
DEVELOPING APPS WITH DAYDREAM AND PROJECT TANGO
shopify

Today, our **first** ambitious goal will be to build a VR scene**.** We'll be using the Google-VR platform.

The Google-VR platform is built to support both Daydream and Cardboard. This allows developers to target **both** platforms at once. This is mostly an exercise in graceful degradation.

You can find all the docs and sample code for the Google VR SDK at [developers.google.com/vr]

Our **second** ambitious goal will be to create a Tango application. You can think of Tango as an Augmented Reality platform. But really, it's more than that. Tango lets devices understand **where** they are in the world, and detect **what** is around them. It does this by leveraging computer vision and specialty sensors.

[TODO Animate in some visuals for these 2 concepts] Our application will use Tango's Motion Tracking and Depth Perception APIs to populate a real space with virtual objects.

We'll be using Java and the native Android toolset to achieve these two goals. There's a reason behind this. I believe in the potential of VR and AR for things other than video games. I want to make VR and AR accessible to Application developers.

Unity and Unreal are typically the go-to platforms for VR content creation. And with good reason! When you need to import 3D models, animate them, simulate physics, edit scenes, and so on, Unity and Unreal are hard to beat.

The only problem is, these platforms are **Game engines**. And as **App developers**, we want access to the awesome APIs and libraries available on the native Android platform. With Unity and Unreal, you effectively lose that. So, how do we otherwise do 3D on mobile?
[NEXT: OPENGL]

OpenGLES is the **standard** when doing native mobile development. And in the last few years, it's gotten a **lot better**. We're starting to see features that used to only be available on the desktop. The **problem** is, OpenGLES is a focused **graphics** library. It's then **your** job to figure out a way to import your 3d models, animate them, detect collisions, simulate physics. None of these things are covered with OpenGL.

To top it all off, the learning curve with OpenGL is pretty steep, and it's an **unforgiving** library. If you set things up improperly, at best you'll get a blank screen, at worst your app will crash with a cryptic error message. Once in a while, you won't even get that error message. OpenGL is a big time-sink.

The solution is a library called Rajawali. It’s a lesser-known Android Java library, built on top of OpenGL. It provides support for building complex 3D scenes. It can also load models from various 3D file formats, helps with animations, and has a slew of other nice features. And since it’s built on top of OpenGL, it still allows you to interact with the OpenGL pipeline when you need to.

How to Build a VR App™

The 10 minute edition

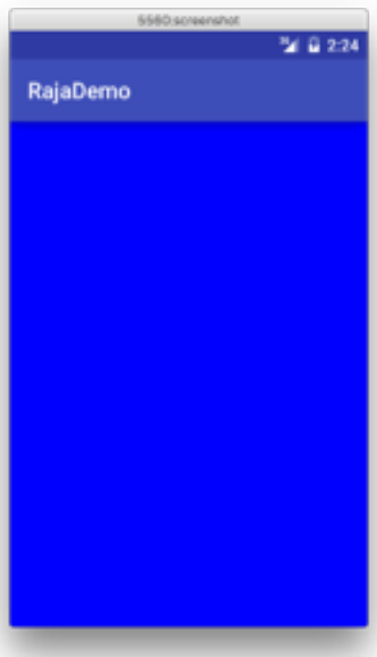
https://github.com/kanawish/rajadaydream

Now, let’s start on our **first** ambitious goal. We do have a problem here. Rajawali's current VR support is based off of the Cardboard-era SDKs, **[CLICK :(]** and with the introduction of Daydream, **breaking** changes have been introduced.

The good news is, **[CLICK :)]** I have you covered. A modified version of the Rajawali VR classes, alongside the sample code for this presentation, can be found on the Github URL below. Now, let’s see "How to build a VR App [tm] - **[CLICK]** The 10 minute edition".

SimpleRenderer
SimpleRenderer(Context, BaselineRenderingFragment)
initScene(): void ↑Renderer
onRender(long, double): void ↑Renderer

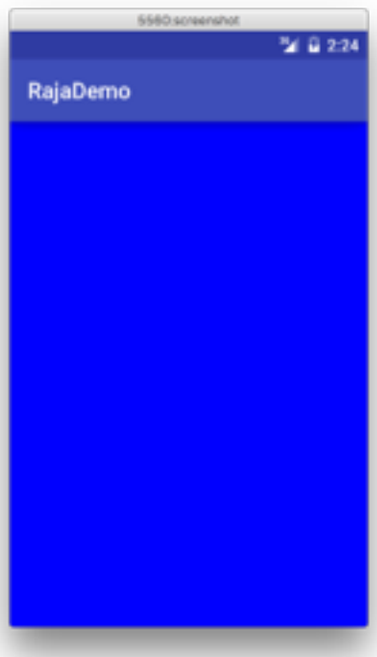
```
protected void initScene() {

        getCurrentScene().setBackgroundColor(Color.BLUE);
```



[TODO: code comments/highlights]

SimpleRenderer
SimpleRenderer(Context, BaselineRenderingFragment)
initScene(): void ↑Renderer
onRender(long, double): void ↑Renderer

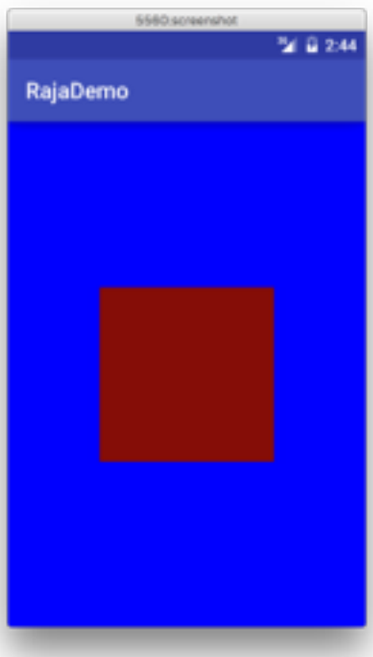
```
// Lights,
DirectionalLight key = new
DirectionalLight(0,0.1,0.2);
key.setPosition(0.0,10.0,0.5);
key.enableLookAt();
key.setPower(2);
getCurrentScene().addLight(key);
```



[TODO: code comments/highlights]

SimpleRenderer
SimpleRenderer(Context, BaselineRenderingFragment)
initScene(): void ↑Renderer
onRender(long, double): void ↑Renderer

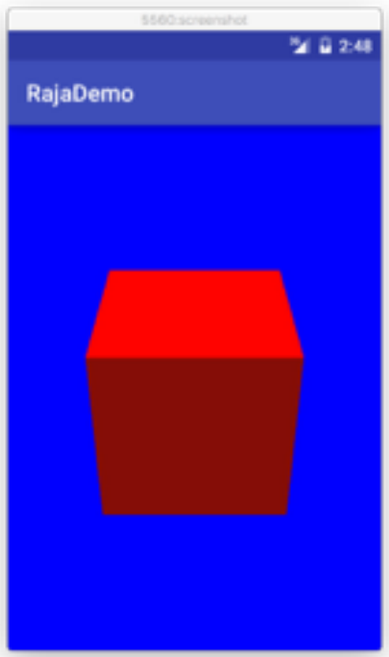
```
// Models
Material material = new Material();
material.setDiffuseMethod(new
DiffuseMethod.Lambert());
material.enableLighting(true);
Cube cube = new Cube(1);
cube.setMaterial(material);
```



[TODO: code comments/highlights]

SimpleRenderer
SimpleRenderer(Context, BaselineRenderingFragment)
initScene(): void ↑Renderer
onRender(long, double): void ↑Renderer

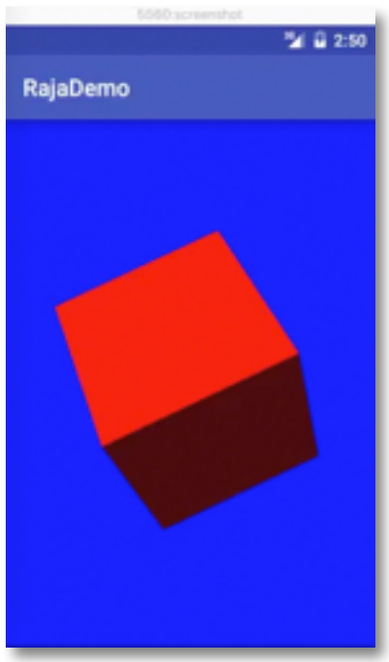
```
// Camera,
getCurrentCamera().setPosition(0, 2, 3);
getCurrentCamera().setLookAt(0, 0, 0);
```



