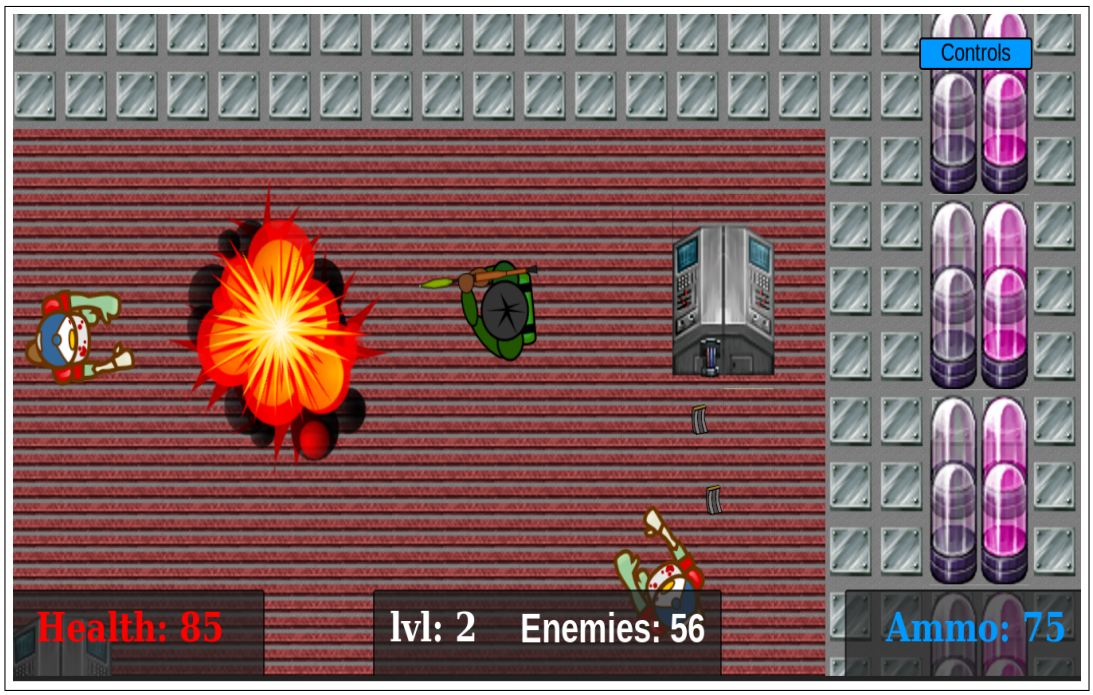
UMBC CMSC 345 - SOFTWARE ENGINEERING

Four to the Fifth Game Project Report

Kevin Coxe Rhea Horton Kasey White Chris Laverdiere
Chiebuka Ezekwenna

May 9, 2014



A Team 4 Production

1 Introduction

1.1 Overview

The goal of this project was to create a two-dimensional, top-down perspective, web-based video game. The genre is partially a shooter, as the player controls an avatar with various weapons, and also a role-playing game, as the player's power increases with each level completed. It is a simple, arcade-like game, by modern standards, and is completable within a thirty minute time frame.

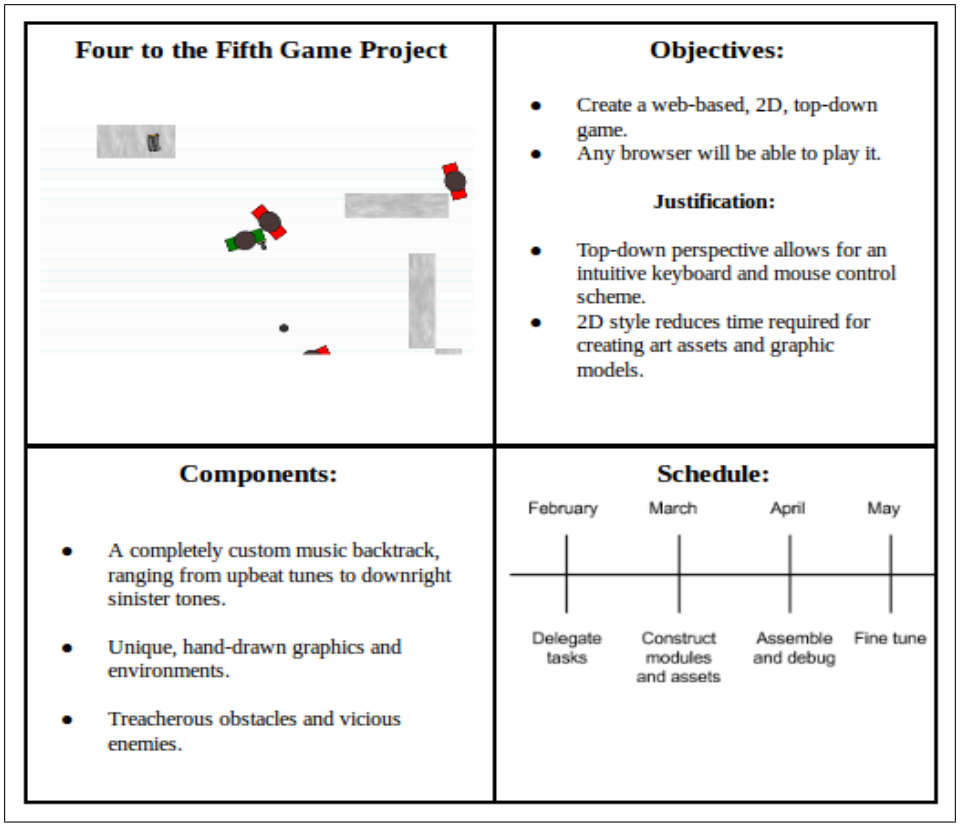
1.2 Skill Matrix

Game development has many different facets. Each member was required to perfect their own craft within the development process, even if they weren't particularly experienced in the area.

	Engine Programming	Script Programming	AI Programming	UI Programming	Front-End Web Development	Music Assets	Sprites and Animation	Level Design	Quality Assurance Testing	Documentation
Chi		√	√	√				√		√
Chris	√		√		√	√				
Rhea						√	√	√	√	
Kevin	√	√		√	√		√	√		√
Kasey		√	√						√	

This Quad Chart, presented early in the semester, details our original thoughts and goals for the project.

1.3 Quad Chart



2 Requirements

The following are a list of requirements we produced prior to developing the project.

2.1 System Requirements

Requirement #1
Upon launching, the game shall greet the player with a graphical title screen.

Functional:
Keyboard input shall proceed the player to the next screen.

Non-Functional:
The title screen shall have instructions for how to proceed.

Requirement #2
The game shall provide the player with an interactive menu.

Functional:
The menu shall include an option for starting a new game.
The menu shall include an option for reading the instructions.
The menu shall include an option for changing in-game options.
The menu options shall be selectable by mouse input.

Non-Functional:
The menu shall be placed in it's own UI layer.

Requirement #3
The game shall allow the player to move around a two-dimensional world.

Functional:
The map should render at no less than 20 frames per second.

Non-Functional:
The perspective should be a top down bird's-eye view.

Requirement #4
The game shall include three distinct levels.

Functional:
Levels shall be accessible from a level-select.

Non-Functional:
Each level shall be staged in a separate scene.
Level assets shall be loaded before runtime.

Requirement #5
The game shall provide collectible items throughout each level.

Functional:
The player colliding with items shall collect them.

Non-Functional:
Image assets shall be in .png format.

Requirement #6
The game shall provide the player with at least one new ability in each level.

Functional:
The player should be able to switch between abilities using keyboard controls.

Non-Functional:
Each ability shall have a cooldown time.

Requirement #7
The game shall update the player's avatar with each new ability gained.

Functional:
Player shall be able to switch between avatars once unlocked.

Non-Functional:
Each avatar shall be mapped onto a sprite-sheet.

Requirement #8
The game shall provide enemy units for the player to fight against.

Functional:
The enemies shall have the ability to move.
The enemies shall be able to attack the player.

Non-Functional:
Enemies shall not be rendered when outside player's view.

Requirement #9
The game shall provide obstacles in each level.

Functional:
Obstacles should be dependent on the player's unlocked abilities.

Non-Functional:
The obstacles shall be placed in a TMX file.

Requirement #10
The game shall include a musical score for each level.

Functional:
The tracks should loop once finished.

Non-Functional:
The audio format shall be in the .wav extension.

Requirement #11
The game shall feature a final boss fight.

Functional:
Defeating the boss will activate the credits sequence.

Non-Functional:
Boss sprite shall have no less than 2 sprite animations.

2.2 System Requirements Analysis

2.2.1 Unambiguous

Each requirement was evaluated prior to the development process, and we found no ambiguities between them. All requirements contain definite language, specifically when mentioning file formats, game features, and limits of the game running on certain technology.

2.2.2 Completeness

All requirements are complete, with both functional and non-functional portions. Each requirement covers only one topic, and is further broken down into the technical specifications, and the in-game specifications.

2.2.3 Consistency

Prior to development, all requirements were tested to not conflict with each other. With only 11 requirements, this was a relatively trivial task, as there isn't much overlap between them.

3 Project Planning and Scheduling

We organized all project tasks into the table below, which details the activity, along with what activities are prerequisites, and the anticipated time frame for completion. We kept the time frame in terms of weeks to simplify the process, as we knew it would be difficult to stick to a rigid-day schedule.

