Introduction to OpenGL & GLUT

CS 381 Computer Graphics

Lecture Slides

Tuesday, September 9, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

Review
Introduction to CG — Synthetic Camera Model

We base our 3-D viewing on a model similar to a camera.

- A point is chosen (the **center of projection**).
- Given an object in the scene, draw a line from it, through the center of projection, to the image.
  - The image lies in a plane, like film in a film camera, or the sensor array in a digital camera.
- Where this line hits the image is where the object appears in the image.

This model is similar to the way the human visual system works.

Review
Introduction to CG — Modeling & Rendering

Images of 3-D scenes are generated in two steps:

- Modeling
- Rendering

Modeling means producing a precise description of a scene, generally in terms of **graphics primitives**.

- Primitives are the pieces from which complex scenes are constructed. They may be points, lines, polygons, bitmapped images, various types of curves, etc.

Rendering means producing an image based on the model.

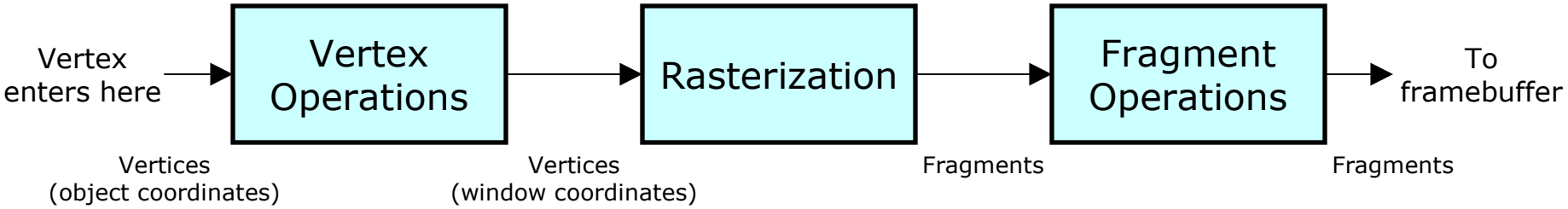
- Images are produced in a **framebuffer**.
- A modern framebuffer is a **raster**: a 2-D array of **pixels** (picture elements).

This class focuses on rendering.

Review
Introduction to CG — Rendering Pipeline [1/2]

In modern graphics architectures, there is a **rendering pipeline**.

- What is good about pipeline-style designs?
- Below is a (highly simplified!) illustration of our rendering pipeline.



Vertices enter.

- A **vertex** might be a corner of a polygon.

Rasterization converts geometry (polygons, etc.) to fragments.

- A **fragment** is a pixel-before-it-becomes-a-pixel.

Fragments leave.

At the end of the pipeline, values are stored in the framebuffer.

Later in the semester, we will fill in some of the details above.

Review
Introduction to CG — Rendering Pipeline [2/2]

In the 1990's, many options were added to graphics libraries, allowing programmers to customize the vertex and fragment operations.

- Far too many options!

In the last few years, graphics hardware became a user-programmable computer. Now we can write our own programs to do the vertex and fragment operations.

- Such a program is a **shader**.
- Shaders run on the graphics hardware, not the main processor!

Two major high-level **shading languages**:

- Cg (nVidia/Microsoft).
- GLSL (OpenGL ARB).

This is the shading language we will use.

Introduction to OpenGL & GLUT
API

API = Application Programming Interface.

- An API specifies how a program uses a library.
  - What calls/variables/types are available.
  - What these do.
- A well-specified API allows implementation details of a library to be changed without affecting programs that use the library.

The graphics API we will be using is called **OpenGL**.

- Sometimes we just say "GL".

Introduction to OpenGL & GLUT
Background — What is OpenGL?

Professional-quality 2-D & 3-D graphics API

- Developed by Silicon Graphics Inc. in 1992.
- Based on Iris GL, the SGI graphics library.

Available in a number of languages.

