

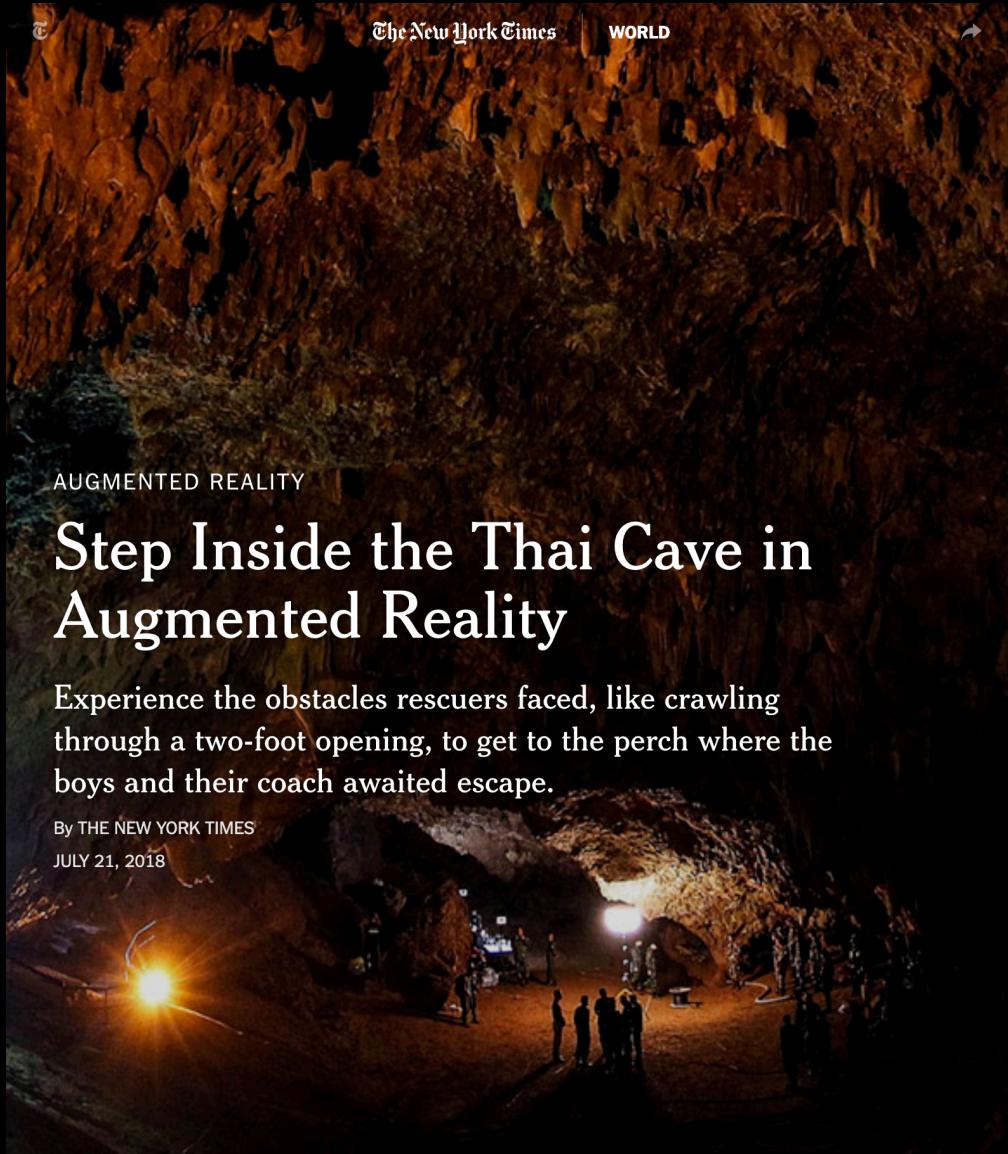
# Welcome to the ARCore & Sceneform Workshop

- Before we get started...
- Clone [github.com/ramonaharrison/ar-workshop](https://github.com/ramonaharrison/ar-workshop)
- Make sure Android Studio is updated to 3.1 or higher
- Set up an ARCore supported device
- Or an ARCore supported emulator

