Outline For Today

- *Scene Graphs*
- Shapes
- Tessellation
Modeling by writing a program

- First two projects: Scene hard-coded in the model
  - The scene exists only in the drawScene() method

- Advantages:
  - Simple,
  - Direct

- Problems
  - Code gets complex
  - Special-purpose, hard to change
  - Special-purpose, hard to make many variants
  - Can’t easily examine or manipulate models
    - Can only “draw”
Sample Scene

KK 5045
1500x450x760mm

KK 5060
1500x600x760mm
Schematic Diagram (Top View)
Top view with Coordinates
Hierarchical Transforms

- Last week, introduced hierarchical transforms
- Scene hierarchy:

```
WORLD
  ┌── Table1
  |   ├── Lamp
  |   └── Book1
  └── Table 2
        ├── Plant
        └── PC
              └── Monitor
                  └── Keyboard
```
Data structure for hierarchical scene

Want:
- Collection of individual models/objects
- Organized in groups
- Related via hierarchical transformations

Use a tree structure

Each node:
- Has associated local coordinates
- Can define a shape to draw in local coordinates
- Can have children that inherit its local coordinates

Typically, different classes of nodes:
- “Transform nodes” that affect the local coordinates
- “Shape nodes” that define shapes
Scene Tree
Node base class

- A Node base class might support:
  - `getLocalTransform()` -- matrix puts node’s frame in parent’s coordinates
  - `getGeometry()` -- description of geometry in this node (later today)
  - `getChild(i)` -- access child nodes
    - `addChild()`, `deleteChild()` -- modify the scene

- Subclasses for different kinds of transforms, shapes, etc.

- Note: many designs possible
  - Concepts are the same, details differ
  - Optimize for: speed (games), memory (large-scale visualization), editing flexibility (modeling systems), rendering flexibility (production systems), …
  - In our case: optimize for pedagogy & projects
class Node {
    // data
    Matrix localTransform;
    Geometry *geometry;
    Node *children[N];
    int numChildren;

    // methods:
    Matrix getLocalTransform() { return localTransform; }
    Geometry *getGeometry() { return geom; }
    Node *getChild(int i) { return children[i]; }
    void addChild(Node *c) { children[numChildren++] = c; }
}
draw(Node node) {
    PushCTM();
    Transform(node.getLocalTransform());
    drawGeometry(node.getGeometry());
    for (i=0; i<node.numChildren; ++i) {
        draw(node.child[i]);
    }
    PopCTM();
}

- Effect is same hierarchical transformation as last week
Modify the scene

- Change tree structure
  - Add nodes
  - Delete nodes
  - Rearrange nodes
- Change tree contents
  - Change transform matrix
  - Change shape geometry data
- Define subclasses for different kinds of nodes
  - Subclass has parameters specific to its function
  - Changing parameter causes base info to update
Example: Translation Node

class Translation(Transformation) {
    private:
        float x,y,z;
    void update() {
        localTransfom.MakeTranslation(x,y,z);
    }

    public:
    void setTranslation(float tx, float ty, float tz) {
        x = tx; y = ty; z = tz;
        update();
    }
    void setX(float tx) { x = tx; update(); }  
    void setY(float ty) { y = ty; update(); }  
    void setZ(float tz) { z = tz; update(); }  
};
Example: Rotation Node

class Rotation(Transformation) {
    private:
        Vector3 axis;
        float angle;
    void update() {
        localTransform.MakeRotateAxisAngle(axis,angle);
    }

    public:
        void setAxis(Vector3 v) {
            axis = v;
            axis.Normalize();
            update();
        }
        void setAngle(float a) {
            angle = a;
            localTransform.MakeRotateAxisAngle(axis,angle);
        }
    }
}
More detailed scene graph
Building this scene

WORLD = new Node();
table1Trans = new Translation(...); WORLD.addChild(table1Trans);
table1Rot = newRotation(...); table1Trans.addChild(table1Rot);
table1 = makeTable(); table1Rot.addChild(table1);
top1Trans = new Translation(...); table1Rot.addChild(top1Trans);

lampTrans = new Translation(...); top1Trans.addChild(lampTrans);
lamp = makeLamp(); lampTrans.addChild(lamp);

book1Trans = new Translation(...); top1Trans.addChild(book1Trans);
book1Rot = newRotation(...); book1Trans.addChild(book1Rot);
book1 = makebook(); book1Rot.addChild(book1);

book2Trans = new Translation(...); top1Trans.addChild(book2Trans);
book2Rot = newRotation(...); book2Trans.addChild(book2Rot);
book2 = makebook(); book2Rot.addChild(book2);

table2Trans = new Translation(...); WORLD.addChild(table2Trans);
table2Rot = newRotation(...); table2Trans.addChild(table2Rot);
table2 = makeTable(); table2Rot.addChild(table2);
top2Trans = new Translation(...); table2Rot.addChild(top2Trans);

