

From Zero to the Appstore Blueprints for an HTML5-Game

A guide to creating and publishing games with
EaselJS, CocoonJS and PhoneGap Build

Olaf J. Horstmann

From Zero to the Appstore Blueprints for an HTML5-Game

A guide to creating and publishing games
with EaselJS, CocoonJS and Phonegap Build

Sample Chapters

Olaf J. Horstmann

© 2013 Olaf Horstmann. All Rights Reserved.

Please do not share or distribute without written permission.
For inquiries contact me through oh@indiegamr.com.

1 Setting up an EaselJS App

1.1 The Folder Structure

Okay let's jump right in and create the app. The first step will be to go to the *htdocs*-folder of **XAMPP** and create a new folder with the name of our app, I will name it *jumpy*, you can use the same name or whatever comes to your mind. Now we will open a new document in our code-editor and save it as *index.html*. Also we will create another folder in our project folder and name it *js*. In this folder we will place the *easeljs-0.6.0.min.js* and create another file that we will name *app.js*. Those are the basic files that we will need for now, let's go ahead and start coding.

index.html

```
1 <html>
2   <head>
3     <title>jumpy</title>
4     <style type="text/css">
5       * {
6         padding: 0px;
7         border: 0px;
8         margin: 0px;
9         background-color: #000;
10      }
11    </style>
12    <script src="js/easeljs-0.6.0.min.js"></script>
13    <script src="js/app.js"></script>
14  </head>
15  <body>
16
17  </body>
18 </html>
```

This is pretty much all you will have to do in HTML throughout the course of this book and the good part is that all you need to know and worry about are the **<script>** tags:

lines 5 - 9

This will remove all the spacings between any element, because all we want to see on the page is the canvas(our game). Also this will make the background black, we will change that later, but for now this is what we want.

line 12

Loading the EaselJS framework into our project. Every file that we want to use in our project has to be included this way.

line 13

Loading the app-script. Note that our `app.js` is still empty but we will change that in a second.

app.js

```
1  var canvas, stage, image, bitmap;
2
3  function init() {
4      // creating the canvas-element
5      canvas = document.createElement('canvas');
6      canvas.width = 500;
7      canvas.height = 250;
8      document.body.appendChild(canvas);
9
10     // initializing the stage
11     stage = new createjs.Stage(canvas);
12
13     // creating a new HTMLImage
14     image = new Image();
15     image.onload = onImageLoaded;
16     image.src = 'assets/hero.png';
17 }
18
19 // creating a Bitmap with that image
20 // and adding the Bitmap to the stage
21 function onImageLoaded(e) {
22     bitmap = new createjs.Bitmap(image);
23     stage.addChild(bitmap);
24
25     // set the Ticker to 30fps
26     createjs.Ticker.setFPS(30);
27     createjs.Ticker.addListener(tick);
28 }
29
30 // update the stage every frame
31 function tick(e) {
32     stage.update();
33 }
34 window.onload = init;
```

line 1

All variables are declared here at once, if we declared them inside a function they would not be accessible from outside that function or another function. But we'll learn more about this later.

lines 5-8

The **canvas** is the most important element, it is the area that everything will be rendered to, later we will learn about how to set its **width** and **height** to the size of the viewport, but for now we'll stick with 500:250px (or whatever size you like).

Canvas

*The canvas is a native HTML-element that can be used to draw and alternate images and render its output to an HTML document. Drawing-methods can be accessed directly through the context of the canvas, however EaselJS will add a layer of abstraction to the canvas by automatically handling all drawing calls so that all you have to do is placing the images and calling **stage.update()**.*

line 11

The **createjs.Stage** is the second most important element. Everything that we want to show on the screen has to be added to the **Stage** or a child of the Stage.

Stage

The Stage is so to say the bottom-most container. Every container or bitmap has to be placed on the Stage (or on another container that is placed on the Stage) in order to be rendered onto the canvas and visible to the user.

Stage.update()

*Every time you updated the position, scale or rotation or any other parameter of an object you have to call **Stage.update()** to make the changes visible on the canvas. The easiest and most simple way to do this is to call **Stage.update()** at the end of every tick (see the tick()-method of this code part). With every **Stage.update()** the stage will also automatically execute a **_tick()**-method on every child(if there is one).*



<http://www.createjs.com/Docs/EaselJS/classes/Stage.html>

lines 14 - 16

Here we create a new **HTMLImage**, assign a method to it that is triggered **onload**(when the page is done loading) and then give the Image a resource-path to load. - This means, that we have to create a folder named **assets** and place an image-file named **hero.png**.

