Programmable GPU & GLSL

Some History

- Cook and Perlin first to develop languages for performing shading calculations
- Perlin computed noise functions procedurally; introduced control constructs
- Cook developed idea of *shade trees* @ Lucasfilm
- These ideas led to development of Renderman at Pixar (Hanrahan *et al*) in 1988.
- Renderman is STILL shader language of choice for high quality rendering !
- Languages intended for offline rendering; no interactivity, but high quality.

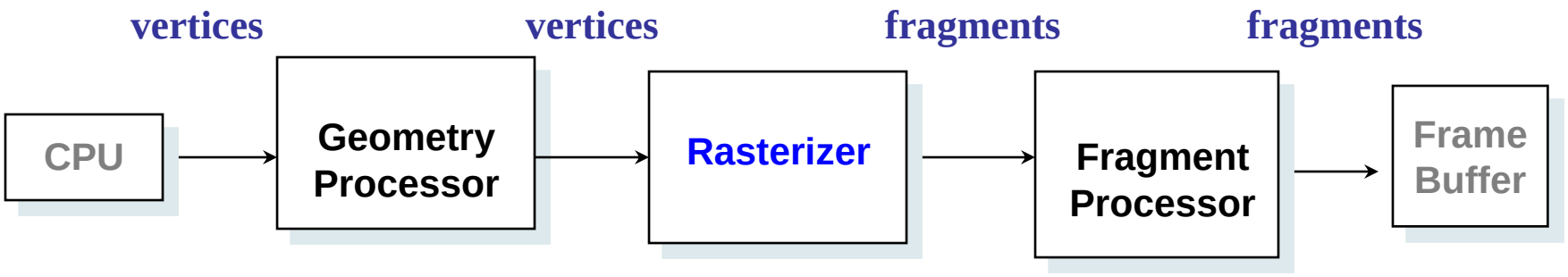
Some History

- After RenderMan, independent efforts to develop high level shading languages at SGI (ISL), Stanford (RTSL).
- ISL targeted fixed-function pipeline and SGI cards : goal was to map a RenderMan-like language to OpenGL
- RTSL took similar approach with programmable pipeline and PC cards
- RTSL morphed into Cg.

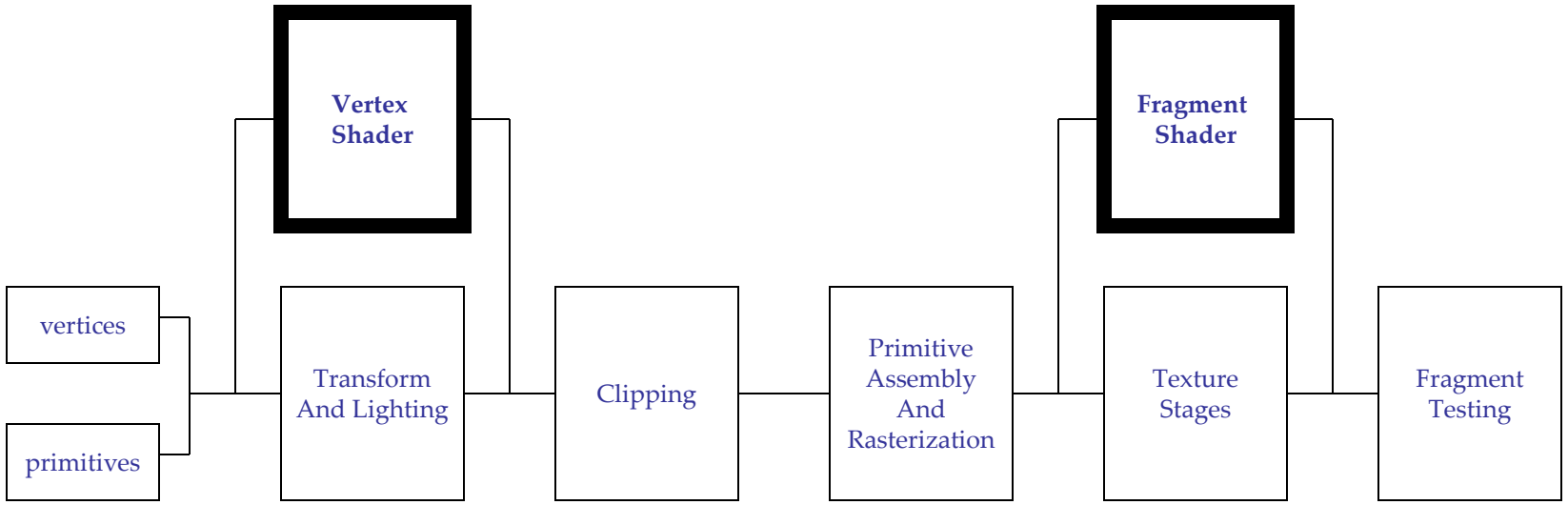
Some History

- Cg was pushed by NVIDIA as a platform-neutral, card-neutral programming environment.
- In practice, Cg tends to work better on NVIDIA cards (better demos, special features etc).
- ATI made brief attempt at competition with Ashli/RenderMonkey.
- HLSL was pushed by Microsoft as a DirectX-specific alternative.
- In general, HLSL has better integration with the DirectX framework, unlike Cg with OpenGL/DirectX.
- GLSL is a part of OpenGL 2.0

