Today

Real-time graphics with OpenGL

Hidden Surface Removal

Near objects hide (occlude) further objects

Hidden Surface Removal

- Color at a rendered pixel depends primarily on the nearest object at that point
- Naïve solution: Sort and render objects back to front (painter’s algorithm)
  - Inefficient
  - Not as easy as it sounds!
    - Is the man behind the wall or the wall behind the man?

Better Solutions

- Raycasting/Raytracing: Trace a ray through the pixel, see which object is hit first
- Z-Buffer
  - Draw objects one-by-one in any order
  - At each pixel, store closest depth value seen so far
    - Z axis is usually assumed to lie along the depth direction, hence this image of depth values is called the z-buffer
  - At pixel $p$, let an object have color $c$ and depth $d$
    - If $d < \text{old depth at } p$
      - new depth at $p = d$
      - new color at $p = c$
Isn’t raycasting simpler?
- Z-buffer algorithms have traditionally been easy to accelerate in hardware
  - No need for complicated data structures
  - Parallelizable in object space: Every object is drawn (nearly) independently of the others
    - Useful when scene can be divided into lots of small components, e.g. triangles
- But raytracing can be accelerated too!
  - Requires more sophisticated hardware
  - Different parallelization characteristics
    - Often better than z-buffers when there’s lots of occlusion
  - Gaining popularity in recent times

Limitations of Z-Buffers
- Hard to do
  - reflections
  - refractions
  - shadows
- In fact, the only thing that’s easy to do is diffuse shading (ADS/Phong) with direct lighting
  - This was good enough for most games for a long time
  - 99.99% of all 3D games use z-buffers, accelerated to ridiculous speeds by graphics processing units (GPUs)

Standard Z-Buffer Based APIs
- Direct3D
  - Windows-only
  - [http://www.microsoft.com/directx](http://www.microsoft.com/directx)
- OpenGL
  - Windows, OS X, Linux, ...
    - ... which is why we’ll look only at OpenGL in this course
  - [http://www.opengl.org](http://www.opengl.org)

Basic OpenGL
- Represent object surface as set of primitive shapes
  - Points
  - Lines
  - Triangles
  - Quadrilateral
- This process is called tessellation
- Draw primitives one by one
  - Batched and parallelized in hardware
- Let the z-buffer figure out which primitive determines the color at each pixel
Tessellating a Sphere with Triangles

A Tessellated Teapot

Tessellated Animals

Tessellated Terrain

Tessellation

- Difficult to get right
  - Primitives must be evenly distributed
  - Primitives must not have awkward shapes (e.g. very “skinny” triangles)
  - This is important not just for display but even more so for physics simulation/finite element methods
- Many sophisticated algorithms exist
  - Often take equations of curved patches as input
  - We won’t cover them in this course
  - In assignments we’ll work with pre-tessellated models

Drawing Triangles in OpenGL

```cpp
glBegin(GL_TRIANGLES);
foreach triangle in object
{
  // Tell OpenGL the normal and color of the triangle
  // Send the 3 vertex positions
}
gEnd();
```

**Note:**
- Every collection of primitives must be placed between a `glBegin`/`glEnd` block
- Every three successive vertices in the block defines a triangle
- Instead of GL_TRIANGLES we could use GL_POINTS (every vertex is a point), GL_LINES (every 2 vertices defines a line), GL_QUADS (every 4 vertices defines a quad) etc.
Drawing Triangles in OpenGL

```cpp
glBegin(GL_TRIANGLES);
foreach triangle in object {
    glNormal3f(0.58f, 0.58f, 0.58f); // (nx, ny, nz)
    glColor3f(1.0f, 0.0f, 0.0f); // (R = 1, G = 0, B = 0)
    glVertex3f(1.0f, 0.0f, 0.0f); // (x, y, z)
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 1.0f);
}
glEnd();
```

