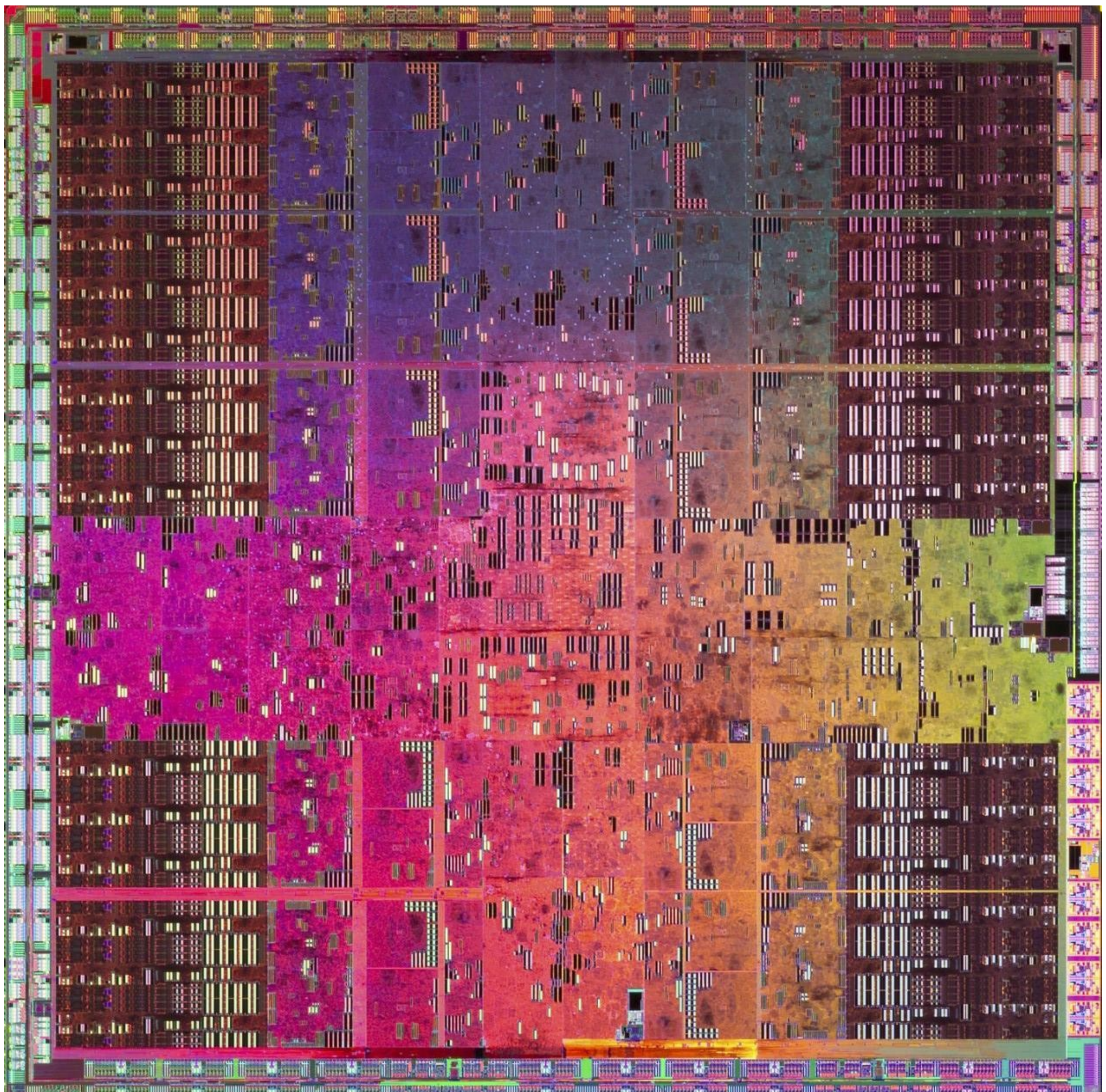
GLSL : The OpenGL Shading Language

- First version was part of the OpenGL 2.0 (2004) specification
- A high-level language similar to C
- Replaced the vertex program extension which defined a low-level GPU assembly language
- Compiled by the video driver into microcode which runs on the GPU

