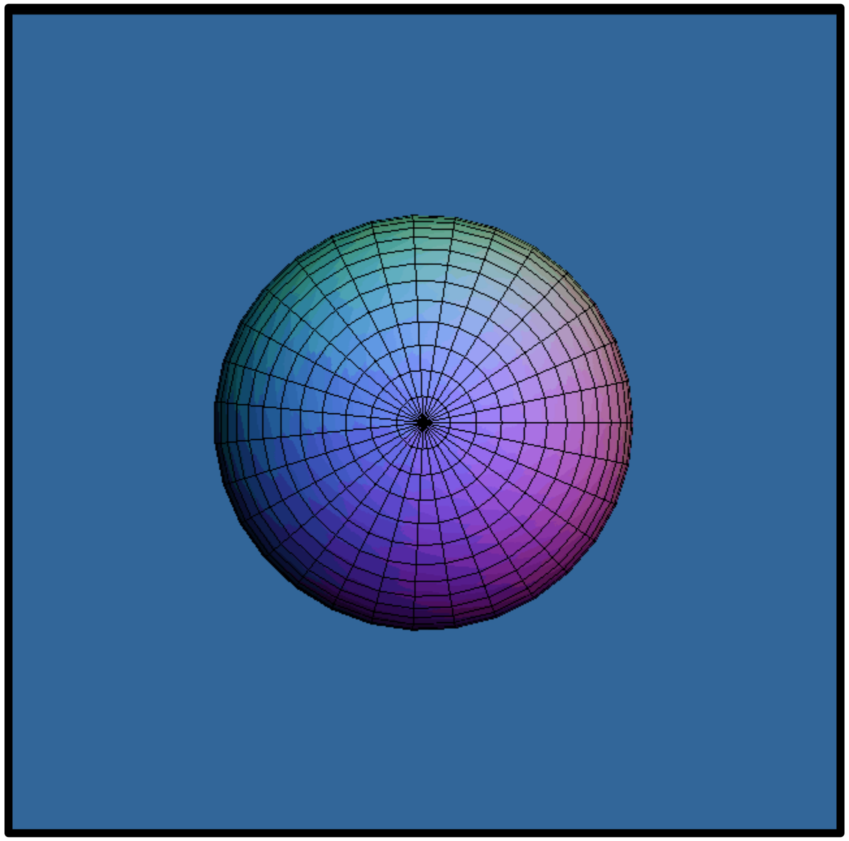
Advanced Graphics



OpenGL and Shaders

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

Supported in part by Google UK, Ltd

Today’s technologies

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)

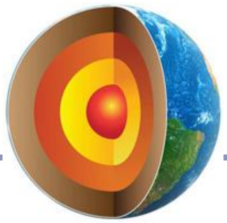
JavaScript (seriously!)
- WebGL is surprisingly popular

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

DirectX/Direct3d (Microsoft)
- Less well-designed (IMHO)
- Microsoft™ only
- DX 10 *required* Vista!
- Dependable updates

Mantle (AMD)
- Targeted squarely at game devel
- AMD-specific

Higher-level commercial libraries
- RenderMan
- AutoDesk / SoftImage



OpenGL

OpenGL is…

- hardware-independent
- operating system independent
- vendor neutral

OpenGL is a *state-based* renderer

- many settings are configured before passing in data; rendering behavior is modified by existing state
- most state is stored in custom graphics hardware
- this is very different from the OOP model, where data would carry its own state

OpenGL

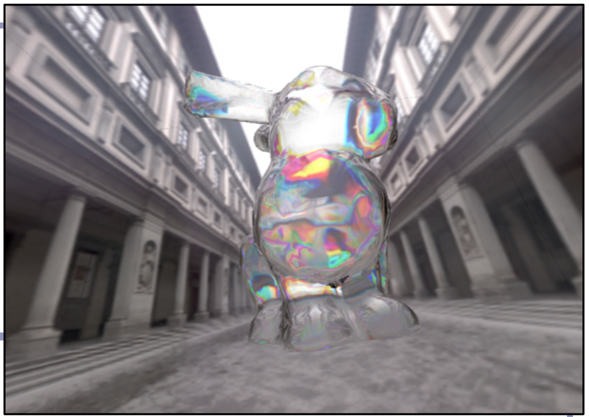
OpenGL is platform-independent, but implementations are platform-specific and often rely on native libraries

- Great support for Windows, Mac, linux, etc
- Support for mobile devices with OpenGL-ES
  - Android, iOS (but not Windows Phone)

Accelerates common 3D graphics operations

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

OpenGL in Java: *JOGL*


JOGL shaders in action.
Image from Wikipedia

JOGL is the Java binding for OpenGL.

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

- 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.

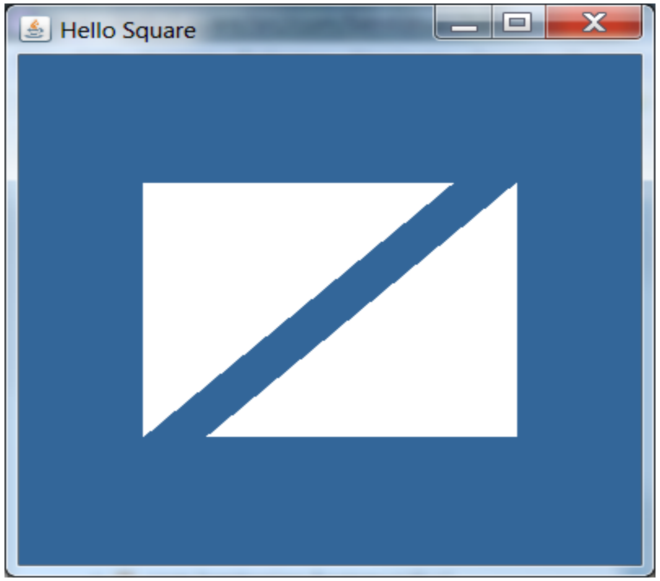
If you're only targeting the web, consider using *WebGL* instead.