- We will use the C-language API.
- There is no C++-specific OpenGL API.

System-Independent API

- Same API under Windows, MacOS, various Unix flavors.
- Programmer does not need to know hardware details.
- Can get good performance from varying hardware.

An Open Standard

- A consortium of companies sets the standard.
- Anyone can implement the OpenGL API.
- Review board handles certification of implementations.

Introduction to OpenGL & GLUT
Background — What does OpenGL Do?

OpenGL provides a system-independent API for 2-D and 3-D graphics. It is primarily aimed at:

- Graphics involving free-form 3-D objects made up (at the lowest level) of polygons, lines, and points.
- Simple hidden-surface removal and transparency-handling algorithms are built-in.
- Scenes may be lit with various types of lights.
- Polygons may have images painted on them (**texturing**).

OpenGL does *not* excel at:

- Text generation (although it does include support for text).
- Page description or high-precision specification of images.
- The kind of graphics involved in a windowing system: window frames, controls, etc.

Introduction to OpenGL & GLUT
Background — Parts of OpenGL

OpenGL Itself

- The interface with the graphics hardware.
- Designed for efficient implementation in hardware. Particular OpenGL implementations may be partially or totally software.
- C/C++ header: **`<GL/gl.h>`**.

The OpenGL Utilities (GLU)

- Additional functions & types for various graphics operations.
- Designed to be implemented in software; calls GL.
- C/C++ header: **`<GL/glu.h>`**.

OpenGL Extensions

- Functionality added to base OpenGL.
- OpenGL specifies rules that extensions are to follow.
- We will use shading-language extensions (GLSL).

Background — GLUT

The OpenGL Utility Toolkit (GLUT)

- A utility library written by Mark Kilgard in the 1990s.
- It provides a system-independent interface to I/O including windows, pop-up menus, the mouse, and the keyboard.
- I do *not* consider GLUT to be appropriate for professional-quality work.
- We use it here because it does what we need in a system-independent way, and it is easy to learn.

The Design of OpenGL — Introduction

OpenGL is an API for rendering raster images of 2-D & 3-D scenes.

- So OpenGL's work ends when the completed image (or **frame**, in an animation context) is in the framebuffer.

We deal with OpenGL via function calls, known as **OpenGL commands**.

- No global variables.
- Most OpenGL functions have few parameters.
- But you make *lots* of function calls.
- No complex data types.
- OpenGL is **function-call intensive**.
  - Think: advantages/disadvantages.

The Design of OpenGL — Attributes & Primitives

OpenGL functions as a **state machine**.

There are three kinds of functions:

- Those that set state.
- Those that return state.
- Those that draw.

Drawing is done via **primitives**.

States are used to set **attributes** of those primitives.

So: all drawn objects are composed of primitives. The properties of these are attributes, which are determined by OpenGL states.

The Design of OpenGL — Example Code

To draw a red triangle with vertices (0,0), (1,0), (1,1):

```
glColor3d(0.9, 0.1, 0.1);  // red (setting an attribute)
glBegin(GL_TRIANGLES);     // starting a primitive
    glVertex2d(0., 0.);    // vertex data
    glVertex2d(1., 0.);
    glVertex2d(1., 1.);
glEnd();                   // ending the primitive
```

Note the indentation here. This is not required (of course), but I have found it helpful.

The Design of OpenGL — Naming Conventions [1/2]

OpenGL (C API)

- Functions
  - Begin with "**`gl`**", words capitalized & run together
  - Example: **`glClearColor`**
  - Can include type information. For example, the "**`2d`**" in "**`glVertex2d`**" indicates two parameters of type **`GLdouble`**.
- Constants
  - Begin with "**`GL`**", all upper-case, "_" between words
  - Example: **`GL_TRIANGLE_STRIP`**
- Types
  - Begin with "**`GL`**", *next* word not capitalized, all words run together
  - Example: **`GLdouble`**

The Design of OpenGL — Naming Conventions [2/2]

Related packages use similar conventions.

