h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Visual Computing

Graphic objects and their programming

Darmstadt University

Prof. Dr Elke Hergenröther
Björn Frömmer
Prof. Dr Benjamin Meyer

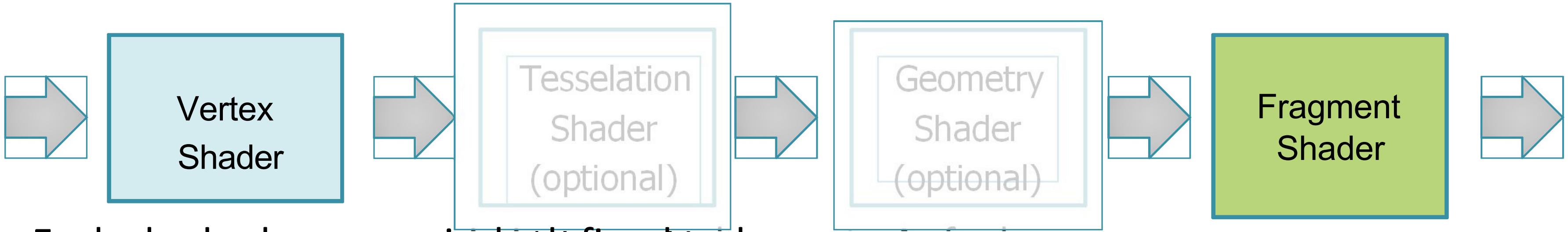
CHAPTER 6

Shader programming

Shader pipeline

- The uploaded data can still be manipulated before rendering.
- The desired calculations are implemented in so-called **shaders**, small programme fragments that can be executed directly on the GPU.

- Computer language of shader programmes: GLSL (**OpenGL** Shading Language)
  - Alternatives: HLSL for pure Windows/DirectX development, Cg (from NVidia, discontinued in 2012).

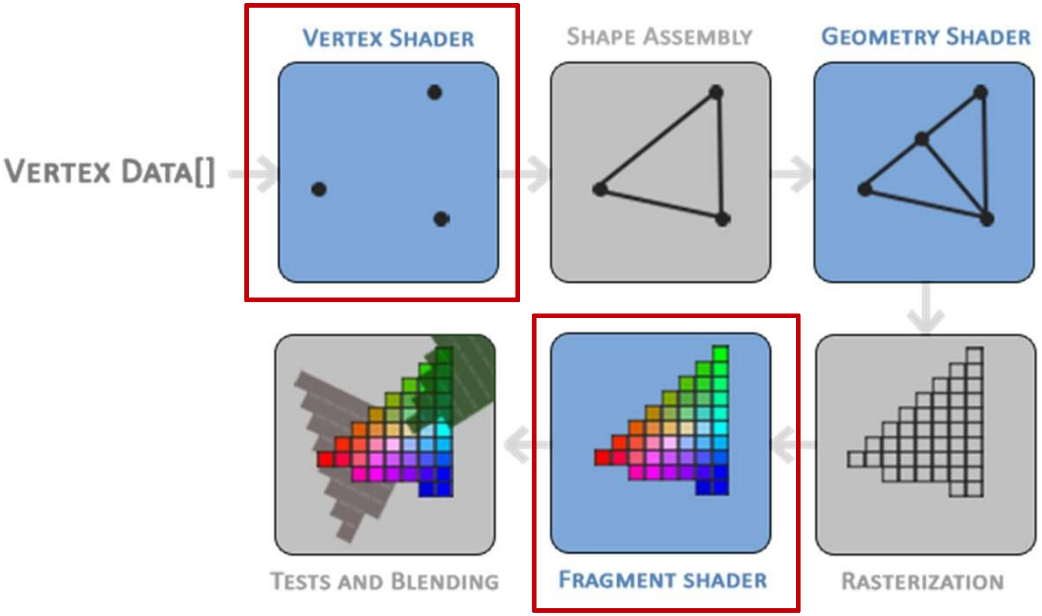


- Each shader has a precisely defined task:
  - **Vertex Shader:** Calculates the **final position of** each vertex in the output image.
  - **Fragment Shader:** Calculates the **colour of each pixel** in the output image
    - (Tesselation Shader: Decomposition of objects into finer triangular meshes)
    - (Geometry Shader: modification of geometry data at runtime)

Tradeoff storage space versus performance

Shader pipeline

Vertex Shader: Calculates the **final position of** each vertex in the output image.

