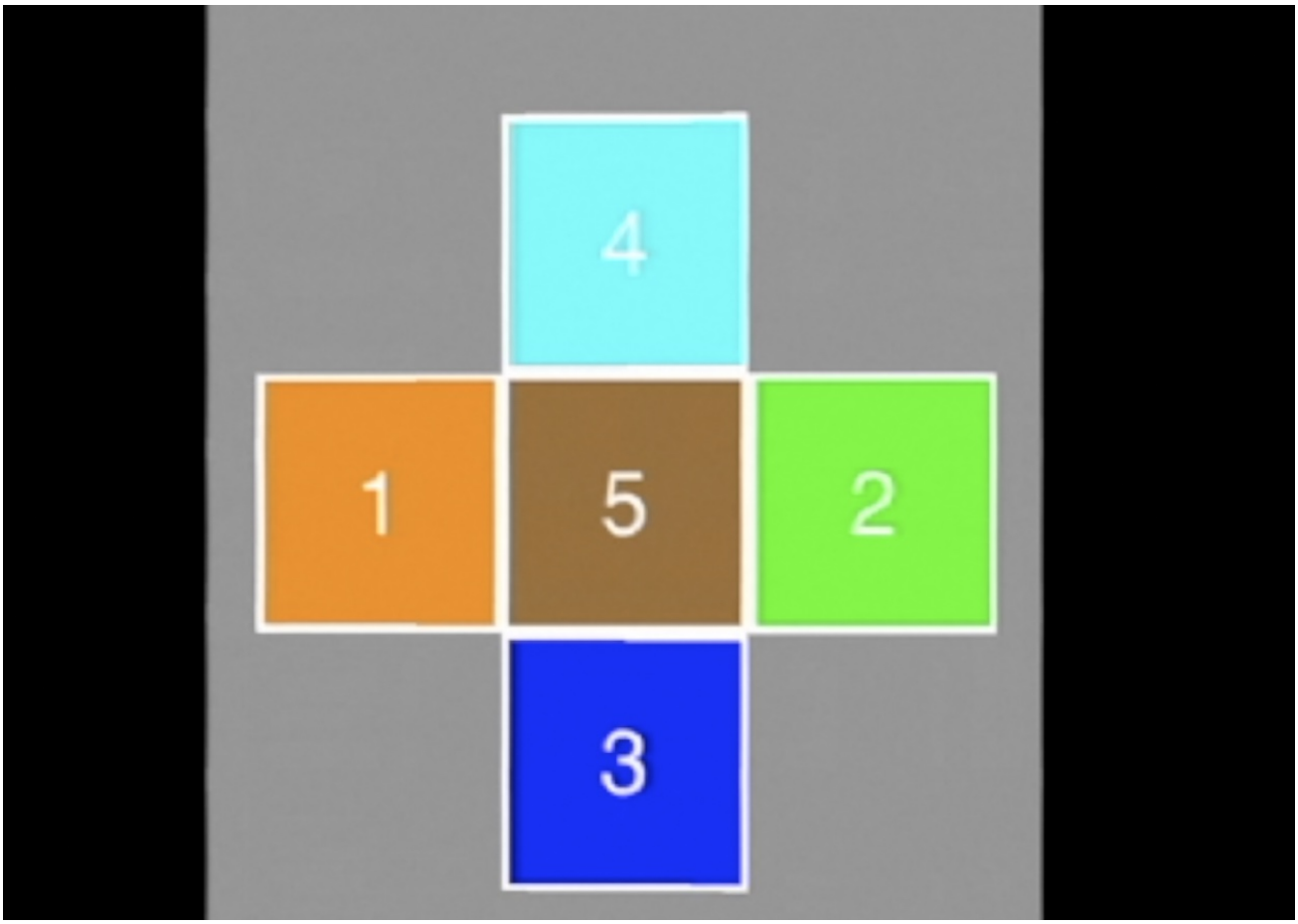
Core Animation Layers forming a 3D cube

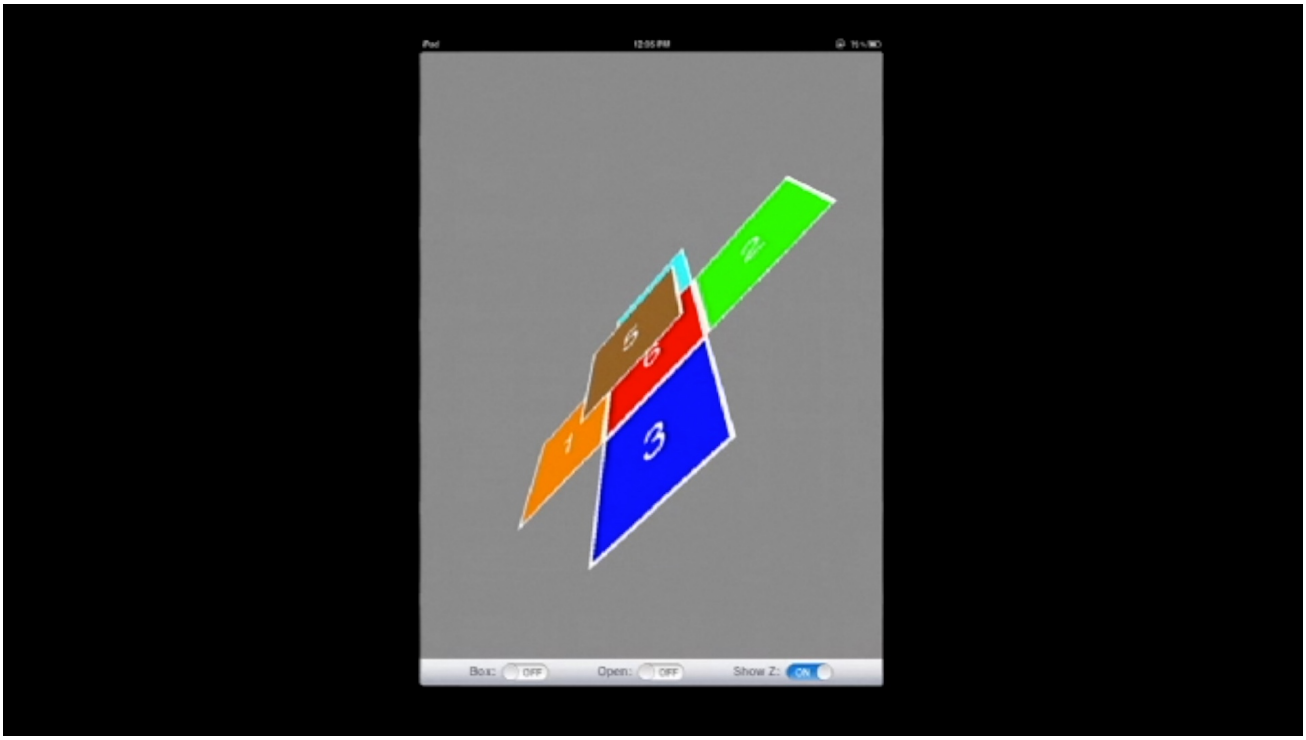
I've been researching `Core Animation` framework for the past few months. I've started with several books on the subject, but I found watching related WWDC videos most rewarding. The presenters put the content in a relevant context which makes it easier to apprehend and learn from it.

1. Introduction

