An Introduction to OpenGL Programming

Ed Angel University of New Mexico

Dave Shreiner ARM, Inc.

SIGGRAPH2013

The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques

SIGGRAPH2013

What Is OpenGL?

- OpenGL is a computer graphics rendering *application programming interface,* or API (for short)
  - 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

SIGGRAPH2013

Course Ground Rules

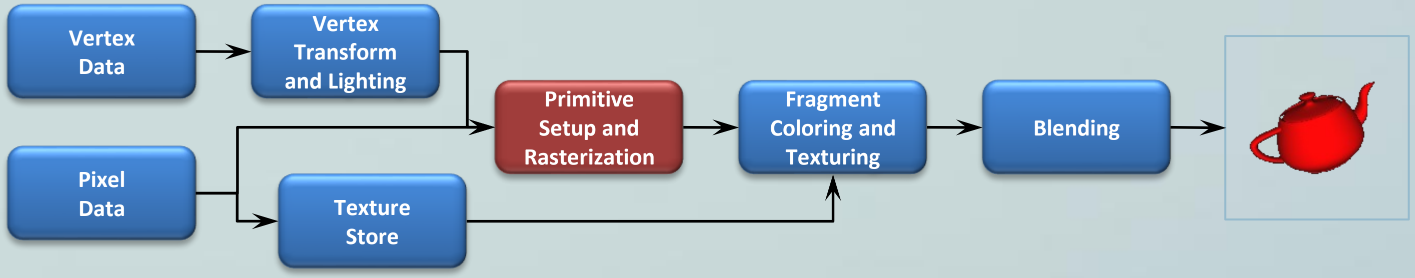
- We’ll concentrate on newer versions of OpenGL
- They enforce a new way to program with OpenGL
  - Allows more efficient use of GPU resources
- Modern OpenGL doesn’t support many of the “classic” ways of doing things, such as
  - Fixed-function graphics operations, like vertex lighting and transformations
- All applications must use *shaders* for their graphics processing
  - we only introduce a subset of OpenGL’s shader capabilities in this course

Evolution of the OpenGL Pipeline

SIGGRAPH2013
The 40th International Conference and Exhibition
on Computer Graphics and Interactive Techniques

In the Beginning …

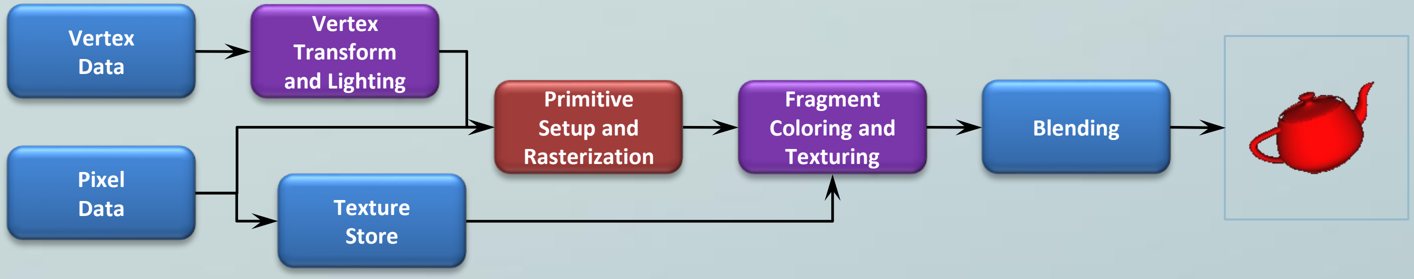
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation


Vertex Data
Vertex Transform and Lighting
Primitive Setup and Rasterization
Fragment Coloring and Texturing
Blending
Pixel Data
Texture Store

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

Beginnings 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


Vertex Data
Vertex Transform and Lighting
Pixel Data
Primitive Setup and Rasterization
Texture Store
Fragment Coloring and Texturing
Blending

SIGGRAPH2013

An Evolutionary Change

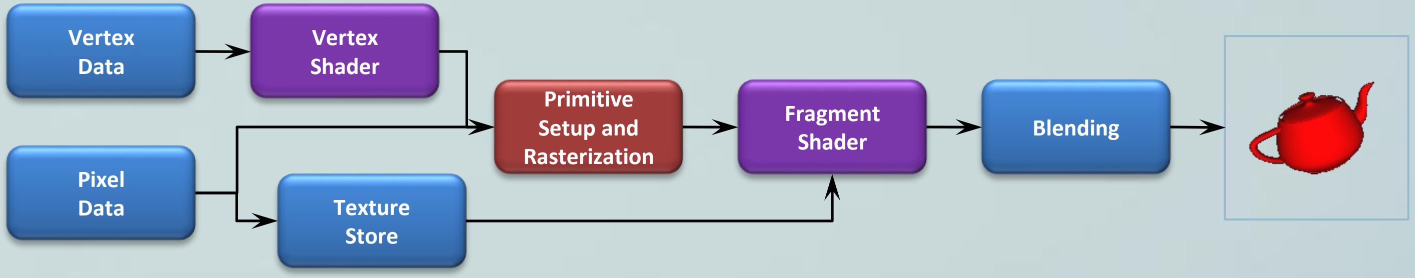
- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

