Contents

1 User Interaction and Collision Detection

In this chapter you will incorporate User Interaction into the object catching game. The first step will be implementing a drag and drop mechanism that lets the user move the pot in order to catch objects. To detect if the player has caught or missed an object we will implement basic collision detection - note that you will later learn how to use the Cocos2D physics engine that provides collision detection out of the box. Whether you want to implement your own collision detection or use the physics engine will depend a lot on the type of game you are developing and we will discuss the advantages of both approaches throughout this book.

When we've implemented the first control scheme we will add a second option for players - controlling the game with the accelerometer of the device, another common way to interact with mobile games.

As a byproduct of implementing these features we will work with translating positions and sizes between different node spaces and the world space, so we will be discussing that important concept throughout this chapter as well.

1.1 Add the pot to the game

The goal of our game will be to move a pot across the screen and try to catch all the vegetables while avoiding catching inedible objects. Before we can implement the drag and drop mechanism we need to add the pot assets to our game, we're going to do that in the SpriteBuilder project, open it now.

Typically we use individual CCB Files for each type of object in our game, however for this game we need to make an exception due to the specific way in which order Cocos2D renders our objects in the game.

1.1.1 Working with the z-order

Throughout this book we are working with a 2D engine. In a 2D engine depth can only be represented by certain objects being placed in front or behind of other objects. Cocos2D uses the following criteria to decide which nodes are rendered in front of other nodes:

1. Child nodes are rendered in front of their parent nodes
2. Siblings (nodes with the same parent) are rendered in order of their `zOrder` property; nodes with higher `zOrder` are rendered in front of nodes with a lower one
3. If two siblings have the same `zOrder` the siblings are rendered in reverse order of how they have been added (the latest added node is rendered in front of all other nodes)

As you can see from the description above the `zOrder` only affects how siblings are ordered, Cocos2D currently does not have a global `zOrder`. For our game we want to create the illusion of objects dropping into a pot, we can do that using the Cocos2D Z-order as shown in the figure below.

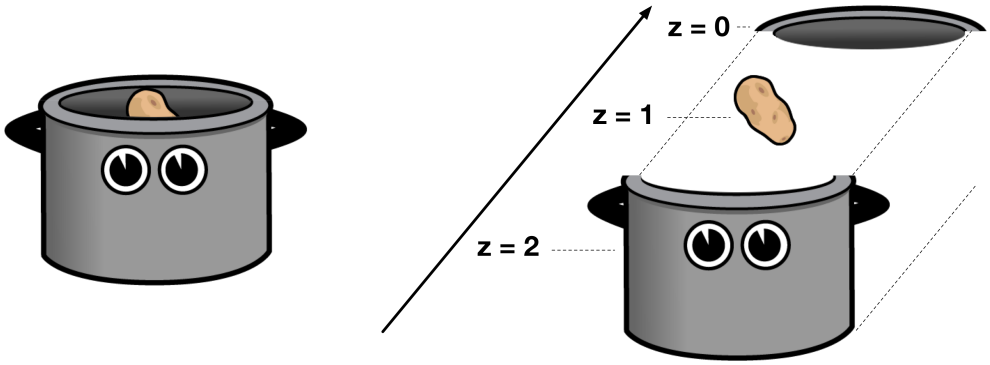


Figure 1.1: Left: Objects on different Layers, Right: How the Z-Order influences on which Layer a node is rendered

For this solution to work all the falling objects and the bottom and top part of our pot need to have the same parent node, otherwise we would not be able to use the Z-Order to place the falling objects between the two parts of the pot.

That is the reason why we are not creating a separate CCB File for the pot object and instead place it inside of `MainScene.ccb`. There would be other ways to work around this issue but adding the pot to the Main Scene is a good solution for this game.

Global Z-order in Cocos2D

While Cocos2D does not have support for global Z-order at the moment, it is being discussed as a potential feature for future releases. Many games run into issues as discussed above due to the lack of this feature. You can follow the discussion on GitHub: https://github.com/cocos2d/cocos2d-swift/issues/662.

