Introduction to Computer Graphics

Introduction to OpenGL

Prasanna Ghali

What is OpenGL API? (1/3)

- OpenGL = Open Graphics Library
- Specification of application programming interface of essential graphics functions
- Serves as software interface to GPU
- GPU driver will translate OpenGL API commands to machine language instructions that can be executed by GPU

What is OpenGL API? (2/3)

- Has evolved from OpenGL 1.0 (1992) to current version OpenGL/GLSL 4.6
- Bindings available for lots of popular programming languages: C/C++, Java, C#, Python, …
- opengl.org is best source about everything related to OpenGL

What is OpenGL? (3/3)

- Maintained by [Khronos Group]

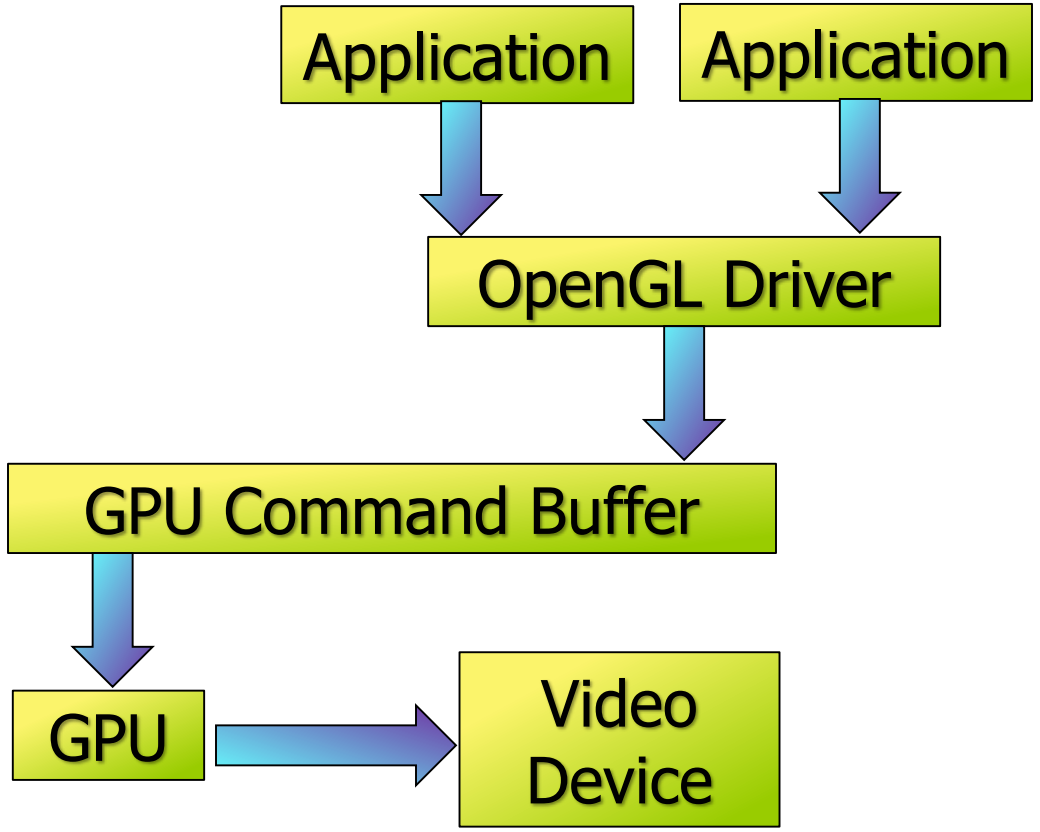
What Does Spec Describe? (1/2)

- Hundreds of commands specifying:
  - Shader programs or shaders
    - Shaders are small user-defined programs that control programmable parts of graphics pipe
  - Data used by shaders
  - State controlling aspects of OpenGL outside scope of shaders

What Does Spec Describe? (2/2)

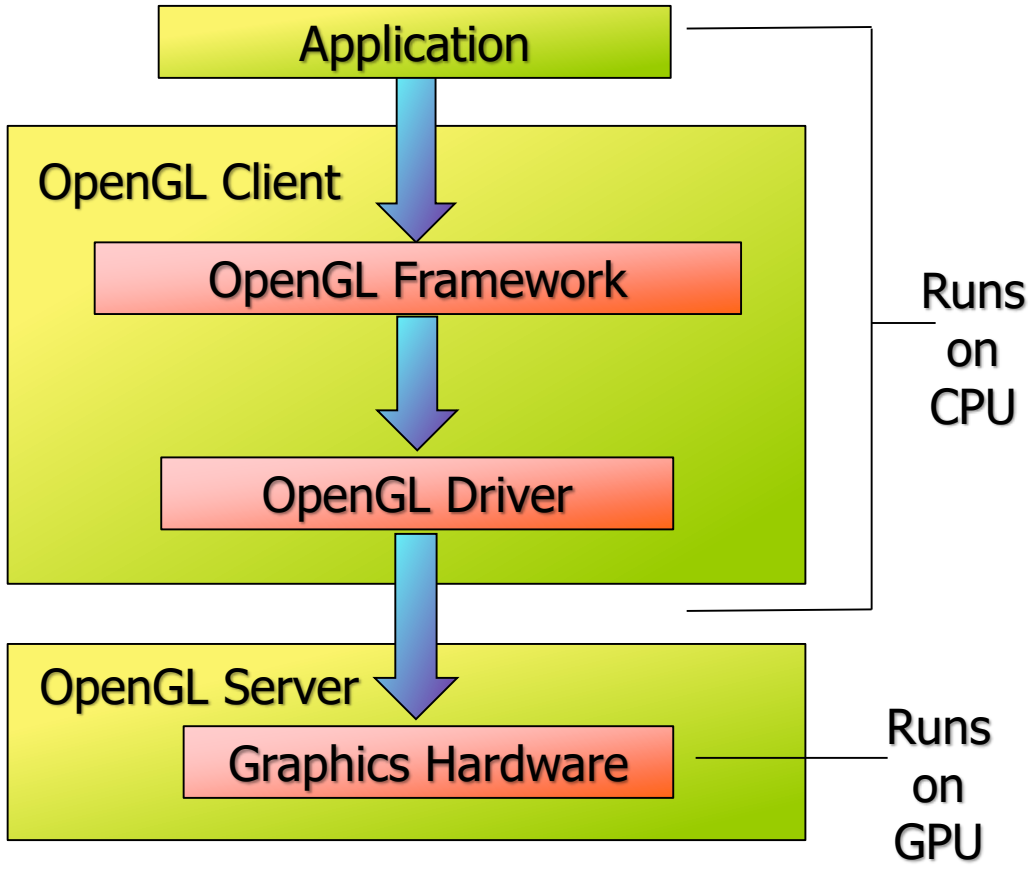
- Spec only says *what* should be done
- Not, *how* it should happen
- Unlike traditional libraries, OpenGL is not binary
- GPU manufacturer provides implementation of API

