Computer Graphics 1

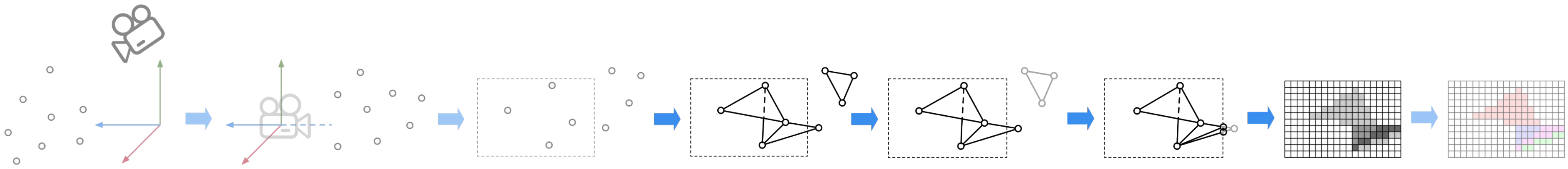
Tutorial 6 Rasterization II

Summer Semester 2021

Ludwig-Maximilians-Universität München

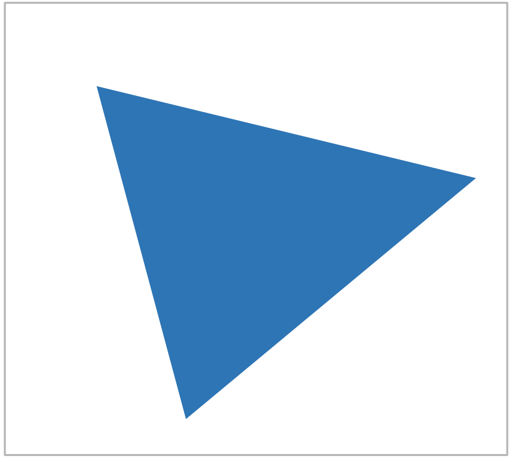
Tutorial 6: Rasterization II

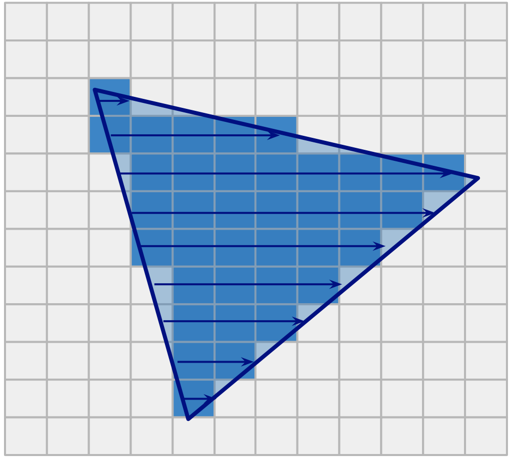
- Drawing Sampling
  - Issues with Bresenham algorithm
  - Point-in-triangle assertion
  - Anti-aliasing
- Modern Rasterization Rendering Pipeline
  - Shader language and shader programs
  - OpenGL Shading Language (GLSL)
  - Vertex Shader
  - Fragment Shader
  - Debugging Shaders

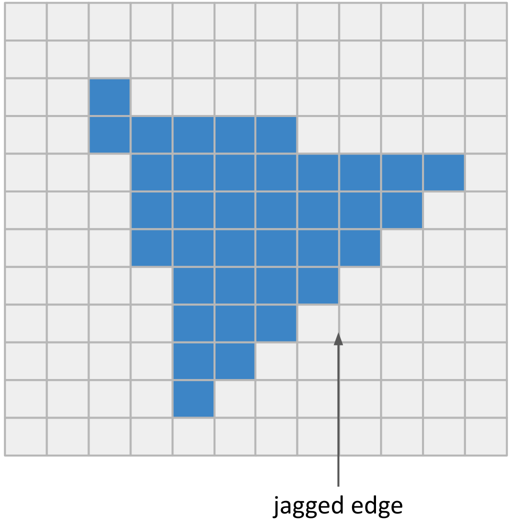


Issues with Bresenham and Scan Line Algorithms

- *Performance*: Desire parallelized execution for all pixels but drawing a line from left to right is sequential
- *Aliasing*: scan converted objects exhibit discretization artifacts (staircase effect)







Point-in-Triangle Assertion

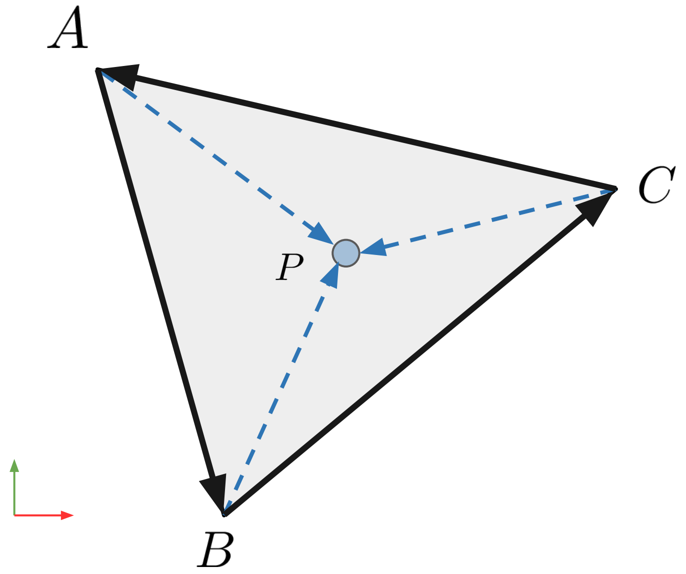
Basic idea: If P is always on the left of all edges

P is on the left side of AB: $\langle \overrightarrow{AB} \times \overrightarrow{AP}, (0,0,1,0)^\top \rangle$

P is on the left side of BC: $\langle \overrightarrow{BC} \times \overrightarrow{BP}, (0,0,1,0)^\top \rangle$

P is on the left side of CA: $\langle \overrightarrow{CA} \times \overrightarrow{CP}, (0,0,1,0)^\top \rangle$

⇒ P is inside triangle ABC



Alternative to scan line algorithm for triangle drawing:

For all pixels in the AABB of a given ABC, if a pixel is inside the triangle, then draw the pixel.

