Interactive Raytracing in the Cave Environment

Matt Ahrens (mahrens@cs.brown.edu)
Vinci Liu (vliu@cs.brown.edu)

May 5, 2000

Motivation

- There are two major types of rendering algorithms
  - Polygon-based renders
    - Currently used to render objects in the cave
    - Uses triangles to approximate an object
    - Hardware-accelerated (OpenGL)
    - Runs in real time (more or less :-)

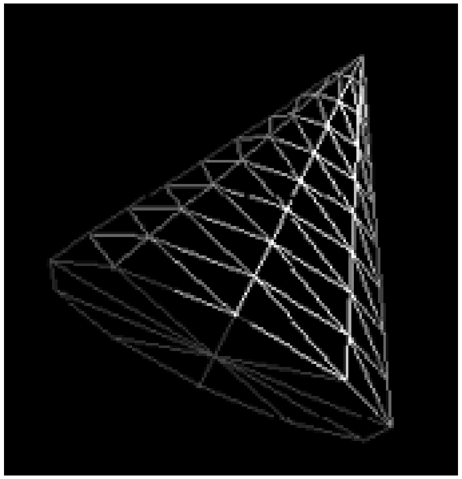
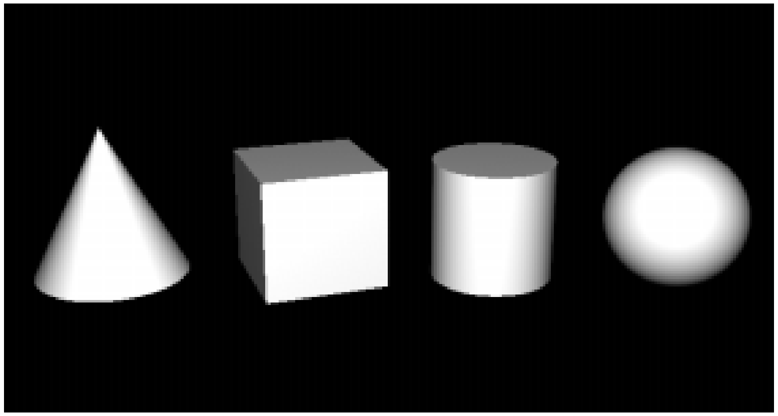


Figure 1: A cone approximated by many small triangles

Motivation

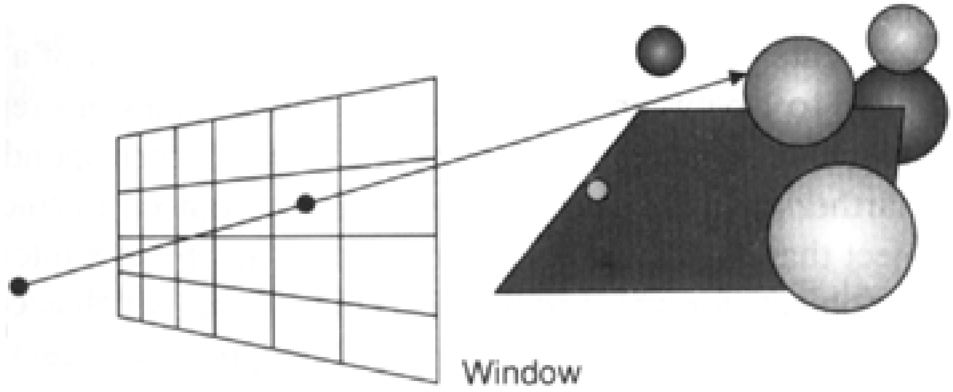
- Raytracer
  - What *we* used to render the cave
  - More realistic model
  - We can easily change the lighting model
  - Objects can have reflections, refractive transparency, etc.
  - Require more processing power
  - *Rarely* runs in real time



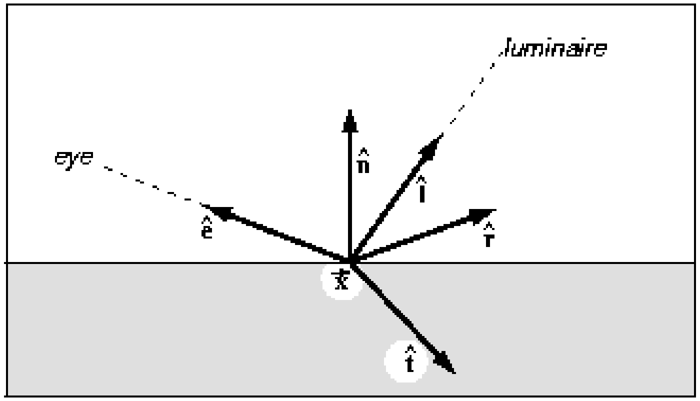
- (we also support triangulated meshes in our raytracer)

