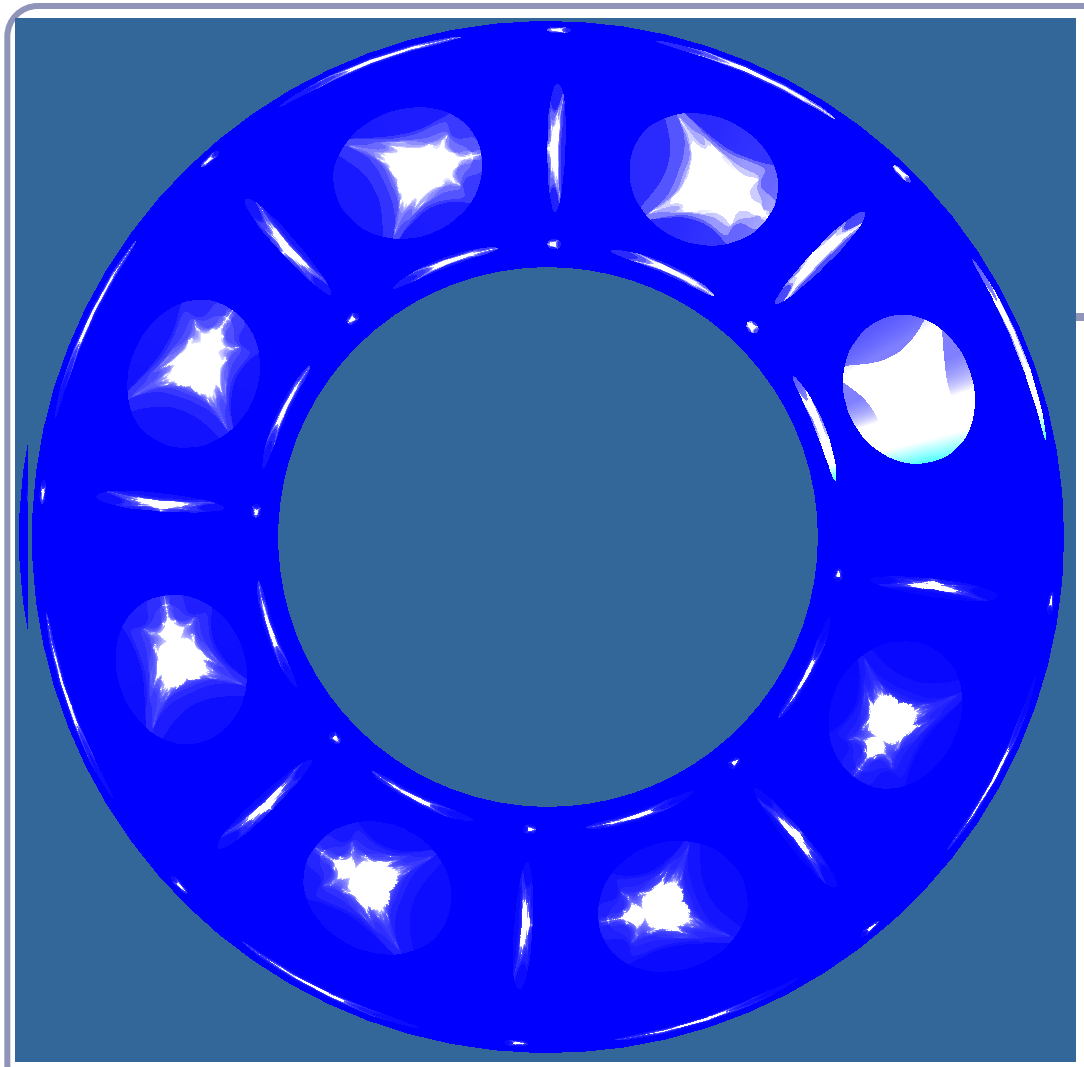
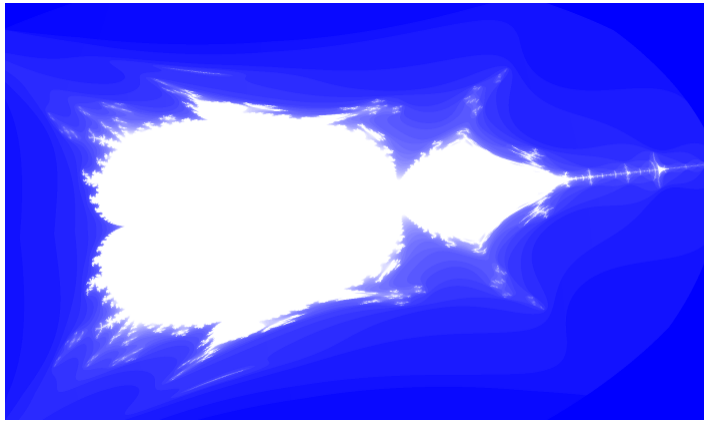
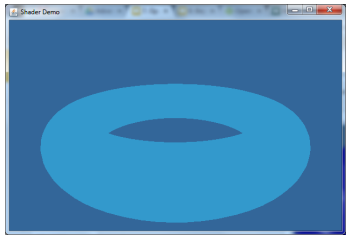
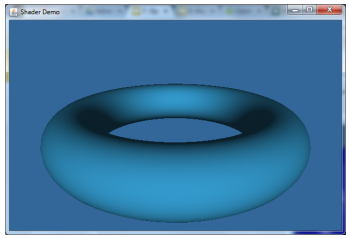
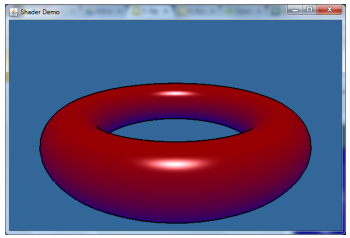


