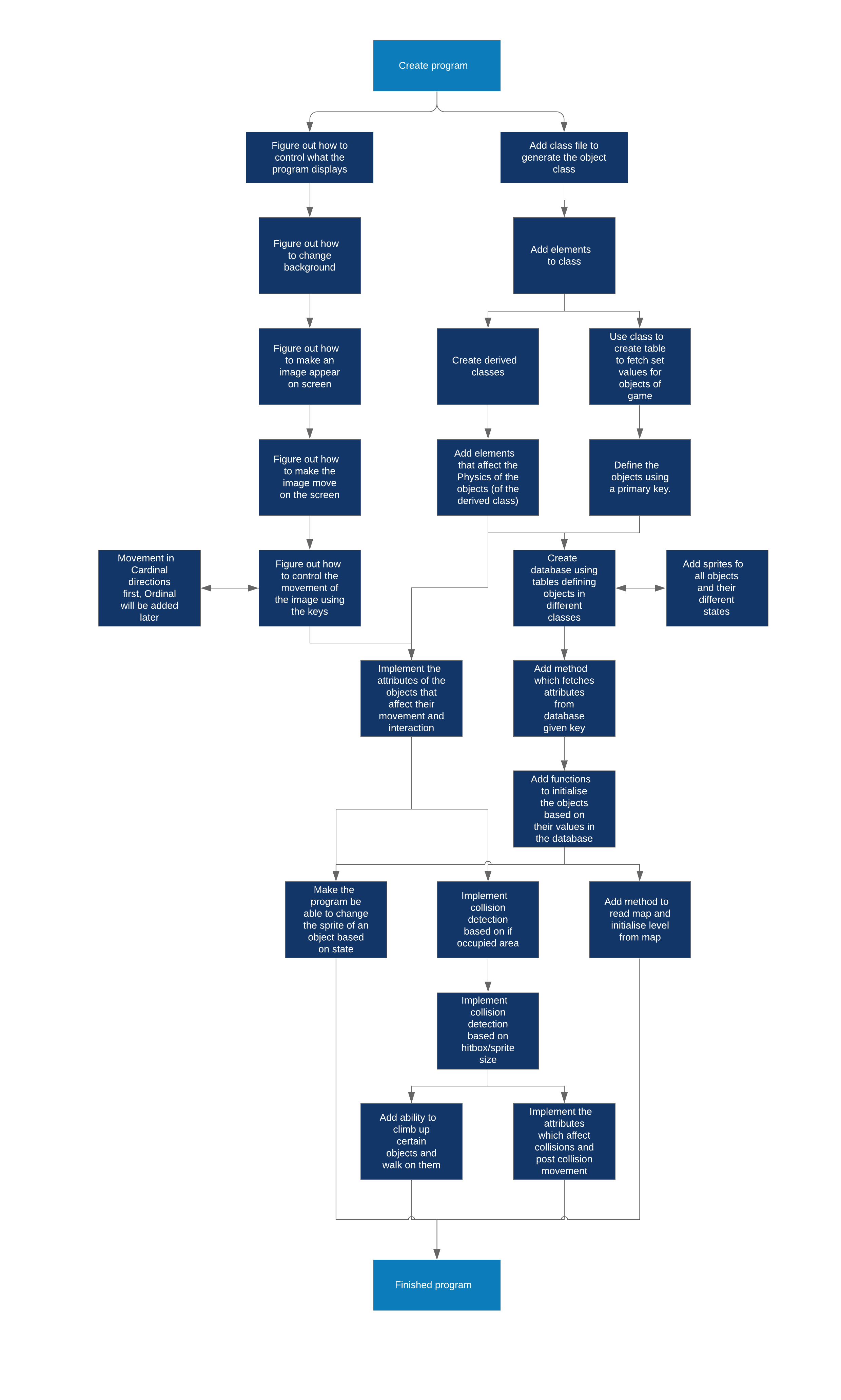
# Design

## 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

### 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.

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. |

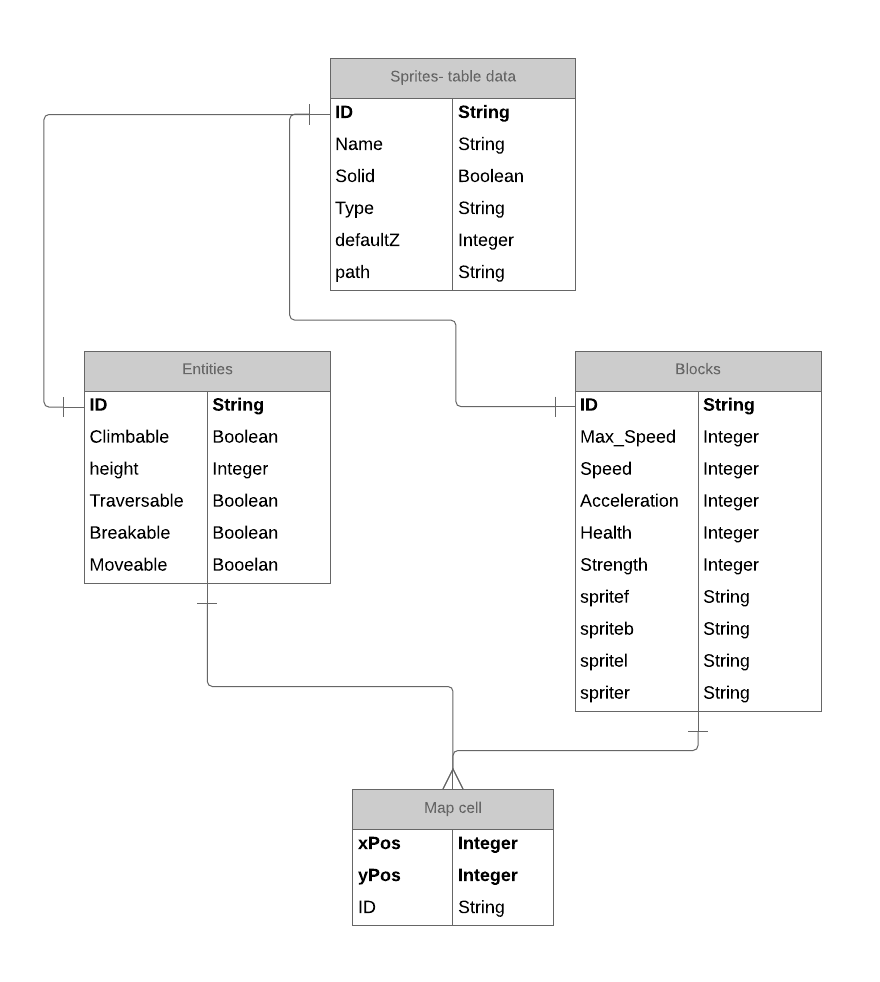
#### Variables

|  |  |  |
| --- | --- | --- |
| 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 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 and a value acceleration to meet part 2.b.i



# 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/.

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.

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/.

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

—. 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/.