# About this workshop

- We are Ramona Harrison & David Morant
- We work on Android @ The New York Times

# AR @ NYT



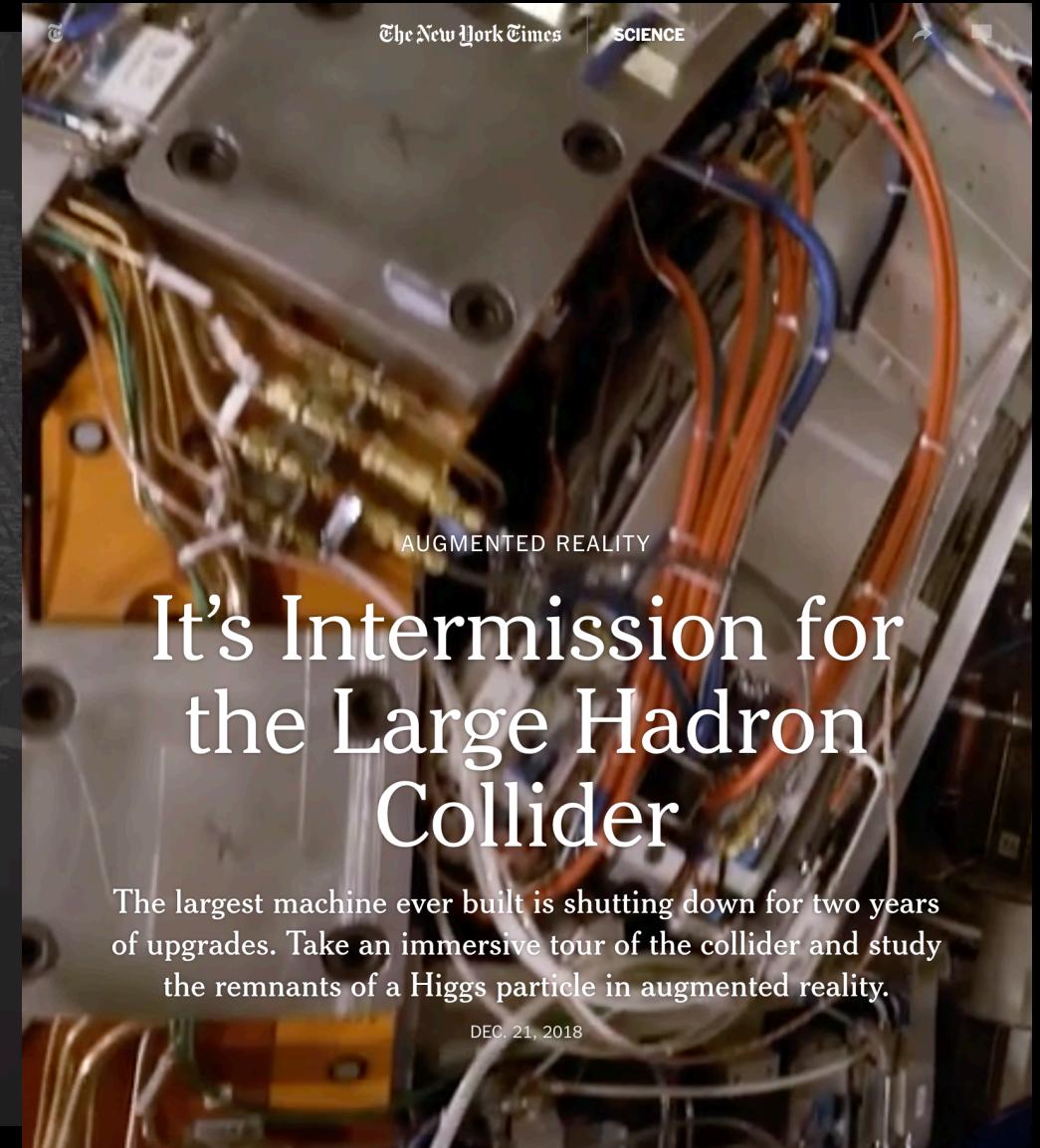
The New York Times | ARTS

Hudson Yards Is Manhattan's Biggest, Newest, Slickest Gated Community.

