Геометрические шейдеры

Компьютерная графика

Впервые появились

- Геометрические шейдеры были введены компанией Microsoft в DirectX10, а затем были добавлены в ядро OpenGL версии 3.2.

Место в контейнере

- Вершинные шейдеры — для вершин
- Тесселяционные шейдеры — для патчей
- **Геометрические шейдеры — для примитивов**
- Растеризация
- Фрагментные шейдеры — для пикселей

Особенности геометрического шейдера

- Может создавать новые примитивы на основе существующих
- Может изменять топологию примитивов
- Работает с целыми примитивами (точками, линиями, треугольниками)
- Необязательный этап конвейера

Уникальность геометрических шейдеров

Уникальный взгляд на модель, в которой **связи между вершинами** доступны разработчику, позволяя строить новые методы, основываясь на этих знаниях.

Геометрический шейдер может изменять примитивы

- **Изменять топологию** входящих примитивов.
  Геометрический шейдер может **принимать** примитивы **любого** типа, но **выводить** может только **списки точек**, **стрип линий** и **стрип треугольников**.

- Геометрический шейдер принимает один примитив и может
  либо **удалить** его полностью,
  либо отправить **на выход** один или **несколько** примитивов
  (это значит, что он **может выпускать и меньше и больше вершин, чем получает**).
  Эта способность известна как **growing geometry**.

Типы примитивов

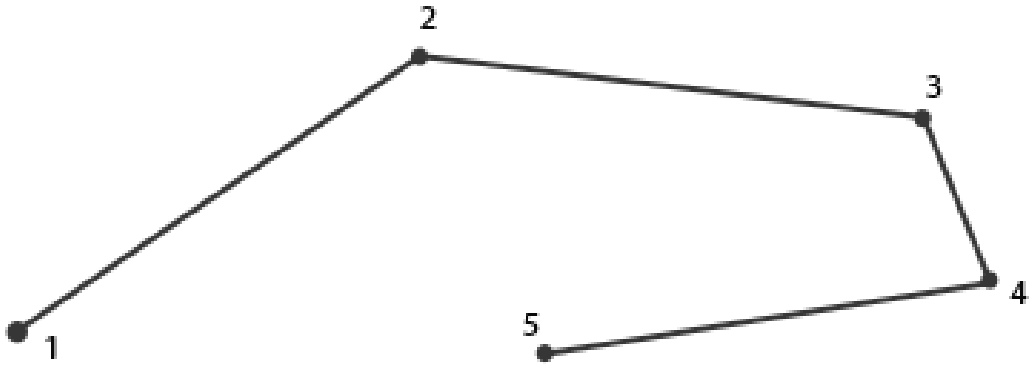
Входные примитивы:

- points — точки
- lines — линии
- triangles — треугольники
- lines_adjacency — линии с смежными вершинами
- triangles_adjacency — треугольники с смежными вершинами

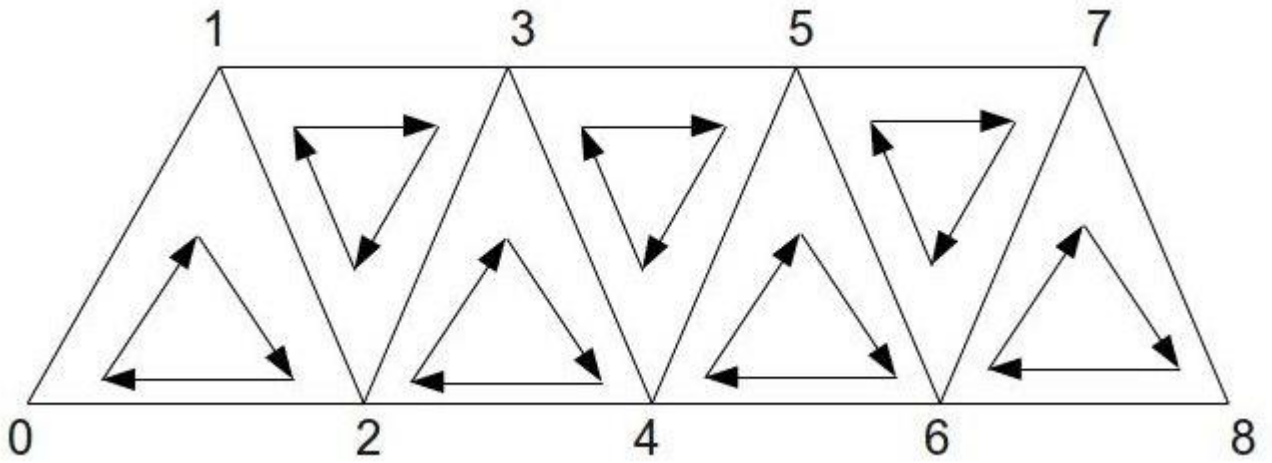
Выходные примитивы:

- points
- line_strip
- triangle_strip

Line Strip



Стрип треугольников



Важное свойство касательно порядка внутри треугольника — порядок номеров вершин обратен у каждого второго треугольника.

Это значит, что порядок таков: [0,1,2], [1,3,2], [2,3,4], [3,5,4] и т.д.

