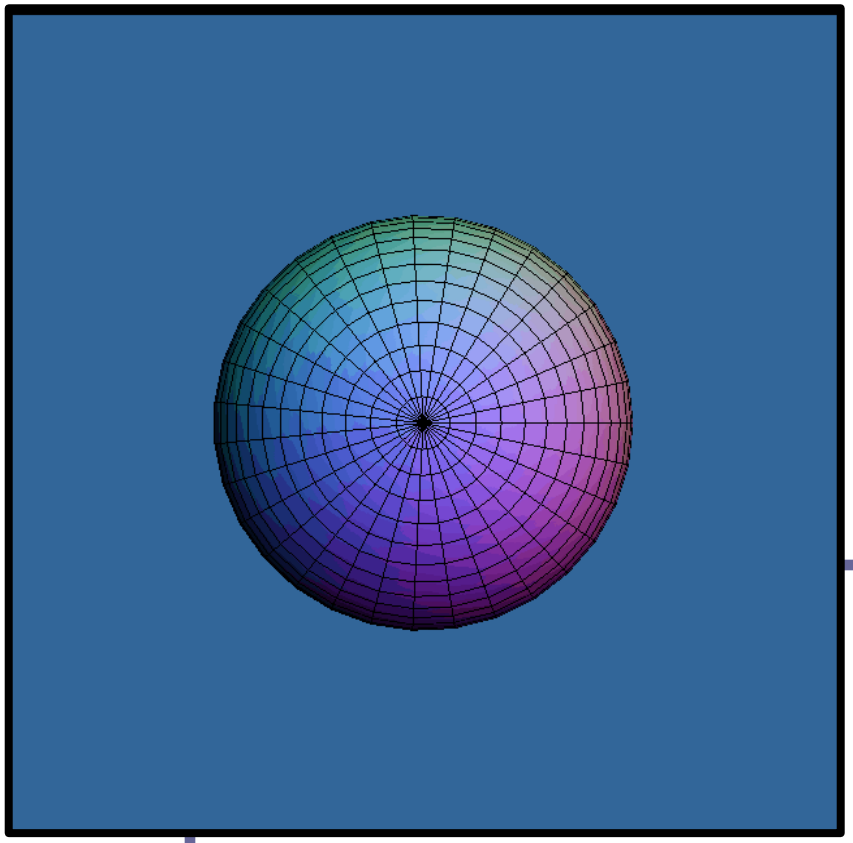
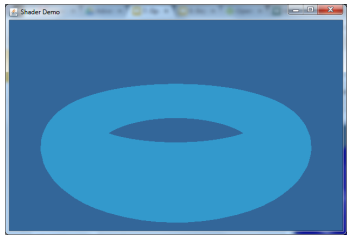
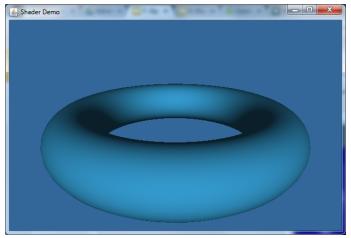
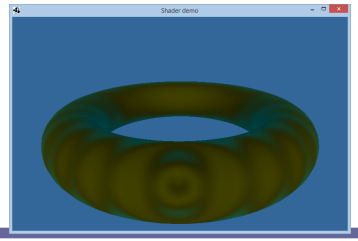
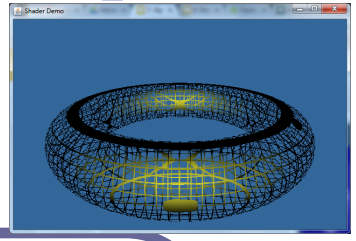




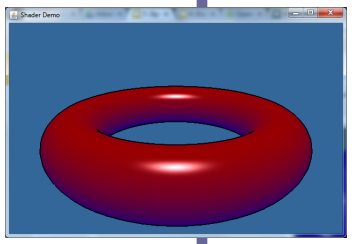


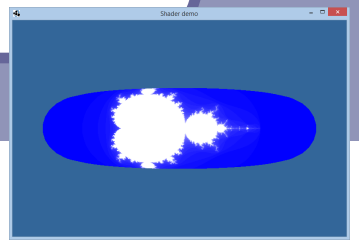
# Advanced Graphics





# OpenGL and Shaders





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

Supported in part by Google UK, Ltd

# 3D technologies today

Java

- Common, re-usable language; well-designed
- Steadily increasing popularity in industry
- Weak but evolving 3D support

C++

- Long-established language
- Long history with OpenGL
- Long history with DirectX
- Losing popularity in some fields (finance, web) but still strong in others (games, medical)

JavaScript

- WebGL is surprisingly popular

OpenGL

- Open source with many implementations
- Well-designed, old, and still evolving
- Fairly cross-platform

DirectX/Direct3d (Microsoft)

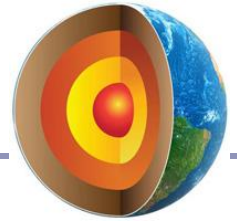
- Microsoft™ only
- Dependable updates


Mantle (AMD)

- Targeted at game developers
- AMD-specific

Higher-level commercial libraries

- RenderMan
- AutoDesk / SoftImage



# OpenGL


OpenGL is…

- Hardware-independent
- Operating system independent
- Vendor neutral

On many platforms

- Great support on Windows, Mac, linux, etc
- Support for mobile devices with OpenGL ES
  - Android, iOS (but not Windows Phone)
  - Android Wear watches!
- Web support with WebGL

