WIZ Game Engine

Ian Castillo

William Torres

Zachary Urrutia

**Table of Contents**

**Table of Contents 2**

**1.Introduction 3**

1.1Project goals 3

1.2 Box 2D 3

1.3 SDL 3

1.4 ImGUI 4

**2.Subsystems 4**

2.1 Movement 4

2.2 Map 5

2.3 Sensors 6

2.4 Game Objects 7

2.5 Texture Manager 9

2.6 Enemy AI 10

**3.Reports 10**

3.1Report 1 10

3.2 Report 2 10

3.3 Report[3](https://docs.google.com/document/d/1R_OabPvOxOHV4rNZ7PpwBhUeySWXI5h_/edit#heading=h.2jxsxqh) 11

3.4 Report 4 11

3.5 Report 5 11

3.6 Final Report 12

3.7 Creators and Maintainers 12

**4. Evaluation 16**

4.1 Possible Additional Implementations 16

4.2 Working Parts 16

4.3 Non-Working Parts 16

4.4 Improvements 16

# Introduction

## 1.1 Project goals

The goal of our game engine was to create a 2D fighting game inspired from the game Super Smash Bros. We hoped to implement many different functions to this engine in order to get it as close as possible to our inspiration in our own way. We wanted to add a sensor function for detecting collision with other objects, a map editor to make new scenes, create an enemy AI to compete with the player, and provide some basic animations for our game objects. We used three different libraries for our project developments Box2D, SDL, ImGui. We also used Visual studios for writing our code and ChatGPT to help come up with ideas on how to code some sections for this project.

## 1.2 Box 2D

We used the Box 2D library to create a physics body on each of our Game Objects. Each game object is defined as a box of 64 x 64. It’s also used for gravity and creating platforms that the player can stand on. Box2D have functions such as ApplyForce() which are used to create movement. Has many convenience functions such as b2vec3(), get position(), get angle(). This library's implementation was carried out for the most part by William.

## 1.3 SDL

Used in creating many of the objects seen on screen, including the screen itself. Other things that used this library were the health bars for the player and enemy, the map editor functionality, input functions for moving and attacking with the player, and converting images into textures so that they can be rendered. This part was a group effort between William and Ian creating functions for player inputs and the health system, Ian more responsible for debugging issues than function creation. Zachary is responsible for implementing the rendering function and map implementation.

**1.4 ImGui**

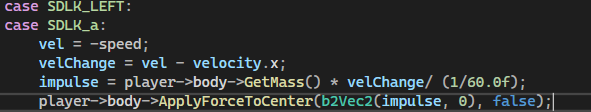
We used the ImGui library so that we could implement our map editor. We had originally planned on using it for a user interface, but we ended up using the SDL library for our UI. This library functionality was implemented by Zachary.

# Subsystems

## 2.1 Movement

Our movement is done by polling for an event. We used SDL2’s built-in functions for event polling. The function is “SDL\_Poll\_Event()” We then check if the event is a key press or not. If it is, we check which specific keyboard key was pressed. SDL has enums for the different keys which is how we check using a switch statement. If the right arrow or ‘d’ key was pressed, we then apply force to the player, so it moves right. If the left arrow or ‘a’ key was pressed, we apply force to the player, so it moves left. If the space or up arrow key was pressed, we then use a variable called jumpsteps which is just an integer.

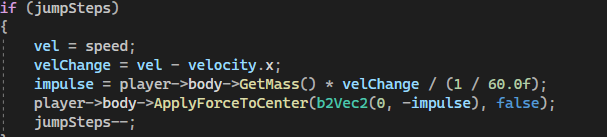
Jumpsteps allow us to configure how many frames we want force applied upwards to the player. Every frame we decrement the jumpsteps variable. This is also to allow the player to keep gaining height even if they are no longer grounded. This also allows the player to move while they are jumping. Since SDL2 can only poll one event at a time, it can only read one key press every frame. Therefore, even as the player is jumping, they don’t need to keep the key pressed and can move in a direction instead. As long Jumpsteps is greater than 0, we apply force upwards to the player.