1.1.2 Setting up the pot assets

Equipped with everything we need to know about Z-order let's add the pot assets to our Main Scene in SpriteBuilder:

Figure 1.2: Add both pot parts to the main scene

> The assets of both pot parts have exactly the same dimensions. Place both of them at 50% X position and at a Y position of 58 Create code connections for both pot parts, linked to the *Document Root*. Name them `potBottom` and `potTop` respectively. Publish the SpriteBuilder project

Next, move to the Xcode project to set up the code connection variables and implement the touch handling code.

> Open `MainScene.swift` and add instance variables for our code connections at the top of the class:
>
> ```
> weak var potTop: CCSprite!
> weak var potBottom: CCSprite!
> ```

There are three important things to remember about these code connections. Firstly, all code connections should be marked as `weak`. `MainScene` has a reference to the pot sprites but does not *own* them. Instead they are owned by their parent node. For any references that don't mark an ownership, `weak` should be used.

Secondly, we always want to declare code connections as *forcefully unwrapped optionals* as denoted by the bang (!) after the type. Swift requires that all instance variables that aren't optionals are either initialized with a default value or get set to a value in one of the initializers of the class, that way the compiler can guarantee that these variables never contain a `nil` value. Code connections however are set up after the object is initialized (they are guaranteed to be set up when `didLoadFromCCB` is called on the node), so technically these should be optional values. Adding a lot of code for `nil` checking would clutter our classes, that's why we prefer using the bang notation which basically says: *I am confident that this value will never be nil when I am trying to access it*. This is true for code connections as we now that Cocos2D guarantees to have set them up by the time `didLoadFromCCB` is called.

Lastly, be careful not to mark these variables as `private`. Otherwise Cocos2D will not have access to them and won't be able to set up the code connections.

Okay, now we have the basics set up and are ready to dive into the details of implementing a drag and drop mechanism!

1.2 Implement a Drag and Drop mechanism

For the very first project in this book we have already implemented a basic touch mechanism. You should remember that `userInteractionEnabled` is the property that activates/deactivates touch handling for a node and that Cocos2D provides four different callbacks for different state transitions in the lifecycle of a touch. Here's the recap:

touchBegan: called when a touch begins

touchMoved: called when the touch position of a touch changes

touchEnded: called when a touch ends because the user stops touching the screen

touchCancelled: called when a touch is cancelled because user moves touch outside of the touch area of a node

Knowing that, how can we implement a drag and drop control scheme for our game? Dragging and dropping includes three different steps:

1. Pick up object
2. Drag object
3. Drop object

1.2.1 Picking up an Object

In order to pick up an object we need to detect a user's touch and determine if the touch is within the boundaries of our object, if that is the case, we start dragging the object.

First of all, let's turn on user interaction for the `MainScene` class, so that we receive touch events.

Add the required line to the `onEnterTransitionDidFinish` method:

```
override func onEnterTransitionDidFinish() {
  super.onEnterTransitionDidFinish()

  self.userInteractionEnabled = true

  // spawn objects with defined frequency
  schedule("spawnObject", interval: spawnFrequency)
}
```

Next, we need to add the touch handling method. The touch handling method will need to check if the touch is within our pot. If that is the case, the method will need to set a state variable that remembers that we are currently dragging this object. If the user moves a finger across the screen and we are currently in object dragging mode, it is important that the object follows the finger of the user.

Add this implementation to `MainScene.swift`:

```
override func touchBegan(touch: CCTouch, withEvent event:
    ↪ CCTouchEvent) {
  if (CGRectContainsPoint(potBottom.boundingBox(), touch.
      ↪ locationInNode(self))) {
    isDraggingPot = true
    dragTouchOffset = ccpSub(potBottom.anchorPointInPoints,
        ↪ touch.locationInNode(potBottom))
  }
}
```

Let's discuss this implementation briefly. You already have seen the usage of `touch.locationInNode(` in the first chapter of this book, where we briefly discussed touch handling (**??**). This method returns the touch position within a given node. In this specific case we are receiving the touch position within `MainScene`.

