Software Design Description

for

Second Screen Experience

Version 1.0 approved

Prepared by Connor Brooks,

Matthew Clark,

Christopher Goulet,

And Jared Prince

Western Kentucky University

3/3/2017

Table of Contents

Table of Contents ii

Revision History ii

1 Introduction 1

2 System Overview 1

3 Design Considerations 2

3.1 Assumptions and Dependencies 2

3.2 Goals and Guidelines 2

4 Architectural Strategies 3

5 System Architecture 3

5.1 Website Front-end 3

5.2 Website Back-end 5

5.3 Game Server 6

5.4 Game/Controller API 6

6 Policies and Tactics 7

7 Detailed System Design 7

7.1 Website Front-end 7

7.2 Website Back-end 8

7.3 Game Server 9

7.4 Game API 10

8 Acronyms and Abbreviations 14

9 Bibliography 14

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason For Changes** | **Version** |
|  |  |  |  |
|  |  |  |  |

# Introduction

This document is designed to be a reference for any person wishing to implement a game using or any person interested in the architecture of the Second Screen Experience.  This document describes the architecture, interfaces, and motivation behind the web server, game servers, website, and game API.  Both high-level and low-level designs are included in this document. This document should be read by an individual with a technical background.

This document includes but is not limited to the following information for the Second Screen Experience: system overview, design considerations, architectural strategies, system architecture, usage, and detailed system design.

# System Overview

High Level Design

https://lh6.googleusercontent.com/Pfoicl6Xcm7BFlwJNH1UnbaijtoAviy6EJJ_pDIY2PyNH53h-KfJdtwWNB4HNj9U6x7EgtkcjFEHk6bb3ADOHPv2SicwXsRCJ6n6HhGredngVisqgfZTk9Qz861e5Rg0cRO7UF9b

# Design Considerations

This section describes many of the issues that needed to be addressed or resolved before attempting to devise a complete design solution.

## Assumptions and Dependencies

The Second Screen Experience makes some assumptions about software and hardware for the server systems and clients.

Both the machine hosting the web and game servers and the clients (both the screen host and the controllers) make the following assumptions about their environment:

* The system can be described by the environmental requirements associated to this document.
* The system the application is executing on will have the required resources available as necessary.  This entails sufficient memory and permanent storage space, an adequate CPU for the necessary application, and a TCP/IP network connection.

The machine hosting the web and game servers makes the following assumptions about its operating environment:

* The machine has opened and forwarded (if necessary) ports 80, 443, 2000 and 3000 with sequential ports above 3000 opened and forwarded based on the max number of games the machine can support.

The clients (both the screen host and controllers) make the following assumptions about their operating environment:

* The clients have access to a web browser with HTML5, JavaScript, and CSS3 capability.

## Goals and Guidelines

The first goal of the Second Screen Experience is to provide a platform for groups of users to play multiplayer games using a single host screen as the display and their smartphones as controllers. This platform must be efficient and simple to use.

The second goal of the application is to ensure that communication between the server and the user and between the controller and the game are fast and reliable. Slow communication will lead to lag between the controller and the game, which will negatively impact the user experience. Unreliable communication will potentially cause problems during gameplay. Because of this the amount of data to be communicated between the server and controller should be limited as much as possible.

The third goal of the application is to offer a simple game API to allow amateur or professional developers to modify their games to fit the second screen model. They should then be able to submit their games for review and (assuming the quality of the game is acceptable) have them added to the games available for play.

# Architectural Strategies

The Second Screen Experience design is broken into 3 subsystems: website, game server, and game/controller API.  The website is broken down further into the front-end and back-end systems.

The website’s major design considerations include a clean interface to allow quick game room creation and easy navigation.  The website is designed so that it will be easy to add new games after a manual review process and have players be able to create games of that type and see all detailed information about it.  The back-end of the website is designed so put little strain on the game server to request a room and minimize the connections going to the website front-end.

The game server’s major design considerations include as little possible delay for controller information pass-through to the screen host and limiting processing for the server machine.

