Компьютерная графика
Современные технологии компьютерной графики и рендеринга

Программируемая геометрия на GPU : тесселяционные и геометрические шейдеры в OpenGL 4.x

Лекция 9

02.04.02 ФИИТ

Разработка мобильных приложений и компьютерных игр

2025-2026

Ограничения классического конвейера

В стандартном OpenGL-конвейере (**вершинный → фрагментный шейдер**) мы сталкиваемся с тремя **фундаментальными ограничениями**:

Ограничение	Проявление
Фиксированная геометрия	Количество вершин неизменно после загрузки в GPU
Отсутствие доступа к топологии	Вершинный шейдер не знает о соседних вершинах
Нет порождения вершин	Нельзя увеличить детализацию или изменить примитив

Что дают программируемые геометрические стадии?

Тесселяция (Tessellation) — разбиение примитива на множество меньших примитивов.

Геометрические шейдеры (Geometry Shaders) — изменение топологии и порождение новых примитивов.

```
// Без тесселяции
glDrawArrays(GL_TRIANGLES, 0, vertexCount);
```

Рисует примитивы типа GL_TRIANGLES
Каждые 3 вершины образуют один треугольник
Вершины идут напрямую в
вершинный шейдер → растеризатор → экран
Нет этапа подразбиения геометрии

```
// С тесселяцией
glPatchParameteri(GL_PATCH_VERTICES, 3);
glDrawArrays(GL_PATCHES, 0, vertexCount);
```

Рисует примитивы типа GL_PATCHES (спец тип для тесселяции)
glPatchParameteri(GL_PATCH_VERTICES, 3) — каждый патч состоит из 3 контрольных точек (как треугольник)
Вершины проходят через дополнительные этапы
Вершинный шейдер → Шейдер тесселяции (контроль) → Тесселятор (аппаратный) → Шейдер тесселяции (оценка) → Геометрический шейдер (опционально) → Растеризатор

Почему это важно?

Ключевая идея:
обе стадии **работают на GPU**, позволяя реализовать LOD, процедурную геометрию и спецэффекты без загрузки CPU.

Метрика	CPU-подход	GPU-подход
Вершин в кадр (ландшафт)	~1 млн	~100 млн
Загрузка PCIe (MB/кадр)	50–200	0.5–2
Задержка смены LOD	5–20 мс	~0.1 мс

LOD (Level of Detail) — динамически увеличивать детализацию ближних объектов
Сглаживание — подразбивать грубые модели на более гладкие
Процедурная геометрия — создавать детали на лету (террейн, вода)

Мотивация

- Качество визуализации
- Экономия памяти
- Динамический уровень детализации
- Выполнение расчетов на уровне вершин

Base model

Bump mapping

Displacement mapping

Области применения тесселяции (tessellation)

Тесселяция — разбиение примитива (патча) на множество меньших примитивов (точек, линий, треугольников)

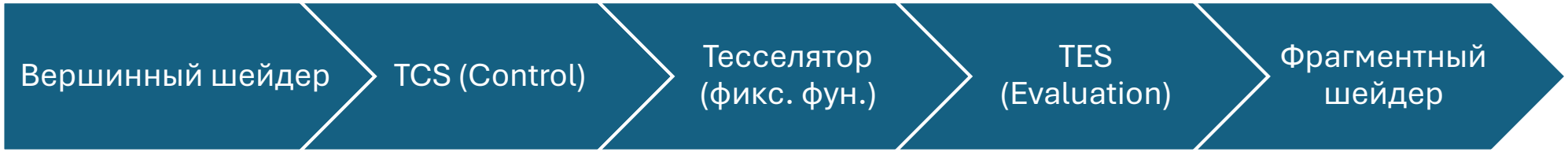
Области применения

- Ландшафты с динамическим LOD
- Повышение детализации геометрических моделей
- Рендеринг травы, волос, частиц
- Подразделение поверхностей (subdivision surfaces)
- PN-треугольники и Фонг-тесселяция для сглаживания

Рендеринг травы, волос, частиц

Конвейер тесселяции в OpenGL 4.0

Вместо обычных примитивов используются патчи (GL_PATCHES) — примитивы с числом вершин от 1 до 32.



Tessellation Control Shader (TCS) — задаёт параметры разбиения

Tessellation Primitive Generator — аппаратное разбиение (фиксированная функция)

Tessellation Evaluation Shader (TES) — вычисляет итоговые вершины

Tessellation Control Shader (TCS)

TCS — это **решатель: сколько геометрии генерировать и как передать данные дальше**

- Выполняется для каждой контрольной вершины патча
- Имеет доступ ко всем вершинам патча (через массивы)
- Задаёт уровни тесселяции для тесселятора

TCS — опциональный этап. Если его нет, то тесселятор использует уровни по умолчанию из glPatchParameterfv().

Входные данные TCS (Tessellation Control Shader)

gl_in[] — геометрические данные патча

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

// количество вершин патча
in int gl_PatchVerticesIn;
 // индекс примитива (Счётчик отрисованных патчей, начинается с 0)
in int gl_PrimitiveID;
// номер текущей вершины (0..vertices-1)
in int gl_InvocationID;
```

Все входные параметры, пришедшие от вершинного шейдера, передаются как массивы (так как шейдер имеет доступ ко всем вершинам патча сразу), при этом размер массива указывать не обязательно.

Выходные данные TCS

gl_out[] — геометрические данные для TES (Tessellation Evaluation Shader**)**

```
out gl_PerVertex {
   vec4 gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
} gl_out[];

// внешние уровни (границы)
out float gl_TessLevelOuter[4];
// внутренние уровни
out float gl_TessLevelInner[2];
```

Ключевые концепции. Режим работы на вершину

```
layout(vertices = 3) out;  // патч из 3 вершин
#define id gl_InvocationID  // id = 0, 1 или 2
```

TCS вызывается для каждой вершины патча отдельно

gl_InvocationID говорит, какую вершину мы сейчас обрабатываем (0, 1 или 2 для треугольника)

Ключевые концепции. Передача атрибутов

vertes[id].position = vertcs[id].position;

vertcs[id] — данные от вершинного шейдера для текущей вершины

vertes[id] — данные, которые получат соответствующие вершины в TES

Ключевые концепции. Асинхронность и барьеры

```
if (id == 0) {
   // Только первая вершина задаёт уровни тесселяции
   gl_TessLevelInner[0] = inner;
   gl_TessLevelOuter[0] = outer;
}
```

Важно: Уровни тесселяции должны быть одинаковыми для всех вершин патча.
Поэтому их обычно задаёт только одна вершина (id == 0).

Если бы все 3 вершины пытались писать в gl_TessLevelOuter, возникла бы гонка данных.

Где и когда нужен barrier()?

Если вы хотите, чтобы вершины общались друг с другом (например, вычисляли среднюю точку), нужно:

```
// Все вершины пишут свои данные
vertes[id].position = vertcs[id].position;
barrier();  // Ждём, пока все вершины запишут

// Теперь можно читать данные других вершин
if (id == 0) {
   vec3 avg = (vertes[0].position + vertes[1].position + vertes[2].position) / 3.0;
   // ...
}
```

Практический пример: LOD по расстоянию