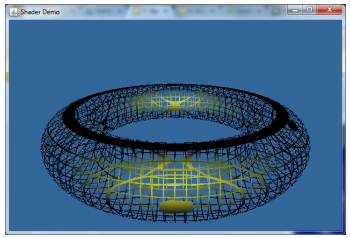
Advanced Graphics









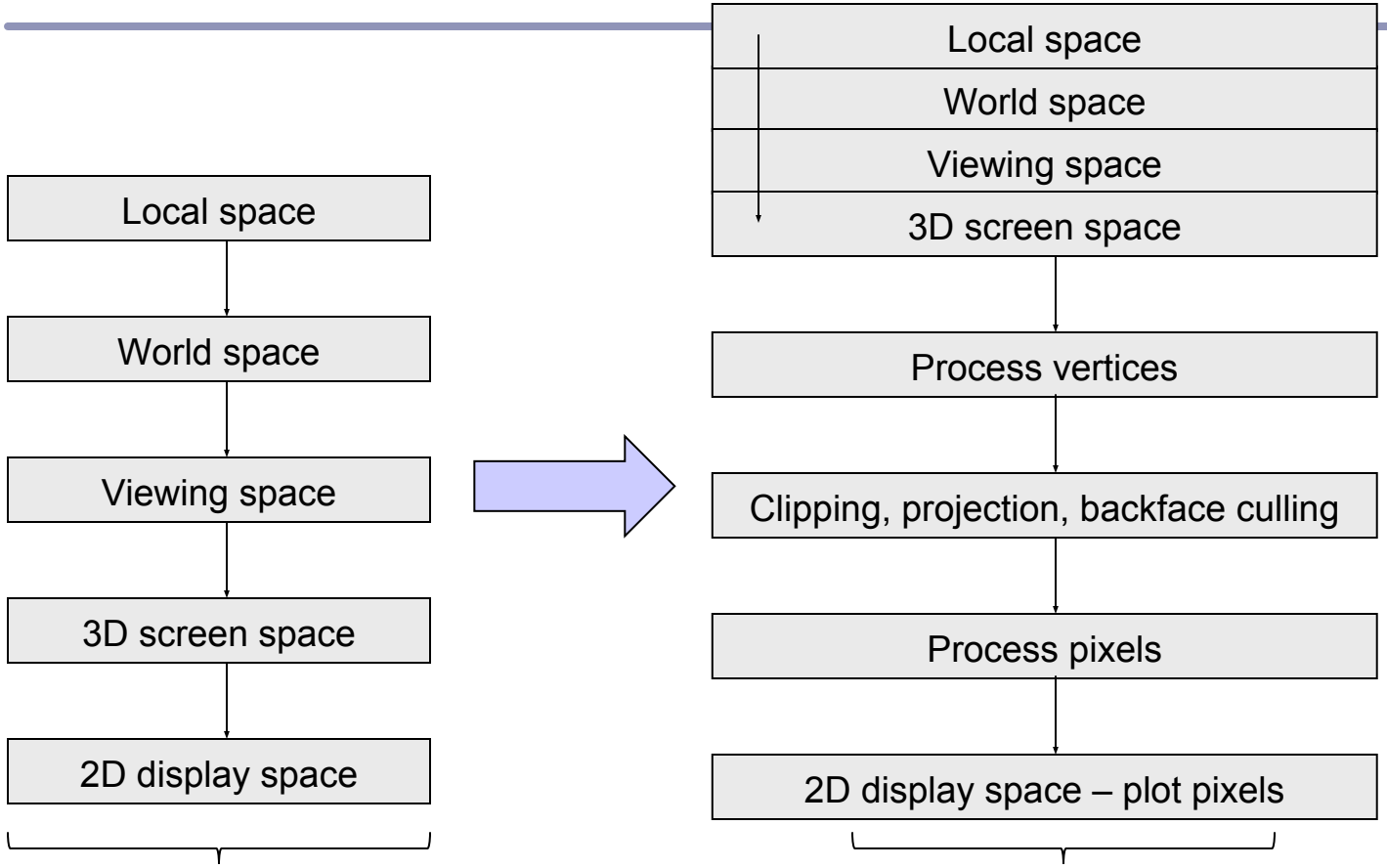


"The Shader knows…"

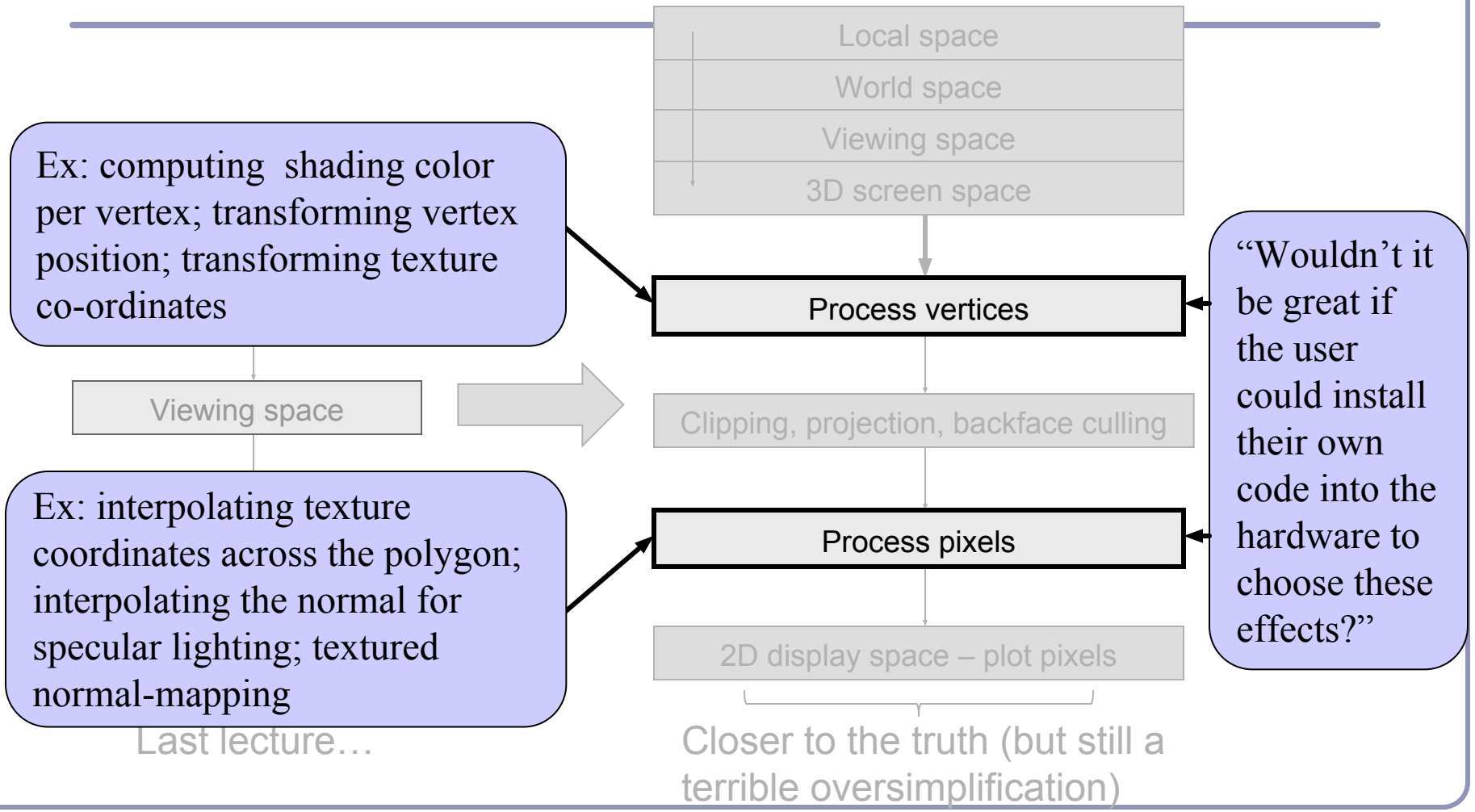
Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