In JOGL, everything renders in a *GLCanvas* instance, which is typically attached to an AWT Frame:

```
Frame frame = new Frame("Hello Square");
GLCanvas canvas = new GLCanvas();
frame.add(canvas);
canvas.addGLEventListener(new HelloSquareRenderer());
```

Minimal JOGL: *Hello Square*

```
public class HelloSquareRenderer implements GLEventListener {

  @Override
  public void init(GLAutoDrawable glDrawable) {
    GL4 gl = glDrawable.getGL().getGL4();
    gl.glClearColor(0.2f, 0.4f, 0.6f, 0.0f);

    createAndBindVertexBuffer(gl);
    fillCurrentVertexBuffer(gl);

    gl.glEnableVertexAttribArray(0);
    gl.glVertexAttribPointer(0, 2, GL.GL_FLOAT, false, 0, 0);
  }

  private void createAndBindVertexBuffer(GL4 gl) {
    int[] arrays = {0};
    gl.glGenBuffers(1, arrays, 0);
    gl.glBindBuffer(GL.GL_ARRAY_BUFFER, arrays[0]);
  }

  private void fillCurrentVertexBuffer(GL4 gl) {
    FloatBuffer vertices = Buffers.newDirectFloatBuffer(new float[] {
      -0.6f,  0.5f,     0.4f,  0.5f,      -0.6f, -0.5f,     // First triangle
       0.6f, -0.5f,     0.6f,  0.5f,      -0.4f, -0.5f,     // Second triangle
    });
    gl.glBufferData(GL.GL_ARRAY_BUFFER,
        Float.SIZE * 2 * 2 * 3, vertices, GL.GL_STATIC_DRAW);
  }
```



```
  @Override
  public void display(GLAutoDrawable glDrawable) {
    GL4 gl = glDrawable.getGL().getGL4();
    gl.glClear(GL.GL_COLOR_BUFFER_BIT);
    gl.glDrawArrays(GL.GL_TRIANGLES, 0, 6);
  }
}
```

This is not good code! It doesn't clean up after itself and it makes some very coarse assumptions about the shader. Caveat coder.

Behind the scenes

Two players:

- The CPU, your processor and friend
- The *GPU* (*Graphical Processing Unit*) or equivalent software

The CPU passes buffers of data--typically vertices and data about them--to the GPU.

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

This process is called the *rendering pipeline*.

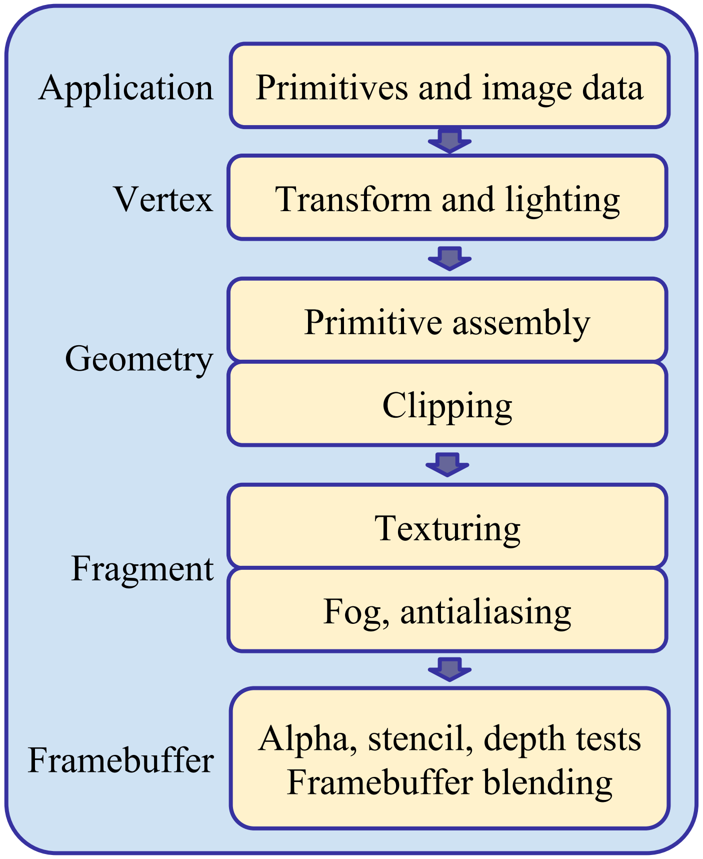
Implementing the rendering pipeline is a joint effort between you and the GPU.

The OpenGL rendering pipeline

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

GL processes every vertex in the primitives, computing the lighting and position of each one.

GL then interpolates the fragments of every pixel covered by every primitive between the vertices, shading each fragment.



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

JOGL's default shaders

If not otherwise instructed, JOGL behaves as though it has a default vertex and fragment shader, a legacy of the old fixed-function pipeline. They act like this:

```
#version 330


in vec4 position;
void main() {
  gl_Position = position;
}
```

```
#version 330


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

These aren't "default" shaders; there's no such thing. This is how the vestigial code of the fixed-funtction pipeline behaves.

Anatomy of a shader

(Briefly. More details next lecture.)

```
#version 330

layout(location = 0)in vec4 vPosition;

uniform mat4 modelToScreen;

