* [Cocos2d-JS tutorial 2: Hello World Cocos2d-JS](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter2/en/index.html)

# Hello World of Cocos2d-JS

In this tutorial, you will learn how to setup a new Cocos2d-JS project from scratch. We will start with a brief description of Cocos2d-JS's directory structure.

cocos2d-js workflow with cocos-console :

<http://www.cocos2d-x.org/wiki/Create_a_new_JS_project_with_cocos-console>

<http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter2/en/index.html>

Making the project skeleton

cocos **new** Parkour -l js

* The **res** directory. It contains all the resource files needed by our project. Right now it only contains some sample images. If you want you can add some meta files for your game or some awesome game music, you should put them in this folder. You should choose a proper name for each file.
* The **src** folder. It contains all of your actual game logic. If your game consists of hundreds of JavaScript source files, you might want to organize them into small chunks using subfolders. Right now our template has two JavaScript source files. The **app.js** file contains the first scene's code. The **resource.js**file defines some global variables for the resources.
* The **index.html** file is the entry point of a HTML5 based web application. It is a HTML5 compatible format. It defines some meta data like setting the viewpoint and fullscreen parameters.
* The **project.json** file is the configuration file for our project. Please refer to [project.json](http://www.cocos2d-x.org/docs/manual/framework/html5/v3/project-json/en) for more details.
* The **main.js** is the place to create your first game scene and show it in the browser. You also can define the resolution policy and preload your resources in it.

#### Analysis of the project's execution path

It is very important to know a program's execution path.

In our project, the game is loaded into browser from index.html. Then it moves to **frameworks/Cocos2d-html5/CCBoot.js**. In this file, it will try to load the project configuration from the project.json file.

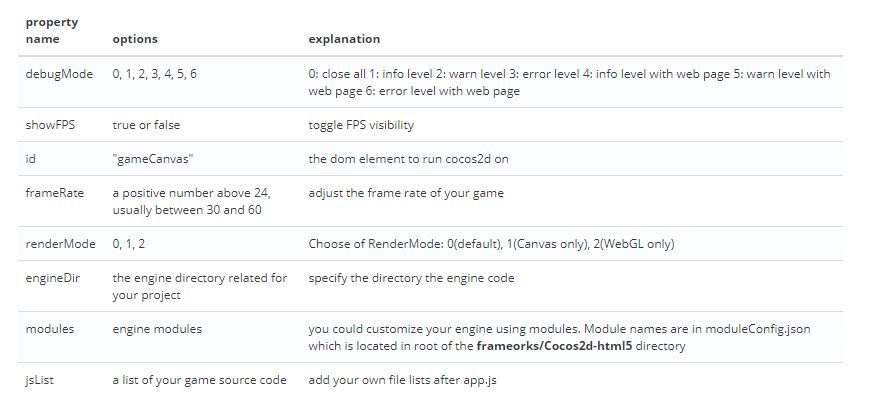
{ "project\_type": "javascript", "debugMode" : 1, "showFPS" : true, "frameRate" : 60, "id" : "gameCanvas", "renderMode" : 0, "engineDir":"frameworks/cocos2d-html5", "modules" : ["cocos2d"], "jsList" : [ "src/resource.js", "src/app.js" ] }

In this piece of code from project.json, there is a object property named **engineDir** which is the key point to decide the execution path of the following program. In the default case, we have specified the engineDir.

The main.js file will be loaded after **frameworks/Cocos2d-html5/CCBoot.js** file and it will initialize the configuration and load all the JavaScript files specified by **modules** and **jsList** in our project.json file.

#### Hiding the FPS in the left corner of your game screen

The section may be a little bit trivial. We can achieve this by modifying **showFPS** property to **false** in **project.json**.

There are many thing we can tweak by modifying this object's properties. Here is a table with each property's intention:

#### Modify the design resolution size

Open you main.js and in the function **cc.game.onStart** change the Resolution Size to 480 by 320.

You should also change the resolution policy to **SHOW\_ALL**:

**cc**.**view**.setDesignResolutionSize(480, 320, **cc**.ResolutionPolicy.SHOW\_ALL);

If you are curious about why you should do this, please refer to [Resolution Policy Design for Cocos2d-html5](http://www.cocos2d-x.org/docs/manual/framework/html5/v2/resolution-policy-design/en) for more information.

* [Cocos2d-JS Tutorial 3: Make Your First Game Scene](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter3/en/index.html)

Before creating the first scene for your game, you should be familiar with some basic concepts of Cocos2d

# Making Your First Game Scene

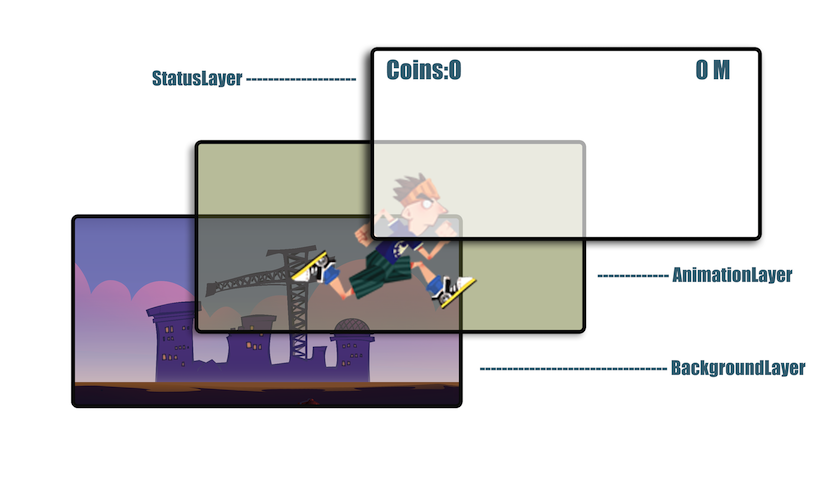
## Basic concepts

In a Cocos2d game, every element is a node. The game is constructed mostly by three kinds of nodes:

* Scene
* Layer
* Sprite
* For now we will focus on the Layers in this game, you can find more details about the Scene and Sprite [here](http://cocos2d-x.org/wiki/Director_Scene_Layer_and_Sprite).

#### Layer

A cc.Layer is a cc.Node which knows how to draw itself and may be semi-transparent, allowing players to see other layers behind them. cc.Layer is very useful for defining your game's appearance and behaviour, so it will take a lot of time to deal with cc.Layer subclasses to reach your expectation.

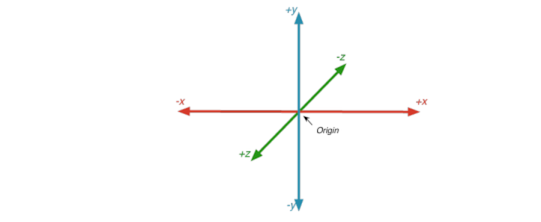


While complex applications will require you to define custom cc.Layer subclasses, Cocos2d provides several predefined layers. Some examples include cc.Menu (a simple menu layer), cc.ColorLayer (a layer that draws a solid color), and cc.LayerMultiplex (a layer that lets you multiplex its children, activating one at a time while disabling the others).

Layers may contain any cc.Node as a child, including cc.Sprite, cc.Label, and even other cc.Layer objects. Because layers are a subclass of cc.Node, they can be transformed manually or by using a cc.Action.

### Coordinate system

Cocos2d-JS uses the same coordinate system as OpenGL, which is the so-called “Right-handed Cartesian Coordinate System”. It is popular in the gaming industry, however, it is different from the traditional top left coordinate system which used in web-page design.



More details about the coordinate system can be found [here](http://cocos2d-x.org/wiki/Coordinate_System)

#### Anchor point

The anchor point is used for both positioning and rotation of an object. The anchor point's coordinate is a relative coordinate. For example, the anchor point in position (0, 0), which we always define in Cocos2d-x as cc.p(0 , 0), corresponds to the most bottom-left point of that object, while cc.p(0.5, 0.5) corresponds to the center of the object. When setting the position of an object, the object is positioned such that the anchor point will be at the coordinates specified with the setPosition() call. Similarly, when rotating the object, it is rotated around the anchor point.

The properties can be set as attributes in Cocos2d-JS v3.0.

For example, this sprite has an anchorPoint of cc.p(0, 0) and a position of cc.p(0,0):

// create sprite

var sprite = new cc.Sprite ( "bottomleft.png" ) ;

sprite.attr({

x: 0,

y: 0,

anchorX: 0,

anchorY: 0

});

this.addChild ( sprite ) ;

### Action

More details about Actions can be found [here](http://cocos2d-x.org/wiki/Actions).

Example of moving a sprite using the cc.MoveBy action:

// Move **a** sprite 50 pixels **to** the **right**, and 10 pixels **to** the top over 2 seconds.

sprite.runAction(**new** **cc**.MoveBy(2, **cc**.**p**(50, 10)));

### Animation

More details about Animations can be found [here](http://cocos2d-x.org/wiki/Animations).

Example of playing an animation:

**var** animation = **new** cc.**Animation** ( ) ;

**for** ( **var** i = 1 ; i < 15 ; i ++ ) {

**var** frameName = "res/Images/grossini\_dance\_" + ( ( i < 10 ) ? ( "0" + i ) : i ) + ".png" ;

animation.addSpriteFrameWithFile ( frameName ) ;

}

animation.setDelayPerUnit ( 2.8 / 14 ) ;

animation.setRestoreOriginalFrame ( true ) ;

**var** action = **new** cc.**Animate** ( animation ) ;

sprite.runAction ( **new** cc.**Sequence**( action, action.reverse ( ) ) ) ;

### Scheduler and Timer callbacks

More details about Scheduler and Timer Callback can be found [here](http://cocos2d-x.org/wiki/Scheduler_and_Timer_Callback).

### EventManager

Cocos2d-JS v3.0 migrated a new mechanism for responding to user events.

The basics:

* **Event listeners** encapsulate your event processing code.
* **Event Manager** manages listeners of user events.
* **Event objects** contain information about the event.

