Technical Design Document for:

**Warp Break**

Authors: Emmett Byrne, Przemek Tomczyk

[**Architecture**](#_57ekneffy6fe) **3**

[Class design diagrams](#_toi6crud00q2) 3

[Sprint 1 diagram](#_mstnpuqcb5ho) 3

[End of Project diagram](#_fxadpuv2nw0j) 4

[**Technology**](#_n27vosbvdxw) **8**

[Research](#_e0c4kpz728jo) 8

[Installation](#_wyntc4sx5j99) 8

[Technical Achievement](#_toh1invww1qe) 9

[**Sprint Reports**](#_u4x85atus0ct) **10**

[Report 1](#_ozqfsr9kqh2s) 10

[Feature Design](#_v074v6i59ql5) 10

[Report 2](#_tc9vjnyyq8b3) 12

[Feature Design](#_d3lg2f7w41bj) 12

[Report 3](#_6k8v3u3dfen8) 14

[Feature Design](#_zhevmicbwa6) 14

# 

# Architecture

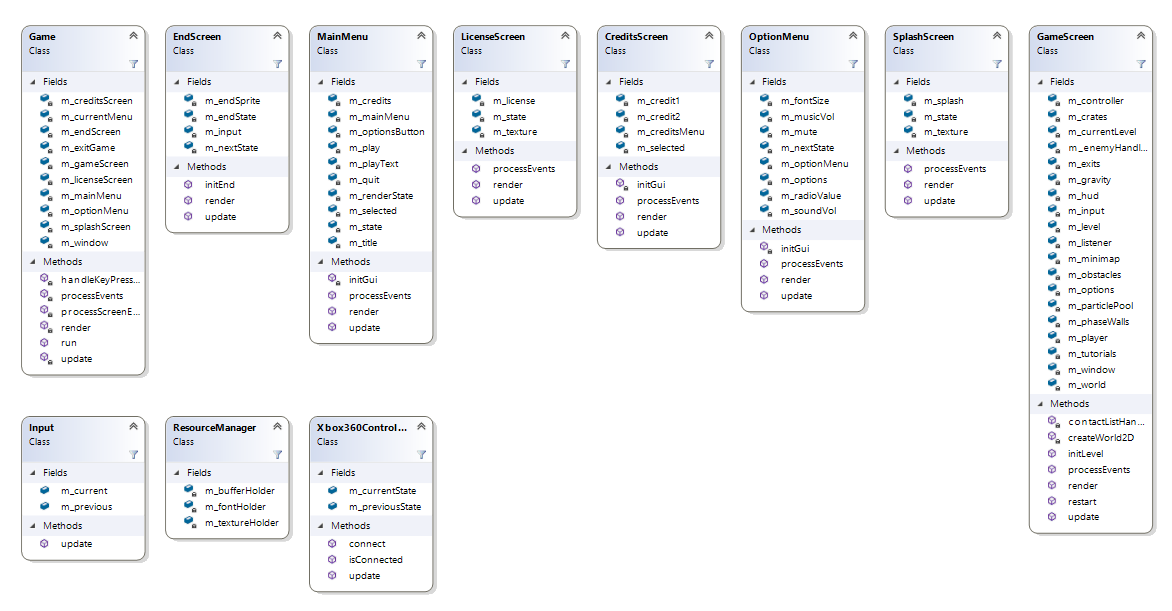
## Class design diagrams

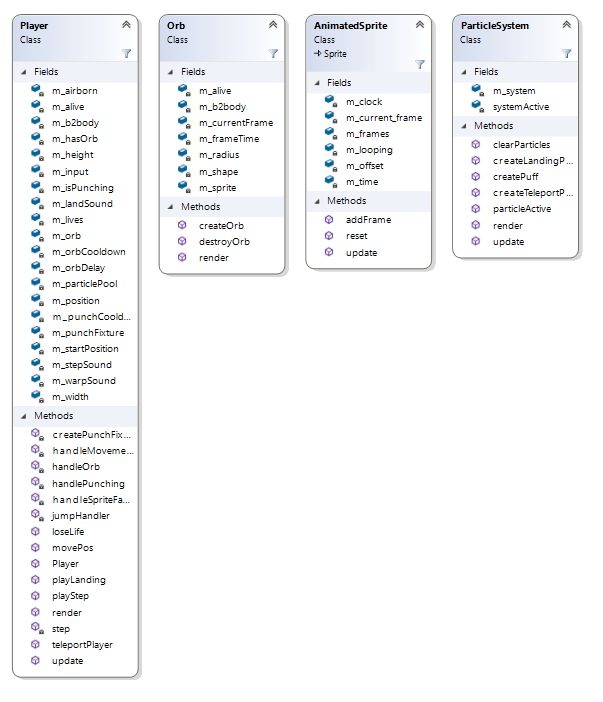
### Sprint 1 diagram

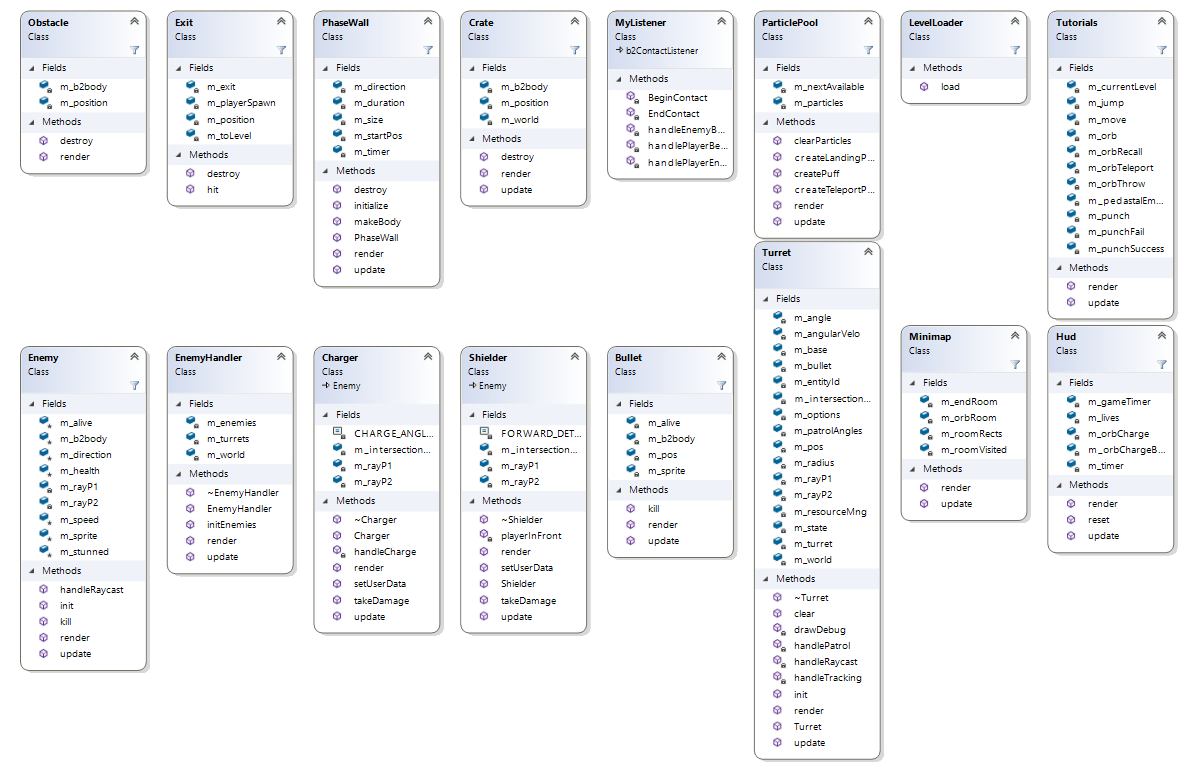
### 

### End of Project diagram

# 







# Technology

## Research

TinyC2: we originally used this collision library in our game. We had major problems at the start of the project with either colliding objects sticking together or jittering like crazy which can be seen in our sprint 2. We researched different collision algorithms/libraries but they were either too simple for our needs or was missing few things meaning we would still have to make our own collision response handlers.

Box2D: we brought this into the project about halfway through as we were having major issues getting collisions, collision resolve/response and physics working in our game. This opened up a lot of possibilities to us like collision filtering, simulating physics on bodies like crates and orb. Box2D handled everything we required and a lot more which we ended up using.

