CS Space
Клуб технологий и науки

Вычисления на видеока

Лекция 11 - **software rasterization**

- Vertex & Fragment shaders
- Растеризация в OpenGL/Vulkan
- Алгоритм **Брезенхэма**
- Проект “**Larrabee**” (Intel)
- Проект “**cudaraster**” (Nvidia)

@UnicornGlade
@PolarNick239
polarnick239@gmail.com
Николай Полярный

Глава 1: Растеризация

Vertex + Fragment shaders
Алгоритм Брезенхэма

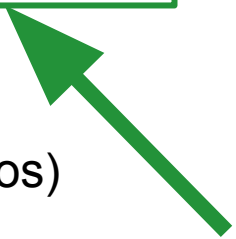
История GPU (см. первую лекцию)

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители с **Fixed pipeline**

2000-2002 - в DirectX 8.0 и OpenGL появляются программируемые ***шейдеры***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

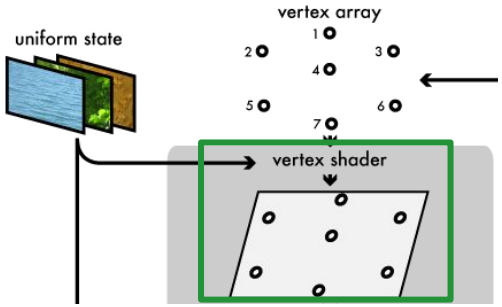


2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)

2012 - Compute Shaders в OpenGL

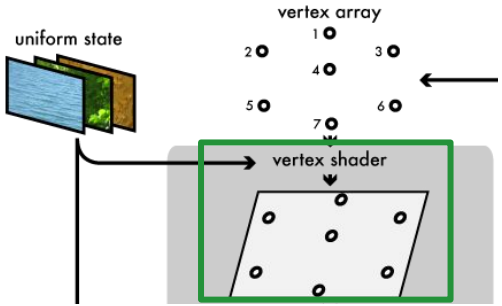
2016 - Vulkan

Программируемый вершинный шейдер
Vertex Shader



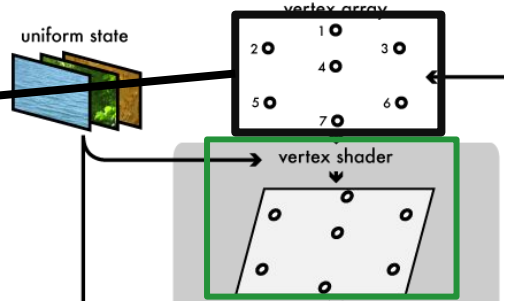
GLSL (Graphics Library Shading Language)

```glsl
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



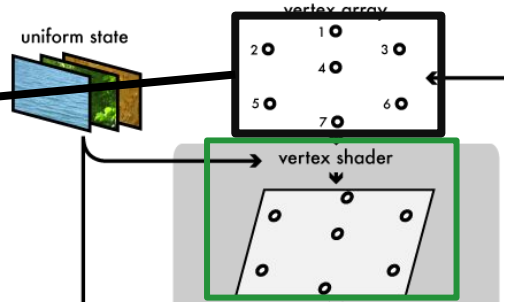
GLSL (Graphics Library Shading Language)

```glsl
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



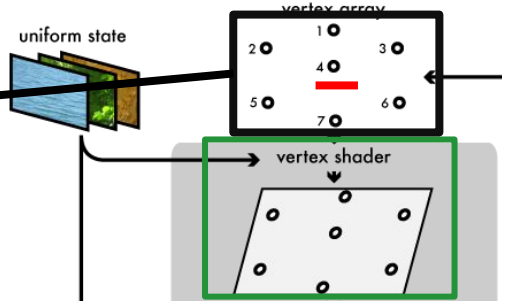
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



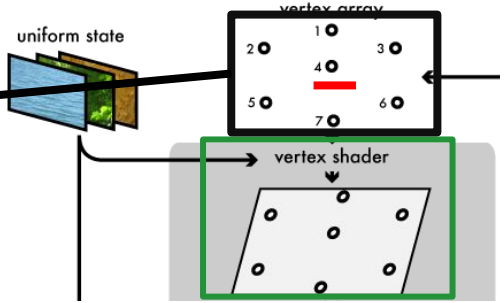
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



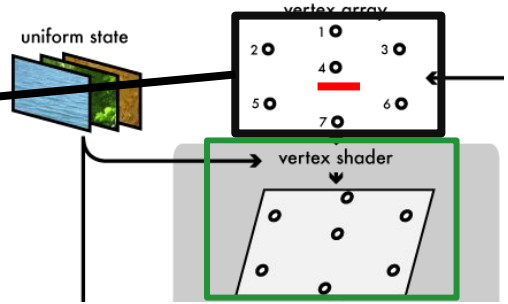
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



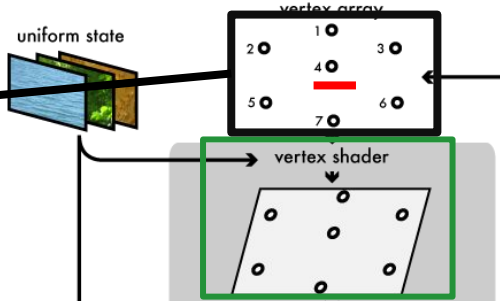
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



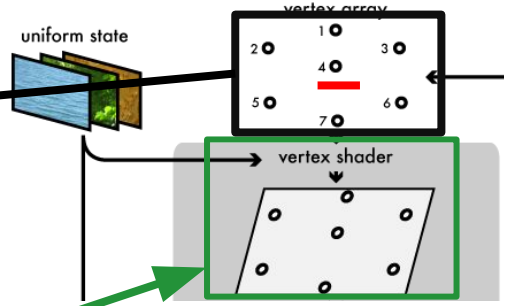
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```



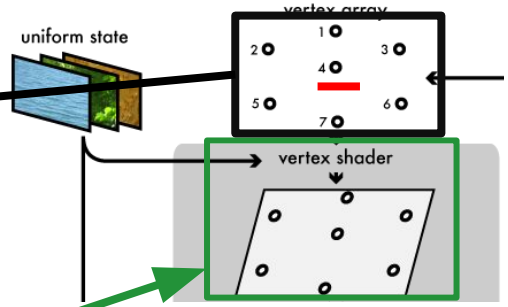
глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

Гдеееее я **на экране**?

GLSL (Graphics Library Shading Language)

```
#version 330 core

layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates

uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;

out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;
    gl_Position = uProj * uView * wp; // Clip-space position
}
```



глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

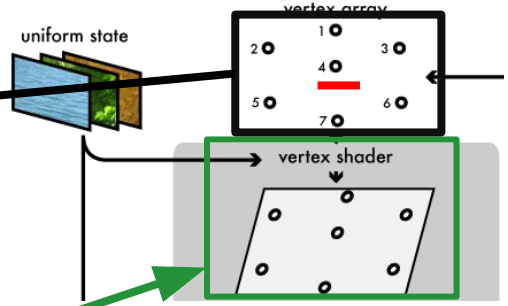
Гдеееe я **на экране**?

А где объявлен gl_Position?

```
#version 330 core
```

GLSL (Graphics Library Shading Language)

```
layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates
```




```
uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;
```

глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

```
out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```

Name

gl_Position — contains the position of the current vertex

Description

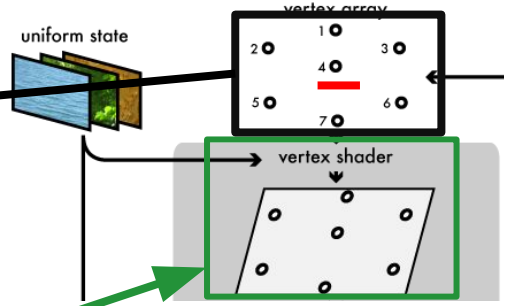
In the vertex, tessellation evaluation and geometry languages, a single global instance of the *gl_PerVertex* named block is available and its *gl_Position* member is an output that receives the homogeneous vertex position. It may be written at any time during shader execution. The value written to *gl_Position* will be used by primitive assembly, clipping, culling and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred.

А где объявлен gl_Position?

```glsl
#version 330 core
```

GLSL (Graphics Library Shading Language)

```glsl
layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates
```




```glsl
uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;
```

глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

```glsl
out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;

// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```

Что это такое?

Name

gl_Position — contains the position of the current vertex

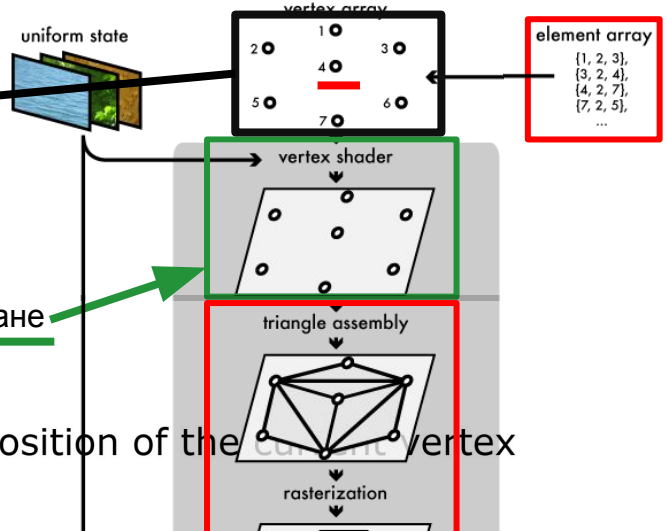
Description

In the vertex, tessellation evaluation and geometry languages, a single global instance of the *gl_PerVertex* named block is available and its *gl_Position* member is an output that receives the homogeneous vertex position. It may be written at any time during shader execution. The value written to *gl_Position* will be used by primitive assembly, clipping, culling and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred.

```glsl
#version 330 core
```

GLSL (Graphics Library Shading Language)

```glsl
layout(location = 0) in vec3 aPos;      // Model-space position
layout(location = 1) in vec3 aNormal;   // Model-space normal
layout(location = 2) in vec2 aUV;       // Texture coordinates
```




```glsl
uniform mat4 uModel;
uniform mat4 uView;
uniform mat4 uProj;
```

глобальные переменные - матрицы преобразования:

1) точка объекта -> точка в мире
2) точка в мире -> точка в системе координат камеры
3) точка в системе координат камеры -> проекция на экране

```glsl
out VS_OUT {
    vec3 worldPos;   // World-space position
    vec3 worldNorm;  // World-space normal
    vec2 uv;         // UV to interpolate
} v;
```

```glsl
// NB: код является галлюцинацией LLM
void main() {
    vec4 wp = uModel * vec4(aPos, 1.0);
    v.worldPos = wp.xyz;

    // Normal matrix = inverse-transpose of model's upper-left 3x3
    mat3 normalMat = transpose(inverse(mat3(uModel)));
    v.worldNorm = normalize(normalMat * aNormal);

    v.uv = aUV;

    gl_Position = uProj * uView * wp; // Clip-space position
}
```

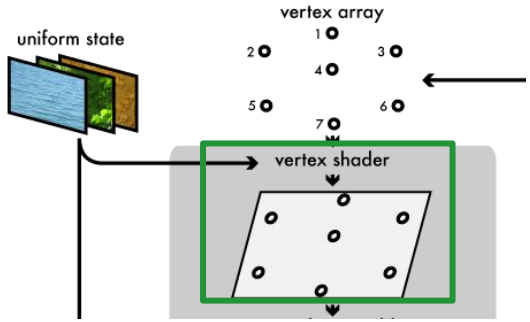
Name

gl_Position — contains the position of the current vertex

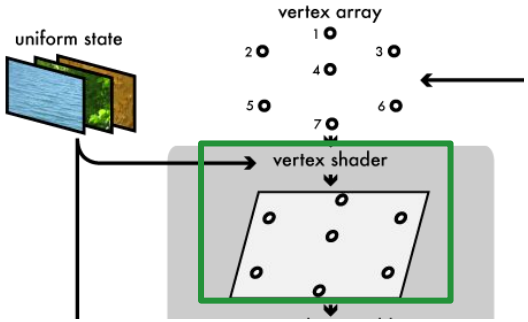
Description

In the vertex, tessellation evaluation and geometry lan[illegible]ngle global instance of the *gl_PerVertex* named block is available [illegible]*osition* member is an output that receives the homogeneous vertex position. It may be written at any time during shader execution. The value written to *gl_Position* will be used by primitive assembly, clipping, culling and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred.

Свойства (атрибуты) каждой вершины, интерполируются так же как gl_Position



A

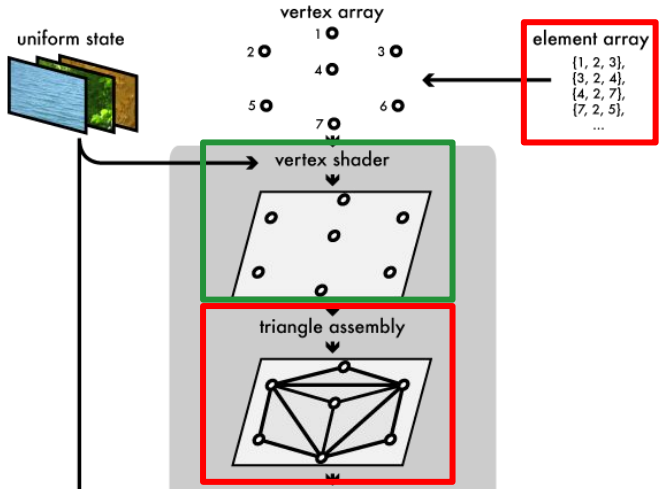


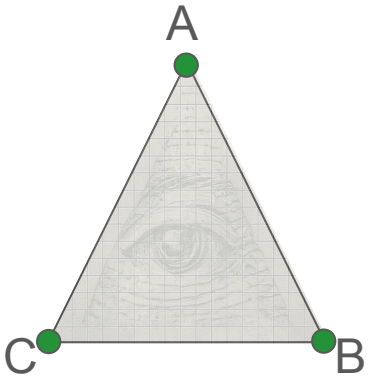
A

D

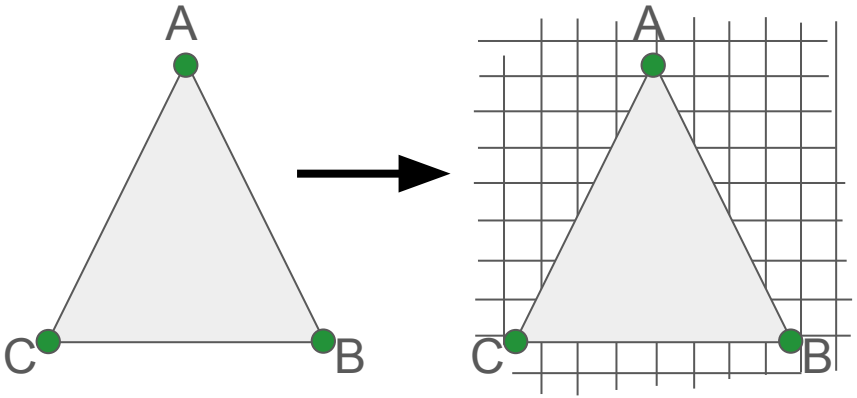
C

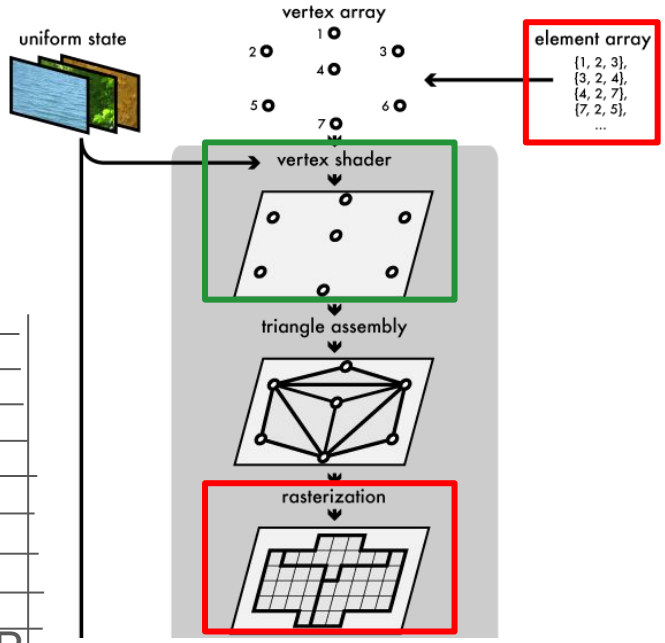
B


vertex array
uniform state
element array
{1, 2, 3},
{3, 2, 4},
{4, 2, 7},
{7, 2, 5},
...
vertex shader
triangle assembly

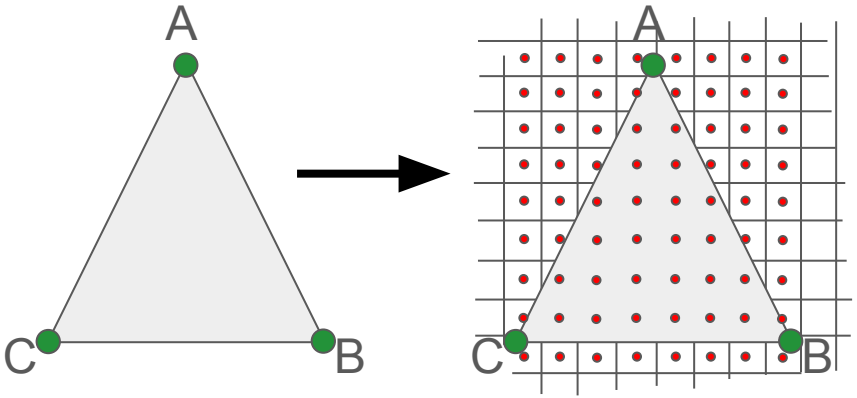

A
C
B

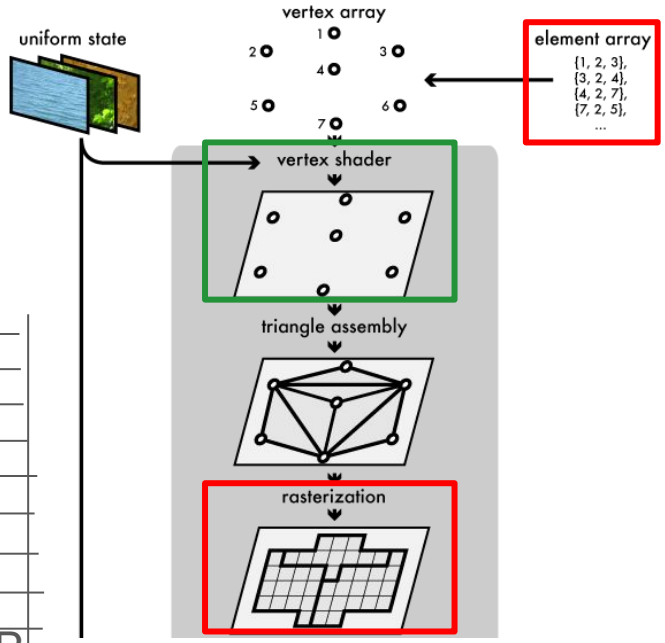
Растеризация



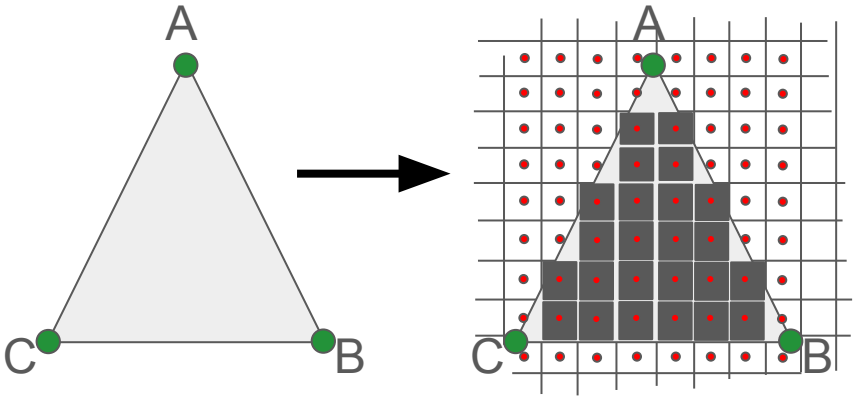


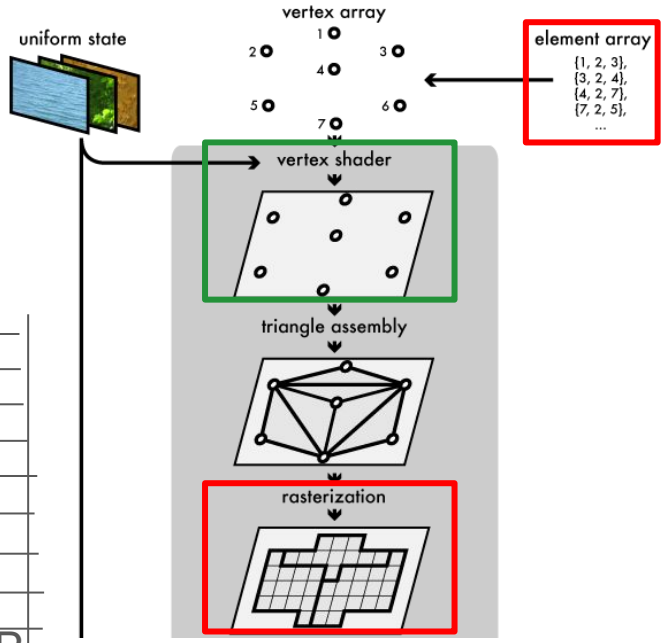
Растеризация



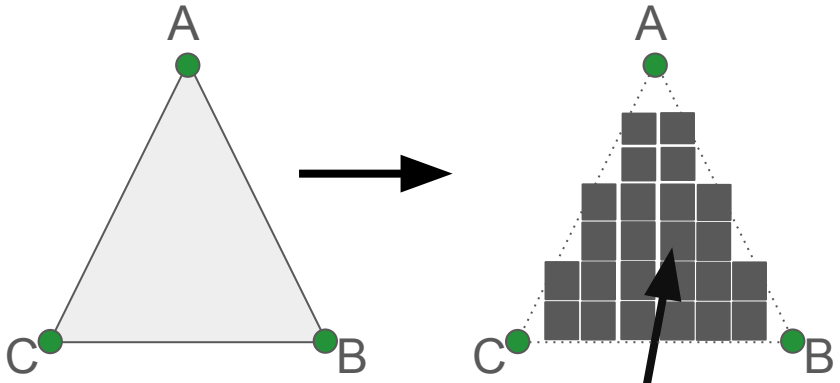


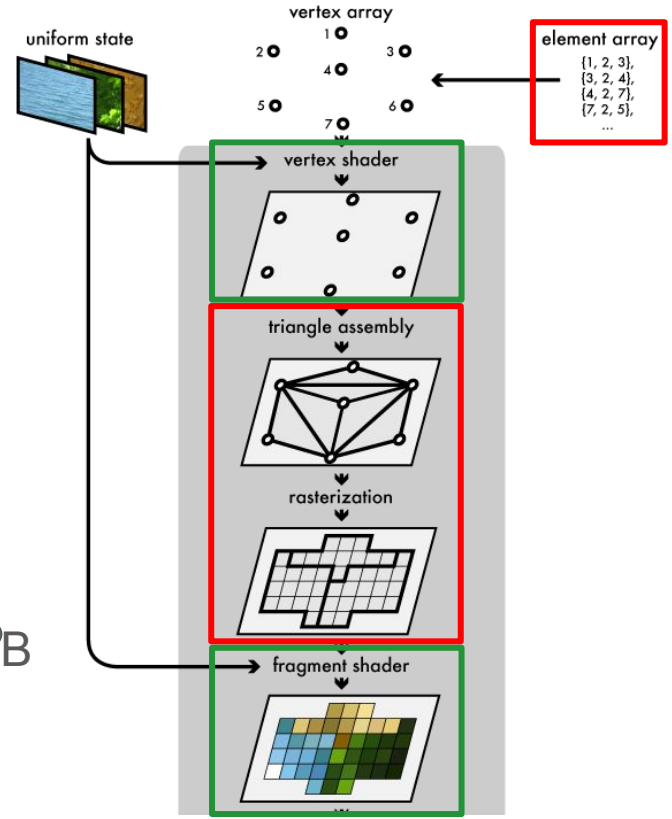
Растеризация





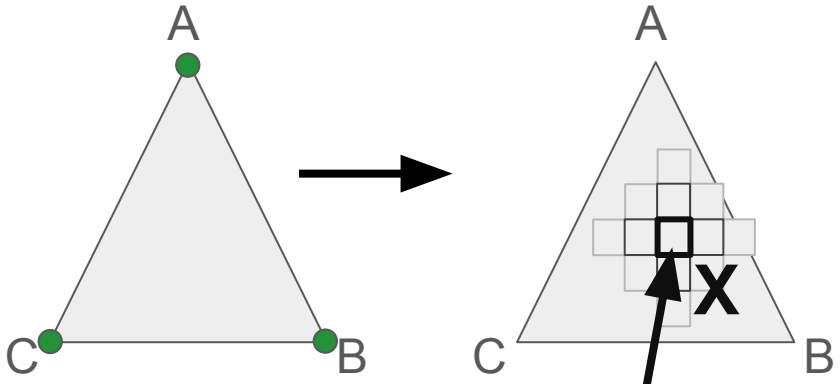
Растеризация

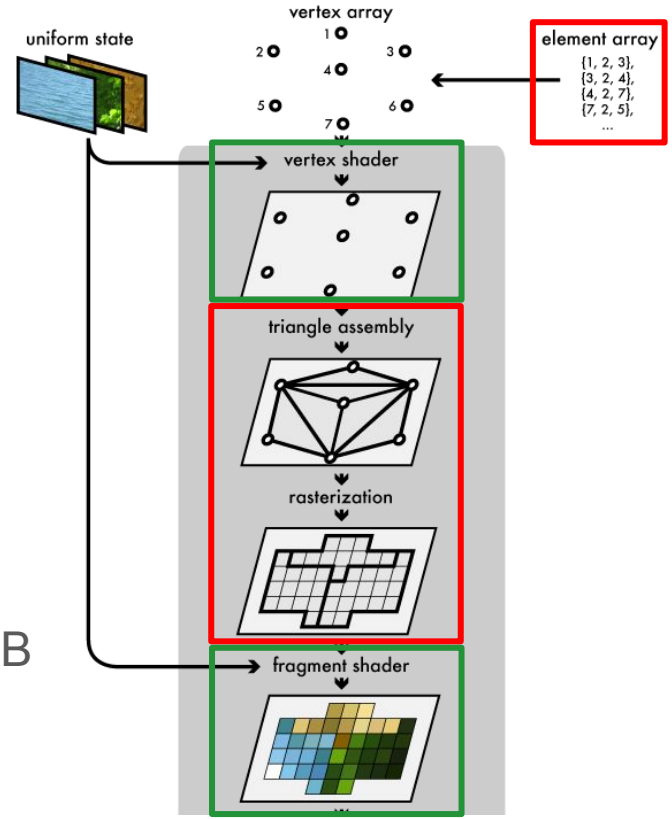




Фрагментный шейдер
выполняется **в каждом фрагменте**

Растеризация

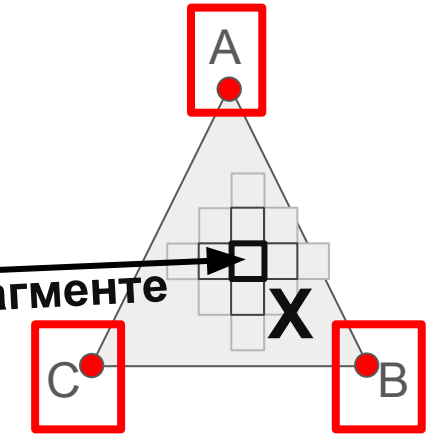




Фрагментный шейдер
выполняется **в каждом фрагменте**

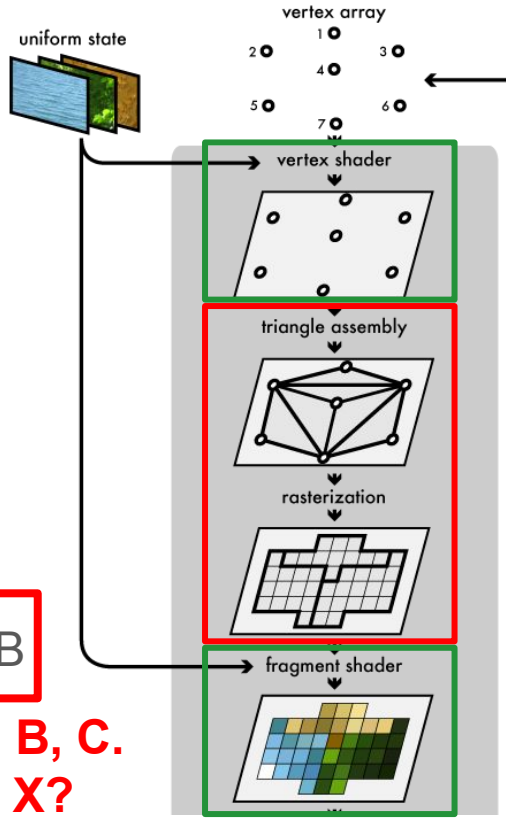
Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse (Lambert)

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Specular (Phong)

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```



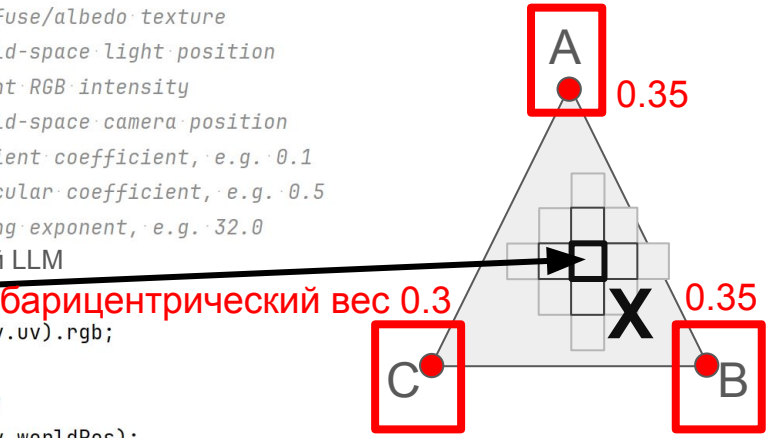
в каждом фрагменте

Известны свойства A, B, C.
Как найти атрибуты X?