Spec Implemented by OpenGL Driver



Client-Server Model

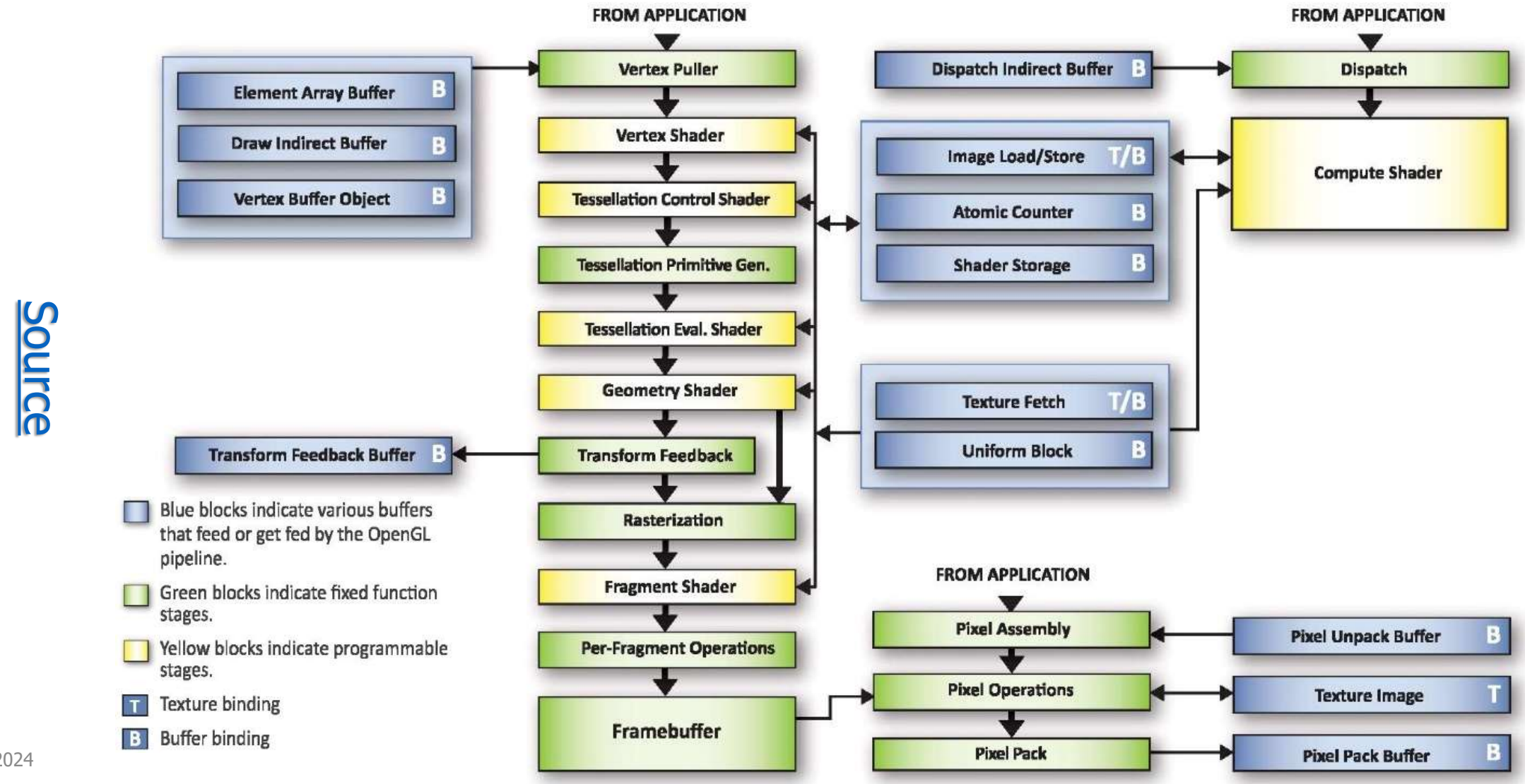
- Graphics applications running on CPU considered *client*
- OpenGL implementations provided by GPU considered *server*
- Client issues commands
- Server interprets and processes commands
- Network transparent - server may or may not operate on same computer or same address space as client



What Does Spec Not Describe?

- Independent of operating or windowing systems
  - No support for performing windowing tasks
  - No support for input and output devices
  - Only device supported is framebuffer [which must be provided by operating system]
- Applications must rely on facilities provided by OS

OpenGL 4.6 Rendering Pipeline



Source

GPU States

- GPU is *stateful*
  - Many stages
  - Large number of operations at each stage
  - Each operation has many options
  - Therefore, considerable amount of information must be set and maintained
  - This information referred to as *state settings*
- Cumulatively, state settings define and specify
  - Behavior of pipe, and
  - Transformation of primitives into pixels

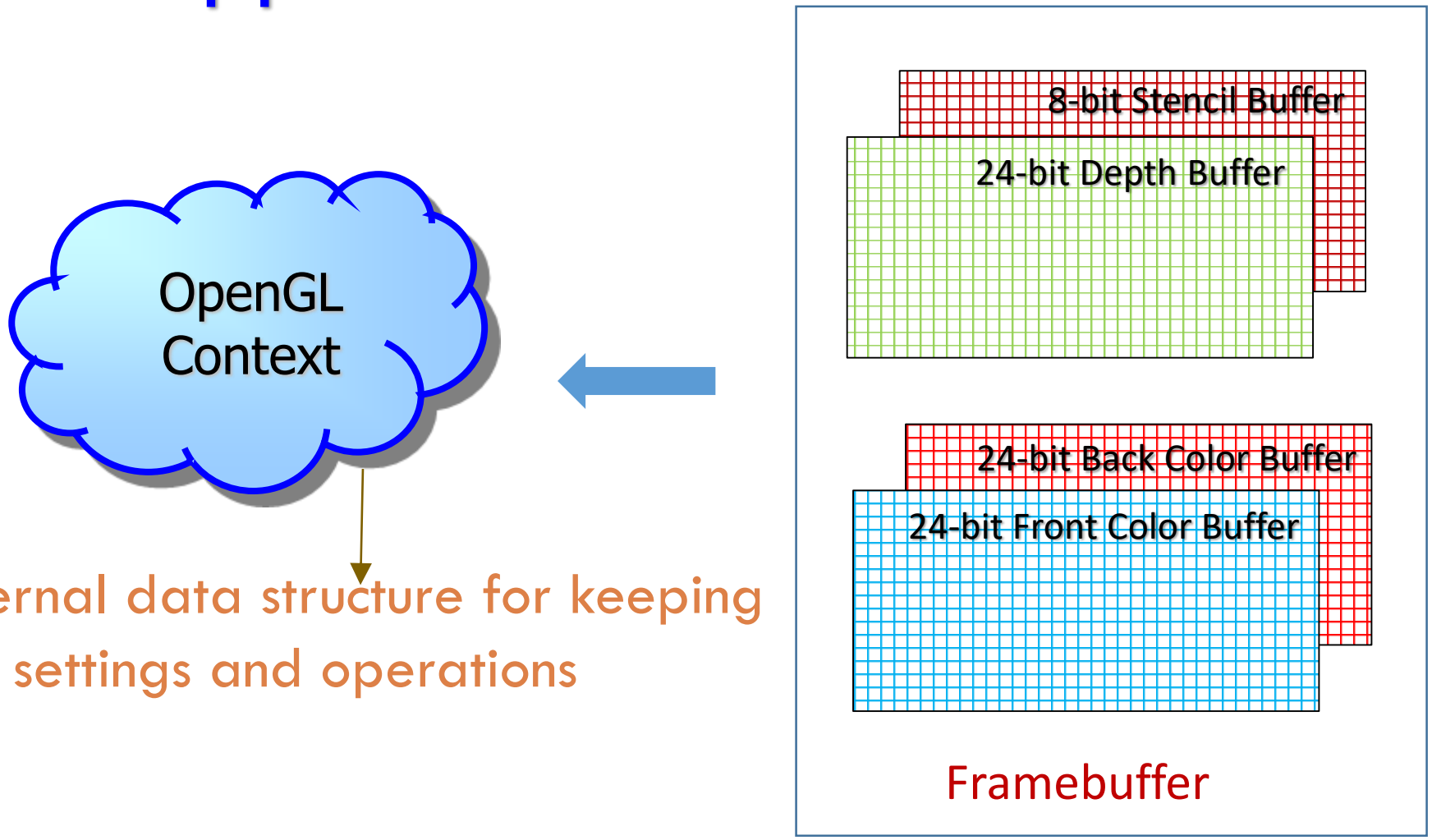
OpenGL API

- OpenGL is *stateful* API that must manage lots of states
- Think of OpenGL as state machine for updating framebuffer contents
- Application programs call OpenGL functions to manage graphics pipes' state machine

OpenGL Context

- Internal, opaque data structure encapsulating *entire* OpenGL state
  - Keeps track of state settings and operations
  - Reflects underlying state of graphics pipe
- Initial graphics context has number of default values
  - Primitives are rendered with white color
  - Background color [to write into colorbuffer] is black
  - Zero active textures
  - ….