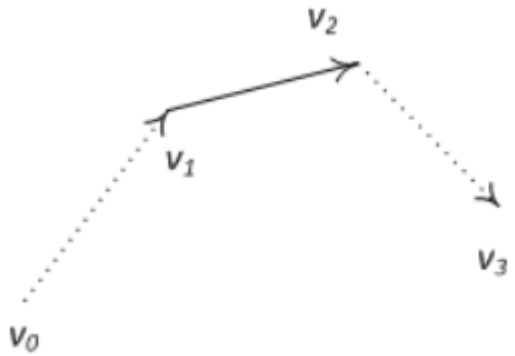
Пример геометрического шейдера

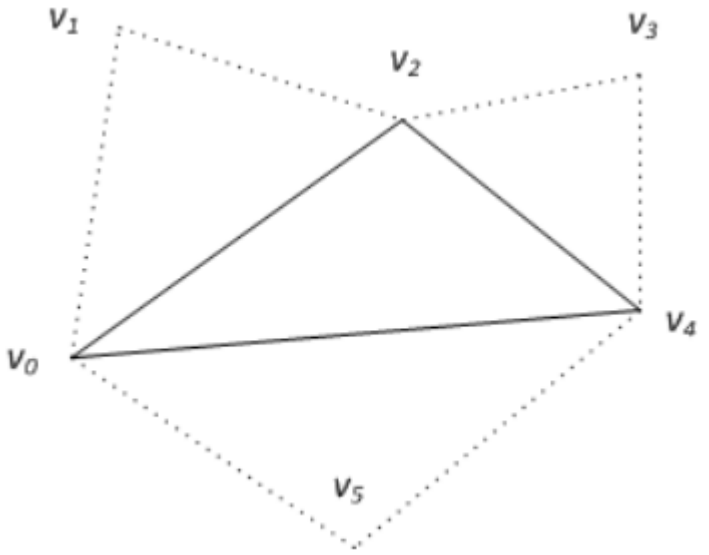
```
#version 330 core

layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

layout(input_primitive) in;

// тип примитива, данные которого поступают со стадии вершинного шейдера

points
lines
triangles
lines_adjacency
triangles_adjacency





Пример геометрического шейдера

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Получение типа входных примитивов из уже слинкованной программы

```
int   type;

glGetProgramiv ( program, GL_GEOMETRY_INPUT_TYPE, &type );
```

layout(output_primitive, max_vertices = vert_count) out;

// Выходной тип примитива с максимальным количеством вершин

- points
- line_strip
- triangle_strip

Пример геометрического шейдера

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Получение типа выходного примитива и максимального числа выводимых вершин из уже слинкованной программы

```
int   type, num;

glGetProgramiv ( program, GL_GEOMETRY_OUTPUT_TYPE,  &type );
glGetProgramiv ( program, GL_GEOMETRY_VERTICES_OUT, &num );
```

Встроенная переменная gl_in

```
//Приблизительное представление

in gl_Vertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

Пример геометрического шейдера

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```

Геометрический шейдер имеет доступ к

- uniform-переменным
- выходным переменным вершинного шейдера
- текстурам

Из вершинного в геометрический

Пусть в вершинном шейдера имеется следующая выходная переменная

```
out vec3 normal;        // normal for vertex
```

Тогда в геометрическом шейдере она должна быть описана следующим образом

```
in vec3 normal [];      // arrays of normals for every vertex
```

Выходные переменные

Результаты работы геометрического шейдера записываются в следующие переменные:

- gl_Position,
- gl_PointSize,
- gl_ClipDistance [],
- gl_PrimitiveID,
- gl_Layer,
- gl_ViewportIndex.

Каждый вызов **EmitVertex()** добавляет текущее значение в переменной **gl_Position** к текущему экземпляру примитива.

Когда же мы вызываем **EndPrimitive()**, все порожденные вершины окончательно связываются в указанный выходной тип примитива.

Пример геометрического шейдера

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex(); // окончание формирования вершины

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex(); // окончание формирования вершины