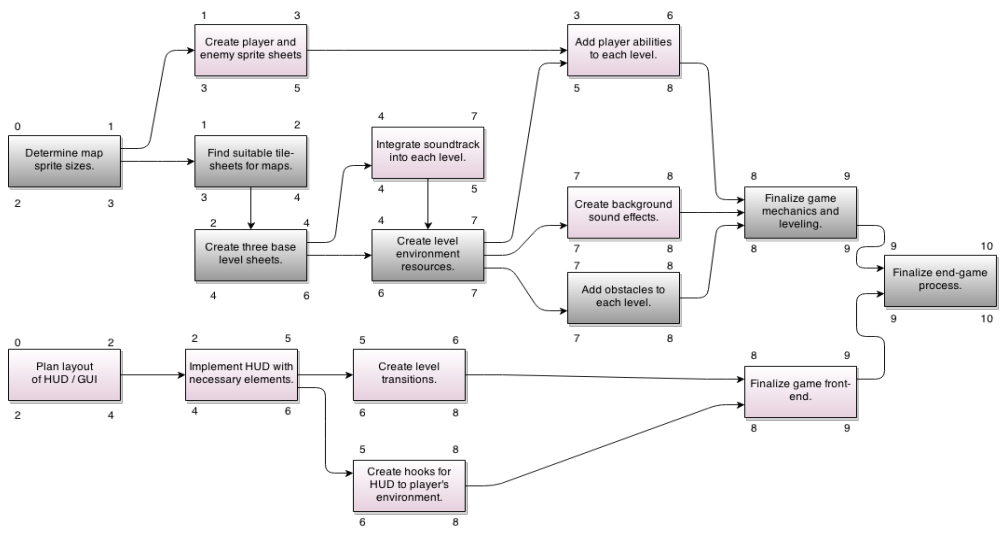
3.1 Project Tasks

Activity	Predecessor	Duration (Weeks)
(a) Determine standard map and sprite sizes.		1
(b) Plan layout of HUD / GUI		2
(c) Create Player and Enemy Spritesheets.	(a)	2
(d) Find suitable tilesheet for maps.	(a)	1
(e) Create three base level sheets.	(a)	2
(f) Implement HUD with necessary elements.	(b)	3
(g) Integrate soundtrack into each level.	(e)	3
(h) Create level environment resources.	(e)	3
(i) Create level transitions.	(f)	1

Activity	Predecessor	Duration (Weeks)
(j) Create hooks to HUD for player.	(f)	3
(k) Add player abilities to each level.	(c), (o)	3
(l) Create background sound effects.	(h)	1
(m) Add obstacles to each level.	(h)	1
(n) Finalize game mechanics and leveling.	(k), (l), (m)	1
(o) Finalize game front-end.	(j), (i)	1
(p) Finalize end-game process.	(n), (o)	1

3.2 Project Scheduling

Below is our Scheduling Diagram with Critical Path included. We were relatively successful in staying on schedule throughout the term.



Notes: Times are in terms of weeks.
Critical path is marked in grey.

4 System Implementation

Our implementation details are separated into a few different categories. Throughout this section, we detail the software resources that went into producing the game, along with general file organization, and abstract class hierarchies / design.

4.1 Implementation Details

4.1.1 System Resources

Engine: Quintus
Quintus is a minimal HTML5 game engine, that provides a simple API for common game tasks like collision detection, event handling, asset loading, and image rendering. It is still very much in its infancy.

Map Editor: Tiled
Tiled is a map editor, in the sense that it provides an easy interface to create exportable XML files, which Quintus can read natively. This allows us to quickly create game level data, with a unique enemies, tilesets, and items for each level.

Graphics Editor: Inkscape
Inkscape is a vector graphics tool we used for creating our sprites and sprite sheets. All image assets, with the exception of tilesets, were drawn by hand inside *Inkscape*.

Music Editor: Pulseboy
Pulseboy is an online music creation tool, specializing in 8-bit classic video game sound. It is very simplistic, in the sense that all music must be created completely from scratch, with only a few 8-bit sounds at your disposal. We used *Pulseboy* to create multiple 8-bit soundtracks for the game.

4.1.2 File Hierarchy

Description:

Our project is organized in the following manner:

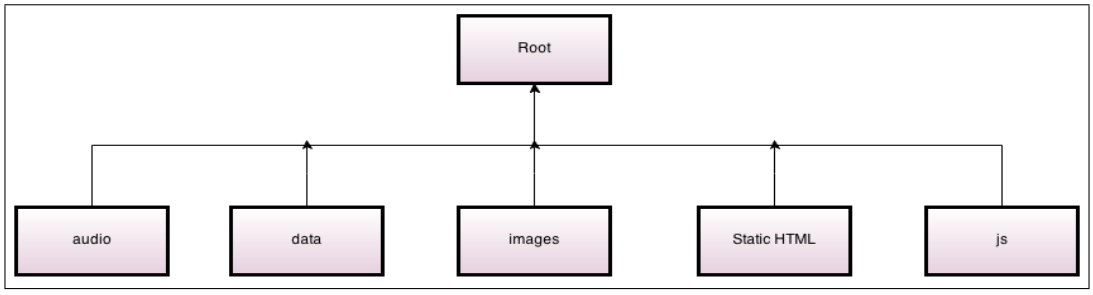
- All audio and music assets are to be loaded from the audio directory.
- All TMX (map) files are to be loaded from the data directory.
- All sprite and tileset resources are to be loaded from the images directory.
- All HTML pages for the game's frontend are served from the root of the project.
- All game code is to be placed in the js (javascript) directory.

Separating our game resources into this directory structure allows us to easily manage a large amount of assets that would otherwise be difficult to track throughout the course of the

project.

To give an idea of the scale of our asset pipeline, towards the end of development, our game consisted of the following:

- 20+ unique audio clips and music tracks.
- 13 TMX level assets, some unused due to bugs with the engine.
- 90+ image assets split between environment and entity resources.
- All Javascript files discussed in future sections.



Had this directory structure not been in place, the project would become unmaintainable, very quickly.

4.2 Code Organization

4.2.1 File Relationships

Description:

The javascript code for our game is further split into five separate modules, and function as follows.

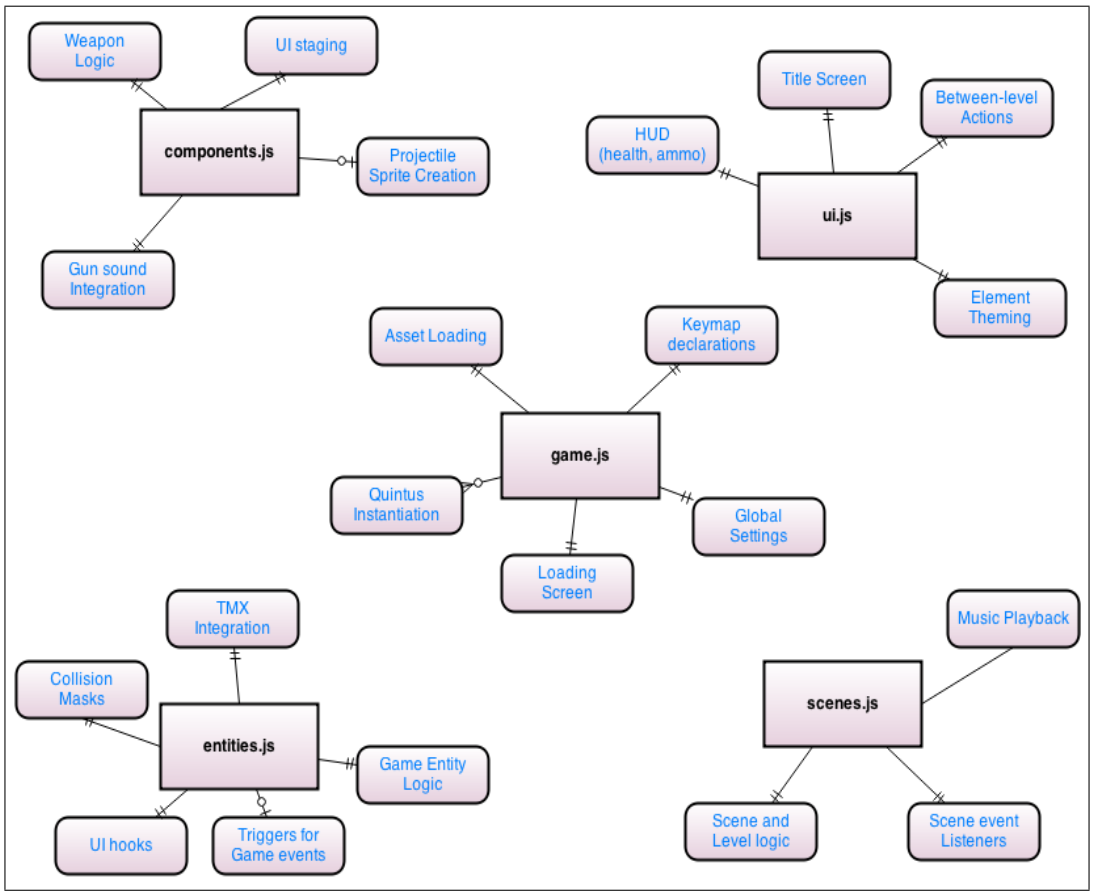
The game.js module serves as the base for setting up the game's environment. From this file, the Quintus engine is instantiated. Afterwards, all image, audio, and level assets are loaded into the engine, and the final portions of the game's setup are passed into the scenes module.