Next, we are using a utility function, `CGRectContainsPoint`, to check if this touch is within the pot. Remember, that `potBottom` and `potTop` are placed at exactly the same position, so we can choose either of them for this check. `CGRectContainsPoint` takes a rectangle as its first argument and a point as its second. It returns true if the point is within the rectangle.

If the touch position is inside of the pot, we set our state variable, `isDraggingPot`, to `true`.

There is one last line that we didn't discuss upfront:

```
dragTouchOffset = ccpSub(potBottom.anchorPointInPoints, touch.
    ↪ locationInNode(potBottom))
```

In order to drag an object smoothly we need to remember where we touched that object when starting dragging. Take a look at the following diagram:

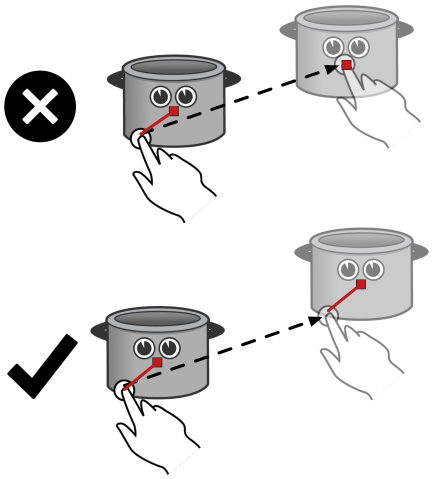


Figure 1.3: *Top Image:* incorrect implementation, object jumps to touch position. *Bottom Image:* correct implementation, touch offset is maintained while dragging the object.

As the user moves the finger, we move the object along. However, the position of the object is not exactly the touch position. Instead it is the touch position *plus* the touch offset determined when we started dragging. We determine that offset by calculating the distance between the anchor point (that's the reference point for positioning a node, typically it's in the center of the node) of the touched object and the exact touch position. No we know why it is important to store the touch offset!

> To wrap up the implementation of `touchBegan` let's add the two instance variables we have referenced: `isDraggingPot` and `dragTouchOffset`. Your list of iVars should now look like this:
>
> ```
> weak var potTop:CCSprite!
> weak var potBottom:CCSprite!
>
> private var fallingObjects = [FallingObject]()
> private let fallingSpeed = 100.0
> private let spawnFrequency = 0.5
> private var isDraggingPot = false
> private var dragTouchOffset = ccp(0,0)
> ```

1.2.2 Moving an Object

Now we'll implement the code that actually moves the pot. That code needs to run whenever a user's finger moves. That means we need to implement the `touchMoved` method.

> Add the `touchMoved` method below the `touchBegan` method:
>
> ```
> override func touchMoved(touch: CCTouch, withEvent event:
>     ↪ CCTouchEvent) {
>   if (!isDraggingPot) {
>     return
> ```

```
    }

    var newPosition = touch.locationInNode(self)
    // apply touch offset
    newPosition = ccpAdd(newPosition, dragTouchOffset);
    // ensure constant y position
    newPosition = ccp(newPosition.x, potBottom.positionInPoints.
        ↪ y);
    // apply new position to both pot parts
    potBottom.positionInPoints = newPosition;
    potTop.positionInPoints = newPosition;
}
```

In the first line we check if we are currently in dragging mode. If not, we do nothing and return immediately. This prevents the pot from jumping to the latest touch position if it has not been picked up beforehand.

If we are in dragging mode we continue. First we get the new touch position. Then we apply the offset that we discussed in figure 1.3 to that new position. The next line ensures that the y position of the pot stays constant, we want to allow horizontal movement only. Finally, we apply that new position to both pot parts. Great, we're pretty close to finishing the drag and drop functionality.

If you test the app in the current state you'll see that there's one simple yet important step missing...

