Z-buffer Pipeline and OpenGL

Reading

- Required:
  - Watt, Section 5.2.2 – 5.2.4, 6.3, 6.6 (esp. intro and subsections 1, 4, and 8–10),
- Further reading:
  - Foley, et al, Chapter 5.6 and Chapter 6
  - David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1990, Chapter 2.
  - I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

Resources

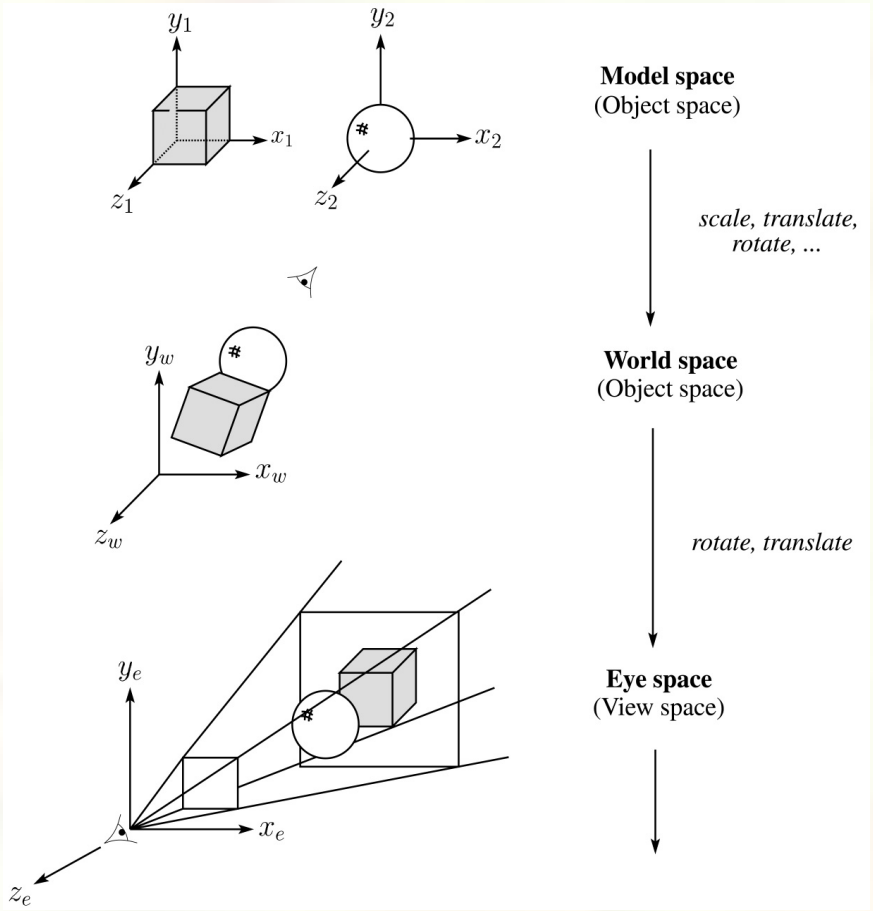
- The OpenGL Programming Guide, 7th Edition
- Interactive Computer Graphics: A Top-down Approach using OpenGL, 6th Edition
- The OpenGL Superbible, 5th Edition
- The OpenGL Shading Language Guide, 3rd Edition
- OpenGL and the X Window System
- OpenGL Programming for Mac OS X
- OpenGL ES 2.0
- WebGL (to appear)

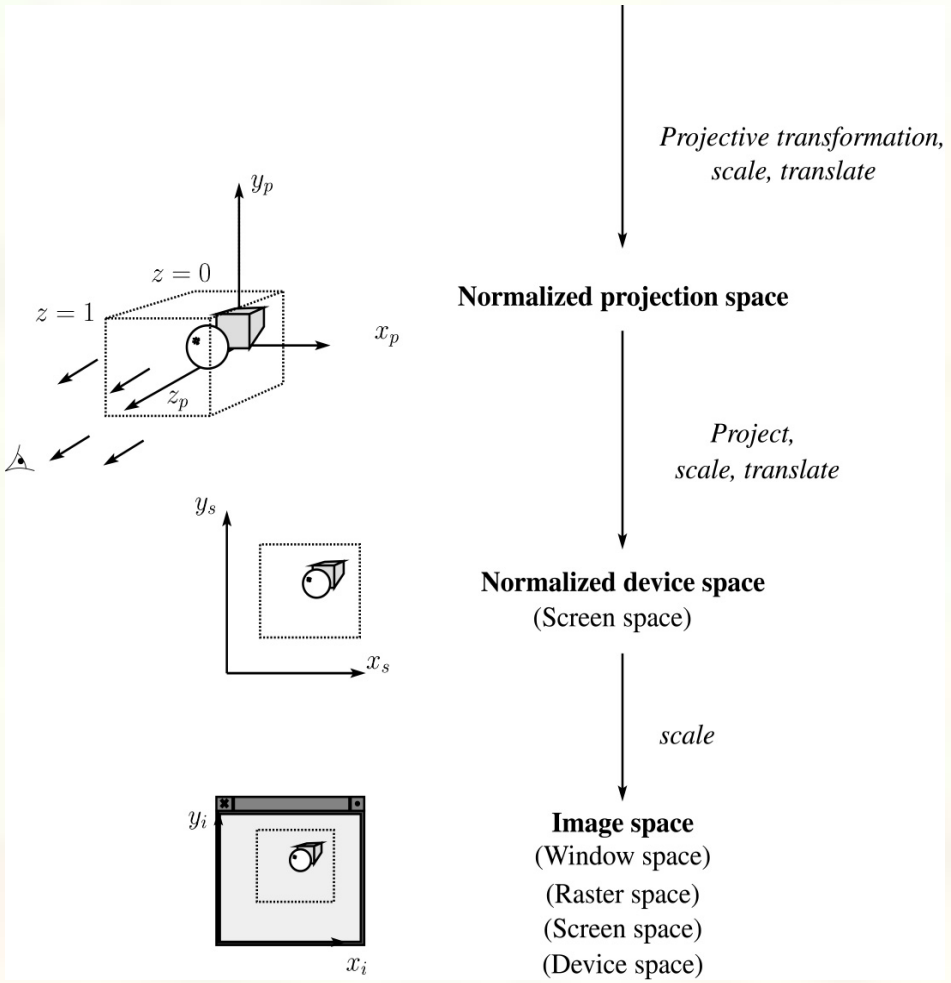
Resources

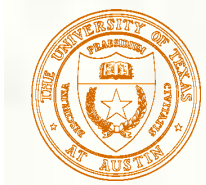
- The OpenGL Website: www.opengl.org
  - API specifications
  - Reference pages and developer resources
  - PDF of the OpenGL Reference Card
  - Discussion forums
- The Khronos Website: www.khronos.org
  - Overview of all Khronos APIs
  - Numerous presentations

3D Geometry Pipeline

- Before being turned into pixels by graphics hardware, a piece of geometry goes through a number of transformations...

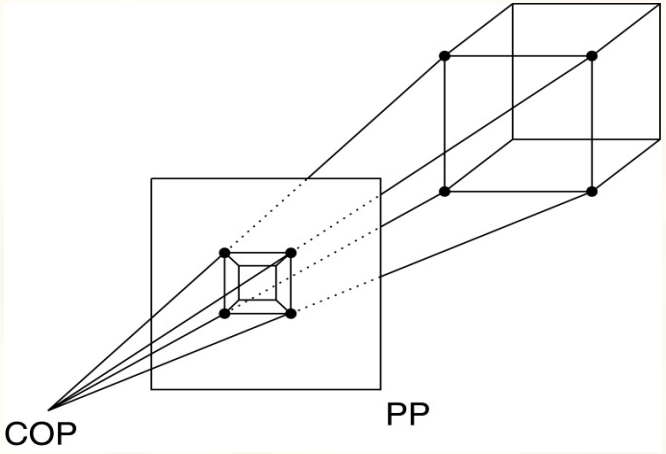


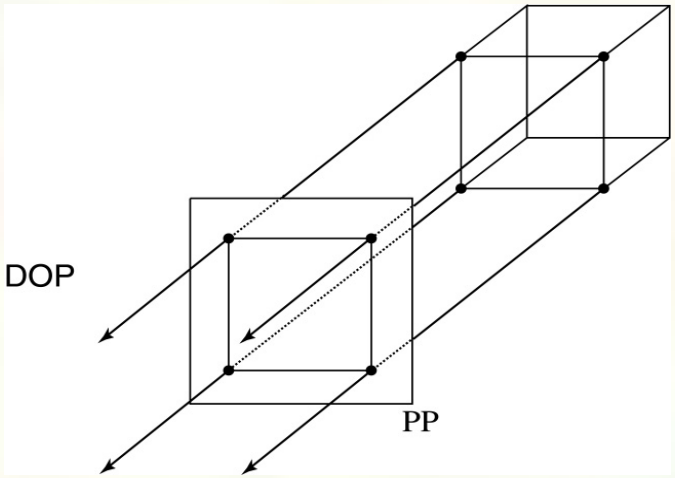




Projections

- **Projections** transform points in *n*-space to *m*-space, where *m*<*n*.
- In 3-D, we map points from 3-space to the **projection plane** (PP) along **projectors** emanating from the **center of projection** (COP):





- The center of projection is exactly the same as the pinhole in a pinhole camera.
- There are two basic types of projections:
  - Perspective – distance from COP to PP finite
  - Parallel – distance from COP to PP infinite

Parallel projections

- For parallel projections, we specify a **direction of projection** (DOP) instead of a COP.
- There are two types of parallel projections:
  - **Orthographic projection** – DOP perpendicular to PP
  - **Oblique projection** – DOP not perpendicular to PP
- We can write orthographic projection onto the z =0 plane with a simple matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

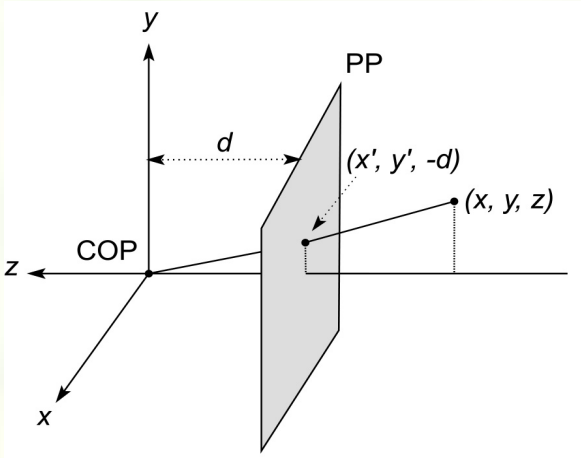
- But normally, we do not drop the z value right away. Why not?

Properties of parallel projection

- Properties of parallel projection:
  - Not realistic looking
  - Good for exact measurements
  - Are actually a kind of affine transformation
    - Parallel lines remain parallel
    - Angles not (in general) preserved
  - Most often used in CAD, architectural drawings, etc., where taking exact measurement is important

Derivation of perspective projection

- Consider the projection of a point onto the projection plane:



$$\frac{x'}{x} = -\frac{d}{z}$$

$$\frac{y'}{y} = -\frac{d}{z}$$

- By similar triangles, we can compute how much the x and y coordinates are scaled:

$$x' = -\frac{d}{z}x \qquad y' = -\frac{d}{z}y$$

- [Note: Watt uses a left-handed coordinate system, and he looks down the $+z$ axis, so his PP is at $+d$.]

Homogeneous coordinates revisited

- Remember how we said that affine transformations work with the last coordinate always set to one.
- What happens if the coordinate is not one?
- We divide all the coordinates by *W*:

$$\begin{bmatrix} X/W \\ Y/W \\ Z/W \\ W/W \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- If $W = 1$, then nothing changes.
- Sometimes we call this division step the "perspective divide."

Homogeneous coordinates and perspective projection

- Now we can re-write the perspective projection as a matrix equation:

$$\begin{bmatrix} X \\ Y \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -z/d \end{bmatrix}$$

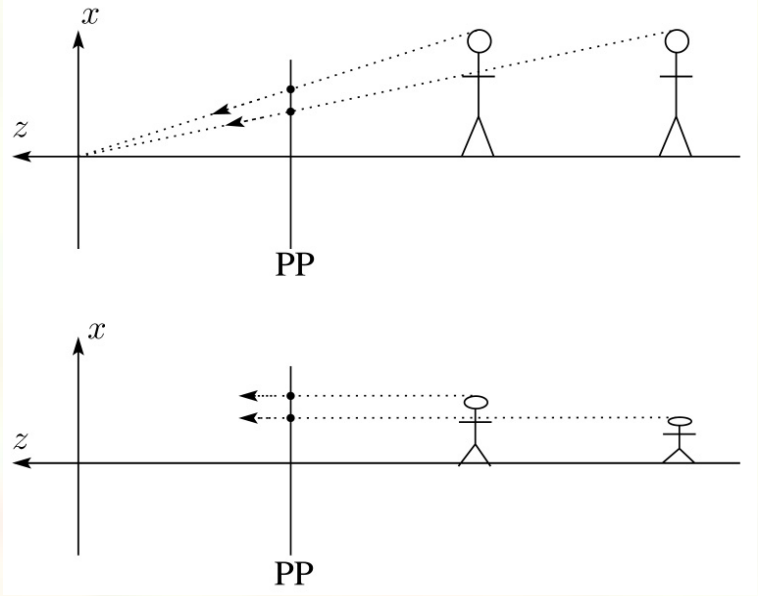
- After division by W, we get:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{x}{z}d \\ -\frac{y}{z}d \\ 1 \end{bmatrix}$$

- Again, projection implies dropping the z coordinate to give a 2D image, but we usually keep it around a little while longer.

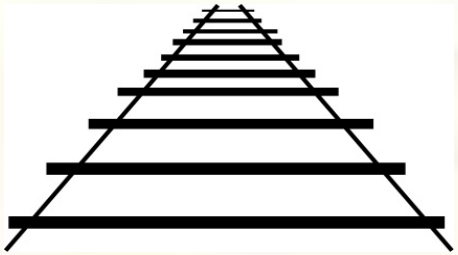
Projective normalization

- After applying the perspective transformation and dividing by *w*, we are free to do a simple parallel projection to get the 2D image.
- What does this imply about the shape of things after the perspective transformation + divide?



