Programming with OpenGL
Part 1: Background

CS 432 Interactive Computer Graphics

Prof. David E. Breen

Department of Computer Science

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 IS0 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/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 graphics 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

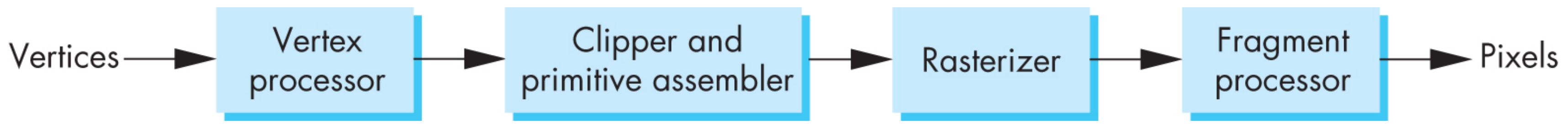
- Easy to use
- Close enough to the hardware to get excellent performance
- Focused on rendering
- Omitted windowing and input to avoid window system dependencies

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

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 (2009)

- 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
  - *deprecate* in CS - To mark (a component of a software standard) as obsolete to warn against its use in the future, so that it may be phased out.
- 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
  - Added 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

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 and other GUI widgets

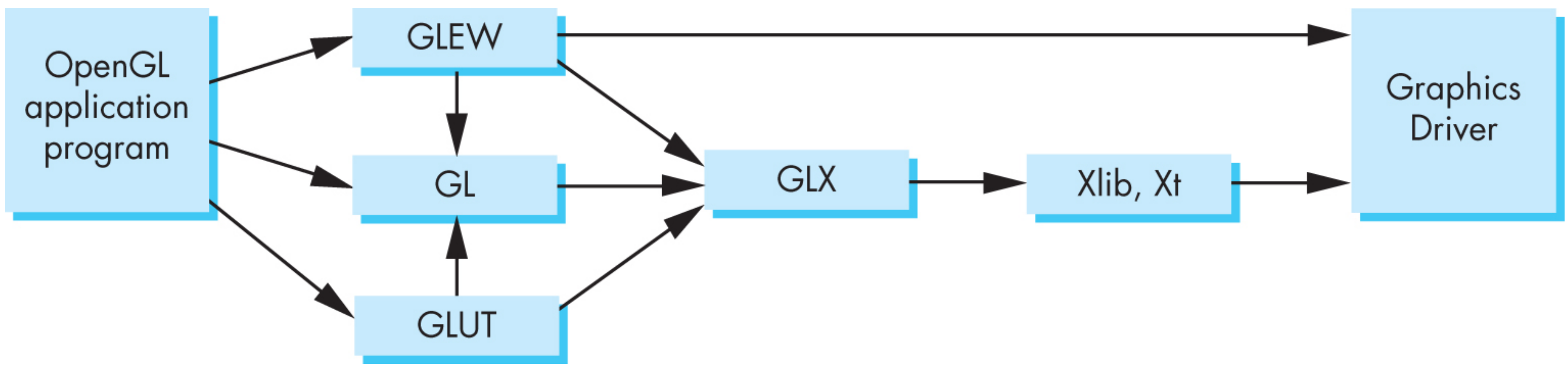
freeglut

- 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

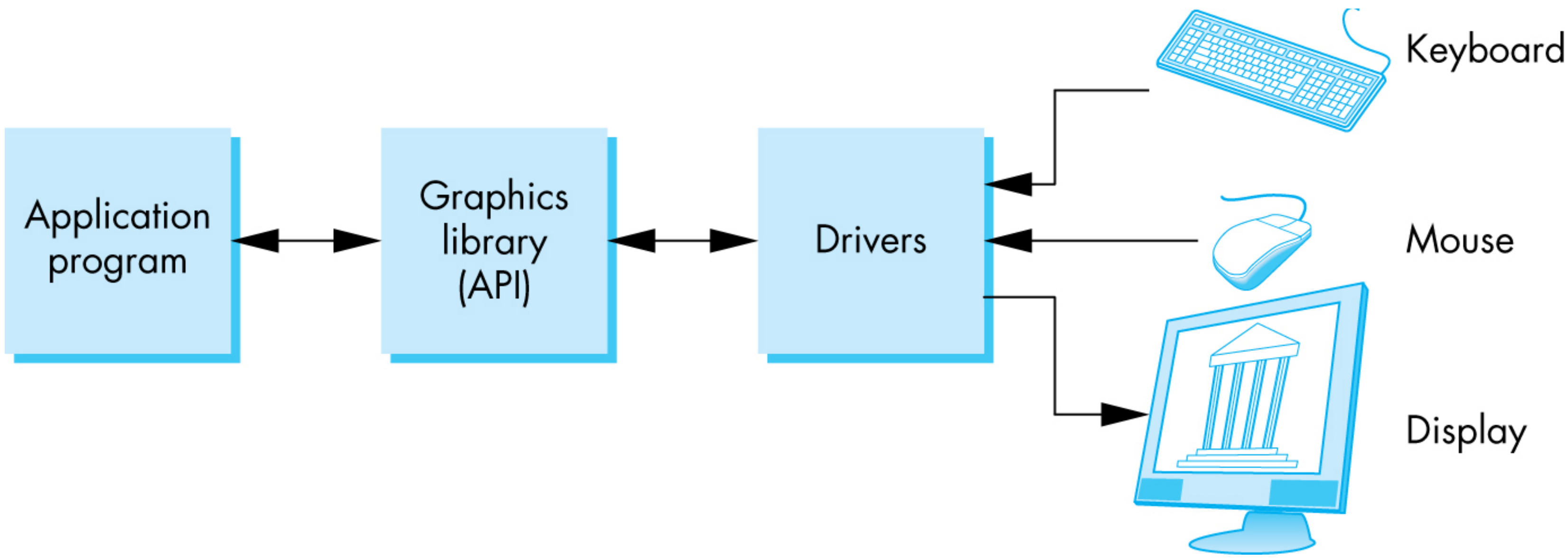
GLEW

- 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



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

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

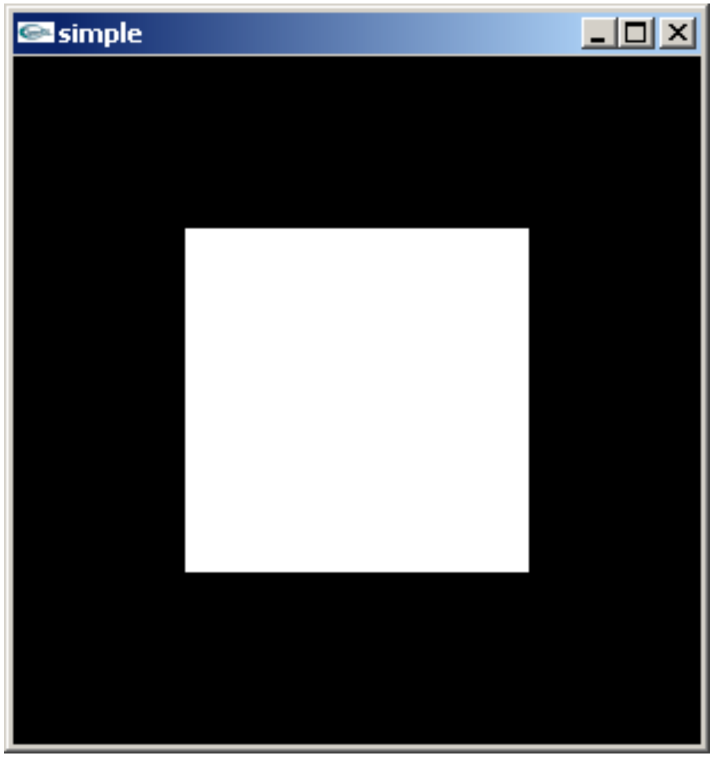
- Shader based OpenGL is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre-3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU

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



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

simple.c

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

// need to fill in this part
// and define shaders
}

int main(int argc, char** argv){
     glutCreateWindow("simple");
     glutDisplayFunc(mydisplay);
     glutMainLoop();
}
```

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 class website for details
- Unix/linux
  - Include files usually in …/include/GL
  - Compile with –lglut –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

Programming with OpenGL Part 2: Complete Programs

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
  - **`initShader()`**: 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 colors
- initShader() will hide 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

```
#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 dimensions)
  - 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

Immediate vs. Retained

- Immediate
  - Every time scene changes, the whole scene must be evaluated and sent to GPU
  - OK, if scene doesn't change much
  - GPU memory doesn't limit scene size
  - Needs high bandwidth between CPU and GPU
- Retained
  - Send scene once. Only send incremental changes
  - Removes CPU-GPU bottleneck
  - GPU needs much more memory, that can be randomly accessed

Display Callback

- Once we get data to GPU, 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
- Vertex array holds these data in application
- 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

```
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 vertex arrays

Buffer Object

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

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

- Data in current buffer is sent to GPU

Why use Buffer Objects?

Only Advantages

- The memory manager in the buffer object will put the data into the best memory locations based on user's hints
- Memory manager can optimize the buffers by balancing between 3 kinds of memory:
  - system, GPU and video memory
- Shares the buffer objects with many clients. Since BO is on the server's side, multiple clients will be able to access the same buffer with the corresponding identifier

Next

- How to
  - Create a BO
  - Draw a BO
  - Update a BO

Creating BOs

- Generate a new buffer object with glGenBuffers()
- Bind the buffer object with glBindBuffer()
  - i.e. make a buffer object "current"
- Copy vertex data to the buffer object with glBufferData()

- glGenBuffers()
  - creates buffer objects and returns the identifiers of the buffer objects

  void glGenBuffers(GLsizei n, GLuint* ids)

    - n: number of buffer objects to create
    - ids: the address of a GLuint variable or array to store a single ID or multiple IDs

glBindBuffer()

- Once the buffer object has been created, we need to connect it with the corresponding ID before use

void glBindBuffer(GLenum target, GLuint id)

- Target can be
  - GL_ARRAY_BUFFER: Any vertex attribute, such as vertex coordinates, texture coordinates, normals and color component arrays
  - GL_ELEMENT_ARRAY_BUFFER: Index array which is used for glDraw[Range]Elements()
- Once first called, the buffer is initialized with a zero-sized memory buffer and sets the initial states

glBufferData()

- You can copy the data into the buffer object with glBufferData() after the buffer has been initialized.

