For our second main assignment in the ‘Games in C++’ module, we were tasked with coding our own platformer game, continuing to use C++ and the SFML code library, like we did for our ‘Pong’ assignment.

Games in C++ – Assignment Exercise 2 – Planning Evidence document  
Creating a platformer game  
Dominik Heiler - 23015707

**GitHub Username:** DominikHHH

**Link to Source Code:** <https://github.com/UWEGames-GiC/platformer-23-24-DominikHHH>

Features:

* Player physics for running, jumping, and bouncing off enemies
* Tile-map-based level loading using external ‘.txt’ file-loading
* Tile-map-based collision detection
* Multiple level loading
* Multiple coin objects that give score points to the player
* Multiple obstacle objects that cause the player to lose a life
* Camera transition system
* Pausing/unpausing
* Game window resizing

Controls:

* A & D – Moving left and right
* Space – Jumping
* Enter – Starting the game, pausing/unpausing
* Esc – Closing the game

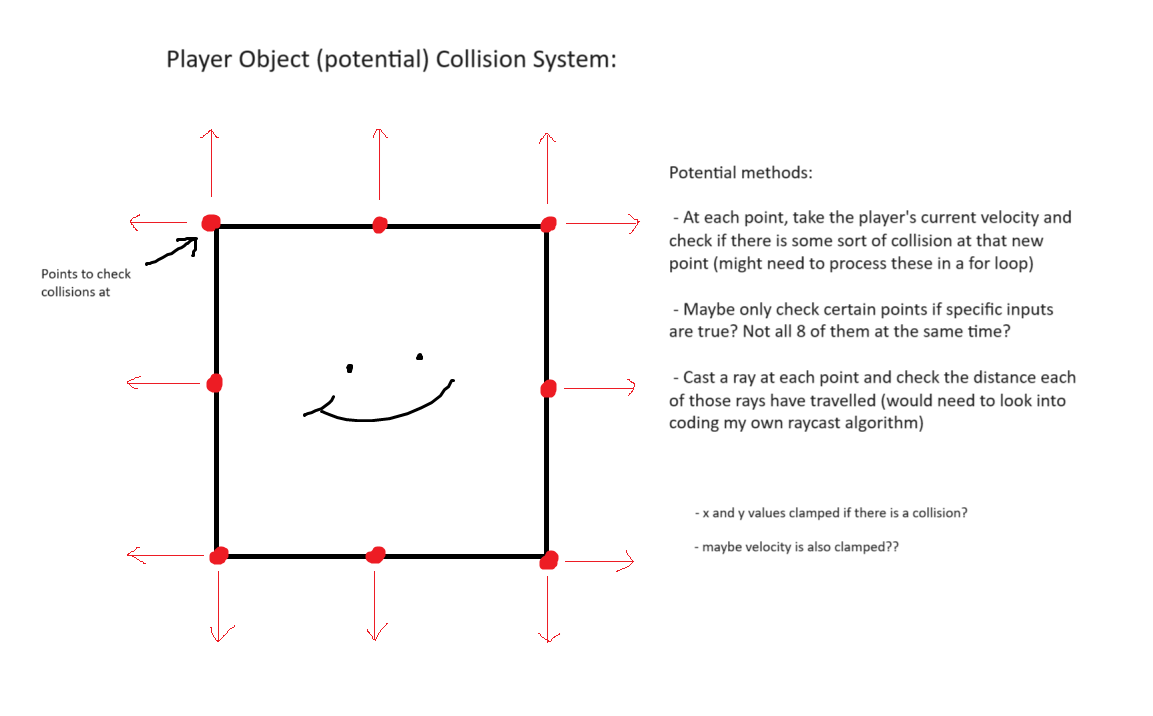
Presented below are all the planning materials that I made for myself while I was coding the actual game assets at the same time.

