Acceleration Algorithms in Computer Graphics

Ivan Xu
Alfred Lam
How can we render hundreds of millions of triangles in real time?
source data provided by and used with permission of the Boeing Company.
Spatial Partitions

*Efficiently locate objects by storing them in a data structure organized by their positions*
Grid

- The most naive but commonly used structure (square, rect, hex...)
  - Simpler
  - Constant Memory usage
  - Faster to update
  - ...

Binary Space Partitioning

- It handles empty space more efficiently
- densely populated areas more efficiently
- \( O(\log(N)) \)
bsp_tree(poly* current_poly)
{
    while(still_polygons)
    {
        partition_polygons(current_poly);
    }
    bsp_tree(current_poly->left);
    bsp_tree(current_poly->right);
}

traverse_tree(bsp_tree* tree, point eye)
{
    location = tree->find_location(eye);

    if(tree->empty())
        return;

    if(location > 0)  // if eye infront of location
    {
        traverse_tree(tree->back, eye);
        display(tree->polygon_list);
        traverse_tree(tree->front, eye);
    }
    else if(location < 0)  // eye behind location
    {
        traverse_tree(tree->front, eye);
        display(tree->polygon_list);
        traverse_tree(tree->back, eye);
    }
    else                  // eye coincidental with partition hyperplane
    {
        traverse_tree(tree->front, eye);
        traverse_tree(tree->back, eye);
    }
}
Quadtree
Quadtree

To update objects:

- remove and re-insert
Octree
K-d Tree

K-d Tree
BVHs

- Utilize bounding volumes of space to detect intersection with ray
- Simpler shapes (spheres, AABB boxes) make for faster intersection detection
- Can also be used for collision detection and culling later on
Culling Algorithms
Culling

- “The fastest triangle to render is the one never sent to the GPU”
- Happens per **object** not per **polygon**
- The **earlier** in the rendering process it is done, the better the performance improvements
Distance Culling

- Simplest form of culling
- Check if an object is close enough to the viewpoint
- Remove it if not
- Can set max/min distance culling based on object size in Unreal Engine
View Frustum Culling

- Remove objects that are not within the field of view

- Can use spatial data structures like bounding volume hierarchies to test for intersection with view frustum

- If bounding volume is **entirely** inside frustum, then **everything** inside is rendered.

- If bounding volume **intersects** the frustum, continue testing its children.
Back Face Culling

- Remove triangles facing away from the viewer
- Determine face direction by taking dot product of vector from triangle to viewpoint, and the triangle’s normal vector - if dot product is negative, it is facing away
- Often just a switch that can be turned on/off in GPUs today - “glEnable(GL_CULL_FACE)”
Occlusion Culling

- The most computationally expensive (checks visibility of every object), so done last in the pipeline
- Occlusion highly dependent on order of drawn objects as well as distance from viewpoint to object - “a matchbox can obscure the Golden Gate Bridge if it’s close enough”
Occlusion Culling can be extremely effective, but it comes at a cost.
Cache Optimization
Caches and Memory
Locality

- Temporal locality
  - If an item has been referenced recently, it will probably be accessed again soon

- Spatial locality
  - If an item has been referenced recently, nearby items will tend to be referenced soon

```
a = p.x*p.x + p.y*p.y;
b = (p.x - q.x);
c = (p.y - q.y);
```

```
for (i=0; i<N; i++)
a[i] = ...
```
Example: Matrix Multiplication

**Blocked (Tiled) Matrix Multiply**

Consider $A, B, C$ to be $N$-by-$N$ matrices of $b$-by-$b$ subblocks where $b = n / N$ is called the block size

for $i = 1$ to $N$
  for $j = 1$ to $N$
    {read block $C(i,j)$ into fast memory}
    for $k = 1$ to $N$
      {read block $A(i,k)$ into fast memory}
      {read block $B(k,j)$ into fast memory}
      $C(i,j) = C(i,j) + A(i,k) * B(k,j)$ {do a matrix multiply on blocks}
    {write block $C(i,j)$ back to slow memory}
How Unity exploit Caches

- Entity Component System
  - Data-Oriented Tech Stack
How Unity exploit Caches

- Traditional: Game object and components is “Has-a”
How Unity exploits cache:

- In the transforms (classic) system:
  - Memory divided by cache lines.
  - 640 bytes on 4 transforms, 128 bytes wasted, 1,920 bytes on unused cache lines.

- In the DataNeededToMove (Entity Component System) system:
  - Memory divided by cache lines.
  - 11,875 bytes on AIMovementData, 14 bytes wasted, 64 bytes on unused cache lines.

Legend:
- Green: Cache Line - 64 bytes
- Light Green: Wasted Memory & Access Time
- Brown: DataNeededToMove - 25 bytes
- Light Blue: Transform - ~160 bytes
How Unity exploit Caches
What can we do with these acceleration algorithms?
Conclusion

Acceleration algorithms are and will continue to be a core part of the rendering process, despite faster and faster GPUs. They enable us to render many times more complex scenes in real time, without having to change the underlying hardware or rendering pipeline. They’re also just some pretty cool algorithms that are fun to learn and implement - and they require knowledge of how rendering as well as the cache/memory hardware works to implement effectively.
Resources


Real-Time Rendering Book by Eric Haines, Natty Hoffman, Tomas Moller

[https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/Cache3.pdf](https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/Cache3.pdf)

Game Design Patterns by Robert Nystrom