A state-based renderer

- many settings are configured before passing in data; rendering behavior is modified by existing state

Accelerates common 3D graphics operations

- Clipping (for primitives)
- Hidden-surface removal (Z-buffering)
- Texturing, alpha blending NURBS and other advanced primitives (GLUT)

# OpenGL in Java

- *JOGL*: "Java bindings for OpenGL"

http://jogamp.org/jogl/

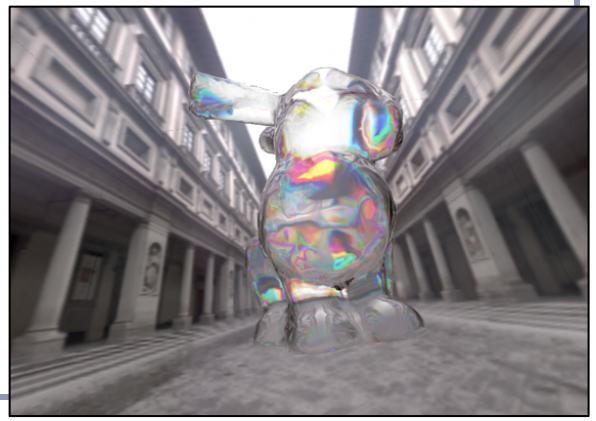
JOGL apps can be deployed as applications or as *applets*, making it suitable for educational web demos and cross-platform applications.

- If the user has installed the latest Java, of course.
- And if you jump through Oracle's authentication hoops.
- And… let's be honest, 1998 called, it wants its applets back.

- *LWJGL*: "Lightweight Java Games Library"

http://www.lwjgl.org/

LWJGL is targeted at game developers, so it's got a really solid threading model and good support for new input methods like joysticks, gaming mice, and the Oculus Rift.



*JOGL shaders in action.*
*Image from Wikipedia*

# OpenGL architecture

The CPU (your processor and friend) delivers data to the GPU (Graphical Processing Unit).

- The GPU takes in streams of vertices, colors, texture coordinates and other data; constructs polygons and other primitives; then uses *shaders* to draw the primitives to the screen pixel-by-pixel.
- The GPU processes the vertices according to the *state* set by the CPU; for example, “every trio of vertices describes a triangle”.

This process is called the *rendering pipeline*. Implementing the rendering pipeline is a joint effort between you and the GPU.

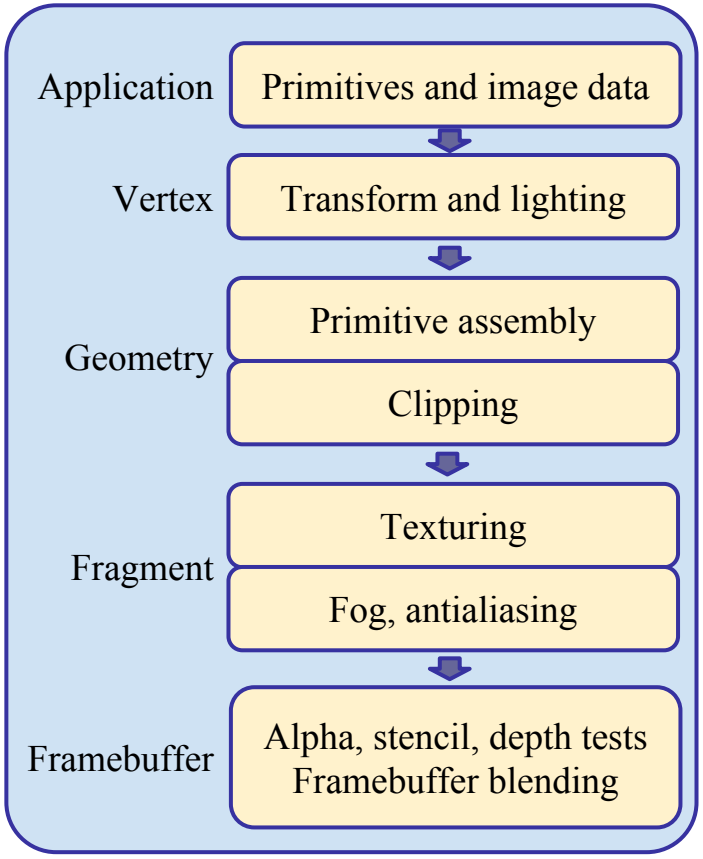
You’ll write shaders in the OpenGL shader language, GLSL.

You’ll write *vertex* and *fragment* shaders. (And maybe others.)

