Extracted from:

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

Jonathan Penn
and Josh Smith

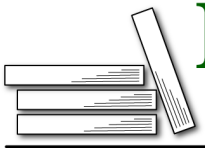
Edited by Rebecca Gulick

Build iOS Games with Sprite Kit

Unleash Your Imagination in Two Dimensions

Jonathan Penn
Josh Smith

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (indexer)
Cathleen Small (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-210-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2014

Following the Finger Around

To move the ship around, we have to update its position property every time a finger comes in contact with the screen. Thankfully, handling touch events in Sprite Kit scenes is the same as elsewhere in iOS. We have all the standard low-level touch event methods.

We'll add this method after the -initWithSize: method to move the ship when a touch begins:

01-SpriteIntro/step03/SpaceRun/RCWMyScene.m

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    CGPoint touchPoint = [touch locationInNode:self];
    SKNode *ship = [self childNodeWithName:@"ship"];
    ship.position = touchPoint;
}
```

In the -touchesBegan:withEvent: method, we grab one of the touches out of the set with the anyObject method. Because our game is meant to be played with a single finger, we're going to let the system pick, just in case more than one touch comes in contact with the screen at the same time.

We then ask the touch to return coordinates in our scene's coordinate space using the -locationInNode: method and passing in the scene as the parameter. Remember that our RCWMyScene class is a subclass of SKScene, which itself is a subclass of SKNode. It's nodes all the way down to the bottom! Each node's children are positioned within that node's local coordinate space, just like UIView objects in normal UIKit. By calling this method with the scene, we are asking the touch object to convert from screen coordinates to scene coordinates so we have the right location to move the ship as the player expects.

Once we have the ship's new coordinates, we're ready to update the position property. But how do we get access to the ship node in this method? Here we're using one of the powerful features of Sprite Kit. We can give nodes names and look them up anywhere in the scene graph. That's what we're doing by calling [self childNodeWithName:@"ship"]. In this case, we're just looking for a direct descendant of this scene with that exact name. You'll learn how to find nodes with more flexible queries later.

Of course, to make this work we have to give the node the name we're looking for. Update the -initWithSize: method to set the name property.

01-SpriteIntro/step03/SpaceRun/RCWMyScene.m

```
NSString *name = @"Spaceship.png";
```

```
SKSpriteNode *ship = [SKSpriteNode spriteNodeWithImageNamed:name];
ship.position = CGPointMake(size.width/2, size.height/2);
ship.size = CGSizeMake(40, 40);
ship.name = @"ship";
[self addChild:ship];
```

Now, when we run the game, tapping anywhere on the screen updates the position property, and the ship jumps under the finger.

But we don't just want the ship to jump when a finger touches. We want the ship to follow the finger on the screen as it moves. Let's do that next.

Making the Ship Glide

As our game is now, we have a mechanical problem with our ship. It only moves when a touch begins on the screen, and we want it to move toward where the finger *drags around* on the screen. Because we get all the standard touch events from iOS, we could copy the same code into -touchesMoved:withEvent: and update the ship's position property there, but there's a simpler way with Sprite Kit.

Let's start with a property that keeps track of the touch that we received until the touch ends. Add this class extension to the top of the RCWMyScene.m file above the @implementation definition:

01-SpriteIntro/step04/SpaceRun/RCWMyScene.m

```
@interface RCWMyScene ()
@property (nonatomic, weak) UITouch *shipTouch;
@end
```

We're declaring the property as weak because we don't want to keep a reference to the object when the system is done with it. UITouch objects live and update themselves for the lifetime of the touch. The touch-handling system releases the objects when the touch is ended. Because our property is weak, it will automatically be set to nil for us.

Now let's set that property in -touchesBegan:withEvent:.

01-SpriteIntro/step04/SpaceRun/RCWMyScene.m

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.shipTouch = [touches anyObject];
}
```

Every time a new touch happens, we'll keep a weak reference to it so we can use it later. Next, we'll update the ship's position every time a frame is drawn by adding this method to the bottom of the RCWMyScene class:

01-SpriteIntro/step04/SpaceRun/RCWMyScene.m

```
- (void)update:(NSTimeInterval)currentTime
```

```
{
    if (self.shipTouch) {
        SKNode *ship = [self childNodeWithName:@"ship"];
        ship.position = [self.shipTouch locationInNode:self];
    }
}
```

The -update: method has special significance on SKScene objects. If Sprite Kit sees this on a scene, it will be called just before every frame is rendered to the screen. This is a great place to update the state of the game, such as making the ship node follow the finger.

In this method, we're checking to see whether the shipTouch property is nil. Remember that because this is a weak property, it will be set to nil for us by the touch-handling system when it releases the touches after they are done.

If the touch is still there, then we find the ship node by name and update its position property like we did before. Except this time, the position will change on every frame, and the ship will keep up with wherever the finger is on the screen.

It's great that our ship can move, but this isn't quite the effect we want. Our game mechanics depend on the ship gliding with a constant speed from where it is now to where the finger is currently on the screen. That makes it more challenging for players so they can't just tap around and cause the ship to jump immediately out of harm's way.

Smoothing Out the Motion

To create a smooth, gliding effect while the ship follows the finger, we'll want to update the ship's position to move closer to the finger over time, rather than jump right to the finger's coordinates. Because the -update: method receives the value of Sprite Kit's clock in the currentTime parameter, we can use that to calculate how far the ship should move by keeping track of the time between frames.

