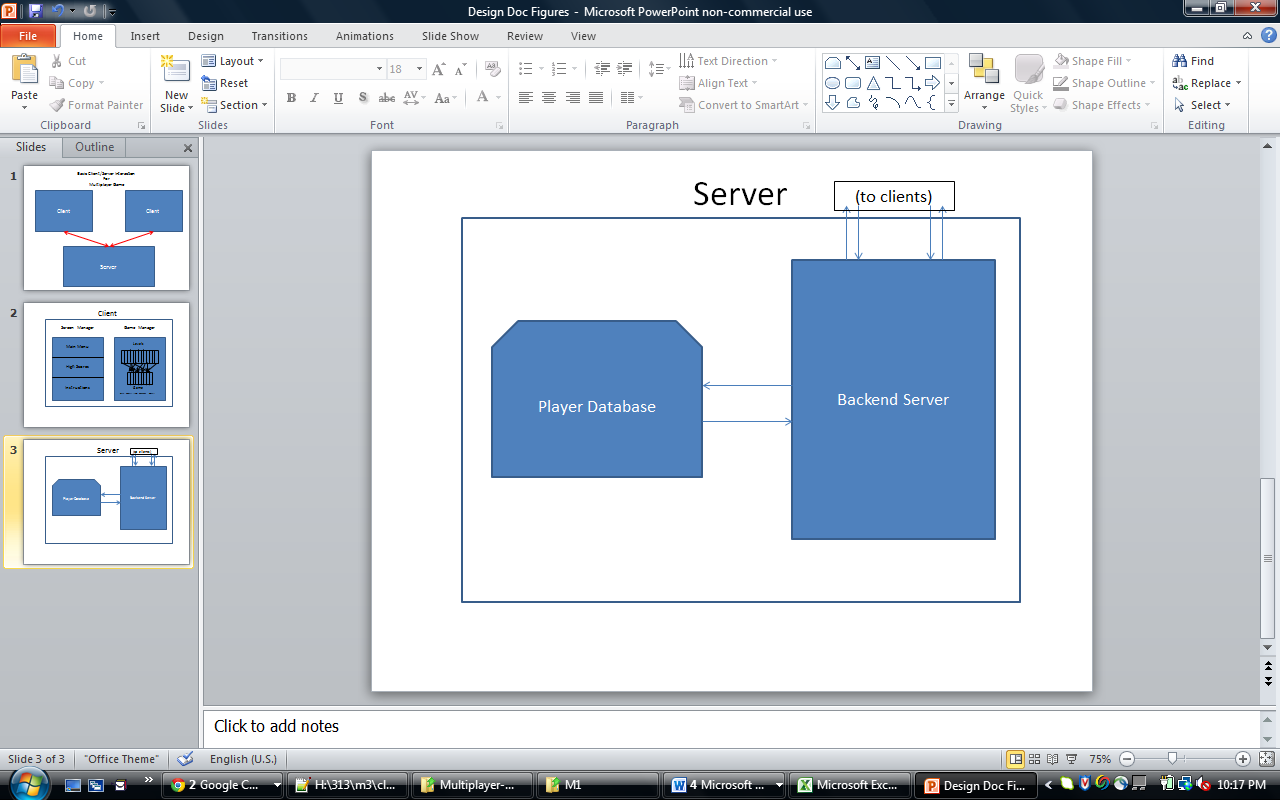
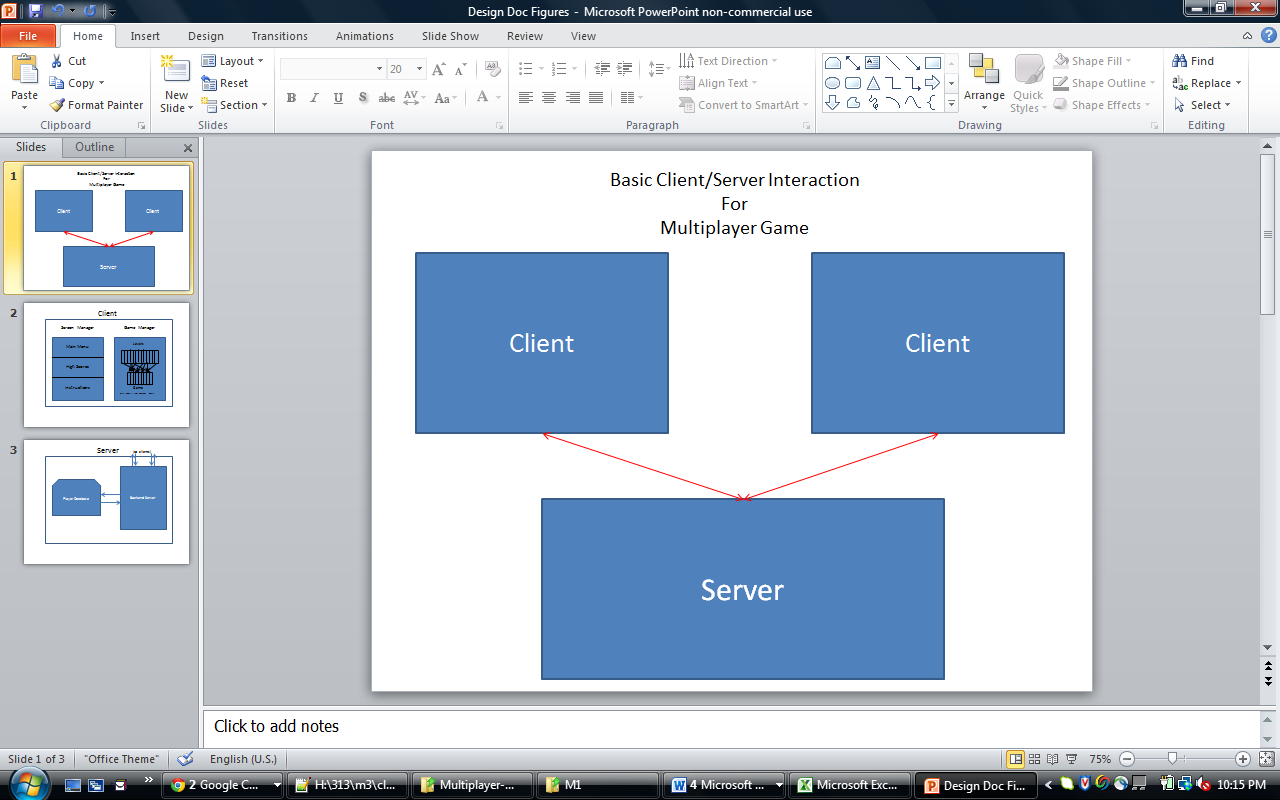
