Code Name: LaserShark

Design Document

Presented by: Dan Henneberger, Xander Scott, and Steven Kennedy

For: COSC 325 / 625 Software Engineering



Scope 5

Introduction
System Objectives
Major Software Requirements
Design Constraints and Limitations

Data Design 7

Introduction
Data Objects and Structures
File Structures

Architectural Design 10

Review of Data and Flow Control
Derived Program and Structure

Interface Design 12

Interface Specifications
Interface Design Rules
Interface

beginning the game

Procedural Design - Main Module 17

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - Artillery Module 22

Abstract
Implementation
Design
Imports / Exports

Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - Debug Module 25

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - Entities Module 29

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - Utility Module 35

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output

Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - User Interface Module 47

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Procedural Design - World Module 50

Abstract
Implementation
Design
Imports / Exports
Pre and Post Conditions
Input / Output
Subparts
Concrete Implementation
Side Effects
Miscellany

Scope

Introduction

Code Name: LaserShark is a traditional top-down shooter injected with modern gaming conventions and a solid dose of immature humor to round out the experience. Like classic shooters such as Galaga or Space Invaders, you control your “ship” in the lower portion of the screen. As it flies forward, enemies and obstacles attempt to prevent you from reaching your goal. You face a variety of enemies and ultimately a boss monster at the end. Defeating your enemies earns you points.

This document presents the the design of this application. This document will contain the design and architectural implementation. The application utilizes a modularized approach where each section of the game works on its own and they each interact. Each section of this document describes the application modules, including implementation specifications, design, imports and exports, conditions, the concrete implementation, and other relevant information. This document describes this application in detail and enables the development team to complete the program. It could also be used for porting this application to other platforms.

System Objectives

Codename: LaserShark is a fair-sized project. Some parts of the application utilize third party libraries to shorten development time. The game is a two dimensional shooter, and as such will not take too much manpower. The goal is to create a light-weight application that still manages to entertain its users. Modules are kept independent and coupling reduced to a minimum, though in an application of this sort, some elements are not entirely useful outside of this specific implementation. Game Entities can be added with little effort, and there are few modules that stand out very differently from others. Simplicity is an important aspect overall for a project of this size with a team of three.

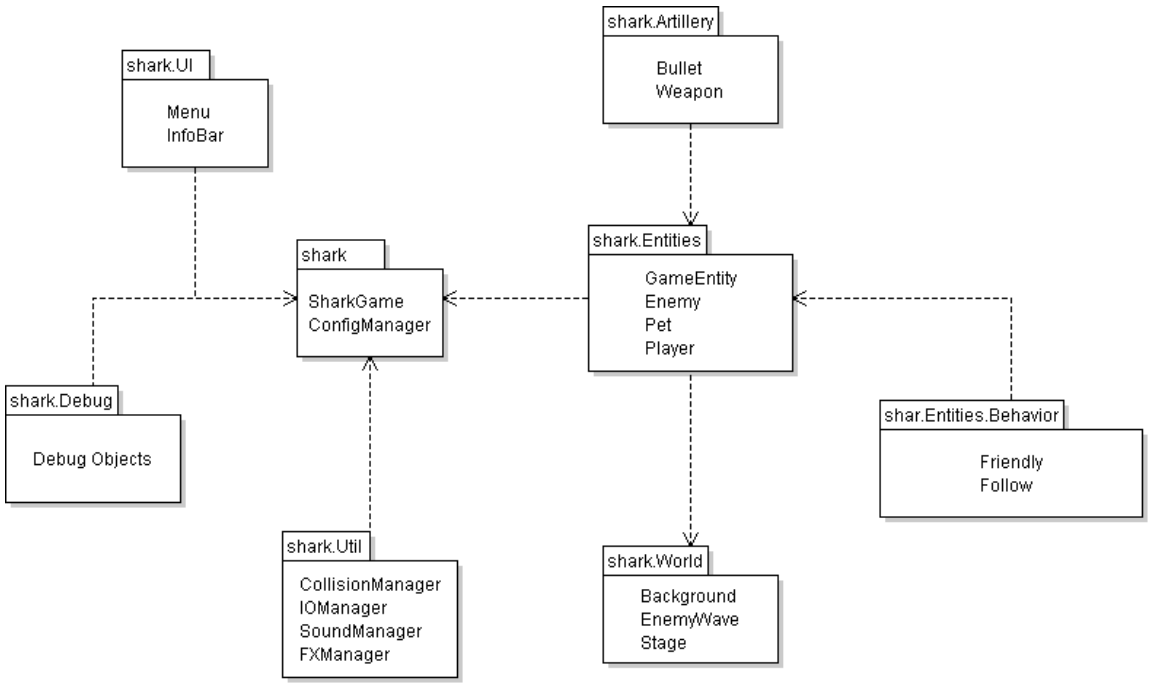
Major Software Requirements

Running this application requires modest hardware running any modern Windows, Linux, or OS X operating system and an OpenGL compatible video card. Since it runs Java, Slick2D is available for most major platforms, but it utilizes the native libraries of LWJGL. These must be available in addition to Java 5.0 to run the LaserShark game, but they can be distributed with the application. As Java exists rather commonly, this requirement is not difficult to meet.

Design Constraints and Limitations

The Java platform presents the primary design limitation for this project other than time itself. Java creates a few problems for games. First, the JRE must be running, which presents extra cost during runtime. On most systems, this would not present an issue, but

the overhead would prevent lower specced systems. Additionally, the garbage collection presents a unique problem for a game. The game is meant to run at a consistent pace, and the regular updates and rendering are key to achieve this result. The garbage collector cannot be controlled, and will run on its own terms. It will interrupt some of these updates. This is mitigated by separating logic and graphical updates, but the existence of the garbage collection is a necessary evil when developing in Java.



package relationships

Data Design

Introduction

LaserShark uses a few different data structures and design patterns to represent the application. The data structures are provided by Java and are described here.

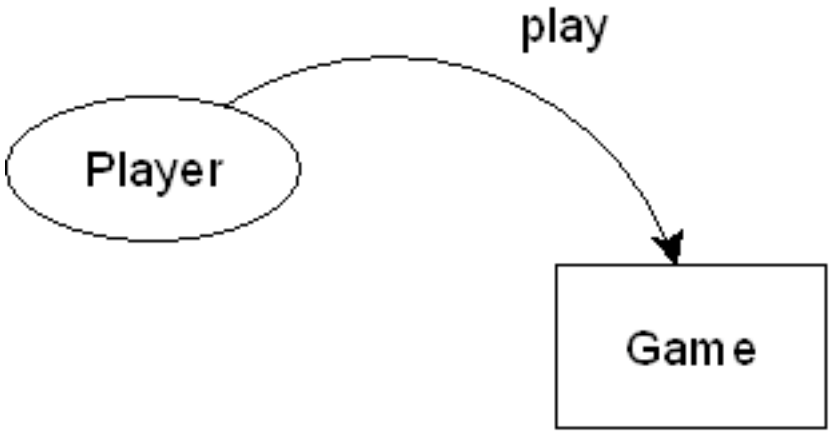
Data Objects and Structures

The main game class maintains references to all relevant objects. Game Entities are referenced in linked lists, separated by specific type. Main entities, bullets, and enemy bullets are each in separated lists. The main game class exists almost entirely as a single loop that ends only when the game does. A few states control the actions of the game. It can be paused, which brings up the in-game menu, and it can be running before or after the actual gameplay. During these loops, the game logic is updated and the graphics are drawn. The main class sends update requests to all active objects, and the rest are discarded for garbage collection. Hash Maps are used for referencing sound files by string, and text files are used for saving data between sessions.

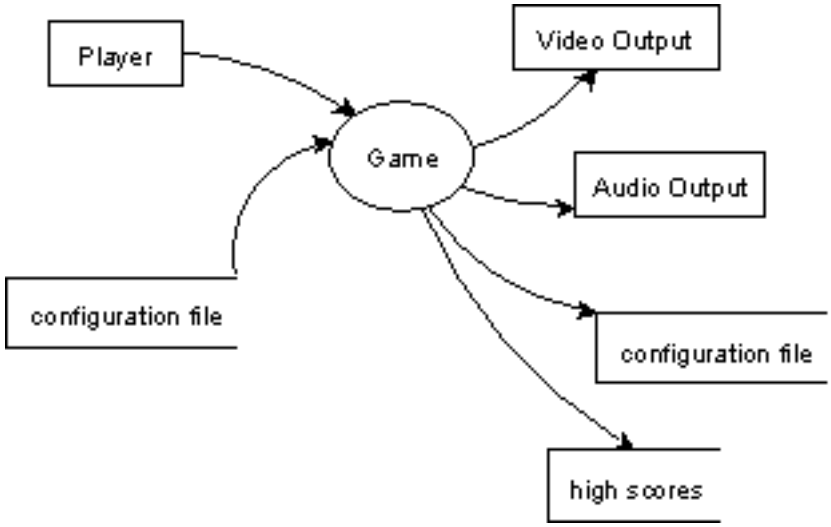
The collision detection compares objects for position within a rectangular area roughly the size of the object. The separate lists effectively prune the number of necessary collision comparisons, as bullets are not compared against other bullets.

The levels themselves are effectively linked lists. When the game requests it, the current stage sends a wave of enemies. Each wave is individually a linked list, and the stage contains a list of waves, sending a wave at a time.

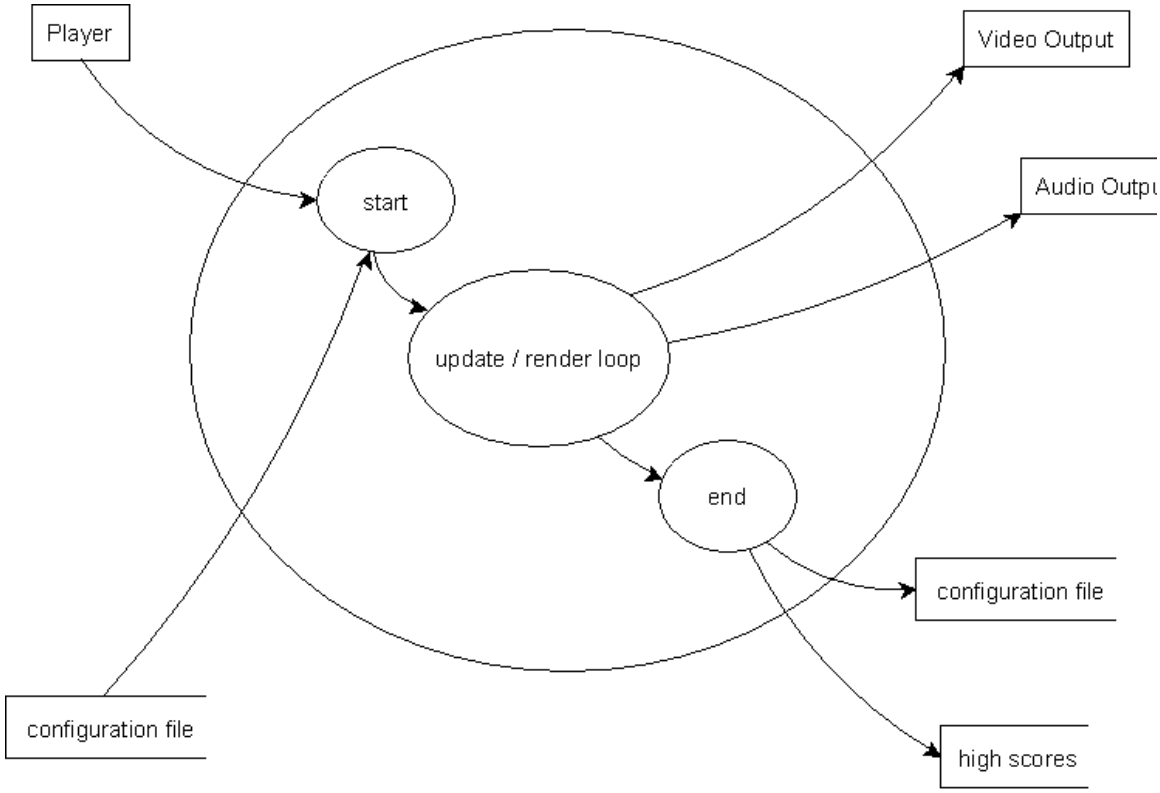
Game Entities encapsulate all of the data that is important for an object to exist in the game world, such as coordinates, name, and health. Behaviors for enemies and pets are separate objects that give commands to those objects. They can change this behavior state for different gameplay. Pets, enemies, and the player are essentially the same, but in the case of the player, the user controls the behavior.



