Technical Design Document (TDD)

Authors: Darren Sweeney, D.J. O’ Leary

Project Supervisor: Philip Burke

[Introduction](#_b1jd59vhbdtz)

[Technical Goals](#_844gp3hhlaaq)

[System Requirements](#_6lzenemc4h6d)

[CRC cards](#_72trjrv80ut9)

[Architecture](#_yn7g5aglwc3a)

[Code Overview](#_tdlqb7xveifo)

[Main game loop](#_phwvrsmqwc8t)

[File formats](#_j7c1y0i31nm2)

[Comments](#_wiqldvf48ga6)

[Naming conventions](#_8wsz1kiqvnnp)

[Coding guidelines](#_pavirnibih06)

[Source control](#_4iaeracd47kf)

[Feature Design](#_lxltp4ogdjr6)

[Setup Tasks](#_o1yngxtohjuc)

[(Feature 1) Player controller movement.](#_spna9l5obie)

[(Feature 2) Shooting](#_ic62zfh3ixlv)

[(Feature 3) GUI System](#_lq6z8sbytncr)

[(Feature 4) Main Menu](#_ee4l98tbewfb)

[(Feature 5) EntityManager](#_9youstup3uf0)

[Audio](#_roy0whocy0k1)

[Features](#_2e0f0uwny33j)

[Formats](#_dxiz1kvhcbrj)

[Graphics](#_xsmnw9lx81op)

[Features](#_fq1olt4gid2u)

[Formats](#_aqloasmn6f6f)

[Textures](#_vtblcpyij4p)

[Vertex and Fragment Shaders](#_agg7oa5lwtoa)

[View modes](#_j2nn96qat5s6)

[Lighting](#_xpv5m0kkfkqm)

[Physics](#_n8mxbx57y6hk)

[Features](#_8t6tyh2c9m9m)

[Background grid](#_mte5xt1tbtip)

[Particle System](#_hokh7pkc94h9)

# 

# 

# Introduction

The purpose of this document is to outline the technical design of our game. The game will have to be designed with performance and fluid gameplay in mind. This is greatly tied to the gameplay experience. Here we will examine the implementation details of features such as player movement, particles effects and menu system. We outline technical goals that need to be reached and different systems that make up the game.

# Technical Goals

The following goals need to be meet and addressed early in the development. These goals will determine the look and feel of the game based on how well they are executed.

