Fighting Team API Presentation

Roles and Responsibilities:

View: Bill, Thomas and Wayne
Controller: Jack, Jerry, and Matt
Model (Game Instance, Maps, Sprites): : Alan, Jimmy, Dayvid
Physics: Wayne

Fighting Game Genre:

- Our genre consists of any 2D game that has a user controlled character (or multpitle user controlled characters) who interacts with his environment via movement and throwing attacks. T
- The basic distinctions that must be made are what constititutes a character, the inanimate environmental objects, and the animate environmental objects (AI).
    - The user controlled character therefore should be able to run on top of inanimate objects, but not be able to run on top of animate objects.
    - Conversely, he should be able to attack (and be attacked by) animate objects, but not be able to attack/ do damage to inanimate objects.
- There are many more subtleties to this situation, which unfortunately we have only began to scratch the surface off…

Original Design Goals:

- To create a Fighting Game API from which a game developer will have the freedom to create a variety of different styled fighting games (eg. SuperSmashBros, StreetFighter) ,
- To allow both single player and multiplayer (on a single console, not across the network).
- To easily support:
    - modifying or creating new characterobjects (fighting characters)
    - modifying or creating new modes (levels)
    - modifying or extending the game rules and goals for each level

A note about game rules and goals:

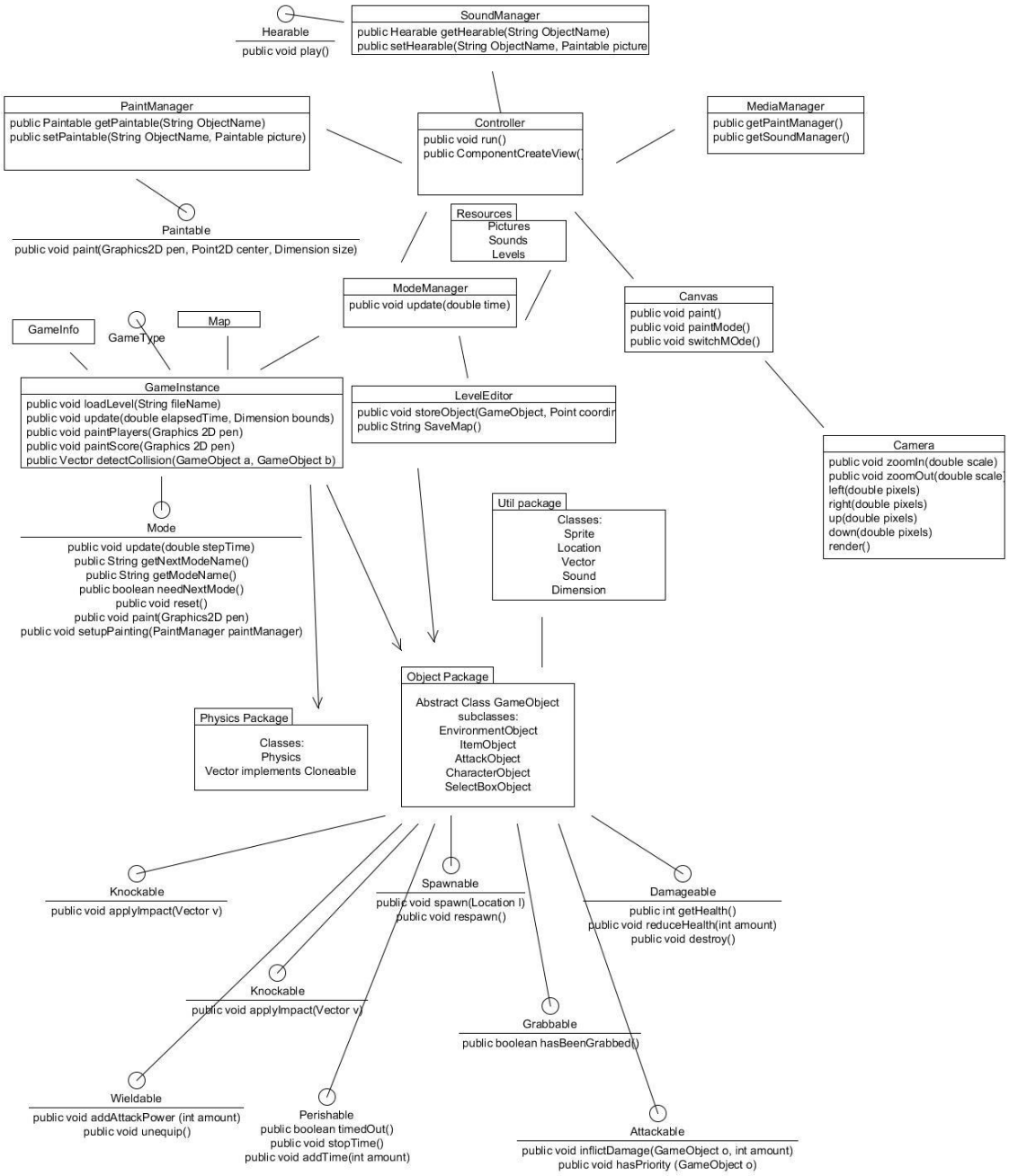
- Very dependent on the levels/characters that are created
- Levels were to "control" when to switch to next level
    - Therefore goals/ rules are very dependent on Game Developers choice in the implementation of a mode (level)

