Lecture 3

Programming
with
OpenGL 3.1 + GLUT + GLEW

OpenGL

The success of GL lead to OpenGL (1992), a platform-independent API that was

- Easy to use
- Close enough to the hardware to get excellent performance
- Focus on rendering
- Omitted windowing and input to avoid window system dependencies

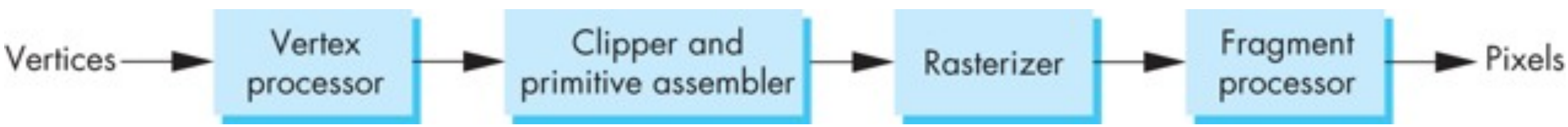
Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

OpenGL evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM,…….
  - Now Kronos Group
  - Was relatively stable (through version 2.5)
    - Backward compatible
    - Evolution reflected new hardware capabilities
      – 3D texture mapping and texture objects
      – Vertex and fragment programs
  - Allows platform specific features through extensions

Modern OpenGL

- Performance is achieved by using GPU rather than CPU
- Control GPU through programs called shaders
- Application's job is to send data to GPU
- GPU does all rendering



OpenGL 3.1

- Totally shader-based
  - No default shaders
  - Each application must provide both a vertex and a fragment shader
- No immediate mode
- Few state variables
- Most 2.5 functions deprecated
- Backward compatibility not required

OpenGL: Other Versions

- OpenGL ES
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- OpenGL 4.1 and 4.2
  - Add geometry shaders and tessellator

What about Direct X ?

- Windows only
- Advantages
  - Better control of resources
  - Access to high level functionality
- Disadvantages
  - New versions not backward compatible
  - Windows only
- Recent advances in shaders are leading to convergence with OpenGL

OpenGL Libraries

- OpenGL core library
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- OpenGL Utility Library (GLU)
  - Provides functionality in OpenGL core but avoids having to rewrite code
- Links with window system
  - GLX for X window systems
  - WGL for Windows
  - AGL for Macintosh

GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
    - Event-driven
  - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

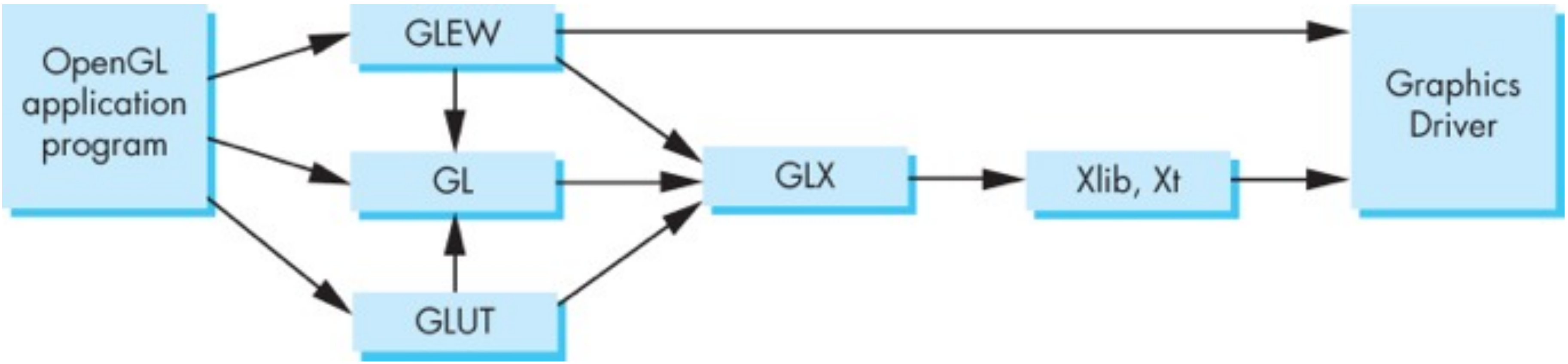
freeglut

- GLUT was created long ago and has been unchanged
  - Amazing that it works with OpenGL 3.1
  - Some functionality can't work since it requires deprecated functions
- freeglut updates GLUT
  - Added capabilities
  - Context checking

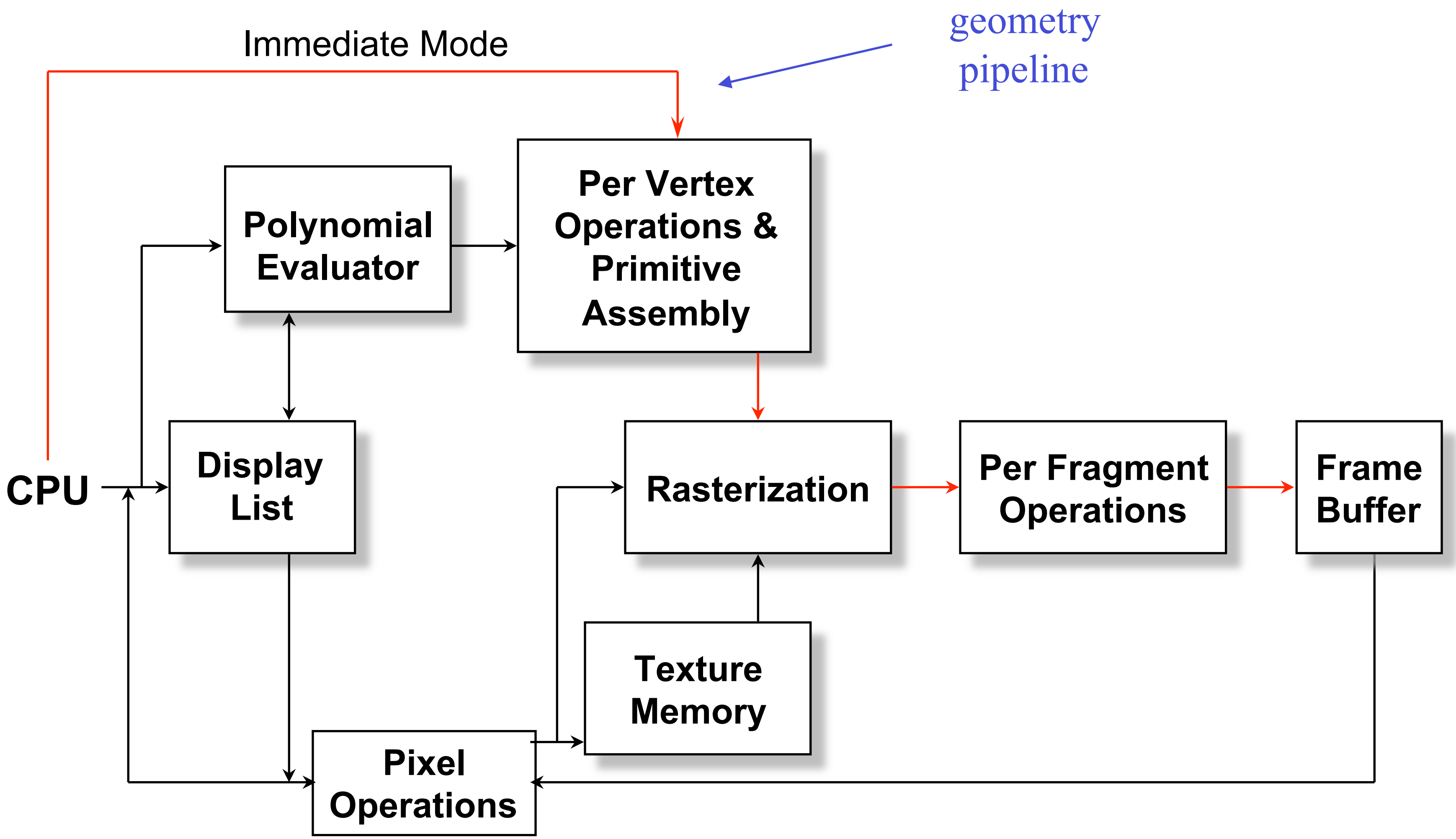
GLEW

- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- Avoids having to have specific entry points in Windows code
- Application needs only to include glew.h and run a glewInit()

Software Organization



OpenGL Architecture



Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Polygons
- Attributes
- Transformations
  - Viewing
  - Modeling
- Control (GLUT)
- Input (GLUT)
- Query

OpenGL State

- OpenGL is a state machine
- OpenGL functions are of two types
  - Primitive generating
    - Can cause output if primitive is visible
    - How vertices are processed and appearance of primitive are controlled by the state
  - State changing
    - Transformation functions
    - Attribute functions
    - Under 3.1 most state variables are defined by the application and sent to the shaders

Not Object Oriented

- OpenGL is not object oriented so that there are multiple functions for a given logical function
  - `glUniform3f`
  - `glUniform2i`
  - `glUniform3dv`
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

OpenGL Function Format

function name

dimensions

`glUniform3f(x,y,z)`

belongs to GL library

`x,y,z` are `floats`

`glUniform3fv(p)`

`p` is a pointer to an array

OpenGL #defines

- Most constants are defined in the include files **`gl.h`**, **`glu.h`** and **`glut.h`**
  - Note **`#include <GL/glut.h>`** should automatically include the others
  - Examples
  - **`glEnable(GL_DEPTH_TEST)`**
  - **`glClear(GL_COLOR_BUFFER_BIT)`**
- include files also define OpenGL data types: **`GLfloat`**, **`GLdouble`**,....

OpenGL and GLSL

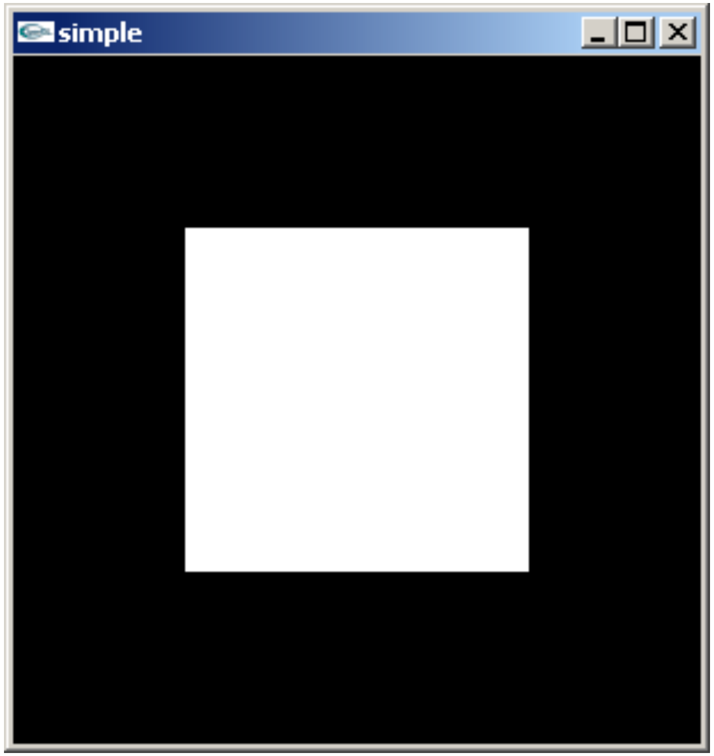
- Shader based OpenGL is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job is application is to get data to GPU

GLSL

- OpenGL Shading Language
- C-like with
  - Matrix and vector types (2, 3, 4 dimensional)
  - Overloaded operators
  - C++ like constructors
- Similar to Nvidia's Cg and Microsoft HLSL
- Code sent to shaders as source code
- New OpenGL functions to compile, link and get information to shaders

A Simple Program (?)

Generate a square on a solid background



It used to be easy (Simple.c)

```
#include <GL/glut.h>
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_POLYGON);
            glVertex2f(-0.5, -0.5);
            glVertex2f(-0.5, 0.5);
            glVertex2f(0.5, 0.5);
            glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
}
int main(int argc, char** argv){
        glutCreateWindow("simple");
        glutDisplayFunc(mydisplay);
        glutMainLoop();
}
```

What Happened ?

- Most OpenGL functions deprecated
- Makes heavy use of state variable default values that no longer exist
  - Viewing
  - Colors
  - Window parameters
- Next version will make the defaults more explicit
- However, processing loop is the same

Simple.c

```
#include <GL/glut.h>
void mydisplay(){
glClear(GL_COLOR_BUFFER_BIT);

// need to fill in this part
// and add in shaders
}
int main(int argc, char** argv){
glutCreateWindow("simple");
glutDisplayFunc(mydisplay);
glutMainLoop();
}
```

Event Loop

- Note that the program defines a *display callback* function named **`mydisplay`**
  - Every glut program must have a display callback
  - The display callback is executed whenever OpenGL decides the display must be refreshed, for example when the window is opened
  - The **`main`** function ends with the program entering an event loop

Compilation Notes

- See website and starter code of Project 1 for example
- Unix/linux
  - Include files usually in …/include/GL
  - Compile with –lglut –lglu –lgl loader flags
  - May have to add –L flag for X libraries
  - Mesa implementation included with most linux distributions
  - Check web for latest versions of Mesa and GLUT

OpenGL Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - `main()`:
    - specifies the callback functions
    - opens one or more windows with the required properties
    - enters event loop (last executable statement)
  - `init()`: sets the state variables
    - Viewing
    - Attributes
  - `initShader()`: read, compile and link shaders
  - callbacks
    - Display function
    - Input and window functions

Simple.c (revisited)

- `main()` function is the same
  - Mostly GLUT functions
- init() will allow more flexible colors
- initShader() will hides details of setting up shaders for now
- Key issue is that we must form a data array to send to GPU and then render it

main.c

```c
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("simple");
 glutDisplayFunc(mydisplay);
 glewInit();
 init();
 glutMainLoop();
}
```

includes `gl.h`

specify window properties

display callback

set OpenGL state and initialize shaders

enter event loop

GLUT functions

- **`glutInit`** allows application to get command line arguments and initializes system
- **`gluInitDisplayMode`** requests properties for the window (the *rendering context*)
  - RGB color
  - Single buffering
  - Properties logically ORed together
- **`glutWindowSize`** in pixels
- **`glutWindowPosition`** from top-left corner of display
- **`glutCreateWindow`** create window with title "simple"
- **`glutDisplayFunc`** display callback
- **`glutMainLoop`** enter infinite event loop

Immediate Mode Graphics

- Geometry specified by vertices
  - Locations in space(2 or 3 dimensional)
  - Points, lines, circles, polygons, curves, surfaces
- Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU
  - Old style uses **`glVertex`**
  - Creates bottleneck between CPU and GPU
  - Removed from OpenGL 3.1

Retained Mode Graphics

- Put all vertex and attribute data in array
- Send array to GPU to be rendered immediately
- Almost OK but problem is we would have to send array over each time we need another render of it
- Better to send array over and store on GPU for multiple renderings

Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012*

Display Callback

- Once we get data to GLU, we can initiate the rendering with a simple callback

```
void mydisplay()
{
glClear(GL_COLOR_BUFFER_BIT);
glDrawArrays(GL_TRIANGLES, 0, 3);
glFlush();
}
```

- Arrays are buffer objects that contain vertex arrays

Vertex Arrays

- Vertices can have many attributes
  - Position
  - Color
  - Texture Coordinates
  - Application data
- A vertex array holds these data
- Using types in **`vec.h`**

```
point2 vertices[3] = {point2(0.0, 0.0),
    point2( 0.0, 1.0), point2(1.0, 1.0)};
```

Vertex Array Object

- Bundles all vertex data (positions, colors, ..,)
- Get name for buffer then bind

```
Glunit abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

- At this point we have a current vertex array but no contents
- Use of glBindVertexArray lets us switch between VBOs

Buffer Object

- Buffers objects allow us to transfer large amounts of data to the GPU
- Need to create, bind and identify data

```
Gluint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER,
sizeof(points), points);
```

- Data in current vertex array is sent to GPU

Initialization

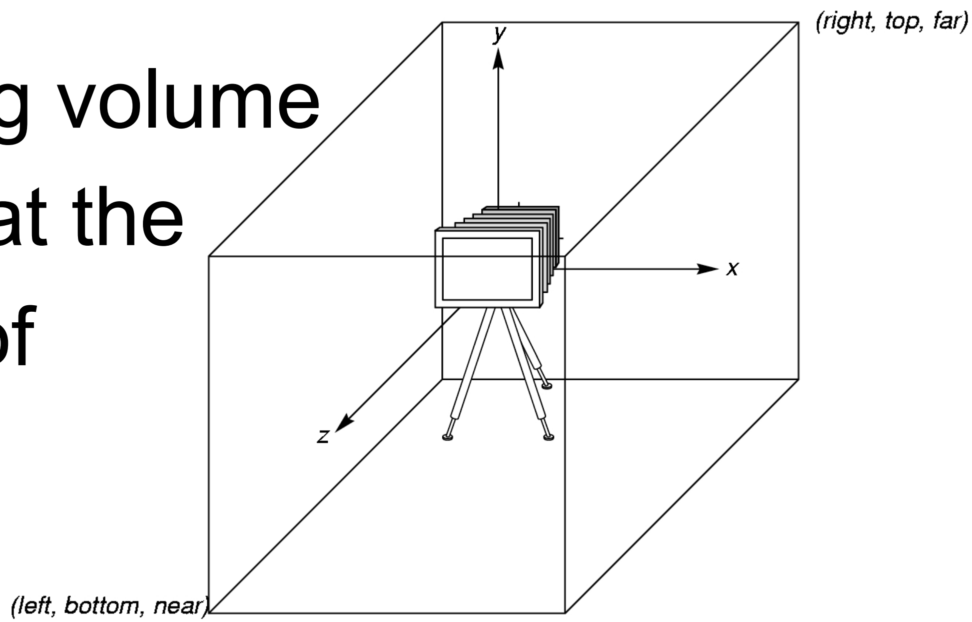
- Vertex array objects and buffer objects can be set up on **init()**
- Also set clear color and other OpeGL parameters
- Also set up shaders as part of initialization
  - Read
  - Compile
  - Link
- First let's consider a few other issues

Coordinate Systems in OpenGL

- The units in `points` are determined by the application and are called *object, world, model* or *problem coordinates*
- Viewing specifications usually are also in object coordinates
- Eventually pixels will be produced in *window coordinates*
- OpenGL also uses some internal representations that usually are not visible to the application but are important in the shaders

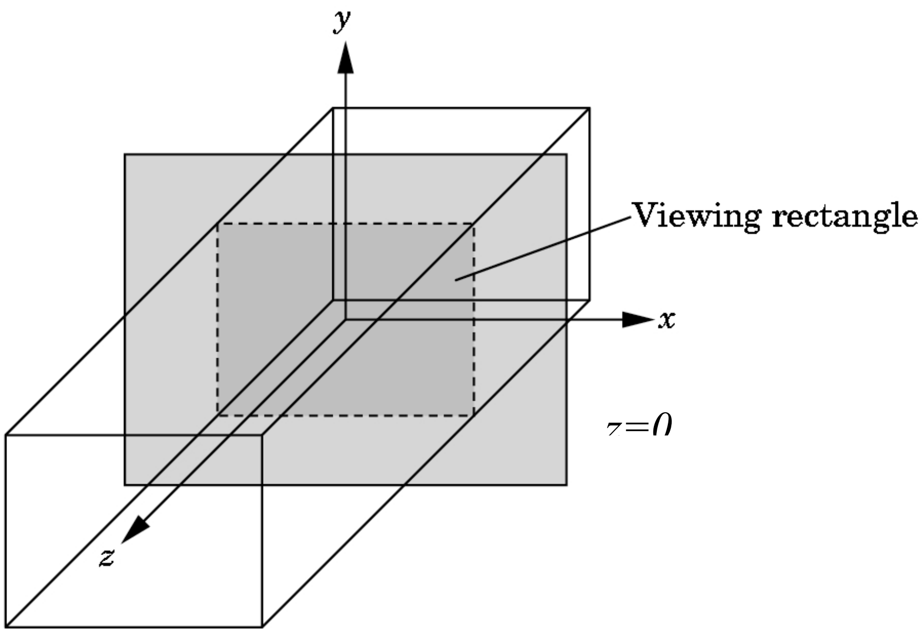
OpenGL Camera I

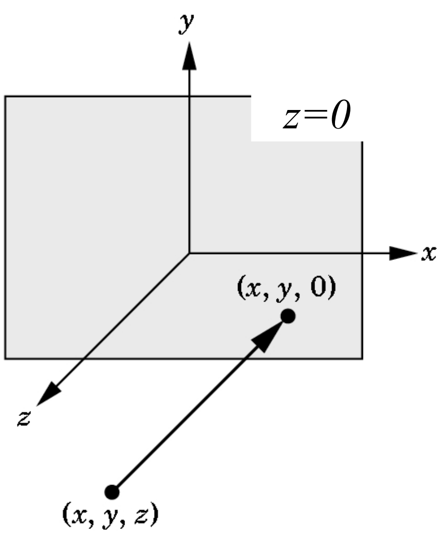
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with a side of length 2



Orthographic Viewing

In the default orthographic view, points are projected forward along the *z* axis onto the plane *z=0*





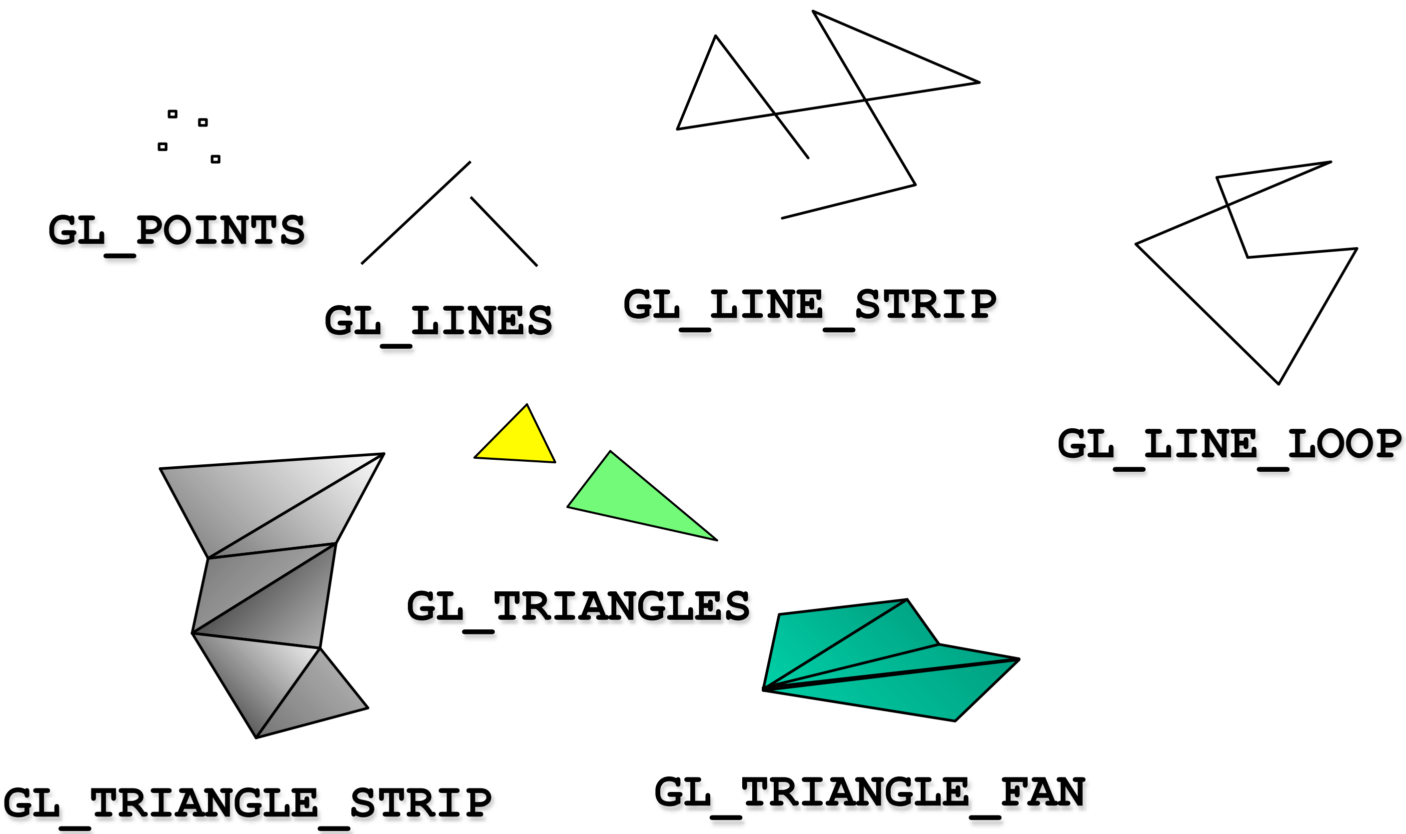
Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

2013

Transformations & Viewing

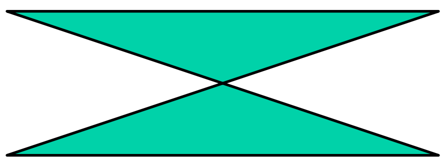
- In OpenGL, projection is carried out by a projection matrix (transformation)
- Transformation functions are also used for changes in coordinate systems
- Pre 3.0 OpenGL had a set of transformation functions which have been deprecated
- Three choices
  - Application code
  - GLSL functions
  - vec.h and mat.h

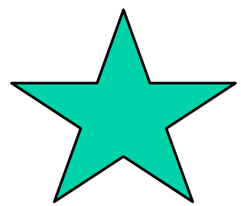
OpenGL Geometry Primitives



Polygons in OpenGL

- OpenGL will only display triangles
  - Simple: edges cannot cross
  - Convex: All points on line segment between two points in a polygon are also in the polygon
  - Flat: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator



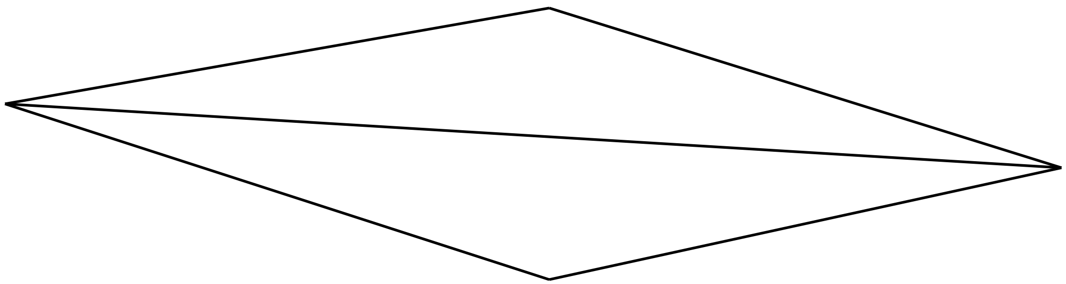


Polygons Testing

- Conceptually simple to test for simplicity and convexity
- Time consuming
- Earlier versions assumed both and left testing to the application
- Present version only renders triangles
- Need algorithm to triangulate an arbitrary polygon

Polygons Testing

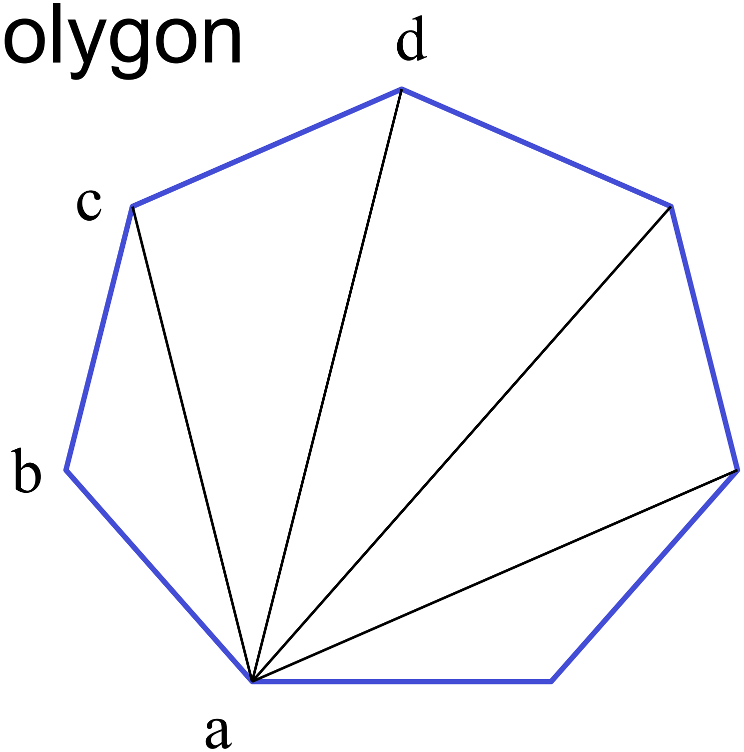
- Long thin triangles render badly



- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points

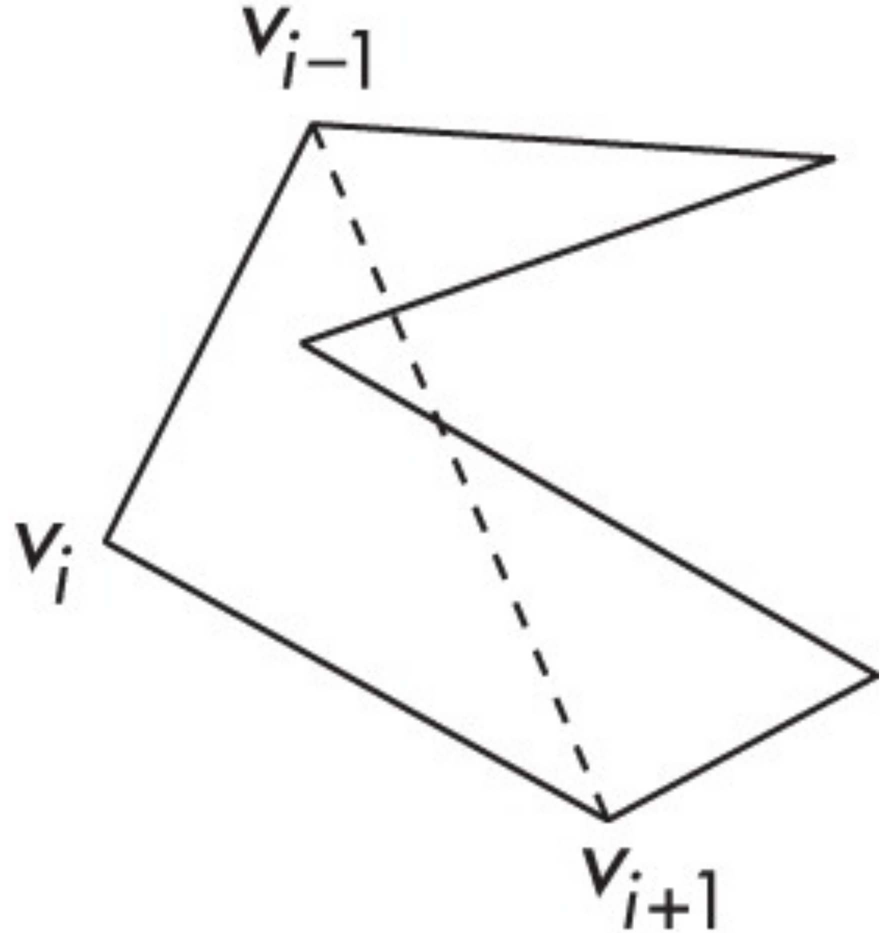
Triangulations

- Convex polygon



- Start with abc, remove b, then acd, ….

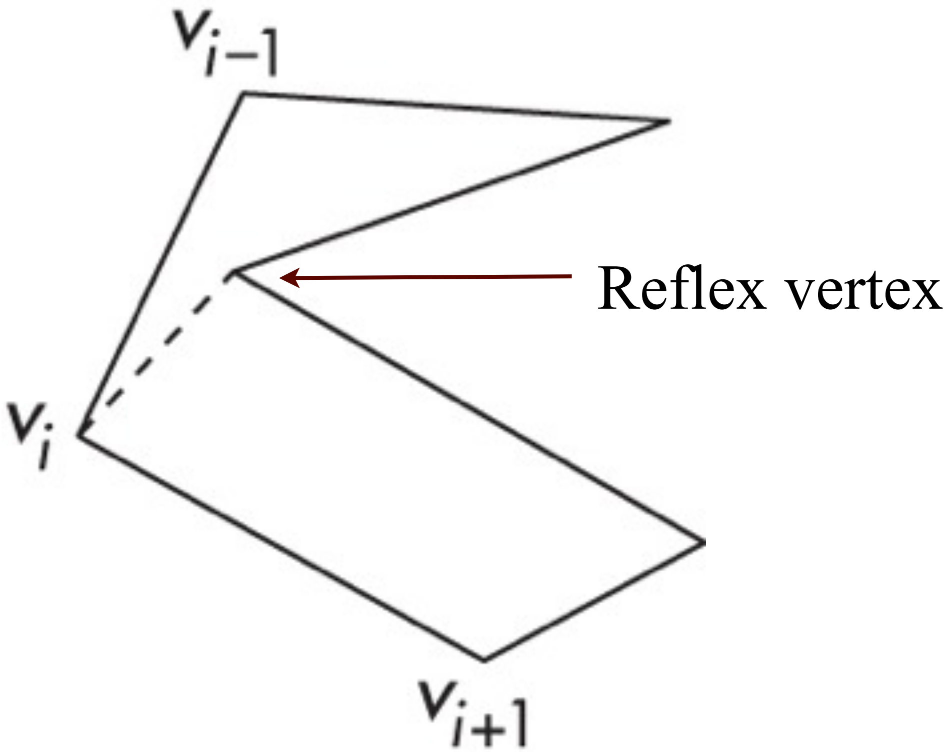
Non-convex (concave)



Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Convex Decomposition

- Find reflex vertices and split.



Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

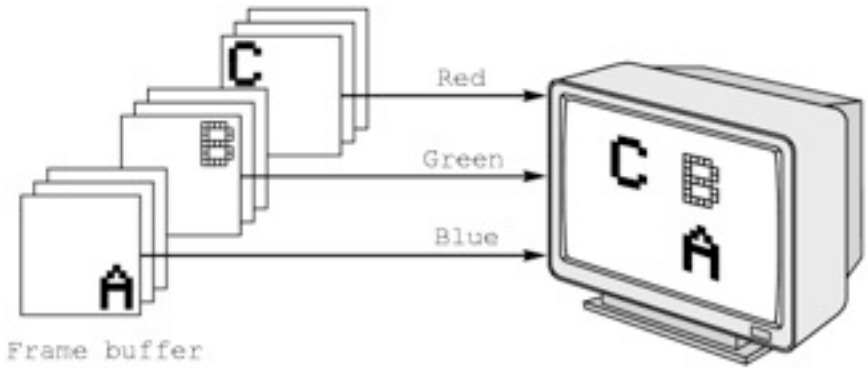
Attributes

- Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
    - Display edges
    - Display vertices
- Only a few (glPointSize) are supported by OpenGL functions

Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

RGB Color in OpenGL

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytels

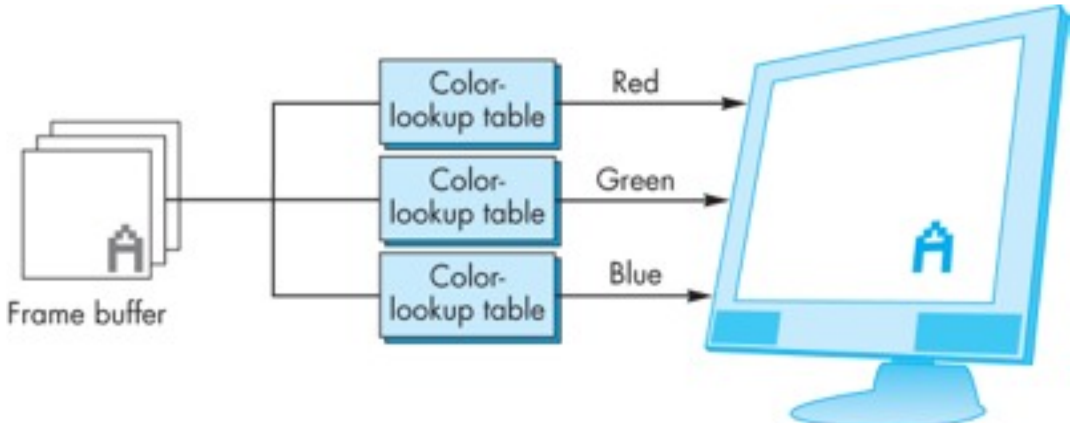


Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

2013

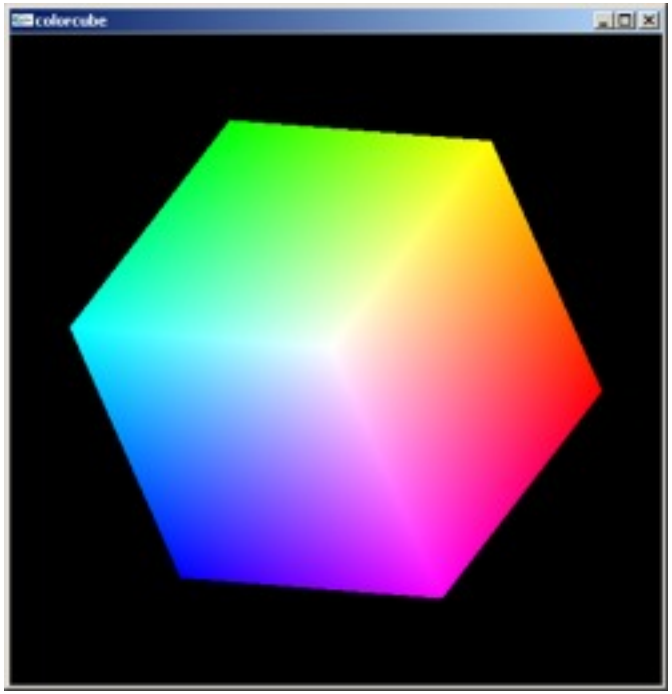
Indexed Color in OpenGL

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



Smooth Color in OpenGL

- Default is *smooth* shading
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is *flat shading*
  - Color of first vertex determines fill color
  - Handle in shader



Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*
2013

Setting Colors

- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Fragment Shader Applications

Per fragment lighting calculations

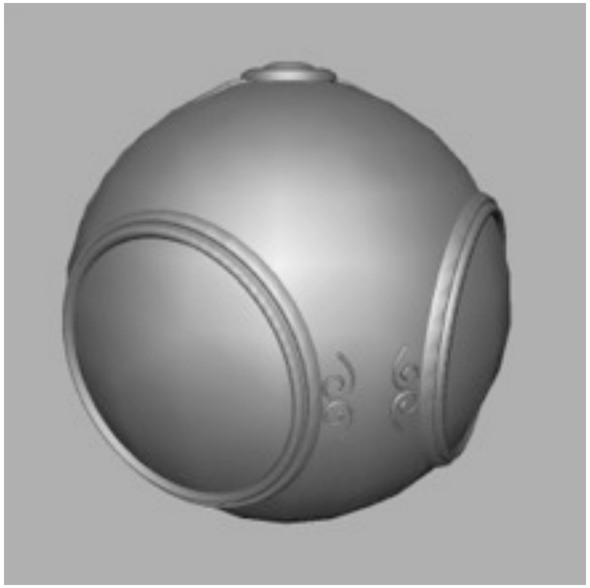
per vertex lighting

per fragment lighting

Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

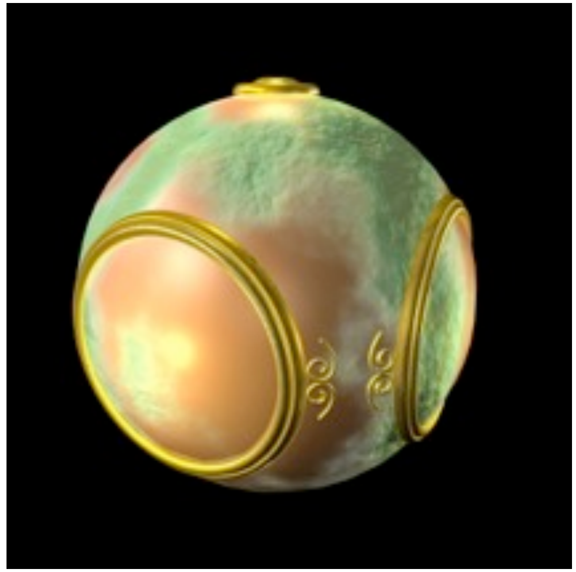
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping



bump mapping

Writing Shader Applications

- First programmable shaders were programmed in an assembly-like manner
- OpenGL extensions added for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

Simple Shader

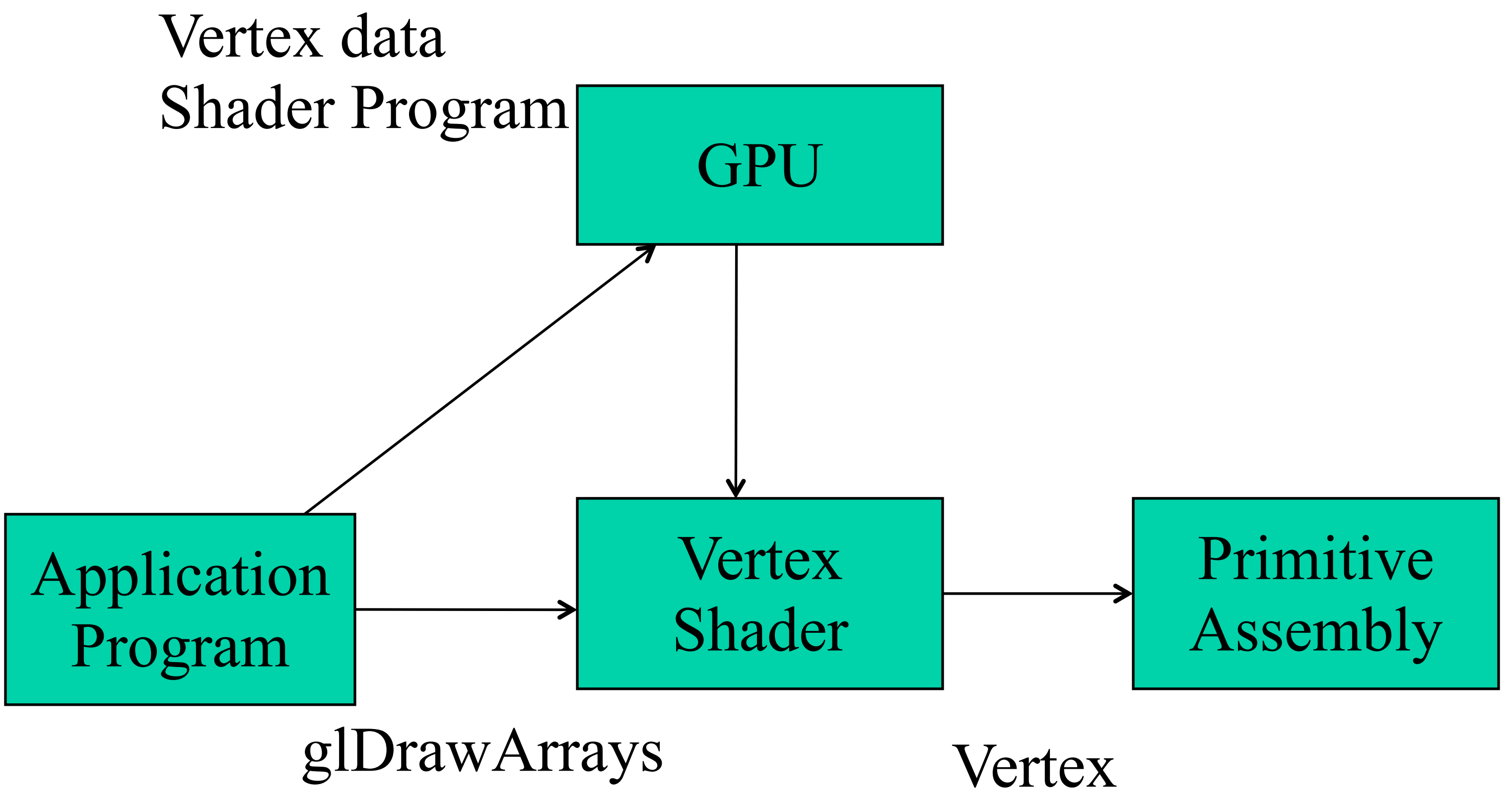
input from application

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

must link to variable in application

built in variable

Execution Model

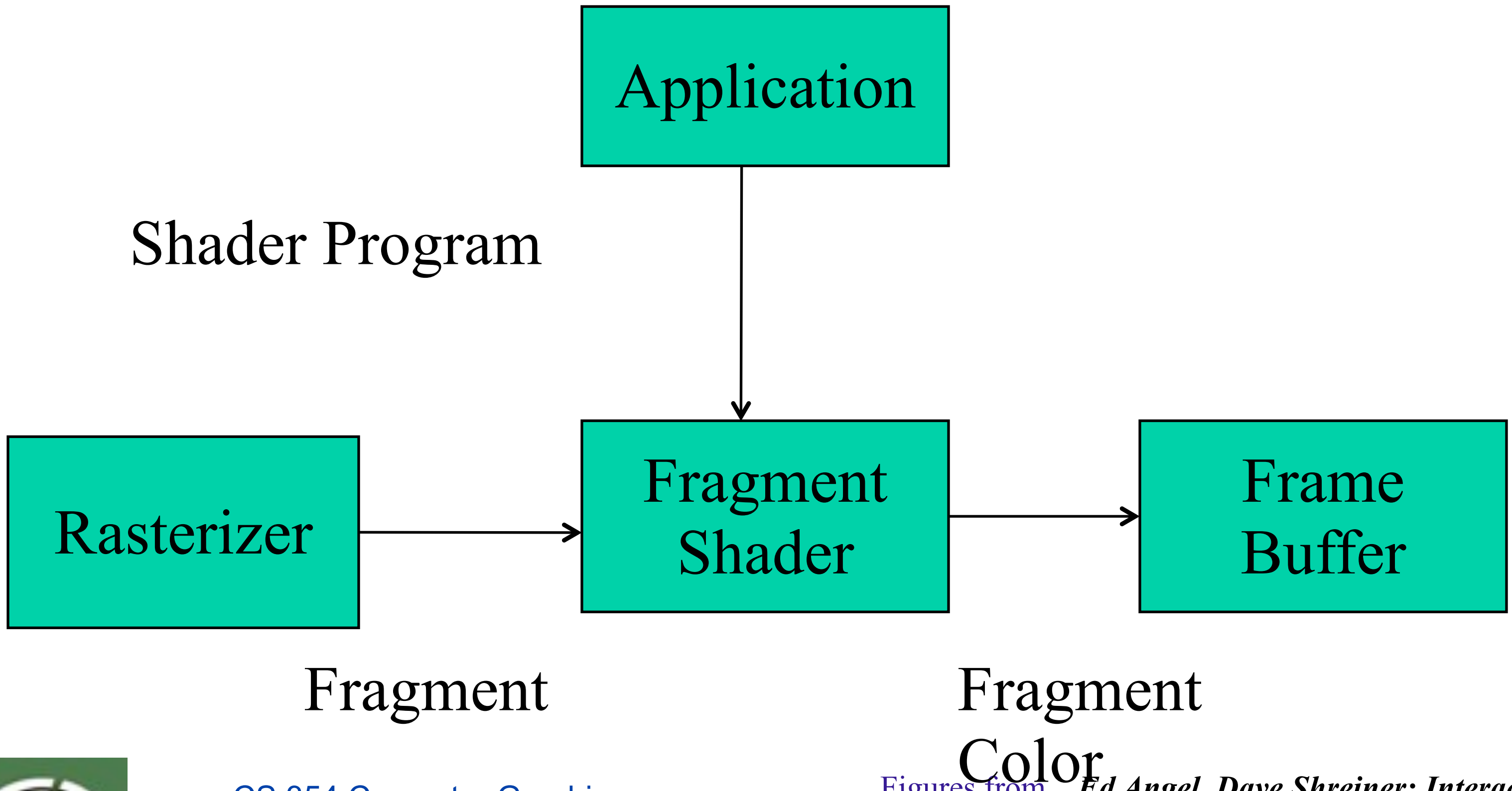


Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Simple Fragment Program

```
void main(void)
{
  gl_FragColor = vec4(1.0, 0.0,
 0.0, 1.0);
}
```

Execution Model



Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

Pointers

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

```
mat3 func(mat3 a)
```

Qualifiers

- GLSL has many of the same qualifiers such as **const** as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
  - Use in qualifier to get to shader
  - **in float temperature**
  - **in vec3 velocity**

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Old style used the varying qualifier

```
varying vec4 color;
```

- Now use **out** in vertex shader and **in** in the fragment shader

```
out vec4 color;
```


Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0,
 0.0, 1.0);
out vec3 color_out;
void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

Required Fragment Shader

```
in vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
// in latest version use form
// out vec4 fragcolor;
// fragcolor = color_out;
```

Passing values

- call by **value-return**
- Variables are copied in
- Returned values are copied back
- Three possibilities
  - **in**
  - **out**
  - **inout** (deprecated)

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored
 as a 1d array
d = a*b; // a row vector stored as
```

Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **a[2]**, **a.b**, **a.z**, **a.p** are the same
- **Swizzling** operator lets us manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
GLuint myProgObj;
myProgObj = glCreateProgram();
/* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function **glShaderSource**
- If the shader is in a file, we can write a reader to convert the file to a string

Shader Reader

```
#include <stdio.h>

static char*
readShaderSource(const char* shaderFile)
{
FILE* fp = fopen(shaderFile, "r");

if ( fp == NULL ) { return NULL; }

fseek(fp, 0L, SEEK_END);
long size = ftell(fp);
```

Shader Reader (cont)

```
fseek(fp, 0L, SEEK_SET);
char* buf = new char[size + 1];
fread(buf, 1, size, fp);

buf[size] = '\0';
fclose(fp);

return buf;
}
```

Adding a Vertex Shader

```
GLuint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource =
    readShaderSource(vShaderFile);
glShaderSource(myVertexObj,
    1, &vertexShaderFile, NULL);
myVertexObj =
    glCreateShader(GL_VERTEX_SHADER);
glCompileShader(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
```

Vertex Attributes

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables

Vertex Attribute Example

```
#define BUFFER_OFFSET( offset )
        ((GLvoid*) (offset))

GLuint loc =
glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Uniform Variable Example

```
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj,
                    "angle");
/* angle defined in shader */

/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

glUniform1f(angleParam, my_angle);
```

Double Buffering

- Updating the value of a uniform variable opens the door to animating an application
  - Execute glUniform in display callback
  - Force a redraw through glutPostRedisplay()
- Need to prevent a partially redrawn frame buffer from being displayed
- Draw into back buffer
- Display front buffer
- Swap buffers after updating finished

Adding Double Buffering

- Request a double buffer
  - glutInitDisplayMode(GLUT_DOUBLE)
- Swap buffers

```
void mydisplay()
{
        glClear(……);
        glDrawArrays();
        glutSwapBuffers();
}
```

Idle Callback

- Idle callback specifies function to be executed when no other actions pending
  - glutIdleFunc(myIdle);

```
void myIdle()
{
    // recompute display
    glutPostRedisplay();
}
```


Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*
2013

Attribute and Varying Qualifiers

- Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
- No changes needed in application
- Vertex shader example:

```
#version 1.4
attribute vec3 vPosition;
varying vec3 color;
```

```
#version 1.5
in vec3 vPosition;
out vec3 color;
```

Adding Color

- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes
- Let's associate a color with each vertex
- Set up an array of same size as positions
- Send to GPU as a vertex buffer object

Setting Colors

```
typedef  vec3 color3;
color3 base_colors[4] = {color3(1.0, 0.0. 0.0), ….
color3 colors[NumVertices];
vec3 points[NumVertices];

//in loop setting positions

colors[i] = basecolors[color_index]
position[i] = …….
```

Setting Up Buffer Object

```
//need larger buffer

glBufferData(GL_ARRAY_BUFFER, sizeof(points) +
  sizeof(colors), NULL, GL_STATIC_DRAW);

//load data separately

glBufferSubData(GL_ARRAY_BUFFER, 0,
  sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points),
  sizeof(colors), colors);
```

Second Vertex Array

// vPosition and vColor identifiers in vertex shader

```
loc = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(0));

loc2 = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(sizeofpoints));
```

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Wave Motion Vertex Shader

-

```
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h; // height scale
void main()
{
vec4 t = vPosition;
t.y = vPosition.y
+ h*sin(time + xs*vPosition.x)
+ h*sin(time + zs*vPosition.z);
gl_Position = t;
}
```

Particle System

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{
vec3 object_pos;
object_pos.x = vPosition.x + vel.x*t;
object_pos.y = vPosition.y + vel.y*t
    + g/(2.0*m)*t*t;
object_pos.z = vPosition.z + vel.z*t;
gl_Position =
ModelViewProjectionMatrix*vec4(object_pos,1);
}
```

Pass Through Fragment Shader

```
/* pass-through fragment shader */

in vec4 color;
void main(void)
{
gl_FragColor = color;
}
```

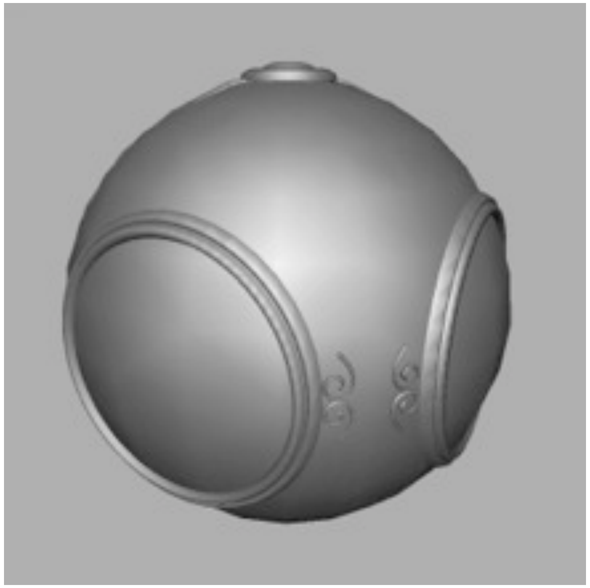
Vertex vs Fragment Lighting

per vertex lighting

per fragment lighting

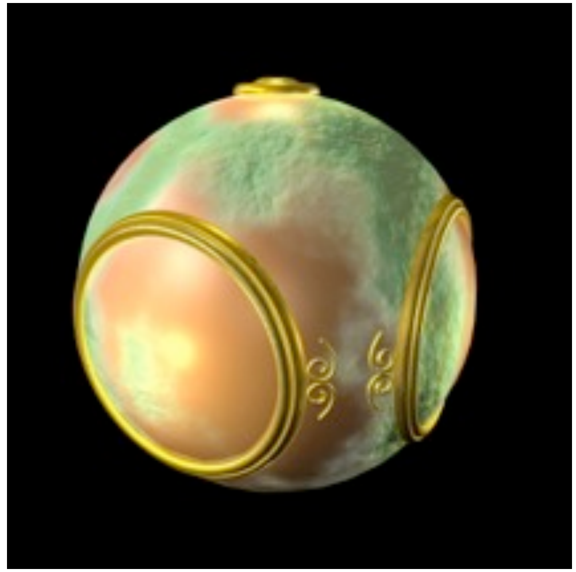
Fragment Shader Applications

Texture mapping



smooth shading

environment mapping

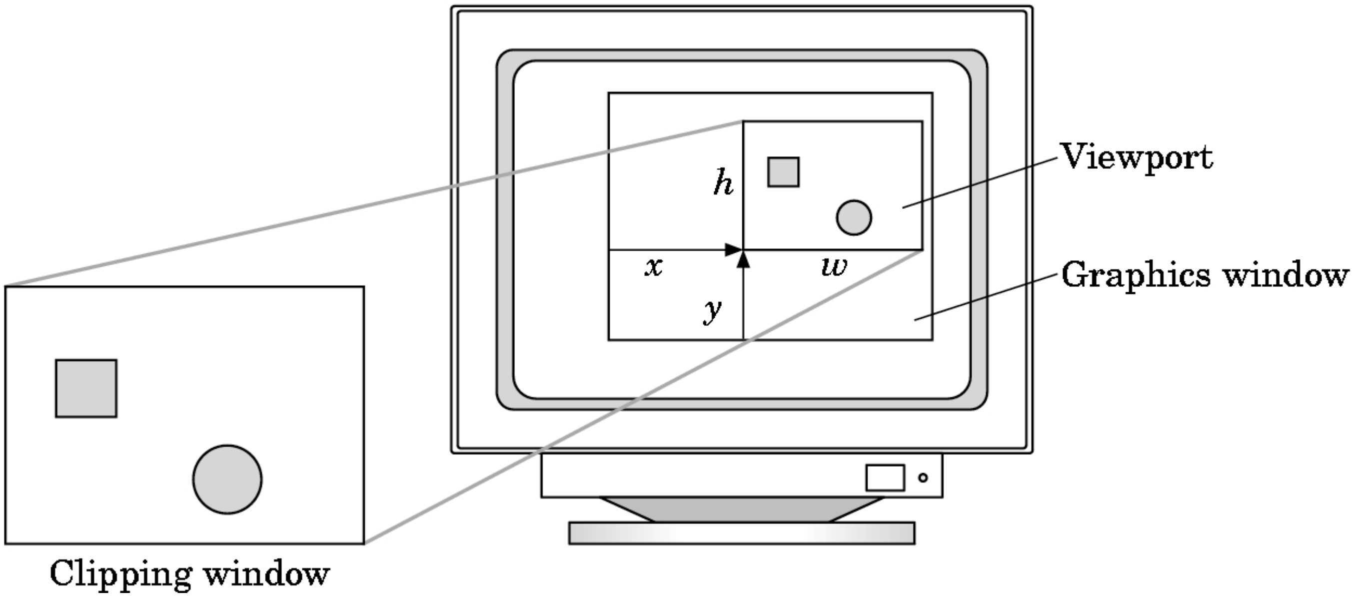


bump mapping

Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*

Viewports

- Do not have use the entire window for the image: **`glViewport(x,y,w,h)`**
- Values in pixels (screen coordinates)



Figures from *Ed Angel, Dave Shreiner: Interactive Computer Graphics, 6th Ed., 2012 © Addison Wesley*