How to Make A Simple HTML5 Game With Enchant.js

This is a post by Tutorial Team member Guts Rodsavas, an iOS development trainer at Software Park Thailand and game developer at Coffee Dog Games. Are you curious about developing cross-platform mobile games that work in a web browser? Well, as you probably know, Apple doesn't allow Flash to run on iOS devices, and Adobe […]

This is a post by Tutorial Team member Guts Rodsavas, an iOS development trainer at Software Park Thailand and game developer at Coffee Dog Games.

Learn how to make a simple browser-based mobile game with HTML5!

Are you curious about developing cross-platform mobile games that work in a web browser?

Well, as you probably know, Apple doesn't allow Flash to run on iOS devices, and Adobe has pulled the plug on Flash for mobile. This makes HTML5 your only solution – but actually a pretty good one!

The thing is, while there are many HTML5 game libraries available, only a few of them support mobile browsers. In this tutorial, you'll try out one HTML5 game library that works great on the iPhone, Android, and the desktop – enchant.js!

You'll make a game called called Penguin Dive, where players will take control of a penguin that's swimming in the ocean, and dodge incoming ice boulders. The longer the penguin stays safely in the ocean, the higher the score. Once an ice boulder hits the penguin, it's game over!

This tutorial assumes you are a complete beginner to enchant.js, but you must have some basic knowledge of JavaScript and HTML. Knowing how to set up your own local server also helps for testing purposes, but isn't required.

If you're ready, then let's get this party started!

Introducing enchant.js

enchant.js is a fairly new HTML5+JavaScript game framework. The version as of writing is 0.51, which is DOM-based. The future version will move to HTML5 Canvas-based.

In case you're wondering, DOM stands for Document Object Model. In this type of JavaScript coding, the code accesses the elements on a page via a structured hierarchy. Usually, the elements on the page are represented like a tree, where each HTML element is a branch or a leaf, and has a name or ID by which it can be accessed directly (or by way of a parent or grandparent).

The Canvas-based approach will rely on the new HTML5 <canvas> tag, which allows the code to define a canvas (or drawing area) and then draw directly on that surface, much as you would use CoreGraphics drawing functions in Objective-C.

Each approach has its pros and cons, but this tutorial won't delve into those, since enchant.js will take care of the details internally. But it's good to know the pros and cons of using enchant.js itself!

Pros of Using enchant.js:

- It has a 2D scene graph system with a simple Flash-like API.
- It's cross-platform. Your game will work on iOS, Android, and desktop browsers.
- It's lightweight.
- There are many plugins available to help you make specific types of games, such as a visual novel or an RPG.
- It's free and open-source!

Cons of Using enchant.js:

- It lacks device orientation-related features.
- It doesn't yet support multi-touch.
- Due to the framework's Japanese origin, there is a lack of resources in English. (But this tutorial is helping to change that!)

Its cross-platform support is what makes enchant.js stand out from other HTML5 game frameworks. While it lacks some features, the current version is functional enough to make a complete working game.

Setting Up the Environment

Here is what you'll need in order to write a game with enchant.js:

1. *enchant.js*: Download the framework from their github page. You can either clone the git repository, or use the ZIP icon at the top of the page to download a ZIP archive of the current source.
2. *Game assets*: Download these assets for the tutorial, and unzip them on your hard drive.
3. *Text Editor*: Since developing using enchant.js means working with JavaScript, decide what IDE/text editor to use. My favorite is Sublime Text 2. You can also use something like jsdo.it, which allows you to develop in your browser. :]
4. *Web Server*: This is optional. However, it's better to test your HTML5 game on a web server rather than from your hard disk, since viewing a web page on your hard disk directly might disable some features. It also lets you test your game from your mobile device! There are many ways to set up a web server, but one of the easiest is to go with XAMMP.

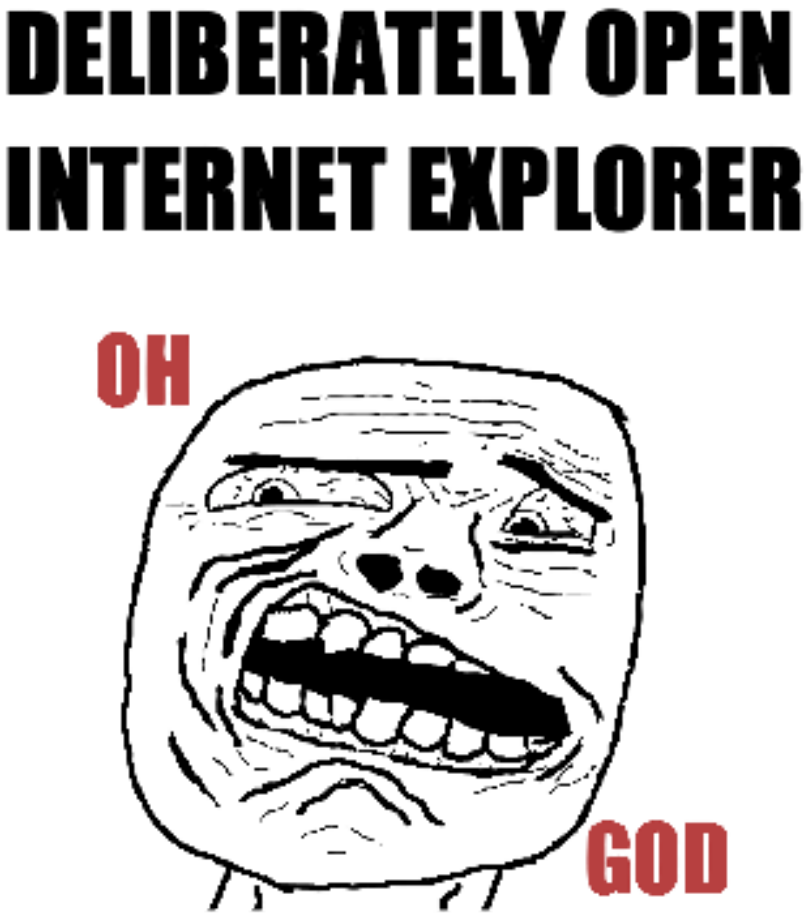
Apart from the above, you need a web browser with a JavaScript console so that you can view the JavaScript output as you code.

Note: Note that although you are making a HTML5 game that will work on mobile browsers, usually you'll want to test the game on a normal browser while you're developing the game. This allows for faster test/run cycles, and lets you easily see the Javascript output.

In this tutorial, you will be using a normal web browser until later on in the tutorial, where I'll show you how to get it working nicely on a mobile browser.

Your choices for a development browser are:

1. *Google Chrome*: You can enable the console by clicking on the *Wrench Icon*, then *Tools*, and selecting the *JavaScript console*.
2. *Safari*: You have to enable the Developer menu first. Go to *Safari\Preferences* and check *Show Develop menu in menu bar* in the *Advanced Tab*. You can then open the console through *Develop\Show Error Console*.
3. *Firefox*: You can either install the Firebug extension, or enable the console via *Extras\Error Console* or *Tools\Web Developer\Error Console* (depending on your version). Note Firebug has some extra cool features beyond the normal console.
4. *Opera*: Enable through *Menu\Page\Developer Tools\Opera Dragonfly*.
5. *Internet Explorer*: Don't use this! :D Unless you really have no other choice – then grit your teeth and press F12 to get to the developer tools.



Preparing Your Project Structure

enchant.js doesn't enforce any directory structure for projects. You can structure the project folders any way you like. So in this tutorial, you're going to structure things in a way that makes sense to me. Hopefully, it will make sense to you, too. ;]

Go to the directory that you want to be your workspace and create a new folder named *penguindive*.

Next, create subfolders within your penguindive folder so that the folder structure is as shown:

```
penguindive/
penguindive/js
penguindive/js/lib
penguindive/res
```

The *res* and *js* folders will be where you store your JavaScript files and your game assets, respectively. The *lib* folder will be where you store third-party JavaScript files/libraries.

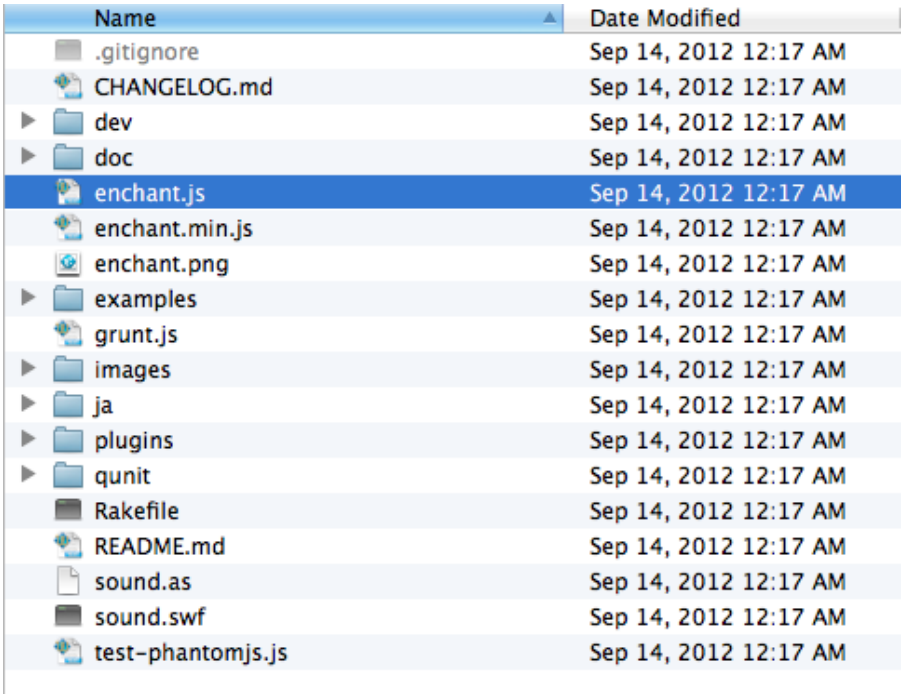
The next step is to put all the game assets you downloaded earlier inside the *res* folder. Simply extract the ZIP file and copy the files. Once you do that, you should have the following files in the *res* folder:

```
BG_Over.png
BG.png
```

```
bgm.mp3
Eat.mp3
fishSheet.png
Hit.mp3
Ice.png
penguinGameOver.png
penguinSheet.png
```

You will not be using all of these files in this tutorial. However, feel free to use them to extend the game on your own, as a challenge. There are some suggestions for how to do this at the end of the tutorial. :]

Next, extract the enchant.js archive you downloaded from GitHub. Copy the *enchant.js* file and put it in your *js\lib* folder.