[TODO: code comments/highlights]

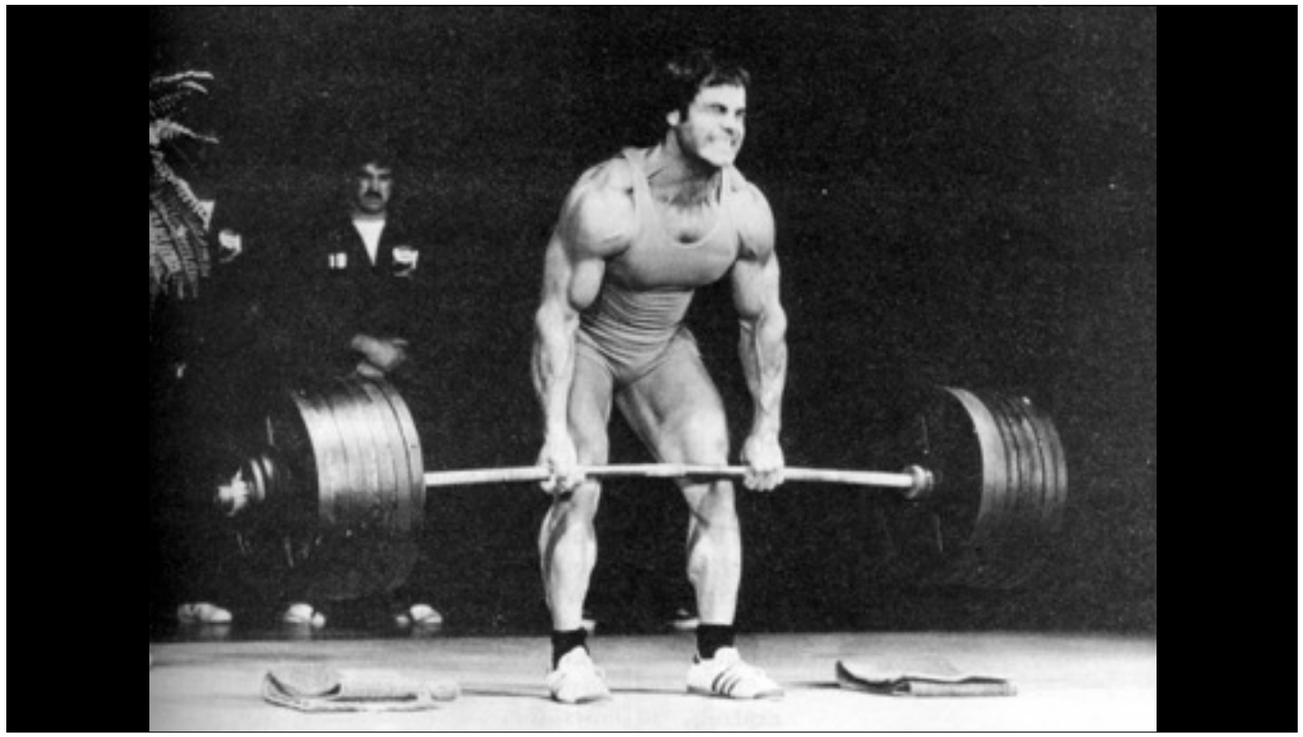
SimpleRenderer
SimpleRenderer(Context, BaselineRenderingFragment)
initScene(): void ↑Renderer
onRender(long, double): void ↑Renderer

```
// Action!
RotateOnAxisAnimation anim = new
RotateOnAxisAnimation(new Vector3(1,1,1), 360);
anim.setTransformable3D(cube);
anim.setDurationMilliseconds(20000);
anim.setRepeatMode(Animation.RepeatMode.INFINITE);
getCurrentScene().registerAnimation(anim);
anim.play();
```



[TODO: code comments/highlights]

Ok, so this is nice… but …



You might be wondering, how much effort is required, in order to turn this little scene into a VR example? Let’s find out.

```java
public class RajaVrDemoActivity extends GvrActivity {
    /** Sets the view to our GvrView and initializes GvrView */
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Initialize Gvr
        setContentView(R.layout.common_ui);

        RajaVrView gvrView = (RajaVrView) findViewById(R.id.gvr_view);

        gvrView.setEGLConfigChooser(8, 8, 8, 8, 16, 8);
        gvrView.setRenderer(new DemoVRRenderer(this));
        gvrView.setTransitionViewEnabled(true);
        setGvrView(gvrView);
    }
```

https://github.com/kanawish/rajadaydream

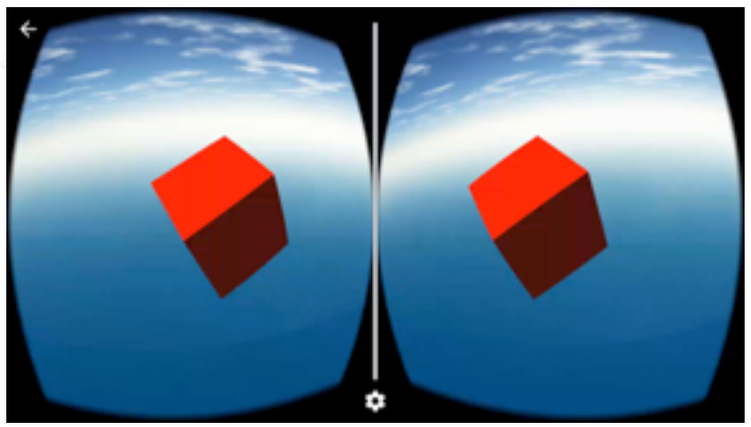
If you're new to Android, I need need to tell you about two basic 'building blocks' of Android Apps: Activities and Views.

If you think of an Android Applications as a Web apps, an Activity is basically one 'page' in your Application. Here is part of the Activity code for our VR example. We'll first need to **extend** `GvrActivity`, the modified Rajawali code found in my github repo.

Then, when creating the Activity, we'll need a View widget as a display. RajaVrView extends Google-VR's `GvrView`. It provides the hooks we need to get Rajawali and Google-VR working together. The setup code is otherwise pretty minimal.

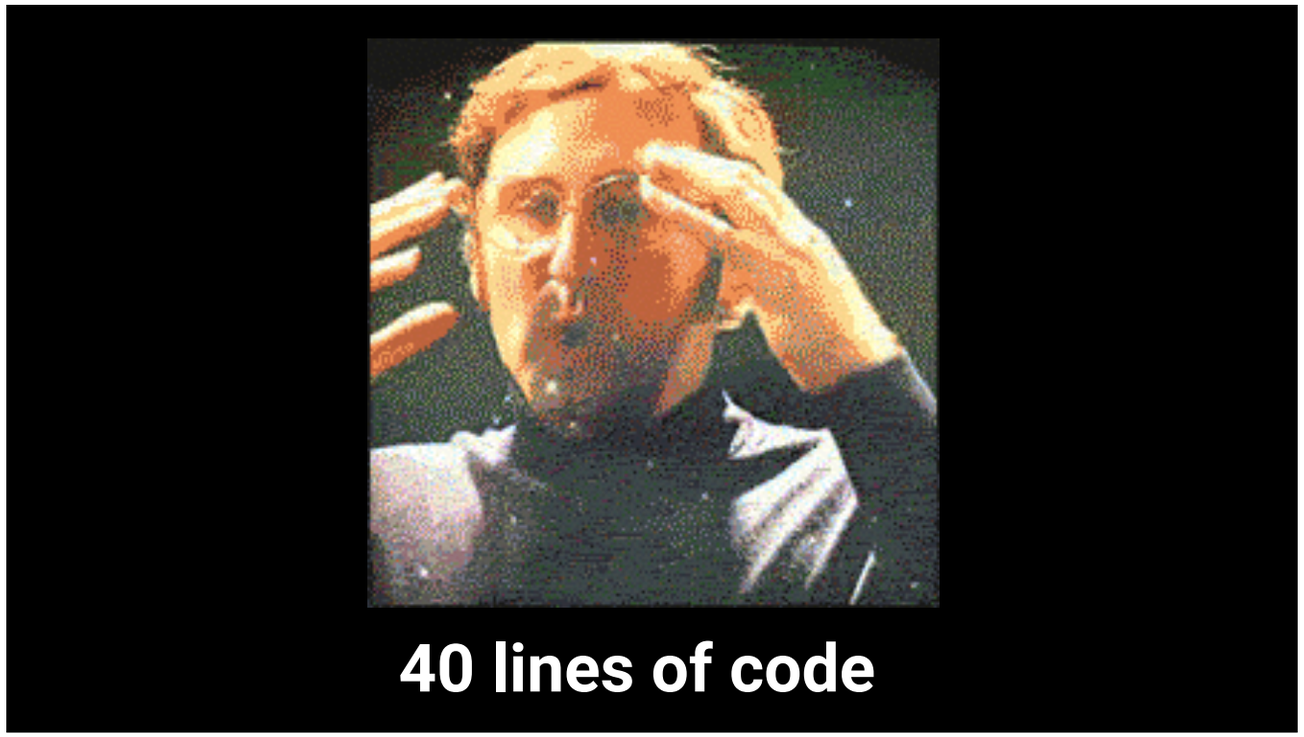
The key parts are: where we hook up a renderer to the `GvrView`, and where we assign the `GvrView` instance to our `GvrActivity`

```
SimpleRenderer
    SimpleRenderer(Context, BaselineRenderingFragment)
    initScene(): void ↑Renderer
    onRender(long, double): void ↑Renderer

// Camera,
getCurrentCamera().setPosition(0, 2, 3);
getCurrentCamera().setLookAt(0, 0, 0);

// Models
Material material = new Materi
material.setDiffuseMethod(new
DiffuseMethod.Lambert());
material.enableLighting(true);
Cube cube = new Cube(1);
cube.setMaterial(material);
cube.setPosition(0, -1, -3);
```