Design Problems:

- We decided to split our API up using the Model View Controller paradigm
- Unfortunately, we found out last night that our groups notion with this paradigm was highly inaccurate, and therefore our original conception of this framework was highly flawed…

Our Original Design:

- The Controller Module:
    - regulated the interactions between the View and the Model
    - encompassed one game loop
    - this game loop cycled through different modes
        - Modes are an interface for cumulative state in the model you would like to pass to the view to paint, including the MainMenu and the levels (GameInstance class)
    - Regulated which mode the View had access to, as well as which mode was updating, based on certain mode interface methods
    - Held the Input, but just passed this to the Mode
- The Model Module
    - Completely responsible for handling and updating all of the state
        - While the game loop was in the controller, this simply called an update method in the model (a mode interface method), which updated the model based on its state, elapsed time, and user input. (think our first games)
    - Held all (or at least) most of the input annotations and methods
    - Consisted of:
        - the GameInstances (think levels)
        - GameObjects (think sprites)
            - Derivations of these game objects, like characters
    - These classes held all of the logic for how to update the game, and therefore held all of the game rules and goals
    - Had paint method to be called by the view
- The View Module
    - Had access to the current mode
    - Used this current mode's paint to display the game state
    - Could choose where to display
    - Also, could use Mode interface to get Player/Game information



Why this is bad design (and does not adhere to MVC framework!):

- From the Controller Perspective:
  - No choice over how the game loop runs
  - Forces Menu GUIs to run through game loop
  - Has no knowledge of Game Goals/ Rules
  - Sets up the Input but really doesn't do the listening
  - Does very little "controlling"

- From the Model Perspective
    - Does far more than just hold the game state
    - Has “control” over updating game state
    - Decides how game state is displayed
    - Has direct connection to the View (Not very shy)
- From the View Perspective
    - Has little choice over how to display game state
    - Talks directly to the Model
    - Really is just a glorified Canvas (maybe not even glorified?)
- From a Game Designer Perspective
    - Very little control over the running of the game, all really contained in the controller
    - Game rules in the Model means a lot of coding to be done in the Model, as this holds game state and game logic
    - Direct contact between Model and View means these are not truly separate components, but codependent.
    - No extensibility to the View outside our precise notion of a Canvas that calls a paint method

Updated (rough) Design:

- The Controller Module:
    - regulates the interactions between the View and the Model (But differently!)
    - One View Controller framework for each GameInstance
        - These “subcontrollers” contain individual game loop, or perhaps the GUI framework, giving Game Designer freedom to subclass these subcontrollers, taking advantage of our given game loops
        - Each subcontroller contains game logic, as well as the update methods to modify Model State
        - Each subcontroller listents to the inputs and uses its game logic to call Model API methods to change model’s state
        - Each subcontroller extracts the relevant state from the Model for painting, and passes this to the View using the Abstract Interface PaintingInfo (which has methods like numberofobjects() objectcoordinates(int), objectpaintables(int), which return an int, a location and a paintable object (think pixmap) respectively.
        - Note: therefore addons like AI will have logic in these subcontrollers (or classes contained in these subcontrollers)
    - One GameController class to encompass the game, with the run() method, as well as the getInstance() method
        - This Game Controller class has charge of the desired MVC instance.
- The Model Module