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)

SIGGRAPH2013

The Exclusively Programmable Pipeline

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

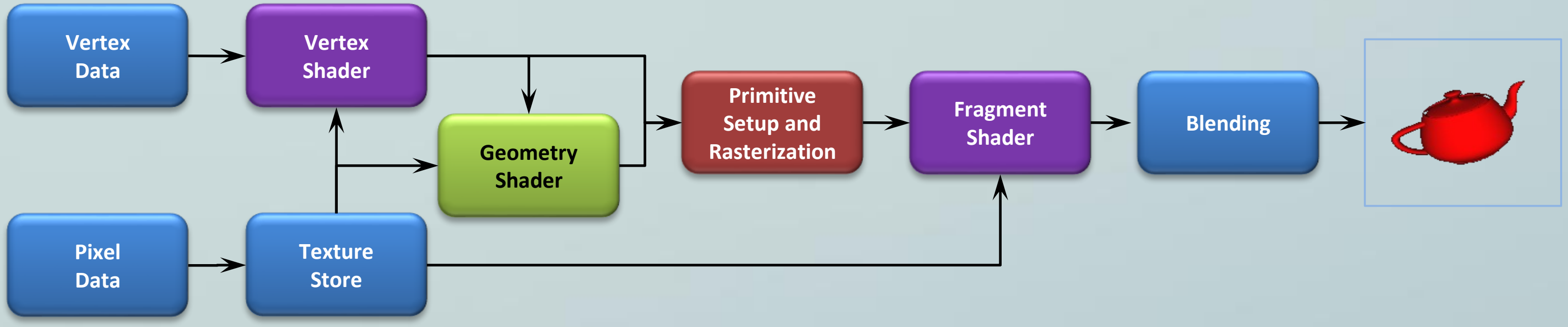


- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

SIGGRAPH2013

More Programmability

- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders
  - modify geometric primitives within the graphics pipeline



SIGGRAPH2013

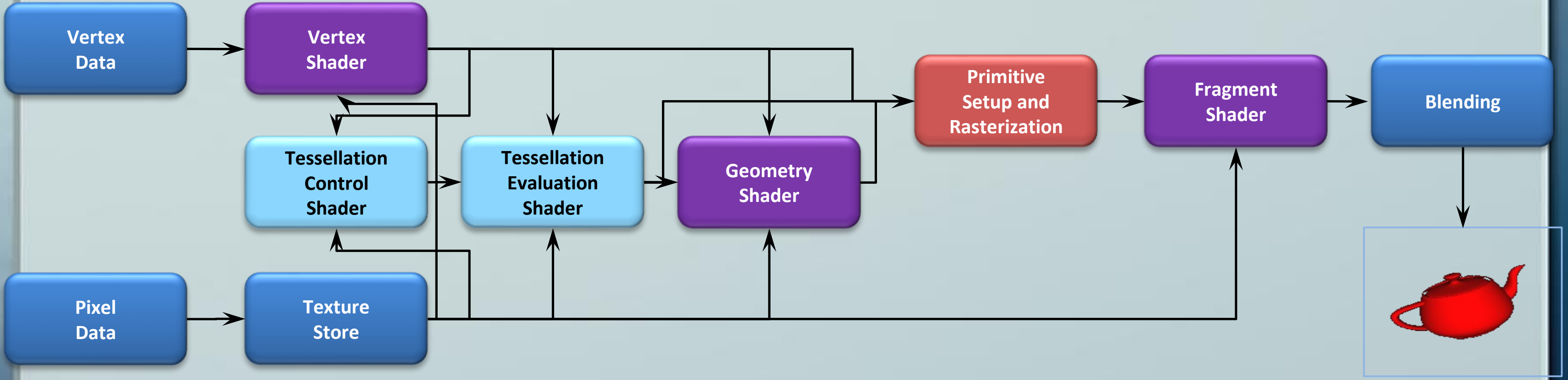
More Evolution – Context Profiles

- OpenGL 3.2 also introduced *context profiles*
  - profiles control which features are exposed
    - it's like `GL_ARB_compatibility`, only not insane ☺
  - 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 4.6



Vertex Data
Vertex Shader
Primitive Setup and Rasterization
Fragment Shader
Blending
Tessellation Control Shader
Tessellation Evaluation Shader
Geometry Shader
Pixel Data
Texture Store