Force is applied using the applyForceToCenter() function of Box2D. It only needs 2 parameters; The x and y of the force, and a bool to determine if the body should be woken up if it is sleeping. Our program never puts bodies to sleep so the bool is irrelevant. Our force is calculated by first calculating the velocity change.

The change is equal to our set speed - the speed the object is already going. With this, we don’t go over our set speed. Next, using our velocity change, we calculate the force to apply by multiplying it by the object’s mass and dividing by the frame rate, which in this case is 60 fps.

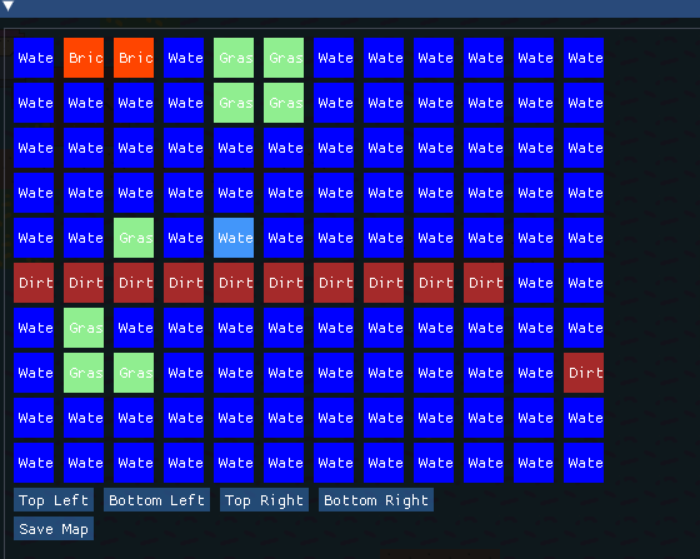
By calculating force this way, we are following a popular physics equation: F = ma or F = m (V1-V2)/t. Our function needs the x and y of a force. Therefore, when we apply force in a specific direction, the other direction will be 0. So, increasing force upwards will have the x of the force be 0. The inverse is true for moving to the sides. Important to note that since SDL2 has the origin at the top left instead of the bottom left, we need to add a negative sign to the y of the force. This same logic is used to move the enemy as well.



## 2.2 Map

Our map subsystem is set up as a 2D array that is 20x25 that takes in different values based on the type of tile being used in the position. 0 is used to represent our water tile, 1 is used to represent grass, and 2 to represent as dirt which is the tile that handles is recognized as the ground for the player and will not let the play fall straight down, 3, 5, 6 and 7 also follow this same logic that allow the player to stand on bricks, wood, ice, and rocks, and 4 and 8 are just background like lava and a marsh. The map grid displayed on the GUI is color coded to help the user better visualize each tile.

Using the ImGui library we have a function that allows the player to fill in the spots using the values to fill in textures in each area. The user can save their work onto a text file by pressing save at the bottom the GUI.

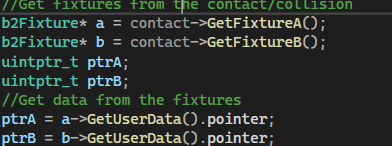


Box 2D functions are used to give a certain block the function to act as the ground for the player to stand on. First the texture for the object is made and the destination for the rectangular body is set with an x and y value which is taken then divided by 100, then the offset is taken by .160 in the x and y direction. After the body is created, a box 2d fixture is added with the dimension of .16f by .16f. Then the body is added to a 2D array of bodies to help set the stage clearly and allow for editing to occur easily.

## 2.3 Sensors

In Box2D, there is a class that can be implemented called contact listeners. They have virtual functions for contacts and collisions. The B2World which is our physics world will call these functions whenever a collision/contact occurs in the world. This is how we check if a GameObject is grounded or not. When a physics body is made for every Game Object, we also make a sensor for it. This is done by creating a b2Fixture then making the isSensor attribute true. We then place it on the bottom of the Game Object.

We also make sure to give the sensor a unique tag in the form of user data. This is to differentiate it from other fixtures in the physics world when we’re checking collisions. Next, we implement the contact listener functions BeginContact() and EndContact(). The BeginContact() function is called when a collision occurs in the physics world. The End Contact() function is called when a collision/contact stops.