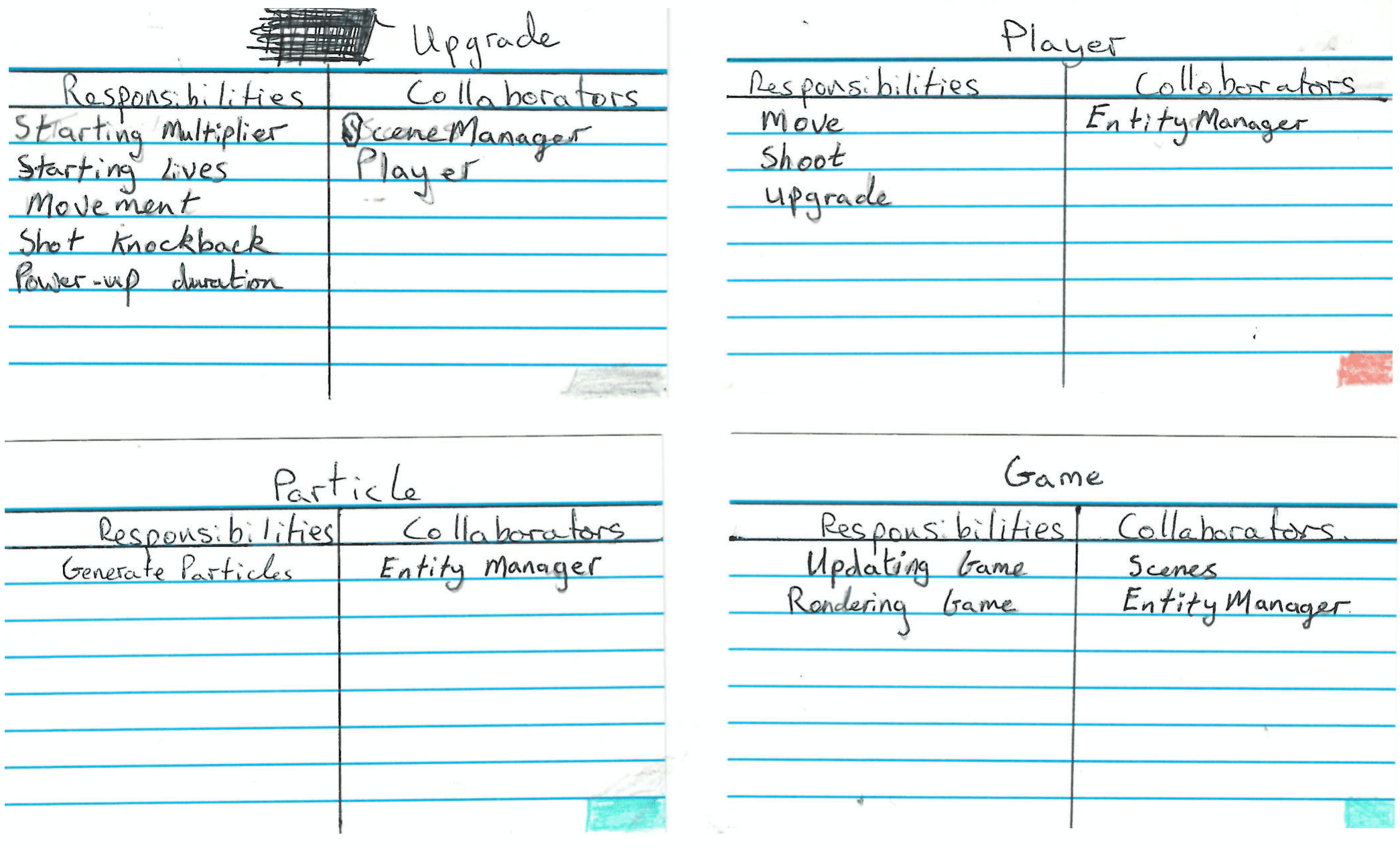
* Bloom lighting for hundreds of particles and entities on screen.
* Seamless smooth menu navigation between different option screen states and gameplay.
* Particle physics simulation of explosions using a circular buffer.
* Cloth spring physics for the world grid.
* Camera follow around game world.
* Smooth linear and rotational movement of player to target location and rotation.

