

Technical Design Document (TDD)

07/10/2015 - \*\*/\*\*/2015

**Fallen Hero**

Authors: Josh Mooney

Student Number: C00165897

Project Supervisor: Ross Palmer

Honor Bachelors in Computer Games Development

Table of contents

[Introduction](#h.1aqbz3nsttp0)

[CRC cards](#h.7yu8mre0e9rr)

[Architecture](#h.gk4p7x3qf751)

[Approaches](#h.9gliq4j6da0p)

[Features](#h.oobjqwcga0bu)

[Setup Project](#h.gqmsbekpqqis)

[Game](#h.uivfzbcb57ir)

[Movement](#h.terfyjvefbxa)

[Jump](#h.t0yrmyfieq4i)

[Level](#h.r3moytttkzo8)

[WorldMap](#h.lraiphbsz2f7)

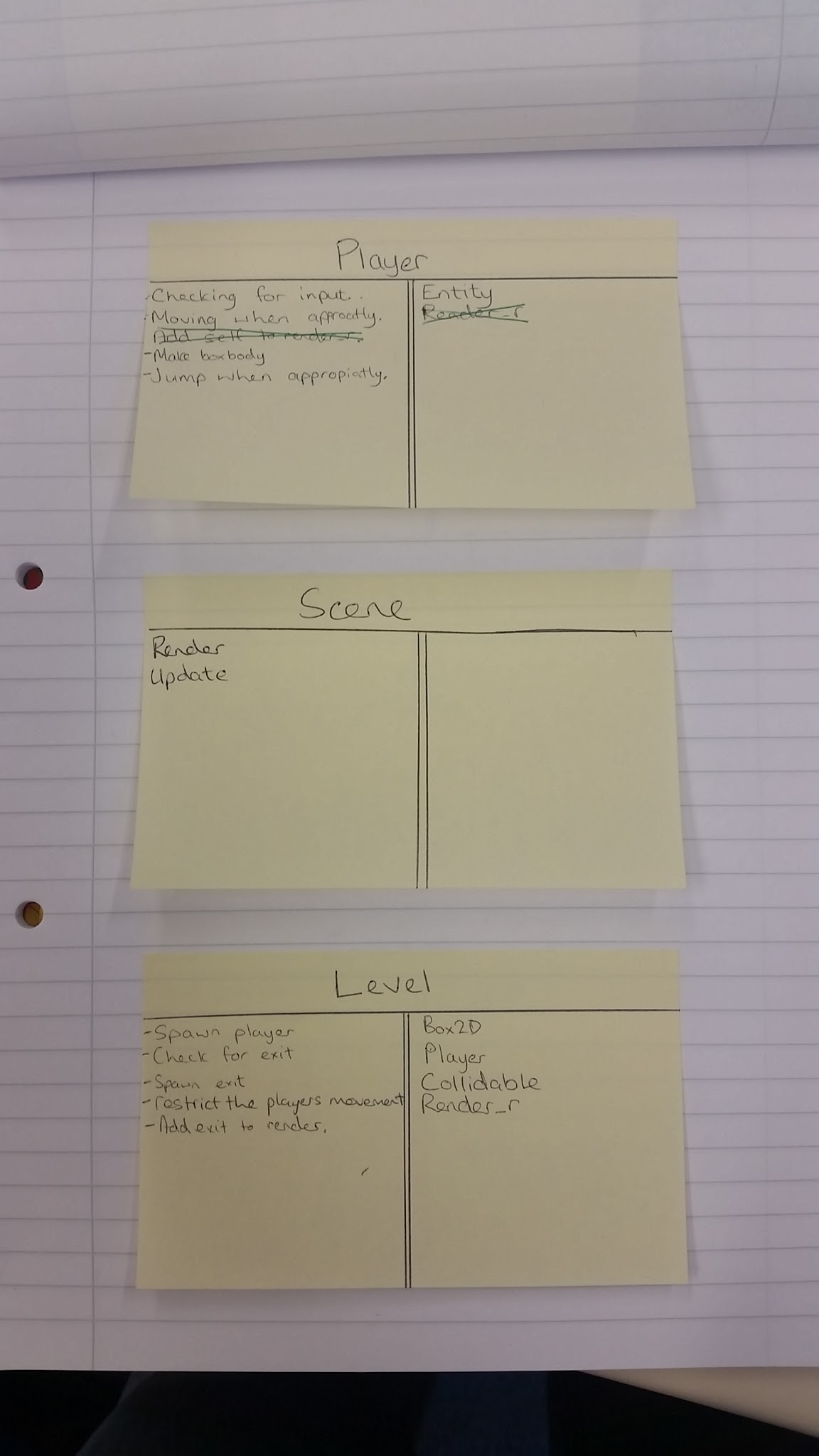
[Exit](#h.jbcis15nvhf2)

