game, which means that instead of fetching the same attributes again for another instance of block 1, the program can just copy the first version fetched, since it only copies the base attributes of the object. This constructor could be useful in creating large groups of objects on purpose.

#### Maps-structure

With the tables being linked by the ID, we can now start to implement the designs for our map system. This will allow us to build a level from a stored table which contains all the data for our maps, where the data for each map is stored sequentially. Since we can identify through the foreign key ID, which instance of an object to set up where when building our map, we will uniquely identify each cell in our maps table using a composite primary key, which is made from combining the MapID (or map number), the x coordinate of the cell and the y coordinate. This will result in the following database, with each instance of a sprite having many possible map cells, whereas the tables have one to one relationships with each other.

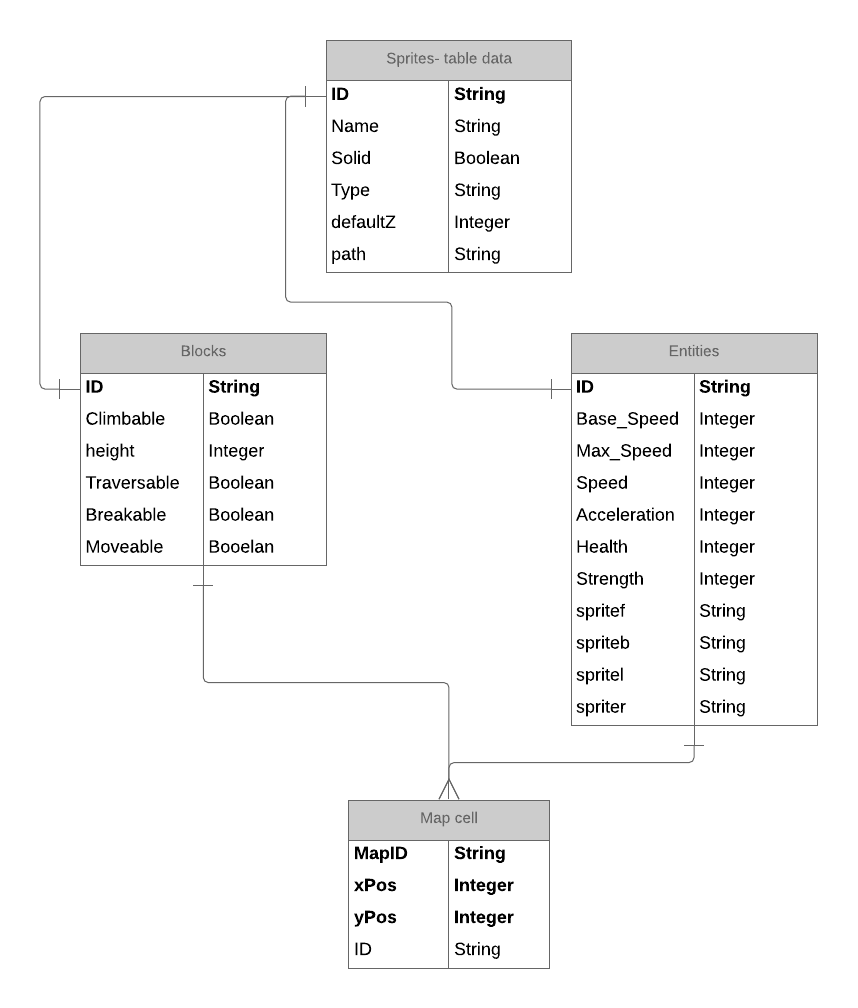


Figure . Sprites ERD

In this case the x and y co-ordinates dictate the tile position of the generated object, such that it is directly stored in the generated objects xPos and yPos attributes (see Variables in Sprite class), so as to make both the map database easier to manage, and implement our tiling system for ease of generation to prevent any overlapping of the objects generated, as well as making the solid platforms appearing next to each other appear as one singular large, uniform platform.

Whereas, our table for the maps will look like Figure 2.6 Map 1 example data, where the data of map 1's blocks are given as an example (since the full table is extremely large). To reduce the searching time and complexity, the table is arranged in order of mapID, such that when the table is being searched, the algorithm used to read it will stop after reaching the next mapID.



Figure . Map 1 example data

Since maps are easier to design and visualise by implementing an excel document where each cell represents a tile of the map, I then decided to implement a function that would be able to fetch these kinds of maps and store them in my maps database (this will be detailed in Fetch\_Data). Below in Figure 2.7, is the original design for the map inserted into the database as an example, with the horizontal positions representing positions from 0-9 from left to right respectively, and the vertical positions representing from 0-9, from bottom to top respectively.



Figure . Map n#1 excel spreadsheet

The reason for not just simply using a large variety of maps stored as such is that by storing all the values in a table as shown above is much faster to read through, as well as centralising all the data meaning there is no possibility of accidentally referencing the wrong file.

#### Use inside the game

When fetched the maps are to be stored in three dimensional arrays such that the player can navigate three dimensionally and blocks can have copies of themselves stored on different levels of the three dimensional array so that the player can detect their presence on these levels more easily, simplifying the collision algorithm used such that it only has to check on the same floor when looking for the objects around it. The use of multiple arrays can then be used to switch between maps, as the current level will have all its objects inside the array, meaning that to switch map, the game needs only to remove the objects inside the array from the game and switch to another array as the current level.