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

GLSL version

Input: vertex data starting at position 0

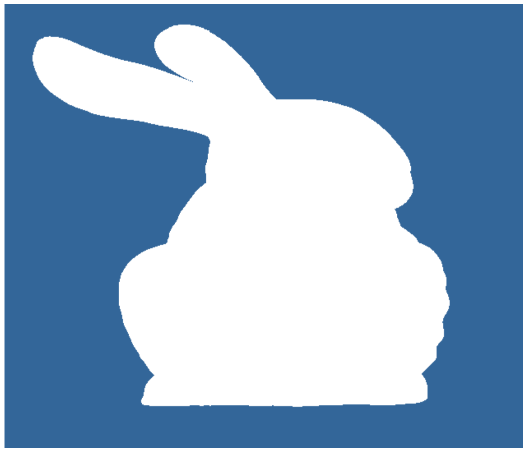
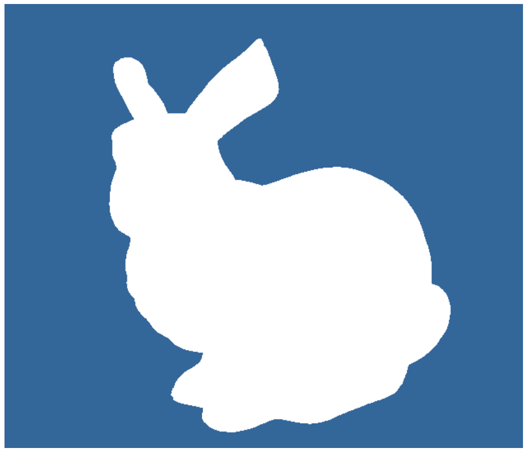
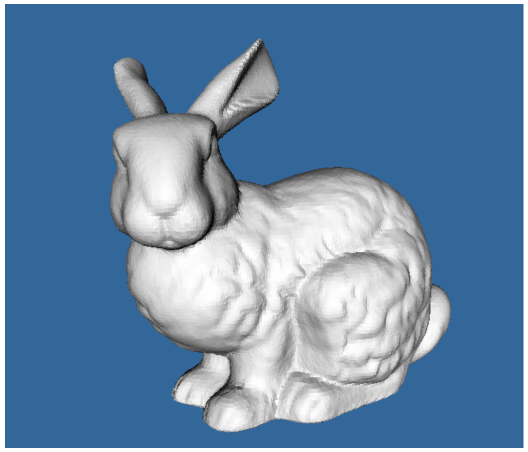
Input: a 4x4 matrix

Output: vertex input transformed by matrix input

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. Passing geometry to OpenGL

Let’s revisit how our Hello Square program wrote geometry.

First, we allocated a *vertex buffer*:

```
private void createAndBindVertexBuffer(GL4 gl) {
  int[] arrays = {0};                                  // Target for new buffer name
  gl.glGenBuffers(1, arrays, 0);                       // Allocate new name
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, arrays[0]);      // Bind ARRAY_BUFFER to name
}
```

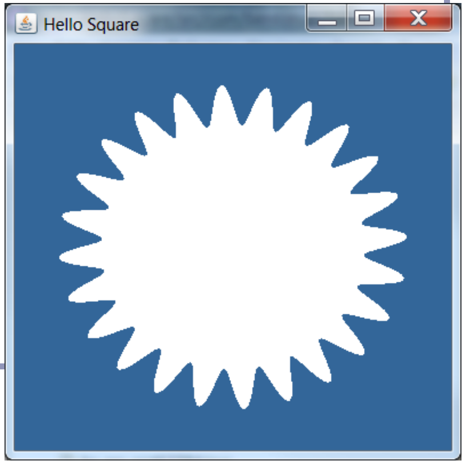
Then we filled the buffer with vertex coordinates:

```
private void fillCurrentVertexBuffer(GL4 gl) {
  FloatBuffer vertices = Buffers.newDirectFloatBuffer(new float[] {
    -0.6f,  0.5f,     0.4f,  0.5f,      -0.6f, -0.5f,     // First triangle
     0.6f, -0.5f,     0.6f,  0.5f,      -0.4f, -0.5f,     // Second triangle
  });
  gl.glBufferData(GL.GL_ARRAY_BUFFER,                  // Fill ARRAY_BUFFER with data
      Float.SIZE * 2 * 2 * 3, vertices, GL.GL_STATIC_DRAW);
}
```

Passing geometry to OpenGL

We could have made the vertices more interesting:

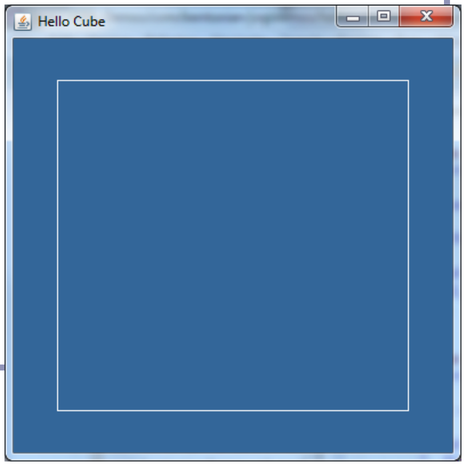
```
private void fillCurrentVertexBuffer(GL4 gl) {
  final double TWO_PI_OVER_360 = 2 * PI / 360;
  FloatBuffer vertices = Buffers.newDirectFloatBuffer(360 * 3 * 2);
  for (int i = 0; i < 360; i++) {
    double scale = 0.7 + 0.1 * sin(i * 24 * TWO_PI_OVER_360);
    vertices.put((float) (cos(i * TWO_PI_OVER_360) * scale));
    vertices.put((float) (sin(i * TWO_PI_OVER_360) * scale));
    scale = 0.7 + 0.1 * sin((i + 1) * 24 * TWO_PI_OVER_360);
    vertices.put((float) (cos((i + 1) * TWO_PI_OVER_360) * scale));
    vertices.put((float) (sin((i + 1) * TWO_PI_OVER_360) * scale));
    vertices.put(0);
    vertices.put(0);
  }
  vertices.rewind();
  gl.glBufferData(GL.GL_ARRAY_BUFFER, Float.SIZE * 360 * 3 * 2,
      vertices, GL.GL_STATIC_DRAW);
}
```



Passing geometry to OpenGL

For that matter, we could make the vertices 3D:

```
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 fillCurrentVertexBuffer(GL4 gl) {
  FloatBuffer vertices = Buffers.newDirectFloatBuffer(INDICES.length * 3);
  for (int index : INDICES) { vertices.put(CORNERS[index]); }
  vertices.rewind();
  int numBytes = Floats.BYTES * INDICES.length * 3;
  gl.glBufferData(
      GL.GL_ARRAY_BUFFER, numBytes, vertices, GL.GL_STATIC_DRAW);
}
// ...
gl.glDrawArrays(GL.GL_LINE_STRIP, 0, INDICES.length);
```



