

## Rendering Pipeline and Programmable Shaders

### Topics

Rendering

Shader Types

OpenGL Shader Language Basics

Review

Review

Some Definitions

**Primitive:**

An object that can be handled by a 3D rendering system such as OpenGL and Vulkan.

OpenGL and Vulkan Primitives: **triangles**, **lines**, **points**.

**Vertex:**

In OpenGL and Vulkan, a set of information about what is usually a geometric vertex. It includes a coordinate and possibly, color, surface normal, etc.

**Vertex Attribute:**

Information associated with a vertex, such as color, normal, etc.

Rendering

Rendering

**Rendering:**

Writing an image to a frame buffer corresponding to some geometry.

Two Rendering Methods:

**Rasterization:**

Operate on each primitive once.

Operate on each pixel multiple times.

**Ray Tracing**

Operate on each primitive many times.

Operate on each pixel once.

## Rasterization

- Transform coordinates to window space.
- Compute list of pixel coordinates covered by triangle. Call these fragments.
- For each coordinate:
  - Compute lighted color at that point.
  - Retrieve texel. Combine with lighted color.
  - Check depth buffer and discard if incumbent closer to eye.
- Write frame buffer.

## Limitations of Rasterization

Can't directly tell if light is blocked (and so there is a shadow).

There are methods such as **shadow volumes** that require extra rendering passes.

Can't directly handle reflections and refractions.

## Ray Tracing

For each pixel in the frame buffer:

    Compute a ray from the eye to the pixel. (Say in object space.)

    Find the primitive...

    ... that is intersected by the ray...

    ... and is closest to the eye.

    Compute a ray from the intersection point to the light.

    Determine whether any primitive is intersected by that ray.

    If surface is reflective, compute a ray ..

    Compute lighted color based on ...

    ... light location ...

    ... and primitive orientation relative to light and eye.

Rendering

## Parallelizability and Computational Efficiency

Good: task b does not depend on task a.

Good: the memory locations that are needed are known in advance.

Easy to parallelize

```
for ( int i=0; i<10000; i++ ) b[i] = a[i] * 7;
```

Not parallelizable:

```
// Elements of a are in the range 0 to 99999.  
for ( int i=0; i<10000; i++ ) idx = a[idx];
```

Rendering

Rasterization:

Array of primitives.

Can divide primitives between threads.

Array of fragments for each primitive.

Can divide fragments between threads.



Rendering

## Ray Tracing

**Ray casting:** Finding the closest primitive.

Naïve: Go through the list..

Actual:

Organize primitives into a **oct-tree** or a **bounding volume hierarchy (BVH)**.

Tree-like organization. Degree 8.

Root: entire world.

Divide up into eight children.

Those in turn divided.

Put a primitive in the lowest position in which it fits in box.

Search this tree.

## Coordinate Spaces

The right coordinate space can simplify a task.

For that reason OpenGL (especially compatibility mode) and Vulkan specify specific coordinate spaces for specific parts of the system.

### Object Space:

The coordinate space of the vertices entering a rendering pipeline or provided for ray tracing. There is no restriction on this coordinate space.

### World Space:

A coordinate space common to all objects.

### Eye Space:

A coordinate space in which the user's eye is at the origin and the user's monitor (projection plane) faces the  $-z$  direction.

### Clip Space:

A coordinate space in which the view volume is within a cube centered at the origin with an edge length of 2.

### Window Space:

A coordinate space in which the origin is at the lower-right of the window and units are in pixels.

## Rasterization

### Rasterization:

The process of finding the location of pixels covered by a primitive.

#### Rasterization Input:

Primitive's vertices ...  
... with coordinates in window space.

Primitive's attributes.

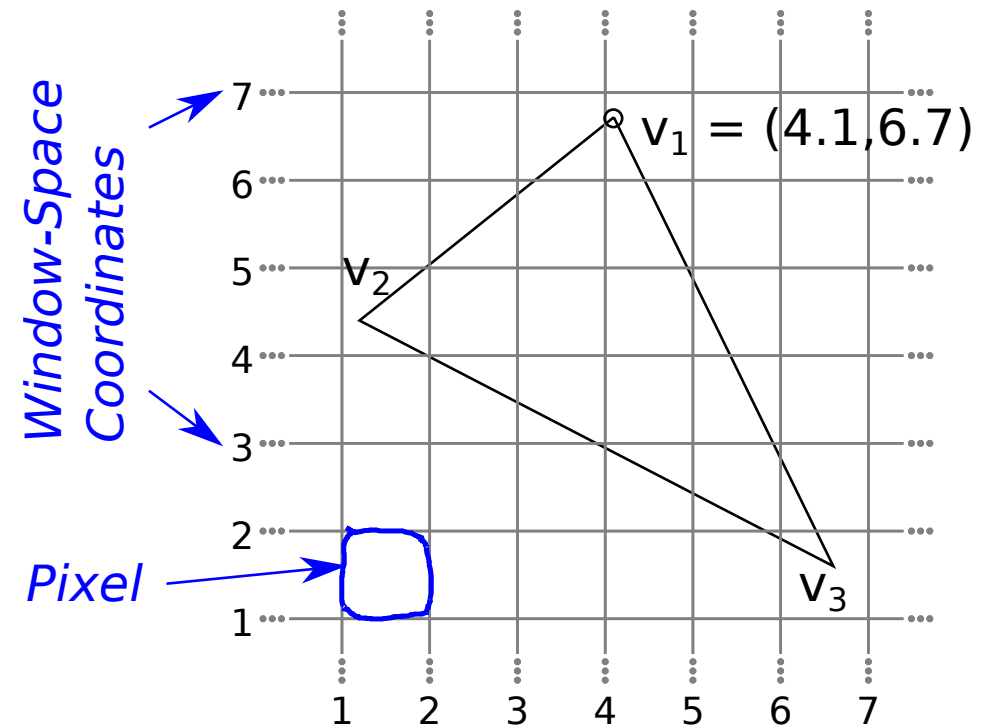
#### Rasterization Output:

Coordinate of each pixel covered by primitive.

Interpolated attributes for each pixel.

### Fragment:

Information on a pixel covered by a primitive.



## Rasterization Question

By how much should pixel be covered by primitive?

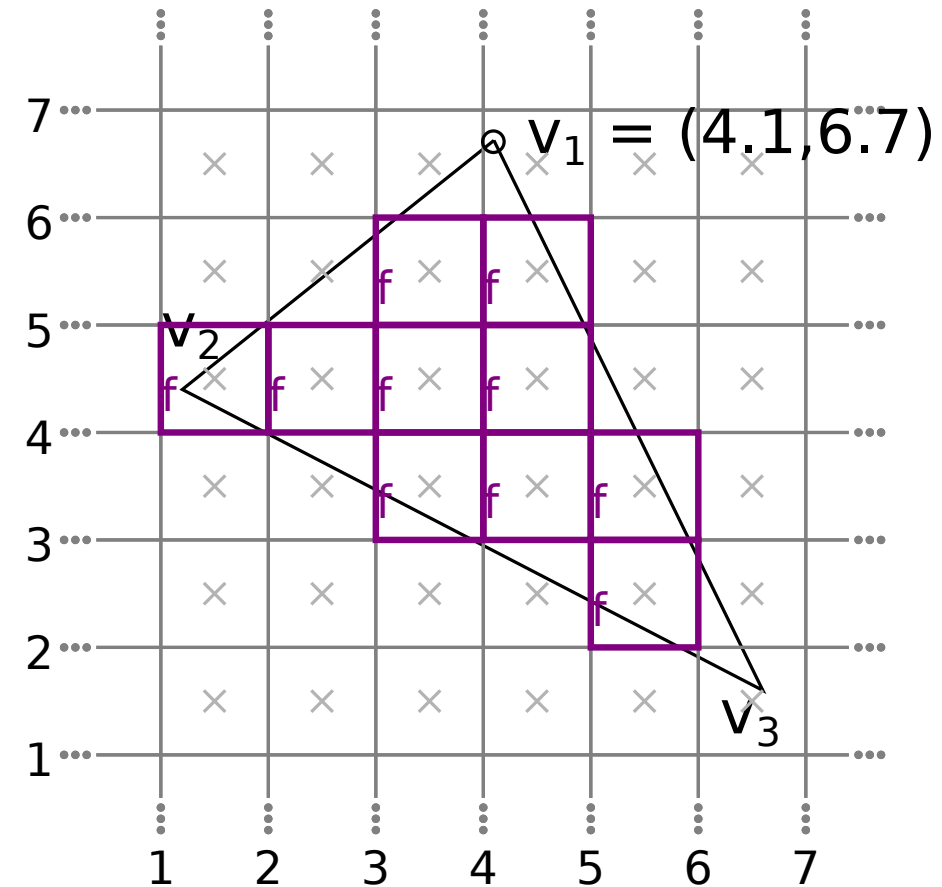
### OpenGL's Answer for Triangles (Polygons)