Inside the scenes.js module, the game's title screen is loaded, followed by all data needed to initiate the level loading sequence. Music playback is also handled within this file. This file also communicates closely with the ui (User Interface) module.

The ui.js module controls all interactions between the game and user. It defines a standard color theme for all interface elements of the game, and all functionality associated with said user interactions. A HUD (Heads-up Display) is defined, allowing the user several options and notifications throughout the course of the game.

The majority of the gameplay logic is contained in both the entities.js and components.js modules. The entities module contains the entire codebase for the player's interactions with their environment. From this file, all enemy, item, and ability constructs are defined. This module is largely the most critical part of the entire project.

As a convenience module, components.js defines all weapon functionality, which creates a simple process for allowing the player to obtain certain weapons throughout the course of the game. It ties closely with entities.js, as well as providing callbacks for updating the games interface through ui.js.



We feel that separating the game's logic into these distinct categories is the most logical approach to small game development. Again, it allows us to keep the project organized throughout development, so team members can work in their area of expertise, without having to venture too far into unfamiliar territory.

4.2.2 Entity Relationships

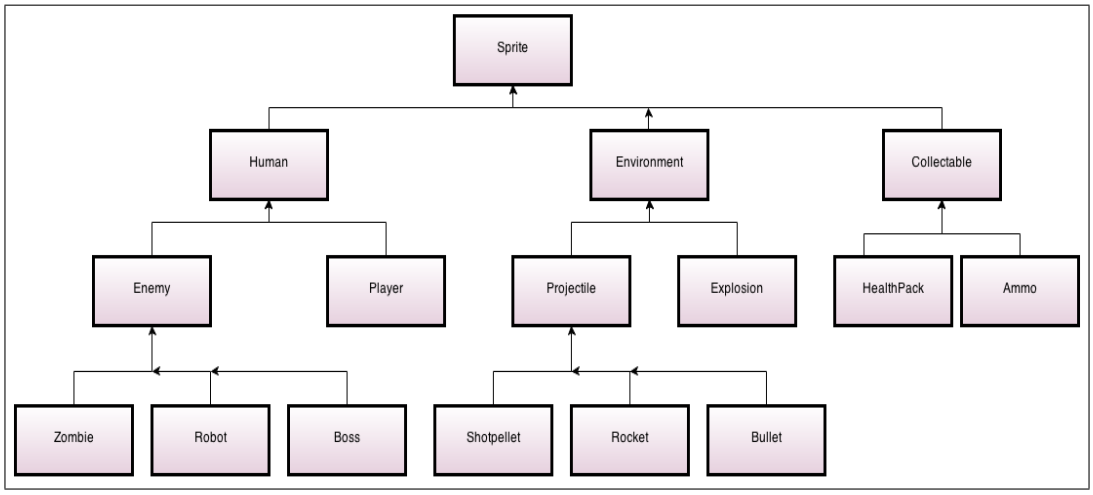
Description:

This Entity Relationship diagram describes the class structure of our entities.js file discussed in the previous section. To give a high level outline of our class code, we have initially that every entity drawn to the screen is derived from the Sprite class given by the Quintus engine. From this Sprite class, we derive three fundamental ideas for the game.

We define the notion of a Human, which represents all intelligent life in the game. This class allows us to define enemy functionality against our player, along with more specific enemies to give more diversity to each level.

We also define the game's Environment through our Sprite class, which includes non intelligent pieces of our game that nonetheless can still interact with our Human entities. For instance, all ammo and explosion objects are fundamental parts of the game's environment, and are derived from our base Environment class.

Finally, we include a simple Collectables class, which allows for simple game mechanics like increasing health or ammo based on certain Sprites that the player encounters throughout each level.



Having three main derived classes from Sprite allows for maximum code reuse and simplification, which is necessary when multiple varieties of enemies and collectables are to be a part of the game.

5 System Testing

We've split our test cases into two distinct categories: Engine tests, and Gameplay tests. The Engine tests consist of all aspects outside of the in-game content, that the player doesn't directly experience while playing the game. The Gameplay tests are much more ingrained with the overall feel of the game. Results of these tests can be found in Appendix B.

5.1 Test Cases and Procedures

5.1.1 Engine Tests

- **Testing Sprite Animation:**
  Quintus has the ability to read spritesheets, which are a collection of images of a character in different sequential poses that when flipped through, give the illusion of animation. It was simple to create the spritesheets for the player character, but implementing the animation sequence in Quintus was not working. The animations were to be for the player switching weapons.

- **Testing TMX map loading:**
  The Quintus engine has an issue with loading the top left quadrant of many TMX map files. We needed a way of testing each map to make sure each player did not encounter this broken region of the map.

- **Testing sprite-sheet asset loading:**
  When our artist drew up many of our spritesheets, the Quintus engine would create an incorrect collision-box around the sprite, due to various padding. We needed a way to test that each sprite had an appropriate hit-box.

- **Testing Browser Compatability:**
  Sometimes there are discrepancies between internet browsers that run web applications. We need to have the game run reasonably well in major browsers like Chrome, Firefox, and Safari.

- **Testing Frames per Second:**
  When a large amount of enemies are rendered on screen, we must make sure that the game doesn't drop below a playable framerate.

5.1.2 Gameplay Tests

- **Testing RPG Explosion functionality:** Initially the explosion was dealing constant damage for the entire duration that it was colliding with an object, which gave it near limitless damage potential. We needed to make sure this was not the case.

- **Testing User Controlled RPG functionality:** All weapons in the game started as simple point and shoot with no variation except for the shotgun, which has a spread. The increased damage potential of the rocket and the associated ammo cost led to the idea

of a user controlled rocket. We needed to check that RPGs correctly followed the user's actions.

- **Enemy AI:** Enemies would initially only swarm the player and incessantly follow them around the map even if they were no longer within the player's FOV. We needed to reduce this view, so that the game was not so computationally expensive.
- **Testing Bullet Collision:** Initially, the bullets would not disappear after striking enemies and could pass through to impact enemies behind the original target. Also, bullets would bounce off walls allowing for shooting around corners, though sometimes they would bounce back at the player. Collision for bullets needed to be tested.
- **Enemy Attack Speed:** Melee enemies initially ran up to the player and would attack without any pause between each blow, resulting in the player's health dropping to zero in under two seconds. This was undesirable, and the health damage needed to be reduced.
- **Testing Ammo Count:** Initially, the ammo counter would not update when the player shot. The ammo count was also off when a new level was loaded. The ammo count would change when enemies shot, which is clearly undesirable.
- **Testing Health Count:** Initially, the health count would not update when player was hit. We needed to test that, on each collision with a bullet, the player's health would drop.
- **Testing Number of Enemies:** Initially, the number of enemies label would not update. When a new level loaded, the number of enemies label was incorrect. Sometimes, the number of enemies would even be negative.
- **Testing Player Death Triggering "Game Over":** Initially, the player did not have a health stat. Once it was implemented, the game was designed to end once the player's health dropped to zero. For a while, player health would drop into negative territory, but the player remained alive and could still continue in the game.
- **Testing Ammo Count is accurate with all weapons:** Each weapon pulls from the same ammo pool. However, the cost associated with firing the shotgun, assault rifle, or RPG needed to be tested, and verified to be higher than the cost of firing a pistol.
- **Testing Enemies Not Stealing Player Ammo:** After the enemies were given the ability to shoot, it was discovered that when they ran over the ammo, it would be deposited in their ammo bank. The ammo was intended to be only for the player.

6 Summary and Conclusions

This project was a major undertaking for all of us. Starting at the beginning of the semester, and working several hours per week, we pushed over 250 commits to our repository, and just barely made the deadline.

Game development requires pulling together several different technologies, specifically ones that programmers aren't traditionally exposed to. Over the course of the project, members of our group needed to become reasonably proficient in sprite graphic design, music composition, and general level design. This is a substantial time investment, placed atop of learning how to incorporate these technologies into a completely foreign game engine.

Interestingly enough, everyone seemed to latch onto their own niche, and fill each role that needed to be filled. One of us specialized in the art assets, while others felt more comfortable with the programming and scripting portions of development. Even while programming, some group members, for instance, felt more comfortable in designing the visual interface, while others wouldn't touch those parts with a ten foot pole, and focused instead in artificial intelligence design, or weapon development. Without everyone in their own comfort zone, the project would not have made it into the state that it is.

Computer Science majors often wish to make video games for a living, without having experienced the sheer amount of work that goes into making even the simplest of games. If nothing else, the development of Four to the Fifth served as both an expansive peek into the rewarding world of game development, and at the same time, a warning of the tedium and frustration along the road to a finished product.

With that said, we'd all like to think you've enjoyed playing our game, and reading exactly what went into making it possible.

And in the end, was it worth the struggle? Without a doubt, the answer is yes, *1024* times over.

7 References

The following are website resources listed previously throughout the report.

> *http://www.pulseboy.com/*

> *http://www.inkscape.org/en/*

> *http://html5quintus.com/*

> *http://www.mapeditor.org/*

> *http://opengameart.org/*

> *http://www.soundjay.com/*

8 Appendix A

8.1 System Implementation Code

Preliminary note: All code can be easily navigated on our Github page.
https://github.com/CLaverdiere/Four-to-the-Fifth

We've only included the Javascript code for this section. Information about other resources used can be found in the System Implementation section.