### Fetch\_Data

Since I needed functions that would read from my database and implement the data into the game, and that to do so, these functions would need to travers the tables for the sprite data as well as my maps. To this end, I generated the Fetch\_Data class, which would house my functions that I would use to access the sprites class, I did not put these functions inside the sprites class itself, as it would add unnecessary values (such as the paths to the resource documents), which an object such as an entity or a block would not need when generated, and would take up space. To compartmentalise and keep things separate, I built this class so that it would only deal with taking the inputs from the files and writing to the maps table.

For my database, I decided to use .csv files, due to my previous experiences with them and knowledge of how to deal with them, as well as the fact that excel can directly export to .csv, making it easier to implement.

Here, when fetching the data, it was important to make sure that it would meet point 3.b 4. and 5. of the specification. Specifically, when calling the reading function, I found it is imperative to make sure that it has the ability to call the correct directory for the tables, regardless of parent. To this end, I created the path\_g (path the directory for general tables, the ones that store the data) and path\_maps (path to the maps directory). These fields should call the Directory.GetParent() and .Parent enough times to find itself in the main project folder, as calling the Directory.GetCurrentDirectory() method will return the location of the Debug folder in the project bin folder. In order to build these strings properly, I will need to create the location field, which returns the directory to the bin/Debug folder inside the project folder. Since, it is necessary for these paths to remain fixed, I will make these attributes private to prevent errors, and call them using methods.

In order to then fetch the data from the database, I have decided to use multiple different functions called Fetch, fetch\_map, Add\_map and Getmap. This lead to the following class:

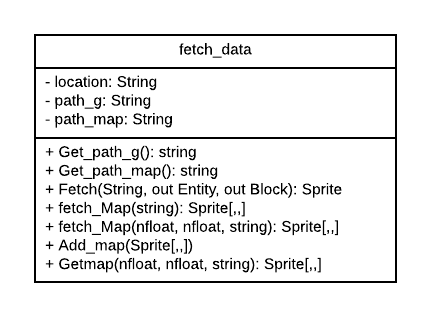


Figure . Fetch\_Data class diagram

#### Get\_path\_g

The purpose of Get\_path\_g is to make the directory stored in the private attribute path\_g accessible, without making the value stored inside susceptible to change to prevent directory errors when trying to fetch a file.

#### Get\_path\_map

the purpose of Get\_path\_map is the same as for Get\_path\_g, except with path\_map in this case.

#### Fetch

Fetch deals with reading from the tables containing the data from the sprites and creating a new instance of these objects based on their static properties. This function is absolutely necessary in the program and is called every time an instance of an object is created.

It uses the unique ID (acting as the primary key) of the desired sprite to fetch it from the database. Due to the small sizes of the tables and the fact that the objects have no particular order other than Type and order in which they were added, it implements a linear search. The reason for it returning two different types of sprite is to bypass the possible error of trying to create an instance of an object of the opposite type (ie trying to create an instance of a Block using an Entity's ID).

#### fetch\_Map

There are two different versions of fetch\_map, both begin by reading a map file like the one shown in Figure 2.7. This is then stored in a three dimensional sprite array (where an instance of each block is created when added to the array), where each position in the three dimensional array reflects the position in the game's tiling system referenced in the maps section above. The difference between the two methods is that the second fully sets up the array so that all the game needs to do is go through the array adding the objects as children to the SpriteKit Scene, whereas the first only creates the array without any other side effects. The reason for the implementation of the first despite Get\_map being a better version is that it allows the testing of the level before saving it to the maps table, as well as allowing the user to make changes to the map more easily, so that a non-functional map is not added to the table by mistake. The reason for taking in two nfloats as an addition is that these values are used by a method from another class it uses to set up the blocks.

The method goes through the map file line by line, going from the top row (row 9) and making its way down as it goes, ignoring cells which contain a 0. Otherwise, when it finds the ID (in this case acting as a foreign key) of the object, it then calls a function to create it and add it to the array which it then returns.

#### Add\_map

Add map takes a sprite array and adds it to the maps table shown in Figure 2.6 . The reason for implementing this class was that during testing it allowed the easy testing out of levels to then add them to the maps table, as well as making table input easier, faster and less tedious.

In order to add a map to the table, it first reads through the file to generate a brand new Map ID for it, and then proceeds to append the map data to the end by going through each cell of the sprites table (from 0,0 to 9,9 row by row), writing down the MapID, the x position, the y position and the ID for the block in that tile.

#### Get\_map

Get\_map is the function that directly fetches from the sprites table, and returns a sprite array, all the while creating instances of the blocks featured in the map. The reason for the use of this function is that although these actions are performed by fetching from a single map, Get\_map gets access to the table containing all of the maps, which means it takes less time to search through the table, since it will not have to check each empty cell.

In order to fetch the map, the program checks line by line if the mapID is equal to the one currently being looked for. Once the correct ID has been found, each new instance of the sprite will be created, fetched and fully developed, using the co-ordinates given as part of its Composite primary key to set up its location in the array. This will make it such that the game only needs to call the array, and add the objects as children to the Scene, to use it.