Black Box View

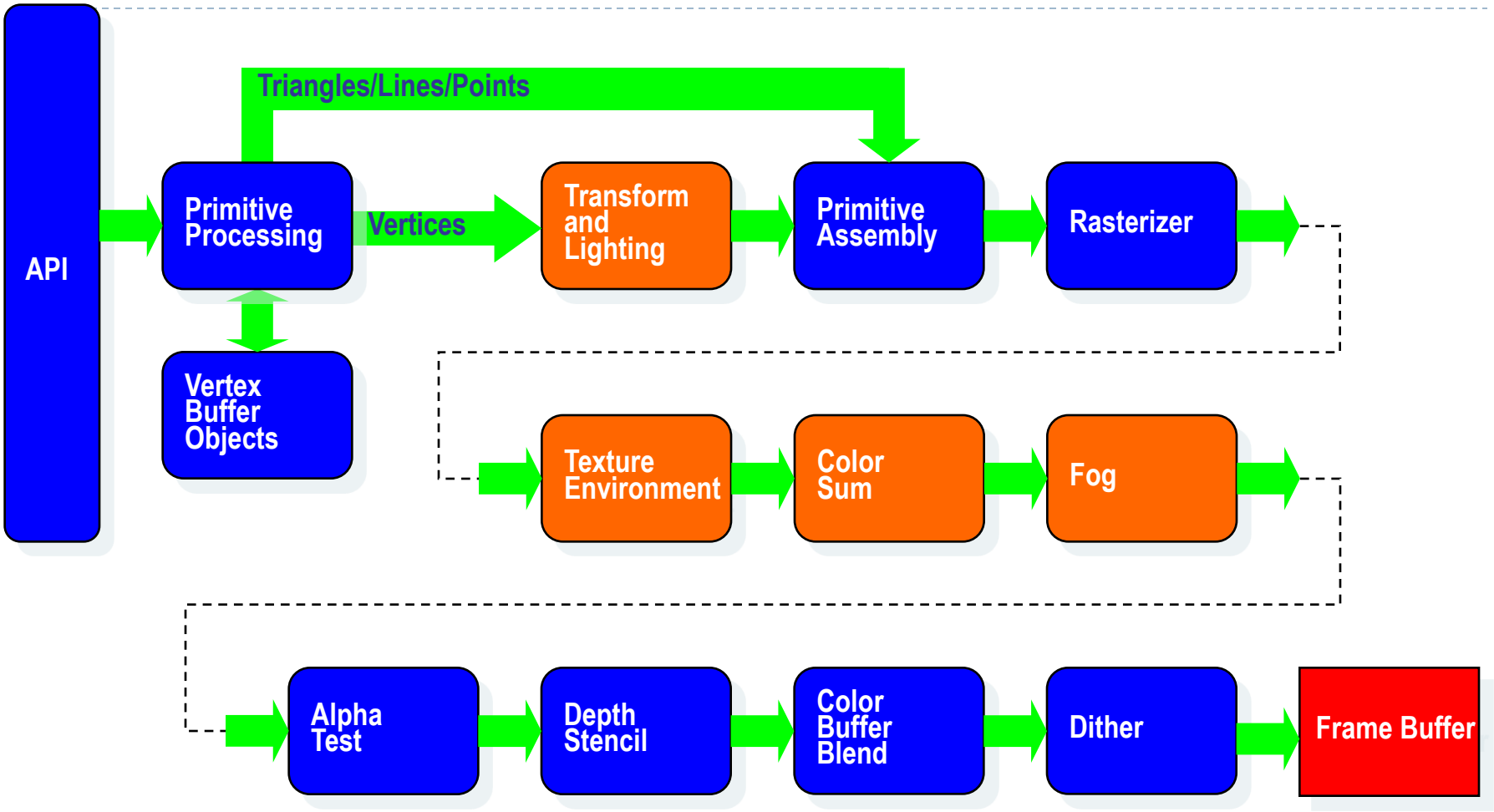


The Programmable GPU

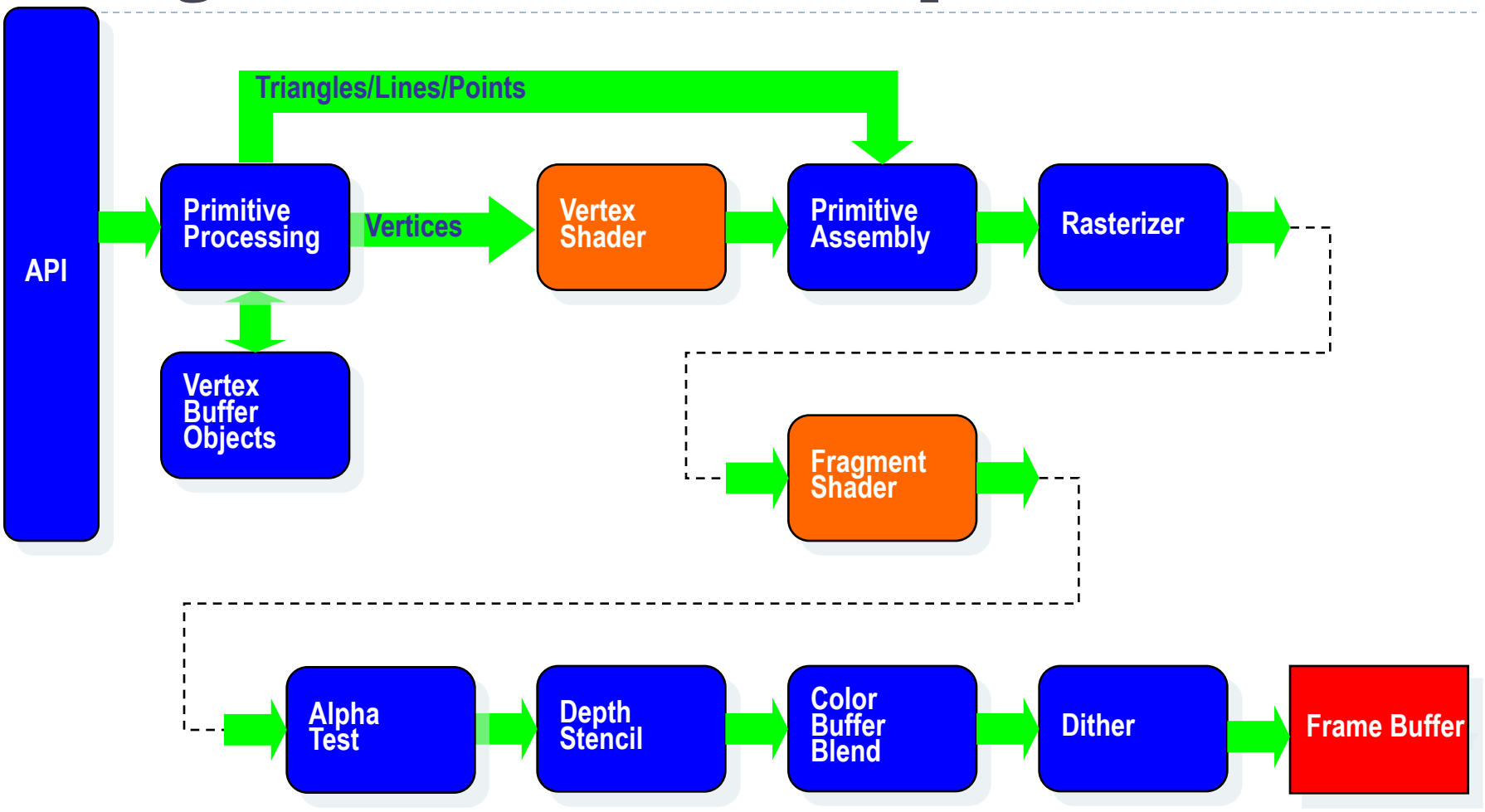


- **GPU** = **vertex shader** (vertex program) + **fragment shader** (fragment program, pixel program)
- Vertex shader replaces per-vertex transform & lighting
- Fragment shader replaces texture stages
- Fragment testing after the fragment shader
- Flexibility to do framebuffer pixel blending

