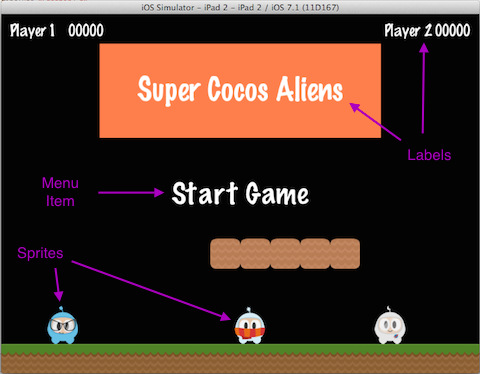
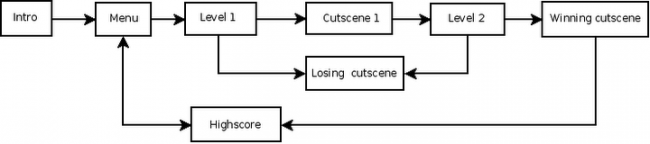
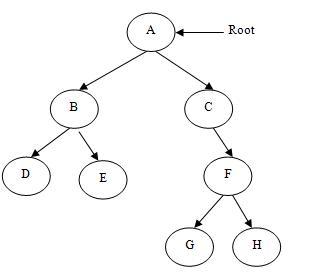
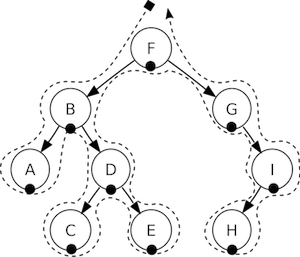
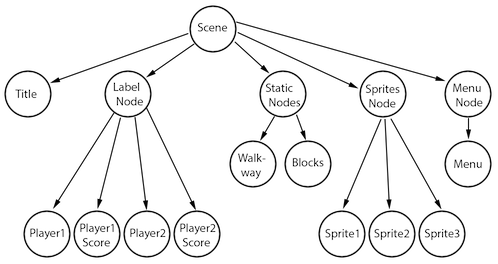
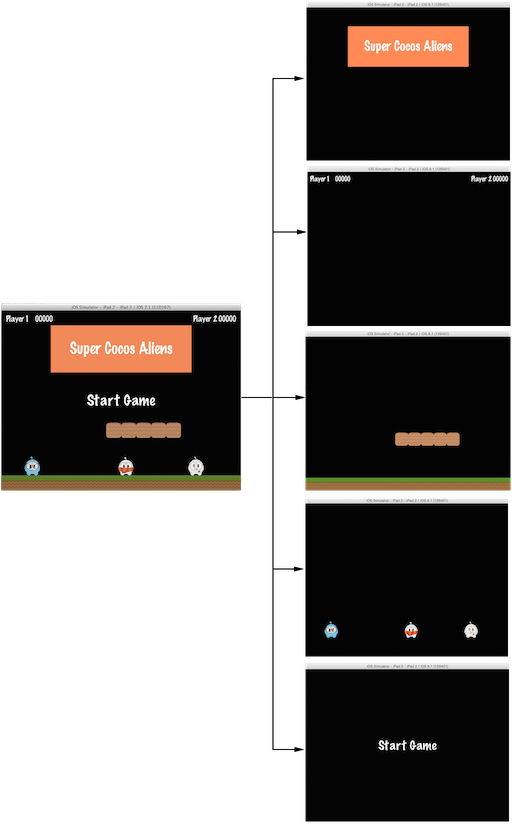
* A vast API of functionality including sprites, actions, animations, particles, transitions, timers, events (touch, keyboard, accelerometer, mouse), sound, file IO, persistence, skeletal animations, 3D

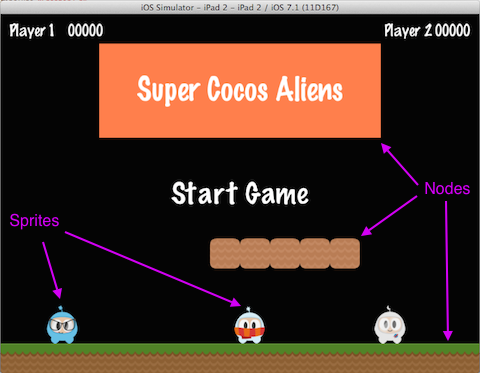
<http://cocos2d-x.org/docs/programmers-guide/basic_concepts/>