        - Completely responsible for storing all game state
        - Has API calls to modify this game state, but never calls these itself
        - Has API calls to extract painting information about this game state, but never calls this itself
        - Consists:
            - GameInstances (think levels)
            - GameObjects (think sprites)
                - Derivations of these game objects, like characters
- The View Module
    - Has getPaintingObject() to receive PaintingObject
    - Has render method to render the info of current painting object
    - Will be customizable (with a HUD object) in order to display Game/Player State info
    - Only speaks to the Controller Module

Why this is better design

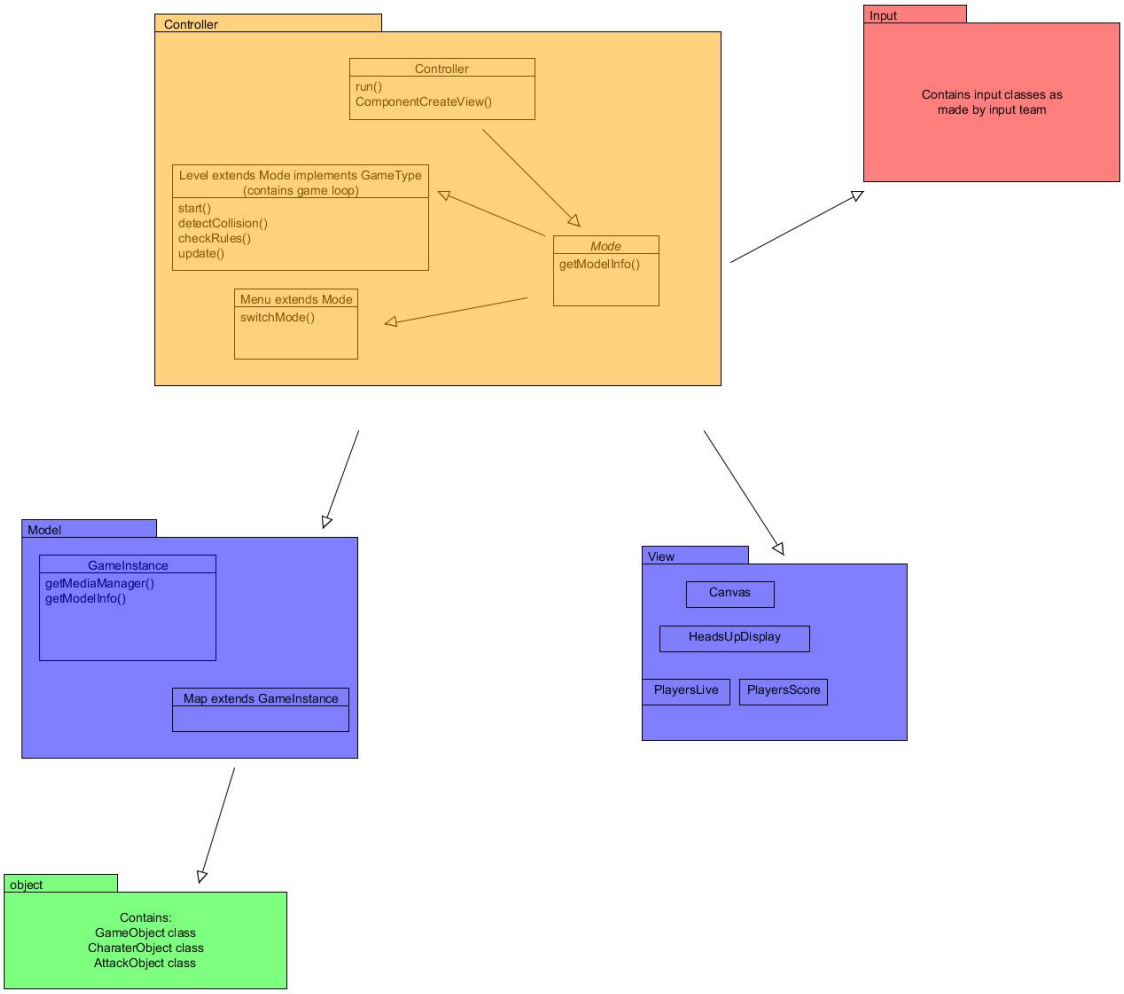
- From the Controller Perspective:
    - Gives the Developer choice over how the game loop runs
    - Has knowledge of Game Goals/ Rules, will provide superclass as well as a few subclasses to demonstrate how to create this game logic
    - Has control over how game state is modified, and gives this control to the Game Developer, while requiring them to write code in a massive GameInstance subclass
- From the Model Perspective
    - Holds Game State /Passes GameStat
    - Very easy modular functionality, making it easy to replace one Model with another model, keeping the same View/Controller
    - Only communicates to Controller based on general API calls
- From the View Perspective
    - Has more choice in how to display game state/ how to display the game in general
    - Only talks to the Controller via general API calls
    - Can be more than a Glorified Canvas!
- Other thoughts
    - Game rules in the subControllers, which can be subclassed to create easy game logic
    - Separate modules means choice in how to use each module, and can build different games using a similar view component (or Model Component)