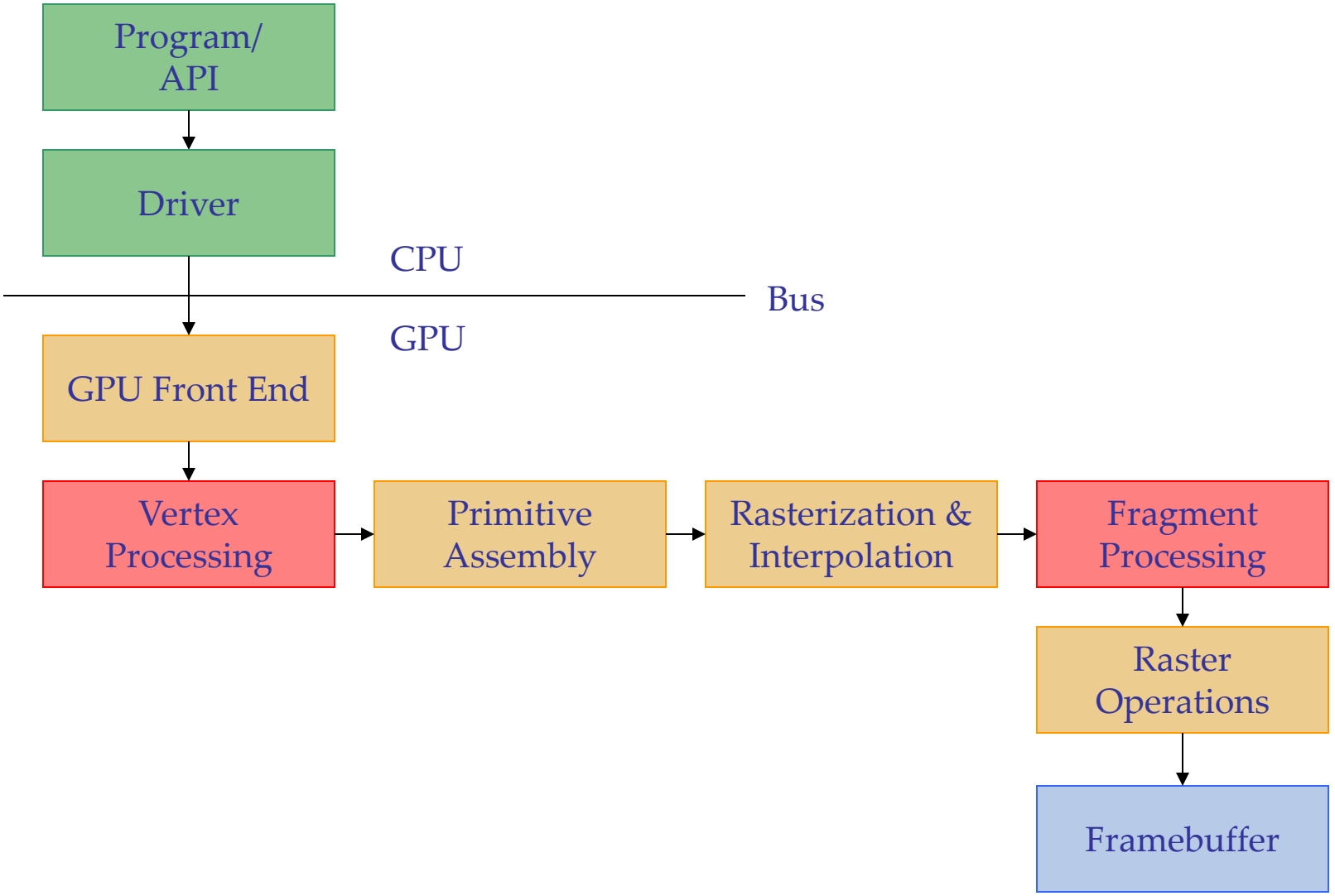
Fixed Functionality Pipeline



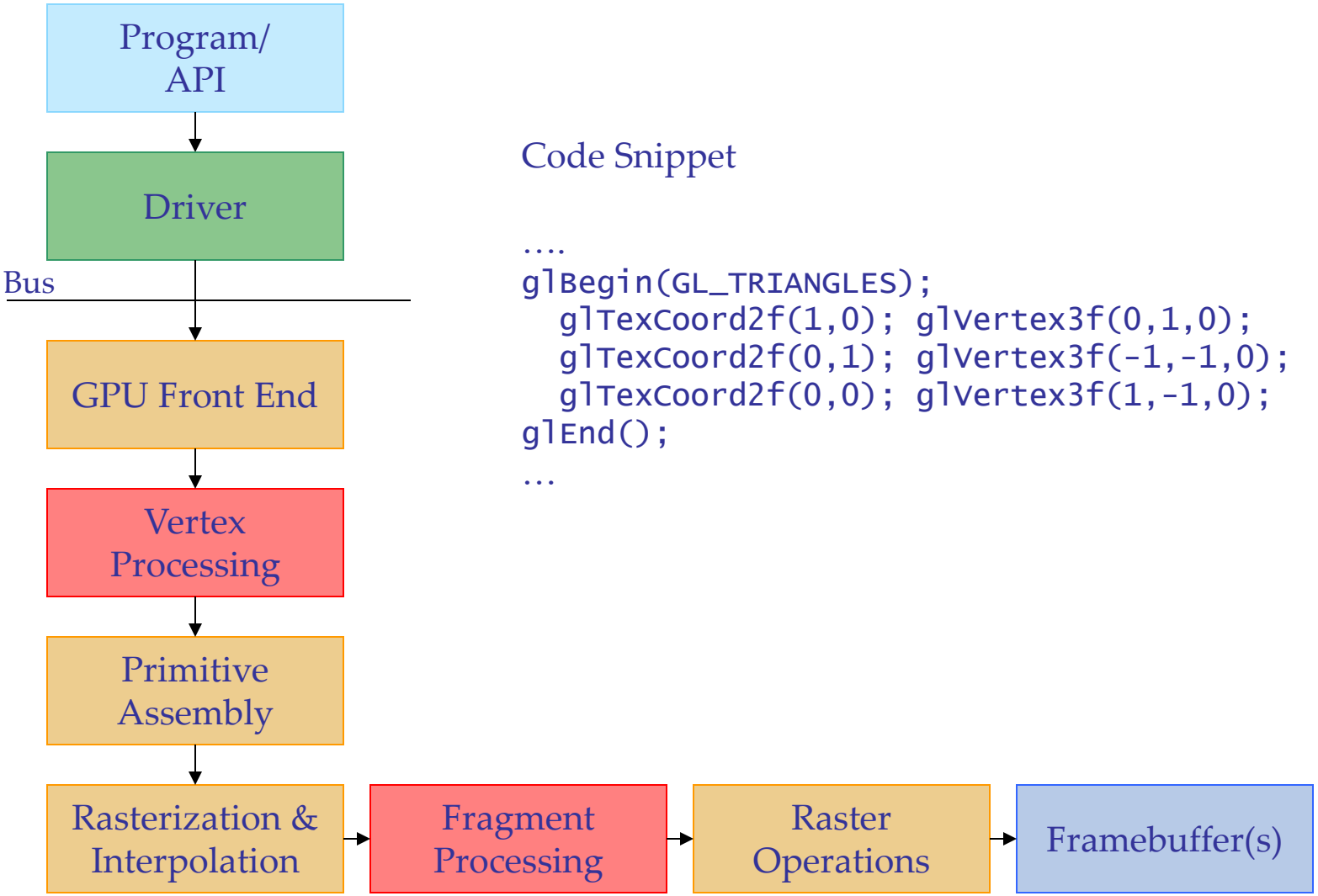
Programmable Shader Pipeline



GPU pipeline



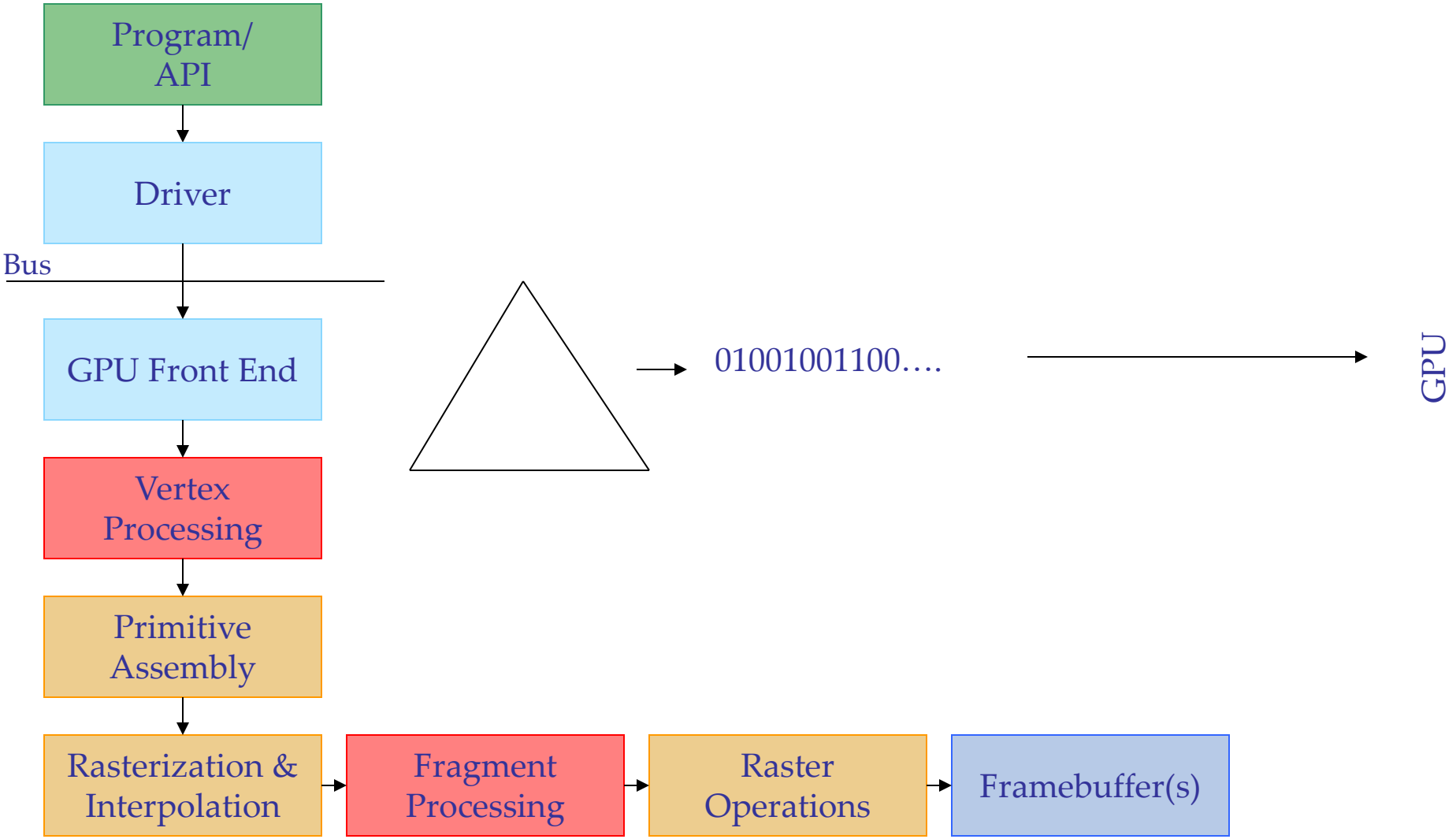
example



