Getting Started with OpenGL: OpenGL Shading Language (GLSL)

Computer Graphics
Instructor: Sungkil Lee

Today

- Shaders and shading languages
- Preview to OpenGL Shaders
- Introduction to GLSL
- More on GLSL with Hello Example
- Difference of OpenGL ES SL from GLSL
- How to Debug GLSL Program

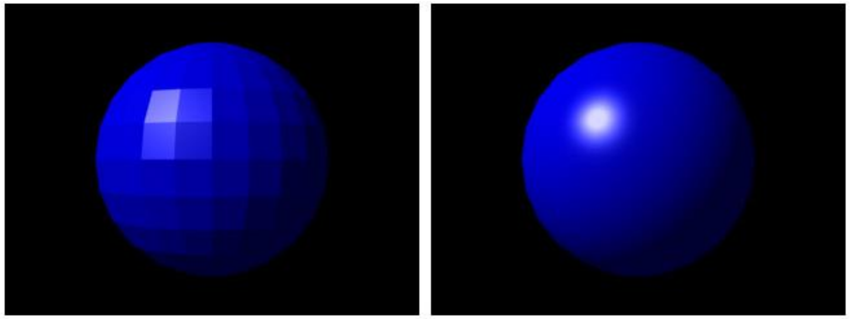
Prerequisites

- **https://thebookofshaders.com/**
  - The concept of shaders and their programming are explained in detail.
  - You can even find Korean translation as well as English version.
    - https://thebookofshaders.com/00/?lan=kr

Shaders and Shading Languages

Shaders?

- **Original definition**
  - A computer program used for shading (calculation of proper light-surface interaction within an image)



FLAT SHADING PHONG SHADING

- **Recent (still evolving) definition**
  - A user-defined GPU program that performs a unit-specific actions
  - Originally limited to pixel/fragment shaders, but extended to vertex traits (position, texture coordinates, normals, and ...).
  - Typical combination is the pair of vertex and fragment shaders

Shading Languages

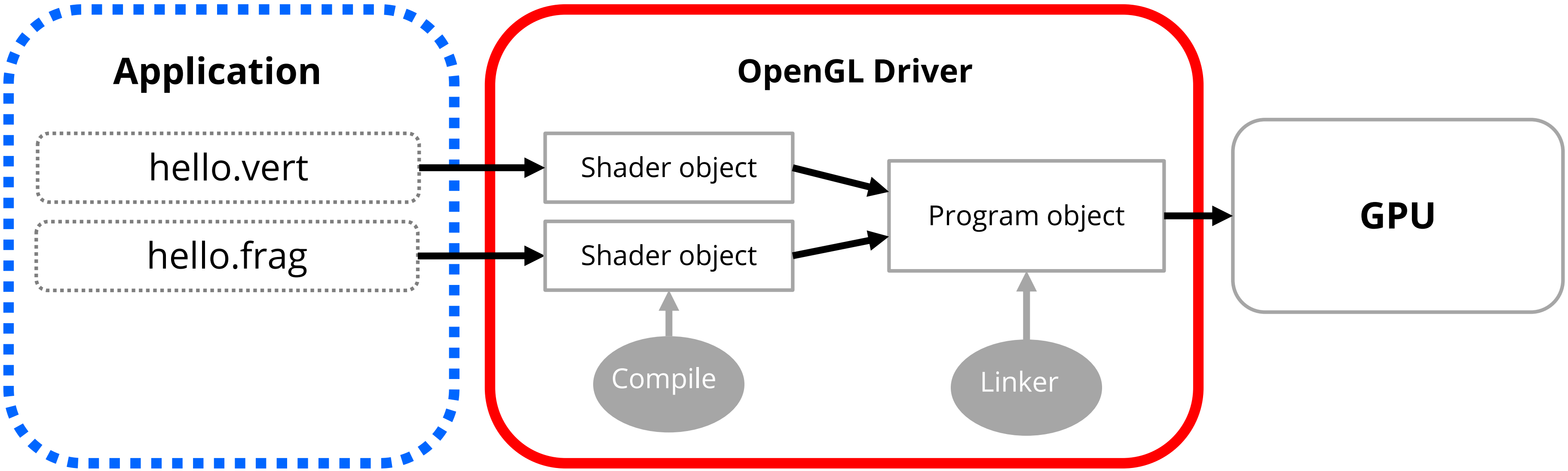
- **Shaders use a scripting language, which means:**
  - The shader code is compiled, linked, and launched with GPU at run-time.
  - Unlike other scripting languages (e.g., Python), it's not interpreted, but compiled.

- **API-specific shaders:**
  - OpenGL uses GLSL.
    - Part of OpenGL 2.0 and higher
    - As of OpenGL 3.1, application must provide shaders.
  - Direct3D uses HLSL (High-Level Shading Language).

Shader-based OpenGL

- **Shader-based OpenGL**
  - is based less on a state machine model than a ***data flow model***.
  - API's job is just for application to get data to GPU
  - Major actions happen in ***shaders*****.**
    - calculating the position/attributes of vertices, and
    - calculating the illumination of the pixels.
  - GPU does **SIMD** (single-instruction-multiple-data) computing:
    - GPU cores invocate many shaders in parallel.

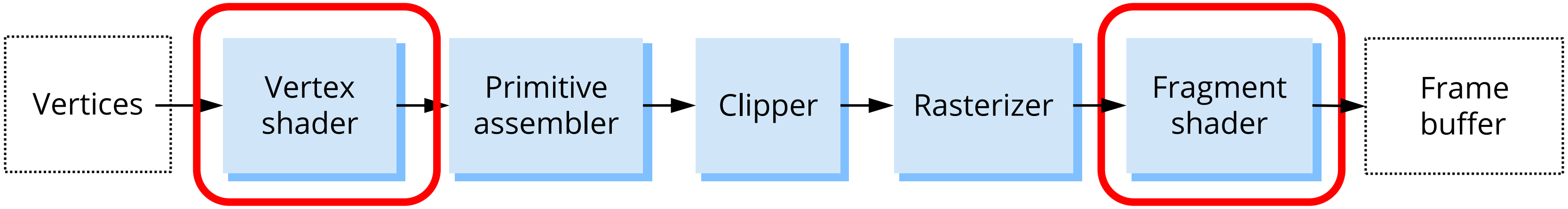
OpenGL Shader Execution Model



Provided by application developer using OpenGL API

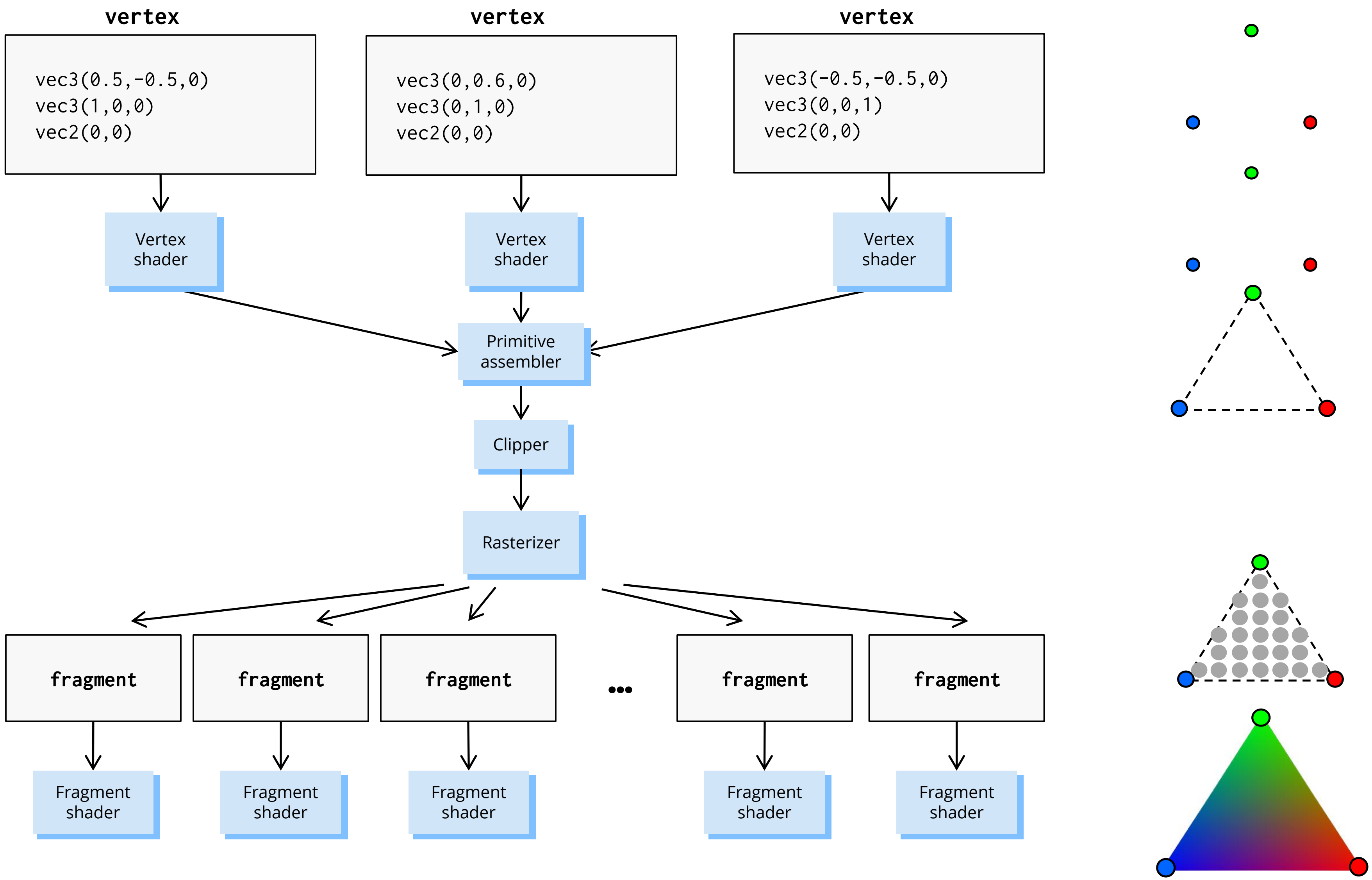
Provided by graphics hardware vender (NVIDIA, AMD, and Intel)

Vertex and Fragment Shaders



- **A vertex processor transforms a single input vertex at a time.**
  - But, this is also performed in parallel by multiple vertex processors.
  - Then, they are combined to primitives (here, to triangles).
  - Then, the rasterizer converts the primitives to pixels on the screen.
- **A fragment processor processes a single fragment at a time.**
  - This is also performed in parallel by multiple fragment processors.

Example Data Flow: Revisited



Preview to OpenGL Shaders

Preview of Shader Program

- **Vertex shader in GLSL**
  - A vertex shader outputs the position of a single input vertex.
  - It can also generate an additional output variable (e.g., vertex_color).
  - Outputs will be passed as inputs to the next shaders.

```
#version 330

layout(location=0) in vec3 position;   // vertex position
layout(location=1) in vec3 normal;     // vertex normal

out vec3 vertex_color;  // output of vertex shader

uniform float theta;    // rotation angle

void main()
{
   // builtin output variable that must be written
   gl_Position = vec4( position, 1 );
   ...
   // another output passed via input variable
   vertex_color = normal;
}
```

Preview of Shader Program

- **Fragment shader in GLSL**
  - A fragment shader outputs the color of the single input fragment.
  - `fragColor` is defined in [0,1], which maps later to [0,255] for 32 bpp color.

```
#version 330

in vec3 vertex_color; // the second input from vertex shader
out vec4 fragColor;   // define output variable to be shown in the display

// uniform variables will be globally shared among all the fragments
uniform bool b_solid_color;
uniform vec4 solid_color;
uniform float theta; // shared with theta in the vertex shader

void main()
{
   fragColor = b_solid_color ? solid_color :vec4(vertex_color,1);
   fragColor *= abs(sin(theta*4.0)); // modulate color by theta
}
```

Introduction to GLSL

OpenGL Shading Language (GLSL)

- **GLSL features**
  - All shaders have a single main() function
  - New data types: vectors, matrices, texture samplers
  - Overloaded operators and C++ like constructors
  - Preprocessors like C supported
  - If you know C, JUST DO IT.

- **But, there are missing C features**
  - No pointers or dynamic memory allocation
  - No call stack (meaning no recursion): all functions are inlined
  - No strings, char, double, short, long
  - No file I/O, console I/O (e.g., printf)

Version

- **GLSL requires to specify its version explicitly.**
  - Without version, version 1.0 is assumed (meaning obsolete)
  - The most basic/safest version is 3.3 for modern-style GLSL.
    - So, we write as follows:

```
#version 330
```

- **For advanced GLSL features, use the latest versions**
  - If you use an NVIDIA or AMD card and up-to-date driver, use 4.6 or 4.5

```
#version 460
```

  - If you use an Intel card, and up-to-date driver, use 4.4.

```
#version 440
```

Data Types

- **Basic data types**
  - `void`, `float`, `int`, `uint`, `bool`
- **Vector data types**
  - `vec2`, `vec3`, `vec4`: `float` vector types (aggregating 2/3/4 floats)
  - `ivec2`, `ivec3`, `ivec4`: `int` vector types
  - `uvec2`, `uvec3`, `uvec3`: `uint` vector types
  - `bvec2`, `bvec3`, `bvec4`: `bool` vector types
- **Matrix data types**
  - `mat2`, `mat3`, `mat4`: 2x2, 3x3, 4x4 matrix
  - non-square matrix types
    - `e.g., mat2x3`: 2x3 matrix

Opaque Data Types

- **Sampler data types (for textures)**
  - `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`, ...
  - Opaque data types (as opposed to the other transparent data types), because their implementation is hidden to the programmer.

Variable Qualifiers

- **All global variables have qualifiers**
  - Though you can write variables without qualifiers, it should be avoided.

- **Three qualifier types:**
  - `in`: the input to the shader stage
    - Layout should be indicated for explicit binding
    - e.g., layout(location=0)
  - `out`: the user-defined output of the shader stage
  - `uniform`: common for all shader types
    - read-only (constant) global variables (cached well);

Variable Qualifier Examples

- **Vertex shader example**

```
layout(location=0) in vec3 position;   // vertex position
layout(location=1) in vec3 normal;     // vertex normal

out vec3 vertex_color;  // output of vertex shader

...
```

  - Note that the attributes we specify in the vertex buffer are connected to the **"in"** variables in the vertex shader.
  - "**layout(location=0)**" indicates position is bound the first attribute of the vertex buffer. We need this for explicit binding.
  - The output of vertex shader (here, vertex_color) needs to be the input to the fragment shader
  - The number of outputs are user-defined; but, do not use too many outputs in practice.

Variable Qualifier Examples

- **Vertex shader example**

```
uniform float theta;    // rotation angle

...
```

  - uniform variables are specified in the host program:
    - in our case, in update() or render()

Variable Qualifier Examples

- **Fragment shader example**

```
// the second input from vertex shader
in vec3 vertex_color;

// must define output variable to be shown in the display
out vec4 fragColor;

// Uniform variables will be globally shared among all the fragments
uniform bool b_solid_color;
uniform vec4 solid_color;
...
```

- Note that the output attributes we specify in the vertex buffer are connected to the in attributes in the vertex shader.
  - in VS: out vec3 vertex_color
  - in FS: in vec3 vertex_color
- uniform variables can be specified in a single shader, or all the shaders.
  - But, the values are all shared across all the shader stages.

Specific to OpenGL ESSL

- **Precision qualifiers**
  - Unless default precision is given, per-value precision or default precision needs to be provided.
  - Default precisions on vertex shaders are:

```
precion highp float;
precion highp int;
precion highp sampler2D;
precion highp samplerCube;
```

  - Default precisions on fragment shaders are:
    - The default precisions on float applies also to vector types.
    - Note that there is no default precision on float; you need to define it.

```
precion mediump int;
precion lowp sampler2D;
precion lowp samplerCube;
```

Specific to OpenGL ESSL

- **Precision Qualifiers: highp support might be missing.**
  - highp support can be detected by testing the macro

```
#define GL_FRAGMENT_PRECISION_HIGH 1
```

- **#version should accompany es**
  - e.g., #version 300 es
  - Current samples add the versions automatically, based on the versions of current OpenGL context.
  - So, you don't see the version in the shader code, but in general cases, you have to add the version explicitly on your own.

Built-in Functions

- **Math**
  - `radians`, `degrees`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
  - `pow`, `exp`, `exp2`, `log2`, `sqrt`, `inversesqrt`
  - `abs`, `sign`, `floor`, `ceil`, `fract`, `mod`, `min`, `max`, `clamp`
- **Interpolations**
  - `mix` (similar to `lerp`; linear interpolation), `step`, `smoothstep`
- **Geometric**
  - `length`, `distance`, `cross`, `dot`, `normalize`, `faceForward`, `reflect`
- **Vector relational**
  - `lessThan`, `lessThanEqual`, `greaterThan`, `greaterThanEqual`, `equal`, `notEqual`, `any`, `all`

More on GLSL Vector Types

- **Accessing components in four ways**
  - position/direction: .x, .y, .z, .w
  - color: .r, .g, .b, .a
  - texture coordinates: .s, .t, .p, .q
  - array indexing: [0], [1], [2], [3]

- **Constructors supported**

```
void main(){ vec3 v = vec3(1.0, 0.0, 0.0); }
```

- **Swizzle operators: select or rearrange components**

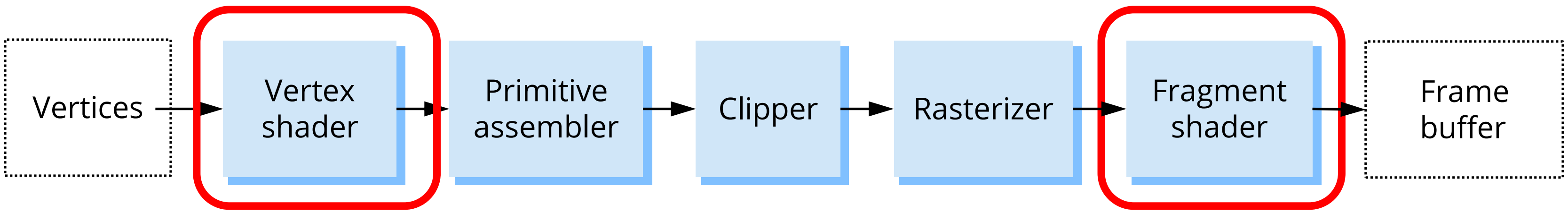
```
void main()
{
   vec4 v = vec4(1.0, 0.0, 0.0, 1.0)
   vec3 v3 = v.rgb;
   vec4 v4 = v.zzyx;
}
```

More on GLSL with Hello Example

Shader Programs

- **Create shader program**
  - "hello.vert" and "hello.frag" are the program sources of GPU programs.
  - Programs are specified in terms of **vertex** and **fragment** shaders.
  - Shader sources are complied/linked at ***run time*.**

```
// initializations and validations of GLSL program
program = cg_create_program( vert_shader_path, frag_shader_path );
```



Vertex Shader: hello.vert

- **A vertex shader outputs the position of the single vertex**
  - The fundamental role of VS is to decide `gl_Position`.
  - We also need to compute other outputs (here, vertex_color).

```
...

void main()
{
    // built-in output variable that must be written
    gl_Position = vec4( position, 1 );

    // rotate the vertices
    float c=cos(theta), s=sin(theta);
    mat2 m = mat2(c,s,-s,c); // column-major rotation matrix
    gl_Position.xy = m * position.xy; // swizzling for easy access

    // another output passed via output variable
    vertex_color = normal; // pass the color in norm to the vertex color output
}
```

Fragment Shader: hello.frag

- **A fragment shader outputs the color of the single input pixel.**
  - Pixel output (fragColor) is defined in [0,1], which maps later to [0,255] for 8-bit color depth.

```
#version 330

in vec3 vertex_color;    // the second input from vertex shader
out vec4 fragColor;      // define output variable to be shown in the display

// uniform variables will be globally shared among all the fragments
uniform bool b_solid_color;
uniform vec4 solid_color;
uniform float theta; // shared with theta in the vertex shader

void main()
{
    fragColor = b_solid_color ? solid_color : vec4(vertex_color,1);
    fragColor *= abs(sin(theta*4.0)); // modulate color by theta
}
```

Difference of OpenGL ES SL from GLSL

Precision Modifier

- **Precision modifier:**
  - We may use faster arithmetics for floating-point numbers by providing hints in vertex/fragment shaders .
  - This is quite useful for OpenGL ES for mobile platform.

```
precision mediump float;

...
```

  - Possible options and usage:
    - highp: vertex positions
    - mediump: normal vectors / texture coordinates
    - lowp: colors

Using OpenGL ES 2.0

- **OpenGL ES 2.0 is equivalent to old-style OpenGL**
  - For further development, do not use it any more.
  - But, for legacy applications, we may need to use it.
  - Here, I indicate required changes only for OpenGL ES 2.0

- **Main differences**
  - in/out qualifier not defined
  - precision modifier required
  - legacy texture functions only (e.g., texture() not supported)

Using OpenGL ES 2.0

- **varying**
  - in/out qualifiers had been introduced to include more shader stages (e.g., geometry shader, tessellation shader)
  - In particular, OpenGL ES 2.0 uses varying for VS output and FS input
    - VS output and FS input should be compatible.
    - We can easily adapt in/out qualifiers to ES 2.0 as follows.

- **Vertex shader**
  - in ➔ attribute
  - out ➔ varying
- **Fragment shader**
  - in ➔ varying
  - out vec4 fragColor ➔ no definition and write to (built-in) gl_FragColor

Using OpenGL ES 2.0

- **texture() → texture2D() or texture1D()**
  - Now, OpenGL ES 3.0 supports unified texture look-up function texture(), but OpenGL ES 2.0 does not.
  - We can easily replace texture() to texture2D(). That's it.

How to Debug GLSL Program

General Information

- **Debugging for GLSL program**
  - No usable debuggers available; there are some, but generally not
  - Console and file I/O are not supported.
  - States and host function calls can be examined, but shader functions not.

- **In general, we need to rely on ad-hoc strategies.**
  - Based on my experiences, there are some ways to debug shader programs, but as expected, not trivial.
  - Let me show some examples in the following pages.

Practical Advices: functions

- **functions: compile shader code as C++ code**
  - Write your shader code in C++, test it in C++, and port it.
  - Most of GLSL resemeble C++, and thus, the porting is not hard.
  - Macros would help to relieve this process.
  - Useful for writing a modular function, but not really for main shaders.
  - You can provide the data manually, and this is often infeasible.

Practical Advices: fragment shader

- **main(): interpret fragment color as a numerical value.**
  - Normalize variable values in [0,1], and check the color using color picker.
  - The following example should show gray (127/255) on the screen.

```
out vec4 fragColor;

// you have a uniform integer value that should be 64
uniform int uniform_to_test;

void main()
{
    // you have integer and vec4 values
    int local_value_to_test1 = 36;
    vec4 local_value_to_test2 = vec4( 1, 2, 3, 4 );

    // you have to normalize them in [0,1], considering their values
    // here, we normalize all to 0.5
    fragColor.x = uniform_to_test/128.0;
    fragColor.y = local_value_to_test1/72.0f;
    fragColor.zw = local_value_to_test2.xy/vec2(2.0,4.0);
}
```

Practical Advices: Reading Framebuffer

- **Often you need to batch-test for the whole framebuffer.**
  - Read the framebuffer and print the values
  - This is bad for performance, but useful for debugging.
  - use `glReadPixels()` after draw call functions

```
GLubyte* pixels = new GLubyte[w*h*4];
glReadPixels( 0, 0, w, h, GL_RGBA, GL_UNSIGNED_BYTE, pixels )

// examine values in pixels

delete[] pixel;
```

  - Make sure the temporary buffer is 4-byte aligned, when width is not the multiple of 4; you have add padding at each row of the buffer.
  - Also, make sure to remove the debugging code to avoid performance drop.

Practical Advices: Final Notes

- **Uniform variables are first tagerts to check.**
  - You may have typos on the uniform update function in C++.
  - You may not call the uniform update functions.
  - The types of the variables can be different.

- **Vertex shader is really hard to debug.**
  - You can debug the color as long as some should be drawn on the screen.
  - But, wrong vertex shader code is unlikely to place primitives on the screen.
  - Fortunately, the vertex shader almost has the same shape in typical cases.

- **If nothing works or it is not the case:**
  - Read carefully the code, and repeat checking individual variables.

References

OpenGL/GLES/GLSL References

- **The Book of Shaders**
  - https://thebookofshaders.com/
- **Official Wiki/Tutorials (very helpful)**
  - https://www.khronos.org/opengl/wiki/
  - https://open.gl/
- **Specification**
  - https://www.khronos.org/registry/OpenGL/specs/gl/
  - https://www.khronos.org/registry/OpenGL/specs/es/3.2/
- **Reference pages**
  - https://www.khronos.org/registry/OpenGL-Refpages/gl4/
  - https://www.khronos.org/registry/OpenGL-Refpages/es3/
- **Quick reference cards**
  - https://www.khronos.org/files/opengl46-quick-reference-card.pdf
  - https://www.khronos.org/files/opengles32-quick-reference-card.pdf

Any questions?