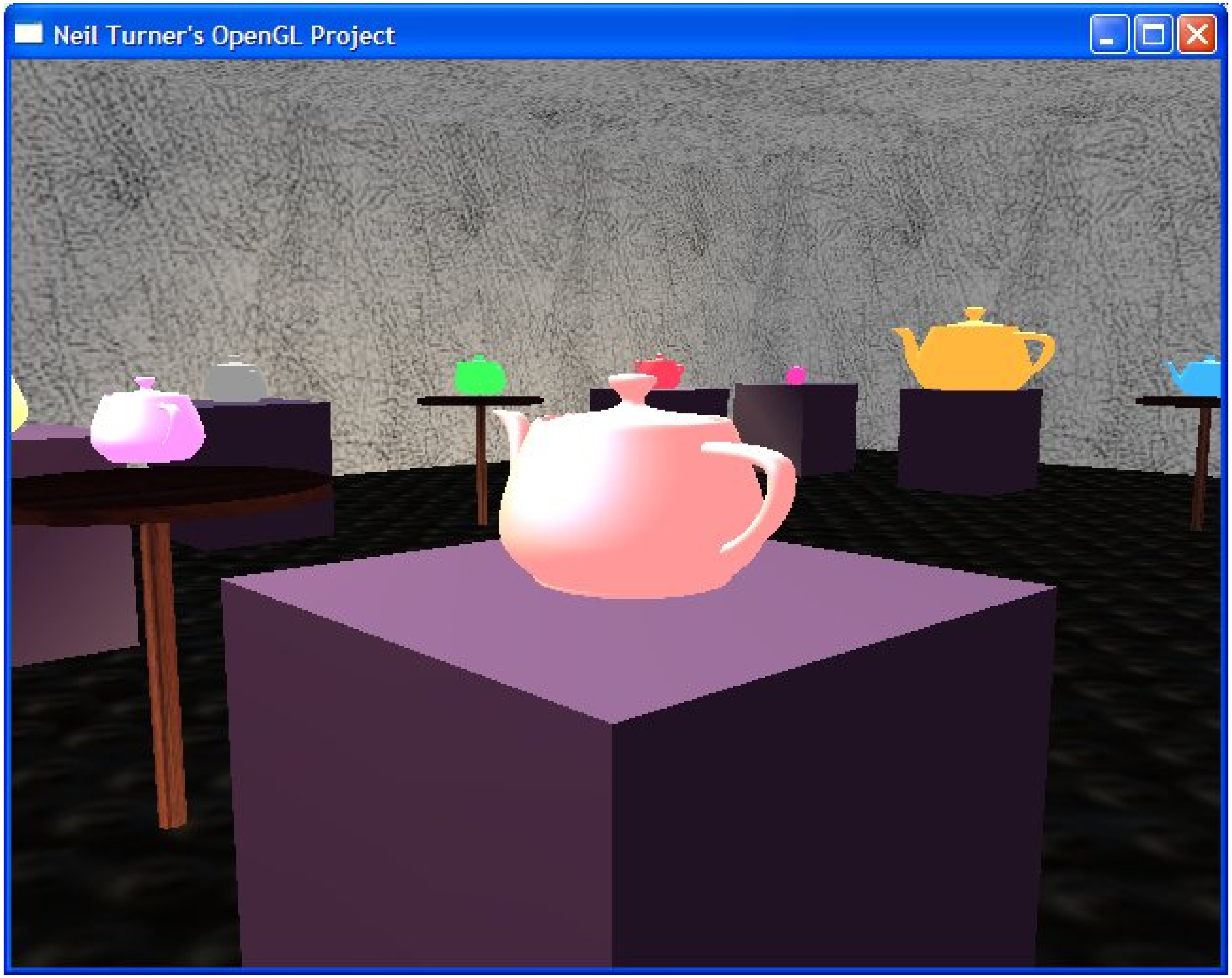
Intro to GLSL

CS4620 Lecture 18

Guest Instructor: Nicolas Savva

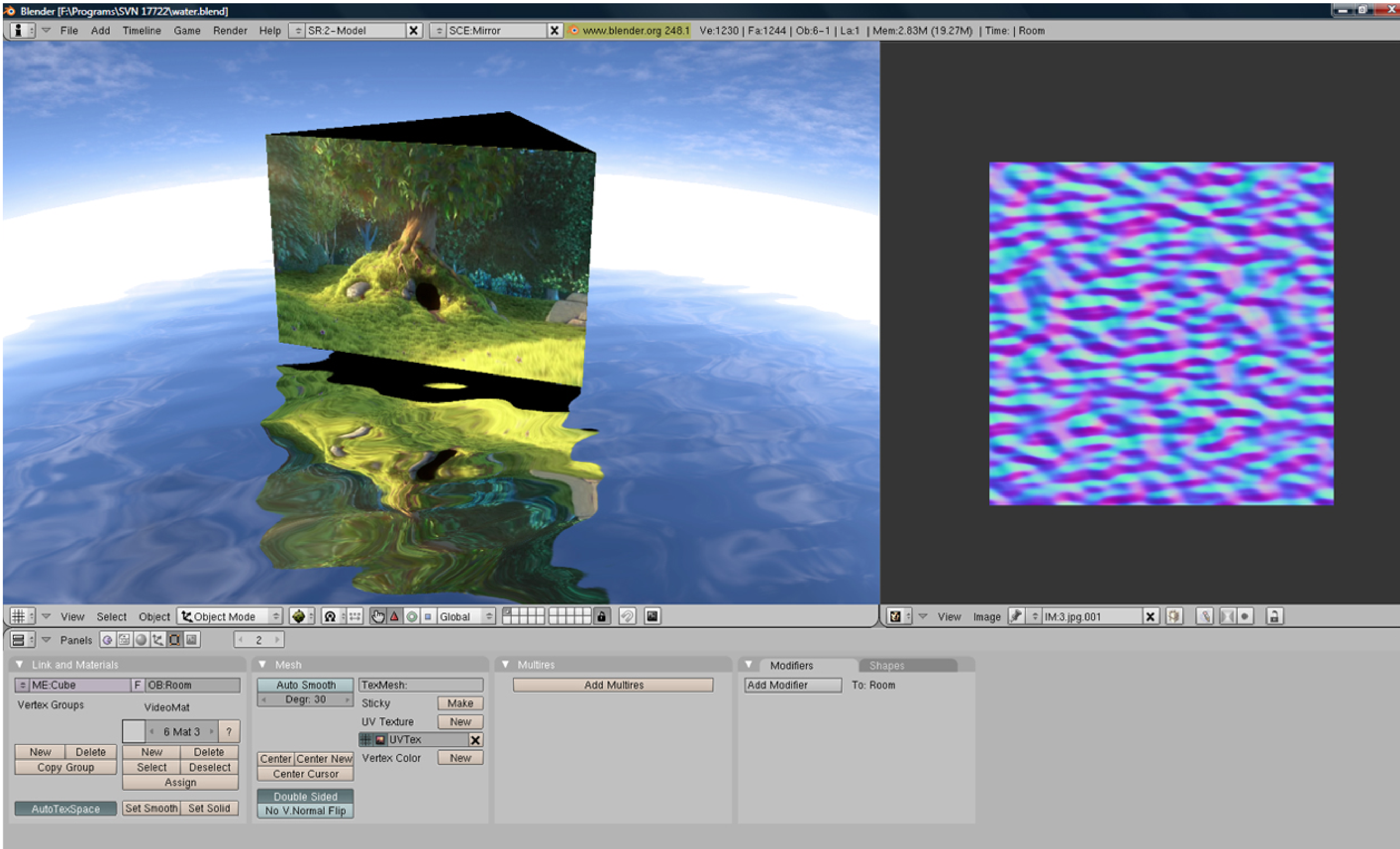
OpenGL 25 years ago



http://www.neilturner.me.uk/shots/opengl-big.jpg

OpenGL 25 years ago

OpenGL 2009 ->


Blender [F:\Programs\SVN 17722\water.blend]
File Add Timeline Game Render Help
SR:2-Model
SCE:Mirror
www.blender.org 248.1
Ve:1230 | Fa:1244 | Ob:6-1 | La:1 | Mem:2.83M (19.27M) | Time: | Room
View Select Object
Object Mode
Global
View Image
IM:3.jpg.001
Panels
Link and Materials
ME:Cube
OB:Room
Vertex Groups
VideoMat
6 Mat 3
New Delete
Copy Group
Select Deselect
Assign
AutoTexSpace
Set Smooth Set Solid
Mesh
Auto Smooth
Degr: 30
TexMesh:
Sticky Make
UV Texture New
UVTex
Vertex Color New
Center Center New
Center Cursor
Double Sided
No V.Normal Flip
Multires
Add Multires
Modifiers
Shapes
Add Modifier
To: Room

What changed?

