Team: Bits Please
Vooga Salad Plan v2.0
Members: Abhishek Balakrishnan, Shreyas Bharadwaj, Chris Bernt, Will Chang, Davis Gossage, Safkat Islam, Arihant Jain, Arjun Jain, Kevin Li, Eli Lichtenberg, Benjamin Reisner, Wesley Valentine

Authoring

- Presents a UI to the user which accesses the Game Engine library to create instances of sprites / actions / events. The user is allowed to create new types of game objects and place them on the screen. This is achieved by composing properties from engine onto game objects.
- The Authoring Environment will use a Model-View-Controller design. The Model and View are not aware of each other, and the Controller will handle communication between the two. View objects will also observe Model data, to handle all updates that need to be visualized on the GUI. Interactions initiated in the View (for example buttons, drag and dropping) will be communicated through the Controller using a series (potentially an inheritance hierarchy) of custom EventHandlers. This allows event handlers instantiated in the controller to have access to all of the parts of the GUI without the parts of the GUI having direct access to each other. All of these changes will be sent down to the model which contains all of the GameData, and different parts of the view such as the LevelView will observe the changes and update for the changes.
- The Model will contain the GameData instance variable which represents the current game and contains the list of available characters, levels, events, images, sounds, and variables.
    - The individual components of the Model will be observed by their corresponding components in the View (for example GameObjectView observes GameObjectCollection). Additions, deletions, and changes to elements or lists of elements will happen in the Model, which will notify the View to change accordingly.
    - GameData is the wrapper object that holds all of the relevant game data. This includes major collections of GameObjects, Levels, Graphics, Sounds, and Conditions. Our GameData object is like a database in that it is meant to store the game data to reference and modify in a passive fashion. It was determined that it would be detrimental to have a lot of behavior in this object since it was to be serialized and sent to the engine/player. This object will store no JavaFX objects, but instead references to images and sounds to ensure that the GameData can be easily serialized.
    - The Model can load existing game files to reauthor and save the current one.
    - Model will also contain some basic methods that interact with the Data module to load and save new game files into the model. It may also include the ability to load elements (such as characters or sprites) into the game so one can repurpose previous creations and work.
- The View contains graphical representations of the available elements that can be selected and dragged onto the level editor.
    - View components are organized on a BorderPane and operate through a hierarchy. This allows us to mix and match the locations of each View component, making it simple to rearrange our GUI.

