Team: Bits Please

Vooga Salad Plan

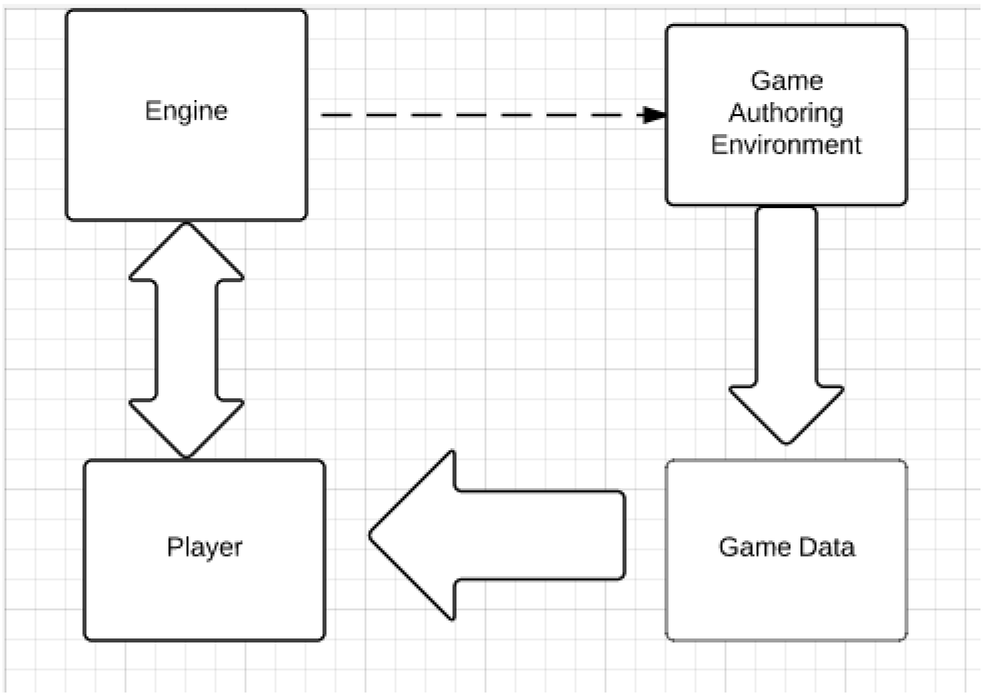
Members: Abhishek Balakrishnan, Shreyas Bharadwaj, Chris Bernt, Will Chang, Davis Gossage, Safkat Islam, Arihant Jain , Arjun Jain, Kevin Li, Eli Lichtenberg, Benjamin Reisner, Wesley Valentine

- **Genre: 2D Scrolling Platform**
  - Games of this genre are viewed from the side (where movement occurs from left to right) or top down (bottom to top). These games are scrolling where the movement of the scene may be forced, or based on the user's movement. It is a game that has a single screen movement viewing box Platform games involve player movement (such as moving, jumping, and/or fighting).
  - Active control of one unit (main player/character) per person.
  - Closed levels vs. open worlds: linear progression.
  - Navigating obstacles

- **Design Goals**
  - Concept of every game action being implementable with actions and events
    - Example: player moves up ladder
      - Action is collision between player and ladder and an 'up key'
      - Event is the player ascends
    - Simple example: collision between player and floor (or another solid object) results in the player stopping and rebounding, if applicable.
  - Authoring a game and playing a game are completely separate activities.
  - Each character in the game is definable by a series of properties, including physics characteristics, visual characteristic, and responses to events. The properties associated with a certain character are composed by the user in the authoring environment.
  - A "Game" that can be saved and loaded is comprised of all of its levels, events, and characters.