25 years ago:

- Vertex transformation/fragment shading hardcoded into GPUs

Now:

- More parts of the GPU are programmable
  (but not all)

What changed?

25 years ago (Fixed pipeline)**:**

- Transform vertices with modelview/projection matrices
- Shade with Phong lighting model only

Contemporary (Programmable hardware)**:**

- Custom vertex transformation
- Custom lighting model
- More complicated visual effects
- Shadows
- Displaced and detailed surfaces
- Simple reflections and refractions

GLSL

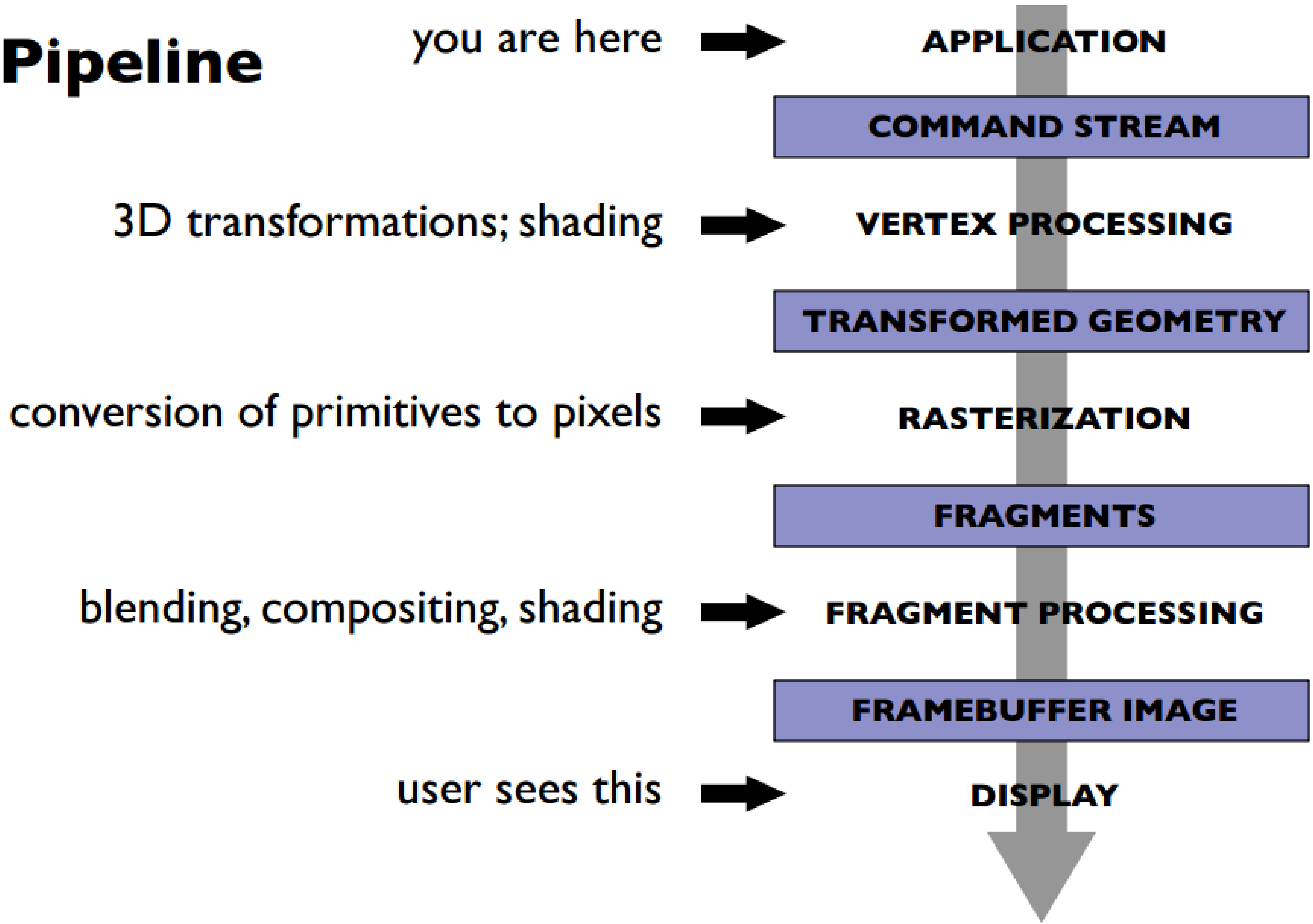
GLSL : **G**raphics **L**ibrary **S**hading **L**anguage

- Syntax similar to C/C++
- Language used to write shaders
  - vertex, tessellation, geometry, fragment, compute
  - We only cover vertex and fragment shaders today
- Based on OpenGL
  - First available in OpenGL 2.0 (2004)
- Alternatives: Nvidia Cg and Microsoft HLSL