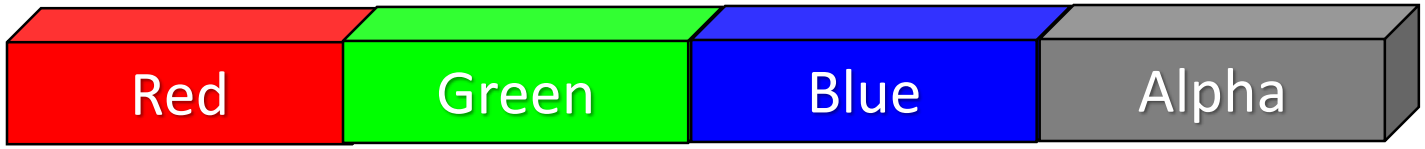
What Resources are Allocated for OpenGL Application?



OpenGL's internal data structure for keeping track of state settings and operations

Color Buffer Format

- RGB or RGBA data



# of bits/color: 8

# of intensities/color: $2^8 = 256$

# of bits/pixel: 24

# of intensities/pixel: $2^{24} = 16{,}777{,}216$

# of bits for alpha: 8

Alpha Values

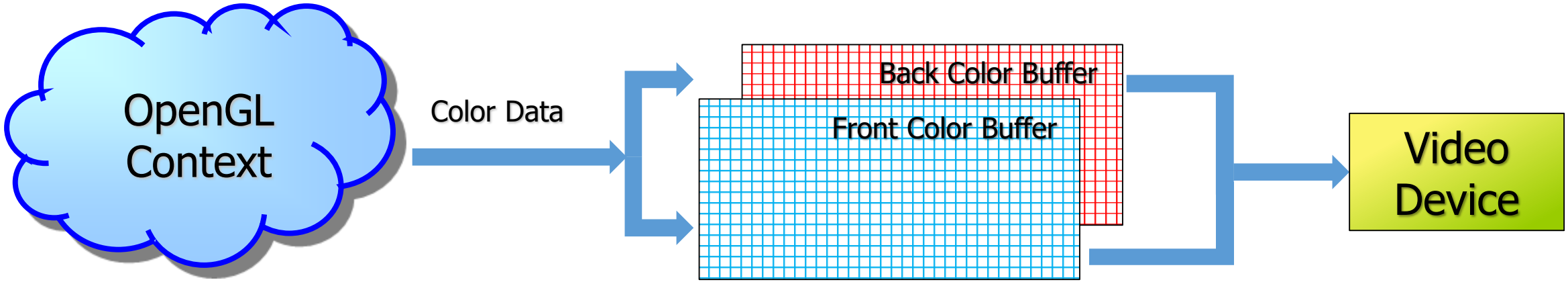
- RGBA format is used to provide per fragment transparency/translucency
  - α is 1.0 implies fragment is opaque
  - α is 0.0 implies fragment is transparent
- 1-bit to 8-bit integral values or 16-bit/32-bit floating point value
- Example of alpha blending equation

$$C = \alpha C_{frag} + (1 - \alpha) C_{colorbuffer}$$

Internal OpenGL Color Format

- Each color component represented using 32-bit floating-point value
  - Values normalized to range [0.0, 1.0] before writing to color buffer
- Programmers can also use floating-point RGBA formats in texture maps
  - Many, many different formats available

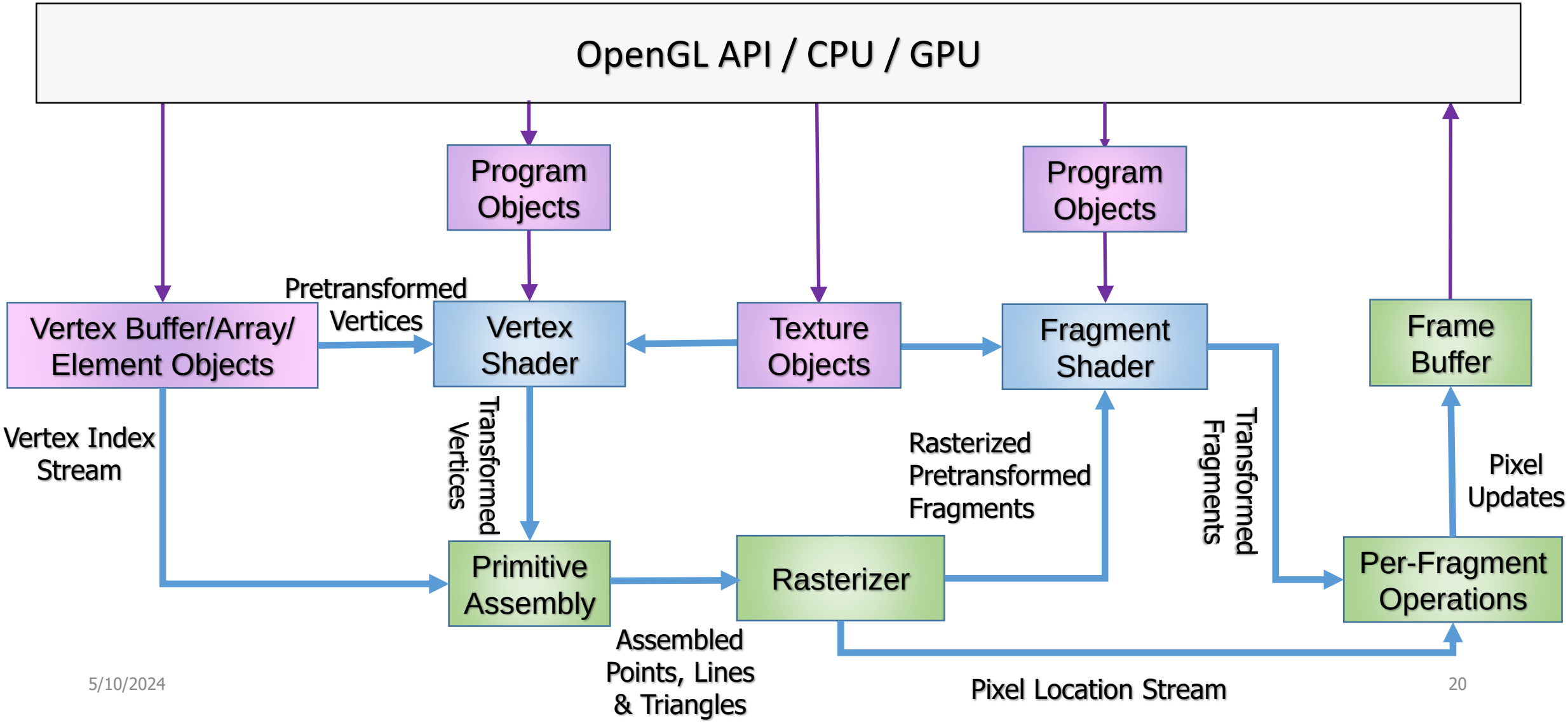
Double Buffered Frame Buffers



OpenGL Programming in a Nutshell

- Create *shader programs*
- Create *buffer objects* and load data into them
- Connect data in buffer objects with shader variables
- Render