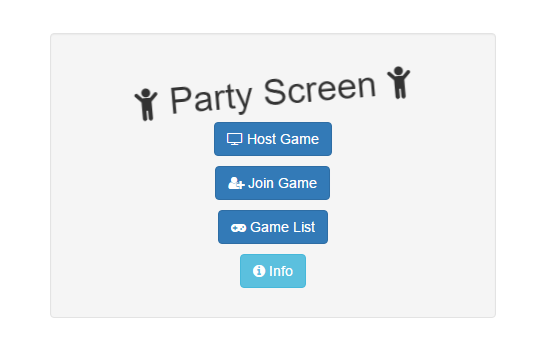
The game API is designed to make it easy for developers to create new game modes by giving them access to website components, like the HTML5 game screen controller button layouts, through functions.  This will also make it easy for developers to port existing games to our system easily as well.

Although our web/game server subsystems can run on a Linux or Windows operating systems, we have optimized for a Linux operating system since it will allow us to minimize operating system resources to maximize our system’s performance.  We chose to program the web server using Go to minimize web server resources, and the game server in C++ to minimize the game server resources.  C++ will allow us to remove a lot of the overhead of modern languages while still allow us to use OOP practices and develop the system quickly.

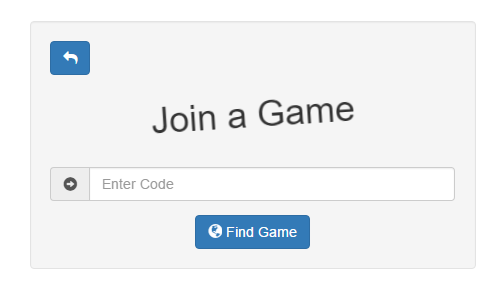
# System Architecture

## Website Front-end

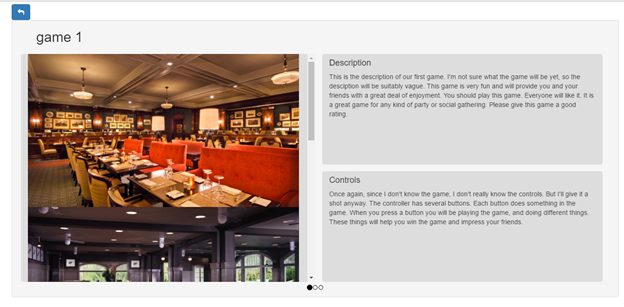
The website consists mainly of 5 sections. The first, the main menu (below), simply has links to the other three menus (host, join, and game), along with information about the layout and use of the website for users.



The second section is the host menu. From this menu, the user can begin hosting a game, choosing a game and any game settings needed. Once all options are confirmed by the user, they will be redirected to the host screen page. It is this page which will display the game itself, along with the unique code used to connect to the game. The next section is the join menu (below).



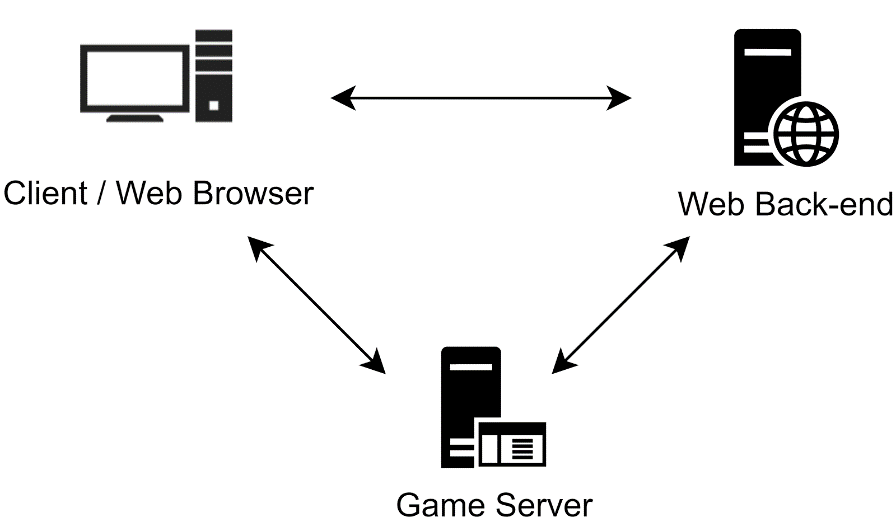
From this menu, a user can enter a game code and select and player specific options available for the game. Once they confirm these options, they will be redirected to the controller page. This page is what will display the controller and allow the user to interact with the game. It is assumed that the device used to access the controller is a smartphone.



