Design Document - Print Stack Trace

Comp Sci 308 Final Project - VOOGASalad

# Genre:

For our project, we will be creating a program that helps create and run platformer games (ex: Super Mario Bros., Flappy Bird, Donkey Kong). Platformer games often involve traversing across platforms and terrain in order to progress. There are a few aspects of this genre that are unique. Firstly, it has different environment platform types. For instance, in some games certain platforms can be passed through from above but not below. Others can have varying levels of friction, such as ice, which causes the player to have reduced control of his character’s movement. Also, some platform games have physical obstacles rather than just enemies that the players have to avoid such as lakes or lava. Additionally, some platform games have objects, such as vines in Donkey Kong, that can traversed to help the player reach another platform.

Additionally, movement is very varied in platformers. The game may scroll either sideways (e.g. Mario), upwards (e.g. DoodleJump), or downwards (FallDown) and gravity levels differ depending on the game. Levels can also progress in different ways. In some scenarios the scene changes to a new one when the hero moves to the other side of the screen. In other scenarios, the screen is constantly scrolling.

There are multiple types of collisions in platformers. One type of collision could be where a player jumps on an enemy and then the enemy is killed. Another could be when a player shoots an enemy in order to kill it. Additionally, blocks such as a spike blocks or lava can damage the player.

Our design will need to be able to handle these multiple types of obstacles and blocks. An extremely important aspect of platformers is handling the collisions between the player and platforms. Few things are more frustrating than having a platform which you can’t reach, or to have a scenario where it seems like you’re about to land on a platform, only to clip through it. We will need to be handle multiple scrolling directions, and handling what events will cause a level end.

# Design Goals:

The primary goal of our project’s design is to create an application that can seamlessly switch between game authoring and game playing environment. We want our Game Engine to be flexible enough to manage the data input from the game authoring engine and run-time game play from the game player simultaneously. This will require the level, sprite, and associated classes to be contained within Game Engine. Game Engine will interact with only the Level, and the Level will know how to update and manage all the sprites and data within it. This flexibility and abstraction will enable us to create hierarchies of levels and sprites, making the design and implementation of new games very simple. We are assuming a work flow of front end for the main application to the front end of either the game player or game author. From either of those, we will interact with the game engine and the rest of the back end, most likely through a singleton pattern. This flexibility enables us to easily achieve our goal of seamless, simultaneous running. The game engine will interact with game data through calling read and write methods that store and retrieve program and game play preferences. This entire workflow ensures that our program stays shy and is easily extensible.

# Primary Modules and Extension Points

**Game Authoring Environment**

The front end will consist of a series of wizards. Levels can be created by dragging and dropping objects (enemies, players, rewards, obstacles) from a sidebar onto a grid. One can then set the properties for the object by clicking them revealing a set of modifiable properties. Properties such as if a block can be moved through will be decided by a drop down menu. Properties dealing with number values will be entered via a text box.

In the backend, objects dragged onto the screen will be stored in collections based on what type of object they represent. The back end will also have to check whether the space where the user wants to add a new element is already occupied. If the space is already occupied, the user will be notified, but has the chose to have two objects on top of each other. Enemies will be stored in an enemy collection, obstacles will be stored in an obstacle collection, players, if there can be more than one in an instance, will be stored in a player’s collection, and so on. We will also allow the setting of level goals (e.g. reach a certain point to win the level, earn a certain number of points, avoid death for as long as possible) and other global level info (screen scrolling, physics, etc.). When saved, all of these collections will be packaged together (into a level object) and sent to Game Data.

We will also give the back end of the Authoring Environment the ability to load a level into the environment builder. This ability will help with changing levels that have already been built but that have a few issues in them.

### Variables:

* enemy collection
  + enemies will have:
    - Point location
    - int health
    - int damageDealtOnContact
    - double speed
    - directionOfMovement
* Player collection
  + Player will have:
    - Point startingLocation
    - int health
    - int lives
    - double speed
* reward collection
  + rewards will have:
    - int value (could be points, health, etc…)
* obstacle collection
  + obstacles will have:
    - boolean isCollidable (if object can be moved through)
    - int damage (damage dealt upon contact)
    - double traction (inhibit or speed up movement)

### Classes:

Public AuthoringEnvironment

* private LevelCollection myCurrentLevelCollection
* public Group initialize()
  + Boots up the Game Authoring Environment
* public void LoadScene(Collection<Level>)
  + Once the backend parses a level based on a saved file into multiple levels, the front end will load the level to be portrayed by the GUI in order to further edit
* private File saveAndExport(Level currentLevel)
  + Saves and exports the built environment for use in the game engine.
* private loadLevel(Level myNewLevel)
  + Load a level into editing environment. This way further tweaking can happen if the user decides that needs to be done.