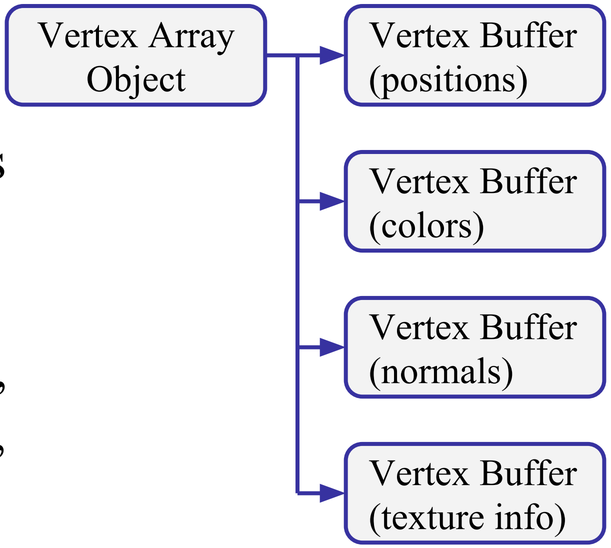
...and it'd be boring, because we have no 3D.

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.



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[] names = new int[1];
  gl.glGenBuffers(1, names, 0);
  gl.glBindBuffer(GL.GL_ARRAY_BUFFER, names[0]);
  return names[0];
}

// Work with your object

void deleteVBO(int vboName) {
  gl.glDeleteBuffers(1, vboName, 0);
}
```

IF YOU ALLOCATE OPENGL RESOURCES AND FORGET TO FREE THEM
INSTRUCTOR
YOU'RE GONNA HAVE A BAD TIME
imgflip.com

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 add apply our perspective matrix in the shader, which means we’ll need to replace JOGL’s default shaders with our own.

3D perspective

JOGL provides utilities to build a perspective matrix. The helper class *PMVMatrix* includes a method *glGetFrustum()* which will assemble a 4x4 grid of floats suitable for passing to OpenGL for perspective.

Or you can build your own:

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

α: Field of view, typically 50°

ar: Aspect ratio of width over height

NearZ: Near clip plane

FarZ: Far clip plane

Applying the transform in the shader

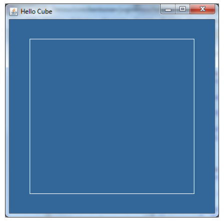
Next we need to modify our shader to transform our vertices by our perspective matrix.

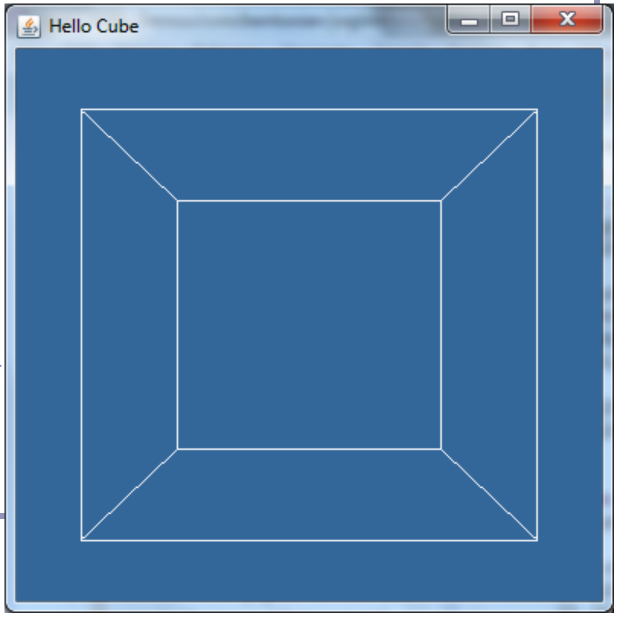
This shader takes a matrix and applies it to each vertex:

```
#version 330

layout(location = 0)in vec4 vPosition;

uniform mat4 modelToScreen;

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

Installing a shader in OpenGL

```
String[] vertexShader = ...;  // Lines of code of the shader
int[] vertexLengths = ...;    // Lengths of each line

int program = gl.glCreateProgram();
int vsName = gl.glCreateShader(GL4.GL_VERTEX_SHADER);
gl.glShaderSource(vsName, 1, vertexShader, vertexLengths, 0);
gl.glCompileShader(vsName);
gl.glAttachShader(program, vsName);

gl.glLinkProgram(program);
gl.glValidateProgram(program);
gl.glUseProgram(program);
```





