

Oogasalad API Design

Dianwen Li, Michael Han, In-Young Jo, Austin Lu, Cody Lieu, Dennis Park, Gary Sheng, Jimmy Fang, Jordan Ly, Lawrence Lin, Stephen Hughes, Kevin Do

Genre

The genre of our game is tower defense. As a subgenre of real-time strategy video games, a tower defense's gameplay style is markedly unique in its focus on "waves" of monsters attempting to bypass the player created obstacles to get rid of the player's "lives". The player must place structures, sometimes towers and other times inanimate obstacles, using resources in an attempt to hold off the flow of monsters and survive as long as possible. Another key characteristics of tower defense games is upgradable towers with numerous different qualities, such as ice and fire, with attacks that have different properties such as splash damage. With our design the author will be given lots of flexibility to decide what type of towers of be allowed in their game and what type of qualities they will have, with similar flexibility in regards to the contents of each wave.

There are numerous qualities that are unique to the tower defense genre that we will need to support with our design. First, the author must have the option of defining which style of tower defense to play. We believe the two core modes in the tower defense genre are a survival mode and a winnable mode. In the survival mode, as levels progress the health of the monsters also increase, making it essentially unwinnable with instead a goal of surviving as long as possible. In a winnable mode we would have a defined "waves" progression, with boss level/s as well as a final level. Our design would support both modes which we believe are core to the tower-defense genre, but also be flexible enough to easily incorporate other more non-standard modes as well.

Design Goals

The goal of this project's design is to be as general as possible in capturing all the possible tower defense games, while ensuring that the distinctive aspects of the genre are not lost. In our opinion, one of the most distinctive aspects of the tower defense genre is that the player can only place static structures, known as towers. Therefore we are making the assumption that towers are always static. Enemies are assumed to come in "waves", although many different behaviors can be simulated with this model (a constant stream of enemy is just consecutive waves, a boss enemy is a wave with only one enemy.)

We also constrain the game designer to one type of "resource", which we call money. Though the idea of multiple resource types can be seen to be an extension of the traditional tower defense game, we hold the opinion that such resource micromanagement is more the domain of

real-time strategy games, and detracts from the purity of tower defense games. The focus should be on building and placing towers, not micromanaging resources. Furthermore, the multiple resource types does not work well when the player does not have much control on his income: in a real-time strategy game, he can allocate his worker units to different resource types depending on his needs. In a tower defense game, the player lacks this liberty.

There is significant freedom for the game designer to push the envelopes of what a tower defense game has to consist of. In addition, we do not limit our designer to having to prescribe a predetermined “path” for his enemies to travel upon. Instead, we want to support a variant of the traditional tower defense game in which the player gets to place lots of towers to decide the path that the enemies will take. This adds a significant strategy element to the game: trying to decide which path is ideal to minimize enemies making it to the goal.

In terms of code design, it's very important to ensure that the modules that we design are as loosely coupled as possible. This is because, with a team of this size, coordination can become problematic. The more independent our modules are, the less communication can be an issue. We make the assumption that the game authoring environment only interacts with other parts through the game blueprint file, which it writes using the data team's API. The game player interacts only with the game engine. The game engine interacts with files through the game data methods.

Primary Classes and Methods

The overall architecture of the program consists of four modules: the game authoring environment, the game data, the game engine, and the game player. The overall hierarchy is shown in Appendix B.

Schemas

The Authoring Environment will create GameBlueprint object that contains different schemas. These schemas specify the design of different game elements, like Monsters and Towers. These GameBlueprint objects will be saved by the Data team. The engine will also use the data API to load in GameBlueprint objects and play the game.

Game Authoring Environment

User interacts with buttons or other components within the GUI, which activates controller (the various action listeners). The controller updates model information, while view displays changes in model state. The authoring environment simply creates a GameBlueprint, so there are no public methods that other teams need to worry about.

The authoring environment at a high level has a Model-View-Controller design. This MVC design allows for a separation of the model (overarching data) and the view via the controller module. The MVC was carefully designed with instances of the Proxy design pattern throughout, to reduce the coupling of classes and increase a clear, clean separation of concerns.