8.1.1 components.js

```
Q.component("gun", {
  added: function() {
    this.entity.p.asset = this.entity.p.pistol_sprite;
    Q.audio.play("gun_cock.wav");
  },

  extend: {
    fire: function() {
      if (this.p.bullets > 0 && !this.p.fire_block){
        Q.audio.play("pistol_shot.wav");
        Q.stage().insert(new Q.Bullet(
        {
          x: this.p.x + 100 * Math.cos(TO_RAD * (this.p.angle+90)),
          y: this.p.y + 100 * Math.sin(TO_RAD * (this.p.angle+90)),
          vx: 1000 * Math.cos(TO_RAD * (this.p.angle+90)),
          vy: 1000 * Math.sin(TO_RAD * (this.p.angle+90)),
        }
        ));
        this.p.bullets -= 1;
        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
          Q.state.dec("ammo", 1);
        }
      }
    }
  },
});

Q.component("shotgun", {
  added: function() {
    this.entity.p.asset = this.entity.p.shotgun_sprite;
    Q.audio.play("gun_cock.wav"); // Should have unique shotgun load sound.
  },

  extend: {
    fire: function() {
      var pellet_nbhd = 6;

      if(SUPER_SHOTGUN) {
```

```
        pellet_nbhd = 12;
      }

      if (this.p.bullets > 0 && !this.p.fire_block){
        Q.audio.play("shotgun_shot.wav");
        for(var i=-pellet_nbhd; i < pellet_nbhd; i++){
          var spread = i*2;
          var speed = Math.random() * 1500 + 850; // Speed between 850 and 1500.
          Q.stage().insert(new Q.ShotPellet(
              {
                x: this.p.x + 100 * Math.cos(TO_RAD * (this.p.angle+90)),
                y: this.p.y + 100 * Math.sin(TO_RAD * (this.p.angle+90)),
                vx: speed * Math.cos(TO_RAD * (this.p.angle+90+spread)),
                vy: speed * Math.sin(TO_RAD * (this.p.angle+90+spread)),
              }
          ));
        }
        this.p.bullets -= 4;
        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
          Q.state.dec("ammo", 4);
        }
      }
    }
  },
});

Q.component("assaultrifle", {
  added: function() {
    this.entity.p.asset = this.entity.p.rifle_sprite;
    Q.audio.play("gun_cock.wav");
  },

  extend: {
    fire: function() {
      if (this.p.bullets > 0 && !this.p.fire_block){
        for(var i=160; i>=100; i-=30){
          Q.audio.play("assault_rifle_shot.wav");
          Q.stage().insert(new Q.Bullet(
          {
            x: this.p.x + i * Math.cos(TO_RAD * (this.p.angle+90)),
            y: this.p.y + i * Math.sin(TO_RAD * (this.p.angle+90)),
            vx: 1000 * Math.cos(TO_RAD * (this.p.angle+90)),
            vy: 1000 * Math.sin(TO_RAD * (this.p.angle+90)),
          }
          ));
        }

        this.p.bullets -= 3;
        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
          Q.state.dec("ammo", 3);
        }
```

```
      }
    }
  },
});

Q.component("machinegun", {
  added: function() {
    this.entity.p.asset = this.entity.p.mg_sprite;
    Q.audio.play("gun_cock.wav");
  },

  extend: {
    fire: function() {
      if (this.p.bullets > 0 && this.p.fire_delay <= 0){
        Q.audio.play("gun_shot.wav");
        Q.stage().insert(new Q.Bullet(
        {
          x: this.p.x + 100 * Math.cos(TO_RAD * (this.p.angle+90)),
          y: this.p.y + 100 * Math.sin(TO_RAD * (this.p.angle+90)),
          vx: 1000 * Math.cos(TO_RAD * (this.p.angle+90)),
          vy: 1000 * Math.sin(TO_RAD * (this.p.angle+90)),
        }
        ));

        this.p.bullets -= 1;
        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
          Q.state.dec("ammo", 1);
        }
      }
    }
  },
});

Q.component("rocketlauncher", {
  added: function() {
    this.entity.p.asset = this.entity.p.rocket_sprite;
    Q.audio.play("gun_cock.wav");
  },

  extend: {
    fire: function() {
      if (this.p.bullets > 0 && !this.p.fire_block){
        Q.audio.play("rocket_fire.wav");
        Q.stage().insert(new Q.Rocket(
        {
          angle: this.p.angle,
          x: this.p.x + 100 * Math.cos(TO_RAD * (this.p.angle+90)),
          y: this.p.y + 100 * Math.sin(TO_RAD * (this.p.angle+90)),
          vx: 500 * Math.cos(TO_RAD * (this.p.angle+90)),
          vy: 500 * Math.sin(TO_RAD * (this.p.angle+90)),
        }
        ));

```

```
        this.p.bullets -= 5;
        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
          Q.state.dec("ammo", 5);
        }
      }
    }
  },
});
```

8.1.2 ENTITIES.JS

```
// Create an abstract Human class
// Reduces duplicate code between Player / Enemies.
Q.Sprite.extend("Human", {
  init: function(p) {
    this._super(p, {
      asset: p.base_sprite,
      bullets: 0,
      stuck: 0,
      stuckCheck: false,
      collisionMask: Q.SPRITE_ACTIVE | Q.SPRITE_ENEMY | Q.SPRITE_DEFAULT | Q.
          SPRITE_PLAYER,
      fire_block: false,
      fire_delay: 100,
      hp: 100,
      shotDelay: 10,
      meleeDelay: 100,
      prior_sprite: p.base_sprite,
      punch_timer: 10,
      pushback: 15,
      sprinting: false,
      shot_delay_inc: 35,
      shot_delay_boss_inc: 15,
      sight_range: 300,
      stepDistance: 10,
      stepDelay: 0.01,
      punching: false,
      x: 300,
      y: 300
    });

    this.add('2d');
    this.on("hit", function(collision){
      if(collision.obj.isA("Bullet") || collision.obj.isA("ShotPellet") ||
          collision.obj.isA("Explosion")){
        if(collision.obj.isA("Bullet")) { this.p.hp -= 5; }
        else if(collision.obj.isA("ShotPellet")) { this.p.hp -= 2; }
        else if(collision.obj.isA("Explosion")) {
          if(HOMING_ROCKETS){
            this.p.hp -= 12;
          }
          else{
            this.p.hp -= 8;
```

```
          }
        }

        if(this.isA("Player")){
          Q.stageScene("ui", 1, this.p);
        }

        if(this.p.hp <= 0){
          this.destroy();

          // Reset to title if player dies.
          if(this.isA("Player")){
            Q.stage().trigger("player_death");
          } else {
            Q.stage().trigger("enemy_killed");
          }

        } else {
          // Human bounces back from being shot.
          this.p.x -= this.p.pushback * Math.cos(TO_RAD * (this.p.angle+90));
          this.p.y -= this.p.pushback * Math.sin(TO_RAD * (this.p.angle+90));
        }

        if(!collision.obj.isA("Explosion")){
          collision.obj.destroy();
        }
      }

      else if(collision.obj.isA("Player")){
        if(collision.obj.p.punching){
          this.p.hp -= 20;
          this.p.x -= 30 * Math.cos(TO_RAD * (this.p.angle+90));
          this.p.y -= 30 * Math.sin(TO_RAD * (this.p.angle+90));
          if(this.p.hp <= 0){
            this.destroy();
            // Reset to title if player dies.
            if(this.isA("Player")){
              Q.stage().trigger("player_death");
            } else {
              Q.stage().trigger("enemy_killed");
            }
          }
        }
      }

      else{ //collision with a wall
        if(!collision.obj.isA("Enemy") && !this.p.stuckCheck){ //colliding
            with wall
            this.p.stuck += 1;
          }
        if(this.p.stuck >= 24){ //game assumes stuck
          this.p.stuckCheck = true;
        }
      }
```

```
    });
  },

  equip_gun: function() {
    this.unequip_guns();
    this.add("gun");
  },

  equip_shotgun: function() {
    this.unequip_guns();
    this.add("shotgun");
  },

  equip_machinegun: function() {
    this.unequip_guns();
    this.add("machinegun");
  },

  equip_rocketlauncher: function() {
    this.unequip_guns();
    this.add("rocketlauncher");
  },

  equip_assaultrifle: function() {
    this.unequip_guns();
    this.add("assaultrifle");
  },

  // Punching event
  punch: function() {
    if(!this.p.punching){
      // Play a random punch sound. Yes, very obtuse code.
      Q.audio.play("punch" + ((Math.floor(Math.random()*10) % 2) + 1) + ".wav
          ")

      this.p.prior_sprite = this.p.asset;
      this.p.asset = this.p.punch_sprite;
      this.p.punching = true;
      this.p.punch_timer = 10;
    }
  },

  // Event to put away weapons and return to base sprite.
  put_away_wep: function() {
    this.unequip_guns();
    this.p.asset = this.p.base_sprite;
  },

  step: function(dt) {
    // Machine gun delay.
    if(this.p.fire_delay < 100){
      this.p.fire_delay += 5;
    }