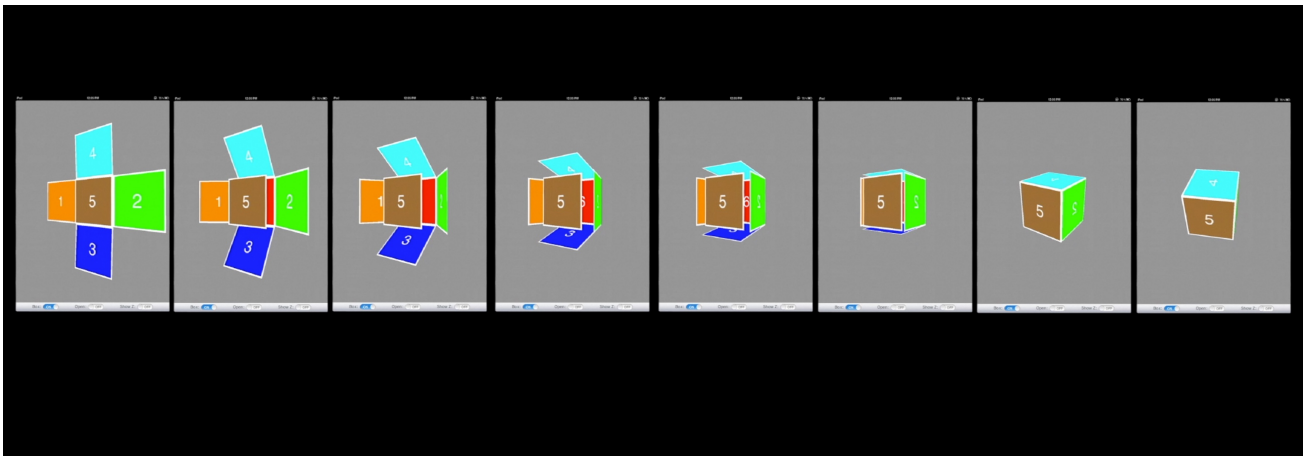
One WWDC session particularly intrigued me: "2011–421 Core Animation Essentials". They presented their demo named: "Layers in Perspective", and it showed six layers, forming a flattened cube:



The sixth layer is hiding behind the layer number 5. It has a lower `zPosition` then the layer above it.



They have also demonstrated opening and closing the cube formation.



So, I have decided to demonstrate that.

