Cocos2d-x Programmers Guide v3.12

v2016.08.08

Authors: SlackMoehrle, Ricardo, Justin, Nite, Kai, Minggo, Wenhai, Tony, Yingtao, Rao, Huabin, Zhe

CSS enhancements: Nico

Contributors/Editors: stevetranby, Maxxx, smitpatel88, IQD, Michael, Meir_yanovich, catch_up, kidproquo, EMebane, reyanthonyrenacia, Fradow, pabitrapadhy, Manny_Dev, ZippoLag, kubas121, bjared, grimfate, DarioDP

Special Thanks: To our users! Without you there is no reason to even write this guide.

PDF Version

eBook Version

Please provide feedback for this guide on GitHub

You can download the samples for this guide on GitHub

What is Cocos2d-x?

About Cocos2d-x

Started in 2010, Cocos2d-x is an open source, cross-platform game engine. It is loaded with powerful features that allows developers to create spectacular games with ease.

Why choose Cocos2d-x

Why would you want to choose Cocos2d-x over other available game engines?

- Modern C++ API (please refer to the modernizing done in **version 3.0**)
- Cross-platform - desktop and mobile
- Capability to test and debug your game on the desktop and then push it to a mobile or desktop target
- A vast API of functionality including sprites, actions, animations, particles, transitions, timers, events (touch, keyboard, accelerometer, mouse), sound, file IO, persistence, skeletal animations, 3D

Where to get Cocos2d-x and what do I get?

You can clone the **GitHub Repo** and follow the steps in the **README**. You can also download as part of the Cocos package on our **download page**. No matter if you choose to develop in C++, JavaScript or Lua, everything you need is in one package. The Cocos family of products has a few different pieces.

- **Cocos2d-x** - this is the game engine, itself. It includes the engine and the **cocos** command-line tool. You can download a **production** release or stay bleeding edge by cloning our **GitHub Repo**.
- **Cocos Creator** - is a unified game development tool. You can create your entire game, from start to finish, using this tool. It uses JavaScript. Lua and C++ support are being added. Read more about **Cocos Creator**.
- **Cocos Launcher** - is a graphical tool to create and manage your projects. This is also an easy way to add SDKBOX plugins to your project. **Documentaion** is available.
- **Coco Studio** - is EOL'd and has been replaced by **Cocos Creator**. **Documentation** is still available.
- **Code IDE** - is EOL'd. Common text editors and IDE's can be used instead.

Conventions used in this book

- `auto` is used for creating local variables.
- `using namespace cocos2d;` is used to shorten types.
- each chapter has **a compilable source code sample** to demonstrate concepts.
- class names, methods names and other API components are rendered using fixed fonts. eg: `Sprite`.
- *italics* are used to notate concepts and keywords.

Learning Resources

- This very guide! You can also get it as a **PDF** and **ePub**
- **Sonar Systems Videos**
- **Android Fundamentals**
- **Make School Tutorials**
- **Game From Scratch**

Spreading the word!

You can help us spread the word about Cocos2d-x! We would surely appreciate it!

- Talk about us on Facebook! Our **Facebook Page**
- Tweet, Tweet! Our **Twitter**
- Read our **Blog** and promote it on your social media.

- Become a **Regional Coordinator**

Where to get help

- **Forums**
- **Bug Tracker**
- IRC. We are in **Freenode** in the *#cocos2d* channel
- `cpp-tests` project. This project is our basis for testing. Use this project to learn how we implement the functionality of the engine. This project is located in **Cocos2d-x_root/build.**
- **API Reference**.

Basic Cocos2d-x Concepts

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.

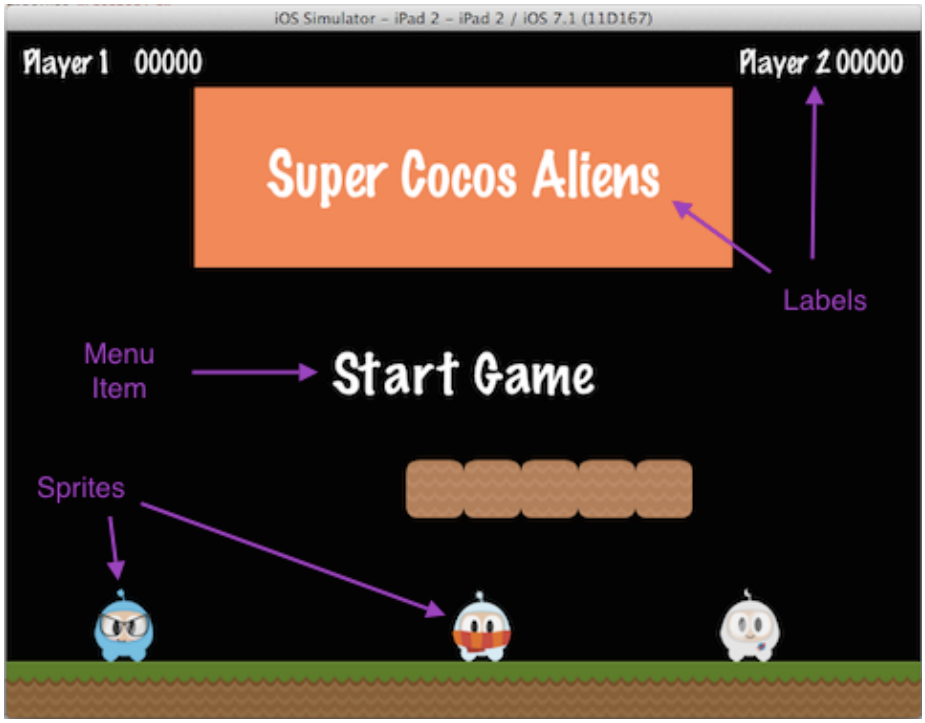
Cocos2d-x provides `Scene`, `Transition`, `Sprite`, `Menu`, `Sprite3D`, `Audio` objects and much more. Everything you need to create your games are included.

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:

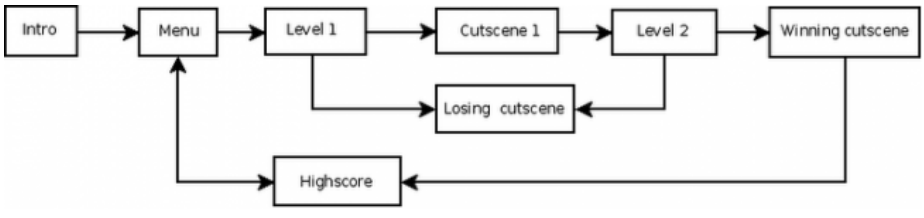


You can see a menu, some sprites and labels, which all have an equivalent in Cocos2d-x. Take a look at a few of your own game design documents,and see what components you have, you'll probably have a few that match up.

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

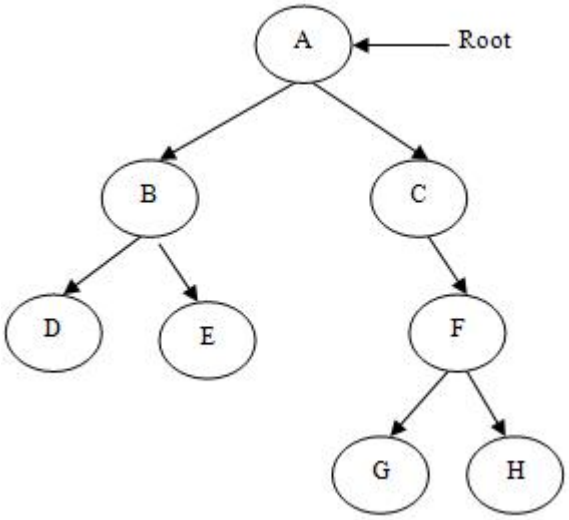
In your game you probably want a main menu, a few levels and an ending scene. How do you organize all of these into the separate pieces they are? You guessed it, `Scene`. When you think about your favorite movie you can see that it's distinctly broken down into scenes, or separate parts of the story 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

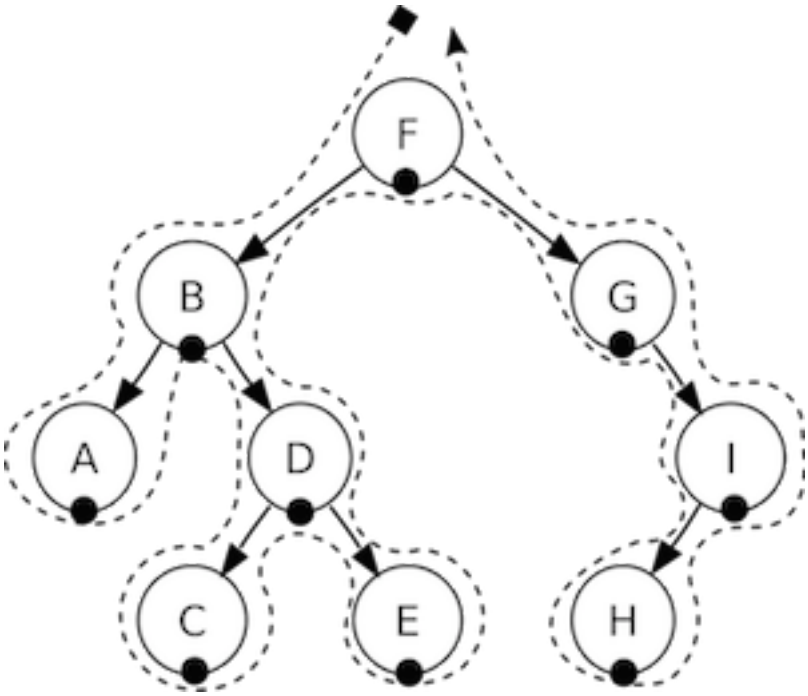
A **scene graph** is a data structure that arranges a graphical scene. A **scene graph** contains `Node` objects in a tree (yes, it is called **scene graph**, but it is actually represented by a **tree**) structure.



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.

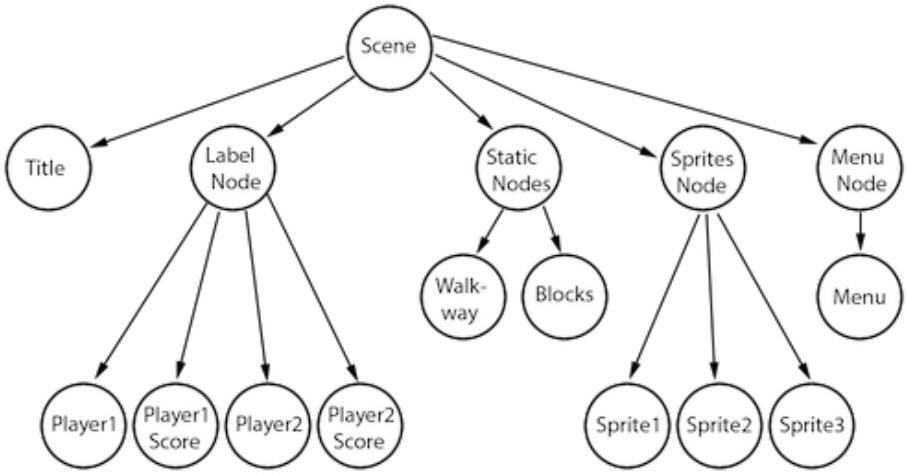
Once you start adding nodes, sprites and animations to your game, you want to make sure you are drawing the things you expect. But what if you are not? What if your sprites are hidden in the background and you want them to be the foremost objects? No big deal, just take a step back and run through the 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**.



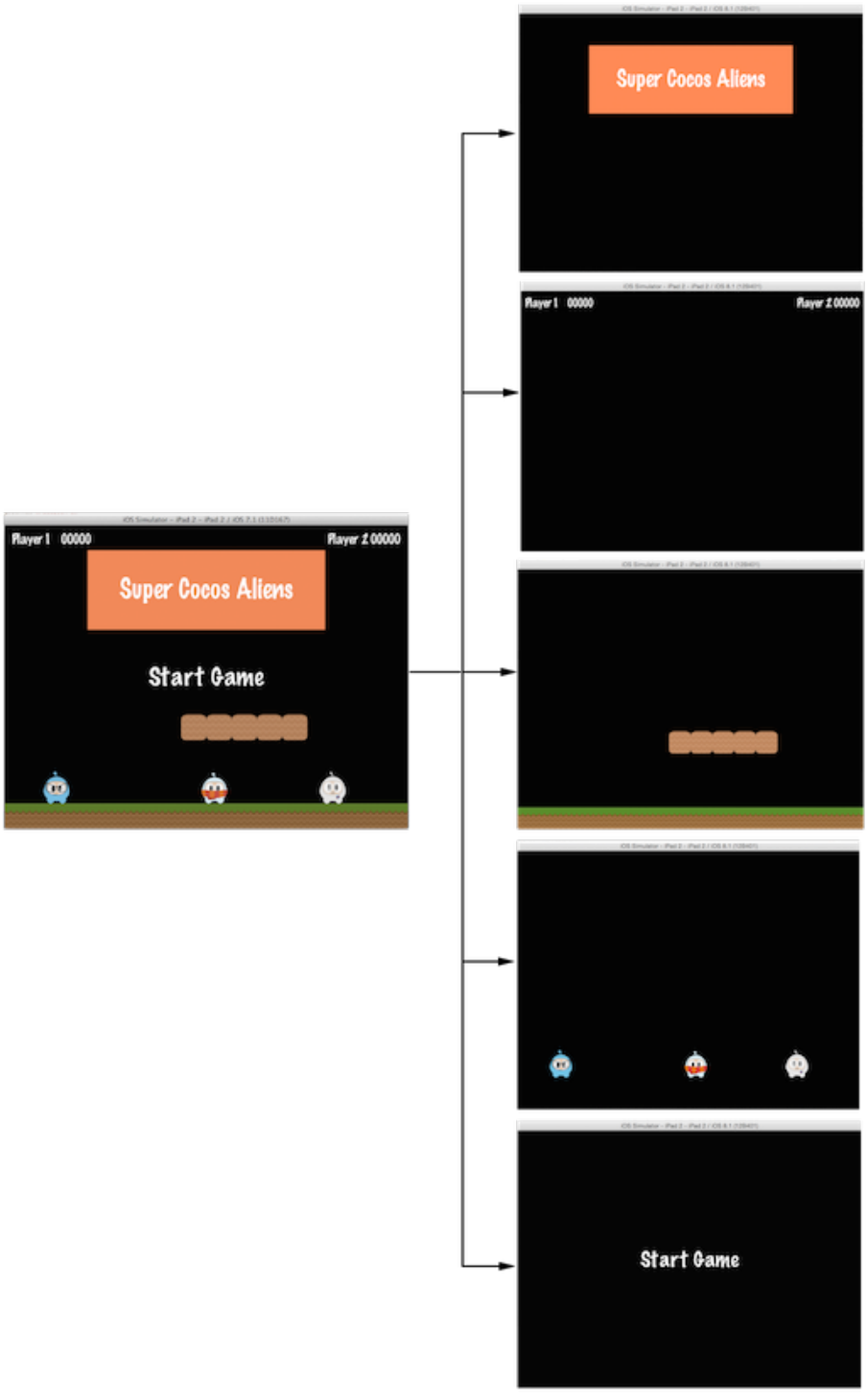
The **scene graph** is easily demonstrated, let's take a look at our game scene broken down:

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`:


Player 1 00000
Player 2 00000
Super Cocos Aliens
Start Game
Super Cocos Aliens
Player 1 00000
Player 2 00000
Start Game

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:

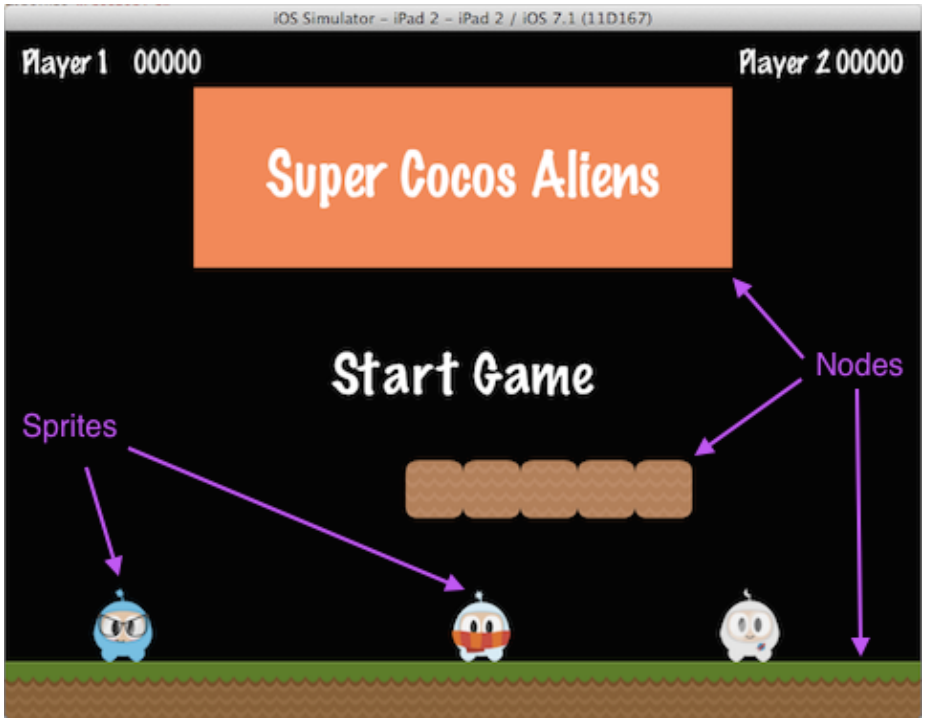
```
// Adds a child with the z-order of -2, that means
// it goes to the "left" side of the tree (because it is negative)
scene->addChild(title_node, -2);

// 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);
```

```
// 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`.

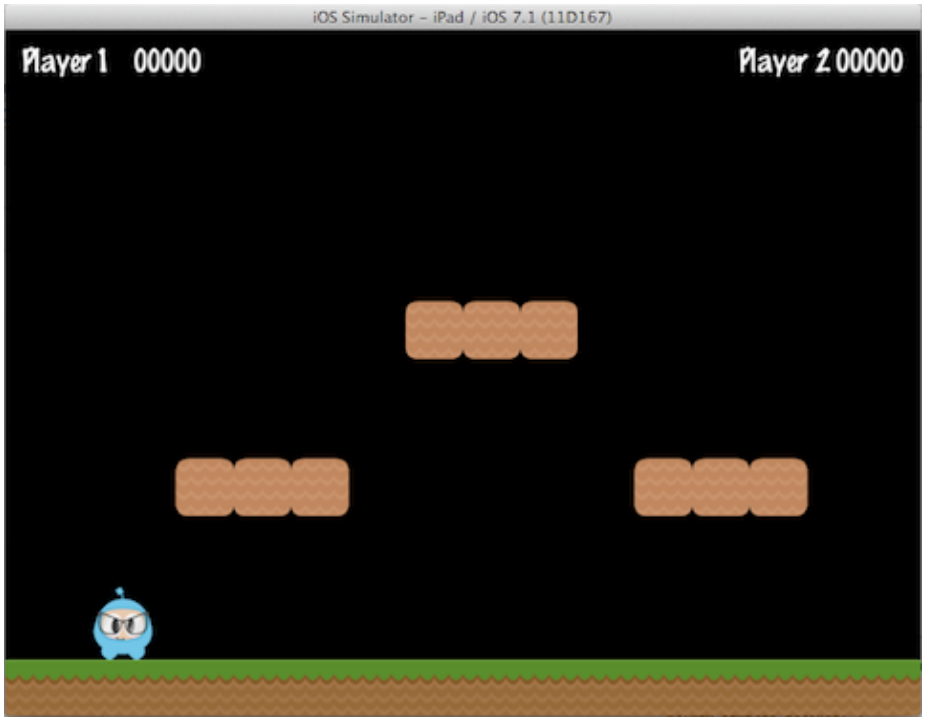
`Sprites` are easy to create and they have configurable properties like: **position**, **rotation**, **scale**, **opacity**, **color** and more.

```
// This is how to create a sprite
auto mySprite = Sprite::create("mysprite.png");

// this is how to change the properties of the sprite
mySprite->setPosition(Vec2(500, 0));

mySprite->setRotation(40);

mySprite->setScale(2.0); // sets both the scale of the X and Y axis uniformly

mySprite->setAnchorPoint(Vec2(0, 0));
```

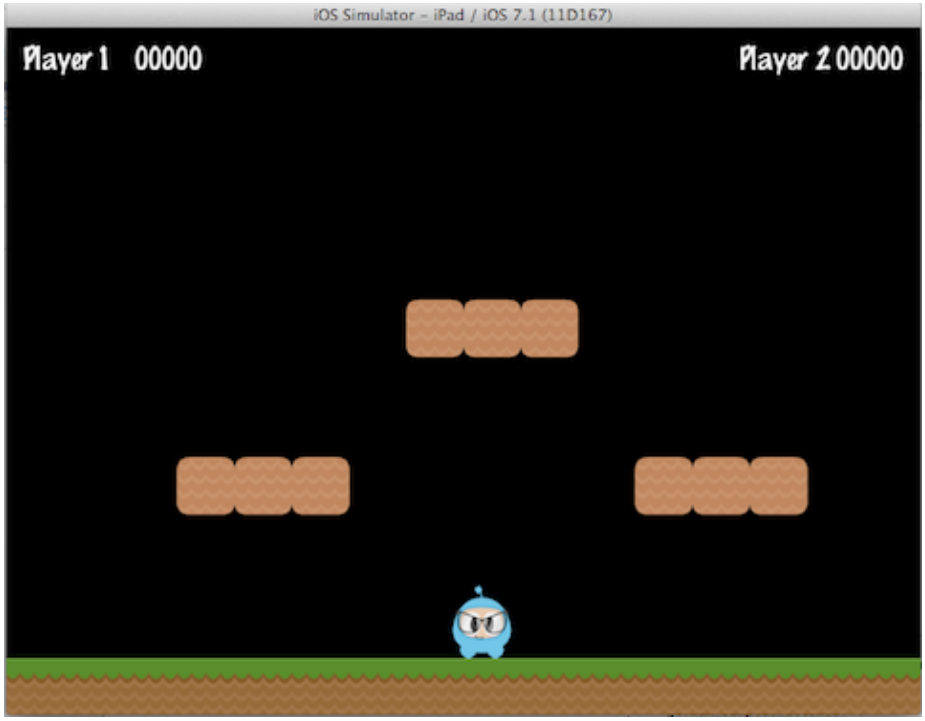
```
// This is how to create a sprite
var mySprite = new cc.Sprite(res.mySprite_png);

// 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:

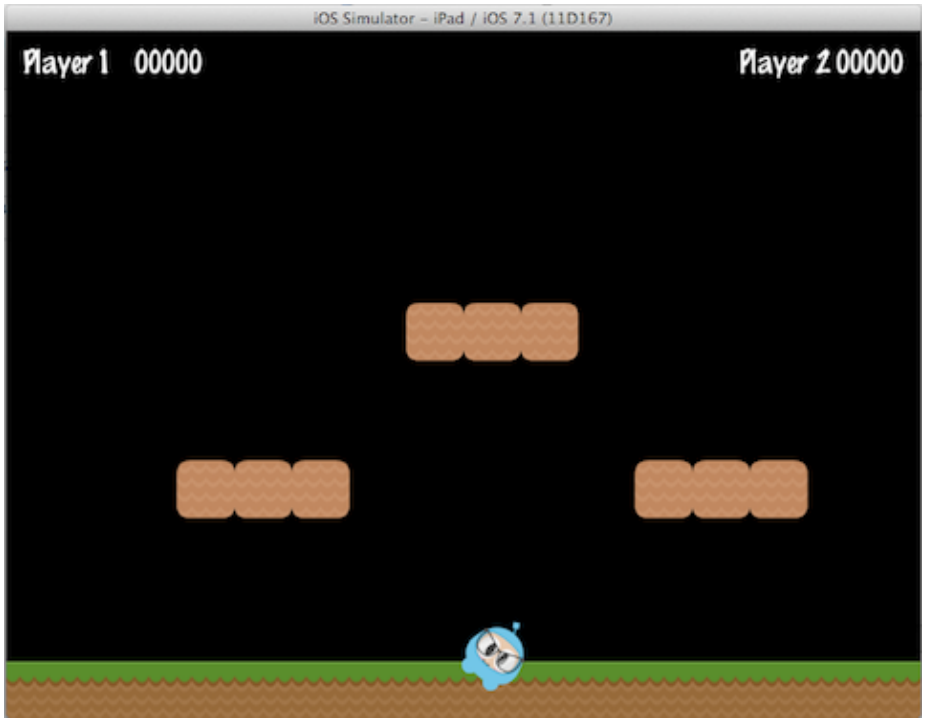


If we set the position using `mySprite->setPosition(Vec2(500, 0));`:



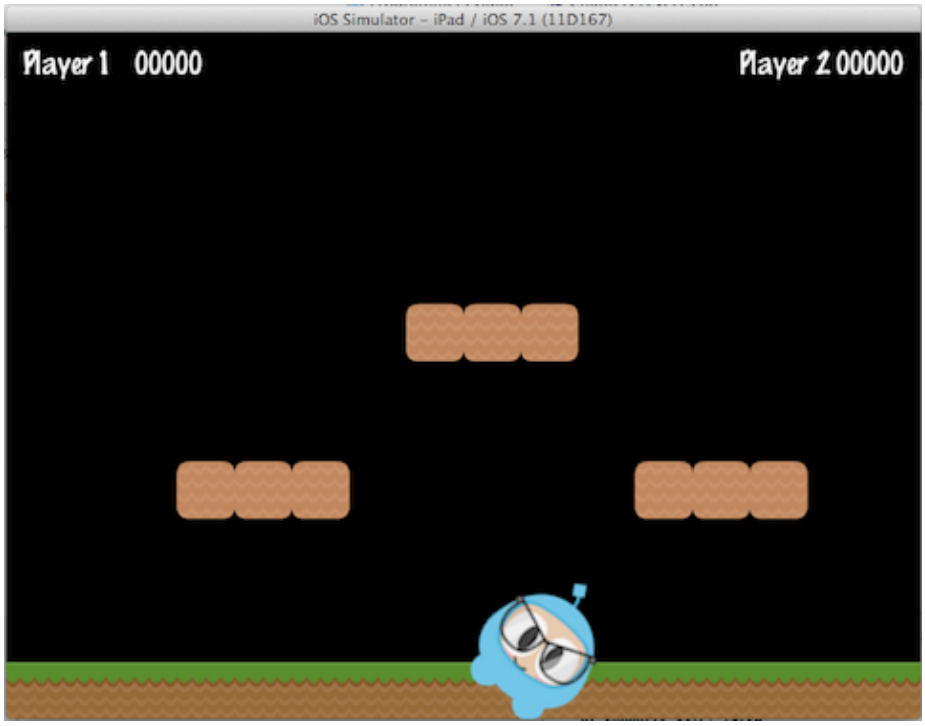
Note that the `Sprite` position has changed from its original position to the new position that we specified.

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.