* [**Basic Cocos2d-x Concepts**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#basic-cocos2d-x-concepts)
* [**Main Components**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#main-components)
* [**Director**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#director)
* [**Scene**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#scene)
* [**Scene Graph**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#scene-graph)
* [**Sprites**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#sprites)
* [**Actions**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#actions)
* [**Sequences and Spawns**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#sequences-and-spawns)
* [**Parent Child Relationship**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#parent-child-relationship)
* [**Logging as a way to output messages**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#logging-as-a-way-to-output-messages)
* [**Conclusion**](http://cocos2d-x.org/docs/programmers-guide/basic_concepts/#conclusion)
* This chapter assumes you've just gotten started with Cocos2d-x, and are ready to start working on the game of your dreams. Don't worry, it will be fun!
* Let's get started!
* Cocos2d-x is a cross-platform game engine. A game engine is a piece of software that provides common functionality that all games need. You might have heard this referred to as an API or framework but in this guide, we'll be calling it a 'game engine'.
* Game engines include many components that when used together will help speed up development time, and often perform better than homemade engines. A game engine is usually comprised of some or all of the following components: a renderer, 2d/3d graphics, collision detection, a physics engine, sound, controller support, animations and more. Game engines usually support multiple platforms thus making it easy to develop your game and then deploy it to multiple platforms without much overhead at all.
* Since Cocos2d-x is a game engine, it provides a simplified API for developing cross-platform mobile and desktop games. By encapsulating the power inside an easy to use API, you can focus on developing your games and worry less about the implementation of the technical underpinnings. Cocos2d-x will take care of as much or as little of the heavy lifting as you want.
* Cocos2d-x provides Scene, Transition, Sprite, Menu, Sprite3D, Audio objects and much more. Everything you need to create your games are included.
* **Main Components**
* It might seem overwhelming at first, but getting started with Cocos2d-x is simple. Before we dive into depth we must understand some of the concepts Cocos2d-x utilizes. At the heart of Cocos2d-x are Scene, Node, Sprite, Menu and Action objects. Look at any of your favorite games, and you will see all of these components in one form or another!
* Let's have a look. This might look a bit similar to a very popular game you might have played:
* 
* Let's take another look, but splitting up the screenshot and identifying the components used to build it:
* 
* You can see a menu, some sprites and labels, which all have an equivalent in Cocos2d-x. Take a look at a few of your own game design documents,and see what components you have, you'll probably have a few that match up.
* **Director**
* Cocos2d-x uses the concept of a Director, just like in a movie! The Director controls the flow of operations and tells the necessary recipient what to do. Think of yourself as the *Executive Producer* and you tell the Director what to do! One common Director task is to control Scene replacements and transitions. The Director is a shared singleton (effectively, there's only one instance of the class at a time) object that you can call from anywhere in your code.
* Here is an example of a typical game flow. The Director takes care of transitioning through this as your game criteria decides:
* 
* You are the director of your game. You decide what happens, when and how. Take charge!
* **Scene**
* In your game you probably want a main menu, a few levels and an ending scene. How do you organize all of these into the separate pieces they are? You guessed it, Scene. When you think about your favorite movie you can see that it's distinctly broken down into scenes, or separate parts of the story line. If we apply this same thought process to games, we should come up with at least a few scenes no matter how simple the game is.
* Taking another look at the familiar image from earlier:
* 
* This is a main menu and it is a single Scene. This scene is made up of several pieces that all fit together to give us the end result. Scenes are drawn by the **renderer**. The **renderer** is responsible for rendering sprites and other objects into the screen. To better understand this we need to talk a bit about the **scene graph**.
* **Scene Graph**
* A **scene graph** is a data structure that arranges a graphical scene. A **scene graph** contains Node objects in a tree (yes, it is called **scene graph**, but it is actually represented by a **tree**) structure.
* 
* It sounds and looks complicated. I'm sure you are asking why should you care about this technical detail if Cocos2d-x does the heavy lifting for you? It really is important to understand how Scenes are drawn by the renderer.
* Once you start adding nodes, sprites and animations to your game, you want to make sure you are drawing the things you expect. But what if you are not? What if your sprites are hidden in the background and you want them to be the foremost objects? No big deal, just take a step back and run through the scene graph on a piece of paper, and I bet you find your mistake easily.
* Since the *Scene Graph* is a tree; you can **walk the tree**. Cocos2d-x uses the **in-order walk** algorithm. An **in-order walk** is the left side of the tree being walked, then the root node, then the right side of the tree. Since the right side of the tree is rendered last, it is displayed first on the **scene graph**.
* 
* The **scene graph** is easily demonstrated, let's take a look at our game scene broken down:
* 
* Would be rendered as a tree, simplified to the following:
* 
* Another point to think about is elements with a negative **z-order** are on the left side of the tree, while elements with a positive **z-order** are on the right side. Keep this in consideration when ordering your elements! Of course, you can add elements in any order, and they're automatically sorted based upon a customizable **z-order**.
* Building on this concept, we can think of a Scene as a collection of Node objects. Let's break the scene above down to see the **scene graph** uses the **z-order** to layout the Scene:
* 
* The Scene on the left is actually made up of multiple Node objects that are given a different **z-order** to make them "stack" on top of each other.
* In Cocos2d-x, you build the **scene graph** using the *addChild()* API call:
* *// Adds a child with the z-order of -2, that means*
* *// it goes to the "left" side of the tree (because it is negative)*
* scene.addChild(title\_node, -2);
* *// When you don't specify the z-order, it will use 0*
* scene.addChild(label\_node);
* *// Adds a child with the z-order of 1, that means*
* *// it goes to the "right" side of the tree (because it is positive)*
* scene.addChild(sprite\_node, 1);

## Sprites

All games have Sprite objects, and you may or may not realize what they are. Sprites are the objects that you move around the screen. You can manipulate them. The main character in your game is probably a Sprite. I know what you might be thinking - isn't every graphical object a Sprite? No! Why? Well a Sprite is only a Sprite if you move it around. If you don't move it around it is just a Node.

Taking another look at the image from above, let's point out what are Sprites and what are Nodes:

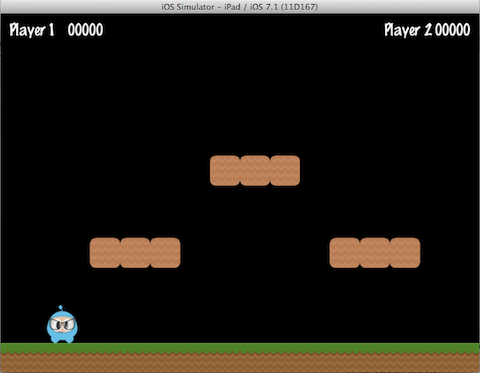


Sprites are important in all games. Writing a platformer, you probably have a main character that is made by using an image of some sort. This is a Sprite.

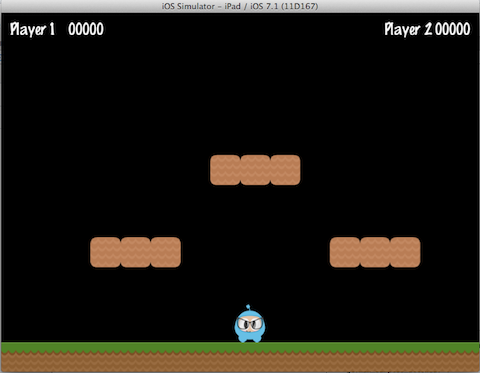
Sprites are easy to create and they have configurable properties like: **position**, **rotation**, **scale**, **opacity**, **color** and more.

*// This is how to create a sprite* **var** mySprite = **new** cc.Sprite(res.mySprite\_png); *// this is how to change the properties of the sprite* mySprite.setPosition(cc.\_p(500, 0)); mySprite.setRotation(40); mySprite.setScale(2.0); *// sets both the scale of the X and Y axis uniformly* mySprite.setAnchorPoint(cc.\_p(0, 0));

Let's illustrate each property, consider the following screenshot from the example code for this chapter:

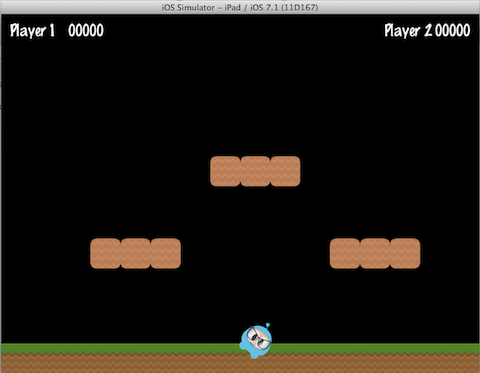


If we set the position using mySprite->setPosition(Vec2(500, 0));:



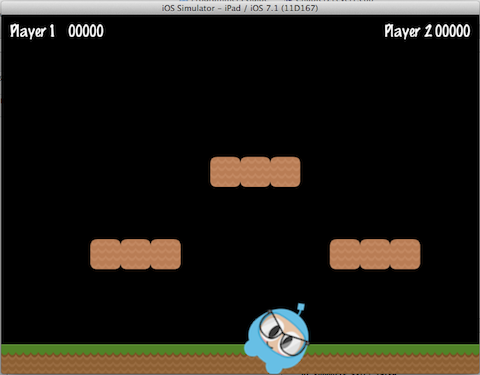
Note that the Sprite position has changed from its original position to the new position that we specified.

If we now set a new rotation, using mySprite->setRotation(40);:



... you can see that the Sprite has been rotated to the new amount that was specified.