## Is This the Neighborhood New York Deserves?

By Michael Kimmelman  
Architecture critic, The New York Times  
March 14, 2019

186



# About this workshop

- We're going to build an AR Stickers app using ARCore and Sceneform.
- We'll cover AR fundamentals, drawing 3D shapes, importing and editing 3D models, user interactions, augmented faces, and cloud anchors.

# About this workshop

- These slides are linked from the README (slides.pdf)
- There's also a raw version where you can grab code snippets (slides.md)

# About ARCore

- Google released ARCore in February 2018.
- ARCore helps devs build apps that can understand the environment around a device and place objects and information in it.

# About Sceneform

- Google followed up with Sceneform at I/O 2018.
- Sceneform helps devs render 3D scenes on Android without needing to learn OpenGL.

# Drawing a 3D cube before Sceneform



# Drawing a 3D cube before Sceneform

# Drawing a 3D cube before Sceneform

```
public void draw( float[] mvpMatrix ) {  
    GLES20.glUseProgram(mProgram);  
    GLES20.glEnableVertexAttribArray(mPositionHandle);  
    GLES20 glVertexAttribPointer(  
        mPositionHandle, 3, GLES20.GL_FLOAT, false, VERTEX_STRIDE, mVertexBuffer);  
    GLES20.glEnableVertexAttribArray(mColorHandle);  
    GLES20 glVertexAttribPointer(  
        mColorHandle, 4, GLES20.GL_FLOAT, false, COLOR_STRIDE, mColorBuffer);  
    GLES20 glUniformMatrix4fv(mMVPMatrixHandle, 1, false, mvpMatrix, 0);  
    GLES20.glDrawElements(  
        GLES20.GL_TRIANGLES, INDICES.length, GLES20.GL_UNSIGNED_BYTE, mIndexBuffer);  
    GLES20.glDisableVertexAttribArray(mPositionHandle);  
    GLES20.glDisableVertexAttribArray(mColorHandle);  
}
```

# Drawing a 3D cube with Sceneform

```
val size = Vector3(1.0f, 1.0f, 1.0f)
val position = Vector3(0.0f, 0.0f, 0.0f)

MaterialFactory.makeOpaqueWithColor(this, Color(Color.RED))
    .thenAccept(material -> ShapeFactory.makeCube(size, position, material))
```

**Let's get started**

# Add the dependency

In app/build.gradle

```
dependencies {  
    // ...  
  
    implementation "com.google.ar.sceneform.ux:sceneform-ux:1.8.0"  
}
```

# Configure the manifest

Setup camera in the <manifest> section

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera.ar" android:required="true" />
```

In the <application> section, add a Play Store filter for users on devices that are not supported by ARCore.

```
<meta-data android:name="com.google.ar.core" android:value="required" />
```

# Add the ARFragment

In content\_main.xml

```
<fragment  
    android:id="@+id/fragment"  
    android:name="com.google.ar.sceneform.ux.ArFragment"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

**Run it!**

# Emulator troubleshooting

- Check that your Android Emulator is updated to 27.2.9 or later.
- Follow the instructions linked in the README.

# **AR Fundamentals**

# Feature points

**feature points** are visually distinct features that ARCore detects in each captured camera image.



# Motion Tracking

- ARCore detects visually distinct **feature points** in each captured camera image.
- It uses these points to compute a device's change in location over time.