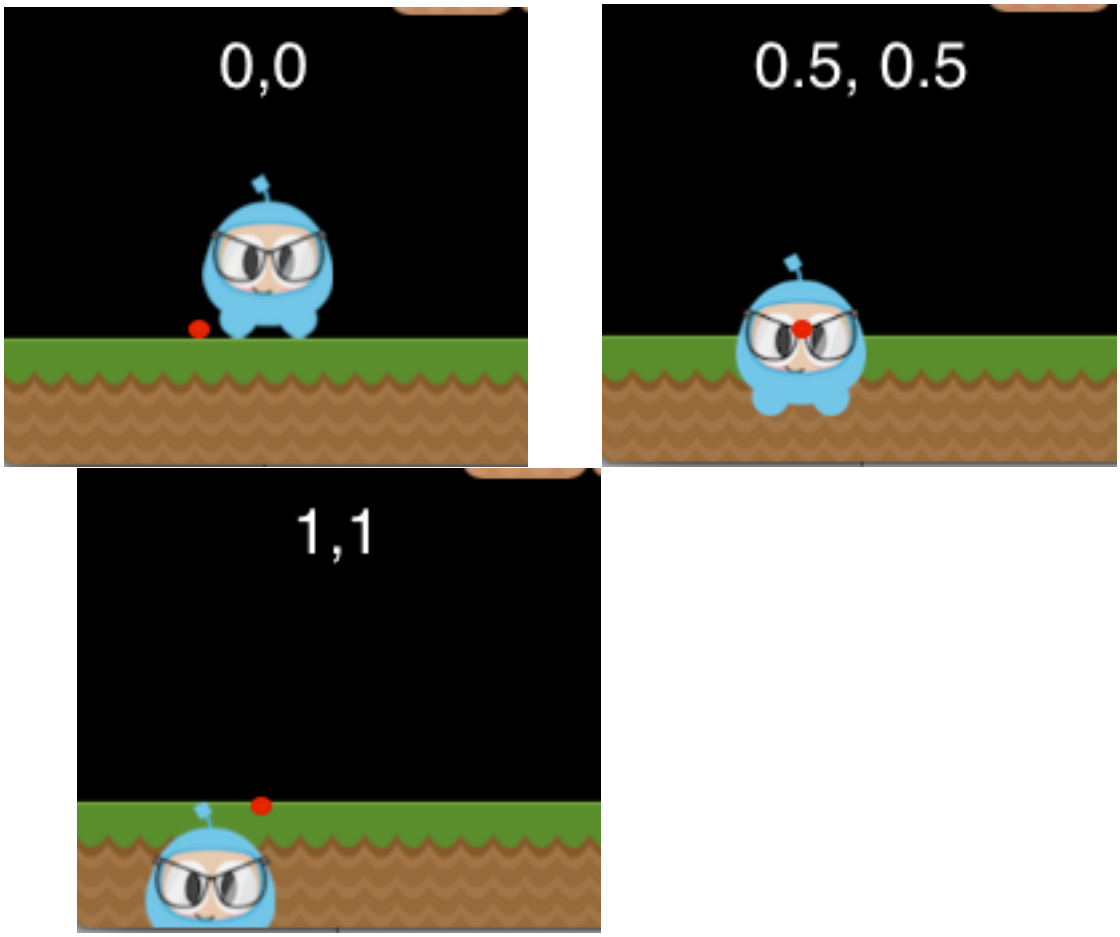
Lastly, all `Node` objects (since a `Sprite` is a subclass of `Node`) have a value for **anchor point**. We haven't talked about this yet, so now is a good time. You can think of **anchor point** as a way of specifying what part of the sprite will be used as a base coordinate when setting the position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

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

```
mySprite->setAnchorPoint(Vec2(0, 0));
```

```
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:



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.

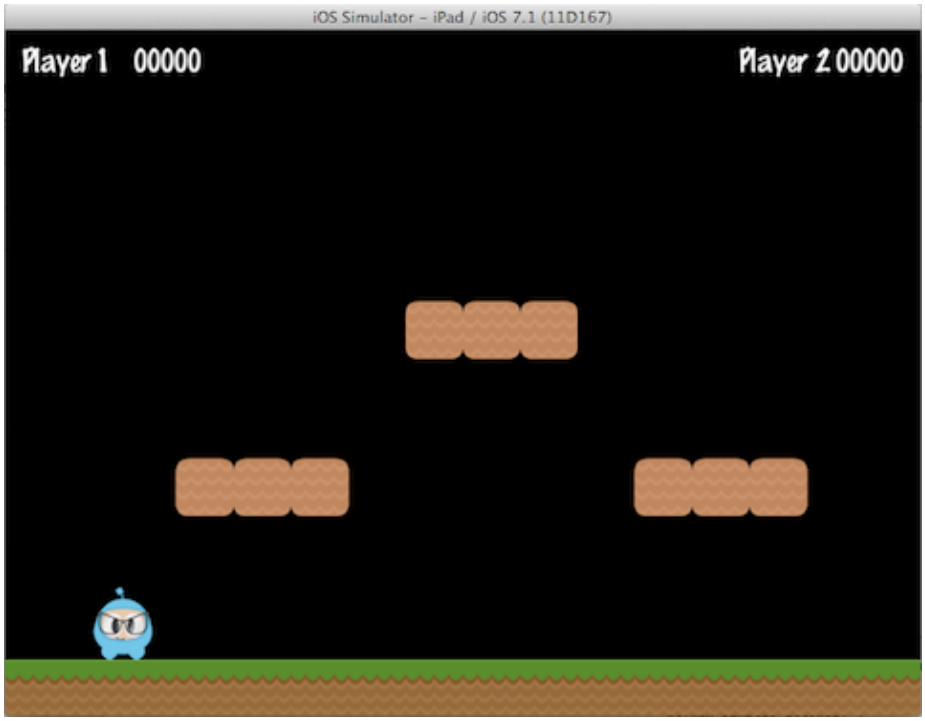
We really can tweak just about every aspect of the `Sprite`. But, what if we wanted to have these same types of changes occur in an automated, time determined manner? Well, keep reading…

Actions

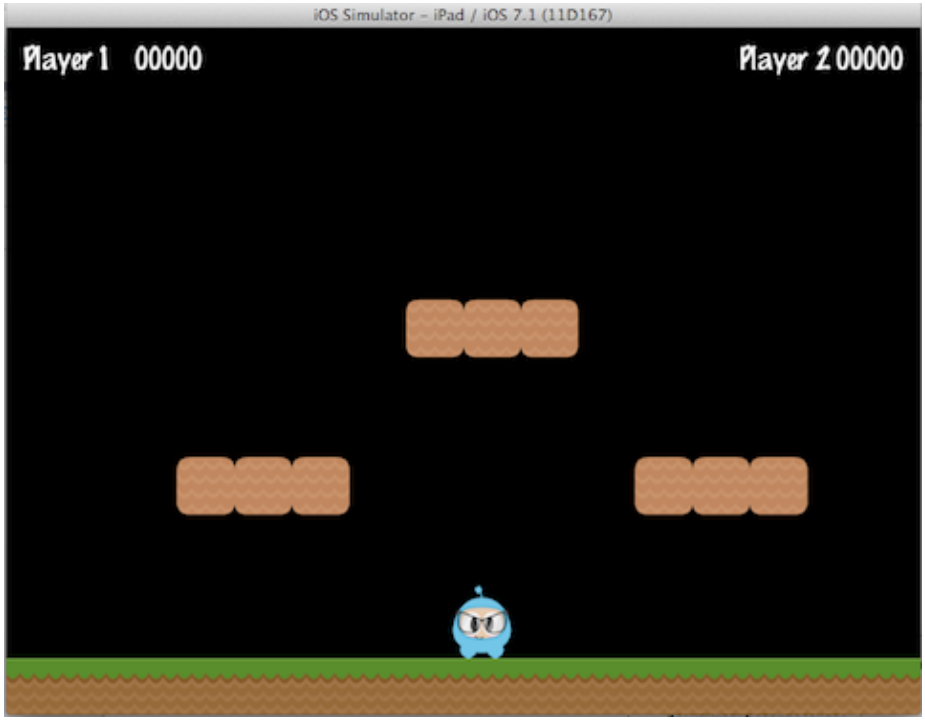
Creating a `Scene` and adding `Sprite` objects on the screen is only part of what we need to do. For a game to be a game we need to make things move around! `Action` objects are an integral part of every game. **Actions** allow the transformation of `Node` objects in time space. Want to move a `Sprite` 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 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:

```
auto mySprite = Sprite::create("Blue_Front1.png");

// Move a sprite 50 pixels to the right, and 10 pixels to the top over 2 seconds.
auto moveBy = MoveBy::create(2, Vec2(50,10));
mySprite->runAction(moveBy);

// Move a sprite to a specific location over 2 seconds.
auto moveTo = MoveTo::create(2, Vec2(50,10));
mySprite->runAction(moveTo);
```

```
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 `Sequence` in 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:

This `Sequence` is easy to make:

```
auto mySprite = Node::create();

// move to point 50,10 over 2 seconds
auto moveTo1 = MoveTo::create(2, Vec2(50,10));

// move from current position by 100,10 over 2 seconds
auto moveBy1 = MoveBy::create(2, Vec2(100,10));

// move to point 150,10 over 2 seconds
auto moveTo2 = MoveTo::create(2, Vec2(150,10));

// create a delay
auto delay = DelayTime::create(1);

mySprite->runAction(Sequence::create(moveTo1, delay, moveBy1, delay.clone(),
moveTo2, nullptr));
```

```
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.

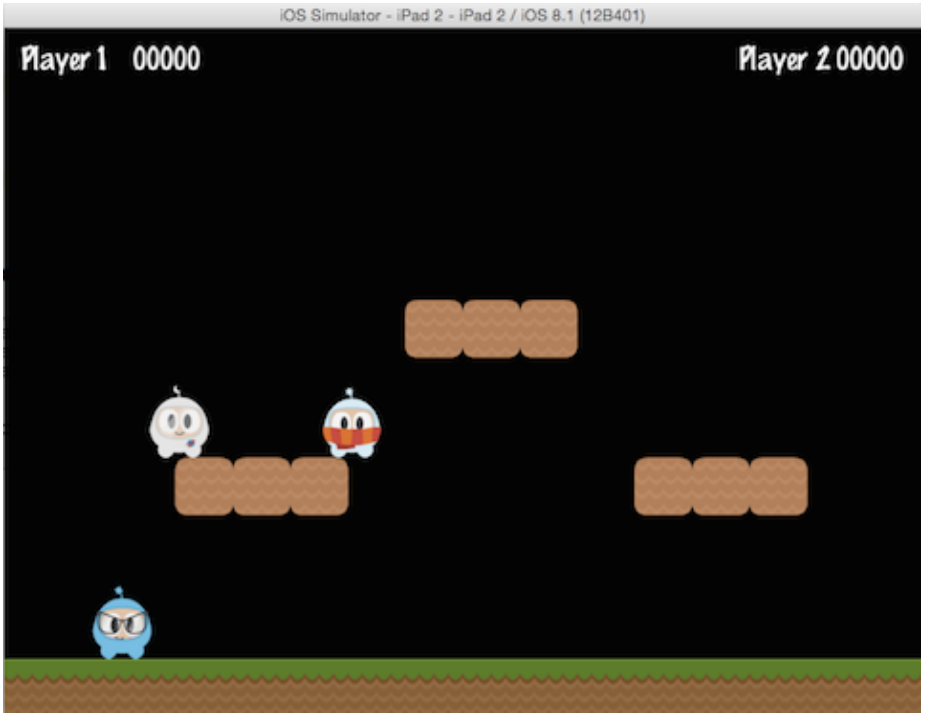
```
auto myNode = Node::create();

auto moveTo1 = MoveTo::create(2, Vec2(50,10));
auto moveBy1 = MoveBy::create(2, Vec2(100,10));
auto moveTo2 = MoveTo::create(2, Vec2(150,10));

myNode->runAction(Spawn::create(moveTo1, moveBy1, moveTo2, nullptr));
```

```
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:


```
auto myNode = Node::create();

// rotating by setting
myNode->setRotation(50);
```

```
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:


```
auto myNode = Node::create();

// scaling by setting
myNode->setScale(2.0); // scales uniformly by 2.0
```

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

```
// a string and a variable
string s = "My variable";
log("string is %s", s);

// a double and a variable
double dd = 42;
log("double is %f", dd);

// an integer and a variable
int i = 6;
log("integer is %d", i);

// a float and a variable
float f = 2.0f;
log("float is %f", f);

// a bool and a variable
bool b = true;
if (b == true)
    log("bool is true");
else
    log("bool is false");
```

And, as expected, if you prefer you can use **std::cout** in place of **log()**, however, **log()** might offer easier formatting of complex output.

```
// 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 are also there to help you with questions.

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.

```
auto mySprite = Sprite::create("mysprite.png");
```

```
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 `Sprite` uses 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.

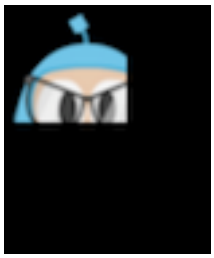
Creating a Sprite with a Rect

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**.

```
auto mySprite = Sprite::create("mysprite.png", Rect(0,0,40,40));
```

```
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.

```
auto mySprite = Sprite::create("mysprite.png");

auto mySprite = Sprite::create("mysprite.png", Rect(0,0,200,200));
```

```
var mySprite = new cc.Sprite(res.mySprite_png);

var mySprite = new cc.Sprite(res.mySprite_png, cc.rect(0,0,200,200));
```

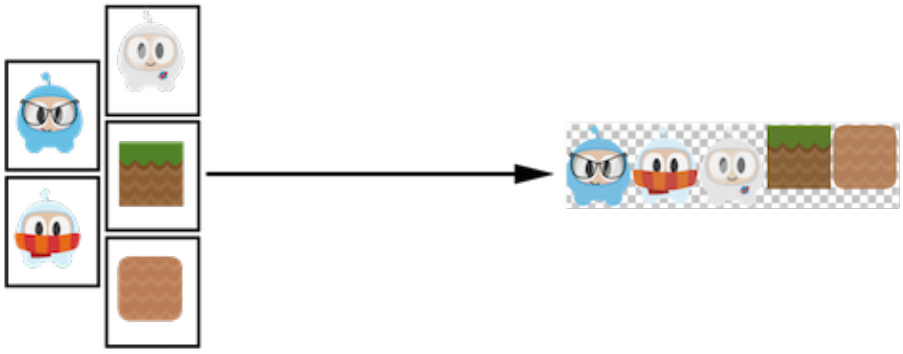
Creating a Sprite from a Sprite Sheet

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
auto spritecache = SpriteFrameCache::getInstance();

// the .plist file can be generated with any of the tools mentioned below
spritecache->addSpriteFramesWithFile("sprites.plist");
```

```
// 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, "mysprite" from the sprite sheet:
auto mysprite = Sprite::createWithSpriteFrameName("mysprite.png");
```

```
// 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.
auto newspriteFrame = SpriteFrameCache::getInstance()->getSpriteFrameByName("Blue_Front1.png");
auto newSprite = Sprite::createWithSpriteFrame(newspriteFrame);
```

```
// 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
- ShoeBox
- Texture Packer

- Zwoptex
- Sprite Sheet Packer

Sprite Manipulation

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

Given:

```
auto mySprite = Sprite::create("mysprite.png");
```

```
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)**.

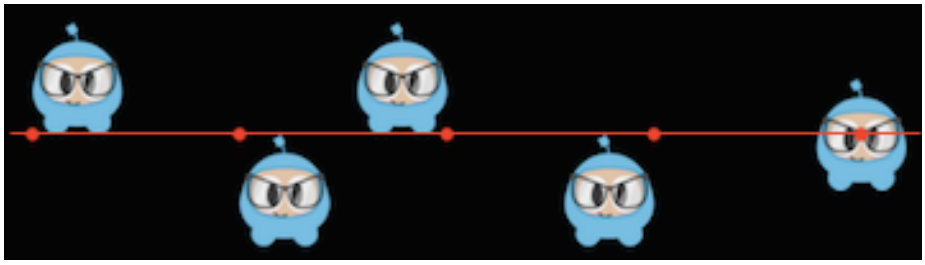
Setting the **anchor point** is easy:

```
// DEFAULT anchor point for all Sprites
mySprite->setAnchorPoint(0.5, 0.5);

// bottom left
mySprite->setAnchorPoint(0, 0);

// top left
mySprite->setAnchorPoint(0, 1);

// bottom right
mySprite->setAnchorPoint(1, 0);
```

```
// top right
mySprite->setAnchorPoint(1, 1);
```

```
// 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:

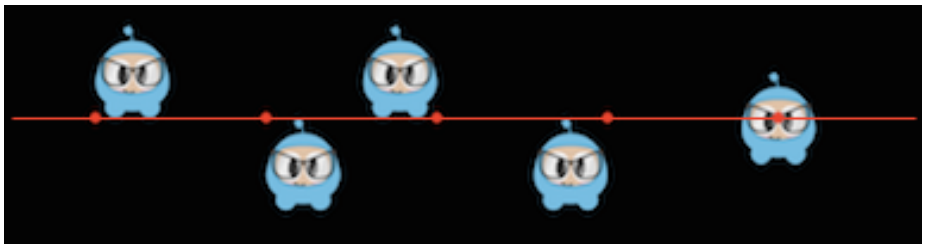


Sprite properties effected by anchor point

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.

```
// position a sprite to a specific position of x = 100, y = 200.
mySprite->setPosition(Vec2(100, 200));
```

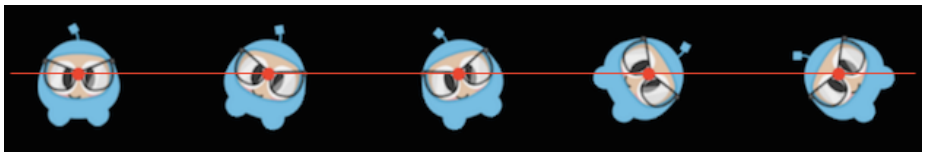
```
mySprite.setPosition(cc._p(100, 200));
```

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(20.0f);

// rotate sprite by -20 degrees
mySprite->setRotation(-20.0f);

// rotate sprite by +60 degrees
mySprite->setRotation(60.0f);

// rotate sprite by -60 degrees
mySprite->setRotation(-60.0f);
```

```
// 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));
```

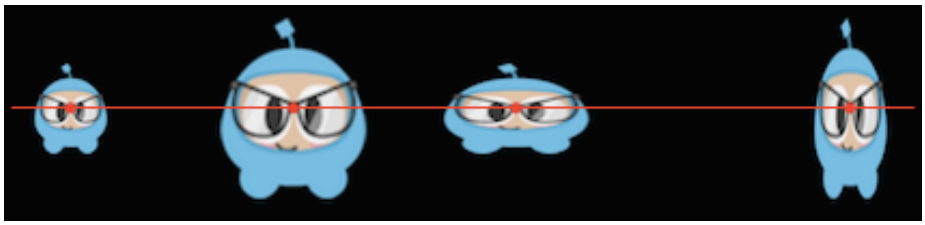


Scale

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(2.0);

// increases just X scale by 2.0
mySprite->setScaleX(2.0);

// increases just Y scale by 2.0
mySprite->setScaleY(2.0);
```

```
// 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(20.0f);

// adjusts the Y skew by 20.0
mySprite->setSkewY(20.0f);
```

```
// 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));
```

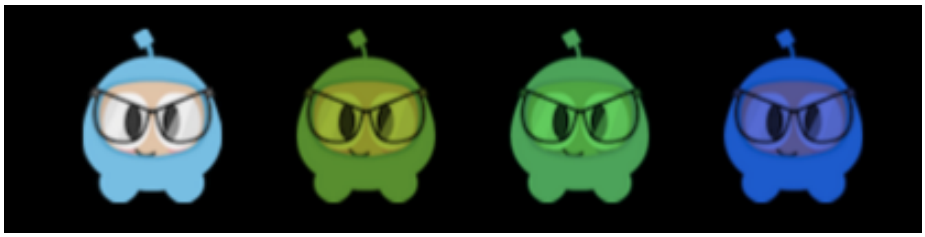
Sprite properties not affected by anchor point

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(Color3B::WHITE);

// Set the color by passing in a Color3B object.
mySprite->setColor(Color3B(255, 255, 255)); // Same as Color3B::WHITE

// 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);
```

```
// Set the opacity to 30, which makes this sprite 11.7% opaque.
// (30 divided by 256 equals 0.1171875...)
mySprite.setOpacity(30);
```

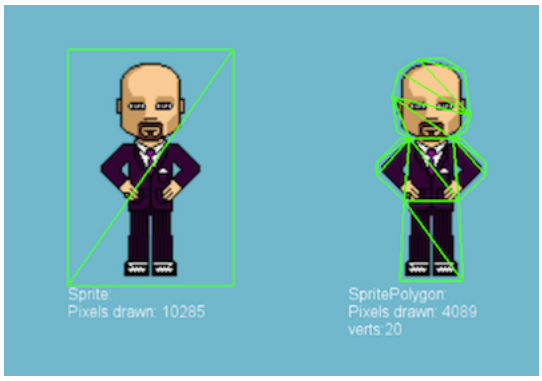
Polygon Sprite

A **Polygon Sprite** is also a `Sprite`, that is used to display a 2d image. However, unlike a normal `Sprite` object, 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.
auto pinfo = AutoPolygon::generatePolygon("filename.png");

// Create a sprite with polygon info.
auto sprite = Sprite::create(pinfo);
```

```
// Generate polygon info automatically.
var pinfo = cc.autopolygon.generatePolygon(res.mysprite_png);

// Create a sprite with polygon info.
var sprite = new cc.Sprite(pinfo);
```

Actions

`Action` objects are just like they sound. They make a `Node` perform a change to its properties. `Action` objects 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:

```
// Move sprite to position 50,10 in 2 seconds.
auto moveTo = MoveTo::create(2, Vec2(50, 10));
mySprite1->runAction(moveTo);

// Move sprite 20 points to right in 2 seconds
auto moveBy = MoveBy::create(2, Vec2(20,0));
mySprite2->runAction(moveBy);
```

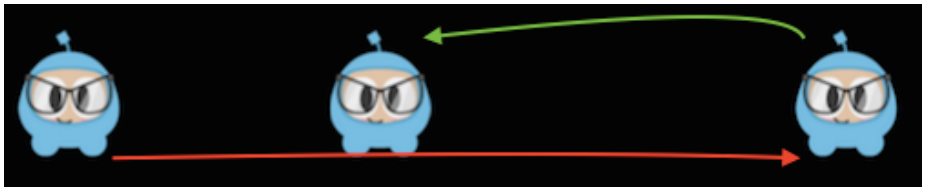
```
// Move sprite to position 50,10 in 2 seconds.
var moveTo = new cc.MoveTo(2, cc._p(50, 10));
mySprite1.runAction(moveTo);

// 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:

```
auto mySprite = Sprite::create("mysprite.png");
mySprite->setPosition(Vec2(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
auto moveBy = MoveBy::create(2, Vec2(500, mySprite->getPositionY()));

// 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.
auto moveTo = MoveTo::create(2, Vec2(300, mySprite->getPositionY()));

// Delay - create a small delay
auto delay = DelayTime::create(1);

auto seq = Sequence::create(moveBy, delay, moveTo, nullptr);

mySprite->runAction(seq);
```

```
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:

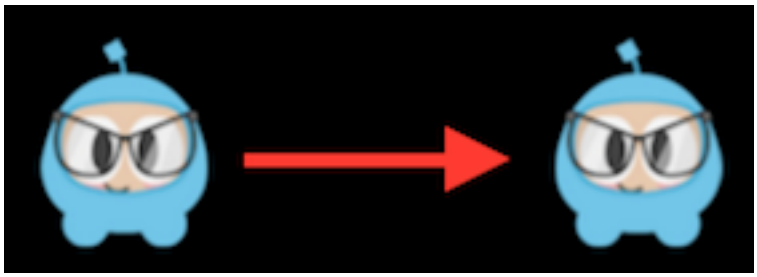
Move

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

```
auto mySprite = Sprite::create("mysprite.png");

// Move a sprite to a specific location over 2 seconds.
auto moveTo = MoveTo::create(2, Vec2(50, 0));

mySprite->runAction(moveTo);

// Move a sprite 50 pixels to the right, and 0 pixels to the top over 2 seconds.
auto moveBy = MoveBy::create(2, Vec2(50, 0));
```

```
mySprite->runAction(moveBy);
```

```
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.

```
auto mySprite = Sprite::create("mysprite.png");

// Rotates a Node to the specific angle over 2 seconds
auto rotateTo = RotateTo::create(2.0f, 40.0f);
mySprite->runAction(rotateTo);

// Rotates a Node clockwise by 40 degree over 2 seconds
auto rotateBy = RotateBy::create(2.0f, 40.0f);
mySprite->runAction(rotateBy);
```

```
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.

```
auto mySprite = Sprite::create("mysprite.png");

// Scale uniformly by 3x over 2 seconds
auto scaleBy = ScaleBy::create(2.0f, 3.0f);
mySprite->runAction(scaleBy);

// Scale X by 5 and Y by 3x over 2 seconds
auto scaleBy = ScaleBy::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleBy);

// Scale to uniformly to 3x over 2 seconds
auto scaleTo = ScaleTo::create(2.0f, 3.0f);
mySprite->runAction(scaleTo);

// Scale X to 5 and Y to 3x over 2 seconds
auto scaleTo = ScaleTo::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleTo);
```

```
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`

```
auto mySprite = Sprite::create("mysprite.png");

// fades in the sprite in 1 seconds
auto fadeIn = FadeIn::create(1.0f);
mySprite->runAction(fadeIn);

// fades out the sprite in 2 seconds
auto fadeOut = FadeOut::create(2.0f);
mySprite->runAction(fadeOut);
```

```
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.

```
auto mySprite = Sprite::create("mysprite.png");

// Tints a node to the specified RGB values
auto tintTo = TintTo::create(2.0f, 120.0f, 232.0f, 254.0f);
mySprite->runAction(tintTo);

// Tints a node BY the delta of the specified RGB values.
auto tintBy = TintBy::create(2.0f, 120.0f, 232.0f, 254.0f);
mySprite->runAction(tintBy);
```

```
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:

```
auto mySprite = Sprite::create("mysprite.png");

// now lets animate the sprite we moved
Vector<SpriteFrame*> animFrames;
animFrames.reserve(12);
animFrames.pushBack(SpriteFrame::create("Blue_Front1.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Front2.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Front3.png", Rect(0,0,65,81)));
```

```
animFrames.pushBack(SpriteFrame::create("Blue_Left1.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Left2.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Left3.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Back1.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Back2.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Back3.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Right1.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Right2.png", Rect(0,0,65,81)));
animFrames.pushBack(SpriteFrame::create("Blue_Right3.png", Rect(0,0,65,81)));

// create the animation out of the frames
Animation* animation = Animation::createWithSpriteFrames(animFrames, 0.1f);
Animate* animate = Animate::create(animation);

// run it and repeat it forever
mySprite->runAction(RepeatForever::create(animate));
```

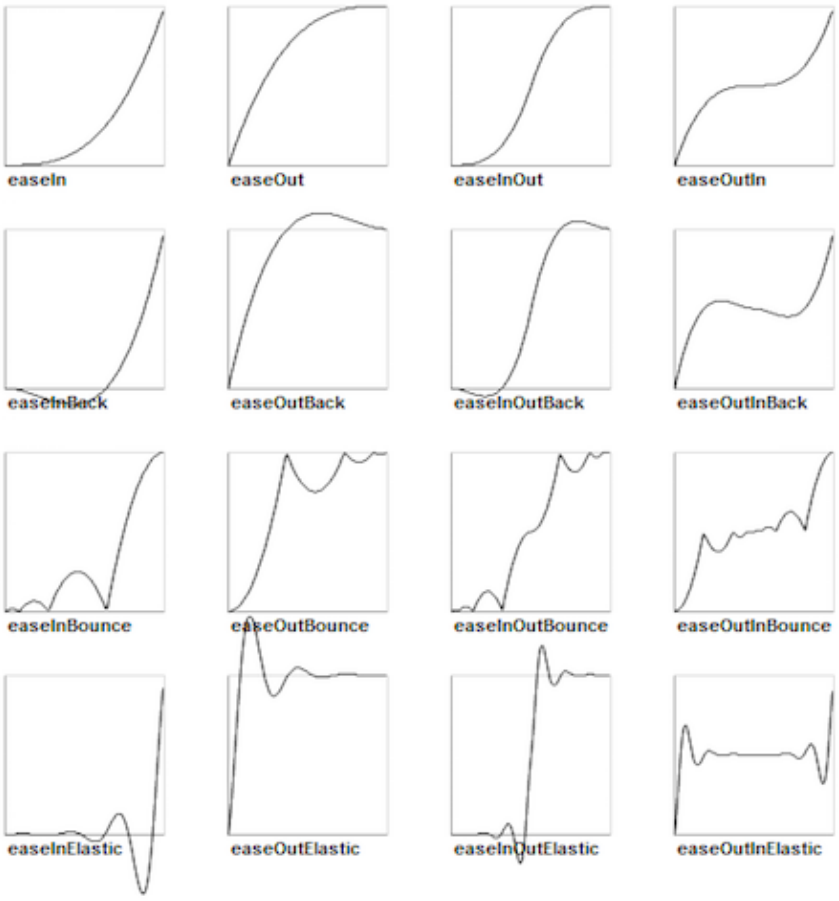
```
var mySprite = new Sprite(res.mysprite_png);

// now lets animate the sprite we moved.
var animFrames = new Array();
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.

