Лабораторна робота №3. Перетворення координат та проєкції. Анімація. Керування за допомогою сенсорів вводу

Мета: отримати навички програмування відображення тривимірних об'єктів засобами графіки OpenGL ES.

У цій лабораторній роботі будуть розглянуті питання програмування рендерингу тривимірних об'єктів та визначення ракурсу їхнього показу.

Завдання

Потрібно створити у середовищі Android Studio проєкт з ім'ям **Lab3_GLES**, зокрема

- написати вихідний текст програми згідно варіанту завдання. Використати мову Java або Kotlin – на вибір
- Налагодити програму. Перевірити роботу програми на емуляторі та на фізичному пристрої Android.

Засоби для виконання лабораторної роботи

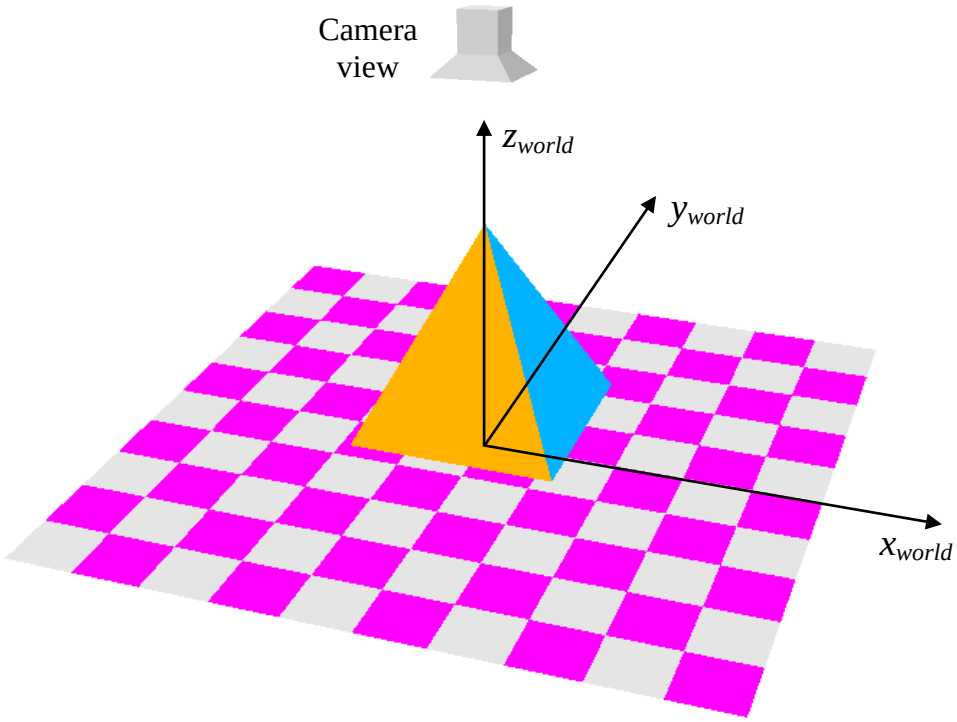
Android Studio – середовище розробки Android-застосунків.

Теоретичні положення та методичні рекомендації

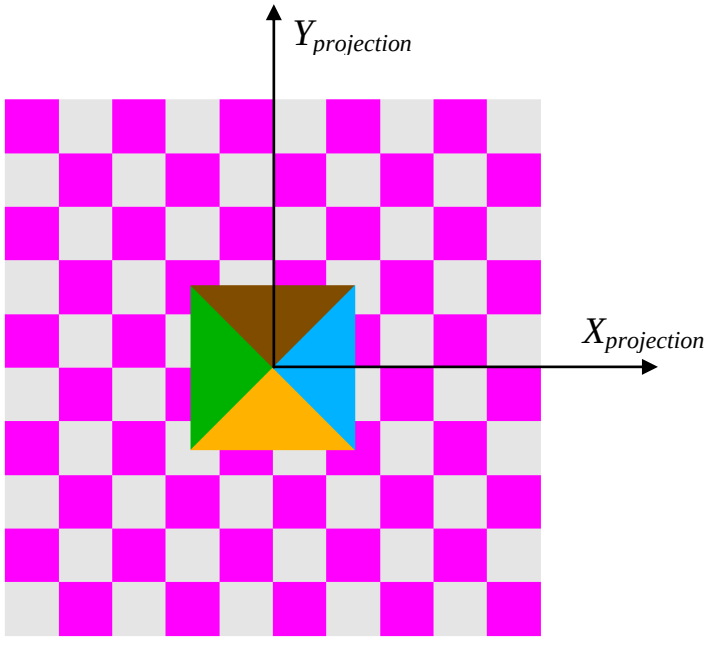
Створення проєкту Android Studio – так само, як і для попередніх лабораторних робіт. Варто нагадати, що для успішного використання OpenGL ES версії 3.2 потрібно вказати цільовий API рівня не нижче 28 (Android 9.0).

Проєкції у OpenGL ES

Оскільки у вікні відображення система OpenGL по замовчуванню спрямовує вісь *Y* від центру догори уверх, то у якості світових координат для опису об'єктів просторових сцен зручно використовувати відповідну систему координат, яку можна позначити як (x_{world} , y_{world} , z_{world}) – далі будемо просто записувати як (x, y, z). Систему координат у вікні – координати проєкції позначимо як ($X_{projection}$, $Y_{projection}$, $Z_{projection}$), причому вісь $Z_{projection}$ спрямовується перпендикулярно площині проєктування на глядача, тобто на нас з вами.



