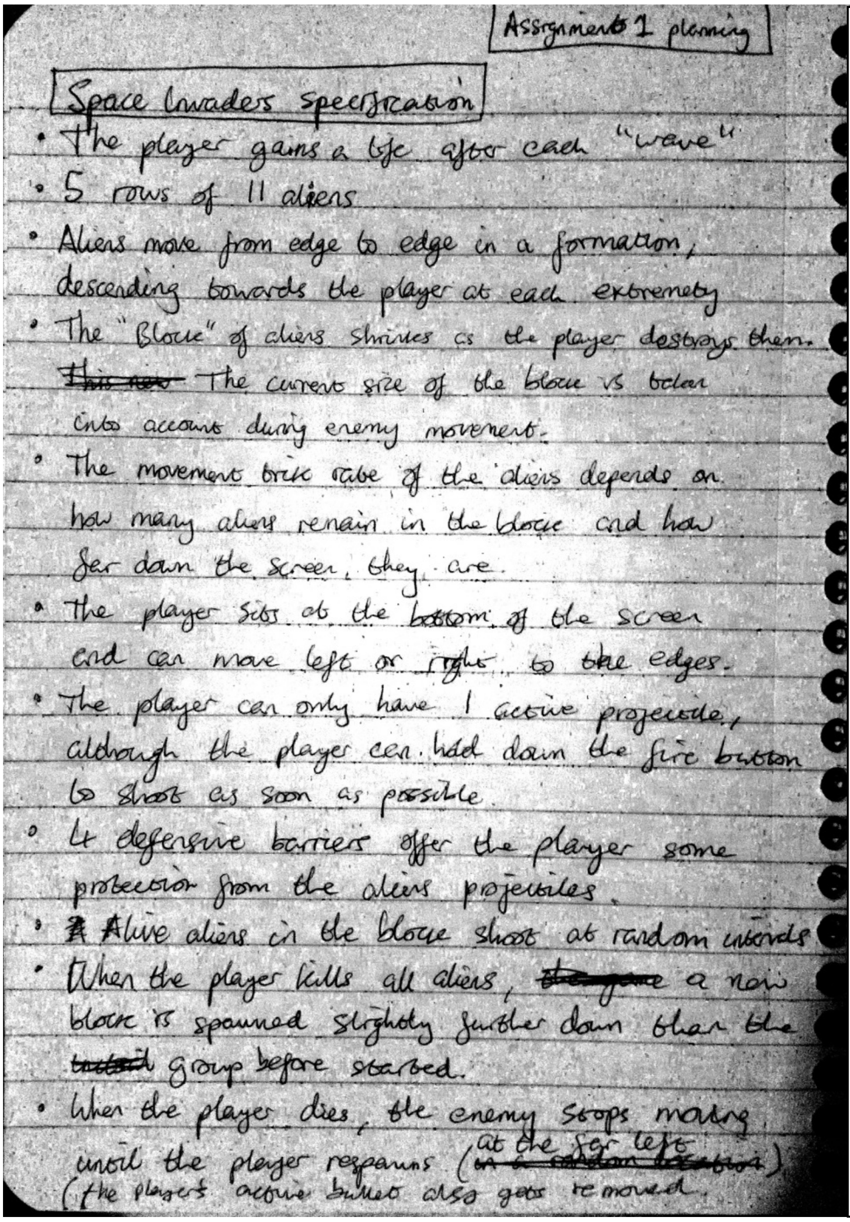
LOW-LEVEL PROGRAMMING ASSIGNMENT 1 - DOCUMENTATION

PLANNING

NOTEBOOK SCANS


Assignment 1 planning

Space Invaders specification

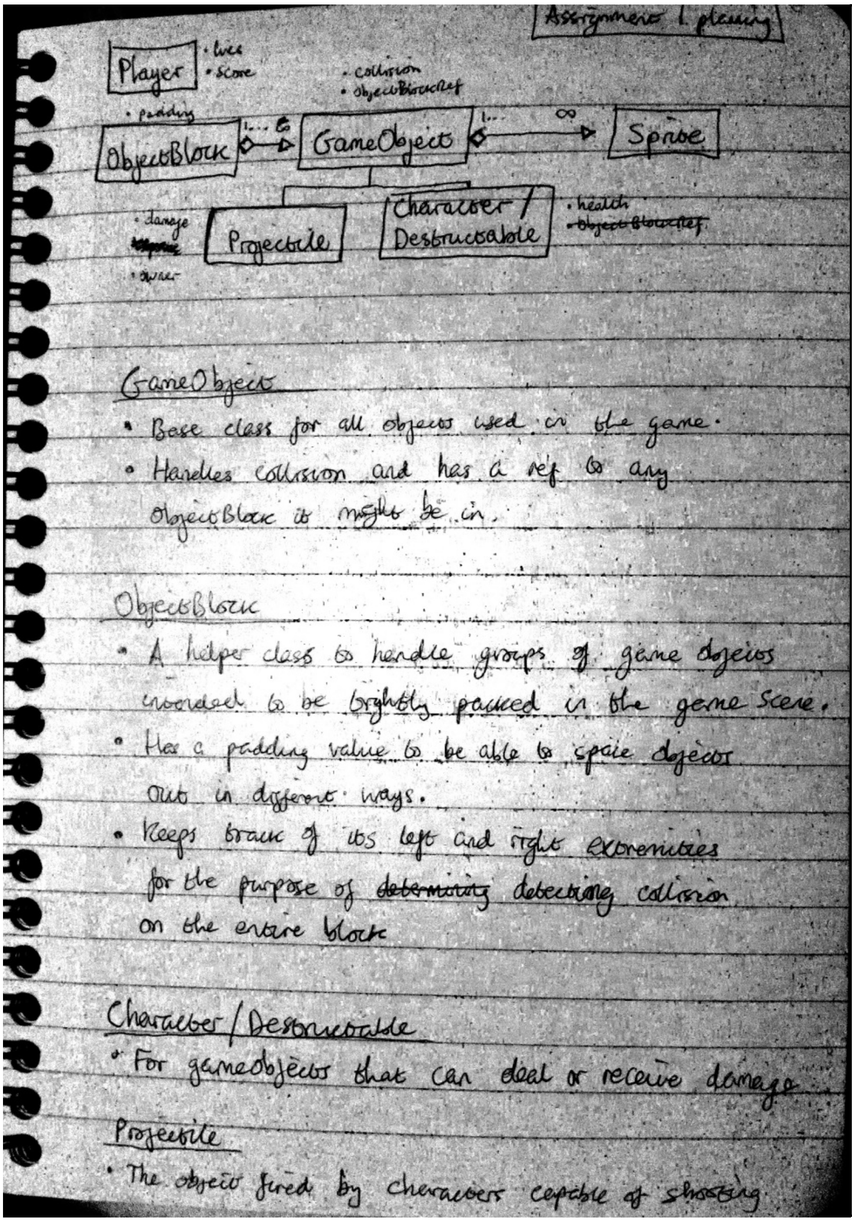
- The player gains a life after each "wave"
- 5 rows of 11 aliens
- Aliens move from edge to edge in a formation, descending towards the player at each extremety
- The "Block" of aliens shrinks as the player destroys them. ~~This wa~~ The current size of the block is taken into account during enemy movement.
- The movement tick rate of the aliens depends on how many aliens remain in the block and how far down the screen they are.
- The player sits at the bottom of the screen and can move left or right to the edges.
- The player can only have 1 active projectile, although the player can hold down the fire button to shoot as soon as possible.
- 4 defensive barriers offer the player some protection from the aliens projectiles.
- ~~A~~ Alive aliens in the block shoot at random intervals
- When the player kills all aliens, ~~the game~~ a new block is spawned slightly further down than the ~~last~~ group before started.
- When the player dies, the enemy stops moving until the player respawns (at the far left ~~on a random location~~) (the players active bullet also gets removed.)

One of the first things I did before starting the implementation of the assignment was conduct research into some of the existing Space Invader games available online.

The point of the research was to identify some of the details and nuance of the gameplay mechanics. For example, I looked at the movement behaviour of the Aliens and how often they shot at the player, as well as how the game progressed past the first wave.

The notebook scan on the left shows a list of details that I compiled from playing a few of these games online.

This helpful for understanding the type of product the assignment was asking for.



Before coding I outlined a few classes that I thought I would need to make up the elements I identified from my research.