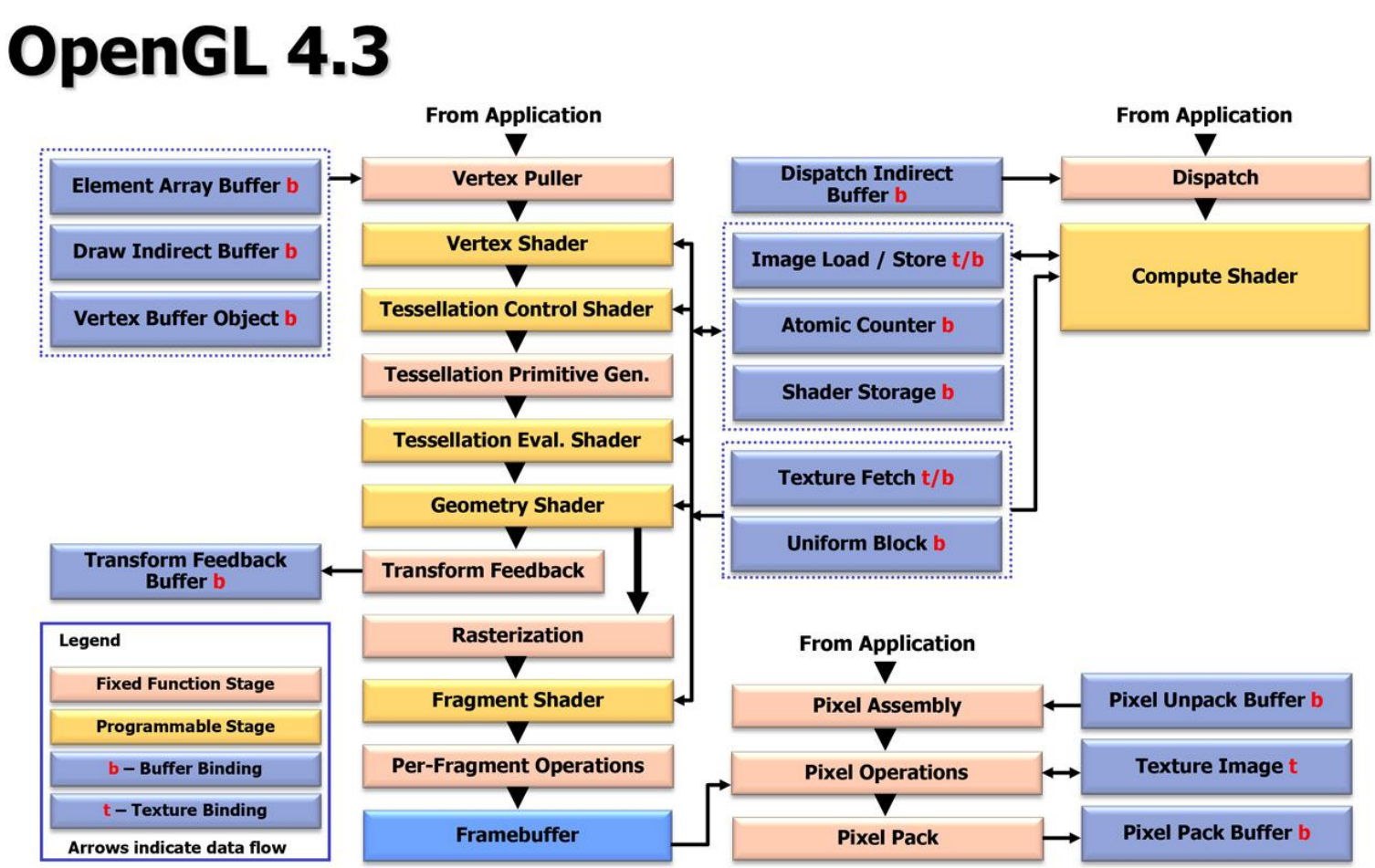
What is… the shader?



What is… the shader?



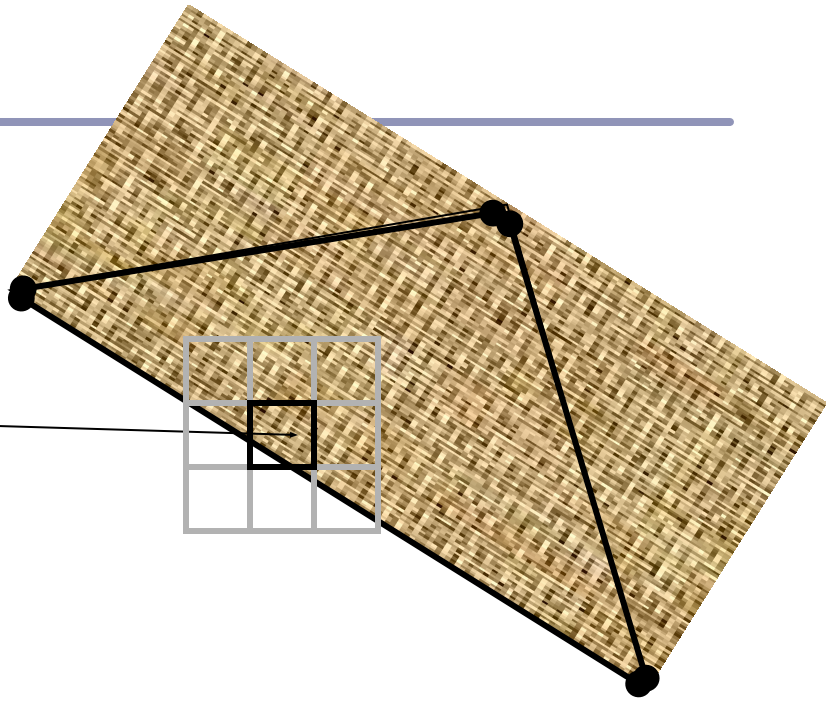
OpenGL programmable processors (not to scale)



The OpenGL 4.3 pipeline, circa 2012

What are we targeting?

OpenGL shaders give the user control over each *vertex* and each *fragment* (each pixel or partial pixel) interpolated between vertices.



After vertices are processed, polygons are *rasterized*. During rasterization, values like position, color, depth, and others are interpolated across the polygon. The interpolated values are passed to each pixel fragment.

Think parallel

Shaders are compiled from within your code

- They used to be written in assembler
- Today they're written in high-level languages

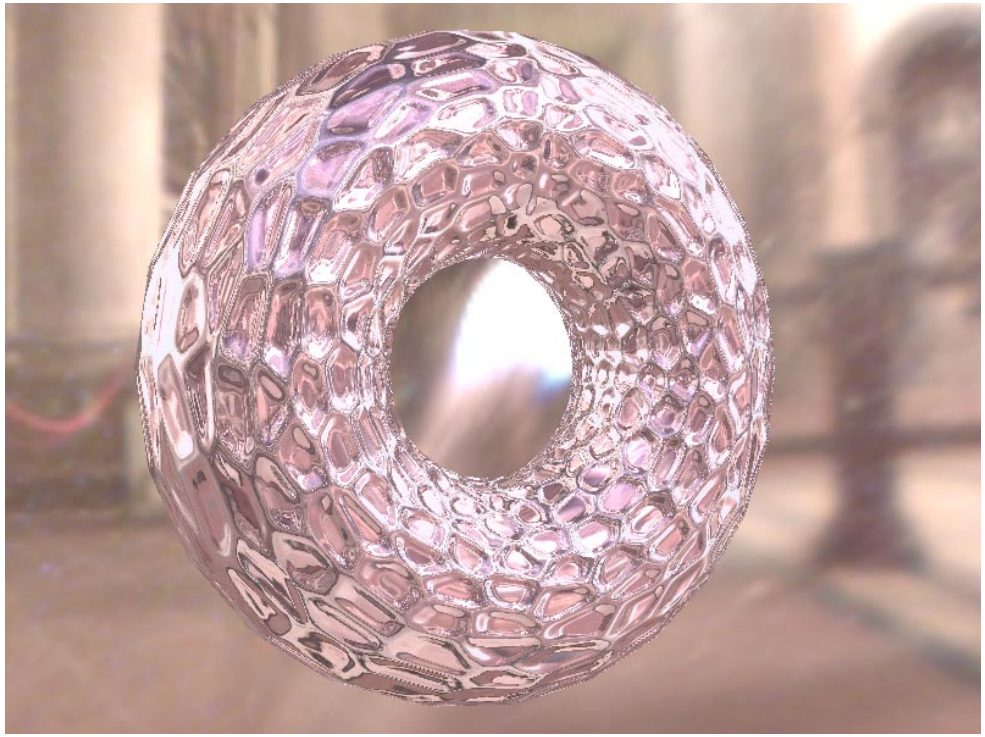
They execute on the GPU

GPUs typically have multiple processing units

That means that multiple shaders execute in parallel

- We're moving away from the purely-linear flow of early "C" programming models

Shader gallery I



Above: Demo of Microsoft's XNA game platform
Right: Product demos by nvidia (top) and ATI (bottom)

What're we talking here?

There are several popular languages for describing shaders, such as:

- *HLSL*, the *High Level Shading Language*
  - Author: Microsoft
  - DirectX 8+
- *Cg*
  - Author: nvidia
- *GLSL*, the *OpenGL Shading Language*
  - Author: the Khronos Group, a self-sponsored group of industry affiliates (ATI, 3DLabs, etc)

Least advanced; most portable and supported; topic of this lecture.