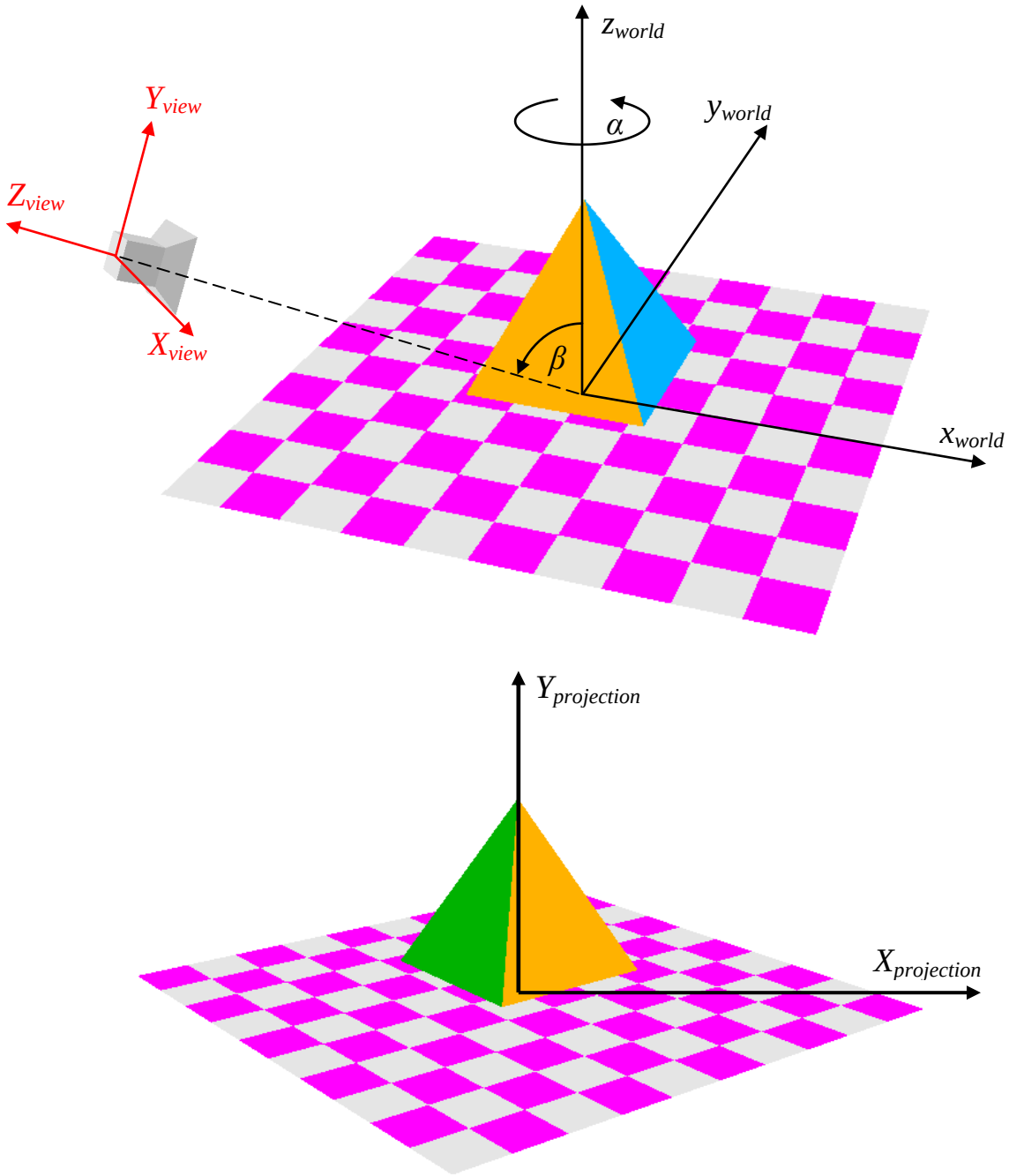
Об'єкти у світовій системі координат.
Приклад розташування точки огляду:
вид зверху вздовж осі z_{world}



Відображення об’єктів у вікні застосунку.
Вид зверху

Видові перетворення координат

Перший приклад. Камера дивиться під кутами огляду α та β у центр світових координат з відстані d