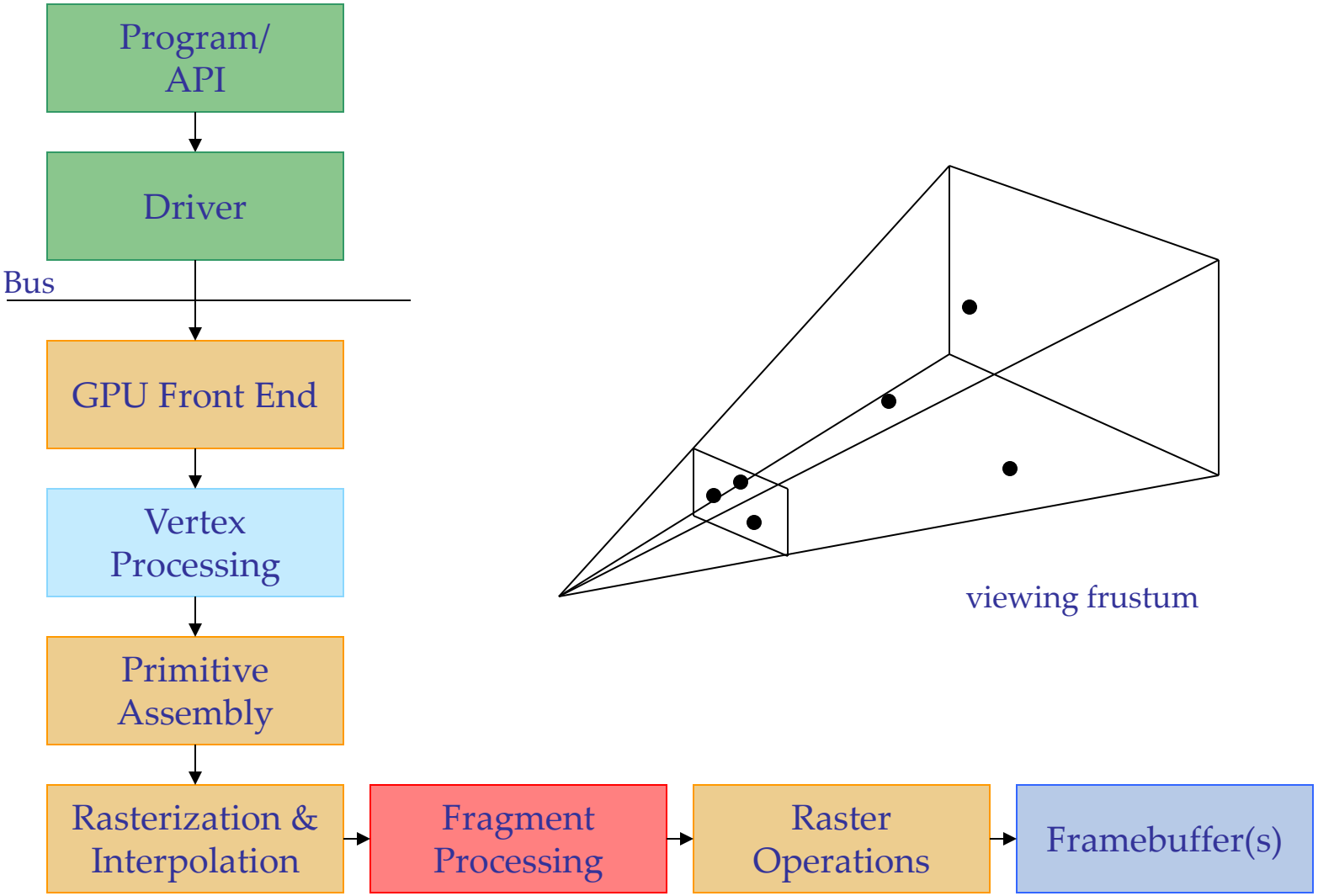
Code Snippet

```
....
glBegin(GL_TRIANGLES);
  glTexCoord2f(1,0); glVertex3f(0,1,0);
  glTexCoord2f(0,1); glVertex3f(-1,-1,0);
  glTexCoord2f(0,0); glVertex3f(1,-1,0);
glEnd();
...
```

