DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Shaders 3.0

Programming the GPU in modern OpenGL

Shaders

- What is it?

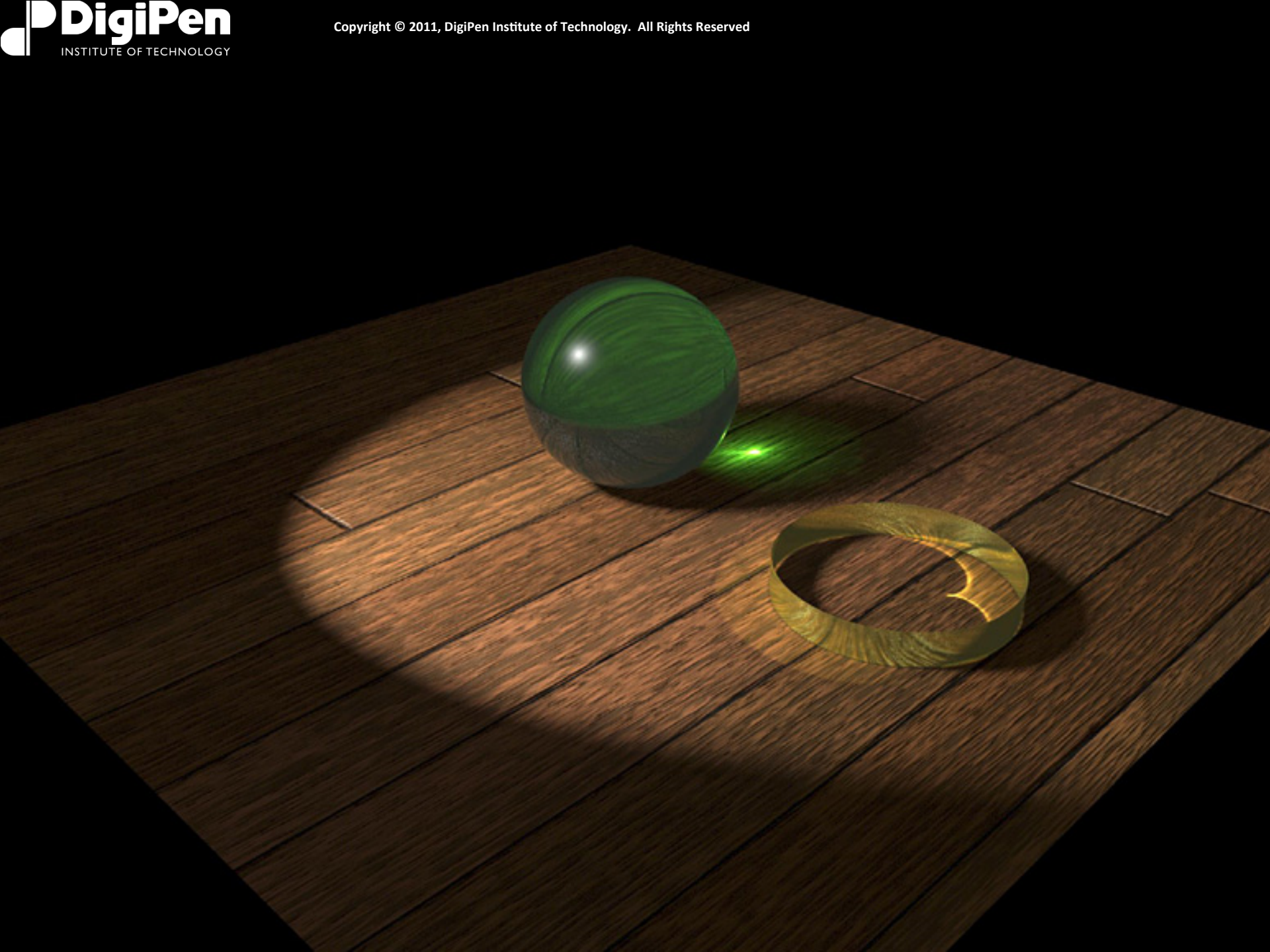
  Technically, a shader is something that is responsible to calculate the vertex/fragment color.

Shaders *(cont'd)*

- Why programmable shaders?
  - Support more popular rendering attribute calculations
  - Better optimization and parallelization of vertex/pixel operations
  - Allow flexibility for developing unique graphical effects
  - Eliminate redundant operations between application and geometry/rasterization engines

Shader Programming

- Why do we need shaders?
- What can shaders offer?
- Where do shaders fit in pipeline?



Shader Programming Features

- Computations independent to vertex or pixel
- Operations are optimized and highly parallelized
- A rich set of instructions are provided
- Fast memory equipped in graphics/video card

Shader Capabilities

- Procedural geometry
  - cloth simulation, soap bubbles, …
- Realistic light effects
  - physically-based light, area light, soft shadow, …
- Vertex blending
  - skinning, geometry morphing, …
- Keyframe interpolation
  - facial expression and speech synchronization, ...

Shader Capabilities *(cont'd)*

- Natural phenomena
  - water, snow, plants, …
- Particle system
  - fire, smoke, clouds, …
- Perspective view effect
  - fish eyeball lens, underwater, …
- Non-photorealistic materials
  - painterly effects, illustration techniques…
- Programmable antialiasing methods

