**Milestone 5**

**Functionality**

The structure of the code has been rebuilt to allow for an ID system. Now every object, sprite, clipping detector and collision detector contains an ID that is set when the object is created. This makes it easier to adjust a singular object by just using its respective ID.

The Simulation class has now been properly set up. It is used to create and control objects. Furthermore, when it is created it automatically creates a visualisation object and gets a pointer to it to allow for easier control over the sprites. You can call visualisation functions through the simulation object. Note that it is possible to create an object without a sprite, and vice versa. Furthermore, it also contains the functions update(), which is used to call any functions that have to be called every frame, and GSupdate(), which is used to call any functions that have to be called every game tick (such as visualisation’s new nextFrame() function).

The Object class has now been set up. It stores the position of an object, whether it is jumping or not and its current vertical and horizontal movement. It also contains a lot of get functions so that this information can be used within the simulation class. Furthermore, a collision detection structure has been set up (collDetector). This stores the position and size of the collision box so that it can be used to detect collisions.

The movement system has been moved to the simulation class and completely rebuilt. Instead of controlling like a top down game, it controls like a 2D platformer. Using the momentum stored in the object and the collision boxes stored in simulation’s solidVector, it is able to detect when a collision is about to happen and snaps the object to whatever it was going to collide with. This prevents the object from going inside the other object, which would prevent the object from being able to move. Using the calculated momentum, it then applies it the object’s position, and then uses the object’s position to apply it to the appropriate sprite.

The visualisation and sprite classes have new code added that allow for animated sprites to work. Now, when adding a sprite, the code will ask for 3 new parameters: frames, frameWidth and frameHeight. These denote the number of frames and the size of that frame. If a 0 or 1 is inputted into frames, then the visualisation class will take note and declare that the sprite is not animated. It will then call the isAnimated() function from the sprite, which then stores the provided information so that it can adjust what it displays accordingly. The visualisation class then uses nextFrame() to tell all animated sprites to go to the next frame of animation.

Finally, you can now change the sprite of an already existing sprite on the go. This takes in the majority of the parameters that the addSprite() function requires (apart from the position), and requires the sprite’s ID to be able to find and adjust that sprite.

**Code Changes**

*Main*

**MOVED** – *void clearScreenBuffer()* – this has been moved from the main code to the visualisation class where is makes more sense to be.

**MOVED** – *void checkAnalogue()* – Moved to the simulation class where it makes more sense to be.

**MOVED** – *void moveInputKeyboard()* - Moved to the simulation class where it makes more sense to be.

**MOVED** – *void moveInputController()* - Moved to the simulation class where it makes more sense to be.

**ADDED** – *Game Speed* – An if has been implemented into the main loop to allow certain functions to be called within the game speed instead of the frame speed. This means that the game will run at a consistent speed as frames can no longer affect how fast it runs.

*Class visualisation*

**REMOVED** - *Std::vector<int> controlSpriteIndex*– Due to the new ID structure and movement system, this vector is no longer necessary for storing which sprites can move.

**ADDED** - *Std::vector<int> animateSpriteIndex* – This is needed for the new nextFrame function, so that it can just go through every animated sprite without the need for an ID.

**CHANGED** – *bool addSprite()* – The parameter bool playable has been removed due to the removal of the controlSpriteIndex vector, new parameters int frames, int frameWidth and int frameHeight have been added to allow for animated sprites, as well as std::string ID to give new sprites an identifier.

**ADDED** – *bool changeSprite()* – This allows for the changing of sprites while the code is running and after the sprite has already been added. The new sprite will still contain the same ID and position. Parameters: std::string ID, std::string filePath, int width, int height, int frames, int frameWidth, int frameHeight.

**REMOVED ­**– *void controlSprite()* – This has been removed due to the new ID structure, as it is no longer necessary.

**ADDED** – *void moveSprite()* – This moves the sprites uses the given position and identifier. Parameters: std::string spriteID, int xPosition, int yPosition

**ADDED** – *void nextFrame()* – This uses the animateSpriteIndex vector to loop through all the animated sprites in the spriteVector and tells them to go to the next frame of animation.