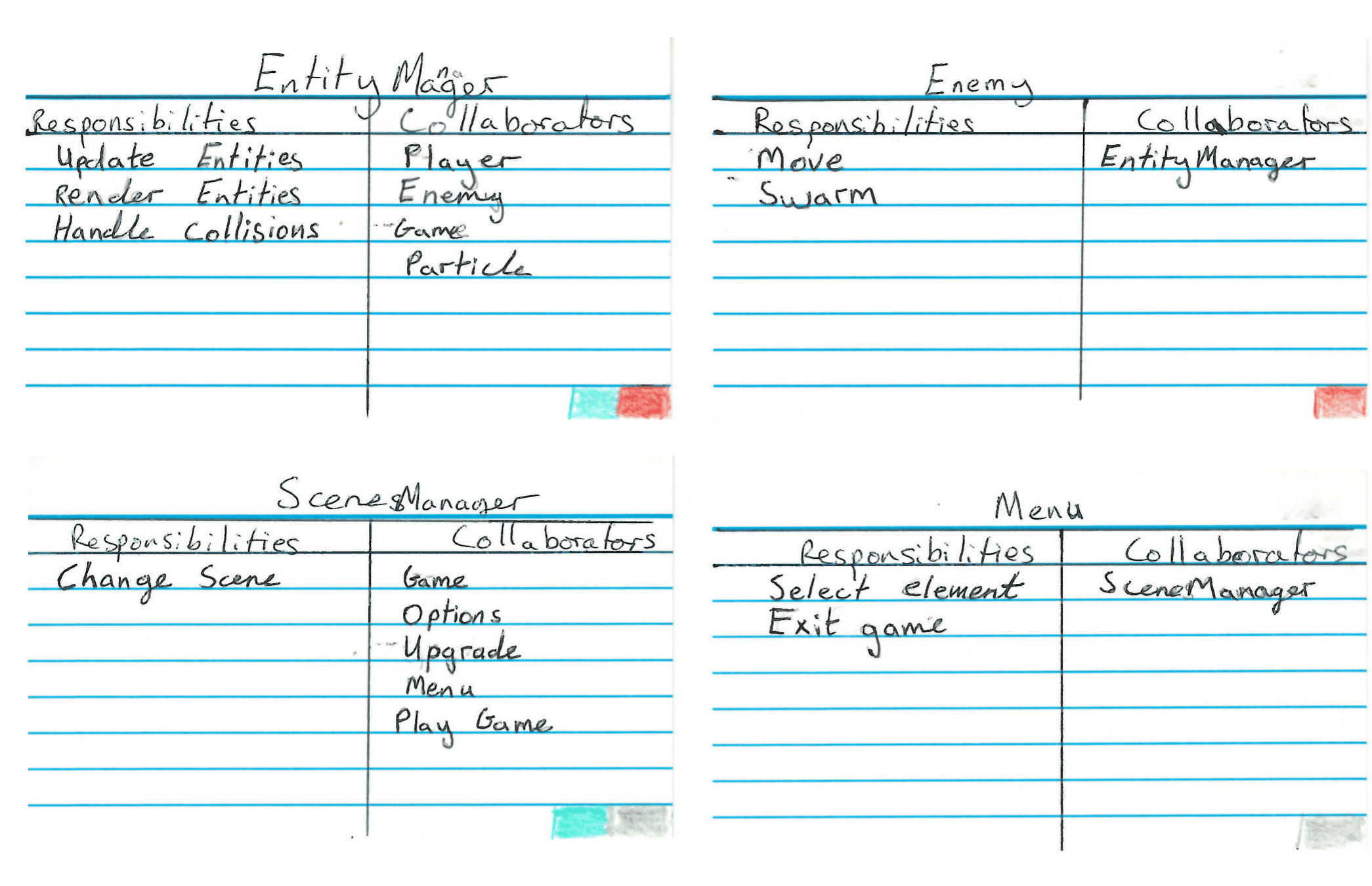
Implementation details are discussed more in [Feature Design](#_lxltp4ogdjr6)