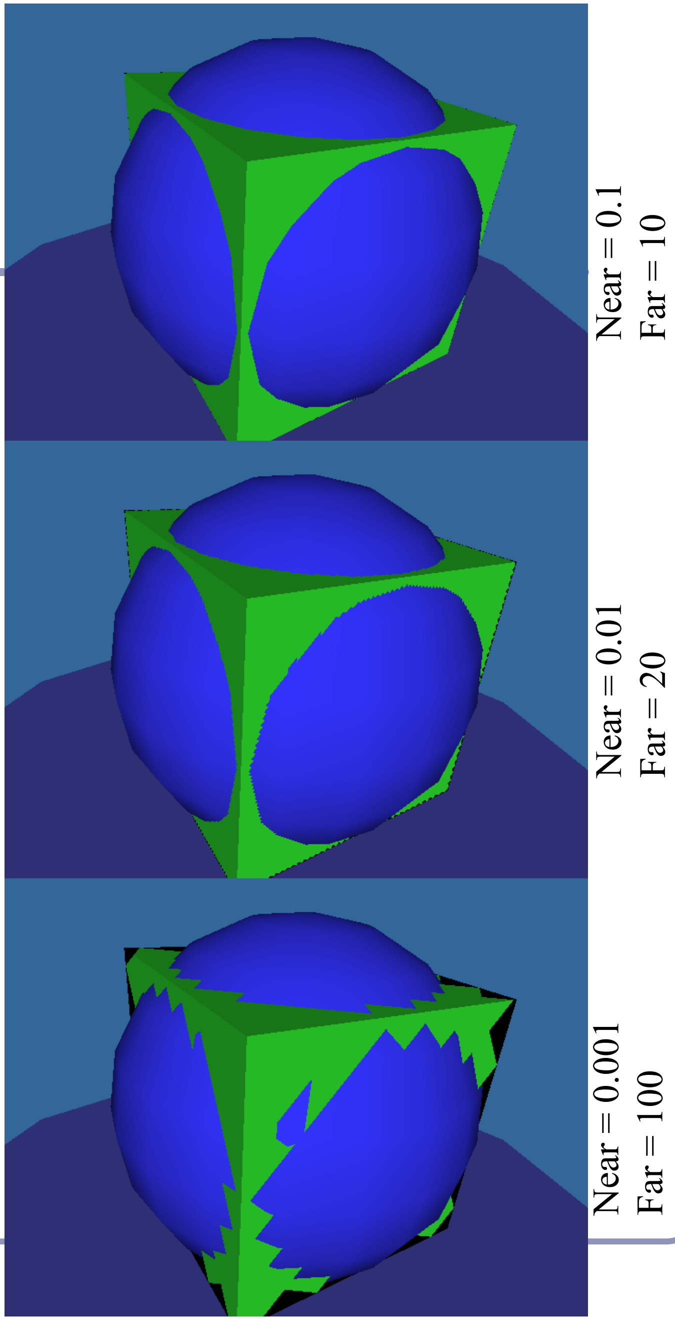
Output from using the code on this slide to apply the shader on the previous slide with the matrix from three slides ago to the cube from seven slides earlier.

Depth buffer bits

OpenGL uses the *Z-Buffer polygon scan conversion* method of hidden-surface removal.

Depth buffer precision is affected by the values specified for *zNear* and *zFar* . The greater the ratio of *zFar* to *zNear*, the less effective the depth buffer will be at distinguishing between surfaces that are near each other, because fewer bits will be available to express the larger dynamic range. According to the OpenGL documentation,

"Roughly $\log_2(zFar / zNear)$ bits of depth buffer precision are lost."



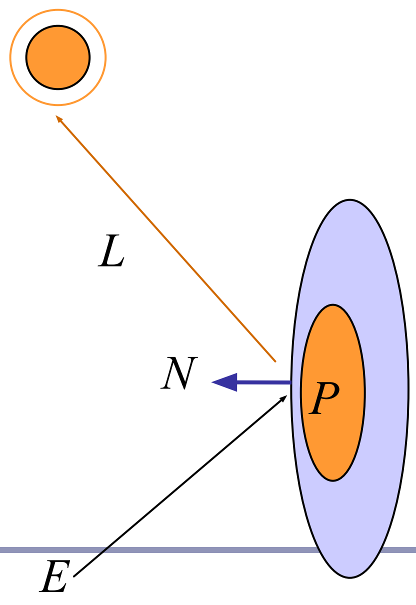
3. Lighting and Shading

You'll recall the equation for *diffuse lighting*, which models how light scatters more brightly off a surface if it strikes that surface full-on rather than at a glancing angle.

To model diffuse lighting with OpenGL, we'll need to compute the normal to our surface and the vector from each point to the light.

Diffuse lighting:

$$d = k_D(N \bullet L)$$



Binding different data types with a VAO

We'll pass both *coordinate* data and *normal* data to OpenGL.

To bind the two arrays of floats together, we build a *Vertex Array Object*:

```
int vertexArrayId = new int[1];
gl.glGenVertexArrays(1, vertexArrayId, 0);
gl.glBindVertexArray(vertexArrayId[0]);
```

We bind a buffer for vertex data, fill it, and then bind another for normals:

```
gl.glBindBuffer(GL.GL_ARRAY_BUFFER, vertexBufferIds[0]);
gl.glBufferData(GL.GL_ARRAY_BUFFER,
    Floats.BYTES * DATA_SIZE, vertices, GL.GL_STATIC_DRAW);
gl.glEnableVertexAttribArray(0);
gl.glVertexAttribPointer(0, 3, GL.GL_FLOAT, false, 0, 0);
```

(Repeat for **`normals`**, enabling vertex attrib array / pointer `1`)

Lastly we can unbind the buffers and work only with the bound vertex array:

```
gl.glBindBuffer(GL.GL_ARRAY_BUFFER, 0);
```

Diffuse lighting in a shader

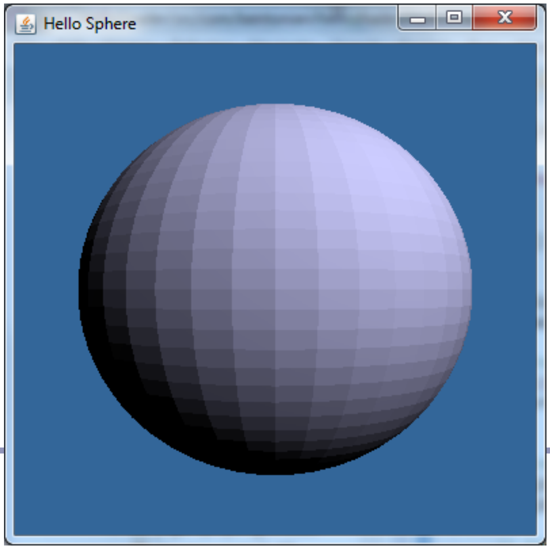
```
#version 330
uniform mat4 modelToScreen;
layout(location = 0)in vec4 vPos;
layout(location = 1)in vec4 vNormal;
out float diffuse;

void main() {
  vec3 L = vec3(10, 10, 10);
  gl_Position = modelToScreen * vPos;
  diffuse = dot(vNormal, normalize(L - vPos));
}
```

