Computer Graphics II
– Geometry Shader

Kai Lawonn

Introduction

- Between vertex & fragment shader, optional shader: geometry shader
- A geometry shader takes as input a set of vertices that form a single primitive (point, triangle, …)
- Can transform these vertices before sending them to the next shader
- Geometry shader interesting because it is able to transform the vertices to completely different primitives (generate more vertices)

Introduction

- Start right off with an example of a geometry shader:

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

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

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

	EndPrimitive();
}
```

Introduction

- At the start, need to declare the type of primitive input
- Do this by declaring a *layout* specifier in front of the *in* keyword

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

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

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

          EndPrimitive();
}
```

Introduction

- Input layout qualifier can take the following primitive values from a vertex shader:

points:	GL_POINTS (1)
lines:	GL_LINES, GL_LINE_STRIP (2)
lines_adjacency:	GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY (4)
triangles:	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN (3)
triangles_adjacency :	GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY (6)

- Number in parenthesis = minimal number of vertices a single primitive contains

Introduction

- Almost all rendering primitives to give to rendering calls like glDrawArrays
- If we like to draw vertices as GL_TRIANGLES → set input to triangles

Introduction

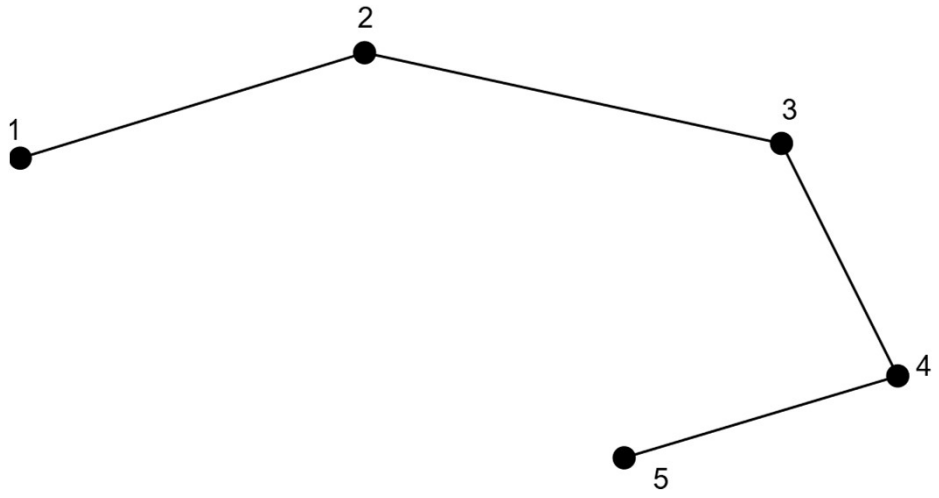
```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;
void main() {
            gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);   EmitVertex();
            gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);   EmitVertex();
            EndPrimitive();
}
```

- Need to specify a primitive type that the geometry shader will actually output → layout specifier in front of the out keyword
- Output layout qualifier can also take several primitive values:
  - points
  - line_strip
  - triangle_strip
- These output specifiers can create almost any shape from the input primitives
- To generate a single triangle: specify triangle_strip as the output and then output 3 vertices

Introduction

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;
void main() {
            gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);   EmitVertex();
            gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);   EmitVertex();
            EndPrimitive();
}
```

- A line strip binds together a set of points to form one continuous line between them with a minimum of 2 points
- Each extra point results in a new line between the new point and the previous point:



Introduction

- Currently only output a single line (maximum number of vertices is 2)
- To generate meaningful results we need some way to retrieve the output from the previous shader stage
- GLSL gives us a built-in variable called gl_in that internally (probably) looks something like this

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

Introduction

- Declared as an interface block with variables of which the most interesting one is gl_Position (contains the similar vector set as the vertex shader's output)
- Note that it is declared as an array, because most render primitives consist of more than 1 vertex and the geometry shader receives all vertices of a primitive as its input

Introduction

- Using the vertex data from the previous vertex shader, can generate new data via EmitVertex and EndPrimitive
- Geometry shader expects to generate/output at least one of the primitives specified as output
- In our case one line strip primitive:

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;
void main() {
        gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
        EmitVertex();

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

        EndPrimitive();}
```

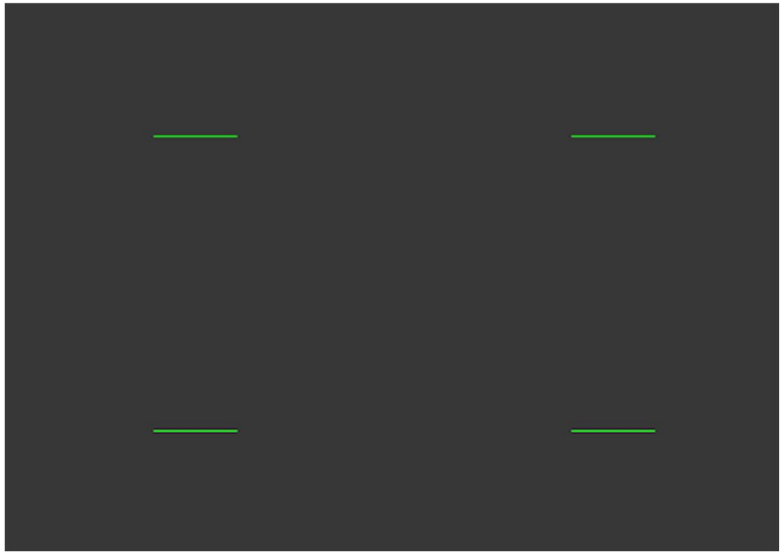
Introduction

- Set gl_Position and call EmitVertex: vector is added to the primitive
- EndPrimitive combines vertices into the specified output primitive
- Repeatedly calling EndPrimitive after one or more EmitVertex calls multiple primitives can be generated
- Here: two translated vertices emitted and combined into a single line

```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;
void main() {
        gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
        EmitVertex();

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

        EndPrimitive();}
```

F5…

- … geometry shader takes a point primitive as its input and creates a horizontal line



Introduction

- This output was generated using just the following render call:

```
glDrawArrays(GL_POINTS, 0, 4);
```

- Shows how we can use geometry shaders to (dynamically) generate new shapes on the fly

Using Geometry Shaders

Using Geometry Shaders

- Render a really simple scene with 4 points on the z-plane in NDCs
- The coordinates of the points are:

```
float points[] = {
        -0.5f,  0.5f, // top-left
         0.5f,  0.5f, // top-right
         0.5f, -0.5f, // bottom-right
        -0.5f, -0.5f, // bottom-left
    };
```

Using Geometry Shaders

- Vertex shader draws the points on the z-plane so (need a basic vertex shader only):

