Lecture 22: Ray Tracing Extensions

November 17, 2020

Revisit: Smoothing Approach

- Identify shared vertices
  - Lookup vertex numbers for a given triangle
- Compute true normal for every surface
  - Here assume A then B then C traversal
  - Compute the average normal at a vertex
    - Exclude adjacent faces too far off in orientation
- Use beta and gamma to interpolate normals

$$N_i = (1 - \beta - \gamma)\, N_A + \beta N_B + \gamma N_C$$

SageMath Notebook

File Notebook

File Edit View Insert Cell Kernel Help CPU: 98% Memory: 254MB Trusted SageMath 9.1

tab Code Validate

Smooth Shading of Polygonal Surfaces

Ross Beveridge, November 17, 2020

In this Notebook demonstrates flat and smooth shading of polygonal object meant to represent a curved surface object. Specifically a sphere made up of triangles. In addition to providing a final result demonstrating how flat shading appears, this code shows a coherent integrated example of how to incorporate objects, made up of triangles, into the ray tracing framework alread established in previous notebooks.

One WARNING - this code is not fast to run. Rendering at 256 by 256 takes about a half hour!

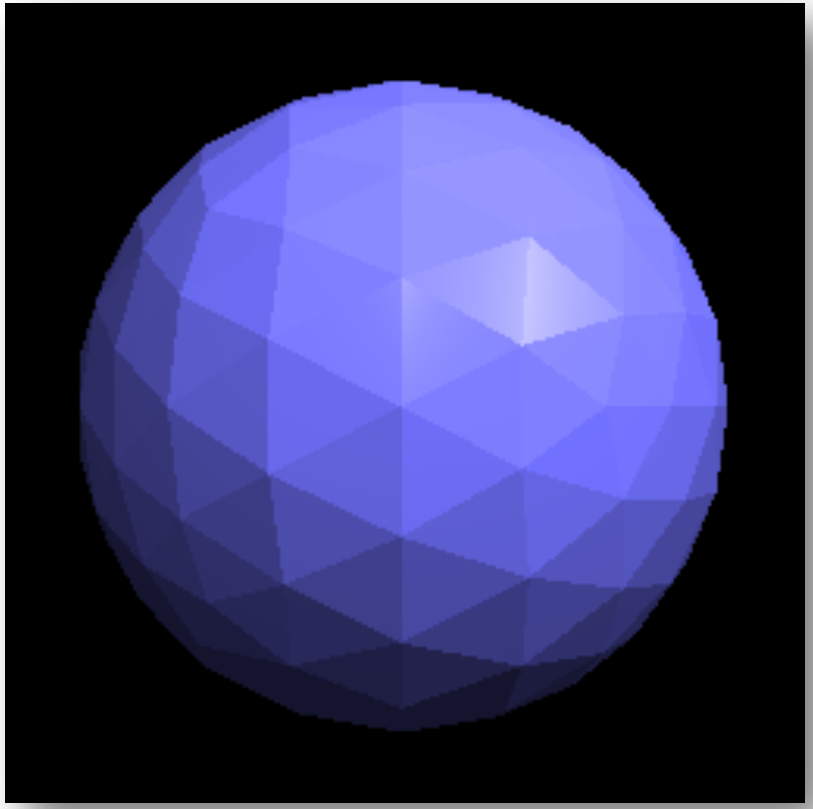
In [332]:

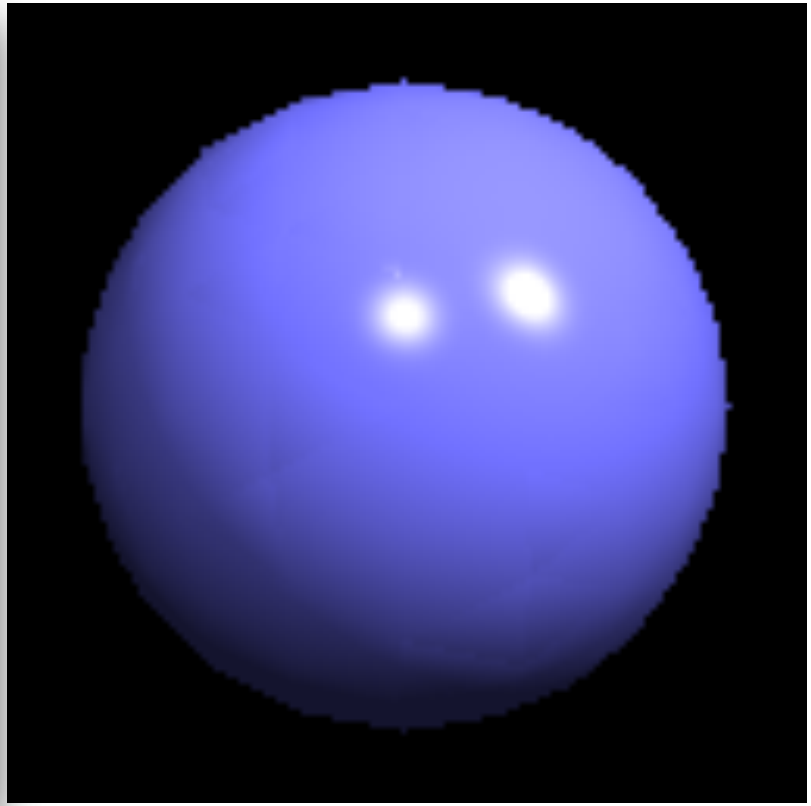
```
# Basic Startup Book Keeping
import numpy as np
import re
```