data flow context



data flow diagram level 0



data flow diagram level 1

File Structures

Two files save the game data. Upon loading the application, the configuration file is loaded. The file has the statistics and configuration for the relevant user. The file is organ-

ized into key maps, personal high scores, and user statistics. The files are organized something like the following example.

```
[user]
[keymap]
UP 99
DOWN 102
RIGHT 100
LEFT 101

[highscore]
999999
135325
3434
11
0
0
0
0

[stats]
fired 9999
died 53
killed 9000

[user2]
...
```

example configuration file

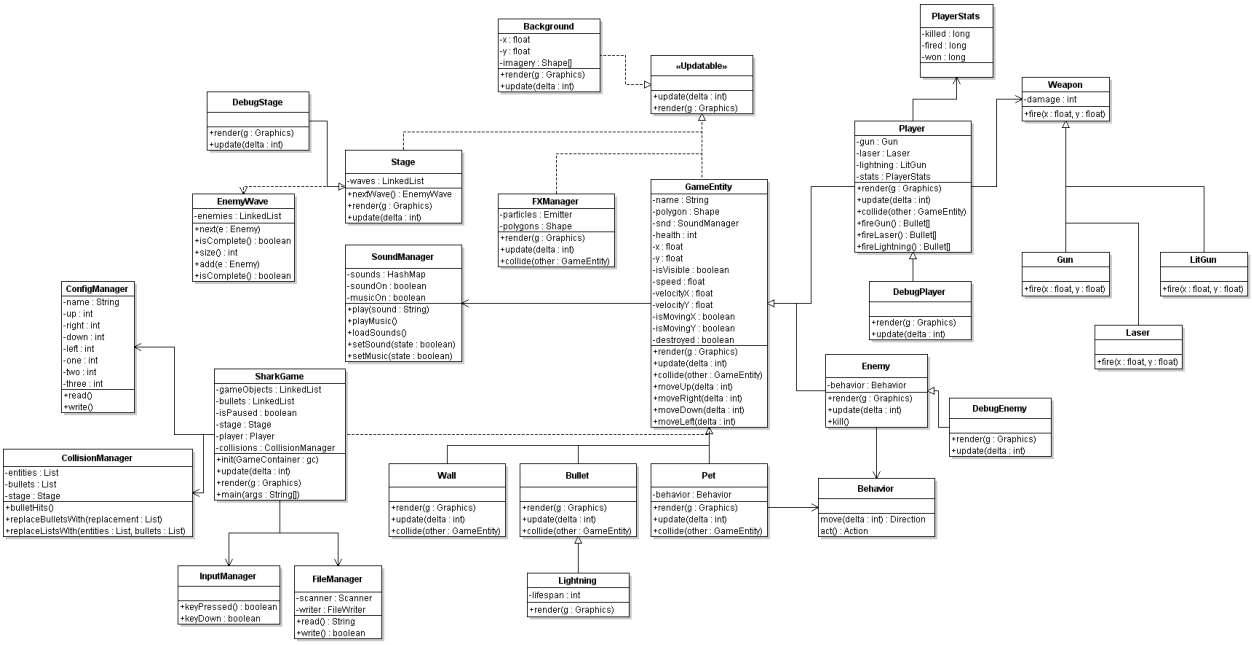
Architectural Design

Review of Data and Flow Control

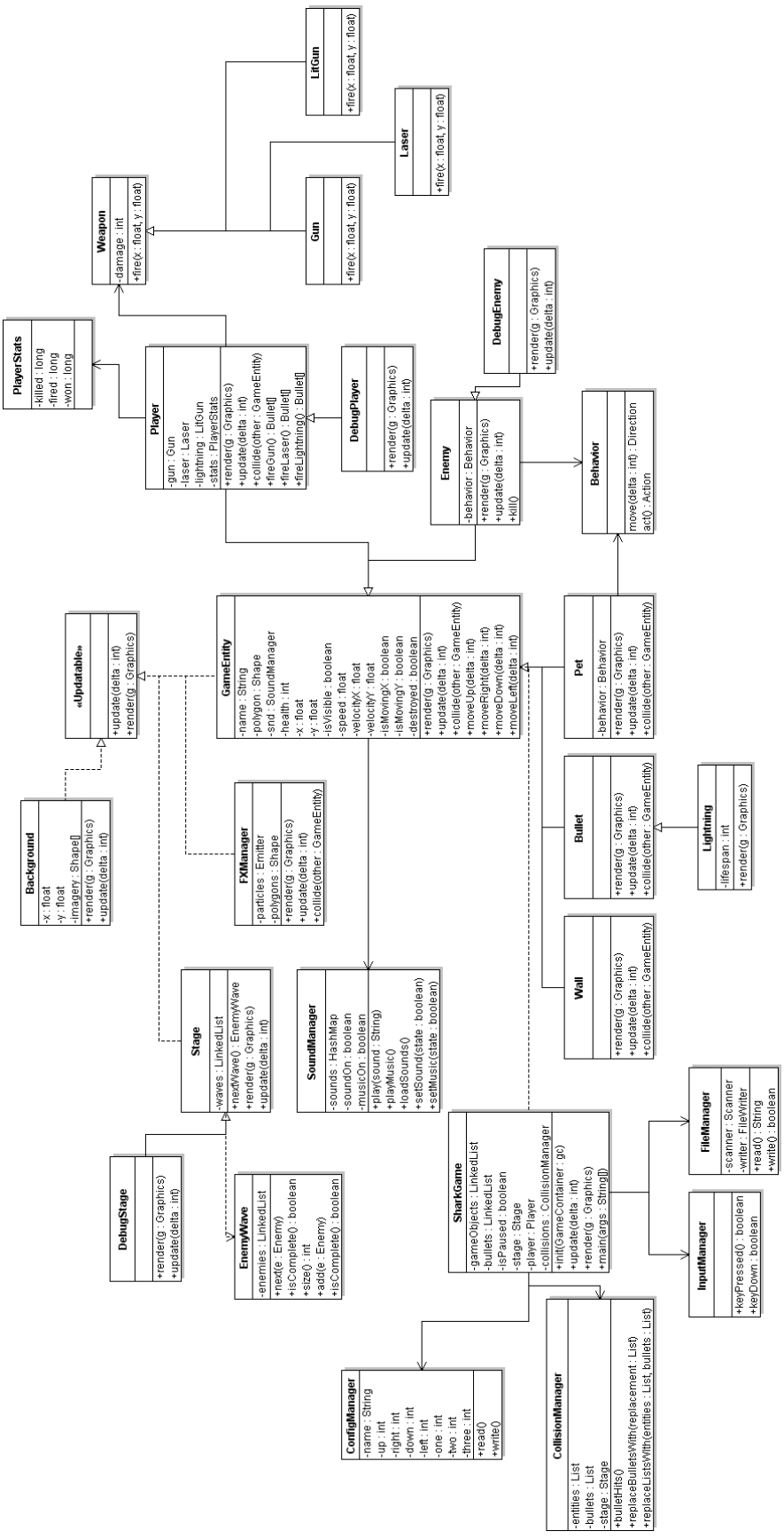
The data comes in from two sources. The first is directly from the user, and the second is the configuration file upon loading. The configuration file is used to initialize the Input and keep track of data for the player.

Derived Program and Structure

The SharkGame class is the core of the entire application. It controls all the utility classes and calls the game entities to update and render. The game entities are descendants of an abstract class that implements an Updatable interface. Other objects that need updated, but that are not inhabitants of the game world can also implement Updatable. The in-game instances are all concrete versions of the relevant classes. The following class diagram demonstrates the class relationships.



class diagram (expanded on next page)


LitGun
+fire(x : float, y : float)
Laser
+fire(x : float, y : float)
Weapon
-damage : int
+fire(x : float, y : float)
Gun
+fire(x : float, y : float)
PlayerStats
-killed : long
-fired : long
-won : long
Player
-gun : Gun
-laser : Laser
-lightning : LitGun
-stats : PlayerStats
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
+fireGun() : Bullet[]
+fireLaser() : Bullet[]
+fireLightning() : Bullet[]
DebugPlayer
+render(g : Graphics)
+update(delta : int)
DebugEnemy
+render(g : Graphics)
+update(delta : int)
Enemy
-behavior : Behavior
+render(g : Graphics)
+update(delta : int)
+kill()
Behavior
move(delta : int) : Direction
act() : Action
«Updatable»
+update(delta : int)
+render(g : Graphics)
GameEntity
-name : String
-polygon : Shape
-snd : SoundManager
-health : int
-x : float
-y : float
-isVisible : boolean
-speed : float
-velocityX : float
-velocityY : float
-isMovingX : boolean
-isMovingY : boolean
-destroyed : boolean
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
+moveUp(delta : int)
+moveRight(delta : int)
+moveDown(delta : int)
+moveLeft(delta : int)
Pet
-behavior : Behavior
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
Background
-x : float
-y : float
-imagery : Shape[]
+render(g : Graphics)
+update(delta : int)
FXManager
-particles : Emitter
-polygons : Shape
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
Bullet
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
Lightning
-lifespan : int
+render(g : Graphics)
Stage
-waves : LinkedList
+nextWave() : EnemyWave
+render(g : Graphics)
+update(delta : int)
SoundManager
-sounds : HashMap
-soundOn : boolean
-musicOn : boolean
+play(sound : String)
+playMusic()
+loadSounds()
+setSound(state : boolean)
+setMusic(state : boolean)
Wall
+render(g : Graphics)
+update(delta : int)
+collide(other : GameEntity)
DebugStage
+render(g : Graphics)
+update(delta : int)
EnemyWave
-enemies : LinkedList
+next(e : Enemy)
+isComplete() : boolean
+size() : int
+add(e : Enemy)
+isComplete() : boolean
SharkGame
-gameObjects : LinkedList
-bullets : LinkedList
-isPaused : boolean
-stage : Stage
-player : Player
-collisions : CollisionManager
+init(GameContainer : gc)
+update(delta : int)
+render(g : Graphics)
+main(args : String[])
FileManager
-scanner : Scanner
-writer : FileWriter
+read() : String
+write() : boolean
InputManager
+keyPressed() : boolean
+keyDown : boolean
ConfigManager
-name : String
-up : int
-right : int
-down : int
-left : int
-one : int
-two : int
-three : int
+read()
+write()
CollisionManager
-entities : List
-bullets : List
-stage : Stage
+bulletHits()
+replaceBulletsWith(replacement : List)
+replaceListsWith(entities : List, bullets : List)

Interface Design

Interface Specifications

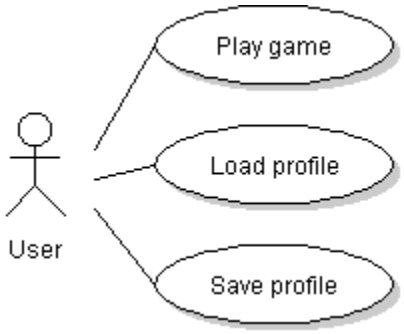
The user interface runs on the appropriate Java platforms. The user's keystrokes are sent to the main game and passed to the classes in control of each part of the game.

Interface Design Rules

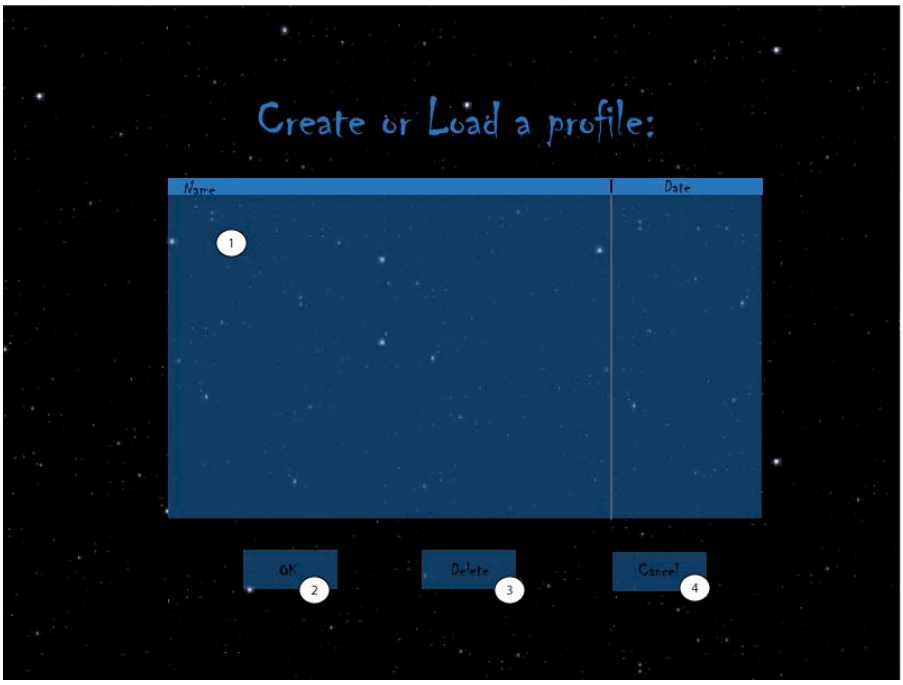
The game's main menu interface works the way classic games such as Doom and Quake. The menu is drawn to the screen when the game is paused and the choices can be selected during playtime. The gameplay itself resembles classic scrolling arcade games. The directional keys move the player's avatar, and the buttons fire the weapons. The keys are mapped by the player.

The positioning of objects are also important. The menus must be centered to the screen and drawn with sans-serif fonts. The status box renders to the edge of the screen where it would not interfere with gameplay. This box allows the user to see through it, but draws important information that should always be visible during gameplay. The player begins each stage from the bottom center of the screen, and the enemies come down from the top. The pets follow the player.

Interface



beginning the game



profile loading (mockup)

The above screen represents the loading area. The player enters their name here. Previous saved profiles will also display here.