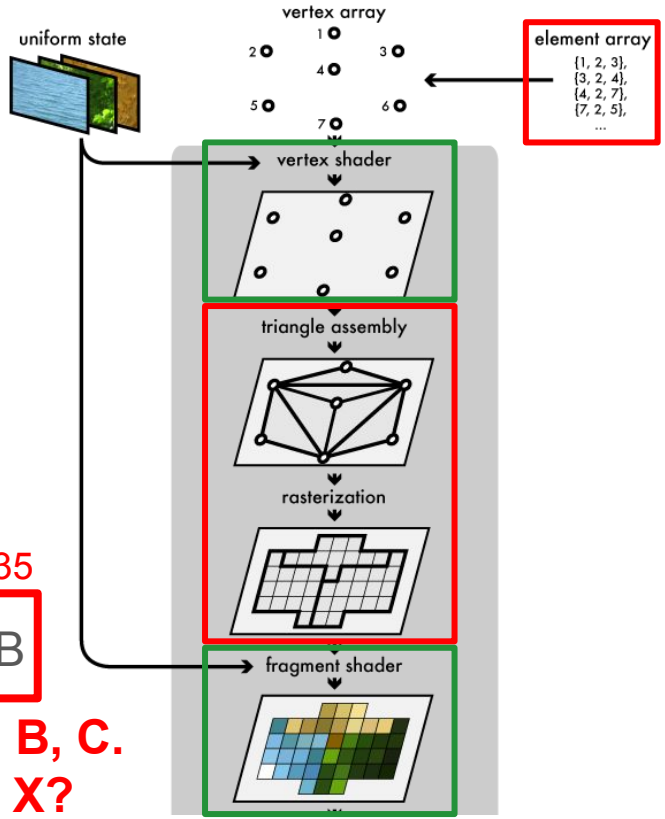


Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse (Lambert)

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Specular (Phong)

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```

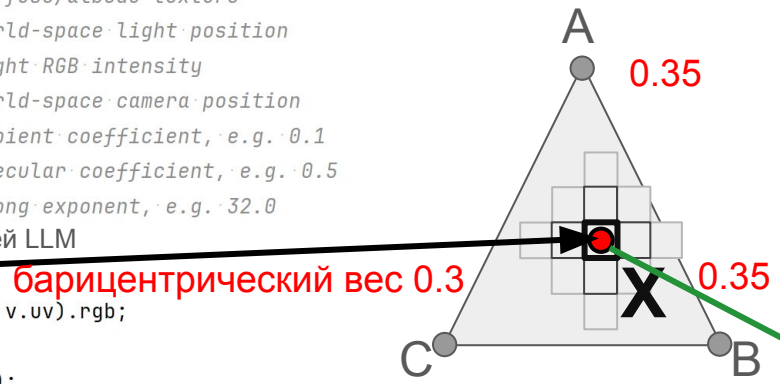


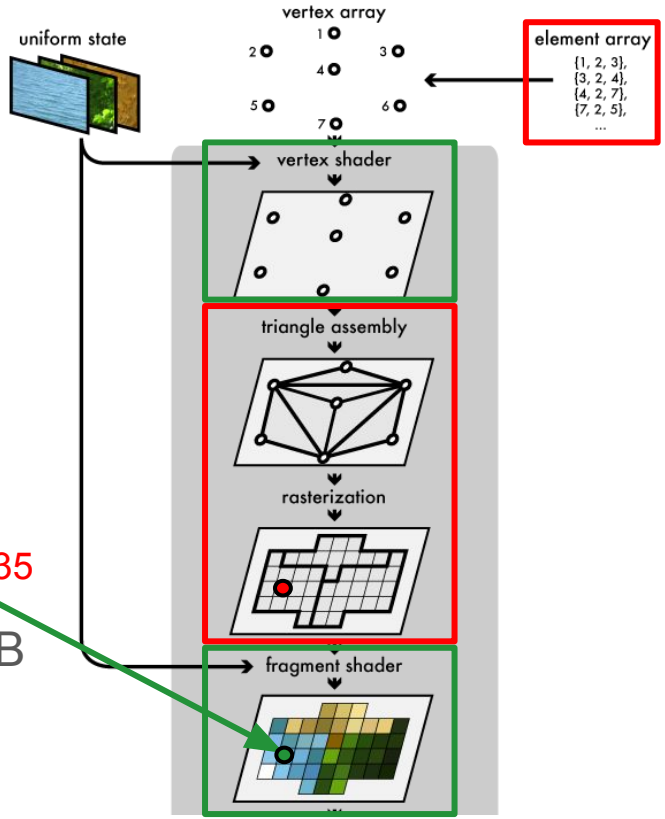
Известны свойства A, B, C.
Как найти атрибуты X?



Программируемый фрагментный шейдер **Fragment Shader**

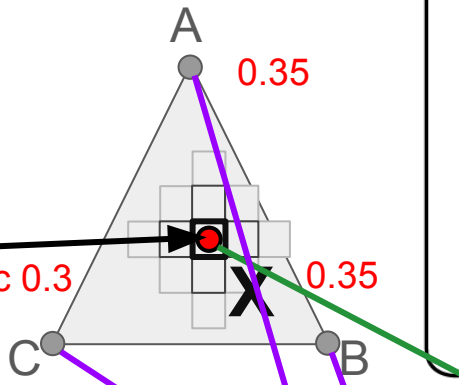
```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse (Lambert)

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Specular (Phong)

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```



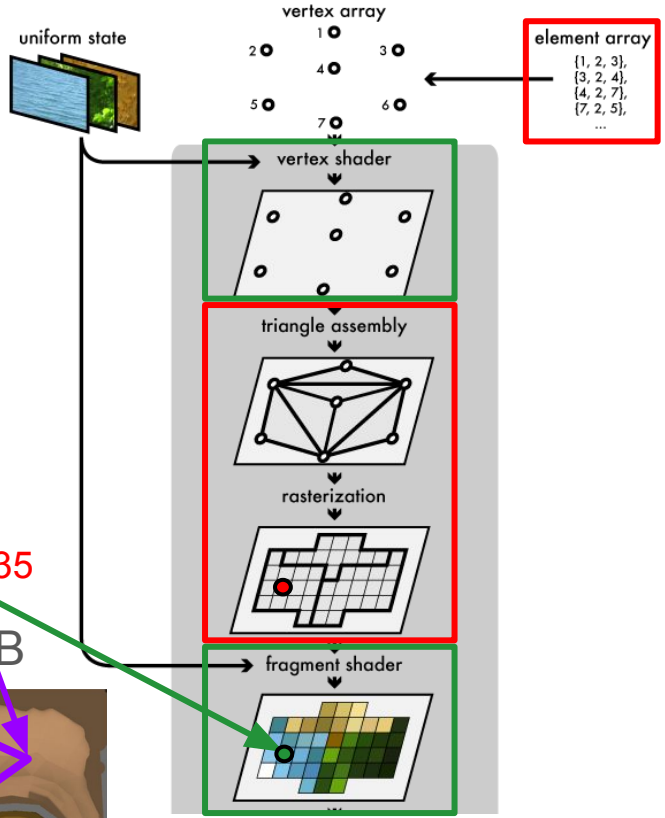


Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```

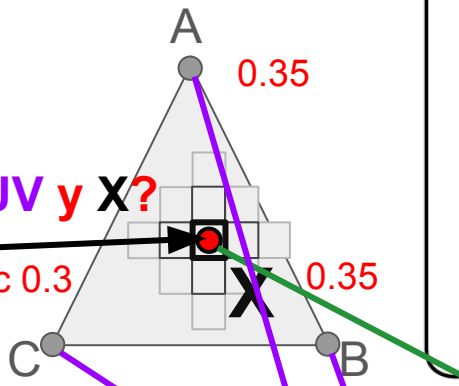


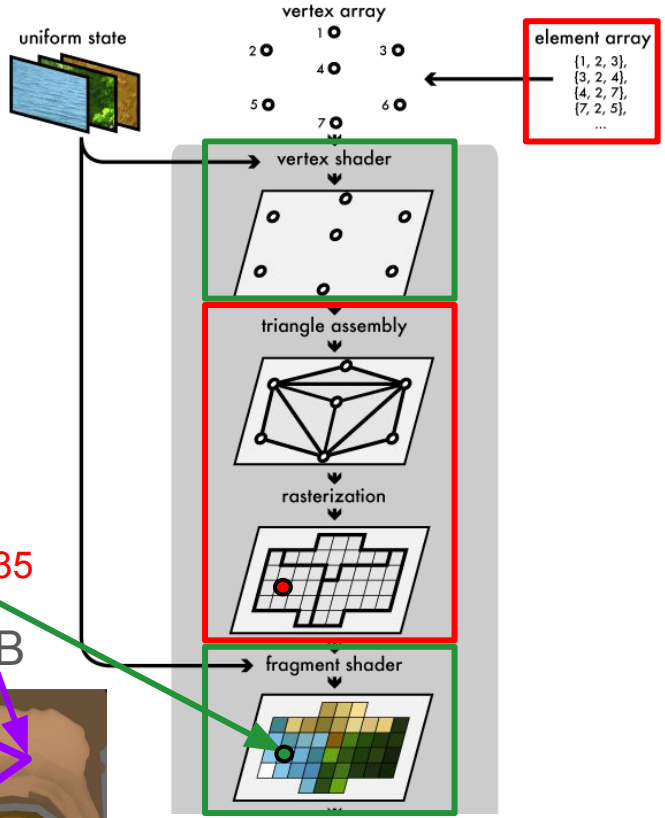
UV
texture atlas



Программируемый фрагментный шейдер **Fragment Shader**

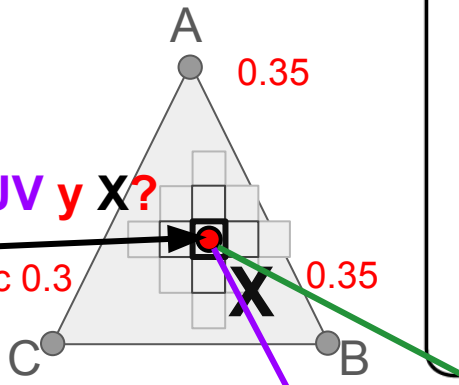
```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 64.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```



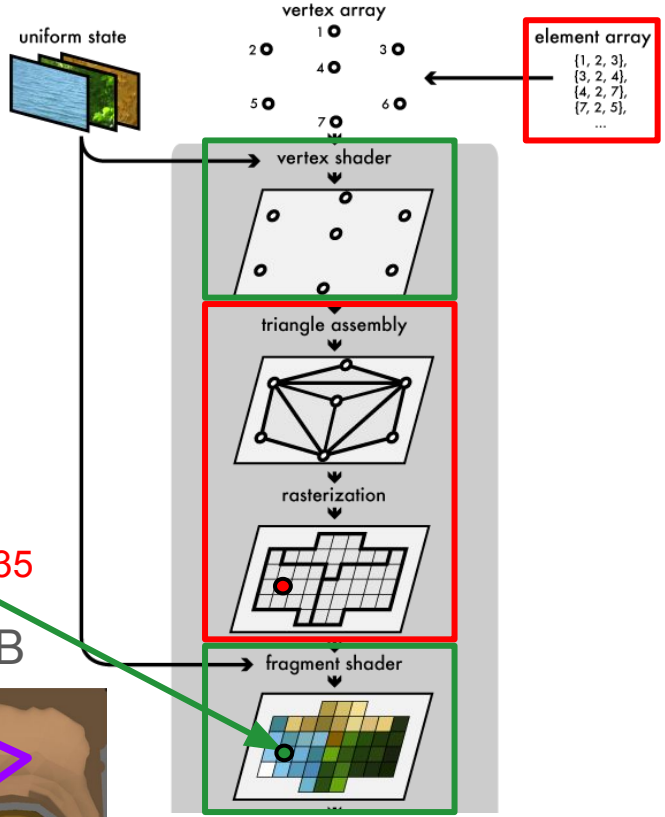


Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 64.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```

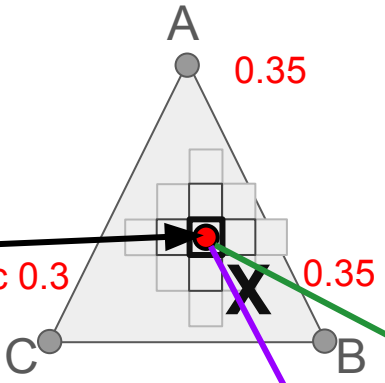


UV texture atlas



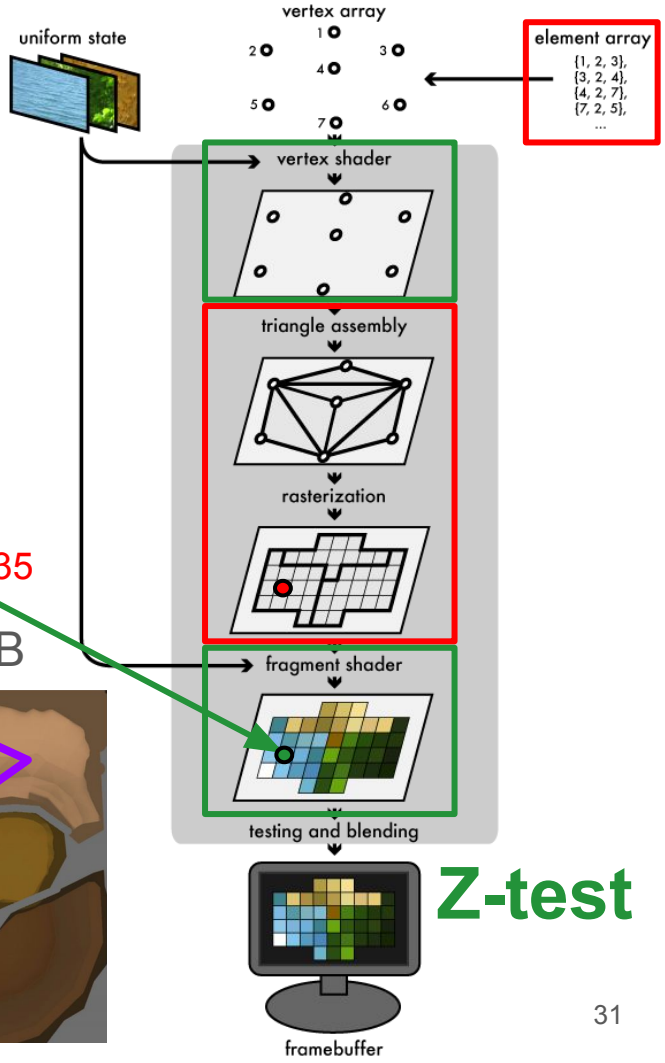
Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```



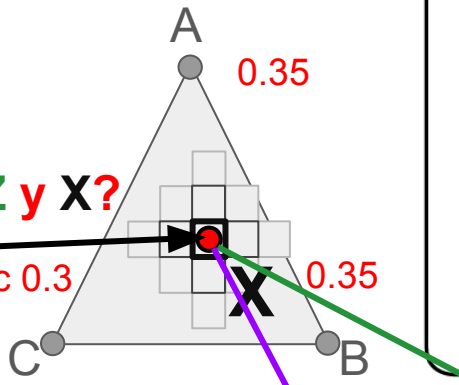
барицентрический вес 0.3

UV
texture atlas

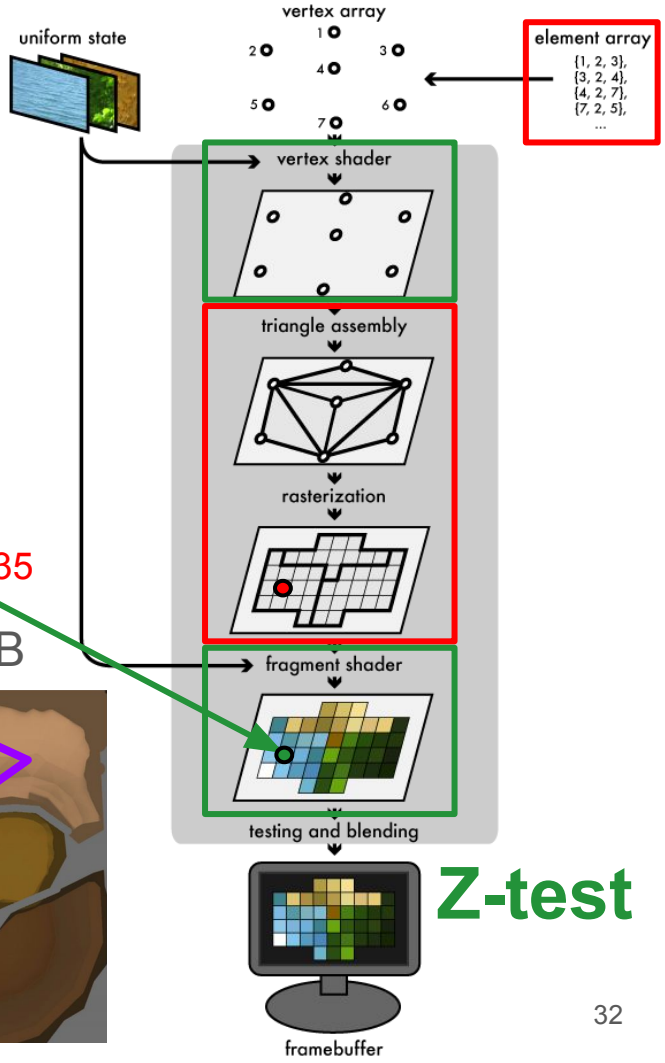


Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 64.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```



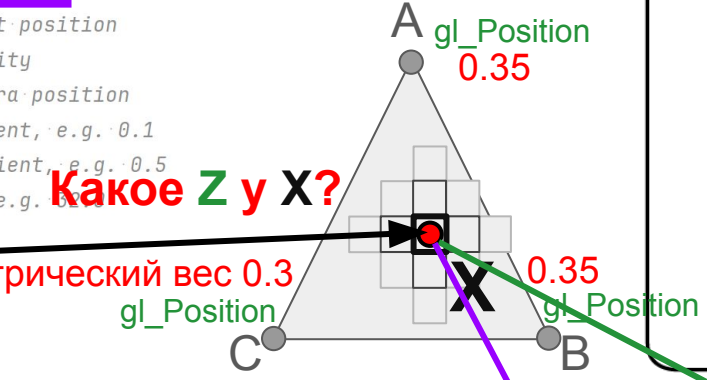
UV
texture atlas



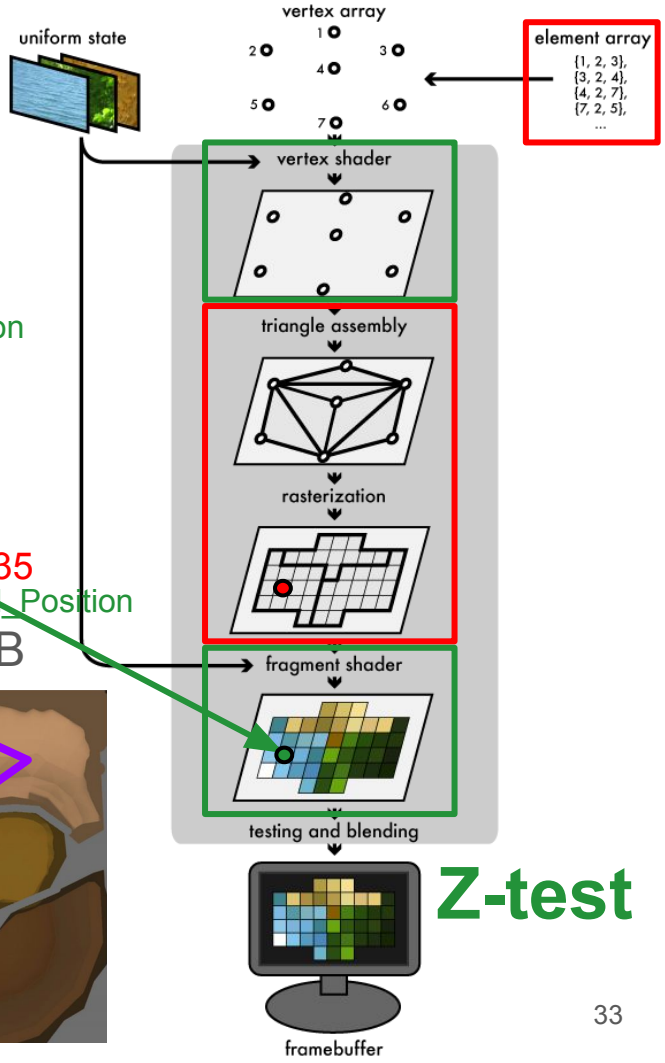
Z-test

Программируемый фрагментный шейдер **Fragment Shader**

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAlbedo;     // Diffuse/albedo texture
uniform vec3 uLightPos;        // World-space light position
uniform vec3 uLightColor;      // Light RGB intensity
uniform vec3 uViewPos;         // World-space camera position
uniform float uAmbientK;       // Ambient coefficient, e.g. 0.1
uniform float uSpecK;          // Specular coefficient, e.g. 0.5
uniform float uShininess;      // Phong exponent, e.g. 64.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```



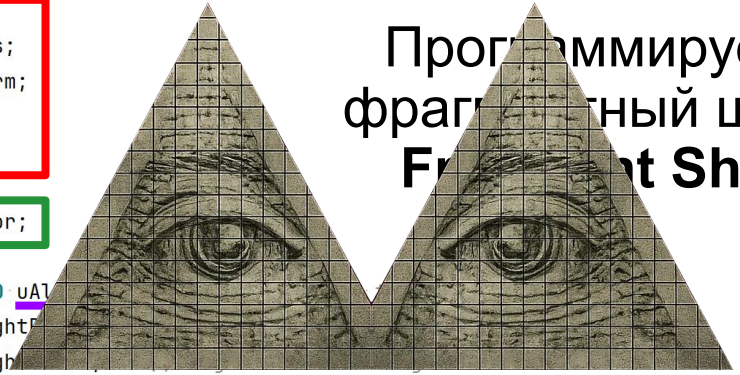
UV
texture atlas



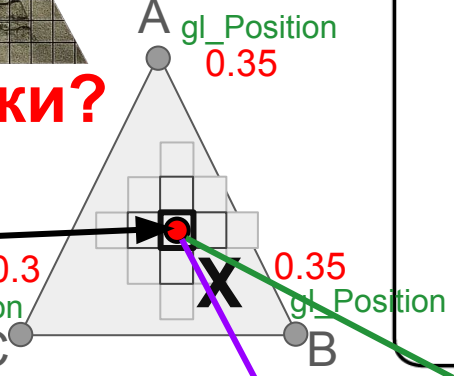
Z-test

Программируемый фрагментный шейдер
Fragment Shader

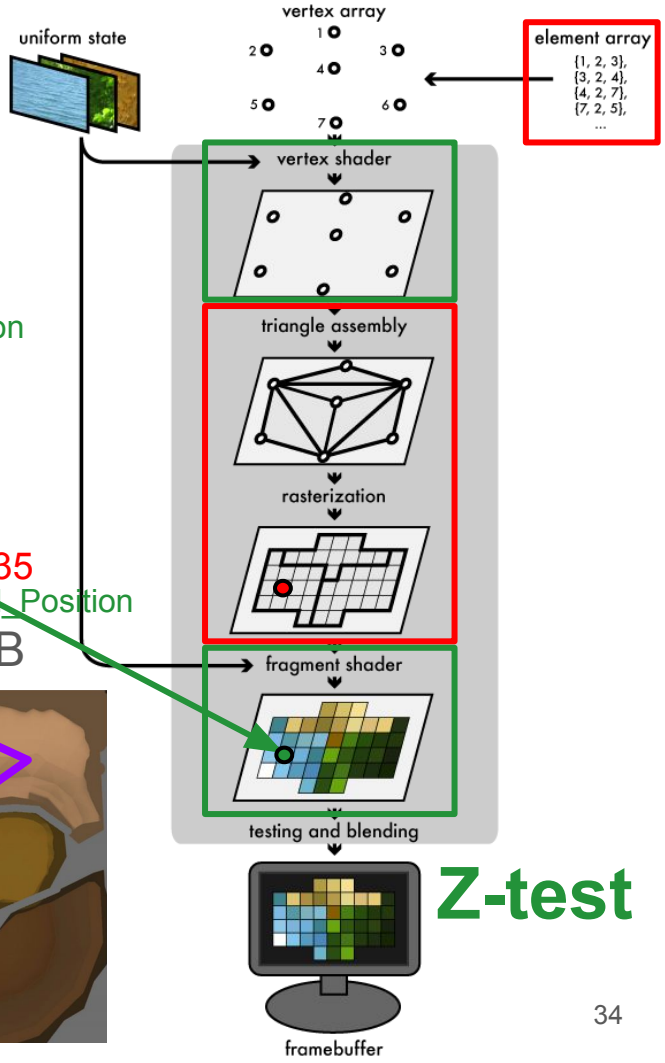
```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAl
uniform vec3 uLightP
uniform vec3 uLigh
uniform vec3 uViewPos;      // World-space camera position
uniform float uAmbientK;    // Ambient coefficient, e.g. 0.1
uniform float uSpecK;       // Specular coefficient, e.g. 0.5
uniform float uShininess;   // Phong exponent, e.g. 32.0
// NB: код является галлюцинацией LLM
void main() {
    vec3 albedo = texture(uAlbedo, v.uv).rgb;

    vec3 N = normalize(v.worldNorm);
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos  - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```



Почему треугольники?



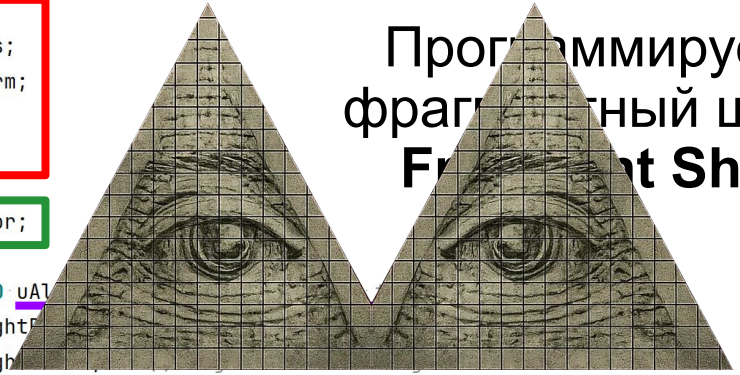
UV
texture atlas



Z-test

```
in VS_OUT {
    vec3 worldPos;
    vec3 worldNorm;
    vec2 uv;
} v;

out vec4 FragColor;

uniform sampler2D uAl
uniform vec3 uLightP
uniform vec3 uLigh
uniform vec3 uViewPos
uniform float uAmbientK;
```

Программируемый фрагментный шейдер
Fragment Shader



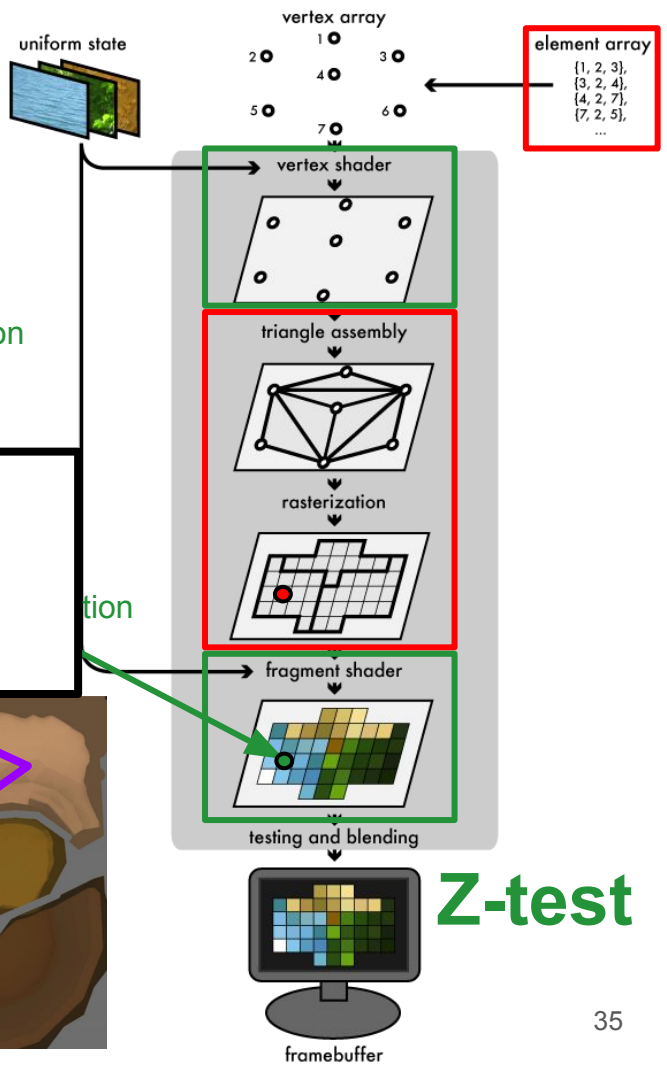
A gl_Position
0.35

Почему треугольники?

- **минимальный** плоский примитив
- любой треугольник - **плоский и выпуклый**
- удобство **барицентрических координат**

```
    vec3 L = normalize(uLightPos - v.worldPos);
    vec3 V = normalize(uViewPos - v.worldPos);

    vec3 ambient = uAmbientK * albedo; // Ambient

    vec3 diffuse = max(dot(N, L), 0.0) * albedo * uLightColor; // Diffuse

    float rdotv = max(dot(reflect(-L, N), V), 0.0);
    vec3 specular = uSpecK * pow(rdotv, uShininess) * uLightColor; // Spec

    FragColor = vec4(ambient + diffuse + specular, 1.0);
```

UV
texture atlas



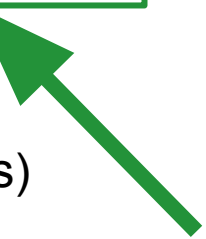
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители с **Fixed pipeline**

2000-2002 - в DirectX 8.0 и OpenGL появляются программируемые ***шейдеры***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**



2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)

2012 - Compute Shaders в OpenGL

2016 - Vulkan

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)
1995-1997 - становятся популярны 3D-ускорители с **Fixed pipeline**
2000-2002 - в DirectX 8.0 и OpenGL появляются программируемые ***шейдеры***
2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)
2006-2009 - **Larrabee** (INTEL): дискретный GPU на базе x86 (т.е. CPU-like)

2012 - Compute Shaders в OpenGL

2016 - Vulkan

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)
1995-1997 - становятся популярны 3D-ускорители с **Fixed pipeline**
2000-2002 - в DirectX 8.0 и OpenGL появляются программируемые ***шейдеры***
2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)
2006-2009 - **Larrabee** (INTEL): дискретный GPU на базе x86 (т.е. CPU-like)
2011 - **cudaraster** (NV): software-based графический пайплайн на CUDA
2012 - Compute Shaders в OpenGL

2016 - Vulkan

История GPU

До **1996** года 3D графика на CPU (DOOM, Qu

1995-1997 - становятся популярны 3D-ускори ne

2000-2002 - в DirectX 8.0 и OpenGL появляют ые ***шейдеры***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - з

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** Khronos)

2006-2009 - **Larrabee** (INTEL): дискретный GPU на базе x86 (т.е. CPU-like)

2011 - **cudaraster** (NV): software-based графический пайплайн на CUDA

2012 - Compute Shaders в OpenGL

Зачем нам это изучать?

Есть ли практическая польза?

2016 - Vulkan

История GPU

До **1996** года 3D графика на CPU (DOOM, Qu

1995-1997 - становятся популярны 3D-ускори

2000-2002 - в DirectX 8.0 и OpenGL появляют

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - з

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA**

2006-2009 - **Larrabee** (INTEL): дискретный GPU на базе x86 (т.е. CPU-like)

2011 - **cudaraster** (NV): software-based графический пайплайн на CUDA

2012 - Compute Shaders в OpenGL

2016 - Vulkan

Зачем нам это изучать?

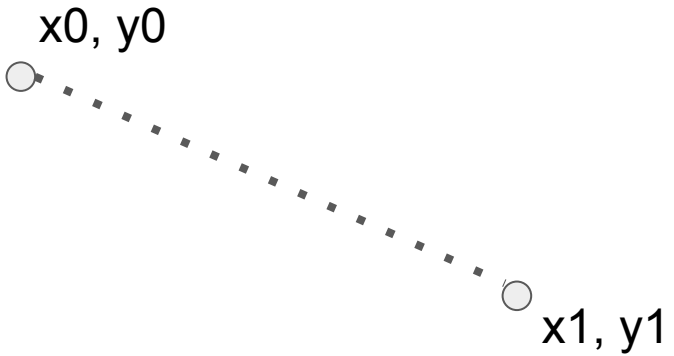
Есть ли практическая польза?

Если вы не знаете как что-то софтварно сэмулировать,
то вы не знаете как оно работает!

Сунь Цзы, Искусство войны, § 69