## Setting up the objects

In order to instantiate classes of the objects and set them up correctly for use in the game, since the fetch\_data section only deals with creating an instance of the class with the existing attributes, it was necessary to create a set of functions that would fully define the sprite by themselves, helping to place it, and managing the display settings when loading a level. I wanted to separate this aspect of class instantiation from the data manipulation since it was likely that this part of the coding would undergo multiple changes as the development progressed to go from basic movements to a more streamlined final result, so that making these functions separate would help to trace errors more easily, as well as making debugging a lot smoother.

Another reason for creating new classes for these functions was that I decided to compartmentalise my code so that it would be easier to navigate, track errors, as well as make changes, which is also why these classes were also separately. It is also due to preventative measures to prevent methods unique to either derived class from being used on the wrong class.

An important feature inside these classes was that they have strong error catching and handling procedures to prevent the program from crashing.

### p1\_setup

p1\_setup is the class used to fully set up a new instance of the character the player will control throughout the game as an instance of the Entity class, generating the paths for all the player sprites, and then using other functions to define its position, using information about the screen. Below is the class diagram:

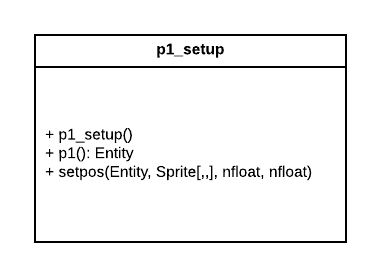


Figure . p1\_setup class diagram

Since the player sprite has to be treated much more uniquely from the other classes since it interacts with the player directly and is their input into the world, so to would need to be the functions that generate it. This is performed by the method p1(). p1() also verifies that there are no errors whilst creating the instance to prevent errors later in the program.

Since the player's hitbox will be different from the blocks, it will also be necessary to set the position of the player differently, so as to be able to define its movements inside the tiles more clearly, taking into account the character's hitbox and the output application frame. This is done by the setpos(Entity, Sprite[,,], nfloat, nfloat) method, which also deals with setting up the sprite's height.

### block\_setup

block\_setup is the class used to fully set up new instances of the Block class. It's goal like p1\_setup is to initiate an instance of the block class fully, although it works differently, since the blocks are unable to move by themselves. The class diagram is shown in full below in Figure 2.10.

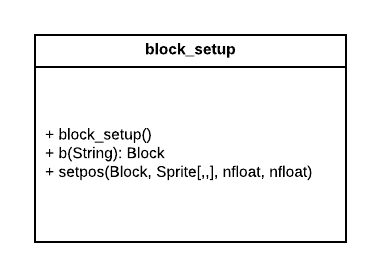


Figure . block\_setup class diagram

The goal of the method b(String) is to properly set up an instance of the block class while checking that there are no errors. Conversely the purpose of the setpos() method being to initiate the blocks' position, as well as generating the in-game sprite and placing it at the correct position. However, what differs this from the block class is that it uses a SpriteNode.zPosition algorithm to prevent geometrically incorrect overlapping of the sprites inside the game.

## Movement

The movement class is in effect the class that contains all the functions that the GameScene (see SpriteKit) file will need to call every time it needs to perform an action. It controls and manages the calculations for the players movements so that the player is able to move about freely, accelerate, collide and experience the Physical laws implemented inside the game. It is what allows the fulfilment of the entirety of 2. of the specification. In this case the movements are calculated using a sprite array containing all the sprites currently in the game, and the fields; actualX, actualY, xPos, yPos and zPos to control and direct movement, as well as making changes to the spriteNode attribute to change the sprite.

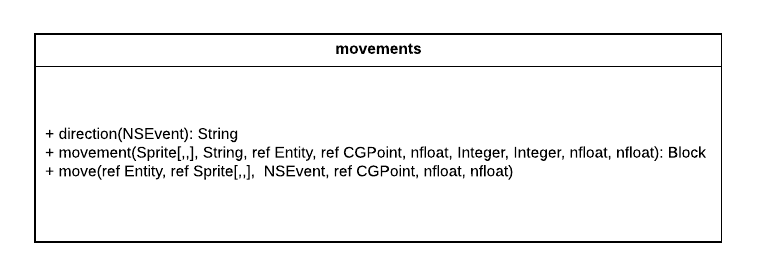


Figure . movements class diagram

### direction

Direction takes in an NSEvent as the event of a key being pressed and uses the key to determine the direction the player wants the character to move in, returning it as a string. The reason for this function is to allow easy remapping or change of the controls without altering the other functions, since the outputs it gives are always going to be the same, since I might need to change the way the controls are managed/taken in without changing the movement algorithms. For the controls, I decided to use w, a, s and d due to their widespread use in games as the most common default character controls for PC games.

### movement

The movement method covers parts of the specification 2.a, 2.b and implements part 3 by using the attributes of the blocks to dictate how the player interacts with them. It is also responsible for managing the collision detection of the game. It is built to act as a general function that performs the calculations for all the possible directions, taking the directions as parameters as two integers representing x and y vectors. For example, when moving vertically upwards; x will be 0 and y will be 1, when moving downwards, y becomes negative. When moving to the left, x will be 1 and y 0, when moving to the right, x becomes negative.

The function allows the player to navigate 2D space using actualX and actualY, as well as the standard movement unit for movement the function then calculates to see what tile the sprite is moving into and verifies if that tile is occupied, if not, the object is able to move freely. If it is occupied, the program then checks to see if the object occupying it is solid, to see if it can pass through. After this is done, and if the object is not solid, it checks whether it can interact with the object in any way or form, if not, the player is unable to move into the block.