Shaders in-context of Graphics Pipeline

T&L

Evaluators

Vertex Data

Vertex Shader

Tessellation Control Shader

Tessellation Evaluation Shader

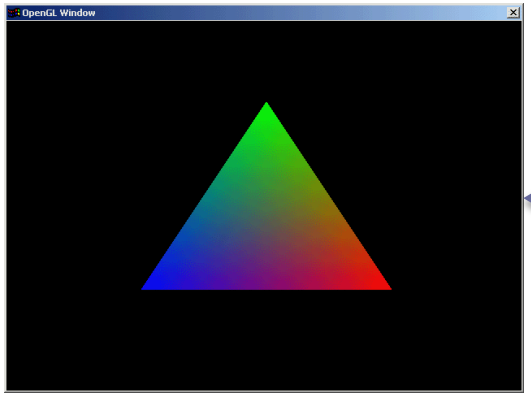
Geometry Shader

Primitive Setup

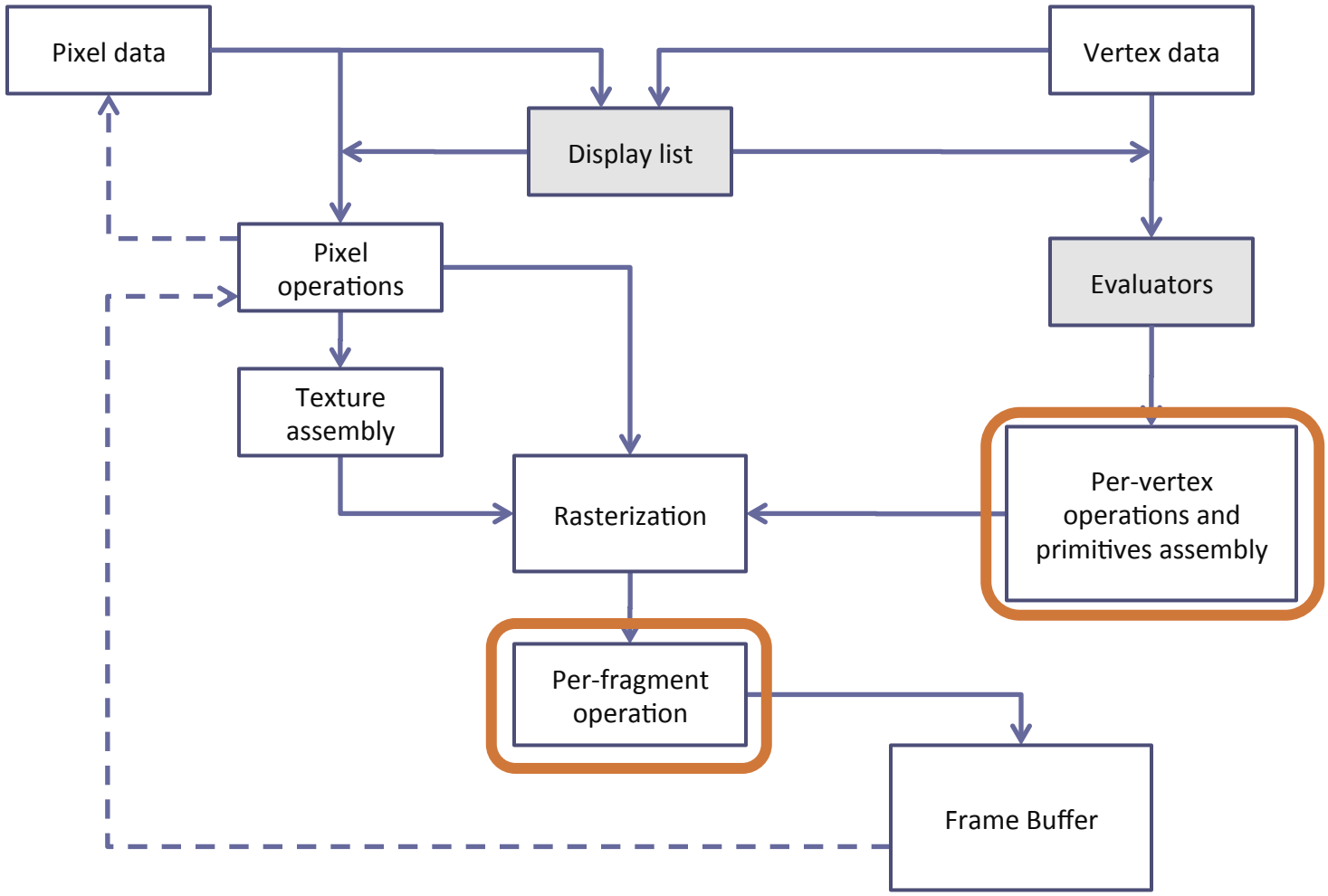
Clipping

Rasterization

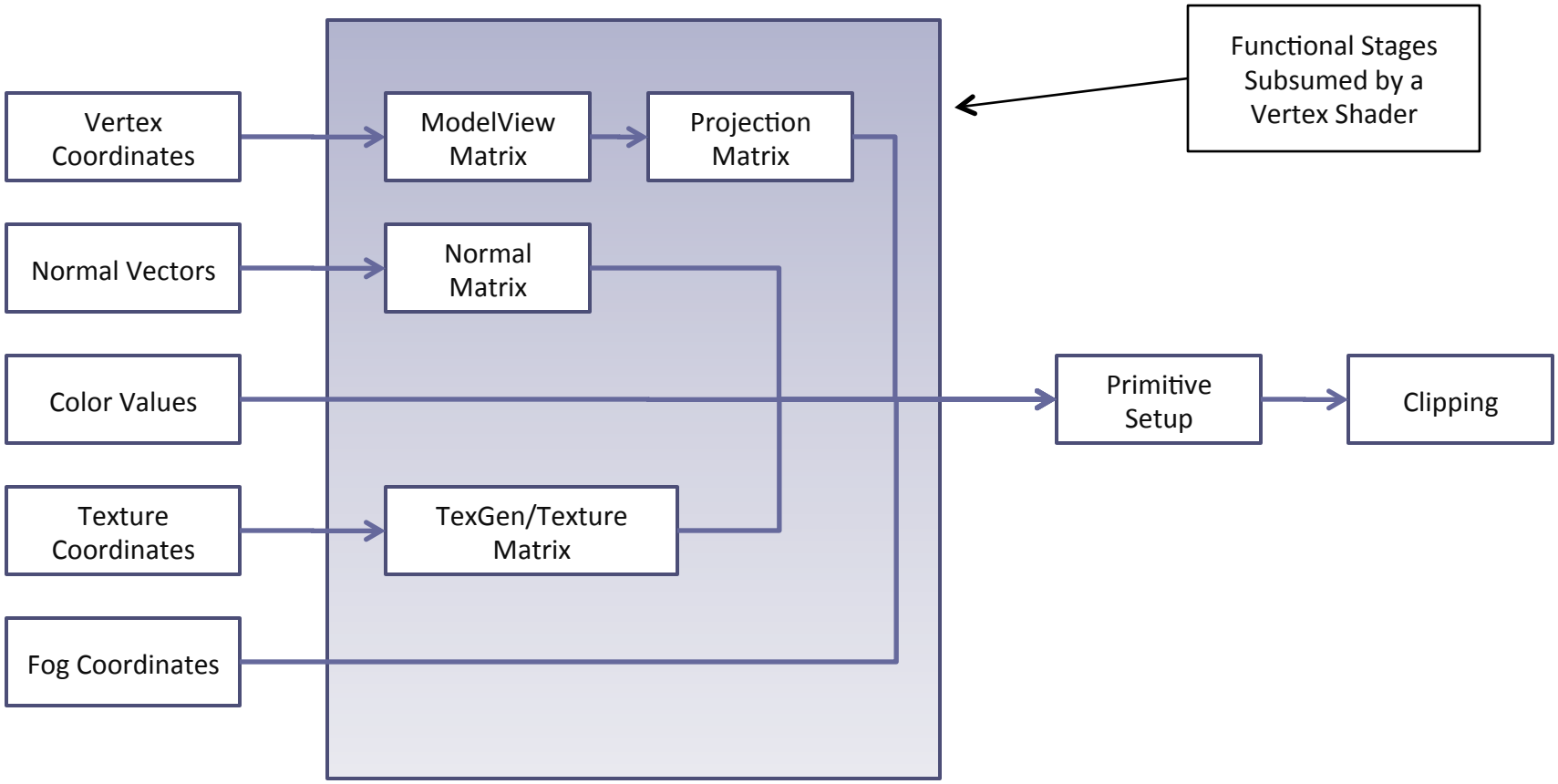
