Intro to OpenGL III

Don Fussell
Computer Science Department
The University of Texas at Austin
Where are we?

- Continuing the OpenGL basic pipeline
OpenGL API Example

```c
glShadeModel(GL_SMOOTH); // smooth color interpolation
glEnable(GL_DEPTH_TEST); // enable hidden surface removal

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES); // every 3 vertexes makes a triangle
glColor4ub(255, 0, 0, 255); // RGBA=(1,0,0,100%)
glVertex3f(-0.8, 0.8, 0.3); // XYZ=(-8/10,8/10,3/10)
glColor4ub(0, 255, 0, 255); // RGBA=(0,1,0,100%)
glVertex3f(0.8, 0.8, -0.2); // XYZ=(8/10,8/10,-2/10)
glColor4ub(0, 0, 255, 255); // RGBA=(0,0,1,100%)
glVertex3f(0.0, -0.8, -0.2); // XYZ=(0,-8/10,-2/10)
glEnd();
```
#include <GL/glut.h>  // includes necessary OpenGL headers

void display() {
    // << insert code on prior slide here >>
    glutSwapBuffers();
}

void main(int argc, char **argv) {
    // request double-buffered color window with depth buffer
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInit(&argc, argv);
    glutCreateWindow("simple triangle");
    glutDisplayFunc(display); // function to render window
    glutMainLoop();
}
Last time we covered how rasterization is done. Lots of linear interpolation in special-purpose hardware, so it’s fast.
We also talked a little about fragment shading, that is, the simple interpolated color shading that can be done in the rasterizer. There’s much more to come.
Color Interpolation

- Our simple triangle is drawn with smooth color interpolation
  - Recall: `glShadeModel(GL_SMOOTH)`
- How is color interpolated?
  - Think of a plane equation to compute each color component (say *red*) as a function of \((x,y)\)
    - Just done for samples positions within the triangle

\[
"redness" = A_{red}x + B_{red}y + C_{red}
\]
Barycentric Coordinates

\[
\begin{align*}
\text{Area}(PMN) & \quad \alpha \\
\text{Area}(LMN) & \\
\text{Area}(LPN) & \quad \beta \\
\text{Area}(LMN) & \\
\text{Area}(LMP) & \quad \gamma \\
\text{Area}(LMN) & \\
\end{align*}
\]

**Note:** \( \alpha + \beta + \gamma = 0 \) by construction

\[
\text{attribute}(P) = \alpha \times \text{attribute}(L) + \beta \times \text{attribute}(M) + \gamma \times \text{attribute}(N)
\]
Top GPUs can setup over a billion triangles per second for rasterization. Triangle setup & rasterization is just one of the (many, many) computation steps in GPU rendering.
A Simplified Graphics Pipeline

Application

Vertex batching & assembly

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing

Color update

Depth buffer

Framebuffer

Ensure closer objects obscure (hide) more distant objects
Interpolating Window Space Z

- Plane equation coefficients (A, B, C) generated by multiplying inverse matrix by vector of per-vertex attributes

\[
\begin{bmatrix}
L_x & L_y & 1 \\
M_x & M_y & 1 \\
N_x & N_y & 1
\end{bmatrix}
^{-1}
\begin{bmatrix}
L_z \\
M_z \\
N_z
\end{bmatrix}
= 
\begin{bmatrix}
A_z \\
B_z \\
C_z
\end{bmatrix}
\]
Assume `glViewport(0,0,500,500)` has been called
And `glDepthRange(0,1)`

$L = (50, 450, 0.65)$
$N = (450, 450, 0.4)$
$M = (250, 50, 0.4)$

$L_z = 0.65$
$M_z = 0.40$
$N_z = 0.40$
Interpolating Window Space Z

- Substitute per-vertex \((x,y)\) and \(Z\) values for the L, M, and N vertexes

\[
\begin{bmatrix}
50 & 450 & 1 \\
250 & 50 & 1 \\
450 & 450 & 1 \\
\end{bmatrix}
\begin{bmatrix}
0.65 \\
0.4 \\
0.4 \\
\end{bmatrix}
= \begin{bmatrix}
A_z \\
B_z \\
C_z \\
\end{bmatrix}
\]

\(A_z = -0.000625\)
\(B_z = 0.0003125\)
\(C_z = 0.540625\)

Complete Z plane equation

\[
Z(x,y) = -0.000625 \times x + 0.0003125 \times y + 0.540625
\]
**Depth Buffer Algorithm**

- **Simple, brute force**
  - Every color sample in framebuffer has corresponding depth sample
  - Discrete, solves occlusion in pixel space
  - Memory intensive, but fast for hardware