And, we're basically done. Couple things to note. First, the camera arrangement is no longer relevant. In our VR scenes, the headset coordinates **are** the camera. The headset is positioned at the origin, (0,0,0), and it's orientation tracks the orientation of the headset itself.

Our cube was automatically placed at the origin so far in our example. So we'll **also** need to move it **in front** of the headset.



Well, we’re done! We have a working VR app. Ignoring the usual Android boilerplate, I counted 40 lines of code in this sample.

And this is really thanks to Rajawali, isolating us from nasty OpenGL setup code.

So, we've now covered the basics of rendering 3D objects, and we've seen how quickly a 3D scene can be turned into a VR experience just by dropping that scene into a Rajawali/Daydream wrapper.

Let's move on to our second ambitious goal, building a small Tango app.

Perhaps you're wondering, *why* make a Tango app to start with? The first Tango consumer device will be the Lenovo Phab 2 Pro. This phone should launch before the end of the year.

Hard to tell for now if it's going to be a successful device. When you walk out of this session, should you try to develop a mass-market app for Tango devices?

You probably shouldn't. At least, not yet. For now, this is a _**specialty**_ device. Let's rule out ideas that depend on mass market adoption. Think of consultancy type scenarios, where end-users will be provided with a Tango devices.

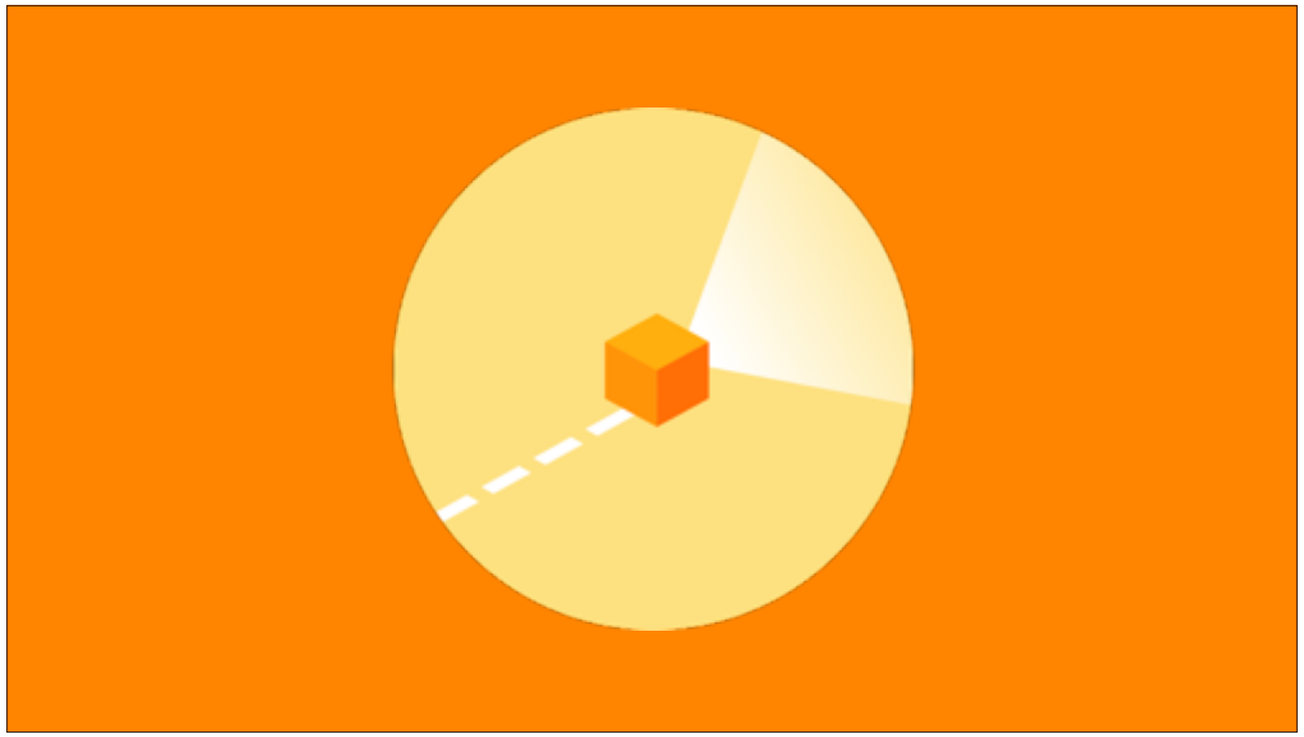
Use case ideas

- Real estate (scanning indoor spaces)
- Museum exhibits (interactive guides)
- Interior decoration (indoor mapping, product placement)
- In-store assistant and analytics (indoor location)
- Inventory management (precise indoor mapping)
- VR content production

On this slide you can see a quick list of potential early adoption use cases.
Some key concepts here are:
- Mapping rooms, volumes and shapes (Key in VR and AR content generation)
- Indoor location (Finding your way around)
- Providing rich interactions on-the-spot (Inventory management, Interactive guides)

Tango gives us three core technologies.

Motion tracking gives us the position and orientation of the device, in real-time*.
Area Learning allows a device to correct errors in motion tracking, and recognize spaces it has seen before. **[CLICK]** Today, we won't be looking at Area Learning.
Finally, **Depth Perception**, which lets the virtual world interact with the real world.



Motion tracking

Ok, first, Motion Tracking. Tango APIs are **great** to map out real world spaces. Let's see how to use the motion tracking and depth perception APIs.

Motion Tracking

[CONCEPT] So, what do we exactly track? The Tango APIs provide the **position** and **orientation** of a user's device in **full six degrees of freedom**. The combination of position and orientation is referred to as the device's **pose**.

```
tango.connectListener(framePairs, new OnTangoUpdateListener() {
    @Override
    public void onPoseAvailable(TangoPoseData pose) {
        // We could process pose data here, but we are not
        // directly using onPoseAvailable() for this app.
        logPose(pose);
    }

...
```