Core Architecture

- **VIEW MODULE**

- AuthoringView
 - talks to:
 - concrete EditorTabs
 - EnemyEditorTab
 - TowerEditorTab
 - TerrainEditorTab
 - ItemEditorTab (not put in use)
 - WaveEditorTab
 - GameSettingEditorTab
 - MainController, which gives and receives instructions to and from the AuthoringView
 - contains the finalize game button, which triggers the saving of the authored game
- EditorTab
 - talks to:
 - a concrete subclass of TabController, which allows data to flow through the controller module into the model module
 - hierarchy (**bold** means concrete class):
 - EditorTab, forces an implementation of saveTabData
 - ObjectEditorTab, allows for easy customization of some form of AbstractSchema
 - **EnemyEditorTab**
 - **TowerEditorTab**
 - **ItemEditorSubTabs**
 - **WaveEditorTab**
 - **GameSettingsEditorTab**
 - **TerrainEditorTab**

- **CONTROLLER MODULE**

- MainController
 - talks to:
 - concrete TabControllers
 - EnemyEditorController
 - TowerEditorController

- TerrainEditorController
 - ItemEditorController (not put in use)
 - WaveEditorController
 - GameSettingEditorController
- AuthoringView
- AuthorModel, which receives data from the MainController
 - kickstarts the authoring environment through main method
- TabController
 - talks to:
 - a corresponding concrete EditorTab
 - MainController, where view data is passed to
 -
- **MODEL MODULE**
 - AuthorModel
 - talks to:
 - nothing
 - contains an instance of GameBlueprint, which gets serialized through a DataHandler

Each tab pushes some concrete set of AbstractSchema components through the controller module into the model, and finally the GameBlueprint. Once an author confirms they are done, he clicks a submit button and the data gets passed to a DataHandler, a module which processes/serializes the GameBlueprint passed from the AuthorModel.

API Methods for Other Subteams

None.

Game Data

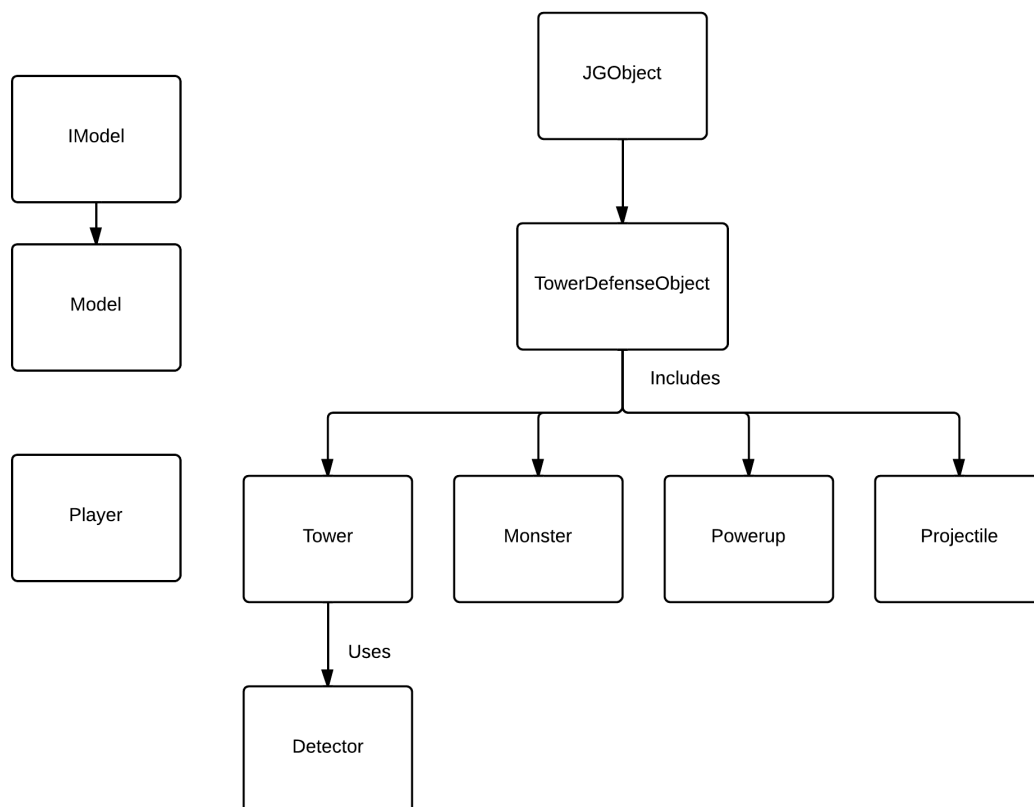
The game data team will offer the following API for the author and engine.

- DataHandler
 - public boolean saveState(GameState currentState, String filePath) throws IOException //Returns whether the object was successfully saved
 - public GameState loadState(String filePath) throws ClassNotFoundException, IOException
 - public boolean saveBlueprint(GameBlueprint blueprint, String filePath) //Returns whether the object was successfully saved
 - public GameBlueprint loadBlueprint(String filePath) throws ClassNotFoundException, IOException

When saving and loading blueprints, the DataHandler will also transparently zip up and unzip the resources folder and save it with the GameBlueprint. This means that any resources that a game designer refers to in the authoring environment will be guaranteed to exist when the engine loads a GameBlueprint.