Fragment Shader



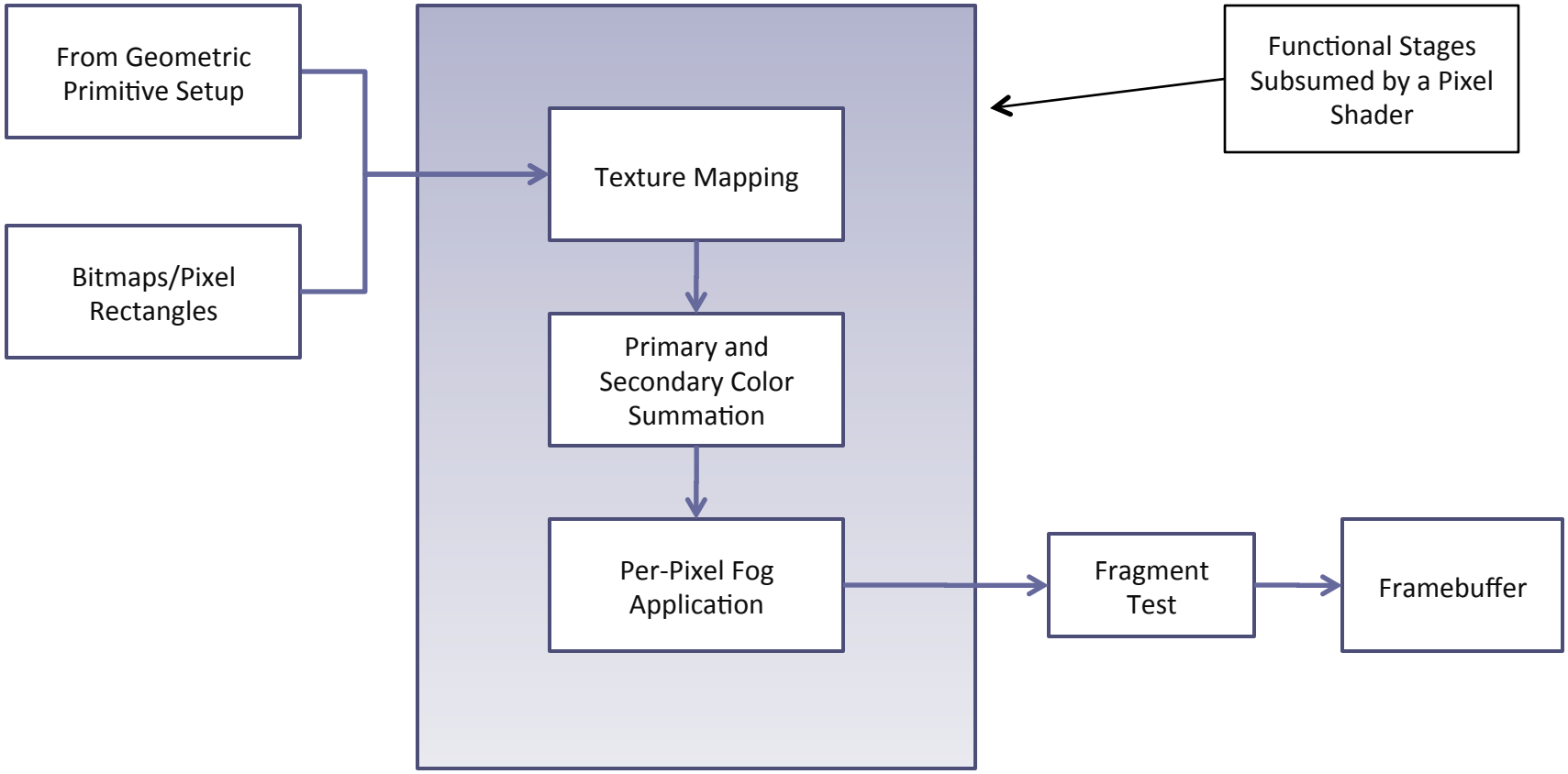
OpenGL Pipeline



Vertex Shader in the Pipeline (GLSL)



Pixel Shader in the Pipeline (GLSL)



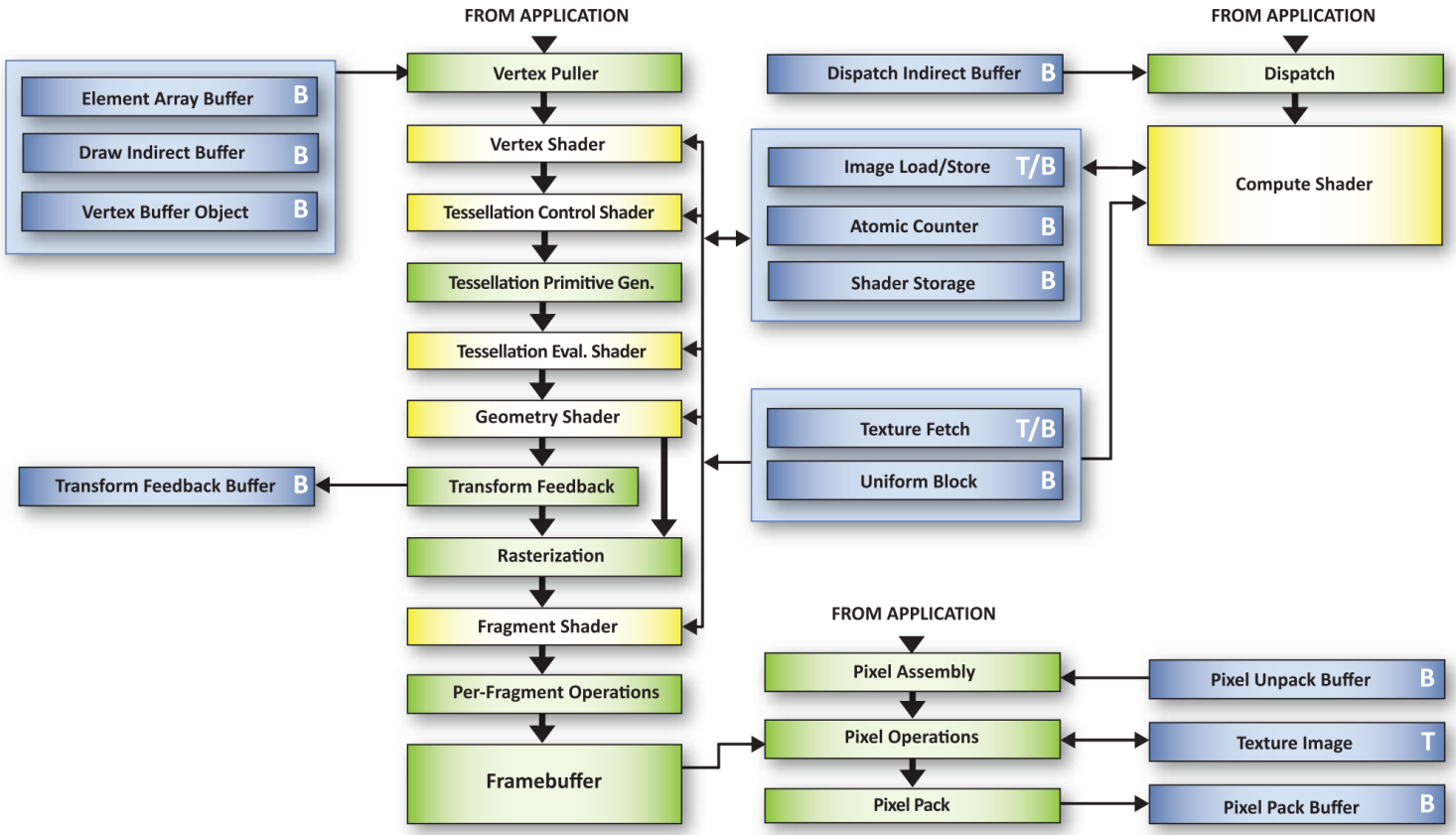
OpenGL Pipeline – "Official" view

OpenGL Pipeline

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline and indicate data flow.

- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding



OpenGL Shader Stages

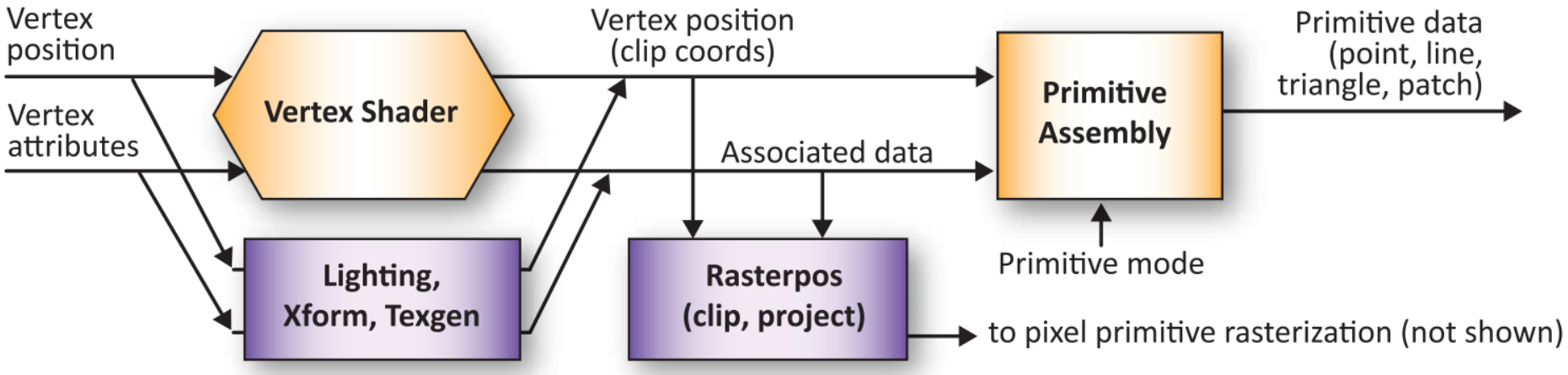
Shader Stage	Required?	Functionality
Vertex Shader (VS)	YES	Process **individual vertex data** that is passed through vertex buffers. Each vertex is processed **independently** of others.
Tessellation Shader (TS)	OPTIONAL	Generate additional geometry within the pipeline instead of specifying explicitly from the CPU. Receives **output of VS** and processes incoming vertices.
Geometry Shader (GS)	OPTIONAL	Can modify **entire geometric primitives** from within the pipeline. Operates on individual primitives (lines, triangles, points). Also allows conversion of primitives – triangles to lines – or discard geometry altogether.

OpenGL Shader Stages

Shader Stage	Required?	Functionality
Fragment Shader (FS)	YES	Process **individual fragments** of a primitive generated by OpenGL rasterizer. Calculate the fragment's color and depth values here and send it over for further processing.
Compute Shader (CS)	OPTIONAL	Non-graphics related shader stage that can be used to program SIMD instructions for a batch of input data.

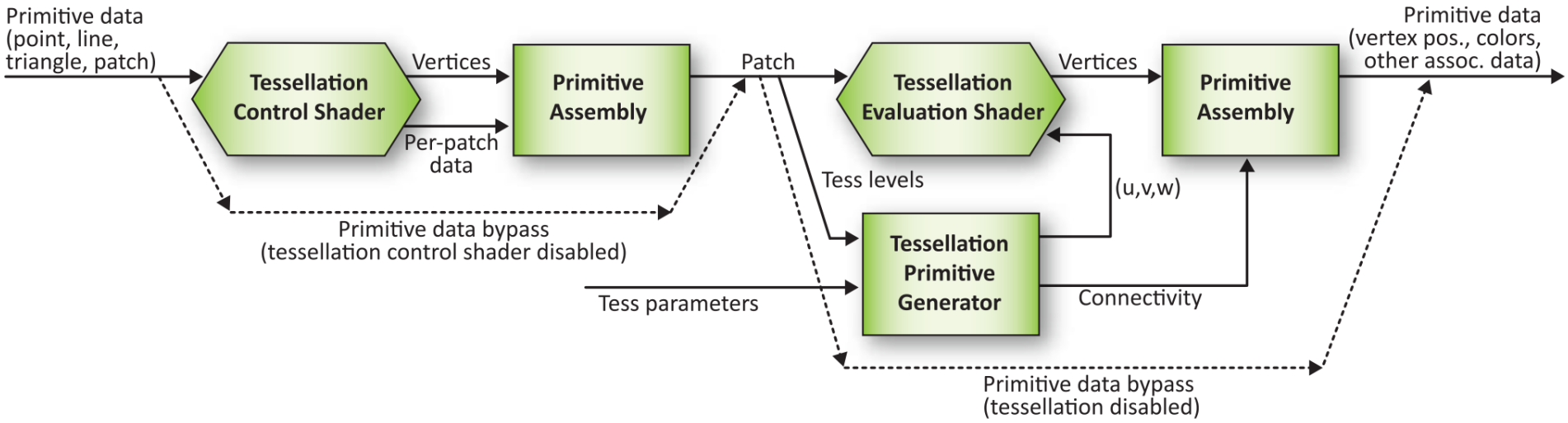
Geometry Path

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



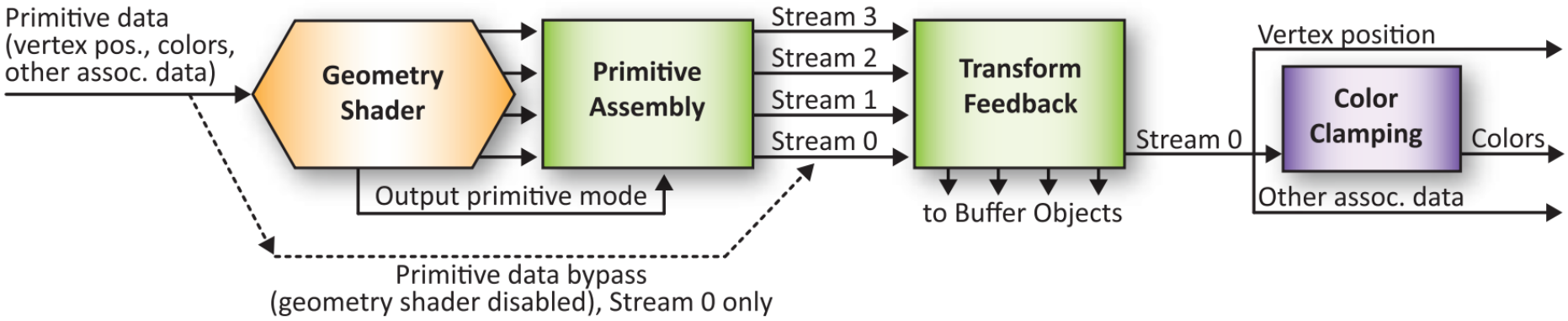
Tessellation Pipeline

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



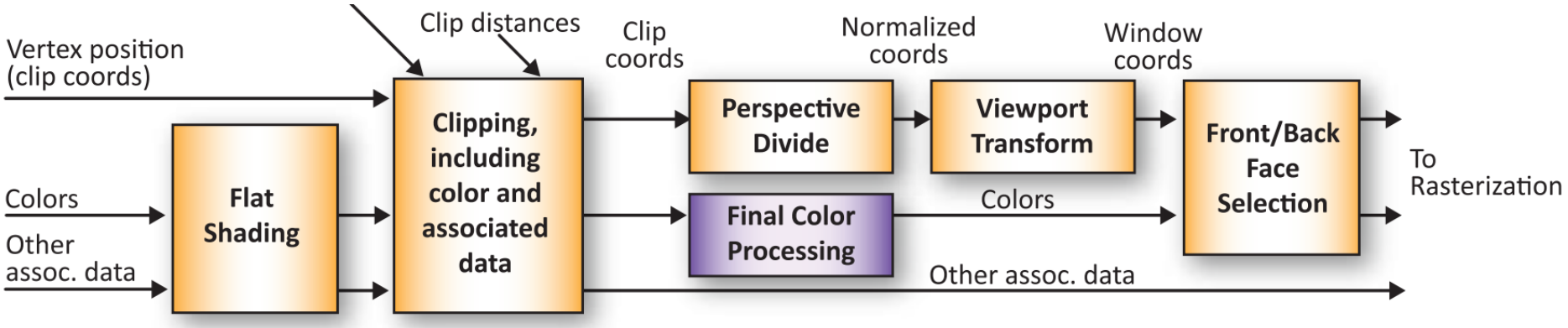
Geometry Shaders

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



Fragment Processing

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



High Level Shading Languages

- Allows to program shaders at an algorithm level
- Avoids hardware details and platform dependency
- Supports code re-usability and readability
- Maximizes efficiency and code optimization