A fragment is generated if center of pixel is inside primitive.

A special case applies if pixel center is on a shared edge.

OpenGL has rules for other primitives, and can apply antialiasing.

For course, only consider triangles without antialiasing.



## Interpolation of Attributes

Fragment includes attributes (associated data) interpolated from primitive vertices.

### Types of Interpolation (OpenGL Shader Language Terminology)

#### smooth (Perspective Correct) Interpolation:

Attribute value at point  $v$  of triangle  $v_1v_2v_3$  is linearly interpolated from the attribute value at each vertex based on object space coordinates.

Computationally costly (requires division), but correct.

#### noperspective Interpolation:

Attribute value at point  $v$  of triangle  $v_1v_2v_3$  is linearly interpolated from the attribute value at each vertex based on projected  $x$  and  $y$  coordinates (clip space or window space).

#### flat (no) Interpolation:

Attribute value at point  $v$  of triangle  $v_1v_2v_3$  is the value of the attribute of  $v_3$  (the provoking vertex).

Saves time when attributes are the same at all three vertices.

## Important Point

One primitive can generate many fragments.

Fragment usually carries:

Window-space coordinates. (Exact position of pixels.)

Interpolated lighted color.

For writing to the frame buffer.

Interpolated  $z$  value.

Used to determine whether fragment is under or over another fragment.

### **Pipeline:**

An organization for software and hardware which defines a fixed sequence of stages. Each stage carries out some operation, receiving its input data from the prior stage and providing its output data to the next stage. All data pass through the same stages in the same order.

### **Rendering Pipeline:**

An organization for the set of steps needed to convert a set of vertices into a frame buffer image.

The term rendering pipeline might be used generically ...  
... or it might refer to something very specific.

“Clipping is the most tedious step in the rendering pipeline.”.

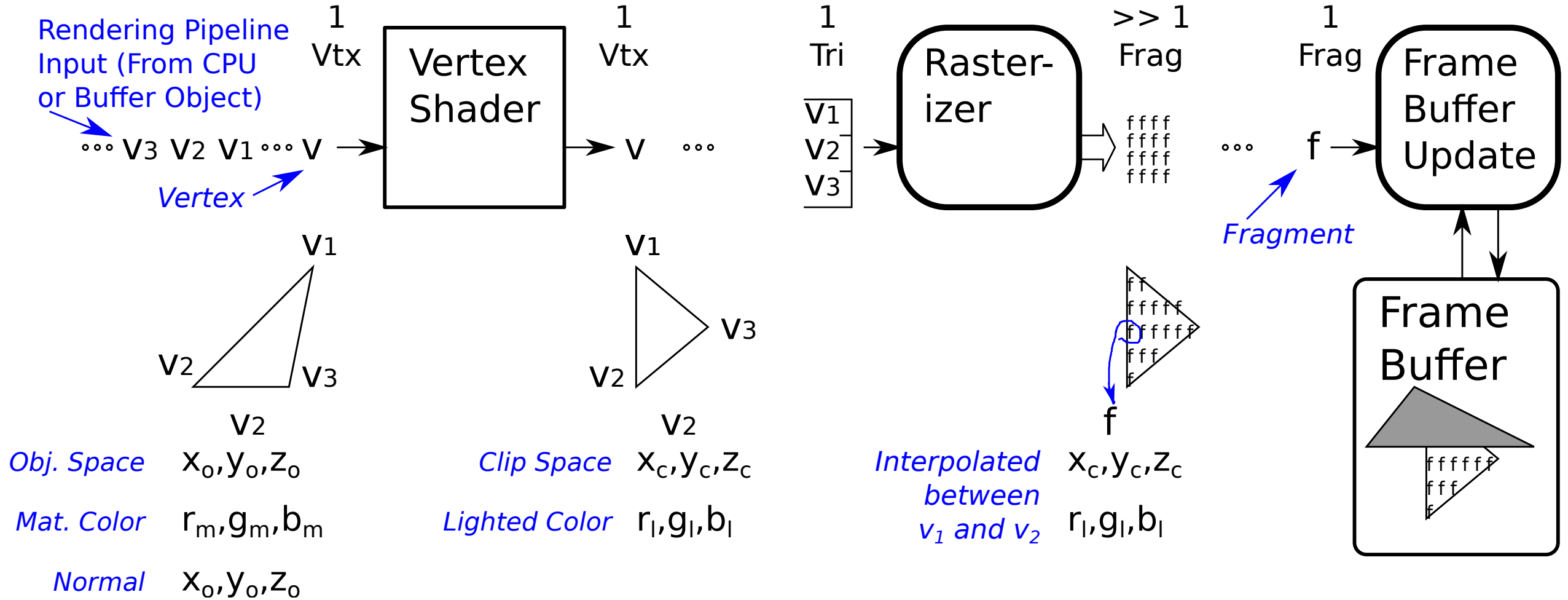
### **OpenGL Rendering Pipeline:**

The sequence of steps defined by OpenGL...  
... that start with a vertex and its attributes ...  
... and usually result in the frame buffer being written.

### **Rendering Pass:**

The use of the rendering pipeline to render some set of primitives.

Simplified OpenGL/Vulkan Rendering Pipeline





Rendering Pipeline » Simplified Rendering Pipeline

## OpenGL and Vulkan Rendering Pipelines

Defined by the OpenGL (4.6) and Vulkan (1.2) standards.

## Definitions

### **Stage:**

A pipeline section.

### **Programmable Stage (or Unit):**

An OpenGL RP stage which can perform its operation by executing user-provided software.

