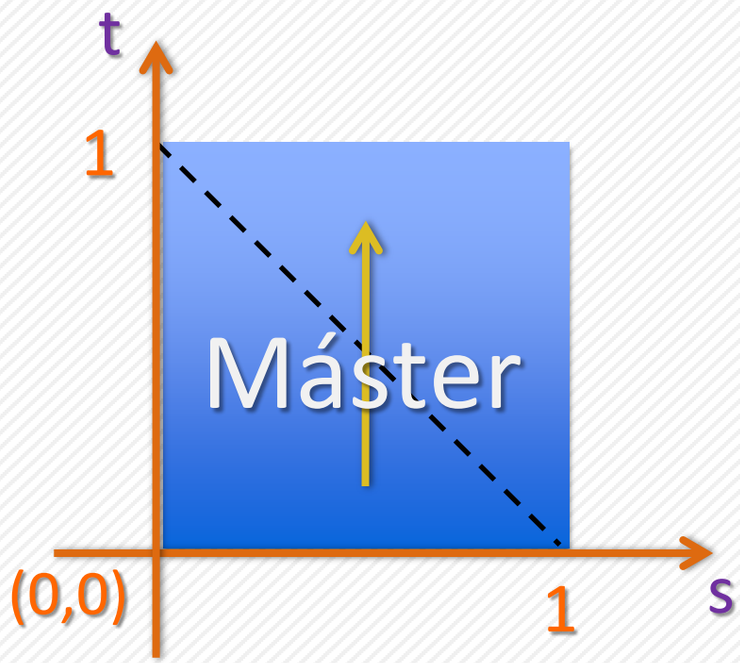
Gráficos y Multimedia

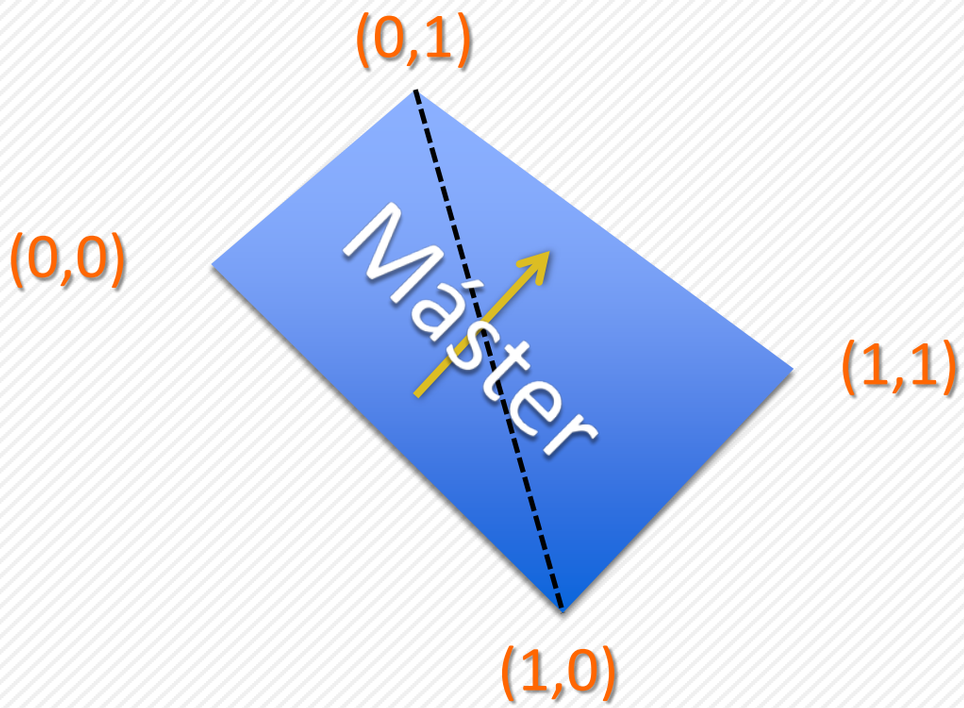
Sesión 2: Texturas y Transparencia. Eventos Táctiles

1. Texturas. Coordenadas

MAPEADO

- Para emplear texturas necesitamos emplear coordenadas de textura en los vértices que creemos para modelar nuestros objetos, estas coordenada se suelen denominar (s, t), también (u, v).
  - donde s y t son coordenadas normalizadas de la textura.
- La textura puede tener el tamaño que queramos, no necesariamente cuadrada, pero su resolución debe ser potencia de 2.