```
void glBufferData(GLenum target, GLsizei size,
                  const void* data, GLenum usage)
```

- target is either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.
- size is the number of bytes of data to transfer.
- The third parameter is the pointer to the array of source data.
- "usage" flag is a performance hint to provide how the buffer object is going to be used: static, dynamic or stream, and read, copy or draw.

Usage Flags

- GL_STATIC_DRAW
- GL_STATIC_READ
- GL_STATIC_COPY
- GL_DYNAMIC_DRAW
- GL_DYNAMIC_READ
- GL_DYNAMIC_COPY
- GL_STREAM_DRAW
- GL_STREAM_READ
- GL_STREAM_COPY

  - Static: data in BO will not be changed
  - Dynamic: the data will be changed frequently
  - Stream: the data will be changed every frame
  - Draw: the data will be sent to GPU in order to draw
  - Read: the data will be read by the client's application
  - Copy: the data will be used both drawing and reading

glBufferSubData()

void glBufferSubData(GLenum target,
GLint offset, GLsizei size, void* data)

- Like glBufferData(),
  - used to copy data into BO
- It only replaces a range of data into the existing buffer, starting from the given offset.
- The total size of the buffer must be set by glBufferData() before using glBufferSubData().

DeleteBuffers()

void glDeleteBuffers(GLsizei n, const GLuint* ids)

- You can delete a single BO or multiple BOs with glDeleteBuffers() if they are not used anymore. After a buffer object is deleted, its contents will be lost.

Initialization

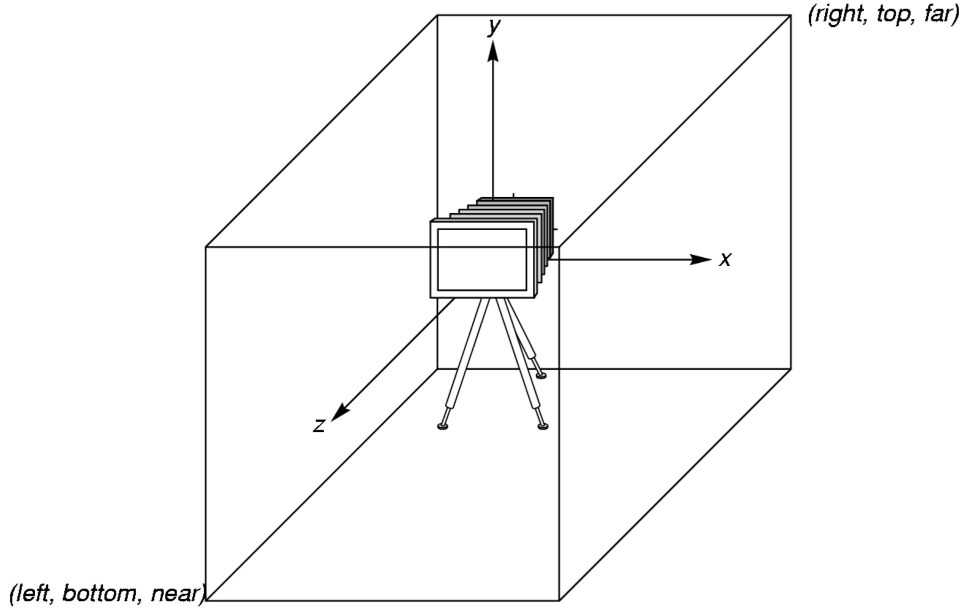
- Vertex array objects and buffer objects can be set up in **init()**
- Also set clear color and other OpenGL parameters
- Also set up shaders as part of initialization
  - Read
  - Compile
  - Link
- First let's consider a few other issues

Coordinate Systems

- 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

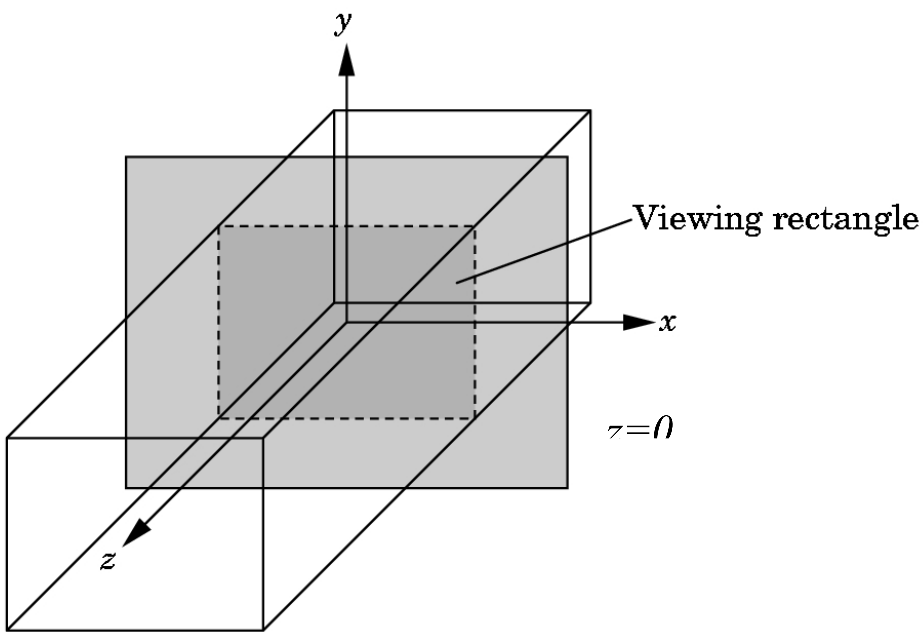
OpenGL Camera

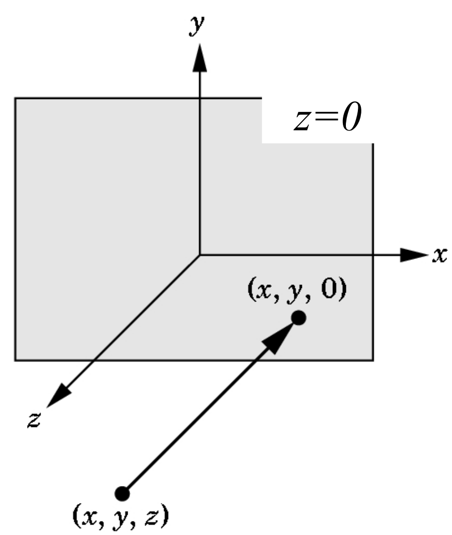
- OpenGL places a camera at the origin in object space pointing in the negative z direction
- The default viewing volume is a box centered at the origin with sides of length 2
- (-1,-1,-1) ➔ (1,1,1)



Orthographic Viewing

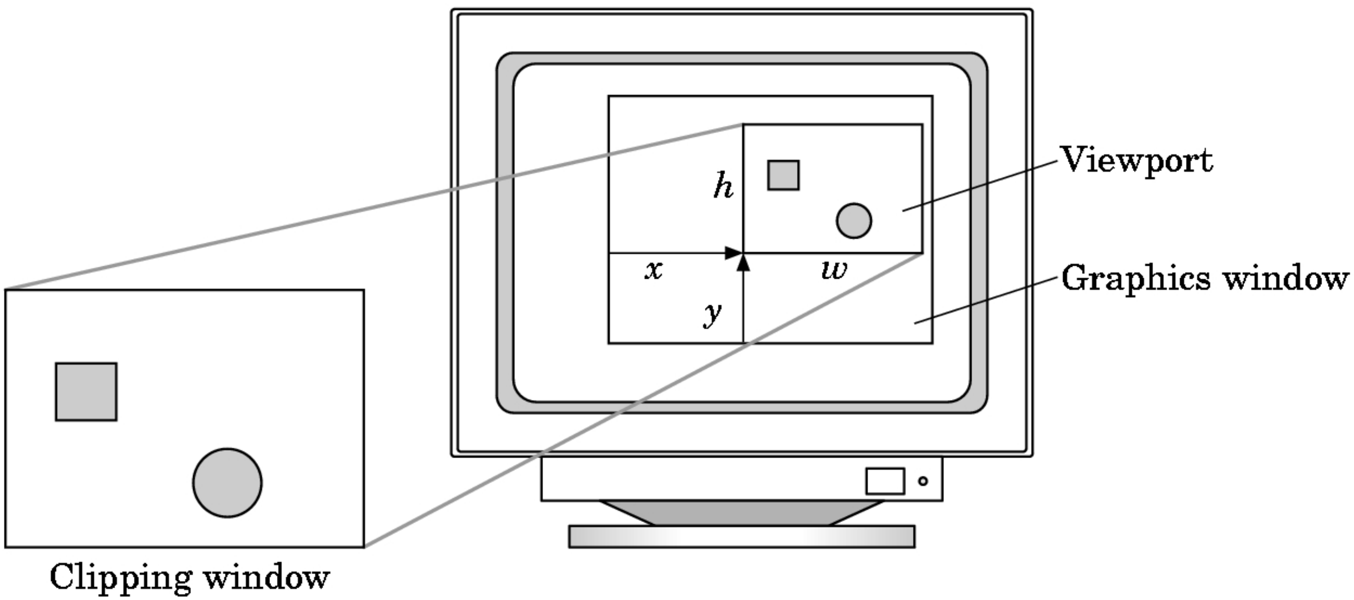
In the default orthographic (parallal) view, points are projected forward along the z axis onto the plane $z = 0$.





Viewports

- Do not have to use the entire window for the image: **`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

First Programming Assignment

- Get test code running
- Make minor modifications to it