Today

Real-time graphics with OpenGL
Crysis (Crytek/EA, 2007)
Hidden Surface Removal

Near objects hide (*occlude*) further objects

Elliott Erwitt, New York, 2000
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?

André Kertész, “Arm and Ventilator”, New York, 1937
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 $p = c$
Z-Buffer: Example

Rendered 3D scene

(Wikipedia)
Z-Buffer: Example

Corresponding z-buffer (dark: near, light: far)
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

- OpenGL
  - Windows, OS X, Linux, ...
    - ... which is why we'll look only at OpenGL in this course
  - http://www.opengl.org
Basic OpenGL

• Represent object surface as set of *primitive shapes*
  – Points
  – Lines
  – Triangles
  – Quad(rilateral)s
• 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

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

**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**

```c
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 \((R, G, B, \alpha)\)
Transforming Objects

• Let's see a simple example first...

\[ \text{glLoadMatrixf}(M); \quad \text{// M is a 4x4 matrix stored in column-major form} \]

\[ \text{// Draw the object using } \text{glBegin}/\text{glEnd} \]

• **Note:**
  
  • The object is transformed by \( M \) before it is drawn
    – Each vertex \( v \) becomes \( M \times v \)
  
  • \( M \) 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

```c
glLoadMatrixf(A); // Initial matrix
glMultMatrixf(B); // Note: MultMatrix, not LoadMatrix
glMultMatrixf(C);
...```

// Draw the object using glBegin/glEnd

• The object is transformed by $A \times B \times C$
  • Each vertex $v$ becomes $A \times B \times C \times v$

• **Note:** Transforms are applied *last-to-first!*
OpenGL Convenience Functions

- \texttt{glLoadIdentity()} \equiv \texttt{glLoadMatrixf(\textit{identity matrix})}
- \texttt{glTranslatef(tx, ty, tz)} \equiv \texttt{glMultMatrixf(T)}
  - \textit{T} is a matrix that translates by \((t_X, t_Y, t_Z)\)
- \texttt{glRotatef(angle, x, y, z)} \equiv \texttt{glMultMatrixf(R)}
  - \textit{R} is a matrix that rotates by \textit{angle} degrees around the axis \((x, y, z)\)
- \texttt{glScalef(sx, sy, sz)} \equiv \texttt{glMultMatrixf(S)}
  - \textit{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
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
    glMultMatrixf(M);
// Draw the object using glBegin/glEnd

glPopMatrix();
```

• **Questions:**
  • Why all the pushing/popping?
  • What's with this MatrixMode 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:
    - $\text{CTM} = \text{CTM} \times \text{T}$
  - 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

- body
- torso
- head
- shoulder
- leftArm
  - upperArm
  - lowerArm
  - hand
- rightArm
  - upperArm
  - lowerArm
  - hand
- hips
  - leftLeg
    - upperLeg
    - lowerLeg
    - foot
  - rightLeg
    - upperLeg
    - lowerLeg
    - foot
Example: Articulated Robot (OpenGL)

```c
glTranslatef(0, 1.5, 0);
drawTorso();
glPushMatrix();
glTranslatef(0, 5, 0);
drawShoulder();
glPushMatrix();
glTranslatef(1.5, 0, 0);
glRotatef(l_shoulder_x);
drawUpperArm();
glPushMatrix();
glTranslatef(0,-2,0);
glRotatef(l_elbow_x, 1, 0, 0);
drawLowerArm();

... glPopMatrix();
glPopMatrix();
```
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 $\Delta$, 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

<table>
<thead>
<tr>
<th>(0, 0)</th>
<th>(1, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 1)</td>
<td>(1, 1)</td>
</tr>
</tbody>
</table>
OpenGL Pixel Coordinates

Pixel centers are at *half-integer* coordinates
Output *fragment* if pixel center is inside area.
Rasterization Rules: Area Primitives

Combine fragment color with existing pixel color
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
**View Volume**

Visible part of scene, typically frustum of a pyramid

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

(everything outside is discarded)
Projective Transformation

- Maps view volume to $[-1, 1]^3$ (NDC)
- Viewer is assumed to be looking along $-Z$
  - Consistent with XY coordinates for viewport
Orthographic (Parallel) Projection

- Viewer at infinity
- Object appears same size regardless of distance
- View volume assumed to have bounding planes

\[
\begin{align*}
x &= l \quad \text{left plane} \\
x &= r \quad \text{right plane} \\
y &= b \quad \text{bottom plane} \\
y &= t \quad \text{top plane} \\
z &= n \quad \text{near plane} \\
z &= f \quad \text{far plane}
\end{align*}
\]
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, \(n > f\)
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

• We construct an orthonormal basis $[\mathbf{u}^\wedge, \mathbf{v}^\wedge, \mathbf{w}^\wedge]$ from $\mathbf{g}$, $\mathbf{t}$

\[
\begin{align*}
\mathbf{w}^\wedge &= \mathbf{g} / \| \mathbf{g} \|
\\
\mathbf{u}^\wedge &= (\mathbf{t} \times \mathbf{w}^\wedge) / \| \mathbf{t} \times \mathbf{w}^\wedge \|
\\
\mathbf{v}^\wedge &= \mathbf{w}^\wedge \times \mathbf{u}^\wedge
\end{align*}
\]

• $\mathbf{u}^\wedge$ is the target X axis, $\mathbf{v}^\wedge$ the target Y axis, and $\mathbf{w}^\wedge$ the target Z axis
Camera Transformation Matrix

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

Change of basis to \(uvw\) coordinates

Translate eye to origin
The Full Transformation Pipeline

Every object is transformed by

\[ T = \text{Projection} \times \text{Camera} \times \text{Model} \]
Back to that `glMatrixMode` thing...

- OpenGL maintains multiple current transformation matrices, and corresponding stacks
- The important ones (for us) are:
  - the *Model-View Matrix* $M$, and
  - the *Projection Matrix* $P$
- The full transformation applied to an object is actually $P \times M$ (in that order, right-to-left)
- By convention, the projective transform (perspective/orthographic) is put in $P$, and everything else (camera, model, ...) in $M$
OpenGL Matrix Modes

- To select the projection matrix (and stack):

  ```c
  glMatrixMode(GL_PROJECTION);
  ```

- To select the model-view matrix (and stack):

  ```c
  glMatrixMode(GL_MODELVIEW);
  ```
The Full Transform Once Again...

Every object is transformed by

$$T = \text{Projection} \times \text{Camera} \times \text{Model}$$

GL_PROJECTION  GL_MODELVIEW
Recap

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