1.2.3 Dropping an object

Right, the user will also want to drop the pot by releasing the finger from the screen. Otherwise we stay in dragging mode forever and the pot will keep jumping to whichever

position the user taps on the screen.

Luckily this can be easily implemented. All we need to do is to set `isDraggingPot` to false as soon as the user stops touching the screen.

> Add the `touchEnded` method below the `touchMoved` method:
>
> ```
> override func touchEnded(touch: CCTouch, withEvent event:
>     ↪ CCTouchEvent) {
>   isDraggingPot = false
> }
> ```

Awesome! Our drag and drop code is complete! Drag and drop mechanisms can be used in many types of games, so what you have learned in this chapter is very valuable.

Now we can move on to the hardest (thus most interesting) part of this chapter - catching objects.

1.3 Catching objects

Implementing drag and drop was a great warm up. In this section we are going to solve a bunch of problems that will bring our little project a large step closer to being a real game. By the end of this section the user will be able to catch and miss objects by dragging the pot below objects with the right timing.

Before we dive into coding let's think about what we actually need to implement. There are three important aspects that need to be covered through our implementation:

1. detecting if the user has caught an object
2. detecting if the user has missed an object

3. visualizing catching / missing correctly

1.3.1 Thinking in states

Our feature outline describes that objects start out as falling objects, directly after they have been spawned. At some later point in time the user can catch or miss these objects. In each of these situations we need our falling objects to behave differently. If they are falling we want them to move down the screen with a constant speed. If they are caught we need some sort of visualisation - ideally the objects move into the pot and disappear. If the user tries to catch an object too late and misses it closely we want to visualize that, too.

From the paragraph above we can extract three different states in which a falling object can be:

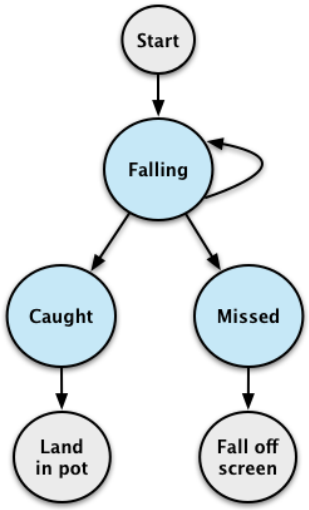


Figure 1.4: Objects start in falling state, then they end up caught or missed

As the diagram shows, a falling object can either stay a falling object or turn into a caught or missed object. It is up to us developers to decide the criteria for a state change. We also need to decide when we want to check for state changes.

For our game I suggest that we check whether a player has caught an object or not in the `update` method. As soon as that object reaches the y position of the top of the pot we decide based on the x position whether the object has been caught or missed

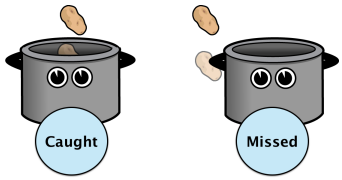


Figure 1.5: Caught objects fall into the pot, missed objects fall behind

Since we are building a 2D game we only have limited ways of expressing that a player missed a falling object - I suggest that we render missed objects behind the pot. That way players can quickly see whether they caught an object or not.

Now we have a good starting point for some coding; we need to store different states for falling objects and we need to write specific behavior code for each of these states. Additionally we need to write code that checks if we have caught or missed an object so that we can assign the correct states to falling objects.

1.3.2 Storing state

Now we'll start implementing everything described earlier. Let's start by adding a `fallingState` to `FallingObject.swift`. That state variable will remember whether an object is currently falling, has been caught or has been missed.

The best way to represent states in Swift is to use an enumeration!

> Add this enum definition to `FallingObject.swift` below the `FallingObjectType` enum:
>
> ```
> enum FallingObjectState {
>   case Falling
>   case Caught
>   case Missed
> }
> ```

As mentioned earlier, associating enum entries with a type is not mandatory. In this case our entries don't need a type (e.g. Int) since the entries will only represent a state - they are values in their own right.