Popular HLSLs

- GLSL: OpenGL Shading Language (OpenGL)
- ISL: Interactive Shading Language (SGI)
- HLSL: High-Level Shading Language (Microsoft)
- Cg: Computer Graphics (NVIDIA)
- RSL: RenderMan Shading Language (Pixar)

OpenGL Shading Language (GLSL)

- Approved by OpenGL ARB in version 2.0
- Implemented as a set of OpenGL extensions
- Adopted procedural language C/C++ syntax
- Built-in vector/matrix operation APIs
- **Featured with qualifiers to handle communications**

Interactive Shading Language (ISL)

- Defined, implemented and supported by SGI
- Based on C and assembly
- Translated into OpenGL rendering passes at run-time using a compiler as a part of driver
- Designed to provide backward hardware portability

High Level Shader Language (HLSL)

- Defined, implemented and supported by Microsoft
- Integrated with DirectX (version 9.0 and later)
- Translated to assembly at compilation time before execution

Computer Graphics (CG)

- Defined, implemented and supported by NVIDIA
- Provided with the Cg run-time library
- Translated to either OpenGL or DirectX calls

RenderMan Shading Language (RSL)

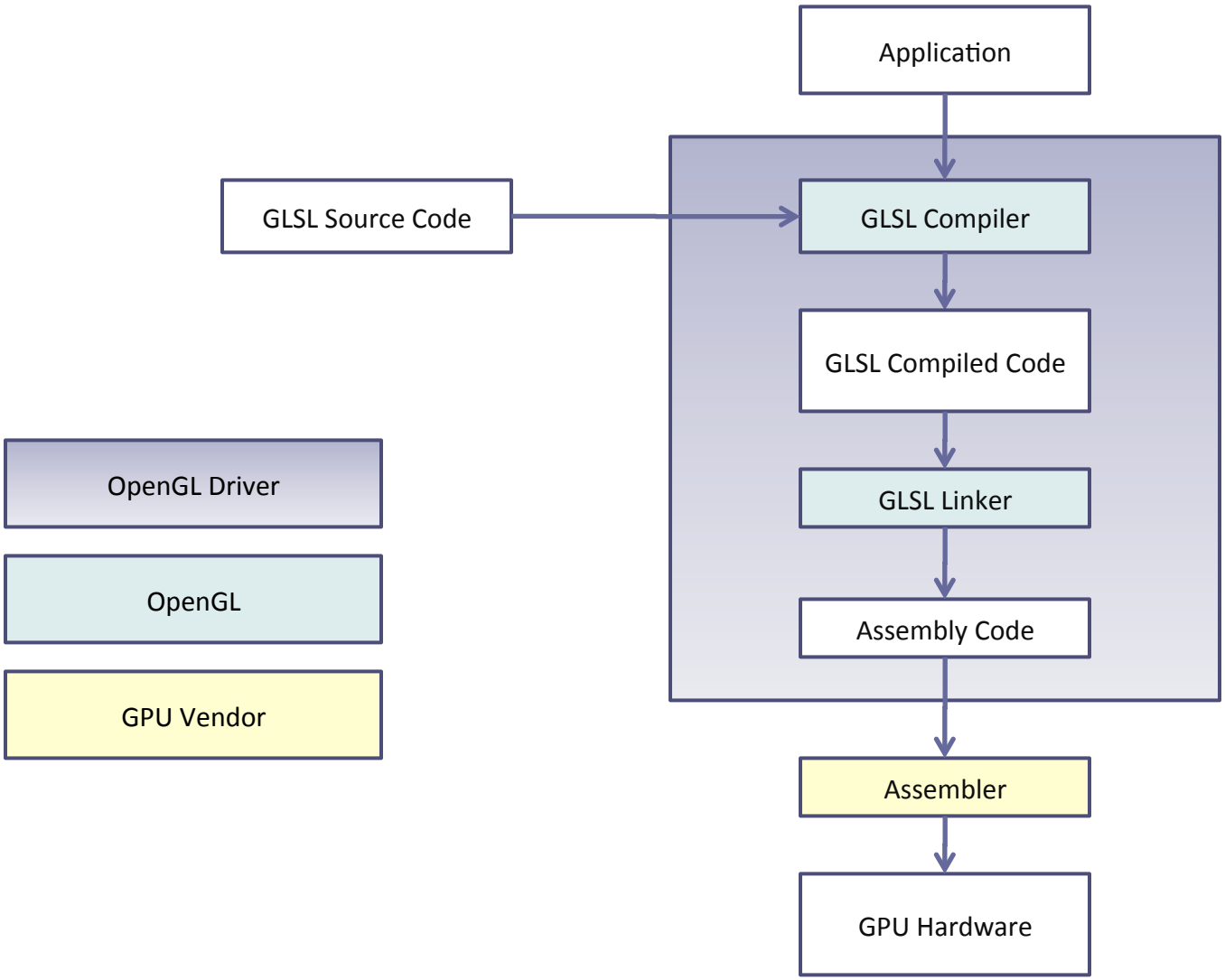
- Used for production quality rendering
- Been used successfully for movie production for over 25 years
  - 2013 was 25th anniversary year for Renderman
- Among the first shading language implemented

