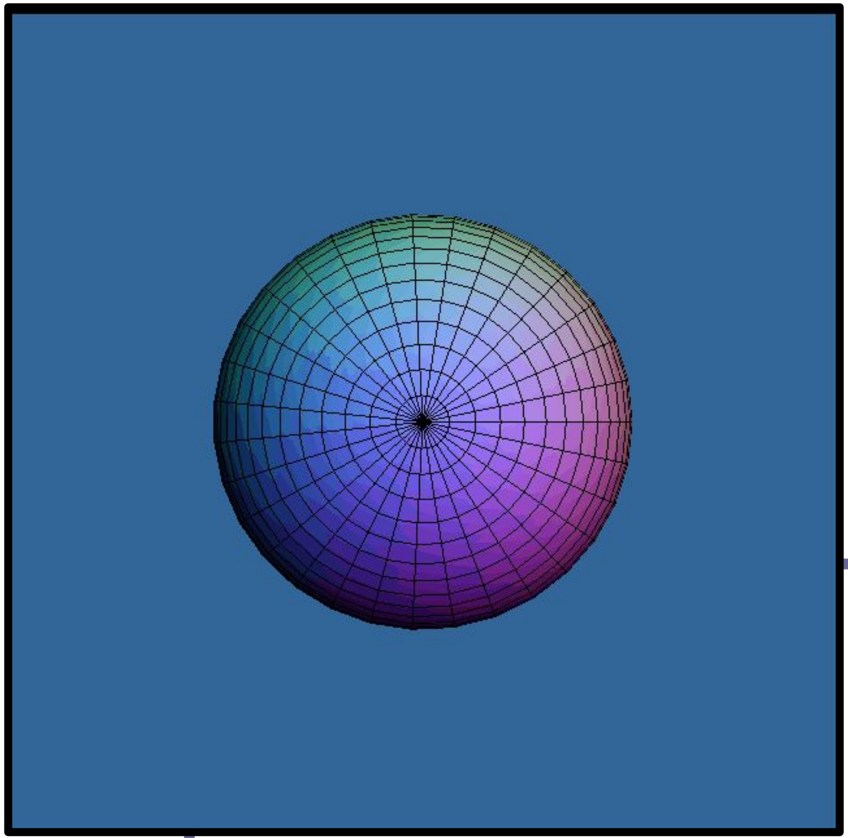
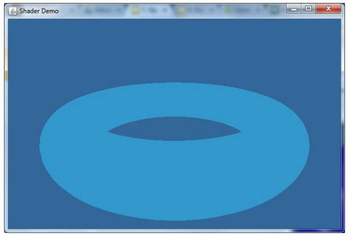
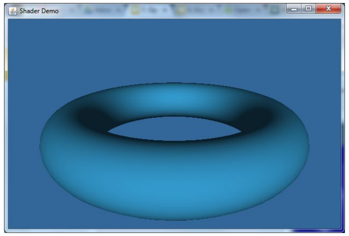
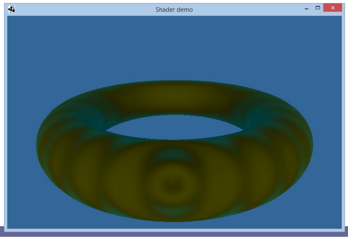
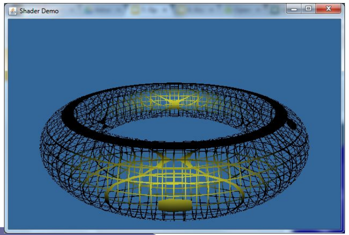




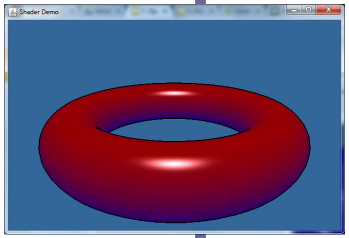


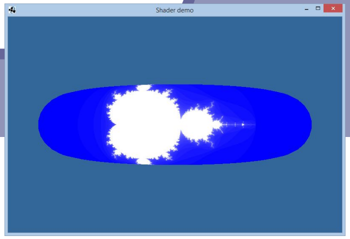
Advanced Graphics





OpenGL and Shaders I





Alex Benton, University of Cambridge – alex@bentonian.com

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)

Mobile GPUs

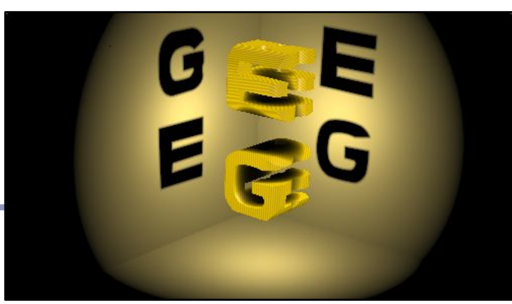
- OpenGL ES 1.0-3.2
  - A stripped-down version of OpenGL
  - Removes functionality that is not strictly necessary on mobile devices (like recursion!)
- Devices
  - iOS: iPad, iPhone, iPod Touch
  - Android phones
  - PlayStation 3, Nintendo 3DS, and more