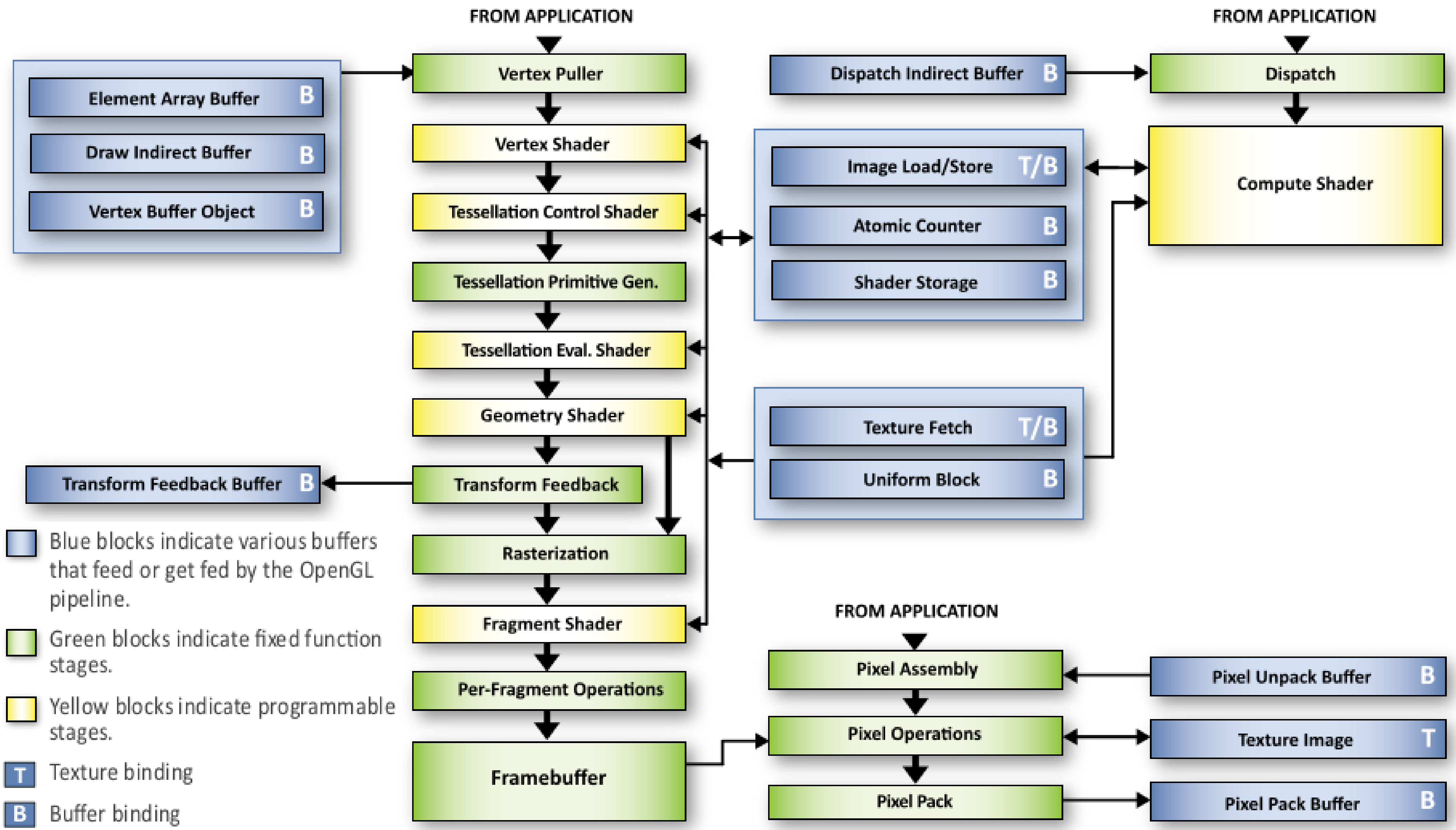
What is a Shader Program?

- A small program to control parts of the graphics pipeline
- Consists of 2 (or more) separate parts:
  - Vertex shader controls vertex transformation
  - Fragment shader controls fragment shading

Pipeline


you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
user sees this
DISPLAY

Modern Graphics Pipeline



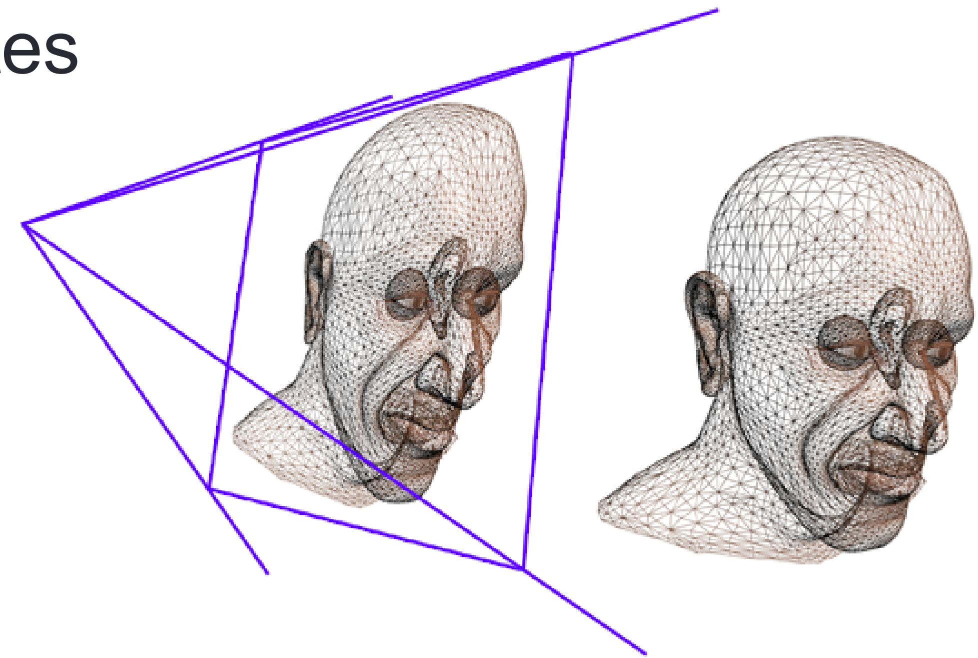
GLSL Program

- Specifies how OpenGL should draw geometry
- Program: A collection of shaders that run together
  - At least one vertex shader or one fragment shader
- At any time, the GPU runs only one program
  - Must specify program to use before drawing geometry

Vertex Shader

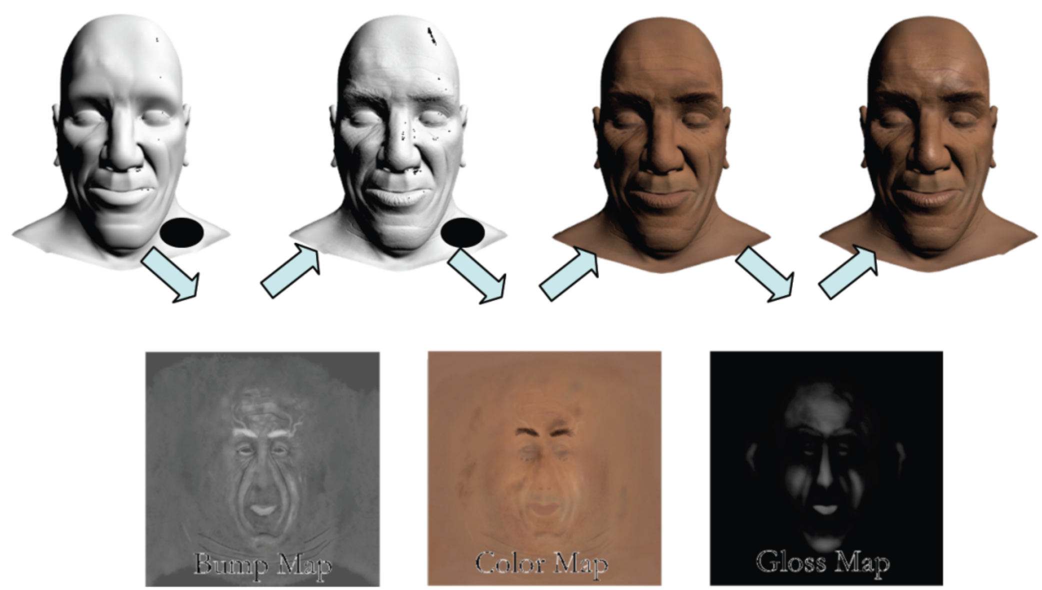
- Transform vertices from object space to clip space

- Compute other data that are interpolated with vertices
  - Color
  - Normals
  - Texture coordinates
  - Etc



Fragment Shaders

- Compute the color of a fragment (i.e. a pixel)
- Take interpolated data from vertex shaders
- Can read more data from:
  - Textures
  - User specified values



To use a GLSL program…

Follow the next 7 steps:

1. Create shader objects.
2. Read source code from files and feed them to the shader objects just created.
3. Compile the shader.
4. Create a program object.
5. Attach the shaders to the program.
6. Link the program.
7. Tell OpenGL to use your shader program.