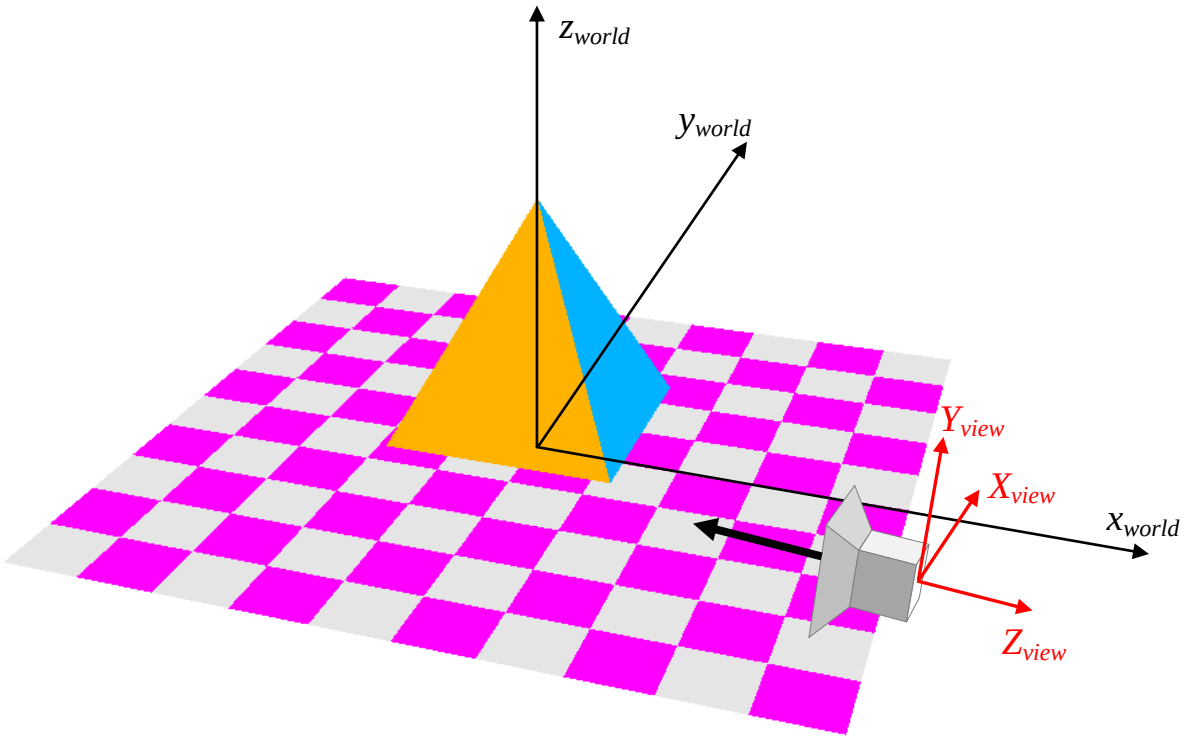


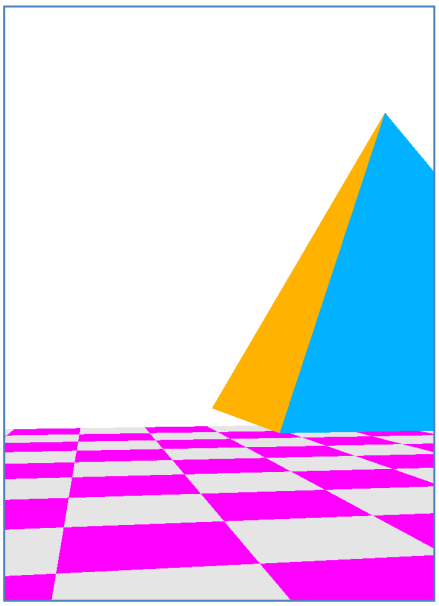
Camera view
$\alpha = -38°$, $\beta = 70°$

Видове перетворення для камери

```
Matrix.setIdentityM(viewMatrix, 0);
Matrix.translateM(viewMatrix, 0, 0, 0, -viewDistance);      //Z-axis shift
Matrix.rotateM(viewMatrix, 0, -betaViewAngle, 1, 0, 0);     //rotation around X
Matrix.rotateM(viewMatrix, 0, -alphaViewAngle, 0, 0, 1);    //rotation around Z
```

Другий приклад. Камера розташовується у довільній точці зі світовими координатами (x, y, z) і нахилена на кути α та β





Camera view

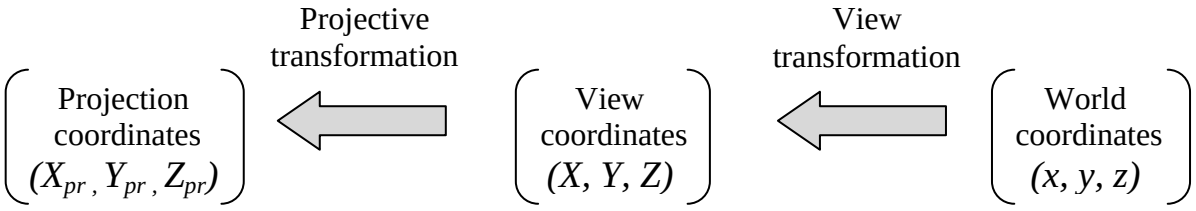
кути: $\alpha = 80°$, $\beta = 95°$

Видове перетворення для камери

```
Matrix.setIdentityM(viewMatrix, 0);
Matrix.rotateM(viewMatrix, 0, -betaViewAngle, 1, 0, 0);          //rotation around X
Matrix.rotateM(viewMatrix, 0, -alphaViewAngle, 0, 0, 1);         //rotation around Z
Matrix.translateM(viewMatrix, 0, -xCamera, -yCamera, -zCamera);  //camera location
```

Узагальнена схема перетворень

Послідовність перетворень координат для відображення тривимірних просторових сцен може бути описана так:



а також у матричній формі:

$$\begin{pmatrix} X_{pr} \\ Y_{pr} \\ Z_{pr} \\ 1 \end{pmatrix} = \begin{pmatrix} \text{Matrix of} \\ \text{Projection} \\ \text{transform} \end{pmatrix} \begin{pmatrix} \text{Matrix of} \\ \text{View} \\ \text{transform} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

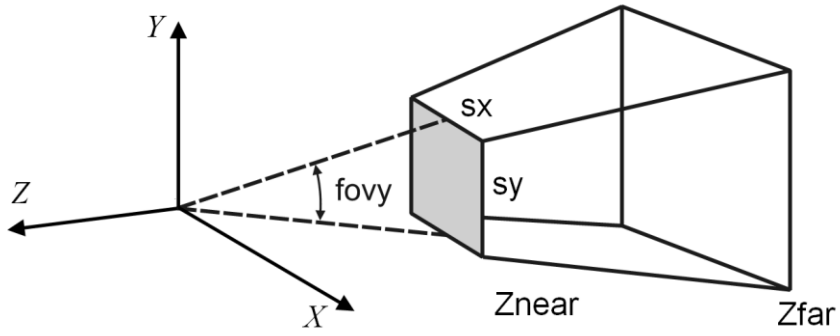
Можна сказати, що такі перетворення описують відображення статичної сцени у деякій проєкції. Але у режимі анімації, коли просторові об'єкти (або їхні окремі частини) можуть рухатися – пересуватися, обертатися тощо, тоді для моделювання їхніх рухів потрібно виконувати вже локальні, окремі трансформації об'єктів. Наприклад, якщо якійсь об'єкт повертається, то для відображення цього спочатку потрібно виконати відповідну трансформацію – назвемо її *модельною*. Якщо таку трансформацію можна описати матрицею, тоді послідовність перетворень набуває наступного вигляду