Vanishing points

- What happens to two parallel lines that are not parallel to the projection plane?
- Think of train tracks receding into the horizon...



- The equation for a line is: $\ell = \mathbf{p} + t\mathbf{v} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} + t \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$
- After perspective transformation we get: $\begin{bmatrix} X \\ Y \\ W \end{bmatrix} = \begin{bmatrix} p_x + tv_x \\ p_y + tv_y \\ -(p_z + tv_z)/d \end{bmatrix}$

Vanishing points (cont'd)

- Dividing by *W*:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\dfrac{p_x + tv_x}{p_z + tv_z} d \\ -\dfrac{p_y + tv_y}{p_z + tv_z} d \\ \dfrac{-(p_z + tv_z)/d}{-(p_z + tv_z)/d} \end{bmatrix}$$

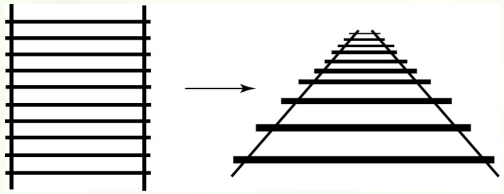
- Letting *t* go to infinity:

$$\begin{bmatrix} -\dfrac{v_x}{v_z} \\ -\dfrac{v_y}{v_z} \\ 1 \\ \\ \end{bmatrix}$$

- We get a point that depends only on **v**
- What happens to the line $\ell = \mathbf{q} + t\mathbf{v}$?
- Each set of parallel lines intersect at a **vanishing point** on the PP.
- **Q**: How many vanishing points are there?

Properties of perspective projections

- The perspective projection is an example of a **projective transformation**.



- Here are some properties of projective transformations:
  - Lines map to lines
  - Parallel lines do not necessarily remain parallel
  - Ratios are not preserved
- One of the advantages of perspective projection is that size varies inversely with distance – looks realistic.
- A disadvantage is that we can't judge distances as exactly as we can with parallel projections.
- **Q**: Why did nature give us eyes that perform perspective projections?
- **Q**: Do our eyes "see in 3D"?

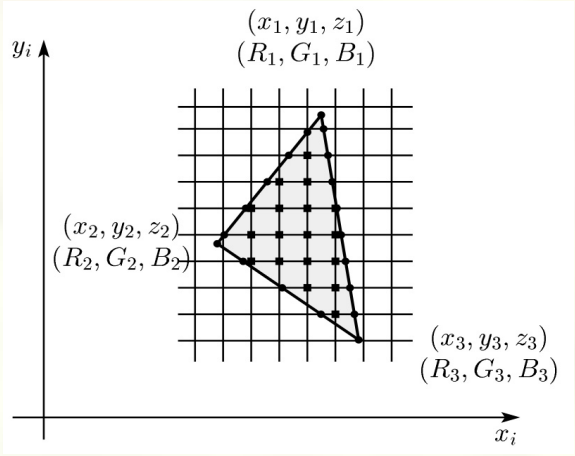
Z-buffer

- We can use projections for **hidden surface elimination**.
- The **Z-buffer'** or **depth buffer** algorithm [Catmull, 1974] is probably the simplest and most widely used of these techniques.
- Here is pseudocode for the Z-buffer hidden surface algorithm:

```
for each pixel (i,j) do
    Z-buffer [i,j] ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each polygon A do
    for each pixel in A do
        Compute depth z and shade s of A at (i,j)
        if z > Z-buffer [i,j] then
            Z-buffer [i,j] ← z
            Framebuffer[i,j] ← s
        end if
    end for
end for
```

Z-buffer, cont'd

- The process of filling in the pixels inside of a polygon is called **rasterization**.
- During rasterization, the z value and shade s can be computed incrementally (fast!).



Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Ray tracing vs. Z-Buffer

Ray tracing:

```
for each ray {
 for each object {
     test for intersection
 }
}
```

Z-Buffer:

```
for each object {
project_onto_screen;
  for each ray {
     test for intersection
  }
}
```

In both cases, optimizations are applied to the inner loop.

Biggest differences:

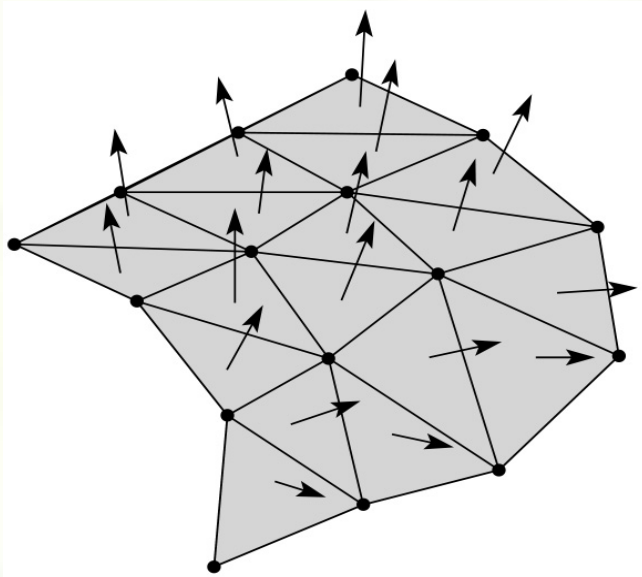
- ray order vs. object order
- Z-buffer does some work in screen space
- Z-buffer restricted to rays from a single center of projection!

Gouraud vs. Phong interpolation

- Does Z-buffer graphics hardware do a full shading calculation at every point? Not in the past, but this has changed!
- Smooth surfaces are often approximated by polygonal facets, because:
  - Graphics hardware generally wants polygons (esp. triangles).
  - Sometimes it easier to write ray-surface intersection algorithms for polygonal models.
- How do we compute the shading for such a surface?

Faceted shading

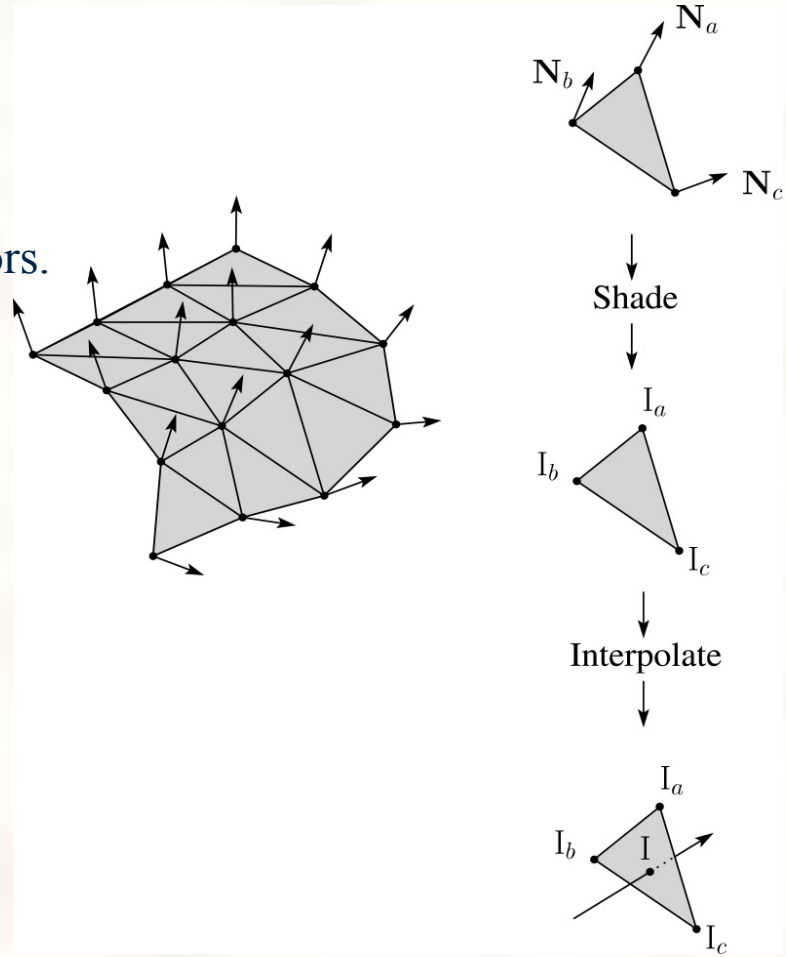
- Assume each face has a constant normal:



- For a distant viewer and a distant light source, how will the color of each triangle vary?
- Result: faceted, not smooth, appearance.

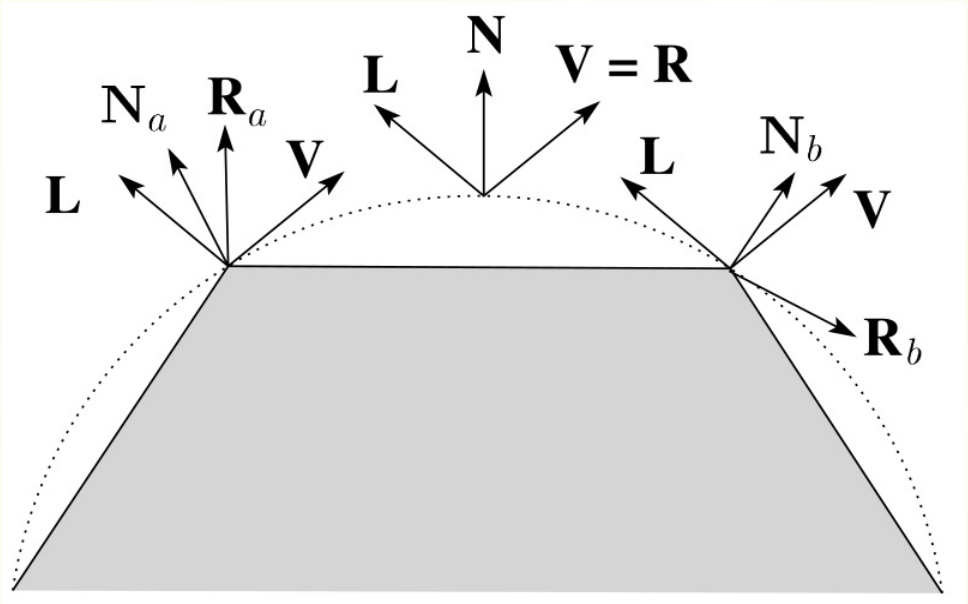
Gouraud interpolation

- To get a smoother result that is easily performed in hardware, we can do **Gouraud interpolation**.
- Here's how it works:
  - Compute normals at the vertices.
  - Shade only the vertices.
  - Interpolate the resulting vertex colors.



Gouraud interpolation, cont'd

- Gouraud interpolation has significant limitations.
  - If the polygonal approximation is too coarse, we can miss specular highlights.

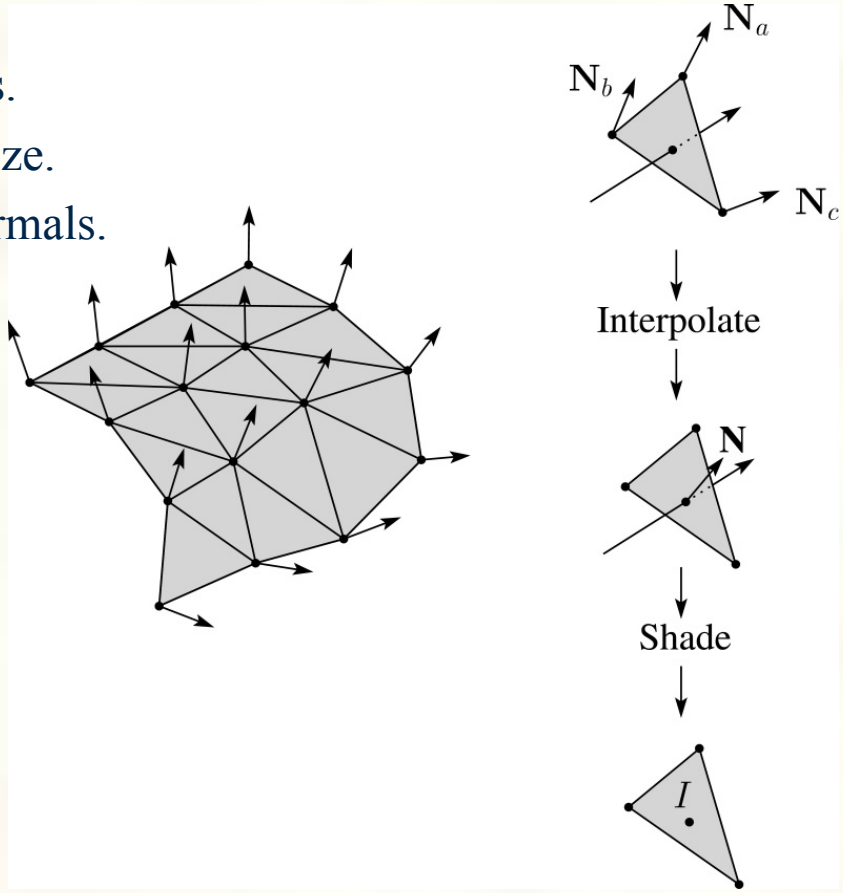


  - We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

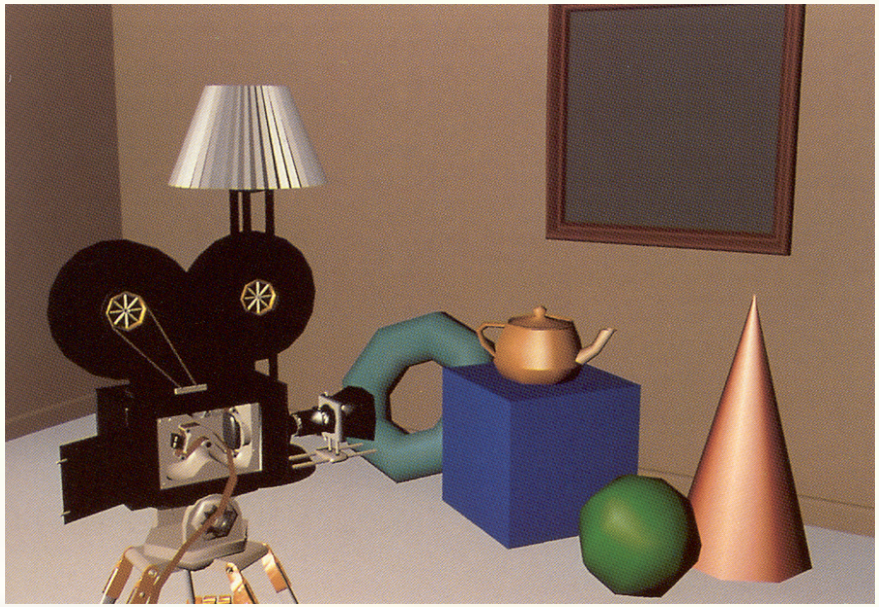
- Alas, this is usually what graphics hardware supported until very recently.
- But new graphics hardware supports…