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

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

Using Geometry Shaders

- Output the color green for all points in the fragment shader:

```
#version 330 core
out vec4 FragColor;

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

Using Geometry Shaders

- Generate a VAO and a VBO for the points' vertex data and then draw them via glDrawArrays:

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

F5…

- … dark scene with 4 (difficult to see) green points:

Using Geometry Shaders

- Now we add a geometry shader to the scene
- First, we create a pass-through geometry shader that takes a point primitive as its input and passes it to the next shader unmodified:

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

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

Using Geometry Shaders

- Geometry shader needs to be compiled and linked to a program (like vertex and fragment shader)
- Create the shader using GL_GEOMETRY_SHADER as the shader type

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

F5…

- … same!

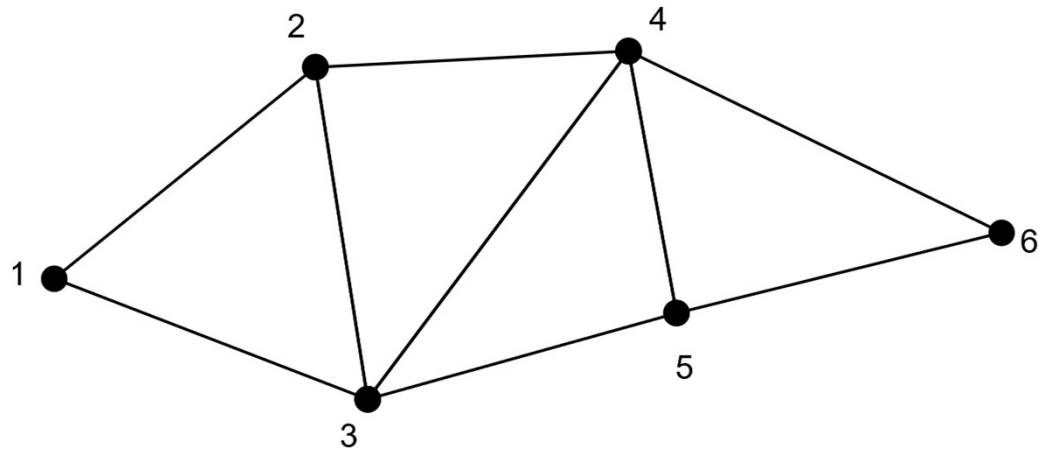
Example: Building Houses

Introduction

- Now, use the geometry shader to draw a house at the location of each point
- For this, set the output of the geometry shader to triangle_strip and draw a total of three triangles (two for a square, one for the roof)

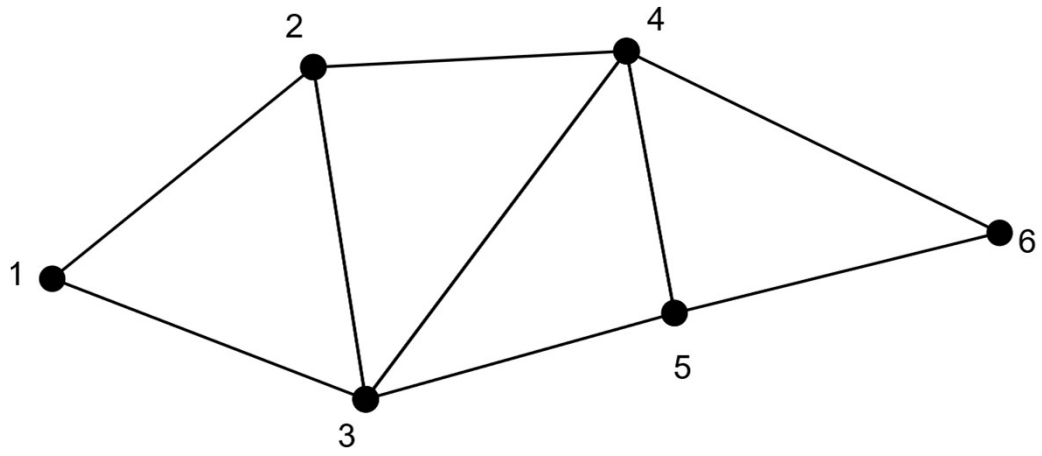
Introduction

- A triangle strip an efficient way to draw triangles using less vertices
- After the first triangle is drawn, each subsequent vertex will generate another triangle: every 3 adjacent vertices will form a triangle
- 6 vertices that form a triangle strip get the following triangles: (1,2,3), (2,3,4), (3,4,5) and (4,5,6)



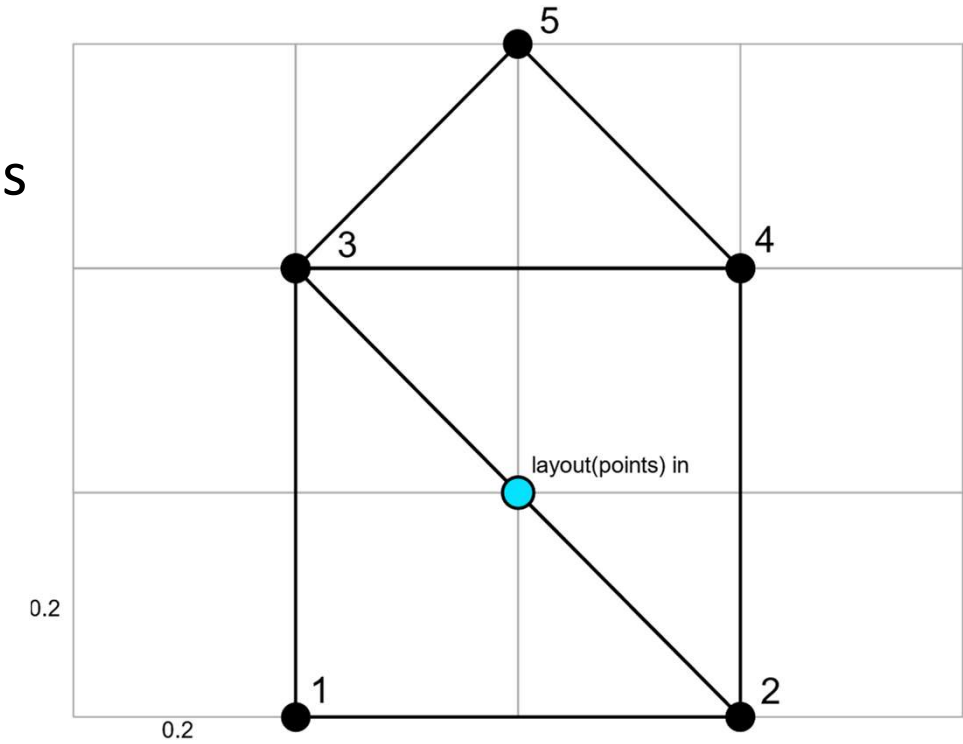
Introduction

- Triangle strip needs at least 3 vertices and will generate N-2 triangles
- With 6 vertices we created 6-2 = 4 triangles

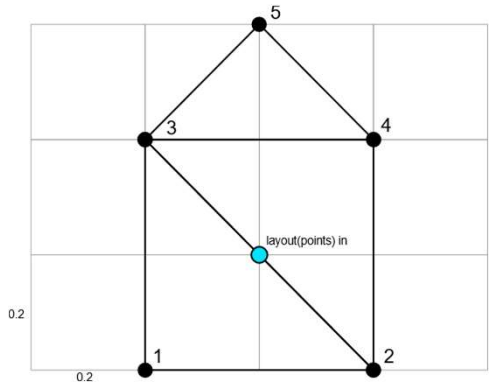


Building Houses

- Create the house shape with a triangle_strip
- Order of the vertices and the coordinates (blue input point):



Building Houses

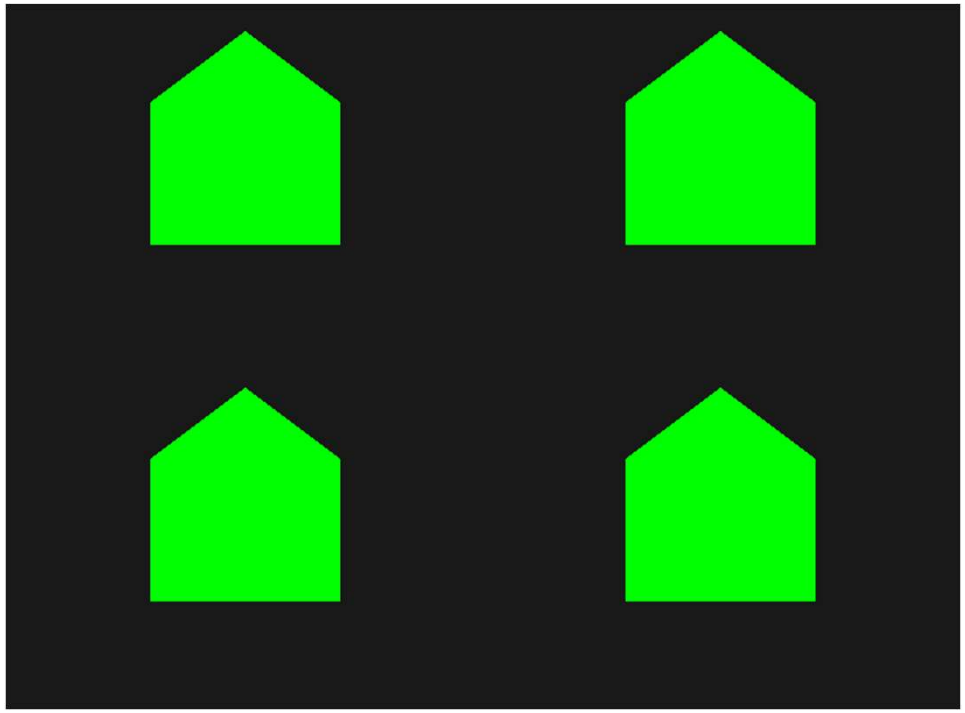


- This translates to the following geometry shader:

```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;
void build_house(vec4 position)
{
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0); // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0); // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0); // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0); // 5:top
    EmitVertex();
    EndPrimitive();
}

void main() {
    build_house(gl_in[0].gl_Position);}
```

F5…

- … 4 houses drawn using just a single point in space
- Now, give each house a unique color → add an extra vertex attribute with color information per vertex in the vertex shader



Building Houses

- The updated vertex data:

```
float points[] = {
        -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, // top-left
         0.5f,  0.5f, 0.0f, 1.0f, 0.0f, // top-right
         0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // bottom-right
        -0.5f, -0.5f, 1.0f, 1.0f, 0.0f  // bottom-left
    };
```

Building Houses

- Update the vertex shader to forward the color attribute to the geometry shader using an interface block:

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

out VS_OUT {
    vec3 color;
} vs_out;

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

Building Houses

- Also declare the same interface block (with a different interface name) in the geometry shader:

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

- Geometry shader acts on a set of vertices as input
- Input from vertex shader is always represented as arrays of data even though we only have a single vertex

Building Houses

Don’t have to use interface blocks, could have also written it as:

```
in vec3 vColor[];
```

If the vertex shader forwarded the color vector as out vec3 vColor

Interface blocks easier to work with in shaders (like geometry shader)

In practice, geometry shader inputs can get quite large and grouping them in one large interface block array makes a lot more sense

Building Houses

- Should also declare an output color vector for the next fragment shader stage:

```
out vec3 fColor;
```

- Fragment shader expects only a single (interpolated) color
- The fColor vector is thus not an array, but a single vector
- When emitting a vertex, each vertex will store the last stored value in fColor for its fragment shader run

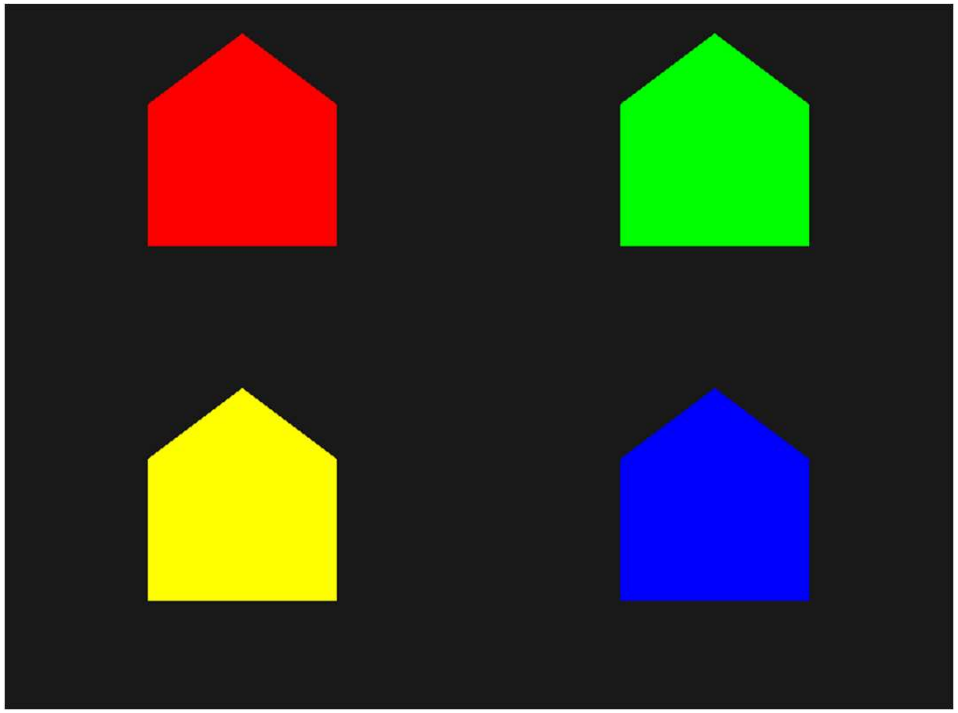
Building Houses

- For the houses, can only fill fColor once with the color from the vertex shader before the first vertex is emitted to color the entire house:

```
void build_house(vec4 position)
{
    fColor = gs_in[0].color; // gs_in[0] since there's only one input vertex
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0); // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0); // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0); // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0); // 5:top
    EmitVertex();
    EndPrimitive();
}
```

F5…

- … all the houses will now have a color of their own:



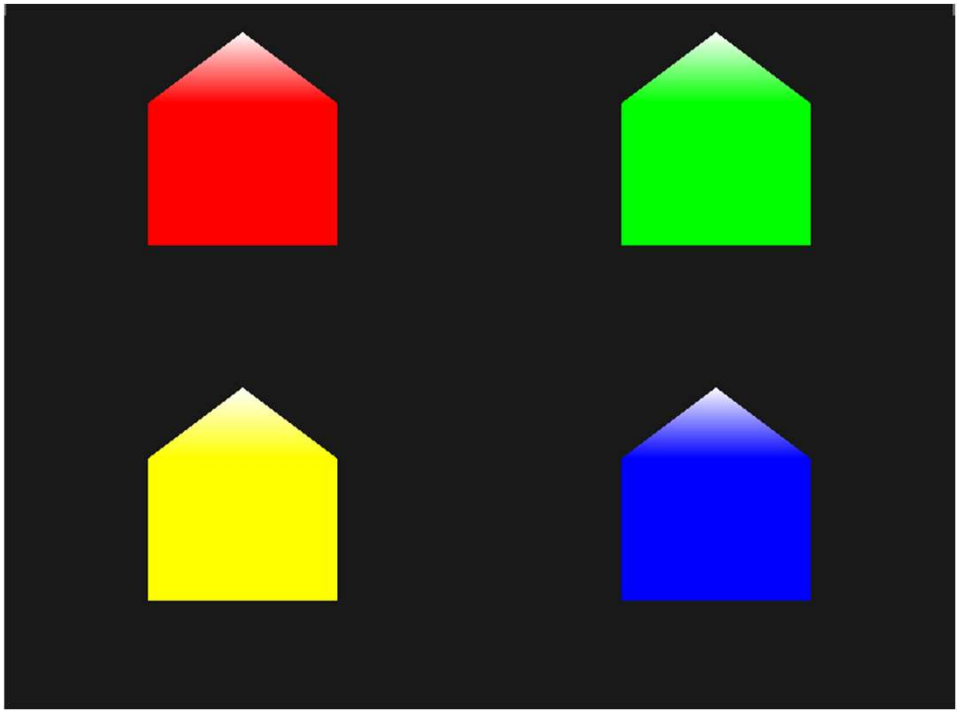
Building Houses

- Now, we add a color for the last vertex:

```
void build_house(vec4 position)
{
    fColor = gs_in[0].color; // gs_in[0] since there's only one input vertex
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0); // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0); // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0); // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0); // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0); // 5:top
    fColor = vec3(1.0, 1.0, 1.0);
    EmitVertex();
    EndPrimitive();
}
```

F5…

- … white roof



Remark

- Pretty creative with geometry shaders
- Because the shapes are generated dynamically on the GPU this is a lot more efficient than defining these shapes yourself within vertex buffers
- Geometry buffers are therefore a great tool for optimization on simple often-repeating shapes like cubes in a voxel world or grass leaves on a large outdoor field

Example: Exploding Objects

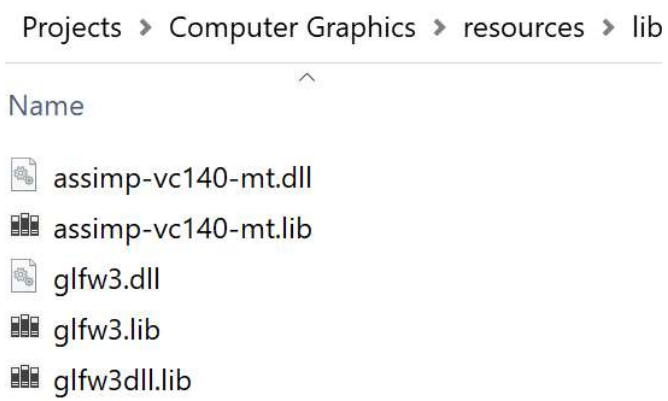
Introduction

- Exploding objects is something we will not going to use that much, but it shows some of the powers of geometry shaders
- Precisely, we will move each triangle along their normal
- The effect is that the entire object's triangles seem to explode

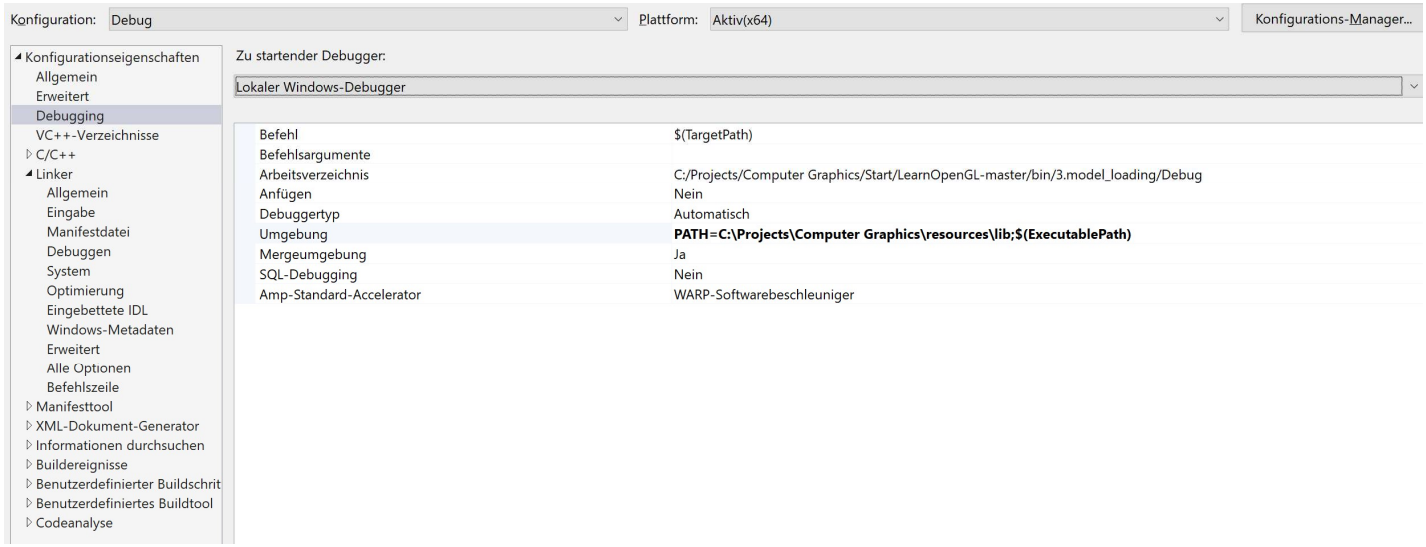
Introduction

- Effect of exploding triangles on the nanosuit model looks a bit like this:

Remark



- Starting the example from the book
- VS go to Project>Properties>Configuration Properties>Debugging in the "Environment" property type: PATH=C:\Projects\Computer Graphics\resources\lib;$(ExecutablePath)



Introduction

- Want to translate vertices along the normal vector: need to calculate this vector
- Calculate a vector that is perpendicular to the surface of a triangle, using just the 3 vertices we have access to → cross product
- With two vectors parallel to the surface of the triangle, retrieve the normal by doing a cross product on those vectors

Exploding Objects

- The following geometry shader function does exactly this to retrieve the normal vector using 3 input vertex coordinates:

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

Exploding Objects

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

- Retrieve two vectors *a* and *b* parallel to the triangle using subtraction
- Subtracting two vectors results in a vector that lie on the triangle plane,
- Note: $cross(a, b) = -cross(b, a)$ (order is important)

Exploding Objects

- Now create an explode function: normal and vertex position
- Returns vector that translates the position vector along the normal:

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

Exploding Objects

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

- *sin(time)* returns a value between -1.0 and 1.0
- Do not want to implode the object → transform *sin* values to the [0,1]
- The resulting value is then multiplied by the normal vector and the resulting direction vector is added to the position vector

Exploding Objects

- Main function in the geometry shader:

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

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

Exploding Objects

- Do not forget to set the time variable in your OpenGL code:

```
shader.setFloat("time", glfwGetTime());
```

- Result is a 3D model that continually explode its vertices over time
- Not very useful, but it shows advanced use of the geometry shader

Example: Visualizing Normal Vectors

Introduction

- Programming lighting shaders we may run into weird visual outputs
- A common cause is due to incorrect normal vectors caused by incorrectly loading vertex data
- What we want is some way to detect if the normal vectors we supplied are correct
- Great way: visualizing them and the geometry shader is an extremely useful tool for this purpose

Visualizing Normal Vectors

- Idea: first draw the scene (without geometry shader), then draw the scene a second time (only displaying normal vectors generated via a geometry shader)
- Geometry shader input a triangle and generates 3 lines in the direction of the normal vector - one normal vector for each vertex:

```
shader.use();
DrawScene();
normalDisplayShader.use();
DrawScene();
```

Visualizing Normal Vectors

- Now, create a geometry shader that uses the vertex normals supplied by the model instead of generating them
- To accommodate for scaling and rotations (view and model matrix), transform the normals first with a normal matrix (before clip-space)
- Geometry shader receives its position vectors as clip-space coordinates → also transform the normal vectors to the same space

Visualizing Normal Vectors

- This can all be done in the vertex shader:

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

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 view;
uniform mat4 model;

void main()
{
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = vec3(vec4(normalMatrix * aNormal, 0.0));
    gl_Position = view * model * vec4(aPos, 1.0);
}
```

Visualizing Normal Vectors

- Geometry shader takes each vertex (with position and normal) and draws a normal vector from each position vector:

```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
} gs_in[];

const float MAG = 0.2;

uniform mat4 projection;

void GenerateLine(int index)
{
    gl_Position = projection * gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = projection * (gl_in[index].gl_Position + vec4(gs_in[index].normal, 0.0) * MAG);
    EmitVertex();
    EndPrimitive();
}…
```

Visualizing Normal Vectors

- In the main, generate a normal for every vertex:

```
void main()
{
    GenerateLine(0); // first vertex normal
    GenerateLine(1); // second vertex normal
    GenerateLine(2); // third vertex normal
}
```