The movement function also allows vertical movement by the player through climbing blocks and falling off of blocks, with the climbing attribute being checked whenever the player interacts with a block by walking into it. Falling is checked first in the function, through looking at the player's zPosition, and seeing if they are moving onto a square which contains a platform for the level they are currently on. If not, they will fall straight down until they either land on another platform or end up on the bottom floor.

All the while, after the possible movement has been chosen, the necessary changes to the player's attributes are made:

* actualX
* actualY
* xPos
* yPos
* spriteNode

The spriteNode itself is altered in multiple ways, with the zPosition for the display of the player being changed such that it correctly appears on top of or behind objects such that it reflects the fact that the player is either on top of a block, or standing behind it. The Texture attribute is also altered to change the sprite such that it reflects the direction in which it is facing.

The program also outputs a CGPoint called 'change' which are the x and y vectors the sprite will have to move by inside the game. This output is dependent on the Height and Width of the screen, which are input as nfloats.

### move

Move is the method eventually called by the game that calls both direction() and movement().

## SpriteKit project bundle

Since, when I started my project it was created as a spritekit bundle, it was important for me to learn the hierarchy used inside the project, as well as where to change the code such that I would be able to implement my game. I would also have to go the file directory to find where I would need to place my resources such that they would be used by the program.

The program itself with the output and user interface is managed inside the GameScene file in the Game\_Engine/Shared folder. It is then called by the GameViewController file in the same folder, which then builds the output to the Game\_Engine.Mac application. As a result of the outputs and inputs being dictated by the code in GameScene, this is where I will join up all my code and functions to manage the game.

### Project Directory

Since, the only relevant part of the project bundle was inside the Directory: Game\_Engine/shared, due to the fact that the other folders only contained files that were to adapt the project to iOS and Mac to allow it to run, with the code being directly copied from the shared folder.