- GLU
  - Function: **`gluScaleImage`**
  - Constant: **`GLU_TESS_ERROR`**
  - Type: **`GLUtesselatorObj`**
- GLUT
  - Function: **`glutInitDisplayMode`**
  - Constant: **`GLUT_MIDDLE_BUTTON`**

The Design of OpenGL — Types [1/2]

OpenGL defines its own types, which have the same (minimum) precision on all systems. Some of these:

- **`GLint`**: at least 32-bit integer
- **`GLfloat`**: at least 32-bit floating-point
- **`GLdouble`**: at least 64-bit floating-point
- and others …

So, for example, **`GLdouble`** is *probably* the same as **`double`**, but may not be.

- Converting (say) a **`GLdouble`** to a **`double`** is fine.
- But be careful when tossing around **`GLdouble *`** and **`double *`**. (Why?)

The Design of OpenGL — Types [2/2]

Some OpenGL commands have several forms allowing for different types.

- For example, **`glVertex`*** can take 2, 3, or 4 parameters of many different types.
  - Function **`glVertex2d`** takes 2 parameters of type **`GLdouble`**.
  - Function **`glVertex3f`** takes 3 parameters of type **`GLfloat`**.
  - Function **`glVertex3fv`** ("v" for "vector") takes a single parameter of type **`GLfloat *`** (should be a pointer to an array of 3 **`GLfloat`**'s).
- The command **`glTranslate`*** always takes three parameters, but they may vary in type.
  - Function **`glTranslated`** takes 3 **`GLdouble`**'s.
  - Function **`glTranslatef`** takes 3 **`GLfloat`**'s.

GLUT Programs — Ideas

The structure of our programs is dictated by GLUT.

So:

Start with an already written program.

- Use the web or your own previous work.
- **Give credit where credit is due!**

GLUT handles overall flow of control.

You write functions to …

- Initialize OpenGL states and your own variables.
- Draw things.
- Handle *events* (mouse clicks, window changes, keypresses, etc.).
- Do something when nothing else happens.

These functions are called by GLUT, not you.

Introduction to OpenGL & GLUT
GLUT Programs — Examples

Look at **`sample2d.cpp`** & **`sample3d.cpp`**.

See **`sample2d.cpp`**, **`sample3d.cpp`**, *on the web page.*

GLUT Programs — Callbacks

Rules for callbacks (*display* in particular):

- You *never* call your callback functions. Only GLUT does that.
- The display function *only* does drawing (which may include GL state changes).
- No drawing is done outside the display function (but state changes may be done).

These rules are for this class.

- Later in life, you may want to break them.
- But think hard first; they're good rules.

OpenGL Primitives — Overview [1/2]

All rendering operations are composed of **primitives**.

- These need to be useful to the programmer and doable efficiently by the library & hardware.

We will now look at those OpenGL primitives that are handled via the **`glBegin-glEnd`** mechanism. There are ten of these; they consist of ways to draw:

- Points.
- Polylines (collections of line segments).
- Filled polygons.

Other primitive rendering operations are handled differently in OpenGL.

- Specifically, those involving screen-aligned rectangles: pixmaps, bitmaps, and screen-aligned rectangular polygons.

OpenGL has no circle/ellipse/curve primitives.