To respond to events, you must first create a cc.EventListener. There are five different kinds of EventListeners:

* cc.EventListenerTouch - responds to touch events
* cc.EventListenerKeyboard - responds to keyboard events
* cc.EventListenerAcceleration - responds to accelerometer events
* cc.EventListenMouse - responds to mouse events
* cc.EventListenerCustom - responds to custom events

Then, attach your event processing code to the appropriate callback on the event listener (e.g. onTouchBeganfor EventListenerTouch listeners, or onKeyPressed for keyboard event listeners).

Next, register your EventListener with the **cc.eventManager**.

When events occur (for example, the user touches the screen or types on the keyboard), the cc.eventManager distributes **Event objects** (e.g. EventTouch, EventKeyboard) to the appropriate EventListeners by calling your callbacks. Each Event object contains information about the event (for example, the coordinates where the touch occurred).

Please refer to [EventManager](http://www.cocos2d-x.org/docs/manual/framework/html5/v3/eventManager/en) for more details.

## Making the game scene

In last tutorial, we have analysed the execution path of a Cocos2d-JS game. Our first game scene is loaded in the function **cc.game.onStart** of main.js. Here is the code snippet which does the real trick:

**cc**.game.onStart = **function**(){ **cc**.**view**.setDesignResolutionSize(480, 320, **cc**.ResolutionPolicy.SHOW\_ALL); **cc**.**view**.resizeWithBrowserSize(true); //load resources **cc**.LoaderScene.preload(g\_resources, **function** () { **cc**.director.runScene(**new** HelloWorldScene()); }, this); };

Here, we use cc.LoaderScene to preload the resources of our game and after loading all resources, the director will run our first scene.

**Note:**

The **cc.game** is actual game object which will initialize game configuration and launch games.

#### Cleanup work

Before we can create our own scene, we need to clean some stuff up.

##### **Cleanup the app.js**

This process is very simple. First, we should delete all the contents of app.js, because we will rewrite it from scratch.

Secondly, we should change this line in main.js:

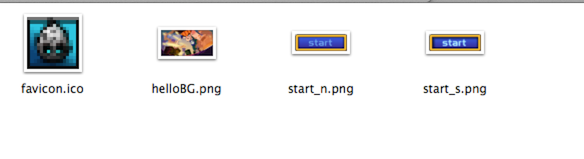
**cc**.director.runScene(**new** HelloWorldScene());

to

**cc**.director.runScene(**new** MenuScene());

When the game starts, we want to run the MenuScene which we will define instead of the HelloWorldScene.

At last, we should add the resources for our scene and define some resource variables for easy access.



Open resource.js and change its content to:

**var** res = {

helloBG\_png : "res/helloBG.png",

start\_n\_png : "res/start\_n.png",

start\_s\_png : "res/start\_s.png"

};

**var** g\_resources = [

*//image*

res.helloBG\_png,

res.start\_n\_png,

res.start\_s\_png

];

#### Define your first scene, the MenuScene

Open app.js and start to define the MenuLayer:

**var** MenuLayer = cc.Layer.extend({

ctor : **function**(){

*//1. call super class's ctor function*

**this**.\_super();

},

init:**function**(){

*//call super class's super function*

**this**.\_super();

*//2. get the screen size of your game canvas*

**var** winsize = cc.director.getWinSize();

*//3. calculate the center point*

**var** centerpos = cc.p(winsize.width / 2, winsize.height / 2);

*//4. create a background image and set it's position at the center of the screen*

**var** spritebg = **new** cc.Sprite(res.helloBG\_png);

spritebg.setPosition(centerpos);

**this**.addChild(spritebg);

*//5.*

cc.MenuItemFont.setFontSize(60);

*//6.create a menu and assign onPlay event callback to it*

**var** menuItemPlay = **new** cc.MenuItemSprite(

**new** cc.Sprite(res.start\_n\_png), *// normal state image*

**new** cc.Sprite(res.start\_s\_png), *// select state image*

**this**.onPlay, **this**);

**var** menu = **new** cc.Menu(menuItemPlay); *//7. create the menu*

menu.setPosition(centerpos);

**this**.addChild(menu);

},

onPlay : **function**(){

cc.log("==onplay clicked");

}

});

Let's go through all the details from 1-6:

1. It calls the init function of its super class.
2. Get the screen size of you game.
3. Calculate the center point of your screen which will be used to center background images.
4. Create a background image using a file name and set its position to the center of the screen. Then, add the sprite to MenuLayer as a child.
5. Call the MenuItemFont's setFontSize function to adjust the font size. It is not used in this example, but if you want to use MenuItemFont to create some menu items, it will affect the menu item label size.
6. Create a menu with two images, one for the normal state and another for the selected state. Then the menu's position is set to the center of the screen. At last, the menu is added to the current layer.

We should also define a Menu scene:

**var** MenuScene = cc.Scene.extend({

onEnter:**function** () {

**this**.\_super();

**var** layer = **new** MenuLayer();

layer.init();

**this**.addChild(layer);

}

});

The process of creating a MenuScene is very straightforward. You defined a variable which derived from cc.Scene. You should remember the sign **extend**, which is used for external classes.

Once the scene is created, the **onEnter** function should be defined. It defines the MenuLayer as it's child. We can also define a **ctor** function instead of the onEnter function. The onEnter function is called after ctor function.

The related sample project can be downloaded [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter3/res/Parkour.zip)

* [Cocos2d-JS Tutorial 4: Design and Make Your Gameplay Scene](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter4/en/index.html)

# Design and Make Your Gameplay Scene

## Introduction

In this tutorial, you will learn how to design and construct the gameplay scene. Every game needs some kind of gameplay scene. This tutorial will show the general scenario of a gameplay scene.

From the previous tutorial, we know that we can use different layers to separate the logic of a specified scene.

Here is the final result of our gameplay scene:



There is a background with some buildings, our hero and some HUD(heads-up display) elements to show some statistics about the current game. We divide the gameplay scene into three parts: the background layer, the animation layer and the status layer.

### Background Layer

Basically, every game needs a background of some sort. Sometimes the background is just a static image which occupies the entire screen. Other times you may want the background layer to move at a constant or varying speed. Sometimes the background even shows us parallax effects--different layers move at various speed, the nearest layer moves faster and the farthest layer moves slower to simulate that objects are near or far.

In later tutorials, we will introduce tiled maps which are very useful to construct parallax background. In this tutorial, in order to keep things simple, we just use a simple static image to represent the game's background.

**Note**: We can move the background to mimic the effect of our game hero running, keeping the hero at the center of the screen. We will see many such tricks during our development process.

### Animation Layer(Gameplay Layer)

The animation layer contains all game elements that animate, collision detection and other game logic. This layer is sometimes also called the **GameplayLayer**. You can choose to name it what you want. In this layer, we organize the key part of our gameplay. In general, we will design game objects, s level spawner(which are also called level managers), collision detection between different game objects and check if the player has won or lost the game.

In theory, we don't need to separate this layer into smaller layers. We can use composition and delegation to handle things properly.

### Status Layer(HUD Layer)

In video gaming, the HUD (head-up display) is the method by which information is visually relayed to the player as part of a game's user interface. It takes its name from the head-up displays used in modern aircraft.

The HUD is frequently used to simultaneously display several pieces of information including the main character's health, items and an indication of game's progression (such as score or level). You can refer to [this link](http://en.wikipedia.org/wiki/HUD_(video_gaming)) for more information on HUDs.

To make things simpler, we put this information into a separate layer called StatusLayer. Because these items are always displayed on top of other game elements, placing them on a separate layer will make our life easier without caring about the Z-order display issues.

## Coding in action

### Preparation

Start by adding two images(**PlayBG.png** and **runner.png**) to the **res** directory.

In the previous tutorial, we have added all resources variables in **resource.js**. Since two more images have been added, **resource.js** should also be changed to this:

sprite in another js files, we can easily access these variables.

Since we will add four JavaScript files: PlayScene.js, AnimationLayer.js, BackgroundLayer.js and StatusLayer.js. We need to tell the Cocos2d-x engine to load these files when the game starts. We do this by changing **project.json** to make it aware of the new source files:

"jsList" : [

"src/resource.js",

"src/app.js",

"src/AnimationLayer.js",

"src/BackgroundLayer.js",

"src/PlayScene.js",

"src/StatusLayer.js"

]

In the future, each time when you add a new JavaScript file into your game, you should change the attribute **jsList** and add more source code file paths to the end of the array.

Lastly, we should display the PlayScene when we click the button in the first MenuScene. Here is the code snippet:

*//this is the callback when the menu is clicked*

onPlay : **function**(){

cc.log("==onplay clicked");

cc.director.runScene(**new** PlayScene());

}

### Coding the PlayScene(PlayScene.js)

Since background layer, animation layer and status layer should be displayed in a different order. We can specify the order explicitly when calling the **addChild** method or we can add them as PlayScene's children in the right order. In this tutorial, we will add them in the right order.

Here is the code snippet of PlayScene:

**var** PlayScene = cc.Scene.extend({

onEnter:**function** () {

**this**.\_super();

*//add three layer in the right order*

**this**.addChild(**new** BackgroundLayer());

**this**.addChild(**new** AnimationLayer());

**this**.addChild(**new** StatusLayer());

}

});

### Coding the BackgroundLayer(BackgroundLayer.js)

Here is our background image: 

Here is the code snippet:

**var** BackgroundLayer = cc.Layer.extend({

ctor:**function** () {

**this**.\_super();

**this**.init();

},

init:**function** () {

**this**.\_super();

**var** winsize = cc.director.getWinSize();

*//create the background image and position it at the center of screen*

**var** centerPos = cc.p(winsize.width / 2, winsize.height / 2);

**var** spriteBG = **new** cc.Sprite(res.PlayBG\_png);

spriteBG.setPosition(centerPos);

**this**.addChild(spriteBG);

}

});

### Coding the AnimationLayer(AnimationLayer.js)

Here is our main character:

In this section, we will run actions on the hero. We will run the **MoveTo** action on the sprite to move the sprite from (80,85) to (300,85) in two seconds.

