

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.

What are we considering a tower defense:

- Specification of either SURVIVAL or WINNABLE based
 - if Survival Mode specified, only enemy health, not speed or other factors, gets changed as levels progress (if someone objects, discuss at meeting)
 - In this case, you will specify the health algorithm of a tower
 - The object of Survival Mode is to survive as long as possible and get as high a score as possible. The Object of Winnable Mode is to progress through progressively harder levels and eventually beat a boss level.
- The game is a 15x10 grid of 32x32 squares
- Enemies cannot fly
- A terrain object will block both tower building and enemy passing or block neither, but not exclusively either
- Towers can be upgraded
- There is a global tower sell price fraction set by game designer

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 one very big wave, 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. For example, traditionally the enemies march on without attacking the towers (i.e. the player never has to worry about towers getting destroyed.) We challenge that assumption. 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.

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 JSON file. 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. There is also a module of common classes. The overall hierarchy is shown in Appendix B.

Common

See specification in the Model section of Game Authoring Environment for high level understanding of objects.

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 JSON (as of now), so there are no public methods that other teams need to worry about.

- View modules (only talks to Controller, think buttons, forms, frames, panels)
 - Splash screen builder
 - Canvas
 - Tile manager
 - Schema builder view module (panels)
 - Tower builder view module (panels)
 - Enemy builder view module (panels)
 - Map builder view module (panels)
 - maybe this view contains the Canvas
 - Menu items
 - Menubar
- Controller (communicates between View and Model, think actionlisteners)
 - Library of action listeners update state of model
 - one for each view module
- Model module (only talks to Controller, think raw data)
 - Model
 - fields
 - myEnemyList
 - myTowerList
 - myMap
 - this contains environmentalElements
 - mySchema
 - Tower: Not a JGObject like the Tower in the engine. It just serves a data wrapper for our attributes.
 - fields
 - myCost
 - myDimension (in terms of discrete tiles)
 - myFireRate

- myGraphic
 - myFireAnimation
 - myCollisionGraphic
 - myImagePath
 - etc.
- EnvironmentalElement
 - fields
 - blocksTowerAndEnemy
- Enemy
 - fields
 - levelToFirstAppear
 - numLevelsBetweenAppearances
 - numMinionsToAppear
 - myKillReward
 - myHealthFormula
 - will change based on level if on survival mode
 - will stay static if not on survival mode
 - myMoveSpeed
 - myAttackDamage
 - myGraphic
 - myFleshGraphic
- Map (the data about where trees are and stuff)
- GameSchema
 - fields
 - myNumLevels
 - myTowerSellPercentage
 - myWinLevel
 - hasEnding
- abstract Actor class, implements draggable and drawable interfaces
- Utilities
 - Grid cell class
 - Direction

Game Data

A data bundle class will be created so that there can be one object type that stores the states of all objects that are either defined/set/updated by the authoring environment or the game engine. (We'll do this by having in the constructor of the data bundle get methods for everything which will represent the game state, ex: list of towers, enemies, player health) When first authoring a game, a new data bundle will be created by the authoring environment and this will store all the information input by the author to establish the conditions for the game. This data bundle will be

passed to the Data module of the project, which will use a JSON serializer to create/update the JSON file associated with the game.

The Data module will create JSON files for new games and load/save variable states from/to existing JSON files. The Data module will interact with the authoring environment and game engine

Here are some of the modules that Game Data will involve:

- Data
 - JSON Serializer
 - Convert data bundle information into a JSON saved in the format of the example JSON file below.
 - JSON Parser
 - DataBundle Class
 - DataBundle()
 - This constructor will initialize the data structures needed to make references to all of the data required to create a game (tower lists, enemy lists, etc.)
 - saveDataBundle()
 - This method will use the JSON serializer to save the information of the data bundle into a JSON
 - loadDataBundle(String fileName)
 - This method will use the JSON parser to fill the data bundle's information with the info gathered from an existing JSON file.
 - getTowers()
 - getEnemies()
 - getEnvironment()
 - getSchema()