The loading/saving of GameStates is separate from the GameBlueprints, since it represents a fundamentally different object. The GameBlueprint is the design of a game. The GameState represents where the player is in a certain game. Though the data team has finished writing the saveState() and loadState() methods, we ended up not using them because we did not have time to test them rigorously with the engine and player, and decided it would be better to leave out a finished but relatively untested feature than to put it in.

Game Engine



Player - keeps track of the stats of a player

- Money
- Lives/health

- Score

LevelManager - spawns and maintains state for waves, current level, and thus game progress

EnvironmentKnowledge - gives individual TDObjets ability to query aggregate information about the other objects on the field (e.g. the nearest monster to a tower, without direct access to the other objects.)

Model

- ITower[][] keeps track of towers on the field
- TDMap definition (e.g. lava field, grass field, sea)
- createTowerDefenseObject(Name objectName, Map<String, String> attributes)
 - NB: use of Class Registration and Reflection
- upgradeTowers() // called when the player clicks on a tower and upgrades it
- createTower() // called when the player chooses to build a specific type of tower at a specific location
- updateGame() // our doFrame method, which update the game states, e.g., increasing score

TDObjctFactory - does construction via reflection util and wrapping of TowerBehaviorDecorations

- tasked with creating objects such as monsters, towers, and powerups.

API Methods for Other Subteams

The API for the model is contained in the IModel class, which is replicated here for convenience.

```

public void addNewPlayer ();

public boolean placeTower (double x, double y, String towerName);

public boolean isTowerPresent (double x, double y);

public List<String> getUnitInfo(double x, double y);

public void checkAndRemoveTower (double x, double y);

public void loadGameBlueprint (String filePath) throws ClassNotFoundException,
IOException;

public void resetGameClock ();

```

```
public double getScore ();

public boolean isGameLost ();

public double getGameClock ();

public int getPlayerLives ();

public int getMoney ();

public boolean isGameWon();

public void setSurvivalMode(boolean survivalMode);

public void doSpawnActivity () throws MonsterCreationFailureException;

public void updateGame () throws MonsterCreationFailureException;

public boolean placeItem (String name, double x, double y);

public void checkCollisions ();

    public boolean upgradeTower (double x, double y) throws
TowerCreationFailureException;

public void decrementLives ();

public List<String> getPossibleTowers ();

public List<String> getPossibleItems ();

public void saveGame (String gameName) throws InvalidSavedGameException;

    public void loadSavedGame (String filename) throws
InvalidSavedGameException;

public String getTowerDescription(String towerName);
```

```
public String getItemDescription(String itemName);
```

```
public void annihilateMonsters();
```

Game Player

Screenshots can be found in Appendix A. The overall interface will be implemented in Swing. The buttons around the game window will be JButtons that call game engine methods when pressed. The Game Player makes up the View and Controller modules of the MVC pattern with the Game Engine's model class serving as the model module. The MainView, TDPlayerEngine, and ViewController classes along with the ITDPlayerEngine interface are contained in the main Player package and make up the Game Player. Main View creates a View Controller, ViewController controls and displays the GUI via the card layout, and TDPlayerEngine gets information from Model needed to play the game. TDPlayerEngine and Model communicate via the IModel interface. Reflection was used throughout player to simplify Swing code and reduce dependencies.

The player.dlc package contains the classes necessary for downloading games from an online repository. The player.panels package contain the various panels used in the ViewController class. GameInfoPanel and UnitInfoPanel implement the observer pattern to get information from TDPlayerEngine(observable). The player.util package contains util classes used by panels and ViewController.

Layout of cards in game player (all in JFrame)

- toolbar: allow selection of languages
 - Languages
- card: welcomeCard
 - JButtons
 - help: goes to helpCard
 - start game: goes to gameCard
 - credits: goes to creditsCard
 - highscore: goes to highScoreCard
 - options: goes to optionsCard
 - quit: exits game
- card: gameCard
 - JButtons:
 - main menu: goes to welcomeCard
 - play/pause: plays/pauses game
 - speed up: speeds up game
 - slow down: slows down game
 - quit: exits game
 - sound on/off: toggles sound