/\*\*

A Collection of Levels that make up an entire game

\*\*/

Public LevelCollection

* private Level myActiveLevel;
  + This is the level the user is currently editing
* protected setActiveLevel(String id)
  + Changes the active level when the user selects what level they want to work on
* protected Level createNewLevel(String levelName)
  + The user creates a new level to work on in their game and is added to the collection, and is set as the active level

/\*\*

Level is a data object containing collections of different types objects

the user can place

\*\*/

Public Level

* protected void setNextLevel(Level l)
  + sets the next level that the user will transition to after you beat the current one
* protected void setTransitionAnimation(String levelId, String transition)
  + sets how you transition to the next level (i.e. a certain type of splash screen to transition between levels)
* protected void setBackgroundImage(String imagePath)
  + add a new background image to the level
* protected void addObstacle(String id)
  + add a new obstacle to the level
* protected void addReward(String id)
  + add a new reward to the level
* protected void addEnemy(String id)
  + add an enemy to the level
* protected void addHero(String id)
  + add a hero to the level
* protected Sprite getSprite(String id)
  + retrieve a specific sprite from the level by iterating through all types of collections
* private boolean canAdd(Point position, int width, int height)
  + Private method used to decide if a sprite can be added at a given location. Because we don’t want sprites to overlap, it is important to check if the css
  + called by all our “add methods” (i.e. addHero, addObstacle, etc)
* private boolean isValidId(String id)
  + This disallows the user from adding a sprite and giving it an ID value that they already assigned to another sprite
  + also called by all our “add methods” (i.e. addHero, addObstacle, etc)

Public interface iSpriteCollection

* protected void addSprite(String id)
  + add a new sprite to the collection
* protected removeSprite(String id)
  + remove a sprite from the collection
* protected void getSprite(String id)
  + retrieve a sprite from the given collection

/\*\*

Super class for all sprite objects. Sprite will be extended by other objects

such as enemies, heroes, obstacles, and rewards.

\*\*/

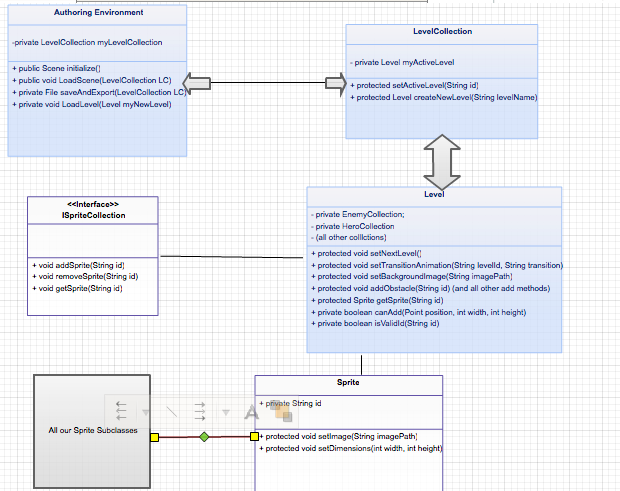
Public abstract Sprite

* public Sprite(String id)
* protected void setImage(String imagePath)
  + Set the image that will represent the sprite
* protected void setDimensions(int width, int height)
  + Set how many grid spots the sprite will take

Public Hero

* public Hero(String id)
* protected setPosition(Point location)
  + Alter the position of the hero whenever he is moved
* protected setHealth(int health)
  + Set the initial health for the hero
* protected setSpeed(double speed)
  + Set the speed at which the hero will move
* protected setStartLocation(Point point)
  + Set the spawning location for the hero