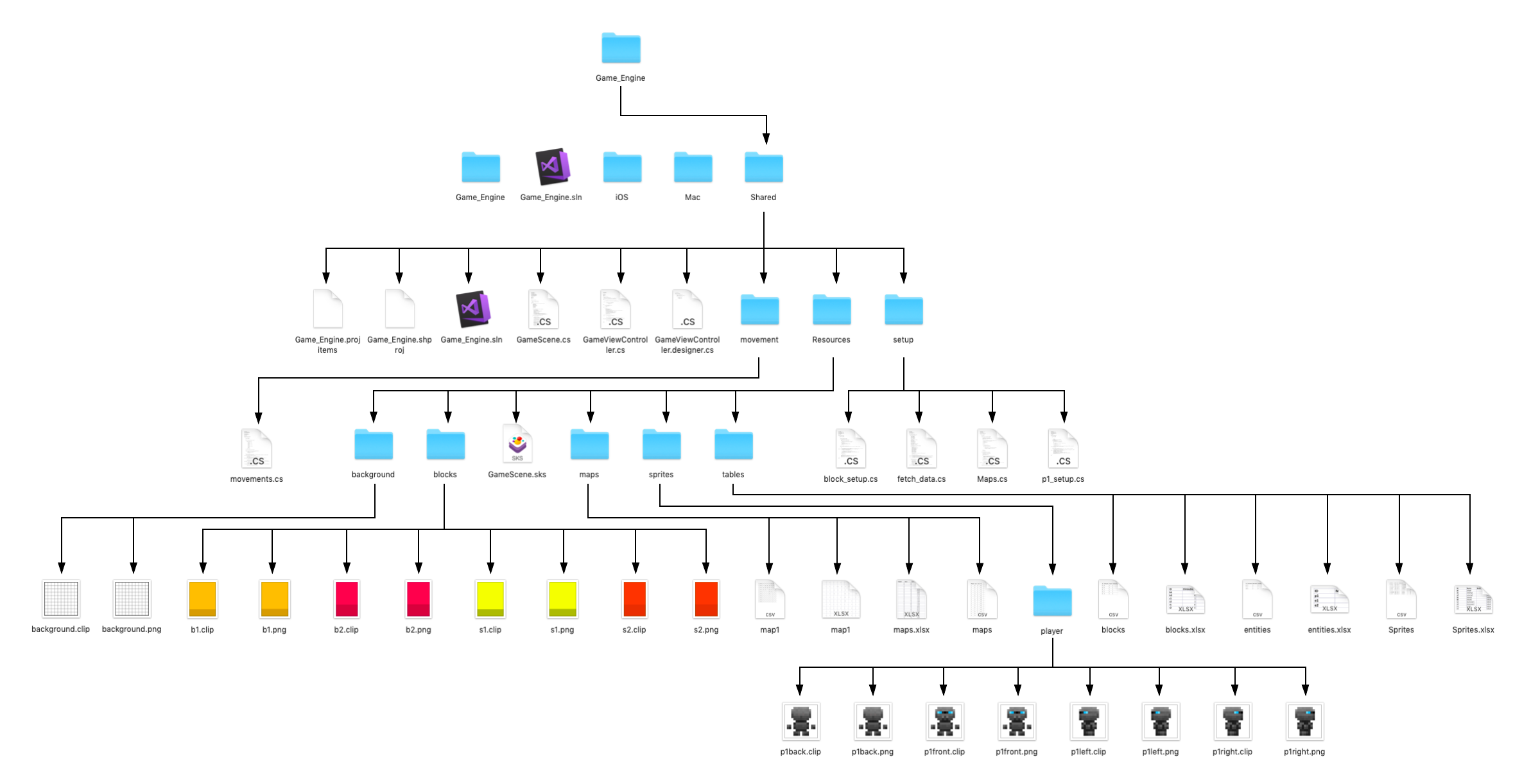


Figure . Project Directory

### Art assets and the GameEngine.projitems file

When the project is being built, it uses a .projitems file stored in the in order to declare which files to declare as bundle resources and which files to declare as Bundle resources, and which files to compile when building the program, as well as the hierarchy when compiling the project as a whole. Whereas Visual Studio would automatically add .cs files added to the project through the solution manager or created inside the project, this was not the case with the assets for the game. So, in order to be able to be able to call the assets int the program, it would be important to refer to each file inside the project directory inside the .projitems file under the 'BundleResource' Item Group.