- add tower: allows addition of towers
 - JComboBox:
 - add tower: list contains all possible towers
 - JLabels to display game information:
 - score
 - lives left
 - money
 - game clock
 - JTextArea to display tower/monster information
 - tower
 - cost
 - buildup time
 - health
 - upgrade tower information
 - damage
 - range
 - monster
 - x-coordinate
 - y-coordinate
 - money value
 - health
 - move speed
 - JPanel for gameplay:
 - TDPlayerEngine
- card: highScoreCard
 - JButtons:
 - main menu: goes to welcomeCard
 - JPanel to take in text:
 - enter name for high score
- card: creditsCard
 - JButton:
 - main menu: goes to welcomeCard
 - JTextArea
 - displays credits
- card: helpCard
 - JButton:
 - main menu: goes to welcomeCard
 - JTextArea
 - displays basic instructions
- card: optionsCard
 - JButton:
 - main menu: goes to welcomeCard
 - JCheckBox

- turn music on or off

API Methods for Other Subteams

None.

Example Code / Two Types of Tower Defense

Survival/Infinite Mode	Winnable/Finite Mode
The goal of this mode is to survive as long as possible and attain as high a score as possible. Each subsequent level will become progressively harder. In this mode, enemies (which will have a specified schedule of appearance (e.g. every two levels)) will increase in health as levels progress. Since the difficulty of the waves of enemies is algorithmic, this will be manageable for the game creator to handle and for the game engine to process simply.	When the timer is not 0 and a specified quota of enemies has been exhausted, model state is updated to a harder level. Levels could be made harder by increasing enemy's quantity, speed, health, etc. Eventually the player will have to beat a boss level or end of game mode. Highscores will also be recorded for this game mode.

Alternative Designs

For TowerDefenseObjects, we considered:

1. Having an all-encompassing Projectile class where the authoring environment can predefine types for the user by filling in some parameters.
2. We subclass Projectile in the game engine, and all possible types

We decided to leverage JGame, over the alternative of building our engine entirely from scratch, for the following reasons:

- High potential reuse of animation and display functionality such as colors, drawing lines, drawing objects, loading/defining images and rotations
- JGObject functionality, frame-by-frame actions, and hit detection
- A tiles system and canvas that fits our needs
- Familiarity with all of the above, from past projects
- We can use as much or as little as necessary, and add our own modifications to JGame as it is open source.

Between Engine and Authoring Environment, we considered three alternatives:

1. Authoring Environment calls get Engine's getAttributes method to dynamically get a list of valid attributes that the game designer change specify.
2. Authoring Environment and Engine each maintains their own list of valid attributes so that no communication is necessary between the two and Authoring Environment can have greater liberty with its GUI design.
3. The Schema for each type of object are known to both sides, and contains all of the attributes needed for constructing that object.

We decided on the 3rd alternative so that duplicate lists of attributes would not be needed, and the schema objects took advantage of code reuse.

In the Engine design for Towers, the choices included having an inheritance hierarchy, and a decorator pattern. The decorator pattern was chosen for its ability to mix and match tower behaviors, for example, towers that could shoot and/or farm money, which would otherwise require a complicated inheritance hierarchy since there is necessarily "is a" relationships between all behaviors.

Roles

Game Authoring Environment Subteam

Roles include design of the overarching MVC, and handling everything involved in specific object creation (e.g. EnemyEditorTab and Controller). Cody Lieu, Stephen Hughes, Dennis Park, and Gary Sheng will be working on the Game Authoring Environment.

Game Data Subteam

Roles include designing a class in which to store data (a DataHandler), developing a flexible way to serialize/deserialize the data into/from a file, managing resources associated with a created game (zipping and unzipping resources), and sorting out other data-handling/validation issues. Jimmy and In-Young worked on game data together with some design input from Kevin Do.

Game Engine Subteam

Roles include: creating TowerDefenseObjects, creating the Model, creating the Player Austin designed Towers/TowerDecorations and the LevelManager, Jordan was responsible for path finding and tiles/maps, while Dan and Lawrence worked on items and extending TDOObjects.