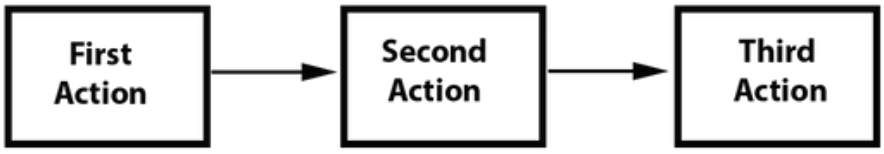
```
// create a sprite
auto mySprite = Sprite::create("mysprite.png");
```

```
// create a MoveBy Action to where we want the sprite to drop from.
auto move = MoveBy::create(2, Vec2(200, dirs->getVisibleSize().height -
 newSprite2->getContentSize().height));
auto move_back = move->reverse();

// create a BounceIn Ease Action
auto move_ease_in = EaseBounceIn::create(move->clone() );

// create a delay that is run in between sequence events
auto delay = DelayTime::create(0.25f);

// create the sequence of actions, in the order we want to run them
auto seq1 = Sequence::create(move_ease_in, delay, move_ease_in_back,
    delay->clone(), nullptr);

// run the sequence and repeat forever.
mySprite->runAction(RepeatForever::create(seq1));
```

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 `Action` objects, **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

```
auto mySprite = Sprite::create("mysprite.png");

// create a few actions.
auto jump = JumpBy::create(0.5, Vec2(0, 0), 100, 1);
```

```
auto rotate = RotateTo::create(2.0f, 10);

// create a few callbacks
auto callbackJump = CallFunc::create([](){
    log("Jumped!");
});

auto callbackRotate = CallFunc::create([](){
    log("Rotated!");
});

// create a sequence with the actions and callbacks
auto seq = Sequence::create(jump, callbackJump, rotate, callbackRotate, nullptr);

// run it
mySprite->runAction(seq);
```

So what does this `Sequence` action do?

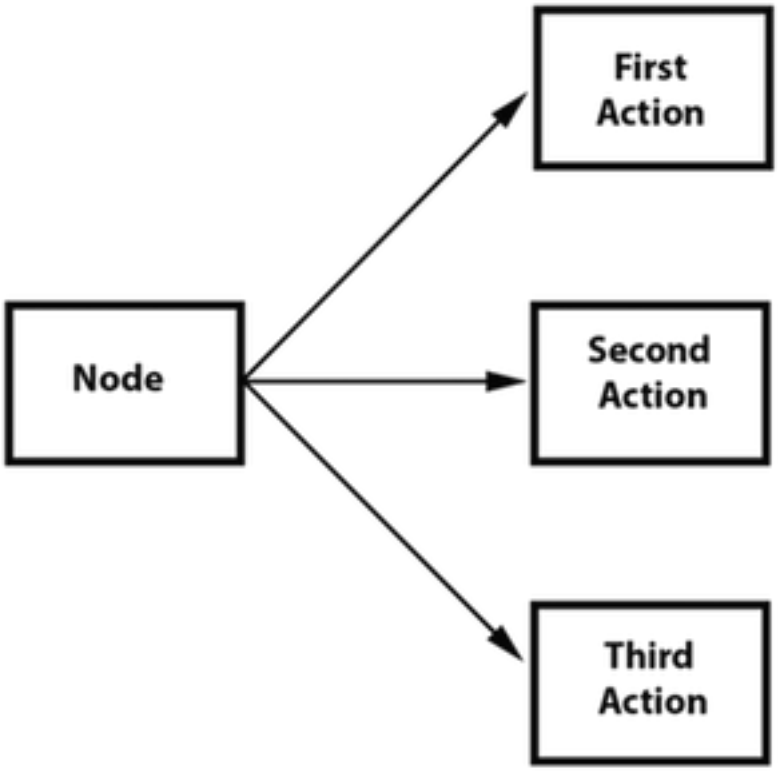
It will execute the following actions sequentially:

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:

```
// create 2 actions and run a Spawn on a Sprite
auto mySprite = Sprite::create("mysprite.png");

auto moveBy = MoveBy::create(10, Vec2(400,100));
auto fadeTo = FadeTo::create(2.0f, 120.0f);
```

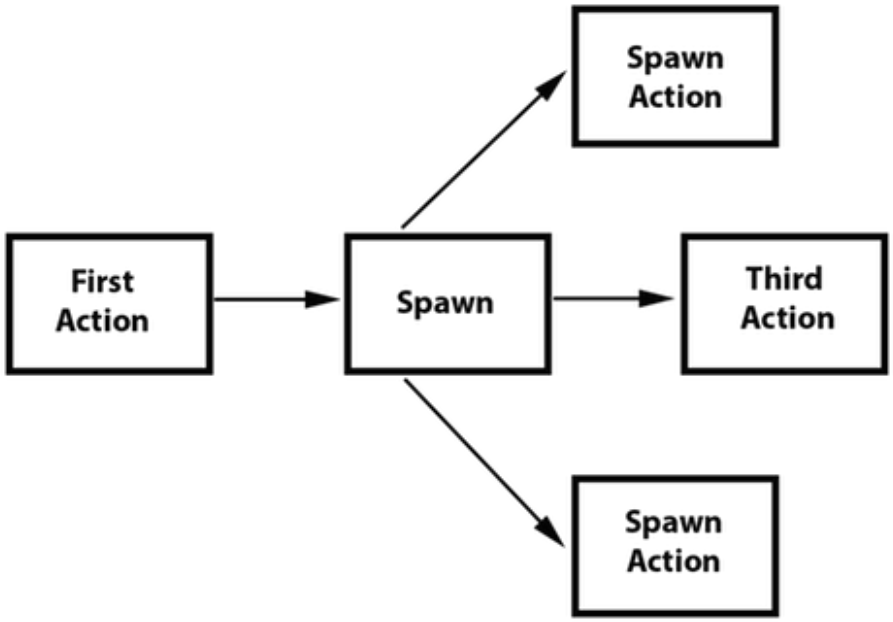
Using a `Spawn`:

```
// running the above Actions with Spawn.
auto mySpawn = Spawn::createWithTwoActions(moveBy, fadeTo);
mySprite->runAction(mySpawn);
```

and consecutive **runAction()** statements:

```
// running the above Actions with consecutive runAction() statements.
mySprite->runAction(moveBy);
mySprite->runAction(fadeTo);
```

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




```
// create a Sprite
auto mySprite = Sprite::create("mysprite.png");

// create a few Actions
auto moveBy = MoveBy::create(10, Vec2(400,100));
auto fadeTo = FadeTo::create(2.0f, 120.0f);
auto scaleBy = ScaleBy::create(2.0f, 3.0f);

// create a Spawn to use
auto mySpawn = Spawn::createWithTwoActions(scaleBy, fadeTo);

// tie everything together in a sequence
auto seq = Sequence::create(moveBy, mySpawn, moveBy, nullptr);

// run it
mySprite->runAction(seq);
```

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 `Action` of:

```
MoveBy::create(10, Vec2(400,100));
```

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:

```
MoveBy::create(10, Vec2(400,100));
```

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.

```
// create our Sprites
auto heroSprite = Sprite::create("herosprite.png");
auto enemySprite = Sprite::create("enemysprite.png");

// create an Action
auto moveBy = MoveBy::create(10, Vec2(400,100));

// run it on our hero
heroSprite->runAction(moveBy);

// run it on our enemy
enemySprite->runAction(moveBy); // oops, this will not be unique!
// uses the Actions current internal state as a starting point.
```

Correctly, using **clone()**!:

```
// create our Sprites
auto heroSprite = Sprite::create("herosprite.png");
auto enemySprite = Sprite::create("enemysprite.png");

// create an Action
auto moveBy = MoveBy::create(10, Vec2(400,100));

// run it on our hero
heroSprite->runAction(moveBy);

// run it on our enemy
enemySprite->runAction(moveBy->clone()); // correct! This will be unique
```

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.

```
// reverse a sequence, spawn or action
mySprite->runAction(mySpawn->reverse());
```

Most `Action` and `Sequence` objects are reversible!

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

```
// create a Sprite
auto mySprite = Sprite::create("mysprite.png");
mySprite->setPosition(50, 56);

// create a few Actions
auto moveBy = MoveBy::create(2.0f, Vec2(500,0));
auto scaleBy = ScaleBy::create(2.0f, 2.0f);
auto delay = DelayTime::create(2.0f);

// create a sequence
auto delaySequence = Sequence::create(delay, delay->clone(), delay->clone(),
delay->clone(), nullptr);

auto sequence = Sequence::create(moveBy, delay, scaleBy, delaySequence, nullptr);
```

```
// run it
newSprite2->runAction(sequence);

// reverse it
newSprite2->runAction(sequence->reverse());
```

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!

Building and Transitioning Scenes

What is a Scene?

A `Scene` is a container that holds `Sprites`, `Labels`, `Nodes` and other objects that your game needs. A `Scene` is responsible for running game logic and rendering the content on a per-frame basis. You need at least one `Scene` to start your game. You can think of this like a movie. The `Scene` is what is running and users see what is happening in real-time. You can have any number of `Scene` objects in your game and transition through them easily. Cocos2d-x provides **scene transitions** and you can even have **scene transitions** with cool effects.

Creating a Scene

It is very easy to create a `Scene`

```
auto myScene = Scene::create();
```

Remember the Scene Graph?

In **Chapter 2** of this guide we learned about a **scene graph** and how it affects the drawing of our game. The important thing to remember is that this defines the drawing order of the GUI elements. Also remember **z-order**!

A Simple Scene

Lets's build a simple `Scene`. Remember that Cocos2d-x uses a **right handed coordinate system**. This means that our *0,0* coordinate is at the bottom left corner of the screen/display. When you start positioning your game elements this is where you should start your calculations from. Let's create a simple `Scene` and add a few elements to it:

```
auto dirs = Director::getInstance();
Size visibleSize = dirs->getVisibleSize();

auto myScene = Scene::create();

auto label1 = Label::createWithTTF("My Game", "Marker Felt.ttf", 36);
label1->setPosition(Vec2(visibleSize.width / 2, visibleSize.height / 2));

myScene->addChild(label1);

auto sprite1 = Sprite::create("mysprite.png");
sprite1->setPosition(Vec2(100, 100));

myScene->addChild(sprite1);
```

When we run this code we shall see a simple `Scene` that contains a `Label` and a `Sprite`. It doesn't do much but it's a start.

Transitioning between Scenes

You might need to move between `Scene` objects in your game. Perhaps starting a new game, changing levels or even ending your game. Cocos2d-x provides a number of ways to do **scene transitions**.

Ways to transition between Scenes

There are many ways to transition through your **scenes**. Each has specific functionality. Let's go through them. Given:

```
auto myScene = Scene::create();
```

runWithScene() - use this for the first scene only. This is the way to start your games first `Scene`.

```
Director::getInstance()->runWithScene(myScene);
```

replaceScene() - replace a scene outright.

```
Director::getInstance()->replaceScene(myScene);
```

pushScene() - suspends the execution of the running scene, pushing it on the stack of suspended scenes. Only call this if there is a running scene.

```
Director::getInstance()->pushScene(myScene);
```

popScene() - This scene will replace the running one. The running scene will be deleted. Only call this if there is a running scene.

```
Director::getInstance()->popScene(myScene);
```

Transition Scenes with effects

You can add visual effects to your `Scene` transitions

```
auto myScene = Scene::create();

// Transition Fade
Director::getInstance()->replaceScene(TransitionFade::create(0.5, myScene, Color3B(0,255,255)));

// FlipX
Director::getInstance()->replaceScene(TransitionFlipX::create(2, myScene));

// Transition Slide In
Director::getInstance()->replaceScene(TransitionSlideInT::create(1, myScene) );
```

UI Components

Taking a look at the common apps you might use, I bet that you can spot UI widgets without necessarily knowing what they are. They aren't specific to games, every application probably uses a few widgets. What does **UI** stand for? What do **UI** widgets do? Oh so many questions!

Widgets, oh, my!

UI is an abbreviation that stands for *user interface*. You know, things that are on the screen. This include items like: **labels**, **buttons**, **menus**, **sliders** and **views**. Cocos2d-x provides a set of **UI** widgets to make it simple to add these controls to your projects. It may sound trivial, but a lot goes in to creating a core class like a `Label`. There are so many aspects of just this one.

Could you imagine having to write your own custom widget set? Don't worry, your needs are covered!

Label

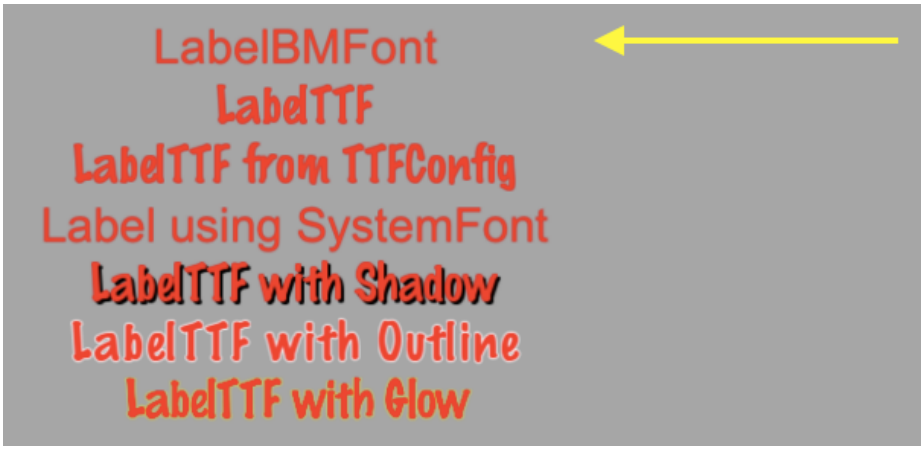
Cocos2d-x provides a `Label` object that can create labels using **true type**, **bitmap** or the built-in system font. This single class can handle all your `Label` needs.

Label BMFont

`BMFont` is a label type that uses a bitmap font. The characters in a bitmap font are made up of a matrix of **dots**. It is very fast and easy to use, but not scalable as it requires a separate font for each size character. Each character in a `Label` is a separate `Sprite`. This means that each character can be rotated, scaled, tinted, have a different **anchor point** and/or most any other property changed.

Creating a `BMFont` label requires two files: a **.fnt** file and an image representation of each character in **.png** format. If you are using a tool like **Glyph Designer** these files are created automatically for you. Creating a `Label` object from a **bitmap font**:

```
auto myLabel = Label::createWithBMFont("bitmapRed.fnt", "Your Text");
```



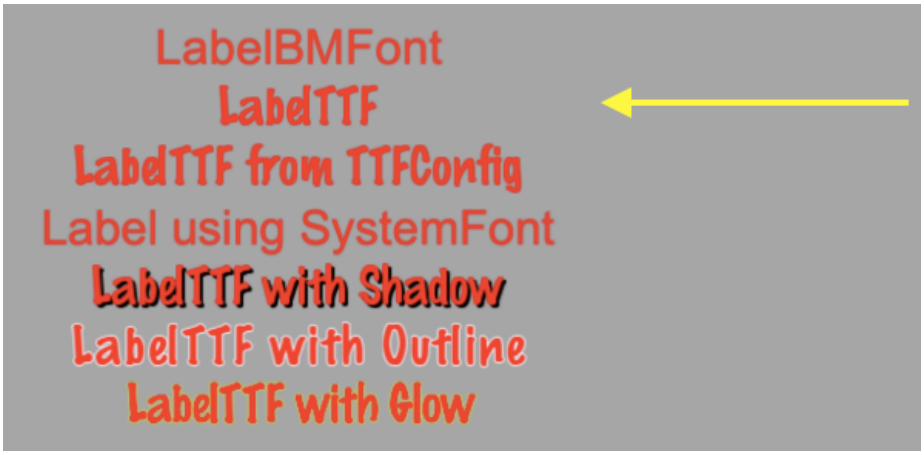
All of the characters in the string parameter should be found in the provided **.fnt** file, otherwise they won't be rendered. If you render a `Label` object and it is missing characters, make sure they exist in your **.fnt** file.

Label TTF

True Type Fonts are different from the **bitmap fonts** we learned about above. With **true type fonts** the outline of the font is rendered. This is convenient as you do not need to have a separate font file for each size and color you might wish to use. Creating a `Label` object that

uses a **true type font** is easy. To create one you need to specify a **.ttf** font file name, text string and a size. Unlike `BMFont`, `TTF` can render size changes without the need for a separate font files. Example, using a **true type font**:

```
auto myLabel = Label::createWithTTF("Your Text", "Marker Felt.ttf", 24);
```

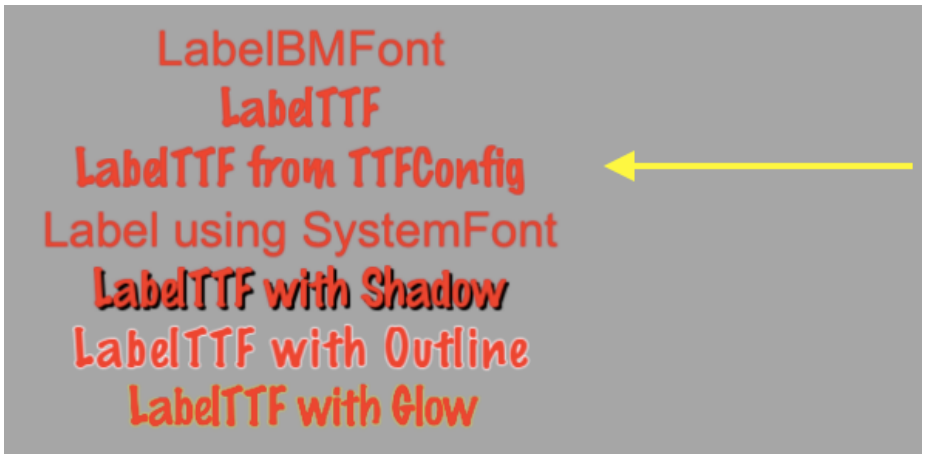


Although it is more flexible than a **bitmap font**, a *true type font* is slower to render and changing properties like the **font face** and **size** is an expensive operation.

If you need several `Label` objects from a **true type font** that all have the same properties you can create a `TTFConfig` object to manage them. A `TTFConfig` object allows you to set the properties that all of your labels would have in common. You can think of this like a *recipe* where all your `Label` objects will use the same ingredients.

You can create a `TTFConfig` object for your `Labels` in this way:

```
// create a TTFConfig files for labels to share
TTFConfig labelConfig;
labelConfig.fontFilePath = "myFont.ttf";
labelConfig.fontSize = 16;
labelConfig.glyphs = GlyphCollection::DYNAMIC;
labelConfig.outlineSize = 0;
labelConfig.customGlyphs = nullptr;
labelConfig.distanceFieldEnabled = false;

// create a TTF Label from the TTFConfig file.
auto myLabel = Label::createWithTTF(labelConfig, "My Label Text");
```

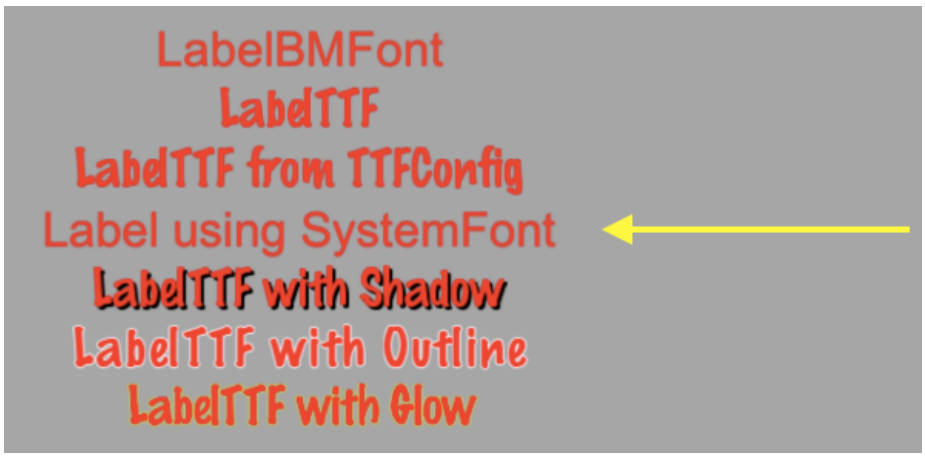


A `TTFConfig` can also be used for displaying Chinese, Japanese and Korean characters.

Label SystemFont

`SystemFont` is a label type that uses the default system font and font size. This is a font that is meant not to have its properties changed. You should think of it as **system font, system rules**. Creating a `SystemFont` label:

```
auto myLabel = Label::createWithSystemFont("My Label Text", "Arial", 16);
```



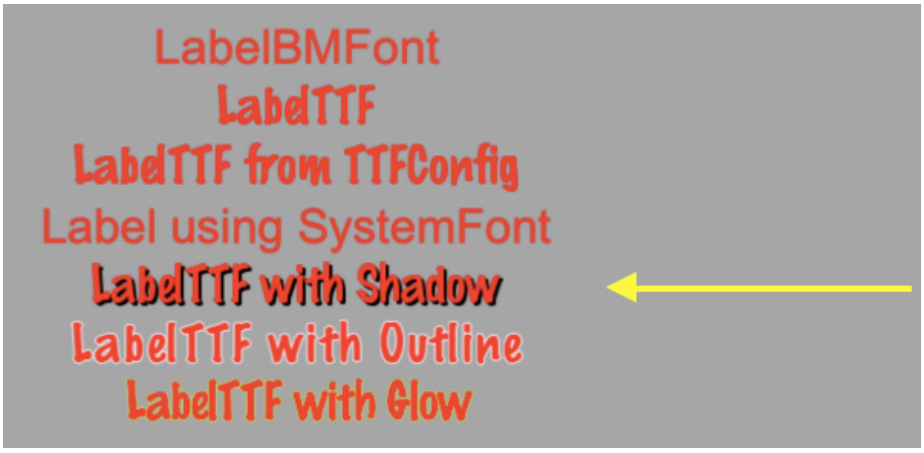
Label Effects

After you have your `Label` objects on screen you might want to make them a bit prettier. Perhaps they look flat or plain. Thankfully you don't have to create your own custom fonts! `Label`

objects can have effects applied to them. Not all `Label` objects support all effects. Some effects include **shadow**, **outline** and **glow**. You can apply one or more effects to a `Label` object easily:

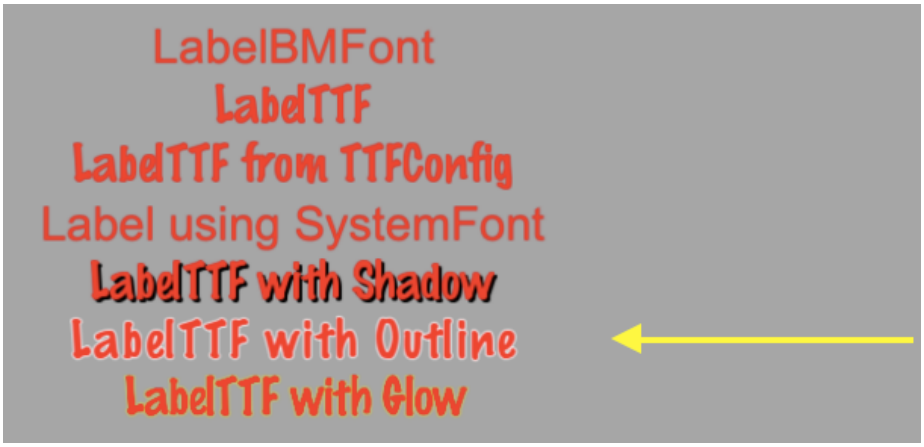
Label with a **shadow** effect:

```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// shadow effect is supported by all Label types
myLabel->enableShadow();
```



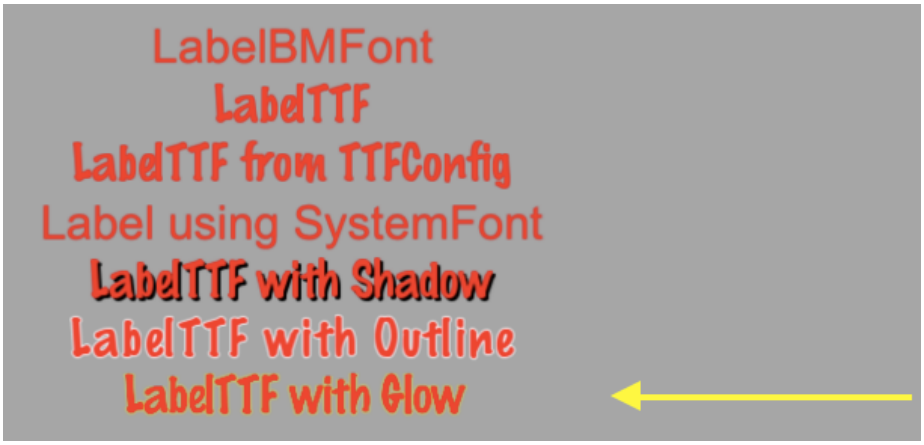
Label with a **outline** effect:

```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// outline effect is TTF only, specify the outline color desired
myLabel->enableOutline(Color4B::WHITE, 1));
```



Label with a **glow** effect:

```
auto myLabel = Label::createWithTTF("myFont.ttf", "My Label Text", 16);

// glow effect is TTF only, specify the glow color desired.
myLabel->enableGlow(Color4B::YELLOW);
```



Menu and Menu Items

We are all probably familiar with what a menu is. We see these in every application we use. In your game you would probably use a `Menu` object to navigate through game options. Menus often contain **buttons** like *Play*, *Quit*, *Settings* and *About*, but could also contain other `Menu` objects for a nested menu system. A `Menu` object is a special type of `Node` object. You can create an empty `Menu` object as a place holder for your **menu items**:

```
auto myMenu = Menu::create();
```

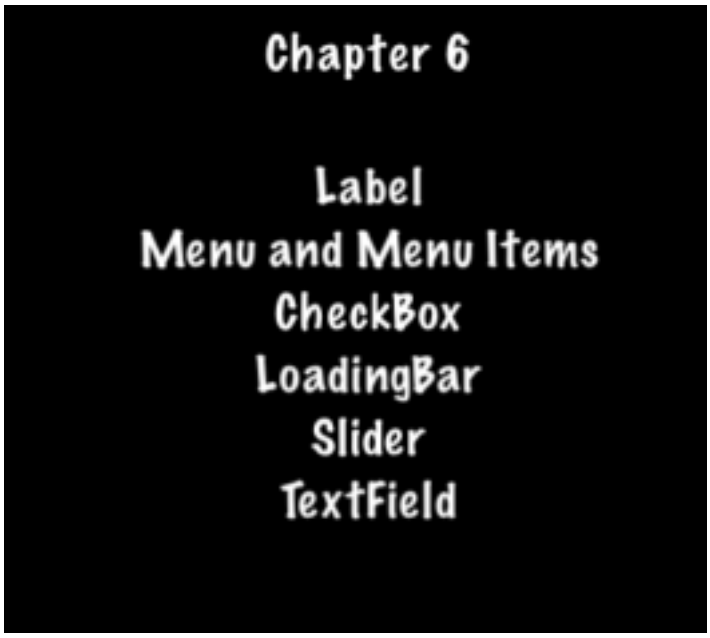
As we described options above of *Play*, *Quit*, *Settings* and *About*, these are your **menu items**. A `Menu` without **menu items** makes little sense. Cocos2d-x offers a variety of ways to create your **menu items** including by using a `Label` object or specifying an image to display. **Menu items** usually have two possible states, a **normal** and a **selected** state. When you tap or click on the **menu item** a **callback** is triggered. You can think of this as a chain reaction. You tap/click the **menu item** and it runs the code you specified. A `Menu` can have just a single item or many items.

```
// creating a menu with a single item

// create a menu item by specifying images
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));

auto menu = Menu::create(closeItem, NULL);
this->addChild(menu, 1);
```

A menu can also be created by using a **vector** of `MenuItem` objects:

```
// creating a Menu from a Vector of items
Vector<MenuItem*> MenuItems;

auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));

MenuItems.pushBack(closeItem);

/* repeat for as many menu items as needed */

auto menu = Menu::createWithArray(MenuItems);
this->addChild(menu, 1);
```

If you run the sample code for this chapter you will see a `Menu` containing `Label` objects for `MenuItems`:



Lambda functions as Menu callbacks

Above we just learned that when you click a **menu item** it triggers a **callback**. C++11 offers **lambda** functions and therefore Cocos2d-x takes full advantage of them! A **lambda** function is a function you write inline in your source code. **Lambdas** are also evaluated at runtime instead of compile time.