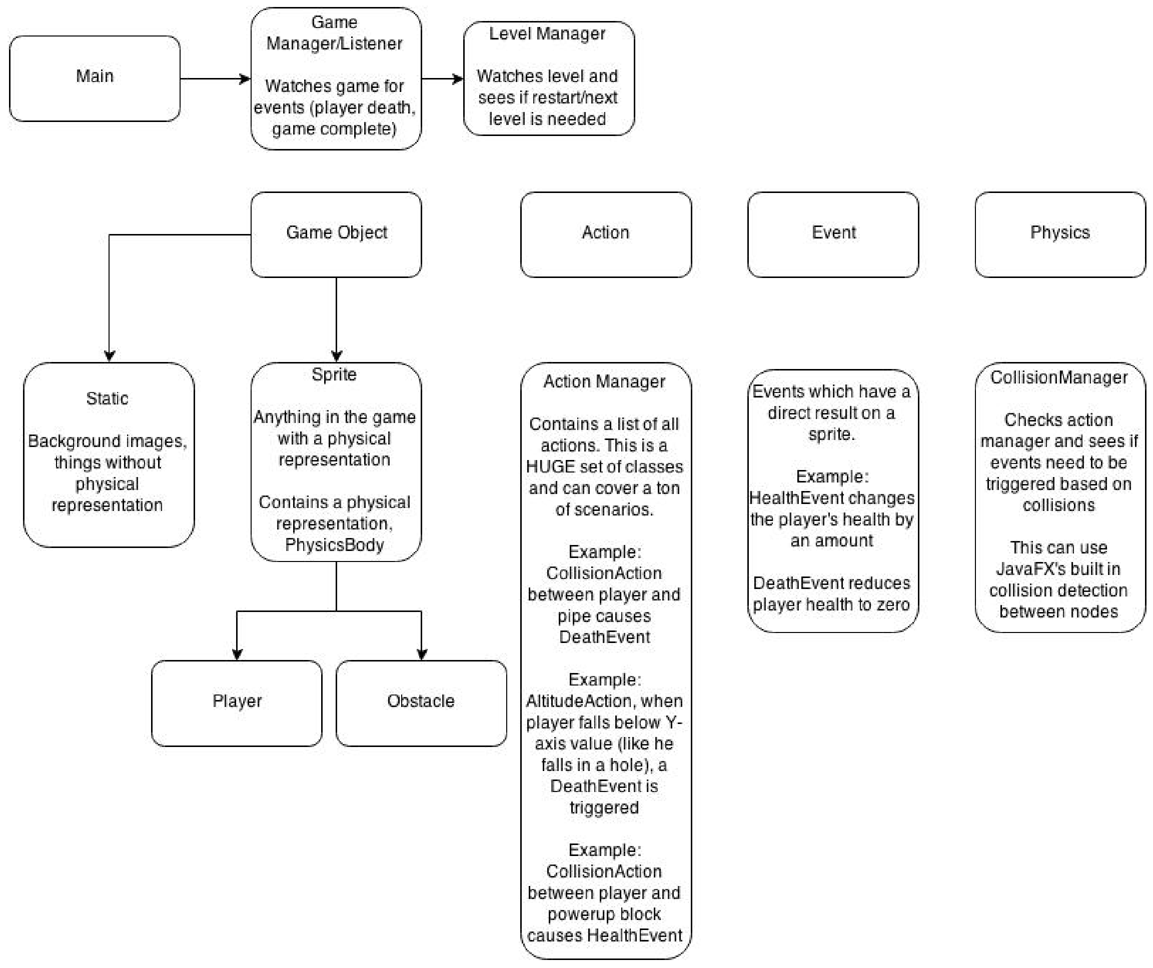
- **Primary Modules and Extension Points**

General VOOGASalad UML Design



- **Game Engine -** Serves as a reference library for the authoring environment. Also performs all computation on objects and runs the game loop
    - Game engine contains the primary Game Objects, which includes the Sprite hierarchy (enemies, NPC's, the player, platforms, blocks)
    - The Action and Event modules are used to compose a Sprite or any other Game Object, defining behavior, and gameplay controls.
    - Actions are any sort of interaction that may occur in a game; These are a set of classes that have conditions. The underlying principle of this system is if action, then event.
    - Events will be the effects of those actions determined by the appropriate response. For example, two sprites colliding would be the action, the event would be the way the players interacted afterwards.
    - The Physics module determines consequences of Game Object interactions and the physical rules of the game world.
    - A GameManager as well as a Level Manager coordinate updates for all game objects and maintains a current level as well as transitions between different levels in a given game.

