CS 378: Computer Game Technology

Beyond Meshes

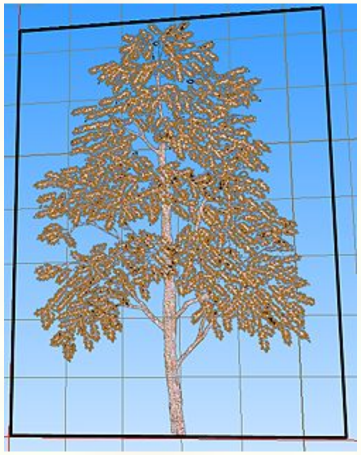
Spring 2012

Today

- Billboards
- Mesh Compression

Billboards

- A billboard is extreme LOD, reducing all the geometry to one or more textured polygons
  - Also considered a form of image-based rendering
  - Then again, all image-based rendering is about replacing geometry
- Issues in designing billboards
  - How are they generated?
  - How are they oriented with respect to the viewer?
  - How can they be improved?
- Also called *sprites*, but a sprite normally stays aligned parallel to the image plane



Generating Billboards

- By hand – a skilled artist does the work
  - Paints color and alpha
  - May generate a sequence of textures for animating
- Automatically:
  - Generate the billboard by rendering a complex model and capturing the image
  - Alpha can be automatically detected by looking for background pixels in the image (easier than blue-screen matting)
  - Can also blend out alpha at the boundary for good anti-aliasing

Billboard Configurations

- The billboard polygons can be laid out in different ways
  - Single rectangle
  - Two rectangles at right angles
  - Several rectangles about a common axis
  - Several rectangles stacked
- Issues are:
  - What sorts of billboards are good for what sorts of objects?
  - How is the billboard oriented with respect to the viewer?
  - How is the billboard rendered?

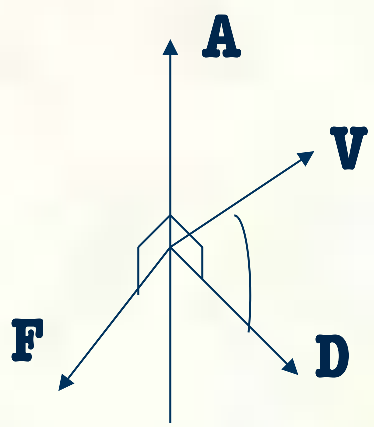
Single Polygon Billboards

- The billboard consists of a single textured polygon
- It must be pointed at the viewer, otherwise it would disappear when viewed from the side
  - Exception: Billboards that are walls, but then they are textured walls!
- Two primary ways of aligning the billboard:
  - Assign an `up` direction for the billboard, and always align it to face the viewer with `up` up
  - Assign an axis for the billboard and rotate it about the axis to face the viewer
- What sort of objects is this method good for, and why?
  - Consider: What will the viewer see as they move around the object?

Aligning a Billboard

- Assume the billboard has a known vector that points out from the face, and an `up` or `axis` vector
- All alignment is done with rotations, but which ones?
- Rotation about an axis:
  - We know what axis to rotate about. Which one?
  - How do we compute the angle through which to rotate?
- Facing the viewer and pointing up:
  - Best to break it into two rotations
  - Rotate about the world up vector. How much? To align what?
  - Then rotate about the apparent horizontal vector. To align what?

Alignment About Axis



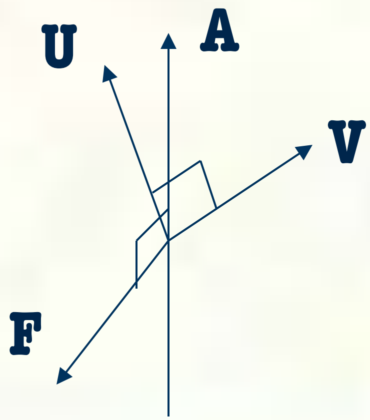
- **A** is axis for billboard, **V** is viewer direction, **F** is current forward, **D** is desired forward
- How do we compute **D**? $\boldsymbol{D} = \boldsymbol{A} \times (\boldsymbol{V} \times \boldsymbol{A})$
- How do we compute the angle, γ, between **F** and **D**?

$$\gamma = \cos^{-1}\left(\frac{\boldsymbol{F} \bullet \boldsymbol{D}}{\|\boldsymbol{F}\|\|\boldsymbol{D}\|}\right)$$

- There is a significant shortcut if **A** is the **z** axis, and **F** points along the **x** axis. What is it?