**MOVED** – *void clearScreenBuffer()* – this has been moved from the main code to the visualisation class where is makes more sense to be.

*Class sprite*

**ADDED** – *std::string spriteID* – This stores the sprite’s ID for the new ID structure.

**ADDED** – *bool animated* – Stores whether the sprite is animated or not.

**ADDED** – *int spriteFrames* – Stores the total number of frames the animated sprite has.

**ADDED** – *int frameWidth* – Stores the width of an individual frame.

**ADDED** – *int frameHeight* – Stores the height of an individual frame.

**ADDED** – *int currentFrame* – Stores the current frame the sprite is on.

**CHANGED** – *sprite()* – new parameter: std::string spriteID. This is used to store the sprite’s ID

**ADDED** – *std::string returnID()* – used to return the ID of the sprite so that it can be identified.

**ADDED** – *void changeFile()* – This is used to change the filepath for the sprite without having to create a new sprite. Parameters: std::string filePath, int width, int height.

**ADDED** – *void isAnimated()* – Tells the sprite that it should be animated, and uses the parameters to store what information is needed for the animation, which will be used in the renderTexTransparent function. Parameters: int frames, int frameWidth, int frameHeight.

**ADDED** – *void nextFrame()* – Will add 1 to currentFrame, essentially moving to the next frame. If currentFrame exceeds spriteFrames, it will go back down to 0.

**CHANGED** – *void renderTexTransparent()* – This will now use the animation information stored to render the frame of a sprite if it is animated.

**CHANGED** – *void moveSprite()* – Now only takes in the position it will be moved to, as the movement calculation has been moved to the new Simulation class. New parameters: int xPos, int yPos.

*Struct collDetector* (**NEW**)

* *Std::string objectID* – Stores the ID of the object that it is connected to.
* *Int xPos* – Stores the x position of the collision box.
* *Int yPos* – Stores the y position of the collision box.
* *Int width* – Stores the width of the collision box.
* *Int height* – Stores the height of the collision box.

*Class simulation* (**NEW**)

**Private**

* *Int scrWidth* – Stores width of the screen.
* *Int scrHeight* – Stores the height of the screen.
* *Int scrBufferSize* – Stores the size of the buffer for the screen.
* *Int leftXAnalogue* – Stores the x analogue for controller input.
* *Int leftYAnalogue* – Stores the y analogue for controller input.
* *Std::vector<object\*> objectVector* – Stores pointers to object class objects that are created through the simulation class.
* *Std::vector<collDetector\*> solidVector* – Stores pointers to collDetector objects that are created through the simulation class.
* *Visualisation\* visObject* – This is a pointer to the visualisation class object that is make when simulation is created.

**Public**

* *Simulation()* – Constructor that creates a visualisation object and stores a pointer to it, as well as stores the necessary screen information. Parameters: int screenWidth, int screenHeight, int screenBufferSize
* *~simulation()* – Destructor that will delete any of the heap memory that is being used in the objectVector, solidVector and visObject.
* *Void update()* – Is used to update anything automatically for each frame. It currently calls visObject’s rumbleSprite(), clearScreen() and renderSprites() functions. Parameters: bool alpha.
* *Void GSupdate()* – Is used to update anything automatically for each game tick. It currently calls visObject’s nextFrame() function.
* *Void createObject()* – Creates a new object class object and stores it in the objectVector. If solid is true, it will also create a collDetector object to detect collisions. Parameters: std::string objectID, int xPos, int yPos, bool solid, int width, int height.
* *Void visAddSprite()* – Calls visObject’s addSprite() function. Parameters: std::string objectID, std::string filePath, int width, int height, float xPos, float yPos, int frames, int frameWidth, int frameHeight.
* *Void visChangeSprite()* – Calls visObject’s changeSprite() function. Parameters: std::string objectID, std::string filePath, int width, int height, int frames, int frameWidth, int frameHeight
* *Void moveObject()* – Used to move an object through the keyboard or controller. First it checks what is being input (Works the exact same as Main’s old moveInputKeyboard() and moveInputController() functions), and then calculates the object’s momentum vertically and horizontally. It does this by using the ID provided in the parameter to figure out which collision boxes it need to compare to, and adjusts the momentum based on whether it is close to colliding, is colliding or is not colliding. It then applies the momentum to the object’s position, and then uses the object’s position to apply that movement to the collision box and the sprite. Parameters: std::string objectID, bool controller, int speed, int maxSpeed, int gravity, int maxFallSpeed.
* *Int objXMomentum()* – Returns the x momentum of the object, which can be used to make certain things happen depending on the object’s momentum. Parameters: std::string objectID.
* *Int objYMomentum()* – Returns the y momentum of the object, which can be used to make certain things happen depending on the object’s momentum. Parameters: std::string objectID.

