**Bill Game**

**Members:** Andre Craddock, Mark Goodwin, Jeffrey Eldridge, Adrian Jacobs

**Abstract:**

Learning core concepts is essential for any programmer and those same core concepts can be applied to any discipline that requires problem solving. Because these concepts are so important, it would be ideal if they could be taught to adults and children in an engaging way. To that end we intend to create a game that will help to teach and reinforce these concepts, but whose priority is to be fun and engaging. The purpose of this game is to teach concepts of problem solving that transcend any programming language and the art of programming itself.

**Introduction:**

The project as a whole will consist of the game itself, a website, and a level builder. The game will require players to navigate through obstacles and then script a Non-Player Character (NPC) to navigate a set of obstacles that will be made up of problems the player has encountered. The website will provide information and some interactivity between users. The level builder will allow users to dream up and create their own challenging game levels, to be used by themselves in game or shared with other users.

**Approach:**

We will be building the game using Python and an OpenGL wrapper known as Pyglet. The website will be built with HTML, JavaScript, and will use PHP for server-side scripting. The level builder will likely be created as an interactive webpage on the website using HTML5 and JavaScript. We will be using git for version control of the project, as well as github.com to provide a central repository to synchronize our work. All of these tools are free, open-source, and cross-platform, as the project will be.

**System Profile:**

The project will consist of three different self contained entities:

**Website:** The website will be simple and consist of only a few pages: a page to download the game, a page to download the level builder, a page containing the user manual, and a page containing a high scores board.

**Game:** The game will be structured using the standard Model View Controller paradigm (MVC) and will therefore consist of three large modules (named model, view, and control) that are themselves broken into submodules. The model module will contain objects and functions for operating on and storing information about the game’s state. The view module will contain objects and functions used to actually draw the game’s user interface onto the screen. The control module will contain the engine and any components it needs to operate on the game state based on the rules of the game and interaction with the user.

**Level Builder:** The level builder will likely be structured the same way that the game itself will be structured: using the MVC paradigm and with three modules (model, view, and control) serving similar purposes. The main difference being that the point of the level builder is to output levels that are playable in the game based on user interaction.

**Software being used:**

**Python** - The programming language we will be using to write the game is python. Python provides an easy, clear syntax and is inherently cross-platform. This means that if we are careful in selecting our other tools our entire project will be able to be built and run on Linux, Mac OS, and Windows operating systems.

**Pyglet** - Pyglet is an OpenGL wrapper written for use in python. It provides all of the functionality and control one could find using OpenGL in C/C++, as well as easier to use interfaces with OpenGL. It has other useful features that will be conducive to creating our game, such as a sound-effect/music library and a built in event handling loop. Overall this means that we can use built in Pyglet features when we can to keep the code simple and concise, while at the same time delving deeper into the OpenGL library to get more performance when necessary.

**Git** - Git is a source code management system that offers a great deal of usability and features for assisting in project version control and synchronization. Git provides great support for non-linear development, which allows for an agile approach. It is also easy to use and has been ported across most major platforms. Git has many features that make it easy to trace-back code and find older implementations if anything ever goes wrong. Overall, Git will simplify the upkeep and building of code.

**github.com** - The web interface and server that host the Git repository. This website will increase accessibility and assist in continuing workflow. The website visualizes how the development of the project is going by showing the current branches of the files in the project.

**Zend Server 5.6** - Zend Server is a complete, enterprise-ready PHP Web Application Server for running and managing PHP applications. A useful tool that will hold our database. It will be our web server that will store high scores.

**PHP** - The programming language that will be used to do server-side scripting and database accessing. PHP has built in features that makes it excellent for integrating databases into websites and also scripting servers to perform task.

