THE

RUBBLE DOCS

Contents

Introduction

When we started at Lasalle College we were unsure about our future. We did mess around with computers a lot, we found them interesting ,but we did not think of them as a serious career choice, we were too intimidated. Even though we felt like we knew quite a lot about computers, we were mesmerized by their complexity. When we would see intricate software, like the latest videogame or OS, we would see them as extraordinary and impossible feats made by only the brightest of men and we believed that we were no match for them.

With a lot of time, hard work and the help of some excellent mentors, we learned how the greatest machine mankind has ever created functioned and we learned how to utilize it to our advantage.

With this project we wanted to prove to ourselves and to the school that we have made massive progress towards becoming bright men, capable of designing and developing complex software.

We wanted to challenge ourselves with a difficult task that really encapsulates all the concepts that we had learnt over the past 3 years. When we looked at what internships where being offered and what was out there, we noticed that none of them really gave much responsibility and would not let us take the risks that we wanted. We therefore chose to do a project of our own and to make it be the most challenging possible that would test all our skills and thats when we decided that we wanted to do an in browser online video game.

Goal

Our goal was to create an online real time in browser mmo using no plugins and only the features of HTML5 and javascript while using minimal external libraries. These would be the features of the game:

- A top down perspective game where you would have an avatar representing you and you would see in real time the location orientation and behavior(attacks and items) of other users avatars.
- Users could collect and manage items to help them become a more powerful player.
- Users could communicate with each other with a in-game chat and that the user would be able to see real time statistics of his player and be able to manage his or her inventory.
- Create some computer controlled monsters that users could battle to gain better statistics and better items.
- Integrate player vs player fighting as part of the main experience by creating “Danger Zones” where players could attack other players depending on how strong they were and make it so that when a player dies they lose experience and all their items. User would feel like they could use the help of other players to gain better items, but could not fully trust them since they would be able to attack each other and benefit from their death. We wanted to give users a difficult moral choice.

System Overview

constraints/limitations

user graphic speed(html5 canvas render speed)

Since we are not using any browser plugins, we had to decide between rendering our game with canvas, svg or simple dom elements. We chose canvas, because it was significantly faster than svg and massively faster than dom elements. Even with it being the fastest of the three we knew our work wasn’t done there. We had to implement all the tricks to optimizing canvas, this includes: using buffered canvases, avoiding floating point renders, verifying whether images are in view and using layered canvases. This almost double our render speed and made it possible to draw the complex scenes that we needed for game, running at 60 frames per second in most modern browsers.

Server cpu speed

Since we are making a game with persistent state across all its clients, we had to make a robust server that could handle the extreme pressure of getting 60 requests a second per client. To do this there are several optimization strategies we had to apply. The first was split our file server from our game state server. We didn't want our game server to be weighed down by all the file requests so we completely detached it from the game server so that they can be run on completely independant machines. We also had to constantly monitor and measure all the code that was added or changed to the server and make sure that it wouldn’t slow down or affect performance, our goal was to make the server be able to run, without latency 60 times a second and we achieved it, using optimized looping and asynchronous functions. We also grouped data into sectors to help with collision detection as well as many other functions that required verifying nearby entities.

Network bandwidth

To achieve minimal network bandwidth we had to use a more complex more recent, lightweight network communication protocol than http. We discovered an amazing protocol called websockets, which we decided to use for this project for its minimal overhead and 2 way communication.Even with this amazing technology we needed to minimize all communication between the server and its clients. Since we are sending and receiving 60 request a second we had to make them as minimal as possible or else the game would not be scalable. First of all to reduce network bandwidth we split the http server from the game server to be able to have a completely dedicated pipe for game server data. Next we monitored all transmissions, we reduced and combined requests to make as lightweight as we could possibly make it. We also split up static game data such as map layout, monster types, item types and ability types and we created json files so that the http server could handle that information freeing up bandwidth for the game server to send dynamic data such as player, monster,ability and item positions. We also grouped data into sectors to help reduce the amount of data needed to be transmitted.

network latency
Network latency is always something that you must be taking into consideration when making real time applications. We had to determine what was the worst latency we would allow and how we would handle people if they have too high latency.

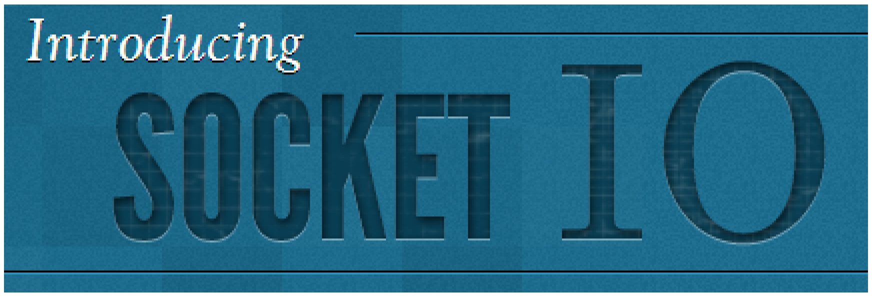
Staff, time and lack of experience
Since we took on such a massive project and we are only 2 inexperienced game developers to work on it in a month, there were cuts that had to be made as far as scope of the project. There were many things that we simply could not do because we didn't have time to do them or didn't have time to learn how to do them.

Platform/Software/libraries used

Web (HTML5)
We chose to develop on the web platform because its easily accessible and requires no downloads for customers to access our application. Its simply the most convenient way for customers to be able to quickly access our app without any headaches. We also chose to only use HTML5 for the same reasons so that customers don't have to download plugins, reducing the amount of steps between finding our application and using it.

Node.js
We decided to develop the server using Node.js for several reasons. The first is that it reduces the number of programming languages we needed to learn because we would be programming in javascript on the client and on the server since Node.js is a javascript based platform. Since we are using the same programming language on both ends, the objects we use require no conversion when sent to the client. We also chose nodeJS because we knew we could use the excellent socket.io library with it for networking with websockets. Node.js is also increasing in popularity and becoming more widely used even though its still in beta.

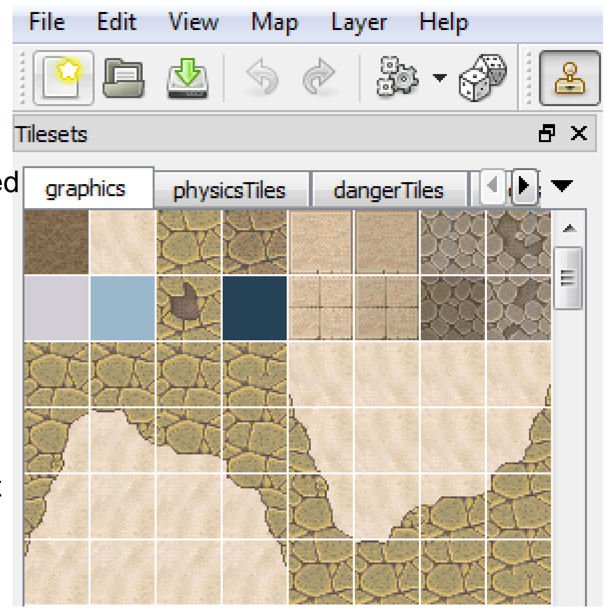
Socket.IO
To do make a real time application like our game we needed to use a lightweight two-way communication protocol like websockets. Since the websocket protocol is quite complicated we learned to use the socket.IO library to simplify using websockets and to also make it more backwards compatible since it can fallback to older communication protocols if the browser is not compatible with websockets.



Require
Javascript on the client side by default has no support for modules which makes it really difficult to program for since you're always programming in global scope which can be error prone and insecure, if you ever using external plugins. We used the Require library to give us the capability to make modules like on the server, which made it easier to organized and made our application more secure.

Tools

Tiled



To simplify map creation we used a tool called Tiled. This tool allowed us to use tilesheets to create layered complex maps. This tooled allowed us to export the map as json data that we could parse using the map builder to create a more appropriate map structure.

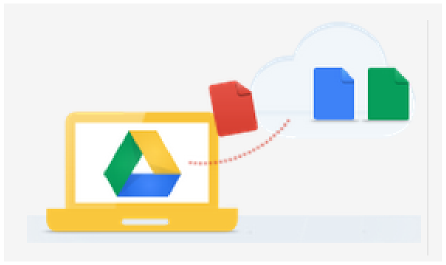
Chrome debug tools

Chrome debug tools allowed us to analyze and help debug our application. We used it to inspect objects and view debug logs. We also used it to inspect frame rate which helped us optimize the canvas.We used Chrome debug tools to inspect the amount of memory our web application would take and we used this information to debug memory leaks and reduce the amount of memory our app was taking. The debug tools also simplified working with css and helped manage and organize the dom elements.

Sublime Text 2

Sublime text 2 is the text editor we both used. It is a simple text editor that has a lot of good shortcut to make coding easy and simplifies redundant tasks.

Google Drive



To work as a team we used google drive to keep our project files synchronized. Google drive almost instantly updates the files we are using so that we are always up to date with what the other is doing. This can make it difficult if we are working on similar things, but we made sure not to do that. We did have some synchronization issues ,but in general it worked well.

BFXR

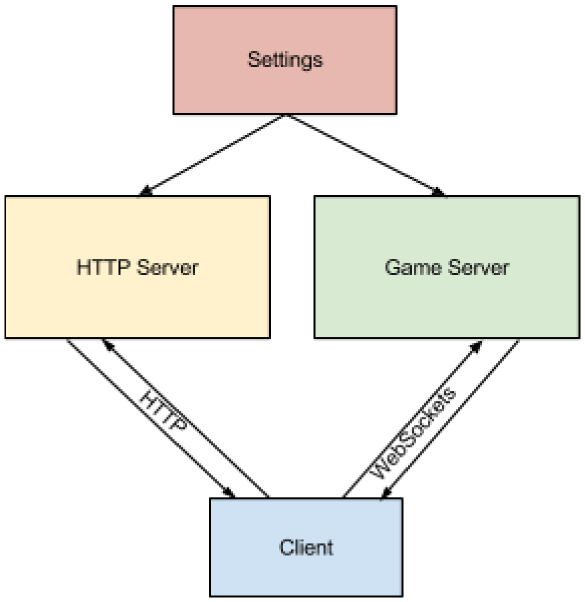
With BFXR we created all our game sounds. It is a simple free tool to create retro arcade sounding sound effects.