### GameScene

# Index

## File Directory

Game\_Engine

├── Game\_Engine

├── Game\_Engine.sln

├── Mac

│   ├── AppDelegate.cs

│   ├── Assets.xcassets

│   │   ├── AppIcon.appiconset

│   │   │   ├── AppIcon-128.png

│   │   │   ├── AppIcon-128@2x.png

│   │   │   ├── AppIcon-16.png

│   │   │   ├── AppIcon-16@2x.png

│   │   │   ├── AppIcon-256.png

│   │   │   ├── AppIcon-256@2x.png

│   │   │   ├── AppIcon-32.png

│   │   │   ├── AppIcon-32@2x.png

│   │   │   ├── AppIcon-512.png

│   │   │   ├── AppIcon-512@2x.png

│   │   │   └── Contents.json

│   │   └── Contents.json

│   ├── Entitlements.plist

│   ├── Game\_Engine.Mac.csproj

│   ├── Info.plist

│   ├── Main.cs

│   ├── MainMenu.xib

│   ├── Resources

│   │   ├── b1.clip

│   │   ├── b1.png

│   │   ├── b2.clip

│   │   ├── b2.png

│   │   ├── background

│   │   │   ├── background.clip

│   │   │   └── background.png

│   │   ├── blocks

│   │   │   ├── b1.clip

│   │   │   ├── b1.png

│   │   │   ├── b2.clip

│   │   │   ├── b2.png

│   │   │   ├── s1.clip

│   │   │   ├── s1.png

│   │   │   ├── s2.clip

│   │   │   └── s2.png

│   │   ├── maps

│   │   │   ├── map1.csv

│   │   │   └── map1.xlsx

│   │   ├── s1.clip

│   │   ├── s1.png

│   │   ├── s2.clip

│   │   ├── s2.png

│   │   ├── sprites

│   │   │   └── player

│   │   │   ├── p1back.clip

│   │   │   ├── p1back.png

│   │   │   ├── p1front.clip

│   │   │   ├── p1front.png

│   │   │   ├── p1left.clip

│   │   │   ├── p1left.png

│   │   │   ├── p1right.clip

│   │   │   └── p1right.png

│   │   └── tables

│   │   ├── Sprites.csv

│   │   ├── Sprites.xlsx

│   │   ├── blocks.csv

│   │   ├── blocks.xlsx

│   │   ├── entities.csv

│   │   └── entities.xlsx

│   ├── bin

│   │   └── Debug

│   │   ├── Game\_Engine.Mac.app

│   │   │   └── Contents

│   │   │   ├── Info.plist

│   │   │   ├── MacOS

│   │   │   │   ├── Game\_Engine.Mac

│   │   │   │   └── Game\_Engine.Mac.dSYM

│   │   │   │   └── Contents

│   │   │   │   ├── Info.plist

│   │   │   │   └── Resources

│   │   │   │   └── DWARF

│   │   │   │   └── Game\_Engine.Mac

│   │   │   ├── MonoBundle

│   │   │   │   ├── Accessibility.dll

│   │   │   │   ├── Accessibility.pdb

│   │   │   │   ├── Game\_Engine.Mac.exe

│   │   │   │   ├── Game\_Engine.Mac.pdb

│   │   │   │   ├── Mono.Posix.dll

│   │   │   │   ├── Mono.Posix.pdb

│   │   │   │   ├── Mono.Security.dll

│   │   │   │   ├── Mono.Security.pdb

│   │   │   │   ├── Mono.WebBrowser.dll

│   │   │   │   ├── Mono.WebBrowser.pdb

│   │   │   │   ├── System.Configuration.dll

│   │   │   │   ├── System.Configuration.pdb

│   │   │   │   ├── System.Core.dll

│   │   │   │   ├── System.Core.pdb

│   │   │   │   ├── System.Data.dll

│   │   │   │   ├── System.Data.pdb

│   │   │   │   ├── System.Drawing.dll

│   │   │   │   ├── System.Drawing.pdb

│   │   │   │   ├── System.EnterpriseServices.dll

│   │   │   │   ├── System.EnterpriseServices.pdb

│   │   │   │   ├── System.Net.Http.dll

│   │   │   │   ├── System.Net.Http.pdb

│   │   │   │   ├── System.Numerics.dll

│   │   │   │   ├── System.Numerics.pdb

│   │   │   │   ├── System.Runtime.Serialization.Formatters.Soap.dll

│   │   │   │   ├── System.Runtime.Serialization.Formatters.Soap.pdb

│   │   │   │   ├── System.Security.dll

│   │   │   │   ├── System.Security.pdb

│   │   │   │   ├── System.Transactions.dll

│   │   │   │   ├── System.Transactions.pdb

│   │   │   │   ├── System.Windows.Forms.dll

│   │   │   │   ├── System.Windows.Forms.pdb

│   │   │   │   ├── System.Windows.dll

│   │   │   │   ├── System.Xml.dll

│   │   │   │   ├── System.Xml.pdb

│   │   │   │   ├── System.dll

│   │   │   │   ├── System.pdb

│   │   │   │   ├── Xamarin.Mac.dll

│   │   │   │   ├── Xamarin.Mac.pdb

│   │   │   │   ├── config

│   │   │   │   ├── libMonoPosixHelper.dylib

│   │   │   │   ├── libcairo.2.dylib

│   │   │   │   ├── libcroco-0.6.3.dylib

│   │   │   │   ├── libexpat.1.6.7.dylib

│   │   │   │   ├── libffi.6.dylib

│   │   │   │   ├── libfontconfig.1.dylib

│   │   │   │   ├── libfreetype.6.dylib

│   │   │   │   ├── libgdiplus.0.dylib

│   │   │   │   ├── libgdiplus.dylib -> libgdiplus.0.dylib

│   │   │   │   ├── libgdk\_pixbuf-2.0.0.dylib

│   │   │   │   ├── libgdk\_pixbuf-2.0.dylib -> libgdk\_pixbuf-2.0.0.dylib

│   │   │   │   ├── libgif.4.1.6.dylib

│   │   │   │   ├── libgio-2.0.0.dylib

│   │   │   │   ├── libglib-2.0.0.dylib

│   │   │   │   ├── libglib-2.0.dylib -> libglib-2.0.0.dylib

│   │   │   │   ├── libgmodule-2.0.0.dylib

│   │   │   │   ├── libgobject-2.0.0.dylib

│   │   │   │   ├── libgobject-2.0.dylib -> libgobject-2.0.0.dylib

│   │   │   │   ├── libgthread-2.0.0.dylib

│   │   │   │   ├── libintl.8.dylib

│   │   │   │   ├── libintl.dylib -> libintl.8.dylib

│   │   │   │   ├── libjpeg.8.dylib

│   │   │   │   ├── libmono-btls-shared.dylib

│   │   │   │   ├── libpango-1.0.0.dylib

│   │   │   │   ├── libpangocairo-1.0.0.dylib

│   │   │   │   ├── libpixman-1.0.dylib