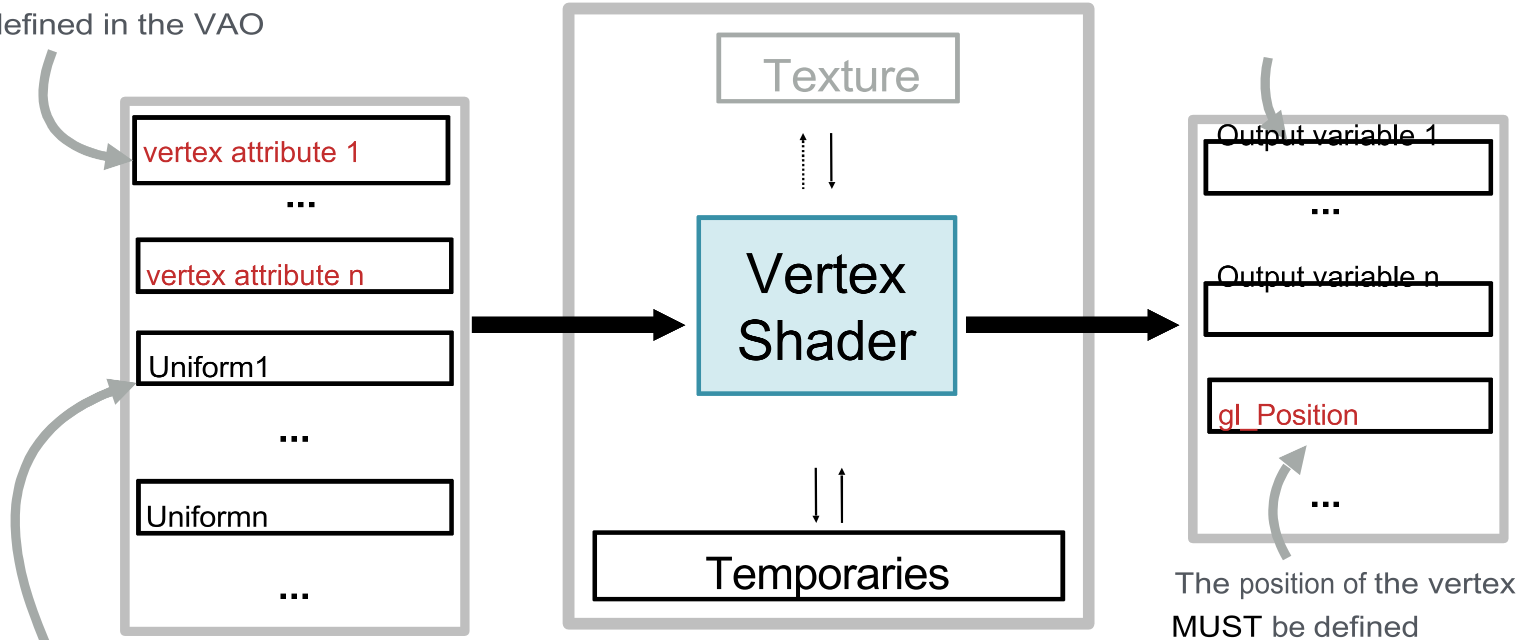


The geometry shader is an optional shader. It can transform individual primitives, such as triangles, and add additional vertices, for example.