Still building the scene hardwired in the program
- But now can more easily manipulate it...
Change scene

- Change a transform in the tree:
  - `table1Rot.setAngle(23);`
  - Table rotates, everything on the table moves with it

- Allows easy animation
  - Build scene once at start of program
  - Update parameters to draw each frame
  - e.g. Solar system:
    ```java
drawScene() {
  sunSpin.setAngle(g_Rotation);
  earthSpin.setAngle(3*g_Rotation);
  earthOrbit.setAngle(2*g_Rotation);
  moonOrbit.setAngle(8*g_Rotation);
  draw(WORLD);
}
```

- Allows interactive model manipulation tools
  - e.g. button to add a book
    - Create subtree with transforms and book shape
    - Insert as child of table
Not just transform nodes

- Shape nodes
  - Contain geometry:
    - cube, sphere (later today)
    - curved surfaces (next week)
    - Etc…

- Can have nodes that control structure
  - Switch/Select: parameters choose whether or which children to enable
  - Group nodes that encapsulate subtrees
  - Etc…

- Can have nodes that define other properties:
  - Color
  - Material
  - Lights
  - Camera
  - Etc…

- Again, different details for different designs
OpenInventor Scene Graph
Maya “Hypergraph”
Scene vs. Model

- No real difference between a scene and a model
  - A scene is typically a collection of “models” (or “objects”)
  - Each model may be built from “parts”
- Use the scene graph structure
  - Scene typically includes cameras, lights, etc. in the graph; Model typically doesn’t (but can)
Parameteric models

- Parameters for:
  - Relationship between parts
  - Shape of individual parts
- Hierarchical relationship between parts
- Modeling robots
  - separate rigid parts
  - Parameters for joint angles
  - Hierarchy:
    - Rooted at pelvis: Move pelvis, whole body moves
    - Neck & Head: subtree; move neck and head, or just move head
    - Arms: Shoulder, Elbow, Wrist joints
    - Legs: Hips, Knee, Ankle joints
- This model idiom is known as: an Articulated figure
- Often talk about degrees of freedom (DOFs)
  - Total number of float parameters in the model
Robot
Screen *Graph*, not Tree

- **Repetition:**
  - A scene might have many copies of a model
  - A model might use several copies of a part

- **Multiple Instantiation**
  - One copy of the node or subtree
  - Inserted as a child of many parents
  - A directed acyclic graph (DAG), not a tree
  - Traversal will draw object each time, with different coordinates

- **Saves memory**
  - Can save time also, depending on cacheing/optimization

- **Change parameter once, affects all instances**
  - This can be good or bad, depending on what you want
  - Some scene graph designs let other properties inherit from parent
Instantiation - scene
Instantiation - model parts
Instantiation (OpenInventor)
Fancier things to do with scene graphs

- Skeletons, skin, deformations
  - Robot-like jointed rigid skeleton
  - Shape nodes that put surface across multiple joint nodes
  - Nodes that change shape of geometry

- Computations:
  - Properties of one node used to define values for other nodes
  - Sometimes can include mathematical expressions
  - Examples:
    - Elbow bend angle -> bicep bulge
    - Our scene has translation to put objects on table…
      - But how much should that translation be?
      - What if the table changes?
Linked parameters
Linked parameters
Other things to do with scene graphs

- Names/paths
  - Unique name to access any node in the graph
  - e.g. “WORLD/table1Trans/table1Rot/top1Trans/lampTrans”

- Compute Model-to-world transform
  - Walk from node through parents to root, multiplying local transforms

- Bounding box or sphere
  - Quick summary of extent of object
  - Useful for culling (next class)
  - Compute hierarchically:
    - Bounding box is smallest box that encloses all children’s boxes

- Collision/contact calculation

- Picking
  - Click with cursor on screen, determine which node was selected

- Edit: build interactive modeling systems
Project 3 Scene Graph

- Just the basics...
- Transform nodes
  - Rotation
  - Translation
- Shapes
  - Cube
  - Sphere
- Traversal/drawing
Outline For Today

- Scene Graphs
- *Shapes*
- Tessellation
Basic shapes

- Geometry objects for primitive shape types
- Various exist.
- We’ll focus on fundamental: Collection of triangles
  - AKA *Triangle Set*
  - AKA *Triangle Soup*
- How to store triangle set?
  - …simply as collection of triangles?
Cube - raw triangles

- 12 triangles:
  - (-1,-1,1) (1,-1,1) (1,1,1)
  - (-1,-1,1) (1,1,1) (-1,1,1)
  - (1,-1,1) (1,-1,-1) (1,1,-1)
  - (1,-1,1) (1,1,-1) (1,1,1)
  - (1,-1,-1) (-1,-1,-1) (-1,1,-1)
  - (1,-1,-1) (-1,1,-1) (1,1,1)
  - (1,-1,-1) (-1,1,-1) (1,1,1)
  - (1,-1,-1) (-1,1,1) (-1,1,1)
  - (-1,1,1) (1,1,1) (1,1,-1)
  - (-1,1,1) (1,1,-1) (-1,1,-1)
  - (1,-1,1) (-1,-1,1) (1,-1,1)
  - (1,-1,1) (-1,1,1) (-1,1,-1)
- 12*3=36 vertices
But....

- A cube only has 8 vertices!
- 36 vertices with x, y, z = 36*3 floats = 108 floats.
  - Would waste memory to store all 36 vertices
  - Would be slow to send all 36 vertices to GPU
  - (Especially when there is additional data per-vertex)
- Usually each vertex is used by at least 3 triangles--often 4 to 6 or more
  - Would use 4 to 6 times as much memory as needed, or more
- Instead: Specify vertex data once, then reuse it
  - Assign a number to each vertex
  - Specify triangles using vertex numbers
Cube - indexed triangles

- 8 vertices:
  - P0: (1, -1, 1)
  - P1: (1, -1, -1)
  - P2: (1, 1, -1)
  - P3: (1, 1, 1)
  - P4: (-1, -1, 1)
  - P5: (-1, -1, -1)
  - P6: (-1, 1, -1)
  - P7: (-1, 1, 1)

- 12 triangles:
  - P4 P0 P3
  - P4 P3 P7
  - P0 P1 P2
  - P0 P2 P3
  - P1 P5 P6
  - P1 P6 P2
  - P5 P4 P7
  - P5 P7 P6
  - P7 P3 P2
  - P7 P2 P6
  - P0 P5 P1
  - P0 P4 P5

8 vertices*3 floats = 24 floats
12 triangles*3 points = 36 integers
Indexed Triangle set

- Array of vertex locations, array of Triangle objects:

```java
Point3 vertices[] = {
    ( 1,-1, 1),
    ( 1,-1,-1),
    ( 1, 1,-1),
    ( 1, 1, 1),
    (-1,-1, 1),
    (-1,-1,-1),
    (-1, 1,-1),
    (-1, 1, 1)};

class Triangle {short p1, p2, p3) triangles[] = {
    (4, 0, 3),
    (4, 3, 7),
    (0, 1, 2),
    (0, 2, 3),
    (1, 5, 6),
    (1, 6, 2),
    (5, 4, 7),
    (5, 7, 6),
    (7, 3, 2),
    (7, 2, 6),
    (0, 5, 1),
    (0, 4, 5)};
```

- Triangles refer to each vertex by its index in the vertex array
Benefits of indexing

- Saves memory
- Saves data transmission time
- Save rendering time: lighting calculation can be done just one for each vertex
- Easy model *deformation*
  - Change vertex position data
  - Triangles automatically follow
- *Topology* (point connectivity) separate from shape (point locations)
(Index vs. pointer)

- Triangle stores indexes into the vertex array.
- Could also use pointer rather than index
  - Can be easier to work with
  - But uses more memory (if pointer is larger than short integer)
  - Can be fragile: if vertex array is reallocated pointers will dangle
Normals

- Normal = perpendicular to surface
- The normal is essential to lighting
  - Shading determined by relation of normal to eye & light
- Collection of triangles with their normals: *Facet Normals*
  - Store & transmit *one normal per triangle*
  - Normal constant on each triangle--but discontinuous at triangle edges
  - Renders as facets
  - Good for faceted surfaces, such as cube
- For curved surface that is approximated by triangles: *Vertex Normals*
  - Want normal to the surface, not to the triangle approximation
  - Don’t want discontinuity: share normal between triangles
  - Store & transmit *one normal per vertex*
  - Each triangle has different normals at its vertices
    - Lighting will interpolate (a few weeks)
    - Gives illusion of curved surface
Facet normals vs. Vertex normals
Color

- Color analogous to normal
  - One color per triangle: faceted
  - One color per vertex: smooth colors
Indexed Triangle Set with Normals & Colors

- **Arrays:**
  ```cpp
  Point3 vertexes[];
  Vector3 normals[];
  Color colors[];
  Triangle triangles[];
  int numVertexes, numNormals, numColors, numTriangles;
  ```

- **Single base class to handle both:**
  - **Facets**
    - one normal & color per triangle
    - `numNormals = numColors = numTriangles`
  - **Smooth**
    - one normal & color per vertex
    - `numNormals = numColors = numVertexes`
Geometry objects base class

- (For our design) Base class supports indexed triangle set