$$\begin{pmatrix} X_{pr} \\ Y_{pr} \\ Z_{pr} \\ 1 \end{pmatrix} = \begin{pmatrix} \text{Matrix of} \\ \text{Projection} \\ \text{transform} \end{pmatrix} \begin{pmatrix} \text{Matrix of} \\ \text{View} \\ \text{transform} \end{pmatrix} \begin{pmatrix} \text{Matrix of} \\ \text{Model} \\ \text{transform} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Визначення проєкції в OpenGL ES

Для створення перспективної (центральної) проєкції зручно використовувати метод perspectiveM( ) класу Matrix

```
void perspectiveM (float[] m,
                   int offset,
                   float fovy,
                   float aspect,
                   float Znear,
                   float Zfar);
```



Точка огляду – центр сходження променів проєктування – розташовується в (0, 0, 0) видових координат. Якщо видова матриця одинична, то видові координати співпадають зі світовими.

Для центральної проєкції видимий об’єм – це конус. Конус відсікається по глибині площинами **Znear** і **Zfar**. Значення параметрів Znear і Zfar визначають відстані до передньої та задньої меж відсікання по глибині конуса видимості. Величини Znear і Zfar повинні бути завжди більше 0.

Параметр **aspect** означає відношення sx/sy ширини та висоти вікна рендерингу. Це використовується для збереження пропорцій об’єктів.

Параметр **fovy** означає вертикальний кут конусу огляду. Визначається у градусах.

Приклад визначення матриці проекції надано нижче

```
//projection matrix definition
float aspect = (float) width / height;
Matrix.perspectiveM(projectionMatrix, 0, 45, aspect, 0.1f, 30);
```

Тест глибини

Для коректного відображення тривимірних об'єктів та сцен потрібно показувати лише видимі точки об'єктів. Якщо на лінії спостереження знаходяться, наприклад, дві точки якихось об'єктів, то яку з них потрібно відображати? Якщо не брати до уваги, що деякі об'єкти можуть бути напівпрозорими, то потрібно відображати найближчу з точку по відношенню до точки спостереження. В OpenGL це зветься тестом глибини (*depth test*).

У якості міри глибини можна використати значення координати *Z* видових координат, враховуючи те, що вісь Z видових координат спрямованя на спостерігача. Загалом відомі методи автоматичного знаходження видимих точок об'єктів з використанням так званого *z-буфера*. Цей z-буфер (*depth buffer*) є растром із розмірами відповідно вікна рендерингу, який зберігає значення глибин для усіх пікселів. При виконанні растеризації для кожної точки вікна виконується порівняння її відстані з поточним значенням у z-буфері. Якщо відстань менша, то точка виводиться у основний растр зображення, а її глибина записується у z-буфер як нове поточне значення.

Як можна це запрограмувати? Підтримка буфера глибини забезпечується апаратно графічним процесором. Open GL надає інтерфейс у вигляді наступних функцій

```
GLES32.glEnable(GLES32.GL_DEPTH_TEST);
```

Це варто робити при визначенні параметрів поверхні рендерингу. А далі, при кожному виклику метода **onDrawFrame( )** перед малюванням об’єктів потрібно спочатку очистити основний растр вікна рендерингу, а також очистити буфер глибини

```
GLES32.glClear(GLES32.GL_COLOR_BUFFER_BIT |
GLES32.GL_DEPTH_BUFFER_BIT);
```

Усе це включено до нового шаблону програмного коду для виконання завдань по програмуванню графіки OpenGL ES.

Шаблон програмного коду

Пропонується шаблон програмного коду для лаб. №3, який можна назвати Шаблон 5. Значною мірою він використовує текст шаблону для OpenGL ES, використаному у попередній лаб. №2.