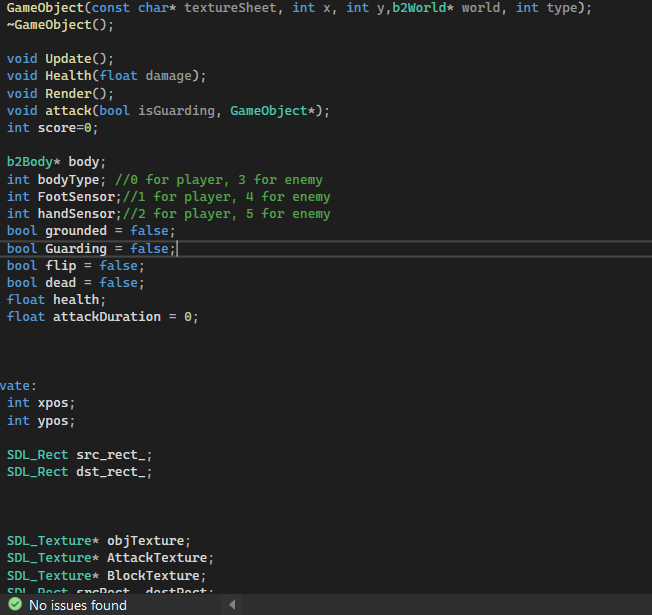


In the BeginContact() function, we get the fixtures involved in the collisions as well as the user data within them to access the unique tags we gave the sensors earlier. We then check if the tags match either player’s sensor tag or the enemy’s sensor tag. If we find the unique tag, then we know the player or enemy is grounded and we let the game know using a bool attribute in Game Object called ‘Grounded’.

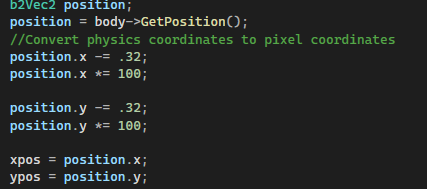
In the End Contact() function, we do similar steps for tag checking. This time, if we find the unique tag, we know the player or enemy is not grounded and we let the game know. This is all to check whether the player or enemy can jump as they shouldn’t be able to jump if they are not grounded as that enables infinite jumping or flying.

## 2.4 GameObjects

This subsystem was used to represent our two main objects in the game, the player and the enemy. They were each given attributes such as health, sensors, textures, physics, bodies, and a render function. The Game Objects include an update function to have their attributes constantly updated to ensure that everything is being kept up to date such as their health and positioning on the map. This subsystem also includes the attack function for the player and enemy to deal damage and alter their health.



Game Objects also have textures and other attributes for rendering purposes as well. This includes its standard texture, an attack texture that highlights when the Game Object is attacking, a block texture that displays when the Game Object is blocking. Finally, it also has x, y coordinates taken from the physics world and SDL\_Rect which signify where the object will be rendered on the screen.

In our update function, we do just this by using getPosition() and converting physics coordinates to pixel coordinates. Important to note that the physics coordinates are the same as pixel coordinates but divided by 100. For example, the coordinates (100,100) on the screen will be at (1,1) in the physics world. This is because Box2D does not function properly for objects with a size greater than 10. It essentially keeps Box2D stable by keeping numbers low and precise.

## 2.5 Texture Manager/Rendering

The rendering subsystem in our game engine utilized the SDL library to render textures on the screen. The process of rendering begins with initializing SDL and the creation of a window which is used to display the rendered textures. The renderer is then created using SDL’s rendering functions to draw textures. By loading image assets and converting them into SDL textures, various textures such as grass, dirt, water, player, and enemy were able to be displayed within our game engine. Utilizing functions such as SDL\_RenderCopy() rendering these textures is made possible. We also utilized a function called SDL\_RenderCopyEx() to render textures flipped. We do this to signify where the player is looking and where they are attacking.

## 2.6 Enemy AI

The Enemy AI function was set up using the same idea used in implementing the player movement with the addition in the logic for making him move independently. It checks the statues of the enemy game object on whether they are alive. It then proceeds to get the positioning for the player and enemy Game Objects, based on where the player is relative to the enemy the enemy will move in the positive or negative direction toward the player. The enemy is also supposed to use sensors to detect whether there is an object in front of him to avoid or if there is ground for him to continue walking forward on, if there is an issue the enemy is to jump or find another way around the obstacle.