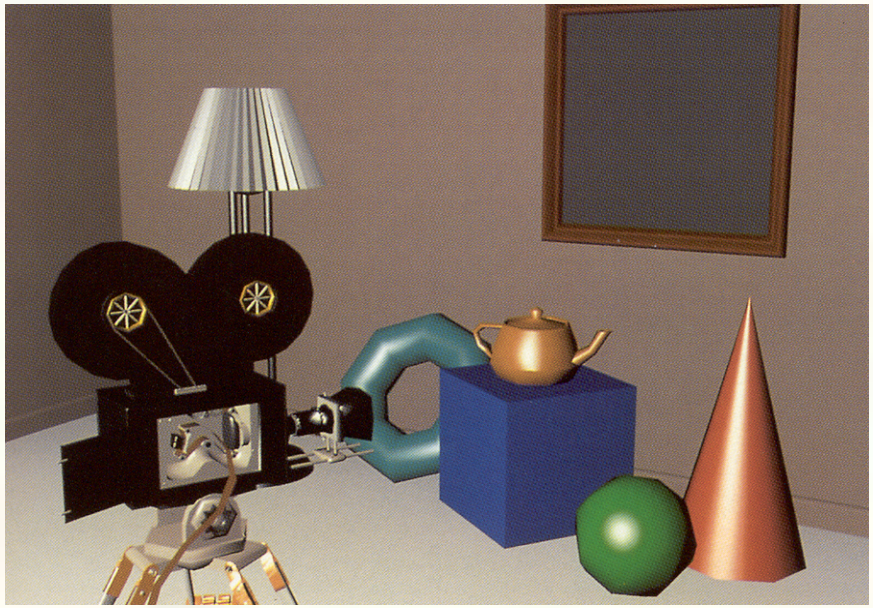
Phong interpolation

- To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.
- Here's how it works:
  1. Compute normals at the vertices.
  2. Interpolate normals and normalize.
  3. Shade using the interpolated normals.



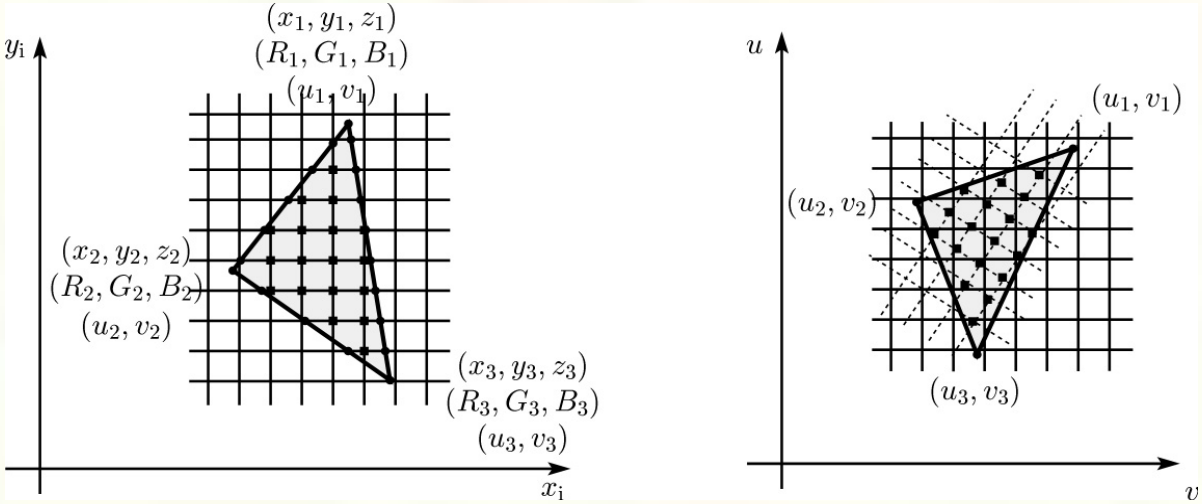
Gouraud vs. Phong interpolation





Texture mapping and the z-buffer

- Texture-mapping can also be handled in z-buffer algorithms.
- Method:
  - Scan conversion is done in screen space, as usual
  - Each pixel is colored according to the texture
  - Texture coordinates are found by Gouraud-style interpolation

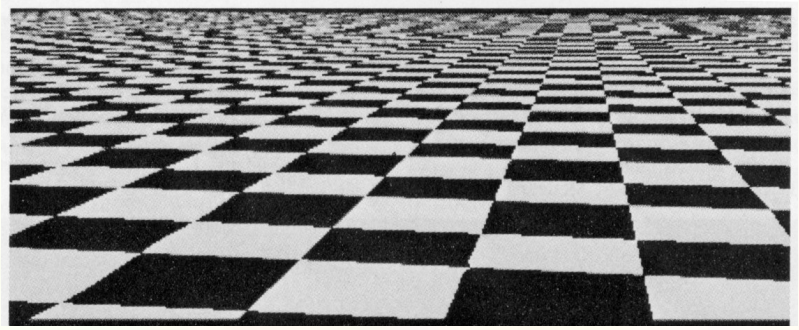


- Note: Mapping is more complicated if you want to do perspective right!

  - linear in world space != linear in screen space

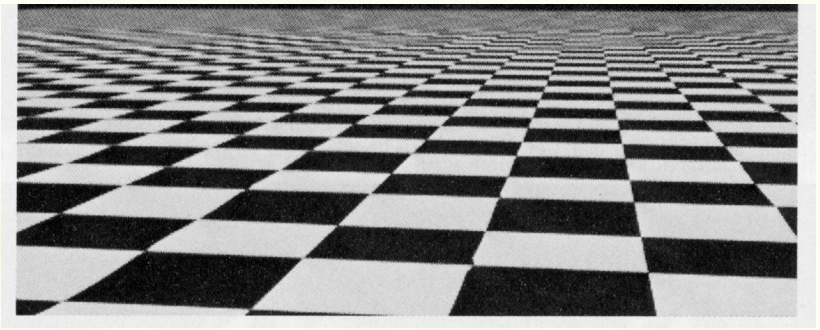
Antialiasing textures

- If you render an object with a texture map using point-sampling, you can get aliasing:



From Crow, SIGGRAPH '84

- Proper antialiasing requires area averaging over pixels:

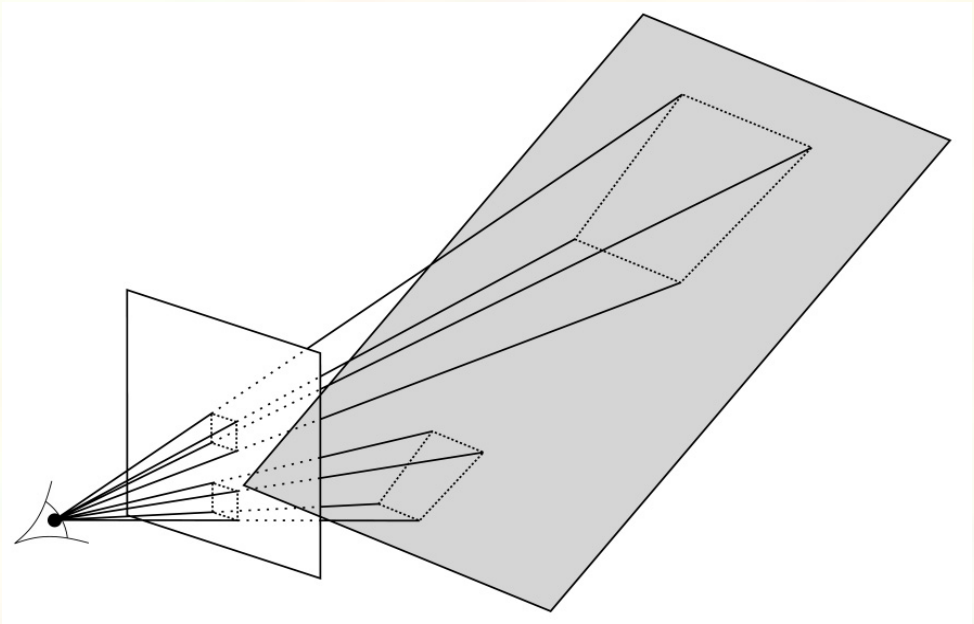


From Crow, SIGGRAPH '84

- In some cases, you can average directly over the texture pixels to do the anti-aliasing.

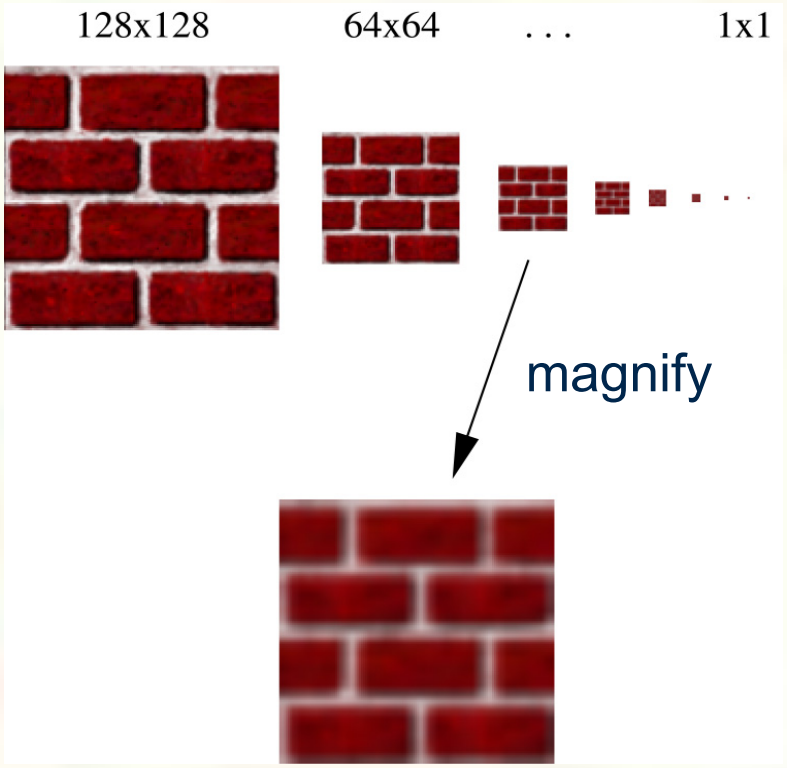
Computing the average color

- The computationally difficult part is summing over the covered pixels.
- Several methods have been used.
- The simplest is **brute force**:
  - Figure out which texels are covered and add up their colors to compute the average.

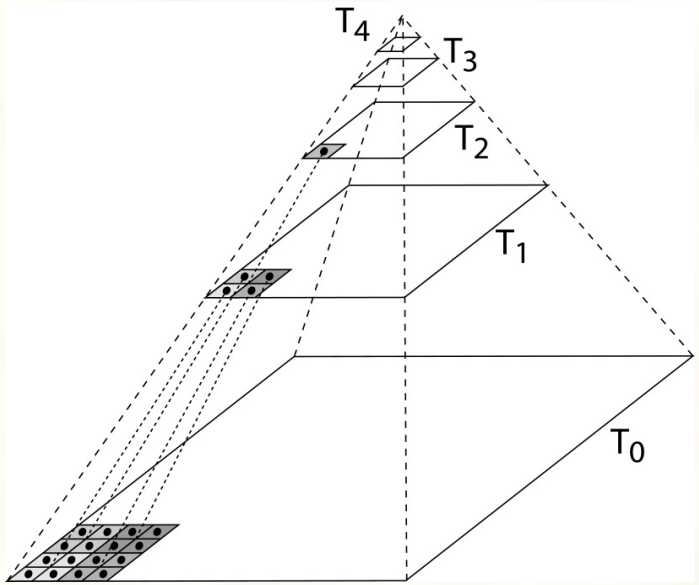


Mipmaps

- A faster method is **mip maps** developed by Lance Williams in 1983:
  - Stands for "multum in parvo" – many things in a small place
  - Keep textures prefiltered at multiple resolutions
  - Has become the graphics hardware standard



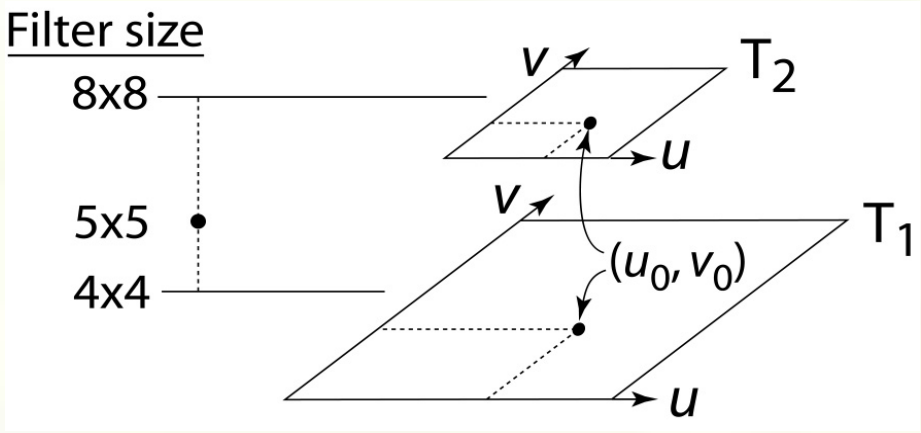
Mipmap pyramid



- The mip map hierarchy can be thought of as an image pyramid:
  - Level 0 ($T_0[i,j]$) is the original image.
  - Level 1 ($T_1[i,j]$) averages over 2x2 neighborhoods of original.
  - Level 2 ($T_2[i,j]$) averages over 4x4 neighborhoods of original
  - Level 3 ($T_3[i,j]$) averages over 8x8 neighborhoods of original

- What's a fast way to pre-compute the texture map for each level?