> Next, add an instance variable to store the current state:
>
> ```
> var fallingState = FallingObjectState.Falling
> ```

This variable should not be private, we want to change the value as the object gets caught or missed. Our default state is `.Falling`, we assign it as part of the variable declaration.

Now we can store a `fallingState` for each falling object; next, let's implement different behaviour based on that state.

1.3.3 Implement state specific behaviour

The majority of our gameplay code is currently inside of the `update` method of `MainScene`. This is fairly common for simple games. Currently we are doing to things in the update method: moving the objects down the screen and checking whether they have left the stage entirely (in which case we delete them). Now however, we are going to add code

that will only run for falling objects in certain states. That will add quite a lot of complexity. Instead of squashing everything into the update method I suggest that we create one method for each of the three states. These methods will contain all state specific code and will be called from the update method.

Replace your existing update method with the following one:

```
override func update(delta: CCTime) {
// use classic for loop so that we can remove objects while
   ↪ iterating over the array
for (var i = 0; i < fallingObjects.count; i++) {
  let fallingObject = fallingObjects[i]

  // let the object fall with a constant speed
  fallingObject.position = ccp(
    fallingObject.position.x,
    fallingObject.position.y - CGFloat(fallingSpeed * delta)
  )

  switch fallingObject.fallingState {
  case .Falling:
    performFallingStep(fallingObject)
  case .Missed:
    performMissedStep(fallingObject)
  case .Caught:
    performCaughtStep(fallingObject)
  }
 }
}
```

Now the update method is really easy to read. We loop over all falling objects. In all cases we move the falling object towards the bottom of the screen. After that we check in which state an object is and invoke a method that contains code specific for that state.

We are going to implement these methods throughout the remainder of this chapter.

1.3.4 Implementing the falling state

Let's start implementing the default state: falling. In this state we will need check whether an object has been caught, has been missed or simply remains falling.

Later in this book you will learn how to use the Cocos2D physics engine and its built in collision detection - for this game however we are not using the physics engine and will implement catch detection code ourselves.

In figure 1.5 we have illustrated what we consider a caught/missed object. So how can we implement this? Basically all we need to do is compare the frame of the falling object to the frame of the pot. However, there is one small issue. The frame of CCSprite is always a rectangle that contains the entire texture, here's what the dimensions of the frames of our pot and a falling object looks like:

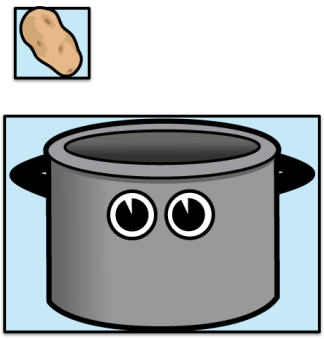


Figure 1.6: The pot frame is too large to use it for collision detection

From the illustration above you can see that the frame of the pot is too large to use it for collision detection. It could easily happen that an object landing on the handle of the pot

would still be considered a catch.

Instead of using the pot dimensions we will need to add a separate, smaller, node in SpriteBuilder that marks the catch area.

> Open the SpriteBuilder project and open *MainScene.ccb*. ...Now publish the SpriteBuilder project and switch back to the Xcode project.

Next, we need to add a variable for the code connection we just set up.

> Add the catch container variable to the other code connection variables:
>
> ```
> weak var catchContainer: CCSprite!
> ```

Now we have a reference container set up that will allow us to test if objects have been caught. We can start implementing falling step function.

> Add the method stub for the falling step to *MainScene.swift*:
>
> ```
> func performFallingStep(fallingObject:FallingObject) {
>
> }
> ```

Before we can dive into collision detection we will have to take a little detour and talk about node transformations. As part of the introduction to Cocos2D we have discussed that nodes are always positioned relative to their parent node (chapter: **??**). The catch container that we just added in SpriteBuilder is a child of the `potBottom` node. We chose that setup so that the catch container always moves together with the pot.