The OpenGL Pipeline

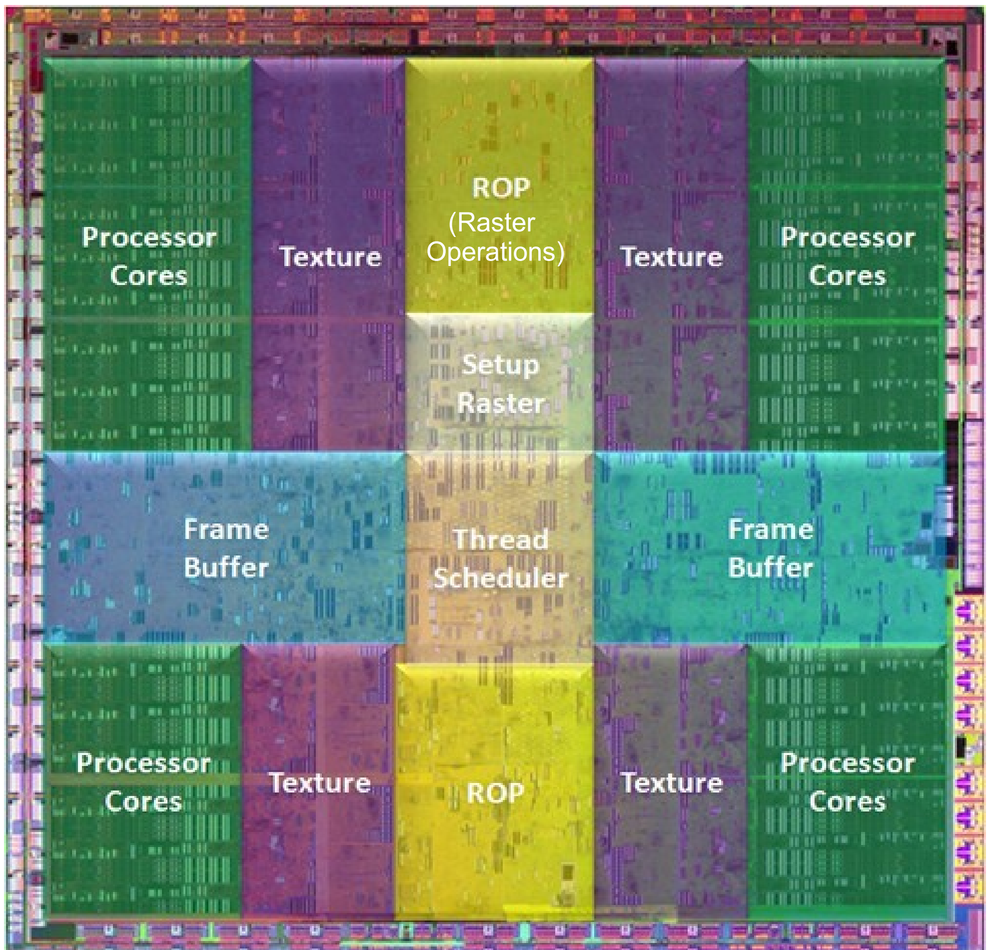
- Pipeline roughly corresponds to dataflow though the GPU



- Nvidia GT200 (2008)
  - 1.4 billion transistors
  - In GTX 280 cards

The OpenGL Pipeline

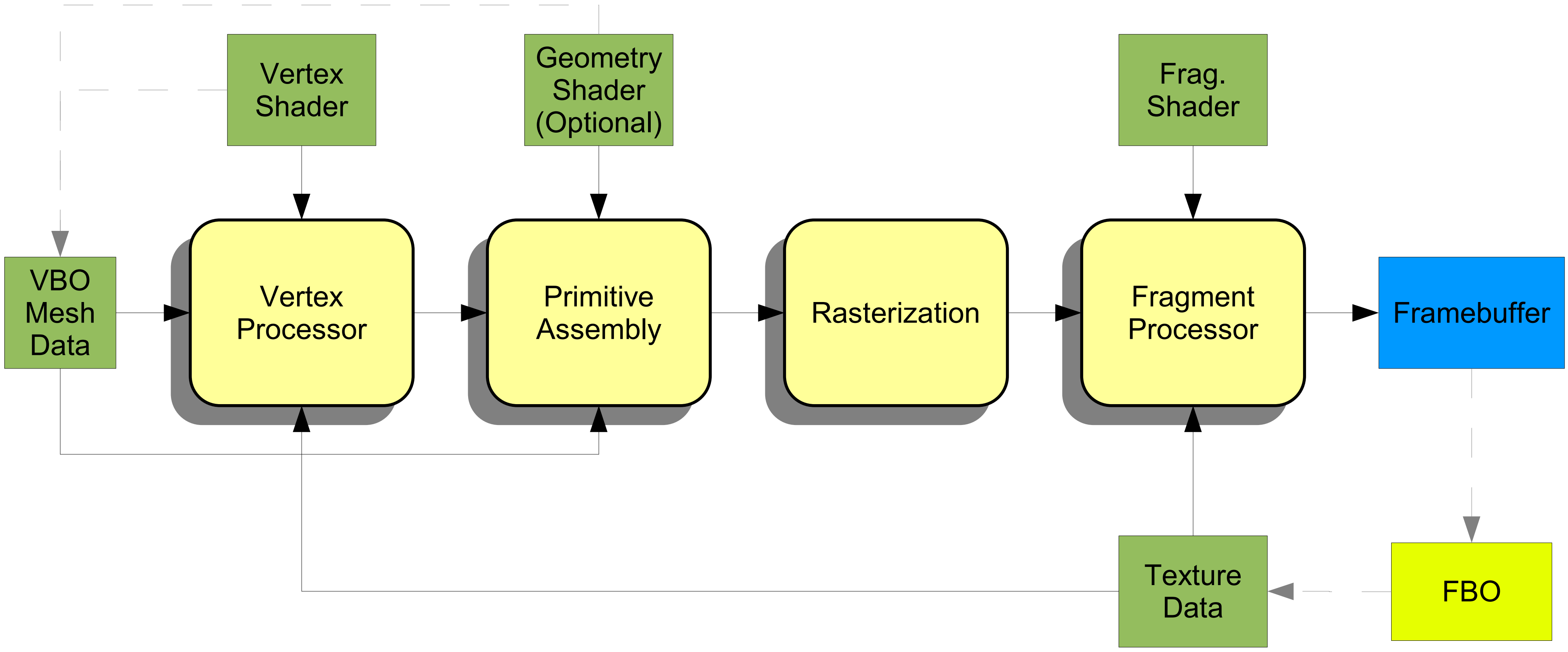
- Pipeline roughly corresponds to dataflow though the GPU



