Лабораторна робота №5. Використання текстур

Мета: отримати навички програмування текстур тривимірних об’єктів засобами графіки OpenGL ES.

У цій лабораторній роботі будуть розглянуті питання програмування рендерингу реалістичних зображень тривимірних об’єктів шляхом накладання текстур.

Завдання

Потрібно створити у середовищі Android Studio проєкт з ім’ям **Lab5_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

Основні поняття текстурування

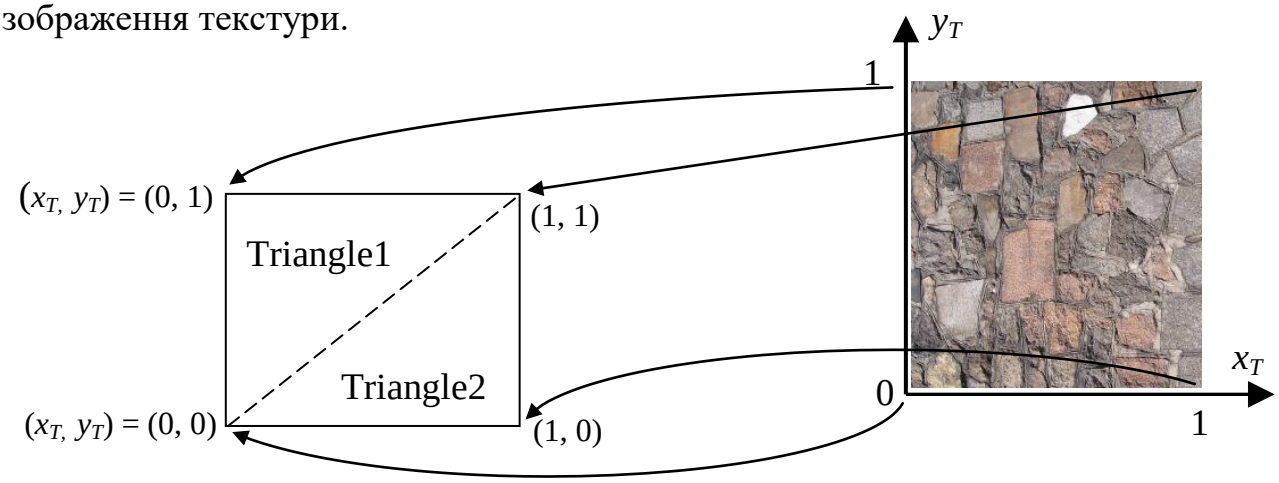
Текстурування можна уявити собі як наклеювання шпалер на поверхню. Для текстур використовуються растрові зображення.

Бажано, щоб розміри растру текстури дорівнювали степеню двійки. Наприклад: 64x64, 1024x256, 1024x2048 тощо. У деяких ранніх версіях OpenGL ця вимога була обов’язковою, тому при виготовлення файлів текстур до завантаження доводилося розтягувати-стискувати або обрізати растри зображень для нормалізації їхніх розмірів. Як здається, і натепер намагаються дотримуватися традицій щодо розмірів зображень текстур.

Таким чином, спочатку треба заздалегідь підготувати потрібні файли зображень текстур. Після виклику застосунку завантажити текстури. А потім ввести у масив вершин координати – просторові та текстурні. Крім того, треба завантажити відповідні шейдери, у яких прописаний програмний код підтримки текстурування.

Визначення текстурних координат вершин граней

Визначення текстурних координат (x_T, y_T) для вершин граней означатиме встановлення відповідності точок грані точкам растрового зображення текстури.



Результат рендерингу текстурованого чотирикутника

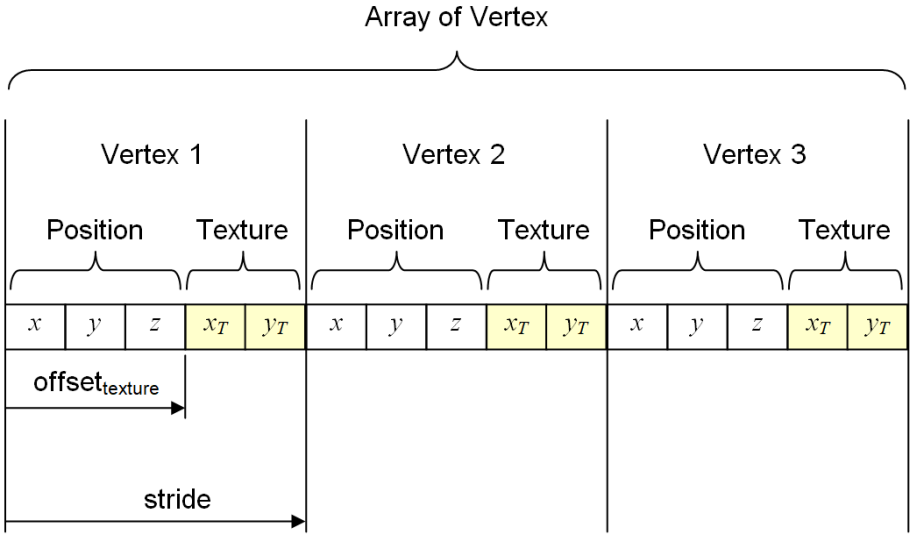
Горизонтальна вісь текстурних координат (x_T) спрямовується зліва направо, вертикальна (y_T) – знизу догори. Значення компонент (x_T, y_T) знаходяться у діапазоні від 0 до 1 кожна.

Як вказано на рисунку вище, значення (x_T, y_T) = (0, 0) у лівому нижньому куті текстури, а значення (x_T, y_T) = (1, 1) – у правому верхньому куті. Проте доволі часто потрібно при прив'язуванні растрових зображень доводиться використовувати зворотний напрям осі (y_T). Це пояснюється тим, що у деяких растрових графічних форматах растр записується в порядку від верхніх рядків до нижніх.

Таким чином, для кожної вершини потрібно зберігати окрім значень 3d просторових координат (x, y, z) ще й відповідні значення координат (x_T, y_T) у певному форматі. Для цього потрібно визначити формат масиву вершин.

Мінімальний формат масиву вершин

Мінімальний варіант формату масиву вершин – це 3d координати позиціонування + текстурні координати



Шейдери для мінімального варіанту формату масиву вершин

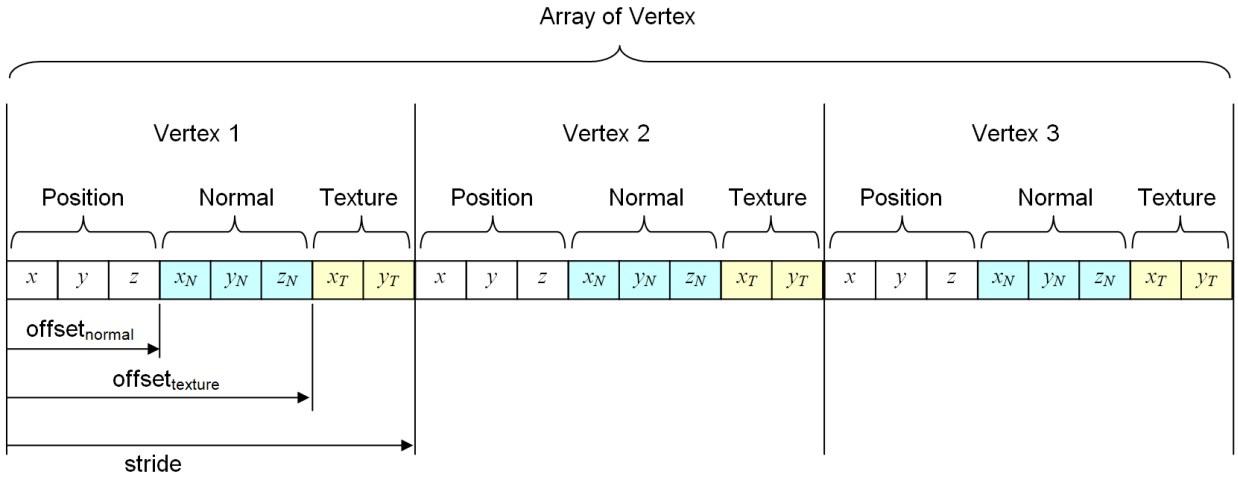
Vertex shader

```
#version 300 es
in vec3 vPosition;
in vec2 vTexture;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjMatrix;
out vec2 currentTexCoord;

void main() {
   gl_Position = uProjMatrix*uViewMatrix*uModelMatrix*vec4(vPosition, 1.0f);
   currentTexCoord = vec2(vTexture.x, vTexture.y);
}
```