MainActivity.java

```
import android.content.Context;
import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

import android.opengl.GLSurfaceView;
import android.opengl.GLES32;
import android.view.Menu;
import android.view.MenuItem;
import android.view.MotionEvent;

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

public class MainActivity extends AppCompatActivity {
    private GLSurfaceView gLView;
    private myWorkMode wmRef = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        gLView = new MyGLSurfaceView(this);
        setContentView(gLView);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        menu.add(0, 1, 0, "First menuitem");
        //... add other menuitems
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        setTitle(item.getTitle());
        switch (item.getItemId()) {
            case 1:
                myModeStart(new myFirstMode(),
                            GLSurfaceView.RENDERMODE_WHEN_DIRTY);
                      // or GLSurfaceView.RENDERMODE_CONTINUOUSLY);
                return true;
            // ... responding to selection of other menu items
            default:break;
        }
        return super.onOptionsItemSelected(item);
    }

    public void myModeStart(myWorkMode wmode, int rendermode) {
        wmRef = wmode;
        gLView.setRenderMode(rendermode);
        gLView.requestRender();
    }
```

```
    public class MyGLSurfaceView extends GLSurfaceView {

        public MyGLSurfaceView(Context context){
            super(context);
            // Create an OpenGL context
            setEGLContextClientVersion(2);      // or (3)
            // Set the Renderer for drawing on the GLSurfaceView
            setRenderer(new MyGLRenderer());
            setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);  //default
        }

        @Override
        public boolean onTouchEvent(MotionEvent e) {
            if (wmRef == null) return false; //workmode object is not created
            if (wmRef.onTouchNotUsed()) return false;
            int cx = this.getWidth();
            int cy = this.getHeight();
            float xtouch = e.getX();
            float ytouch = e.getY();
            switch (e.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    if (wmRef.onActionDown(xtouch, ytouch, cx, cy))
                        requestRender();
                    break;
                case MotionEvent.ACTION_MOVE:
                    if (wmRef.onActionMove(xtouch, ytouch, cx, cy))
                        requestRender();
                    break;
                default:break;
            }
            return true;
        }
    }

    public class MyGLRenderer implements GLSurfaceView.Renderer {
        private int myRenderHeight = 1, myRenderWidth = 1;

        public void onSurfaceCreated(GL10 unused, EGLConfig config) {
            // Set the background frame color (dark blue for example)
            GLES32.glEnable(GLES32.GL_DEPTH_TEST);
            GLES32.glClearColor(0, 0, 0.3f, 1.0f);
        }

        public void onSurfaceChanged(GL10 unused, int width, int height) {
            GLES32.glViewport(0, 0, width, height);
            myRenderWidth = width;
            myRenderHeight = height;
        }

        public void onDrawFrame(GL10 unused) {
            GLES32.glClear(GLES32.GL_COLOR_BUFFER_BIT |
                           GLES32.GL_DEPTH_BUFFER_BIT);
            if (wmRef == null) return;
            if (wmRef.getProgramId() < 0) return;    //previous call error
            if (wmRef.getProgramId() == 0)  //for the first onDrawFrame call
                wmRef.myCreateShaderProgram();
            if (wmRef.getProgramId() <= 0) return;     //an error
            GLES32.glUseProgram(wmRef.getProgramId());
            wmRef.myUseProgramForDrawing(myRenderWidth, myRenderHeight);
        }
    }
}
```

myWorkMode.java

```
import android.opengl.GLES32;
import static android.opengl.GLES32.GL_COMPILE_STATUS;

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

//The base class for various concrete OpenGL ES examples
public class myWorkMode {
    protected int gl_Program = 0;

    protected int VAO_id = 0;            //Vertex Array Object
    protected int VBO_id = 0;            //Vertex Buffer Object

    protected float[] arrayVertex = null;
    protected int numVertex = 0;

    protected float alphaViewAngle = 0;
    protected float betaViewAngle = 0;

    protected float[] modelMatrix;   //matrix for objects transformations
    protected float[] viewMatrix;            //matrix for camera view
    protected float[] projectionMatrix;      //matrix for projection


    myWorkMode() {
        gl_Program = 0;
        VAO_id = VBO_id = 0;
        modelMatrix = new float[16];
        viewMatrix = new float[16];
        projectionMatrix = new float[16];
    }

    public int getProgramId() {
        return gl_Program;
    }

    protected int myCompileShader(int shadertype, String shadercode) {
        int shader_id = GLES32.glCreateShader(shadertype);
        GLES32.glShaderSource(shader_id, shadercode);
        GLES32.glCompileShader(shader_id);
        //check shader compiling errors
        int[] res = new int[1];
        GLES32.glGetShaderiv(shader_id, GL_COMPILE_STATUS, res, 0);
        if (res[0] != 1) return 0;
        return shader_id;
    }

    protected void myCompileAndAttachShaders(String vsh, String fsh) {
        gl_Program = -1;      //shader compiling error by default
        int vertex_shader_id = myCompileShader(GLES32.GL_VERTEX_SHADER, vsh);
        if (vertex_shader_id == 0) return;     //shader compiling error
        int fragment_shader_id = myCompileShader(GLES32.GL_FRAGMENT_SHADER, fsh);
        if (fragment_shader_id == 0) return;   //shader compiling error

        gl_Program = GLES32.glCreateProgram();
        GLES32.glAttachShader(gl_Program, vertex_shader_id);
        GLES32.glAttachShader(gl_Program, fragment_shader_id);
        GLES32.glLinkProgram(gl_Program);
        GLES32.glDeleteShader(vertex_shader_id);     //no longer needed
        GLES32.glDeleteShader(fragment_shader_id);
    }

    protected void getId_VAO_VBO() {
        int[] tmp = new int[2];
        GLES32.glGenVertexArrays(1, tmp, 0);
        VAO_id = tmp[0];              //Vertex Array Object id
```

```
        GLES32.glGenBuffers(1, tmp, 0);
        VBO_id = tmp[0];              //Vertex Buffer Object id
    }

    protected void myVertexArrayBind(float[] src, String atrib) {
        if (gl_Program <= 0) return;
        ByteBuffer bb = ByteBuffer.allocateDirect(src.length*4);
        bb.order(ByteOrder.nativeOrder());

        FloatBuffer vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(src);
        vertexBuffer.position(0);

        getId_VAO_VBO();

        // Bind the Vertex Array Object first
        GLES32.glBindVertexArray(VAO_id);

        //then bind and set vertex buffer(s) and attribute pointer(s)
        GLES32.glBindBuffer(GLES32.GL_ARRAY_BUFFER, VBO_id);
        GLES32.glBufferData(GLES32.GL_ARRAY_BUFFER,src.length*4, vertexBuffer,
                            GLES32.GL_STATIC_DRAW);

        int handle = GLES32.glGetAttribLocation(gl_Program, atrib);
        GLES32.glEnableVertexAttribArray(handle);
        GLES32.glVertexAttribPointer(handle,3, GLES32.GL_FLOAT,false,3*4,0);
        GLES32.glEnableVertexAttribArray(0);

        GLES32.glBindBuffer(GLES32.GL_ARRAY_BUFFER, 0);
        GLES32.glBindVertexArray(0);
    }

    protected void myVertexArrayBind2(float[] src, int stride,
                                      String atrib1, int offset1,
                                      String atrib2, int offset2) {
        if (gl_Program <= 0) return;
        ByteBuffer bb = ByteBuffer.allocateDirect(src.length*4);
        bb.order(ByteOrder.nativeOrder());

        FloatBuffer vertexBuffer = bb.asFloatBuffer();
        vertexBuffer.put(src);
        vertexBuffer.position(0);

        getId_VAO_VBO();

        // Bind the Vertex Array Object first
        GLES32.glBindVertexArray(VAO_id);

        //then bind and set vertex buffer
        GLES32.glBindBuffer(GLES32.GL_ARRAY_BUFFER, VBO_id);
        GLES32.glBufferData(GLES32.GL_ARRAY_BUFFER,src.length*4, vertexBuffer,
                            GLES32.GL_STATIC_DRAW);

        //then define attribute pointers
        int handle = GLES32.glGetAttribLocation(gl_Program, atrib1);
        GLES32.glEnableVertexAttribArray(handle);
        GLES32.glVertexAttribPointer(handle, 3, GLES32.GL_FLOAT, false,
                                     stride*4, offset1);

        handle = GLES32.glGetAttribLocation(gl_Program, atrib2);
        GLES32.glEnableVertexAttribArray(handle);
        GLES32.glVertexAttribPointer(handle, 3, GLES32.GL_FLOAT, false,
                                     stride*4, offset2);

        GLES32.glEnableVertexAttribArray(0);
        GLES32.glBindBuffer(GLES32.GL_ARRAY_BUFFER, 0);
        GLES32.glBindVertexArray(0);
    }
```

```
    public void myCreateShaderProgram() { }

    protected void myCreateScene() { }

    public void myUseProgramForDrawing(int width, int height) {
        //Default implementation for simply triangulated scenes
        GLES32.glBindVertexArray(VAO_id);
        GLES32.glDrawArrays(GLES32.GL_TRIANGLES, 0,numVertex);
        GLES32.glBindVertexArray(0);
    }

    public boolean onTouchNotUsed() { return true; }

    public boolean onActionDown(float x, float y, int cx, int cy) { return false; }

    public boolean onActionMove(float x, float y, int cx, int cy) { return false; }
}
```