- Nvidia GT200 (2008)
  - 1.4 billion transistors
- Nvidia GK110 (2013)
  - 7.1 billion transistors
- Nvidia TU102 (2018)
  - 18.6 billion transitors
- Compare with 10-core i7
  - 3.2 billion transistors
  - Lots of it is cache

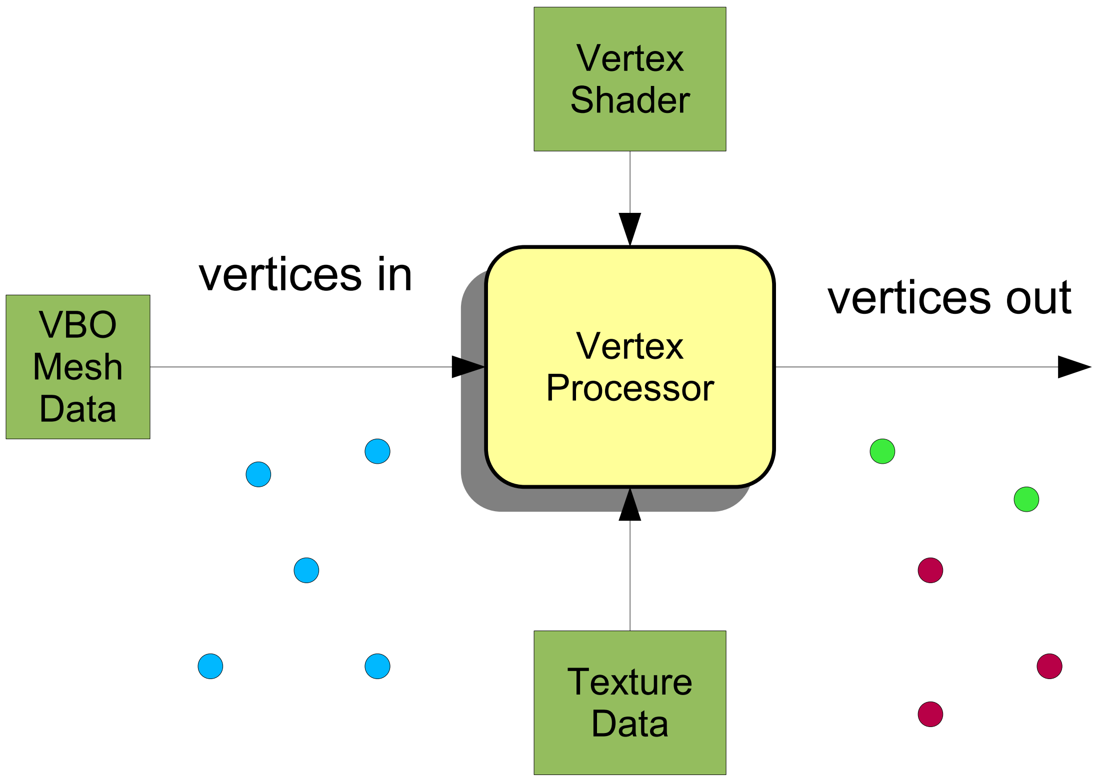
The OpenGL 4-Stage Pipeline

- Block diagram



Items in green are supplied by your application.

