Team Print Stack Trace

Comp Sci 308 Final Project - VOOGASalad



[Contact Info](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.cpwh56ek2xhz)

[Calendar and Resources](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.z46xoghbnk0t)

[Design Document](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.mj8ttxszokg3)

[Genre:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.855jjr5qfhcj)

[Design Goals:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.d8v8bcdi2gqe)

[Primary Modules and Extension Points](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.2qps64jieexu)

[Game Authoring Engine](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.hbw2efgapi9c)

[Game Runtime Engine](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.xjiu9a216y7j)

[Game Data](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.n1ovnp9uhpkg)

[Example Code](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.3lrymr3gyovf)

[Alternate Designs:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.6p808pdufw3j)

[Team Roles:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.hki8labpyjth)

[Front End:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.u55yagws8dkk)

[Overview of Tasks:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.gpv3s6azxopf)

[Team Member Roles:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.n3c1t8n0q1k7)

[Back End:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.lpaw38sanv49)

[Overview of Tasks:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.nnelflrpw8ps)

[Game Authoring Engine Roles:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.pe4s3vbcjcdc)

[Game Runtime Engine Roles:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.8a1xkmejv92x)

[Game Data Roles:](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.kbolcxv21qrr)

[UML and Figures](https://docs.google.com/document/d/14uzSaHRBRBGfDwYF6fB-0FteUzsng0Dybf4Mh0NODL8/edit#heading=h.bhwivxmgvaba)

# **Contact Info**

* Daniel MacDonald
  + 336-501-8905
  + [daniel.k.macdonald@gmail.com](mailto:daniel.k.macdonald@gmail.com)
* Pranava Raparla
  + 703-732-7984
  + [pranava.raparla@gmail.com](mailto:pranava.raparla@gmail.com)
  + Keohane 4D Room 316
  + Net ID: pkr6
* Justin Carrao
  + 805-218-7273
  + [carrao.justin@gmail.com](mailto:carrao.justin@gmail.com)
  + Keohane 4D Room 316
  + NetID: jdc61
  + Unique ID: 0589888
* Jack Baskin
  + 254-744-6535
  + [jackbaskin93@gmail.com](mailto:jackbaskin93@gmail.com)
  + #Will be gone for final presentation
* Marcus Cain
  + [phswrestler125@gmail.com](mailto:phswrestler125@gmail.com) (imessage that email)
  + NetID: mac86
* Ashwin Kommajesula
  + 609-578-7828
  + [ashwin.kommajesula@gmail.com](mailto:ashwin.kommajesula@gmail.com)
  + NetID: ahk12
* Ethan Chang
  + 209-327-0291
  + echang18@gmail.com
* Nick Widmaier
  + 914-837-9816
  + [nickwidmaier@gmail.com](mailto:nickwidmaier@gmail.com)
* Zach Podbela
  + 516-776-4300
  + [zachary.podbela@duke.edu](mailto:zachary.podbela@duke.edu)
  + [skierzp@gmail.com](mailto:skierzp@gmail.com)
  + netid: zjp3
* Petra Ronald
  + 859-489-8091
  + petra2432@gmail.com

# **Calendar and Resources**

* Assignment Overview: <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/>
* 308 Calendar: <http://www.cs.duke.edu/courses/compsci308/current/classwork/>
* *Plan* Due: 8 AM on Friday, November 7th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part2.php>
* *Basic Implementation* Due: 8 AM on Tuesday, November 18th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part3.php>
* *Revised Plan* Due: 8 AM on Friday, November 21st, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part4.php>
* *Basic In-Class Demo* Due: 8 AM on Monday, November 24th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part5.php>
* *Complete Implementation* Due: 8 AM on Sunday, December 7th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part6.php>
* *Final In-Class DEMO* Due: 5 PM on Saturday, December 13th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part7.php>
* *Analysis* Due: 8 AM on Sunday, December 14th, 2014; <http://www.cs.duke.edu/courses/compsci308/current/assign/04_voogasalad/part8.php>

# **Design Document**

## 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

**Front End - Game Authoring**

The Game Authoring front end contains a visual minimization of the scene. From the sidebars, you can drag in sprites, platforms, and background images. Some pictures for this are included at the end of this document. The sprites include a keyframe editing option which is used to create custom animations. The reactions for collision are set in a table of dropdown menus which contain the permissible options (image of table at the end). The authoring environment also allows the user to add goals for the level, which can include both winning goals and game-ending/life-ending “goals.”

**Front End - Game Player**

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. It will start off with a splash screen. 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/quitting/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.

All the calls to the back end will go through our GameEngine.java class, which is the portal to the back end. Most methods will be pass through methods to the RuntimeEngine.java.