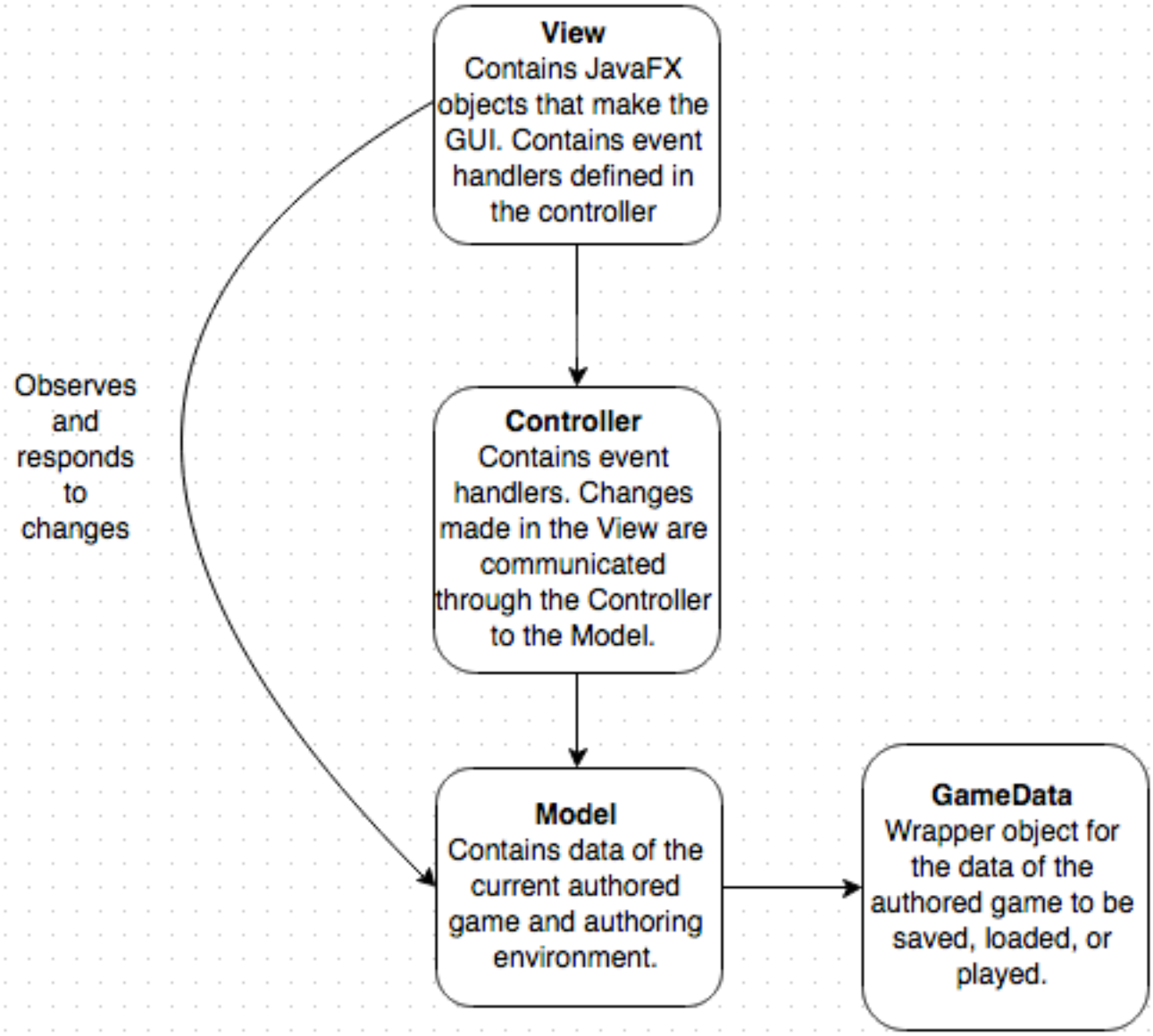
    - The objects that are actually displayed on the view will also contain their corresponding data objects in the model, so that a user can select that element and see all of its properties.
    - This system also allows the user to edit properties from the front end that adjust properties in the back end.
    - Each of these objects has the ability to update itself upon a change.
    - On the view, all of the elements that the user can interact with (such as buttons), will be created from a GUI feature hierarchy, which have actions and the ability to make the correct tool.
- Characters can be created with editable properties. These properties will be provided by the game engine and composed on the initially blank character object the user created.
- While the game will explicitly contain the list of available characters to be created, a level (which a game also contains) will explicitly contain instances of those characters for that particular level.

Authoring API:

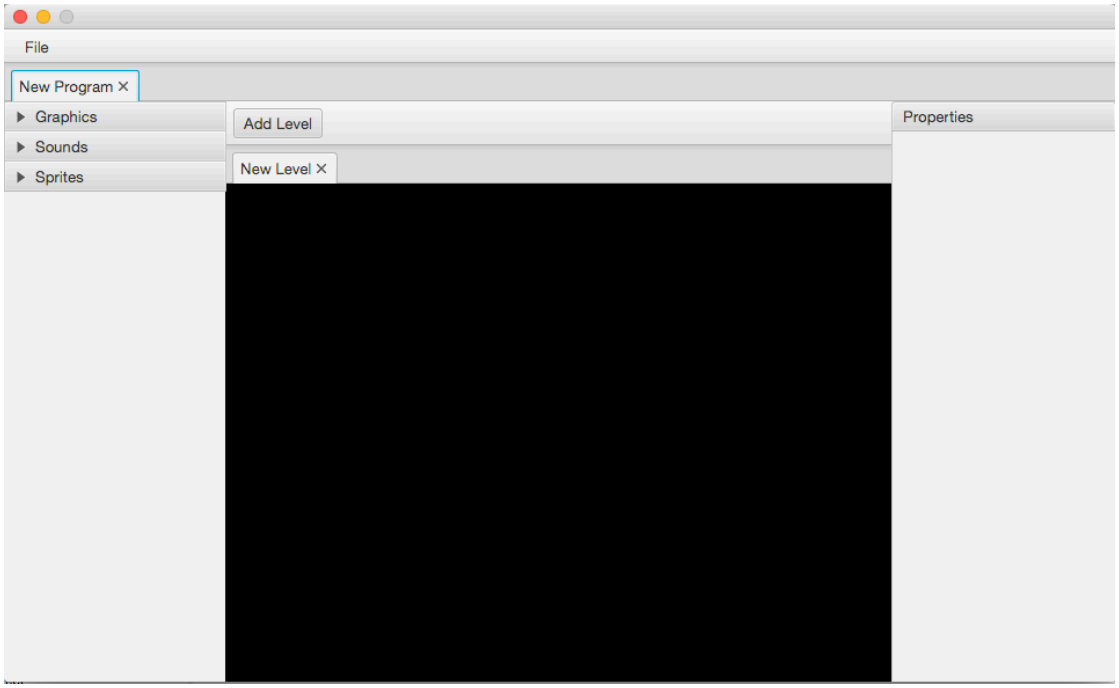
The Authoring Environment does not have many public methods for its external API since not many parts are explicitly dependent on it. There are however a few methods in the AuthoringModel class that interact with the Data Module to save, load, and import data from and to the authoring environment.

```
/**
* Save the current GameData using serialization
*/
public void save() {
}


/**
* Replaces the current GameData file with a new file that is loaded in
*/
public void load() {
}


/**
* Allows the user to extract certain elements from a GameData object to import into their current project
* @param gd Represents the GameData object from which we want to extract elements
*/
public void importData(GameData gd) {
}
```

Authoring UML:



User Interface:



Engine

Game Engine provides the primary building blocks to create and run any game.

The *GameManager*, *LevelManager*, *Physics*, *Rendering*, *CollisionDetection,* and *Camera* packages serve as the backbone and foundation module of the Engine, providing the essential logic for running the game properly.

The *GameManager* initializes all the moving parts of a game (the classes listed above and below), and sets up the game loop.

The *LevelManager* tracks and manages progression through the various *Levels* of a game.

Physics provides the logic and calculations, allowing a simulated game environment.

CollisionDetection currently serves to check for collisions between *GameObjects.*

Rendering interacts with the *Canvas* provided by the *GamePlayer*, generating *RenderedNodes* to encapsulate the non-serializable JavaFX portion of the GameObjects.

Camera manages the viewing window from which a user might play and interact with a given game. Also will manage scrolling within the game.

The *Condition*, *Action*, *Level*, *GameObject*, and *Component* packages provide customizable functionality for the user to define characteristics and rules for any game desired.

Conditions allow binding of key presses, in game events, checkpoints, initializations, and more by associating with *Actions*. They extend *Listeners* from the Java library.

Objects that implement the *Action* interface serve as function wrappers, in the sense that they encapsulate the public method calls of other Game Engine classes to allow grouping of execution, and are triggered if *Conditions* have been met.

The *GameObject* provides a template for users to define any game character with as many customizable *Components* as desired (eg. health, attack, damage, items inventory, mana). These components would be held in container objects.

All these objects (will) implement the *IEnabled* interface, allowing the *LevelManager* and/or the *Level* to better manage and account for these user defined objects. The implementing the *IEnabled* interface gives an object an ID, and the ability to be "enabled" and "disabled" depending on where a player might be within a running game.

Layers of Functionality:

- GameManager -
    - Contains Canvas, the game loop, Renderer, (Physics), Camera
    - LevelManager - contains a master list of Game Objects, Conditions, and Levels
        - Default initialization information for the Game
        - Game Objects
            - RenderedNode
            - PhysicsBody
            - Components (fields such as Health, Mana, Attack, Defense, Damage)
        - Conditions
            - Actions
        - Levels
            - Contains initialization information for specific Game Objects for the level, as well as enabled Conditions.