Finally, the game menu (above) consists of a series of informational boxes, one for each game. These boxes have the game name, description, and instructions for play, along with a series of screenshots from the game. This will allow users to learn about the available games. This information is retrieved from a .json file provided with a game submission. This file is required for the game to be considered.

## Website Back-end

The role of the web back-end in our system is to be a place to initiate our web application and provide central repository of files. The key roles for the backend is to deliver the files to the users and initiate the communication between the game server and the client. These files include the HTML, CSS, and JS files that make up our front end and the games and their associated files. The backend initiates the communication to the game server by acting as a middleman between the client and the game server since the client has no means to know how to communicate with the game server without the provided JavaScript files and the game server address.



An important differentiator between the game server and the web back-end, is that the website backend does not maintain any persistent data about the client. This means that the web server treats all requests the same. The reason for this is for security, modularity and scalability.  Without any persistent user data on our web back-end, there is no personal information to steal, and there are no user account permissions to manage. The lack of accounts was a conscious decision to simplify the user experience and engage them as quickly as possible instead of burdening them with registration and login. Since every request is essentially is treated the same regardless who sends it, the Web Back-end can be scaled across multiple nodes if needed. A node in this case would be a content server of some sort. Every request could be sent to a different node and the client would get the same content. The only part that needs to maintain some persistency is the game server, which is why it is always denoted separate from the web-backend.

## Game Server

The role of the game server is to group controllers into a “game room” with a host screen and allow the controllers to forward their button presses over the connection to the screen.  The game server allocates a maximum number of rooms and when a client requests a new room through the website, it sends the port and room code to the host.  When clients (controllers) request to connect to a room via the room code, it checks if that room exists, then provides the port if it does.

Once the screen host and controller clients are connected, the screen can send game information, such as the game mode, to allow the controller to change its setup.  The controller can then send its button presses and any settings, such as the controller user’s name, to the screen host.

## Game/Controller API

The role of the Game API is to facilitate creation of new games by developers. The API provides an object structure for game developers to use, as well as hooks that the games must use in order to run on the Second Screen Experience system. The actual execution of the game is controlled within the API. This also will allow for further updates on the platform involving lobby joining or pre-game timeouts that will not require updates to the actual game instances.  
  
Currently, controllers are fixed based on predetermined options, but a Controller API will also be created that will allow developers more freedom in determining user experience by customizing the player’s controller. This API will function similarly to the Game API, with a few predefined hooks that the developer must use. The developer will be able to specify how many buttons, where they are placed, and how they function (i.e. push buttons vs joysticks, etc.).

# Policies and Tactics

The Second Screen Experience system was broken into sections to allow each individual to develop their pieces individually and have clearly defined interfaces between components.

# Detailed System Design

## Website Front-end

Along with the javascript code to run a game, each game will have the controls, descriptions, etc. in a .json object with the following format:

“Game Name" : {

“options” : [

“color” : {

“name” : “Player Color”,

“choices” : [“Red”, “Blue”, “Green”],

“values” : [“#FF0000”, “#00FF00”, “#0000FF”]

}

],

“gallery" : [“screenshot-1.jpeg”, “screenshot-2.jpeg”],

"name" : "The name of the game.",

"description" : "A description of the game.",

"controls" : "An explanation of the game controls."

}

The first sub-object contains all options for the game. Each option has a descriptive name, a series of descriptive choices, and a series of values. The each choice has a corresponding value which is passed as an argument when the game is created. In the above example, for instance, the option has a title of “Player Color”, choices of green, red, or blue, and the hex color values of each. The other four variables are the displayed in the game list menu.