    EndPrimitive();
}
```

Повторяя вызовы **EndPrimitive()** после одного или более вызовов **EmitVertex()** можно продолжать создавать новые экземпляры примитивов.

Конкретно в примере генерируется по две вершины, смещенные на небольшое расстояние от положения входной вершины, а затем выполняется вызов **EndPrimitive()**, формирующий из этих двух сгенерированных вершин один **line strip**, содержащий две вершины.

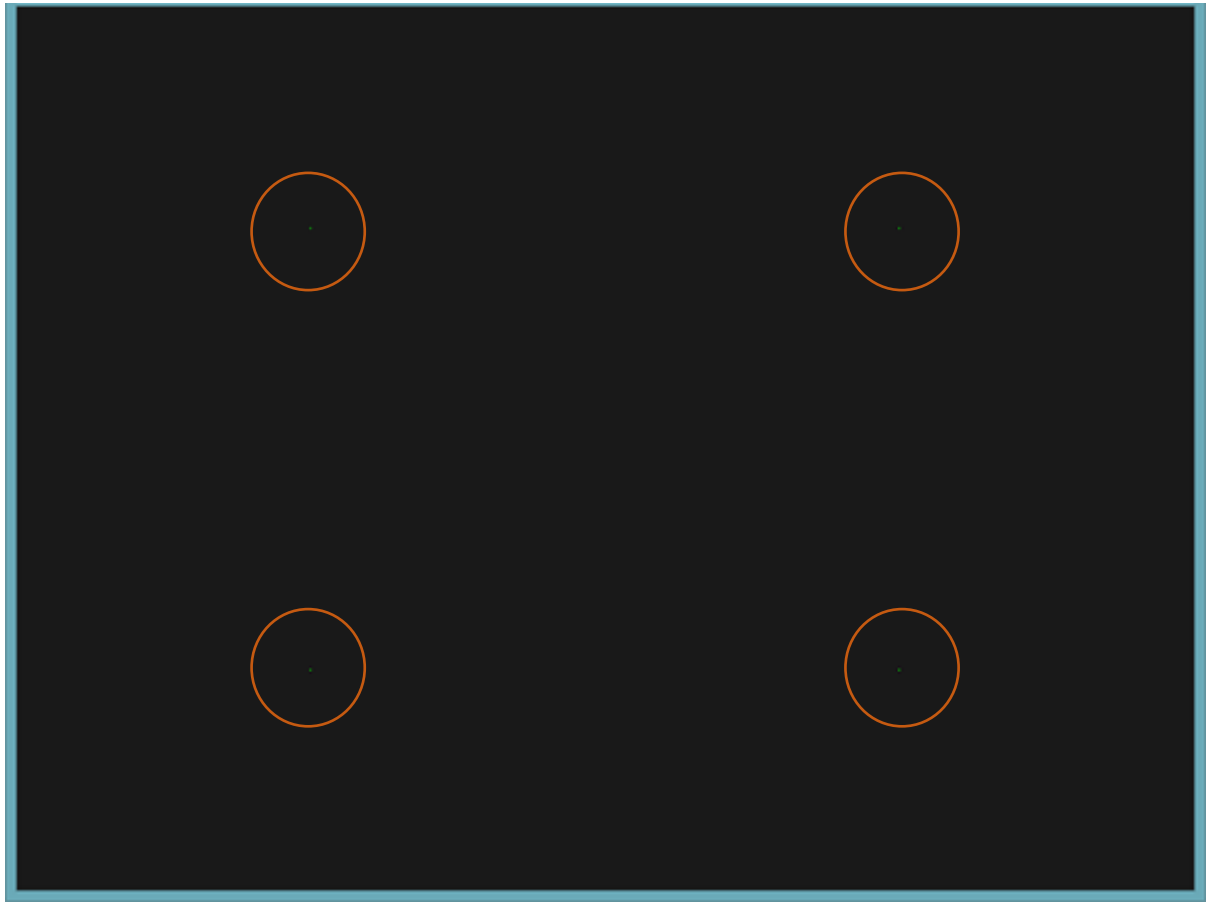
Получили такие результаты, выполнив всего лишь один вызов отрисовки:

glDrawArrays(GL_POINTS, 0, 4);

Компиляция и линковка

```
geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geometryShader, 1, &gShaderCode, NULL);
glCompileShader(geometryShader);
...
glAttachShader(program, geometryShader);
glLinkProgram(program);
```

Использование геометрического шейдера



Входные данные

```
float points[] = {
        -0.5f,  0.5f, // верхняя-левая
         0.5f,  0.5f, // верхняя-правая
         0.5f, -0.5f, // нижняя-правая
        -0.5f, -0.5f  // нижняя-левая
};
```

Вершинный шейдер

```
#version 330 core
layout (location = 0) in vec2 aPos;

void main() {
   gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

Геометрический шейдер, который просто берет данные входного примитива и отправляет на выход без изменений

```
#version 330 core
layout (points) in;
layout (points, max_vertices = 1) out;

void main() {
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
    EndPrimitive();
}
```

Фрагментный шейдер

```
#version 330 core
out vec4 FragColor;

void main() {
    FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

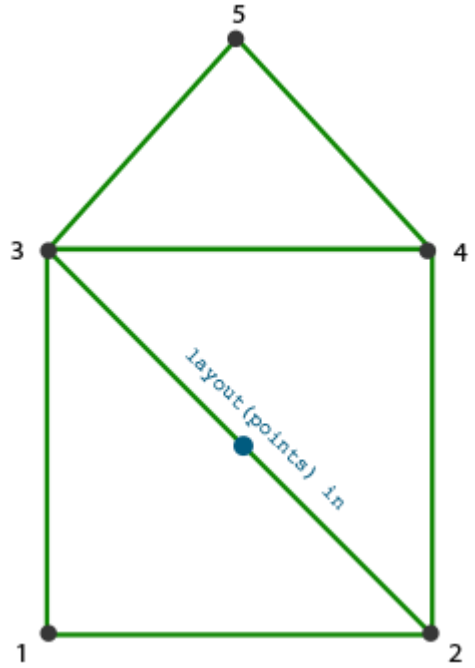
В коде программы

Создаем VAO и VBO для вершинных данных и осуществляем рендер вызовом glDrawArrays():