HTMLImage

*An HTMLImage is the representation of an image-file that is loaded by setting the HTMLImage's parameter **src** to the path of the image-file. One instance of an HTMLImage can be used and displayed multiple times through the use of a **createjs.Bitmap**. So in order to display the same image twice, no second HTMLImage has to be created and the image-file won't have to be loaded twice.*

Game Art

In this book you will not learn about creating graphical assets for your project. If you have no experience in creating graphics, feel free to use the assets in the `Resources/`-folder or browse through OpenGameArt to find free suitable graphics for your game:
<http://opengameart.org/>

lines 22 - 23

Once the Image is loaded we will create a `createjs.Bitmap` with it and add it to the `Stage` via `addChild()`.

Bitmap

The Bitmap is probably one of the most commonly used classes of the EaselJS framework. It basically wraps an `HTMLImage` and gives it the ability to be added to the Stage and easily adjust certain properties like its position or its scale through the simple assignment of property values like `myBitmap.scaleX = 2;`

 <http://www.createjs.com/Docs/EaselJS/classes/Bitmap.html>

addChild()

The `addChild()`-method is available to every Container-Object, the Stage for example is a Container. Adding an object as a child to another object means that the added child-object will be positioned relative to the parent-object and will be repositioned in the render tree. An object can only be child to one parent at a time. If an object is assigned to another parent it is automatically removed from its previous parent.

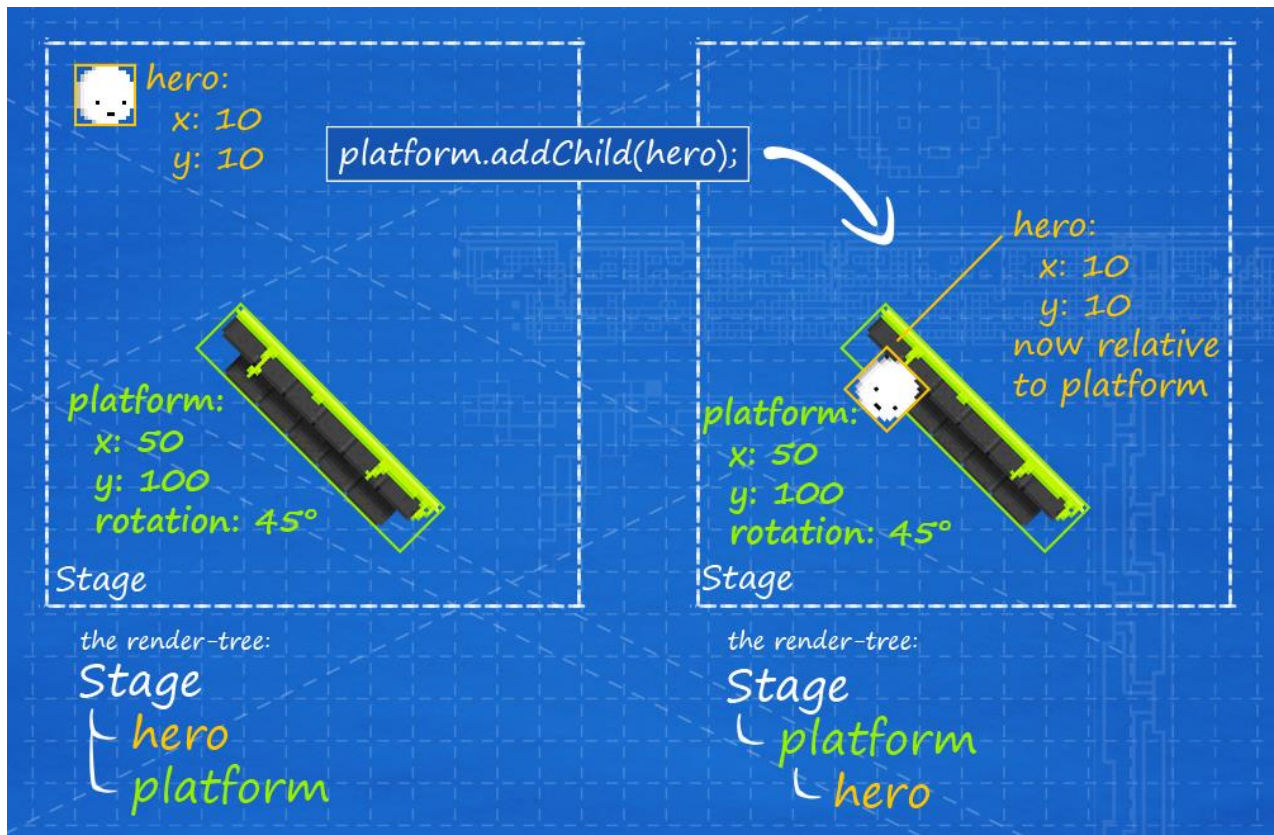


Figure 1 - `addChild()` example

lines 26 - 27

Setting the `createjs.Ticker` to 30FPS and adding the tick-method as a listener.