OpenGL ES and WebGL

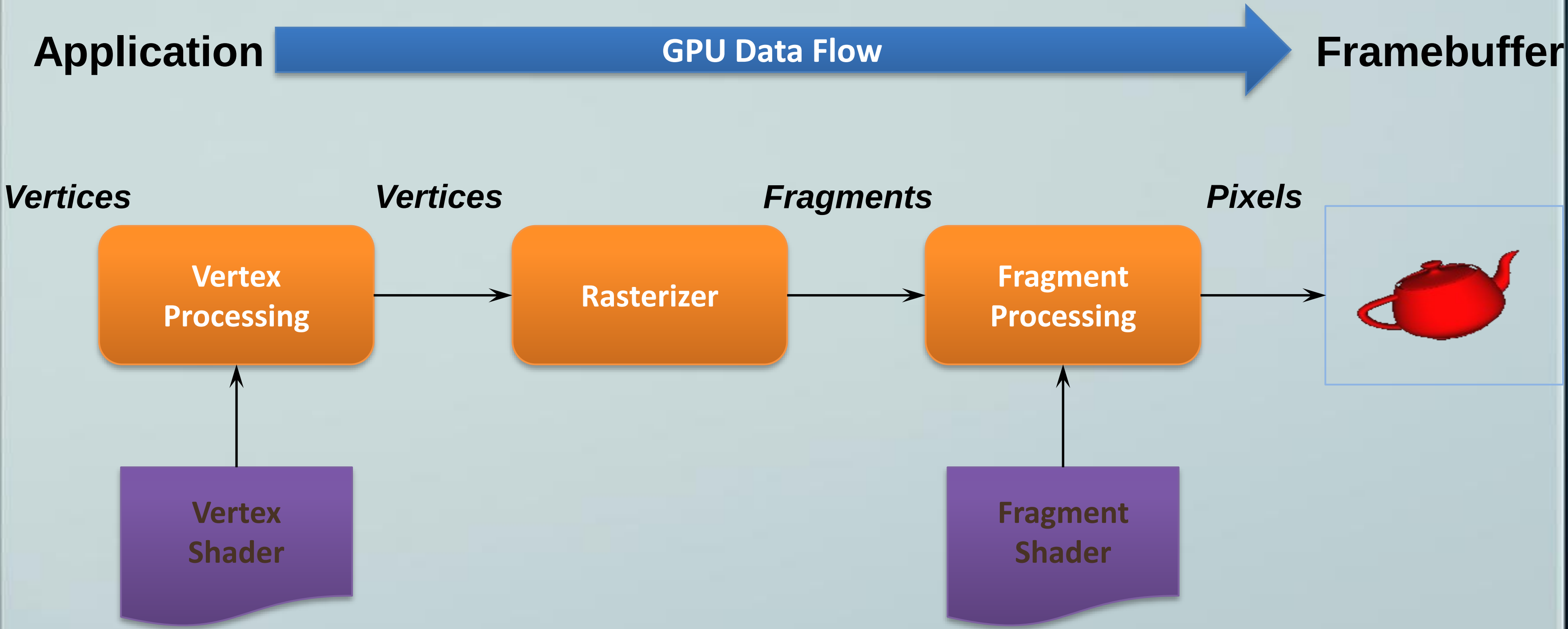
- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

OpenGL Application Development

SIGGRAPH 2013
The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques

SIGGRAPH2013

A Simplified Pipeline Model



OpenGL Programming in a Nutshell

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

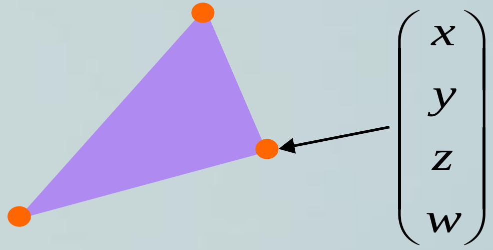
SIGGRAPH2013

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
- On Android, we have built-in functions that can help create a render target and a window.
  - GLSurfaceView

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
  - Allows us to express all vertex transformations using matrix-vector multiplies
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array objects (VAOs)



$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Vertex Array Objects (VAOs)

- VAOs store the data of an 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()`

Connecting Vertex Shaders with Geometric Data

- 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()`

Drawing Geometric Primitives

- For contiguous groups of vertices

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

- Usually invoked in display callback
- Initiates vertex shader

Shaders and GLSL

SIGGRAPH2013
The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques

SIGGRAPH2013

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);
```

SIGGRAPH2013

Operators

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

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

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

Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```

 - all refer to the same element

- Component swizzling:

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

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

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

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Functions

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

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Simple Vertex Shader for Cube Example

```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

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

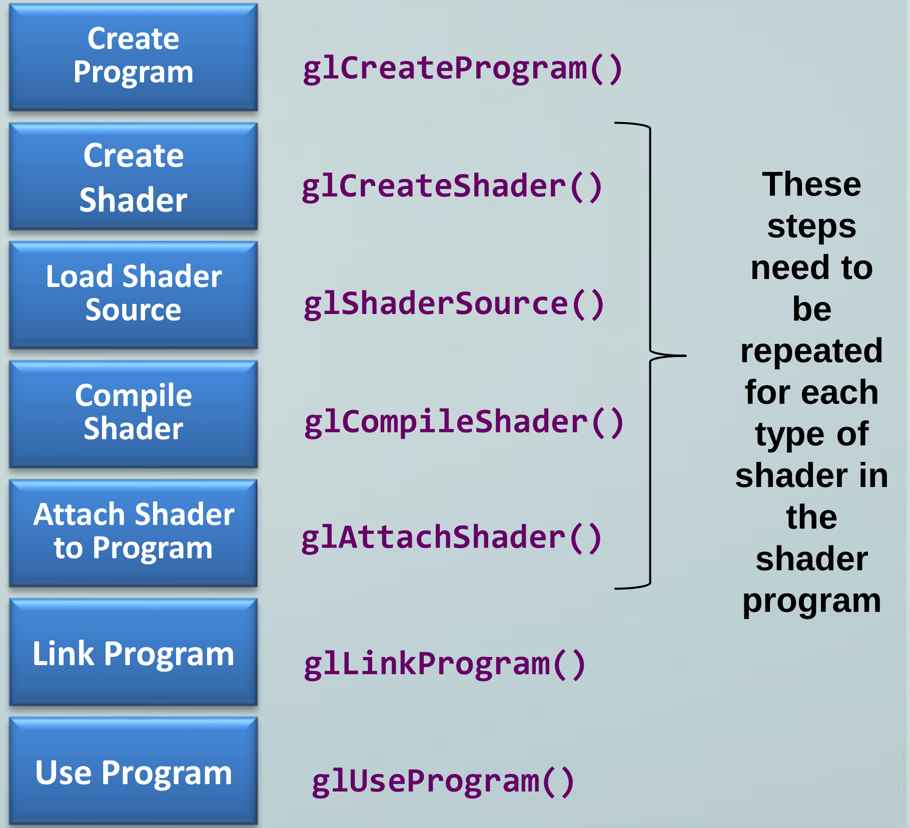
The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

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

SIGGRAPH2013

Getting Your 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



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

SIGGRAPH2013

Determining Locations After Linking

- Assumes you already know the variables’ names

```
GLint loc = glGetAttribLocation( program, “name” );

GLint loc = glGetUniformLocation( program, “name” );
```

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

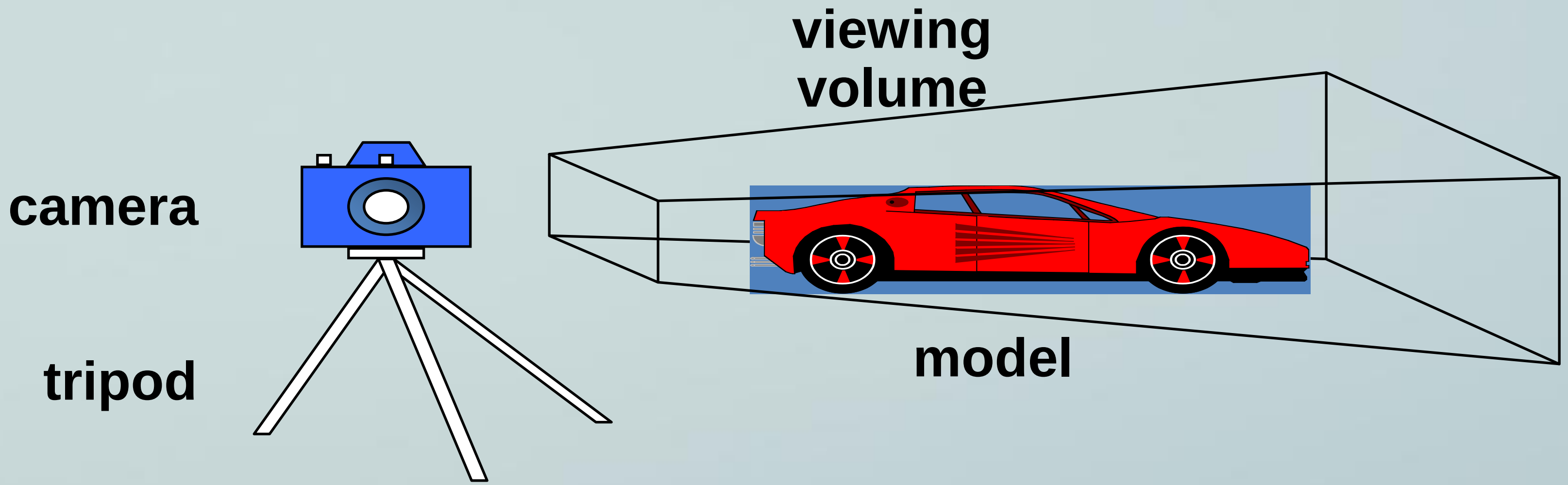
Transformations

SIGGRAPH2013

The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques

Camera Analogy

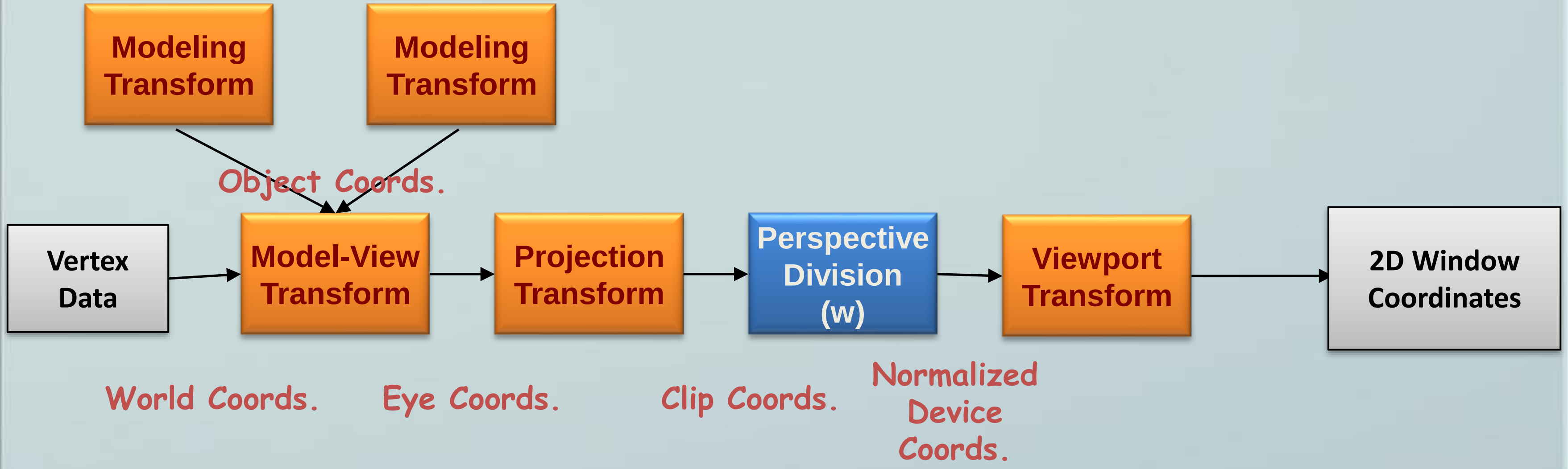
- 3D is just like taking a photograph (lots of photographs!)



SIGGRAPH2013

Transformations

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



Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph

SIGGRAPH2013

3D 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
- Product of matrix and vector is $\mathbf{M}\vec{v}$

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

Specifying What You Can See

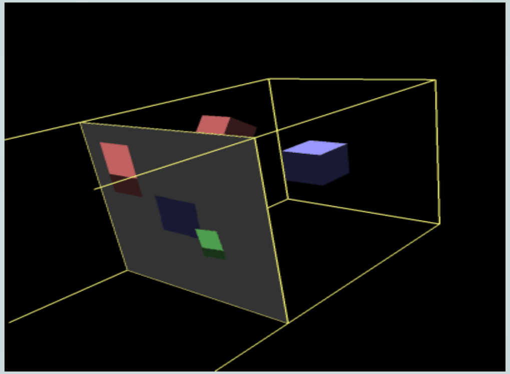
- Set up a viewing frustum to specify how much of the world we can see
- Done in two steps
  - specify the size of the frustum (projection transform)
  - specify its location in space (model-view transform)
- Anything outside of the viewing frustum is clipped
  - primitive is either modified or discarded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

Specifying What You Can See (cont’d)

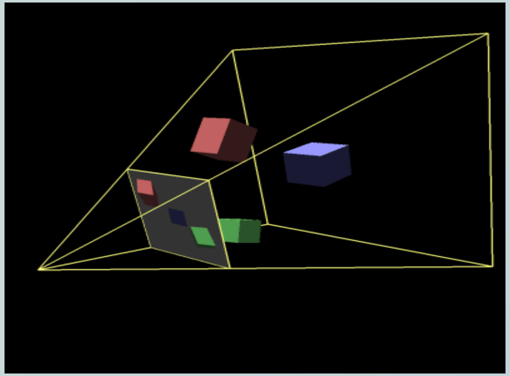
- OpenGL projection model uses eye coordinates
  - the “eye” is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom,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

Specifying What You Can See (cont'd)

Orthographic View



Perspective View



$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene

-
  ```
  LookAt( eyex, eyey, eyez,
          lookx, looky, lookz,
          upx, upy, upz )
  ```
  - up vector determines unique orientation
  - careful of degenerate positions

Creating the LookAt Matrix

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|}$$

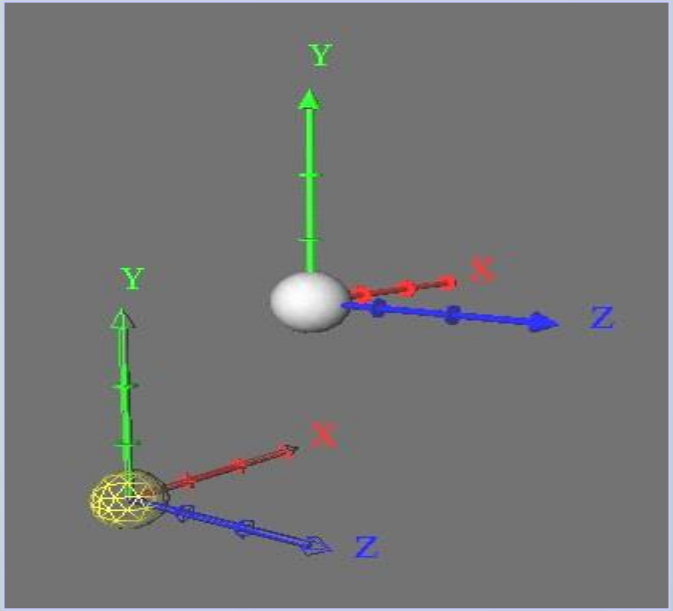
$$\hat{v} = \hat{u} \times \hat{n}$$

$$\Rightarrow \begin{pmatrix} u_x & u_y & u_z & -(eye \cdot \vec{u}) \\ v_x & v_y & v_z & -(eye \cdot \vec{v}) \\ -n_x & -n_y & -n_z & -(eye \cdot \vec{n}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Move the origin to a new location

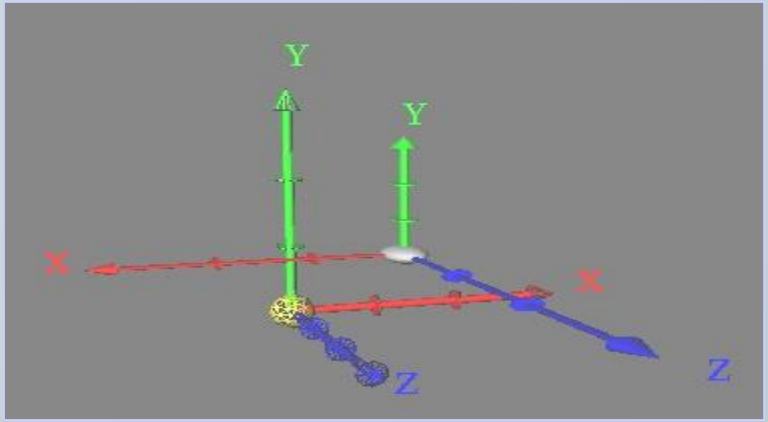
$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Scale

- Stretch, mirror or decimate a coordinate direction

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

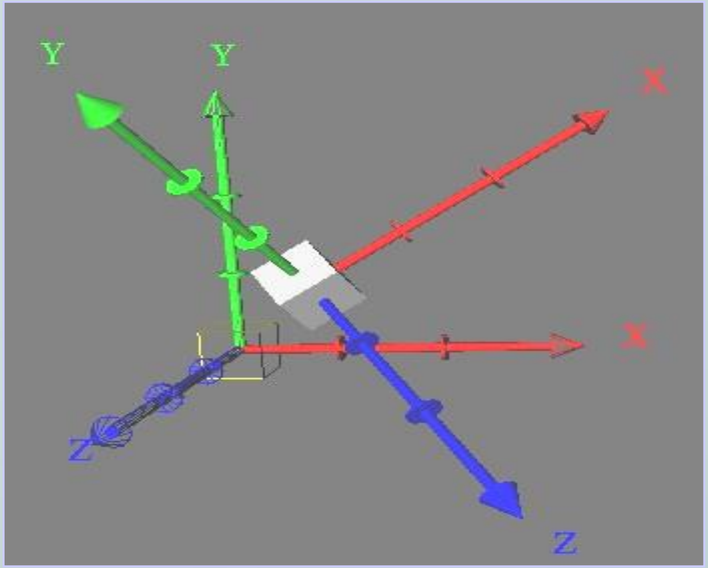


Note, there's a translation applied here to make things easier to see

SIGGRAPH2013

Rotation

- Rotate coordinate system about an axis in space



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

Rotation (cont'd)

$$\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}$$

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = \begin{pmatrix} x' & y' & z' \end{pmatrix}$$

$$M = \vec{u}^t\vec{u} + \cos(\theta)(I - \vec{u}^t\vec{u}) + \sin(\theta)S$$

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_{\vec{v}}(\theta) = \begin{pmatrix} & \boldsymbol{M} & & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

SIGGRAPH2013

Vertex Shader for Rotation of Cube

```
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 );
```

SIGGRAPH2013

Vertex Shader for Rotation of Cube (cont'd)

```
// 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 Rotation of Cube (cont'd)

```
    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

- Here, we compute our angles (Theta) in our mouse callback

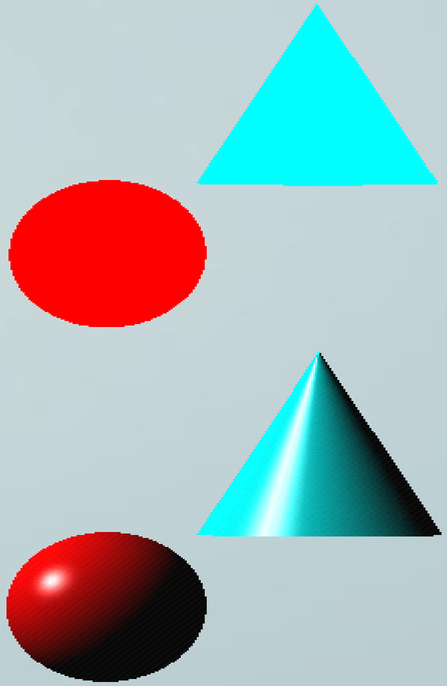
```
GLuint theta;  // theta uniform location
vec3  Theta;   // Axis angles

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

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

   glutSwapBuffers();
}
```

Lighting

SIGGRAPH2013

Lighting Principles

- Lighting simulates how objects reflect light
  - material composition of object
  - light’s color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
  - fragment shader for nicer shading