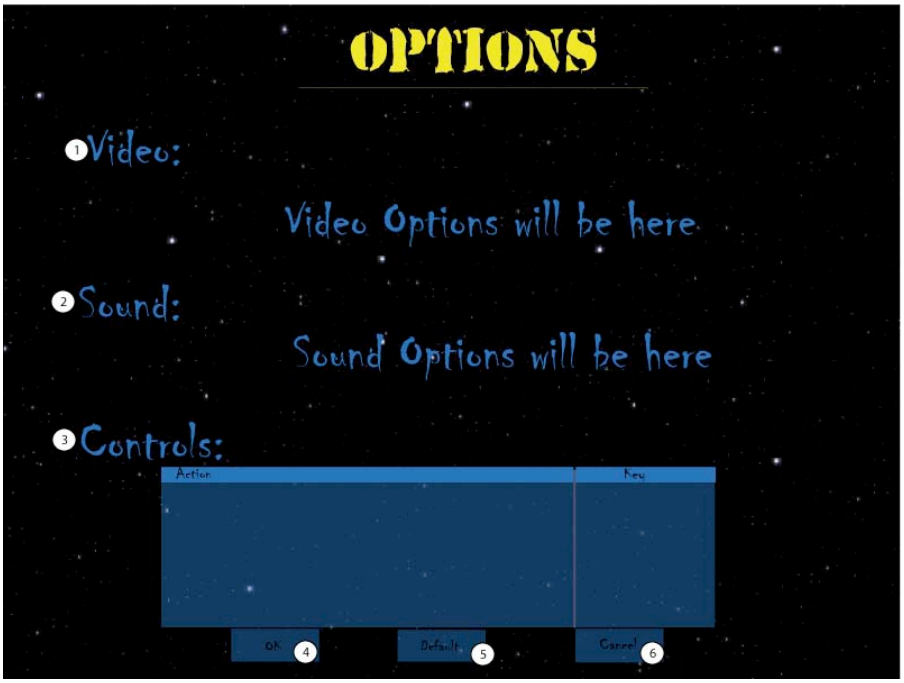
1) The input / out area
2) Confirmation button
3) Profile delete button
4) Canceling the loading screen will exit the game



game menu (mockup)

The in-game menu allows the user to make minor changes or exit the game at play-time. This menu comes up when the game is paused.

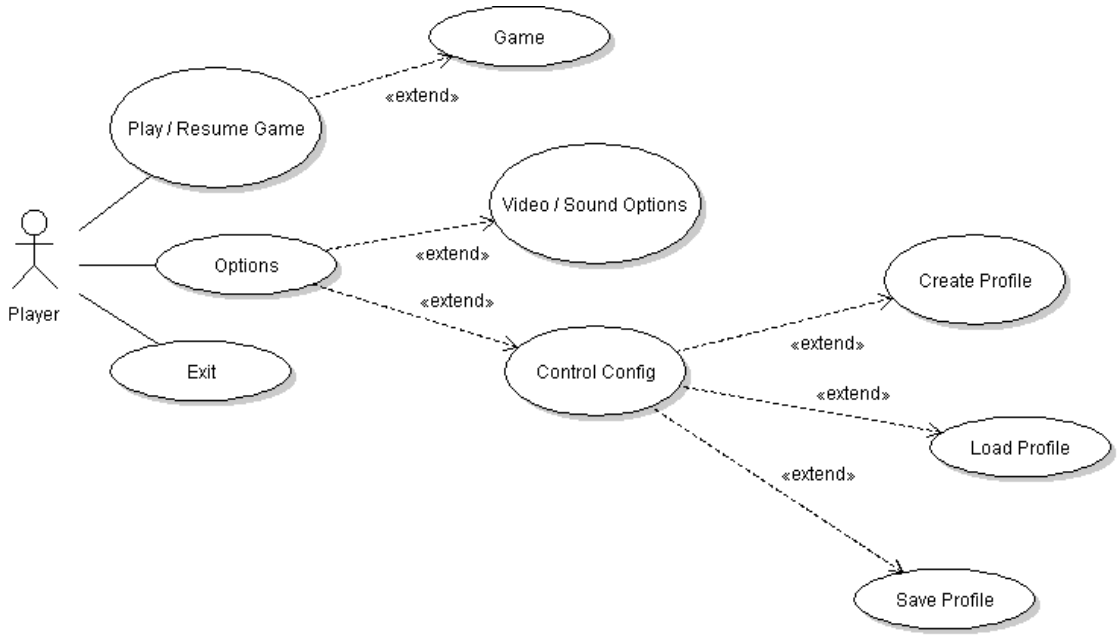
1) the icon showing the currently selected option
2) play / resume the game
3) adjust options
4) exit the game



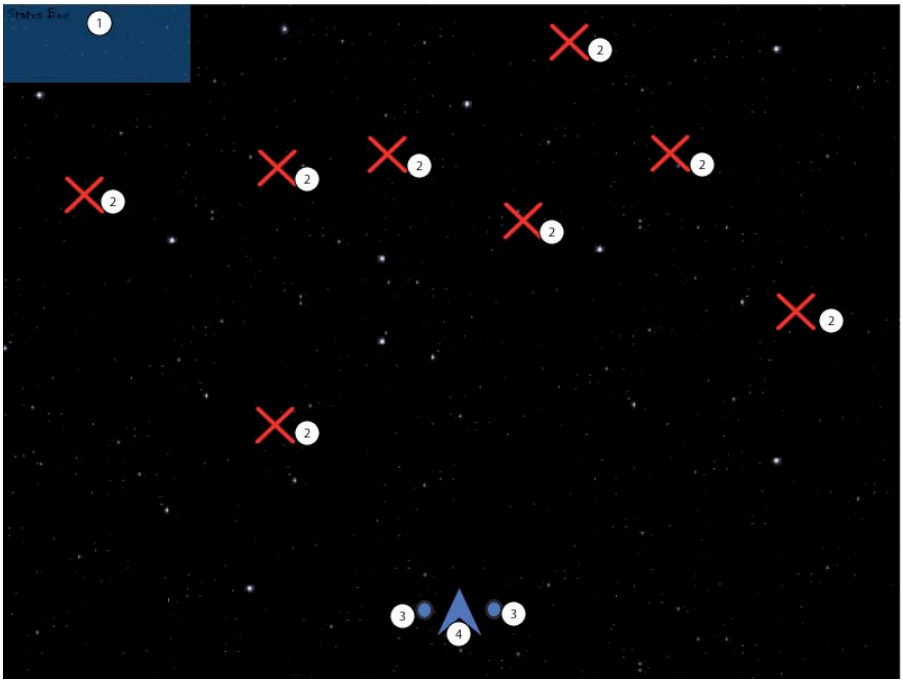
options menu (mockup)

A mockup options menu shows the types of options the player may find.

4) save the game
5) cancel and return the user to the previous menu



menu selection



gameplay (mockup)

The gameplay mockup shows how the general gameplay progresses.
1) the status box contains the player health, score, etc.
2) the enemies
3) the player's pets
4) the player's in-game avatar



pause menu (mockup)

The pause version of the game menu is essentially the same as the startup version.

1) the icon pointing to the currently selected option
2) resume the game
3) open the save profile and write to it
4) bring up the options menu
5) exit the game

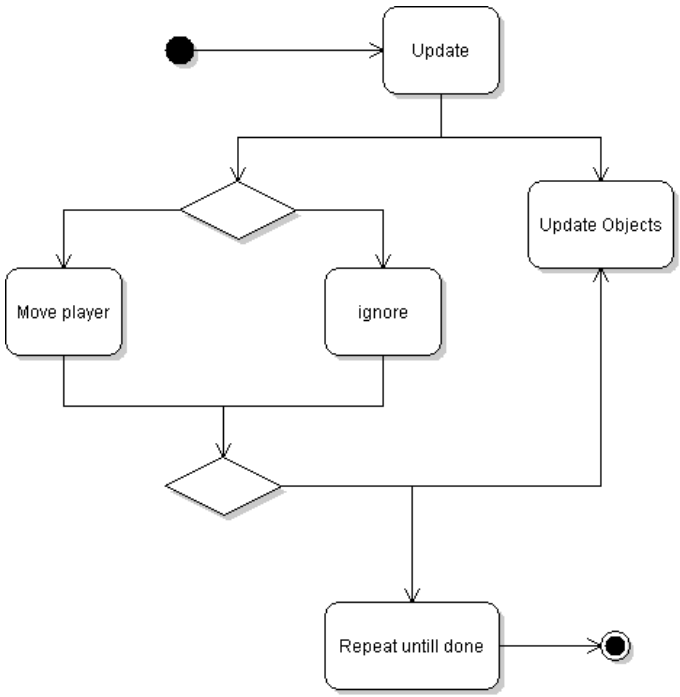
Procedural Design - Main Module

Abstract

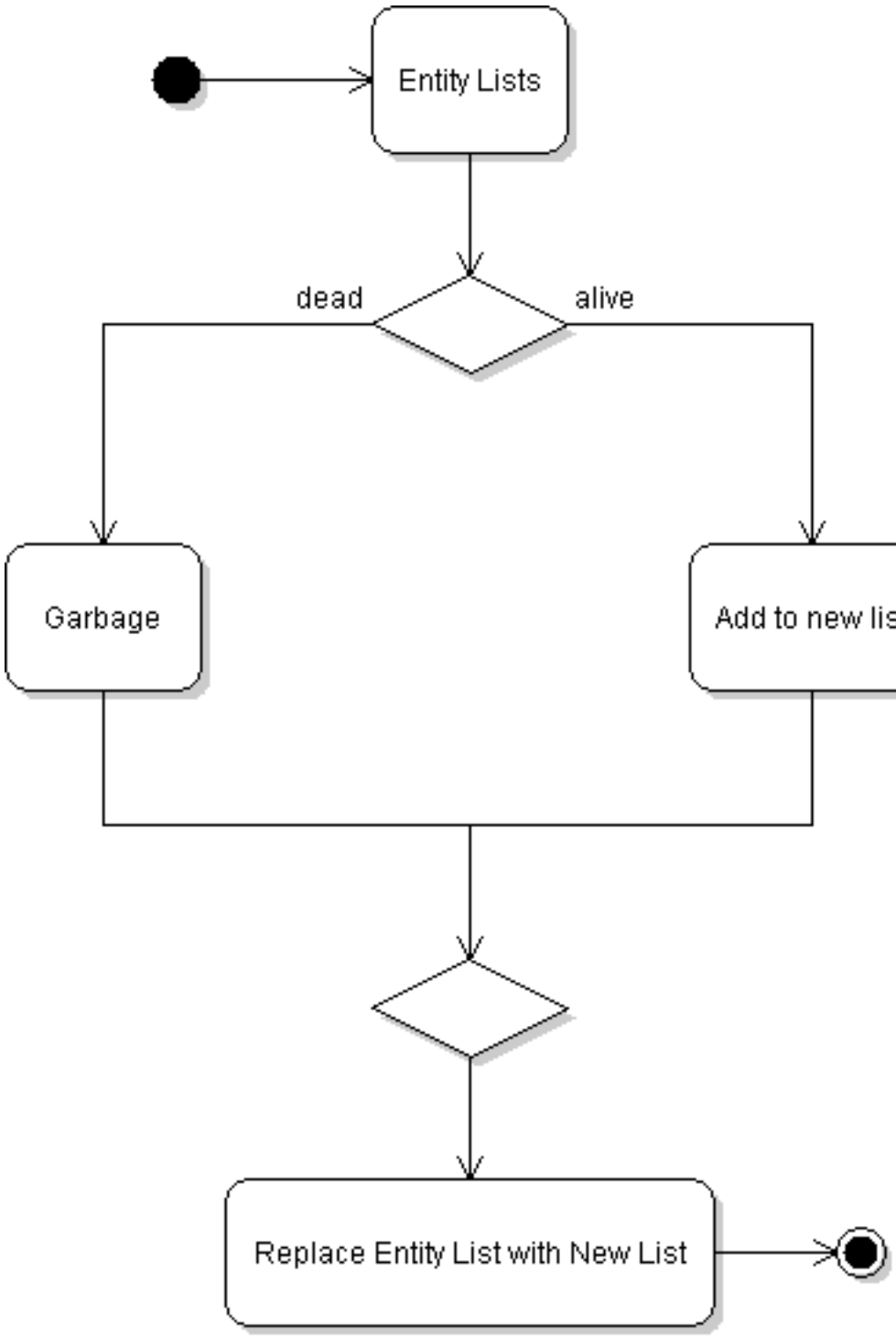
The main module contains the main game object and the configuration. The main object is responsible for all core features of the application, and the configuration handles settings that have been adjusted by the user. The main object references all other relevant objects, such as the stage, input handler, and collision handler. It calls upon each game object to update and render. The rendering occurs separately from the updates, allowing for unimportant renders to be skipped while the game updates continue. Relying on the relative time that has passed between updates, it maintains a constant pace on any capable system.

Implementation

The implementation for the main components is handled by Slick2D. The time between loop iterations is averaged to calculate how often the render method should be called. For LaserShark, the graphics are rendered 60 times a second. Essentially every object in the game is updated.



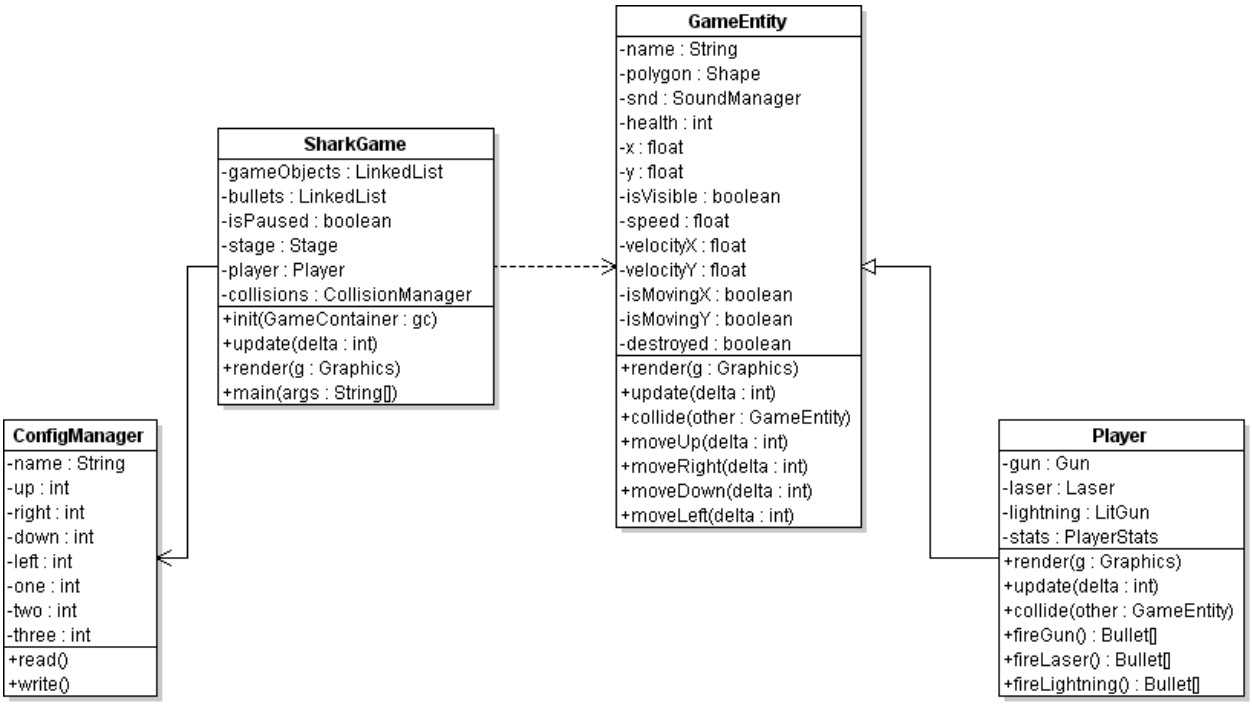
update activity



entity cleanup

Design

The main module controls the game. When the user begins the application, the main module controls the sequence of events that loads the configuration.



classes in this module

Imports / Exports

The main class imports the important Java data structure implementations as well as the utilities to read files and output OpenGL and sound.

