API Design Document

Genre

Describe your game genre and what qualities make it unique that your design will need to support

We are going to make an RPG game building engine. Thus, it will need to support grid-based overhead movement, and a multiple-state system to handle map navigation, menu navigation, and battles. So we will need to have support for multiple states, tile setup, interaction with objects on tiles, items, and player and enemy stats.

Design Goals

Describe the goals of your project's design (i.e., what you want to make flexible and what assumptions you may need to make), the structure of how games are represented, and how these work together to support a variety of games in your genre.

There are 4 main subteams for this project. The Data team will have its components communicate with the Authoring Environment and the Player teams. In addition, the Engine will interact with the Player.

The user will be able to create a game through the Game Authoring Environment, which will allow them to build their world map-grid of images, place GridObjects such as walls, enemy-encounter spots, NPCs, or other things that would emit dialogue and give/take items such as treasure chests. We will also allow the user to upload enemy images and stats to be used in a battle system associated with the random-encounter tiles, with character stats and leveling. A menu system involving items will allow interactions with the NPCs to be item-based and more complex, so we intend to add this as well. All of this will allow the user to do a plethora of things--they are only limited by the grid structure, and their ideas behind how the user is to interact with the NPC-type grid objects. The user could make a Final Fantasy-like game, a maze full of riddles, a multiple-storyline journey like The Stanley Parable, or any number of things.

As the user builds the game, the Authoring Environment gives its information to Data to be processed and stored. Player will then access this data and build the specific game the user has created, using the underlying framework created by Engine.

There are a few goals we see as essential to the design of this project:

Focused Flexibility: Given how open ended the RPG genre is, it is imperative that we focus our project on a subset of possible features within RPGs. However, within that subset, we want to be as flexible as possible with regards to how features or elements of the game can be utilized and combined. The goal here is that a user designing with our authoring environment immediately knows what specific type of game we support, but feels totally free to design a game within the bounds of that game type.

Clearly defined API's for each module: Given the massive scope of this project (with respect to the types of project we have had experience with), it is imperative that we define the publically accessible methods for each module early on and do not stray too far from our initial design unless absolutely necessary. Otherwise, we will spend far too much time having to work on the interfacing between

modules, rather than on features of our project

Enclosed modules: The only way each module team will be able to continually add, extend, and expand upon their code is if they do not have to worry about impacting other modules' code when making changes. As such, it is essential that data, game engine, player, and authoring environment teams focus on encapsulation.