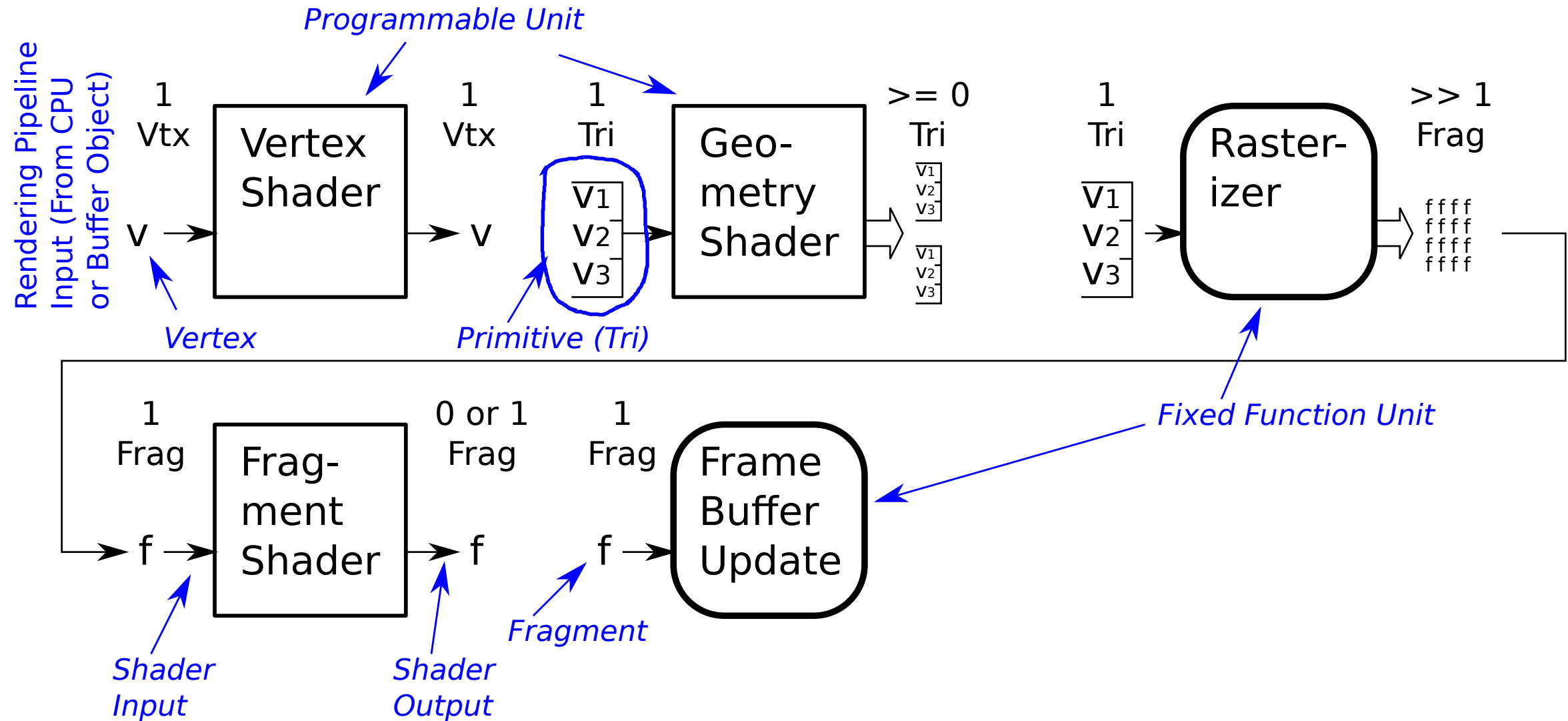
### **Fixed-Function Stage (or Unit):**

An OpenGL RP stage which cannot be programmed, its functionality is specified by the standard and provided by the implementation.

### **Shader:**

A program set up to run in a programmable stage, or the programmable stage itself.

Typical OpenGL Rendering Pipeline



## Performing a Rendering Pass

Two Methods in OpenGL:

```
glBegin(PRIMITIVE); ... glEnd();
```

Slow because vertices supplied one by one.

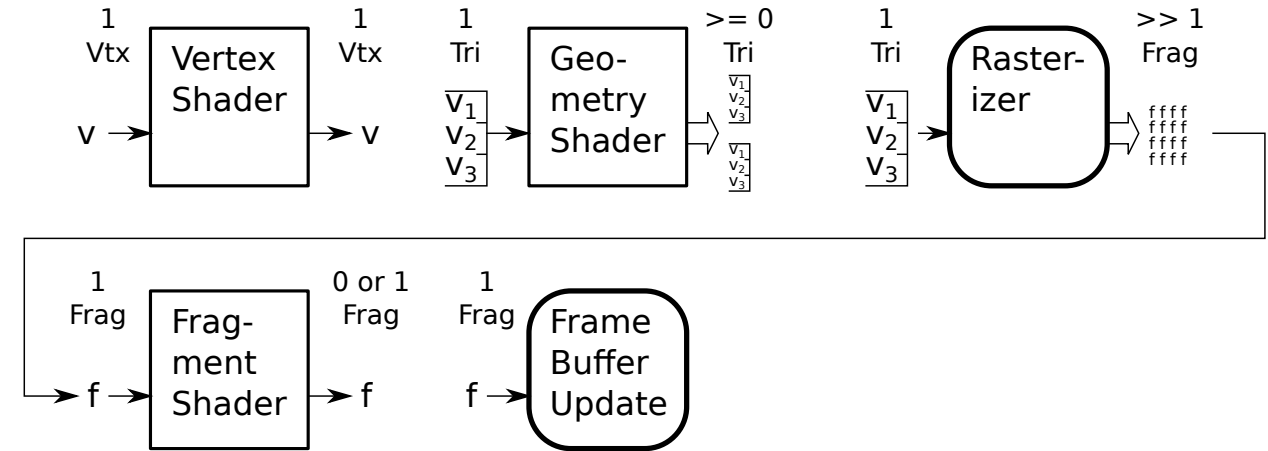
```
glDrawArray(PRIMITIVE, ...) and related commands.
```

Fast because vertices supplied in an array.

Example: Invoking a rendering pass using `glBegin`:

```
glBegin(GL_TRIANGLE_STRIP); // Start rendering pass using ..
glColor3fv(lsu_spirit_gold); // .. a rendering pipeline for triangles.
for ( int i=0; i<size; i++ )
    glNormal3f(norms[i].x,norms[i].y,norms[i].z);
    glVertex3f(coords[i].x,coords[i].y,coords[i].z); }
glEnd(); // End of rendering pass.
```

During the rendering pass the CPU sends vertices into the rendering pipeline ...  
 ... which results in the frame buffering being updated at the RP end.



## Programmable Shaders

### Programmable Shaders' Role

Written by OpenGL Programmers (that would be us).

The shaders implement the functionality of a RP stage.

## Shader Invocation

### Big Difference with “Normal” Programming, Like C

C:

Must write a routine called `main`.

The routine `main` is started once each time the program is run.

This should seem obvious. How else would one do it?

### OpenGL Shading Language

For each rendering pass ...

... can provide a shader for each programmable unit.

Shader is run once **for each item** passing through pipeline.

Item can be a vertex, primitive, or fragment.

## Shader Inputs and Output

### Types of Storage

- **in**            Input data from a previous stage.
- **out**            Output data for a subsequent stage.
- **uniform**    Read-only data provided by CPU.
- **buffer**        A buffer object. Can be read and written.

## Predefined Shader Inputs and Outputs

Each stage has predefined inputs and outputs.

Used to communicate with fixed-function hardware.

For example, `gl_Position` is used by rasterizer.

## Compatibility Profile Inputs and Outputs

These are for GPUs that could not be fully programmed.

For example, `gl_Color`.

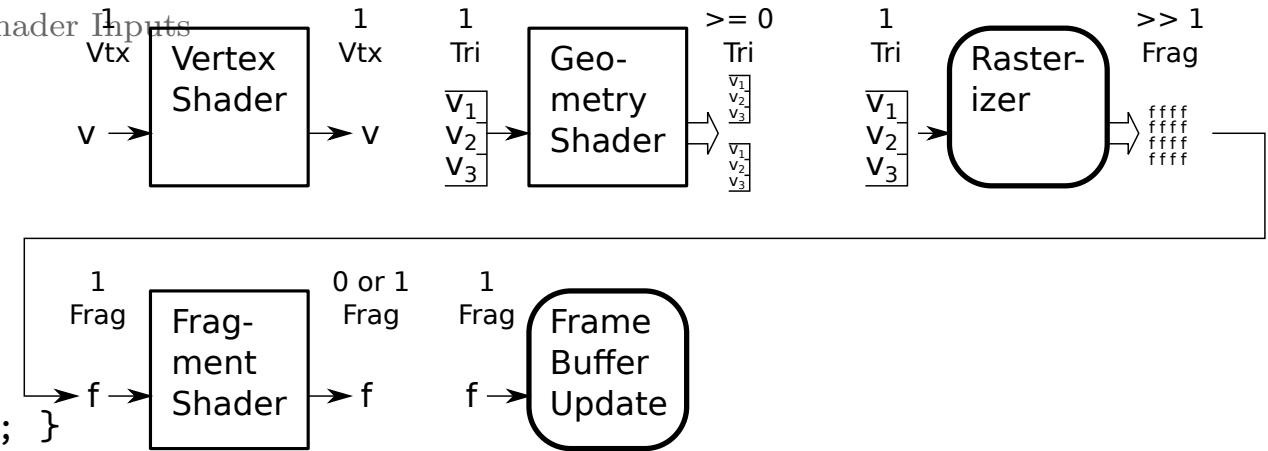


### Compatibility Vertex Shader Inputs

```

glBegin(GL_TRIANGLE_STRIP);
glColor3fv(lsu_spirit_gold);
for ( int i=0; i<size; i++ )
    glNormal3f(norms[i].x,norms[i].y,norms[i].z);
    glVertex3f(coords[i].x,coords[i].y,coords[i].z); }
glEnd();