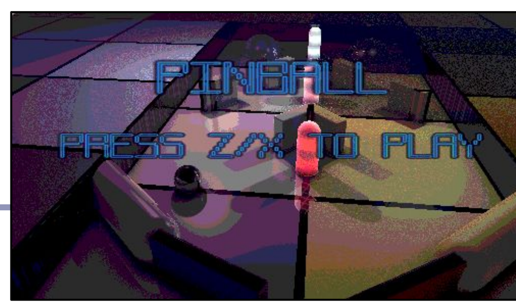
OpenGL ES 2.0 rendering (iOS)

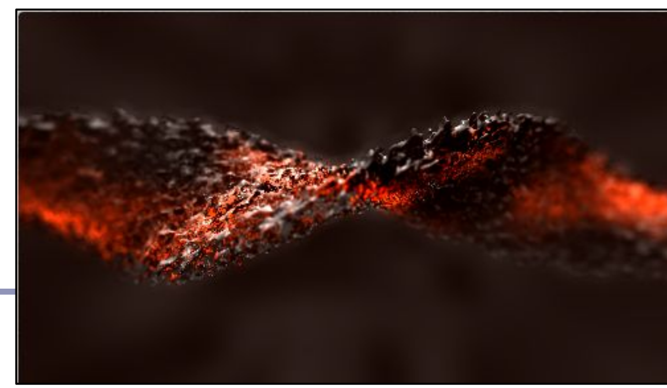
WebGL

- JavaScript library for 3D rendering in a web browser
  - Based on OpenGL ES 2.0
  - Many supporting JS libraries
  - Even gwt, angular, dart...
- Most modern browsers support WebGL, even mobile browsers
  - Enables in-browser 3D games
  - Enables realtime experimentation with glsl shader code

Samples from Shadertoy.com





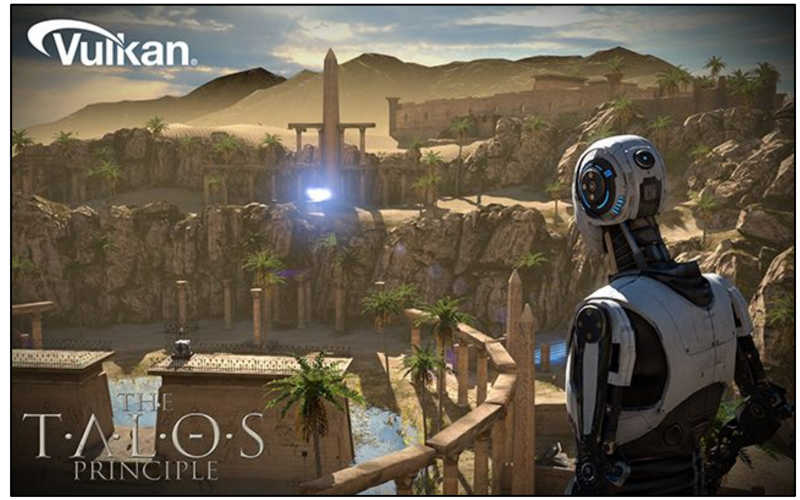


Vulkan

Vulkan is the next generation of OpenGL: a cross-platform open standard aimed at pure performance on modern hardware

Compared to OpenGL, Vulkan--

- Reduces CPU load
- Has better support of multi-CPU core architectures
- Gives finer control of the GPU



The Talos Principle running on Vulkan (via www.geforce.com)

--but--

- Drawing a few primitives can take 1000s of lines of code
- Intended for game engines and code that must be very well optimized

OpenGL in Java - choices

***JOGL*: "Java bindings for OpenGL"**

jogamp.org/jogl

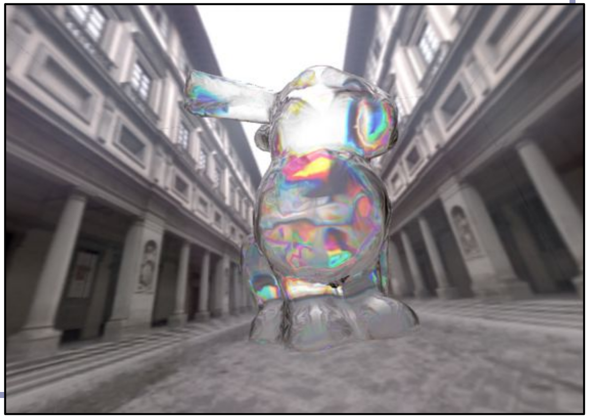
JOGL apps can be deployed as applications or as *applets*, making it suitable for educational web demos and cross-platform applications.

- 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"**

www.lwjgl.org

LWJGL is targeted at game developers, so it's got a 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.