Mipmap resampling



- What would the mipmap return for an average over a 5 x 5 neighborhood at location (u_0, v_0)?
- How do we measure the fractional distance between levels?
- What if you need to average over a non-square region?

Summed area tables

- A more accurate method than mipmaps is **summed area tables** invented by Frank Crow in 1984.
- Recall from calculus:

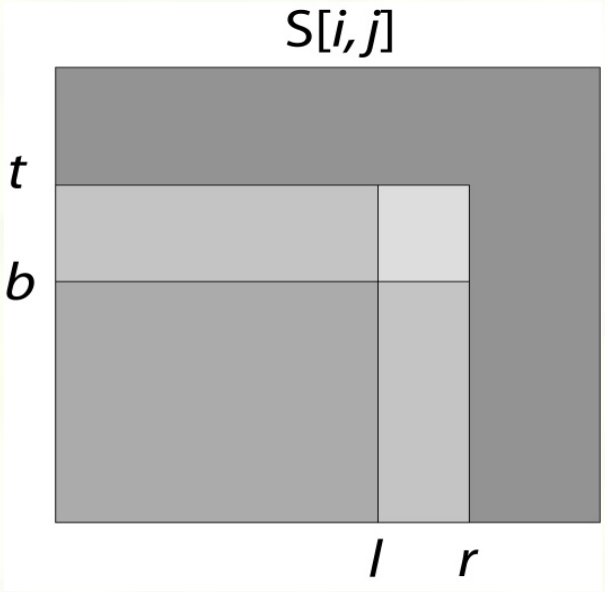
$$\int_a^b f(x)dx = \int_{-\infty}^{b} f(x)dx - \int_{-\infty}^{a} f(x)dx$$

- In discrete form: $$\sum_{i=k}^{m} f[i] = \sum_{i=0}^{m} f[i] - \sum_{i=0}^{k} f[i]$$

- **Q**: If we wanted to do this real fast, what might we pre-compute?

Summed area tables (cont'd)

- We can extend this idea to 2D by creating a table, S[*i*,*j*], that contains the sum of everything below and to the left.



- **Q**: How do we compute the average over a region from (*l*, *b*) to (*r*, *t*)?
- Characteristics:
  - Requires more memory and precision
  - Gives less blurry textures

Comparison of techniques

Point sampled

MIP-mapped

Summed area table

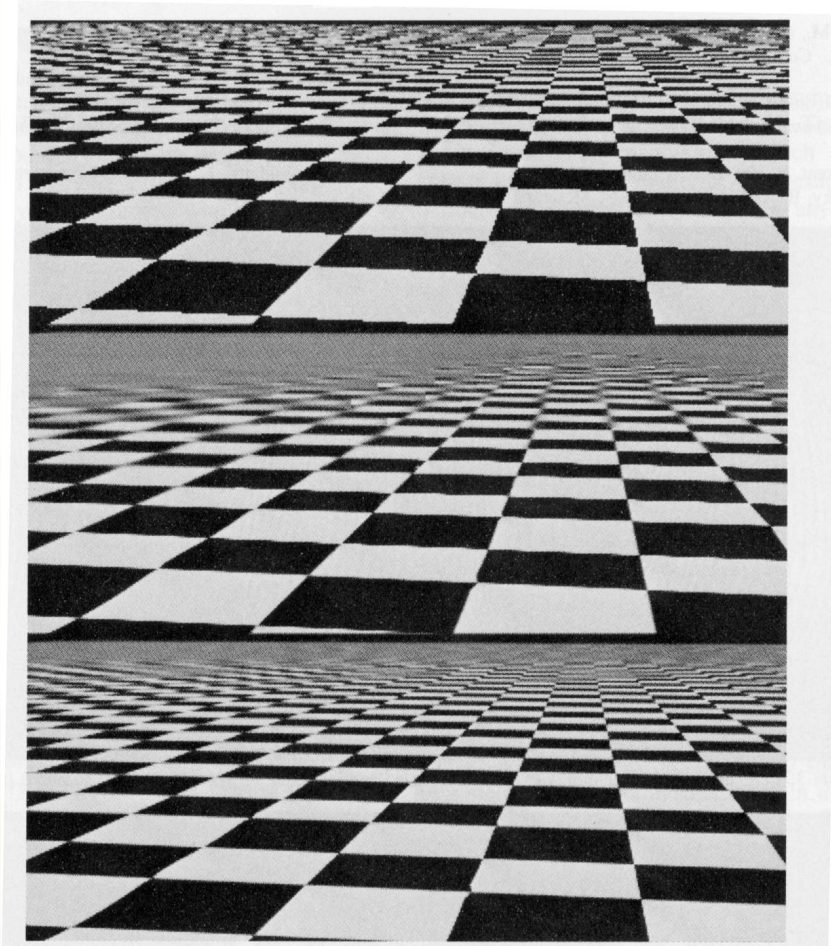


Figure 5: CheckerBoards mapped onto a square showing vertically compressed texture.

From Crow, SIGGRAPH '84

Cost of Z-buffering

- Z-buffering is ***the*** algorithm of choice for hardware rendering (today), so let's think about how to make it run as fast as possible…
- The steps involved in the Z-buffer algorithm are:

1. Send a triangle to the graphics hardware.
2. Transform the vertices of the triangle using the modeling matrix.
3. Transform the vertices using the projection matrix.
4. Set up for incremental rasterization calculations
5. Rasterize
   (generate "fragments" = potential pixels)
6. Shade at each fragment
7. Update the framebuffer according to *z*.

- What is the overall cost of Z-buffering?

Cost of Z-buffering, cont'd

- We can approximate the cost of this method as:

$$k_{bus}v_{bus} + k_{xform}v_{xform} + k_{setup}t + k_{shade}(dm^2)$$

where:

k_{bus} = bus cost to send a vertex

v_{bus} = number of vertices sent over the bus

k_{xform} = cost of transforming a vertex

v_{xform} = number of vertices transformed

k_{setup} = cost of setting up for rasterization

t = number of triangles being rasterized

k_{shade} = cost of shading a fragment

d = depth complexity
(average times a pixel is covered)

m^2 = number of pixels in frame buffer

Accelerating Z-buffers

- Given this cost function:

$$k_{bus} v_{bus} + k_{xform} v_{xform} + k_{setup} t + k_{shade} (dm^2)$$

what can we do to accelerate Z-buffering?

Accel method	v_{bus}	v_{xform}	t	d	m

Introduction to Modern OpenGL Programming

Adapted from SIGGRAPH 2012 slides by

Ed Angel
University of New Mexico

and

Dave Shreiner
ARM, Inc

Outline

- Evolution of the OpenGL Pipeline
- A Prototype Application in OpenGL
- OpenGL Shading Language (GLSL)
- Vertex Shaders
- Fragment Shaders
- Examples

What Is OpenGL?

- OpenGL is a computer graphics *rendering* API
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

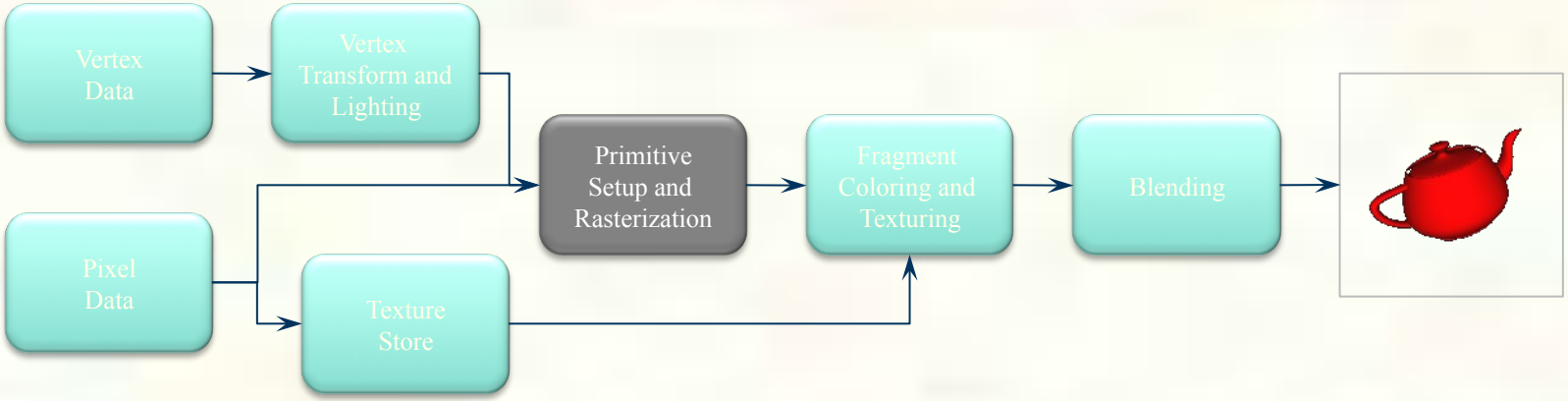
This is the “new” OpenGL

- We’ll concentrate on the latest versions of OpenGL
- They enforce a new way to program with OpenGL
  - Allows more efficient use of GPU resources
- If you’re familiar with “classic” graphics pipelines, modern OpenGL doesn’t support
  - Fixed-function graphics operations
    - lighting
    - transformations
- All applications must use shaders for their graphics processing

The Evolution of the OpenGL Pipeline

In the Beginning …

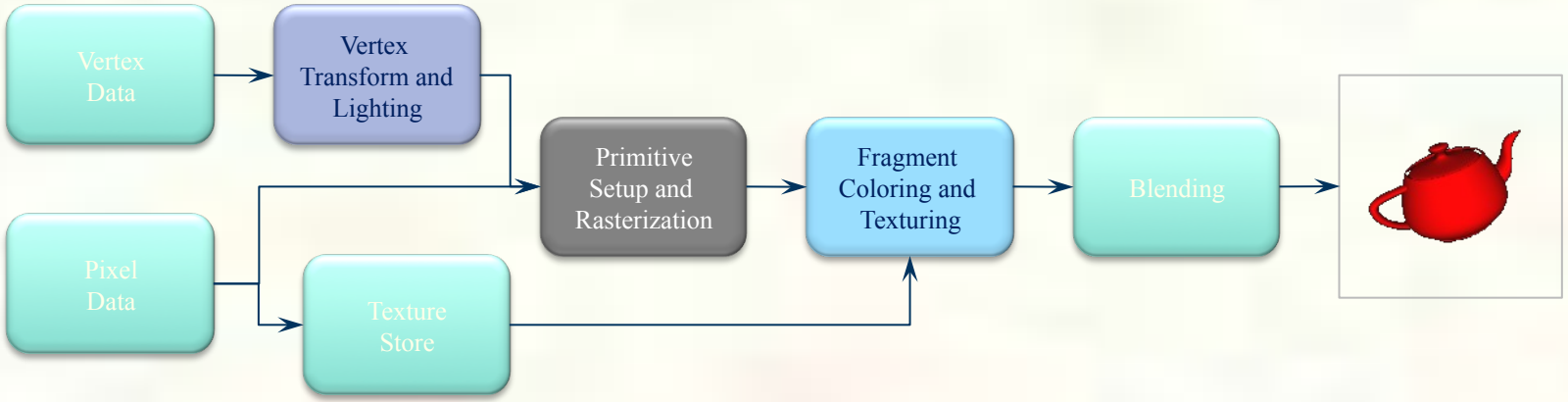
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



- The pipeline evolved, but remained fixed-function through OpenGL versions 1.1 through 2.0 (Sept. 2004)

The Start of the Programmable Pipeline

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



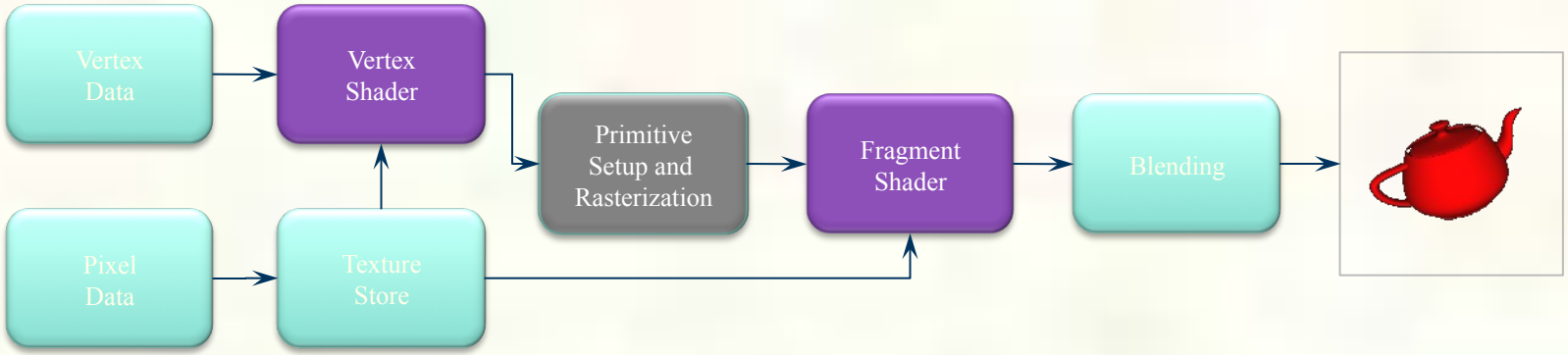
An Evolutionary Change

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

The Exclusively Programmable Pipeline

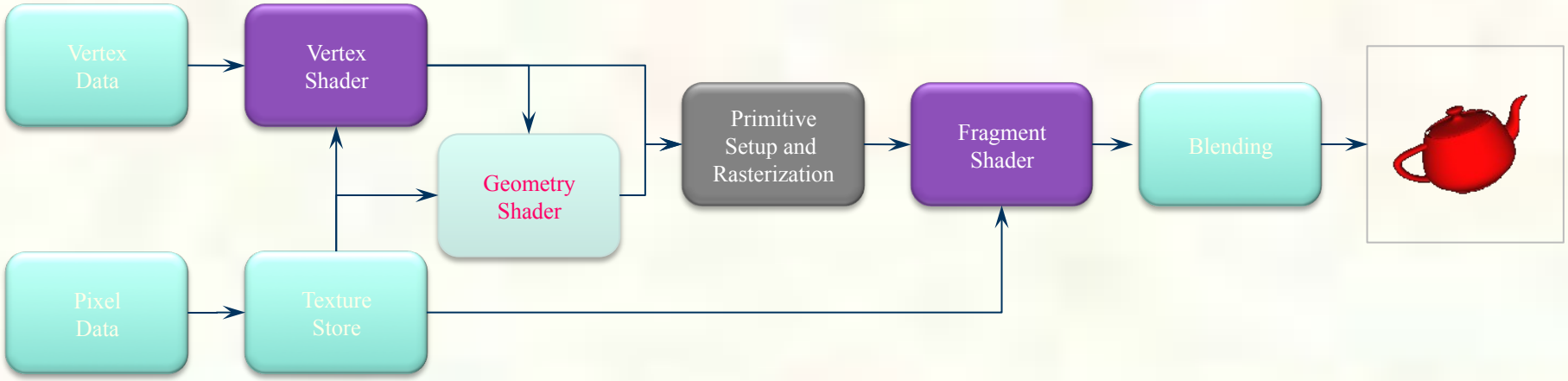
- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders



- Additionally, almost all data is *GPU-resident*
  - all vertex data sent using buffer objects

More Programmability

- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – *geometry shaders*



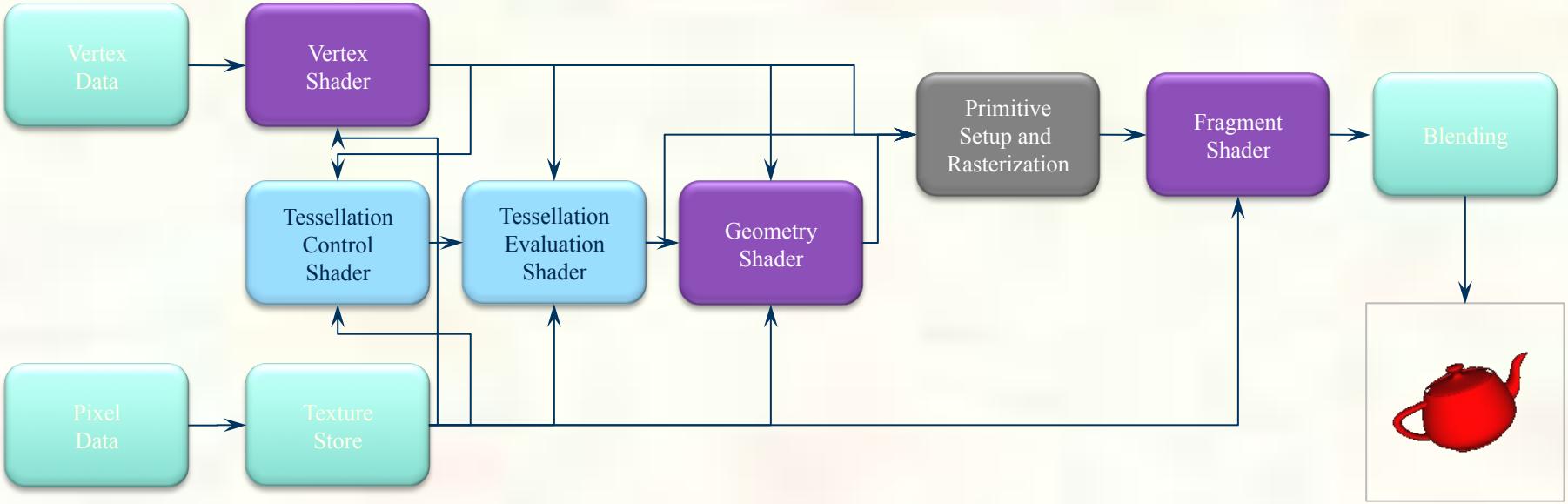
More Evolution – Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
  - currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control and tessellation-evaluation shaders*
- Latest version is 4.3

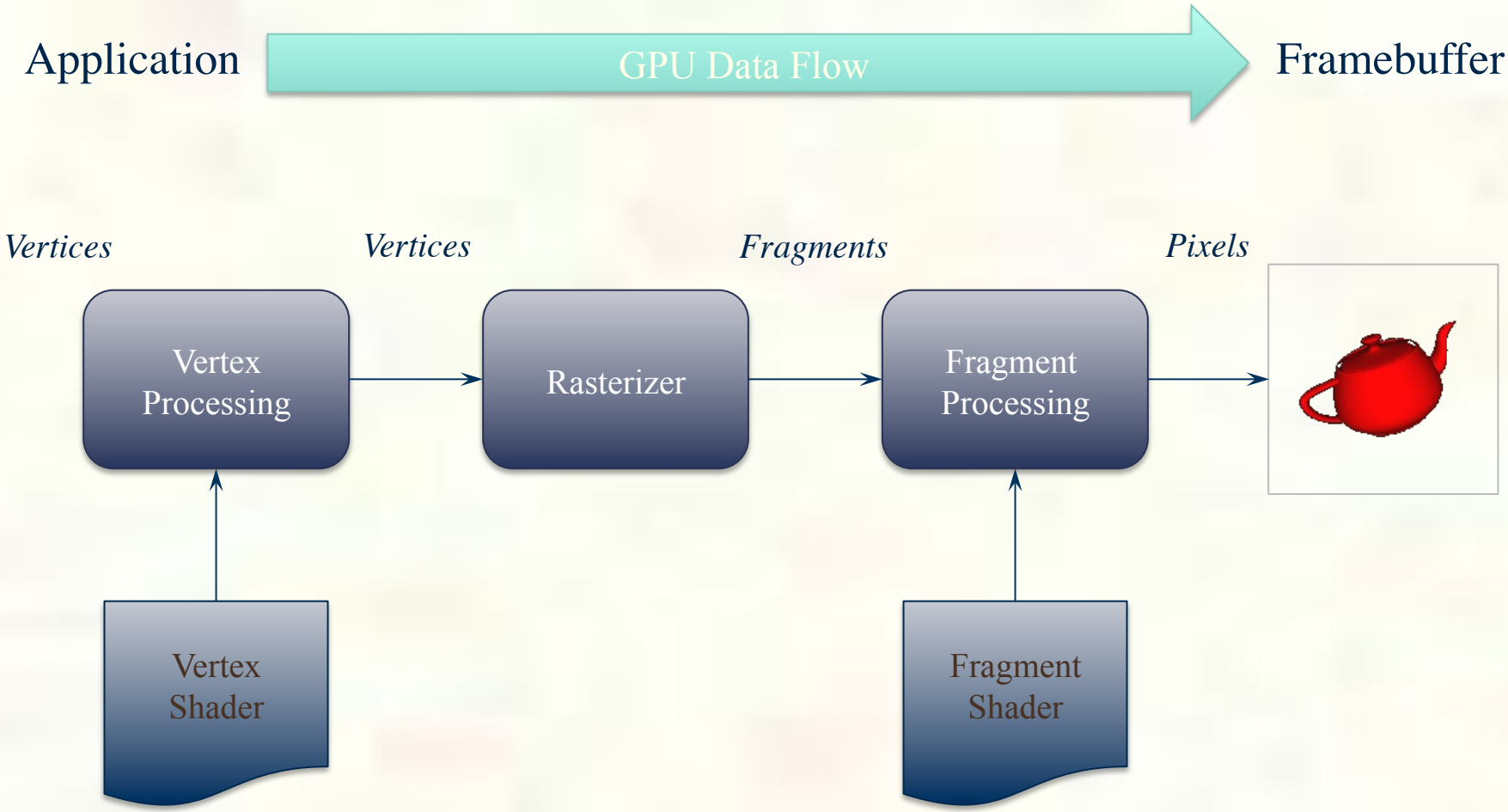


OpenGL ES and WebGL

- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

OpenGL Application Development

A Simplified Pipeline Model



OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  1. Create shader programs
  2. Create buffer objects and load data into them
  3. “Connect” data locations with shader variables
  4. Render

Application Framework Requirements

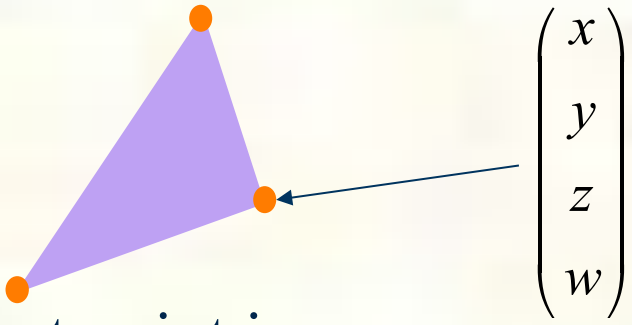
- OpenGL applications need a place to render into
  - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut)
  - simple, open-source library that works everywhere
  - handles all windowing operations:
    - opening windows
    - input processing

Simplifying Working with OpenGL

- Operating systems deal with library functions differently
  - compiler linkage and runtime libraries may expose different functions
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent
- We use another open-source library, GLEW, to hide those details

Representing Geometric Objects

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space



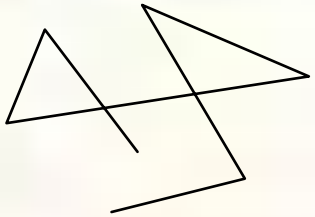
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in *vertex buffer objects* (VBOs)
- VBOs must be stored in *vertex array objects* (VAOs)

OpenGL's Geometric Primitives

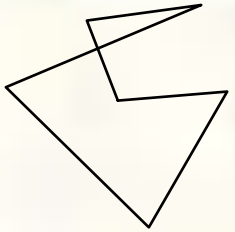
- All primitives are specified by vertices

GL_POINTS

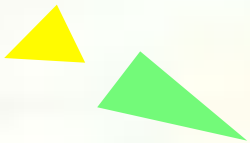
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

A First Program

Rendering a Cube

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a vec4 class (implemented in C++) similar to GLSL's vec4 type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

Initializing the Cube's Data (cont'd)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
point4  points[NumVertices];
color4  colors[NumVertices];
```

Cube Data

```
// Vertices of a unit cube centered at origin, sides aligned
   with axes

point4 vertex_positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

```
// quad() generates two triangles for each face and assigns
   colors to the vertices
int Index = 0;  // global variable indexing into VBO arrays

void quad(int a, int b, int c, int d) {
    colors[Index] = vertex_colors[a]; points[Index] =
   vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] =
   vertex_positions[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] =
   vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] =
   vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] =
   vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] =
   vertex_positions[d]; Index++;
}
```

Generating the Cube from Faces

```
// generate 12 triangles: 36 vertices and 36
   colors
void
colorcube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Vertex Array Objects (VAOs)

- VAOs store the data of a geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in Code

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling `glBindBuffer(GL_ARRAY_BUFFER, …)`
  - load data into VBO using `glBufferData(GL_ARRAY_BUFFER, …)`
  - bind VAO for use in rendering `glBindVertexArray()`

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
             sizeof(colors), NULL,
  GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0,
                sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
  sizeof(colors), colors);
```

Connecting Vertex Shaders with Geometry

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation(program,
  "vPosition”);
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT,
  GL_FALSE, 0, BUFFER_OFFSET(0));
GLuint vColor = glGetAttribLocation(program,
  "vColor”);
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT,
  GL_FALSE, 0, BUFFER_OFFSET(sizeof(points)));
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```

- Usually invoked in display callback
- Initiates vertex shader

Shaders and GLSL

GLSL Data Types

Scalar types: float, int, bool

Vector types: vec2, vec3, vec4
ivec2, ivec3, ivec4
bvec2, bvec3, bvec4

Matrix types: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D, sampler3D, samplerCube

C++ style constructors: vec3 a = vec3(1.0, 2.0, 3.0);

Operators

- Standard C/C++ arithmetic and logic operators
- Operators overloaded for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

For vectors can use [ ], xyzw, rgba or stpq

Example:

```
vec3 v;
```

`v[1], v.y, v.g, v.t` all refer to the same element

Swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

- in, out
  - Copy vertex attributes and other variables to/from shaders
  - `in vec2 tex_coord;`
  - `out vec4 color;`
- Uniform: variable from application
  - `uniform float time;`
  - `uniform vec4 rotation;`

Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Functions

- Built in
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect
- User defined

Built-in Variables

- `gl_Position`: output position from vertex shader
- `gl_FragColor`: output color from fragment shader
  - Only for ES, WebGL and older versions of GLSL
  - Present version use an out variable