1. Texturas en OpenGL|ES

Cargando Texturas en Android

- Vamos a crear una nueva clase en nuestro proyecto Android llamada `TextureHelper` con el siguiente método:

```
public static int loadTexture(Context context, int resourceId) {
```

- El código será:

```
final int[] textureObjectIds = new int[1];
glGenTextures(1, textureObjectIds, 0);
if (textureObjectIds[0] == 0) {
    if (LoggerConfig.ON) {
        Log.w(TAG, “No se pudo generar una nueva textura OpenGL.");
    }
    return 0;
}
```

- Al comienzo de la clase añadimos también:

```
private static final String TAG = "TextureHelper";
```

1. Texturas en OpenGL|ES

Cargando Texturas en Android

- Para cargar la textura en un *bitmap* escribimos el código:

```
final BitmapFactory.Options options = new BitmapFactory.Options();
options.inScaled = false;
final Bitmap bitmap = BitmapFactory.decodeResource(
      context.getResources(), resourceId, options);
if (bitmap == null) {
      if (LoggerConfig.ON) {
            Log.w(TAG, “El Recurso con ID " + resourceId + " no puede ser
decodificado.");
      }
      glDeleteTextures(1, textureObjectIds, 0);
      return 0;
}
```

- Seleccionamos (activamos) la textura si la carga correctamente con:

```
glBindTexture(GL_TEXTURE_2D, textureObjectIds[0]);
```

1. Texturas en OpenGL|ES

Cargando Texturas en Android

- El filtrado que emplearemos por defecto será *mipmapping* con filtrado *trilineal*, para evitar salto en los píxeles:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

- Seleccionamos (activamos) la textura si la carga correctamente con:

```
glBindTexture(GL_TEXTURE_2D, textureObjectIds[0]);
```

- Creamos la textura en OpenGL:

```
texImage2D(GL_TEXTURE_2D, 0, bitmap, 0);
```

- Para usar `texImage2D()` hay que añadir:

```
import static android.opengl.GLUtils.*;
```

- Y como ya no necesitamos el *bitmap*, liberamos su memoria:

```
bitmap.recycle();
```

1. Texturas en OpenGL|ES

Cargando Texturas en Android

- Podemos generar los *mipmaps* fácilmente usando la llamada:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

- Deseleccionamos (desactivamos) la textura pasando un 0:

```
glBindTexture(GL_TEXTURE_2D, 0);
```

- Para terminar retornamos el ID de la textura:

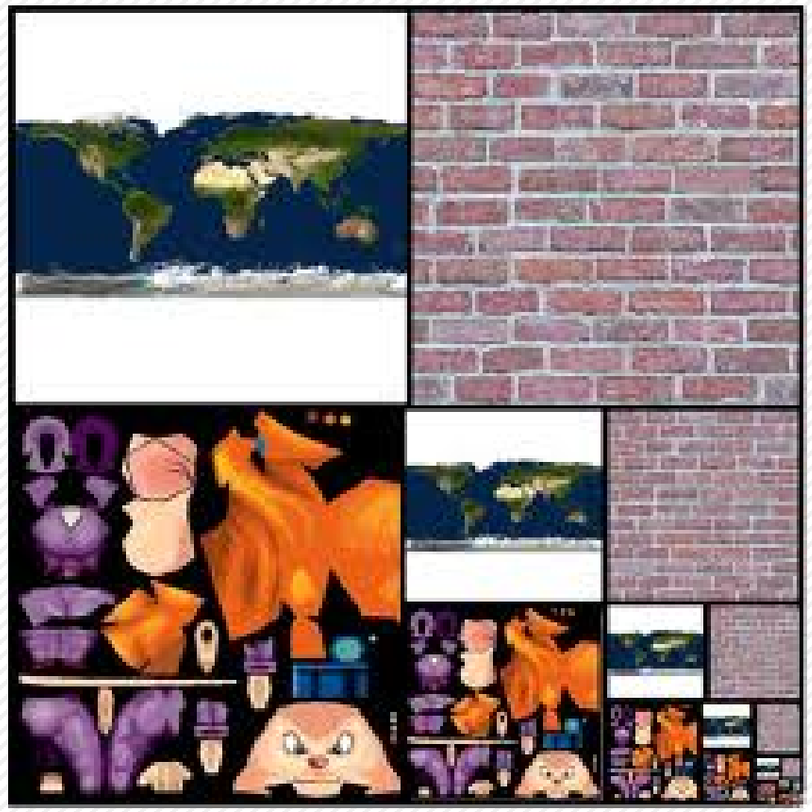
```
        return textureObjectIds[0];
    }
```

1. Texturas en OpenGL|ES

Filtrado de las Texturas

- La textura se puede filtrar de muchas maneras:
  - filtrado por el vecino más cercano,
  - filtrado bilineal,
  - filtrado trilineal y
  - filtrado mipmapping.

1. Texturas en OpenGL|ES



Filtrado de las Texturas

- Los valores que pueden tomar los valores de filtrado son:
  - GL_NEAREST — filtrado al vecino más cercano
  - GL_NEAREST_MIPMAP_NEAREST — filtrado al vecino más cercano con mipmaps
  - GL_NESAREST_MIPMAP_LINEAR — filtrado al vecino más cercano con interpolación entre mapas mipmapping
  - GL_LINEAR — filtrado bilineal
  - GL_LINEAR_MIPMAP_NEAREST — filtrado bilineal con mipmaps
  - GL_LINEAR_MIPMAP_LINEAR entre mapas — filtrado trilineal (bilineal con interpolación mipmapping)

1. Texturas en OpenGL|ES

Filtrado de las Texturas

- **Texel**: las texturas están compuestas matrices de elementos conocidos como *texels*, que contienen un color y un valor alfa, **tex**ture pix**el**.
- Hay tres casos:
  - Cada *texel* corresponde con más de un píxel, esto se denomina *magnification*
  - Cada *texel* corresponde exactamente con un píxel, no se aplicado filtrado.
  - Cada *texel* corresponde con menos de un píxel, esto se denomina *minification*.
- Valores que se pueden emplear en cada caso:
  - `Minification` `GL_NEAREST`
    `GL_NEAREST_MIPMAP_NEAREST`
    `GL_NESAREST_MIPMAP_LINEAR`
    `GL_LINEAR`
    `GL_LINEAR_MIPMAP_NEAREST`
    `GL_LINEAR_MIPMAP_LINEAR`
  - `Magnification` `GL_NEAREST`
    `GL_LINEAR`

1. Texturas en OpenGL|ES

Modificando nuestros shaders

- Debemos modificar nuestros *shaders* para que empleen la textura, nuestra clase `Resorce3DSReader()`, nos proporciona las coordenadas de textura si el modelo .3ds que cargamos dispone de las mismas.

- Por tanto los cambios en nuestro ***vertex shader*** serán:

```
attribute vec2 a_UV;              // in: Coordenadas UV de mapeado de textura
uniform sampler2D u_TextureUnit;// in: Unidad de Textura
```

- En el `main()` el cálculo del color queda como:

```
v_Color = u_Color*ambient+attenuation*(u_Color*texture2D(u_TextureUnit,
     a_UV)*diffuse + specularColor*specular);
```

- Nuestro ***fragment shader*** no sufre modificaciones.