```



Each vertex in this example has the following attributes:

- A coordinate (specified by `glVertex3f`).
- A normal (specified by `glNormal3f`).
- A color (specified by `glColor3fv`).

Each execution of `glVertex3f` sends one vertex into the vertex shader.

In the diagram a vertex, including all its attributes, shown as `v`, `v1`, etc.

## Major Rendering Pipeline Stages

The major stages summarized in the next few slides.

Some are programmable, some are fixed function.

## The Vertex Shader

Input: One vertex.

Output: One vertex.

Historical Role:

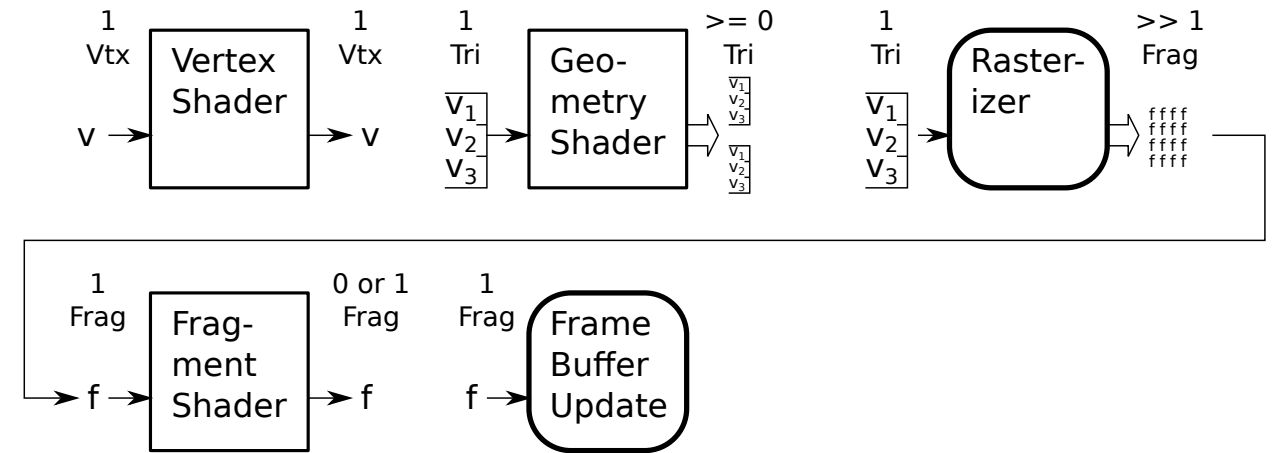
- Compute lighted color of vertex.

- Convert object-space coordinates to clip space.

Current Role:

- Provide data for geometry shader (completely user determined).

- If no geometry shader, output must include clip-space coordinates.



## The Geometry Shader

Input: One primitive.

Output zero or more primitives.

Input primitive type must be compatible with primitive specified by `glBegin` or `glDraw`.

Input data is an array of vertex shader outputs ...

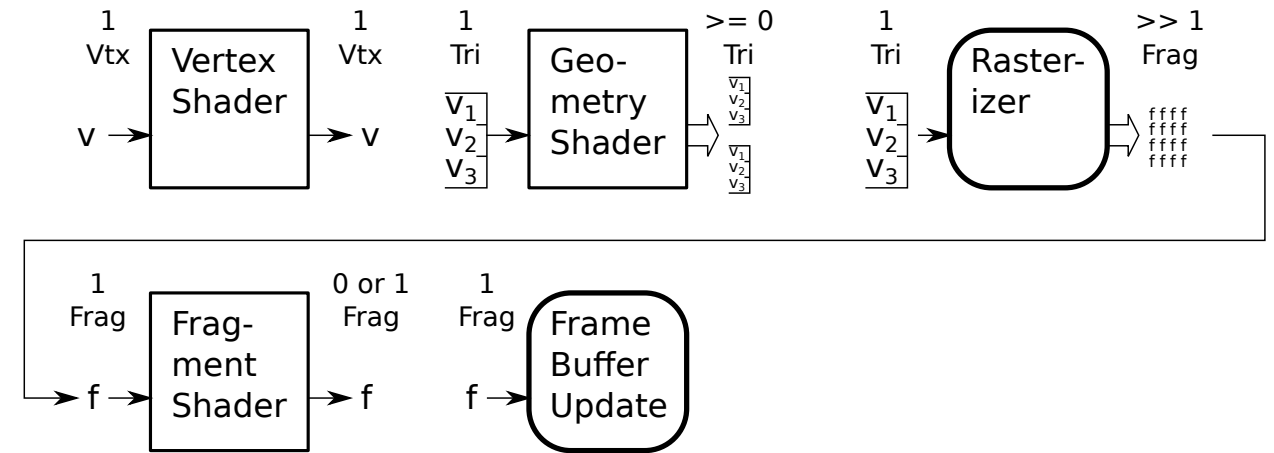
... the size of the array is determined by primitive type.

Output primitive type can be freely chosen.

Current Role:

Must write clip-space coordinates to `gl_Position`.

Also writes whatever other data fragment shader needs.



## Rasterizer

Input: One primitive ...

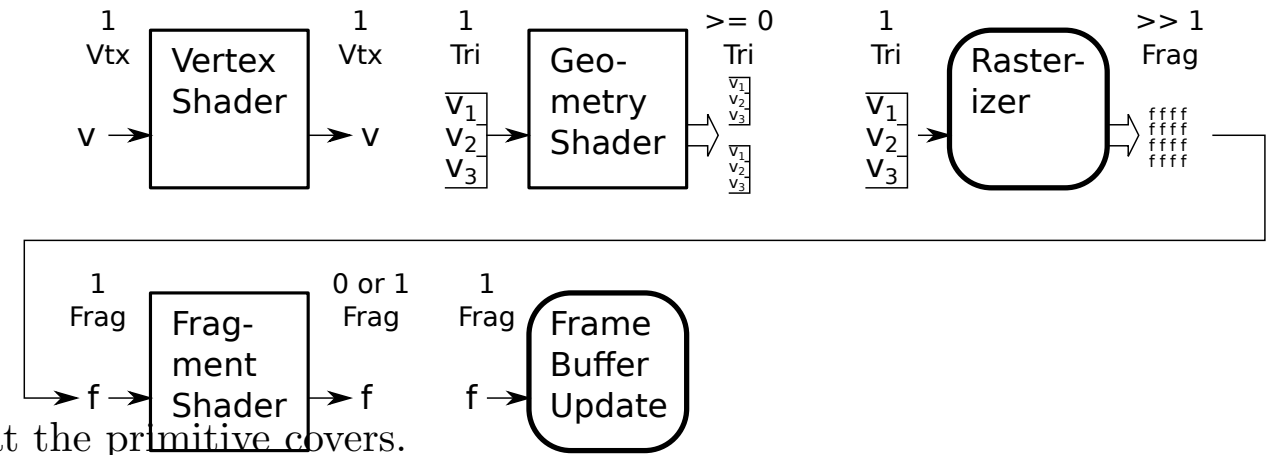
... (type determined by geometry shader).