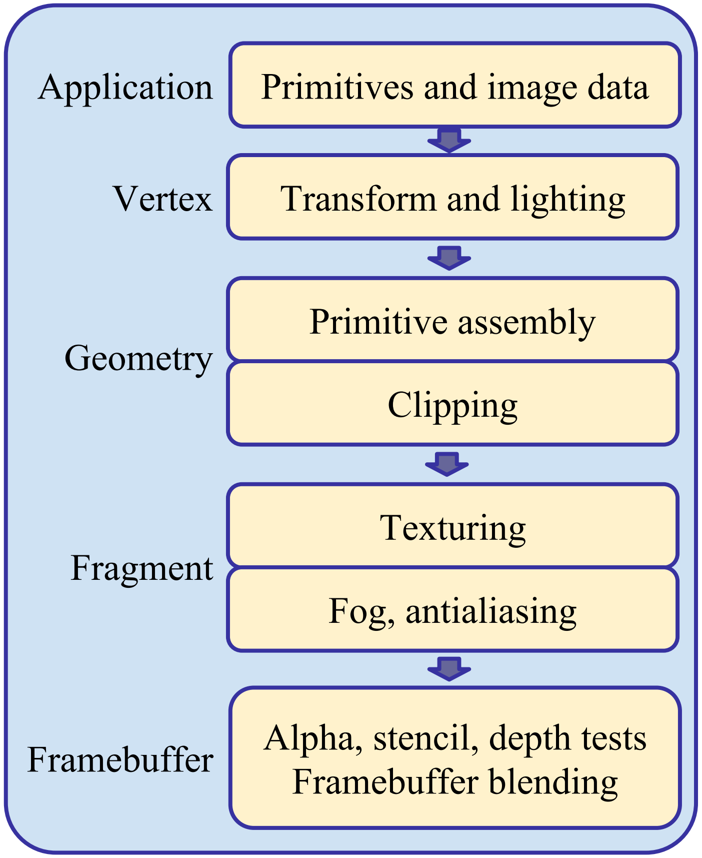
You'll write shaders in the OpenGL shader language, GLSL.

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

The OpenGL rendering pipeline

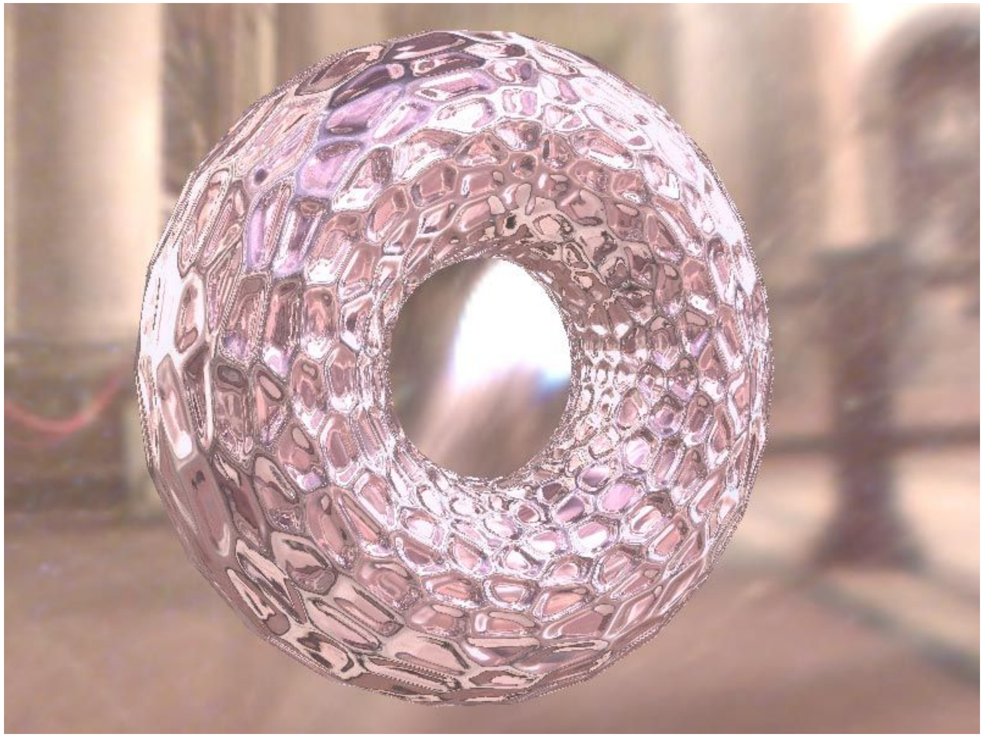
An OpenGL application assembles sets of *primitives*, *transforms* and *image data*, which it passes to OpenGL's GLSL shaders.