Simple Vertex Shader for Cube

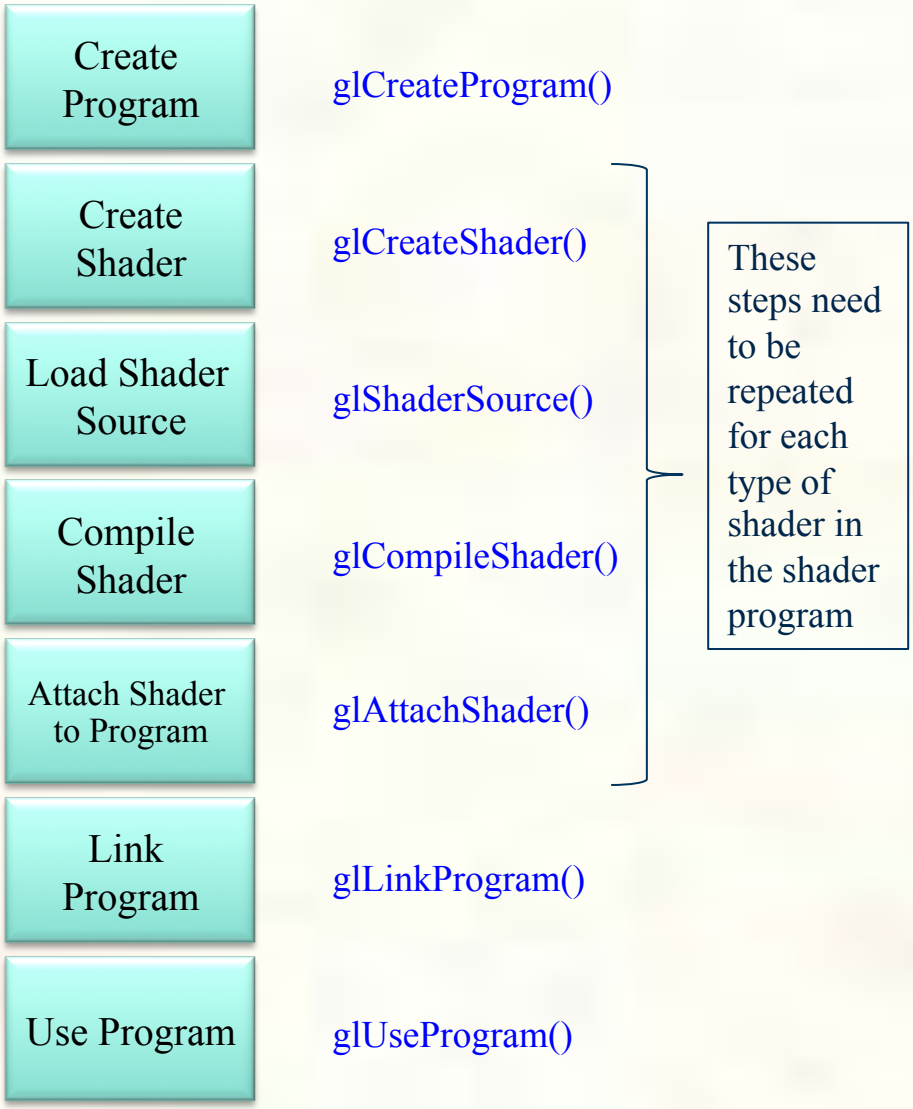
```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main() {
    color = vColor;
    gl_Position = vPosition;
}
```

The Simplest Fragment Shader

```
in vec4 color;
out vec4 FragColor;

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

Getting Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



A Simpler Way

- We've created a routine for this course to make it easier to load your shaders
  - available at course website

```
GLuint InitShaders( const char* vFile, const char*
                    fFile);
```

- `InitShaders` takes two filenames
  - `vFile` for the vertex shader
  - `fFile` for the fragment shader
- Fails if shaders don't compile, or program doesn't link

Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

Determining Locations After Linking

Assumes you already know the variables' name

```
GLint  idx =
  glGetAttribLocation(program, "name");

GLint  idx =
  glGetUniformLocation(program, "name");
```

Initializing Uniform Variable Values

Uniform Variables

```
glUniform4f(index, x, y, z, w);

Glboolean transpose = GL_TRUE;
    // Since we're C programmers
Glfloat mat[3][4][4] = { … };

glUniformMatrix4fv(index, 3, transpose, mat);
```

Finishing the Cube Program

```
int main(int argc, char **argv) {
 glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE |
  GLUT_DEPTH);
  glutInitWindowSize(512, 512);
  glutCreateWindow("Color Cube”);
  glewInit();
  init();
  glutDisplayFunc(display);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return 0;
}
```

Cube Program GLUT Callbacks

```
void display(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y) {
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

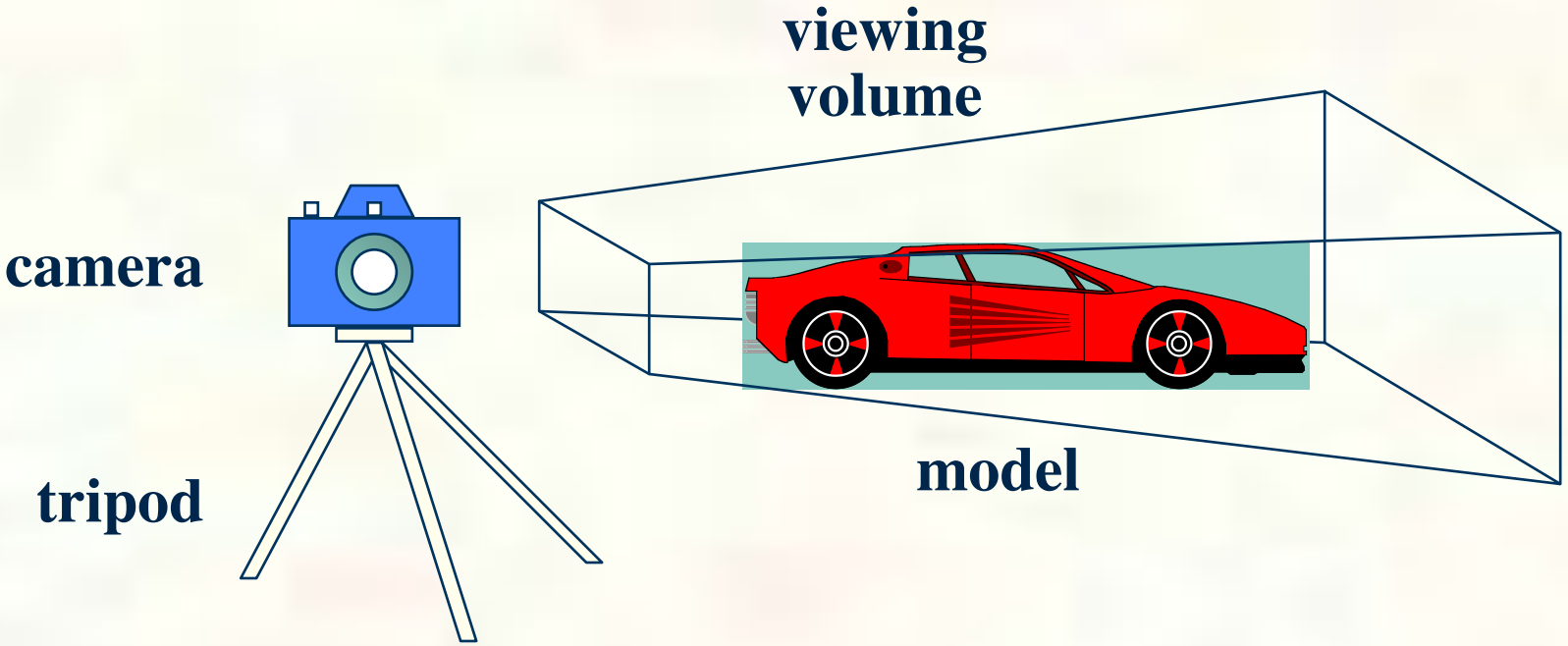
Vertex Shader Examples

- A vertex shader is initiated by each vertex output by `glDrawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions

Transformations

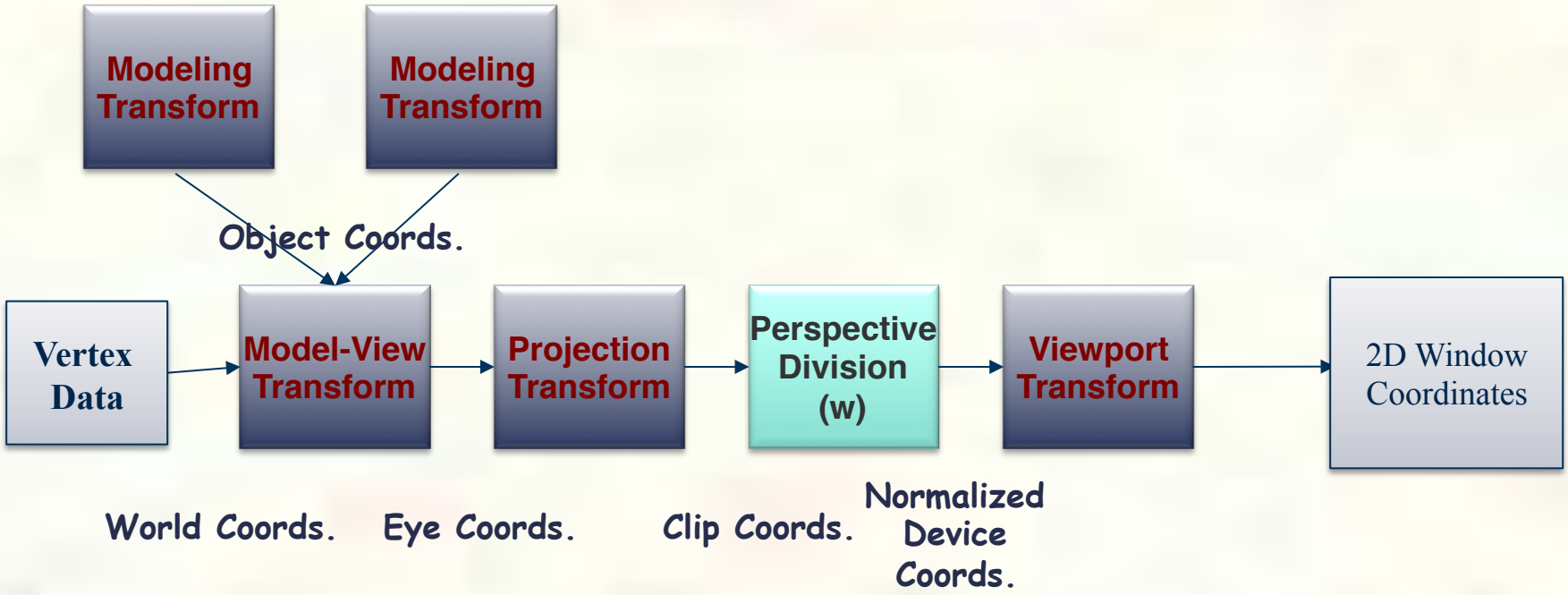
Camera Analogy

3D is just like taking a photograph (lots of photographs!)



Transformations

- Transformations take us from one "space" to another
  - All of our transforms are 4×4 matrices



Camera Analogy Transform Sequence

- Modeling transformations
  - assemble the world and move the objects
- Viewing transformations
  - define position and orientation of the viewing volume in the world
- Projection transformations
  - adjust the lens of the camera
- Viewport transformations
  - enlarge or reduce the physical photograph

3D Homogeneous Transformations

- A vertex is transformed by 4×4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
    - this is opposite of what "C" programmers expect
- matrices are always post-multiplied
- product of matrix and vector is

$$\mathbf{M}\vec{v}$$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

View Specification

- Set up a *viewing frustum* to specify how much of the world we can see
- Done in two steps
  - specify the size of the frustum (*projection transform*)
  - specify its location in space (*model-view transform*)
- Anything outside of the viewing frustum is *clipped*
  - primitive is either modified or discarded (if entirely outside frustum)

View Specification (cont'd)

- OpenGL projection model uses *eye coordinates*
  - the "eye" is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom, left & right

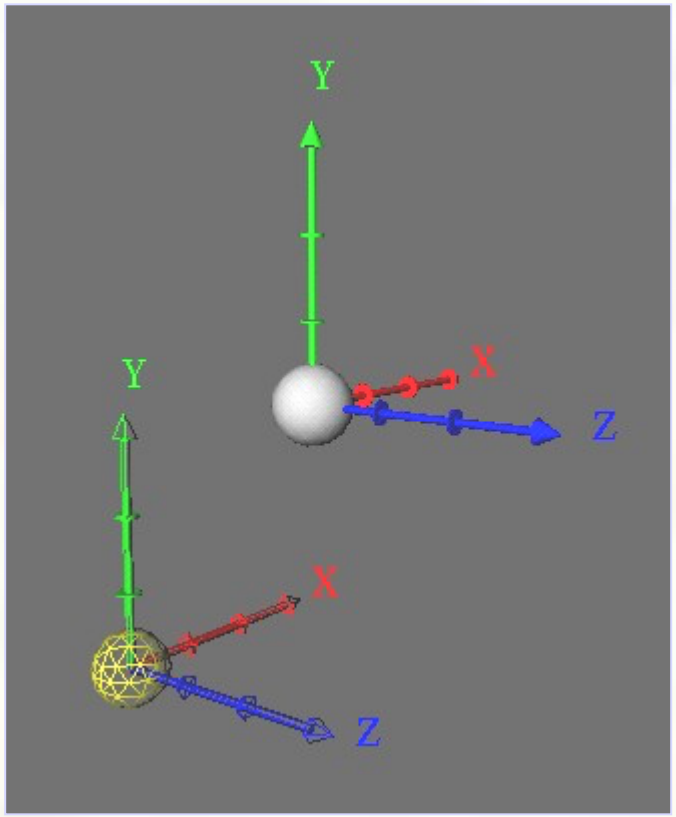
Viewing Transformations

- Position the camera/eye in the scene
- To "fly through" a scene
  - change viewing transformation and redraw scene
- `LookAt(`eye_x`, `eye_y`, `eye_z`,`
  `look`$_x$`, look`$_y$`, look`$_z$`,`
  `up`$_x$`, up`$_y$`, up`$_z$`)`
  - up vector determines unique orientation
  - careful of degenerate positions

Translation

Move object or change frame origin

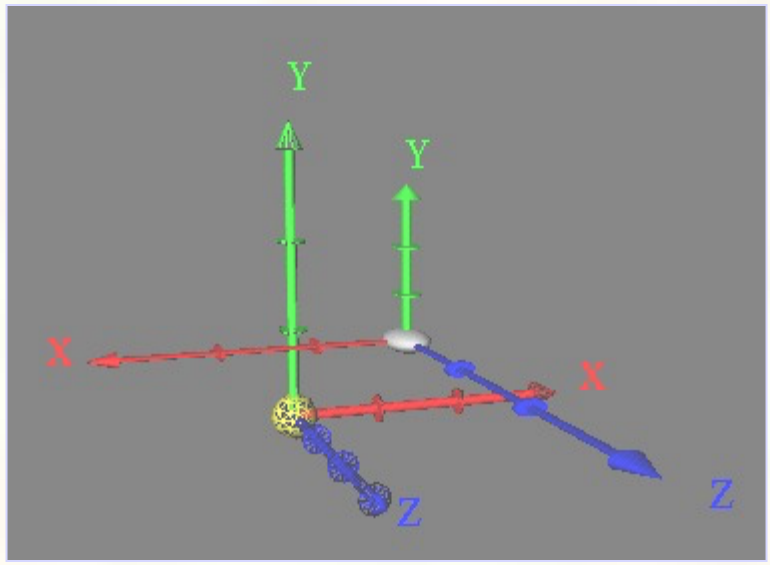
$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Scale

