3D Object Modeling

- In the world of model-view-controller, there may be sublayers within layers.

- Case in point: the model
  - Some parts of the model are closer to the view than others

```
player geometry
- limb articulation
- movement
- appearance
```

This is what we’re talking about

```
pure presentation
pure content
```

player
- name
- health
- inventory

General Development Flow

- Define the pure content (the “model model”) in your application
  - The parts that are not affected by the presentation
  - They make sense whether your application is in 3D, over the Web, or even just text

- Create “block” versions of 3D objects while putting together the view and controller of your application
  - Depending on the application, this can take you quite a ways

- Integrate more detailed 3D models; this should generally no longer affect the core model nor the pure view and controller
  - Exception: application involves changes in the articulation of 3D objects, such as a human body…but even for this, good design should still keep things separate
General Runtime Flow

- **Initialization**
- **Core Model**
- **3D Object Model(s)**
- **View**
  - View-related state affects camera position, transformations, HUD, etc.
  - 3D object models translate into vertices, lighting, materials, textures
- **Controller**
  - Maps user actions to changes to the model
  - Idle time may also change the model
- **User**

3D Model Criteria

- **General**: the model should be capable of representing as many objects as possible
- **Unambiguous**: the model should not allow a single representation to correspond to more than one object
- **Unique**: a particular object should be represented in exactly one way — needed if you sometimes need to compare two objects for “equality”
- **Accurate**: the model should produce representations that exactly denote the object (i.e. no approximations)
- **Valid**: the model should protect against creating bad representations (incomplete data, garbage, null)
- **Closed**: transforming objects in the model should produce other valid objects in the model
- **Compact**: the less required storage, the better
- **Efficient**: algorithms that process the model should be as “lean and mean” as possible
Constructive Solid Geometry

• “Lego” approach to 3D models
  – Start with a core set of primitives
  – Modify primitives by:
    • Using transforms to position, scale, and rotate them
    • For 2D primitives, use 2D-to-3D techniques like extrusion, lathes
    • Combining primitives into composites
  – Additional attributes at a per-primitive or per-composite level:
    • Color, textures
    • Lighting, shininess, translucence

• Object composition
  – union
  – intersection
  – difference

• GLUT primitives can be viewed as a start for CSG

Each CSG primitive would have default properties.

sphere: radius 1, center (0, 0, 0)
cube: side 1, corner (0, 0, 0)
cylinder: radius 1, height 2, center (0, 0, 0)
cone: radius 1, height 2, center (0, 0, 0)

…define as many primitives as you like — even a teapot!
Modification by transform:

- **cube**
  - translate by 5, 10, 20
  - scale by 5, 10, 20
  - rotate 40° about (1, 1, 0)

- **cylinder**
  - translate by –1, 1, 3
  - scale by 2, 2, 2
  - rotate –10° about (1, 0, 1)

- **polygon**
  - extrude by 2
  - rotate –10° about (1, 0, 1)

Compose by union, intersection, or difference:

- **union**
  - scale by 1, 2, 1
  - translate by 0, 3.5, 0

- **cube**
  - translate by 0, 3.5, 0
  - scale by 1, 1.5, 1

- **sphere**

Union is easiest to implement; intersection and difference take more work.
Curves and Surfaces

- Represent objects as arrays of control points
  - one-dimensional array: curve
  - two-dimensional array: surface

- Use interpolation techniques to generate the final curve or surface
  - many techniques: Bezier, assorted splines, Hermite, Catmull-Clark
  - consult Angel Chapter 9 for details

- Can be performance intensive…
- …but “resolution independent”
  - Thus, curves and surfaces can be used as an offline model generator: not used in real time, but used to create sets of vertices for meshes or CSG
Curves and Surfaces Implementation

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

Vertex generator: iterate through the control points, interpolating the “final” vertices

Curve/surface renderer: line segments connect vertices on curve; polygons connect vertices on surface

Other 3D model: CSG primitive, polygon mesh

Remember to take other attributes into account, if available.
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

- Key ideas:
  - A mesh is a set of vertices.
  - However, vertices are not enough. Vertices 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
Mesh Observations

- Wireframe rendering only needs the edges, and just the vertex information at that.
  - If doing wireframes, the edge list would have visual properties the way faces do: color, style, thickness, etc.
- Vertex order matters! (because…?) Convention is counterclockwise.
- 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

- 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 listed as an edge endpoint must also appear in the vertex list for each surface listed by the edge
Mesh Implementation

abstract data type - data structure(s) - management routines (add, remove, query) - error handling

mesh I/O (optional) - parse from string - read from file - generate from curve or surface model

mesh renderer - single piece of code - may loop through multiple meshes, or…

top-level CSG tree - treats meshes as CSG primitives - the usual CSG goodies - typically known as a scene graph - how Java3D does things

Mesh Renderer Fragment (Wireframe, Java)

```java
public void paintMesh(Graphics g) {
    g.clearRect(0, 0, getSize().width, getSize().height);
    for (Iterator meshIt = meshes.iterator(); meshIt.hasNext(); ) { 
        Mesh m = (Mesh)meshIt.next();
        for (Iterator surfIt = m.surfaces.iterator(); surfIt.hasNext(); ) {
            List v = (List)surfIt.next();
            Vertex3D lastV = null;
            for (Iterator vIt = v.iterator(); vIt.hasNext(); ) {
                Vertex3D v1 = (Vertex3D)vIt.next();
                Vertex3D newV = projection.multiply(modelView.multiply(v1));
                if (lastV != null) paintEdge(g, lastV, newV);
                lastV = newV;
            }
        }
    }
}
```

For painted faces in 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

• But, despite very different per-application specifics, it is still very possible (and recommended) to shoot for highly reusable code

• OpenGL design decision: leave these modeling decisions to the user; provide lower-level support