Bounding Volumes

Motivation

- We've discussed raytracing and some general speedups
- We obtained some interesting speedups
  - 16.60 seconds originally
  - 8.52 seconds after changes
- But: $\frac{1}{8}$ frames per second is not real-time!

Motivation

- Today, we'll look at other speedup techniques specialized for *spatial location*
- These can be applied in a variety of areas
  - Ray tracing (of course)
  - Collision detection
  - Physics simulations (gravity, wind, proximity-based effects)
  - User interface/GUI (picking)

Problem

- A considerable amount of work we're doing is wasted
- Let's add some statistical output to the raytracer. What could be useful to know?

Output

- How many rays were cast
- How many intersection tests were performed
- How many rays intersected something

Results

- Here's what I got:
- ```
  8.624 seconds
  Rays cast: 262,144
  Tests performed: 805,306,368
  Rays that missed everything: 242,473
  Rays that hit something: 19,671 = 7%
  ```
- 93% of our rays represent wasted work!
  ```

Idea

- Reduce the amount of tests we do
  - Surround the object with something that's quick & easy to test for intersection
  - If ray misses *bounding volume* then it can't possibly hit the object
- What kind of structure should we use for bounding volume?

Bounding Volume

- Several common choices:
  - Sphere
  - Axis aligned bounding box
  - Oriented bounding box
  - Slabs
- Roughly arranged from fastest test to slowest test
- But: Spheres are bad for long/thin objects
- AABB's bad for thin objects that are not facing $\pm X$, $\pm Y$, $\pm Z$ direction

Sphere

- We'll start with spheres since we already know how to use them
- Question: How to compute bounding sphere for an object?
  - Several methods exist
  - Easy one: Compute bounding box for points
    - Then let sphere center = centroid of box
    - Let sphere radius = distance to furthest point from center

Results

- The results are very impressive:

```
0.895276 seconds
Rays cast: 262,144
Tests performed: 87,330,816
Rays that missed everything: 242,473
Rays that hit something: 19,671 = 69%
Bounding volume hits: 28,428
Bounding volume misses: 233,716
```

Analysis

- Is this good enough for us to wrap things up and call it a day?

Analysis

- Is this good enough for us to wrap things up and call it a day?
  - Of course not!
    - Don't be silly!

Analysis

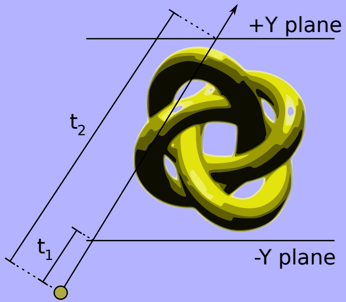
- There's still quite a bit of room for improvement
  - 0.9 sec/frame is not interactive
  - 31% of the rays that hit the bounding sphere miss the underlying object
  - If a ray hits the bounding box, we still test it against every single point on the mesh

Bounding Box

- What if we look at AABB’s (axis aligned bounding boxes)
- We need a quick ray-box intersection test
  - We don’t care where on the box the intersection is

Box

- Axis aligned box can be defined by six planes: One for each of $\pm$\{X,Y,Z\} directions
- Shown: The $\pm$Y planes
  - Want to know distance along ray where we hit each plane
  - This gives us a range: (t_1, t_2)
  - Repeat for X & Z



Test

- Let t_1' = the *largest* of the three t_1 values found
- Let t_2'= the *smallest* of the three t_2 values found
- Ray intersects box iff $t_1' \leq t_2'$

Box

- Suppose we have a function:

```
float rayPlaneIntersection( const vec3& N, float D, vec3& s, vec3&
    v)
{
    float denom = dot(N,v);
    //if denom is zero, we get t=infinity
    float numer = -(D + dot(N,s) );
    float t = numer/denom;
    return t;
}
```

Box

- Now we can write another function

```
void planePairIntersection( const vec4& p1, const vec4& p2, vec3&
    s, vec3& v, float& t1, float& t2){
     t1 = rayPlaneIntersection( p1.xyz(), p1.w, s,v );
     t2 = rayPlaneIntersection( p2.xyz(), p2.w, s,v );
     if( t1 > t2 ){
         float tmp = t1;
         t1=t2;
         t2=tmp;
     }
}
```

- This uses a vec4 to hold the plane A,B,C,D values

Box

- Finally, we can write our box test

