Leganés, 11 y 12 de febrero

Javier Torres Niño

# Creando un pequeño juego de realidad aumentada

# Índice

# Introducción

# Introducción

Vamos a crear un pequeño juego de realidad aumentada para Android. Utilizamos la cámara del dispositivo para detectar marcadores (similares a códigos QR) y proyectar objetos 3D encima de ellos en la pantalla. La pantalla actúa como un portal que nos deja ver dentro del mundo virtual.

- Utilizamos **ARToolKit** para el procesado de la cámara. La librería detecta los marcadores y nos proporciona la posición/ángulo. Se pueden usar marcadores personalizados (imágenes B/W) o códigos 2D.
- Como motor gráfico usamos **libGDX** por su sencillez. Renderizamos objetos 3D en las posiciones dadas por ARToolKit y procesamos pulsaciones en pantalla para interactuar con el mundo.
- Empezamos sobre un esqueleto que tan sólo tiene el código de inicialización e integración de ambas librerías.

# Configurando el entorno

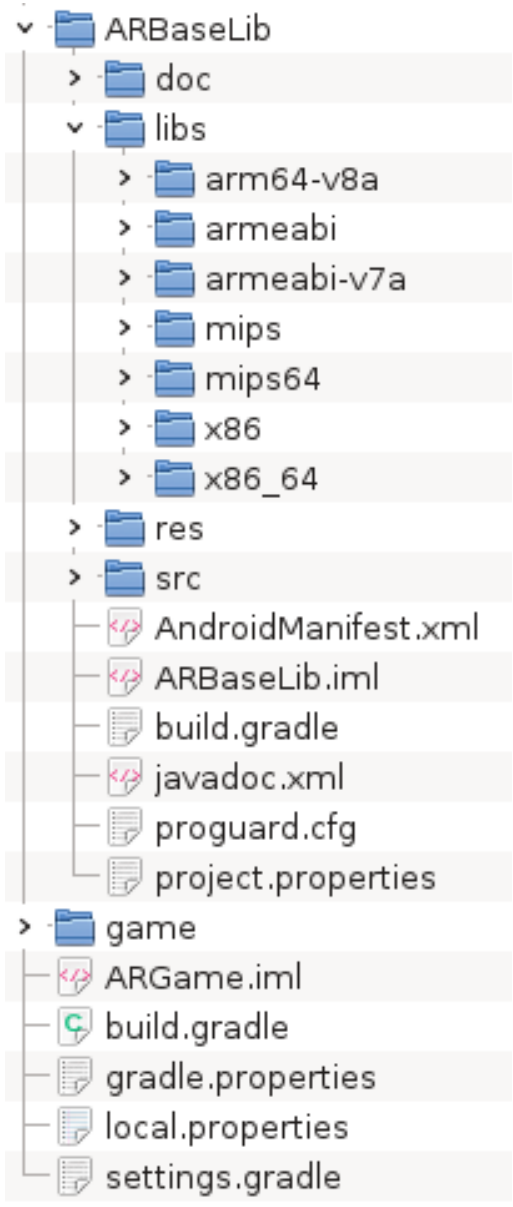
# Dependencias

- Android Studio `http://developer.android.com/sdk`
- ARToolKit para Android
  `http://artoolkit.org/download-artoolkit-sdk`
- libGDX https://libgdx.badlogicgames.com/download.html
  - No lo usaremos en este taller: se proporciona preconfigurado y se descarga automáticamente vía Gradle.
- Esqueleto del proyecto
  `https://javiertorres.eu/t3chfest/esqueleto.zip`
- Estas diapositivas
  `https://javiertorres.eu/t3chfest/slides.pdf`