Output: Zero or more fragments—one fragment for each pixel that the primitive covers.

The data for each fragment is some combination of the data ...

... for each vertex in the input primitive.

The rasterizer cannot be programmed.



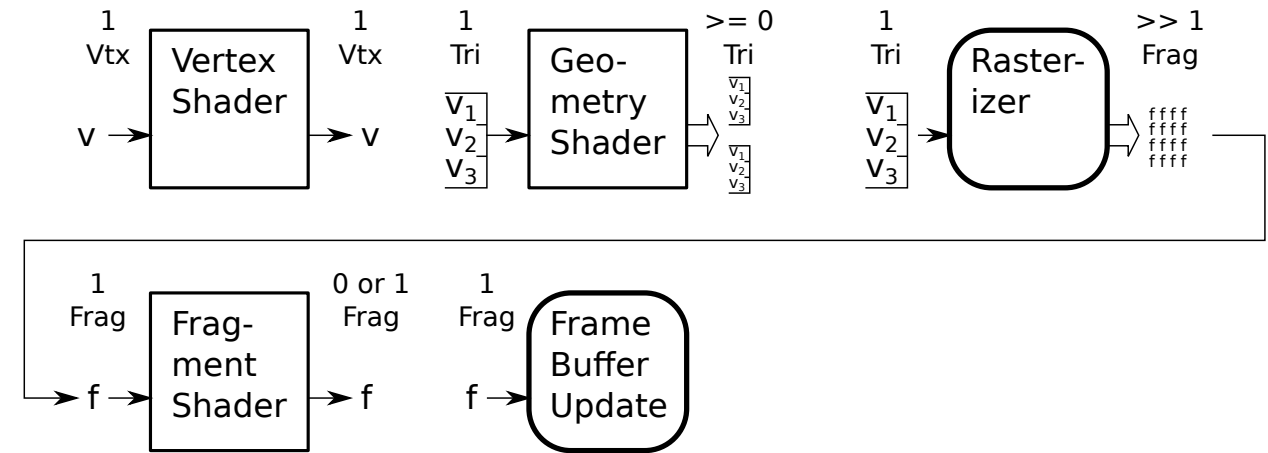
## Fragment Shader

Input: One fragment.

Output: Zero or one fragments.

Typical Role:

Read texels and blend with lighted color.



## Frame Buffer Update

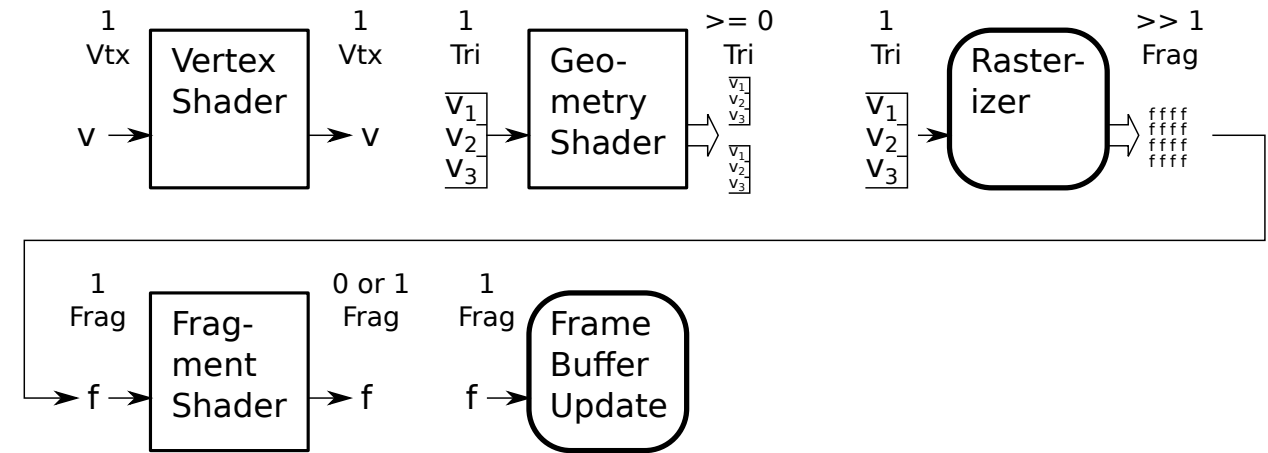
Input one fragment. Output: None.

Typical Role

Applies depth, stencil, and other tests to fragment.

Blends or writes passing fragment to color plane of frame buffer.

Frame buffer update is not programmable.



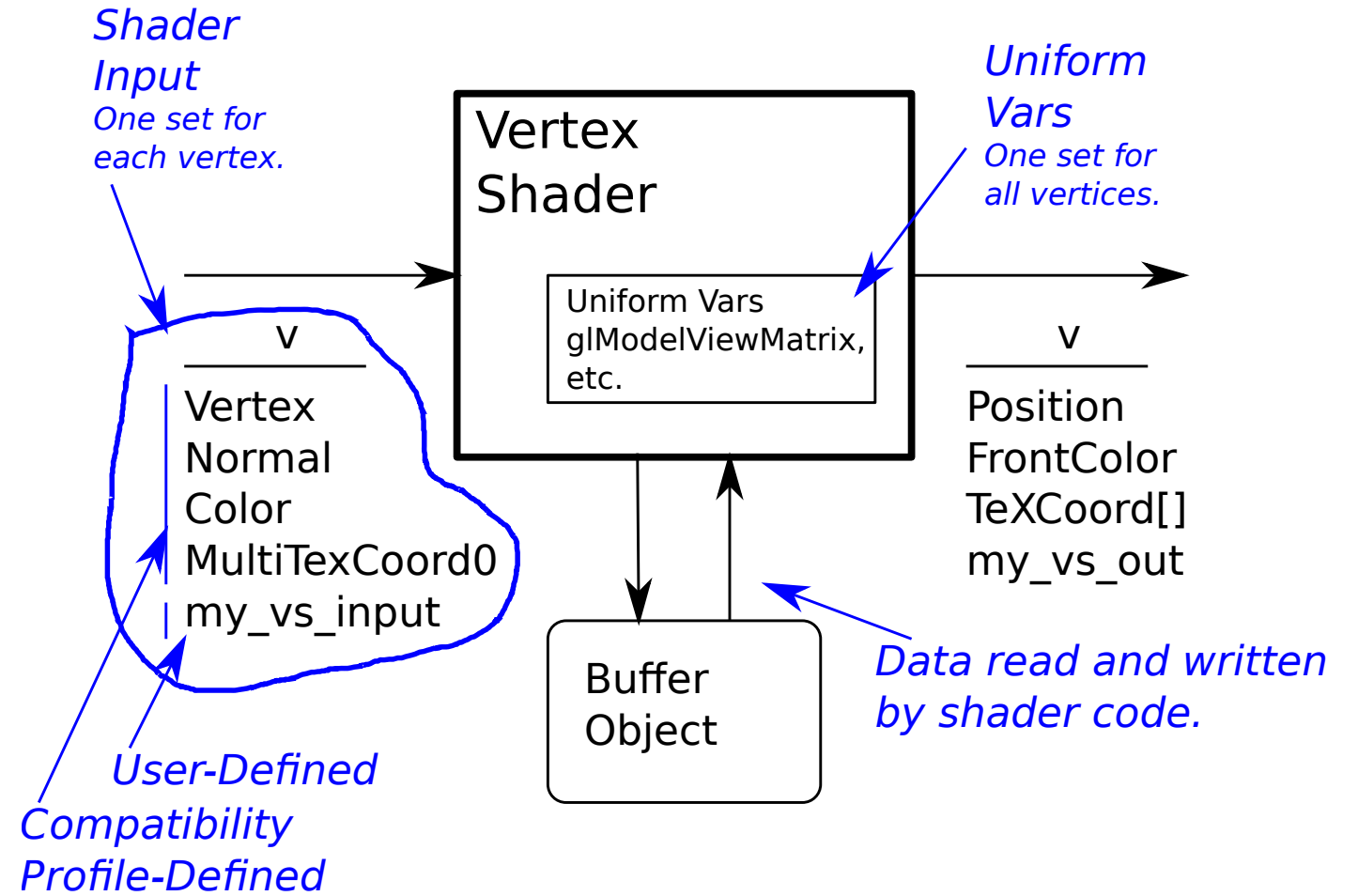
## Data Access by Shaders

Categories

Shader inputs and outputs.