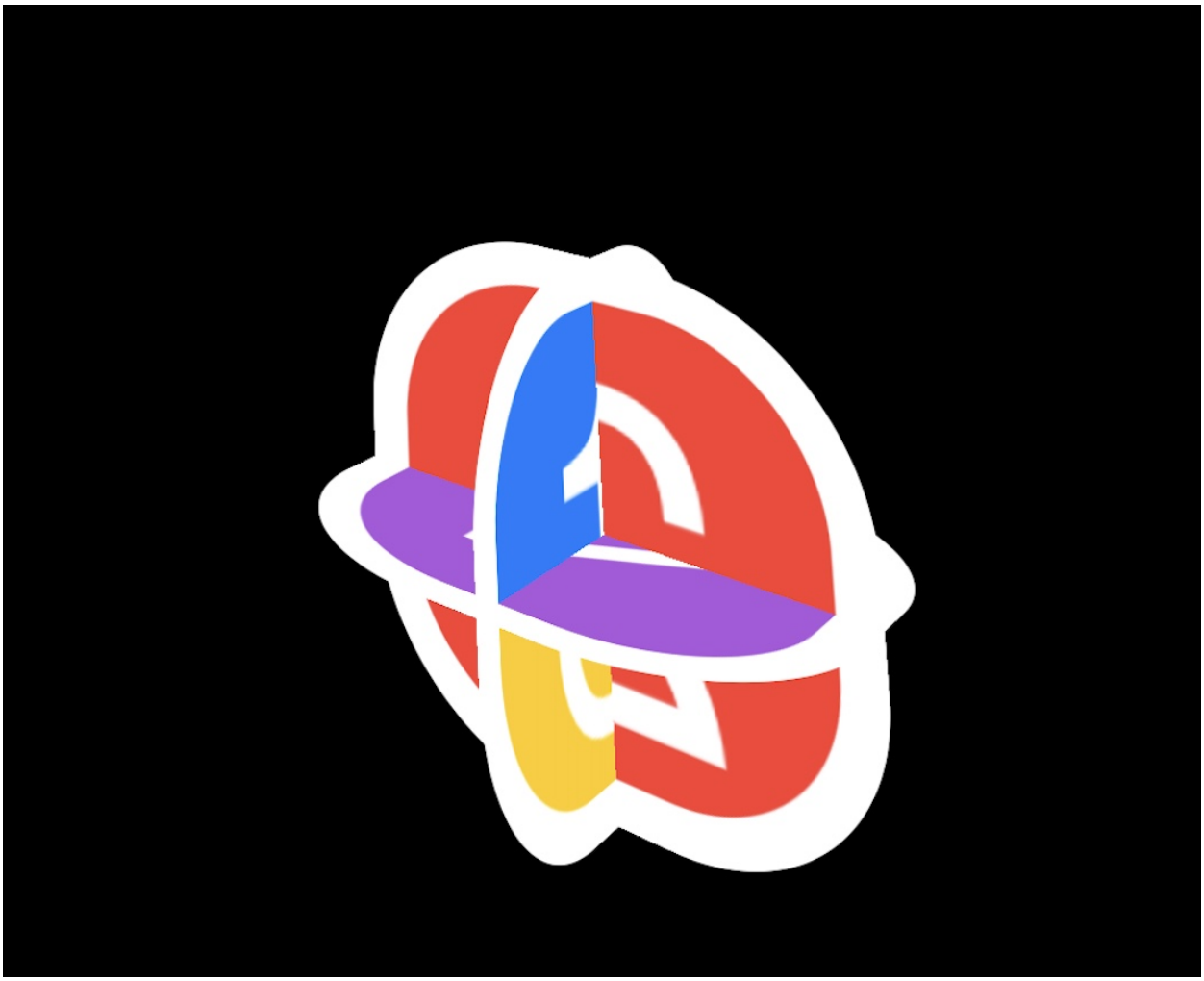
Here's a link to a Codeberg repo with the source code: Core Animation 3D Cube

Here's the animation:

You can also see it on You Tube

Layers in `Core Animation` live in `3D` geometry. But a layer is a `2D` construct, so `Core Animation` coordinate space is called a `2.5D` geometry.

To illustrate that just see what happens when you mess up your transformations.



Layers are `2D` objects, they don't understand the third dimension. They are like playing cards in space., and there is no `depth buffer` available.

And also, intersecting layers should be avoided because in the image above, Core Animation needs to do a lot of work. So just to draw the `red` layer intersecting only the `blue` layer, Core Animation needs to

cut the `red` layer into two pieces
render back part of the `red` layer
then render the full `blue` layer
then render the front part of the `red` layer again

And all that work is for just intersection, and here we have multiple.

2. Building the Cube in 3D

2.1. Setting up the multi-platform project

I wanted the project to fun on the `macOS` , `iOS` and `iPadOS`, so I used AllApples Swift package.

After removing the storyboards and pimping up the `AppDelegate` and `main.swift` for the `macOS` version, and `SceneDelegate` for the mobile versions, I was ready to start.