OpenGL Primitives — Overview [2/2]

The ten **`glBegin`**-style OpenGL Primitives

- Points (1 primitive)
  - **`GL_POINTS`**
- Polylines (3 primitives)
  - **`GL_LINES`**
  - **`GL_LINE_STRIP`**
  - **`GL_LINE_LOOP`**
- Filled Polygons (6 primitives)
  - Triangles
    - **`GL_TRIANGLES`**
    - **`GL_TRIANGLE_STRIP`**
    - **`GL_TRIANGLE_FAN`**
  - Quadrilaterals
    - **`GL_QUADS`**
    - **`GL_QUAD_STRIP`**
  - General Polygons
    - **`GL_POLYGON`**

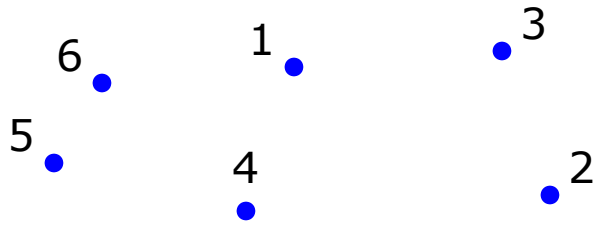
OpenGL Primitives — Points

A primitive is given a number of vertices (specified with **`glVertex…`**). Now we look at what the primitives do with the vertices they are given.

- Numbers indicate vertex ordering.
- Blue objects mark what is actually rendered.

