Fall 2012

JAVASCRIPT GAME ENGINE FOR MOBILE USING HTML5

Nakul Vishwas Natu
San Jose State University

JAVASCRIPT GAME ENGINE FOR MOBILE USING HTML5

A Writing Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Nakul Vishwas Natu

December 2012

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

JAVASCRIPT GAME ENGINE FOR MOBILE USING HTML5

By

Nakul Vishwas Natu

APPROVED FOR THE DEPARTMENT OF
COMPUTER SCIENCE

Dr. Chris Pollett, Department of Computer Science	Date
Dr. Soon Tee Teoh, Department of Computer Science	Date
Mr. Anirudha Shinde, M.S. Software Engineering	Date

ABSTRACT

JavaScript Game Engine for Mobile using HTML5

The goal of this project was to develop an HTML5-based mobile JavaScript game engine. Developers can use this game engine to create platform independent mobile games. It provides support for sprite animation, physics, event handling, orientation handling, collision detection and entity management. HTML5 local storage was used to achieve game persistence.

The games created using this game engine can be stored on the home screen of the device as a data URL. When the user launches the data URL, the game opens up in a browser and the entire process is similar to launching a native application. HTML5 has many restrictions on its usage of local storage and connecting it to the data URL was a challenging aspect of this project.

In order to support our project, we created test games using this engine and also provided an app store to distribute games that were created using this engine.

ACKNOWLEDGEMENTS

I would like to express my immense gratitude to Dr. Chris Pollett. Without his help this project would not have been realized. I would like to thank him for his continuous support at every step and pushing me to my limits. I would also like to express my appreciation for the help received from my committee members Dr. Teoh and Mr. Shinde. I would like to thank them for spending time on reading my report and giving excellent feedback on my project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Smartphone usage is on rise. As of 2012, almost half of mobile users are smartphone users [14]. Every day we see major technological enhancements in mobile phones and tablets such as faster processors with more RAM and enhanced GPU capabilities. If we do a comparison amongst Samsung's flagship phones over the last three years, we can see that processing power has almost doubled with each model [4]. Today, we can effortlessly run graphic intensive applications on smartphones and it has become a very large market for games.

Currently, the market offers a large variety of mobile devices with different platforms such as iOS, Android, Blackberry, and the latest Windows 8 operating system. The mobile device market has split platform usage statistics. Android has around 51% of the market share, while iOS has 33% [7]. Mobile games constitute a major share of the applications in these various operating systems. They are amongst the most successful, and highest grossing apps in app stores today.

Each phone platform available in the market is based on a different technology. Namely, iOS apps are written in Objective-C, Java drives Android application development, and Windows Phone development is done in C#. In order to develop an application which runs on all these platforms, one needs to learn and master all these technologies. The investment in terms of learning, designing, development and testing games on all of these platforms can be hard for smaller companies and start-ups. The question is, can we use some other technology such as HTML5, which is also rapidly growing?

In this project, this issue is specifically addressed. "MobiGameJS" is a game engine designed to create portable games. This game engine supports data persistence, sprite animation, physics, event handling, and collision detection. Applications can also be deployed in a way that is very similar to running a native application. The

games created using MobiGameJS engine can be run on different mobile platforms with a minimum of modifications.

A lot of developers today develop games using game engines, as it simplifies the design process and reduces the time to design radically. There are numerous frameworks present in different languages such as Box2D in C++, Unity3D in flash, Crafty in JavaScript, etc., that are created to develop desktop or console based video games. However, a game engine designed specifically to develop mobile games is of great interest and also is an upcoming field in today's market. We experimented with Crafty, a JavaScript game engine, in the process of designing MobiGameJS. We created some games to better understand the working of the framework. These experiments are discussed in the later part of the report along with my other findings.

In order to create MobiGameJS, we leveraged JavaScript and HTML5 functionalities. These include canvas for drawing and local storage for data persistence. Our engine supports gravity, elastic collisions, and spring forces. We also used data URI for scaling down the game and also to provide the user with a seamless look and feel of a native application. Three test games were created with each testing a specific set of functionalities provided by the engine. A small website acting as an app store for the games was created using MobiGameJS.

This report is divided into five major sections. In the first section, we discuss game engines, the importance of HTML5, JavaScript and previous experiments. In second section, we look into the actual design of game engine MobiGameJS. We give a detailed explanation of the different components of the engine and their workings. We will also discuss the monetization aspect of this game engine. We then discuss the different test games that we created by using the engine and their purpose. Finally, a conclusion is given with a mention of the references.

2. Background

This section describes the idea behind this project and the functionality of different parts of this project. The pros and cons of some specific technologies that were selected for the implementation of this engine and some of the experiments carried out in the beginning of this project are also discussed. We will also see how these experiments helped us in the implementation and design of the game engine. The game engine, HTML5 elements, JavaScript, and data URI are discussed in the later sections of the report.

2.1 Game Engine

The core part of our project lies in the design of the game engine. According to Chris Stead on IGN blogs, a game engine is defined as an “overall architecture to develop and run a game – it gives developers tools to create the disparate elements of a videogame and then pull them together to create a functioning whole. From the renderer to the physics system, sound architecture, scripting, AI and networking, game engines either natively power every aspect of a game, or they allow other specialized middleware to slot into the game's framework. In any case, game engines are the workhorses of modern videogame development.”[3]. But, from the perspective of a game developer, a game engine is the code that will convert the static entities into a dynamic, interactive game.

The different elements present in a game get updated very frequently. A lot of work is required in order to display them, update and draw these game elements on the screen to the user in real time. A game also contains animations, physics, event handling, etc. Therefore it is appropriate to incorporate all those features into a single piece of code. That code will act as a building block for any game. It will also help the developers reuse the code and reduce their development time.

As discussed in the previous section, the main crux of game engine is the code, as it handles the update of objects and then draws them. Two important parameters are the screen's refresh rate and the frame rate. The frame rate is measured in frames per second. It calculates and dictates how many times a particular frame is drawn on the screen. An element's requirement on the screen needs to be checked before it is drawn on the screen. Each game element's position needs to be updated as well its collision course with another element. Next, the decision of when we need to draw those elements on the screen is processed. This mechanism is called, Game Loop and it forms the heart of our framework. A lot of features in a game change without any user interaction. This feature change is similar to quickly drawn frames of paper animation. The display of the game on the screen might not display correctly if we wait for the user events to happen. Therefore, the game loop architecture is used in the game engine instead of the event driven architecture.

Feature	Box 2D	Unity 3D	Crafty
Sprite Animation	No	Yes	Yes
Collision Detection	Yes	Yes	Yes
Media Management	No	Yes	Yes
Data Persistence	No	Yes	Yes
Mobile Event Handling	No	Yes	No
Physics	Yes	No	No
2D/3D	2D	3D	2D
Networking	No	Yes	No
Appstore/Delivery	No	No	No

Table 1: Comparison of different game engines

The table above gives a comparison of the different game engines that we have considered. As can be seen, there is no single game engine available that supports all the functionalities listed. This comparison has been used to decide the requirements of our game engine and is described in detail in the following section.