# The OpenGL rendering pipeline

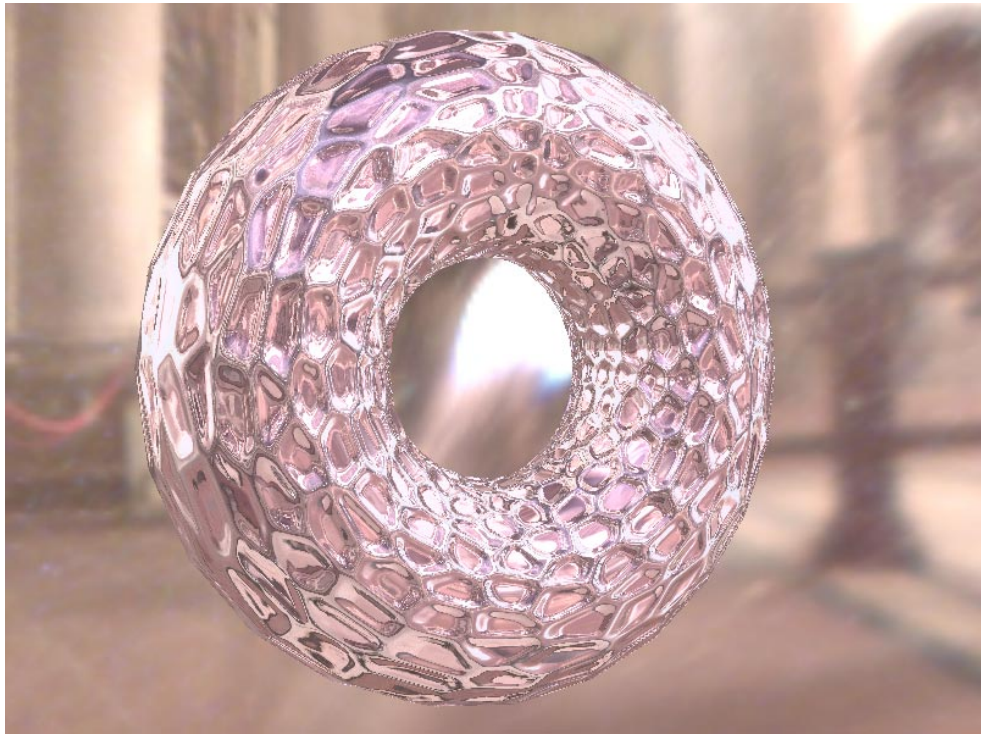
An OpenGL application assembles sets of *primitives*, *transforms* and *image data*, which it passes to OpenGL's GLSL shaders.

- *Vertex shaders* process every vertex in the primitives, computing info such as position of each one.
- *Fragment shaders* compute the color of every fragment of every pixel covered by every primitive.




The OpenGL *rendering pipeline*
*(simplified view)*

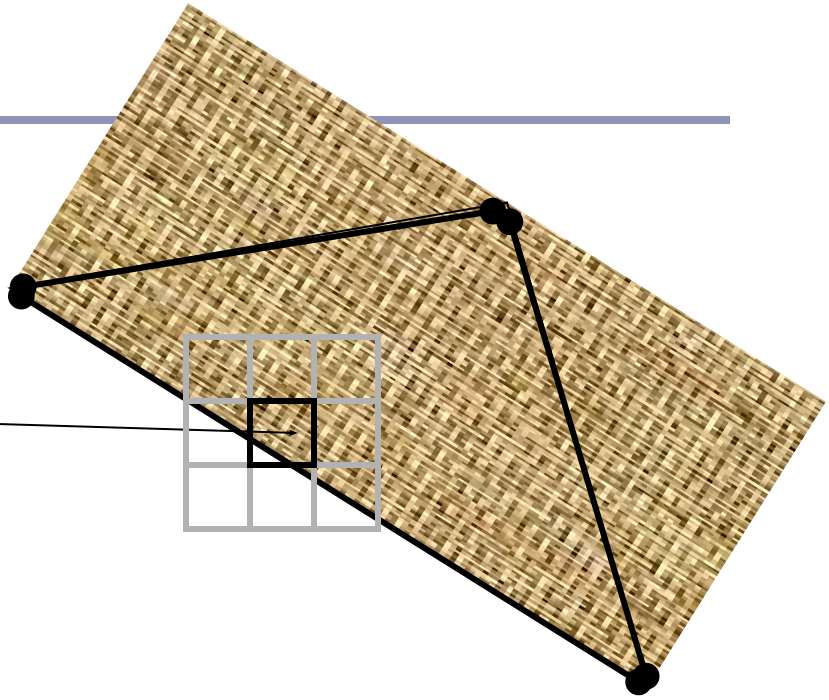
# Shader gallery I




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

# 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

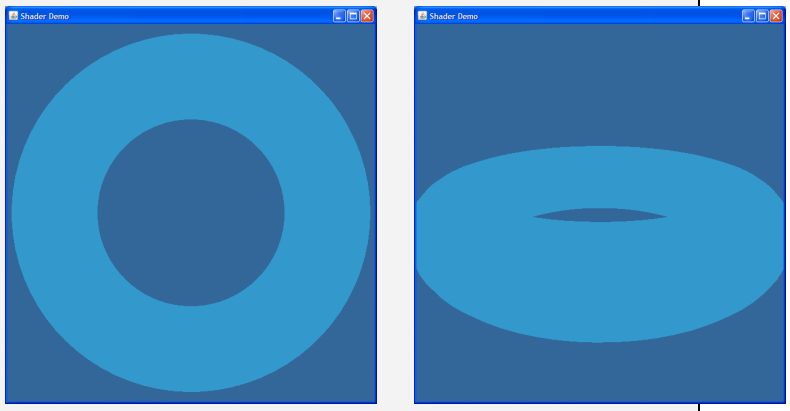
- We're moving away from the purely-linear flow of early "C" programming models

# Shader example one – ambient lighting

```
#version 330

uniform mat4 mvp;

in vec4 vPosition;

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

// Vertex Shader

```
#version 330

out vec4 fragmentColor;

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




// Fragment Shader

# GLSL

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.

