

Лабораторна робота №1. Знайомство з базовими засобами комп'ютерної графіки деяких операційних платформ

Мета: отримати перші навички створення програм то набути знання щодо базових засобів відображення графіки для різних операційних платформ.

У цій лабораторній роботі буде знайомство з такими графічними підсистемами операційних систем:

- GDI Windows;
- Android Graphics Canvas;
- Android OpenGL ES

Зміст

Завдання

Засоби для виконання лабораторної роботи

Теоретичні положення та методичні рекомендації

Частина 1. Графіка GDI Windows

Частина 2. Графіка Android Graphics Canvas

Частина 3. Графіка Android OpenGL ES

Варіанти завдань та основні вимоги

Зміст звіту

Контрольні запитання

Завдання

Кожному студенту потрібно зробити 3 проєкта, вказані нижче.

1. Створити у середовищі MS Visual Studio C++ проєкт з ім'ям **Lab1**.
 - Написати вихідний текст програми згідно варіанту завдання.
 - перевірити роботу програми. Налаштувати програму.
2. Створити у середовищі Android Studio проєкт з ім'ям **Lab1_Canvas**.
 - написати вихідний текст програми згідно варіанту завдання. Використати мову Java або Kotlin – на вибір
 - Налаштувати програму. Перевірити роботу програми на емуляторі та на фізичному пристрої Android.
3. Створити у середовищі Android Studio проєкт з ім'ям **Lab1_GLES**.
 - написати вихідний текст програми згідно варіанту завдання. Використати мову Java або Kotlin – на вибір
 - Налаштувати програму. Перевірити роботу програми на емуляторі та на фізичному пристрої Android.

Засоби для виконання лабораторної роботи

1. Microsoft Visual Studio C++ – середовище розробника на платформі ОС Windows. Рекомендується інстальювати версію Visual Studio Community не старіше 2019 року. При інсталяції вказати наявність підтримки C++ та Win SDK. Посилання для завантаження:

<https://visualstudio.microsoft.com/>

2. Android Studio – середовище розробки Android-застосунків. Функціонує на платформі ОС Windows або Linux або MacOS. Рекомендується інстальювати версію Android Studio не старіше 2024 року. Посилання для завантаження: <https://developer.android.com/studio>

Теоретичні положення та методичні рекомендації

Робота складається з трьох частин з метою початкового знайомства з програмуванням графіки на трьох платформах.

Частина 1. Графіка GDI Windows

Ця частина присвячена знайомству з базовими можливостями графіки, які закладені у Application Programming Interface (API) для Windows. Одною зі складових цього API є Graphics Device Interface (GDI). Цей API розроблений для використання програмістами C/C++. Потрібне знання архітектури застосунків для Windows, керованих повідомленнями.

1.1. Створення проєкту Visual Studio C++

Потрібно створити проєкт C++ типу Windows Desktop Application

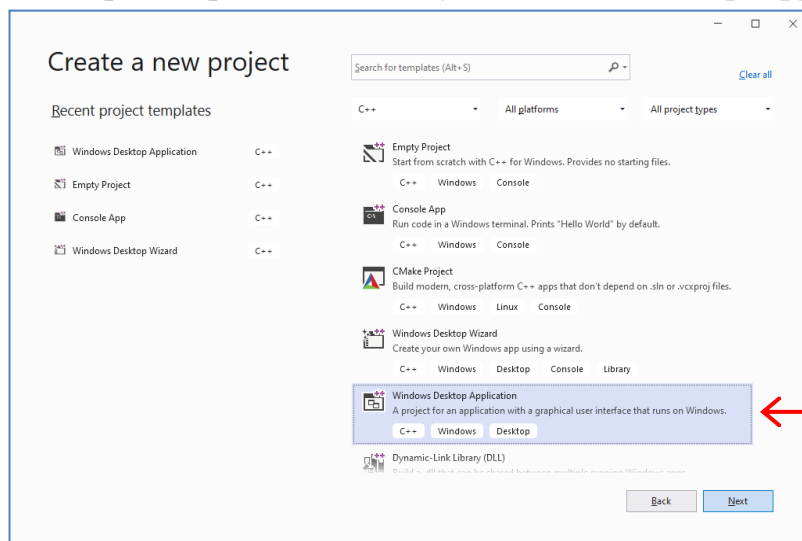


Рис. 1. Вибір типу проєкту Visual Studio

Далі потрібно вписати ім'я проекту (студентам – згідно завдання)

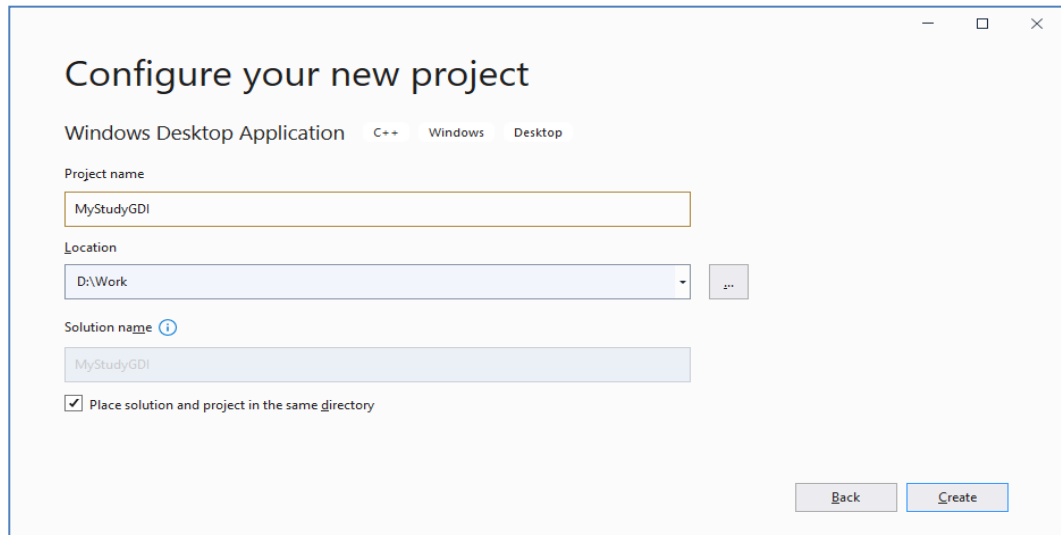


Рис. 2. Визначення імені та розташування проекту

Потрібно натиснути кнопку [Create]

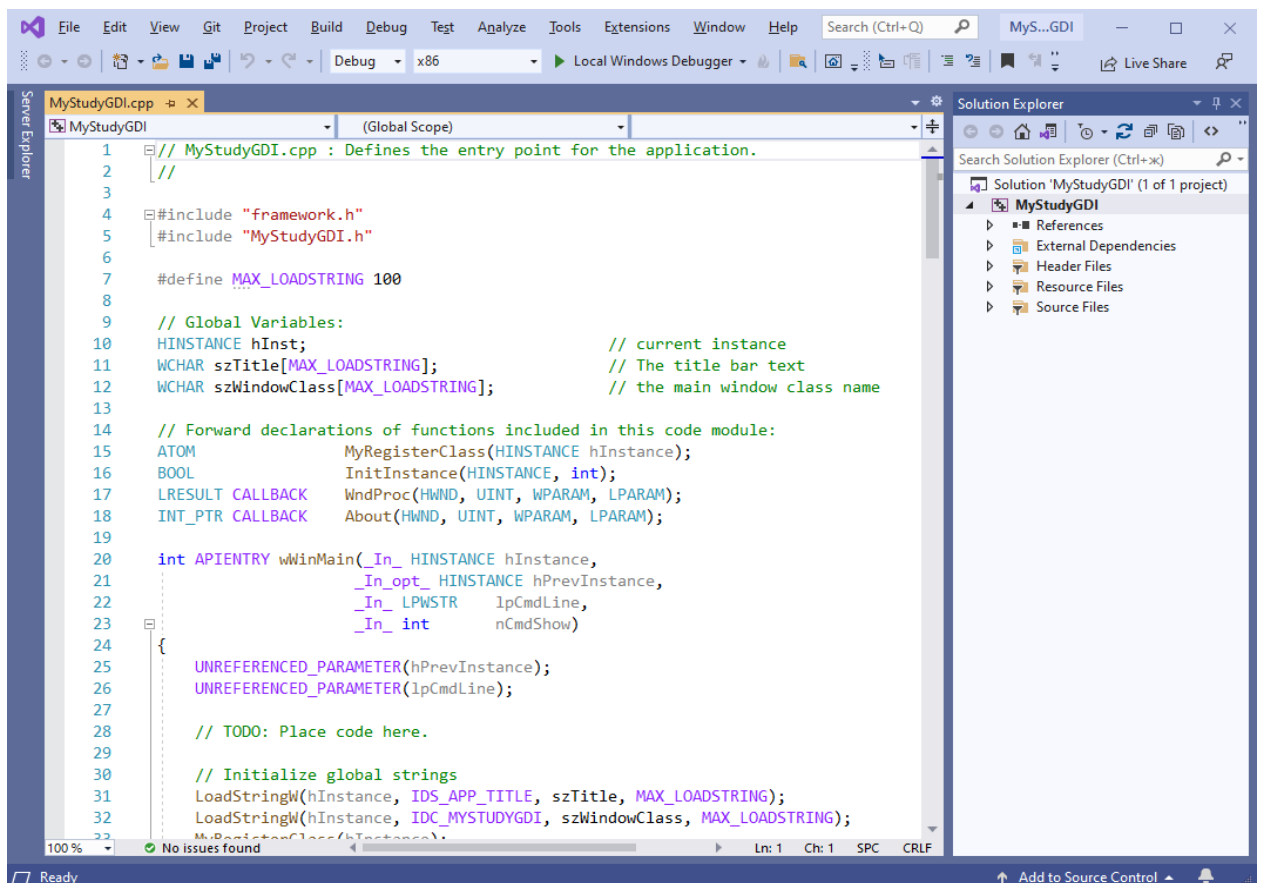


Рис. 3. Проект Visual Studio C++ одразу після створення

У середовищі Visual Studio по замовчуванню спочатку показується головний файл – у цьому випадку MyStudyGDI.cpp та вікно списку файлів проекту (Solution Explorer).

Перевіримо новостворений проєкт. Спробуємо викликати на виконання нашу програму. Для цього виберемо меню “Debug -> Start Debugging”

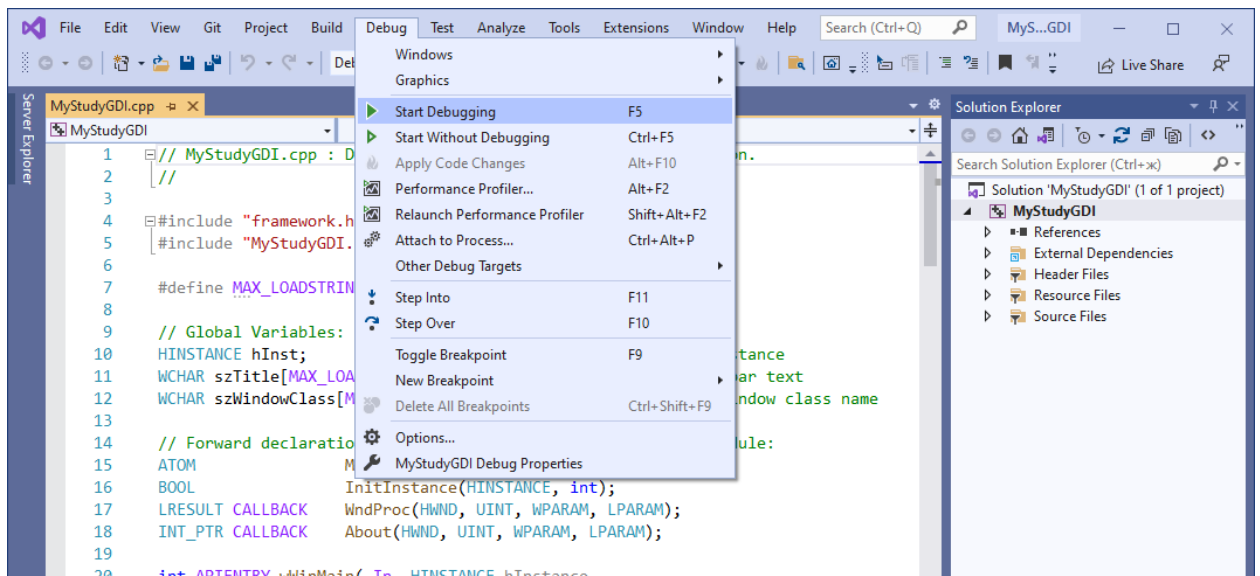


Рис. 4. Виклик програми на виконання

Виконується компіляція, лінування і у разі успіху цільова програма викликається на виконання і ми повинні побачити її вікно

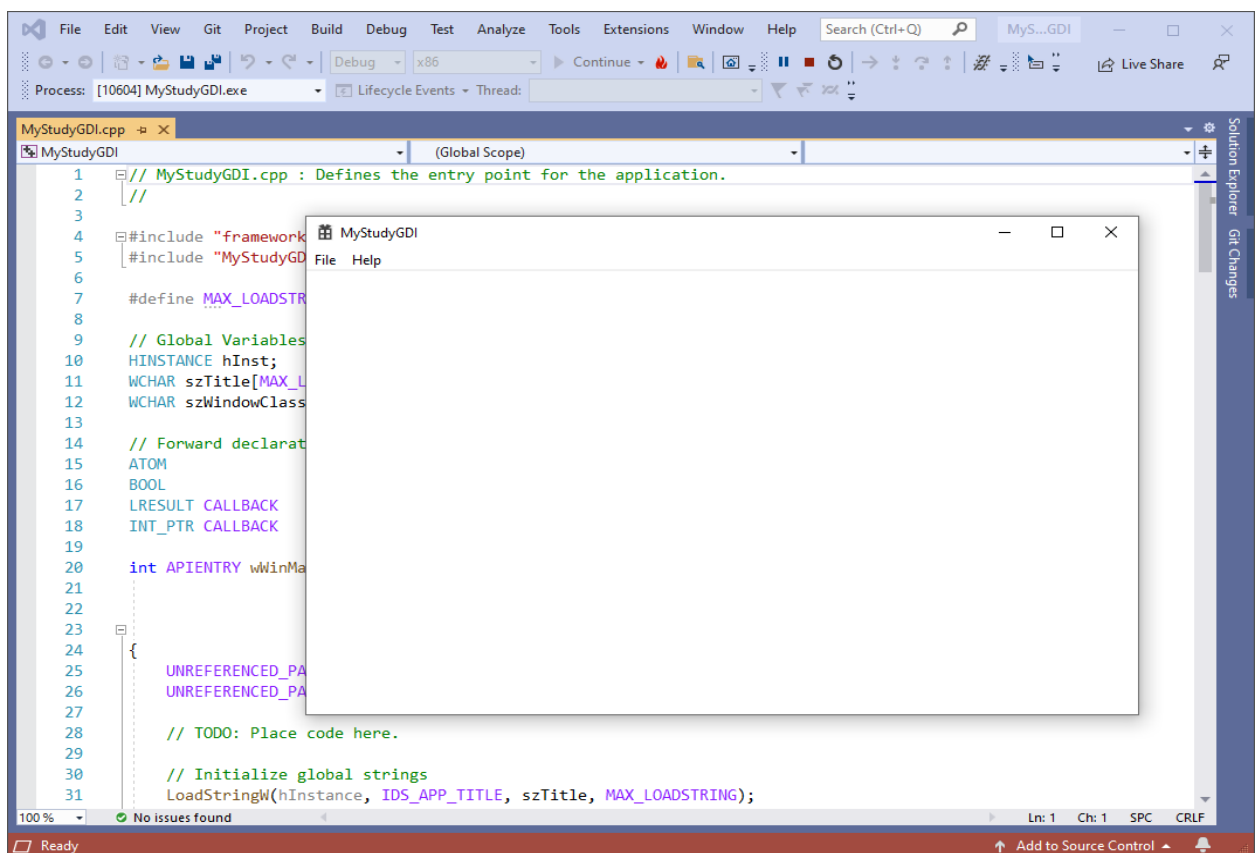


Рис. 5. Вікно працюючої програми на тлі Visual Studio

1.2. Шаблон для програмування графіки у вікні застосунку Windows

Розглянемо фрагмент програмного коду – текст визначення функції головного вікна. Це callback функція WndProc – обробник повідомлень Windows

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
                case IDM_ABOUT:
                    DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
            }
        }
        break;
        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code that uses hdc here...
            EndPaint(hWnd, &ps);
        }
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Мабуть, можна було б вписати програмний код графіки безпосередньо між дужками для case WM_PAINT – як рекомендовано у автоматично згенерованому рядку //TODO: Add any drawing code . . , проте програмний код тоді буде громіздким та не зручним для роботи з проєктом.

Зробимо програмний код структурованим. Оформимо обробник повідомлення WM_PAINT у вигляді нашої власної функції onPaint(), яка у свою чергу буде викликати функцію myDraw().

```

//Forward declarations of functions
. . .
void onPaint(HWND hWnd);
void myDraw(HWND hWnd, HDC hdc);
. . .

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                          WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        . . .
        case WM_PAINT:
            onPaint(hWnd);
            break;
        . . .
    }
    return 0;
}

. . .
void onPaint(HWND hWnd)
{
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hWnd, &ps);
    myPaint(hWnd, hdc);
    EndPaint(hWnd, &ps);
}

void myDraw(HWND hWnd, HDC hdc)
{
    // TODO: Add any drawing code that uses hdc here...
}

```

Після того, як ми зробили невеличкий рефакторинг коду, сформовано шаблон для подальшого програмування. Далі потрібно буде записувати програмний код у тілі функції myDraw().

1.3. Огляд базових графічні примітивів GDI Windows

Для вивчення графічних примітивів графіки GDI Windows рекомендується звернутися до першоджерела: Microsoft Learn – Windows App Development – Windows GDI за посиланням <https://learn.microsoft.com/en-us/windows/win32/gdi/windows-gdi>

Поняття контексту графічного пристрою

Спочатку треба вказати, що для усіх функцій виводу графічних примітивів першим параметром вказується хендл (ідентифікатор) контексту графічного пристрою (HDC – Handle of Device Context). Значення hdc можна отримати для вікна так, як вже розглянуто вище

```
HDC hdc = BeginPaint(hWnd, &ps);  
//. . . щось малюємо  
EndPaint(hWnd, &ps);
```

Необхідно зазначити, що парні функції `BeginPaint` – `EndPaint` можна використовувати лише при обробці повідомлення `WM_PAINT`.

Для відображення у вікні також (але не для `WM_PAINT`) можна використати комплект функцій `GetDC` та `ReleaseDC` – так як записано нижче

```
HDC hdc = GetDC(hWnd);  
//. . . щось малюємо  
ReleaseDC(hWnd, hdc);
```

Малювання окремих пікселів

Функція **`SetPixel`** малює один піксел растру.

```
SetPixel(hdc, x, y, clr);
```

Вона має такі параметри:

- **`hdc`** – хендл контексту графічного пристрою;
- **`x, y`** - координати точки;
- **`clr`** – колір пікселя. Параметр `clr` має тип 4-байтового `COLORREF`, причому три молодших байта є компонентами Blue, Green, Red (у діапазоні від 0 до 255 кожна). Колір формату `COLORREF` зручно вказувати макросом `RGB(r, g, b)`.

Функція **`SetPixel`** повертає `COLORREF` значення кольору пікселя.

Також є функція **`GetPixel(hdc, x, y)`** для отримання кольору будь-якого пікселя растру, який асоціюється з даним `hdc`.

Малювання ліній

У складі GDI Windows є декілька графічних примітивів, які призначені для малювання ліній:

`AngleArc` — дуга еліпсу із вказуванням кутів повороту;

`Arc, ArcTo` — дуга еліпсу;

`LineDDA` – вивід відрізка прямої лінії з використанням визначеної користувачем функції виводу точок ліній;

LineTo – малювання відрізка прямої від поточної позиції до визначеної точки;

MoveToEx – визначення поточної позиції графічного виводу;

PolyBezier, PolyBezierTo – один або декілька зв'язаних між собою кубічних сплайнів Безьє;

PolyDraw – декілька зв'язаних відрізків прямих і сплайнів Безьє;

Polyline, PolylineTo – ламана лінія із багатьох зв'язаних між собою відрізків прямих (полілінія);

PolyPolyline – декілька поліліній як єдиний об'єкт.

Джерело: Windows App Development. Line and Curve Functions
<https://learn.microsoft.com/en-us/windows/win32/gdi/line-and-curve-functions>

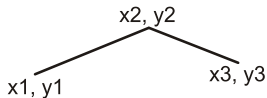
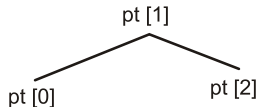
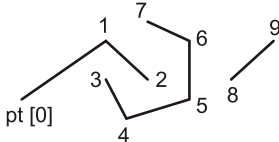
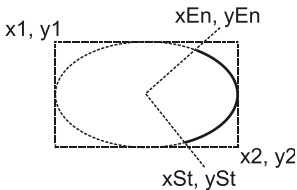
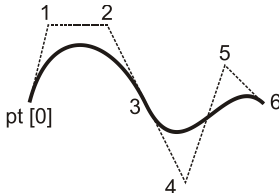
Зразки виводу графіки	Програмний код
	<pre>MoveToEx(hdc, x1, y1, NULL); LineTo(hdc, x2, y2); LineTo(hdc, x3, y3);</pre>
	<pre>POINT pt[3]; Polyline(hdc, pt, 3);</pre>
	<pre>POINT pt[10]; DWORD npt[3]={3, 5, 2}; PolyPolyline(hdc, pt, npt, 3);</pre>
	<pre>Arc(hdc, x1, y1, x2, y2, xSt, ySt, xEn, yEn);</pre>
	<pre>POINT pt[7]; PolyBezier(hdc, pt, 7);</pre>

Рис. 6. Різновиди примітивів для малювання ліній

По замовчуванню усі лінії малюються чорні, суцільні. Щоб змінити стиль, потрібно створити перо, намалювати одну або декілька ліній, а коли перо стає непотрібним, його обов'язково треба знищити. Нижче наведено

шаблон програмного коду для визначення кольору та товщини суцільних ліній

```
HPEN hPenOld, hPen;  
  
hPen = CreatePen(PS_SOLID, 3, Колір); //суцільна лінія товщиною 3  
hPenOld = (HPEN)SelectObject(hdc, hPen);  
  
Малюється лінія (або декілька ліній однакового стилю)  
  
SelectObject(hdc, hPenOld);  
DeleteObject(hPen);
```

Окрім суцільних ліній можна малювати різноманітні пунктирні лінії. Але при малюванні пунктирної лінії можна використовувати лише одиничне значення товщини

```
HPEN hPenOld, hPen;  
  
hPen = CreatePen(PS_DOT, 1, Колір); //пунктирна лінія  
hPenOld = (HPEN)SelectObject(hdc, hPen);  
  
Малюється лінія (або декілька ліній однакового стилю)  
  
SelectObject(hdc, hPenOld);  
DeleteObject(hPen);
```

Товсті пунктирні лінії доводиться малювати трохи складніше – програмувати декілька тонких ліній зі зсувом, або використати функцію LineDDA.

Малювання фігур із заповненням

У складі GDI Windows є декілька графічних примітивів, які призначені для малювання фігур із заповненням:

Ellipse – еліпс;

Chord – хорда еліпсу;

Pie – сектор еліпсу;

Polygon – полігон;

PolyPolygon – декілька полігонів та (або) полигони з дірками;

Rectangle – прямокутник;

RoundRect – прямокутник зі скругленими кутами.

Джерело: Windows App Development. Filled Shapes

<https://learn.microsoft.com/en-us/windows/win32/gdi/filled-shapes>

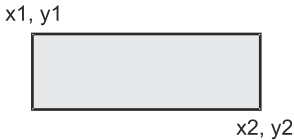
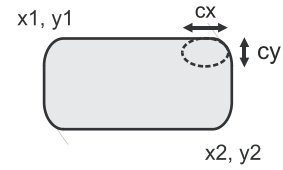
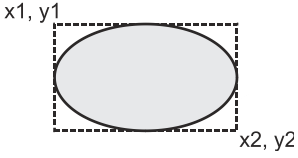
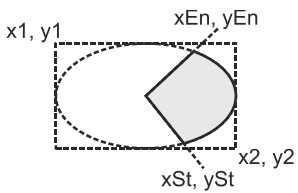
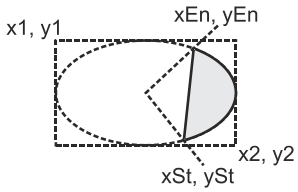
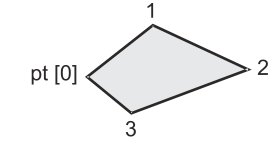
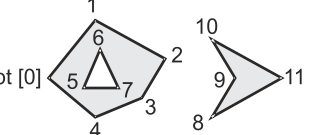
Зразки виводу графіки	Програмний код
	<pre>Rectangle(hdc, x1, y1, x2, y2);</pre>
	<pre>RoundRect(hdc, x1, y1, x2, y2, cx, cy);</pre>
	<pre>Ellipse(hdc, x1, y1, x2, y2);</pre>
	<pre>Pie(hdc, x1, y1, x2, y2, xSt, ySt, xEn, yEn);</pre>
	<pre>Chord(hdc, x1, y1, x2, y2, xSt, ySt, xEn, yEn);</pre>
	<pre>POINT pt[4]; Polygon(hdc, pt, 4);</pre>
	<pre>POINT pt[12] int npt[3] = {5, 3, 4}; PolyPolygon(hdc, pt, npt, 3);</pre>

Рис. 7. Графічні примітиви для малювання фігур із заповненням

Визначення кольору заповнення фігур можна запрограмувати у такий спосіб:

```
HBRUSH hBrush, hBrushOld;

hBrush = (HBRUSH)CreateSolidBrush(colorfill); //новий пензель
hBrushOld = (HBRUSH)SelectObject(hdc, hBrush); //пензель -> у hdc

Ellipse(hdc, x1, y1, x2, y2); //або інша фігура

SelectObject(hdc, hBrushOld); //відновлюється пензель-попередник
DeleteObject(hBrush); //створений пензель знищується
```

Спочатку потрібно створити пензель та помістити його у контекст пристрою (hdc). Далі виклик функцій малювання потрібних одної або декількох фігур. Після того, як такий пензель вже більше не потрібен, або треба створити новий, то спочатку вибирається попередній пензель, а потім створений знищується (інакше будуть витіки пам'яті).

Аргументом функції **CreateSolidBrush** є ціле число, яке визначає колір заповнення. У якості аргументу можна рекомендувати макрос RGB, наприклад, для жовтого кольору заповнення:

```
hBrush = (HBRUSH)CreateSolidBrush(RGB(255,255,0));
```








Колір заповнення		RGB
	білий	RGB (255, 255, 255) (по замовчуванню)
	жовтий	RGB (255, 255, 0)
	світло-зелений	RGB (0, 255, 0)
	блакитний	RGB (0, 255, 255)
	рожевий	RGB (255, 0, 255)
	помаранчевий	RGB (255, 128, 0)
	сірий	RGB (192, 192, 192) (для градацій сірого R = G = B)

Рис. 8. Значення компонент RGB для деяких кольорів

Для того, щоб намалювати фігуру із штриховим, візерунковим заповненням можна скористатися спеціалізованою функцією CreateHatchBrush() або багатоцільовою функцією CreateBrushIndirect().

Джерело: Windows App Development. Brush Functions (Windows GDI)

<https://learn.microsoft.com/en-us/windows/win32/gdi/brush-functions>

1.4. Система координат вікна застосунку для Windows

Головне вікно застосунку для Windows має прямокутну форму. На поверхні вікна розташовуються наступні елементи:

- верхня стрічка заголовку із системним меню (згортання вікна, максимізація вікна, закриття);
- прямокутник клієнтської частини вікна для відображення графіки;
- бордюр.

Вказані вище елементи є стандартними, обов'язковими і присутні, як правило, у вікні будь-якого застосунку Windows. Окрім таких елементів, у головному вікні можна побачити такі елементи, які є необов'язковими і визначаються програмістом вже для конкретного застосунку:

- стрічка меню;
- повзуни скролінгу;
- стрічка або панель кнопок засобів (toolbar), стрічка стану (status bar);
- інші дочірні вікна

Відображення графіки у контексті вікна (HDC) виконується в клієнтській частині вікна. Вісь координат X по горизонталі зліва на право, вісь координат Y по вертикалі з гори до низу. Центр координат (0,0) у лівому верхньому куті клієнтської частини вікна. Одиниці вимірів координат – піксели екрану.

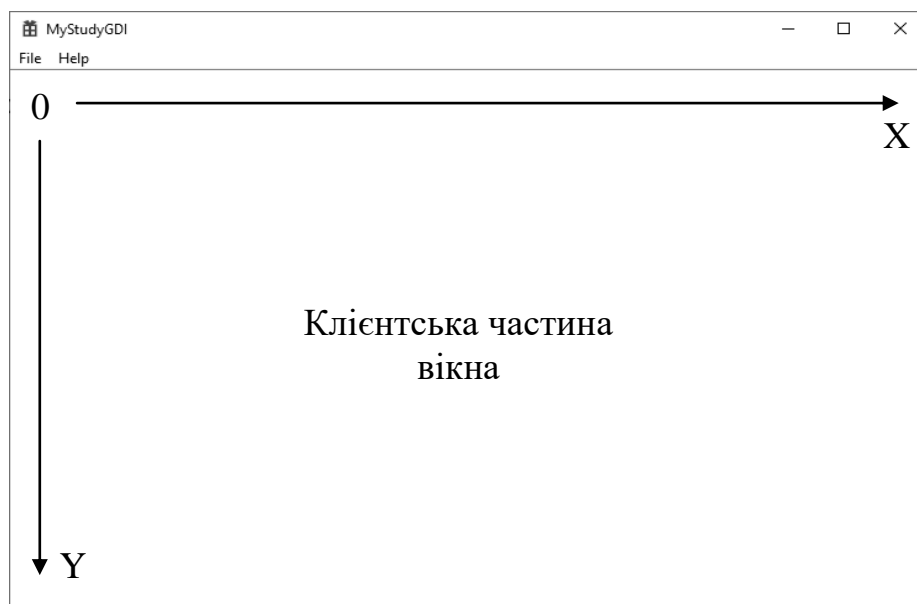


Рис. 9. Система координат вікна застосунку для Windows GDI

Примітка. Вказана вище система координат встановлюється Windows по замовчуванню. У Windows API передбачені можливості змінити систему координат програмним шляхом впродовж роботи застосунку.

Частина 2. Графіка Android Graphics Canvas

Для програмування застосунків для Android будемо використовувати мову Java, хоча студенти можуть, за власним вибором, використати Kotlin.

2.1. Створення проєкту Android Studio

Для виконання завдань лабораторної роботи потрібно спочатку створити проєкт згідно початковому шаблону порожнього проєкту “Empty Views Activity”

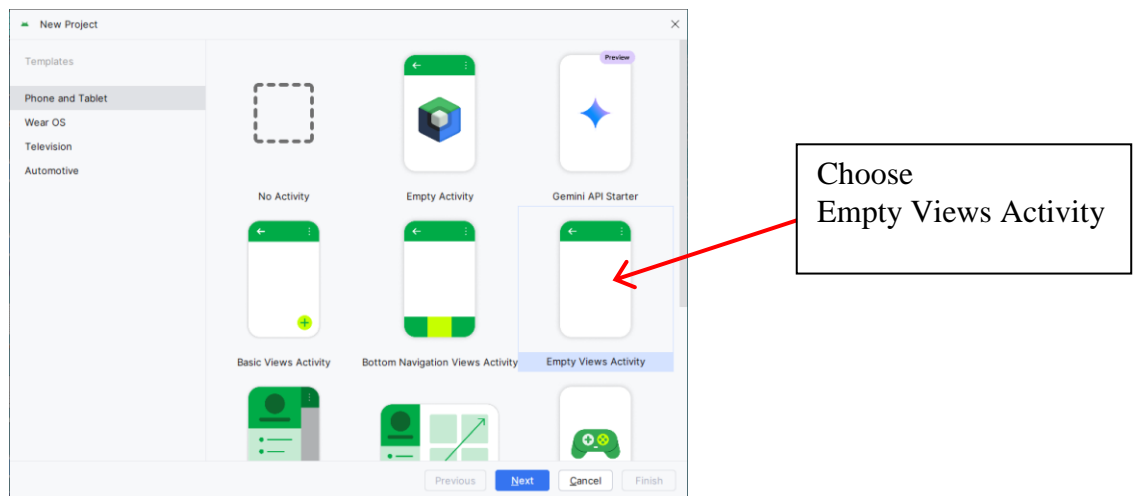


Рис. 10. Створення нового проєкту в Android Studio

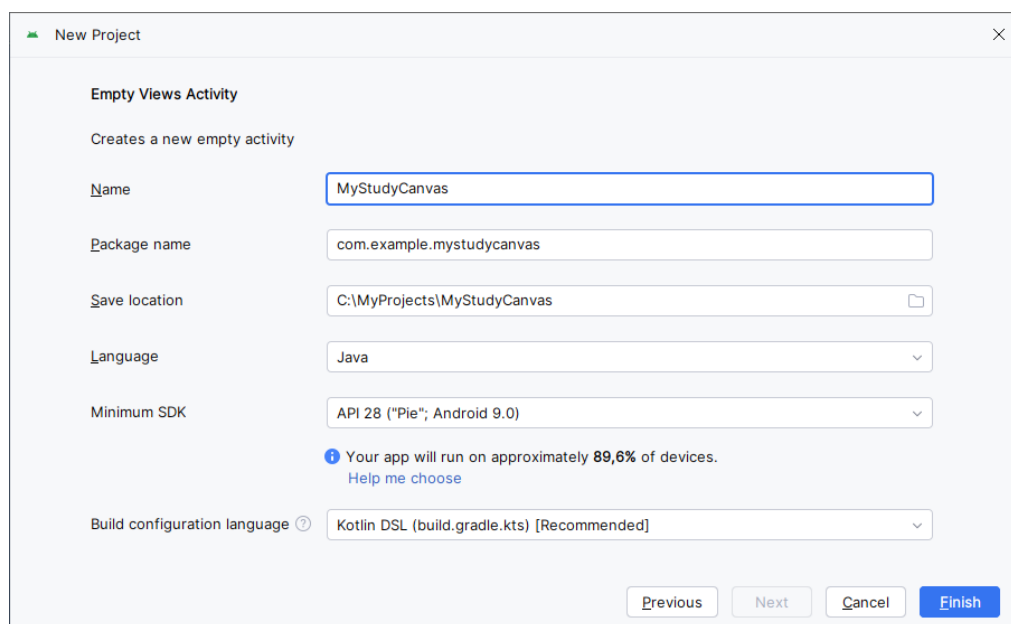


Рис. 11. Визначення імені проєкту, мови програмування Java та рівня API 28

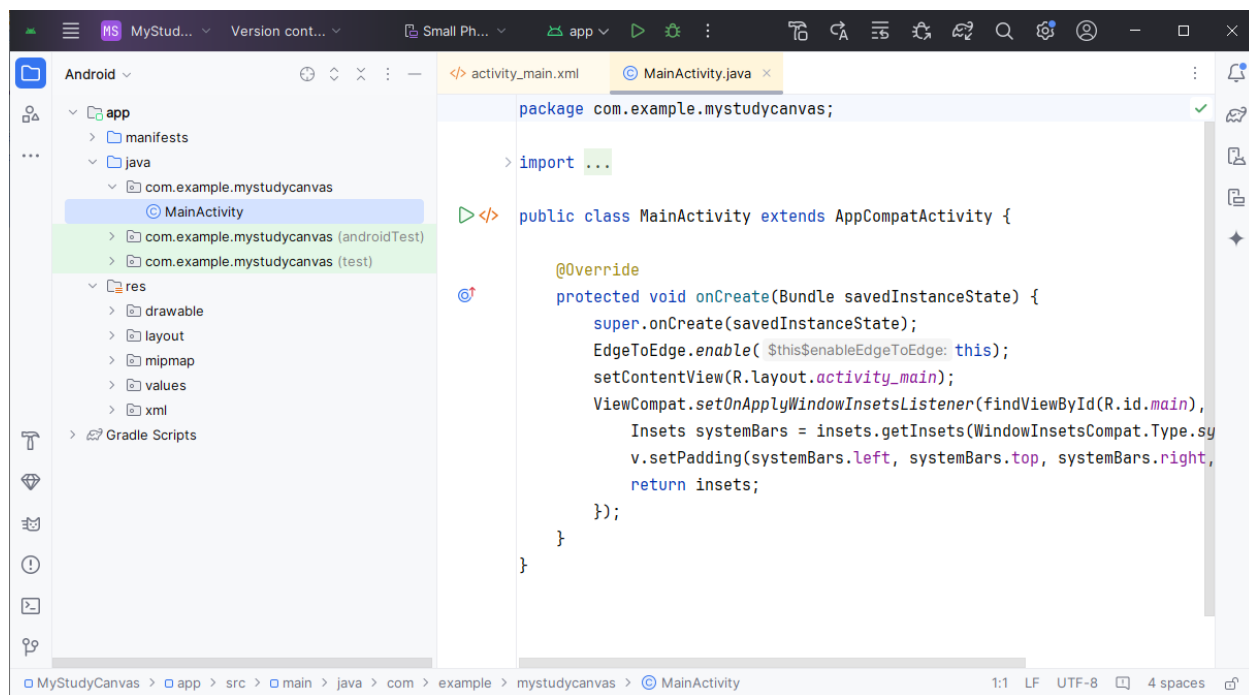


Рис. 12. Вигляд новоствореного проєкта в середовищі Android Studio

Перевірку функціонування застосунку та налагодження програмного коду у середовищі Android Studio можна виконати з використанням віртуального пристрою. Для цього потрібно створити-визначити віртуальний пристрій. Для цього можна вибрати меню Tools–Device Manager, і далі вибрати опцію (+) Create Virtual Device. З’явиться наступне вікно

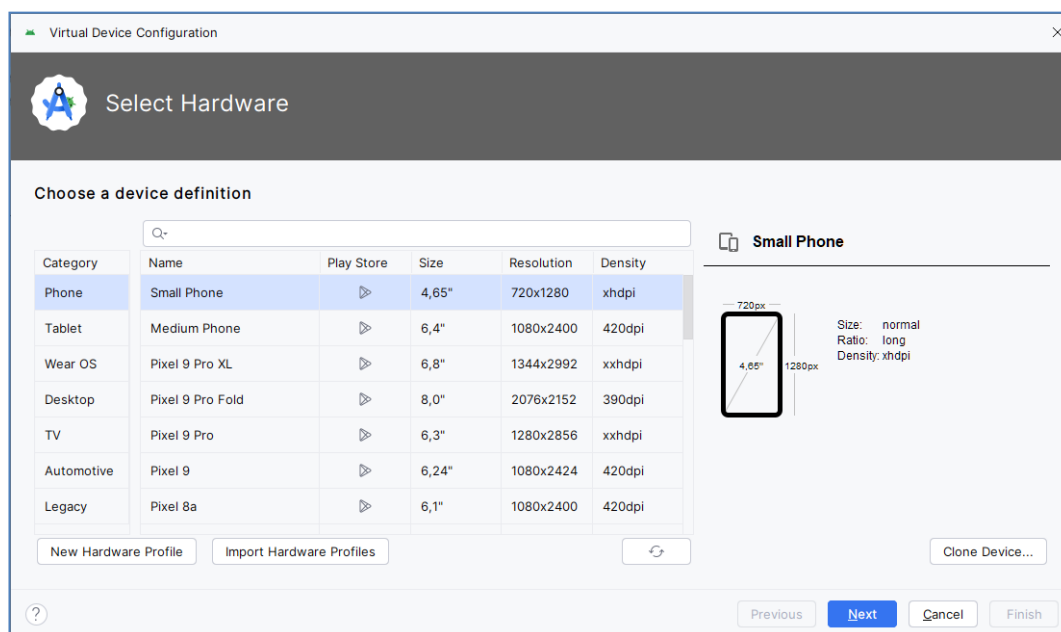


Рис. 13. Список віртуальних пристроїв

Далі потрібно з'ясувати параметри пристрою, зокрема, рівень API – необхідно, щоб рівень API пристрою був не меншим рівня API проекту. У нашому випадку цільовим рівнем є API 28.

Після визначення віртуального пристрою можна перевірити роботу застосунку. Для цього меню Run→Run app.

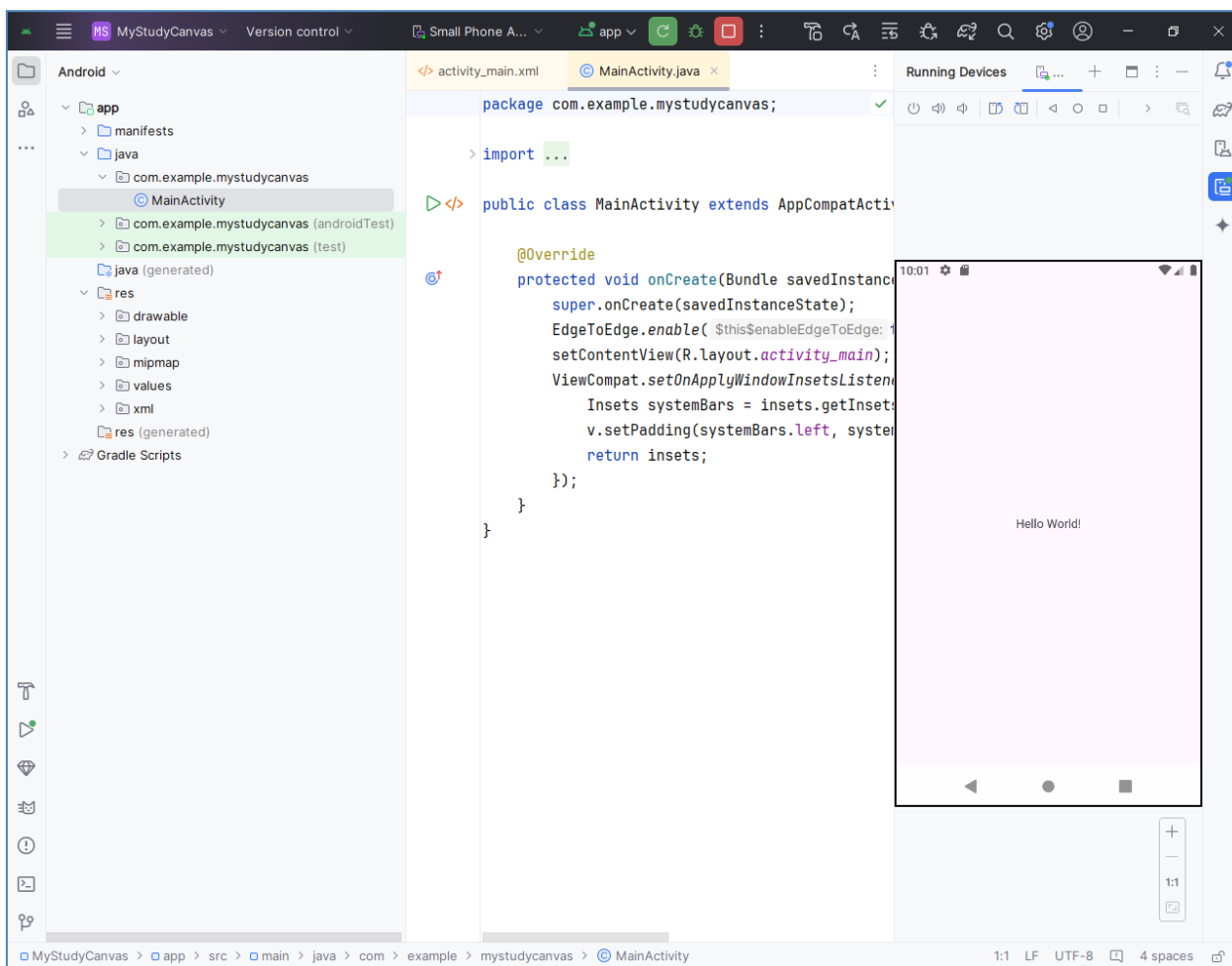


Рис. 14. Перевірка роботи застосунку в емуляторі віртуального пристрою

Для того, щоб перевірити роботу застосунку на фізичному пристрої, наприклад, смартфоні, потрібно його під'єднати кабелем USB до комп'ютера, на якому функціонує Android Studio. Після цього Android Studio повинен розпізнати цей пристрій – у верхній стрічці Android Studio має з'явитися ім'я пристрою. Виклик на виконання застосунку на фізичному пристрої робиться так само, як і для віртуального пристрою – меню Run→Run app (або натискуванням зеленого трикутника у верхній стрічці вікна Android Studio).

2.2. Основні відомості про Android Graphics Canvas

Вичерпні відомості про Android Graphics можна знайти у першоджерелі для розробників-програмістів – у одному з розділів опису Android API, а саме <https://developer.android.com/reference/android/graphics/package-summary>. Як там вказано, “android.graphics Provides low level graphics tools such as canvases, color filters, points, and rectangles that let you handle drawing to the screen directly”. Тобто, це набір достатньо низькорівневих засобів для швидкого відображення графіки безпосередньо на екрані.

Одним з класів системи android.graphics є клас Canvas. Повний перелік членів цього класу з коментарями щодо їхнього використання можна знайти у розділі <https://developer.android.com/reference/android/graphics/Canvas>.

Клас Canvas містить багато десятків різноманітних методів, зокрема, методи малювання, які позначені переважно як drawXXX, наприклад, drawARGB, drawBitmap, drawOval та інші. Велику множину з таких методів складають графічні примітиви для відображення певних геометричних форм, такі як drawArc, drawOval, drawRect, drawText тощо.

2.3. Базові графічні примітиви Android Graphics Canvas

Розглянемо множину методів класу android.graphics.Canvas, виклик яких означатиме дію графічних примітивів по відображенню чогось на растровій поверхні Canvas Bitmap. Перелік таких методів взято з керівництва розробника для Android і наведено нижче

Деякі методи класу android.graphics.Canvas

Method name	Comments
drawARGB	Fill the entire canvas' bitmap (restricted to the current clip) with the specified ARGB color, using srcOver porterduff mode
drawArc	Draw the specified arc
drawBitmap	Draw the bitmap
drawBitmapMesh	Draw the bitmap through the mesh, where mesh vertices are evenly distributed across the bitmap
drawCircle	Draw the specified circle using the specified paint
drawColor	Fill the entire canvas' bitmap (restricted to the current clip) with the specified color
drawDoubleRoundRect	Draws a double rounded rectangle using the specified paint
drawGlyphs	Draw array of glyphs with specified font
drawLine	Draw a line segment with the specified start and stop x,y coordinates, using the specified paint
drawLines	Draw a series of lines

drawMesh	Draws a mesh object to the screen
drawOval	Draw the specified oval using the specified paint
drawPaint	Fill the entire canvas' bitmap (restricted to the current clip) with the specified paint
drawPatch	Draws the specified bitmap as an N-patch (most often, a 9-patch.)
drawPath	Draw the specified path using the specified paint
drawPicture	Draw the picture, stretched to fit into the dst rectangle
drawPoint	Helper for drawPoints() for drawing a single point
drawPoints	Draw a series of points
drawRGB	Fill the entire canvas' bitmap (restricted to the current clip) with the specified RGB color, using srcover porterduff mode
drawRect	Draw the specified Rect using the specified paint
drawRegion	Draws the given region using the given paint
drawRenderNode	Draws the given RenderNode.
drawRoundRect	Draw the specified round-rect using the specified paint.
drawText	Draw the text, with origin at (x,y), using the specified paint.
drawTextOnPath	Draw the text, with origin at (x,y), using the specified paint, along the specified path.
drawTextRun	Draw a run of text, all in a single direction, with optional context for complex text shaping.
drawVertices	Draw the array of vertices, interpreted as triangles (based on vertex mode). Enum values of Canvas.VertexMode are: TRIANGLES TRIANGLE_FAN TRIANGLE_STRIP

Джерело: <https://developer.android.com/reference/android/graphics/Canvas>

Для багатьох методів малювання окрім певних параметрів, таких як координати розташування, параметри опису форми, посилання на деякі масиви, як правило, також вказується параметр-посилання на об'єкт класу Paint, завдяки якому можна визначити потрібні кольори, товщину ліній, стилі та режими малювання.

Розглянемо приклад малювання полігонального об'єкта. Як намалювати, наприклад, трикутник із заповненням? Або взагалі полігон з N вершинами? Для цього можна скористатися методом `drawPath()`, який має вельми багато можливостей. Потрібно створити об'єкт класу `Path` і у нього, як контейнер, можна додавати багато елементів-примітивів. Після того, як потрібні елементи додані в об'єкт `Path`, треба викликати метод `drawPath()`. Потім варто закрити `Path` викликом метода `close()`. Нижче наведений шаблон програмного коду для виводу зображення довільного полігону.

```
Paint p = new Paint();  
p.setStyle(Paint.Style.FILL);  
  
Path path = new Path();  
path.moveTo(x1, y1);           //first vertex  
path.lineTo(x2, y2);           //next vertexes  
...  
path.lineTo(xN, yN);           //last vertex  
path.lineTo(x1, y1);           //close the polygon outline  
canvas.drawPath(path, p);      //render polygon  
path.close();
```

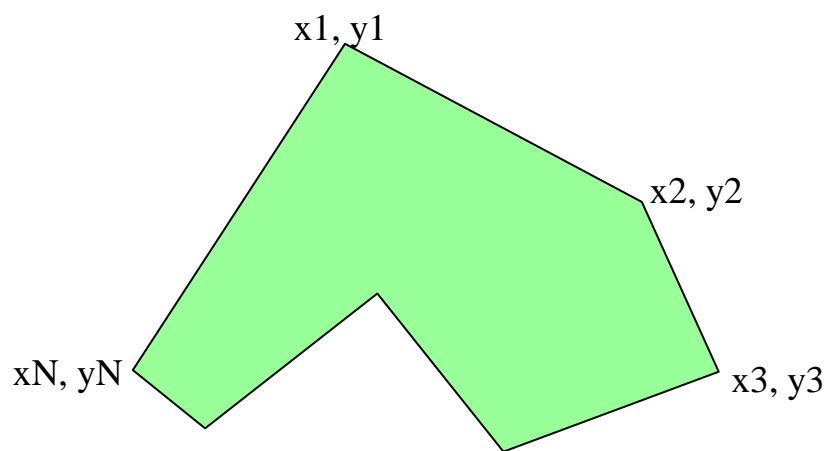


Рис. 15. Щодо відображення довільного полігону з використанням `drawPath`

2.4. Шаблон для програмування графіки Android Graphics Canvas

Для програмування графіки потрібно написати перевизначення (override) метода `onDraw` класу `View`. Для цього оголоسیمо похідний клас, наприклад, `myGraphicsView` і вставимо відповідний об'єкт у якості `View` у контекст `MainActivity`.

Нижче наведено програмний код для MainActivity, який можна вважати найпростішим шаблоном для програмування графіки Android Graphics Canvas.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.os.Bundle;
import android.view.View;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

    public class myGraphicsView extends View {
        myGraphicsView(Context context) {
            super(context);
        }

        @Override
        protected void onDraw(Canvas canvas) {
            Paint p = new Paint();

            // ... calling canvas.something,
            // for example canvas.drawOval
            // First we define the following colors and styles
            p.setARGB(255, 255, 0, 128);
            p.setStyle(Paint.Style.FILL_AND_STROKE);
            //and then we draw the figure directly
            canvas.drawOval(100,50,500,700, p);
        }
    }
}
```

Таким чином, у тілі метода onDraw буде виконуватися програмний код відображення графіки на екрані у вікні MainActivity застосунку для Android.

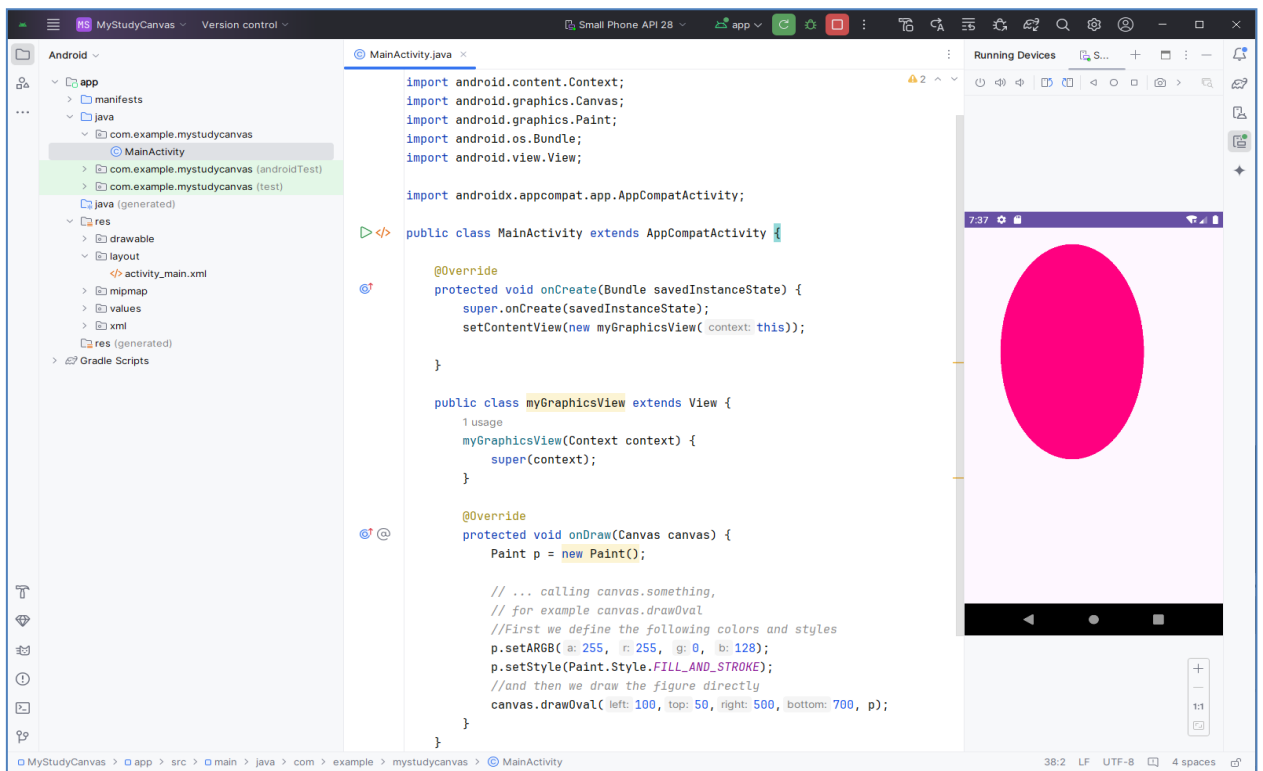


Рис. 16. Перевірка шаблону програмного коду для Android Graphics Canvas

2.5. Система координат у вікні застосунку для Android

Вісь X горизонтально зліва направо, вісь Y вертикально з гори до низу. Центр координат (0,0) у лівому верхньому куті області відображення.

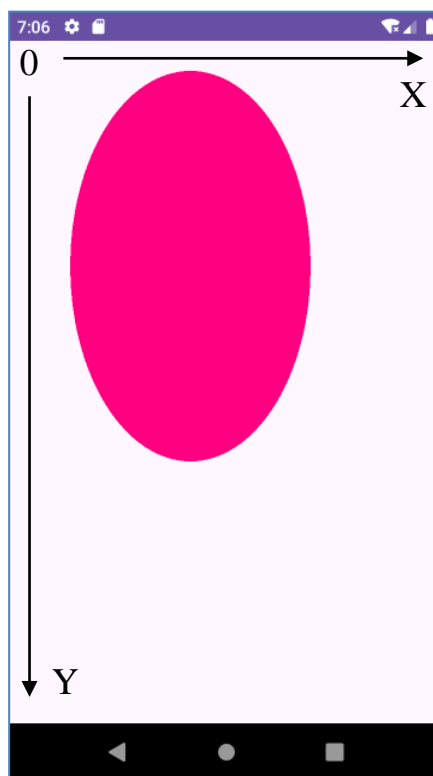


Рис. 17. Система координат виводу графіки у вікні MainActivity

Частина 3. Графіка Android OpenGL ES

OpenGL (*Open Graphics Library*) є інтерфейсом використання можливостей відеокарт та графічних процесорів для програмування високошвидкісної якісної графіки. OpenGL ES (*OpenGL for Embedded Systems*) – це інтерфейс для створення програмного забезпечення для мобільних (вбудованих) пристроїв, зокрема, пристроїв Android.

3.1. Створення проєкту Android Studio OpenGL ES

Створення проєкту OpenGL ES в середовищі Android Studio так само, як вже було розглянуто вище. А тепе назвемо новий проєкт MyStudyGL_First.

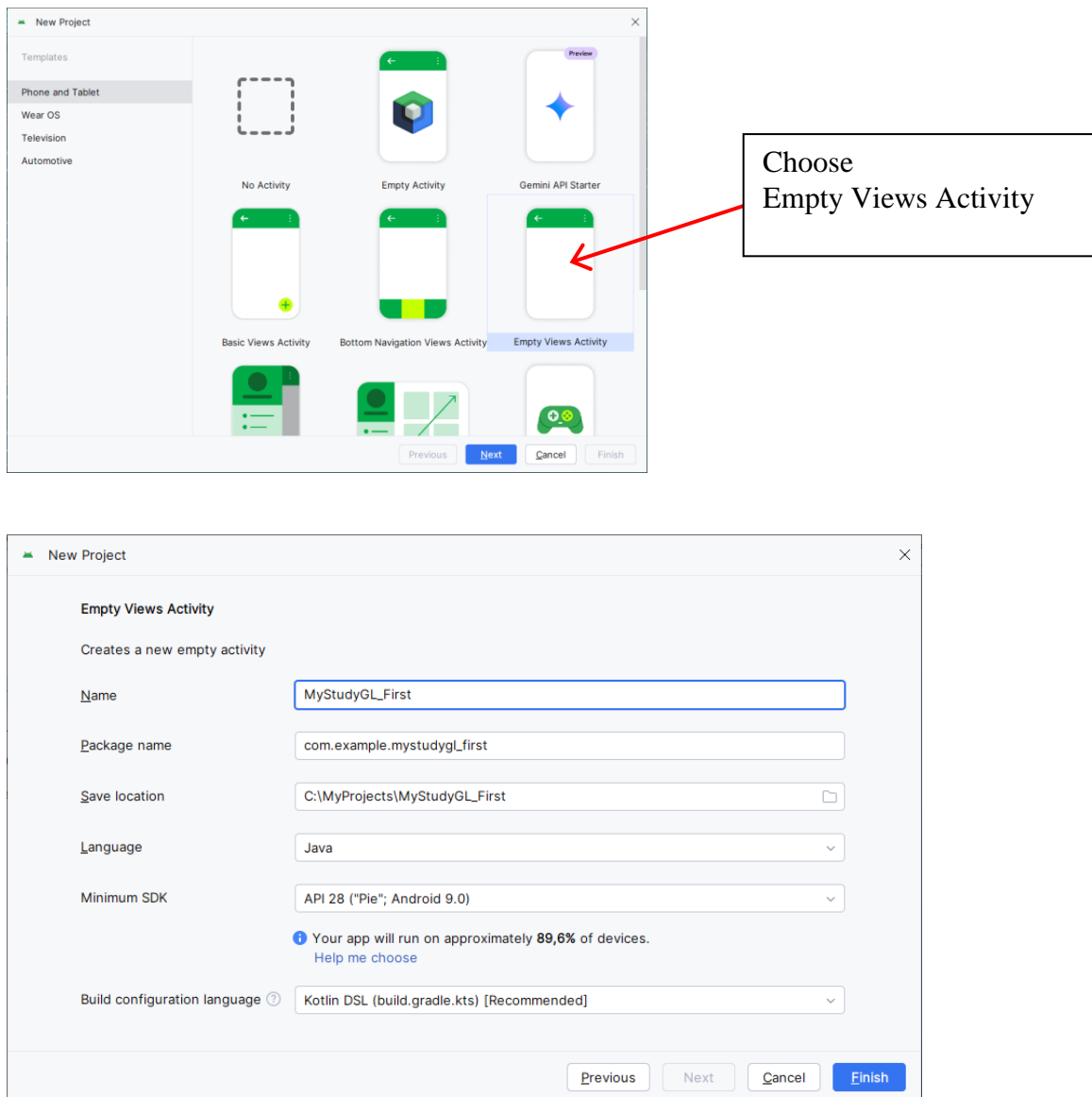


Рис. 18. Створення в Android Studio проєкту MyStudyGL_First

Як бачимо, створення нового проекту виконано так само, як у попередній частині лабораторної роботи, проте для підтримки OpenGL ES 3.2 обов'язково потрібно обрати API не менше 28 (Android 9.0).

Після натискання кнопки [Finish] далі Android Studio автоматично генерує вихідні тексти початкового стану проекту і показує вміст файлу MainActivity.java у головному вікні

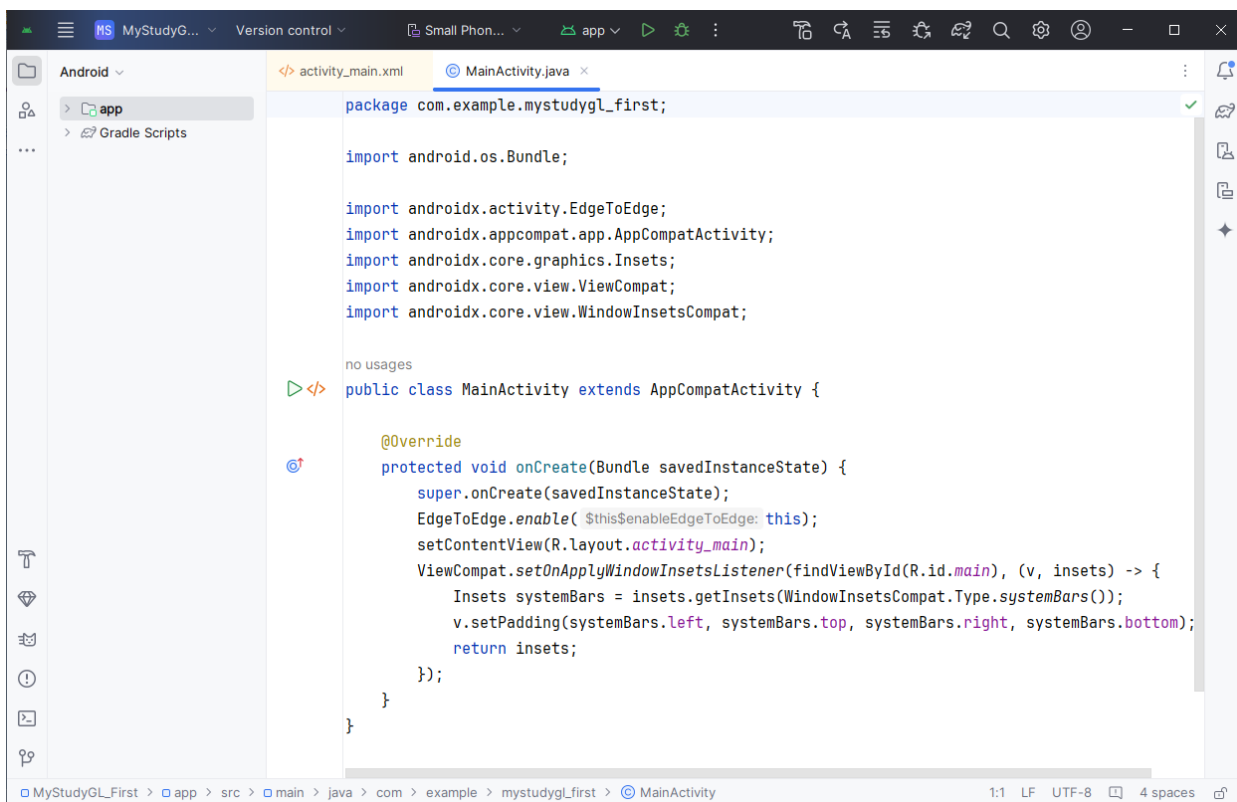


Рис. 19. Вигляд проекту MyStudyGL_First в середовищі Android Studio одразу після створення

3.2. Найпростіший шаблон застосунку з графікою OpenGL ES

Для того, щоб включити графіку OpenGL ES в Activity, можна порекомендувати скористатися класом GLSurfaceView. Створимо свій клас MyGLSurfaceView, похідний від GLSurfaceView. Далі об'єкт класу MyGLSurfaceView викликом setContentView оберемо у якості контейнера (робочої поверхні) виводу графіки в MainActivity. Це можна зробити один раз на початку роботи Activity, наприклад, в методі onCreate.

Крім того, потрібно імплементувати методи інтерфейсу GLSurfaceView.Renderer, зокрема, такі методи, як onSurfaceCreated, onSurfaceChanged та onDrawFrame. При написанні програмного коду графіки в методах, які імплементуються від GLSurfaceView.Renderer у власному класі, наприклад, з ім'ям, потрібно враховувати як та коли будуть викликатися ці методи. Так, зокрема, метод onSurfaceCreated викликається один раз від початку, тому варто туди помістити якісь початкові налаштування рендерингу, наприклад, визначення кольору фону.

Метод onSurfaceChanged, як слідує з його назви, викликається тоді, коли змінюються розміри поверхні відображення. Так, наприклад, якщо гаджет перевернути на 90 градусів – тоді змінюються пропорції екрану і нові розміри передаються через параметри width та height метода onSurfaceChanged.

Основну роль для рендерингу поточного стану сцени виконує метод onDrawFrame. Цей метод буде викликатися у залежності від того, який режим буде вказано значенням параметру метода setRenderMode():

- GLSurfaceView.RENDERMODE_WHEN_DIRTY означатиме виклик onDrawFrame лише один раз. Для того, щоб відобразити зображення сцени наступного разу, потрібно викликати функцію requestRender();
- GLSurfaceView.RENDERMODE_CONTINUOUSLY – автоматичне циклічне повторення викликів метода onDrawFrame. Цей режим рендерингу можна використати, зокрема, для анімації.

Для того, щоб почати вивчати програмування графіки на основі OpenGL ES, можна запропонувати спочатку найпростіший шаблон програмного коду на Java для MainActivity. Назвемо його Шаблон1.

```

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

import android.opengl.GLSurfaceView;
import android.opengl.GLES32;

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

public class MainActivity extends AppCompatActivity {

    private GLSurfaceView glView;

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

    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
        }
    }

    public class MyGLRenderer implements GLSurfaceView.Renderer {

        public void onSurfaceCreated(GL10 unused, EGLConfig config) {
            // Set the background frame color (pale blue for example)
            GLES32.glClearColor(0.8f, 0.9f, 1.0f, 1.0f);
        }

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

        public void onDrawFrame(GL10 unused) {
            GLES32.glClear(GLES32.GL_COLOR_BUFFER_BIT);

            // . . . TODO some rendering code
        }
    }
}

```

А тепер вставимо програмний код цього Шаблону1 у файл MainActivity.java у якості класу MainActivity. Виконаємо Run – перевіримо роботу програми на емуляторі пристроя Android.

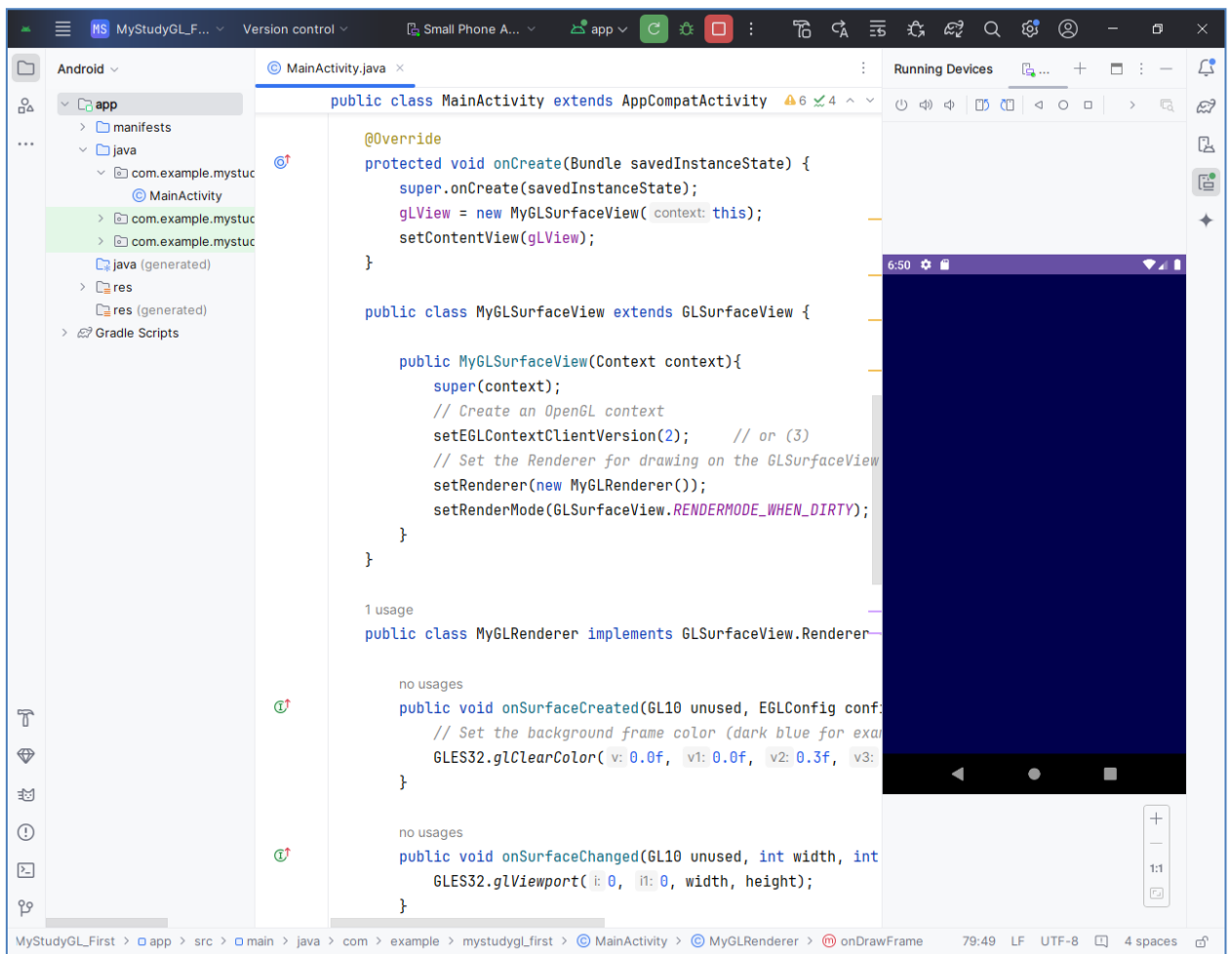


Рис. 20. Перевірка роботи коду відповідно Шаблону1

Якщо помилок немає, то побачимо вікно з темно-синьою площиною відображення – згідно з командою

```
GLS32.glClearColor(0.0f, 0.0f, 0.3f, 1.0f);
```

3.3. Базові графічні примітиви OpenGL ES

Зображення об'єктів формується з множини простих елементів – графічних примітивів. Можна сказати, що базовими графічними примітивами в OpenGL ES є: точка, відрізок прямої лінії, трикутник. Малювання, графічний вивід таких примітивів виконується викликом метода `glDrawArrays()` під час роботи метода `onDrawFrame`.

```
public void onDrawFrame(GL10 unused) {
    . . .
    GLES32.glDrawArrays(mode, startindx, count);
    . . .
}
```

Параметри `glDrawArrays`:

mode: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`,
`GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`;

startindx: початковий індекс в буфері вершин;

count: кількість вершин за один виклик `glDrawArrays`.

Так, наприклад, для виводу одного трикутника можна записати

```
GLES32.glDrawArrays(GLES32.GL_TRIANGLES, 0, 3);
```

На рис. проілюстровано роботу `glDrawArrays` для різних `mode`

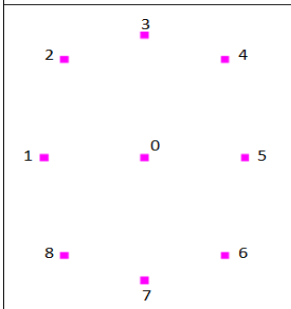
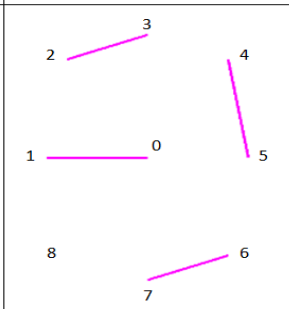
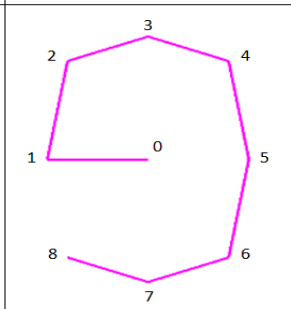
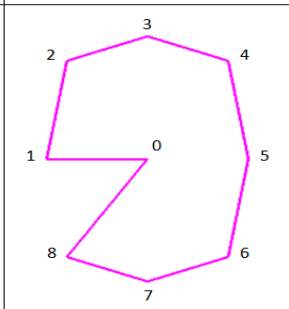
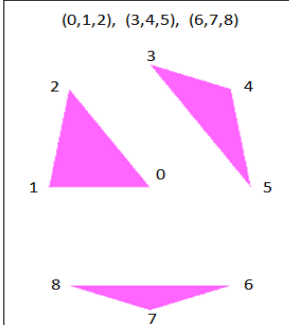
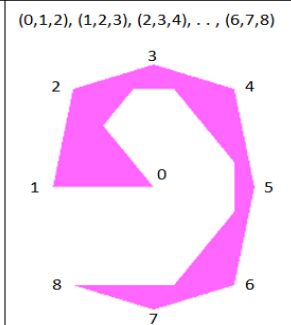
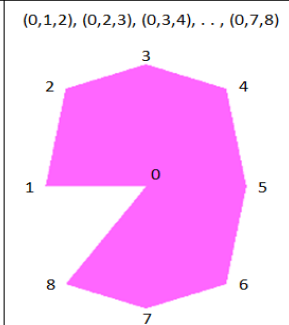
GL_POINTS	GL_LINES	GL_LINE_STRIP	GL_LINE_LOOP
			
	GL_TRIANGLES	GL_TRIANGLE_STRIP	GL_TRIANGLE_FAN
			

Рис. 21. Основні різновиди графічних примітивів для `glDrawArrays`

Варто зазначити, що в OpenGL ES версії 3.2 для `glDrawArrays` відкинуті деякі значення `mode`, які були у перших версіях OpenGL, зокрема `GL_QUADS` та `GL_POLYGON`. Чому такі режими (графічні примітиви) були потім відкинуті і тепер їх вже немає? Можна сказати, головною причиною цього було намагання досягнути максимальну продуктивність для графічних процесорів. Пов'язаним з цим було також вирішення проблем рендерінгу довільних полігонів – якщо трикутник завжди є опуклою фігурою і усі його вершини завжди в одній площині, то чотирикутники та полігони взагалі можуть бути і не опуклими, а їхні вершини можуть і не належати одній площині.

Як намалювати чотирикутник засобами OpenGL ES? Взагалі будь який полігон або багатогранну поверхню можна представити множиною трикутників (триангулювати). Чотирикутник можна відобразити трикутниками двома способами

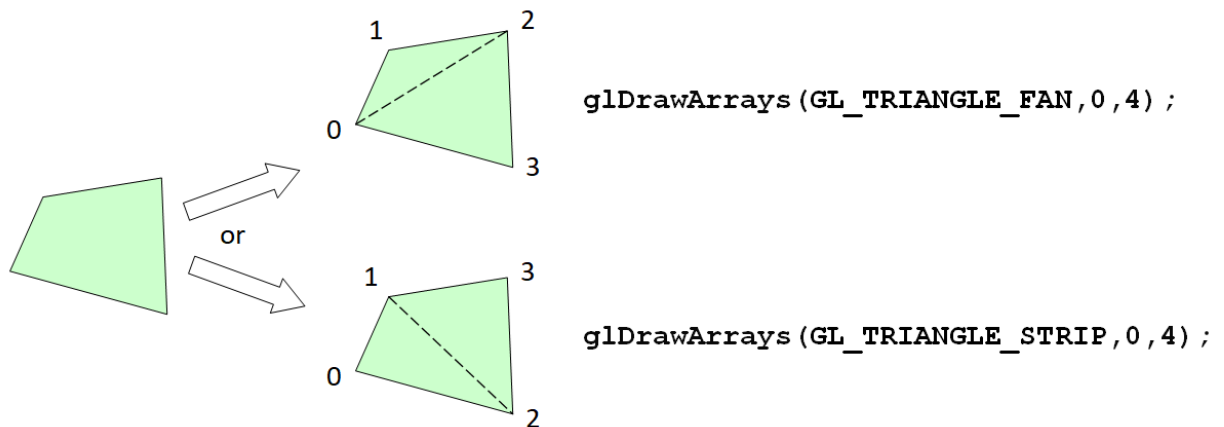


Рис. 22. Два варіанти триангуляції чотирикутника

Зверніть увагу на відмінність індексації вершин чотирикутника для наведених вище двох варіантів триангуляції. Крім того, в режимі `GL_TRIANGLE_FAN` порядок обходу вершин для усіх трикутників є однаковим – у даному прикладі по годинниковій стрілці. А у режимі `GL_TRIANGLE_STRIP` порядок обходу вершин суміжних трикутників не є однаковим. Для простих режимів показу об'єктів це не критично, проте може викликати певні складнощі при автоматичному обчисленні векторів нормалей при моделюванні освітлення. Це саме стосується не тільки чотирикутників а й взагалі представлення полігональних (багатограних) поверхонь.

3.4. Просте відображення трикутника у реальній програмі

У навчанні програмуванню графіки OpenGL ES, як і у навчанні чомусь іншому, варто рухатися від простого, поступово засвоюючи більш складні моменти. Але для того, щоб зробити вже перший крок програмування графіки навіть для простих об'єктів, треба засвоїти достатньо багато інформації щодо особливостей організації графіки OpenGL ES. Необхідно забезпечити виконання нашою програмою, як мінімум, наступних дій:

1. Підготувати та завантажити шейдерну програму. А для цього необхідно:
 - написати шейдер вершин (*Vertex Shader*) та фрагментний шейдер (*Fragment Shader*);
 - скомпілювати та завантажити шейдери;
 - скомпілювати та завантажити шейдерну програму
2. Створити та завантажити в масив координат вершин об'єкта. А для цього треба:
 - визначити масив координат вершин об'єкта;
 - сформувати об'єкт масиву вершин (*VAO – Vertex Array Object*);
 - сформувати об'єкт буфера вершин (*VBO – Vertex Buffer Object*);
 - під'єднати-завантажити об'єкти VAO, VBO;
 - визначити посилання шейдерних атрибутів для масиву координат вершин
3. Записати у програмному коді метода `onDrawFrame` виклик функції `GL ES32.glDrawArrays()` безпосередньо вже для рендерінгу об'єктів сцени.

3.4.1. Шейдери для першого прикладу

Шейдер (англ. *shader*) – це програма, яка завантажується у пам'ять відеокарти і виконується графічним процесором (*GPU*) безпосередньо в ході рендерінгу. Програмний код шейдерів для OpenGL ES має вигляд невеличкої програми з C-подібним синтаксисом. Для нашої програми `MyStudyGL_First` текст шейдера вершин буде таким:

```
#version 300 es
in vec3 vPosition;
void main() {
    gl_Position = vec4(vPosition, 1.0f);
}
```

Цей шейдер просто сприймає у форматі `vec3` вектори 3-d координат об'єкта з масиву вершин, перетворює у формат чотирьохкомпонентних векторів (`vec4`) і передає координати далі для наступних блоків графічному конвеєру, записуючи результат у вихідну перемінну `gl_Position`.

Фрагментний шейдер формує колір пікселів у растровому зображенні рендерованої сцени. Нижче текст фрагментного шейдера

```
#version 300 es
precision mediump float;
out vec4 outColor;
void main() {
    outColor = vec4(1.0f, 0.5f, 0.0f, 1.0f);
}
```

У цьому шейдері визначено помаранчевий колір, однаковий для усіх пікселів – точок об'єктів на растрі рендерингу. Результат у форматі четвірки (R,G,B,A) записується у вихідну перемінну, яку визначає програміст (у даному прикладі це перемінна з ім'ям `outColor`).

3.4.2. Масив координат вершин об'єкта

Визначимо 3d координати XYZ вершин трикутника наступними значеннями

```
float[] triangleCoords = new float[] {
    0.5f, -0.5f, 0.0f,    // Bottom Right
    -0.5f, -0.5f, 0.0f,  // Bottom Left
    0.0f, 0.5f, 0.0f};   // Top
```

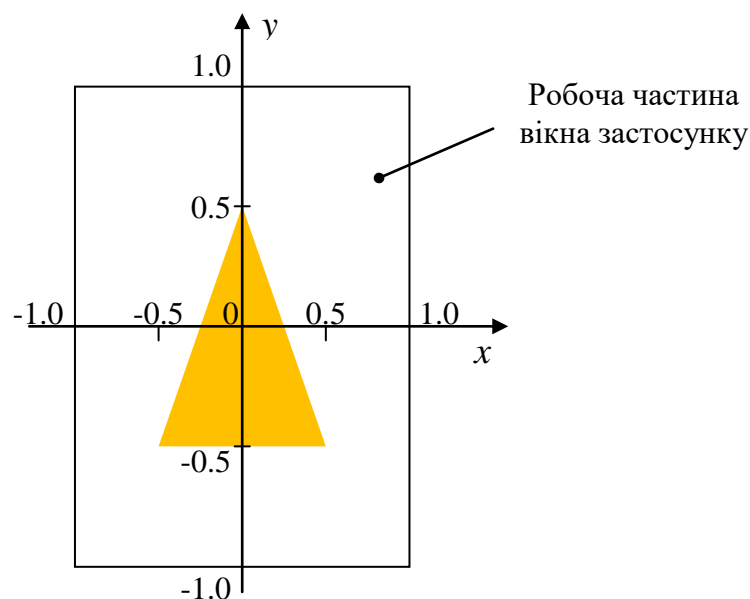


Рис. 23. Координати для рендерингу у робочій частині вікна застосунку

Зауваження. Для визначення координат трикутника використані тривимірні координати (XYZ), хоча у цьому першому прикладі, як здається, запрограмована суто двовимірна графіка. Це так, але в данному випадку вісь Z опису координат трикутника спрямовується перпендикулярно площині XOY – на нас. OpenGL ES працює з тривимірними координатами.

Варто враховувати, що оскільки значення координат у вікні рендерингу в діапазоні від -1.0 до +1.0, то в ідеалі екран пристроя має бути квадратним. А якщо розміри екрану в пікселях є різними по ширині та по висоті, то відповідно змінюються пропорції об'єктів, що відображаються. Так, зокрема, якщо висота вікна виводу більша ширини, то об'єкт немов би витягнутий по вертикалі. Виправлення такого дефекту відображення буде розглянуто далі у наступних прикладах.

3.4.3. Програмний код застосунка

Файл MainActivity.java

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

import android.opengl.GLSurfaceView;
import android.opengl.GLES32;

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

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

public class MainActivity extends AppCompatActivity {
    private GLSurfaceView glView;

    protected int gl_Program = 0;           //shader program
    protected int VAO_id = 0;               //Vertex Array Object id
    protected int VBO_id = 0;               //Vertex Buffer Object id

    private String vertexShaderCode1 =
        "#version 300 es\n" +
        "in vec3 vPosition;\n" +
        "void main() {\n" +
        "    gl_Position = vec4(vPosition, 1.0f);\n" +
        "}\n";

    private String fragmentShaderCode1 =
        "#version 300 es\n" +
        "precision mediump float;\n" +
        "out vec4 outColor;\n" +
        "void main() {\n" +
        "    outColor = vec4(1.0f, 0.5f, 0.0f, 1.0f);\n" +
        "}\n";

    private float[] triangleCoords = new float[] {
        0.5f, -0.5f, 0.0f,    // Bottom Right
        -0.5f, -0.5f, 0.0f,   // Bottom Left
        0.0f,  0.5f, 0.0f};   // Top
```

```

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

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
    }
}

public class MyGLRenderer implements GLSurfaceView.Renderer {

    public void onSurfaceCreated(GL10 unused, EGLConfig config) {
        // Set the background frame color (dark 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);
    }

    public void onDrawFrame(GL10 unused) {
        GLES32.glClear(GLES32.GL_COLOR_BUFFER_BIT);

        //---make and attach shader program---
        int vertex_shader_id = GLES32.glCreateShader(
            GLES32.GL_VERTEX_SHADER);
        GLES32.glShaderSource(vertex_shader_id, vertexShaderCode1);
        GLES32.glCompileShader(vertex_shader_id);

        int fragment_shader_id = GLES32.glCreateShader(
            GLES32.GL_FRAGMENT_SHADER);
        GLES32.glShaderSource(fragment_shader_id, fragmentShaderCode1);
        GLES32.glCompileShader(fragment_shader_id);

        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);

        //---make and bind vertex arrays---
        ByteBuffer bb = ByteBuffer.allocateDirect(
            triangleCoords.length*4);
        bb.order(ByteOrder.nativeOrder());

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

        //VAO and VBO id's
        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

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

```

```

// then bind and set vertex buffer
GL ES32.glBindBuffer(GL ES32.GL_ARRAY_BUFFER, VBO_id);
GL ES32.glBufferData(GL ES32.GL_ARRAY_BUFFER,
                    triangleCoords.length*4,
                    vertexBuffer, GL ES32.GL_STATIC_DRAW);

// then define attribute pointer
int handle = GL ES32.glGetAttribLocation(gl_Program, "vPosition");
GL ES32.glEnableVertexAttribArray(handle);
GL ES32.glVertexAttribPointer(handle, 3,
                             GL ES32.GL_FLOAT, false, 3*4, 0);
GL ES32.glEnableVertexAttribArray(0);

GL ES32.glBindBuffer(GL ES32.GL_ARRAY_BUFFER, 0);

// Rendering
GL ES32.glUseProgram(gl_Program);
GL ES32.glDrawArrays(GL ES32.GL_TRIANGLES, 0, 3);
GL ES32.glBindVertexArray(0);
}
}
}

```

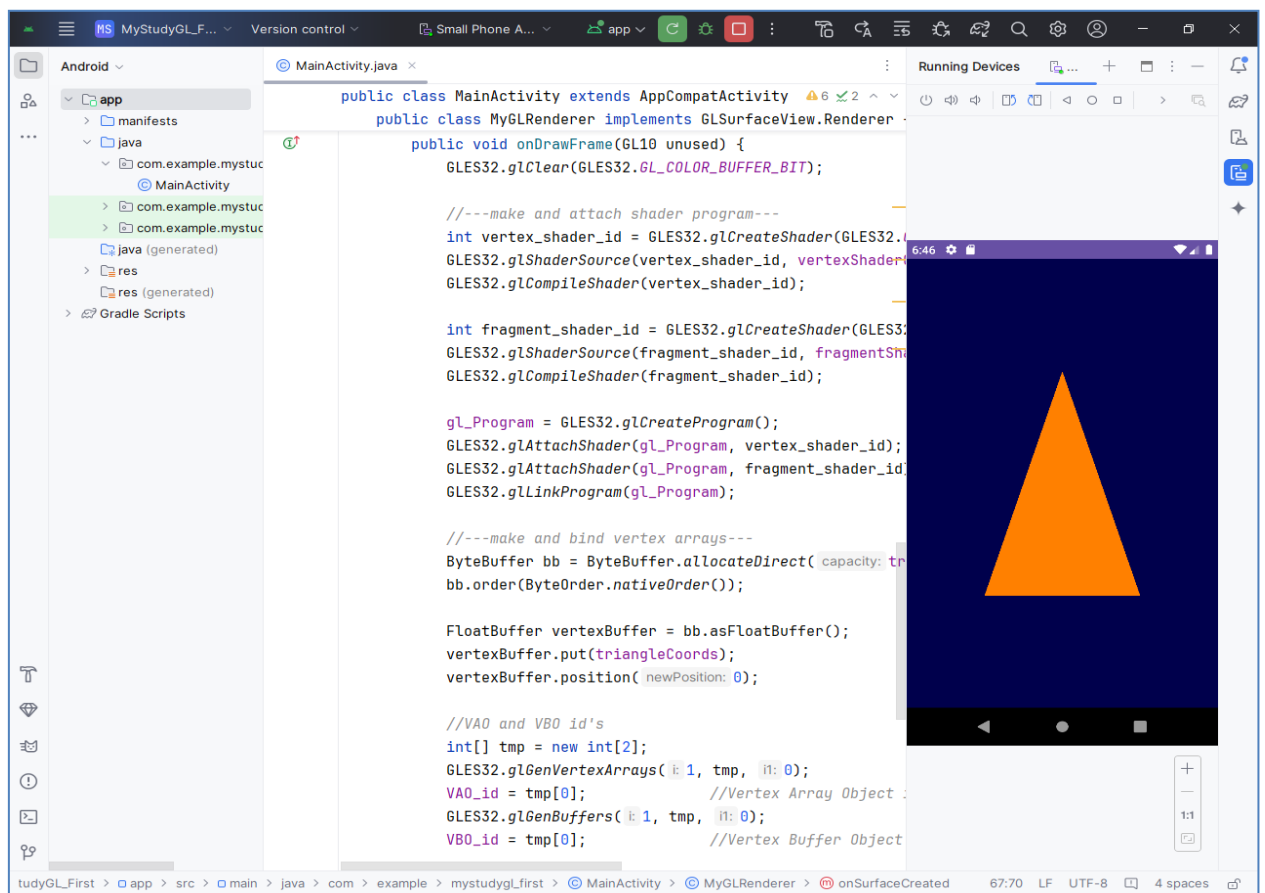


Рис. 24. Перевірка роботи застосунка в емуляторі

Незважаючи на те, що застосунок, як здається, успішно працює, проте рішення для цього проекту є вельми недосконалим. По-перше, програмний код обсягом 135 рядків є громіздким, доволі неструктурованим, двічі повторюються однакові рядки створення та компіляції шейдерів тощо. А по-друге, і це головний недолік – усі дії по створенню, компіляції шейдерів та

шейдерної програми, формування масивів та буферів вершин записані в тілі метода `onDrawFrame` і, відповідно, виконуються заново при кожному виклику `onDrawFrame` коли потрібно перемалювати кадр зображення сцени. Якщо рендеринг потрібно виконати лише один раз – як у цьому навчальному застосунку, то це може вважатися прийнятним. Але, якщо уявити, що достатньо ресурсомісткі операції компіляції, завантаження шейдерних програм а також формування та завантаження масивів та буферів вершин будуть виконуватися для кожного кадру рендерингу в режимі анімації, тоді це буде явно неприйнятним.

Важливо навчитися розподіляти дії програми, у тому числі й використання функцій OpenGL, так щоб довготривалі операції, такі як компіляція шейдерів, формування буферів вершин, завантаження текстур тощо, виконувалися на підготовчих етапах – перед викликами `onDrawFrame`.

3.4.4. Рефакторинг коду. Друга версія шаблону

Виконаємо рефакторинг програмного коду. По-перше, зробимо код структурованим – розділимо його на окремі методи, а по-друге, винесемо по-можливості усі підготовчі дії з циклу можливих частих викликів `onDrawFrame` для рендерингу сцени.

Програмний код не просто розподілимо в окремих методах, а оформимо такі методи у складі нового класу у окремому java-файлі. Назвемо цей клас `myWorkMode`. Нижче наведений програмний код проєкту `MyStudyGL_First` після рефакторингу. Програмний код складається вже з декількох класів, розподілених по різних файлам-модулям.

Файл MainActivity.java

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

import android.opengl.GLSurfaceView;
import android.opengl.GLES32;

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);
        wmRef = new mySimplestTriangleMode(); //set concrete example state
    }

    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
        }
    }

    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.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) {
            GLES32.glClear(GLES32.GL_COLOR_BUFFER_BIT);
            if (wmRef == null) return;
            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);
        }
    }
}
```

Файл 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;

    myWorkMode() {
        gl_Program = 0;
        VAO_id = VBO_id = 0;
    }

    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 = 0;
        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) {

    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);
}

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);
}
}

```

Програмний код модулів-класів файлів **MainActivity.java** та **myWorkMode.java** можна вважати Шаблоном2 – шаблоном вже структурованого рішення. У класі MainActivity буде міститися програмний код верхнього рівня – ініціалізації застосунку, підтримка інтерфейсу користувача. Клас myWorkMode містить методи, які інкапсують подробиці компіляції шейдерів та активування шейдерних програм, формування масивів і буферів VAO та VBO. Клас myWorkMode містить також методи, які призначені для імплементації конкретних прикладів сцен у похідних класах. Більше того, для метода myUseProgramForDrawing визначена реалізація по замовчуванню, яка може бути використана для достатньо великого різноманіття конкретних сцен, об'єкти яких описуються множиною окремих трикутників.

Клас `mySimplestTriangleMode` імплементує конкретну сцену – поки що це дуже проста сцена з одним об’єктом відображення – трикутником.

Файл `mySimplestTriangleMode.java`

```
public class mySimplestTriangleMode extends myWorkMode {

    mySimplestTriangleMode() {
        super();
        myCreateScene();
    }

    @Override
    protected void myCreateScene() {
        arrayVertex = new float[] {           // triangle vertex coords
            0.5f, -0.5f, 0.0f,                // Bottom Right
            -0.5f, -0.5f, 0.0f,               // Bottom Left
            0.0f, 0.5f, 0.0f};                // Top
        numVertex = 3;
    }

    @Override
    public void myCreateShaderProgram() {
        myCompileAndAttachShaders(
            myShaders.vertexShaderCode1,
            myShaders.fragmentShaderCode1);
        myVertexArrayBind(arrayVertex, "vPosition");
    }
}
```

Таким чином у класі `mySimplestTriangleMode` визначаються лише властивості конкретної сцени – значення координат вершин, їхня кількість а також посилання на відповідні шейдери. Можна сказати, що такий підхід може суттєво полегшити вивчення програмування графіки OpenGL ES наступних прикладів. В ідеалі було б так, щоб базовий клас `myWorkMode` залишився вже незмінним. Але він поки що інкапсулює замало можливостей від OpenGL ES, тому згодом цілком можливо, що доведеться розширити і його.

Щодо шейдерів. Вочевидь, для різних типів сцен можуть знадобитися відповідно різні унікальні шейдери. Але загалом, може бути сумнівним рішенням інкапсулювати програмний код конкретних шейдерів в класі опису конкретної сцени, такому як `mySimplestTriangleMode`. Може статися, що для деяких класів реалізації різних сцен можуть використовуватися однакові шейдери – але не обов’язково це буде повторне використання тієї самої пари шейдерів (шейдер вершин + фрагментний шейдер). Це робить проблемним можливості вільного успадкування класів конкретних сцен. Тому представляється доцільним винести програмний код шейдерів у окремий

клас як деяку бібліотеку, яку можна буде потім розширювати та повторно спільно використовувати.

Файл **myShaders.java**

```
public class myShaders {  
  
    public static final String vertexShaderCode1 =  
        "#version 300 es\n" +  
        "in vec3 vPosition;\n" +  
        "void main() {\n" +  
        "    gl_Position = vec4(vPosition, 1.0f);\n" +  
        "}\n";  
  
    public static final String fragmentShaderCode1 =  
        "#version 300 es\n" +  
        "precision mediump float;\n" +  
        "out vec4 outColor;\n" +  
        "void main() {\n" +  
        "    outColor = vec4(1.0f, 0.5f, 0.0f, 1.0f);\n" +  
        "}\n";  
}
```

Таким чином, замість одного файлу у проєкті MyStudyGL_First програмний код розподілено вже по чотирьом класам. Проєкт стає більш структурованим та зручнішим для подальшого вдосконалення та розширення.

Для перевірки результатів рефакторингу подивимося роботу застосунку в емуляторі Android-пристрою.

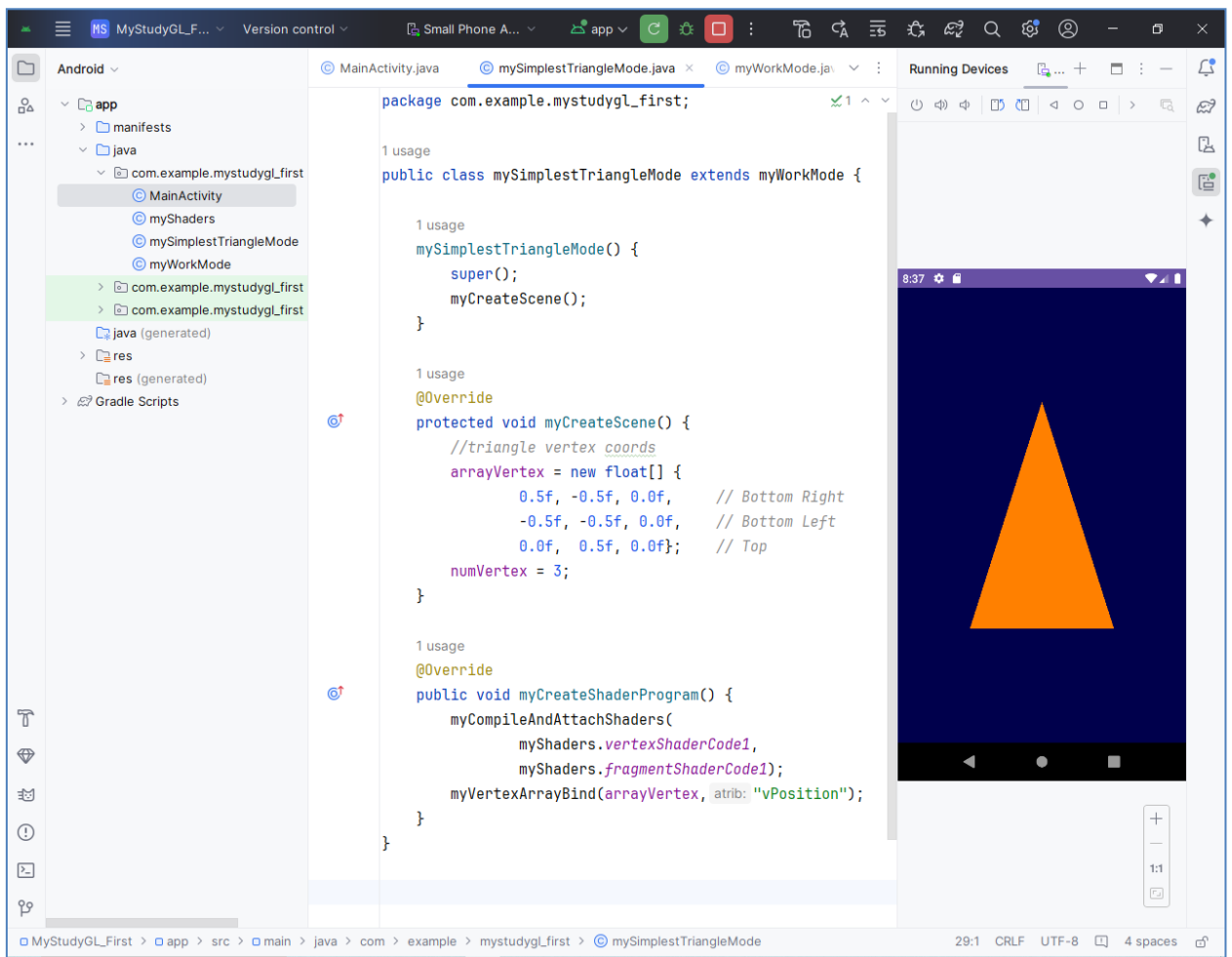


Рис. 25. Перевірка коректності роботи застосунку

Зауваження. Як можна бачити у вікні емулятора, трикутник виглядає витягнутим по висоті. Це тому, що висота робочої частини вікна MainActivity більша, ніж його ширина. Але ж відповідно координатами вершин ширина та висота трикутника однакова і дорівнює 1. Як виправити похибку відображення? Для створення ілюзії правильності пропорцій відображення цього трикутника можна було б порекомендувати зменшити координату Y верхньої вершини. Тобто, для вершини, яка позначена у вихідному тексті як // Top – замість 0.5f задати, скажімо 0.3f (це ще залежить від розмірів екрану вашого Android-пристрою).

У подібний спосіб можна було б рекомендувати стиснути у-координати для інших геометричних об'єктів. Для лаб1 це можна вважати прийнятним, але взагалі такий підхід є поганим. Модель повинна правильно визначати об'єкти незалежно від особливостей вікна відображення. У лаб. №3 ми розглянемо дієвий спосіб компенсації таких похибок відображення.

3.5. Щодо можливостей виводу множини об'єктів

А якщо потрібно виводити не один, а багато об'єктів? Тоді варто відзначити наступні моменти.

Якщо усі об'єкти сцени складаються із множини окремих трикутників, то це буде найпростіший випадок. Скільки б трикутників не було б – якщо усі їхні вершини записані у масиві вершин VAO, то для рендерингу їх усіх разом достатньо одного виклику `glDrawArrays`

```
GLES32.glDrawArrays(GLES32.GL_TRIANGLES, 0, numVertex);
```

А тепер уявимо, що сцена містить деякі об'єкти, для представлення яких загалом потрібно декілька прямих ліній (`GL_LINES`), одну полігональну поверхню визначену як `GL_TRIANGLE_STRIP`, та декілька окремих трикутників. Тоді рішенням може бути наступний програмний код рендерингу

```
//render set of lines
GLES32.glDrawArrays(GLES32.GL_LINES,
                    startVertexLines, numVertexLines);

//render one triangle strip
GLES32.glDrawArrays(GLES32.GL_TRIANGLE_STRIP,
                    startVertexStrip, numVertexStrip);

//render set of triangles
GLES32.glDrawArrays(GLES32.GL_TRIANGLES,
                    startVertexTriangles, numtVertexTriangles);
```

Взагалі, необхідно вказати, що навіть ту саму однакову сцену можна описати по-різному. Відповідно, побудова структур даних та рендерінг можуть суттєво відрізнятися. Також можна вказати, що є деяке протиріччя між швидкодією та витратами пам'яті. І усе це повною мірою стосується програмування графіки на основі OpenGL.

Варіанти завдань та основні вимоги

1. Потрібно створити три програмних застосунка:

- **Lab1** на основі GDI Windows
- **Lab1_Canvas** на основі Android Graphics Canvas
- **Lab1_GLES** на основі Android OpenGL ES

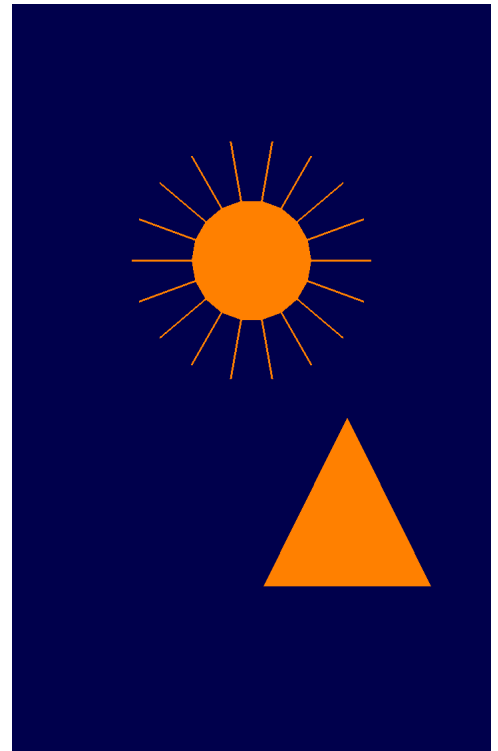
2. Кожний з трьох застосунків п.1 повинен відображати у головному вікні однакове зображення, наведене праворуч, і яке містить:

- фонове зафарбовування площини вікна кольором C;
- N-кутник з N промінями («сонце») кольору CO;
- трикутник кольору CO

3. Номер варіанту (V) обчислюється як залишок від ділення номера залікової книжки (НЗК) на 8

$$V = \text{НЗК} \bmod 8$$

4. Кольори C, CO та кількість N згідно номеру варіанту V з таблиці нижче



Номер варіанту (V)	Колір фону (C)	Колір об'єктів (CO)	Кількість кутів (N)
0	Світло-Сірий	Синій	18
1	Чорний	Червоний	20
2	Сірий	Малиновий	16
3	Світло-рожевий	Чорний	22
4	Світло-рожевий	Темно-зелений	24
5	Білий	Червоний	32
6	Темно-синій	Червоний	26
7	Світло-синій	Жовтий	28

Зміст звіту

1. Титульний аркуш
2. Варіант завдання
3. Вихідний тексти файлів *.cpp, *.java
4. Ілюстрації (скріншоти)
5. Висновки

Контрольні запитання

1. Що таке HDC для графіки GDI Windows та як отримати hdc вікна застосунку?
2. Як можна запрограмувати трикутник із заповненням та без заповнення (лише контур) на GDI Windows?
3. Що таке пензль (Brush) GDI Windows?
4. Як визначити колір об'єктів малювання в Android Graphics Canvas?
5. Як можна запрограмувати трикутник із заповненням та без заповнення (лише контур) в Android Graphics Canvas?
6. Як можна запрограмувати кольоровий фон зафарбовування площини відображення для вікна MainActivity застосунку Android?
7. Що таке шейдер в OpenGL ES?
8. Як можна запрограмувати коло із заповненням в OpenGL ES?