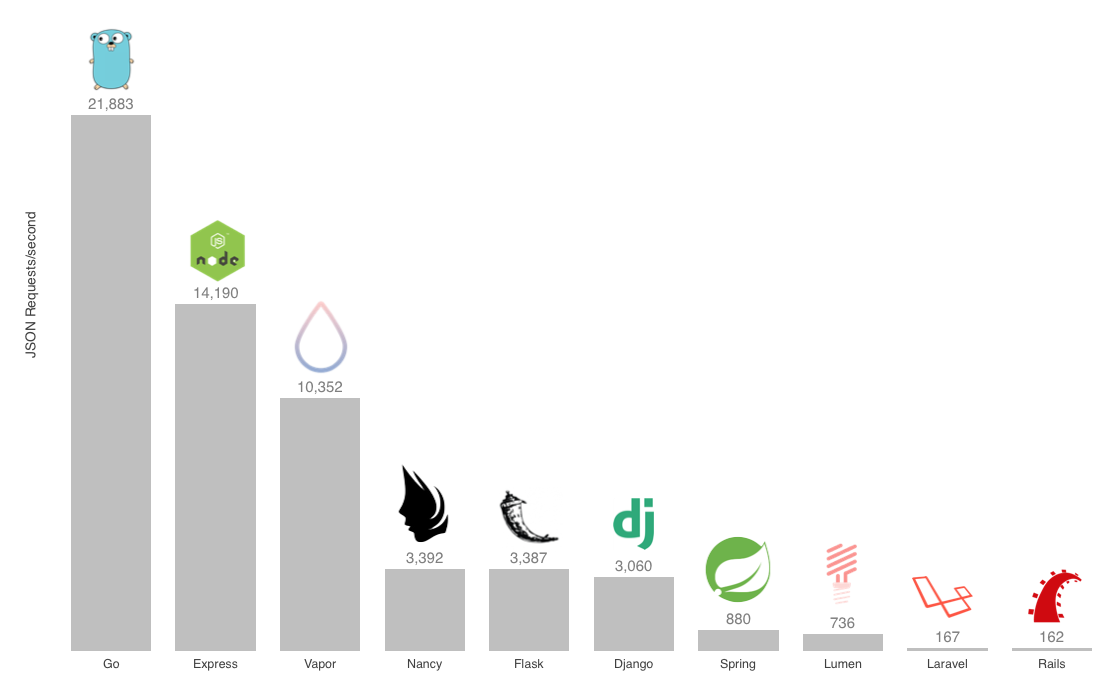
When a user hosts a game, the website will send a request for a new game lobby to the server. The request will include a unique identifier for the selected game and all the parameters needed for the game (ie. the game options). Likewise, when a user enters a code to join a game, the website will send a request to the main server to be directed to the game server with the given lobby code.

requestNewLobby(gameid, gameParameters[])

joinLobby(lobbyCode)

## Website Back-end

The key roles for the backend is to deliver the files to the client and initiate the communication between the game server and the client. To do this we needed to select a platform that we could build our code on that had these features. Since security and performance are huge considerations for our project, we wanted to have complete control on how each component worked with each other. A very popular way of doing this is to use a LAMP stack. This means to use the Linux operating system, apache webserver, MySQL database and PHP.  For our goal of creating a secure and performance focused product, this stack was not suitable for our needs. A database was not needed and thus an installation of one would be unnecessary. A dedicated web server would be needed to configured properly every time a new instance of this project would be deployed and misconfiguration of it could present a security risk. The introduction of a server side language that is on top of the web server presents a layer a complex when it comes to both security and performance. PHP is a scripted language and then cached when the first instance of the script is running, but the performance of this is not the best. Larger sites, like Facebook, compile their PHP code into binaries using a custom compiler that they programmed called HipHop.



**Figure: Go Performance vs Other Popular Web Frameworks**

To solve these problems, we chose to use the Go programming language and its standard library to deploy and power our web application. Go is an open source, compiled, garbage collected, statically typed programming language that was created by Google. One of Go’s many great features are its main concurrency construct, the goroutine. Goroutines are lightweight processes that are managed by the Go runtime. This means many goroutines can run on a single thread, which allows for more concurrency to happens. Go has a rich standard library that has all the features we need to power our project. The following packages are what the web application uses now.

"html/template" data-driven templates for generating HTML output safe against code injection.

"net/http" - HTTP client and server implementations.

"encoding/json" - encoding and decoding of JSON

“Net" - portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets

Using this language also means that the whole web application is one binary, if recompiled, it is also portable across different operating systems. Thus, to redeploy, the binary can be download and ran. This simplification means less risk for misconfiguration.

## Game Server

The game server handles sending “game room” information to the web server and creates a pass-through for the controllers to the host screen.