example

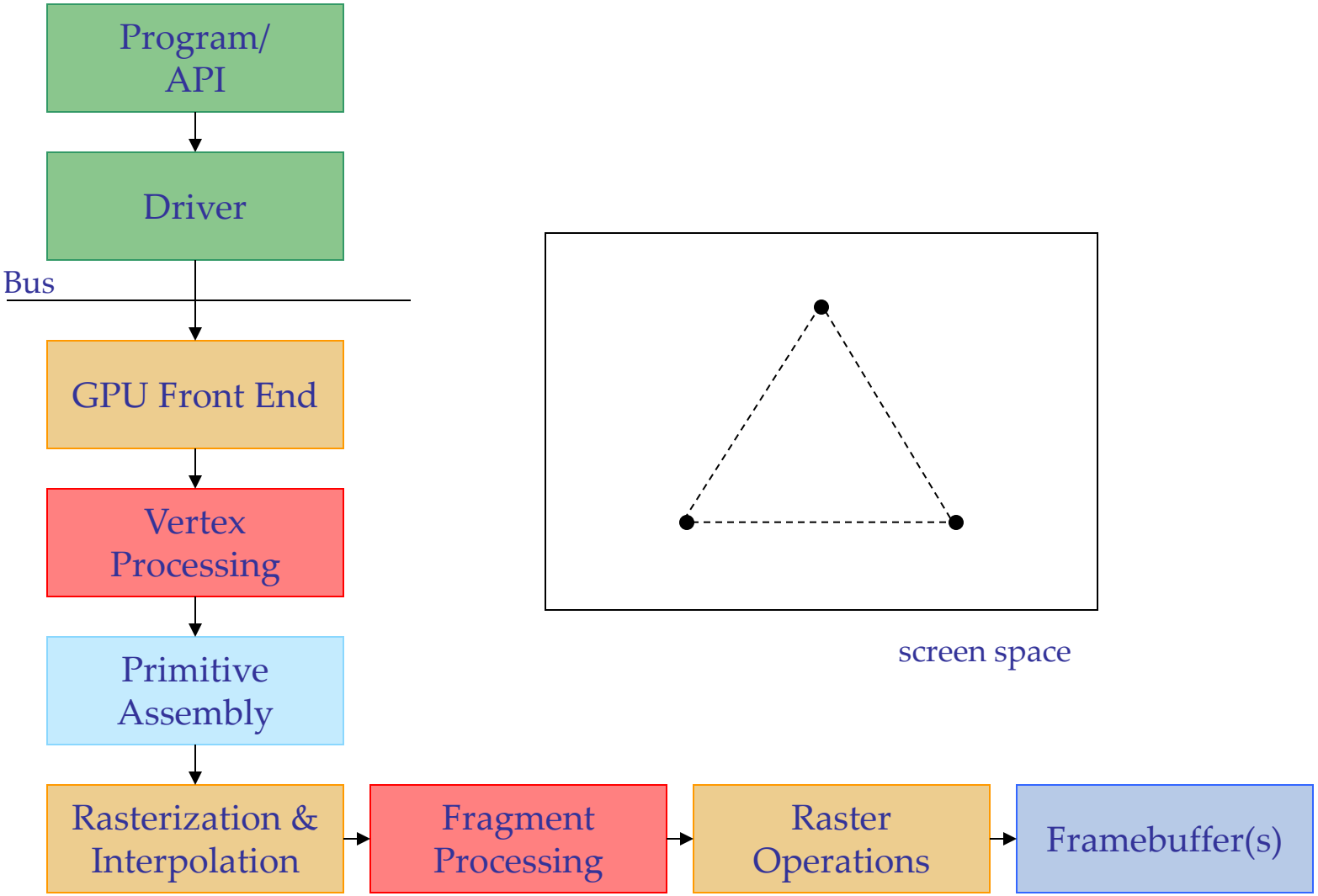

Program/
API
Driver
Bus
GPU Front End
Vertex
Processing
Primitive
Assembly
Rasterization &
Interpolation
Fragment
Processing
Raster
Operations
Framebuffer(s)
01001001100….
GPU

