12 February 2010

How To Make A Simple iPhone Game with Cocos2D Tutorial

Ninjas Going Pew-Pew!

Update 7/1/12: Chinese version now available!

Cocos2D is a powerful library for the iPhone that can save you a lot of time while building your iPhone game. It has sprite support, cool graphical effects, animations, physics libraries, sound engines, and a lot more.

I am just starting to learn Cocos2D, and while there are various useful tutorials on getting started with Cocos2D out there, I couldn't find anything quite like what I was looking for – making a very simple but functional game with animation, collisions, and audio without using too many advanced features. I ended up making a simple game of my own, and thought I'd write a tutorial series based on my experience in case it might be useful to other newcomers.

This tutorial series will walk you through the process of creating a simple game for your iPhone with Cocos2D, from start to finish. You can follow along with the series, or just jump straight to the sample project at the end of the article. And yes. There will be ninjas.

(Jump to Part 2 or Part 3 of the series.)

Downloading and Installing Cocos2D

You can download Cocos2D from the Cocos2D Google Code page.

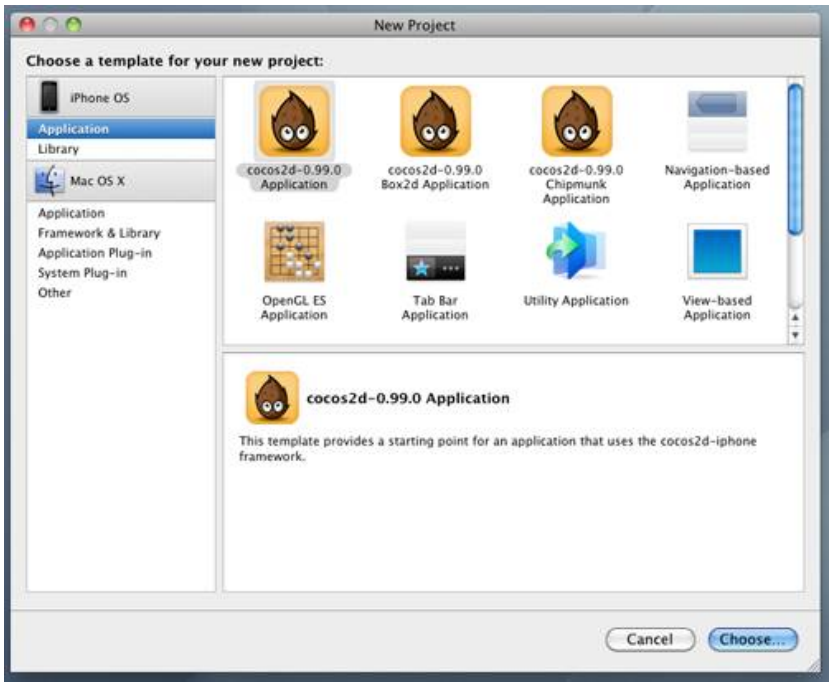
After you pull down the code, you'll want to install the useful project templates. Open up a Terminal window to the directory you downloaded Cocos2D to, and enter the following command: ./install-templates.sh -f -u

Note that you can optionally pass a parameter to the install script if you have XCode installed to a non-standard directory (like you might

have done if you have more than one version of the SDK on your machine).

Hello, Cocos2D!

Let's start by getting a simple Hello World project up and running by using the Cocos2D template we just installed. Start up XCode and create a new Cocos2D project by selecting the cocos2d Application template, and name the project "Cocos2DSimpleGame".



Go ahead and build and run the template as-is. If all works OK, you should see the following:

Cocos2D is organized into the concept of "scenes", which are kind of like "levels" or "screens" for a game. For example you might have a scene for the initial menu for the game, another for the main action of the game, and a game over scene to end. Inside scenes, you can have a number of layers (kind of like in Photoshop), and layers can contain nodes such as sprites, labels, menus, or more. And nodes can contain other nodes as well (i.e. a sprite could have a child sprite inside it).