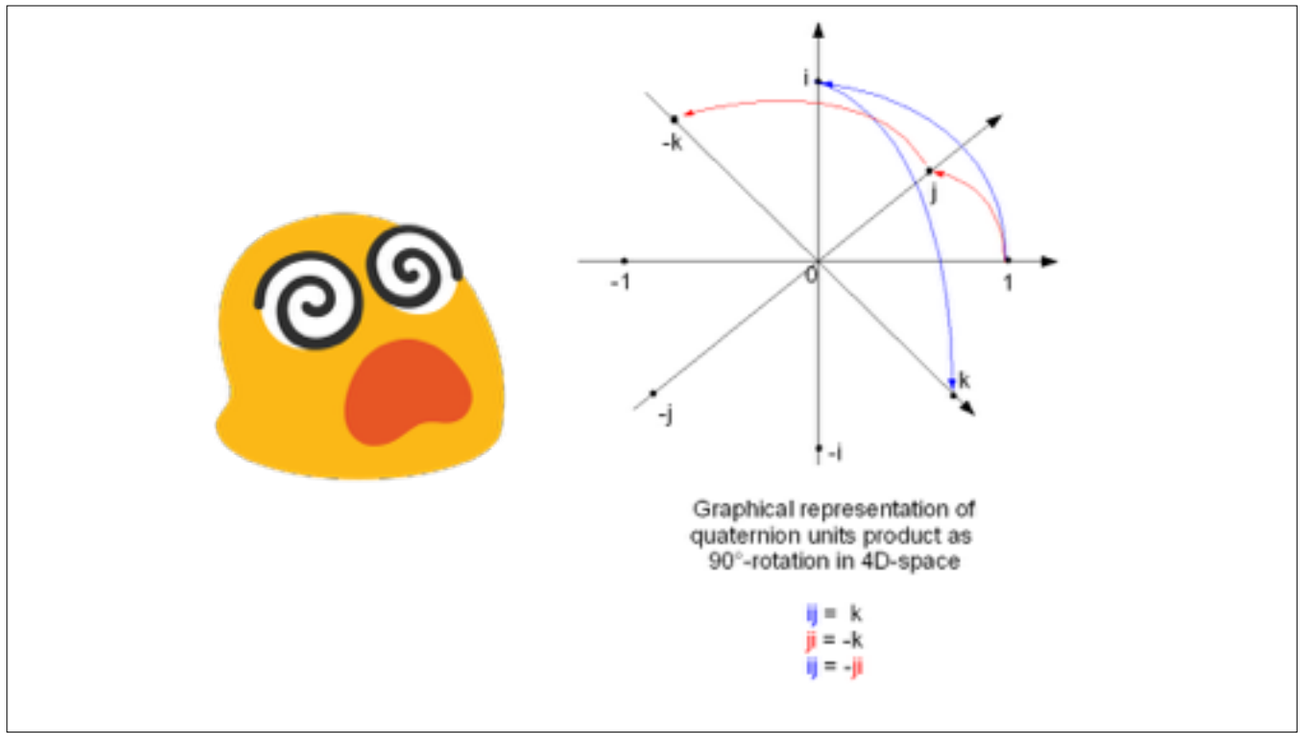
https://github.com/kanawish/rajadaydream

Now, we get Tango Pose data in one of two ways. We can do polling for it, but typically, we'll want to get our data via **callbacks**.

```
typedef struct TangoPoseData {
  int version;
  double timestamp;                 // In milliseconds
  double orientation[4];            // As a quaternion
  double translation[3];            // In meters
  TangoPoseStatusType status_code;
  TangoCoordinateFramePair frame;
  int confidence;                   // Currently unused
  float accuracy;                   // Currently unused
} TangoPoseData;
```

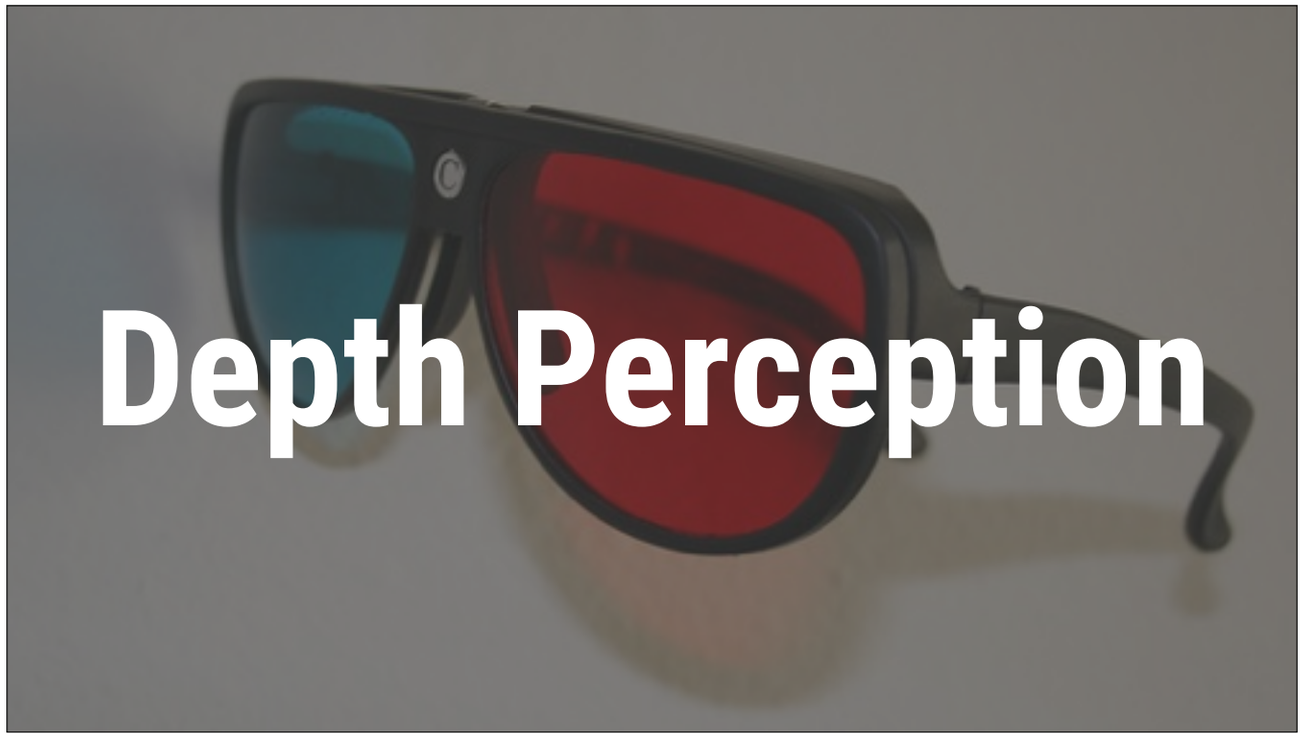
https://github.com/kanawish/rajadaydream

As your device moves through 3D space, it's position and orientation is calculated up to **100 times** per second. This is what you receive. For now, there are **two** important parts I'd like you to focus on. The position of the device is passed in the **translation** array, representing a 3d vector, and the **orientation** is stored as a quaternion.

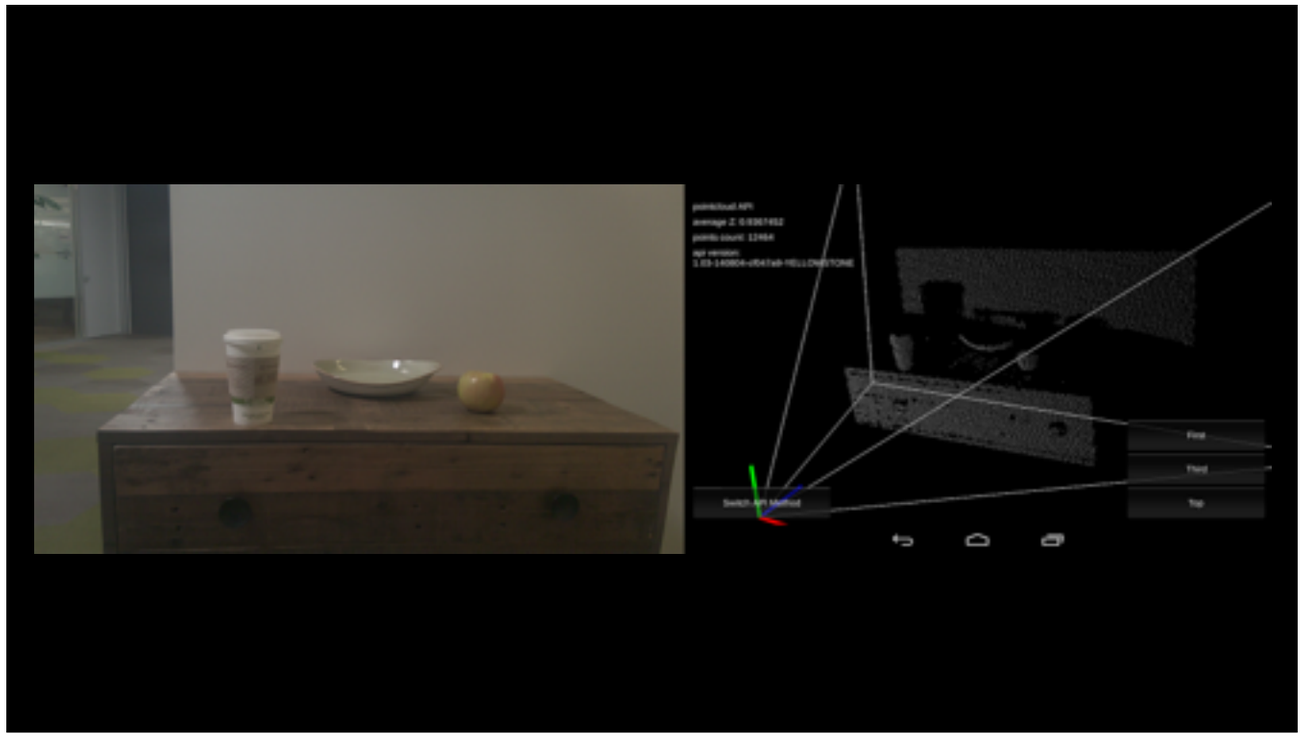


Okay, so what exactly *__is__* a quaternion? It's the quotient of 2 vectors. Quaternions fit in an array of 4 doubles. Instead of Quaternions, you *could* define orientation by specifying a rotation alongside the x, y and z axis. **But**, you'll run into a problem called Gimbal Lock, which makes it mathematically harder to define an orientation. Quaternions are not subject to Gimbal Lock, and are simpler to compose.