A simple **lambda**:

```
// create a simple Hello World lambda
auto func = [] () { cout << "Hello World"; };

// now call it someplace in code
func();
```

Using a **lambda** as a `MenuItem` callback:

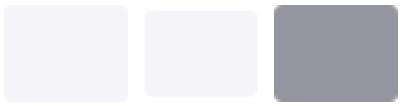
```
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
[&](Ref* sender){
    // your code here
});
```

Buttons

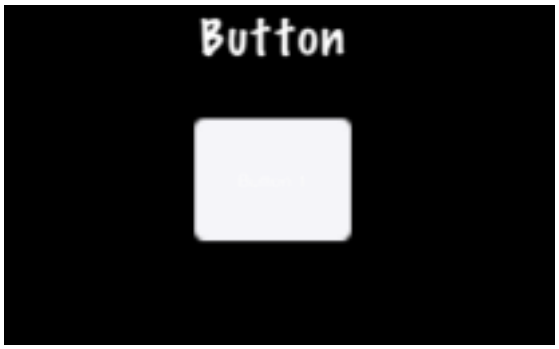
I doubt that we need to explain buttons much. We all know them as those things we click on to make something happen in our games. Perhaps you might use a button to change **scenes** or to add `Sprite` objects into your game play. A button intercepts a touch event and calls a pre-defined callback when tapped. A `Button` has a **normal** and **selected** state. The appearance of the `Button` can change based upon it's state. Creating a `Button` and defining its **callback** is simple:

```
#include "ui/CocosGUI.h"

auto button = Button::create("normal_image.png", "selected_image.png", "disabled_image.png");

button->setTitleText("Button Text");

button->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "Button 1 clicked" << std::endl;
                        break;
                default:
                        break;
        }
});

this->addChild(button);
```

As you can see in the above example we specify a *.png* image for each of the possible states the button can be in. A `Button` is made up of 3 graphics that might look like this:



On screen a `Button` might look like this:



CheckBox

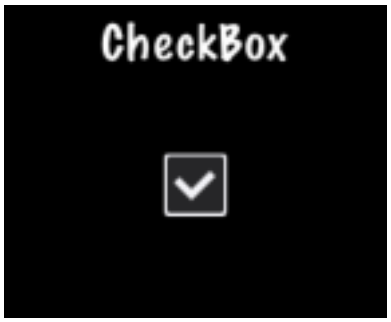
We are all used to filling out **checkboxes** on paper forms like job applications and rental agreements. You can also have **checkboxes** in your games. Perhaps, you want to have the ability for your player to make a simple **yes** or **no** choice. You might also hear this referred to as a **binary** choice (0 and 1). A `CheckBox` permits the user to make this type of choice. There are 5 different **states** a `Checkbox` can have: **normal**, **selected** and **disabled**. It is simple to create a `CheckBox`:

```
#include "ui/CocosGUI.h"

auto checkbox = CheckBox::create("check_box_normal.png",
                                 "check_box_normal_press.png",
                                 "check_box_active.png",
                                 "check_box_normal_disable.png",
                                 "check_box_active_disable.png");

checkbox->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "checkbox 1 clicked" << std::endl;
                        break;
                default:
                        break;
        }
```

```
});

this->addChild(checkbox);
```

As you can see in the above example we specify a *.png* image for each of the possible states the `Checkbox` can be in. Since there are 5 possible states that a `CheckBox` can be in, it is up 5 graphics, one for each of its states. Example graphics:

On screen a `Checkbox` might look like this:



LoadingBar

Have you ever played a game where you had to wait while it loaded up all the content it needed? It probably showed you a bar, filling in as it made progress accomplishing its task. This is often referred to as a **progress bar**, **status bar** or a **loading bar**. Creating a `LoadingBar`:

```
#include "ui/CocosGUI.h"

auto loadingBar = LoadingBar::create("LoadingBarFile.png");

// set the direction of the loading bars progress
loadingBar->setDirection(LoadingBar::Direction::RIGHT);

this->addChild(loadingBar);
```

In the above example a **loading bar** is created and we set the direction it should fill towards as progress is made. In this case to the right direction. However, you probably need to change the percentage of the `LoadingBar`. This is easily done:

```
#include "ui/CocosGUI.h"

auto loadingBar = LoadingBar::create("LoadingBarFile.png");
loadingBar->setDirection(LoadingBar::Direction::RIGHT);
```

```
// something happened, change the percentage of the loading bar
loadingBar->setPercent(25);

// more things happened, change the percentage again.
loadingBar->setPercent(35);

this->addChild(loadingBar);
```

As you can see in the above example we specify a *.png* image for the `LoadingBar` objects texture:

On screen a `LoadingBar` might look like this:

Slider

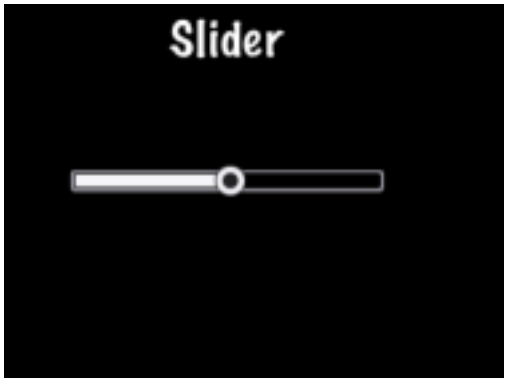
Sometimes it is necessary to change a value slightly. Perhaps you have a character and you want to allow the player to adjust the strength of attacking an enemy. A `Slider` allows users to set a value by moving an indicator. To create a `Slider`:

```
#include "ui/CocosGUI.h"

auto slider = Slider::create();
slider->loadBarTexture("Slider_Back.png"); // what the slider looks like
slider->loadSlidBallTextures("SliderNode_Normal.png", "SliderNode_Press.png", "SliderNode_Disable
slider->loadProgressBarTexture("Slider_PressBar.png");

slider->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
        switch (type)
```

```
        {
                case ui::Widget::TouchEventType::BEGAN:
                        break;
                case ui::Widget::TouchEventType::ENDED:
                        std::cout << "slider moved" << std::endl;
                        break;
                default:
                        break;
        }
});

this->addChild(slider);
```

As you can see in the above example we specify a *.png* image for each of the possible states the slider can be in. A `Slider` is made up of 5 graphics that might look like this:

On screen a `Slider` might look like this:



TextField

What if you wanted the player of your game to type in a special name to call the main character? Where would they type it into? Yes, a **text field**, of course. A `TextField` widget is used for inputting text. It supports touch event, focus, percent positioning and percent content size. To create a `TextField` widget:

```
#include "ui/CocosGUI.h"

auto textField = TextField::create("","Arial",30);

textField->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
```

```
                std::cout << "editing a TextField" << std::endl;
});

this->addChild(textField);
```

In this example a `TextField` is created and a **callback** specified.

`TextField` objects are versatile and can meet all of your input needs. Would you like the user to enter a secret password? Do you need to limit the number of characters a user can input? `TextField` objects have this all built-it and much more! Let's take a look at an example:

```
#include "ui/CocosGUI.h"

auto textField = TextField::create("","Arial",30);

// make this TextField password enabled
textField->setPasswordEnabled(true);

// set the maximum number of characters the user can enter for this TextField
textField->setMaxLength(10);

textField->addTouchEventListener([&](Ref* sender, Widget::TouchEventType type){
                std::cout << "editing a TextField" << std::endl;
});

this->addChild(textField);
```

On screen a `TextField` might look like this:

When you are editing a `TextField`, the onscreen keyboard comes up:

Other Node Types

You are using `Sprite`, `Label` and `Action` objects in your game and it is making progress. Besides the basic `Node` types described in previous chapters, Cocos2d-x also provides more advanced `Node` types to help build special functionality. Perhaps you want to make a **tile-based** game? Or maybe a **2d side scroller**? Or maybe you want to add particle effects to your game? Cocos2d-x provides `Node` objects to help you accomplish these goals!

TileMap

TileMaps are maps made up of **tiles**. Each *tile* can have independent behavior. **TileMaps** are stored in an XML-based map format called `TML`. `TMX` was originally designed for tile-based maps but is also suitable for more generic game levels due to its support for various object types. `TMX` objects are easy to create:

```
// reading in a tiled map.
auto map = TMXTiledMap::create("TileMap.tmx");
addChild(map, 0, 99); // with a tag of '99'
```

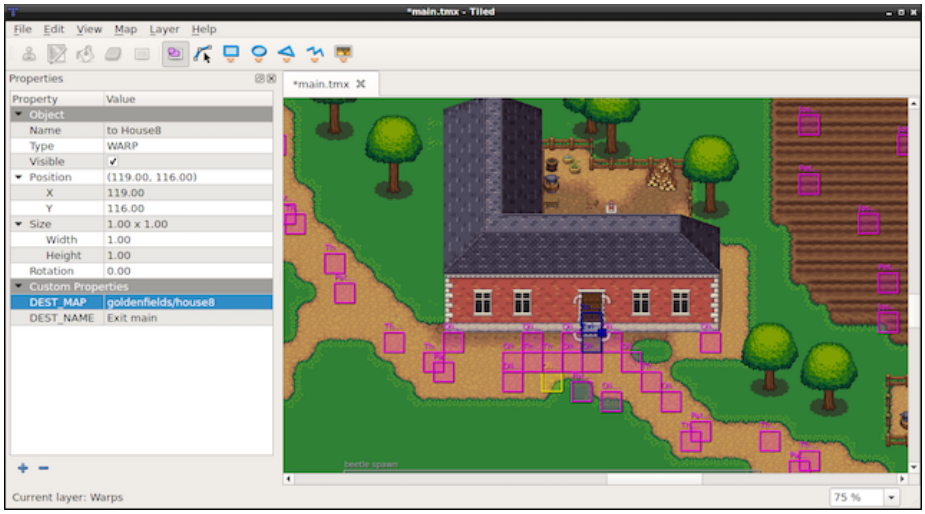
Tile-based maps can have many layers, determined by a **z-order**. You can access a specific layer by it's name:

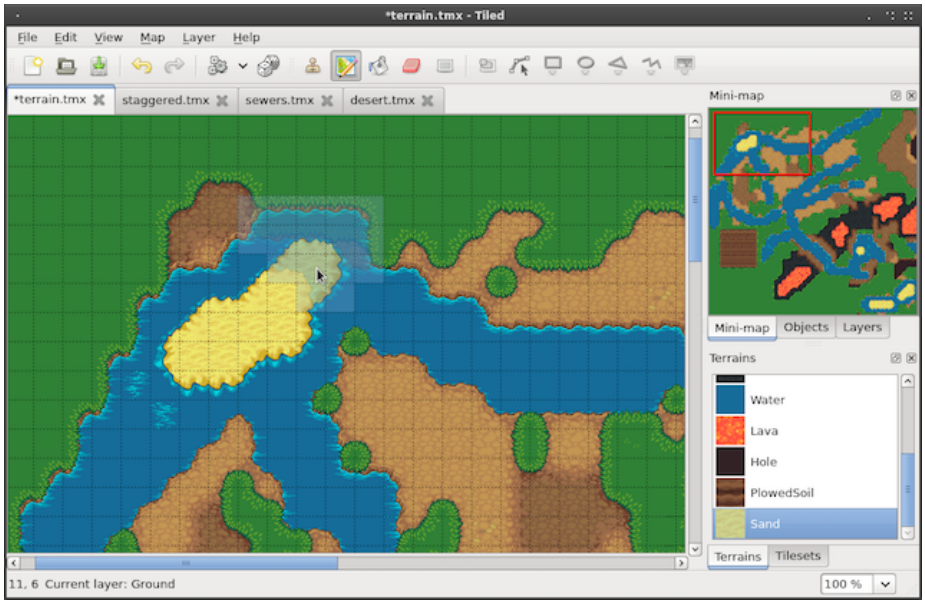
```
// how to get a specific layer
auto map = TMXTiledMap::create("TileMap.tmx");
auto layer = map->getLayer("Layer0");
auto tile = layer->getTileAt(Vec2(1, 63));
```

Each tile has a unique position and id. This makes it very easy to cherry pick specific tiles. You can access any tile by its id:

```
// to obtain a specific tiles id
unsigned int gid = layer->getTileGIDAt(Vec2(0, 63));
```

Example tiled-map layouts:

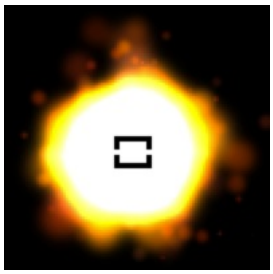




How do you make a tiled-map? There are many tools that do this. Tiled is a popular tool. It is actively developed and has a great user community. The screen-shots above are actual **Tiled** projects.

Particle System

Perhaps your game needs effects like burning fire, spell casting visuals or explosions. How would you make such complex effects? Is it even possible? Yes, it is. Using a **particle system**. The term *particle system* refers to a computer graphics technique that uses a large number of very small sprites or other graphic objects to simulate certain kinds of **fuzzy** phenomena, which are otherwise very hard to reproduce with conventional rendering techniques. Some realistic examples might include highly chaotic systems, natural phenomena, or processes caused by chemical reactions. Here are a few examples of **particle effects**:



Tools for creating Particle Effects

Even though you can always create *particle effects* by hand, massaging each property to your liking. There are several third party tools for creating *particle effects*. A few of these tools are:

1. Particle Designer: A very powerful particle effects editor on Mac
2. V-play particle editor: A cross-platform particle editor for Cocos2d-x
3. Particle2dx: An online web particle designer

These tools usually export a `.plist` file that you can read in with Cocos2d-x to use your creation inside your game. Just like with all of the other classes we have worked with so far we use the `create()` method:

```
// create by plist file
auto particleSystem = ParticleSystem::create("SpinningPeas.plist");
```

Built-In Particle Effects

Are you ready to add *particle effects* to your game? We hope so! Are you not yet comfortable with creating custom *particle effects*? For ease of convenience there are a number of built-in *particle effects* that you can choose from. Take a look at this list:

-ParticleFire: Point particle system. Uses Gravity mode.

-ParticleFireworks: Point particle system. Uses Gravity mode.

-ParticleSun: Point particle system. Uses Gravity mode.

-ParticleGalaxy: Point particle system. Uses Gravity mode.

-ParticleFlower: Point particle system. Uses Gravity mode.

-ParticleMeteor: Point particle system. Uses Gravity mode.

-ParticleSpiral: Point particle system. Uses Gravity mode.

-ParticleExplosion: Point particle system. Uses Gravity mode.

-ParticleSmoke: Point particle system. Uses Gravity mode.

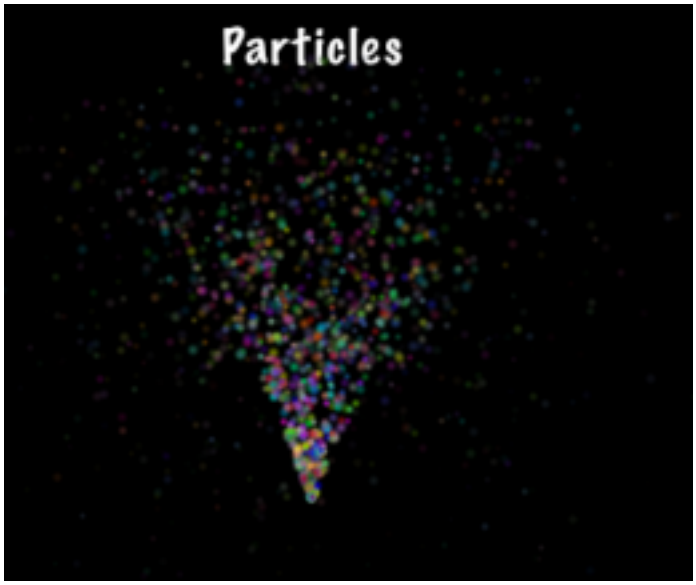
-ParticleSnow: Point particle system. Uses Gravity mode.

-ParticleRain: Point particle system. Uses Gravity mode.

Using `ParticleFireworks` as an example, you can use the built-in effects easily:

```
auto emitter = ParticleFireworks::create();

addChild(emitter, 10);
```

The result is a *particle effect* that looks something like:



But what do you do if your **particle effect** isn't quite the way you want? That's right, you can manually manipulate it! Let's take the same fireworks example above and manipulate it even further by manually changing its properties:

```
auto emitter = ParticleFireworks::create();

// set the duration
emitter->setDuration(ParticleSystem::DURATION_INFINITY);

// radius mode
```

```
emitter->setEmitterMode(ParticleSystem::Mode::RADIUS);

// radius mode: 100 pixels from center
emitter->setStartRadius(100);
emitter->setStartRadiusVar(0);
emitter->setEndRadius(ParticleSystem::START_RADIUS_EQUAL_TO_END_RADIUS);
emitter->setEndRadiusVar(0);    // not used when start == end

addChild(emitter, 10);
```

Parallax

A `Parallax` Node is a special `Node` type that simulates a **parallax scroller**. What did you say? A *para*.. what? Yes, **parallax** Simply put you can consider a `ParallaxNode` to be a **special effect** that makes it appear that the position or direction of an object appears to differ when viewed from different positions. Simple every day examples include looking through the viewfinder and the lens of a camera. You can think of many games that function this way, Super Mario Bros being a classic example. `ParallaxNode` objects can be moved around by a `Sequence` and also manually by mouse, touch, accelerometer or keyboard events.

Parallax nodes are a bit more complex than regular nodes. Why? Because they require the use of multiple nodes to function. A `ParallaxNode` cannot function by itself. You need at least 2 other `Node` objects to complete a `ParallaxNode`. As usual, in true Cocos2d-x fashion, a `ParallaxNode` is easy to create:

```
// create ParallaxNode
auto paraNode = ParallaxNode::create();
```

Since you need multiple `Node` objects, they too are easily added:

```
// create ParallaxNode
auto paraNode = ParallaxNode::create();

// background image is moved at a ratio of 0.4x, 0.5y
paraNode->addChild(background, -1, Vec2(0.4f,0.5f), Vec2::ZERO);

// tiles are moved at a ratio of 2.2x, 1.0y
paraNode->addChild(middle_layer, 1, Vec2(2.2f,1.0f), Vec2(0,-200) );

// top image is moved at a ratio of 3.0x, 2.5y
paraNode->addChild(top layer, 2, Vec2(3.0f,2.5f), Vec2(200,800) );
```

OK, looks and feels familiar, right? Notice a few items! Each `Node` object that was added is given a unique **z-order** so that they stack on top of each other. Also notice the additional 2 `Vec2` type parameters in the **addChild()** call. These are the **ratio** and **offset**. These parameters can be thought of as the **ratio** of speed to the parent `Node`.

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

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

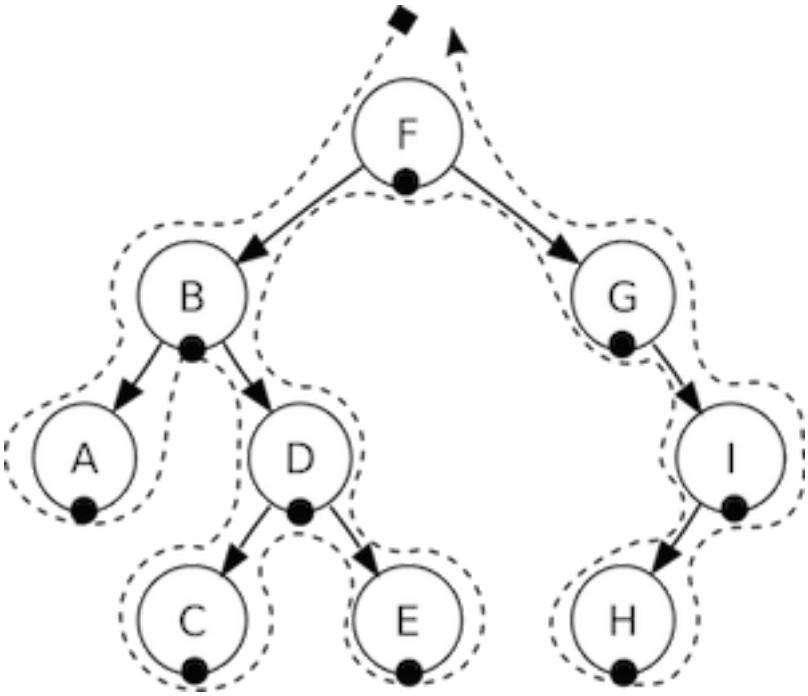
FixedPriority vs SceneGraphPriority

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.

3D

You probably started with Cocos2d-x and know it as a 2D game engine. Starting with version 3, 3D features are being added and refined. 3D gaming is a huge market and Cocos2d-x is adding all the features you need for 3D development. 3D development might be new to you and use some terminology that you are unfamiliar with. There are also additional software tools that you need to become familiar with. Let's jump right in and get our feet wet.

Terminology

When using 3D, there are some commonly used terms that you should be familiar with:

- **Mesh** - vertices that construct a shape and texture with which you are rendering.
- **Model** - an object that can be rendered. It is a collection of meshes. In our engine `Sprite3D`.
- **Texture** - All surfaces and vertices of a 3D model can be mapped to a texture. In most cases you will have multiple textures per model, unwrapped in a texture atlas.
- **Camera** - Since a 3D world is not flat, you need to set a camera to look at it. You get different scenes with different camera parameters.

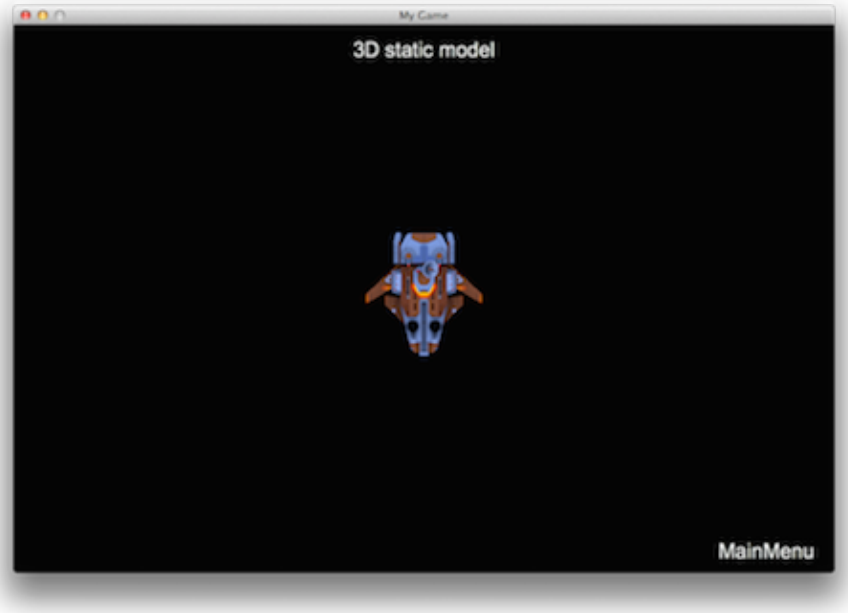
- **Light** - Lightening is applied to make scenes look realistic. To make an object look real, the color should change according to the light. When you face the light it is bright and the opposite is dark. *Lightening* an object means computing the object's color according to the light.

Sprite3D

Just like 2D games, 3D games also have `Sprite` objects. `Sprite` objects are a core foundation of any game. One of the main differences between `Sprite` and `Sprite3D` is `Sprite3D` objects have 3 axes it can be positioned on: **x**, **y** and **z**. `Sprite3D` works in many ways just like a normal `Sprite`. It is easy to load and display a `Sprite3D` object:

```
auto sprite = Sprite3D::create("boss.c3b"); //c3b file, created with the FBX-converter
sprite->setScale(5.f); //sets the object scale in float
sprite->setPosition(Vec2(200,200)); //sets sprite position
scene->addChild(sprite,1); //adds sprite to scene, z-index: 1
```

This creates and positions a `Sprite3D` object from `.c3b` file. Example:



Now let's rotate the model in a loop. For this we will create an action and run it:

```
//rotate around the X axis
auto rotation = RotateBy::create(15, Vec3(0, 360, 0));
//our sprite object runs the action
sprite->runAction(RepeatForever::create(rotation));
```

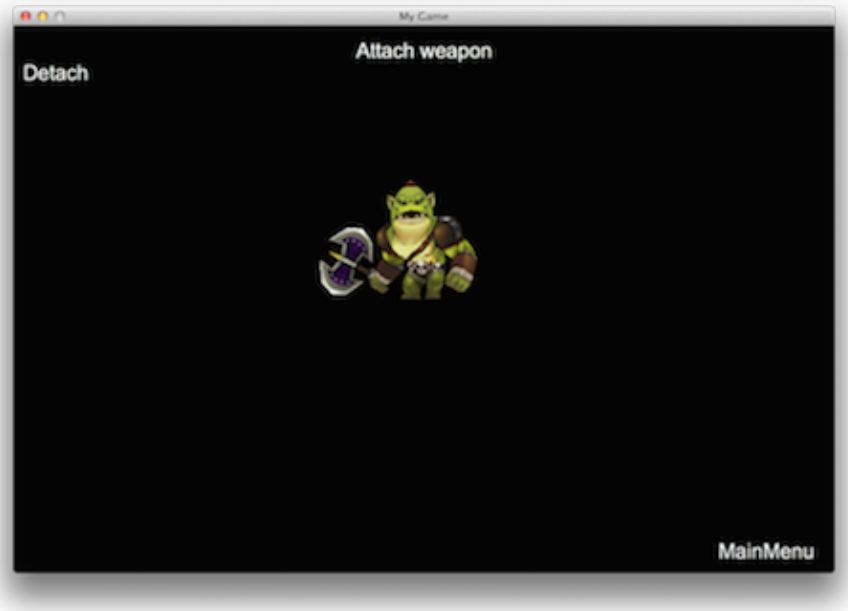
To set an anchor point on the `Sprite` or `Sprite3D` use:

```
sprite->setAnchorPoint(Point(0.0f,0.0f));
```

Attaching 3D models to Sprite3D objects.