### Game Authoring Engine

In the backend, a sprite characteristics file will be sent and saved in the backend when an image is dragged onto the canvas. There is a map that will map an id assigned to the characteristics file, to the file itself, that way the the backend can deal with properties in the characteristics files modified in the front end.  We also have a level characteristics and goal characteristics class that are created in the front end and passed to the backend to define characteristics to define the level.

The game authoring environment’s main purpose is to populate all of the variables and states specific to each level of the game being created.  All of this information is saved in the level model class.  This class has maps for the level’s sprites and goals. Each map has an integer key that the front end uses to keep track of the different sprites and goals without having to deal with the actual objects themselves.  The level characteristics class, camera type, and physics engine are also held by the level model. Once editing is completed, this level model object will be passed to game data to be saved.

To deal with goals, we implement the visitor design pattern. To implement this, we have all goals implement a GoalElement object containing an accept method, which accepts a GoalChecker object. This GoalChecker object contains method to deal with each type of subclass for the goals. This makes it so the RunTimeEngine doesn’t need to know what subtype of goal it takes in. It just needs to call the goal’s accept method and give it a levelChecker to check if the goal is finished. This prevents us from needing to use an ugly series of conditional statements. We have, however, begun to realize that this may be slightly overkill for the project as our goal subclasses no longer have unique behaviors. As a result, we may look into modifying this in the future.

Additionally, 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.

For API refer to UML documentation at the end of the document.

### Game Runtime Engine

The backend of the Game Player will live in the RuntimeEngine class, which will receive all its calls from the Game Engine pass through class. 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 scene with new images, new levels, new scores, etc. The backend (Runtime Engine) will be responsible for making changes to that data. This is will we will keep the two ends independent. This design will allow for changes to be made without extensive writing of complex code on the other end. With any project, we want the Game Player to be designed well to allow future extensions so that the code is scalable and resilient to changes, and we want our code to be testable. With this size of the project, we need to ensure that we have testable subsets of code whenever changes are made. Testable subsets of code will ensure that the fault domain of changed code will be explicitly known, making the process of debugging relatively straightforward.

The flow of our calls will be from GamePlayer >> GameEngine >> RuntimeEngine >> LevelModel, GameWorldModel, LevelManager, associated Sprite/Goal classes, etc. When the front end Game Player needs to redraw the scene, it will make a getStatus() call to Game Engine which passes through to Runtime Engine. In each frame, the Runtime Engine updates the LevelModel class which holds all the goals, sprites, etc. and then generates a RuntimeModel with RuntimeSprites that is passed back to the front end. We will also have a Viewport/Camera class that passes back only the objects that are in the current scene so that the front end doesn’t have to redraw an entire level each frame, just the specific objects in the existing view for optimization.

Updating each level will involve the following: each sprite gets its own SoloPhysicsHandler to update it’s location and vector information in each frame.

This is followed by a process of iteration through each of the sprites in the LevelModel and using CollisionDetector to see if the current sprite has intersected with any other.  In the event of an intersection (collision), we will employ our CollisionResults matrix to provide the outcome of each specific collision.

The Runtime Engine will also keep track of points, and scores, which can be saved via calls to the Game Data subsection of the back end. In each update, the Runtime Engine will update each sprites’ movements with the solo physics handler, check and handle for collisions, clean up dead sprites in the level, and use a camera/viewport to update

### Game Data

The game Data will have two main methods that will be called from the other modules. These methods are write, and read.

The parameters for the write(Object, o BufferedOutputStream bos) method are an object that the user wants to store as o, and the bufferedoutputstream is the outputstream that the user wants to write to. The bufferedoutputstream is used over another type of outputstream since it has better performance. This method will take the object o and attempt to use Gson to serialize it into a json representation. Another issue that will arise through the use of Gson is the call to gson will attempt to serialize the javafx instance variables. The best way to fix this is to label those variables as transient so that the do not get serialized. The bufferedoutputstream must be written to using a byte array, so after gson converts the objects to a string, a byte array that represents the string must be created which is then written to the file specified by the bufferedouputstream.

The other method in GameData is the read(BufferedInputStream bis, and Class<?> c). This method will use the bufferedinputstream to determine where to read the file from and then return a byte array which will then be converted into a json representation (a string). Then gson will use the generic type that is passed into the method in order to create a new instance of that class. For example, if a levelmodel is passed in, then the gson call will know to create a levelmodel object. This is then returned.

High score saving and loading are done similarly as above, except they will not be written to the same file as the actual game. They will be written to a file elsewhere that can be overwritten.