This fragment shader implements the most basic ambient lighting by setting its one output, `col`, to a fixed value.

# 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
```

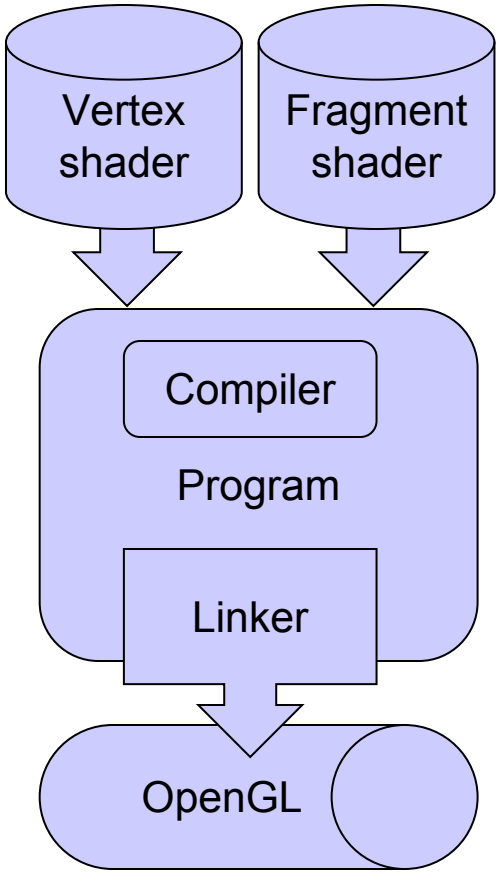
Function parameters are labeled as **in**, **out**, or **uniform**.

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

# OpenGL / GLSL API - setup

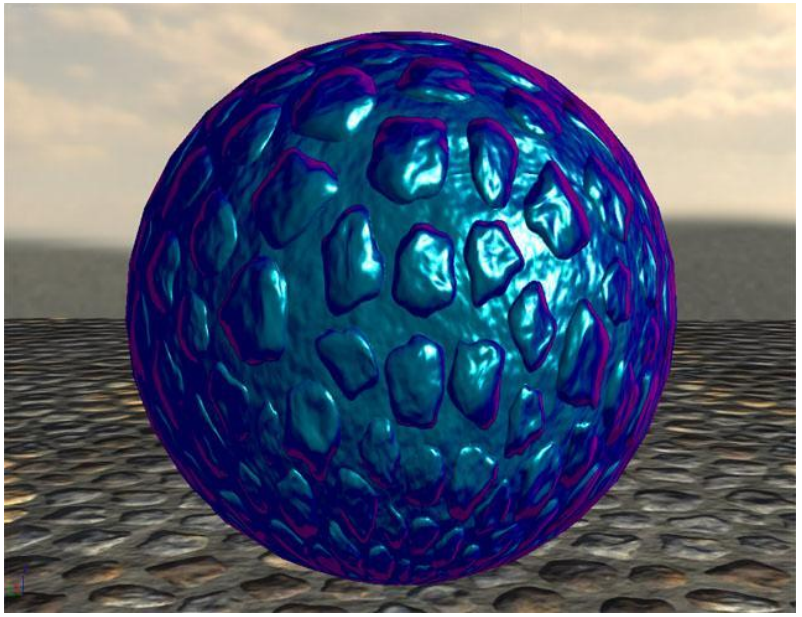
To install and use a shader in OpenGL:

1. Create one or more empty *shader objects* with **glCreateShader**.
2. Load source code, in text, into the shader with **glShaderSource**.
3. Compile the shader with **glCompileShader**.
4. Create an empty *program object* with **glCreateProgram**.
5. Bind your shaders to the program with **glAttachShader**.
6. Link the program (ahh, the ghost of C!) with **glLinkProgram**.
7. Activate your program with **glUseProgram**.



# Shader gallery II

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

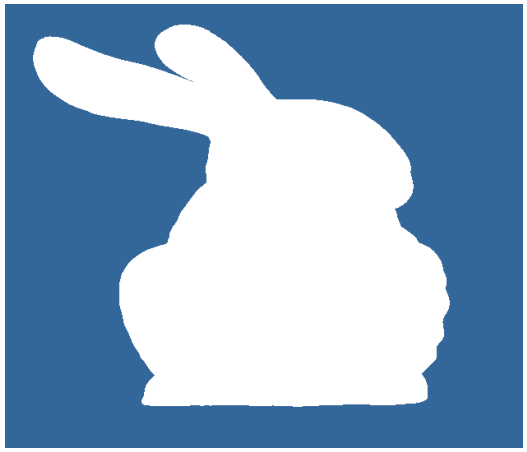
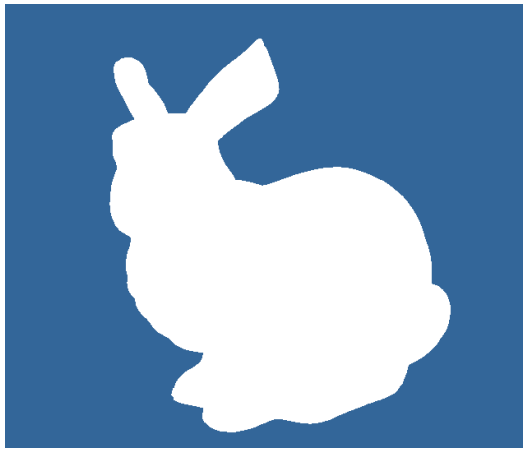
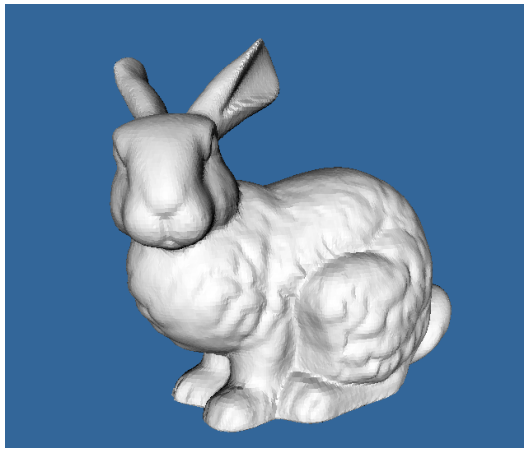


Above: Ben Cloward ("Car paint shader")

# What will you have to write?

It's up to you to implement perspective and lighting.

1. Pass geometry to the GPU
2. Implement perspective on the GPU
3. Calculate lighting on the GPU

# 1. OpenGL / GLSL API - variables

GLSL shaders use named parameters which can be looked up from OpenGL.

GLSL