```
shader.use();
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 4);
```

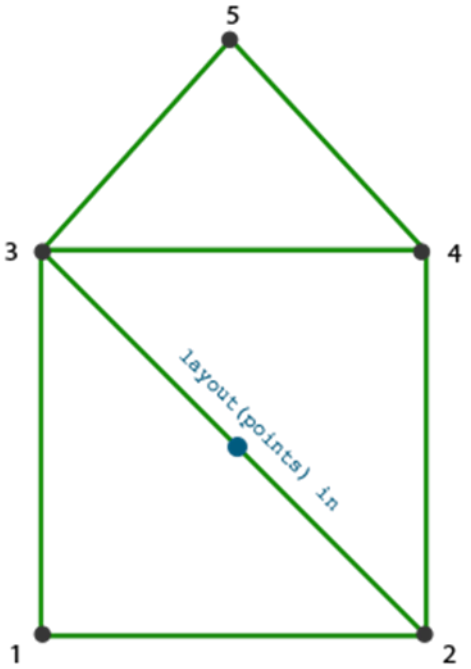
Строим домики

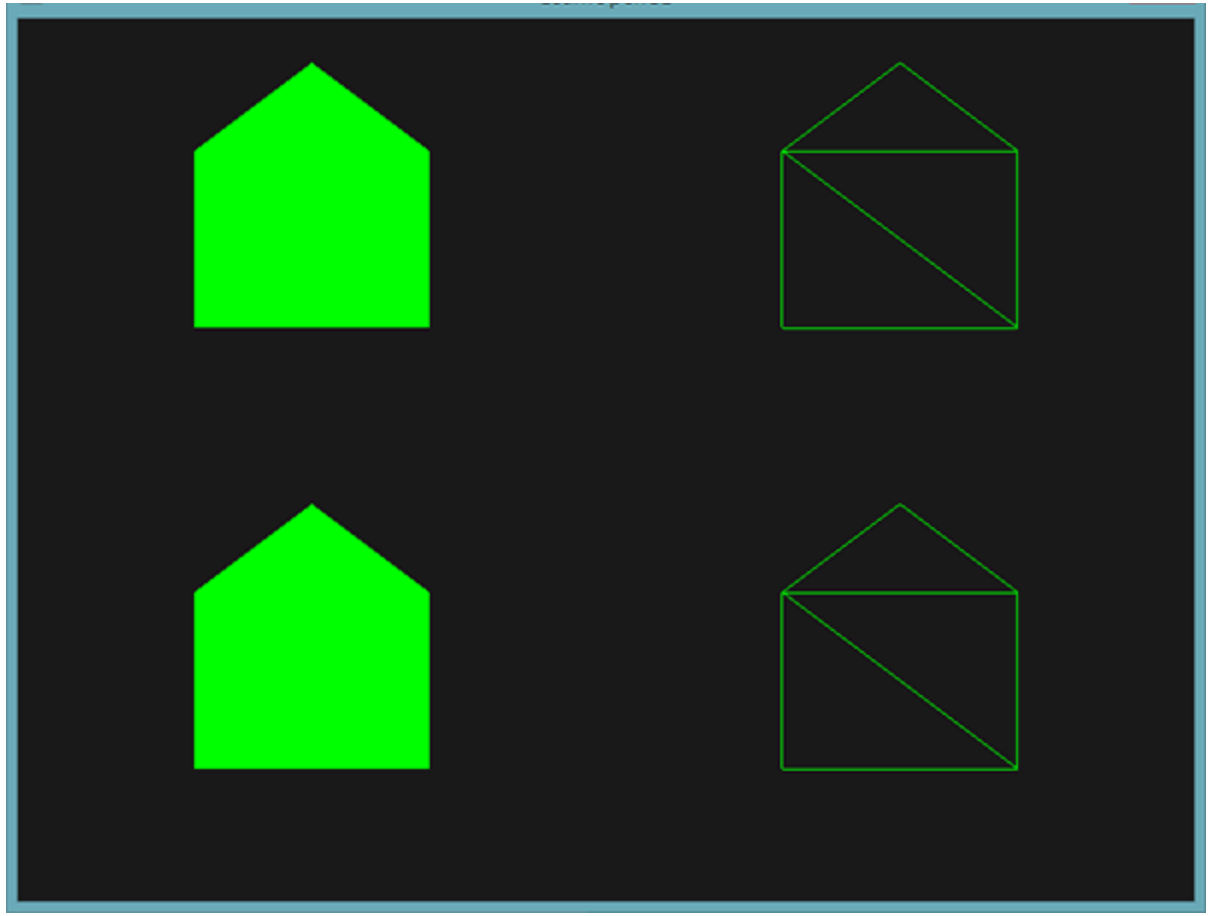
Понадобится сменить тип выходного примитива **на triangle_strip** и нарисовать три треугольника: два для создания квадратной основы и один для крыши



Результирующий геометрический шейдер

```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position) {
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:top
    EmitVertex();
    EndPrimitive();
}
void main() {
    build_house(gl_in[0].gl_Position);
}
```





Обновленные данные вершин (с цветом)