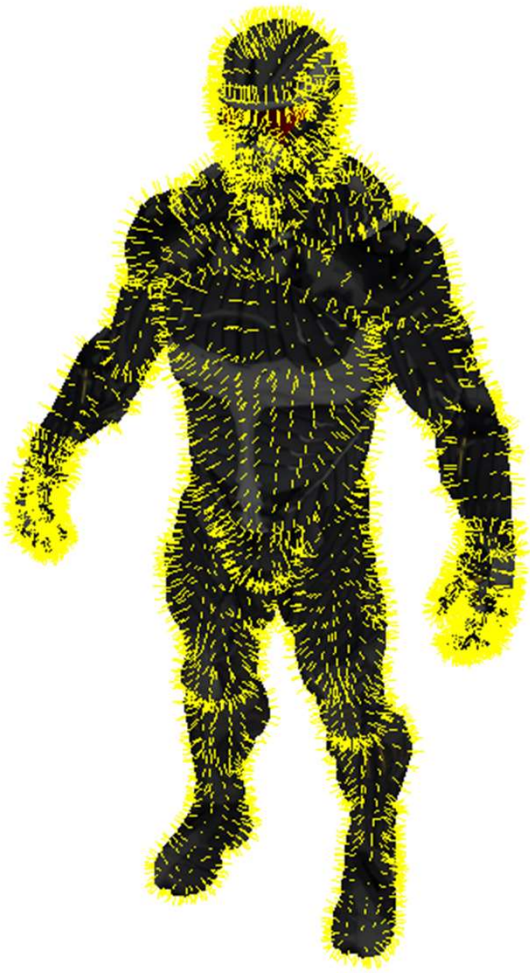
Visualizing Normal Vectors

- Single color with the help of the fragment shader:

```
#version 330 core
out vec4 FragColor;

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

F5…

- … yellow normals



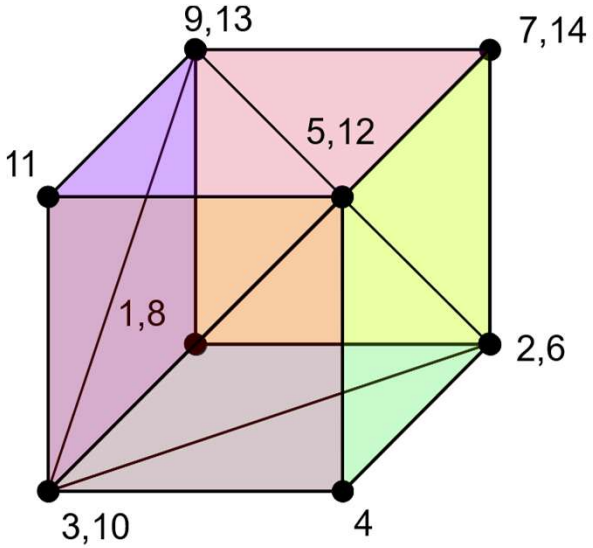
Example: Building Cubes*

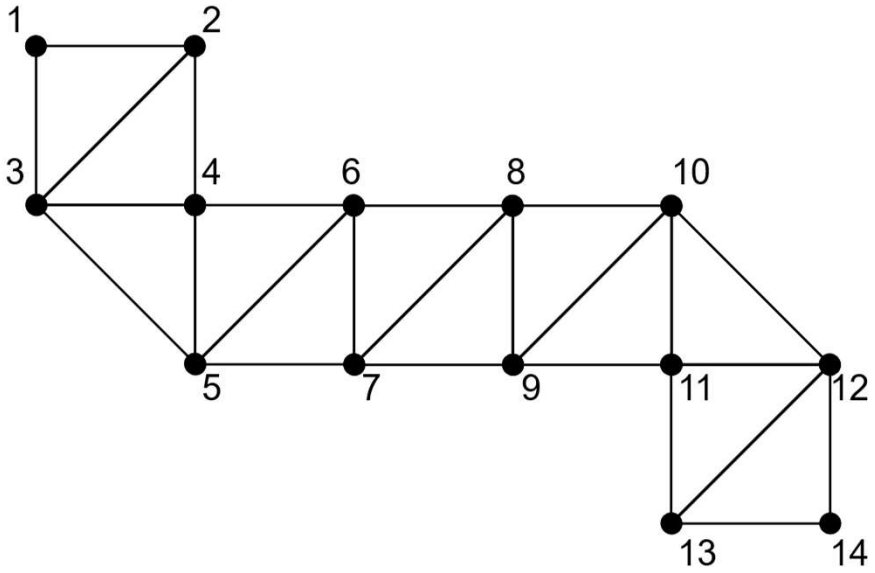
Introduction

- Instead of drawing simple lines, we will generate cubes on the surface
- We will keep the shaders and the scene
- We will only change the geometry and the fragment shader

Building Cubes

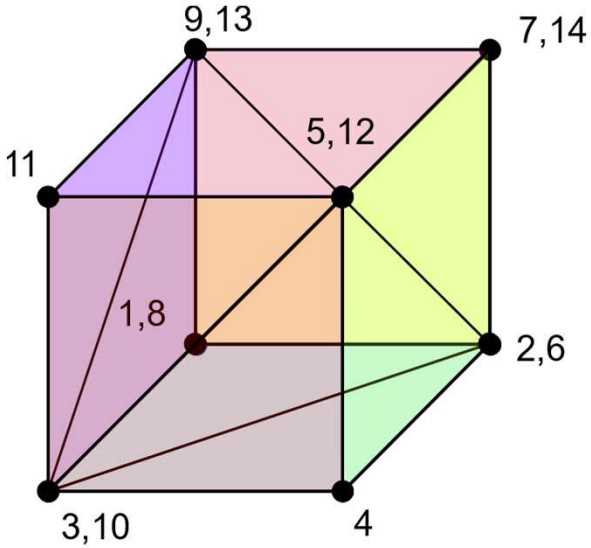
- Let us have a look how to generate a cube with a triangle strip:

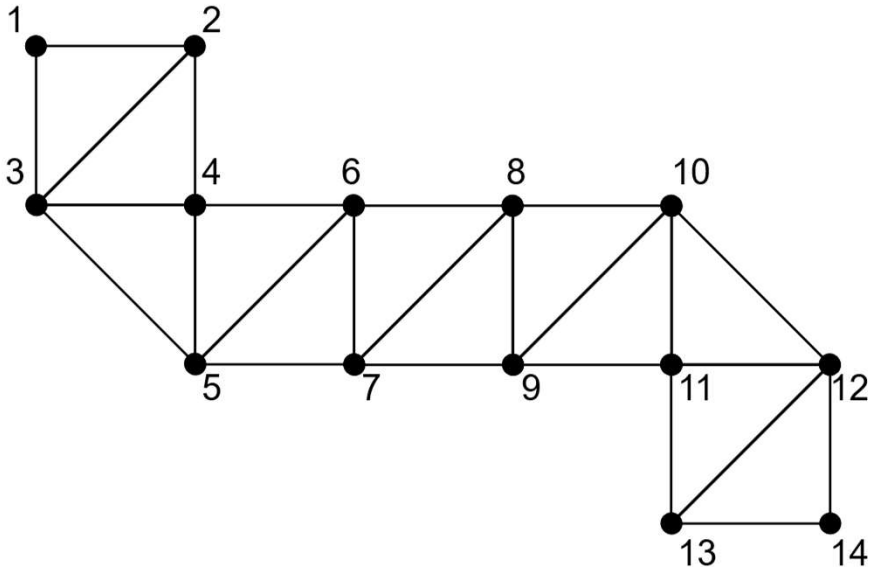




Building Cubes

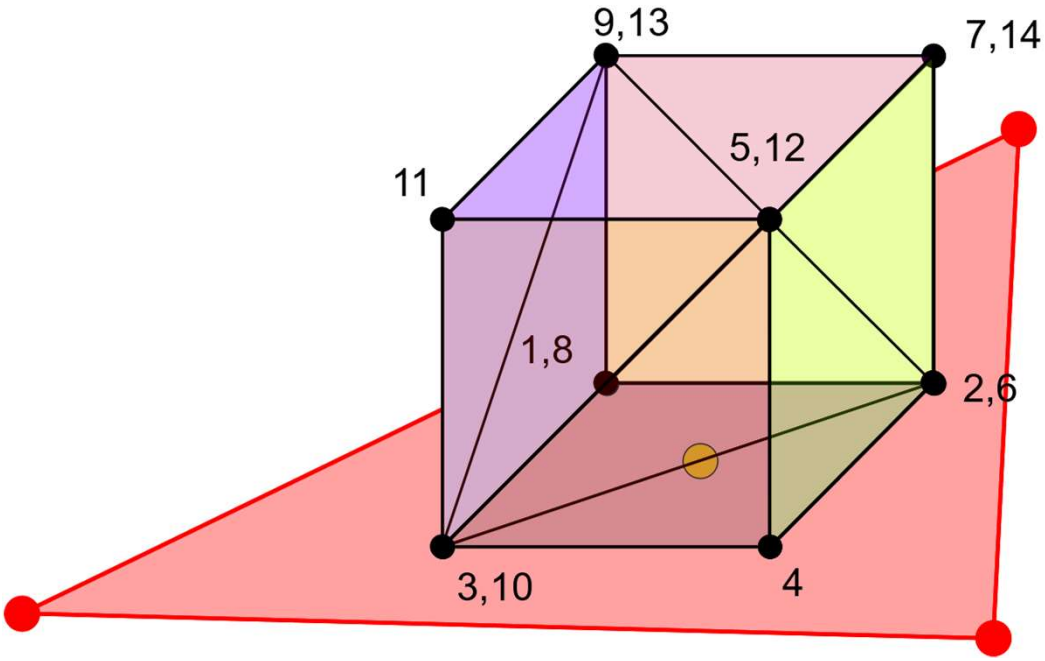
- The cube can be unfolded to the net (triangle_strip)

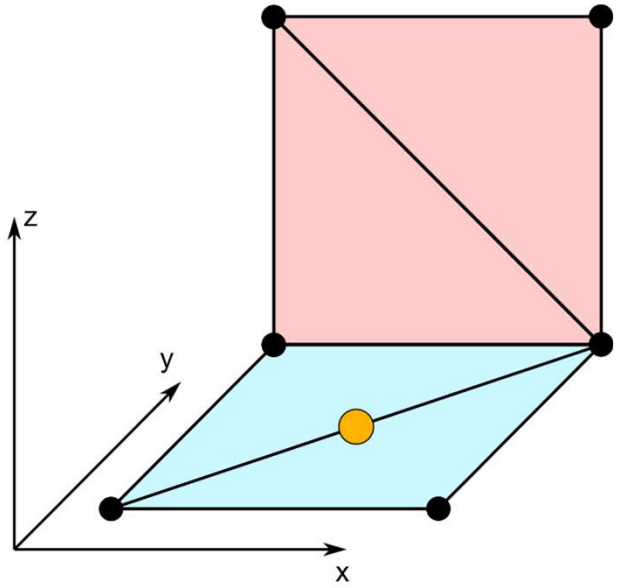




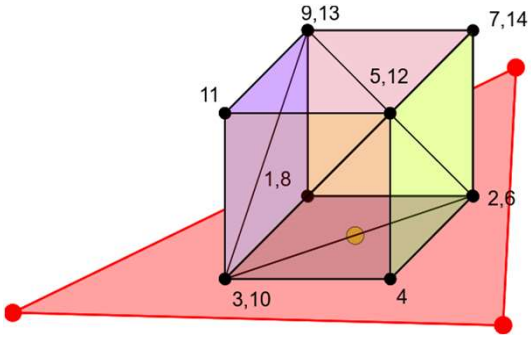
Building Cubes

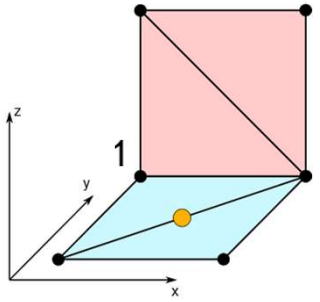
- Input triangle = red triangle (place cube at the start point)
- Orange point = start point, coordinates can be determined





Building Cubes

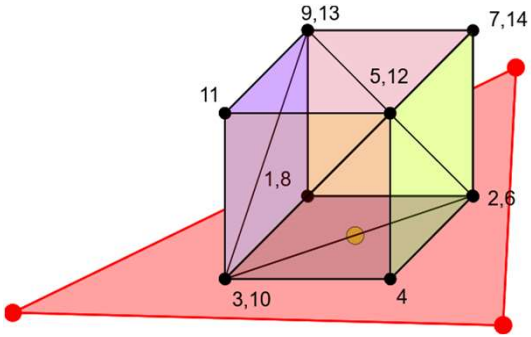


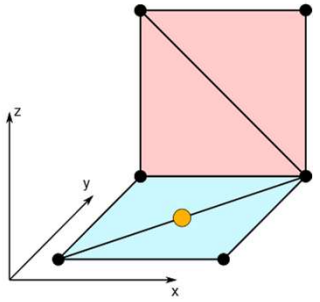


- Let's write a function that generates the cube (it uses another function CubePoint, which we will discuss later):

