3D Object Modeling

- The model-view-controller perspective actually includes many layers
- Example: some parts of an application’s model are closer to the view than other parts

**pure presentation**  
**pure content**

```
Player Avatar
vertices
poses
skins
Player
name
health
inventory
```

- Start by defining the pure content (the “model model”)
  - Unaffected by presentation issues
  - You can tell that information belongs to the “model model” if it makes sense (or matters) regardless of whether you present the application in a 3D environment (e.g., if you did a text version of your program, what information do you still need?)

- Define the minimal additions necessary for “block” 3D renditions of your objects
  - Do enough so that you can work on the view and controller aspects of your application
  - If subelements of an object affect the logic of your application (e.g., arm articulation, line of sight), integrate that at this point also

- Integrate more detailed 3D renderings; these should no longer affect the “model model” nor the pure view and controller directly

The 3D rendering is what we’re talking about today…
MVC with 3D Graphics

3D Object Model Criteria

<table>
<thead>
<tr>
<th>General</th>
<th>Model should be capable of representing as many objects as possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unambiguous</td>
<td>Model should not allow a single representation to correspond to more than one object</td>
</tr>
<tr>
<td>Unique</td>
<td>A particular object should be represented in exactly one way — needed when comparing two objects for &quot;equality&quot;</td>
</tr>
<tr>
<td>Accurate</td>
<td>Model should produce representations that exactly denote the object (i.e., no approximations)</td>
</tr>
<tr>
<td>Valid</td>
<td>Model should protect against creating bad representations (incomplete data, garbage, null)</td>
</tr>
<tr>
<td>Closed</td>
<td>Transforming objects in the model should produce other valid objects in the model</td>
</tr>
<tr>
<td>Compact</td>
<td>The less required storage, the better</td>
</tr>
<tr>
<td>Efficient</td>
<td>Algorithms that process the model should be as &quot;lean and mean&quot; as possible</td>
</tr>
</tbody>
</table>
Constructive Solid Geometry

- “Lego” approach to 3D models
  - Core set of primitives with modifiable attributes (color, textures, lighting, shininess, translucence)
  - Modify primitives with transforms (translate, scale, rotate), other techniques (extrusion, lathing)
  - Allow composites (combining primitives or other composites into a single construct), including intersection and difference in addition to union
- GLUT shapes can be viewed as a possible CSG basis
CSG Display Algorithm

• Individual components:
  ◦ Preserve the current matrix
  ◦ Apply transforms
  ◦ Set properties (colors, textures)
  ◦ Draw the component
  ◦ Restore the previous matrix

• Composites (via union, difference, or intersection) require additional processing (union is easiest)

Images by Dr. Neil Dodgson, University of Cambridge
CSG Implementation

**Tree Data Structure:**
- Primitives (leaves)
- Composites (non-leaves)
- Transforms (linear chains over components)

**Rendering Engine:**
- Traverse CSG tree, set transforms, implement compositions, draw component

**Primitives Library:**
- Direct drawing functions
- Pre-stored meshes
- Curve or surface representations

**GLUT (or GLUT-like):**
- Needs mesh renderer
- Needs curve/surface renderer

CSG Examples

Images by Dr. Neil Dodgson, University of Cambridge

Images from "The Online POV-Ray Tutorial"
Curves and Surfaces

- Represent objects as arrays of control points
  - One-dimensional array: curve
  - Two-dimensional array: surface
- Interpolation generates the final curve
  - Many techniques: Bezier, assorted splines, Hermite, Catmull-Clark — check Angel Chapter 9 for details
- Can be performance-intensive, but resolution-independent — useful for offline vertex generation

Curves and Surfaces Implementation

data structure: control points (1D or 2D); curve or surface specification; other attributes (color, textures, lighting, shininess, translucence)

vertex generator: interpolate the "final" vertices through the control points

data structure: control points (1D or 2D); curve or surface specification; other attributes (color, textures, lighting, shininess, translucence)

curve/surface renderer: on curves, line segments connect vertices; on surfaces, polygons connect vertices

other attributes come into play here — color, textures, etc.

other 3D model (CSG primitive, polygon mesh)
Polygon Meshes

- Most general (and common) 3D modeling technique
- Varying levels of redundancy (you’ll see what I mean)
- Very amenable to implementation as abstract data type or object-oriented class
- Nate Robins tutorials include a simple polygon mesh implementation

Key Ideas

- A mesh is a set of vertices
- Just vertices are not enough; they must be aggregated, either in edges, faces, or both
- In geometry terms, a mesh defines a polyhedron
- Additional attributes are stored either per vertex, per edge, or per face:
  - Color, textures
  - Lighting, shininess, translucence
- Wireframe rendering only needs the edges
  ◆ If doing wireframes, the edge list would have visual properties the way faces do: color, style, thickness, etc.

- Vertex order matters! (because…?) — OpenGL convention is counterclockwise from the “front”

- Normals can be derived in a standard way, or customized for special effects (typically lighting)

- With open meshes (meshes with holes), you may need to double up some surfaces so that both the “front” and “back” are visible

- Note some redundancy between the edge and face lists — implications include:
  ◆ Possible automation of some mesh-building routines
  ◆ In the case of an arbitrary mesh (i.e., read from a file), may need consistency checks
Mesh Consistency Rules

- Every vertex should be in at least 2 edges
- Every edge should be in at least one face (at least 2 faces for a closed mesh)
- At least one shared edge per face
- Vertex list in face cannot be collinear
- When > 3 vertices, must be coplanar — not fatal, but may result in some rendering issues
- Every vertex that is an edge endpoint must also appear in the vertex list for each surface listed by the edge

Mesh Implementatation

- Mesh abstract data type: vertex, edge, and face data structures; management routines (add/remove, get/set); error handling
- Mesh renderer: one uniform piece of code; may loop through multiple polygon meshes
- Mesh I/O: theoretically optional, but in practice necessary — read mesh object from a file or string, or build from a curve or surface model
- Top-level CSG tree: treat meshes as CSG components — requires generalized implementations of CSG composition
Mesh Renderer Fragment: Wireframe, Java

```java
/**
 * Syntax is specific to Java 1.5 or later.
 */
public void paintMesh(Graphics g) {
    g.clearRect(0, 0, getSize().width, getSize().height);
    for (Mesh m: getMeshList()) {
        for (List<Vertex3D> vList: m.getSurfaceList()) {
            Vertex3D lastV = null;
            for (Vertex3D v1: vList) {
                Vertex3D newV = projection.multiply(modelView.multiply(v1));
                if (lastV != null) paintEdge(g, lastV, newV);
                lastV = newV;
            }
        }
    }
}
```

In JOGL or OpenGL, this would be some combination of `glBegin(…), glVertex*(), and glEnd()`.

Mix and Match!

- No single modeling approach is the “end all be all”
- Modeling approaches are not mutually exclusive: you can combine them as needed
- Very much driven by your application’s needs
- Despite different per-application specifics, it is still very possible (and recommended) to develop reusable code
- OpenGL designers decided to leave these modeling decisions to the developer, choosing to provide lower-level support instead