Objectives

- Development of the OpenGL API
- OpenGL Architecture
  - OpenGL as a state machine
  - OpenGL as a data flow machine
- Functions
  - Types
  - Formats
- Simple program

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

PHIGS and X

- Programmers Hierarchical Graphics System (PHIGS)
  - Arose from CAD community
  - Database model with retained graphics (structures)
- X Window System
  - DEC/VT 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 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

OpenGL Evolution – Stable?

- 1.0 Jan 1992
- 1.1 Jan 1997
- 1.2 Mar 1998 1.2.1 Oct 1998
- 1.3 Aug 2001
- 1.4 Jul 2002
- 1.5 Jul 2003
- 2.0 Sep 2004 – programmable shaders
- 2.1 Sep 2006 – Pixel buffer obj, non-square mكيف، GLSL 1.2
- 3.0 Jul 2008 – frame buffer objs, vertex array objs, HW instancing, GLSL 1.3, new precisions, texture arrays
- 3.1 Mar 2009 – GLSL 1.4 ter buff obj, uni buf obj, signed bfr, instancing, copybuffer, primitive restart
- 3.2 Aug 2009 – GLSL 1.5, geom shader,
- 3.3 Mar 2010 – Same as 4.0 allows 4.0 on older HW
- 4.0 Mar 2010 – GLSL 4.0 new shdr stages (tessellation), better fragment shading, shdr subroutines, 64-bit, etc.
- 4.1 Jul 2010 – Compatible with OpenGL 2.0, binary shdr
- 4.2 Aug 2011 – atomic ops, mostly compressed texture, pack 6/16 in 32
- 4.3 Aug 2012 – compute shaders, storage Buff Objs, ES3.0 compat, mem security.
- 4.4 Jul 2013 – Buffer placement control, aync queries, shader var layout, bind obj

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 3.2

- Core and compatibility profile
  - Core – new stuff
  - Compatibility – deprecated fixed-function APIs
  - Geometry shaders
OpenGL 4.0/3.3
- GLSL 4.0
- New shader stages for tessellation
- Dynamically link shader subroutines
- Better fragment shading
- Support for OpenGL
- 3.3 for older hardware
- 4.0 for newer

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
  - Compatible with OpenGL ES2
- OpenGL 4.2
  - Add geometry shaders
- OpenGL 4.3
  - Computer Shaders
- OpenGL 4.4
  - Looks like mostly efficiencies

What About Direct X and Direct 3D?
- 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

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
  - 3.7 came out in 1998
  - Development has been abandoned
  - License does not allow redistributing modified code
- freeglut updates GLUT
  - Added capabilities
  - Context checking
Alternatives to GLUT (c,c++)

- GLFW
- SDL – few OpenGL examples
- SFML
- Allegro 5 – didn’t compile on linux out of box
- FLTK
- GTK
- Qt
- wxWidgets
- Others ...

- Better windows/interaction
- Steeper learning curves

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()
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

```
function name
   x, y, z
```

belongs to GL library

x, y, z are floats

```
function name
   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 of 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

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

It used to be easy

```
#include <GL/glut.h>
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_QUAD);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glEnd()
}
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

Event Loop

- Note that the program specifies 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
Notes on compilation

• See website and ftp for examples
• Unix/linux
  • Include files usually in .../include/GL
  • Compile with -lglut -lgl library 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

Compilation on Windows

• Visual C++
  • Get glut.h, glut32.lib and glut32.dll from web
  • Install in same places as corresponding OpenGL files
  • Create an empty application
  • Add glut32.lib to project settings (under link tab)
  • Same for freetype and GLEW
• Cygwin (Linux under Windows)
  • Can use gcc and similar makefile to Linux
  • Use -lopengl32 -lglut32 flags

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()
    • 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
  • initShaders(): read, compile and link shaders
  • callbacks
    • Display function
    • Input and window functions

simple.c revisited

• main() function similar to last lecture
  • Mostly GLUT functions
• init() will allow more flexible for colors
• initShaders() 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();
}
```

GLUT functions

- `glutInit` allows application to get command line arguments and initializes system
- `glutInitDisplayMode` 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

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

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 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

```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

**WARNING!** Does not work with GLEW and OpenGL > 3.1 on linux
- See code page for fixes

Buffer Object

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

```c
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices);
```
- Data in current vertex array is sent to GPU

Initialization

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

Coordinate Systems

- You define the coordinate system for your application
- Pick ones that make sense for you application
  - E.g. if you are doing a 2d problem like drawing lines, pick coordinates that you know are going to be visible on the screen
  - Having your world coordinate system align with the pixel coordinate system (except for Y being flipped) makes things easier
- You pick the units
  - Pick inches or feet, or meters, or whatever makes sense for the problem.
  - If you are modeling a solar system don’t pick meters as your units.
  - If you are modeling a house, don’t pick light years as you units.

Coordinate Systems

- 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

- 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

Objectives

- Simple Shaders
  - Vertex shader
  - Fragment shaders
- Programming shaders with GLSL
- Finish first program

SHADERS
**Vertex Shader Applications**

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

**Fragment Shader Applications**

Per fragment lighting calculations

- Texture mapping
- smooth shading
- environment mapping
- bump mapping

**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;
}
```
**Execution Model**

- **Vertex Shader**
- **GPU**
- **Primitive Assembly**
- **Vertex data**

**Application Program**

```
glDrawArrays
```

**Simple Fragment Program**

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

**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.
  ```c
  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;`

Example: Vertex Shader
```glsl
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
```glsl
in vec3 color_out;
void main(void)
{
  gl_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 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.s, a.p are the same
  ```
- **Swizzling** operator lets us manipulate components
  ```
  vec4 a;
  a.yz = vec2(1.0, 2.0);
  ```