[Main Menu](#h.sfs9t8wp59w9)

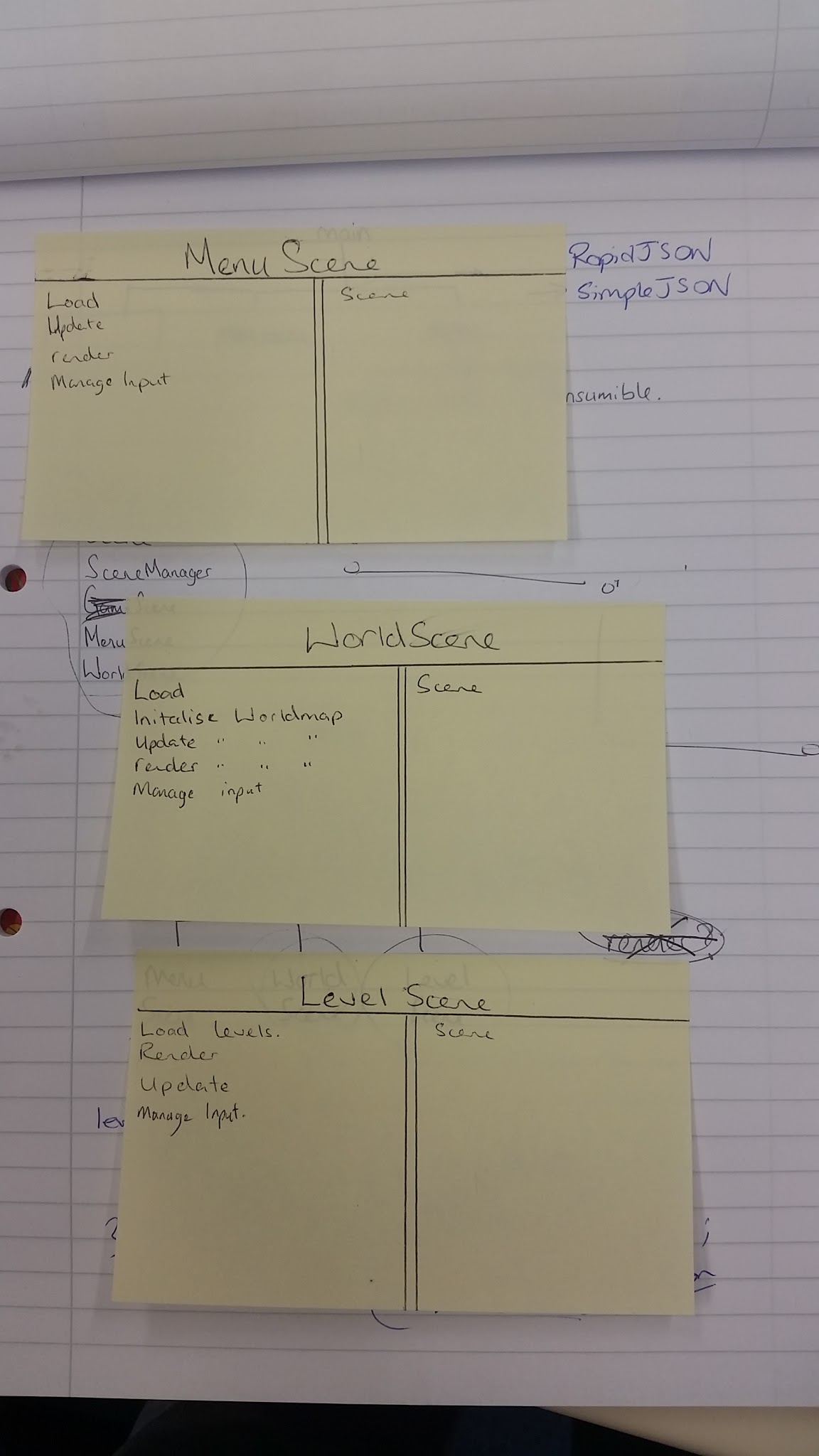
Introduction

## 

## CRC cards20151008_115248.jpg

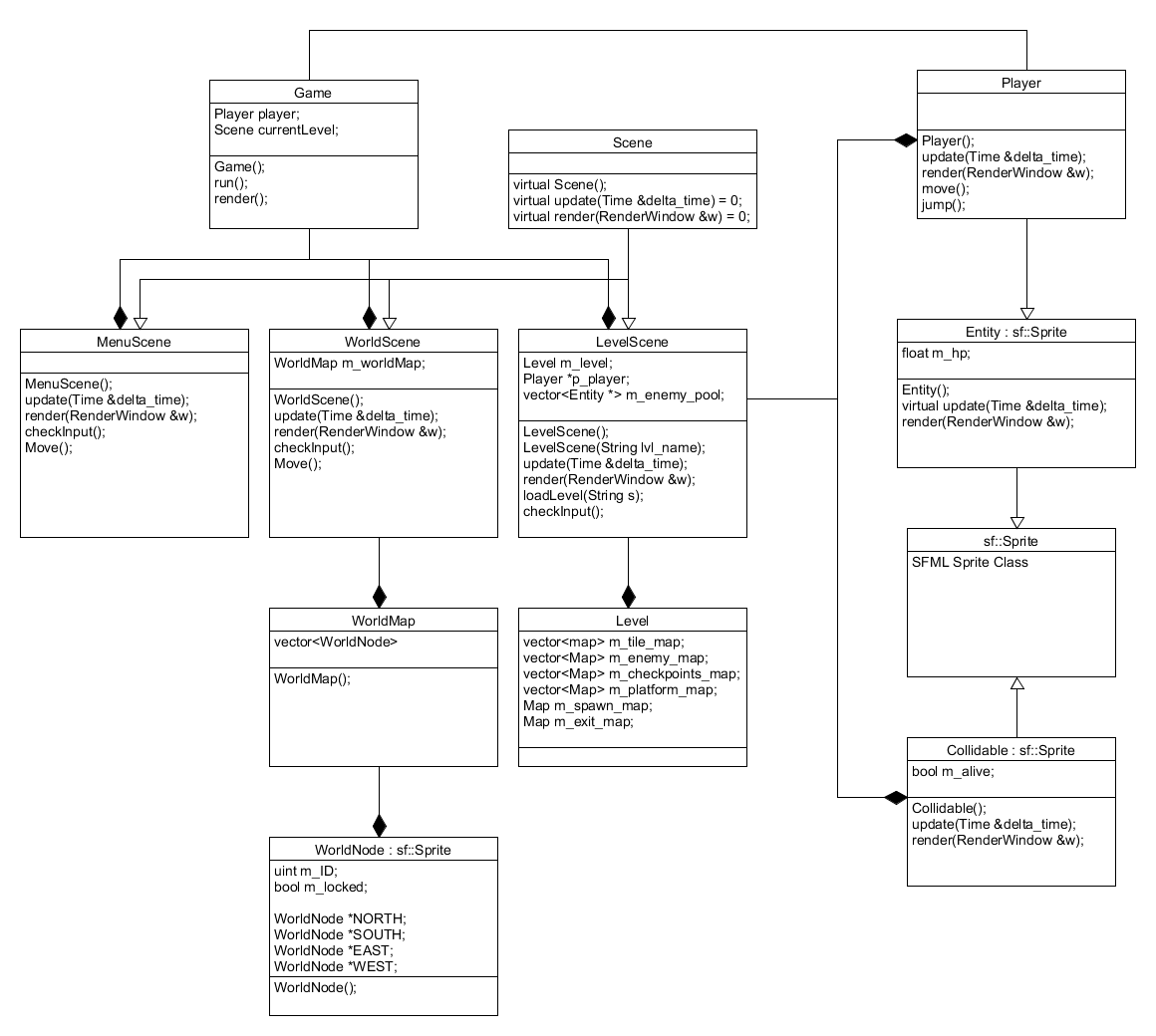


## 20151008_115301.jpg



## 

## 25Architecture



## Approaches

SFML is a widely used cross platform software development library designed to provide a simple interface for various multimedia components such as Rendering, Networking and Sound. I choose to use this library over a similar one such as SDL because my past experiences with SFML have gone quite well and I feel like it provides a great base for my project. I hope to take advantage of specific features within SFML such as: sf::View,