**Note:**
- We set the normal and color per triangle (they can actually be set anywhere, anytime, and apply to all subsequent vertices)
- We set the positions per vertex

What’s this …3f business?
- glVertex has variants glVertex3f, glVertex3d
- The first takes 3 float arguments (x, y, z)
- The second takes 3 double arguments
- OpenGL also has functions with a …3i suffix – these obviously take 3 integers
- There’s also glVertex2f
- z is assumed to be 0
- ... and glVertex4f
- Last argument is homogenous coordinate h, which is otherwise assumed to be 1
- Similarly glColor4f is used to specify (R, G, B, α)

Transforming Objects

- Let’s see a simple example first...
  ```cpp
gLoadMatrixf(M); // M is a 4x4 matrix stored in column-major form
  // Draw the object using glBegin/glEnd
  
  **Note:**
  - The object is transformed by M before it is drawn
    - Each vertex v becomes M * v
  - M is column-major!
    - Array of 16 numbers: first column, then second column, ...
  - M is column-major!!
  - Did we mention M is column-major??
```

Composing Transformations

- Just specify the matrices to be composed one after the other
  ```cpp
gLoadMatrixf(A); // Initial matrix
  gMultMatrixf(B); // Note: MultMatrix, not LoadMatrix
gMultMatrixf(C);
  ...
  // Draw the object using glBegin/glEnd
  
  The object is transformed by A * B * C
  - Each vertex v becomes A * B * C * v
  **Note:** Transforms are applied last-to-first!
```

OpenGL Convenience Functions

- gLoadIdentity() ≡ gLoadMatrixf(<identity matrix>)
- gTranslatef(tx, ty, tz) ≡ gMultMatrixf(T)
  - T is a matrix that translates by (t_x, t_y, t_z)
- gRotatef(angle, x, y, z) ≡ gMultMatrixf(R)
  - R is a matrix that rotates by angle degrees around the axis (x, y, z)
- gScalef(sx, sy, sz) ≡ gMultMatrixf(S)
  - S is a matrix that scales by s_x along x, s_y along y and s_z along z
  - (All the functions have …d versions, of course)

Transforming Objects

- A more complicated example:
  ```cpp
gMatrixMode(GL_MODELVIEW);
gPushMatrix();
gMultMatrixf(M);
  // Draw the object using glBegin/glEnd
gPopMatrix();
```
  **Questions:**
  - Why all the pushing/popping?
  - What’s with thisMatrixMode business?
Hierarchical Modeling

- Graphics systems maintain a current transformation matrix (CTM)
  - All geometry is transformed by the CTM
  - CTM defines object space in which geometry is specified
  - Transformation commands are concatenated onto the CTM (glMultMatrix). The last one added is applied first:
    - CTM = CTM * T
  - The CTM is reset with glLoadMatrix
- Graphics systems also maintain transformation stack
  - The CTM can be pushed onto the stack (glPushMatrix)
  - The CTM can be restored from the stack (glPopMatrix)

Example: Articulated Robot

Example: Articulated Robot (OpenGL)

Recap

- Z-buffer to detect visible surfaces
- Surfaces tessellated into simple primitives
- Draw primitives with glBegin/glEnd blocks
  - glVertex, glNormal, glColor
- Nested transform blocks
  - glPushMatrix, glPopMatrix, glLoadMatrix, glMultMatrix

(We’ll address the glMatrixMode business a little later)

Drawing Triangles

- Problem: Given triangle Δ, color the pixels that it covers
  - This is called rasterization
- Two-step solution:
  - Project the triangle to screen space
  - Compute the pixels covered by the projection

OpenGL Pixel Coordinates

The pixel grid is called the framebuffer
OpenGL Pixel Coordinates
Pixel centers are at **half-integer** coordinates

Rasterization Rules: Area Primitives
Output *fragment* if pixel center is inside area

What do we mean by “combine”?
- Typically, we test the *fragment depth* against the z-buffer and replace the existing pixel if the fragment is closer
- For specific effects, we can:
  - Use other tests
  - *Blend* the fragment color with the existing color instead of replacing it
    - E.g., when combined with back-to-front rendering, can approximate transparency
  - We need to be very careful when doing this in parallel!