Fragment Shader: Calculates the **colour of each pixel** in the output image

From: https://learnopengl.com/Getting-started/Hello-Triangle

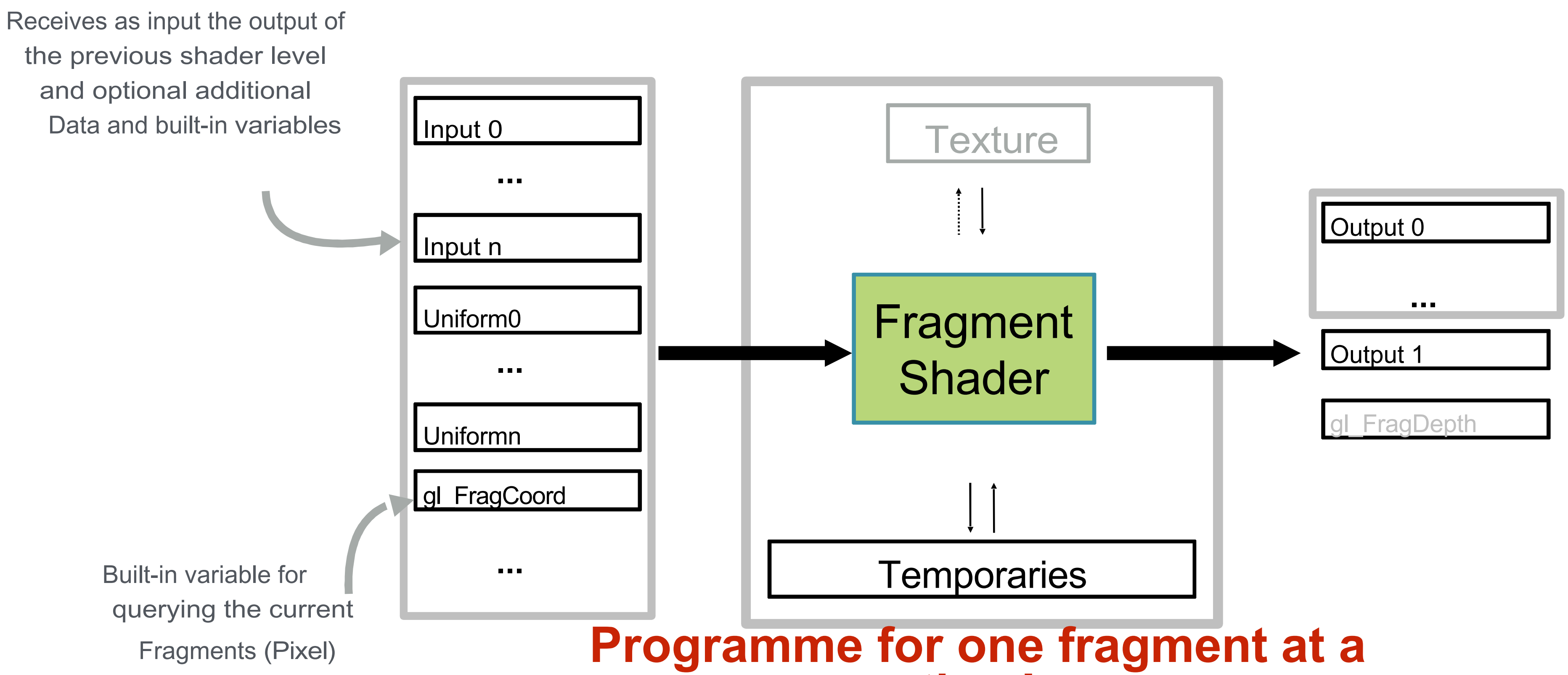
Vertex Shader Stage



Programme for one vertex at a time

Simplified: Images the input vertex on the screen

Fragment Shader Stage



Programme for one fragment at a time!

Simplified: Defines the output colour for each pixel

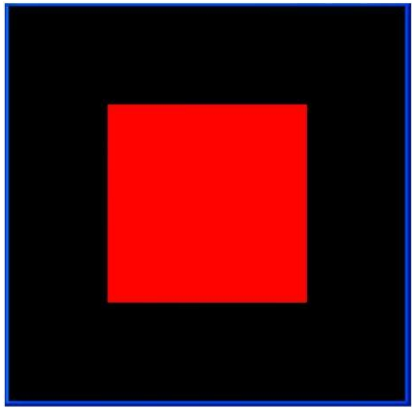
A simple example

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4
colour;

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



Output

A simple example

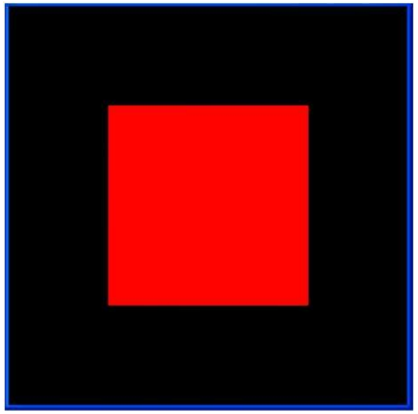
Vertex Shader

Definition of the OpenGL/GLSL version used

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4
colour;

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



Output

A simple example

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4
colour;