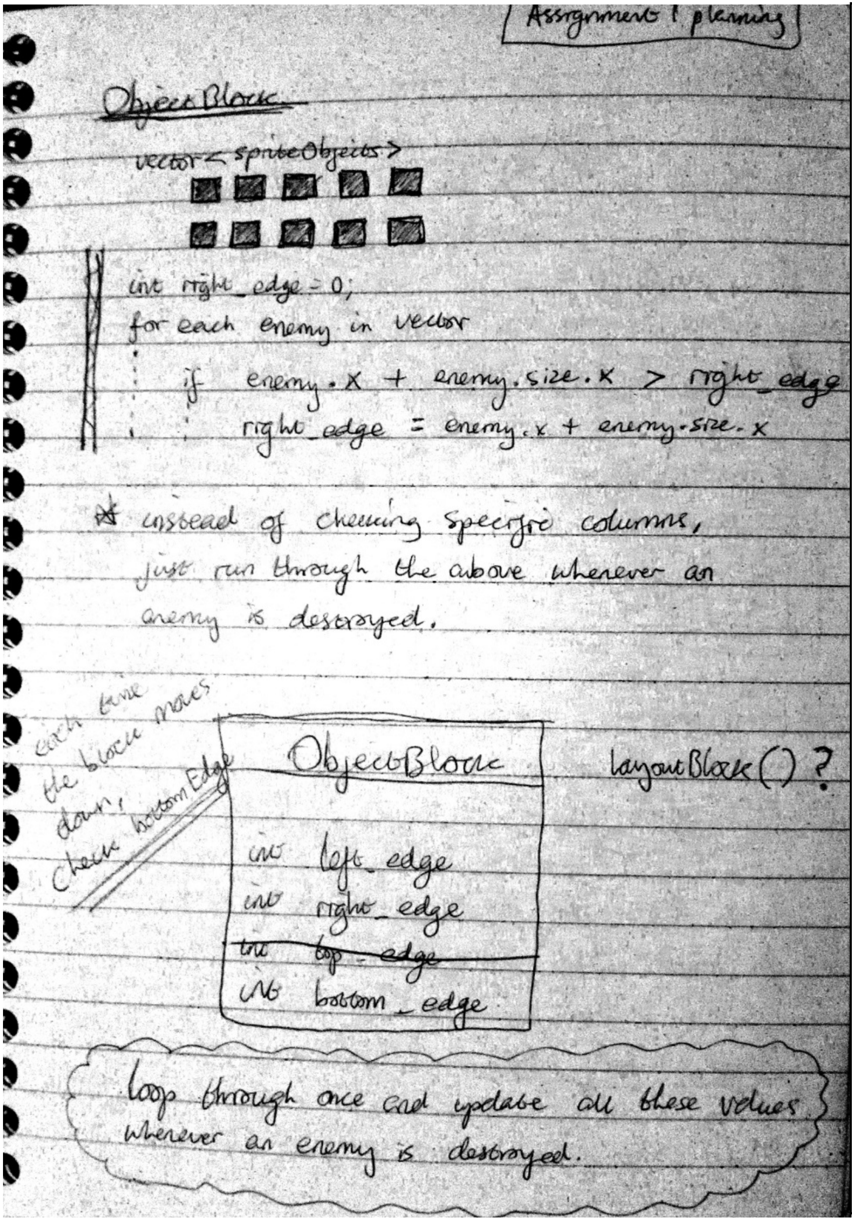
I had planned for a base class to be the root of all objects I was going to use in the game.

The intention was to minimise code duplication and make the code more modular, as I could expand on the base class for more specific cases, and easily add managers that utilised polymorphism for things like collision.

In the end a lot of this was changed to better suit the engine that was provided.

I found that the abstraction of a "Character" or "Player", and even a "Projectile" wasn't necessary.

All I needed were SpriteObjects and TextObjects, the game logic would handle the rest.



From my research, I identified an interesting trend with the typical Space Invaders game.

The trend was that a lot of the game's behaviour revolved around groups of objects.