If we now specify a new scale using mySprite->setScale(2.0);:



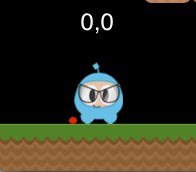
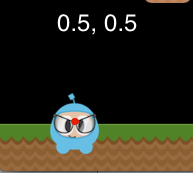
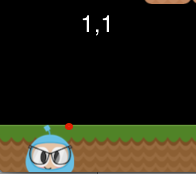
Again, we can see that the Sprite now has changed according to our code changes.

Lastly, all Node objects (since a Sprite is a subclass of Node) have a value for **anchor point**. We haven't talked about this yet, so now is a good time. You can think of **anchor point** as a way of specifying what part of the sprite will be used as a base coordinate when setting the position of it.

Using the character from our example game, and setting the anchor point to **0, 0** using:

mySprite.setAnchorPoint(cc.\_p(0, 0));

would result in the lower left corner of our sprite being used as the basis for any **setPosition()** call. Let's see a few of these in action:

 http://cocos2d-x.org/docs/programmers-guide/basic_concepts-img/smallSpacer.png  http://cocos2d-x.org/docs/programmers-guide/basic_concepts-img/smallSpacer.png 

Take a look at the red dot in each picture. This red dot illustrates where the anchor point is!

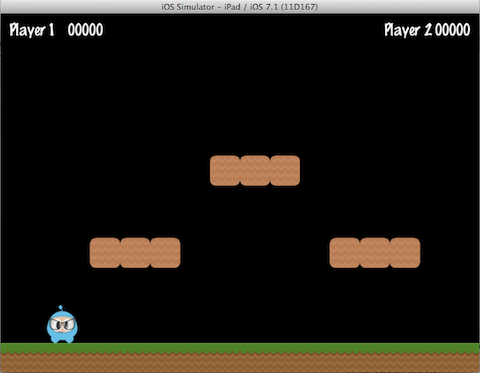
As you can see **anchor point** is very useful when positioning Nodes. You can even adjust the **anchor point**dynamically to simulate effects in your game.

We really can tweak just about every aspect of the Sprite. But, what if we wanted to have these same types of changes occur in an automated, time determined manner? Well, keep reading...

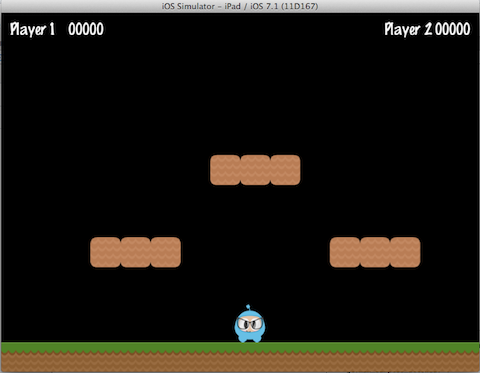
## Actions

Creating a Scene and adding Sprite objects on the screen is only part of what we need to do. For a game to be a game we need to make things move around! Action objects are an integral part of every game. **Actions**allow the transformation of Node objects in time space. Want to move a Sprite from one Point to another and use a callback when complete? No problem! You can even create a Sequence of Action items to be performed on a Node. You can change Node properties like position, rotation and scale. Example Actions: MoveBy, Rotate, Scale. All games use **Actions**.

Taking a look at the [sample code](https://github.com/chukong/programmers-guide-samples) for this chapter, here are **Actions** in work:



and after 5 seconds the sprite will move to a new position:



Action objects are easy to create:

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

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

**var** moveBy = **new** cc.MoveBy(2, cc.\_p(50,10));

mySprite.runAction(moveBy);

*// Move a sprite to a specific location over 2 seconds.*

**var** moveTo = **new** cc.MoveTo(2, cc.\_p(50,10));

mySprite.runAction(moveTo);

## Sequences and Spawns

With moving Sprite objects on the screen we have everything we need to create our game, right? Not quite. What about running multiple **Actions**? Yes, Cocos2d-x handles this too in a few different ways.

Just like it sounds, a Sequence is multiple Action objects run in a specified order. Need to run the Sequencein reverse? No problem, Cocos2d-x handles this with no additional work.

Take a look at the flow of an example Sequence for moving a Sprite gradually:

http://cocos2d-x.org/docs/programmers-guide/basic_concepts-img/2_sequence_scaled.png

This Sequence is easy to make:

**var** mySprite = **new** cc.Node();

*// move to point 50,10 over 2 seconds*

**var** moveTo1 = **new** cc.MoveTo(2, cc.\_p(50,10));

*// move from current position by 100,10 over 2 seconds*

**var** moveBy1 = **new** cc.MoveBy(2, cc.\_p(100,10));

*// move to point 150,10 over 2 seconds*

**var** moveTo2 = **new** cc.MoveTo(2, cc.\_p(150,10));

*// create a delay*

**var** delay = **new** cc.DelayTime(1);

mySprite.runAction(Sequence.create(moveTo1, delay, moveBy1, delay.clone(),

moveTo2));

This example runs a Sequence, in order, but what about running all the specified **Actions** at the same time? Cocos2d-x supports this too and it is called Spawn. Spawn will take all the specified Action objects and executes them at the same time. Some might be longer than others, so they won't all finish at the same time if this is the case.

**var** myNode = **new** cc.Node();

**var** moveTo1 = **new** cc.MoveTo(2, cc.\_p(50,10));

**var** moveBy1 = **new** cc.MoveBy(2, cc.\_p(100,10));

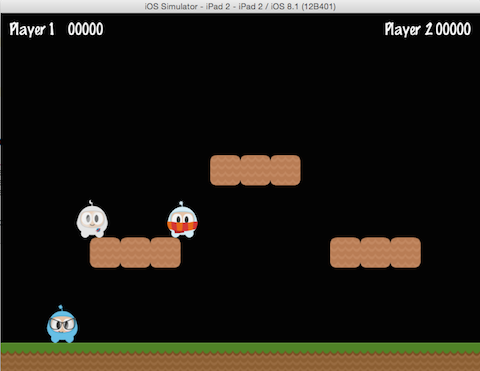
**var** moveTo2 = **new** cc.MoveTo(2, cc.\_p(150,10));

myNode.runAction(Spawn.create(moveTo1, moveBy1, moveTo2));

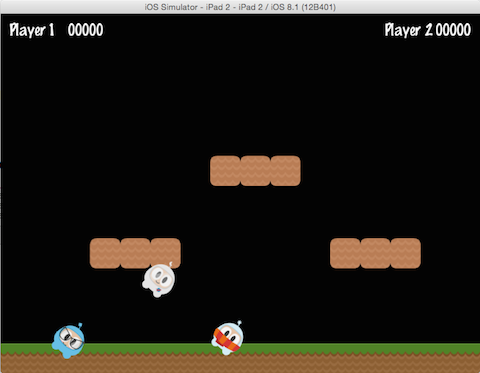
Why Spawn actions? Is there ever a reason? Sure! What if your main character has multiple **Actions** when obtaining a power up? Maybe beating the boss at the end of a level has multiple **Actions** that need to happen to end the level.