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

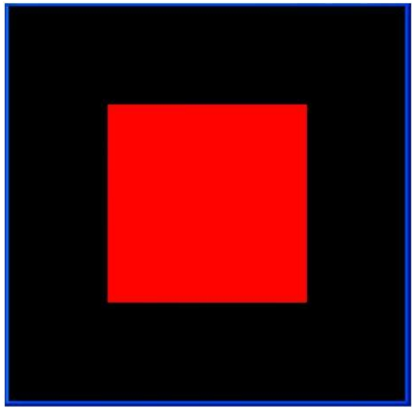
general form:

`layout(location = index) in type name;`

The index refers to the respective VAO attribute from

`glVertexAttribPointer(GLuint index, ...)`

□ **The programmer must know what the attribute contains.**



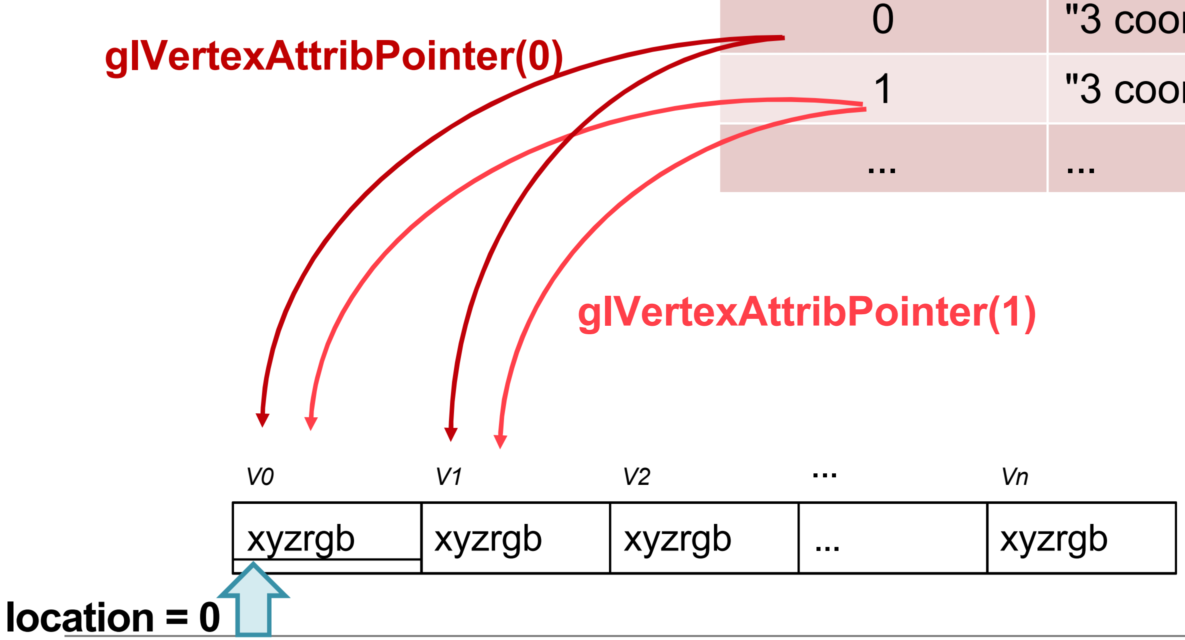
Output

Vertex Buffer Object (VBO), Vertex Array Object (VAO) and VertexAttribPointer

Vertex Array Object (VAO)

Task: Know the structure in which the data is stored in the VBO. The "VertexAttribPointers" are the objects in this array that store/know the storage structure of the data.

Attribute index	Description
0	"3 coordinates of the type GL_FLOAT per vertex".
1	"3 coordinates of the type GL_FLOAT per vertex".
...	...



Vertex Buffer Object (VBO)

Task: Collects the data and loads the data from the CPU to the GPU, from where it is then visualised, for example.

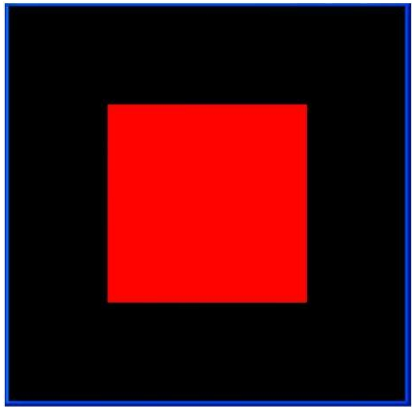
A simple example

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

```
#version 330
out vec4
colour;

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



Output

Data types

- Floatingpoint: `float, vec2, vec3, vec4`
- Integer: `int, ivec2, ivec3, ivec4`
- Boolean: `bool, bvec2, bvec3, bvec4`
- Matrix: `mat2 (2x2), mat3 (3x3), mat4 (4x4)`

- Access to different elements of the vector/matrix:
  .xyzw, .rgba, .stqp, [i], [i][j] (for matrices)