0.005 seconds

Complete implementation of smoothing and interpolation with an angle cutoff.

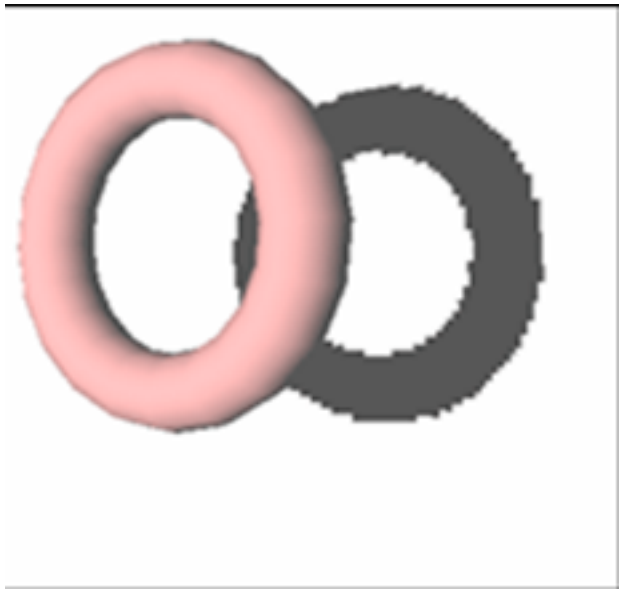
Flat Versus Smooth





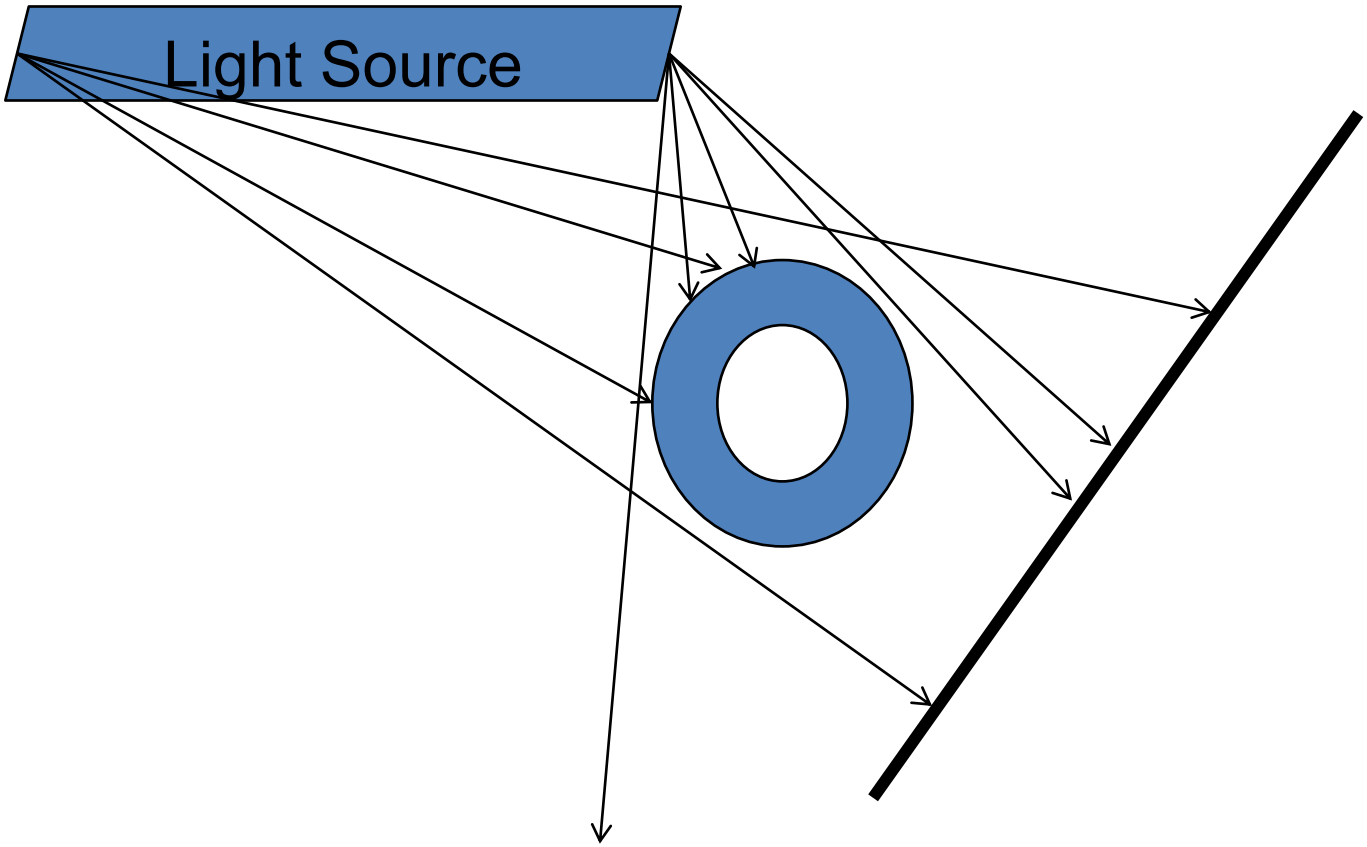
Extension #1: Extended light sources

- Point light sources give rise to excessively sharp shadows



First – Revisit Smoothing

Extended Light Sources (II)



Extended (III)

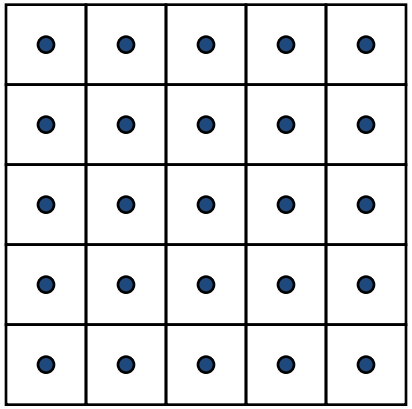
- Extended light sources produce softer shadows
- But where is the vector L?
- Model Extended light sources as a polygon
- Randomly sample points over the polygon
  - Throw a ray L to each point on the light source
  - Divide the light source brightness by the number of rays

Pseudo-code (extended light sources)

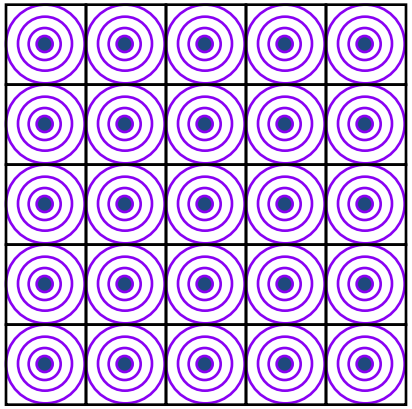
```
Reflectance(ray, world, ksproduct) {
  poi = intersect(ray, world)        // intersects with every polygon
  I = K_d B_A                        // ambient lighting
  for every light source
    for every point on light source   // this loop is the extension
      throw ray to light source
      if not shadowed
        I += K_d B_ls (N·L) + k_s (V·R)^α
  ksproduct *= ks;
  if (ksproduct > threshold)
    I += Reflectance(Reflect(ray), world, ksproduct)   // recursion for
reflection
    // recursion for refraction
    if (!opaque) I += Reflectance(Refract(ray), world, ksproduct)
  return I
}
```