Note: You'll notice that there are two versions of enchant.js in the archive – *enchant.js* and *enchant.min.js*. The second file is the minified version, where extra spaces and carriage returns have been removed to make the smallest downloadable version possible for the code.

Since that version also replaces variable names and methods with short (usually incomprehensible) alternatives, it doesn't lead to very human-readable code. So it's best to use the standard version of the code during development and debugging (*enchant.js*) and to use the minified version in production (*enchant.min.js*).

You're now ready to start the actual coding!

Why Hello There, Ocean!

Let's start with something simple: saying hi to the world where your game will take place. :]

Launch your favorite text editor and start by creating the *index.html* file, which as I'm sure you're aware, will be the first file loaded when the game folder is accessed via a browser:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Penguin Dive</title>
        <script src="js/lib/enchant.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body>
    </body>
</html>
```

Save the file (name it *index.html*) in the *penguindive* folder.

The HTML code is fairly simple. You've included two JavaScript files – the enchant.js library itself, and a main.js, which will be the file where you write code for your game.

Next, create *main.js* and add the following code to it (as the index.html file indicated, the main.js file goes in the *js* subfolder when you save it):

```
// 1 - Start enchant.js
enchant();


// 2 - On document load
window.onload = function() {
        // 3 - Starting point
        var game = new Game(320, 440);
        // 4 - Preload resources
        game.preload('res/BG.png');
        // 5 - Game settings
        game.fps = 30;
        game.scale = 1;
        game.onload = function() {
                // 6 - Once Game finishes loading
                console.log("Hi, Ocean!");
        }
        // 7 - Start
        game.start();
};
```

The code above is all you need to get an enchant.js game working. Let's go over it step-by-step:

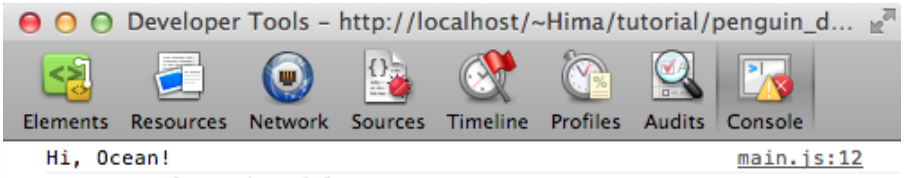
1. Export all the library classes globally. This allows you to use classes from enchant.js without having to type the namespace every time.
2. Create a function that will be invoked once the HTML document is done loading. You will initialize your game inside this function.

3. Create an instance of the Game class, which is the main application class of an enchant.js game. The constructor for the Game class takes two arguments: the width and height of the game's screen in pixels.
4. Preload the background image for the game. When you have resources that take a lot of time to load (like big images or background soundtracks), it's a good idea to load them before you actually need to use them.
5. Configure the game settings. Notice the line setting the game frame rate to 30 fps (frames per second). This does not mean that the game will always run at 30 fps. It's very likely that the fps will drop on mobile platforms. Think of this as a maximum fps that the game will try its best to achieve.
6. The Game object's onload event will be invoked once the game finishes loading everything in the preload queue. This is the entry point of your game.
7. As the method name implies, this will start your game. Starting your game will initiate the preloading process, and invoke the onload method once the preloading is finished.

Note: The game's frame rate and scale cannot be changed after you start the game. Make sure to set them before calling game.start()!

Save your *main.js*. It's time to see if your game is running correctly on the desktop browser. Simply open *index.html*, or, if you're using a web server, navigate to the game folder via localhost.

When you open the game page, there shouldn't be anything on the screen – after all, you haven't put anything there yet! But if you check your browser JavaScript console, you should see "Hi, Ocean!" displayed.



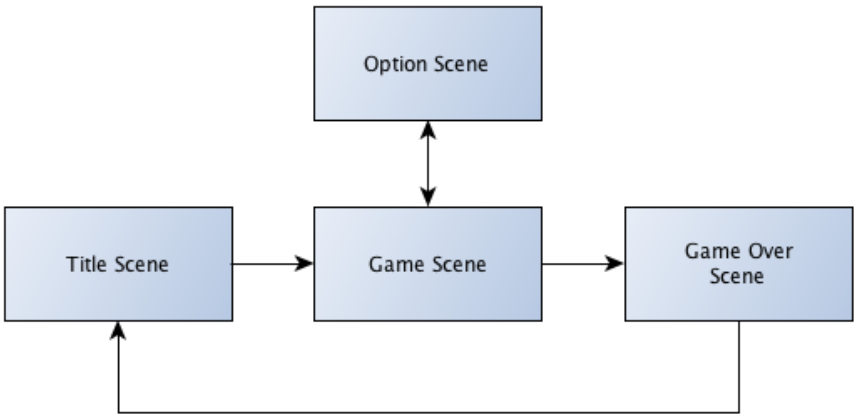
Node, Scene and Game: The Basic Concepts

When working with enchant.js, there are some basic concepts that you need to know:

Scene

enchant.js uses the concept of a scene to handle the game flow. A game can have many scenes, but only one of them can be running at a time. By switching scenes, you can change the game's state/screen.

For example, a game's flow might look like this:



According to the diagram, there are four scenes in the game. Starting from the Title Scene, players can go to the Game Scene, where the main gameplay takes place. From there, they can switch between the Game Scene and the Option Scene to adjust the game's settings.

Once the player is back in the Game Scene and the game is over, the game moves to the Game Over Scene, before returning to the Title Scene again, ready for the next game session.

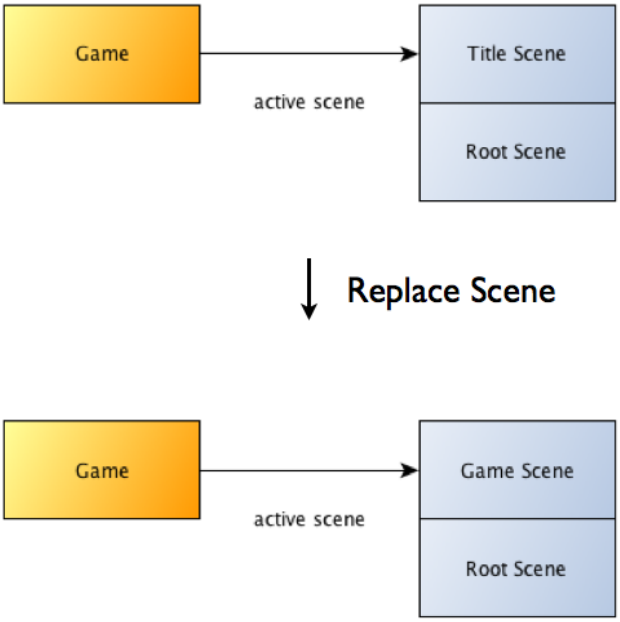
Game

As stated earlier, the Game class is the main application class. One Game object represents one single enchant.js game. The Game class also acts as a Scene Manager. It stores instances of scenes in a stack, where the top scene on the stack is the active scene.

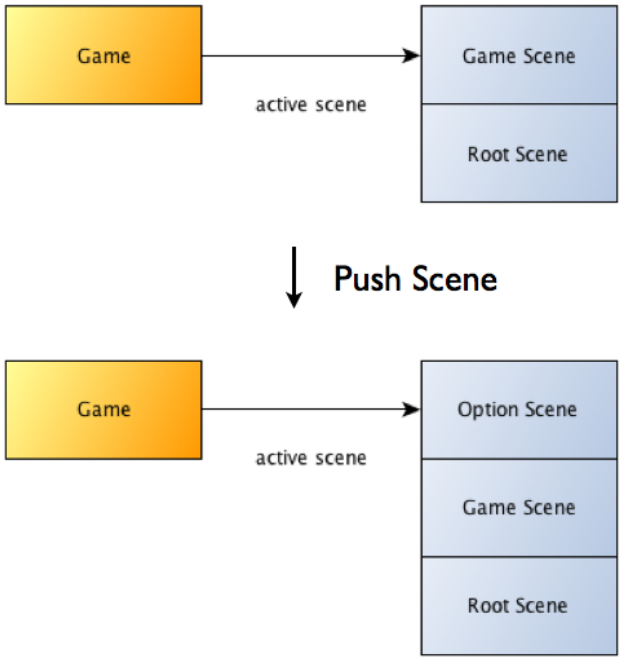
In enchant.js, a game always comes with a scene called the *Root Scene* (i.e. the first scene in the stack). By manipulating the scene stack, you can change the game's state easily.

There are three ways you can manipulate the scene stack through a Game object:

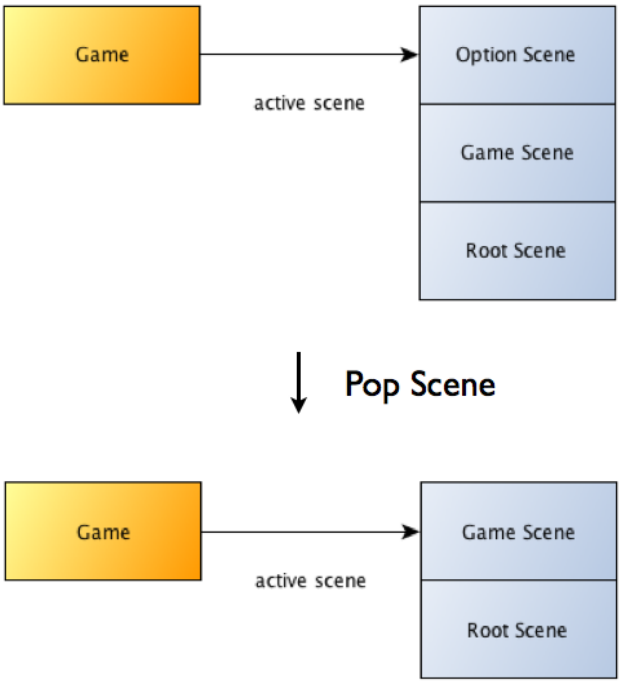
- *Replace*: This replaces the active scene with a new one. You can't go back to the old scene, unless you store a reference to it somewhere before replacing it.



- *Push*: This pushes a new scene on top of the stack, making it the active scene. The old scene is kept inactive in the stack.



- *Pop*: This removes the scene at the top of the stack, making the next scene in the stack the active scene.



Node

Scenes in enchant.js are implemented using a tree data structure. Hence, many objects in an enchant.js game are nodes. In fact, if you are familiar with Cocos2D development, the concept is fairly similar to that of a node in Cocos2D.

There are many types of nodes in enchant.js. Here are a few examples, all of which you'll use shortly:

- *Sprite*: a node for displaying static images and animation.
- *Label*: a node for displaying text.
- *Group*: a node for grouping many nodes together.

Therefore, a scene is actually a tree of nodes, where the root is a Scene node. It may have some children nodes that are sprites, labels, and so on.

Do you "node" what I mean? Not quite? Let's write some code to help you understand this better! :]

Add the following lines to *main.js* after you print "Hi, Ocean!" to the console:

```
// 1 - Variables
var scene, label, bg;
// 2 - New scene
scene = new Scene();
// 3 - Add label
label = new Label("Hi, Ocean!");
// 4 - Background
bg = new Sprite(320,440);
bg.image = game.assets['res/BG.png'];
// 5 - Add items
scene.addChild(bg);
```

```
scene.addChild(label);
// 6 - Start scene
game.pushScene(scene);
```

Let's take a closer look at each section:

- Define some variables that will be used in the following code.
- Create an empty scene. You will use this scene as your main game scene.
- Create a Label node to display the text "Hi, Ocean!" on the screen. The Label class constructor takes one argument, which is the text you want to display.
- Create a Sprite node for the background image. The constructor takes two arguments: the width and the height of the image you want to display. Once the sprite is created, you assign the image you want this sprite to display.
  You can access the image data you have loaded into the game through the game's assets dictionary. This dictionary will map the path to the loaded resource. Since you told the game to preload a file at path *res/BG.png*, this path will be used as a key that maps to the ocean background.

- Add your new nodes to the scene. The *addChild* method means that the node you add will become one of the scene's child nodes.
  Note: The order in which you add nodes to a scene is important. In enchant.js, the framework will draw a newly-added node on top of those that were added previously.

  If you switch the order in the code above, you won't see the label even though it is actually in the scene. This is because the background would be on top of the label.

- Your scene is ready. It's time to make this scene your active scene. To do this, you simply push the scene onto the game's scene stack.

Save *main.js* and reload your browser page. You should see something similar to the following:

Object-Oriented Style With enchant.js

enchant.js already comes with its own object-oriented framework. Without this, you would have to create a scene and add everything within the game's onload event. While this is simple, the OOP framework that enchant.js provides makes it easier for you to extend your code, and your code will be cleaner in the long run!

OOP in enchant.js uses the same implementation as Prototype.js, in case you're familiar with Prototype. So, in order to create a new class, you only need to do the following:

```
var ClassA = Class.create({
        // Constructor
        initialize: function(){
        },

        // Foo method
        foo: function(){
```

```
        }
});
```

initialize is a constructor function. It will be invoked when you create a new instance of the class.

To add more methods, simply add a comma at the end of the last method before starting the new one.

If you want to create ClassB that is a subclass of ClassA, you would implement ClassB like this:

```
var ClassB = Class.create( ClassA, {
        // Override Foo method
        foo: function(){
        }
});
```

Equipped with this knowledge, let's refactor the previous code using the enchant.js OOP system.

Go to the end of *window.onload* in *main.js* (after game.start();) and add these lines:

```
// SceneGame
var SceneGame = Class.create(Scene, {
     // The main gameplay scene.
    initialize: function() {
        var game, label, bg;

        // 1 - Call superclass constructor
        Scene.apply(this);
        // 2 - Access to the game singleton instance
        game = Game.instance;
        // 3 - Create child nodes
        label = new Label("Hi, Ocean!");
        bg = new Sprite(320,440);
        bg.image = game.assets['res/BG.png'];
        // 4 - Add child nodes
        this.addChild(bg);
        this.addChild(label);
    }
});
```

The above code creates SceneGame as a subclass of Scene. Let's go over the code:

1. Invoke the constructor of Scene, which is the superclass of your SceneGame, to do any initialization it needs.
2. Often you'll find yourself in need of accessing the game object, especially when you want to access the assets dictionary. Fortunately, the game instance is a singleton, meaning there is only one single instance of Game and it can be accessed from anywhere. This line of

code will assign the singleton Game instance to the *game* variable for later use.
3. Create child nodes for the scene, just like before.
4. Add the child nodes as before, but this time use *this* instead of *scene* because the *this* variable refers to the current instance of SceneGame. Hence the line is still doing the same thing – that is, adding nodes to your scene node.

With this, you have finally created your first enchant.js class!

It's time to use SceneGame. Replace the code inside *game.onload* with the following:

```
// Once Game finishes loading
console.log("Hi, Ocean!");
var scene = new SceneGame();
game.pushScene(scene);
```

The game's onload is now much cleaner. You instantiate a SceneGame object, and add it to the scene stack.

Save your changes and reload the browser. It should be exactly the same as before, except now your code is ready to be extended into a full game!

Who Let the Penguin Out!

It's time to introduce the ice-water-loving hero of your game – the penguin!

Add *res/penguinSheet.png* to the preload resource list in *window.onload* in *main.js*:

```
game.preload('res/BG.png',
             'res/penguinSheet.png');
```

Go to the SceneGame constructor and add a new variable named *penguin* to the first line:

```
var game, label, bg, penguin;
```

Then add the following code right after you assign an image to the bg node:

```
// Penguin
penguin = new Sprite(30,43);
penguin.image = game.assets['res/penguinSheet.png'];
penguin.x = game.width/2 - penguin.width/2;
penguin.y = 280;
```

The first two lines are similar to what you did with the background. You create a new Sprite node sized 30×43 pixels, and assign the image file to the penguin sprite.

The next two lines set the penguin's position. Notice how easy it is to access the screen and the penguin's width. You can access and modify node heights similarly.

Finally, don't forget to add the penguin to the scene. Otherwise you won't see it on the screen!

Add the following code right after the line adding the bg node to the scene. Remember that the order in which you add nodes is important!

```
this.addChild(penguin);
```

You want the penguin to be on top of the background, but under the label. Therefore, you must add him after the background, but before the label.

Now save your changes and reload the page in your browser to see the current state of your game:

You should now see the penguin on the screen. Your first visual evidence of progress! :]

A Closer Look at Sprites

While the penguin looks like it's just another Sprite node, it's slightly different when compared to the background Sprite.

Take a look at how you created the penguin sprite:

```
penguin = new Sprite(30,43);
```

Remember what this means? This line created a sprite with a size of 30×43.

However, if you look at the file *penguinSheet.png*, you'll see that the image's size is actually 60×43. Why, that is certainly not 30×43!

So what does this mean? When you instantiate the sprite with a size of 30×43, but assign an image sized 60×43 to it, this is how enchant.js sees the image:

Sprite node in enchant.js treats the image you assign as a sprite sheet. It will use the size you've given as its frame size, and assign an index number to each frame. The index will start from 0, counting from left to right and top to bottom.

You can change the animation frame by setting the *frame* property of a Sprite. This is what you are going to do next!

Basic Animation

Since there are going to be many things going on with your penguin, it's better to put him into a class of his own. And that's how your upstart, boulder-challenging penguin prefers it.

Add the code for the Penguin class right below SceneGame in *main.js*, as follows:

```
// Penguin
var Penguin = Class.create(Sprite, {
    // The player character.
    initialize: function() {
        // 1 - Call superclass constructor
        Sprite.apply(this,[30, 43]);
        this.image = Game.instance.assets['res/penguinSheet.png'];
        // 2 - Animate
        this.animationDuration = 0;
        this.addEventListener(Event.ENTER_FRAME, this.updateAnimation);
    }
});
```

Penguin is a subclass of Sprite. As you've probably noticed, the line that calls the superclass constructor is a bit different from what you used when you subclassed Scene. This is because a Sprite constructor takes two arguments. To send arguments to your superclass constructor, you pass them as an array in the second argument.

The two lines in section #2 animate your penguin. In the first line, you declare an instance variable, which will be used as a timer for your animation. The second

line introduces you to the event system in enchant.js. If you are familiar with ActionScript or Corona development, you'll find this easy to understand.

Simply put, objects in enchant.js can fire events that other objects might be interested in. If an object is interested in a specific event, you can tell that object to listen for that event and specify functions that should be invoked once that specific event occurs.

Hence, in the above line, you tell the penguin to listen for the ENTER_FRAME event, and call a method named *updateAnimation* every time this event occurs.

ENTER_FRAME is an event that is fired every frame, similar to the update function in a game loop.

Your Penguin class doesn't have an *updateAnimation* method yet. That's right, you're going to add one!

Keeping the previous instructions for writing classes in mind, add a comma after the closing curly brace for *initialize* in the Penguin class, and then add the following code to the Penguin class:

```
updateAnimation: function (evt) {
    this.animationDuration += evt.elapsed * 0.001;
    if (this.animationDuration >= 0.25) {
        this.frame = (this.frame + 1) % 2;
        this.animationDuration -= 0.25;
    }
}
```

Any method that is invoked as part of an event listener can receive one argument, which is the event information. The actual information will differ, depending on the type of event.

For the ENTER_FRAME event, one item of information you can access is the amount of time that has passed since the last frame. This information can be accessed through the *elapsed* property.

The elapsed property stores time in milliseconds. You can convert it to seconds by multiplying by 0.001. You'll use the *animationDuration* variable to keep track of how much time has passed in seconds.

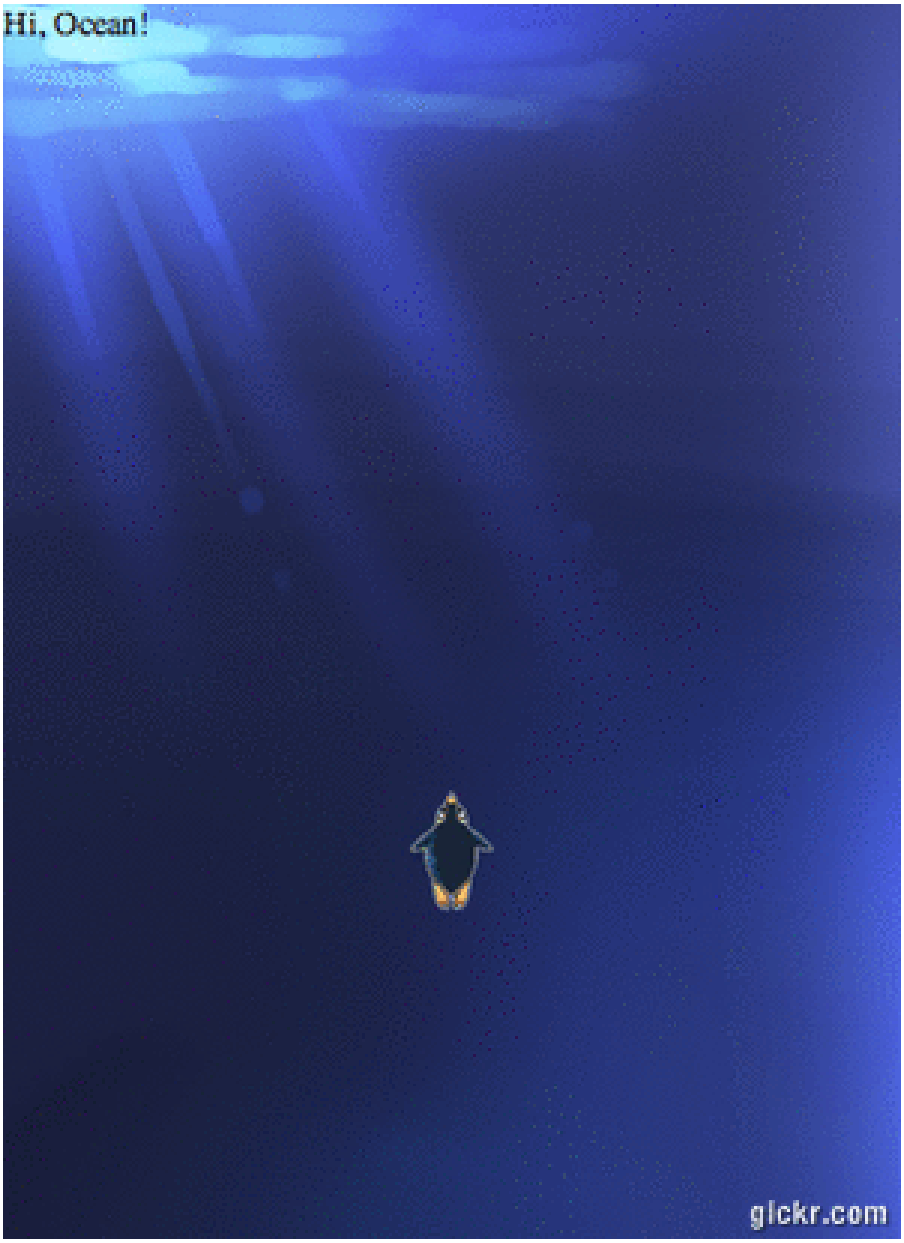
The next line checks if enough time has passed for your penguin to change his animation frame. The if block tells the penguin to flap his wings every 0.25 seconds.

It's time to use your newly created Penguin class. Go back to where you created the penguin in the SceneGame constructor and replace the code with this:

```
// Penguin
penguin = new Penguin();
penguin.x = game.width/2 - penguin.width/2;
penguin.y = 280;
this.penguin = penguin;
```

Since you've moved all the sprite-related stuff to the Penguin class, all you need to do is instantiating an instance of the Penguin class and position it.

Save the file and refresh the browser to see what you've accomplished.



Your penguin is now moving! Er, at least he's flapping his flippers. He's alive – and the water currents are beckoning. Let's get him on his way.

Touch Detection

Now that you've got the penguin flapping, it's time to get him responding to player touches. In order to do this, you will divide the screen into three vertical sections. When the player clicks or touches on any section, the penguin will quickly move to that section of the screen.

Touch/click detection in enchant.js is as simple as listening for touch events. Add the following code to the end of the SceneGame constructor:

```
// Touch listener
this.addEventListener(Event.TOUCH_START,this.handleTouchControl);
```

TOUCH_START is one of the touch events you can detect. The three available touch events are:

- *TOUCH_START*: fires when the mouse button is clicked or a finger touches the screen.
- *TOUCH_MOVE*: keeps firing as long as the player drags a mouse while pressing the button, or moves a finger that's continually touching the screen.
- *TOUCH_END*: fires once the player releases the mouse button, or lifts the finger off the screen.

Since you listen for TOUCH_START, *handleTouchControl* will be called as soon as the player touches the screen.

Add *handleTouchControl* to the SceneGame class (don't forget the comma after the previous method):

```
handleTouchControl: function (evt) {
    var laneWidth, lane;
    laneWidth = 320/3;
    lane = Math.floor(evt.x/laneWidth);
    lane = Math.max(Math.min(2,lane),0);
    this.penguin.switchToLaneNumber(lane);
}
```

You divide the screen's width by 3 to get the width of each section/lane.

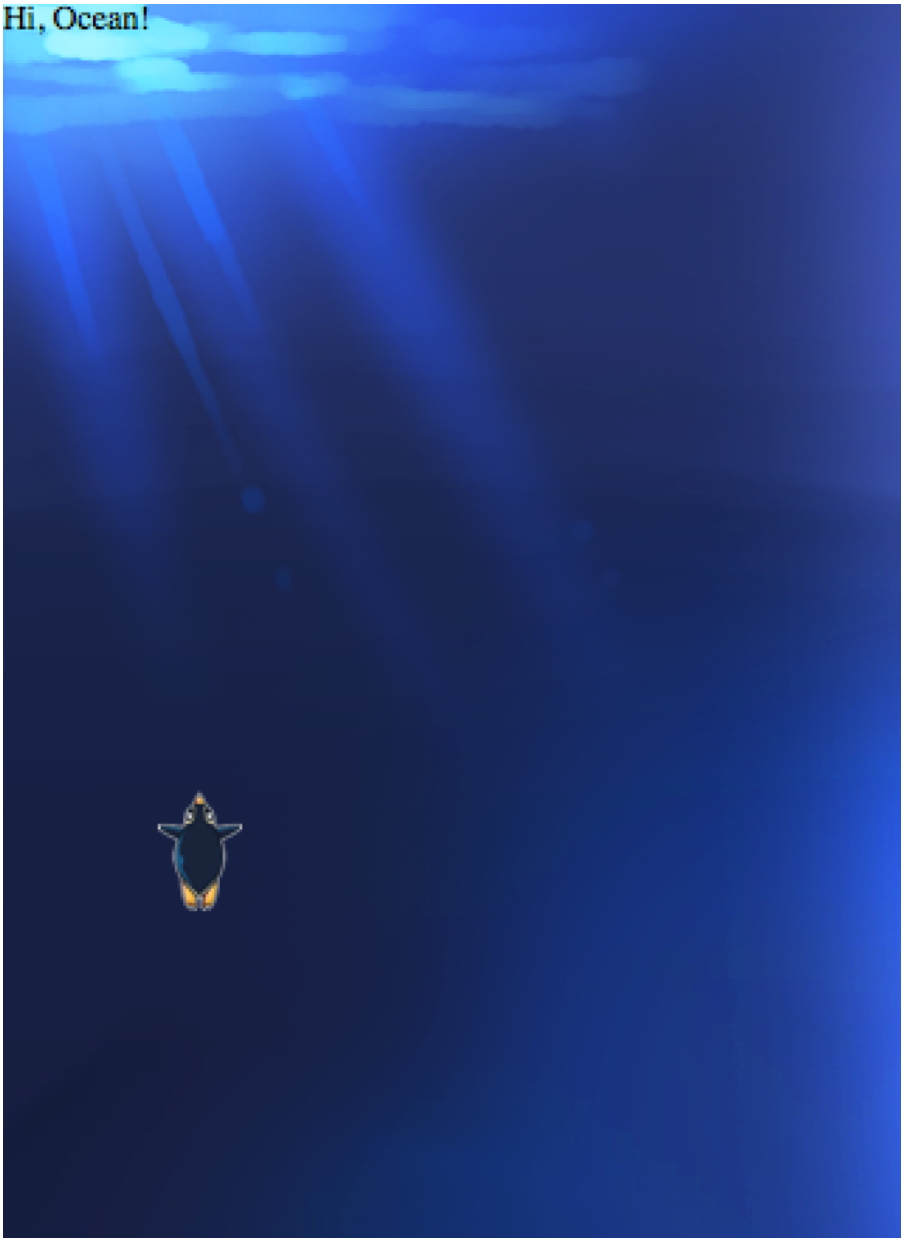
The *evt* value for a touch event contains information about the touch position, and you can access it via x and y properties. Then you use this information to find the lane number. After you've determined which lane was clicked/touched, you tell the penguin to move to that lane.

Add the *switchToLaneNumber* method to the Penguin class (note that the new method goes in *Penguin* and not *SceneGame*):

```
switchToLaneNumber: function(lane){
    var targetX = 160 - this.width/2 + (lane-1)*90;
    this.x = targetX;
}
```

There's nothing complicated here – you just calculate the x position for the penguin, given the lane number.

Save main.js and refresh the browser again. Now try clicking on the screen, and you should see the penguin move to the lane you selected.



He’s getting ready for the Polar Olympics!

Ice, Ice Baby

Your penguin can now swim about, but if this is to be a proper game, there should be some obstacles for the penguin to dodge. It’s time you added some – don’t let the penguin off easy!

Start by adding an Ice class to the end of *main.js*:

```
// Ice Boulder
var Ice = Class.create(Sprite, {
    // The obstacle that the penguin must avoid
    initialize: function(lane) {
        // Call superclass constructor
        Sprite.apply(this,[48, 49]);
        this.image  = Game.instance.assets['res/Ice.png'];
        this.rotationSpeed = 0;
        this.setLane(lane);
        this.addEventListener(Event.ENTER_FRAME, this.update);
```

```
    }
});
```

The Ice class is also a subclass of the Sprite class. The Ice constructor takes an argument indicating the lane where it will appear.

Notice that your Ice class is missing two methods. The first one is *setLane*, which you will use to set the object's position according to lane number. The second is *update*, which you call every time ENTER_FRAME occurs.

Add *setLane* to the Ice class as follows:

```
setLane: function(lane) {
        var game, distance;
        game = Game.instance;
        distance = 90;

        this.rotationSpeed = Math.random() * 100 - 50;

    this.x = game.width/2 - this.width/2 + (lane - 1) * distance;
    this.y = -this.height;
    this.rotation = Math.floor( Math.random() * 360 );
}
```

It's a straightforward method. It calculates a random rotation speed for the Ice, which you will use to animate the ice in update method. Then the Sprite's position is set, based on the lane.

One method down, one more to go! Add the *update* method as follows:

```
update: function(evt) {
        var ySpeed, game;

        game = Game.instance;
    ySpeed = 300;

        this.y += ySpeed * evt.elapsed * 0.001;
        this.rotation += this.rotationSpeed * evt.elapsed * 0.001;
    if (this.y > game.height) {
        this.parentNode.removeChild(this);
    }
}
```

Once again, this is pretty straightforward. You keep adding speed to the ice's y position so that it'll move from the top of the screen to the bottom. You also use *rotationSpeed* to keep the ice boulder rotating.

Once the ice moves beyond the bottom of the screen, it is removed from the parent node, effectively removing it from the scene.

You can access a parent node of any node through the *parentNode* property. With it, you can tell the ice's parent node to remove the ice from the tree structure once it moves beyond the bottom of the screen. Remember the node and scene

relationship? If you add the ice to the scene, the scene becomes the parent node of the ice.

You're using a resource, *res/Ice.png*, that you haven't told the game to preload. So let's go back to the beginning of main.js and add it to the preload list.

Modify the preload line to look like this:

```
game.preload('res/BG.png',
             'res/penguinSheet.png',
             'res/Ice.png');
```

And with this, your Ice class is frozen solid and ready to deploy. Why don't you test it by making an ice boulder appear every three seconds!

To do this, go to SceneGame constructor. Add the following line after the TOUCH_START event listener line:

```
// Update
this.addEventListener(Event.ENTER_FRAME, this.update);
```

Once again, you tell the scene to listen for ENTER_FRAME, and report the event to the *update* method.

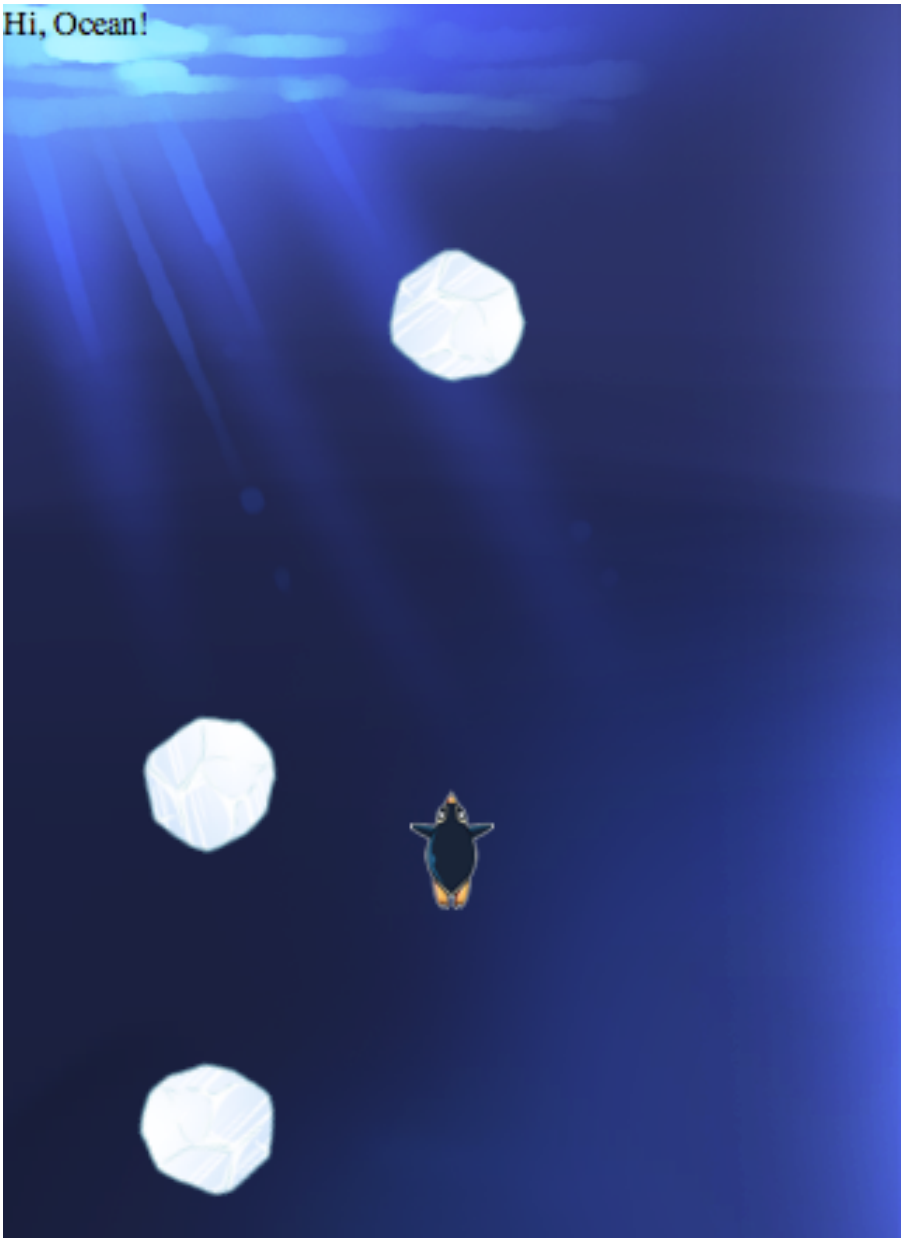
You need a timer to know when to generate an ice boulder. Add this to the end of SceneGame's constructor:

```
// Instance variables
this.generateIceTimer = 0;
```

Add the *update* method to SceneGame as follows:

```
update: function(evt) {
    // Check if it's time to create a new set of obstacles
    this.generateIceTimer += evt.elapsed * 0.001;
    if (this.generateIceTimer >= 0.5) {
        var ice;
        this.generateIceTimer -= 0.5;
        ice = new Ice(Math.floor(Math.random()*3));
        this.addChild(ice);
    }
}
```

Save your changes and run the game again. Now the ice boulders should appear at random positions every 0.5 seconds!



Collision Detection

Having run the game, you've probably noticed that…

Yep, the ice boulders go right through the penguin. You should do something about this, or your penguin will get lazy and before you know it, get eaten by a killer whale. Make him fight for his life! :]

Start by adding a new variable to the SceneGame constructor, so that the first line looks like this:

```
var game, label, bg, penguin, iceGroup;
```

Then add this code in the SceneGame constructor, right after creating the penguin:

```
// Ice group
iceGroup = new Group();
this.iceGroup = iceGroup;
```

Here you have created a Group node called iceGroup. Still remember what a Group node is?

A Group is a node that can contain other nodes, just like what the scene's doing with your sprite and label. You'll use this group to store all the ice boulders, so that you can manage them all from one place.

Next, add the new group to the scene. Go to where you add nodes to the scene in the SceneGame constructor and add the following code, immediately after where you add the bg node:

```
this.addChild(iceGroup);
```

By adding the iceGroup after the background but before the penguin, you make sure that the penguin will always be above the ice. Using Group nodes, you can create a layer system that gives you more control over the rendering order.

Go to SceneGame's *update* method. Remember where you generate an ice boulder? Make the following change:

```
//this.addChild(ice);
this.iceGroup.addChild(ice);
```

You're not adding new boulders directly to the scene anymore. Instead, newly created boulders are added to the ice group. But they'll still be rendered on the scene, because you already added the group to the scene in the constructor.

That's all fine and good, I hear you say. But what does all this have to do with boulders hitting the penguin? You're getting there. :] In fact, it's time to work on the collision detection so that you can find out when a boulder hits the penguin!

One good thing about a Group node is that you have assembled a collection of nodes you are interested in. You'll go through each boulder in the group and see if it collides with your penguin.

Still inside SceneGame's *update* method, add the following lines after the *if* block:

```
// Check collision
for (var i = this.iceGroup.childNodes.length - 1; i >= 0; i--) {
    var ice;
    ice = this.iceGroup.childNodes[i];
    if (ice.intersect(this.penguin)){
        this.iceGroup.removeChild(ice);
        break;
    }
}
```

A Group node has a *childNodes* array that keeps track of all of its children. This block of code iterates through each child and checks if it collides with the penguin.

An instance of the Sprite class has an *intersect* method that you can use to check if two sprites are intersecting. Since Ice and Penguin are both subclasses of Sprite, you can check if the two collide by using this method. If a boulder collides with the penguin, you remove that boulder from the group.

Save *main.js* and refresh your browser. Now, when a boulder collides with your penguin, it should disappear!

For now, you'll leave the penguin to imagine that he's smashing apart those boulders with his beak each time he collides with one. You'll implement the grim reality soon enough. ;]

Tally the Score

This game could use a way to keep score. So, let's change the "Hi, Ocean" label into something more useful.

Go to where you create the label in the SceneGame constructor and modify the code to look like this:

```
// Label
label = new Label('SCORE<br>0');
label.x = 9;
label.y = 32;
label.color = 'white';
label.font = '16px strong';
label.textAlign = 'center';
label._style.textShadow ="-1px 0 black, 0 1px black, 1px 0 black, 0 -1px black";
this.scoreLabel = label;
```

This time you set the text to *SCORE
0*. The
 between SCORE and 0 is the line-break tag.

The rest of the code sets the label's properties, such as its position, color, font, and text alignment. These should be self-explanatory.

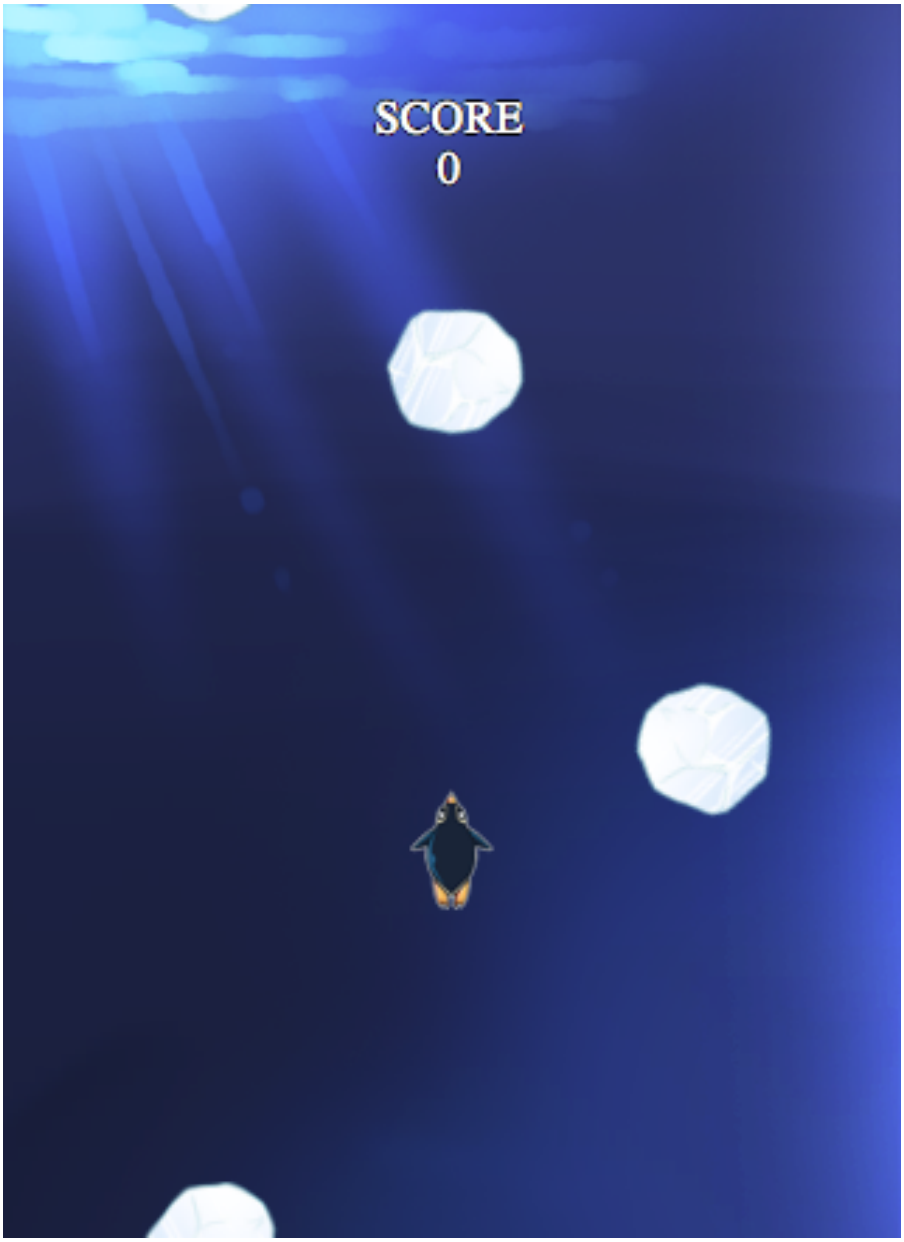
For some interesting effects, you can edit the CSS style of a label by accessing the *_style* property. For example:

```
label._style.textShadow ="-1px 0 black, 0 1px black, 1px 0 black, 0 -1px black";
```

The above line uses the text shadow property to add a black border around the text.

Finally, you keep a reference to the label in your SceneGame instance under the name *scoreLabel*.

Save and run the game again. The "Hi, Ocean" label should now be in the top center of the game screen, and should read, "SCORE 0".



Of course, currently there is no scoring mechanism! You are going to increase the score as time passes.

Add the following two variables at the end of SceneGame's constructor:

```
this.scoreTimer = 0;
this.score = 0;
```

You use *scoreTimer* as a timer to increase the game score as time passes, while the *score* variable contains the game score (obviously!).

You want the label to report the player's current score. Whenever the score is modified, the text on the label should be updated as well. Therefore, it is a better idea to wrap the score modification into a method.

Add a method called *setScore* to SceneGame:

```
setScore: function (value) {
    this.score = value;
    this.scoreLabel.text = 'SCORE<br>' + this.score;
}
```

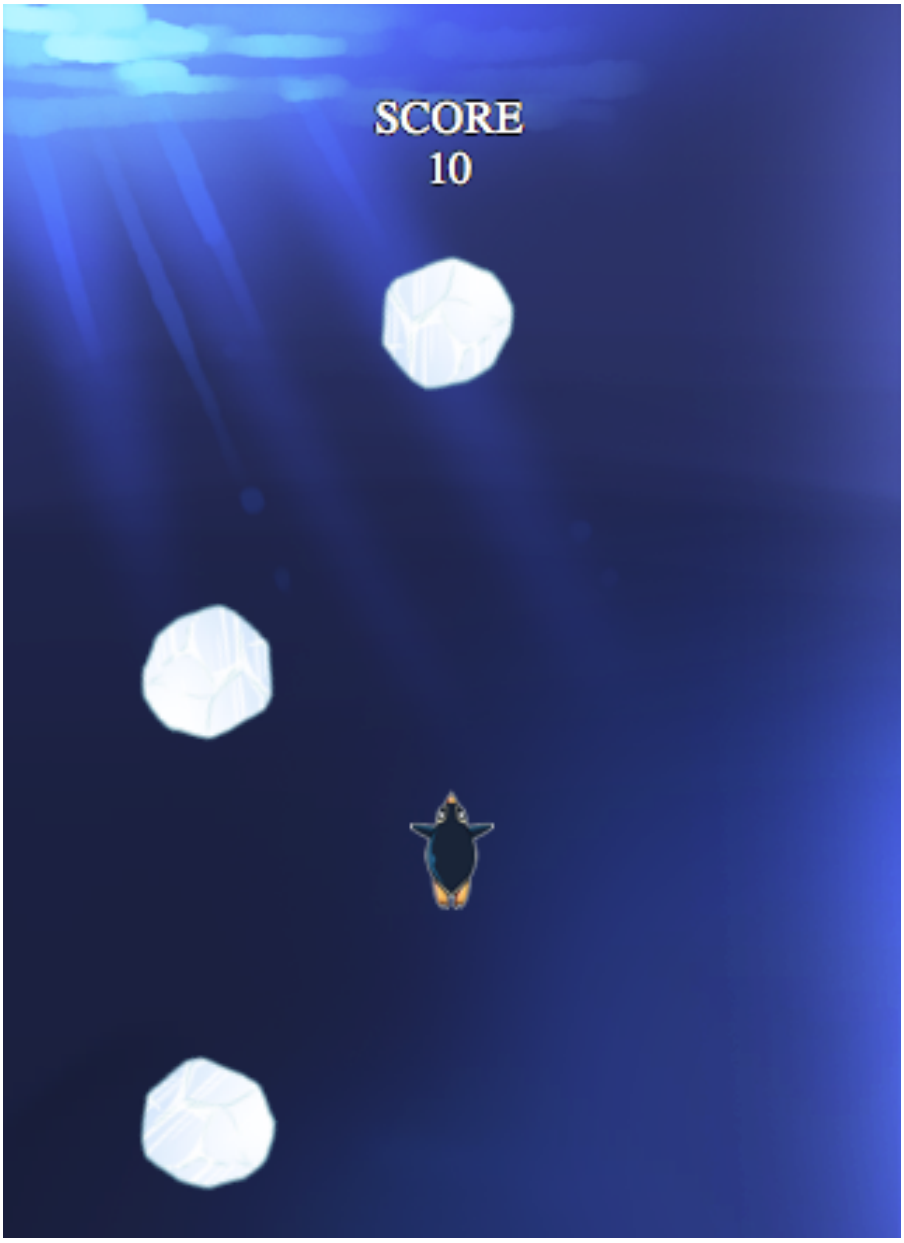
This method does exactly what you want. It changes the score, and updates the score label to reflect the current score.

Go to SceneGame's *update* method and add the following lines at the beginning:

```
// Score increase as time passes
this.scoreTimer += evt.elapsed * 0.001;
if (this.scoreTimer >= 0.5) {
    this.setScore(this.score + 1);
    this.scoreTimer -= 0.5;
}
```

You're probably familiar with this pattern by now: every 0.5 seconds, you increase the score by 1 point using the setScore method.

Save and run the game again. Now the score increases with each passing second, and you don't even have to pretend to play! Wow, could this game get more difficult or exciting? :P



Don't worry, it will! Your penguin's field day won't last much longer.

A Little Water Music

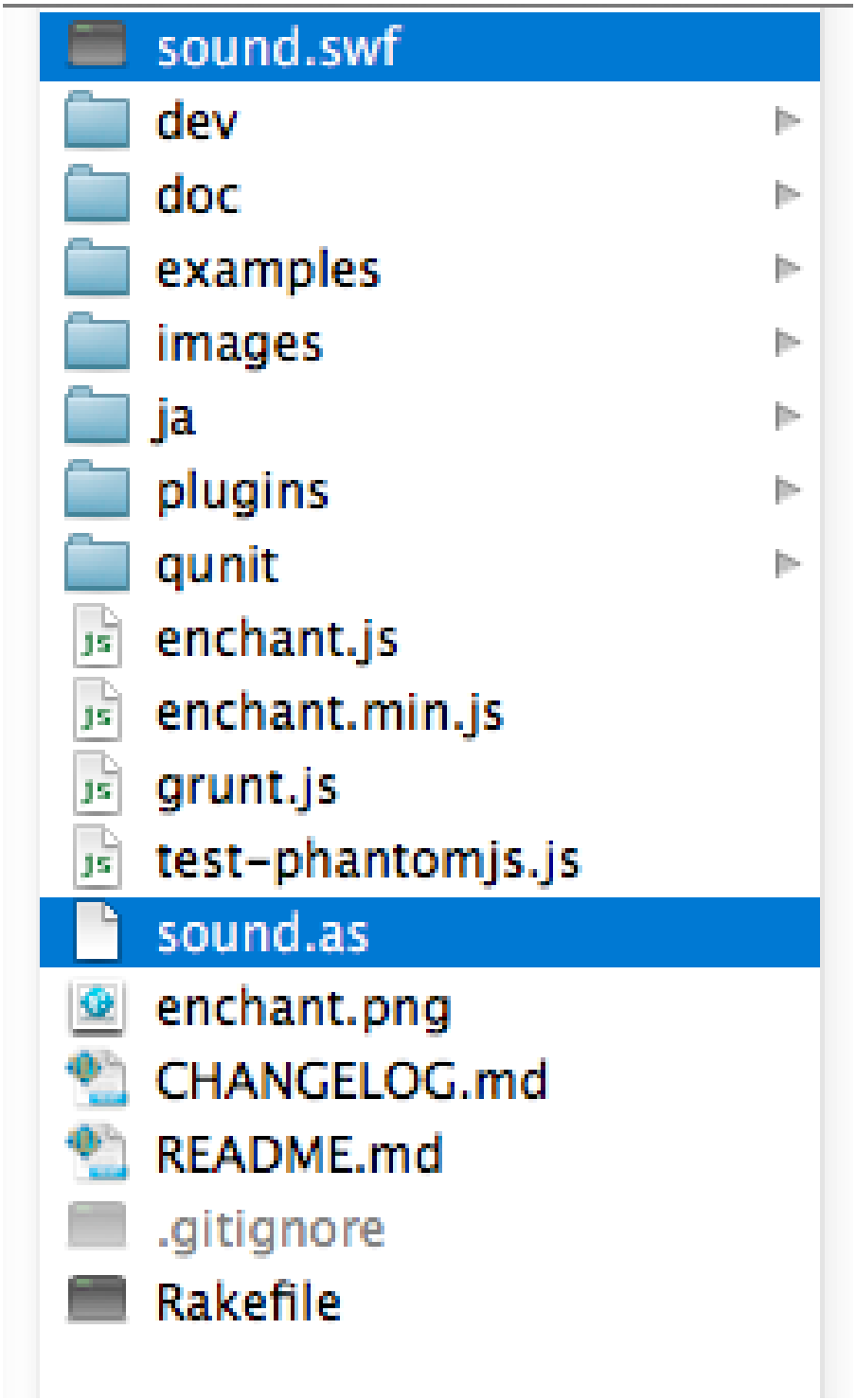
Before you configure your end game, let's pause to ask: what's a good game without some snazzy sound effects and music?

enchant.js also comes with a simple audio system. However, while the audio system works pretty well on a desktop browser, this isn't quite the case for mobile browsers.

Audio on mobile browsers is still a big problem on any OS. Currently, it is acceptable for a mobile HTML5 game to NOT have any audio at all. Hopefully, things will improve on mobile browsers soon.

The audio system in enchant.js uses the Web Audio API with Flash fallback. In order to enable Flash fallback, you have to add some helper files to your project.

Go to the enchant.js folder (the one into which you extracted the files from the GitHub project) and copy *sound.as* and *sound.swf*:



Go back to your project folder and paste the copied files into the root folder, *penguindive*. Now, if a browser doesn't support Web Audio, it will use Flash to play the sound file instead.

It's time to preload more stuff! Modify the *preload* line in *main.js* to be as follows:

```
game.preload('res/BG.png',
             'res/penguinSheet.png',
             'res/Ice.png',
             'res/Hit.mp3',
             'res/bgm.mp3');
```

Inside SceneGame's *update* method, go to the *if* block where you check for collision. Modify the if block to look like this:

```
if (ice.intersect(this.penguin)){
    var game = Game.instance;
    game.assets['res/Hit.mp3'].play();
    this.iceGroup.removeChild(ice);
    break;
}
```

Unlike images, audio files loaded by enchant.js will be kept as Sound objects inside the assets dictionary. You can access these objects directly and call the *play* method to play the audio.

Save and run the game. When the penguin hits a boulder, the hit sound effect should play, which sounds kinda like a board slamming a wall (well, at least if you're using a browser that supports it).

Note: Audio playback might or might not work correctly, depending on the browser. It worked fine for me on Safari on a Mac, but would not work correctly with the latest Firefox beta if I was loading the file directly. But if I loaded the game via a local web server, then it worked fine on Firefox as well.

So, if you get a loader bar (it usually appears very quickly without the sound pre-loading, and so you might not even notice it) and it doesn't complete, then try a different browser or, if you are loading the file directly, try loading it via a local web server. If nothing works, remove the sound pre-loading, and things should work correctly again.

Next, you'll add the background music to the game. Go to where you declare instance variables inside the SceneGame constructor. Add the following after the instance variables:

```
// Background music
this.bgm = game.assets['res/bgm.mp3']; // Add this line
// Start BGM
this.bgm.play();
```

The above code stores a background music sound object for later use, and then tells the background music to play.

You want the background music to loop indefinitely, so go to *update* in SceneGame and add the following lines to the end:

```
// Loop BGM
if (this.bgm.currentTime >= this.bgm.duration ){
    this.bgm.play();
}
```

Since there is no loop option in the enchant.js audio system, you have to implement it manually. This block of code checks to see if the audio's current time has reached the end of the audio file. If it has, then the audio is played from the start again.

Save and run the game, and you should hear the background music play – a happy and silly tune, perfect for this game!

Every Penguin Has His Day

You're almost there! It's time to make life challenging for your carefree penguin. He'll be punished for hitting the ice with death, and when he dies, you'll allow the player to restart the game.

Start by creating a SceneGameOver class. Add this code to the end of *main.js*, just as with all the other classes:

```
// SceneGameOver
var SceneGameOver = Class.create(Scene, {
    initialize: function(score) {
        var gameOverLabel, scoreLabel;
        Scene.apply(this);
        this.backgroundColor = 'black';
    },
});
```

Like SceneGame, SceneGameOver is a subclass of the Scene class. You'll be creating two labels to show on the screen. One label will say "Game Over", and the other will report the final score.

You also set the background color of the scene this time, instead of using an image. Currently the color is set to black, but you can try other colors if you'd like.

Note: Instead of color names, you can also use hex codes or RGB color values. For example:

```
        this.backgroundColor = '#000000'; // Hex Color Code version
        this.backgroundColor = 'rgb(0,0,0)'; // RGB value version
```

Add the following code right after where you set the background color in SceneGameOver:

```
// Game Over label
gameOverLabel = new Label("GAME OVER<br>Tap to Restart");
gameOverLabel.x = 8;
gameOverLabel.y = 128;
gameOverLabel.color = 'white';
gameOverLabel.font = '32px strong';
gameOverLabel.textAlign = 'center';
```

This is straightforward: you create a label and adjust its style.

Next create a label to report the final score. Add the following code right after the previous code:

```
// Score label
scoreLabel = new Label('SCORE<br>' + score);
scoreLabel.x = 9;
scoreLabel.y = 32;
scoreLabel.color = 'white';
```

```
scoreLabel.font = '16px strong';
scoreLabel.textAlign = 'center';
```

Just as you did before, this will create a score label. The score variables come from the SceneGameOver constructor. This means that when you create an instance of this scene, you have to pass a final score to it.

Your labels are ready. It's time to add them to the scene!

Add the following code immediately after the previous block:

```
// Add labels
this.addChild(gameOverLabel);
this.addChild(scoreLabel);
```

Finally, make the scene listen for the TOUCH_START event, so that players can restart the game by tapping on the screen. [TODO: Put this where?]

```
// Listen for taps
this.addEventListener(Event.TOUCH_START, this.touchToRestart);
```

SceneGameOver is missing a method for the touch handler. Fix that by adding it [TODO: Where?]:

```
touchToRestart: function(evt) {
    var game = Game.instance;
    game.replaceScene(new SceneGame());
}
```

This method does nothing but replace the current scene with SceneGame, meaning that the game will go back to SceneGame again.

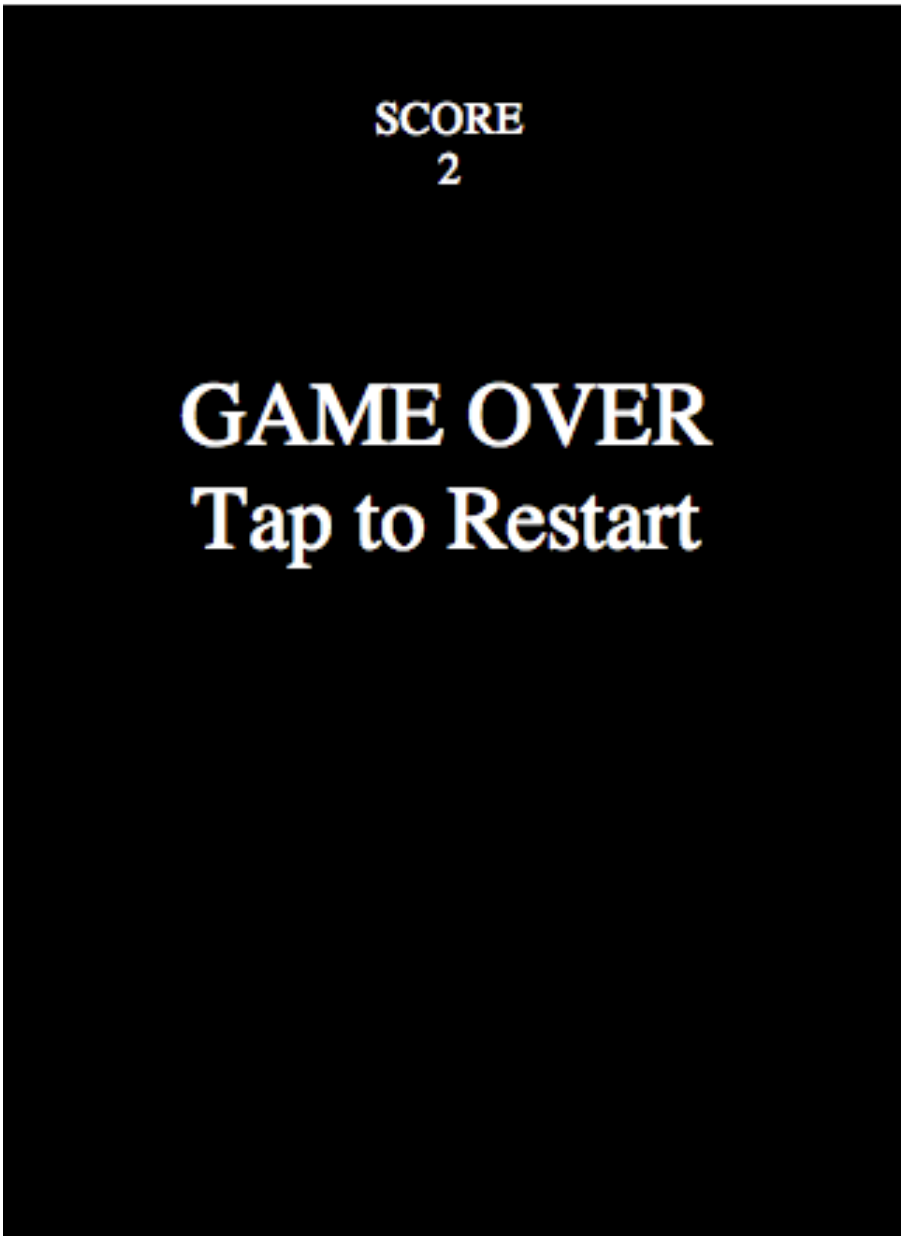
Now that you have your SceneGameOver ready, let's use it in the game!

Since the game will move to SceneGameOver when the penguin is hit by an ice boulder, modify your collision detection code to the following:

```
if (ice.intersect(this.penguin)){
    .
    .
    .
    // Add the following lines
    // Game over
    this.bgm.stop();
        game.replaceScene(new SceneGameOver(this.score));
    break;
}
```

The code stops the background music, and then tells the game to replace the current scene with an instance of SceneGameOver, using the current score as the argument for instantiation.

Save *main.js* and refresh your browser again. Let the penguin get hit by a boulder and watch the screen go black. Clicking on the game over screen should return you to the game.



Your game is finished! However, all the while you've been testing this game on your desktop browser. Isn't the charm of enchant.js that you can run it on a mobile browser?

Let's do that now!

Testing Your Game on Mobile Browsers

Before proceeding, note that for this to work, your machine needs to be running a web server.

While you can't just type "localhost" into your mobile browser address bar to access your local server, you can access it through your local IP address.

There are many ways to get your desktop's IP address. If you're running Windows, you can follow this tutorial. On OS X, you can get it from Network Preferences.

Once you've got your local IP address in hand, you can access your game by using the same URL as on the desktop, but replace localhost with your local IP

address.

For example, let's say your local IP address is 10.10.10.10. If the URL to your game on your desktop is:

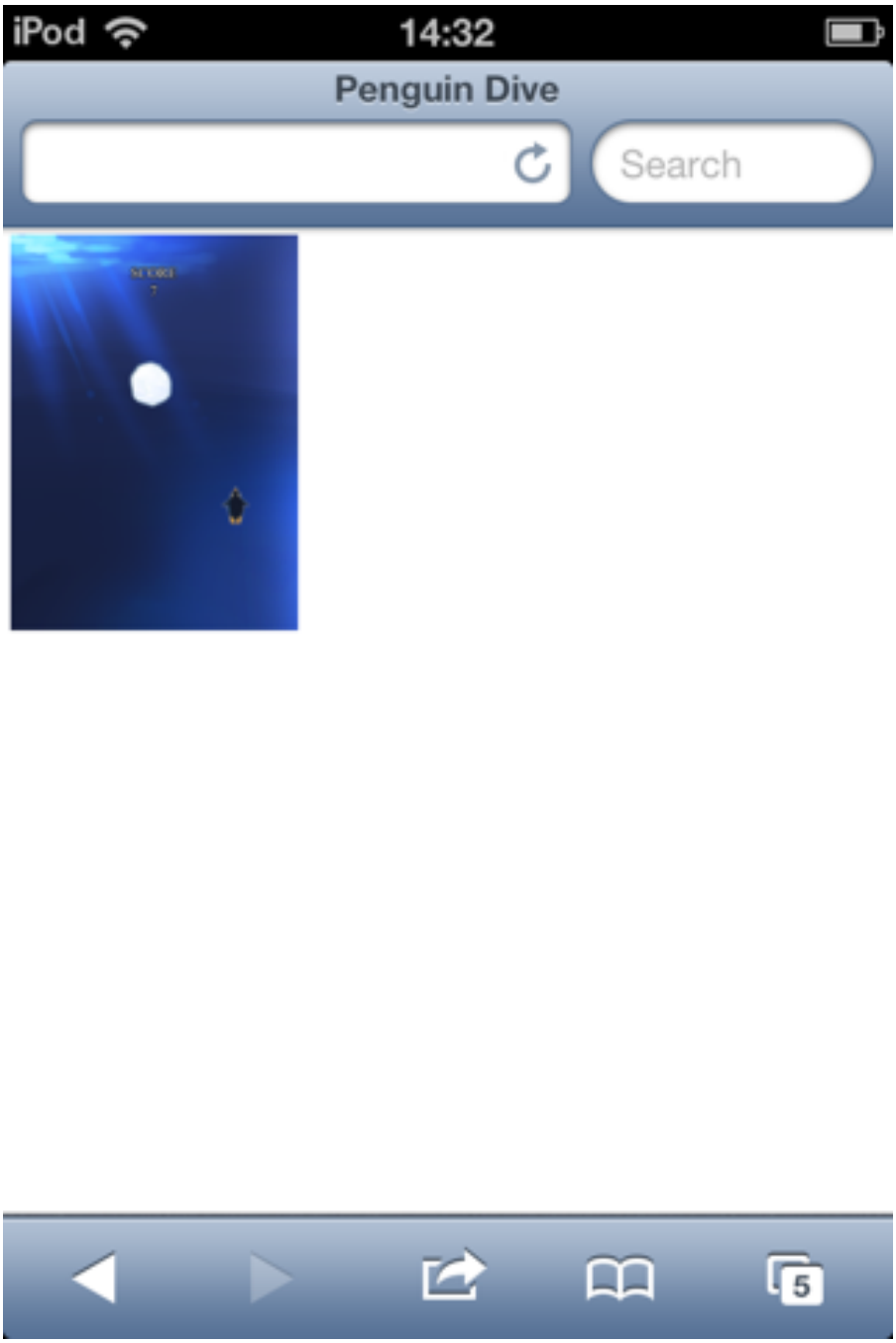
```
http://localhost/~Home/penguindive/
```

Then the URL on your mobile phone would be :

```
http://10.10.10.10/~Home/penguindive/
```

Prepare for iOS Device

If you try running the game from your retina display iOS device, you'll notice that the game is really tiny:



There's no way you're playing that with your big human paws! :]

To fix this, open *index.html* and add the following line to the header section:

```
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1, maximum-sc
```

This sets the size of your game's viewport. To understand more about the viewport, you can read the Apple Documentation on the subject. Simply put, the viewport is the area that determines how your content will be laid out in a mobile browser.

In enchant.js, the Game object will check to see if your game is running on a retina display device. If it is, it will modify this viewport meta tag and adjust the game size to appropriately, so you don't have to worry about your game being displayed incorrectly on retina display devices.

One more thing you'll notice on an iOS device is that your game isn't really on the top left – there are some margins. Let's fix this as well. Modify your HTML body in *index.html* to be as follows:

```
<body style="margin: 0;"></body>
```

This sets the body margin to zero, leaving no white gap between the game and the top left of the browser window.

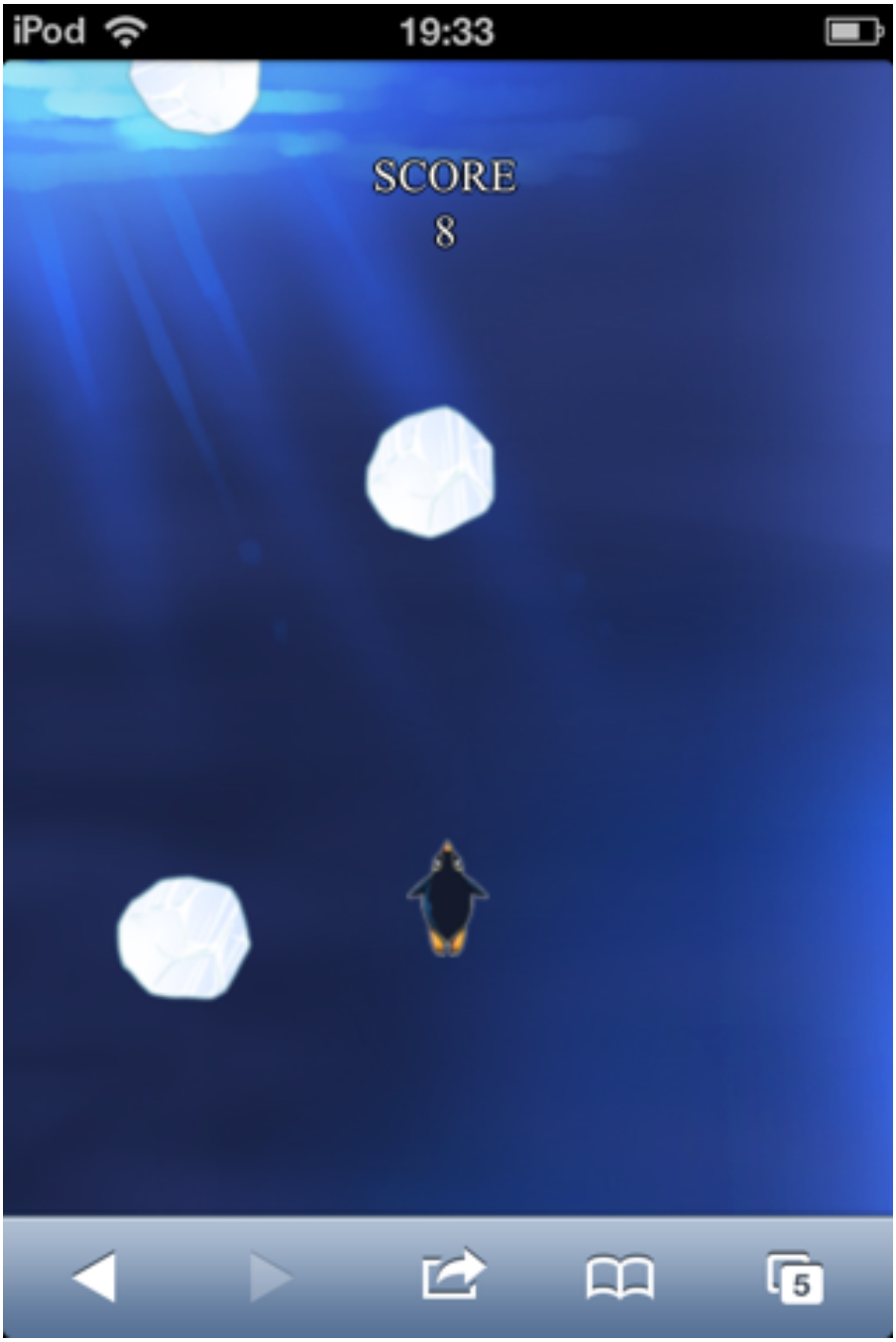
Now the only thing that is in your way is the address bar. Fortunately, you can get rid of it very easily.

Go to *main.js* and add this line after game.start():

```
window.scrollTo(0, 0);
```

On iOS, this line tells the browser to scroll down until the address bar is out of sight, giving you a nice full-screen effect.

Save *main.js* and run the game again. Here's what it should look like on your device:



You can also view the game running on our web site here.

Where To Go From Here?

Congratulations! You have finished your first mobile browser game with enchant.js!

Here is the final project with all of the code from the tutorial.

If you are interested, here are some additional challenges for you:

- Add a fish object that randomly appears like the ice boulder. If the penguin hits the fish, the player get 5 points!
- Add a scrolling overlay on top of the background, using BG_Overlay graphic, for a nice flowing ocean effect.
- Using penguinGameOver.png to show that the penguin got hit instead of suddenly moving to SceneGameOver.

- Increase the game's difficulty over gameplay time. Maybe add more ice boulders? Or have the boulders move faster and faster?

If you want to learn more about enchant.js, here are some great resources I've found:

- The enchant.js official site has some tutorials in English that you might want to check out.
- The enchant.js official blog also has some interesting tutorials.
- jsdo.it even lets you write enchant.js games online!
- Check out 9leap for some games made using enchant.js

I hope you enjoyed this tutorial! If you have any questions or comments, please join the forum discussion below!

Credits

- Background music from oo39
- Sound effects by OSA
- Art by Piti Yindee

This is a post by Tutorial Team member Guts Rodsavas, an iOS development trainer at Software Park Thailand and game developer at Coffee Dog Games.