Object View of OpenGL Pipe



What is an OpenGL Object? (1/2)

- Encapsulation of related OpenGL state in opaque data structure
- Many object types
  - Regular objects: buffer objects, shader objects, program objects, texture objects, sampler objects, …
  - Container objects contain references to other objects: vertex array objects, framebuffer objects, …
- Each object type corresponds to distinct set of commands which manage objects of that type
- Provides mechanism to encapsulate particular group of state and create, modify, query, and destroy many such instances

What is an OpenGL Object? (2/2)

- Explicitly switching state for each state variable can be inefficient because of high overhead on OpenGL driver
  - Every state change requires validation at draw time which causes slowdown in rendering

Why not collect related stage changes and persistently store them inside OpenGL context?

Rather than explicitly switching states of many variables, single function call will suffice

Other advantage, we can reuse collected state changes

OpenGL API Functions

- Three types of functions
  - Those that change state
  - Those that return state information
  - Those that specify primitives, programs, textures that flow through pipeline

Enabling/Disabling State in OpenGL

- Set and disable OpenGL state using `glEnable()` and `glDisable()`:
  - `glEnable(GL_DEPTH_TEST);`
  - `glEnable(GL_CULL_FACE);`
  - `glDisable(GL_PROGRAM_POINT_SIZE);`

Query State in OpenGL

- Test whether state is enabled using `glIsEnabled()` and `glIsEnabledi()`
- Query states using `glGet*()`:
  - Various calls available depending on what to query
  - `glGetBooleanv(GL_BLEND, &blend_flag);`

Object Management: Name Spaces

- Object types are opaque – defined by implementation
- OpenGL API represents handles to objects using *names* of type unsigned integers
- Name zero reserved by GL
  - For some object types, zero used to name default object
  - For other object types, zero will never correspond to actual instance of that object type

Object Management: Name Generation

- Objects' named by generating unused names using functions starting with **`glGen*()`** followed by object type
  - For example, **`glGenBuffers()`** returns one or more previously unused buffer object names

Object Management: Object Creation (1/2)

- Generated names don't initially correspond to an object instance – no resources allocated yet
- Objects (with generated names) created by ***binding*** generated name to context using **`glBindBuffer()`**
  - Binding allocates resources for object and its state
- For example:
  - Name of buffer object generated by **`glGenBuffers()`**
  - Buffer object created by calling **`glBindBuffer()`** with name returned by **`glGenBuffers()`**

Object Management: Object Creation (2/2)

- Objects also created directly by functions that return new name or names representing freshly initialized object
- Some functions return single object name directly
  - For example `glCreateProgram()` for program objects
- Some functions able to create large number of new objects, returning their names in array
  - Examples are `glCreateBuffers()` for buffers, `glCreateTextures()` for textures, and `glCreateVertexArrays()` for vertex arrays

Object Management: Object Instances

- Can instantiate many objects of same type
- Particular instance can made *active* using **`glBindBuffers()`**
  - Allows state encapsulated by instance to be mapped to OpenGL context
  - Means functions operating on bound object will now affect OpenGL context

Object Management: Name Delection and Object Deletion

- Objects deleted by calling deletion functions specific to that object type
  - For example, **`glDeleteBuffers`** is passed array of buffer object names to delete
- After object is deleted, it has no contents
- Object's name is once again marked unused for purpose of name generation

Object Management: What are Buffer Objects?

- Since cost of communication more expensive than cost of computation, data transfer in GL happens through *buffer objects*
- Chunks of server memory managed by server
- Populated with data that will be consumed by server
  - Stores information about vertices, indices, adjacency information, and user-defined data

Life-Cycle of Buffer Objects

- Generate named buffer objects
- Bind buffer object
  - If buffer object doesn't exist, create it
  - If buffer object exists, then make buffer object current or active
- Perform data transfer to/from buffer
- Use buffer to draw
- Delete buffer object when no longer required
  - Failure to do so will leave memory leak in GL implementation

Handles to Buffer Objects

- Client code cannot directly access GL's server-side memory
- Indirect references
  - Pointers are useless
  - Integers are used as indices into server-side memory pool

Comparison to C++ Resource Managment

OpenGL

```
GLuint buffers[4]; // handle
glGenBuffers(4, buffers);
glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBindBuffer(GL_TEXTURE_BUFFER, buffers[1]);
```

C++

```
void **buffers;
buffers = new (void*)[4];
buffers[0] = (Array*) new Array();
buffers[1] = (Texture*) new Texture();
```

What is Binding?

- Binding specifies type of data that will be written to/read from specific buffer object
- **`glBindBuffer( GLenum target, GLuint buffer )`**
  - **`target`**: a valid “buffer type” as specified by GL - 14 types in version 4.0
  - **`buffer`**: previously generated name for GL buffer object

Fill It Up!!!

- Preceding steps create empty buffer object
- Data transfer
  - More than one way to do it – most common `glBufferData()` and `glBufferSubData()`
  - Client replaces data partially through larger chunk
  - Server generates data and records it into buffer object

Discarding Buffer Data

- Inform GL server that data is not required anymore using `glInvalidateBufferData()` or `glInvalidateBufferSubData()`