**Game Author UML**



## 

## Game Engine

* sprite/collision mangement
* object states/ levels (abstract hierarchies)
* Rule/object interactions (abstract enough to read in new rules)
* Front End: animation of collisions/call object updates
* functionality for puasing
* Rules:
  + fly/jump/physics
  + background scroll vs move w/ character
  + collision stuff/animation

### Helper Classes:

* CollisionHandler
* Abstract Sprite
  + Player
  + Abstract Enemy
    - Minions
    - Bosses
  + Platform
  + Abstract PowerUp
    - FullHealth
    - Speed
    - SloMo
  + Abstract Bullet
    - Laser
    - etc.
* Abstract PhysicsEngine
  + MotionHandler

### Public Methods:

private static GameEngine singletonInstance;

private LevelModel currentLevel;

//-------------------CONSTRUCTORS-------------------//

/\*\*

\* Constructor Method. It is private because this class is designed to only

\* be used as a Singleton. Call the static method getSharedInstance to

\* access the Singleton.

\*/

private GameEngine() {

//TODO: Implement Constructor

}

//-------------------PUBLIC METHODS-------------------//

public RuntimeModel getStatus() {

RuntimeModel ret = new RuntimeModel();

/\*TODO: Implement -- this is looks like an accessor to everyone else but

\*a RuntimeModel should be created at the time it is called and built

\*from all of the various at time of call. \*/

return ret;

}

//-------------------ACCESSORS-------------------//

public LevelModel getCurrentLevel() {

return currentLevel;

}

public void setCurrentLevel(LevelModel currentLevel) {

this.currentLevel = currentLevel;

loadLevel(currentLevel);

}

public static GameEngine getSharedInstance() {

if(singletonInstance == null) {

singletonInstance = new GameEngine();

}

return singletonInstance;

}

//-------------------PRIVATE METHODS-------------------//

private void loadLevel(LevelModel level) {

//TODO: Implement -- Will be used for setLevel but also reseting level.

}

* static GameEnging getSharedInstance()
  + GameEngine.java class is the portal between the front-end and back-end. This is an accessor method to return the Singleton instance for the app’s backend.
* RuntimeModel getStatus()
  + returns a large model object that holds many details about the Model
* pauseGame()
  + pauses the game
* loadGame()
  + loads a new game to play
* saveGame()
  + saves progress for the game
* viewGames()
  + method for displaying the games available to play
* viewScore()
  + view current score
* viewHighScores()
  + viewing the stored and ordered scores from all games played (of a certain type)

### Private Methods:

* handleCollisions()
* updateSprite()
* saveScore()
* void isFinished()
  + True:
    - winLevel()
    - loseLevel()
  + False:
    - do nothing

### Variables:

* private Collection<Level> myLevels
* private Level currentLevel
* private CollisionHanlder
* private PhysicsEngine

## Game Player

* Hold pointer to GameData and GameEngine
* Front End (backend = gameEngine)?
* Read specific level/game data
* run engines/save progress (checkpoints)
* pause/play
  + Gameloop?
    - GameEngineUpdate
    - isLevelChanged

The Game Player section of this project will be an additional front end. This front end will be used to show GUI interactions for when the game is played. The difference between the front end for the Game Authoring Environment and the Game Player is that the Game Authoring Environment’s front end will be for created games/levels; the front end of the Game Player will be the GUI interactions of the actual game being played, such as player movement, player collisions (graphically), level transitions and animations. The Game Player will also contain buttons for saving/quiting/pausing/resuming the game, text field for entering names for high scores, labels and boxes for displaying the score of the player, health, and/or player status. Additionally the Game player will host the eventHandlers for the GUI Elements.

The backend of the Game Player will be coded into the Game Engine. The design goal that we want for the Game Player is to keep the front and back ends as independent as possible. We don’t want changes to one to cause destruction in the other. The front end (Game Player) should just update the scence with new images, new levels, new scores, etc. The backend (Game Engine) will be responsible for making changes to those data. This is will we will keep the two ends independent. This design will allow for changes to be made without extensible writing complex code on the other end. With any project, We want the Game Player to be design well to allow future extensions, so that the code is scalable and resilient to changes, and we want our code to be testable. Designing our code with JUnit Test in mind is extremely important. With this size of the project, we need to ensure that we have testable subsets of code whenever changes are made. This will ensure that the fault domain of changed code will be explicitly known.