TinyFSM: we brought this library to handle our animation states and transitioning between different animations.

We used a Menu library/system we created as a project in 2nd year which ended up doing a lot of the stuff we needed for the main menu and other screens. It would’ve fit the colour scheme/palette a lot more if we changed the colours of the buttons but we were not too worried about that was just colours.

YAML: we didn’t really consider any libraries for loading levels until we made our first level and decided that the time we will spend adding YAML and creating nodes to fit our purposes will take a lot less time than creating each level by hand and having magic numbers all over the place. On top of that, it allowed us to edit levels on the go by reloading the level using debug functions rather than rebuilding the project every time.

## Installation

**CMAKE:** <https://cmake.org/>

This program is used to generate files for IDE of your choosing which then has to be used to generate needed files such as release/debug libraries (.lib) and dynamic-link library (.dll)

**tinyFSM:** <https://github.com/digint/tinyfsm>

Just simply include tinyFSM.hpp where needed

**YAML**: <https://github.com/jbeder/yaml-cpp>

Used CMAKE to generate a VS project and then build library files for debug and release versions for our project.

**Bromeon Thor**: <http://www.bromeon.ch/libraries/thor/download/index.html>

CMAKE was also used to generate a VS project and then build a release and debug versions of a library and dll

**Box2D**: <https://box2d.org/downloads/>

minGW was used to generate CMAKE files and then a release and debug libraries were generated using VS

## Technical Achievement

**Emmett’s technical achievement:**I feel like my technical achievement was the implementation of the more advanced features of Box2D such as the Contact listener. The contact listener listened for when a collision began and ended each each of our box2D bodies were set up with userdata that held and id for it’s parent object and pointer to it so that when a collision occurred we were able to easily control the objects that were colliding for example it made it very easy to make the player die when shot etc.

I really like how the Contact listener came out it made coding the various collisions that happen in the game quick and easy which made our lives much easier when making the project.

While I do feel that it came out quite well especially given that I had never worked with box2D before I do think it could be improved still especially to fix one of our only known bugs where the player will get locked in it’s falling state if it’s touching the ground of multiple bodies. I know exactly how the bug happens but was never able to fix it.

**Przemek’s technical achievement:**

I took on creating a base polymorphic Enemy class which was used for most, not the Turret as it doesn’t have a Box2D body, enemies allowing for a single vector of Enemy pointers with the objects’ allocated on the heap. This allowed to keep the EnemyHandler class much cleaner as it required a single for loop in init, render and update. Since we were passing in an enemy type using yaml and then setting it inside Enemy, we knew what each base pointer was pointing at, allowing for very simple casting and calling specific functions out of Shielder or Charger.

I was very happy how it turned out as we ended up with no memory leaks using this and if we had more time, we could’ve used this to very easily and quickly add new enemies to the game without big changes/updates to the EnemyHandler.

I feel this was a good technical achievement as it allows for the game to be easily expanded upon by adding more unique enemies but I do feel it wasn’t as technical nor challenging as some of the Box2D features we have used in the project but that’s also because we have never used Box2D before.

# 

# Sprint Reports

## Report 1

### Feature Design

**Feature 1: Player Movement**

Task 1:

Add in player movement and jumping logic based on velocity.

* void handleMovement(float t\_deltaTime)
  + Takes a delta time and updates player velocity and position

Task 2:

Make player movement based on friction and gravity.

* handleMovement function edited to handle friction and gravity

**Feature 2: Animation Handling**

Task 1: Load Sprites

Store frames in the AnimatedSprite class. Add Method:

* void addFrame(IntRect&)
  + Takes an IntRect and adds it to a vector of IntRects

Task 2: Update Animation

Iterate through the vector of Intrects. Add Method

* void update()
  + Increments an int that holds the current frame number

Task 3: Render the Animation

The animated sprite class returns the current frame.

* const IntRect& getFrame(int)
  + Returns an intRect of a specific frame

**Feature 3: Finite State Machine**

Task 1: Import TinyFSM

Import, setup and learn the TinyFSM library

Task 2: Add Events

Add every event that can be received by each state.

Task 3: Add States

Add each state, the Entry and exit functions and the reactions to each event they can receive.

**Feature 4: Menus**

Task 1: Import menu system

Add a previously made menu system to our project.