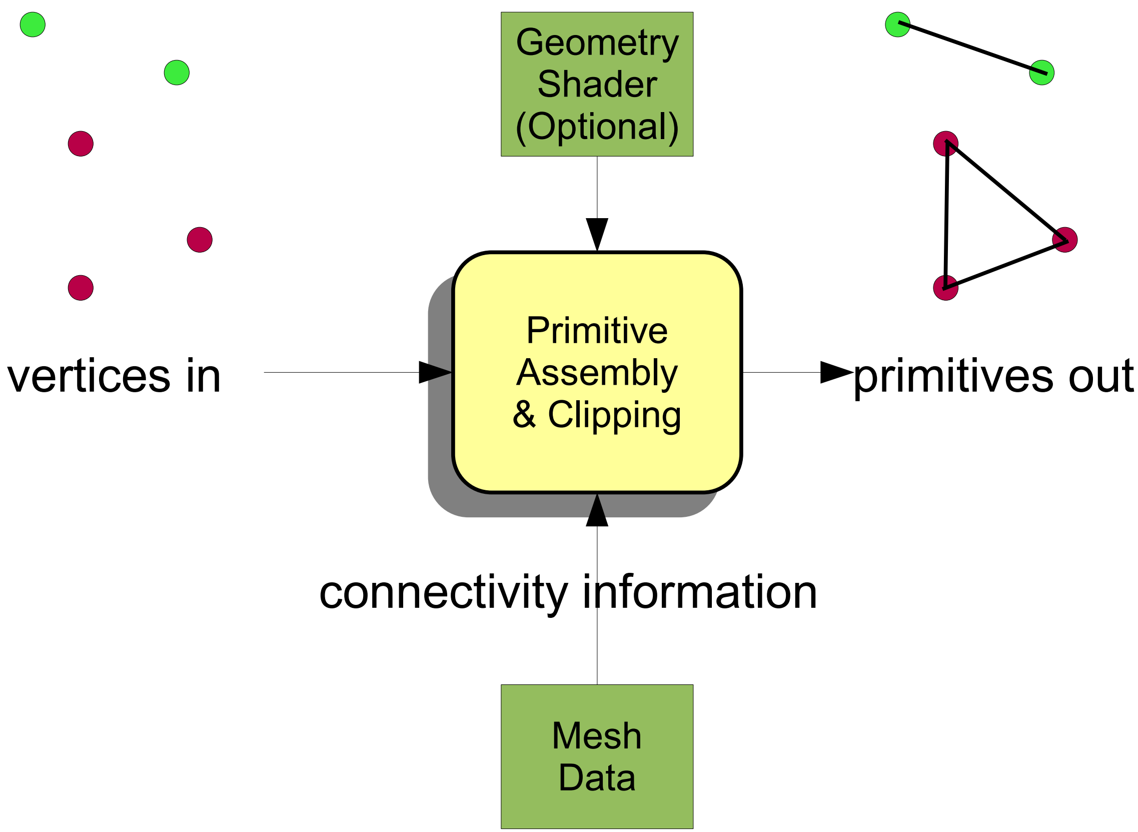
Stage 1 : The Vertex Processor



- Transform vertex locations
- Modify vertex **attributes**
  - Normal vector
  - Color
  - Texture Coordinates

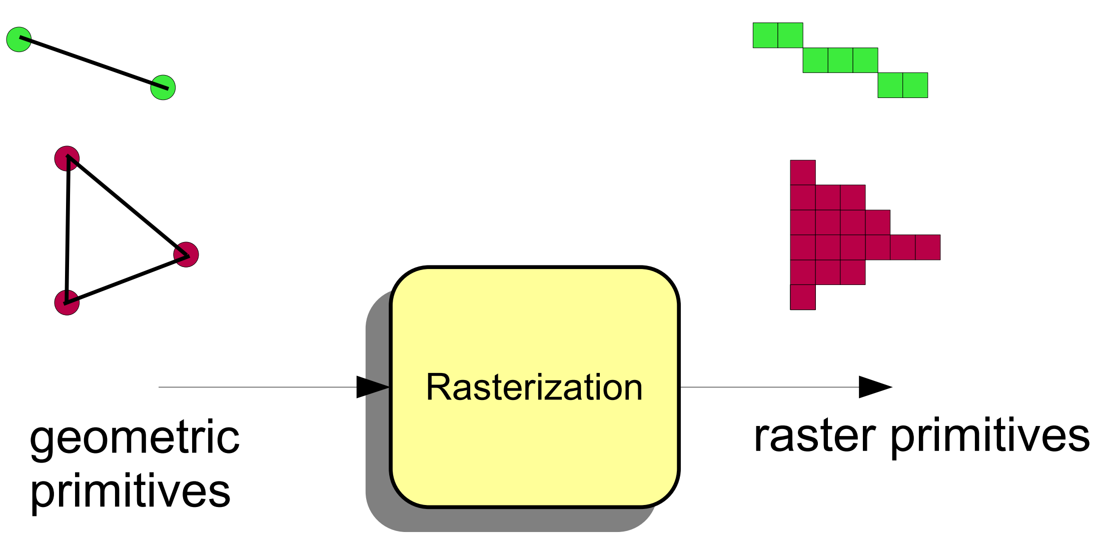
"**attributes**" are variables associated with each vertex (declared as ***in*** in the vertex shader)