Pre and Post Conditions

N/A, because the main class begins and ends the game, so when it is not running, the application has exited.

Input / Output

The user's input is sent to the main class. The input is sent to the player's avatar or the menus when appropriate. After the player has moved or fired, the game continues updating. The collision detection manager makes comparisons during updates, and the corresponding actions occur. After entities are updated, the data structures are traversed to seek the status of the entities. Objects that have become inactive or killed are removed from the data structures and left for garbage collection.

Subparts

The important parts of this module are the main class and the configuration object. The configuration objects assists in writing the config in the right format as well as reading in and defining the required values. The background image, stage data, polygons, map boundaries each represent separate parts of the application.

Concrete Implementation

```
// prepare data and initialize objects
loop(untilGameEnds) {
   // the graphics context is taken from Java
   render(Graphics g);
   // the delta is the ms since the last update
   update(int delta);
   // clean up inactive objects
   cleanUp();
}

render(Graphics g) {
   // call every game entitiy's render
   // some methods from Slick2D will force OpenGL
   g.setColor(Color.gray);
   for (Shape s : imagery) {
      ShapeRenderer.fill(s)
}

update(int delta) {
   // call every game entity's update
   for (GameEntity g : objects) {
      g.update(delta);
   }
}

cleanUp() {
   for (GameEntity g : objects) {
      if (g.isInactive())
         removeFromList();
   }
}
```

Side Effects

N/A, though in some sense, the entire game is a side effect.

Miscellany

N/A

Procedural Design - Artillery Module

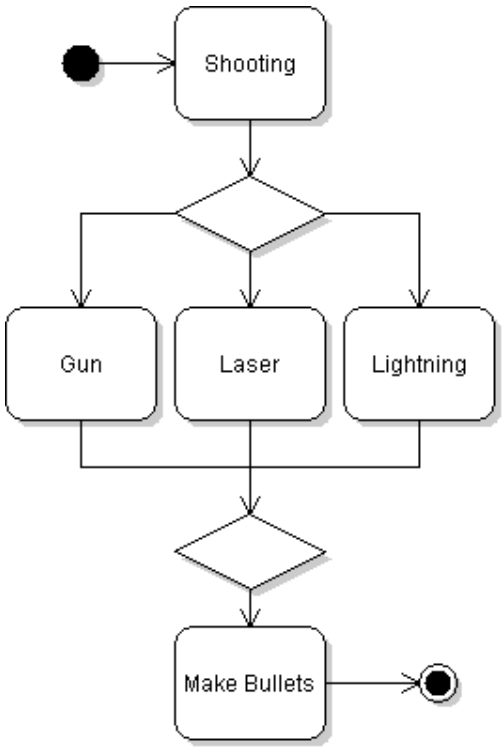
Abstract

In this specification, all bullets are considered game entities. Weapons represent containers that can create the bullets and place them into the game world. The bullets maintain their speed and direction as Game Entities, but the weapon that fires them also hold their damage and type.

Implementation

The bullets are game entities with no behavior. They continue in the direction they began until they hit another non-bullet entity or exit the play area. Upon colliding with another entity, the bullet and that entity are sent to the collision manager for action. Each type of concrete bullet implementation exists as its own class. While most bullets don't render differently from a standard object, the lightning bullets animate as lightning. The lightning traces a couple of jagged lines to the top of the viewport repeatedly for effect.

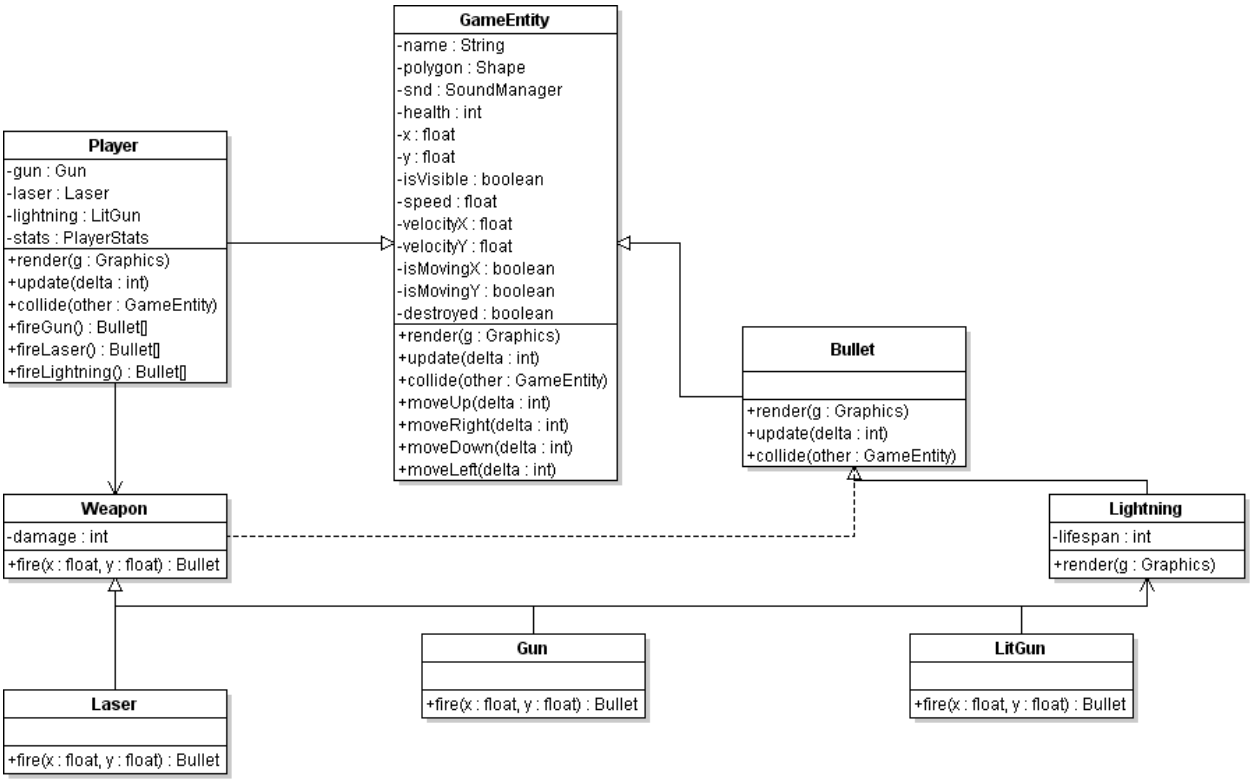
The weapons are simple data structures that create the bullets and fire them into the game world. The weapons manage the details and create an array of bullets to send to the main game class. An array is created, as multiple bullets can be fired simultaneously.



shooting

Design

The bullets are extensions of GameEntities, which provide all the necessary requirements for their existence in the game world. The weapons send the bullets into the game world.



artillery classes

Imports / Exports

N/A. The artillery module does not require any imports, but the use of the module does create a number of bullets that the calling game world must be able to manage.

Pre and Post Conditions

In order for the bullets to fire properly, the weapon needs to exist in an object that is already in the game world. Once the weapon has fired, an array of bullets is added to the world. The specifics are handled by the bullets themselves.

Input / Output

The fire requests are sent from the enemies and player to the corresponding weapon objects. The weapons return the appropriate bullets.

Subparts

The only parts of the artillery module are the bullets and the weapons. The concrete implementations of each of these specify the attributes and / or behaviors specific to the specific weapon / bullet.

Concrete Implementation

```
// the primary weapon abstract class
public abstract class Weapon {
   // when fired, the weapon adds the bullets to the world
   public Bullet[] fire(float x, float y);
}

// inherits from Game Entity
public class Bullet {
   render(Graphics g);
   update(int delta);
   collidesWith(GameEntity other);
}
```

Side Effects

The animation is handled in the same thread, so the drawing is a side effect of this implementation.

Miscellany

N/A

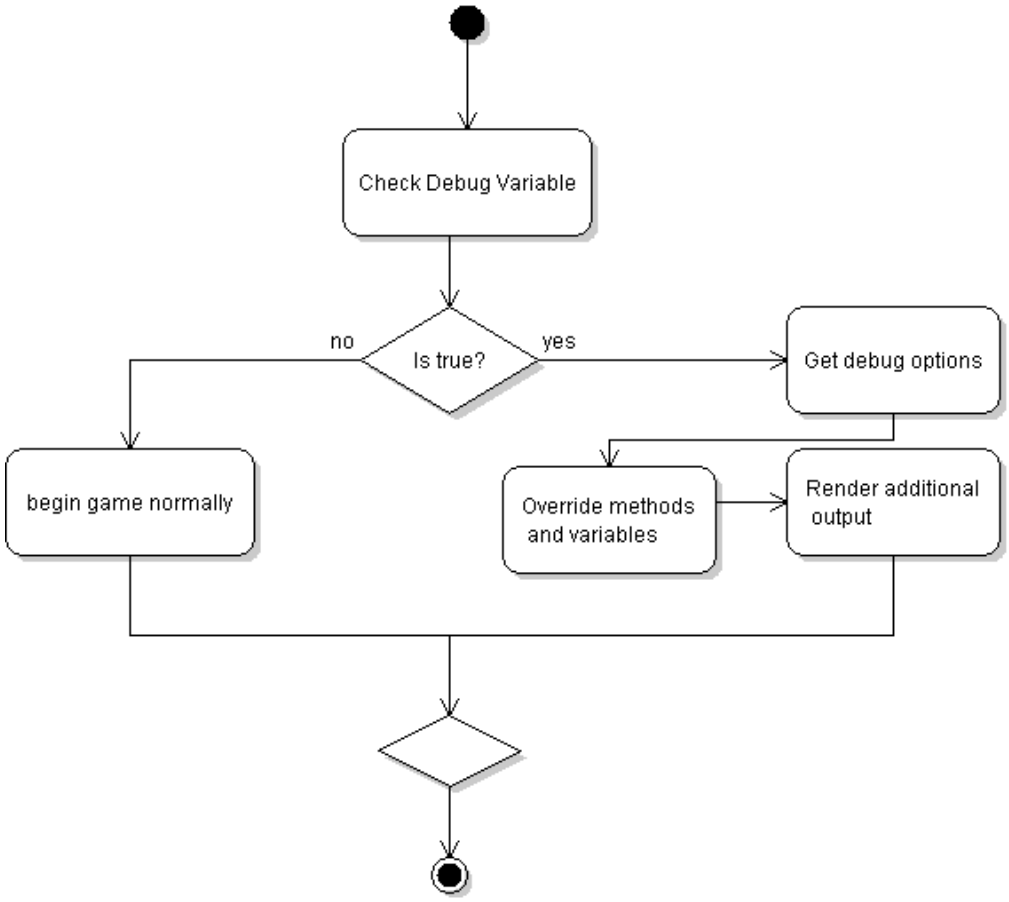
Procedural Design - Debug Module

Abstract

The debug module never shows itself to the user. It provides more verbose information to the developers and testers. In addition, debug objects set up idealized situations for testing. Debug objects are essentially subclasses of other objects with additional statements for outputting extra information. When running in debug mode, the game displays the frame rate and enforces certain variables and results.

Implementation

The main class controls the debug objects the same as standard game objects, though it will load debug objects where necessary. The debug objects themselves override methods to extend or change the behavior in a way that is suitable for testing. For example, a debug enemy can print out its current direction and when it plans to fire at the player rather than or in addition to the action itself.



debug load-up procedure

Design

Debug objects are extensions of normal game objects.

Imports / Exports

N/A

Pre and Post Conditions

The game world must be active in order for the debug objects to mean anything to the user testing the game. The debug flag must be set to true, or the debug objects will not be instantiated. The post conditions vary depending on the user input but stay essentially the same. The added output will be in the form of a system printout or log file.

Input / Output

The input stays the same, but additional output is sent to the screen.