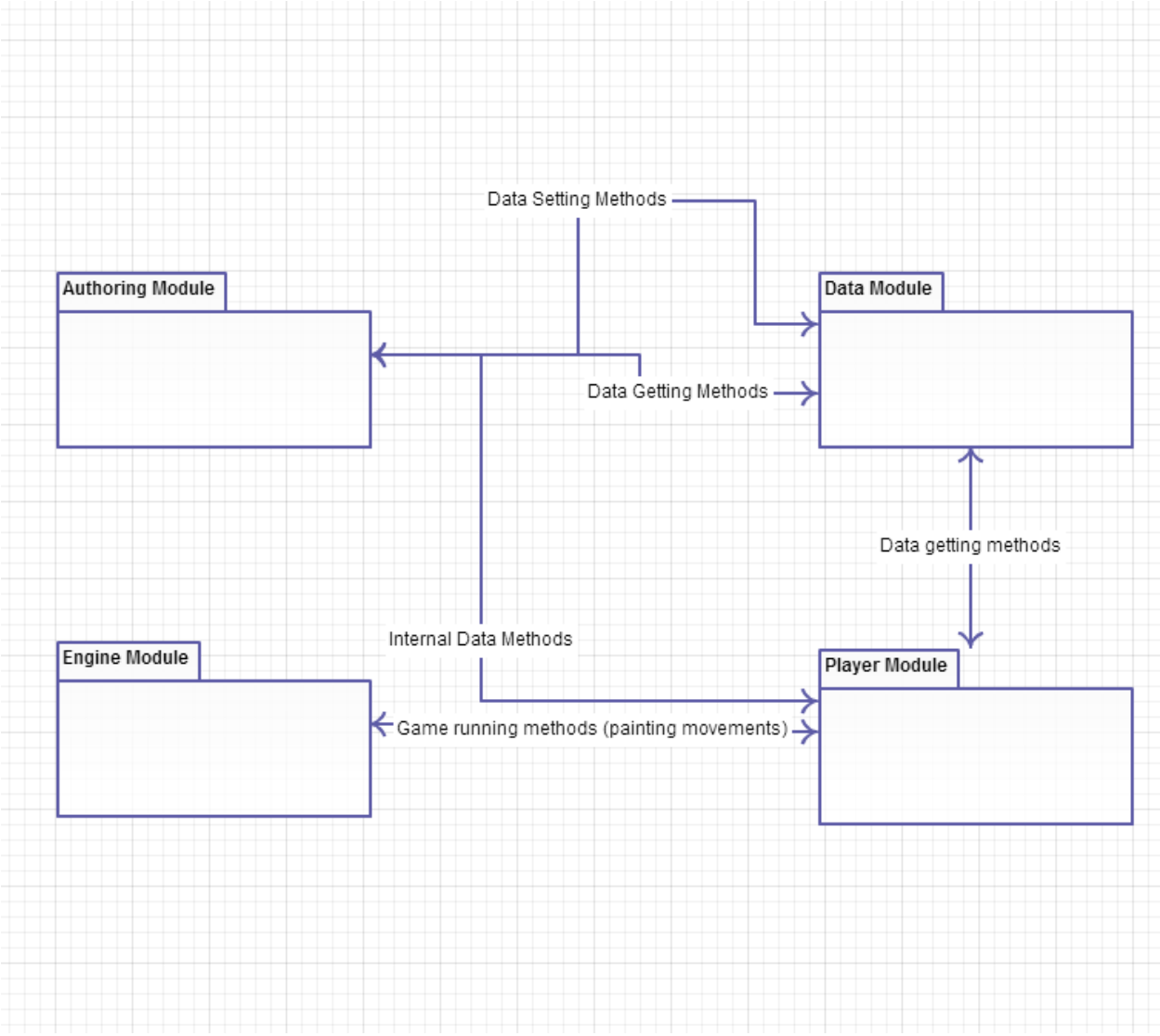
Test Driven Environments: Given the fact that multiple teams will be independently making changes, altering code, and pushing those changes back to the main repository, it is essential that teams are able to ensure that the changes they have made did not break the project. The only way to do this effectively and efficiently on a scale like this is to have all-encompassing, versatile tests for each aspect of Oogasalad.

Communication: The only way a group of this size can be successful in handling the various issues that come up in the process of designing a project so large is if the members communicate well. This includes actively working with members of the group outside of your team so that the interface between modules is designed well in addition to constantly updating members within your team so that people are on the same page.