```
vec2 v2; vec3 v3; vec4 v4;
v2.x      // results in a float
v2.z      // Error: undefined for type
v4.rgba   // results in a vec4
v4.xy     // results in a vec2
v4.xgp    // Error: mismatching components
```

Data Types - Swizzling and Smearing

R-Values (reading):

```
v4.wzyx     //  correct, gives a vec4(w,z,y,x) correct,
v4.xxx      //  gives vec3(x,x,x)
v4.yyxx     //  correct, doubles x and y, gives a vec4(y,y,x,x) error: too
v2.yyzz     //  many components for type
```

L-Values (writing):

```
vec4 v4 = vec4( 1.0, 2.0, 3.0, 4.0);
v4.xw = vec2( 5.0, 6.0);              // 5.0, 2.0, 3.0, 6.0)
v4.wx = vec2( 7.0, 8.0);              // (8.0, 2.0, 3.0, 7.0)
v4.xx = vec2( 9.0,10.0);              // Error: x used twice
v4.yz = 11.0;                         // Error: Type mismatch
v4.yz = vec2( 12.0 );                 // (8.0, 12.0, 12.0, 7.0)
```

Predefined functions

If possible, use built-in functions instead of your own!

- Angles & Trigonometry:
  - `radians, degrees, sin, cos, tan, asin, acos, ...`
- Exponential functions:
  - `pow, exp, log, sqrt, ...`
- General functions:
  - `abs, ceil` (rounds the passed floating point value to the next higher natural number), `clamp` (restrict a value to a value between two other values), `min, max, ...`
- Geometric functions:
  - `cross, dot, length, normalize, reflect` (calculation of the reflection direction for an incident (light) vector), `...`

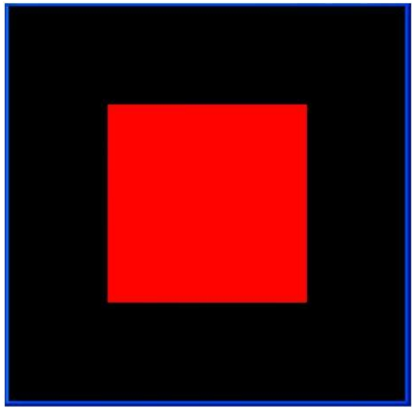
A simple example

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
}
```

