Last week
Overview of a graphics system

- Input devices
- Image formed and stored in frame buffer
- Output device

© Machiraju/Zhang/Möller
Ray tracing: the algorithm

for each pixel on screen
determine ray from eye through pixel
  if ray shoots into infinity, return a background color
  if ray shoots into light source, return light color appropriately
find closest intersection of ray with an object
  cast off shadow ray (towards light sources)
  if shadow ray hits a light source, compute light contribution according to some illumination model
cast reflected and refracted ray, recursively calculate pixel color contribution
return pixel color after some absorption
The algorithm

for each polygon in the scene
  project its vertices onto viewing (image) plane
  for each pixel inside the polygon formed on viewing plane
    determine point on polygon corresponding to this pixel
    get pixel color according to some illumination model
    get depth value for this pixel (distance from point to plane)
    if depth value < stored depth value for the pixel
      update pixel color in frame buffer
      update depth value in depth buffer (z-buffer)
  end if
end if

Question: What does the depth buffer store after running the z-buffer algorithm for the scene? Does polygon order change depth? Image?
Development of the OpenGL API
Early History of APIs

• IFIPS (1973) formed two committees to come up with a standard graphics API
  – Graphical Kernel System (GKS)
    • 2D but contained good workstation model
  – Core
    • Both 2D and 3D
  – GKS adopted as ISO and later ANSI standard (1980s)

• GKS not easily extended to 3D (GKS-3D)
  – Far behind hardware development

© Angel/Shreiner/Möller
PHIGS and X

- **Programmers Hierarchical Graphics System (PHIGS)**
  - Arose from CAD community
  - Database model with retained graphics (structures)

- **X Window System**
  - DEC/MIT effort
  - Client-server architecture with graphics

- **PEX combined the two**
  - Not easy to use (all the defects of each)
SGI and GL

• Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
• To access the system, application programmers used a library called GL
• With GL, it was relatively simple to program three dimensional interactive applications
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
OpenGL Evolution

- Originally controlled by an Architectural Review Board (ARB)
  - Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, …
  - Now Khronos 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

© Angel/Shreiner/Möller
OpenGL Architecture
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
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
  – Will only work with legacy code

• Links with window system
  – GLX for X window systems
  – WGL for Windows
  – AGL for Macintosh

© Angel/Shreiner/Möller
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
Graphics libraries

• GLUI: GLUT-based user interface library (libglui)

• Offer controls to OpenGL applications, e.g.,
  – buttons,
  – checkboxes,
  – radio buttons,
  – etc.
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

© Angel/Shreiner/Möller
OpenGL is an API …

- **Application Programmers’ Interface:**
  - low-level: graphics hardware
  - high-level: application program you write

OpenGL ES (for embedded systems): a subset of desktop OpenGL, providing lightweight interface for 2D/3D graphics on mobile and hand-held devices, etc.
OpenGL Functions
OpenGL Functions

- Primitives
  - Points
  - Line Segments
  - Triangles
- 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
Lack of Object Orientation

• 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

\[ \text{glUniform3f}(x, y, z) \]

- Function name: \text{glUniform3f}
- Dimensions: 3 floats: \(x, y, z\)
- Belongs to GL library
- \(x, y, z\) are floats

\[ \text{glUniform3fv}(p) \]
- \(p\) is a pointer to an array

© Angel/Shreiner/Möller
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`, ...

© Angel/Shreiner/Möller
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
Today

• Development of the OpenGL API
• OpenGL Architecture
  – OpenGL as a state machine
  – OpenGL as a data flow machine
• Functions
  – Types
  – Formats
• Simple program
• Shaders
• Colours and attributes

Readings: Sections 2.1-2.10
First Simple Program
Objectives

• Build a complete first program
  – Introduce shaders
  – Introduce a standard program structure

• Simple viewing
  – Two-dimensional viewing as a special case of three-dimensional viewing

• Initialization steps and program structure
Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - `main()`:
    - enters event loop (last executable statement)
  - `init()`: sets the state variables
    - specifies the callback functions
    - opens one or more windows with the required properties
    - viewing
    - attributes
  - `initShader()`:
    - create geometry
    - read, compile and link shaders
  - callbacks
    - Display function
    - Input and window functions
#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();
}
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
- **glutInitWindowSize** in pixels
- **glutInitWindowPosition** 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
Display Callback