Subparts

N/A

Concrete Implementation

```
// The debug classes will extend concrete classes
@Override
public void kill() {
   super.kill();
   // additional output is added (for humans)
   System.out.println(this + " died.  " + isDestroyed());
}

// debug objects initialize values in specific ways
public DebugEnemy(float dx, float dy) {
   super("Debug Enemy", new TestBehavior());
   flash = new Color(Color.red);
   timer = 0;
   polygon = new Circle(320, 100, 10);
   x = dx;
   y = dy;
   speed = 3.0f;
   id = nEnemies++;
}

// updates add additional steps for the specific class
public void update(int delta) {
   super.update(int delta);
   if (timer >= DONE) {
      color = flash;
      flash = new Color(Color.white);
      timer -= DONE;
   }
   // action = TestBehavior.act();
   // etc
}

// add steps to the render
public void render(Graphics g) {
   super.render(g);
   g.drawString("hello world");
}
```

Side Effects

When running in debug mode, there is additional overhead created by the extra output. This would never matter in a real world gameplay situation, but debug users need to be aware of this fact.

Miscellany

The debug will never be fully complete or polished, as it will not matter for deployment. Testing users need to be aware of this fact. The debug is primarily used for setting up scenarios for testing purposes, and will be used almost exclusively by those on the development team.

Procedural Design - Entities Module

Abstract

Any object in that exists in our SharkGame world is a GameEntity. Each entity has the relevant attributes that allow it to exist in the simulated space, such as coordinates, a polygonal shape, a collision area, and the relevant behaviors, namely updating and rendering, specified by the Updatable Interface. Any object that would be added to the game world must extend GameEntity in order to behave properly. After doing so, it would function in a very basic fashion. Adding specialized attributes, behaviors, and images would make for a simple addition to the world population.

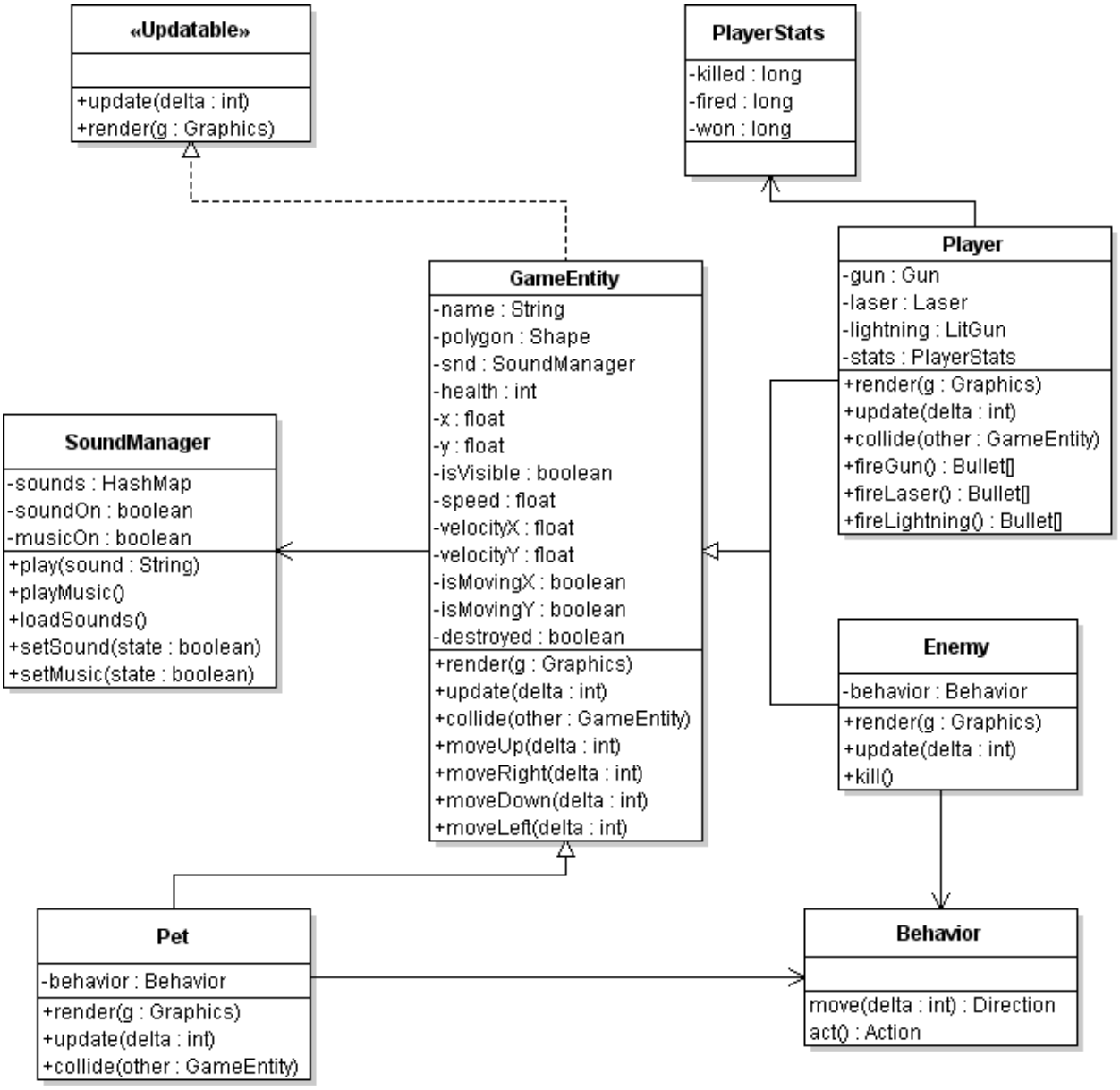
Implementation

GameEntity exists as an abstract class, and as such each type of game entity must subclass it. The primary types of subclasses are enemies, bullets, and the player. Each adds its own variations, but are all essentially the same type of object. The enemies and player objects add a few significant differences. The enemies have a behavior state that can change depending on the required situation. For instance, when appearing, they will move into a specific position, after which, they will change to an attacking behavior. The player's pets are essentially enemies, but they are given friendly behaviors.

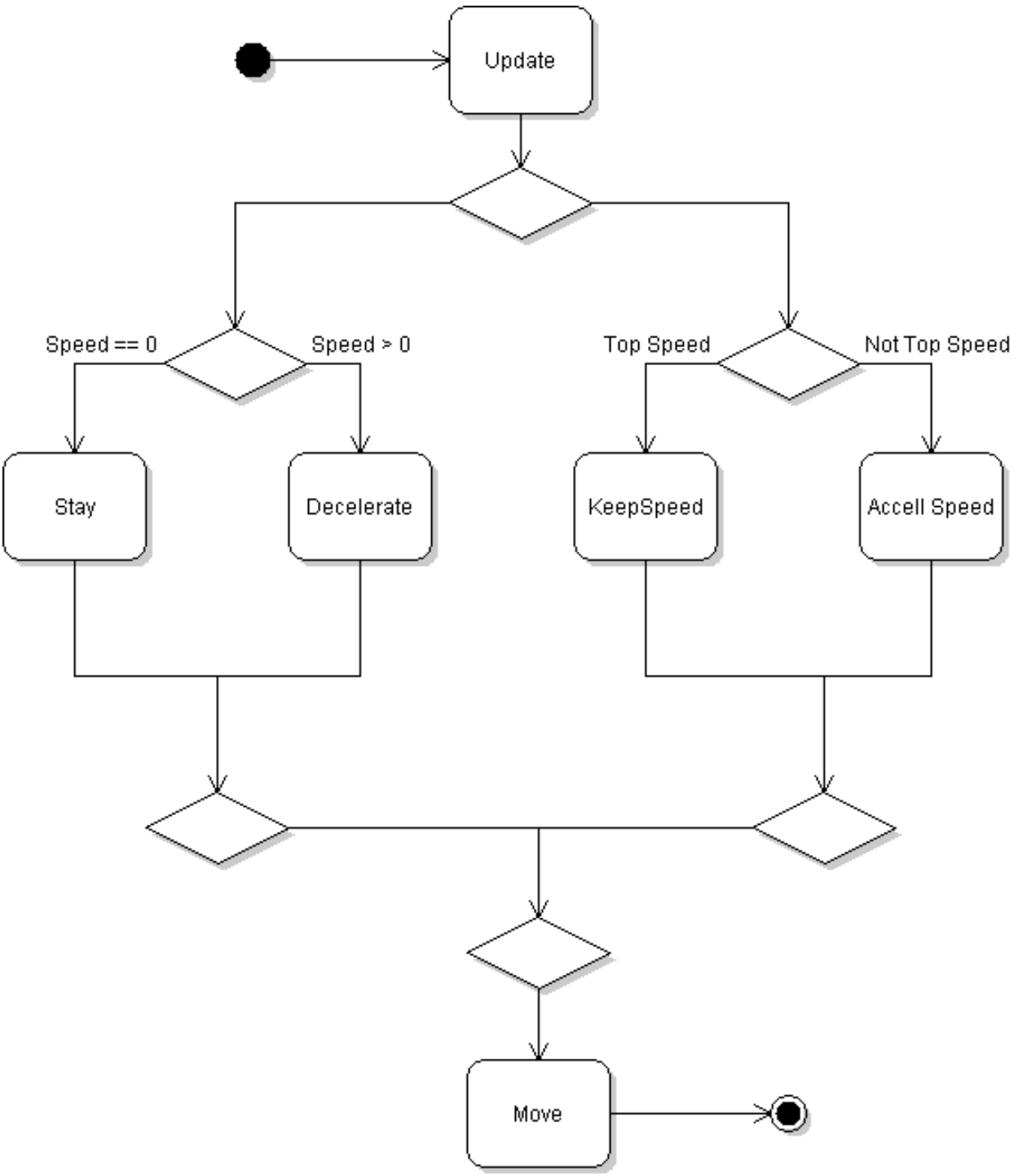
The player object is rather different. The player's state is held in this object in addition to the relevant entity data. The SharkGame refers to the player regularly to allow the user to control the movement and firing. In addition, the player has special collision checks just for it.

Design

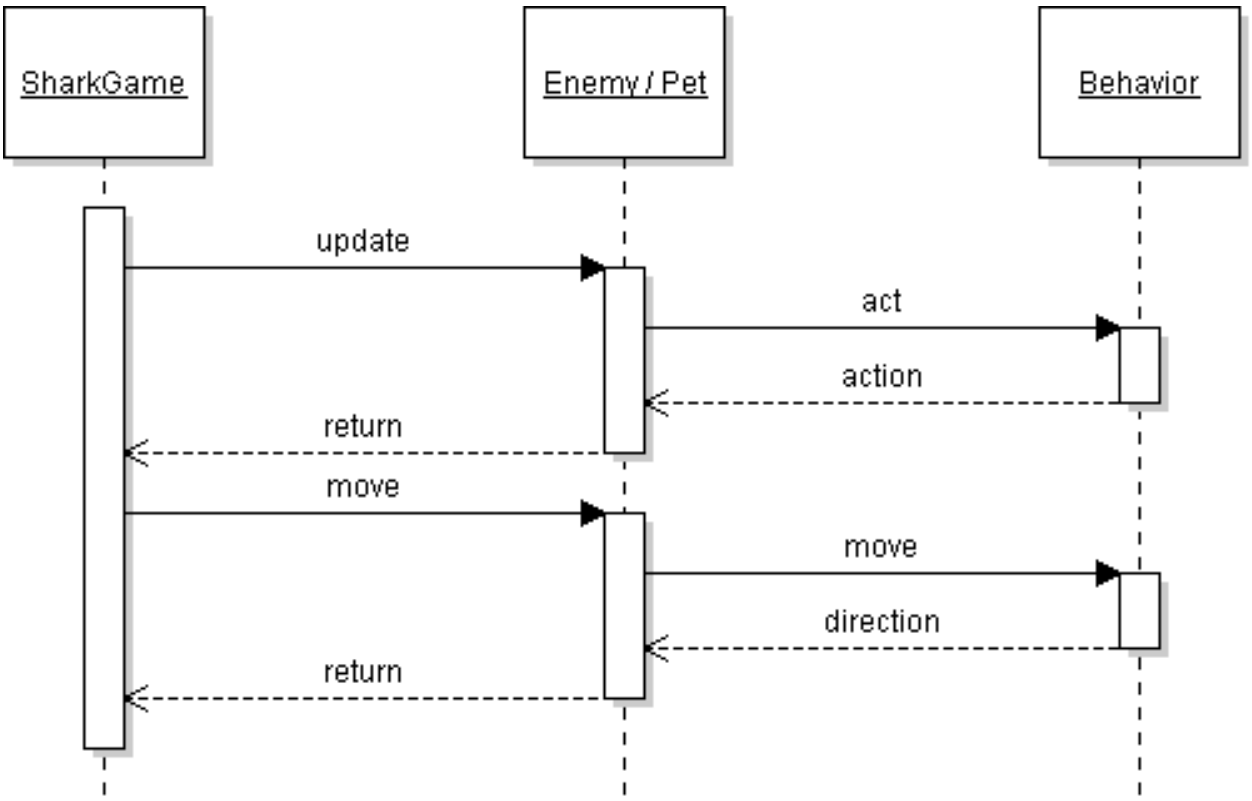
As each object added to the game world is essentially similar, they can all inherit important behaviors and attributes from a parent class. Aside from specialized behaviors, there is not much additional modification needed. Having objects to represent behavior allows enemies and pets to have specified routines that they can act out.



game entities


Update
Speed == 0
Speed > 0
Top Speed
Not Top Speed
Stay
Decelerate
KeepSpeed
Accell Speed
Move

movement



behavior sequence

Imports / Exports

Nothing is imported into the game entities from other modules, but the entities are highly important elsewhere. They are the objects that are collided, rendered, updated, and interacted with.

Pre and Post Conditions

There need be a game world for the entities to exist. The main class will handle the render and update calls. When the objects are created, they exist until they are destroyed, at which point they are mostly forgotten, excepting the player, which passes his score and stats for record keeping.

Input / Output

The only input comes from the player. The keys entered call the appropriate actions, and the player reacts accordingly. All output is drawn to the screen using methods inherited from BasicGame (and ultimately from LWJGL's native OpenGL libraries) or sent to the speakers (sound).