```
//planes 0,1 = x min/max    planes 2,3 = y min/max
//planes 4,5 = z min/max
bool rayBoxIntersection(array<vec4,6>& planes, vec3& s, vec3& v ){
    using namespace std;
    float tx1,tx2;
    planePairIntersection(planes[0], planes[1], s,v, tx1, tx2 );
    float ty1,ty2;
    planePairIntersection(planes[2], planes[3], s,v, ty1, ty2 );
    float tz1,tz2;
    planePairIntersection(planes[4], planes[5], s,v, tz1, tz2 );
    float tmin = max(tx1,max(ty1,tz1));
    float tmax = min(tx2,min(ty2,tz2));
    return tmin <= tmax;
}
```

Results

- Here's what I got:

```
1.26347 seconds
Rays cast: 262,144
Tests performed: 113,246,208
Rays that missed everything: 242,473
Rays that hit something: 19,671 = 53%
Bounding volume hits: 36,864
Bounding volume misses: 225,280
```

- For this object, the bounding sphere is a better choice

Problem

- No matter what type of bounding primitive we use, it won't be good enough by itself
  - If bounding box is hit: We must test every face of the object
- Better: Use a *hierarchy* of bounding volumes
- This is a common pattern: Divide & Conquer: You see this applied *everywhere* in algorithm design

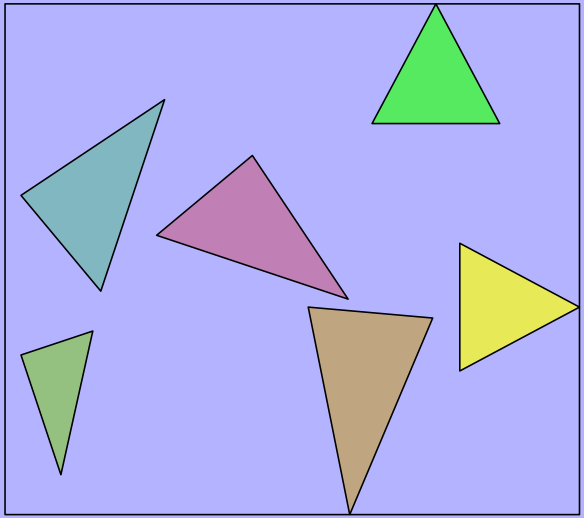
Octree

- Octree is probably the easiest to explain, so we'll start here
- Well, we'll start with its 2D analogue: The *quadtree*
- Define a function split():

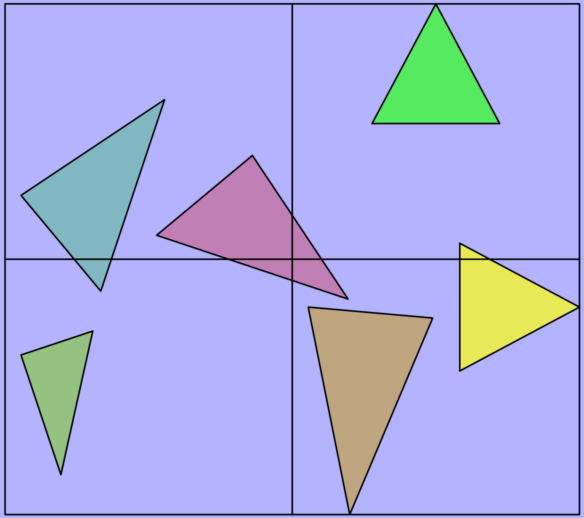
```
def split(box):
    if more than one thing in box:
        split box in fourths
        partition out things to each sub-box
        call split() on each sub-box
```

- Create a box around the whole scene and call split() on it

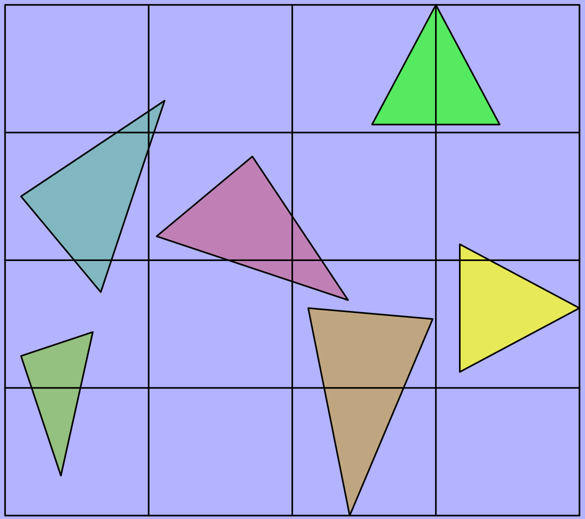
Example



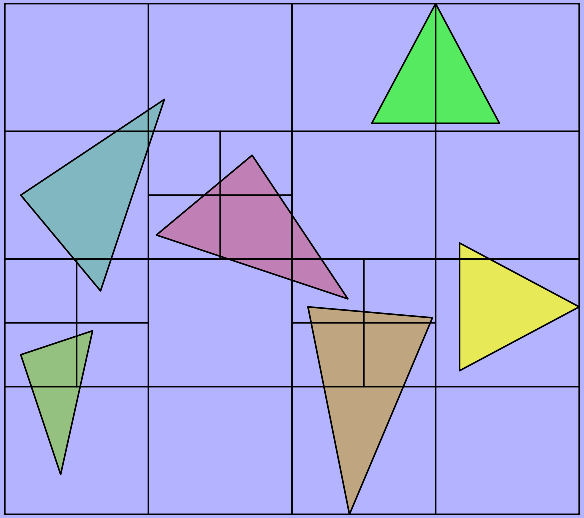
Example



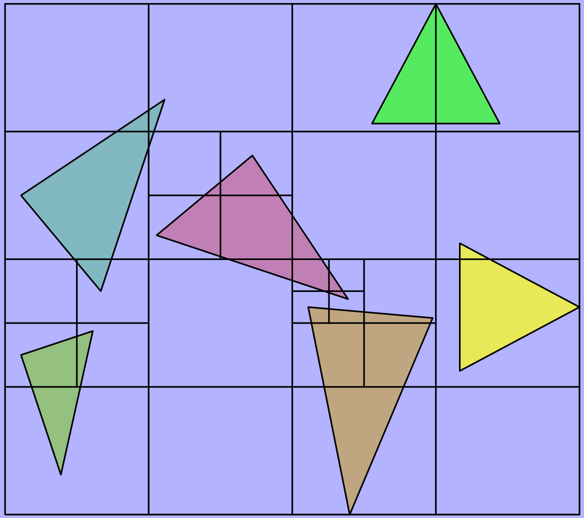
Example



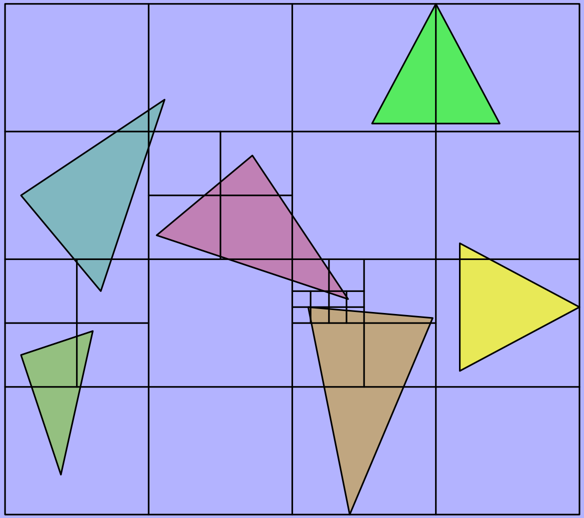
Example



Example



Example



Note

- We might terminate subdivision before we have just one triangle per node
  - Want to balance memory usage + time spent traversing hierarchy with the expected speedup

Build

- Note: On 64 bit platforms, pointers are 8 bytes
- But we usually have < 2^{32} nodes
- So we can use 32 bit integer array indices to represent nodes
  - Saves space (and cache)
- If we have fewer than 2^{16} nodes, we can use 16 bit indices (even better cache usage)

Octree Node