2.1.1. `main.swift` for the macOS

```
import Cocoa
let delegate = AppDelegate()
NSApplication.shared.delegate = delegate
_ = NSApplicationMain(CommandLine.argc, CommandLine.unsafeArgv)
```

2.1.2. AppDelegate for the `macOS`

```
import Cocoa
import AllApples

class AppDelegate: NSObject, NSApplicationDelegate {
  private var window: NSWindow?
  func applicationDidFinishLaunching(_ aNotification: Notification) {
    window = AppSceneDelegate.makeWindow_Mac(theVC: CommonViewController())
  }
}
```

2.1.3. SceneDelegate for the `iOS` and `iPadOS`

```
import UIKit
import AllApples

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
  var window: UIWindow?
  func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
    guard let aScene = (scene as? UIWindowScene) else { return }
    window = AppSceneDelegate.makeWindow_iOS(theScene: aScene, theVC:
CommonViewController())
  }
}
```

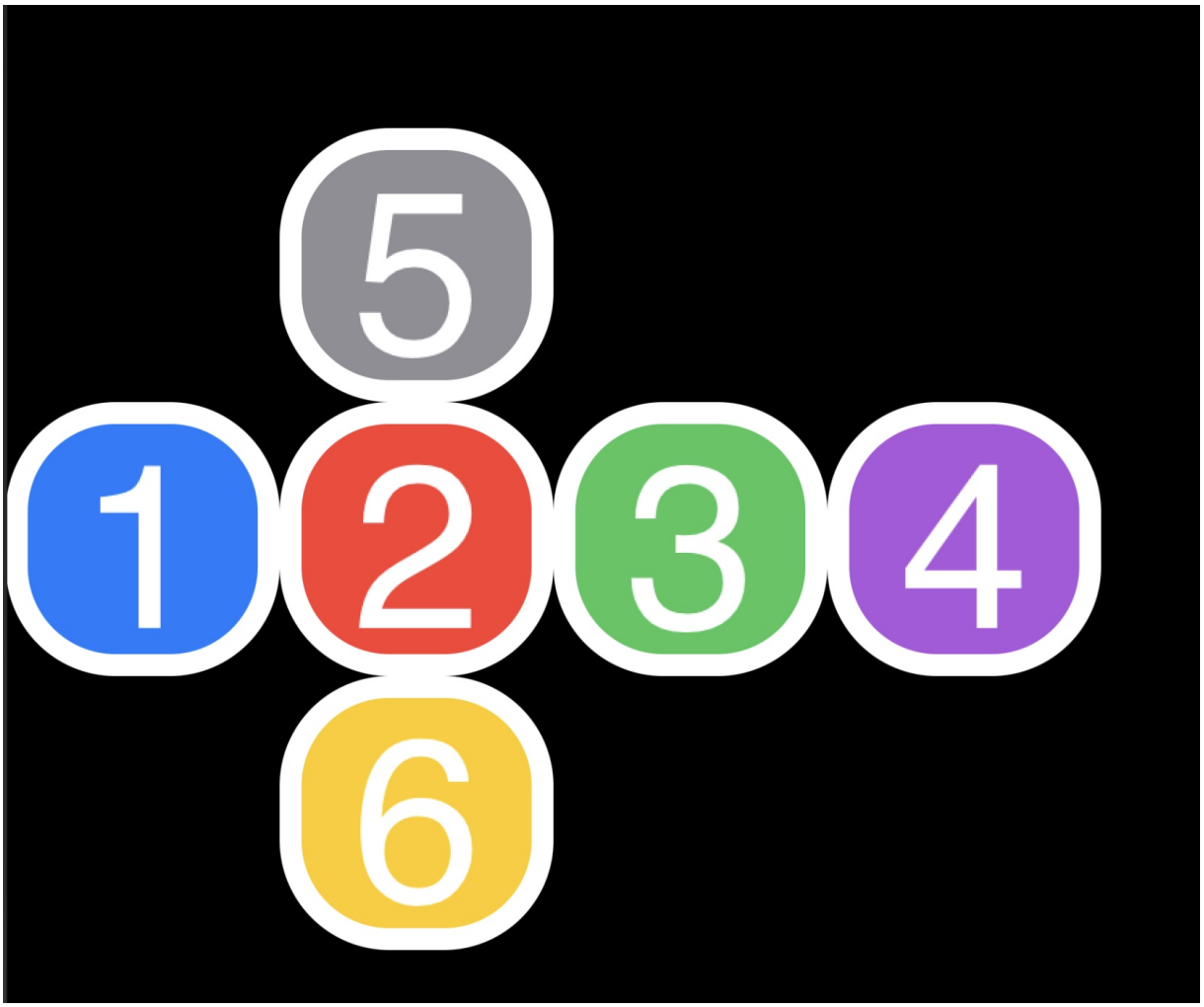
2.2. Building the Basic Blocks

The first I did is make a `PlainLayerView` . It is an `AView` descendant, which means that it is `typedef-ed` to be either a `UIView` or a `NSView`.

It is an intermediary object just to set up a thing on the `macOS` as well (doesn't work yet).

I then created `CustomLayerView` to have nice sides for the cube, with `CATextLayer` as the number of the cube side, and a nice rounded border, so that we can peek into our cube while rotating.