```cpp
class Geometry {
    Point3 vertices[];
    Vector3 normals[];
    Color colors[];
    Triangle triangles[];
    int numVertices, numNormals, numColors, numTriangles;
};

class Triangle {
    int vertexIndices[3];
    int normalIndices[3];
    int colorIndices[3];
};
```

- Triangle indices:
  - For facet normals, set all three `normalIndices` of each triangle to same value
  - For vertex normals, `normalIndices` will be same as `vertexIndices`
  - Likewise for color
class Cube(Geometry) {
    Cube() {
        numVertices = 8;
        numTriangles = numNormals = 12;
        vertices = {
            ( 1, 1, 1), ( 1, 1, 1),
            ( 1, 1, 1), ( 1, 1, 1),
            ( 1, 1, 1), ( 1, 1, 1),
            ( 1, 1, 1), ( 1, 1, 1) };
        triangles = {
            (4, 0, 3), (4, 0, 3),
            (0, 1, 2), (0, 1, 2),
            (1, 5, 6), (1, 5, 6),
            (5, 4, 7), (5, 4, 7),
            (7, 3, 2), (7, 3, 2),
            (0, 5, 1), (0, 5, 1) };
        normals = {
            (0, 0, 1), (0, 0, 1),
            (1, 0, 0), (1, 0, 0),
            (0, 0, -1), (0, 0, -1),
            (1, 0, 0), (1, 0, 0),
            (0, 1, 0), (0, 1, 0),
            (1, 0, 0), (1, 0, 0) };
    }
}
Smooth surfaces

- **Tessellation**: approximating a smooth surface with a triangle mesh
  - Strictly speaking, “tessellation” refers to regular tiling patterns
  - In computer graphics, often used to mean any *triangulation*
- E.g. Sphere class fills in triangle set (will get to this shortly…)
  
  ```
  class Sphere(Geom) {
    private:
      float radius;
    void tesselate() {
      vertices = …
      triangles = …
      normals=…
    }
    public:
      Sphere(float r) { radius = r; tesselate(); }
      void setRadius(float r) { radius = r; tesselate(); }
  }
  ```

- Other smooth surface types
  - Bezier patch (next week)
  - NURBS
  - Subdivision surface
  - Implicit surface
Drawing the indexed triangle set

- OpenGL supports “vertex arrays”
  - But it’s awkward to use
- So for project 3:
  - Use indexed triangle set for base storage
  - Draw by sending all vertex locations for each triangle:
    ```
    for (i=0; i<numTriangles; i++) {
      glVertex3fv(vertexes[triangles[i].p1]);
      glVertex3fv(vertexes[triangles[i].p2]);
      glVertex3fv(vertexes[triangles[i].p3]);
    }
    ```
- So we get memory savings in Geometry class
- We don’t get speed savings when drawing.
Basic indexed triangle set is unstructured: “triangle soup”

GPUs & APIs usually support slightly more elaborate structures

Most common: triangle strips, triangle fans

- Store & transmit ordered array of vertex indexes.
  - Each vertex index only sent once, rather than 3 or 4-6 or more

- Even better: store vertexes in proper order in array
  - Can draw entire strip or fan by just saying which array and how many vertexes
  - No need to send indexes at all.

- Can define triangle meshes using adjacent strips
  - Share vertexes between strips
  - But must use indexes
Vertex Buffers

- Graphics hardware systems often support for *vertex buffer*
  - Memory on the GPU side
  - (AKA other things too)
- Particularly useful if model doesn’t deform
- Send vertex array data to GPU once
  - Includes per-vertex color or normal data
- Once data is on GPU, can be reused quickly
  - More than one triangle set or strips/fans referring to shared points
  - For animation: don’t need to send vertex data each frame!
- Index buffers too:
  - Store vertex index arrays in GPU memory
  - Don’t need to transmit index array each frame
Model I/O

- Usually have the ability to load data from some sort of file