If you take a look at the sample project, you'll see there's just one layer – HelloWorldLayer – and we're going to start implementing our main gameplay in there. Go ahead and open it up – you'll see that right now in the init method it's adding a label that says "Hello World" to the layer. We're going to take that out, and put a sprite in instead.

Adding A Sprite

Before we can add a sprite, we'll need some images to work with. You can either create your own, or use the ones my lovely wife has created for the project: a Player image, a Projectile image, and a Target image.

Once you've obtained the images, drag them over to the resources folder in XCode, and make sure "Copy items into destination group's folder (if needed)" is checked.

Now that we have our images, we have to figure out where we want to place the player. Note that in Cocos2D the bottom left corner of the screen has coordinates of (0,0) and the x and y values increase as you move to the upper right. Since this project is in landscape mode, this means that the upper right corner is (480, 320).

Also note that by default when we set the position of an object, the position is relative to the center of the sprite we are adding. So if we wanted our player sprite to be aligned with the left edge of the screen horizontally, and vertically centered:

- For the x coordinate of the position, we'd set it to [player sprite's width]/2.
- For the y coordinate of the position, we'd set it to [window height]/2

Here's a picture that helps illustrate this a bit better:

So let's give it a shot! Open up the Classes folder and click on HelloWorldLayer.m, and replace the init method with the following:

```
-(id) init
{
  if( (self=[super init] )) {
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    CCSprite *player = [CCSprite spriteWithFile:@"Player.png"
      rect:CGRectMake(0, 0, 27, 40)];
    player.position = ccp(player.contentSize.width/2, winSize.height/2);
    [self addChild:player];
  }
  return self;
}
```

You can compile and run it, and your sprite should appear just fine, but note that the background defaults to black. For this artwork, white would look a lot better. One easy way to set the background of a layer in Cocos2D to a custom color is to use the CCLayerColor class. So let's give this a shot. Click on HelloWorldLayer.h and change the HelloWorld interface declaration to read as follows:

```
@interface HelloWorldLayer : CCLayerColor
```

Then click on HelloWorldLayer.m and make a slight modification to the init method so we can set the background color to white:

```
if( (self=[super initWithColor:ccc4(255,255,255,255)] )) {
```

Go ahead and compile and run, and you should see your sprite on top of a white background. w00t our ninja looks ready for action!

60.0

Moving Targets

Next we want to add some targets into our scene for our ninja to combat. To make things more interesting, we want the targets to be moving – otherwise there wouldn't be much of a challenge! So let's create the targets slightly off screen to the right, and set up an action for them telling them to move to the left.

Add the following method right before the init method:

```
-(void)addTarget {

  CCSprite *target = [CCSprite spriteWithFile:@"Target.png"
    rect:CGRectMake(0, 0, 27, 40)];

  // Determine where to spawn the target along the Y axis
  CGSize winSize = [[CCDirector sharedDirector] winSize];
  int minY = target.contentSize.height/2;
  int maxY = winSize.height - target.contentSize.height/2;
  int rangeY = maxY - minY;
  int actualY = (arc4random() % rangeY) + minY;

  // Create the target slightly off-screen along the right edge,
  // and along a random position along the Y axis as calculated above
  target.position = ccp(winSize.width + (target.contentSize.width/2), actualY);
  [self addChild:target];

  // Determine speed of the target
  int minDuration = 2.0;
  int maxDuration = 4.0;
  int rangeDuration = maxDuration - minDuration;
  int actualDuration = (arc4random() % rangeDuration) + minDuration;

  // Create the actions
  id actionMove = [CCMoveTo actionWithDuration:actualDuration
    position:ccp(-target.contentSize.width/2, actualY)];
  id actionMoveDone = [CCCallFuncN actionWithTarget:self
    selector:@selector(spriteMoveFinished:)];
  [target runAction:[CCSequence actions:actionMove, actionMoveDone, nil]];

}
```

I've spelled out things in a verbose manner here to make things as easy to understand as possible. The first part should make sense based on what we've discussed so far: we do some simple calculations to determine where we want to create the object, set the position of the object, and add it to the scene the same way we did for the player sprite.