```

```
    // Punching animation
    if(this.p.punching){
      this.p.punch_timer--;
      if(this.p.punch_timer < 0){
        this.p.punching = false;
        this.p.asset = this.p.prior_sprite;
      }
    }
  },
    //function to set powerups on
    powerUpFunc: function() {
      if(COOLDOWN == 0){
        Q.audio.play("health_collect.wav");
        COOLDOWN = 500;
        Q.state.set("COOLDOWN", 500);
        if(Q.state.get("level") == 1){
          HOMING_ROCKETS = true;
        }
        else if(Q.state.get("level") == 2){
          HOMING_ROCKETS = true;
          SUPER_EXPLOSIONS = true;
        }
        else{
          HOMING_ROCKETS = true;
          SUPER_EXPLOSIONS = true;
          SUPER_SHOTGUN = true;
        }
      }
    },

  // Remove all guns event.
  unequip_guns: function() {
    this.del("gun");
    this.del("shotgun");
    this.del("machinegun");
    this.del("rocketlauncher");
    this.del("assaultrifle");
  },
});

// Create Player class
Q.Human.extend("Player", {
  init: function(p) {
    this._super(p, {
      collisionMask: Q.SPRITE_ACTIVE | Q.SPRITE_ENEMY | Q.SPRITE_DEFAULT,
      type: Q.SPRITE_PLAYER,
    });

    this.add('stepControls');
    this.on("step", this, "step_player");
    this.equip_gun();

    Q.input.on("fire", this, function(){ this.fire()

```

```
    });
    Q.input.on("wep1", this, "put_away_wep");
    Q.input.on("wep2", this, "equip_gun");
    Q.input.on("wep3", this, "equip_shotgun");
    Q.input.on("wep4", this, "equip_machinegun");
    Q.input.on("wep5", this, "equip_rocketlauncher");
    Q.input.on("wep6", this, "equip_assaultrifle");
    Q.input.on("powerUp", this, "powerUpFunc");
    Q.input.on("punch", this, "punch");
    Q.input.on("pause", this, function(){
      Q.state.inc("pause", this, !Q.state.get("pause"));
    });
  },

  step_player: function(dt) {

    // Update player angle based on mouse position.
    if (!this.p.punching){

      // We keep track of the previous mouse coordinates each step.
      // We then only update the angle when the mouse coords have changed.
      if( prev_mouse_coords[0] != Q.inputs['mouseX'] || prev_mouse_coords[1] != Q.inputs['mouseY'] ){
        var dmx = Q.inputs['mouseX'] - this.p.x;
        var dmy = Q.inputs['mouseY'] - this.p.y;

        prev_mouse_coords = [Q.inputs['mouseX'], Q.inputs['mouseY']];
        this.p.angle = -1 * TO_DEG * Math.atan2(dmx, dmy);
      }
    }

    // When pressing the 'forward' key, the human follows their orientation.
    if(Q.inputs['forward']){
      this.p.x += (this.p.stepDistance) * Math.cos(TO_RAD * (this.p.angle+90));
      this.p.y += (this.p.stepDistance) * Math.sin(TO_RAD * (this.p.angle+90));
    }

    // Create a block on firing so we don't shoot repeatedly when button held down.
    // Maybe make an exception for automatic guns, if ever added.
    if(Q.inputs['fire']){
      this.p.fire_block = true;
      if(this.p.fire_delay > 0){
        this.p.fire_delay -= 20;
      }
    } else {
      this.p.fire_block = false;
    }

    // Sprint input activation and deactivation.
    if(Q.inputs['sprint']){
      if(!this.p.sprinting){
```

```
        this.p.sprinting = true;
        this.p.stepDistance *= 1.5;
      }
    } else {
      if(this.p.sprinting){
        this.p.sprinting = false;
        this.p.stepDistance /= 1.5;
      }
    }

    // Send event to all enemies to look at and chase the player.
    var enemies = Q("Enemy");
    var boss = Q("Boss");
    enemies.trigger("chase_player", this);
    boss.trigger("kill_player", this);

    if(COOLDOWN > 0){
      if(COOLDOWN == 100){
        SUPER_EXPLOSIONS = false;
        HOMING_ROCKETS = false;
        SUPER_SHOTGUN = false;
      }
      COOLDOWN--;
      Q.state.dec("COOLDOWN", 1);
    }
  }
});


Q.Human.extend("Enemy", {
  init: function(p) {
    this._super(p, {
      boss_ai: false,
      collisionMask: Q.SPRITE_ACTIVE | Q.SPRITE_PLAYER | Q.SPRITE_ENEMY | Q.
          SPRITE_DEFAULT,
      scale: 1,
      speed: 1,
      type: Q.SPRITE_ENEMY,
    });

    this.add("gun");
    this.on("chase_player");
    this.on("face_player");
    this.on("frenzy");
    if(this.p.boss_ai) {
      this.on("step", this, "step_boss");
    }
  },

  chase_player: function(player){
    this.face_player(player);

    // Stop, and shoot at player if they get close.
    // We use a shotDelay to make sure the enemies only
```

```
    //   shoot every so often. Yes, slightly redundant as we already
    //   have fire_delay as well. Should refactor.
    if(Math.abs(this.p.x - player.p.x) < this.p.sight_range && Math.abs(this.
        p.y - player.p.y) < this.p.sight_range){
      if(!this.p.boss_ai) {
        if(this.p.shotDelay-- <= 1){
          this.fire();
          this.p.shotDelay += this.p.shot_delay_inc;
        }
      } else {
        if(this.p.shotDelay-- <= 1){
          this.p.fire_delay = 0;
          this.fire();
          this.p.shotDelay += this.p.shot_delay_boss_inc;
        }
      }
    }

    else if(Math.abs(this.p.x - player.p.x) > this.p.sight_range * 1.5 &&
        Math.abs(this.p.y - player.p.y) > this.p.sight_range * 1.5){
      //sight range ends here
    }

    else {
      if(this.p.stuckCheck){ //stuck so move back 4 times
        this.p.stuck -= 6;
        this.p.x -= this.p.speed * Math.cos(TO_RAD * (this.p.angle+90));
        this.p.y -= this.p.speed * Math.sin(TO_RAD * (this.p.angle+90));

        if(this.p.stuck <= 0){
          this.p.stuckCheck = false; //reset check after stuck is 0
        }
      }
      else{
        // Chase player if out of range.
        this.p.x += this.p.speed * Math.cos(TO_RAD * (this.p.angle+90));
        this.p.y += this.p.speed * Math.sin(TO_RAD * (this.p.angle+90));
      }
    }
  },

  face_player: function(player){
    this.p.angle = -1 * TO_DEG * Math.atan2( (player.p.x - this.p.x), (player
        .p.y - this.p.y) );
  },

  frenzy: function(player){
    this.p.speed *= 1.5;
  },

  step_boss: function(){
    this.p.pushback = 5;
    if(this.p.hp < .40 * this.p.max_hp){
      // Decide on final form.
```

```
      if(!this.has("machinegun")){
        this.equip_machinegun();
        this.p.shot_delay_boss_inc = 0;
        this.p.speed *= .8;
      }
    }
    else if(this.p.hp < .60 * this.p.max_hp){
      if(!this.has("shotgun")){
        this.equip_shotgun();
        this.p.speed *= 1.2;
      }
    }
    else if(this.p.hp < .85 * this.p.max_hp){
      if(!this.has("assaultrifle")){
        this.equip_assaultrifle();
        this.p.speed *= 1.1;
      }
    }
    else {
      // Stay on pistol.
    }
  },

});

Q.Enemy.extend("Zombie", {
  init: function(p) {
    this._super(p, {
      collisionMask: Q.SPRITE_ACTIVE | Q.SPRITE_PLAYER | Q.SPRITE_ENEMY | Q.
          SPRITE_DEFAULT | Q.SPRITE_ZOMBIE,
      type: Q.SPRITE_ZOMBIE
    });

    this.on("chase_player");
    this.on("maul_player");
  },

  chase_player: function(player){
    this.face_player(player);
    this.p.x += this.p.speed * Math.cos(TO_RAD * (this.p.angle+90));
    this.p.y += this.p.speed * Math.sin(TO_RAD * (this.p.angle+90));
  },

  maul_player: function(collision){
    if(collision.obj.isA("Player")){
      collision.obj.p.hp -=7;
      collision.obj.p.x -= 15 * Math.cos(TO_RAD * (this.p.angle+90));
      collision.obj.p.y -= 15 * Math.sin(TO_RAD * (this.p.angle+90));
      this.p.speed *= 0.9;
    }
  },
});
```

```
// Should make this more generic, extendable for more ammo types, obviously.
Q.Sprite.extend("Ammo", {
  init: function(p) {
    this._super(p, {
      asset: "ammo_clip.png",
      collisionMask: Q.SPRITE_PLAYER,
      capacity: 15,
    });

    this.add('2d');

    this.on("hit", function(collision){
      if(collision.obj.isA("Player")){
        // ammo collected.
        Q.audio.play("gun_cock.wav");
        this.destroy();
        collision.obj.p.bullets += this.p.capacity;
        Q.state.inc("ammo", this.p.capacity);
        Q.stageScene("ui", 1, collision.obj.p);
      }
    });
  }
});

Q.Sprite.extend("HealthPack", {
    init: function(p) {
      this._super(p, {
        asset: "health_pack.png",
        collisionMask: Q.SPRITE_PLAYER,
        capacity: 35,
      });

      this.add('2d');

      this.on("hit", function(collision){
        if(collision.obj.isA("Player")){
          Q.audio.play("health_collect.wav");
          this.destroy();
          collision.obj.p.hp += this.p.capacity;
          Q.state.inc("player_health", this.p.capacity);
          Q.stageScene("ui", 1, collision.obj.p);
        }
      });
    }
  });

Q.Sprite.extend("Bullet", {
  init: function(p) {
    this._super(p, {
      asset: "bullet.png",
      collisionMask: Q.SPRITE_ENEMY | Q.SPRITE_ACTIVE | Q.SPRITE_DEFAULT,
      type: Q.SPRITE_POWERUP,
    });