What can you control?

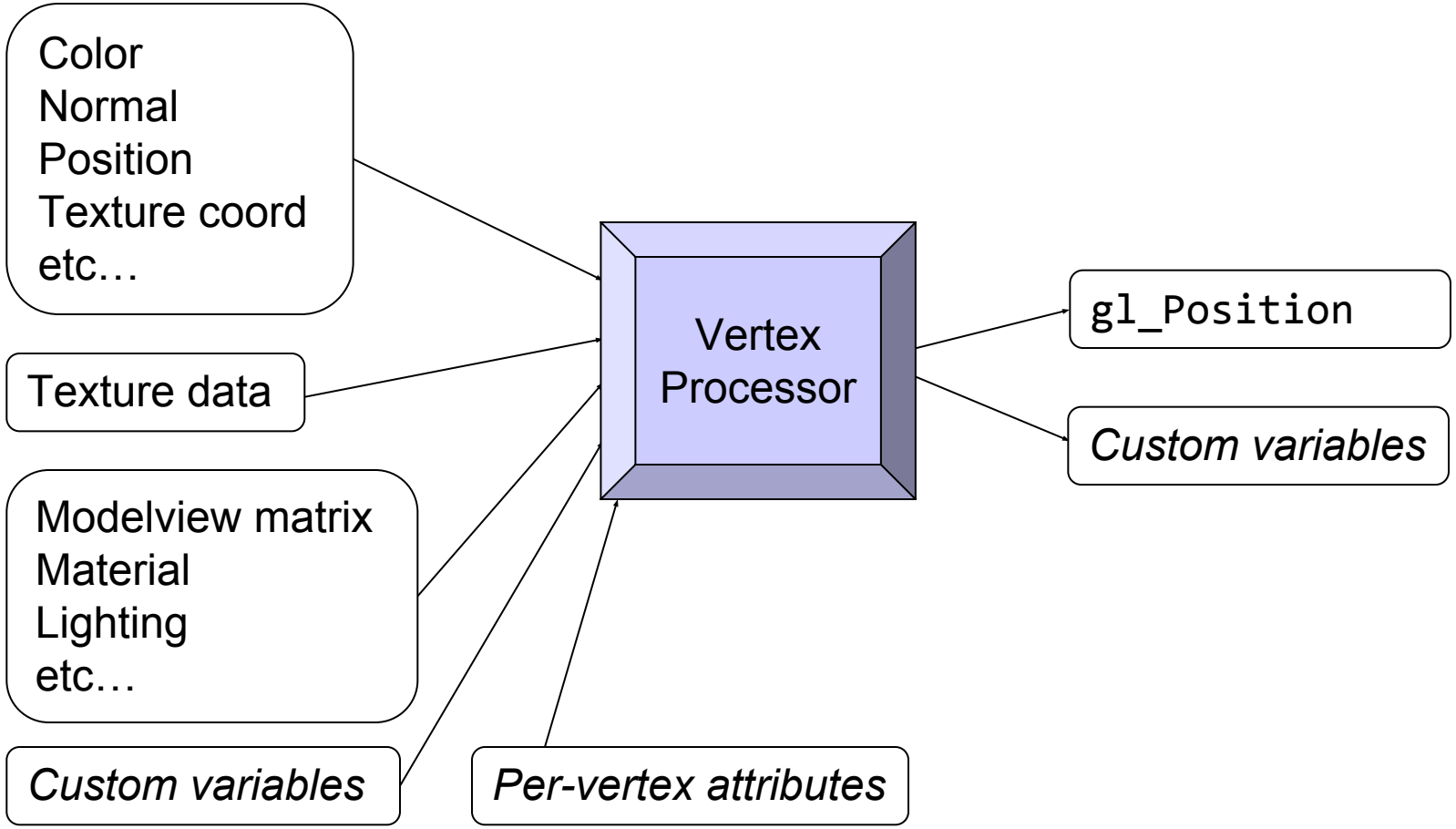
Per vertex:

- Vertex transformation
- Normal transformation and normalization
- Texture coordinate generation
- Texture coordinate transformation
- Lighting
- Color material application

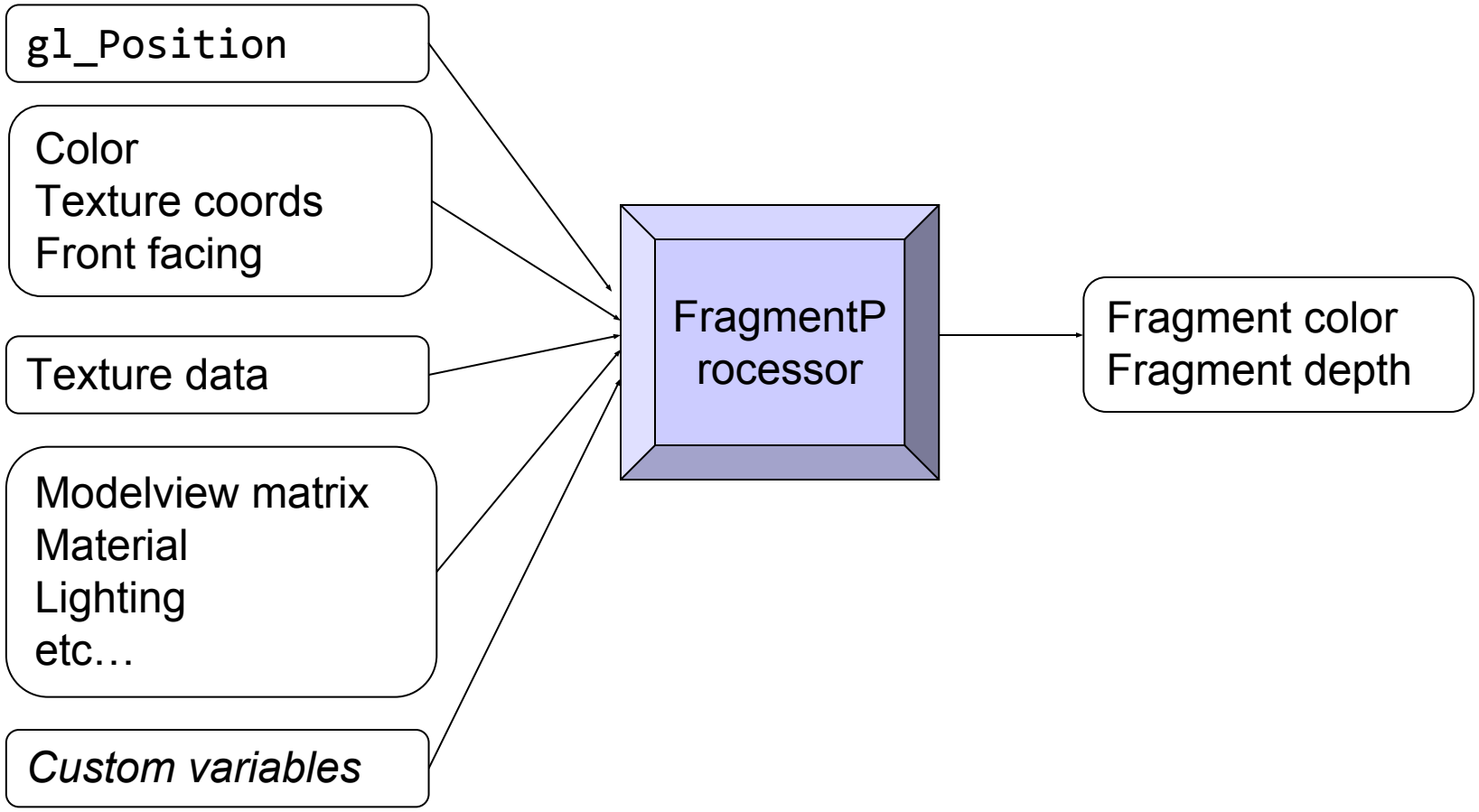
Per fragment (pixel):

- Operations on interpolated values
- Texture access
- Texture application
- Fog
- Color summation
- Optionally:
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution

Vertex processor – inputs and outputs



Fragment processor – inputs and outputs



How do the shaders communicate?