The new element here is adding actions. Cocos2D provides a lot of extremely handy built-in actions you can use to animate your sprites, such as move actions, jump actions, fade actions, animation actions, and more. Here we use three actions on the target:

- *CCMoveTo:* We use the CCMoveTo action to direct the object to move off-screen to the left. Note that we can specify the duration for how long the movement should take, and here we vary the speed randomly from 2-4 seconds.
- *CCCallFuncN:* The CCCallFuncN function allows us to specify a callback to occur on our object when the action is performed. We

are specifying a callback called “spriteMoveFinished” that we haven’t written yet – more below.
- *CCSequence:* The CCSequence action allows us to chain together a sequence of actions that are performed in order, one at a time. This way, we can have the CCMoveTo action perform first, and once it is complete perform the CCCallFuncN action.

Next, add the callback function that we referred to in the CCCallFuncN action. You can add this right before addTarget:

```
-(void)spriteMoveFinished:(id)sender {
  CCSprite *sprite = (CCSprite *)sender;
  [self removeChild:sprite cleanup:YES];
}
```

The purpose of this function is to remove the sprite from the scene once it is off-screen. This is important so that we don’t leak memory over time by having tons of unused sprites sitting off-screen. Note that there are other (and better) ways to address this problem such as having reusable arrays of sprites, but for this beginner tutorial we are taking the simple path.

One last thing before we go. We need to actually call the method to create targets! And to make things fun, let’s have targets continuously spawning over time. We can accomplish this in Cocos2D by scheduling a callback function to be periodically called. Once per second should do for this. So add the following call to your init method before you return:

```
[self schedule:@selector(gameLogic:) interval:1.0];
```

And then implement the callback function simply as follows:

```
-(void)gameLogic:(ccTime)dt {
  [self addTarget];
}
```

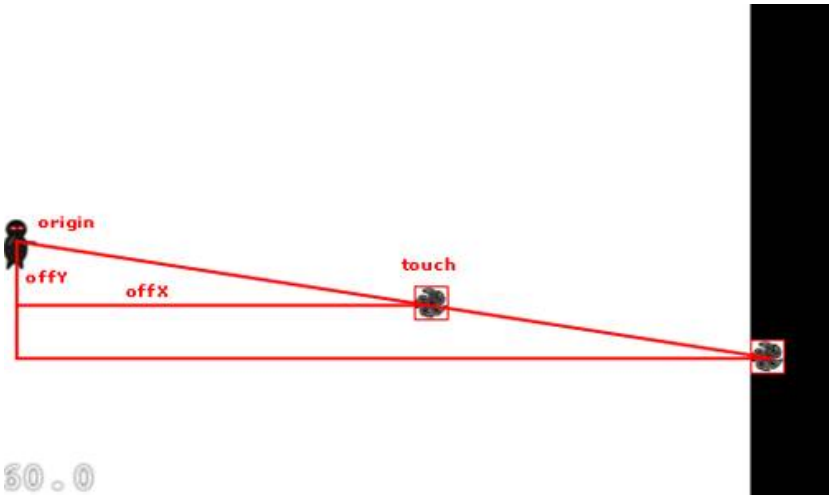
That’s it! So now if you compile and run the project, now you should see targets happily moving across the screen:

Shooting Projectiles

At this point, the ninja is just begging for some action – so let’s add shooting! There are many ways we could implement shooting, but for this game we are going to make it so when the user taps the screen, it shoots a projectile from the player in the direction of the tap.

I want to use a CCMoveTo action to implement this to keep things at a beginner level, but in order to use this we have to do a little math. This is because the CCMoveTo requires us to give a destination for the projectile, but we can’t just use the touch point because the touch point represents just the direction to shoot relative to the player. We actually want to keep the bullet moving through the touch point until the bullet goes off-screen.