GLSL

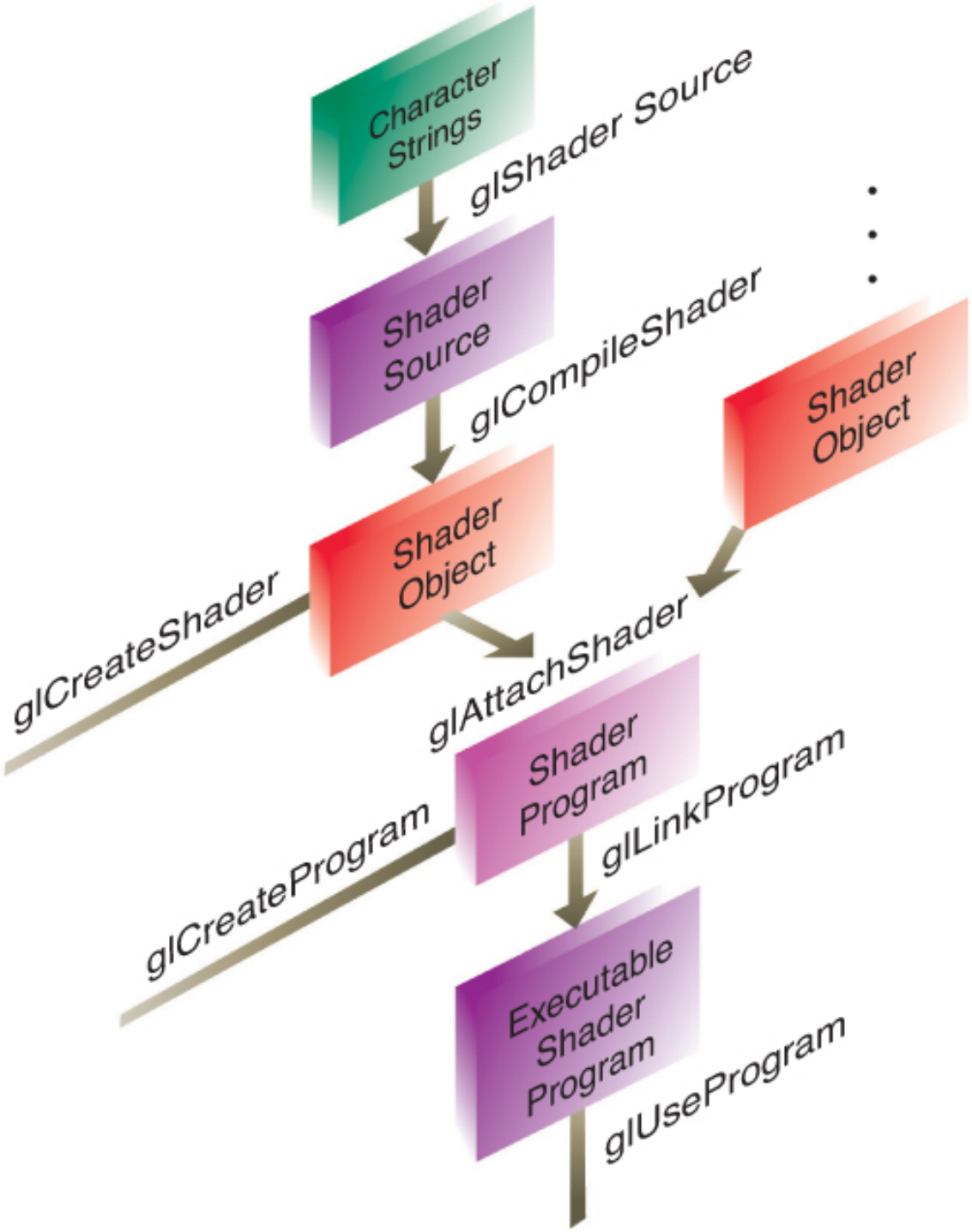
GLSL Production Pipeline

- Shaders are programmed individually and stored as external source files.
- Imported, compiled, linked and installed by application using OpenGL functions.
  - Similar to compilation, linking and execution of programs on CPU
- Supported by hardware vendors in the driver

GLSL Production Pipeline *(cont'd)*



OpenGL Shader processing commands



Benefits of GLSL Driver Model

- Tight integration
  - Relatively easy to include shaders into application
  - OpenGL API entry points accessed effortlessly.
  - Architecture and programming environment easily adapted.

- Runtime compilation
  - Portable and readable shader source code.
  - Easy to modify and maintain.
  - Efficient for compilation optimization.
  - No need for multitude of binary files.
  - No additional runtime libraries.
  - Runtime shader code generation and JIT optimizations.

Benefits of GLSL Driver Model *(cont'd)*

- Modular programming
  - Various algorithms are implemented in a collection of source files.
  - Run-time compilation, link and installation allow flexibility to select and combine different shaders to achieve a variety of visual effects.
- Cross-platform standard
  - Shader source files are independent to low-level assembly languages.
  - Supported by a group of hardware vendors for multi-platforms.

GLSL Shader Programming – An Example

```
#version 330  core

in vec4  vPosition;
in vec4  vColor;

out vec4 color;

uniform mat4  MVPMatrix;

void main(void)
{
  color = vColor;
  gl_Position = MVPMatrix * vPosition
}
```

Shader Programming in GLSL

How to write shaders ?

- Data types, interface, control flows, built-in functions, ...

How to put pieces together ?

- Shader object creation, compilation, link, installation.

Language Definition

- Entry point
- Data types
  - Basic types, vectors, matrices, arrays, structs and samplers
- Interface (qualifiers)
- Control flows
- Built-in functions

Entry point

- "main" function

#version 430 core

required as the first line of the program. Signals the OpenGL server to expect certain level of functionality at source and run-time level.

"void main(void)"

IMP: No command line arguments to shaders on GPU
How do we pass the data?
-- storage qualifiers (in, out, inout, uniform)

Basic Data Types

- Transparent types (follow IEEE standard for implementation)
  - void
  - float, vec2, vec3, vec4
  - int, ivec2, ivec3, ivec4
  - bool, bvec2, bvec3, bvec4
  - mat2, mat3, mat4
- Opaque types (we do not know how they are implemented)
  - samplers, images, atomic counters

Vectors

- No explicit way to tell if a vector is a color, a position or a coordinate.
- Accessible by <x, y, z, w>, <r, g, b, a> or <s, t, p, q>
- Also indexed as a zero-based one dimensional array.
- Example: If v is declared as a 3D vector, then v.y, v[1], v.g, and v.t are all indexing to the same data field.

Vector Swizzling

Structure members can be selected or rearranged by listing their component names after the “swizzle operator” (.)

```
float u1;
vec2 u2;
vec3 u3;
vec4 u4, v4;

u4 = v4.rgab;          // same as v4
u3 = v4.rgb;           // it becomes a 3D vector with <r, g, b>
u1 = v4.b;             // now it is a float
u2 = v4.xy;            // components names can be changed,
                       // as long as they are in the same set.
u4 = v4.xyba;          // illegal, components are not in the same set
u4 = v4.arbg;          // ok. Components can be rearranged.
u4 = v4.xxyy;          // they can even be repeated
```

Matrix

- mat2 – 2×2 matrix of floating point numbers,
- mat3 – 3×3 matrix of floating point numbers,
- mat4 – 4×4 matrix of floating point numbers.
- Arranged as an array of column vectors.
- Accessed through zero-based indices.
- Examples: If m is declared as a mat4 matrix, then:

```
m[2][3]    // 4th component of the 3rd column vector
m[1]       // 2nd column vector
m[4][1]    // undefined
```

Component-Wise Operations

- Can be applied to scalars, vectors and matrices.
- Behaves as if applied to each component independently
- Exceptions: multiplications
- Examples:

```
vec3 u, v = vec3(1.0, 2.0, 3.0);
mat2 m, n = mat2(1.0, 2.0, 3.0, 4.0);
float f = 5.0;
u = v + f;    // equivalent to u.x=v.x+f; u.y=v.y+f; u.z=v.z+f;
m = n + f;    // equivalent to adding f to each element of n;
```

Scalar / Vector / Matrix Multiplications

- Applied among scalars, vectors and matrices.
- Behaves as defined in standard linear algebra.
- Examples:

```
vec4  u, v, w;
mat4  m, n, k;
float f;

u = m * v;        // multiply m with v, resulting in a vec4;
n = f * m;        // multiply f with each component of m;
u = v * m;        // multiply v with m, results in a vec4;
k = m * n;        // multiply m and n, resulting in a mat4;
```

Arrays

- As defined in standard C/C++ with restrictions:
  - Only one dimensional
    - GLSL 4.3 – allows arrays of arrays
  - Any basic type and struct
  - No initialization at declaration.
  - Can be declared as unsized
  - Out-of-bounds indices not permitted
  - Get the size of the array with “.length()” method

Arrays (unsized)

- It must be declared again (only once) with a fixed size before referenced:

```
vec4 points[ ];  // unsized array of 4D vectors
vec4 points[10]; // declared again as an array of size 10
vec4 points[20]; // illegal, declared with a size again;
vec3 points[ ];  // illegal, declared with different type;
```

- All indices are statically referenced at the compile time:

```
vec4 points[ ];           // unsized array of 4D vectors
points[2] = vec4(1.0);    // an array of size 3
points[7] = vec4(2.0);    // an array of size 8
```

Arrays

- Can also be declared as follows

```
vec4[4] rectangle; // define a "type" of 4 float vectors
```