Here is the code snippet of AnimationLayer:

**var** AnimationLayer = cc.Layer.extend({

ctor:**function** () {

**this**.\_super();

**this**.init();

},

init:**function** () {

**this**.\_super();

*//create the hero sprite*

**var** spriteRunner = **new** cc.Sprite(res.runner\_png);

spriteRunner.attr({x: 80, y: 85});

*//create the move action*

**var** actionTo = **new** cc.MoveTo(2, cc.p(300, 85));

spriteRunner.runAction(**new** cc.Sequence(actionTo));

**this**.addChild(spriteRunner);

}

});

### Coding the StatusLayer(StatusLayer.js)

In this section, we will add two indicators: the coin quantity indicator and the distance indicator. Both indicators are labels in Cocos2d-html5. Labels are very useful to display HUD information to players and the code to create and use labels is very easy thanks to cocos2d framework.

Here is the code snippet we need to set up the layer:

**var** StatusLayer = cc.Layer.extend({

labelCoin:null,

labelMeter:null,

coins:0,

ctor:**function** () {

**this**.\_super();

**this**.init();

},

init:**function** () {

**this**.\_super();

**var** winsize = cc.director.getWinSize();

**this**.labelCoin = **new** cc.LabelTTF("Coins:0", "Helvetica", 20);

**this**.labelCoin.setColor(cc.color(0,0,0));*//black color*

**this**.labelCoin.setPosition(cc.p(70, winsize.height - 20));

**this**.addChild(**this**.labelCoin);

**this**.labelMeter = **new** cc.LabelTTF("0M", "Helvetica", 20);

**this**.labelMeter.setPosition(cc.p(winsize.width - 70, winsize.height - 20));

**this**.addChild(**this**.labelMeter);

}

});

We can use **new cc.LabelTTF** for creating a text label. The first parameter is the displayed texts, the second parameter is the font family and the third parameter is the font size. We can also use the **setColor** function of LabelTTF to set the color of labels. **cc.color(0,0,0)** represents the color black.

## Summary

In this tutorial, we have divided the game scene into different layers. Each layer has it's own logic and responsibility. You can download the entire project [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter4/res/Parkour.zip).

The code and logic are pretty straightforward, so we haven't cover them all in details. If you have any questions or suggestions, let us know and we will do our best to support you.

## Where to go from here

In the next tutorial, you will learn how to run animations on the runner and how to pack small images into a sprite sheet. You will also be introduced to an awesome tools named **TexturePacker**.

* [Cocos2d-JS Tutorial 5: Let the Player Run Animations](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter5/en/index.html)

# Let the Player Run Animations

## Introduction

In the last tutorial, we have moved the player from one point to another. But the movement is a little wired. In this tutorial, I will show you how to run animations on the player. Thus the player's movement will be more realistic.

Before that, I want to show you an awesome tool named TexturePacker.

## Introduction to TexturePacker

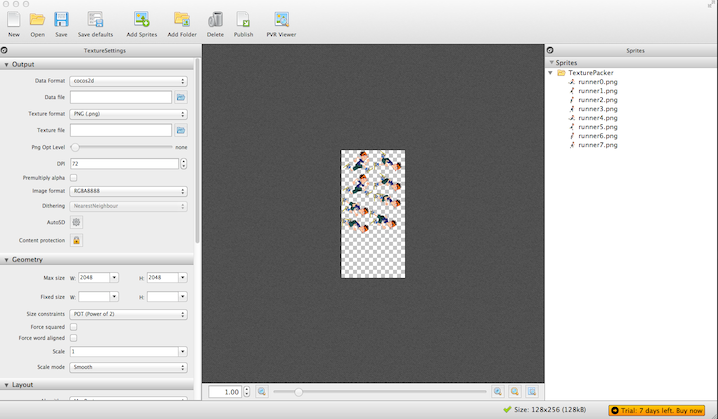
TexturePacker is a cross-platform GUI and command line tool to create sprite sheets! If you want to learn more about TexturePacker, go to [this website](http://www.codeandweb.com/texturepacker/documentation) for more information.

Here I will give you a brief introduction for using TexturePacker to generate the animation files we need in our game.

### Build Your Own Animations with TexturePacker

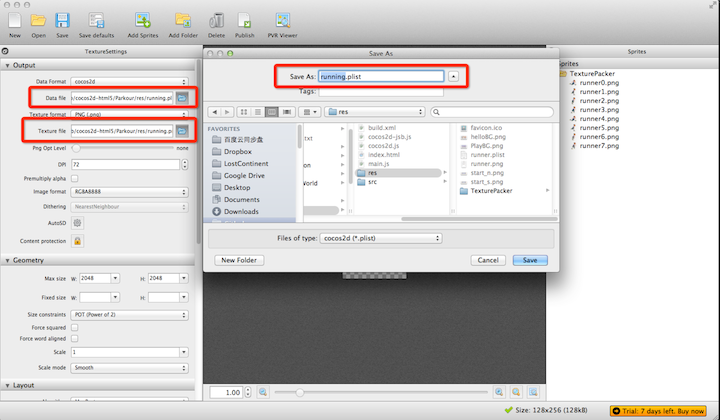
Here is the building process:

1. Open TexturePacker and drag the folder TexturePacker under res/TexturePacker in to TexturePacker's sprite area.

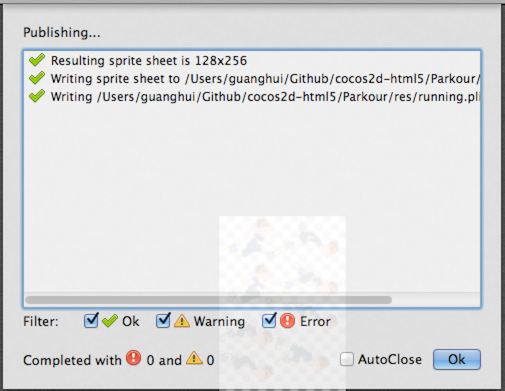


When you add new images into the TexturePacker folder, TexturePacker will automatically detect the changes and load the images.

1. Choose "Data format" to "cocos2d" and "Texture format" to "png" from the Settings panel(which is the default settings).
2. Specify the Data file and Texture file path. Here we set the path at res directory and we set the data file name to "running.plist" and the texture file to "running.png".



1. Click publish. Now a dialog will appear, if there are no errors, then it will generate "running.png" and "running.plist" in the right place.



Ok, we have successfully generate the running animation files. Next let's play with the generated animation files.

## Load Animations Files in Cocos2d-JS

### Preparation

At first, we should add the running.plist and running.png to resource.js file.

var res = {

helloBG\_png : "res/helloBG.png",

start\_n\_png : "res/start\_n.png",

start\_s\_png : "res/start\_s.png",

PlayBG\_png : "res/PlayBG.png",

runner\_png : "res/running.png",

runner\_plist : "res/running.plist"

};

var g\_resources = [

*//image*

res.helloBG\_png,

res.start\_n\_png,

res.start\_s\_png,

res.PlayBG\_png,

res.runner\_png,

res.runner\_plist

];

Here, we have changed the variable runner\_png's value to "running.png" which is a spritesheet. We will create our player sprite from the running.png in the future.

### Create Player Animation

At first, we should add the following member variables in AnimationLayer.js:

spriteSheet:null,

runningAction:null,

sprite:null,

Then we should replace the player creation method to:

**this**.sprite = **new** cc.Sprite("#runner0.png");

The dash ("#") character at the beginning of the sprite's filename is a convention, and that means it will create a sprite from the sprite frame's name. In our case "runner0.png" is part of the sprite sheet.

We can easily create a animation with the following code:

//1.load spritesheet

cc.spriteFrameCache.addSpriteFrames(res.runner\_plist);

//2.create spriteframe array

var animFrames = **[**];

for (var i = 0; i < 8; i++) {

var str = "runner" + i + ".png";

var frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

//3.create a animation with the spriteframe array along with a period time

var animation = new cc.Animation(animFrames, 0.1);

//4.wrap the animate action with a repeat forever action

this.runningAction = new cc.RepeatForever(new cc.Animate(animation));

The animation is constructed from a series of small images(from runner0.png to runner7.png) in the spritesheet.

Here is the completely process to create a animation in Cocos2d-JS:

1. Load spritesheet plist file into SpriteFrameCache class.
2. Add animation frames to the array animFrames
3. Create a cc.Animation object from the animation frame array along with a delay time between each sprite frame.
4. Create the final cc.Animate object and wrap it when a RepeatForever action. Thus the animation will run infinite.

Generally, if we use animations in Cocos2-JS, we always use SpriteBatchNode to boost game performance in WebGL mode or in Cocos2d-x JSB mode.

The final code of the AnimationLayer.js is:

**var** AnimationLayer = cc.Layer.extend({

spriteSheet:null,

runningAction:null,

sprite:null,

ctor:**function** () {

**this**.\_super();

**this**.init();

},

init:**function** () {

**this**.\_super();

*// create sprite sheet*

cc.spriteFrameCache.addSpriteFrames(res.runner\_plist);

**this**.spriteSheet = **new** cc.SpriteBatchNode(res.runner\_png);

**this**.addChild(**this**.spriteSheet);

*// init runningAction*

**var** animFrames = [];

**for** (**var** i = 0; i < 8; i++) {

**var** str = "runner" + i + ".png";

**var** frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

**var** animation = **new** cc.Animation(animFrames, 0.1);

**this**.runningAction = **new** cc.RepeatForever(**new** cc.Animate(animation));

**this**.sprite = **new** cc.Sprite("#runner0.png");

**this**.sprite.attr({x:80, y:85});

**this**.sprite.runAction(**this**.runningAction);

**this**.spriteSheet.addChild(**this**.sprite);

}

});

Now, you can run the project and you will get a infinite running player in your game screen.



## Summary

In this tutorial, we have learned how to use TexturePacker to generate animation and how to run animation on a sprite within Cocos2d-JS.

You can download the entire project from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter5/res/Parkour.zip).

## Where to Go from Here?

In the next tutorial, we will add chipmunk physics into our game world. Thus our game will look more realistic.

