3
Interacting with Our Game

In this chapter, we will discuss the ways in which we can get input from the player. As there are many different ways to handle user input, we will see the advantages and disadvantages of each of them. Some of the ways to interact with the game are as follows:

- Touching the screen (taps)
- Gesture recognizers
- Accelerometer and gyroscope
- Game controller events

Handling touches

When the user touches the iOS device screen, our current running scene will receive a call to one of the following methods:

- `-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`
- `-(void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *) event`

Each method gets an `NSSet` class of a `UITouch` object (which holds the coordinates of our touch), and the event holds information about the window and view in which the touch was triggered as well as information about the touch. Let's see what each of them does:

- `touchesBegan:`: This method gets triggered each time the user touches the screen, unrelated to whether this touch continues as a long press or not.
- `touchesMoved:`: This method provides you with events that occurred if the user moved a finger while on screen, and provides new touches.
- `touchesEnded:`: This method gets called when the user takes their finger off the screen. This can be useful, for example, if you want some animation to play after the user removes their finger, or to re-enable physics calculations on the node.
- `touchesCancelled:`: This method is rarely needed, but you should implement it anyway. It is called when some event cancels the touch, for example, an on-screen alert, phone call, notifications (such as a push notification or calendar notification), and others. You will likely want to trigger the same code as in `touchesEnded:`.

Sprite Kit offers a few helper methods to assist us with touches and detecting nodes in which touches happened. We will list some of them here. These are all methods of SKScene:

- `[touch locationInNode:(SKNode*)node`: This is an example of one of the methods to convert location from UIView coordinates to coordinates in SKNode. This is useful since touch objects in those methods have UIView coordinates and we operate with Sprite Kit coordinates.
- `[self nodeAtPoint:(CGPoint)p]`: This method returns the node in the scene's hierarchy that intersects the provided point. This can be useful to detect touches on certain nodes without tedious calculations.

Now that we have learned the basics, let's implement the basic functionality of touch handling. We will move the character sprite by dragging it around. We will need to perform a few steps to accomplish this, which are as follows:

1. Add `@property (assign) BOOL selected;` to your player class in `ERGPlayer.h`—we will use this to handle the state of the player sprite; whether we should drag it when the user drags a finger on the screen or disregard such touches.

2. Add the following code to the beginning of the `touchesBegan:` method in ERGScene:

```
    UITouch *touch = [touches anyObject];
    SKSpriteNode *touchedNode = (SKSpriteNode *)[self
nodeAtPoint:[touch locationInNode:self]];

    if (touchedNode.name == playerName) {
        ERGPlayer *player = (ERGPlayer *) touchedNode;
        player.selected = YES;
        return;
    }
```

On the first line, we get the `touch object` from the `touches` set and then try to find if it intersects any node. To do that, we call the `nodeAtPoint:` method and provide the location of the touch by converting `touch` to the Sprite Kit coordinates in the `locationInNode:` method. We want to ignore taps on the background itself, so this is why we check to make sure the player was tapped.

If we tap the player, we set `selected` to `YES` so that the `touchesMoved:` method knows that we are dragging the character sprite. After that, we call `return` to exit the method, as after this return statement, we have other code that we don't want to trigger.

The next thing that we need to handle is the `touchesMoved:` method. We haven't used that before, so type it as follows:

```
-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];

    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:playerName];
    if (player.selected) {

    player.position = [touch locationInNode:self];
    }
}
```

This method (`touchesMoved:`) checks if the player is actually selected, and if it is, the coordinates are changed to the touch location. We only want to drag the character sprite if touches began on it.

And the thing that we need to do last is remove the selected property after the touch has ended:

```
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:playerName];
    if (player.selected) {
        player.selected = NO;
    }
}
```

Now, if you build and run the project, you will be able to drag the character sprite around. Another option to consider is to move the score label around the screen to the position that you would like.

You might have noticed that we are checking the view hierarchy each time we want to grab a pointer to the player. This is not the optimal way to do it. Another option to consider is using a player as a property of the scene. This looks like a fast and easy way to do it, but consider that you might have a few objects on screen, or even dozens of them that need interaction. Having 20 properties for this reason does not seem feasible.

You should also know that traversing a node tree is not a very fast or efficient operation. Imagine that you have 1,000 nodes. If you need to find a node by name and it is not there, the code will traverse each of these nodes as it needs to make sure that there is none with your name.

This traversal might happen a few times during just one frame calculation, and since this happens many times per second, you might get a very real slowdown from this. You might want to cache certain nodes in weak pointers in case you need them later. Usually, you might want to avoid many node tree traversals.