Fragment shader

```
#version 300 es
precision mediump float;
in vec2 currentTexCoord;
uniform sampler2D vTextureSample;
out vec4 resultColor;

void main() {
    resultColor = texture(vTextureSample, currentTexCoord);
}
```

Формат масиву вершин: координати + нормалі + текстури



Поєднання координат вершин, нормалей та текстурних координат у єдиному масиві вершин

Шейдери для розширеного формату масиву вершин

Vertex shader

```
#version 300 es
in vec3 vPosition;      //вхідний потік даних з масиву вершин
in vec3 vNormal;
in vec2 vTexture;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjMatrix;

out vec3 currentPos;
out vec3 currentNormal;
out vec2 currentTexCoord;

void main() {
   gl_Position = uProjMatrix*uViewMatrix*uModelMatrix*vec4(vPosition, 1.0f);
   currentPos = mat3(uModelMatrix) * vPosition;
   currentNormal = mat3(uModelMatrix) * vNormal;
   currentTexCoord = vec2(vTexture.x, vTexture.y);
}
```

Після завантаження масиву вершин у пам’ять GPU далі вже у процесі рендерингу шейдер вершин сприймає потік координат позиціонування,

координат нормалей та текстурних координат. Далі відбувається інтерполяція нормалей вершин та текстурних координат відповідно поточної точки кожної грані. І потім вже у фрагментному шейдері формується колір піксела відповідно текстурі на який накладається колір відбиття світла.

Fragment shader

```
#version 300 es
precision mediump float;

in vec3 currentPos;
in vec3 currentNormal;
in vec2 currentTexCoord;

uniform sampler2D vTextureSample;

uniform vec3 vColor;    //може бути власний колір об'єкту
// ...   та інші uniform-перемінні, якщо потрібні

out vec4 resultColor;

void main() {
    // ... якісь розрахунки відбиття світла
    // ... з використанням перемінних currentPos, currentNormal та інших
    // позначимо усе це як SomeFunc(..)
    vec3 vClr = SomeFunc(vColor, ...);
    resultColor = texture(vTextureSample,currentTexCoord) * vec4(vClr, 1.0f);
}
```

Щодо можливих різновидів форматів масиву вершин. Необхідно відзначити, що загалом можливі різноманітні комбінації компонент (атрибутів) масиву вершин, таких як: світові координати (x, y, z), координати нормалей (x_N, y_N, z_N), текстурних координат (x_T, y_T), та кольорів (R,G,B). Цілком вірогідно, що в окремих випадках може бути успішно застосований навіть максимальний варіант з усіма переліченими вище атрибутами вершин, незважаючи на те, що на вожну вершину буде вже 11 елементів типу float, або якогось іншого типу.

Шаблон програмного коду застосунку

Нижче наводиться наступна версія шаблону застосунку OpenGL ES, який є розвитком можливостей програмування графіки вже з урахуванням текстурування полігональних об’єктів.

Текст файлу **MainActivity.java** майже повністю повторює шаблон попередньої версії, але є невеличкі відмінності. Так, зокрема, суттєво змінився текст метода onDrawFrame

```
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 Context contextMain;
    private myWorkMode wmRef = null;

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
    //... add menuitems
    return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        setTitle(item.getTitle());
        switch (item.getItemId()) {
        // ... responding to selection of menu items
        }
        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;
            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 default background frame color (dark blue for example)
            GLES32.glClearColor(0.0f, 0.0f, 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) {
            if (wmRef != null) wmRef.clearColor();
            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.myCreateScene(contextMain);
             }
            if (wmRef.getProgramId() <= 0) return;     //an error
            GLES32.glUseProgram(wmRef.getProgramId());
            wmRef.myUseProgramForDrawing(myRenderWidth, myRenderHeight);
        }
    }
}
```

Основною відмінністю поточної версії класу myWorkMode є наявність метода myLoadTexture() а також підтримка до трьох атрибутів масиву вершин методом myVertexArrayBind3(). Методи myVertexArrayBind() та myVertexArrayBind2() з цього шаблону вилучені.

Файл **myWorkMode.java**

```
import static android.opengl.GLES20.GL_COMPILE_STATUS;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.opengl.GLES32;
import android.opengl.GLUtils;

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 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;
    }
```

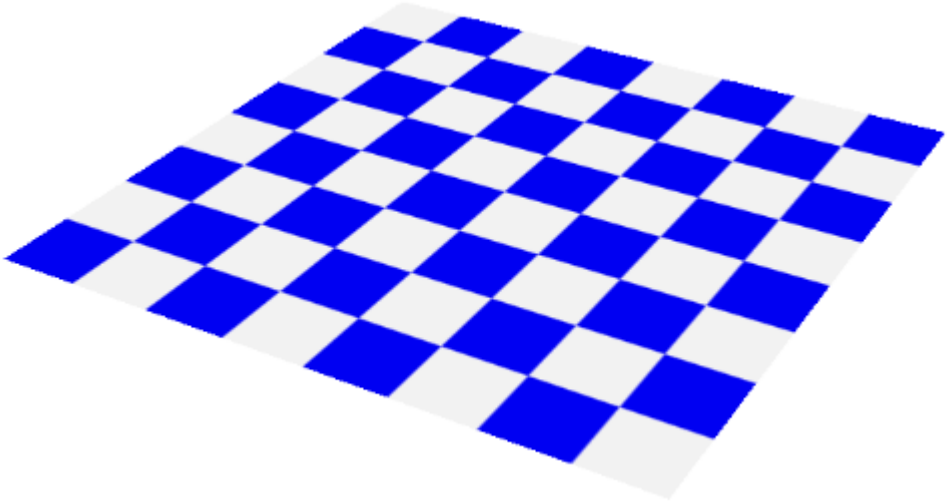
```
protected void myCompileAndAttachShaders(String vsh, String fsh) {
    gl_Program = -1;      //shader compiling error by default
    int vertex_shader_id = myCompileShader(GLES32.GL_VERTEX_SHADER, vsh);
    if (vertex_shader_id == 0) return;     //shader compiling error
    int fragment_shader_id = 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 myVertexArrayBind3(float[] src, int stride,
                                  String atrib1, int offset1,
                                  String atrib2, int offset2,
                                  String atrib3, int offset3) {
    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;
    if (!atrib1.isEmpty()) {
        handle = GLES32.glGetAttribLocation(gl_Program, atrib1);
        GLES32.glEnableVertexAttribArray(handle);
        GLES32.glVertexAttribPointer(handle, 3, GLES32.GL_FLOAT,
                                     false, stride*4, offset1);
    }
    if (!atrib2.isEmpty()) {
        handle = GLES32.glGetAttribLocation(gl_Program, atrib2);
        GLES32.glEnableVertexAttribArray(handle);
        GLES32.glVertexAttribPointer(handle, 3, GLES32.GL_FLOAT,
                                     false, stride*4, offset2);
    }
    if (!atrib3.isEmpty()) {
        handle = GLES32.glGetAttribLocation(gl_Program, atrib3);
```

```
            GLES32.glEnableVertexAttribArray(handle);
            GLES32.glVertexAttribPointer(handle, 3, GLES32.GL_FLOAT,
                                         false,stride*4, offset3);
        }

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

    public int myLoadTexture(Context context, int resourceId, int wrap) {

        //need to create the texture object first using glGenTextures()
        int[] tmp = new int[2];
        GLES32.glGenTextures(1, tmp, 0);
        int handle = tmp[0];

        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inScaled = false;
        Bitmap bitmap = BitmapFactory.decodeResource(context.getResources(),
                                                     resourceId, options);
        //and then bind it using glBindTexture().
        GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, handle);

        GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                    GLES32.GL_TEXTURE_WRAP_S, wrap);
        GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                    GLES32.GL_TEXTURE_WRAP_T, wrap);
        GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                    GLES32.GL_TEXTURE_MIN_FILTER, GLES32.GL_LINEAR);
        GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                    GLES32.GL_TEXTURE_MAG_FILTER, GLES32.GL_LINEAR);

        //upload texture data into GPU memory
        GLUtils.texImage2D(GLES32.GL_TEXTURE_2D, 0, bitmap, 0);

        bitmap.recycle();       //remove tmp bitmap
        GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, 0);
        return handle;
    }

    public void myCreateScene(Context context) {
        myInitTextures(context);
        myCreateObjects();
        myCreateShaderProgram();
    }

    protected void myInitTextures(Context context) { }
    protected void myCreateObjects() { }
    protected void myCreateShaderProgram() { }

    protected void myCreateProjection(int width, int height) { }
    protected void myDrawing() { }

    public void myUseProgramForDrawing(int width, int height) {
        GLES32.glEnable(GLES32.GL_DEPTH_TEST);
        myCreateProjection(width, height);
        GLES32.glBindVertexArray(VAO_id);
        myDrawing();
        GLES32.glBindVertexArray(0);
    }
```

```
    public void clearColor() {
        GLES32.glClearColor(1.0f, 1.0f, 1.0f, 1.0f);  //by default
    }

    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;
    }
}
```