* [Cocos2d-JS Tutorial 6: Add Chipmunk Physic Engine to Our Game](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter6/en/index.html)

## Introduction

Cocos2d-JS can give us the power to create impressive game world. But it lacks some sort of realistic. Though we can do complex compute to make the game world more realistic, but there is an another option which can ease our life. The answer is physic engine.

Physic engine provides gravity, collide detection and physic simulation which can make our game world looks more realistic.

In this tutorial, we will introduce Chipmunk physics engine into our parkour game.

## Why Chipmunk Physics?

Why should we choose Chipmunk physic engine? Because it is give us much more power than any other 2D physics engine.

Despite Chipmunk physic engine, there is another option - Box2D.

Box2D is a nice 2D physic engine and it has been existed for a very long time. Many 2d games have been using Box2D for their game physics.

But Chipmunk has it's own advantages. You can go to Chipmunk's [website](http://chipmunk-physics.net/) for more information.

## Enable Chipmunk Physic in Cocos2d-JS

### Preparation

At first, let's enable Chipmunk in Cocos2d-JS.

Open project.json file, and modify:

"modules" : ["cocos2d"],

```

to:

"modules" : ["cocos2d","chipmunk"], ```

Thus, when Cocos2d-JS finish launching, it will load Chipmunk library automatically.

Next, let's create a new file named globals.js and add two global variables into it.

**var** g\_groundHeight = 57;

**var** g\_runnerStartX = 80;

At last, we should tell the framework to load globals.js file when engine launches. Append globals.js path to the end of jsList array:

"jsList" : [

"src/resource.js",

"src/app.js",

"src/AnimationLayer.js",

"src/BackgroundLayer.js",

"src/PlayScene.js",

"src/StatusLayer.js",

"src/globals.js"

]

Note: Whenever you add a new file in Cocos2d-JS, you should remember to add it to the jsList array.

### Initialize Chipmunk physic world

In Chipmunk, there is a space object to represent the physic world.

At first, let's add a new member variable named space in PlayScene.js file:

space:null,

In general, one game needs only a space object. The space object can be shared by different layers. We usually put the space initialize code in PlayScene.

Here is the code to setup the physic world:

*// init space of chipmunk*

initPhysics:**function**() {

*//1. new space object*

**this**.space = **new** cp.Space();

*//2. setup the Gravity*

**this**.space.gravity = cp.v(0, -350);

*// 3. set up Walls*

**var** wallBottom = **new** cp.SegmentShape(**this**.space.staticBody,

cp.v(0, g\_groundHeight),*// start point*

cp.v(4294967295, g\_groundHeight),*// MAX INT:4294967295*

0);*// thickness of wall*

**this**.space.addStaticShape(wallBottom);

},

The above code is self-explanatory so we can safely leave it out. If you want to know the details of these API, you should refer to Chipmunk's official documentation for more information.

Next, let's define our game's main loop:

update:**function** (dt) {

*// chipmunk step*

**this**.space.step(dt);

}

In the update function, we tell Chipmunk start to simulate physics.

Before we go any further, let's add a minor change to AnimationLayer. Since we will create physic actor in AnimationLayer, so we should modify the constructor of AnimationLayer to pass the space object in.

ctor:**function** (space) {

**this**.\_super();

**this**.space = space;

**this**.init();

},

Of course, we should define a weak ref member variable in AnimationLayer and initialize it to null.

Thus our preparation work has been done. let's wrap the end and call these method in onEnter function:

onEnter:**function** () {

**this**.\_super();

**this**.initPhysics();

**this**.addChild(**new** BackgroundLayer());

**this**.addChild(**new** AnimationLayer(**this**.space));

**this**.addChild(**new** StatusLayer());

**this**.scheduleUpdate();

},

**Note** You should initilize the space of physics and pass it to AnimationLayer.

### Add Physics Component to the Runner Sprite

In the last tutorial, we create the runner by using spritsheet. In this section, we will rewrite the runner by using PhysicsSprite.

The PhysicsSprite is a reusable component which can combine the physic body with a cocos2d sprite.

Here is the code to create the runner with PhysicsSprite:

*//1. create PhysicsSprite with a sprite frame name*

**this**.sprite = **new** cc.PhysicsSprite("#runner0.png");

var contentSize = **this**.sprite.getContentSize();

*// 2. init the runner physic body*

**this**.body = **new** cp.Body(1, cp.momentForBox(1, contentSize.width, contentSize.height));

*//3. set the position of the runner*

**this**.body.p = cc.p(g\_runnerStartX, g\_groundHeight + contentSize.height / 2);

*//4. apply impulse to the body*

**this**.body.applyImpulse(cp.v(150, 0), cp.v(0, 0));*//run speed*

*//5. add the created body to space*

**this**.space.addBody(**this**.body);

*//6. create the shape for the body*

**this**.shape = **new** cp.BoxShape(**this**.body, contentSize.width - 14, contentSize.height);

*//7. add shape to space*

**this**.space.addShape(**this**.shape);

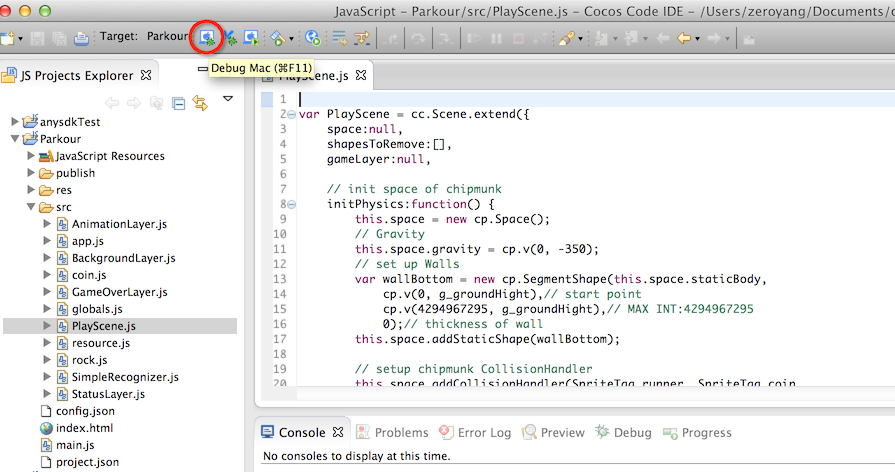
*//8. set body to the physic sprite*

**this**.sprite.setBody(**this**.body);

The code and comment are self-explanatory. Add these code in AnimationLayer's init method.

### Debug and Test

Congratulations. You have done all the bolts and nuts. You can just press the debug button within Webstorm.



Now you can see the runner run pass through the screen.

## Summary

In this tutorial, we have show you how to setup Chipmunk physic world, how to setup the physic world boundary, how to create a rigid body and the associated shape. we also add physics to sprite to make it act more realistic. You can grab the entire source code from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter6/res/Parkour.zip).

## Where to go from here

In the next tutorial, we will introduce the camera movement into the game. And we will also replace the background image with tiledMap. More importantly we will make the background infinite loop displayed in the game. Keep tuning with the next tutorial.

* [Cocos2d-JS Tutorial 7: Explorer with TiledMap and Display](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter7/en/index.html)

## Introduction

In this tutorial, I will show you how to add TiledMap to the parkour game as the new background.

We will also learn how to make the background to scroll infinitely and the player to run infinitely.

The magic behind these is all about moving the cocos2d layer.

## Do Some Preparation Stuff

Before we get our hands dirty, let's add the resource files and the corresponding names to our game.

### Setup Resource and Globals

Since we will need to refer to other layers within each layers. So the best way to retrieve the layer is via **tag**.

Add the following code into globals.js:

**if**(typeof TagOfLayer == "undefined") {

var TagOfLayer = {};

TagOfLayer.background = 0;

TagOfLayer.Animation = 1;

TagOfLayer.Status = 2;

};

Here we give background layer, animation layer and status layer a tag name, thus we can retrieve other layer by tag.

We also need add the resource variables in resource.js:

*//Our two tiled map are named s\_map00 and s\_map01.*

var res = {

helloBG\_png : "res/helloBG.png",

start\_n\_png : "res/start\_n.png",

start\_s\_png : "res/start\_s.png",

PlayBG\_png : "res/PlayBG.png",

runner\_png : "res/running.png",

runner\_plist : "res/running.plist",

map\_png: "res/map.png",

map00\_tmx: "res/map00.tmx",

map01\_tmx: "res/map01.tmx"

};

var g\_resources = [

*//image*

res.helloBG\_png,

res.start\_n\_png,

res.start\_s\_png,

res.PlayBG\_png,

res.runner\_png,

res.runner\_plist,

res.map\_png,

res.map00\_tmx,

res.map01\_tmx

];

The above code is self-explanation so let's go to the next section.

### Enable Chipmunk Debug Drawing

If we are doing Chipmunk physics, you'd better to enable debug drawing. So the debug process will be more handy.

Add the following code into AnimationLayer.js 's ctor function:

**this**.\_debugNode = **new** cc.PhysicsDebugNode(**this**.space);

*// Parallax ratio and offset*

**this**.addChild(**this**.\_debugNode, 10);

When you run the game again, you will see a red box above the running player:



## Introduction to TiledMap

TiledMap is a very common used concepts in 2d games. It is useful for building large level map and some parallax scrolling background.

TiledMap consumes less memory than normal PNG file. If you want to build some huge level map, it is definitely your right choice.

Without further ado, let's dive into the TiledMap.

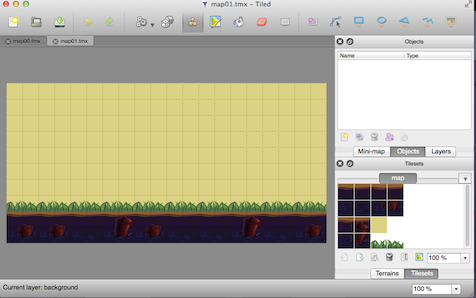
### Design and Make the TiledMap Background

At first, you should download TiledMap. You can download it from [here](http://www.mapeditor.org/download.html). TiledMap is a cross platform software and there are many different kinds of version available. You can choose a version according to your operating system. After downloading the Tiled editor, you should be familiar with it's usage. You may want to take a look at it's [documentation](https://github.com/bjorn/tiled/wiki).