```
uniform vec3 cameraPos;

void main() {
   vertes[id] = vertcs[id];  // копируем данные
   barrier();  // Ждём, пока все вершины запишут
   if (id == 0) {
      // Вычисляем центр патча
      vec3 center = (vertcs[0].position +  vertcs[1].position +  vertcs[2].position) / 3.0;
      float dist = distance(cameraPos, center);
      // Близко - много деталей, далеко - мало
      float level = max(1.0, 16.0 * (1.0 - dist / 100.0));

      gl_TessLevelInner[0] = level;
      gl_TessLevelInner[1] = level;
      gl_TessLevelOuter[0] = level;
      gl_TessLevelOuter[1] = level;
      gl_TessLevelOuter[2] = level;
      gl_TessLevelOuter[3] = level;
   }
}
```

Типичные ошибки

Забыли layout(vertices = N) out → ошибка компиляции

Разные уровни тесселяции у вершин → неопределённое поведение

Не передали gl_Position → дальше пойдут мусорные координаты

Уровень <= 0 → патч полностью отбрасывается (хороший способ для frustum culling!)

Ключевые моменты TCS (Tessellation Control Shader)

Размер массивов определяется layout(vertices = N) out или glPatchParameteri(GL_PATCH_VERTICES, N)

Один вызов TCS = одна вершина (из-за gl_InvocationID)

Уровни тесселяции задаются только один раз для всего патча

Выходные данные будут интерполироваться тесселятором между контрольными точками

gl_out[] обязателен, пользовательские данные — опциональны

TCS получает целый патч, но обрабатывает его по вершинам, а на выход задаёт правила подразбиения и передаёт данные для интерполяции.

Тесселятор (фиксированная стадия)

Используя факторы тесселяции преобразует входной патч в набор примитивов:

- точки
- линии
- треугольники

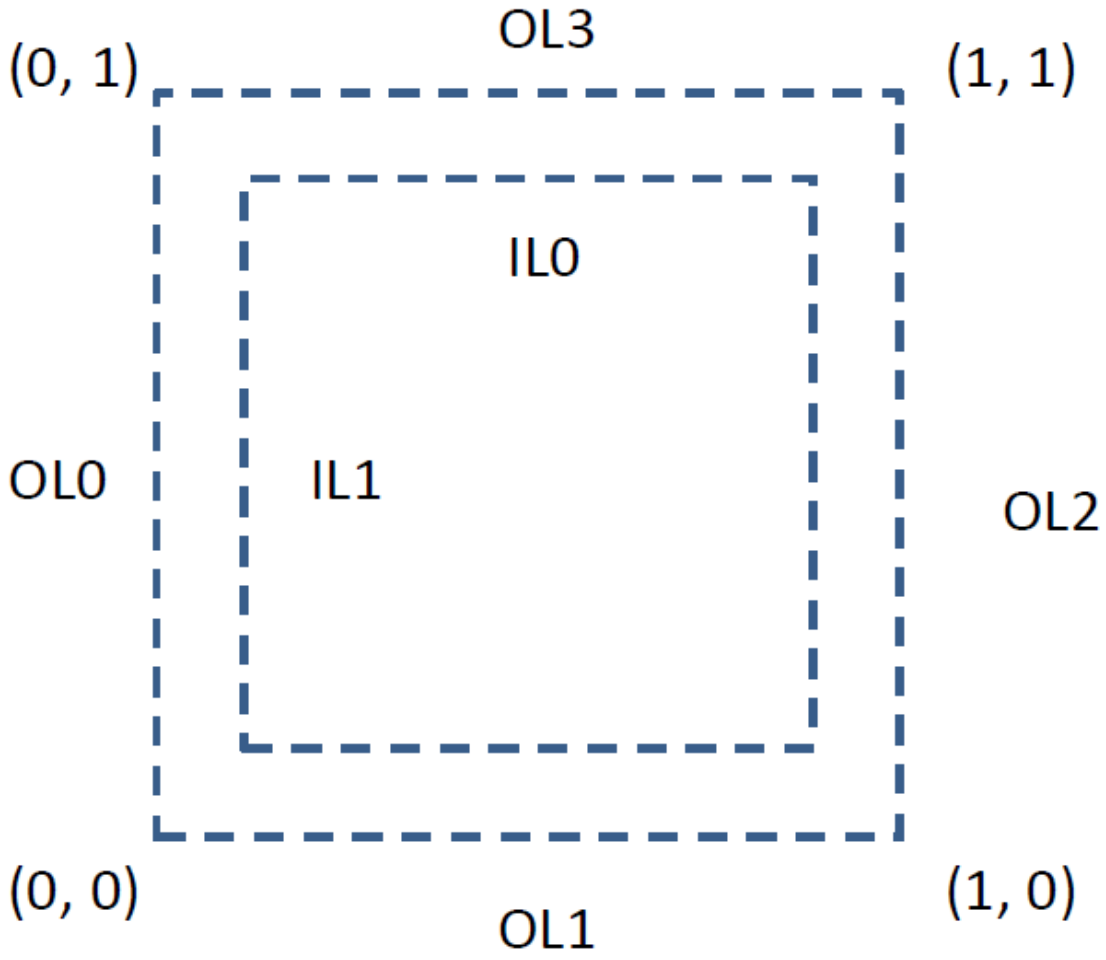
Типы разбиения (задаются в TES через layout):

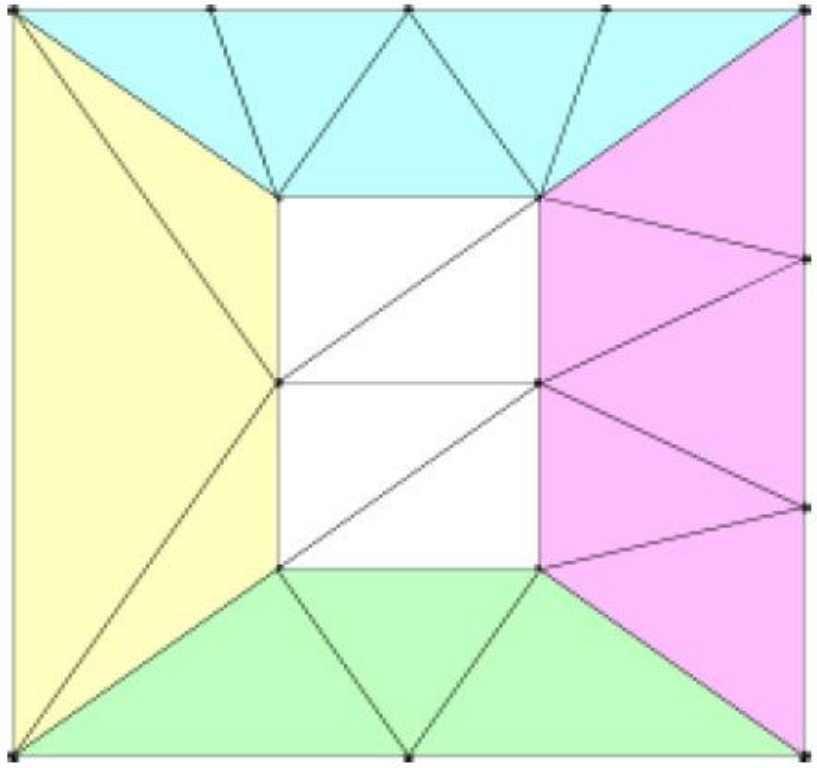
- triangles — для треугольных патчей
- quads — для четырёхугольных патчей
- isolines — для изолиний

Режимы разбиения:

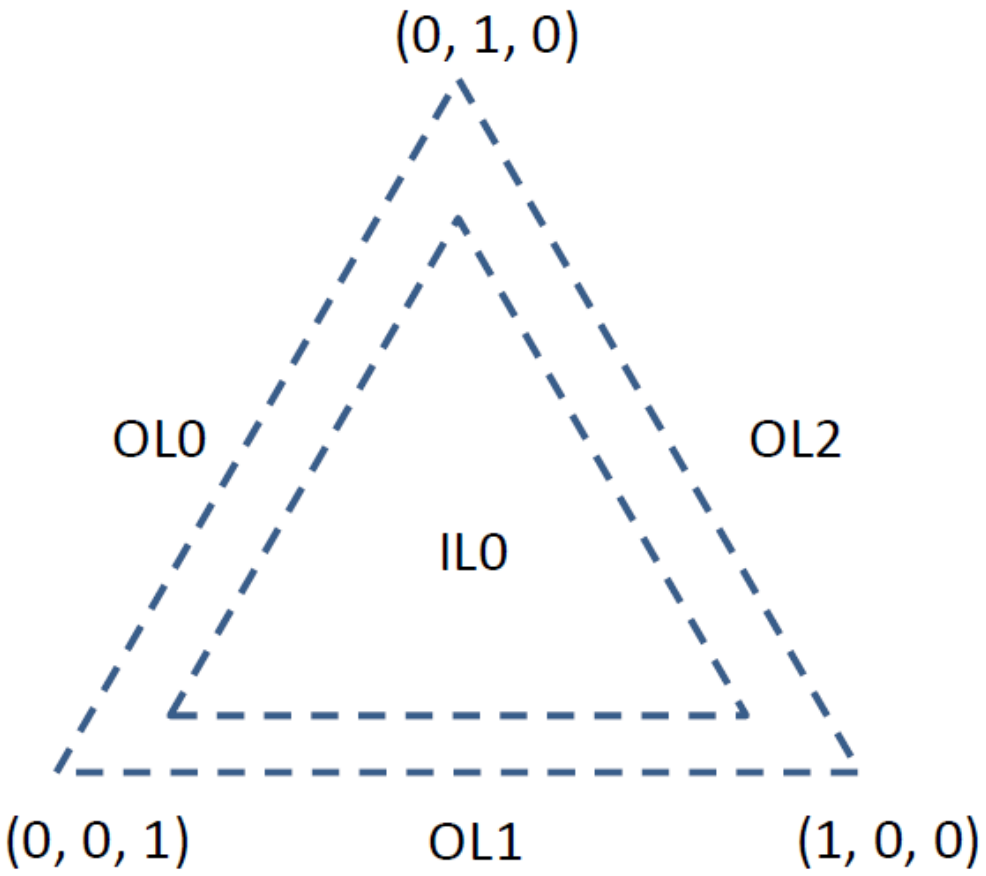
- equal_spacing — равномерное
- fractional_even_spacing — дробное чётное
- fractional_odd_spacing — дробное нечётное

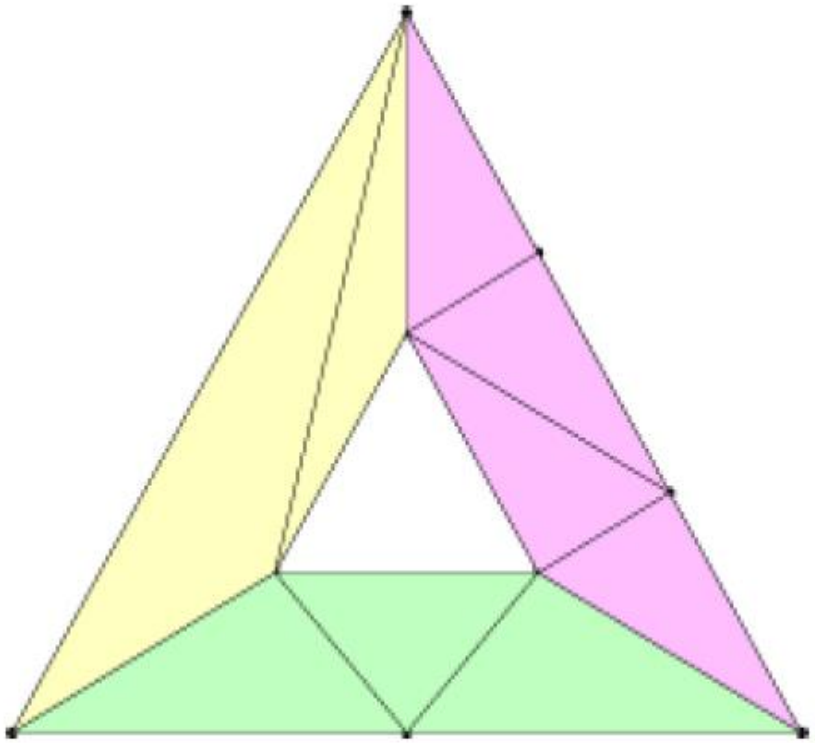
Тесселятор. Типы разбиений. Четырехугольники





Тесселятор. Типы разбиений. Треугольники





Tessellation Evaluation Shader (TES)

Что делает:

- Вызывается для каждой вершины, созданной тесселятором
- Вычисляет финальную позицию и атрибуты вершины

Входные данные:

```
in vec3 gl_TessCoord;       // координаты в патче (барицентрические для треугольников)
in float gl_TessLevelOuter[4];
in float gl_TessLevelInner[2];
```

Пример TES --- 1

```
#version 410 core

// ТЕССЕЛЯЦИЯ: треугольный патч, равномерное размещение, против часовой стрелки
layout(triangles, equal_spacing, ccw) in;

uniform mat4 modelViewProjectionMatrix;
uniform sampler2D heightmap;

in VertexES {              // от control shader (интерполированные для текущей вершины)
   vec3 position;
   vec2 texcoord;
   vec3 normal;
} vertes[];

out VertexFS {             // в геометрический шейдер или растеризатор
   vec3 position;
   vec2 texcoord;
   vec3 normal;
} vertfs;
```

Пример TES --- 2

```
// Функции интерполяции

vec2 interpolate2D(vec2 v0, vec2 v1, vec2 v2) {
   return gl_TessCoord.x * v0 + gl_TessCoord.y * v1 + gl_TessCoord.z * v2;
}

vec3 interpolate3D(vec3 v0, vec3 v1, vec3 v2) {
   return gl_TessCoord.x * v0 + gl_TessCoord.y * v1 + gl_TessCoord.z * v2;
}
```

Пример TES --- 3

```
void main() {
   // Интерполяция атрибутов
   vertfs.position = interpolate3D(vertes[0].position, vertes[1].position, vertes[2].position);
   vertfs.texcoord = interpolate2D(vertes[0].texcoord, vertes[1].texcoord, vertes[2].texcoord);
   vertfs.normal = normalize(interpolate3D(vertes[0].normal, vertes[1].normal, vertes[2].normal));

   // Вычисление позиции со смещением по карте высот
   vec4 pos = vec4(vertfs.position, 1.0);
   float height = texture(heightmap, vertfs.texcoord).r;
   pos.y += height * 0.25;

   gl_Position = modelViewProjectionMatrix * pos;
}
```

layout(triangles, equal_spacing, ccw) in --- 1

triangles — тип патча (обязательный)
Определяет геометрическую форму входного патча

Параметр	Описание	Количество вершин
triangles	Треугольный патч	3
quads	Четырёхугольный патч	4
isolines	Изолинии (линии)	4 (обычно)

Для треугольников:

- Тесселятор разбивает треугольник на более мелкие треугольники
- Работает с уровнями: gl_TessLevelOuter[0-2] и gl_TessLevelInner[0]

layout(triangles, equal_spacing, ccw) in --- 2

equal_spacing — способ распределения вершин
Определяет, как координаты тесселяции отображаются в фактические позиции вершин

Параметр	Описание	Визуальный эффект
equal_spacing	Равномерное расстояние вдоль рёбер	Вершины на равном расстоянии
fractional_even_spacing	Дробное чётное	Плавные переходы при изменении LOD
fractional_odd_spacing	Дробное нечётное	Плавные переходы (другой паттерн)

//Пример. Уровень тесселяции = 3.5

// equal_spacing: округляет до 4, но размещает равномерно
// fractional_even_spacing: размещает с переменным шагом (меньше прыжков)

layout(triangles, equal_spacing, ccw) in --- 3

ccw — порядок обхода вершин (ориентация)

Параметр	Описание
ccw	Counter-Clockwise (против часовой стрелки)
cw	Clockwise (по часовой стрелке)

Влияние

- Определяет, какая сторона полигона считается лицевой
- Влияет на back-face culling (отбраковку тыльных граней)
- Влияет на направление нормалей (если они не заданы явно)

Продвинутые тесселяционные схемы

PN-треугольники (Point-Normal Triangles)
Строит кубический патч Безье для каждого треугольника, используя позиции и нормали вершин.

$$b111 = E * 3/2 + V * 1/2$$
$$E = (b210 + b120 + b012 + b012 + b102 + b201) / 6$$
$$V = (b003 + b030 + b300) / 3$$

Результат: плавное сглаживание геометрии без дополнительных данных.

Фонг-тесселяция
Вместо интерполяции позиций, интерполируются проекции вершин на касательные плоскости, заданные парой (P, n).

$$P\_phong = P0* u + P1* v + P2* w$$

Где Pi* — проекции точки P на плоскости, заданные нормалями.
Преимущество: сохраняет резкие края там, где это нужно.

Адаптивный LOD и эвристики

Разбиение по расстоянию до камеры:

```
float dist = distance(cameraPosition, patchCenter);
float level = max(1.0, 64.0 / dist);
```

Разбиение по ориентации патча:

```
float orientation = abs(dot(normal, viewDir));
float level = mix(minLevel, maxLevel, orientation);
```

Адаптивное разбиение в экранном пространстве:
Оценивает проекционный размер патча на экране и назначает уровень тесселяции.

Геометрические шейдеры (Geometry Shaders) Возможности и ограничения

История: появились в DirectX10, затем в OpenGL 3.2.

На вход: примитивы любого типа
На выход: только points, line_strip, triangle_strip

Что могут:

- Изменять топологию
- Породить больше или меньше вершин, чем получили (growing geometry)
- Полностью удалить примитив

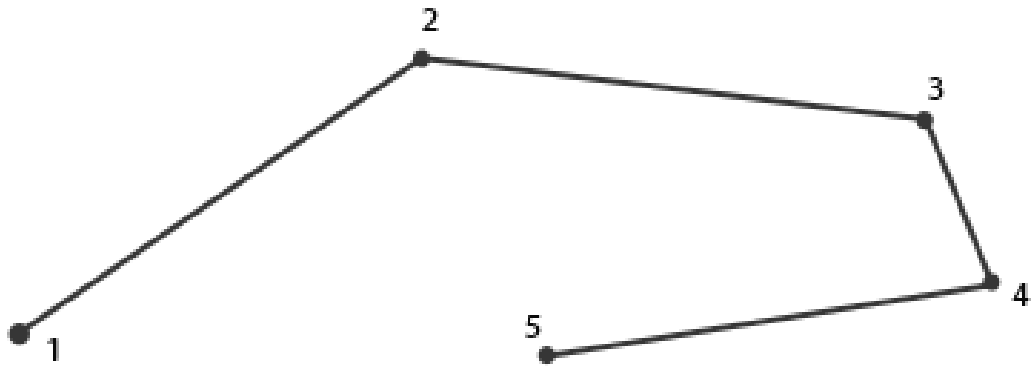
Ограничения:

- Максимальное количество выходных вершин ограничено (зависит от реализации, обычно 256-1024)

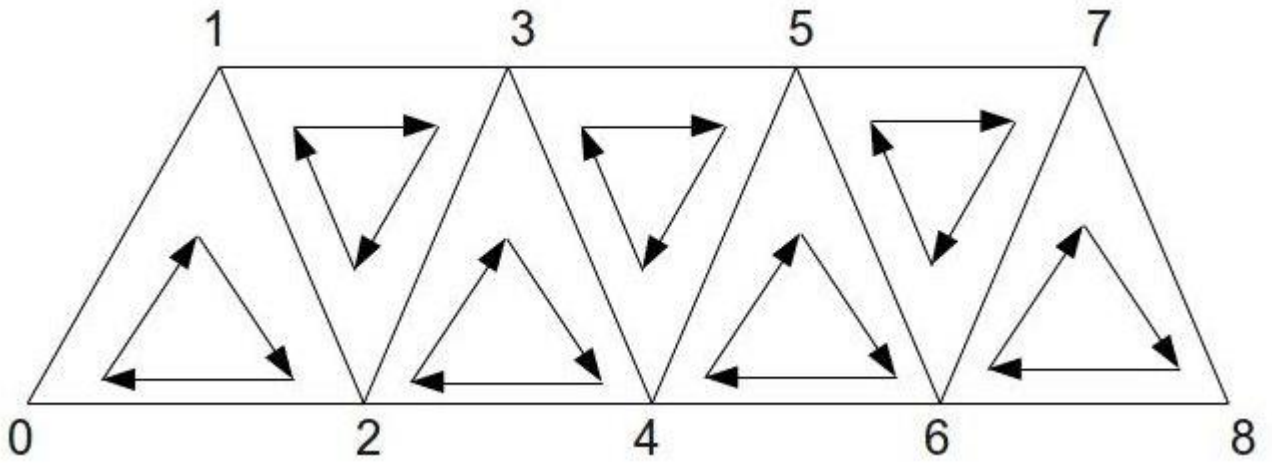
Структура геометрического шейдера

```
#version 330 core

// Входной тип примитива
layout (points) in;         // points, lines, triangles, lines_adjacency, triangles_adjacency

// Выходной тип и максимальное число вершин
layout (triangle_strip, max_vertices = 5) out;

void main() {
    // Генерация вершин
    gl_Position = ...;
    EmitVertex();           // добавляет вершину в текущий стрип

    gl_Position = ...;
    EmitVertex();

    EndPrimitive();         // завершает текущий примитив
}
```

Line Strip



Стрип треугольников



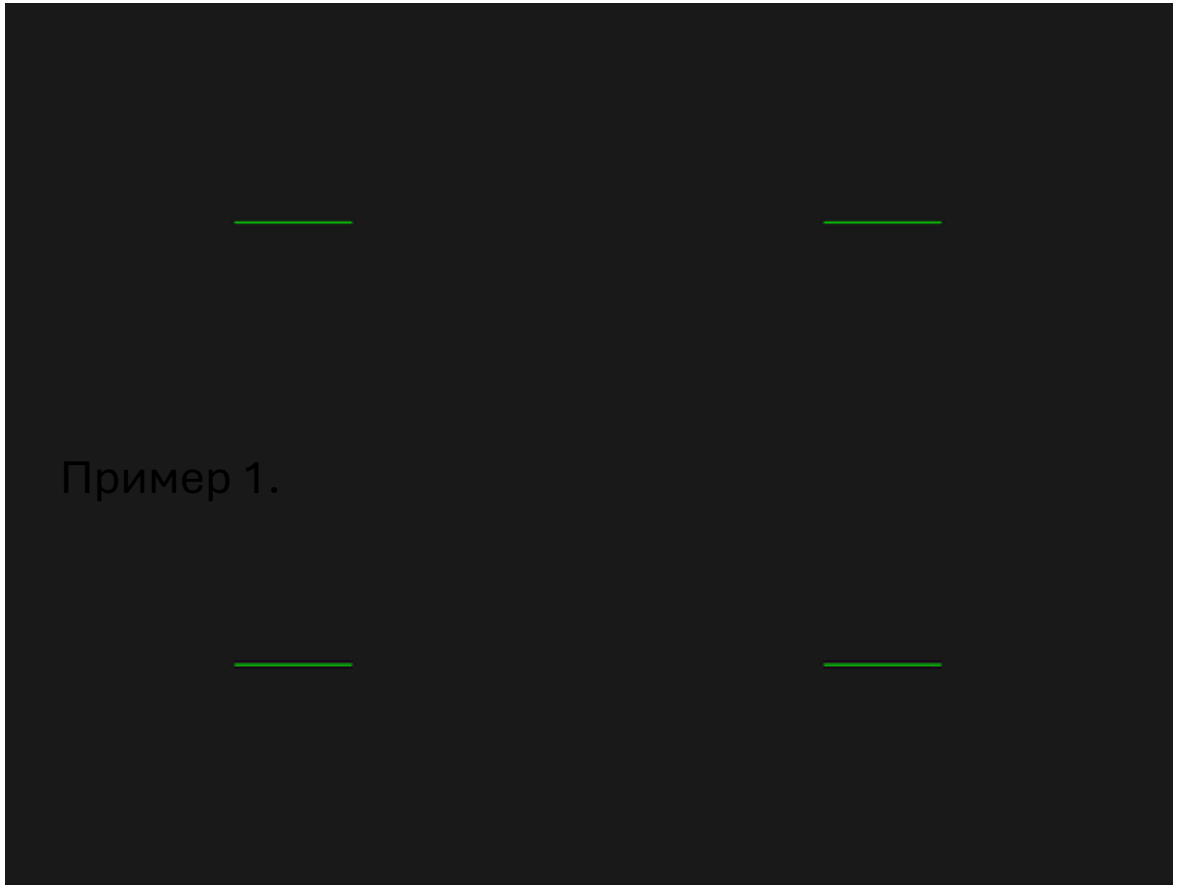
Важное свойство касательно порядка внутри треугольника — порядок обратен у нечётных треугольников.

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

Пример 1. Простейший геометрический шейдер