Циклічне повторення зображення текстури

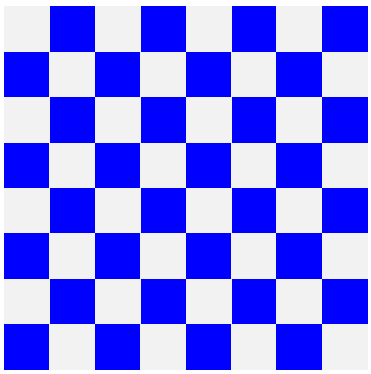
Як отримати зображення такої шахової дошки?



Здавалося б, усе дуже просто. Створимо зображення текстури, наприклад, розмірами 64x64, і запишемо його у файл. Потім завантажимо цю текстуру і визначимо координати чотирикутника у масиві вершин. Якось так

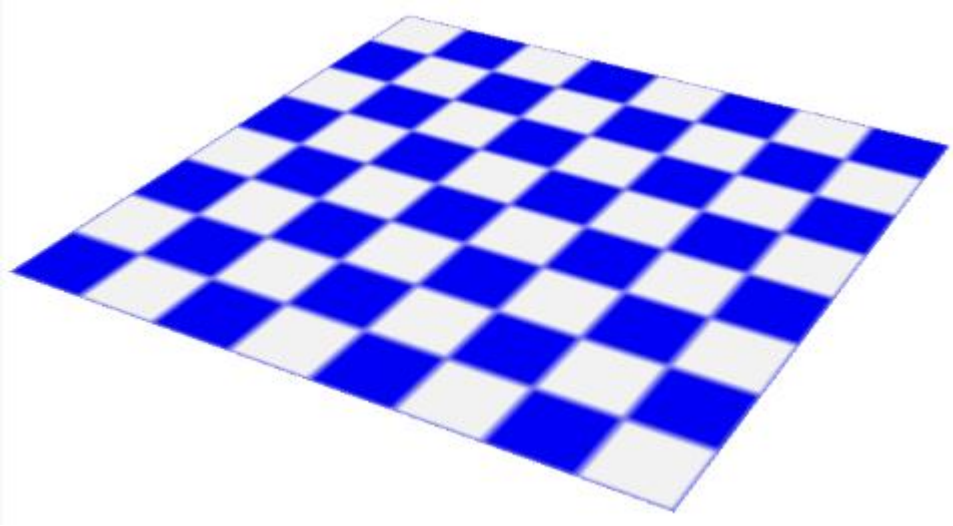
Вміст масиву вершин:

```
float[] aVertex = {
        //position    xt yt
        -1, -1, 0,    0, 1,
        -1,  1, 0,    0, 0,
         1,  1, 0,    1, 0,
         1, -1, 0,    1, 1 };
```



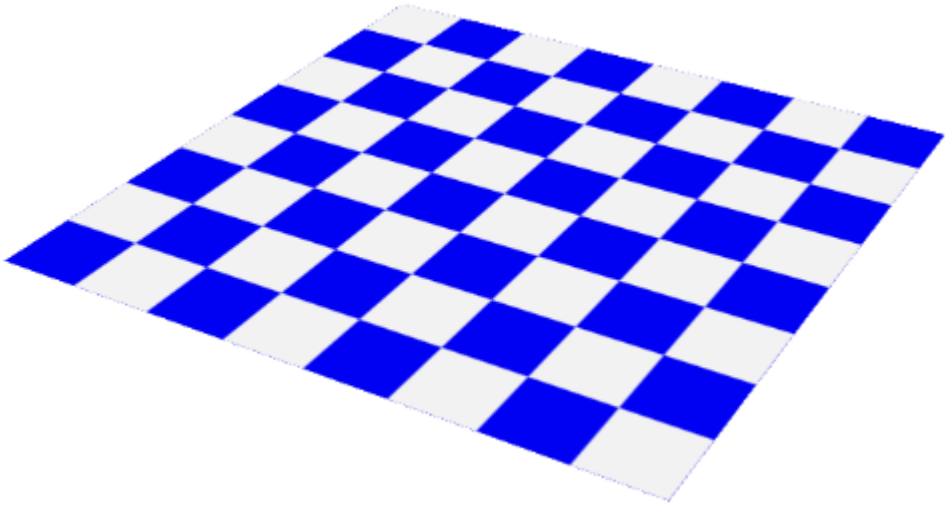
Текстура

Результат рендерингу наведений нижче

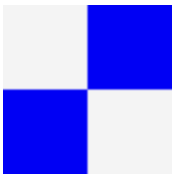


Таке зображення є нечітким, розмитим – це результат фільтрації текстури із невеличкою роздільною здатністю.

Створимо зображення текстури із вчетверо більшою роздільною здатністю – розмірами 256x256 і накладемо на чотирикутник.



Результат вже майже той, що потрібно – але якою ціною? Растрове зображення розмірами 256 x 256 у форматі 24 бітів на піксел потребує відкриття массиву обсягом 256 x 256 x 3 = 196 608 Байтів = 192 КБ. А якщо взагалі відмовитися від текстури і малювати 64 клітинки різноколірними чотирикутниками, то, враховуючи те, що для кожного чотирикутника потрібно 4 вершини, а кожна вершина – це від 5 до 8 (у залежності від формату массиву вершин) чотирибайтових значень типу float, тоді буде від 64x4x5x4 = 5120Б до 64x4x8x4 = 8192Б = 8КБ. Це явно менше 192 КБ.