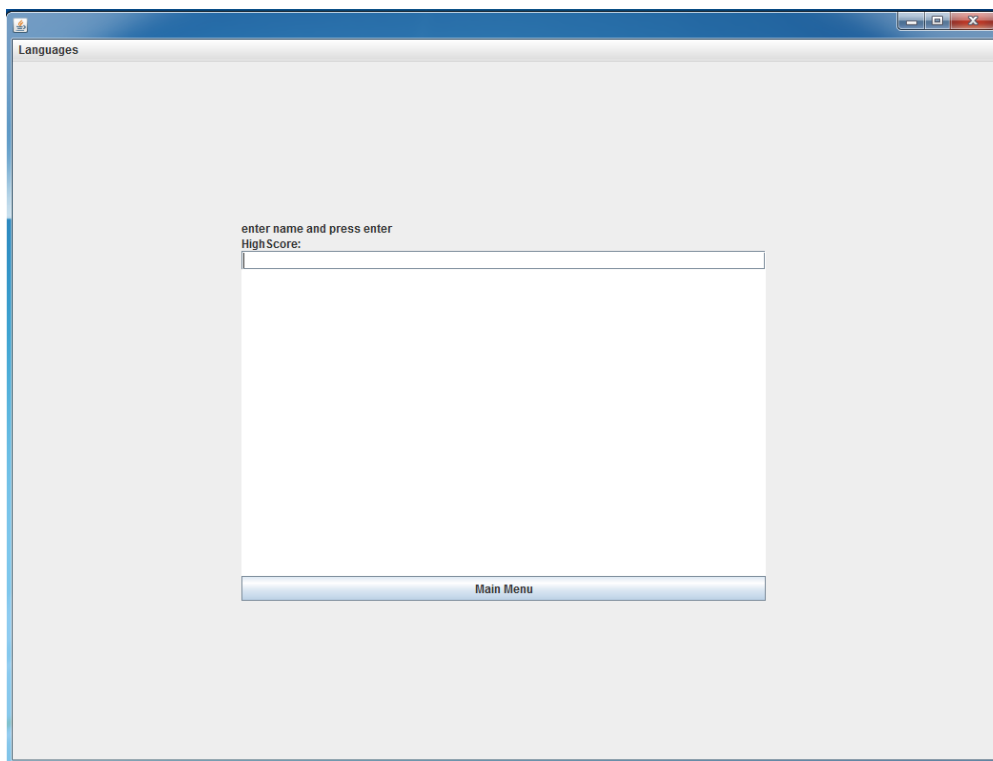
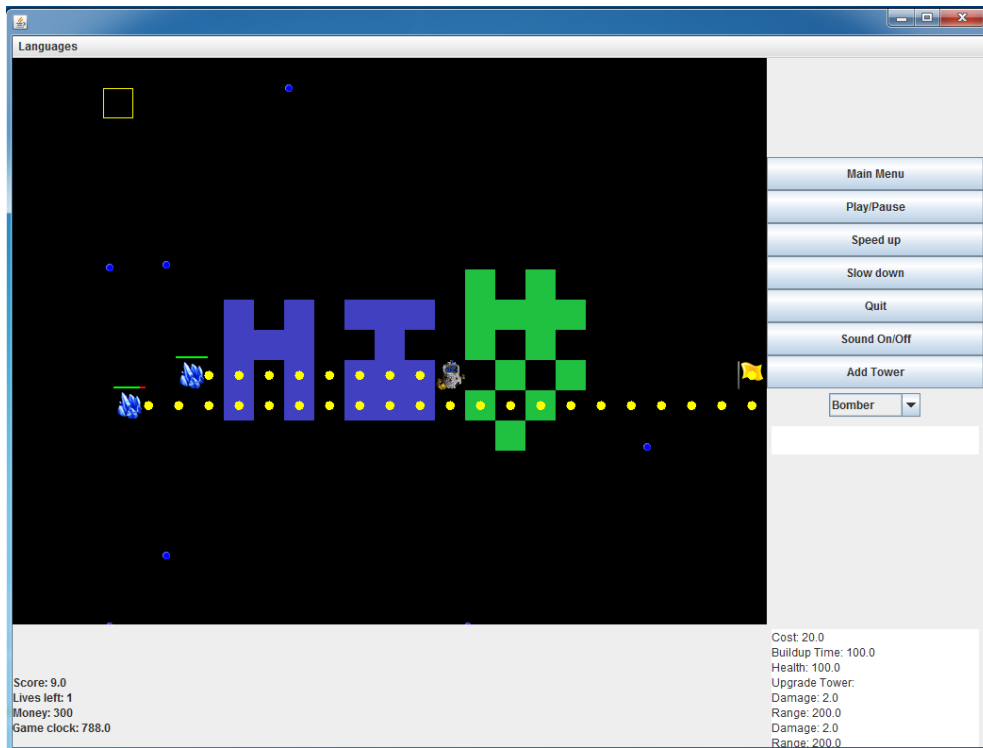
Game Player Subteam

Michael Han and Kevin Do will pair-program the entire game player.