Diffuse lighting:

$$d = k_D(N \bullet L)$$

Expressed as a shader

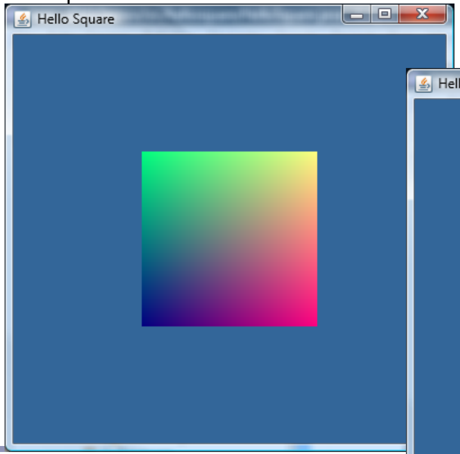
```
#version 330

in float diffuse;
out vec4 color;

void main() {
  vec3 purple = vec3(0.8, 0.8, 1);
  color = vec4(purple * diffuse, 1);
}
```

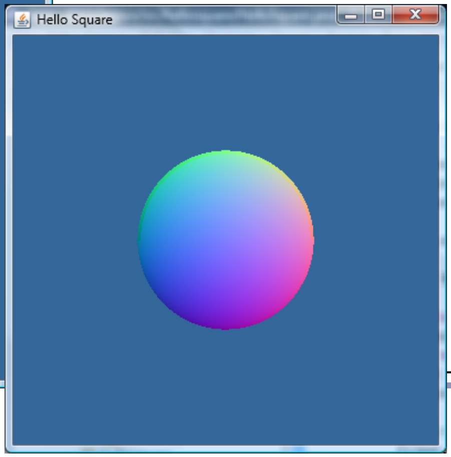


Putting it into practice: Simple parametric surfaces

```
FloatBuffer vertexData = new...
FloatBuffer colorData = new...
public void vertex(GL gl,
    float x, float y, float z) {
  vertexData.add(x, y, z);
  colorData.add(
    (x+1)/2.0f,
    (y+1)/2.0f,
    (z+1)/2.0f);
}
```

```
public void sphere(
    GL gl, double u, double v) {
  vertex(gl, cos(u)*cos(v),
      sin(u)*cos(v),
      sin(v));
}

for (double u=0; u<=2*PI; u+=0.1) {
  for (double v=0; v<=PI; v+=0.1) {
    sphere(gl, u, v);
    sphere(gl, u+0.1, v);
    sphere(gl, u+0.1, v+0.1);
    sphere(gl, u, v+0.1);
  }
}
```



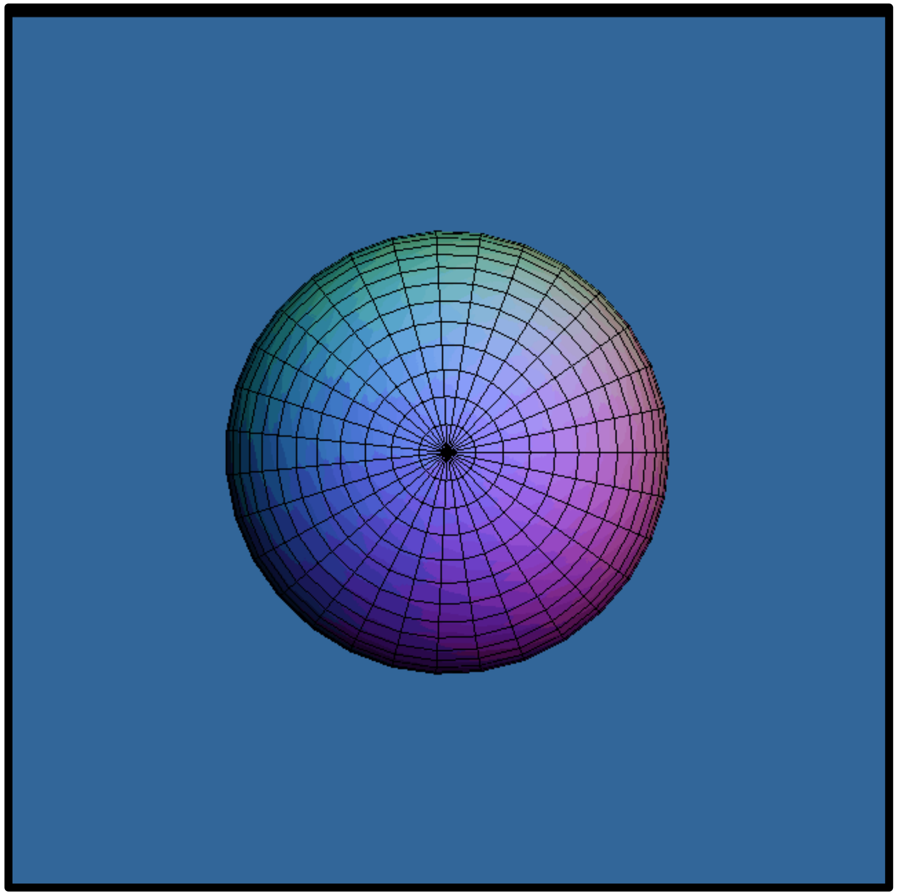


Animating a parametric surface

The animation at right shows the linear interpolation between four parametric surface functions.

- Colors are by XYZ.
- For surface functions A, B, each grid point is interpolated as

  $P(i, j) = (1 - t)A(i, j) + (t)B(i, j)$

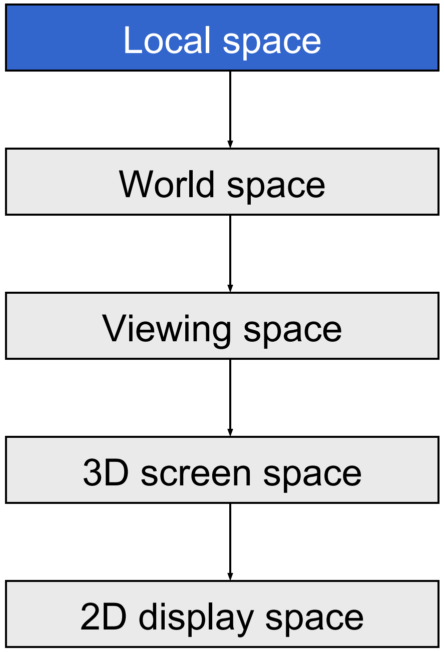


Using OpenGL to accelerate ray-tracing

To accelerate first raycast, don’t raycast: use existing hardware.