Ticker

The Ticker is the global time giver of the EaselJS framework. It can be viewed as a heart-beat of the game. Every update of an animation or a tween is based on the Ticker and its FPS-value. The FPS-value determinates how often per second those update-calls are executed. The Ticker will automatically execute any added listener-function or the tick-function of any added listener-object, if adding a listener-object, make sure that the object contains a `tick()`-function to execute.

 <http://www.createjs.com/Docs/EaselJS/classes/Ticker.html>

lines 31 - 33: `tick()`

The tick-method is called 30 times per second and executes a `stage.update()` every time meaning that the stage-contents are redrawn 30 times per second.

line 35

The `init()`-function from line 3 is set as the window's `onload`-method. This means that `init()` will be executed once everything is loaded.

Executing the Code

Before you go ahead and execute the code, make sure that there is an image by the name *hero.png* in the `assets`-folder for a reference you can also take a look at the resources of *ChapterResources/Chapter1/src/*. When we now open the *index.html* it should look like the following figure.

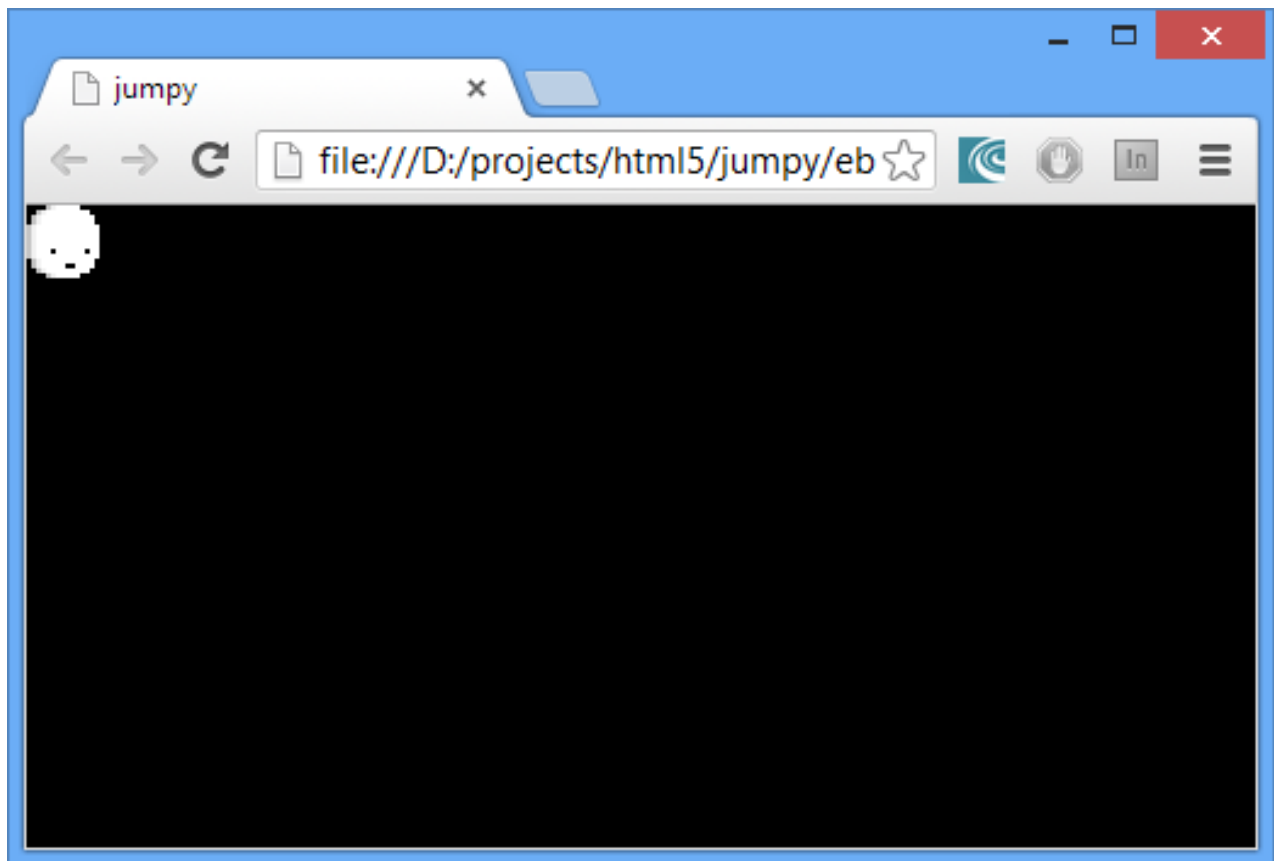


Figure 2 - Chapter 1 Result

1.2 Debugging the Game

If your game should at any point not do what you expect it to do, or not run at all, you should open the **Developer-Tools** and look for errors there. You can open the **Developer-Tools** in Chrome by pressing **F12** (or for Firefox click on the FireBug-Symbol). Now navigate to the **Console-Tab** and look for any red text. That's usually the part that will cause your app to fail. In the example below you can see a message stating an error in line 15 of our *app.js*. The error says **imag is not defined** – the error here was caused by a typo, instead of **image** I wrote **imag**, and because there is no such variable defined in the game, it will fail.