Box2D is a free open source 2D physics simulator engine written in C++. I choose to use this library as it will drastically reduce development time, I will be using box2D for Collision and managing my game world entities. Classes such as the base entity, base tile and collidable will utilize the Box2D library.

Boost Library

Singleton is a design pattern that restricts the instantiation of a class to one object. This can be useful when exactly one object is needed to coordinate actions across the game. Classes such as Resource Manager and Game will use this pattern.

## Features

### Setup Project

This is the initial step for starting off the project.

**Task 1 - Create Blank C++ console application**

* Open Visual Studio 2015.
* File > New > Project.
* Navigate the left column to templates and select Visual C++.
* Navigate the Center panel to “Win32 Console Application”
  + Rename project.
  + Select directory to save project to.
  + Click Ok and Finish.

**Task 2 - Add required external Libraries**

* Download Box2D and newest SFML.
  + SFML: (<http://www.sfml-dev.org/download/sfml/2.3.2/>)
  + Box2D: (https://github.com/erincatto/Box2D)
* Create environmental variable for each
  + <http://superuser.com/questions/949560/how-do-i-set-system-environment-variables-in-windows-10>
* Open properties of the project by pressing Alt + Enter.
* Navigate to C/C++ and under Additional Include Directories add the include folder for the environment variables (e.g. $(SFML\_SDK)\include)
* Navigate to Linker and under Additional Library Directories add the library files (e.g. $(SFML\_SDK)\lib)

**Task 3 - Add project to Git repository**

* Add files to repository
* Commit files to repository with commit message “Initial push”
* Push project

### Game

This is the Game class. This class will be a singleton and will manage every aspect of the game including updating, rendering and managing game objects. The main game loop will need to be created here also.

**Task 1 - Create Game Class**

* Methods
  + Game() //Default Constructor
    - Create Box2DWorld
  + run() //This will be the main update loop for the game
    - Update Box2DWorld
  + renderer(RenderWindow &w) //This will render all the game assets

**Task 2 - Create Game object**

Add gameobject to the main.cpp and call game.run() this will create the main game loop.

### Movement

The player class will need to move left or right, the movement will be restricted to the x-axis and the player will need to know when they are moving and register input.

**Task 1 - Create Base Entity Class**

* Inherit from sf::Sprite
* Variables:
  + float m\_hp;
  + sf::Texture m\_texture;
* Methods:
  + virtual update(sf::Time dt);
  + virtual render(RenderWindow &w);

**Task 2 - Create Player Class**

* Inherit from Entity
* Variables
  + bool m\_isMoving;
  + b2Body m\_boxBody;
  + float m\_speed;
  + bool m\_direction;
  + const float MAX\_SPEED;
* Methods
  + Player()
    - Create BoxBody
  + handleEvent(sf::Event event)
    - This will take in an event and check it for keyboard input
    - If Left key is pressed update direction and call move
    - Do the same for the Right Key
  + move()
    - Uses direction and speed to move the boxbody and player.
  + update(sf::Time dt)
  + render(RenderWindow &w)

**Task 3 - Create Player Object**

* Create Player Object in Game Class
* Call Player.handleEvent in Game.run
* Update player using Player.update in Game.run
* Render player using Player.render in Game.render

### Jump

The player class will be updated to include a jump function which will allow the player to move on the y-axis this must also include falling under gravity. This will require updating the handleEvent method. The player will jump with a force of 8.0f on y-axis. The player will be limited to a single jump but this can be extended through the upgrade tree. Gravity will be applied in a downward force. Certain levels will have gravity enhanced sections such as an underwater level where gravity will be reduced to 0.6f.