1. Texturas en OpenGL|ES

Modificando nuestros shaders

- Debemos añadir todas las variable para acceder a los nuevos *uniforms* y *attributes*.
- Además debemos asociar el vector de UV con el *attribute* al igual que hicimos con la posición y la normal:

```
// Asociamos el vector de UVs
obj3DS.dataBuffer[i].position(
        POSITION_COMPONENT_COUNT+NORMAL_COMPONENT_COUNT);
glVertexAttribPointer(aUVLocation, NORMAL_COMPONENT_COUNT, GL_FLOAT,
        false, STRIDE, obj3DS.dataBuffer[i]);
```

- Para cargar la textura en `SurfaceCreated()` añadimos después de glClear():

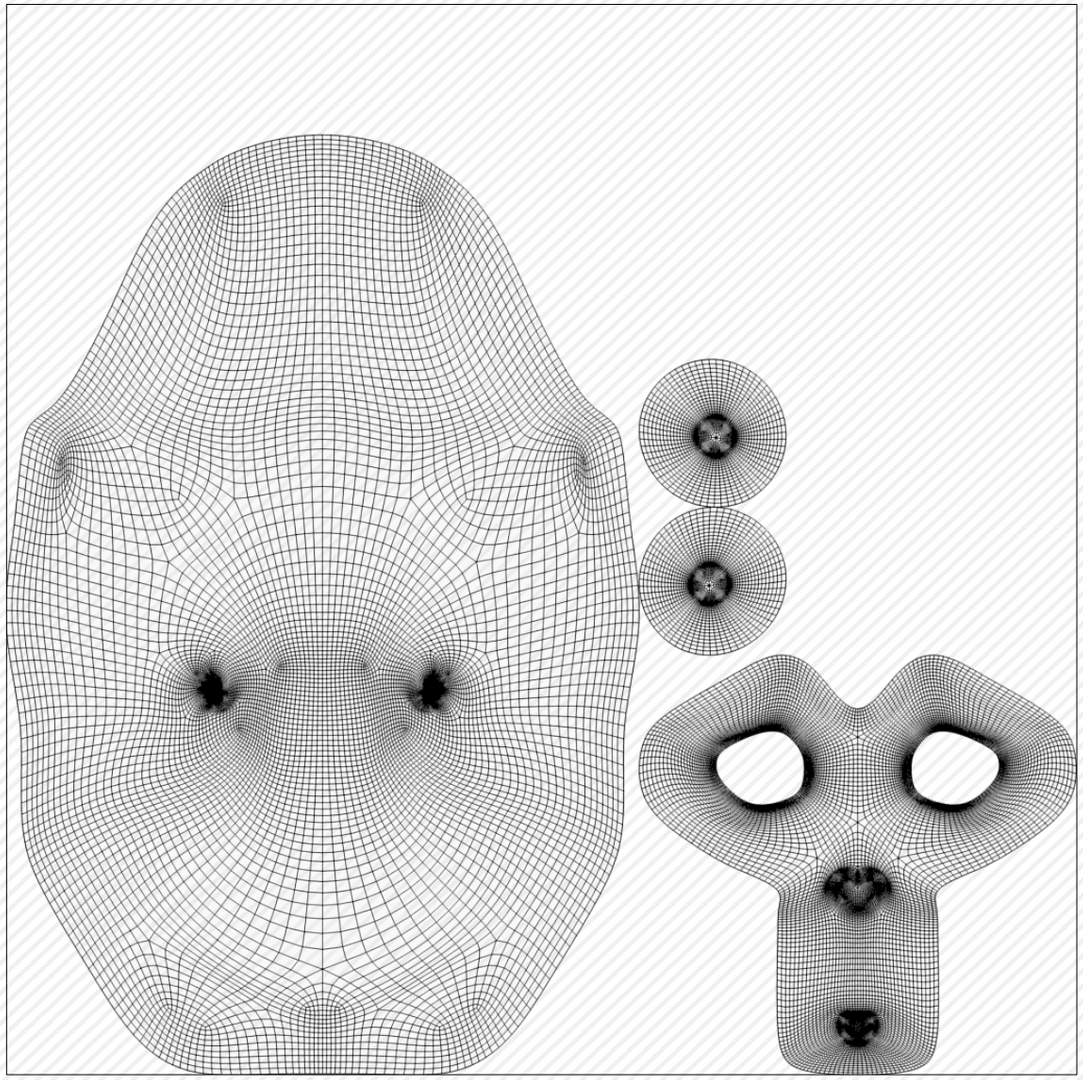
```
// Cargamos la textura desde los recursos
texture = TextureHelper.loadTexture(context, R.drawable.mono_tex);
```