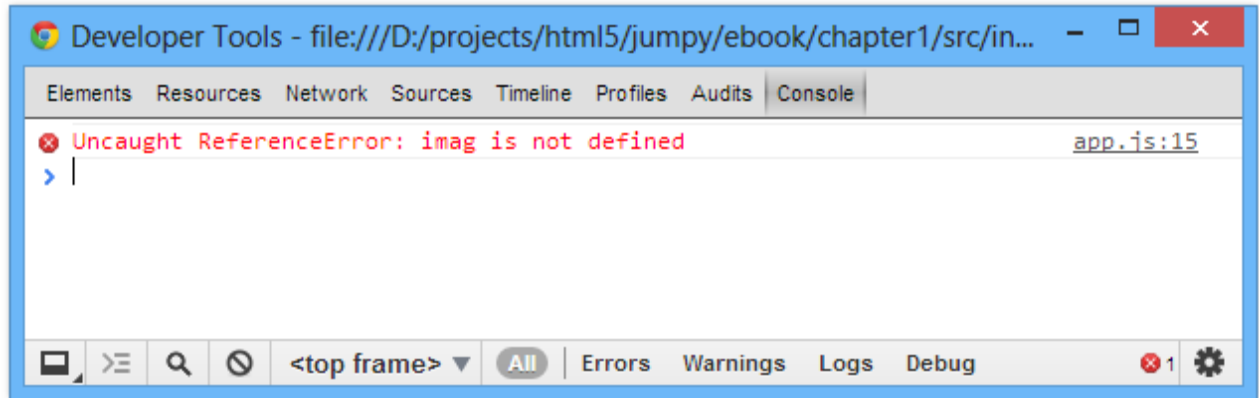


Figure 3 - Chrome Developer Console

Chapter Recap

- Setup a basic EaselJS project
- Create a canvas and initialize the stage
- Load an image-file and create a Bitmap from it
- Add elements to the stage
- Change the refresh rate of the application

If this introduction to EaselJS was too short or not detailed enough, you can take a look at this video on getting started with EaselJS by *Sebastian DeRossi* from *gskinner.com*.



<https://youtu.be/OWHJa0jKJgo>

2 Creating a Hero that moves

In this chapter we will create a **class** that describes our hero and it is going to be based on the Bitmap-class that was introduced in the last chapter. (No worries, you will learn in a second what a **class** is)

hero.js

```
1 (function (scope) {
2     function Hero(image) {
3         this.initialize(image);
4     }
5     Hero.prototype = new createjs.Bitmap();
6
7     // save the original initialize-method so
8     // it won't be gone after overwriting it
9     Hero.prototype.Bitmap_init = Hero.prototype.initialize;
10
11    // initialize the object
12    Hero.prototype.initialize = function (image) {
13        this.Bitmap_init(image);
14        this.snapToPixel = true;
15
16        this.velocity = {x:0,y:-15};
17    }
18
19    Hero.prototype.onTick = function () {
20        this.velocity.y += 1;
21        this.y += this.velocity.y;
22    }
23
24    scope.Hero = Hero;
25 } (window));
```

lines 1 & 25

The Hero-class is being wrapped inside an anonymous function and executed with the **window** as its namespace.

Classes in JavaScript

JavaScript does not really implement the concept of a **class**. This means that if we want to create a custom class like the hero we can only do so by utilizing functions and using the functions **prototype** to define properties and methods. The concept of this can be rather complicated and for the reason that explaining all of it would go beyond the scope of this book I will only refer to another source if you want to know more about this topic.

 [https://developer.mozilla.org/en-US/docs/JavaScript/Introduction to Object-Oriented JavaScript](https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript)

 <http://phrogz.net/JS/classes/OOPinJS.html>

line 5

The Hero is created as a sub-class of Bitmap. This means that our Hero-class is inheriting every method, property and ability from the Bitmap-Class which is now the Super-class to the Hero-class.