Trade offs

- Requires the Game Developer to write a decent amount of code, based on the freedom they have to chose

- Will still have to make key decisions in what Model/Subcontroller/ ViewComponent classes we give them to subclass. The more we give them the less they have to do, but also, the less freedom they have (if they don't want to code too much!)

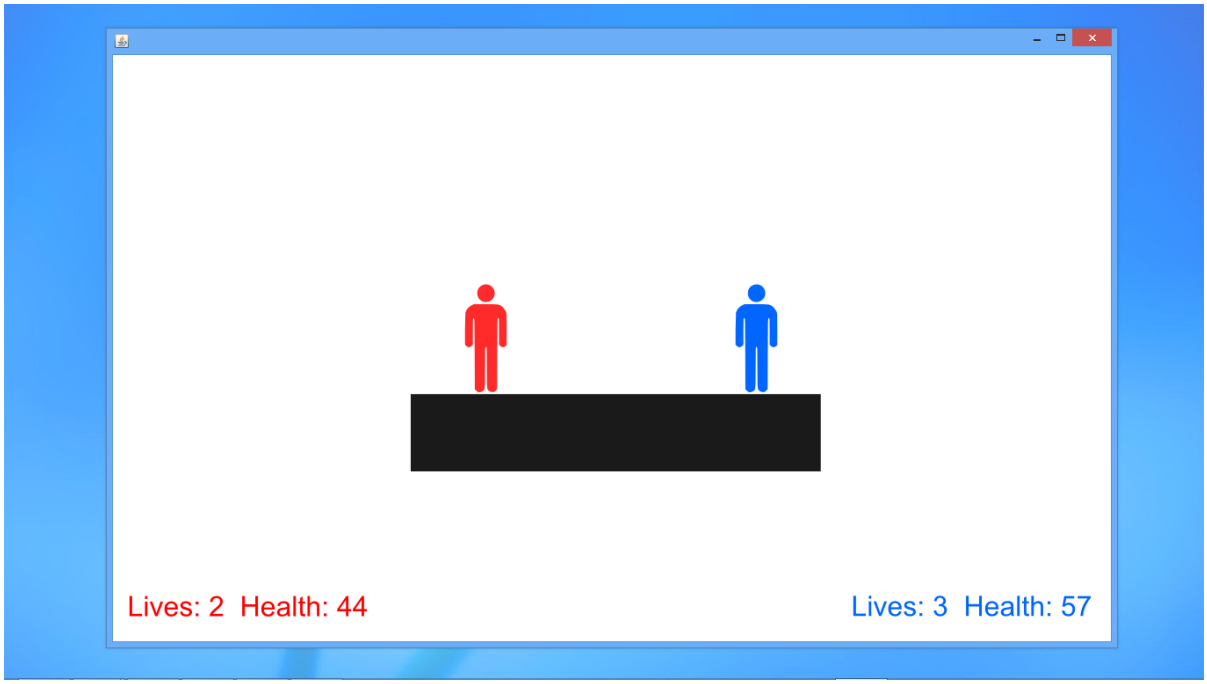
Rough UML



SubTeam Thoughts:

View: Bill, Thomas and Wayne
Controller: Jack, Jerry, and Matt
Model (Game Instance, Maps, Sprites): : Alan, Jimmy, Dayvid
Physics: Wayne

Sample view:


Lives: 2 Health: 44
Lives: 3 Health: 57