Subparts

The entity behaviors exist as a subpart to this module. Enemies and pets act according to the individual behavior specifications.

Concrete Implementation

```
public class ExampleEntity extends GameEntity {
   update(int d) {
   // do stuff relative to d
   }
}

render(graphics g) {
   g.draw(polygon);
}

act() {
   switch(behavior.act()) {
   case MOVE:   move();       break;
   case FIRE:   fire();       break;
}
```

Side Effects

The majority of core results of having a GameEntity are side effects, as they do not pass data. They are rendered to the screen, killed, replaced, etc.

Miscellany

N/A

Procedural Design - Utility Module

Abstract

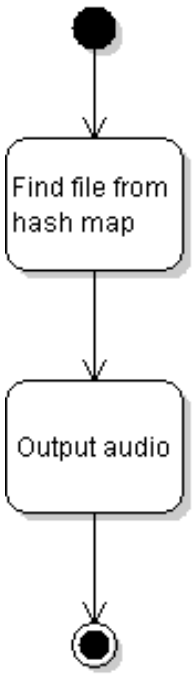
The utility module is a fairly involved module that handles the collision detection system, the file I/O, the sound manager, and additional graphical effects not rendered directly by Slick2D.

The collision's are called by the main class. The calls are pruned preemptively by separating the types of entities. The collision manager compares each entity in the lists it is sent against each other. As game entities, they have collision methods that are called by the manager. By default, the entities are destroyed immediately, but those entities can override that behavior for their use. The comparison method is a brute force strategy, so in order to prevent comparisons with inactive objects, they are removed from the lists before they are passed to the manager.

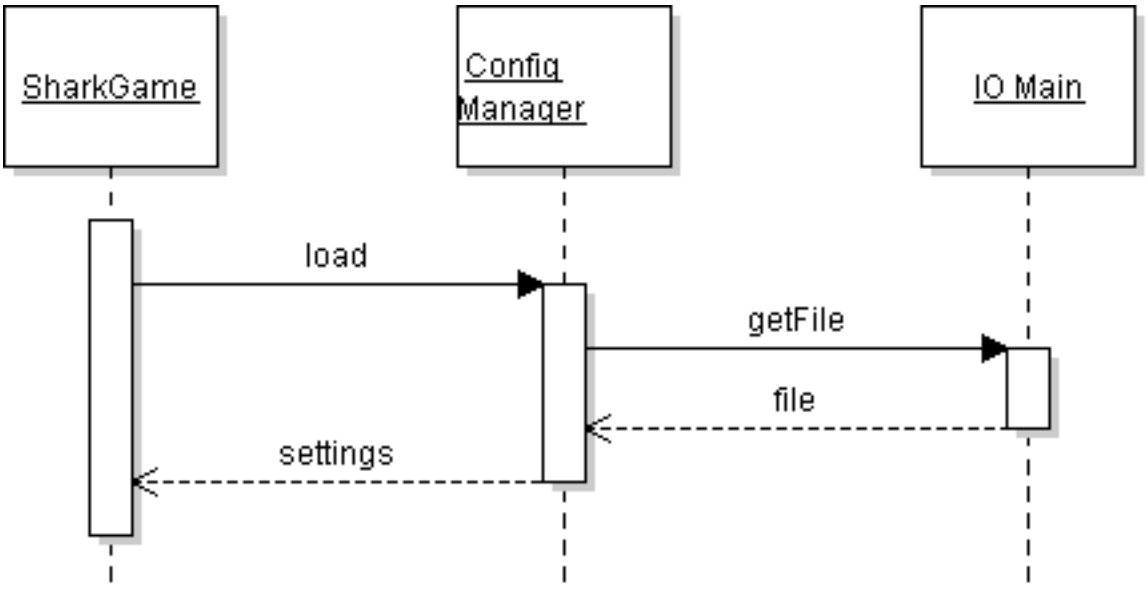
The file manager handles the I/O processes. The process will be directed by Java's Scanner implementation and FileWriter objects. The sound manager directs which sounds should be played and when. The Input Manager assists in accepting key input and passing those commands to the right places. The configuration manager could be considered a part of this module, but for our purposes, we have included it in the main module.

Implementation

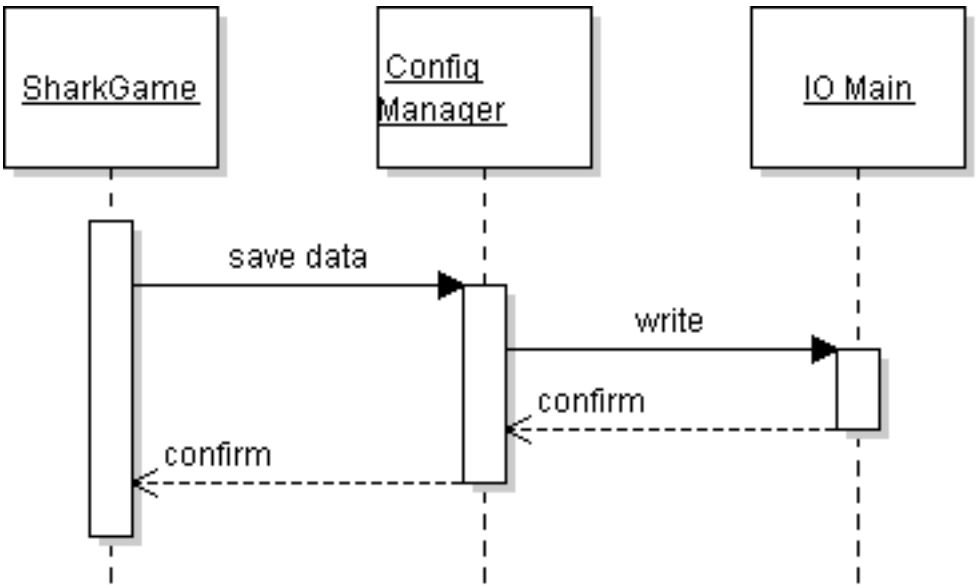
The collision manager makes simple comparisons based on the position and collision area of the objects passed. The file manager directs Java's file I/O objects for reading and writing files. The sound manager and input manager utilize the tools provided by Slick2D to handle which file is played on which channel. A Hash Map maps the sound effects to strings. The music plays on a distinct channel from the game effects. The effects manager handles extra effects that are not controlled by Slick2D directly.