Inheritance

Inheritance means that the newly created class, that inherits from another class has exactly the same capabilities as the other class *PLUS* all additional capabilities that we will implement into the new class.

e.g.: Abilities like defining an x- and y-position or a rotation of the Hero are automatically implemented.

 [https://developer.mozilla.org/en-US/docs/JavaScript/Introduction to Object-Oriented JavaScript#Inheritance](https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript#Inheritance)

line 9

We reference the original initialize-method so we can still use it through the saved reference after overwriting it.

Overwriting the Initialize-Method

Why do we overwrite the original initialize-method? —The concept of EaselJS is that every class has an initialize-method that can be called to initialize an object. And we will follow this pattern. There are several reasons, to give one example: If every class was different it would be a lot harder to read it at a later point in case we wanted to change anything and the same goes for when working on a project with more than one person.

lines 12 - 18

This is the new initialize-method. It calls the old method and hands over the image-parameter so the Bitmap-class can take care of the image-creation. **snapToPixel** is set to **true** and a velocity is being initialized. Finally the initialized object is added as a listener to the Ticker that will execute the hero's tick()-function every frame.

.snapToPixel

Pixelsnapping will, as the name already says, automatically round the rendering-position of the object to full pixels. This is usually done for performance reasons, because anti-aliasing and subpixel-rendering can be a heavy burden to the hardware of some devices.

Note: In order for **snapToPixel** to work, it also has to be set to **true** on every parent object and on the Stage you will have to set the property **snapToPixelEnabled** to **true** as well.



http://www.createjs.com/Docs/EaselJS/classes/DisplayObject.html#property_snapToPixel



http://www.createjs.com/Docs/EaselJS/classes/Stage.html#property_snapToPixelEnabled

lines 19 - 22

Defining a onTick()-method that is called every frame to update the hero's properties.

*For every DisplayObject that is added to the stage the **onTick**-Method is automatically called every frame. If an object is not child of the stage, we would have to set a custom listener to the createjs.Ticker.*



http://www.createjs.com/Docs/EaselJS/classes/DisplayObject.html#event_tick

lines 20 & 21

These two lines update the hero's velocity as well as its y-position by the value of the y-velocity.

Velocity

*The **velocity** is initialized with a negative value, this means that it points upwards on the stage. With every frame the velocity is increased by one, so within 16 frames it will be positive and pointing downwards.*

Updating the position of the hero by its current velocity will so result in a very simple but still good looking form of jumping up and falling down as illustrated by the figure below (the figure is an example, positions are not exact).

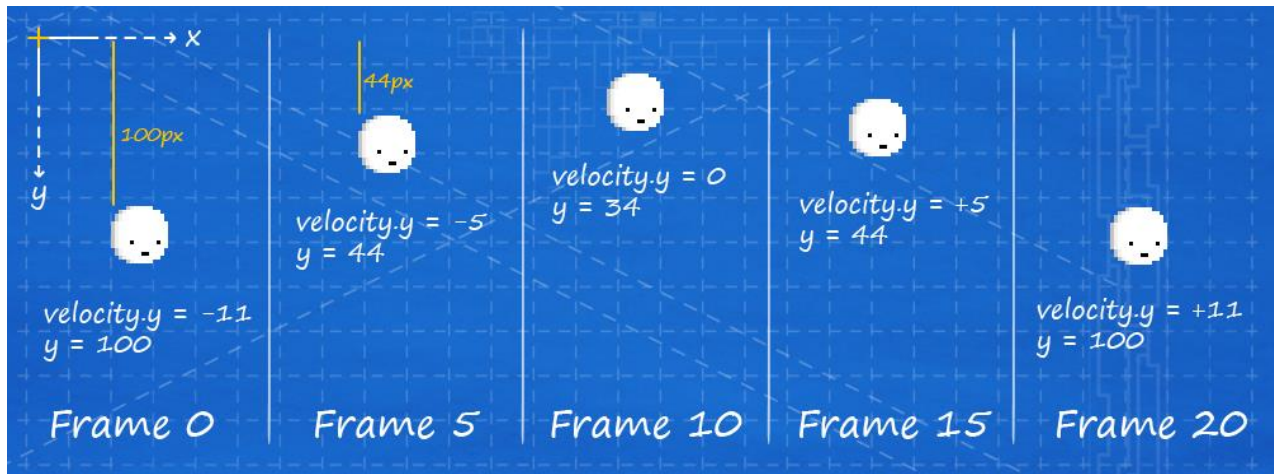


Figure 4 - Illustrating Velocity

line 24

The Hero-class is handed over to the namespace which is in this case the window. We can now create a Hero-object from anywhere within the application by executing **new Hero(imageForTheHero)**.