2.2 Why JavaScript and HTML5?

The main goal of this project was to make it portable on different mobile platforms such as iOS and Android. Both these platforms use different technologies such as Objective C and Java; we needed to create a common platform for both the platforms so that the developers using MobiGameJS would not be required to learn two different technologies. The obvious choice was to use web based technologies since both the platforms support different browsers and have them mainly synched with their desktop counterparts.

We also had an option to use technologies such as Flash or Adobe AIR technologies, but they are not as universal as HTML5. Also, HTML5 was chosen for experimentation because of its progress visualization and performance. JavaScript became the first choice of language to write the game engine when the decision of using HTML5 was made. It is a powerful language and it also supports object-oriented principles such as classes, inheritance and polymorphism. Apart from those features, it is also used by majority of developers, so the learning curve would be really small.

The major reason for choosing HTML5 was its graphic capabilities. HTML5 has shown progress in this field with SVG and Canvas. The method chosen for displaying the different entities in games was also an important consideration. SVG stands for "Scalable Vector Graphics." It is an XML markup language for describing two-dimensional vector graphics" [9]. Google maps use SVG. Canvas is "the <canvas> element. It is an HTML element which can be used to draw graphics via scripting (usually JavaScript). For example, it can be used to draw graphs, make photo compositions, create animations, or even do real-time video processing"[1]. Both SVG and Canvas have their own pros and cons. A plethora of information is available on the Internet about both methods. After scraping through various tests available in

the Internet community, the canvas method was determined to be better at drawing multiple entities in real time than SVG.

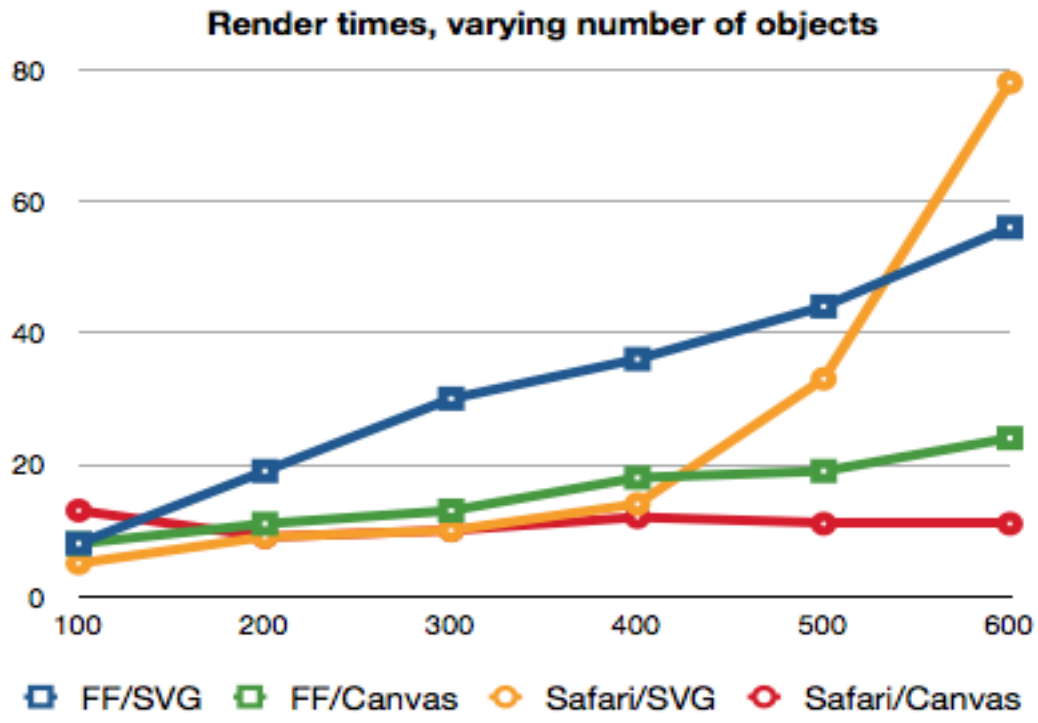

Figure 1: Canvas, SVG Comparison based on number of objects [2]

One of the important factors that influenced this decision was cross browser support, since it was one of the major goals of this project. Canvas proved more robust in this sense. SVG demonstrated to still be in the development phase for Webkit and Gecko browsers. SVG has its own benefits and it is more appropriate for drawing in larger areas. Canvas has different methods for drawing text, images that proved to be very useful. This will be discussed in the design of MobiGameJS.

HTML5 and JavaScript's data persistence abilities also factored into the decision, since any game would require data storage facility. Game players would want to store their scores, view statistics, and compare both with others. The Local Storage API of HTML5 is designed for such purposes, as well as its session storage advancements. It

is also robust, easy to use, and just requires key value pairs to store data. However, it also has its own limitations and they will be discussed in the experiments section.

2.3 Data URI

Data URI is an important factor in this MobiGameJS project. It unites the most important features in this project: saving the game on the device and displaying like a native application.

Data URL is the URL that includes small data items and it behaves in such a way that the data is included externally. Some applications also need to embed media types into URL [13].

```
data:[<mediatype>][;base64],<data>

The <mediatype> is an Internet media type specification (with
optional parameters.) The appearance of ";base64" means that the data
is encoded as base64. Without ";base64", the data (as a sequence of
octets) is represented using ASCII encoding for octets inside the
range of safe URL characters and using the standard %xx hex encoding
of URLs for octets outside that range.  If <mediatype> is omitted, it
defaults to text/plain;charset=US-ASCII.  As a shorthand,
"text/plain" can be omitted but the charset parameter supplied.
```

Figure 2: Data URI Syntax [10]

Data URIs have also been used in test games to store the images and they make the code self-sustainable and compact without any external server links. Data URI consumes little bandwidth and inline data frees up a download connection for other content. But, the data URI needs to be altered every single time there is a change in the data. In the case of this project, this is not a major concern because we needed to create images or data URI of the game only once. It takes the same amount of time to load a game saved as data URI as a game saved to a device as a native app. Some difficulties, which were caused by data URIs, will be addressed later in this report.

2.4 Experiments

In order to test the functionality of every aspect of this project, various experiments focused on data URI and local Storage using different browsers. Additional tests analyzed how the game engine works and how it could be designed. Some test games were created using JavaScript game engine called Crafty. An ancient game of Pong was also designed to experiment with game loop. The following are the findings of the experiments conducted:

1. During these experiments, it was found that game loop is an important and integral part of the game engine. The performance of the game engine depends on this small piece of code. If the game loop is running on very low fps, the game will also run very slowly and the animations will not be smooth. On the other hand if fps is very high then the game will run smoothly but the user interaction will not be possible to handle. We needed to synch the game loop with respect to the refresh rate of the browser.

