**Documentation**

**Key Responsibilities**

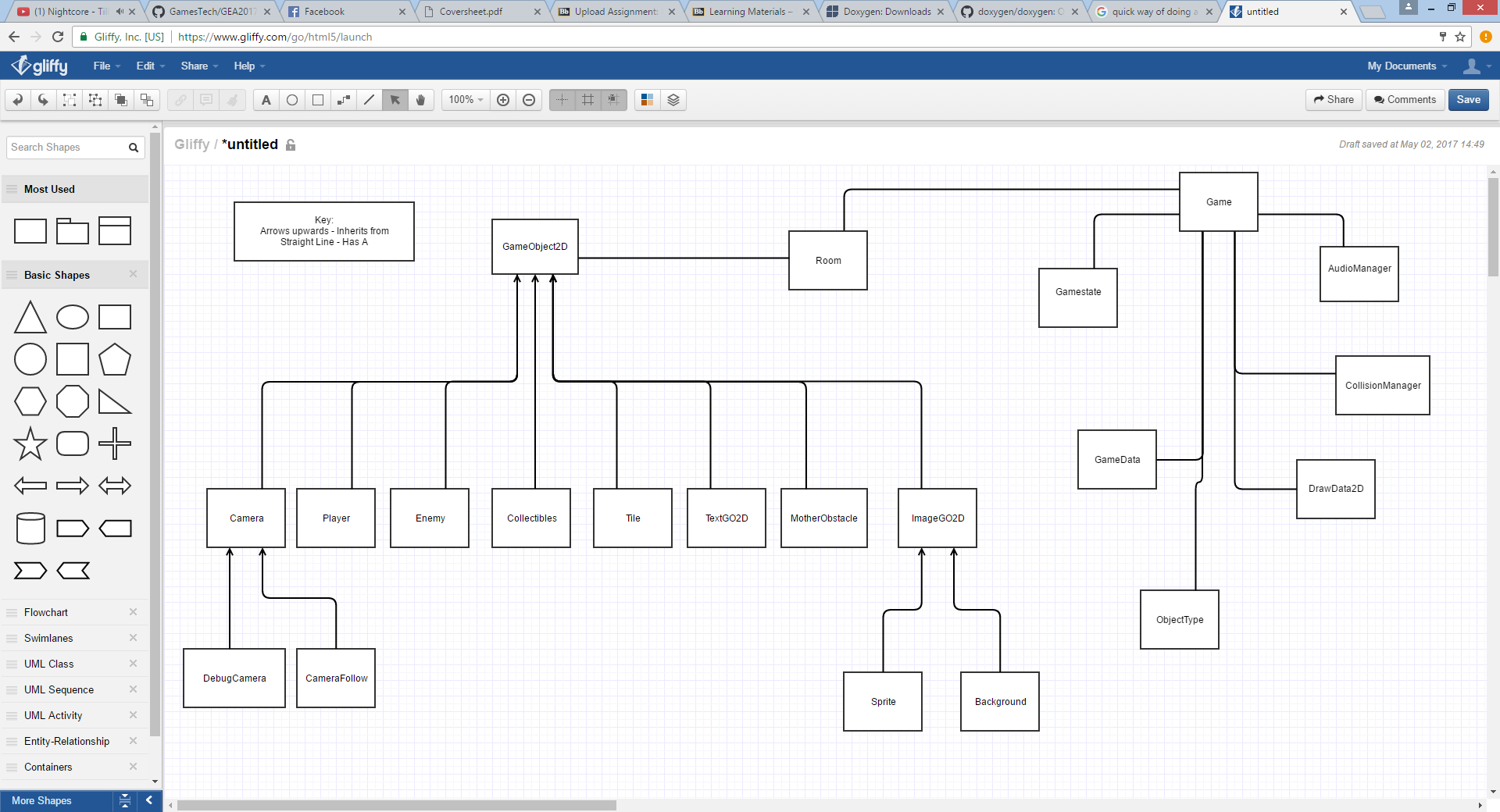
Ben Meredith – Collision; Finite State Machine; Lives

Matthew Holmes – Level Editor and file reading; Enemies; contributing to different platform types; Collectibles

Tim Penfold – HUD; Player Physics; Contributing to different platform types

Matthew Cheung – Collision; Contributing to different platform types; Optimisation

**Class Structure**



Most of the in-game objects inherit from GameObject2D, which contains most of the variables used for physics behaviour, allowing for physics to be applied to any object it is deemed necessary for. Game objects are created within the room, which is created by the game loading in the necessary text files and iterating to create each tile as dictated by the contents of the text file. The game also manages the game’s states and various types of data (draw data and game data) as well as the other managers for audio and collision between the different objects.