First, we'll add a new property to the class extension of the RCWMyScene object. We'll use this to record the last time we updated the frame.

01-SpriteIntro/step05/SpaceRun/RCWMyScene.m

```
@interface RCWMyScene ()
@property (nonatomic, weak) UITouch *shipTouch;
@property (nonatomic) NSTimeInterval lastUpdateTime;
@end
```

Then, in the -update: method, we'll subtract the value of that property to calculate the time delta since the last frame.

01-SpriteIntro/step05/SpaceRun/RCWMyScene.m

```
- (void)update:(NSTimeInterval)currentTime
{
    if (self.lastUpdateTime == 0) {
        self.lastUpdateTime = currentTime;
    }
    NSTimeInterval timeDelta = currentTime - self.lastUpdateTime;

    if (self.shipTouch) {
        [self moveShipTowardPoint:[self.shipTouch locationInNode:self]
                     byTimeDelta:timeDelta];
    }
    self.lastUpdateTime = currentTime;
}
```

We're checking to see whether the lastUpdateTime property is zero first, because if it is, that means this is the first frame rendered of this scene. We need to initialize this property before we can get meaningful time-delta calculations, but we don't know what to initialize it to until the first time we are called.

Next, we calculate the timeDelta value by subtracting the currentTime parameter from the lastUpdateTime property. Then, if the shipTouch property holds a touch object, we call a new method to move the ship according to the touch point by how much time has passed. We're asking the UITouch object itself to give us the coordinate of the touch within the scene's local coordinate system. After all the work is done, we set the lastUpdateTime property to currentTime so we are ready to calculate the time difference of the next frame.

Let's write the -moveShipTowardPoint:byTimeDelta: method to nudge the ship by the appropriate amount for this frame.

01-SpriteIntro/step05/SpaceRun/RCWMyScene.m

```
- (void)moveShipTowardPoint:(CGPoint)point byTimeDelta:(NSTimeInterval)timeDelta
{
    CGFloat shipSpeed = 130; // points per second
    SKNode *ship = [self childNodeWithName:@"ship"];
    CGFloat distanceLeft = sqrt(pow(ship.position.x - point.x, 2) +
                                pow(ship.position.y - point.y, 2));
    if (distanceLeft > 4) {
        CGFloat distanceToTravel = timeDelta * shipSpeed;
        CGFloat angle = atan2(point.y - ship.position.y,
                              point.x - ship.position.x);
        CGFloat yOffset = distanceToTravel * sin(angle);
        CGFloat xOffset = distanceToTravel * cos(angle);
        ship.position = CGPointMake(ship.position.x + xOffset,
                                    ship.position.y + yOffset);
    }
}
```

Yikes! If you'd like to take a moment to write apology notes to your high school trigonometry teacher, go right ahead. We did, too. Game development is a great way to refresh the mind on all the math we thought wouldn't be necessary in real life. Don't worry, we'll break down this code together. Figure 6, *Calculating the distance to travel this frame*, on page 6 provides a figure to help visualize what's happening:

First off, we are setting a `shipSpeed` variable to keep track of how many points per second the ship should travel. We find the ship node and calculate `distanceLeft` using the Pythagorean theorem with the ship's current location and final destination.[4]

Before we actually move the ship, we're checking to see whether this `distanceLeft` variable is greater than four points. If not, then we don't want to move the ship anymore. We're close enough. If we kept trying to move the ship anyway, then it's possible that the ship would jitter around the touch point because of the imprecision of the floating-point calculations. Four points is far enough away that any rounding errors won't wiggle the ship around the destination point and close enough that the player will have the impression the ship reached the finger.

Assuming we're not close enough, then we calculate the `distanceToTravel` variable by multiplying the `timeDelta` by the `shipSpeed`. This is how far we should move for just this frame. We have to convert that distance back into `x`- and `y`-coordinates, so we use the `atan2()` function and some more basic trigonometry to set the ship node's `position` property.

Now run the game, and the ship will glide at a nice, constant rate to wherever your finger is on the screen. This is an important game mechanic because players will have to think about how to maneuver around obstacles as they approach. No cheating!

And that's it for our whirlwind Sprite Kit introduction! You've learned a little bit about how Sprite Kit draws things to the screen, you've learned how to track touches and update the ship's position over time, and you've learned about the frame update loop along the way.

This is a great start, but weren't we supposed to be able to shoot and dodge obstacles? Yup, and to do that we'll have to tackle the next topic, Sprite Kit actions!

4. http://en.wikipedia.org/wiki/Pythagorean_theorem

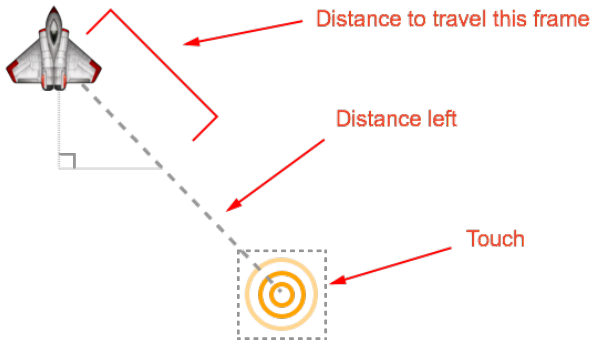


Figure 6—Calculating the distance to travel this frame