CS 4620/4621 Framework

- Contains GLProgram class to abstract OpenGL calls:
  - Added convenience methods
  - Help keep conventions straight
  - Controls mapping between attribute variables and vertex buffers

Now, to create a GLSL program…

- Create a GLProgram object

```
private GLProgram program;

public void onEntry(GameTime gameTime) {

  program = new GLProgram();
  program.quickCreateResource(
      "cs4620/gl/Grid.vert",   // Path to vertex shader
      "cs4620/gl/Grid.frag",   // Path to fragment shader
      null);                 // Optional attribute list
}
```

OpenGL/GLSL Plumbing

- Suppose we have already created the program
- We tell OpenGL to use it.
- We then instruct OpenGL to draw the two triangles:

HelloWorldScreen.java:

```
// The vertices in our vertex buffer, initialized earlier
float [] vertexPositions = {
           -0.5f, -0.5f,      // vertex 0
            0.5f, -0.5f,      // vertex 1
            0.5f,  0.5f,      // vertex 2
           -0.5f,  0.5f       // vertex 3
                     };
//...
// In the draw method
program.use();

glDrawElements(...);

GLProgram.unuse();
```

GLSL Data Types

- Both in GLSL and Java
  - float, int
- GLSL has, but Java does not have
  - vec2, vec3, vec4: vectors
  - mat2, mat3, mat4: matrices
  - sampler1D, sampler2D, sample3D, samplerCube, etc: textures
- Java has, but GLSL does not have
  - Object
  - String
  - etc...

vec2

- Represents a vector in 2D (each component is a float)

```
vec2 a;
a.x = 0.0;
a.y = 1.0; // a = (0,1)

vec2 b;
b.s = 10.0;
b.t = 12.5; // b = (10,12.5)

vec2 c;
c[0] = 9.0;
c[1] = 8.0; // c= (9,8)
```

vec3

```
vec3 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0;  // a = (10,  20,  30)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;   // a = (0.1, 0.2, 0.3)
a.s = 1.0,  a.t = 2.0;  a.p = 3.0;   // a = (1, 2, 3)

vec3 b = vec3(4.0, 5.0, 6.0);

vec3 c = a + b; // c = (5, 7, 9)
vec3 d = a - b; // d = (-3, -3, -3)
vec3 e = a * b; // e = (4, 10, 18)
vec3 f = a * 3; // e = (3, 6, 9)
float g = dot(a,b); // g = 32
vec3 h = cross(a,b); // h = (-5,6,-3)
float i = length(a); // i = 3.742
```

vec4

```
vec4 a;
a.x = 10.0; a.y = 20.0; a.z = 30.0; a.w = 40.0;
// a = (10, 20, 30, 40)
a.r = 0.1;  a.g = 0.2;  a.b = 0.3;  a.a = 0.4;
// a = (0.1, 0.2, 0.3, 0.4)
a.s = 1.0;  a.t = 2.0;  a.p = 3.0;  a.q = 4.0;
// a = (1, 2, 3, 4)

vec4 b = vec4(5, 6, 7, 8);

vec4 c = a + b; // c = (6, 8, 10, 12)
vec4 d = a - b; // d = (-4, -4, -4, -4)
vec4 e = a * b; // e = (5, 12, 21, 32)
vec4 f = a * 3; // f = (3, 6, 9, 12)
float g = length(a); // g = 5.477
```

mat2

- Represents a 2 by 2 matrix (each component is a float)

```
mat2 A = mat2(1.0, 2.0, 3.0, 4.0); // in column-major order

vec2 x = vec2(1.0, 0.0);
vec2 y = vec2(0.0, 1.0);

vec2 a = A * x; // a = (1,2)
vec2 b = A * y; // b = (3,4)
```

mat3

```
mat3 A = mat3(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
// in column-major order

vec3 x = vec3(1.0, 0.0, 0.0);
vec3 y = vec3(0.0, 1.0, 0.0);
vec3 z = vec3(0.0, 0.0, 1.0);

vec3 a = A * x; // a = (1,2,3)
vec3 b = A * y; // b = (4,5,6)
vec3 c = A * z; // c = (6,7,8)
```

mat4

- 4x4 matrices (can store affine transformations)

```
mat4 A = mat4(1.0,  2.0,  3.0,  4.0,
              5.0,  6.0,  7.0,  8.0,
              9.0,  10.0, 11.0, 12.0
              13.0, 14.0, 15.0, 16.0); // in column-major order

vec4 x = vec4(1.0, 0.0, 0.0, 0.0);
vec4 y = vec4(0.0, 1.0, 0.0, 0.0);
vec4 z = vec4(0.0, 0.0, 1.0, 0.0);
vec4 w = vec4(0.0, 0.0, 0.0, 1.0);

vec4 a = A * x; // a = (1,2,3,4)
vec4 b = A * y; // b = (5,6,7,8)
vec4 c = A * z; // c = (9,10,11,12)
vec4 d = A * w; // d = (13,14,15,16)
```

Array

- We can declare fixed-size arrays (size known at compile time)
- Use C syntax.

```
float A[4];
A[0] = 5; A[3] = 10;

vec4 B[10];
B[3] = vec4(1,2,3,4);
B[8].y = 10.0;
```

Swizzling

- Used to construct a vector from another vector by referring to multiple components at one time.

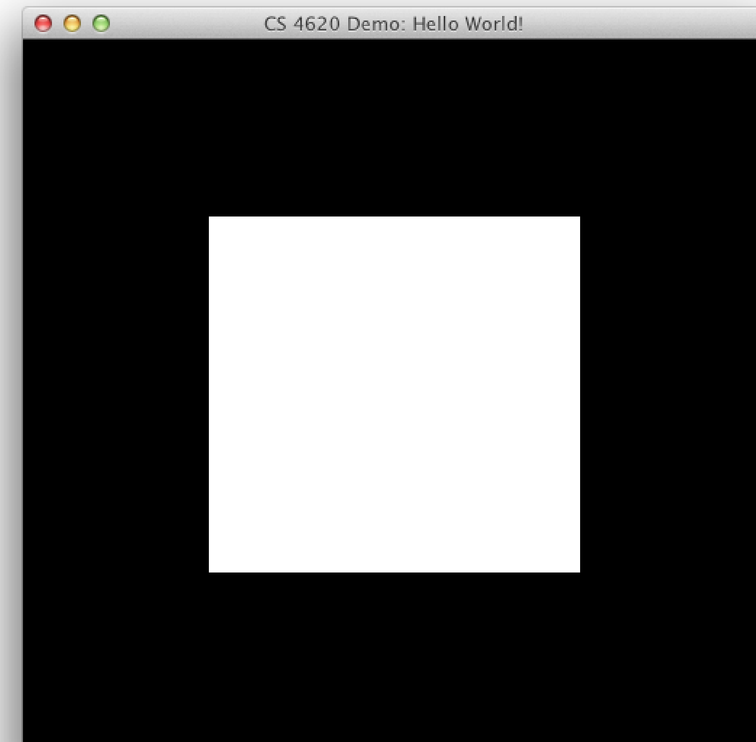
```
vec4 a = vec4(1,2,3,4);

vec3 b = a.xyz;  // b = (1,2,3)
vec2 c = a.qp;   // c = (4,3)
vec4 d = a.xxyy; // d = (1,1,2,2)
```

Type Conversion

- Syntax: <<variable>> = <<type>>(<<value>>);
- Expression on RHS = “constructor expression.”
- Example:

```
float a = 1.0;
int b = int(a);
```

Revisiting Hello World!