Fragment Shader

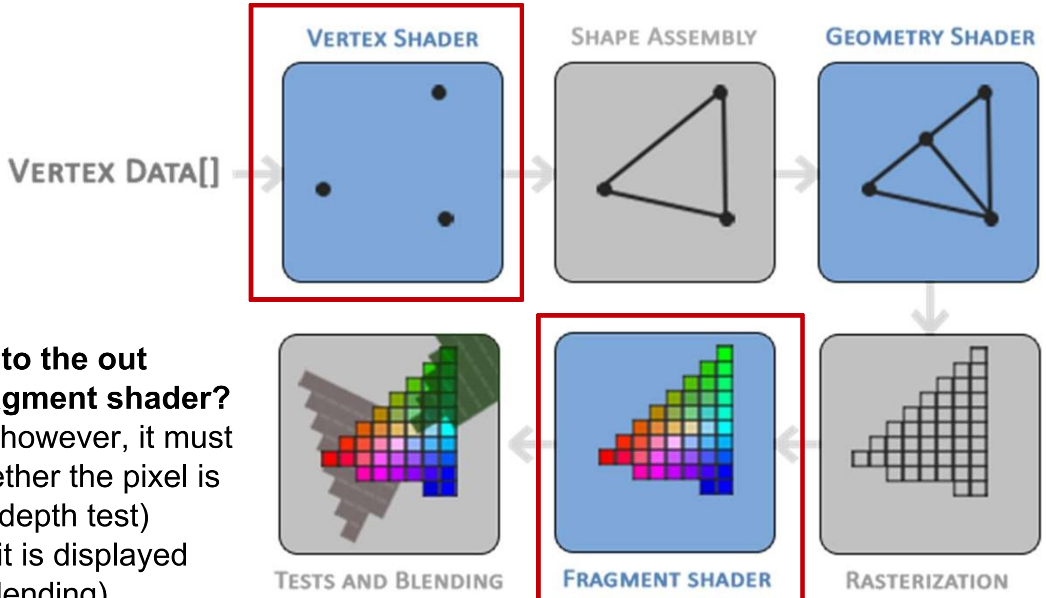
```
#version 330
out vec4 colour;

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



Output

Shader pipeline

Vertex Shader: Calculates the **final position of** each vertex in the output image.



The geometry shader is an optional shader. It can transform individual primitives, such as triangles, and add additional vertices, for example.

What happens to the out value of the fragment shader? In the last step, however, it must be checked whether the pixel is rendered at all (depth test) and/or whether it is displayed transparently (blending).

Fragment Shader: Calculates the **colour of each pixel** in the output image

From: https://learnopengl.com/Getting-started/Hello-Triangle

Storage types

- Required for communication between shaders and application
- **in**
  - Link to the shader from the previous stage
  - Input per vertex into the vertex shader of OpenGL or the application (READ-ONLY) or: Input per fragment for fragment shaders (READ-ONLY).
- **out**
  - Link from the shader to the next level
  - Passing vertex (READ/WRITE) to fragment shader for final output, interpolated
- **uniform**
  - Input into any shader program of OpenGL or an application (READ-ONLY)
  - Constant during the rendering process

Uniforms

- Uniforms are user-supplied variables from the application to the shaders

- **Upload uniforms**
  - Finding the memory location of the variables in the shader prog
    - `GLint glGetUniformLocation(GLuint programID, const GLchar *name);`
  - Upload uniform
    - `void glUniform{1,2,3,4}{i,f}(GLint location, GLfloat v0[, v1, v2, v3]);`
    - `void glUniformMatrix{234}fv(GLint location, GLsize count, GLboolean transpose, const Glfloat *value);`
    - `count` indicates the number of matrices to be changed. 1 if the target variable is not a matrix array and 1 or more if it is an array of matrices.

Addressing shader variables by variable name!

In our framework: **(for matrices)**

```
ShaderProgram::setUniform(string variable, TYPE value, [bool transpose]);
```

A simple example

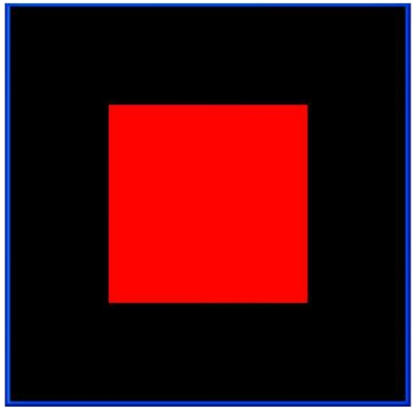
Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;


out vec4 passOn;

uniform mat4 model;

void main()
{

    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Same variable name and type!

Fragment Shader

```
#version 330
out vec4
colour;


in vec4 passOn;

void main()
{
    colour =
    passOn;
}
```



Output

A simple example

Vertex Shader

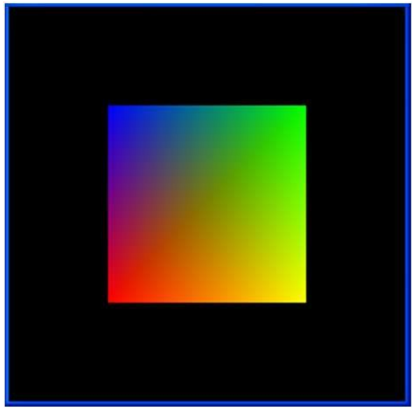
```
#version 330
layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 colour;

out vec4 passOn;

uniform mat4 model;

void main()
{

    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(colour, 1.0);
}
```

automatic interpolation

between the vertex data!

Fragment Shader

```
#version 330
out vec4
colour;

in vec4 passOn;

void main()
{
    colour =
    passOn;
}
```



Output

A simple example

Vertex Shader

```
#version 330
layout(location = 0) in vec3 vertex;
layout(location = 1) in vec3 colour;

out vec4 passOn;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(vertex, 1.0);
    passOn = vec4(colour, 1.0);
}
```

Multiplication of the old vertex exposure with the accumulated transformation matrix

Each vertex shader MUST fill the (built-in) variable gl_Position with the homogeneous vertex position!

Fragment Shader