For our collision detection algorithm we want to compare the position of a falling object to the position of our catch container. **Here the problem arises: falling objects and the catch container have different parent nodes, that means we cannot compare their positions and frames directly.** Since the position is relative to the parent node, comparing

nodes with different parents would resolve in unexpected behavior. Take the following illustration as an example:

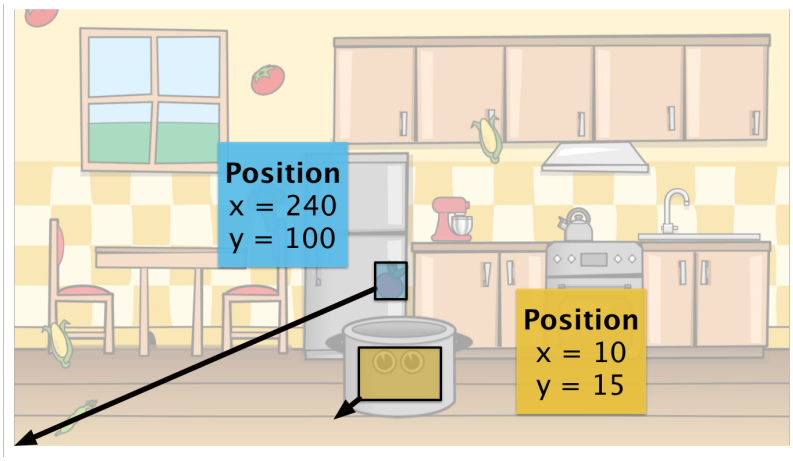


Figure 1.7: Even though the two nodes illustrated above are close to each other, their position values are entirely different, since they are placed relative to different parents

How can we work around this? Luckily Cocos2D exposes a couple of variables and methods that allows us to transform positions and frames between different *node spaces*. Each node lives in the *node space* of its parent. In our example the catch container is in the node space of the pot and the pot is in the node space of the main scene.

If we want to know the position and size of the catching container in the main scene node space, we need to apply the following transform:

```
let containerWorldBoundingBox = CGRectApplyAffineTransform(
  catchContainer.boundingBox(), catchContainer.parent.
    ↪ nodeToParentTransform()
);
```

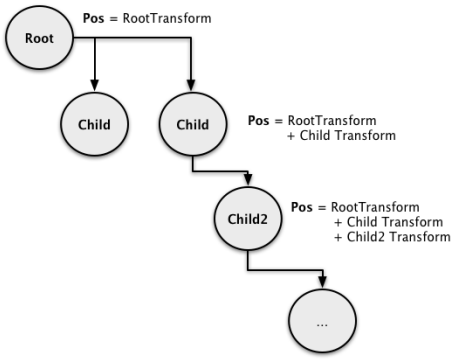
We are transforming the bounding box of the catch container using the `nodeToParentTransform` of the catch container's parent node (the pot). The Cocos2D documentation describes the `nodeToParentTransform` as following: *Returns the matrix that transform the node's (local) space coordinates into the parent's space coordinates.*

This means after applying the transform we know the position of the catch container in the main scene space.

If you are new to graphics programming this concept will likely seem a little confusing; frankly you won't need it too often when working with Cocos2D. If you aren't getting a hold of transforms yet, don't worry too much!

The role of transforms in graphics programming

Transforms are an essential part of all graphics engines - also of Cocos2D. When determining the positions for all nodes in a scene, Cocos2D starts with the root node. After the root node is laid out, the engine moves to the children of the root node, calculates their position and places them *relative to the root node*. This is repeated all the way down the node hierarchy:



Now that we have a solution for the transformation issue, the rest of the code that we need for the falling step is not too complicated.