System Architecture

Our Design

Our initial design was to have a simple client/server architecture, but while testing we noticed that there was too much weight on the server so we decided to separate it into 2 separate servers: the game server and the http server. Like we mentioned previously, we did this to reduce the load on the cpu since each server can be implemented on separate machines using separate processors and separate bandwidth.

There is also a global settings file that defines all the settings for both servers. It contains information such as the domain and port for each server, as well as which modules are enabled on the game server.



Process Analysis and Design

Website use cases

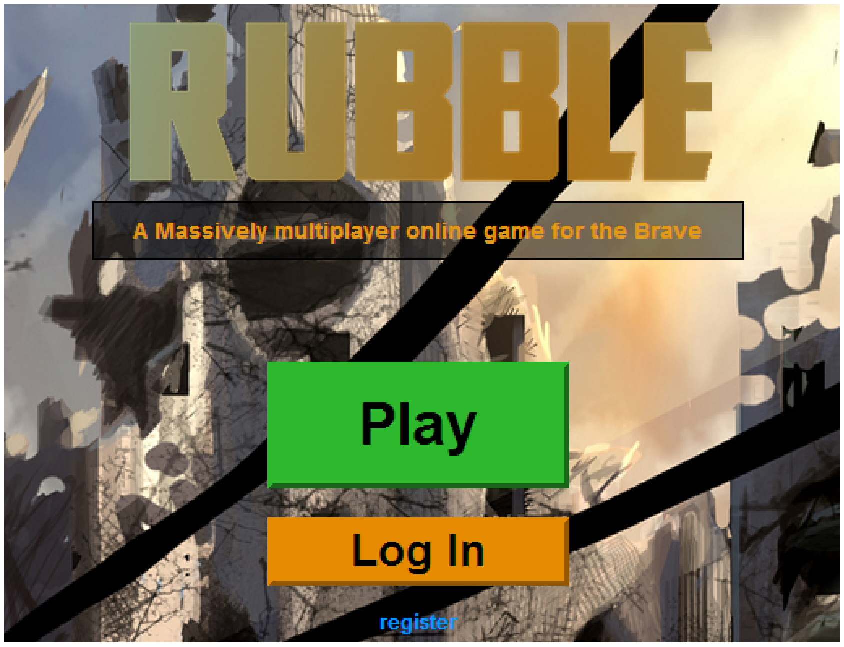
These are the use cases related to the website itself, but not the main application/game. They have been kept rather simple as the main priority within our project was the functionality of the game itself and not its gateway.

Playing as a guest

The user can simply click play and start playing the game, a temporary alias will be assigned to the player. The player's progress will not be saved. It allows for quick access to the game without requiring the user to register for an account.

Registering an account

By choosing a username, a password and a valid email, a user can register for an account. This allows users to keep any progress within the game saved automatically.



Logging in

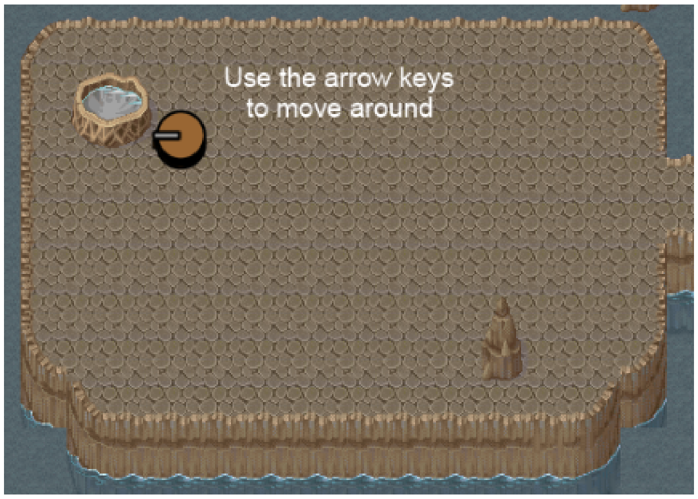
Once a member has an account. As long as he or she has a valid password and the account is not logged in already, the user can access their account. Users only have 5 attempts to input the right password before the server blocks the person from logging in into that account.

Game use cases

The following are all use cases regarding the user and within the game.

Moving around

Using the directional arrows a player can move around realistically using velocity, acceleration and friction to navigate our two-dimensional world. The speed is determined from the player's stats.



Colliding with objects

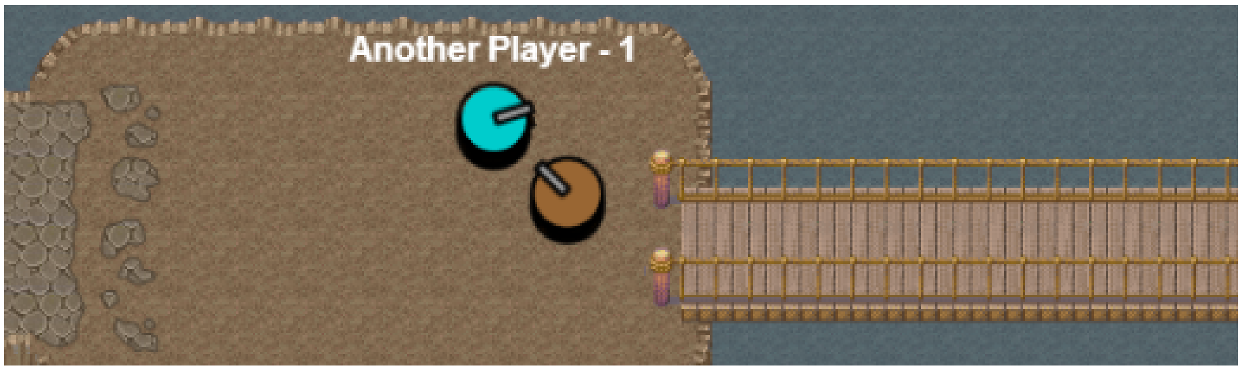
Players are restrained from going out of bounds or going through walls by our client sided physics engine.

Locking rotation

By holding spacebar the current location you are facing is locked.
This allows to shoot backwards while running away from monsters or other players.

Viewing other players

Other players can be spotted around the map.
They will be coloured differently and their name and level will be on top of their heads
.



Viewing your current stats

On the side bar of the game, you can see a few things.

-Your level.

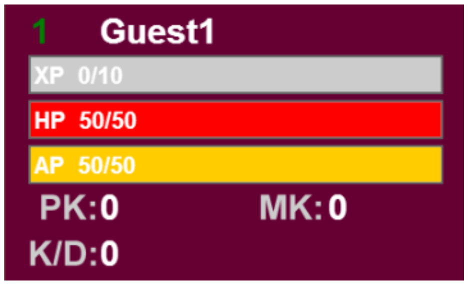
-Your nickname.

-Your experience.

-Your HP bar, which represents your hit points or life.

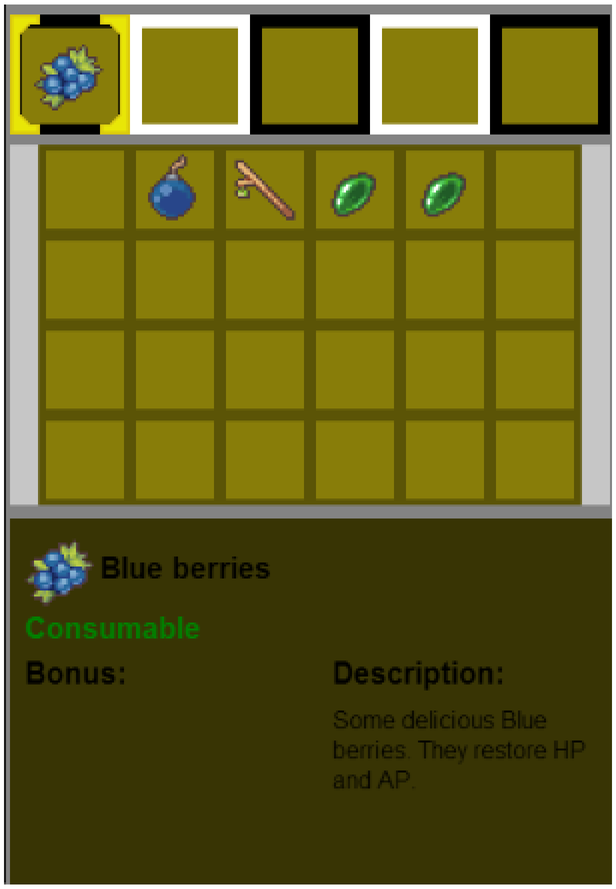
-Your AP bar, which is drained when using abilities.

-You can also see the amount of players you've killed (PK) the amount of monsters you've killed (MK), and your kill to death ratio (K/D).



Viewing your inventory

Right under the player stats, there is the inventory. This is where the player keeps his items. The top 5 spots in the inventory are items the player can use by using the keys: **A**, **S**, **D**, **F** and **G**.



Blue berries

Consumable

Bonus:

Description:

Some delicious Blue berries. They restore HP and AP.

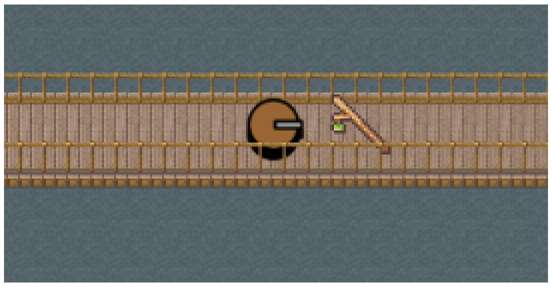
Moving around the inventory

By pressing **R** the player can access the inventory. In this mode the arrow keys travel through the inventory slots.