- *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*
(a massively simplified view)

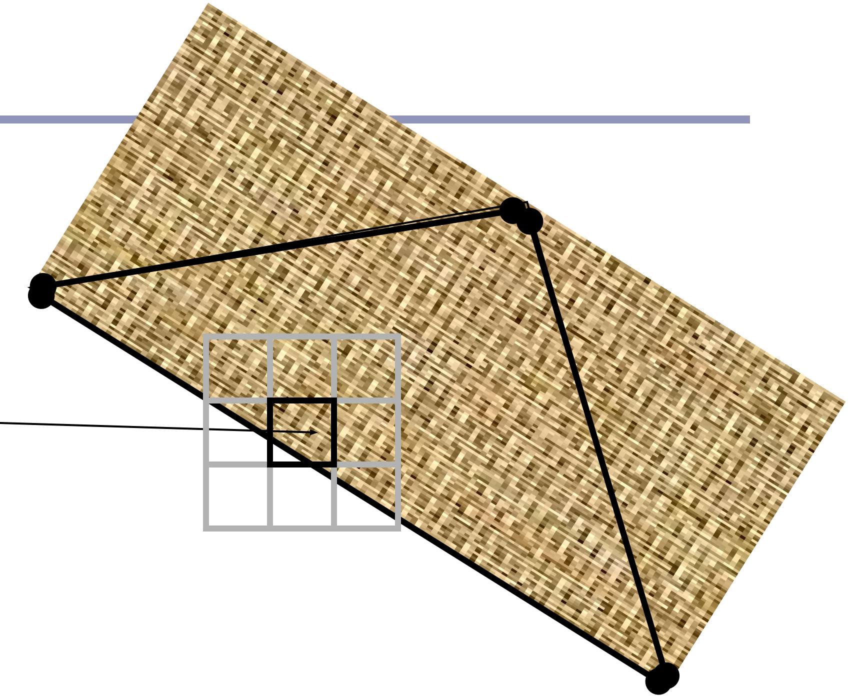
Shader gallery I



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

OpenGL: **Shaders**

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
- Vulkan's SPIR-V lets developers code in high-level GLSL but tune at the machine code level

GPUs typically have multiple processing units

That means that multiple shaders execute in parallel

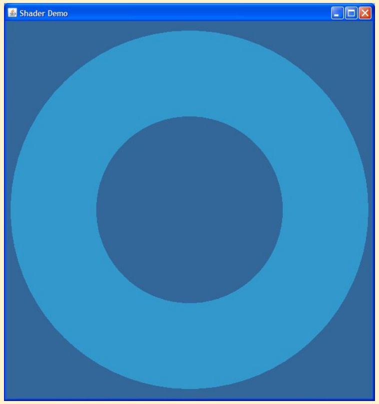
- We're moving away from the purely-linear flow of early "C" programming models

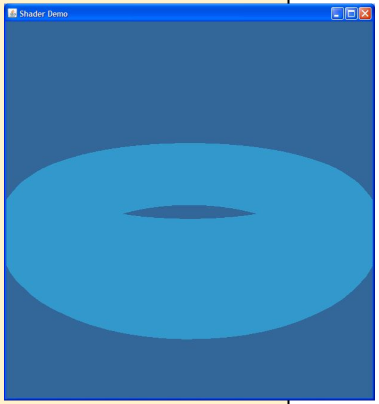
Shader example one – ambient lighting

```
#version 330

uniform mat4 mvp;

in vec4 vPos;

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

// Vertex Shader

```
#version 330

out vec4 fragmentColor;

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





// Fragment Shader

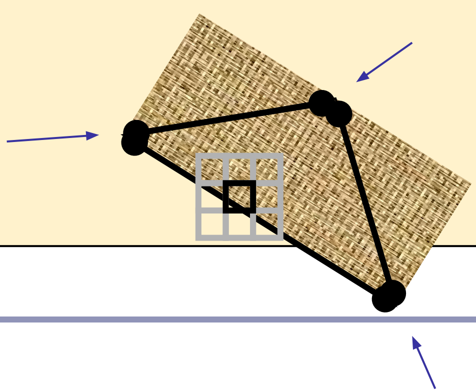
Vertex outputs become fragment inputs

Input

```
#version 330

uniform mat4 mvp;

in vec4 vPos;
out vec3 c;

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

Output

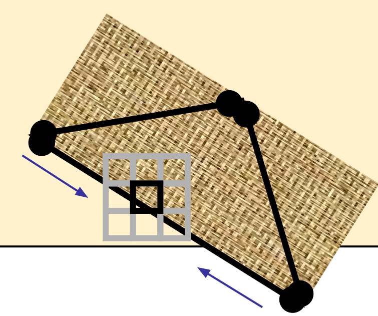
interpolation



Input