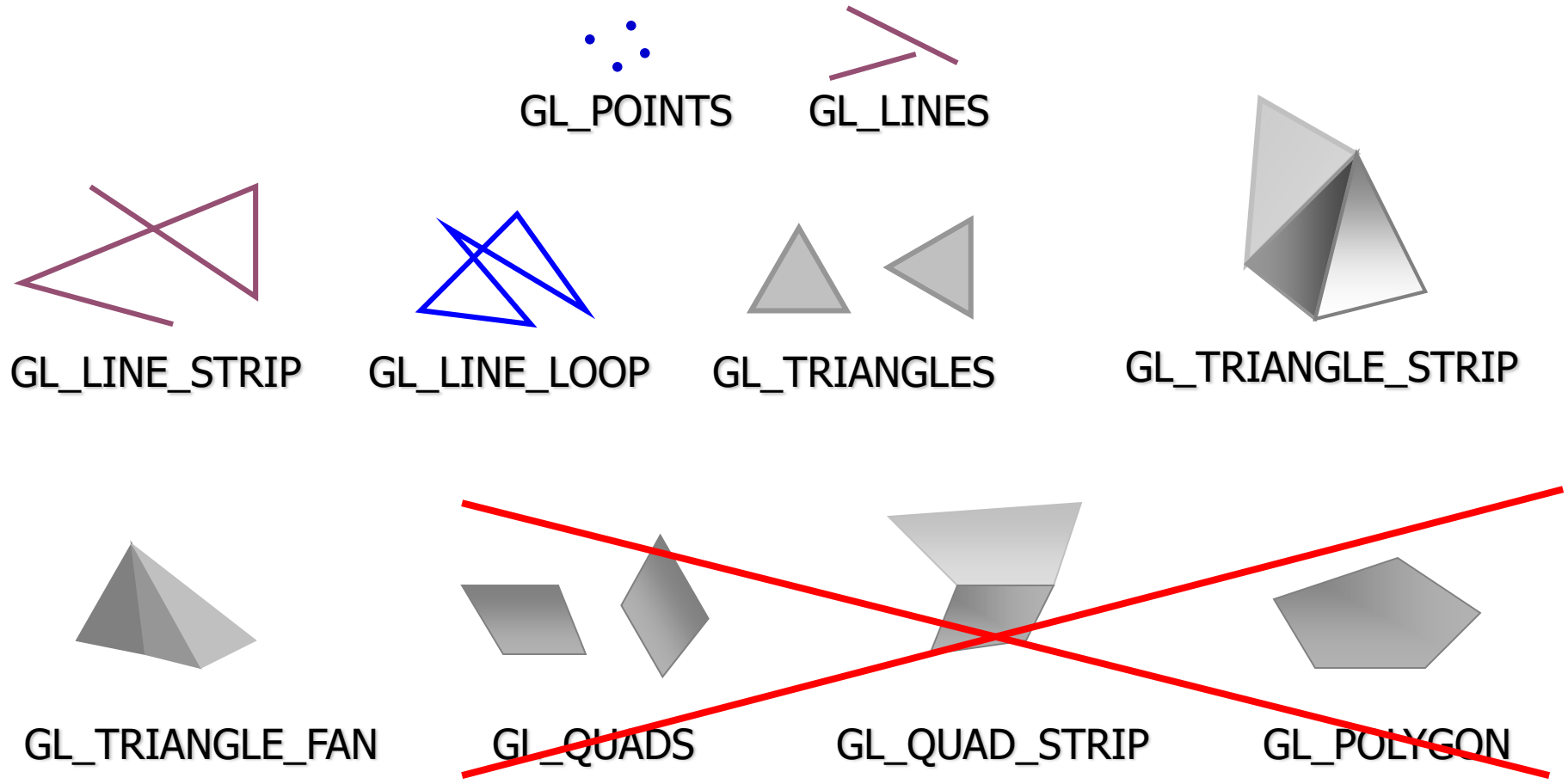
Giving It Up!!!

- When no longer required, return data store back to server memory pool using **`glDeleteBuffers()`**

OpenGL Geometric Primitives (1/2)

- GL provides mechanisms to describe how complex geometric objects are to be rendered
- Doesn’t provide mechanisms to describe the complex objects themselves

OpenGL Geometric Primitives (2/2)



OpenGL Data Types

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type int is referred to as GLint outside this document, and is not necessarily equivalent to the C type int. An implementation must use exactly the number of bits indicated in the table to represent a GL type.
ptrbits is the number of bits required to represent a pointer type; in other words, types intptr, sizeiptr, and sync must be large enough to store any CPU address. sync is defined as an anonymous struct pointer in the C language bindings while intptr and sizeiptr are defined as integer types large enough to hold a pointer.

OpenGL 4.6 (Core Profile) - May 14, 2018

Types in table to right must be prefixed with `GL` as in `GLboolean`

GL Type	Bit Width	Description
boolean	8	Boolean
byte	8	Signed two's complement binary integer
ubyte	8	Unsigned binary integer
char	8	Characters making up strings
short	16	Signed two's complement binary integer
ushort	16	Unsigned binary integer
int	32	Signed two's complement binary integer
uint	32	Unsigned binary integer
fixed	32	Signed two's complement 16.16 scaled integer
int64	64	Signed two's complement binary integer
uint64	64	Unsigned binary integer
sizei	32	Non-negative binary integer size
enum	32	Enumerated binary integer value
intptr	*ptrbits*	Signed twos complement binary integer
sizeiptr	*ptrbits*	Non-negative binary integer size
sync	*ptrbits*	Sync object handle (see section 4.1)
bitfield	32	Bit field
half	16	Half-precision floating-point value encoded in an unsigned scalar
float	32	Floating-point value
clampf	32	Floating-point value clamped to [0, 1]
double	64	Floating-point value
clampd	64	Floating-point value clamped to [0, 1]

Source: OpenGL 4.6 API Specification

OpenGL Constants

- Many, many constants prefixed with `GL_`
  - `GL_POINTS`, `GL_LINES`, `GL_TRIANGLE`, ...

OpenGL Function Syntax

- Spec describes GL functions using ANSI C syntax
- Some functions such as **`glViewport`** have straightforward functionality
- Not so much with other functions such as **`glUniform*`** or **`glGet*`**
  - Various group of functions perform same operation but differ in how arguments are supplied to them
  - This variation accommodated using specific notation for describing functions and their arguments

Deciphering OpenGL Function Syntax (1/2)

```
rtype Name{ϵ|1|2|3|4}{ϵ|b|s|i|i64|f|d|ub|us|ui|ui64}{ϵ|v}
      ([args,] T arg1,..., T argN [,args]);
```

- **`rtype`** is function's return type
- Implementations add prefix **`gl`** to **`Name`**
- Arguments enclosed in brackets (**`[args,]`** and **`[,args]`**) may or may not be present
- Braces (**`{ϵ b s i i64 f d ub us ui ui64}`**) enclose series of type descriptors which are used for specifying corresponding types for parameters
  - **`ϵ`** indicates no type descriptor
- **`N`** arguments **`arg1`** thro' **`argN`** have type **`T`** which is associated with type descriptor
  - If type descriptor is not used, then **`T`** is explicitly given

Deciphering OpenGL Function Syntax (2/2)

```
rtype
Name{ϵ|1|2|3|4}{ϵ|b|s|i|i64|f|d|ub|us|ui|ui64}{ϵ|v}
     ([args,] T arg1,..., T argN [,args]);
```

- If final character is not **`v`**, then **`N`** is given by digit **`1`**, **`2`**, **`3`**, or **`4`**
  - **`ϵ`** indicates there is no digit and number of arguments is fixed
- If final character is **`v`**, then only **`arg1`** is present and it is array of **`N`** values of indicated type

OpenGL Function Syntax: `Uniform{1234}{if}`

```
void Uniform{1234}{if} (GLint location, T value)
```

indicates eight declarations:

```
void glUniform1i(GLint location, GLint value);
void glUniform1f(GLint location, GLfloat value);
void glUniform2i(GLint location, GLint v0, GLint v1);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3i(GLint location, GLint v0, GLint v1, GLint v2);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4i(GLint loc, GLint v0, GLint v1, GLint v2, GLint v3);
void glUniform4f(GLint loc, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);
```