А тепер спробуємо використати те, що на зображенні шахівниці циклічно повторюються однакові фрагменти. Створимо текстуру розмірами 64x64 із зображенням лише чотирьох суміжних клітинок, як відображено праворуч.

При звичайному прив'язуванні значення текстурних координат (x_T, y_T) знаходяться у діапазоні від 0 до 1. А для того, щоб повторювати зображення текстури на поверхні грані 4x по горизонталі та 4x по вертикалі, треба вказати текстурні координати 0 та 4.

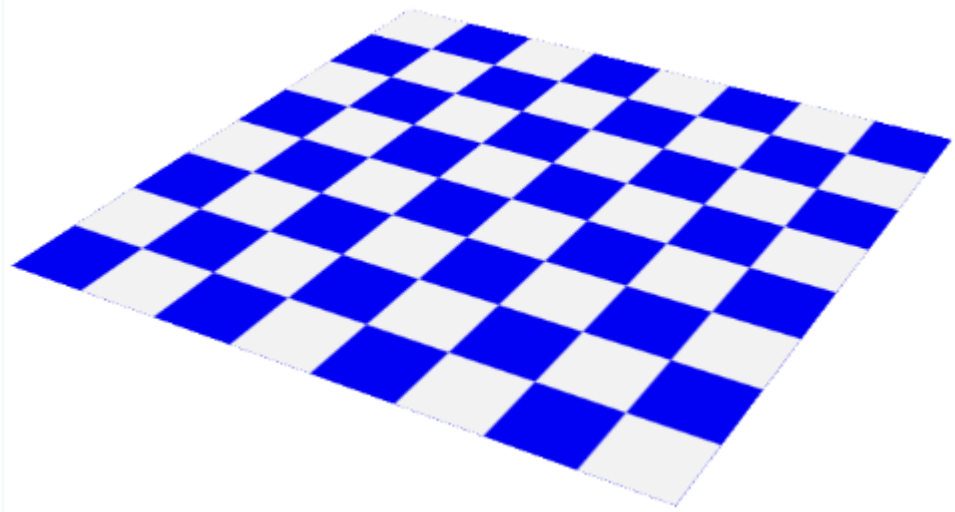
```
float[] aVertex = {
        //position    xt yt
        -1, -1, 0,    0, 4,
        -1,  1, 0,    0, 0,
         1,  1, 0,    4, 0,
         1, -1, 0,    4, 4 };
```

Крім того, потрібно при завантаженні текстури вказати

```
GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                       GLES32.GL_TEXTURE_WRAP_S, GLES32.GL_REPEAT);
GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                       GLES32.GL_TEXTURE_WRAP_T, GLES32.GL_REPEAT);
```

Якщо для завантаження текстур використовувати метод myLoadTexture() з наведеного вище шаблону програмного коду, тоді

```
textureHandle = myLoadTexture(context, R.drawable.img,
                              GLES32.GL_REPEAT);
```



Рекомендація. Для зменшення ефекту небажаного синього контуру у світлих клітин на краях текстури, можна порекомендувати для координат x_T та y_T замість значень 0 вказувати 0.01f, а замість 4 – значення 3.99f.

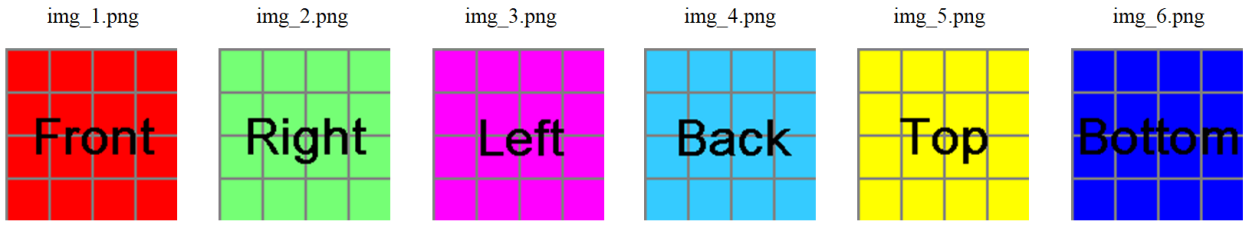
Завантаження й використання багатьох текстур

Зазвичай просторові сцени містять певну кількість об'єктів, які у свою чергу, відображаються множиною трикутників. Кількість трикутних граней загалом може сягати сотень тисяч та навіть мільйонів. І кожна грань може бути текстурованою. Таким чином, при текстуруванні потрібно забезпечити зберігання, своєчасне завантаження відповідної множини текстур та накладання їх на трикутні грані.

Розглянемо деякі аспекти множинності текстур на конкретних прикладах.

Текстурування кубу можна виконати, як мінімум, двома способами.

Перший спосіб. Зберігання кожної текстури у окремому файлі. Це можна вважати універсальним підходом. Так, зокрема, для 6 граней кубу потрібно мати відповідно 6 файлів з іменами, наприклад: img_1.png ... img_6.png.



Завантаження 6 текстур з файлів drawable-ресурсів проєкту Android Studio можна запрограмувати наступним чином

```
texHandles = new int[6];
texHandles[0] = myLoadTexture(context, R.drawable.img_1, w);  //front
texHandles[1] = myLoadTexture(context, R.drawable.img_2, w);  //right
texHandles[2] = myLoadTexture(context, R.drawable.img_3, w);  //left
texHandles[3] = myLoadTexture(context, R.drawable.img_4, w);  //back
texHandles[4] = myLoadTexture(context, R.drawable.img_5, w);  //top
texHandles[5] = myLoadTexture(context, R.drawable.img_6, w);  //bottom
```

Примітка: w = GLES32.GL_CLAMP_TO_EDGE або GLES32.GL_REPEAT

Визначення координат вершин та запис їх у масив вершин. Нехай ім'я масиву вершин буде arrayVertex. Звісно, можна прямо записати координати усіх вершин (включно із текстурними та нормалями) безпосередньо при створенні-ініціалізації масиву arrayVertex. Але це можна вважати доцільним лише тоді, коли уся сцена складається з єдиного кубу. Для складніших сцен варто розглянути більш універсальний підхід.

Уявимо собі, що у нас є власний клас уведення вершин у масив вершин, і у цього класу є метод addQuad( ) для уведення координат вершин для текстурованих чотирикутників. Як здається, для вказування координат вершин можна у методі addQuad( ) передбачити відповідні параметри. Які параметри – цілком на розсуд програміста. Як здається, універсальний метод уведення чотирикутних граней може стати у нагоді у майбутньому, а зараз зосередимося суме на кубі.

Нижче наведений один з можливих варіантів – через тимчасовий масив опису кубу

```
float[] cubeVertexArray = {
      //position     xt yt
       -1, -1, -1,   0, 1,    //front face
       -1, -1,  1,   0, 0,
        1, -1,  1,   1, 0,
        1, -1, -1,   1, 1,

        1, -1, -1,   0, 1,    //right
        1, -1,  1,   0, 0,
        1,  1,  1,   1, 0,
        1,  1, -1,   1, 1,

       -1,  1, -1,   0, 1,    //left
       -1,  1,  1,   0, 0,
       -1, -1,  1,   1, 0,
       -1, -1, -1,   1, 1,

        1,  1, -1,   0, 1,    //back
        1,  1,  1,   0, 0,
       -1,  1,  1,   1, 0,
       -1,  1, -1,   1, 1,

       -1, -1,  1,   0, 1,     //top
       -1,  1,  1,   0, 0,
        1,  1,  1,   1, 0,
        1, -1,  1,   1, 1,

       -1,  1, -1,   0, 1,     //bottom
       -1, -1, -1,   0, 0,
        1, -1, -1,   1, 0,
        1,  1, -1,   1, 1 };

int pos = 0;
for (int i=0; i<6; i++)
    pos = mygp.addQuad(arrayVertex, pos,        //destination vertex array
                       cubeVertexArray, i*20);  //source array
```

Навіщо взагалі присутній додатковий тимчасовий масив cubeVertexArray? Лише для того, щоб викликати метод addQuad( ) у циклі. Але, звісно, можна і по-іншому.