Physics Hierarchy

Vector valued functions, scalars which will be a map

· Vector valued quantities-physical properties/characteristics that are vector quantities. These things can also have subclasses, such as the subclasses of force (types of forces)

o These are then applied to physicsBody in different ways

· Scalars are scalar valued quantities, usually constants

o Are then used generally to create new forces, but also serve different purposes like providing characteristics for the Game Object like whether it moves or not when something collides with it.

PhysicsBody

· Has several functions to update itself and its helpers

· Has public methods for setting/getting, which would have been simpler, but it was very difficult and less efficient

·

The public methods for interfacing with the Game Manager are:

void	**clear**() remove and cleanup the existing game
void	**initialize**() initialize the game
void	**processFrame**() run updates on every sprite and every condition
void	**setGameSpeed**(double speed, boolean play) change the game speed
void	**togglePause**() toggle the play/pause state of the game timeline

GameObjectConditions are the primary Condition class, they are constructed with a list of sprites and events

- public GameObjectCondition(java.util.List<Action> myActions,
  java.util.List<GameObject> myGameObjects)

The public methods for interfacing with the ButtonConditionManager, the manager responsible for button press events are:

void	**addBinding**(javafx.scene.input.KeyCode key, **Action** action) bind a key event to an action object
void	**beginListeningToScene**(javafx.scene.Scene scene) tell the button condition manager to begin listening for key presses in a particular scene
void	**clearAllBindings**() clear all of the bindings
void	**frameElapsed**() method to call to increment the frame counter for conditions that are frame based
void	**removeBinding**(javafx.scene.input.KeyCode key) remove a binding for an existing key

Actions have different constructors based on their type, but they all share the ability to execute a defined action

void	**execute**() Will execute the consequences resulting from applying an Action.

The Game Object is the root object for anything that appears on the screen

void	**disable**()
void	**enable**()
boolean	**getCollisionConstant**()
Component	**getComponent**(java.lang.String iD) Temporary Map based getter...
java.lang.String	**getCurrentImageName**()
java.awt.geom.Point2D	**getDefaultPosition**() Gets the Position of Sprite
double	**getHeight**()
java.lang.String	**getID**()
double	**getOrientation**() Gets the Orientation of Sprite
PhysicsBody	**getPhysicsBody**()
java.awt.geom.Point2D	**getPosition**() Gets the Position of Sprite
RenderedNode	**getRenderedNode**()
double	**getTranslateX**()
double	**getTranslateY**()
double	**getWidth**()
javafx.beans.property.DoubleProperty	**getXPositionProperty**() Deprecated, all transforms are performed on the node Gets the x position property of the sprite (for listeners)
javafx.beans.property.DoubleProperty	**getYPositionProperty**() Deprecated, all transforms are performed on the node Gets the y position property of the sprite (for listeners)
boolean	**isEnabled**()
java.util.Iterator<**Component**>	**iterator**()
void	**saveCurrentState**()
void	**setCurrentImagePath**(java.lang.String imageName)
void	**setDefaultPosition**(java.awt.geom.Point2D position)
void	**setOrientation**(double orientation) Sets Orientation of Sprite
void	**setPhysicsBody**(**PhysicsBody** physicsBody)
void	**setPosition**(java.awt.geom.Point2D point) Sets Location of Sprite
void	**setRenderedNode**(**RenderedNode** node)

void	**setTranslateX**(double xCoord) Sets X-Coordinate of Object
void	**setTranslateY**(double yCoord) Sets Y-Coordinate of Object
void	**update**() Updates all components of GameObject TODO Check if necessary...

The game object contains a PhysicsBody which holds physical information for the game object:

double	**getCollisionBodyHeight**()
double	**getCollisionBodyWidth**()
Mass	**getMass**(double y) Return Y-Coordinate of Object
void	**getPositionChange**(**GameObject** sprite)
Velocity	**getVelocity**() Return X-Coordinate of Object
void	**handleCollision**(**GameObject** thisSprite, **GameObject** sprite)
void	**setMass**(**Mass** m) Sets mass of object
void	**setVelocity**(**Velocity** v) Sets Velocity of object

The game object also initializes a list of components which are characteristics of the game object

void	**addProperty**(**IProperty** property) Adds property to Component
void	**disable**()
void	**enable**()
boolean	**isEnabled**()
java.util.Iterator<**IProperty**>	**iterator**() Returns an iterator for properties
void	**removeProperty**(**IProperty** property) Removes property from Properties List
void	**update**() updates all properties of Components

Examples of components include the Attributes, ImageReference, and SoundReference classes.