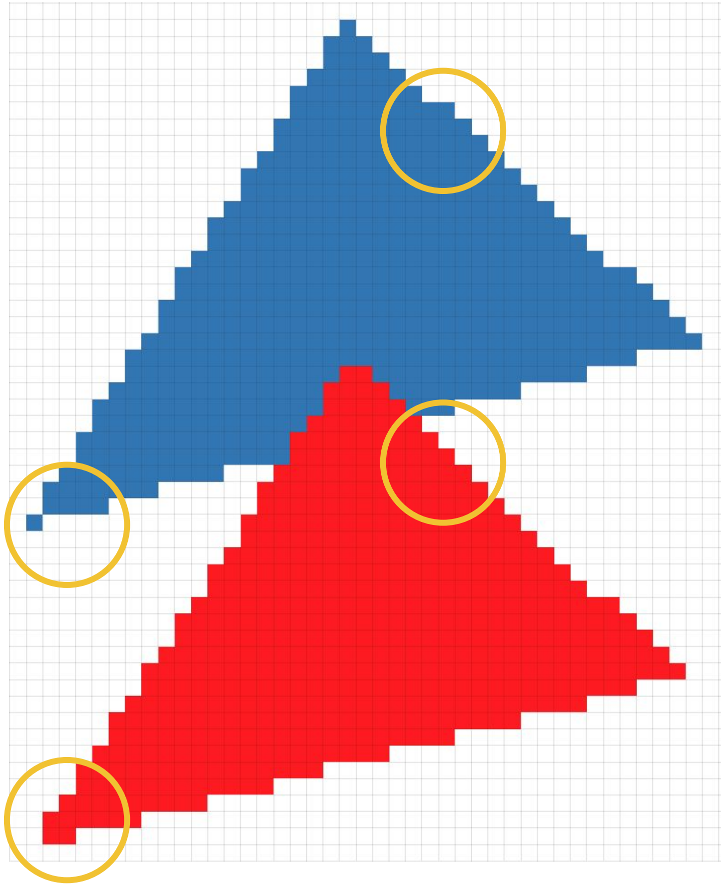
Point-in-triangle assertion is implemented on the GPU as fixed, specialized function. The GPU executes this test for all pixels parallelly and efficiently. Testing point-in-triangle is the most practical and efficient approach to draw a triangle.

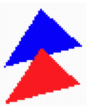
Scan line vs. Point-in-Triangle Assertion based Drawing

- Scanline algorithm embeds numeric issue inside the algorithm design: when should the coordinates of a vertex position be numerically rounded (i.e. which pixel to initiate the drawing)?
- Point-in-triangle assertion is a boolean assertion to check if pixel center is inside the triangle, and can be easily optimized and executed in parallel

Scan Line Drawing

Point-in-Triangle Drawing

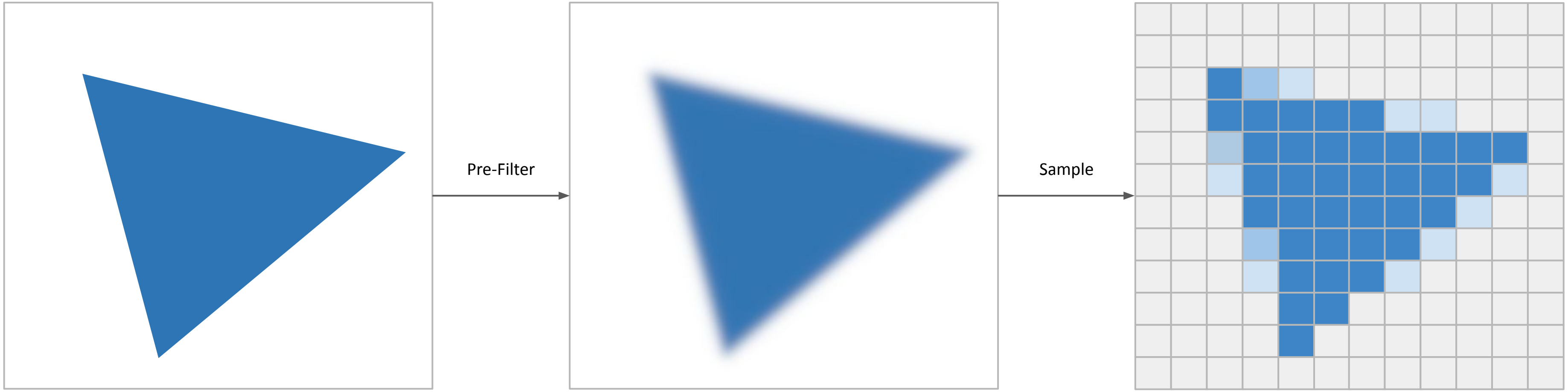




*Corner case: if a pixel center is exactly at the edge of the triangle: decide yourself in the implementation

Aliasing and Antialiasing

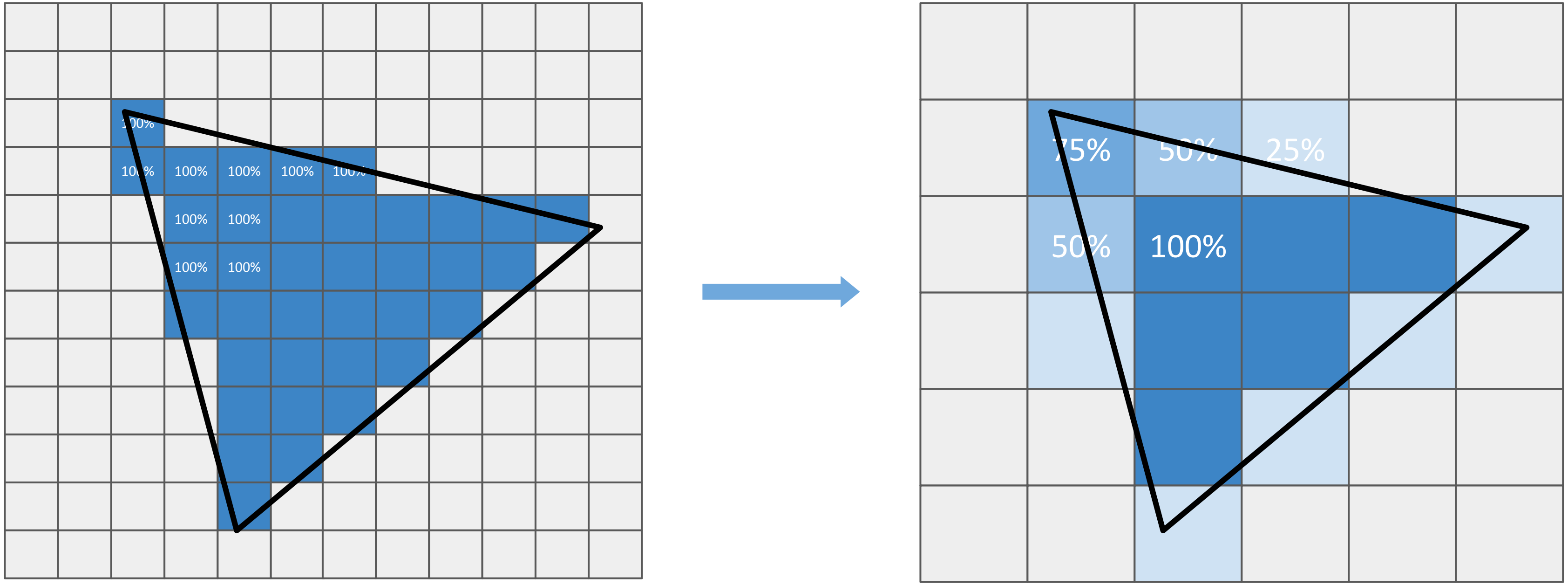
- Both scan line algorithm, and point-in-triangle assertion based drawing introduces line *aliasing* issue
- How to reduce aliasing issue?
  - Higher resolution display (therefore higher frame buffer) i.e. + €€€
    - Disadvantage: adds more computation cost to software, and needs high resolution on hardware
  - **Antialiasing**