# Motion tracking

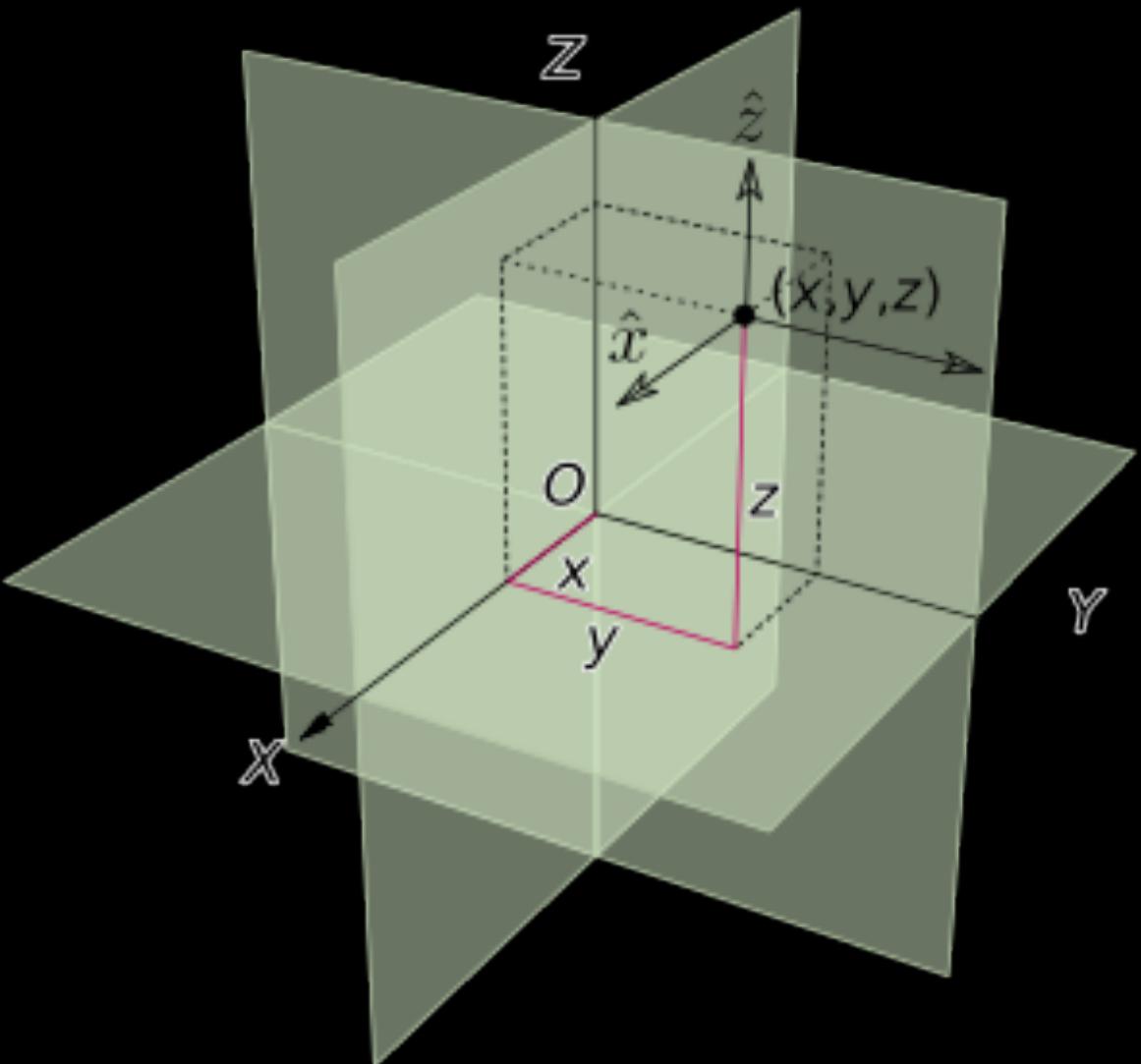
Visual feature point information is combined with measurements from the device's Inertial Measurement Unit (IMU).

# Motion tracking

This combined data is used to estimate the **pose**, defined by **position** and **orientation**, of the device camera relative to the world over time.

# World space

- World space is the 3D coordinate space in which the camera and other objects are positioned.
- Three fixed axes:  $x$ ,  $y$ ,  $z$ .
- The positions of the camera and other objects are updated from frame to frame as they move within the space.



# Pose

Everything in an AR scene has a **pose** within the world space.