This works together with the sensor subsystem as the enemy’s sensor is what is used to detect whether or not they are about to fall.

This subsystem also checks the distance to see if the player becomes close enough if the attack function should be executed to deal damage to the player to score a point. Every Game Object has an attack function that checks whether or not.

# Reports

## 3.1 Report 1

We have installed all the libraries we will be using now and may add more in the future. We also have a simple program to test SDL2 to get a window open. Zachary and William worked together on this part of the implementation, Ian and Gordon at the time tried helping with the coding for it.

## 3.2 Report 2

Ian Castillo: Created logic for player input for movements and actions William helped with implementing movement logic. Also investigated adding an audio subsystem. Idea was later scrapped due to debugging issues and time constraints.

William Torres: Researched box2D and added some physics code for use in the future such as defining the physics world and creating the ground and player bodies along with colliders. Designed Guy™ player model.

Zachary Urrutia: Created the foundation of the game engine, debugging code, figured out issues causing the ImGui library not to run properly, and helped everyone in linking the files correctly so that the window and GUI would run properly.

## 3.3 Report 3

Ian Castillo: Researched to help in designing map editor using and started in coding implementation, map will be saved into text file and load it for the map editor, along with starting to add code to add blocks and painting blocks for map creation, updated by Zachary for final map implementation to function properly

William Torres: Continued with and modified the previous physics code to test it with reasonable values. Player now collides with the ground. Added some code to handle player movement but incomplete, Ian helped with debugging. Worked/researched on adding images and displaying them as textures using SDL\_image.

Zachary Urrutia:began looking into implementing the camera subsystem using SDL. The goal is to allow the player to move around while keeping the game view focused on the player character. Later scrapped the idea.

## 3.4 Report 4

Ian Castillo: Worked on AI for enemy game object to have enemy move and get around obstacles to get to player game object, and adding functions for player input on attacking guarding and gameobjects health.

William Torres:Added player movement and physics. Allowed the map class to render physics platforms that the player/enemy can stand on. Added respawning mechanics as well (if the enemy goes off screen, teleport them to a platform). Ian later helped with debugging to fix issues.

Zachary Urrutia: Restructured format of the files and finalized the level design creation . Restructured the project by adding classes and header files instead of the game engine being just one file, as well as added map and object rendering.

## 3.5 Report 5

Ian Castillo: worked on helping fixing AI movement and fixing logic for guarding function for player and enemy with Williams help. Also began looking into adding health functions and created initial code using ChatGPT, and later had with help from William debugged to get working.

William Torres: Implemented contact listeners and helped with the enemy AI. Implemented sensors to detect when the player and the enemy is grounded.

Zachary Urrutia: Added the DearImGui to the main project. Implemented simple in-game map editor utilization DearImGui. Also looked into a system to save the map.

## 3.6 Final Report

This report is for the week of May after we presented and the issues we resolved before the final submission.

Ian Castillo: Worked with William on debugging issues with enemy AI, fixing issues with jumping and attacking, and the score system making all work better or function correctly. Also added implementation to save maps to map.txt file. Also discussed animations with William

William Torres: Worked on debugging issues. Also changed functions implementing box 2D to fix issues with frame rate, and gave final design for player and enemy and quick animation for them, adjusted how game object death works; Now properly destroy the game object along with its physics body.

Zachary Urrutia: worked on ImGui to fix the map editor to go with the new map.txt file. Also worked on animations. Added an FPS counter as well as a FPS cap.

## 3.7 Creators and Maintainers

### Main.cpp

Creator: Zachary Urrutia

Maintainer: William Torres, Zachary Urrutia, Ian Castillo.

William- Added and tweaked it to work with Box2D. Specifically added a b2World reference as well as adding physics world steps and its associate variables (velocity iterations, position iterations)

Zachary- Added FPS counter and FPS Cap.

Ian- Helped with debugging.