Using gesture recognizers

Gesture recognizers allow us to not bother with the low-level code that was explained earlier in this chapter. Without gesture recognizers, certain things might be extremely hard to implement, such as pinching in and out or rotating. Thankfully, Apple has handled all of this for us.

We might want to increase and decrease the speed of scrolling for testing reasons, so we will implement this feature.

The first thing that comes to mind is adding a gesture recognizer to our scene, pointing it to some method, and be done with it. But unfortunately, this won't work—SKNodes and SKScenes do not support adding gesture recognizers. But there is a way to make this work.

SKScene has a handy method, `– (void)didMoveToView:(SKView *)view`, which we can use, and it gets called each time a scene is about to be attached to some view. When we run our game, this happens right after the scene creation, and thus it is a useful place to set up our gesture recognizers. The following is the code for this method:

```
- (void)didMoveToView:(SKView *)view
{
    UISwipeGestureRecognizer *swiper = [[UISwipeGestureRecognizer
alloc] initWithTarget:self action:@selector(handleSwipeRight:)];
    swiper.direction = UISwipeGestureRecognizerDirectionRight;
    [view addGestureRecognizer:swiper];

    UISwipeGestureRecognizer *swiperTwo = [[UISwipeGestureRecognizer
alloc] initWithTarget:self action:@selector(handleSwipeLeft:)];
    swiperTwo.direction = UISwipeGestureRecognizerDirectionLeft;
    [view addGestureRecognizer:swiperTwo];
}
```

We create two gesture recognizers, set the methods to be called, and specifically to `UISwipeGestureRecognizer`, set the swipe direction. Thus, if we swipe to the right, one method is called, and if we swipe to the left, another one is triggered. These methods are fairly straightforward, as they only increase or decrease speed:

```
- (void) handleSwipeRight:(UIGestureRecognizer *)recognizer
{
    if (recognizer.state == UIGestureRecognizerStateRecognized) {
        backgroundMoveSpeed += 50;
    }
}

- (void) handleSwipeLeft:(UIGestureRecognizer *)recognizer
{
    if (recognizer.state == UIGestureRecognizerStateRecognized &&
backgroundMoveSpeed > 50) {
        backgroundMoveSpeed -= 50;
    }
}
```

The interesting part here is a second check for background speed in the `handleSwipeLeft:` method. We don't want to go into negative or zero speed, since our background generator works only with positive scrolling (from right to left), and if we have negative scrolling, the background will end.

Another thing that we need to remember is to remove the gesture recognizer once the scene gets removed from the view, as we might get another scene in the same view that doesn't know how to handle these methods. Thus, your application will crash at this point. To prevent this, add this method:

```
-(void)willMoveFromView:(SKView *)view
{
    for (UIGestureRecognizer *recognizer in view.gestureRecognizers) {
        [view removeGestureRecognizer:recognizer];
    }
}
```

This gets called when our scene is about to be removed from the view. It iterates over the gesture recognizers in the view, removing each of them.

Accelerometer

Accelerometers and gyroscopes in modern devices are probably the things that have contributed to much of the popularity of iOS devices. Many people still play Doodle Jump, a game where you control a character that jumps on platforms, and is controlled by tilting your device.

Accelerometer controls have many advantages, which are as follows:

- You don't clutter the screen with game controls
- You don't need to cover the (already small) screen space with fingers while playing your game
- The accelerometer controls can imitate real-life interactions—such as a car steering wheel
- With new hardware (gyroscope) available on the latest devices (starting with iPhone 4), you get precise data about the device's position, which leads to smooth controls

Some games benefit greatly from accelerometer support, and we are going to implement this in our game.

Accelerometer data is available only on devices. Thus, if you test it on the simulator, this code won't do anything, as the accelerometer will return 0 all the time. Consider getting an Apple developer license if you have not got one already, as it offers the option of testing on a real device. You can get it at `https://developer.apple.com/programs/ios/`. It is especially useful for developing games, as the frame rate in Sprite Kit in the simulator is far from the one that you get on real devices. The simulator uses a much faster CPU and much faster memory, but the emulated rendering pipeline is much slower, and you can get very low fps in most trivial situations. Don't trust the frame rate in the simulator!

The accelerometer is handled by the Core Motion library, so you will need to include that in your project. To do that, simply add `@import CoreMotion;`, where all imports are at the top of the `ERGMyScene.h` file.

`@import` is the new way to import frameworks into your code, and it was introduced with iOS 7 and Xcode 5. Earlier, you had to add a framework to your project and use `#import` everywhere you needed it. Now you can use `@import` and forget about adding a framework in the project's settings. The only drawback to this is that you can only use the Apple frameworks this way. Let's hope it will change to all libraries some day.