## Parent Child Relationship

Cocos2d-x uses a **parent and child** relationship. This means that properties and changes to the parent node are applied to its children. Consider a single Sprite and then a Sprite that has children:



With children, changing the rotation of the parent will also change the rotation to all children:



**var** myNode = **new** cc.Node(); *// rotating by setting*

myNode.setRotation(50);

Just like with rotation, if you change the scale of the parent the children will also get scaled:



**var** myNode = **new** cc.Node();

*// scaling by setting*

myNode.setScale(2.0); *// scales uniformly by 2.0*

Not all changes to the **parent** are passed down to its **children**. Changing the **parent** **anchor point** only affects transform operations (scale, position, rotate, skew, etc...) and does not affect children positioning. In fact, children will be always added to the bottom-left (0,0) corner of its parent.

## Logging as a way to output messages

Sometimes, when your app is running, you might wish to see messages being written to the console for informational or debug purposes. This is built into the engine, using **log()**. Example:

*// a simple string*

cc.log("This would be outputted to the console");

*// outputting more than a simple string*

**var** pos = cc.\_p(sender.x, sender.y);

cc.log("Position x: " + pos.x + ' y:' + pos.y);

## Conclusion

We have gone through a lot of Cocos2d-x concepts. Take a deep breath. Don't worry. Just dive in with your ideas and take it one step at a time. Cocos2d-x and programming in general are not skills that are learned overnight. These take practice and understanding. Remember that the [forums](http://discuss.cocos2d-x.org/) are also there to help you with questions.

<http://cocos2d-x.org/docs/programmers-guide/sprites/index.html>

* [**Sprites**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#sprites)
* [**What are Sprites**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#what-are-sprites)
* [**Creating Sprites**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#creating-sprites)
* [**Creating a Sprite from a Sprite Sheet**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#creating-a-sprite-from-a-sprite-sheet)
* [**Sprite Manipulation**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#sprite-manipulation)
* [**Polygon Sprite**](http://cocos2d-x.org/docs/programmers-guide/sprites/index.html#polygon-sprite)

# Sprites

## What are Sprites

A Sprite is a 2D image that can be animated or transformed by changing its properties, including **rotation**, **position**, **scale**, **color**, etc.

## Creating Sprites

There are different ways to create Sprites depending upon what you need to accomplish. You can create a Sprite from an image with various graphic formats including: **PNG**, **JPEG**, **TIFF**, and others. Let's go through some create methods and talk about each one.

### Creating a Sprite

A Sprite can be created by specifying an image file to use.

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



The statement above creates a Sprite using the **mysprite.png** image. The result is that the created Spriteuses the whole image. Sprite has the same dimensions of **mysprite.png**. If the image file is 200 x 200 the resulting Sprite is 200 x 200.

### Creating a Sprite with a Rect

In the previous example, the created Sprite has the same size as the original image file. If you want to create a Sprite with only a certain portion of the image file, you can do it by specifying a Rect.

Rect has 4 values: **origin x**, **origin y**, **width** and **height**.

**var** mySprite = **new** cc.Sprite(res.mySprite\_png, cc.rect(0,0,40,40));



Rect starts at the top left corner. This is the opposite of what you might be used to when laying out screen position as it starts from the lower left corner. Thus the resulting Sprite is only a portion of the image file. In this case the Sprite dimension is 40 x 40 starting at the top left corner.

If you don't specify a Rect, Cocos2d-x will automatically use the full width and height of the image file you specify. Take a look at the example below. If we use an image with dimensions 200 x 200 the following 2 statements would have the same result.

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

**var** mySprite = **new** cc.Sprite(res.mySprite\_png, cc.rect(0,0,200,200));

## Creating a Sprite from a Sprite Sheet

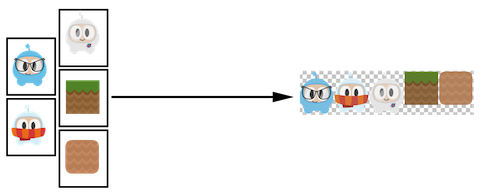
A **sprite sheet** is a way to combine sprites into a single file. Using a **sprite sheet** helps achieve better performance by **batching** the **draw calls**. They can also save disk and video memory in cases where the sprites can be packed on a sheet more efficiently (generally requires special tools). You will read more about this in the Advanced Chapter, but it is but it is one of many standard methods in the industry for increasing game performance.

When using a **sprite sheet** it is first loaded, in its entirety, into the SpriteFrameCache. SpriteFrameCache is a caching class that retains the SpriteFrame objects added to it, for future quicker access. The SpriteFrame is loaded once and retained in the SpriteFrameCache

Here is an example sprite sheet:



It doesn't look like much but let's take a closer look at what is happening:



As you can see the **sprite sheet**, at a minimum it reduces unneeded space and consolidates all sprites into a single file.

Let's tie this all together!

### Loading a Sprite Sheet

Load your **sprite sheet** into the SpriteFrameCache, probably in **AppDelegate**:

*// load the Sprite Sheet*

**var** spritecache = cc.SpriteFrameCache;

*// the .plist file can be generated with any of the tools mentioned below*

spritecache.addSpriteFramesWithFile(res.sprites\_plist);

Now that we have a **sprite sheet** loaded into SpriteFrameCache we can create Sprite objects by utilizing it.

### Creating a Sprite from SpriteFrameCache

This creates a Sprite by pulling it from the SpriteFrameCache.

*// Our .plist file has names for each of the sprites in it. We'll grab*

*// the sprite named, "Blue\_Front1" from the sprite sheet:*

**var** mysprite = cc.Sprite.createWithSpriteFrameName(res.mySprite\_png);



### Creating a Sprite from a SpriteFrame

Another way to create the same Sprite is by fetching the SpriteFrame from the SpriteFrameCache, and then creating the Sprite with the SpriteFrame. Example:

*// this is equivalent to the previous example,*

*// but it is created by retrieving the SpriteFrame from the cache.*

**var** newspriteFrame = cc.SpriteFrameCache.getSpriteFrameByName(res.sprites\_plist);

**var** newSprite = cc.Sprite.createWithSpriteFrame(newspriteFrame);



### Tools for creating Sprite Sheets

Creating a **sprite sheet** manually is a tedious process. Fortunately there are tools that can generate them automatically. These tools can provide even more ways to adjust your **sprite sheet** for maximum optimization!

Here are a few tools:

* [Cocos Studio](http://www.cocos2d-x.org/wiki/CocoStudio)
* [ShoeBox](http://renderhjs.net/shoebox/)
* [Texture Packer](https://www.codeandweb.com/texturepacker)
* [Zwoptex](https://www.zwopple.com/zwoptex/)

## Sprite Manipulation

After creating a Sprite you will have access to a variety of properties it has that can be manipulated.

Given:

**var** mySprite = **new** Sprite(res.mysprite\_png);



### Anchor Point and Position

**Anchor Point** is a point that you set as a way to specify what part of the Sprite will be used when setting its position. **Anchor Point** affects only properties that can be transformed. This includes **scale**, **rotation**, **skew**. This excludes **color** and **opacity**. The **anchor point** uses a bottom left coordinate system. This means that when specifying X and Y coordinate values you need to make sure to start at the bottom left hand corner to do your calculations. By default, all Node objects have a default **anchor point** of is **(0.5, 0.5)**.