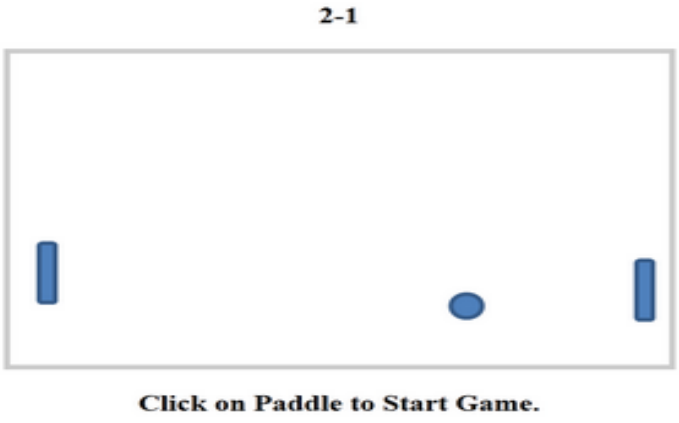

Figure 3: Game of Pong

2. The process of writing the game required an in depth analysis of which features need to be given the most importance and then how that needs to be developed. To

achieve this, we created some small games using the Crafty game engine, which is created in JavaScript.

3. Data URI is an excellent technology, but various browsers have different constraints on using local storage within data URI. There are many size constraints, which can lead to performance issues. Using local storage within data URI can also throw a security exception.

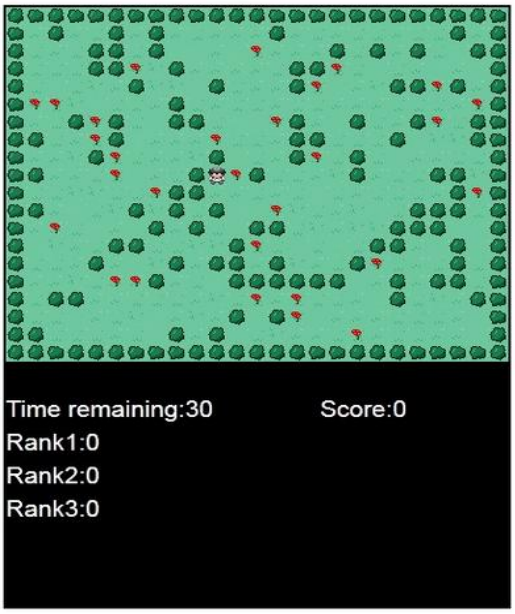

Figure 4: Flower Picker - Crafty game engine evaluation

3. JavaScript Bookmarklets were also used as a means of giving games that were created using MobiGameJS a native look. Yet, even though bookmarklets were powerful they had size constraints and their behavior was different on different browsers. Also, their support on mobile browsers was limited.

4. The results of the experiments determined the choice to support Safari on iOS and Firefox on Android. The scope was limited to 2D animations. The requirements and designing of MobiGameJS will be discussed in the following sections of this report

3. MobiGameJS

So far this report has provided an explanation of the idea behind this project, its importance along with the different technologies chosen to implement the idea. The end product of this project is called the **Mobi**le **Game J**ava**S**cript Engine. In this section, description of the software engineering created while developing the MobiGameJS is provided. Also, this section gives a description of all the tasks performed. It spans from the gathering of requirements to the testing/experiments conducted.

3.1 Requirements and Design

As mentioned in the previous sections, the game engine is a large framework. The initial thought of designing MobiGameJS consisted of creating a set of necessary requirements that were necessary to create a minimal game. A game required a mechanism of adding, updating, and drawing entities. It also became necessary to provide those entities with some kind of physics, as well as animation. It was also necessary to have some kind of functionality in order to detect a collision between two entities. Event handling was also required for user interaction. Finally, data persistence and audio were also necessary considerations. We created a list of following requirements for MobiGameJS:

1. GameLoop to run the games
2. A mechanism to add elements or entities into the game
3. Sprite functionality with animation support
4. Game Persistence
5. Event Handling specifically for touch events
6. Physics
7. Some kind of minimal Collision Detection

8. Audio
9. Native app look using data URI

After the requirements were gathered, the design of the game engine was started. The requirements were converted into classes using object-oriented design. In MobiGameJS, there are thirteen classes with hierarchical inheritance. All the classes can be seen in the figure below.

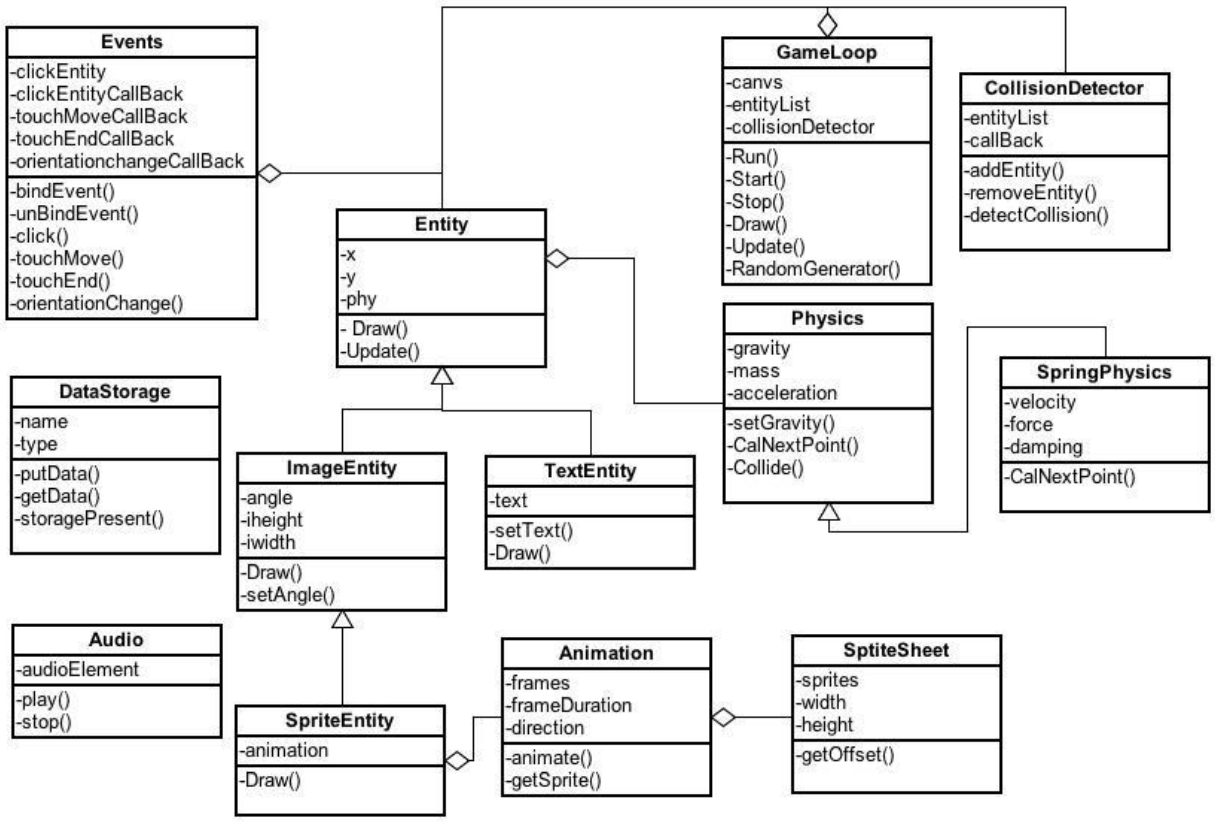

Figure 5: Class diagram of MobiGameJS

The Game Loop will be described in detail in section 3.2. Entity and the inheriting classes are described in 3.3, and section 3.4 provides the detailed working of the Collision Detector class. The Animation and SpriteSheet classes are explained in

section 3.5. The process of how MobiGameJS has incorporated physics through Physics and Spring Physics classes is described in 3.6. Event handling and audio are described in 3.7. The data persistence section will explain the Data Storage class.