Метод addQuad( ) також може взяти на себе автоматичне обчислення координат нормалей, якщо використовується відповідний формат масиву вершин.

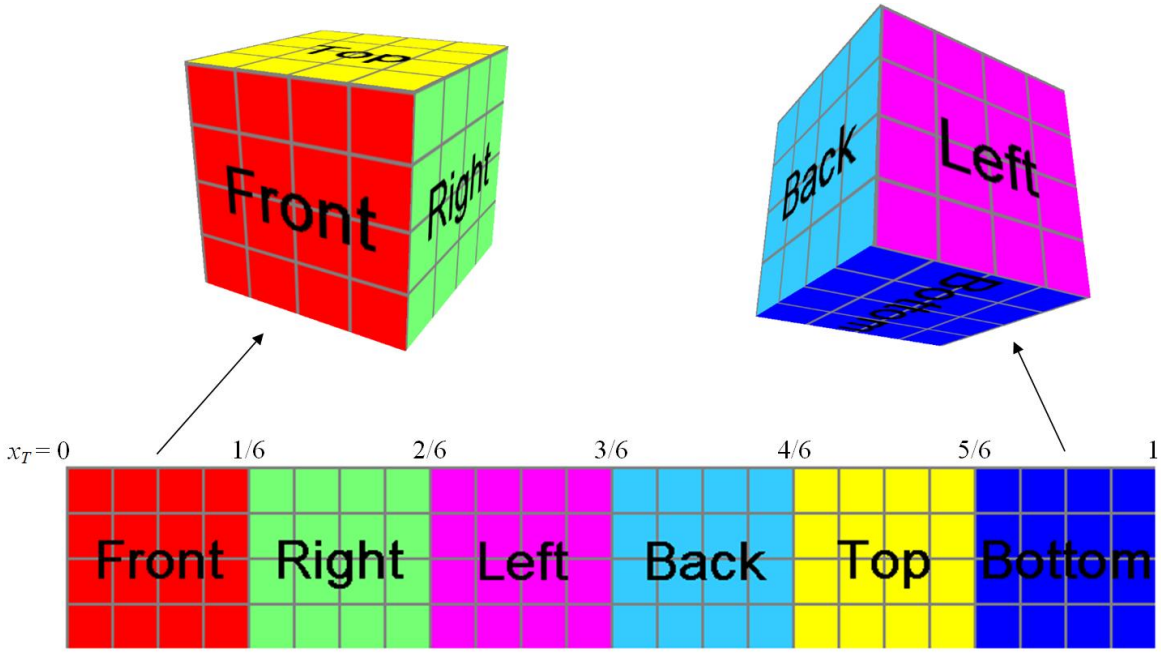
Після того, як усі координати вершин записано у масиві вершин і він завантажений у буфер графічного процесора, можна виконувати рендеринг.

Рендеринг кубу. В ході рендерингу, для того, щоб відобразити один або декілька трикутників з відповідною текстурою, треба перед виконанням glDrawArrays() прікріпити потрібну текстуру викликом glBindTexture( ). Як здається, шість текстур зручніше обробляти у циклі

```
GLES32.glBindVertexArray(VAO_id);

for (int i=0; i<6; i++) {
   GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, texHandles[i]);
   GLES32.glDrawArrays(GL_TRIANGLE_FAN, 4*i, 4);  //or GL_TRIANGLE_STRIP
}

GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, 0);
GLES32.glBindVertexArray(0);
```

Кожна чотирикутна грань кубу представляється двома зв’язаними трикутниками у конфігурації GL_TRIANGLE_FAN або, можливо GL_TRIANGLE_STRIP.

Другий спосіб. Атлас множини текстур у одному файлі. Тут необхідно враховувати, що текстурні координати граней повинні вказувати вже на частки зображення такого атласу.



Зображення усіх текстур для 6 граней кубу згруповані у горизонтальну стрічку і зберігаються у одному файлі, наприклад, img.png. Це дозволяє для усіх текстур граней кубу мати єдиний хендл, що є доволі зручним.

Завантаження такого атласу текстур з одного з файлів drawable-ресурсів проєкту Android Studio можна запрограмувати наступним чином

```
textureHandle = myLoadTexture(context, R.drawable.img,  wrap);
```

Визначення координат вершин та запис їх у масив вершин можна зробити так само, як і для попереднього способу. Але варто враховувати, що значення x_T для текстурних координат будуть вже не 0 або 1, а відповідно розташуванню зображення кожної текстури у спільному растрі атласу текстур. Оскільки у атласі 6 текстур, то значення координати x_T кратні 1/6.

Рендеринг можна виконати наступним чином

```
GLES32.glBindVertexArray(VAO_id);
GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, textureHandle);
for (int i=0; i<6; i++) {
    GLES32.glDrawArrays(GL_TRIANGLE_FAN, 4*i, 4); //or GL_TRIANGLE_STRIP
}
GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, 0);
GLES32.glBindVertexArray(0);
```

Варто відзначити, що зменшення загальної кількості завантажуваних файлів текстур може сприяти покращенню роботи застосунку. Загалом, для підвищення швидкодії рендерингу OpenGL рекомендується зменшувати кількість перемикань-переходів станів у циклі рендерингу одного кадру. Такі операції, як зміна поточної текстури (блоку текстур) шляхом виклику glBindTexture( ), виклики glDrawArrays( ) варто намагаися робити якомога рідше.

Те, що усі грані кубу прив'язані лише до одної текстури, відкриває ще одну можливість для прискорення рендерингу. Якщо куб представити окремими трикутниками, то його можна малювати єдиним викликом glDrawArrays( )