Recall above that a 3D model is a collection of **meshes**. You can attach 3D models to other 3D models to create rich effects. An example would be adding a weapon to a character. To do this you need to find the attachment point where the weapon is to be added. For this use the **getAttachNode(attachment_point_name)** function. And then we just add the new model to the attachment point as a child with **addChild()**. You can think of this as combining multiple simpler 3D models to create more complex models. For example adding a model to a `Sprite3D` object:

```
auto sp = Sprite3D::create("axe.c3b");
sprite->getAttachNode("Bip001 R Hand")->addChild(sp);
```



Swap 3D Model

When doing 3D development you might want to make dynamic changes to your model. Perhaps due to power-ups, costume changes or visual cues to notify the user about status changes of your model. If your 3D model is comprised from **meshes** you can access the **mesh data** using **getMeshByIndex()** and **getMeshByName()**. Using these functions it is possible to achieve

effects like swapping a weapon or clothing for a character. Let's take a look at an example of a girl wearing a coat:

We can change the coat that the girl is wearing by changing the visibility of the **mesh** objects we are using. The following example demonstrates how to do this:

```
auto sprite = Sprite3D::create("ReskinGirl.c3b");

// display the first coat
auto girlTop0 = sprite->getMeshByName("Girl_UpperBody01");
girlTop0->setVisible(true);

auto girlTop1 = sprite->getMeshByName("Girl_UpperBody02");
girlTop1->setVisible(false);

// swap to the second coat
girlTop0->setVisible(false);
girlTop1->setVisible(true);
```

The results:

Animation

`Sprite3D` objects are essential to our game! We have learned how to manipulate them. However, we might want a more rich experience. Enter animation! To run a 3d animation, you can use the `Animation3D` and `Animate3D` objects. You then create an `Animate3D` action using the `Animation3D` object. Example:

```
// the animation is contained in the .c3b file
auto animation = Animation3D::create("orc.c3b");

// creates the Action with Animation object
auto animate = Animate3D::create(animation);

// runs the animation
sprite->runAction(RepeatForever::create(animate));
```

Run the example **Programmer Guide Sample** code to see this in action! Please keep in mind that 3D animations are exactly the same concepts as 2D. Please refer to Chapter 4 in this guide.

Multiple animations

What do you do when you want to run multiple **animations** at the same time? Using both the **animation start time** and **animation length** parameters you can create multiple animations. The unit for both parameters is seconds. Example:

```
auto animation = Animation3D::create(fileName);

auto runAnimate = Animate3D::create(animation, 0, 2);
sprite->runAction(runAnimate);

auto attackAnimate = Animate3D::create(animation, 3, 5);
sprite->runAction(attackAnimate);
```

In the above example there are two animations that get run. The first starts immediately and lasts for *2 seconds*. The second starts after *3 seconds* and lasts for *5 seconds*.

Animation speed

The **speed** of the animation is a positive integer for forward while a negative speed would be reverse. In this case the speed is set to *10*. This means that this animation can be considered to be *10* seconds in length.

Animation blending

When using multiple animations, **blending** is automatically applied between each animation. The purpose of **blending** is to create a smooth transition between effects. Given two animations, A and B, the last few frames of animation A and the first few frames of animation B overlap to make the change in animation look natural.

The default transition time is 0.1 seconds. You can set the transition time by using **Animate3D::setTransitionTime**.

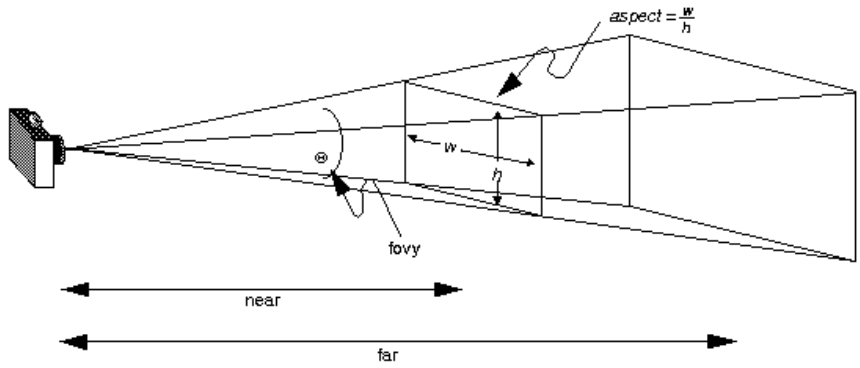
Cocos2d-x only supports **linear interpolation** between keyframes. This fills in **gaps** in the curve to ensure a smooth path. If you use other interpolation methods in the model production, our built-in tool, **fbx-conv** will generate additional keyframes to compensate. This compensation is completed in accordance with the target frame. For more information on **fbx-conv** please refer to the section discussing it at the end of this chapter.

Camera

Camera objects are an important aspect of 3D development. Since a 3D world is not flat you need to use a `Camera` to look at it and navigate around it. Just like when you are watching a movie and the scene pans to the left or right. This same concept is applied when using a `Camera` object. The `Camera` object inherits from `Node` and therefore supports most of the

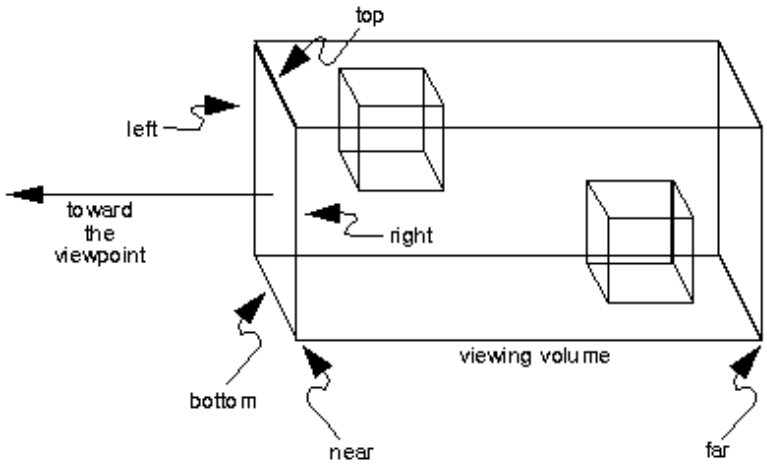
same `Action` objects. There are two types of `Camera` objects: **perspective camera** and **orthographic camera**.

The **perspective camera** is used to see objects having a near to far effect. A **perspective camera** view might look like this:



As you can see with a **perspective camera**, objects in the *near* are larger and objects in the **far** are smaller.

The **orthogonal camera** is used to see objects as large distance. You can think about it as converting a 3D world to a 2D representation. An **orthogonal camera** view might look like this:



As you can see with an **orthogonal camera**, objects are the same size regardless of how far away from the `Camera` object they are. **Mini Maps** in games are commonly rendered with an **orthogonal camera**. Another example would be a top - down view, perhaps in a dungeon style game.

Camera Use

Don't worry! `Camera` objects may sound complicated but Cocos2d-x makes them easy. When using 3D you don't have to do anything special to create a `Camera` object. Each `Scene` automatically creates a default camera, based on the projection properties of the `Director` object. If you need more than one camera, you can use the following code to create one:

```
auto s = Director::getInstance()->getWinSize();
auto camera = Camera::createPerspective(60, (GLfloat)s.width/s.height, 1, 1000);

// set parameters for camera
camera->setPosition3D(Vec3(0, 100, 100));
camera->lookAt(Vec3(0, 0, 0), Vec3(0, 1, 0));

addChild(camera); //add camera to the scene
```

Creating orthogonal camera

The default `Camera` is a **perspective camera**. If you want to create an **orthogonal camera**, it's easy to do by calling:

```
Camera::createOrthographic();
```

Example:

```
auto s = Director::getInstance()->getWinSize();
auto camera = Camera::createOrthographic(s.width, s.height, 1, 1000);
```

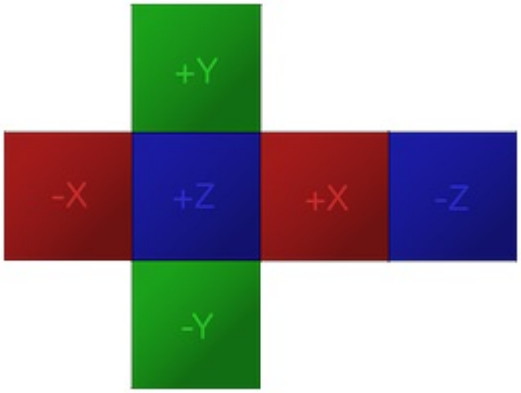
Hiding objects from the camera

Sometimes you don't want to have all objects visible in a `Camera` view. Hiding an object from one camera is very easy. Use **setCameraMask(CameraFlag)** on the `Node` and **setCameraFlag(CameraFlag)** on the `Camera`. Example:

```
//Camera
camera->setCameraFlag(CameraFlag::USER1);

//Node
node->setCameraMask(CameraFlag::USER1);
```

Cubemap Texture

A **cube map texture** is a collection of six separate square textures that are put onto the faces of an imaginary cube. Most often they are used to display infinitely far away reflections on objects, similar to how **sky box** displays far away scenery in the background. This is what an expanded cube map might look like:



In Cocos2d-x,you can create a **cube map texture** in this way:

```
// create a textureCube object with six texture assets
auto _textureCube = TextureCube::create("skybox/left.jpg",  "skybox/right.jpg",
"skybox/top.jpg", "skybox/bottom.jpg", "skybox/front.jpg","skybox/back.jpg");

// set cube map texture parameters
Texture2D::TexParams tRepeatParams;
tRepeatParams.magFilter = GL_NEAREST;
tRepeatParams.minFilter = GL_NEAREST;
tRepeatParams.wrapS = GL_MIRRORED_REPEAT;
tRepeatParams.wrapT = GL_MIRRORED_REPEAT;
_textureCube->setTexParameters(tRepeatParams);

// create and set our custom shader
auto shader = GLProgram::createWithFilenames("cube_map.vert","cube_map.frag");
auto _state = GLProgramState::create(shader);

// bind cube map texture to uniform
_state->setUniformTexture("u_cubeTex", _textureCube);
```

Skybox

`Skybox` is a wrapper around your entire scene that shows what the world looks like beyond your geometry. You might use a `Skybox` to simulate infinite sky, mountains and other phenomena.

Creating a `Skybox`:

```
// create a Skybox object
auto box = Skybox::create();

// set textureCube for Skybox
box->setTexture(_textureCube);

// attached to scene
_scene->addChild(box);
```

Light

Light is really important for building mood and ambiance for a game. There are currently 4 lighting techniques supported. You would use different lighting techniques depending upon your needs. Each lighting effect achieves a different result.

Ambient Light

An `AmbientLight` object will apply light evenly for everything in the scene. Think of lighting in an office environment. The lights are overhead and when you look at objects around the office you see them in the same light. Example:

```
auto light = AmbientLight::create (Color3B::RED);
addChild(light);
```

This produces:

Directional Light

`DirectionalLight` is often used to simulate a light source such as sunlight. When using `DirectionalLight` keep in mind that is has the same density no matter where you are in relationship to it. Also think about being outside on a sunny day with the sun beaming down on you. When you look directly at the sun, it is an intense light even if you move a few steps in any direction. Example:

```
auto light = DirectionLight::create(Vec3(-1.0f, -1.0f, 0.0f), Color3B::RED);
addChild(light);
```

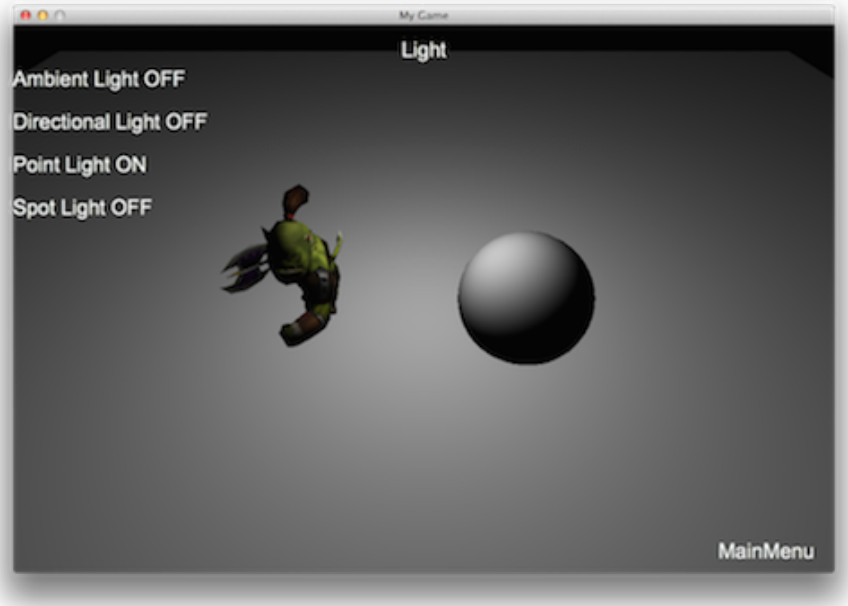
This produces:

Point Light

`PointLight` is often used to simulate the effect of light bulbs, lamps or torches. The direction of a `PointLight` is from the lighted position to the `PointLight`. Keep in mind that the density is different depending upon the distance from the `PointLight`. What does this mean? If you are close to the start of the `PointLight` it will be really strong. If you are towards the end of the `PointLight` it will be dim. `PointLight` also becomes weaker the larger the distance it is projected. Example:

```
auto light = PointLight::create(Vec3(0.0f, 0.0f, 0.0f), Color3B::RED, 10000.0f);
addChild(light);
```

This produces:



Spot Light

A `SpotLight` object is often used to simulate a flashlight. This means that it is emitted in just one direction in the shape of a cone. Think about the power going out in your house. You may need to take a flashlight down into your basement to reset your circuits. The flashlight produces a cone shaped lighting pattern and you can only see objects inside that cone pattern. Another example is in dark, dungeon based games where your path is light by torches. You can only see the limited cone shape that those torches emit. Example:

```
auto spotLight = SpotLight::create(Vec3(-1.0f, -1.0f, 0.0f), Vec3(0.0f, 0.0f, 0.0f),
Color3B::RED, 0.0, 0.5, 10000.0f) ;
addChild(spotLight);
```

This produces:

Light Masking

What do you use for lights in your kitchen or living room? Probably a few lamps? Do you ever notice that perhaps you only use a lamp to light up a certain portion of the room? You are essentially applying a **lighting mask**!

A **lighting mask** is used on a `Node` to only apply a particular **lighting source** to it. For example, if you had multiple lights in a `Scene`, a `Node` can only be lighted by one of the lights instead of all three. You can use **setLightFlag(LightFlag)** to control which `Node` objects are effected by the light. It is important to note that all lighting sources are rendered in a single pass. Due to mobile platform performance issues the use of multiple light sources is not recommended. The default maximum is 1. If you want to open multiple light sources you must define the following keys in **info.plist**:

```
<key> cocos2d.x.3d.max_dir_light_in_shader </key>
<integer> 1 </integer>
<key> cocos2d.x.3d.max_point_light_in_shader </key>
<integer> 1 </integer>
<key> cocos2d.x.3d.max_spot_light_in_shader </key>
<integer> 1 </integer>
```

Terrain

`Terrain` is an important component in 3D game. A texture is used to stand for the height map. And up to 4 textures can be used to blend the details of the terrain, grass, road, and so on.

HeightMap

`HeightMap` objects are the core of the terrain. Different from the common image the height map represents the height of vertices. It determines the terrain's geometry shape.

DetailMap

`DetailMap` objects are a list of textures determining the details of the terrain, up to four textures can be used.

AlphaMap

`AlphaMap` objects are an image whose data is the blend weights of **detail maps**. The blending result is the final terrain's appearance.

LOD policy

`Terrain` uses an optimization technique called **Level Of Detail** or **LOD**. This is a rendering technique that reduces the number of **verticies** (or triangles) that are rendered ,for an object, as its distance from camera increases. Users can set the distance to the `Camera` by calling the **Terrain::setLODDistance(float lod1, float lod2, float lod3)** method.

Neighboring chunks of `Terrain` objects, which have different **LOD** may cause the **crack** artifacts. `Terrain` provide two functions to avoid them:

`Terrain::CrackFixedType::SKIRT`

`Terrain::CrackFixedType::INCREASE_LOWER`

Terrain::CrackFixedType::SKIRT will generate four, skirt-like meshes at each edge of the chunk.

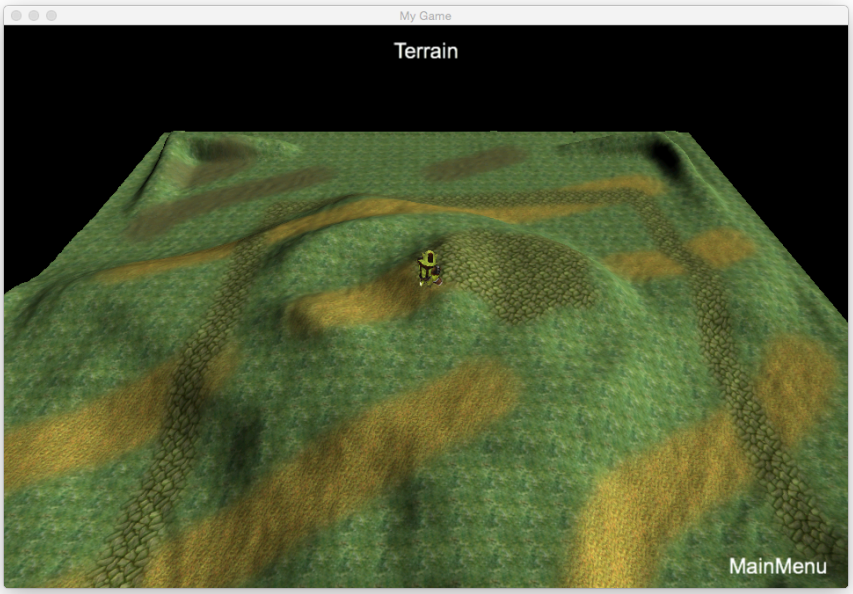
Terrain::CrackFixedType::INCREASE_LOWER will dynamically adjust each chunks **indices** to seamlessly connect them.

How to create a terrain

Creating a `Terrain` takes a few steps. Example:

The following code snippet is creating a player and place it on the terrain:

```
auto player = Sprite3D::create("chapter9/orc.c3b");
player->setScale(0.08);
player->setPositionY(terrain->getHeight(player->getPositionX(),player->getPositionZ()));
```



- create all `DetailMap` objects (up to four), you need pass the `DetailMap` objects to the **Terrain::DetailMap** struct:

```
Terrain::DetailMap r("dirt.dds");
Terrain::DetailMap g("grass.dds");
Terrain::DetailMap b("road.dds");
Terrain::DetailMap a("greenSkin.jpg");
```

- to create a `TerrainData` variable with **detail maps**, you need to specify the terrain's **height map** file path and **alpha map** file path:

```
Terrain::TerrainData data("chapter9/heightmap16.jpg","TerrainTest/alphamap.png", r, g, b, a);
```

- pass the `TerrainData` object to **Terrain::create**, the last parameter determines the LOD policy (as talked about above). Example:

```
_terrain = Terrain::create(data, Terrain::CrackFixedType::SKIRT);
```

- If you set a `Terrain` objects **camera mask** and add it to a `Node` or a `Scene`, be careful. When `Terrain` is added into a `Node` or a `Scene`, you can not use **transform(translate, scale)** on it anymore. If you do this after calling **addChild()**, some of the terrain's methods may calculate wrong results.

Get Terrain Height

Use the method **Terrain::getHeight(float x, float z, Vec3 * normal= nullptr)** to get the specified position's height. This method is very useful when you want to put a `Sprite3D` object or any `Node` on the terrain's surface.

Ray-Terrain intersection test

A **Ray-Terrain** intersection test will calculate the intersection point by giving a specified position.

Terrain::CrackFixedType::SKIRT will generate four skirt-like meshes at each chunks edge.

Terrain::CrackFixedType::INCREASE_LOWER will dynamically adjust each chunks index to seamlessly connect them.

3D Software Packages

3D Editors

3D editors are collections of tools that you use to build your 3D graphics. There are both commercial and free tools available. These are the most popular editors:

- Blender (Free)
- 3DS Max
- Cinema4D
- Maya

Most 3D editors usually save files in a common collection of formats for easy use within other editors as well as a standard way for game engines to import your files for use.

Cocos2d-x Provided Tools

Cocos2d-x provides tools to help with converting your 3D models to formats that Cocos2d-x uses to provide access to all aspects of your 3D files.

fbx-conv command-line tool

fbx-conv allows the conversion of an FBX file into the Cocos2d-x proprietary formats. FBX is the most popular 3D file format and is being supported by all the major editors. **fbx-conv** exports to **.c3b** by default. It is simple to use with just a few parameters:

```
fbx-conv [-a|-b|-t] FBXFile
```

The possible switches are:

- -?: show help
- -a: export both text and binary format

- -b: export binary format
- -t: export text format

Example:

```
fbx-conv -a boss.FBX
```

There are a few things to note about **fbx-conv**: * The model needs to have a material that contains at least one texture * it only supports skeletal animation. * it only supports one skeleton object no multiple skeleton support yet. * You can create a 3d scene by exporting multiple static model * The maximum amount of vertices or indices a mesh is 32767

3D File Formats

Cocos2d-x currently supports two 3d file formats:

- Wavefront Object files: **.obj** files
- Cocos2d-x 3d ad-hoc format:**c3t**, **c3b** files.

The **Wavefront** file format is supported because it has been widely adopted by 3D editors and it is extremely easy to parse. It is, however, limited and doesn't support advanced features like animations.

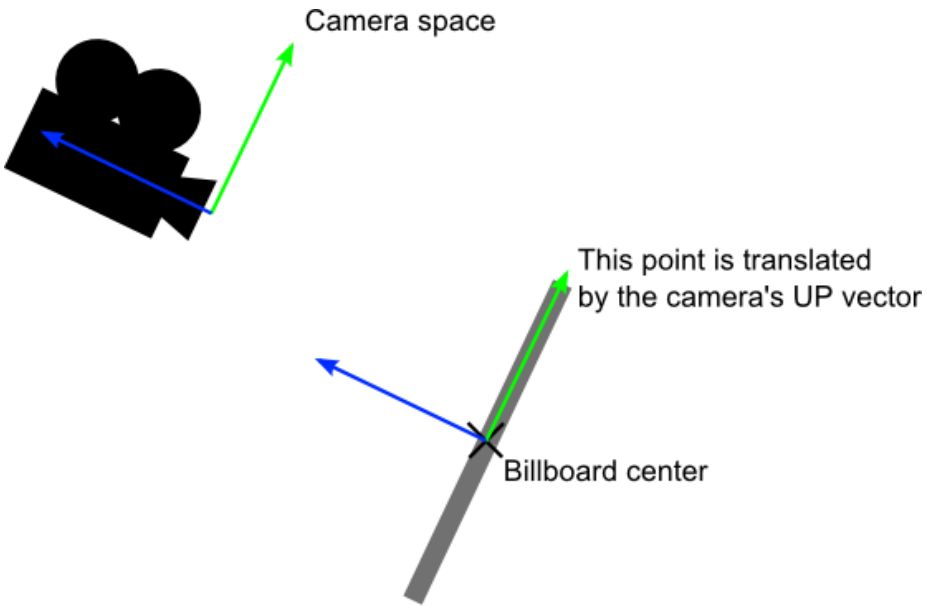
On the other hand, **c3t** and **c3b** are Cocos2d-x proprietary file formats that were created to allow animations, materials and other advanced 3d features. The suffix **t** means **text**, while the suffix **b** means **binary**. Developers must use **c3b** for production because it is more efficient. In case you want to debug the file and track its changes in Git or any other version control system, you should **c3t** instead. Also, `Animation3D` objects can be created with **c3b** or **c3t** files as it is not possible to animate **obj** files.

Advanced Topics

BillBoard

You may not have heard of a `BillBoard` before. No, I'm not talking about an advertisement on the side of a highway. Rather, `Billboard` is a special `Sprite` that always faces the `Camera`. As you rotate the `Camera`, `Billboard` objects also rotate. Using a`BillBoard` is a very common rendering technique. Take for example a downhill skiing game. Any trees, rocks or other objects that are in the way of the skier are `Billboard` objects.

This is how `Camera` and `Billboard` objects relate to each other.



`Billboard` objects are easy to create. `BillBoard` is derived from `Sprite`, so it supports most of the features as a `Sprite` object. We can create one using the following create method:

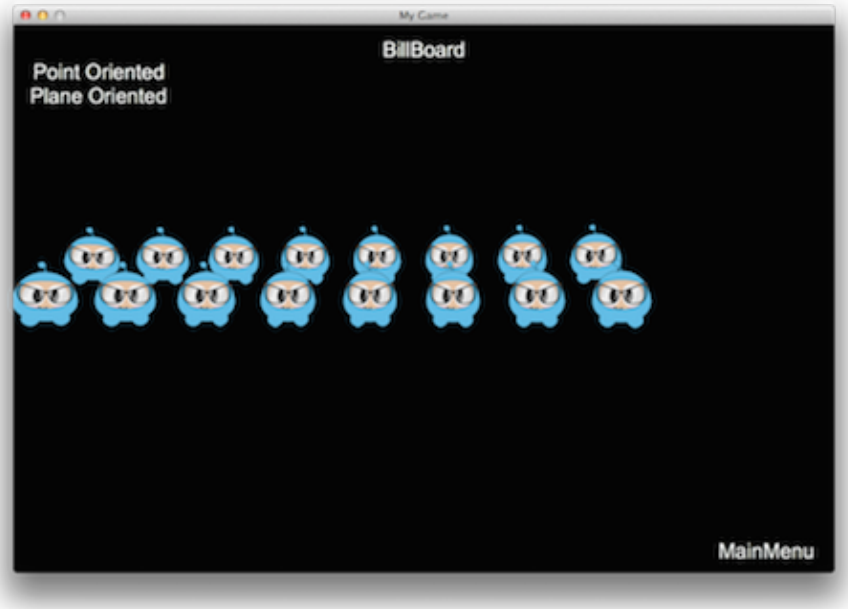
```
auto billboard = BillBoard::create("Blue_Front1.png", BillBoard::Mode::VIEW_POINT_ORIENTED);
```

You can also create a `Billboard` object for the camera XOY plane (like the plane of a floor) by changing the `BillBoard` objects mode:

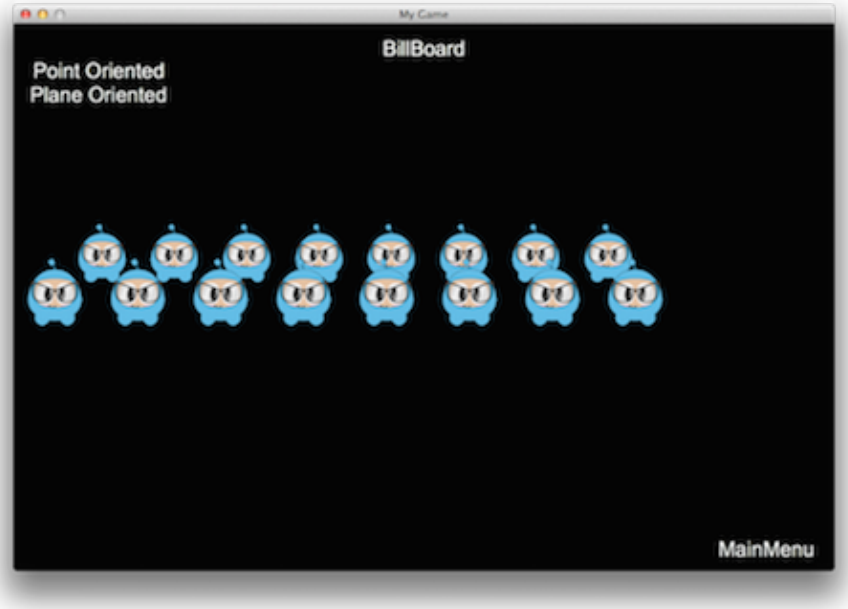
```
auto billboard = BillBoard::create("Blue_Front1.png", BillBoard::Mode::VIEW_PLANE_ORIENTED);
```

These *create* methods look a little different since an additional parameter of **BillBoard::Mode** is passed in. There are two **BillBoard::Mode** types, **VIEW_POINT_ORIENTED** and **VIEW_PLANE_ORIENTED**.