Multi-Sample Anti-aliasing (MSAA)

Multi sample antialiasing (MSAA): Sampling high resolution samples then render in a lower resolution

MSAA computes the coverage of a triangle area on a pixel



2x2 Super sampling **Averaging down**

Antialiasing Today

The antialiasing methods that appear in many video games:

- Fast Approximate Antialiasing (FXAA, 2009)
- Temporal Antialiasing (TXAA, 2012)
- Deep Learning Super Sampling (DLSS 2.0, 2020)

Breakout: Implement Point-in-Triangle Assertion

Enter folder demos/06-raster2/1-draw (live demo)

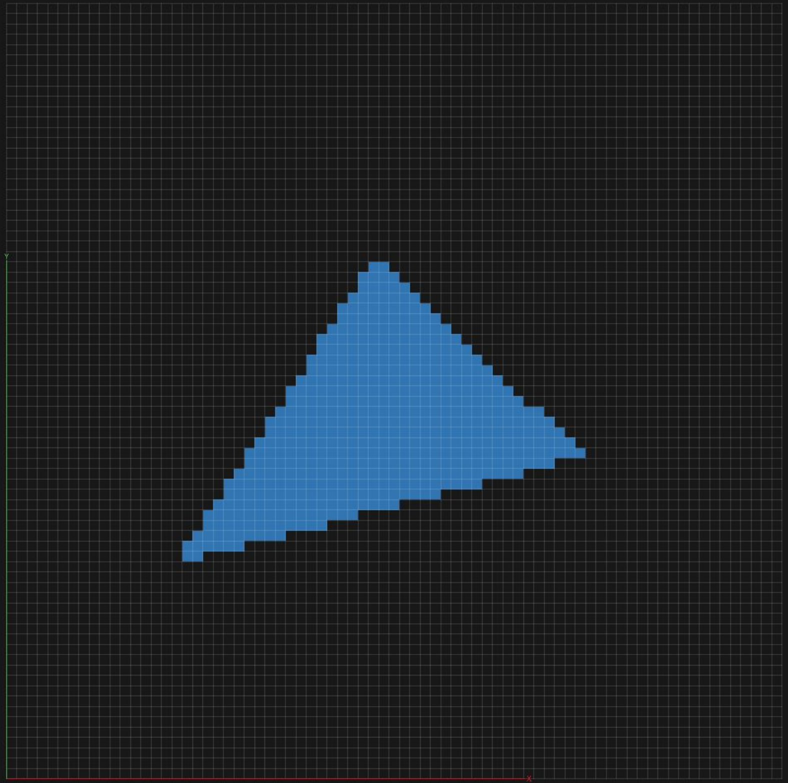
Look for TODO comments in the `main.ts`

1. Implement the `drawTriangle` function for the point-in-triangle assertion based drawing of a given triangle.

2. Modify vertex positions of the triangle, answer these questions:

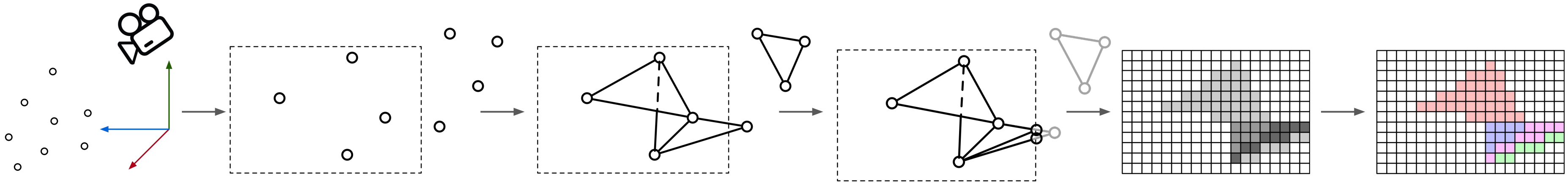
How efficient is the point-in-triangle assertion?

Does the shape of the triangle impact the performance?