Viewing an item's description

When the menu selector is above an item, information is displayed under the inventory. This shows the item's stats and description.

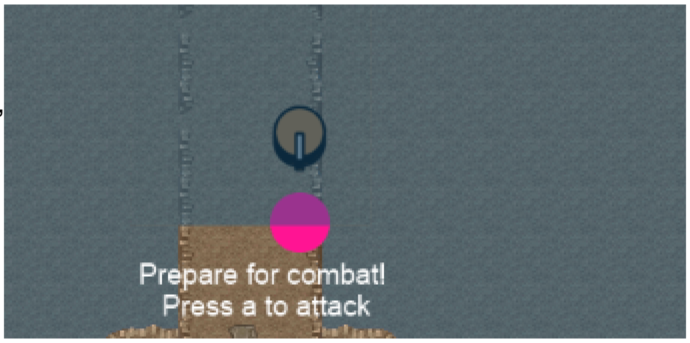
Managing inventory
Using the menu selector, the arrow keys, and the space bar you can switch the positions of items within the menu. Certain items of the same kind even stack!



Picking up items from the floor
By pressing **Z** a player can pickup an item from the floor. He or she can also manually grab items from the list of floor items right under the inventory.
The list appears when standing on at least 1 item.

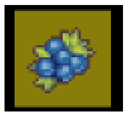
Dropping items
By pressing D while in the inventory. A player can drop an item. If the item he drops is a stack of similar items, The player will drop all the items. When a player drops an item anyone else standing around can see the items on the floor.

Using the default attack
Even if no item is equipped on the top row,
If there is no item on the first slot which corresponds to the **A** key,
a default attack will be triggered.

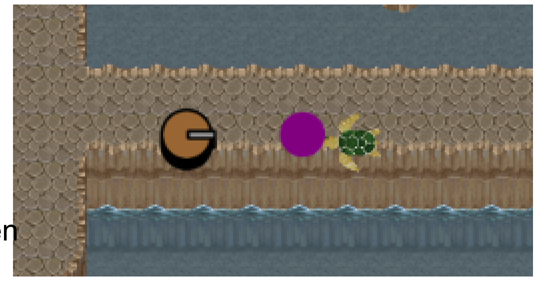

Prepare for combat!
Press a to attack

Using an ability
If an item equipped in the top row has an ability, using the appropriate key will trigger that ability, abilities have many properties such as damage, cooldown, etc. Some abilities such as the ability to set bombs or certain guns require certain other items to function. Using an ability always uses AP (Ability points). the AP bar is located on the top right under the HP bar.

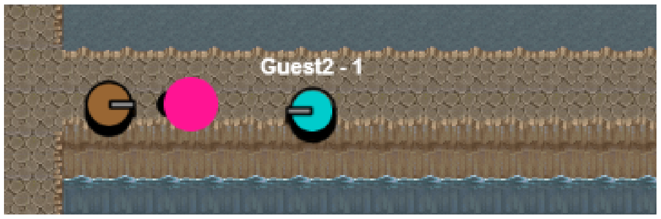
Using consumables
Certain items such as blueberries can be put into the top row, but instead of allowing the player to use an ability they have an instant effect, and then they disappear. For Example: Blueberries refill health.



Fighting monsters
By using abilities a player can fight a monster.
Once the monster dies it gives out exp to the players that have damaged it. Depending on how much damage the monster has taken from each player.
It will then randomly drop some of the items it has been assigned to drop, based on the item's chance of drop for that monster.



Fighting players
Fighting players is very similar to fighting monsters. The only difference is that they are much smarter. Fighting players is much more rewarding as all the items they've collected in their travels will drop for you to feast upon

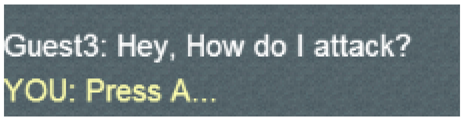


Viewing the danger level
Players cannot be fought anywhere. The danger level is what indicates how dangerous an area is. If the danger level is zero, then the zone is safe: players cannot fight players.
However if the danger level is two, then anyone two levels above or below can be attacked by you, as well as attack you. The more dangerous the zone, the highest chance you'll encounter a high level player able to attack you. The danger level is situated on top right.

Leveling up
By defeating players or monsters a player gains experience. When certain milestones are reached the player will level up. this will raise many of the player's stats making him more powerful.

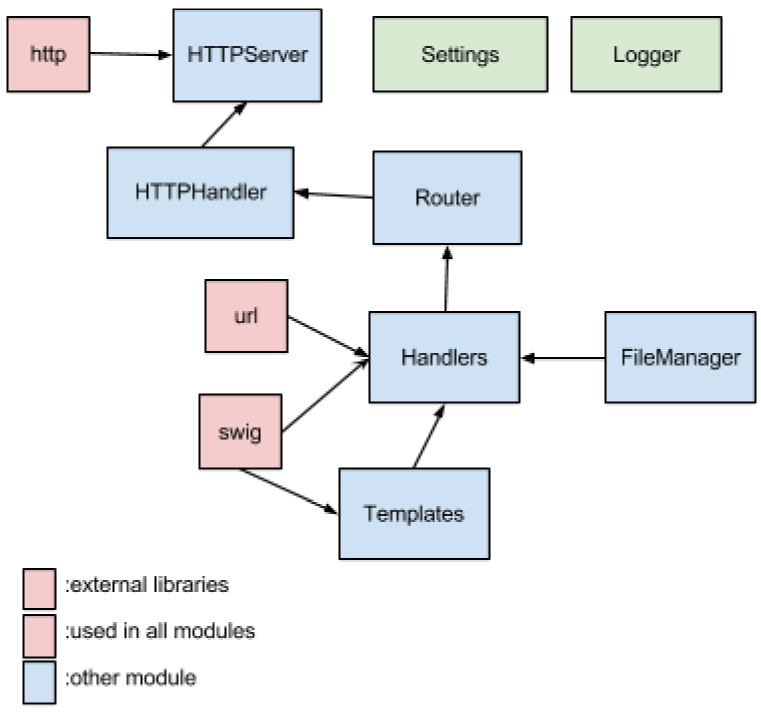
Dieing
An inevitable part of life. When a player's HP reaches zero, the player dies and is respawned in a safe location within the map he is in.

Chatting
By pressing the enter key during the game, players can chat with players nearby, perfect to talk about the weather or to scheme a betrayal.


Guest3: Hey, How do I attack?
YOU: Press A...

Changing maps
Any player can simply walk into a portal and will change maps. Depending on the map of origin the player will be placed in the appropriate place in the destination map.

Module Structure - HTTP Server



Our Design
The HTTP server modules deal explicitly with HTTP requests. Overall, it gives us complete control of what URLs return to the client. The idea behind it was inspired by the MVC pattern which is very popular nowadays in within many different contexts.

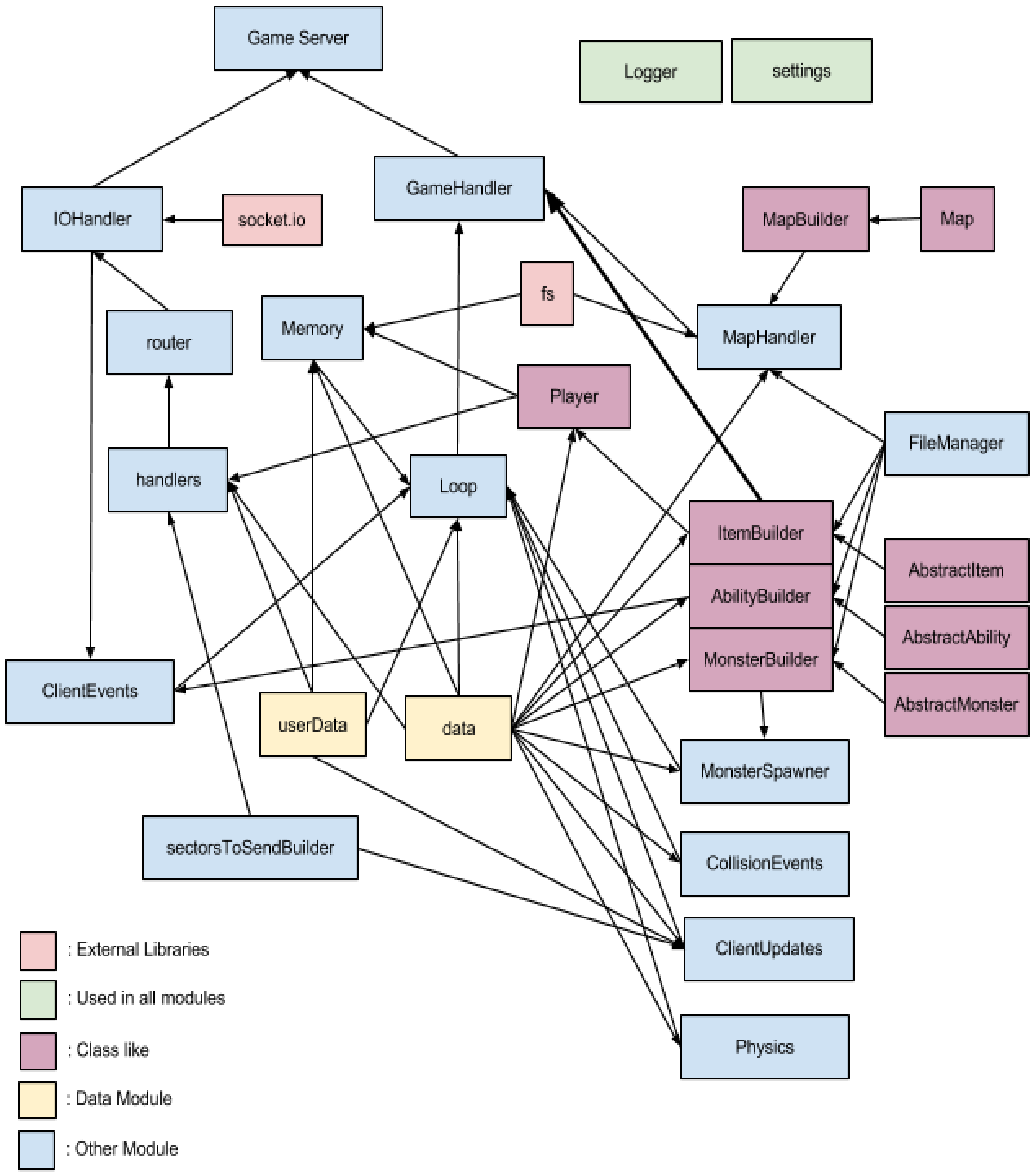
HTTPHandler
This module deals with the HTTP Server. It is the root of the HTTP server and point of access for other modules and requests. requests are then passed down to the router from here on.