Player

Player primarily represents the front end GUI elements of the game. Player stands between the Game Data and Game Engine in that it calls on the DataManager class to serialize or deserialize an overall game DataWrapper object, which consists of Levels, GameObjects, and Conditions. DataWrapper is further used to create a GameManager in the Engine - it is passed into the constructor of GameManager, where GameManager will deal with separating it.

Player is split into a PlayerView and PlayerModel. PlayerView holds the front end classes, and PlayerModel sets up the two-way communication between Data and the one-way communication to Game Engine.

The GUI is being constructed primarily through an FXML file tied to a JavaFX controller class. The controller allows onAction methods to be defined for UI elements. This allows for the code to be cleaner - each UI element does not have to be defined in the code behind and attached to the specific method. Instead, the controller class contains all of the methods. Furthermore, we are reducing the amount of Java code by using CSS for styling.

The PlayerModel holds multiple instances of DataWrapper. One represents the default values/state for that game, and another one is for the current state of the game. This is meant to deal with reloading levels in case a character dies, or the timer expires, etc. It is also to contain the data in one location. Once Engine has pointers to these DataWrappers, they need not communicate back to Player in any other way.

The public methods for interfacing with Game Data are:

```
/**
* Get data from file through DataManager
* @param fileName Name of Json file.
* @return Object representing game.
*/
public DataWrapper getData(String fileName)
```

```
/**
* Write data to file through DataManager
* @param DataWrapper object representing game, fileName Name of Json file.
```

```
*/
public void setData(DataWrapper gameData, String fileName)
```

The public methods for interfacing with Game Engine are:

```
/**
* Get data from file through DataManager and create GameManager with this information to make the general game. ** always happens first
public void loadGameFile() */
/**
* Get specific progress data from file through DataManager and create GameManager with this information to make the general game.
public void loadProgressFile() */
```

Data

DataManager
The DataManager class is what primarily handles data reading and writing. It receives authored game file data from the authoring environment and serializes it to a JSon document. When the game needs to be loaded in order to play, the DataManager class is called upon to read a JSon file and output a wrapper class containing all the data given. The DataManager class also can save and read saved gameplay files. The following are the public methods in DataManager.

```
/**
* Gets an object representing a game from a Json file.
* @param fileName Name of Json file.
* @return Object representing game.
*/
public DataWrapper readGameFile(String fileName)
/**
* Creates a Json file based on an object representing a game.
* @param obj Object representing game.
* @param file Name Name of Json file.
```

```
 * @return Returns true if successfully writes file.
 * @throws IOException
 */
public boolean writeGameFile(DataWrapper wrapper, String fileName) throws IOException
/**
 * Gets an object representing a progress state from a Json file.
 * @param fileName Name of Json file.
 * @return Object representing progress state.
 */
public ProgressObject readProgressFile (String fileName)
/**
 * Creates a Json file based on an object representing a progress state.
 * @param obj Object representing progres state.
 * @param fileName Name of Json file.
 * @return Returns true if successfully writes file.
 */
public boolean writeProgressFile (ProgressObject obj, String fileName)
```

GenericTypeAdapter
This class is used to serialize and deserialize an object's specific subclass even if it is declared as its superclass. This class implements JsonSerializer<Condition> and JsonDeserializer<Condition>. It has the following public methods.

```
@Override
public JsonElement serialize(Condition src, Type typeOfSrc, JsonSerializationContext context)
@Override
public Condition deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context) throws
JsonParseException
```

Utilities

IN PROGRESS:
Controller Utilities:

- Xbox Controller
- Playstation Controller

- Xbox Kinect - This will use the J4K library to create Events for different Kinect actions that can be bound to keys similar to keyboard keys. For example waving your hands in different directions can be interpreted as different directional "keys"