Example JSON file:

```
{ "gameName": "SuperAwesomeTowerDefenseGame",  
  "model": {  
    "myEnemyList": {  
      "e1", "e2"  
    },  
    "myTowerList": {  
      "t1", "t2"  
    },  
    "myMap": {  
      "environmentalElement1", "environmentalElement2"  
    },  
  },  
}
```

```

    "mySchema": {
        "s1", "s2"
    }
},
"tower": {
    "t1": {
        "myCost": 10,
        "myDimensionX": 1,
        "myDimensionY": 1,
        "myFireRate": 10,
        "myGraphic": "/resources/towers/t1.jpg",
        "myFireAnimation": "/resources/fires/fire1.jpg"
        "myCollisionGraphic": "/resources/collisions/col1.gif"
    },
    "t2": {
        "myCost": 20,
        "myDimensionX": 2,
        "myDimensionY": 2,
        "myFireRate": 20,
        "myGraphic": "/resources/towers/t2.jpg",
        "myFireAnimation": "/resources/fires/fire2.jpg"
        "myCollisionGraphic": "/resources/collisions/col2.gif"
    }
},
"environmentalElement": {
    "environmentalElement1": {
        "posx": 10,
        "posy": 10,
        "myGraphic": "/resources/environmental/environmentalElement1.jpg",
        "otherInfo": 10
    },
    "environmentalElement2": {
        "posx": 20,
        "posy": 20,
        "myGraphic": "/resources/environmental/environmentalElement2.jpg",
        "otherInfo": 20
    }
},
"enemy": {
    "e1": {
        "levelToFirstAppear": 1,
        "numLevelsBetweenAppearances": 1,
        "numMinionsToAppear": 10,

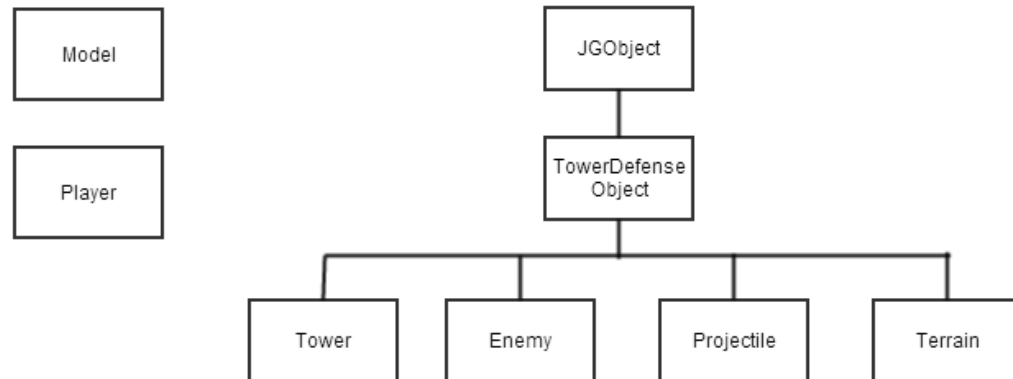
```

```

        "myKillReward": 10,
        "myHealthFormula": 10,
        "myMoveSpeed": 10,
        "myGraphic": "/resources/environmental/e1.jpg",
        "myFleshGraphic": "/resources/flesh/flesh1.jpg"
    },
    "e2": {
        "levelToFirstAppear": 2,
        "numLevelsBetweenAppearances": 1,
        "numMinionsToAppear": 20,
        "myKillReward": 20,
        "myHealthFormula": 20,
        "myMoveSpeed": 20,
        "myGraphic": "/resources/environmental/e2.jpg",
        "myFleshGraphic": "/resources/flesh/flesh2.jpg"
    }
},
"schema": {
    "s1": {
        "myNumLevels": 1
        "myTowerSellPercentage": 10
        "myWinLevel": 1
        "hasEnding": 1
    },
    "s2": {
        "myNumLevels": 2
        "myTowerSellPercentage": 20
        "myWinLevel": 2
        "hasEnding": 0
    }
}
}