Router
This module routes HTTP requests to the appropriate handlers.

Handlers
This module defines what we do with requests after they have been routed to the module, it utilizes the templates module to return customized views as well as triggering many more behaviors.

Templates
This module contains all the templates or "views" that the http handlers return to clients. The templates can be customized before returning them to the clients.

Module Structure IO/Game Server



<u>**Module Structure - IOServer**</u>

Our Design
The IO server is designed very similarly to the HTTP server, the main difference being that no views or templates are returned. Requests are instead stored in queues, although they can be also dealt with immediately.

<u>IOHandler</u>
This module deals with the IO Server. It is the root of the IO server and point of access for other modules and client requests. It defines what we do when we receive a Socket IO request. It might put certain messages received in a queue for other modules to loop through them when needed, for example: the game server module.

<u>Handlers</u>
This module defines all the different socket event types and defines whether they should be queued or not.

<u>Router</u>
This module creates the listeners for all the different type of handlers as well as creating the appropriate queues for each event.

Module Structure - Game Server

Our Design

The game server is where we had the most freedom for designing and organizing. Compared to the HTTP server or the IO server where we mostly created wrappers for existing Node.js libraries. The basic structure is very similar to any existing game: The game loop, the heart of the game, runs 60 times a second.

Each loop all the game logic takes place, it evaluates and updates game data according to the new information received, it spawns new monsters, it updates the clients with new game information, it evaluates collisions and handles them accordingly and it updates the state of all entities(monsters, items, abilities and players). The game loop module utilizes all of the other modules to accomplish its task.

Builders

The builders are modules that create objects of a certain type: item builder, creates new instances of a certain type of item, monster builder, creates new instances of a certain type of monster, etc. The builders use the object definition file such as the monster.json for the monster builder and the abstract module to define the functions for the object.

- AbilityBuilder

  Create new ability objects (the attacks for the monsters and players).

- ItemBuilder

  Creates new item objects.

- MapBuilder

  Creates new map objects.

- MonsterBuilder

  Create new monsters objects.

- SectorsToSendBuilder

  Determines the sectors to be sent to the client player (the 9 sectors around the player).

Game Logic Modules
These modules are used in the game loop.

- CollisionEvents

  This module defines what entities do when they collide with each other. It uses the collision list obtained from the physics module.

- MonsterSpawner

  This module decides when and where to spawn monsters as well as which kind depending on the monster spawn rate.

- Physics

  This module checks for collisions between, monsters,players,abilities and items by using circular collision detection, because it is significantly faster than square collision detection and adds them to a queue, it is also optimized by only checking nearby sectors for collisions and not the whole map. It then returns this queue for the collision events module to handle.

IO

- ClientEvents

  This module loops through the queues that are created by the IO Servers router module and handles them appropriately. Currently it handles all the item logic (what to do when the player picks up, drops or uses an item) and the chat messages received.

- ClientUpdates

  Client updates sends messages to the clients informing them of the current game state. It only sends the 9 sectors around the player as to not use too much bandwidth. It uses the sectorToSendBuilder to calculate what sectors need to be sent to each player.

Maps

- MapHandler

  Loads the map files generated by the Tiled program. It then uses the map builder to recreate the maps with sectors, walls, danger levels, monster spawns, etc.

- settings

  Defines certain settings for the map handlersuch as the size that the sectors should be.

Users

- UserData

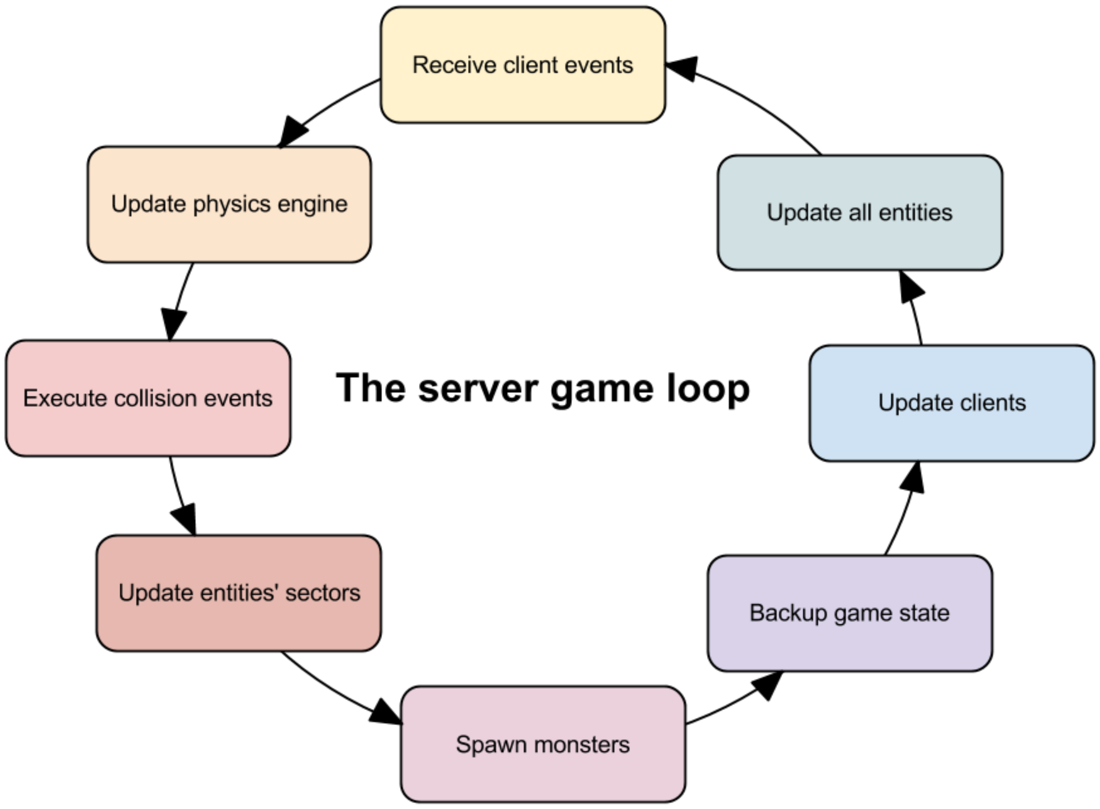
  Similar to the data module, it contains the data for all the users in memory, it also calculates the time left for invalid login attempts.

Data
This module contains all the variables that will be used in the game server (maps,players,items,monsters,abilities). Its loaded at the beginning when the game server is initialized using the memory module. This is the centralized storage module for the entire game. And it is responsible for keeping the game's state.

GameHandler
The GameHandler module is in charge of initializing the game. It initializes all the builder classes and starts the game loop.

Loop

The loop module is like we've said before: the heart of the game server. It is a big loop that handles new data from clients, updates all the entities, evaluates collision and sends the updated data back to the clients. It runs 60 times a second, but since we use delta time for all time related behavior it can be set to any interval and would not change the behavior of the game, it would only make time related behavior more precise and require more processing power.



Memory

The memory module is the module in charge of backing up the game. It saves all the player and user data into json files at a certain interval, as to not use too many resources. Its also in charge of parsing the saved files and loading the player and user data when the game server is initialized.

Utility

- FileManager

  This module manages loading and removing cached Files from memory.