Uniform variables.

Buffer objects.



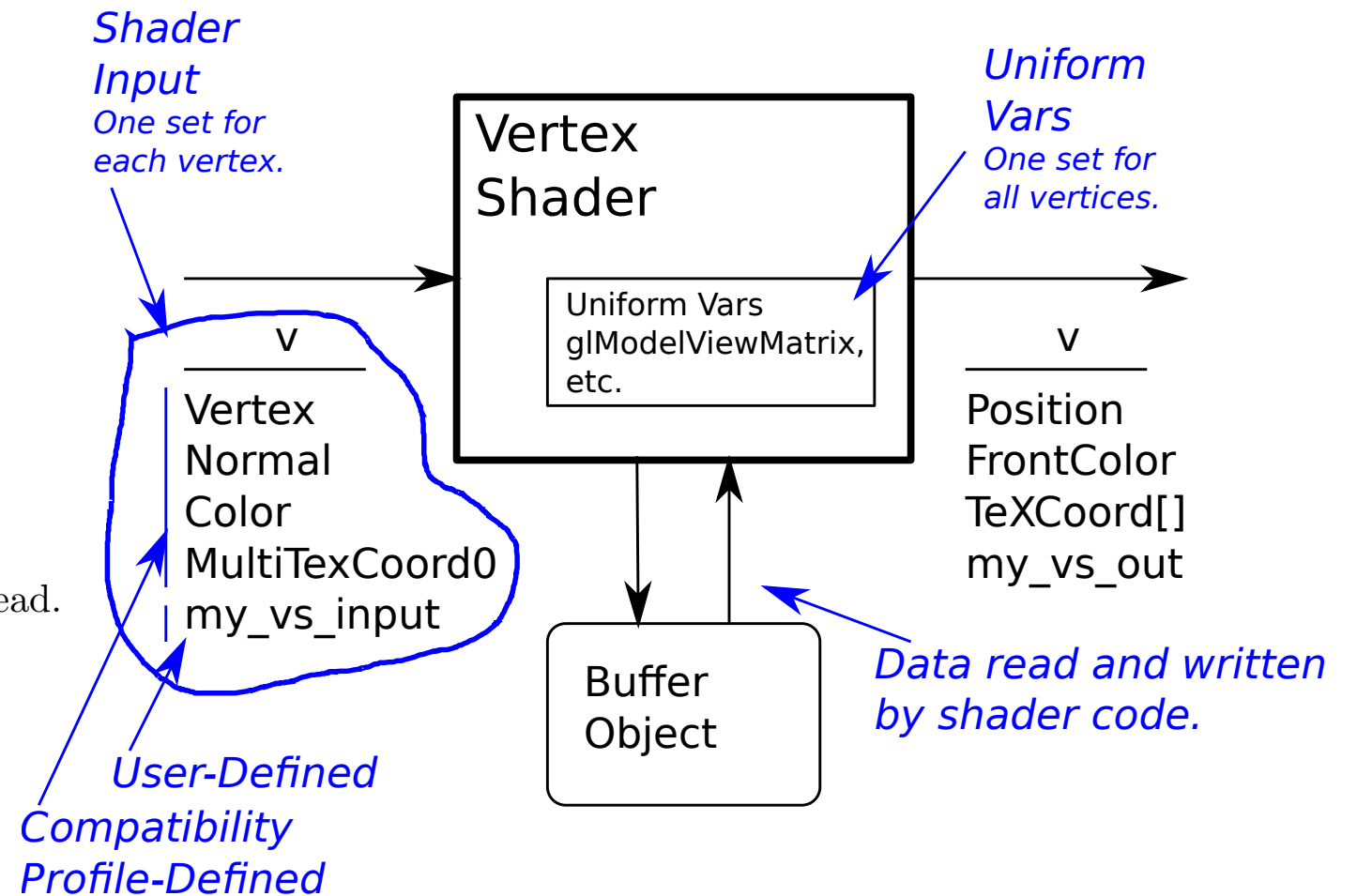


## Shader Inputs and Outputs

One set of data **for each** vertex, primitive, or fragment.

To avoid waste, should not include values common to all.

For example, don't make color a shader input...  
... if all vertices are the same color, use a uniform instead.