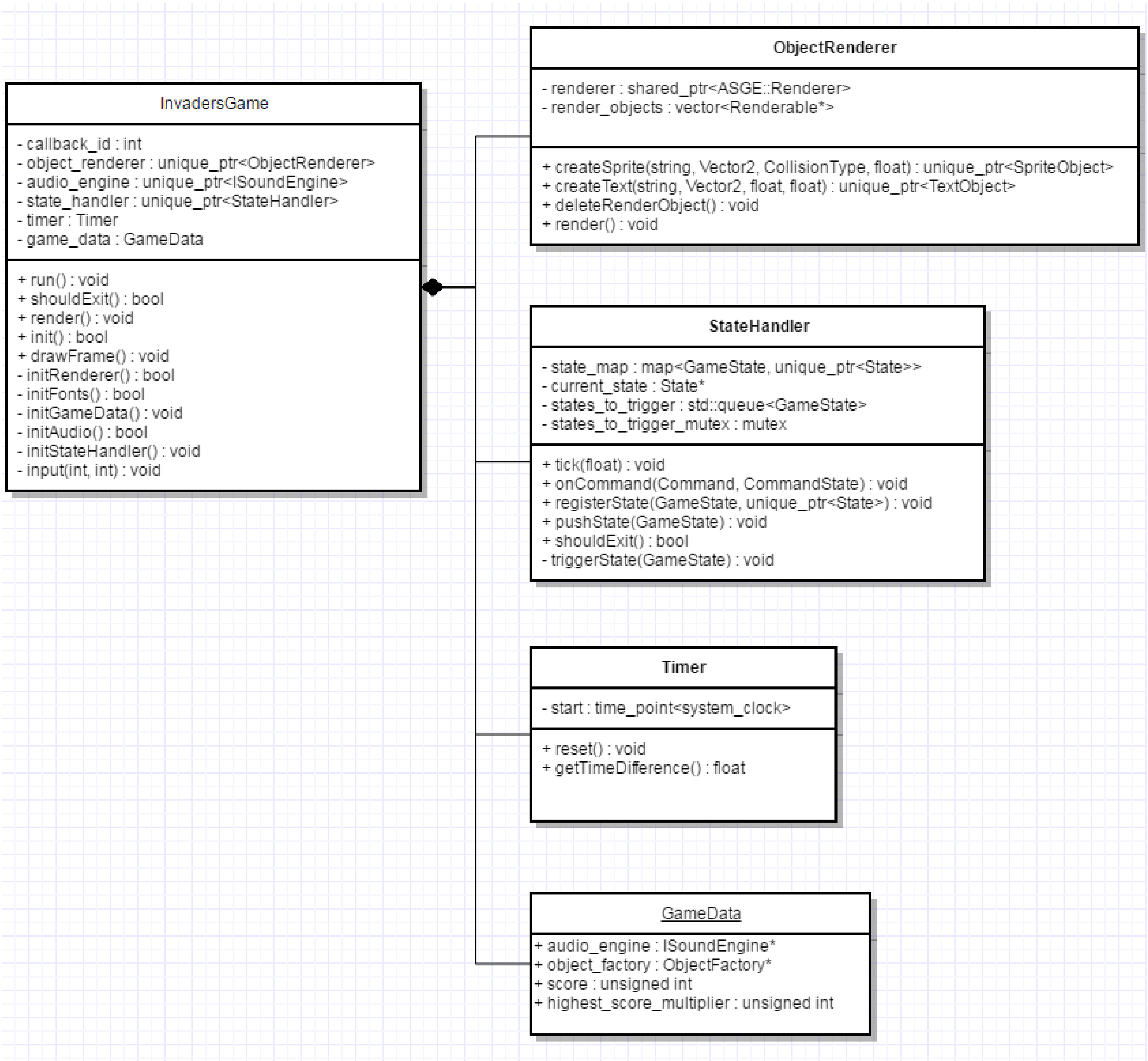
For example, the aliens, the stack of player lives, and also the barriers I considered all to have similar properties, in that they were a collection of objects with values to lay them out in a specific way.

So with this in mind, I decided on incorporating an ObjectBlock class into my game for the purpose of grouping SpriteObjects together, with the intention of utilising it to display the aliens, barriers and player lives.

This meant the ObjectBlock would be generic, and thus reusable for many aspects of my game.