The next thing that you need to do is add the Core Motion manager as a property to your scene.

To do that, add the following line to `ERGMyScene.h`:

```
@property (strong, nonatomic) CMMotionManager *manager;
```

The manager handles all accelerometer data, and we need it to use the accelerometer's input. In order to start getting accelerometer data, we need to call a few more methods.

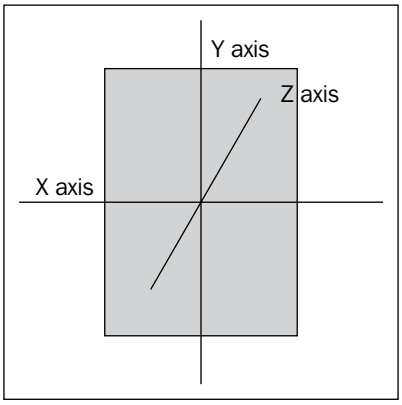
In the `initWithSize:` method of `ERGMyScene.m`, add the following lines:

```
self.manager = [[CMMotionManager alloc] init];
self.manager.accelerometerUpdateInterval = 0.1;
[self.manager startAccelerometerUpdates];
```

In the first line, we instantiate the manager, and in the second one, we set the update interval—how often the values in the motion manager will be updated. On the third line, we tell the manager to start updating, as it won't do anything before that.

Using the accelerometer increases the power use of the device. Heavy use will drain the battery faster, thus making your game unlikely to be played further. Only use the accelerometer when you actually need to, and stop polling it if something stops the application or if you don't need to parse the accelerometer data at this time. You can use `[self.manager stopAccelerometerUpdates];` to stop the accelerometer.

Accelerometer data can be read from `manager.accelerometerData`, and it consists of three fields—X, Y, and Z axis angle. Each of them represents a value from -1 to 1.



Accelerometer data axes

As you can see in the previous diagram, there are three axes, and rotating the device around each of them changes only that axis' values. You can see this if you add the following line to the `update:` method of `ERGScene`:

```
NSLog(@"%@", self.manager.accelerometerData);
```

In order to see what is going on, run the application and start rotating the device along a different axis, and you will see the value of that go from -1 to 1. We will use the accelerometer data to move the character sprite. Remember that the accelerometer does not work on the simulator.

How can we do that? We may read the accelerometer data and set the sprite position to certain values based on that. But how do we set these positions? We need some baseline value so that our character stands on the ground.

As you rotate the device, you can see that values change very fast, and different device positions yield different results. For every person, that baseline will be different, and we need to acknowledge that. To compensate that baseline, we will read the starting value on the launch of an application and base our movement on it.

For our landscape mode game, reading the X axis' coordinates will be most useful. To get it, we poll `self.manager.accelerometerData.acceleration.x`.

In order to get the data that we need, we have to save that baseline value. This is where it gets interesting. Accelerometer data is not available straight away when you ask it to start updating. It takes some time to poll hardware. First, add `@property (assign) float baseline;` to `ERGMyScene.h`—we will store the offset for accelerometer data here.

In the `init` method, right after the `startAccelerometerUpdates` method, add the `performSelector` call to execute the method that sets the baseline:

```
[self performSelector:@selector(adjustBaseline) withObject:nil
afterDelay:0.1];
```

After this, we need to add the `adjustBaseline` method:

```
- (void) adjustBaseline
{
    self.baseline = self.manager.accelerometerData.acceleration.x;
}
```

We will also need some value to be multiplied with the accelerometer data, and having it as a magic number is never good, so let's add a new line to the `Common.h` file:

```
static NSInteger accelerometerMultiplier = 15;
```

Add the following code to the bottom of your update method in `ERGMyScene.m`:

```
ERGPlayer *player = (ERGPlayer *) [self childNodeWithName:playerName];
    player.position = CGPointMake(player.position.x, player.position.y
- (self.manager.accelerometerData.acceleration.x - self.baseline) *
accelerometerMultiplier);

    if (player.position.y < 68) {
        player.position = CGPointMake(player.position.x, 68);
    }
    if (player.position.y > 252) {
        player.position = CGPointMake(player.position.x, 252);
    }
```

First, we get the pointer to our player sprite. Then, we calculate the new position by multiplying the accelerometer data and the out multiplier to get the sprite position on the Y axis. As we don't want to have our character fly off screen, we add a check; if it goes below or over a certain point, its vertical position is reset to the minimum value. Build and run the project to see if everything works.

Physics engine

The next big topic that we will discover is physics in our application. Physics-based games are very popular on the App Store. Angry Birds, Tiny Wings, and Cut the Rope are all physics-based. They offer lifelike interactions that are fun and appealing.