### Public Methods:

* public Group initialize()
* public Scene updateScene()

### Private Methods:

* private pause()
  + Pauses the game as its running
* private resume()
  + Resumes the game after it is paused
* private stop()
  + Stops the game
* updateScore()
  + Update the score displayed to the user as they score points
* private setHandlersForGuiElements()

### Variables:

* private Collection<Button> myButtons (GUI elements: textfield, boxes, Labels,etc)
* private Score currentScore
* private Collection<Score> highScores
* private Collection<ImageView> myImages

## Game Data

* Acts as Database; Authoring Environment and GamePlayer pull information from.
* Rules,goals,behavior/interaction
* Data Type: JSON/GSON
* translate frontend Data
* Front End passes data to here (via public export)
* Once level information parsed into JSON file, made available for gamePlayer and game Authoring Environment to pull

### Public Methods:

* public void writeLevel(Level) (called by Authoring Environment)
  + Writes a particular level to a data file
* public void writeGameData(Collection <Level>)
  + GamePlayer pushing game info
* public JSONFile loadFile(JSON File name) :
  + GamePlayer pulling initial game info

### Private Methods:

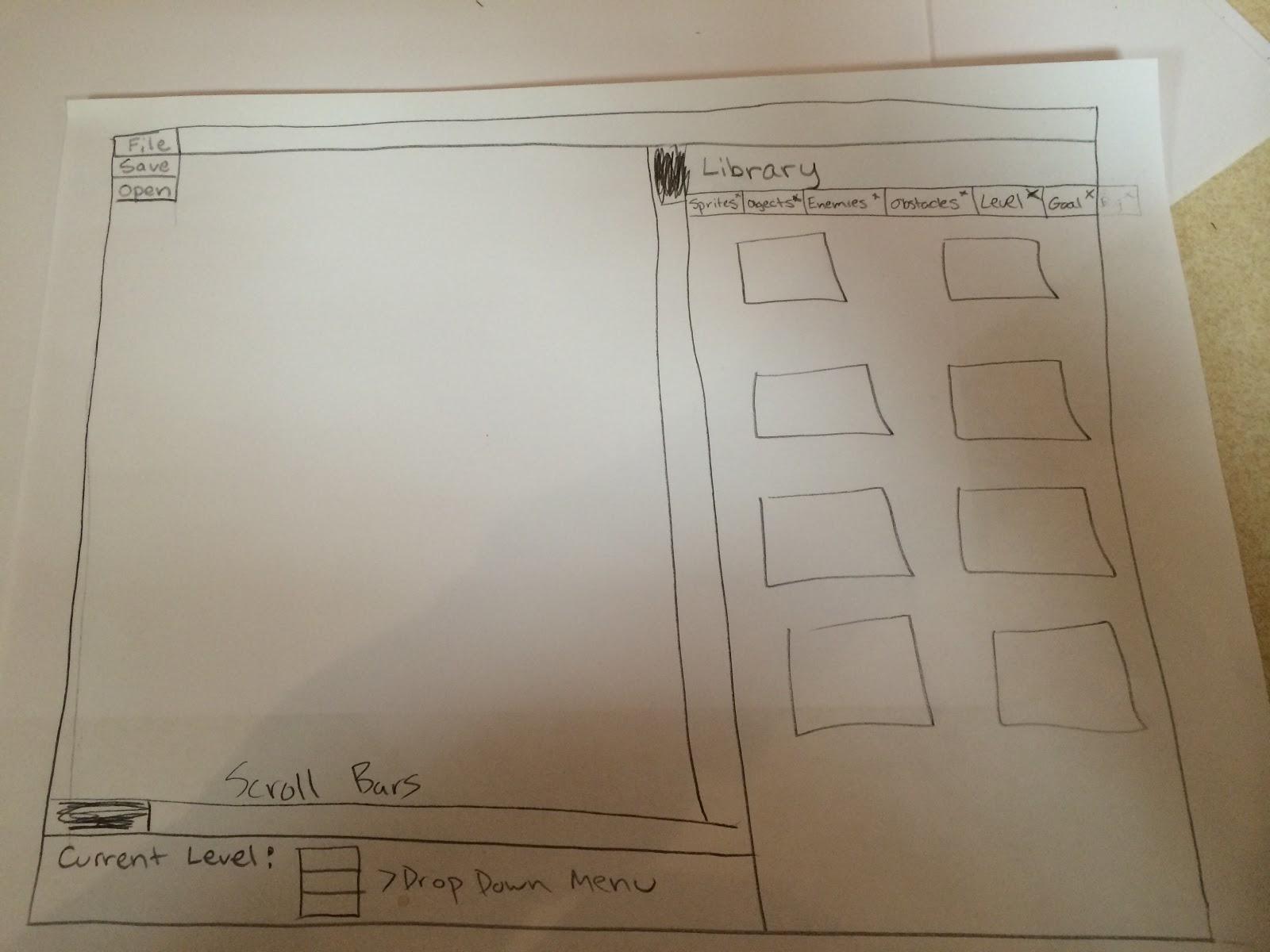
none

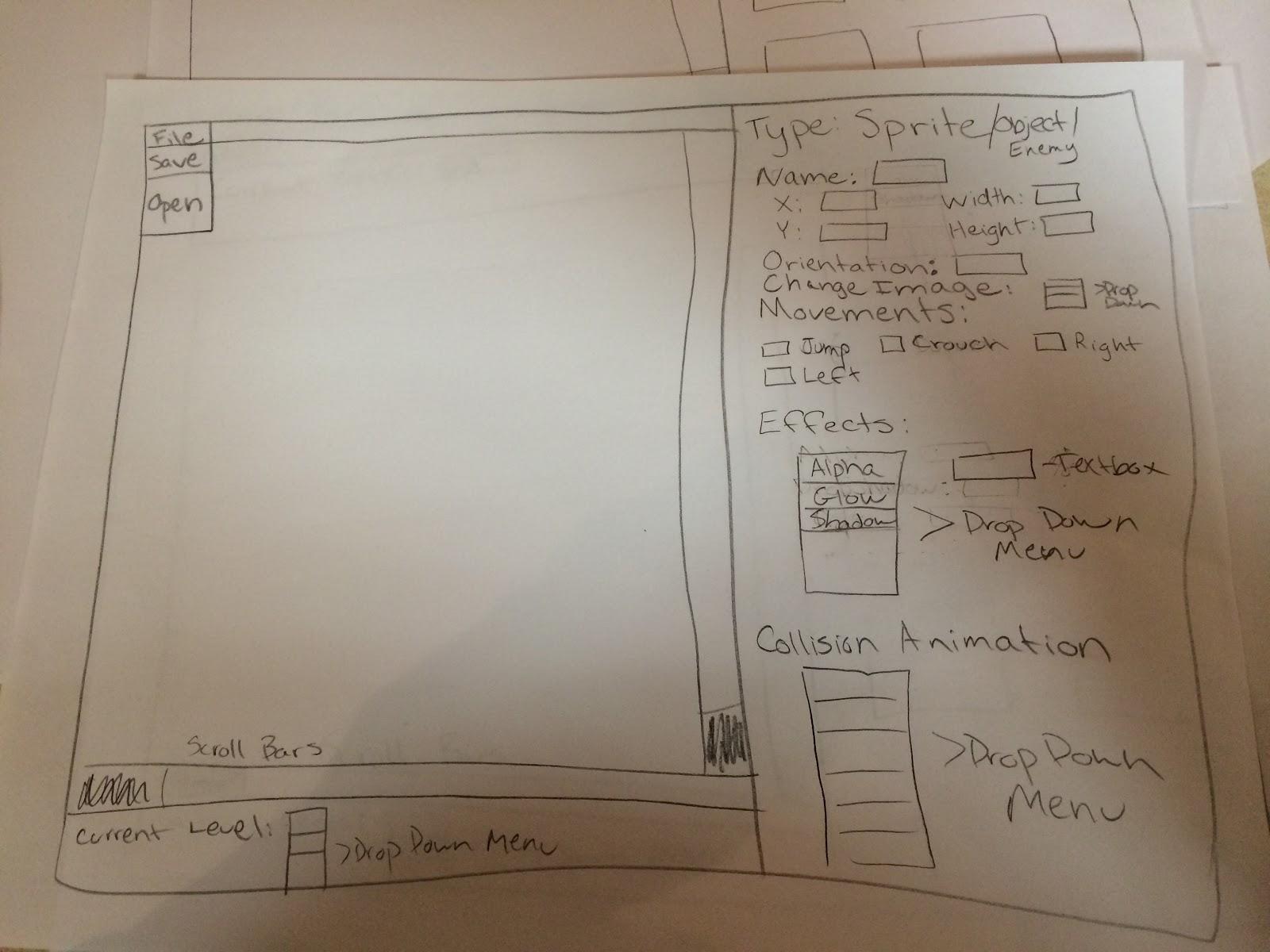
### Variables:

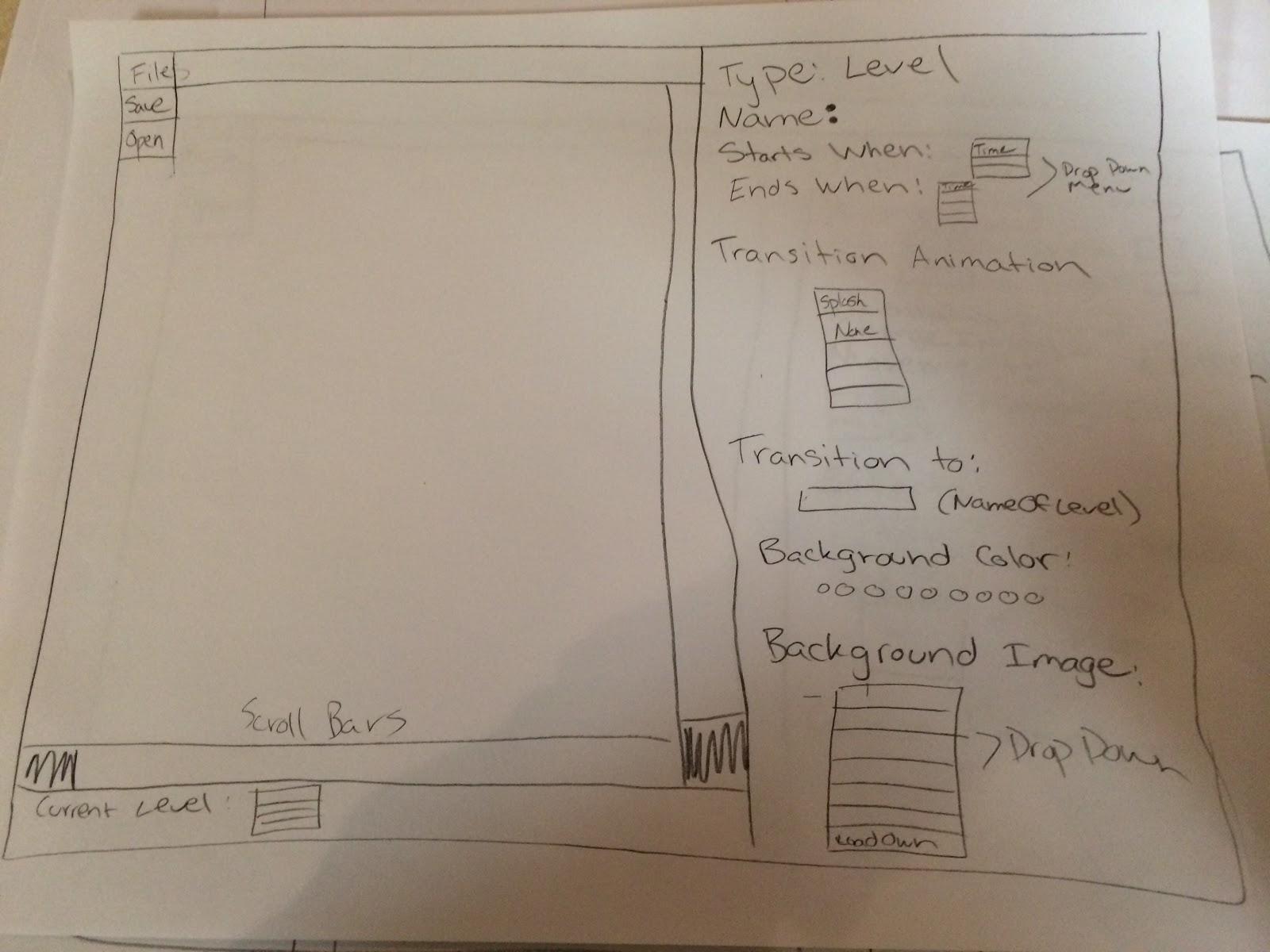
private JSONFile File

private String fileName

**Front End Diagrams**







# Example Code:

SUPER MARIO BROS.: Super Mario Bros. involves multiple platform types, multiple enemy types, environments that scrolls when the character moves, and the goal of reaching a certain point or defeating a boss. Our level creator will work for all of these features. First, we will allow users to set create the goal of reaching a flagpole for the normal levels or defeat bowser in a boss level. Enemy behavior in Mario is fairly simple, as enemies really just go forward and change course when they hit an object. This type of behavior can be set in the Game Authoring environment. The scroll with the player feature will also be one of our scrolling options (others include side scrolling and vertical scrolling at set speeds). Player actions include horizontal walking, jumping, and possible powerup (fire flower!) usage. Finally, the platform behavior is determined when the level is created. The engine will be in charge of handling all of the collision detection.

DOODLE JUMP: This should be simple. The goal is to accumulate as many points as possible, which is one of the preset options. The platform types can be set in the authoring environment, though the majority will be one-way platforms (i.e. can jump up through them, but cannot fall back down). Once again, the engine will handle collision detection in the game loop to deal with this. Player actions will include horizontal walking and jumping.

FLAPPY BIRD: Side scrolling will be set in the authoring environment, as well as danger platform blocks (i.e. blocks that kill you when you touch them, all blocks in Flappy Bird are of this type). The only player action allowed is “jumping” (i.e. increasing elevation). Active gravity is set in the authoring environment. The only thing the engine does here is process jumping, collision detections, and scroll the obstacles portion of the level. The total distance traveled before dying will be tracked and reported upon death.

# Alternate Designs:

For the Authoring environment, we originally planned on assigning a goal property to objects. This works in scenarios such as defeating a boss or touching a flag pole in order to finish a level. However, we found that this implementation would make it difficult to handle goals such as reaching a certain amount of points or surviving a long enough time. As a result, we decided to create a goal hierarchy to handle different types of goals, which can be used by the game engine to enforce the end of a level.

