PRACTICE #04

OpenGL ES

(*Keyword: OpenGL-ES, GLES)*

I. Goals

- Students can use OpenGL ES to implement algorithms as well as computer graphics applications on the Mobile environment (Android OS for this content).

II. Introduction

OpenGL ES - OpenGL for Embedded Systems

- OpenGL® ES is a royalty-free, cross-platform API for rendering advanced 2D and 3D graphics on embedded and mobile systems - including consoles, phones, appliances, and vehicles.

- It consists of a well-defined subset of desktop OpenGL suitable for low-power devices and provides a flexible & powerful interface between software and graphics acceleration hardware.

- OpenGL ES is the "most widely deployed 3D graphics API in history.

- The API is cross-language and multi-platform. The libraries GLUT and GLU are not available for OpenGL ES. OpenGL ES is managed by the non-profit technology consortium Khronos Group. Vulkan, a next-generation API from Khronos, is made for simpler high-performance drivers for mobile and desktop devices.

III. Content

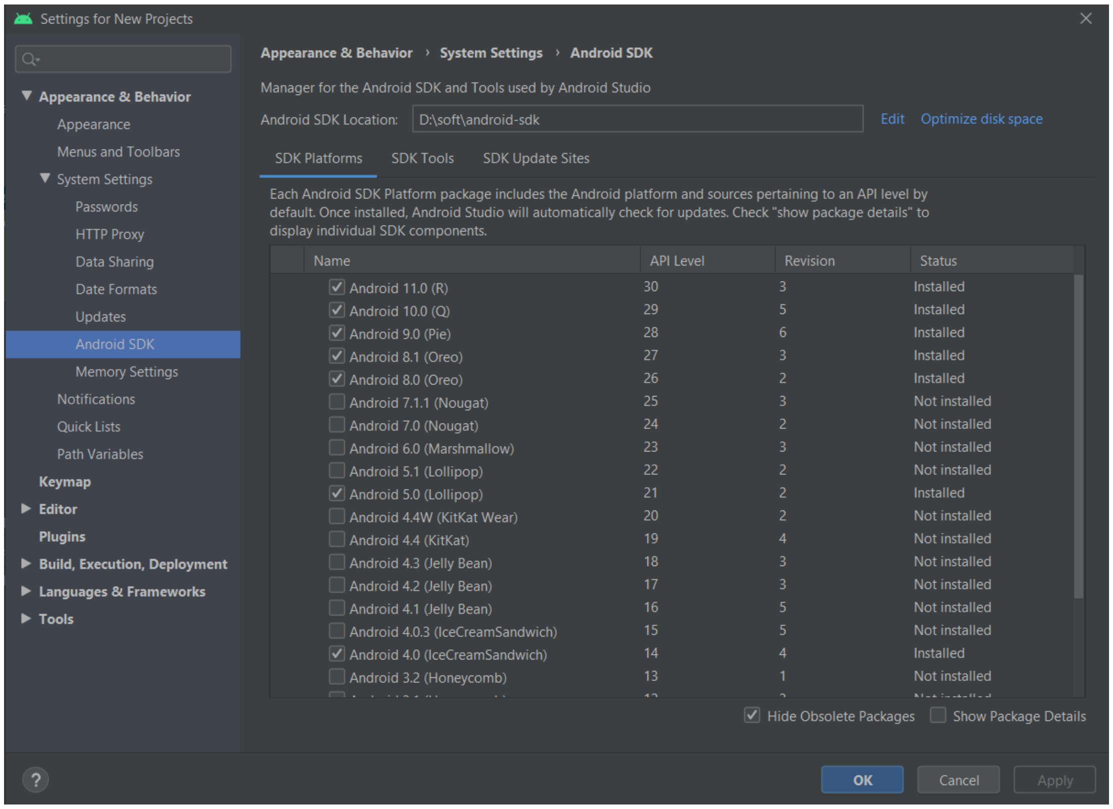
1. Prepare the necessary programming environment
   - Android Studio at https://developer.android.com/studio
   - Download the corresponding Android SDK
   - Prepare Android device or using virtual devices in AVD
2. Features
   Implement a simple graphical mobile application whose interface serves the following requirements:
   - Use the touch screen to draw geometric objects such as points, lines, squares, rectangles, circles.
   - Allow users to choose the thickness, shape (solid/dashed line...), or color…
   - Other optional extras features/functions (consider plus points).

IV. Requirements

1. The directory structure of the compressed submission
   - *doc*: report files include MSSV_report_th04.doc and MSSV_report_th04.pdf
   - *release*: contains APK file
   - *source*: contains entire source code, removed temporary files, intermediate compiled files...
   - *bonus*: optional, for plus points
2. Other requirements
   - The report should be presented clearly and intuitively: list the functions supported by the program with proof images, summarize the usage and implementation (through pseudo-code, description of methods, or how to do it, *do not copy the source code into the report*).
   - The source code needs to be commented on the corresponding lines.
   - If implemented for other environments, such as iOS, can submit other additional separate compressed files into the folder bonus (bonus), but must have the required submission for the Android OS mobile environment.