Setting the **anchor point** is easy:

*// DEFAULT anchor point for all Sprites*

mySprite.setAnchorPoint(cc.\_p(0.5, 0.5));

*// bottom left*

mySprite.setAnchorPoint(cc.\_p(0, 0));

*// top left*

mySprite.setAnchorPoint(cc.\_p(0, 1));

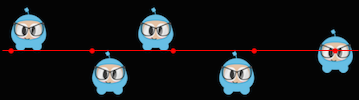
*// bottom right*

mySprite.setAnchorPoint(cc.\_p(1, 0));

*// top right*

mySprite.setAnchorPoint(cc.\_p(1, 1));

To represent this visually:

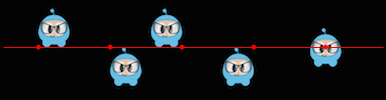


### Sprite properties effected by anchor point

Using **anchor point** effects only properties that can be transformed. This includes **scale**, **rotation**, **skew**.

#### Position

A **sprite's** position is affected by its **anchor point** as it is this point that is used as a starting point for positioning. Let's visually look at how this happens. Notice the colored line and where the sprite's position is in relation to it. Notice, as we change the **anchor point** values, the sprite's position changes. It is important to note that all it took was changing the **anchor point** value. We did not use a setPosition() statement to achieve this:



There are more ways to set position than just **anchor point**. Sprite objects can also be set using the setPosition() method.

mySprite.setPosition(cc.\_p(100, 200));

#### Rotation

Changes the **sprite's** rotation, by a positive or negative number of degrees. A positive value rotates the Sprite object clockwise, while a negative value rotates the Sprite object counter-clockwise. The default value is **0**.

*// rotate sprite by +20 degrees*

mySprite.setRotation(cc.\_p(20.0));

*// rotate sprite by -20 degrees*

mySprite.setRotation(cc.\_p(-20.0));

*// rotate sprite by +60 degrees*

mySprite.setRotation(cc.\_p(60.0));

*// rotate sprite by -60 degrees*

mySprite.setRotation(cc.\_p(-60.0));



#### Scale

Changes the **sprite's** scale, either by x, y or uniformly for both x and y. The default value is 1.0 for both x and y.

*// increases X and Y size by 2.0 uniformly*

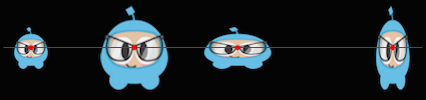
mySprite.setScale(cc.\_p(2.0));

*// increases just X scale by 2.0*

mySprite.setScaleX(cc.\_p(2.0));

*// increases just Y scale by 2.0*

mySprite.setScaleY(cc.\_p(2.0));



#### Skew

Changes the **sprite's** skew, either by x, y or uniformly for both x and y. The default value is 0,0 for both x and y.

*// adjusts the X skew by 20.0*

mySprite.setSkewX(cc.\_p(20.0));

*// adjusts the Y skew by 20.0*

mySprite.setSkewY(cc.\_p(20.0));



### Sprite properties not affected by anchor point

There are a few properties of Sprite objects that are not affected by **anchor point**. Why? Because they only change superficial qualities like **color** and **opacity**.

#### Color

Changes the sprite's color. This is done by passing in a Color3B object. Color3B objects are **RGB** values. We haven't encountered Color3B yet but it is simply an object that defines an **RGB color**. An **RGB color** is a 3 byte value from 0 - 255. Cocos2d-x also provides pre-defined colors that you can pick from. Using these will be a bit faster since they are pre-defined. A few examples: Color3B::White and Color3B::Red.

*// set the color by passing in a pre-defined Color3B object.*

mySprite.setColor(cc.color.WHITE);

*// Set the color by passing in a Color3B object.*

mySprite.setColor(cc.color(255, 255, 255)); *// Same as Color3B::WHITE*



#### Opacity

Changes the sprite's opacity by the specified value. An opaque object is not transparent at all. This property expects a value from 0 to 255, where 255 means fully opaque and 0 means fully transparent. Think: **zero opacity means invisible**, and you'll always understand how this works. The default value is 255 (fully opaque).

*// Set the opacity to 30, which makes this sprite 11.7% opaque.*

*// (30 divided by 256 equals 0.1171875...)*

mySprite.setOpacity(30);



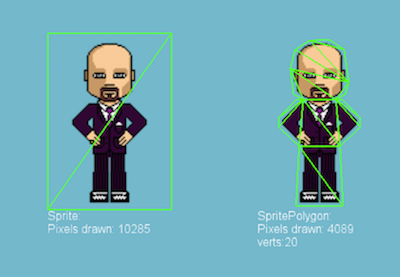
## Polygon Sprite

A **Polygon Sprite** is also a Sprite, that is used to display a 2d image. However, unlike a normal Spriteobject, which is a rectangle made of just 2 triangles, PolygonSprite objects are made of a series of triangles.

#### Why use a Polygon Sprite?

Simple, **performance**!

There is a lot of technical jargon that we can toss around here about **pixel fill rate** but the take home lesson is that a PolygonSprite draws based upon the shape of your Sprite, not a simple rectangle around the largest width and height. This saves a lot of unnecessary drawing. Consider this example:



Notice the difference between the left and right versions?

On the left, a typical Sprite drawn in rectangular fashion by the use of 2 triangles.

On the right, a PolygonSprite drawn with many smaller triangles.

Whether or not this trade-off is worth it for purely performance reasons depends on a number of factors (sprite shape/detail, size, quantity drawn on screen, etc.), but in general, vertices are cheaper than pixels on modern GPUs.

#### AutoPolygon

AutoPolygon is a helper class. It's purpose is to process an image into a 2d polygon mesh at runtime.

There are functions for each step in the process, from tracing all the points, to triangulation. The result, can be then passed to a Sprite objects **create** function to create a PolygonSprite. Example:

*// Generate polygon info automatically.*

**var** pinfo = cc.autopolygon.generatePolygon(res.mysprite\_png);

*// Create a sprite with polygon info.*

**var** sprite = **new** cc.Sprite(pinfo);

<http://cocos2d-x.org/docs/programmers-guide/actions/index.html>

* [**Actions**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html#actions)
* [**By and To, what is the difference?**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html#by-and-to-what-is-the-difference)
* [**Basic Actions and how to run them**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html#basic-actions-and-how-to-run-them)
* [**Sequences and how to run them**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html#sequences-and-how-to-run-them)

# Actions

Action objects are just like they sound. They make a Node perform a change to its properties. Actionobjects allow the transformation of Node properties in time. Any object with a base class of Node can have Action objects performed on it. As an example, you can move a Sprite from one position to another and do it over a span of time.