**Task 1 - Add handling variables to Player**

* bool m\_isJumping

**Task 2 - Add functionality to Player.handleEvent**

* handleEvent(sf::Event event)
  + This will take in an event and check it for keyboard input
  + If Left key is pressed update direction and call move
  + Do the same for the Right Key
* Jump()

### Level

The level will be responsible for containing all of the level data required for creating and managing the level. The Level will need to be created and loaded from its own scene for this reason LevelScene will have to be written this will be responsible for the loading of the level as well as the data within the level class to generate the world. The level will be updated and rendered from the levelscene also. LevelScene will inherite from a base abstract scene class.

**Task 1 - Create Level Class**

* Create class definition
  + #define vector<map> Level\_Data
* Variables:
  + Level\_Data m\_enemy\_data;
  + Level\_Data m\_checkpoint\_data;
  + Level\_Data m\_platform\_data;
  + map m\_spawn\_data;
  + map m\_exit\_data;
* Methods:
  + Level();

**Task 2 - Create Abstract Scene Class**

* Methods:
  + **virtual** update() = 0;
  + **virtual** render() = 0;
  + **virtual** handleEvent() =0;

**Task 3 - Create LevelScene Class**

* Inherit from Scene class.
* Variables:
  + Player \*p\_player;
  + Level \*p\_level;
* Methods:
  + update();
  + render();
  + handleEvent();
  + LoadLevel(string lvl\_name);

**Task 4 - Add Level Scene functionality to Game class**

* Variables:
  + enum STATE{LEVEL};
  + STATE m\_current\_scene;
* Add switch statement to implement level state in gameloop.

### WorldMap

The world map is the game hub and is used for level selection. The player should be able to move around a series of nodes traveling on a specific grid, each of these nodes will have a level ID associated to it. This will be used for level loading at a later stage.

**Task 1 - Create WorldNode Class**

* Inherit from sf::Sprite.
* Variables:
  + string m\_ID;
  + bool m\_is\_locked;
  + WorldNode \*p\_north;
  + WorldNode \*p\_east;
  + WorldNode \*p\_south;
  + WorldNode \*p\_west;

**Task 2 - Create WorldScene Class**

* Inherit from Scene
* Variables:
  + vector<map> m\_world\_map;
* Methods:
  + update();
  + render();
  + handleEvent();
  + loadWorldMap();

**Task 3 - Add WorldScene functionality to Game class**

* Variables:
  + enum STATE{LEVEL, WORLD};
* Add scene implementation to gameloop;

### Exit

Exit will be a simple collidable object that will have extra functionality in the game class. A Collidable is a simple class that holds a sprite and knows when it has been picked up and hada boost signal to represent methods to be classed when triggered.

**Task 1 - Create Collidable Class**

* Variables:
  + Box2DBody m\_box\_body;
  + bool m\_is\_collide;
  + boost::signals2::signal<void()> m\_signal;
* Methods:
  + Collidable();
  + destroySelf();
    - Connect m\_signal to this method.
  + collide();

**Task 2 - Update LevelScene Class**

* Variable:
  + Collidable m\_exti;

**Task 3 - Update Game Class**

* Add on collide with exit functionality to gameloop to return to exit back to worldScene
* Method:
  + onCollideExit();
    - Functionally for exiting a level and returning to the worldScene.
    - Connect Level.m\_signal to this method.

### Main Menu

Main menu will be how the user navigates the front end of the game interface. To create this functionality a MenuScene class will be created. This will update, render and handleEvents for the menu and its state. There will be options menu and the functionality to see a splash screen, continue to a menu interface and be able to continue to the world map scene.

**Task 1 - Create MenuScene Class**

* Inherit from Scene
* Variables:
  + sf::Sprite m\_splash\_scr;
  + sf::Sprite m\_menu\_bg;
  + sf::Text t\_play;
  + sf::Text t\_exit;
  + int m\_current\_item;
  + bool m\_is\_splash;
* Methods:
  + MenuScene();
  + update();
  + render();
  + handleEvent();
  + Move();--

**Task 2 - Add MenuScene functionality to Game Class**

* Variables:
  + enum STATE{LEVEL, WORLD, MENU};
* Add scene implementation to gameloop;