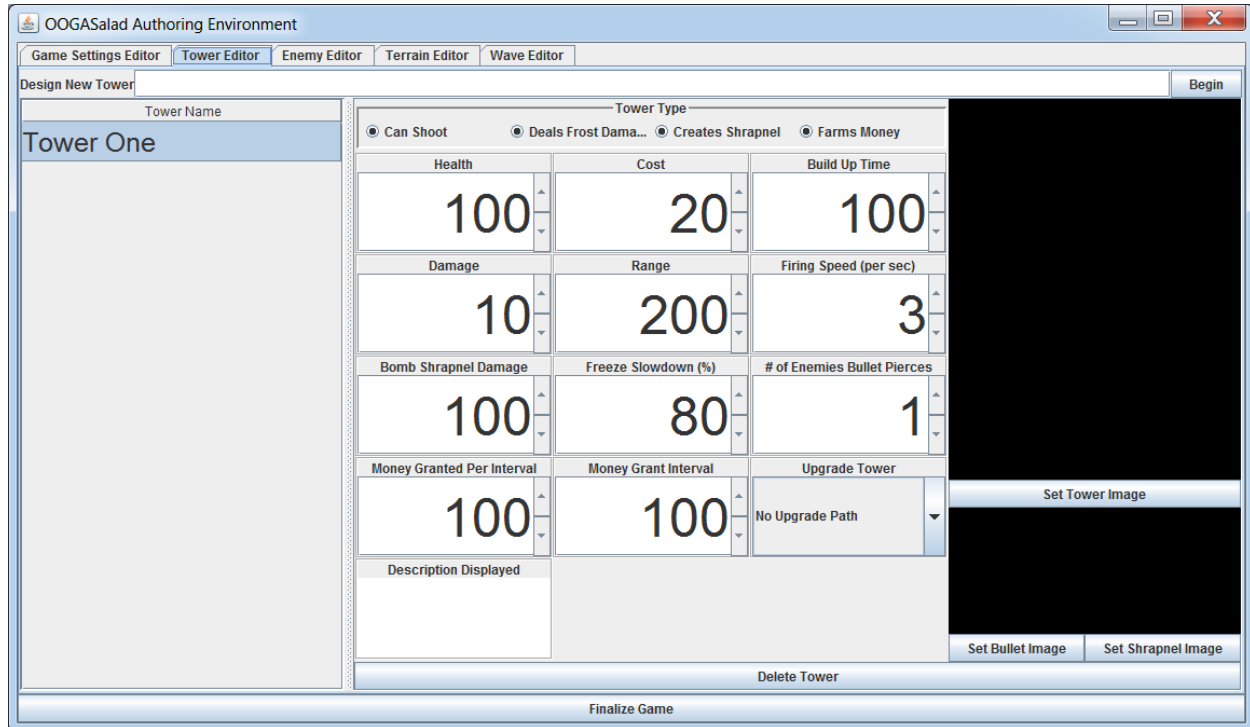
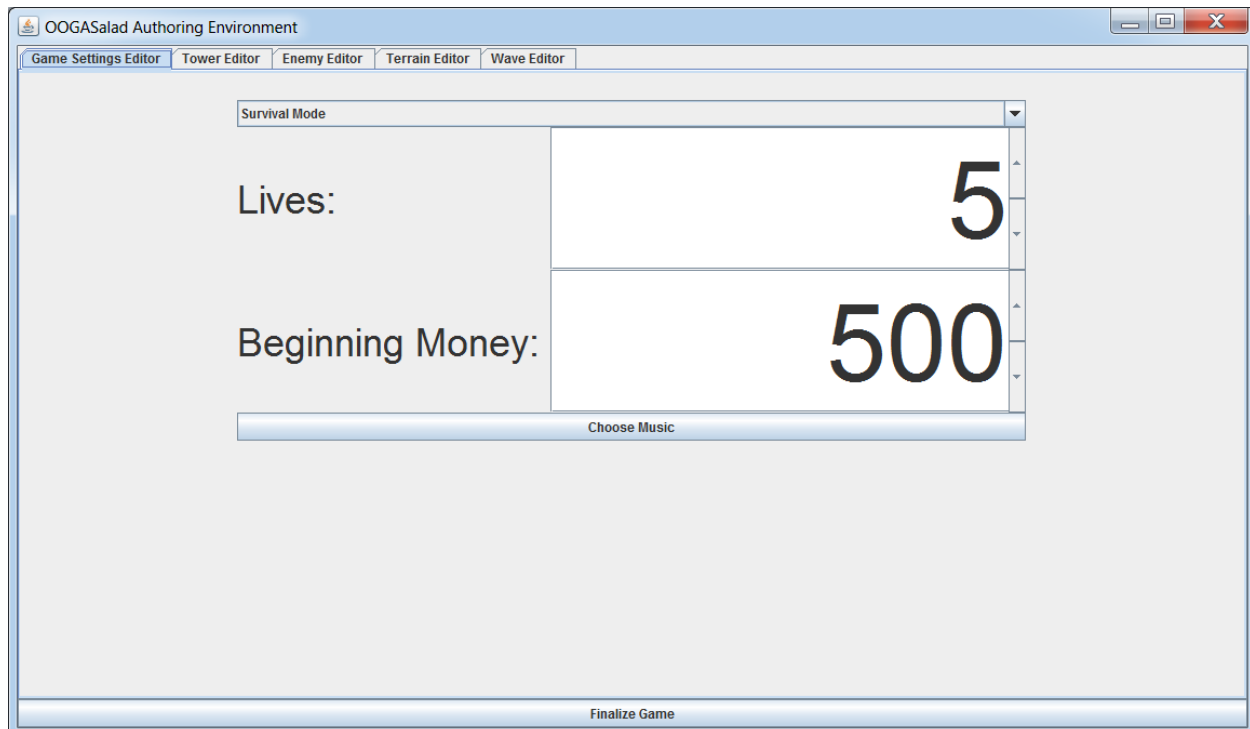
Appendix A: Screenshots