Example of MoveTo and MoveBy action:

* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

*// Move sprite to position 50,10 in 2 seconds.*

**var** moveTo = **new** cc.MoveTo(2, cc.\_p(50, 10));

mySprite1.runAction(moveTo);

*// Move sprite 20 points to right in 2 seconds*

**var** moveBy = **new** cc.MoveBy(2, cc.\_p(20,0));

mySprite2.runAction(moveBy);

### By and To, what is the difference?

You will notice that each Action has a **By** and **To** version. Why? Because they are different in what they accomplish. A **By** is relative to the current state of the Node. A **To** action is absolute, meaning it doesn't take into account the current state of the Node. Let's take a look at a specific example:

* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

mySprite.setPosition(cc.\_p(200, 256));

*// MoveBy - lets move the sprite by 500 on the x axis over 2 seconds*

*// MoveBy is relative - since x = 200 + 200 move = x is now 400 after the move*

**var** moveBy = **new** cc.MoveBy(2, cc.\_p(500, mySprite.y));

*// MoveTo - lets move the new sprite to 300 x 256 over 2 seconds*

*// MoveTo is absolute - The sprite gets moved to 300 x 256 regardless of*

*// where it is located now.*

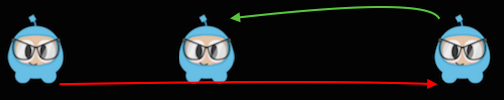
**var** moveTo = **new** cc.MoveTo(2, cc.\_p(300, mySprite.y));

*// Delay - create a small delay*

**var** delay = **new** cc.DelayTime(1);

**var** seq = **new** cc.Sequence(moveBy, delay, moveTo);

mySprite.runAction(seq);



## Basic Actions and how to run them

Basic actions are usually a singular action, thus accomplishing a single objective. Let's take a look at a few examples:

### Move

**Move** a Node over a set period of time.

* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

*// Move a sprite to a specific location over 2 seconds.*

**var** moveTo = **new** cc.MoveTo(2, cc.\_p(50, 0));

mySprite.runAction(moveTo);

*// Move a sprite 50 pixels to the right, and 0 pixels to the top over 2 seconds.*

**var** moveBy = **new** cc.MoveBy(2, cc.\_p(50, 0));

mySprite.runAction(moveBy);



### Rotate

**Rotate** a Node clockwise over 2 seconds.

* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

*// Rotates a Node to the specific angle over 2 seconds*

**var** rotateTo = **new** cc.RotateTo(2.0, 40.0);

mySprite.runAction(rotateTo);

*// Rotates a Node clockwise by 40 degree over 2 seconds*

**var** rotateBy = **new** cc.RotateBy(2.0, 40.0);

mySprite.runAction(rotateBy);



### Scale

**Scale** a Node by 10 over 2 seconds.

* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

*// Scale uniformly by 3x over 2 seconds*

**var** scaleBy = **new** cc.ScaleBy(2.0, 3.0);

mySprite.runAction(scaleBy);

*// Scale X by 5 and Y by 3x over 2 seconds*

**var** scaleBy = **new** cc.ScaleBy(2.0, 3.0, 3.0);

mySprite.runAction(scaleBy);

*// Scale to uniformly to 3x over 2 seconds*

**var** scaleTo = **new** cc.ScaleTo(2.0, 3.0);

mySprite.runAction(scaleTo);

*// Scale X to 5 and Y to 3x over 2 seconds*

**var** scaleTo = **new** cc.ScaleTo(2.0, 3.0, 3.0);

mySprite.runAction(scaleTo);



### Fade In/Out

**Fade** a Node.

It modifies the opacity from 0 to 255. The reverse of this action is FadeOut

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

*// fades in the sprite in 1 seconds*

**var** fadeIn = **new** cc.FadeIn(1.0);

mySprite.runAction(fadeIn);

*// fades out the sprite in 2 seconds*

**var** fadeOut = **new** cc.FadeOut(2.0);

mySprite.runAction(fadeOut);



### Tint

Tint a Node that implements the NodeRGB protocol from current the tint to \ a custom tine.

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** cc.Sprite(res.mysprite\_png);

*// Tints a node to the specified RGB values*

**var** tintTo = **new** cc.TintTo(2.0, 120.0, 232.0, 254.0);

mySprite.runAction(tintTo);

*// Tints a node BY the delta of the specified RGB values.*

**var** tintBy = **new** cc.TintBy(2.0, 120.0, 232.0, 254.0);

mySprite.runAction(tintBy);



### Animate

With Animate it is possible to do simple **flipbook** animation with your Sprite objects. This is simply replacing the **display frame** at set intervals for the duration of the animation. Let's consider this example:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

**var** mySprite = **new** Sprite(res.mysprite\_png);

*// now lets animate the sprite we moved.*

**var** animFrames;

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Front1\_png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Front2.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Front3.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Left1.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Left2.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Left3.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Back1.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Back2.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Back3.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Right1.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Right2.png), cc.Rect(0,0,65,81)));

animFrames.push(**new** cc.SpriteFrame(res.Blue\_Right3.png), cc.Rect(0,0,65,81)));

*// create the animation out of the frames*

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

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

*// run it and repeat it forever*

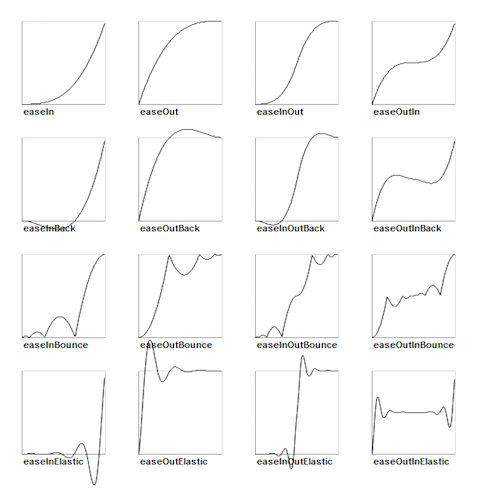
mySprite.runAction(cc.RepeatForever(animate));

It's hard to show an animation in text, so please run the example **Programmer Guide Sample** code to see this in action!

### Easing

**Easing** is animating with a specified acceleration to make the animations smooth. A few things to keep in mind is that regardless of speed, ease actions always start and finish at the same time. **Ease** actions are a good way to **fake** physics in your game! Perhaps you want a few simulated physics effects but don't want the overhead and complexity of adding it all for a few very basic actions. Another good example is to animate menus and buttons.

Here are common easing functions displayed over a graph:



Cocos2d-x supports most of the easing function in the above graph. They are also simple to implement. Lets look at a specific use case. Lets drop a Sprite object from the top of the screen and make it bounce.

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

Run the example **Programmer Guide Sample** code to see this in action!

## Sequences and how to run them

**Sequences** are a series of Action objects to be executed sequentially. This can be any number of Actionobjects, **Functions** and even another Sequence. Functions? Yes! Cocos2d-x has a CallFunc object that allows you to create a **function()** and pass it in to be run in your Sequence. This allows you to add your own functionality to your Sequence objects besides just the stock Action objects that Cocos2d-x provides. This is what a Sequence looks like when executing:



### An example sequence

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

So what does this Sequence action do?