Here's the image of all six layers drawn by using a `CustomLayerView`



This layout was abandoned because I couldn't make the transformation of `purple` view to work when transforming in `3D`.

The solution is to add an additional `CATransformLayer` to the `green` layer, and mount the `purple` layer onto it. As explained in this blog post by Oliver Drobnik [Cubed CoreAnimation Conundrum.]

But I didn't want to lose the linearity of the solution, and I used the approach demonstrated in the mentioned WWDC session: "2011–421 Core Animation Essentials"

They used the `zOrder` property of a layer, and so I put `purple` layer on top of the `red` layer to achieve that.

```
    if number == 4 {
      view.layer.zPosition = 1
    }
```

As you can see in the image below, the `purple` layer is in front of the `red` layer, which is obvious when we rotate the flattened cube.

2.3. Turning the Transform On and Off

I did take the approach that Over Drobnik did in his article: [Cubed CoreAnimation Conundrum](), and used it like this:

```
    side4.layer?.zPosition = on ? CACube3DView.sideWidth : 1
    side1.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(1,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
    side2.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(2,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
    side3.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(3,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
    side4.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(4,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
    side5.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(5,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
    side6.layer?.transform = on ? CATransform3D.transformFor3DCubeSide(6,
zWidth: CACube3DView.sideWidth)  : CATransform3DIdentity
```

2.4. Transforming the Layers to Form a Cube

I didn't use his transform code, since he used that additional `CATransformLayer`, so it wouldn't work. So, here's a small extension on the `CATransform3D`

```
public extension CATransform3D {
  static func transformFor3DCubeSide(_ number: Int, zWidth: CGFloat) ->
CATransform3D {
```

```
    let halfPi = CGFloat(Double.pi) / 2.0
    var trans = CATransform3DIdentity
    switch number {
      case 1:
        trans = CATransform3DMakeRotation(halfPi, 0, 1, 0)
        break
      case 2:
        trans = CATransform3DIdentity
        break
      case 3:
        trans = CATransform3DMakeRotation(-halfPi, 0, 1, 0)
        break
      case 4:
        trans = CATransform3DMakeTranslation(0, 0, zWidth)
        break
      case 5:
        trans = CATransform3DMakeRotation(-halfPi, 1, 0, 0)
        break
      case 6:
        trans = CATransform3DMakeRotation(halfPi, 1, 0, 0)
        break
      default:
        break
    }
    return trans
  }
}
```

I actually used the approach form that WWDC session, and also used `anchorPoint` properties of the `CALayer`.