When you feel comfortable with Tiled, you can design your tiled map with the tilesets we provided.

The detail process of making the two tiled map is out of scope of this tutorial.

(Note: If you can't make the tiled map by yourself, you can safely skip the process and use the tiled map provided by us.)



### Replace Previous Background with TiledMap

Now, it's time to replace the old static background image with our new awesome tiled map.

We will do this in BackgroundLayer.js. At first, we should add four member variables in BackgroundLayer class:

map00:null,

map01:null,

mapWidth:0,

mapIndex:0,

The we should delete the old code we needed to create the static background.

(Note: Here I uncomment the code snippets, you can safely delete all of them.)

*// var winSize = cc.Director.getInstance().getWinSize();*

*//*

*// var centerPos = cc.p(winSize.width / 2, winSize.height / 2);*

*// var spriteBG = new cc.Sprite(s\_PlayBG);*

*// spriteBG.setPosition(centerPos);*

*// this.addChild(spriteBG);*

At last, we will add the new code snippets to create the tiled map background.

**this**.map00 = **new** cc.TMXTiledMap(res.map00\_tmx);

**this**.addChild(**this**.map00);

**this**.mapWidth = **this**.map00.getContentSize().width;

**this**.map01 = **new** cc.TMXTiledMap(res.map01\_tmx);

**this**.map01.setPosition(cc.p(**this**.mapWidth, 0));

**this**.addChild(**this**.map01);

Save all the changes and run it:



Here, we add two maps. The map01 is right beside the map00 background. In the later section, We will explain why we should add two maps.

## Introduction to Scene Display

Since the physic body will move right infinitely and the sprite will synchronize it's position with the physic body.

A few seconds later, the player will go outside of the screen, just as it is in the last tutorial.

So we need to move the game layer's x position each frame to make it in a visible range. Here is the code snippets of AnimationLayer.js:

getEyeX:**function** () {

**return** **this**.sprite.getPositionX() - g\_runnerStartX;

},

Here the getEyeX function computes the delta movement of animation layer.

We should move the same delta movement of this.gameLayer which contains background layer and animation layer in opposite direction, so we could call the update method each frame by adding the following code at the end of update method in PlayScene.js:

update:**function** (dt) {

*// chipmunk step*

**this**.space.step(dt);

**var** animationLayer = **this**.gameLayer.getChildByTag(TagOfLayer.Animation);

**var** eyeX = animationLayer.getEyeX();

**this**.gameLayer.setPosition(cc.p(-eyeX,0));

}

### Move the Background Layer

The process to setup the movement of background layer is almost the same as we do in the last section. But we need to do some calculations of the two tiled map.

So let's do it. Add a new member function checkAndReload to BackgroundLayer:

checkAndReload:**function** (eyeX) {

**var** newMapIndex = parseInt(eyeX / **this**.mapWidth);

**if** (**this**.mapIndex == newMapIndex) {

**return** false;

}

**if** (0 == newMapIndex % 2) {

*// change mapSecond*

**this**.map01.setPositionX(**this**.mapWidth \* (newMapIndex + 1));

} **else** {

*// change mapFirst*

**this**.map00.setPositionX(**this**.mapWidth \* (newMapIndex + 1));

}

**this**.mapIndex = newMapIndex;

**return** true;

},

When the eyeX has exceeded the width of the screen, the expression parseInt(eyeX / this.mapWidth) will get a value greater than 0.

We will use the newMapIndex to decide which map need to move and how many pixels need to move.

Then we should call this function in each frame.

update:**function** (dt) {

**var** animationLayer = **this**.getParent().getChildByTag(TagOfLayer.Animation);

**var** eyeX = animationLayer.getEyeX();

**this**.checkAndReload(eyeX);

}

At last, we should call scheduleUpdate at the end of background layer's init method:

**this**.scheduleUpdate();

## Wrap it up

Ok. We should do some last ending work.

Modify the onEnter method of PlayScene to add **tag** of layers, and add background layer and animation layer to game layer:

onEnter:**function** () {

**this**.\_super();

**this**.initPhysics();

**this**.gameLayer = **new** cc.Layer();

*//add Background layer and Animation layer to gameLayer*

**this**.gameLayer.addChild(**new** BackgroundLayer(), 0, TagOfLayer.background);

**this**.gameLayer.addChild(**new** AnimationLayer(**this**.space), 0, TagOfLayer.Animation);

**this**.addChild(**this**.gameLayer);

**this**.addChild(**new** StatusLayer(), 0, TagOfLayer.Status);

**this**.scheduleUpdate();

},

Cheers! You have successfully finish this tutorial. Run and take a look at it.

Note: If you don't want to display the debug drawing information of a chipmunk rigid body. You can safely add the following code right after the creation of the PhysicsDebugNode:

**this**.\_debugNode.setVisible(false);

## Summary

In this tutorial, we have met TiledMap and display. These two concepts are very important ones when you development a physic endless running game.

You can download the entire project from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter7/res/Parkour.zip).

## Where to go from here?

In the next tutorial, we will add coins and obstacles into our game. At that tutorial, we will also learn how to refactor our game code and make it more extensible.

We will also do some cleanup work in PlayScene and encapsulate two class named Coin and Rock.

Keep tuning with the next tutorial and happy coding!

* [Cocos2d-JS Tutorial 8: Add Coin and Obstacles Into Our Game](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter8/en/index.html)

## Introduction

In this tutorial, we will try to add Coin and Obstacles into our parkour game.

After this tutorial, our player should be able to collect the coin when he is running and he will die when he collides with the obstacle.

We will also cover how to design a game level with tiled map editor. Since the game logic is a little bit complex than before, so we will refactor the code before we adding new game components.

## Preparation

Before we start, let's finish the preparation stuff.

### Setup Resource and Globals

Since we will add two more game elements into our parkour game. So we need add some more global integer tags to identify each game items.

Let's add the following code snippets at the end of globals.js:

// collision type for chipmunk

**if**(typeof SpriteTag == "undefined") {

var SpriteTag = {};

SpriteTag.runner = 0;

SpriteTag.coin = 1;

SpriteTag.rock = 2;

};

Here we use 0,1,2 to represent runner,coin and rock.

We also introduce another spritesheet named background.png and background.plist. We have packed the coins and rocks sprites into the spritesheet named background.png.

The details of how to pack these sprites are leave out for the next subsection.

Next, let's copy the resource files into our res directory and add two more variables for further referring.

var res = {

helloBG\_png : "res/helloBG.png",

start\_n\_png : "res/start\_n.png",

start\_s\_png : "res/start\_s.png",

PlayBG\_png : "res/PlayBG.png",

runner\_png : "res/running.png",

runner\_plist : "res/running.plist",

map\_png : "res/map.png",

map00\_tmx : "res/map00.tmx",

map01\_tmx : "res/map01.tmx",

background\_png :"res/background.png",

background\_plist : "res/background.plist"

};

var g\_resources = [

*//image*

res.helloBG\_png,

res.start\_n\_png,

res.start\_s\_png,

res.PlayBG\_png,

res.runner\_png,

res.runner\_plist,

res.map\_png,

res.map00\_tmx,

res.map01\_tmx,

res.background\_png,

res.background\_plist

];

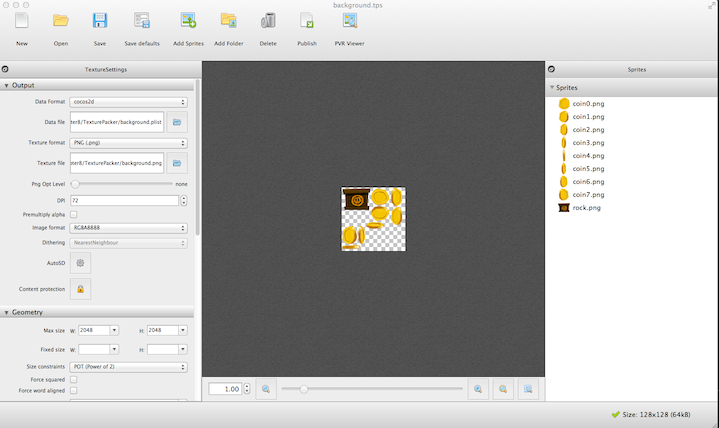
### Pack Coins and Rocks into Spritesheet with TexturePacker

In the previous chapter, we have learned how to pack a bunch of small sprites into a big large compact spritesheet. Let's pack another spritesheet.

At first, you should launch TexturePacker and drag all the assets under res/TexturePacker/coins and rocksdirector.(Note: You can get the whole game resource from the download as before.)

After dragging the resource, you should specify the Data file and Texture format with some path like xxx/chapter8/res/background.png or xxx/chapter8/res/background.plist.

If you don't want to any optimization of the spritesheet, just leave them out and press Publish to generate the final spritesheet.

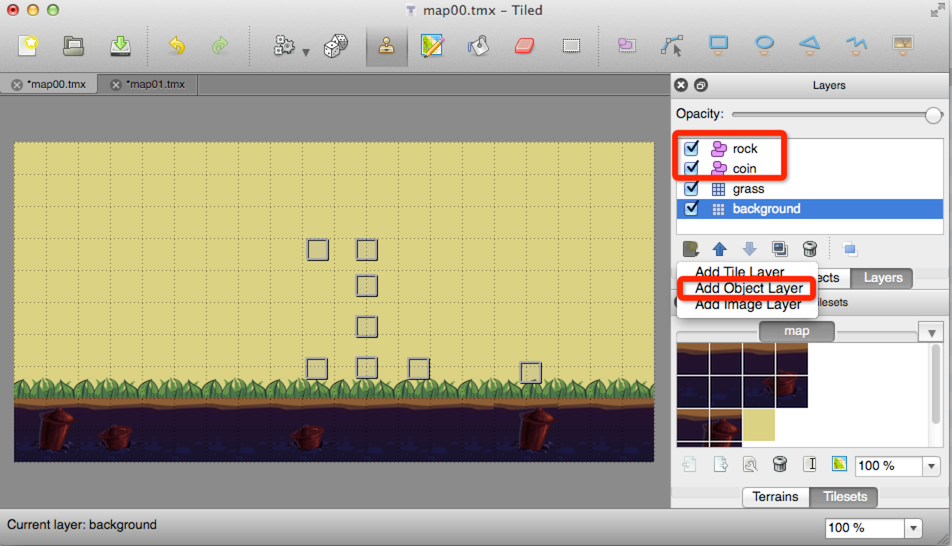


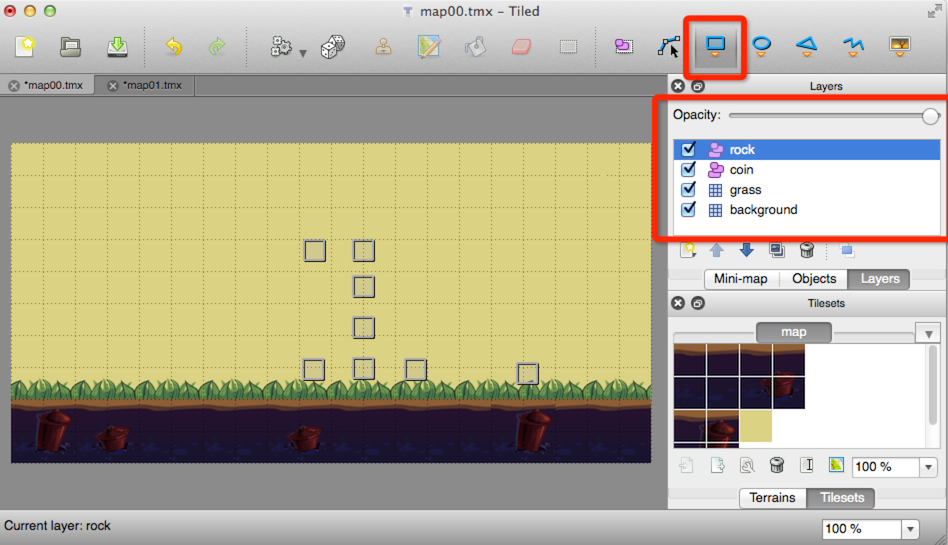
## Introduction to TiledMap Object Layer

We have used TiledMap for our level map, but it lacks game items. So in this section, we will cover how to design level items with TiledMap object layer.

### Add Coin Object Layer

At first, we'll add Coin object layer.

1. Launch Tiled and open map00.tmx and map01.tmx.
2. Create an Object layer named coin in map00.tmx and map01.tmx. 
3. Design object layer by dragging and dropping rectangle object into the map. You can change the rectangle size and it's position. You can also duplicate or delete the objects.



1. Some tips on designing object layer: You can change opacity of the layers in the tiled map so that you can easily place the object.

### Add Rock Object Layer

The process to create the Rock object layer is more or less the same as creating coin object layer.

