Computer Graphics

Lecture 18
Hierarchical Transformations
The Graphics Pipeline
Announcements

• No more problems will be added to HW2.
Transformation Hierarchies
AKA Scene Graphs

- Represent a drawing ("scene") as a list of objects
- Transform for each object
  - can use minimal primitives: ellipse is transformed circle
  - transform applies to points of object
Example

- Can represent drawing with flat list
  - but editing operations require updating many transforms
Example

- Can represent drawing with flat list
  - but editing operations require updating many transforms
Groups of objects

• Treat a set of objects as one
• Introduce new object type: group  
  – contains list of references to member objects
• This makes the model into a tree  
  – interior nodes = groups  
  – leaf nodes = objects  
  – edges = membership of object in group
Demo: Drawing in PowerPoint
Example

- Add group as a new object type
  - lets the data structure reflect the drawing structure
  - enables high-level editing by changing just one node
Example

• Add group as a new object type
  – lets the data structure reflect the drawing structure
  – enables high-level editing by changing just one node
Groups of objects

- Treat a set of objects as one
- Introduce new object type: group
  - contains list of references to member objects
- This makes the model into a tree
  - interior nodes = groups
  - leaf nodes = objects
  - edges = membership of object in group
The Scene Graph (tree)

• A name given to various kinds of graph structures (nodes connected together) used to represent scenes

• Simplest form: tree
  – just saw this
  – every node has one parent
  – leaf nodes are identified with objects in the scene
Instances

- Simple idea: allow an object to be a member of more than one group at once
  - transform different in each case
  - leads to linked copies
  - single editing operation changes all instances
Example: Whiteboard
Questions?
Questions?

• That wraps up our discussion of transformations.

• We have an (almost) fully-featured wireframe rendering framework.
  • We haven't implemented clipping yet for geometry outside the view volume.

• Next up:
  • more realism: occlusion, shading
  • speed: using hardware
Graphics Pipeline: Overview

1. **Application**
2. **Command Stream**
3. **Vertex Processing**
4. **Transformed Geometry**
5. **Rasterization**
6. **Fragments**
7. **Fragment Processing**
8. **Framebuffer Image**
9. **Display**

- You are here
- 3D transformations; shading
- Conversion of primitives to pixels
- Blending, compositing, shading
- User sees this
Application sends geometric primitives to renderer (e.g., to GPU)

Vertices are transformed to image space (we've done this part!)

Primitives are converted into pixel-shaped "fragments"; values are interpolated across primitives.

Fragments are shaded, blended, and composited to determine pixel colors.

Pixel colors written to the framebuffer appear on the screen.
Command Stream

Application sends geometric primitives to renderer (e.g., to GPU)

What primitives?

- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that’s all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved surfaces? Approximate them with triangles
- Trend over the decades: toward minimal primitives
  - simple, uniform, repetitive: good for parallelism
Vertex Processing

Vertices are transformed to image space (we've done this part!)

Missing piece:
Rasterization

• First job: enumerate the pixels covered by a primitive
  – which pixels fall inside triangle
  – includes "clipping" content outside view volume

• Second job: interpolate values across the primitive
  – e.g. colors computed at vertices
  – e.g. normals at vertices
  – e.g. texture coordinates
Fragment Processing

• Hidden surface removal (occlusion) - only the closest object is drawn
• Per-fragment shading:
  • determine color of the pixel based on a shading model
  • diffuse color might come from a texture
• Blending, compositing - e.g.:
  • anti-aliasing
  • transparency / alpha blending
Hidden Surface Removal

• Two motivations: realism and efficiency
Back face culling

• For closed shapes you will never see the inside
  therefore only draw surfaces that face the camera
  implement by checking $\mathbf{n} \cdot \mathbf{v}$
Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm:** draw them back-to-front
- Topological sort on the occlusion graph:
  - if A ever occludes B, it must come after B in the drawing order
Handling Occlusion

• What if multiple triangles are facing the viewer at different depths?

• **Painter's algorithm**: draw them back-to-front

• Topological sort on the occlusion graph:
  • if A ever occludes B, it must come after B in the drawing order
Handling Occlusion

- What if multiple triangles are facing the viewer at different depths?
- **Painter's algorithm**: draw them back-to-front
- Topological sort on the occlusion graph:
  - if A ever occludes B, it must come after B in the drawing order

Works great if the ordering is easy to find...
Handling Occlusion

• What if multiple triangles are facing the viewer at different depths?

• **Painter's algorithm:** draw them back-to-front

• Topological sort on the occlusion graph:
  • if A ever occludes B, it must come after B in the drawing order

Works great if the ordering is easy to find...

... but often it isn't.
The z buffer

- In many (most) applications maintaining a z sort is too expensive
  - changes all the time as the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - allocate extra channel per pixel to keep track of closest depth so far
  - when drawing, compare object’s depth to current closest depth and discard if greater
  - this works just like any other compositing operation
The z buffer

– another example of a memory-intensive brute force approach that works and has become the standard
Precision in z buffer

• The precision is distributed between the near and far clipping planes
  – this is why these planes have to exist
  – also why you can’t always just set them to very small and very large distances

• Generally use $z'$ (not world $z$) in z buffer
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space ≠ linear interp. in world (eye) space
Interpolating in projection

linear interp. in screen space $\neq$ linear interp. in world (eye) space