The camera class is a base for the CameraFollow class which is used during normal gameplay but also for the DebugCamera class, which is created to allow users to press space and pause gameplay, then allow for the user to click where they’d like to create additional platforms and thusly edit the level during gameplay.

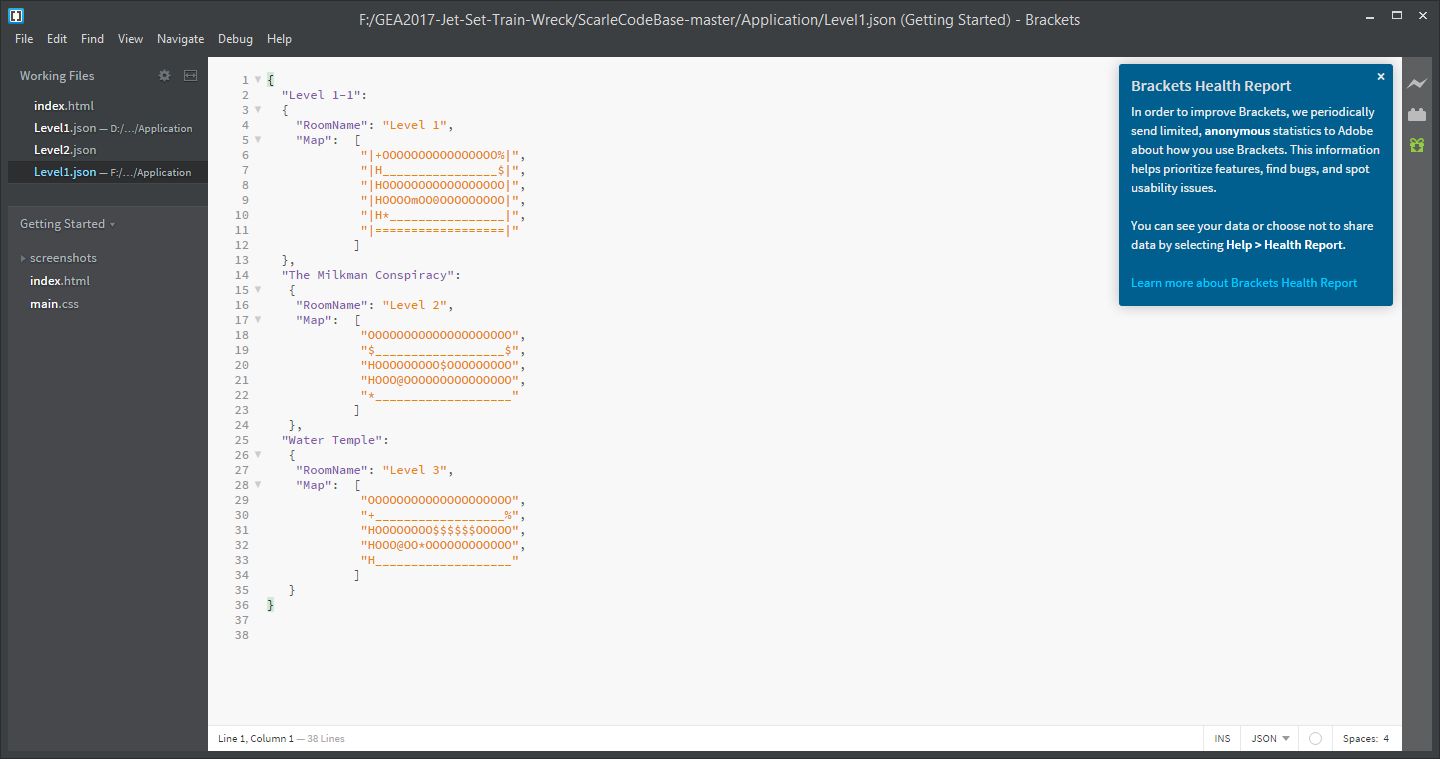
The ImageGO2D class is a fairly self-explanatory base for all the images throughout the game, consisting of the sprites and background.

Another class of note is the Tile class, which is where ObjectType is mainly used. The ObjectType data is used when creating the rooms to apply different behaviours to different tiles while keeping those tiles in one class. For example, a tile with ObjectType:: Ladder will work with the ladder collision logic during gameplay.