There are three types of shader parameter in GLSL:

uniform parameters

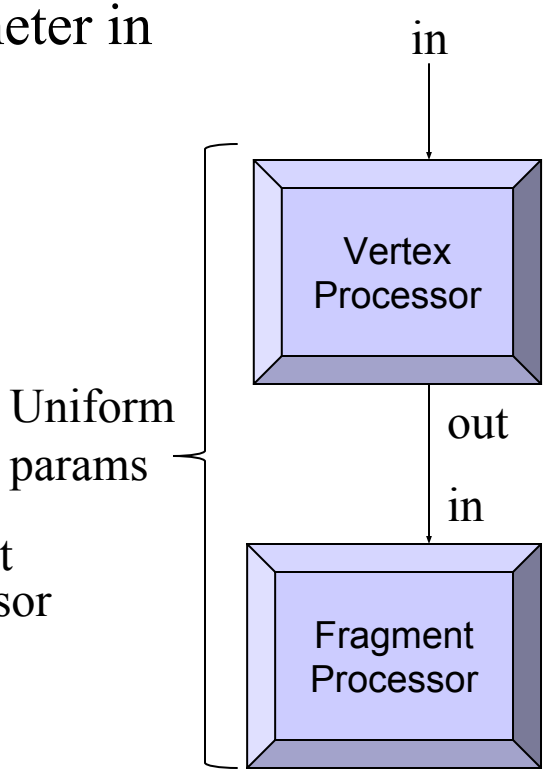
- Set throughout execution
- Ex: surface color

in parameters

- Inputs per vertex and per fragment
- Ex: local tangent

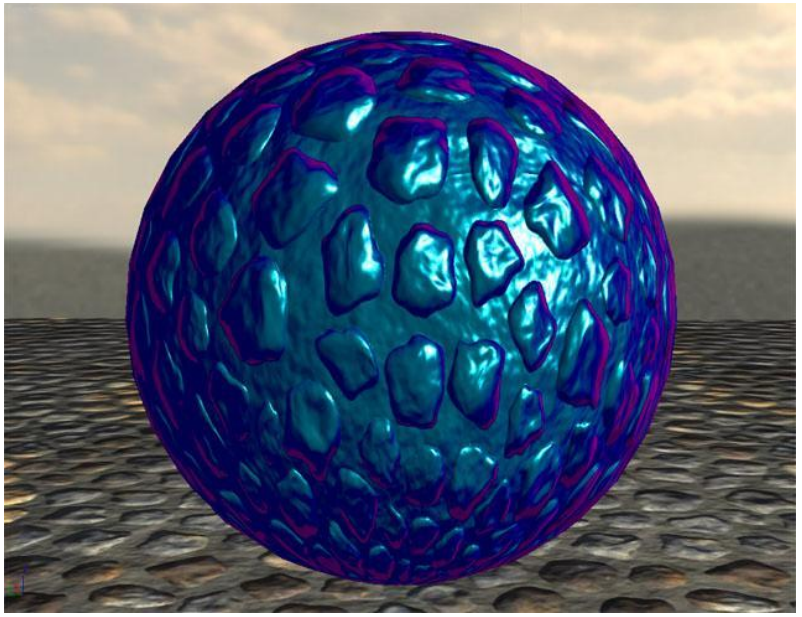
out parameters

- Passed from vertex processor to fragment processor, and out from fragment processor
- Ex: transformed normal



Shader gallery II

Above: Kevin Boulanger (PhD thesis, "*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*", 2005)



Above: Ben Cloward ("Car paint shader")

Shader sample one – ambient lighting

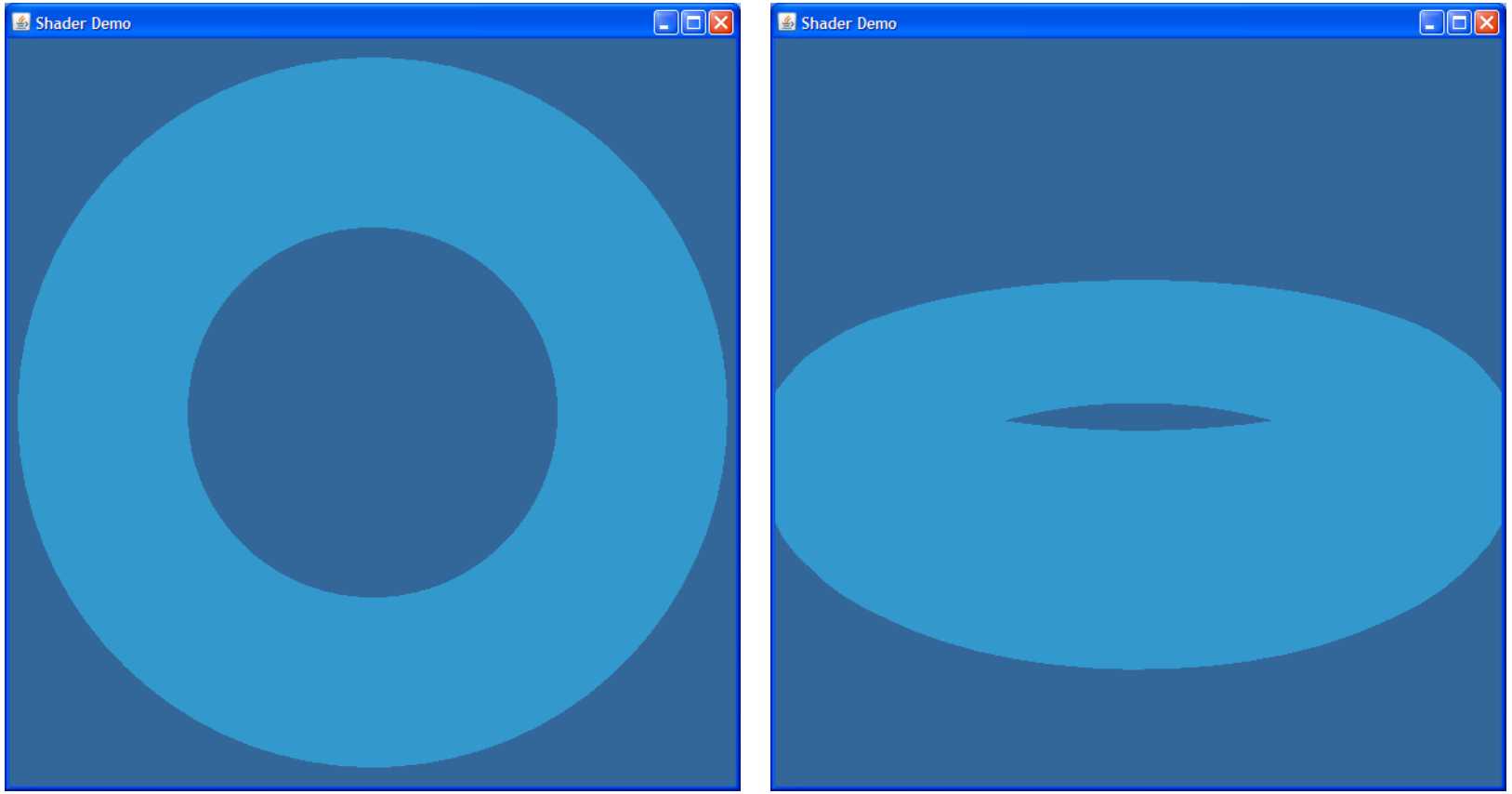
```
#version 330

uniform mat4 mvp;
in vec4 vPosition;

void main() {
  gl_Position =
      mvp * vPosition;
}
```

// Vertex Shader

```
#version 330



out vec4 color;

void main() {
  color =
     vec4(0.2, 0.6, 0.8, 1);
}
```

// Fragment Shader

Shader sample – ambient lighting



Shader sample – ambient lighting

Notice the C-style syntax

```
void main() { … }
```

The vertex shader uses two inputs, one four-element `vec4` and one four-by-four `mat4` matrix; and one standard output, `gl_Position`.