Namespacing

Namespacing is used when multiple frameworks are used together and you want to prevent classes from interfering with another class from another framework that has coincidentally the same name. The EaselJS-framework for example is using the namespace **createjs**. The namespace we chose, **window**, does not have to be typed in the code as everything in JavaScript is a property of **window**, so window equals a **global** or **no namespace**.

Custom Namespacing

If you choose to use a custom namespace, you simply have to hand over your namespace instead of **window** in **Line 27**. However you have to make sure that the namespace exists by defining it once before a class is loaded, for example you could add this **before Line 1**:

```
this.mynamespace = this.myNameSpace || {};
```

and in **Line 25**:

```
} (this.mynamespace));
```

But remember: Whenever you want to create a new object from one of your classes, you have to go through the namespace:

```
var myObject = new mynamespace.MyClass();
```



<http://www.codeproject.com/Articles/19030/Namespace-in-JavaScript>

Updates in app.js and index.html

So now that we have our Hero-Class, how do we create and add a Hero to the stage? — Just like we created a Bitmap in the last chapter, we now replace all Bitmap-reference with our Hero-Class:

`app.js` line 1

Instead of naming the variable **bitmap**, we will give it the name **hero**.

The name of the variable does not HAVE to be hero, you can chose whatever name you like, however you will have to remember it in order to use it later.

`app.js` lines 22-24

The second step is replacing the instantiation of the Bitmap with the instantiation of the Hero and giving the Hero a position:

```
22     hero = new Hero(image);
23     hero.x = hero.y = 150;
24     stage.addChild(hero);
```

Now one more change in the `index.html` and we are good to go: We have to tell the index-file that it needs to load the hero.js otherwise no hero could be created if there is no class by that name, so we add the following code one line above the loading of app.js

`index.html` line 12

```
12     <script src="js/hero.js"></script>
```

If you open the `index.html` you should now see a jumping hero, you can press F5 to reload the page and see it jump again.

Chapter Recap

- Creating a custom class (`hero.js`)
 - utilizing the initialize-method
 - implementing a custom tick()-function and assigning it to the createjs.Ticker
- Working with velocities and motion by updating the position of an object every frame
- Implementing the new class to the app, creating an instance from it and adding it to the stage

7 Implementing a collision detection

In case you did not read the previous chapter: There is a file in the resources-folder: `'ndgmr.collision.js'`, this is a utility class that will take of the collision detection for you, all we have to do is add a couple of lines to our `hero.js` in order to have the game checking for a collision and taking certain actions if needed.

7.1 The Code

index.html

So first we need to place the `ndgmr.Collision.js` in our project's js-folder and load it in the `index.html` just like the other JavaScript files.

hero.js

There are many ways to decide what objects we want to include in the collision check, in this case we take all children of the hero's parent-container. This is usually not a good idea when that parent-container is a stage, but in this case it's okay since there are only two objects on the stage. Later when we are going to add some eye-candy we will have to create an own container for everything to get no confusion with the collisions. So in order to detect the collision and take action we add this to the `onTick()`-method:

```
22     var c, col, collObjs=this.parent.children, dir;
23     for ( c = 0; c < collObjs.length; c++ ) {
24         if ( collObjs[c] == this ) continue;
25         col = ndgmr.checkRectCollision(this,collObjs[c]);
26         if ( col ) {
27             dir = this.velocity.y < 0 ? 1 : -1;
28             this.y += col.height * dir;
29             this.velocity.y = 0;
30             //not always save to "break" but here it is ;- )
31             break;
32         }
33     }
```

lines 23 & 24

Initializing the variables and retrieving and parsing through all children of the hero's parent container.

line 25

If the current child is the hero itself, we continue with the next child, because we don't want the hero to check for a collision with itself.

line 26

Using the `ndgmr.Collision`-class for collision detection, returns **null** if no collision occurred or returns the size of the intersection in case of a collision.

line 27

Depending on whether the hero is jumping up, bumping his head on the bottom of a platform, or flying down, landing on a platform, we have to determine the direction of travel to adjust the position in the opposite of that direction.

lines 28 - 29

If a collision occurred, we reposition the hero by the size of the intersection and set the vertical velocity to 0, the hero landed on an obstacle and the 'correct' physical response is to stop.

line 31

Just in case a collision occurred we break the loop, we don't need to check for additional collisions. In other scenarios you might need to check of every collision, so don't do it always like this.

After implementing the loop into the hero's `tick()`-method the hero should now not pass through the platform any more. Congratulations, with just a few lines of code you now added a collision-detection and with that a little bit more realism to your game.

7.2 Possible Issues

Passing through

With this way of checking for a collision there is one possible issue that might occur is some very rare cases. Imagine the hero moving so fast that in one frame it is above a platform and in the next frame it is below a platform and therefore it should have collided but no collision is detected. However we will look past this issue for now, the following figure should emphasize how very unlikely this will happen.