```
uniform mat4 modelToScreen;
in vec4 vPosition;
...
```

The OpenGL API looks up the location integers of these parameters and uses the location as an address:

OpenGL

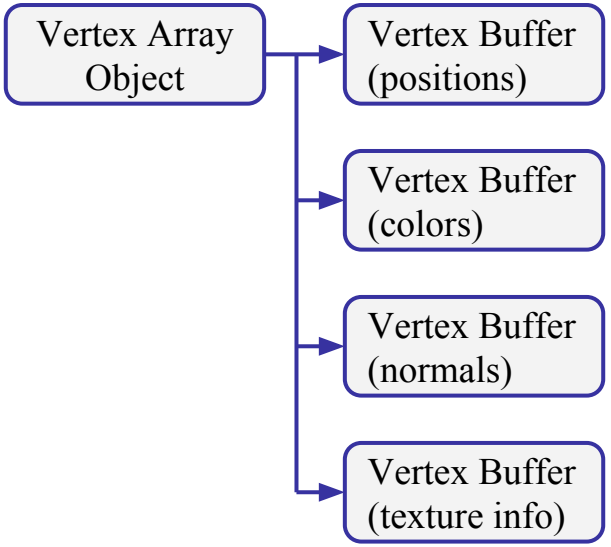
```
int attributeId = glGetAttribLocation(program,
"vPosition");
glEnableVertexAttribArray(attributeId);
glVertexAttribPointer(attributeId, ...);
```

# Passing geometry to OpenGL

V*ertex buffer objects* store arrays of vertex data--positional or descriptive. With a vertex buffer object ("VBO") you can compute all vertices at once, pack them into a VBO, and pass them to OpenGL *en masse* to let the GPU processes all the vertices together.

To group different kinds of vertex data together, you can serialize your buffers into a single VBO, or you bind and attach them to a *Vertex Array Objects*. Each vertex array object ("VAO") can contain multiple VBOs.

Although not required, VAOs help you to organize and isolate the data in your VBOs.



# Vertex arrays contain vertex buffers

First, we allocate a *vertex array*:

```
private void createAndBindVertexBuffer() {
  int vertexArrayId = glGenVertexArrays();
  glBindVertexArray(vertexArrayId);
}
```

Then we fill attach a *vertex buffer* with vertex coordinates:

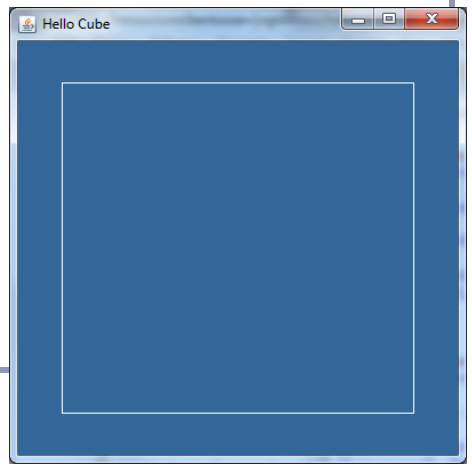
```
private void addVertexBuffer(String name, FloatBuffer data) {
  int BufferId = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, bufferId);
  glBufferData(GL_ARRAY_BUFFER, data, GL_STATIC_DRAW);
  int attributeId = glGetAttribLocation(program, name);
  glEnableVertexAttribArray(attributeId);
  glVertexAttribPointer(attributeId, 3, GL11.GL_FLOAT, false, 0, 0);
}
```

# Vertex buffers contain vertex data

In Java, vertex data is typically packed into a FloatBuffer:

```
static final float[][] CORNERS = {
  {-0.8f, 0.8f, 0.8f}, { 0.8f, 0.8f, 0.8f}, { 0.8f, 0.8f,-0.8f}, {-0.8f, 0.8f,-0.8f},
  {-0.8f,-0.8f, 0.8f}, { 0.8f,-0.8f, 0.8f}, { 0.8f,-0.8f,-0.8f}, {-0.8f,-0.8f,-0.8f},
};
static final int[] INDICES = { 0, 1, 2, 3, 0, 4, 5, 1, 5, 6, 2, 6, 7, 3, 7, 4 };
private void drawCube() {
  FloatBuffer vertices = Buffers.newDirectFloatBuffer(INDICES.length * 3);
  for (int index : INDICES) { vertices.put(CORNERS[index]); }
  vertices.rewind();
  fillCurrentVertexBuffer("vPosition", vertices);
  // ...
  glDrawArrays(GL_LINE_STRIP, 0, INDICES.length);
}
```

...and it's boring, because we have no 3D.



# Binding multiple buffers in a VAO

Need more info? We can pass more than just *coordinate* data--we can create as many buffer objects as we want for different types of per-vertex data.

To bind two arrays of floats together, we build a *vertex array object* as before:

```
int vertexArrayId = glGenVertexArrays();
glBindVertexArray(vertexArrayId);
```

We bind a vertex buffer object for coordinate data, then another for normals:

```
addVertexBuffer("vPosition", vertices);
addVertexBuffer("vNormal", normals);
```

Later, to render, we'll unbind the buffers and work only with the vertex array:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glDrawArrays(GL_LINE_STRIP, 0, INDICES.length);
```

# Memory management: Lifespan of an OpenGL object

Most objects in OpenGL are created and deleted explicitly. Because these entities live in the GPU, they're outside the scope of Java's garbage collection.

The typical creation and deletion of an OpenGL object look like this:

```
int createAndBindVBO() {
  int name = glGenBuffers();
  glBindBuffer(GL_ARRAY_BUFFER, name);
  return name;
}

// Work with your object

void deleteVBO(int vboName) {
  glDeleteBuffers(vboName);
}
```

# 2. Getting some perspective

To add *3D perspective* to our flat model, we face three challenges:

- Compute a 3D perspective matrix
- Pass it to OpenGL, and on to the GPU
- Apply it to each vertex

To do so we're going to need to apply our perspective matrix in the shader, which means we'll need to build our own 4x4 perspective transform.

# 4x4 perspective matrix transform

Every OpenGL package provides utilities to build a perspective matrix. You'll usually find a method named something like *glGetFrustum()* which will assemble a 4x4 grid of floats suitable for passing to OpenGL.

Or you can build your own:

$$P = \begin{pmatrix} \frac{1}{ar \cdot \tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-NearZ - FarZ}{NearZ - FarZ} & \frac{2 \cdot FarZ \cdot NearZ}{NearZ - FarZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$\alpha$: Field of view, typically 50°

*ar*: Aspect ratio of width over height

*NearZ*: Near clip plane

*FarZ*: Far clip plane

# Passing uniform data to GLSL

The method glGetUniformLocation() will look up the location of a uniform parameter in a shader program.

(This is analogous to the attribute lookup seen earlier.)

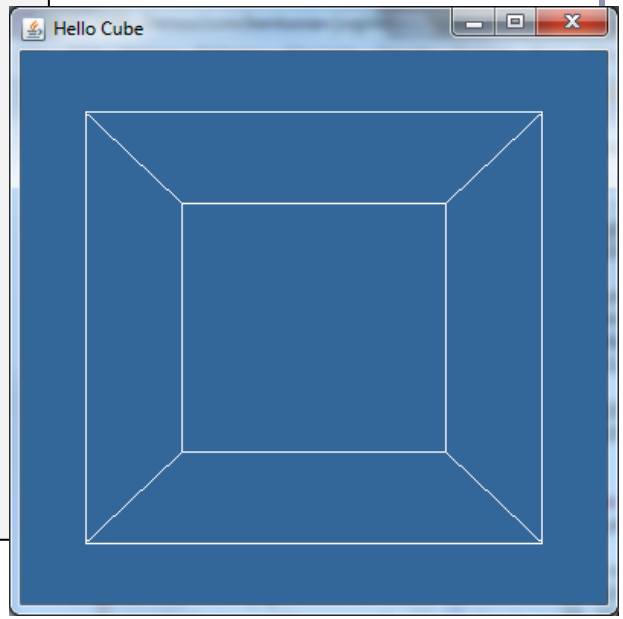
```
private void updateM4x4(String name, M4x4 T) {
  int uniform = glGetUniformLocation(program, name);
  if (uniform != -1) {
    glUniformMatrix4(uniform, false, T.asFloats());
  }
}
```

# Reading uniform data in GLSL

Next we need to modify our shader to transform our vertices by our perspective matrix.

This shader takes a matrix and applies it to each vertex:

```
#version 330

uniform mat4 modelToScreen;

in vec4 vPosition;

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



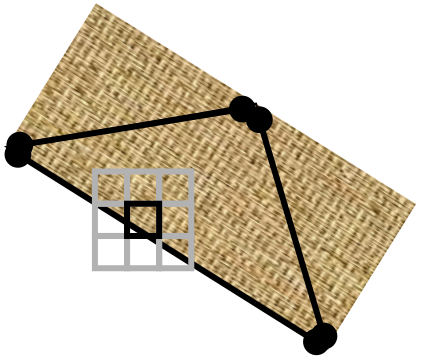
# Multiple uniforms

```
#version 330

uniform mat4 modelToScreen;
uniform mat4 modelToWorld;
uniform mat3 normalToWorld;

in vec4 vPosition;
in vec3 vNormal;

void main() {
  vec3 p = (modelToWorld * vPosition).xyz;
  vec3 n = normalize(normalToWorld * vNormal);
  // ...
```

Use multiple uniforms for different fields that are constant throughout the rendering pass, such as transform matrices and lighting coordinates.

# 3. Lighting and Shading

- Vertex shader outputs are interpolated across fragments.



This makes the implementation of classic illumination models like *Gouraud shading* very straightforward.