3.2 Game Loop

As mentioned in the previous sections of this report, Game Loop is the heart of any game. The performance of the game, drawing, updating and user involvement depend on the game loop. As a result of this, the design of the game loop has a lot of importance in any game.

The game loop was designed using the window.setInterval function of DOM during the earlier stages of the project in Pong game experiments. It was used to call a specified function repeatedly after a certain time interval. In this project, the GameLoop() was called every 10 milliseconds. The position of the call was updated and the pedal was moved appropriately in the game loop function. The updating and the drawing were carried out at the same time. That method proved to have a serious flaw. The draw operation for every 10 milliseconds is equal to 100 frames per second and normally a refresh rate is around 60fps. That implied the burning of a number of CPU cycles, which resulted in a slow movement of the game at the same time.

In order to overcome this flaw, two particular functions, “window.webkitRequest AnimationFrame” or its counterpart “window.mozRequest AnimationFrame,” were used since they were available for both the Mozilla as well as the Webkit browsers [5]. Both the browsers called us when it was required to update or draw frame. That removed the problem of taking care of syncing. At this point, they are not completely in synch with the screen refresh rate, but still prove to be smoother than setInterval. In order to give fallback we provided the setInterval method, which would be used when a browser was not supporting those two methods.

The Game Loop class contained: the canvas, the entity list, the collision detector instance, methods for starting and stopping game loop, as well as methods for adding and removing entities. It also constituted the update and draw methods.

In the construction of the Game Loop class, the canvas was initialized to provide the width and height. Any new entities could be added to the game by using the addEntity method and could be removed using the removeEntity method. The game loop was initialized to run method in the start method. Therefore for each frame refresh, the run method was called and the stop methods remove the run method from frame refresh.

Run method is the game loop of MobiGameJS. It handles the drawing and handling of different entities. In order to appropriately synchronize the refreshing, a delay was added to updating and drawing. The update and draw methods were called after every 20 skips. In the update method, every entity in the list is updated accordingly and a check for the collision was performed. In the GameLoop's draw method, each added entities' draw method was called. A random generator was also added based on the lower and higher values. To programmatically change the orientation, a flipOrientationmethod was added - which in turn called every entity's flipOrientation method.

3.3 Entity

Every single element, image, and control is treated as a separate Entity in MobiGameJS. Every entity updates and draws itself. ImageEntity and TextEntity inherit the Entity class. SpriteEntity inherits the ImageEntity. Every entity has its own position on the canvas with x and y coordinates. It position also contains a pointer to physics and also a name for the entity. Each entity also maintains a pointer callback to call, every time the Game Loop calls an entity to update. When a call for flip orientation occurs, the x, y coordinates are exchanged and the physic pointer is asked to perform that task.

As the name suggests, TextEntity is used to display text on canvas. One can set text and get text using the getter and setter methods. The draw method for set text sets the font and color for the specific requirements and then uses the fillText method of canvas to draw the text on screen. A developer can create a text entity in the following ways:

```
var score_text = new TextEntity("Score : 0", ind_width-120 , 10 , "score");
test.AddEntity(score_text);
```

Figure 6: Adding TextEntity to the game

The ImageEntity class inherits all the functions of the entity class. It adds height, width and angle variables, which will be used while drawing images on the screen. It also overrides the Draw method. The angle is set in degrees, which is firstly converted into radians before drawing an image. That angle was needed in order to rotate our image. The same image could be reused to do some animations. The coordinate system was then set to the x and y co-ordinates of the entity and then the drawImage method of canvas was used.

```
var TO_RADIANS = Math.PI/180;
var context = document.getElementById("viewport").getContext("2d");
// save the current co-ordinate system
// before we screw with it
context.save();

// move to the middle of where we want to draw our image
context.translate(this.x+(this.iwidth/2), this.y+(this.iheight/2));

// rotate around that point, converting our
// angle from degrees to radians
context.rotate(this.angle * TO_RADIANS);
try
{
    context.drawImage(this.image, -(this.iwidth/2), -(this.iheight/2),this.iwidth,this.iheight);
}
catch(err)
{
    console.log(this.name + " " + this.x +" " + this.y + " " + this.iwidth + " " + this.iheight);
}
// and restore the co-ords to how they were when we began
context.restore();
```

Figure 7: ImageEntity Draw Method Algorithm

SpriteEntity is used specifically for displaying the sprite entities. It contains the object for the Animation class and the current frame to display. The overloaded method of drawImage is used where the width, height and scaling are specified. The sprite entity is created. Next, it is followed by the animation and the animation is then added to that entity. It is then added for collision detection and finally to the game.

```
var f1= new SpriteEntity(src_sprite,i * 16,j * 16,16,16,"flower")
f1.AddAnimation(flower);
colli.addEntity(f1,test_colli);
test.AddEntity(f1);
```

Figure 8: Adding SpriteEntity in the game

3.4 Collision Detection

Collision Detection was one of the important pieces of MobiGameJS code. There were lot of entities present in the game and they were frequently updating, changing their position. They also collided with each other sometimes due to there own properties and some due to the user's interaction. We need to detect this collision, as well as act accordingly.

Circular, rectangular, and polygon collision detections provided ways to predict a possible collision. In circular collision detection, we calculated the distance between the centers of two entities and checked to see if they are that close to each other. In MobiGameJS collision detection was performed by using rectangular detection. It's very simple to implement and very fast for detecting simple collisions.