**MOVED** – *void checkAnalogue()* – Moved from the main code to the simulation class where it makes more sense to be.

*Class object* (**NEW**)

**Private**

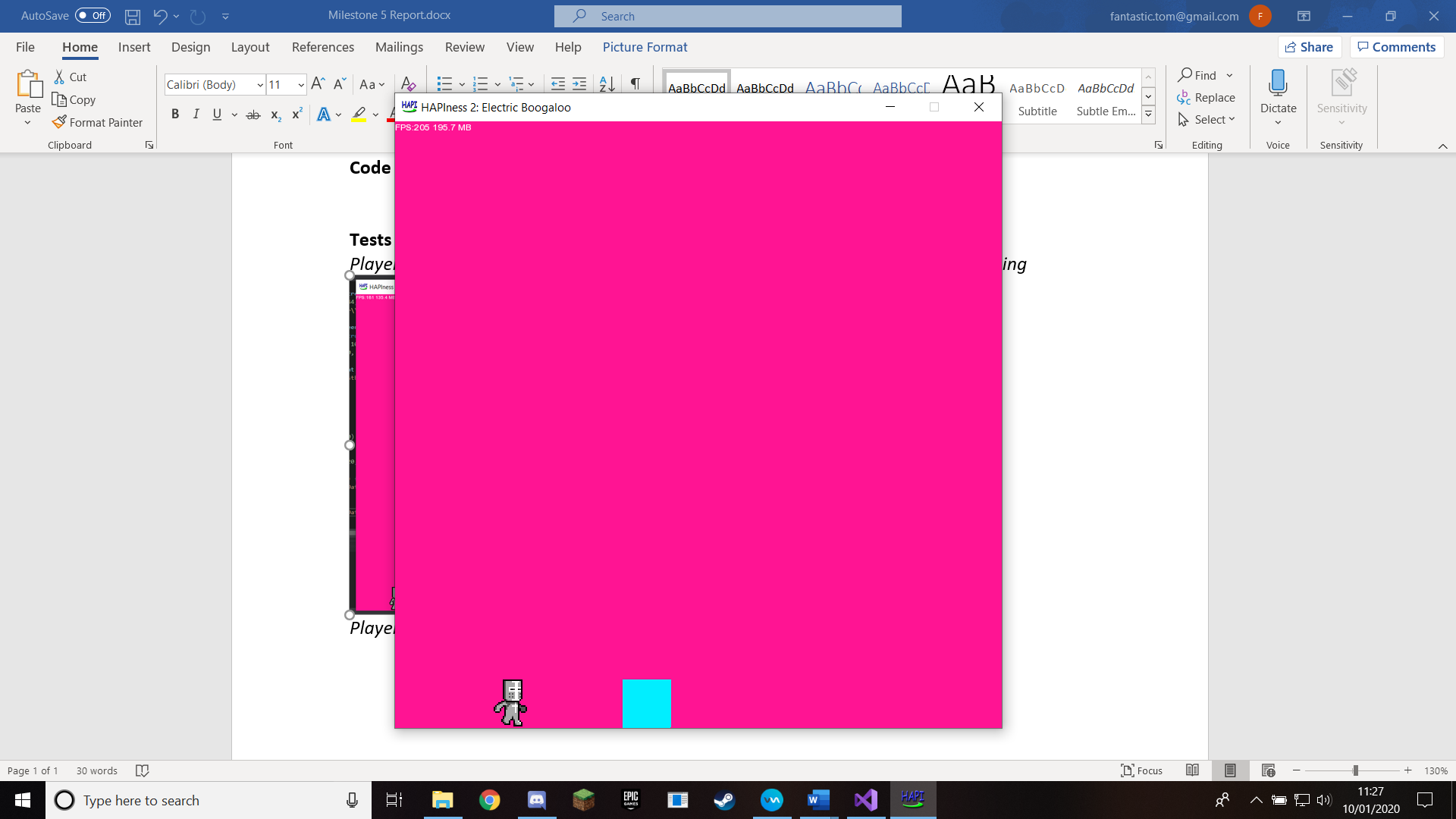
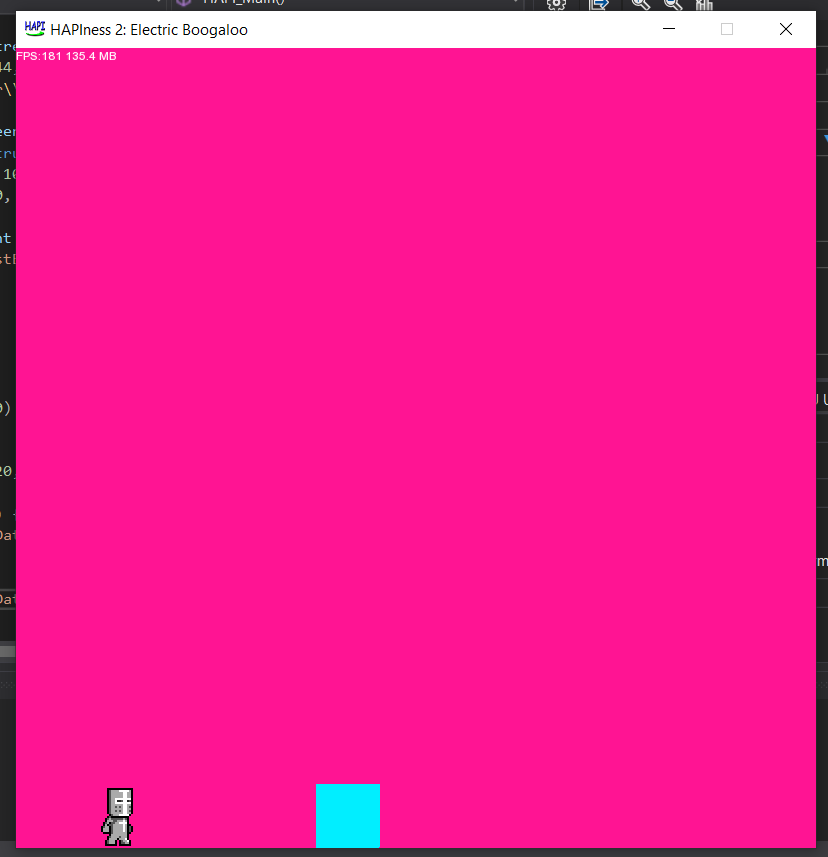
* *Std::string ID* – Used to store the identifier of the object so that it can be identified.
* *Int posX* – Stores the x position of the object.
* *Int posY* – Stores the y position of the object.
* *Bool jumping* – If the object is jumping, it is true. If not, it is false. This is to prevent infinite jumping without touching the ground.
* *Int xMomentum* – Stores the x momentum of the object. This is so it can provide more dynamic movement.
* *Int yMomentum* – Stores the y momentum of the object. This is so it can provide more dynamic movement.