Stage 2 : Primitive Assembly



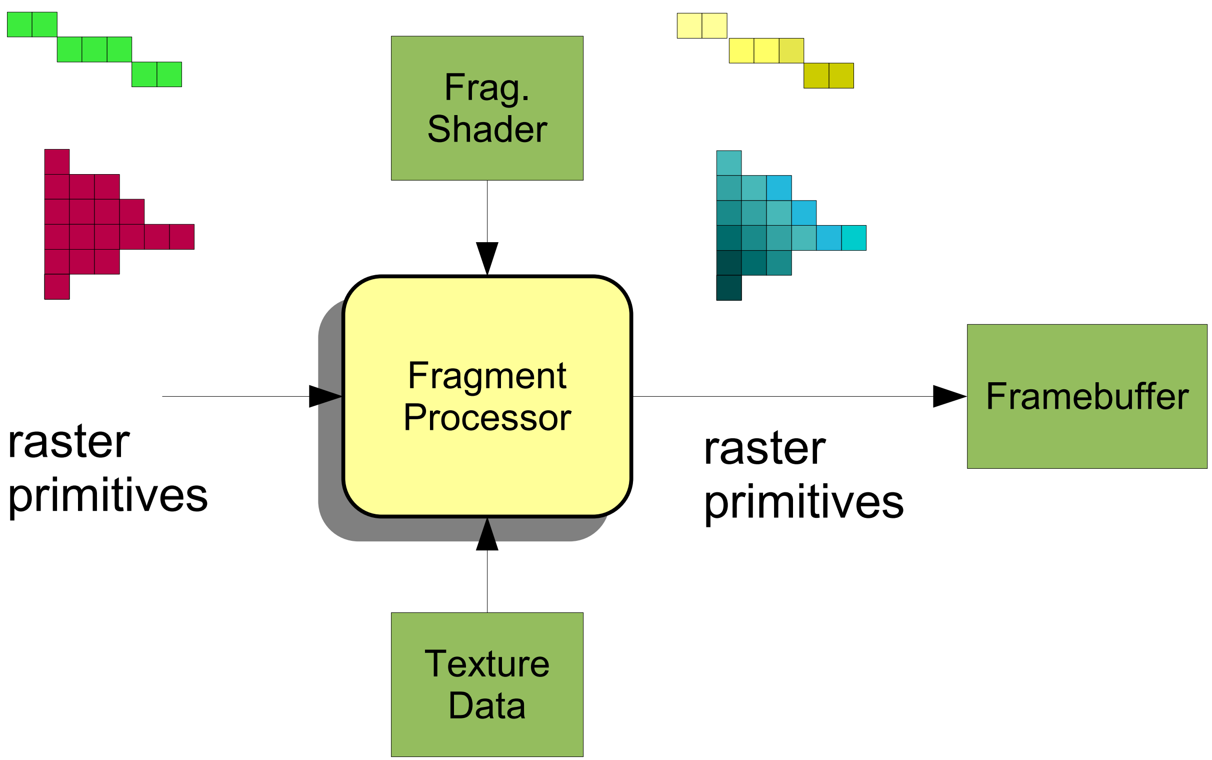
- Assemble points, lines, triangles
- Vertices come from vertex processor
- Connectivity information comes from mesh data supplied by client application
- Clipping also happens here

Stage 3 : Rasterization



- Determine which pixel locations are associated with each geometric primitive
- Not programmable

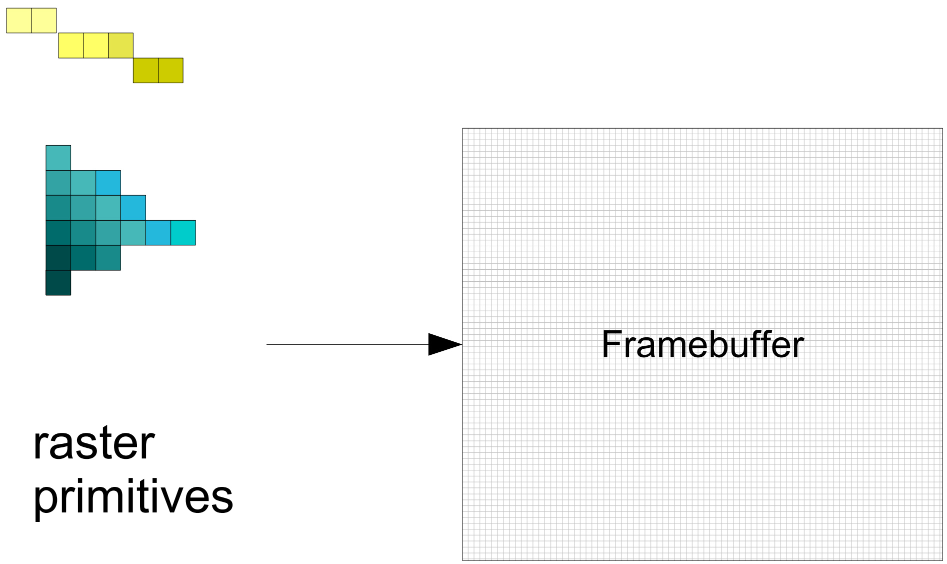
Stage 4 : Fragment Processor



- Determine final appearance of each fragment
  - Evaluate per-pixel lighting
  - Texture mapping

Recall that a fragment is **not** a pixel.
A fragment may be eventually discarded and not seen.