The Ugly Truth About Pixels

Ray tracing (as described so far) treats pixels as points on the image plane



Simple Pixel Center

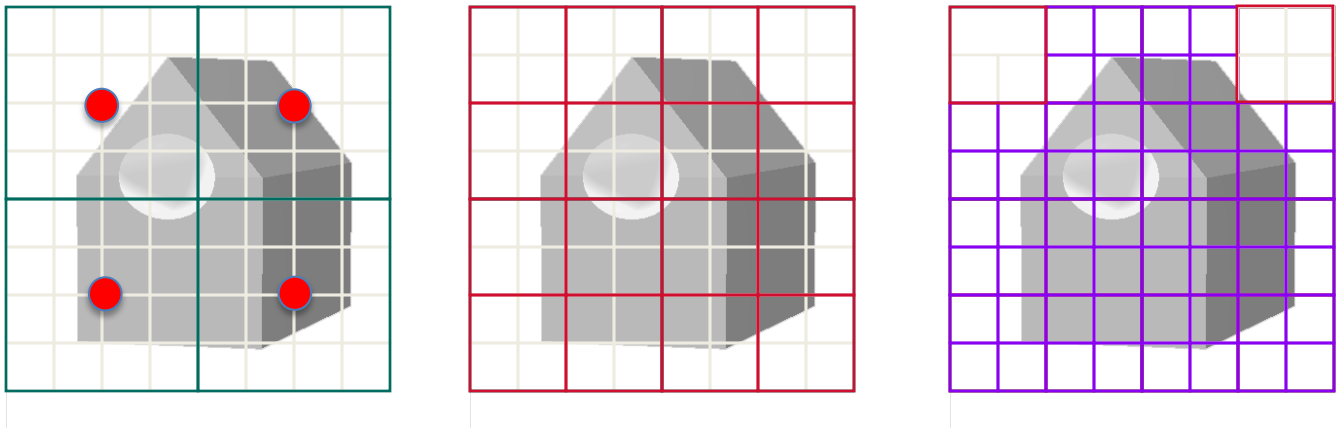


Weighted Area Sample

Pixels actually sample light over some *area*

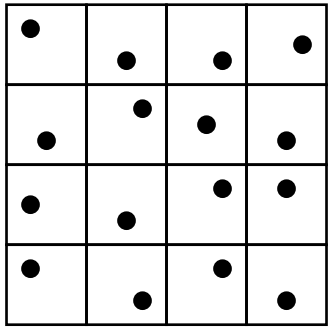
Uniform Adaptive Sampling

- Sample middle of pixel quadrants.
- If values are nearly equal, use their average.
- Else, subdivide into four sub pixels and repeat.



Stochastic Sampling

- Stochastic sampling for distributing rays
  - Divide the pixel into N subpixels
    - 16 subpixels are shown
  - Uniformly draw a point from within each
  - Average the result
- Here is an example:



Pseudo-code (uniform adaptive sampling)

```
For every pixel {
  image[i,j] =
        PixelReflectance(i+0.5, j+0.5,
                                camera, world, 0.25)
}
```

Wait… that was too easy!

Pseudo-code (uniform adaptive sampling) II

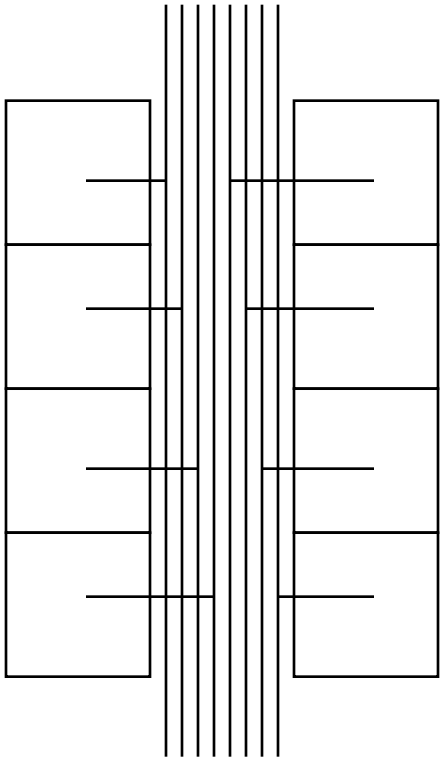
```
PixelReflectance(I, j, camera, world, scale){
  // Reflectance is previous pseudo-code function
  v1 = Reflectance(Ray(i+scale, j+scale), world, 1.0)
  v2 = Reflectance(Ray(i-scale, j+scale), world, 1.0)
  v3 = Reflectance(Ray(i+scale, j-scale), world, 1.0)
  v4 = Reflectance(Ray(i-scale, j-scale), world, 1.0)
   if (v1 ≅ v2 ≅ v3 ≅ v4) return (v1 + 2 + v3 + v4) / 4
  else return // divide and recurse
     (PixelReflectance(i+scale, j+scale,camera, world,scale)
     + PixelReflectance(i-scale, j+scale,camera, world,scale)
     + PixelReflectance(i+scale, j-scale,camera, world,scale)
     + PixelReflectance(i-scale, j-scale,camera, world,scale) ) / 4
}
```

Even Uglier Truths About Digital Pixels (but don't model these)

Gaps between columns (or rows) for wiring space

Leads to “non-square” pixels, dead zones

If pixel is overexposed, value is “clipped” to max, but excess current may “bloom” neighboring pixels



Efficiency

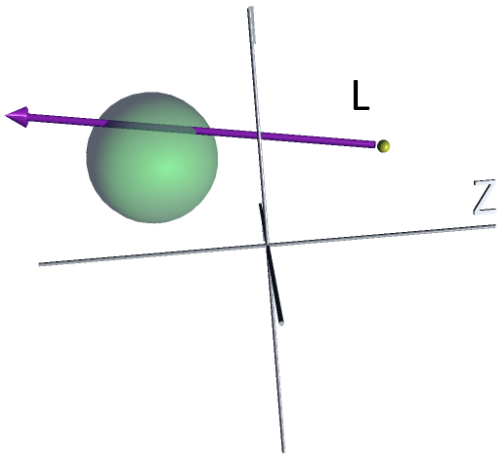
- Every extension throws more rays
  - Some in loops
    - Extended light sources
    - Thin lens model
  - Some recursively
    - Translucency
    - Adaptive sampling
- The cost of ray tracing still lies in computing intersections
  - Don’t believe me? Valgrind your ray tracer
- Efficient ray tracing uses data structures to reduce the number of intersections

An easy, small optimization

- In the loop that intersects a ray with a polygon:
  - Pass in the smallest positive t value so far
  - For polygons, if the t value of the intersection of the plane & ray is larger, skip the inside/outside test.