Stretch, mirror or decimate a coordinate direction

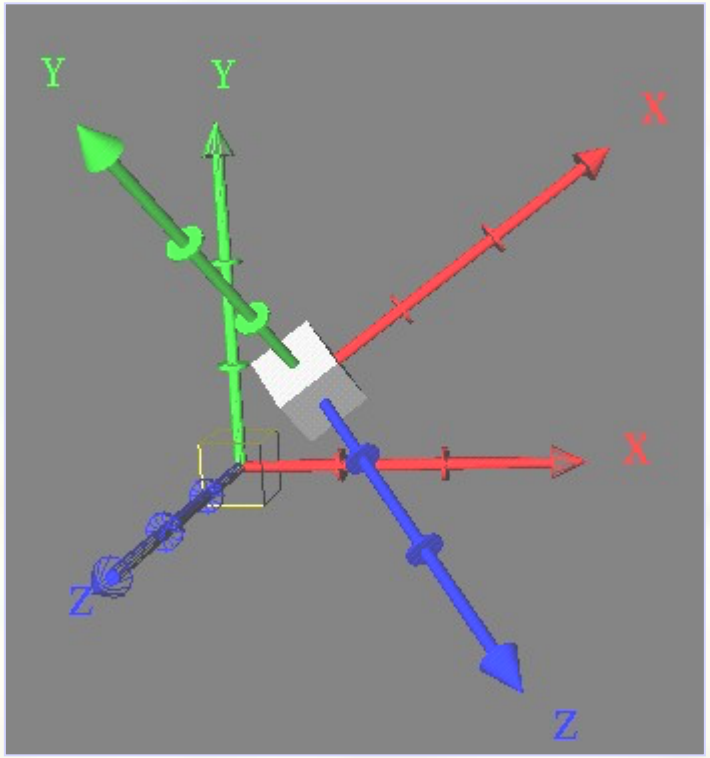
$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Note, there's a translation applied here to make things easier to see

Rotation

Rotate coordinate system about an axis in space



Note, there's a translation applied here to make things easier to see

Vertex Shader for Cube Rotation

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main() {
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians(theta);
    vec3 c = cos(angles);
    vec3 s = sin(angles);
```

Vertex Shader for Cube Rotation

```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Vertex Shader for Cube Rotation

```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx * vPosition;
}
```

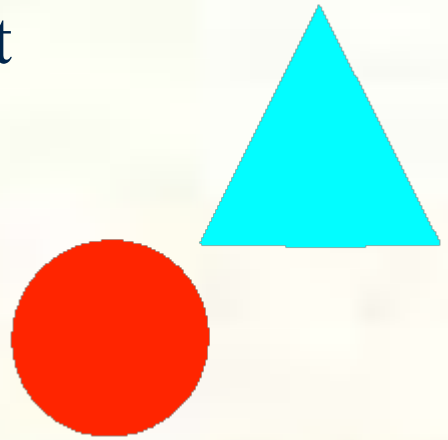
Sending Angles from Application

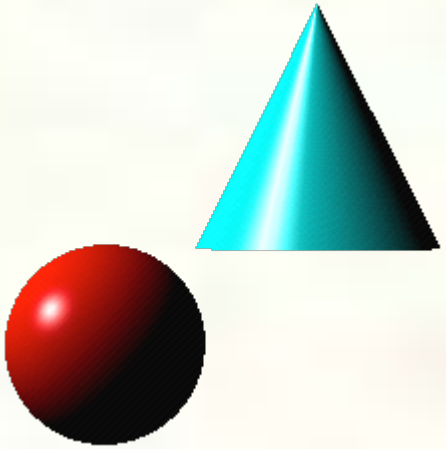
```
// compute angles using mouse and idle callbacks
GLuint theta;  // theta uniform location
vec3  Theta;   // Axis angles

void display(void) {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glUniform3fv(theta, 1, Theta);
   glDrawArrays(GL_TRIANGLES, 0, NumVertices);

   glutSwapBuffers();
}
```

Vertex Lighting

Lighting Principles

- Lighting simulates how objects reflect light
  - material composition of object
  - light's color and position
  - global lighting parameters
- Lighting functions deprecated in 3.1
- Can implement in
  - Application (per vertex)
  - Vertex or fragment shaders





Modified Phong Model

- Computes a color or shade for each vertex using a lighting model (the modified Phong model) that takes into account
  - Diffuse reflections
  - Specular reflections
  - Ambient light
  - Emission
- Vertex shades are interpolated across polygons by the rasterizer

Modified Phong Model

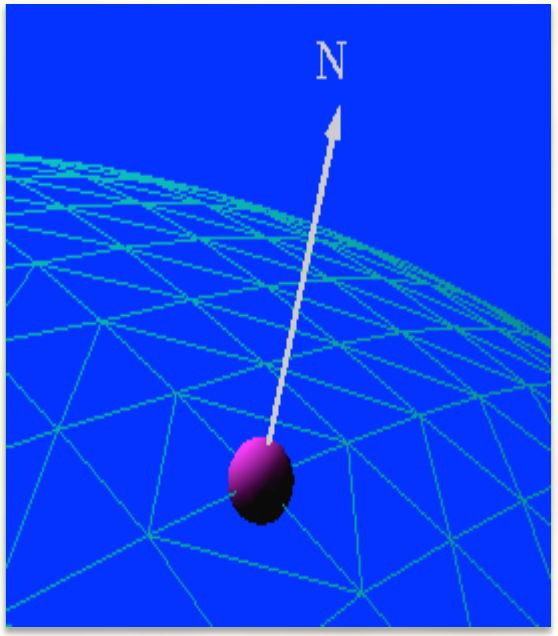
- The model is a balance between simple computation and physical realism
- The model uses
  - Light positions and intensities
  - Surface orientation (normals)
  - Material properties (reflectivity)
  - Viewer location
- Computed for each source and each color component

OpenGL Lighting

- Modified Phong lighting model
  - Computed at vertices
- Lighting contributors
  - Surface material properties
  - Light properties
  - Lighting model properties

Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex atttribute
  - Current normal is used to compute vertex's color
  - Use *unit* normals for proper lighting
    - scaling affects a normal's length



Material Properties

- Define the surface properties of a primitive

Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

  - you can have separate materials for front and back

Adding Lighting to Cube

```
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4 AmbientProduct, DiffuseProduct,
   SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Adding Lighting to Cube

```
void main() {
   // Transform vertex  position into eye coordinates
   vec3 pos = (ModelView * vPosition).xyz;

   vec3 L = normalize(LightPosition.xyz - pos);
   vec3 E = normalize(-pos);
   vec3 H = normalize(L + E);

   // Transform vertex normal into eye coordinates
   vec3 N = normalize(ModelView * vec4(vNormal, 0.0)).xyz;
```

Adding Lighting to Cube

```
// Compute terms in the illumination equation
    vec4 ambient = AmbientProduct;
    float Kd = max(dot(L, N), 0.0);
    vec4  diffuse = Kd*DiffuseProduct;
    float Ks = pow(max(dot(N, H), 0.0), Shininess);
    vec4  specular = Ks * SpecularProduct;
    if(dot(L, N) < 0.0)
        specular = vec4(0.0, 0.0, 0.0, 1.0)

    gl_Position = Projection * ModelView * vPosition;

    color = ambient + diffuse + specular;
    color.a = 1.0;
}
```

Shader Examples

Fragment Shaders

- A shader that's executed for each "potential" pixel
  - fragments still need to pass several tests before making it to the framebuffer
- There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Bump Mapping
  - Environment (Reflection) Maps

Per Fragment Lighting

- Compute lighting using same model as for per vertex lighting but for each fragment
- Normals and other attributes are sent to vertex shader and output to rasterizer
- Rasterizer interpolates and provides inputs for fragment shader

Shader Examples

- Vertex Shaders
  - Moving vertices: height fields
  - Per vertex lighting: height fields
  - Per vertex lighting: cartoon shading
- Fragment Shaders
  - Per vertex vs. per fragment lighting: cartoon shader
  - Samplers: reflection Map
  - Bump mapping

Height Fields

- A height field is a function $y = f(x, z)$ where the y value represents a quantity such as the height above a point in the x-z plane.
- Heights fields are usually rendered by sampling the function to form a rectangular mesh of triangles or rectangles from the samples $y_{ij} = f(x_i, z_j)$

Displaying a Height Field

- Form a quadrilateral mesh

```
for(i=0;i<N;i++) for(j=0;j<N;j++) data[i][j]=f(i, j, time);

vertex[Index++] = vec3((float)i/N, data[i][j], (float)j/N);
vertex[Index++] = vec3((float)i/N, data[i][j], (float)(j+1)/N);
vertex[Index++] = vec3((float)(i+1)/N, data[i][j], (float)(j+1)/N);
vertex[Index++] = vec3((float)(i+1)/N, data[i][j], (float)(j)/N);
```

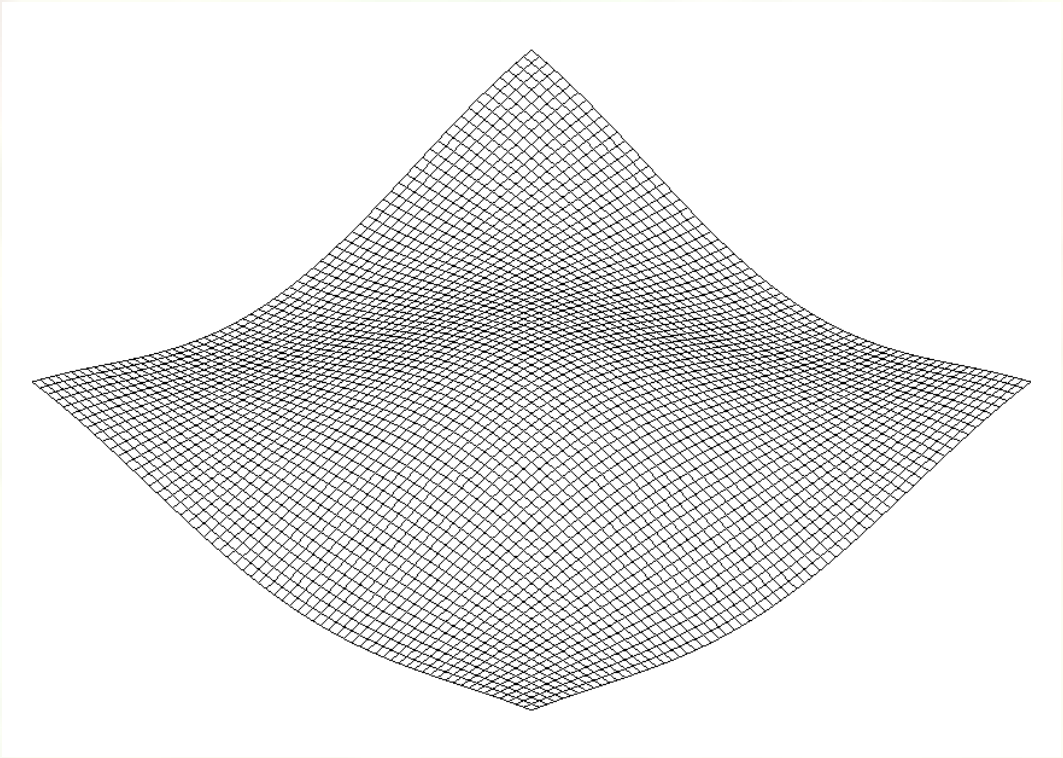
- Display each quad using

```
for(i=0;i<NumVertices ;i+=4) glDrawArrays(GL_LINE_LOOP, 4*i, 4);
```

Time Varying Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;

uniform float time; /* in milliseconds */
uniform mat4 ModelView, ProjectionMatrix;

void main()  {
    vec4  v = vPosition;
    vec4  t = sin(0.001*time + 5.0*v);
    v.y = 0.1*t.x*t.z;

    gl_Position = ModelViewProjectionMatrix * t;
}
```

Mesh Display



Adding Lighting

- Solid Mesh: create two triangles for each quad
- Display with

```
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```

- For better looking results, we'll add lighting
- We'll do per-vertex lighting
  - leverage the vertex shader since we'll also use it to vary the mesh in a time-varying way

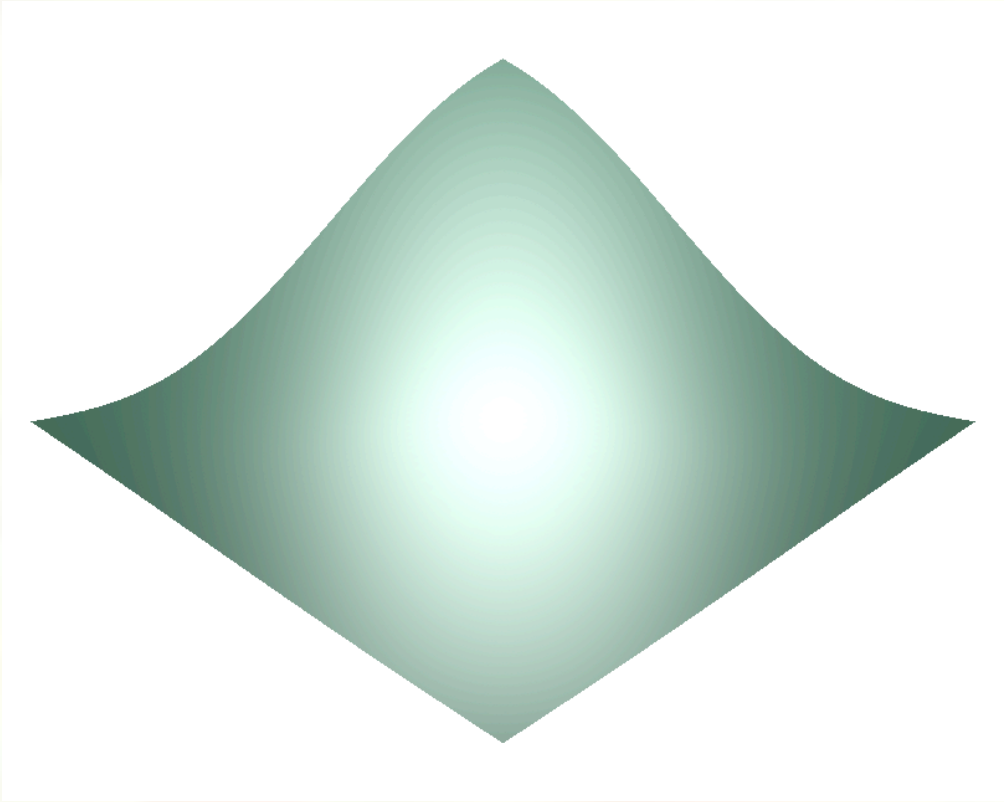
Mesh Shader

