

Faculty of Science



Shading with OpenGL 3.3

Alexander Christensen

Department of Computer Science
University of Copenhagen

2018



Overview

- 1 Shading pipeline
- 2 glsl - OpenGL Shading Language
- 3 Shader programs
 - Vertex shader
 - Fragment shader
- 4 Examples



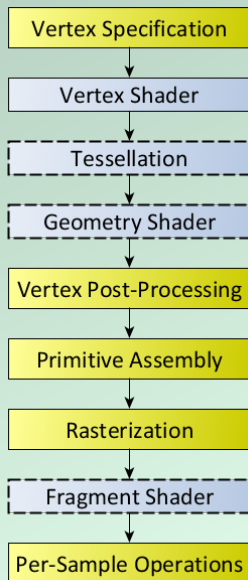
Shading pipeline - states

Primitives go through a pipeline before being rendered to screen.

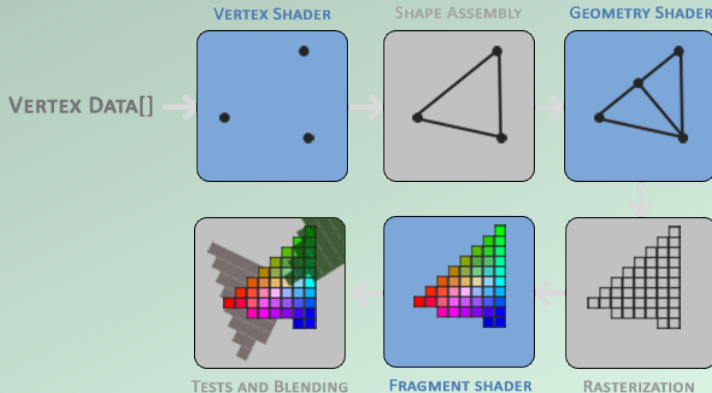
Each pipeline stage has a well-defined purpose.

All the stages are performed on the graphics hardware.

Some of these stages are programmable - we refer to these as shaders!



Shading pipeline - shading a triangle



Blue areas are the shaders we write (we will focus on vertex and fragment shaders).



glsl - OpenGL Shading Language

glsl is a DSL for writing programmable shaders

History with OpenGL (fixed-function pipeline vs. programmable shaders)

glsl has a C-like language syntax

Shaders are purely run on the GPU!



glsl - OpenGL Shading Language

Language features:

- primitive data types: `void`, `bool`, `float`, `int`, `uint`, ...
- vector/matrix data types: `mat2`, `mat3`, `mat4`, `vec2`, `vec3`, `vec4`, `bvec2`, `ivec2`, `uvec2`, ...
- special types: `struct`, `enum`, arrays
- functions, control flow (`switch`, `if`, `then`, `else`)
- overloaded operators (`+`, `-`, `*`, `/`) for built-in data types.
- Bitwise operators, logical operators, relational operators, ...
- texture samplers (`sampler2D`, `sampler3D`, ...)



glsl - OpenGL Shading Language

Built-in functions: ²

- **vector operations:** dot, normalize, length, distance, cross, reflect, refract
- **matrix operations:** outerProduct, transpose, determinant, inverse
- **trigonometry:** sin, cos, tan, asin, acos, atan, atanh, radians, degrees, ...
- **math:** pow, exp, exp2, log, log2, sqrt, inversesqrt
- **arithmetics:** ³ min, max, clamp, mix, step, smoothstep, sign, floor, ceil, fract, trunc, round, mod

²This is not a complete language reference.

³Most arithmetic functions also work on vectors.



glsl - OpenGL Shading Language

Language limitations:

- No while loops
- No recursion
- No pointers
- No exceptions
- No memory allocations (???)

Important guarantee: determinable running time!

Remarks:

Shader invocations are independent and run in parallel

Shader invocations cannot read/modify each other's values

Execution units on GPU's typically cannot do branch-prediction very well, so avoid `if`-statements as much as possible.



Shader programs - Vertex shader

Per-vertex processing

Built-in variables: `gl_Position`, `gl_PointSize`

Outputs a vertex position (x, y, z, w) , which must be in NDC. ⁴

Typical usage:

Camera projection, calculate vertex normals, animation

⁴Normalized Device Coordinates $x, y, z \in [-1, 1]$, $w \in [0, 1]$.



Shader programs - Fragment shader

Per-fragment processing.

Each fragment is typically the size of a pixel

Built-in variables: `gl_FragCoord`, `gl_FrontFacing`,
`gl_PointCoord`, `gl_FragDepth`

Outputs a color value (r, g, b, a) , in normalized coordinates $([0, 1])$.

Typical usage:

Lighting calculations, texture sampling, post-processing effects.



Shader programs - How many times is it run?

Imagine a triangle, in NDC, with coordinates $(-1, -1, 0)$, $(-1, 1, 0)$, $(1, 1, 0)$.

Assume application window of size 800×600

→ 3 vertex shader calls

→ $\approx (800 \cdot 600)/2 = 240000$ fragment shader calls! ⁵

Conclusion:

If we can do some calculation in vertex shader, we should.

⁵Could be more due to multisampling



Shader variables - Attributes

Attributes are used in vertex shaders:

```
layout (location = 0) in vec3 vertexPosition;  
layout (location = 1) in vec2 texCoord;
```

Data, such as vertex positions, are buffered to the GPU.
Attributes are pointers into this data.

```
GLfloat data[] = {  
    // vertexPosition    texCoord  
    0.0f, 0.5f, -0.3f,    0.0f, 0.0f,  
    // ...  
}
```



Shader variables - Uniforms



Shader variables - input/output

Input/Output variables allow shaders to pass variables down the pipeline. (!!!)

```
// vertex shader  
out vec4 vertexColor;
```

```
void main() {  
    vertexColor = vec4(1.0f, 0.0f, 0.3f, 1.0f);  
}
```

```
// fragment shader  
in vec4 vertexColor;
```



Creating a shader program

- Compilation
- Linking
- Activate program, thus setting the state of OpenGL (there is always a default shader)
- All drawing calls use the currently activated program (remember, OpenGL is a state machine)



Debugging

Black screen / weird rendering results?

- Check error status when compiling and linking your shader program.
- Check variables before they are sent to the GPU.
- Do only one thing at a time - verify that it works before next step.



Examples...

"Hello, world!\n" ...



Summary

We have been given a tour of the graphics pipeline for the OpenGL 3.3 specification.

We have looked at glsl, a DSL for describing programmable shaders in a C-like syntax.

We have seen some examples of how programmable shaders can be used to interpolate colors, perform linear transformations, and create animations.



References