Each **pose** is composed of:

- x-axis translation
- y-axis translation
- z-axis translation
- rotation

# Pose

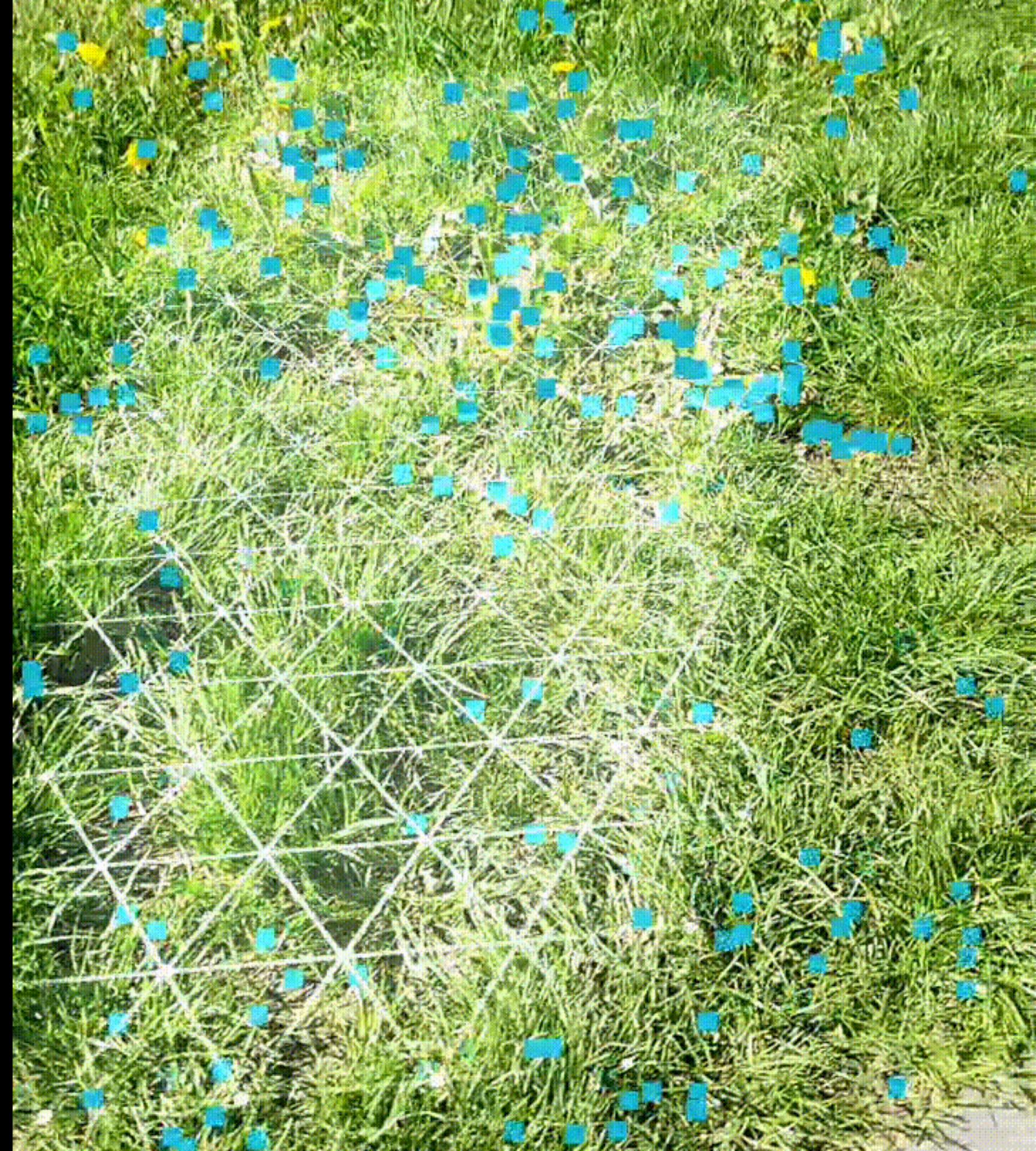
Sceneform aligns the pose of the **virtual camera** that renders your 3D content with the pose of the device's camera provided by ARCore.

# Pose

Because the rendered virtual content is overlaid and aligned on top of the camera image, it appears as if your virtual content is part of the real world.