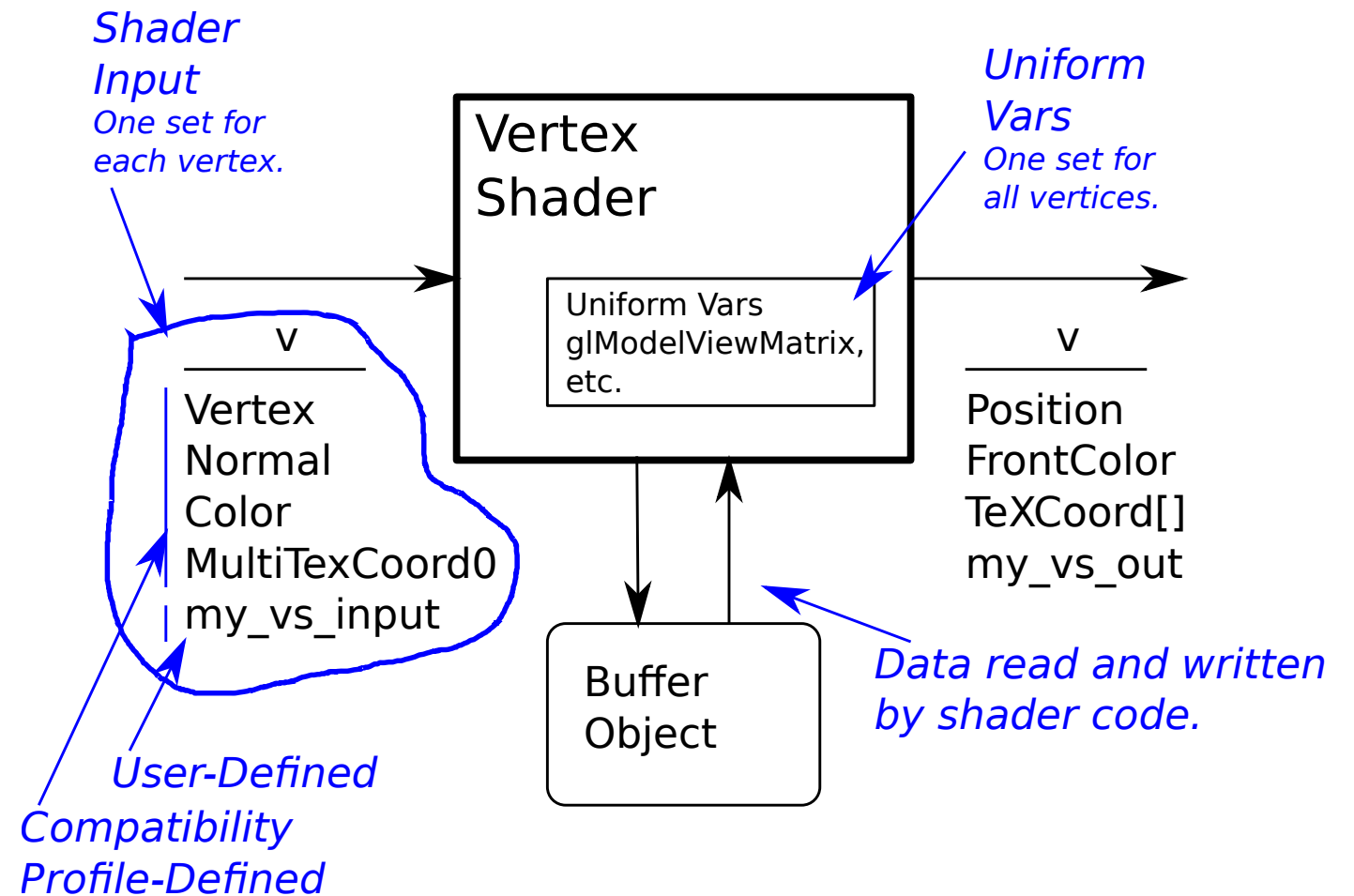
## Uniform Variables

One set of data shared by all.

Read only.

Size is limited, typically 64 kiB.

Usually cached (especially if < 8 kiB accessed).



## Buffer Objects

Shared by all...

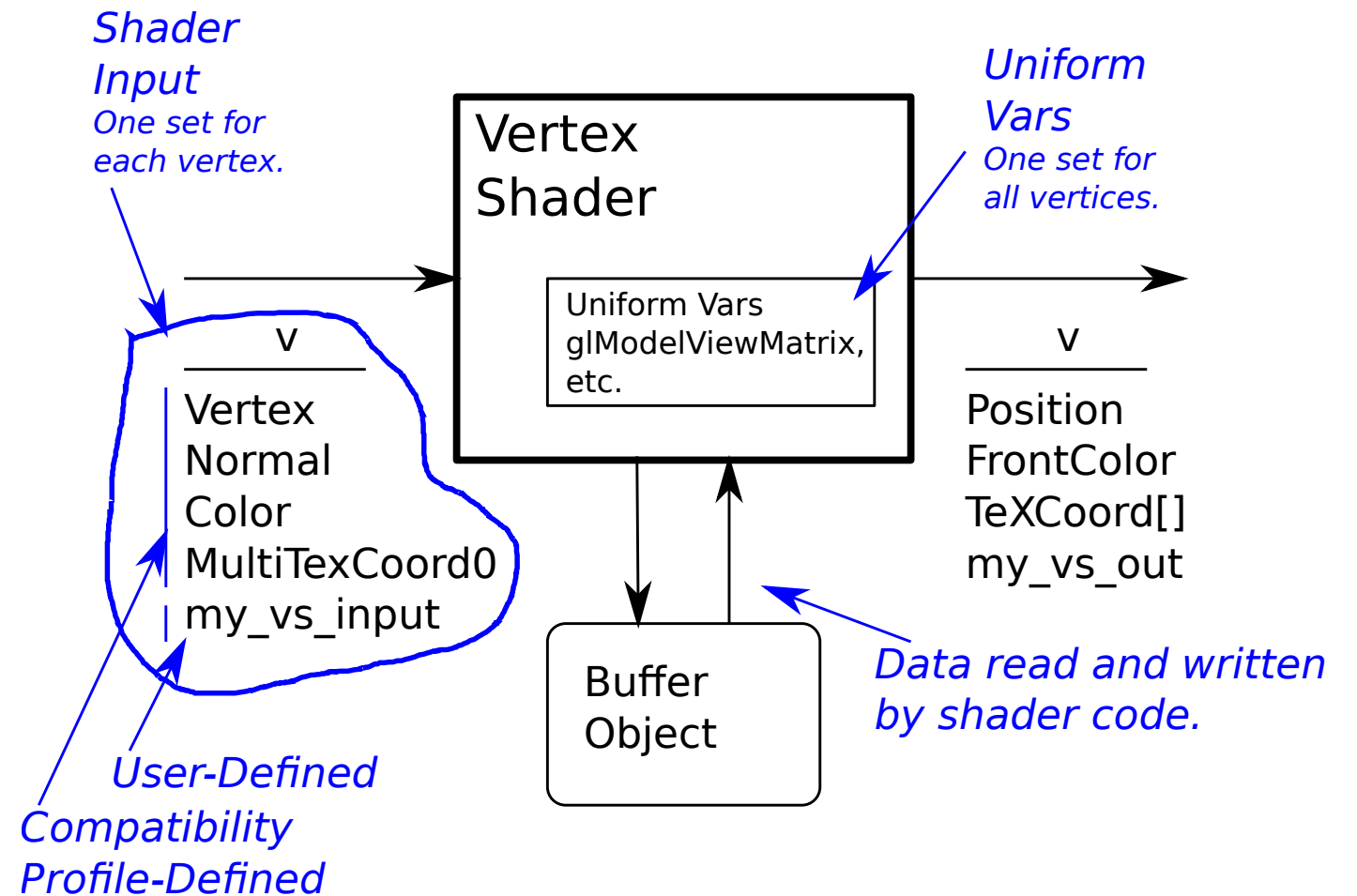
... (vertices, primitives, fragments).

Can be read and written,...

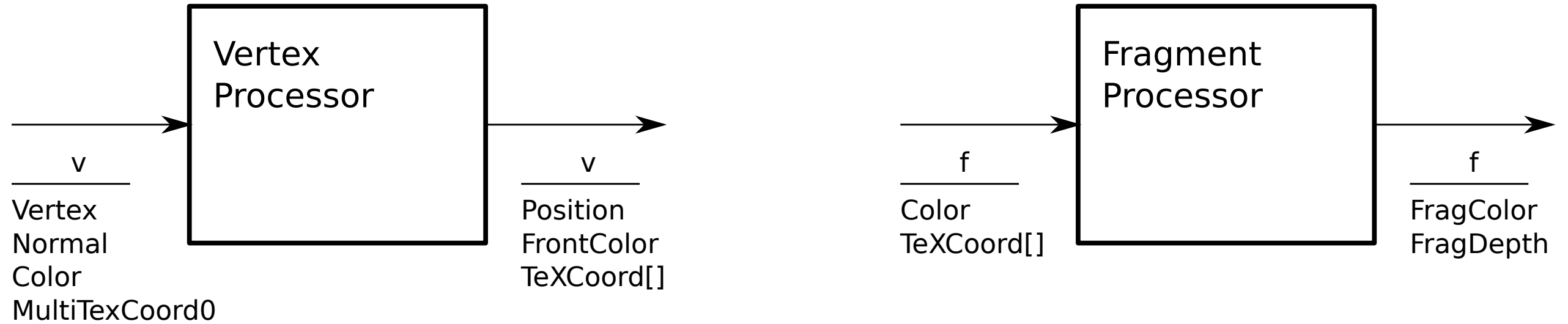
... typically as an array.

Size is large.

Usually not cached.



### OpenGL Shaders



## Major Rendering Pipeline Steps

### Preparation

These activities are performed before invoking pipeline.

CPU specifies transforms, material properties, etc.

Calling, say, `glTranslatef`, helps set up pipeline...  
... but does not start it running or feed it data.

### Feed Data to Pipeline

Data enters in a unit including a vertex and its attributes.

This initiates the steps.

## Vertex Processing Steps (By GPU for each vertex.)

- *Apply modelview transform to vertex.*

Main result is vertex coordinate in eye space.

- *Compute lighted color of vertex.*

Main result is lighted color.

- *Apply projection transform to eye-space vertex.*

Result is vertex coordinate in clip space.

## Primitive Assembly Steps

These steps operate on a primitive (a group of primitives).