Game Engine Module

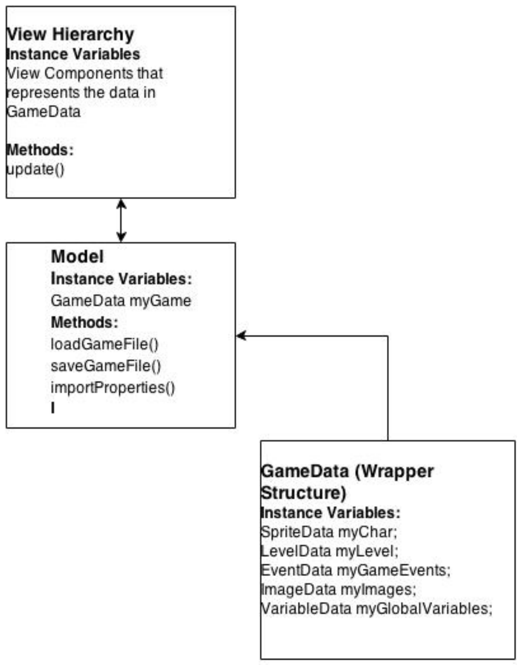

Main
Game Manager/Listener
Watches game for events (player death, game complete)
Level Manager
Watches level and sees if restart/next level is needed
Game Object
Action
Event
Physics
Static
Background images, things without physical representation
Sprite
Anything in the game with a physical representation
Contains a physical representation, PhysicsBody
Player
Obstacle
Action Manager
Contains a list of all actions. This is a HUGE set of classes and can cover a ton of scenarios.
Example: CollisionAction between player and pipe causes DeathEvent
Example: AltitudeAction, when player falls below Y-axis value (like he falls in a hole), a DeathEvent is triggered
Example: CollisionAction between player and powerup block causes HealthEvent
Events which have a direct result on a sprite.
Example: HealthEvent changes the player's health by an amount
DeathEvent reduces player health to zero
CollisionManager
Checks action manager and sees if events need to be triggered based on collisions
This can use JavaFX's built in collision detection between nodes

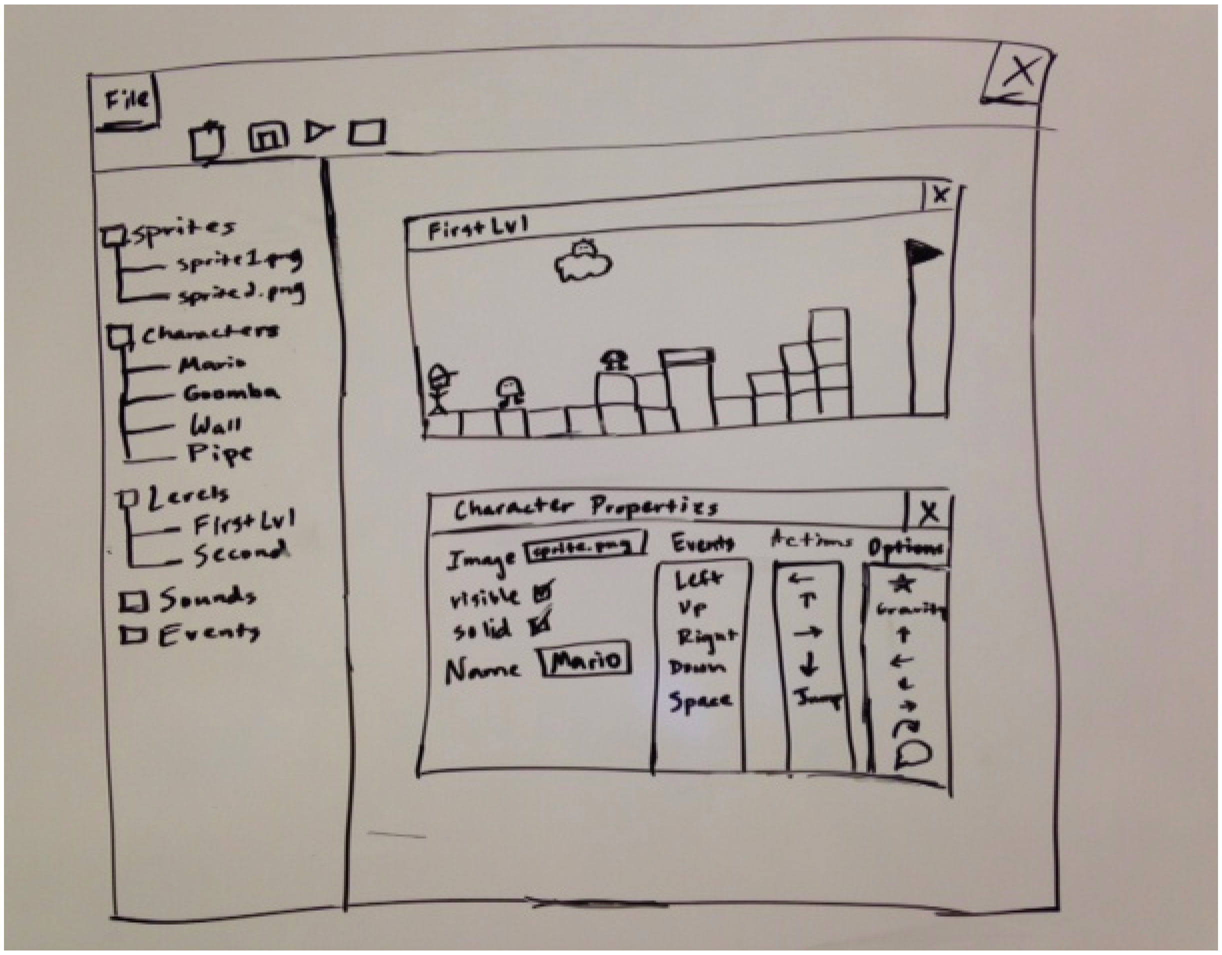
- **Authoring Environment -** Presents a UI to the user which accesses the Game Engine library to create instances of sprites / actions / events. The user is allowed to create new types of game objects and place them on the screen. This is achieved by composing properties from engine onto game objects.
  - The 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 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.
            - 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.