# Plane detection

ARCore looks for clusters of feature points that appear to lie on common horizontal or vertical surfaces and provides this data to Sceneform as **planes**.



# Plane detection

A **plane** is composed of:

- A center **pose**
- An **extent** along each axis
- A **polygon**, a collection of 2D vertices approximating the detected plane.

# Plane detection

The Sceneform fragment renders a plane-grid to indicate via the UI where these planes exist.



# **Goal:**

**When a user taps a point on the screen that intersects with a plane, we want to place an object at that point.**

# Hit test

We want to determine if a **ray** extending from the **x, y** coordinate of our tap intersects a plane.

If it does, we'll place an **anchor** at the **pose of the intersection**.

# Hit test

In MainActivity, set up the ArFragment with an OnTapArPlaneListener.

```
private lateinit var arFragment: ArFragment

override fun onCreate(savedInstanceState: Bundle?) {
    //...
    arFragment = fragment as ArFragment
    arFragment.setOnTapArPlaneListener { hitResult, plane, motionEvent ->
        // We've hit a plane!
    }
}
```

# Hit Result

Represents an *intersection* between a **ray** and estimated real-world **geometry** (e.g. a Node or a Plane).

We can use a HitResult to determine the Pose of the intersection, the distance from the camera, or to create a new Anchor at the pose of intersection.

# Shapes

Let's use Sceneform's ShapeFactory API to create a sphere.

# Shapes

Create a new function in MainActivity:

```
private fun addSphere(color: Int, anchor: Anchor, radius: Float, centerX: Float, centerY: Float, centerZ: Float) {  
    MaterialFactory.makeOpaqueWithColor(this, com.google.ar.sceneform.rendering.Color(color))  
    .thenAccept { material ->  
        val shape = ShapeFactory.makeSphere(radius, Vector3(centerX, centerY, centerZ), material)  
        addNodeToScene(anchor, shape)  
    }  
}
```

# Nodes

All of the virtual content in a AR experience is organized as a **scene graph**.

A **scene graph** is basically an n-tree, made up of **nodes** which can each have 0...n children.

# Nodes

We need a way to add our sphere to the **scene**. We'll do that by creating a Node, attached to an Anchor at the point of intersection.

# Nodes

Create a new function in MainActivity:

```
private fun addNodeToScene(anchor: Anchor, renderable: Renderable) {  
    val anchorNode = AnchorNode(anchor)  
    val node = TransformableNode(arFragment.transformationSystem)  
    node.renderable = renderable  
    node.setParent(anchorNode)  
    arFragment.arSceneView.scene.addChild(anchorNode)  
}
```

# Anchors

Now we want to create an **anchor** based on the **hit result**, so that we can anchor our node to the pose on the plane we tap.

# Anchors

Create an anchor based on the HitResult, and use it to add a sphere to the scene.

```
arFragment.setOnTapArPlaneListener { hitResult, plane, motionEvent ->
    val anchor = hitResult.createAnchor()
    addSphere(Color.RED, anchor, 0.1f, 0.0f, 0.15f, 0.0f)
}
```

**Run it!**

# Sceneform plugin

Sceneform has an Android Studio plugin for importing, editing, and previewing 3D models.

# Sceneform plugin: installation

Android Studio > Preferences > Plugins > browse repositories  
Google Sceneform Tools (Beta)

# Supported formats

Look in app/sampleddata/models. We've provided some models.

- .obj - encodes the 3D geometry of the model (e.g. vertices, polygon faces)
- .mtl - material referenced by the .obj, describes the surface of the model (e.g. color, texture, reflection)
- .png - optional visual texture referenced by the .mtl to be mapped onto the surface of the model

# Supported formats

In addition to Wavefront obj, Sceneform also supports importing:

- FBX, with or without animations
- glTF (animations not supported)

# Sceneform assets

1. Select app/sampleddata/models/coffee.obj and then right mouse click to get the menu.
2. Pick New > Sceneform asset.
3. Click Finish.

# Sceneform assets

We've now converted into Sceneform's .sfa and .sfb formats.