```
void line(x0, y0, x1, y1) {
```

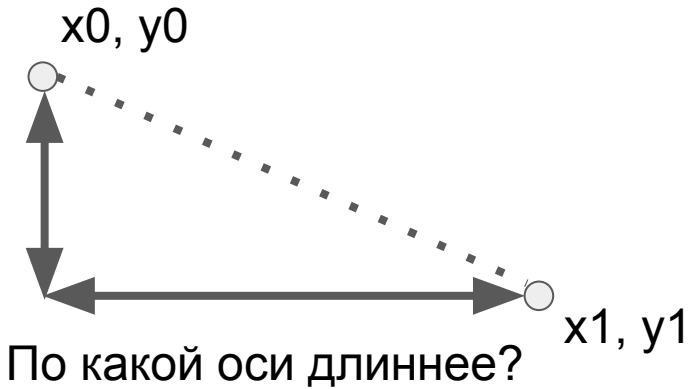
Алгоритм Брезенхэма - растеризация отрезка



Подробнее: habrahabr - пишем упрощённый OpenGL своими руками

Алгоритм Брезенхэма - растеризация отрезка

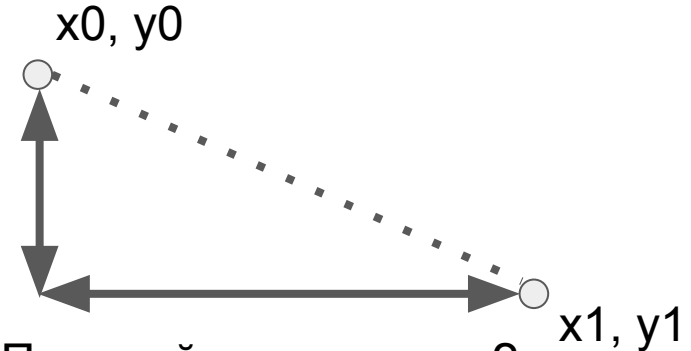
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
```



Подробнее: habrahabr - пишем упрощённый OpenGL своими руками

Алгоритм Брезенхэма - растеризация отрезка

```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
```



По какой оси длиннее?
Теперь точно **по оси x!**

Подробнее: habrahabr - пишем упрощённый OpenGL своими руками

Алгоритм Брезенхэма - растеризация отрезка

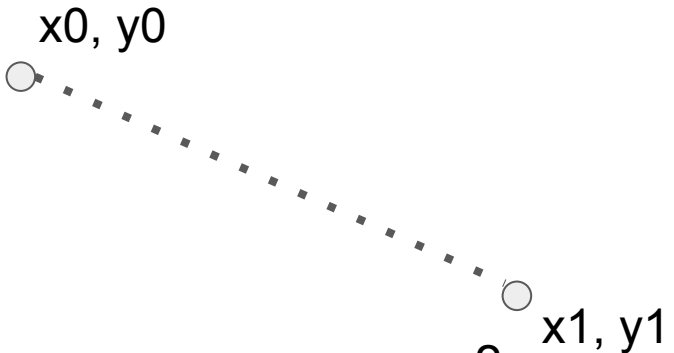
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }
```

Слева направо или справа налево?

Подробнее: habrahabr - пишем упрощённый OpenGL своими руками

Алгоритм Брезенхэма - растеризация отрезка

```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }
```

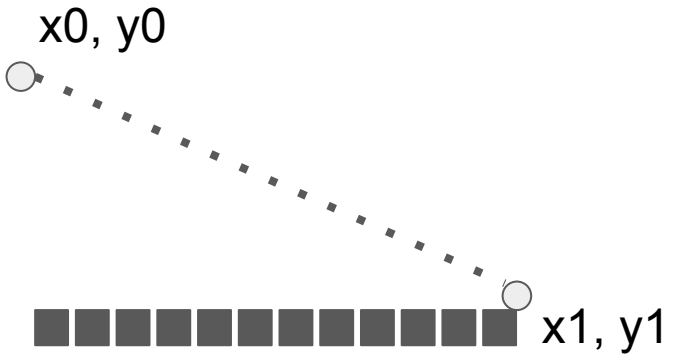


Слева направо или справа налево?
Теперь точно **слева направо!**

Подробнее: habrahabr - пишем упрощённый OpenGL своими руками

Алгоритм Брезенхэма - растеризация отрезка

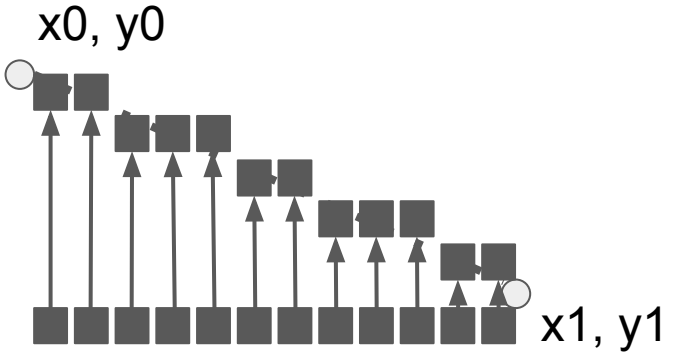
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }

    for (int x = x0; x <= x1; ++x) {
```



Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

Алгоритм Брезенхэма - растеризация отрезка

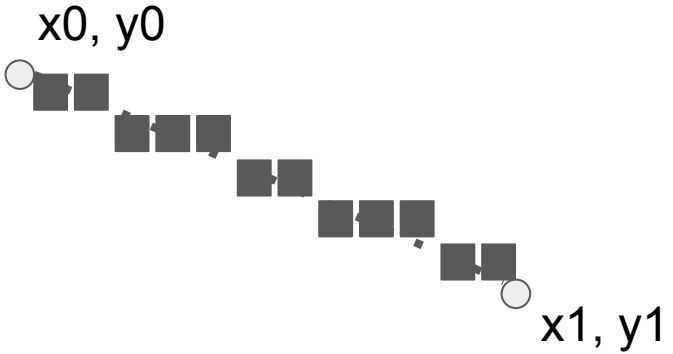
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }

    for (int x = x0; x <= x1; ++x) {
        float t = (x - x0) / (x1 - x0);
        int y = y0 * (1 - t) + y1 * t;
```



Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

Алгоритм Брезенхэма - растеризация отрезка

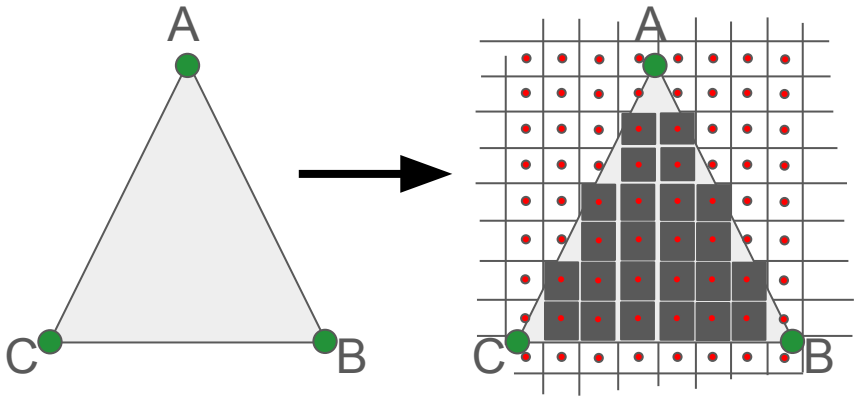
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }

    for (int x = x0; x <= x1; ++x) {
        float t = (x - x0) / (x1 - x0);
        int y = y0 * (1 - t) + y1 * t;
        if (steep) {
            put_pixel(y, x); // if transposed, de-transpose
        } else {
            put_pixel(x, y);
        }
    }
}
```



Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

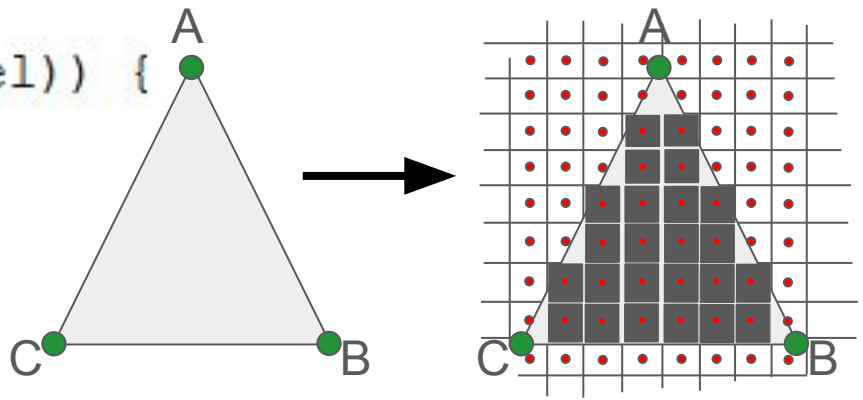
растеризация треугольника

Как сделать тривиальную версию?



Как сделать тривиальную версию?

```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```

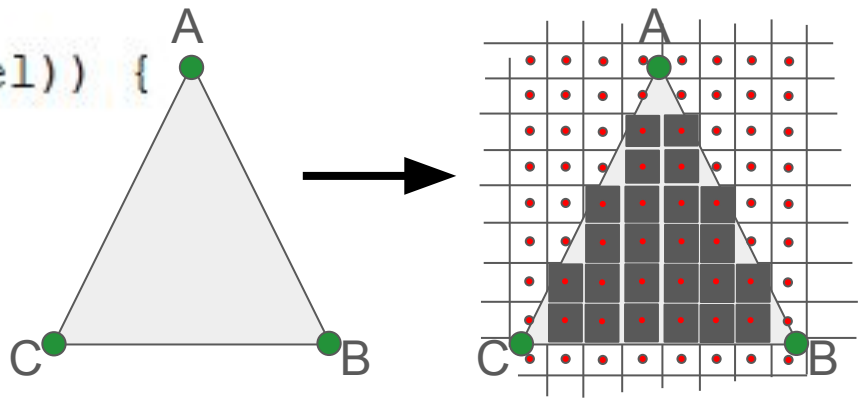


растеризация треугольника

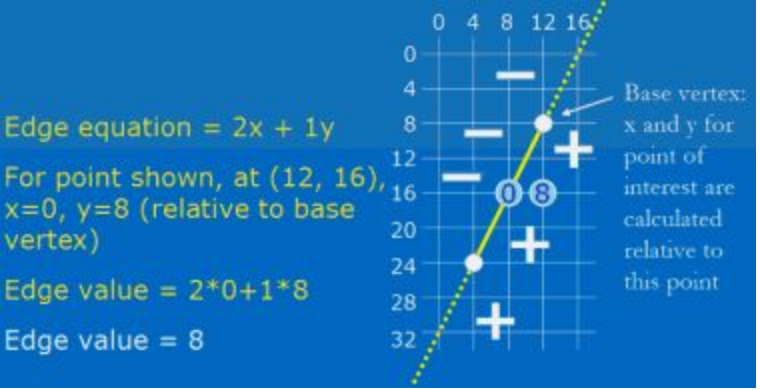
Как сделать тривиальную версию?

Как проверить что точка внутри треугольника?

```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```



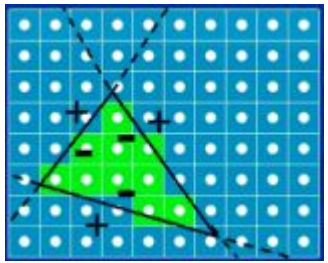
растеризация треугольника

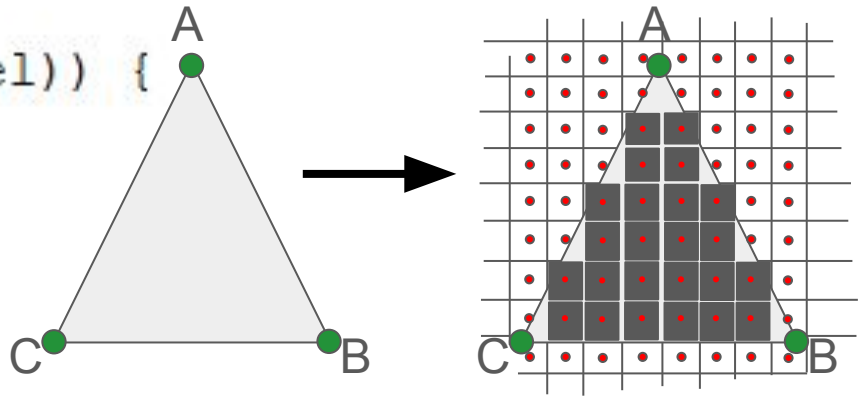


Как сделать тривиальную версию?

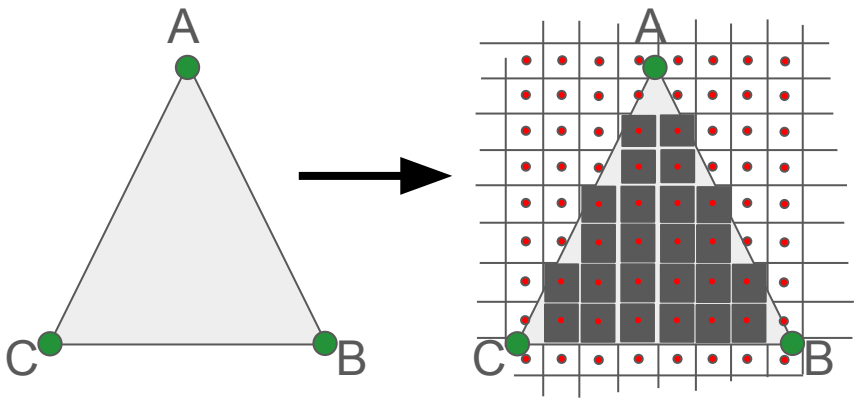
Как проверить что точка внутри треугольника?

```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```



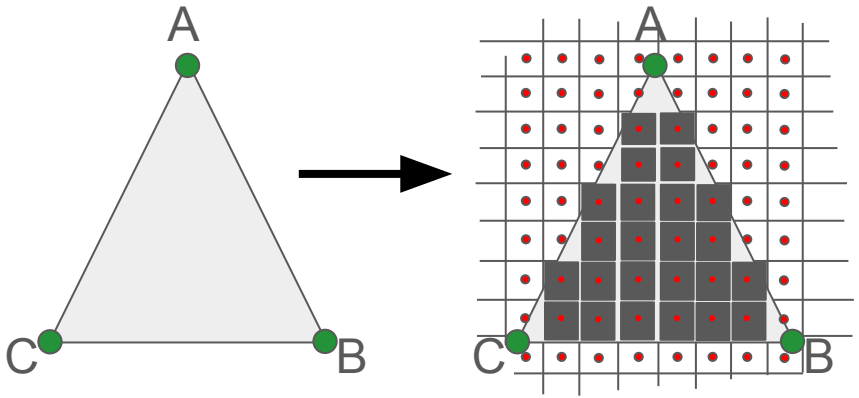


Как сделать однопоточную эффективную версию?



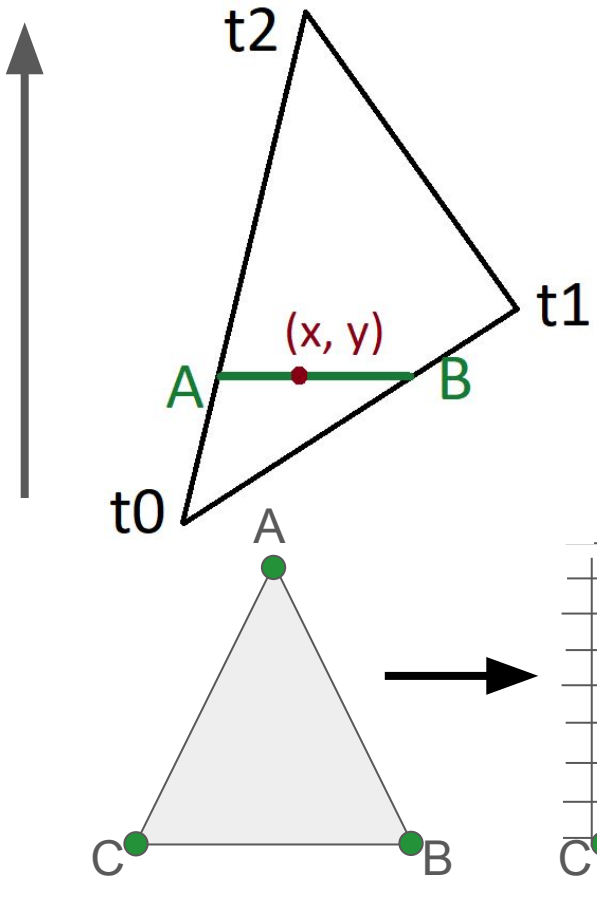
Алгоритм Брезенхэма - растеризация треугольника

Как сделать однопоточную эффективную версию?



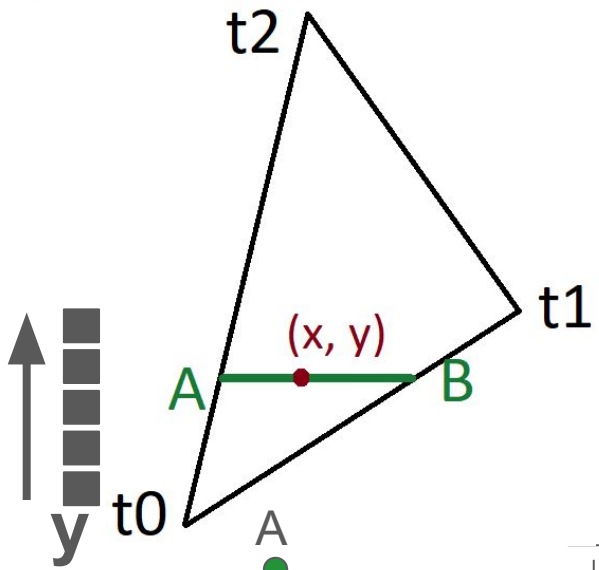
Алгоритм Брезенхэма
растеризация треугольника

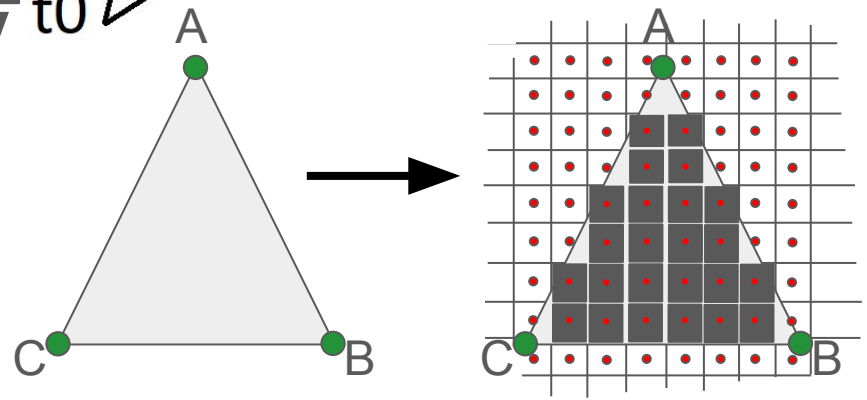
```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);
```



Алгоритм Брезенхэма
растеризация треугольника

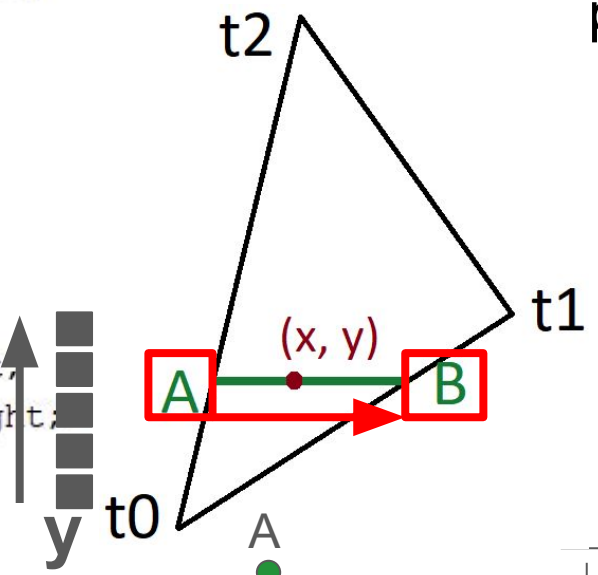
```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
```

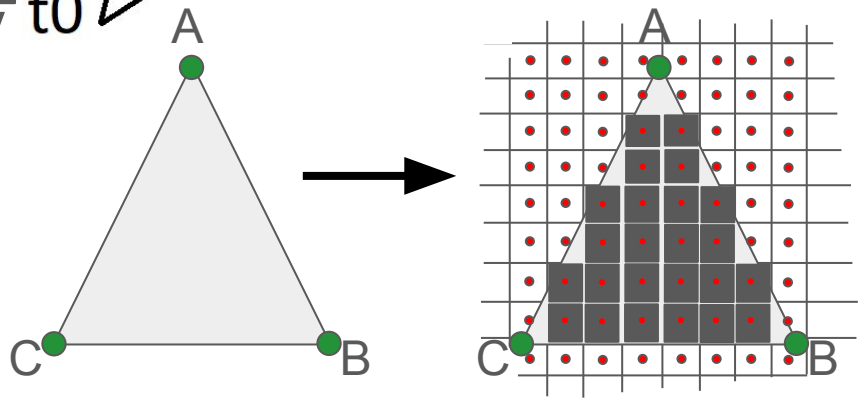




Алгоритм Брезенхэма
растеризация треугольника

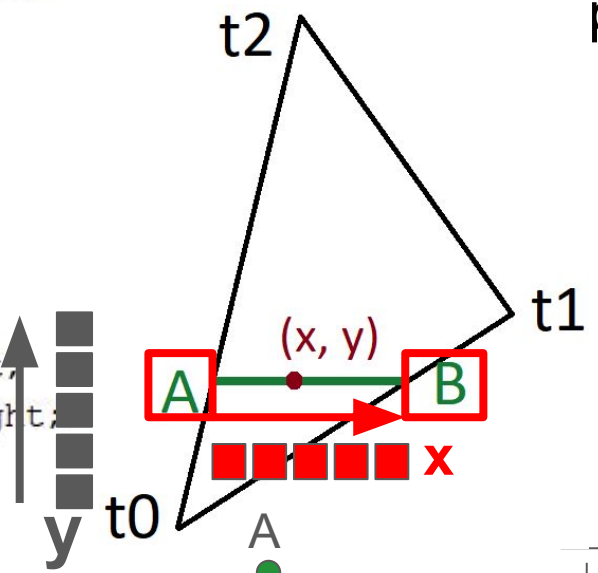
```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
```

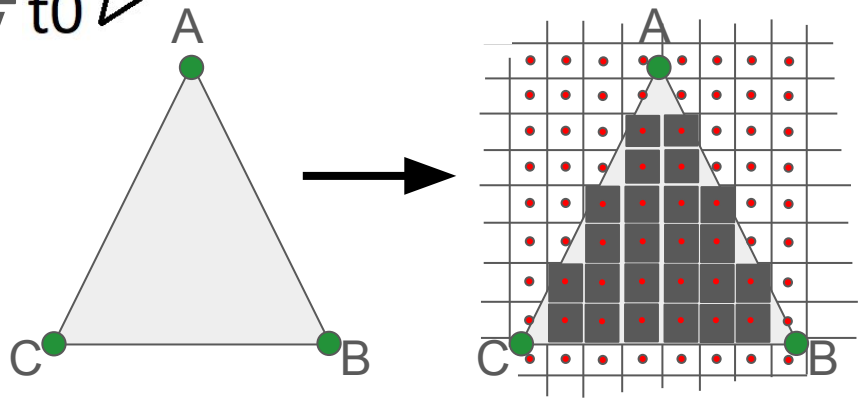




Алгоритм Брезенхэма
растеризация треугольника

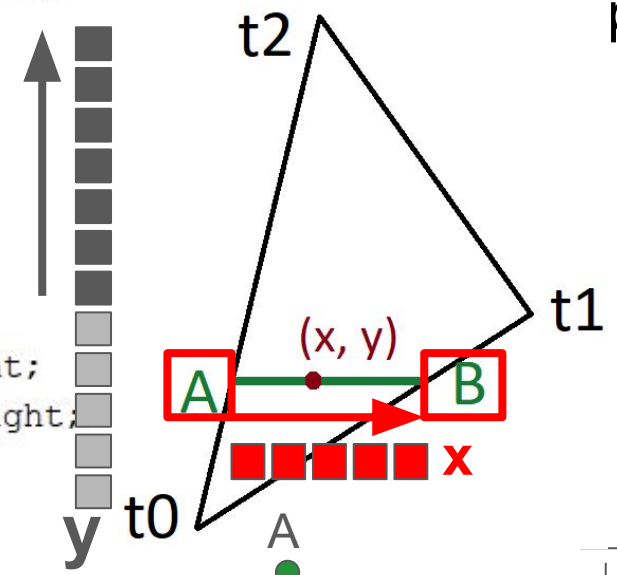
```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
        for (int x = A.x; x <= B.x; ++x) {
            put_pixel(x, y);
        }
    }
```

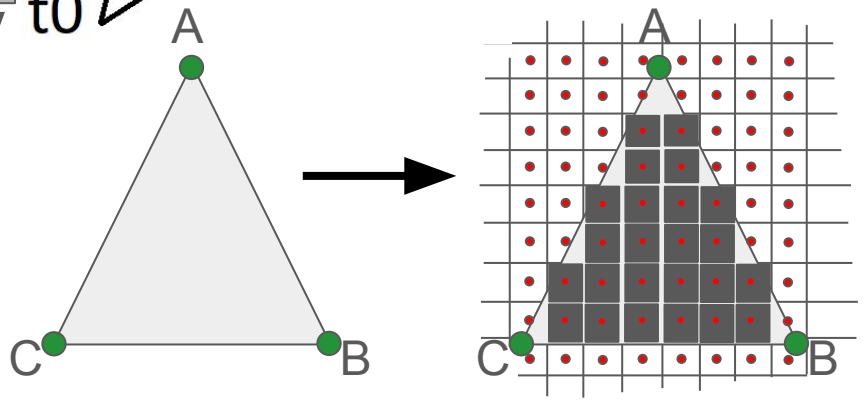




Алгоритм Брезенхэма
растеризация треугольника