CLASS HIERARCHIES

INVADERSGAME



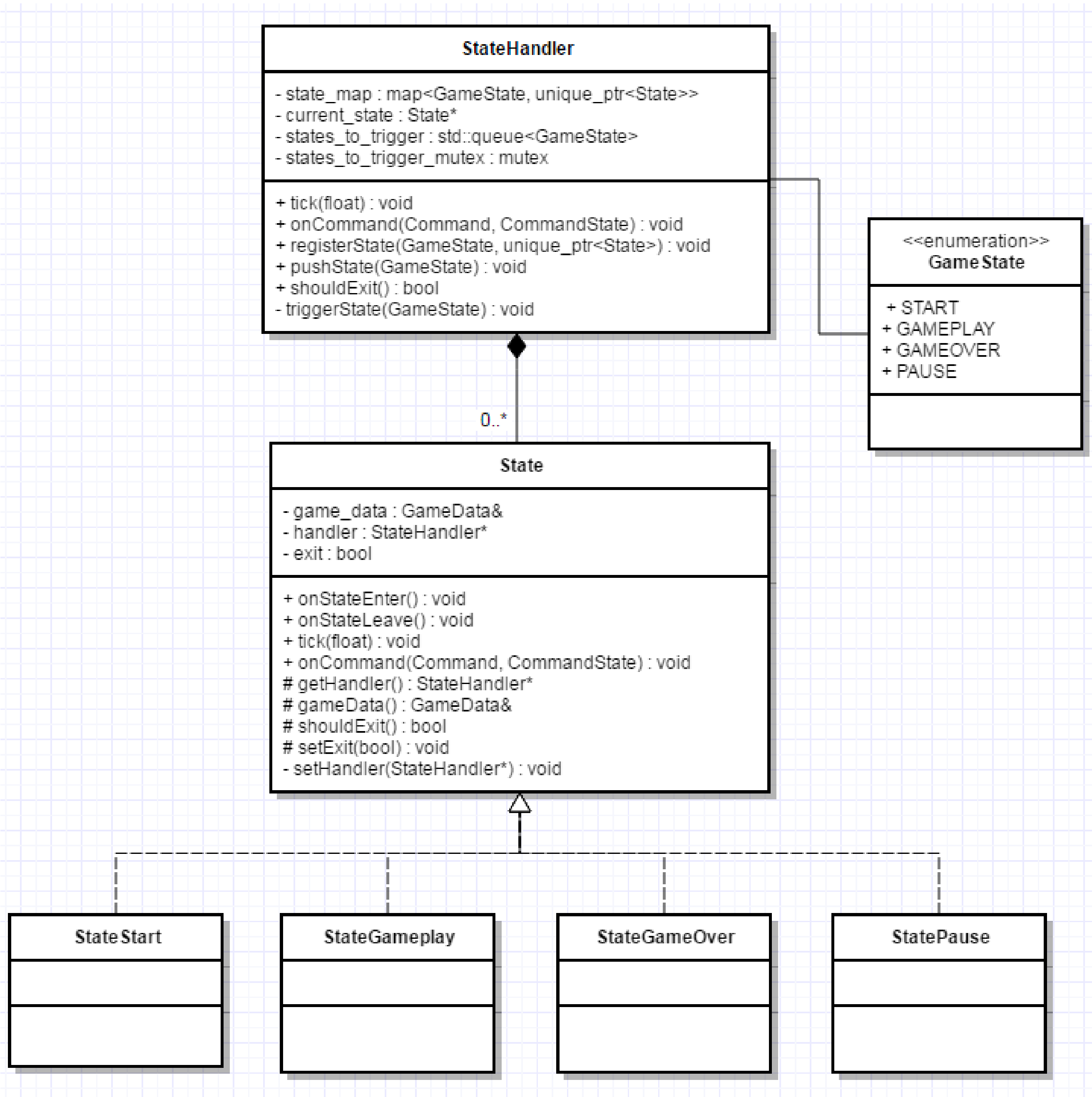
InvadersGame is fairly “ignorant” in the sense that it has no real idea what goes on once the game is started. All it does is initialise game-critical data and trigger the initial GameState through the use of the StateHandler.

Thereafter, all of the game’s behaviour is handled within the States, which are managed by the StateHandler, and the rending of all game objects are handled by the ObjectRenderer.

A simple Timer class is used to compare the time difference in seconds between the current cycle and the last cycle. The process of this gives us a basic version of Delta Time for use with frame-dependent behaviour.

A GameData struct is used to pass common important data between States.