```

Game Engine



Player class:

- Money
- Lives/health
- Score
- Skills (e.g. tower buildup rate)

Model

- Levels
- Stages/Terrain (e.g. lava field, grass field, sea)

// public methods for Authoring Environment

- `List<String> getTowerAttributes()` // return a string of different valid tower types
- `List<String> getEnemyAttributes()` // return a string of different valid enemy types
- `List<String> getProjectileAttributes()` // return a string of different valid projectile types
- `List<String> getTerrainAttributes()` // return a string of different valid terrain types
- `createTowerDefenseObject(Name objectName, Map<String, String> attributes)`
 - NB: use of Class Registration and Reflection

// public methods for Player

- `loadState(Filepath fp)` //load in the JSON file created by authoring environment
- `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
- `update()` // update the game states, e.g., increasing score

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.

All in JFrame

- toolbar in jframe
 - load game
 - preferences
 - reset high scores, etc
- card: main screen
 - buttons
 - start game
 - continue/load previous games
 - help
 - options
 - credits
- card: transition from main screen to ingame(difficulty level, other settings)
 - buttons
 - difficulty setting
- card: ingame
 - buttons:
 - main menu
 - play/resume
 - pause
 - speed up
 - save
 - quit
 - sell
 - drop down list:
 - add tower: list contains all possible towers and prices
 - upgrade: list of upgradeable attributes based on tower clicked
 - jpanel to display text:
 - info about health, resources, wave
 - info on the object clicked
 - jpanel for gameplay:
 - main game
- card: endgame/highscore
 - buttons:
 - main menu
 - jpanel to take in text:
 - enter name for high score

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 two 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.

Both designs have merits and perils and we decided to use a design that's somewhere in between: the Engine will pass a list of attributes in the form of a List<String> and the Authoring Environment can specify special GUI functions when it comes to special attributes (e.g., if the

attribute is “range”, display it together with the tower so that the designer can see the effect as he/she changes the range; if the attribute is “image”, use a JFileChooser).

Roles

Game Authoring Environment Subteam

Roles will include who works on View/Controller, and Model. Cody Lieu, Stephen Hughes, Dennis Park, and Gary Sheng will be working on the Game Authoring Environment.

Game Data Subteam

Jimmy Fang and Inyoung Jo will pair-program a JSON Serializer and JSON parser to save/load game state.