### Sample Methods

*Method:*

int nextEmptyRoom(std::vector<GameRoom\*> vecGameRooms)

*Purpose:*

Returns the index of the next empty room in the pool

*Parameters:*

vecGameRooms - vector array of all the game rooms in the pool

*Method:*

int getRoomFromCode(Json jsonData, std::vector<GameRoom\*>\* vecGameRooms)

*Purpose:*

Returns the index of the room matching the given code

*Parameters:*

jsonData - data with room code in it

vecGameRooms - vector array of all game rooms in the pool

*Method:*

void GameRoom::initGame(json11::Json jsonData)

*Purpose:*

Initialize the game room’s values and start listening for connections

*Parameters:*

jsonData - room data (room code, port number)

*Method:*

std::thread GameRoom::initGameThread(json11::Json jsonData)

*Purpose:*

Returns a thread that wraps the initGame() function for multithreading

*Parameters:*

jsonData - room data (room code, port number)

*Method:*

void GameRoom::generateRoomCode(std::vector<GameRoom \*> \*vecGameRooms)

*Purpose:*

Generate the 6 character room code used to join a room

*Parameters:*

vecGameRooms - vector array of all game rooms in the pool to make sure a collision doesn’t occur

*Method:*

void GameRoom::generateUUID()

*Purpose:*

Generate the UUID of the host screen so only one host screen can be in the room at a time

*Method:*

void GameRoom::sendHostUpdate(std::string controllerData)

*Purpose:*

Send controller updates to the host screen

*Parameters:*

controllerData - wrapped data from the controller to send

Method:

void GameRoom::controllerBroadcast(std::string msgBroadcast)

*Purpose:*

Broadcast data to all controllers in the room

*Parameters:*

msgBroadcast - message to send to all controllers

*Method:*

void GameRoom::readClientUpdates(SOCKET clientSocket)

*Purpose:*

Read controller button presses or other updates

*Parameters:*

clientSocket - socket to listen for new information from

*Method:*

void GameRoom::processConnections()

*Purpose:*

Listen for new connections to the game room

## Game API

The Game API consists of three hierarchical object types - Game, Canvas, and Dynamic Object. To illustrate the usage of the Game API in detail, after explaining each object’s structure, we will walk through the creation of a simple “Pong”-style game.

### Game Object

To begin, a Game object is instantiated by passing in two arguments representing the minimum and maximum number of players, respectively. The game lobby will use these numbers in determining when the game is eligible to start. Games will be loaded on a blank HTML page. Because of this, games are responsible for creating the Canvas (and any other HTML) elements needed for display. This can be done by using the *addCanvas* function of a Game object, which receives an ID for the canvas, the width, height, and styling options. Styling options can be used to specify positioning. This creates both an HTML canvas element, as well as a Canvas object as defined by the Game API. Alternatively, if other HTML objects are to be added to the game, they can be added using the *addHTMLObject* function. Canvas objects, once created, are stored in the *canvs* dictionary of the Game object, and can be accessed using the ID used in their creation. Another important feature of the Game object is the *params* object contained within it. This JSON object contains all parameters and global variables used in the game, and can be added to at any time using the *addParam* function. The remaining Game functions are *setFrameRate*, which sets the frame rate for the game, *setControlHandler*, which allocates a function for handling input, and *startGame*, for beginning a game once ready. Note that this last function will not technically begin the game, but rather set the status of the game as ready to begin. Once the lobby approves the game’s start, the API will actually begin the game.  
In summary, this is the public structure of the Game object:

Fields:

* *params* - JSON object that stores data during the game, referenced by unique IDs
* *canvs -* Array of Canvas objects, stored by ID
* *htmlObjects* - Array of HTML objects, stored by ID

Functions:

* *Game(min\_players, max\_players*) - Constructor for the Game object
* *addCanvas(ID, width, height, styling)* - Creates and registers both a new Canvas object and a corresponding new canvas HTML element.
* *addHTMLObject(object, ID*) - Registers an already-created HTML object that can be referenced in the game
* *addParam(ID, defaultValue)* - Adds a new parameters
* *setFrameRate(number)* - Sets new frame rate (in frames/second)
* *setControlHandler(function)* - Sets *function* as the handler for controls input
* *startGame()* - Called once the game is ready to begin

