Computer Graphics Programming

CGT 520

Computer Graphics Technology Dept.
Purdue University

January 27, 2015

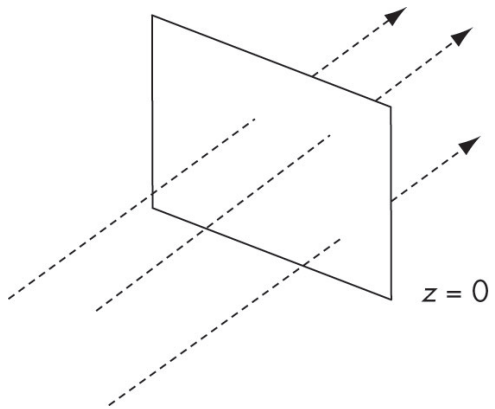
Outline

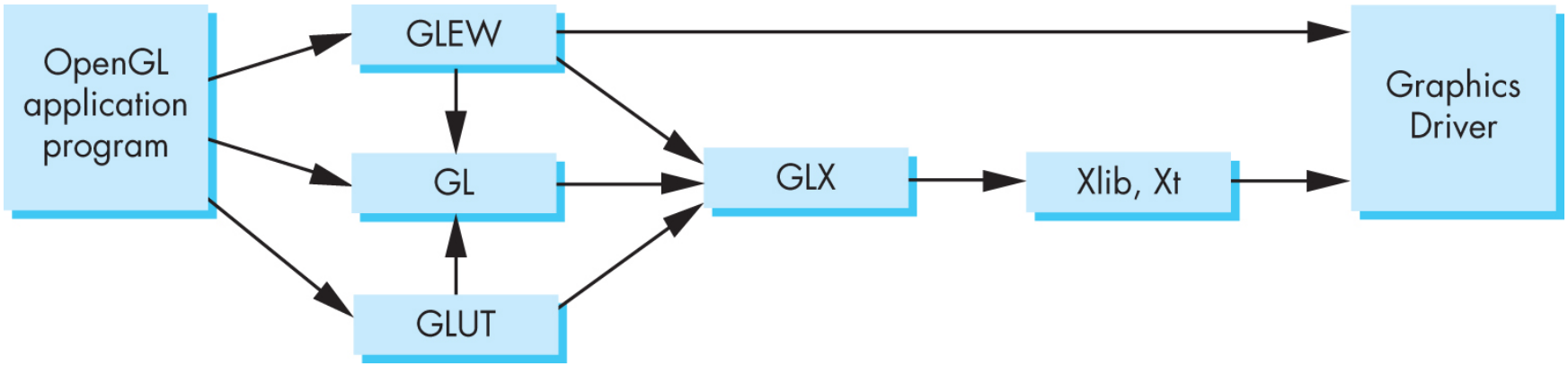
2D Applications

OpenGL is inherently 3D. Specifying points in 2D as p=(x,y) is equivalent to specifying them as p=(x,y,0).

If no modeling or viewing transformations are specified, vertex coordinates are assumed to be in **clip coordinates**.

Clip coordinates in the cube from (-1, -1, -1) to (+1, +1, +1) will be visible on the screen.





On MS Windows GLX, Xlib, Xt are replaced by wGL.

glew: *gl extension wrangler* loads new OpenGL functions and extensions

OpenGL Introduction

- Software interface to graphics hardware.
- Relatively low-level primitives (points, lines, triangles).
- Client-server design: The application is a client which sends commands to OpenGL (graphics server). The server and client need **not** be on the same computer.

7 kinds of graphics API functions

- Primitive functions: (OpenGL)
- Attribute functions: (OpenGL)
- Viewing functions: (OpenGL and matrix lib)
- Transformation functions: (OpenGL and matrix lib)
- Input functions: Keyboard, mouse, etc. (glut)
- Control functions: Interaction with windowing system (glut)
- Query functions: (OpenGL and glut)

The rest of this lecture will focus mainly on OpenGL functionality.

OpenGL Syntax

Function naming convention:

```
glUniform3f
```

- All OpenGL functions are prefixed 'gl'
- Function name : Uniform, ...
- Optional Size : 2, 3, 4 elements
- Optional Data type : float, double, byte, short, int, ...

The final letter 'v' indicates that a pointer is being passed

Example:

```
GLfloat color_array[] = {1.0f, 0.0f, 0.0f};
glUniform3fv(loc, 1, color_array);
```