**HTML** - Used for the foundation of the website that will be hosting the leaderboards and level builder.

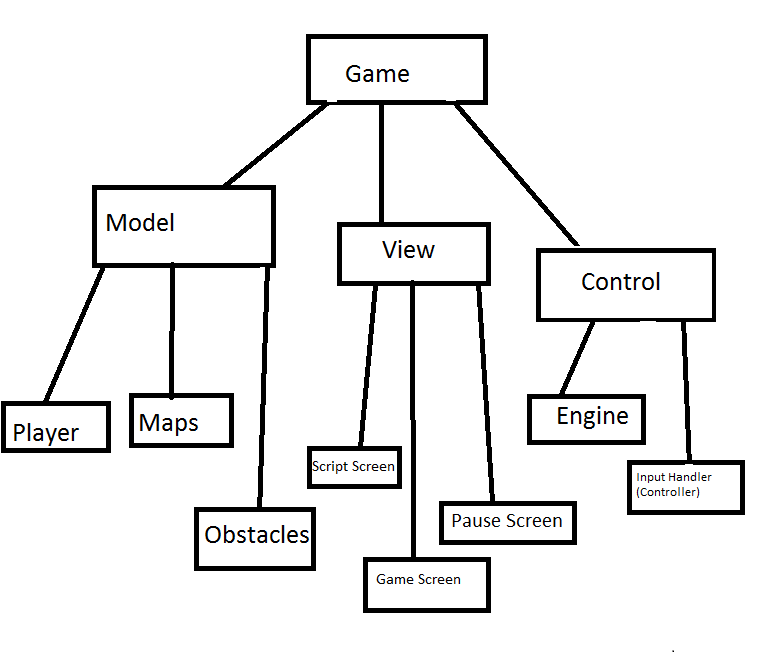
**JavaScript** - Being used to create and run the level builder and to make the website experience as a whole smoother and more user-friendly.

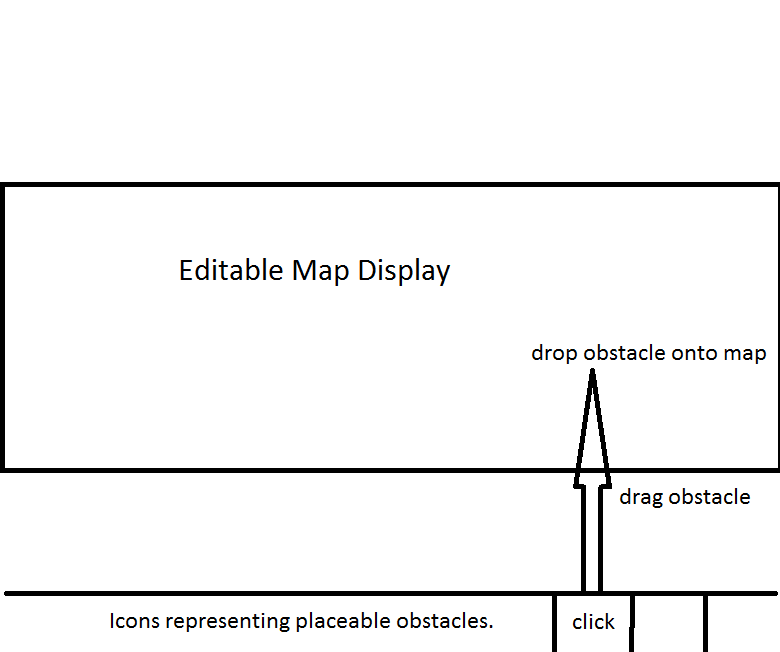
**Feasibility and Draft Models:**

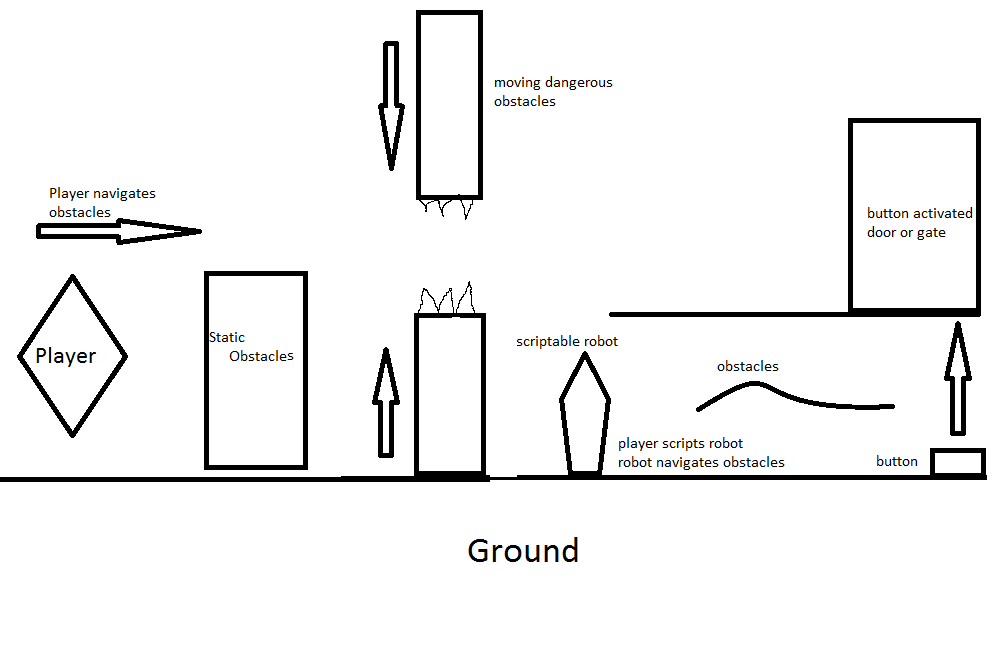
The creation of a game is a large undertaking. Most games produced by major studios take years to create, with whole separate teams assigned to art, sound, design, and programming. Even so called ‘indie’ games (which often have more simplified graphics and sound, and are produced by smaller teams) can take more than a year to produce. Super Meat Boy, a game released on Steam and the Xbox Live Arcade by a very small studio (Team Meat), took almost two years to be completed as it exists today, despite its 2-dimensional graphics and simple gameplay.