example

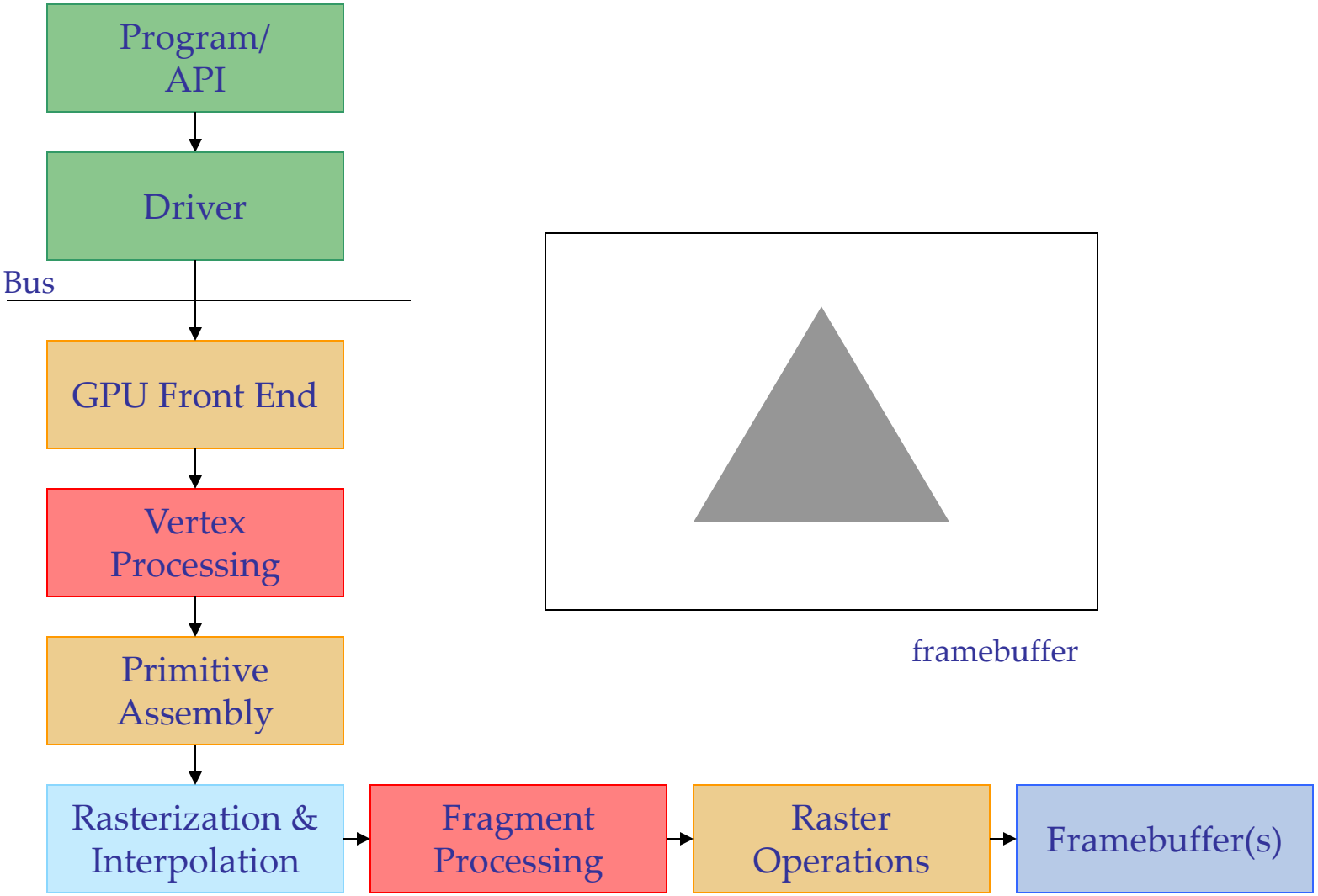


viewing frustum

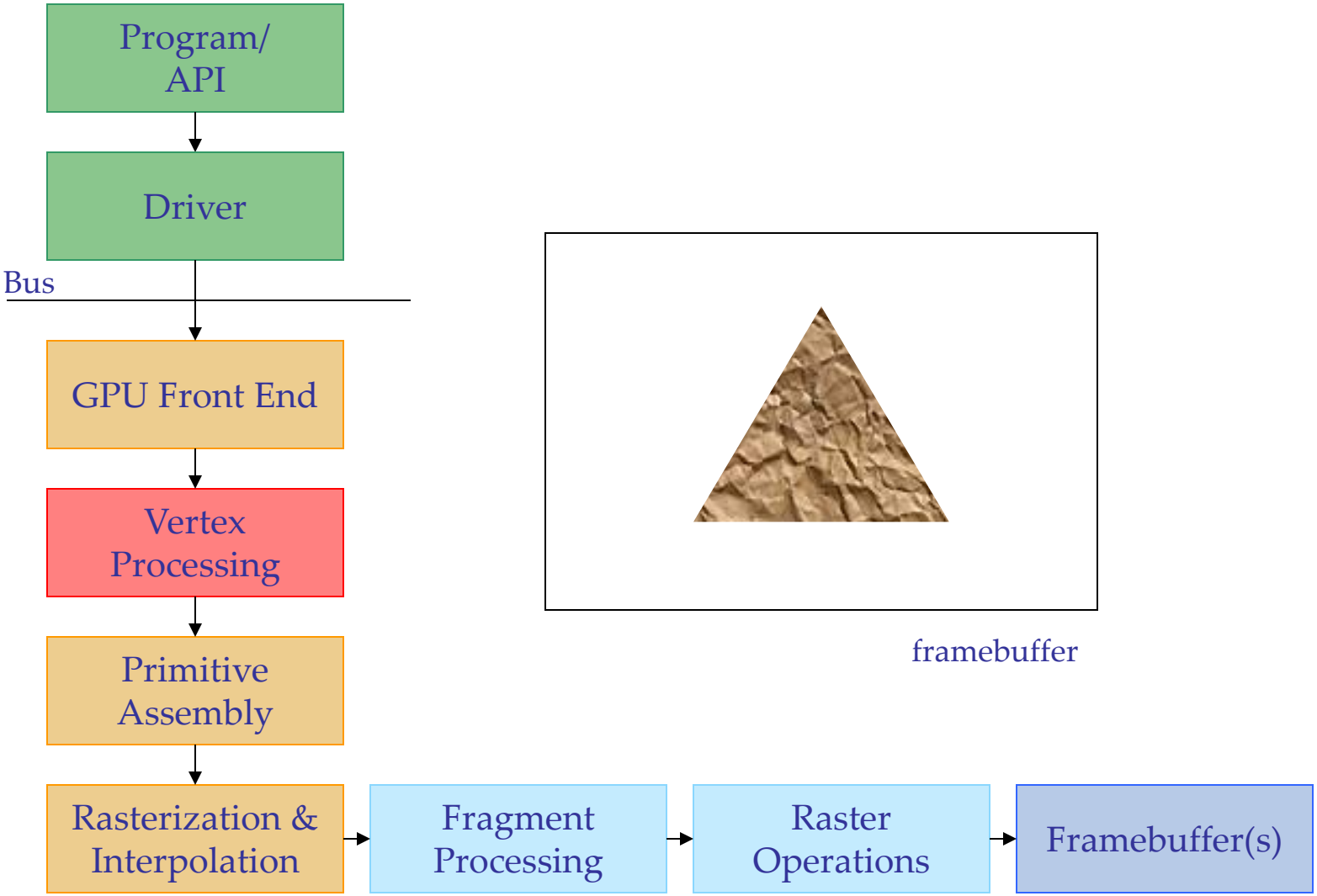
example


Program/
API
Driver
Bus
GPU Front End
Vertex
Processing
Primitive
Assembly
Rasterization &
Interpolation
Fragment
Processing
Raster
Operations
Framebuffer(s)
screen space

example


Program/
API
Driver
Bus
GPU Front End
Vertex
Processing
Primitive
Assembly
Rasterization &
Interpolation
Fragment
Processing
Raster
Operations
Framebuffer(s)
framebuffer

example


Program/
API
Driver
Bus
GPU Front End
Vertex
Processing
Primitive
Assembly
Rasterization &
Interpolation
Fragment
Processing
Raster
Operations
Framebuffer(s)
framebuffer

Per-Vertex Operations

(Geometric and Color calculation)

- **Geometric data**: set of vertices + type
  - ✓ Can come from program, evaluator, display list
  - ✓ **type:** point, line, polygon(triangle)
  - ✓ Vertex data can be
    - (x,y,z,w) coordinates of a vertex (glVertex)
    - Normal vector
    - Texture Coordinates
    - RGBA color
    - Other data: color indices, edge flags
    - Additional user-defined data in GLSL

Geometric Calculations

- Vertex locations are transformed by the model-view matrix into eye coordinates
- Normals must be transformed with **the inverse transpose** of the model-view matrix so that $v \cdot n = v' \cdot n'$ in both spaces

  Assumes there is no scaling($M^{-1} = M^t$)
- and then **into the projection space**

Lighting(Color) Calculation

- Done on **a per-vertex basis Phong model**

$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{v} \cdot \mathbf{r})^a + k_a I_a$$

- Phong model requires computation of **r** and **v** at every vertex

Primitive Assembly

- Vertices are next assembled into primitives that are clipped. The **potentially visible primitives** that are not clipped out are rasterized, generating fragments.
  - ✓ Polygons
  - ✓ Line Segments
  - ✓ Points
- Transformation by projection matrix
- Perspective Division
- Clipping
- Viewport mapping

Rasterization

- Geometric entities are rasterized into **fragments**
- Each fragment corresponds to a point on an integer grid: a displayed pixel
- Hence each fragment is ***a potential pixel***
- Each fragment has
  - ✓ A color
  - ✓ Possibly a depth value
  - ✓ Texture coordinates