**File Structure**

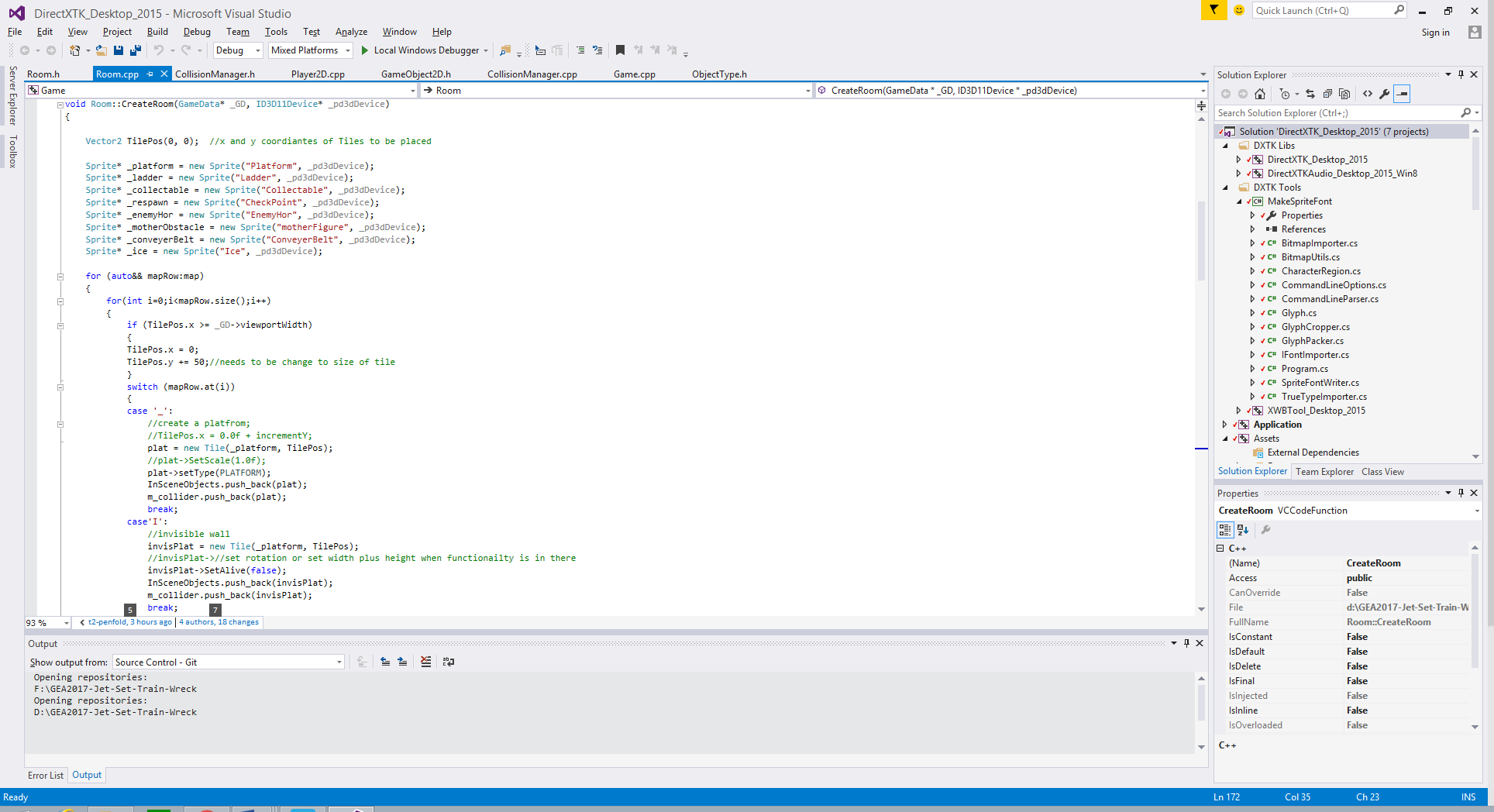
The main game data used is that of sprites used for the various graphics within the game and the text files used to map out the game’s levels.

The text files contain a Room name for each room and then a map of each room as a set of characters, as shown in the image below.



When a room is created, the file is loaded and each character is read. Depending on the value of each character, a different object is loaded. The image below shows how a value is checked and if, for example, the character is an underscore (‘\_’) then a platform tile is created in the room. It then increments the TilePos to ensure that as tiles are loaded consecutively, they are placed next to the tile before them.

The room name is also loaded in to be displayed on the in-game HUD.



This facilitates a tile-based system within our game and allows the game to create more than one of a given object without having the hard-code in each instantiation, instead basing level structure on the text file loaded in. This also allows easy changing of levels as one simply needs to change the text file, as opposed to recoding each aspect of the level’s creation, such as the order in which objects are made.

One issue that the game was designed to avoid is loading in each sprite that was needed more than once when creating a room, as this is highly unnecessary and involves using a lot of memory to store an instance of the sprite each time an object is created. To this end, when a room is created, it uses a series of pointers to store each image file needed for the different object types (as shown above). Then, as the game loads in different objects it will refer to a given pointer for the image, rather than directly loading a new instance of the image. This saves a lot of memory and makes performance much smoother as the memory not being taken up can be used for other functionality.