ATLS 4120/5120: Mobile Application Development
Week 7: Animations

Depending on the complexity and level of control needed for your animations, iOS provides different methods to create animations. There are also outside libraries you can integrate if desired.

Core Graphics
The Core Graphics Framework provides the basic building blocks for drawing and positioning views.
- C API (not object-oriented)

Coordinate System
(slide)
- The iOS coordinate system is set up so that point (0,0) is in the top-left corner.
  - Y values grow larger going down the screen
  - X values grow larger going across the screen
- Units are points not pixels
- Points represent a logical coordinate space and are not always equal to pixels
  - device and screen size independent.
- The result is that if you draw the same content on two similar devices, with only one of them having a high-resolution screen, the content appears to be about the same size on both devices.
- Before iOS 4, this scale factor was assumed to be 1.0, but in iOS 4 and later it may be either 1, 2, or 3, depending on the resolution of the device.
- CGPoint is a struct that defines a point {x,y}
  https://developer.apple.com/documentation/coregraphics/cgpoint
  - center property defines the center point
  - changing the values moves the element
- CGFloat is used for floating points in Core Graphics
  https://developer.apple.com/documentation/coregraphics/cgfloat
- CGSize is a struct that defines the size through width and height {width, height}
- CGRect is a struct that describes the rectangle a UIView lives in {origin x, origin y, size width, size height}

Timer
- The Timer class creates timer objects which can call a method at a regular interval
- The scheduledTimer(timeInterval:target:selector:userInfo:repeats:) method creates a timer with an interval and starts it
- After a Timer instance is started you cannot change its firing interval. You must stop the timer and create a new one.
- The invalidate() method stops the timer

UIKit/UIView Animations
The UIView class in UIKit provides some basic animation functions.
UIView coordinates
(slide)
- A UIView object tracks its size and location using its frame, bounds, and center properties
- Frame: a rectangle positioned from the perspective of the parent view
- Bounds: a rectangle positioned from the perspective of the view itself, usually at (0,0)
- Center: the center of your view in the frame
- To move a view you need to either set its center or frame property.

UIView Animations
Creating animations on your views is a matter of changing properties on them and letting UIKit animate them automatically. https://developer.apple.com/documentation/uikit/uiview#1652828
UIView has the following animatable properties.

- center
- frame
- bounds
- alpha – view's transparency (default is 1 not transparent)
- transform – translation, rotation, and scale
- backgroundColor

All animations involve a change of one or more of these properties.

The UIViewPropertyAnimator class provides more control over animations by enabling control over the dynamic modification of those animations.
https://developer.apple.com/documentation/uikit/uiviewpropertyanimator
When creating a property animator object, you specify the following:

- A block containing code that modifies the properties of one or more views.
- The timing curve that defines the speed of the animation over the course of its run.
    - Predefined curves: UIView.AnimationCurve https://developer.apple.com/documentation/uikit/uiview/animationcurve
- The duration (in seconds) of the animation.
- An optional completion block to execute when the animations finish.

Blocks, or closures are self-contained blocks of functionality that can be passed around and used in your code. They are similar to anonymous functions or lamdas in other programming languages.
Closures can capture and store references to any constants and variables from the context in which they are defined. This is known as *closing over* those constants and variables.

The UIViewAnimating protocol provides methods for implementing the basic flow control for animations, including the ability to start, stop, and pause animations.
https://developer.apple.com/documentation/uikit/uiviewanimating
The UIViewImplicitlyAnimating protocol provides methods to modify animations while they're running. https://developer.apple.com/documentation/uikit/uiviewimplicitlyanimating

Transforms
https://developer.apple.com/documentation/uikit/uiview/1622459-transform
The transform property is of type CGAffineTransform which is a struct for holding an affine transformation matrix. https://developer.apple.com/documentation/coregraphics/cgaffinetransform
An affine transformation is a special type of mapping that preserves parallel lines in a path but does not necessarily preserve lengths or angles.
In geometry an affine transformation is a linear transformation which preserves co-linearity and ratio of distances. This means that all the points lying on a line initially will remain in a line after the transformation, with the respective distance ratios between them maintained.

- Translation
- Rotation
- Scaling

Core Animation
The Core Animation framework is the backbone for any UIKit animation. So if you want more control over every frame in an animation you should use the underlying core animation APIs directly.

UIKitDynamics
UIKit Dynamics is the physics engine for UIKit which enables you to add any physics behaviors like collision, gravity, push, snap, etc, to the UIKit controls.

In our example we'll create four versions of an animation using UIKit and the Timer classes.

Animation
Create a new Single-view app called animation.
Add tennis_ball152.png into your Assets folder and rename it ball.
For Interface Builder set the view to be iPad.
Add an image view and in the attributes inspector set the image to ball.
Set the size of the image view to the size of the image (152 x 152) (or Editor | Size to Fit Content)
Add 2 labels and a slider.
One label should say Interval and the other one will hold the slider's value.
The slider should have a min of 0 and a max of 1. Set the initial value to .1
Now create outlet connections for the slider called slider, for its label called sliderLabel, and for the image called imageView.
Also create an action for the slider called sliderMoved for when the slider is moved.

