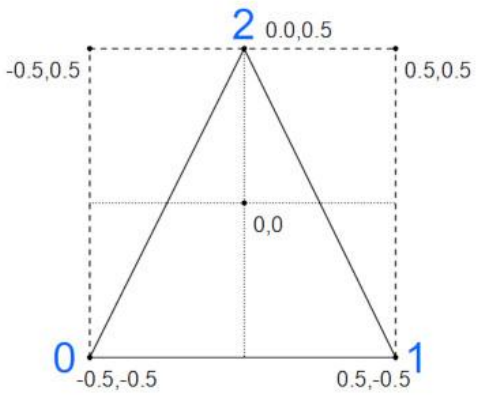
Android - OpenGL ES 2 - Tutorial 2

Disegnare un Triangolo

Consideriamo il triangolo dell'immagine seguente:



Continuando sul progetto base 2D, creiamo una nuova classe "ObjTriangle" e sostituiamo il codice con il seguente:

```
package opengles2.tutorial2;

import android.opengl.GLES20;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.opengles.GL10;

public class ObjTriangle {
    private FloatBuffer vertexBuffer;
    private ShortBuffer indexBuffer;

    private final int COORDS_PER_VERTEX = 2;
    private final int vertexStride = COORDS_PER_VERTEX * 4;
    private float[] vertices = {
            -0.5f, -0.5f,
            0.5f, -0.5f,
            0.0f, 0.5f,
    };

    private short[] indices = {
            0, 1, 2
    };

    // Color for all vertices (Red).
    private float[] color = {
            1.0f, 0.0f, 0.0f, 1.0f
    };

    private final String vertexShaderCode =
            "uniform mat4 uMVPMatrix;" +
            "attribute vec4 vPosition;" +
            "void main() {" +
            "  gl_Position = uMVPMatrix * vPosition;" +
```

```
            "}";

    private final String fragmentShaderCode =
            "precision mediump float;" +
            "uniform vec4 vColor;" +
            "void main() {" +
            "  gl_FragColor = vColor;" +
            "}";

    private int mProgram;

    public ObjTriangle() {
        // Set vertex buffer from data.
        ByteBuffer vbb = ByteBuffer.allocateDirect(this.vertices.length * 4);
        vbb.order(ByteOrder.nativeOrder());
        this.vertexBuffer = vbb.asFloatBuffer();
        this.vertexBuffer.put(this.vertices);
        this.vertexBuffer.position(0);

        // Set index buffer from data.
        ByteBuffer ibb = ByteBuffer.allocateDirect(this.indices.length * 2);
        ibb.order(ByteOrder.nativeOrder());
        this.indexBuffer = ibb.asShortBuffer();
        this.indexBuffer.put(this.indices);
        this.indexBuffer.position(0);
    }
    public void SetProgramShader() {
        // Load shaders.
        int vertexShader = MainRenderer.LoadShader(GLES20.GL_VERTEX_SHADER,
this.vertexShaderCode);
        int fragmentShader = MainRenderer.LoadShader(GLES20.GL_FRAGMENT_SHADER,
this.fragmentShaderCode);

        // Create empty OpenGL ES Program.
        this.mProgram = GLES20.glCreateProgram();
        // Add the vertex shader to program.
        GLES20.glAttachShader(this.mProgram, vertexShader);
        // Add the fragment shader to program.
        GLES20.glAttachShader(this.mProgram, fragmentShader);
        // Creates OpenGL ES program executables.
        GLES20.glLinkProgram(this.mProgram);
    }

    private int positionHandle;
    private int colorHandle;
    private int modelHandle;

    public void DrawTriangle(float[] modelMatrix) {
        // Add program to OpenGL ES environment.
        GLES20.glUseProgram(this.mProgram);

        // Get handle to vertex shader's vPosition member.
        this.positionHandle = GLES20.glGetAttribLocation(this.mProgram, "vPosition");
        // Enable a handle to the triangle vertices.
        GLES20.glEnableVertexAttribArray(this.positionHandle);
        // Prepare the triangle coordinate data.
        GLES20.glVertexAttribPointer(this.positionHandle, COORDS_PER_VERTEX,
GLES20.GL_FLOAT, false, this.vertexStride, this.vertexBuffer);

        // Get handle to fragment shader's vColor member.
        this.colorHandle = GLES20.glGetUniformLocation(this.mProgram, "vColor");
        // Set color for drawing the triangle.
        GLES20.glUniform4fv(this.colorHandle, 1, this.color, 0);

        // Get handle to shape's transformation matrix.
        this.modelHandle = GLES20.glGetUniformLocation(this.mProgram, "uMVPMatrix");
        // Pass the projection and view transformation to the shader.
        GLES20.glUniformMatrix4fv(this.modelHandle, 1, false, modelMatrix, 0);
```

```
        // Draw the triangle.
        GLES20.glDrawElements(GL10.GL_TRIANGLES, this.indices.length,
GL10.GL_UNSIGNED_SHORT, this.indexBuffer);

        // Disable vertex array.
        GLES20.glDisableVertexAttribArray(this.positionHandle);
    }

}
```