V. Additional guidance

1. Android SDK (software development kit)
   - Check & download the corresponding Android SDK Platform package (menu *Tool / SDK Manager*)



Appearance & Behavior › System Settings › Android SDK

Manager for the Android SDK and Tools used by Android Studio

Android SDK Location: D:\soft\android-sdk   Edit   Optimize disk space

SDK Platforms   SDK Tools   SDK Update Sites

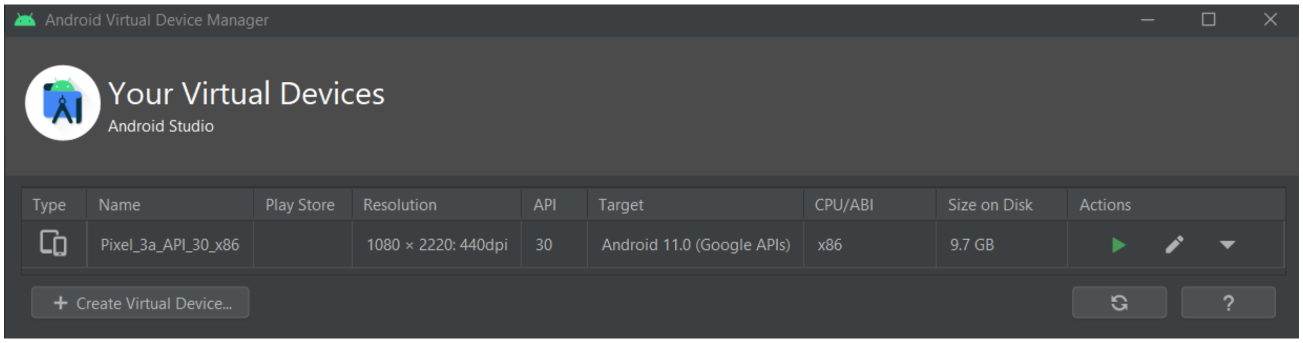
Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, Android Studio will automatically check for updates. Check "show package details" to display individual SDK components.

Name	API Level	Revision	Status
✓ Android 11.0 (R)	30	3	Installed
✓ Android 10.0 (Q)	29	5	Installed
✓ Android 9.0 (Pie)	28	6	Installed
✓ Android 8.1 (Oreo)	27	3	Installed
✓ Android 8.0 (Oreo)	26	2	Installed
Android 7.1.1 (Nougat)	25	3	Not installed
Android 7.0 (Nougat)	24	2	Not installed
Android 6.0 (Marshmallow)	23	3	Not installed
Android 5.1 (Lollipop)	22	2	Not installed
✓ Android 5.0 (Lollipop)	21	2	Installed
Android 4.4W (KitKat Wear)	20	2	Not installed
Android 4.4 (KitKat)	19	4	Not installed
Android 4.3 (Jelly Bean)	18	3	Not installed
Android 4.2 (Jelly Bean)	17	3	Not installed
Android 4.1 (Jelly Bean)	16	5	Not installed
Android 4.0.3 (IceCreamSandwich)	15	5	Not installed
✓ Android 4.0 (IceCreamSandwich)	14	4	Installed
Android 3.2 (Honeycomb)	13	1	Not installed

✓ Hide Obsolete Packages   Show Package Details

2. Android Virtual Device Manager (AVD)

- Create the virtual Android device, for use in testing your application directly on the PC.



Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Pixel_3a_API_30_x86		1080 × 2220: 440dpi	30	Android 11.0 (Google APIs)	x86	9.7 GB	

3. Sample code guidance (draw a triangle)
   - Add the OpenGL ES dependency by editing *androidManifest.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.opengl_es">
    <uses-feature android:glEsVersion="0x00020000"
android:required="true" />
</manifest>
```

   - Create a class for holding the view, it will represent the area on the screen that we want to be visualized. This class extends the *GLSurfaceView* class and implement some needed functionality:

```
package com.example.opengl_es;

import android.content.Context;
import android.opengl.GLSurfaceView;

public class CGLSurfaceView extends GLSurfaceView{
    private final CGLRenderer renderer;

    public CGLSurfaceView(Context context){
        super(context);

        // create an OpenGL ES 2.0 context
        setEGLContextClientVersion(2);

        renderer = new CGLRenderer();

        // set the Renderer for drawing on the GLSurfaceView
        setRenderer(renderer);
    }
}
```

- In the above class, we created an instance of class *CGLRenderer*, this will handle the processing for any coordinates we want to visualize. The *MainActivity* holds the view which holds the renderer. The renderer controls what is drawn on the *GLSurfaceView*:

```java
package com.example.opengl_es;

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

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

public class CGLRenderer implements GLSurfaceView.Renderer {
    private CTriangle cTriangle;

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

        cTriangle = new CTriangle();
    }

    public void onDrawFrame(GL10 unused) {
        // redraw background color
        GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT);

        cTriangle.draw();
    }

    public void onSurfaceChanged(GL10 unused, int width, int height) {
        // set the viewport to the size of the view.
        GLES20.glViewport(0, 0, width, height);
    }