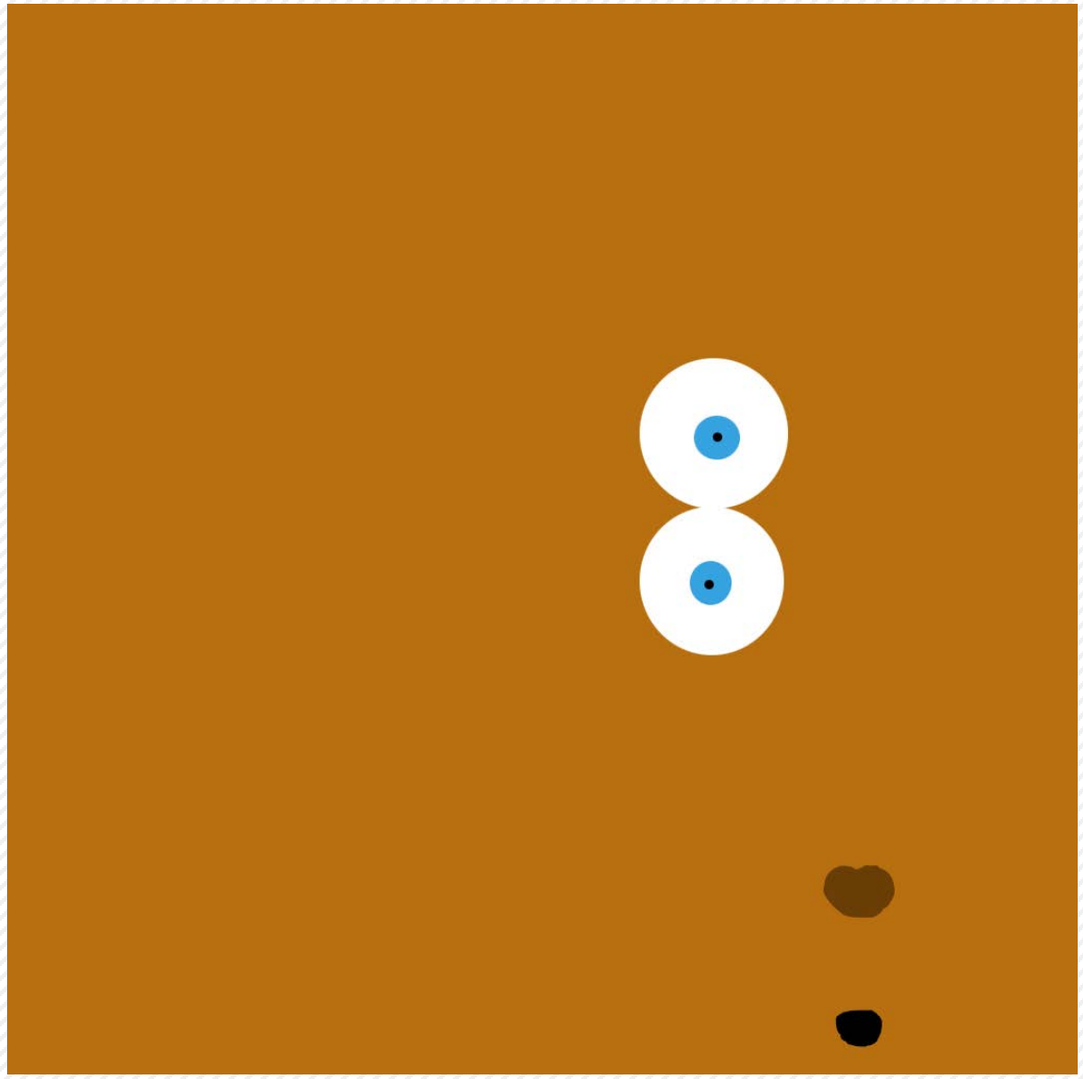
- La textura la colocamos en la carpeta res/drawable.