So we will leave it out for your own implementation.

## Refactor BackgroundLayer Class and Add Some Helper Method

Sometimes, when you are coding, you may find that it is extremely hard to add new functionality into the existing structure.

It is a bad code smell and we should stop and do refactor work right now.

### Refactor BackgroundLayer Class

Since we will add Chipmunk physic body into our background, so we need a method to obtain the space object created in PlayScene.

Let's change the name of ctor function in Background Layer and pass a parameter named space into it. We should also add a new member variable into the BackgroundLayer class. Here is the code snippets:

ctor:function (space) {

this.\_super();

// clean old array here

this.objects = **[**];

this.space = space;

this.init();

},

Here we have added additional init code. We added a array named objects and initialize it to an empty array.

(Note: You should call this.init() method right after the assignment of this.space = space. Because we will create physic objects in the init method)

### Add Helper Method

1. Add more member variables into BackgroundLayer class:

space:null,

spriteSheet:null,

objects:[],

1. Initialize spritesheet in the init method:

*// create sprite sheet*

cc.spriteFrameCache.addSpriteFrames(res.background\_plist);

**this**.spriteSheet = **new** cc.SpriteBatchNode(res.background\_png);

**this**.addChild(**this**.spriteSheet);

1. Add a method named loadObject to initialize rock and coins.

loadObjects:**function** (map, mapIndex) {

*// add coins*

**var** coinGroup = map.getObjectGroup("coin");

**var** coinArray = coinGroup.getObjects();

**for** (**var** i = 0; i < coinArray.length; i++) {

**var** coin = **new** Coin(**this**.spriteSheet,

**this**.space,

cc.p(coinArray[i]["x"] + **this**.mapWidth \* mapIndex,coinArray[i]["y"]));

coin.mapIndex = mapIndex;

**this**.objects.push(coin);

}

*// add rock*

**var** rockGroup = map.getObjectGroup("rock");

**var** rockArray = rockGroup.getObjects();

**for** (**var** i = 0; i < rockArray.length; i++) {

**var** rock = **new** Rock(**this**.spriteSheet,

**this**.space,

rockArray[i]["x"] + **this**.mapWidth \* mapIndex);

rock.mapIndex = mapIndex;

**this**.objects.push(rock);

}

},

Here we iterate all the objects info in the tiled map and create responding Chipmunk rigid bodies. Finally we store these object into the objects array.

All these code are self-explanation. You should only pay attention to the mapIndex parameter. We use the parameter to calculate where we should place the rigid body.

We need call loadObject method at the end of init method to create the physic objects in the first two screen maps.

**this**.loadObjects(**this**.map00, 0);

**this**.loadObjects(**this**.map01, 1);

1. Add another two helper methods for removing unused chipmunk rigid bodies.

The first method is called removeObjects. It removes a object by mapIndex. Here is the implementation:

removeObjects:**function** (mapIndex) {

**while**((**function** (obj, index) {

**for** (**var** i = 0; i < obj.length; i++) {

**if** (obj[i].mapIndex == index) {

obj[i].removeFromParent();

obj.splice(i, 1);

**return** true;

}

}

**return** false;

})(**this**.objects, mapIndex));

},

The other method is called removeObjectByShape:

removeObjectByShape:**function** (shape) {

**for** (**var** i = 0; i < **this**.objects.length; i++) {

**if** (**this**.objects[i].getShape() == shape) {

**this**.objects[i].removeFromParent();

**this**.objects.splice(i, 1);

**break**;

}

}

},

This method will remove a chipmunk object by its shape.

### Wrap it up: Add Creation and Disposable Logic in checkAndReload Method

When the map is moved, we should also call loadObject method to recreate the "Coins & Rocks".

And also we sould remove all unused objects by calling removeObjects method.

Here is the code snippets:

checkAndReload:**function** (eyeX) {

**var** newMapIndex = parseInt(eyeX / **this**.mapWidth);

**if** (**this**.mapIndex == newMapIndex) {

**return** false;

}

**if** (0 == newMapIndex % 2) {

*// change mapSecond*

**this**.map01.setPositionX(**this**.mapWidth \* (newMapIndex + 1));

**this**.loadObjects(**this**.map01, newMapIndex + 1);

} **else** {

*// change mapFirst*

**this**.map00.setPositionX(**this**.mapWidth \* (newMapIndex + 1));

**this**.loadObjects(**this**.map00, newMapIndex + 1);

}

**this**.removeObjects(newMapIndex - 1);

**this**.mapIndex = newMapIndex;

**return** true;

},

## Add Coin and Rock

Now it's time to add the coin and rock implementation. Despite the implementation details, you should also pay attention to the design idea behind these two classes. Here we prefer to inherit from cc.Class instead of cc.Sprite. We let each object to own a instance of cc.Sprite.

### Design and Implement Coin Class

1. Create a new file named coin.js. We will define our Coin class in this file. Make sure you have this filed located in your src directory.
2. Derived a class named Coin from cc.Class, let's take a look at the whole implementation:

**var** Coin = cc.Class.extend({

space:null,

sprite:null,

shape:null,

\_mapIndex:0,*// which map belongs to*

**get** mapIndex() {

**return** **this**.\_mapIndex;

},

**set** mapIndex(index) {

**this**.\_mapIndex = index;

},

*/\*\* Constructor*

*\* @param {cc.SpriteBatchNode \*}*

*\* @param {cp.Space \*}*

*\* @param {cc.p}*

*\*/*

ctor:**function** (spriteSheet, space, pos) {

**this**.space = space;

*// init coin animation*

**var** animFrames = [];

**for** (**var** i = 0; i < 8; i++) {

**var** str = "coin" + i + ".png";

**var** frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

**var** animation = **new** cc.Animation(animFrames, 0.2);

**var** action = **new** cc.RepeatForever(**new** cc.Animate(animation));

**this**.sprite = **new** cc.PhysicsSprite("#coin0.png");

*// init physics*

**var** radius = 0.95 \* **this**.sprite.getContentSize().width / 2;

**var** body = **new** cp.StaticBody();

body.setPos(pos);

**this**.sprite.setBody(body);

**this**.shape = **new** cp.CircleShape(body, radius, cp.vzero);

**this**.shape.setCollisionType(SpriteTag.coin);

*//Sensors only call collision callbacks, and never generate real collisions*

**this**.shape.setSensor(true);

**this**.space.addStaticShape(**this**.shape);

*// add sprite to sprite sheet*

**this**.sprite.runAction(action);

spriteSheet.addChild(**this**.sprite, 1);

},

removeFromParent:**function** () {

**this**.space.removeStaticShape(**this**.shape);

**this**.shape = null;

**this**.sprite.removeFromParent();

**this**.sprite = null;

},

getShape:**function** () {

**return** **this**.shape;

}

});

Let's explain the code piece by piece.

At first, we add three member variables named: space, sprite and shape. We will use these variables to create the coin object's physic body and its display attribution.

Then, we added another member variable \_mapIndex. We used the get/set syntax sugar to define accessor of the variable.

The ctor method is the constructor of Coin class. We will create a Coin class with a spritesheet, a space and a position object later.

Since the coins are circular shape, so we have created CircleShape attached to the rigid body. The remaining part of the ctor function is self-explanation.

At last, we need to define a method to do the cleanup work. It's the removeFromParent method. It firstly remove the rigid body from the space and then remove the sprite from its parent. The getShape method is just a getter method used for accessing the shape object stored in the coin object.

### Design and Implement Rock Class

The principle of designing the Rock class is more or less as the Coin class except for the rigid shape type part.

Because our Rock class is a rectangle box. So we use cp.BoxShape to replace the cc.CircleShape in Coin class.

Here is the full source code of rock.js:

var Rock = cc.Class.extend({

space:**null**,

sprite:**null**,

shape:**null**,

\_map:0,*// which map belong to*

get map() {

**return** **this**.\_map;

},

set map(newMap) {

**this**.\_map = newMap;

},

*/\*\* Constructor*

*\* @param {cc.SpriteBatchNode \*}*

*\* @param {cp.Space \*}*

*\* @param {cc.p}*

*\*/*

ctor:function (spriteSheet, space, posX) {

**this**.space = space;

**this**.sprite = **new** cc.PhysicsSprite("#rock.png");

var body = **new** cp.StaticBody();

body.setPos(cc.p(posX, **this**.sprite.getContentSize().height / 2 + g\_groundHeight));

**this**.sprite.setBody(body);

**this**.shape = **new** cp.BoxShape(body,

**this**.sprite.getContentSize().width,

**this**.sprite.getContentSize().height);

**this**.shape.setCollisionType(SpriteTag.rock);

**this**.space.addStaticShape(**this**.shape);

spriteSheet.addChild(**this**.sprite);

},

removeFromParent:function () {

**this**.space.removeStaticShape(**this**.shape);

**this**.shape = **null**;

**this**.sprite.removeFromParent();

**this**.sprite = **null**;

},

getShape:function () {

**return** **this**.shape;

}

});

## Improve the PlayScene

### Refactor onEnter function of PlayScene

1. At first, let's add a extra array named shapesToRemove and initialize it at the beginning of onEnter function in PlayScene.js.