Task 2: Adjust the menu system

Adjust the menus to work for our project and each state.

## 

## Report 2

### Feature Design

**Feature 1: tinyc2 Implementation**

Task 1:

Implement tinyc2

* Add tinyc2 library into the project

Task 2:

Implement tinyc2 debug

* Add tinyc2 debug so it can be used to draw aabb for debugging

Task 3:

Give interactable objects a tinyc2 aabb collision boxes

* Add private aabb member variables
* Add public getters and setters

Task 4:

Make player hit box adjust depending on animation frame size

* void setSize()
  + Adjusts aabb by getting IntRect from AnimatedSprite class

Task 5:

Check and resolve player - terrain/platforms collisions

* void checkCollisions(aabb t\_playerBox, aabb t\_obstacle)
  + Takes in player’s aabb and an obstacle’s aabb
  + Checks if they’re colliding and returns true/false
* void resolveCollision(aabb t\_playerBox, aabb t\_obstacle)
  + Takes in previous aabbs
  + Checks what side player is on to the obstacle
  + Resolves the collision by moving player back slightly
  + Depending on which side player is on, their velocity is reset etc

**Feature 2: Add a particle system**

Task 1:

* Setup the thor Particle library
  + Add ParticlePool.h and ParticleSystem.h and edit them to work for the project.

Task 2

* Edit the finite state machine to return when a step is taken by the player.

Task 3

* Create particles at the point where the players foot hits the ground.

Task 4:

Play matching sounds with particles

* void playStep()
  + If step sound has finished playing, play step
* void playLand()
  + If landing sound has finished playing, play landing sound

## 

## Report 3

### Feature Design

**Feature 1: Box2D Implementation**

Task 1:

Implement Box2d

* Add Box2d library into the project

Task 2:

Remove tinyc2 from the project as it is no longer required

**Feature 2: Add Box2D collision listener**

Task 1:

Add a box2d collision listener class

* Add a MyListener class that inherites b2ContactListener to handle collisions

Task 2:

Add methods that handle collisions between certain objects

**Feature 3: Add the ability for the Player to punch**

Task 1:

Add a fixture sensor to the player that is created and destroyed when the appropriate key is pressed

* Edit the InputHandler to check for the punch key.
* Add a fixture sensor to the player
* Add a timer and bool for the player to handle when the player is punching.

Task 2:

Edit collision listener to listen and handle collisions between the player punch and enemies.

* Edit the player collision method to handle the collision between the punch fixture and enemies.

**Feature 4: Player jump and punch animation**

Task 1:

Add the jump and punch animations to the FSM and trigger the animations when appropriate

**Feature 5: Enemy base class**

Task 1:

Add a base class Enemy that all enemies will inherit from

* Add virtual and pure virtual functions
* Box2D body etc

**Feature 6: Shielder enemy**

Task 1:

Implement Shielder enemy

* Add a Shielder class that inherits from the Enemy class
* Implement basic logic for the Shielder enemy for patrolling

Task 2:

Implement taking damage from behind and knockback

* Edit player collision method in the listener to check for collisions between the player and the enemy

Task 3:

Expanding Enemy class

* Add raycasting to Enemy class to prevent enemies walking into walls and off ledges
* Filter out raycast collisions with Player and Orb

**Feature 7: Charger enemy**

Task 1:

Implement Charger enemy

* Add a Charger class that inherits from the Enemy class
* Implement basic logic for patrolling

Task 2:

Implement unique charging behaviour

* When horizontal raycast hits Player, Charger starts a windup in place and then rushes at the Player

**Feature 8: Level Loading with YAML**

Task 1:

Implement YAML

* Add YAML libraries

Task 2:

Make LevelLoader class to load YAML files

* Make PlayerData, EnemyData, ObstacleData etc structs that hold position, size and other information
* Load a YAML file and use LevelData to set up levels using YAML

**Feature 9: Levels**

Task 1:

Add a number of levels to the game

* Add a number of level yaml files.
* Populate each file with walls, enemies, exits etc

**Feature 10: Orb**

Task 1:

Add the ability for the player to throw a teleportation orb.

* Create a class for the orb to handle it’s Box2D bodies etc.
* Add Orb class to the player and allow the player to throw the orb.