## 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

We considered a number of alternate designs throughout the design process:

1. We considered using a subclass hierarchy and a factory design pattern for Sprite creation and management. However, as we begin to develop the interactions between our Game Author Engine and other high scale components, we soon realized composition would be a better route than inheritance. Most of the components of the subclasses were shared and having subclasses masked the scope to the super class anyway. So in the end we decided to go create the SpriteCharactoristics class and simple use it for creation, management, and storage of sprites.
2. When we first attacked the problem of Goal checking, we passed a level model into the Goal subclass using the method check(RuntimeModel level). However, this was not ideal as it was not SHY and gave the Goal too much access. The entire level was not necessary to simply check if a single goal was completed. Instead, we decided to implement a visitor pattern where the visitor, an owner of the RuntimeModel (the level), would visit the Goal classes and check them, altering their isCompleted state if necessary.
3. For our basic implementation of saving and loading, we implemented multimethods for saving and loading levels by providing a LevelModel and a variety of mediums (a URL, a String representing a path, or a File). After realizing how limiting this was, we switched to allow users to pass in a BufferedInput or BufferedOutput. Being as they all accepted several different mediums for a constructor (including those named above), this allowed us to offshore most of the work of varying medium to their constructors. However, we then realized the design itself was limiting so we changed it to automatically save Levels to a user's working directory, with an additional “export” feature if they wanted to specify a path.

## 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 (Game Authoring Engine), creating the runtime game engine (Game Runtime Engine), and dealing with the game data (Game Data). We have split the back end members into three separate teams, one to handle each of these tasks.

#### **Game Authoring Engine Roles:**

* Ethan Chang: Creating sprite hierarchy
* Jack Baskin: Creating main collections for holding level objects
* Nick Widmaier: Creating level classes

#### **Game Runtime Engine Roles:**

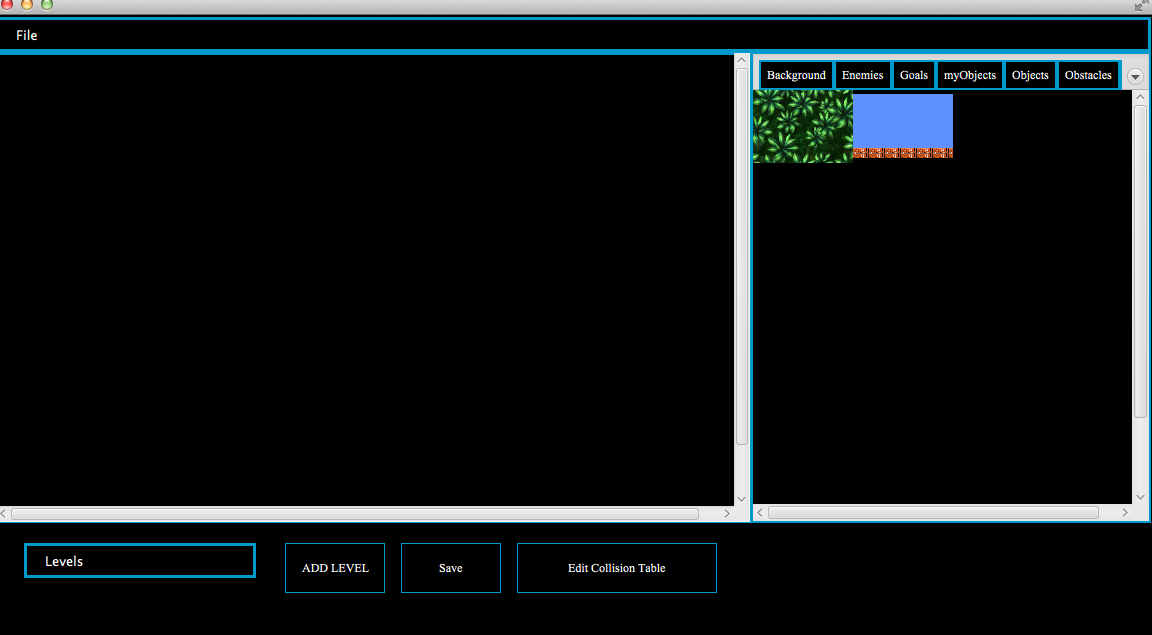
* Justin Carrao: Collision detection
* Zach Podbela: Physics Engine
* Pranava Raparla: Communication between game data and game engine