Here’s a picture that illustrates the matter:



So as you can see, we have a small triangle created by the x and y offset from the origin point to the touch point. We just need to make a big triangle with the same ratio – and we know we want one of the endpoints to be off the screen.

Ok, so onto the code. First we have to enable touches on our layer. Add the following line to your init method:

```
self.isTouchEnabled = YES;
```

Since we've enabled touches on our layer, we will now receive callbacks on touch events. So let's implement the ccTouchesEnded method, which is called whenever the user completes a touch, as follows:

```
- (void)ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

  // Choose one of the touches to work with
  UITouch *touch = [touches anyObject];
  CGPoint location = [touch locationInView:[touch view]];
  location = [[CCDirector sharedDirector] convertToGL:location];

  // Set up initial location of projectile
  CGSize winSize = [[CCDirector sharedDirector] winSize];
  CCSprite *projectile = [CCSprite spriteWithFile:@"Projectile.png"
    rect:CGRectMake(0, 0, 20, 20)];
  projectile.position = ccp(20, winSize.height/2);

  // Determine offset of location to projectile
  int offX = location.x - projectile.position.x;
  int offY = location.y - projectile.position.y;

  // Bail out if we are shooting down or backwards
  if (offX <= 0) return;

  // Ok to add now - we've double checked position
  [self addChild:projectile];

  // Determine where we wish to shoot the projectile to
  int realX = winSize.width + (projectile.contentSize.width/2);
  float ratio = (float) offY / (float) offX;
  int realY = (realX * ratio) + projectile.position.y;
  CGPoint realDest = ccp(realX, realY);

  // Determine the length of how far we're shooting
  int offRealX = realX - projectile.position.x;
  int offRealY = realY - projectile.position.y;
  float length = sqrtf((offRealX*offRealX)+(offRealY*offRealY));
  float velocity = 480/1; // 480pixels/1sec
  float realMoveDuration = length/velocity;

  // Move projectile to actual endpoint
  [projectile runAction:[CCSequence actions:
    [CCMoveTo actionWithDuration:realMoveDuration position:realDest],
    [CCCallFuncN actionWithTarget:self selector:@selector(spriteMoveFinished:)],
    nil]];

}
```

In the first portion, we choose one of the touches to work with, get the location in the current view, then call convertToGL to convert the coordinates to our current layout. This is important to do since we are in landscape mode.

Next we load up the projectile sprite and set the initial position as usual. We then determine where we wish to move the projectile to, using the vector between the player and the touch as a guide, according to the algorithm described previously.

Note that the algorithm isn't ideal. We're forcing the bullet to keep moving until it reaches the offscreen X position – even if we would have gone offscreen in the Y position first! There are various ways to address this including checking for the shortest length to go offscreen, having our game logic callback check for offscreen projectiles and removing rather than using the callback method, etc. but for this beginner tutorial we'll keep it as-is.

The last thing we have to do is determine the duration for the movement. We want the bullet to be shot at a constant rate despite the direction of the shot, so again we have to do a little math. We can figure out how far we're moving by using the Pythagorean Theorem. Remember from geometry, that is the rule that says the length of the hypotenuse of a triangle is equal to the square root of the sum of the squares of the two sides.

Once we have the distance, we just divide that by the velocity in order to get the duration. This is because velocity = distance over time, or in other words time = distance over velocity.

The rest is setting the actions just like we did for the targets. Compile and run, and now your ninja should be able to fire away at the oncoming hordes!

Collision Detection

So now we have shurikens flying everywhere – but what our ninja really wants to do is to lay some smack down. So let's add in some code to detect when our projectiles intersect our targets.

There are various ways to solve this with Cocos2D, including using one of the included physics libraries: Box2D or Chipmunk. However to keep things simple, we are going to implement simple collision detection ourselves.