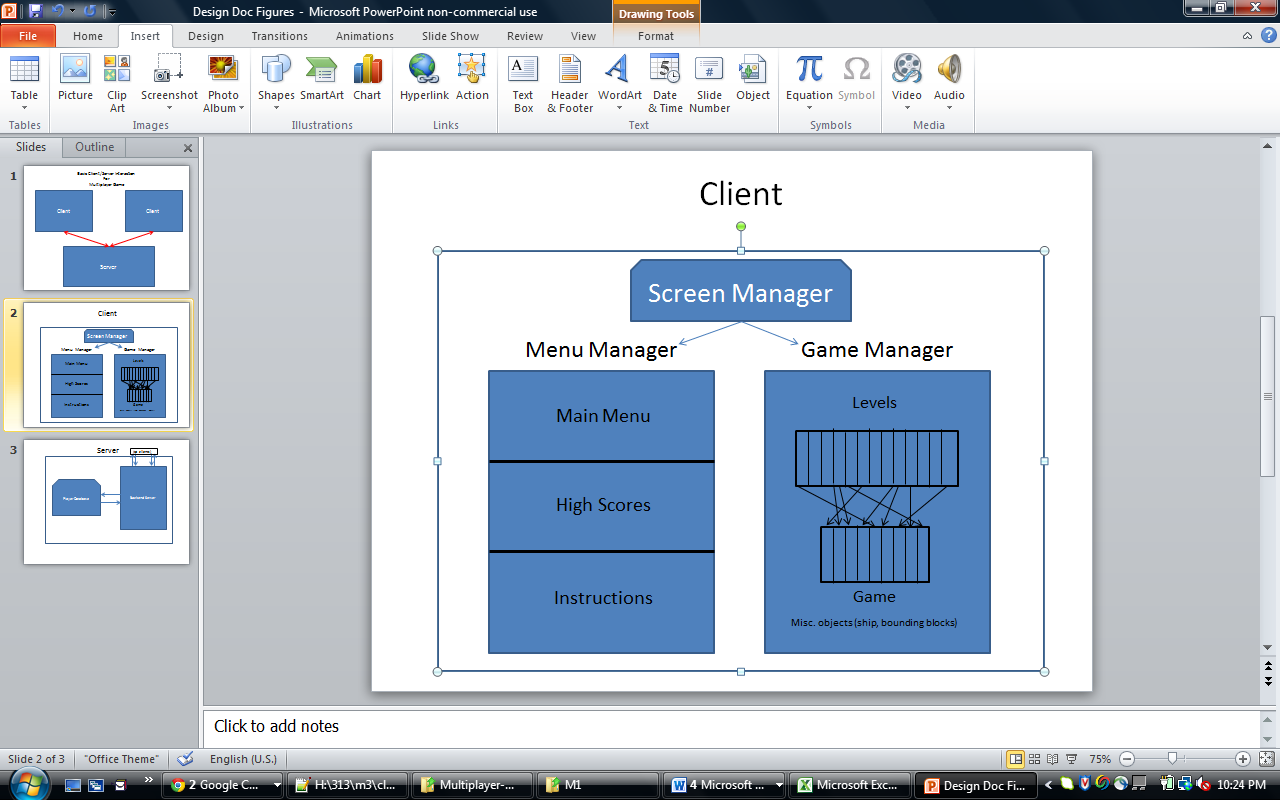
**Section 1 – State the purpose of your project/sub-system:**