│   │   │   │   ├── libpng14.14.dylib

│   │   │   │   ├── librsvg-2.2.dylib

│   │   │   │   ├── librsvg-2.dylib -> librsvg-2.2.dylib

│   │   │   │   ├── libtiff.5.dylib

│   │   │   │   ├── libxml2.2.dylib

│   │   │   │   ├── machine.config

│   │   │   │   ├── mscorlib.dll

│   │   │   │   └── mscorlib.pdb

│   │   │   ├── PkgInfo

│   │   │   └── Resources

│   │   │   ├── GameScene.sks

│   │   │   ├── MainMenu.nib

│   │   │   ├── background

│   │   │   │   ├── background.clip

│   │   │   │   └── background.png

│   │   │   ├── blocks

│   │   │   │   ├── b1.clip

│   │   │   │   ├── b1.png

│   │   │   │   ├── b2.clip

│   │   │   │   ├── b2.png

│   │   │   │   ├── s1.clip

│   │   │   │   ├── s1.png

│   │   │   │   ├── s2.clip

│   │   │   │   └── s2.png

│   │   │   ├── maps

│   │   │   │   ├── map1.csv

│   │   │   │   └── map1.xlsx

│   │   │   ├── runtime-options.plist

│   │   │   ├── sprites

│   │   │   │   └── player

│   │   │   │   ├── p1back.clip

│   │   │   │   ├── p1back.png

│   │   │   │   ├── p1front.clip

│   │   │   │   ├── p1front.png

│   │   │   │   ├── p1left.clip

│   │   │   │   ├── p1left.png

│   │   │   │   ├── p1right.clip

│   │   │   │   └── p1right.png

│   │   │   └── tables

│   │   │   ├── Sprites.csv

│   │   │   ├── Sprites.xlsx

│   │   │   ├── blocks.csv

│   │   │   ├── blocks.xlsx

│   │   │   ├── entities.csv

│   │   │   └── entities.xlsx

│   │   ├── Game\_Engine.Mac.exe

│   │   ├── Game\_Engine.Mac.pdb

│   │   ├── stderr.log

│   │   └── stdout.log

│   └── obj

│   └── x86

│   └── Debug

│   ├── Game\_Engine.Mac.csproj.CoreCompileInputs.cache

│   ├── Game\_Engine.Mac.csproj.FileListAbsolute.txt

│   ├── Game\_Engine.Mac.csprojAssemblyReference.cache

│   ├── Game\_Engine.Mac.exe

│   ├── Game\_Engine.Mac.pdb

│   ├── actool

│   │   ├── \_BundleResourceWithLogicalName.items

│   │   ├── \_PartialAppManifest.items

│   │   ├── asset-manifest.plist

│   │   ├── bundle

│   │   └── cloned-assets

│   │   └── Assets.xcassets

│   │   ├── AppIcon.appiconset

│   │   │   ├── AppIcon-128.png

│   │   │   ├── AppIcon-128@2x.png

│   │   │   ├── AppIcon-16.png

│   │   │   ├── AppIcon-16@2x.png

│   │   │   ├── AppIcon-256.png

│   │   │   ├── AppIcon-256@2x.png

│   │   │   ├── AppIcon-32.png

│   │   │   ├── AppIcon-32@2x.png

│   │   │   ├── AppIcon-512.png

│   │   │   ├── AppIcon-512@2x.png

│   │   │   └── Contents.json

│   │   └── Contents.json

│   ├── ibtool

│   │   └── MainMenu.nib

│   ├── ibtool-manifests

│   │   └── MainMenu.nib

│   ├── mmp-cache

│   │   ├── arguments

│   │   ├── entry-points.txt

│   │   ├── exported-symbols-list

│   │   └── main.m

│   └── response-file.rsp

├── Shared

│   ├── GameScene.cs

│   ├── GameViewController.cs

│   ├── GameViewController.designer.cs

│   ├── Game\_Engine.projitems

│   ├── Game\_Engine.shproj

│   ├── Game\_Engine.sln

│   ├── Resources

│   │   ├── GameScene.sks

│   │   ├── background

│   │   │   ├── background.clip

│   │   │   └── background.png

│   │   ├── blocks

│   │   │   ├── b1.clip

│   │   │   ├── b1.png

│   │   │   ├── b2.clip

│   │   │   ├── b2.png

│   │   │   ├── s1.clip

│   │   │   ├── s1.png

│   │   │   ├── s2.clip

│   │   │   └── s2.png

│   │   ├── maps

│   │   │   ├── map1.csv

│   │   │   ├── map1.xlsx

│   │   │   ├── maps.csv

│   │   │   └── maps.xlsx

│   │   ├── sprites

│   │   │   └── player

│   │   │   ├── p1back.clip

│   │   │   ├── p1back.png

│   │   │   ├── p1front.clip

│   │   │   ├── p1front.png

│   │   │   ├── p1left.clip

│   │   │   ├── p1left.png

│   │   │   ├── p1right.clip

│   │   │   └── p1right.png

│   │   └── tables

│   │   ├── Sprites.csv

│   │   ├── Sprites.xlsx

│   │   ├── blocks.csv

│   │   ├── blocks.xlsx

│   │   ├── entities.csv

│   │   └── entities.xlsx

│   ├── movement

│   │   └── movements.cs

│   └── setup

│   ├── Maps.cs

│   ├── block\_setup.cs

│   ├── fetch\_data.cs

│   └── p1\_setup.cs

└── iOS

├── AppDelegate.cs

├── Assets.xcassets

│   ├── AppIcon.appiconset

│   │   └── Contents.json

├── Entitlements.plist

├── Game\_Engine.IOS.csproj

├── Info.plist

├── LaunchScreen.storyboard

├── Main.cs

├── Main.storyboard

├── Resources

├── bin

│   └── iPhoneSimulator

│   └── Debug

│   ├── Game\_Engine.IOS.app

│   │   ├── GameScene.sks

│   │   ├── Game\_Engine.IOS

│   │   ├── Game\_Engine.IOS.exe

│   │   ├── Game\_Engine.IOS.pdb

│   │   ├── Info.plist

│   │   ├── LaunchScreen.storyboardc

│   │   │   ├── 01J-lp-oVM-view-Ze5-6b-2t3.nib

│   │   │   ├── Info.plist

│   │   │   └── UIViewController-01J-lp-oVM.nib

│   │   ├── Main.storyboardc

│   │   │   ├── BV1-FR-VrT-view-3se-qz-xqx.nib

│   │   │   ├── Info.plist

│   │   │   └── UIViewController-BV1-FR-VrT.nib

│   │   ├── Mono.Security.dll

│   │   ├── Mono.Security.pdb

│   │   ├── MonoTouchDebugConfiguration.txt

│   │   ├── PkgInfo

│   │   ├── Settings.bundle

│   │   │   └── Root.plist