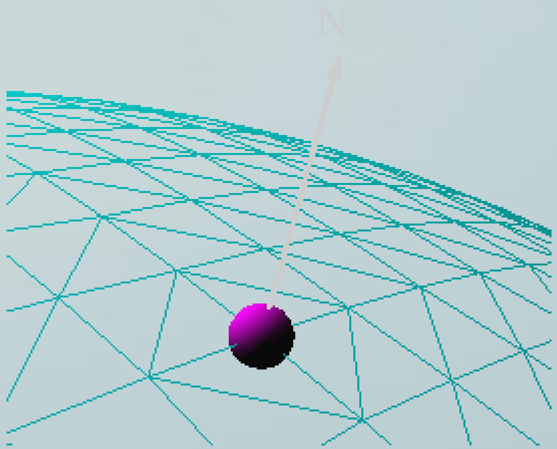


Modified Phong Model

- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer’s position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    - more accurate results

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

SIGGRAPH2013

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 (cont'd)

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

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

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(vec3(ModelView * vNormal));
```

SIGGRAPH2013

Adding Lighting to Cube (cont'd)

```
    // 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;
}
```

Fragment Shaders

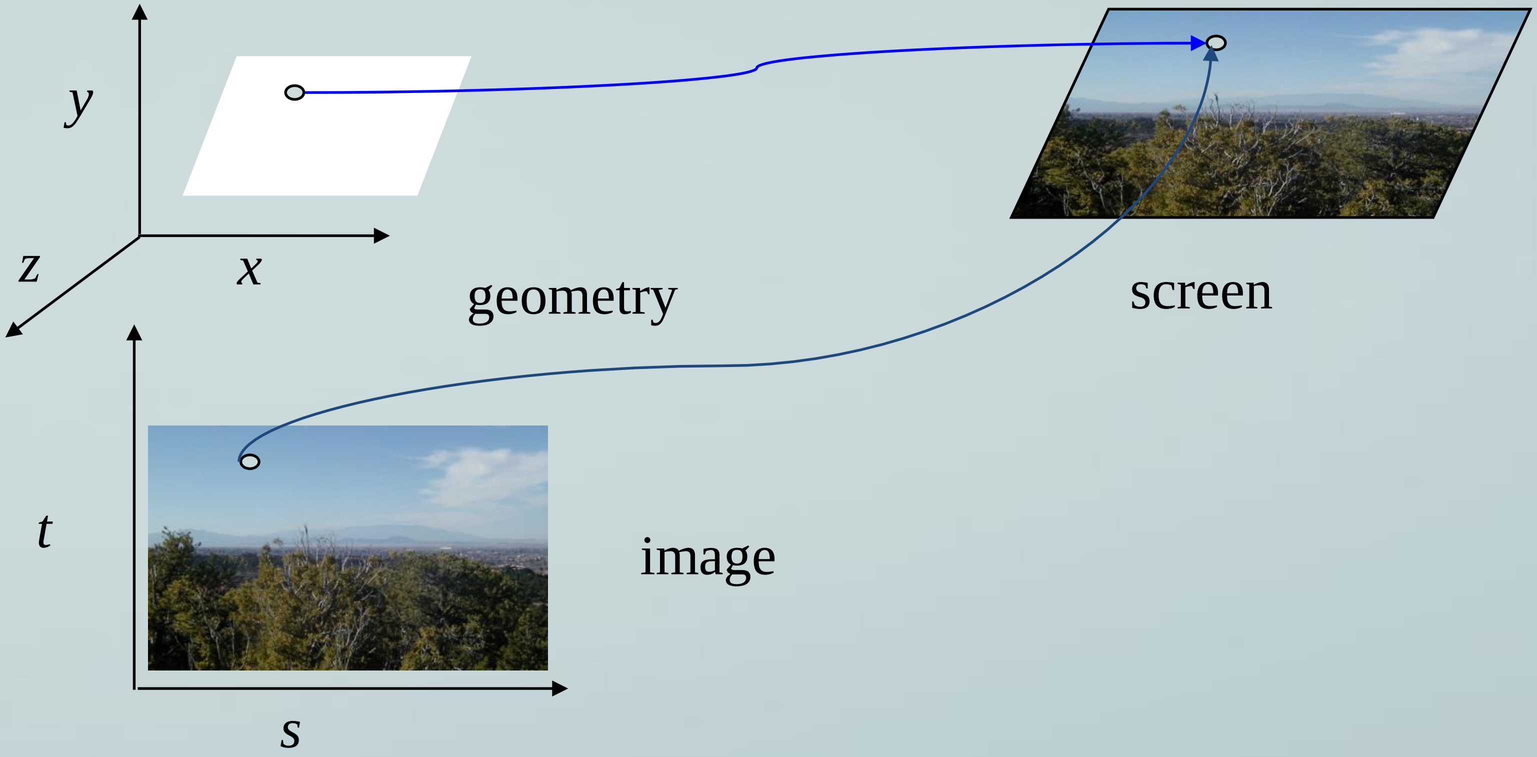
SIGGRAPH2013
The 40th International Conference and Exhibition on Computer Graphics and Interactive Techniques

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
  - Texture and bump Mapping
  - Environment (Reflection) Maps

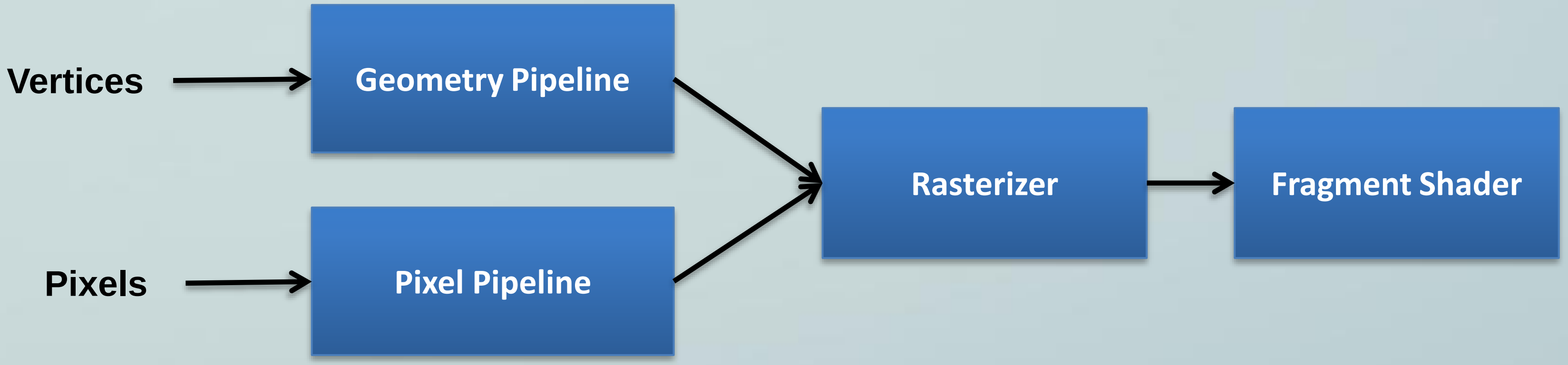
Texture Mapping

Texture Mapping



Texture Mapping and the OpenGL Pipeline

- 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

SIGGRAPH2013

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 );
```