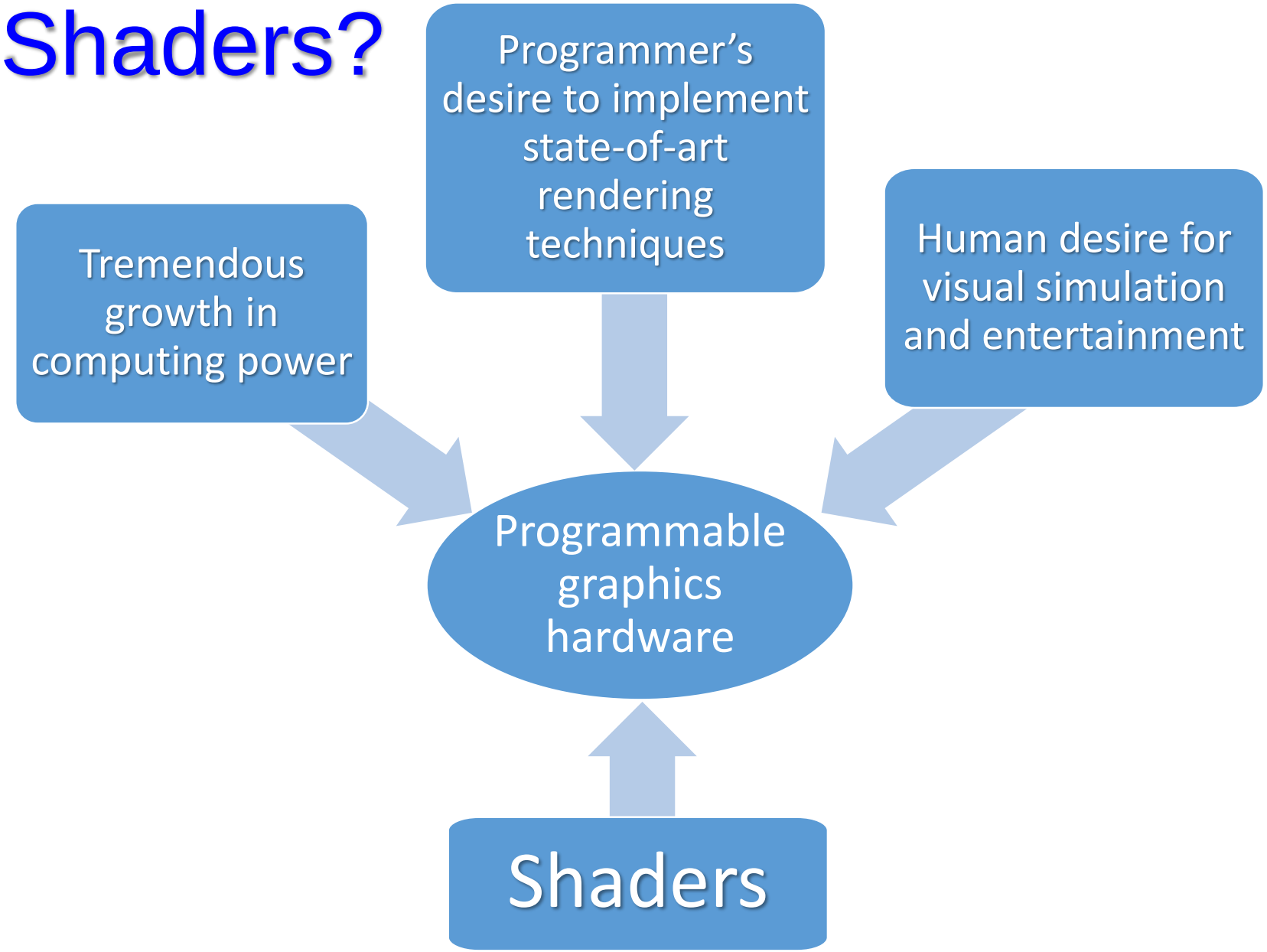
OpenGL Function Syntax: `Uniform{1234}{if} {v}`

```
void Uniform{1234}{if} (GLint location, T value)
```

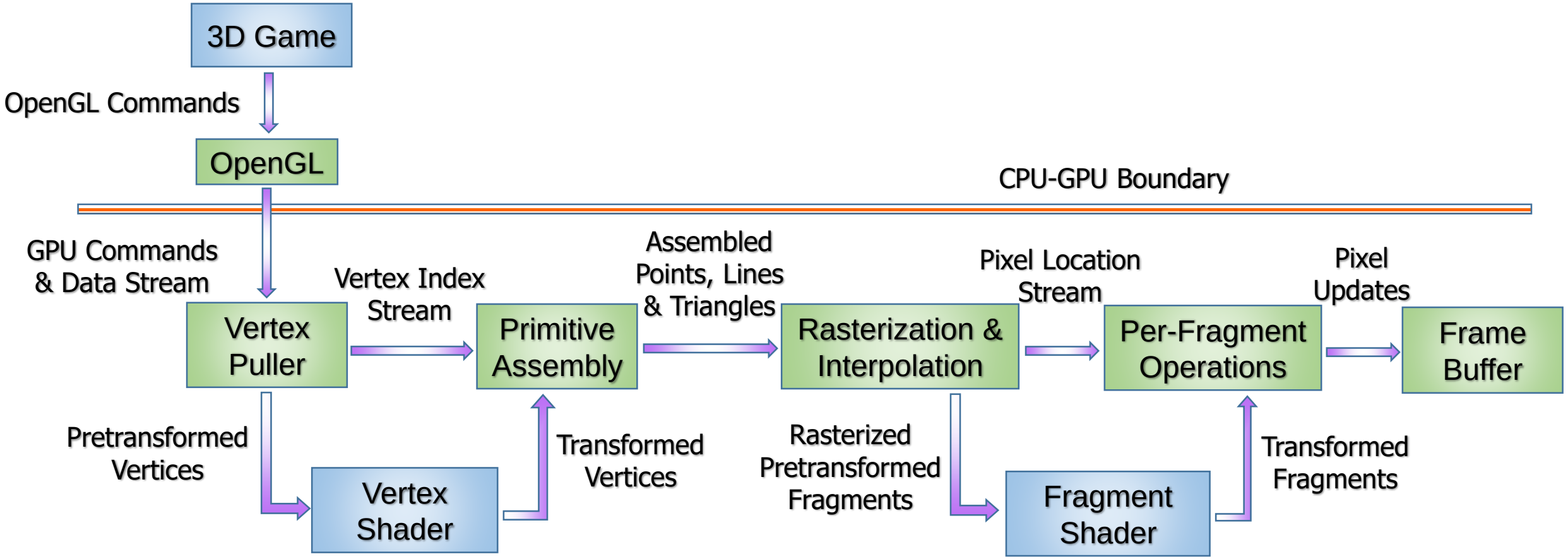
indicates eight declarations:

```
void glUniform1fv(GLint location, GLsizei count, const GLfloat * value);
void glUniform2fv(GLint location, GLsizei count, const GLfloat * value);
void glUniform3fv(GLint location, GLsizei count, const GLfloat * value);
void glUniform4fv(GLint location, GLsizei count, const GLfloat * value);
void glUniform1iv(GLint location, GLsizei count, const GLint * value);
void glUniform2iv(GLint location, GLsizei count, const GLint * value);
void glUniform3iv(GLint location, GLsizei count, const GLint * value);
void glUniform4iv(GLint location, GLsizei count, const GLint * value);
```

Why Shaders?

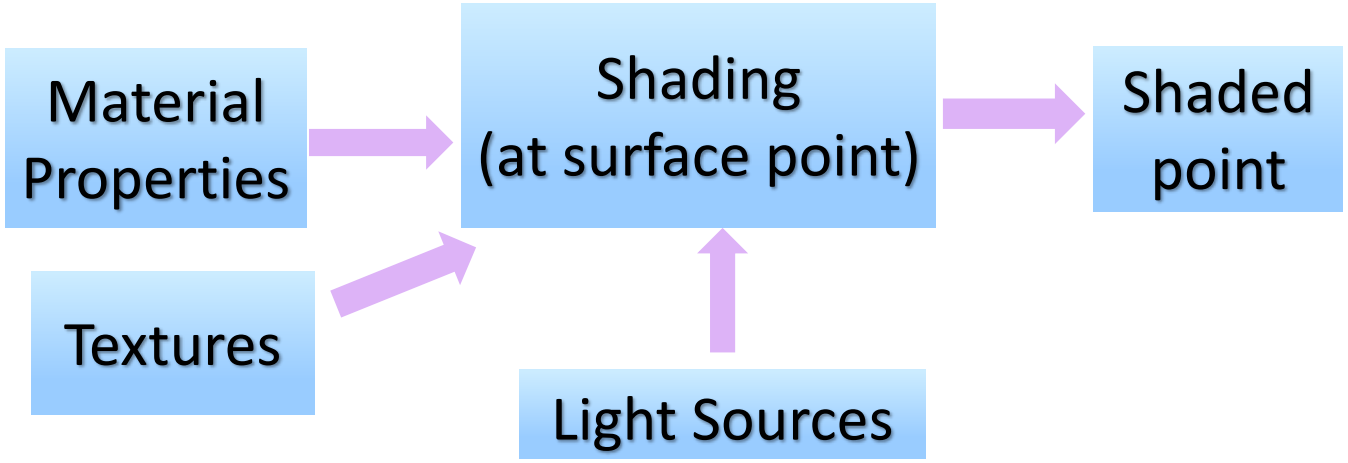

Programmer's desire to implement state-of-art rendering techniques
Tremendous growth in computing power
Human desire for visual simulation and entertainment
Programmable graphics hardware
Shaders

