Radiosity

Lightscape, Autodesk
Another Example

Modeling

- How to represent real environments
  - geometry: modeling surfaces, volumes
  - photometry: light, color, reflectance
- How to *build* these representations
  - declaratively: write it down
  - interactively: sculpt it
  - programmatically: let it grow
  - via 3D sensing: scan it in
Modeling by Sculpting

Freeform from Sensable Technologies

Synapse Modemaking
Modeling by Growing

Modeling by Growing

Seashell model with grown reaction diffusion pattern.
P. Prusinkiewicz, Deborah Fowler, Hans Meinhardt
Modeling by Scanning

Cyberware
Rendering

• What’s an image?
  – distribution of light energy on 2D “film”: \(E(x,y,\lambda,t)\) (\(\lambda\) is wavelength.)

• How do we represent and store images
  – sampled array of “pixels”: \(p[x,y]\)

• How to generate images from scenes
  – input: 3D description of scene, camera
  – solve light transport through environment
    • ray tracing
    • radiosity
  – project to camera’s viewpoint
What is OpenGL?

- A low-level graphics API for 2D and 3D interactive graphics. OS independent.
- Descendent of GL (from SGI)
- Implementations: For the Linux PCs we have Mesa, a freeware implementation.
What it isn’t:
A windowing program or input driver because those couldn’t be OS independent.

GL: core graphics capability
GLU: utilities on top of GL
GLUT: input and windowing functions
How does it work?

From the programmer’s point of view:
- Specify geometric objects
- Describe object properties
- Define how they should be viewed
- Move camera or objects around for animation
How does it work?

State machine with input and output:

- State variables: color, current viewing position, line width, material properties...
- These variables (the state) then apply to every subsequent drawing command
- Input is description of geometric object
- Output is pixels sent to the display
How does it work?

From the implementor’s perspective:
OpenGL pipeline

Walk through the pipeline...
Primitives: drawing a polygon

- Put GL into draw-polygon state
  \begin{verbatim}
  glBegin(GL_POLYGON);
  \end{verbatim}

- Send it the points making up the polygon
  \begin{verbatim}
  glVertex2f(x0, y0);
  glVertex2f(x1, y1);
  glVertex2f(x2, y2) ... 
  \end{verbatim}

- Tell it we’re finished
  \begin{verbatim}
  glEnd();
  \end{verbatim}

Build models in appropriate units (microns, meters, etc.). Transform to screen coordinates (pixels) later.
Primitives: points, lines, polygons

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_POINTS

GL_TRIANGLE_STRIP

GL_QUAD_STRIP

GL_POINTS

GL_POLYGON

GL_QUADS

GL_TRIANGLES
Primitives: points, lines, polygons

Why triangles, quads, and strips?

Hardware may be more efficient for triangles
Strips require processing less data (fewer vertices)
Primitives: Material Properties

- `glColor3f(r, g, b);`
  All subsequent primitives will be this color—
colors are not attached to objects but this call
changes the state of the system
Everyone who learns gl gets bitten by this!

Red, green & blue color model
Components are 0-1
Primitives: Material Properties

Many other material properties available:

```c
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(MASK); /* 32x32 pattern of bits */
...
glDisable (GL_POLYGON_STIPPLE);
```
Primitives: Material Properties

Ambient: same at every point on the surface
Diffuse: scattered light independent of angle (rough)

Specular: dependent on angle (shiny)
Transforms

- Rotate
- Translate
- Scale
- `glPushMatrix(); glPopMatrix();`
Position it relative to the camera

Different views of the objects in the world

(a)  
(b)  
(c)
Position it relative to the camera

Lines from each point on the image are drawn through the center of the camera lens (the center of projection).
Many camera parameters…
For a physical camera:
position (3)
orientation (3)
lens (field of view)

Orthographic projection: long telephoto lens.
Flat but preserving distances and shapes. All the projectors are now parallel.
`glOrtho` (left, right, bottom, top, near, far);
Position it relative to the camera

Perspective projection
Camera Transformations

Camera positioning just results in more transformations on the objects: transformations that position the object wrt to the camera.
Clipping

Not everything should be visible on the screen
Rasterizer

Go from pixel value in world coordinates to pixel value in screen coordinates

World coordinates  Raster coordinates

Clipping window

Viewport

Graphics window

(a)

(b)
void DrawBox()
{
    MakeWindow("Box", 400, 400);

    glOrtho(-1, 1, -1, 1, -1, 1);

    glClearColor(0.5, 0.5, 0.5, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f(1.0, 0.0, 0.0);

    glBegin(GL_POLYGON);
    /* or GL_LINES or GL_POINTS... */

    glVertex2f(-0.5, -0.5);
    glVertex2f( 0.5, -0.5);
    glVertex2f( 0.5,  0.5);
    glVertex2f(-0.5,  0.5);

    glEnd();
}
Setting up the window

- **The coordinate system**
  
  `glOrtho(left, right, bottom, top, near, far);`
  
  e.g., `glOrtho(0, 100, 0, 100, -1, 1);`
  
  For now, near & far should always be -1 & 1

- **Clearing the screen**
  
  `glClearColor(r, g, b, a);`
  
  `a` is the alpha channel; set this to 0.
  
  `glClear(GL_COLOR_BUFFER_BIT);`
  
  `glClear` can clear other buffers as well, but we’re only using the color buffer...