```
float points[] = {
    -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, // верхняя-левая
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, // верхняя-правая
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // нижняя-правая
    -0.5f, -0.5f, 1.0f, 1.0f, 0.0f  // нижняя-левая
};
```

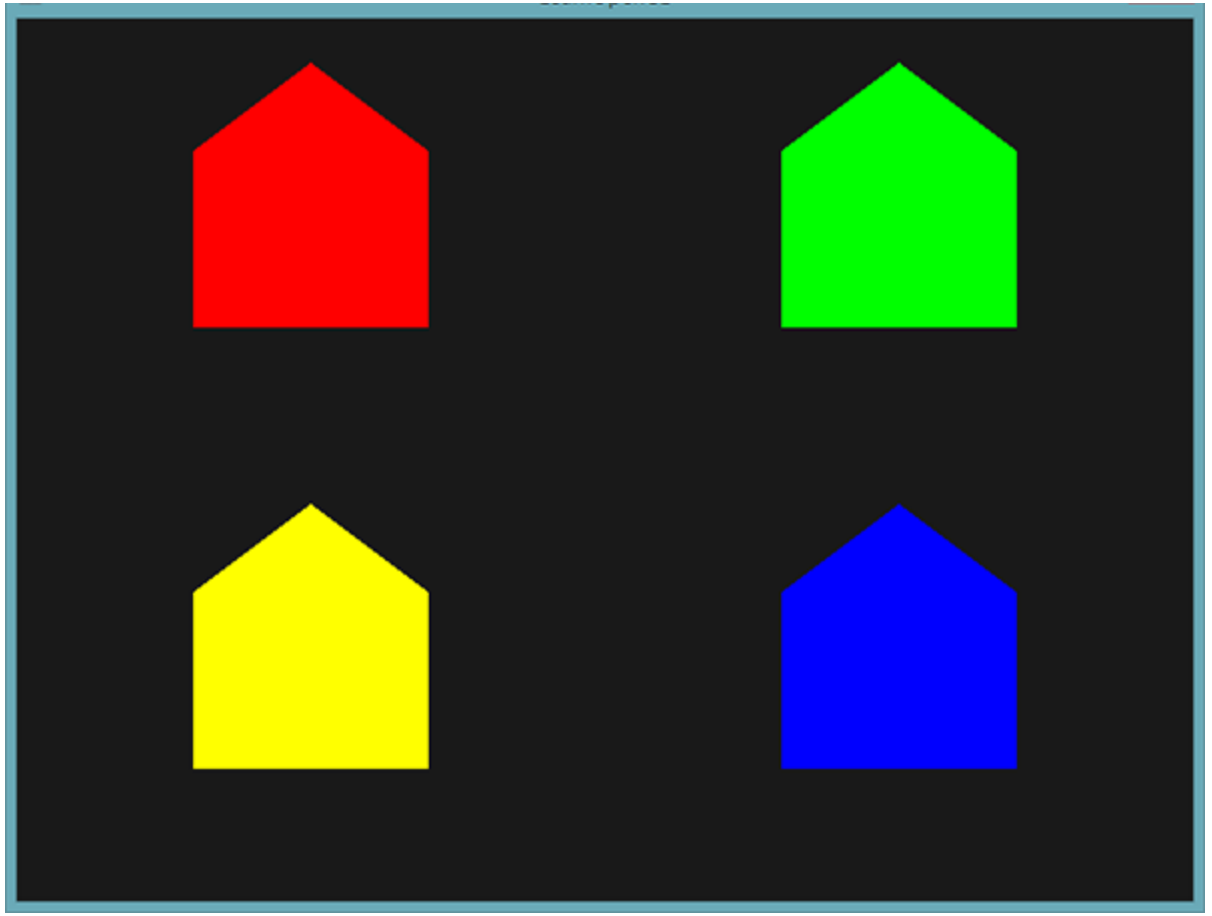
Вершинный шейдер для передачи атрибута цвета в геометрический шейдер с использованием интерфейсного блока

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out VS_OUT {
    vec3 color;
} vs_out;

void main() {
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    vs_out.color = aColor;
}
```

Интерфейсный блок того же типа (но с другим именем) в геометрическом шейдере

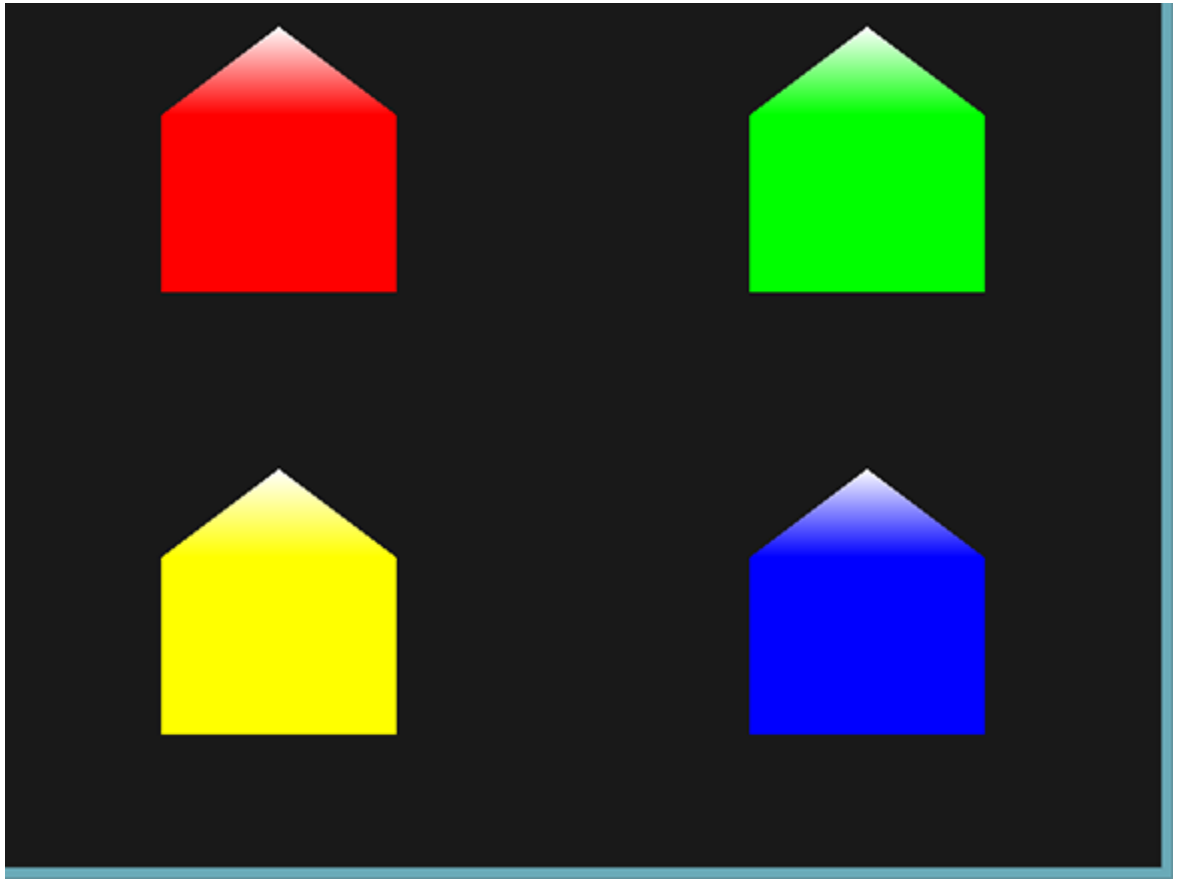
```
in VS_OUT {
    vec3 color;
} gs_in[];
```

```
out vec3 fColor;

…

fColor = gs_in[0].color; // используется gs_in[0] поскольку на входе у нас единственная вершина
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:нижняя-левая
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:нижняя-правая
EmitVertex();
gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:верхняя-левая
EmitVertex();
gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:верхняя-правая
EmitVertex();
gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:крыша
EmitVertex();
EndPrimitive();
```