Example: Hello World's draw()

```
@Override
public void draw(GameTime gameTime) {
    GL11.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    GL11.glClear(GL11.GL_COLOR_BUFFER_BIT);

    program.use();

              GLUniform.setST(program.getUniform("VP"),
              new Matrix4(), false);
              GLUniform.set(program.getUniform("uGridColor"),
              new Vector4(1, 1, 1, 1));

              vb.useAsAttrib(program.getAttribute("vPos"));
              ib.bind();
              GL11.glDrawElements(GL11.GL_TRIANGLES, indexCount,
              GLType.UnsignedInt, 0);
              ib.unbind();

    GLProgram.unuse();
}
```

Shader Structure

```
/*
Multiple-lined comment
*/

// Single-lined comment

//
// Global variable definitions
//

void main()
{
    //
    // Function body
    //
}
```

Vertex Shader

Grid.vert:

```
#version 120

uniform mat4 VP;

attribute vec4 vPos;

void main()
{
    gl_Position = VP * vPos;
}
```

Each time the screen is drawn, this main() function is called once per vertex, as if it were in a for loop.

Vertex Shader

Grid.vert:

```
#version 120

uniform mat4 VP;

attribute vec4 vPos;

void main()
{
    gl_Position = VP * vPos;
}
```

The first thing to do is specify the GLSL version.
(Note: syntax in other versions can be rather different!)

Vertex Shader

Grid.vert:

```
#version 120

uniform mat4 VP;

attribute vec4 vPos;

void main()
{
    gl_Position = VP * vPos;
}
```

Uniforms are one type of input to the shader. They are the same for each vertex drawn during one draw function. We saw how to set them in the OpenGL lecture.

Vertex Shader

Grid.vert:

```
#version 120

uniform mat4 VP;

attribute vec4 vPos;

void main()
{
    gl_Position = VP * vPos;
}
```

Attribute variables link to vertex attributes, or data associated with each vertex. This one is set to the vertex position buffer. Each time main() is executed, vPos is set to the vertex currently being processed.

Vertex Shader

Grid.vert:

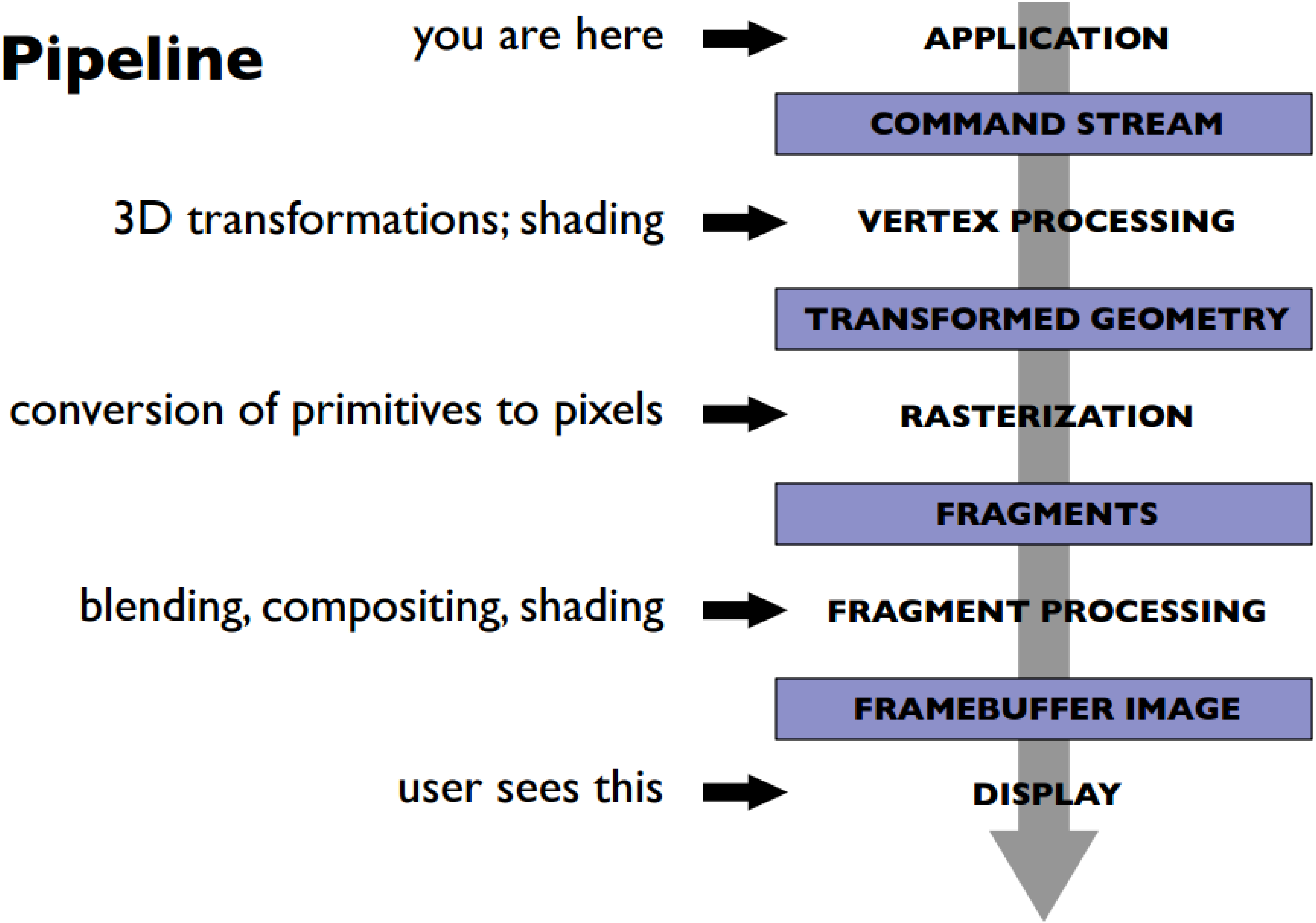
```
#version 120

uniform mat4 VP;

attribute vec4 vPos;

void main()
{
    gl_Position = VP * vPos;
}
```

gl_Position is a special variable that holds the position of the vertex in clip space.

Since a vertex shader's main output is the position in clip space, it must **always** set gl_Position.

This vertex shader just transforms each vertex position (by the VP matrix).

Pipeline


you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
user sees this
DISPLAY

Fragment Shader

Grid.frag:

```
#version 120

uniform vec4 uGridColor;

void main()
{
  gl_FragColor = uGridColor; //vec4(1,1,1,1);
}
```

Each time the screen is drawn, this main() function is called once per pixel.

Fragment Shader

Grid.frag:

```
#version 120


uniform vec4 uGridColor;


void main()
{
  gl_FragColor = uGridColor; //vec4(1,1,1,1);
}
```

gl_FragColor is a special variable that stores the color of the output fragment.

Since a fragment shader computes the color of a fragment, it must **always** set gl_FragColor.