2.5. Positioning Cube Sides

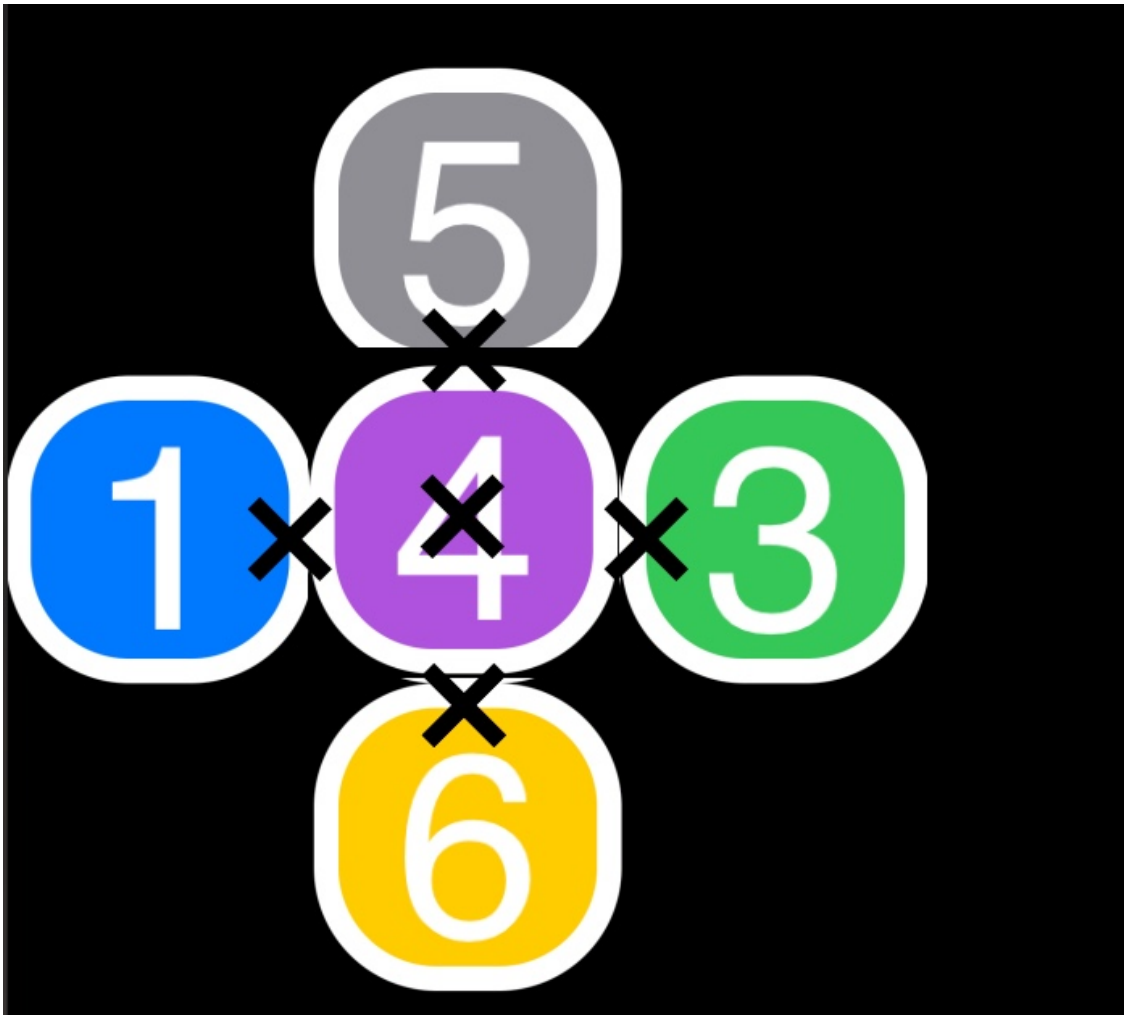
Here's a little extension on `CGPoint` that returns a tuple of our cube side positions and anchor points:

```
public extension CGPoint {
  static func anchorPointAndPositionForCubeSideLayer(number: Int, sideSize:
CGFloat) -> (anchorPoint: CGPoint, position: CGPoint) {
    var resultAnchorPoint = CGPoint(x:0.5, y:0.5)
    var resultPosition = CGPoint(x:0.0, y:0.0)
    let halfSideSize: CGFloat = sideSize / 2.0
    switch number {
      case 1:
        resultAnchorPoint = CGPoint(x:1.0, y:0.5)
        resultPosition = CGPoint(x:-halfSideSize, y:0.0)
        break
      case 2:
        resultAnchorPoint = CGPoint(x:0.5, y:0.5)
        resultPosition = CGPoint(x:0.0, y:0.0)
        break
```

```
      case 3:
        resultAnchorPoint = CGPoint(x:0.0, y:0.5)
        resultPosition = CGPoint(x:halfSideSize, y:0.0)
        break
      case 4:
        resultAnchorPoint = CGPoint(x:0.5, y:0.5)
        resultPosition = CGPoint(x:0.0, y:0.0)
        break
      case 5:
        resultAnchorPoint = CGPoint(x:0.5, y:1.0)
        resultPosition = CGPoint(x:0.0, y:-halfSideSize)
        break
      case 6:
        resultAnchorPoint = CGPoint(x:0.5, y:0.0)
        resultPosition = CGPoint(x:0.0, y:halfSideSize)
        break
      default:
        break
    }
    return (anchorPoint: resultAnchorPoint, position: resultPosition)
  }
}
```

In the image below I have marked where the `anchor points` are for each layer:

The only fallacy in the image above, is that the `purple` layer is actually above our `red layer`, but I wanted to show where those anchor points are. So, the actual image looks like this, but we now don't see the `red` layer.



An Anchor point is a center of rotation. It determines how will the layer rotate. Imagine holding a playing card with two fingers. Then try to spin the card. The anchor point of that card is where you are holding it with fingers.

3. Responding to Gestures

I made a small `GestureRecognizerView` to be able to respond to gestures and move, rotate and scale our layers.

It hooks-up:

- `NSPanGestureRecognizer` and `UIPanGestureRecognizer`
- `func rotate(with event: NSEvent)` and `UIRotationGestureRecognizer`
- `NSClickGestureRecognizer` and `UITapGestureRecognizer`
- `func magnify(with event: NSEvent)` and `UIPinchGestureRecognizer`

It then exposes all those events to the developer to use:

```
public extension GestureRecognizerView {
  @objc func rotationChanged(degrees: Float) {}
  @objc func rotationChanged(radians: Float) {}
```

```
  @objc func displacementChanged(displacement: CGPoint) {}
  @objc func scaleChanged(scale: CGFloat) {}
  @objc func tapHappened() {}
}
```

4. Building a Layer to hold a Cube

`CACube3DView` will hold the six layers than (can) make a cube. In order for `Core Animation` to render the transformed views in perspective, there is a property `sublayerTransform`.

