The Shaders’ View of the Basic Computer Graphics Pipeline

- In general, you want to have a vertex and fragment shader as a minimum.
- A missing stage is OK. The output from one stage becomes the input of the next stage that is there.
- The last stage before the fragment shader feeds its output variables into the rasterizer. The interpolated values then go to the fragment shaders.

- Fixed Function
- Programmable

Oregon State University
Computer Graphics
Vulkan Shader Stages

Shader stages

```c
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
```

Vulkan: GLSL Differences from OpenGL

Detecting that a GLSL Shader is being used with Vulkan/SPIR-V:

- In the compiler, there is an automatic
  `#define VULKAN 100`

Vertex and Instance indices:

- `gl_VertexIndex` and `gl_InstanceIndex`
  - Both are 0-based

`gl_FragColor`:

- In OpenGL, it broadcasts to all color attachments
- In Vulkan, it just broadcasts to color attachment location #0
- Best idea: don't use it – explicitly declare out variables to have specific location numbers
Shader combinations of separate texture data and samplers:
uniform sampler s;
uniform texture2D t;
vec4 rgba = texture( sampler2D( t, s ), vST );

Descriptor Sets:
layout( set=0, binding=0 ) . . . ;

Push Constants:
layout( push_constant ) . . . ;

Specialization Constants:
layout( constant_id = 3 ) const int N = 5;
• Can only use basic operators, declarations, and constructors
• Only for scalars, but a vector can be constructed from specialization constants

Specialization Constants for Compute Shaders:
layout( local_size_x_id = 8, local_size_y_id = 16 );

• gl_WorkGroupSize.z is still as it was

Vulkan: GLSL Differences from OpenGL

Vulkan: Shaders’ use of Layouts for Uniform Variables

All opaque (non-sampler) uniform variables must be in block buffers
### Vulkan Shader Compiling

- You pre-compile your shaders with an external compiler
- Your shaders get turned into an intermediate form known as SPIR-V
- SPIR-V gets turned into fully-compiled code at runtime
- SPIR-V spec has been public for a couple of years – new shader languages are surely being developed
- OpenGL and OpenCL will be moving to SPIR-V as well

<table>
<thead>
<tr>
<th>GLSL Source</th>
<th>External GLSL Compiler</th>
<th>SPIR-V</th>
<th>Compiler in driver</th>
<th>Vendor-specific code</th>
</tr>
</thead>
</table>

### Advantages:

1. Software vendors don’t need to ship their shader source
2. Syntax errors appear during the SPIR-V step, not during runtime
3. Software can launch faster because half of the compilation has already taken place
4. This guarantees a common front-end syntax
5. This allows for other language front-ends

---

### SPIR-V, from the Khronos Group

The first open standard intermediate language for parallel compute and graphics:

- SPIR (Standard Portable Intermediate Representation) was initially developed for use by OpenCL and SPIR versions 1.2 and 2.0 were based on LLVM. SPIR has now evolved into a true cross-API standard that is fully defined by Khronos with native support for shader and kernel features – called SPIR-V.
- SPIR-V is the first open standard, cross-API intermediate language for natively representing parallel compute and graphics and is incorporated as part of the core specification of both OpenCL 2.1 and OpenCL 2.2 and the new Vulkan graphics and compute API.
- SPIR-V exposes the machine model for OpenCL 1.2, 2.0, 2.1, 2.2 and Vulkan - including full flow control, and graphics and parallel constructs not supported in LLVM. SPIR-V also supports OpenCL C and OpenCL C++ kernel languages as well as the GLSL shader language for Vulkan.
- SPIR-V 1.1, launched in parallel with OpenCL 2.2, now supports all the kernel language features of OpenCL C++ in OpenCL 2.2, including initializer and finalizer function execution modes to support constructors and destructors. SPIR-V 1.1 also enhances the expressiveness of kernel programs by supporting named barriers, subgroup execution, and program scope pipes.
- SPIR-V is catalyzing a revolution in the language compiler ecosystem - it can split the compiler chain across multiple vendors’ products, enabling high-level language front-ends to emit programs in a standardized intermediate form to be ingested by Vulkan or OpenCL drivers. For hardware vendors, ingesting SPIR-V eliminate the need to build a high-level language source compiler into device drivers, significantly reducing driver complexity, and will enable a broad range of language and framework front-ends to run on diverse hardware architectures.
- For developers, using SPIR-V means that kernel source code no longer has to be directly exposed, kernel load times can be accelerated and developers can choose the use of a common language front-end, improving kernel reliability and portability across multiple hardware implementations.

[https://www.khronos.org/spir](https://www.khronos.org/spir)
SPIR-V:
Standard Portable Intermediate Representation for Vulkan


Shaderfile extensions:
.vert  Vertex
.tesc Tessellation Control
.tese Tessellation Evaluation
.geom  Geometry
.frag Fragment
.comp  Compute

(Can be overridden by the –S option)
-V  Compile for Vulkan
-G  Compile for OpenGL
-I  Directory(ies) to look in for #includes
-S  Specify stage rather than get it from shaderfile extension
-c  Print out the maximum sizes of various properties

Windows:  glslangValidator.exe
Linux:   setenv LD_LIBRARY_PATH /usr/local/common/gcc-6.3.0/lib64/

You Can Run the SPIR-V Compiler on Windows from a Bash Shell

1. Click on the Microsoft Start icon
2. Type word bash
You Can Run the SPIR-V Compiler on Windows from a Bash Shell

Pick one:

- Can get to your personal folders
- Does not have make

- Cannot get to your personal folders
- Does have make

Running glslangValidator.exe

```
ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 185

`glslangValidator.exe -V sample-vert.vert -o sample-vert.spv` Sample.vert.vert

ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 186

`glslangValidator.exe -V sample-frag.frag -o sample-frag.spv` Sample.frag.frag

ONID=mjb8pooh MINGW64 /y/Vulkan/Sample2017
$ 187
```
You can also run SPIR-V from a Linux Shell

$ glslangValidator.exe -V sample-vert.vert -o sample-vert.spv

$ glslangValidator.exe -V sample-frag.frag -o sample-frag.spv

Compile for Vulkan ("-G" is compile for OpenGL)

Specify the output file

The input file. The compiler determines the shader type by the file extension:
- .vert  Vertex shader
- .tccs  Tessellation Control Shader
- .tecs  Tessellation Evaluation Shader
- .geom  Geometry shader
- .frag  Fragment shader
- .comp  Compute shader
How do you know if SPIR-V compiled successfully?

Same as C/C++ -- the compiler gives you no nasty messages.

Also, if you care, legal .spv files have a magic number of \textbf{0x07230203}.

So, if you do an \texttt{od -x} on the .spv file, the magic number looks like this:

\begin{verbatim}
0203 0723 . . .
\end{verbatim}

---

Reading a SPIR-V File into a Vulkan Shader Module

```cpp
VkResult Init12SpirvShader(std::string filename, VkShaderModule * pShaderModule)
{
    FILE *fp;
    (void) fopen_s( &fp, filename.c_str(), "rb");
    if( fp == NULL )
    {
        fprintf( FpDebug, "Cannot open shader file '%s'
", filename.c_str( ) );
        return VK_SHOULD_EXIT;
    }
    uint32_t magic;
    fread( &magic, 4, 1, fp );
    if( magic != SPIRV_MAGIC )
    {
        fprintf( FpDebug, "Magic number for spir-v file '%s' is 0x%08x -- should be 0x%08x
", filename.c_str( ), magic, SPIRV_MAGIC );
        return VK_SHOULD_EXIT;
    }
    fseek( fp, 0L, SEEK_END );
    int size = ftell( fp );
    rewind( fp );
    unsigned char *code = new unsigned char [size];
    fread( code, size, 1, fp );
    fclose( fp );
}```
Reading a SPIR-V File into a Shader Module

```cpp
VkShaderModuleCreateInfo vsmci;
vsmci.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
vsmci.pNext = nullptr;
vsmci.flags = 0;
vsmci.codeSize = size;
vsmci.pCode = (uint32_t *)code;

VkResult result = vkCreateShaderModule( LogicalDevice, &vsmci, PALLOCATOR, pShaderModule );
fprintf( FpDebug, "Shader Module '%s' successfully loaded
", filename.c_str() );
delete[] code;
return result;
```

Vulkan: Creating a Pipeline

![Diagram](image.png)
You can also take a look at SPIR-V Assembly

```bash
glslangValidator.exe -V -H sample-vert.vert -o sample-vert.spv
```

This prints out the SPIR-V “assembly” to standard output. Other than nerd interest, there is no graphics-programming reason to look at this. 😊

---

For example, if this is your Shader Source

```glsl
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable
layout( std140, set = 0, binding = 0 ) uniform matBuf {
    mat4 uModelMatrix;
    mat4 uViewMatrix;
    mat4 uProjectionMatrix;
    mat3 uNormalMatrix;
} Matrices;

// non-opaque must be in a uniform block:
layout( std140, set = 1, binding = 0 ) uniform lightBuf {
    vec4 uLightPos;
} Light;

layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;

layout ( location = 0 ) out vec3 vNormal;
layout ( location = 1 ) out vec3 vColor;
layout ( location = 2 ) out vec2 vTexCoord;

void
main( )
{
    mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;
    gl_Position = PVM * vec4( aVertex, 1.);
    vNormal = Matrices.uNormalMatrix * aNormal;
    vColor = aColor;
    vTexCoord = aTexCoord;
}
```
This is the SPIR-V Assembly, Part I

Capability Shader

1. ExtensibleType "GLSL.std.450"
MemoryModel Logical GLSL450
EntryPoint Vertex 4 "main" 34 37 48 53 56 57 61 63
Source GLSL 400
SourceExtension "GL_ARB_separate_shader_objects"
SourceExtension "GL_ARB_shading_language_420pack"
Name 4 "main"
Name 10 "P"null;
Name 13 "matBuf"
MemberName 13(matBuf) 0 "AWorldMatrix"
MemberName 13(matBuf) 1 "AViewMatrix"
MemberName 13(matBuf) 2 "AProjectionMatrix"
MemberName 13(matBuf) 3 "ANormalMatrix"
Name 15 "Matrices"
Name 32 "gl_PerVertex"
MemberName 32(gl_PerVertex) 0 "gl_Position"
MemberName 32(gl_PerVertex) 1 "gl_PointSize"
MemberName 32(gl_PerVertex) 2 "gl_ClipDistance"
Name 34 ""null;
Name 37 "vAvertex"
Name 48 "vNormal"
Name 58 "vColor"
Name 57 "vColor"
Name 61 "vFaceCoord"
Name 63 "vFaceCoord"
Name 65 "lightBuf"
MemberName 65(lightBuf) 0 "LightPos"
Name 67 "Light"
MemberDecorate 13(matBuf) 0 CoArray
MemberDecorate 13(matBuf) 0 Offset 0
MemberDecorate 13(matBuf) 1 MatrixStride 16
MemberDecorate 13(matBuf) 1 Offset 64
MemberDecorate 13(matBuf) 1 MatrixStride 16
MemberDecorate 13(matBuf) 2 CoArray
MemberDecorate 13(matBuf) 2 Offset 128
MemberDecorate 13(matBuf) 2 MatrixStride 16
MemberDecorate 13(matBuf) 3 CoArray
MemberDecorate 13(matBuf) 3 Offset 192
MemberDecorate 13(matBuf) 3 MatrixStride 16
Decorate 13(matBuf) Block
Decorate 13(matBuf) DescriptorSet 0

2:             TypeVoid
3:             TypeFunction 2
6:             TypeFloat 32
7:             TypeVector 6(float) 4
8:             TypeMatrix 7(fvec4) 4
9:             TypePointer Function 8
11:             TypeVector 6(float) 3
12:             TypeMatrix 11(fvec3) 3
13(matBuf):             TypeStruct 8 8 8 12
14:             TypePointer Uniform 13(matBuf)
15(Matrices):     14(ptr) Variable Uniform
16:             TypeInt 32 1
17:     16(int) Constant 2
18:             TypePointer Uniform 8
21:     16(int) Constant 0
29:     16(int) Constant 1
30:     16(int) Constant 0
31:     16(int) Constant 1
32:     6(float) Constant 1
33:             TypePointer Output 7(fvec4)
34:             TypePointer Output 11(fvec3)
35:     16(int) Constant 3
36:    6(float) Constant 1
37:             TypePointer Output 11(fvec3)
38:             TypePointer Output 11(fvec3)
39:             TypePointer Output 11(fvec3)
40:             TypePointer Output 11(fvec3)
41:     16(int) Constant 3

This is the SPIR-V Assembly, Part II
This is the SPIR-V Assembly, Part III

This is SPIR-V assembly code. It's a translation of the SPIR-V assembly into a human-readable form. The code contains instructions for the execution of a computer program. Each instruction is a line of code that specifies a particular operation or data transfer. The code includes labels, functions, and various mathematical operations.

The code is part of a larger body of work that focuses on the printing of the configuration. It includes instructions for setting up and configuring the program, as well as the actual execution of the program.

The code is written in a language called SPIR-V, which is a standard for representing fragment shader programs. It is used in computer graphics applications to specify how pixels are rendered on a graphics card.

The code is part of the SPIR-V assembly, which is a subset of the SPIR assembly language. It is designed to be used with the GLSL (Graphics Language Shading Language) compiler, which is used to generate SPIR-V assembly from GLSL source code.

The code includes a series of instructions that set up the environment for the program, including the setup of variables, data structures, and functions. It also includes a series of mathematical operations, such as additions, subtractions, and multiplications.

The code is designed to be executed by a computer program, and it is used to produce a final output, such as a rendered image or animation.

The code includes a series of instructions that set up the environment for the program, including the setup of variables, data structures, and functions. It also includes a series of mathematical operations, such as additions, subtractions, and multiplications.

The code is designed to be executed by a computer program, and it is used to produce a final output, such as a rendered image or animation.

The code includes a series of instructions that set up the environment for the program, including the setup of variables, data structures, and functions. It also includes a series of mathematical operations, such as additions, subtractions, and multiplications.

The code is designed to be executed by a computer program, and it is used to produce a final output, such as a rendered image or animation.
SPIR-V: More Information

SPIR-V Tools:
http://github.com/KhronosGroup/SPIRV-Tools

1. Open Settings.
2. Click on Update & security.
3. Click on For Developers.
4. Under "Use developer features", select the Developer mode option to setup the environment to install Bash.
5. On the message box, click Yes to turn on developer mode.
6. After the necessary components install, you'll need to restart your computer.
7. Once your computer reboots, open Control Panel.
8. Click on Programs.
9. Click on Turn Windows features on or off.
10. Check the Windows Subsystem for Linux (beta) option.
11. Click OK.
12. Once the components installed on your computer, click the Restart now button to complete the task.

After your computer restarts, you will notice that Bash will not appear in the "Recently added" list of apps, this is because Bash isn't actually installed yet. Now that you have setup the necessary components, use the following steps to complete the installation of Bash.

1. Open Start, do a search for bash.exe, and press Enter.
2. On the command prompt, type y and press Enter to download and install Bash from the Windows Store.
3. Then you'll need to create a default UNIX user account. This account doesn't have to be the same as your Windows account. Enter the username in the required field and press Enter (you can't use the username "admin").
4. Close the "bash.exe" command prompt

Now that you completed the installation and setup, you can open the Bash tool from the Start menu like you would with any other app.