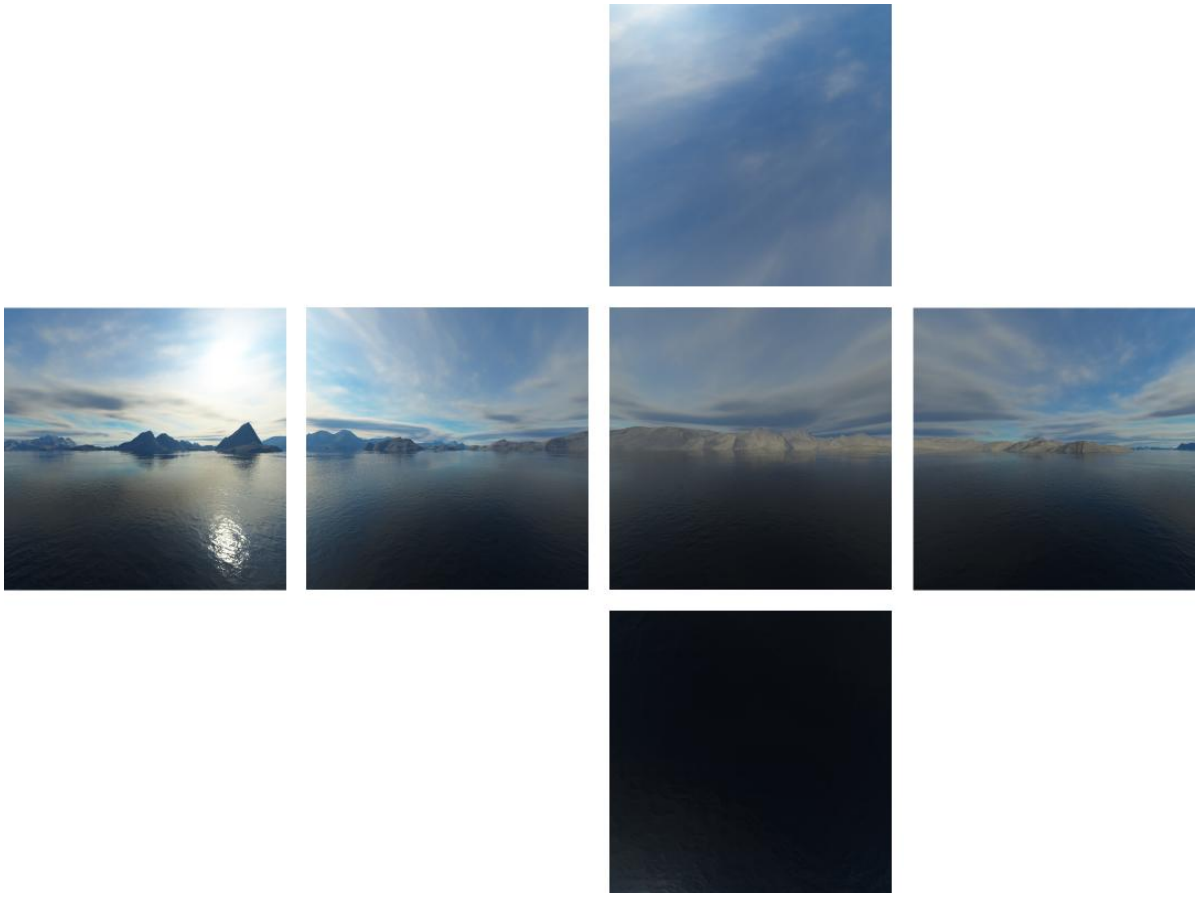
```
GLES32.glBindVertexArray(VAO_id);
GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, textureHandle);
GLES32.glDrawArrays(GL_TRIANGLES, 0, 36);
GLES32.glBindTexture(GLES32.GL_TEXTURE_2D, 0);
GLES32.glBindVertexArray(0);
```

Проте у останньому варіанті замість 6x4 = 24 вершин у масиві вершин потрібно зберігати вже 6x6 = 36 вершин.

Текстури зображення фону навколишнього середовища

Уявимо, що ми з певної точки на відкритій місцевості зробимо фотокамерою шість знімків у напрямках відповідно прямо, праворуч, ліворуч, назад, уверх та вниз. Такі фотознімки збережемо у 6 файлах зображень.

Нижче наведений приклад таких зображень, які звуться текстурами Skybox.



Джерело файлів текстур Skybox: *learnopengl.com/img/textures/skybox.zip*

з книги Joey de Vries. LearnOpenGL *https://learnopengl.com/book/book_pdf.pdf* на сайті https://learnopengl.com/

Потім такі зображення накладемо як текстури на відповідні грані кубу. Примітка. Кожна з 6 текстур Skybox має розміри 2048x2048 при 24-бітовому форматі растру – а це 12 МБ пам’яті. Для того, щоб полегшити справу, можна порекомендувати зменшити розміри текстур вдвічі – до 1024x1024, або ще суттєвіше – за допомогою відповідного графічного редактора.

Вигляд текстурованого кубу Skybox зовні при різних кутах огляду

Для створення повноцінної ілюзії кругового огляду віддаленого навколишнього середовища необхідно, щоб камера огляду завжди розташовувалася всередині точно у центрі кубу Skybox (або навпаки – якщо камера рухається, тоді куб також немов би рухається разом з камерою, щоб центр кубу був у точці огляду).

Вигляд кубу Skybox зсередини у одному з ракурсів повороту камери

Для того, щоб не було видно якихось артефактів на лініях дотику суміжних граней, при завантаженні текстури у пам'ять GPU потрібно вказати

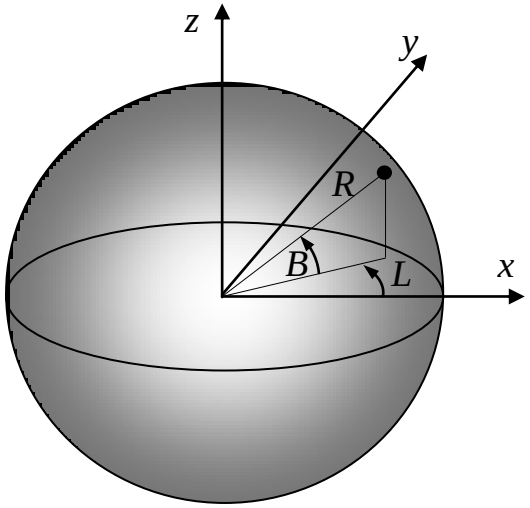
```
GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                       GLES32.GL_TEXTURE_WRAP_S, GLES32.GL_CLAMP_TO_EDGE);
GLES32.glTexParameteri(GLES32.GL_TEXTURE_2D,
                       GLES32.GL_TEXTURE_WRAP_T, GLES32.GL_CLAMP_TO_EDGE);
```

Основне призначення кубу Skybox – відображення навколишньої оболонки навколо деякої сцени, для зображення фону навколишнього середовища десь у безкінечності. Для відображення такого зображення існують різні спооби – наприклад, куб Skybox малюється першим, але з вимкненим тестом глибини (z-буфером). А потім усі решта об’єктів сцени накладаються вже поверх фону з увімкненим тестом глибини.

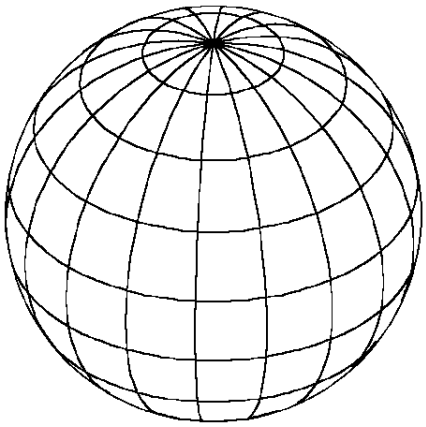
```
GLES32.glEnable(GLES32.GL_DEPTH_TEST);
. . .
GLES32.glDepthMask(false);                 //тимчасово вимикаємо тест глибини
GLES32.glDrawArrays(GL_TRIANGLES, 0, nvSkybox);  //малюємо куб Skybox першим
. . .
GLES32.glDepthMask(true);                       //вмикаємо тест глибини
GLES32.glDrawArrays(GL_TRIANGLES, s0, nv);     //малюємо решту об'єктів сцени
```

Доеладніше про технологію Skybox можна дізнатися у різноманітних інформаціних ресурсах

Текстурування кулі



Кутові координати точки на поверхні кулі



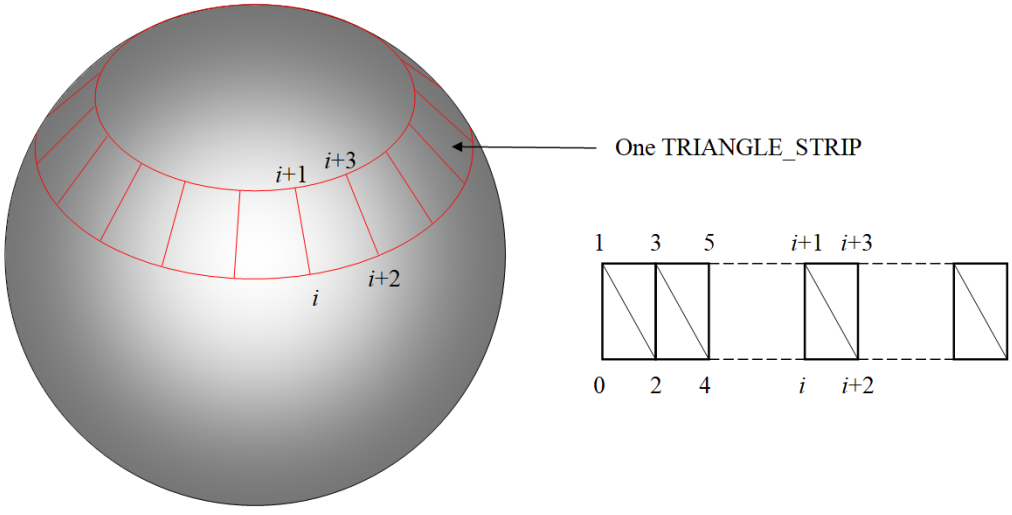
Сітка меридіанів та паралелей

Координати точок поверхні кулі визначаються як функції двох змінних (параметрів) – широти (B) та довготи (L)

$$x = R \cos B \cos L ,$$
$$y = R \cos B \sin L ,$$
$$z = R \sin B ,$$

де: B – широта (змінюється від -90º до +90º),
L – довгота (змінюється від –180º до +180º або від 0º до 360º).