Game Player






Game Authoring Environment



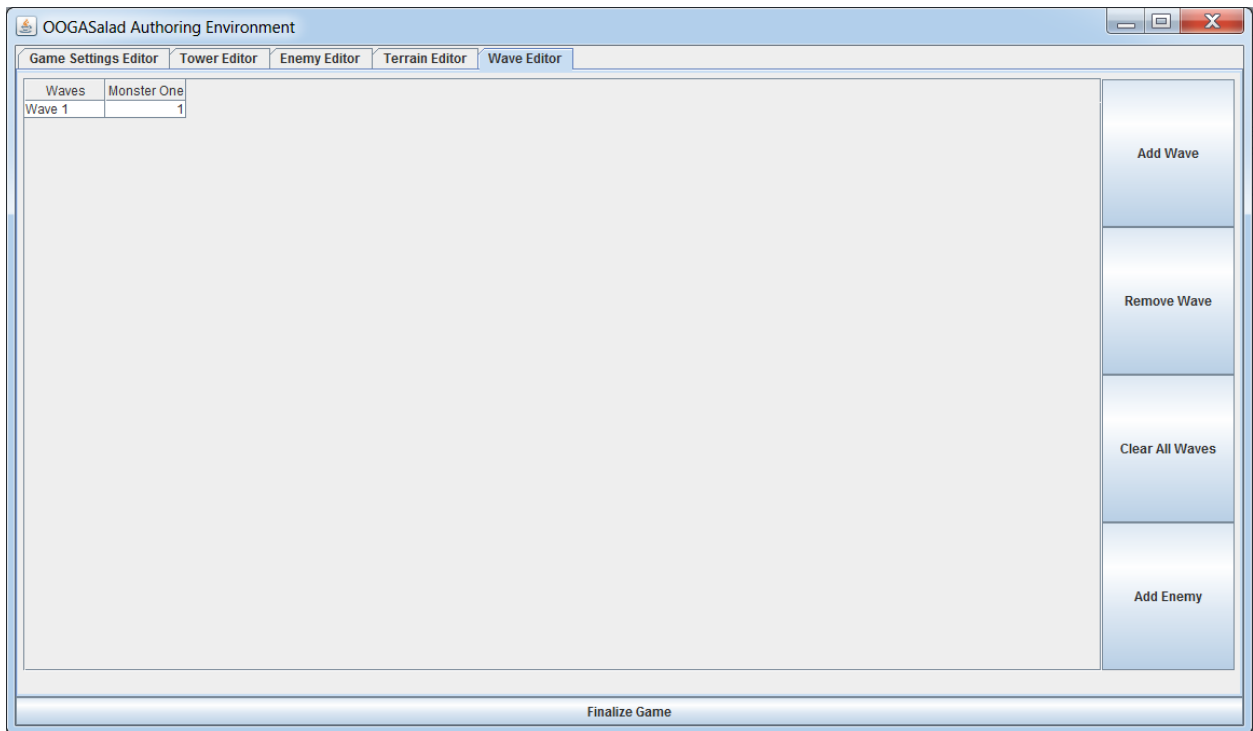
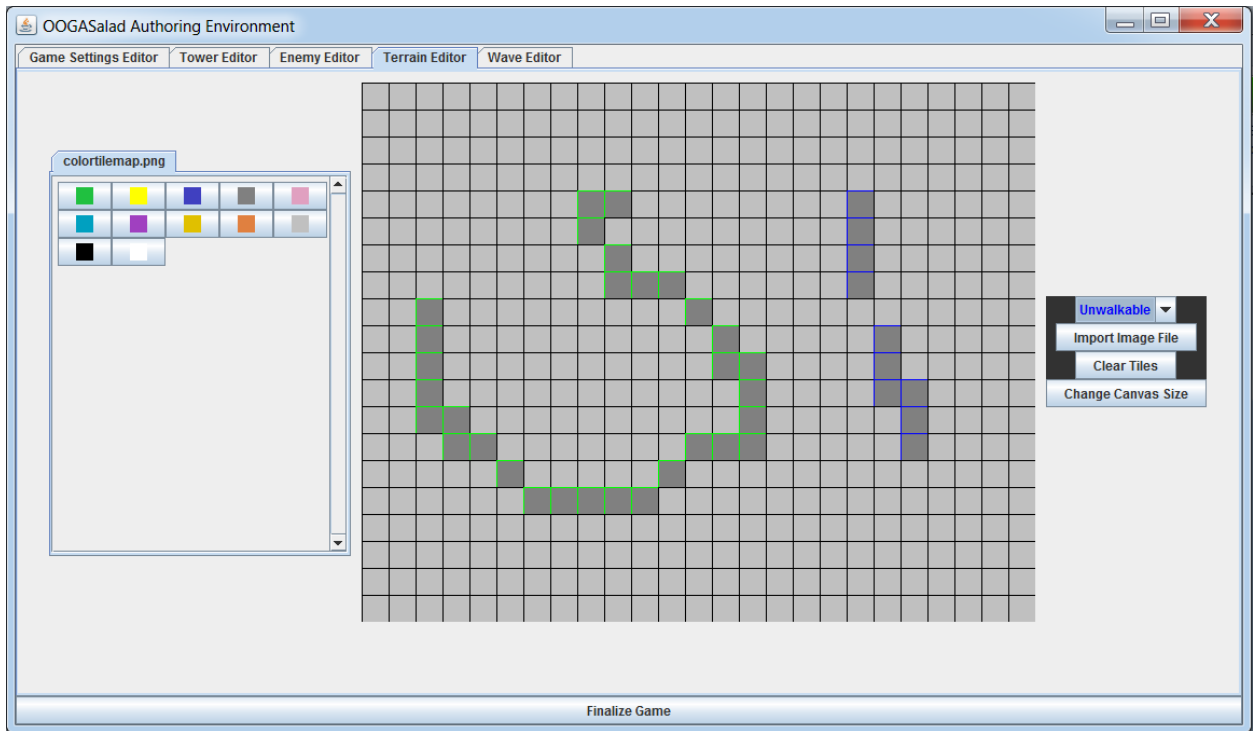
OOGASalad Authoring Environment