# Configurar proyecto (I)



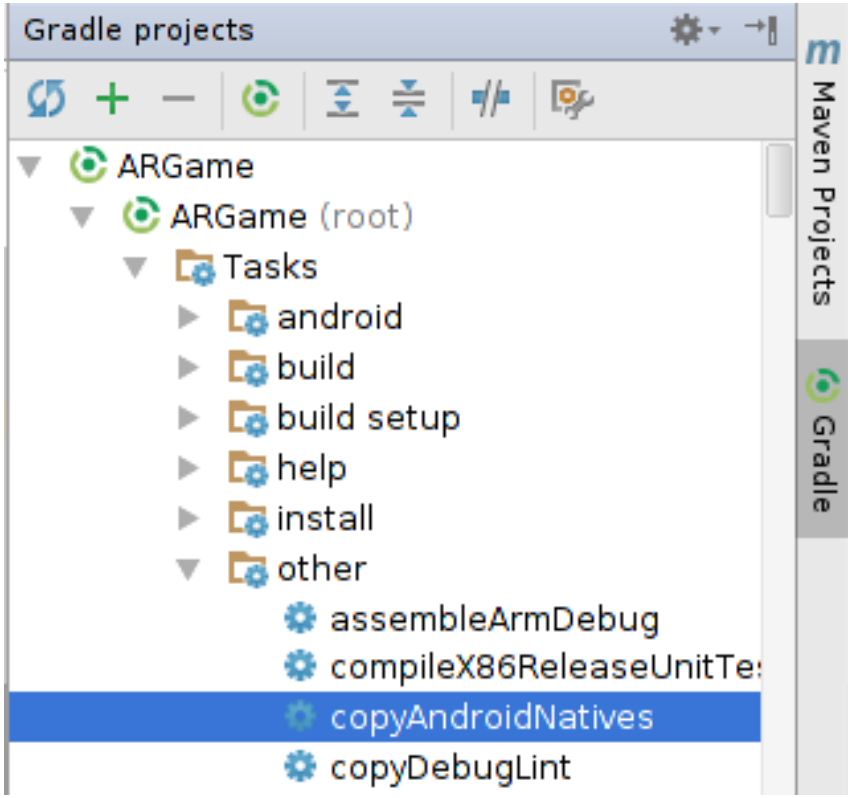
1. Descomprimir esqueleto
2. Copiar contenido de ARToolkit/EclipseProjects/ARBaseLib → ARGame/ARBaseLib
   - El esqueleto viene con la configuración de Gradle para poder usar el proyecto con Android Studio.
3. Copiar ARToolkit/android/libs → ARGame/ARBaseLib
   - Sólo harían falta las correspondientes a la arquitectura del dispositivo o del emulador. Copiamos todas y el compilador decide en función de la configuración.



# Configurar proyecto (II)



1. Ejecutar Android Studio, importar proyecto ARGame
2. Seleccionar arquitectura en Build Variants (barra izquierda, abajo).
3. Ejecutar proyecto: se descargarán dependencias (libGDX) y compilará el proyecto
4. Si falla, copiar librerías gdx (barra derecha, Gradle)



# Detectando marcadores

# Inicialización de ARToolKit



Tenemos que registrar en ARToolKit los marcadores a detectar.

```
1  private Map<Integer, Integer> markerIds;
2  private boolean detectionInit = false;
3  public void initDetection() {
4    // Detectar marcadores de 3x3 con paridad
5    NativeInterface.arwSetPatternDetectionMode(
   ↪    NativeInterface.AR_MATRIX_CODE_DETECTION);
6    NativeInterface.arwSetMatrixCodeType(
   ↪    NativeInterface.AR_MATRIX_CODE_3x3_PARITY65);
7    // Registramos los primeros 24 marcadores a tamaño de 50mm y
   ↪    guardamos el id asignado por ARToolKit.
8    markerIds = new HashMap<Integer, Integer>();
9    for (int i = 0; i < 24; i++) {
10     int mId = ARToolKit.getInstance().addMarker(
   ↪    "single_barcode;" + i + ";50");
11     markerIds.put(i, mId);
12   }
13   detectionInit = true;
14 }
```

# Detección de marcadores



En cada frame de la cámara, detectamos marcadores y obtenemos transformada. De momento, sólo logueamos la posición.

```
1  public void render() {
2    if (!detectionInit)
3      return;
4    // Recorremos los 24 marcadores, utilizando el markerID
5    // asignado para consultar si el marker está visible
6    for (int i = 0; i < 24; i++) {
7      int markerId = markerIds.get(i);
8      if (ARToolKit.getInstance().queryMarkerVisible(markerId)) {
9        Matrix4 mat = new Matrix4(
   ↪    ARToolKit.getInstance().queryMarkerTransformation(markerId));
10       Vector3 vec = new Vector3();
11       mat.getTranslation(vec);
12       Log.d("ARGame", "Visible marcador " + markerId + " en " +
   ↪    vec.x + "," + vec.y + "," + vec.z);
13     }
14   }
15 }
```