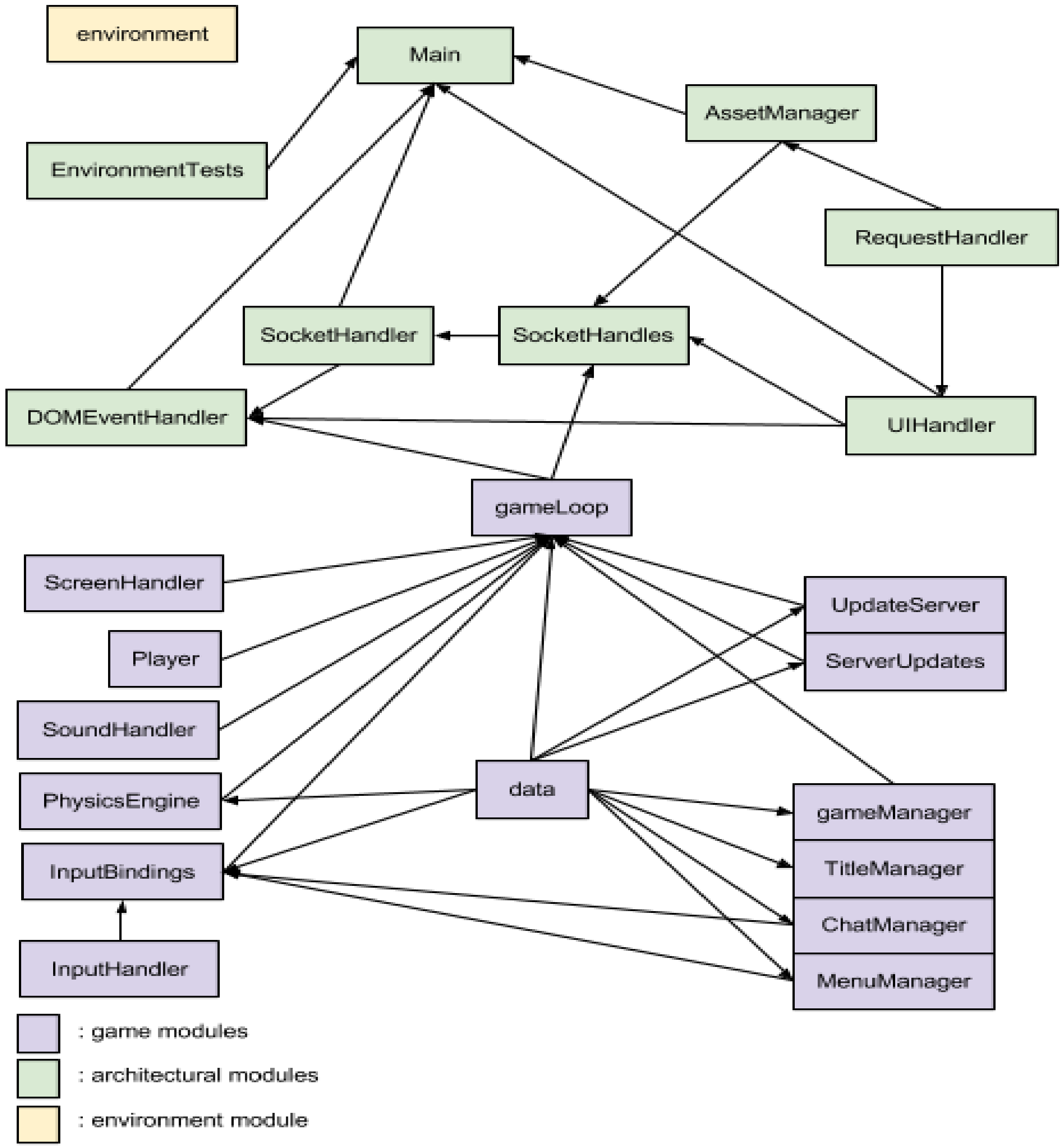
- Logger

  Logger module simply logs info, warnings or debug info depending on server settings

Module Structure - Client

Our Design

Client module design was a bit different than the module design in the server. First of all, while Node.js has a module system already established using Common.js, the browser does not have a module system, so we ended up using Require.js to modularize our javascript files.



Player
The player module contains the constructor function for creating new player. A new instance of a player is only created once at the beginning of every playthrough, and its prototype has functions such as *move*, *onCollision*, and *animate*.

ServerUpdates
This module uses the client-side Socket IO library to set the listeners for the messages sent by the server to the clients. The client then updates its state with the new relevant server information.

UpdateServer
This module also uses the client-side Socket IO library to send updates to the server regarding the state of the client. It’s a very small module with an *update* function which creates a small data structure containing all the of the player’s updates such as: x, y, rotation, item changes, and abilities used. It is then sent to the IO Server.

ChatManager
The chat manager deals with the chat window in the game. It has many responsibilities such as making sure messages disappear after a certain time, adding new messages received from the IO server, adding messages to the queue of messages to send, as well as more complex tasks such as preparing messages to send, character by character.

GameManager
The game manager uses a ScreenHandler instance tied to the game canvas to draw everything that happens in the game window. It draws the sectors, abilities, monsters, items on the floor, the player itself, and other graphic layers that are on top of the player. It also has the duty of dealing with some other minor game related tasks not related to drawing.

MenuManager
This module also uses a ScreenHandler instance which is tied to the menu canvas to draw the stats of the player, their nickname, level, as well as the items they are carrying and any items in the floor optionally.
It also has all the menu functionality as a duty, such as moving the selector within the menu, and queuing changes such as item switches, pickups or drops into the appropriate array which is then sent to the server for validation purposes.

TitleManager
The title manager has all the responsibilities regarding drawing and managing titles. Titles are all the messages which appear on a transparent canvas located on top of the game and menu canvases. Titles can include the animated titles which appear when you start the game, die, level up, or change maps into a newer area.

Data
The data module within the client contains the client player, an array of nine sectors relevant to the player holding dynamic data such as other monsters, players, items or abilities, an array of static maps which hold graphic and structural information regarding maps, static objects containing monster, item, and ability info, a data structure used to store all changes that will be sent to the server within the next game loop, and many other minor data used during the game cycle. This module is imported in many places.

GameLoop
The game loop module in the client is the heart of the game within the client's browser. It is very similar to the server loop which runs at 60 frames per second, and is structured very similarly. The only difference is that instead of doing expensive computational tasks, most of the processing power is spent drawing. It uses updated data at the beginning of the game loop to illustrate the server's state, and then sends the server any updates about the client player.

InputBindings
This module uses the InputHandler module and defines what different keystrokes will do depending on the state of the input handler. Some of these states may include: chatting, moving the selector in the menu, or playing the game. In all this scenarios we want the same keys to act differently, elegantly.

InputHandler
The input handler works at a lower level than the input bindings module. It is the layer in between the DOM events, and input bindings. It keeps track of what functions to execute depending on DOM events triggered, and on the state of the input handler.

PhysicsEngine
The physics engine module is pretty simple in the context of what it does. It basically handles the physics of the client player. It will validate new positions and cancel certain velocities depending on if collisions are detected with walls and such.

<u>ScreenHandler</u>
The screen handler module returns a constructor function which creates screen handler objects. These objects are a wrapper around canvases with many added drawing functionalities. The ability to draw things relative to a player's perspective for example, is due to the screen handler object. It also draws text, images, rotated images, shapes, caches images for later use, and has many optimization features such as double buffering encapsulated within it.
It is given all the image assets contained in the asset manager internal array when it is first created.

<u>AssetManager</u>
This module is in charge of loading all game assets such as sounds, music, images, json files, html files, or css files into an internal array. It uses the RequestHandler module to asynchronously download all specified assets using AJAX.
The strength of this module is that all our assets are loaded, cached, and completely controlled by us and not the browser. This means we know the exact moment in which everything necessary for the game is loaded, or if any errors occur with specific files.

<u>DomEventHandler</u>
Using the UIHandler module, this module defines what click does within the website, and tells the UI handler to reach accordingly to each scenario. It ends up being a great way to organize our click events within our dynamic website without having to define what each button does inline or on the load event of our website; something which might be impossible most of the time since not all our buttons technically "exist" at all times

<u>EnvrionmentTests</u>
This module contains a function which can be ran to test if the client browser has all the features needed for our game to run smoothly. This may include many HTML5 features such as audio or canvas objects, or any other browser native objects.

<u>Main</u>
This module is the entry point to our client scripts. It begins the loading of all assets, which happen to be defined within this module, and uses the UI handler to set the initial state of our website.

RequestHandler
The request handler module is in charge of queueing and sending AJAX requests to our server asking for any files it might need. This module is heavily used within the asset manager module to obtain all the files asynchronously and verify the asset's state within the callback to know if there have been any errors before storing it in the assets array used by all other modules.

SocketHandler
This module uses the client-side Socket IO script to establish a connection with our IO server. It contains the socket object which is either valid or invalid after connection is attempted.

SocketHandles
The socket handles module is in charge of defining what certain messages from our IO server will trigger within the client. Events defined here are mostly validation related, such as guest validation, user validation, or registering validation. They may define what the client will do depending on if the server has validated login, or guest login.

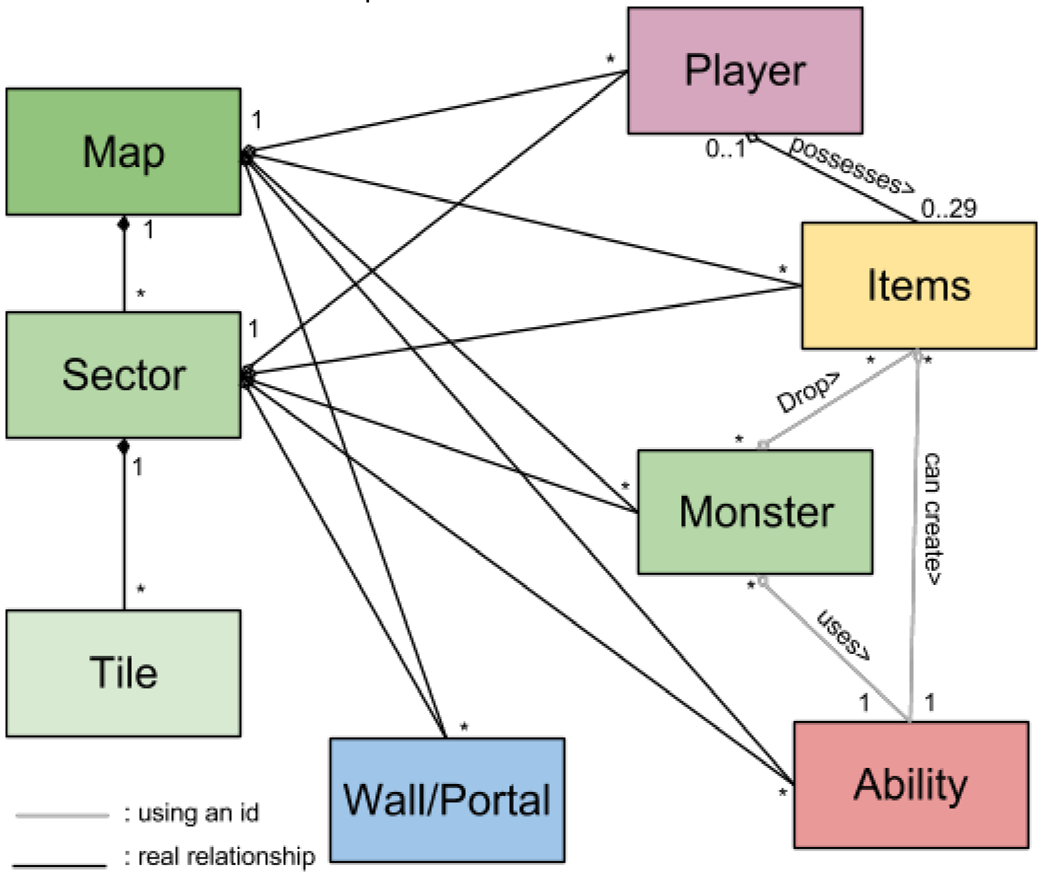
SoundHandler
The sound handler is in charge of queueing sounds to play, which are stored in an internal array after being loaded through the asset manager module. It also contains a Sound constructor function which encapsulates the HTML5 Audio element and adds many useful functionalities.