Data Flow in Graphics Pipe



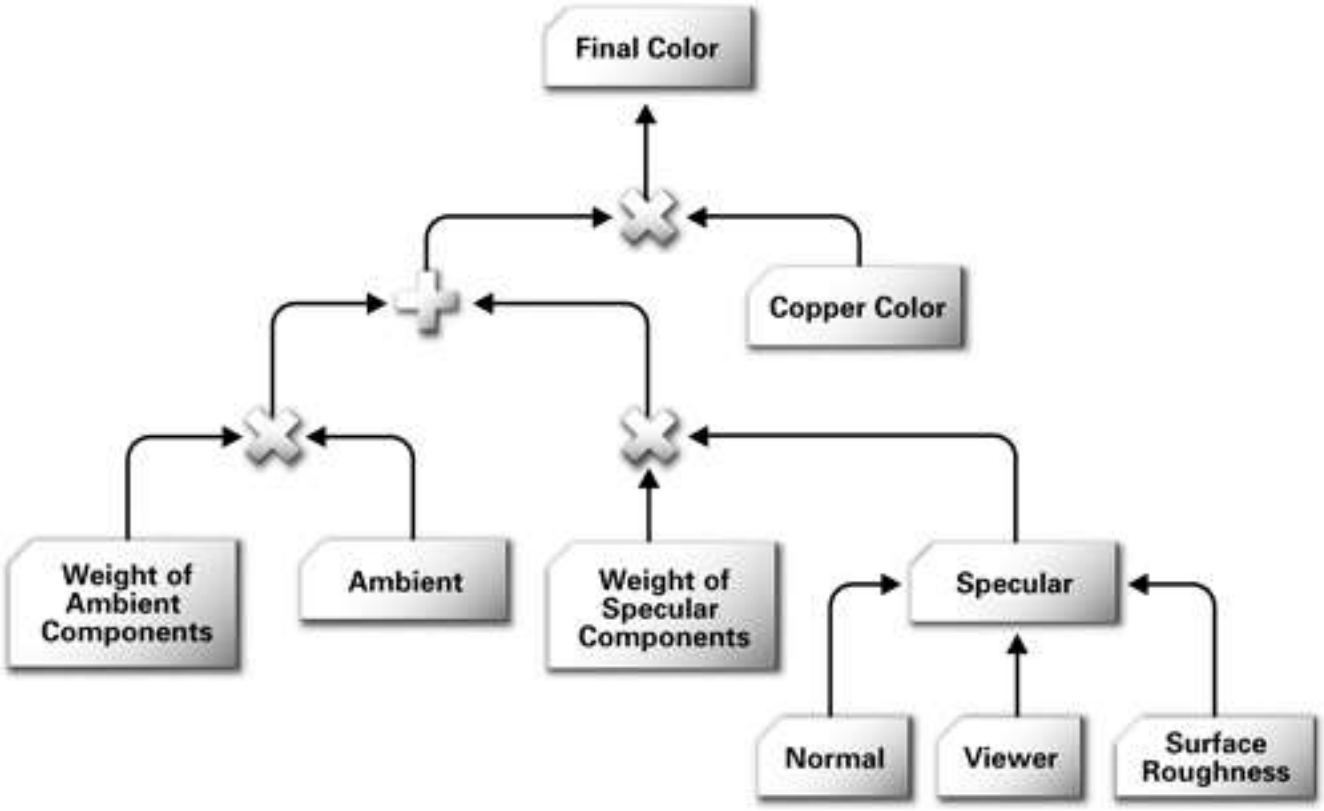
What is a Shader? (1/2)

- *Shading* is process of using equation to compute color (at surface point or pixel) based on light sources and surface's material properties



What is a Shader? (2/2)

- *Shader* is particular implementation of shading process
- Now, shader represents *any* programmer-defined operations executed by any GPU programmable stage



Reference

Shader Environment

- Shaders are programmed individually and stored as external source files
- OpenGL-based graphics application now behaves as host application to manage shaders and their resources

Vertex Shader: Model-to-Clip Xform

```
static GLchar const* vertex_shader_source[] = {
  "#version 450 core                                    \n"
  "                                                     \n"
  "layout (location=0) in vec3 vVertexPosition;         \n"
  "layout (location=1) in vec3 vVertexColor;            \n"
  "                                                     \n"
  "smooth out vec3 vSmoothColor;                        \n"
  "                                                     \n"
  "uniform mat4 uMVP;                                   \n"
  "                                                     \n"
  "void main () {                                       \n"
  "  gl_Position = uMVP * vec4(vVertexPosition, 1.f);   \n"
  "  vSmoothColor = vVertexColor;                       \n"
  "}                                                    \n"
};
```

Vertex Shader Source in File

```
#version 450

layout (location=0) in vec3 vVertexPosition;
layout (location=1) in vec3 vVertexColor;

out vec3 vSmoothColor;

uniform mat4 uMVP;

void main() {
  vSmoothColor = vVertexColor;
  gl_Position = uMVP * vec4(vVertexPosition, 1.f);
}
```

Fragment Shader: Write Through (Ver 1)

```
static GLchar const* fragment_shader_source[] = {
  "#version 450 core                                  \n"
  "                                                   \n"
  "in vec3 vSmoothColor;                              \n“
  "out vec4 fFragColor;                               \n"
  "                                                   \n"
  "void main () {                                     \n"
  "  fFragColor = vec4(vSmoothColor, 1.f);            \n"
  "}                                                  \n"
};
```

Fragment Shader: Write Through (Ver 2)

```
static GLchar const* fragment_shader_source[] = {
  "#version 450 core                                  \n"
  "                                                   \n"
  "in vec3 vSmoothColor;                              \n"
  "                                                   \n"
  "void main () {                                     \n"
  "  gl_FragColor = vec4(vSmoothColor, 1.f);          \n"
  "}                                                  \n"
};
```