Another reason why we might want to have a physics engine is that it offers a lot of functionality "for free". You no longer need to calculate different complicated collisions, forces, and all things that may affect your nodes. Most of these things are handled for you by a physics engine.

Physics simulation basics

If you want your node to participate in physics simulation, you have to add a physics body to it. There are many available methods to generate physics bodies, and they are as follows:

- `BodyWithCircleOfRadius`: This method creates a circular physics body. It is the fastest physics body, and if you have many objects that need to be simulated, consider setting their physics body to a circle.
- `BodyWithRectangleOfSize`: This is the same as the previous one, but with a rectangle. Usually, passing a frame of a node is sufficient enough for most games.
- `BodyWithPolygonFromPath`: This creates a physics body from CGPath. This is useful if you have a very complex sprite and you want it to be simulated as real as possible.
- `BodyWithEdgeFromPoint:toPoint`: This is useful to create edges such as ground level.

There are more methods available; you can check them in the `SKPhysicsBody` class reference.

In order to simulate physics on many bodies in real time, there are many different optimizations and simplifications. Let's cover some of them:

- All movements, just like in real life, are handled by impulses and forces. The impulse is applied instantaneously; for example, if a moving ball collides with a standing ball, the standing ball gets an impulse in one moment. Force is a continuous effect that moves the body in some direction. For example, launching a rocket is slow and steady as force gets applied that pushes it up. All forces and impulses add to each other; thus, you can have very complex movements with little hassle.
- All bodies are considered rigid—they can't deform as a result of physics simulation. Of course, you can write your system over existing physics simulation to do that for you, but it is not provided out of the box.

Each physics body has many properties, some of which are as follows:

- **Mass**: This is the mass of the body in kilograms. The more massive the body is, the harder it is to move by impulses or forces.
- **Density**: This indicates how much mass the body has in a fixed amount of volume. The denser the body is, the more mass it has for the same volume.
- **Dynamic**: This is the property that allows you to apply impulses or forces to the body. Sometimes it is not needed; for example, ground will always be static and not dynamic. You might also pick dynamic if you want to move your body yourself, without the physics engine touching it.
- **Restitution**: This is the property that determines how "jumpy" or "bouncy" the body is. The higher this is, the higher the knockback is. This ranges from 0 to 1, but you can set this higher than 1, and in this case, the body will accelerate each time after the collision, which may eventually lead to a crash.
- **usesPreciseCollisionDetection**: This is used to handle very fast moving objects. There are certain situations in which collision might not get detected, for example, if in one frame, the body is in front of another body, and in the next frame it is completely behind it without touching it; in this case, the collision will never be detected, and you don't want this. This method is very CPU-intensive, so use it only if absolutely necessary.
- **affectedByGravity**: This property is set to `NO` if you don't want some objects such as balloons to be affected by gravity.
- **categoryBitMask**: This is the bitmask that identifies the body. You might choose different bitmasks for different objects.

- **collisionBitMask**: This is the bitmask that identifies what bodies this body collides with. You might want some objects to collide with others, and other objects to pass through others. You will need to set the same bitmasks for objects that you want to collide.
- **contactBitMask**: This is the bitmask that identifies the body for handling collisions manually. You might want to have special effects when things collide; that's why we have this. When bodies have the same bitmask, a special method will get called for you to handle such collisions manually.

Implementing the physics engine

We might as well start implementing physics in our engine in order to see what the physics engine gives us. First, we need to set the physics body to our player sprite. In order to do that, we need more constants in `Common.h`:

```
static NSInteger playerMass = 80;
static NSInteger playerCollisionBitmask = 1;
```

Next, change your `init` method in `ERGPlayer.m` to look as follows:

```
-(instancetype)init
{
    self = [super initWithImageNamed:@"character.png"];
    {
        self.name = playerName;
        self.zPosition = 10;
        self.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:CGSi
zeMake(self.size.width, self.size.height)];
        self.physicsBody.dynamic = YES;
        self.physicsBody.mass = playerMass;
        self.physicsBody.collisionBitMask = playerCollisionBitmask;
        self.physicsBody.allowsRotation = NO;
    }
    return self;
}
```

After creating the sprite and setting the name and z order, we start with the physics body. We create a physics body with the same size as that of our sprite—it is accurate enough for our game. We set it as dynamic, as we want to use forces and impulses on our character. Next up, we set the mass, collision bitmask, and disallow rotation, as we expect our character sprite to never rotate.

This looks sufficient for our character sprite. What can we do with backgrounds?