**Public**

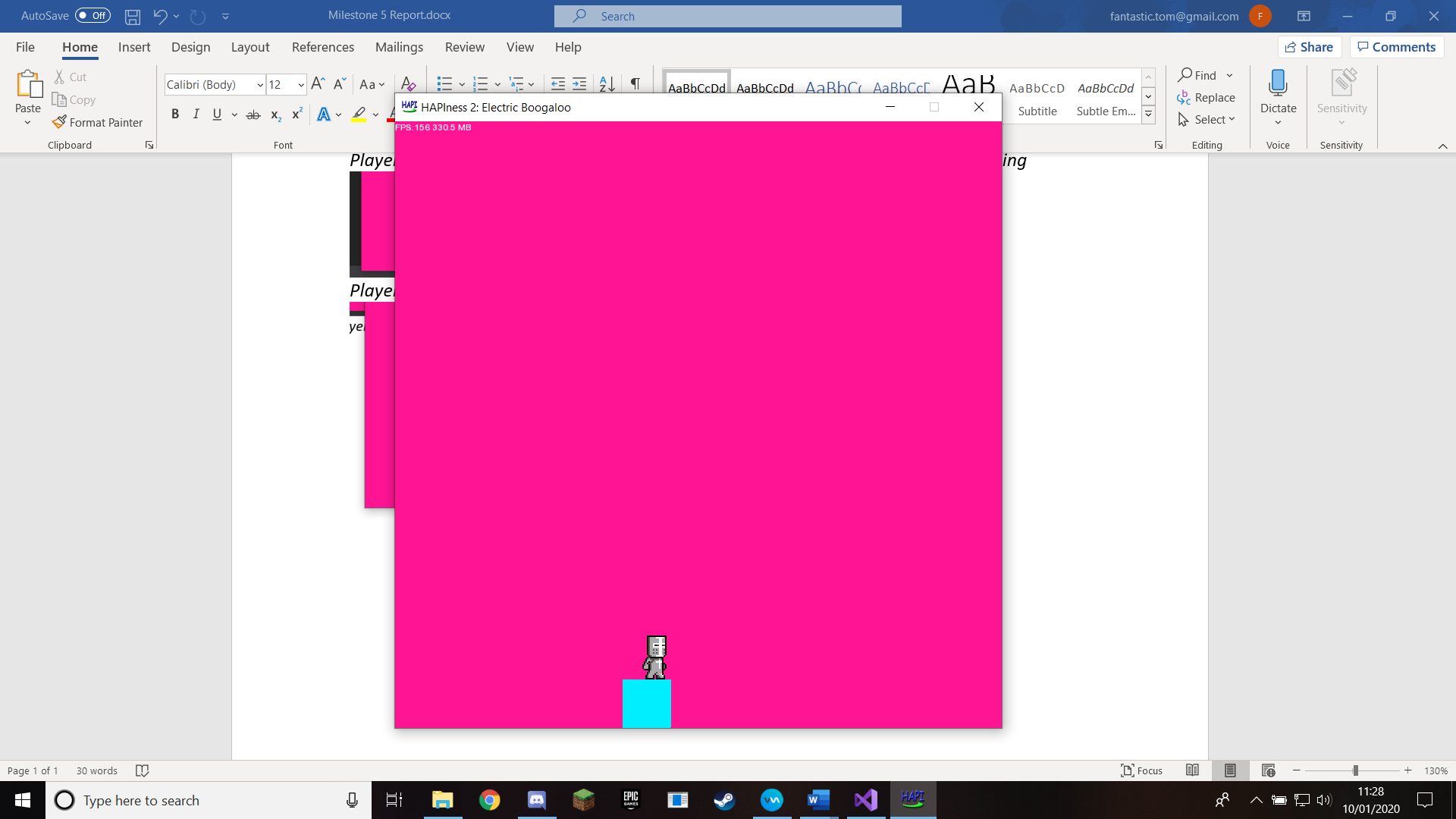
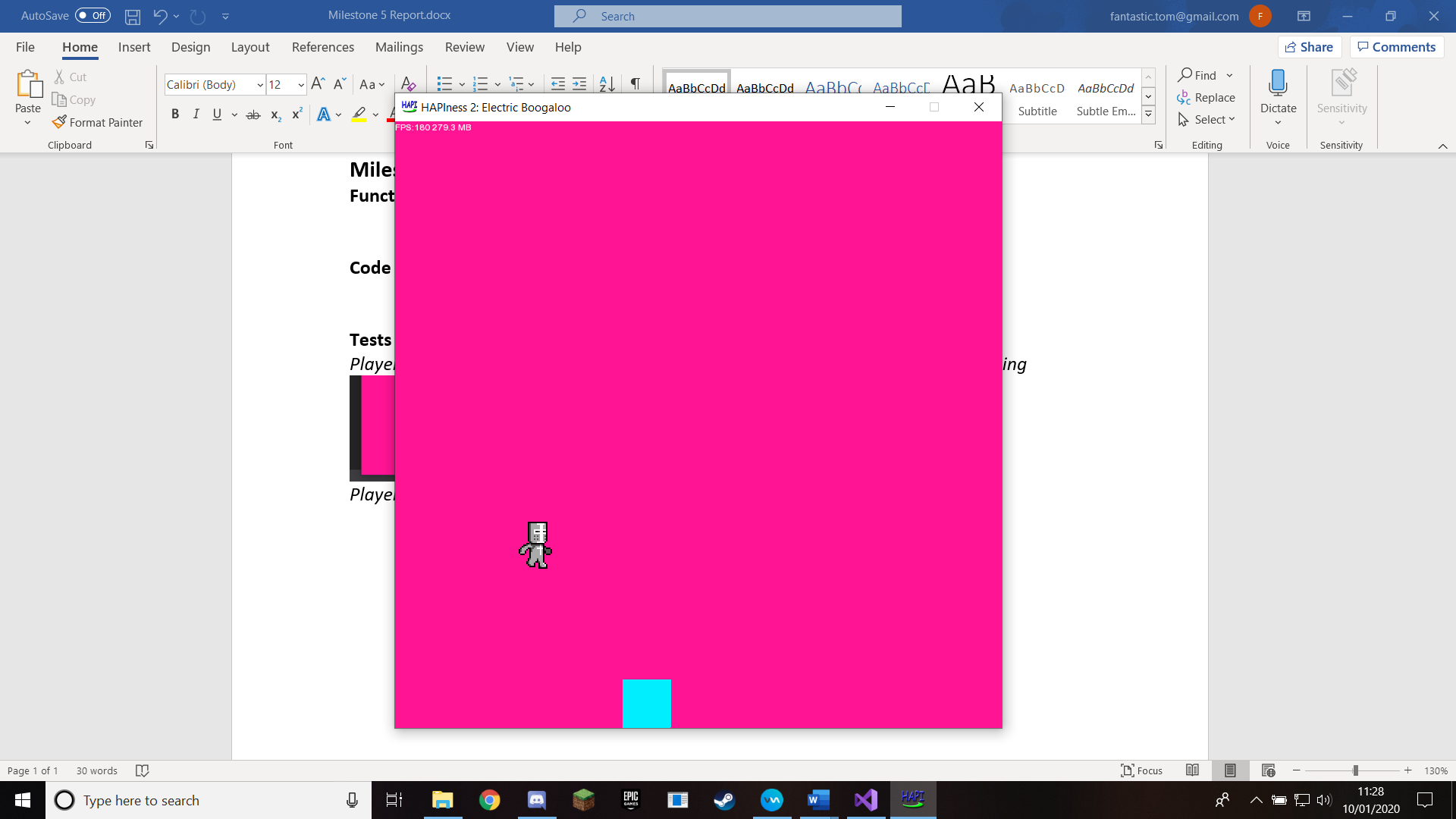
* *Object()* – Constructor. Parameters: std::string objectID, int xPosition, int yPosition
* *~object()* – Destructor.
* *Std::string returnID()* – Returns the ID of the object so that it can be identified.
* *Void setPosition()* – Allows the position of the object to be set. Parameters: int xPosition, int yPosition.
* *Int getPosX()* – Gets the x position of the object.
* *Int getPosY()* – Gets the y position of the object.
* *Bool& getJump()* – Returns a reference to the jumping variable, so that it can be changed. Used to prevent jumping in midair.
* *Int& getXMomentum()* – Returns a reference to the xMomentum variable, so that dynamic movement can be applied.
* *Int& getYMomentum()* – Returns a reference to the yMomentum variable, so that dynamic movement can be applied.

**Tests**

*Player character animates a walking animation when moving left, right, and jumping*



*Player is able to jump onto platforms with little to no bugs*



**Additional things I need to work on in the next milestone**

I need to implement a function that returns what the current filePath of a sprite is, as when I have the code change what file that sprite has it is constantly changing it, meaning it can’t animate as it is always being reset. Having a getFilePath() function will allow me to only change it if it isn’t already that file.