*//the following line goes in init member variable define area*

shapesToRemove :[],

*//the following line goes at the beginning of the \*onEnter\* function.*

**this**.shapesToRemove = [];

1. Secondly, modify the creation of BackgroundLayer. Here we simply pass the space object into BackgroundLayer's constructor.

**this**.gameLayer.addChild(**new** BackgroundLayer(**this**.space), 0, TagOfLayer.background);

### Add Collision Detection Callbacks

* At first, we should call these two functions at the end of initPhyiscs method:

*// setup chipmunk CollisionHandler*

**this**.space.addCollisionHandler(SpriteTag.runner, SpriteTag.coin,

**this**.collisionCoinBegin.bind(**this**), null, null, null);

**this**.space.addCollisionHandler(SpriteTag.runner, SpriteTag.rock,

**this**.collisionRockBegin.bind(**this**), null, null, null);

The addCollisionHandler method needs a callback when collision occurs.

* Then, let's define these two callbacks to handle player collide with coins and rocks.

collisionCoinBegin:**function** (arbiter, space) {

**var** shapes = arbiter.getShapes();

*// shapes[0] is runner*

**this**.shapesToRemove.push(shapes[1]);

},

collisionRockBegin:**function** (arbiter, space) {

cc.log("==game over");

},

* Delete unused rigid bodies in background layer. You should add the following code at the end of updatemethod:

*// Simulation cpSpaceAddPostStepCallback*

**for**(var i = 0; i < **this**.shapesToRemove.length; i++) {

var shape = **this**.shapesToRemove[i];

**this**.gameLayer.getChildByTag(TagOfLayer.background).removeObjectByShape(shape);

}

**this**.shapesToRemove = [];

We can't delete physic bodies during the physic simulation process. so we use an extra array named shapesToRemove to hold the temporal data needed to be deleted.

## Wrap all these things up

Congratulations! You almost reach the end. Before we hit the debug button to see the results. Let's add some extra glue code to connect everything together.

Open project.json and append two more array items at the end of jsList array.

"jsList" : [

"src/resource.js",

"src/app.js",

"src/AnimationLayer.js",

"src/BackgroundLayer.js",

"src/PlayScene.js",

"src/StatusLayer.js",

"src/globals.js",

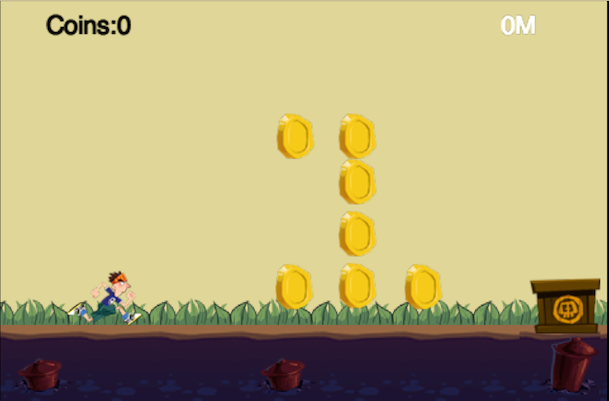
"src/coin.js",

"src/rock.js"

]

Build and run! Cheers, we did it!:)

Let's see our final fruits:



## Summary

In this tutorial, we have enjoyed a very long journey. But worth it, isn't it?

We have learned how to use TiledMap's object layer to design complex game levels and how to customize your own class to extend your code structure.

You can download the full source code from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter8/res/Parkour.zip).

## Where to go from here?

In the next tutorial, we'll cover how to update game HUD constantly and we'll also add game over logic and simple gesture recognizer into our game to make the player to jump over the obstacles. Keep tuning!

* [Cocos2d-JS Tutorial 9: Implement Game Over Logic and More](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter9/en/index.html)

## Introduction

In this tutorial, we'll implement out game over logic, updating HUD and a simple gesture recognizer. With out any further ado, let's dive into the tutorial.

## Update Game HUD

### Update Player Running Meters

At first, let's add a updateMeter method into the StatusLayer Class:

updateMeter:**function** (px) {

**this**.labelMeter.setString(parseInt(px / 10) + "M");

}

This method constantly change the value of labelMeter. Here we use parseInt function to convert the result into a integer value.

The argument px stands for pixel, so every 10 px is one meter.

Now, we should call this method in each frame.

Open AnimationLayer.js and add the following code at the beginning of update function:

// **update** meter

**var** statusLayer = this.getParent().getParent().getChildByTag(TagOfLayer.**Status**);

statusLayer.updateMeter(this.sprite.getPositionX() - g\_runnerStartX);

### Update Coin Count

When the player collect a coin, we should update the coin indicator.

At first, let's add a addCoin method in StatusLayer:

addCoin:**function** (num) {

**this**.coins += num;

**this**.labelCoin.setString("Coins:" + **this**.coins);

},

When the player collide with the coin, we should call this method.

Now it's time to implement it.

Open PlayScene.js and add the following code at the end of colisionCoinBegin method:

**var** statusLayer = **this**.getChildByTag(TagOfLayer.Status);

statusLayer.addCoin(1);

Every time the player collides with the coin, the colisionCoinBegin method will be called and our coin count will be increase by one.

Save all files and git it a try:)

Here is the screenshot:



## Add Game Over Logic into the Game

### Design and Implement the Game Over Layer

In order to keep things simple, we just add a menu item at the center of our game over layer.

When you click the restart menu, the game will restart again.

So the design is very trivial, let's implement it.

Here is the whole implementation of GameOverLayer.js:

**var** GameOverLayer = cc.LayerColor.extend({

*// constructor*

ctor:**function** () {

**this**.\_super();

**this**.init();

},

init:**function** () {

**this**.\_super(cc.color(0, 0, 0, 180));

**var** winSize = cc.director.getWinSize();

**var** centerPos = cc.p(winSize.width / 2, winSize.height / 2);

cc.MenuItemFont.setFontSize(30);

**var** menuItemRestart = **new** cc.MenuItemSprite(

**new** cc.Sprite(res.restart\_n\_png),

**new** cc.Sprite(res.restart\_s\_png),

**this**.onRestart, **this**);

**var** menu = **new** cc.Menu(menuItemRestart);

menu.setPosition(centerPos);

**this**.addChild(menu);

},

onRestart:**function** (sender) {

cc.director.resume();

cc.director.runScene(**new** PlayScene());

}

});

Here we used two sprite named s\_restart\_n and s\_restart\_s to create our restart menu item.

So we should add the resource file into the res directory and define the resources path.

Open resource.js and add the following code lines:

restart\_n\_png : "res/restart\_n.png",

restart\_s\_png : "res/restart\_s.png"

*//add the following two lines at the end of g\_resources array.*

res.restart\_n\_png,

res.restart\_s\_png

The code of init method is self-explanation. But you should pay attention to the callback onRestart method.

We have called the resume function of cc.Director. Why should be do this? Because we call pause method when the player die.

### Game Over When The Player Collide with a Rock

Now, let's display the game over layer when the player collide with the rock.

Open PlayScene and add the following code lines at the end of collisionRockBegin method:

collisionRockBegin:**function** (arbiter, space) {

cc.log("==game over");

cc.director.pause();

**this**.addChild(**new** GameOverLayer());

},

Yeah, it's done. Now run the game again.

Here is the final screenshot:



## Implement Your Own Simple Gesture Recognizer

At this section, we will design a very simple gesture recognizer.

When we swipe our finger on the screen from bottom to up, the recognizer will detect it.

The algorithm for detecting the swipe gesture is very straightforward. When the touch begin event is detected, we store the first touch point in the array. When the touch moves event is detected, we append the touch point at the end of point array.

We can simply compare the difference of x axis or y axis of two adjacent point to determine the swipe direction.

Here is the code snippets to do the magic:

function **Point**(x, y)

{

this.X = x;

this.Y = y;

}

// class define

function **SimpleRecognizer**()

{

this.points = [];

this.result = "";

}

// be called **in** onTouchBegan

**SimpleRecognizer**.prototype.beginPoint = function(x, y) {

this.points = [];

this.result = "";

this.points.push(new **Point**(x, y));

}

// be called **in** onTouchMoved

**SimpleRecognizer**.prototype.movePoint = function(x, y) {

this.points.push(new **Point**(x, y));

**if** (this.result == "not support") {

**return**;

}

**var** newRtn = "";

**var** len = this.points.length;

**var** dx = this.points[len - 1].X - this.points[len - 2].X;

**var** dy = this.points[len - 1].Y - this.points[len - 2].Y;

**if** (**Math**.abs(dx) > **Math**.abs(dy)) {

**if** (dx > 0) {

newRtn = "right";

} **else** {

newRtn = "left";

}

} **else** {

**if** (dy > 0) {

newRtn = "up";

} **else** {

newRtn = "down";

}

}

// first **set** result

**if** (this.result == "") {

this.result = newRtn;

**return**;

}

// **if** diretcory change, **not** support **Recongnizer**

**if** (this.result != newRtn) {

this.result = "not support";

}

}

// be called **in** onTouchEnded

**SimpleRecognizer**.prototype.endPoint = function() {

**if** (this.points.length < 3) {

**return** "error";

}

**return** this.result;

}

**SimpleRecognizer**.prototype.getPoints = function() {

**return** this.points;

}

After the gesture is detected, we can call endPoint of the SimpleRecognizer to get the final result.

Currently it supports four simple types: up, down, left and right. You can extend your own more complex one.