Fragment Shader

Grid.frag:

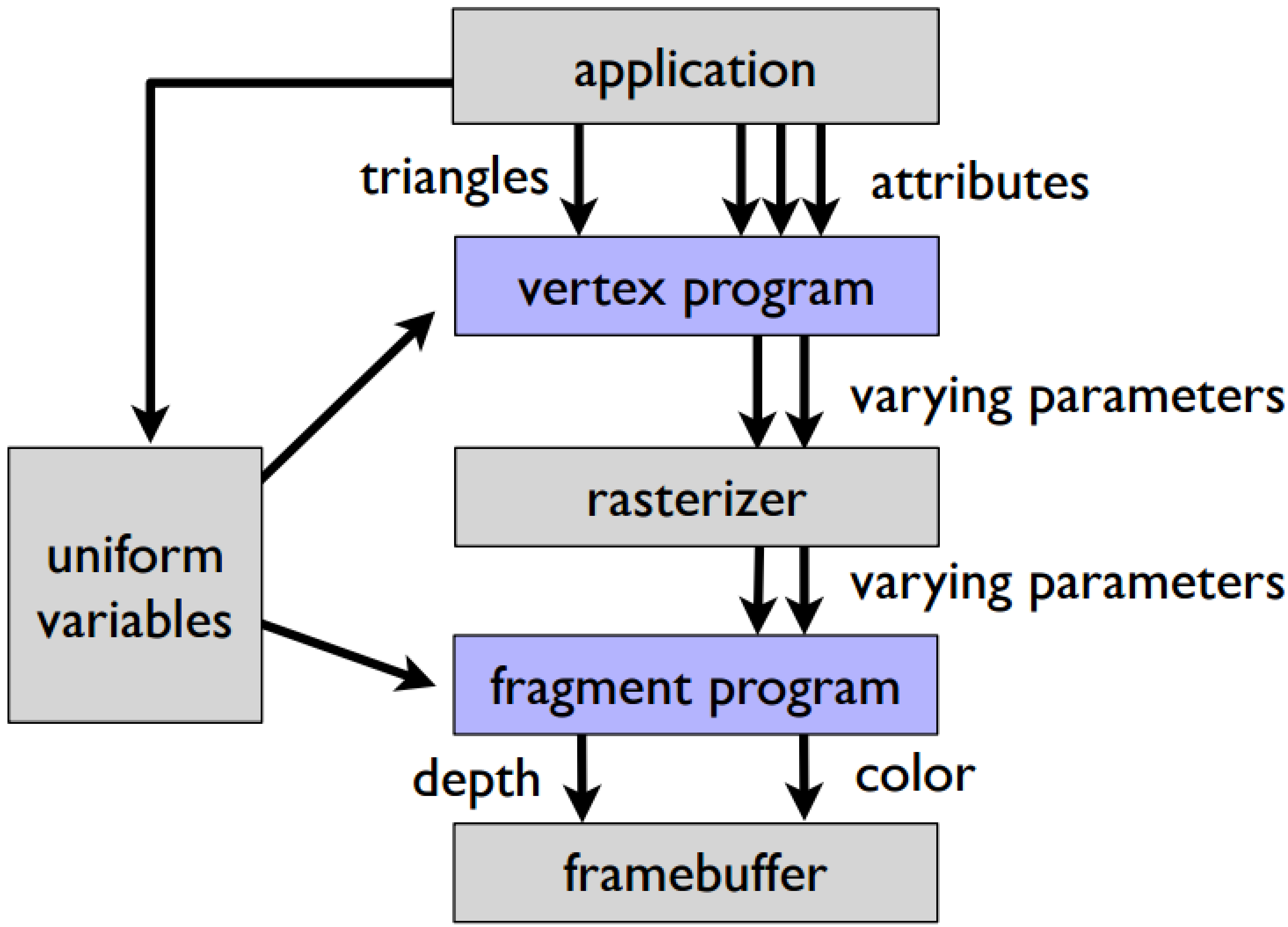
```
#version 120


uniform vec4 uGridColor;


void main()
{
  gl_FragColor = uGridColor;
//vec4(1,1,1,1);
}
```

vec4 is the data type of 4D vectors.

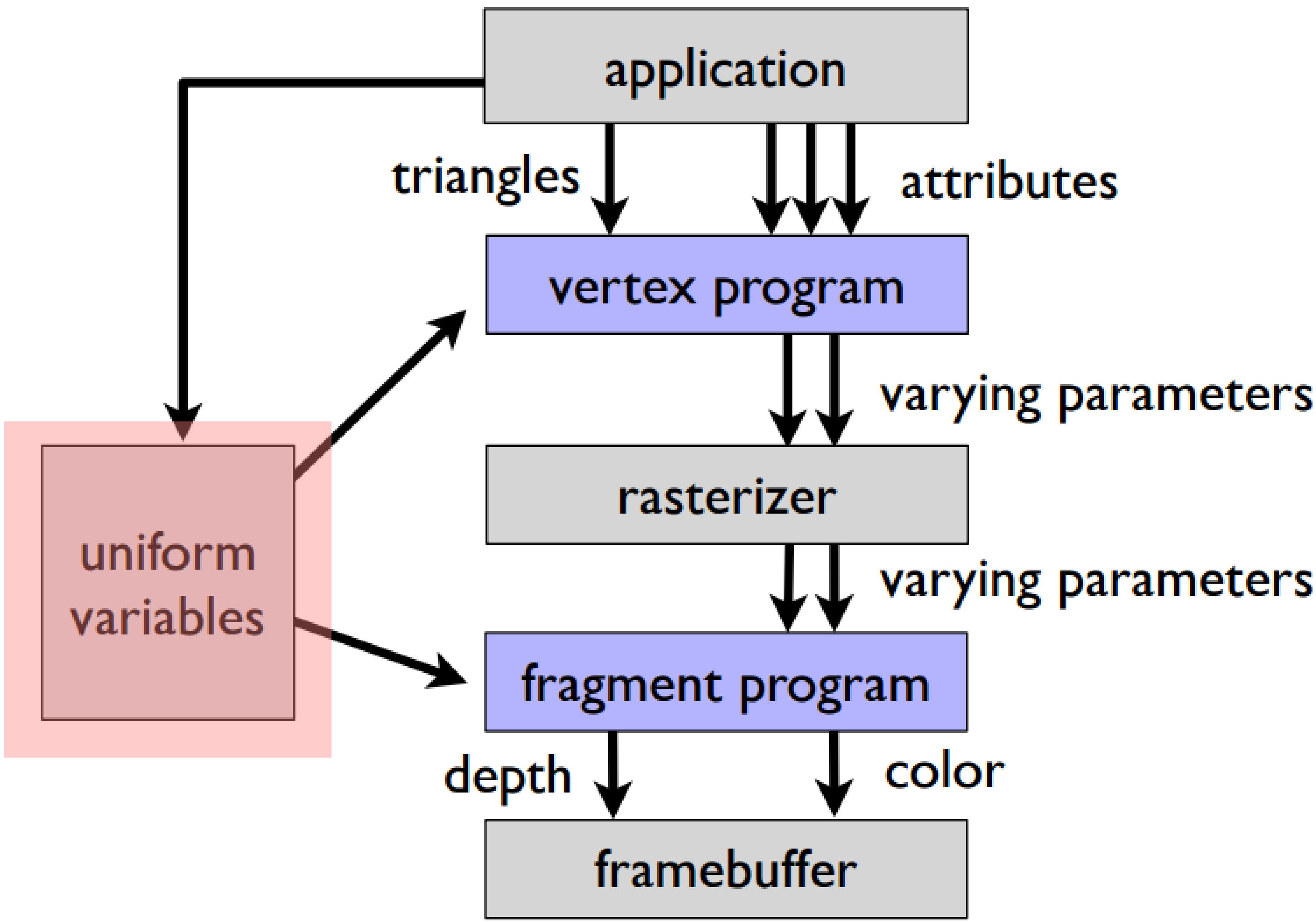
Can be used to store:
- homogeneous coordinates
- RGBA color

vec4(1,1,1,1) constructs an RGBA tuple with R=1, G=1, B=1, A=1, which is white. (Note it is commented out here. We are passing the same information for the color using the vec4 uniform uGridColor)

GLSL passing data around



GLSL passing data around



Uniform Variable

- A GLSL variable the user can specify value from the C/Java side.
- Its value is constant while drawing each vertex and pixel.
- Suitable for specifying
  - Material properties
  - Transformation matrices
  - Light sources
  - Textures

Declaring a Uniform Variable in GLSL

- Declare as a global variable (outside functions).
- Prefix the variable type with keyword “uniform”
- Examples:

```
// The values for these are initialized in the Java code!
uniform float shininess;
uniform vec3 color;
uniform mat4 model_transform;

void main()
{
    // Code here...
}
```