- No `typedef` in GLSL (C++ like behavior)

```
vec4[4] CreateRectangle( … )
{
    vec4[4] rect = …

    return rect;
}
```

Structures

Similar to the standard C/C++:

- Members can be any supported types.
- *struct* can be nested.
- Some restrictions: No qualifiers, No bit fields, No forward referencing, No in-place definitions, No anonymous struct.
- Example:

```
struct S { float f; };
struct T {
    S;              // Error: anonymous structures disallowed
    struct { ... };  // Error: embedded structures disallowed
    S s;            // Okay: nested structures with name are allowed
};
```

Samplers

- Opaque pointers to encapsulate texture look-up.
- Hardware implementation independent.
- Initialized by applications with texture maps.
- Information sent to shaders as uniform variables.
- Values can not be modified by shaders.

sampler1D	one-dimensional texture
sampler2D	two-dimensional texture
sampler3D	three-dimensional texture
samplerCube	cube-mapped texture
sampler1DShadow	1D depth texture with comparison
sampler2DShadow	2D depth texture with comparison

Storage Qualifiers

- Optional keyword put in front of the variable type to specify how that particular variable is linked.
- 4 types:
  - const
  - in
  - out
  - uniform
  - buffer
  - shared (compute shaders only)

Storage qualifiers

Type modifier	Description
const	Indicates that variable is read-only. Must be initialized when declared.
in	Indicates that the variable is an input to the shader stage. Input may be vertex attributes (for VS) or output from previous shader stage.
out	Indicates that the variable is an output of the current shader stage.
uniform	The variable is initialized by application before the shader is executed. Thereafter, the value remains constant across the primitive being processed. Is read-only in the shader.

Storage qualifiers (2)

Type modifier	Description
buffer	The recommended way to share a large buffer with the CPU application. Can be modified by the shader. Typically used in buffer blocks (coming up).
shared	Used only in compute shaders (CS). Indicates block of memory shared by threads in the local work group.

Specifying Uniform variables

- Two step process
  - Get the handle to the variable on the GPU
  - Set the value pointed by the handle

- GetUniformLocation(Gluint program_id, char *name)
- Uniform / UniformMatrix / UniformMatrixMxN
  - See GLSL specification for full list of functions

Code example

```
Glint           timeLocation;
Glfloat         timeValue;

…
// read current time into variable “timeValue”

// send this value to the shader
timeLocation = glGetUniformLocation( program, “time”);
glUniform1f( timeLocation, timeValue );
```

Functions

- C-like behavior
  - variable parameters to and from the function
  - no concept of “pointers” or “reference” on the GPU
  - Use access modifiers
    - in
    - const in
    - out
    - inout

Access Modifiers

Parameter modifier	Description
in	Value copied into the scope of the function (default access)
const in	read-only value copied into the function
out	Value copied out of the function. Undefined when first entering the function.
inout	Value copied into and out of the function

GLSL Operators

- Unary + − ++ -- ~ !
- Binary: + − * /
- Relational: < <= > >=
- Equality: == !=
- Logical: && || ^^
- Bitwise (reserved): & ^ | << >>
- Selection: ? :
- Assignment: = += −= *= /=
- Index: [ ]
- Grouping: ( )

See *OpenGL Shading Language Specification* [1] for the complete list of operator and their precedence.

GLSL Flow Control

- condition: *if, if-else*
  - Use sparingly!! This is #1 culprit for introducing bottlenecks in your shader code
- loop: *for, while, do-while*
- selection: *exp ? trueExp : falseExp*
- exit: *return, break, continue, discard* (fragment only)

Note*: discard* marks the fragment (gl_FragColor, gl_FragDepth) to be discarded rather than used to update the frame buffer.

GLSL Preprocessor Directives

```
#

#define
#undef

#if
#ifdef
#ifndef
#else
#elif
#endif
```

```
#error  message

#pragma optimize(on|off)
#pragma debug(on|off)

#line line file
__LINE__
__FILE__
__VERSION__

// comment
/* comment */
```

GLSL Interface Blocks

- Sending a batch of variables to the shader in the same command
- Uniform – **uniform** blocks
- input & output – **in** and **out** blocks
- shader storage – **buffer** blocks

Example

```
uniform myBlock
{
      vec4   v1;
      bool   v2;
} myVar;
```

- Use ‘myBlock’ as the block specifier for external (CPU) communication
  - `blockIndex = glGetUniformBlockIndex(p, ”myBlock");`
- Use ‘myVar’ as variable in shader
  - `myVar.v1 OR myVar.v2`

Uniform blocks

- Can use “glMapBuffer()” to access the memory block
- Only transparent types can be use in uniform blocks
  - implementation of opaque types is not standardized
- Layout control
  - programmer can specify how to lay out the individual elements of the block
  - use qualifiers to variables

Uniform layout qualifiers

Layout qualifier name	Description
shared	This uniform block is shared by multiple shader programs. (default layout)
packed	Layout favors minimal memory footprint
std140	Use standard layout for uniform blocks or shared storage buffer blocks*
std430	Use a standard layout for buffer blocks*
row_major	Store matrices in uniform block in row-major ordering
column_major	Store matrices in uniform block in column-major ordering (default ordering)

* See GLSL Specification for more details about default buffer layout

Shader Programming in GLSL

Definitions – how to write shaders ?

- Data types, interface, control flows, built-in functions,

Extensions – how to put pieces together ?

- Shader object creation, compilation, link, installation.

Examples – how to write rendering algorithm ?

- Multi-texture, lighting models, bump mapping, antialiasing, fractal, …

GLSL Production Pipeline

- Programmed separately and stored as external source files.
- Imported, compiled, linked and installed by application as a set of extension APIs.
- Supported by hardware vendors in a driver model.
- Implemented in a run-time loading library.