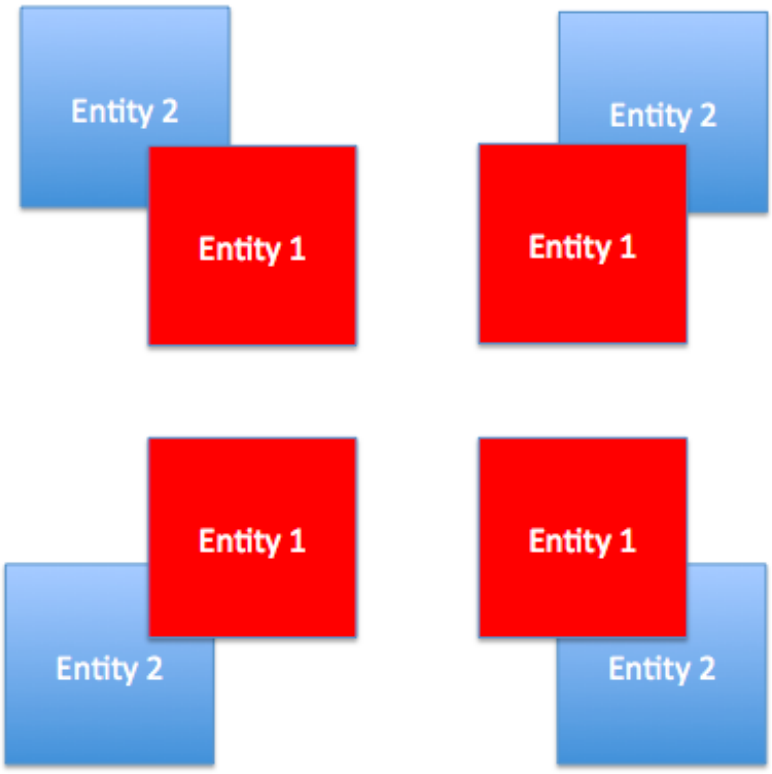

Figure 9: Rectangular Collision Detection

We just needed to check the rectangular edges with each other to see if they are colliding with each other. In MobiGameJS, there is a CollisionDetector class. We can registered which entities we want to check for the collisions. We also had a detectCollision method, which first checked to see if the entities were registered for detection. After that, we checked for the collision itself.

```
// Get the perfect positions of both the entities
var right1 = entity.x + entity.iwidth;
var right2 = CollisionDetector.prototype.entityList[i].x
    + CollisionDetector.prototype.entityList[i].iwidth;

var bottom1 = entity.y + entity.iheight;
var bottom2 = CollisionDetector.prototype.entityList[i].y
    + CollisionDetector.prototype.entityList[i].iheight;

var left1 = entity.x;
var left2 = CollisionDetector.prototype.entityList[i].x

var top1 = entity.y;
var top2 = CollisionDetector.prototype.entityList[i].y
// Check collision
if ( !((bottom1 < top2) ||(top1 > bottom2) ||
    (left1 > right2) || (right1 < left2) ))
{

    CollisionDetector.prototype.callBack[i](entity,
        CollisionDetector.prototype.entityList[i]);
}
```

Figure 10: Collision Detection Algorithm

3.5 Animation

One of the main features MobiGameJS implements is sprite animation. Sprite Animation is a widely used animation technique. It consists of a sprite sheet and animation using that. Sprite sheet is a type of an image, which consists of different images. They can form a complete set of actions. We can show a walking man as a sprite sheet example. It contains four different types of stages of walking man. The animation is achieved by displaying successive images after some time intervals.

Figure 11: Sprite Sheet Example of walking man

In MobiGameJS each sprite entity is associated with sprite animation while animation is formed by a sprite sheet. In the SpriteSheet class we took the width, height of sprite, as well as the sprites with their names and starting coordinates. The method we used to return the sprites frame is called getOffset. It basically checked the entire length of sheet and if the name matches the argument of the function, it returns the x, y coordinates and the height and width.

```
for(var i = 0; i < this._sprites.length; i++)
{
    var sprite = this.sprites[i];

    if(sprite.name == spriteName) {
        return {
            x: (sprite.x * this.width),
            y: (sprite.y * this.height),
            width: this.swidth,
            height: this.sheight
        };
    }
}
```

Figure 12: Calculating correct sprite offset

The animation class had different sprites to animate and the duration of each animation as input. The Animation class has a method called animate. It basically checks on how much time is passed in current animation. If it is more than the specified time, we moved to the next sprite in the animation.

```
animate: function() {
    //Reduce time passed from the duration to show a frame
    if ( ((new Date).getTime() - this._frameDuration) >
        (this._frames[this._frameIndex].time*1000) )
    {
        this._frameDuration= (new Date).getTime();
        //Change to next frame, or the first if ran out of frames
         if(this._dir == 0)
        {
            this._frameIndex++;

            if(this._frameIndex == this._frames.length) {
                this._frameIndex = 0;
            }
        }
        else
        {
            this._frameIndex--;

            if(this._frameIndex <= 0) {
                this._frameIndex = this._frames.length-1;
            }
        }
    }
},
```

Figure 13: Sprite Animation Algorithm

3.6 Physics

Every game has the entities relating to real word objects. The developer has to think about how these objects will work in the real world and how he can give those characteristics to the game objects. Objects have properties such as mass, acceleration. There are different forces acting upon them, likely gravity. Also, when a collision happens we need to see how it is handled, because there are many different forces associated with the collision.

MobiGameJS supports physics through Physics and SpringPhysics classes. In MobiGameJS we have provided support for gravity in each axis of the device. Basically, one can specify an axis where the object is generated and the object will fall on the opposite axis. There is a method called CalNextPoint, which provides the next point of that motion. We calculated that point using the distance between the current point and the final destination, as well as by taking into consideration the acceleration of the object.

As we saw there are different types of collisions, such as elastic and inelastic collisions. MobiGameJS uses elastic collisions. The Rudy Rucker's book suggests that we should abide by the laws of momentum, conservation and energy. The equation taken from the book is as follows:

$$newVa = [(1 - massratio) * Va + 2 * massratio * Vb] / (1 + massratio)$$
$$newVb = [2 * Va + (massratio - 1) * Vb] / (1 + massratio)$$

wheremassratio = mass2/mass1 Va and Vb = velocity before collision

Figure 14: Elastic Collision Formula [15]

MobiGameJS also supports spring forces. Once can add spring physics to any entity and it will act as it is connected to a spring. We used Hooke's Law to achieve this. Basically Hooke's Law states that the "the extension of a spring is in direct proportion with the load applied to it.". The equation for Hooke's law is as follows:

$$F = -kx$$

where

x is the displacement of the spring's end from its equilibrium position (a distance, in SI units: metres);
F is the restoring force exerted by the spring on that end (in SI units: N or kg·m/s^2); and
k is a constant called the *rate* or *spring constant* (in SI units: N/m or kg/s^2).

Figure 15: Hooke's Formula [8]

SpringPhysics overrides the CalNextPoint method of the physics class. It calculates the current location of the object based on the Hooke's Law. Until we are touching the object, we can pool that object. When we release the object it will act like a spring and it will converge based on the damping we have provided while applyingSpringPhysics.

3.7 Event Handling and Audio

Games are all about user interactions! Players should be able to interact with your games as they want and we needed to handle them properly at the correct time. JavaScript and document object models currently provide lot of support to the mobile device platforms iOS and Android. We were able to listen to the events using window.addEventListener. MobiGameJS has the ability to listen to the different events and developers can bind to those events. They can specify callback methods to call when a specific event happens. MobiGameJS also provides developers an ease of connecting entities with events. They don't have to check to see if the event that is happening is associated with their entity or not. The game engine itself using the current location of the entities does it.

```
// Get the correct location of the touch
var posx = 0;
var posy = 0;
if (e.pageX || e.pageY)     {
    posx = e.pageX;
    posy = e.pageY;
}
else if (e.clientX || e.clientY)    {
    posx = e.clientX + document.body.scrollLeft
        + document.documentElement.scrollLeft;
    posy = e.clientY + document.body.scrollTop
        + document.documentElement.scrollTop;
}
for (var i=0; i < Events.prototype.clickEntity.length; i++) {
        if(posx >= Events.prototype.clickEntity[i].x && posx <=
            (Events.prototype.clickEntity[i].x + Events.prototype.clickEntity[i].iwidth) )
        {
            // Check if the event is asociated with the current entity
            if(posy >= Events.prototype.clickEntity[i].y && posy <=
                (Events.prototype.clickEntity[i].y + Events.prototype.clickEntity[i].iheight) )
            Events.prototype.clickCallBack[i](e,Events.prototype.clickEntity[i]);
        }
}
```

Figure 16: Connecting event and entity Algorithm

The above algorithm first gets the correct position of the touch, then it checks if any entity that is bonded with touch event is near to the touch generated. If it is, then MobiGameJS will call the corresponding callback.