A screenshot of a computer

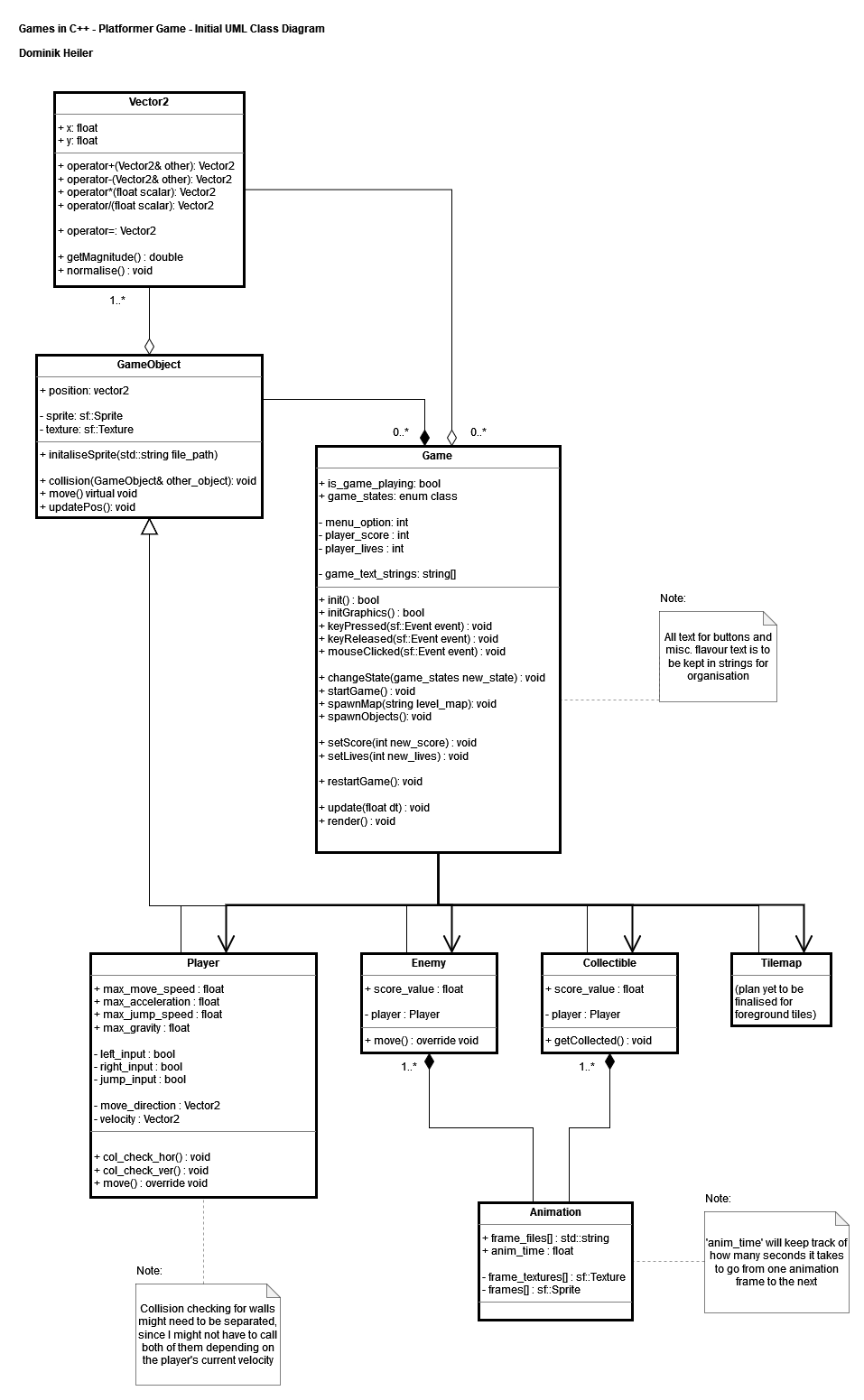
Description automatically generated

A screenshot of a computer

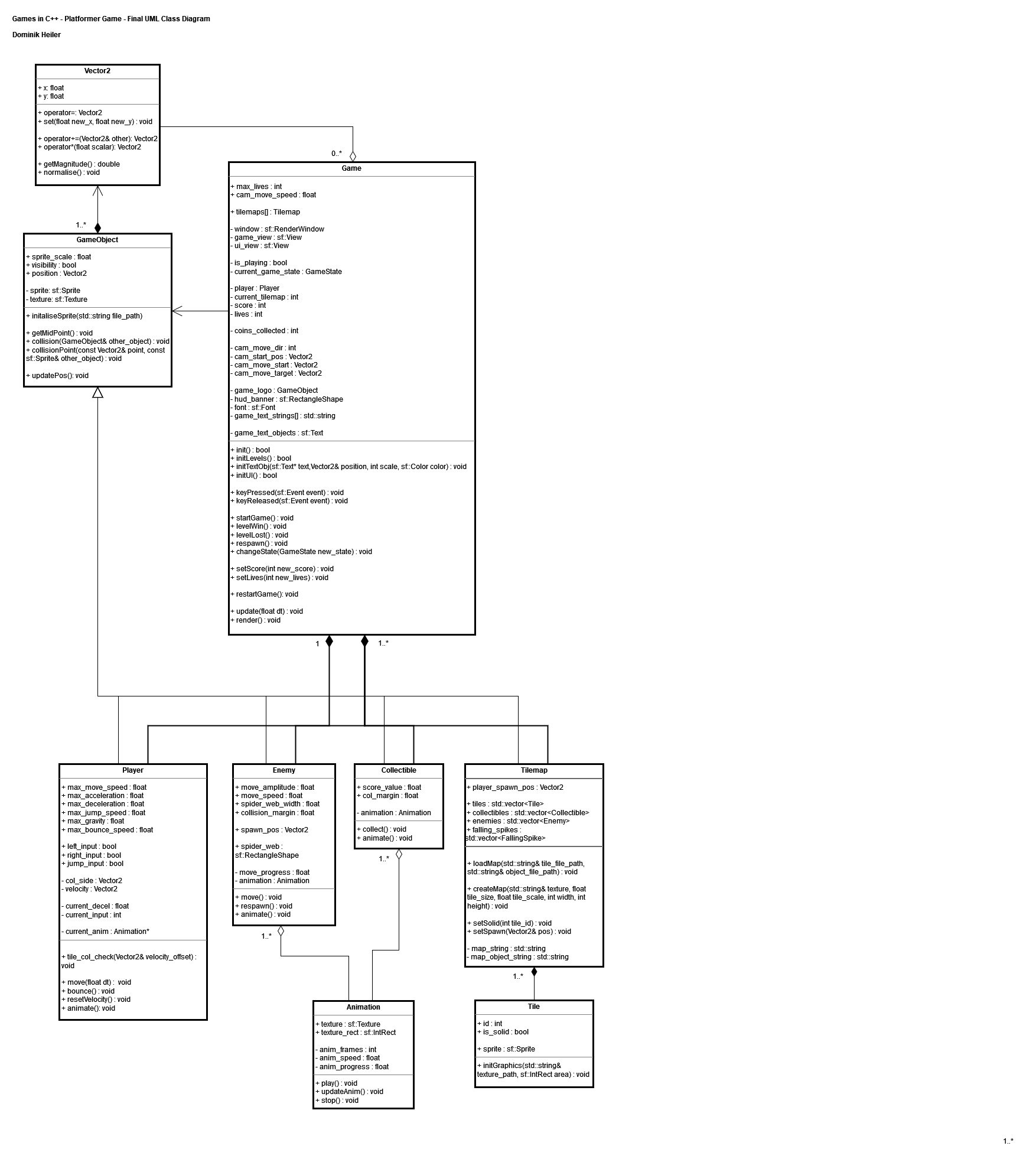
Description automatically generated



(Above: screenshots of my notes, pseudo-code snippets, and doodles that I made for myself at the start of, and during, the project’s development, having used ‘Notepad’ and ‘MSPaint’ for future ease of access. The doodles, in particular, were made just for myself to keep track of what I needed to work on, so they are quite rough in some places. The notes were also made with a slightly different version of the game in mind, with numerous ambitious features needing to be scrapped or redesigned by the time the submission deadline was starting to approach.)