- .sfb - Sceneform Binary, points to the models, material definitions, and textures in the source asset.
- .sfa - Sceneform Asset Definition, a human-readable description of the .sfb

# **Import the rest of the models**

Take a few minutes to import the three remaining models:  
`pasta.obj`, `pizza.obj`, and `tiramisu.obj`

**Goal: When a user  
taps, we want to  
place the selected  
object from the  
gallery at that point.**

# Renderables

A *renderable* is an object that can be attached to a *node* to render in 3D space.

# Loading renderables from model assets

Make a new function in `MainActivity` to load the renderable from it's URI and attach it at the anchor.

```
private fun placeObject(anchor: Anchor, model: Uri) {
    ModelRenderable.builder()
        .setSource(fragment.context, model)
        .build()
        .thenAccept { renderable -> addNodeToScene(anchor, renderable) }
        .exceptionally { throwable ->
            Toast.makeText(this@MainActivity, "Something went wrong!", Toast.LENGTH_SHORT).show()
            null
        }
}
```

# Update the tap listener

Get the URI from the selected gallery item and pass it, along with the anchor, to placeObject.

```
arFragment.setOnTapArPlaneListener { hitResult, plane, motionEvent ->
    val anchor = hitResult.createAnchor()
    val uri = galleryAdapter.getSelected().getUri()
    placeObject(anchor, uri)
}
```

**Run it!**

# Google Poly

Thousands of open source 3D models can be found at  
[poly.google.com](https://poly.google.com)

# Adjusting Scale

In coffee.sfa

```
model: {  
    attributes: [  
        'Position',  
        'Orientation',  
    ],  
    collision: {},  
    file: 'sampledata/models/coffee.obj',  
    name: 'coffee',  
    recenter: 'root',  
    scale: 0.50,  
},
```

**Run it!**

# Snap a photo

We can take a photo of our AR scene, including the virtual content, by capturing the `SurfaceView` (the class that `ArSceneView` descends from).

# Snap a photo

The app already includes a CameraHelper class. Let's wire it up so that when the button is clicked, we take a photo.

```
private lateinit var camera: CameraHelper

override fun onCreate(savedInstanceState: Bundle?) {
    // ...

    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.WRITE_EXTERNAL_STORAGE), 0)
    }

    camera = CameraHelper(this, arFragment.arSceneView)
    fab.setOnClickListener { camera.snap() }
}
```

**Run it!**

**Now for the fun  
stuff...**

# **Augmented Faces**

# Augmented Faces

Let's extend our stickers app to experiment with Sceneform's AugmentedFaces API.

# Augmented Faces

- The Augmented Faces API helps you to identify different regions of a detected face.
- You can use those regions to anchor nodes and renderables so that they move with the face.

# Augmented Faces

When a user's face is detected by the camera, ARCore detects:

- The **center pose**: the physical center point of the user's head, inside the skull directly behind the nose.
- The **face mesh**: the hundreds of vertices that make up the face, defined relative to the center pose.
- The **face regions**: three distinct poses on the user's face (left forehead, right forehead, nose tip).

# Augmented Faces

These elements are used by Augmented Faces APIs as regions to align 3D assets to the face.

# Extend the ArFragment

Create a new class, FaceArFragment. Override  
getSessionConfiguration to enable augmented face mode.

```
class FaceArFragment : ArFragment() {  
  
    override fun getSessionConfiguration(session: Session): Config {  
        val config = Config(session)  
        config.augmentedFaceMode = Config.AugmentedFaceMode.MESH3D  
        return config  
    }  
  
}
```

# Extend the ArFragment

Configure the session to use the front-facing camera.

```
class FaceArFragment : ArFragment() {  
    ...  
  
    override fun getSessionFeatures() = EnumSet.of<Session.Feature>(Session.Feature.FRONT_CAMERA)  
}
```

# Extend the ArFragment

Turn off the plane discovery controller, since plane detection doesn't work with the front-facing camera.

```
override fun onCreateView(inflater: LayoutInflater,  
    @Nullable container: ViewGroup?,  
    @Nullable savedInstanceState: Bundle?): View? {  
    val frameLayout = super.onCreateView(inflater, container, savedInstanceState) as FrameLayout?  
  
    planeDiscoveryController.hide()  
    planeDiscoveryController.setInstructionView(null)  
  
    return frameLayout  
}
```

# Add the fragment to the layout

In content\_faces.xml

```
<fragment  
    android:id="@+id/fragment"  
    android:name="com.nytimes.android.ramonaharrison.FaceArFragment"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent" />
```

# **Import the face model**

From the `samp1edata` directory, import `fox_face.fbx` as a Sceneform asset.