Rasterization Rules: Line Primitives
Output fragment if line intersects “diamond”

Specifying the Viewport
- **Viewport**: Active section of framebuffer
- `glViewport(int x, int y, int width, int height)`
  - Lower left corner (in pixels)
  - Viewport size (in pixels)
  - Initially set to entire framebuffer
Normalized Device Coordinates

- Maps viewport to \([-1, 1]^2\)
- Allows us to use a consistent set of coordinates for projection
- OpenGL handles the mapping from NDC to pixel coordinates

Orthographic Projection Matrix

\[
\begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Maps \([l, r] \times [b, t] \times [f, n]\) to \([-1, 1]^3\)
- Since \(n\) and \(f\) are negative, \(n > f\)

View Volume

Visible part of scene, typically frustum of a pyramid

Mapped to \([-1, 1]^3\) in normalized device coordinates (everything outside is discarded)

Orthographic (Parallel) Projection

- Viewer at infinity
- Object appears same size regardless of distance
- View volume assumed to have bounding planes
  - \(x = l\) \(\equiv\) left plane
  - \(x = r\) \(\equiv\) right plane
  - \(y = b\) \(\equiv\) bottom plane
  - \(y = t\) \(\equiv\) top plane
  - \(z = n\) \(\equiv\) near plane
  - \(z = f\) \(\equiv\) far plane

Perspective Projection

- Objects further away appear smaller
- Rays converge at eye, assumed to be at origin
- \((l, r, b, t)\) now specify boundaries of view volume at near clipping plane
Perspective Projection Matrix

\[
\begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\
0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\
0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

- We finally use that homogenous coordinate!
- Remember to divide by \(h\) to get the final point

Camera Transformation

- The last missing piece is to align the camera with the direction of view
- Camera orientation is specified (in world coordinates) by:
  - the eye position \(e\)
  - the gaze direction \(g\)
  - the view-up vector \(t\)
  - (neither \(g\) nor \(t\) need be unit, and \(t\) need not even be exactly perpendicular to \(g\))

Camera Transformation Matrix

\[
\begin{bmatrix}
u_x & u_y & u_z & 0 \\
v_x & v_y & v_z & 0 \\
w_x & w_y & w_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
1 & 0 & 0 & -e_x \\
0 & 1 & 0 & -e_y \\
0 & 0 & 1 & -e_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Change of basis to \(uvw\) coordinates
Translate eye to origin

The Full Transformation Pipeline

Every object is transformed by

\[T = \text{Projection} \times \text{Camera} \times \text{Model}\]
OpenGL Matrix Modes

- To select the projection matrix (and stack):
  \[ \text{glMatrixMode}(	ext{GL_PROJECTION}); \]
- To select the model-view matrix (and stack):
  \[ \text{glMatrixMode}(	ext{GL_MODELVIEW}); \]

The Full Transform Once Again...

Every object is transformed by

\[ T = \text{Projection} \times \text{Camera} \times \text{Model} \]

Recap

- **Z-buffer** to detect visible surfaces
- Surfaces **tessellated** into simple primitives
- **Draw primitives** with \text{glBegin}/\text{glEnd} blocks
  - \text{glVertex}, \text{glNormal}, \text{glColor}
  - Primitives are rasterized to framebuffer
- **Nested transform blocks**
  - \text{glPushMatrix}, \text{glPopMatrix}, \text{glLoadMatrix}, \text{glMultMatrix}
- **Projection * Camera * Model** transform applied to each object
  - Perspective/orthographic projection, camera \( (uvw) \) coordinates, \text{GL_PROJECTION, GL_MODELVIEW}