```

```
    this.add('2d');

    this.on("hit", function(collision){
      this.destroy();
    });
  }
});

Q.Sprite.extend("ShotPellet", {
  init: function(p) {
    this._super(p, {
      asset: "shot_pellet.png",
      collisionMask: Q.SPRITE_ENEMY | Q.SPRITE_ACTIVE | Q.SPRITE_DEFAULT,
      type: Q.SPRITE_POWERUP,
    });

    this.add('2d');

    this.on("hit", function(collision){
      this.destroy();
    });
  }
});

Q.Sprite.extend("Rocket", {
  init: function(p) {
    this._super(p, {
      asset: "rocket.png",
      atk_type: "projectile",
      collided: false,
      rebounded: false,
      speed: 1,
      collisionMask: Q.SPRITE_ENEMY | Q.SPRITE_ACTIVE | Q.SPRITE_DEFAULT,
      type: Q.SPRITE_POWERUP,
    });

    this.add('2d');

    this.on("hit", function(collision){
      if(collision.obj.p.boss_ai){
        console.log("Rebounding rocket.");
        if(!this.p.rebounded){
          this.p.rebounded = true;
          this.p.angle -= 180;
        }
      }
      else {
        if(!this.collided){
          Q.audio.play("rocket_explode.wav");
          if(SUPER_EXPLOSIONS){
            Q.stage().insert(new Q.SuperExplosion(
              {
                x: this.p.x,
                y: this.p.y,
```

```
              }
            ));
          }
          else{
            Q.stage().insert(new Q.Explosion(
              {
                x: this.p.x,
                y: this.p.y,
              }
            ));
          }

          this.collided = true;
        }
        this.destroy();
      }
    });
  },

  step: function(dt) {
    if(!this.p.rebounded && HOMING_ROCKETS) {
      this.p.angle = Q("Player").first().p.angle;
    }
    this.p.vx = this.p.speed * 500 * Math.cos(TO_RAD * (this.p.angle+90));
    this.p.vy = this.p.speed * 500 * Math.sin(TO_RAD * (this.p.angle+90));
    this.p.speed *= 1.05;
    if(this.p.speed > 100) {
      this.destroy();
    }
  },
});


Q.Sprite.extend("Explosion", {
  init: function(p) {
    this._super(p, {
      asset: "explosion.png",
      angle: 0,
      duration: 30,
      atk_type: "melee",
      collisionMask: Q.SPRITE_ENEMY | Q.SPRITE_ACTIVE,
      scale: .1,
      type: Q.SPRITE_POWERUP,
    });

    this.add('2d');
  },

  step: function(dt) {
    // Add logarithmic growth function to explosion.
    this.p.scale = Math.log(30 - this.p.duration) / 2;

    // Spin explosion as it goes off.
    this.p.angle += 25
```

```
    if(--this.p.duration <= 0){
      this.destroy();
    }
  },
});

Q.Sprite.extend("PowerUp", {
  init: function(p) {
    this._super(p, {
      asset: p.base_sprite,
      collisionMask: Q.SPRITE_PLAYER,
    });
  }
});

Q.Sprite.extend("SuperExplosion", {
  init: function(p) {
    this._super(p, {
      asset: "explosion.png",
      angle: 0,
      duration: 30,
      creation: 3, //can create 3 more normal explosions
      atk_type: "melee",
      collisionMask: Q.SPRITE_ENEMY | Q.SPRITE_ACTIVE| Q.SPRITE_DEFAULT,
      scale: .1,
      type: Q.SPRITE_POWERUP,
    });

    this.add('2d');
    this.on("hit", function(collision){
      if(this.p.creation > 0){
        Q.stage().insert(new Q.Explosion(
          {
            x: this.p.x,
            y: this.p.y,
          }
        ));
      }
      this.p.creation--;
    });
  },

  step: function(dt) {
    // Add logarithmic growth function to explosion.
    this.p.scale = Math.log(30 - this.p.duration) / 2;

    // Spin explosion as it goes off.
    this.p.angle += 25

    if(--this.p.duration <= 0){
      this.destroy();
    }
  },
```

```
});
```

8.1.3 GAME.JS

```
// Setup Quintus instance
var Q = Quintus({ development: true, audioSupported: [ 'wav' ] })
          .include("Sprites, Scenes, Input, 2D, Audio, Anim, Touch, UI, TMX")
          .enableSound()
          .setup({ maximize:true })
          .touch();

// GAME SETTINGS
var MUSIC_ENABLED = true;

// ABILITY SETTINGS
var COOLDOWN = 0;
var SUPER_EXPLOSIONS = false;
var HOMING_ROCKETS = false;
var SUPER_SHOTGUN = false;

// Increment this if you add a new map in pattern
// ie: (level1.tmx, level2.tmx, ..., levelN.tmx).
var NUM_MAPS = 4
var BOSS_TEXT_PAGE = 1
var POWER_UP = ""

// USEFUL GLOBALS
// Keep track of change in mouse coords.
var prev_mouse_coords = [0, 0];

// All music tracks.
var tracks = [ "disp_heroes.wav", "test.wav", "metal.wav" ];

// Global next track control
var play_next_track = function() {
    console.log("Playing next track");
    Q.state.set("track_playing", true);
    if(MUSIC_ENABLED){
      Q.audio.stop();
      Q.state.inc("track_id", 1);
      if(Q.state.get("track_id") >= tracks.length) Q.state.set("track_id", 0);
      Q.audio.play(tracks[Q.state.get("track_id")], { loop: true });
    }
}

// Turn off gravity, the game is top down.
Q.gravityX = 0;
Q.gravityY = 0;

// Define custom key mappings
Q.KEY_NAMES.Q = 81;
Q.KEY_NAMES.E = 69;
Q.KEY_NAMES.W = 87;
```

```
Q.KEY_NAMES.A = 65;
Q.KEY_NAMES.S = 83;
Q.KEY_NAMES.D = 68;
Q.KEY_NAMES.E = 69;
Q.KEY_NAMES.F = 70;
Q.KEY_NAMES.SHIFT = 16;
Q.KEY_NAMES.ONE   = 49;
Q.KEY_NAMES.TWO   = 50;
Q.KEY_NAMES.THREE = 51;
Q.KEY_NAMES.FOUR  = 52;
Q.KEY_NAMES.FIVE  = 53;
Q.KEY_NAMES.SIX   = 54;
Q.KEY_NAMES.SEVEN = 55;
Q.KEY_NAMES.BACKSPACE = 8;

// Some useful constants for speeding things up.
var TO_RAD = Math.PI / 180
var TO_DEG = 180 / Math.PI

// Key actions
Q.input.keyboardControls({
  UP:     'up',     W: 'up',
  LEFT:   'left',   A: 'left',
  DOWN:   'down',   S: 'down',
  RIGHT:  'right',  D: 'right',
  SPACE:  'fire',
  SHIFT:  'sprint', Q: 'powerUp',
  E:      'forward',
  F:      'punch',
  ONE:    'wep1',
  TWO:    'wep2',
  THREE:  'wep3',
  FOUR:   'wep4',
  FIVE:   'wep5',
  SIX:    'wep6',
  SEVEN:  'wep7',
  BACKSPACE:  'pause',
});

Q.input.mouseControls({ cursor: "on" });

// Set initial game state.
Q.state.set({ killed: 0,
              alive: 0,
              player_health: 100,
              ammo: 50,
              level: 1,
              paused: false,
              track_id: 0,
              track_playing: false,
              pause: 0,
              COOLDOWN: 0,
              SUPER_EXPLOSIONS: false,
              HOMING_ROCKETS: false,
```

```
                SUPER_SHOTGUN: false,
});


// Load other resources
Q.loadTMX([
          "start_level.tmx",

          "ammo_clip.png",
          "boss_base.png",
          "boss_pistol.png",
          "boss_punch.png",
          "boss_gatling.png",
          "boss_assault_rifle.png",
          "boss_rocket.png",
          "boss_shotgun.png",
          "bullet.png",
          "cave.png",
          "explosion.png",
          "enemy.png",
          "health_pack.png",
          "robot_dual.png",
          "robot_gatling.png",
          "rocket.png",
          "shot_pellet.png",
          "soldier_base.png",
          "soldier_pistol.png",
          "soldier_punch.png",
          "soldier_gatling.png",
          "soldier_assault_rifle.png",
          "soldier_rocket.png",
          "soldier_shotgun.png",
          "tough_guy.png",
          "zombie1.png",
          "zombie2.png",

          "between_levels.wav",
          "boss_fight.wav",
          "disp_heroes.wav",
          "game_over.wav",
          "metal.wav",
          "test.wav",
          "victory.wav",

          "assault_rifle_shot.wav",
          "gun_cock.wav",
          "gun_shot.wav",
          "health_collect.wav",
          "minigun_shot.wav",
          "pistol_shot.wav",
          "punch1.wav",
          "punch2.wav",
          "rocket_fire.wav",
          "rocket_fire2.wav",
```

```
        "rocket_explode.wav",
        "rocket_explode2.wav",
        "shotgun_shot.wav",
        "shotgun_shot2.wav",
        ], function() {
    console.log("Done loading assets.");
    Q.stageScene("start_level", 0);
    Q.stageScene("title", 1);
}, {
  progressCallback: function(loaded, total) {
    // Load map resources
    for(var i=1; i<=NUM_MAPS; i++){
      Q.loadTMX("level" + i + ".tmx");
    }

    var ld = document.getElementById("loading");
    var ls = document.getElementById("loading_status");
    ls.innerHTML = "Now Loading... " + Math.floor(loaded / total * 100) + "%"
        ;
    if(loaded == total){
      ld.remove();
    }
  }
});
```