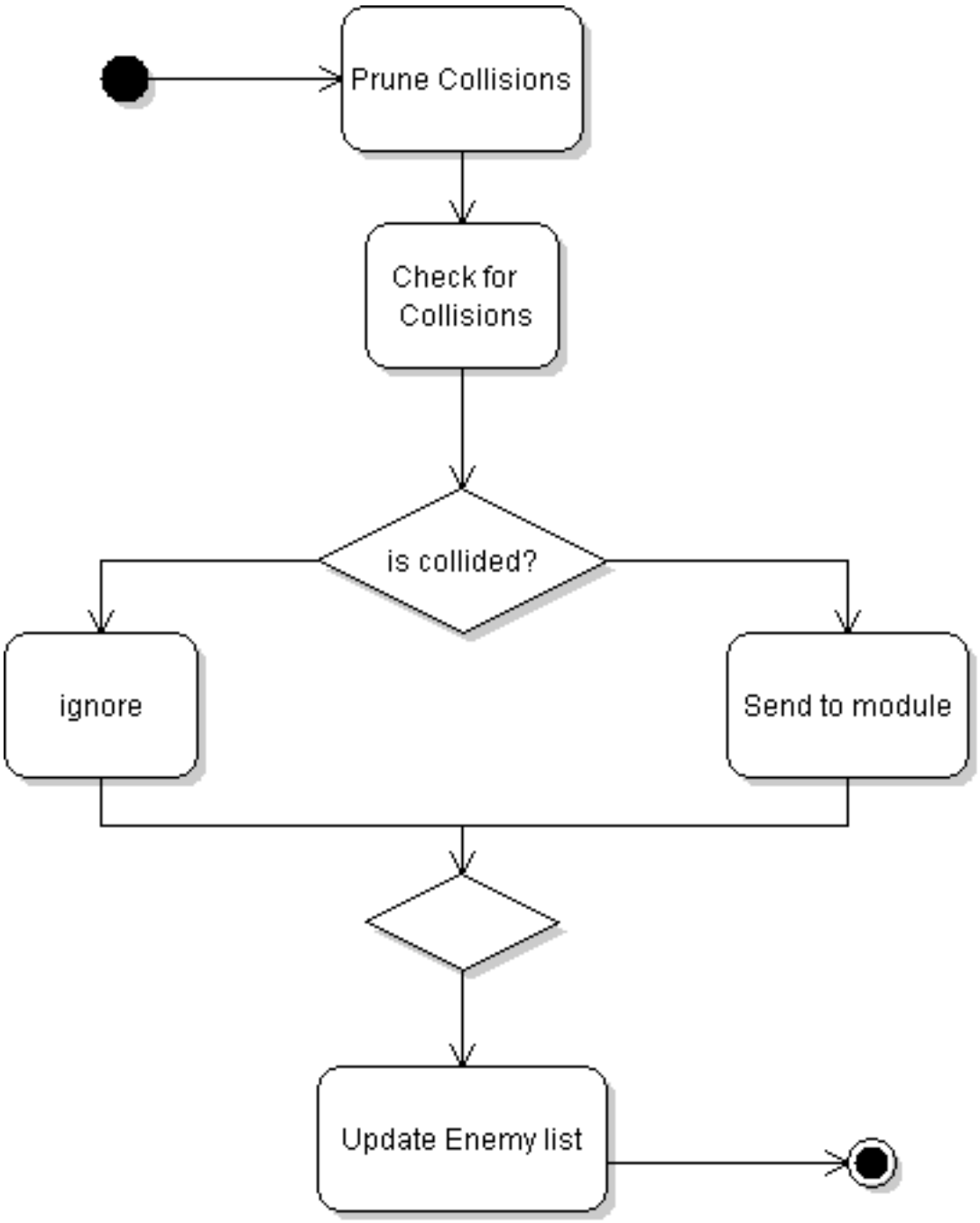
sound file loading



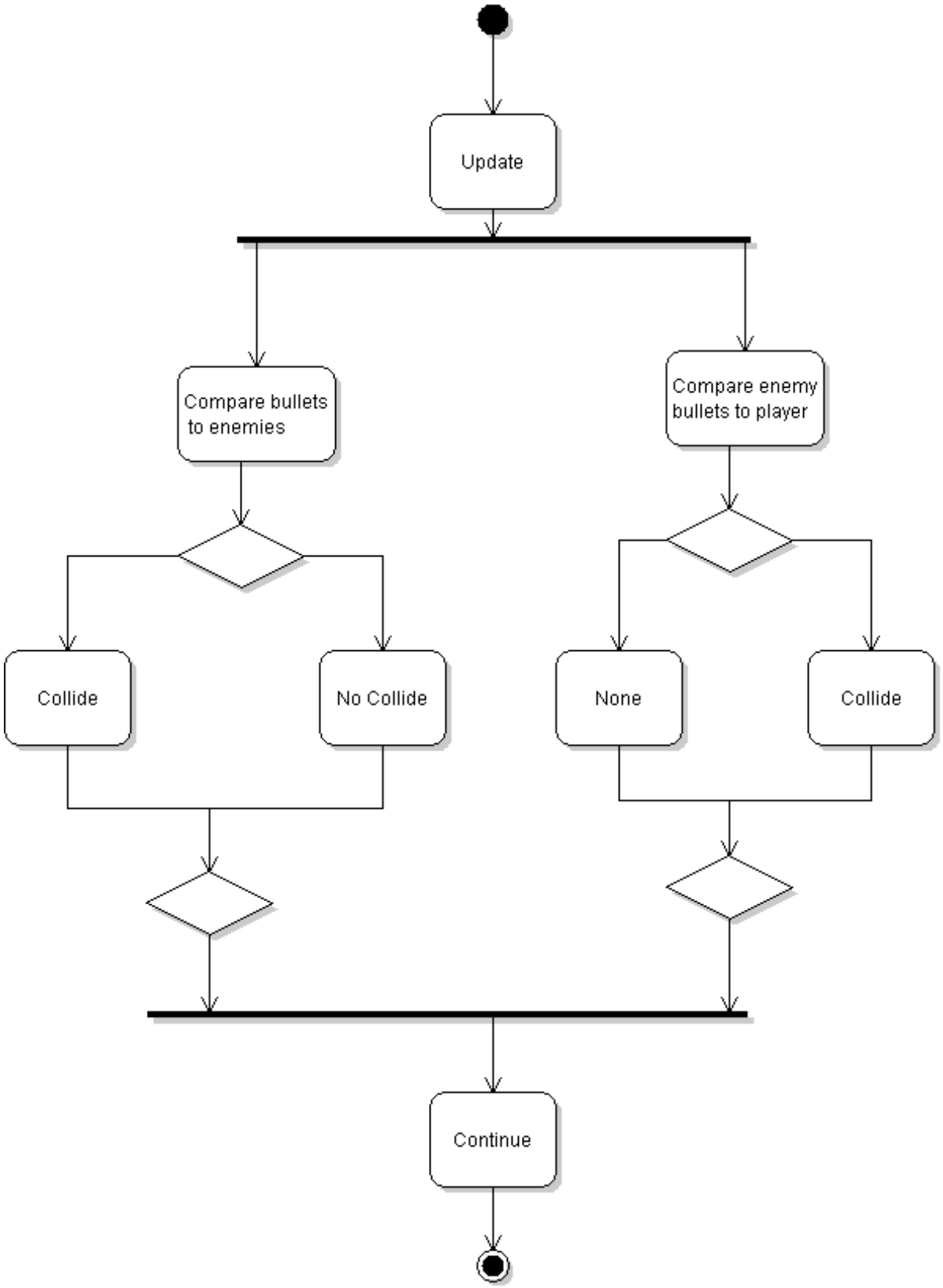
loading sequence



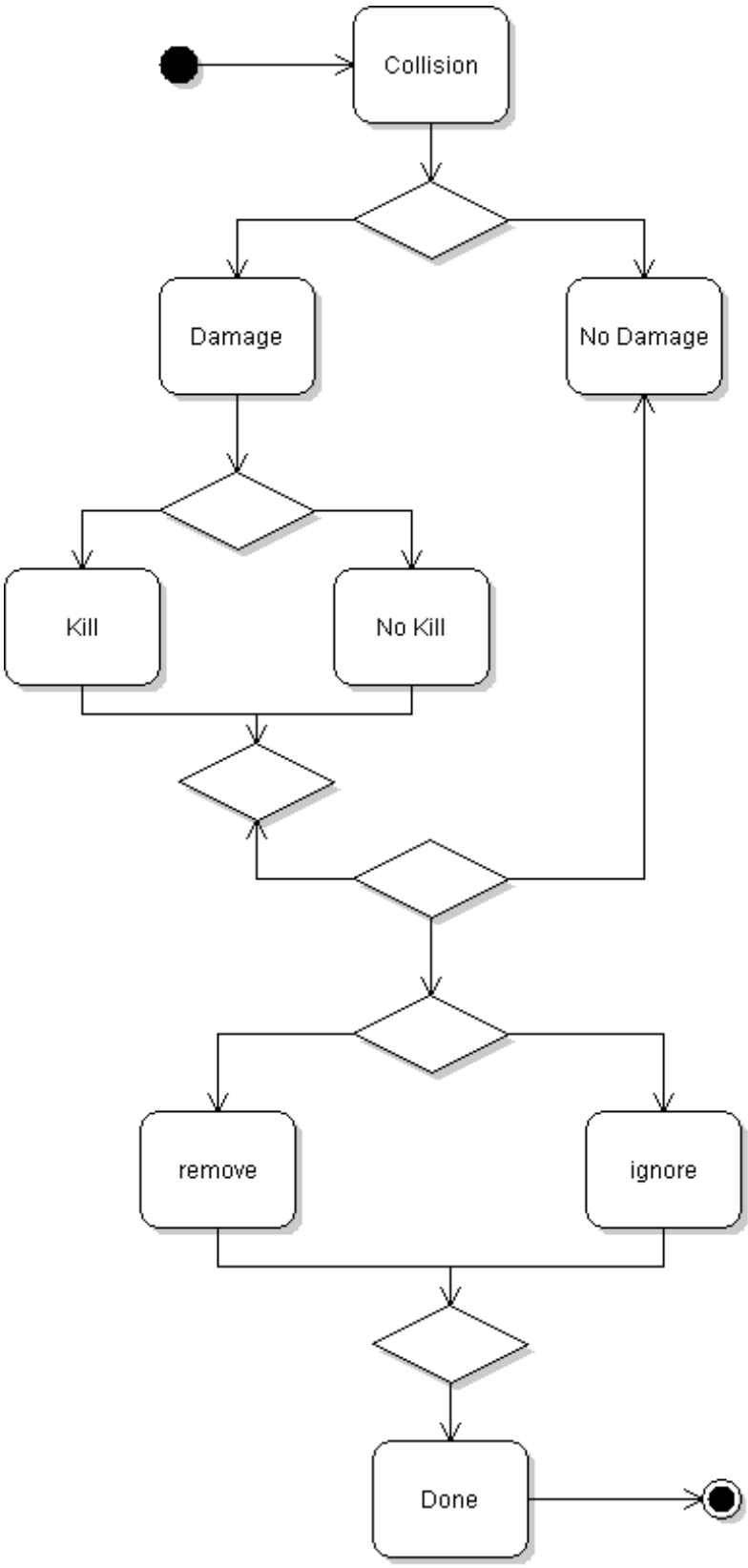
saving sequence



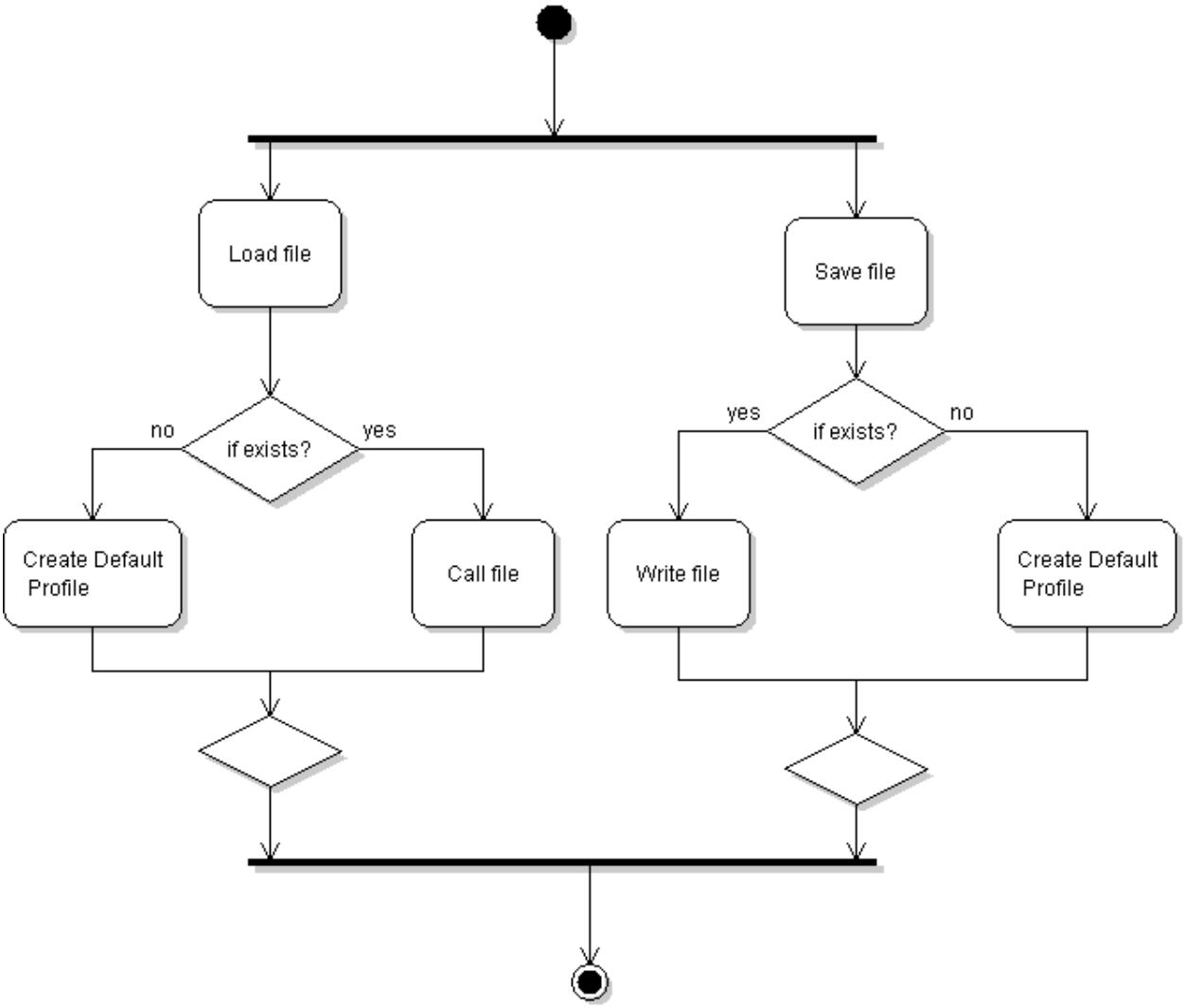
collision detection



collisions (2)



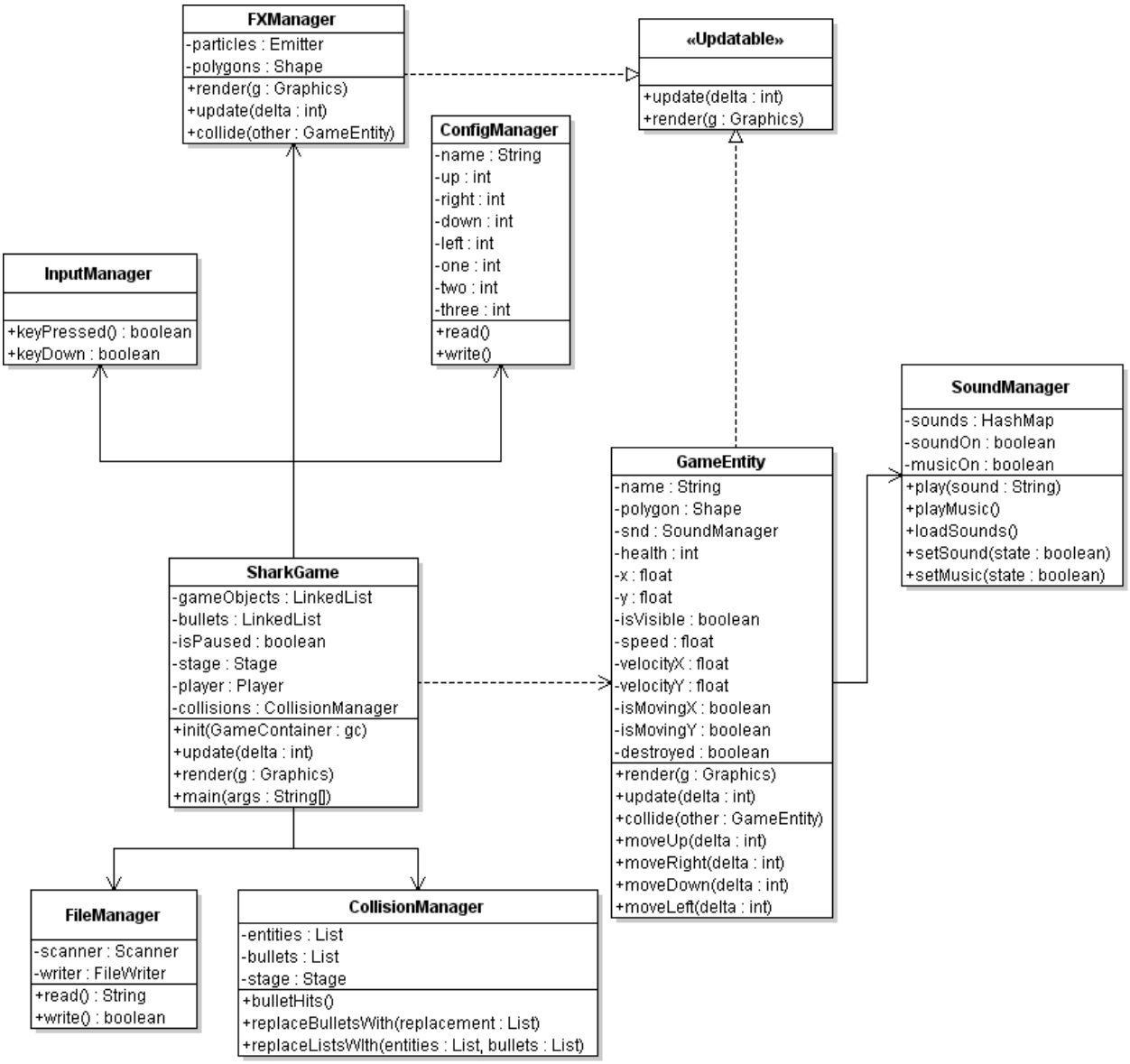
collision detection (3)


Load file
Save file
no
if exists?
yes
yes
if exists?
no
Create Default Profile
Call file
Write file
Create Default Profile

file IO

Design

The collision manager is passed pruned lists, as each pass runs a multiplicative number of comparisons. The sound manager simplifies calls to Slick2D's audio handling routines.



utility classes

Imports / Exports

The collision handling relies on the game entity module, and returns the results to the main module. The file handing relies on Java's Scanner and FileWriter classes. The

Sound Manager utilizes Slick2D utilities and sends files directly to output. The Input Manager also relies on Slick2D's interface.

Pre and Post Conditions

The only precondition is the state of the game. It must be running in order for the utilities to be useful. The utilities do not change state and thus there are no postconditions.

Input / Output

subpart	input	output
collision detection	N/A	N/A
file manager	data files	data files
sound manager	the sound file	audio
input manager	the keys pressed	N/A

inputs and outputs of each subpart

Subparts

The collision manager, file manager, sound manager, input manager, and potentially the configuration manager make up the utility module.