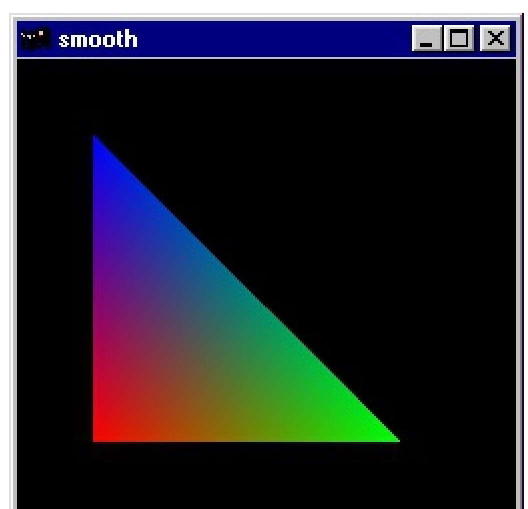
The Framebuffer



- Framebuffer Operations
- Determine final **pixel** appearance
  - Depth Test
  - Alpha Blending
  - Stencil Operations

Vertex Shaders

- Set vertex position
  - Commonly: Modeling, viewing and projection
  - Animation
- Set value of *varying* variables used by fragment shader
  - Declared as ***out*** in vertex shader, ***in*** in fragment shader
    - Color
    - Texture coordinates
    - User-defined quantities which will be interpolated over the primitive



Datatypes in glsl

- Data types
  - Some familiar from C: float, double, int, bool
  - New vector types: vec2, vec3, vec4, dvec*
  - And square matrices: mat2, mat3, mat4
  - Also integer and boolean vectors: ivec, bvec
- Operators
  - * operator performs matrix-vector and matrix-matrix multiplication
- Built-in geometric functions
  - dot(), cross(), normalize(), length(), reflect()

See full specification for details:

http://www.opengl.org/registry/doc/GLSLangSpec.4.00.9.clean.pdf

Vector Component Access

Vectors are structs that can be interpreted as

- Points / vectors
  - p.x, p.y, p.z, p.w
- Colors
  - c.r, c.g, c.b, c.a
- Texture coordinates
  - tex.s, tex.t, tex.p, tex.q
- Arrays
  - a[0], a[1], a[2], a[3]

'Swizzling' is allowed

```
p.xyz = q.zyx;
```

Vector construction examples

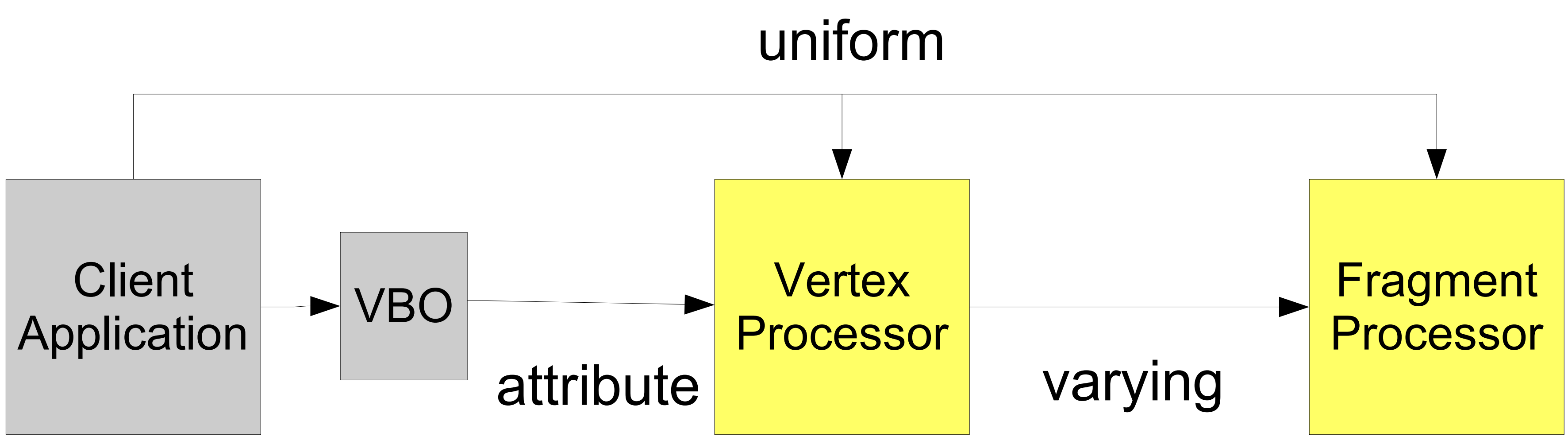
- vec4 v = vec4(1.0, 0.0, 0.0, 1.0);
  - **Can't** init as vec4 v(1.0, 0.0, 0.0, 1.0);
- vec3 v = vec3(1.0); // all components initialized to 1
- vec3 u = vec3(1.0); vec4 v = vec4(u, 1.0);
- vec2 u = vec2(0.0, 1.0); vec4 v = vec4(u, 0.0, 1.0);
- vec2 a; vec3 b;... vec4 v(a.zx, b);

