The Basics of doing 3D Graphics

- Stuff you need to know to write programs
- Toolkit details best done by looking at code
  - And trying it yourself!
- See online tutorials (e.g. Survival Guide)
- See the red book
- Try to refresh the concepts behind using library
- Goal: get you to know enough to do Project

List of stuff you need to know

- Basics of Toolkits
- Dealing with a window
- Double buffering
- Drawing context
- Transformations / Coordinate Systems / Cameras
- 3D Viewing / Visibility (Z-Buffer)
- Polygon drawing
- Lighting
- Picking and UI

Basics of a toolkit

- OpenGL is for drawing graphics in a window
- Doesn't care where the window comes from
  - Need something to deal with Operating system
- Less good for text and widgets
- Use some toolkit to do windowing and UI support
  - FlTk – supports OpenGL well
  - Glut – simple, designed for doing OpenGL demos
  - Native windows – um, I can't comment

The Drawing Context

- OpenGL is stateful
  - Draw in the current window, current color, …
  - Contrast with stateless systems
draw(x1,y1,x2,y2)
draw(window, coordsys, x1, y1, x2, y2, color, pattern, …)
- Where is all that state kept?
  - Drawing Context
- Each window has its own state
  - Need mechanisms for keeping track of it
  - Making it the current state
  - FlTk does this for you (in draw, or with make_current)
- Beware! You can only draw with a current context

When does drawing happen

- Two different types of graphics toolkits
  - Immediate mode – stuff goes right to frame buffer
  - Retained mode – keep 3D objects on list, system draws all at once
- OpenGL supports both (usually immediate mode)
- What happens with a triangle
Double Buffering

- Double Buffering – independent of immediate/retained!
- Prevent from seeing partially drawn results
- (potentially) keep synced with screen refresh
- Draw into back buffer
- Swap-buffers
- FlTK will take care of this for you

When do I draw?

- When the window is “damaged”
- Periodically (animation / interaction)
- With FlTK:
  - It calls the draw function when needed
    - NEVER call it yourself
  - If you want to force a redraw, damage your window
    - It will be redrawn when appropriate

Where do I draw

- Screen coordinates – the main place everyone can agree
- OpenGL uses unit coordinates
  - Depth is -1 to 1 as well
- The Viewport
  - GL lets you limit things to a rectangular area of the screen
    - This is the only thing measured in pixels!
- Need to correct for aspect ration of screen

Getting my own coordinate system

- OpenGL only knows 1 coordinate system
  - The “Normalize Device Coordinates” - NDC
  - Viewport mapped to unit cube
    - There is actually 1 other coord system, but that’s a detail for lighting
- If you’re transformation is the identity, you get NDC
- All points transformed by the “current transformation”

OpenGL coordinate transforms

- OpenGL has 2 “current” transforms
  \[ n = P M x \]
  \[ n = \text{point in NDC} \quad x = \text{point in your coordinate system} \]
  \[ P = \text{projection matrix} \quad M = \text{Model View matrix} \]
  \[ P \text{ and } M \text{ are both stacks (although } P \text{ is a short stack} \]
- Why 2 matrices?
  - Esoteric detail of lighting
- Only the perspective transform goes into \( P \)
  - Unless you’re doing something weird
- \( M \) gives “camera coordinates”
  - Only lighting happens there in GL

Is OpenGL Post-Multiply?

- An internal detail – unless you look at the matrices
- Think of it as Post-Multiply
  - And if everything is being transposed, no big deal
- Only “load” is to load the transpose
  - OpenGL used to be pre-multiply, but since everyone else is post-multiply
How do I set the transform?

- Need to pick which matrix “stack”
  - Projection, ModelView
- Can either load, or post-multiply
  - Almost everything does a post-multiply
  - Except for the load operations
  - BEWARE: make sure to do a load identity first!

- Most matrix operations build a matrix and post-multiply it onto the “current” stack

Getting your coordinate systems

- Need things in camera coordinates
- Rotate and translate the world coordinates (and possibly scale)
- Think of placing and pointing the camera

Getting the camera scale?

- Projection does some scaling (by Z)
- Projection puts eye at z?
- Projection puts near clipping plane at -1, far plane at 1
- Use OpenGL’s projection matrix
- Field of view/aspect ratio

Moving coordinate systems

- Multiplying matrix means changing the coordinate system
- Or think about it as things closest to the object go first

Your own coordinate system

- Draw your triangle…
  - On a piece of paper
  - In your hand
  - When you’re on a platform
  - On a crane
- Build transforms!
  - Camera->world
  - World->crane
  - Crane->top of crane
  - Crane->platform
  - Platform->person
  - Person->arm
  - Arm->paper . . .

Convenient ways to make transforms

- Projection
  - gluFrustum, glPerspective
- Matrix handling
  - Load, get, pushmatrix, popmatrix
  - Rarely load anything but the identity
Actually drawing

- Begin / end blocks of points
- Send each point by itself (or as an array)
- Uniformity in how you draw different things
  - Lines
  - Triangles
  - Strips of triangles
  - Quads
- Things are drawn in the “current” state
- Color, line style, …

What color are things?

- Turn off lighting – and say colors directly
- Turn on lighting – and let the games begin!
- Idea: color of object is affected by lights
  - Need some light to see things
  - Direction of light affects how things look
  - Say where the lights are, how strong they are
  - What the reflectance of the surfaces are
- A whole topic for days in this class

Normal Vectors

- Assign per-vertex or per-triangle
- Unit vector towards the “outside”
- Will be very useful for lighting, so get in the habit

What happens to stuff off the screen?

- Clipping
  - Things get chopped by a plane
  - Each side of the viewing volume
  - Other planes as well – if you want
- Important to do correctly and efficiently
- A lot of work into the methods – but really boring

Visibility

- A topic for later in the class:
  How to get objects to occlude each other
- Give polygons in any order (even back ones last)
- Use a Z-Buffer to store depth at each pixel
- Things that can go wrong:
  - Near and far planes DO matter
  - Backface culling and other tricks can be problematic
  - You may need to turn the Z-buffer on
  - Don’t forget to clear the Z-Buffer!

So, I got a black screen…

- Celebrate – you’ve gotten a window, and that’s step 1!
- Are you drawing at the right time?
- Do you have a drawing context?
- Are you drawing objects?
- Is the camera pointing at them?
- Are they getting mapped to the screen?
- Is something occluding them?
- Are they in the view volume?
- Are they lit correctly?
- And a zillion other things that can go wrong…