8.1.4 SCENES.JS

```
// Create player scene
Q.scene("level", function(stage) {
  var fmod = 4;
  var frenzied_enemies = false;

  Q.stageTMX("level" + Q.state.get("level") + ".tmx", stage);
  stage.add("viewport").follow(Q("Player").first());

  // Initialize enemy amount
  Q.state.set("alive", Q("Enemy").length);

    if(Q.state.get("level") == 4){
      Q.audio.stop();
      Q.audio.play("boss_fight.wav", { loop: true });
    } else {
      play_next_track();
    }

  // pause game
  stage.on("pause_game", function(){
    if(!Q.state.get("paused")) {
      Q.state.set("paused", true);
      Q.pauseGame();
      Q.audio.stop();
    }
    else {
      Q.state.set("paused", false);
```

```
      Q.unpauseGame();

      if(Q.state.get("track_playing")){
        Q.audio.play(tracks[Q.state.get("track_id")], { loop: true });
      }
    };
  });

  // Handle event for when an enemy is killed.
  stage.on("enemy_killed", function(){
    Q.state.inc("killed", 1);
    Q.state.dec("alive", 1);
    frenzied_enemies = false;

    // Every few enemies killed, let's trigger a frenzy.
    if(!frenzied_enemies && Q.state.get("killed") % fmod === 0){
      Q("Enemy").trigger("frenzy");
      fmod *= 2;
      frenzied_enemies = true;
    }

    // Check if game over.
    if(Q("Enemy").length <= 1){
      console.log("Level beaten. Staging Next level.");
      stage.trigger("beat_level");
    }
  });

  // Handle event for when player finishes a level.
  stage.on("beat_level", function() {

      // If there's still a level after, proceed to the next level.
      if(Q.state.get("level") < NUM_MAPS){
        Q.state.inc("level", 1);
        Q.stageScene("story_scene", 0);
        Q.stageScene("null", 1);
        Q.stageScene("null", 2);
      } else {
        console.log("Game beaten.");
        Q.stageScene("endgame", 0);
        Q.stageScene("null", 1);
        Q.stageScene("null", 2);
      }
  });

  stage.on("player_death", function() {
    if(MUSIC_ENABLED){
      Q.audio.stop();
      Q.audio.play("game_over.wav", { loop: true });
    }
    Q.stageScene("title", 1);
    Q.stageScene("null", 2);
  });

```

```
});

// ==========================================

Q.scene("story_scene", function(stage) {

  if(MUSIC_ENABLED){
    Q.audio.stop();
    Q.audio.play("between_levels.wav", { loop: true });
  }

  var story_text = "";

  if (Q.state.get("level") == 1) {
    story_text = "Introduction\n\n"
      + "You find yourself in a lab full of zombies. What's going on?";
  } else if (Q.state.get("level") == 2) {
    story_text = "The Loading Bay\n\n"
      + "You make your way to the lab basement. How do I get out of here?";
  } else if (Q.state.get("level") == 3) {
    story_text = "The Sub-Basement\n\n"
      + "Finally, out of that lab. A cave! I'm almost out!";
  } else if (Q.state.get("level") == 4) {
    story_text = "The Final Boss\n\n"
      + "Ah, a cave exit! Who's blocking the door? HE'S GOT A GUN!";
  }

  // Container for text
  var story_text_cont = stage.insert(new Q.UI.FttFContainer({
    label: story_text,
    x: Q.width/2,
    y: Q.height/2,
  }));

  var story_label = stage.insert(new Q.UI.FttFText({
    label: story_text,
    y: -Q.height/4,
  }), story_text_cont);

  // Button to restart Game.
  var play_next_btn = stage.insert(new Q.UI.Button({
    border: 2,
    fill: FG_COL,
    label: "Start Level",
    radius: 3,
  }, function() {
    // If there's still a level after, proceed to the next level.
    if(Q.state.get("level") <= NUM_MAPS){
      COOLDOWN = 0;
      Q.stageScene("level", 0);
      Q.stageScene("ui", 1, Q('Player').first().p);
      Q.stageScene("menu", 2);
    } else { // Otherwise, we've beaten the game.
      console.log("Game beaten.");
```

```
      Q.stageScene("endgame", 0);
      Q.stageScene("null", 1);
      Q.stageScene("null", 2);
    }
  }), story_text_cont);

});

// ==========================================

// The ending screen.
Q.scene("endgame", function(stage) {
  if(MUSIC_ENABLED){
    Q.audio.stop();
    Q.audio.play("victory.wav", { loop: true });
  }

  // Victory text.
  var victory_label = stage.insert(new Q.UI.FttFText({
    label: "Congrats, you've saved the world from all those killer zombies.",
    x: Q.width/2,
    y: Q.height/4,
  }));

  // Button to restart Game.
  var restart_btn = stage.insert(new Q.UI.Button({
    border: 2,
    fill: FG_COL,
    label: "Play Again?",
    radius: 3,
    x: Q.width/2,
    y: Q.height/2,
  }, function() {
    Q.state.set("level", 1);
    Q.state.set("ammo", 50);
    Q.state.set("player_health", 100);
    Q.stageScene("level", 0);
    Q.stageScene("ui", 1);
    Q.stageScene("menu", 2);
  }));
});

// Create player scene
Q.scene("start_level", function(stage) {
  var fmod = 4;
  var frenzied_enemies = false;

  Q.stageTMX("start_level.tmx", stage);
  stage.add("viewport").follow(Q("Enemy").first());

  // Initialize enemy amount
  Q.state.set("alive", Q("Enemy").length);

  play_next_track();
```

```
});
```