To do this, we first need to keep better track of the targets and projectiles currently in the scene. Add the following to your HelloWorldLayer class declaration:

```
NSMutableArray *_targets;
NSMutableArray *_projectiles;
```

And initialize the arrays in your init method:

```
_targets = [[NSMutableArray alloc] init];
_projectiles = [[NSMutableArray alloc] init];
```

And while we're thinking of it, clean up the memory in your dealloc method:

```
[_targets release];
_targets = nil;
[_projectiles release];
_projectiles = nil;
```

Now, modify your addTarget method to add the new target to the targets array and set a tag for future use:

```
target.tag = 1;
[_targets addObject:target];
```

And modify your ccTouchesEnded method to add the new projectile to the projectiles array and set a tag for future use:

```
projectile.tag = 2;
[_projectiles addObject:projectile];
```

Finally, modify your spriteMoveFinished method to remove the sprite from the appropriate array based on the tag:

```
if (sprite.tag == 1) { // target
  [_targets removeObject:sprite];
} else if (sprite.tag == 2) { // projectile
  [_projectiles removeObject:sprite];
}
```

Compile and run the project to make sure everything is still working OK. There should be no noticeable difference at this point, but now we have the bookkeeping we need to implement some collision detection.

Now add the following method to HelloWorldLayer:

```
- (void)update:(ccTime)dt {

  NSMutableArray *projectilesToDelete = [[NSMutableArray alloc] init];
  for (CCSprite *projectile in _projectiles) {
    CGRect projectileRect = CGRectMake(
      projectile.position.x - (projectile.contentSize.width/2),
      projectile.position.y - (projectile.contentSize.height/2),
      projectile.contentSize.width,
      projectile.contentSize.height);

    NSMutableArray *targetsToDelete = [[NSMutableArray alloc] init];
    for (CCSprite *target in _targets) {
      CGRect targetRect = CGRectMake(
        target.position.x - (target.contentSize.width/2),
        target.position.y - (target.contentSize.height/2),
        target.contentSize.width,
        target.contentSize.height);

      if (CGRectIntersectsRect(projectileRect, targetRect)) {
        [targetsToDelete addObject:target];
      }
    }

    for (CCSprite *target in targetsToDelete) {
      [_targets removeObject:target];
      [self removeChild:target cleanup:YES];
    }

    if (targetsToDelete.count > 0) {
      [projectilesToDelete addObject:projectile];
    }
    [targetsToDelete release];
  }

  for (CCSprite *projectile in projectilesToDelete) {
    [_projectiles removeObject:projectile];
    [self removeChild:projectile cleanup:YES];
  }
  [projectilesToDelete release];
}
```

The above should be pretty clear. We just iterate through our projectiles and targets, creating rectangles corresponding to their bounding boxes, and use CGRectIntersectsRect to check for intersections. If any are found, we remove them from the scene and from the arrays. Note that we have to add the objects to a "toDelete" array because you can't remove an object from an array while you are iterating through it. Again, there are more optimal ways to implement this kind of thing, but I am going for the simple approach.

You just need one more thing before you're ready to roll – schedule this method to run as often as possible by adding the following line to your init method:

```
[self schedule:@selector(update:)];
```

Give it a compile and run, and now when your projectiles intersect targets they should disappear!

Finishing Touches

We're pretty close to having a workable (but extremely simple) game now. We just need to add some sound effects and music (since what kind of game doesn't have sound!) and some simple game logic.

If you've been following my blog series on audio programming for the iPhone, you'll be extremely pleased to hear how simple the Cocos2D developers have made it to play basic sound effects in your game.

First, drag some background music and a shooting sound effect into your resources folder. Feel free to use the cool background music I made or my awesome pew-pew sound effect, or make your own.