The line

```
gl_Position = mvp * gl_Vertex;
```

applies our model-view-projection matrix to calculate the correct vertex position in perspective coordinates.

The fragment shader applies basic ambient lighting, setting its one output, `color`, to a fixed value.

Shader sample – Phong shading

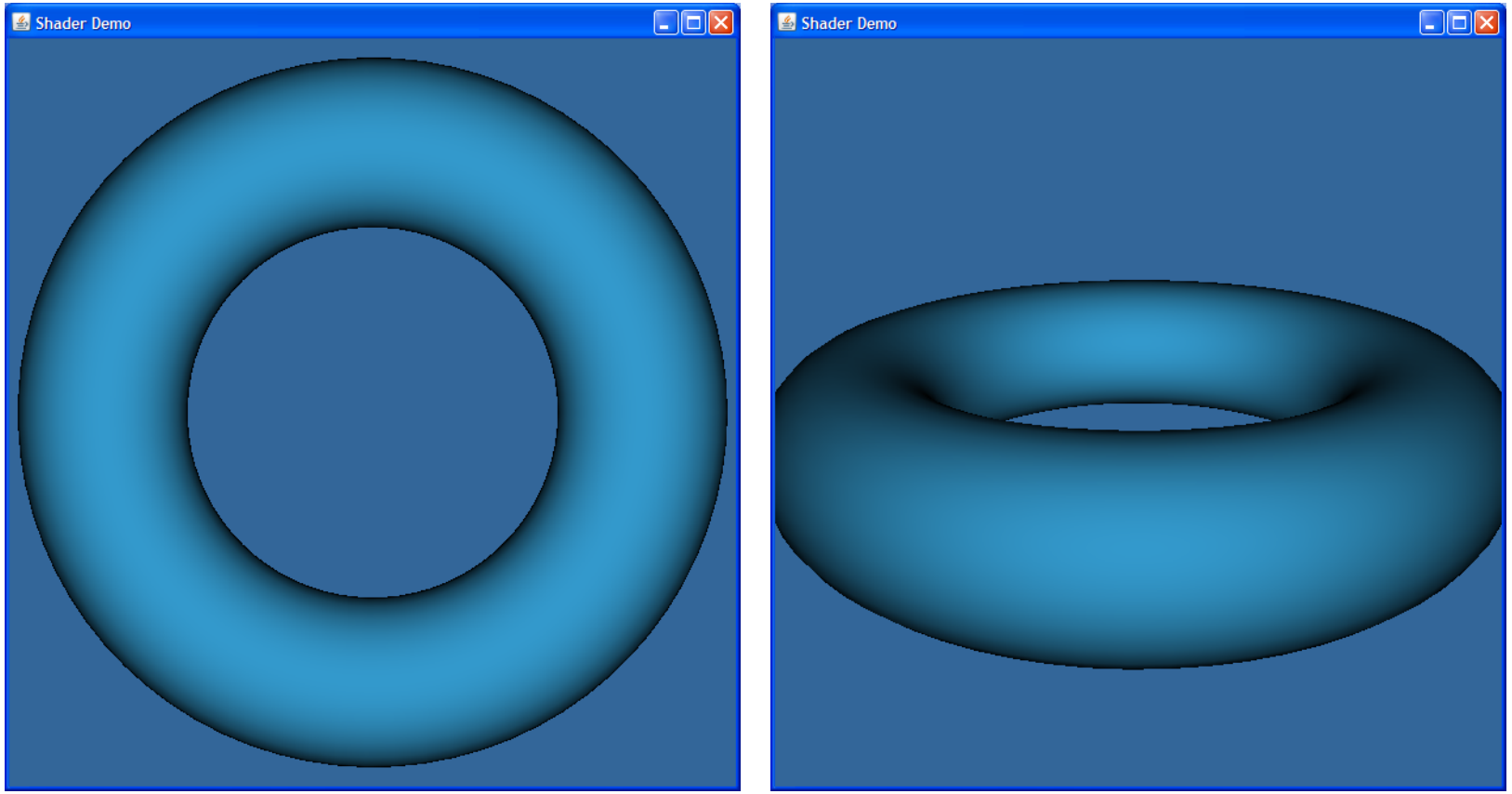
```
#version 330

uniform mat4 modelToScreen;
uniform mat3 normalToWorld;

in vec4 vPosition;
in vec3 vNormal;

out vec3 normal;

void main() {
  gl_Position =
      modelToScreen * vPosition;
  normal = normalize(
      normalToWorld * vNormal);
}
```

// Vertex Shader

```
#version 330

uniform vec3 lightDirection;

in vec3 normal;

out vec4 color;

vec3 blue = vec3(0.2, 0.6, 0.8);

void main() {
  vec3 n = normalize(normal);
  float diff = clamp(dot(n,
      lightDirection), 0.2, 1.0);
  color = vec4(blue * diff, 1.0);
}
```

// Fragment Shader

Shader sample – Phong shading



Shader sample – Phong shading

This examples uses *in* and *out* parameters to pass info from the vertex shader to the fragment shader, and *uniform* parameters to pass data from OpenGL to all shaders.

- The parameters `Norm` and `ToLight` are automatically linearly interpolated between vertices across every polygon.
- Each fragment shader sees the surface normal at that exact point on the surface.
- The exact illumination is computed locally.

Shader sample – Phong shading

Notice the different matrix transforms used in this example:

```
gl_Position = modelToScreen * vPosition;
Norm = normalToWorld * vNormal;
```

We defined different transforms because we needed different effects.

- `modelToScreen` transforms a vertex from local coordinates to perspective coordinates for display
- `normalToWorld` transforms a normal from local coordinates to world coordinates using the inverse of the transpose of the upper 3x3 submatrix of the model-view transform.

You'll need to always be aware of which coordinate space you need to transform from and to.

GLSL – design goals

GLSL was designed with the following in mind:

- Work well with OpenGL
  - Shaders should be optional extras, not required.
  - Fit into the design model of "set the state first, then render the data in the context of the state"
- Support upcoming flexibility
- Be hardware-independent
  - The GLSL folks, as a broad consortium, are far more invested in hardware-independence than, say, nvidia.
  - That said, they've only kinda nailed it: I get different compiler behavior and different crash-handling between my high-end home nVidia chip and my laptop Intel x3100.
- Support inherent parallelization
- Keep it streamlined, small and simple

GLSL

The language design in GLSL is strongly based on ANSI C, with some C++ added.

- There is a preprocessor--**#define**, etc!
- Basic types: int, float, bool
  - No double-precision float
- Vectors and matrices are standard: **vec2**, **mat2** = 2x2; **vec3**, **mat3** = 3x3; **vec4**, **mat4** = 4x4
- Texture samplers: **sampler1D**, **sampler2D**, etc are used to sample multidemensional textures
- New instances are built with *constructors*, a la C++
- Functions can be declared before they are defined, and operator overloading is supported.

GLSL

Some differences from C/C++:

- No pointers, strings, chars; no unions, enums; no bytes, shorts, longs; no unsigned. No switch() statements.
- There is no implicit casting (type promotion):

```
float foo = 1;
```

  fails because you can't implicitly cast **int** to **float**.