#### **Game Data Roles:**

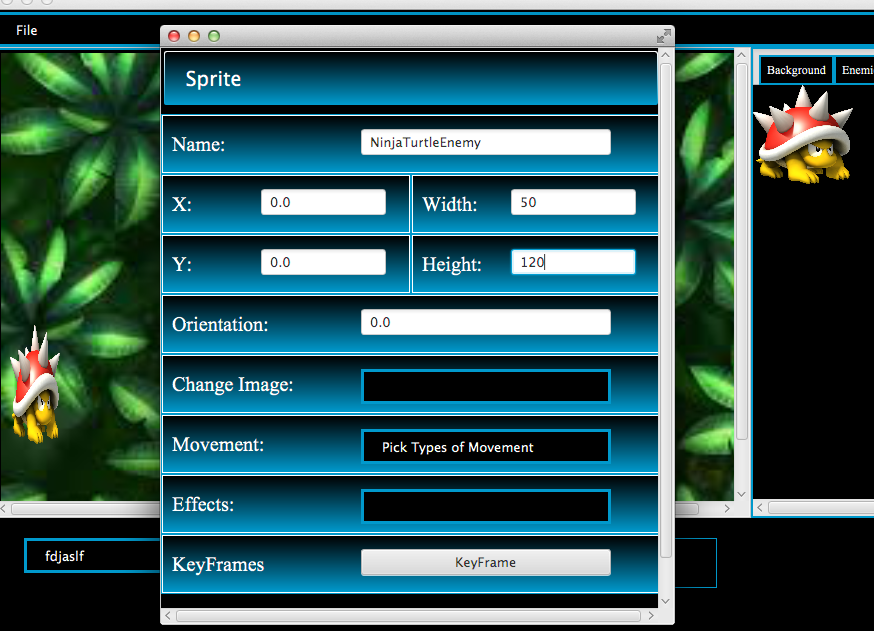
* Marcus Cain: Game data
* Ashwin Kommajesula: Game data

# **UML and Figureshttps://lh3.googleusercontent.com/3rQUy0uoghljUSq9Hy9SQopCxRZEy0MpfFJCTPvzGJ4iiSDcZ78rYknbF8cgCi7_le3EZjGn9KxfP3WDHYix0yQWsY_y2GKAvuIVp82KmSDhNMFW0GRLerPVxPeiMsJJiAhttps://lh6.googleusercontent.com/FjDTXJB3NjFpMlAVL75G7gjV_1W8IOyVCdKwuwIilCZ_R9iaTaDCSPP8XlhKHglWc5AbFSDONDksU08xAlkJl9dcKDqwR3MA3i80JH9QjPcugvLm_WH58CGW6S5IVxj31Ahttps://lh4.googleusercontent.com/-pud8eKrCs9MbZE4RhHQmdXkU9pUI11IWGyE8Qya2gYHS9bIJ3dYZHbZ5Uz8gH8gNxdxbf2bLXKxY_u33BF7sYukoRCDbbjVZG8PWWcb5xxrjK-EkqdqdmznAPJ0MqsMEghttps://lh4.googleusercontent.com/kS3ny0XkkcRJvsE8dDZmZvYWh1Z3xuaNf38-oZdYkkD2-g8qn8sNU5plHmj3uorZw2qeRt8iPFBP0Lvj71eiwJrY0hgjray0-50fs_3Y6SzOtTBI-hLWvXSbM7DjHD147w**

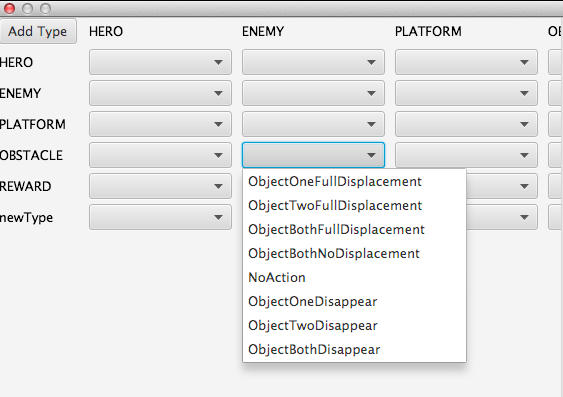
**Authoring Environment:**

****

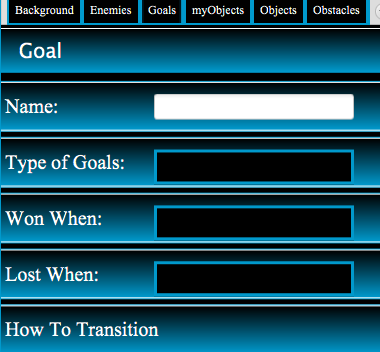
**Adding an enemy:**

****

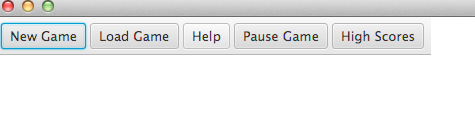
**Table for choosing collision consequences (note: more options will be added):**

****

**Adding a goal:**

****

**Game Player Buttons:**

****