```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
        for (int x = A.x; x <= B.x; ++x) {
            put_pixel(x, y);
        }
    }
    for (int y = t1.y; y <= t2.y; ++y) {
        ...
    }
}
```

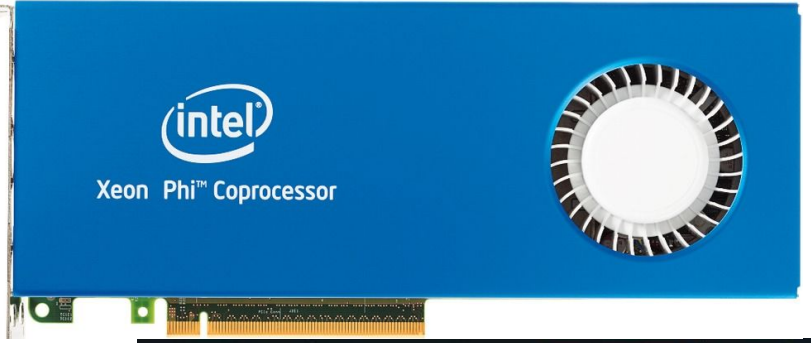




Глава 2: **Larrabee**

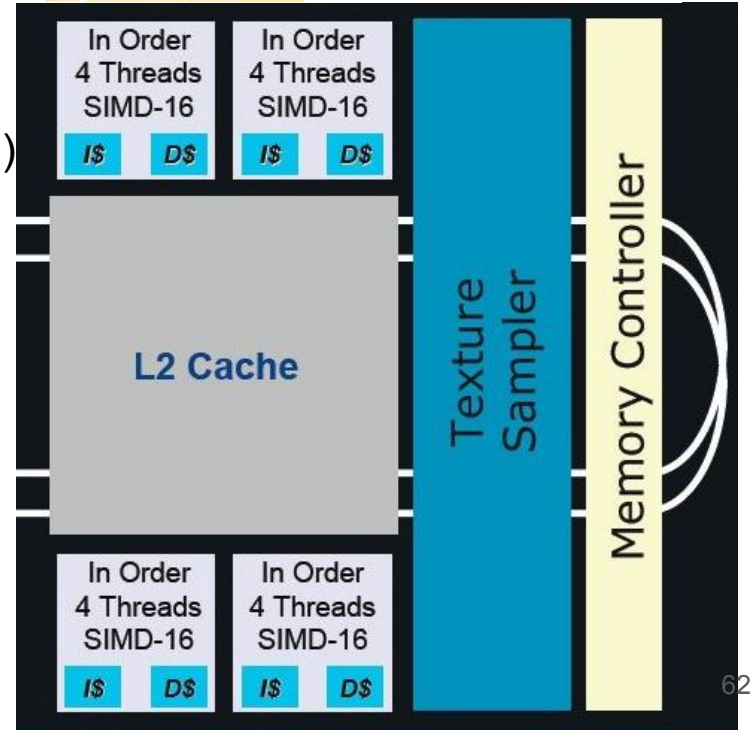
Софтварная растеризация на multi-core x86 CPU

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

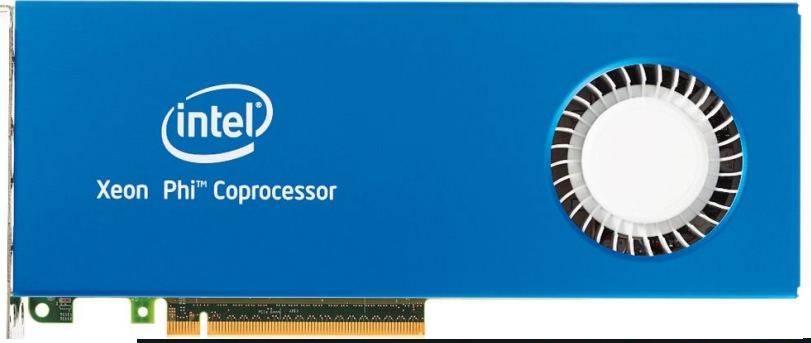


Предтеча **AVX-512**

- Несколько CPU-like ядер (примерно 10-60)
  (родственники Pentium P54C)
- 4-поточная **SMT** для сокрытия latency (**x4** regfile, **x4** L1)
- Широкие **SIMD-16** инструкции
  (предтеча **AVX-512**)



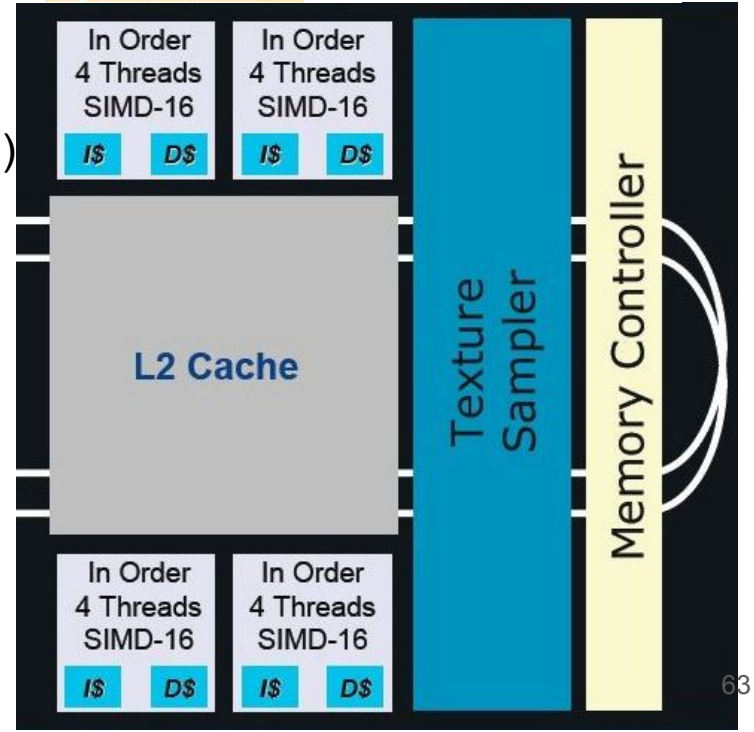
# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**



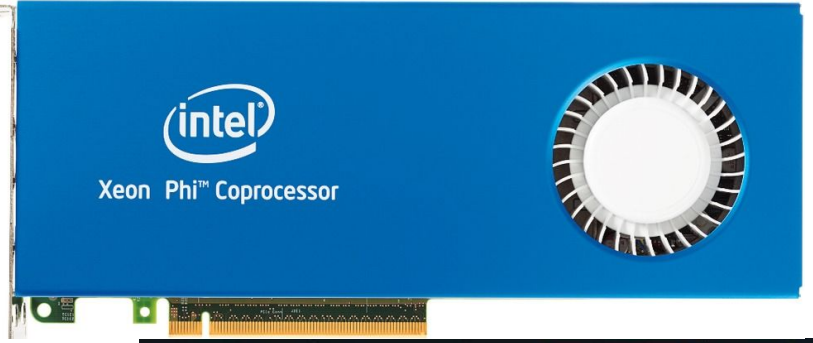
Предтеча **AVX-512**

- Несколько CPU-like ядер (примерно 10-60)
  (родственники Pentium P54C)
- 4-поточная **SMT** для сокрытия latency (**x4** regfile, **x4** L1)
- Широкие **SIMD-16** инструкции
  (предтеча **AVX-512**)

- Должна была соревноваться с GTX 285
- В **hardware** только блоки **texture sampler!**
- **Software rasterization!**



# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

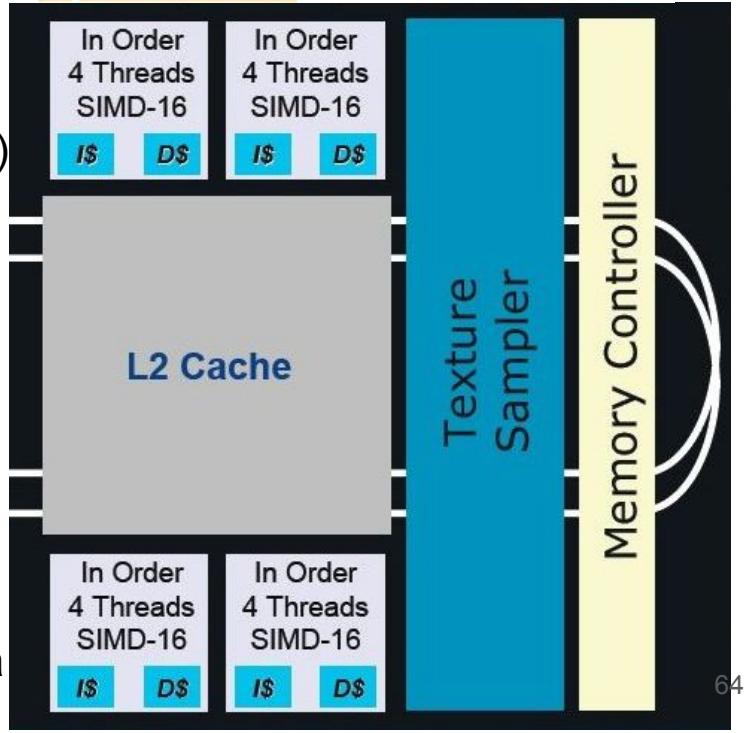


Предтеча **AVX-512**

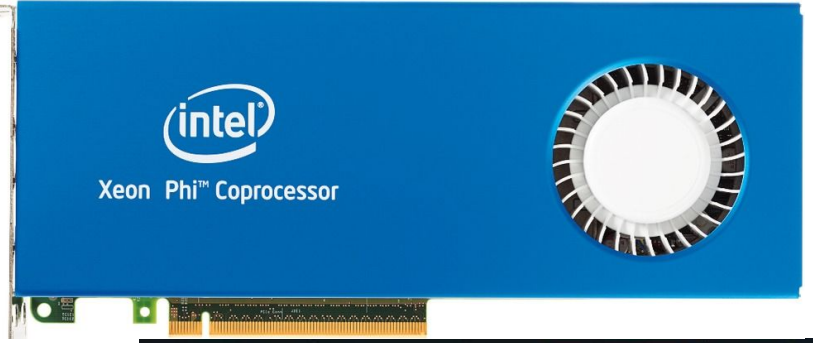
- Несколько CPU-like ядер (примерно 10-60)
  (родственники Pentium P54C)
- 4-поточная **SMT** для сокрытия latency (**x4** regfile, **x4** L1)
- Широкие **SIMD-16** инструкции
  (предтеча **AVX-512**)

- Должна была соревноваться с GTX 285
- В **hardware** только блоки **texture sampler!**
- **Software rasterization!**

- “Software is the new Hardware” - гибкость!
- Проблемы в **производительности** (5x FP32 слабее)
- Проблемы ring bus, x86 memory model, видеодрайвера
- Проект перетек в **HPC** область

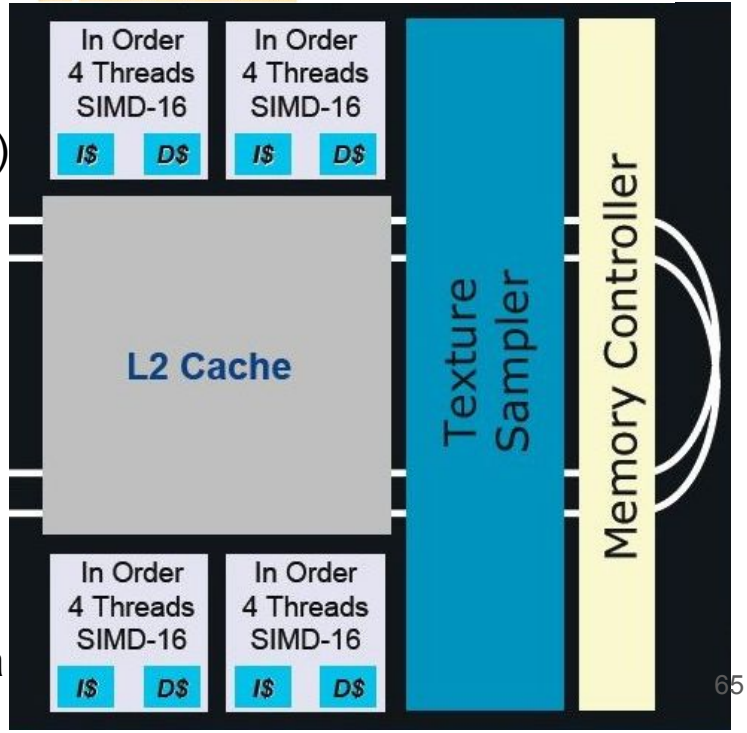


# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**



Предтеча **AVX-512**

- Несколько CPU-like ядер (примерно 10-60)
  (родственники Pentium P54C)
- 4-поточная **SMT** для сокрытия latency (**x4** regfile, **x4** L1)
- Широкие **SIMD-16** инструкции
  (предтеча **AVX-512**)



- Должна была соревноваться с GTX 285
- В **hardware** только блоки **texture sampler!**
- **Software rasterization!**

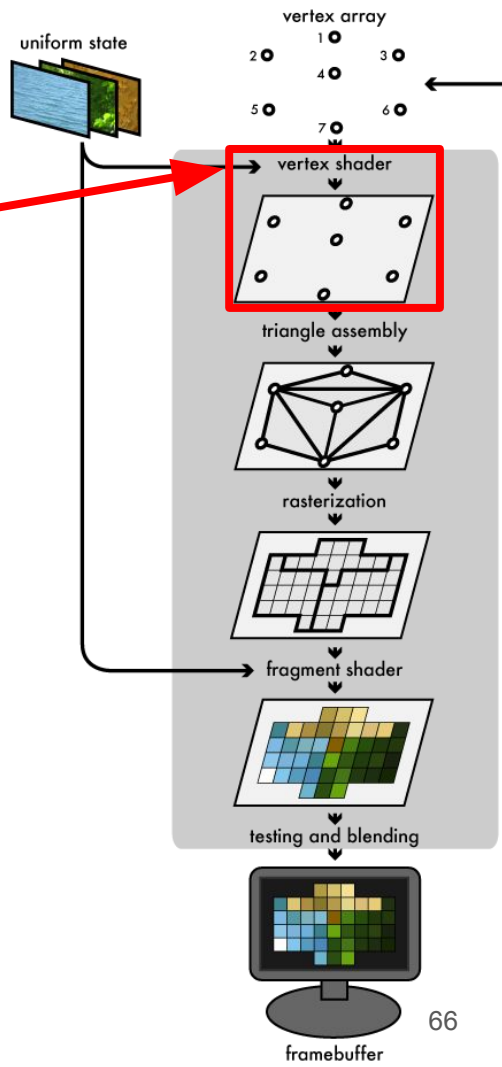
Что представляет сложность?

- **“Software is the new Hardware” - гибкость!**
- Проблемы в **производительности** (5x FP32 слабее)
- Проблемы ring bus, x86 memory model, видеодрайвера
- Проект перетек в **HPC** область

Larrabee

Как реализовать **вершинный шейдер**?

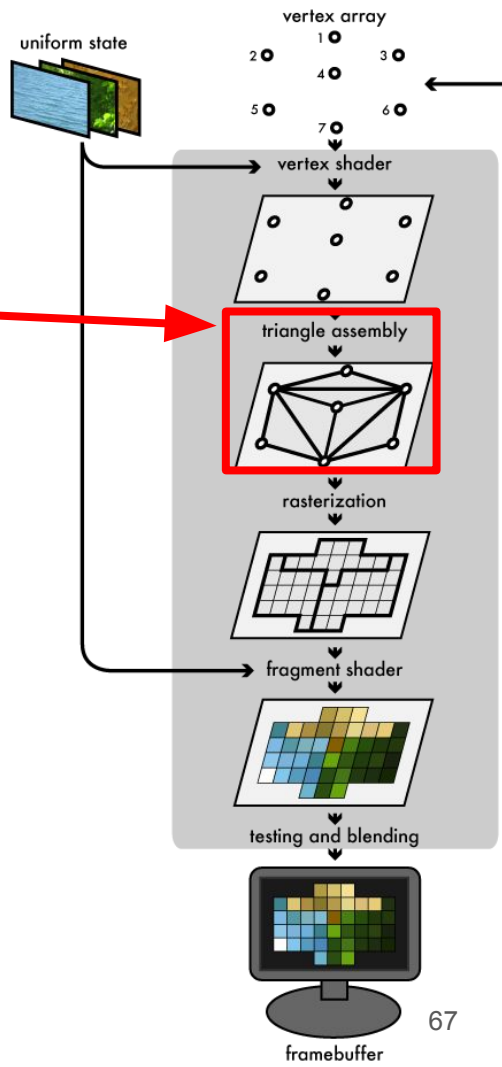
Легко ли равномерно распределить выч. нагрузку?



Larrabee

Как реализовать **triangle assembly**?

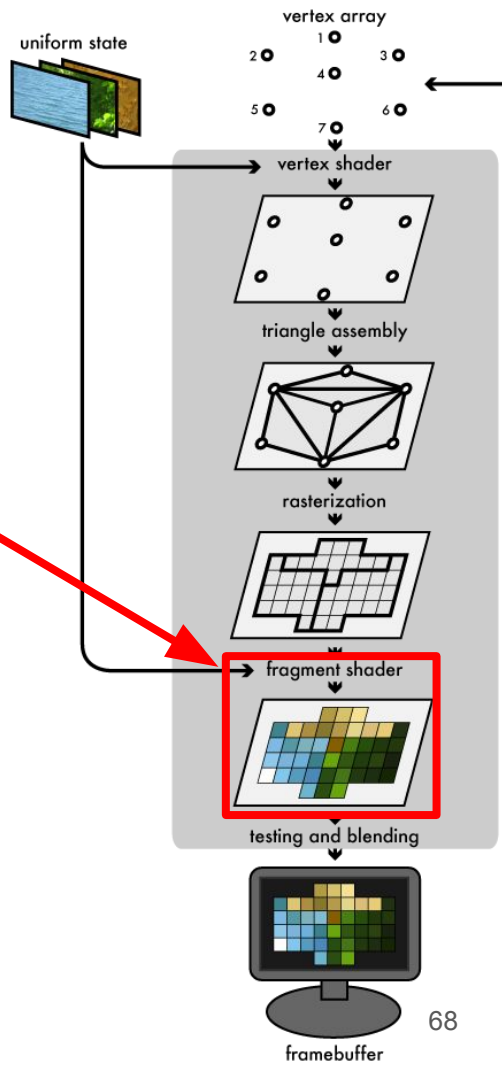
Легко ли равномерно распределить выч. нагрузку?



Larrabee

Как реализовать **фрагментный шейдер**?

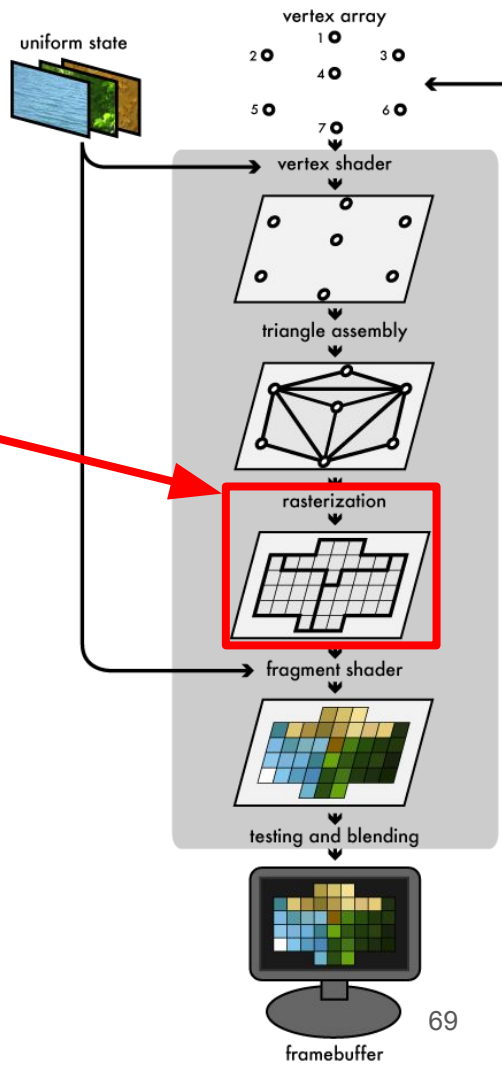
Легко ли равномерно распределить выч. нагрузку?



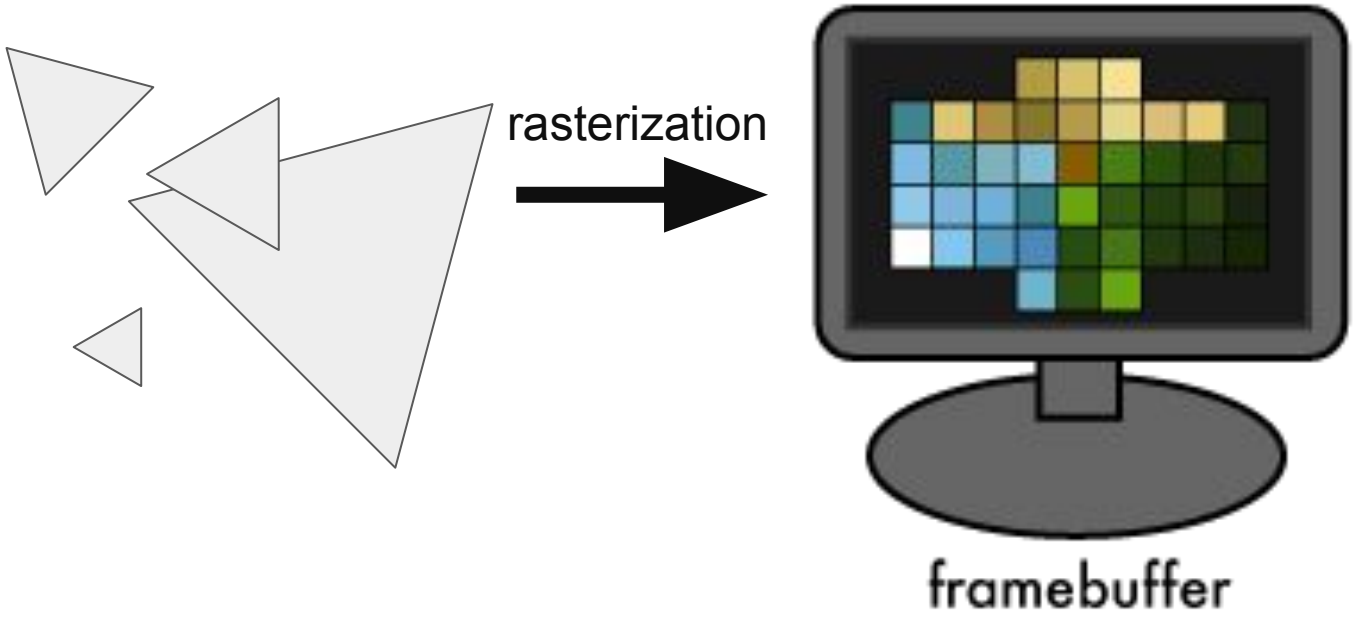
Larrabee

Как реализовать **растеризацию**?

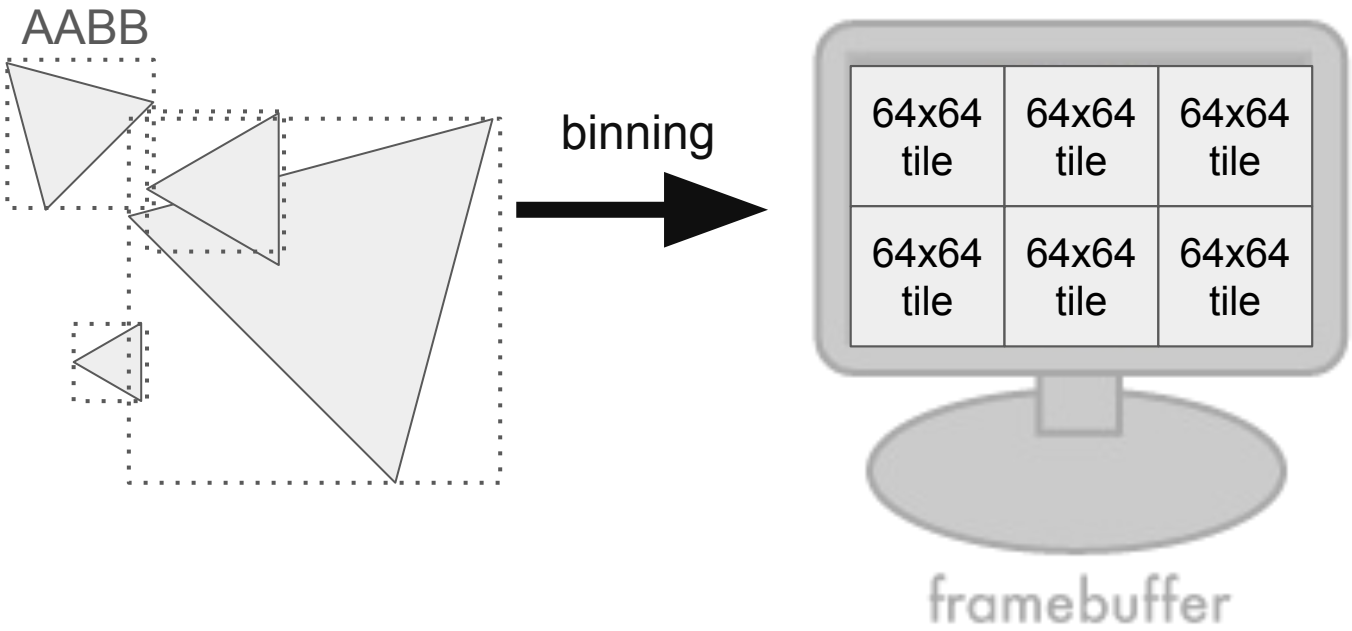
Легко ли равномерно распределить выч. нагрузку?



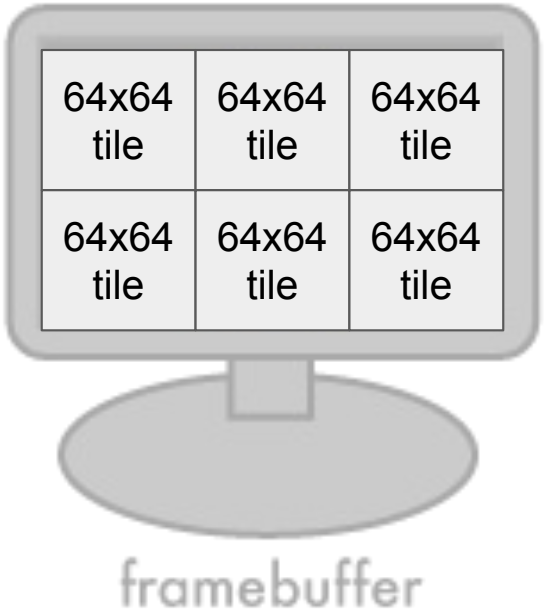
Larrabee растеризация: **hierarchical approach, binning**



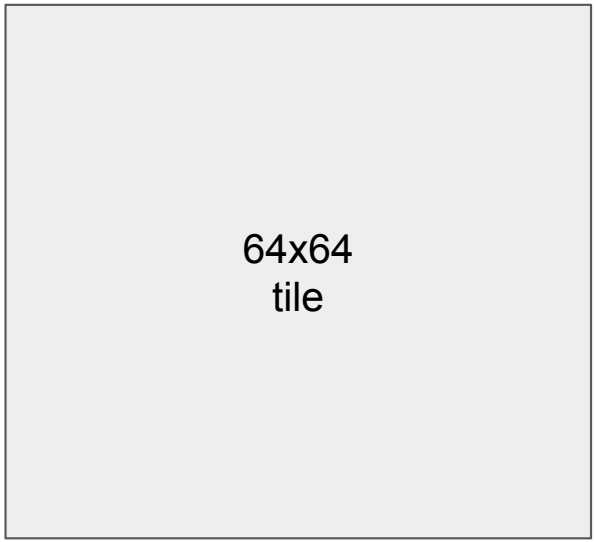
Larrabee растеризация: **hierarchical approach, binning**



Larrabee растеризация: **hierarchical approach, binning**



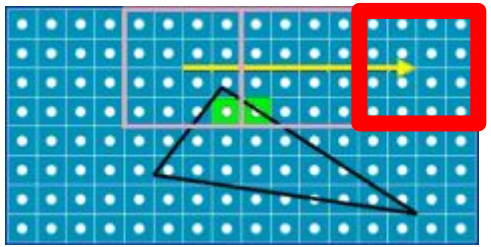
Один тайл со списком треугольников - **одно ядро**
(с SIMD-512 инструкциями, тайл влезает в L2 кэш)



Larrabee растеризация: **hierarchical approach, binning**

Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).

Как одним потоком с SIMD-16 посчитать маску блока?

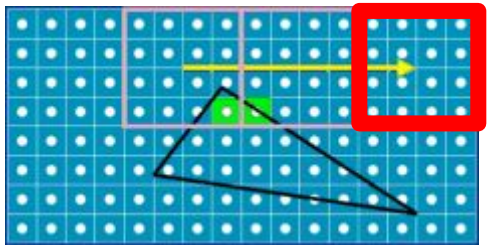


Подробнее: Rasterization on Larrabee: Why Rasterization Was the Problem Child

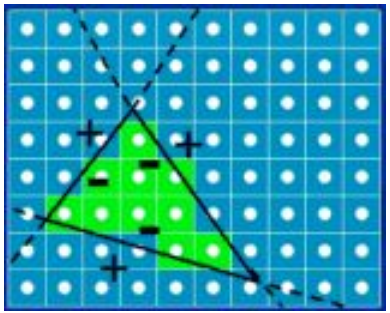
Larrabee растеризация: **hierarchical approach, binning**

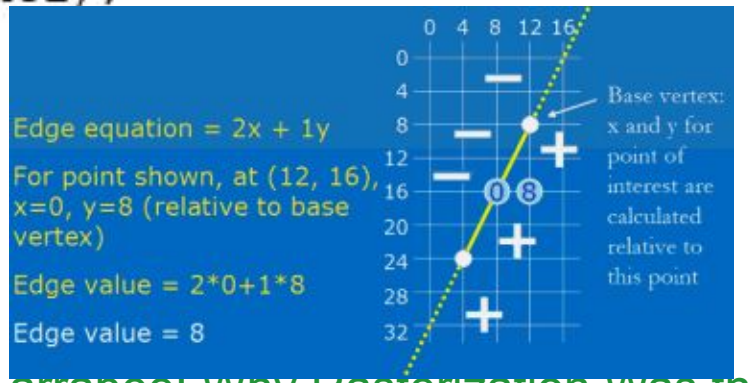
Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).

Как одним потоком с SIMD-16 посчитать маску блока?



```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```

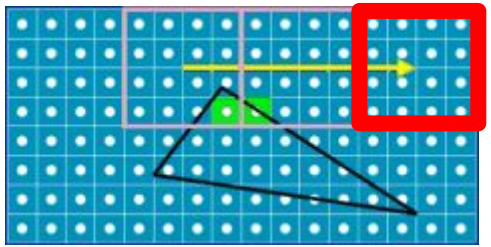




Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

Larrabee растеризация: **hierarchical approach, binning**

Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).



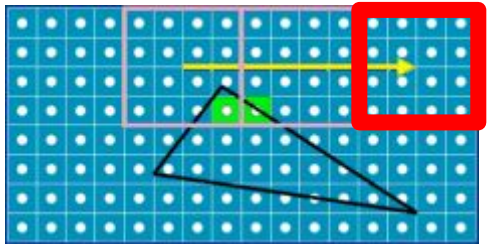
Начав с верхней вершины треугольника:

1) Скользим вправо, пока треугольник не закончится.

Подробнее: Rasterization on Larrabee: Why Rasterization Was the Problem Child

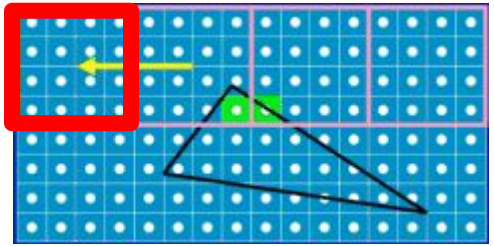
Larrabee растеризация: **hierarchical approach, binning**

Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).



Начав с верхней вершины треугольника:

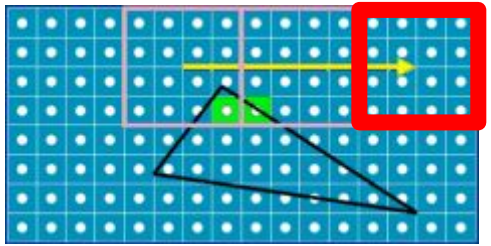
1) Скользим вправо, пока треугольник не закончится.
2) Скользим влево, пока треугольник не закончится.



Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

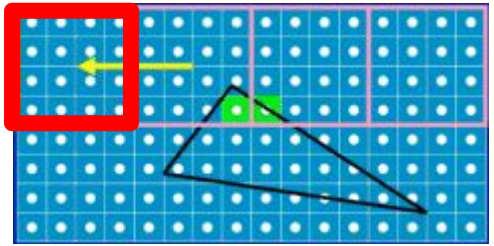
Larrabee растеризация: **hierarchical approach, binning**

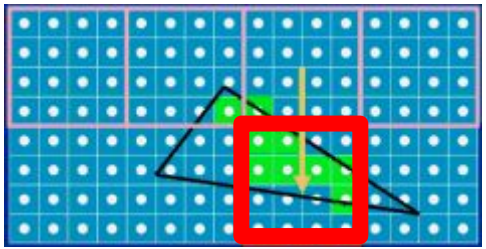
Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).



Начав с верхней вершины треугольника:

1) Скользим вправо, пока треугольник не закончится.
2) Скользим влево, пока треугольник не закончится.
3) Делаем шаг вниз и повторяем процедуру.

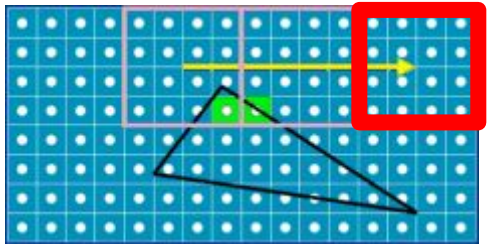




Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

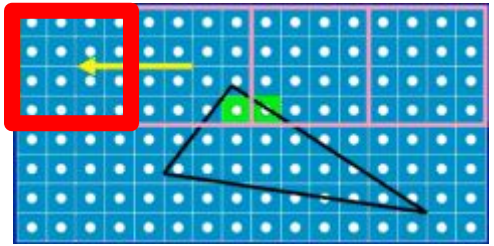
Larrabee растеризация: **hierarchical approach, binning**

Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).

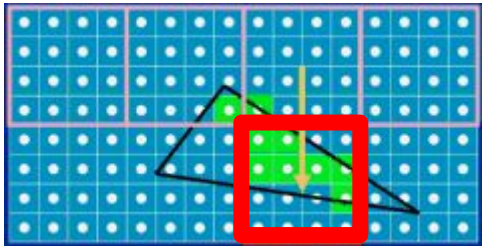


Начав с верхней вершины треугольника:

1) Скользим вправо, пока треугольник не закончится.
2) Скользим влево, пока треугольник не закончится.
3) Делаем шаг вниз и повторяем процедуру.



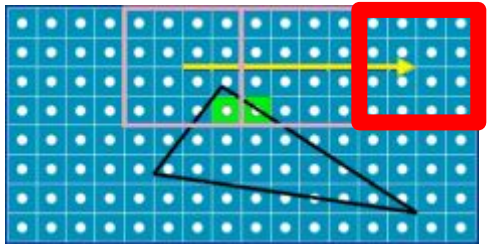
Высокоуровнево это напоминает…



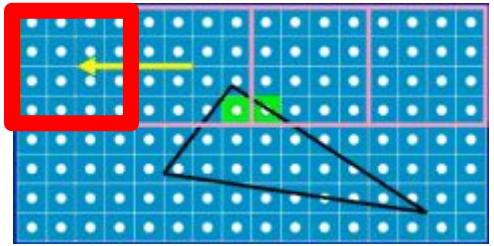
Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

Larrabee растеризация: **hierarchical approach, binning**

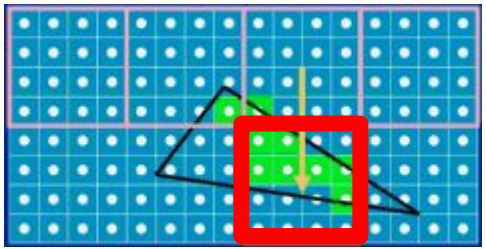
Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).



Начав с верхней вершины треугольника:



1) Скользим вправо, пока треугольник не закончится.
2) Скользим влево, пока треугольник не закончится.
3) Делаем шаг вниз и повторяем процедуру.