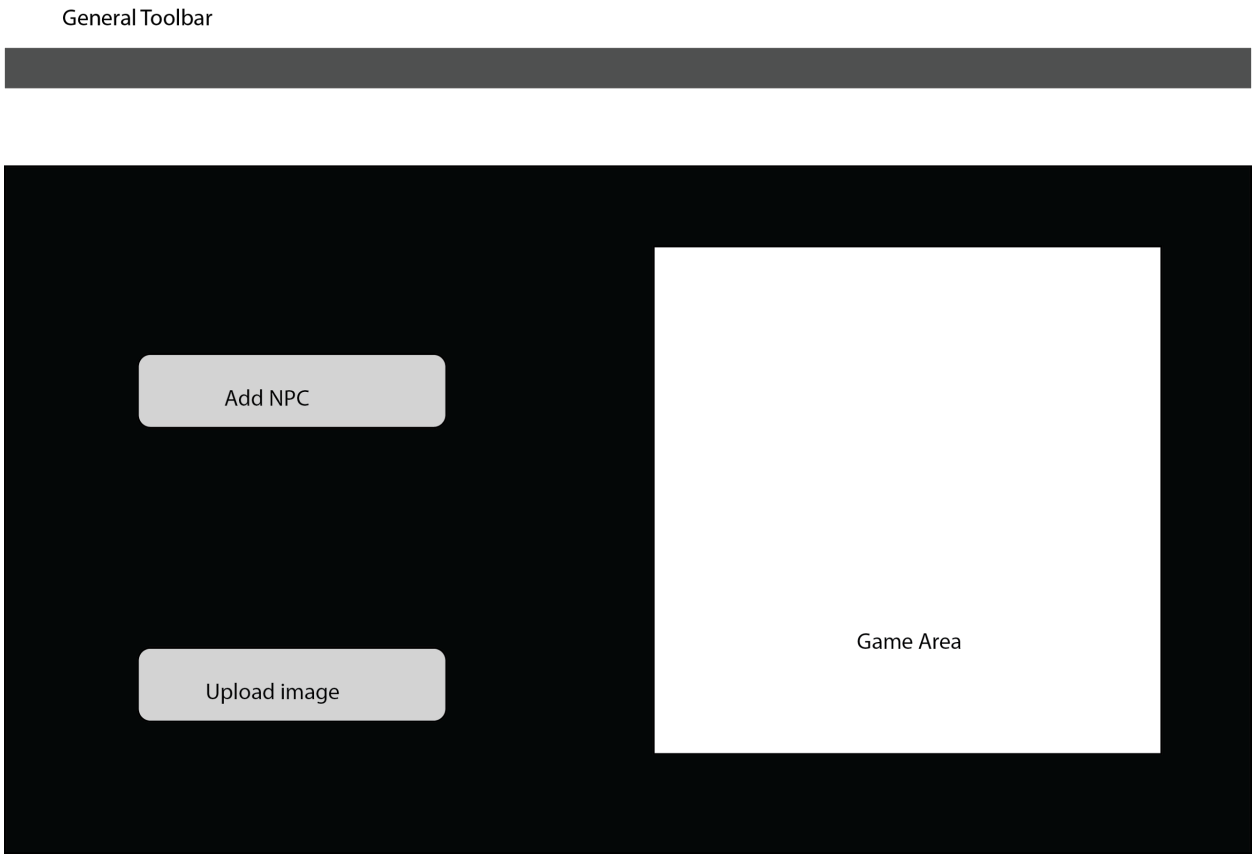
Primary Classes and Methods for each module

Describe the program's core architecture (focus on behavior not state), including pictures of a UML diagrams and "screen shots" of your intended user interface. Each sub-project should have its own API for others in the overall team.

A basic model of how we envision the various modules in the program interacting is shown below. Further details about the methods and classes of each module follow later in this document.



On the next page is a basic module of how we envision the authoring environment GUI to look.



Authoring

Methods Other Modules Call:
WorldData.getTileData(int x, int y): Returns the TileData object of the given coordinates
TileData.getImageName(): Returns the image name of the given tile--calling Data's getImage method with this name would yield the tile's image
TileData.getGridObjectDatas(): Returns the list of GridObjectDatas present on this tile
GridObjectData.isSteppable(): Returns whether or not the GridObjectData is steppable
GridObjectData.getTileData(): Returns the Tile (given by a TileData object) this GridObjectData is on
GridObjectData.getImageName(): Returns the image name of the given tile--calling Data's getImage method with this name would yield the tile's image

Methods We Will Call On Other Modules:
Data: storeImage(String name, File image)
getImage(string name)
storeWorldData(String name, WorldData world)
getWorldData(String name)
storeGridObjectData(String name, GridObjectData t)
getGridObjectDatas()

The Authoring module will interact with the Data module to provide all necessary information for a game to be created. The user will be able to upload images for tiles, and on-tile objects. When the user uploads an image for later use, they will give it a name and they will use a file chooser to select a file it goes with. Then we would use Data's storeImage(String name, File image) method to let Data know to add this image to its map of strings to images. Later, Authoring and Player would be able to use Data's getImage(String name) method to retrieve that image for later use. For map storage, we would use Data's storeWorldData(String name, WorldData world) method to let Data know to add this map's WorldData object to its map of strings to WorldDatas (the Player module could then use getWorldData(String name) to retrieve these maps, which would contain getTileData(int x, int y) that would allow them to retrieve TileData objects corresponding to tiles on the map). TileData objects that the WorldData returns would contain coordinates, the name of their corresponding image, and a list of the GridObjectData objects corresponding to all the things on that tile. For GridObjectData storage, we would use Data's storeGridObjectData(String name, GridObjectData t) method to store GridObjectData objects in their string to GridObjectData map. Then we could use getGridObjectData(String name) to retrieve these objects for display and potential edit. This allows for smooth communication of Data with the Player, as all they need to do is retrieve WorldData objects to build the map (using its getTileData method to access coordinates and images of tiles), and retrieve all the information it needs for GridObjects by calling getGridObjectData method.