```
class OctreeNode{
  public:
    vec3 min, max;
    std::array<unsigned,8> children;
    std::vector<Triangle> triangles; //only used for leaf
    std::array<vec4,6> planes;       //six planes of this node
    static vector<OctreeNode> nodes;    //all nodes
}
```

Construction

- Initializing an octree node:

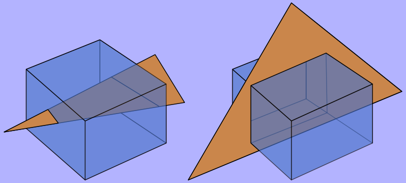
```
void initialize(int depth, vector<Triangle>& tris){
    if( depth >= MAXDEPTH || tris.size() <= MAXTRIS ){
        this->triangles = tris;
        set children[0...7] to 0
    } else {
        idx = nodes.size();
        add 8 child to nodes[]
        vector<Triangle> TV[8];
        for(Triangle& T : tris ){
            for(int j=0;j<8;++j){
                if( nodes[idx+j].contains(T))
                    TV[j].push_back(T);
            }
        }
        for(int j=0;j<8;++j)
            nodes[idx+j].initialize(TV[j]);
    }
}
```

Contains

- How do we know if an octree node contains a triangle?
- (Don't peek ahead!)

Contains

- One of these three cases must be true:
  - At least one point of triangle is inside node
  - At least one edge of triangle intersects node face
  - At least one edge of node intersects triangle



Construction

- contains() test then looks like this:

```
bool contains( Triangle& T){
    for(i=0;i<3;++i){
        if( pointInBox(T.p[i], this) )
            return true;
    }
    for(i=0;i<3;++i){
        if( segmentBoxIntersection( T.p[i], T.p[(i+1)%3], this) )
            return true;
    }
    for(all twelve edges e of this){
        if( segmentTriangleIntersection(e,T) )
            return true;
    }
    return false;
}
```

Note

- Which is faster: ray-sphere tests or ray-box tests?
- Only way to know is to benchmark it!
- I benchmarked just ray-bounding volume intersections
  - Ray-sphere:

    ```
    0.0869179 seconds
    Rays cast: 4,194,304
    ```

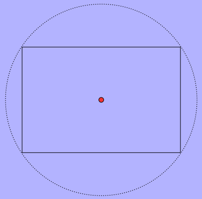
  - Ray-box:

    ```
    0.153369 seconds
    Rays cast: 4,194,304
    ```

- This implies that spheres would be faster

Approach

- We can build the octree as usual and then convert the nodes to spheres
  - Sphere center = octree cell center
  - Sphere radius = distance to furthest corner of cell
- 2D example (The further it is from square, the more “slop” space):



Intersection

- Performing ray intersection test is straightforward: Suppose we have octree node N
  - If N is a leaf: Test ray against geometry “owned” by N; return closest intersection
  - Test ray against box N. If it misses, return
  - Else, recursively test intersection against each child of N. Retain closest intersection

Code

```
//ray starts at s and goes in direction v
bool trace(const vec3& s, const vec3& v, unsigned nodeIndex, float
    & t){
    OctreeNode& node = OctreeNode::nodes[nodeIndex];
    if( rayBoxIntersection( node.planes, s, v )  ){
        if( node.children.empty() ){    //a leaf
            for(Triangle& T : node.triangles ){
                float t1;
                if( rayTriangleIntersection(T,s,v,t1) && t1 < t )
                    t=t1;
            }
        } else {
            for(unsigned childIndex : node.children )
                trace(s, v, childIndex, t);
        }
    }
}
```

Problem

- Recursion can incur speed penalty
  - Need to save registers, push return address (and maybe parameters) to stack, transfer control
  - When returning, need to clean stack and restore registers
- We can always transform recursive logic to iterative logic

Example

- Example of converting recursive to iterative: Recall *binary tree preorder traversal:*

```
struct Node{
    Node *left, *right;
    int data;
}
void traverse(Node* n){
    cout << n->data << "\n";
    if( n->left) traverse(n->left);
    if( n->right) traverse(n->right);
}
```

Example

- We can convert to iterative using an explicit stack:

```
void traverse(Node* root){
    stack<Node*> stk;
    stk.push(root);
    while( !stk.empty() ){
        Node* n = stk.pop();
        cout << n->data << "\n";
        if( n->right ) stk.push(n->right);
        if( n->left ) stk.push(n->left);
    }
}
```

- Notice we need to push right then left to maintain same traversal order
- Apply same idea to the octree-traversal code

Results

- Enough talk! Let’s see some results!
- I used boxes (i.e., not spheres), max of 1 triangle per leaf
- I didn’t include preprocessing time (which was about 0.08 sec)

Results

- Remember: Original time was 16.6 seconds; 8.52 seconds after we applied algorithmic speedups; 0.89 seconds after single-level bounding sphere

Octree Depth	Octree Nodes	Time (sec)
0	1	1.22
1	9	0.302
2	73	0.121
3	553	0.090
4	3529	0.102
5	17849	0.140

Results

- What if we use spheres instead of boxes?
- I tested with max depth of 4: 70.96 seconds!
  - Most likely because too much “slack” space

Assignment

- Implement octree-based raytracing acceleration.

Sources

- https://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part2
- Christer Ericson. Real Time Collision Detection. CRC Press.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org