```
uniform float time, shininess;
uniform vec4 vPosition, light_position diffuse_light,
specular_light;
uniform mat4 ModelViewMatrix, ModelViewProjectionMatrix,
    NormalMatrix;

void main() {
   vec4  v = vPosition;
   vec4  t = sin(0.001*time + 5.0*v);
   v.y = 0.1*t.x*t.z;

   gl_Position = ModelViewProjectionMatrix * v;

   vec4 diffuse, specular;
   vec4 eyePosition = ModelViewMatrix * vPosition;
   vec4 eyeLightPos = light_position;
```

Mesh Shader (cont'd)

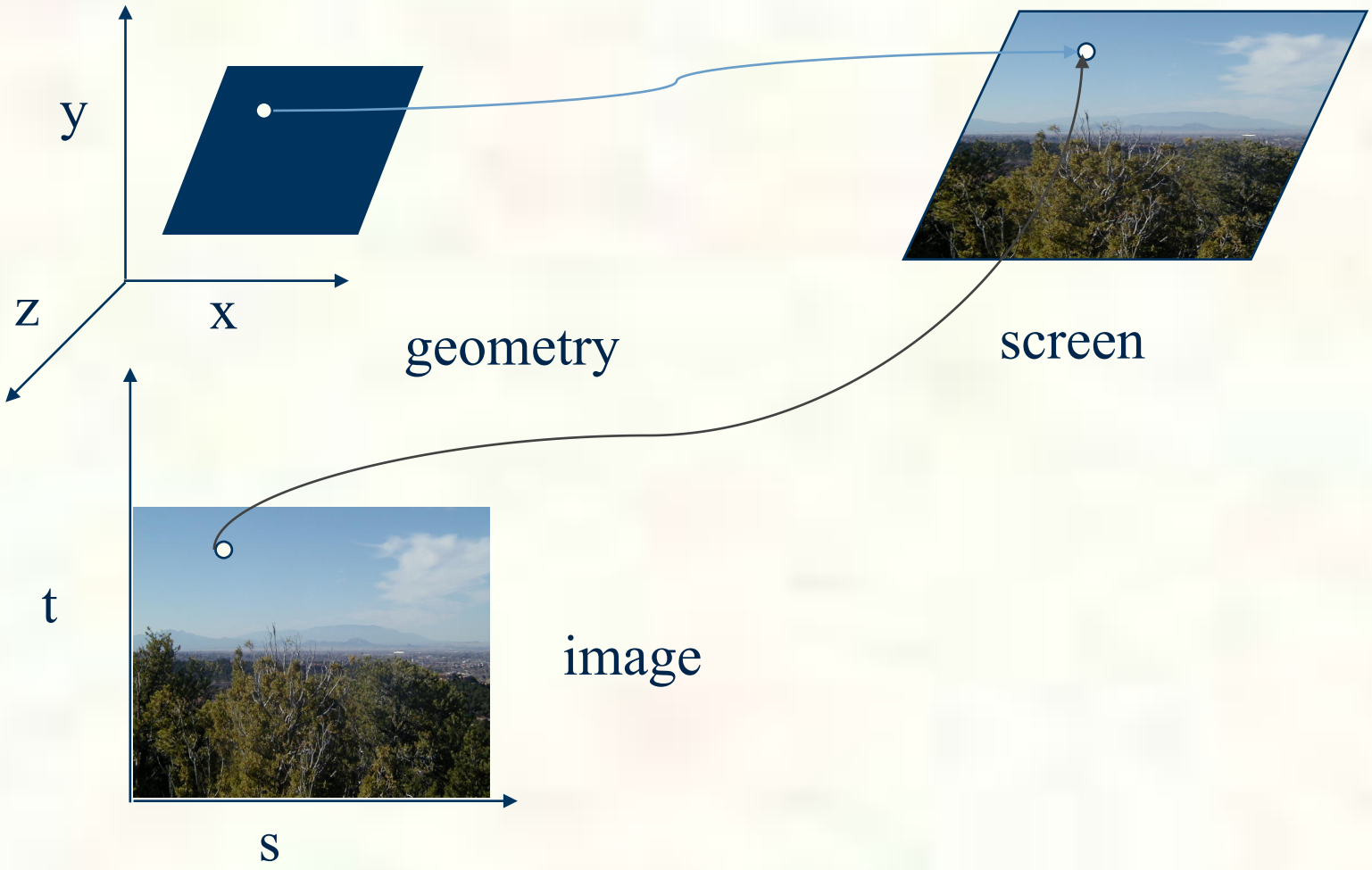
```
    vec3 N = normalize(NormalMatrix * Normal);
    vec3 L = normalize(eyeLightPos.xyz - eyePosition.xyz);
    vec3 E = -normalize(eyePosition.xyz);
    vec3 H = normalize(L + E);

    float Kd = max(dot(L, N), 0.0);
    float Ks = pow(max(dot(N, H), 0.0), shininess);
    diffuse  = Kd*diffuse_light;
    specular = Ks*specular_light;
    color    = diffuse + specular;
}
```

Shaded Mesh



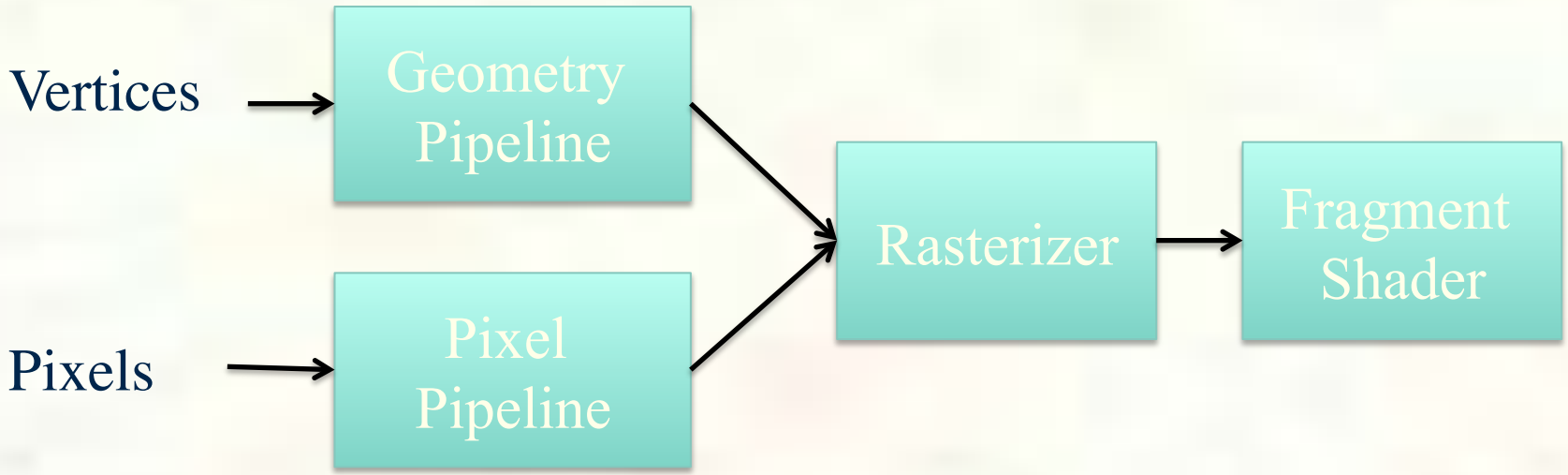
Texture Mapping

Texture Mapping



Texture Mapping in OpenGL

- Images and geometry flow through separate pipelines that join at the rasterizer
  - "complex" textures do not affect geometric complexity



Applying Textures

- Three basic steps to applying a texture
  1. specify the texture
     - read or generate image
     - assign to texture
     - enable texturing
  2. assign texture coordinates to vertices
  3. specify texture parameters
     - wrapping, filtering

Applying Textures

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex

Texture Objects

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts
- Generate texture names

```
glGenTextures(n, *texIds);
```

Texture Objects (cont'd.)

- Create texture objects with texture data and state
  - `glBindTexture(target, id);`
- Bind textures before using
  - `glBindTexture(target, id);`

Specifying a Texture Image

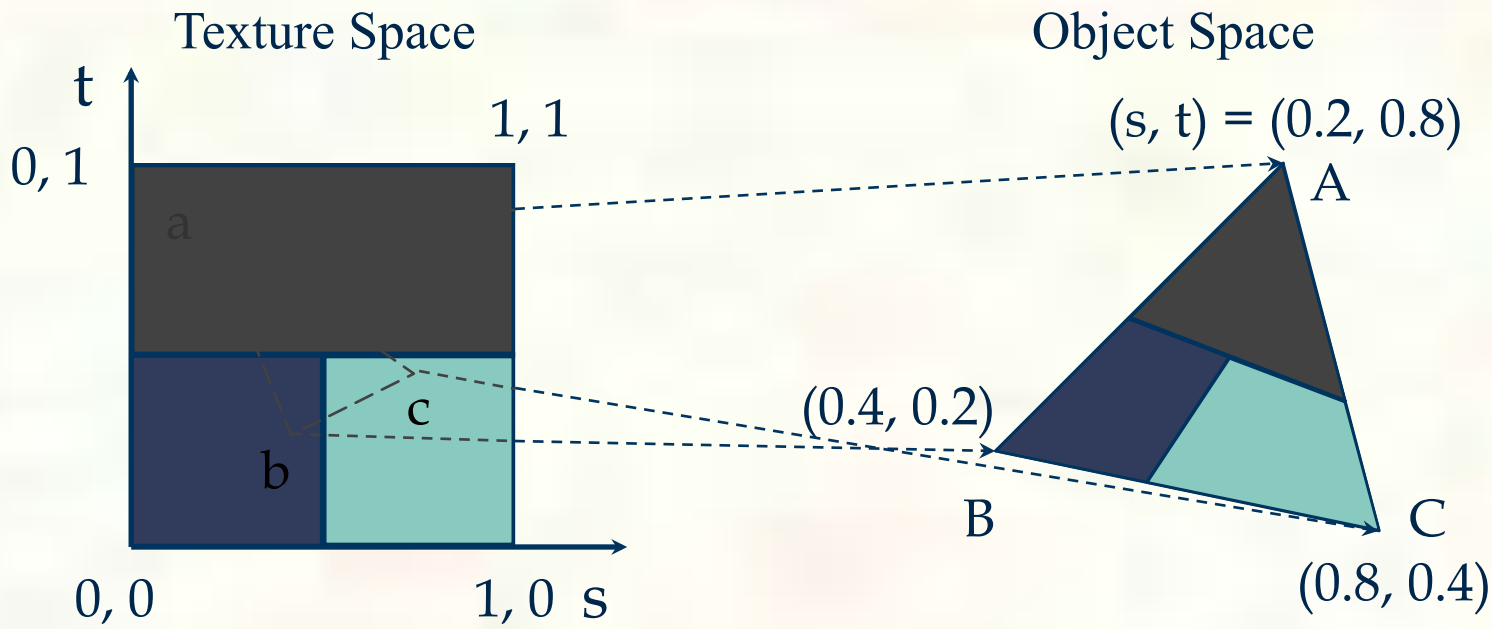
- Define a texture image from an array of texels in CPU memory

```
glTexImage2D(target, level, components,
    w, h, border, format, type, *texels);
```

- Texel colors are processed by pixel pipeline
  - pixel scales, biases and lookups can be done

Mapping a Texture

- Based on parametric texture coordinates
- Coordinates need to be specified at each vertex



Applying the Texture in the Shader

```
// Declare the sampler
uniform sampler2D diffuse_mat;
// GLSL 3.30 has overloaded texture();
// Apply the material color
vec3 diffuse = intensity *
    texture2D(diffuse_mat, coord).rgb;
```

Texturing the Cube

```
// add texture coordinate attribute to quad
function

quad(int a, int b, int c, int d) {
    quad_colors[Index] = vertex_colors[a];
    points[Index] = vertex_positions[a];
    tex_coords[Index] = vec2(0.0, 0.0);
    Index++;
    … // rest of vertices
}
```

Creating a Texture Image

```
// Create a checkerboard pattern
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        GLubyte c;
        c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) * 255;
        image[i][j][0]  = c;
        image[i][j][1]  = c;
        image[i][j][2]  = c;
        image2[i][j][0] = c;
        image2[i][j][1] = 0;
        image2[i][j][2] = c;
        }
    }
```

Texture Object

```
GLuint textures[1];
glGenTextures(1, textures);

glBindTexture(GL_TEXTURE_2D, textures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
             TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glActiveTexture(GL_TEXTURE0);
```

Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main() {
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```

Fragment Shader

```
in vec4 color;
in vec2 texCoord;
out vec4 FragColor;

uniform sampler texture;

void main() {
   FragColor = color * texture(texture, texCoord);
}
```

Next class: Visual Perception

- Topic:
  - How does the human visual system?
  - How do humans perceive color?
  - How do we represent color in computations?
- Read:
  - Glassner, Principles of Digital Image Synthesis, pp. 5-32. [Course reader pp.1-28]
  - Watt , Chapter 15.
  - Brian Wandell. Foundations of Vision. Sinauer Associates, Sunderland, MA, pp. 45-50 and 69-97, 1995. [Course reader pp. 29-34 and pp. 35-63]