Classes

- AuthoringViewer
    - Uses Feature objects in features package
        - MapBuildFeature
        - GridObjectFeature
        - ImageAddFeature
        - EnemyAddFeature
        - ItemAddFeature
        - PlayerEditFeature
    - Data-Passing Objects
        - WorldData
        - TileData
        - GridObjectData

Data

Data will be responsible for storing classes retrieved from the authoring environment, parsing them in JSON and passing on any required information to the player class. The authoring environment will communicate with the data to let data know what setters they want to call on the information in data. The player will communicate with the data by letting data know what information they want to be able to retrieve. The authoring environment is going to pass WorldData objects to data which player can then retrieve to access to retrieve the information they need.

Methods Other Modules Call:

Authoring Environment will call on Data:
storeImage(String name, File image)
getImage(string name)
storeWorldData(String name, WorldData world)
getWorldData(String name)
storeGridObjectData(String name, GridObjectData t)
getGridObjectDatas()

Player will call on Data:
getWorldData()
getImage(String s)

Module 1 (interaction with authoring environment)
The data is going to have "front end" storage and "back end" storage. The front end storage is essentially going to be updated incrementally as the authoring environment passes information to data using the storeData method. When the create game button is called, the incrementally stored data will be converted to a JSon file.

Module 1 Classes
DataCompiler: The "front end" back end compiler. Stores the information being sent to us, loads the correct image so the front end can get them, compiles data to be stored
-This may include subclasses/partner classes that help store specific components
DataConverter: Converts data to the JSON file.

Module2 – Load module (interaction with player)
The front end will select the data they want to load at which point the data will call the load data method. When the load data method is called, the stored Json files will be parsed and the necessary information will be passed to the player class (i.e. WorldData objects).

Module 2 Classes
Parser/Data Reading: This looks at the data files selected via the front end

Player

Methods Other Modules Call:

Methods Player Calls:

From Engine:
Tile: Tile()
Engine: Engine

From Data:
getWorldData(int x, int y) -> getTileData
getTileData - > getImageName()/getGridObjectDatas()
GridObjectData -> isSteppable()/getTileData()/getImageName()

getImage(String s)

The Player module will interact with both the Engine and the Data. The Data team will parse the XML/JSON file Player can serve as a view for the Data. For instance, sprites, maps, etc can be displayed from the Player module through a GUI.

Interaction with the Data team will largely be through getMethods(). The Player team will be communicating the Authoring team to decide on what features/properties the Data team needs to set() and get().

The interactions between the Player and the Engine are that the Player will utilize the methods/classes implemented in the Engine to create the game. For example, the Engine will be responsible for defining classes such as Tile, Tile Object, as well as methods such as setTileXY, setImage. The Player will actually use these to create the map, and the NPCs and the playable character.

- Classes
  - GameView - GUI that displays sprites, tiles, map from world.
  - TileMap - Instantiates tiles to create actual playable world.

Game Engine

The Game Engine will provide a functionality interface to the Player, allowing the Player to run a given game given specific data. Given that, the Game Engine will not be communicating with any other part of the overall program.

We plan on creating our own Engine but taking specific parts of the JGame Engine that we feel would be helpful to us. For example, the canvas drawing mechanism, JGObject superclass, and view vs. play area would be implemented in our design. Specifically, the Engine will provide the Player module to be able to instantiate objects on the screen, move objects given user input, control movement of sprites on the canvas, transition between different “worlds,” and keep track of game state. See below for our planned

public methods as well as the over all structure to the Engine module.