1. Texturas en OpenGL|ES

Modificando nuestros shaders

- La textura la colocamos en la carpeta *res/drawable*.





1. Texturas en OpenGL|ES

Modificando nuestros shaders

- Es hora de comprobar las capacidades de nuestro dispositivo, instalaremos la app "**OpenGL-ES Info**" de la Play Store. Nos informa de todos los detalles que nuestro dispositivo soporta bajo OpenGL|ES.

- Para comprobar el número de *texture units* soportadas en el *vertex shader* debemos comprobar el valor de MAX_VERTEX_TEXTURE_IMAGE_UNITS.

- Para comprobar el número de *texture units* soportadas en el *fragment shader* debemos comprobar el valor de MAX_TEXTURE_IMAGE_UNITS.

1. Texturas en OpenGL|ES

Modificando nuestros shaders

- El código que debemos emplear será el siguiente:

```
int[]   maxVertexTextureImageUnits = new int[1];
int[]   maxTextureImageUnits       = new int[1];

// Comprobamos si soporta texturas en el vertex shader
glGetIntegerv(GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
maxVertexTextureImageUnits, 0);
if (LoggerConfig.ON) {
    Log.w(TAG, "Max. Vertex Texture Image Units: "
        + maxVertexTextureImageUnits[0]);
}
// Comprobamos si soporta texturas (en el fragment shader)
glGetIntegerv(GL_MAX_TEXTURE_IMAGE_UNITS, maxTextureImageUnits, 0);
if (LoggerConfig.ON) {
    Log.w(TAG, "Max. Texture Image Units: "+maxTextureImageUnits[0]);
}
```

1. Texturas en OpenGL|ES

Modificando nuestros shaders

- Podemos escribir dos *vertex* y *fragment shaders* distintos y así usar el adecuado, para ellos declaramos los *Strings* al principio del método y:

```
// Leemos los shaders
if (maxVertexTextureImageUnits[0]>0) {
    // Textura soportada en el vertex shader
    vertexShaderSource = TextResourceReader
        .readTextFileFromResource(context, R.raw.specular_vertex_shader);
    fragmentShaderSource = TextResourceReader
        .readTextFileFromResource(context, R.raw.specular_fragment_shader);
} else {
    // Textura no soportada en el vertex shader
    vertexShaderSource = TextResourceReader
        .readTextFileFromResource(context, R.raw.specular_vertex_shader2);
    fragmentShaderSource = TextResourceReader
        .readTextFileFromResource(context, R.raw.specular_fragment_shader2);
}
```

2. Transparencia en OpenGL|ES

Blending

- Para emplear la transparencia en OpenGL|ES debemos activar la propiedad *blending*, normalmente desactivaremos el *culling* y el z-buffer.
  - `glEnable(GL_BLEND);`
  - `glDisable(GL_CULL_FACE);`
  - `glDisable(GL_DEPTH_TEST);`

- Además disponemos de dos funciones adicionales para modificar el comportamiento del blending:
  - `glBlendFunc(…);`
  - `glBlendEquation(…);`

2. Transparencia en OpenGL|ES

glBlendEquation()

- El parámetro de `glBlendEquation(mode)` admite tres valores:
  - `GL_FUNC_ADD (valor por defecto),`
  - `GL_FUNC_SUBSTRACT y`
  - `GL_FUNC_REVERSE_SUBTRACT.`

Más información en :
https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBlendEquation.xml

2. Transparencia en OpenGL|ES

glBlendFunc()

- La función `glBlendFunc(GLenum sfactor, GLenum dfactor)` tiene dos parámetros que definen como se calcula el origen (**s**ource) y el destino (**d**estination).