# Rendimiento

¡Se crean/destruyen 48 objetos por frame!
Mejor usar uno sólo y reescribirlo, menos recolecciones de basura.

```
1  private Matrix4 arTransform = new Matrix4();
2  private Vector3 vecPosition = new Vector3();
3  public void render() {
4    if (!detectionInit)
5      return;
6    // Recorremos los 24 marcadores, utilizando el markerID
7    // asignado para consultar si el marker está visible
8    for (int i = 0; i < 24; i++) {
9      int markerId = markerIds.get(i);
10     if (ARToolKit.getInstance().queryMarkerVisible(markerId)) {
11       arTransform.set(
   ↪   ARToolKit.getInstance().queryMarkerTransformation(markerId));
12       arTransform.getTranslation(vecPosition);
13       Log.d("ARGame", "Visible marcador " + markerId + " en " +
   ↪   vecPosition.x + "," + vecPosition.y + "," + vecPosition.z);
14     }
15 }
```

# Pintando cubos

# Creando el modelo



Creamos un modelo ("clase") y una instancia ("objeto") de un cubo. Varias instancias comparten el modelo pero pueden estar en distintas posiciones.

```
1  private ModelInstance cube;
2
3  public void create() {
4    [...]
5
6    // Modelo en mm (misma unidad que barcodes)
7    ModelBuilder modelBuilder = new ModelBuilder();
8    Model cubeModel = modelBuilder.createBox(60f, 60f, 60f, new
   ↪   Material(ColorAttribute.createDiffuse(Color.LIME)),
   ↪   VertexAttributes.Usage.Position |
   ↪   VertexAttributes.Usage.Normal);
9    cube = new ModelInstance(cubeModel);
10 }
```

# Pintando el modelo



Pintamos el cubo encima de cada marcador.

```
1  private ModelBatch batch;
2  public void create() {
3    [...]
4    batch = new ModelBatch();
5  }
6  public void render() {
7    batch.begin(cam);
8    for (int i = 0; i < 24; i++) {
9      int markerId = markerIds.get(i);
10     if (ARToolKit.getInstance().queryMarkerVisible(markerId)) {
11       arTransform.set(
   ↪   ARToolKit.getInstance().queryMarkerTransformation(markerId));
12       cube.transform.set(arTransform);
13       batch.render(cube);
14     }
15   }
16   batch.end();
17 }
```

# Limpiando la pantalla



Tenemos que limpiar la pantalla cada vez, o tendremos el efecto "Solitario de Windows".

```
1 public void render() {
2   if (!detectionInit)
3     return;
4   // Limpiar pantalla con fondo transparente para que
5   // se vea la cámara por debajo de la vista OpenGL.
6   Gdx.gl.glViewport(0, 0, Gdx.graphics.getWidth(),
  ↪   Gdx.graphics.getHeight());
7   Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT |
  ↪   GL20.GL_DEPTH_BUFFER_BIT);
8   [...]
9 }
```

# ¡Qué se haga la luz!



Para que nuestros cubos parezcan cubos, necesitamos iluminación.

```
1  private Environment environment;
2
3  public void create() {
4    [...]
5    environment = new Environment();
6    environment.set(new ColorAttribute(ColorAttribute.AmbientLight,
   ↪   0.4f, 0.4f, 0.4f, 1f));
7    environment.add(new DirectionalLight().set(0.8f, 0.8f, 0.8f,
   ↪   -1f, -0.8f, -0.2f));
8  }
9
10 public void render() {
11   [...]
12         batch.render(cube, environment);
13   [...]
14 }
```

# Cargando modelos 3D

# El formato G3DB



- LibGDX utiliza los formatos G3DB (binario) o G3DJ (JSON).
- Se pueden convertir desde otros formatos como FBX (exportable desde muchas herramientas) con la herramienta fbx-conv.
- Para este taller, incluyo algunos modelos sacados de internet:
  - Unas llaves (keys.g3db)
  - Un cofre abierto y cerrado (open.g3db/closed.g3db)
    - Podría ser un único modelo con animaciones.
  - Las texturas del cofre (1.jpg ...)
- Usaremos las llaves para abrir el cofre

# Cargando los modelos

Utilizamos AssetManager de GDX para cargar los modelos.
Permite carga asíncrona, pero no lo utilizamos en este ejemplo.

```
1  private ModelInstance keys;
2
3  public void create() {
4    [...]
5    AssetManager assets = new AssetManager();
6    assets.load("open.g3db", Model.class);
7    assets.load("closed.g3db", Model.class);
8    assets.load("keys.g3db", Model.class);
9    assets.finishLoading();
10
11   Model modelKeys = assets.get("keys.g3db");
12   keys = new ModelInstance(modelKeys);
13   [...]
14 }
```