```
void GenerateCube(vec4 position)
{
    float size = 0.02;
    height=0; // out float
    gl_Position = projection * position + CubePoint(-size,size,0); // 1
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,size,0);  // 2
    EmitVertex();
    gl_Position = projection * position + CubePoint(-size,-size,0);// 3
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,-size,0); // 4
    EmitVertex();
…
}
```

Building Cubes

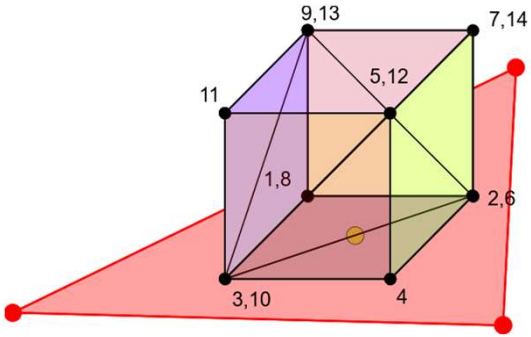


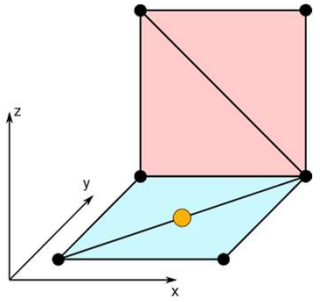


- Let's write a function that generates the cube (it uses another function CubePoint, which we will discuss later):

```
void GenerateCube(vec4 position)
{
…

    gl_Position = projection * position + CubePoint(size,-size,size); // 5
    height=1;
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,size,0); // 6
    height=0;
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,size,size);// 7
    height=1;
    EmitVertex();
    gl_Position = projection * position + CubePoint(-size,size,0);// 8
    height=0;
    EmitVertex();
…
}
```

Building Cubes

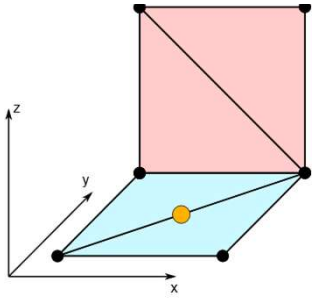




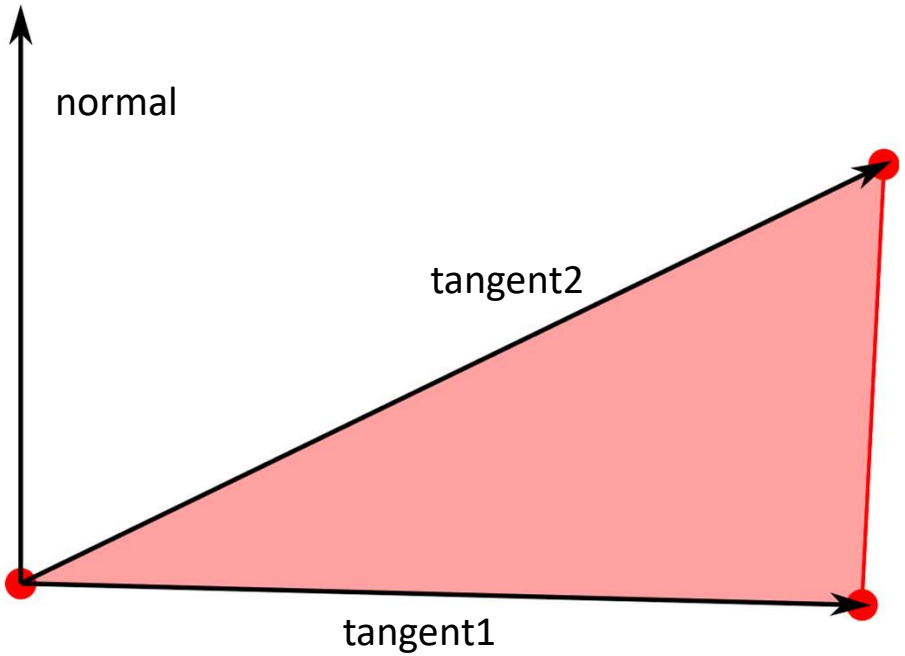
- Let's write a function that generates the cube (it uses another function CubePoint, which we will discuss later):