Here's the entire function that implements the stub that you added earlier:

```
func performFallingStep(fallingObject:FallingObject) {
  let containerWorldBoundingBox = CGRectApplyAffineTransform(
    catchContainer.boundingBox(), catchContainer.parent.
      ↪ nodeToParentTransform()
  );

  let yPositionInCatchContainer = CGRectGetMinY(fallingObject.
     ↪ boundingBox()) < CGRectGetMaxY(
     ↪ containerWorldBoundingBox)
  let xPositionLargerThanLeftEdge = CGRectGetMinX(
     ↪ fallingObject.boundingBox()) > CGRectGetMinX(
     ↪ containerWorldBoundingBox)
  let xPositionSmallerThanRightEdge = CGRectGetMaxX(
     ↪ fallingObject.boundingBox()) < CGRectGetMaxX(
     ↪ containerWorldBoundingBox)

  // check if falling object is inside catching pot, trigger
     ↪ this when object reaches top of pot
  if (yPositionInCatchContainer) {
    if (xPositionLargerThanLeftEdge &&
       ↪ xPositionSmallerThanRightEdge) {
      // caught the object
      let fallingObjectWorldPosition = fallingObject.parent.
         ↪ convertToWorldSpace(fallingObject.
         ↪ positionInPoints)
      fallingObject.removeFromParent()
      fallingObject.positionInPoints = potTop.
         ↪ convertToNodeSpace(fallingObjectWorldPosition)
      potTop.addChild(fallingObject)
      fallingObject.fallingState = .Caught
    } else {
      fallingObject.fallingState = .Missed
    }
```

```
        }
    }
```

We have already discussed the first statement extensively, we transform the bounding box of our catch container. That allows us to compare its position to the position of falling objects.

The next three lines are each used to determine if the falling object is within the relevant boundaries of our transformed catch container. The `CGRectGetMin...` utility functions are used to get the lowest/highest value on a certain axis from the bounding box. These three statements check for the conditions outlined in figure 1.5. If all three are true the player has caught the object.

Next, we have an if statement that combines the three boolean variables. The first if statement checks if the falling object is in the *critical area* using the `yPositionInCatchContainer` constant. Here the y position of the falling object is the only relevant metric. If we aren't in the critical area we do nothing at all - the object is still too far above the pot for us to decide whether the player caught it or not.

If the object is in the critical area we now need to determine if it has been caught or missed. This is where we need the two x position variables. If the object is outside of the bounds we set the `fallingState` to `.Missed`.

If the object is inside of the bounds we set the `fallingState` to `.Caught`. Additionally we need to ensure that once the object is caught it stays within the pot. Without additional code the caught objects are not attached to the pot. The player could move the pot left or right and the objects would fall out to the side of the pot. As soon as an object is caught we need to turn it into a child node of the pot, that way they will stick together.

Here we once again need a transform. We want to turn the falling object into a child of the pot instead of being a child of main scene. That means we are moving the object to

a different node space. We don't want the player to see this move happen; visually the object should stay at exactly the same position.

In such situations we need to use a two step transform. First, we need to find the *world space* position of the node that we are moving to a different node space. The position in the world space is expressed relative to the world root (in most cases the bottom left corner of the screen) and not relative to the parent node. You can think of the position in world space as a global or absolute position. We can use the world position to find the corresponding relative position in any node space.

Let's take a look at our specific code. First we call:

```
let fallingObjectWorldPosition = fallingObject.parent.
    ↪ convertToWorldSpace(fallingObject.positionInPoints)
```

This line asks: *What is the global position, independent of the parent node, of this falling object?* The node that receives this question needs to be the parent node of `fallingObject`, because that is the node responsible for placing the `fallingObject` node.

Now that we have saved the position, we remove the node from its parent. Next we perform the second step of the transformation:

```
fallingObject.positionInPoints = potTop.convertToNodeSpace(
    ↪ fallingObjectWorldPosition)
```

This line asks: *Dear pot, I have a global position for this falling object, could you tell me what the relative position to you would need to be? I want the falling object to remain at the same global position after adding it to you as a child.* After we have determined the right position we finally add the falling object to the pot. The object has switched to a different node space without that the player could realize it!

1.3.5 Implementing the missed state

1.3.6 Implementing the caught state

Index