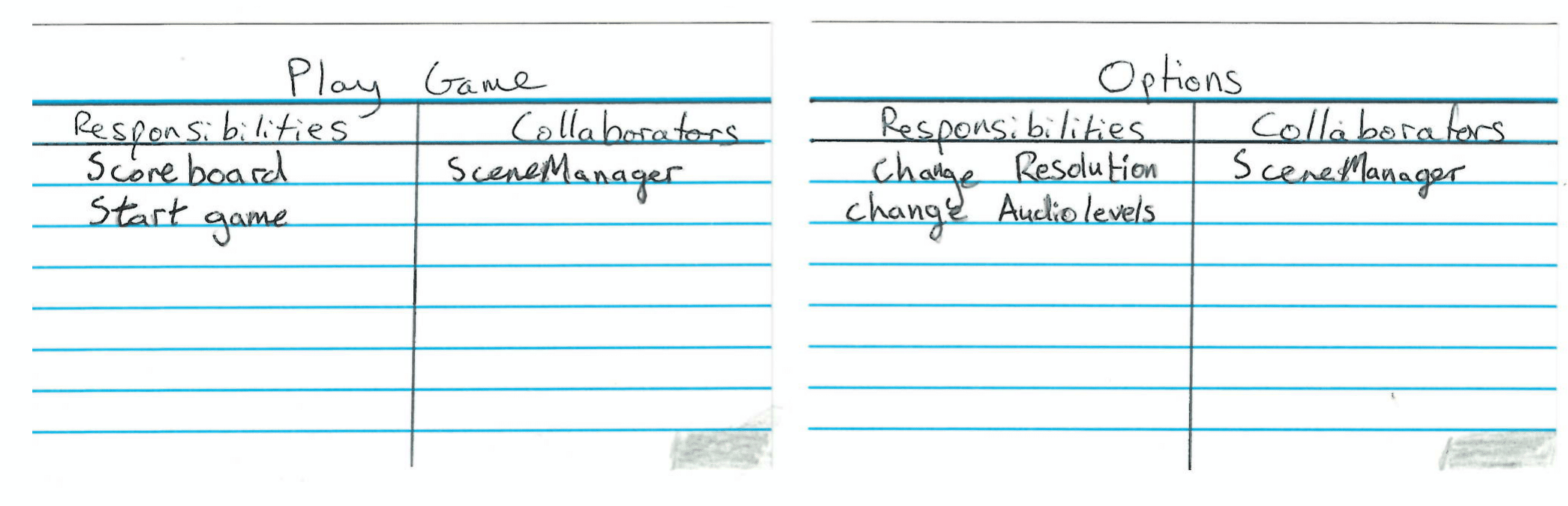
# System Requirements

Windows 7 and up, 1 GHz 32-bit (x86) or 64-bit (x64) CPU, 512MB RAM, OpenGL 3.1, OpenGL Video Card (128MB memory - Shader Model 2.0 support required), DirectSound-compatible sound card, 150MB free HD space.

# CRC cards

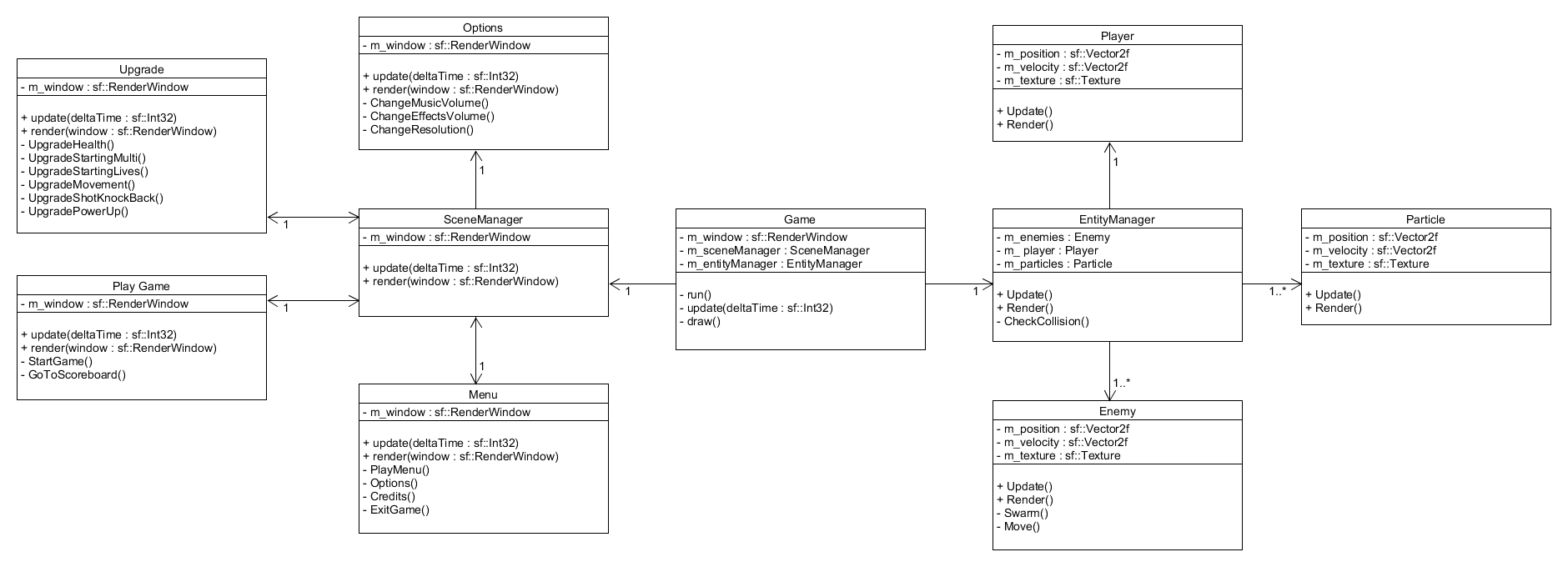






# Architecture

UML class diagram showing relationships between classes identified with CRC cards

****

# Code Overview

## Main game loop

The main game loop will exist in Game.cpp. Each system such as entity manager and particle manager will have three functions called here.

## File formats

All code should be contained in standard C++ source files (.cpp) and standard C/C++ headers (.h) with Author/Contributors names at the top.