- Explicit type casts are done by constructor:

```
vec3 foo = vec3(1.0, 2.0, 3.0);
vec2 bar = vec2(foo);  // Drops foo.z
```

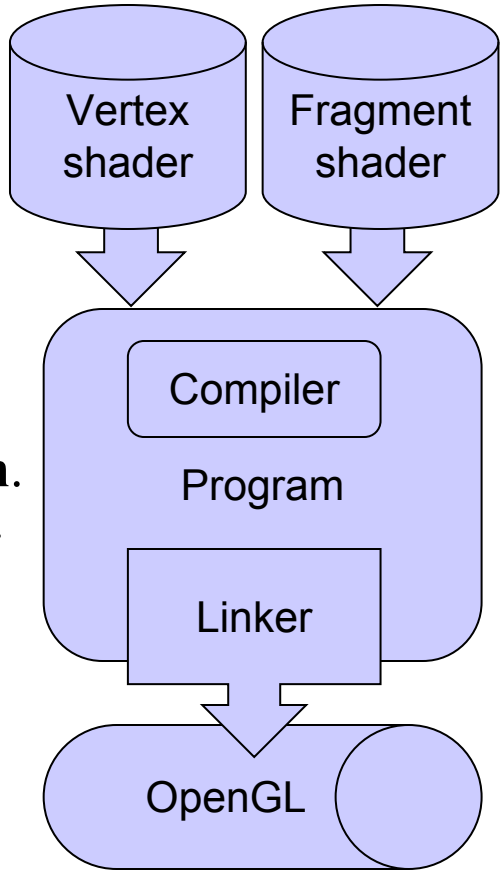
Function parameters are labeled as **in**, **out**, or **uniform**.

- Functions are called by *value-return*, meaning that values are copied into and out of parameters at the start and end of calls.

The OpenGL GLSL API

To install and use a shader in OpenGL:

- Create one or more empty *shader objects* with **glCreateShader**.
- Load source code, in text, into the shader with **glShaderSource**.
- Compile the shader with **glCompileShader**.
  - The compiler cannot detect every program that would cause a crash.
    (And if you can prove otherwise, see me after class.)
- Create an empty *program object* with **glCreateProgram**.
- Bind your shaders to the program with **glAttachShader**.
- Link the program (ahh, the ghost of C!) with **glLinkProgram**.
- Register your program for use with **glUseProgram**.



Tips for debugging OpenGL / GLSL

- Use *glGetError* to check the current state of OpenGL:

```
int error = gl.glGetError();
if (error != 0) {
  throw new RuntimeException("OpenGL Error " + error
      + ": " + (new GLU()).gluErrorString(error));
}
```

- Use *GL_VERSION* to check what version of GL your graphics chip is capable of running:

```
gl.glGetString(GL.GL_VERSION)
```

Tips for debugging OpenGL / GLSL

- Use *glGetShaderInfoLog* to validate a compiled shader

```
int[] length = new int[1];
byte[] buffer = new byte[BUFFER_SIZE];
gl.glGetShaderInfoLog(shader, BUFFER_SIZE, length, 0,
buffer, 0);
int logLength = length[0];
if (logLength > 1) {
  String infolog = new String(buffer);
  if (infolog.trim().compareTo("No errors.") != 0) {
    throw new RuntimeException(shader + ": " + infolog);
  }
}
```

Tips for debugging OpenGL / GLSL

- Use *glValidateProgram* and *glGetProgramInfoLog* to validate a compiled and linked GLSL program

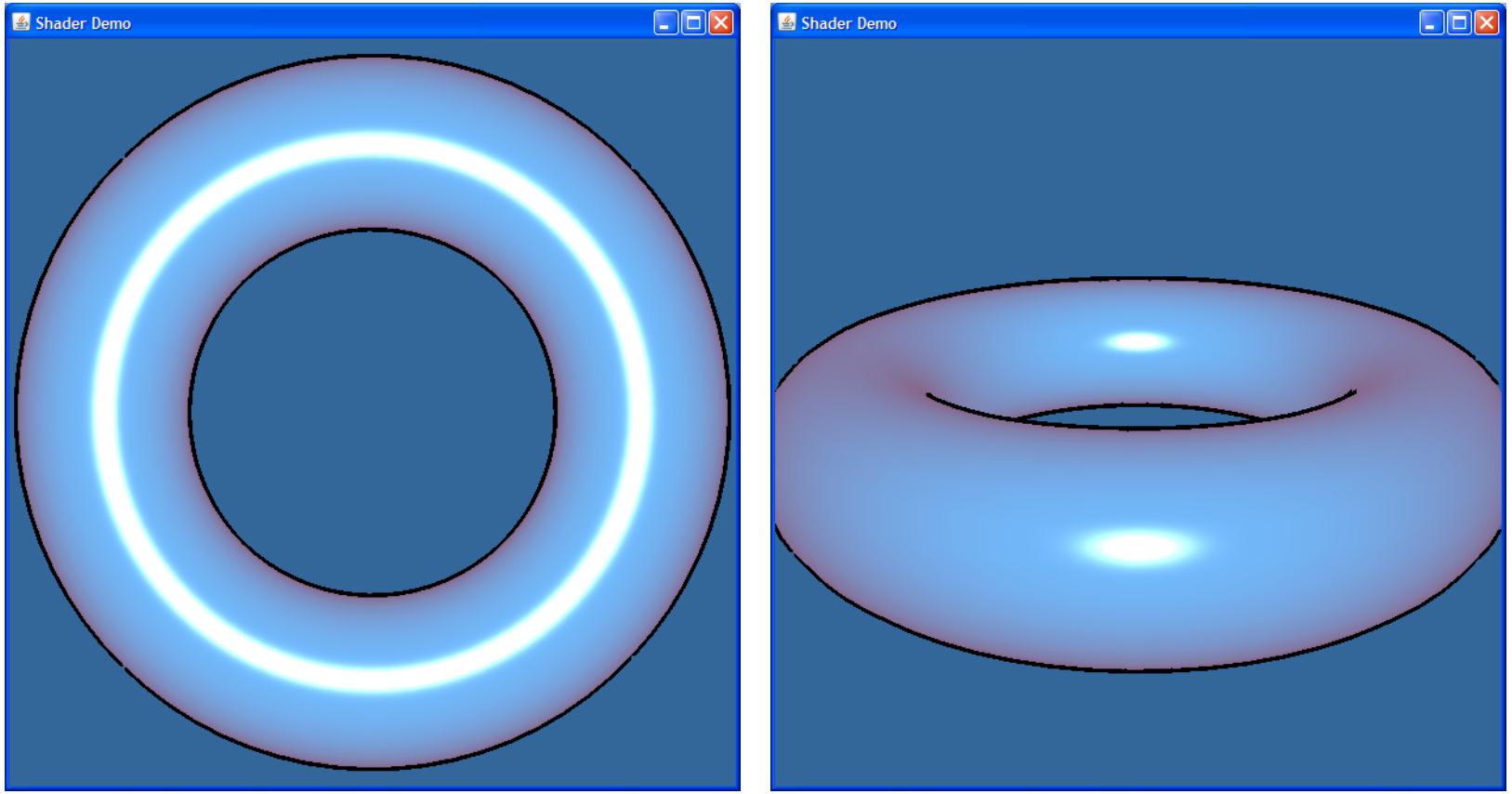
```
gl.glValidateProgram(program);
int[] length = new int[1];
byte[] buffer = new byte[BUFFER_SIZE];
gl.glGetProgramInfoLog(
    program, BUFFER_SIZE, length, 0, buffer, 0);
if (length[0] > 1) {
  String infolog = new String(buffer).trim();
  if (!infolog.equalsIgnoreCase("No errors.")) {
    throw new RuntimeException(
        "Program: " + program + ": " + infolog);
  }
}
```