    - To be run, the game will be packaged into a wrapper object that represents the game, including its characters, levels, and events. This wrapper will be passed to the Data module.

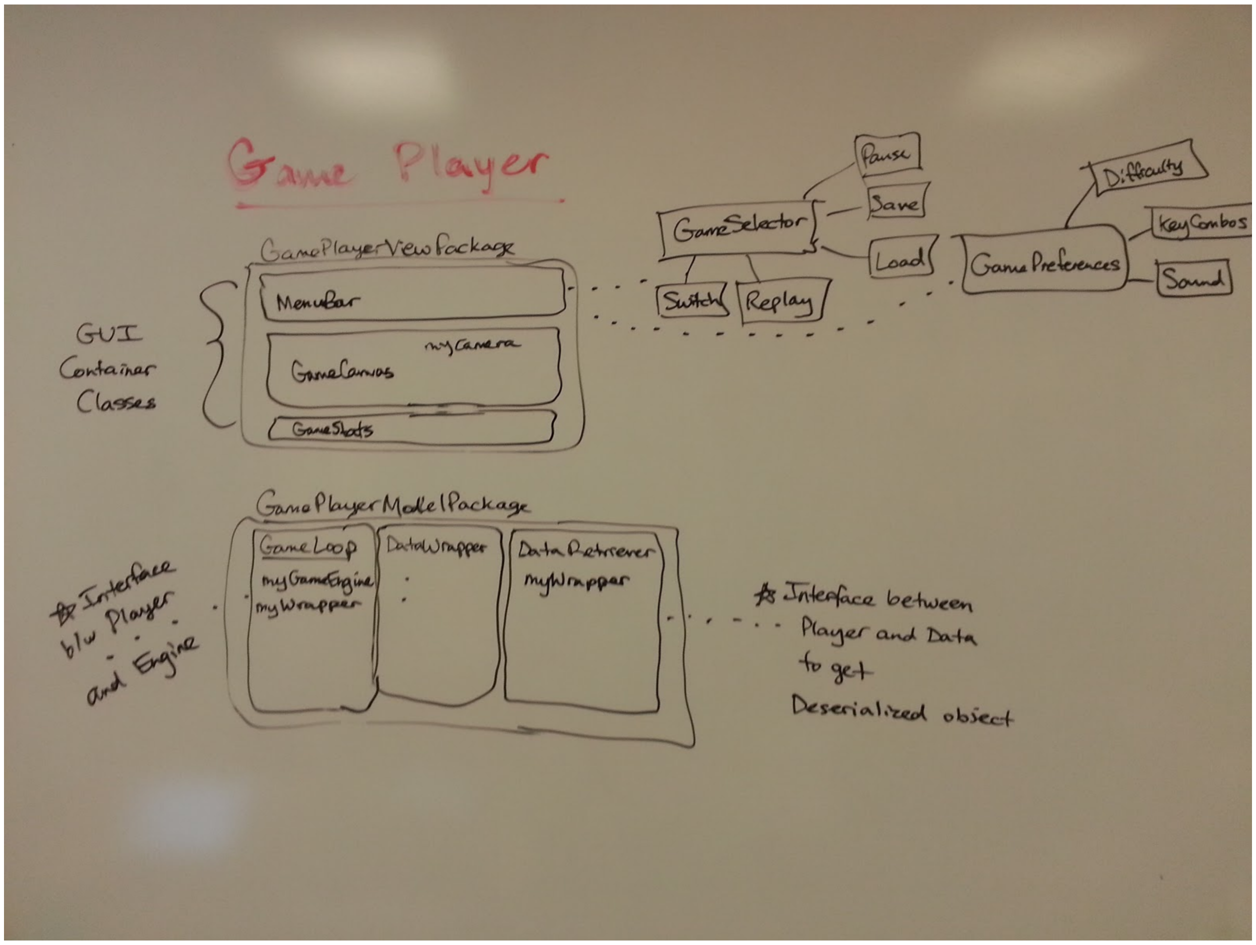
Authoring Module



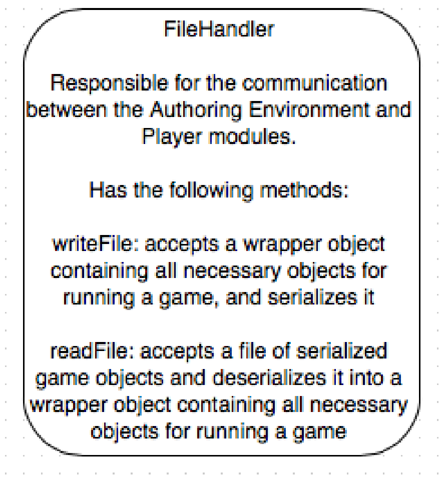


- **Player -** Accepts list of instances from data which are used by the engine to display the game. Holds and passes the current state of the game to the engine and accepts changes to be displayed and resolved.
  - Individual classes under a GUI hierarchy that represents options and elements displayed on the screen
    - View will include the GUI hierarchy
      - Control Menu
        - GameSelector
          - SwitchGame
          - Replay
          - Load
          - Save
          - Pause
        - GamePreferences
          - Difficulty
          - KeyCombos (define what keys map to which functions)

            - Sound - change volume
        - PlatformScrollerDisplay/GameCanvas - for displaying the actual platform game and the characters
            - Current Score
            - Time
            - HealthBar
            - Level
            - Lives
        - GameStats - this is broader than the context of the single game.
            - High Scores
            - Ranking
- Model
    - DataWrapper - This Wrapper object will contain all the data from the level, including Characters (including the player, enemies, and game play elements such as ladders, ropes, and walls). As the game progresses and more elements are created, this wrapper object will hold onto to the objects created by the Game Engine. This object will always have the current state of the game, which can also be saved when particular EventHandler is called.
    - DataRetriever - will load the deserialized Wrapper object from the Data module into the DataWrapper object.
    - GameLoop - This class will have an instance of DataWrapper and GameEngine. This class will handle the frame data and will contain the logic for any user input. The GameEngine will receive a reference to and act on the list of objects (the current state).