Then, add the following import to the top of your HelloWorldLayer.m:

```
#import "SimpleAudioEngine.h"
```

In your init method, start up the background music as follows:

```
[[SimpleAudioEngine sharedEngine] playBackgroundMusic:@"background-music-aac.caf"];
```

And in your ccTouchesEnded method play the sound effect as follows:

```
[[SimpleAudioEngine sharedEngine] playEffect:@"pew-pew-lei.caf"];
```

Now, let's create a new scene that will serve as our "You Win" or "You Lose" indicator. Click on the Classes folder and go to File\New File, and choose Objective-C class, and make sure subclass of NSObject is selected. Click Next, then type in GameOverScene as the filename, and make sure "Also create GameOverScene.h" is checked.

Then replace GameOverScene.h with the following code:

```
#import "cocos2d.h"

@interface GameOverLayer : CCLayerColor {
  CCLabelTTF *_label;
}
@property (nonatomic, retain) CCLabelTTF *label;
@end

@interface GameOverScene : CCScene {
  GameOverLayer *_layer;
}
@property (nonatomic, retain) GameOverLayer *layer;
@end
```

Then replace GameOverScene.m with the following code:

```
#import "GameOverScene.h"
#import "HelloWorldLayer.h"
```

```
@implementation GameOverScene
@synthesize layer = _layer;

- (id)init {

  if ((self = [super init])) {
    self.layer = [GameOverLayer node];
    [self addChild:_layer];
  }
  return self;
}

- (void)dealloc {
  [_layer release];
  _layer = nil;
  [super dealloc];
}

@end

@implementation GameOverLayer
@synthesize label = _label;

-(id) init
{
  if( (self=[super initWithColor:ccc4(255,255,255,255)] )) {

    CGSize winSize = [[CCDirector sharedDirector] winSize];
    self.label = [CCLabelTTF labelWithString:@"" fontName:@"Arial" fontSize:32];
    _label.color = ccc3(0,0,0);
    _label.position = ccp(winSize.width/2, winSize.height/2);
    [self addChild:_label];

    [self runAction:[CCSequence actions:
      [CCDelayTime actionWithDuration:3],
      [CCCallFunc actionWithTarget:self selector:@selector(gameOverDone)],
      nil]];

  }
  return self;
}

- (void)gameOverDone {

  [[CCDirector sharedDirector] replaceScene:[HelloWorldLayer scene]];

}

- (void)dealloc {
  [_label release];
  _label = nil;
  [super dealloc];
}

@end
```

Note that there are two different objects here: a scene and a layer. The scene can contain any number of layers, however in this example it just has one. The layer just puts a label in the middle of the screen, and schedules a transition to occur 3 seconds in the future back to the Hello World scene.

Finally, let's add some extremely basic game logic. First, let's keep track of the projectiles the player has destroyed. Add a member variable to your HelloWorldLayer class in HelloWorldLayer.h as follows:

```
int _projectilesDestroyed;
```

Inside HelloWorldLayer.m, add an import for the GameOverScene class:

```
#import "GameOverScene.h"
```

Increment the count and check for the win condition in your update method inside the targetsToDelete loop right after removeChild:target:

```
_projectilesDestroyed++;
if (_projectilesDestroyed > 30) {
  GameOverScene *gameOverScene = [GameOverScene node];
  _projectilesDestroyed = 0;
  [gameOverScene.layer.label setString:@"You Win!"];
  [[CCDirector sharedDirector] replaceScene:gameOverScene];
}
```

And finally let's make it so that if even one target gets by, you lose. Modify the spriteMoveFinished method by adding the following code *inside the tag == 1 case* right after removeChild:sprite:

```
GameOverScene *gameOverScene = [GameOverScene node];
[gameOverScene.layer.label setString:@"You Lose :["];
[[CCDirector sharedDirector] replaceScene:gameOverScene];
```

Go ahead and give it a compile and run, and you should now have win and lose conditions and see a game over scene when appropriate!

Gimme The Code!

And that's a wrap! Here's the full code for the simple Cocos2D iPhone game that we developed thus far.

Where To Go From Here?