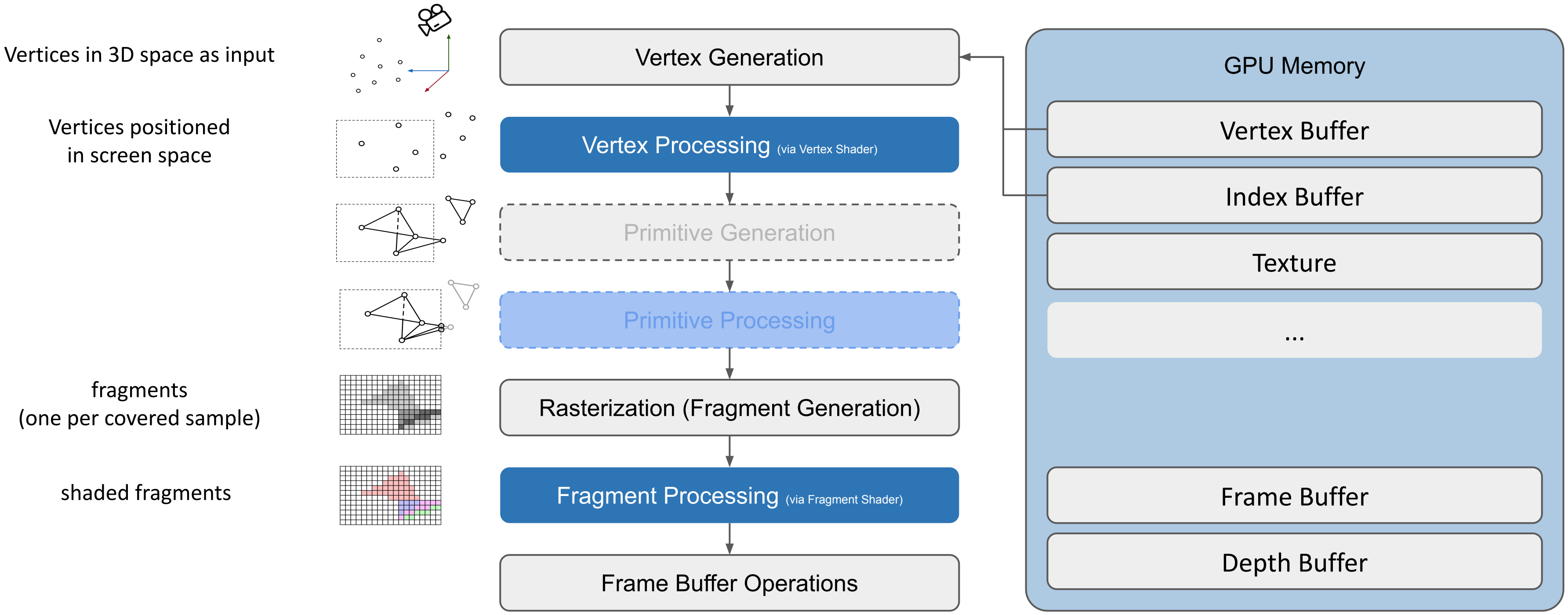
Tutorial 6: Rasterization II

- Drawing Sampling
  - Issues with Bresenham algorithm
  - Point-in-triangle Assertion
  - Anti-aliasing
- Modern Rasterization Rendering Pipeline
  - Shader language and shader programs
  - OpenGL Shading Language (GLSL)
  - Vertex Shader
  - Fragment Shader
  - Debugging Shaders



Modern Rasterization Rendering Pipeline (on GPU)

The pipeline can be executed for multiple ***pass***es, and one rendering ***pass*** means: 1) create a frame buffer, 2) specify one or more buffers as output, and 3) render content from an output buffer



OpenGL *Deprecated!*

OpenGL is a standardized set of APIs that describes the previous rasterization rendering pipeline on a GPU.

Advantage:

- Cross platform

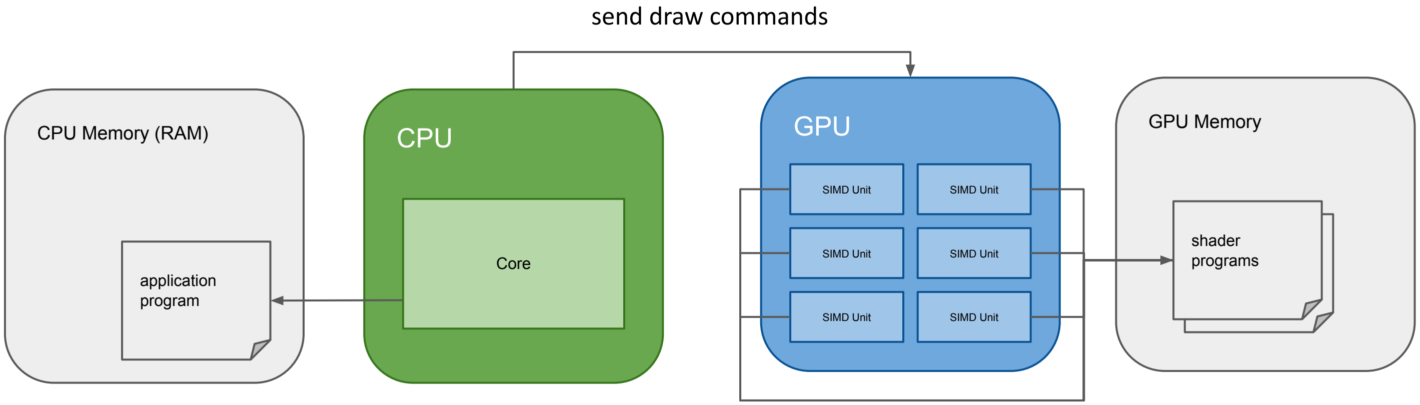
Disadvantages:

- Compatibility: different versions have different set of APIs or different API behaviors
- State-machine programming model, C-style and not easy to use
- Debugging is (or was) non-trivial

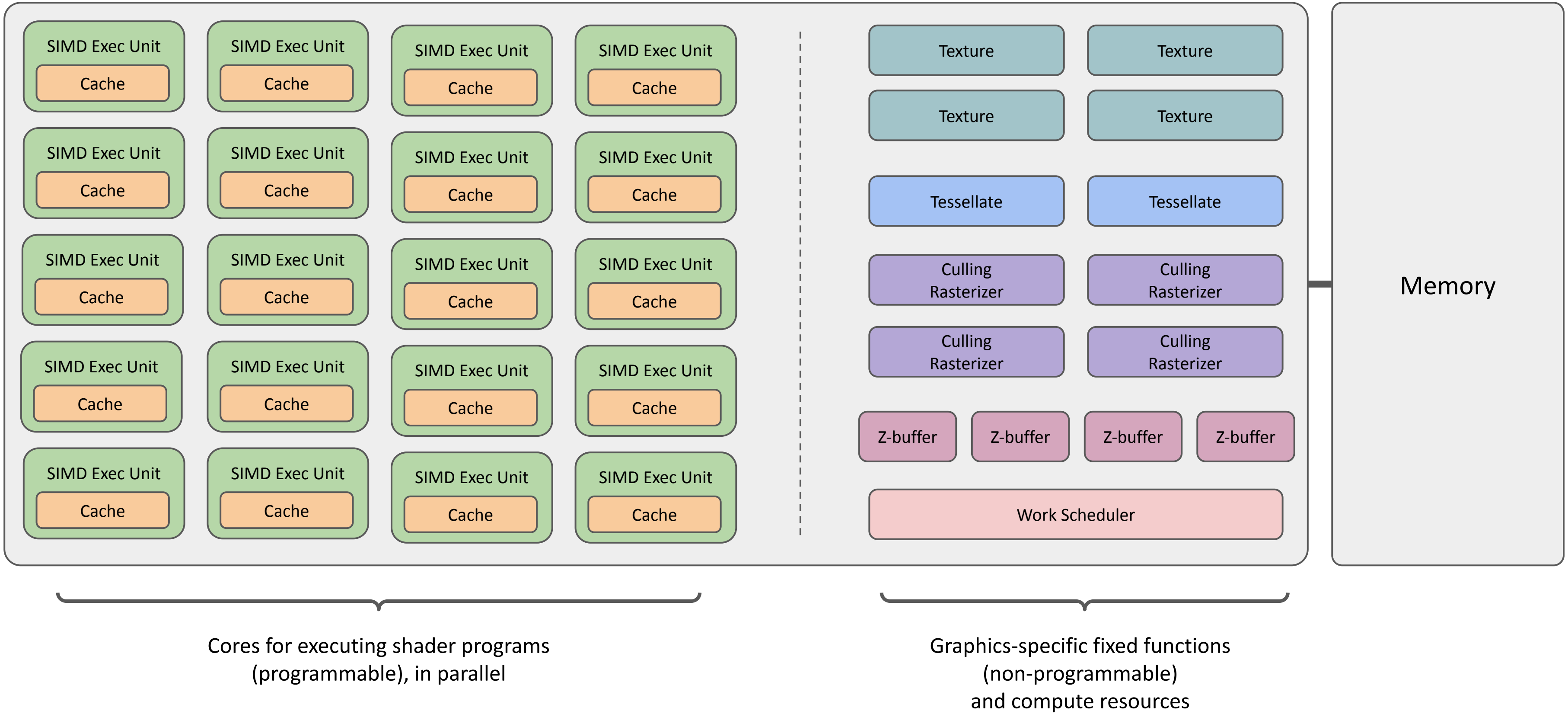
Fore more, see http://docs.gl/. We will not discuss OpenGL in detail. Instead...