Відображення поверхні кулі варто робити викликами функції glDrawArrays(TRIANGLE_STRIP, . . .) Вершини граней для кожної STRIP розташовуються у вузлах сітки, які належать сусіднім паралелям.

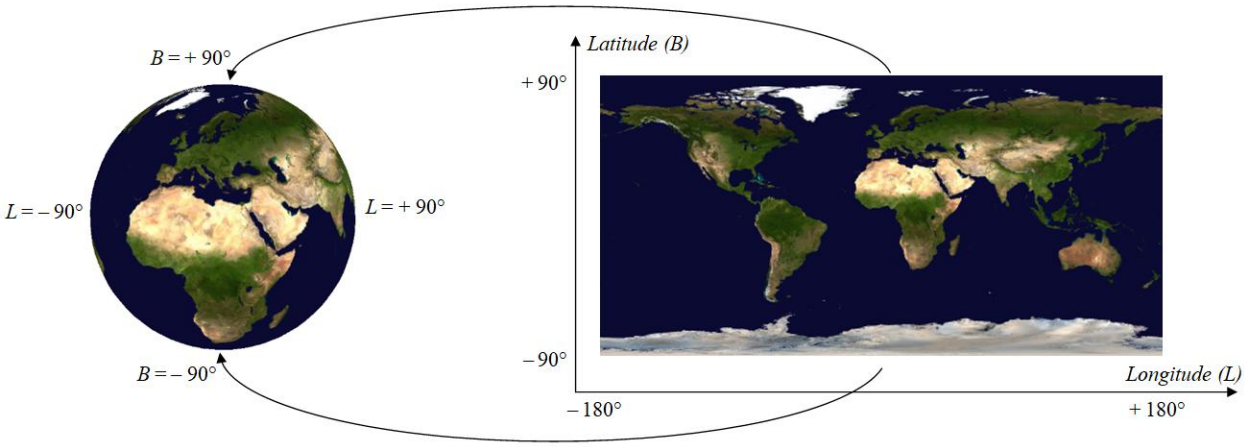


Цикл формування текстурованих стрічок полігональної сітки для кулі

```
startB = -0.5*pi
for (int bb=0; bb<seglat; bb++) {
    startNewStrip()
    for (int ll=0; ll<=seglong; ll++) {
        L = 2pi*ll/seglong            //longitude
        B = startB + pi*bb/seglat     //latitude
        addSphereVertexToStrip(B, L)           //i-th grid node

        B = startB + pi*(bb+1)/seglat)
        addSphereVertexToStrip (B, L)          //i+1-th grid node
    }
    endStrip()
}
```

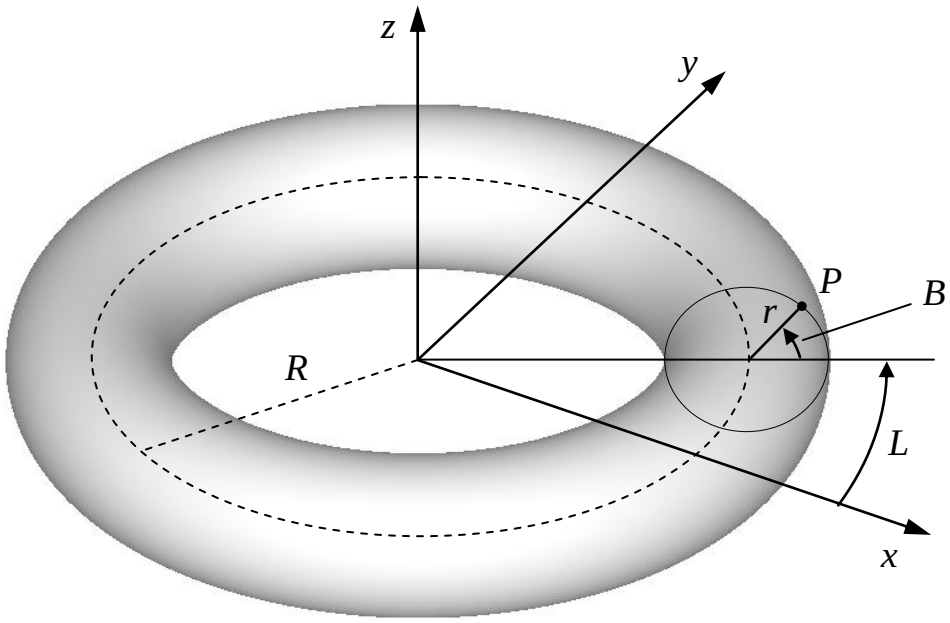
```
addSphereVertexToStrip(B, L) {
    x = cos(B)*cos(L)
    y = cos(B)*sin(L)
    z = sin(B)
    xt = L/2pi
    yt = 0.5 + B/pi
    yt = (1 - yt)    //such inversion is required for most raster texture formats
    storeVertex(arrayVertex,
        xcenter+x*radius,      //position
        ycenter + y*radius,
        zcenter + z*radius,
        x, y, z,       //normal vector
        xt, yt)        //texture coords
}
```

Яке зображення можна використати у якості текстури планети? Виберемо вертикальну циліндричну рівнопроміжну проекцію кулі, для якої найпростіше перераховуються координати – із кутових координат планети (широти, довготи) у піксельні координати (x_T, y_T) текстури-розгортки.



Текстурування поверхні тору

По аналогії з кулею використаємо кутові координати широти (*B*) та довготи (*L*) для опису розташування довільної точки на поверхні тора.

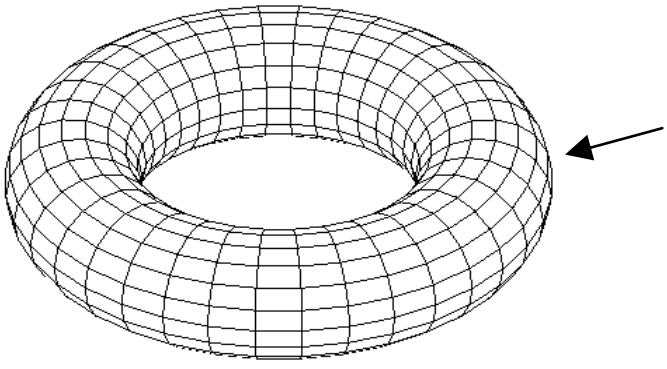


Координати точки поверхні тору

$$x = (R + r\cos B)\cos L\ ,$$
$$y = (R + r\cos B)\sin L\ ,$$
$$z = r\sin B\ ,$$

де: R – великий (зовнішній) радіус,
r – внутрішній радіус, причому $r < R$,
B – широта у діапазоні (від 0º до 360º),
L – довгота у діапазоні (від 0º до 360º)