```
#version 330

in vec3 c;
out vec4 fragmentColor;

void main() {
  fragmentColor = vec4(c, 1);
}
```

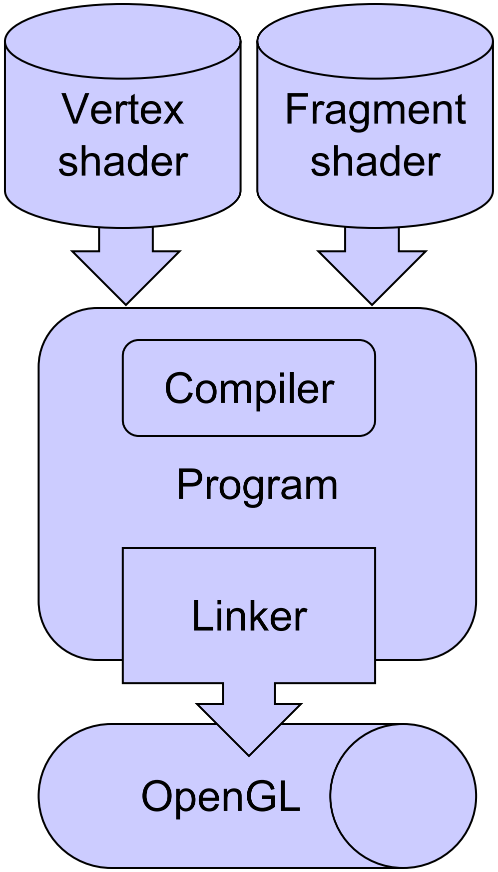
Color output



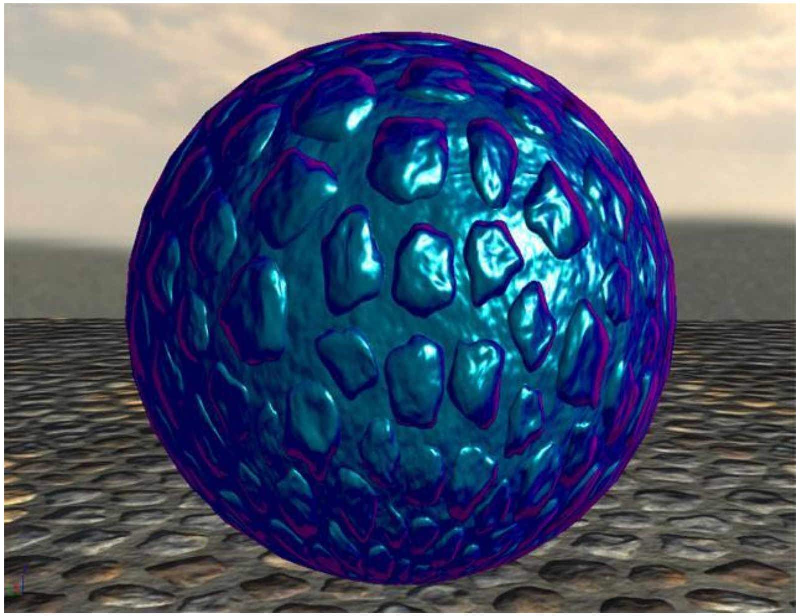
OpenGL / GLSL API - setup

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



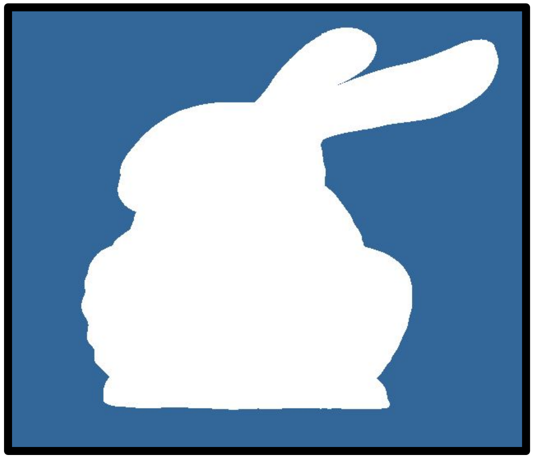
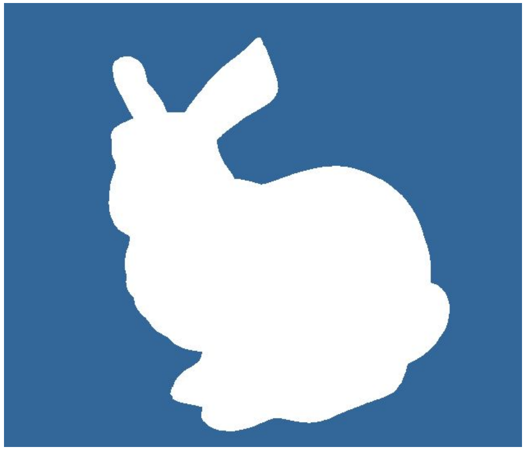
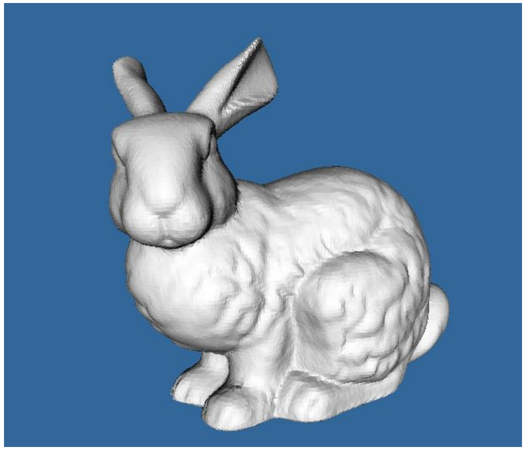
Above: Kevin Boulanger (PhD thesis, "*Real-Time Realistic Rendering of Nature Scenes with Dynamic Lighting*", 2005)