Modifichiamo la classe "MainRenderer" nel seguente modo:

```
package opengles2.tutorial2;

import android.opengl.GLES20;
import android.opengl.GLSurfaceView;
import android.opengl.Matrix;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

public class MainRenderer implements GLSurfaceView.Renderer {
    int width, height;

    private final float[] projMatrix = new float[16];
    private final float[] viewMatrix = new float[16];
    // Projection and view transformation.
    private final float[] basicMatrix = new float[16];
    // Temp models matrix.
    private float[] modelMatrix = new float[16];

    ObjTriangle objtriangle;

    // Constructor.
    public MainRenderer() {
        // Set class.
        this.objtriangle = new ObjTriangle();
    }

    @Override
    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        GLES20.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

        // Set program shader.
        this.objtriangle.SetProgramShader();
    }

    @Override
    public void onSurfaceChanged(GL10 unused, int width, int height) {
        this.width = width;
        this.height = height;

        GLES20.glViewport(0, 0, this.width, this.height);
        Matrix.orthoM(this.projMatrix, 0, 0.0f, this.width - 1.0f, 0.0f, this.height -
1.0f, 1.0f, -1.0f);
        Matrix.setLookAtM(this.viewMatrix, 0, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f,
1.0f, 0.0f);
        // Calculate the projection and view transformation for models.
        Matrix.multiplyMM(this.basicMatrix, 0, this.projMatrix, 0, this.viewMatrix, 0);
    }

    @Override
    public void onDrawFrame(GL10 unused) {
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);

        this.ResetModelMatrix();
        // Translate triangle.
```

```
        Matrix.translateM(this.modelMatrix, 0, this.width / 2.0f, this.height / 2.0f,
0.0f);
        // Scale triangle.
        Matrix.scaleM(this.modelMatrix, 0, 100.0f, 100.0f, 0.0f);
        // Draw triangle.
        this.objtriangle.DrawTriangle(this.modelMatrix);
    }

    public static int LoadShader(int type, String shaderCode) {
        // Create a shader type.
        int shader = GLES20.glCreateShader(type);
        // Add the source code to the shader and compile it.
        GLES20.glShaderSource(shader, shaderCode);
        GLES20.glCompileShader(shader);

        return shader;
    }

    private void ResetModelMatrix() {
        this.modelMatrix = this.basicMatrix.clone();
    }

}
```

Notiamo la presenza del costruttore "MainRenderer()" in quanto abbiamo la necessità di definire la classe "ObjTriangle" nel momento in cui il renderer viene definito in "MainView".

Con "this.objtriangle = new ObjTriangle();" richiamiamo il costruttore della classe "ObjTriangle" dove settiamo i dati del nostro triangolo. I vertici del triangolo presenti in "vertices" e "indices" (scritti in senso antiorario) non sono in un formato utile ad OpenGL, nel costruttore vengono converititi e memorizzati in "vertexBuffer" e "indexBuffer".

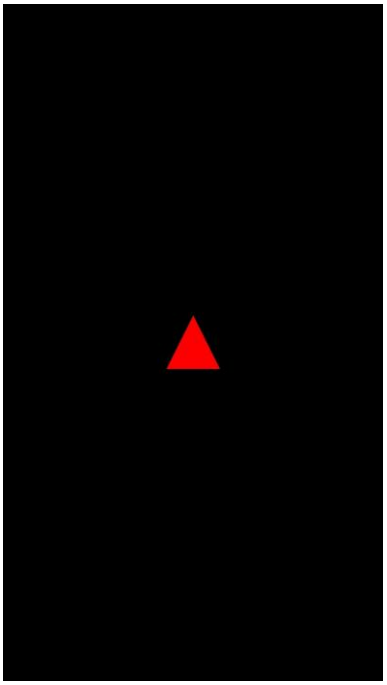
Nel metodo "onSurfaceCreated()" inizializziamo il program shader con "this.objtriangle.SetProgramShader();".

Nel metodo "onSurfaceChanged()", dopo aver impostato la matrice di proiezione e di camera, impostiamo la matrice di base per i modelli, in pratica moltiplichiamo la matrice di proiezione per la matrice di camera.

Nel metodo "onDrawFrame()":

1. cancello il frame;
2. resetto la matrice del modello;
3. effettuo una traslazione sulla matrice corrente;
4. effettuo uno scalo sulla matrice corrente;
5. disegno il triangolo richiamando la funzione presente nella classe "ObjTriangle" e passando la matrice di base per apportare le modifiche alle coordinate del triangolo.

Il risultato è:



Sapendo che il nostro sistema di coordinate ortogonali è di tipo "1:1" con lo schermo ed il triangolo di dimensioni "1x1", traslare di 200 equivale ad uno spostamento di 200 e scalare di 10 equivale ad un ridimensionamento di 10. Ma fate attenzione all'ordine! Se prima scaliamo di 10, quando trasleremo di 200 ci sposteremo di 2000, cioè la nuova "unità di base" (10) per 200 volte.