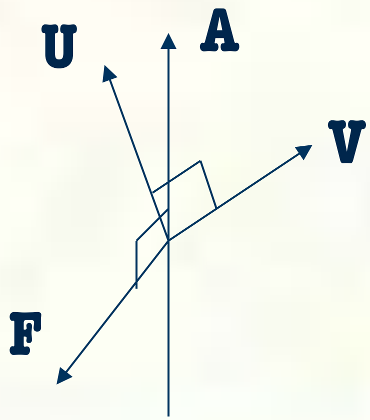
$$\gamma = \tan^{-1}\left(\frac{\boldsymbol{V}_y}{\boldsymbol{V}_x}\right)$$

Alignment To Point at Viewer



- **A** is axis for billboard, **V** is viewer direction, **F** is current forward, **U** is desired up vector
- Step 1: Align **F** and **V**. How?
  - Compute **F×V**
  - Direction is axis, magnitude is sin γ
- Step 2: Align **U**. How? Hint: previous slide
  - Desired **U=V×(A×V)**
  - Compute original **U** after rotating by Step 1
  - Rotate about **V** by angle computed using method on previous slide

Alignment To Point at Viewer



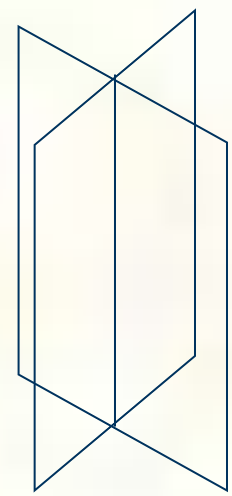
- Simpler method if the original forward direction is the **x** axis, and the original up direction is the **z** axis
- Form the rotation matrix directly (**A** and **V** unit vectors):

$$\boldsymbol{R} = \left[V \quad A \times V \quad V \times \left(A \times V \right) \right]$$

  - Each vector forms a column

Multi-Polygon Billboards

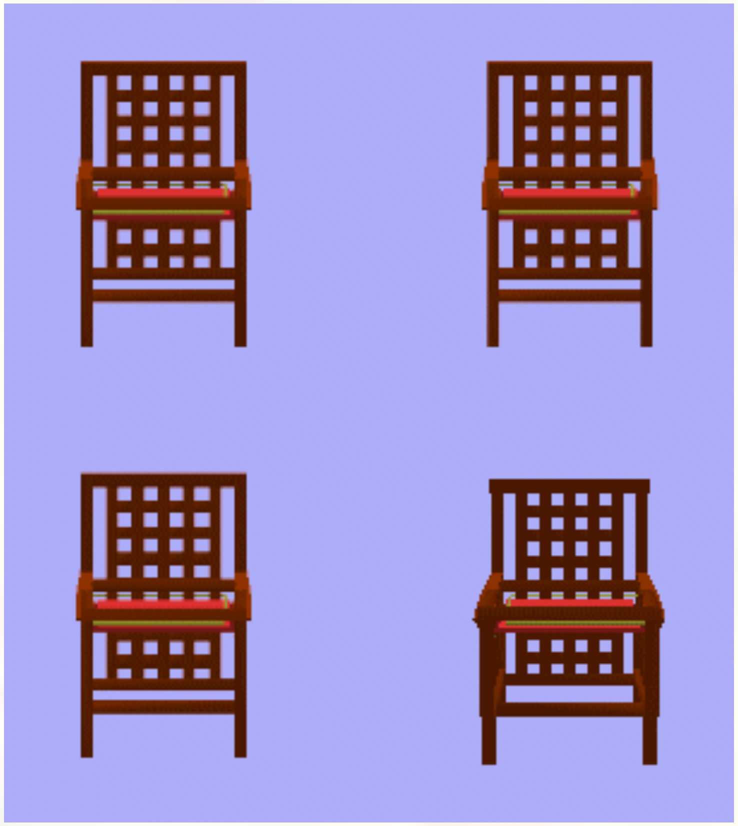
- Use two polygons at right angles:
  - No alignment with viewer
  - What is this good for?
  - How does the apparent width change with viewing angle?
- Use more polygons is desired for better appearance
  - How does it affect the apparent width?
- Rendering options: Blended or just depth buffered