- There are a variety of 3D model formats, but no universally accepted standards
- More formats for mostly geometry (e.g. indexed triangle sets) than for complete complex scene graphs
  - File structure unsurprising: List of vertex data, list(s) of triangles referring to the vertex data by name or number
Modeling Operations

- Surface of Revolution
- Sweep/Extrude
- Mesh operations
  - Stitching
  - Simplification -- deleting rows or vertices
  - Inserting new rows or vertices
- Filleting
- Boolean combinations
- Digitize
- Procedural modeling, scripts…

*Could be some interesting final projects here
Materials & Grouping

- Usually models are made up from several different materials
- The triangles are usually grouped and drawn by material
  - Minimize changes to “graphics state”--typically expensive to change
  - Using scene graph:
    - Geometry nodes with same material grouped together
    - “Material” nodes that define surface properties
Outline For Today

- Scene Graphs
- Shapes
- *Tessellation*
Tessellation

- Given a description of a surface
- Construct a triangle set (typically a mesh)
- Triangle set is an approximation
  - Fewer triangles: Faster, but less accurate
    - *Polygonal artifacts*
    - Especially at silhouettes
  - More triangles: slower, but more accurate
  - In the extreme, make each triangle the size of a pixel (or less)
- Fancy algorithms: *adaptive*
  - E.g., Make smaller triangles near silhouettes
  - E.g., Use fewer triangles when objects are far away
  - But must update/recompute tessellation each frame
    - Balance between cost of adaptive tessellation vs. rendering savings
Tessellating a sphere

- Various ways to do it
- We’ll pick a straightforward one:
  - North & South poles
  - Latitude circles
  - Triangle strips between latitudes
  - Fans at the poles
Latitude circles

Given:

\[ M = \# \text{ latitude circles} \]
\[ R = \text{ radius of sphere} \]

For \( i \)th circle: \( i \) from 1 to \( M \)

\[ r_i = R \sin \left( \frac{i \pi}{M + 1} \right) \]
\[ z_i = -R \cos \left( \frac{i \pi}{M + 1} \right) \]
Points on each latitude circle

Given $i$th circle:

\[ N = \# \text{ points in each circle} \]
\[ r_i = \text{radius of } i\text{th circle} \]
\[ z_i = \text{height of } i\text{th circle} \]

For $j$th point: $j$ from 0 to $N - 1$

\[ P_{ij} = (r_i \cos(2\pi j / N), r_i \sin(2\pi j / N), z_i) \]

\[
P_{ij} = \left( R \sin \left( i \frac{\pi}{M + 1} \right) \cos \left( j \frac{2\pi}{N} \right), R \sin \left( i \frac{\pi}{M + 1} \right) \sin \left( j \frac{2\pi}{N} \right), -R \cos \left( i \frac{\pi}{M + 1} \right) \right) \]

\[ (r_i \cos(7\pi / 4), r_i \sin(7\pi / 4), z_i) \]
Topological structure

\[ P_{ij} \leftrightarrow P[(i-1)N+1+j] \]
Normals

- For a sphere, normal per vertex is easy!
  - Radius vector from origin to vertex is perpendicular to surface
  - I.e., use the vertex coordinates as a vector, normalize it
Algorithm Summary

- Fill vertex array and normal array:
  - South pole = (0,0,-R);
  - For each latitude i, for each point j in the circle at that latitude
    - Compute coords, put in vertexes
      - Put points in vertices[0].vertices[M*N+1] as per previous slides
  - North pole = (0,0,R)
  - Normals coords are same as point coords, normalized

- Fill triangle array:
  - N triangles between south pole and Lat 1
  - 2N triangles between Lat 1 & Lat 2, etc.
  - N triangles between Lat M and north pole.
Whew

- Project3 has you implement scene graph, indexed model set, sphere tessellation, and build a scene or two
  - Be warned: Project3 probably more work than projects 1 or 2!

- Next class: cameras, perspective & viewing, culling (material for project 4)