- **Data -**
  - Accepts list of instances from the authoring environment, where the objects are serialized and saved to a JSON file. Upon retrieval of the file, this data is converted back into instantiated objects where it is sent to the game engine at which point the game is running.



FileHandler

Responsible for the communication between the Authoring Environment and Player modules.

Has the following methods:

writeFile: accepts a wrapper object containing all necessary objects for running a game, and serializes it

readFile: accepts a file of serialized game objects and deserializes it into a wrapper object containing all necessary objects for running a game

- **Example**
- Three examples of platforming games that our program will be able to implement:
  - Super Mario:
    - Horizontal scroller
    - Jump and fall down with gravity
    - Enemies and powerups
      - Enemies are a subclass of the main sprite class which contains certain customizable options. Examples of options would be health, damage coefficient, and jump style. The sprite would also have physical properties including mass, spring constant, and velocity. Actions related to the enemy would be implemented using the action object, for example a collision between the player and enemy would result in a death event.
      - Powerups would be visually represented as a sprite. The action of the collision between the player and the powerup block would trigger an event, for example a health increase event.
    - Some levels will have auto-scrolling
      - There will be a specific camera class.
    - Some levels will have that the player can't go backwards
      - This will also be controlled by camera.
    - Tunnels/Distinct paths
      - Special sprite with event handler such that when Player interacts or collides, initializes a transition sequence to a different location/level in the Level Manager
    - Item Boxes
      - Subclass main sprite, give it an Event Handler such that when Player collides or interacts with the Box body, the event is triggered and an item is generated in the game level and displayed.