UIHandler
This module handles everything related to our website when it comes to content and transitions. It manages fade transitions from our asynchronously loaded pages, which include the game page containing all the canvases.
It also manages history which makes our back buttons within the website work properly even if the URL is not changed. It works in unison with the DOMEventHandler module to create a smooth experience while visiting our website.

Data Structures in the Server

Our design

Designing the data structures within the server was no easy feat. There were many things to take into consideration. First of all, since server-client latency is very important within our game we've had to simplify our structures to be as simple as possible, since they have to be sent to the client with the lowest overhead possible.



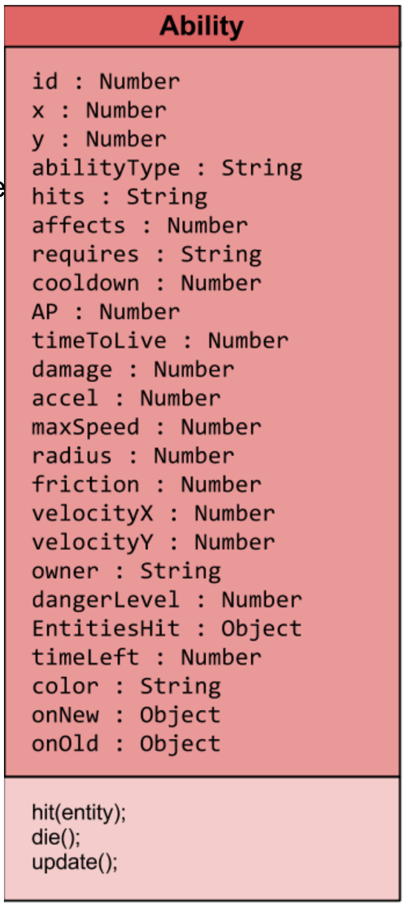
The only place where we keep our data structures is within the data module and the user data module. All other modules act upon this data when they need it. In the user data module we have a collection of users, and in the main data module we have a few collections of objects as well which include: players, monsters, items, abilities, and maps.

The server data structures in detail

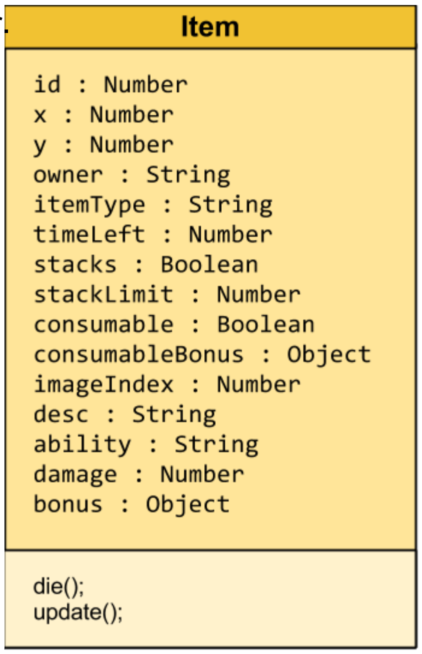
A short description of each data structure within our game will be given in the section below, followed by some diagrams showing the relationships in between these data structures as well as a more detailed definition.

Please note that the best generalization possible will be given regarding the data structures used, but nothing is set in stone within our "classes", as we fully utilize javascript's dynamic typing to our advantage.

The ability data structure
Abilities are one of the most basic form of entities within our game, they have locations within the map,a unique ID,an ability type, an owner, velocity, rotation, friction, acceleration, base damage, a cooldown, AP used, a predetermined time to live, the amount of entities it can collide with, its behavior, and even items it might require before being created.
Abilities are basically all attacks within our game. We often represent them with simple colorful circles.

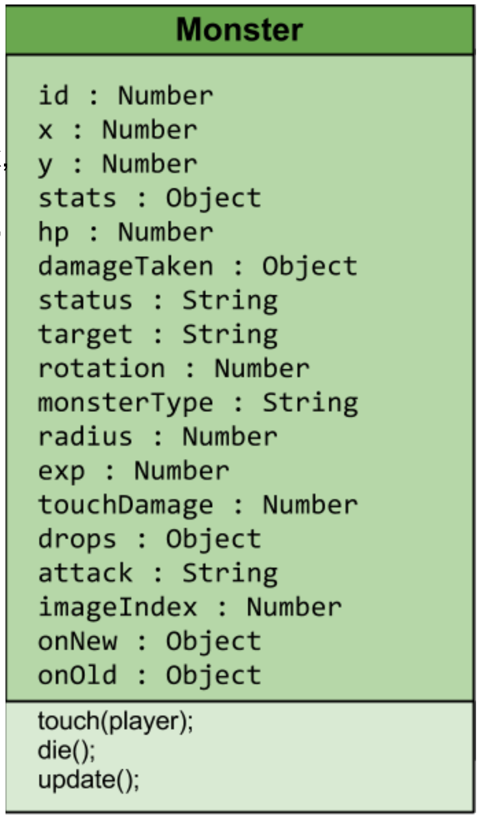


The item data structure
Items can either be in the map itself, or in possession of a player. They are created whenever a monster is defeated. Items have a unique ID, an item type, a location within the map, an owner, a boolean indicating whether the item is consumable or stackable, the amount of items of its kind that can be stacked in a single inventory slot, a description, and an image index. An item can also possess an ability, as well as bonus stats for the player wielding it.



The monster data structure

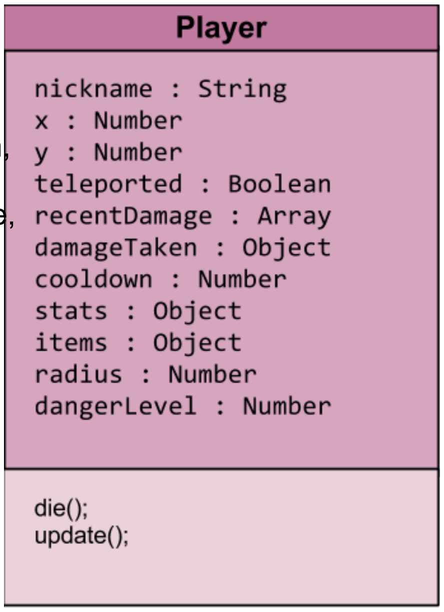
Monsters are essential within our game, they possess a unique ID, a monster type, its unique set of stats, Health points, exp given when defeated, a radius, an image index, its behavior when idle, attacked, or about to be defeated, the ability it uses, an array of damage taken and by whom, as well as an array of the items it can possible drop and their chance of dropping.



```
Monster

id : Number
x : Number
y : Number
stats : Object
hp : Number
damageTaken : Object
status : String
target : String
rotation : Number
monsterType : String
radius : Number
exp : Number
touchDamage : Number
drops : Object
attack : String
imageIndex : Number
onNew : Object
onOld : Object

touch(player);
die();
update();
```

The player data structure

The player data structure contains many things. A player has a nickname, a location within the map, a radius, rotation, the current danger level, an array of damage taken and by whom, the current cooldown before using the next ability will be possible, stats such as level, exp, strength, dexterity, defense, HP, max HP, AP, max AP, players killed, monsters killed, deaths, an array of bonuses given by items worn, an array of objects which represents the player inventory, which can contain either an item or an array of items.

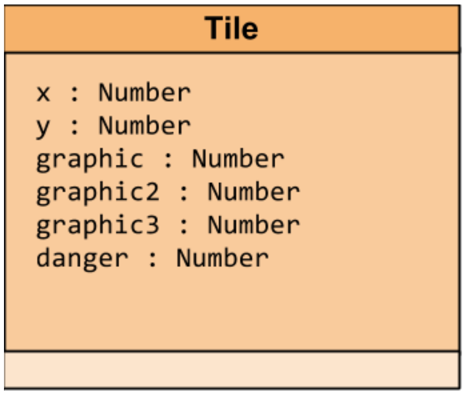


```
Player

nickname : String
x : Number
y : Number
teleported : Boolean
recentDamage : Array
damageTaken : Object
cooldown : Number
stats : Object
items : Object
radius : Number
dangerLevel : Number

die();
update();
```

The wall data structure
Not much to mention, it has a global location, a width, a height, and a wall type; the wall type changes the wall's behavior to either block players from moving, block abilities from moving, or both. It has no methods.