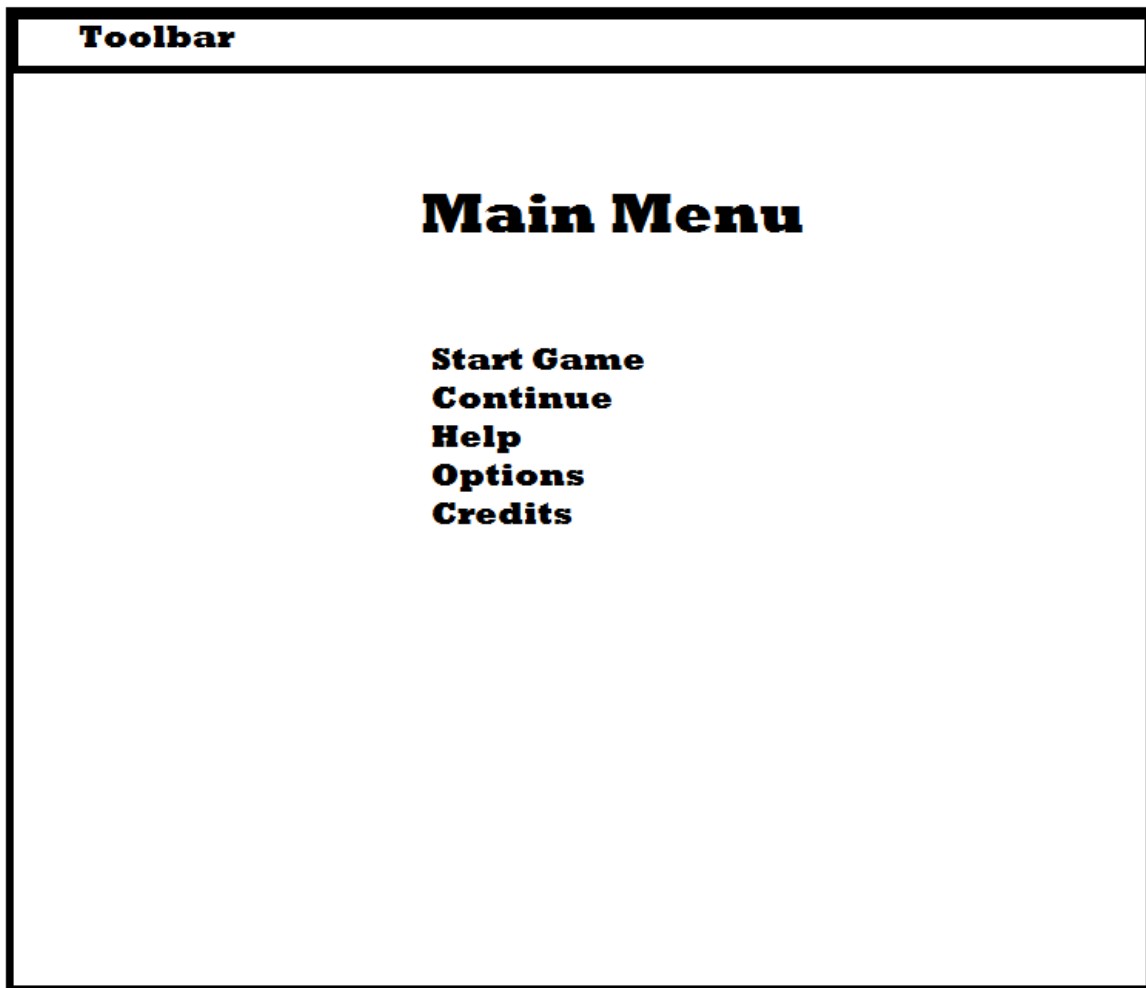
Game Engine Subteam

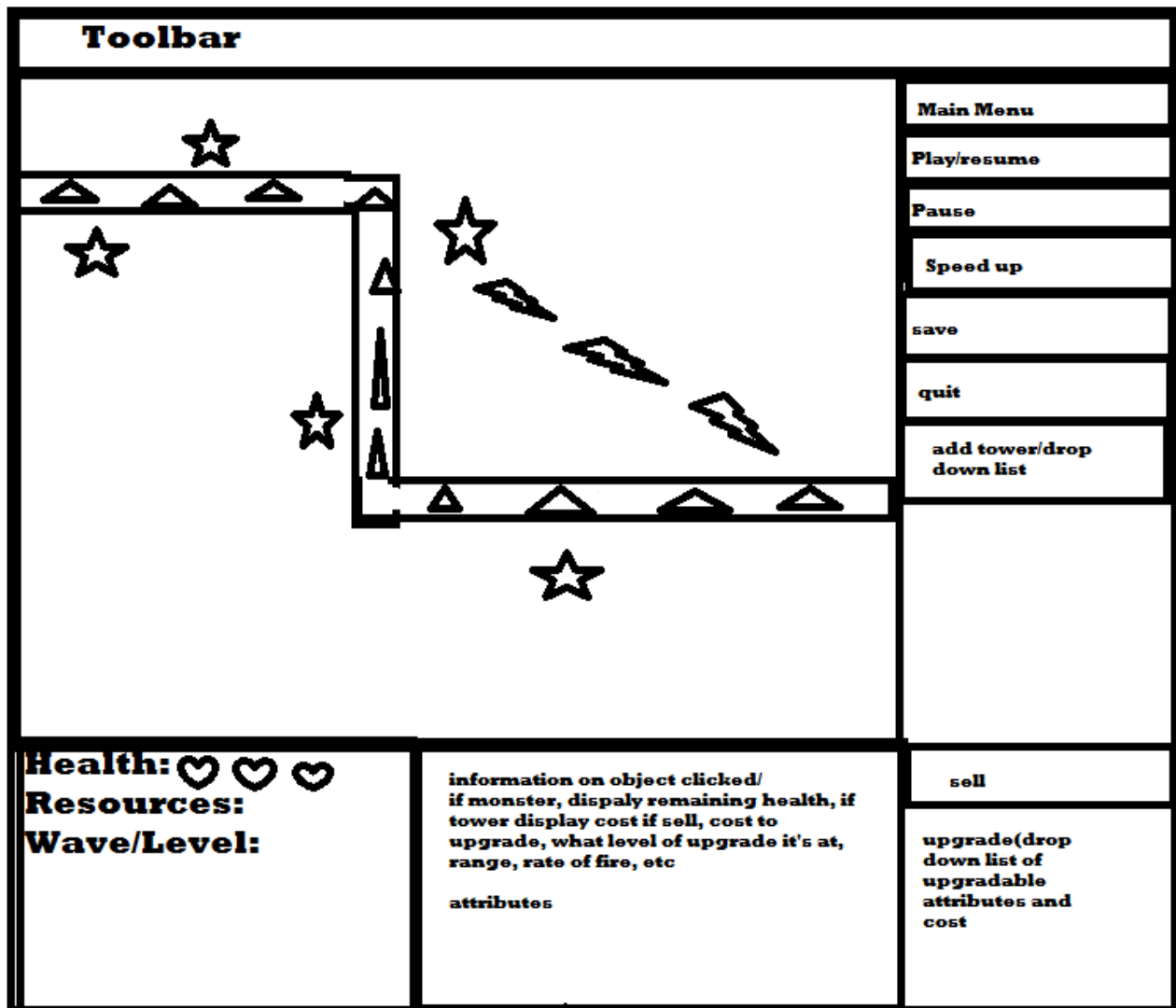
Roles include: creating TowerDefenseObjects, creating the Model, creating the Player
This will be pair programmed between all 4 of us: Dan, Austin, Jordan, and Lawrence.

Game Player Subteam