Shader Program and Shading Language

- Shader is a small program that runs on GPU instead of CPU
- Shader programs are written in language similar to C but with restrictions, called *shading language*
- To run a shader program (on GPU), similar to CPU programs, one must:
  1. create shaders for compilation
  2. compile shaders for execution
  3. link shader programs together and the application
  4. use shader program when necessary

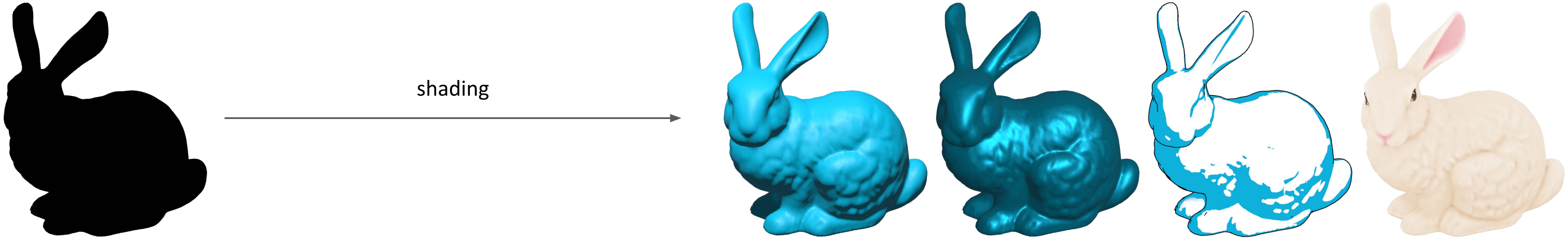


Executing Shaders on a Multi-core Processor (GPU)



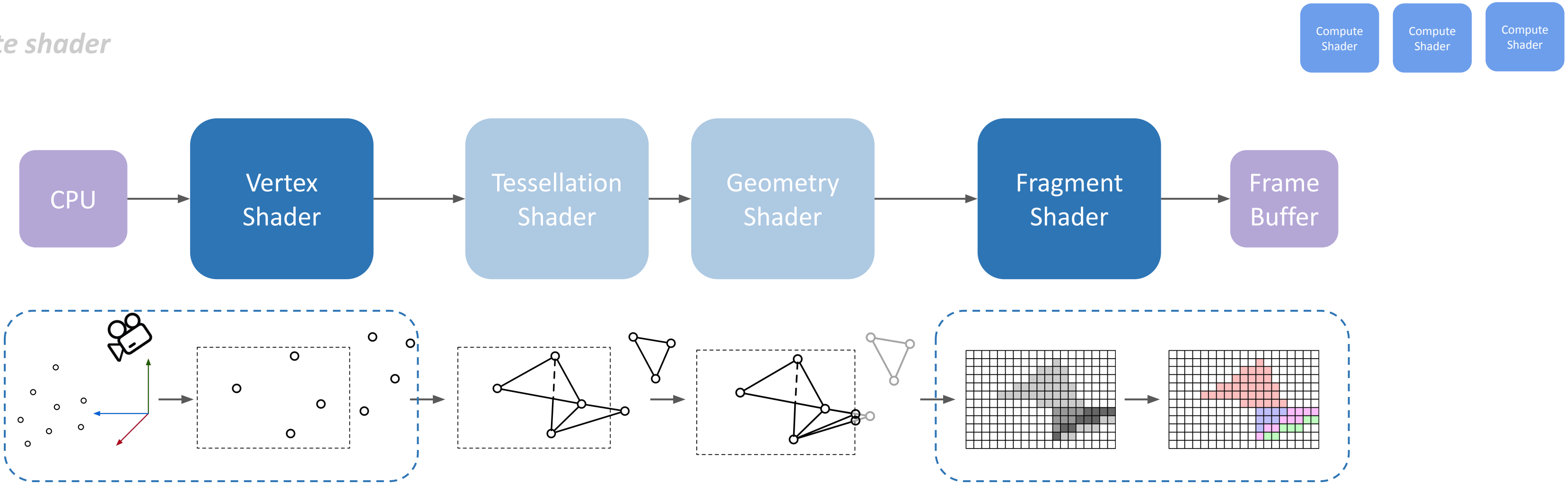
Why A Language?

- High-level, domain-specific language to describe *shading behavior*
  - Better utilize GPU and can customize
  - In ancient times: assembly on GPUs
  - e.g. *GLSL* in OpenGL, *HLSL* in DirectX
- Shading is a *local* behavior for a specific material
- A rasterizer turns geometries into pixels via sampling but does not include the process of how to figure out what is the "correct" color of a pixel, e.g. different shading behavior