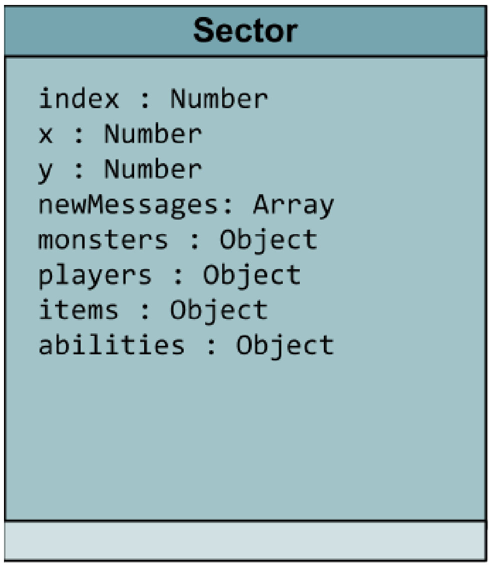
The portal data structure
A portal is a very similar entity as a wall. It has a global location, and a map property. A portal can either be a receiver or a sender, it allows players to travel from one map to another on contact.

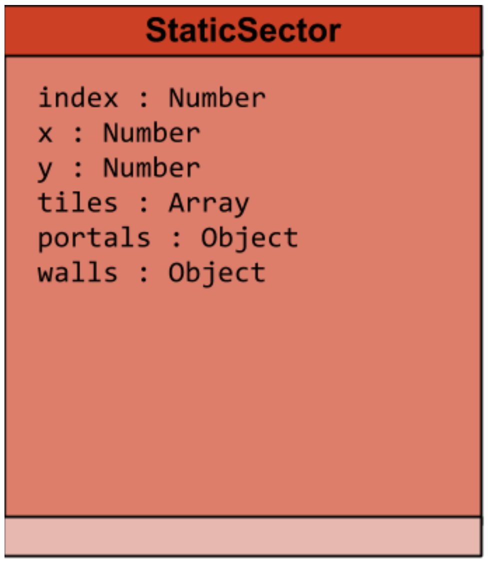
The tile data structure
A tile is simply an object with a global location, a corresponding graphic ID, a danger level, and a predetermined size. Visually it is the small 32px by 32px squares which we see in the background of the game.



The sector data structure
A sector is a pretty complex data structure. It possesses an array of entities including: players, monsters, items, abilities, walls and portals, a global location, and an array of tiles.

The purpose of sectors is so that no matter how big a map is, we always have small collections which are relevant to a player, the sector he is in and the ones around him. This is a hugely important deal as updating a client with all the map's activity would be brutal, not to mention collision detection. Instead of receiving a whole map of information, a client receives the state of the sectors around him.

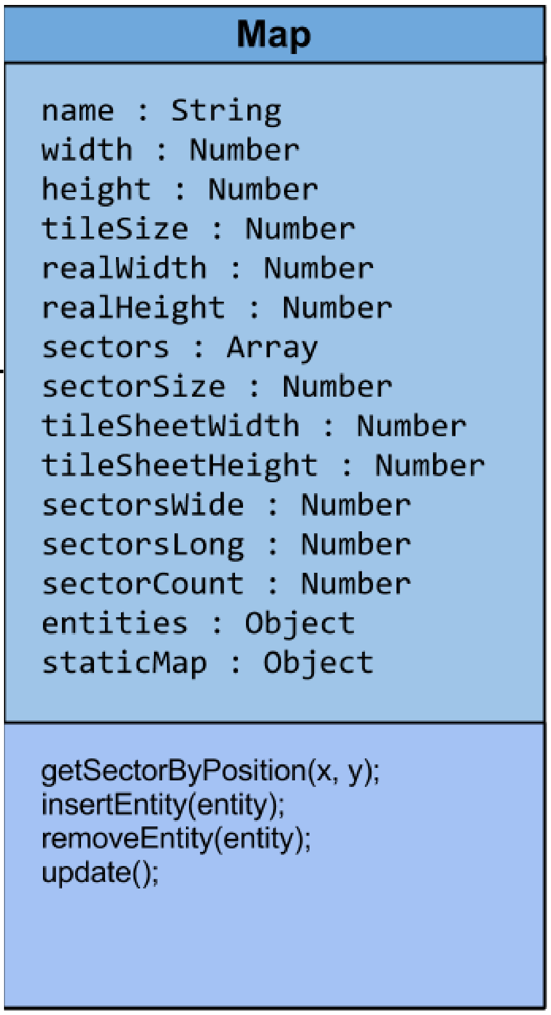




The map data structure
The map data structure is the heaviest and the most complex data structure of them all. While our server might contain many maps, a map is composed of many sub sectors, which in turn is composed of many different tiles.

A map is composed of many properties such as a name, a width, a height, the width in tiles, the height in tiles, an array of entities consisting of all the data structures mentioned above, and an array of sectors.

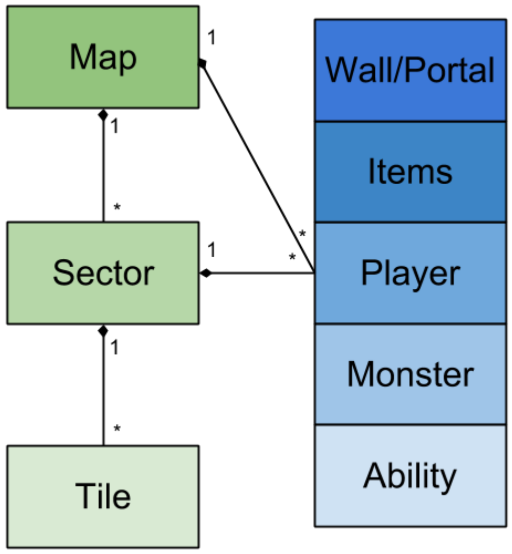
Our maps are all automatically divided in sectors, whom's size is based on the server's settings regarding map creation.



```
name : String
width : Number
height : Number
tileSize : Number
realWidth : Number
realHeight : Number
sectors : Array
sectorSize : Number
tileSheetWidth : Number
tileSheetHeight : Number
sectorsWide : Number
sectorsLong : Number
sectorCount : Number
entities : Object
staticMap : Object
```

```
getSectorByPosition(x, y);
insertEntity(entity);
removeEntity(entity);
update();
```

A simple model representing the map data structure



Data structure's functionality
While no true polymorphism exists within javascript, let alone classes, players, monsters, abilities, and maps all have an *update* function which runs during the game loop execution, 60 times a second.

The *update* function contains what an entity does within one game loop, this includes validating position, updating the velocity, rotation, and any other behavior which the entity might perform. Players and monsters have a *die* function which executes when they die, abilities have a *hit* function which deals with the behavior of the ability hitting another entity, as well as some other private helper methods.

Other properties in common
All data structures within our game do have something in common which one could say they 'inherit' from a base class, It's not exactly what happens behind the scenes but it is as close as it gets to a base entity class.

All entities have a ***map*** and a ***sector*** property, which are used internally to manage which array the entity is kept in during runtime. all entities have a ***type*** property which is used to determine what "class" the entity is an instance of. It is used for duck-typing.
All entities also have an ***old*** and ***new*** properties. these properties are booleans which act as flags to know within our game loop when an entity just started its life cycle or when it is finishing it. They are very useful for the client to know when to draw certain animations or play certain sounds when entities are old or new.

<u>Data Structures on the Client</u>

Our design

On the client side all the data structures except one are the same as the servers with a few minor changes. The only data structure that is unique to the client is the structure we use to update the server with new information from the client such as: new messages,new player location,item changes, abilities used,etc. There are some differences between some of the client data structures compared to the server ones. The player data structure is similar to the server one except it has a few extra properties and methods to keep track of the velocity, accel, friction for the client side physics engine. The client also has no functions on all the objects (maps,enemies,items) that it receives from the server and the definitions(static properties) stay in a separate object instead of being added to the objects the client receives from the server.

Because having constructor functions in the client side for the simple sake of organization is a big overhead within the client, the client simply depends on the structures it receives from the server and tests if it has certain properties to make sure everything is OK.
Constructor functions also felt unnatural, not to mention takes more processor time, to use since nothing is created on the client; all creation and destruction of objects is within the game server. The client simply knows what the server tells the it, the client CANNOT be in charge of creating any game entities as it would be terribly unsafe.

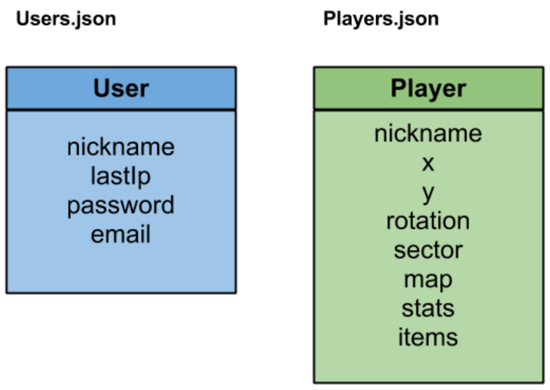
Database Design

Our design
Database design within our project was a bit different than the usual scenarios we've had to deal with in the past. Since our game loop runs 60 times a second, and the state of our information changes constantly. we had no choice but to use something efficient for data storage and for persistent data storage.

All of the data in our game, for example: Maps, sectors, players, items, monsters, the player's statistics, entities' locations, attacks, etc, are simply stored in RAM. querying and committing to databases hundreds of times a second would be too slow for our needs and would make the game unplayable.

For persistent storage, we decided to simply write and read to files because of its speed advantage and simplicity. Depending on the settings on our game server, the game will back itself up into a JSON file every now and then. Since our game data is in RAM in form of javascript objects, storing it in a file in JSON notation requires no parsing; it is as simple as writing the objects into the file, and loading them at the beginning of the server's lifecycle.

What do we store?
The only things we need to store in case the server either crashes or it is restarted are two things: the users and the state players. This is what they look like:

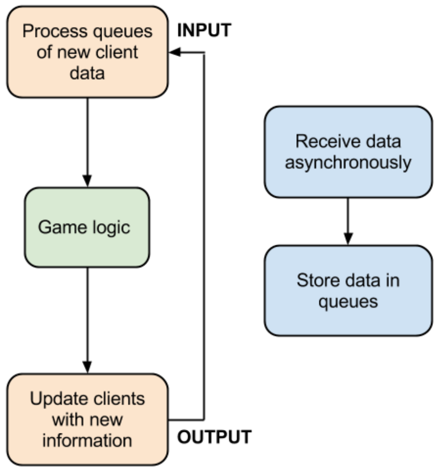


Network Interface (Input/Output Design)

Our design
The input from clients into the game server and the output from the server to those clients was one of the hardest things we've had to do during our project as it had to be very robust, efficient, and safe. It is a huge part of an online multiplayer game and a huge part of our overall system design.