OpenGL Syntax

Defined constants:

```
GL_COLOR_BUFFER_BIT
```

- All caps
- Start with GL
- Words separated by underscores

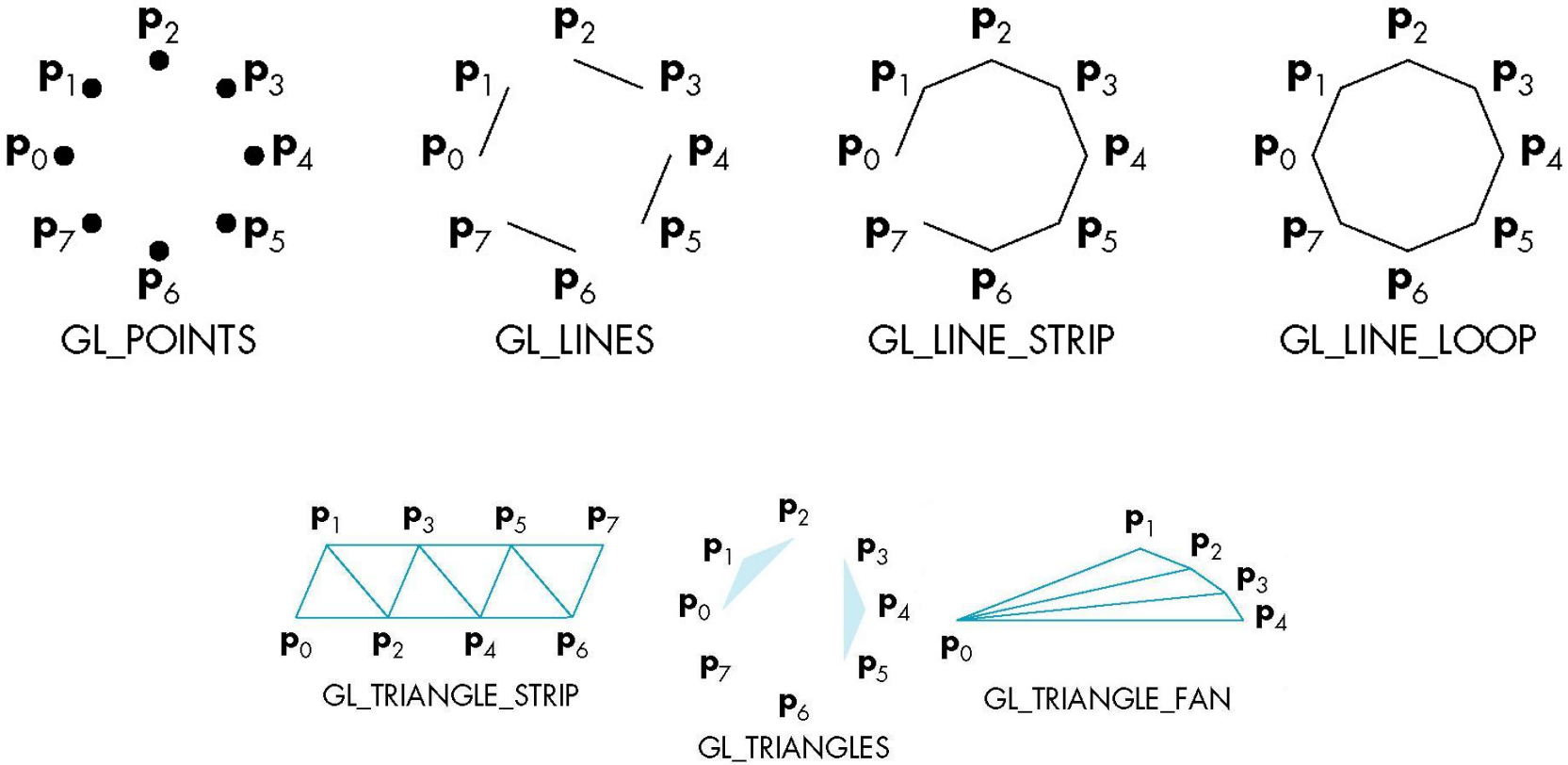
Defined types:

- GLbyte
- GLint
- GLfloat
- ...

Use of these type is optional. GLfloat == float

Geometric Primitives

Primitives are the basic building block for drawing objects



Specifying geometric Primitives

Delimited by glBegin / glEnd before OpenGL 3.1:

```
glBegin(primitive_type);
    glVertex*(...);
    ...
    glVertex*(...);
glEnd();
```

If you are familiar with this ”immediate mode” use of OpenGL, forget it.

Modern OpenGL has made this obsolete and we won’t be using it in this class. Homework using deprecated features such as these will be marked wrong.

(Can be made backward-compatible for the purposes of compiling legacy code.)

Specifying geometric Primitives

The new way to do this is to specify an array of data at once:

```
StoreSomeDataOnGPU(); // you write this function
glDrawArrays(primitive_type, 0, n);
```