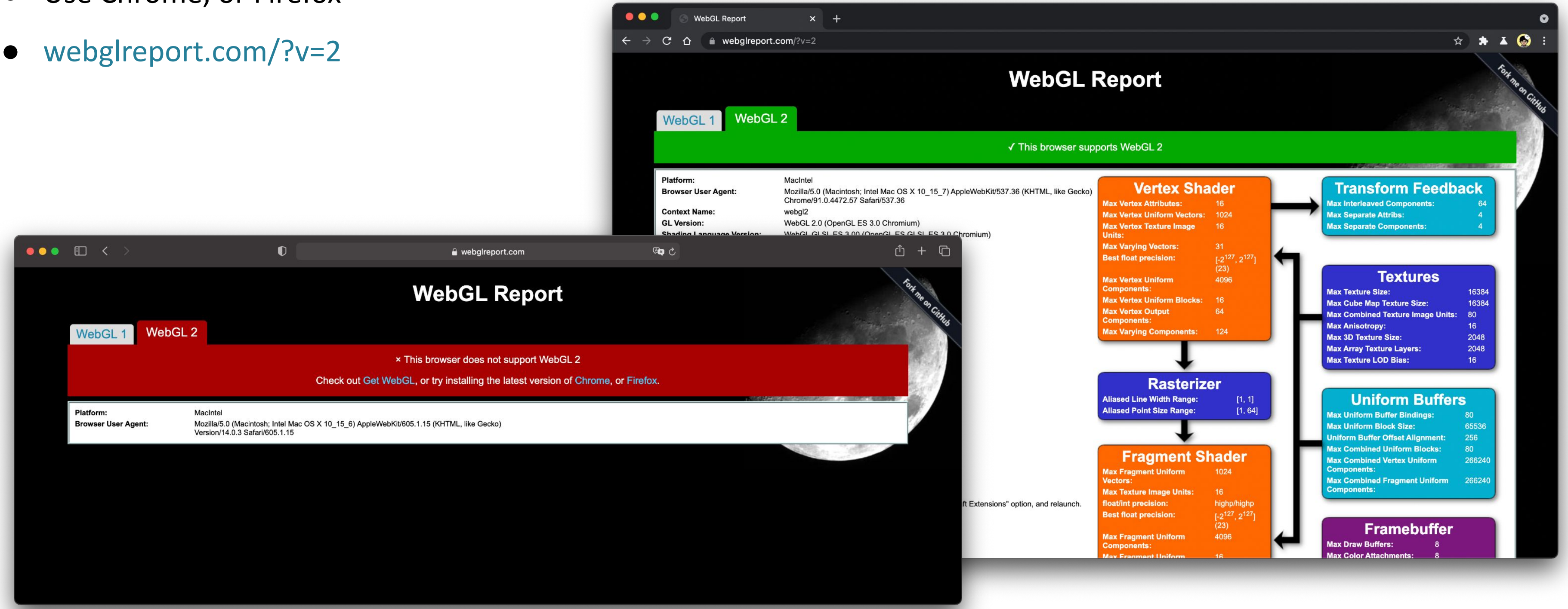
OpenGL ES Shading Language (GLSL ES)

- GLSL ES (shortly GLSL) enables programmable stages of graphics pipeline computing using GPU in WebGL
- Different shader stages
  - ***vertex shader***
  - *tessellation shader*
  - *geometry shader*
  - ***fragment shader***
  - *compute shader*
  - ...



WebGL Shader Support

- Safari doesn't work with WebGL2 (why Apple? why?)
- Use Chrome, or Firefox
- webglreport.com/?v=2



Basic GLSL Concepts

- ***types:*** int, float, vec2, vec3, vec4, mat2, mat3, mat4, sampler2D, struct, array

```
vec4 position; // position.x, position.yz (result in vec2) to access components
vec4 normal;   // x, y, z, w for coordinates
vec4 uv;       // s, t, p, q for texture coordinates
vec4 color;    // r, g, b, a for color channels
```

- ***quantifiers***: in, out, inout, uniform

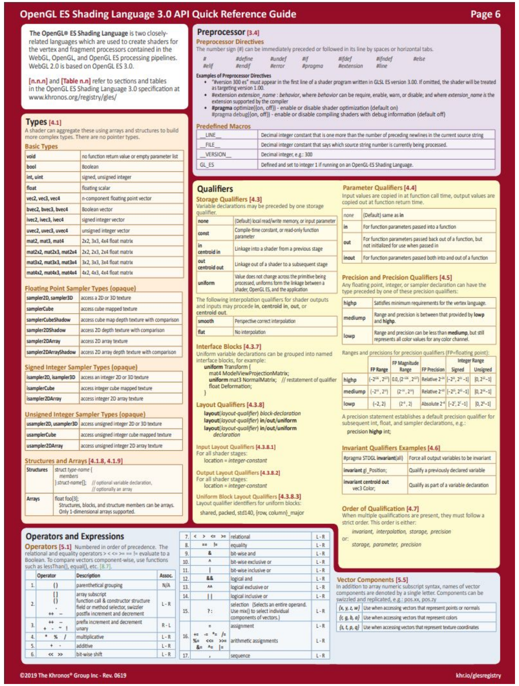
```
in  vec4 color;               // an input of a shader
out vec4 colors;              // an output of a shader, "varyings" before WebGL 2.0
uniform float ka, kd, ks, p; // constant (but not compile-time constant)
```

- ***function***: a code block maps a list of parameters to a list of return values

```
float rand(vec2 co){          // generates a random number
   return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453);
}
```

- ***flow control***: `if/else/for` statements (in almost every-language)

See a summarized reference note: https://www.khronos.org/files/webgl20-reference-guide.pdf, page 6-8



Attributes and Uniforms

- (Vertex) attributes are user defined
  - Global variables that can be different for each vertex (e.g. normal vector)
  - Read-only, **only available in Vertex Shader**
  - Definable in program code
- uniforms are
  - Parameters that are the same for many/all vertices/primitives are defined, they are identified via their GLSL variable names (analogous to attributes)
  - Each variable is assigned a "location" (index)
    - compare strings more efficiently than with every change
  - Can be read in vertex and fragment shaders (read-only)

Shader Programs: Vertex Shader

- Transformation of single vertices and their attributes (e.g. normals, ...)
  - No vertex generation
  - No vertex destruction (handled by clipping)
- Calculation of all attributes that remain constant per vertex
  - Saves computing time compared to the Fragment Shader
  - e.g. lighting by vertex (old-fashioned)
- Set attributes to be interpolated per fragment
  - e.g. normals for per-pixel lighting
- **`gl_Position`**: *must* be written in the vertex shader.
- Determines the position of the vertices, otherwise cannot continue to the subsequent stages of the pipeline.

Example: A Minimum Vertex Shader

```
in vec3 position;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main() {
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Built-in output attribute for Vertex Shader (required) → `gl_Position`

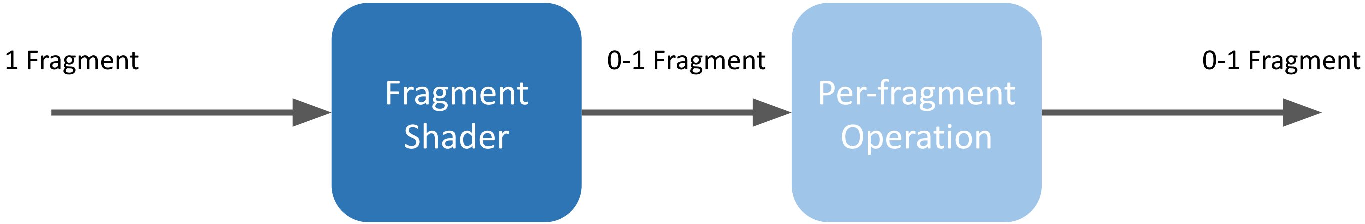
Perspective/Orthographic Projection → `projectionMatrix`

Model and View Transformation → `modelViewMatrix`

Model Position → `position`

Shader Programs: Fragment Shader

- Allows calculation per result pixel that ends up in the output buffer
  - Per-pixel lighting/shading
  - Sampling of data within the primitive, e.g. for
    - volume rendering
    - Implicit surfaces, glyphs
- The **in** attributes are *interpolated* (discussed later) within the primitive (can be turned off)
- Fragments can be discarded: `discard`
- Fragment operations: Tests, blending and etc.
- The **`out`** (in Fragment Shader): stores the color of a fragment.



Example: A Minimum Fragment Shader