TLDR; Don’t worry too much about these, the APIs will do most of the heavy lifting. You just need to know what they represent.



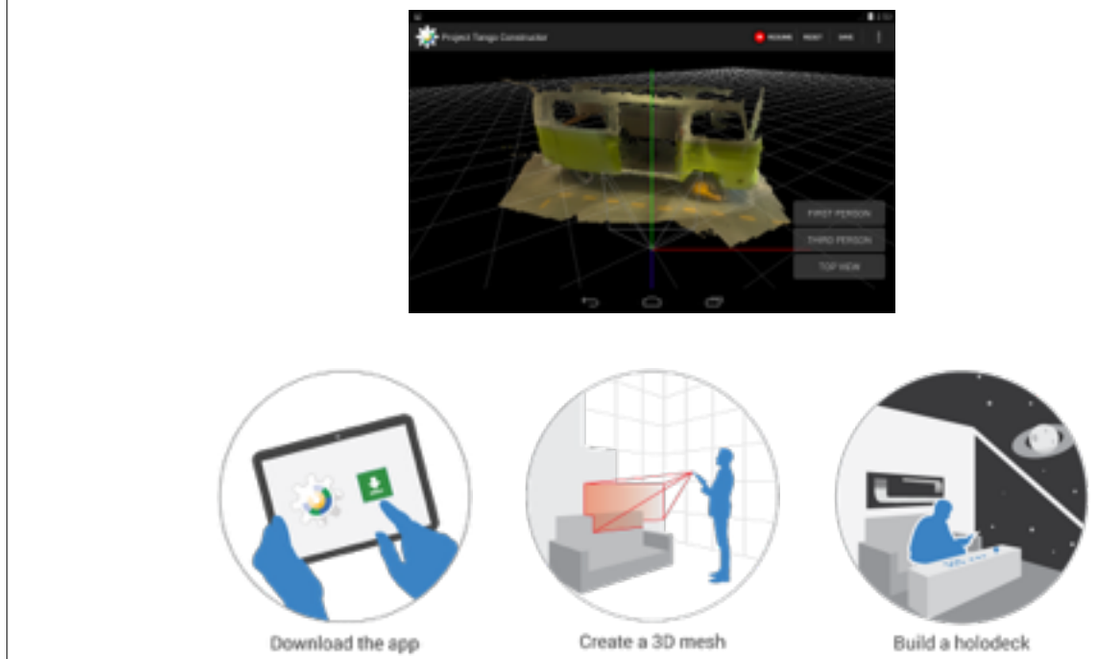
Thanks to **motion tracking**, we have an idea of **where** we are in the world. We also need to find out what exactly is out there, and this is where the **Depth Perception API** comes in. This API can tell us the **shape** of the world around us.



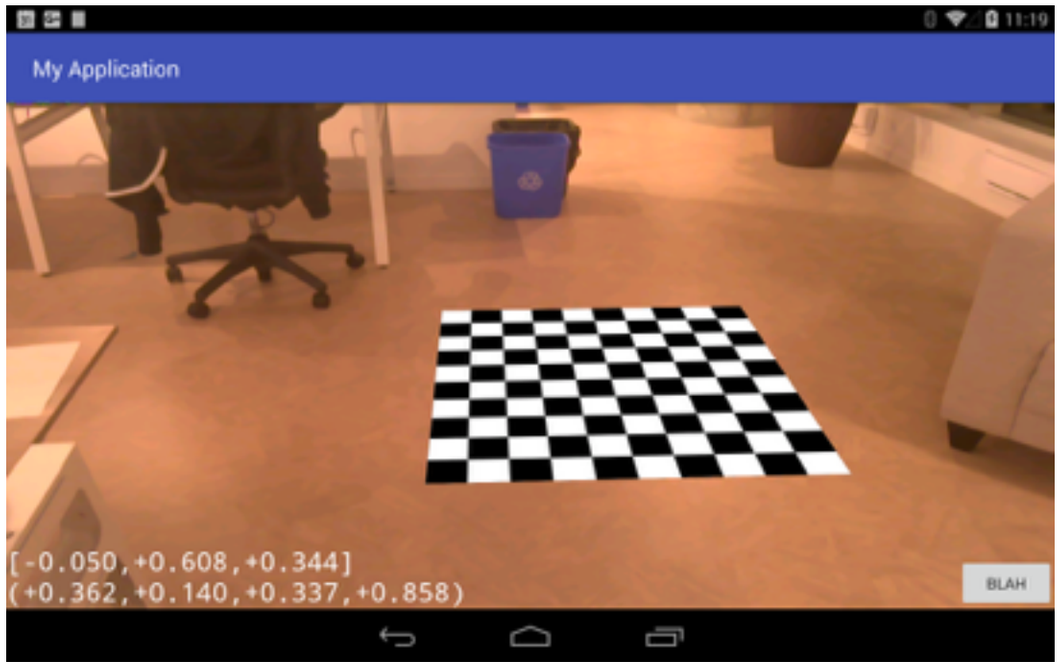
Tango Services provides depth data in the form of a **point cloud**. You can see what this point cloud looks like on the right.
This format gives (x, y, z) coordinates for as many points as possible.
One important thing to understand here is that these coordinates are **relative to the depth-sensing camera itself**. To understand what this means, the **camera viewport** is represented by this white pyramid, and the red/green/blue lines representing the **origin** of the camera.

```
tango.connectListener(framePairs, new OnTangoUpdateListener() {
    @Override
    public void onPoseAvailable(TangoPoseData pose) {
        // We could process pose data here, but we are not
        // directly using onPoseAvailable() for this app.
        logPose(pose);
    }

    @Override
    public void onXyzIjAvailable(TangoXyzIjData xyzIj) {
        // Save the cloud and point data for later use.
        tangoPointCloudManager.updateXyzIj(xyzIj);
    }

...
```

We get this point cloud data via callback, just like with the pose data seen earlier.



When we combine Depth Perception and Motion Tracking, we can start building a full picture of the space around us. If you've had the chance to play around with a Tango tablet, you might have seen the Tango Constructor application. It allows you to create detailed 3D models of a room and it's contents.