## Handling Touch Event, Player Jumping Animation and Logic

### Add Jumping Animation of the Player

In order to implement the jumping animation, we should prepare our game arts first. Here we have done it for you.

You can download the whole project from the Summary section and copy&paste the running.plist and running.png file into the res directory.

When the game starts, the player will be running infinite until him collide with the rock. We want to let the player jump by swiping upwards.

Thus we can play the game a little bit longer.

When the player jumps up or jumps down, we need to play corresponding animations.

So at first, let's add two more animation action member variables into the AnimationLayer:

jumpUpAction:null,

jumpDownAction:null,

And then let's add a new method named initAction:

initAction:**function** () {

*// init runningAction*

**var** animFrames = [];

*// num equal to spriteSheet*

**for** (**var** i = 0; i < 8; i++) {

**var** str = "runner" + i + ".png";

**var** frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

**var** animation = **new** cc.Animation(animFrames, 0.1);

**this**.runningAction = **new** cc.RepeatForever(**new** cc.Animate(animation));

**this**.runningAction.retain();

*// init jumpUpAction*

animFrames = [];

**for** (**var** i = 0; i < 4; i++) {

**var** str = "runnerJumpUp" + i + ".png";

**var** frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

animation = **new** cc.Animation(animFrames, 0.2);

**this**.jumpUpAction = **new** cc.Animate(animation);

**this**.jumpUpAction.retain();

*// init jumpDownAction*

animFrames = [];

**for** (**var** i = 0; i < 2; i++) {

**var** str = "runnerJumpDown" + i + ".png";

**var** frame = cc.spriteFrameCache.getSpriteFrame(str);

animFrames.push(frame);

}

animation = **new** cc.Animation(animFrames, 0.3);

**this**.jumpDownAction = **new** cc.Animate(animation);

**this**.jumpDownAction.retain();

},

In this function, we have initialized all the animations of the players.

At lasts, let's remove the initialize code of runningAction we did before in the init function and call initActionmethod instead.

//init actions

**this**.initAction();

// // init runningAction

// var animFrames = [];

// **for** (var i = 0; i < 8; i++) {

// var str = "runner" + i + ".png";

// var frame = cc.spriteFrameCache.getSpriteFrame(str);

// animFrames.push(frame);

// }

// var animation = **new** cc.Animation(animFrames, 0.1);

// **this**.runningAction = **new** cc.RepeatForever(**new** cc.Animate(animation));

### Handling Touch Event

Now it's time to handle touch event. At first, we should enable touch handling of the AnimationLayer.

Add the following code snippets at the end of init method:

cc.eventManager.addListener({

event: cc.EventListener.TOUCH\_ONE\_BY\_ONE,

swallowTouches: true,

onTouchBegan: this.onTouchBegan,

onTouchMoved: this.onTouchMoved,

onTouchEnded: this.onTouchEnded

}, this)

These two code lines can activate the touch dispatching function.

Now let's add three callbacks we needed to handle our touch event:

onTouchBegan:**function**(touch, event) {

**var** pos = touch.getLocation();

event.getCurrentTarget().recognizer.beginPoint(pos.x, pos.y);

**return** true;

},

onTouchMoved:**function**(touch, event) {

**var** pos = touch.getLocation();

event.getCurrentTarget().recognizer.movePoint(pos.x, pos.y);

},

onTouchEnded:**function**(touch, event) {

**var** rtn = event.getCurrentTarget().recognizer.endPoint();

cc.log("rnt = " + rtn);

**switch** (rtn) {

**case** "up":

event.getCurrentTarget().jump();

**break**;

**default**:

**break**;

}

},

When you touch the screen the onTouchBegan method will be called. When you hold your finger and move it around, the onTouchMoved method will be called. When you release your finger, the onTouchEnded method will be called.

Here we have used our simple gesture recognizer to detect the "swipe out" gesture.

### Wrap them all

Now it's time to wrap them all.

At first, add the following enumerations in the beginning of the AnimationLayer:

// define enum for runner status

**if**(typeof RunnerStat == "undefined") {

var RunnerStat = {};

RunnerStat.running = 0;

RunnerStat.jumpUp = 1;

RunnerStat.jumpDown = 2;

};

We use these enums to represent different state of the player.

Then we should add another two member variables in the AnimationLayer:

recognizer:null,

stat:RunnerStat.running,*// init with running status*

And init the recognizer at the end of init method:

**this**.recognizer = **new** SimpleRecognizer();

At last, we should implement our jump method:

jump:**function** () {

cc.log("jump");

**if** (**this**.stat == RunnerStat.running) {

**this**.body.applyImpulse(cp.v(0, 250), cp.v(0, 0));

**this**.stat = RunnerStat.jumpUp;

**this**.sprite.stopAllActions();

**this**.sprite.runAction(**this**.jumpUpAction);

}

},

And also we should tie these things up int the update function:

*//in the update method of AnimationLayer*

*// check and update runner stat*

**var** vel = **this**.body.getVel();

**if** (**this**.stat == RunnerStat.jumpUp) {

**if** (vel.y < 0.1) {

**this**.stat = RunnerStat.jumpDown;

**this**.sprite.stopAllActions();

**this**.sprite.runAction(**this**.jumpDownAction);

}

} **else** **if** (**this**.stat == RunnerStat.jumpDown) {

**if** (vel.y == 0) {

**this**.stat = RunnerStat.running;

**this**.sprite.stopAllActions();

**this**.sprite.runAction(**this**.runningAction);

}

}

One more word, don't forget the cleanup stuff. We should release the created actions when the AnimationLayer exit.

onExit:**function**() {

**this**.runningAction.release();

**this**.jumpUpAction.release();

**this**.jumpDownAction.release();

**this**.\_super();

},

You may also want to check whether all the created js files are loaded in cocos2d.js file or not.

## Summary

Congratulations! You have finished another epic tutorial again.

Let's see what we have done in this tutorial.

At first, we have learned how to update our game hud elements.

Then we added the game over logic.

At last, we have created a simple gesture recognizer to handle the controlling of our hero's jumping action.

You can download the final project from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter9/res/Parkour.zip).

## Where to go from here?

In the next tutorial, we will cover the final bolts and nuts of the parkour game. Keep tuning!

* [Cocos2d-JS Tutorial 10: Add Audio Effect into Parkour Game](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter10/en/index.html)

## Introduction

In this tutorial, I want to show you how to add audio support into our Parkour game.

## Time in Action

### Audio Formats Supported by HTML5

At first, let's see which formats are supported by HTML5: 

Here we use ogg format. Because it is well supported by Android and Chrome.

### Add Audio Support into Our Game

#### Copy Audio Resources

At first, let's copy required audio files into our res directory. Audio files can be found in the sample project which can be downloaded from the Summary section.

We use background.ogg for background music, jump.ogg and pickup\_coin.ogg for jumping and collecting the coins.

Secondly, let's define some global variables in the resource.js file:

var res = {

*// Append to the list*

background\_mp3 : "res/background.mp3",

jump\_mp3 : "res/jump.mp3",

pickup\_coin\_mp3 : "res/pickup\_coin.mp3"

};

var g\_resources = [

*// Append to the list*

res.background\_mp3,

res.jump\_mp3,

res.pickup\_coin\_mp3

];

**Note:** It is recommended to have the same audio file in both "mp3" and "ogg" format at the same time to ensure compatibility across all browsers. And you don't need to list both in resource.js, please just list "mp3"format which has better compatibility with iOS and Mac when you run your game on devices.

#### Add Audio Handling Code into The Game

At first, add the background music when entering the PlayScene:

Note: Add the following code at the end of onEnter function in PlayScene.js.

*//add background music*

cc.audioEngine.playMusic(res.background\_mp3, true);

**this**.scheduleUpdate();

Then, when the game is over, we should stop the background music. So add the following code snippets in collosionRockBegin method:

//**stop** bg music

cc.audioEngine.stopMusic();

At last, let's add jumping audio effect and collecting coin audio effect.

//add the jumping audio effect **in** \*jump\* **method** **of** **AnimationLayer**

//**Jump** music

cc.audioEngine.playEffect(res.jump\_mp3);

//add the collect coin audio effect **in** \*collisionCoinBegin\* **method** **of** **PlayScene**

cc.audioEngine.playEffect(res.pickup\_coin\_mp3);

Save all the changes and run the game. Now your game rock with a world full of audios.

## Summary

In the tutorial, we have seen that how easy it is to add audio support into an existing game. You can download the whole project from [here](http://cocos2d-x.org/docs/tutorials/javascript/parkour-game-with-javascript-v3.0/chapter10/res/Parkour.zip).

## Where to go from here?

Now the journey of Parkour game has reached a milestone. In the following tutorials, we will add cross-platform support and do some encryption to our js files.

Keep tuning!