8.1.5 UI.JS

```
// UI color theme
var FG_COL = "#0099ff";
var BG_COL = "#202020";

// Base theme class for game text.
Q.UI.FttFText = Q.UI.Text.extend("UI.FttFText", {
  init: function(p) {
    this._super(p, {
      label: "Insert Text",
      family: "nintendo_nes_font",
      color: FG_COL,
    });
  }
});


// Base theme class for containers.
Q.UI.FttFContainer = Q.UI.Container.extend("UI.FttFContainer", {
  init: function(p) {
    this._super(p, {
      border: 2,
      color: FG_COL,
      fill: BG_COL,
      opacity: .7,
      radius: 3,
    });
  }
});


// Base theme class for buttons.
// I cannot get this to work...
Q.UI.FttFButton = Q.UI.Button.extend("UI.FttFButton", {
  init: function(p, callback) {
    this._super(p, callback, {
      border: 2,
      fill: FG_COL,
      label: "Insert Text",
      radius: 3,
    });
  }
});


Q.scene("menu", function(stage){
  // options container
  var options_cont = stage.insert(new Q.UI.FttFContainer({
    w: 200,
    h: 60,
    x: Q.width - 150,
```

```
    y: 10,
    hidden: true,
  }));

  Q.state.on("change.pause", function(){
    options_cont.p.hidden = !(options_cont.p.hidden);
  });

  // Next level button
  var next_lvl_btn = stage.insert(new Q.UI.Button({
    border: 1,
    w: 200,
    h: 30,
    x: 0,
    y: 80,
    label: "Next Level",
  }, function() {
    options_cont.p.hidden = !(options_cont.p.hidden);
    Q.stage(0).trigger("beat_level");
    //start_cont.p.hidden = !(start_cont.p.hidden);
  }),options_cont);

  // Change game zoom
  var zoom_toggle = stage.insert(new Q.UI.Button({
    border: 1,
    w: 200,
    h: 30,
    x: 0,
    y: 120,
    label: "Toggle zoom level"
  }, function(){
    var zoom = Q.stage(0).viewport.scale;
    if(zoom > 3) {
      zoom = .5;
    } else {
      zoom *= 1.5;
    }
    Q.stage(0).viewport.scale = zoom;
  }), options_cont);

  // Options button
  var options_btn = stage.insert(new Q.UI.Button({
    border: 2,
    fill: FG_COL,
    label: "Controls",
    color: FG_COL,
    radius: 3,
    w: 140,
    h: 30,
    x: Q.width - 150,
    y: 40,
  }, function() {
    options_cont.p.hidden = !(options_cont.p.hidden);
  }));
```

```
105
106   // Toggle music on or off option.
107   var music_toggle = stage.insert(new Q.UI.Button({
108     border: 1,
109     w: 200,
110     h: 30,
111     x: 0,
112     y: 160,
113     label: "Music on/off"
114   }, function(){
115     if(Q.state.get("track_playing")){
116       Q.audio.stop();
117       Q.state.set("track_playing", false);
118     } else{
119       Q.audio.stop();
120       Q.audio.play(tracks[Q.state.get("track_id")], { loop: true });
121       Q.state.set("track_playing", true);
122     }
123   }), options_cont);
124
125   // Switch music track
126     var music_track = stage.insert(new Q.UI.Button({
127       border: 1,
128       w: 200,
129       h: 30,
130       x: 0,
131       y: 200,
132       label: "Next Music Track"
133     }, function(){
134       Q.audio.stop();
135       Q.state.inc("track_id", 1);
136       if(Q.state.get("track_id") >= tracks.length){
137         Q.state.set("track_id", 0);
138       }
139       Q.state.set("track_playing", true);
140       Q.audio.play(tracks[Q.state.get("track_id")], { loop: true });
141     }), options_cont);
142
143   options_cont.fit(30,20);
144
145 });
146
147
148 Q.scene("ui", function(stage){
149
150   // Power up Container
151   var power_up_cont = stage.insert(new Q.UI.FttFContainer({
152     w: 200,
153     h: 60,
154     x: 200,
155     y: 40,
156   }));
157
158   if (Q.state.get("level") == 1) {
```

```
    POWER_UP = "Homing Rockets";
  } else if(Q.state.get("level") == 2) {
    POWER_UP = "Super Explosion";
  } else {
    POWER_UP = "Super Guns";
  };

  // Power up label
  var power_up_label = stage.insert(new Q.UI.FttFText({
    size: 40,
    label: POWER_UP,
  }), power_up_cont);

  // Update power up label.
  Q.state.on("change.COOLDOWN", function(){
    if (Q.state.get("COOLDOWN") > 100) {
      power_up_label.p.label = POWER_UP;
      power_up_label.p.color = "#ff0000";
    }
    else {
      power_up_label.p.color = FG_COL;
    }
  });


  // Weapon container
  var weapon_cont = stage.insert(new Q.UI.FttFContainer({
    w: 200,
    h: 60,
    x: Q.width - 150,
    y: Q.height - 40,
  }));

  // Total ammo label
  var ammo_label = stage.insert(new Q.UI.FttFText({
    size: 40,
    label: "Ammo: " + stage.options.bullets,
  }), weapon_cont);

  // Update ammo label.
  Q.state.on("change.ammo", function(){
    ammo_label.p.label = "Ammo: " + (stage.options.bullets > 0 ? stage.
        options.bullets : 0);
  });

  // Info container
  var info_cont = stage.insert(new Q.UI.FttFContainer({
    w: 200,
    h: 60,
    x: 150,
    y: Q.height - 40,
  }));

  // Health container
```

```
212  var health_cont = stage.insert(new Q.UI.FttFContainer({
213    // ...
214  }), info_cont);
215
216  // Health label
217  var health_label = stage.insert(new Q.UI.FttFText({
218    color: "#f00",
219    size: 40,
220    label: "Health: " + stage.options.hp,
221  }), info_cont);
222
223  // Update player_health label event.
224  Q.state.on("change.player_health", function(){
225    health_label.p.label = "Health: " + stage.options.hp
226  });
227
228  //level container
229  var level_cont = stage.insert(new Q.UI.FttFContainer({
230    w: 400,
231    h: 60,
232    x: Q.width/2,
233    y: Q.height - 40,
234  }));
235
236  //level label
237  var level_label = stage.insert(new Q.UI.FttFText({
238    color: "#fff",
239    size: 40,
240    x: -150,
241    label: "lvl: " + Q.state.get("level"),
242  }), level_cont);
243
244  // Total enemys left label
245  var enemy_left_label = stage.insert(new Q.UI.Text({
246    color: "#fff",
247    size: 40,
248    x: 75,
249    label: "Enemies: " + Q.state.get("alive"),
250  }), level_cont);
251
252  // Update number enemys left label.
253  Q.state.on("change.alive", function(){
254    enemy_left_label.p.label = "Enemies: " + Q.state.get("alive")
255  });
256
257  // Update level label event.
258  Q.state.on("change.level", function(){
259    level_label.p.label = "lvl: " + Q.state.get("level")
260  });
261
262  power_up_cont.fit(20,20);
263  level_cont.fit(20,20);
264  weapon_cont.fit(20,50);
265  info_cont.fit(20,50);
```

```
  health_cont.fit(5,5);
});


// The initial title screen.
Q.scene("title", function(stage) {
  //var title = document.getElementById("start_title");

  // Title container
  var title_cont = stage.insert(new Q.UI.FttFContainer({
    x: Q.width/2,
    y: Q.height/8,
  }));

  // Title label
  var start_title_label = stage.insert(new Q.UI.FttFText({
    label: "Four-To-The-Fifth",
    size: 80,
    y: 100,
  }), title_cont);

  // Container for start up
  var start_cont = stage.insert(new Q.UI.FttFContainer({
    x: Q.width/2,
    y: Q.height/2,
  }));

  // Button to Start Game.
  var start_btn = stage.insert(new Q.UI.FttFButton({
    border: 2,
    fill: FG_COL,
    label: "Start Game",
  }, function() {
    Q.audio.stop();
    Q.stageScene("story_scene", 0);
    Q.stageScene("null", 1);
  }), start_cont);

  // Button to show options.
  var start_options_btn = stage.insert(new Q.UI.Button({
    border: 2,
    fill: FG_COL,
    label: "Options",
    color: FG_COL,
    radius: 3,
    y: 50,
  }, function() {
    start_options_cont.p.hidden = !(start_options_cont.p.hidden);
    start_cont.p.hidden = !(start_cont.p.hidden);
  }), start_cont);

  // Container for start options
  var start_options_cont = stage.insert(new Q.UI.FttFContainer({
    hidden: true,
```

```
    x: Q.width/2,
    y: Q.height/2,
  }));

  // Controls Header label
  var start_controls_label = stage.insert(new Q.UI.FttFText({
    label: "Controls",
    y: -80,
  }), start_options_cont);

  // Controls label
  var controls_label = stage.insert(new Q.UI.FttFText({
    label: "Movement: WASD or E \nSwitch Weapon: NUMKEYS \nFire weapon: SPACE
         \nPunch: F \nAbility: Q \nPause: BACKSPACE",
    size: 16
  }), start_options_cont);

  // Button to go back to start menu.
  var return_to_start_btn = stage.insert(new Q.UI.Button({
    border: 2,
    fill: FG_COL,
    label: "Back",
    y: 100,
  }, function() {
    start_options_cont.p.hidden = !(start_options_cont.p.hidden);
    start_cont.p.hidden = !(start_cont.p.hidden);
  }), start_options_cont);

  title_cont.fit(100, 400);
  start_cont.fit(50, 75);
  start_options_cont.fit(10, 10);
});
```

9 Appendix B

9.1 Test Results

These are the results of the tests from the *Test Cases and Procedures* section.

9.1.1 Engine Test Results

- **Testing Sprite Animation:** This test partially passed. We were not able to directly load our animations into the engine, but instead gave the illusion of animation through updating the player sprite smoothly to each new weapon sprite.
- **Testing TMX map loading:** The test for this case passed, with every map in our game simply blocking off the region that was broken. Every level, including the boss level, passes.
- **Testing sprite-sheet asset loading:** The test for this passed, as we piped all images through the unix tool âĂŸconvert' which rid the image of any extra padding, that was causing collision issues.
- **Testing Browser Compatability:** This test partially passed, with Firefox, Chrome, and Safari running the game reasonably well, but with some other major browsers like Internet Explorer untested.
- **Testing Frames per Second:** This test passed, with the game fully capable of rendering at a smooth 60 frames per second on decent hardware. The game will, however render, faster in Chrome, as it has the superior V8 Javascript engine.

9.1.2 Gameplay Test Results

- **Testing RPG Explosion functionality:** This test passed, as we gave the Explosion a collision box, which fixed any issues with infinite damage.
- **Testing User Controlled RPG functionality:** This test passed, as we calculated the angle that the player was currently facing, and updated the rocket's velocity and position depending on that.
- **Enemy AI:** This test passed, by implementing a range check on each enemy, to make sure an enemy would only move when within a reasonable range of the player.
- **Testing Bullet Collision:** This test passed, as we implemented specific collision flags for walls and bullets, to make sure bullets would not pass through them.
- **Enemy Attack Speed:** This test passed, as we removed the ability for enemies to attack in such a short interval.
- **Testing Ammo Count:** The test for this passed, as we passed in the player object to the ui.js everytime the player shot. This updated the label on screen easily.

- **Testing Health Count:** The test for this passed, as we passed in the player object to the ui.js everytime the player was hit. This also updated the label on screen, similar to ammo.
- **Testing Number of Enemies:** The test for this passed, as we created a global variable that held the number of enemies. We would update this when an enemy was killed, or when a new level was loaded.
- **Testing Player Death Triggering âĂIJGame OverâĂİ:** This test passed with some simple front-end testing from the ui.js module.
- **Testing Ammo Count is accurate with all weapons:** This test partially passed, with some occasional negative ammo bugs appearing. We were not able to locate the source of this, but it only happens rarely, and is often not noticeable, and doesn't affect the flow of the game.
- **Testing Enemies Not Stealing Player Ammo:** This test passed, as we implemented our maps with barriers, placed around the ammunition that only the player could cross.

End of report