Our goal today is more modest. Let’s allow our users to place **virtual objects** in the room they're standing in. First, we’ll want to show a live feed of the camera on screen for users moving around a room. To put down a virtual object, users will tap on the desired surface, through that camera feed.

```
// Use default configuration for Tango Service, plus low latency IMU integration.
TangoConfig config = tango.getConfig(TangoConfig.CONFIG_TYPE_DEFAULT);
config.putBoolean(TangoConfig.KEY_BOOLEAN_DEPTH, true);
config.putBoolean(TangoConfig.KEY_BOOLEAN_COLORCAMERA, true);

// NOTE: Low latency integration is necessary to achieve a precise alignment of
// NOTE: virtual objects with the RBG image and produce a good AR effect.
config.putBoolean(TangoConfig.KEY_BOOLEAN_LOWLATENCYIMUINTEGRATION, true);

// NOTE: These are extra motion tracking flags.
config.putBoolean(TangoConfig.KEY_BOOLEAN_MOTIONTRACKING, true);
config.putBoolean(TangoConfig.KEY_BOOLEAN_AUTORECOVERY, true);

tango.connect(config);
```

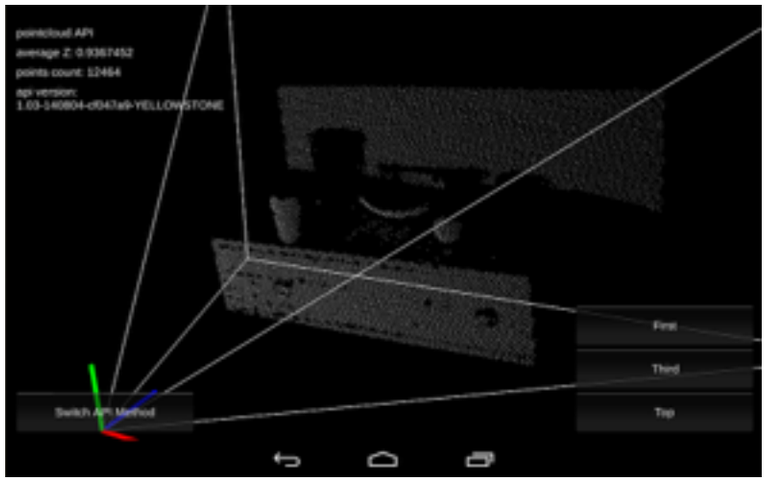
This is how we turn on the Tango services we need for our example. We've just touched on the Depth system that will provide us with a Point Cloud.
We'll also need a color camera feed and low latency support, to achieve precise alignment between virtual objects and the camera feed.
In orange we can see motion tracking configuration flags.

```
tango.connectListener(framePairs, new OnTangoUpdateListener() {

    public void onPoseAvailable(TangoPoseData pose) {
        // We could process pose data here, but we are not
        // directly using onPoseAvailable() for this app.
        logPose(pose);
    }

    public void onFrameAvailable(int cameraId) {
        // Check if the frame available is for the camera we want and update its frame on the view.
        if (cameraId == TangoCameraIntrinsics.TANGO_CAMERA_COLOR) {
            // Mark a camera frame is available for rendering in the OpenGL thread
            isFrameAvailableTangoThread.set(true);
            surfaceView.requestRender();
        }
    }

    public void onXyzIjAvailable(TangoXyzIjData xyzIj) {
        // Save the cloud and point data for later use.
        tangoPointCloudManager.updateXyzIj(xyzIj);
    }

...
```

Once that’s done, we’ll need set up our Tango update listener. **onFrameAvailable** is the new element. **[NEXT SLIDE]**

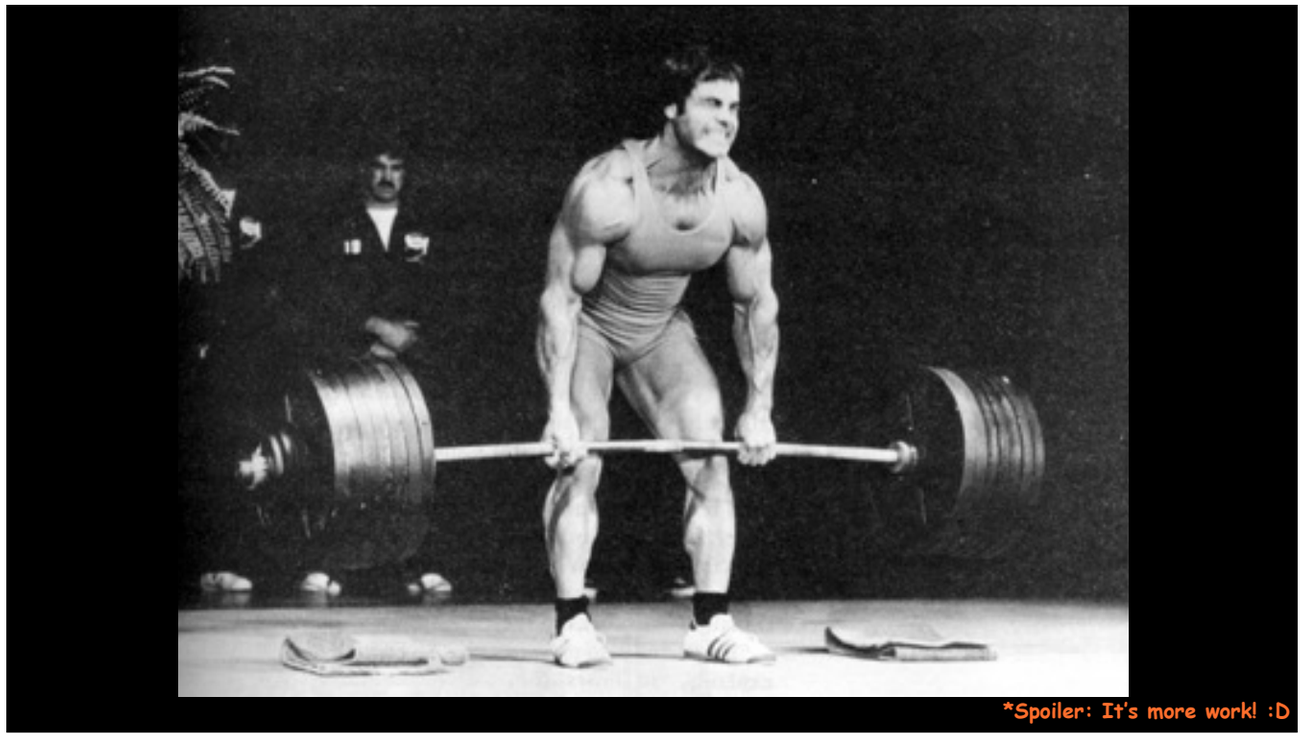
```
public void onFrameAvailable(int cameraId) {
    // Check if frame is for the right camera
    if (cameraId == TangoCameraIntrinsics.TANGO_CAMERA_COLOR) {
        // Mark a camera frame is available for rendering
        isFrameAvailableTangoThread.set(true);
        surfaceView.requestRender();
    }
}
```

`onFrameAvailable()` is triggered every time a new image is available from the RGB or Fisheye cameras. Here when we get frame data, we let our `**surfaceView**` know about it, and it will take care of updating itself.




```
public void onXyzIjAvailable(TangoXyzIjData xyzIj) {
    // Save the cloud and point data for later use.
    tangoPointCloudManager.updateXyzIj(xyzIj);
}
```

then, `**onXyzIjAvailable()**` gets called every time a new point cloud is available. There, we use a Tango support class called the `**TangoPointCloudManager**`, to store the information for future use.



You might be wondering, how much effort is required to get a **point in 3d space,** from a simple **tap on a screen**? Let's find out!

```
public boolean onTouch(View view, MotionEvent motionEvent) {
    if (motionEvent.getAction() == MotionEvent.ACTION_UP) {

        // Calculate click location in u,v (0;1) coordinates.
        float u = motionEvent.getX() / view.getWidth();
        float v = motionEvent.getY() / view.getHeight();

        try {
           float[] planeFitTransform;
            synchronized (this) {
                planeFitTransform = doFitPlane(u, v, rgbTimestampGlThread);
            }

            if (planeFitTransform != null) {
                // Update the position of the rendered cube
                // to the pose of the detected plane
                renderer.updateObjectPose(planeFitTransform);
            }

        } catch (TangoException t) {
            ...
```

When a user taps the screen, we call the `doFitPlane` method with a pair of **touch coordinates**, and a **timestamp**. The timestamp **is** important here. We'll get back to it.