```
//Превращает каждую точку в горизонтальную линию

#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();
}
```

Пример 1. Результат



Получили такие результаты, выполнив всего лишь один вызов отрисовки:
glDrawArrays(GL_POINTS, 0, 4);

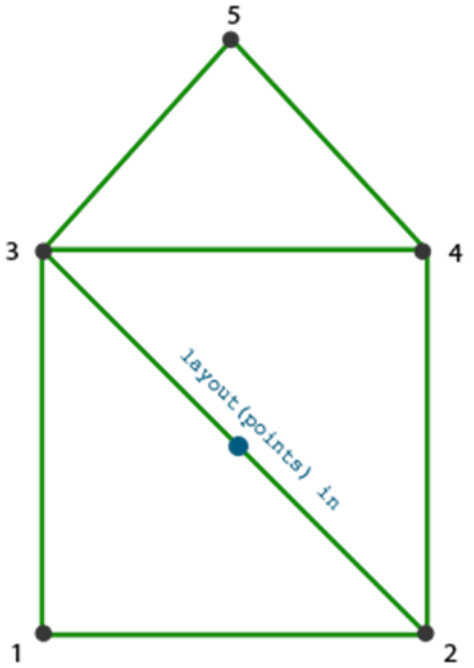
Компиляция и линковка

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

Пример 2. Строим домики из точек

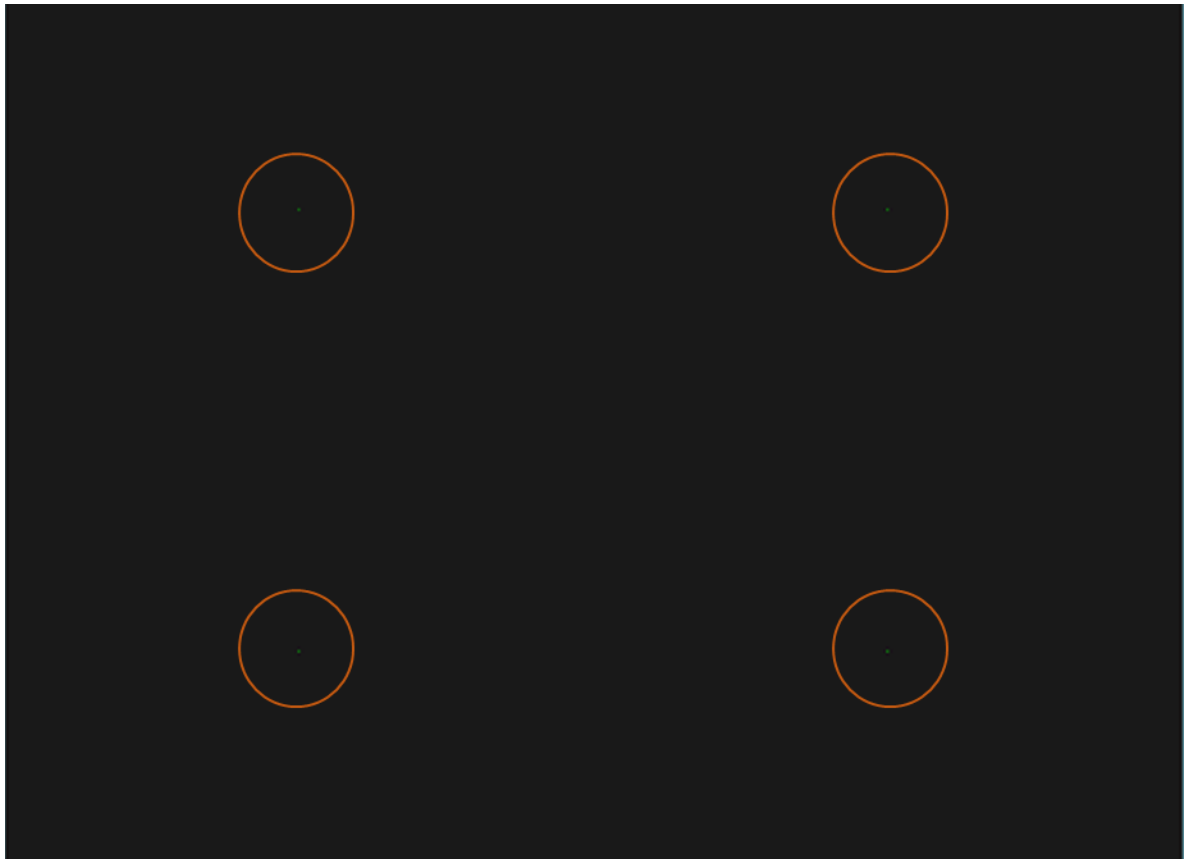
```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;
void build_house(vec4 position) {
   // Квадратная основа (4 вершины)
   gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);   // bottom-left
   EmitVertex();
   gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);   // bottom-right
   EmitVertex();
   gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);   // top-left
   EmitVertex();
   gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);   // top-right
   EmitVertex();
   // Крыша (5-я вершина)
   gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);   // roof top
   EmitVertex();
   EndPrimitive();
}
void main() {
   build_house(gl_in[0].gl_Position);
}
```

На входе — 4 точки, на выходе — по 5 вершин на каждую точку (квадрат + крыша):

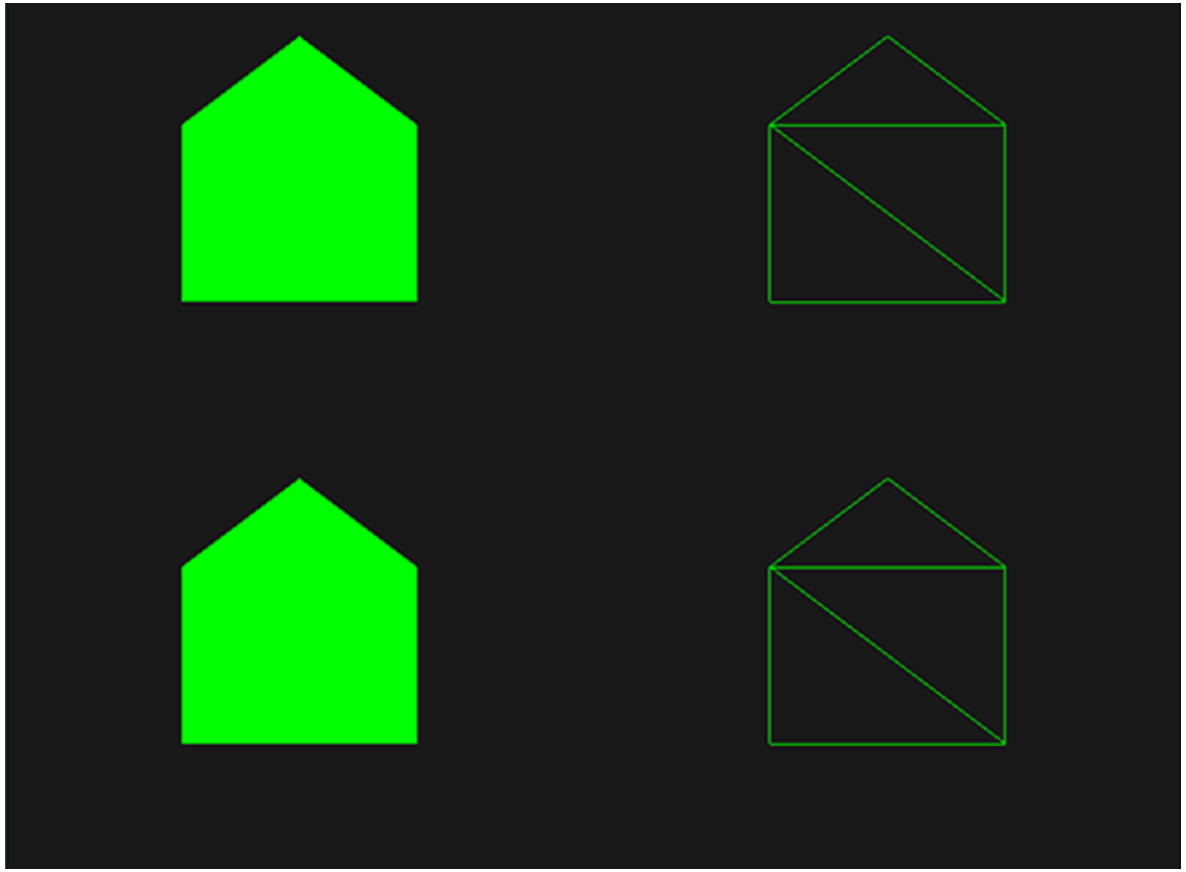


Входные данные