MobiGameJS also helps developers to detect orientation changes. Different browsers handle orientation change events differently. While Safari or webkit based browsers support the window.orientation object of DOM., Firefox on Android still does not respond to those events. It uses a more modern api window.DeviceOrientationEvent. But this api generates events for slight modifications in an angle of device. We don't want that, we just want to know the portrait and landscape modes. An alternate method is to check for window resize event, as in portrait mode height is greater than width and vice versa in landscape

```
if (window.DeviceOrientationEvent) {
    // Listen for the deviceorientation event and handle DeviceOrientationEvent object
    window.addEventListener('resize', Events.prototype.orientationchangeE, false);
} else if (window.OrientationEvent) {
    // Listen for the MozOrientation event and handle OrientationData object
    window.addEventListener('orientationchange', Events.prototype.orientationchangeE, false);
}
```

Figure 17: Orientation Handling

Audio:

HTML5 has a very simple to use audio element where you can specify which file you want to play. There are different functions such as play, pause,and stop for this element. MobiGameJS takes the responsibility of creating and adding this element to the document model.

3.8 Data Persistence

Scoring is the integral part of gaming and the players always want to track their previous scores. They want to keep tabs on a high score or how they perform. Developers would also love to save the state of the game, if the game has mechanism for exiting in between. If there are different levels in the game data storage performs an important role in game play. There are currently multiple options available to achieve this using SQLDB, IndexDB, Local Storage or local file system.

MobiGameJS currently provides data storage facilities by using the local storage api of HTML5 [11]. Browsers manage local storage and they store data with respect to the domains of the URL. It has very simple apis for getting and putting the data "getItem and putItem" into local storage. We can also check to see if the key is present in local storage by using "key in localStorage". It's very straightforward to code and very lightweight to use; and that's why we used it in MobiGameJS.

MobiGameJS has the class called DataStorage, which takes name and type as the input for its initialization. We wanted to add type for adding different types of storage later on. Name is given to every database and it is checked if it's already present in the local storage. It just acts like the connection URL that we use for connecting to various databases. It also has methods, such as putData and getData for inserting and getting data.

Though local storage is very easy to use, it has its own complications with data URI as we saw in the earlier experiments. On Android, we could use data URI and local storage without much complication. But on any webkit based browsers we would get the security exceptions and they would not allow us to use this or any other data storage apis. We could only access data of our own domain. The data uri has its domain set to null and therefore we could not use it to access local storage. After much pounding on this we found about "Same origin policy" of these storage apis and

security restrictions on it. There are a lot of discussions about how to bypass this policy and access cross-domain data. We tried different methods and finally we were able to do it using window.postMessage method.

Window.postMessage [6] is basically used for sending messages to other windows as name suggests. It requires a target domain to which we want to send the message. In MobiGameJS we used an html page “lsapi.html” which just contained methods for storing data. We give it as source to hidden iFrame in the game. By using the data storage class we could send the messages to this iFrame and use its domain to store the data. In lsapi its listening to the event “message” by using “window. addEventListener(‘message’,call,false)”. The problem using this method is that iFrame loads asynchronously while the game is loading itself. Therefore, we did not know when the local storage or iFrame would be available to us. To resolve this issue data storage waited for the signal from iFrame to start the conversation like a handshake protocol. This method was used only in the case of iOS, because on Android we can access local storage directly on Firefox. The Window.navigator object provide us all the details about platform we are currently in.

3.9 Delivery

As we saw in the abstract as well as introduction and other parts of the report, MobiGameJS is not just about creating games. It is also about how you deliver them to the users. The user should feel the game created using MobiGameJS is just like a native application created for that specific platform. One should be able to start it from the home screen. iOS and Android have some specific useful functionalities for our purpose.

First of all - we created a website, or one could say an appstore where users can download games and developers can download the game engine and upload their created games. The games will be in the form of data URIs, so that they can be self-sustainable. There is a functionality in both these platforms in order to pin the website to the home screen. We use that to give the user a feeling that he is downloading a game and created shortcut, or even say it just installed an application to the user's device. The user can then just click the shortcut and game will open in the designated browser. The website has the functionality to play a demo in the appstore.

On iOS it's pretty simple to add the game to the home screen, while on Android it little bit difficult. To normalize the process we thought of showing the message to the user. There are lot of different methods that we could have done using popups or showing notifications. While searching for the answer, we came across a web page cubiq.org [12]. Here they have created a small bubble to display a message to show the user how to add the web app to the home screen. We used that widget to display the message on bottom of the screen to download the game.

Figure 18: Cubiq.org Add to Home Screen bubble

Even if the game playing is for fun, mobile gaming is itself a big industry and lot of people hope to make money from it. Monetizing the games created using MobiGameJS is a big process and and it is out of the scope of this project. However, there are some ideas to achieve this. We will see them in the Delivery section.

4. Testing

Testing is an integral part of any software development process and there are not any exceptions in its importance in the game development process. There are a lot of functionalities that are provided by the MobiGameJS game engine and some substantial test cases needed to be created in order to test them. The best approach to test the MobiGameJS game engine is to create new games using the engine. Therefore, three different games were built to test different functionalities. A detailed explanation of the testing and results can be seen in the following sections.

4.1 Flower Picker

Flower picker was the same game, which was mentioned in the background section of this report. This game was previously built using the Crafty game engine and was a desktop-based browser game using keyboard controls. The same game was replicated using MobiGameJS. The different functionalities tested in this game were Sprite Animation, Data Storage, Audio, Entity classes and Event Handling.

Game Play: There is a man walking in the garden with walls of trees andgrass in the background. The man cannot climb or cut the trees and needs to walk around them. He needs to pick as many flowers as he can till the timer running in the background stops. When he picks up a flower, the flower will disappear from garden and picking sound is played. After the elapsed time if the score is greater than the high score it will be stored in local storage.

Figure 19: Sprite Sheet Used for Flower Picker

The sprite sheet used was the same as that of the earlier experiment. It contained the sprite for a walking man, flowers, grass and bushes. An arrow image is provided to give the user control over the man. There were text entities for the latest score and high score.