Sphere Intersection : review

- **Consider a sphere centered at P_c with radius r.**
  - **Section 4.4.1 of your text**



$$\left|P - P_c\right|^2 = r^2$$

$$\left|L + sU - P_c\right|^2 - r^2 = 0$$

substitute

$$T = P_c - L$$

Yielding a quadratic equation

$$(sU - T)\cdot(sU - T) - r^2 = 0$$

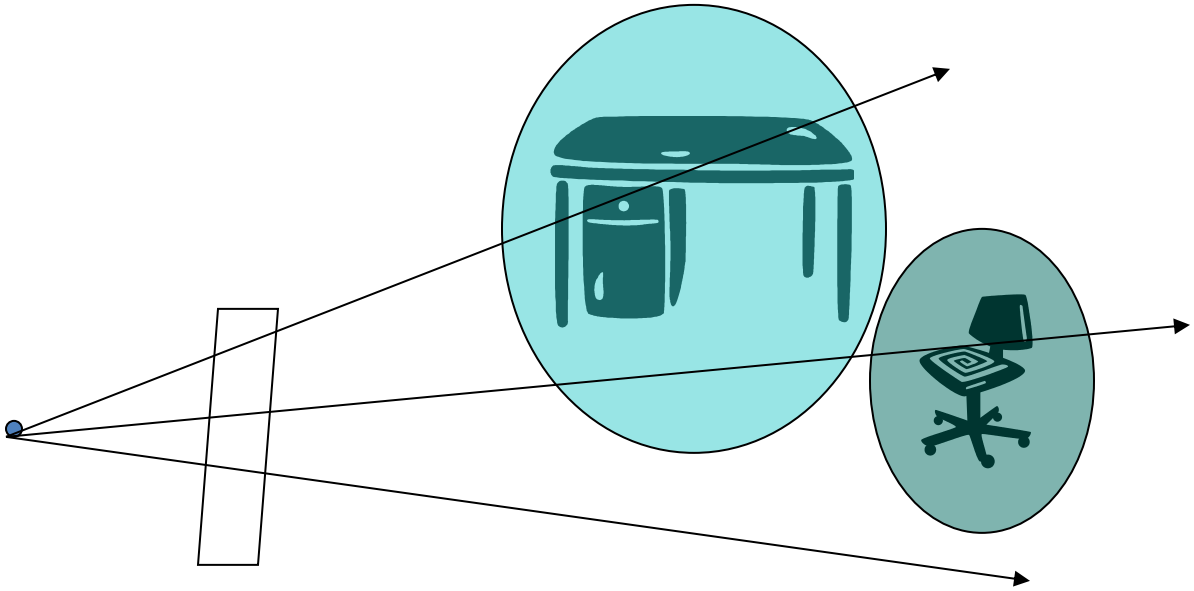
$$s^2 - 2(U\cdot T)s + T\cdot T - r^2 = 0$$

To find s (called t elsewhere):

$$s = (U\cdot T) \pm \sqrt{(U\cdot T) - T^2 + r^2}$$

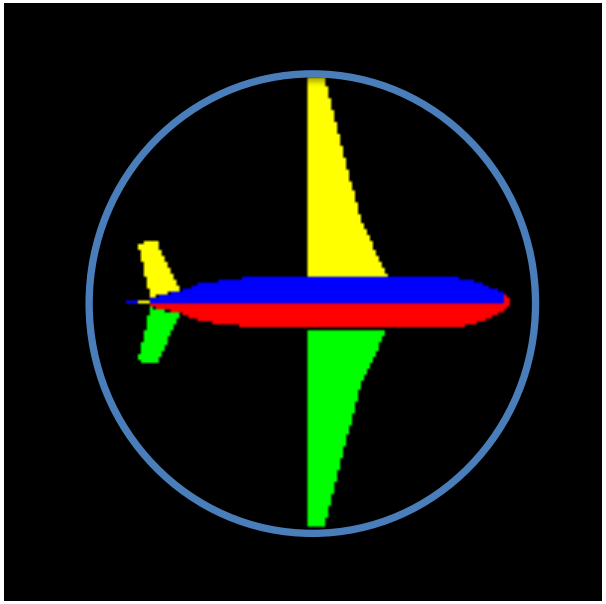
Bounding Volumes

- The simplest effective technique is to throw a sphere around “objects” and only test object surfaces if the ray intersects the sphere



Hierarchical Example

- One sphere around airplane
- Look how many pixels have no ray/polygon intersections!