Methods other modules call:
Tile: isSteppable(), setTileImage(), setTileObject(), removeTileObject()
Engine: gameloop(), checkCollisions(), checkGoals(), setGoals() - set the goals for this game
Sprite: setImage/Animation(),move(),getXPosition(),getYPosition()
DialogueBox:showDialogue(), continueDialogue()
NPC: giveItem() (triggered by event?), doConversation(),
Player: useItem(),die(),storeItem(), doConversation(),
World:initializeBattle() - Arena mode, changeWorld(World nextWorld)
Items:isEquipable(),isUsable()
MenuBox: addSelectableOption(),

Classes

- World (abstract)
    - Three types
        - Walkaround
            - list of battle trigger locations(if applicable)
            - List of tiles in the world
            - Contains pointers to other Worlds player can enter from **this** world
            - Rule Objects
            - initializeBattle()
        - Arena (turn based)
            - all sprites in arena
            - active sprite
            - MenuBox (image itself)
                - Currently highlighted Option
                - List of possible Options
            - MenuSystem (tree - for use of item selection, move selection, etc)
            - Rule objects
        - ItemListWorld
- WorldManager
    - this manages which of the worlds is currently active
    - Keeps track of all worlds
- Sprite (abstract)
    - Instance Variables
        - image/animations sequence
        - x/y position
        - dx/dy
        - myCollisionHandler
        - Conversation
    - Methods

        - doCollision
        - move (abstract)
        - setimage/animation
        - set instance variables
        - doConversation
    - Subclasses
        - Player
            - has a Stats class
            - has an itemlist
            - has actions they can do in arena
            - has actions they can do in the walkaround
            - storeItem() - puts item in itemlist
            - useItem()
            - die()
        - NPC - Just talking
            - implement move
            - has conversations
            - give item
        - Enemy (extends NPC)
            - aggro trigger location
            - Stats class
        - Buildings/Caves
            - implement move (do nothing)
            -
        - Barrier
            - implement move (do nothing)
        - Items
            - isEquipable()
            - isUsable()
- Conversation(tree)
    - contains dialogue boxes
- DialogueBox
    - showDialogue()
- Stats
    - list of stats
- Tile
    - List of objects (npc/enemy/walls/grass) on it
    - dimensions and range of x/y positions it governs
    - isSteppable instance var - Can this tile be stepped on or not?
    - Constructor: x,y,name,list of objects
- CollisionHandler
- MenuSystem
- MenuBox

- Weapon
    - keeps track of stats (power, weight, etc…)
- Goal
    - this object is composed of a sequence of Event objects which must be completed in order for the goal to be deemed completed
- GoalManager
    - Keep track of all of the goals
- Engine
    - instance variables:
    - Methods:
        - gameloop method
            - checkCollisions()
            - checkGoals()
        -

Example code

Rather than providing specific Java code, you should describe two or three example games from your genre in detail that differ significantly and the data files that represent them and could be saved and loaded by your project. You should use these examples to help make concrete the abstractions you have identified in your design.

Our design will be extensible enough to create many different games within the RPG genre. Beyond the obvious example of any sort of Final Fantasy/Pokemon type game, choose-your-own-adventure esque games such as The Stanley Parable would be possible thanks to NPC interactions, items, map changes, etc. Other possibilities are maze games, puzzle games, and even dating simulators!

All game types will use JSON data files, which will be handled by Data and passed to Authoring and Player when necessary.

Alternatives

Explain some alternatives to your design, and why you choose the one you did

For the tile, it was recommended that NPC characters and events be inherent methods of a tile. But because such a relation could cause many dependencies, it was decided that NPC characters and events would be separate from the tile, and lay on a separate “layer” on top of the tiles that make the world/game map. With the chosen design, each tile will have a list of events and NPCs that are related to the respective tile. This separation between the tile and events/NPCs provides a layer of abstraction that allows for maintainability and extensibility.

One alternative design was to have Data and Player parse the JSON files. However this was deemed to be too cumbersome. The task would be repetitive, and it would be better for the Data to have a flexible API that the Player can take advantage of. Therefore, the Data will be parsing the JSON file.

The Data suggested that all objects be saved before rendering to the view panel and processing the player and game engine. However, to make the game interface more usable, it was suggested that instant object retrieval be possible. Therefore, Data should save as the creation of the game progresses. The end result would be the possibility of a soft and hard save.

Roles

List of each team member's role in the project and a breakdown of what each person is expected to work on.

Authoring Environment

- Jacob
- Pritam
- Richard

Inside the Authoring Environment we hope to be able to eventually compartmentalize our work enough to be able to work independently,

Engine

- Parker
- Lee
- Chad

Data

- Davis
- Sanmay

Player

- Peter
- Brandon
- Benson