# Renderizando modelos



Modificamos el código de renderizado para cambiar el modelo por cada marker.

```
1  private static final int MARKER_KEYS = 0, MARKER_CHEST = 1;
2  public void render() {
3    [...]
4      if (ARToolKit.getInstance().queryMarkerVisible(markerId)) {
5        arTransform.set(
   ↪    ARToolKit.getInstance().queryMarkerTransformation(markerId));
6        switch(i) {
7          case MARKER_KEYS:
8            keys.transform.set(arTransform);
9            batch.render(keys, environment);
10           break;
11         case MARKER_CHEST:
12           chestClosed.transform.set(arTransform);
13           batch.render(chestClosed, environment);
14           break;
15       }
16   [...]
```

# Un toque de animación



Podemos hacer las llaves más llamativas con un poco de rotación.

```
1  private long time;
2  public void create() {
3    [...]
4    time = TimeUtils.millis();
5  }
6  public void render() {
7    [...]
8          case MARKER_KEYS:
9            keys.transform.set(arTransform);
10           keys.transform.rotate(0, 0, 1,
   ↪   TimeUtils.timeSinceMillis(time) / 10);
11           batch.render(keys, environment);
12           break;
13   [...]
14 }
```

# Interacción

# Abriendo el cofre



- Vamos a hacer que al arrastrar (en la pantalla) las llaves encima del cofre, este se abra.
- Trataremos los eventos elementales (down, drag, up).
- También existe GestureListener para eventos complejos (tap, fling, pinch, ...)
- Antes de nada, un pequeño cambio de renderizado:

```
1  private boolean chestIsOpen = false;
2  public void render() {
3    [...]
4          case MARKER_CHEST:
5            ModelInstance chest = chestIsOpen ? chestOpen :
   ↪   chestClose;
6            chest.transform.set(arTransform);
7            batch.render(chest, environment);
8            break;
9    [...]
10 }
```

# Recibiendo eventos

Registramos un procesador de entrada.

```
1  public class ARGame implements ApplicationListener,
   ↪  InputProcessor {
2    public void create() {
3      [...]
4      Gdx.input.setInputProcessor(this);
5    }
6    public boolean touchDown(int screenX, int screenY, int pointer,
   ↪  int button) {
7      Log.d("TOUCH", "touchDown " + screenX + "," + screenY);
8      return false;
9    }
10   public boolean touchUp(int screenX, int screenY, int pointer,
   ↪  int button) {
11     Log.d("TOUCH", "touchUp " + screenX + "," + screenY);
12     return false;
13   }
14 }
```

# Detectando objeto pulsado

Para detectar el objeto pulsado, necesitamos hallar la intersección del rayo proyectado por el dedo con los posibles objetos. Por rendimiento se usan cajas o esferas en vez del modelo.

```
1  private BoundingBox keysBbox = new BoundingBox();
2  private boolean draggingKeys = false;
3  public void create() {
4    keys.calculateBoundingBox(keysBbox);
5  }
6  public boolean touchDown(int screenX, int screenY, [...]) {
7    Ray ray = cam.getPickRay(screenX, screenY);
8    BoundingBox keysTransformed = new BoundingBox(keysBbox);
9    keysTransformed.mul(keys.transform);
10   if (Intersector.intersectRayBoundsFast(ray, keysTransformed)) {
11     Log.d("TOUCH", "touchDown intersects keys");
12     draggingKeys = true;
13     return true;
14   }
15   return false;
16 }
```

# Detectando objeto pulsado



```
1  public boolean touchUp(int screenX, int screenY, [...]) {
2    Log.d("TOUCH", "touchUp " + screenX + "," + screenY);
3    if (draggingKeys) {
4      Ray ray = cam.getPickRay(screenX, screenY);
5      BoundingBox chestTransformed = new BoundingBox(chestBbox);
6      chestTransformed.mul(chestClose.transform);
7      if (Intersector.intersectRayBoundsFast(ray,
   ↪    chestTransformed)) {
8        Log.d("TOUCH", "touchUp intersects chest");
9        chestIsOpen = true;
10       draggingKeys = false;
11       return true;
12     }
13   }
14   draggingKeys = false;
15   return false;
16 }
```

# Gracias