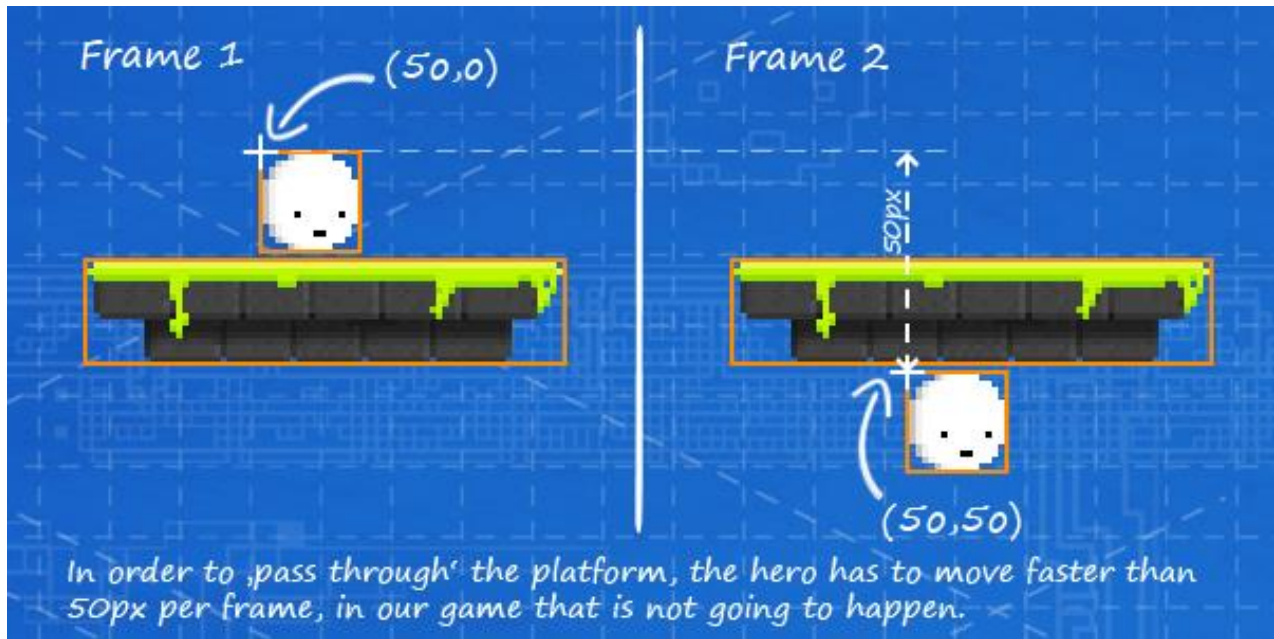


Figure 5 – The hero passing through a platform

Being stuck “inside”

Another possible issue, that can occur, is that the hero can be surrounded by another object or to surround another object, so that after we adjust the position of the hero, the collision still occurs. This can be caused if the velocity of the hero is greater than the size of the colliding object (so either with a very high velocity or a very small obstacle). This issue can be resolved by simply rechecking for a collision after the hero’s position was adjusted and repeat the process if there was still a collision detected.