Boolean relations

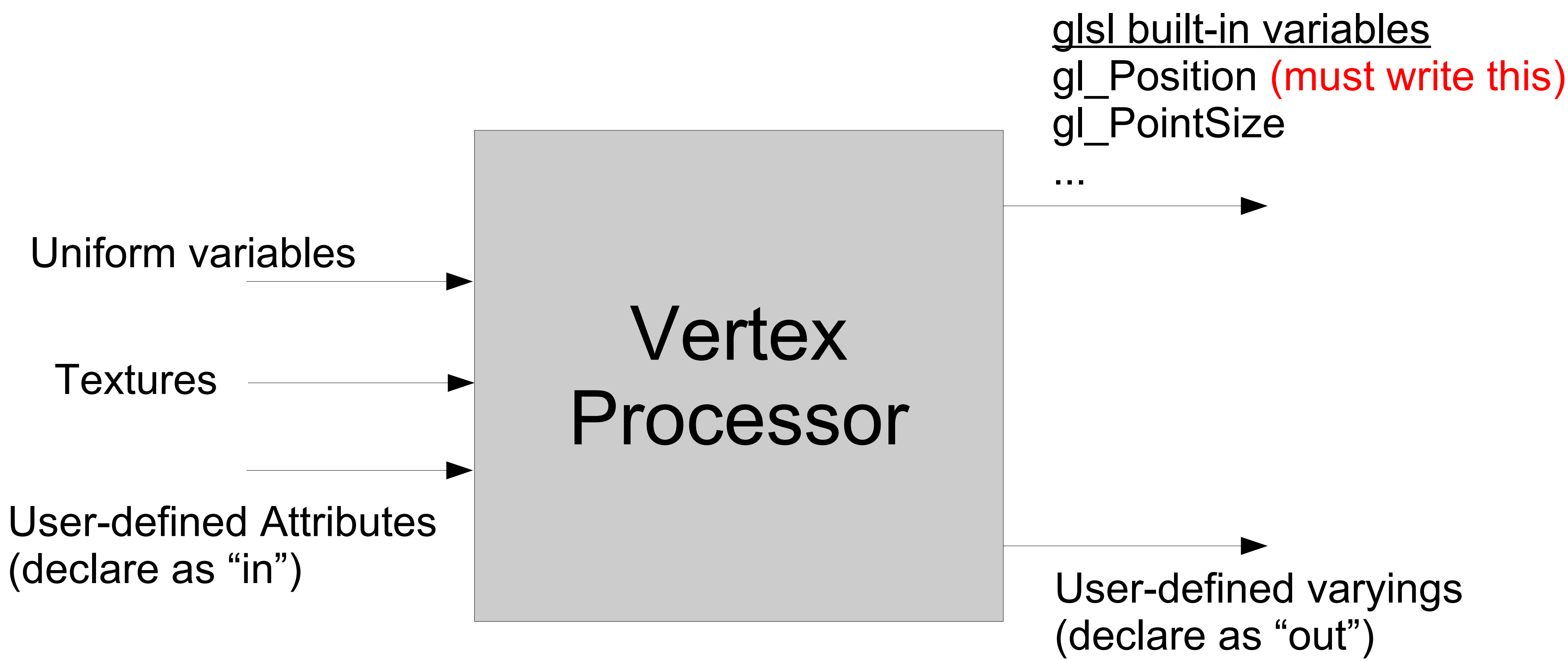
- bool all(bvec)
  - Are all vector components true?
- bool any(bvec)
  - Is any vector component true?
- Componentwise comparisons
  - bvec3 equal(vec3 a, vec3 b)
    - Component i is true if a[i] == b[i]
    - greaterThan, lessThan work similarly