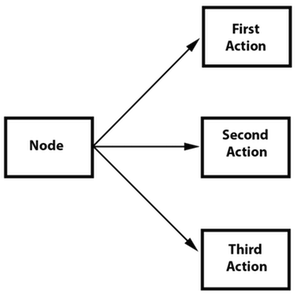
It will execute the following actions sequentially:

**Jump** -> **callbackJump()** -> **Rotate** -> **callbackRotate()**

Run the example **Programmer Guide Sample** code to see this in action!

### Spawn

**Spawn** is very similar to Sequence, except that all actions will run at the same time. You can have any number of Action objects and even other Spawn objects!



Spawn produces the same result as running multiple consecutive **runAction()** statements. However, the benefit of spawn is that you can put it in a Sequence to help achieve specific effects that you cannot otherwise. Combining Spawn and Sequence is a very powerful feature.

Example, given:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

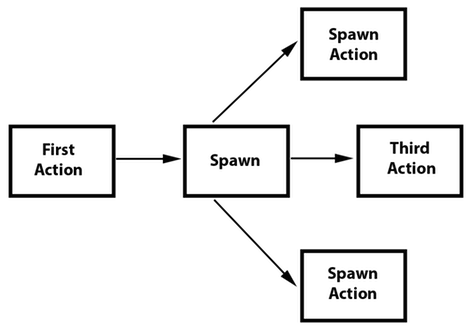
Using a Spawn:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

and consecutive **runAction()** statements:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

Both would produce the same result. However, one can use Spawn in a Sequence. This flowchart shows how this might look:



* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

Run the example **Programmer Guide Sample** code to see this in action!

### Clone

**Clone** is exactly like it sounds. If you have an Action, you can apply it to multiple Node objects by using clone(). Why do you have to clone? Good question. Action objects have an **internal state**. When they run, they are actually changing the Node objects properties. Without the use of clone() you don't truly have a unique Action being applied to the Node. This will produce unexpected results, as you can't know for sure what the properties of the Action are currently set at.

Let's hash through an example, say you have a **heroSprite** and it has a position of **(0,0)**. If you run an Actionof:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

This will move **heroSprite** from (0,0) to (400, 100) over the course of 10 seconds. **heroSprite** now has a new position of (400, 100) and more importantly the Action has this position in it's **internal state**. Now, say you have an **emenySprite** with a position of (200, 200). If you were to apply this same:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

to your **enemySprite**, it would end up at a position of (800, 200) and not where you thought it would. Do you see why? It is because the Action already had an **internal state** to start from when performing the MoveBy. **Cloning** an Action prevents this. It ensures you get a unique version Action applied to your Node.

Let's also see this in code, first, incorrect.

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

Correctly, using **clone()**!:

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

### Reverse

**Reverse** is also exactly like it sounds. If you run a series of actions, you can call reverse() to run it, in the opposite order. Otherwise known as, backwards. However, it is not just simply running the Action in reverse order. Calling reverse() is actually manipulating the properties of the original Sequence or Spawn in reverse too.

Using the Spawn example above, reversing is simple.

* [**C++**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)
* [**Javascript**](http://cocos2d-x.org/docs/programmers-guide/actions/index.html)

Most Action and Sequence objects are reversible!

It's easy to use, but let's make sure we see what is happening. Given:

What is really happening? If we lay out the steps as a list it might be helpful:

* **mySprite** is created
* **mySprite** position is set to (50, 56)
* **sequence** starts to run
* **sequence** moves **mySprite** by 500, over 2 seconds, **mySprite** new position (550, 56)
* **sequence** delays for 2 seconds
* **sequence** scales **mySprite** by 2x over 2 seconds
* **sequence** delays for 6 more seconds (notice we run another sequence to accomplish this)
* we run a **reverse()** on the sequence so we re-run each action backwards
* **sequence** is delayed for 6 seconds
* **sequence** scales **mySprite** by -2x over 2 seconds
* **sequence** delays for 2 seconds
* **sequence** moves **mySprite** by -500, over 2 seconds, **mySprite** new position (50, 56)

You can see that a reverse() is simple for you to use, but not so simple in its internal logic. Cocos2d-x does all the heavy lifting!

<http://cocos2d-x.org/docs/programmers-guide/event_dispatch/>

* [**Event Dispatcher**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#event-dispatcher)
* [**What is the EventDispatch mechanism?**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#what-is-the-eventdispatch-mechanism)
* [**5 types of event listeners.**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#5-types-of-event-listeners)
* [**FixedPriority vs SceneGraphPriority**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#fixedpriority-vs-scenegraphpriority)
* [**Touch Events**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#touch-events)
* [**Swallowing Events**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#swallowing-events)
* [**Creating a keyboard event**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#creating-a-keyboard-event)
* [**Creating an accelerometer event**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#creating-an-accelerometer-event)
* [**Creating a mouse event**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#creating-a-mouse-event)
* [**Creating a Custom Event**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#creating-a-custom-event)
* [**Registering event with the dispatcher**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#registering-event-with-the-dispatcher)
* [**Removing events from the dispatcher**](http://cocos2d-x.org/docs/programmers-guide/event_dispatch/#removing-events-from-the-dispatcher)

# Event Dispatcher

## What is the EventDispatch mechanism?

**EventDispatch** is a mechanism for responding to user events.

The basics:

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

## 5 types of event listeners.

EventListenerTouch - responds to touch events

EventListenerKeyboard - responds to keyboard events

EventListenerAcceleration - responds to accelerometer events

EventListenMouse - responds to mouse events

EventListenerCustom - responds to custom events

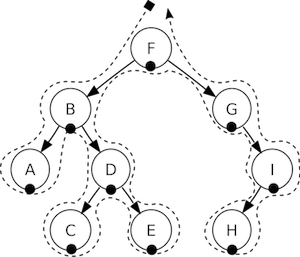
## FixedPriority vs SceneGraphPriority

The **EventDispatcher** uses priorities to decide which listeners get delivered an event first.

**Fixed Priority** is an integer value. Event listeners with lower Priority values get to process events before event listeners with higher Priority values.

**Scene Graph Priority** is a pointer to a Node. Event listeners whose Nodes have higher **z-order** values (that is, are drawn on top) receive events before event listeners whose Nodes have lower **z-order** values (that is, are drawn below). This ensures that touch events, for example, get delivered front-to-back, as you would expect.

Remember Chapter 2? Where we talked about the **scene graph** and we talked about this diagram?



Well, when use **Scene Graph Priority** you are actually walking this above tree backwards... **I**, **H**, **G**, **F**, **E**, **D**, **C**, **B**, **A**. If an event is triggered, **H** would take a look and either **swallow** it (more on this below) or let is pass through to \_I**. Same thing,**I**will either**consume**it or let is pass through to**G**and so on until the event either**swallowed\_\_ it or does not get answered.

