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
4. Shader variables
5. Examples

Shading pipeline - stages

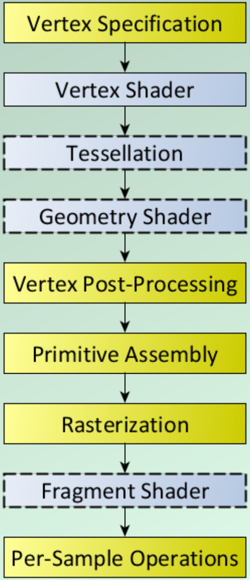
Primitives go through a pipeline before being rendered to screen.

Each pipeline stage has a well-defined purpose.

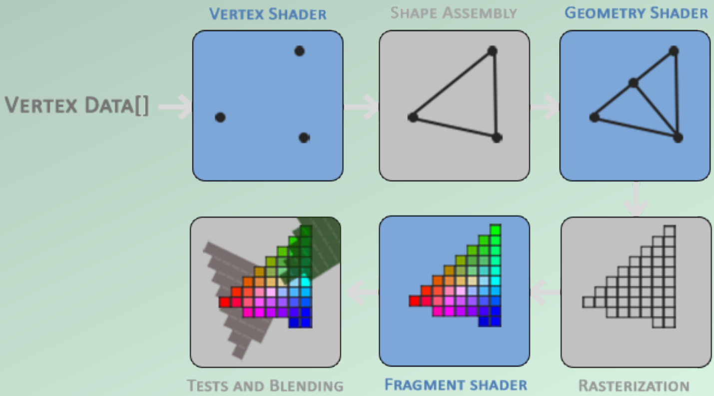
All the stages are performed on the graphics hardware.

Some of these stages are programmable -

we refer to them 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

Shaders are purely run on the GPU!

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

glsl has been available since OpenGL 2.2 [2]

glsl has a C-like language syntax

[2] April 2004

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`, ...)
- Preprocessor directives (`#define MAX_HEIGHT 127.0f`)

glsl - OpenGL Shading Language

Built-in functions: [3]

- **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:** [4] `min`, `max`, `clamp`, `mix`, `step`, `smoothstep`, `sign`, `floor`, `ceil`, `fract`, `trunc`, `round`, `mod`

---

[3] This is not a complete language reference.

[4] Most 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. [5]
Built-in variable `gl_Position` must be set!

Typical usage:

Camera projection, calculate vertex normals, animation

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

Shader programs - Fragment shader

Per-fragment processing.
Each fragment is typically the size of a pixel [6]
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.

[6] Can be smaller if multisampling is enabled.

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
$\rightarrow$ 3 vertex shader calls
$\rightarrow$ $\approx (800 \cdot 600)/2 = 240000$ fragment shader calls! [7]

Conclusion:

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

[7] Could be more due to multisampling

Shader variables - Attributes

Attributes are used in vertex shaders:

```
// vertex shader
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.

```
// application
GLfloat data[] = {
    // vertexPosition    texCoord
    0.0f, 0.5f, -0.3f,   0.0f, 0.0f,
    // ...
}
glBufferData( ... );
```

Shader variables - Uniforms

Uniforms are immutable values that are shared across all shader stages

They can be arbitrary data types

```
// vertex shader
uniform mat4 model; // model matrix
uniform mat4 viewPerspective; // projection matrix
uniform vec3 cameraPosition;


// fragment shader
struct pointLight {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    float specular;
};
uniform pointLight lamp;
uniform vec3 cameraPosition;
```

Shader variables - input/ouput

Input/Output variables allow us to pass variables down the pipeline, from one shader to the next.

```
// vertex shader
out vec4 vertexColor;

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


// fragment shader
in vec4 vertexColor; // this color value will be smoohtly interpolated
```

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

The geforce 6 series gpu architecture.
https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter30.html.
Accessed: 2018-12-16.

Rendering pipeling overview.
https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.
Accessed: 2018-12-16.

Learnopengl.
http://learnopengl.com.
Accessed: 2018-12-16.