Obviously, we need the player to collide with the top and bottom of the screen. This is when other methods come in handy. Add this code right before the return statement in the `generateNewBackground` method in `ERGBackground.m`:

```
    background.physicsBody = [SKPhysicsBody bodyWithEdgeFromPoint:CGPo
intMake(0, 30) toPoint:CGPointMake(background.size.width, 30)];
    background.physicsBody.collisionBitMask = playerCollisionBitmask;
```

We set the bitmask to be the same as we want the player to collide with the background. But here comes the problem. Since all physics bodies need to be convex, we can't create the physics body that accommodates both the top and bottom collision surfaces. We might think of adding a second collision body to the same node, but it is not supported.

There is a small hack around this—we will just create a new node, attach it as a child to the background node, and attach the top collision body to it. Add the following code after the previous code and right before the return statement:

```
    SKNode *topCollider = [SKNode node];
    topCollider.position = CGPointMake(0, 0);
    topCollider.physicsBody = [SKPhysicsBody bodyWithEd
geFromPoint:CGPointMake(0, background.size.height - 30)
toPoint:CGPointMake(background.size.width, background.size.height -
30)];
    topCollider.physicsBody.collisionBitMask = 1;
    [background addChild:topCollider];
```

Here, we create a new physics body with `EdgeFromPoint`, from the left edge to the right edge. This way, we effectively have two colliding bodies on one background.

As we now want to have physics-based controls, you may remove all the code that handles movement. These tapping and actions methods are as follows:

- `touchesBegan:`
- `touchesMoved:`
- `touchesEnded:`
- `swipeLeft`
- `swipeRight`

Or, you may just follow us. Change the `didMoveToView` method to the following:

```
- (void) didMoveToView:(SKView *)view
{
    UILongPressGestureRecognizer *tapper =
[[UILongPressGestureRecognizer alloc] initWithTarget:self action:@
selector(tappedScreen:)];
    tapper.minimumPressDuration = 0.1;
    [view addGestureRecognizer:tapper];
}
```

Here, we initialize and use the new gesture recognizer that will set the player's state to `accelerating`, meaning that the player is moving up now. To handle this, add a new property to `ERGPlayer.h`—`@property (assign) BOOL accelerating;`.

After this, add the following code to `ERGMyScene.m`:

```
- (void) tappedScreen:(UITapGestureRecognizer *)recognizer
{
    ERGPlayer *player = (ERGPlayer *)[self
childNodeWithName:@"player"];
    if (recognizer.state == UIGestureRecognizerStateBegan) {
        player.accelerating = YES;
    }

    if (recognizer.state == UIGestureRecognizerStateEnded) {
        player.accelerating = NO;
    }
}
```

As said before, we need this method to set an accelerated property on the player—if it is active, we will apply a certain force on each frame of the player. To do this, add the following to your update method:

```
    [self enumerateChildNodesWithName:@"player" usingBlock:^(SKNode
*node, BOOL *stop) {
        ERGPlayer *player = (ERGPlayer *)node;
        if (player.accelerating) {
            [player.physicsBody applyForce:CGVectorMake(0,
playerJumpForce * timeSinceLast)];
        }
    }];
```

This code uses the `playerJumpForce` variable, which sets it in `Common.h`, as well as the gravity vector:

```
static NSInteger playerJumpForce = 8000000;
static NSInteger globalGravity = -4.8;
```

We set gravity to custom as the real world's gravity is too high, and this strips away the fun and precise controls of our game.

To affect gravity, add the following statement in `ERGMyScene.m` in the `initWithSize:` method:

```
self.physicsWorld.gravity = CGVectorMake(0, globalGravity);
```

In order to make everything work, we need to comment out or delete the old methods that are in the way. Comment out or delete the following methods: `touchesBegan`, `touchesMoved`, `touchesEnded`, and `touchesCancelled`. If you still have other gesture recognizers such as the long press gesture recognizer in the `didMoveToView:` method, remove them too.

After these changes, we have a really precise and fun way to control our character. We can control the jump of the player sprite just by tapping on the screen for a certain amount of time. This mechanism is used in many endless runners, and is both fun and addictive. You have to not only react fast but also predict what can happen, since applying force to the character does not change your position instantaneously as your movement carries some momentum, so you have to plan accordingly.

Summary

In this chapter, we have learned how to use the touch controls and gesture recognizers to control characters on the screen. We have found out how to move sprites with touch and learned the basics of the physics engine in Sprite Kit. Integrating the physics engine in your game is a great way to handle collision detection and add many interesting behaviors without bloating your code.

In the next chapter, we will learn about animations, how to implement them, and related details, such as textures and texture atlases.