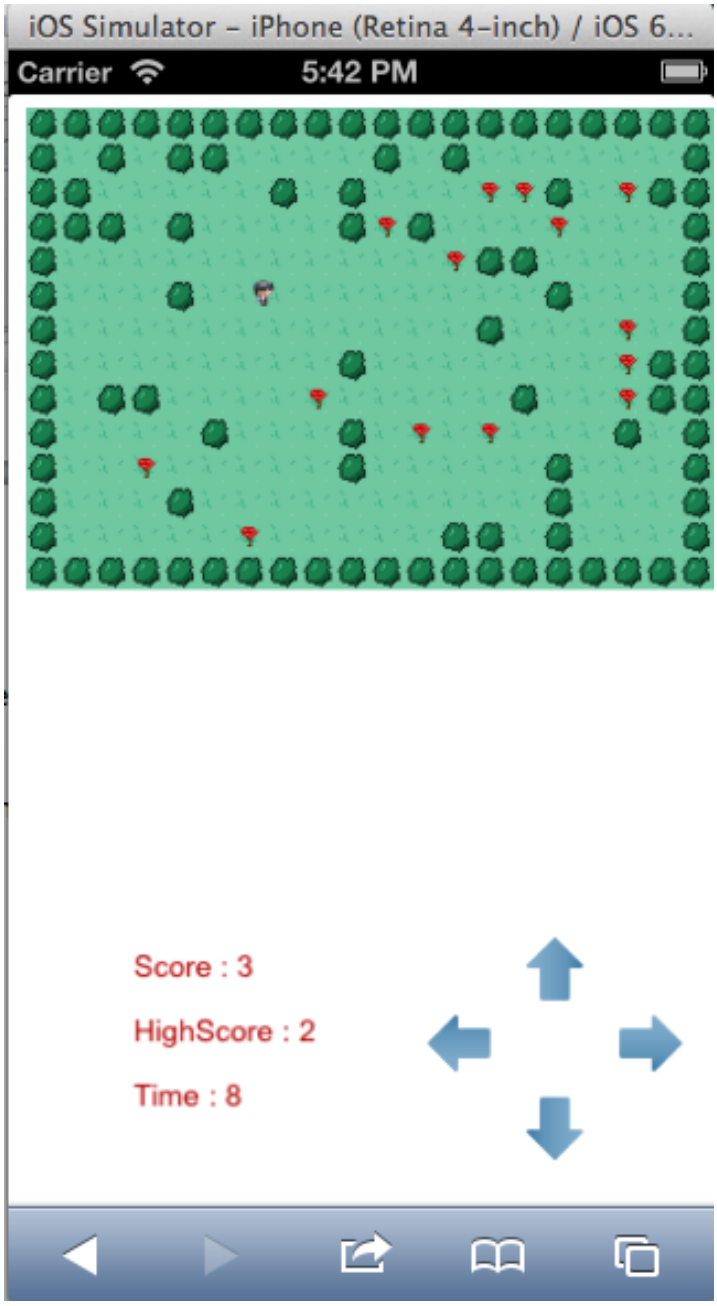

Figure 20: Flower Picker End Product

4.2 Asteroid

Asteroid is one of the earliest games known. It involves shooting down asteroids and alien ship using the fighting plane. This game was created to test the performance of the game loop. The physics functionality, collision detection, orientation events, and image entity class were also tested.

Game Play: The game has a fighting plane, which can be rotated 360 degrees. Asteroids will be flowing around from all the four directions. There are two direction buttons and clicking them would rotate the flight and also fire the bullets. There are also the three text entities for the current score, high score and the health of plane. The asteroid disappears and the score is incremented when the bullet hits the asteroid,. There is a small animation of explosions and the condition of the plane deteriorates when the asteroid hits the plane. The game finishes and the high score is updated when the state of plane becomes zero.

Figure 21: Asteroid in Landscape

The asteroid games contain a lot of moving entities such as: different asteroids, background images, arrows, fighting plane, and bullets. In order to update and draw these entities at the same time, a robust game loop is required. MobiGameJS did not display issues when the asteroids are increased up to 100. When the count was increased to 1000, it slightly slowed down the game engine. Hence, it can be concluded that the performance of the game loop of MobiGameJS performs as expected. This game also shows that the collision detection works fairly well as there are many collisions happening at the same time. The bullets collide with the asteroids; asteroids collide with each other and also asteroids collide with the plane. It handles them well in spite of the elastic collisions happening with the asteroids. A gravity element can also be seen, while colliding and moving in the opposite directions. The proper handling of orientation can be seen in the figure below.

Figure 22: Asteroid in portrait with local storage testing

4.3 Spring Physics

The final game created acts as a demonstration of what can be achieved in terms of physics using MobiGameJS. The spring physics class created was also tested.

Game Play: The balls with different masses are attached to a bar using spring ropes can be seen in the figure below. We can touch the balls to pull them. Mass of the ball will decide the amount of force that is required to pull the balls. After the release of the balls, they would reciprocate until the force applied on them is nullified.

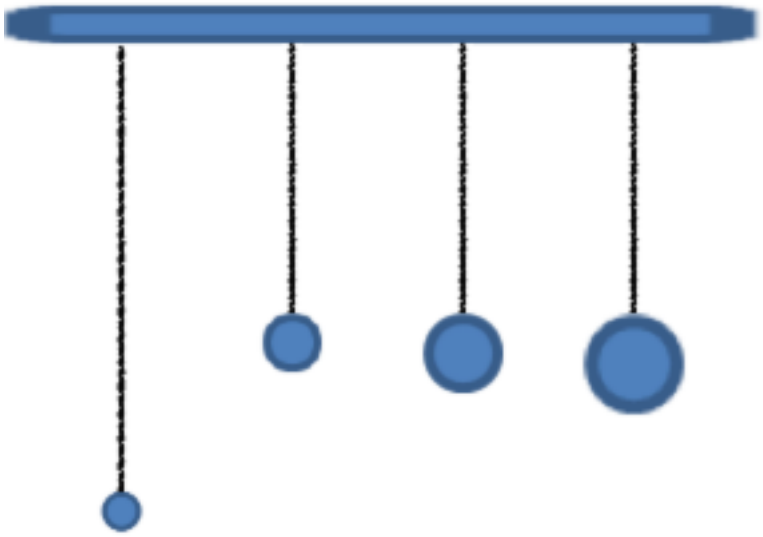

Figure 23: Spring Physics Demonstration

4.4 Delivery

The delivery of the game is an important feature of the MobiGameJS game engine. The game had to be delivered so that the user could gain the feeling of playing a native game. An html website was designed in order to deliver the games created using MobiGameJS. It is a website that is designed for mobile devices using template http://mobifreaks.com/. The first page of the website is the index page where the MobiGameJS game engine can be downloaded. It also provides information about the game engine and its usage. Using the menu, one can navigate to the other pages of the website.