Присыпим крыши домиков снегом

```
fColor = gs_in[0].color;
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:нижняя-левая
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:нижняя-правая
EmitVertex();
gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:верхняя-левая
EmitVertex();
gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:верхняя-левая
EmitVertex();
gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:крыша
fColor = vec3(1.0, 1.0, 1.0);
EmitVertex();
EndPrimitive();
```



Взрываем объекты

Перемещение каждого треугольника вдоль направления нормали с течением времени.

В результате этот эффект даёт подобие взрыва объекта, разделяя его на отдельные треугольники, движущиеся по направлению своего вектора нормали.

Использование геометрического шейдера позволяет эффекту работать на любом объекте, вне зависимости от его сложности

Вычисление вектора нормали по трём вершинам входного треугольника

```
vec3 GetNormal() {
  vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
  vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
  return normalize(cross(a, b));
}
```

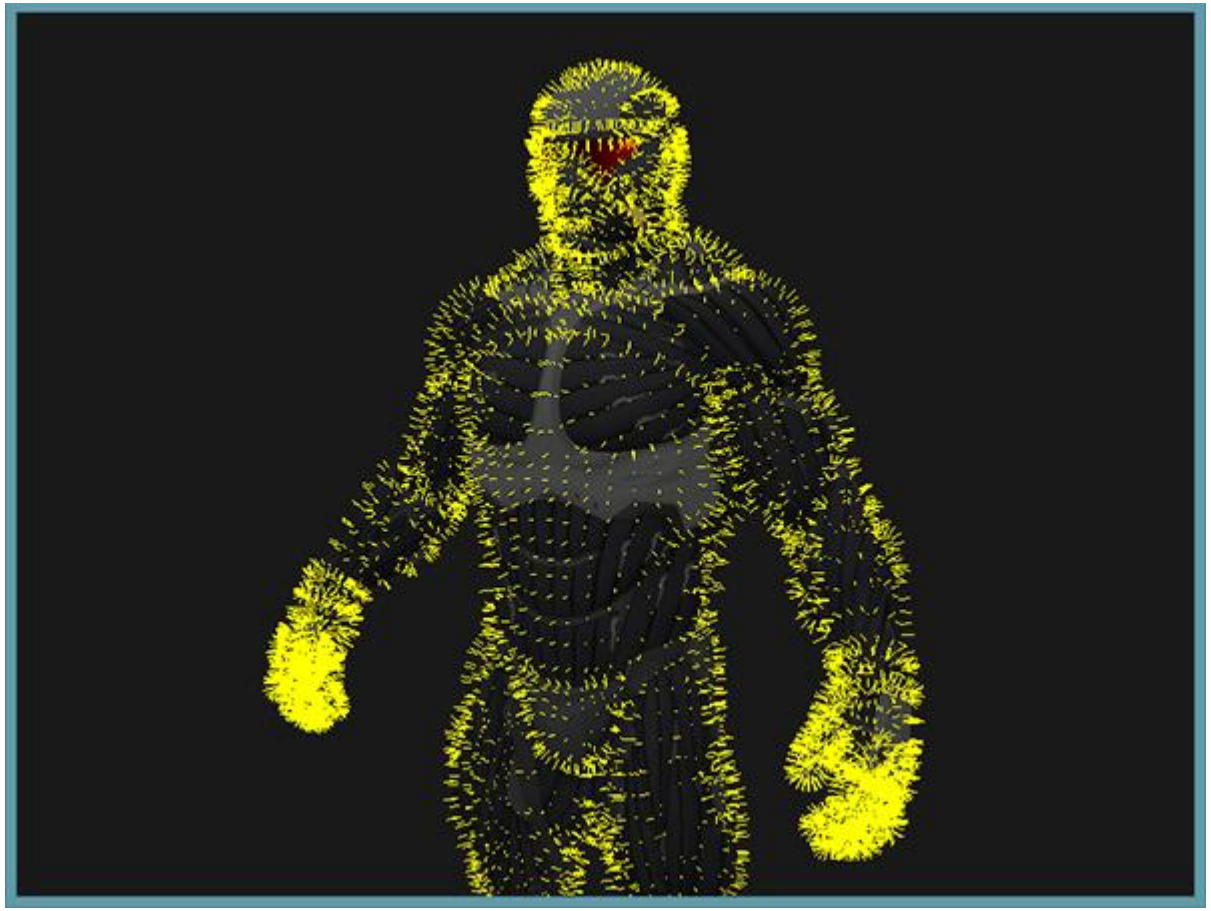
Функция принимает вектора нормали и положения вершины, а возвращает новое положение вершины, смещённое вдоль нормали

```
vec4 explode(vec4 position, vec3 normal) {
    float magnitude = 2.0;
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;
    return position + vec4(direction, 0.0);
}
```