Bounding Volumes II

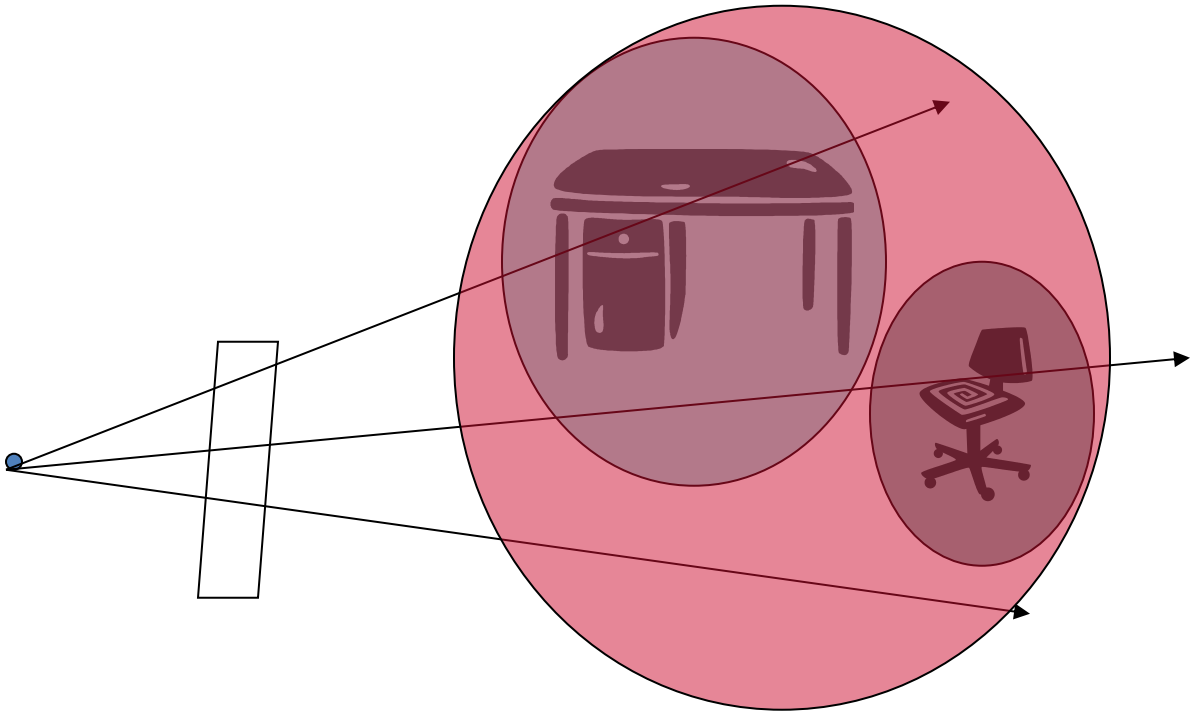
- Simple
- Remarkably effective – most rays miss most spheres (or have negative intersections)
- Fast – see previous notes on sphere intersection

Implementing Bounding Volumes

- Objects are the key
  - Define a ray/object intersection method
  - It intersects the ray with the bounding sphere
  - If the ray intersects the sphere, it intersects the ray with all the surfaces in the object
    - Return the smallest positive S
    - Return the surface as well

Hierarchical Bounding Volumes

- You can do this recursively, reducing the complexity further:

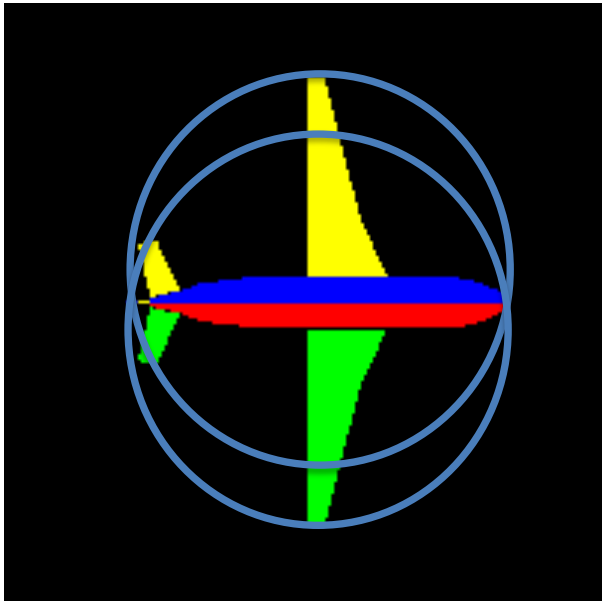


Hierarchical Bounding Volumes (Beyond Objects)