```
void GenerateCube(vec4 position)
{
…
    gl_Position = projection * position + CubePoint(-size,size,size); // 9
    height=1;
    EmitVertex();
    gl_Position = projection * position + CubePoint(-size,-size,0); // 10
    height=0;
    EmitVertex();
    gl_Position = projection * position + CubePoint(-size,-size,size);// 11
    height=1;
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,-size,size);// 12
    EmitVertex();
    gl_Position = projection * position + CubePoint(-size,size,size);// 13
    EmitVertex();
    gl_Position = projection * position + CubePoint(size,size,size);// 14
    EmitVertex();
    EndPrimitive();
}
```

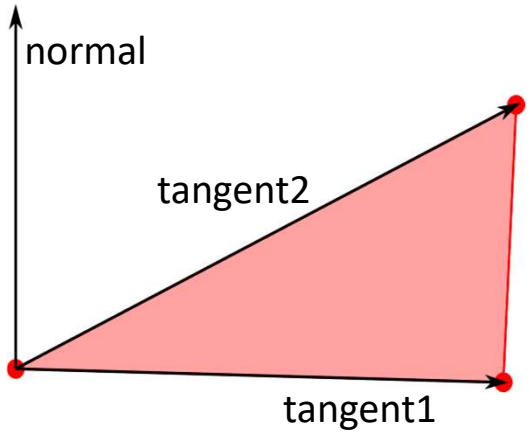
Building Cubes



- We worked with a local coordinate system (x,y,z)
- Have to work with a coordinate system on the triangle



Building Cubes



- Build the coordinate system on the triangle:

```
tangent1 = normalize(vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position));
tangent2 = normalize(vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position));
normal = -cross(tangent1, tangent2);

tangent2 = cross(normal,tangent1); // ensure to be orthogonal
```

Building Cubes

- Start point is the average of the triangle vertices:

```
void main()
{
    vec4 position = gl_in[0].gl_Position+gl_in[1].gl_Position+gl_in[2].gl_Position;
    position/ = 3.0;
    // tangent1,tangent2,normal global variables
    tangent1 = normalize(vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position));
    tangent2 = normalize(vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position));
    normal = -cross(tangent1, tangent2);
    tangent2 = cross(normal,tangent1);

    GenerateCube(position);
}
```

Building Cubes

- Finally, the CubePoint function:

```
vec4 CubePoint(float x, float y, float z)
{
    return projection*vec4(x*tangent1+y*tangent2+2*z*normal,0.0);
}
```

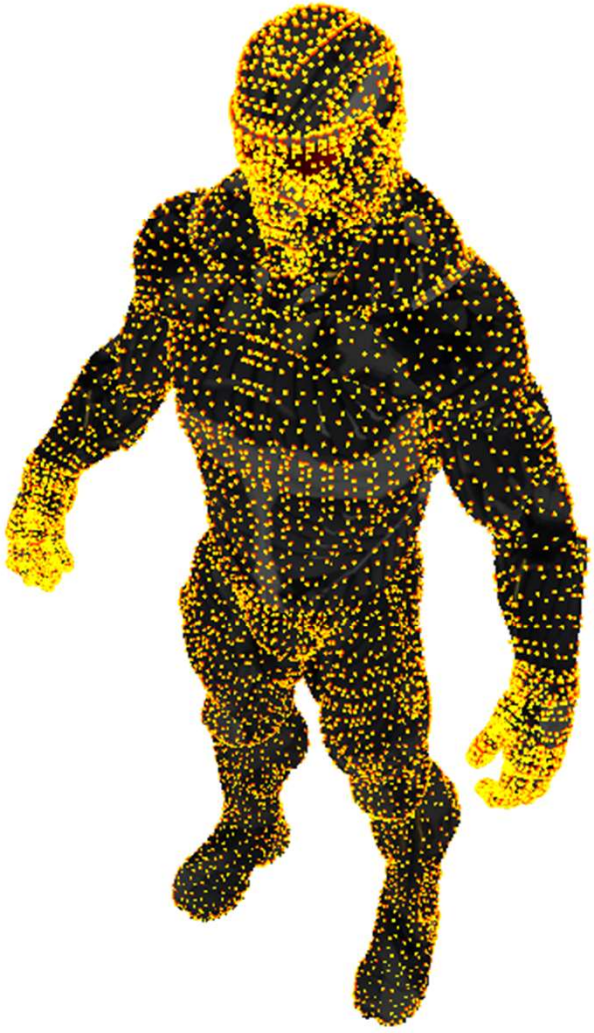
Building Cubes

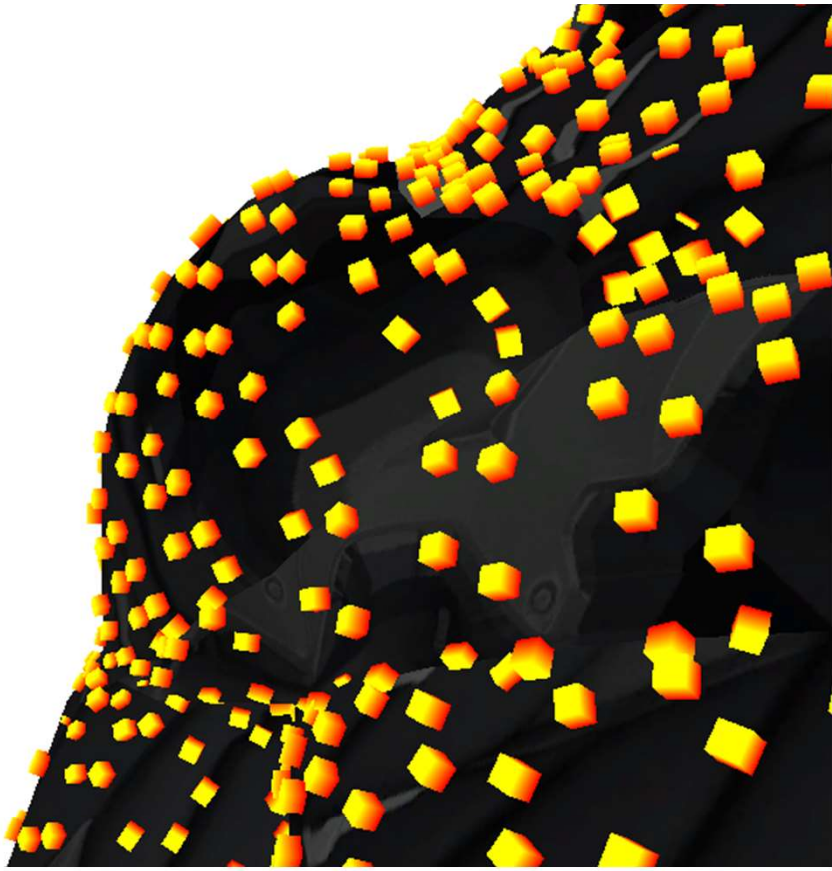
- Fragment shader:

```
#version 330 core
out vec4 FragColor;

in float height;

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

F5…

- … beautiful cubes





Questions???