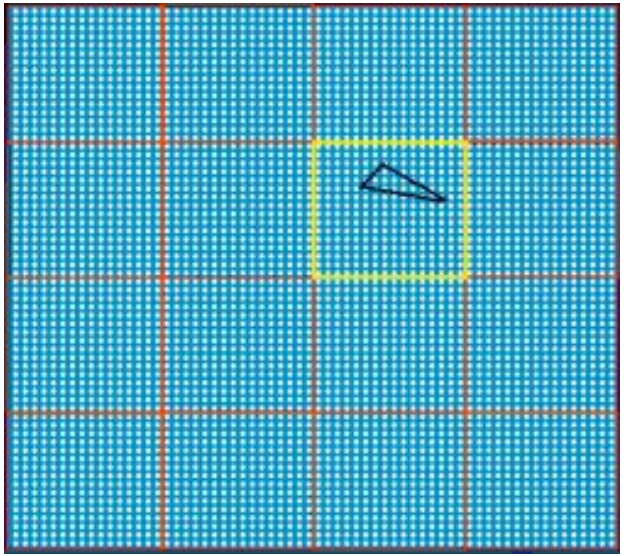


Высокоуровнево это напоминает Брезэнхема!

Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

Larrabee растеризация: **hierarchical approach, binning**

64x64 Tile: двухуровневый спуск по **4x4 решетке** (16 - ширина SIMD):

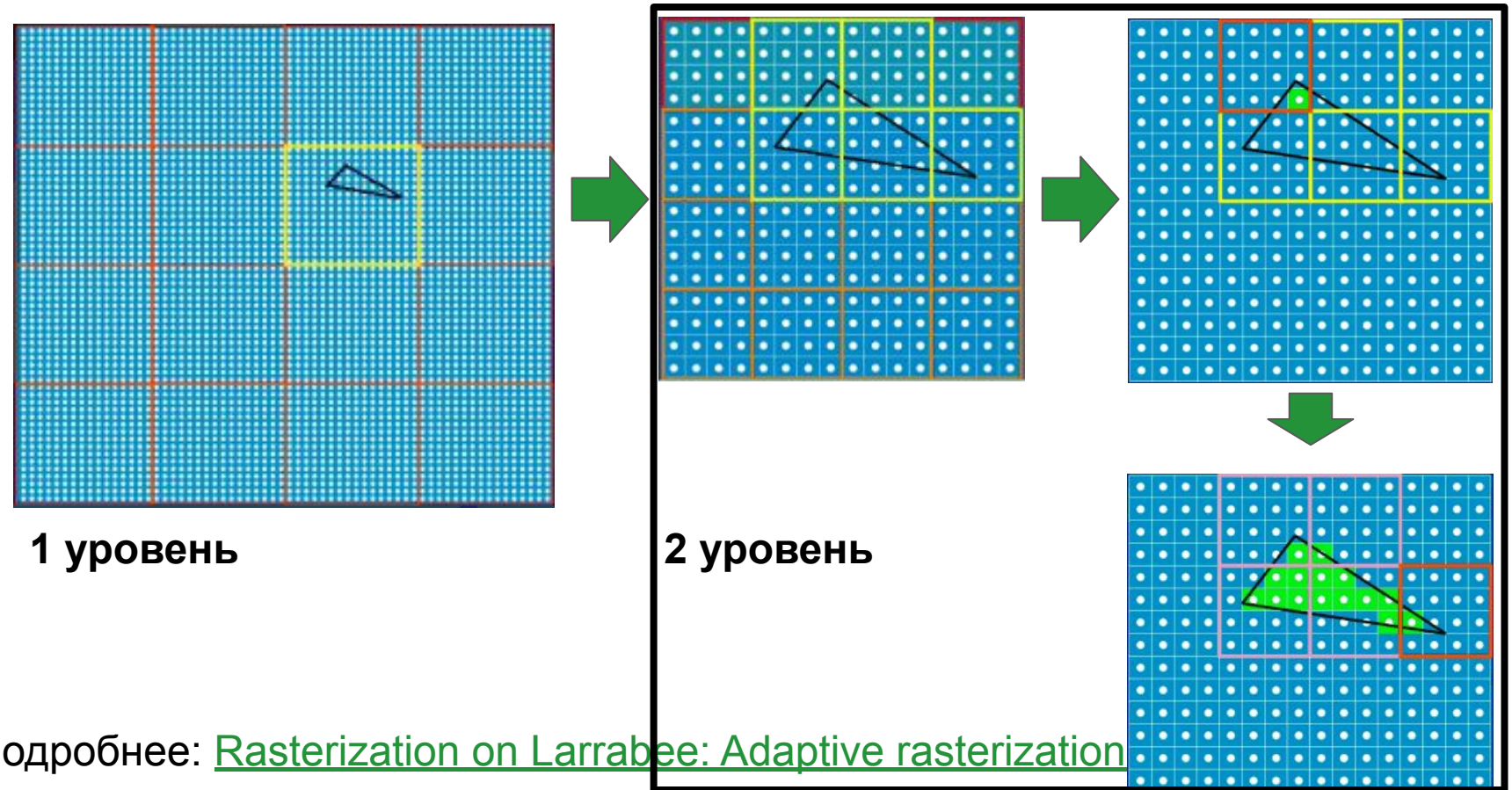


1 уровень

Подробнее: Rasterization on Larrabee: Adaptive rasterization

Larrabee растеризация: **hierarchical approach, binning**

64x64 Tile: двухуровневый спуск по **4x4 решетке** (16 - ширина SIMD):



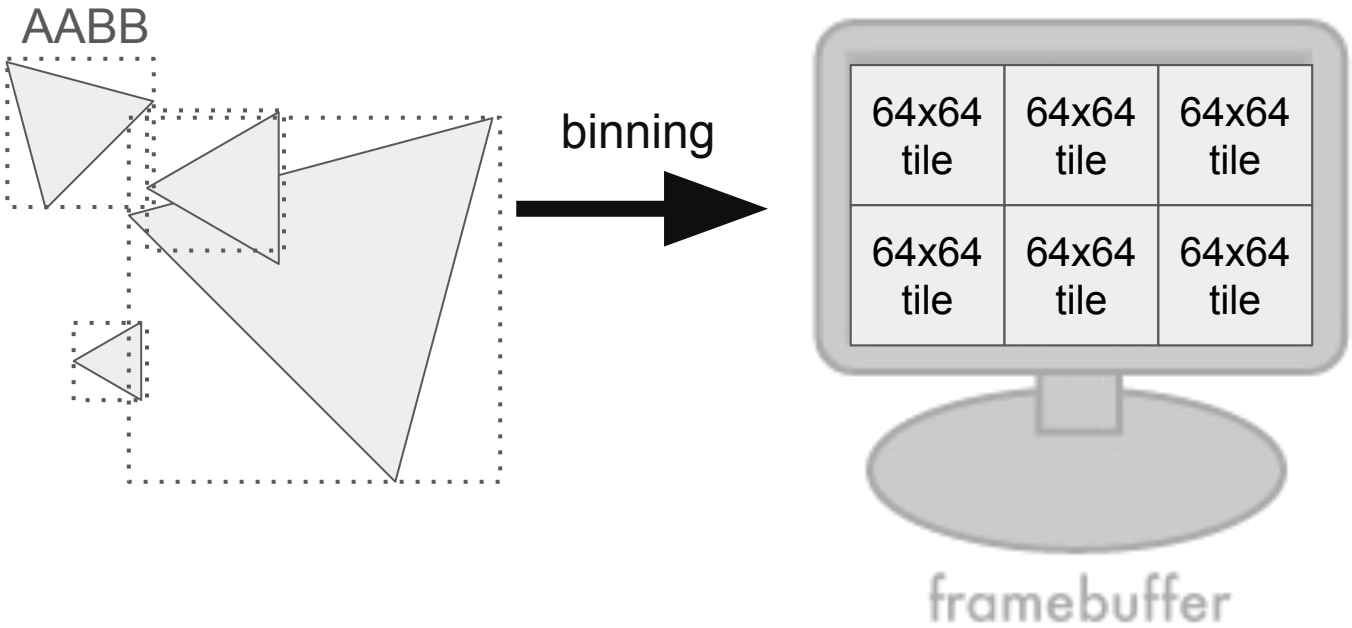
1 уровень

2 уровень

Подробнее: Rasterization on Larrabee: Adaptive rasterization

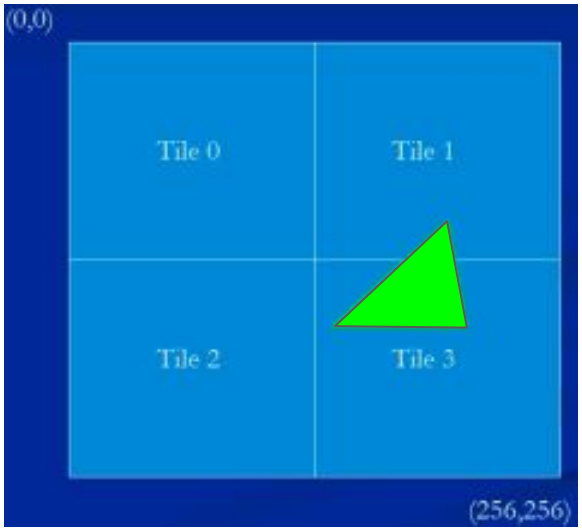
Larrabee растеризация: **hierarchical approach, binning**

Можно ли сделать **tile assignment** лучше?



Larrabee растеризация: **hierarchical approach, binning**

Как проверить какие **tiles** треугольник пересекает?



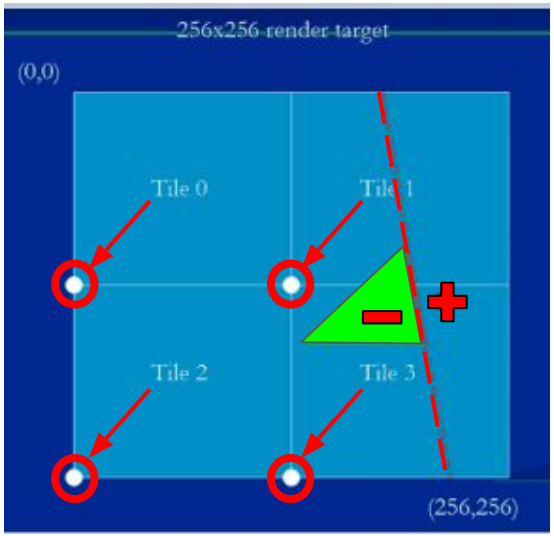
Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

Larrabee растеризация: **hierarchical approach, binning**

Trivial reject corner - угол **Tile** с наиболее **отрицательным** результатом уравнения прямой (наиболее внутри).

В разных случаях разные **trivial reject corner**.

Все - нижние левые:



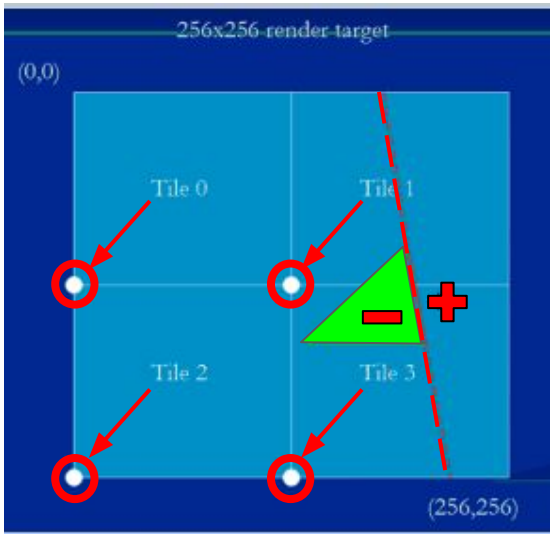
Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

Larrabee растеризация: **hierarchical approach, binning**

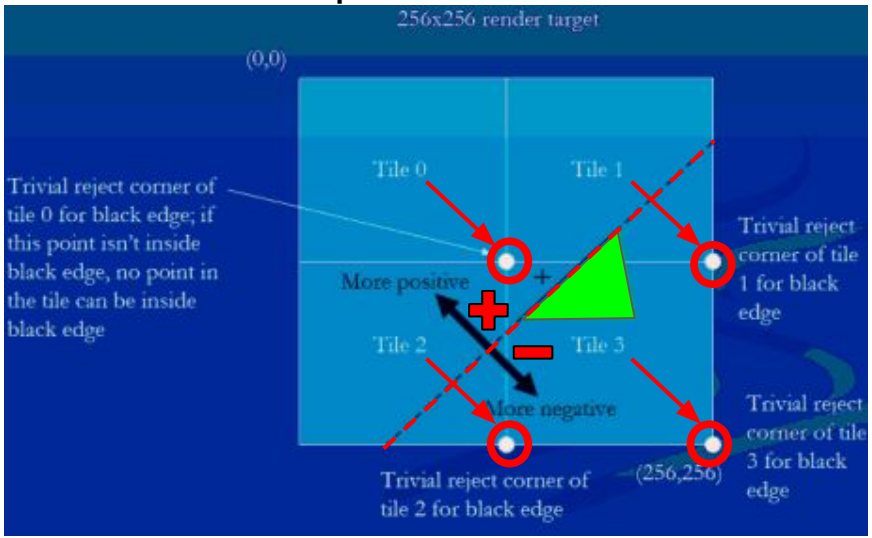
Trivial reject corner - угол **Tile** с наиболее **отрицательным** результатом уравнения прямой (наиболее внутри).

В разных случаях разные **trivial reject corner**.

Все - нижние левые:



Все - нижние правые:



Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

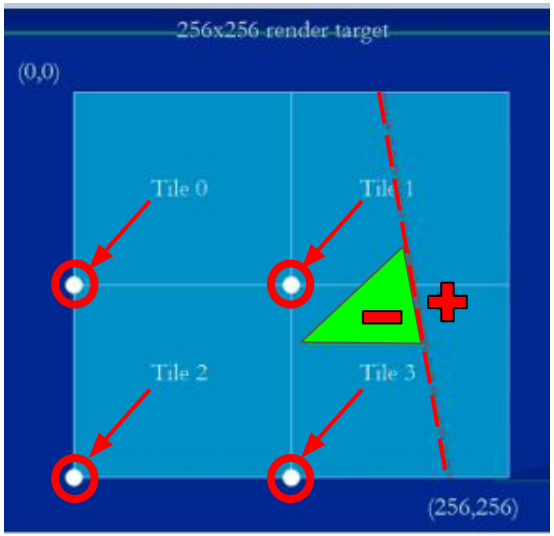
Larrabee растеризация: **hierarchical approach, binning**

Trivial reject corner - угол **Tile** с наиболее **отрицательным** результатом уравнения прямой (наиболее внутри).

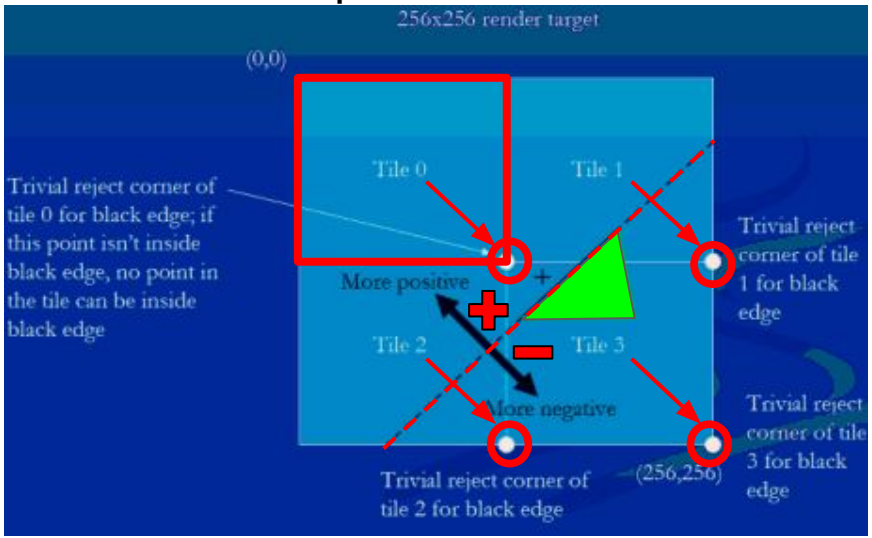
В разных случаях разные **trivial reject corner**.

Что можно сказать про **Tile 0**?

Все - нижние левые:



Все - нижние правые:



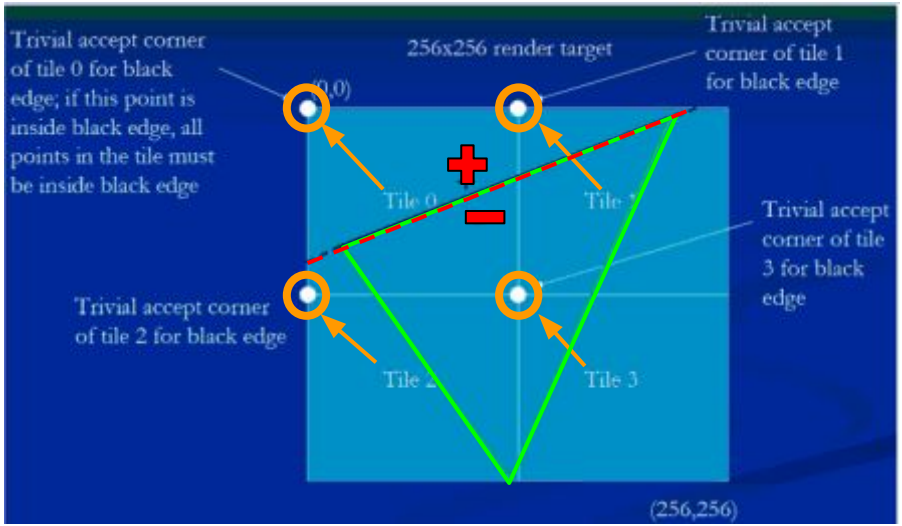
Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

Larrabee растеризация: **hierarchical approach, binning**

Trivial accept corner - угол **Tile** с наиболее **положительным** результатом уравнения прямой (наиболее снаружи).

Trivial accept corner лежит напротив **trivial reject corner**.

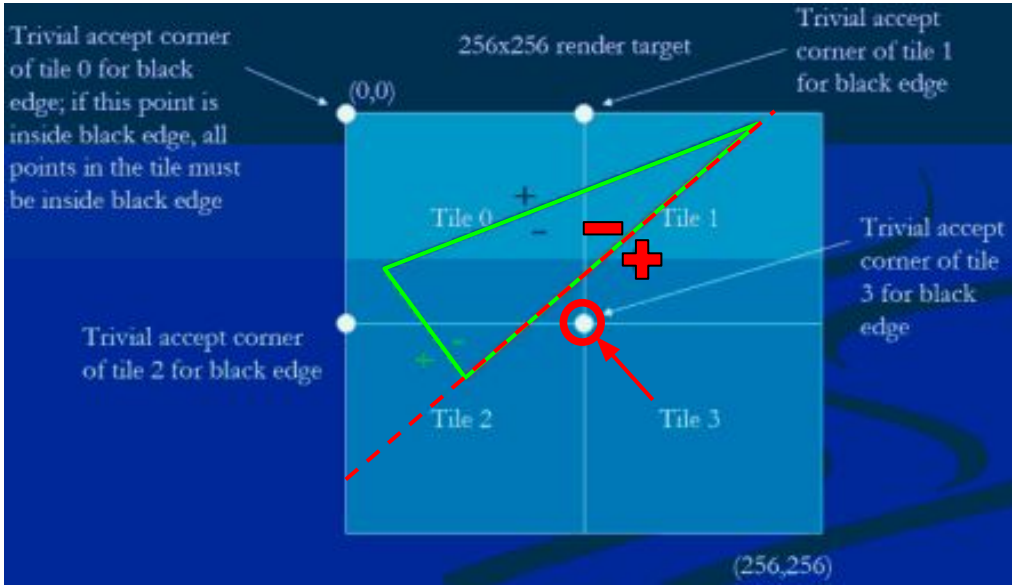
Tile 2 и Tile 3 проходят trivial accept test:



Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

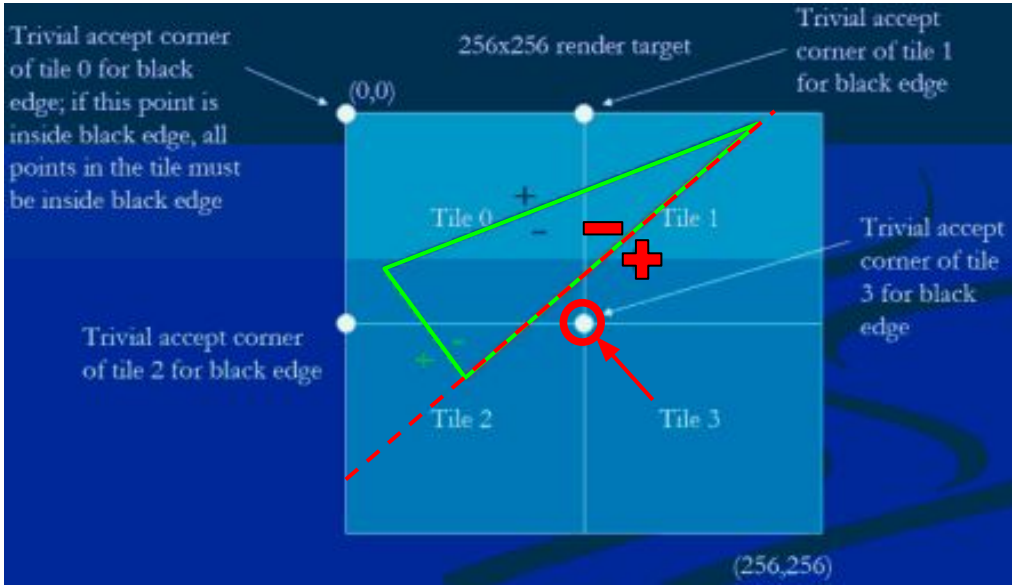
Larrabee растеризация: **hierarchical approach, binning**

Если tile проходит **trivial reject test** относительно одной из сторон треугольника - треугольник и **тайл 3 ???**.



Larrabee растеризация: **hierarchical approach, binning**

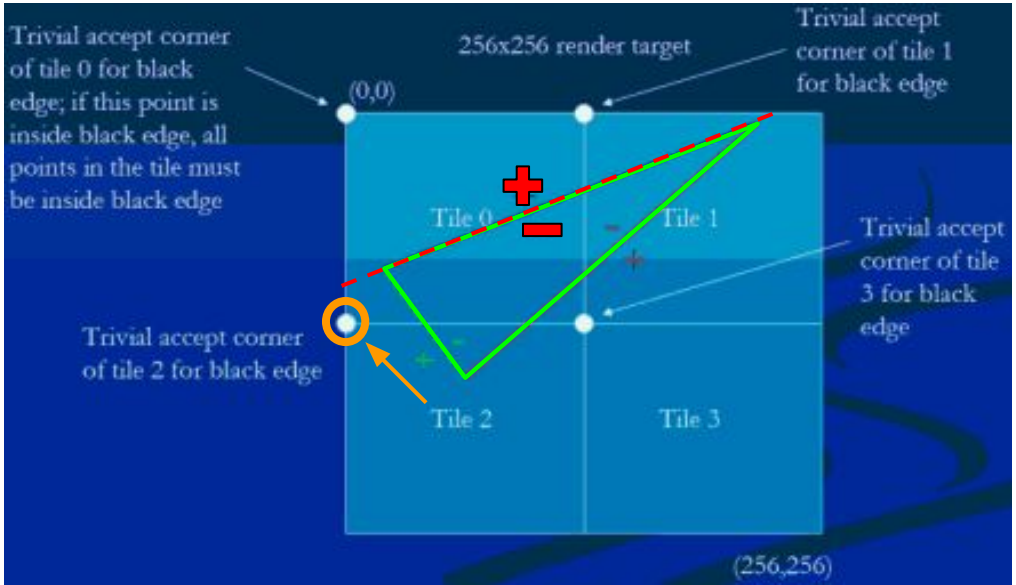
Если tile проходит **trivial reject test** относительно одной из сторон треугольника - треугольник **не пересекает tile**.



Larrabee растеризация: **hierarchical approach, binning**

Если tile проходит **trivial reject test** относительно одной из сторон треугольника - треугольник **не пересекает tile**.

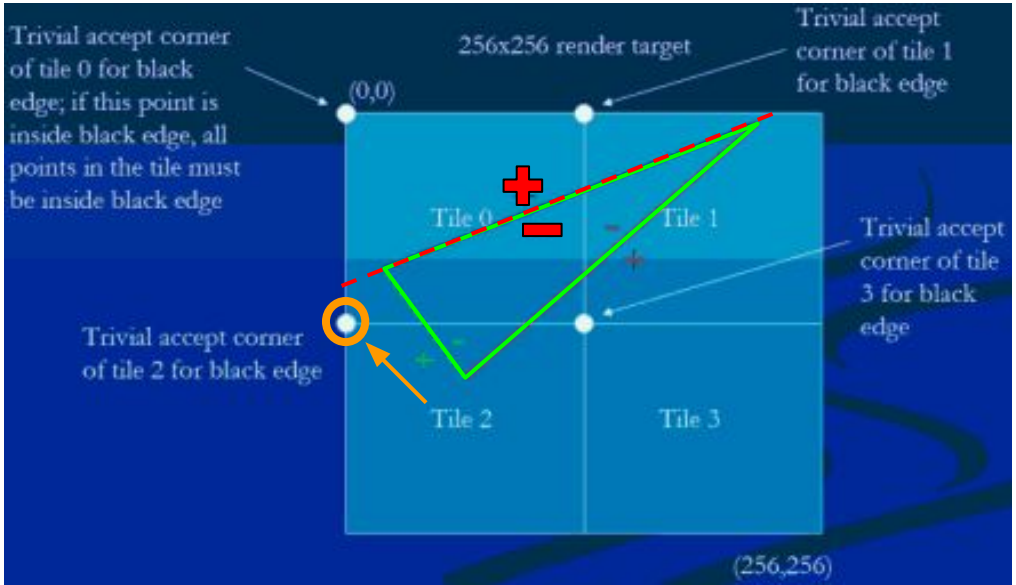
Если tile проходит **trivial accept test** относительно одной из сторон треугольника - треугольник и **тайл 2 ???**.



Larrabee растеризация: **hierarchical approach, binning**

Если tile проходит **trivial reject test** относительно одной из сторон треугольника - треугольник **не пересекает tile**.

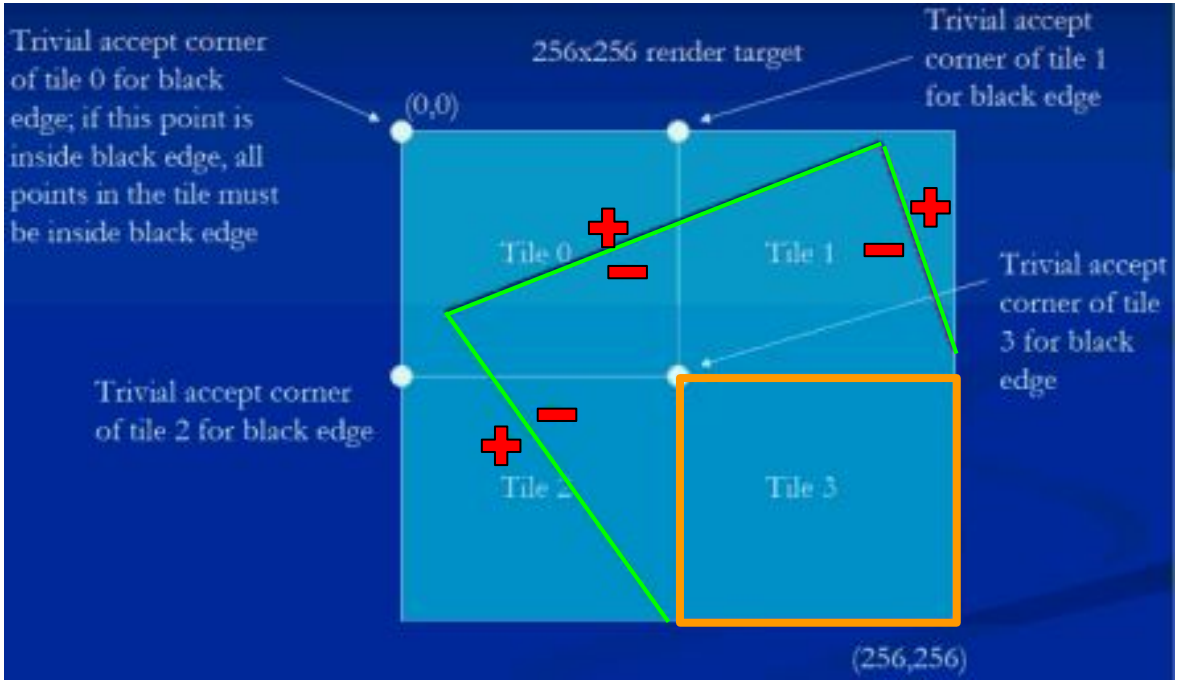
Если tile проходит **trivial accept test** относительно одной из сторон треугольника - только относительно этой стороны не нужны дальнейшие проверки, но про остальные стороны ничего неизвестно:



Larrabee растеризация: **hierarchical approach, binning**

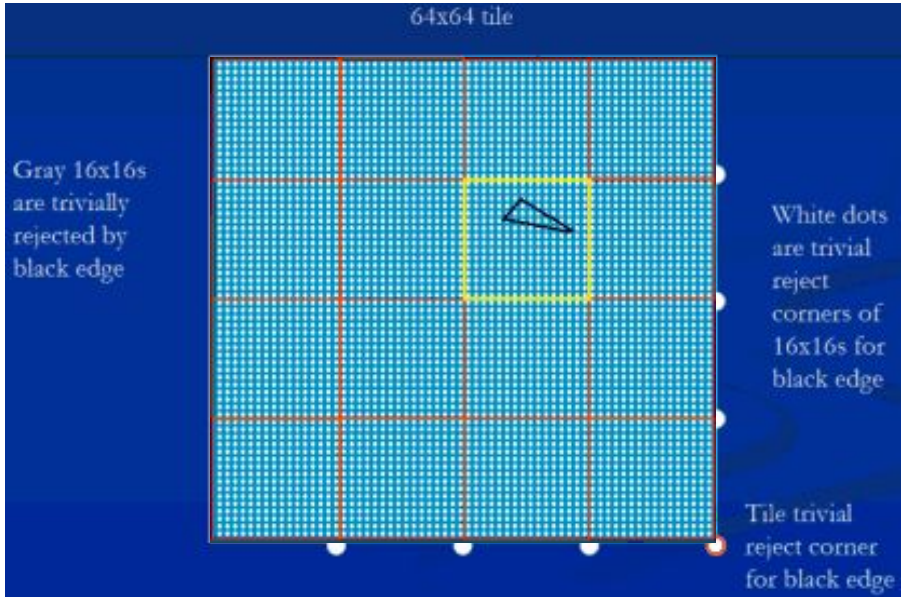
Но иногда дарят подарок!