Raytracer

- Samples the observer's field of vision with light rays



- Color of each ray is dependent on secondary rays at the intersection point



Optimizations

- Based on work done at the University of Utah in 1999
  - **Interactive Ray Tracing** by Parker, Martin, Sloan, Shirley, Smits, and Hansen

  1. Bounding Volumes
  2. Rendering only the user's field of vision
  3. Scheduling (we need multiple processors to run a real time raytracer)

Scheduling

- Assume goal is to keep slave nodes constantly busy to maximize framerate

- But we have to do other things besides trace rays
  - Track position
  - Compute which rays to cast and transmit to slaves
  - Collect traced data from slave nodes
  - Assemble traced colors and display to screen

- Solution: parallelism

- Separate processes/threads running on separate CPUs for each of the above tasks (6 CPUs — one display CPU per wall)

- We keep the 54 slave nodes tracing rays $>$ 95% of the time!

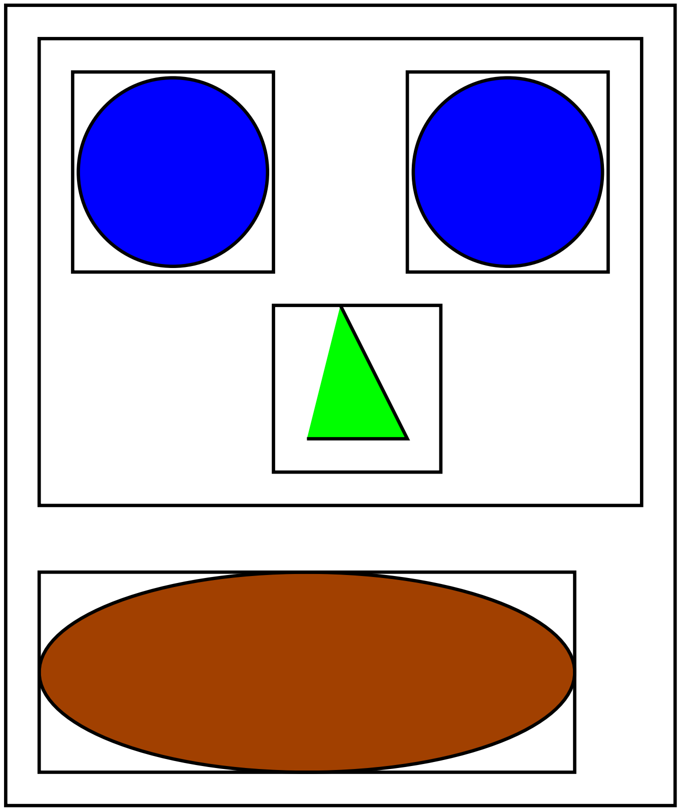
- Downside: increased latency

Which rays to cast

- Rendering only the User's Field of Vision
  - Uses VRPN (Virtual Reality Peripheral Network from Univ. of North Carolina) to interface with head tracker
- We cast fixed number of rays in fixed directions *relative to user's head*
- Result:
  - User perceives constant resolution
  - Performance increases, since we cast fewer rays
  - Less widely varying framerate, since we cast constant number of rays per frame
- Side note: We get our traced rays onto the screen by putting the data into an OpenGL texture map and drawing a 2D textured triangle mesh

Bounding Volumes

- N-ary tree of axis-aligned bounding volumes
- Leaf nodes are primitives contained by bounding volumes
- Interior nodes are bounding volumes which contain bounds of their children
- Bounding volumes are grouped spatially
  - Build the tree so as to minimize cost function
  - cost $= \sum_{nodes}$ surface area × number of children
  - **Automatic create of object hierarchies for ray tracing**, Goldsmith and Salmon, 1987
- If a ray misses the bounds of an interior node, then it misses all the node's children



- Implementation details
  - Each ray-bounding volume intersection test can be done with 2 subtracts, 2 multiplies, and 3 compares
  - Don't actually recursively traverse tree, make array with skip pointers which tell where to go if we miss

Results

- Resolution of 512 × 512 to 1024 × 512 rays per eye
- Using IBM SP2's 60 Power3 Processors, running in parallel
  - 6 master, 54 slave processors
- Scene files up to 40,000 primitives
  - With bounding boxes, runtime of a raytracer is sub-linear in the number of geometric primitives
  - We get 10-20 fps
- $512 \times 512 \times 2$ (for stereo) $\times$ 10-20 fps = 5-10 million rays per second!!!
- The maximum bandwidth of the IBM SP2 is 40MB/sec
  - Maximum frame-rate at $512 \times 512 \times 2$ is 26fps

Finally

- Special thanks to:
  - Andy Forsberg
  - Tim Rowley
- On to the DEMO!!!