main() геометрического шейдера

```
void main() {
    vec3 normal = GetNormal();

    gl_Position = explode(gl_in[0].gl_Position, normal);
    TexCoords = gs_in[0].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[1].gl_Position, normal);
    TexCoords = gs_in[1].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[2].gl_Position, normal);
    TexCoords = gs_in[2].texCoords;
    EmitVertex();
    EndPrimitive();
}
```

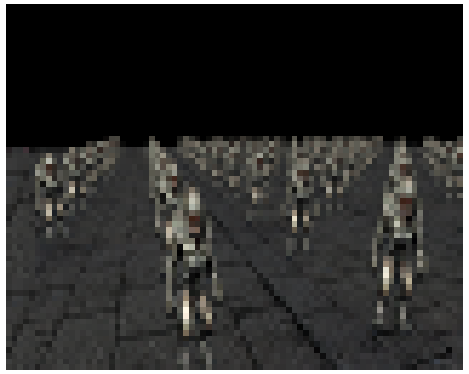
Визуализация нормалей

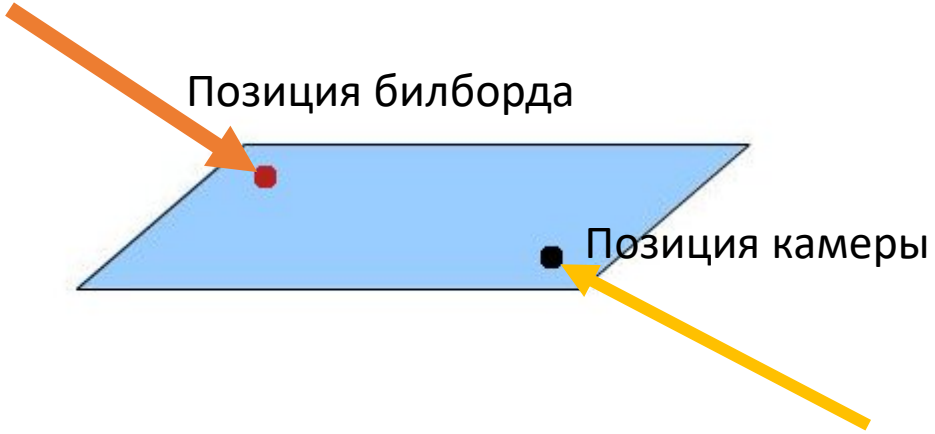


Реализация метода billboarding

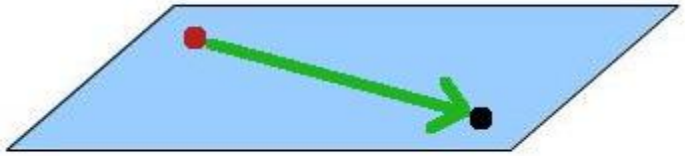
Billboarding - прямоугольник, который всегда направлен в камеру.

При движении камеры по сцене billboard вращается за ней так, что вектор из billboard до камеры всегда перпендикулярен поверхности billboard.