VIEW_POINT_ORIENTED is where the `BillBoard` object is oriented to the `Camera`. Example:



VIEW_PLANE_ORIENTED is where the `BillBoard` is oriented towards the XOY plane of the `Camera`. Example:



You can also set properties for a `BillBoard` just like with any other `Node`. These include, but are not limited to: **scale**, **position**, **rotation**. Examples:

```
billboard->setScale(0.5f);
billboard->setPosition3D(Vec3(0.0f, 0.0f, 0.0f));
billboard->setBlendFunc(BlendFunc::ALPHA_NON_PREMULTIPLIED);
addChild(billboard);
```

ParticleSystem3D

In Chapter 7, you learned about 2D particles and how to use them. When you use 3D you might also want to use a 3D particle system for rich, advanced effects. Many of the same concepts apply for a 3D particle system as they did with a 2D particle system. Cocos2d-x currently supports **Particle Universe** (http://www.fxpression.com/) for particle system construction. **Particle Universe** provides a special particle editor that allows you to quickly and easily set up a variety of effects, such as explosions, fire, blood and other special effects. This editor uses a **pu** file extension when saving or exporting.

When you are happy with your particle and ready to use it in code, exporting to its built-in format of **pu** is enough! Cocos2d-x supports this format directly. Also, as `ParticleSystem3D` is derived from `Node`, it supports most of the features that `Node` supports. `PUParticleSystem3D` is an object type specifically for dealing with **Particle Universe** particles. `PUParticleSystem3D` offers two ways for creating particles.

The first way is to build a particle by passing in a **Particle Universe** file and its corresponding **material file**. Remember from Chapter 7 that a **material file** is what describes the particle. This is required. Example:

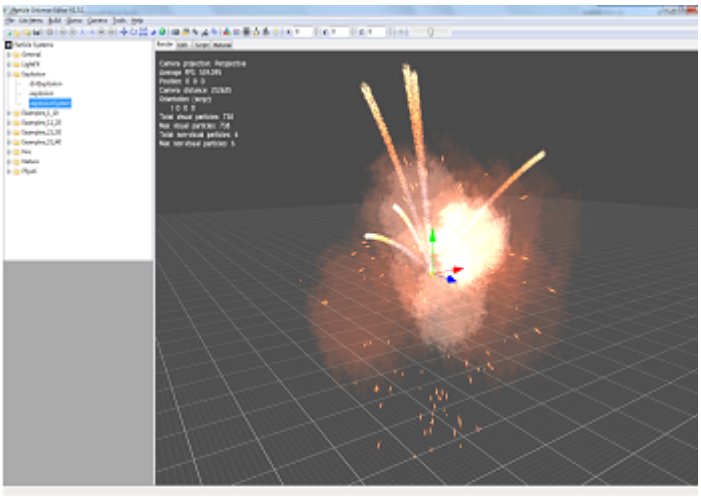
```
auto ps = PUParticleSystem3D::create("lineStreak.pu", "pu_mediapack_01.material");
ps->startParticleSystem();
this->addChild(ps);
```

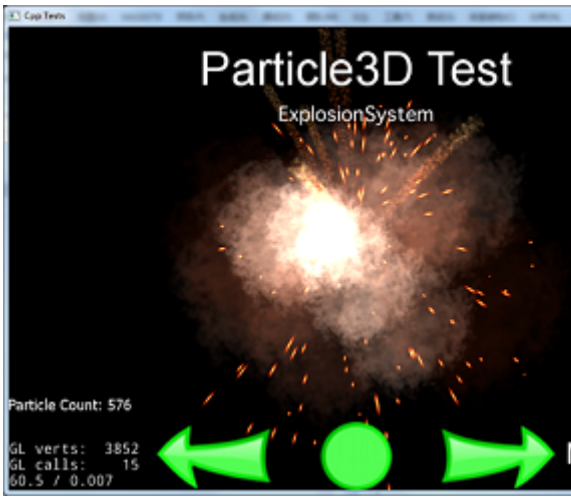
The second way is to build the particle system only by passing a **particle universe** file. When you create a particle this way, besides loading the particle, all **material files** in the same folder as the particle file will automatically be loaded. Here is an example:

```
auto ps = PUParticleSystem3D::create("electricBeamSystem.pu");
ps->startParticleSystem();

this->addChild(ps);
```

Note: using this method will result in an increase in loading times and consumes more memory since everything will be loaded. If you know what **material** you want to use and don't need to load everything, using the first method would be preferred.

In these images below, on the left is the particle in **particle universe**, while on the right is the effect running in Cocos2d-x:





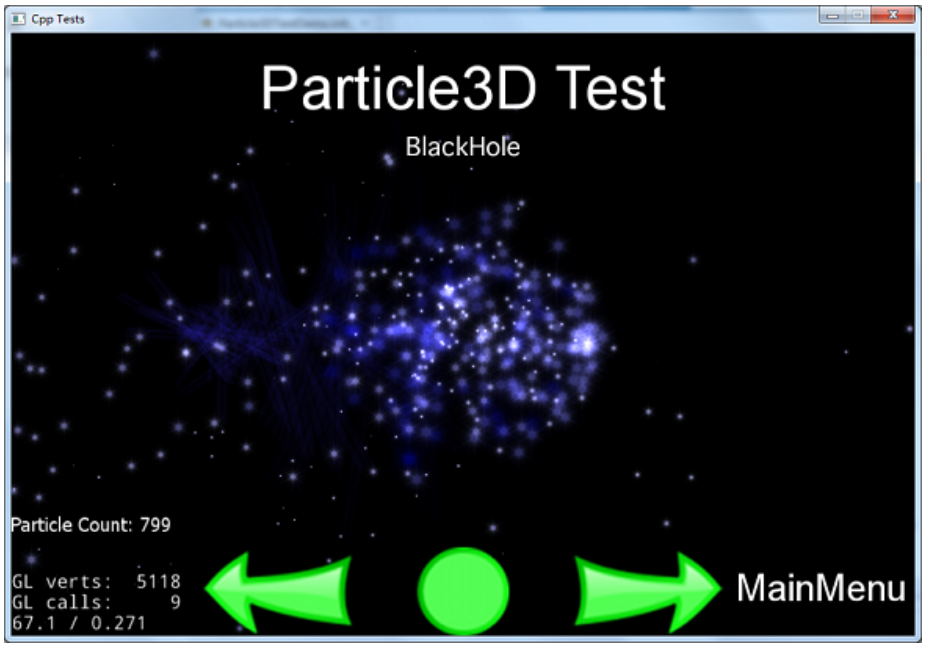
Once you have your particle, you can interact with it it fairly obvious ways. You can interact with with the **particle system** as a whole, starting, stopping, pausing, resuming and obtaining the total number of particles:

```
virtual void startParticleSystem() override;
virtual void stopParticleSystem() override;
virtual void pauseParticleSystem() override;
virtual void resumeParticleSystem() override;
virtual int getAliveParticleCount() const override;
```

As `PUParticleSystem3D` is derived from `Node` you can run `Action` and `Sequence` objects on your particles! Example:

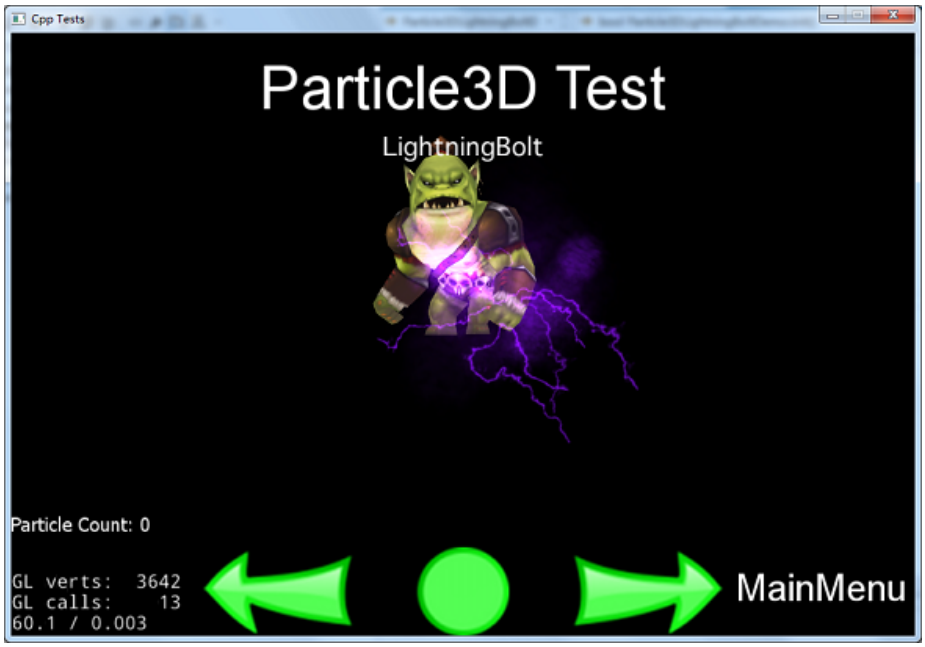
```
auto ps = PUParticleSystem3D::create("blackHole.pu", "pu_mediapack_01.material");
ps->setPosition(-25.0f, 0.0f);

auto moveby = MoveBy::create(2.0f, Vec2(50.0f, 0.0f));
auto moveby1 = MoveBy::create(2.0f, Vec2(-50.0f, 0.0f));

ps->runAction(RepeatForever::create(Sequence::create(moveby, moveby1, nullptr)));
ps->startParticleSystem();
```

Combining `Action` and `Sequence` objects could produce an interesting black hole effect:



Just like with other 3D objects you can also combine 3D objects using `AttachNode`. This allows for creating rich models. Example:

```
auto sprite3d = Sprite3D::create("orc.c3b");
sprite3d->setPosition3D(Vec3(0.0f, 0.0f, 0.0f));
sprite3d->setRotation3D(Vec3(0.0f, 180.0f, 0.0f));

auto animation = Animation3D::create("orc.c3b");
if (animation)
{
    auto animate = Animate3D::create(animation);
    sprite3d->runAction(RepeatForever::create(animate));
}

auto handler = PUParticleSystem3D::create("lightningBolt.pu");
handler->startParticleSystem();
sprite3d->getAttachNode("Bip001 L Hand")->addChild(handler);

this->addChild(sprite3d);
```



Scripting

Script component

Script component is used to extend c++ `Node` objects. You can add a **script component** to a `Node`, then the **script component** will receive **onEnter**, **onExit** and **update** events.

Script component supports both JavaScript and LUA. You should use the proper **script component** type for the language you are developing with. If you are developing with JavaScript, you would use `ComponentJS`, if you are developing with Lua, you would use `ComponentLUA`. But, you cannot mix them or use them in a c++ project! This is because the proper bindings for that language are required and these bindings are only available in their respective project types.

Example with Lua:

```
// create a Sprite and add a LUA component
auto player = Sprite::create("player.png");

auto luaComponent = ComponentLua::create("player.lua");
player->addComponent(luaComponent);
```

```
-- player.lua

local player = {
```

```
    onEnter = function(self)
        -- do some things in onEnter
    end,

    onExit = function(self)
        -- do some things in onExit
    end,

    update = function(self)
        -- do some things every frame
    end
}

-- it is needed to return player to let c++ nodes know it
return player
```

Example with JavaScript:

```
// create a Sprite and add a LUA component
auto player = Sprite::create("player.png");

auto jsComponent = ComponentJS::create("player.js");
player->addComponent(jsComponent);
```

```
// player.js
Player = cc.ComponentJS.extend({
    generateProjectile: function (x, y) {
        var projectile = new cc.Sprite("components/Projectile.png", cc.rect(0, 0, 20, 20));
        var scriptComponent = new cc.ComponentJS("src/ComponentTest/projectile.js");
        projectile.addComponent(scriptComponent);
        this.getOwner().getParent().addChild(projectile);

        // set position
        var winSize = cc.director.getVisibleSize();
        var visibleOrigin = cc.director.getVisibleOrigin();
        projectile.setPosition(cc.p(visibleOrigin.x + 20, visibleOrigin.y + winSize.height/2));

        // run action
        var posX = projectile.getPositionX();
        var posY = projectile.getPositionY();
        var offX = x - posX;
        var offY = y - posY;

        if (offX <= 0) {
            return;
        }
```

```
        var contentSize = projectile.getContentSize();
        var realX = visibleOrigin.x + winSize.width + contentSize.width/2;
        var ratio = offY / offX;
        var realY = (realX * ratio) + posY;
        var realDest = cc.p(realX, realY);

        var offRealX = realX - posX;
        var offRealY = realY - posY;
        var length = Math.sqrt((offRealX * offRealX) + (offRealY * offRealY));
        var velocity = 960;
        var realMoveDuration = length / velocity;

        projectile.runAction(cc.moveTo(realMoveDuration, realDest));
    },

    onEnter: function() {
        var owner = this.getOwner();
        owner.playerComponent = this;
        cc.eventManager.addListener({
            event: cc.EventListener.TOUCH_ALL_AT_ONCE,
            onTouchesEnded: function (touches, event) {
                var target = event.getCurrentTarget();
                if (target.playerComponent) {
                    var location = touches[0].getLocation();
                    target.playerComponent.generateProjectile(location.x, location.y);
                    jsb.AudioEngine.play2d("pew-pew-lei.wav");
                }
            }
        }, owner);
    }
});
```

One difference to keep in mind, between JavaScript and LUA components, is you should return the **object** in LUA component, in JavaScript, you only have to extend `cc.ComponentJS`

For more detailed usage, please refer to tests projects: `tests/lua-tests/src/ComponentTest` and `tests/js-tests/src/ComponentTest`.

Physics

Your game is coming along nicely. You have `Sprite` objects, gameplay mechanics and your coding efforts are paying off. You are starting to feel like your game is playable. What do you do when you realize your game needs to simulate real world situations? You know, **collision detection**, **gravity**, **elasticity** and **friction**. Yes, you guessed it! This chapter is on **physics** and the use of a **physics engine**. Let's explore the *when*, *wheres* and *whys* of using a **physics**

engine.

Physics is scary, do I really need it? Please tell me no!

Please don't run away there are no physics monsters under your bed! Your needs might be simple enough to not need to use a **physics engine**. Perhaps a combination of using a `Node` objects **update()** function, `Rect` objects and a combination of the **containsPoint()** or **intersectsRect()** functions might be enough for you? Example:

```
void update(float dt)
{
  auto p = touch->getLocation();
  auto rect = this->getBoundingBox();

  if(rect.containsPoint(p))
  {
      // do something, intersection
  }
}
```

This mechanism works for **very simple** needs, but doesn't scale. What if you had 100 `Sprite` objects all continuously updating to check for intersections with other objects? It could be done but the the CPU usage and **framerate** would suffer severely. Your game would be unplayable. A **physics engine** solves these concerns for us in a scalable and CPU friendly way. Even though this might look foreign, let's take a look at a simple example and then nut and bolt the example, terminology and best practice together.

```
// create a static PhysicsBody
auto physicsBody = PhysicsBody::createBox(Size(65.0f , 81.0f ), PhysicsMaterial(0.1f, 1.0f, 0.0f)
physicsBody->setDynamic(false);

// create a sprite
auto sprite = Sprite::create("whiteSprite.png");
sprite->setPosition(Vec2(400, 400));

// sprite will use physicsBody
sprite->addComponent(physicsBody);

//add contact event listener
auto contactListener = EventListenerPhysicsContact::create();
contactListener->onContactBegin = CC_CALLBACK_1(onContactBegin, this);
_eventDispatcher->addEventListenerWithSceneGraphPriority(contactListener, this);
```

Even though this example is simple, it looks complicated and scary. It really isn't if we look closely. Here are the steps that are happening: * A `PhysicsBody` object is created. * A `Sprite` object is created. * The `Sprite` object applies the properties of the `PhysicsBody` object. * A listener is created to respond to an **onContactBegin()** event.

Once we look step by step the concept starts to make sense. To better understand all the details of a **physics engine** you should understand the following terms and concepts:

Physics terminology and concepts

Bodies

A `PhysicsBody` holds the physical properties of an object. These include **mass**, **position**, **rotation**, **velocity** and **damping**. `PhysicsBody` objects are the backbone for shapes. A `PhysicsBody` does not have a shape until you attach a shape to it.

Material

Materials describe material attributes□

-density□It is used to compute the mass properties of the parent body.

-friction□It is used to make objects slide along each other realistically.

-restitution□It is used to make objects bounce. The restitution value is usually set to be between 0 and 1. 0 means no bouncing while 1 means perfect bouncing.

Shapes

Shapes describe collision geometry. By attaching shapes to bodies, you define a body's shape. You can attach as many shapes to a single body as you need in order to define a complex shape. Each shape relates to a `PhysicsMaterial` object and contains the following attributes: **type**, **area**, **mass**, **moment**, **offset** and **tag**. Some of these you might not be familiar with:

-*type*□describes the categories of shapes, such as circle, box, polygon, etc.

-*area*: used to compute the mass properties of the body. The density and area gives the mass.

-*mass*: the quantity of matter that a body contains, as measured by its acceleration under a given force or by the force exerted on it by a gravitational field.

-*moment*: determines the torque needed for a desired angular acceleration.

-*offset*: offset from the body's center of gravity in body local coordinates.

-*tag*: used to identify the shape easily for developers. You probably remember that you can assign all `Node` objects a tag for identification and easy access.

We describe the various **shapes** as: >-`PhysicsShape`: Shapes implement the `PhysicsShape` base class.

-`PhysicsShapeCircle`: Circles are solid. You cannot make a hollow circle using the circle shape.

-`PhysicsShapePolygon`: Polygon shapes are solid convex polygons.

-`PhysicsShapeBox`: Box shape is one kind of convex polygon.

-`PhysicsShapeEdgeSegment`: A segment shape.

-`PhysicsShapeEdgePolygon`: Hollow polygon shapes. A edge-polygon shape consists of multiple segment shapes.

-`PhysicsShapeEdgeBox`Hollow box shapes. A edge-box shape consists of four segment shapes.

-`PhysicsShapeEdgeChain`: The chain shape provides an efficient way to connect many edges together.

Contacts/Joints

Contacts and **joint** objects describe how bodies are attached to each other.

World

A **world** container is what your physics bodies are added to and where they are simulated. You add **bodies**, **shapes** and **constraints** to a world and then update the world as a whole. The **world** controls how all of these items interact together. Much of the interaction with the physics API will be with a `PhysicsWorld` object.

There is a lot to remember here, keep these terms handy to refer back to them as needed.

Physics World and Physics Body

PhysicsWorld

A `PhysicsWorld` object is the core item used when simulating physics. Just like the world we live in, a `PhysicsWorld` has a lot of things happening at once. `PhysicsWorld` integrates deeply at the `Scene` level because of it's many facets. Let's use a simple example that we can all relate to. Does your residence have a kitchen? Think of this as your **physics world**! Now your world has `PhysicsBody` objects, like food, knives, appliances! These bodies interact with each other inside the world. These objects touch and also react to those touches. Example: use a knife to cut food and put it in an appliance. Does the knife cut the food? Maybe. Maybe not. Perhaps it isn't the correct type of knife for the job.

You can create a `Scene` that contains a `PhysicsWorld` using:

```
auto scene = Scene::createWithPhysics();
```

Every `PhysicsWorld` has properties associated with it: >-gravity: Global gravity applied to the world. Defaults to Vec2(0.0f, -98.0f).

-speed: Set the speed of physics world, speed is the rate at which the simulation executes. Defaults to 1.0.

-updateRate: set the update rate of physics world, update rate is the value of EngineUpdateTimes/PhysicsWorldUpdateTimes.

-substeps: set the number of substeps in an update of the physics world.

The process of updating a `PhysicsWorld` is called **stepping**. By default, the `PhysicsWorld` **updates through time** automatically. This is called **auto stepping**. It automatically happens for you, each frame. You can disable **auto steping** of the `PhysicsWorld` by setting **setAutoStep(false)**. If you do this, you would **step** the `PhysicsWorld` manually by setting **step(time)**. **Substeps** are used to step the `PhysicsWorld` forward multiple times using a more precise time increment than a single frame. This allows for finer grained control of the **stepping** process including more fluid movements.

PhysicsBody

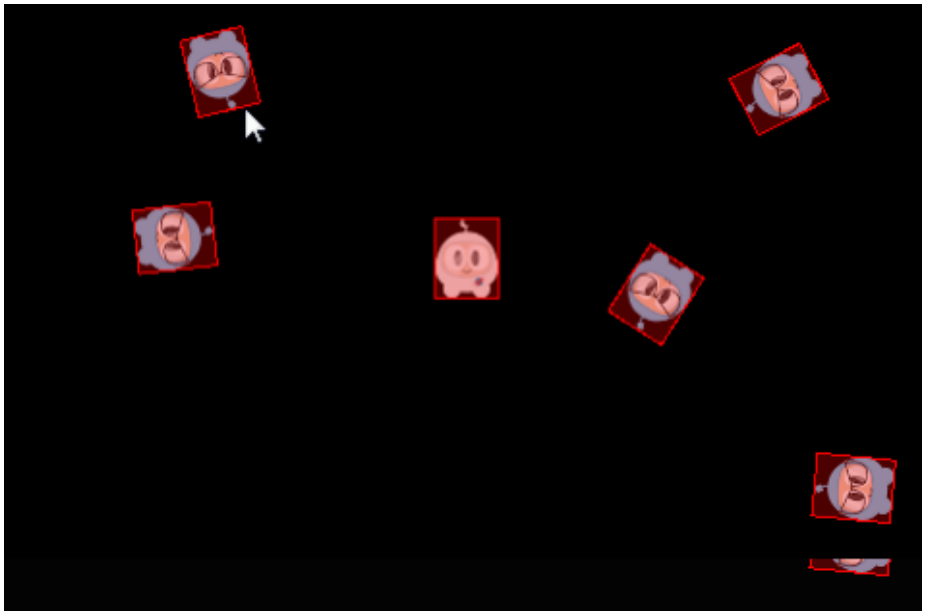
`PhysicsBody` objects have **position** and **velocity**. You can apply **forces**, **movement**, **damping** and **impulses** (as well as more) to `PhysicsBody` objects. `PhysicsBody` can be **static** or **dynamic**. A **static** body does not move under simulation and behaves as if it has infinite **mass**. A **dynamic** body is fully simulated. They can be moved manually by the user, but normally they move according to forces. A dynamic body can collide with all body types. `Node` provides **setPhysicsBody()** to associate a `PhysicsBody` to a `Node` object.

Lets create a static and 5 dynamic `PhysicsBody` objects that are a box shape:

```
auto physicsBody = PhysicsBody::createBox(Size(65.0f, 81.0f),
                        PhysicsMaterial(0.1f, 1.0f, 0.0f));
physicsBody->setDynamic(false);

//create a sprite
auto sprite = Sprite::create("whiteSprite.png");
sprite->setPosition(s_centre);
addChild(sprite);

//apply physicsBody to the sprite
sprite->addComponent(physicsBody);

//add five dynamic bodies
for (int i = 0; i < 5; ++i)
{
    physicsBody = PhysicsBody::createBox(Size(65.0f, 81.0f),
                    PhysicsMaterial(0.1f, 1.0f, 0.0f));

    //set the body isn't affected by the physics world's gravitational force
    physicsBody->setGravityEnable(false);
```

```
    //set initial velocity of physicsBody
    physicsBody->setVelocity(Vec2(cocos2d::random(-500,500),
                cocos2d::random(-500,500)));
    physicsBody->setTag(DRAG_BODYS_TAG);

    sprite = Sprite::create("blueSprite.png");
    sprite->setPosition(Vec2(s_centre.x + cocos2d::random(-300,300),
                s_centre.y + cocos2d::random(-300,300)));
    sprite->addComponent(physicsBody);

    addChild(sprite);
}
```

The result is a stationary `PhysicsBody` with 5 additional `PhysicsBody` objects colliding around it.



Collision

Have you ever been in a car accident? What did you collide with? Just like with cars, `PhysicBody` objects can come in contact. **Collisions** are what happens when `PhysicBody` objects come in contact with each other. When a **collision** takes place it can be ignored or it can trigger events to be fired.

Filtering Collisions

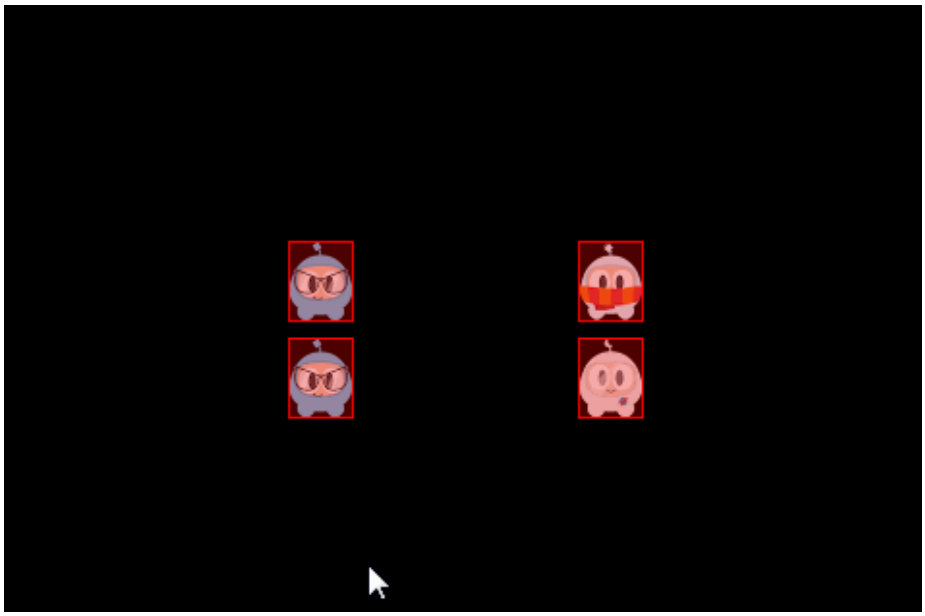
Collision filtering allows you to enable or prevent collisions between shapes. This **physics engine** supports collision filtering using **category and group bitmasks**.