│   │   ├── System.Core.dll

│   │   ├── System.Core.pdb

│   │   ├── System.Xml.dll

│   │   ├── System.Xml.pdb

│   │   ├── System.dll

│   │   ├── System.pdb

│   │   ├── Xamarin.iOS.dll

│   │   ├── Xamarin.iOS.pdb

│   │   ├── background

│   │   │   └── background.png

│   │   ├── blocks

│   │   │   ├── b1.png

│   │   │   ├── b2.png

│   │   │   ├── s1.png

│   │   │   └── s2.png

│   │   ├── mscorlib.dll

│   │   ├── mscorlib.pdb

│   │   ├── runtime-options.plist

│   │   └── sprites

│   │   └── player

│   │   ├── p1back.png

│   │   ├── p1front.png

│   │   ├── p1left.png

│   │   └── p1right.png

│   ├── Game\_Engine.IOS.exe

│   ├── Game\_Engine.IOS.pdb

│   └── mtouch.stamp

└── obj

└── iPhoneSimulator

└── Debug

├── Game\_Engine.IOS.csproj.CoreCompileInputs.cache

├── Game\_Engine.IOS.csproj.FileListAbsolute.txt

├── Game\_Engine.IOS.csprojAssemblyReference.cache

├── Game\_Engine.IOS.exe

├── Game\_Engine.IOS.pdb

├── actool

│   ├── \_BundleResourceWithLogicalName.items

│   ├── \_PartialAppManifest.items

│   ├── asset-manifest.plist

│   ├── bundle

│   ├── cloned-assets

│   │   └── Assets.xcassets

│   │   ├── AppIcon.appiconset

│   │   │   └── Contents.json

│   │   └── Contents.json

│   └── partial-info.plist

├── coremlc

│   ├── \_BundleResourceWithLogicalName.items

│   └── \_PartialAppManifest.items

├── ibtool

│   ├── LaunchScreen.storyboardc

│   │   ├── 01J-lp-oVM-view-Ze5-6b-2t3.nib

│   │   ├── Info.plist

│   │   └── UIViewController-01J-lp-oVM.nib

│   ├── Main.storyboardc

│   │   ├── BV1-FR-VrT-view-3se-qz-xqx.nib

│   │   ├── Info.plist

│   │   └── UIViewController-BV1-FR-VrT.nib

│   └── \_BundleResourceWithLogicalName.items

├── ibtool-link

│   ├── LaunchScreen.storyboardc

│   │   ├── 01J-lp-oVM-view-Ze5-6b-2t3.nib

│   │   ├── Info.plist

│   │   └── UIViewController-01J-lp-oVM.nib

│   └── Main.storyboardc

│   ├── BV1-FR-VrT-view-3se-qz-xqx.nib

│   ├── Info.plist

│   └── UIViewController-BV1-FR-VrT.nib

├── ibtool-manifests

│   ├── LaunchScreen.storyboardc

│   ├── Main.storyboardc

│   └── link

├── mtouch-cache

│   ├── arguments

│   ├── assembly-references.txt

│   └── entry-points.txt

├── mtouch-symbols.list

├── optimized

│   ├── background

│   │   └── background.png

│   ├── blocks

│   │   ├── b1.png

│   │   ├── b2.png

│   │   ├── s1.png

│   │   └── s2.png

│   └── sprites

│   └── player

│   ├── p1back.png

│   ├── p1front.png

│   ├── p1left.png

│   └── p1right.png

└── response-file.rsp

Figure . Project file directory

# Bibliography

Apple inc. 2018. *About OpenGL for OS X.* 4 June. Accessed March 17, 2019. https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl\_intro/opengl\_intro.html#//apple\_ref/doc/uid/TP40001987-CH207-TP9.

Apple inc. 2019. *Metal.* 11 March. Accessed March 17, 2019. https://developer.apple.com/metal/.

Bulbapedia. 2018. *Tweaking.* 3 May. Accessed March 17, 2019. https://bulbapedia.bulbagarden.net/wiki/Tweaking.

Fandom. 2019. *Metroid.* 9 February. Accessed March 17, 2019. https://vgsales.fandom.com/wiki/Metroid.

—. 2019. *The Legend of Zelda.* 31 January. Accessed March 17, 2019. https://vgsales.fandom.com/wiki/The\_Legend\_of\_Zelda.

Kerr, David. 2014. *OpenGL in .NET - Getting Started.* 8 April. Accessed March 17, 2019. https://www.codeproject.com/Articles/749543/OpenGL-in-NET-Getting-Started.

Nintendo. 2018. *Top Selling Title Sales Units.* 31 March. Accessed March 17, 2019. https://www.nintendo.co.jp/ir/en/finance/software/ds.html.

OpenGL. 2018. *OpenGL coding resources.* 8 September. Accessed March 17, 2019. https://www.opengl.org/resources/.

Stolpe, Philip. 2019. *Monogame.* 12 March. Accessed March 17, 2019. http://www.monogame.net.

u/Just\_Farming\_DownVs. 2017. *What can I use for basic, fast 2D graphics?* 19 February. Accessed March 17, 2019. https://www.reddit.com/r/csharp/comments/5uvyw9/what\_can\_i\_use\_for\_basic\_fast\_2D\_graphics/.

Wijman, Tom. 2018. *Mobile Revenues Account for More Than 50% of the Global Games Market as It Reaches $137.9 Billion in 2018.* Newzoo. 30 April. Accessed March 16, 2019. https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/.

Wikipedia. 2019. *DirectX.* 15 March. Accessed March 17, 2019. https://en.wikipedia.org/wiki/DirectX.

—. 2019. *Film Industry.* 7 March. Accessed March 16, 2019. https://en.wikipedia.org/wiki/Film\_industry.

—. 2019. *OpenGL.* 5 March. Accessed March 17, 2019. https://en.wikipedia.org/wiki/OpenGL.

Zeffrit. 2013. *Looking for the best Graphic Library to use with C# (No Xna Please).* 3 December. Accessed March 17, 2019. https://www.gamedev.net/forums/topic/650755-looking-for-the-best-graphic-library-to-use-with-c-no-xna-please/.

1. This section ties in with "Intuitive" [↑](#footnote-ref-1)
2. Tied in with Mechanics [↑](#footnote-ref-2)
3. See paragraph 4 in The Legend of Zelda (1986) [↑](#footnote-ref-3)
4. The player can move faster by running or riding on a bike on solid ground [↑](#footnote-ref-4)