(Above: the UML class diagram that I made at the *start* of the project using ‘draw.io,’ having attempted to plan out as much of the base game as possible, so that I will not have to make too many changes to my code structure during production)



(Above: the second UML class diagram that I made near the *end* of the project, once again using ‘draw.io,’ this time with correct lines connecting all the classes together, with how I had not fully grasped how to draw and design UML class diagrams effectively at the start of this project)

Some notable changes, tweaks, and differences that I made to this code structure throughout development include the following:

* I am happy to say that the rough structure of the classes themselves did not change too much, with how I was able to use my GameObject class to define basic object behaviours in my game, while also using supplementary classes like ‘Tileset’ and ‘Animation’ to add more features to my game overall. The only addition I made, later in development, was with the ‘Tile’ class, with how I soon found out that storing all the complicated tile data inside of a ‘Tileset’ instance would have led to unnecessarily complicated code. As a result, a dedicated ‘Tile’ class was created to store simple pieces of data, such as a Boolean value for whether the tile is solid, as well as the ‘sf::Texture’ and ‘sf::Sprite’ values that should be displayed in the Render loop itself.
* The ‘Enemy’ and ‘Collectible’ classes were initially meant to be used as base classes for more specific types of enemy and collectible objects. For example, the ‘Enemy’ class would have been used to create ‘Snake’ and ‘Spider’ objects, each with their own behaviours. In the end, I had to adapt these classes to fit the specific needs of the levels that I designed up to that point, but, if a few of the more specific variables were removed, I still think they could be repurposed as base/parent classes, and because of that, I’m fairly satisfied with the way I structured my code here.
* The player’s collision system was what changed the most during development, and so the ‘Player’ class itself also changed a lot as well. I am happy to have produced all the collision detection plans that I did at the start of the project, as it got me thinking heavily about all the diverse ways the player will potentially be interacting with the environment, but in the end, the final system had to be simplified. Certain changes included:
  + Scraping the individual point-checking system, and instead taking the player’s entire collision box and offsetting it by various positions to check for collisions. While, in hindsight, this would’ve been a much more sophisticated and tweakable system for all sorts of collision detecting, I was worried, at the start, that checking for corner points on the player object would make it difficult to see whether the player is colliding with a wall or a floor/ceiling, and so I was hesitant to try to implement this system once I started actually coding my game.
  + This also explains why the two separate collision functions that I wrote were turned into one collision check, that I used in conjunction with my ‘move’ function as well.
  + A few additional physics variables were added to make the player controls feel a bit more natural and satisfying. These included ‘max\_acceleration’ and ‘max\_deceleration’ for when running on the floor, as well as a ‘bounce speed’ variable, for when you jump on and bounce off enemies in the game, though this ended up only being used for the spider enemies in the final game demo.
* Some other notable changes to the game’s code include changes to the ‘Animation’ class, though the general plan stayed the same. I initially had the idea of having each animation frame made into its own ‘sf::Sprite’ value (with an accompanying ‘sf::Texture’ object as well), before finding out that SFML supports loading in sprite-sheets without the entire area of the texture being rendered. From there, the logic was mostly the same, with how I had variables set up for how quickly the animation would play, as well as functions for stopping and starting the animations, should they need to be paused at any point during gameplay.
* If I were to, hypothetically, work on this project for longer, these additional functions would give me much more control to start and stop playing animations during gameplay if, for example, some sort of cutscene event was playing, where player inputs are not being processed and objects are being instructed to move around automatically.
* Plans for the game’s tile-map were not finalised at this early point of development, as I was not sure of what features SFML was capable of at the time, not to mention how we were also told that tile-map loading would be covered in a future lecture at some point, and so I held off on planning too far ahead for now. The SFML documentation website had example code where tiles would be loaded in using vertex arrays and then creating one big Texture file that I would render. In the end, though, I found it much easier to make each tile its own ‘sf::Sprite’ object, so that I could process collision detection more easily, as well as structure my code in a cleaner manner.
* For a larger scale project, this would also give me more freedom to enable and disable certain tiles if I wanted to, and potentially even animate them to make the game environment more immersive and interesting, but these features were never implemented, due to the smaller scope of the project at hand.

  
(Above: some basic mock-ups featuring simple visuals that I drew in ‘Aseprite.’ These assets were from an old personal project of mine, unrelated to the assignment, which just so happened to also be for a platformer game, and so they ended up being re-used here)









(Above: various screenshots of the final game running in action)

C++ code and graphical assets made by Dominik Heiler

Software used: CLion, Gitkraken Aseprite, draw.io

‘Factor’ font designed and drawn by Damien Guard (<https://damieng.com/typography/zx-origins/factor/>)