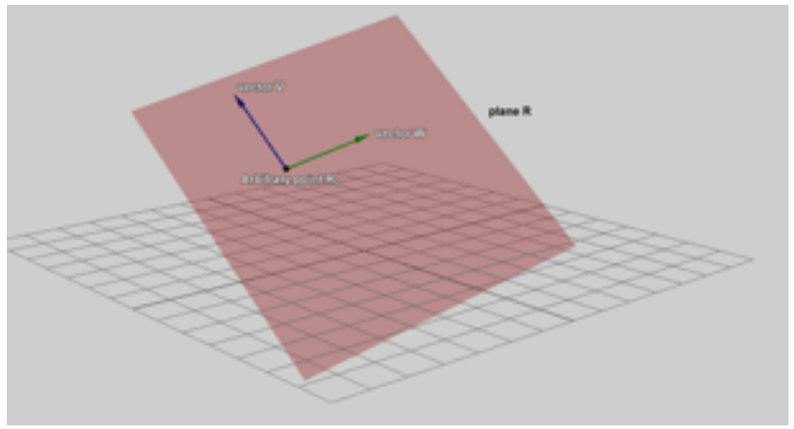
```
/**
 * Use the TangoSupport library with point cloud data to calculate the
plane
 * of the world feature pointed at the location the camera is looking.
 * It returns the transform of the fitted plane in a double array.
 */
private float[] doFitPlane(float u, float v, double rgbTimestamp) {
    TangoXyzIjData xyzIj = tangoPointCloudManager.getLatestXyzIj();

    if (xyzIj == null) {
        return null;
    }
...
```

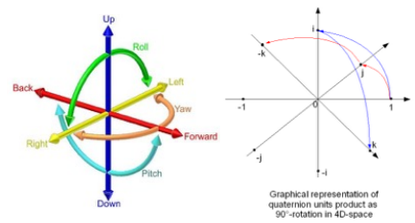
In doFitPlane, we first ask the cloudPointManager for the latest point cloud.

```
// We need to calculate the transform between the color camera at the
// time the user clicked, and the depth camera at the time the depth
// cloud was acquired.
TangoPoseData colorTdepthPose =
    TangoSupport.calculateRelativePose(
        rgbTimestamp, TangoPoseData.COORDINATE_FRAME_CAMERA_COLOR,
        xyzIj.timestamp, TangoPoseData.COORDINATE_FRAME_CAMERA_DEPTH);
```

Then, using the TangoSupport class, we get an instance of `TangoPoseData` calculated on our behalf. This relative pose tells us where the **color camera** is, **relative** to our **depth** information.

Note the rib camera timestamp being passed here. It's used by Tango, alongside the Cloud Point timestamp, to keep our calculated pose accurate.






```
// Perform plane fitting with the latest available point cloud data.
IntersectionPointPlaneModelPair intersectionPointPlaneModelPair =
    TangoSupport.fitPlaneModelNearClick(
        xyzIj, tangoCameraIntrinsics, colorTdepthPose, u, v);
```

`**fitPlaneModelNearClick**` gets us the **plane** at the **touch coordinates** we provided. That **plane** represents the spot we have clicked on screen, mapped to the real world.

There's a lot going on here, math wise. But thankfully, we barely have any work to do, except pass the right values along. **xyzIj** is our point cloud, **colorTdepthPose** is our relative camera pose, and the touch coordinates are **u** and **v**

You can ignore **cameraInstrinsics,** we get it from the tango system at initialization time.

```
// Get the transform from depth camera to OpenGL world at
// the timestamp of the cloud.
TangoMatrixTransformData transform =
    TangoSupport.getMatrixTransformAtTime(
        xyzIj.timestamp,
        TangoPoseData.COORDINATE_FRAME_START_OF_SERVICE,
        TangoPoseData.COORDINATE_FRAME_CAMERA_DEPTH,
        TANGO_SUPPORT_ENGINE_OPENGL,
        TANGO_SUPPORT_ENGINE_TANGO);

if (transform.statusCode == TangoPoseData.POSE_VALID) {
    float[] openGlTPlane = calculatePlaneTransform(
        intersectionPointPlaneModelPair.intersectionPoint,
        intersectionPointPlaneModelPair.planeModel, transform.matrix);

    return openGlTPlane;
} else {
...
```

One last conversion step is needed to map from the Tango system into an OpenGL based system. It's needed to take the **Camera/Cloud Point plane** we just calculated, and map it to our **openGL (or Rajawali)** coordinate system.

[CLICK] Dizzy? That's understandable. There's a lot going on! **[NEXT: TLDR!]**

```java
public boolean onTouch(View view, MotionEvent motionEvent) {
    if (motionEvent.getAction() == MotionEvent.ACTION_UP) {

        // Calculate click location in u,v (0;1) coordinates.
        float u = motionEvent.getX() / view.getWidth();
        float v = motionEvent.getY() / view.getHeight();

        try {
           float[] planeFitTransform;
            synchronized (this) {
                planeFitTransform = doFitPlane(u, v, rgbTimestampGlThread);
            }

            if (planeFitTransform != null) {
                // Update the position of the rendered cube
                // to the pose of the detected plane
                renderer.updateObjectPose(planeFitTransform);
            }

        } catch (TangoException t) {
            ...
```

This is your TLDR slide. All of this hard work happens under the **doFitPlane()** method in the sample code. You call it with u, v and a timestamp, in return you get a plane transform that indicates where in your openGL/Rajawali coordinate system the user's screen tap has landed.

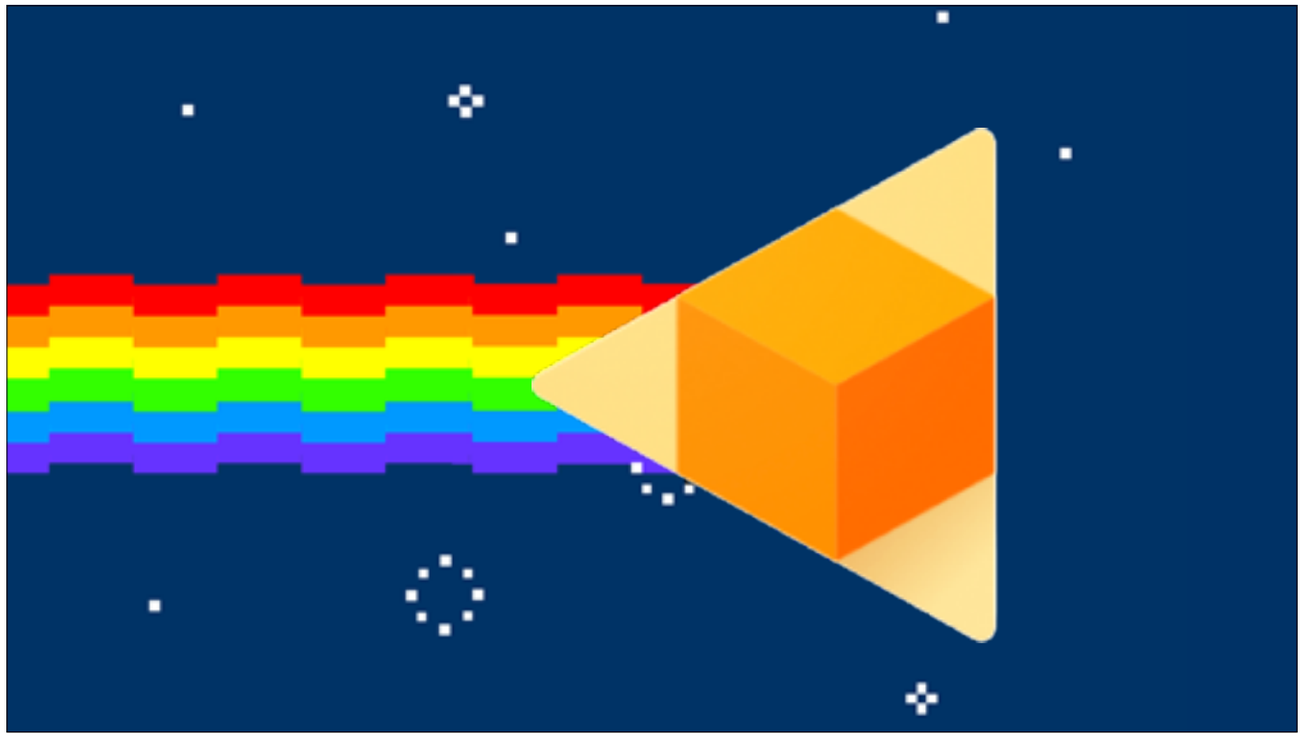
Then all that's left to do is pass that information along to our Rajawali renderer.

```java
/**
 * Save the updated plane fit pose to update the AR object on the next render pass.
 * This is synchronized against concurrent access in the render loop above.
 */
public synchronized void updateObjectPose(float[] planeFitTransform) {
    objectTransform = new Matrix4(planeFitTransform);
    objectPoseUpdated = true;
}
```