STATE HANDLER

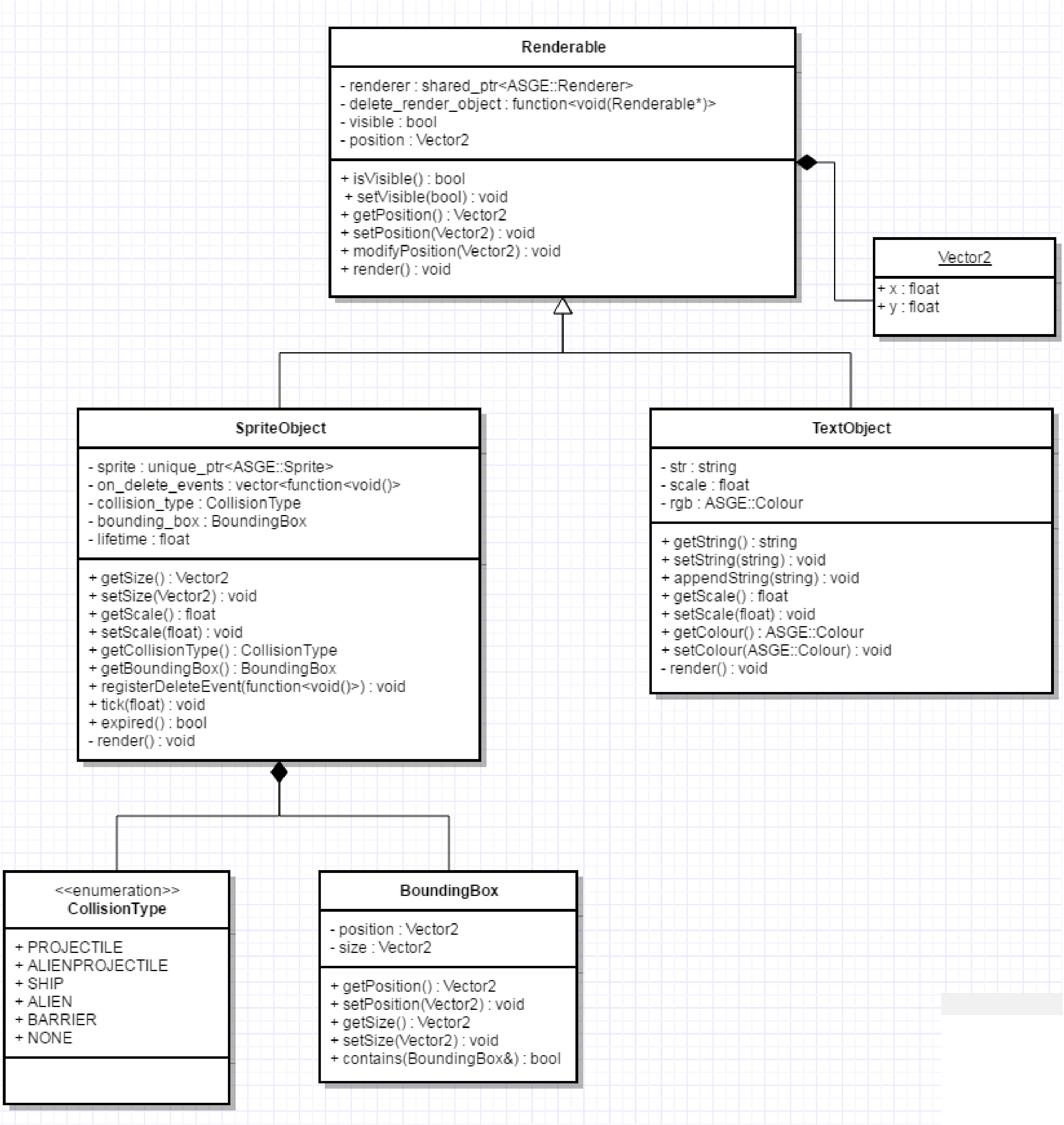


This all-important class handles all of the State transitions in the game. By using a separate class for handling states, InvadersGame is decoupled from the States themselves, and the States are decoupled from each other. It also means that it's always abundantly clear what the functionality of the State is when you're in it, because everything there is to do with that State only. No code clutter.

For the most part, the StateHandler simply sits between InvadersGame and the States and ensures that the correct game functionality is being executed. Information such as input commands are passed from InvadersGame to the StateHandler, which are then forwarded to the current State.

All of the States have a pointer to the StateHandler they belong to so that they can tell the StateHandler when a State transition should occur.