This project could be a nice basis for playing around some more with Cocos2D by adding some new features into the project. Maybe try adding in a bar chart to show how many more targets you have to destroy before you win (check out the drawPrimitivesTest sample project for examples of how to do that). Maybe add cooler death animations for when the monsters are destroyed (see ActionsTest, EffectsTest, and EffectsAdvancedTest projects for that). Maybe add more sounds, artwork, or gameplay logic just for fun. The sky's the limit!

If you want to keep going with this tutorial series, check out part two, How To Add A Rotating Turret, or part three, Harder Monsters and More Levels!

Also, if you'd like to keep learning more about Cocos2D, check out my tutorials on how to create buttons in Cocos2D, intro to Box2D, or how to create a simple Breakout game.

Feel free to chime in if you know of any better ways to do various things with this project or if there are any problems – like I said this is the first time I've played with Cocos2D so I have a lot left to learn!

Category: iPhone

Tags: audio, cocos2D, game, iPhone, sample code, tutorial

I'd love to hear your thoughts!

144 Comments! Add a comment!

Recent Tutorials

- Yet More Translations – Spanish and Chinese!
- How To Make a Simple Playing Card Game with Multiplayer and Bluetooth, Part 3
- How To Create an App Like Instagram With a Web Service Backend – Part 2/2
- How To Create an App Like Instagram With a Web Service Backend – Part 1/2
- Simple Cocos2D Game Tutorial Translated to Chinese!
- How To Make a Simple Playing Card Game with Multiplayer and Bluetooth, Part 2
- Core Data on iOS 5 Tutorial: How To Work with Relations and Predicates
- Simple iPhone App Tutorial Translated to Chinese!
- How to Parse HTML on iOS
- How To Make a Simple Playing Card Game with Multiplayer and Bluetooth, Part 1

Vote for the Next Tutorial!

Every week, we alternate between Cocoa Touch and Gaming tutorial votes. This week: Cocoa Touch!

- ○ Introduction to Test Flight for Beta Testing
- ○ How to Use OpenCV for Object Detection
- ○ How to use Unit Tests in Xcode
- ○ Introduction to the Model View Controller and Singleton Patterns in iOS
- ○ How to use Automator and ActionScript iOS Developers
- ○ Top 10 Most Useful iOS Libraries to Know and Love

Vote

Last week's winner: How to Create an Isometric Tile Based Game. Coming soon!

Suggest a Tutorial - Past Results

Books and Starter Kits

iOS Tutorial Team

Tope Abayomi

Gustavo Ambrozio

Abdul Azeem

Steve Baranski

Linda Burke

Adam Burkepile

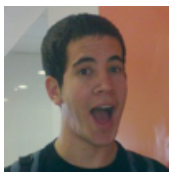
Marcelo Fabri

Matt Galloway

Jacob Gundersen

Ali Hafizji

Matthijs Hollemans



Felipe Laso

Scott McAlister

Jean-Yves Mengant

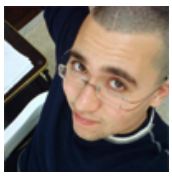
Connor Osborn

Cesare Rocchi

Pablo Ruiz

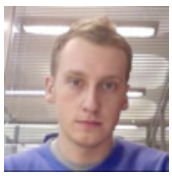
Allen Tan

Marin Todorov



Malek Trabelsi

Jason Van Lint



Krzysztof Zablocki

Forum Moderators

Adam Burkepile

Richard Casey

Adam Eberbach

Marcio Valenzuela

Nick Waynik

Editors

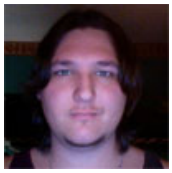
Chris Belanger

Fahim Farook

B.C. Phillips

Ray Wenderlich

Reader's Apps Reviewer



Ryan Poolos

Translation Team

Juan Gonzalez

Xiangping Meng

Di Peng

Sonic Zhao

Search

Search for: Search