There are 32 supported collision categories. For each shape you can specify which category it belongs to. You can also specify what other categories this shape can collide with. This is done with masking bits. For example:

```
auto sprite1 = addSpriteAtPosition(Vec2(s_centre.x - 150,s_centre.y));
sprite1->getPhysicsBody()->setCategoryBitmask(0x02);    // 0010
sprite1->getPhysicsBody()->setCollisionBitmask(0x01);   // 0001

sprite1 = addSpriteAtPosition(Vec2(s_centre.x - 150,s_centre.y + 100));
sprite1->getPhysicsBody()->setCategoryBitmask(0x02);    // 0010
sprite1->getPhysicsBody()->setCollisionBitmask(0x01);   // 0001

auto sprite2 = addSpriteAtPosition(Vec2(s_centre.x + 150,s_centre.y),1);
sprite2->getPhysicsBody()->setCategoryBitmask(0x01);    // 0001
sprite2->getPhysicsBody()->setCollisionBitmask(0x02);   // 0010

auto sprite3 = addSpriteAtPosition(Vec2(s_centre.x + 150,s_centre.y + 100),2);
sprite3->getPhysicsBody()->setCategoryBitmask(0x03);    // 0011
sprite3->getPhysicsBody()->setCollisionBitmask(0x03);   // 0011
```

You can check for collisions by checking and comparing *category* and *collision* bitmasks like:

```
if ((shapeA->getCategoryBitmask() & shapeB->getCollisionBitmask()) == 0
   || (shapeB->getCategoryBitmask() & shapeA->getCollisionBitmask()) == 0)
{
   // shapes can't collide
   ret = false;
}
```



Collision groups let you specify an integral group index. You can have all shapes with the same group index always collide (positive index) or never collide (negative index and zero index). Collisions between shapes of different group indices are filtered according the category and mask bits. In other words, group filtering has higher precedence than category filtering.

Contacts/Joints

Recall from the terminology above that **joints** are how contact points are connected to each other. Yes, you can think of it just like **joints** on your own body. Each joint type has a definition that derives from `PhysicsJoint`. All joints are connected between two different bodies. One body may be static. You can prevent the attached bodies from colliding with each other by **joint->setCollisionEnable(false)**. Many joint definitions require that you provide some geometric data. Often a joint will be defined by anchor points. The rest of the joint definition data depends on the joint type.

-PhysicsJointFixed: A fixed joint fuses the two bodies together at a reference point. Fixed joints are useful for creating complex shapes that can be broken apart later.

-PhysicsJointLimit: A limit joint imposes a maximum distance between the two bodies, as if they were connected by a rope.

-PhysicsJointPin: A pin joint allows the two bodies to independently rotate around the anchor point as if pinned together.

-PhysicsJointDistance: Set the fixed distance with two bodies

-PhysicsJointSpring: Connecting two physics bodies together with a spring

-PhysicsJointGroove: Attach body a to a line, and attach body b to a dot

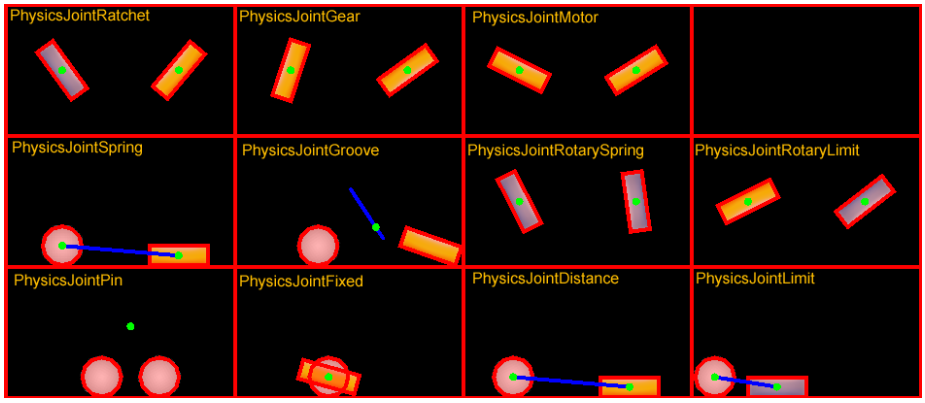
-PhysicsJointRotarySpring: Likes a spring joint, but works with rotary

-PhysicsJointRotaryLimit: Likes a limit joint, but works with rotary

-PhysicsJointRatchet: Works like a socket wrench

-PhysicsJointGear: Keeps the angular velocity ratio of a pair of bodies constant

-PhysicsJointMotor: Keeps the relative angular velocity of a pair of bodies constant



Collision detection

Contacts are objects created by the **physics engine** to manage the collision between two shapes. **Contact** objects are not created by the user, they are created automatically. There are a few terms associated with contacts.

-contact point: A contact point is a point where two shapes touch.

-contact normal: A contact normal is a unit vector that points from one shape to another.

You can get the `PhysicsShape` from a **contact**. From those you can get the bodies.

```
bool onContactBegin(PhysicsContact& contact)
{
    auto bodyA = contact.getShapeA()->getBody();
    auto bodyB = contact.getShapeB()->getBody();
    return true;
}
```

You can get access to **contacts** by implementing a **contact listener**. The **contact listener** supports several events: **begin**, **pre-solve**, **post-solve** and **separate**.

-begin: Two shapes just started touching for the first time this step. Return true from the callback to process the collision normally or false to cause physics engine to ignore the collision entirely. If you return false, the *preSolve()* and *postSolve()* callbacks will never be run, but you will still receive a separate event when the shapes stop overlapping.

-pre-solve: Two shapes are touching during this step. Return false from the callback to make physics engine ignore the collision this step or true to process it normally. Additionally, you may override collision values using *setRestitution()*, *setFriction()* or *setSurfaceVelocity()* to provide custom restitution, friction, or surface velocity values.

-post-solve: Two shapes are touching and their collision response has been processed.

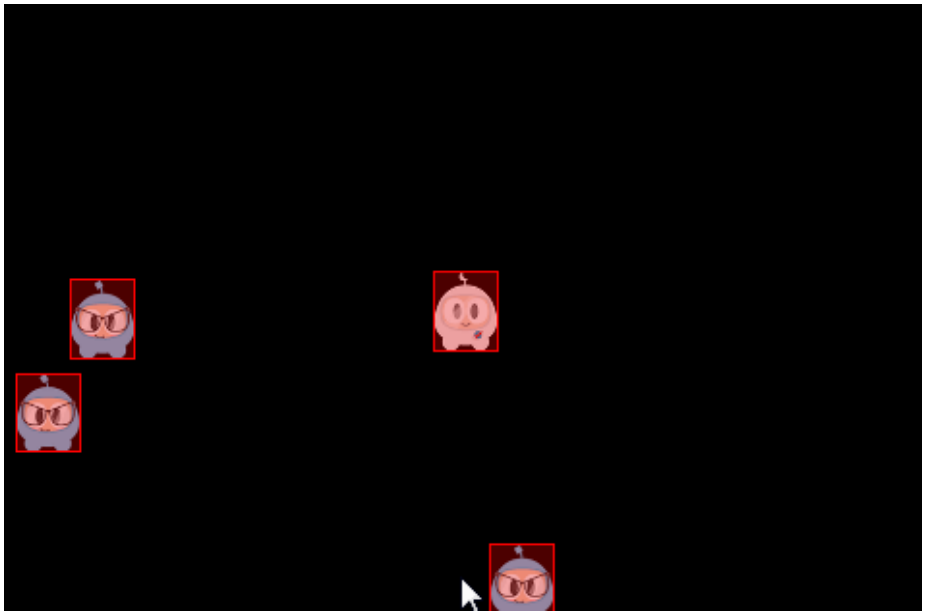
-separate: Two shapes have just stopped touching for the first time this step.

You also can use `EventListenerPhysicsContactWithBodies`, `EventListenerPhysicsContactWithShapes`, `EventListenerPhysicsContactWithGroup` to listen for the event you're interested with bodies, shapes or groups. Besides this you also need to set the physics contact related bitmask value, as the contact event won't be received by default, even if you create the relative **EventListener**.

For example:

```
bool init()
{
    //create a static PhysicsBody
    auto sprite = addSpriteAtPosition(s_centre,1);
    sprite->setTag(10);
    sprite->getPhysicsBody()->setContactTestBitmask(0xFFFFFFFF);
    sprite->getPhysicsBody()->setDynamic(false);

    //adds contact event listener
    auto contactListener = EventListenerPhysicsContact::create();
    contactListener->onContactBegin = CC_CALLBACK_1(PhysicsDemoCollisionProcessing::onContactBeg
    _eventDispatcher->addEventListenerWithSceneGraphPriority(contactListener, this);

    schedule(CC_SCHEDULE_SELECTOR(PhysicsDemoCollisionProcessing::tick), 0.3f);
    return true;

    return false;
}

void tick(float dt)
{
    auto sprite1 = addSpriteAtPosition(Vec2(s_centre.x + cocos2d::random(-300,300),
      s_centre.y + cocos2d::random(-300,300)));
    auto physicsBody = sprite1->getPhysicsBody();
```

```
    physicsBody->setVelocity(Vec2(cocos2d::random(-500,500),cocos2d::random(-500,500)));
    physicsBody->setContactTestBitmask(0xFFFFFFFF);
}

bool onContactBegin(PhysicsContact& contact)
{
    auto nodeA = contact.getShapeA()->getBody()->getNode();
    auto nodeB = contact.getShapeB()->getBody()->getNode();

    if (nodeA && nodeB)
    {
        if (nodeA->getTag() == 10)
        {
            nodeB->removeFromParentAndCleanup(true);
        }
        else if (nodeB->getTag() == 10)
        {
            nodeA->removeFromParentAndCleanup(true);
        }
    }

    //bodies can collide
    return true;
}
```



Queries

Have you ever stood in one position and looked around? You see things **near** to you and **far** from you. You can gauge how close things are to you. **Physics engines** provide this same type of **spatial query**. `PhysicsWorld` objects currently support **point queryies**, **ray casts** and **rect queries**.

Point Queries

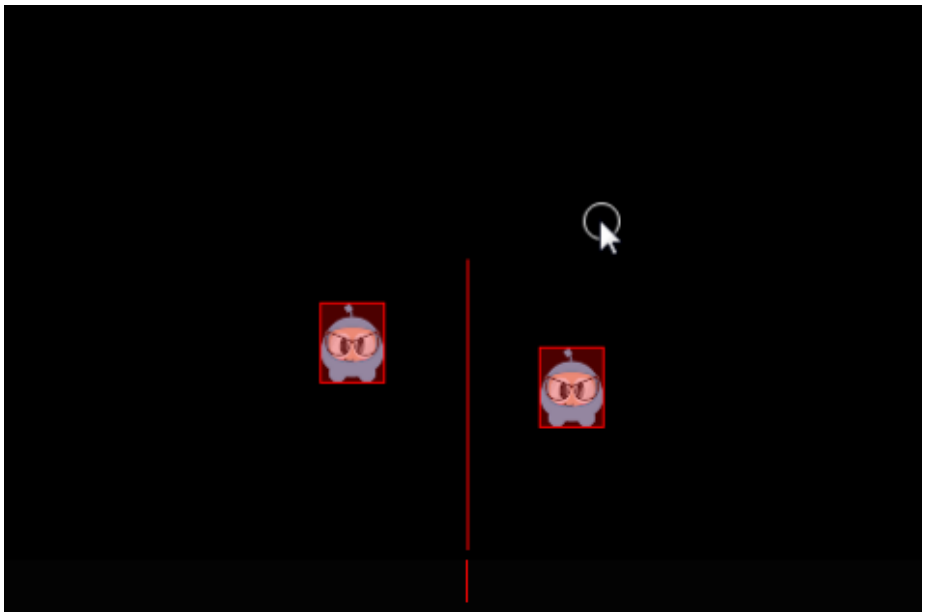
When you touch something, say your desk, you can think of this as a **point query**. They allow you to check if there are shapes within a certain distance of a point. **Point queries** are useful for things like **mouse picking** and **simple sensors**. You can also find the closest point on a shape to a given point or find the closest shape to a point.

Ray Cast

If you are looking around, some object within your sight is bound to catch your attention. You have essentially performed a **ray cast** here. You scanned until you found something interesting to make you stop scanning. You can **ray cast** at a shape to get the point of first intersection. For example:

```
void tick(float dt)
{
    Vec2 d(300 * cosf(_angle), 300 * sinf(_angle));
    Vec2 point2 = s_centre + d;
    if (_drawNode)
    {
        removeChild(_drawNode);
    }
    _drawNode = DrawNode::create();

    Vec2 points[5];
    int num = 0;
    auto func = [&points, &num](PhysicsWorld& world,
        const PhysicsRayCastInfo& info, void* data)->bool
    {
        if (num < 5)
        {
            points[num++] = info.contact;
        }
        return true;
    };

    s_currScene->getPhysicsWorld()->rayCast(func, s_centre, point2, nullptr);

    _drawNode->drawSegment(s_centre, point2, 1, Color4F::RED);
```

```
    for (int i = 0; i < num; ++i)
    {
        _drawNode->drawDot(points[i], 3, Color4F(1.0f, 1.0f, 1.0f, 1.0f));
    }
    addChild(_drawNode);

    _angle += 1.5f * (float)M_PI / 180.0f;
}
```



Rect Queries

Rect queries provide a fast way to check roughly which shapes are in an area. It is pretty easy to implement:

```
auto func = [](PhysicsWorld& world, PhysicsShape& shape, void* userData)->bool
{
    //Return true from the callback to continue rect queries
    return true;
}

scene->getPhysicsWorld()->queryRect(func, Rect(0,0,200,200), nullptr);
```

A few examples of using a **rect query** while doing a *logo smash*:

Debugging Physics Body and Shapes

If you ever wish to have red boxes drawn around your **physics bodies** to aid in debugging, simple add these 2 lines to your core, where it makes sense to you. Perhaps `AppDelegate` is a good place.

```
Director::getInstance()->getRunningScene()->getPhysics3DWorld()->setDebugDrawEnable(true);
Director::getInstance()->getRunningScene()->setPhysics3DDebugCamera(cameraObjecct);
```

Disabling Physics

Using the built-in **physics engine** is a good idea. It is solid and advanced. However, if you wish to use an alternative **physics engine** you can. All you need to do is disabling **CC_USE_PHYSICS** in **base/ccConfig.h**.

Audio

Your game will surely need sound! Cocos2d-x provides an audio engine called `SimpleAudioEngine`. It can be used to play background music as well as sound effects through out your game play. `SimpleAudioEngine` is a shared singleton object so you can simple call it from anywhere in your code. When creating a sample **HelloWorld** project we do all the setup required for you, out of the box. It also supports a variety of formats, including **mp3** and **Core Audio Format**

Getting Started

The `SimpleAudioEngine` API is very easy to use.

Play background music

Play an audio file for use as background music. This can be repeated continuously.

```
auto audio = SimpleAudioEngine::getInstance();

// set the background music and continuously play it.
audio->playBackgroundMusic("mymusic.mp3", true);

// set the background music and play it just once.
audio->playBackgroundMusic("mymusic.mp3", false);
```

Play a sound effect.

Play a sound effect.

```
auto audio = SimpleAudioEngine::getInstance();

// play a sound effect, just once.
audio->playEffect("myEffect.mp3", false, 1.0f, 1.0f, 1.0f);
```

Pausing, stopping, resuming music and sound effects

After you start to play music and sound effects you might need to pause, stop or resume after certain operations. This can be done easily.

Pause

```
auto audio = SimpleAudioEngine::getInstance();

// pause background music.
audio->pauseBackgroundMusic();

// pause a sound effect.
audio->pauseEffect();

// pause all sound effects.
audio->pauseAllEffects();
```

Stop

```
auto audio = SimpleAudioEngine::getInstance();

// stop background music.
audio->stopBackgroundMusic();

// stop a sound effect.
audio->stopEffect();

// stops all running sound effects.
audio->stopAllEffects();
```

Resume

```
auto audio = SimpleAudioEngine::getInstance();

// resume background music.
audio->resumeBackgroundMusic();

// resume a sound effect.
audio->resumeEffect();

// resume all sound effects.
audio->resumeAllEffects();
```

Advanced audio functionality

Setup

It is easy to get started using the `SimpleAudioEngine` API. There are considerations to keep in mind when using audio in your game. Mostly when operating on mobile devices such as phones and tablets. What happens when you multi-task on your phone and are switching between apps? Or when a phone call comes in? You need to handle these exceptions in your game. Fortunately, we help you here.

In `AppDelegate.cpp`, notice the following methods:

```
// This function will be called when the app is inactive. When comes a phone call,
// it's be invoked too
void AppDelegate::applicationDidEnterBackground() {
    Director::getInstance()->stopAnimation();

    // if you use SimpleAudioEngine, it must be pause
    // SimpleAudioEngine::getInstance()->pauseBackgroundMusic();
}

// this function will be called when the app is active again
void AppDelegate::applicationWillEnterForeground() {
    Director::getInstance()->startAnimation();

    // if you use SimpleAudioEngine, it must resume here
    // SimpleAudioEngine::getInstance()->resumeBackgroundMusic();
}
```

Notice the commented out lines for `SimpleAudioEngine`? Make sure to uncomment these lines out if you are using audio for background sounds and sound effects.

Pre-loading sound

When your game starts you might want to pre-load the music and effects so they are ready when you need them.

```
auto audio = SimpleAudioEngine::getInstance();

// pre-loading background music and effects. You could pre-load
// effects, perhaps on app startup so they are already loaded
// when you want to use them.
audio->preloadBackgroundMusic("myMusic1.mp3");
audio->preloadBackgroundMusic("myMusic2.mp3");

audio->preloadEffect("myEffect1.mp3");
audio->preloadEffect("myEffect2.mp3");
```

```
// unload a sound from cache. If you are finished with a sound and
// you wont use it anymore in your game. unload it to free up
// resources.
audio->unloadEffect("myEffect1.mp3");
```

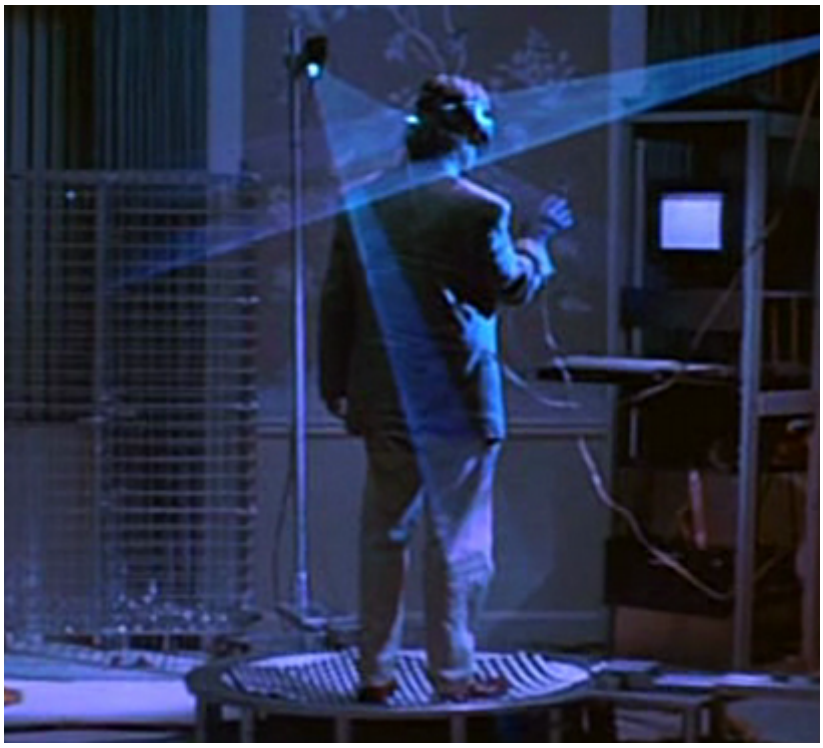
Volume

You can increase and decrease the volume of your sounds and music programmatically.

```
auto audio = SimpleAudioEngine::getInstance();

// setting the volume specifying value as a float
audio->setEffectsVolume(5.0f);
```

Virtual Reality (VR)

You have probably heard the term **Virtual Reality** or **VR** used before. **VR** isn't new. Its roots can be traced back to earlier than the 1970's. The original goal of **VR** was to take an environment or situation, both realistic and unrealistic and let the user feel what it is like to experience it by simulating their physical presence in the environment. You can think of it as *transporting* the user to another experience, all the while never leaving their physical surroundings. You might even associate **VR** with wearing a *head-mounted display* or special gloves or even taking place on a special platform.



Modern **VR** is focused around *games* and *immersive video*.

Is VR production ready?

No, **VR** is still in the early phases of development. Please consider it **experimental**! In fact, we are providing a **generic renderer** implementation to use as a proof-of-concept. You can use this in a simulator or with a **Google Cardboard** *head-mounted display*. You cannot trust the **generic renderer** to produce 100% correct results. It is always necessary to test with a supported SDK and supported hardware.

We support the popular **VR SDKs**:

SDK	Company	Runtime Platform
GearVR	Samsung	Galaxy Note 5/S6/S6 Edge/S6 Edge+
GVR(Cardboard And Daydream)	Google	Android 4.4 (KitKat) or higher
DeepoonVR	Deepoon	Galaxy Note 5/S6/S6 Edge/S6 Edge+
OculusVR	Oculus	Oculus Rift(Windows 7+)

Is your game a good VR candidate?

If, late on a Friday evening, after a night of dinner, dance and drink, you find yourself thinking ***let me take my current game and turn it into a VR game***. Pause… longer… and make sure you are not dreaming! Seriously, you need to stop and ask yourself a few questions:

- How do I interact with the game currently? Touch? Gamepad? Keyboard?
- In 2d games: what does moving the camera mean? 2d games are not usually made in the *first person*.
- Is your game done in a *first person* scenario? *First person* games can be made into **VR** games easier than others types of games.
- Is my 2D or 3D game a good candidate for a VR game after answering the above questions?

When using **VR** it is important to note the following items:

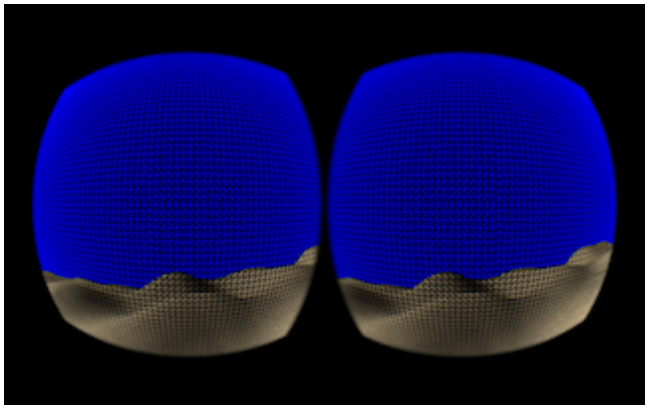
- Touch events don't work as expected when developing for **VR**. In fact, touch events should be disabled in **VR** games.
- **VR** games should be configured to use a gamepad and/or another external input device, such as a *head-mounted display*.

With this knowledge and a *can do* attitude, get started…

How to get started

First, it is important to double check your hardware to make sure your device supports **VR**. VR needs two things:

- Stereo rendering (distortion mesh): available on every platform



- headset input: available only on iOS and Android

Importing VR

Second, use the **Cocos Package Manager**, which is part of the **Cocos Command-Line Tool** to add **VR** to your project:

You always need to **import** the `vrsdkbase`. This step takes care of modifying your projects to support **VR**.

```
$ cocos package import -v -b vrsdkbase --anysdk
```

Notice in `AppDelegate.cpp` code has been added to enable **VR**:

```
// VR_PLATFORM_SOURCES_BEGIN
auto vrImpl = new VRGenericRenderer;
glview->setVR(vrImpl);
// VR_PLATFORM_SOURCES_END
```

Import the **VR SDK** that you need. Currently, **Gear**, **Deepoon**, **GVR** and **Oculus** are supported.

```
$ cocos package import -v -b SDK_NAME --anysdk
```

Examples:

```
# add the GearVR package
$ cocos package import -v -b gearvr --anysdk

# add the Deepoon VR package
$ cocos package import -v -b deepoon --anysdk

# add the Google VR package
$ cocos package import -v -b gvr --anysdk

# add the Oculus VR package
$ cocos package import -v -b oculus --anysdk
```

Compiling and Running with VR

IOS

If you are running iOS, you are limited to running the **generic renderer** on hardware only, you can use **cocos compile cocos run** as you typically would.

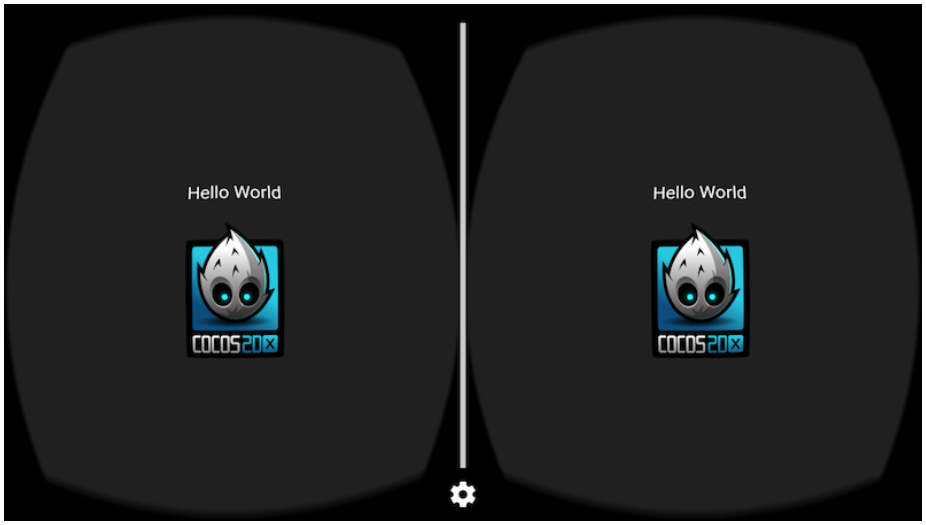
Android

If you are running on **Android** and planning on targeting a specific **VR SDK** you need to perform a few additional steps. Running **switchVRPlatform.py** from your projects root directory will take care of everything. Here is an example for installing **GearVR** in C++, JavaScript and Lua:

```
## in C++

# first, install vrsdkbase
$ cocos package import -v -b vrsdkbase --anysdk

# second, install GearVR
$ cocos package import -v -b gearvr --anysdk

# third, switch to using GearVR
$ python vrsdks/switchVRPlatform.py -p gearvr-sdk
```

```
## in JavaScript and Lua

# first, install vrsdkbase
$ cocos package import -v -b vrsdkbase --anysdk

# second, install GearVR
$ cocos package import -v -b gearvr --anysdk

# third, switch to using GearVR
$ python frameworks/runtime-src/vrsdks/switchVRPlatform.py -p gearvr-sdk
```

Attention: you should using `$ python vrsdks/switchVRPlatform.py -h` to check the name of SDK, here is `gearvr-sdk`.



For Android there is also a few special steps that must happen. These are dependent upon your **Runtime Platform**. Please refer to the table at the start of this document.

GearVR/Deepoon VR/GVR Compilation and Running.