# Load the face model as a renderable

In `FacesActivity`, create a function to load the imported asset as a renderable.

```
private fun loadFaceRenderable() {  
    ModelRenderable.builder()  
        .setSource(this, R.raw.fox_face)  
        .build()  
        .thenAccept { modelRenderable: ModelRenderable ->  
            modelRenderable.isShadowCaster = false  
            modelRenderable.isShadowReceiver = false  
            faceRegionsRenderable = modelRenderable  
        }  
}
```

# Load the face texture

The project also include a texture that we'll superimpose over the entire face.

```
private fun loadFaceTexture() {  
    Texture.builder()  
        .setSource(this, R.drawable.fox_face_mesh_texture)  
        .build()  
        .thenAccept { texture: Texture ->  
            faceMeshTexture = texture  
        })  
}
```

# Setup the scene

In `FacesActivity`, set up the fragment.

```
private lateinit var arFragment: ArFragment

override fun onCreate(savedInstanceState: Bundle?) {
    // ...

    arFragment = fragment as ArFragment
    val sceneView = arFragment.arSceneView
    sceneView.cameraStreamRenderPriority = Renderable.RENDER_PRIORITY_FIRST
}
```

# Update loop

Add an `OnUpdateListener` to the scene.

```
val scene = sceneView.scene
scene.addOnUpdateListener { frameTime: FrameTime ->
    if (faceRegionsRenderable != null && faceMeshTexture != null) {
        // Attach nodes for tracked faces and remove untracked faces
    }
}
```

# Handle tracked faces

In `FacesActivity`, create a `faceNodeMap` member variable.

```
class FacesActivity : AppCompatActivity() {  
  
    val faceNodeMap = HashMap<AugmentedFace, AugmentedFaceNode>()  
  
    // ...  
}
```

# Handle tracked faces

Create a function handleTrackedFaces.

```
private fun handleTrackedFaces(sceneView: ArSceneView, scene: Scene) {  
    val faceList = sceneView.session?.getAllTrackables(AugmentedFace::class.java) ?: emptyList()  
    for (face in faceList) {  
        if (!faceNodeMap.containsKey(face)) {  
            val faceNode = AugmentedFaceNode(face)  
            faceNode.setParent(scene)  
            faceNode.faceRegionsRenderable = faceRegionsRenderable  
            faceNode.faceMeshTexture = faceMeshTexture  
            faceNodeMap[face] = faceNode  
        }  
    }  
}
```

# Handle untracked faces

We also want to remove faces that have disappeared from the current frame.

```
private fun handleUntrackedFaces() {  
    val iterator = faceNodeMap.entries.iterator()  
    while (iterator.hasNext()) {  
        val entry = iterator.next()  
        val face = entry.key  
        if (face.trackingState == TrackingState.STOPPED) {  
            val faceNode = entry.value  
            faceNode.setParent(null)  
            iterator.remove()  
        }  
    }  
}
```

# Update the update loop

Invoke these two new functions from the update loop.

```
val scene = sceneView.scene
scene.addOnUpdateListener { frameTime: FrameTime ->
    if (faceRegionsRenderable != null && faceMeshTexture != null) {
        handleTrackedFaces(sceneView, scene)
        handleUntrackedFaces()
    }
}
```

**Run it!**

# Cloud anchors

# Cloud anchors

Cloud Anchors let you share ARCore (or ARKit!) anchors across multiple devices to create a shared AR experience.

# Cloud anchors

ARCore uses Google servers to host and resolve anchors.

**Goal: Create an  
augmented sculpture  
garden**

**Grazie!**