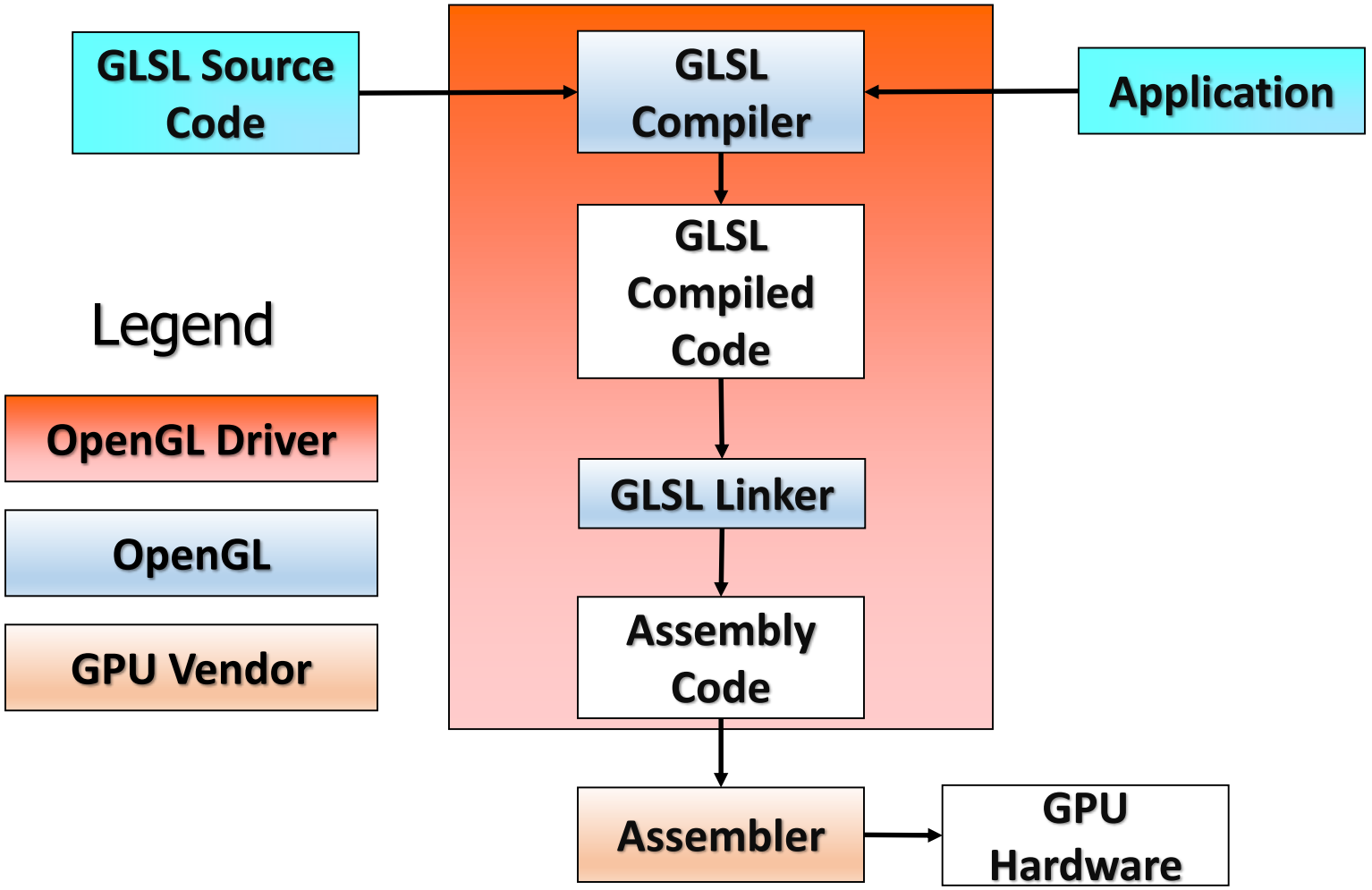
Fragment Shader Source in File

```
#version 450

in vec3 vSmoothColor;

void main() {
  gl_FragColor = vec4(vSmoothColor, 1.f);
}
```

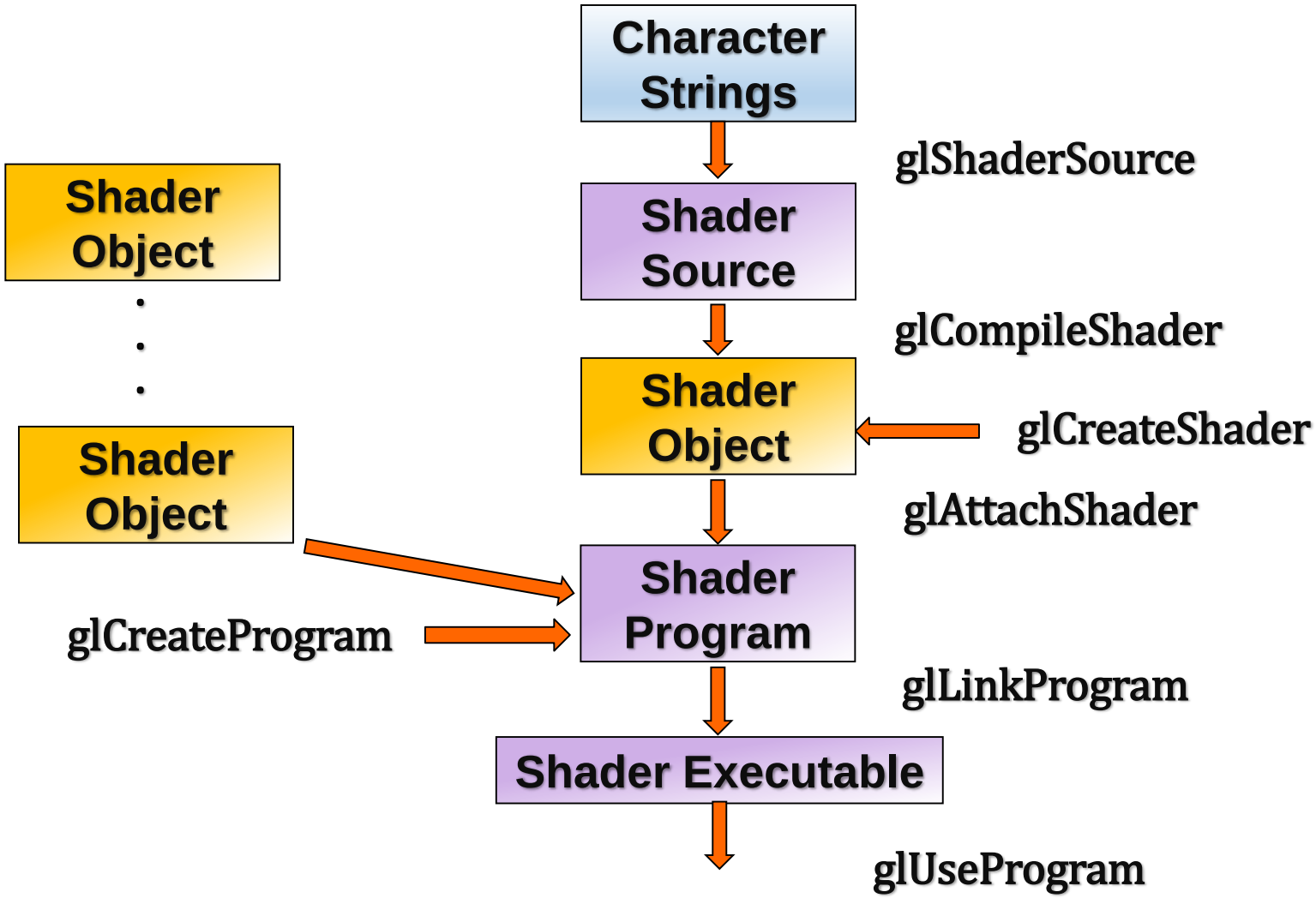
GLSL Production Pipeline (1/2)

- How to compile shader and pass compiled shader to GPU for execution?
  - OpenGL API is used to compile, link, execute and debug shaders
  - OpenGL 4.1 added ability to save compiled shader programs to file to avoid runtime overhead of shader compilation

GLSL Production Pipeline (2/2)



OpenGL Shader Processing Commands



GLSL Production Pipeline