- Use hardware rendering (eg OpenGL) to write to an offscreen buffer.
- Set the color of each primitive equal to a pointer to that primitive.
- Render your scene in gl with z-buffering and no lighting.
- The ‘color’ value at each pixel in the buffer is now a pointer to the primitive under that pixel.

Anatomy of a rendering pipeline

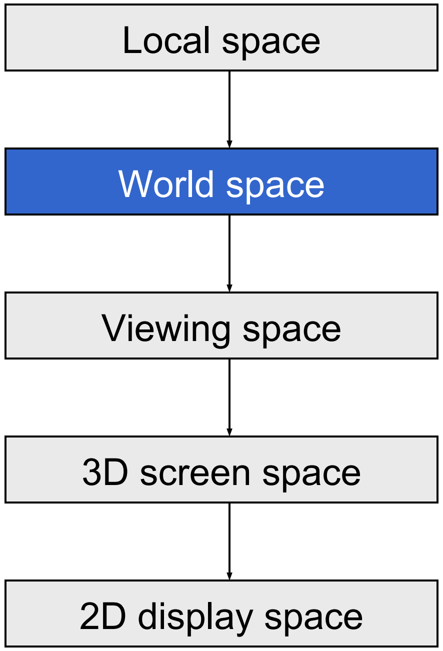


1) You'll generally define your geometry in *local space*. The vertices and coordinates of a surface are specified relative to a local basis and origin.

This encourages re-use and replication of geometry; it also saves the tedious math of storing rotations and other transformations within the vertices of the shape itself.

This means that changing the position of a highly complex object requires only changing a 4x4 matrix instead of recalculating all vertex values.

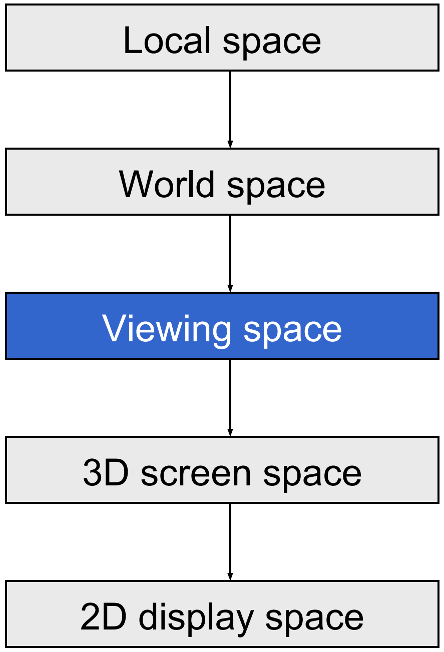
Anatomy of a rendering pipeline



2) You’ll want to transform vertices and surface normals from *local* to *world* space for operations like intersection, world-space clipping, and shading with lights positioned in the world.

A series of matrices are concatenated together to form the single transformation which is applied to each vertex. The pipeline is responsible for associating the state that transforms each group of vertices with the actual vertex values themselves (most often via a vertex shader in OpenGL).

Anatomy of a rendering pipeline



3) Rotate and translate the geometry from *world* space to *viewing* or *camera* space.

At this stage, all vertices are positioned relative to the point of view of the camera. (The world really does revolve around you!)

For example, a cube at (10,000, 0, 0) viewed from a camera (9,999, 0, 0) would now have relative position (1, 0, 0). Rotations would have similar effect.

This makes operations such as clipping and hidden-object removal much faster.

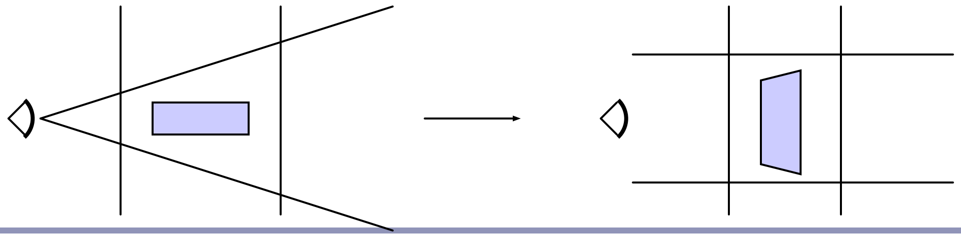
Anatomy of a rendering pipeline

$$\begin{tikzcd}
\text{Local space} \arrow[d] \\
\text{World space} \arrow[d] \\
\text{Viewing space} \arrow[d] \\
\boxed{\text{3D screen space}} \arrow[d] \\
\text{2D display space}
\end{tikzcd}$$

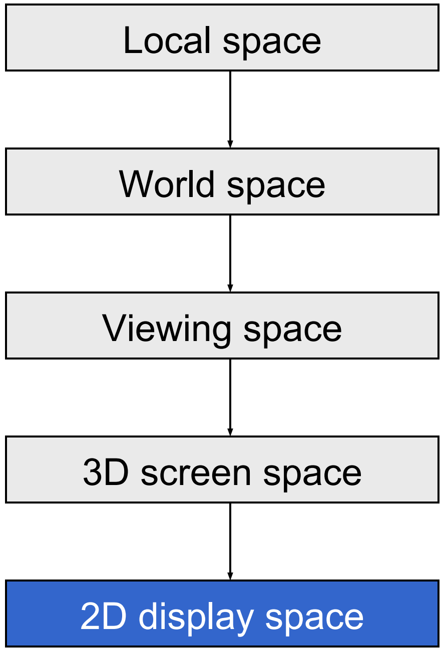
4) Perspective: Transform the viewing frustum into an axis-aligned box with the near clip plane at z=0 and the far clip plane at z=1. Coordinates are now in *3D screen space*.

This transformation is *not affine*: angles will distort and scales change.

Hidden-surface removal can be accelerated here by clipping objects and primitives against the viewing frustrum. Depending on implementation this clipping could be before transformation or after or both.