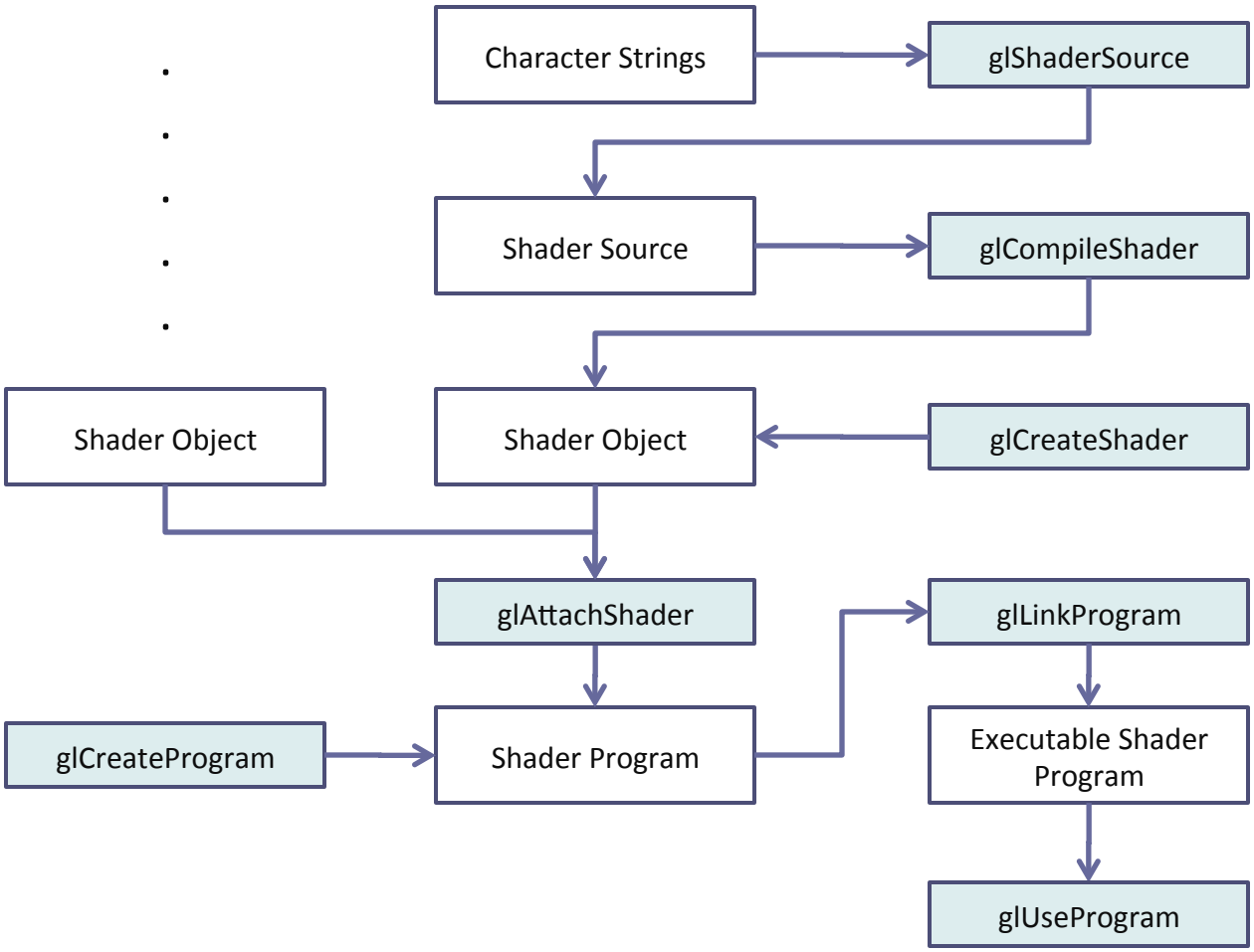
OpenGL Shader Extensions

1. Create vertex and fragment shader objects.
2. Load source code into the shader objects.
3. Compile each shader (and get compilation status).
4. Create a program object.
5. Attach compiled shaders to program object.
6. Link the program object (and get link status).
7. Install the program object.

8. Obtain uniform variables locations and assign values.
9. Associate attribute variables and assign values.

10. Delete shader objects when they are no longer in use.

GLSL Production Pipeline



Related Functions (Shader Init)

- GLuint glCreateShader (GLenum shaderType)
  - Create an empty shader object and return its id. “shaderType” determine the shader type (GL_VERTEX_SHADER or GL_FRAGMENT_SHADER)
- void glShaderSource (GLuint shader, GLsizei count, const GLchar** string, const GLint* length)
  - Associate the shader object with a shader source code. If “length” is 0, the string is assumed to be null terminated.
- void glCompileShader (GLuint shader)
  - Compile the shader source code. Use “glGetShaderiv” with GL_COMPILE_STATUS to check if the compile is successful.
- void glGetShaderiv(GLuint shader, GLenum pname, GLint* params)
  - Get parameter value of a given shader object (GL_COMPILE_STATUS, GL_GET_INFO_LENGTH, etc).
- void glGetShaderInfoLog (GLuint shader, GLsizei maxLength, GLsizei* length, GLchar* infoLog)
  - Returns the information log for the specified shader object.

Related Functions (Program Init)

- GLuint glCreateProgram (void)
  - Create an empty progam object and returns its id.
- void glAttachShader (GLuint program, GLuint shader)
  - Attach shader object(s) to a program.
- void glLinkProgram (GLuint program)
  - Link all attached shaders in the program. Use glGetProgramiv with GL_LINK_STATUS to get the link status.
- void glGetProgramiv (GLuint program, GLenum pname, GLint* params)
  - Returns the parameter values of a given program object.
- void glGetProgramInfoLog (GLuint program, GLsizei maxLength, GLsizei* length, GLchar* infoLog)
  - Returns the information log of a given program object.
- void glUseProgram (GLuint program)
  - Installs a program object as part of current rendering state

Related Functions (Runtime)

- GLint glGetUniformLocation (GLuint program, const GLchar *name)
  - Returns an integer that represents the location of a specific uniform variable within a program object. Returns “-1” if “name” is not found or start with “gl_”.
- void glGetUniformfv (GLuint program, GLint location, GLfloat*params)
  - Get the value(s) of the specified uniform variable.
- void glGetActiveUniform (GLuint program, GLuint index, GLsizei bufSize, GLsizei* length, GLint* size, GLenum* type, GLchar* name)
  - Returns information about an active uniform variable for the specified program object
- void glUniform*
  - modifies the value of a uniform variable or a uniform variable array.
  - Please refer to OpenGL manual for complete list of the function variants.

Related Functions (Runtime) *(cont'd)*

- GLint glGetAttribLocation (GLuint program, const GLchar* name)
  - Returns the location of attribute variable in the given program.
  - Returns "-1" if "name" is not found or start with "gl_".
- void glGetActiveAttrib (GLuint program, GLuint index, GLsizei bufSize, GLsizei* length, GLint* size, GLenum* type, GLchar* name)
  - Returns information about an active attribute variable for the specified program object
- void glVertexAttrib*
  - Pass in generic per-vertex attribute value to the vertex shader.
  - Please refer to OpenGL manual for complete list of the function variants.

Related Functions (Deallocation)

- void glDeleteShader (GLuint shader)
  - Deallocate the shader object memory and invalidates its name.
- void glDeleteProgram(GLuint program)
  - Deallocate the program object memory and invalidates its name.
- See `ShaderManager.h` and `ShaderProgram.h` in the supplied framework for details of implementation.

Advanced Shaders

Advanced Shaders (4.2+)

- Program pipeline object
  - Modular approach to building shaders
- One program == a subset of CG pipeline stages
- Link multiple programs into a “pipeline program”

+ Allows code reuse

- Strict requirements on interface matching (in/out variables must match from different program objects)

Interface Matching

- Output of one stage matches input of subsequent stage if
  - the names and types of the variables match
  - the qualifiers for the variables match
- If using blocks, the order of the variables has to match too!

Subroutine Shaders

- Using Program Pipeline Object can be useful, but very expensive
  - Switching between program objects is slow
- Alternative − use **subroutine uniforms**
- Sort of pointer-to-function

(Very Simple) Example

```
// declaration
subroutine vec4 subRoutine1( vec4 n );

subroutine (subRoutine1)
vec4 myFunction1( vec4 n )
{
      return normalized( n );
}
// subroutine “pointer”
subroutine uniform subRoutine1 Foo_func;

// usage
in vec4 n;
vec4 normal = Foo_func( n );
```

Program binaries

- Hints the OpenGL server to keep compiled and linked binaries of shaders in a “cache” on the CPU side
- Can be written to disk to be used later
  - No standard format among OpenGL providers

Wrangler Library (GLEW)

- An OpenGL C/C++ extension library
- Efficient run-time functions for loading, compiling, linking and installing shaders
- Available for a variety of operating systems
- Cross-platform and open source
- See http://glew.sourceforge.net/ to download the library and other information

References

- Sellers et al, "OpenGL SuperBible" 7th edition, Addison-Wesley, 2016
- Shreiner et al, "OpenGL Programming Guide" 8th edition (The Red Book), Addison-Wesley, 2014
- Rost, R., and Licea-Kane, Bill "OpenGL Shading Language"(The Orange Book)
- http://glew.sourceforge.net/