```
// ...
out vec4 color;
void main() {
  vec3 N = // ...
  vec3 L = // ...
  float diffuse = Kd * clamp(0, dot(N, L),
1);
  color = vec4(PURPLE * diffuse, 1.0);
}
```

Diffuse lighting
$d = k_D(N \bullet L)$
expressed as a shader

// Vertex Shader

```
// ...
in vec4 color;
out vec4 fragmentColor;

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

// Fragment Shader

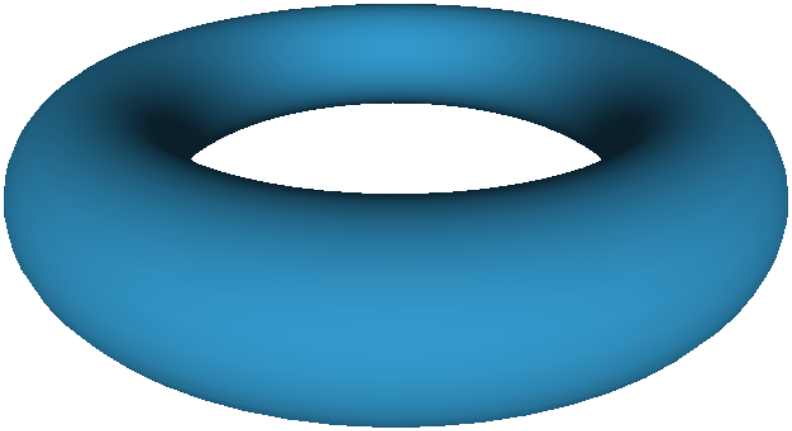
# Gouraud and Phong

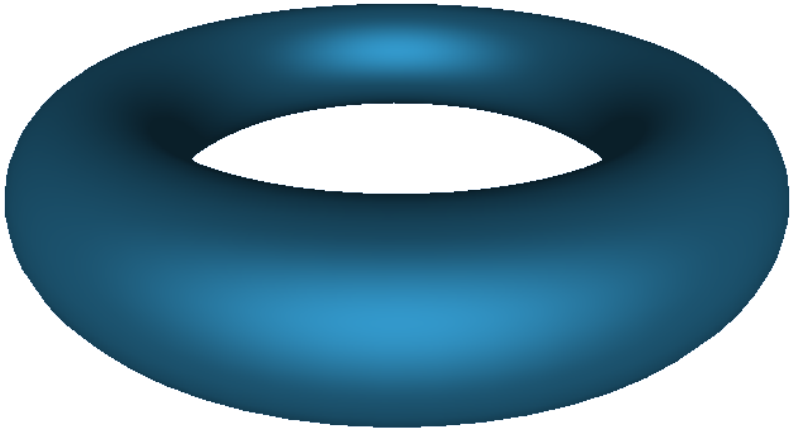
## Gouraud shading

- Compute color in vertex shader
- Let OpenGL interpolate color across fragments
- Output interpolated color

## Phong shading

- Compute normal in vertex shader
- Let OpenGL interpolate normal across fragments
- Compute color separately for each fragment

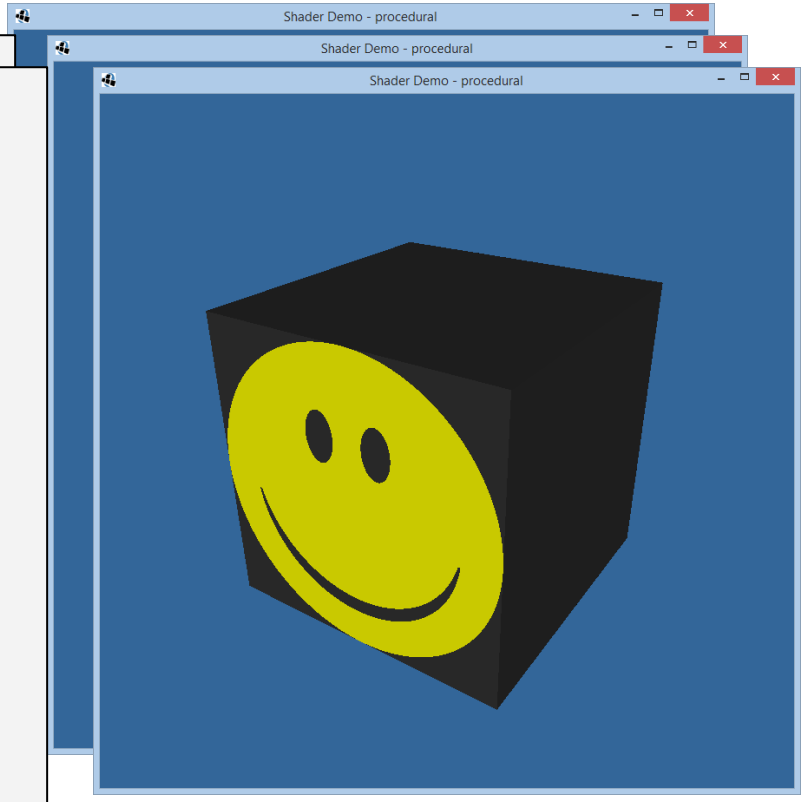




GLSL includes handy helper methods for illumination, such as a reflect() method that reflects one vector across another--perfect for specular highlighting. For a few examples, check out the demo source code on github.

