Announcements

• Reminder: Homework 1 is due Thursday
• Questions?
• Class next Tuesday (9/18) will be held in SN 014

Last Time

• Extended transformations to 3D
• Introduced some principles of computer animation
  • Lasseter’s “Principles of Traditional Animation Applied to 3D Computer Graphics”
  • How to create “The Illusion of Life”

Today

• Learning how to program in OpenGL
  • OpenGL
  • C/C++
  • GLUT, FLTK, Cocoa

OpenGL in Java

• I have never used Java for OpenGL programming
  • I can’t be much help in getting it set up
• If you really want to try using OpenGL in Java
  • The JOGL API Project
    • https://jogl.dev.java.net/
  • Go there and follow the instructions
**What is OpenGL?**

- The standard specification defining an API that interfaces with the computer's graphics system
- Cross-language
- Cross-platform
- Vendor-independent
- Competes with DirectX on Windows

**The Rendering Datanpath**

- Computer Software
- OpenGL API
- Graphics Driver
- PCI Express Bus
- Frame Buffer
- High-Speed, Guaranteed Latency Interconnect
- Scanout Logic
- Display

**The “Red Book”**

*An older version is available (free!) online: [http://fly.cc.fer.hr/~unreal/theredbook](http://fly.cc.fer.hr/~unreal/theredbook)*

**Online Resources**

- [http://nehe.gamedev.net](http://nehe.gamedev.net)
- [http://www.opengl.org](http://www.opengl.org)

**The Camera Analogy**

**OpenGL’s World**

- Camera Description
- Scene Object (Intrinsic & Extrinsic)
- What To Display On Screen
- Code Specified Parameters
Contexts and Viewports?

• Each OpenGL application creates a context to issue rendering commands to
• The application must also define a viewport, a region of pixels on the screen that can see the context
• Can be
  • Part of a window
  • An entire window
  • The whole screen

OpenGL as a State Machine

• OpenGL is designed as a finite state machine
• Graphics system is a “black box”
• Most functions change the state of the machine
• One function runs input through the machine

OpenGL State

• Some attributes of the OpenGL state
  • Current color
  • Camera properties (location, orientation, field of view, etc.)
  • Lighting model (flat, smooth, etc.)
  • Type of primitive being drawn
  • And many more...

Our First OpenGL Code

```c
glClearColor(0.0, 0.0, 0.0, 0.0); glClear(GL_COLOR_BUFFER_BIT); glColor3f(1.0, 1.0, 1.0); glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0); glBegin(GL_QUADS); glVertex2f(-0.5, -0.5); glVertex2f(-0.5, 0.5); glVertex2f(0.5, 0.5); glVertex2f(0.5, -0.5); glEnd(); glFlush();
```

OpenGL Input

• All inputs (i.e. geometry) to an OpenGL context are defined as vertex lists
• glVertex*
  • * = nt OR ntv
  • n - number (2, 3, 4)
  • t - type (i = integer, f = float, etc.)
  • v - vector

OpenGL Types

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Data Type</th>
<th>Typical Corresponding C-Language Type</th>
<th>OpenGL Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8-bit integer</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>a</td>
<td>16-bit integer</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>long</td>
<td>GLint, GLsizei</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point</td>
<td>float</td>
<td>GLfloat, GLclampf</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point</td>
<td>double</td>
<td>GLdouble, GLclampd</td>
</tr>
<tr>
<td>ub</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
<td>GLubyte, GLboolean</td>
</tr>
<tr>
<td>ua</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
<td>GLuint</td>
</tr>
<tr>
<td>ul</td>
<td>32-bit unsigned integer</td>
<td>unsigned long</td>
<td>GLuint, GLenum, GLvoid</td>
</tr>
</tbody>
</table>

OpenGL as a State Machine

• OpenGL is designed as a finite state machine
• Graphics system is a “black box”
• Most functions change the state of the machine
• One function runs input through the machine
OpenGL Input

- Examples:
  - `glVertex2i(5, 4);`
    - Specifies a vertex at location (5, 4) on the $z = 0$ plane
    - "2" tells the system to expect a 2-vector (a vertex defined in 2D)
    - "i" tells the system that the vertex will have integer locations

- More examples:
  - `glVertex3f(.25, .25, .5);`
  - `double vertex[3] = {1.0, .33, 3.14159}; glVertex3dv(vertex);`
    - "v" tells the system to expect the coordinate list in a single data structure, instead of a list of n numbers

OpenGL Primitive Types

- All geometry is specified by vertex lists
  - But can draw multiple types of things
    - Points
    - Lines
    - Triangles
    - etc.
- The different things the system knows how to draw are the system primitives

Specifying the OpenGL Primitive Type

- `glBegin(primitiveType);` // A list of glVertex* calls goes here
  // ...
- `glEnd();`

- primitiveType can be any of several things
  - See the next slide

OpenGL Primitives Example

```java
glBegin(GL_POLYGON);
glVertex2f(0.0, 0.0);
glVertex2f(0.0, 3.0);
glVertex2f(3.0, 3.0);
glVertex2f(4.0, 1.5);
glVertex2f(3.0, 0.0);
glEnd();
```
Color in OpenGL

- Monitors can have different color resolutions
  - Black & white
  - 256 color
  - 16.8M color
- Want to specify color in a device-independent way