```
out vec4 outColor;
void main() {
   outColor = vec4(1.0, 1.0, 0.0, 1.0); // yellow
}
```

Output color of the current fragment (pixel)

Red channel

Green channel

Blue channel

Alpha channel

Shader Support in three.js

Similar to all graphics APIs, three.js treats shader programs as string input, and supports `ShaderMaterial` and `RawShaderMaterial` for customizable shaders.

The `RawShaderMaterial` compiles raw shaders without any additional information. For the better collaboration with three.js internal states (e.g. transformation matrices). `ShaderMaterial` adds convenient default uniform and attributes.

In a vertex shader:

```
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 cameraPosition;
in vec3 position; // vertex position
in vec3 normal;   // vertex normal
in vec2 uv;       // vertex UV
in vec4 color;    // vertex color
```

In a fragment shader:

```
uniform mat4 viewMatrix;
uniform vec3 cameraPosition;
```

*There are more default uniform and attributes, see https://threejs.org/docs/index.html#api/en/renderers/webgl/WebGLProgram

Using `ShaderMaterial` in three.js

Similar to all graphics APIs, to use shader in three.js, pass shader program as a string to the material of a mesh, then three.js will do the rest of the job for us:

```
import vert from './shaders/vs.glsl';
import frag from './shaders/fs.glsl';

... // create geometry

const mesh = new Mesh(geometry, new ShaderMaterial({
  vertexColors: true,   // use vertex colors the are specified in three.js
  glslVersion: GLSL3,   // use the latest GLSL version (3.0)
  vertexShader: vert,   // vert is a loaded string
  fragmentShader: frag, // vert is also a loaded string
}));
this.scene.add(mesh);
```

Example: A Minimum Vertex Shader using `ShaderMaterial`

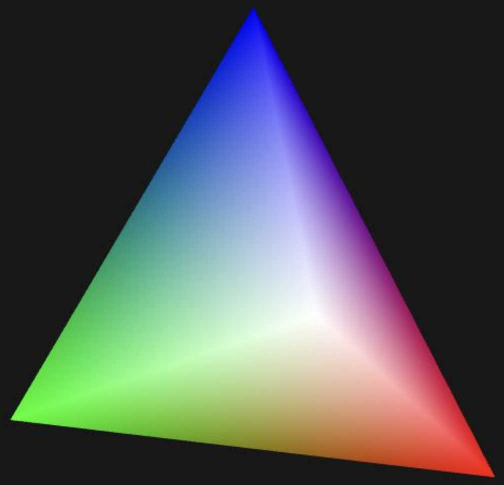
```
in vec3 position;               // provided by three.js automatically
uniform mat4 modelViewMatrix;   // provided by three.js automatically
uniform mat4 projectionMatrix;  // provided by three.js automatically

void main() {
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

Breakout: Getting Started with GLSL

In folder demos/06-raster2/2-glsl (live demo), look for TODO comments in the `main.ts`, `shaders/vs.glsl` and `shaders/fs.glsl`, Implements the two shaders (vertex and fragment) for the tetrahedron we had worked in the previous geometry tutorial breakout.

Answer: How does the colors of the tetrahedron vertices be visualized?



Breakout: Getting Started with GLSL

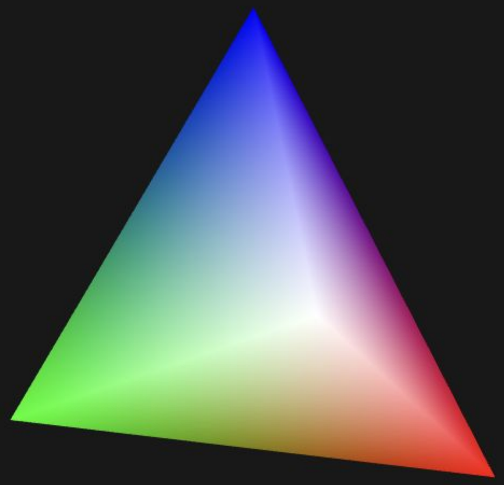
The color propagates along: Vertex color attributes ⟶ ShaderMaterial vertex color enabled ⟶ VertexShader vertexColor ⟶ Fragment Shader vertexColor ⟶ Fragment Shader outColor ⟶ Display

1. Vertex shader implementation

```
out vec3 vertexColor;
void main() {
   gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(
       position.x, position.y, position.z, 1.0
   );
   vertexColor = color;
}
```

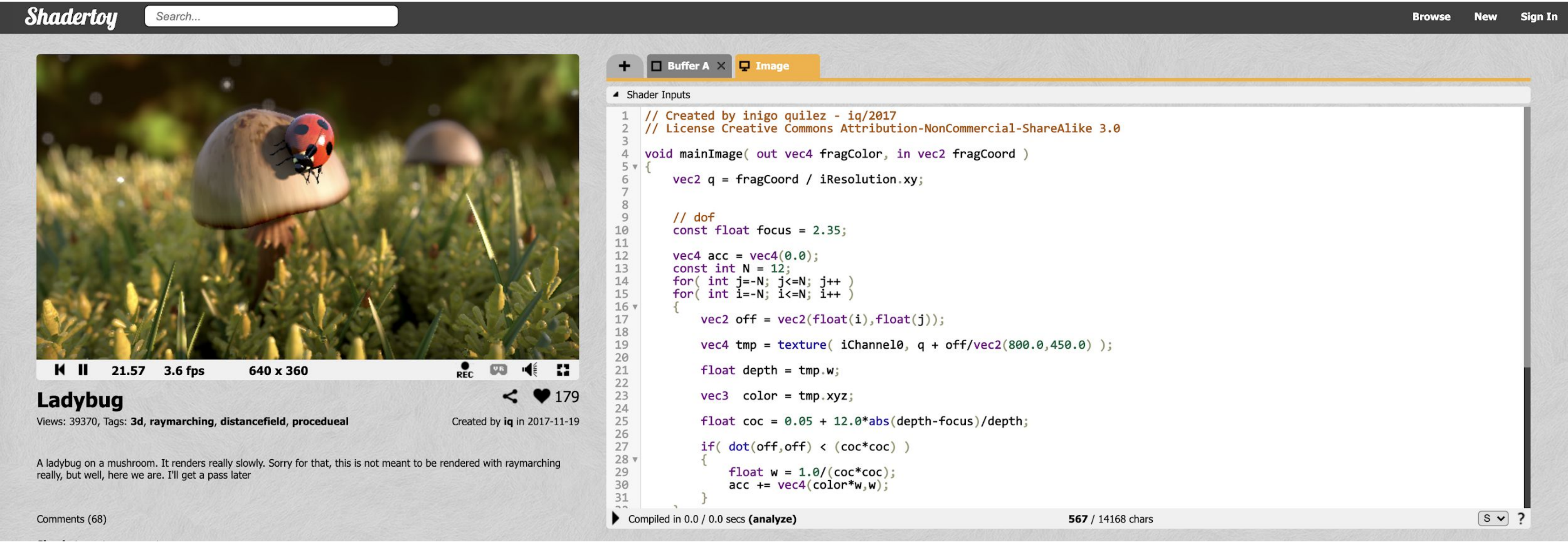
2. Fragment shader implementation

```
in vec3 vertexColor;
out vec4 outputColor;
void main() {
   outputColor = vec4(vertexColor, 1.0);
}
```