### Canvas Object

Next, the Canvas object interacts with a specific canvas element, as well as holds all DynamicObjects that are within that particular canvas. A Canvas object should not be instantiated directly by the developer, but should instead be created by calling the *addCanvas* function on the Game object. This ensures all Canvas objects are tied to a single Game instance. The function *setBaseState* is used to set the default background of the canvas such that each time the canvas is redrawn this background will be used. The other public function, *addGameObject*, creates and registers a new DynamicObject to this Canvas. There are four important fields on the Canvas object. First, *htmlCanv* contains a reference to the corresponding HTML element. The *game* field is a reference to the Game instance which holds this Canvas, and *ctx* is the drawing context that should be used by all drawing functions. Finally, *dynamicObjects* stores all DynamicObjects registered to this Canvas. Here is the summary of the Canvas object:

Fields:

* *htmlCanv* - A reference to the corresponding canvas HTML element
* *game -* A reference to the containing Game object
* *ctx* - The drawing context that all DynamicObjects within this Canvas should use
* *dynamicObjects* - An array of DynamicObjects, stored by ID

Functions:

* *setBaseState()* - Sets the current state of the Canvas as the background to draw
* *addDynamicObject(ID, drawFunction, parameters, updateFunction)* - Creates and registers a new DynamicObject to this Canvas

### DynamicObject Object

Finally, the third type of object is the DynamicObject object. This object stores instructions for drawing and updating a single sprite or other game entity. Its constructor should not be used, but rather the *addDynamicObject* function of the Canvas object should be called to create a new DynamicObject object. The only field of the DynamicObject that should be directly accessed after its instantiation is the *expired* field, which is used to signify that the object should be deleted.

Fields:

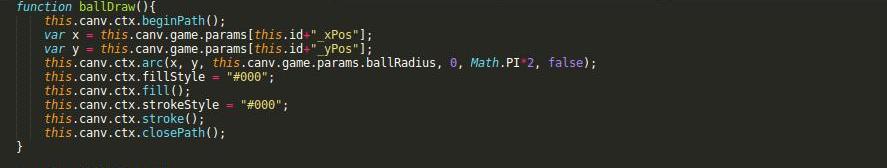
* *expired* - Boolean that flags if an object should be deleted

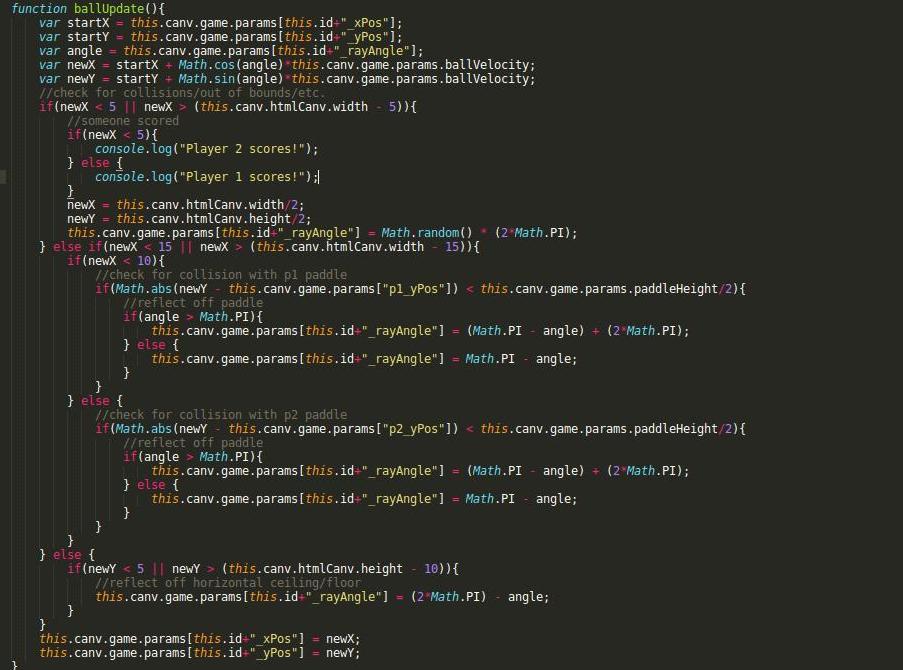
Functions:

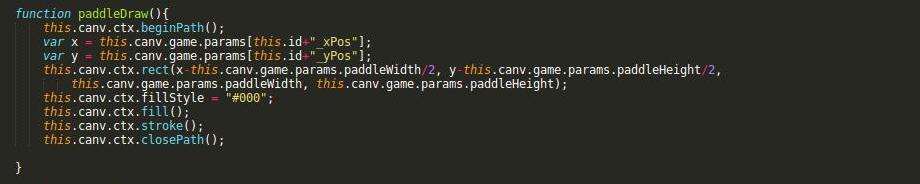
* (no publicly accessible functions)

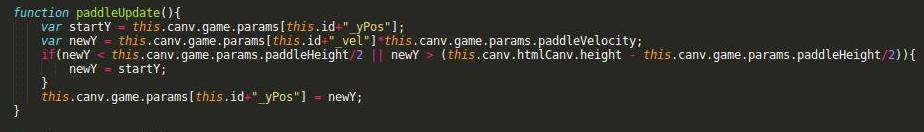
### Sample Pong game

Now that the full structure of the API game has been explained, we will walk through the creation of a simple Pong game using this API. First, we create functions for drawing and updating ball objects and paddle objects using HTML5 Canvas commands:



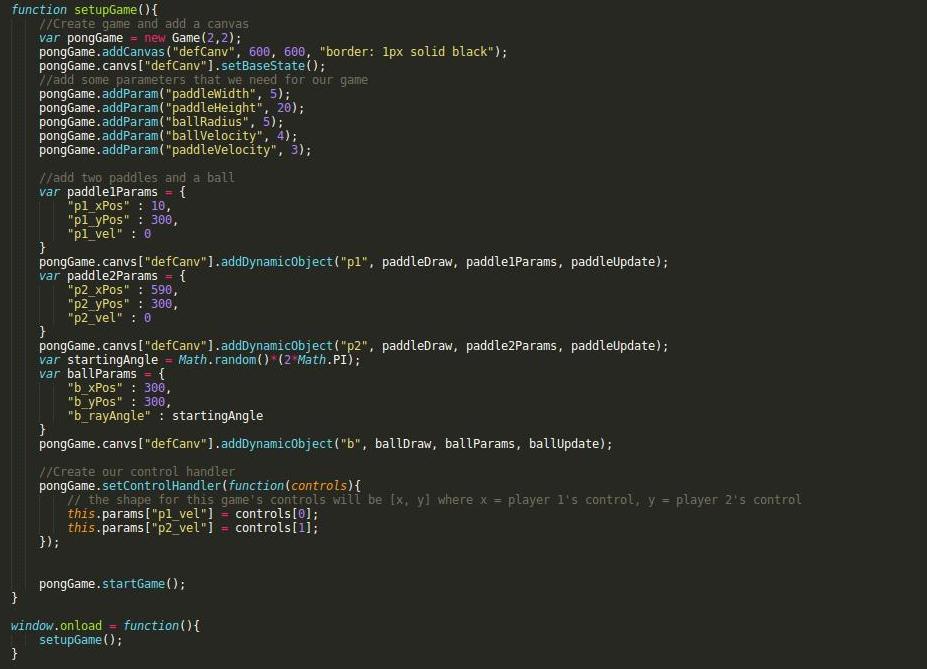






Note that these four functions make heavy use of the Game object’s parameter list in order to store and retrieve data necessary for drawing/updating the positioning of the objects.

Then, once these four functions are defined, we can create our game. We do this by instantiating a Game object, registering a single Canvas object to it, then registering two paddles and a ball (all three are DynamicObjects) to the Canvas object. Parameters are used to set the default positioning and movement of each DynamicObject, as well as some unchanging game information (such as the size of the paddles). A control handler is registered that changes the direction of the paddles based on user input, and the Game is started.

****

Though the drawing and updating functions can become somewhat lengthy (especially if involving things such as collision detection, as the ball updating function here did), the creation of a game is relatively straightforward. This example game was created in under 130 lines of code, with less than 50 after discounting the drawing and updating functions!

# Acronyms and Abbreviations

SS - Second Screen

# Bibliography

Software Requirements Specification for Second Screen Experience v1.0