You either use that property of your parent layer, or add another layer class to your layer hierarchy: `CATransformLayer`. I opted to use that.

```
private(set) public lazy var transformedLayer: CALayer = {
    let l = CATransformLayer()
    l.name = "Transform Layer"
    #if os(OSX)
    l.isGeometryFlipped = true
    #endif
    return l
}()
```

When adding sublayers, I add them to this `transformedLayer` property, and not my view's layer.

```
func addSideSubview(_ subview: AView) {
    addSubview(subview)

    #if os(iOS) || os(tvOS)
    transformedLayer.addSublayer(subview.layer)
    #endif

    #if os(OSX)
    if let aLayer = subview.layer {
      transformedLayer.addSublayer(aLayer)
    } else {
      fatalError("`subview.layer` == `nil`")
    }
    #endif
}
```

5. Perspective & Rotation

When the app first runs it shows in perspective. I made a little extension:

```
public extension CATransform3D {
  static func somePerspectiveTransform() -> CATransform3D {
    var perspective = CATransform3DIdentity;
    perspective.m34 = -1.0 / 500.0;
    perspective = CATransform3DRotate(perspective, CGFloat(Double.pi) / 8, 1,
```

```
0, 0);
    perspective = CATransform3DRotate(perspective, CGFloat(Double.pi) / 8, 0,
1, 0);
    perspective = CATransform3DScale(perspective, 0.7, 0.7, 0.7)
    return perspective
  }
}
```

The part: `perspective.m34 = -1.0 / 500.0;` sets the perspective. The `.34` field of a matrix shows the amount of perspective distortion applied. The amount `500` is often used in examples. If it were smaller, the layers would seem very close and distorted, like a fish-eye effect.

This is the initial transform, but we want to be able to move and rotate our cube (flattened or not) with our fingers.

Here's the code:

```
public extension CATransform3D {

  func rotationFromDisplacement(_ displacement: CGPoint, sideWidth: CGFloat,
is3D: Bool) -> CATransform3D {

    var currentTransform = self

    let totalRotation: CGFloat = sqrt(displacement.x * displacement.x +
displacement.y * displacement.y)
    let angle: CGFloat = totalRotation * .pi / 180.0

    let xRotationFactor = displacement.x / angle
    let yRotationFactor = displacement.y / angle

    if is3D {
      currentTransform = CATransform3DTranslate(currentTransform, 0, 0,
sideWidth / 2.0)
    }

    var rotationalTransform = CATransform3DRotate(currentTransform, angle,
                                                  (xRotationFactor *
currentTransform.m12 - yRotationFactor * currentTransform.m11),
                                                  (xRotationFactor *
currentTransform.m22 - yRotationFactor * currentTransform.m21),
                                                  (xRotationFactor *
currentTransform.m32 - yRotationFactor * currentTransform.m31))

    if (is3D) {
      rotationalTransform = CATransform3DTranslate(rotationalTransform, 0, 0,
-sideWidth / 2.0);
    }

    return rotationalTransform
  }
}
```

We call it from our pan-gesture methods

```
  override func displacementChanged(displacement: CGPoint) {
    guard !(displacement.x == 0 && displacement.y == 0) else { return }

    let rotationTransform =
transformedLayer.sublayerTransform.rotationFromDisplacement(displacement,
sideWidth: CACube3DView.sideWidth, is3D: isOn)
    transformedLayer.sublayerTransform = rotationTransform
  }
```

We hooked up the pan-gestures prior:

```
#if os(OSX)
private func setupPanGestureRecognizer() {
    let panGR = NSPanGestureRecognizer(target: self, action:
#selector(handlePanGesture(_:)))
    addGestureRecognizer(panGR)
}
@objc func handlePanGesture(_ gestureRecognizer: NSPanGestureRecognizer) {
    let displacement: CGPoint = gestureRecognizer.translation(in: self)
    handlePan(displacement: displacement, changed: gestureRecognizer.state
== .changed)
}
#endif

#if os(iOS) || os(tvOS)
private func setupPanGestureRecognizer() {
    let panGR = UIPanGestureRecognizer(target: self, action:
#selector(handlePanGesture(_:)))
    addGestureRecognizer(panGR)
}

@objc func handlePanGesture(_ gestureRecognizer: UIPanGestureRecognizer) {
    let displacement: CGPoint = gestureRecognizer.translation(in: self)

    handlePan(displacement: displacement, changed: gestureRecognizer.state
== .changed)

    if gestureRecognizer.state == .changed {
      gestureRecognizer.setTranslation(.zero, in: self)
    }
}
#endif
```

We can also add a simple rotation transform for the rotation events/ gestures.

```
  override func rotationChanged(radians: Float) {
    let transform = transformedLayer.sublayerTransform
    let rot = CATransform3DRotate(transform, CGFloat(radians), 0, 1, 0)
    transformedLayer.sublayerTransform = rot
  }
```

And scale, in all three axes:

```
  override func scaleChanged(scale: CGFloat) {
    let scaleTransform = CATransform3DScale(transformedLayer.sublayerTransform,
scale, scale, scale)
    transformedLayer.sublayerTransform = scaleTransform
  }
```

The `tap` turns on and off our 3D transform

```
  override func tapHappened() {
    set3DCube(on: isOn)
  }
```