```
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
};
```



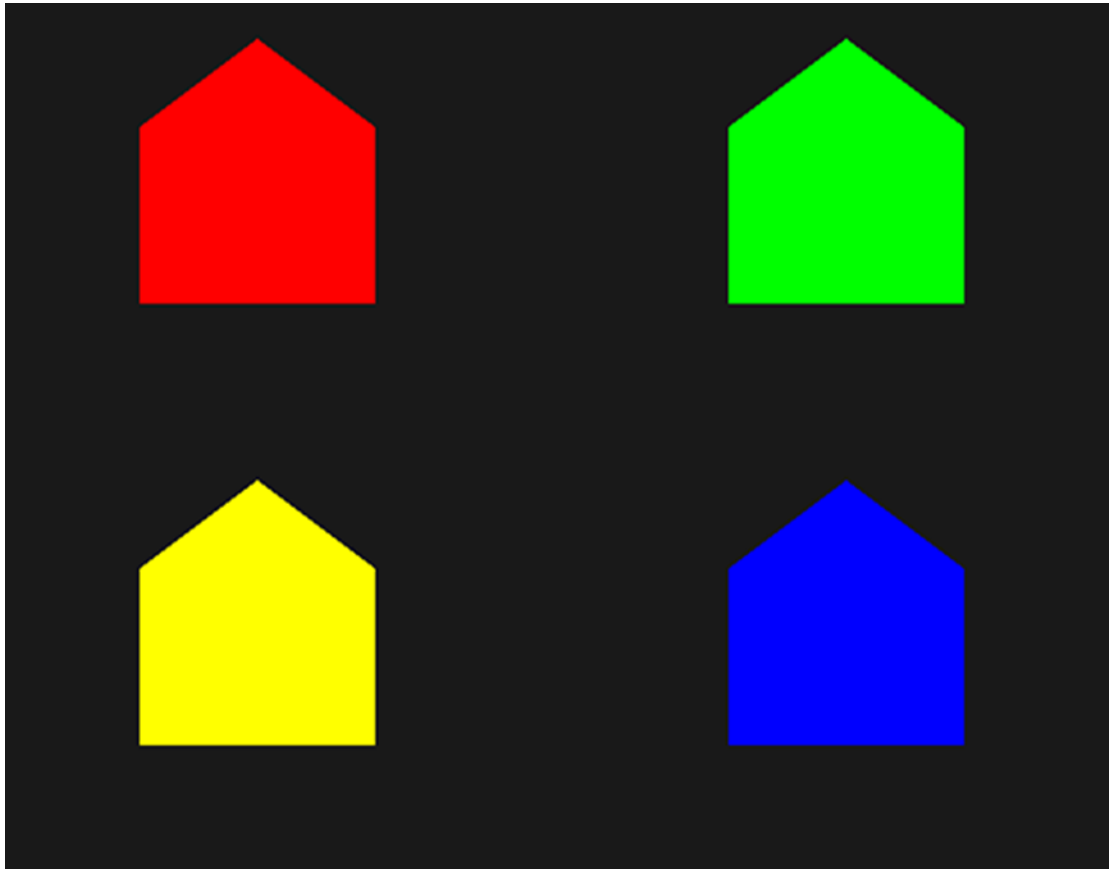
Пример 2. Результат

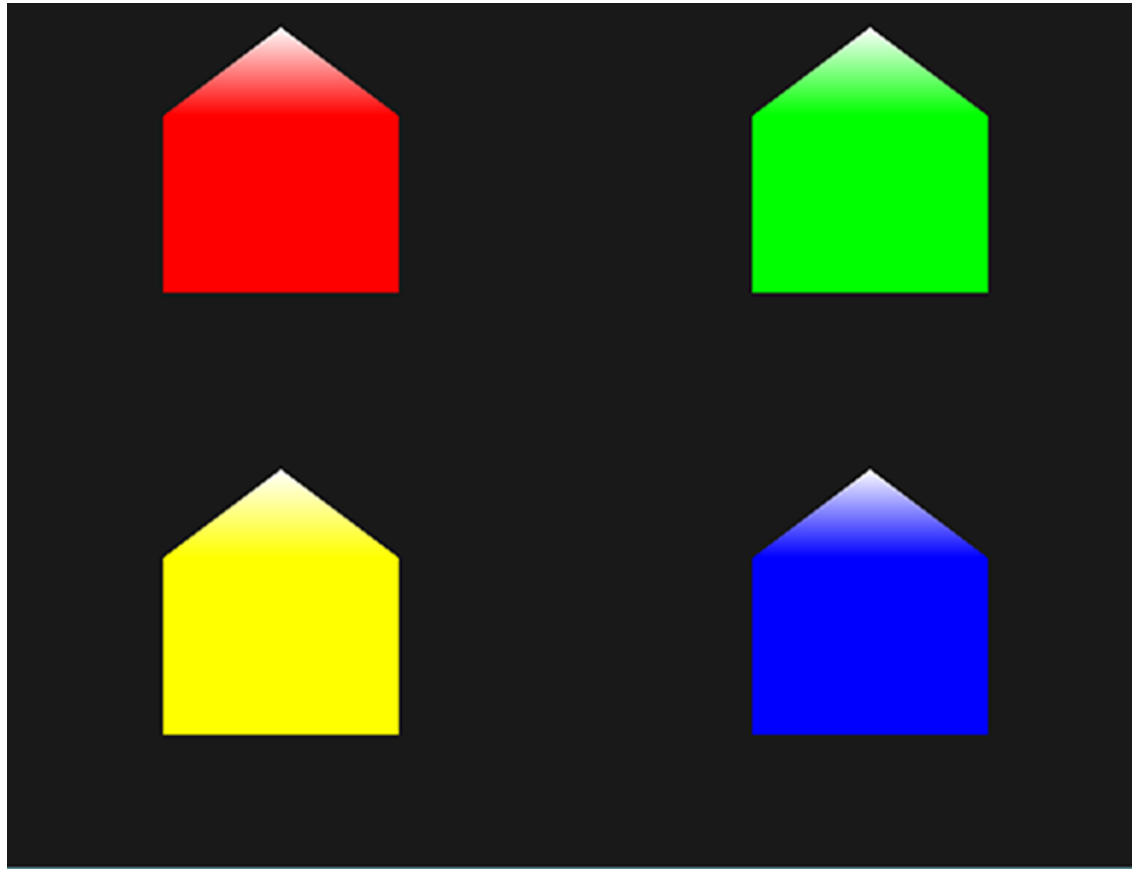


Домики с передачей цвета

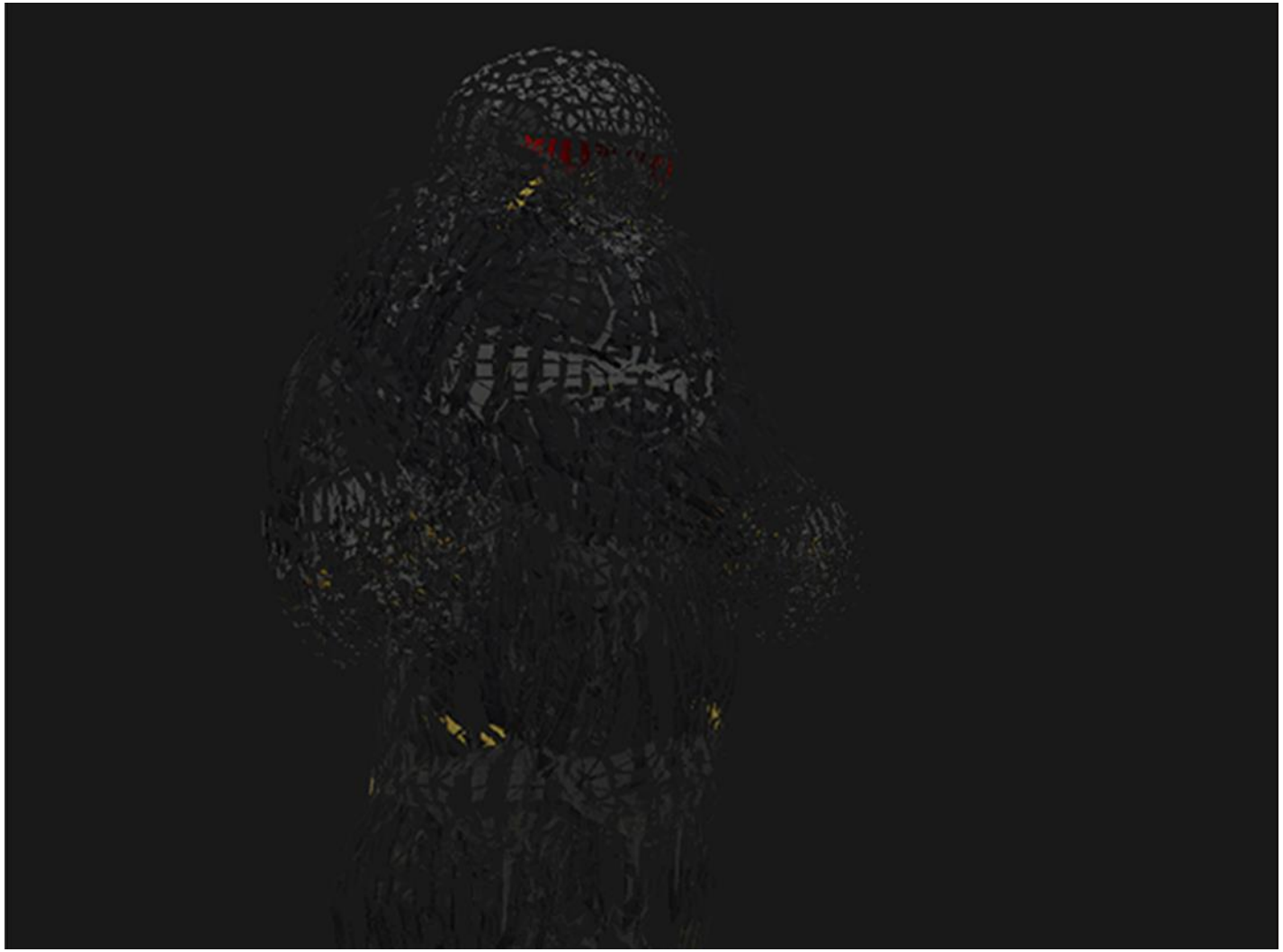
```
// Вершинный шейдер
out VS_OUT {
    vec3 color;
} vs_out;
// Геометрический шейдер
in VS_OUT {
    vec3 color;
} gs_in[];
out vec3 fColor;