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

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

• Arrays are buffer objects that contain vertex arrays

© Angel/Shreiner/Möller
Vertex Arrays

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

```c
point2 points[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
  
  ```c
  GLuint 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

• Buffer 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, GL_STATIC_DRAW);

• Data in current vertex array is sent to GPU
Initialization

• Vertex array objects and buffer objects can be set up in `init()`
• Set clear color and other OpenGL parameters
• Set up shaders as part of initialization
  – Read
  – Compile
  – Link
• First let’s consider a few other issues
Coordinate Systems

• The units in **points** are determined by the application and are called
  – *object* (or *model*) coordinates
  – *world* coordinates

• Viewing specifications usually are also in object coordinates transformed through
  – *eye* (or *camera*) coordinates
  – *clip* coordinates
  – normalized *device* coordinates
  – *window* (or *screen*) coordinates

• OpenGL also uses some internal representations that usually are not visible to the application but are important in the shaders
OpenGL Camera

• 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 sides of
  – length 2
Orthographic Viewing

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

• Do not have to use the entire window for the image: \texttt{glViewport(x,y,w,h)}

• Values in pixels (window coordinates)
Transformations and 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 Shaders
Objectives

• Simple Shaders
  – Vertex shader
  – Fragment shaders

• Programming shaders with GLSL

• Finish first program
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

© Angel/Shreiner/Möller
Fragment Shader Applications

- Texture mapping

smooth shading  environment mapping  bump mapping

© Angel/Shreiner/Möller
Writing Shaders

• 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)
GLSL

• OpenGL Shading Language
• Part of OpenGL 2.0 and up
• High level C-like language
• New data types
  – Matrices
  – Vectors
  – Samplers
• As of OpenGL 3.1, application must provide shaders
Simple Vertex Shader

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

- **input from application**
- **must link to variable in application**
- **built in variable**
Simple Fragment Program

```cpp
out vec4 fColor;

void main(void)
{
    fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```
Execution Model

Application Program

Vertex data

Shader Program

Shader Program

Vertex processor

Clipper and primitive assembler

Rasterizer

Fragment processor

Vertices

ResponseType

Fragment

Fragment Color

GPU

glDrawArrays
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 out from GLSL
  functions, e.g.

  \texttt{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
  – not at all (`const`)
  – Once per primitive (‘uniform’)
  – Once per vertex (‘attribute’)
  – Once per fragment (‘varying’)
  – At any time in the application
• Vertex attributes are interpolated by the rasterizer into fragment attributes
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
Attribute Qualifier

• Attribute-qualified variables can change at most once per vertex

• There are a few built in variables such as \texttt{gl\_Position} but most have been deprecated

• User defined (in application program)
  – Use \texttt{in} qualifier to get to shader
    – \texttt{in float temperature}
    – \texttt{in vec3 velocity}
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;`
Example: Vertex Shader

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

Required Fragment Shader
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 a 1D array
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

• **Swizzing** operator lets us manipulate components
  – `vec4 a;`
  – `a.yz = vec2(1.0, 2.0);`
Colour and attributes
Objectives

• Expanding primitive set
• Adding color
• Vertex attributes
• Uniform variables
OpenGL Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN

© Angel/Shreiner/Möller
Polygon Issues

• 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

nonsimple polygon  nonconvex polygon
Polygon 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
Good and Bad Triangles

• Long thin triangles render badly

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

- Convex polygon

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

• Attributes determine the appearance of objects
  – Colour (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
RGB colour

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

• Colours 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 Colour

• Default is *smooth* shading
  – OpenGL interpolates vertex colors across visible polygons

• Alternative is *flat* shading
  – Colour of first vertex determines fill color
  – Handle in shader
Setting Colours

• Colours are ultimately set in the fragment shader but can be determined in either shader or in the application
• Application colour: 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
More GLSL
Objectives

• Coupling shaders to applications
  – Reading
  – Compiling
  – Linking

• Vertex Attributes

• Setting up uniform variables

• Example applications
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);
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
#define BUFFER_OFFSET( offset )
    ((GLvoid*) (offset))

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

```c
void mydisplay()
{
    glClear(......);
    glDrawArrays();
    glutSwapBuffers();
}
```
Idle Callback

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

```c
void myIdle()
{
    // recompute display
    glutPostRedisplay();
}
```
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
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 = glGetUniformLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(0));

loc2 = glGetUniformLocation(program, "vColor");
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeof(points)));