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




Appendix A: Screenshots





Authoring Environment Mock Up:

Tower Editor
Enemy Editor
Wave Editor
Terrain Editor


Attributes


Health


Rate of Fire

Range

Tower Types

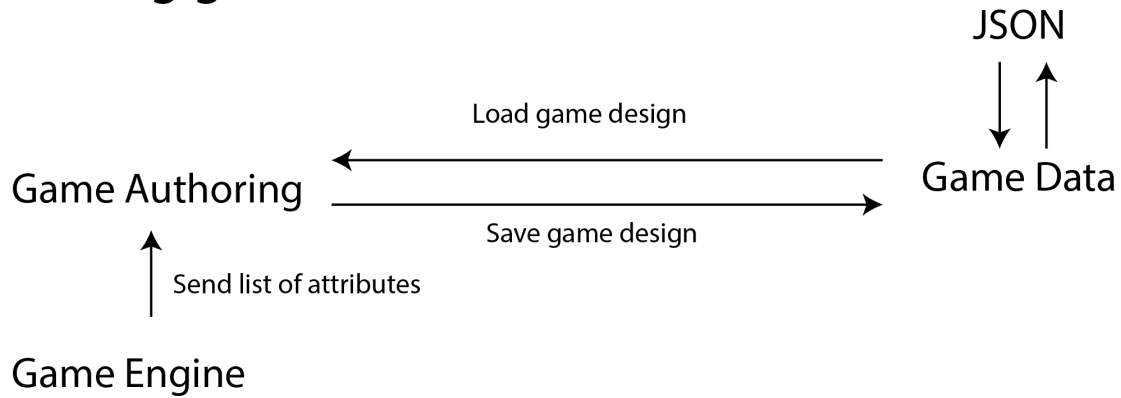
Ice


Fire


Bomb


Appendix B

Authoring game



Playing game