void main() {
    fColor = gs_in[0].color;      // цвет для стен
    // ... вершины основы ...
    EmitVertex();

    fColor = vec3(1.0, 1.0, 1.0); // белый снег для крыши
    gl_Position = position + vec4(0.0, 0.4, 0.0, 0.0);
    EmitVertex();
    EndPrimitive();
}
```

Пример 2. Разноцветные домики





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

Пример 3. Взрыв объекта --- 1

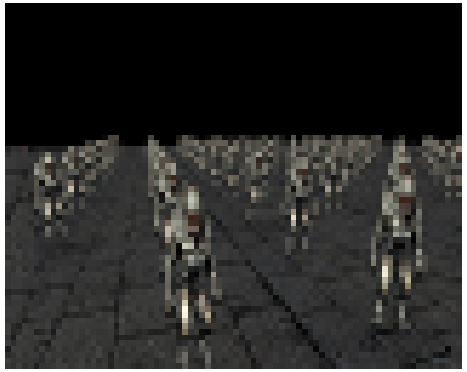
Вычисляем нормаль треугольника и смещаем вершины вдоль неё с течением времени

```
uniform float time;   // время от приложения

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));
}

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);
}
```

Пример 3. Взрыв объекта --- 2

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

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

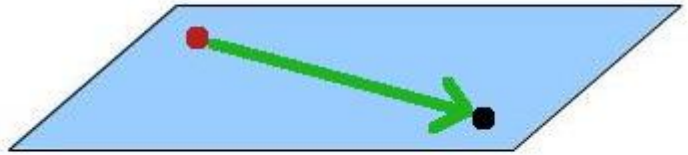
Реализация метода billboarding



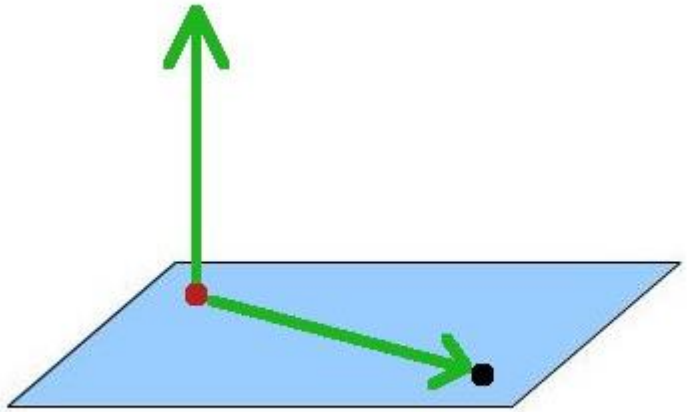
Billboarding - прямоугольник, который всегда направлен в камеру.

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

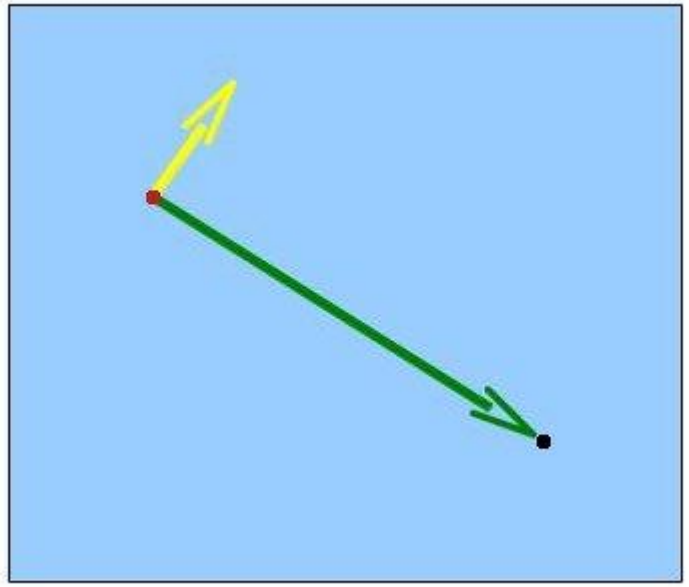
Как направить billboard на камеру?



вектор из позиции billboard в камеру:



вектор (0,1,0):



желтый вектор — результат векторного произведения

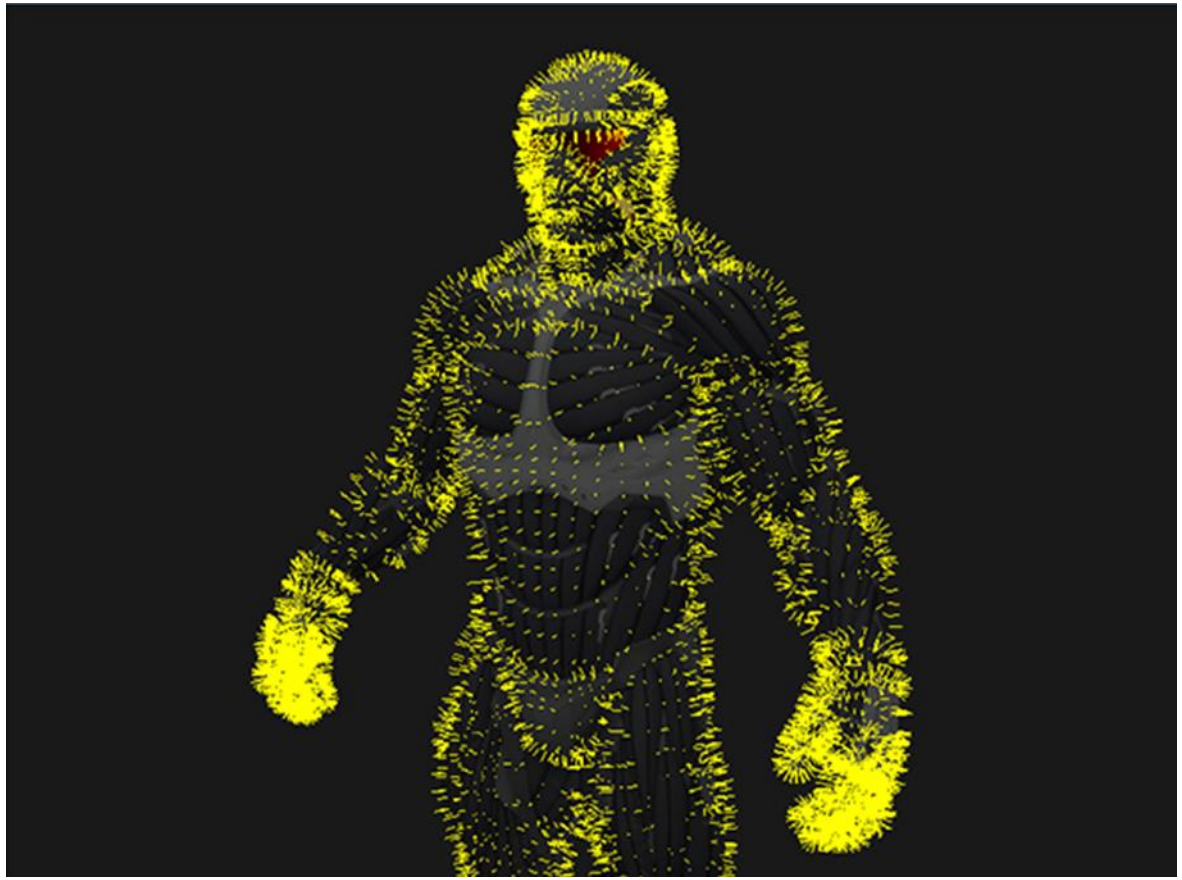
Пример 4. Billboarding (спрайты, всегда повёрнутые к камере) --- 1