Поверхню тору також можна представити замкненою полігональною сіткою

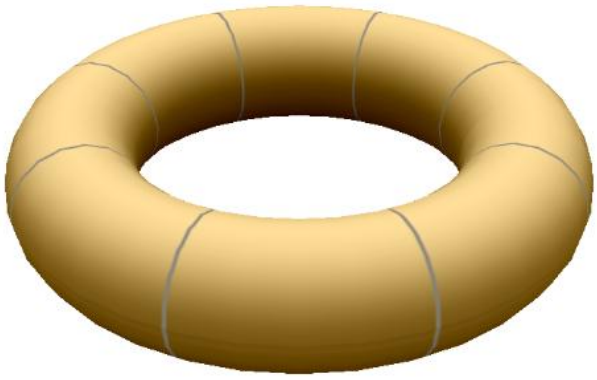


Полігональна сітка поверхні тору

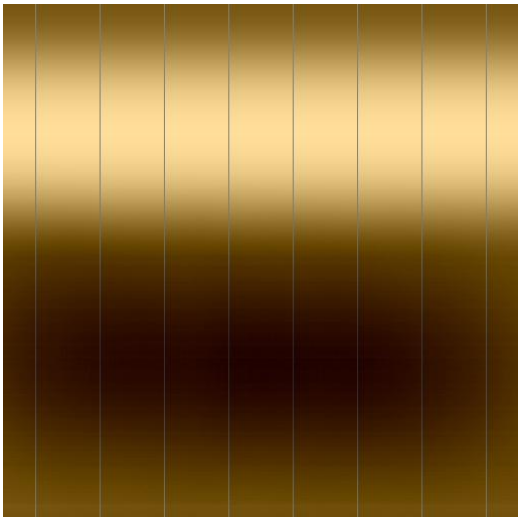
Цикл формування текстурованих стрічок полігональної сітки для тору

```
for (int bb=0; bb<seglat; bb++) {
    startNewStrip()
    for (int ll=0; ll<=seglong; ll++) {
        L = 2pi*ll/seglong              //longitude
        B = 2pi*bb/seglat              //latitude
        addTorusVertexToStrip(B, L)           //i-th grid node

        B = 2pi*(bb+1)/seglat)
        addTorusVertexToStrip (B, L)          //i+1-th grid node
    }
    endStrip()
}
```

```
addTorusVertexToStrip (B, L) {
    x = R*cos(L);
    y = R*sin(L);
    xn = cos(B)*cos(L)
    yn = cos(B)*sin(L);
    zn = sin(B);
    xt = L/2pi
    yt = B/2pi        //or 1 - B/2pi
    storeVertex(arrayVertex,
        xcenter+xn*radius+x,    //x position
        ycenter+yn*radius+y,    //y position
        zcenter+zn*radius,      //z position
        xn, yn, zn,           //normal vector
        xt, yt)               //texture coords
}
```



Torus



Texture

Текстура може використовуватися для імітації відбиття світла, зокрема, на цьому рисунку текстура створює ілюзію дифузного відбиття світла.

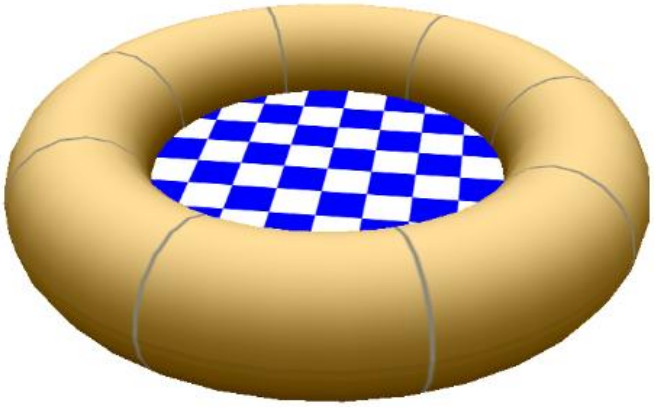
Варіанти завдань та основні вимоги

1. Застосунок **Lab5_GLES** для вибору режиму роботи повинен мати меню з трьома пунктами:

- Torus
- Earth
- Seaview

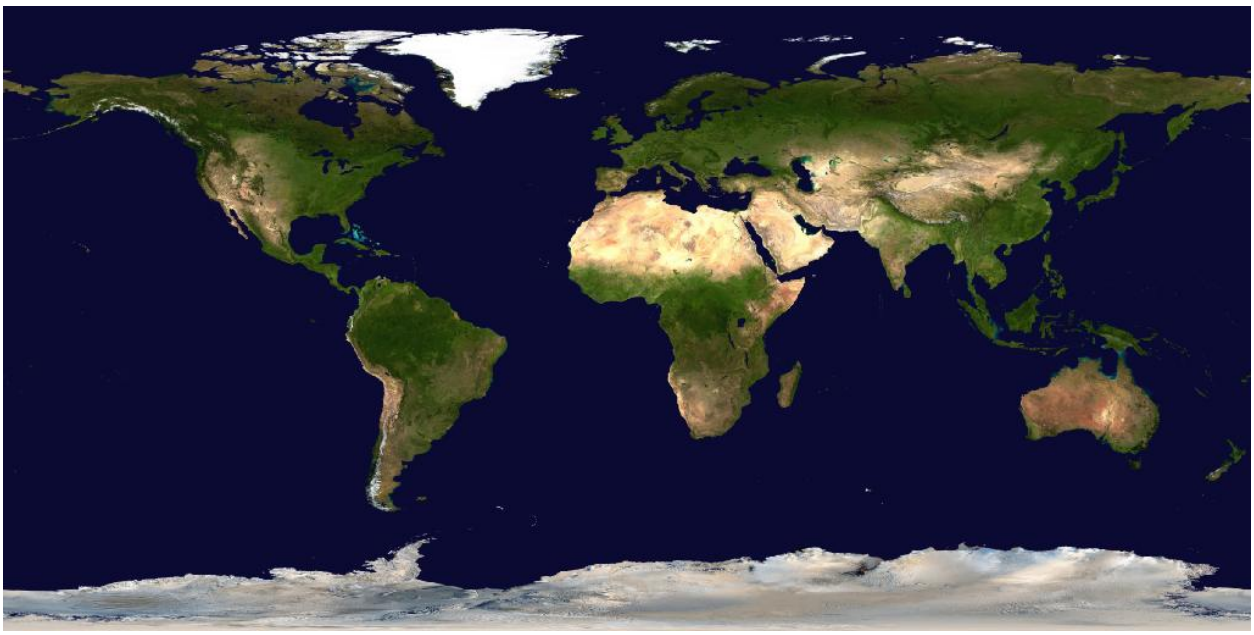
2. Меню має забезпечувати вибір потрібного режиму роботи

3. У режимі Torus потрібно запрограмувати рендеринг текстурованого тора та чотирикутника шахової дошки.



4. У режимі Earth потрібно запрограмувати текстуроване зображення планети.

У якості текстури використати розгортку з космічних знімків, яка надана нижче



Розгортка-текстура планети

5. У режимі Seaview запрограмувати куб Skybox. Центр кубу – у центрі системи координат. Усередині кубу Skybox розташувати тор із шаховою дошкою (п. 3).

6. В усіх режимах потрібно забезпечити керування ракурсом показу. Потрібно зробити керування поворотами по горизонталі та вертикалі а також для наближення-віддалення (збільшення-зменшення).

Зміст звіту

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

Контрольні запитання

1. Що таке текстура?
2. Як завантажити текстуру?
3. Що таке атлас тестур і які переваги він може забезпечити?
4. Як зробити циклічне повторення зображення текстури?
5. Що таке текстурні координати?
6. Як можна імітувати зображення фону навколишнього середовища?
7. Що спільного у відображенні тору та кулі?