Fragment Operations

- Texture generation
- Fog
- Anti-aliasing
- Scissoring
- Alpha test
- Blending
- Dithering
- Logical Operation
- Masking

Vertex Processor(H/W module)

- Takes in **vertices**
  - ✓ Position attribute
  - ✓ Possibly color
  - ✓ OpenGL state
- Produces
  - ✓ Position in clip coordinates

    (clipping space = projection space)
  - ✓ Vertex color

Fragment Processor(H/W module)

- Takes in **output of rasterizer (fragments)**
  - ✓ Vertex values have been interpolated over primitive by rasterizer
- Outputs a fragment
  - ✓ Color
  - ✓ Texture
- Fragments still go through fragment tests
  - ✓ Hidden-surface removal
  - ✓ alpha

Programmable Shaders

- Replace fixed function vertex and fragment processing by programmable processors called **shaders**
- Can replace either or both
- If we use a programmable shader, we must do *all* required functions of the fixed function processor

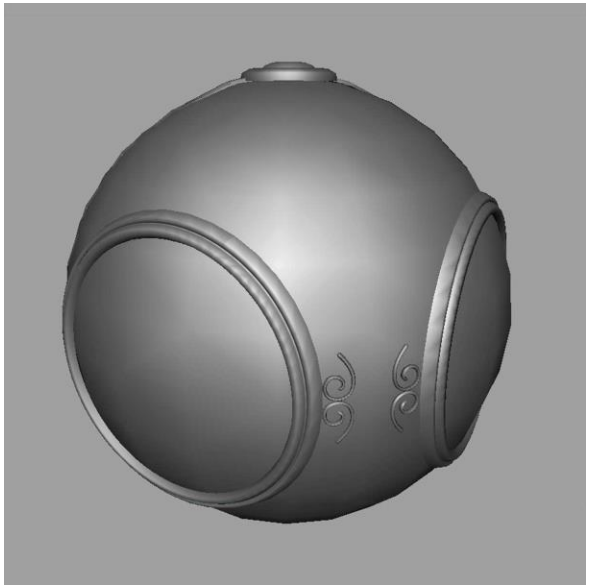
- GLSL

Vertex Shader Applications

- Moving vertices
  - ✓ Morphing
  - ✓ Wave motion
- Lighting
  - ✓ More realistic models
  - ✓ Cartoon shaders

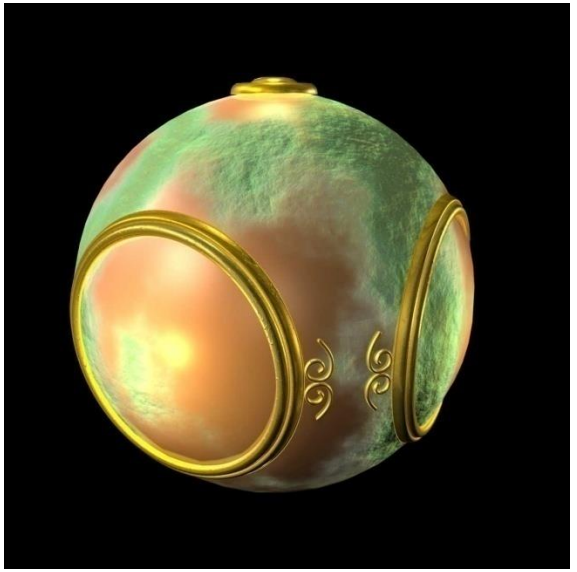
Fragment Shader Applications

Texture mapping



Smooth shading

Environment mapping



Bump mapping

Lighting Calculations

per vertex lighting

per fragment lighting

Shaders in Use

- First programmable shaders were programmed in an assembly-like manner
- **OpenGL extensions** added for vertex and fragment shaders
- **Cg (C for graphics)** C-like language for programming shaders
  - ✓ Works with both OpenGL and DirectX
  - ✓ Interface to OpenGL complex
- OpenGL Shading Language **(GLSL)**

GLSL

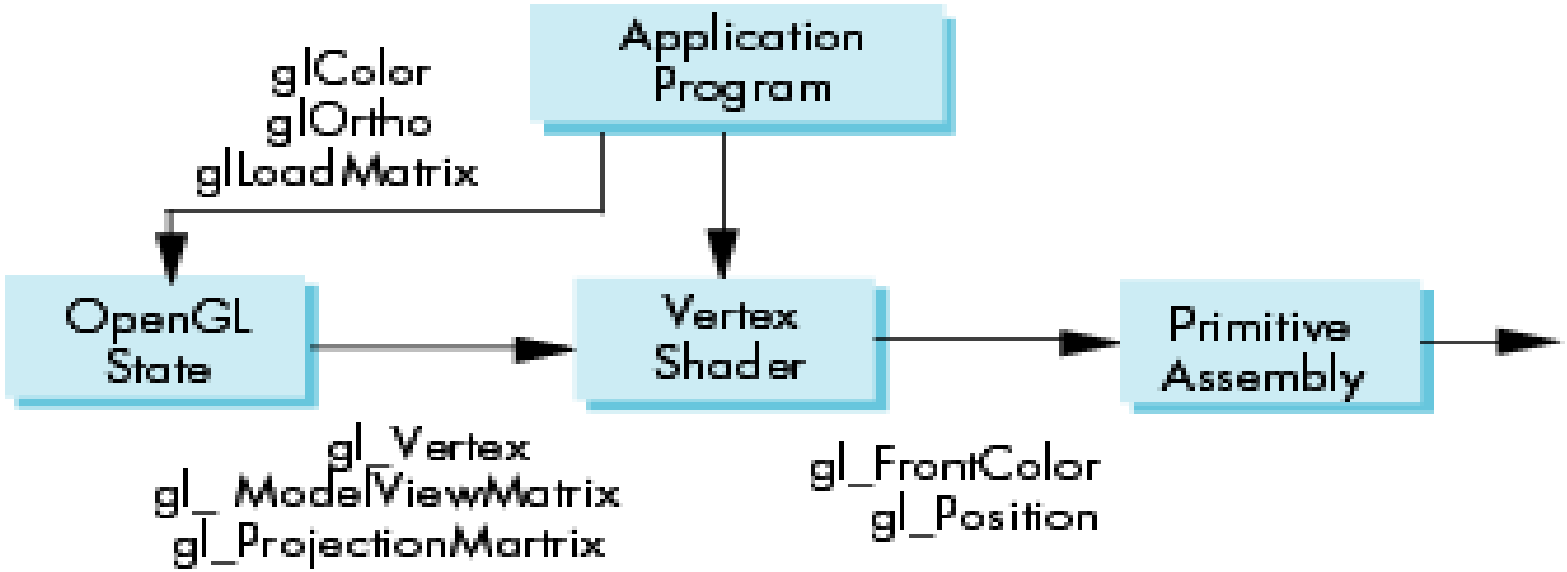
- OpenGL Shading Language
- **Part of OpenGL 2.0**
- High level C-like language
- New data types
  - ✓ Matrices
  - ✓ Vectors
  - ✓ Samplers
- OpenGL state available through built-in variables