- Build a top-down hierarchy
  - Compute the bounding box of your polygons
  - Select the greatest dimension
  - Split vertices 50/50 along the dimension
  - Create two spheres
  - Recurse
  - For leaf spheres, assign a polygon to a sphere if they intersect

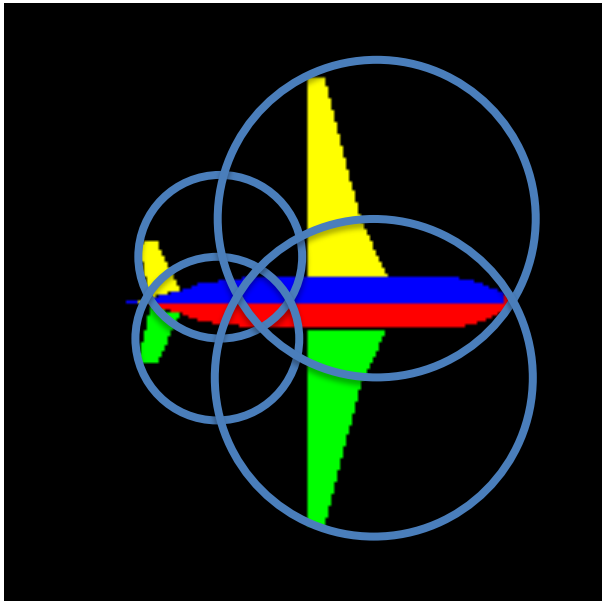
Hierarchical Example

- First “greatest” dimension of airplane is along the wings of the plane (too bad)
- But your vertices split 50/50



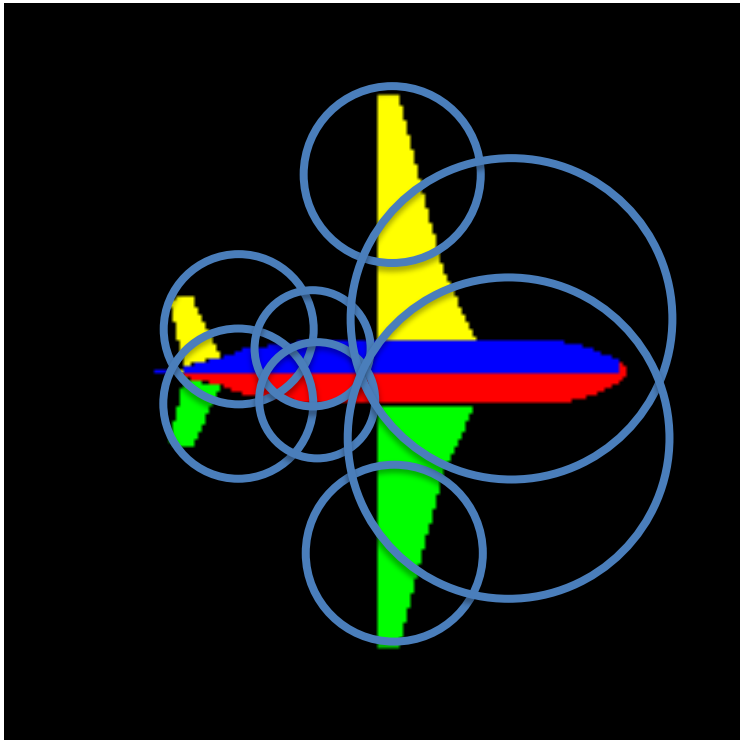
Hierarchical Example II

- Now recurse, splitting the sets of vertices



Hierarchical Example II

- One more time….



Polygon/Sphere (abstract)