## Comments

Single line comments should be used as often as possible to elaborate code flow. Block style comments for mode technical difficult implementations to understand at first glance.

Specific marker tags:

* “@todo(name): ” Indicates where the programmer has skipped some functionality to work on something else that needs to be done first.
* “@refactor(name): ” Indicates that the current state of the code needs a rethink.
* “@note(name): ”: Indicates an important note that should be considered by any programmer looking at the line or lines of code.

These specific markers tags are very important. They provide easy search as to what needs to be done. The name tag is also important. It shows who left the tag. A single detailed line after the colon to explain the tag is a must.

## Naming conventions

All naming conventions must apply with the The Institute of Technology, Carlow programming standard. The standard can be found here: [Link](https://drive.google.com/open?id=0B6gny4LvztHhR0dMTE9icDdKNnM)

## Coding guidelines

* Don’t solve the general problem. Solve the problem that is needed.
* Unnecessarily virtual functions should be rewritten to remove virtual requirements.
* Use forward declaration as much as possible in headers .
* The game should be running all the time, every commit **must** compile and run.
* A Release build and Debug build should be tested after each feature.
* Sloppy code or “hacks” must be marked with @refactor(name).

## Source control

All code must be committed to this branch: [Link](https://github.com/ITCGamesProg2/project3-darren-dj)

Programmer must ensure that there username and email is associated with the commit otherwise, the local machine name will be used and the contributor will not have that commit associated with their account. More information here: [Link](https://help.github.com/articles/setting-your-username-in-git/)

For each different feature a branch must be made and **deleted when merged**. For more information on branches check here: [Link](https://help.github.com/articles/creating-and-deleting-branches-within-your-repository/)

# Feature Design

## Setup Tasks

**Task 1: Initial documentation**

* Read.me - Brief description of the project, set up instructions if any.

**Task 2: Initial Application stub**

* Create simple application which loads image, font and sound as a test.
* Test release and debug build.
* Create local repo & github repo.
* Push Repo to remote.

**Task 3: Each team member**

* install sfml dependencies.
* pull from remote.

## (Feature 1) Player controller movement.

**Task 1: Setup controller support**

* Implement controller support and test each button and axis input with xbox controller

**Task 2: Create base class entity**

* Player and enemies will all inherit from this base class
* Base class entity should have functions such as render and update that can be overridden.

**Task 3: Create basic player representation on screen**

* Represent player as simple square and test render and update functions

**Task 4: Create Game loop**

Updates the entities, delta time, clears window and draws. Test this is working by applying a fixed force to player on key press.

* void Game::processGameEvents()
  + Update the Keyhandler object
* void Game::update()
  + Player::update(double dt)

Game::render() is called from Game::run() every update.

### 

**Task 5: Set Up controller handler in player**

Store xbox controller object as property of Player class. Add method:

* void Player::handleInput()
  + Use controller object to check what keys are down and take appropriate action. Call handle input in the update method.

**Task 6: Give player maximum acceleration & velocity**

Adjust player so accelerates uniformly to 3 units/second^2, and velocity is capped at 7 units/second and decelerate to rest at -3u/s^2 if no keys pressed

* Player::update(double dt)

## 

## (Feature 2) Shooting

**Task 1: Implement controller support**

* Add controller support and test each button and axis input with xbox controller.

**Task 2: Create a bullet class**

* Create a class that stores the bullets position and velocity.
* Should have functions such as render and update.

**Task 3: Render bullet**

* Represent bullets on screen with a image.
  + Void bullet::render(sf::RenderWindow)

**Task 4: Update bullet**

* Updates the bullets position on screen using its velocity.
  + Void bullet::update(dt)

## (Feature 3) GUI System

**Task 1: Create a base class widget**

* This widget will have a set position method, set text, and a draw function that can be overridden.