- Para la transparencia se recomienda la siguiente combinación, pero se debe ordenar las primitivas de lejos a cerca:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Más información en :
https://www.khronos.org/opengles/sdk/docs/man/xhtml/glBlendFunc.xml

3. Pantalla táctil

Touching Events

- Queremos emplear nuestra pantalla táctil para mover nuestros objetos. Para ello debemos añadir el siguiente código en nuestra aplicación, en primer lugar hacemos un pequeño cambio en `OnCreate()` en nuestra actividad:

```
final OpenGLRenderer openGLRenderer = new OpenGLRenderer(this);
```

- El siguiente código se cambiará para usar esa variable:

```
if (supportsEs2) {
        …
        // Asigna nuestro renderer.
        glSurfaceView.setRenderer(openGLRenderer);
```

3. Pantalla táctil

Touching Events

- Necesitamos añadir el siguiente método antes de setContentView():

```
glSurfaceView.setOnTouchListener(new OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
        if (event != null) {
           // Convert touch coordinates into normalized device
           // coordinates, keeping in mind that Android's Y
           // coordinates are inverted.
           final float normalizedX =   (event.getX()/(float) v.getWidth())*2 - 1;
           final float normalizedY = -((event.getY()/(float) v.getHeight())*2 - 1);
           if (event.getAction() == MotionEvent.ACTION_DOWN) {
             glSurfaceView.queueEvent(new Runnable() {
                 @Override
                 public void run() {
                     openGLRenderer.handleTouchPress(normalizedX, normalizedY);
                 }
             });
```

3. Pantalla táctil

Touching Events

```
        } else if (event.getAction() == MotionEvent.ACTION_MOVE) {
            glSurfaceView.queueEvent(new Runnable() {
                @Override
                public void run() {
                    openGLRenderer.handleTouchDrag(normalizedX, normalizedY);
                }
            });
        }
        return true;
    } else {
        return false;
    }
  }
});
```

Hay que recordar que el interfaz de usuario de Android (en nuestra *activity*) se ejecuta en un hilo, mientras que `glSurfaceView` ejecuta OpenGL en otro hilo separado, para que se puedan comunicar estos dos hilos debemos hacer uso de `queueEvent()`.

3. Pantalla táctil

Touching Events

- Para terminar nos queda añadir en OpenGLRenderer añadir los métodos para controlar los eventos (*touch & drag*):

```
public void handleTouchPress(float normalizedX, float normalizedY) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Touch Press ["+normalizedX+", "+normalizedY+"]");
    }
}
public void handleTouchDrag(float normalizedX, float normalizedY) {
    if (LoggerConfig.ON) {
        Log.w(TAG, "Touch Drag ["+normalizedX+", "+normalizedY+"]");
    }
    rX = -normalizedY*180f;
    rY = normalizedX*180f;
}
```

3. Pantalla táctil

Touching Events

- Añadimos dos variables rX y rY para controlar la rotación en esos eje, de forma que en onDrawFrame() la modelMatrix la cosntruimos de la siguiente forma:

```
// Creamos la matriz del modelo
setIdentityM(modelMatrix, 0);
translateM(modelMatrix, 0, 0f, 0.0f, -4.0f);
// Rotación alrededor del eje x e y
rotateM(modelMatrix, 0, rY, 0f, 1f, 0f);
rotateM(modelMatrix, 0, rX, 1f, 0f, 0f);
```

Empleamos el movimiento horizontal para la rotación Y y el movimiento vertical para la rotación X.