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

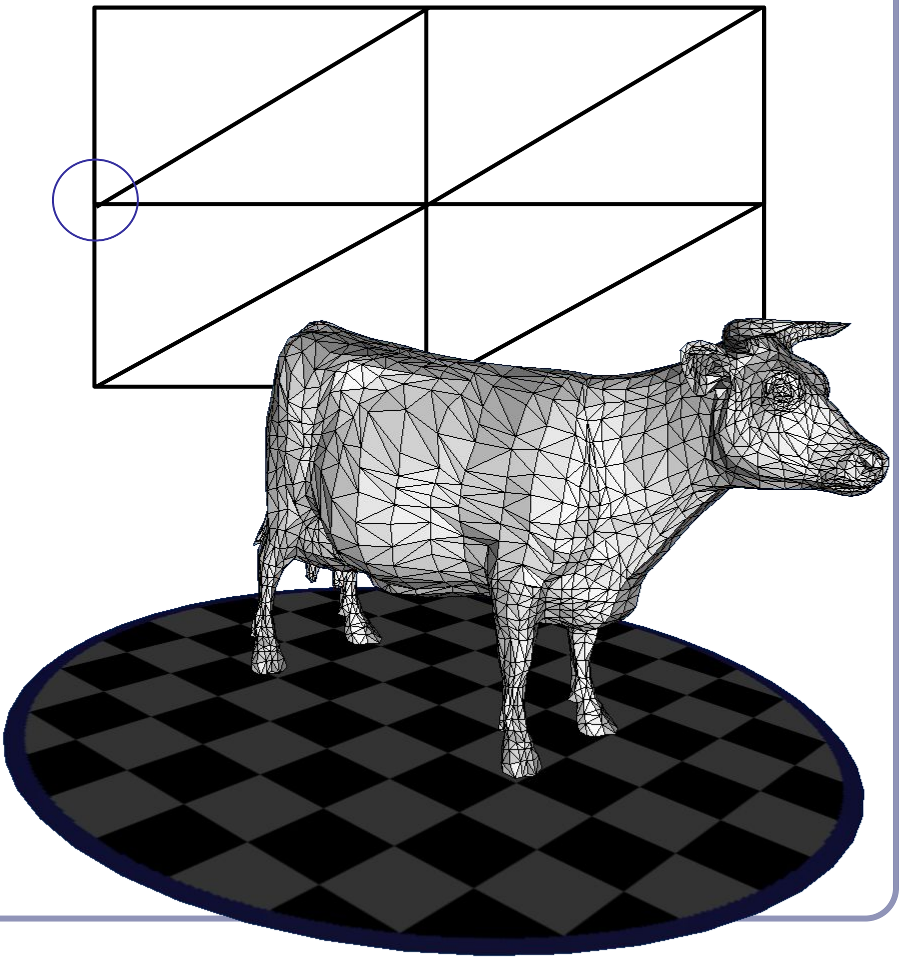
Geometry in OpenGL

The atomic datum of OpenGL is a **vertex**.

- 2d or 3d
- Specify arbitrary details

The fundamental primitives in OpenGL are the **line segment** and **triangle**.

- Very hard to get wrong
- {vertices} + {ordering} = surface

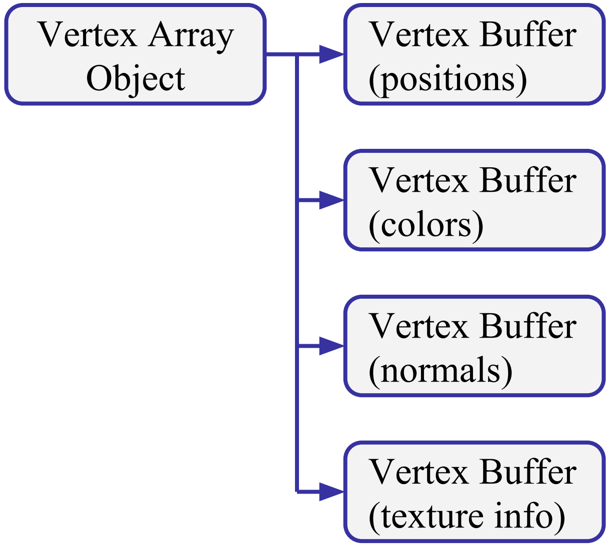


Geometry in OpenGL

Vertex 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 can bind and attach them to *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.



HelloGL.java [1/4]