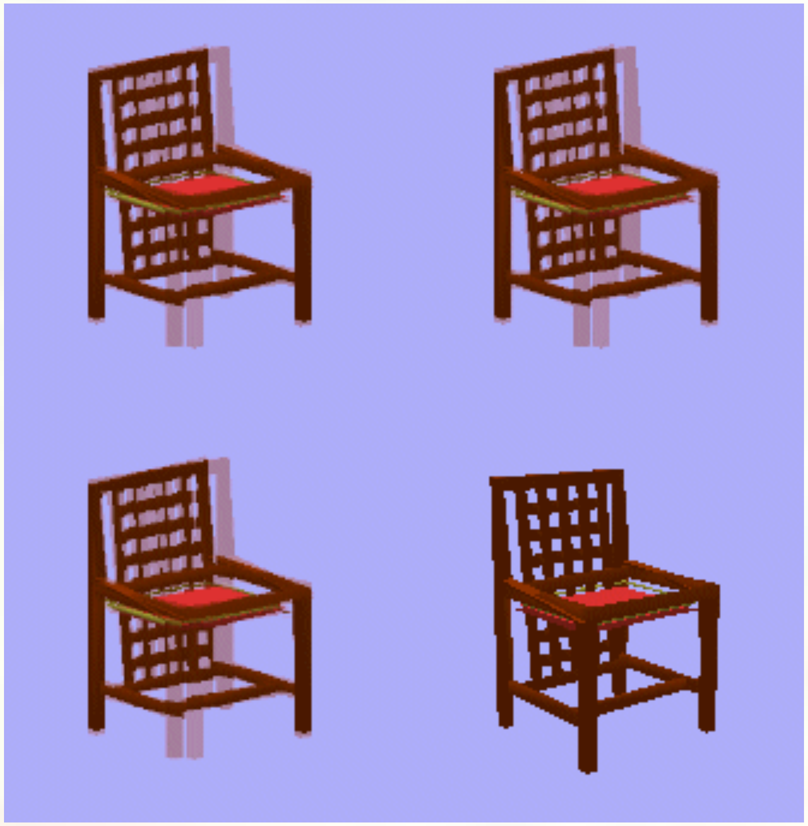


View Dependent Billboards

- What if the object is not rotationally symmetric?
  - Appearance should change from different viewing angles
- This can still be done with billboards:
  - Compute multiple textures each corresponding to a different view
  - Keep polygon fixed but vary texture according to viewer direction
  - Best: Interpolate, with texture blending, between the two nearest views
    - Can use 3D textures and hardware texture filtering to achieve good results
- Polygons are typically fixed in this approach, which restricts the viewing angles
  - Solution: Use more polygons each with a set of views associated with it