Concrete Implementation

```
// the collision manager makes the comparisons
public class CollisionManager {
   // lists that will be compared
   private List<GameEntity> entities;
   private List<Bullet> bullets;

   public CollisionManager(List entities, List bullets) {
      this.entities = entities;
      this.bullets = bullets;
   }

   public void bulletHits() {
      for (Bullet b : bullets) {
         for (GameEntity e : entities) {
            if (b.collidesWith(e)) {
               b.destroy();
               if (e instanceof Enemy)
                  ((Enemy)e).kill();
            }
         }
      }
   }

   // new bullets to compare
    public void replaceBulletsWith(List<Bullet> replacement) {
      bullets = replacement;
   }

   // new entities
   public void replaceListsWith(List entities, List bullets) {
      this.entities = entities;
      this.bullets = bullets;
   }
}
```

```
// example file manager
class FileManager {
   Scanner s;
   FileWriter f;

   // read a file
   load() {
      s = new Scanner(“config.shark”);
      // get info from file
      String name = s.nextLine();
      // ...
   }

   save() {
      String contents = // get contents from attributes
      f.write(contents, “config.shark”);
   }
}
```

```
// sound
class SoundManager {
   HashMap<String, String> files;

   // play a file
   void play(String fname) throws IOException {
      playAudio(files.get(fname));
   }

   void PlayAudio() {
      // audio methods
   }
}
```

```
// InputManager
   // each direction is compared individually
   // there is no hierarchy, because an object
   // could be moving in all four directions at once
   // while it is accellerating or decellerating
   if (keyIsPressed.(Input.KEY_UP))
      player.moveUp();
   if (keyIsPressed.(config.KEY_DOWN))
      player.moveDown();
   if (keyIsPressed.(config.KEY_LEFT))
      player.moveLeft();
   if (keyIsPressed.(config.KEY_RIGHT))
      player.moveRight();
   if (keyIsPressed.(config.FIRE))
      player.moveUp();
```

Side Effects

N/A

Miscellany

N/A

Procedural Design - User Interface Module

Abstract

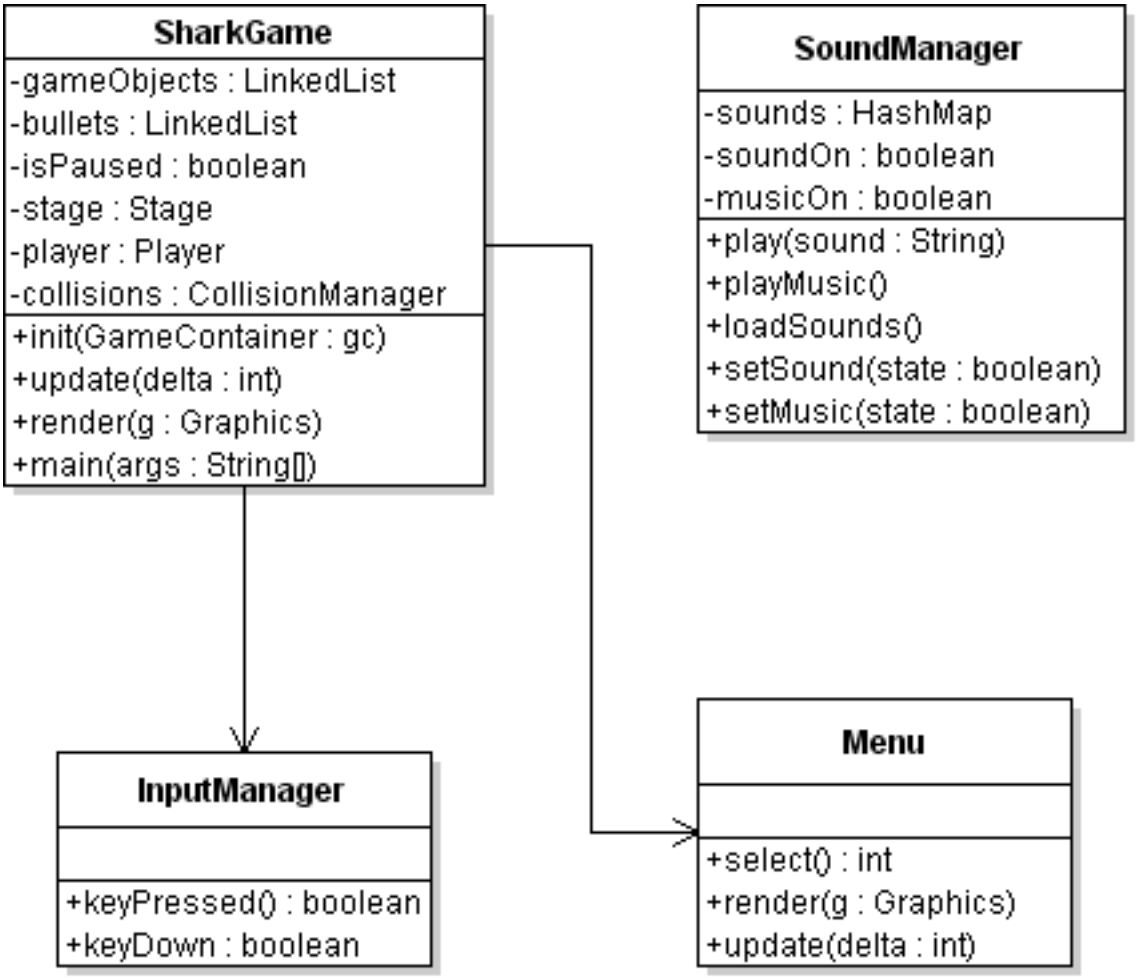
The user interface module is responsible for display relevant data about the user and displaying menus. The relevant data includes the information about the user such as weapon type and the health of the user. The menu is only visible when it is called and will pause the operation of the game until the resume is called.

Implementation

The implementation is simple as for displaying relevant data, it will render a box and display the needed variables. The menu implementation pauses the game then displays a box with control options such as resume, save, load, options, and exit. The menu presents choices for the player, and the specified result yields the next step.

Design

The menus resemble menus from games such as Doom and Quake, meaning that an important menu displays when the game is paused, in addition to the menus at startup.



IO classes

Imports / Exports

The only export between modules is when the menu is called, it must halt the operation of the game and when the save or is called, it must notify the file manager to save or load data.

Pre and Post Conditions

The only precondition is that the game is running. The postconditions vary, as the menu options can change the game state or other options.

Input / Output

The input comes directly from the user, and the output is sent to the main game for the next steps.

Subparts

The menus and the graphics make up the User Interface module.

Concrete Implementation

```
// menu
   if (isPaused)
      displayMenu();
   else
      // update game normally

   // …

   void displayMenu() {
      selection = choice();
      switch(selection) {
         case resume:   // continue game
         case end:      // end game
         case option:   // go to options
      }

  }
```

Side Effects

N/A

Miscellany

The actual input and output are actually part of the utility module. The user interface is entirely the representations of the menus and status to the player.

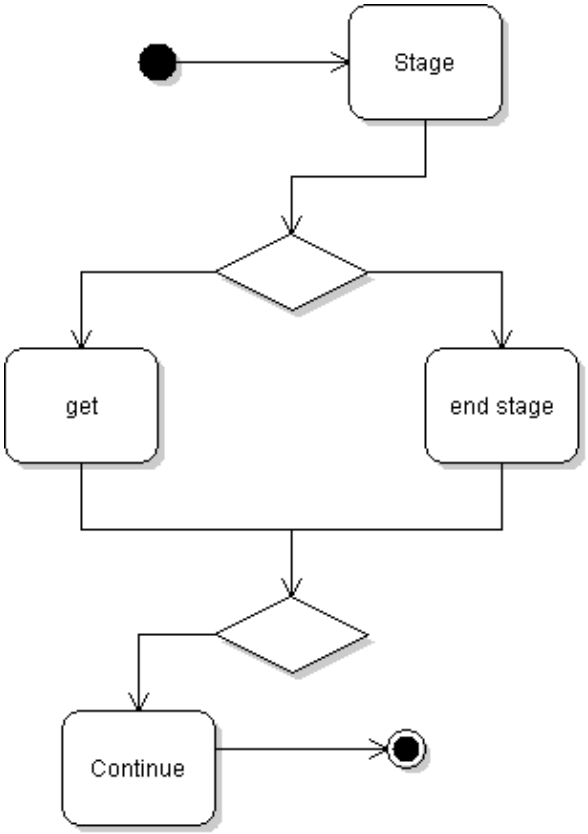
Procedural Design - World Module

Abstract

The remaining parts of the game make up the World module. The world module contains the elements of the application that simulate the game world. This includes drawing the background and giving space to the entities to move about in.

Implementation

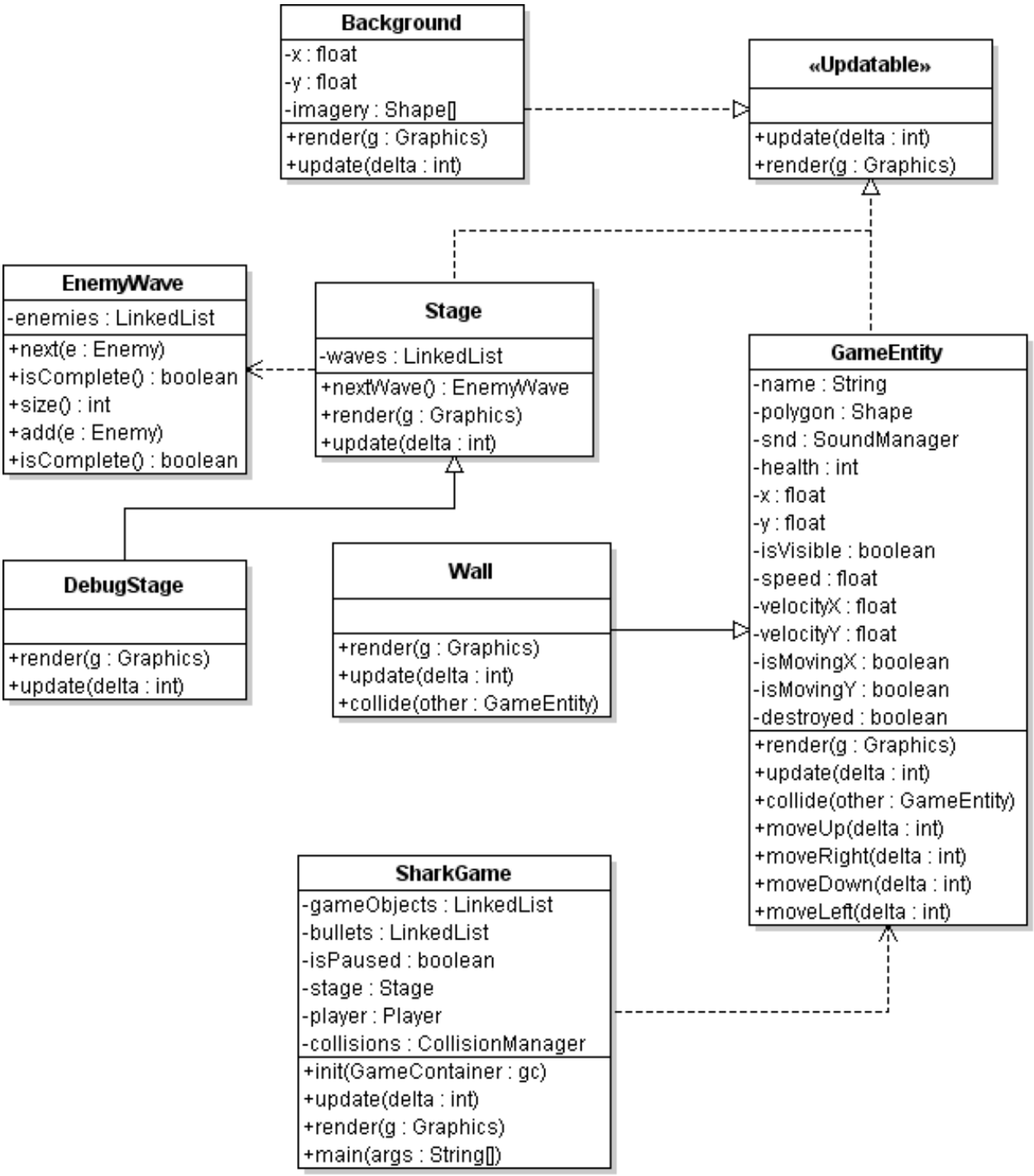
The stages are made up of lists of enemy waves. The waves themselves are lists of enemies. When called, the stage provides the next enemy wave. The background moves the entire time to simulate movement, but the background does not actually directly correspond to the player's movement. The background scrolls constantly. Each individual stage is hard coded with its enemy lists.



stage progression

Design

The lists contain the enemy and enemy wave references. The background is little more than a scrolling image.



world classes

Imports / Exports

N/A

Pre and Post Conditions

The game world is required for the stages to run, so there are no preconditions, but the postcondition is the running game stages.

Input / Output

N/A

Subparts

The background remains distinct from the stage objects, but the module works as a whole to provide the gameplay experience for the user.

Concrete Implementation

```
// the background class is its own variety of updatable object
public class Background implements Updatable {

	private float x;
	private float y;
	// the background images are textures repeated
	private Shape[] imagery;

	public Background() {
		imagery = new Shape[2];
		x = 0.0f;
		y = 120.0f;
		imagery[0] = new Rectangle(x, y, 800, 240);
		imagery[1] = new Rectangle(x, y - 600, 800, 300);
	}

	// render each image
	public void render(Graphics g) {
		for (Shape s : imagery) {
			g.setColor(Color.gray);
			ShapeRenderer.fill(s);
		}
	}

	// move the background images
	public void update(int delta) {
		for (Shape s : imagery) {
			s.setY((float) (s.getY() + 0.1 * delta));
			if (s.getY() > 600.0f)
				s.setY(s.getY() - 1200.0f);
		}
	}

}
```

```
public abstract class Stage implements Updatable {
   protected LinkedList<EnemyWave> waves;
   protected Background bg;

   // need to make wave and set up the background
   public Stage(String name, Background bg) {
      waves = // get waves
      this.bg = bg;
      stageName = name;
      ready = false;
   }

   // get the next wave
   public EnemyWave nextWave() {
      EnemyWave w = waves.removeFirst();
      nCurrent = w.size();
      return w;
   }

   // while not a game entity, stage still updates
   public void render(Graphics g) {
      bg.render(g);
   }

   public void update(int delta) {
      bg.update(delta);
   }

   //Check to see if there are any enemies left.
   public boolean isComplete() {
      if (ready)
         return waves.size() == 0;
      else
         return false;
   }
}
```

Side Effects

N/A

Miscellany

Each stage is individually hard coded rather than reading from input. Given further development time, the file I/O would be used here to generate the stages.