### Game.cpp

Creator: Zachary Urrutia

Maintainer: William Torres, Zachary Urrutia, Ian Castillo

William- Added the Box2D references and initializations as well as did the input handling that would later go into KeyboardInput.hpp. Added respawning located in the Update() function. Also, added CreateCeiling() and did half of the EnemyAI(). Helped with debugging RenderHealthBar() as well as RenderScoreSystem(). Did some adjustments to DeathCheck().

Zachary- Restructured input handling to accept ImGui and to another header file called KeyboardInput.hpp.

Ian- Did half of the EnemyAI(). Did most of the code for RenderHealthBar() and RenderScoreSystem(). Did first iteration of DeathCheck().

### Game.hpp

Creator: Zachary Urrutia

Maintainers: William Torres, Zachary Urrutia, Ian Castillo

We simply made changes to accommodate changes in game.cpp.

### GameObject.cpp

Creator: Zachary Urrutia

Maintainers: Everyone in the group.

William- Did most of the Game Object constructor. Did all the Box2D specific things. Added onto the render function to add block and attack animations. Helped debug the attack function as it was buggy at first. Adjusted update() to accommodate physics.

Zachary- Created the foundation of the Game Object class, allowing game objects such as the player and enemy to be created.

Ian- created the attack and health functions and helped with debugging issues with original iteration.

### GameObject.hpp

Creator: Zachary Urrutia

Maintainers: Everyone in the group.

We all adjusted the attributes and function headers to accommodate changes to GameObject.cpp.

### GUI.cpp

Creator: Zachary Urrutia

Maintainers: Zachary Urrutia

Zachary- created and maintained all of the function for this class.

### GUI.hpp

Creator: Zachary Urrutia

Maintainers: Zachary Urrutia

Zachary- created and maintained all of the functions for this class.

### KeyboardInput.hpp

Creator: Zachary Urrutia

Maintainers: Everyone in the group.

William- Created initial function for the keyboard inputs and debugged issues that later arose with added code related to this.

Zachary- Reformatted the file and helped with debugging issues.

Ian- add functions for attack and guard inputs and worked on debugging.

### ContactListener.cpp

Creator: William Torres

Maintainers: William Torres

William- Implemented and maintained everything in this class.

### ContactListener.hpp

Creator: William Torres

Maintainer: William Torres

William- Implemented and maintained everything in this class.

### Map.cpp

Creator: Zachary Urrutia

Maintainers: Everyone in the group.

William- Added all Box2D functions. Adjusted Draw() to create a physics platform on certain tiles and destroy physics platforms for others. Added b2World and platform bodies references.

Zachary-

Ian- Wrote the WriteToFile() and ReadFromFile() functions. Helped with debugging other functions. Added code for other tile types besides water, grass, dirt.

### Map.hpp

Creator: Zachary Urrutia

Maintainers: Everyone in the group.

We simply made changes to function headers and attributes to accommodate changes made in Map.cpp.

### TextureManager.cpp

Creator: Zachary Urrutia

Maintainers: Zachary Urrutia

Zachary- created and maintained all functions in this class.

### TextureManager.hpp

Creator: Zachary Urrutia

Maintainers: Zachary Urrutia

Zachary- created and maintained all functions in this class.

# Evaluation

## 4.1 Possible Additional Implementations

One thing would be a true animator function. We added in effect to create visuals for attacking and defending but not an animator function. Debugging tools is another function we did not quite implement in our design. Also, we left out a sound design function. We were also unable to complete the map editor function, the map can be saved onto a text file, but we ran out of time and were unable to complete the load function to bring in the maps saved on the file.

## 4.2 Working Parts

We have a functioning game with our map being set and the player moving based on inputs from the user. The Enemy AI is basic but does target and attack the player. The Physics for the game are fully functioning. Textures are correctly displayed in do intended functions. The health and scoring systems are properly displayed at the bottom of the screen for the player to see. We fixed FPS issues; it was related to our map creator function.

## 4.3 Non-Working Parts

Attacking works but has some issues where it can be buggy. The jumping is also a functional feature, but there are times where the player is in a spot and is unable to jump even though they are grounded. All sides of the player’s hitbox must be touching a platform to jump. The map editor works and can save to a text file but cannot be loaded.

## 4.4 Improvements to Add

With more time we could have better implemented the AI function, add in a sound component, an animation component, add more physics to the attack function, and finish the load function for the map.