Shaders are powerful!

- Shaders can do more than you might think, **but also non-trivial to write**
- ~800 lines of code:



https://www.shadertoy.com/view/4tByz3

Compute Shader

- Compute shaders allows general purpose, parallel computing on the GPU (with many many cores)
  - Examples: Physics calculations, particle systems, fluid or substance simulations
- Compute shader is located outside the rendering pipeline
  - No input from inside the pipeline and no output to the pipeline
- Can read and write textures, images and shader buffers
- WebGL Support
  - No support, and will not be supported :(
  - (Yet) very early alpha support in WebGPU and requires Chromium nightly builds

Debugging Shaders

Difficulties in order to debug shaders:

- print out values: shaders are executed on GPU but print out is on CPU
- set breakpoints: shaders are executed on GPU in parallel and unclear which and what break (sometimes)

Traditional debuggers are less used with increasing coding experience because:

- Most difficult errors in complex programs are conceptual bugs where the wrong thing is being implemented
- It is easy to waste large amounts of time stepping through variable values with without detecting such cases
- Tools are platform and hardware specific. For example: RenderDoc (no macOS support, why Apple? why?)

⇒ Review the code carefully can solve almost all problems

Tool-independent, most general approach: **Just render value as color then use a color picker to get the output value**

Breakout: Experiment with Shaders

Enter folder demos/06-raster2/3-shaders (live demo)

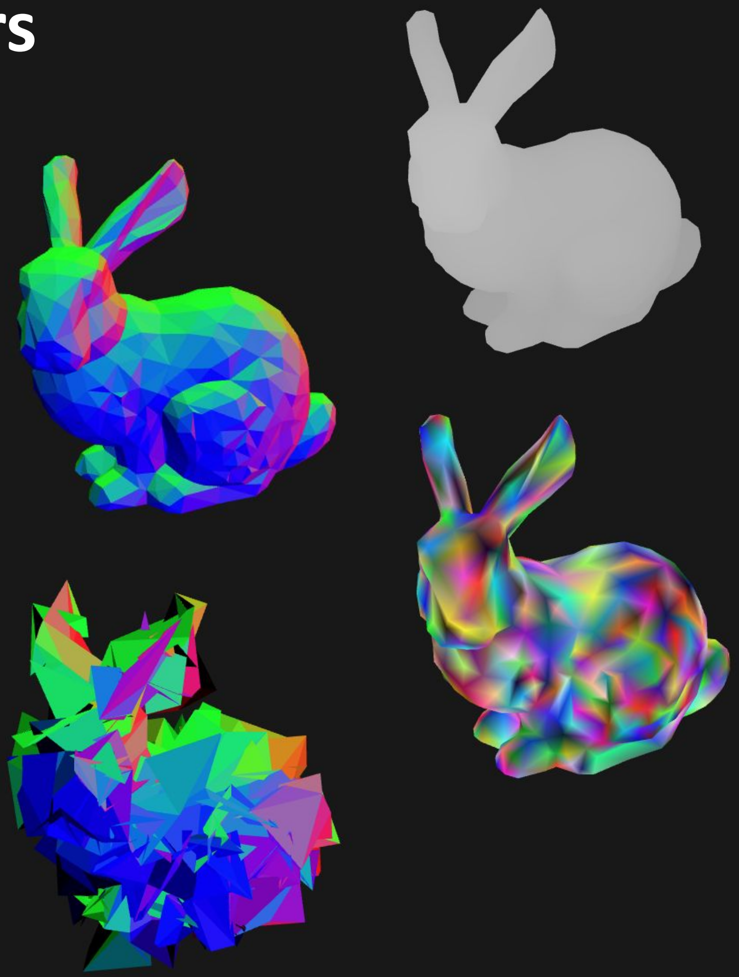
Look for TODO comments in the `src/shaders` folder.

Render the bunny:

1. Using z coordinate as vertex color
2. Using vertex normal as vertex color
3. Using random value as vertex color
4. Adding random noise to the vertex position

…

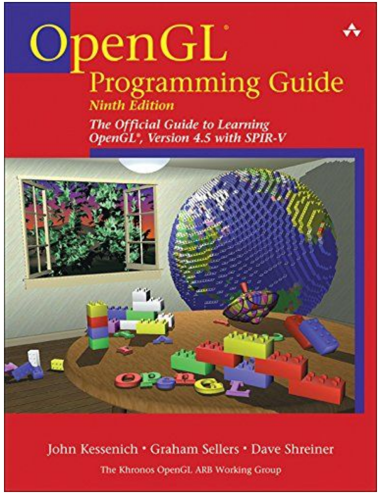
Be creative ;-)



Summary

- We covered:
  - Issues with Bresenham and scan line algorithms and an alternative drawing approach that using point-in-triangle assertion
  - Aliasing and antialiasing sampling issue in drawing
  - The modern rasterization rendering pipeline and its components
  - GLSL as a programming language for writing shader programs that execute on a GPU

Check the canonical OpenGL book for the more details on a "historical" graphics standard:



To learn more about the history of shading language, check out these research papers:

- Robert L. Cook. 1984. Shade trees. SIGGRAPH Comput. Graph. 18, 3 (July 1984), 223–231.
- Pat Hanrahan and Jim Lawson. 1990. A language for shading and lighting calculations. SIGGRAPH Comput. Graph. 24, 4 (Aug. 1990), 289–298.

How to create pictures like this?

Source: Cyberpunk 2077

Next

Texture