Рекомендації щодо програмування рендерингу 3D сцен

Спочатку стосовно шейдерів. Шейдер вершин буде забезпечувати перемноження координат вершин на матриці: на видову матрицю, матрицю проекції та на матрицю попереднього перетворення об’єктів (зокрема обертання одного з об’єкту відповідно завданню лаб. роботи). Шейдер вершин буде підтримувати такий формат вершин – разом з координатами у масиві вершин будуть вказуватися індивідуальні кольори вершин, як це було у попередній лаб. №2. Нижче наведено текст шейдера вершин

```
#version 300 es
uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjMatrix;
in vec3 vPosition;
in vec3 vColor;
out vec3 outColor;
void main() {
  gl_Position = uProjMatrix * uViewMatrix * uModelMatrix * vec4(vPosition, 1.0f);
  outColor = vColor;
}
```

Фрагментний шейдер використаємо той самий, що і у попередній лаб.2

```
#version 300 es
precision mediump float;
in vec3 outColor;
out vec4 resultColor;
void main() {
   resultColor = vec4(outColor, 1.0f);
}
```

Далі наведемо приклад, як можна організувати формування матриці проекції, видової та модельної матриць та передачу їх у шейдер вершин в ході виконання рендерингу сцени.

```
public void myUseProgramForDrawing(int width, int height) {
    Matrix.setIdentityM(modelMatrix, 0);

    //view matrix definition
    Matrix.setIdentityM(viewMatrix, 0);
    Matrix.rotateM(viewMatrix, 0, -betaViewAngle, 1, 0, 0);  //rotation around X
    Matrix.rotateM(viewMatrix, 0, -alphaViewAngle, 0, 0, 1);  //rotation around Z
    Matrix.translateM(viewMatrix, 0, -xCamera, -yCamera, -zCamera);  //camera location

    //projection matrix definition
    float aspect = (float) width / height;
    Matrix.perspectiveM(projectionMatrix, 0, 45, aspect, 0.1f, 30);

    //pass the transformation matrices to the shader
    int vMatrixHandle = GLES32.glGetUniformLocation(gl_Program, "uViewMatrix");
    GLES32.glUniformMatrix4fv(vMatrixHandle, 1, false, viewMatrix, 0);
    vMatrixHandle = GLES32.glGetUniformLocation(gl_Program, "uProjMatrix");
    GLES32.glUniformMatrix4fv(vMatrixHandle, 1, false, projectionMatrix, 0);
    vMatrixHandle = GLES32.glGetUniformLocation(gl_Program, "uModelMatrix");
    GLES32.glUniformMatrix4fv(vMatrixHandle, 1, false, modelMatrix, 0);
    //drawing
    GLES32.glBindVertexArray(VAO_id);

    GLES32.glDrawArrays(...);

    //optional model transformation for some objects
    //for example rotation around Z
    Matrix.rotateM(modelMatrix, 0, rotationAngle, 0, 0, 1);
    GLES32.glUniformMatrix4fv(vMatrixHandle, 1, false, modelMatrix, 0);

    GLES32.glDrawArrays(...);   //some objects drawing

    GLES32.glBindVertexArray(0);
}
```

Якщо у режимі рендерингу *RENDERMODE_CONTINUOUSLY* потрібно запрограмувати якийсь рухомий об'єкт, який, безперервно рухається – наприклад, обертається, то рекомендується використати прийом, описаний у попередній лаб. №2 для імітації періодичної зміни яскравості. Але тут можна використати виміри часу для змін кута повороту потрібного об'єкта і потім відповідно сформувати матрицю modelMatrix для відповідного додаткового перетворення координат у шейдері вершин перед відображенням цього об'єкта.