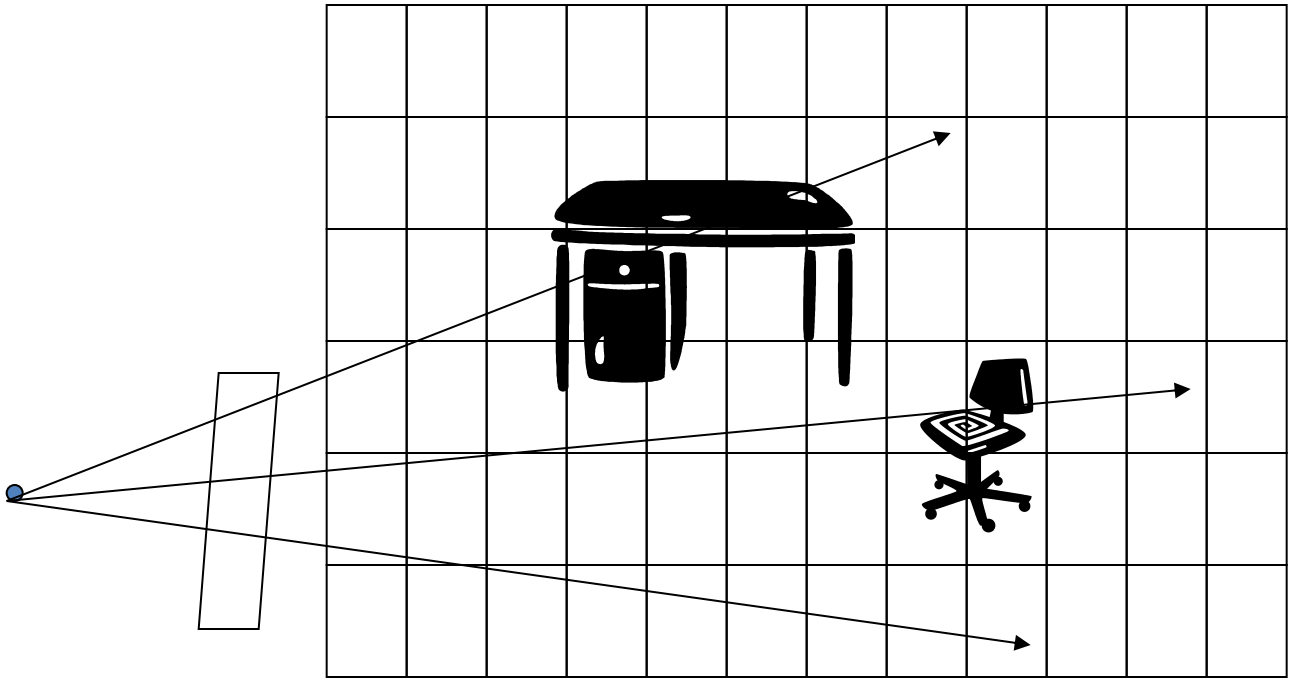
- If the sphere does not intersect the plane of the polygon, it doesn't intersect the polygon
- Otherwise, let the circle be the intersection of the sphere and plane
- For the sphere & polygon to intersect, one of the following must be true
  - The circle is entirely inside the polygon
  - The polygon is entirely inside the circle
  - The circle (and sphere) intersects an edge of the polygon

Polygon/Sphere intersection (alg)

- Test if sphere intersects plane of polygon
  - Plug sphere center into f(ax+by+cz+d) -> m
  - If |m| < r, the sphere intersects the plane.
- Test if center of sphere projects inside polygon
  - Poi = C + mN, C = center of sphere, N is plane normal
- Test if vertex of polygon is inside sphere
  - $(x-cx)^2 + (y-cy)^2 < r^2$
- Test if sphere intersects polygon boundaries
  - This is just ray/sphere intersection
    - Ray begins at one vertex
    - Value of t must be less than distance to other vertex
  - If yes, sphere intersects polygon

Uniform Spatial Partition

- Put 3D grid over space, link surfaces to grid cells they intersect.



Boxes Lead to Octree

en.wikipedia.org/wiki/Octree

Octree - Wikipedia

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute

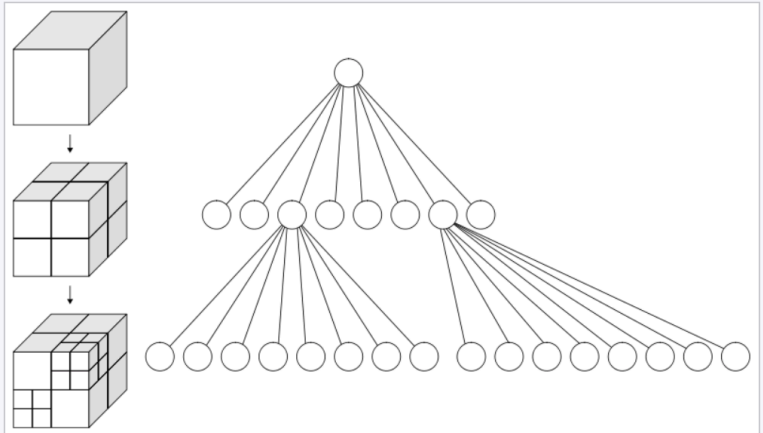
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools

Octree

From Wikipedia, the free encyclopedia

An **octree** is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants. Octrees are the three-dimensional analog of quadtrees. The name is formed from *oct* + *tree*, but note that it is normally written "*octree*" with only one "t". Octrees are often used in 3D graphics and 3D game engines.



Left: Recursive subdivision of a cube into octants. Right: The corresponding octree.

Contents [hide]

1 For spatial representation
2 History
3 Common uses