RENDERABLE

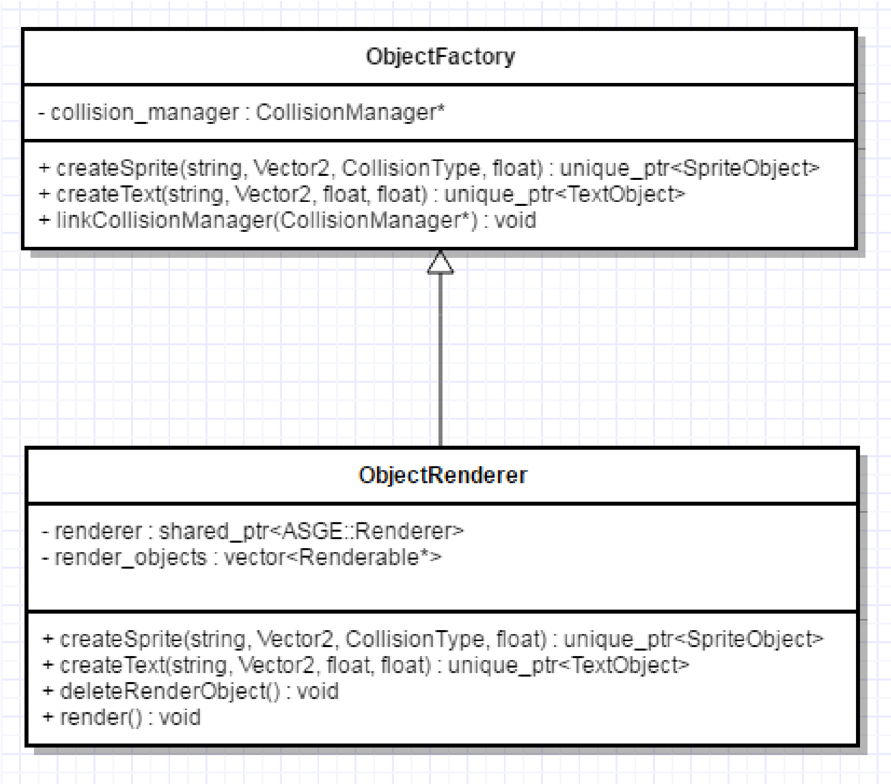


This class structure was mostly for convenience. I found it difficult to handle text and sprites in such drastically different manners, when I could see a lot of common functionality between them.

The idea behind Renderables was to hide some of the differences and treat text and sprites as equally as possible. With this implementation the gameplay logic will likely be concerned with either a SpriteObject or TextObject, while managerial classes will utilise polymorphism and treat either as a Renderable, for example when it comes to rendering either object.

SpriteObjects have BoundingBoxes for the purposes of determining collision, although the CollisionManager only bothers with SpriteObjects that have a CollisionType that isn't NONE.

OBJECT FACTORY

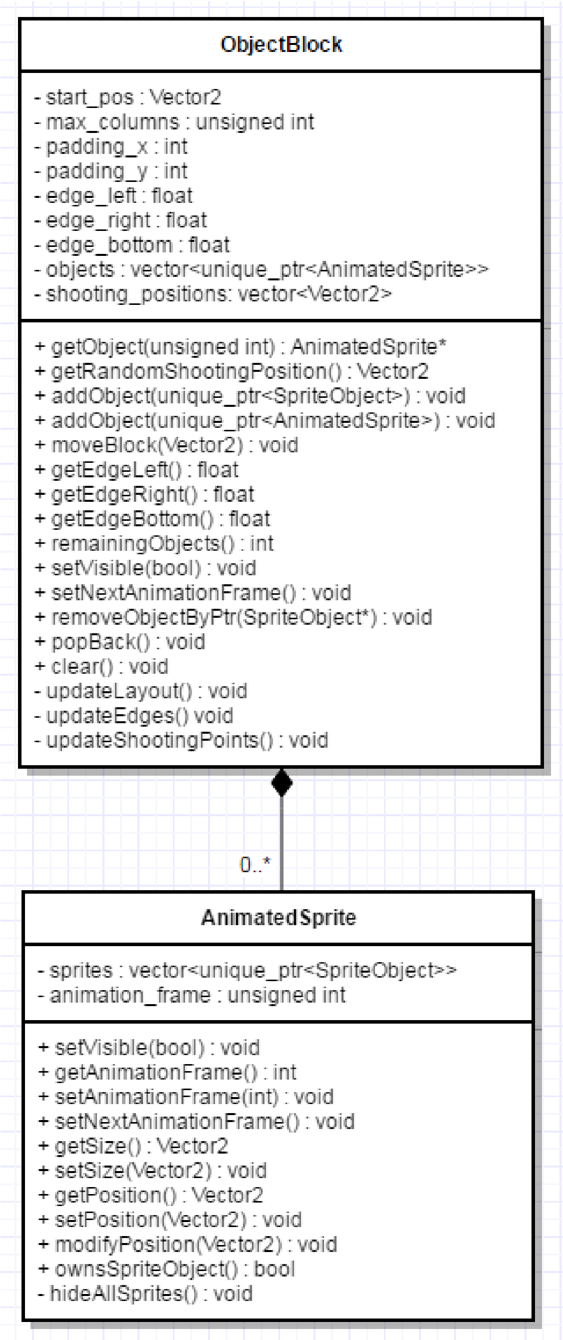


The ObjectFactory is what is used to create all objects used in the game. In reality, InvadersGame owns an ObjectRenderer which has extended functionality to keep a list of all the objects so it can ensure they are rendered.