Caveats

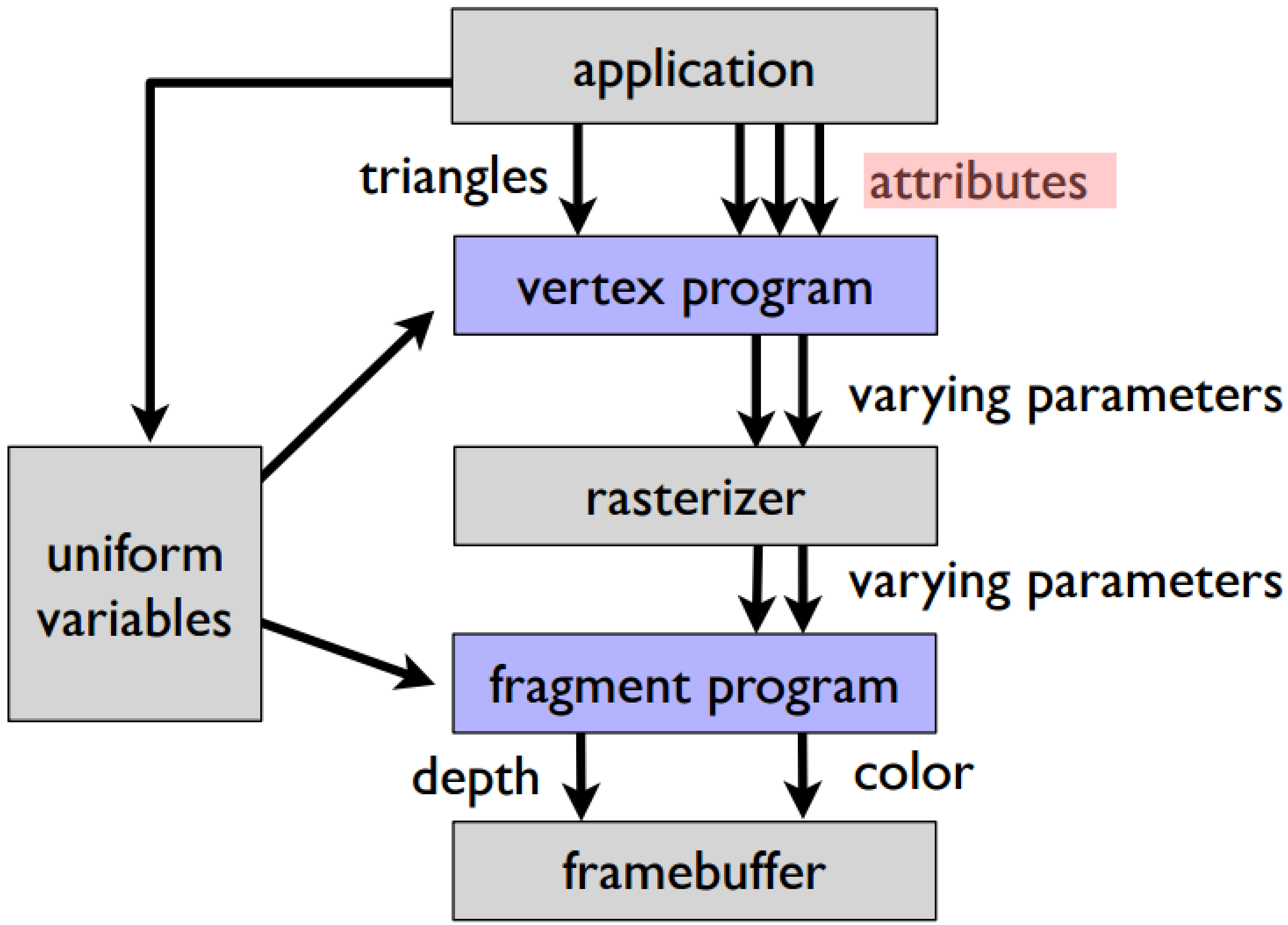
- Uniform variables are shared between vertex and fragment shaders
  - Declare once in vertex shader and once more in fragment shader.
- As a result, types of uniform variables in vertex and fragment shaders must be consistent.
- Cannot have **uniform int x**; in vertex shader, but **uniform float x;** in fragment shader.
- Uniforms that are declared but not used are “optimized” out
  - OpenGL throws an error if you try to set a nonexistent uniform

Using Uniform Variables in the CS4620/4621 Framework

- Uniform variables are encapsulated by `GLUniform` class.
- Use `program.getUniform(<name>)` to get the instance (an integer) corresponding to the name.
- Set values by `GLUniform.set**(...)` methods.

```
// In GLSL: uniform vec3 color;
program.use();
Vector3 c = new Vector3(1.0f, 0.5f, 1.0f);
GLUniform.set(program.getUniform("uGridColor"), c);

// In GLSL: uniform mat4 MVP;
// For matrices, use setST, not set! A boolean is provided
// for transposing.
GLUniform.setST(program.getUniform("VP"),
    camera.mViewProjection, false);
```

GLSL passing data around



Attribute Variables

- A variable containing an attribute for a single vertex.
- Position, normal, texture coordinate, etc.
- Each time the shader is run, the attribute variables receive the values for the current vertex.
- These only appear in vertex shaders (Why?)

Attribute Mapping

- Attribute variables map to OpenGL buffers.
- OpenGL buffers have an index, GLSL attribute variables have a name.
- Must ensure the mapping from buffer indices to variable names is correct.
- In the provided framework:

```
// Create a data buffer to fill in the attribute data
GLBuffer vertexPositions = new
GLBuffer(BufferTarget.ArrayBuffer, BufferUsageHint.StaticDraw,
true);

vertexPositions.setAsVertexVec3();

// Set vertexPositions, e.g. by reading in a Mesh

vertexPositions.useAsAttrib(program.getAttribute("vPos"));
```

Demo: Twisting

2D Twisting

- We transform vertices according to the following equation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\left(t\sqrt{x^2+y^2}\right) & -\sin\left(t\sqrt{x^2+y^2}\right) \\ \sin\left(t\sqrt{x^2+y^2}\right) & \cos\left(t\sqrt{x^2+y^2}\right) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where

- (x,y) is the vertex position in object space.
- (x’,y’) is the vertex position in clip space.
- t is the twisting factor,
  which is stored in the uniform variable “twisting”

Vertex Shader Code

```
#version 120

uniform float twisting;

void main()
{
    float angle = twisting * length(gl_Vertex.xy);
    float s = sin(angle);
    float c = cos(angle);
    gl_Position.x = c * gl_Vertex.x - s * gl_Vertex.y;
    gl_Position.y = s * gl_Vertex.x + c * gl_Vertex.y;
    gl_Position.z = 0.0;
    gl_Position.w = 1.0;
}
```