```
uniform mat4 viewMatrix;   // матрица вида
uniform mat4 projMatrix;   // матрица проекции

void main() {
   // Получаем базовую позицию в пространстве вида
   vec4 center = viewMatrix * gl_in[0].gl_Position;

   // Векторы right и up в пространстве вида
   vec3 right = vec3(1.0, 0.0, 0.0);
   vec3 up = vec3(0.0, 1.0, 0.0);

   // Размер биллборда
   float width = 0.5;
   float height = 0.5;
```

Пример 4. Billboarding (спрайты, всегда повёрнутые к камере) --- 2

```
    // Строим 4 вершины
    gl_Position = projMatrix * (center + vec4(-width, -height, 0.0, 0.0));
    EmitVertex();
    gl_Position = projMatrix * (center + vec4( width, -height, 0.0, 0.0));
    EmitVertex();
    gl_Position = projMatrix * (center + vec4(-width,  height, 0.0, 0.0));
    EmitVertex();
    gl_Position = projMatrix * (center + vec4( width,  height, 0.0, 0.0));
    EmitVertex();
    EndPrimitive();
}
```

Визуализация нормалей



Пример 5. Визуализация нормалей

```
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

void main() {
   vec3 normal = GetNormal();
   vec3 center = (gl_in[0].gl_Position.xyz +  gl_in[1].gl_Position.xyz + gl_in[2].gl_Position.xyz) / 3.0;

   // Рисуем исходный треугольник
   for (int i = 0; i < 3; i++) {
      gl_Position = gl_in[i].gl_Position;
      EmitVertex();
   }
   EndPrimitive();

   // Рисуем нормаль (линия из центра)
   gl_Position = vec4(center, 1.0);
   EmitVertex();
   gl_Position = vec4(center + normal * 0.3, 1.0);
   EmitVertex();
   EndPrimitive();
}
```

Характеристики

Характеристика	Тесселяция	Геометрический шейдер
Основная цель	Увеличение детализации	Изменение топологии/порождение
Масштабирование	Квадратичное/кубическое	Линейное (max_vertices)
Производительность	Очень высокая (аппаратный тесселятор)	Средняя (может быть узким местом)
Доступ к соседям	Да (весь патч)	Да (все вершины примитива)
Управление LOD	Встроенное (уровни тесселяции)	Ручное
Типичное применение	Ландшафты, subdivision surfaces, сглаживание	Спецэффекты, трава, billboard, отладка

Когда что использовать?

Выбираем **тесселяцию**, если:

- Нужно динамически увеличивать детализацию
- Геометрия имеет регулярную структуру (ландшафты, патчи)
- Важна производительность и аппаратная поддержка
- Нужен плавный LOD без загрузки CPU

Выбираем **геометрические шейдеры**, если:

- Нужно менять топологию (точки → треугольники, и т.д.)
- Требуются спецэффекты (взрывы, частицы)
- Нужно отладочно визуализировать нормали
- Объём геометрии не очень большой

Ключевые выводы

Тесселяция — переносит вычисление LOD и детализации на GPU, избавляя CPU от тяжёлой работы.

Геометрические шейдеры — позволяют динамически изменять топологию, создавать процедурную геометрию и спецэффекты.

Обе стадии работают на GPU — это главное преимущество: тысячи потоков параллельно обрабатывают геометрию без загрузки шины PCIe.

Практические рекомендации

// Рекомендуемая архитектура для продвинутого рендерера
CPU:
- Загружает базовые меши (патчи/точки)
- Управляет камерой и временем
- Отправляет uniform-переменные (уровни LOD, матрицы)

GPU:
- VS : Мировые координаты + нормали + UV
- TCS: вычисляет уровни тесселяции (distance-based LOD)
- Тесселятор: аппаратное разбиение
- TES: интерполяция + displacement mapping
- Геометрический шейдер: спецэффекты (опционально)
- Фрагментный шейдер: материалы и освещение

Современные альтернативы

В современных API (Vulkan, DirectX 12 Ultimate) появились
Mesh Shaders — более гибкая альтернатива,
объединяющая возможности тесселяции и геометрических шейдеров.

Однако **тесселяция остаётся актуальной благодаря аппаратной оптимизации**.

Примечание для современных движков (Unreal Engine 5, Unity 2022+):

- Геометрический шейдер почти не используется (медленный, плохо параллелится)
- Mesh Shaders (NVIDIA RTX) заменяют VS + TCS + TES для некоторых задач
- Compute Shaders иногда делают тесселяцию на вычислительных шейдерах
- Но классический конвейер VS→TCS→TES→FS всё ещё актуален для террейнов и водных поверхностей

Вопросы для самопроверки

Почему тесселяция эффективнее CPU-генерации геометрии?

В чём разница между TCS и TES?

Какие типы примитивов может порождать геометрический шейдер?

Как реализовать адаптивный LOD по расстоянию?

Когда геометрический шейдер может быть неэффективен?

Вопросы с ответами

Почему тесселяция эффективнее CPU-генерации геометрии?
Ответ: Работает на GPU, не нагружает шину PCIe, аппаратно оптимизирована.

В чём разница между TCS и TES?
Ответ: TCS задаёт сколько разбивать, TES — как вычислить каждую новую вершину.

Какие типы примитивов может порождать геометрический шейдер?
Ответ: Только points, line_strip, triangle_strip.

Как реализовать адаптивный LOD по расстоянию?
Ответ: В TCS вычислить расстояние до камеры и задать уровни тесселяции пропорционально 1/distance.

Когда геометрический шейдер может быть неэффективен?
Ответ: При порождении огромного количества вершин (лучше использовать тесселяцию или compute shaders).