        - Subgoals to progress through level (eg, key to unlock door)
            - These will also extend the Sprite hierarchy which contain Event Handlers which initialize a sequence in the Level Manager, which changes state within the current level to allow the user to progress further into the game.
        - Specific levels
            - Authoring environment can specify what to add into new instances of Levels, and will be linked together by transitions in the Level Manager.
        - Story/Dialogue
            - Might have to incorporate cutscenes and multiple levels in a linear progression
        - Interactive obstacles
    - Fall Down:
        - Vertical Scroller
            - Camera Javafx Class will change the view on the games as player moves his sprite.
        - Frame boundary has an effect on character
            - Game Engines Level Class will determine boundaries and handle player position and
        - Randomly generated levels
            - one possible way to do this would be to generate at most [(width of screen)/(minimum size of a gap) -1] platform objects in [(width of screen)/(minimum size of a gap)] spots
            - maybe generate so that there are only 2 or 3 gaps, placed in random places
        - Wrap around
            - This will be a component of an action and event class. If a sprite hits a wall, it triggers a collsion event, that calculates the player's position after crossing an edge.
        - Infinite/Continuous Scrolling
            - This requires two things, first that the game randomly generates the platforms
        - Gravity or Constant speed
            - This will use the physics properties (mass) within the sprite's class. Then there is a universal gravity constant for the level.
        - Minimal controls
            -
    - Speedrunners:
        - Competitive scroller with multiple players
            - Multiple instances of Player objects that can interact with each other through the Game Engine
        - Forks in the road, and track is circular but still linear progression

        - These are just paths made out of ceilings and floors. Multiple paths give the character a choice of where to go.
    - Dynamic screen
        - Handled by Camera class.
    - Wall jumping
        - Combination of the jump key event and the collision between a wall and the player would trigger a jump event.
    - Other players can hit you
        - These other players would be enemies with their own properties. A collision between these enemy players and the player would trigger a health or death event
    - Competitive co-op and multiplayer
        - Multiple player instances that are individually controlled.
    - Weapons
        - Weapons would contain relevant action objects such as a key event action triggers a bullet sprite creation. The collision of this bullet with an enemy would cause a health event.
    - Physics and friction
        - If you are stuck on a wall, gravity will bring you down the wall but friction will decrease the velocity at which you fall
        - There are air tunnels that can push a player in a certain direction

General Platforming Elements:

- Obstacles include gaps, spikes, enemies (like Goombas), explosive boxes
- Elements include moving boxes (horizontal and vertical boxes) used for transportation, ladders, swinging ropes to jump across gaps
- Movement consist of jumping, rolling, climbing, swinging
- Powerups like invincibility, super size
- Sound events based on actions. Handled by Game Player

Alternate Design

- Player:
    - An alternate design plan was that the Player module was purely front end and that the Game Engine module would retrieve and store the data from the Data module. We decided to have the model of the Player module hold onto the data because philosophically the Game Engine should not be holding onto state. The Player module will pass current state to the Game Engine every frame or whenever a user input is registered.
    - Another topic of discussion was whether or not the level should be generated as the player scrolls through, or whether the level is defined first and the Camera just shifts its view as the player moves. We decided to have every game have a Camera that will change the view, while randomly generated levels as the player scrolls can be the

protocol for some types of games (like Fall Down) and defined levels for other types of games (like Mario).

- Game Engine:
    - Player front end, Game Engine backend with the game loop
- Data:
    - An alternate design for the Data module would be to use the instantiated objects to create a different file type, such as an XML file. Instead of simply serializing into and deserializing from a JSON file, the Data module can write an XML file based on what objects are passed to it, and then parse the XML file to recreate the same objects in order to pass them to the Player. Using XML, we would not have access to the GSON library to help create our data file, and we would need to write a more detailed parser ourselves.
- Authoring Environment:
    - Another possible way for model and view to interact is through a controller that contains all of the ways to get certain properties for anything displayed on the screen. That way everything that is changed on the view by the user can be propagated down to the model which contains the information associated with any object in the game. This can also work the other way, so that when a new game file is loaded into the the authoring environment, the data is deconstructed by the controller and represented in the view.

Team Roles

- Ari, Davis, Will, and Ben will work on the game engine module
    - Ben - Physics module
    - Ari - Sprites / Game Objects
    - Will - Events module
    - Davis - Actions module
- Shreyas and Abhishek will work together on the game player module, both the View and the Model
- Chris, Wes, Arjun, and Kevin will work on the authoring environment module
- Eli and Safkat will work on the game data module. Along with this, they will work with the game player and authoring environment modules to figure out the wrapper object that will be passed from authoring to data and data to player