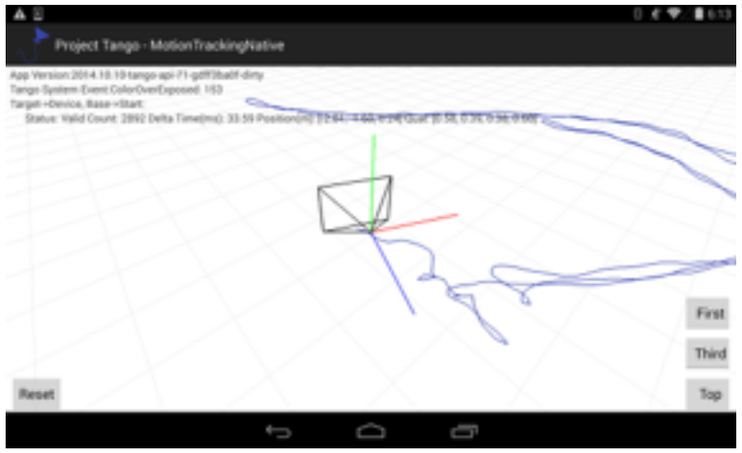
In the renderer, all we do here is store the information. Next time we render a frame on screen… [CLICK]

```
protected void onRender(long elapsedRealTime, double deltaTime) {
    // Update the AR object if necessary
    // Synchronize against concurrent access with the setter below.
    synchronized (this) {
        if (objectPoseUpdated) {
            sphere.setPosition(objectTransform.getTranslation());
            sphere.setOrientation(
                new Quaternion().fromMatrix(objectTransform).conjugate());
            sphere.moveForward(0.25f);
            sphere.setVisible(true);
            objectPoseUpdated = false;
        }
    }
    super.onRender(elapsedRealTime, deltaTime);
}
```

We'll take that objectTransform (our "touchPlane" we just got from Tango), and use it to place an object in our virtual space.



There's one **last** operation that needs to happen. We know where in 3d space the user clicked now. But, the Tango tablet itself is moving through space! It's own position in our virtual world needs to be updated.

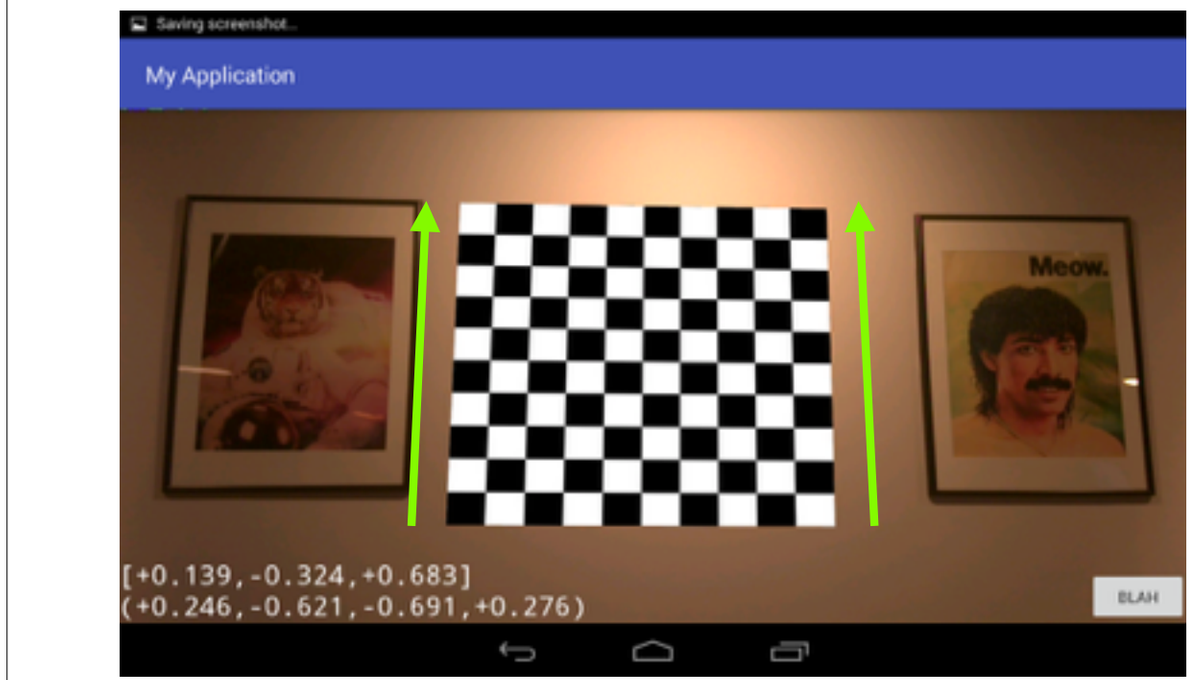



```
public void updateRenderCameraPose(TangoPoseData cameraPose) {
    float[] position = cameraPose.getTranslationAsFloats();
    float[] orientation = cameraPose.getRotationAsFloats();

    getCurrentCamera().setPosition(position[0], position[1], position[2]);
    Quaternion quaternion = new Quaternion(
        orientation[3], orientation[0], orientation[1], orientation[2]);
    getCurrentCamera().setRotation(quaternion.conjugate());
}
```

`updateRenderCameraPose()` receives the device's poseData. Same concept as our virtual objects, we have the *position* and the *orientation* of the Tango device itself. You'll note that the orientation data needs some extra massaging for it to be compatible with Rajawali.

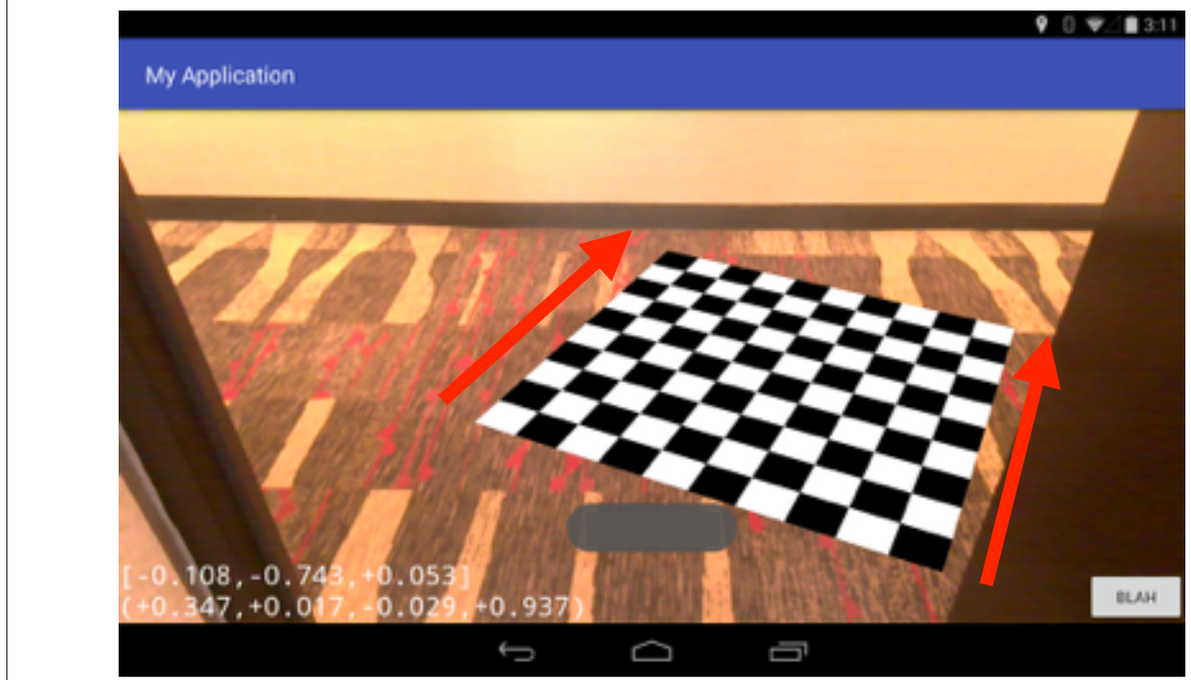
There are a couple more manipulations needed for the camera and the VR renderings to stay in tune, but for all intents and purposes this is boilerplate code I won’t cover here. You’ll find it all in the GitHub repo.

Okay! This is the scary part. Let’s see what this sample code can do. I may or may not have a video as a backup. Let’s hope we don’t need to find out.
[DEMO/VIDEO]



It's worth nothing how some extra operations can be necessary to properly place your virtual objects.

For example, orientation is great when we grab planes on a wall, like this example. The plane always pointed the right way during my tests.



But, orientation is not always what you might want when placing objects on the floor. That can require some extra finessing on the UI end, to build affordances that let users fine-tune object placement, manually orienting the plane after the fact.

And that’s it for me!
- I'll be making the code seen today available on github, but please note, a lot of what we've seen is adapted from official Tango sample code.
- Keep an eye on the official samples, we _might_ see an update when consumer devices go on sale. https://github.com/kanawish/rajadaydream