    // create a vertex shader type (GLES20.GL_VERTEX_SHADER)
    // or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)
    public static int loadShader(int type, String shaderCode) {
        int shader = GLES20.glCreateShader(type);
        GLES20.glShaderSource(shader, shaderCode);
        GLES20.glCompileShader(shader);
        return shader;
    }
}
```

There are three lifecycle methods here that are called while the renderer is running.
- *onSurfaceCreated()*: when the renderer is created.
- *onDrawFrame()*: when the view needs to be redrawn.
- *onSurfaceChanged()*: when the orientation on the device changes.

- We also need to create a new triangle in the *CGLRenderer* class and also initialize it in *onSurfaceCreated()*.

For more details, here, the OpenGL ES will use C++ shaders to handle the graphics rendering. We could create the C++ source code as strings into our object and pass them to the renderer. Then, we could add the shaders into the Triangle class at the top of the class before the line where *FloatBuffer* is created.

We will need to compile our shaders to handle these shaders in our renderer, wrap that process in a function to pass in the type and content of the shader. And we need to use these shaders in the Triangle's constructor.

```
package com.example.opengl_es;

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

import android.opengl.GLES20;

public class CTriangle {
    private final String vertex_shader_code =
            "attribute vec4 vPosition;" +
                    "void main() {" +
                    " gl_Position = vPosition;" +
                    "}";

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

    private FloatBuffer vertex_buffer;

    // number of coordinates per vertex in this array
    static final int COORS_PER_VERTEX = 3;

    // counterclockwise order
    static float[] triangle_coors = {
            0.0f, 0.311004243f, 0.0f, // top
            -0.5f, -0.311004243f, 0.0f, // bottom left
            0.5f, -0.311004243f, 0.0f // bottom right
    };

    // set color with red, green, blue and alpha (opacity) values
    float color[] = {0.3f, 0.5f, 0.7f, 1.0f};

    private final int program;
```

- To start the rendering and set our base configuration, we will implement a *draw* method in our Triangle. This will also be where our program binds to the C++ shader we added earlier.

```
private int positionHandle;

private int colorHandle;

private final int vertexCount = triangle_coors.length / COORS_PER_VERTEX;

// 4 bytes per vertex
private final int vertexStride = COORS_PER_VERTEX * 4;

// create a vertex shader type (GLES20.GL_VERTEX_SHADER)
// or a fragment shader type (GLES20.GL_FRAGMENT_SHADER)
public static int loadShader(int type, String shaderCode) {
    int shader = GLES20.glCreateShader(type);
    // add the shader 's source code and compile it
    GLES20.glShaderSource(shader, shaderCode);
    GLES20.glCompileShader(shader);
    return shader;
}

public CTriangle() {
    // initialize vertex byte buffer for shape coordinates
    // (number of coordinate values * 4 bytes per float)
    ByteBuffer bb = ByteBuffer.allocateDirect(triangle_coors.length * 4);

    // use the device hardware’s native byte order
    bb.order(ByteOrder.nativeOrder());

    // create a floating point buffer from the ByteBuffer
    vertex_buffer = bb.asFloatBuffer();
    // add the coordinates to the FloatBuffer
    vertex_buffer.put(triangle_coors);
    // set the buffer to read the first coordinate
    vertex_buffer.position(0);

    int vertexShader = CGLRenderer.loadShader(GLES20.GL_VERTEX_SHADER,
vertex_shader_code);
    int fragmentShader = CGLRenderer.loadShader(GLES20.GL_FRAGMENT_SHADER,
fragment_shader_code);

    // create empty OpenGL ES Program
    program = GLES20.glCreateProgram();

    // add the vertex shader to program
    GLES20.glAttachShader(program, vertexShader);

    // add the fragment shader to program
    GLES20.glAttachShader(program, fragmentShader);

    // creates OpenGL ES program executables
    GLES20.glLinkProgram(program);
}
```

- We will call the draw method created below, implement this at the end of the *onDrawFrame* method in the renderer.

```java
public void draw() {
    // add program to OpenGL ES environment
    GLES20.glUseProgram(program);

    // get handle to vertex shader’s vPosition member
    positionHandle = GLES20.glGetAttribLocation(program, "vPosition");

    // enable a handle to the triangle vertices
    GLES20.glEnableVertexAttribArray(positionHandle);

    // prepare the triangle coordinate data
    GLES20.glVertexAttribPointer(positionHandle, COORS_PER_VERTEX,
            GLES20.GL_FLOAT, false,
            vertexStride, vertex_buffer);

    // get handle to fragment shader’s vColor member
    colorHandle = GLES20.glGetUniformLocation(program, "vColor");

    // set color for drawing the triangle
    GLES20.glUniform4fv(colorHandle, 1, color, 0);

    // draw the triangle
    GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, vertexCount);

    // disable vertex array
    GLES20.glDisableVertexAttribArray(positionHandle);
}
```

- The displayed result from the example source code: