КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

Оптимізаційні техніки рендерингу графіки

Виконав студент 4-го курсу
Бондар Денис ___________________
(підпис)

Науковий керівник:
доктор фізико-математичних наук
Юрій Крак ___________________
(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _______________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики
«____»_________________2023 р., протокол № ____
Завідувач кафедри
Юрій КРАК ______________
(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 44 сторінок, 34 ілюстрацій, 23 джерела посилань.

Об'єктом роботи є вивчення оптимізаційних технік рендерингу графіки.
Предметом роботи є реалізація рендер-рушія Dengine за допомогою графічного інтерфейсу OpenGL та C#.

Метою роботи є розроблення та програмна реалізація рендер-рушія для вивчання комп'ютерної графіки.

Інструменти розроблення: інтегроване середовище розробки: Rider, OpenTK (бібліотека-обгортка OpenGL для C#), ImGUI.NET.

Результати роботи: реалізовано рендер-двигун Dengine та на його основі виконано 2 лабораторні роботи з «Інтелектуальних систем»: Пакман та генетичний алгоритм (генерація зображення), а також досліджено деякі оптимізаційні техніки рендерингу графіки.

Розроблене програмне забезпечення може використовуватись навчальними закладами середньої та вищої освіти. Може бути додатково змінено та використано в усіх галузях, де потрібна візуалізація.

ЗМІСТ

Скорочення та умовні позначення 3

Вступ 3

1 Статичний батчинг (Static batching) 3

1.1 Статичний батчинг з модельними матрицями 11

2 Динамічний батчинг (Dynamic batching) 19

3 Дублювання геометрії (GPU instancing) 25

4 Шейдери обчислення (Compute shaders) 30

5 Рівень деталізації (LOD) 37

Висновок 42

Перелік джерел 42

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

CPU – центральний процес

GPU – графічний процесор

VBO – вершинний буфер

VAO – інтерфейс меша, масив вершинних буферів

DC – draw call (команда до GPU намалювати об’єкт)

FPS – frames per second (кількість кадрів в секунду)

ВСТУП

Оптимізація рендеру – комплексне завдання, яке вимагає від графічного програміста певних знань та навичок. Першочерговою, звісно, є профілювання окремого кадру з просадкою FPS, щоб локалізувати вузьке місце, а потім оптимізувати його. У комп'ютерній графіці є дуже багато речей, які можуть негативно вплинути на продуктивність програми:

- Неефективне спілкування з GPU: часте звертання CPU до GPU або GPU до CPU
- Неефективне керування пам'яттю GPU, наприклад, перевиділення буферів в процесі виконання програми, тощо
- Світло: техніки, як ray-tracing чи global illumination.
- Висока роздільність екрану: рендер з 4K або вище вимагає більше обчислювальної потужності та пам'яті
- Екранне згладжування
- Пост-процес
- Висока складність сцени

Тут наведено лише деякі пункти, які можуть негативно впливати на продуктивність рендеру, проте в цій роботі розглядається оптимізація останнього пункту – складність сцени, так як він частіше всього є причиною просадки FPS.

Окрім очевидного: тримати моделі якомога з меншою кількістю полігонів та використовувати більш оптимізовані шейдери, розберемо як і коли саме малювати об'єкти.

1 СТАТИЧНИЙ БАТЧИНГ (STATIC BATCHING)

У графічному програмуванні статичний батчинг — це техніка для оптимізації рендерингу великої кількості об’єктів на сцена.

Статичний батчинг працює шляхом поєднання кількох окремих мешів в один великий меш, який згодом можна намалювати за один DC.

Статичний батчинг особливо корисний для сцен, які містять велику кількість маленьких статичних об’єктів, які не змінюють положення, обертання чи масштаб протягом гри.

Припустімо, ми хочемо намалювати сітку з 10 тисячами елементів за допомогою простого колір-шейдера. Сітка складається з чотирикутників і трикутників. Є лише 35 тисяч вершин і 15 тисяч полігонів для обробки, що є досить легким завданням для сучасних GPU.

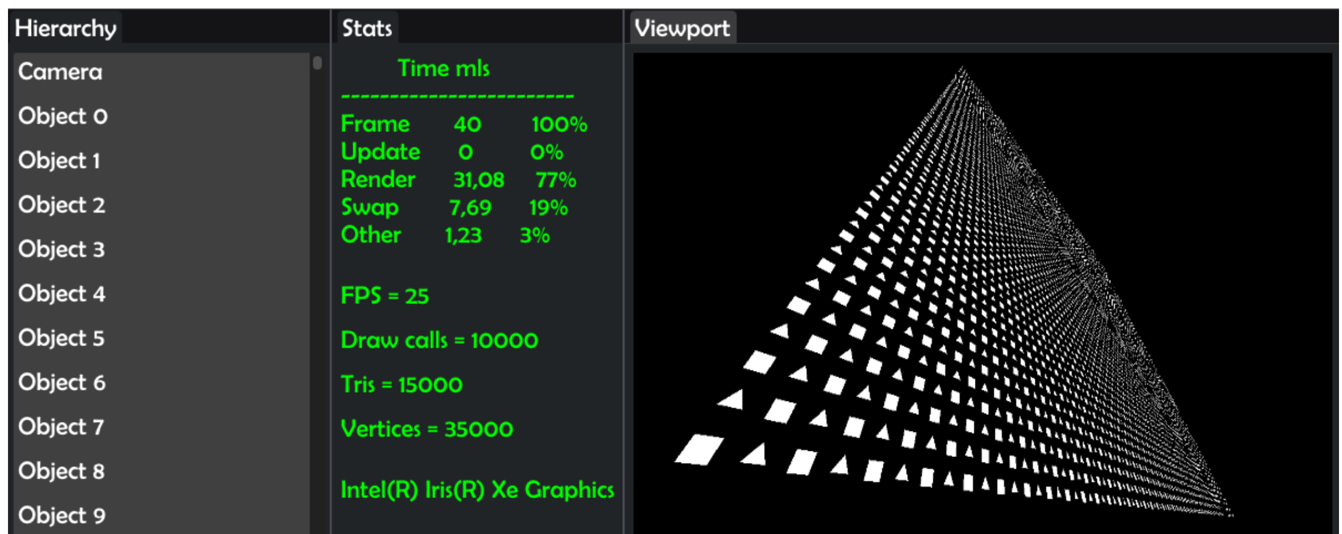


Рис 1.1 – 10 тисяч елементів з окремим DC при 25 FPS

Проте, як ми можемо бачити, приклад працює з 25 FPS, а рендер забирає 75% загального часу кадру. Чому? Так як ми малюємо кожен елемент окремо, ми

отримуємо 10 тисяч DC кожен кадр, що і є головним вузьким місцем в цьому прикладі.

Щоб покращити ефективність програми, ми можемо використати статичний батчинг, щоб об’єднати квадрати і трикутники у один великий меш VBO, а потім намалювати їх за один DC.

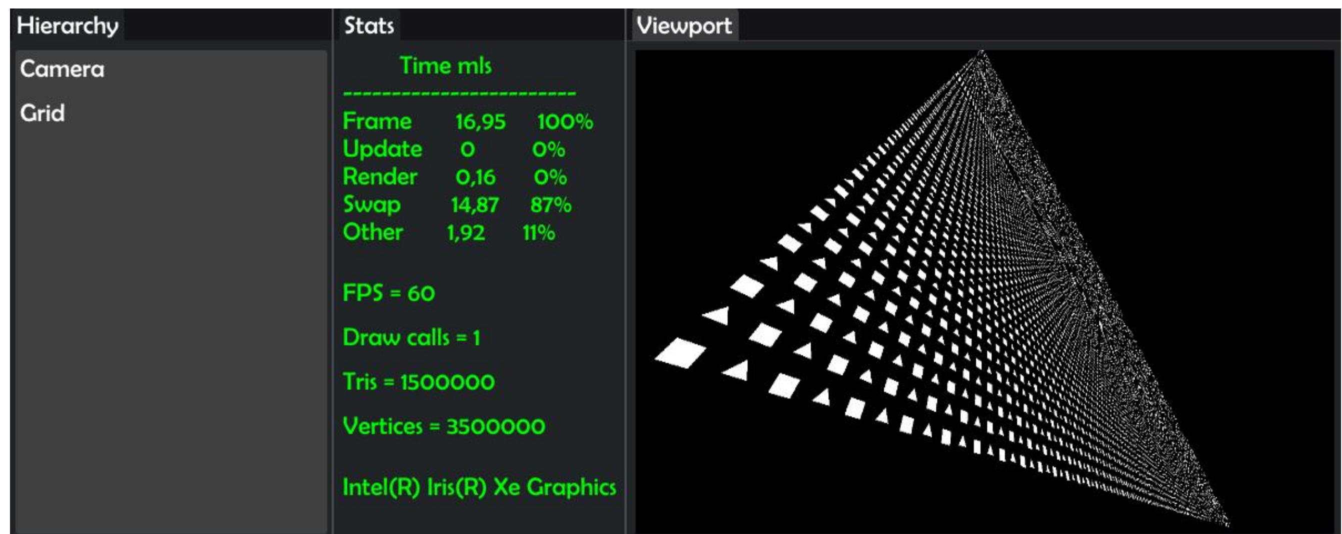


Рис 1.2 – 1 мільйон елементів з 1 DC при 60 FPS

Після того, як ми побачили результат батчингу, давайте реалізуємо свій простий статичний батчинг на прикладі схожої сітки. Для початку, нам необхідна функція, яка будує світку квадратів.

```
public Mesh Build()
{
    Mesh result = new();

    for (int i = 0; i < _count; ++i)
    {
        for (int j = 0; j < _count; ++j)
        {
            Vector2 position = GetPositionForQuad(i, j);

            AddQuad(result, position);
        }
    }

    return result;
}

private void AddQuad(Mesh mesh, Vector2 position)
{
    mesh.Indices.AddRange(new[]
```

```
    {
        0 + _indexOffset, 1 + _indexOffset, 2 + _indexOffset,
        2 + _indexOffset, 3 + _indexOffset, 0 + _indexOffset,
    });

    mesh.Vertices.AddRange(new[]
    {
        position.X + _objectSize,   position.Y,
        position.X + _objectSize,   position.Y + _objectSize,
        position.X,    position.Y + _objectSize,
        position.X,    position.Y,
    });

    _indexOffset += 4;
}
```

Ось ми отримали 64 (або скільки ви забажаєте) квадратів, які малюються за один раз.

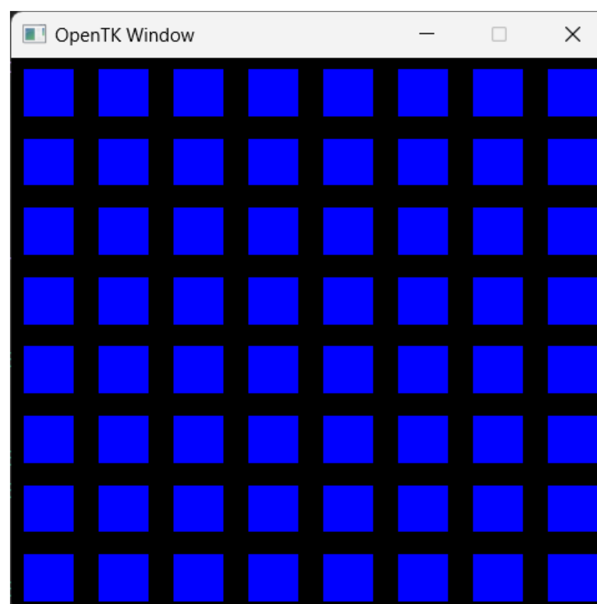


Рис 1.3 – Сітка з однакових елементів з 1 DC

Чудово, проте головна перевага статичного батчингу це поєднання **різних** мешей. Тому, додамо буквально декілька рядочків коду і вже отримаємо більш цікавий результат.

```
public Mesh Build()
{
    Mesh result = new();

    for (int i = 0; i < _count; ++i)
    {
        for (int j = 0; j < _count; ++j)
        {
            Vector2 position = GetPositionForObject(i, j);
```

```
                if ((i + j) % 2 == 0)
                {
                    AddQuad(result, position);
                }
                else
                {
                    AddTriangle(result, position);
                }
            }
        }

        return result;
    }

    private void AddTriangle(Mesh mesh, Vector2 position)
    {
        mesh.Indices.AddRange(new[]
        {
            0 + _indexOffset, 1 + _indexOffset, 2 + _indexOffset,
        });

        mesh.Vertices.AddRange(new[]
        {
            position.X,   position.Y,
            position.X + _objectSize,   position.Y,
            position.X + _objectSize / 2f,   position.Y + _objectSize
        });

        _indexOffset += 3;
    }
```

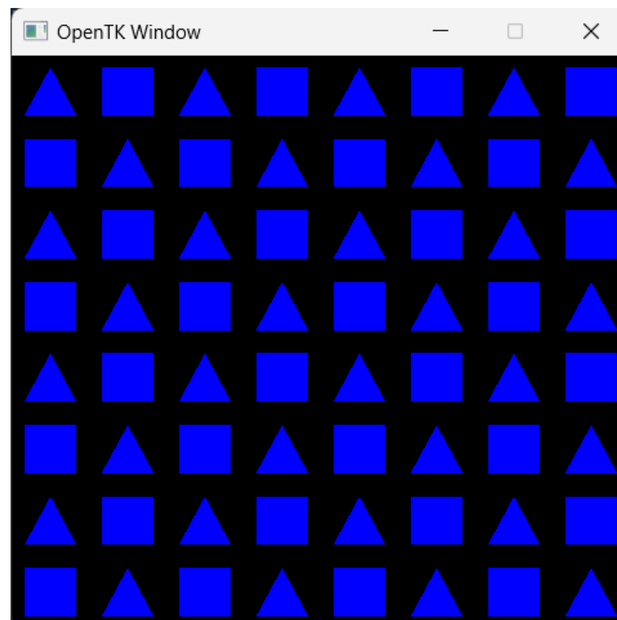


Рис 1.4 – Сітка з різних елементів з 1 DC

Чудово. Проте, що ми можемо зробити, якщо ми хочемо, щоб наші елементи виглядали по-різному?

Відповідь криється в установці ID для кожної вершини трикутнику чи квадрату. І, використовуючи ID, змінювати вигляд у фрагментому шейдері. Звучить трохи складнувато, проте насправді це зовсім не так.

Розглянемо допоміжний приклад: намалюємо 3 неоднакові фігури з різними кольорами за один DC.

```
Mesh mesh = new()
{
    Vertices =
    {
         // position      id
        -0.95f, -0.25f,   0,
        -0.45f, -0.25f,   0,
        -0.7f,   0.25f,   0,

        0.25f,   0.25f,   1,
        0.25f,  -0.25f,   1,
        -0.25f, -0.25f,   1,
        -0.25f,  0.25f,   1,

        0.45f,   0.25f,   2,
        0.95f,   0.25f,   2,
        0.7f,   -0.25f,   2,
    },

    Indices =
    {
        0, 1, 2,
        3, 4, 6,
        4, 5, 6,
        7, 8, 9
    }
};
```

Після установки ID для кожної вершини геометрії, тепер ми можемо прочитати його у шейдері і на його основі змінювати поведінку рендерингу простим if / else.

```
// Vertex shader (vert.glsl)
#version 330 core
layout(location = 0) in vec2 position;
layout(location = 1) in float vertexId;

out float id;

void main(void)
{
    id = vertexId;
    gl_Position = vec4(position, 0.0, 1.0);
}


// Fragment shader (frag.glsl)
#version 330 core
```

```
in float id;

out vec4 outputColor;

void main()
{
    vec4 result = vec4(1, 1, 1, 1);

    if (id == 0)
    {
        result = vec4(1, 0, 0, 1);
    }

    else if (id == 1)
    {
        result = vec4(0, 1, 0, 1);
    }

    else if (id == 2)
    {
        result = vec4(0, 0, 1, 1);
    }

    outputColor = result;
}
```

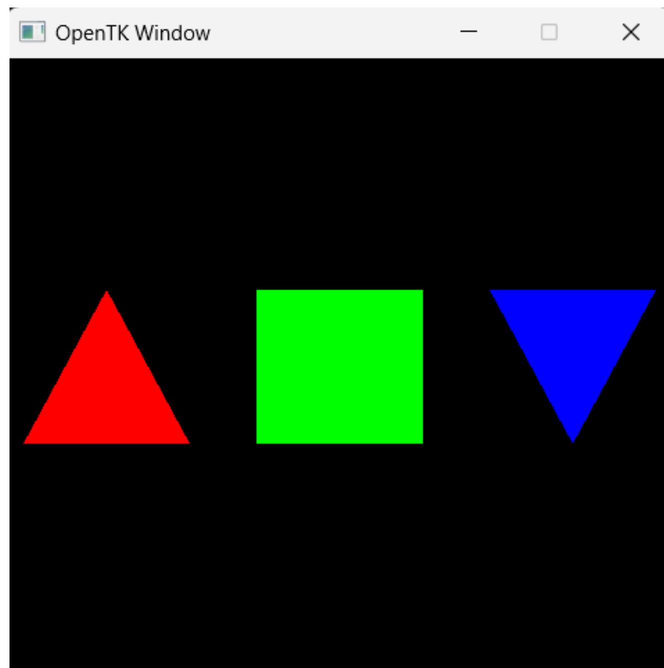


Рис 1.5 – Різні елементи, які виглядають по-різному з 1 DC

Звісно, є декілька інших способів досягнути подібного результату, проте ID це більш загальний підхід.

У фрагментому шейдері ми використовували if / else. Для такого прикладу це не є погано, проте ми легко можемо замінити це масивом кольорів, який будемо передавати з хост-мови (C#).

```
#version 330 core
uniform vec4[30] colors; // 30 is like Capacity in List. It is NOT Count
uniform int colorsCount; // real Count

in float id;

out vec4 outputColor;

void main()
{
    outputColor = colors[id % colorsCount];
}
```

Передача кольорів у шейдер з C#.

```
int program = ShaderProgram.Create("vert.glsl", "frag.glsl");

int countIndex = GL.GetUniformLocation(program, "colorsCount");
int colorsIndex = GL.GetUniformLocation(program, "colors");

GL.UseProgram(program);
GL.Uniform1(countIndex, 6);
GL.Uniform4(colorsIndex, 6, new float[]
{
  255, 255, 0, 255, // yellow
  0, 255, 255, 255, // cyan
  255, 0, 255, 255, // purple
  255, 0, 0, 255, // red
  0, 255, 0, 255, // green
  0, 0, 255, 255, // blue
});
```

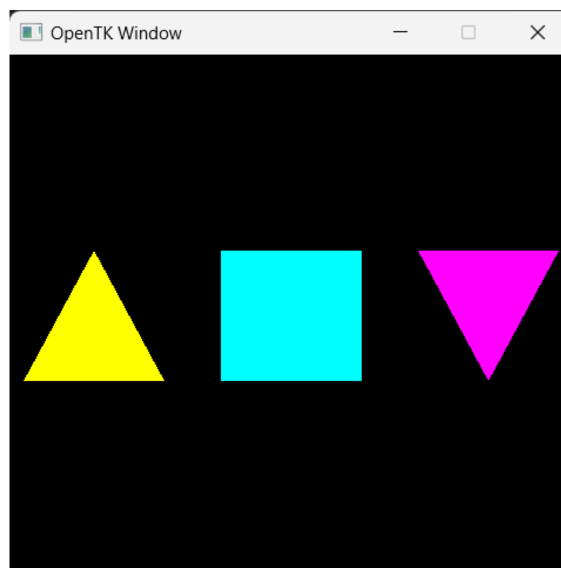


Рис 1.6 – Контроль над виглядом елементів з хост-мови

Чистіший код шейдеру та більший контроль над ним з хост-мови: вбили 2 зайців одним пострілом.

Повернемось до прикладу з сіткою. Додамо до її генерації ID та подивимось на результат.

```
// Quad
mesh.Vertices.AddRange(new[]
{
    position.X + _objectSize,   position.Y, _id,
    position.X + _objectSize,   position.Y + _objectSize, _id,
    position.X,   position.Y + _objectSize, _id,
    position.X,   position.Y, _id,
});

// Triangle
mesh.Vertices.AddRange(new[]
{
    position.X,   position.Y, _id,
    position.X + _objectSize,   position.Y, _id,
    position.X + _objectSize / 2f,   position.Y + _objectSize, _id
});
```

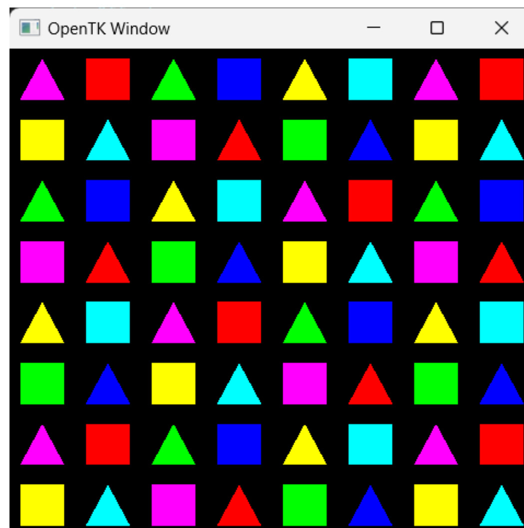


Рис 1.7 – Сітка з різних елементів, які виглядають по-різному з 1 DC

Як я казав раніше, ID це більш загальний підхід для зміни вигляду геометрії, коли використовується батчинг.

Використовуючи цей підхід, ми можемо встановлювати текстуру з масиву текстур або використовувати якісь кастомні рендер-функції.

1.1 СТАТИЧНИЙ БАТЧИНГ З МОДЕЛЬНИМИ МАТРИЦЯМИ

З цим підходом, ми можемо піти навіть далі і зробити так, щоб наша геометрія рухалась, нівелюючи основний недолік батчингу: неможливість керувати

виглядом і положенням окремо взятого елементу. Для вирішення цієї задачі, нам необхідно трохи змінити генерацію мешу сітки.

Раніше, будуючи наш меш (VBO), ми встановлювали позиції вершин вручну, і, відповідно, малювали великий меш за один DC, використовуючи різні позиції вершин.

Розглянемо допоміжний приклад: намалюємо 4 неоднакові фігури з різними кольорами за один DC. Давайте на глянемо на VBO цієї фігури.

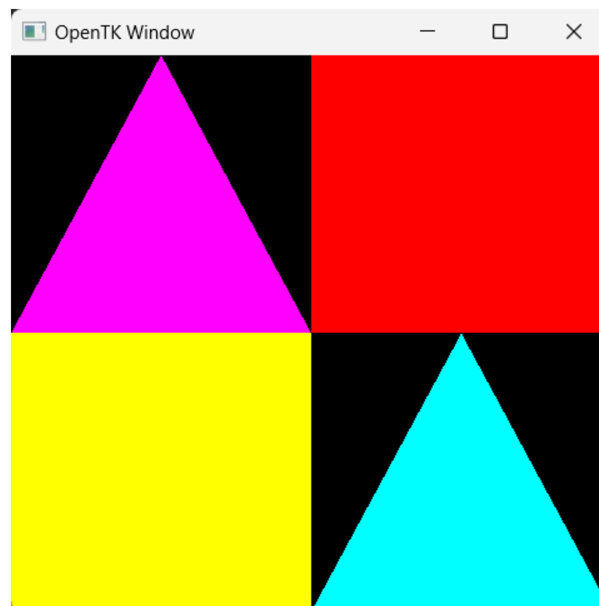


Рис 1.1.1 – Статичний батчинг з «захардкоженими» вершинами у буфері

```
float[] vbo =
{
 // position   id
     0,    -1,    0,
     0,     0,    0,
    -1,     0,    0,
    -1,    -1,    0,

    0,     -1,    1,
    1,     -1,    1,
    0.5f,   0,    1,

    -1,     0,    2,
    0,      0,    2,
    -0.5f,  1,    2,

    1,      0,    3,
    1,      1,    3,
    0,      1,    3,
    0,      0,    3,
```

```
};
```

І це працює чудово. Проте, якщо ми хочемо, щоб наша геометрія рухалась, нам потрібно відмовитись від хардкоду позицій вершин і керувати ними ззовні.

Для цього необхідно вирівняти нашу геометрію і передавати модельні матриці у вершинний шейдер.

```
float[] vbo =
{
   // position       id
    0.5f,    0.5f,    0,
    0.5f,   -0.5f,    0,
    -0.5f,  -0.5f,    0,
    -0.5f,   0.5f,    0,

    0.5f,   -0.5f,    1,
    -0.5f,  -0.5f,    1,
    0,       0.5f,    1,

    0.5f,   -0.5f,    2,
    -0.5f,  -0.5f,    2,
    0,       0.5f,    2,

    0.5f,    0.5f,    3,
    0.5f,   -0.5f,    3,
    -0.5f,  -0.5f,    3,
    -0.5f,   0.5f,    3,
};
```

Тепер наша геометрія відцентрована.

```
#version 330 core
uniform mat4[200] modelMatrices;  // Ті самі керуючі матриці, які нам потрібно
передати ззовні з хост-мови.

layout(location = 0) in vec2 position;
layout(location = 1) in float vertexId;

out float id;

void main(void)
{
    id = vertexId;
    gl_Position = vec4(position, 0.0, 1.0) * modelMatrices[int(id)];
}
```

Вершинний шейдер з масивом модельних матриць

```
private void SendDataToShader()
{
    for (int i = 0; i < _transforms.Length; ++i)
    {
        for (int j = 0; j < 4; ++j)
        {
            for (int k = 0; k < 4; ++k)
```

```
            {
                Matrix4 modelMatrix = _transforms[i].ModelMatrix;
                _rawModelMatrices[i * 16 + j * 4 + k] = modelMatrix[j, k];
            }
        }
    }

    int location = GL.GetUniformLocation(_shaderProgramId, "modelMatrices");
    GL.ProgramUniformMatrix4(_shaderProgramId, location, _rawModelMatrices.Length / 
16, true, _rawModelMatrices);
}
```

Звісно, якщо передати одиничні (необроблені) матриці ми отримаємо не те, що очікуємо.

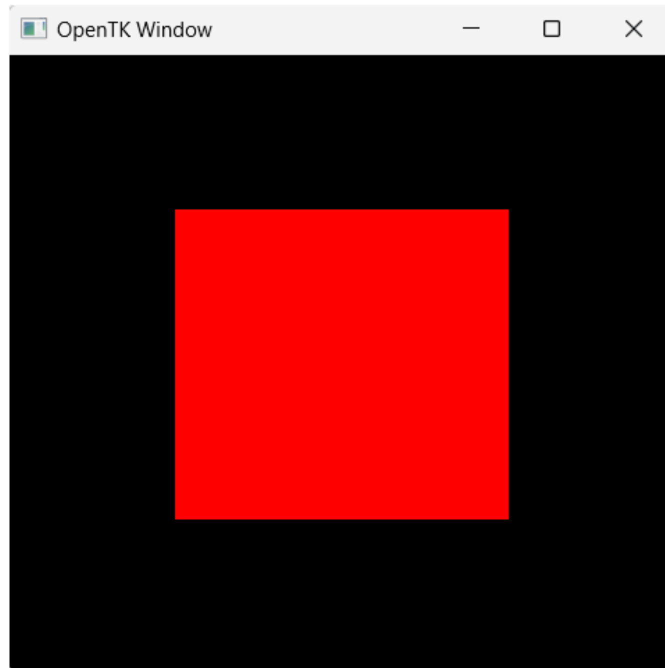


Рис 1.1.2 – Статичний батчинг з відцентрованими елементами

Це відбулось тому, що квадрат перекрив всі інші елементи, так як всі вони відцентровані. Щоб це вирішити, необхідно всього лише додати зсув у матриці.

```
modelMatrices[i * count + j] = new Transform()
{
    Position = new Vector3()
    {
        X = -1 + j * (objectSize * 2 + offset) + objectSize + offset / 2,
        Y = -1 + i * (objectSize * 2 + offset) + objectSize + offset / 2,
        Z = 0,
    }
};
```

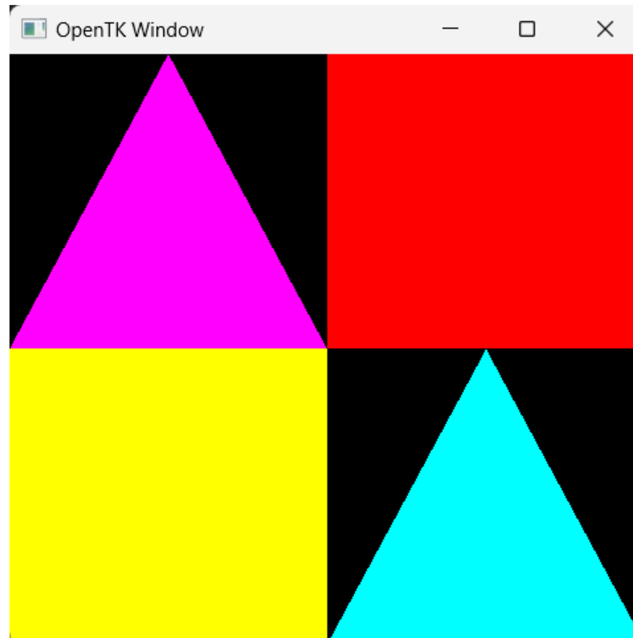


Рис 1.1.3 – Статичний батчинг з контролем трансформації елементів з хост-мови

Чудово, ми це виправили. Причина по якій ми робили ці маніпуляції полягає в тому, що тепер ми можемо звертатись до трансформації окремо взятого елементу так, наче він малюється окремо.

Щоб це продемонструвати, давайте додаймо невеличку анімацію до нашої сітки.

```
private void UpdateMatrices(float deltaTime)
{
    _time += deltaTime;
    float lerp = Algorithms.ScaledSin(_time * 2);

    for (int i = 0; i < _transforms.Length; ++i)
    {
        Vector3 eulerAngles = Vector3.Lerp(Vector3.Zero, Vector3.UnitZ * 
_animationData[i].TargetAngle, lerp);

        _transforms[i].Rotation = Quaternion.FromEulerAngles(eulerAngles);
        _transforms[i].Position = Vector3.Lerp(_animationData[i].StartPosition, 
_animationData[i].TargetPosition, lerp);
        _transforms[i].Scale = Algorithms.Lerp(1, _animationData[i].TargetScale, 
lerp) * Vector3.One;
    }
}
```

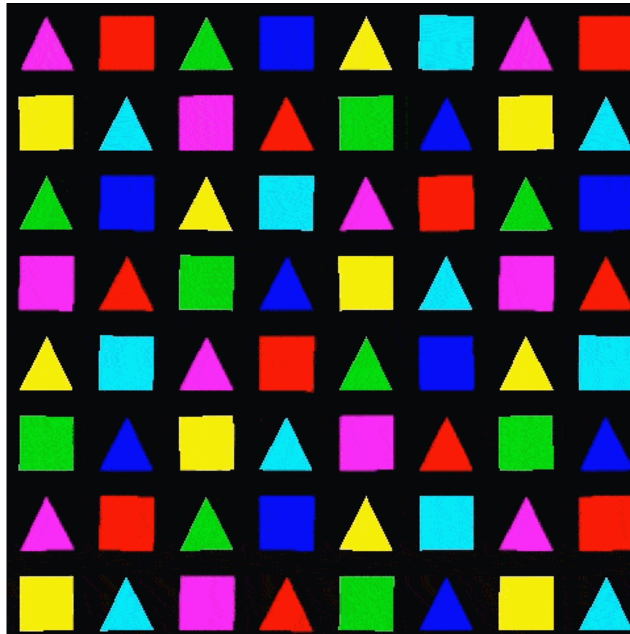


Рис 1.1.4 – Анімована сітка з різними елементами з 1 DC

Статичний батчинг був застосований у першій лабораторній роботі з «Інтелектуальних систем» для малювання карти гри «Пакман» за один раз.

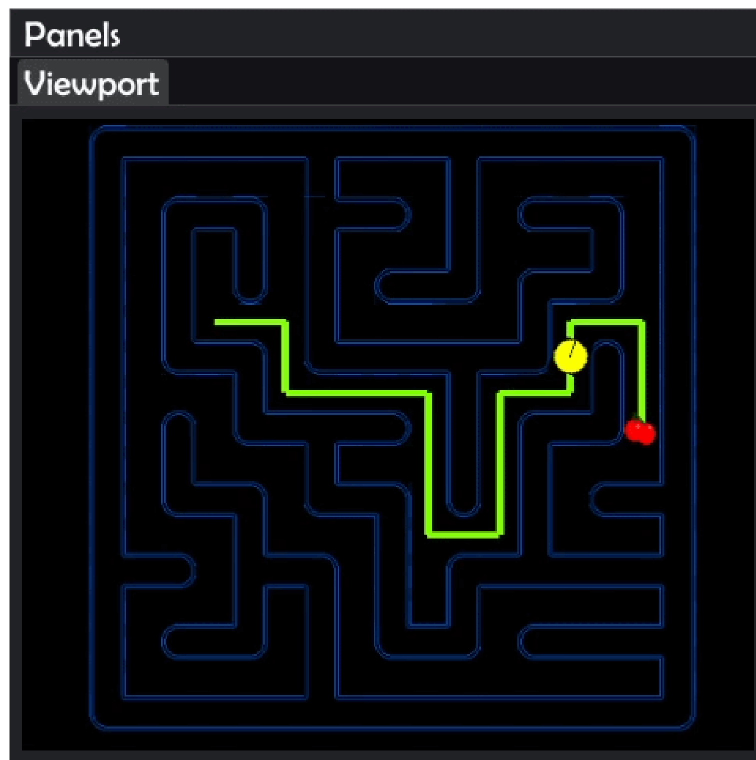


Рис 1.8 – Приклад роботи лабораторної роботи «Пакман» з «Інтелектуальних систем»

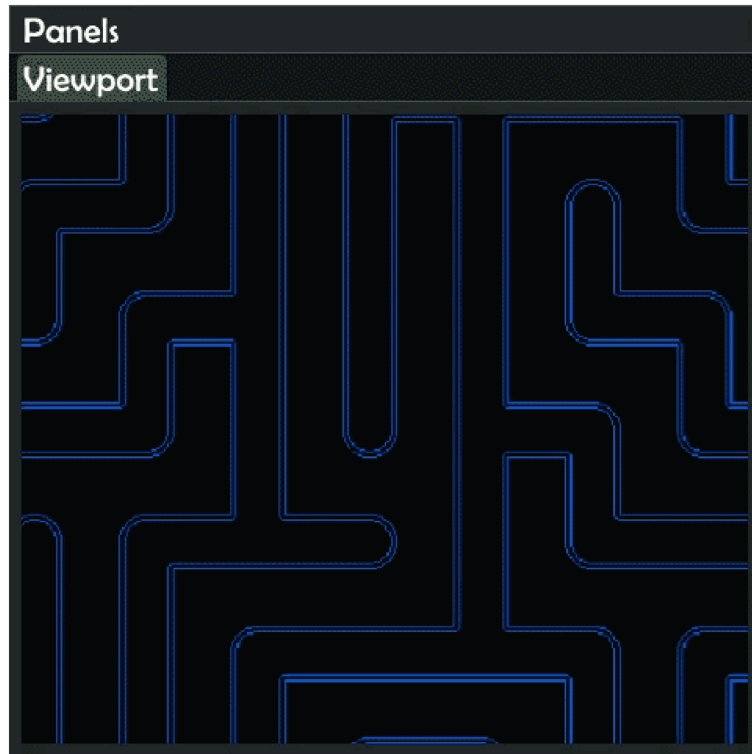


Рис 1.9 – Приклад використання статичного батчингу для малювання карти

Карта 500x500 генерується алгоритмом RecursiveBacktracker. 1 мільйон полігонів при 90 FPS.

```
private Mesh GetMazeMesh()
{
    Mesh[] tiles = new Mesh[_obstacles.Count];

    int index = 0;
    foreach (Vector2i obstacle in _obstacles)
    {
        tiles[index] = _obstacleSpawner.Create(obstacle, GetNeighbours(obstacle));
        ++index;
    }

    return StaticBatching.Concatenate(tiles);
}
```

```
public static class StaticBatching
{
    public static Mesh Concatenate(IList<Mesh> objects)
    {
        return new Mesh(ConcatenateAttributes(objects, 0, objects.Count))
        {
            Indices = ConcatenateIndices(objects, 0, objects.Count),
        };
    }
```

```
    private static List<VertexAttribute> ConcatenateAttributes(IList<Mesh> objects, 
int startIndex, int endIndex)
    {
        List<VertexAttribute> result = new();
        int verticesCount = objects.Sum(startIndex, endIndex, mesh => 
mesh.VerticesCount);

        foreach (KeyValuePair<string, VertexAttribute> attribute in 
objects.First().Attributes)
        {
            float[] buffer = new float[verticesCount * attribute.Value.Size];
            string attributeName = attribute.Key;
            int destinationIndex = 0;

            for (int i = startIndex; i < endIndex; ++i)
            {
                float[] subData = objects[i].Attributes[attributeName].Data;
                Array.Copy(subData, 0, buffer, destinationIndex, subData.Length);
                destinationIndex += subData.Length;
            }

            result.Add(new VertexAttribute(attributeName, attribute.Value.Index, 
attribute.Value.Size, buffer));
        }

        return result;
    }

    private static uint[] ConcatenateIndices(IList<Mesh> objects, int startIndex, 
int endIndex)
    {
        int count = objects.Sum(startIndex, endIndex, meshData => 
meshData.Indices.Length);
        uint[] buffer = new uint[count];
        int index = 0;
        uint offset = 0;

        for (int i = startIndex; i < endIndex; ++i)
        {
            for (int j = 0; j < objects[i].Indices.Length; ++j, ++index)
            {
                buffer[index] = objects[i].Indices[j] + offset;
            }

            offset += (uint)objects[i].VerticesCount;
        }

        return buffer;
    }
}
```

Static batching клас з рушія Dengine.

2 ДИНАМІЧНИЙ БАТЧИНГ (DYNAMIC BATCHING)

Динамічний батчинг – це оптимізаційна техніка в комп’ютерній графіці, яка використовується для зменшення DC.

Статичний та динамічний батчинг обидва використовують об’єднання вершинних буферів в один, який потім малюється за один раз.

Слід зазначити, що динамічний батчинг зазвичай об’єднує меші під час виконання гри, тоді як статичний це робить в редакторі рушія, а потім зберігає отриманий VBO у файлах гри.

Ми вже бачили в попередній секції, що статичний батчинг це техніка, яка об’єднує зазвичай статичні об’єкти у єдиний вершинний буфер, проте вона також може бути застосована з динамічними об’єктами, якщо передавати модельні матриці в шейдер.

Принципова різниця між статичним та динамічним батчингом в тому, що перший не обновлює отриманий VBO, тоді як динамічний обновлює.

Звучить трішки заплутанно, проте на прикладі все стане зрозуміло.

Припустимо, ми хочемо розробити наш власний UI-фреймворк на кшталт ImGui для нашого ігрового рушія.

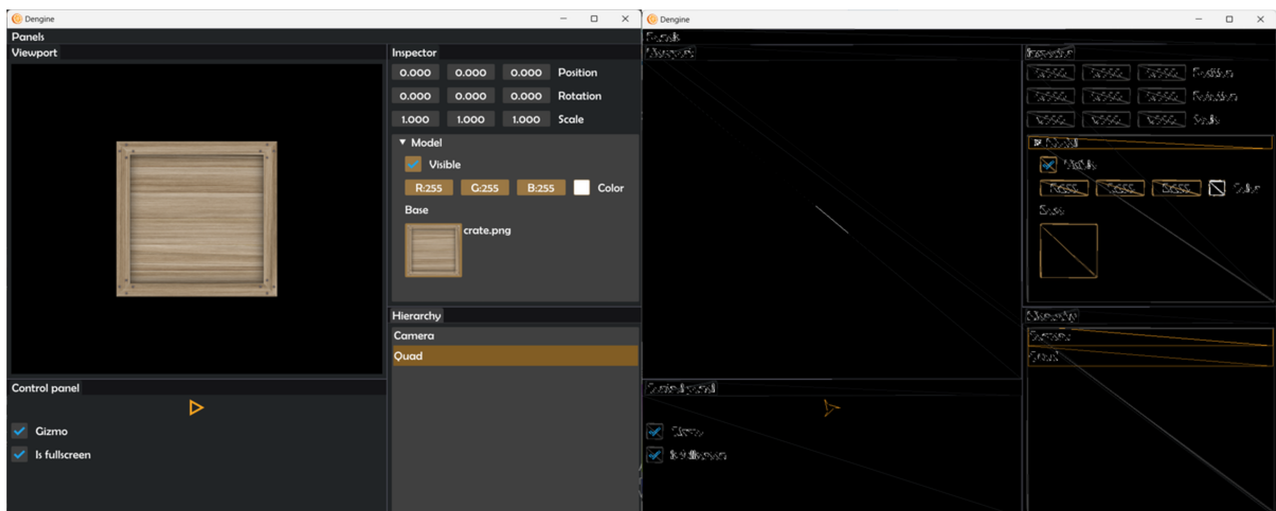


Рис 2.1 – Демонсрація, що інтерфейс малюється таким ще чином, як і вся інша геометрія

Як ви можете бачити, елементи користувацього інтерфейсу є нічим іншим, як квадратами, які малюються точно таким же чином, як і вся інші геометрія на сцені.

Також ви можете помітити, що у ААА іграх, як The Witcher 3 інтерфейс може бути досить комплексним.

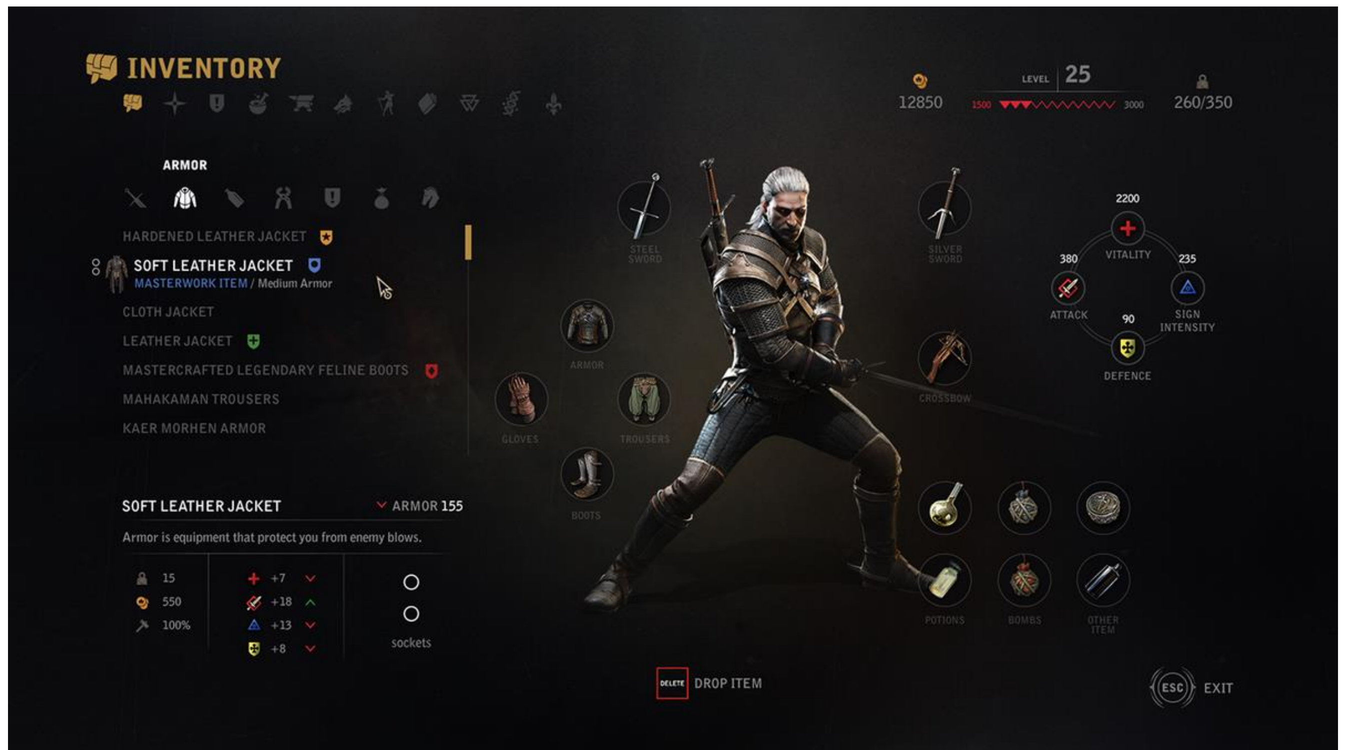


Рис 2.2 – Приклад складноного інтерфейсу

Це означає, якщо ми не будемо батчити інтерфейс, щоб відмалювати його за один раз, це може негативно вплинути на загальну продуктивність гри.

Ви можете запитати, чому просто не використати статичний батчинг з модельними матрицями: кожен елемент інтерфейсу отримає власну модельну матрицю, всі анімації будуть програватись, і весь UI відмалюється за один DC. Ляпота. Це спрацює, проте в цьому випадку вам доведеться тримати весь UI в пам’яті.

Більше того, більша частина тієї пам’яті буде витрачена даремно, так як лише частина інтерфейсу є видимою в будь-який момент часу.

З іншого боку, **динамічний батчинг збирається на ЦП, що означає, що ми маємо повний контроль над тим, що батчити, а що ні**, використовуючи хост-мову.

Для прикладу, подивіться на список броні на скриншоті зверху. Використовуючи динамічний батчинг, ми можемо відмалювати лише видимі предмети з того списку. Це було б неможливо зі статичним батчингом, так як він вимагає збереження всього списку в пам’яті, незалежно від того, що є видимим.

Гаразд, давайте реалізуємо власний динамічний батчинг. Припустимо, у нас є панель з анімованими кнопками, які зникають, коли ми на них клікаємо.

```
public class DynamicBatch
{
    private readonly Mesh _overallMesh = new();
    private readonly RenderingObject _overallView;

    public DynamicBatch(string vertShaderPath, string fragShaderPath)
    {
        _overallView = new RenderingObject(vertShaderPath, fragShaderPath);
    }

    public void Clear()
    {
        _overallMesh.Indices.Clear();
        _overallMesh.Vertices.Clear();
        _overallMesh.Attributes.Clear();
    }

    public void Add(Mesh mesh, Transform transform)
    {
        TryAddAttributes(mesh.Attributes);
        AddIndices(mesh.Indices);
        AddVertices(mesh.Vertices, transform);
    }

    public void Draw()
    {
        _overallView.BufferData(_overallMesh, BufferUsageHint.DynamicDraw);
        _overallView.Draw();
    }
}
```

Клас динамічного батчингу має просту та зрозумілу API:

- Clear
- Add(Mesh, Transform)

- Draw

Ви можете запитати, навіщо нам потрібен об’єкт “Transform”? Це тому, що нам потрібно отримати світову позицію вершин, щоб додати її до загального мешу “_overallMesh”. Ми можемо досягнути цього множенням локальної позиції

```
// AddVertices stage
Matrix4 modelMatrix = transform.ModelMatrix;

for (int i = 0; i < vertices.Count; i += stride)
{
    Vector4 position = new(vertices[i], vertices[i + 1], 0, 1);
    Vector2 worldPosition = (position * modelMatrix).Xy;

    _overallMesh.Vertices[verticesOverall + i + 0] = worldPosition.X;
    _overallMesh.Vertices[verticesOverall + i + 1] = worldPosition.Y;
}
```

Використання:

```
public void Render(FrameEventArgs args)
{
    _dynamicBatch.Clear();

    foreach (Button button in _buttons)
    {
        if (button.IsVisible)
        {
            _dynamicBatch.Add(button.Mesh, button.Transform);
        }
    }

    _dynamicBatch.Draw();
}
```

І це все! Не так важко, як здавалось. Зараз ми малюємо лише видиму частину всього інтерфейсу за один DC.

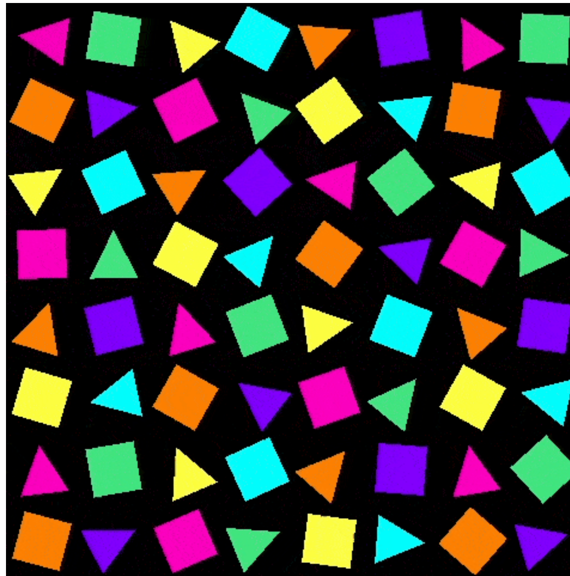


Рис 2.3 – Недоліки та обмеження

Приклад з інтерфейсом був вибраний невипадково. Кожен елемент є, по суті, звичайним квадратом, тому їх компонування на центральному процесорі не є проблемою.

Зазвичай динамічний батчинг ефективний для обробки невеликих об'єктів, оскільки об'єднання їх в один батч може стати проблематичним, якщо об'єкти досить великі або їх кількість значна.й

Тому, якщо ви спробуєте динамічно об'єднувати велику кількість високополігональних моделей, ви б точно отримали просадки продуктивності у вашій грі / програмі

Тому, динамічний батчинг досить часто використовується для розробки системи частинок (particle system). Також зазвичай він використовується для об'єктів, які часто створюються і знищуються.

Загалом, динамічний батчинг найбільш ефективний у тих випадках, коли витрати на оновлювання вершинного буфера і властивостей матеріалу менші за витрати малювання кожного елементу окремо.

Давайте розглянемо, як такий популярний рушій, як Unity використовує динамічний батчинг.

Limitations

In the following scenarios, Unity either can't use dynamic batching at all or can only apply dynamic batching to a limited extent:

- Unity can't apply dynamic batching to meshes that contain more than 900 vertex attributes and 225 vertices. This is because dynamic batching for meshes has an overhead per vertex. For example, if your **shader** uses vertex position, vertex normal, and a single UV, then Unity can batch up to 225 vertices. However, if your shader uses vertex position, vertex normal, UV0, UV1, and vertex tangent, then Unity can only batch 180 vertices.
- If GameObjects use different material instances, Unity can't batch them together, even if they are essentially the same. The only exception to this is shadow caster rendering.
- GameObjects with lightmaps have additional renderer parameters. This means that, if you want to batch lightmapped GameObjects, they must point to the same **lightmap** location.
- Unity can't fully apply dynamic batching to GameObjects that use multi-pass shaders.
  - Almost all Unity shaders support several lights in **forward rendering**. To achieve this, they process an additional render pass for each light. Unity only batches the first render pass. It can't batch the draw calls for the additional per-pixel lights.
  - The Legacy Deferred rendering path doesn't support dynamic batching because it draws GameObjects in two render passes. The first pass is a light pre-pass and the second pass renders the GameObjects.

Рис 2.4 – Головні умови для динамічного батчингу

- Однаковий матеріал (це досить очевидно, всі техніки батчингу вимагають це)
- Розмір мешу до компонування обмежений. Кількість вершин * Розмір вершини < певний ліміт.

По замовчуванню Unity використовує динамічний батчинг для її системи частинок, так як ця техніка сумісна з всіма графічними API.

Також Unity пропонує GPU instancing для рендерингу частинок, що навіть покращує ефективність, проте ця технологія не сумісна з всіма API, по типу OpenGL ES 2.0.

3 ДУБЛЮВАННЯ ГЕОМЕТРІЇ (GPU INSTANCING)

GPU Instancing - техніка, що використовується у графічних програмах і іграх для оптимізації відображення великої кількості ідентичних або схожих об'єктів.

Традиційно, при відображенні багатьох однакових об'єктів, таких як дерева, трава, каміння або кущі, кожен окремий об'єкт передається GPU окремо, що вимагає великих обчислювальних ресурсів.

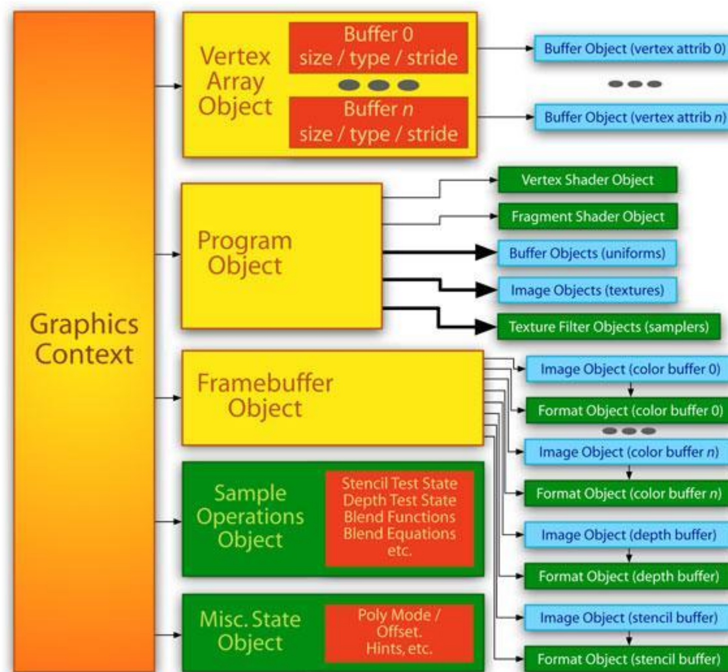


Рис 3.1 – Схема конвеєру OpenGL

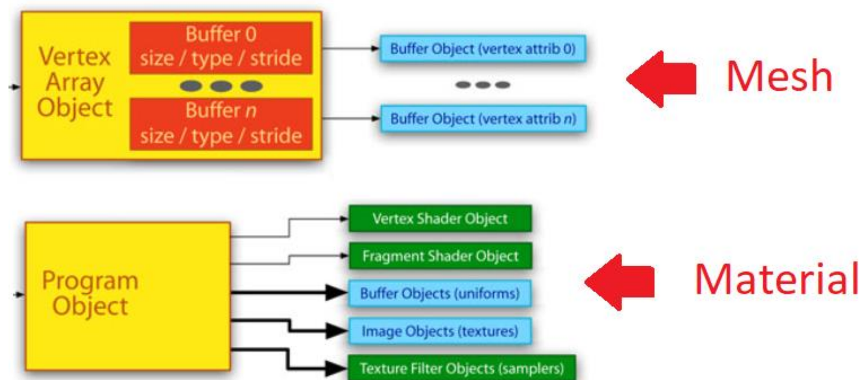


Рис 3.2 – Абстрації у вигляді понять «Mesh» та «Material»

Іншими словами, в сучасних графічних API, по типу, OpenGL4/Metal/Vulcan пам’ять під меш та матеріал виділяється та заповнюється одразу на GPU. В процесі виконання програми, клієнту лише необхідно викликати функцію Draw, щоб намалювати об’єкт, так як API вже знає, де лежать необхідні їй для малювання дані. Такий підхід суттєво покращує продуктивність рендерингу графіки.

Однак, якщо об’єктів (особливо однакових) на сцені стає досить багато, це призводить до проблеми: велика кількість DC та великий дублікат даних на GPU. Звісно, можна закешувати матеріал, вершинний буфер та використати батчинг техніки, які були розглянуті вище (статичний/динамічний), проте в решті-решт вийде досить громіздке рішення.

Тому на світ у 2010 році у OpenGL 3.3 з’явилась нова батчинг-техніка GPU instancing.

Суть полягає в тому, що дані про окремі об'єкти, такі як їх положення, масштаб і параметри, зберігаються в спеціальному буфері на GPU. Потім, використовуючи цей буфер, GPU може швидко відображати всі копії об'єкта безпосередньо на екрані. Тому інша назва цієї техніки називається «Дублювання геометрії».

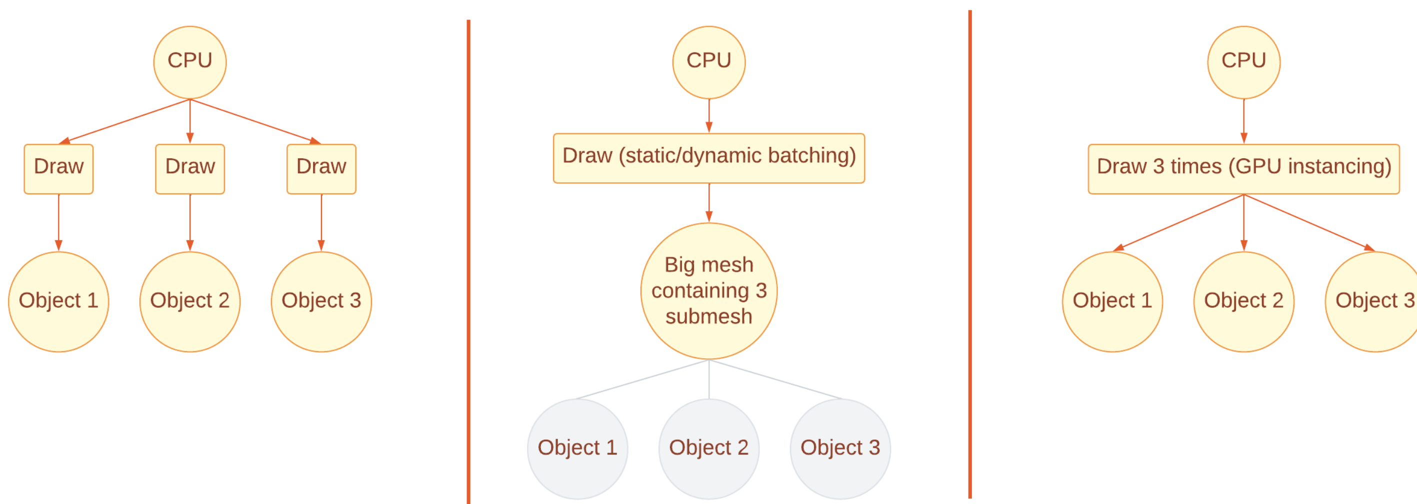


Рис 3.3 – Схема різних типів малювання геометрії.

Проте, клієнтське використання 1 і 3 підходів суттєво відрізняється. Нехай нам потрібно намалювати 3 астероїди.

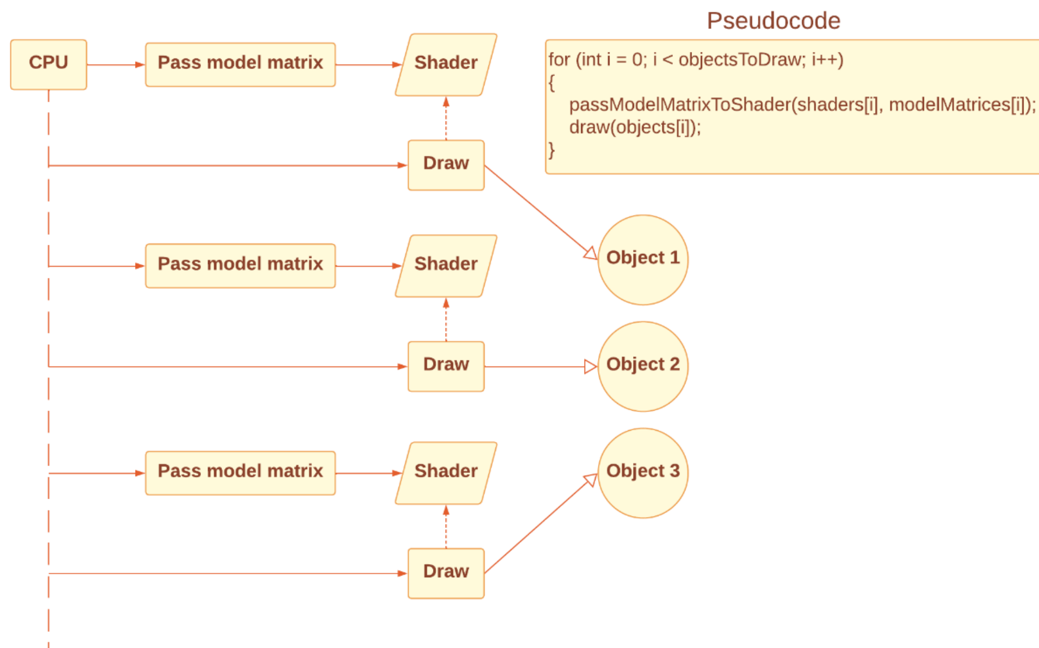


Рис 3.4 – Діаграма традиційного малювання 3 однакових об’єктів з псевдокодом

Малюючи астероїди «традиційно», ми маємо повний контроль над ними: передаємо потрібні їм дані для малювання напряму в шейдер (uniform values), такі як модельна матриця, потім малюємо – все просто.

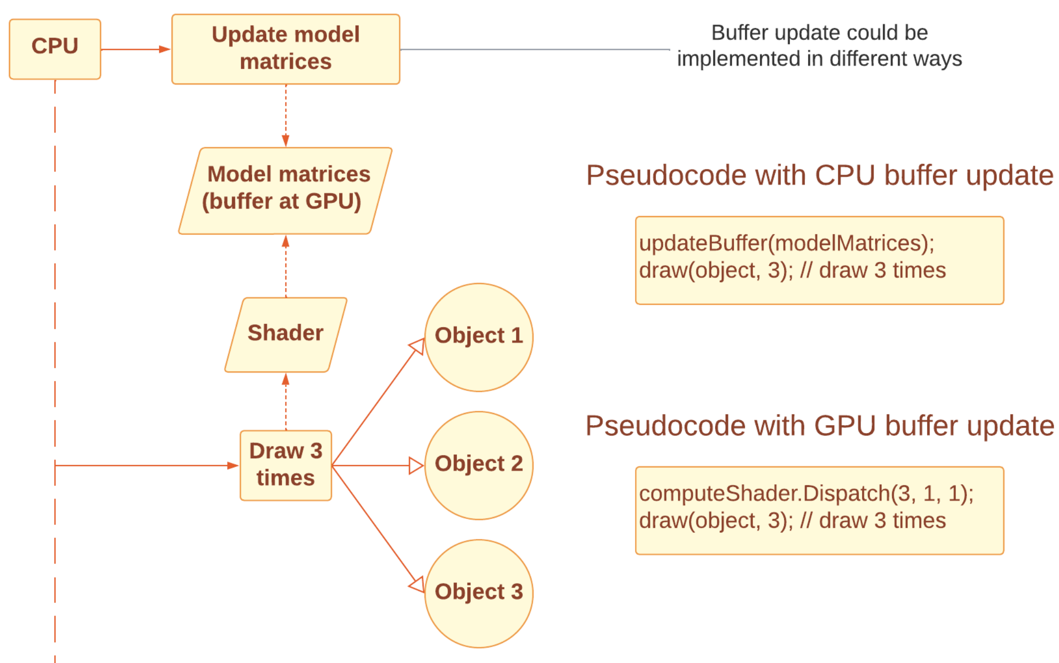


Рис 3.5 – Малювання з GPU instancing 3 однакових об’єктів з псевдокодом

Проте, використовуючи батчинг-техніки, ми не можемо звертатись до окремо взятого об’єкту і передати необхідні йому дані, тому потрібно створити буфер, який буде виступати швом між хост-мовою та шейдером.
І от вже цей буфер можна оновлювати звідки завгодно: з хост-мови, чи, наприклад, з шейдеру обчислення.

У OpenGL «Draw» командою буде `glDrawArrays`, тоді як «Draw 3 times» є `glDrawElementsInstanced`. Проте у сучаснішому графічному API Vulcan, техніка дублювання геометрії «вшита» по замовчуванню у всі команди малювання.

```
// Provided by VK_VERSION_1_0
void vkCmdDrawIndexed(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    indexCount,
    uint32_t                                    instanceCount,
    uint32_t                                    firstIndex,
    int32_t                                     vertexOffset,
    uint32_t                                    firstInstance);
```

Параметр instanceCount вказує, скільки геометрії має бути здубльовано.

Перевага GPU instancing очевидна: суттєве зменшення кількості DC та зайнятої пам’яті GPU.

Проте GPU instancing не є заміною традиційного батчингу, оскільки між ними є основна відмінність. Традиційний батчинг, дозволяє об'єднувати різні геометрічні об'єкти в один батч, тоді як GPU instancing дублює один і той самий об'єкт багато разів.

Нижче наведена демостраційна сцена «GPU instancing» з оновленням буферу за допомогою шейдеру обчислення.

```
#version 430 core
// Vertex shader

layout (binding = 1, std430) buffer ModelMatrices
{
    mat4[] modelMatrices;
};
```

```
void main(void)
{
    textureCoordinates = vertexTextureCoordinates;
    fragmentPosition = vec3(modelMatrices[gl_InstanceID] * vec4(vertexPosition,
1.0));
    normal = vertexNormal * mat3(inverse(modelMatrices[gl_InstanceID]));

    gl_Position = projectionMatrix * viewMatrix * modelMatrices[gl_InstanceID] *
vec4(vertexPosition, 1.0);
}
```

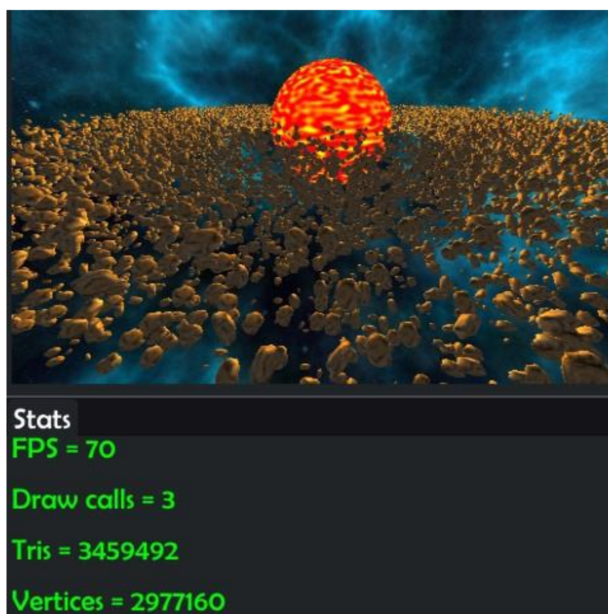


Рис 3.6 – 3.5 мільйонів, 18 тисяч динамічних астероїдів при 70 FPS.

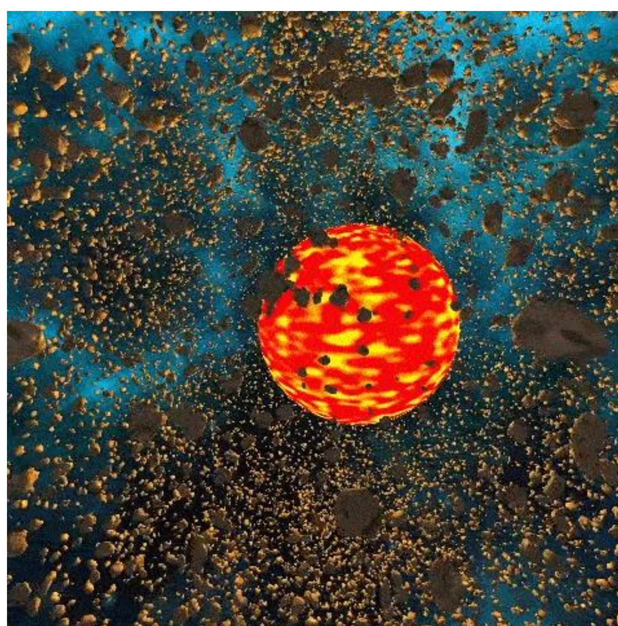


Рис 3.7 – Демонстрація сцени “GPU instancing”

4 ШЕЙДЕРИ ОБЧИСЛЕННЯ (COMPUTE SHADERS)

Хоча, шейдери обчислення не є частиною графічного конвеєру (Render Pipeline) і можуть використовуватись взагалі без візуалізації, проте вони досить часто використовуються для рендерингу графіки, тому розглянемо їх детальніше.

Що це?

- Шейдери обчислення - це невеликі програми, які виконуються на відеокарті та не є частиною графічного конвеєру.
- Так як виконання відбувається на GPU, шейдери обчислення мають одну потужну особливість: паралелізм.

Для чого?

- Графічне програмування: водна поверхня, вогонь, тіні, освітлення, тощо.
- Фізичне моделювання: динаміка рідини, поведінка тканин, симуляція фізики тіл, тощо.
- Обробка даних: обробка зображень, компресія даних, генерація та обробка великого обсягу геометричних даних.

Давайте реалізуємо нашу власну систему частинок: одну з найбільш показових застосувань шейдерів обчислення. Показова вона тому, що побудувати ефективну систему частинок практично неможливо без використання потужностей GPU.

Перед розробкою системи частинок, потрібно з’ясувати 2 речі:

- Як малювати
- Як оновлювати

Не є великим секретом, що система частинок має малюватись за один DC, так як навіть 10 тисяч частинок (10 тисяч DC кожен кадр) негативно вплинуть на загальну продуктивність програми.

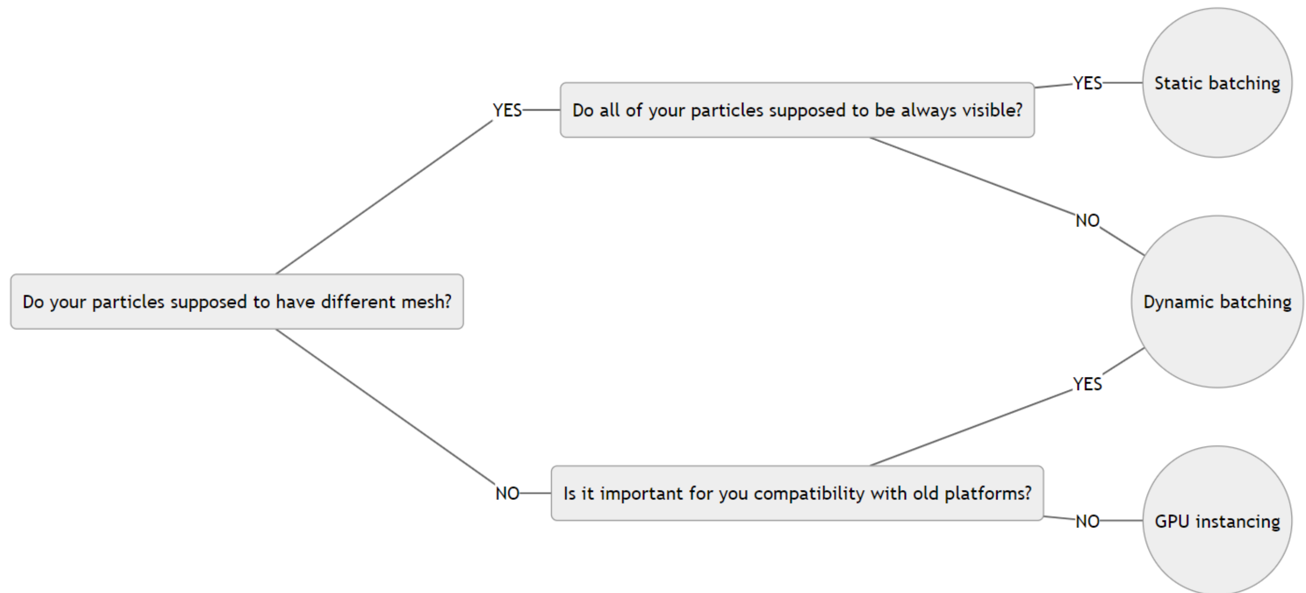


Рис 4.1 – Схема для вибору малювання частинок

Unity, для прикладу, по замовчуванню використовує Динамічний батчинг, але також пропонує GPU instancing для кращої продуктивності. Ми ж використаємо GPU instancing.

Добре, ми розібрались з першим питанням. Як бути з оновленням системи?

У нас є 3 шляхи:

- Синхронне виконання на CPU
- Багатопоточне виконання на CPU
- Виконання на відеокарті (шейдер обчислення)

Насправді, це тема дискусії, що і при яких умовах буде працювати швидше, проте коли кількість частинок досить велика, вам точно слід використовувати шейдер обчислення.

Добре. GPU instancing для рендерингу, шейдер обчислення для оновлення. Проте, так як шейдер оновлення не є частиною графічного конвеєру, ми маємо придумати, як зв’язати рендер і логіку.

Вихід є: SSBO. SSBO – це буфер, який виділяється прямо на GPU, призначений для зберігання великих об’ємів даних.

Слід зазначити, що SSBO працює лише з версії OpenGL 4.3.
Зваживши все сказане, ми отримали головний головний цикл виконання системи частинок:

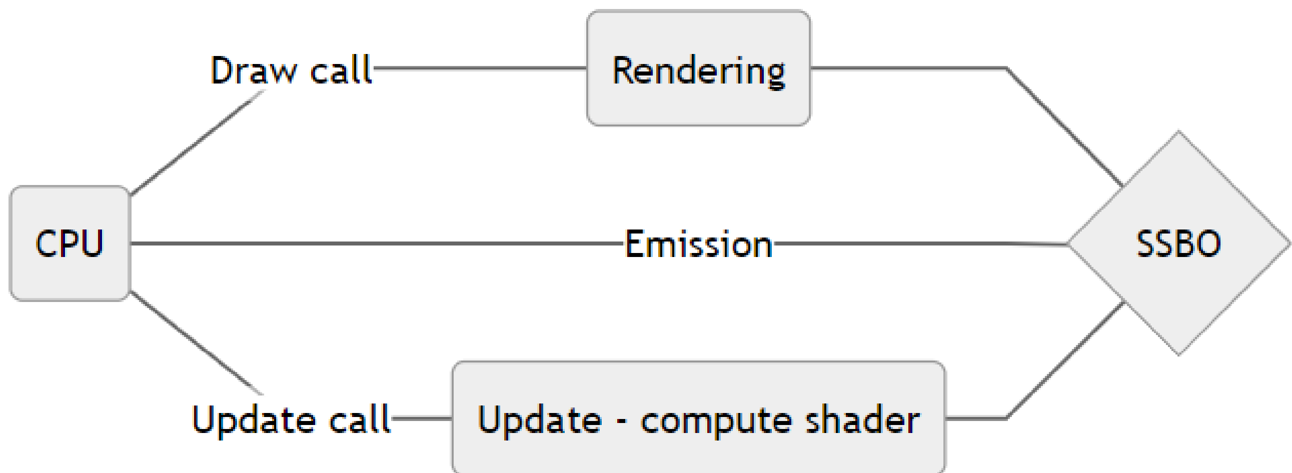


Рис 4.2 – Архітектура системи частинок

```
public class ParticleSystem : IDrawable
{
    public bool Emitting = true;
    public readonly Transform Transform;
    private readonly ParticlesUpdate _update;
    private readonly ParticlesView _view;
    private readonly Timer _timer;

    public ParticleSystem(Transform transform, ParticleSystemData data)
    {
        Transform = transform;
        _timer = new LocalTimer(1f / data.ParticlesPerSecond, Emit);
        _view = new ParticlesView(transform, data);
        _update = new ParticlesUpdate(data);
    }

    void IGameComponent.Initialize()
    {
```

```
        _view.Initialize();
        _update.Initialize();
    }

    void IGameComponent.Update(float deltaTime)
    {
        _update.Update(deltaTime);

        if (Emitting)
        {
            _timer.Update(deltaTime);
        }
    }

    private void Emit()
    {
        _update.Emit(Transform.Position, (int)(_timer.Difference / _timer.Rate));
    }

    public void Draw(in Matrix4 projectionMatrix, in Matrix4 viewMatrix)
    {
        _view.Draw(in projectionMatrix, in viewMatrix);
    }

    public void Dispose()
    {
        _view.Dispose();
    }
}


// Compute shader
#version 430 core
layout(local_size_x = 32, local_size_y = 1, local_size_z = 1) in;

uniform int colorsSize;
uniform vec4[30] colors;

uniform int sizeArrayLength;
uniform float[30] sizes;

uniform float deltaTime;
uniform float lifeTime;
uniform float speed;

struct Particle
{
    vec4 position;
    vec4 rotation;
    vec4 velocity;
    vec4 color;
    float scale;
    float elapsedTime;
    int enabled;
};

layout (binding = 0, std430) buffer ParticlesData // Той самий SSBO, про який йшла
мова вище

{
    Particle[] Particles;
```

```
};

bool needToUpdate(inout Particle particle)
{
    if (particle.enabled == 1)
    {
        particle.elapsedTime += deltaTime;

        if (particle.elapsedTime > lifeTime)
        {
            particle.enabled = 0;
        }
    }

    return particle.enabled == 1;
}

vec4 getColor(float lerp)
{
    float scale = 1 / float(colorsSize);
    int firstIndex = int(lerp / scale);
    int secondIndex = firstIndex + 1;
    lerp = smoothstep(firstIndex * scale, secondIndex * scale, lerp);

    return mix(colors[firstIndex], colors[secondIndex], lerp);
}

float getSize(float lerp)
{
    float scale = 1 / float(sizeArrayLength);
    int firstIndex = int(lerp / scale);
    int secondIndex = firstIndex + 1;
    lerp = smoothstep(firstIndex * scale, secondIndex * scale, lerp);

    return mix(sizes[firstIndex], sizes[secondIndex], lerp);
}

void update(inout Particle particle)
{
    particle.position += particle.velocity * deltaTime * speed;
    particle.rotation.z += deltaTime * 4;

    float lerp = particle.elapsedTime / lifeTime;

    particle.scale = getSize(lerp);
    particle.color = getColor(lerp);
}

void main()
{
    Particle particle = Particles[gl_GlobalInvocationID.x];

    if (needToUpdate(particle))
    {
        update(particle);
        Particles[gl_GlobalInvocationID.x] = particle;
    }
}

// Vertex shader
#version 430 core
```

```
layout(location = 0) in vec3 vertexPosition;
uniform int verticesCount;

struct Particle
{
    vec4 position;
    vec4 rotation;
    vec4 velocity;
    vec4 color;
    float scale;
    float elapsedTime;
    int enabled;
};

layout (binding = 0, std430) buffer ParticlesData // Той самий SSBO, про який йшла мова вище
{
    Particle[] Particles;
};

out vec4 inputColor;



void main(void)
{
    Particle particle = Particles[gl_InstanceID];

    mat4 model = build_transform(particle.position, particle.rotation, particle.scale);
    inputColor = particle.color;

    model = transpose(model);
    gl_Position = projectionMatrix * viewMatrix * model * vec4(vertexPosition, particle.enabled);
}
```

Слід зазначити, що у цій реалізації системи частинок, головним вузьким місцем буде створення частинок, так як звертання до пам'яті GPU з CPU.

Насправді, будь-які звертання GPU до CPU чи CPU до GPU в процесі виконання програми часто стають вузьким місцем.

Рис 4.3 – Демонстрація системи частинок за допогою шейдеру обчислення

В решті-решт, ми отримали гарненьку кастомізовану систему частинок, які оновлюються за відеокарті за допомогою шейдеру обчислень. Без нього, було б неможливо оброблювати такий великий масив даних ефективно. На gif зображенні присутні 100 тисяч частинок при 100 FPS.

Шейдери-обчислення також були використані генерації зображення для лабораторної роботи з «Інтелектуальних систем» - «Генетичний алгоритм».

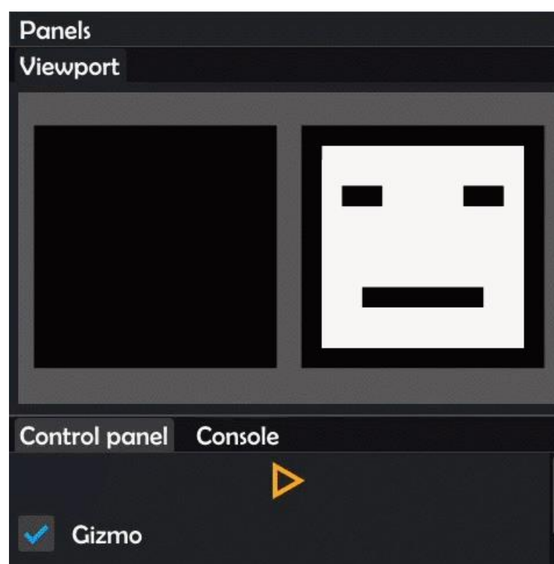


Рис 4.4 – Демонстрація використання шейдеру обчислення в лабораторній роботі

5 РІВЕНЬ ДЕТАЛІЗАЦІЇ (LOD)

Lod Group (Level of Detail Group) є важливою технікою оптимізації рендерингу графіки. Level of Detail (LOD) означає різні рівні деталізації моделей, які використовуються залежно від їх віддаленості від камери.

Відповідно, коли камера досить близько до об’єкта використовується LOD_0 (найкраща деталізація, найбільша кількість полігонів), коли далеко – LOD_N (найгірша деталізація, найменша кількість полігонів).

LOD Group є одною з найстаріших та найпростіших оптимізаційних технік, проте є досить потужною.

Алгоритм простий: в залежності від дистанцій камери до об’єкта, вибираємо меш з необхідною деталізацією. В прикладі будуть використовуватись безкоштовні моделі ялинок з Sketchfab з 4 рівнями деталізації.

Рис 5.1 – Приклад використаних моделей

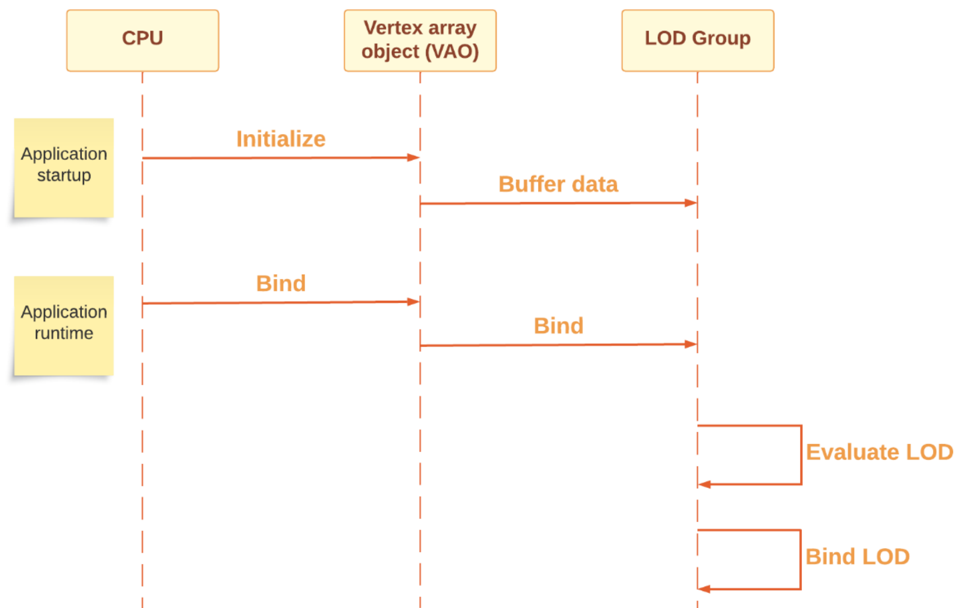


Рис 5.2 – Sequence-діаграма використання LOD групи

```
private MeshBinding EvaluateLOD()
{
    float distance = Vector3.Distance(_camera.Transform.Position, _transform.Position);
    MeshBinding result = _lodMeshes[0].Mesh;

    for (int i = 0; i < _lodMeshes.Length - 1; ++i)
    {
        if (distance > _lodMeshes[i].Distance)
        {
            result = _lodMeshes[i + 1].Mesh;
        }
    }

    return result;
}
```

Функція **EvaluateLOD** - вибір моделі, згідно з дистанцією камери до об'єкту.

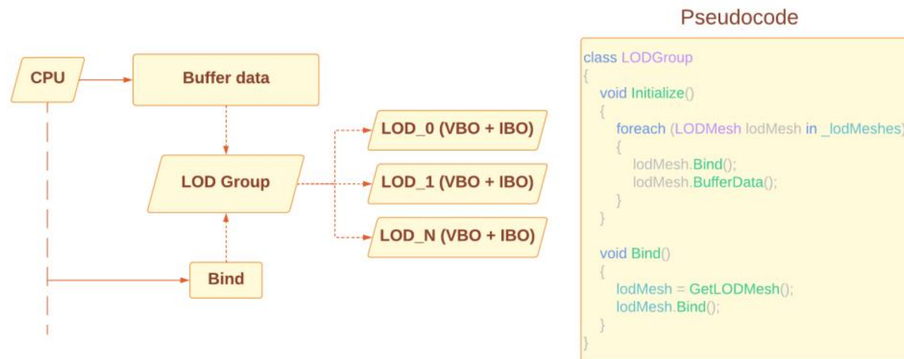


Рис 5.3 – Загальна архітектура LOD групи з псевдокодом

Важливим нюансом в реалізації LOD Group є його ініціалізація: щоб мінімізувати роботу з пам'яттю (яка досить сильно впливає на продуктивність програми), спочатку виділяємо і заповнюємо буфери під всі екземпляри моделі.

В процесі виконання програми, в залежності від дистанції обираємо буфер з потрібною нам деталізацією і віддаємо на рендер.

Таким чином, в процесі виконання програми відсутнє спілкування з пам’яттю, що є великим плюсом.

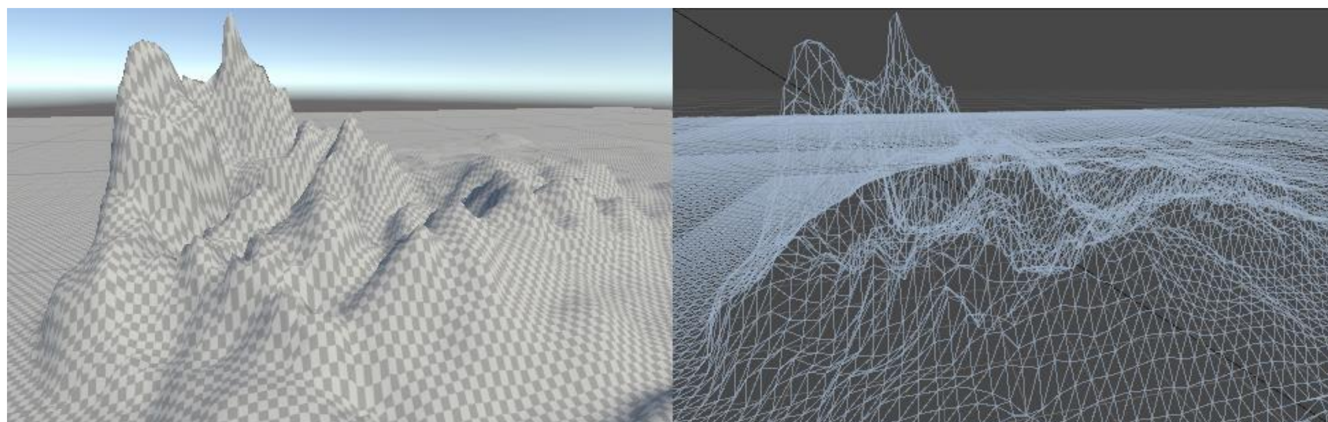
Нижче наведена демонстраційна сцена «Ліс ялинок», на якій помітний приріст FPS при віддаленості камери.

Рис 5.4 – Демонстрація сцени “LOD Group”

Виграш у майже 100 FPS. Проте слід зазначити, що ялинки самі по собі були у стилі low-poly, тому виграш по полігонам лише у два рази. На великій сцені з камінням, травою і т.д. виграш був би набагато суттєвіший.

Проте, рівень деталізації об’єкта не обов’язково має бути попередньо визначений. Найяскравішим прикладом є рель’єф місцевості або «Terrain».

Термін "terrain" в розробці ігор відноситься до фізичної або географічної структури, яка складається з поверхні, на якій розгортається гра. Він описує ландшафт, який включає гори, долини, ліси, річки, водойми, пустелі, міста та інші елементи оточення. І дуже важливо зробити террейн якомога легшим для рендеру. Тому, террейн зазвичай має свій власний алгоритм обрахунку рівня деталізації.



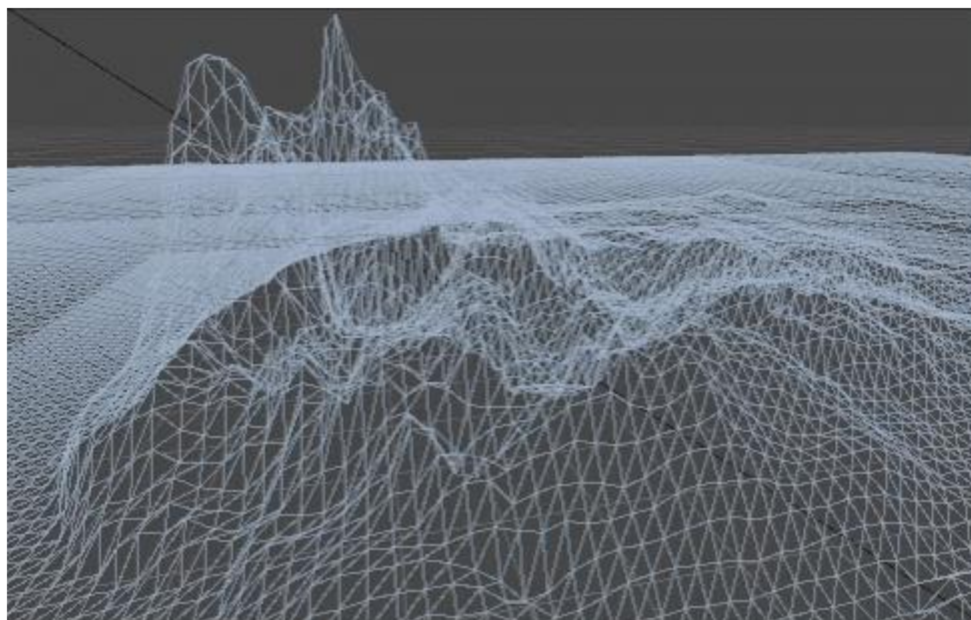


Рис 5.5 – Приклад рівня деталізації (LOD) террейну з рушія Unity.

Також слід згадати інноваційну технологію Nanite з рушія Unreal Engine 5, яка програмно обраховує LOD до бажаного об'єкту. Тут є декілька великих переваг:

- Відсутність потреби 3d-artist створювати попередньо визначені LOD: досить однієї high-poly моделі. Рушій власноруч обрахує LOD.

- З першого пункту отримаємо суттєвий приріст по займаної фізичної пам'яті (немає потреби у декількох варіантах однiє моделі)

- Якість LODу. При різкому зменшенні полігонів у попередньо визначених рівнях деталізації, це буде помітно середньостатистичному гравцю, що знизить занурення в гру. Алгоритм робить перехід більш плавним та непомітним.

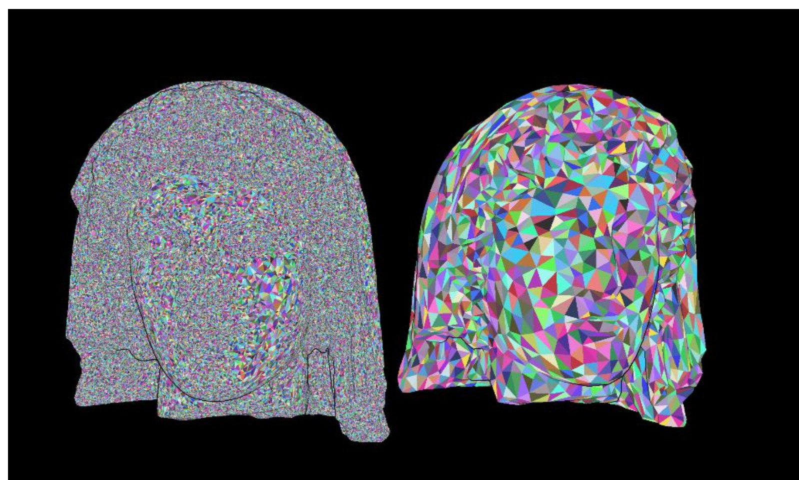


Рис 5.6 – Приклад технології Nanite з рушія Unreal Engine 5.

ВИСНОВОК

У даній роботі були розглянуті оптимізаційні техніки рендерингу графіки з метою поліпшення продуктивності програми. Досліджено такі методи, як статичний батчинг, динамічний батчинг, дублювання геометрії, шейдери обчислення та рівень деталізації.

Статичний батчинг використовується для об'єднання об'єктів з однаковими матеріалами та текстурами в один об'єднаний об'єкт, що дозволяє зменшити кількість викликів до графічного процесора і покращити продуктивність. Динамічний батчинг дозволяє об'єднувати об'єкти, які змінюють своє положення або властивості в кожному кадрі.

Дублювання геометрії (GPU instancing) дозволяє використовувати однакову геометрію для кількох об'єктів, що мають однакові властивості, зменшуючи тим самим навантаження на графічний процесор.

Шейдери обчислення (Compute shaders) використовуються для виконання обчислень на графічному процесорі, що дозволяє виконувати складні обчислювальні завдання та зменшує навантаження на центральний процесор.

Рівень деталізації (LOD) дозволяє змінювати деталізацію об'єктів залежно від їх віддаленості від камери, що допомагає зменшити обсяг обчислень та покращити продуктивність.

У результаті проведених досліджень було реалізовано рендер-рушій Dengine за допомогою графічного інтерфейсу OpenGL та C#, на його основі виконано дві лабораторні роботи з «Інтелектуальних систем»: Пакман та генетичний алгоритм (генерація зображення).

ПЕРЕЛІК ДЖЕРЕЛ

1. https://docs.unity3d.com/560/Documentation/Manual/DrawCallBatching.html - Batching
2. https://docs.unity3d.com/Manual/static-batching.html - Static Batching
3. https://docs.unity3d.com/Manual/dynamic-batching.html - Dynamic batching
4. https://learnopengl.com/Advanced-OpenGL/Instancing - GPU instancing
5. https://www.khronos.org/opengl/wiki/Compute_Shader - Compute shaders
   https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction - GPU computing
6. http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/ - Particle system
7. https://learnopengl.com/In-Practice/2D-Game/Particles - Particle system
   https://docs.unity3d.com/Manual/class-LODGroup.html - LOD Group
8. https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/AutomaticLODGeneration/ - LOD Group
   https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/ - Nanite