Framebuffer Operations

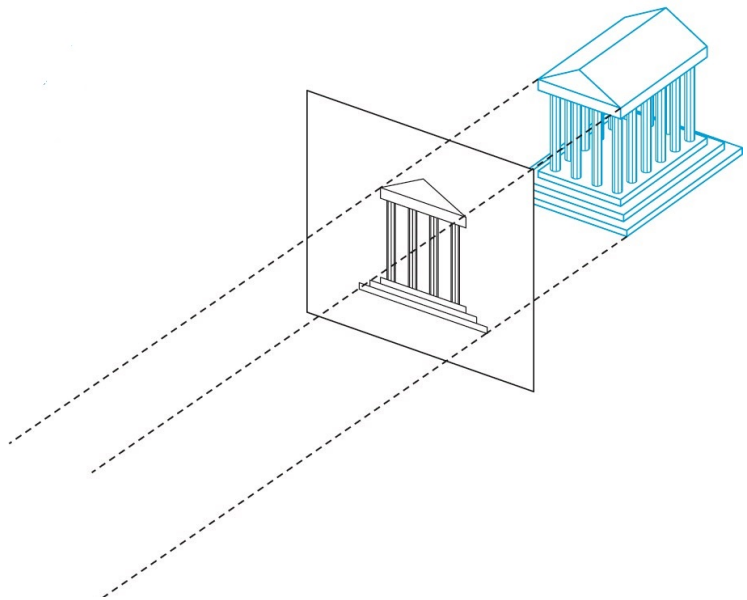
For framebuffer clearing

- `glClearColor(r, g, b, a);` : Sets the color to which the framebuffer will be cleared
- `glClear(GL_COLOR_BUFFER_BIT);` : Clears the framebuffer.

The default camera

By default the images generated by OpenGL will use an orthographic projection.

There will be no perspective foreshortening, so distant objects will look the same size as nearby objects.



OpenGL is a state machine

You will hear this phrase a lot in this class, the textbook, online forums...

- OpenGL state variables control rendering (e.g. clear color)
- States have default values. Change the value using various OpenGL functions.
- The values won't change unless you change them.

Good: pass fewer parameters in rendering function calls.
Bad: you may forget what the last state you set was. Defaults may not be dependable

Some state variables that control the rendering of primitives

- glPolygonMode(...); // draw as points (`GL_POINT`), lines (`GL_LINE`) or filled (`GL_FILL`)
- glPointSize(...); // in pixels
- glLineWidth(...); //in pixels

Other states

Set OpenGL state by enabling/disabling:

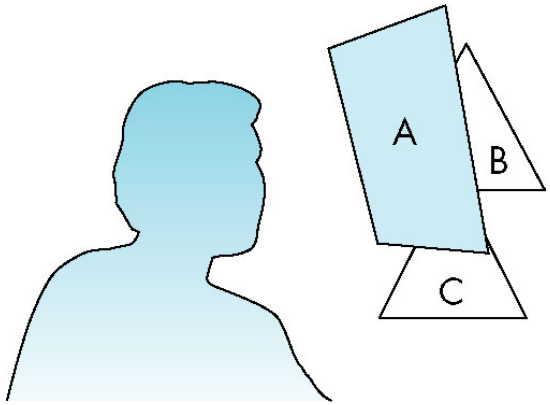
- glEnable(GLenum name); (ex. `GL_DEPTH_TEST`, `GL_BLEND`)
- glDisable(GLenum name);

Query OpenGL state

- glIsEnabled(GLenum name);
- glGetFloatv(GLenum name, pointer);

Hidden Surface Removal : The problem

By default OpenGL will draw primitives in the order that they are sent. So nearby objects may be obscured by objects farther away.



An object based solution:

Painter's algorithm - Draw the objects in back-to-front order.

- Need to sort. Data structures (BSP tree) can help.
- May need to split objects which intersect.

OpenGL can use different solution : the Z-buffer. This is **not** enabled by default.

Z-buffer (depth buffer) algorithm

- Visibility will be determined per **fragment**.
- This will require an additional buffer the same width and height as the color buffer.

For each fragment, compare fragment depth value to the frame buffer depth value.

- If fragment.z $<$ framebuffer.z then framebuffer.color = fragment.color, framebuffer.z = fragment.z
- else, discard the fragment.

Using the Z-buffer

To use the depth buffer in your application:

- Specify a frame buffer containing a z-buffer (glut).
- Enable the depth testing (OpenGL).
- Clear the depth buffer at the same time as the color buffer (OpenGL).

The corresponding function calls are:

- `glutInitDisplayMode(GLUT_DEPTH | ...)`
- `glEnable(GL_DEPTH_TEST)`
- `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`

Shaders: an important piece of state

Shaders specify how vertices and fragments should be processed.

In modern OpenGL there are no default shaders.

Simple Vertex Shader

Sometimes called ”pass-through” vertex shader

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

OpenGL will interpret `gl_Position` as being in clip-coordinates

Simple Fragment Shader

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

Assigns every fragment a red color.

Shader source code is stored as text

The author provides a helper function (declared in Angel.h) to load the shaders.

```
GLuint program;
program = InitShader("vsource.glsl", "fsource.glsl");
```

Your application loads, compiles, links and runs shaders at application run time.

Using Vertex Attributes in Shaders

Vertex shader:

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

Fragment shader:

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

Fragment color is determined by vertex attributes.

Using Vertex Attributes in Shaders

```
GLuint loc;
loc = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 4, GL_FLOAT,...);
```

- vColor is the name of the attribute in vertex shaders
- vColor is a 4 component float vector
- glVertexAttribPointer tells the shader where in memory to find the attributes.