Here's the 3D cube code

```
  func set3DCube(on: Bool) {
    side4.layer.zPosition = on ? CACube3DView.sideWidth : 1
    side1.layer.transform = on ? CATransform3D.transformFor3DCubeSide(1,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
    side2.layer.transform = on ? CATransform3D.transformFor3DCubeSide(2,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
    side3.layer.transform = on ? CATransform3D.transformFor3DCubeSide(3,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
    side4.layer.transform = on ? CATransform3D.transformFor3DCubeSide(4,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
    side5.layer.transform = on ? CATransform3D.transformFor3DCubeSide(5,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
    side6.layer.transform = on ? CATransform3D.transformFor3DCubeSide(6,
zWidth: CACube3DView.sideWidth) : CATransform3DIdentity
  }
```

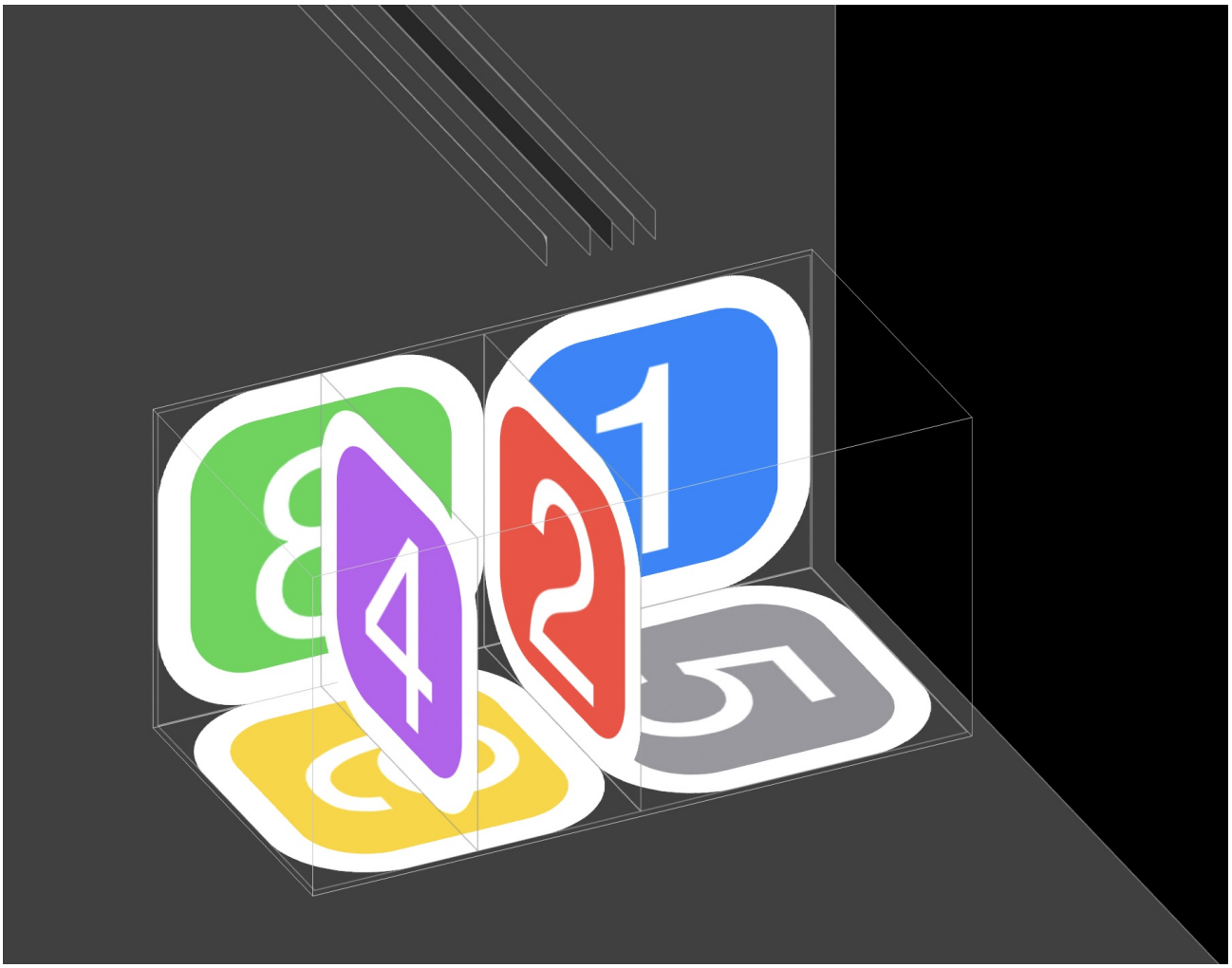
We either set the `identity transform`, which means `no transform`, to our cube sides, or the transform appropriate for that particular side.

6. macOS Troubles

I suppose I need to do further investigation in how `macOS` treats layers of `NSView`, for this little experiment is not working on the `macOS`, yet. Here's how the flattened cube looks on the `macOS`

So, the positioning of the layers is not respected.

And here is the cube in 3D:



I did try to force the `isGeometryFlipped = true` everywhere. Anyway this needs more work.

If you want to help with the `macOS` implementation, please, be my guest.

Here's a link to a Codeberg repo with the source code: [Core Animation 3D Cube]