```
///////////////////////////////////////////////////////////////////////////
// Set up GLFW window

GLFWErrorCallback errorCallback = GLFWErrorCallback.createPrint(System.err);
GLFW.glfwSetErrorCallback(errorCallback);
GLFW.glfwInit();
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MAJOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_CONTEXT_VERSION_MINOR, 3);
GLFW.glfwWindowHint(GLFW.GLFW_OPENGL_PROFILE, GLFW.GLFW_OPENGL_CORE_PROFILE);
GLFW.glfwWindowHint(
    GLFW.GLFW_OPENGL_FORWARD_COMPAT, GLFW.GLFW_TRUE);
long window = GLFW.glfwCreateWindow(
    800 /* width */, 600 /* height */, "HelloGL", 0, 0);
GLFW.glfwMakeContextCurrent(window);
GLFW.glfwSwapInterval(1);
GLFW.glfwShowWindow(window);

///////////////////////////////////////////////////////////////////////////
// Set up OpenGL

GL.createCapabilities();
GL11.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);
GL11.glClearDepth(1.0f);
```

"HelloGL.java" - github.com/AlexBenton/AdvancedGraphics

HelloGL.java [2/4]

```
////////////////////////////////////
// Set up minimal shader programs

// Vertex shader source
String[] vertex_shader = {
  "#version 330\n",
  "in vec3 v;",
  "void main() {",
  "  gl_Position = ",
  "      vec4(v, 1.0);",
  "}"
};

// Fragment shader source
String[] fragment_shader = {
    "#version 330\n",
    "out vec4 frag_colour;",
    "void main() {",
    "  frag_colour = ",
    "      vec4(1.0);",
    "}"
};
```

```
// Compile vertex shader
int vs = GL20.glCreateShader(
    GL20.GL_VERTEX_SHADER);
GL20.glShaderSource(
    vs, vertex_shader);
GL20.glCompileShader(vs);

// Compile fragment shader
int fs = GL20.glCreateShader(
    GL20.GL_FRAGMENT_SHADER);
GL20.glShaderSource(
    fs, fragment_shader);
GL20.glCompileShader(fs);

// Link vertex and fragment
// shaders into active program
int program =
    GL20.glCreateProgram();
GL20.glAttachShader(program, vs);
GL20.glAttachShader(program, fs);
GL20.glLinkProgram(program);
GL20.glUseProgram(program);
```

"HelloGL.java" - github.com/AlexBenton/AdvancedGraphics

HelloGL.java [3/4]

```
////////////////////////////////////////////////////////////////////////////
// Set up data

// Fill a Java FloatBuffer object with memory-friendly floats
float[] coords = new float[] { -0.5f, -0.5f, 0,  0, 0.5f, 0,  0.5f, -0.5f, 0 };
FloatBuffer fbo = BufferUtils.createFloatBuffer(coords.length);
fbo.put(coords);                                // Copy the vertex coords into the
floatbuffer
fbo.flip();                                     // Mark the floatbuffer ready for reads

// Store the FloatBuffer's contents in a Vertex Buffer Object
int vbo = GL15.glGenBuffers();                  // Get an OGL name for the VBO
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo);   // Activate the VBO
GL15.glBufferData(GL15.GL_ARRAY_BUFFER, fbo, GL15.GL_STATIC_DRAW);  // Send VBO data to GPU

// Bind the VBO in a Vertex Array Object
int vao = GL30.glGenVertexArrays();             // Get an OGL name for the VAO
GL30.glBindVertexArray(vao);                    // Activate the VAO
GL20.glEnableVertexAttribArray(0);              // Enable the VAO's first attribute (0)
GL20.glVertexAttribPointer(0, 3, GL11.GL_FLOAT, false, 0, 0);  // Link VBO to VAO attrib 0
```

HelloGL.java [4/4]

```
////////////////////////////////////////////////////////////////////////////
// Loop until window is closed

while (!GLFW.glfwWindowShouldClose(window)) {
  GLFW.glfwPollEvents();

  GL11.glClear(GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT);
  GL30.glBindVertexArray(vao);
  GL11.glDrawArrays(GL11.GL_TRIANGLES, 0 /* start */, 3 /* num vertices */);

  GLFW.glfwSwapBuffers(window);
}

////////////////////////////////////////////////////////////////////////////
// Clean up

GL15.glDeleteBuffers(vbo);
GL30.glDeleteVertexArrays(vao);
GLFW.glfwDestroyWindow(window);
GLFW.glfwTerminate();
GLFW.glfwSetErrorCallback(null).free();
```

“HelloGL.java” - github.com/AlexBenton/AdvancedGraphics

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. This lets us bind vertices with **normals**, **colors**, **texture coordinates**, etc...

Here we bind a vertex buffer object for position data and another for normals:

```
int vao = glGenVertexArrays();
glBindVertexArray(vao);
GL20.glEnableVertexAttribArray(0);
GL20.glEnableVertexAttribArray(1);
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_0);
GL20.glVertexAttribPointer( 0, 3, GL11.GL_FLOAT, false, 0, 0);
GL15.glBindBuffer(GL15.GL_ARRAY_BUFFER, vbo_1);
GL20.glVertexAttribPointer( 1, 3, GL11.GL_FLOAT, false, 0, 0);
```

Later, to render, we work only with the vertex array:

```
glBindVertexArray(vao);
glDrawArrays(GL_LINE_STRIP, 0, data.length);
```

Caution--all VBOs in a VAO must describe the same number of vertices!

Accessing named GLSL attributes from Java

```
// Vertex shader
// …

#version 330

in vec3 v;
void main() {
  gl_Position =
      vec4(v, 1.0);
}

// …
```

```
// …

glEnableVertexAttribArray(0);
glVertexAttribPointer(0,
    3, GL_FLOAT, false, 0, 0);

// …
```

The HelloGL sample code hardcodes the assumption that the vertex shader input field 'v' is the zeroeth input (position 0).

That's unstable: never rely on a fixed ordering.

Instead, fetch the attrib location:

```
int vLoc =
  GL20.glGetAttribLocation(program, "v");
GL20.glEnableVertexAttribArray( vLoc);
GL20.glVertexAttribPointer( vLoc,
    3, GL_FLOAT, false, 0, 0);
```

This enables greater flexibility and Java code that can adapt to dynamically-changing vertex and fragment shaders.

Improving data throughput

You configure how OpenGL interprets the vertex buffer. Vertices can be interpreted directly, or *indexed* with a separate integer indexing buffer. By re-using vertices and choosing ordering / indexing carefully, you can reduce the number of raw floats sent from the CPU to the GPU dramatically.

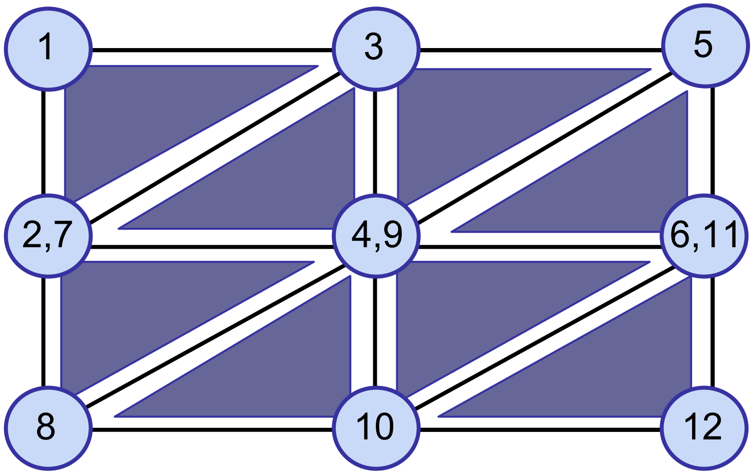
Options include line primitives--

- `GL_LINES`
- `GL_LINE_STRIP`
- `GL_LINE_LOOP`

--triangle primitives--

- `GL_TRIANGLES`
- `GL_TRIANGLE_STRIP`
- `GL_TRIANGLE_FAN`

--and more. OpenGL also offers *backface culling* and other optimizations.



Triangle-strip vertex indexing (counter-clockwise ordering)

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.

This means that **you must handle your own memory cleanup**.

```
// create and bind buffer object
int name = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, name);

// work with your object
// …

// delete buffer object, free memory
glDeleteBuffers(name);
```

Emulating classic OpenGL1.1 direct-mode rendering in modern GL

The original OpenGL API allowed you to use *direct mode* to send data for immediate output:

```
glBegin(GL_QUADS);
  glColor3f(0, 1, 0);
  glNormal3f(0, 0, 1);
  glVertex3f(1, -1, 0);
  glVertex3f(1, 1, 0);
  glVertex3f(-1, 1, 0);
  glVertex3f(-1, -1, 0);
glEnd();
```

Direct mode was very inefficient: the GPU was throttled by the CPU.

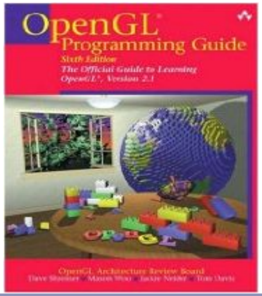
You can emulate the GL1.1 API:

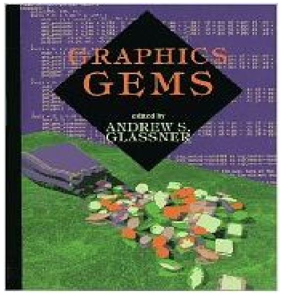
```
class GLVertexData {
  void begin(mode) { … }
  void color(color) { … }
  void normal(normal) { … }
  void vertex(vertex) { … }
  …
  void compile() { … }
}
```

The method compile() can encapsulate all the vertex buffer logic, making each instance a self-contained buffer object.

Check out a working example in the class `framework.GLVertexData` on the course github repo.

Recommended reading





Course source code on Github -- many demos
(https://github.com/AlexBenton/AdvancedGraphics)

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