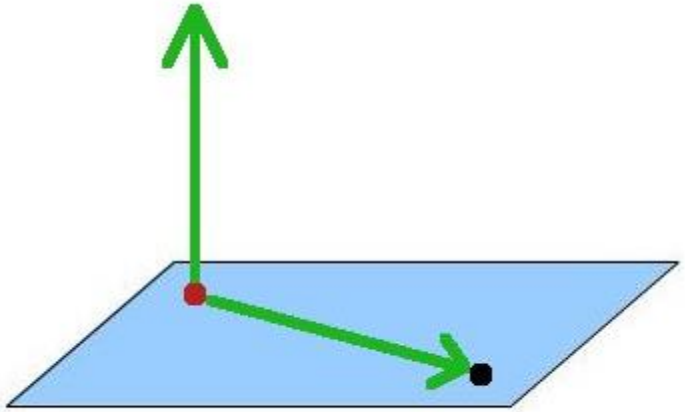


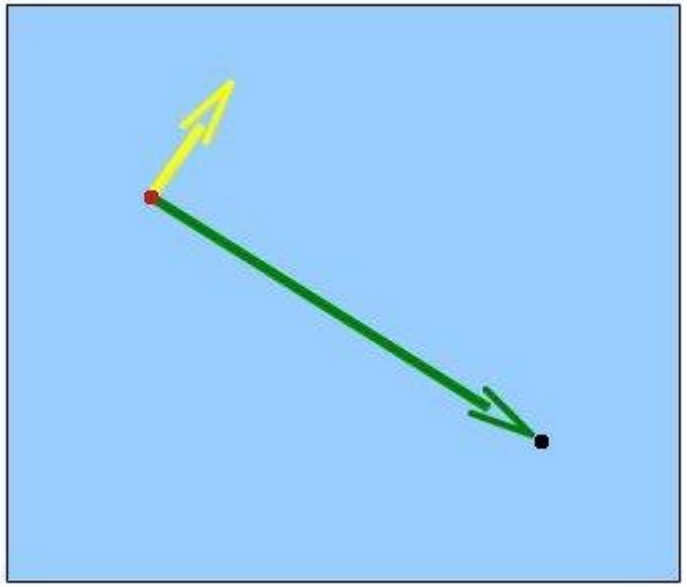


вектор из позиции billboard в камеру



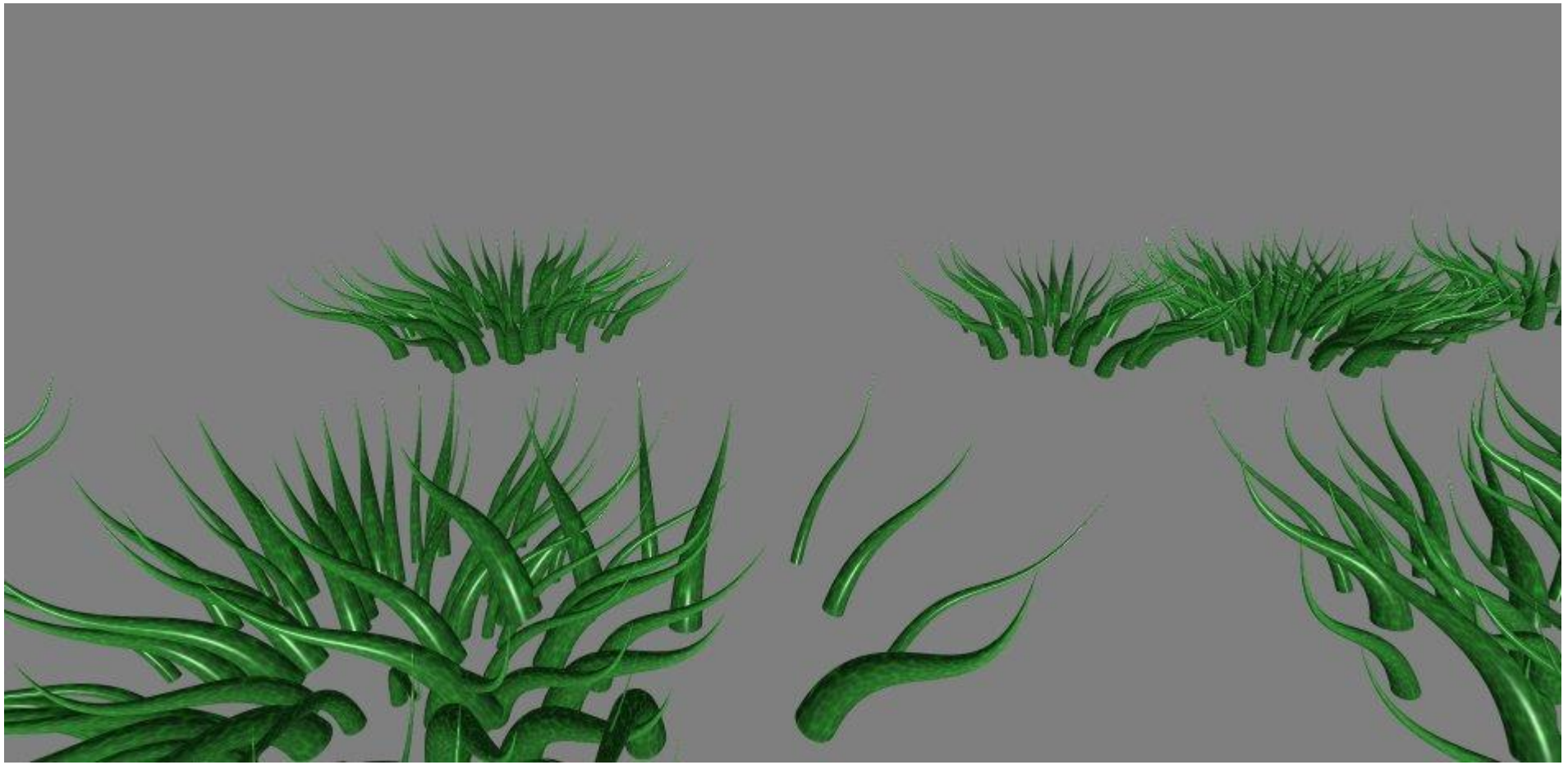
вектор (0,1,0):





желтый вектор — результат векторного произведения

Процедурная растительность



Проблемы производительности

- Отсутствие кэширования: каждый примитив обрабатывается независимо
- Перерасход вершин: если max_vertices установлен слишком высоким
- Разветвление выполнения: сложные условия в шейдере снижают производительность

Рекомендации по оптимизации

- Минимизировать количество генерируемых вершин
- Избегать сложных ветвлений
- Использовать для небольших преобразований
- Рассмотреть альтернативы (тесселяционные шейдеры, compute shaders)

Ключевые моменты

- Геометрические шейдеры — мощный, но специфический инструмент
- Идеальны для простой генерации и преобразования геометрии
- Требуют внимания к производительности
- Постепенно заменяются более современными технологиями

Современные тенденции и альтернативы

Mesh Shaders (DirectX 12 Ultimate, Vulkan)

- Более гибкая модель, заменяющая геометрические/тесселяционные шейдеры
- Работают с группами примитивов
- Лучшая производительность и контроль

Compute-based geometry processing

- Использование compute shaders для сложных геометрических преобразований
- Больший контроль над памятью и выполнением
- Требует больше кода для интеграции в конвейер