This design describes on original game which attempts to provide all of the aspects of meaningful play. It is designed as an online, multiplayer racing game with a unique control system application. The style targets a retro aesthetic appeal, which is complimented by a simplistic control scheme.

**Section 2 – Define the high level entities in your design:**

High Level Entities:

In the game, we have:

The primary high level interaction in this design is between a client and a server, since the game is run in a multiplayer environment. Each player runs a local client on their own webpage, which communicates with a server, which in turn sends updates to another client, and vice versa. Furthermore, each of the high level entities can be further subdivide (especially the client side).

The client consists of a number of different aspects, which in turn create the game.

* Stage: hosts all of the local graphics. This is the easeljs underlying framework that we use to draw on the canvas.
* Screen Manager: passes control to the appropriate control manager with parameterized function calls. The two managers that it can pass of control to are the Menu Manager, and the Game Manager
  + Menu Manager: Contains all information about menus, and has access to server to retrieve high score information. Deals with any screen involving text only. Handles mouse inputs for these screens as well
  + Game Manager: Contains all information about the ongoing game. Included in the game manager is a list of all possible levels, as well as a list of the current game set up. The game manager contains a final tier of objects
    - Level: A numerical description of the various block-obstacle arrangements
    - Ship: The ship itself which the player controls, along with its various variables
    - CurrentGame: The randomly generated list of levels that make up a course, as well as the endpoints.
    - Collision Detector: An object to create simple collision detection. ***Notice that this is hosted by each client separately.***
    - Animator: Object to manipulate objects on the screen, mainly being the ship. Handles events such as a crash and respawn.

The server is a much smaller application, in charge mostly of relaying messages between the two clients. The main objects in the server are:

* Database: This is a database containing all of the users of the game, and their rating and password protection information.
* Server State Manager: A manager that keeps track of who is online, and who is making game requests. This is useful when the client wants to know what games are available.
* Server Game Manager: A manager that relays updates from a client to another client, and keeps track of the current game layout.

Finally, we have one shared object, being an Update object. An update is the JavaScript object which the client sends to/receives from the server. It contains all of the necessary information that the other client needs to track the progress of the other player.

**Section 3 – For each entity, define the low level design:**

In describing the low level design, let us start with the client side. As stated earlier, the client primarily consists of a stage, which is where all of the objects are drawn. Simply stated, it is the parent to everything else. A stage is an easeljs framework object, from which objects can be attached and detached in order to display and remove them. The stage does little more than this, to provide an entity on which we can draw. The functions that are used with the stage are:

* detach(o) – detaches an object from the screen
* attach(o) – attaches an object from the screen.

From there, we move on to the screen manager. This is really no more than a state which describes if we are in game mode, in in menu mode. This is useful for when the server makes calls to the client, as well as for aspects such as applying event handlers. But in essence, this is really just a design abstraction that makes the flow easier to understand.

So first, let us address the menu manager. Whenever the menu manager is opened, a mouse event handler is added, to handle clicks on the text which redirect the user. The text will be drawn again by using the easeljs Text object framework, which can be added to the stage and drawn easily. The main control flow of the screen manager is run through the function:

function displayScreen( index );

The indices of possible screens (highscore, main menu, instructions) are pre declared as constants in an enumeration style beforehand, and so this function can be used to call any menu (ie. at startup the main function can call displayScreen(MAIN\_MENU) to easily and clearly start the main menu). All other function calls through the menu manager are through the easeljs framework, in creating text objects to display, and attaching/detaching them from the stage.

From the screen manager, the program can also call startGame( type, host ), which begins a game using enumeration style constants type: SINGLE\_TIME, SINGLE\_CHALLENGE, MULTI\_RACE, MULTI\_CHALLENGE. This is passed as a parameter, which determines which type of levels to generate. The host is a Boolean value which is passed for multiplayer game modes, and determines if the current client will generate the levels, or if the other client will. This is not to be confused as meaning that the client itself hosts the gameplay.