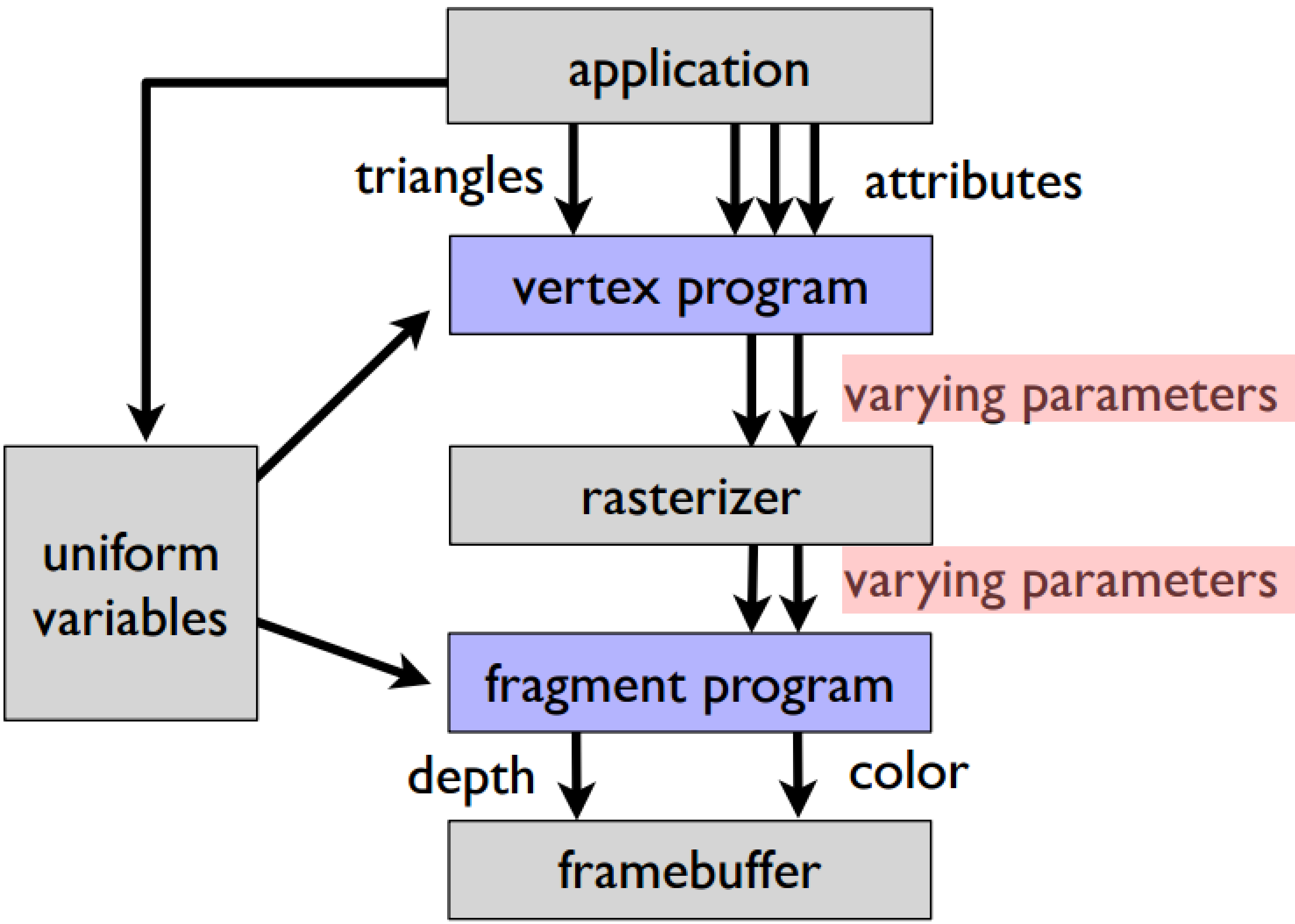
Fragment Shader Code

```
#version 120


uniform vec3 color;

void main()
{
    gl_FragColor = vec4(color, 1);
}
```

Using the GLSL Program

```
public void draw(GameTime gameTime) {
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
   glClear(GL2.GL_COLOR_BUFFER_BIT);

   program.use();

   // Set the uniforms
   GLUniform.set(program.getUniform("color"), color);
   GLUniform.set(program.getUniform("twisting"), 0.5f);

   // Set the attribute
   vertexPositions.useAsAttrib(program.getAttribute("vPos"));

   glDrawElements(...); // Draw the mesh

   GLProgram.unuse();
}
```

GLSL passing data around



Varying Variables

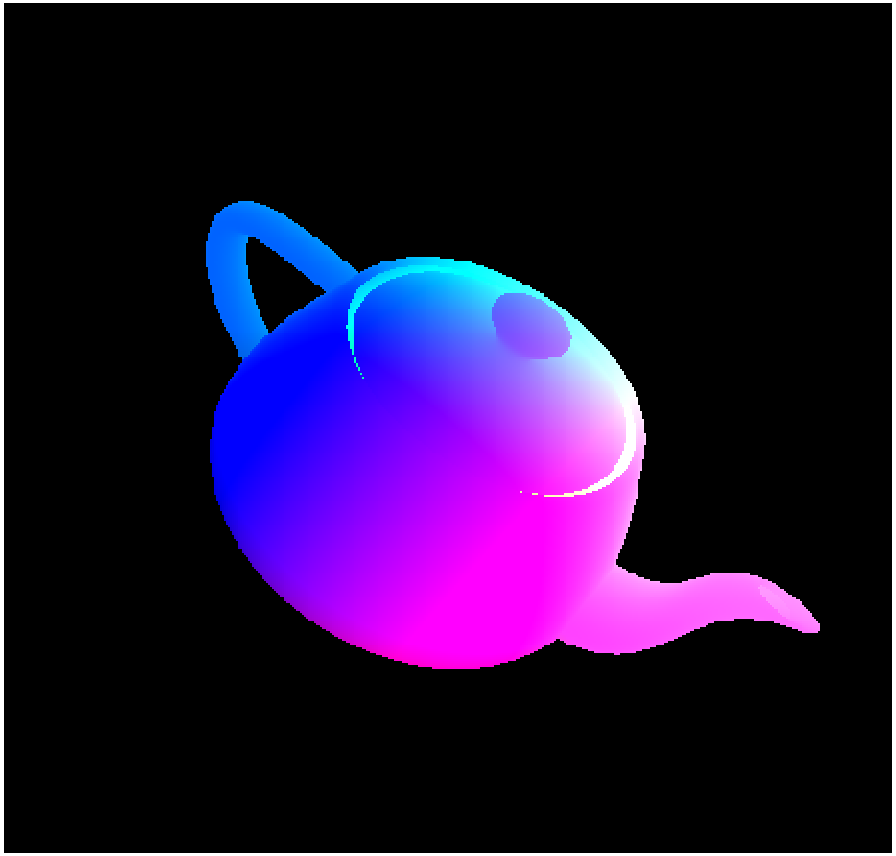
- Interface between vertex and fragment shaders.
- Vertex shader outputs a value at each vertex, writing it to this variable.
- Fragment shader reads a value from the same variable, automatically interpolated to that fragment.
- No need to declare these in the Java program (Why?)

Declaring Varying Variables

- Declare as a global variable (outside functions).
- Syntax: varying <<type>> <<name>>;
- Example:

```
varying vec3 color;

void main()
{
    // Some code here...
}
```

Demo: Position as Color



Position as Color

Compute the color of each fragment from its position in object space

color = (position + (1,1,1)) / 2

Vertex Shader Code (older syntax)

```
#version 120

varying   vec3 color;
attribute vec3 vPos;
uniform   mat4 VP;

void main()
{
   gl_Position = VP * vec4(vPos,1);
   color = (vec3(gl_Position.xyz) + vec3(1,1,1)) * 0.5;
}
```

Fragment Shader Code (older syntax)

```
#version 120

varying vec3 color;

void main()
{
   gl_FragColor = vec4(color, 1);
}
```

Vertex Shader Code (modern syntax)

```
#version 330


in vec3 vPos;
uniform mat4 VP;

out vec3 color;
out vec4 vPosition;

void main()
{
   vPosition = VP * vec4(vPos,1);
   color = (vec3(gl_Position.xyz) + vec3(1,1,1)) * 0.5;
}
```

Fragment Shader Code (modern syntax)

```
#version 330

in vec3 color;
out vec3 vFragColor;

void main()
{
   vFragColor = vec4(color, 1);
}
```