Підтримка сенсора екрану для керування анімацією

Для забезпечення підтримки сенсорного вводу – натискування на сенсорний екран та пересування вказівника по екрану, у класі **MyGLSurfaceView** потрібно зробити перевизначення метода **onTouchEvent**.

У шаблоні програмного коду це зроблено у наступний спосіб. Звернемо увагу на такі фрагменти

```
@Override
public boolean onTouchEvent(MotionEvent e) {
    if (wmRef == null) return false;   //workmode object is not created
    if (wmRef.onTouchNotUsed()) return false;
    // . . .
    switch (e.getAction()) {
        case MotionEvent.ACTION_DOWN:
             if (wmRef.onActionDown(xtouch, ytouch, cx, cy))
                 requestRender();
             break;
        case MotionEvent.ACTION_MOVE:
             if (wmRef.onActionMove(xtouch, ytouch, cx, cy))
                 requestRender();
             break;
        default:break;
    }
    return true;
}
```

Для того, щоб усе це працювало, потрібно створити екземпляр об'єкта класу, який є похідним від myWorkMode і описує конкретний приклад завдання. У цьому похідному класі необхідно визначити методи onActionDown() (натискування у певному місці сенсорного екрану) та метод onActionMove() (пересування стілуса по сенсорному екрану).

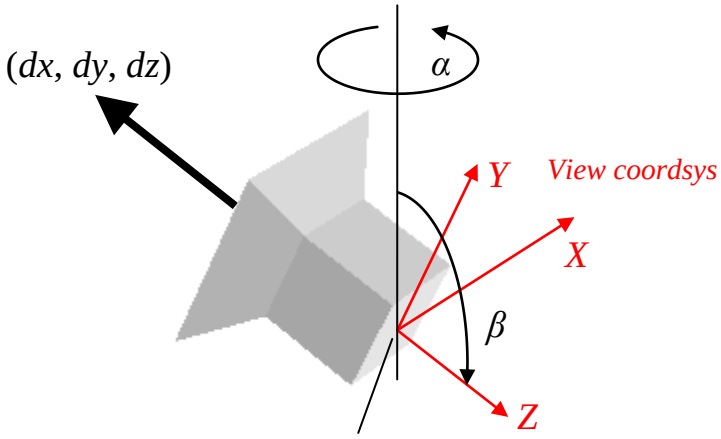
Нижче наведено приклад програмного коду цих методів

```
public boolean onActionDown(float x, float y, int cx, int cy) {
    xTouchDown = x;
    yTouchDown = y;
    alphaAnglePrev = alphaViewAngle;
    viewDistancePrev = viewDistance;
    return false;
}

public boolean onActionMove(float x, float y, int cx, int cy) {
    alphaViewAngle = alphaAnglePrev + Kalpha * (xTouchDown - x);
    viewDistance = viewDistancePrev - Kdist * (yTouchDown - y);
    return true;
}
```

У даному прикладі кут (α) камери змінюється, якщо стілус рухається по горизонталі (x) праворуч або ліворуч. Також відстань камери до об’єктів змінюється, якщо стілус рухати по вертикалі (y). У наведеному тут прикладі використані перемінні – коефіцієнт чутливості для змін кута повороту

камери (Kalpha) та коефіцієнт для змін відстані камери (Kdist). Знайдіть прийнятні для вас значення цих коефіцієнтів чутливості. Підказка: мабуть, ці значення знаходяться у діапазоні 0.1 – 0.01.

Як керувати камерою, яка може рухатися довільною траєкторією у тривимірному просторі? Тут може бути багато варіантів. Так, зокрема, можна спробувати реалізувати спрощену імітацію літального аппарату. Наприклад, якщо визначити, що при зміні положення стілуса вздовж вертикалі буде рух уперед або назад у напрямку осі камери. Нагадаємо, що напрямок камери запропоновано описувати у вигляді кутів α та β, причому α – це повороти в горизонтальній площині навколо осі z, а кут β – це нахил камери



Компоненти (*dx*, *dy*, *dz*) вектора напрямку руху відносно поточного розташування камери (*xCamera*, *yCamera*, *zCamera*) можна представити як

$$dx = -\text{step}\sin(\alpha),$$
$$dy = \text{step}\cos(\alpha),$$
$$dz = -\text{step}\cos(\beta),$$

де step – коефіцієнт руху за одиницю часу. Він може визначатися у залежності від чутливості реагування на пересування стілуса, наприклад:

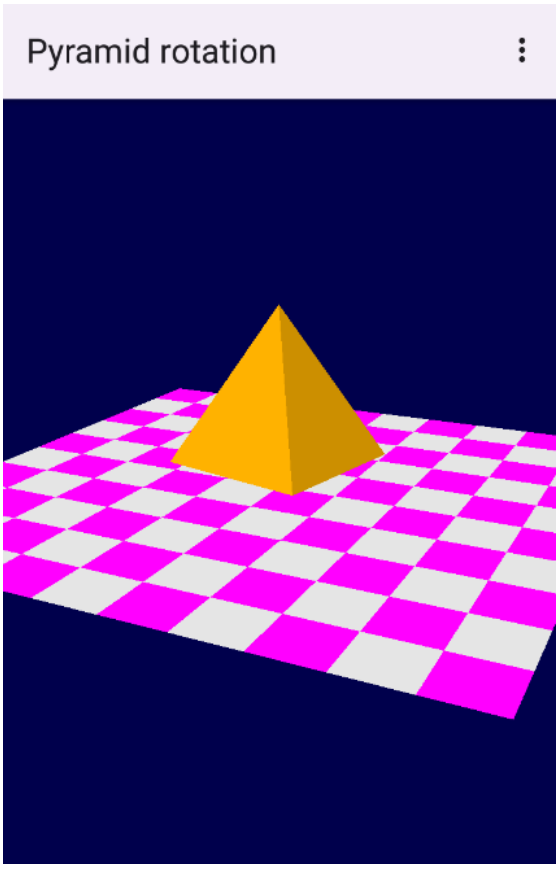
```
public boolean onActionMove(float x, float y, int cx, int cy) {
    //. . .  зміна alphaViewAngle при горизонтальному руху стілуса

    float step = Speed *(yTouchDown - y);
    xCamera -= step*(float)Math.sin(0.017453*alphaViewAngle);
    yCamera += step*(float)Math.cos(0.017453*alphaViewAngle);
    zCamera -= step*(float)Math.cos(0.017453*betaViewAngle);
    yTouchDown = y;
    return true;
}
```

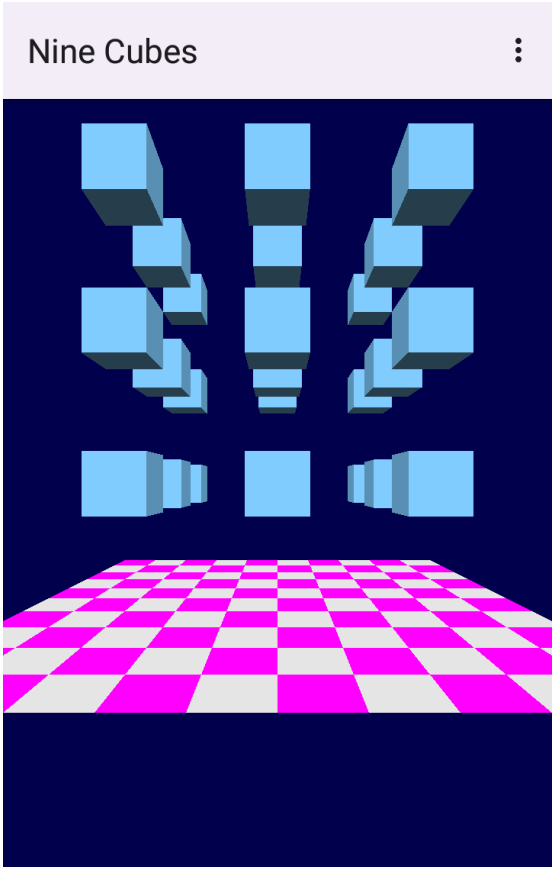
Прийнятне значення для коефіцієнту Speed знайдіть експериментально самостійно. Підказка: мабуть його значення суттєво менше 1.

Варіанти завдань та основні вимоги

1. Застосунок **Lab3_GLES** для вибору режиму роботи повинен мати меню з двома пунктами:

- Pyramid rotation
- Nine Cubes





2. Обрати кольори фону, шахового поля, піраміди, кубів, вертикальний кут конусу огляду – на свій розсуд, продемонструвавши, як їх можна змінити.

3. У режимі Pyramid rotation забезпечити безперервне обертання піраміди вертикалі над шаховим полем. Рендеринг має бути у режимі *RENDERMODE_CONTINUOUSLY*.

4. Запрограмувати, щоб у режимі Pyramid rotation можна було б за допомогою пересування стілуса (пальця) по екрану змінювати ракурс показу сцени наступним чином:

- обертати камеру навколо вертикальної осі (змінювати кут α)
- наближати-віддаляти камеру відносно центру сцени

При будь-яких змінах ракурсу показу камера постійно повинна дивитися у центр сцени.

5. У режимі Nine Cubes показ сцени статичний, з постійним ракурсом – якщо не торкатися сенсорного екрану. Сцена складається з шахового поля та решіткі з 27 кубів. Рендеринг у режимі *RENDERMODE_WHEN_DIRTY*.

6. Запрограмувати, щоб у режимі Nine Cubes можна було б за допомогою пересування стілуса (пальця) по екрану змінювати ракурс показу сцени наступним чином (імітувати рух на літальному апараті):

- рухатися вперед-назад вздовж напрямку зору камери
- робити повороти вправо-вліво,
- змінювати нахил камери уверх-вниз і потім відповідно рухатися вздовж нового напрямку зору камери

7. У режимі Nine Cubes продемонструвати проходження користувача програми серед кубів без наїзду на них.

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний тексти
4. Ілюстрації (скріншоти)
5. Висновки

Контрольні запитання

1. Що таке видова система координат?
2. Як виконуються перетворення координат у шейдері вершин?
3. Як визначається ракурс показу сцени і як його змінити?
4. Як вказати межі конусу видимості для перспективної проєкції?
5. Як врахувати співвідношення розмірів вікна виводу для компенсації викривлень пропорцій форми об'єктів?
6. Як визначити вертикальний кут конусу видимості камери?
7. Як запрограмуванти якісь дії на пересування стілуса (пальця) по екрану?