- **Basic algorithm**
  - Clear the depth buffer to its “maximum far” value (generally 1.0)
  - Interpolate fragment’s Z
  - Read fragment’s corresponding depth buffer sample Z value
  - If interpolated Z is less than (closer) than Z from depth buffer
    - Then replace the depth buffer Z with the fragment’s Z
    - And also allow the fragment’s shaded color to update the corresponding color value in color buffer
  - Otherwise discard fragment
    - Do **not** update depth or color buffer
Depth Buffer Example

- Fragment gets rasterized
- Fragment’s Z value is interpolated
  - Resulting Z value is 0.65
- Read the corresponding pixel’s Z value
  - Reads the value 0.8
- Evaluate depth function
  - 0.65 \texttt{GL\_LESS} 0.8 is \texttt{true}
  - So 0.65 replaces 0.8 in the depth buffer

- Second primitive rasterizes same pixel
- Fragment’s Z value is interpolated
  - Resulting Z value is 0.72
- Read the corresponding pixel’s Z value
  - Reads the value 0.65
- Evaluate depth function
  - 0.72 \texttt{GL\_LESS} 0.65 is \texttt{false}
  - So the fragment’s depth value and color value are discarded
Depth Test Operation

- Fragment depth: 0.65
- Pixel depth: 0.8
- Time: 0.8

0.65 < 0.8 is true
0.65 < 0.8 is false

Depth test passes
Depth test fails
Depth Buffer Visualized

Depth-tested 3D scene

Z or depth values
white = 1.0 (far), black = 0.0 (near)
OpenGL API for Depth Testing

- Simple to use
  - Most applications just “enable” depth testing and hidden surfaces are removed
  - Enable it: `glEnable(GL_DEPTH_TEST)`
    - Disabled by default
    - **Must** have depth buffer allocated for it to work
      - **Example**: `glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)`

- More control
  - Clearing the depth buffer
    - `glClear(GL_DEPTH_BUFFER_BIT | otherBits)`
    - `glClearDepth(zvalue)`
      - Initial value is 1.0, the maximum Z value in the depth buffer
  - `glDepthFunc(zfunc)`
    - `zfunc` is one of `GL_LESS, GL_GREATER, GL_EQUAL, GL_GEQUAL, GL_LEQUAL, GL_ALWAYS, GL_NEVER, GL_NOTEQUAL`
    - Initial value is `GL_LESS`
  - `glDepthMask(boolean)`
    - True means write depth value if depth test passes; if false, don’t write
    - Initial value is `GL_TRUE`
  - `glDepthRange`
    - Maps NDC Z values to window-space Z values
    - Initially [0,1], mapping to the entire available depth range
Not Just for View Occlusion
Depth Buffers also Useful for Shadow Generation

Without Shadows
Projected Shadow Map
With Shadows

Light’s View
Light’s View Depth
A Simplified Graphics Pipeline

Application

Vertex batching & assembly

Clipping

NDC to window space

Rasterization

Fragment shading

Depth testing

Color update

Depth buffer

Framebuffer
Next Lecture

- Graphics Math, Transforms
  - Interpolation, vector math, and number representations for computer graphics
Programming tips

- 3D graphics, whether OpenGL or Direct3D or any other API, can be frustrating
- You write a bunch of code and the result is

Nothing but black window; where did your rendering go??
Things to Try

- Set your clear color to something other than black!
  - It is easy to draw things black accidentally so don’t make black the clear color
  - But black is the initial clear color
- Did you draw something for one frame, but the next frame draws nothing?
  - Are you using depth buffering? Did you forget to clear the depth buffer?
- Remember there are near and far clip planes so clipping in Z, not just X & Y
- Have you checked for glGetError?
  - Call glGetError once per frame while debugging so you can see errors that occur
  - For release code, take out the glGetError calls
- Not sure what state you are in?
  - Use glGetIntegerv or glGetFloatv or other query functions to make sure that OpenGL’s state is what you think it is
- Use glutSwapBuffers to flush your rendering and show to the visible window
  - Likewise glFinish makes sure all pending commands have finished
- Try reading
  - http://www.slideshare.net/Mark_Kilgard/avoiding-19-common-opengl-pitfalls
  - This is well worth the time wasted debugging a problem that could be avoided
Thanks

- Presentation approach and figures from
  - David Luebke [2003]
  - Brandon Lloyd [2007]
  - *Geometric Algebra for Computer Science* [Dorst, Fontijne, Mann]
  - via Mark Kilgard