Regardless of the seeming enormity of the task, the tools that we have selected should allow us to complete our project at a fast pace. Tools like Python and Pyglet are conducive to quick development, and allow us to focus more on the flow of the program and the design of the game, and only delve into low level graphics manipulation when necessary.

By ordering our tasks correctly, completing certain modules will allow us to quickly build other modules. We will first focus on completing the game’s engine, renderer, and basic model prototypes. Once this groundwork is done, we will be able to begin working on the level builder and this will allow us to rapidly develop levels for the game. Once the basics in the game are working and we have levels to test our models and interfaces, we will be able to add new models and interfaces more rapidly. The website will likely be designed in parallel with the other parts of the project, since it is not dependent on progress milestones with the game or level builder.

Structure of game source code

Concept for level builder

Concept for a single level of a game

**System / Algorithm Analysis:**

**Models -** Models are “plain dumb objects”. In general they only store data and algorithms associated with modifying themselves. Players, maps, objects, mechanisms, and enemies will be in the model module. These classes will hold positional, image, and other state data related to the objects themselves, and only the objects themselves.

**Views -** Views will be modules and objects associated with rendering the user interface. Renderers for the gameplay and menus will be stored here. Any modules stored here will actively carry out the drawing of the user interface and will be called directly by Pyglet’s *on\_draw* function.

**Controllers -**  Dictates and issues commands to the models in accordance to what the player indicates. The engine is located in this module. The engine is used call update functions to update the overall game state, buttons pressed, screen location, and the positions of objects. Does all of the computation for forces such as gravity, and updates the position of objects after gravity has affected it.

**Data Dictionary:**

**Model:**

**Player:**  This is the object that holds all the information concerning the player.

*Fields:*

x - The position of the Player on the x-axis.

y - The position of the Player on the y-axis.

MAX\_DX - The max position the player can move on the x-axis.

MAX\_DY - The max position the player can move on the y-axis.

dx - The how fast the player is moving on the x-axis.

dy - The how fast the player is moving on the y-axis.

*Functions:*

draw(self): produces an image of the player for the renderer.

**Controller:**

**Player\_Controller:** Handles user input that controls the player during gameplay.

*Fields:*

keys\_down - stores the currently pressed keys.

*Functions:*

on\_key\_press(symbol, modifiers): sets keys pressed to “True” in the keys\_down field.

on\_key\_release(symbol, modifiers): sets keys released to “False” in the keys\_down field.

handle\_input(player): changes the player’s state based on what keys have been pressed.

jump(player): causes the player to jump.

**Engine:** Handles the main functions of the game including updating the game state.

*Fields:*

FLOOR - the lowest possible x-axis the character can go

GRAVITY - the constant that will slow or speed the rate of the character

moving

*Functions:*

update(self, dt): updates the position of all models for next iteration of

game loop

gravity(object): adds gravity to the object, either increasing or decreasing speed

**View:**

**Renderer:** Draws the game models and graphics to the window

*Fields:*

engine: a reference to the games engine used to gain access to objects to

draw.

*Functions:*

render(self): Calls the draw function of all objects to be drawn into window that update cycle.