<u>Constraints:</u>
I gave the slider a width constraint (150) to avoid it shrinking.
I gave the label a width constraint (35) so it didn't move the other elements as its width changed.
Embed both labels and slider in a horizontal stack view and add some spacing (20). Add constraints for the stack view: center horizontally and align to bottom (30).
If we give the image constraints, the image will go back to that position every time we change the slider so instead we'll set its starting position programmatically.

ViewController.swift
First we're going to move our image using the Timer class.

We will need some variables in our class.
```
    var delta = CGPoint(x: 12, y: 4) //initialize the delta to move 12
pixels horizontally, 4 pixels vertically
    var ballRadius = CGFloat() //radius of the ball image
    var ballTimer = Timer() //animation timer
```

We need a method that changes the position of the image view.
```
    //changes the position of the image view
    @objc func moveImage(){
        let newCenter = CGPoint(x: imageView.center.x + delta.x, y:
imageView.center.y + delta.y)
        imageView.center = newCenter
        if imageView.center.x > view.bounds.size.width-ballRadius ||
imageView.center.x < ballRadius {
            delta.x = -delta.x
        }
        if imageView.center.y > view.bounds.size.height-ballRadius ||
imageView.center.y < ballRadius {
            delta.y = -delta.y
        }
    }
```

We need the @obj attribute to make the method available to Objective-C code, which we'll need to do because of the Timer class.

Now we create a method that's going to create a timer based on the chosen interval.

```
    //updates the timer and label with the current slider value
    func changeSliderValue() {
        sliderLabel.text = String(format: "%.2f", slider.value)
        ballTimer = Timer.scheduledTimer(timeInterval: Double(slider.value),
target: self, selector: #selector(self.moveImage), userInfo: nil, repeats:
true)
    }
```

We have to use #selector because that parameter is expecting an Objective-C function.

Now we implement the method that will be called when the user moves the slider.

```
    @IBAction func sliderMoved(_ sender: UISlider) {
        ballTimer.invalidate()
        changeSliderValue()
    }
```

Let's do some set up in viewDidLoad

```
    override func viewDidLoad() {
        //ball radius is half the width of the image
        ballRadius=imageView.frame.size.width/2
        //set the starting position for the image view
        imageView.center.x = view.bounds.size.width/2
        imageView.center.y = view.bounds.size.height/2
        changeSliderValue()
        super.viewDidLoad()
}
```

Your ball should now move.

At the slower speeds it's pretty choppy. Let's make the animation smoother using UIView animations to move the image view.

<u>Animation</u>

In `moveImage()` after the newCenter constant comment out/remove:

```
imageView.center = newCenter
```

And replace with the following:

```
        let duration = Double(slider.value)
        let animator = UIViewPropertyAnimator(duration: duration, curve:
.linear, animations: {self.imageView.center=newCenter})
        animator.startAnimation()
```

Now try it. At the faster speeds it's smoother.

Now let's use transforms in Core Graphics to do translation, rotation, and scaling.

Translation

In translation we'll use the transform property instead of center.

In ViewController.swift add

```
var translation = CGPoint(x: 0.0, y: 0.0) //specifies how many pixels the
image will move
```

Then edit moveImage and replace in the animation block

```
animations: {self.imageView.center=newCenter}
```

with the following:

```
animations: {self.imageView.transform=CGAffineTransform(translationX:
self.translation.x, y: self.translation.y)
            self.translation.x += self.delta.x
            self.translation.y += self.delta.y
        })
```

Modify to use translation

```
        if imageView.center.x + translation.x > self.view.bounds.size.width-
ballRadius || imageView.center.x + translation.x < ballRadius{
            delta.x = -delta.x
        }
        if imageView.center.y + translation.y > self.view.bounds.size.height
- ballRadius || imageView.center.y + translation.y < ballRadius {
            delta.y = -delta.y
        }
```

Rotation

The rotation transformation enables you to rotate a view using the angle you specify.

ViewController.swift

```
var angle: CGFloat=0.0 //angle for rotation
```

Edit moveImage

Change the animation block to implement rotation

```
animations: {self.imageView.transform=CGAffineTransform(rotationAngle:
self.angle)
            self.imageView.center=CGPoint(x: self.imageView.center.x +
self.delta.x, y: self.imageView.center.y + self.delta.y)
        }
```

Remove

```
        self.translation.x += self.delta.x
        self.translation.y += self.delta.y
```

Add

```
        angle += 0.02 //amount by which you increment the angle
        //if it's a full rotation reset the angle
        if angle > .pi {
            angle=0
        }
```

Modify (remove translation piece)

```
        if imageView.center.x > self.view.bounds.size.width-ballRadius ||
imageView.center.x < ballRadius{
            delta.x = -delta.x
        }
        if imageView.center.y  > self.view.bounds.size.height-ballRadius ||
imageView.center.y  < ballRadius {
            delta.y = -delta.y
        }
```

Scaling

You can change the transform to scale by using angle as the factor to scale the x and y axis
Change the animations block:

```
self.imageView.transform=CGAffineTransform(scaleX: self.angle, y:
self.angle)
```