For the Game Engine, we could also have our physics and colliosns stored directly within our game engine class. We could also make it non-singleton instead of using the singleton pattern as we described above.

# Team Roles:

## Front End:

### Overview of Tasks:

The front end team will be responsible for creating the GUI and passing user input to the back end. We have three front end team members that will work together for the design and implementation. Both the Game Authoring Environment and the Game Player will have significant front end components, while the Game Engine and Game Data will have minimal to no interaction with the front end. Game Auth and Game Player will have specific front end point people. The third person will be responsible for the main application and switching between the Game Auth and Game Player modes.

### Team Member Roles:

#### Nick Balkissoon:

Working with Petra on panels for Game Authoring. Will also help back end tie in with front end (point man on back-front end communication).

#### Daniel MacDonald:

In charge of handling transition between different games (including splash screens), replaying, general outer menu stuff. Secondary responsibility of game authoring GUI.

#### Petra Ronald:

In charge (along with Nick) of creating GUI panels for game authoring.

## Back End:

### Overview of Tasks:

The back end is composed of handling the objects created in the authoring environment, creating the game engine, and dealing with the game player and game data. We have split the back end members into three separate teams, one to handle each of these tasks.

### Game Authoring Roles:

#### Ethan Chang: Creating sprite hierarchy

#### Jack Baskin: Creating main collections for holding level objects

#### Nick Widmaier: Creating level classes

### Game Engine Roles:

#### Justin Carrao: Collision detection

#### Zach Podbella: Physics Engine

#### Pranava Raparla: Communication between game data and game engine

### Game Player and Game Data Roles:

#### Marcus Cain: Game player

#### Ashwin Kommajesula: Game data

# Shoutouts

Shoutout to our King and Dear Leader, Professor Robert Duvall.

Shoutout to the current CS201 students for showing us the importance of not copy-pasting.

Shoutout to Teddy Ward for his dope halloween costume.