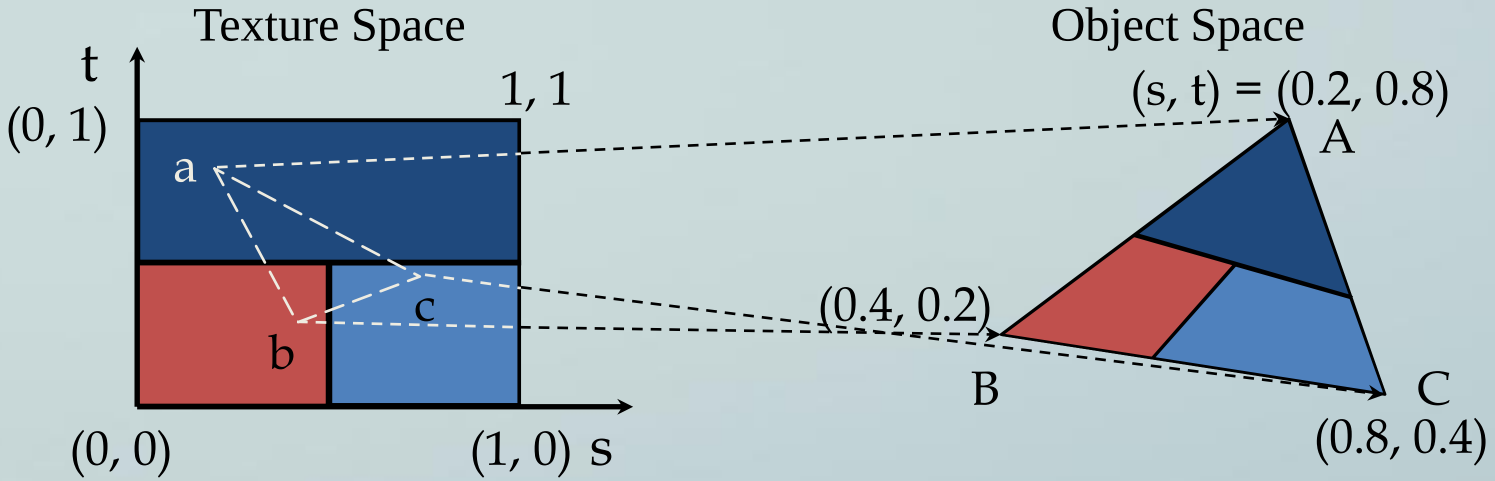
Specifying a Texture Image

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

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

Mapping a Texture

- Based on parametric texture coordinates
- coordinates needs to be specified at each vertex
- So goes into vertex shader



Texture Space
Object Space
t
(0, 1)
1, 1
a
b
c
(0, 0)
(1, 0) s
(s, t) = (0.2, 0.8)
A
(0.4, 0.2)
B
C
(0.8, 0.4)