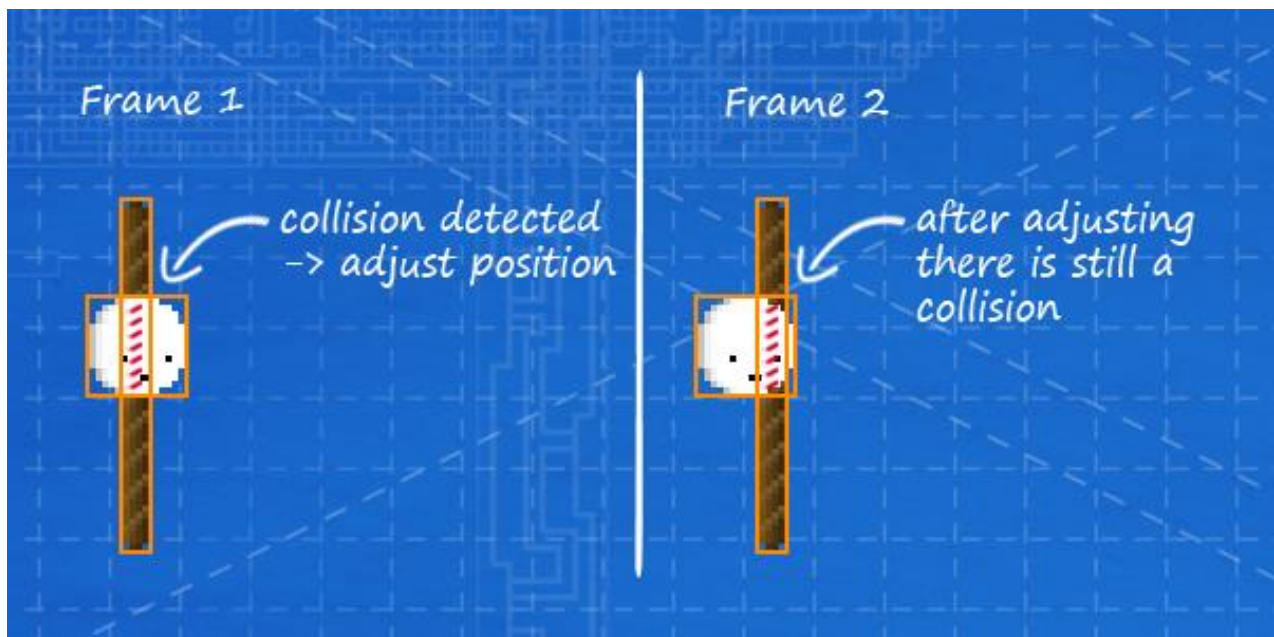


Figure 6 - The hero being stuck

But again: Right now we don't want to overcomplicate things and with the current setup we don't need to worry about that issue. So for now it is enough that we are aware, that such an issue exists and we can tackle it once we change the setup and there is a chance that it might occur.

PART II

Deploying the Application to Mobile Devices

1 Technologies

1.1 Available Technologies

There are quite a few technologies that will help you to bring your HTML5 app to mobile devices. Here is a list of the few most important tools and services:

- 1) PhoneGap: <http://phonegap.com> (not part of this book)
- 2) Ejecta: <http://impactjs.com/ejecta> (not part of this book)
 - EaselJS adaption: <https://github.com/apitaru/Ejecta-HEART-CreateJS>
- 3) **PhoneGap Build Service:** <http://build.phonegap.com>
- 4) **CocoonJS Build Service:** <https://ludei.com>
- 5) TriggerIO Build-Service: <http://trigger.io> (not part of this book)
- 6) AppMobi, directCanvas: <http://appmobi.com> (not part of this book)

And there are even more, however what I learned while creating HTML5 apps was that both the PhoneGap and the CocoonJS-service are the tools that are the most easiest to use and you won't have to change one single line of code to both run the app in the browser and as an app. But since both build-services rely on different ways and technologies to package the application there are pros and cons for each of the two.

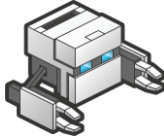

PhoneGap Build Service

What the PhoneGap Build Service will do, is to package your contents as a native app and display them on a WebView, which is basically a mobile browser without all the controls like the url-input. So the performance and the capabilities really depend (especially for Android) on the device's specifications and capabilities.

CocoonJS Build Service

In contrast to PhoneGap is CocoonJS more like an abstraction-layer or an interface that will translate your app's HTML5-Canvas- and JavaScript-operations to be executed in a native context. Through this method the packaged application is able to take full advantage of the device's GPU and has a much better performance and runs more reliable on a broader range of devices (especially Android). Rendering HTML-elements and CSS can be achieved through an extension that is available to CocoonJS.

1.2 Comparison

	 PhoneGap Build-Service	 COCOON ^{JS}
Feature		
Price	Free Plan available, 10\$/mo.	Currently Free(beta)
Platforms	iOS, Android, WindowsPhone, BlackBerry, WebOS, Symbian	iOS, Android
Performance	WebView	Native
Max. Project-Filesize	10MB	Currently: 30MB 200MB Premium planned
Included Extensions/Plugins	Childbrowser, Barcode Scanner, Analytics, FacebookConnect, GenericPush, Custom Plugins can be imported (deeper knowledge required)	Ads(almost any Network), WebView, InApp Purchase (AppStore and PlayStore), iOS Gamecenter, Multiplayer, Camera, Notification, Twitter, Facebook, Box2D,
Testing	App needs to be built and installed once and then can be updated through the web interface.	Testing-App available to import and load HTML5-app as ZIP-file(from online and local). Debug console available.
Service	Building, Codesigning, (technology is available for local building)	Building
Branding	Splash-Screen (removable)	Splash-Screen (not removable in free version)
Conculsion	If your app works fine in the mobile browser and is less than 10MB, this is probably the best service to use.	If your app relies on intense graphics that require native performance you should choose this service.
Android Specific ¹	CAN run/render differently on each Android device	Runs/Renders identical on almost any (even older) Android device

¹ based on personal testing on various Android devices

If you liked the sample,
you can get the full book + resources at:
<http://indiegamr.com/zerotoappstore>