```
#version 330
out vec4 colour;

in vec4 passOn;

void main()
{
    colour = passOn;
}
```

Definition of the output variables

Each fragment shader MUST define at least one output variable and fill it with the final colour of the fragment!

Uniforms - slide from chapter 6

- Uniforms are user-supplied variables from the application to the shaders

- **Upload uniforms**
  - Finding the memory location of the variables in the shader prog
    - `GLint glGetUniformLocation(GLuint programID, const GLchar *name);`
  - Upload uniform
    - `void glUniform{1,2,3,4}{i,f}(GLint location, GLfloat v0[, v1, v2, v3]);`
    - `void glUniformMatrix{234}fv(GLint location, GLsize count, GLboolean transpose, const Glfloat *value);`
    - `count specifies` the number of matrices to be changed. 1 if the target variable is not a matrix array and 1 or more if it is an array of matrices.

Addressing shader variables by variable name!

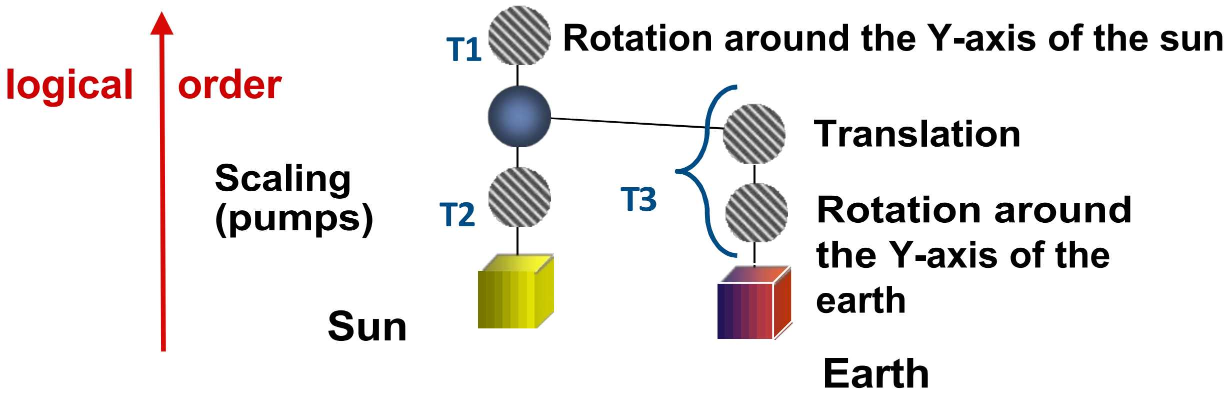
In our framework: **(for matrices)**

```
ShaderProgram::setUniform(string variable, TYPE value, [bool transpose]);
```

Scene graph - example implementation

Pseudocode:

```
Matrix4f T1 = new Matrix4f();          // Create new matrix T1
T1.rotate( alpha, 0.0, 1.0, 0.0);      // Rotation with angle alpha around the vector
                                          (0,1,0)t

Matrix4f T2 = new Matrix4f();          // Create new matrix T2
T2.scale( s, s, s);                    // uniform scaling by factor s