During the lifecycle of our game server, many messages are received asynchronously from many different players. The messages usually contain updated coordinates, whether the player has tried to pick up any items, drop any items, switch them within his or her inventory, or even if an ability has been attempted to be used.
Since these messages can come at any point within the game loop, we decided it was important to queue them and process them in batch at the right time. The order in which things execute within our game loop or any for that matter is extremely important.

After messages received from clients are queued, we begin to process them at the beginning of our game loop, it is the very first step within our game loop.
Once processed, and validated, many other modules go into action such as the monster spawning module, or the memory module to save state, and eventually and at the end of the game loop, we update the clients with new information.

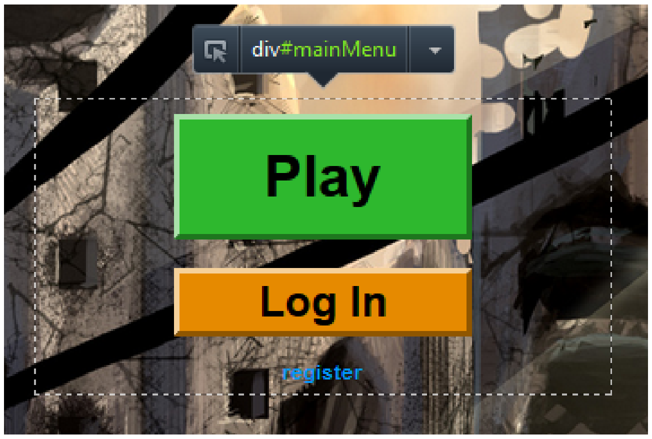

Process queues of new client data
INPUT
Game logic
Update clients with new information
OUTPUT
Receive data asynchronously
Store data in queues

This new information is carefully picked out to be as small as possible, but not small enough for the client to miss anything that is happening within the server. Since our maps are divided in sectors which contain collections of different types of entities, we only send the information regarding the sector the client is currently in as well as the 8 sectors around him.
Our map data structure has been carefully planned out so that this step becomes a breeze. Imagine how much processing power would be needed to parse and filter the entire map and customize it to every player.. every game loop.. every couple milliseconds! Instead, it's as easy as sending the state of the collections near the player's location.

User Interface Design

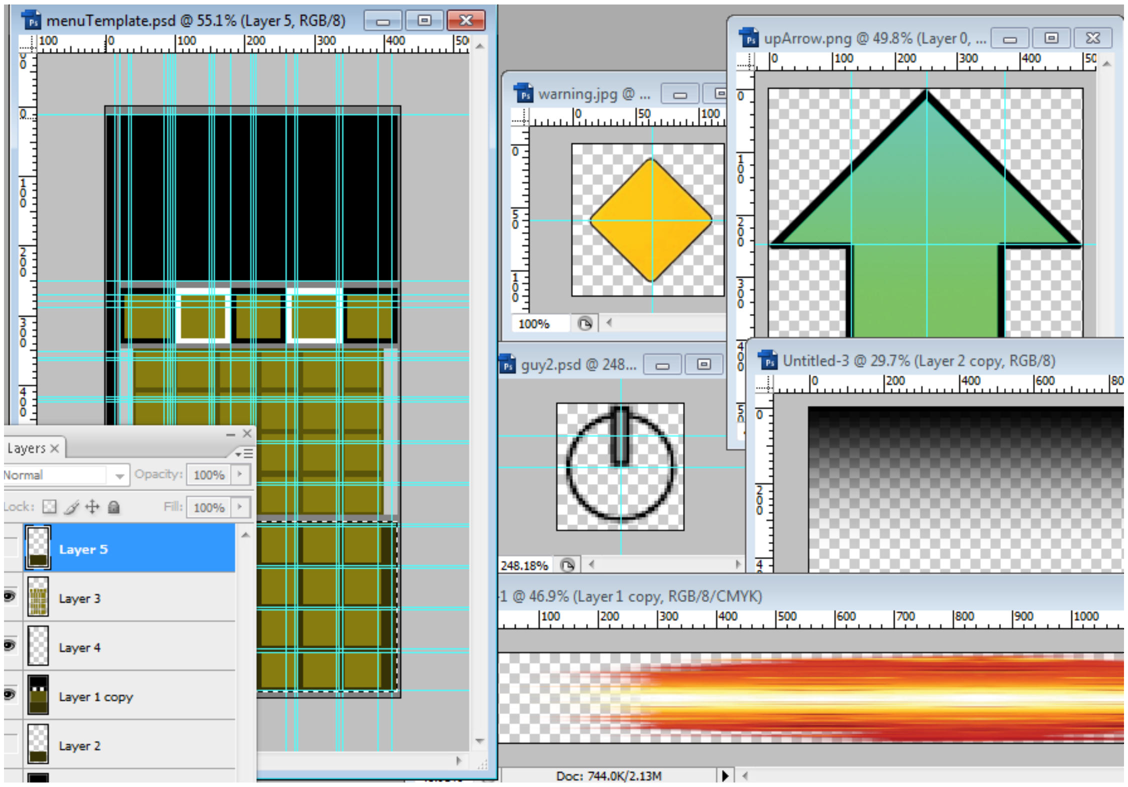
Our design
User interface was heavily thought about during the length of our project. While it is something we originally designed, it went through some changes as our project matured to make it feel just right.

The website
The website design was designed to be simple, minimalistic, and smooth. Only the essential options are within the main page: play, login, and register.
We wanted the user to be able to jump into the game straight away, without the hassle of having to create an account. All the different pages within the website are loaded on the same DOM container and loaded asynchronously when requested, while that functionality is behind the scenes, it was a core design principle for the transitions in between pages, without the need to reload the whole website again and with a nice fading effect.



The game

The game's interface stays true to many traditional MMO style of games, most of the elements are there: the stats, the inventory, the HP bar and the chat.
Many usual elements such as having a minimap in the corner were intentionally removed, and many unusual features such as a screen tall bar reflecting the danger level were added.


menuTemplate.psd @ 55.1% (Layer 5, RGB/8)
warning.jpg @ ...
upArrow.png @ 49.8% (Layer 0, ...
guy2.psd @ 248...
Untitled-3 @ 29.7% (Layer 2 copy, RGB/8)
1 @ 46.9% (Layer 1 copy, RGB/8/CMYK)
Layers
Normal
Opacity: 100%
Lock:
Fill: 100%
Layer 5
Layer 3
Layer 4
Layer 1 copy
Layer 2
100%
248.18%
Doc: 744.0K/2.13M

Within the game canvas, other than the actual game's visuals, we can see the chat at the bottom, and the current danger level of the area at the top right. What was truly hard to design was the sidebar where everything else is located at.
Everything within the sidebar has been carefully measured and thought of in advance. From top to bottom we have:

<u>First section</u>
- Nickname and player level
- The experience bar
- The HP bar
- The AP bar
- Number of player kills, monster kills, and kill/death ratio

<u>Second section</u>
- The equipped items which are ready to be used
- The player's inventory

<u>Third section</u>
This section alternates between what is shown depending on the game state.
If the user is accessing his inventory, a description of the item which you are currently hovering on will be displayed in this section.

If the user is standing on items on the floor, this section will show those items in the same fashion that player items are shown in the inventory. The player grab whichever items he likes and drag them to his or her inventory.

Lastly, at all other times, that section simply shows the players' stats such as: strength, dexterity, defense, speed or luck, as well as any bonuses applied to those stats by items being worn at the time.

Next Step (what we will do in the future)

With the month and a half we had to work on this project we managed to do a small part of what we really want to accomplish. The project as is right now represents and alpha build or even a prototype of what we are hoping to achieve. We will keep working on this project in the future and attempt to turn it into a fully functional massively multiplayer online game. There's still a massive amount of work left to put into this game, heres a few use cases we would like to ad:

- a functional donation system.
- facebook/google login
- being able to change a registered user's password
- being able to register from a guest account
- creating more complex abilities (shields, dash, etc)
- ability to have friends
- ability to create alliances
- player/enemy animations
- optimizing network communication
- use some predictive algorithms for laggy users
- much larger maps
- more control options(mouse/gamepad)
- ability to use custom controls(key mapping)
- item trading
- duel arenas
- banks

We found working on such a complex network and processor intensive project really forced us to understand all the tiny little details about javascript and programming in general. Having to optimize everything forced us to think in depth about how the programming language functions at the most fundamental level. It made us better javascript programmers and gave us insight to how computers function. It put to use our mathematical skills: trigonometry for rotations, derivatives for velocity and acceleration.

We had some horrible experiences and some amazing ones while doing this project. It demonstrated how difficult working in a team can be, even when its your best friend. At times we hated each other, other times we laughed uncontrollably. Sometimes the project weighed down on our souls other times, it made us feel proud. Without a doubt, we will keep working on this project once we have graduated from LaSalle. We will take the time to celebrate our graduation and we will be proud of our achievements, but we know theres still a long, difficult journey ahead of us, but now that we are atop of the hill we see a beautiful view.

Acknowledgments

Background on main page by dinmoney
http://dinmoney.deviantart.com/art/ruin-city-107150180

Loading bar animation was taken from:
http://www.ajaxload.info/

Temporary main music theme by Logan
http://zweihander.bandcamp.com

Tilesheet by Mataraelfey
http://mataraelfay.deviantart.com/art/tileset-3-rpg-maker-xp-271510691

Item icons from an unknown source.

Appendix