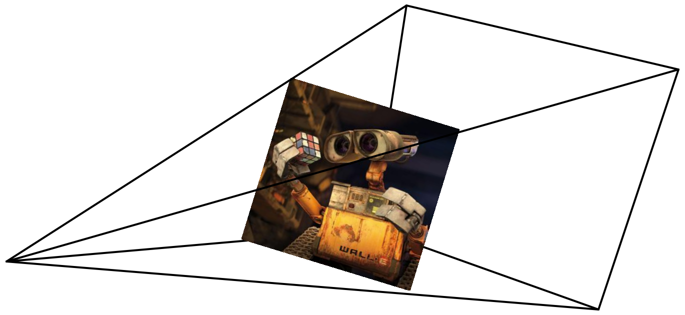
Anatomy of a rendering pipeline



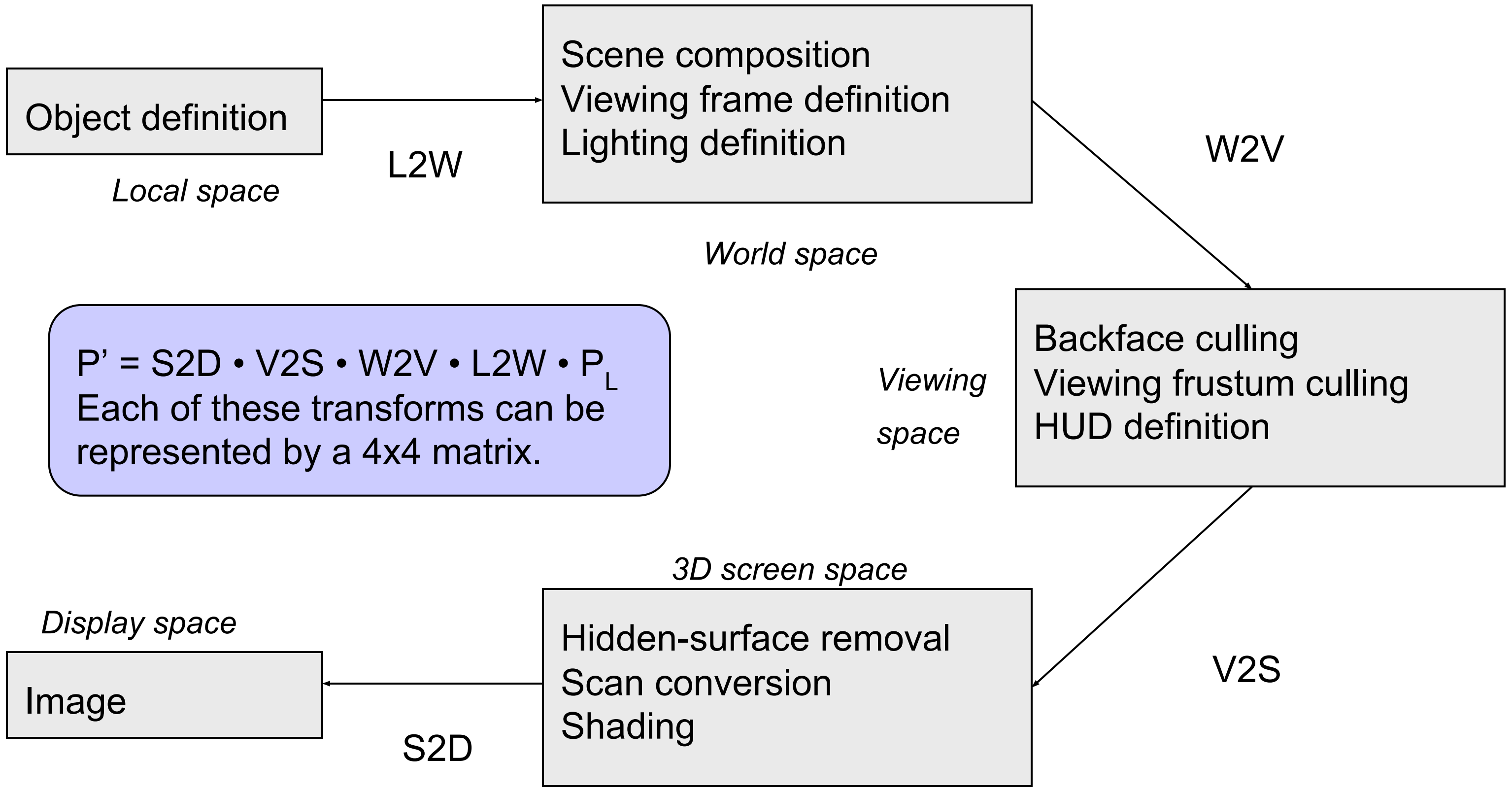
5) Collapse the box to a plane. Rasterize primitives using Z-axis information for depth-sorting and hidden-surface-removal.

Clip primitives to the screen.

Scale raster image to the final raster buffer and rasterize primitives.

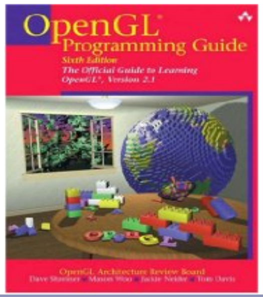


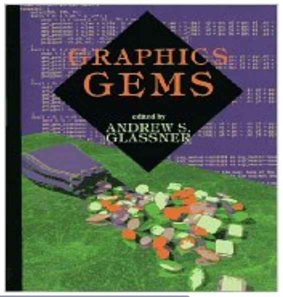
Recap: sketch of a rendering pipeline



$P' = S2D \cdot V2S \cdot W2V \cdot L2W \cdot P_L$
Each of these transforms can be represented by a 4x4 matrix.

Recommended reading





The OpenGL Programming Guide

- Some folks also favor *The OpenGL Superbible* for code samples and demos
- There's also an OpenGL-ES reference, same series

The *Graphics Gems* series by Glassner et al

- All the maths you've already forgotten

The NeonHelium online OpenGL tutorials

- http://nehe.gamedev.net/