Matrix4f T3 = new Matrix4f();          // Create new matrix T3
T3.translate(p1, p2, p3);              // Translation relative to the sun, position (p1, p2,
                                          p3)
T3.rotate( beta, 0.0, 1.0, 0.0);       // Rotation with angle beta around the vector (0,1,0)t
```



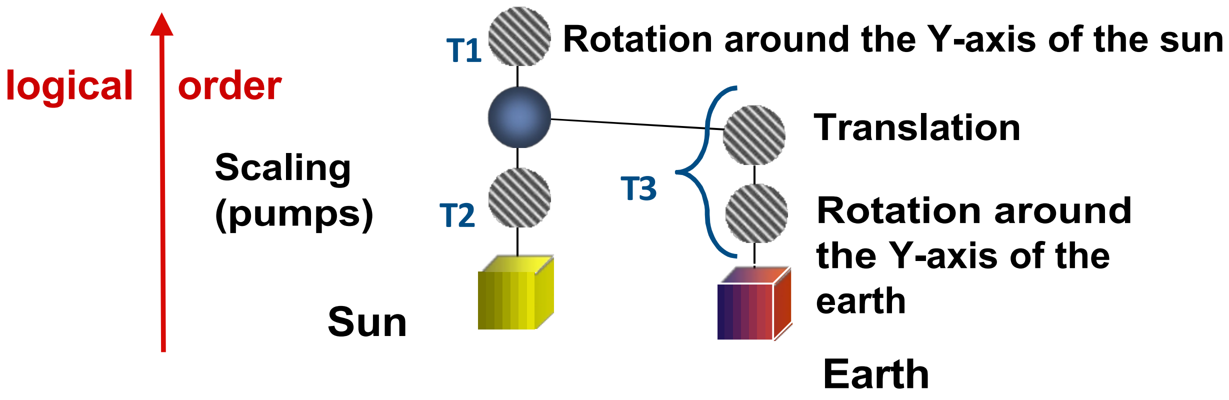
accumulated matrix for..

the earth: $_{ME}$ = T1* T3;

the sun: $_{MS}$ = T1* T2;

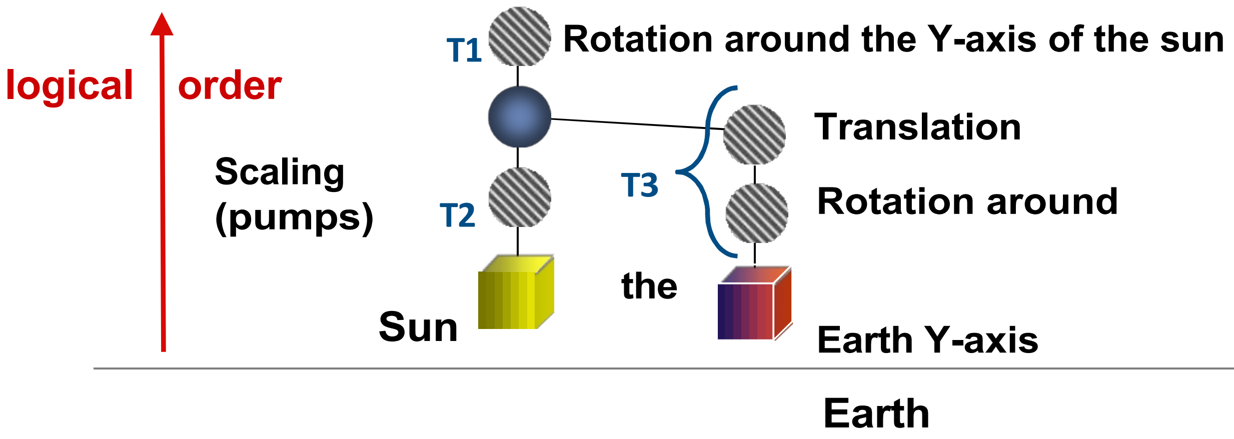
Extract from `void Scene::render(float dt)` in `Scene.cpp`

```
// First, the matrices T1, T2 and T3 are filled with the appropriate values T1
= new Transform; // Creates unit matrix
T1->rotate(glm::vec3(0, 0.2 * dt, 0)); // Rotation in the Y-axis.
                                        // The angle increases with by dt

T2 = new Transform; // Creates unit matrix
T2->scale(glm::vec3(scaling*dt, scaling*dt, scaling*dt)); // dt changes scaling

T3 = new Transform; // Creates unit matrix
T3->translate(glm::vec3(0.8f, 0, 0)); // scale the earth once T3-
>rotate(glm::vec3(0, 0.4f*dt, 0));
```



Code extract from `Scene.cpp`

```
//Loading the shaders in: bool Scene::init()
m_assets.addShaderProgram("shader",AssetManager::createShaderProgram("...vertex.glsl",
                                                                      ".../fragment.glsl"));
m_shader = m_assets.getShaderProgram("shader");
m_shader->use();
// In: void Scene::render(float dt)
// Implementation of the left scene graph branch - Sun: T1*T2
m_shader->setUniform("mm", T1->getTransformMatrix() * T2->getTransformMatrix(), false);
// Implementation of the left scene graph branch - Earth : T1*T3
m_shader->setUniform("mm", T1->getTransformMatrix() * T3->getTransformMatrix(), false),
```



The accumulation of the matrices is done by the Uniform-Parameters - see chapter 6

the earth: $_{ME}$ `= T1* T3;`

the sun: $_{MS}$ `= T1* T2;`