Points

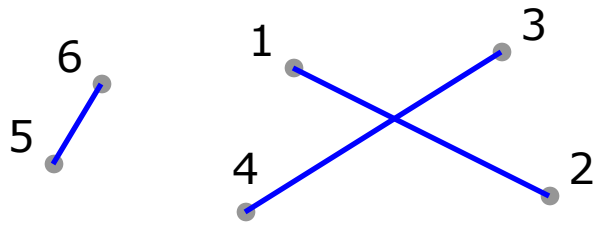
- **`GL_POINTS`**



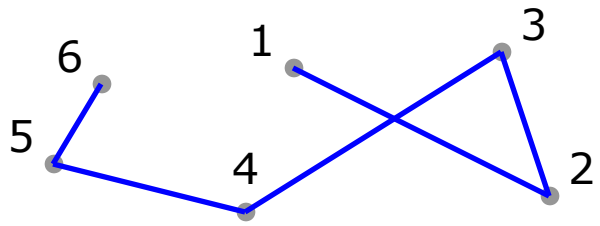
OpenGL Primitives — Polylines

Polylines

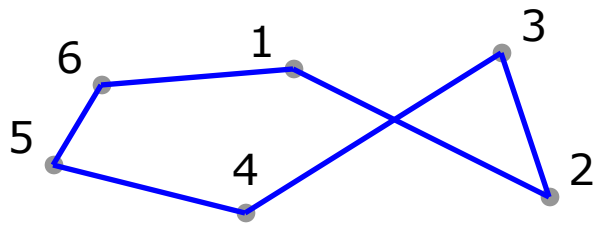
- **GL_LINES**



- **GL_LINE_STRIP**



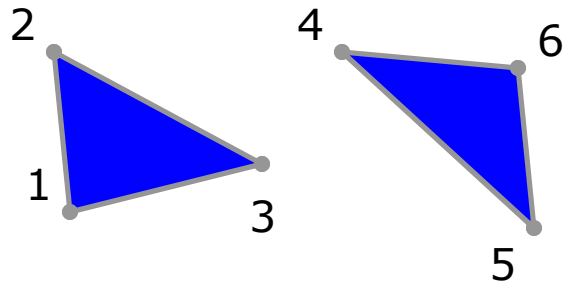
- **GL_LINE_LOOP**



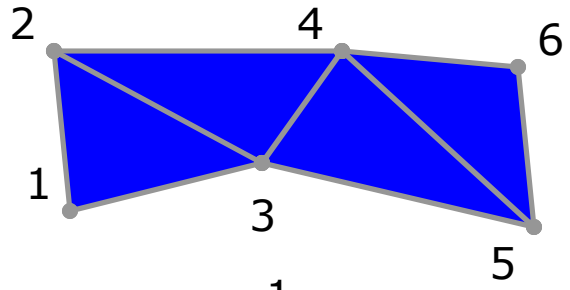
OpenGL Primitives — Polygons: Triangles

Polygons: Triangles

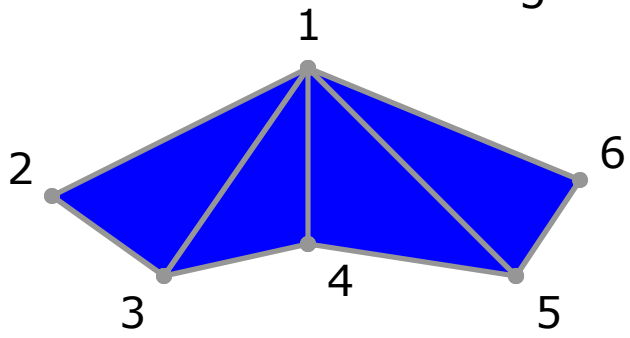
- **GL_TRIANGLES**
  - Clockwise or counterclockwise does not matter (yet).



- **GL_TRIANGLE_STRIP**



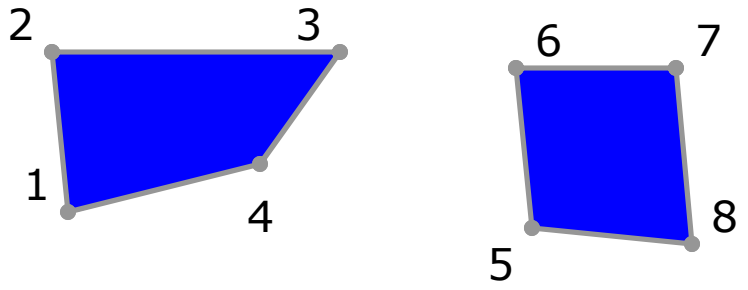
- **GL_TRIANGLE_FAN**



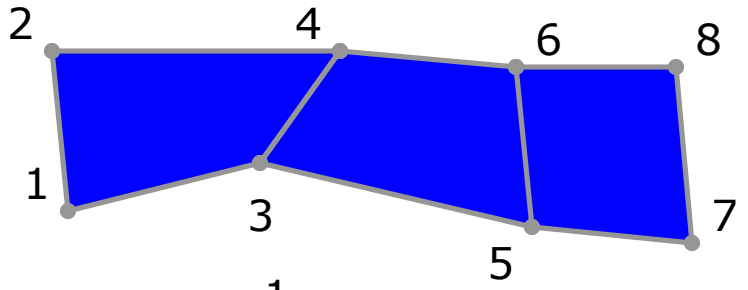
OpenGL Primitives — Polygons: Quads, General

Polygons: Quadrilaterals

- **`GL_QUADS`**
  - Clockwise or counterclockwise does not matter (yet).

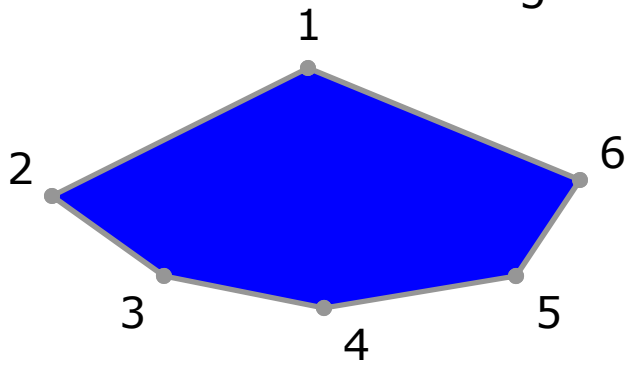


- **`GL_QUAD_STRIP`**
  - Note differences in vertex ordering!



Polygons: General

- **`GL_POLYGON`**



OpenGL Primitives — Restrictions

When drawing points, lines, and triangles, vertices can be in any positions you like.

Individual quadrilaterals and general polygons must be:

- Planar (this is easy in 2-D).
- Simple (no crossings, holes).
- Convex (bulging outward; no concavities).

Know the ten primitives! **Know** the associated vertex orderings!