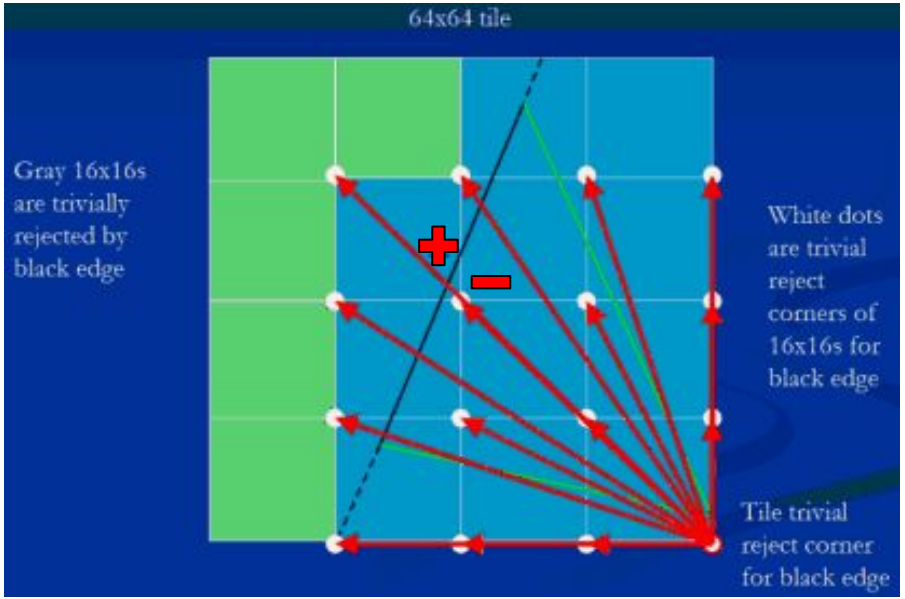
Иногда tile проходит **trivial accept test** относительно **всех** трех ребер:



И это означает …?

Larrabee растеризация: **trivial accept/reject test (для 64x64)**

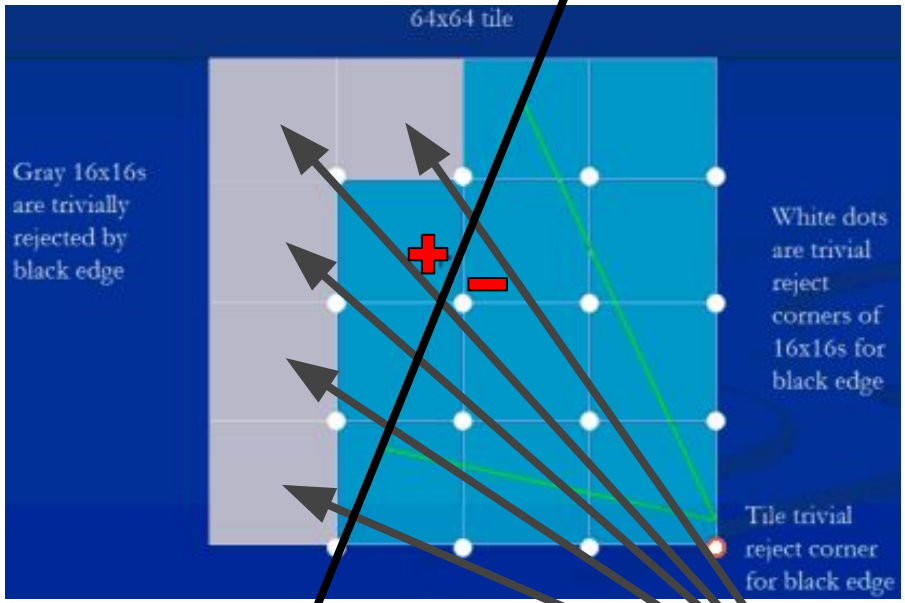


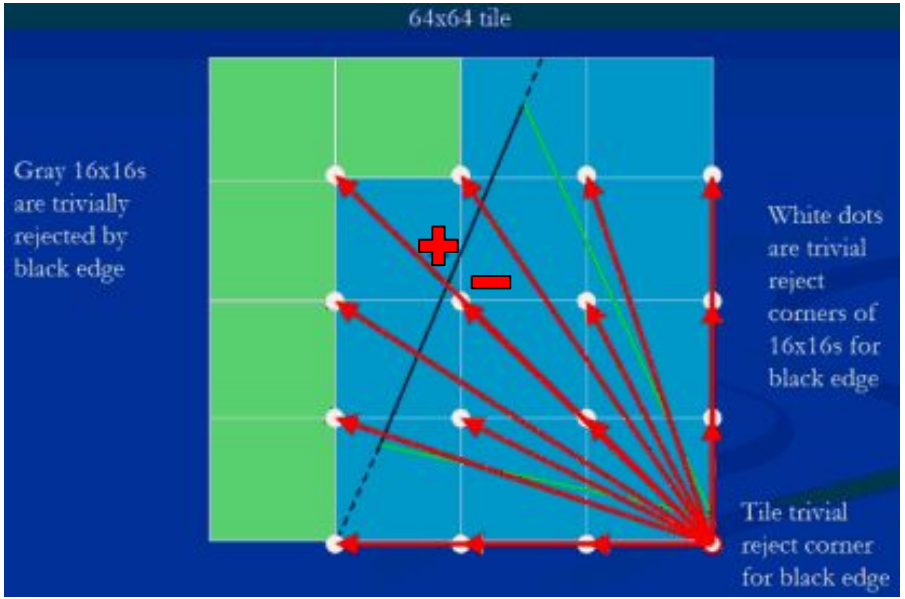


Аналогично проводим оба вида тестов чтобы выкинуть лишнюю работу и собрать подарки. Ложится на **SIMD-16**.

Подробнее: Intra-tile Rasterization: 16×16 Blocks

Larrabee растеризация: **trivial accept/reject test (для 64x64)**



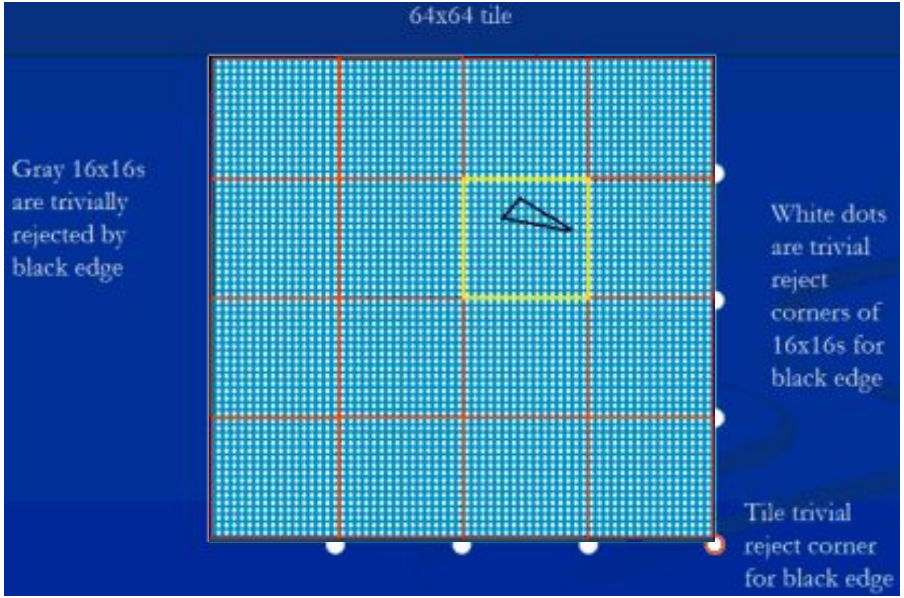


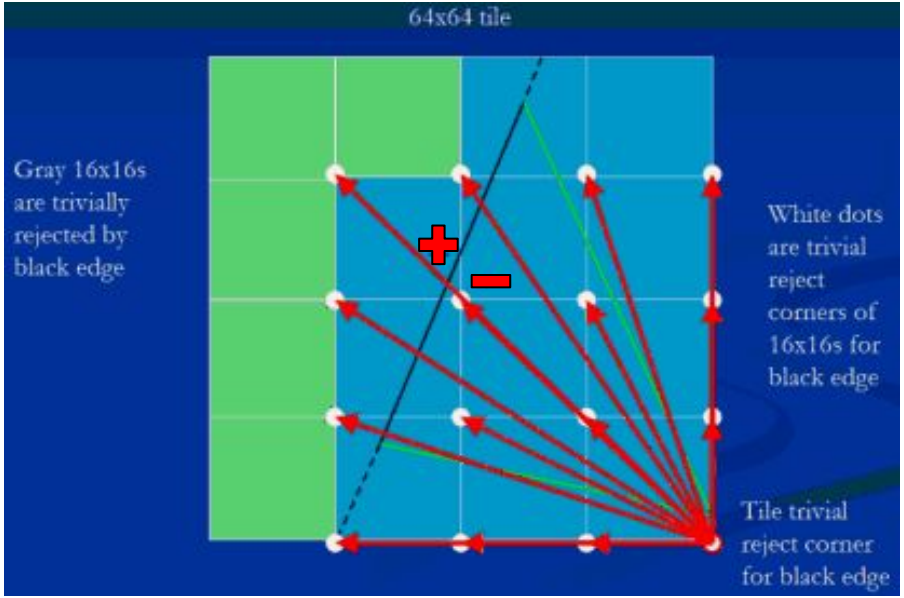
Аналогично проводим оба вида тестов чтобы выкинуть лишнюю работу и собрать подарки. Ложится на **SIMD-16**.

Серые тайлы - отсеклись **черной стороной** по **trivial reject test**.

Подробнее: [Intra-tile Rasterization: 16×16 Blocks]

Larrabee растеризация: **trivial accept/reject test (для 64x64)**



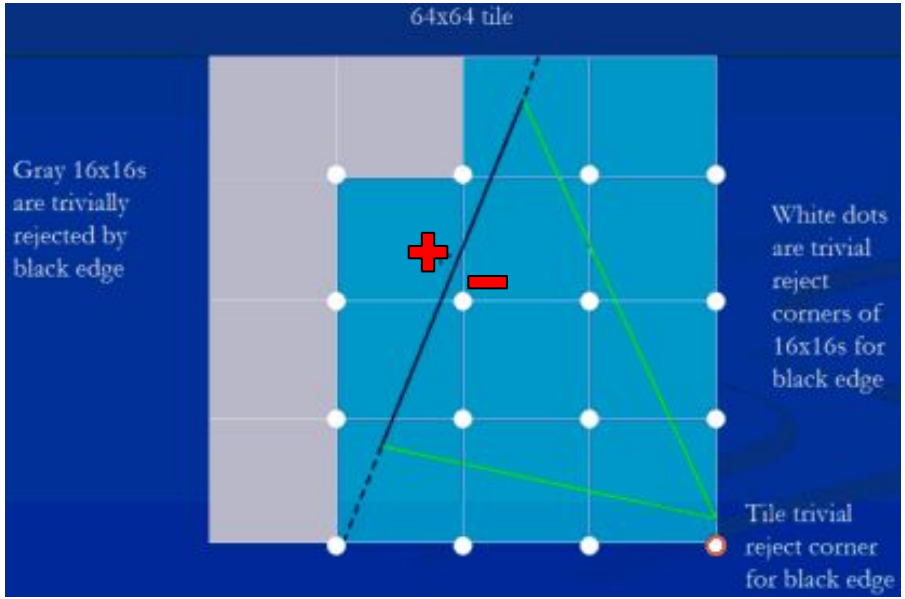


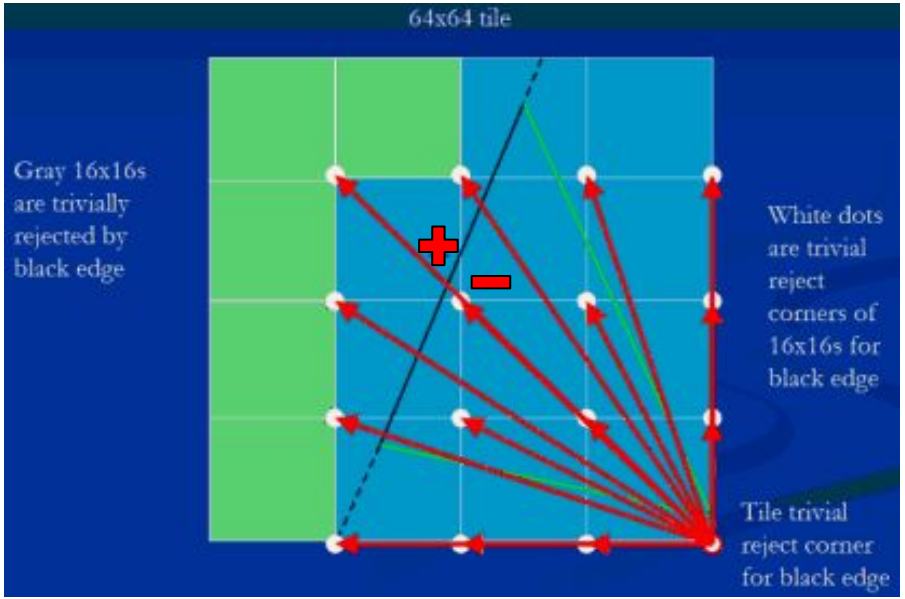
Аналогично проводим оба вида тестов чтобы выкинуть лишнюю работу и собрать подарки. Ложится на **SIMD-16**.

Затем на **по-пиксельном** уровне **(для 4x4)** делаем только trivial reject test.

Подробнее: Intra-tile Rasterization: 16×16 Blocks

Larrabee растеризация: **trivial accept/reject test (для 64x64)**



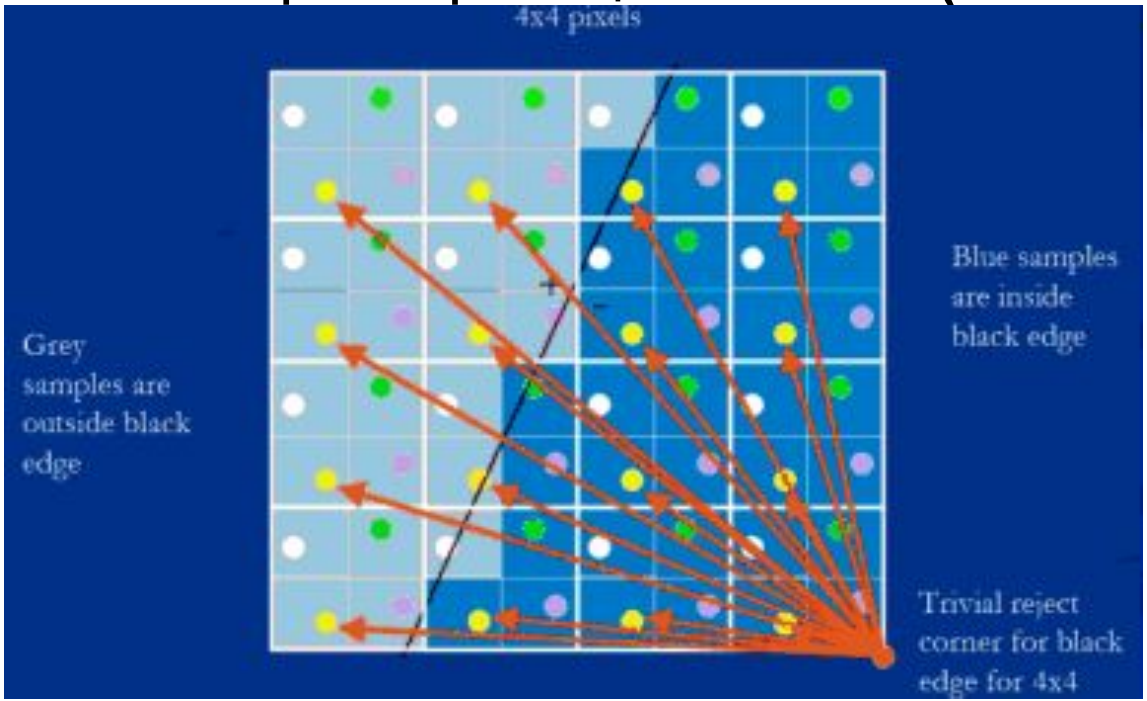


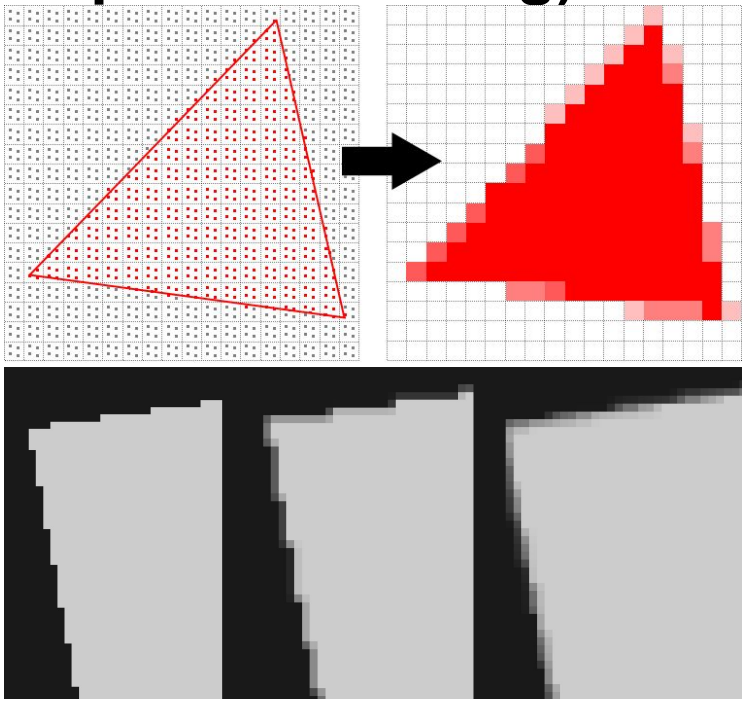
Аналогично проводим оба вида тестов чтобы выкинуть лишнюю работу и собрать подарки. Ложится на **SIMD-16**.

Затем на **по-пиксельном** уровне **(для 4x4)** делаем только trivial reject test.

Подробнее: [Intra-tile Rasterization: 16×16 Blocks]

Larrabee растеризация: **MSAA (multisample antialiasing)**





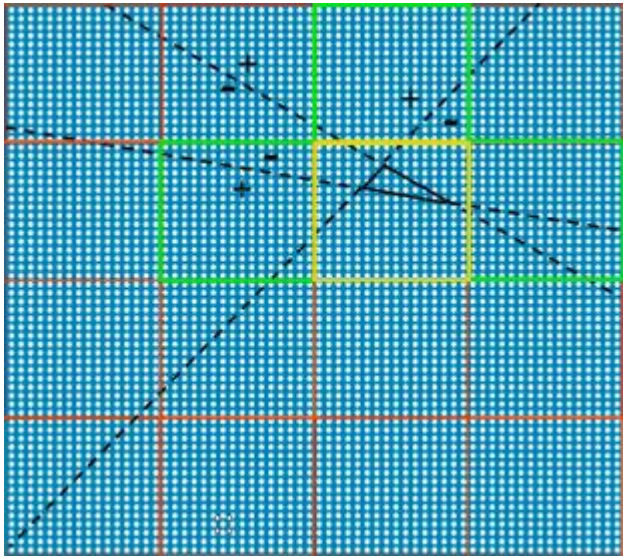
Для **trivially accepted tiles** - нет дополнительных операций.

Для **partially accepted 4x4 tiles** - просто еще один прогон trivial reject corner по 16 сэмплов.

Подробнее: Intra-tile Rasterization: 16×16 Blocks

Larrabee пример: Level 1

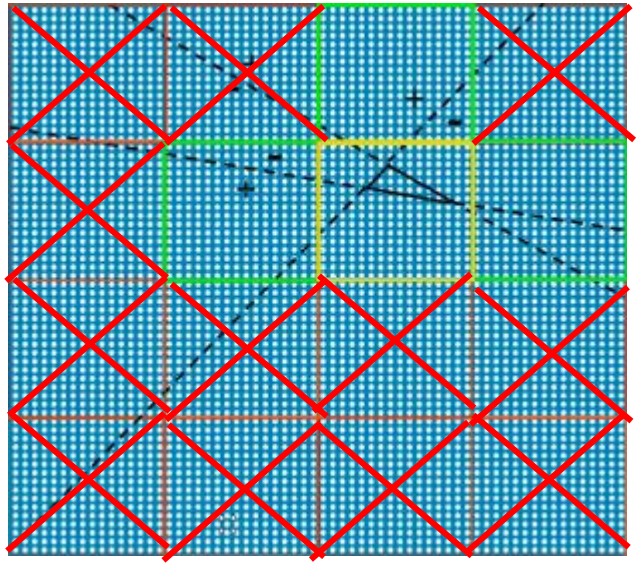
Подробнее: Larrabee: Putting It All Together



Есть 4x4 сетка из 64x64 тайлов.

Larrabee пример: Level 1

Подробнее: Larrabee: Putting It All Together



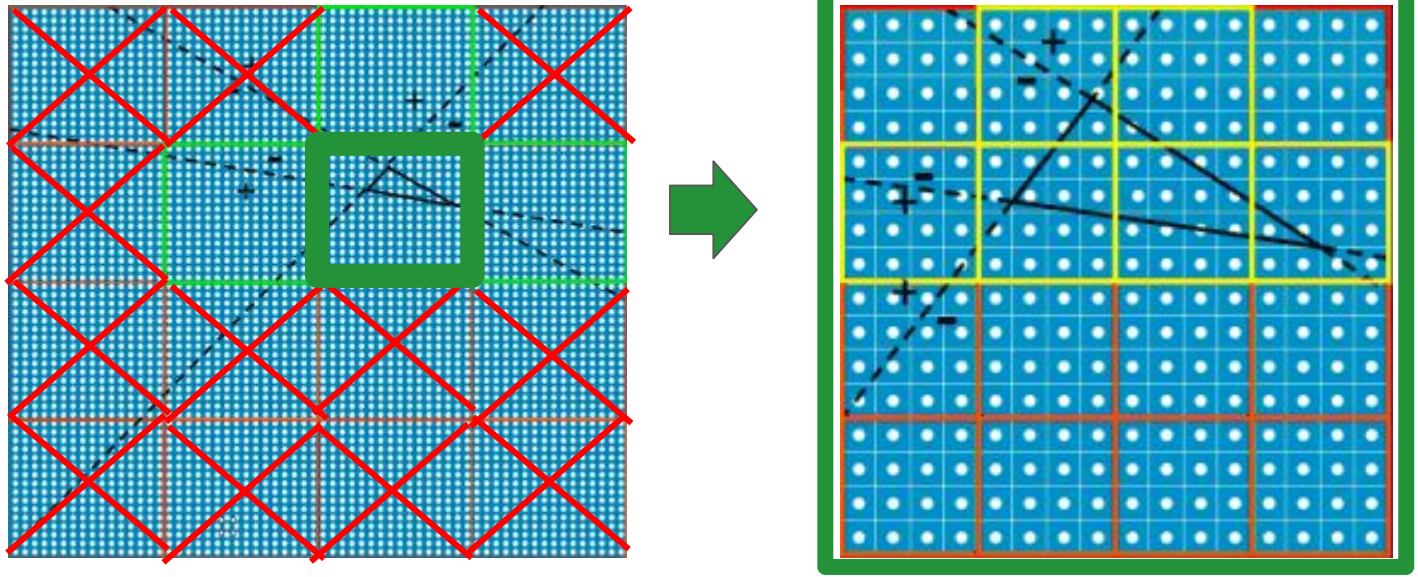
Есть 4x4 сетка из 64x64 тайлов.

12 тайлов отсеклись по trivial reject test.

(trivial accept test случается только для больших треугольников)

Larrabee пример: Level 1

Подробнее: Larrabee: Putting It All Together



Есть 4x4 сетка из 64x64 тайлов.

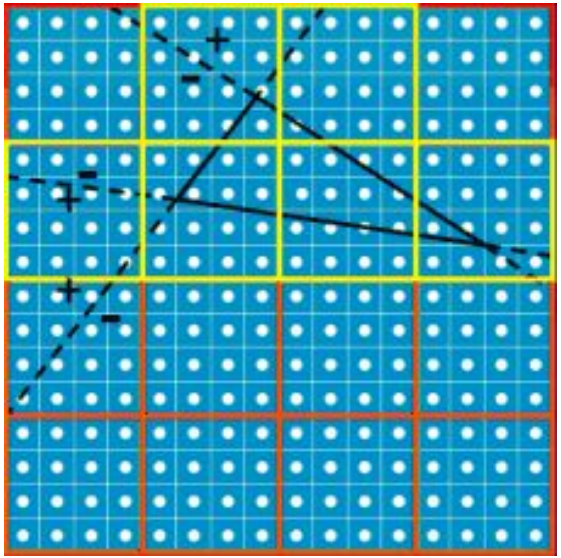
12 тайлов отсеклись по trivial reject test.

(trivial accept test случается только для больших треугольников)

Спускаемся в оставшиеся 4 тайла.

Larrabee пример: Level 2

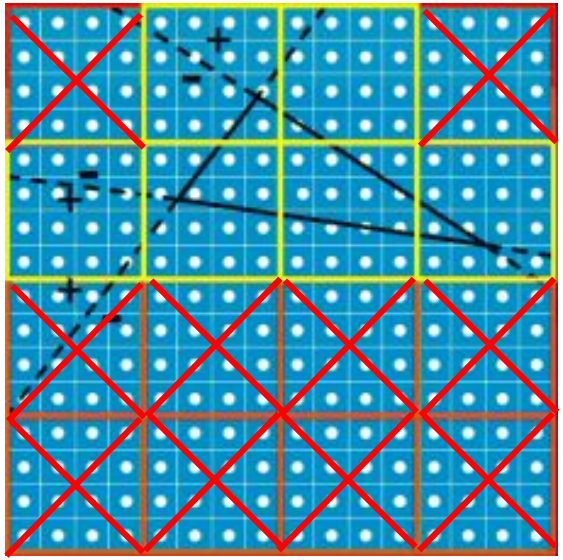
Подробнее: Larrabee: Putting It All Together



Есть 4x4 сетка из 16x16 тайлов.

Larrabee пример: Level 2

Подробнее: Larrabee: Putting It All Together

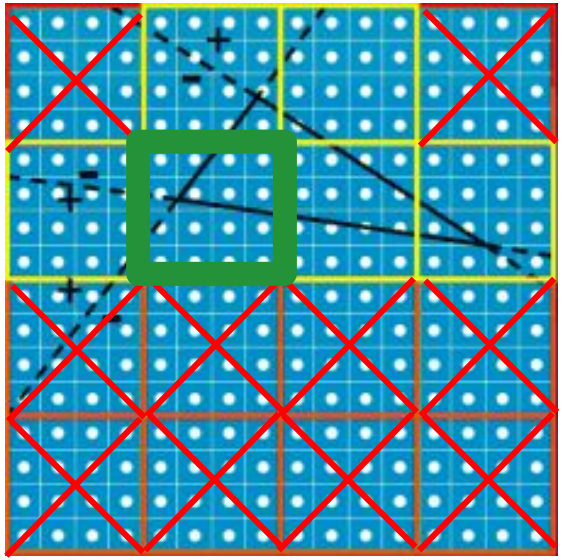


Есть 4x4 сетка из 16x16 тайлов.

10 тайлов отсеклись по trivial reject test.

Larrabee пример: Level 2

Подробнее: Larrabee: Putting It All Together



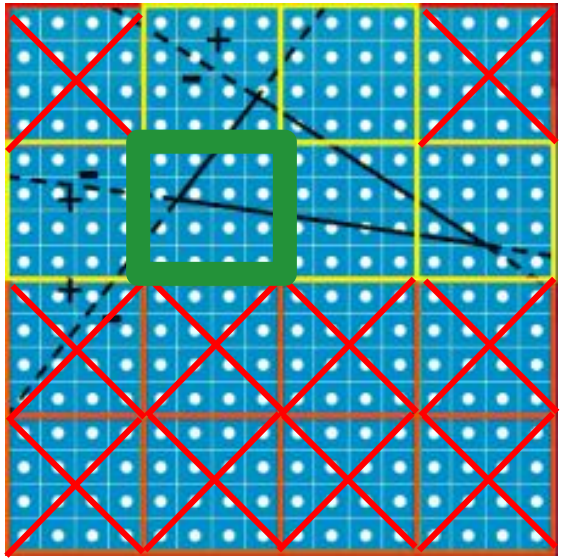
Есть 4x4 сетка из 16x16 тайлов.

10 тайлов отсеклись по trivial reject test.

Спускаемся в оставшиеся 6 тайлов.

Larrabee пример: Level 2

Подробнее: Larrabee: Putting It All Together



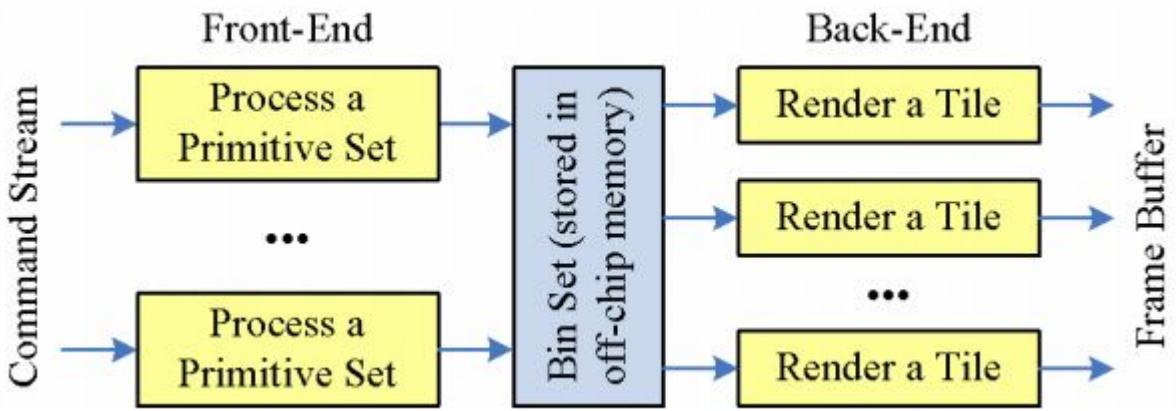
Есть 4x4 сетка из 16x16 тайлов.

10 тайлов отсеклись по trivial reject test.

Спускаемся в оставшиеся 6 тайлов.

Level 3: Наконец в блоке 4x4 пикселей считаем маску.

Larrabee



1) Для каждого треугольника находятся пересекающиеся корзины-тайлы и для них вычисляются маски покрытия (или флажок **trivial accept**).
2) В каждой корзине для всех треугольников выполняется фрагментный шейдер.

Размер корзины таков, чтобы ее tile влезал в L2 cache (128x128 для 256 KB).

Подробнее: Larrabee: A Many-Core x86 Architecture for Visual Computing

Larrabee

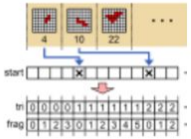
- Dr.Dobb's: Rasterization on Larrabee
- Larrabee: A Many-Core x86 Architecture for Visual Computing
- Why didn't Larrabee fail?
- An Inconvenient Truth: Intel Larrabee story revealed


Перерыв!
Streaming
Pubiprocestor
VIDIA

Глава 3: **cudaraster**

Софтварная растеризация на CUDA

High-Performance Software Rasterization on GPUs



In this paper, we implement an efficient, completely software-based graphics pipeline on a GPU. Unlike previous approaches, we obey ordering constraints imposed by current graphics APIs, guarantee hole-free rasterization, and support multisample antialiasing. Our goal is to examine the performance implications of not exploiting the fixed-function graphics pipeline, and to discern which additional hardware support would benefit software-based graphics the most. We present significant improvements over previous work in terms of scalability, performance, and capabilities. Our pipeline is malleable and easy to extend, and we demonstrate that in a wide variety of test cases its performance is within a factor of 2–8x compared to the hardware graphics pipeline on a top of the line GPU. Our implementation is open sourced and available at http://code.google.com/p/cudaraster/.