View Dependent Textures

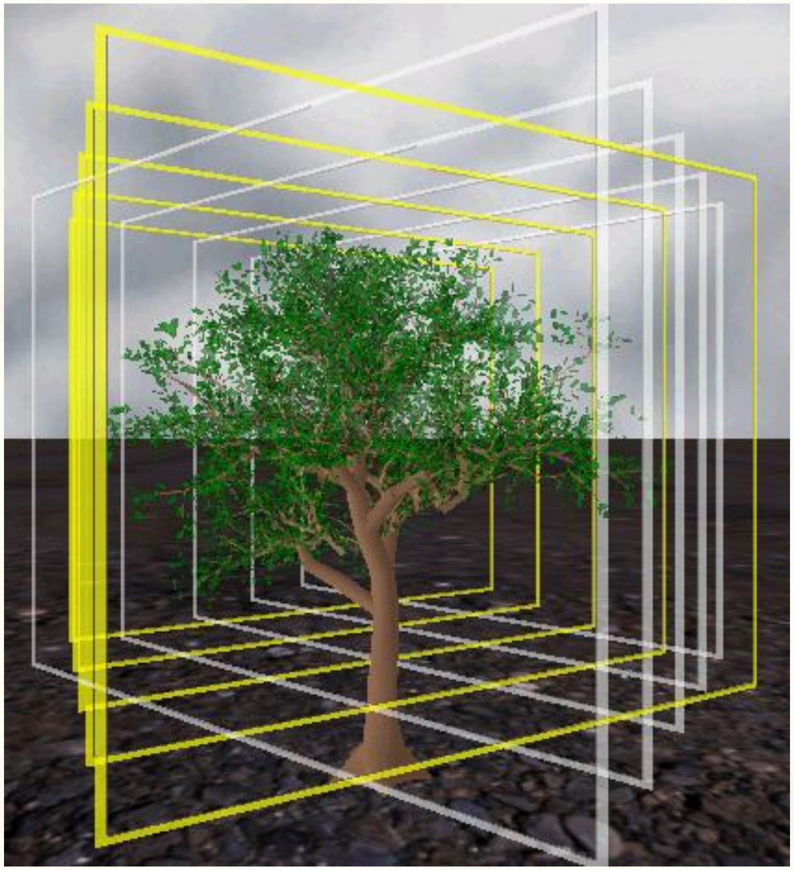




Screen shots from an Nvidia demo

Impostor Example

- Another methods uses slices from the original volume and blends them



Pipeline Efficiency

- The rendering pipeline is, as the name suggests, a pipeline
  - The slowest operation in the pipeline determines the throughput (the frame rate)
  - For graphics, that might be: memory bandwidth, transformations, clipping, rasterization, lighting, buffer fill, …
- Profiling tools exist to tell you which part of your pipeline is slowing you down
- Now we focus on reducing the complexity of the geometry
  - Impacts every part of the pipeline up to the fragment stage
    - Assumption: You will touch roughly the same pixels, even with simpler geometry

Reducing Geometry

- Assume we are living in a polygon mesh world
- Several strategies exist, with varying degrees of difficulty, reductions in complexity, and quality trade-offs:
  - Reduce the amount of data sent per triangle, but keep the number of triangles the same
  - Reduce the number of triangles by ignoring things that you know the viewer can't see – *visibility culling*
  - Reduce the number of triangles in view by reducing the quality (maybe) of the models – *level of detail (LOD)*

Compressing Meshes

- Base case: Three vertices per triangle with full vertex data (color, texture, normal, …)
- Much of this data is redundant:
  - Triangles share vertices
  - Vertices share colors and normals
  - Vertex data may be highly correlated
- Compression strategies seek to avoid sending redundant data
- Impact memory bandwidth, but not too much else
  - Of prime concern for transmitting models over a network

Compression Overview

- Use triangle strips to avoid sending vertex data more than once
  - Send a stream of vertices, and the API knows how to turn them into triangles
- Use vertex arrays
  - Tell the API what vertices will be used
  - Specify triangles by indexing into the array
  - Reduces cost per vertex, and also allows hardware to cache vertices and further reduce memory bandwidth
- Non-shared attributes, such as normal vectors, limit the effectiveness of some of these techniques

Mesh Compression

- Pipelined hardware typically accepts data in a stream, and has small buffers
  - Can't do de-compression that relies on holding the entire mesh, or any large data structure
- Network transmission has no such constraints
  - Can do decompression in software after downloading entire compressed mesh
- Typical strategies
  - Treat connectivity (which vertices go with which triangles) separately from vertex attributes (location, normal, …)
  - Build long strips or other implicit connectivity structures
  - Apply standard compression techniques to vertex attributes