Shader sample – Gooch shading

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform mat4 modelToCamera;
uniform mat4 modelToScreen;
uniform mat3 normalToCamera;

vec3 LightPosition = vec3(0, 10, 4);

in vec4 vPosition;
in vec3 vNormal;

out float NdotL;
out vec3  ReflectVec;
out vec3  ViewVec;

void main()
{
  vec3 ecPos      = vec3(modelToCamera * vPosition);
  vec3 tnorm      = normalize(normalToCamera * vNormal);
  vec3 lightVec   = normalize(LightPosition - ecPos);
  ReflectVec      = normalize(reflect(-lightVec, tnorm));
  ViewVec         = normalize(-ecPos);
  NdotL           = (dot(lightVec, tnorm) + 1.0) * 0.5;
  gl_Position     = modelToScreen * vPosition;
}
```

```
#version 330

// Original author: Randi Rost
// Copyright (c) 2002-2005 3Dlabs Inc. Ltd.

uniform vec3 vColor;

float DiffuseCool = 0.3;
float DiffuseWarm = 0.3;
vec3 Cool = vec3(0, 0, 0.6);
vec3 Warm = vec3(0.6, 0, 0);

in float NdotL;
in vec3  ReflectVec;
in vec3  ViewVec;

out vec4 result;

void main()
{
  vec3 kcool  = min(Cool + DiffuseCool * vColor, 1.0);
  vec3 kwarm  = min(Warm + DiffuseWarm * vColor, 1.0);
  vec3 kfinal = mix(kcool, kwarm, NdotL);

  vec3 nRefl  = normalize(ReflectVec);
  vec3 nview  = normalize(ViewVec);
  float spec  = pow(max(dot(nRefl, nview), 0.0), 32.0);

  if (gl_FrontFacing) {
    result = vec4(min(kfinal + spec, 1.0), 1.0);
  } else {
    result = vec4(0, 0, 0, 1);
  }
}
```

Shader sample – Gooch shading



Shader sample – Gooch shading

Gooch shading is an example of *non-realistic rendering*. It was designed by Amy and Bruce Gooch to replace photorealistic lighting with a lighting model that highlights structural and contextual data.

- They use the term of the conventional lighting equation to choose a map between 'cool' and 'warm' colors.
  - This is in contrast to conventional illumination where lighting simply scales the underlying surface color.
- This, combined with edge-highlighting through a second renderer pass, creates models which look more like engineering schematic diagrams.

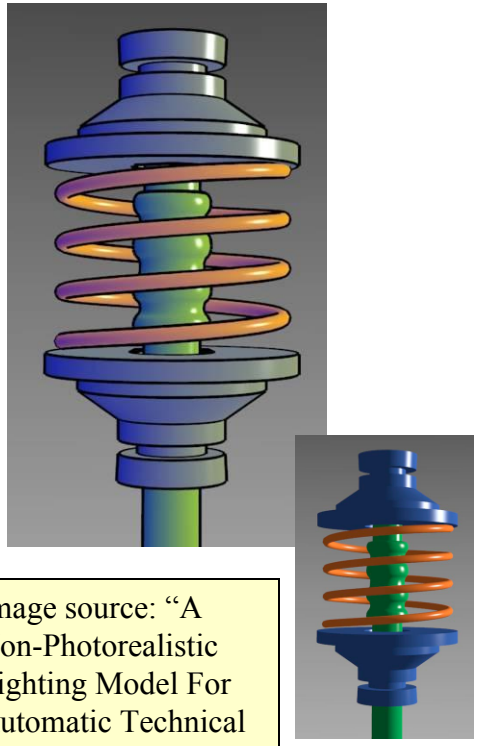


Image source: "A Non-Photorealistic Lighting Model For Automatic Technical Illustration", Gooch, Gooch, Shirley and Cohen (1998). Compare the Gooch shader, above, to the Phong shader (right).

Shader sample – Gooch shading

In the vertex shader source, notice the use of the built-in ability to distinguish front faces from back faces:

```
if (gl_FrontFacing) {...
```

This supports distinguishing front faces (which should be shaded smoothly) from the edges of back faces (which will be drawn in heavy black.)

In the fragment shader source, this is used to choose the weighted color by clipping with the `a` component:

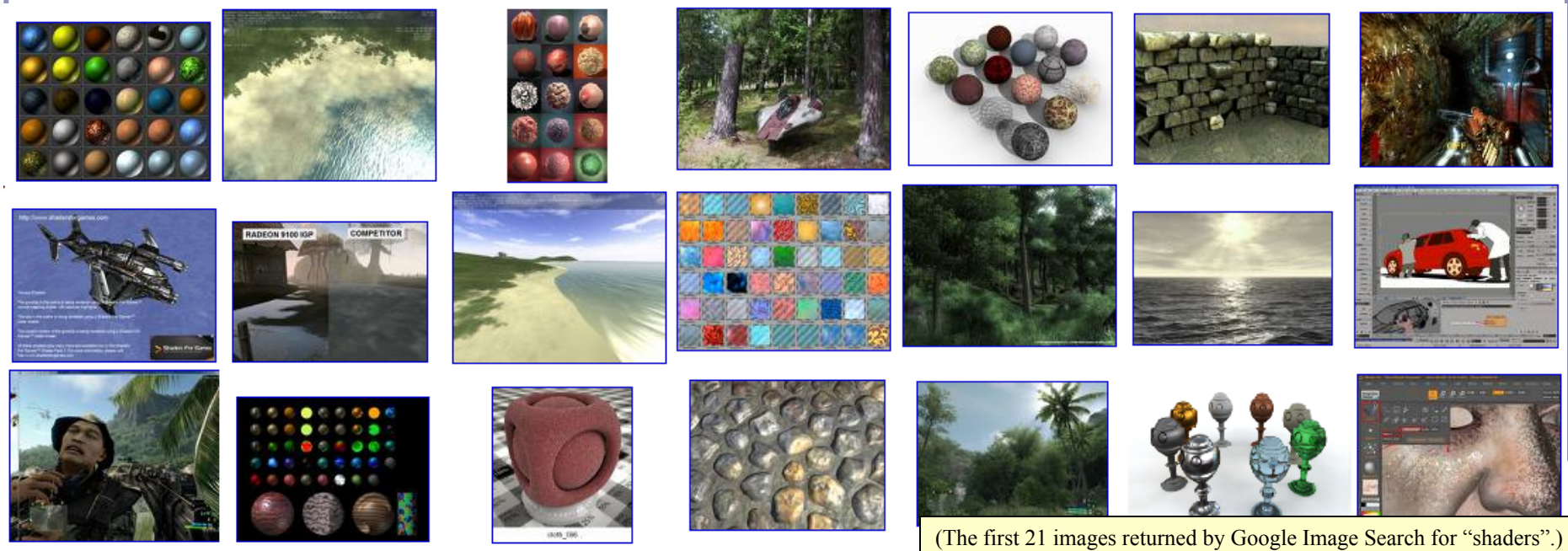
```
vec3 kfinal = mix(kcool, kwarm, NdotL);
```

Here `mix()` is a GLSL method which returns the linear interpolation between `kcool` and `kwarm`. The weighting factor is `NdotL`, the lighting value.

Demo!

Recap

- Shaders give a powerful, extensible mechanism for programming the vertex and pixel processing stages of the GPU pipeline.
- GLSL is a portable, multiplatform C-like language which is compiled at run-time and linked into an executable shader program.
- Shaders can be used for a long list of effects, from procedural geometry and non-photorealistic lighting to advanced textures, fog, shadows, raycasting, and visual effects; in fact, many of the topics covered in this course!



(The first 21 images returned by Google Image Search for “shaders”.)