Authors: Samuli Laine
Tero Karras

Tero Karras

Publication Date: Monday, August 1, 2011

Published in: High-Performance Graphics 2011

Research Area: Computer Graphics

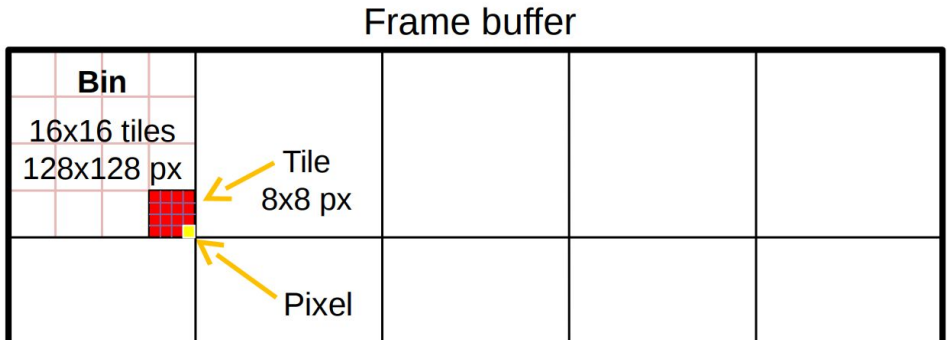
External Links: Source code

Uploaded Files: laine2011hpg_paper.pdf 7.93 MB
laine2011hpg_slides.pptx 1.35 MB

Подробнее: cudaraster

cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

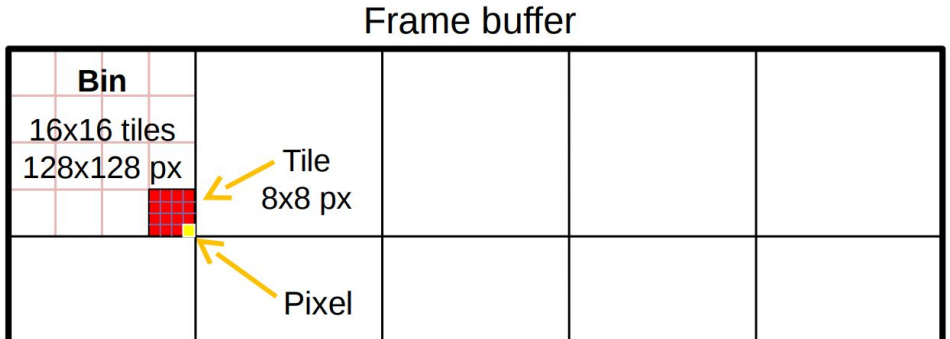


Подробнее: [cudaraster]

cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

- Массовый параллелизм. Хорошая гранулярность и балансировка рабочей нагрузки.

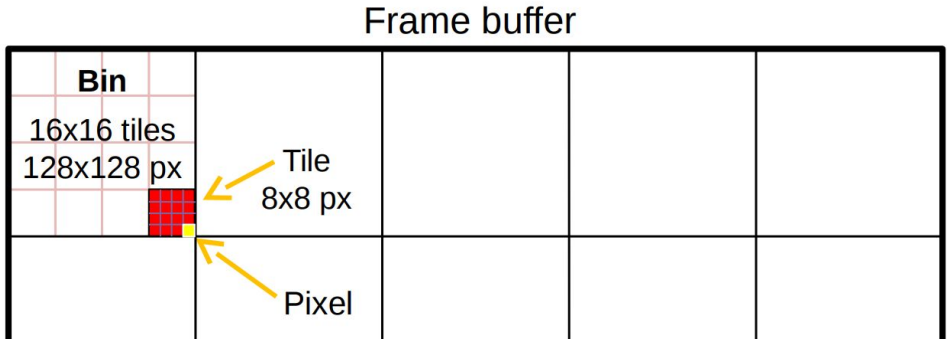


Подробнее: cudaraster

cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

- Массовый параллелизм. Хорошая гранулярность и балансировка рабочей нагрузки.
- Минимизировать число синхронизаций между Streaming Multiprocessor (**SM**, в статье упоминаются как **CTA** = **C**ooperative **T**hread **A**rrays). Т.е. минимизировать число глобальных атомарных операций.



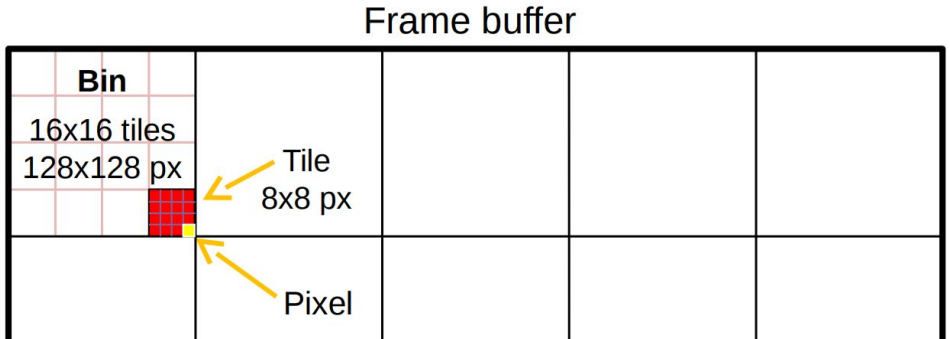
Подробнее: cudaraster

cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

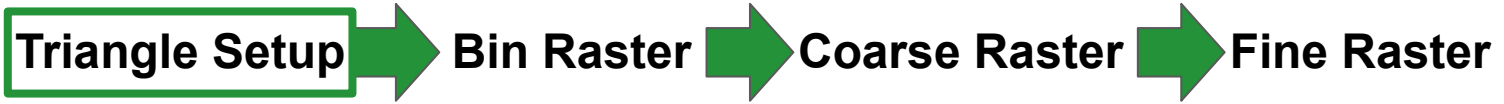
- Массовый параллелизм. Хорошая гранулярность и балансировка рабочей нагрузки.
- Минимизировать число синхронизаций между Streaming Multiprocessor (**SM**, в статье упоминаются как **CTA** = **C**ooperative **T**hread **A**rrays). Т.е. минимизировать число глобальных атомарных операций.

Общая идея все та же - распределить треугольники по корзинам и обработать по тайлам:



Подробнее: cudaraster

cudaraster

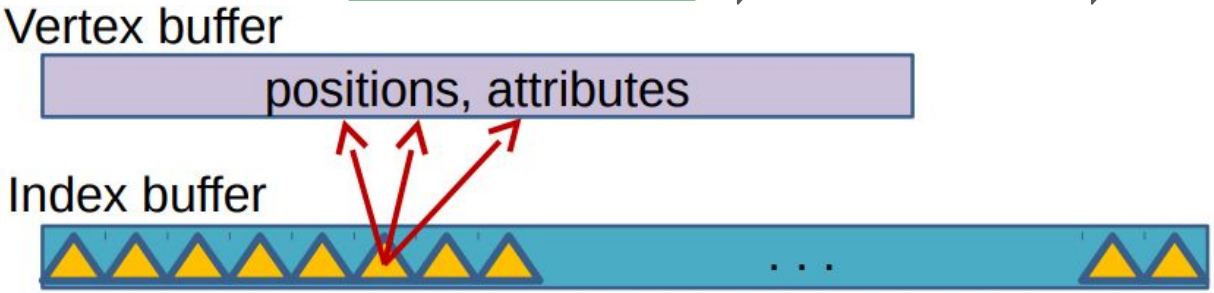


Vertex buffer

positions, attributes

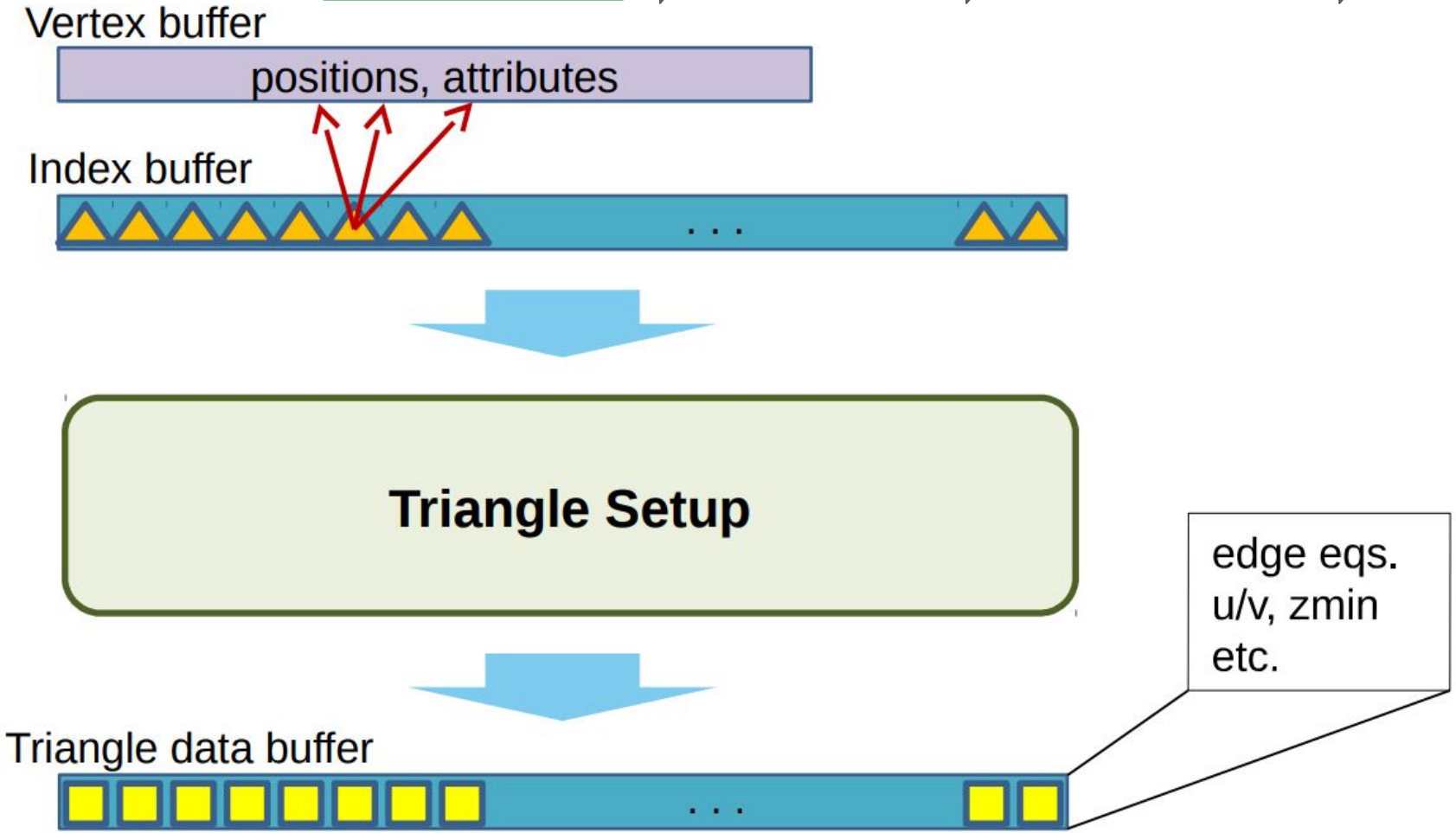
cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

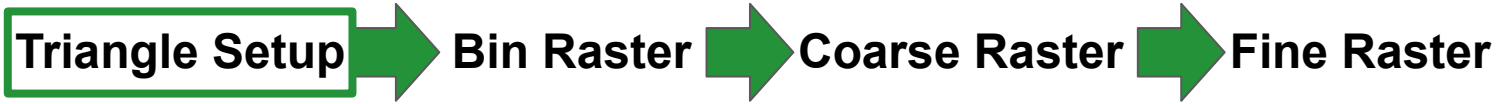


cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**



cudaraster



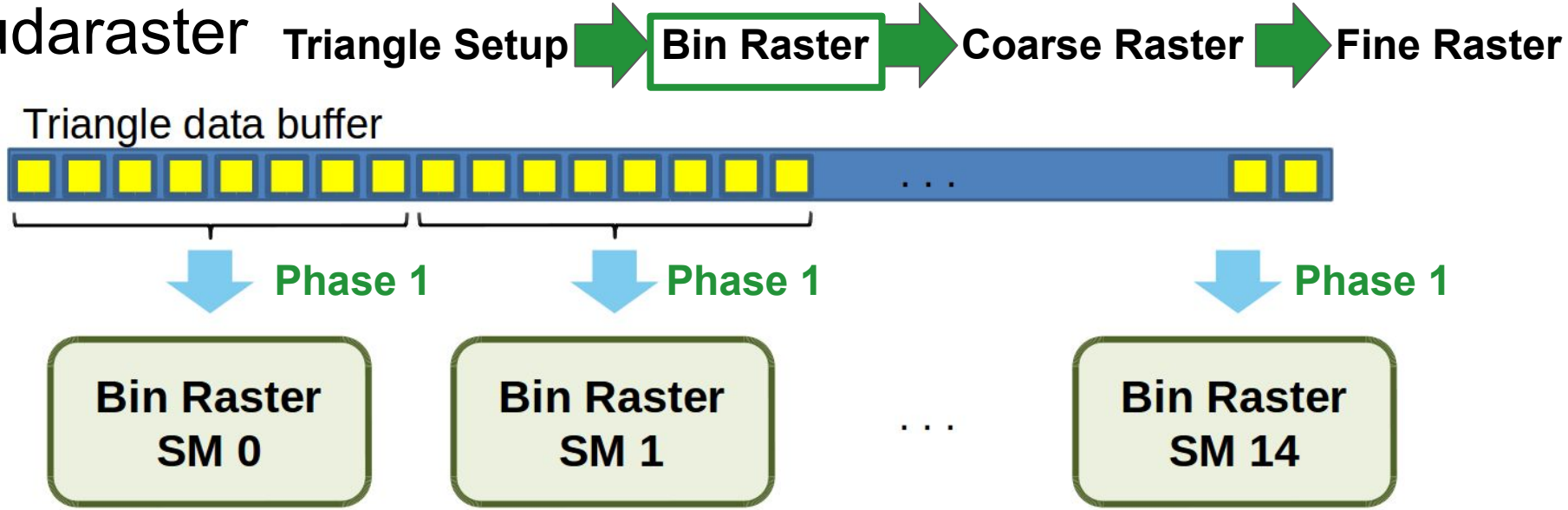
Множество **culling tests** для каждого треугольника:

- Если площадь ноль - выкинули
- Если площадь отрицательная (если backface culling) - выкинули
- Если bounding box между рядов семплов - выкинули
  (тонкие вертикальные/горизонтальные - например наблюдаемые под прямым углом стены вдалеке)

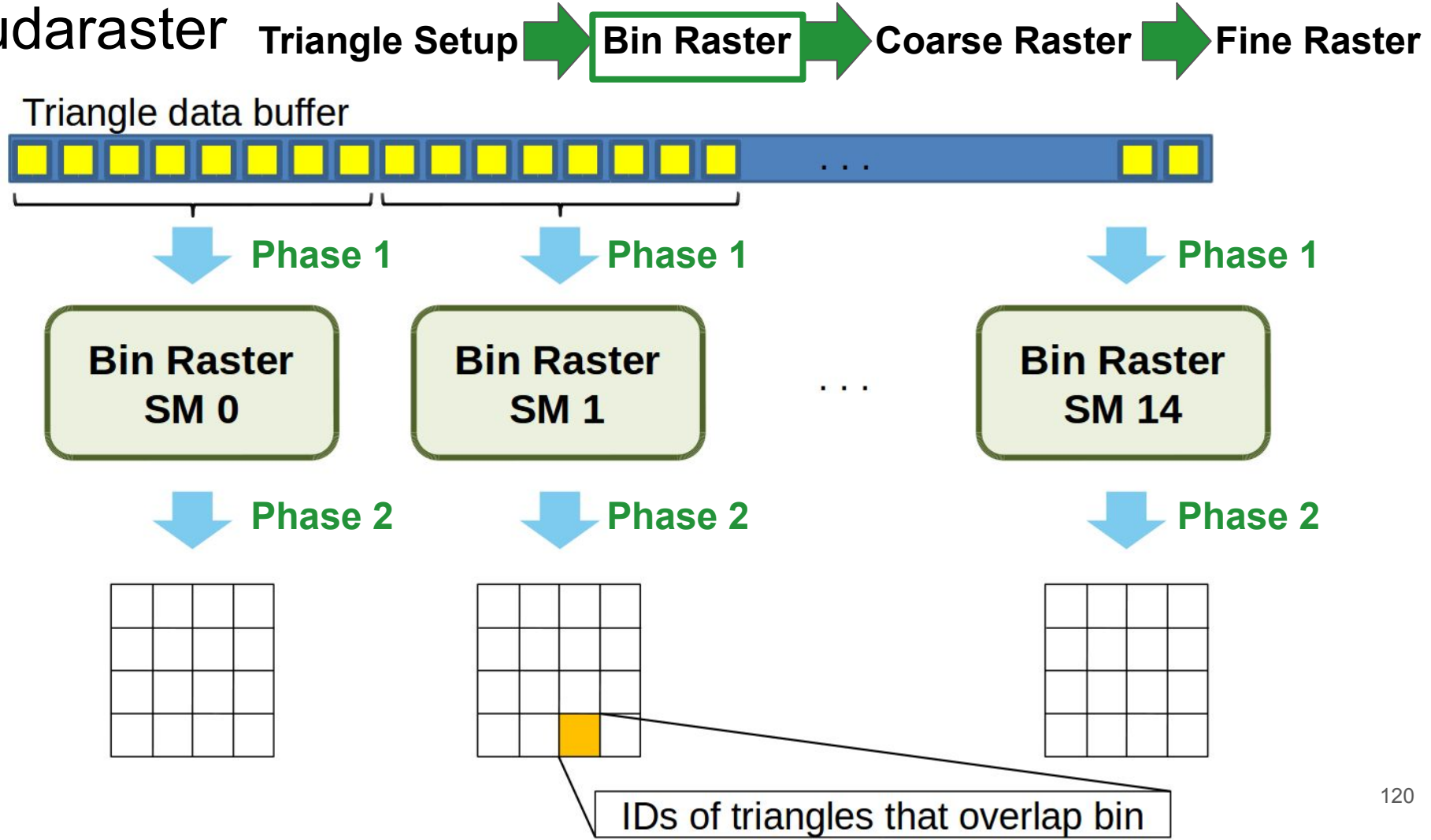
cudaraster

Triangle data buffer

cudaraster

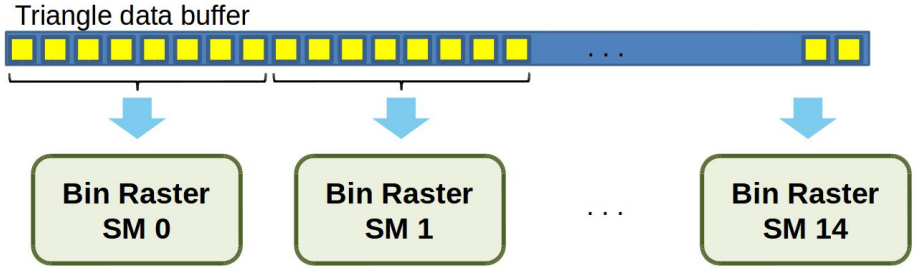

Triangle Setup
Bin Raster
Coarse Raster
Fine Raster
Triangle data buffer
. . .
Phase 1
Phase 1
Phase 1
Bin Raster SM 0
Bin Raster SM 1
. . .
Bin Raster SM 14

cudaraster


Triangle Setup
Bin Raster
Coarse Raster
Fine Raster
Triangle data buffer
Phase 1
Phase 1
Phase 1
Bin Raster SM 0
Bin Raster SM 1
Bin Raster SM 14
Phase 2
Phase 2
Phase 2
IDs of triangles that overlap bin

cudaraster

1) **SM** бронирует очередной пакет треугольников (**atomic**, 16K треугольников)
2) Забирает из пакета часть - 512 треугольников
3) Выполняет **culling/clipping** и в локальной памяти префиксными суммами определяет сколько треугольников эффективно получилось
4) Пока треугольников эффективно оказывается мало - повторять шаги

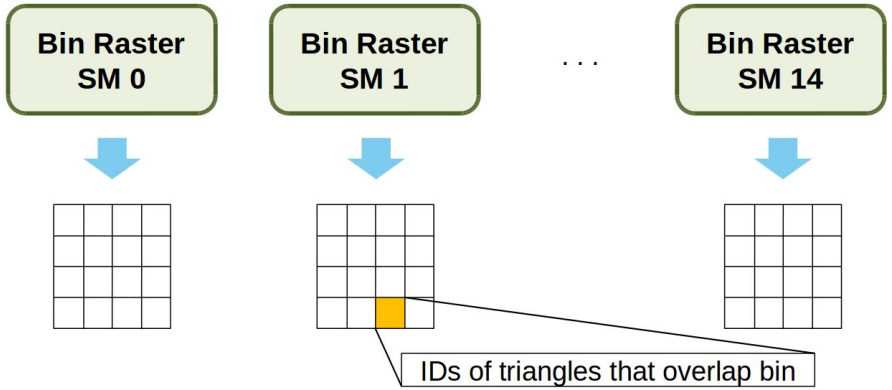


cudaraster

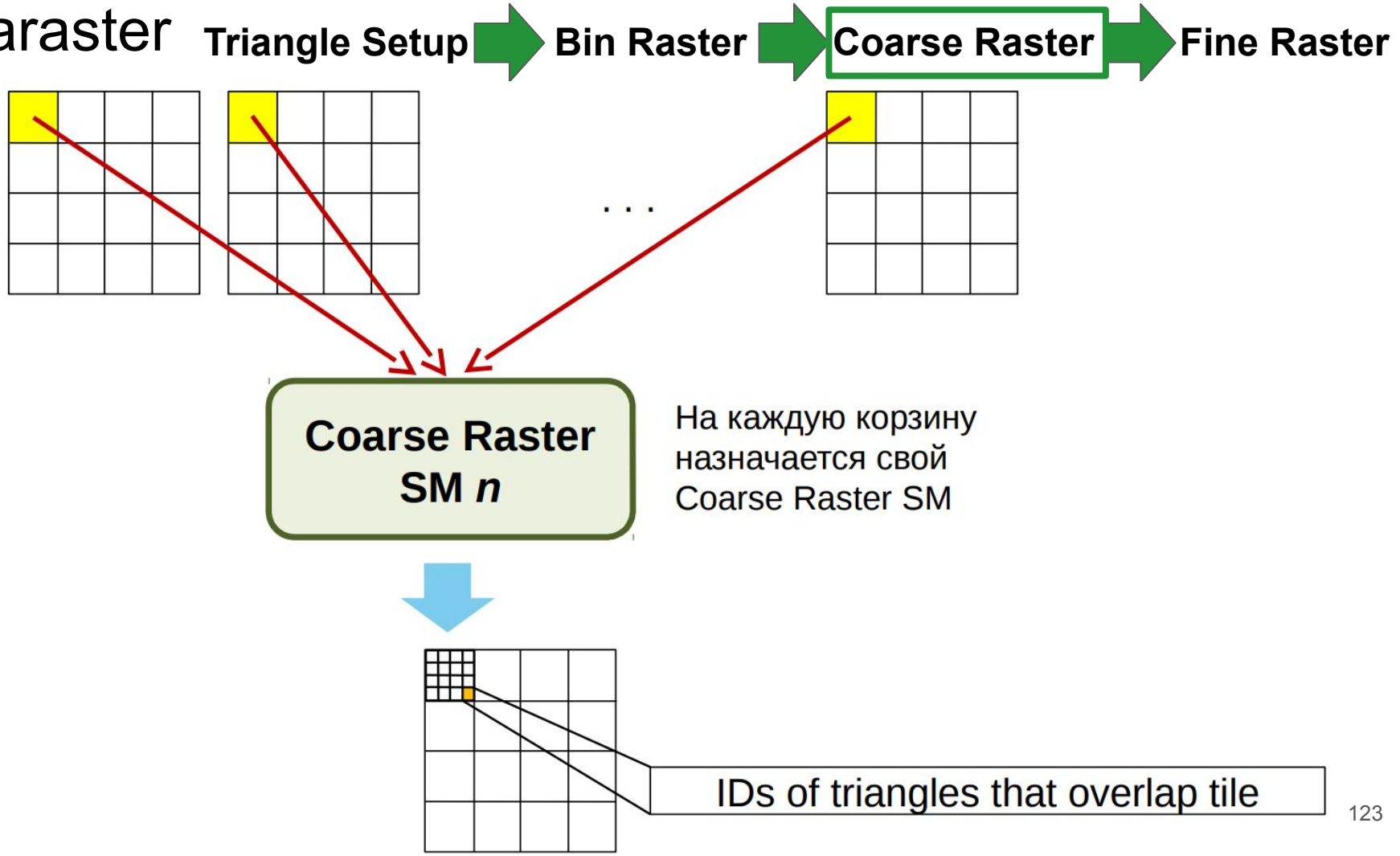
Когда треугольников подгружено достаточно для утилизации всех потоков - каждый треугольник обрабатывается своим потоком:

- Определяются **корзины** пересекающие треугольник
- Оптимизированный код для случая треугольника 2x2 пикселя и меньше

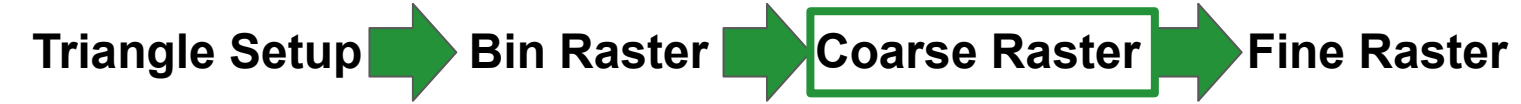
Результат пишется в специально выделенную для данного SM очередь (соответственно не нужна синхронизация между разными SM)



cudaraster


Triangle Setup
Bin Raster
Coarse Raster
Fine Raster
. . .
Coarse Raster
SM n
На каждую корзину
назначается свой
Coarse Raster SM
IDs of triangles that overlap tile

cudaraster



Подгружаем треугольники из одной и той же корзины из всех очередей (т.е. результаты всех Bin Raster **SM**s).

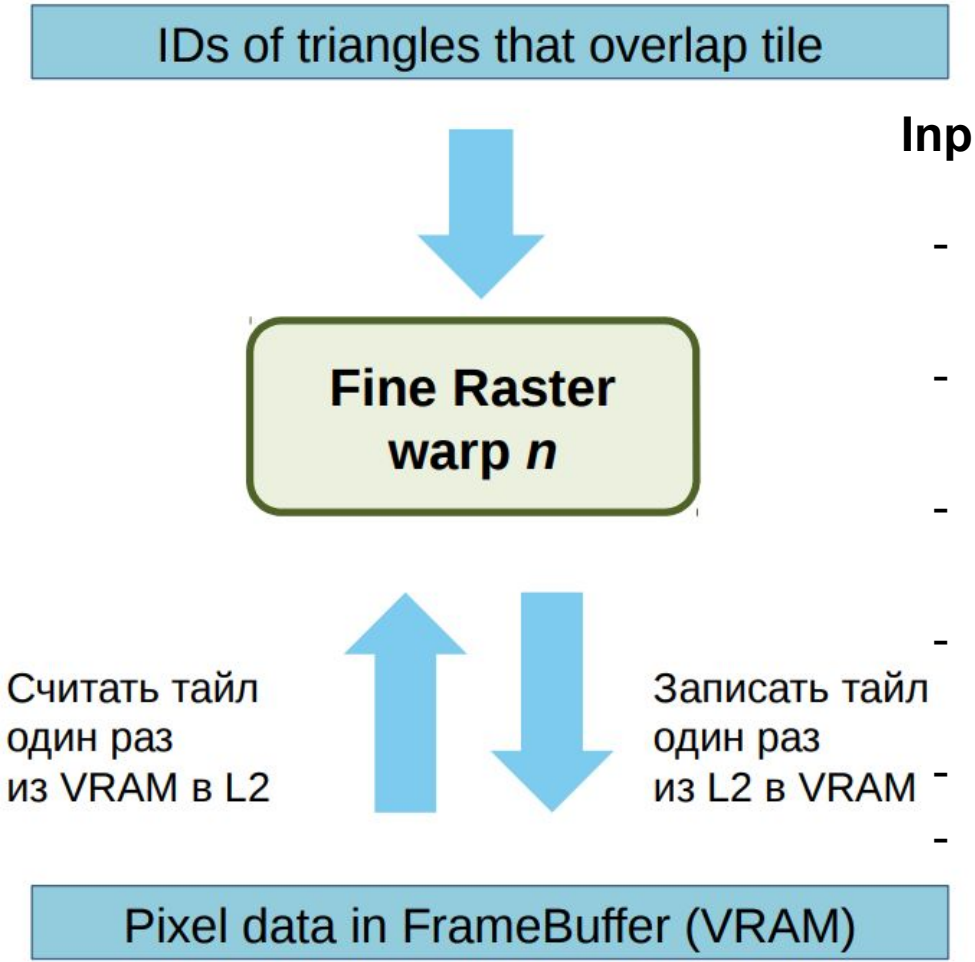
Когда треугольников подгружено достаточно для утилизации всех потоков - каждый треугольник обрабатывается своим потоком:

- Определяются **тайлы** пересекающие треугольник
- Оптимизированный код для очень маленьких и очень больших треугольников

Результат пишется в специально выделенную для данной корзины очередь (соответственно не нужна синхронизация между разными SM).