Game Settings Editor Tower Editor **Enemy Editor** Terrain Editor Wave Editor

Design New Monster Begin

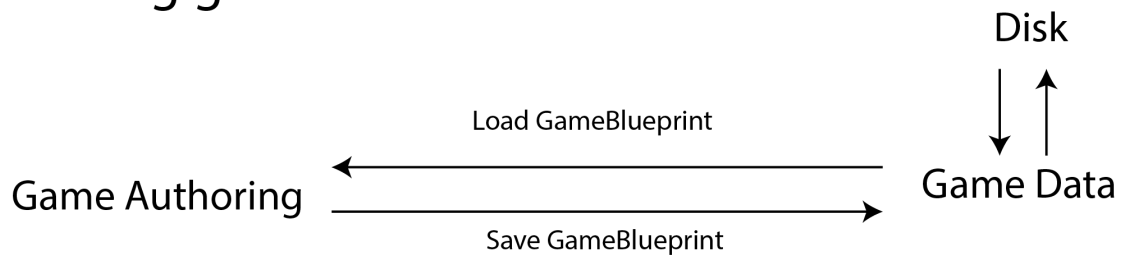
Monster Name	Health	Speed
Monster One	30	1
	Attack Damage	Bounty Reward
	10	50
	Tile Size	Flying Or Ground Type
	<input checked="" type="radio"/> Small <input type="radio"/> Medium <input type="radio"/> Large	<input type="radio"/> Flying <input checked="" type="radio"/> Ground
Monsters to Spawn Upon Death		
Monster To Spawn		
No Resurrection Path		
How many		
0		
Delete Monster		

Finalize Game



Appendix B

Authoring game



Playing game