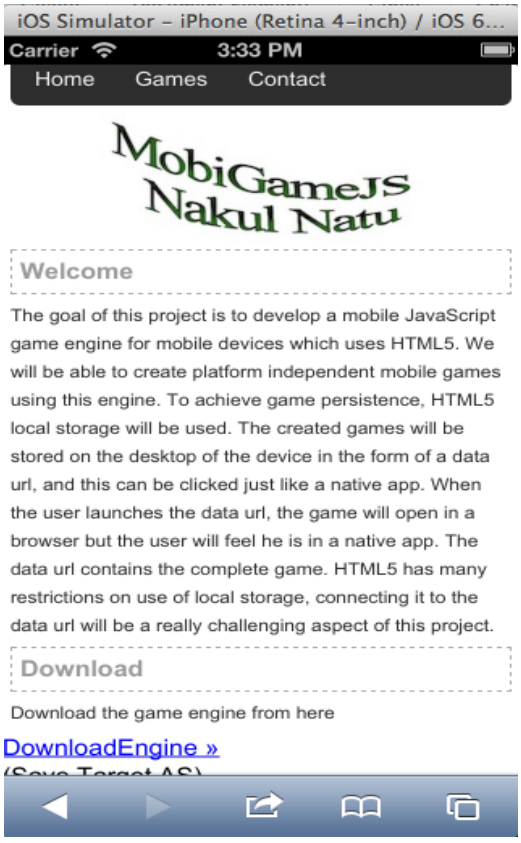

Figure 24: Delivery of MobiGameJS

A contact page is also provided where the developers can contact us about the game engine. The most interesting page on the website is the Games page and this

page contains the information about the game that the developers created. It works as an app store where the users can download, buy games and upload their games. It contains a small description about the game, a snapshot, the demo link and an option for buying. When the user navigates to this page, a bubble is displayed to the user stating the directions to download the game. The user can then navigate to the game and add it to home screen. A shortcut will then be created at the home screen and user can launch the game whenever he wants to.

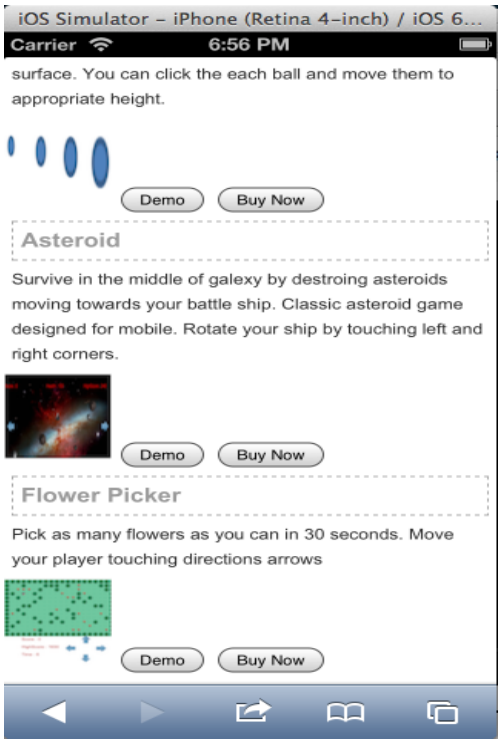

Figure 25: Delivery of games created using MobiGameJS

Every device has an id irrespective of whether it is iOS or Android, which can be extracted. If the ID is not available, it can be created programmatically. When a user launches a game, a list of devices for which a game is downloaded can be maintained and kept track of. If the device is used to download the game, the game can be started, or else a shutdown signal can be sent to stop the game loop. This can be achieved using the same iFrame technique that is used for data persistence.

5. Conclusion

Mobile gaming is on the rise. Although a lot of mobile users are not hard-core gamers, a significant lot of them try the games on smartphones in their leisure time. Therefore, a lot of developers are looking for an opportunity to get into this market of mobiles. There are some roadblocks in order to start this development, which include the platform to choose, the different types of hardware, and most importantly, the learning curve. The idea of this project was to specifically address these issues.

A game engine called MobiGameJS has been created with the help of web development technologies such as HTML5, Canvas, local storage and hardware related JavaScript events and APIs. As shown in the test games, MobiGameJS is simpler to use. It also implements a lot of the requirements necessary for a game engine such as: sprite animation, physics, event handling, media, collision detection etc. The games developed in MobiGameJS can be deployed on any platform using the data URI.

This project is a good start towards the portability of mobile games and in turn the mobile applications. Future improvements to the system could include improvements in the user's experience of installing and launching games, more robust data persistence technique. Appropriate monetization processes can also be deployed, so that the developers can sell their work comfortably.

6. References

[1] Benjamin Smedberg, Nickolay. (Nov 1, 2012) Canvas- Mozilla Developer Network. Retrieved from https://developer.mozilla.org/en-US/docs/HTML/Canvas

[2] Boris Smus. (Jan 19, 2009). PERFORMANCE OF CANVAS VERSUS SVG. Retrieved from http://smus.com/canvas-vs-svg-performance/

[3] Chris Stead (July 15, 2009). The 10 Best Game Engines of This Generation. Retrieved from http://www.ign.com/articles/2009/07/15/the-10-best-game-engines-of-this-generation

[4] David Rahimi (Aug 9, 2012). Samsung Galaxy S1 Vs. S2 Vs. S3, How The Galaxy S Has Evolved Over Time. Retrieved from http://www.phonebuff.com/2012/08/samsung-galaxy-s1-vs-s2-vs-s3-galaxy-evolved-time/

[5] Eric Shepherd, Jared Wein. (Oct 14, 2012). Animation Frames - Mozilla Developer Network. Retrived from https://developer.mozilla.org/en-US/docs/DOM/window.requestAnimationFrame

[6] Eric Shepherd, Joel Overton. (Sept 7, 2012). Post Message - Mozilla Developer Network. Retrieved from https://developer.mozilla.org/en-US/docs/DOM/window.postMessage

[7] Frederic Lardinois (Sept 4, 2012). Apple's Share Of U.S. Smartphone Market Now Over 33%, RIM Drops To Under 10%. Retrieved from http://techcrunch.com/2012/09/04/comscore-apples-share-of-u-s-smartphone-market-now-over-33-rim-drops-to-under-10/

[8] Hooke's Law. (2012). Retrieved from http://en.wikipedia.org/wiki/Hooke's_law

[9] Jeff Schiller, Nickolay. (Oct 14 , 2012) SVG-Mozilla Developer Network. Retrieved from https://developer.mozilla.org/en-US/docs/SVG

[10] L. Masinter (1998). The "data" URL scheme. Retrieved from http://www.ietf.org/rfc/rfc2397.txt

[11] Mark Pilgrim. (2012). THE PAST, PRESENT & FUTURE OF LOCAL STORAGE FOR WEB APPLICATIONS. Retrieved from http://diveintohtml5.info/storage.html

[12] Matteo Spinelli. (Oct 10, 2011). Add to Home Screen. Retrieved from http://cubiq.org/add-to-home-screen

[13] Nakul Natu. (2011). JavaScript Game Engine for Mobile using HTML5. Retrieved from http://www.cs.sjsu.edu/faculty/pollett/masters/Semesters/Fall11/nakul/CS297Report.pdf

[14] Nielsen (March 29, 2012). Smartphones Account for Half of all Mobile Phones, Dominate New Phone Purchases in the US. Retrieved from http://blog.nielsen.com/nielsenwire/online_mobile/smartphones-account-for-half-of-all-mobile-phones-dominate-new-phone-purchases-in-the-us/

[15] Rudy Rucker. (2002) Software Engineering and Computer Games. Addison-Wesley