Сильно разное количество покрытых тайлов от треугольника к треугольнику (т.е. от потока к потоку). Нужно равномерно распределить запись результата.

cudaraster **Triangle Setup** → **Bin Raster** → **Coarse Raster** → **Fine Raster**



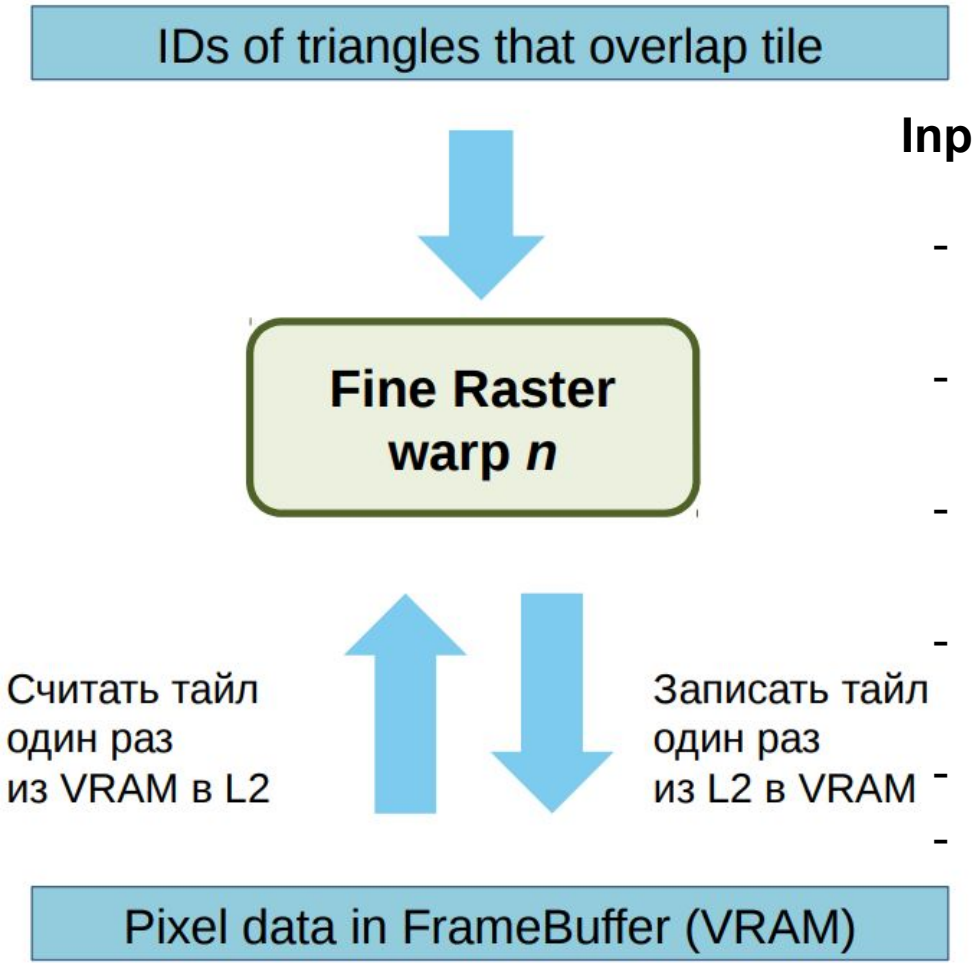
Работаем с тайлом в **local memory**

Input phase:

- Подгружаем в **L2** 32 треугольника пересекающих обрабатываемый тайл
- Откидываем треугольники учтя их **zmin** с **zmax** относительно тайла
- Рассчитываем pixel coverage используя **L**ook **U**p **T**ables: 8x8, 153 инструкции
- Повторяем пока не наберется хотя бы 32 фрагмента
- Когда набралось - **Shading phase**:
- Один фрагмент - один поток

cudaraster

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**



Работаем с тайлом в **local memory**

Input phase:

- Подгружаем в **L2** 32 треугольника пересекающих обрабатываемый тайл
- Откидываем треугольники учтя их **zmin** с **zmax** относительно тайла
- Рассчитываем pixel coverage используя **L**ook **U**p **T**ables: 8x8, 153 инструкции
- Повторяем пока не наберется хотя бы 32 фрагмента
- Когда набралось - **Shading phase**:
- Один фрагмент - один поток

cudaraster

Pixel Coverage LUTs

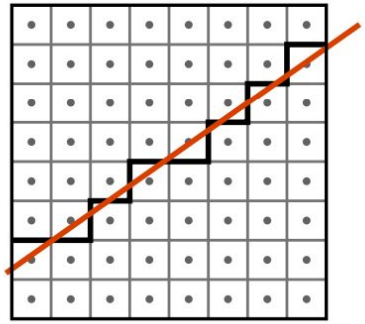


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a)*

cudaraster

Pixel Coverage LUTs

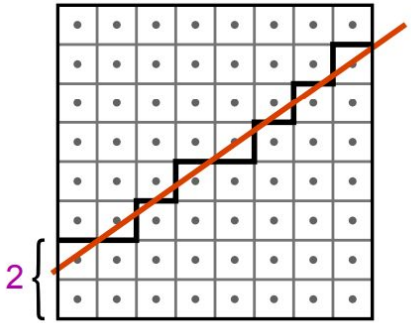


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column,*

cudaraster

Pixel Coverage LUTs

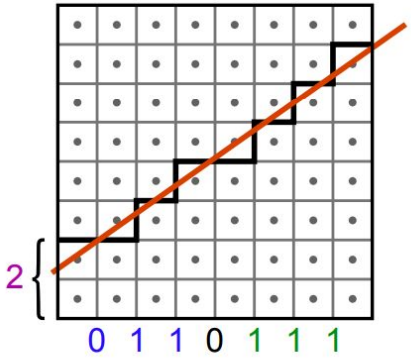


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is*

cudaraster

Pixel Coverage LUTs

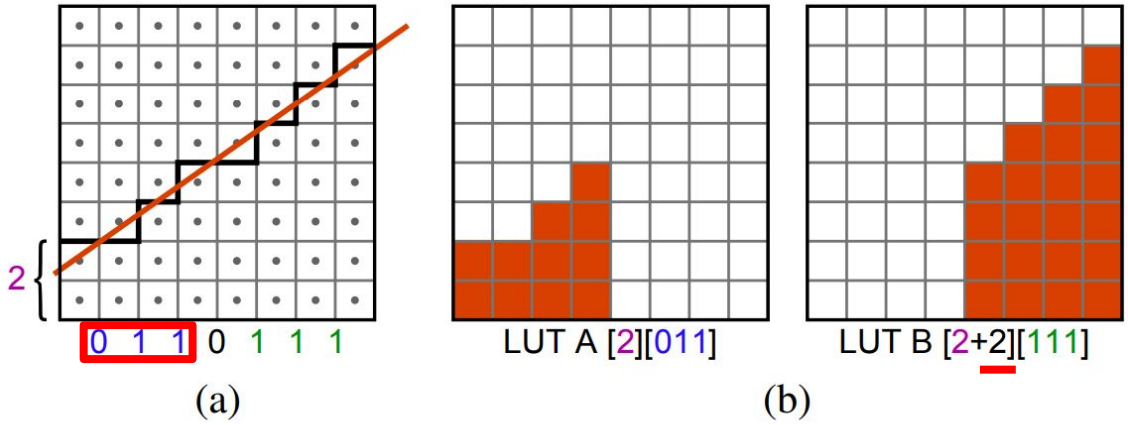


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8*

cudaraster

Pixel Coverage LUTs

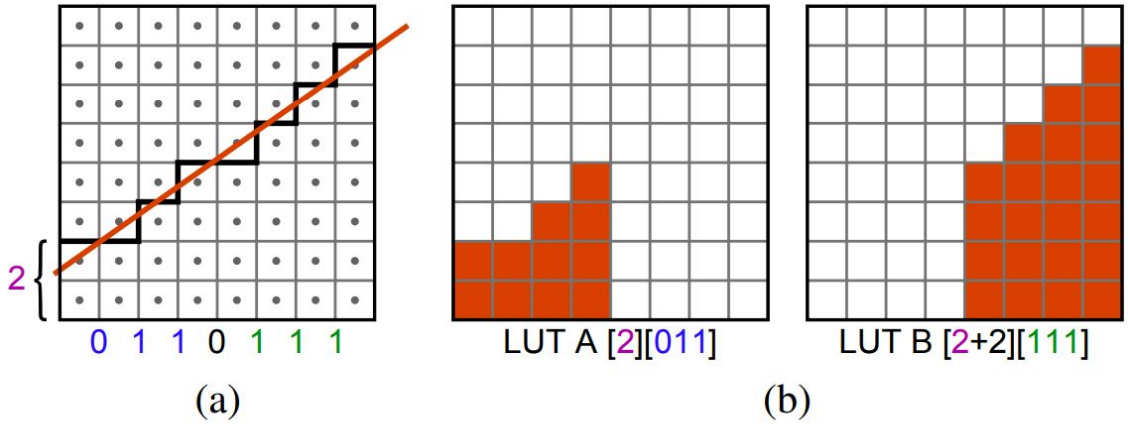
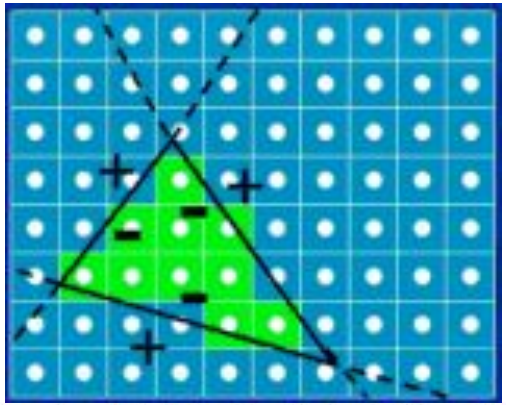


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.*

cudaraster

Pixel Coverage LUTs



```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```

Как это соотносится с Brute Force?

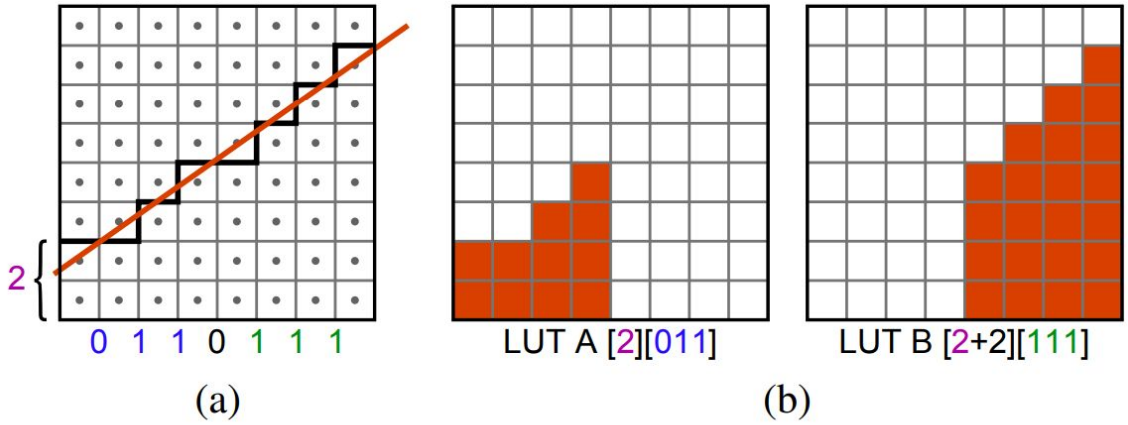
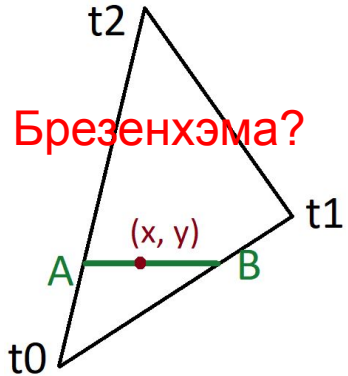


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.*

```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
        for (int x = A.x; x <= B.x; ++x) {
            put_pixel(x, y);
        }
    }
    for (int y = t1.y; y <= t2.y; ++y) {
        ...
    }
}
```

А относительно Брезенхэма?



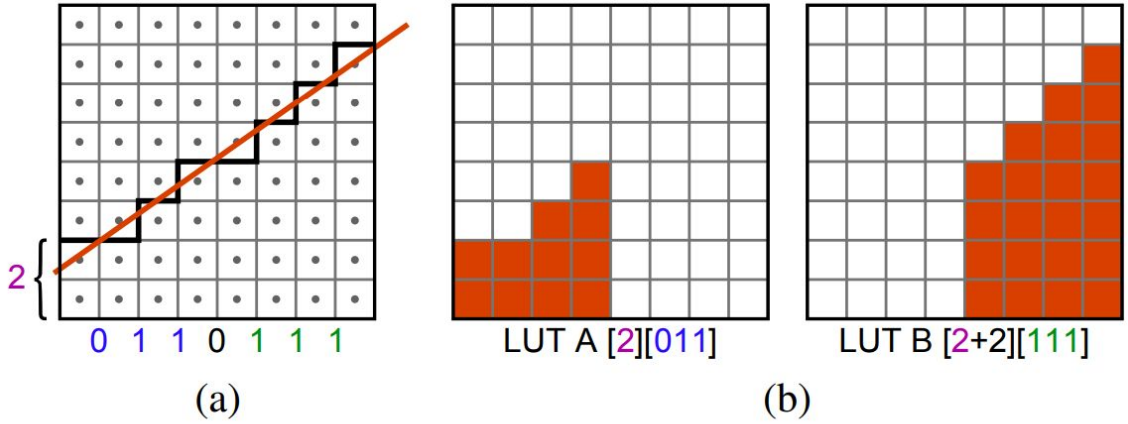


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.*

```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
        for (int x = A.x; x <= B.x; ++x) {
            put_pixel(x, y);
        }
    }
    for (int y = t1.y; y <= t2.y; ++y) {
        ...
    }
}
```

А относительно Брезенхэма?

Code Divergence?

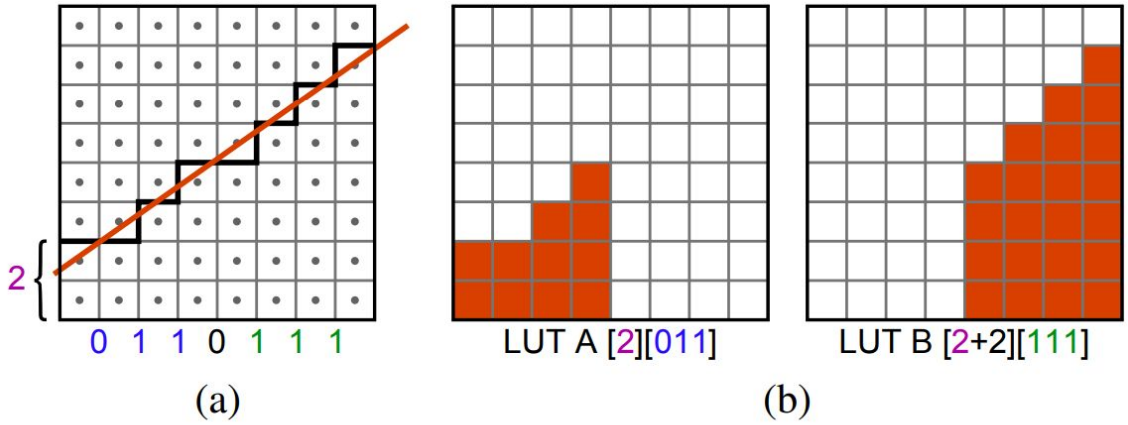
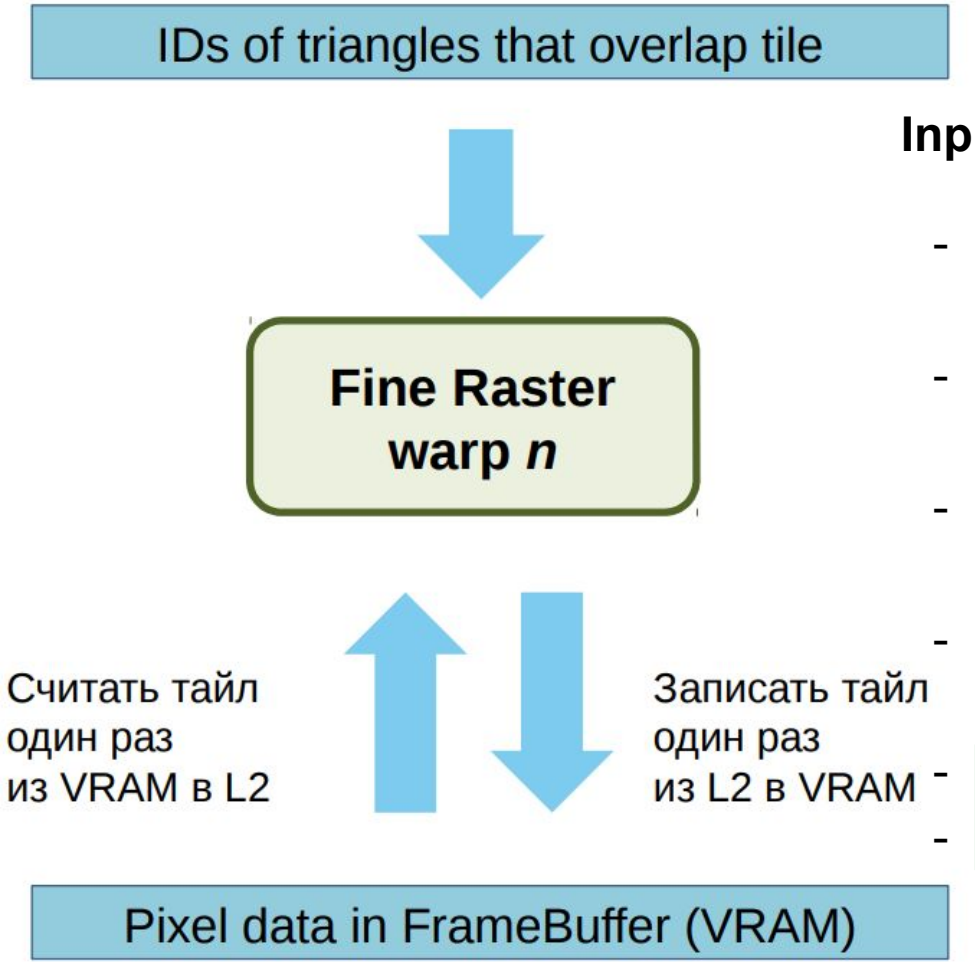


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.*

cudaraster **Triangle Setup** → **Bin Raster** → **Coarse Raster** → **Fine Raster**

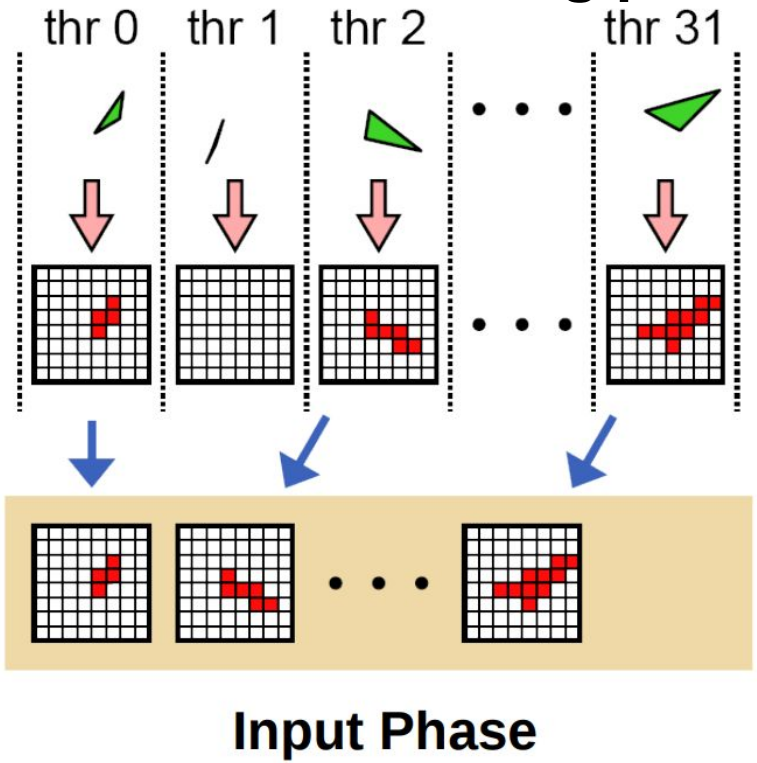


Работаем с тайлом в **local memory**

Input phase:

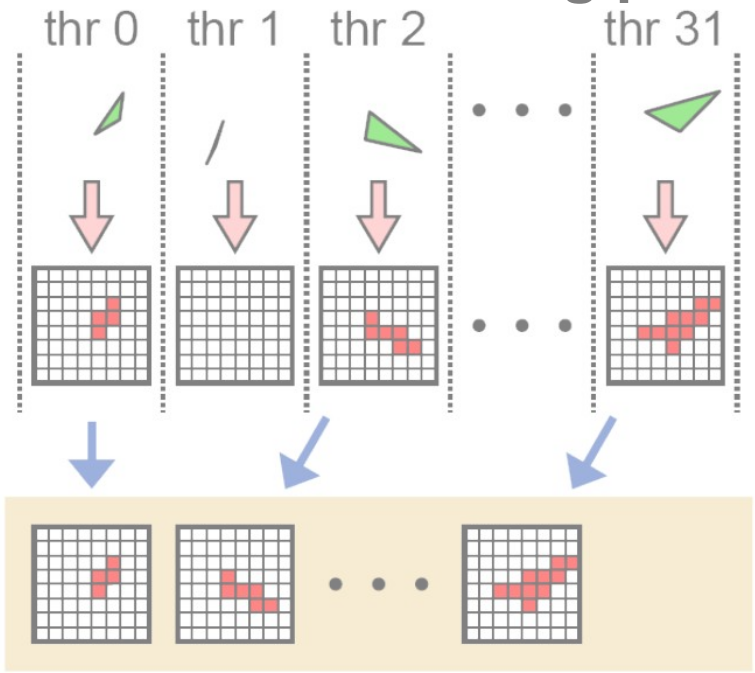
- Подгружаем в **L2** 32 треугольника пересекающих обрабатываемый тайл
- Откидываем треугольники учтя их **zmin** с **zmax** относительно тайла
- Рассчитываем pixel coverage используя **L**ook **U**p **T**ables: 8x8, 153 инструкции
- Повторяем пока не наберется хотя бы 32 фрагмента
- Когда набралось - **Shading phase**:
- Один фрагмент - один поток

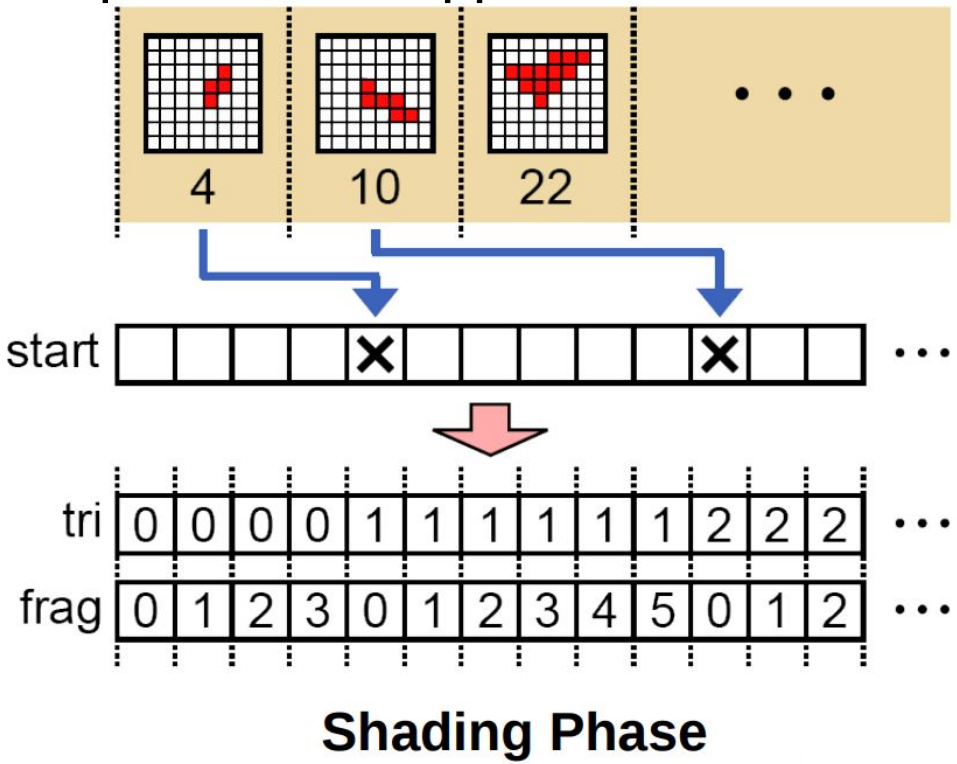
cudaraster **Shading phase:** распределение фрагментов по **SM**



- **В первой фазе** рассчитываем покрытые и пишем в список

cudaraster **Shading phase:** распределение фрагментов по **SM**

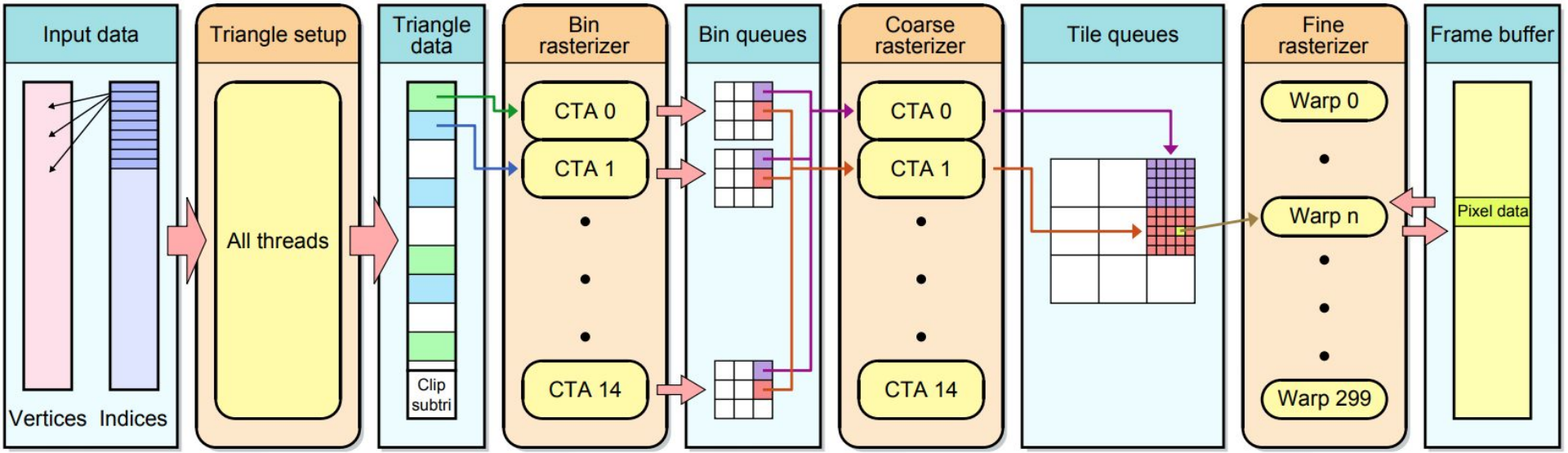




- **В первой фазе** рассчитываем покрытые и пишем в список
- **Во второй фазе** рассчитываем префиксными суммами индекс фрагмента треугольник среди все фрагментов всех обрабатываемых в данный момент треугольников

cudaraster

Triangle Setup → Bin Raster → Coarse Raster → Fine Raster



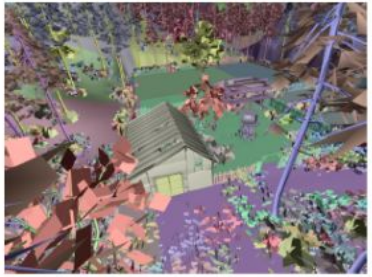
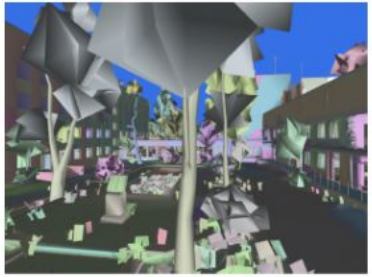
cudaraster

Подробнее: cudaraster







SAN MIGUEL, 189MB
5.44M tris, 25% visible
2.4 pixels / triangle

JUAREZ, 24MB
546K tris, 37% visible
14.6 pixels / triangle

STALKER, 11MB
349K tris, 41% visible
14.1 pixels / triangle

CITY, 51MB
879K tris, 21% visible
16.3 pixels / triangle

BUDDHA, 29MB
1.09M tris, 32% visible
1.4 pixels / triangle

Call of Juarez scene courtesy of **Techland**
S.T.A.L.K.E.R.: Call of Pripyat scene courtesy of **GSC Game World**

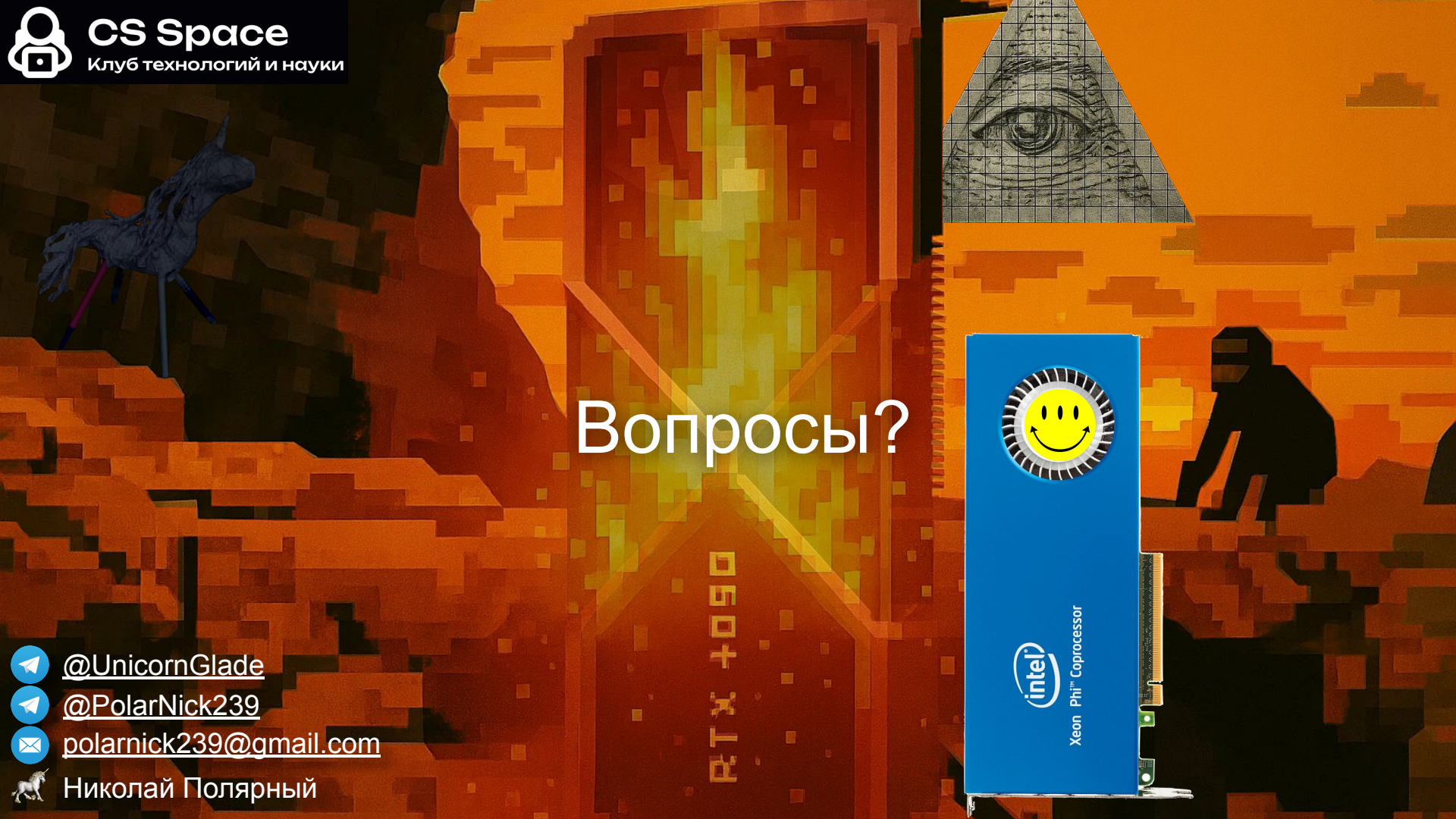
cudaraster

Подробнее: cudaraster

Scene	Resolution	HW	Our (SW)	FreePipe (FP)	SW:HW ratio	FP:SW ratio
San Miguel	512×384	5.37	7.82	130.14	1.46	16.65
	1024×768	5.43	9.48	510.20	1.74	53.84
	2048×1536	5.86	15.44	1652.52	2.64	107.06
Juarez	512×384	0.59	2.71	5.34	4.56	1.97
	1024×768	0.67	3.28	18.63	4.87	5.69
	2048×1536	1.03	7.06	72.45	6.84	10.26
Stalker	512×384	0.31	1.81	23.47	5.91	12.96
	1024×768	0.39	2.31	92.73	5.96	40.14
	2048×1536	0.67	5.41	386.07	8.10	71.36
City	512×384	0.93	2.16	64.56	2.32	29.88
	1024×768	1.04	3.13	251.86	3.01	80.54
	2048×1536	1.42	6.79	1032.83	4.77	152.13
Buddha	512×384	1.06	2.09	2.14	1.98	1.02
	1024×768	1.07	2.66	3.08	2.50	1.16
	2048×1536	1.11	4.01	6.96	3.62	1.73

CS Space
Клуб технологий и науки

Вопросы?



@UnicornGlade
@PolarNick239
polarnick239@gmail.com
Николай Полярный