## Touch Events

**Touch events** are the most important event in mobile gaming. They are easy to create and provide versatile functionality. Let's make sure we know what a touch event is. When you touch the screen of your mobile device, it accepts the touch, looks at where you touched and decides what you touched. Your touch is then answered. It is possible that what you touched might not be the responding object but perhaps something underneath it. Touch events are usually assigned a priority and the event with the highest priority is the one that answers. Here is how you create a basic touch event listener:

*// Create a "one by one" touch event listener*

*// (processes one touch at a time)*

**auto** listener1 = EventListenerTouchOneByOne::create();

*// trigger when you push down*

listener1->onTouchBegan = [](Touch\* touch, Event\* event){

*// your code*

**return** **true**; *// if you are consuming it*

};

*// trigger when moving touch*

listener1->onTouchMoved = [](Touch\* touch, Event\* event){

*// your code*

};

*// trigger when you let up*

listener1->onTouchEnded = [=](Touch\* touch, Event\* event){

*// your code*

};

*// Add listener*

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener1, **this**);

As you can see there are 3 distinct events that you can act upon when using a touch event listener. They each have a distinct time in which they are called.

**onTouchBegan** is triggered when you press down.

**onTouchMoved** is triggered if you move the object around while still pressing down.

**onTouchEnded** is triggered when you let up on the touch.

## Swallowing Events

When you have a listener and you want an object to accept the event it was given you must **swallow** it. To say it another way, you **consume** it so that it doesn't get passed to other objects in highest to lowest priority. This is easy to do.

*// When "swallow touches" is true, then returning 'true' from the*

*// onTouchBegan method will "swallow" the touch event, preventing*

*// other listeners from using it.*

listener1->setSwallowTouches(**true**);

*// you should also return true in onTouchBegan()*

listener1->onTouchBegan = [](Touch\* touch, Event\* event){

*// your code*

**return** **true**;

};

## Creating a keyboard event

For desktop games, you might want find using keyboard mechanics useful. Cocos2d-x supports keyboard events. Just like with touch events above, keyboard events are easy to create.

*// creating a keyboard event listener*

**auto** listener = EventListenerKeyboard::create();

listener->onKeyPressed = CC\_CALLBACK\_2(KeyboardTest::onKeyPressed, **this**);

listener->onKeyReleased = CC\_CALLBACK\_2(KeyboardTest::onKeyReleased, **this**);

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener, **this**);

*// Implementation of the keyboard event callback function prototype*

**void** KeyboardTest::onKeyPressed(EventKeyboard::KeyCode keyCode, Event\* event)

{

log("Key with keycode %d pressed", keyCode);

}

**void** KeyboardTest::onKeyReleased(EventKeyboard::KeyCode keyCode, Event\* event)

{

log("Key with keycode %d released", keyCode);

}

## Creating an accelerometer event

Some mobile devices come equipped with an accelerometer. An accelerometer is a sensor that measures g-force as well as changes in direction. A use case would be needing to move your phone back and forth, perhaps to simulate a balancing act. Cocos2d-x also supports these events and creating them is simple. Before using accelerometer events, you need to enable them on the device:

Device::setAccelerometerEnabled(**true**);

*// creating an accelerometer event*

**auto** listener = EventListenerAcceleration::create(CC\_CALLBACK\_2(

AccelerometerTest::onAcceleration, **this**));

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener, **this**);

*// Implementation of the accelerometer callback function prototype*

**void** AccelerometerTest::onAcceleration(Acceleration\* acc, Event\* event)

{

*// Processing logic here*

}

## Creating a mouse event

As it always has, Cocos2d-x supports mouse events.

\_mouseListener = EventListenerMouse::create();

\_mouseListener->onMouseMove = CC\_CALLBACK\_1(MouseTest::onMouseMove, **this**);

\_mouseListener->onMouseUp = CC\_CALLBACK\_1(MouseTest::onMouseUp, **this**);

\_mouseListener->onMouseDown = CC\_CALLBACK\_1(MouseTest::onMouseDown, **this**);

\_mouseListener->onMouseScroll = CC\_CALLBACK\_1(MouseTest::onMouseScroll, **this**);

\_eventDispatcher->addEventListenerWithSceneGraphPriority(\_mouseListener, **this**);

**void** MouseTest::onMouseDown(Event \*event)

{

*// to illustrate the event....*

EventMouse\* e = (EventMouse\*)event;

string str = "Mouse Down detected, Key: ";

str += tostr(e->getMouseButton());

}

**void** MouseTest::onMouseUp(Event \*event)

{

*// to illustrate the event....*

EventMouse\* e = (EventMouse\*)event;

string str = "Mouse Up detected, Key: ";

str += tostr(e->getMouseButton());

}

**void** MouseTest::onMouseMove(Event \*event)

{

*// to illustrate the event....*

EventMouse\* e = (EventMouse\*)event;

string str = "MousePosition X:";

str = str + tostr(e->getCursorX()) + " Y:" + tostr(e->getCursorY());

}

**void** MouseTest::onMouseScroll(Event \*event)

{

*// to illustrate the event....*

EventMouse\* e = (EventMouse\*)event;

string str = "Mouse Scroll detected, X: ";

str = str + tostr(e->getScrollX()) + " Y: " + tostr(e->getScrollY());

}

## Creating a Custom Event

The event types above are defined by the system, and the events (such as touch screen, keyboard response etc) are triggered by the system automatically. In addition, you can make your own custom events which are not triggered by the system, but by your code, as follows:

\_listener = EventListenerCustom::create("game\_custom\_event1", [=](EventCustom\* event){

std::string **str**("Custom event 1 received, ");

**char**\* buf = **static\_cast**<**char**\*>(event->getUserData());

str += buf;

str += " times";

statusLabel->setString(str.c\_str());

});

\_eventDispatcher->addEventListenerWithFixedPriority(\_listener, 1);

A custom event listener has been defined above, with a response method, and added to the event dispatcher. So how is the event handler triggered? Check it out:

**static** **int** count = 0;

++count;

**char**\* buf[10];

sprintf(buf, "%d", count);

EventCustom **event**("game\_custom\_event1");

event.setUserData(buf);

\_eventDispatcher->dispatchEvent(&event);

The above example creates an EventCustom object and sets its UserData. It is then dispatched manually with \_eventDispatcher->dispatchEvent(&event). This triggers the event handler defined previously. The handler is called immediately so a local stack variable can be used as the UserData.

## Registering event with the dispatcher

It is easy to register an event with the **Event Dispatcher**. Taking the sample touch event listener from above:

*// Add listener*

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener1,

sprite1);

It is important to note that a touch event can only be registered once per object. If you need to use the same listener for multiple objects you should use **clone()**.

*// Add listener*

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener1,

sprite1);

*// Add the same listener to multiple objects.*

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener1->clone(),

sprite2);

\_eventDispatcher->addEventListenerWithSceneGraphPriority(listener1->clone(),

sprite3);

## Removing events from the dispatcher

An added listener can be removed with following method:

\_eventDispatcher->removeEventListener(listener);

Although they may seem special, built-in Node objects use the **event dispatcher** in the same way we have talked out. Makes sense, right? Take Menu for an example. When you have a Menu with MenuItems when you click them you are dispatching a event. You can also **removeEventListener()** on built-in Node objects.