Running **GearVR**, **Deepoon VR** or **Google VR** on **Android** requires a change in compile flags. Example:

```
# from a command-line
$ cocos run -p android --app-abi armeabi-v7a

# using Android Studio
$ cocos run -p android --android-studio --app-abi armeabi-v7a
```

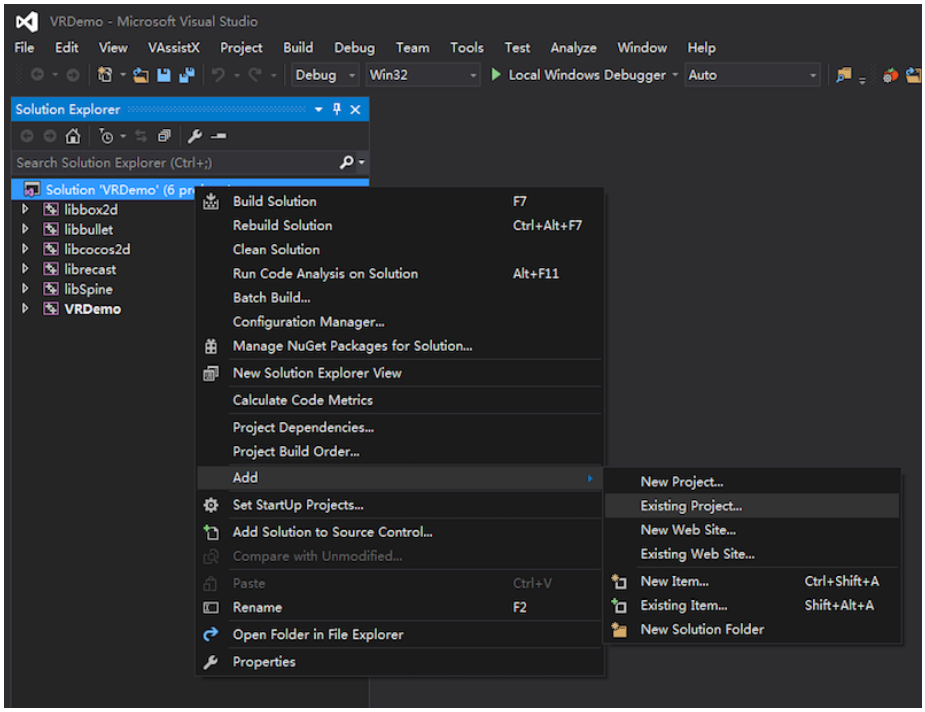
Attention: All mobile VRSDK(GearVR/Deepoon VR/GVR) only support armeabi-v7a architecture. GVR only support Android Studio. So it can only use the second command to compilation.

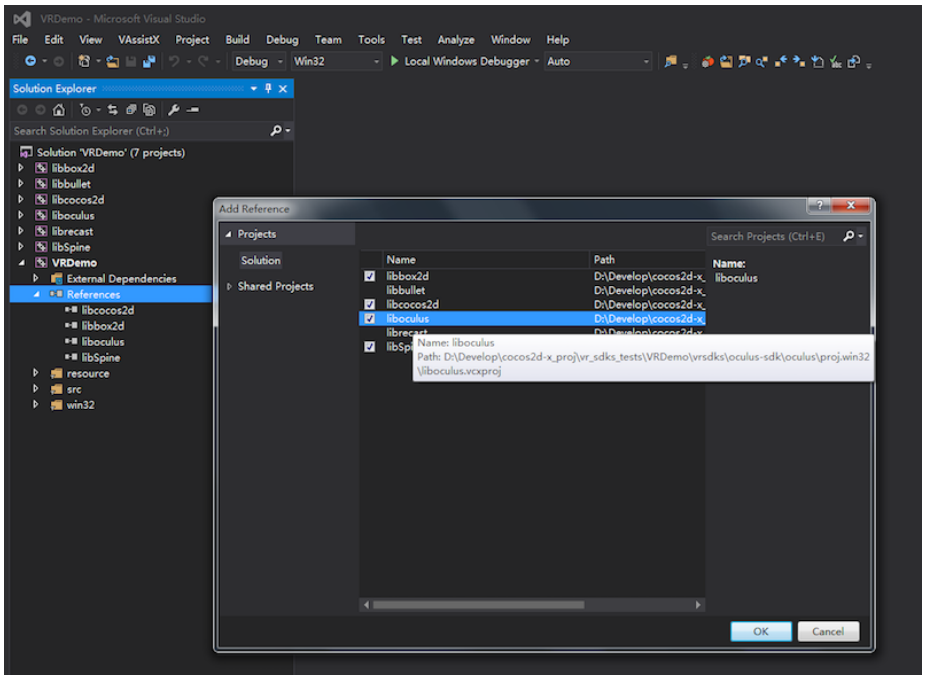
If **GearVR** or **Deepoon VR** crashes at runtime, please check to ensure you have an Oculus signature file in **assets** folder.

Oculus Compilation

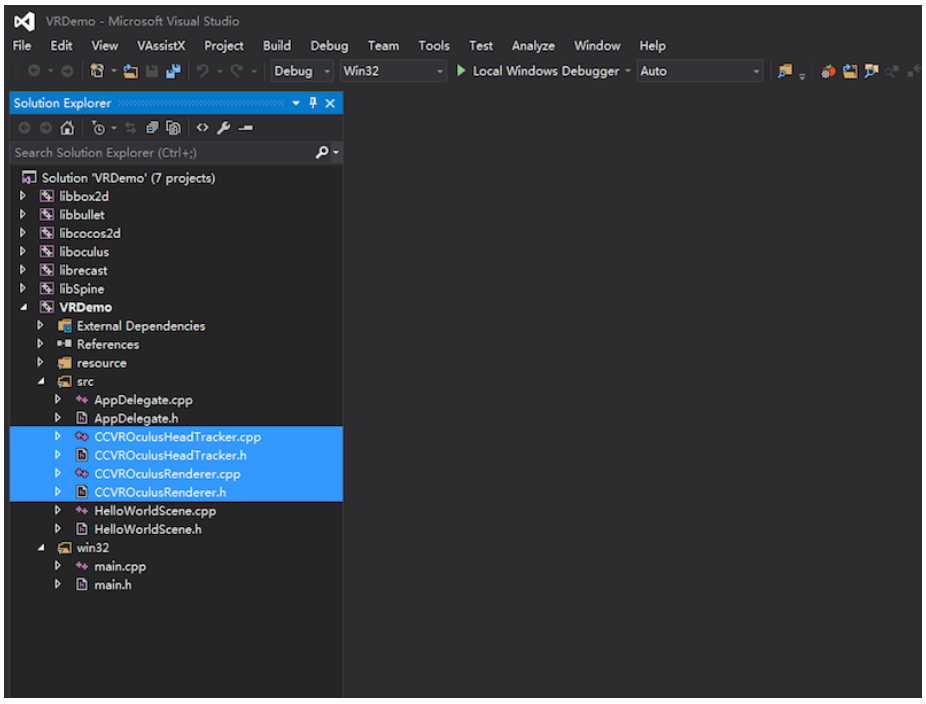
OculusVR is for the desktop PC platform. This requires **Visual Studio 2015**.

First, import **liboculus.vcxproj** into your project(in `oculus-sdk/oculus/proj.win32/` folder) and add a reference to it:

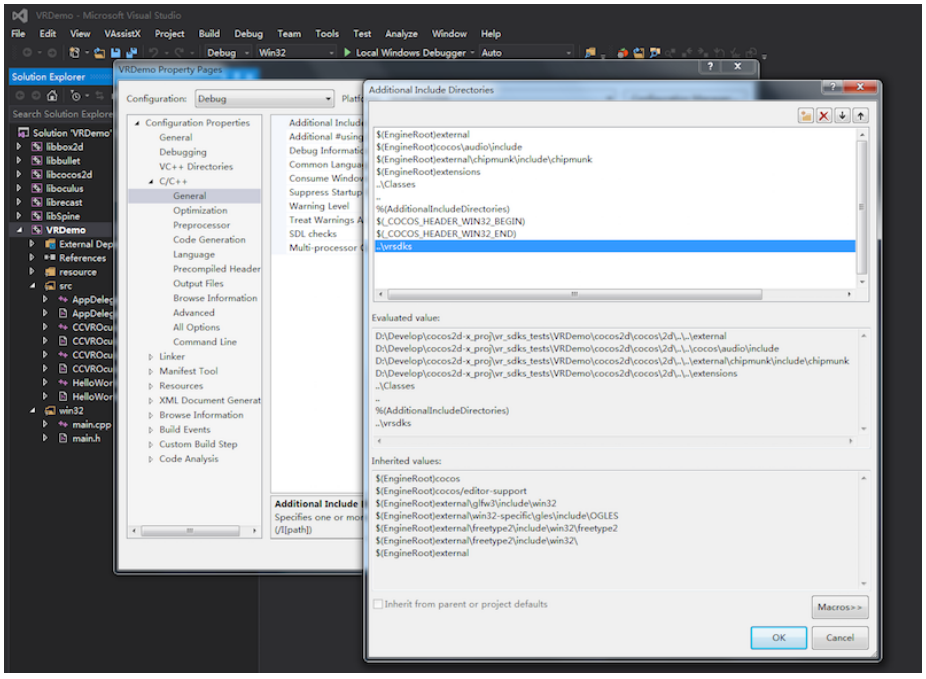




Second, import the `CCVROculusRenderer` and `CCVROculusHeadTracker` classes(in `oculus-sdk/` folder):



Finally, add the search path of VR-SDK (`..\vrsdks`) to your project:



If **Oculus** crashes at runtime, please check your installation of the Oculus Rift Runtime.

Advanced Topics

Wow! You are on the last chapter. Good Job! By now you should feel comfortable creating your games with Cocos2d-x. However, please realize there is no limit to what you can create. This chapter covers **advanced** concepts. Note that this chapter gets more technical in its content and format.

File System Access

Even though you can use functions in **stdio.h** to access files it can be inconvenient for a few reasons: * You need to invoke system specific API to get full path of a file. * Resources are packed into .apk file on Android after installing. * You want to load a resource (such as a picture) based on resolution automatically.

The `FileUtils` class has been created to resolve these issues. `FileUtils` is a helper class to access files under the location of your `Resources` directory. This includes reading data from a file and checking file existence.

Functions to read file content

These functions will read different type of files and will return different data types:

function name	return type	support path type
getStringFromFile	std::string	relative path and absolute path
getDataFromFile	cocos2d::Data	relative path and absolute path
getFileDataFromZip	unsigned char*	absolute path
getValueMapFromFile	cocos2d::ValueMap	relative path and absolute path
getValueVectorFromFile	std::string	cocos2d::ValueVector

Functions to manage files or directories

These functions will manage a file or a directory:

function name	support path type
isFileExist	relative path and absolute path
isDirectoryExist	relative path and absolute path
createDirectory	absolute path
removeDirectory	absolute path

function name	support path type
removeFile	absolute path
renameFile	absolute path
getFileSize	relative path and absolute path

Networking with HTTP

Sometimes it might be helpful to obtain resources or data from another source. One common way of doing this is by using an `HTTP` request.

HTTP networking has three steps: 1. Create an `HttpRequest` 2. Create a **setResponseCallback()** callback function for replying to requests. 3. Send `HttpRequest` by `HttpClient`

`HttpRequest` can have four types: **POST**, **PUT**, **DELETE**, **UNKNOWN**. Unless specified the default type is **UNKNOWN**. The `HTTPClient` object controls sending the **request** and receiving the data on a **callback**.

Working with an `HTTPRequest` is quite simple:

```
HttpRequest* request = new (std :: nothrow) HttpRequest();
request->setUrl("http://just-make-this-request-failed.com");
request->setRequestType(HttpRequest::Type::GET);
request->setResponseCallback(CC_CALLBACK_2 (HttpClientTest::onHttpRequestCompleted, this));

HttpClient::getInstance()->sendImmediate(request);

request->release();
```

Notice that we specified a **setResponseCallback()** method for when a response is received. By doing this we can look at the data returned and use it how we might need to. Again, this process is simple and we can do it with ease:

```
void HttpClientTest::onHttpRequestCompleted(HttpClient* sender, HttpResponse* response)
{
  if (!response)
  {
    return;
  }

  // Dump the data
  std::vector<char>* buffer = response->getResponseData();

  for (unsigned int i = 0; i <buffer-> size (); i ++)
  {
    log ("% c", (* buffer) [i]);
```

```
    }
}
```

Shaders and Materials

What is a Shader

From wikipedia:

In the field of computer graphics, a shader is a computer program that is used to do shading: the production of appropriate levels of color within an image, or, in the modern era, also to produce special effects or do video post-processing. A definition in layman's terms might be given as "a program that tells a computer how to draw something in a specific and unique way".

In other words, it is a piece of code that runs on the GPU (not CPU) to draw the different Cocos2d-x Nodes.

Cocos2d-x uses the OpenGL ES Shading Language v1.0 for the shaders. But describing the GLSL language is outside the scope of this document. In order to learn more about the language, please refer to: OpenGL ES Shading Language v1.0 Spec.

In Cocos2d-x, all `Node` objects that are **renderable** use shaders. As an example `Sprite` uses optimized shaders for 2d sprites, `Sprite3D` uses optimized shaders for 3d objects, and so on.

Customizing Shaders

Users can change the predefined shaders from any Cocos2d-x `Node` by calling:

```
sprite->setGLProgramState(programState);
sprite3d->setGLProgramState(programState);
```

The `GLProgramState` object contains two important things:

- A `GLProgram`: Basically this is *the* shader. It contains a vertex and fragment shader.
- And the **state**, which basically are the uniforms of the shader.

In case you are not familiar with the term *uniform* and why it is needed, please refer to the OpenGL Shading Language Specification

Setting uniforms to a `GLProgramState` is as easy as this:

```
glProgramState->setUniformFloat("u_progress", 0.9);
glProgramState->setUniformVec2("u_position", Vec2(x,y));
glProgramState->setUniformMat4("u_transform", matrix);
```

You can even set callbacks as a uniform value:

```
glProgramState->setUniformCallback("u_progress", [](GLProgram* glProgram, Uniform* uniform)
  {
      float random = CCRANDOM_0_1();
      glProgram->setUniformLocationWith1f(uniform->location, random);
  }
);
```

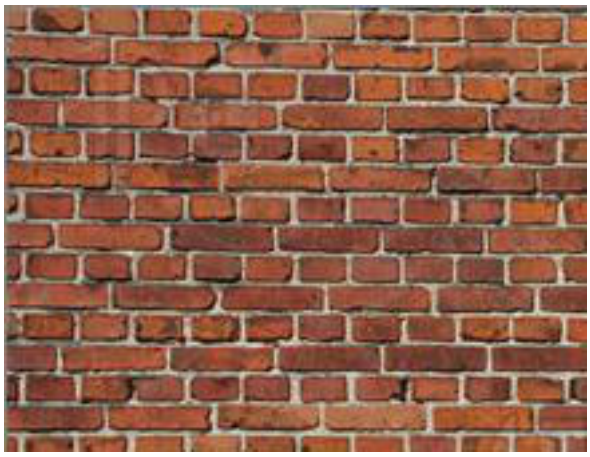
And although it is possible to set `GLProgramState` objects manually, an easier way to do it is by using `Material` objects.

What is a Material

Assume that you want to draw a sphere like this one:

The first thing that you have to do is to define its geometry, something like this:

…and then define the brick texture, like:



- But what if you want to use a lower quality texture when the sphere is far away from the camera?
- or what if you want to apply a blur effect to the bricks?
- or what if you want to enable or disable lighting in the sphere ?

The answer is to use a `Material` instead of just a plain and simple texture. In fact, with `Material` you can have more than one texture, and much more features like multi-pass rendering.

`Material` objects are created from `.material` files, which contain the following information:

- `Material` can have one or more `Technique` objects
- each `Technique` can have one more `Pass` objects
- each `Pass` object has:
- a `RenderState` object,
- a `Shader` object including the uniforms

As an example, this is how a material file looks like:

```
// A "Material" file can contain one or more materials
material spaceship
{
    // A Material contains one or more Techniques.
    // In case more than one Technique is present, the first one will be the default one
    // A "Technique" describes how the material is going to be renderer
    // Techniques could:
    //  - define the render quality of the model: high quality, low quality, etc.
    //  - lit or unlit an object
    // etc...
    technique normal
    {
        // A technique can contain one or more passes
        // A "Pass" describes the "draws" that will be needed
        //   in order to achieve the desired technique
```

```
        // The 3 properties of the Passes are shader, renderState and sampler
        pass 0
        {
            // shader: responsible for the vertex and frag shaders, and its uniforms
            shader
            {
                vertexShader = Shaders3D/3d_position_tex.vert
                fragmentShader = Shaders3D/3d_color_tex.frag

                // uniforms, including samplers go here
                u_color = 0.9,0.8,0.7
                // sampler: the id is the uniform name
                sampler u_sampler0
                {
                    path = Sprite3DTest/boss.png
                    mipmap = true
                    wrapS = CLAMP
                    wrapT = CLAMP
                    minFilter = NEAREST_MIPMAP_LINEAR
                    magFilter = LINEAR
                }
            }
            // renderState: responsible for depth buffer, cullface, stencil, blending, etc.
            renderState
            {
                cullFace = true
                cullFaceSide = FRONT
                depthTest = true
            }
        }
    }
}
```

And this is how to set a `Material` to a `Sprite3D`:

```
Material* material = Material::createWithFilename("Materials/3d_effects.material");
sprite3d->setMaterial(material);
```

And if you want to change between different `Techniques`, you have to do:

```
material->setTechnique("normal");
```

Techniques

Since you can bind only one `Material` per `Sprite3D`, an additional feature is supported that's designed to make it quick and easy to change the way you render the parts at runtime. You can define multiple techniques by giving them different names. Each one can have a completely

different rendering technique, and you can even change the technique being applied at runtime by using **Material::setTechnique(const std::string& name)**. When a material is loaded, all the techniques are loaded ahead too. This is a practical way of handling different light combinations or having lower-quality rendering techniques, such as disabling bump mapping, when the object being rendered is far away from the camera.

Passes

A `Technique` can have one or more **passes** That is, multi-pass rendering. And each `Pass` has two main objects:

- `RenderState`: contains the GPU state information, like **depthTest**, **cullFace**, **stencilTest**, etc.
- `GLProgramState`: contains the shader (`GLProgram`) that is going to be used, including its uniforms.

Material file format in detail

Material uses a file format optimized to create Material files. This file format is very similar to other existing Material file formats, like GamePlay3D's and OGRE3D's.

Notes:

- Material file extensions do not matter. Although it is recommended to use **.material** as extension
- **id** is optional for material, technique and pass
- Materials can inherit values from another material by optionally setting a **parent_material_id**
- Vertex and fragment shader file extensions do not matter. The convention in Cocos2d-x is to use **.vert** and **frag**

```
// When the .material file contains one material
sprite3D->setMaterial("Materials/box.material");
// When the .material file contains multiple materials
sprite3D->setMaterial("Materials/circle.material#wood");
```

```
material material_id : parent_material_id
{
  renderState {}                    [0..1]   block
  technique id {}                   [0..*]   block
}
```

```
technique technique_id
{
  renderState {}              [0..1]   block
```

```
  pass id {}                    [0..*]    block
}
```

```
passpass_id
{
  renderState {}    [0..1]    block
  shader {}         [0..1]    block
}
```

```
renderState
{
  blend = false                                  [0..1]    bool
  blendSrc = BLEND_ENUM                          [0..1]    enum
  blendDst = BLEND_ENUM                          [0..1]    enum
  cullFace = false                               [0..1]    bool
  depthTest = false                              [0..1]    bool
  depthWrite = false                             [0..1]    bool
}
  frontFace = CW | CCW                           [0..1]    enum
  depthTest = false                              [0..1]    bool
  depthWrite = false                             [0..1]    bool
  depthFunc = FUNC_ENUM                          [0..1]    enum
  stencilTest = false                            [0..1]    bool
  stencilWrite = 4294967295                      [0..1]    uint
  stencilFunc = FUNC_ENUM                        [0..1]    enum
  stencilFuncRef = 0                             [0..1]    int
  stencilFuncMask = 4294967295                   [0..1]    uint
  stencilOpSfail = STENCIL_OPERATION_ENUM        [0..1]    enum
  stencilOpDpfail = STENCIL_OPERATION_ENUM       [0..1]    enum
  stencilOpDppass = STENCIL_OPERATION_ENUM       [0..1]    enum
```

```
shadershader_id
{
  vertexShader = res/colored.vert          [0..1]    file path
  fragmentShader = res/colored.frag        [0..1]    file path
  defines = semicolon separated list       [0..1]    string

  uniform_name = scalar | vector           [0..*]    uniform
  uniform_name = AUTO_BIND_ENUM            [0..*]    enum
  sampler uniform_name {}                  [0..*]    block
}
```

sampler uniform_name		
{		
path = res/wood.png | @wood	[0..1]	image path
mipmap = bool	[0..1]	bool
wrapS = REPEAT | CLAMP	[0..1]	enum
wrapT = REPEAT | CLAMP	[0..1]	enum
minFilter = TEXTURE_MIN_FILTER_ENUM	[0..1]	enum
magFilter = TEXTURE_MAG_FILTER_ENUM	[0..1]	enum
}		

Enums:

TEXTURE_MIN_FILTER_ENUM	
NEAREST	Lowest quality non-mipmapped
LINEAR	Better quality non-mipmapped
NEAREST_MIPMAP_NEAREST	Fast but low quality mipmapping
LINEAR_MIPMAP_NEAREST	
NEAREST_MIPMAP_LINEAR	
LINEAR_MIPMAP_LINEAR	Best quality mipmapping

TEXTURE_MAG_FILTER_ENUM	
NEAREST	Lowest quality
LINEAR	Better quality

BLEND_ENUM	
ZERO	ONE_MINUS_DST_ALPHA
ONE	CONSTANT_ALPHA
SRC_ALPHA	ONE_MINUS_CONSTANT_ALPHA
ONE_MINUS_SRC_ALPHA	SRC_ALPHA_SATURATE
DST_ALPHA	

CULL_FACE_SIDE_ENUM	
BACK	Cull back-facing polygons.
FRONT	Cull front-facing polygons.
FRONT_AND_BACK	Cull front and back-facing polygons.

FUNC_ENUM	
NEVER	ALWAYS
LESS	GREATER
EQUAL	NOTEQUAL
LEQUAL	GEQUAL

STENCIL_OPERATION_ENUM	
KEEP	REPLACE
ZERO	INVERT
INCR	DECR
INCR_WRAP	DECR_WRAP

Types:

- scalaris float, int or bool.
- vector is a comma separated list of floats.

Predefined uniforms

The following are predefined uniforms used by Cocos2d-x that can be used in your shaders:

- `CC_PMatrix`: A `mat4` with the projection matrix
- `CC_MVMatrix`: A `mat4` with the Model View matrix
- `CC_MVPMatrix`: A `mat4` with the Model View Projection matrix
- `CC_NormalMatrix`: A `mat4` with Normal Matrix
- `CC_Time`: a `vec4` with the elapsed time since the game was started
- CC_Time[0] = time / 10;
- CC_Time[1] = time;
- CC_Time[2] = time * 2;
- CC_Time[3] = time * 4;
- `CC_SinTime`: a `vec4` with the elapsed time since the game was started:
- CC_SinTime[0] = time / 8;
- CC_SinTime[1] = time / 4;
- CC_SinTime[2] = time / 2;
- CC_SinTime[3] = sinf(time);
- `CC_CosTime`: a `vec4` with the elapsed time since the game was started:
- CC_CosTime[0] = time / 8;
- CC_CosTime[1] = time / 4;
- CC_CosTime[2] = time / 2;
- CC_CosTime[3] = cosf(time);
- `CC_Random01`: A `vec4` with four random numbers between 0.0f and 1.0f
- `CC_Texture0`: A `sampler2D`

- `CC_Texture1`: A `sampler2D`
- `CC_Texture2`: A `sampler2D`
- `CC_Texture3`: A `sampler2D`

How to optimize the graphics performance of your Cocos2d-x games

Golden rules

Know the bottlenecks and optimize the bottlenecks.

When doing optimization, we should always stick to this rule. Only 20% code in your system contribute to the 80% performance issue.

Always use tools to profile the bottleneck, don't guess randomly.

There are many tools available now for profiling the graphics performance. Though we are optimize the performance of Android games, but XCode could also be helpful to debugging.

- Xcode: https://github.com/rstrahl/rudistrahl.me/blob/master/entries/Debugging-OpenGL-ES-With-Xcode-Profile-Tools.md and the official document: https://developer.apple.com/library/ios/documentation/3D

There are three major mobile GPU vendors nowadays and they provide decent graphics profiling tools:

- For ARM Mali GPU: http://malideveloper.arm.com/resources/tools/mali-graphics-debugger/
- For Imagination PowerVR GPU: https://community.imgtec.com/developers/powervr/tools/pvrtune/
- For Qualcomm Adreno GPU: https://developer.qualcomm.com/software/adreno-gpu-profiler

Use these tools when you suffer from graphics issues. **But not at the first beginning, usually the bottleneck resides on CPU.**

Know your target device and your game engine

Know the CPU/GPU family of your target device which is important when sometimes the performance issues only occurs on certain kind of devices. And you will find they share the same kind of GPU(ARM or PowerVR or Mali).

Know the limitations of your currently used game engine is also important. If you know how your engine organize the graphics command, how your engine do batch drawing. You could avoid many common pitfalls during coding.

The principle of "Good enough".

("If the viewer cannot tell the difference between differently rendered images always use the cheaper implementation".) As we know a PNG with RGBA444 pixel format has lower graphics quality than the one with RGBA888 pixel format. But if we can't tell the difference between the two, we should stick to RGBA4444 pixel format. The RGBA444 format use less memory and it will less likely to cause the memory issue and bandwidth issue.

It is the same goes for the audio sample rate.

Common Bottlenecks

As a rules of thumb, your game will suffer CPU bottlenecks easily than graphics bottlenecks.

The CPU is often limited by the number of draw calls and the heavy compute operations in your game loop

Try to minimize the total draw calls of your game. We should use batch draw as much as possible. Cocos2d-x 3.x has auto batch support, but it needs some effort to make it work.

Also try avoid IO operations when players are playing your game. Try to preload your spritesheets, audios, TTF fonts etc.

Also don't do heavy compute operations in your game loop which means don't let the heavy operations called 60 times per frame.

Never!

The GPU is often limited by the overdraw(fillrate) and bandwidth.

If you are creating a 2D game and you don't write complex shaders, you might won't suffer GPU issues. But the overdraw problem still has trouble and it will slow your graphics performance with too much bandwidth consumption.

Though modern mobile GPU have TBDR(Tiled-based Defered Rendering) architecture, but only PowerVR's HSR(Hidden Surface Removal) could reduce the overdraw problem significantly. Other GPU vendors only implement a TBDR + early-z testing, it only reduce the overdraw problem when you submit your opaque geometry with the order(font to back). And Cocos2d-x always submit rendering commands ordered from back to front. Because in 2D, we might have many transparency images and only in this order the blending effect is correct.

Note: By using poly triangles, we could improve the fillrate. Please refer to this article for more information: https://www.codeandweb.com/texturepacker/tutorials/cocos2d-x-performance-optimization

But don't worry too much of this issue, it doesn't perform too bad in practice.

Simple checklist to make your Cocos2d-x game faster

1. Always use batch drawing. Package sprite images in the same layer into a large atlas(Texture packer could help).
2. As rule of thumb, try to keep your draw call below 50. In other words, try to minimize your draw call number.
3. Prefer 16bit(RGBA4444+dithering) over raw 32bit(RGBA8888) textures.
4. Use compressed textures: In iOS use PVRTC texture. In Android platform, use ETC1. but ETC1 doesn't has alpha, you might need to write a custom shader and provide a separate ETC1 image for the alpha channel.
5. Don't use system font as your game score counter. It's slow. Try to use TTF or BMFont, BMfont is better.
6. Try to preload audio and other game objects before usage.
7. Use armeabi-v7a to build Android native code and it will enable neon instructors which is very fast.
8. Bake the lighting rather than using the dynamic light.
9. Avoid using complex pixel shaders.
10. Avoid using **discard** and alpha test in your pixel shader, it will break the HSR(Hidden surface removal). Only use it when necessary.