**Task 2: Create a button widget**

* Button class inherits from base class widget.
* Store a callback function, which will be called when the button is pressed.
* Pass the controller into process input
  + Bool Button::processInput(XboxController &controller)

**Task 3: Create a GUI manager**

* The GUI manager will handle all gui elements in the game.
* The GUI will have an add function which allows widgets to be added
  + Void Gui::add(Widget \*widget)
* Create a process input method so all widgets can handle controller input
  + Bool Gui::processInput(XboxController &controller)
  + Check all widgets and pass the controller

**Task 4: Test GUI manager**

* Create a button widget and add it to the gui manager.
* Render and Update the button using the gui manager.
* Test to see is the function callback called

## (Feature 4) Main Menu

**Task 1: Create a main menu class**

* Create a menu class

**Task 2:**

* Store a reference to the GUI manager

**Task 3:**

* Store each GUI widget as a member and add it to the GUI manager

**Task 4:**

* Render each widget

## (Feature 5) EntityManager

**Task 1: Add entities to the manager**

* Store a pointer to each entity to a list.

**Task 2: Update entities**

* Entities are update through the manager.

**Task 3: Render entities**

* Entities are rendered through the manager.

**Task 4: Entity collision**

* Collision between entities checked using the bounding boxes and position of each entity.

# Audio

## Features

The SFML audio utility offers a range of features for playing sound. The features that our useful to use which we will use are as follows:

* 2D sound effects
* Set pitch, volume, attenuation and pause.
* Stack music tracks.

## Formats

SFML supports .mp3 and .wav file formats. .wav format will support signed 16-bit PCM and .mp3 will support quality of 170-210 kbps.

# Graphics

## Features

We will be using SFMLs graphics module along with OpenGL vertex and fragment shaders. SFML provides features that let us get things rendered on screen quick as possible with us writing our own renderer from scratch. Features such as:

* Rendering 2D text to screen.
* Render images and sfml graphics to screen.
* Ability to create custom drawable objects.

## Formats

#### Textures

Texture loading is done through SFML’s library. So all common formats are supported.

#### Vertex and Fragment Shaders

Vertex and fragment shaders will be created as standard .vs and .fs files respectively. The shaders will compile using SFML’s OpenGL’s shader utility.

## View modes

We will have screen resolution independent rendering. We will be able to achieve this by rendering the game to a render texture and scale that texture up and down with the window. For windowed and fullscreen mode we will be using SFML’s built in method for create a new window context.

## Lighting

Lighting in our game will consist of adding bloom to all objects and UI element in the game. Instead of applying a bloom shader to each enemy and increasing the draw calls on the GPU we will be rendering the entire scene to a render target and then apply bloom to that render target.

Bloom will be achieved by first extracting the bright parts of the image, blurring the bright parts then removing the blurred image with the original image while doing some brightness and saturation adjustments.

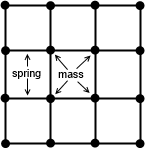
# Physics

## Features

The physics in the game will be our own custom systems. Particle effects and the background grid will be custom coded. We will have to create features such as:

* Spring damping physics for the grid.
* Explicit euler integration.

## Background grid



These springs will only pull and never push. To keep the grid in position over the game border the masses will be anchored in place on the border.

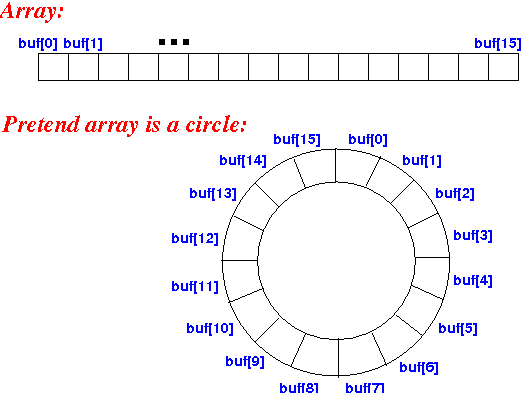
The spring connects two point massed and if stretched past its natural length a force will be applied pulling the masses together. The spring will follow a modified version of Hooke’s Law with damping.

**F = −kx−bv**

* F is the force produced by the spring.
* k is the spring constant, or the stiffness of the spring.
* x is the distance the spring is stretched beyond its natural length.
* b is the damping factor.
* v is the velocity.

We can optimize the grid by improving the visual quality by simply making the grid denser by adding line segments inside the existing grid cells. Without having to simulate these grid point. We do so by drawing lines from the midpoint of one side of the cell to the midpoint of the opposite side.

## Particle System



The particle system for the game will use a circular array, this way we can allocate the memory on startup and simple reuse the memory as particle spawn and die. This is so we can avoid memory allocations during frames that have lots of objects on screen.

## 