# Procedural texturing in the fragment shader

```
// ...
const vec3 CENTER = vec3(0, 0, 1);
const vec3 LEFT_EYE = vec3(-0.2, 0.25, 0);
const vec3 RIGHT_EYE = vec3(0.2, 0.25, 0);
// ...

void main() {
  bool isOutsideFace = (length(position - CENTER) > 1);
  bool isEye = (length(position - LEFT_EYE) < 0.1)
      || (length(position - RIGHT_EYE) < 0.1);
  bool isMouth = (length(position - CENTER) < 0.75)
      && (position.y <= -0.1);

  vec3 color = (isMouth || isEye || isOutsideFace)
      ? BLACK : YELLOW;

  fragmentColor = vec4(color, 1.0);
}
```




(Code truncated for brevity--again, check out the source on github for how I did the curved mouth and oval eyes.)

Bonus slide

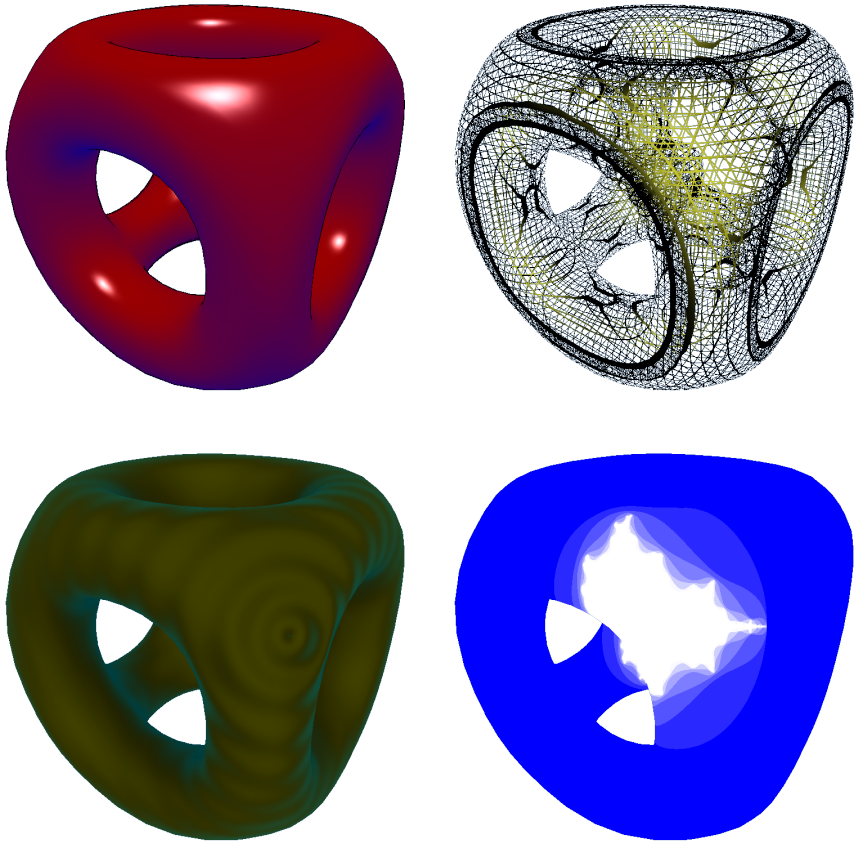
# Voronoi diagrams in the fragment shader

- For a limited set of generating points, can compute the *Voronoi Diagram* in the fragment shader.
- Simple version: “F2-F1”: find the nearest two generating points by iteration, render the isolines where their forces = 0.
- Better: With a two-pass solution, can generate the isolines *within* the cell as well (see link)

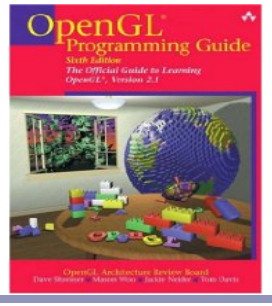
Iñigo Quilez (Pixar, Oculus)
http://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm

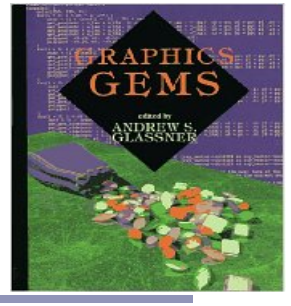
# More advanced surface effects

- Specular highlighting
- Non-photorealistic illumination
- Volumetric textures
- Bump-mapping
- Interactive surface effects
- Ray-casting in the shader
- Higher-order math in the shader
- ...much, much more!



# Recommended reading




- Course source code on Github -- many sample shaders (https://github.com/AlexBenton/AdvancedGraphics/tree/master/AdvGraph1415)
- *The OpenGL Programming Guide* (2013), by Shreiner, Sellers, Kessenich and Licea-Kane
  - Some also favor *The OpenGL Superbible* for code samples and demos
  - There's also an OpenGL-ES reference, same series
- *OpenGL Insights* (2012), by Cozzi and Riccio
- *OpenGL Shading Language* (2009), by Rost, Licea-Kane, Ginsburg et al
- The *Graphics Gems* series from Glassner
- ShaderToy.com, a web site by Inigo Quilez (Pixar) dedicated to amazing shader tricks and raycast scenes