Simple Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMartrix*gl_Vertex;

    gl_FrontColor = red;
}
```

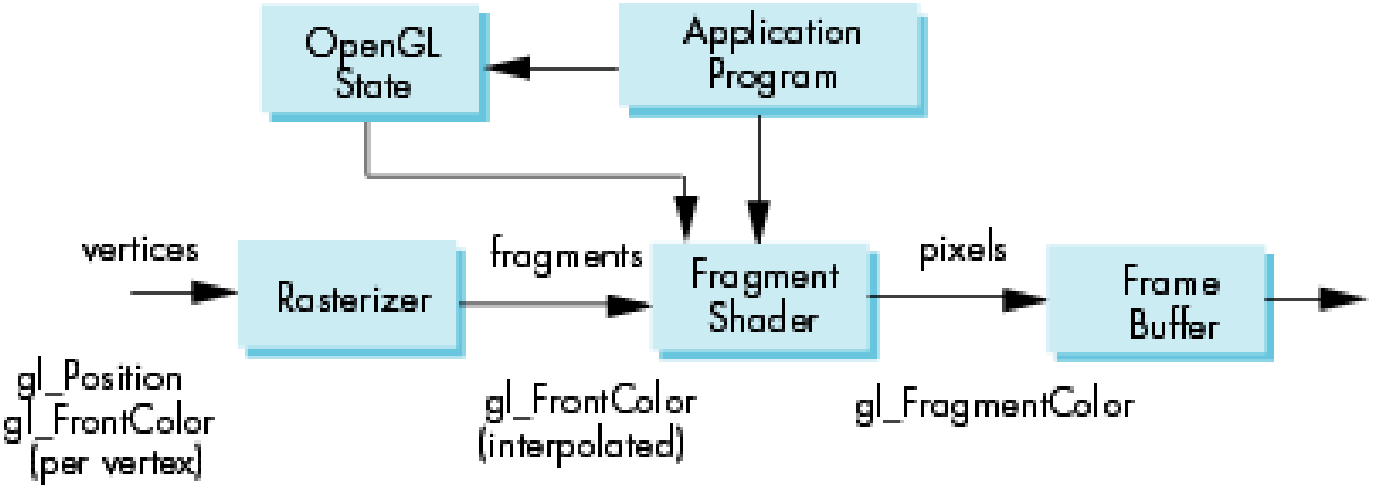
Execution Model



Simple Fragment Program

```
void main(void)
{
  gl_FragColor = gl_FrontColor;
}
```

Execution Model



Data Types(1)

- Atomic types: int, float, bool
- Vectors:
  - ✓ float vec2, vec 3, vec4
  - ✓ Also int (ivec2, 3, 4) and boolean (bvec2, 3, 4)
  - ※ indexing : [](array), xyzw(coordinates), rgba(color), stpq(texture coordinates)
- Matrices: mat2, mat3, mat4(individually, square matrices)
  - ✓ Stored by columns
  - ✓ Standard referencing m[row][column]
- C++ style constructors
  - ✓ vec3 a =vec3(1.0, 2.0, 3.0)
  - ✓ vec2 b = vec2(a)

Data Types(2)

void – used for functions that do not return a value
bool – conditional type, values may be either true or false
int – a signed integer

float – a floating point number
vec2 – a 2 component floating point vector
vec3 – a 3 component floating point vector
vec4 – a 4 component floating point vector
bvec2 – a 2 component Boolean vector
bvec3 – a 3 component Boolean vector
bvec4 – a 4 component Boolean vector
ivec2 – a 2 component vector of integers
ivec3 – a 3 component vector of integers
ivec4 – a 4 component vector of integers

mat2 – a 2X2 matrix of floating point numbers
mat3 – a 3X3 matrix of floating point numbers
mat4 – a 4X4 matrix of floating point numbers

sampler1D – a handle for accessing a texture with 1 dimension
sampler2D – a handle for accessing a texture with 2 dimensions
sampler3D – a handle for accessing a texture with 3 dimensions
samplerCube – a handle for accessing cube mapped textures
sampler1Dshadow – a handle for accessing a depth texture in one dimension
sampler2Dshadow – a handle for accessing a depth texture in two dimensions

Pointers??? (not supported)

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

```
matrix3 func(matrix3 a); // call by value
```

Qualifiers

Attribute, uniform, varying, or const

- GLSL has many of the same qualifiers such as **`const`** as C/C++
  - const float one = 1.0;
  - const vec3 origin = vec2(1.0, 2.0, 3.0);
- Need others due to the nature of the execution model
- Variables can change
  - ✓ Once per primitive
  - ✓ Once per vertex
  - ✓ Once per fragment
  - ✓ At any time in the application
- **Vertex attributes are interpolated by the rasterizer into fragment attributes**

Attribute Qualifier

- Attribute-qualified variables **can change at most once per vertex**
  - ✓ **Cannot be used** in fragment shaders

    (Just can be used in vertex shader)

    ※ Because they vary on a vertex-by-vertex basis, vertex attributes cannot be declared in a fragment shader.
- **Built in** (OpenGL state variables)
  - ✓ `gl_Color`
  - ✓ `gl_ModelViewMatrix`
- **User defined** (assigned in application program)

  ※ Only floating point types can be attribute-qualified
  - ✓ `attribute float temperature`
  - ✓ `attribute vec3 velocity`

Uniform Qualified variable

- Variables that are constant for an entire primitive
- Can be changed in application outside scope of **`glBegin`** and **`glEnd`**
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

In Application,

```
GLint        timeParam;
timeParam = glGetUniformLocation(program, "time");
```

In Shader,

```
uniform float time;
```

Varying Qualified variable

- Variables that are passed from **vertex shader** to **fragment shader**
- Automatically interpolated by the rasterizer
- Built in
  - ✓ Vertex colors
  - ✓ Texture coordinates
- User defined
  - ✓ Requires the same user defined varying qualified variable in fragment shader

※ User-defined varying variables that are set in the vertex program are automatically interpolated bye the rasterizer. It does not make sense to define a varying variable in a vertex shader and not use it in fragment shaders, Consequently, if we use such a variable in the vertex shader, we must write a corresponding fragment shader.

Passing values

- call by **value-return**
- Variables are copied in
- Returned values are copied back
- Three possibilities
  - ✓ **in**
  - ✓ **out**
  - ✓ **inout**

Operators and Functions

- Standard C functions
  - ✓ Trigonometric
  - ✓ Arithmetic
  - ✓ Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Toon Shading Example

- Toon Shading
  - Characterized by abrupt change of colors
  - Vertex Shader computes the vertex intensity (declared as varying)
  - Fragment Shader computes colors for the fragment based on the interpolated intensity

Vertex Shader

```
uniform vec3 lightDir;
varying float intensity;
void main() {
   vec3 ld;
   intensity = dot(lightDir,gl_Normal);
   gl_Position = ftransform();
}
```

Fragment Shader

```
varying float intensity;

void main() {
   vec4 color;
   if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
   else if (intensity > 0.5) color = vec4(0.6,0.3,0.3,1.0);
   else if (intensity > 0.25) color = vec4(0.4,0.2,0.2,1.0); else color
   = vec4(0.2,0.1,0.1,1.0);
   gl_FragColor = color;
 }
```