The reason for the inheritance is so that the ObjectRenderer can be passed around without exposing render specific functionality, which the rest of the game won't be concerned with. The main draw of the ObjectRenderer is its ability to create SpriteObjects and TextObjects.

Another reason why the ObjectFactory is important is because it centralises the creation of all game objects. This means that we can handle special circumstances such as when a SpriteObject is created that needs collision.

OBJECT BLOCK

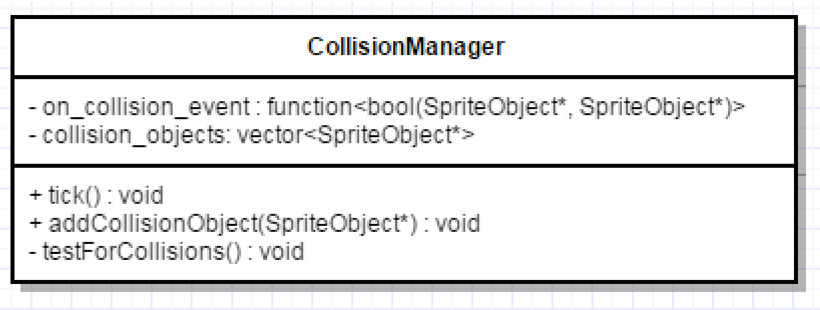


This is the final implementation of the ObjectBlock, which was conceptualised early on in the planning stage. Its primary function is to contain a list of special objects called AnimatedSprites, which are also essentially collection of SpriteObjects.

This lets us handle these AnimatedSprites as a big group, and lay them out based on variables such as columns, rows and padding. It's generic enough to make it more reusable and modular.

Ideally, ObjectBlock would be templated so that it could store either SpriteObjects or AnimatedSprites, or even both, as the conversion to AnimatedSprites results in a lot of unnecessary vectors.

COLLISION MANAGER



The last thing worth mentioning here is the CollisionManager, which StateGameplay owns an instance of.

The CollisionManager's primary concern is maintaining a list of all objects that have been passed by the ObjectFactory. Since only objects with an important CollisionType are passed, the CollisionManager can safely iterate through its list of objects and refer to their BoundingBoxes to determine a collision.

What's clever here is that when a collision occurs, the CollisionManager simply calls the function it was told to, which then informs the CollisionManager if it should update its list or not.

This means that as long as the function passed to the CollisionManager follows this format, it can be plugged into any State that requires collision.

Having the CollisionManager like this lets us easily expand our game if needed. If a new game object is added that also requires collision, it simply needs to be passed to the CollisionManager, instead of manually handling the collision elsewhere in code and potentially duplicating functionality.

POST MORTEM

Overall, this was a fairly difficult assignment. I realise there are some weaknesses in my final implementation, for example with the overuse of vectors, but there are some strengths too with the reusability of the ObjectBlock class for multiple game elements, and the CollisionManager for centralising collision events. Ultimately I feel that a more generic and abstract approach to code leads to a longer lasting codebase, and can actually be easier to understand in the long run.

The biggest problem I had early on with the engine was trying to treat both text and sprites the same. As discussed earlier in my documentation, I could see a lot of common functionality between the two. They were both assets that could be scaled, positioned and hidden as desired.

Introducing the Renderable class and ObjectRenderer removed a lot of these problems. It meant that sprites and text could find common ground as a Renderable, and the ObjectRenderer would ensure that both objects were rendered in the way that they needed to be.

I like to test my creations during development, and I tested this project heavily. From my research of existing Space Invaders clones, I could see a lot of nuance in the design of alien movement, player manoeuvrability and the ramp in difficulty as the game progressed.

Getting the gameplay to its current state took a lot of trial and error; I would often find myself making tiny adjustments to float values just to see what made the game more "fun". I can see the never-ending loop that game balancing poses, but it's definitely fun to tinker with a game's rules once the gameplay is there just to see how the balance of the game shifts.

Understanding the way threads interacted with each other was also difficult at first. It was unclear what control a thread separate to the main thread had over the behaviour of the program, and debugging runtime errors didn't always help pinpoint the issue. In the end I tried to do as little work in the input thread as possible, and only do basic things like change the state of standard variables (bool, int, etc.). I found this to be more reliable within the ASGE engine.