Now we move on to the game manager itself, which is where the game mechanics and objects begin to come into play. The game manager primarily contains information on level design and the current layout of the levels. In addition, it contains functionalities for collision detection, as well as the ship itself, and the top and bottom bounding blocks. As explained above, the menu manager transfers control to the game manager by calling function startGame( type, host ). If the client is the host, it generates a level layout as explained later, and if not, then the client can do nothing better than to wait for the server to send it the game layout generated by the other client.

\*As a quick side note, the rationale behind generating the level design in the client is thus; the client already needs a level generator for single player, unless we want the client to have communication with the server while in single player mode. So from there, it would be repetitive and unnecessary to put this in the server code as well.

To generate a level layout in race/time trial mode, the program calls:

function generateLevelLayout() – this returns an array of levels.

Taking this a step deeper, levels themselves are hardcoded in as predesigned arrays of arranged blocks. Blocks themselves are merely just an array of points (later we explain the addition of lines to these blocks for collision detection). So for each level, we have a function stored in an array called:

function generateLevelX( index ) - The index describes how many levels are before it in the level layout. Practically, that means that for each block, we need to offset it by (levelWidth)\*(index). levelWidth is constant across all level designs to aid in the simplicity of the design, and furthermore is set to the width of the screen (which is set to a constant value of 1000).

So stepping back up to the function at hand, generateLevelLayout(), the functionality takes a series of non-repetitive random numbers in the set [0..NUM\_LEVELS-1]. We then call the function from the array of functions described above, call it, and add the returned level to the array of levels which make up the level layout. This is repeated from 1 to LAYOUT\_LENGTH which describes the number of levels which make up a layout.

Generating a level layout in challenge mode is slightly more complicated. Because we do not know when the game will end, we cannot compile the list of levels beforehand. Instead, what the program does is constantly store 3 levels. One is the current level, and then it buffers the next two. Any time the ship passes to the next level, a new level is generated. The low level design of this is an array of 3 levels called levelBuffer, and an integer describing the current level. Thus, whenever we pass to a new level, we generate a new buffered level, store it in levelBuffer[ currentLevel ], and then increment current level. Finally, levels can of course be repetitive, but will not be allowed to repeat more than once every three levels. This is very easy, since when generating a new level in challenge mode, it can simply check that the level generated is not one already stored in the levelBuffer. The functional specification of this would be:

function generateChallengeLevel( index ) – index again describes how many levels have become before it. Thus the level generation itself occurs in the same way, and this function just picks a random level, repeats until that level is one not in the buffer, and then returns the compiled level.

**Usage**

Describe in a paragraph how the object is used and what function it serves. If an object will interface with an external object or system, it is a good idea to show the interface for the object. Most importantly, you must again describe your thought process for defining the object as you did. List the benefits and risks. If an object provides an encapsulation, describe in a sentence why the encapsulation adds value. Use your descriptions to give meaning to the diagrams. They don’t have to be verbose, just enough to get the point across.

**Configuration**

If your object needs any special configuration or initialization, this is a good place to describe it. If not, this section can be left out.

**Model**

Figure 2 shows an example of a to supplement the System Security entity from figure 1. It is not perfect UML, but has some aspects of UML. Most importantly, it describes the design.

[](http://blog.slickedit.com/wp-content/uploads/2007/04/figure2.jpg)

Figure 2 (click to see full size)

Don’t worry about perfection in your models, but be sure to describe exactly what is going on in the diagram. Here, two concrete security objects derive from a base security object, and a security factory will create one or the other for a client depending on the security model of the system.

**Interaction**

This is also a good section for interaction diagrams. An interaction diagram shows how a set of objects or entities communicate with each other to perform a complex task. Figure 3 shows an example of an to show how a user might log in. It uses objects from the various entities shown in figure 1.

[](http://blog.slickedit.com/wp-content/uploads/2007/04/figure3.jpg)

Figure 3 (click to see full size)

Again, this diagram is not perfect UML, but it explains the communication sequence to accomplish a complex task. Interaction diagrams are most useful when you want to diagram how an object in your system will communicate with an object in another subsystem. This type of diagram will let the other developer verify that the interaction is correct.

**Section 4 – Benefits, assumptions, risks/issues:** In this section, make a list of 5-6 top benefits of the design, a list of **ALL** known risks/issues and a list of ALL assumptions. Some of this may simply be rehashing what you wrote in a previous section of the document. What’s important is getting all of these items into one section so that the reader doesn’t have to read the whole document to understand what the benefits, risks and assumptions are.

Never remove anything from this section! As risks become non-risks, document that they are now non-risks and why they became non-risks. Never erase them from the document. The same holds true for assumptions. You should be able to look at this section and know instantly what the current risks are to your design.