Variable storage classes

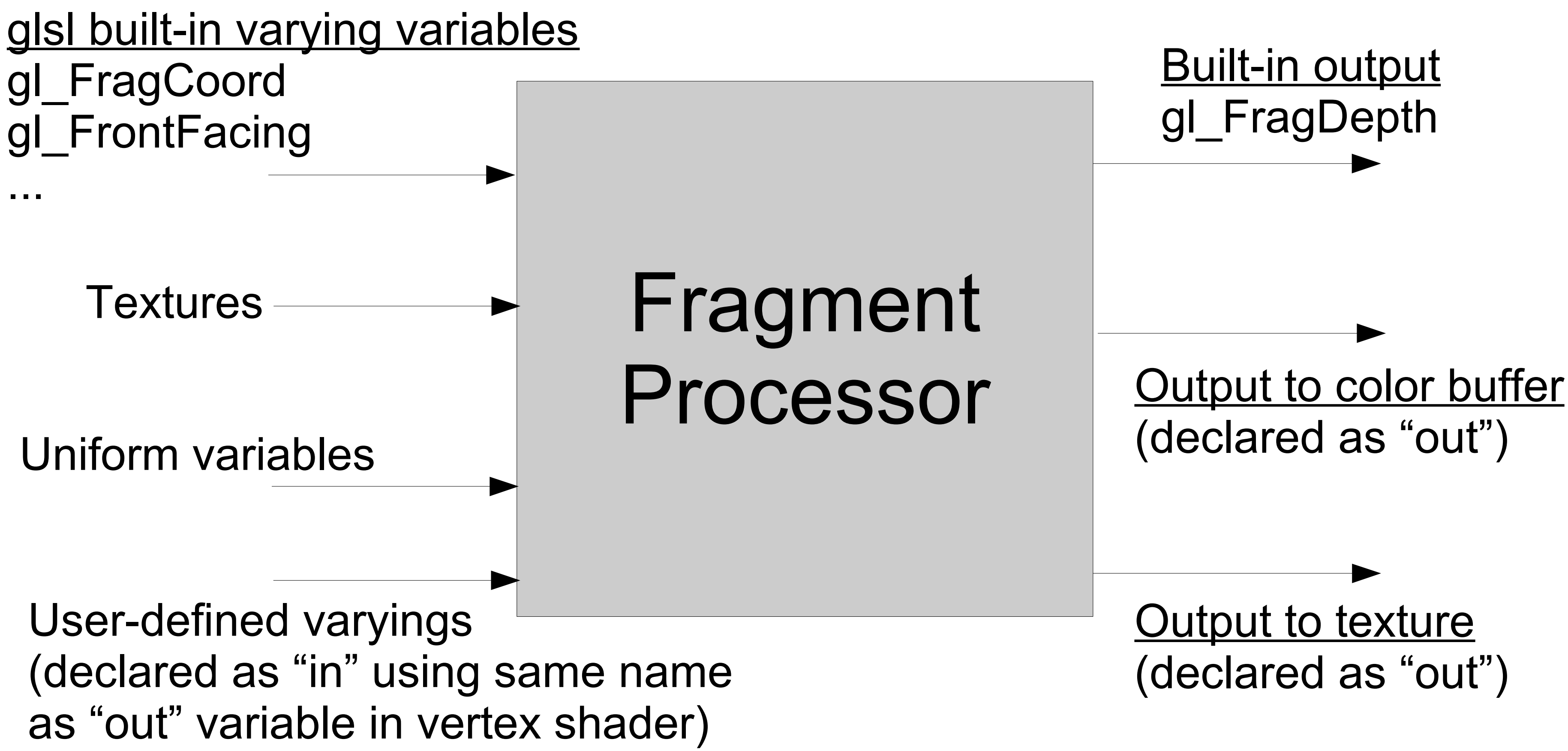
- **Uniform**: input to VP and FP from application.
  - Changes value per draw call (set before draw calls with glUniform*())
- **Attribute** : input to VP from VBO
  - Can change value per vertex (like normal, tex coord)
- **Varying** : output from VP, input to FP
  - Interpolated value per fragment available in FP
  - (like smooth-shaded color)
- **Const** : compile time constant



Vertex Shader Inputs and Outputs



Fragment Shader Inputs and Outputs



Fragment Shaders

- Set fragment color
  - Lighting, texturing
- Set fragment depth
  - **Cannot** change screen-space (x,y) position
- Write output
  - Write to framebuffer
  - Write to zero or more textures
    - Using frame buffer object functionality

Fragment Shader Functions

discard;

- Only allowed in fragment shader
- Abandon the operation on the current fragment.
- The fragment to be discarded and no updates to any buffers will occur.

```
if (alpha <= 0.0)
discard;
```

Basic Vertex and Fragment Shaders

- Vertex program
  - Transform incoming vertex coordinates
  - Send variables to fragment (or other) shader

```
#version 400        //use version 4 of glsl
uniform mat4 M;     //set by client application
in vec4 vPosition; //a vertex attribute
out vec4 color;     //a varying variable

void main()
{
     gl_Position = M*vPosition;
     color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Basic Vertex and Fragment Shaders

- Fragment program
  - Receive varibles from other shader stages
  - Compute a color to send to frame buffer

```
#version 400
in vec4 color;        //variable coming from vertex shader
out vec4 fragcolor;   //color sent to framebuffer

void main( void )
{
    fragcolor = color;
}
```

Structures

- You can group multiple types into a single structure
- This can simplify passing groups of values into functions

```
struct Particle
{
	float lifetime;
	vec3 position;
	vec3 velocity;
}

//declare a particle
Particle p = Particle(10.0, pos, vel);

//declare a function that takes a particle
void Update(Particle a);
```

Flow control

- Glsl supports C-style looping and flow control
  - If /else
  - Switch / case
  - While
  - Do / while

Loading shaders

- The shader code will be stored in strings or character arrays in your program
  - Hard coded, or read from files, network
- Shader programs will be compiled and linked when your application runs

```
const char* vshader = {
"#version 400 \n"
"uniform mat4 M; \n"
"in vec3 vPos; \n"
"void main() \n"
"{"
"    gl_Position = M*vec4(vPos, 1.0);"
"}"
};
```

- More details about creation later...