- *Primitive Assembly (Group vertices into a primitive).*

Result is, say, a group of 3 describing a triangle.

- *Clip (remove) off-screen parts of primitive.*

Result is fewer and maybe different primitives.

- *Rasterize*

Result is the set of fragments (fb locations) covered by primitive.

## Fragment Processing Steps

These steps operate on a fragment.

- *Fetch texels, filter and blend.*

Result is a frame-buffer ready color.

- *Frame Buffer Update*

If fragment passes depth and other tests, write or blend.



Less Organized Material

## Programmable Units

### **Programmable Unit:**

Part of the pipeline that can be programmed (as defined by some API).

Choice of what is and isn't programmable constrained by:

Need to allow for parallel (multithreaded, SIMD, MIMD) execution.

Simple memory access.

## Major OpenGL Programmable Units

### **Vertex Processor:**

Transform vertex and texture coordinates, compute lighting.

### **Geometry Processor:**

Using a transformed primitive and its neighbors generates new primitives. For example, replace one triangle with many triangles to more closely match a curved surface.

### **Fragment Processor:**

Using interpolated coordinates, read filtered texels and combine with colors.

Less Organized Material

## Languages

### **Shader:**

A programmable part of a GPU. The name “shader” is now misleading but is still in common use.

### **Shader Language:**

An language for programming shaders.

Less Organized Material

High-Level Shader Languages

High-Level Shader Languages

## OpenGL Shader Language

OpenGL standard.

Syntax very similar to C.

Language designed for vertex and fragment shaders.

## Cg

Originated with ATI, adopted in Direct3D.

Syntax very similar to C.

Language designed for stream programs ...

... geometry, vertex, and fragment programs can be in stream form.

Less Organized Material

## OpenGL Shader Language (OGSL)

### OpenGL Shader Language Important Features

C-like

CPP-like preprocessor directives.

Library of useful geometry functions.

Includes vector and matrix types and operators.

## OGSL Data Types Example

### Example

---

```
vec4 vertex_e = gl_ModelViewMatrix * o_point;
vec3 norm_e = gl_NormalMatrix * gl_Normal;
vec4 light_pos = gl_LightSource[1].position;
float phase_light = dot(norm_e, normalize(light_pos - vertex_e).xyz);
float phase_user = dot(norm_e, -vertex_e.xyz);
float phase = sign(phase_light) == sign(phase_user) ? abs(phase_light) : 0.0;
```

---

## OGSL Storage Qualifiers

### Storage Qualifiers

Used in a variable declaration, specifies where data stored.

Below, `in`, `uniform`, `constant`, `out`, and `buffer` are storage qualifiers.

---

```
in vec4 force;           // Input to this shader, different for each primitive.
uniform float x;        // Input to shader, value rarely changed.
const int sides = 5;    // Can never be changed.
out vec2 nudge;         // Output of this shader (input to some other).
out vec2 nudge;         // Output of this shader (input to some other).
// Buffer object accessed from shader code as helix_coord ..
// .. and from CPU and binding point 7.
layout ( binding = 7 ) buffer Helix_Coord { vec4 helix_coord[]; };
```

---

## Storage Qualifier Types

### **uniform:**

Read-only by shader. Written by client, change is time consuming.

Typical use: transformation matrices.

### **in:**

Input to shader. Read-only by shader that made the **in** declaration. Value is set either by client (using `glVertexAttrib` and friends) or by a prior stage shader (by writing an **out** variable).

Typical uses: vertex material properties (color), normal.

### **out:**

Output of shader. Value is written by shader in which **out** declaration appears and read by shader in subsequent stage.

### **buffer:**

Variable is a buffer object. Value can be read or written by shader. Variable name in declaration is used by shader code, binding point in declaration is used by CPU code.



## Interpolation Qualifiers

Used for fragment shader inputs.

Specify how value should be interpolated.

### **flat:**

No interpolation.

### **smooth:**

Linear interpolation in object space. (Perspective-correct interpolation).

### **noperspective:**

Linear interpolation in clip space. (Faster, but an approximation.)

## Deprecated Storage Qualifiers.

These were used in earlier versions of OGSL.

They have been replaced by `in` and `out`.

### **attribute:**

Deprecated. Like an `in` but only can be used for vertex shader.

### **varying:**

Deprecated. When used in a vertex shader is the same as `out`, when used in a fragment shader is the same as `in`.

## Storage Qualifier Example

---

```
// For vertex and fragment shaders:
```

```
uniform float gs_constant;  
uniform vec2 ball_size;  
layout ( location = 4 ) uniform vec3 gravity_force;
```

```
// Vertex Shader Only (Based on what our shader needs.)
```

```
in float step_last_time;  
in vec4 position_left, position_right, position_above, position_below;  
layout { location = 5 } in vec3 ball_speed;
```

```
out vec4 out_position;  
out vec3 out_velocity;
```

```
// Fragment Shader Only (Based on what our shader needs.)
```

```
//  
in vec4 out_position;  
in vec3 out_velocity;
```

---

Less Organized Material

## OGSL Functions

### Function Parameters

OpenGL Shading Language 4.6 Section 6.1.1

Call by value.

Parameter Qualifiers:

in (default)

out

inout

### Built In Functions

See OpenGL Shading Language 4.6 Section 8

## OpenGL Shading Language Use

Steps for adding a typical shader to existing OpenGL code:

Define what the shader is supposed to do.

Identify appropriate programmable units (vertex, geometry, fragment, etc).

Identify data that shaders will use.

If data from client (CPU) determine whether attribute or uniform.

For attributes and uniforms, determine if pre-defined or user-defined.

Write shader code.

In CPU code follow steps for installing shader. (*E.g.*, use `pShader`).

Get names of any new uniforms and attributes.

As necessary, initialize uniforms and attributes.

Turn shader on and off as necessary.

## Example—Phong Shader

### Phong Shader:

A lighting model in which the lighted color is computed at each fragment. (Otherwise the lighted color is computed at each vertex of a primitive and those lighted colors are interpolated across the fragments.)

### Phong Shader Steps

- *Define what shader does.*  
Computes lighting at fragment using interpolated normal...
- *Identify appropriate units.*  
For computing lighting: fragment shader.  
  
For passing along normal and color info, vertex shader.
- *Identify data that shaders use.*  
VS: Lighting data, normal. (All pre-defined.)  
  
FS: Normal (interpolated), eye-space vertex coordinates. User def.

## OpenGL Calls, from Initialization to Use (See OGL 4.6 Chapter 7)

Create Program Object (Once)

```
pobject = glCreateProgram();
```

For Each Shader (Vertex, Geometry, Fragment, etc.):

Create Shader Object

```
subject = glCreateShader(GL_VERTEX_SHADER)
```

Provide Source Code to Shader Object and Compile

```
glShaderSource(subject, 1, &shader_text_lines, NULL);
```

```
glCompileShader(subject);
```

Attach

```
glAttachShader(pobject, subject);
```

Link (Once)

```
glLinkProgram(pobject);
```

Use (Many Times, *e.g.*, once per frame.)

```
glUseProgram(pobject);
```

## OGSL Use

### Obtaining and Using Variable References

At run time variables identified by number.

At Initialization get **location** (index) of attributes and uniforms:

```
vsal_pos_left = glGetAttribLocation(pobject,"position_left");  
sun_ball_size = glGetUniformLocation(pobject,"ball_size");
```

During Render (Infrequently) Specify Uniform Value (Using location)

```
glUniform2f(sun_ball_size,ball_size,ball_size_sq);  
glUniform3fv(4,gravity_force);
```

During Render (Per Vertex Okay) Specify Attribute Value (Using location)

```
glVertexAttrib4fv(vsal_pos_left,pos_left);  
glVertexAttrib3fv(5,ball_velocity);
```

Done before each glVertex.

Same options as vertex, such as client and buffer object arrays.