Color in OpenGL

- `glColor4f(r, g, b, a);`
- `r, g, b, a` - should all be between \([0.0, 1.0]\)
- `r, g, b` - amounts of red, green, and blue
- `a` - alpha
  - Defines how opaque a primitive is
  - \(0.0\) = totally transparent, \(1.0\) = totally opaque
  - Usually want `a = 1.0`

Finishing Up Your OpenGL Program

- OpenGL commands are not executed immediately
  - They are put into a command buffer that gets fed to the hardware
- When you're done drawing, need to send the commands to the graphics hardware
  - `glFlush()` or `glFinish()`

glFlush vs. glFinish

- `glFlush();`
  - Forces all issued commands to begin execution
  - Returns immediately (asynchronous)
- `glFinish();`
  - Forces all issued commands to execute
  - Does not return until execution is complete (synchronous)

Matrices in OpenGL

- Vertices are transformed by 2 matrices:
  - ModelView
    - Maps 3D to 3D
    - Transforms vertices from object coordinates to eye coordinates
  - Projection
    - Maps 3D to 2D (sort of)
    - Transforms vertices from eye coordinates to clip coordinates

The ModelView Matrix

- In OpenGL, the viewing and modeling transforms are combined into a single matrix - the modelview matrix
  - Viewing Transform - positioning the camera
  - Modeling Transform - positioning the object
- Why?
  - Consider how you would "translate" a fixed object with a real camera
Placing the Camera

- `gluLookAt(eyeX, eyeY, eyeZ, midX, midY, midZ, upX, upY, upZ)`
  - `(eyeX, eyeY, eyeZ)` - location of the viewpoint
  - `(midX, midY, midZ)` - location of a point on the line of sight
  - `(upX, upY, upZ)` - direction of the up vector
  - By default the camera is at the origin, looking down negative z, and the up vector is the positive y axis

WARNING! OpenGL Matrices

- In C/C++, we are used to row-major matrices
- In OpenGL, matrices are specified in column-major order

![Row-Major Order](image1.png) ![Column-Major Order](image2.png)

Using OpenGL Matrices

- Use the following function to specify which matrix you are changing:
  - `glMatrixMode(whichMatrix);`
    - `whichMatrix = GL_PROJECTION | GL_MODELVIEW`
    - To guarantee a “fresh start”, use `glLoadIdentity();`
      - Loads the identity matrix into the active matrix

Using OpenGL Matrices

- To load a user-defined matrix into the current matrix:
  - `glLoadMatrixf(m)`
- To multiply the current matrix by a user defined matrix
  - `glMultMatrixf(m)`
- SUGGESTION: To avoid row-/column-major confusion, specify matrices as `m[16]` instead of `m[4][4]`

Transforms in OpenGL

- OpenGL uses 4x4 matrices for all its transforms
  - But you don’t have to build them all by hand!
- `glRotatef(angle, x, y, z)`
  - Rotates counter-clockwise by `angle` degrees about the vector `(x, y, z)`
- `glTranslatef(x, y, z)`
- `glScalef(x, y, z)`

WARNING! Order of Transforms

- In OpenGL, the last transform in a list is applied FIRST
  - Think back to right-multiplication of transforms
- Example:
  - `glRotatef(45.0f, 0.0f, 0.0f, 0.0f);` `glTranslatef(10.0f, 0.0f, 0.0f);`
  - Translates first, then rotates
**Projection Transforms**

- The projection matrix defines the viewing volume
- Used for 2 things:
  - Projects an object onto the screen
  - Determines how objects are clipped
- The viewpoint (the location of the “camera”) that we’ve been talking about is at one end of the viewing volume

**Perspective Projection**

- The most noticeable effect of perspective projection is foreshortening
- OpenGL provides several functions to define a viewing frustum
  - `glFrustum(...)`
  - `gluPerspective(...)`

**gluPerspective**

- This GL Utility Library function provides a more intuitive way (I think) to define a frustum
  - `gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far)`
  - `fovy` - field of view in y (in degrees)
  - `aspect` - aspect ratio (width / height)
  - `near` and `far` should always be positive

**glFrustum**

- `glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`
- `(left, bottom, -near) and (right, top, -near)` are the bottom-left and top-right corners of the near clip plane
- `far` is the distance to the far clip plane

**Orthographic Projection**

- With orthographic projection, there is no foreshortening
- Distance from the camera does not change apparent size
- Again, there are several functions that can define an orthographic projection
  - `glOrtho()`
  - `gluOrtho2D()`
glOrtho
- glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)
  - Arguments are the same as glPerspective()
  - (left, bottom, -near) and (right, top, -near) are the bottom-left and top-right corners of the near clip plane
  - near and far can be any values, but they should not be the same

 gluOrtho2D
- This GL Utility Library function provides a more intuitive way (I think) to define a frustum
- gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)
  - (left, bottom) and (right, top) define the (x, y) coordinates of the bottom-left and top-right corners of the clipping region
  - Automatically clips to between -1.0 and 1.0 in z

Viewport
- The viewport is the part of the window your drawing is displayed to
  - By default, the viewport is the entire window
  - Modifying the viewport is analogous to changing the size of the final picture
    - From the camera analogy
    - Can have multiple viewports in the same window for a split-screen effect

Setting the Viewport
- glViewport(int x, int y, int width, int height)
  - (x, y) is the location of the origin (lower-left) within the window
  - (width, height) is the size of the viewport
  - The aspect ratio of the viewport should be the same as that of the viewing volume