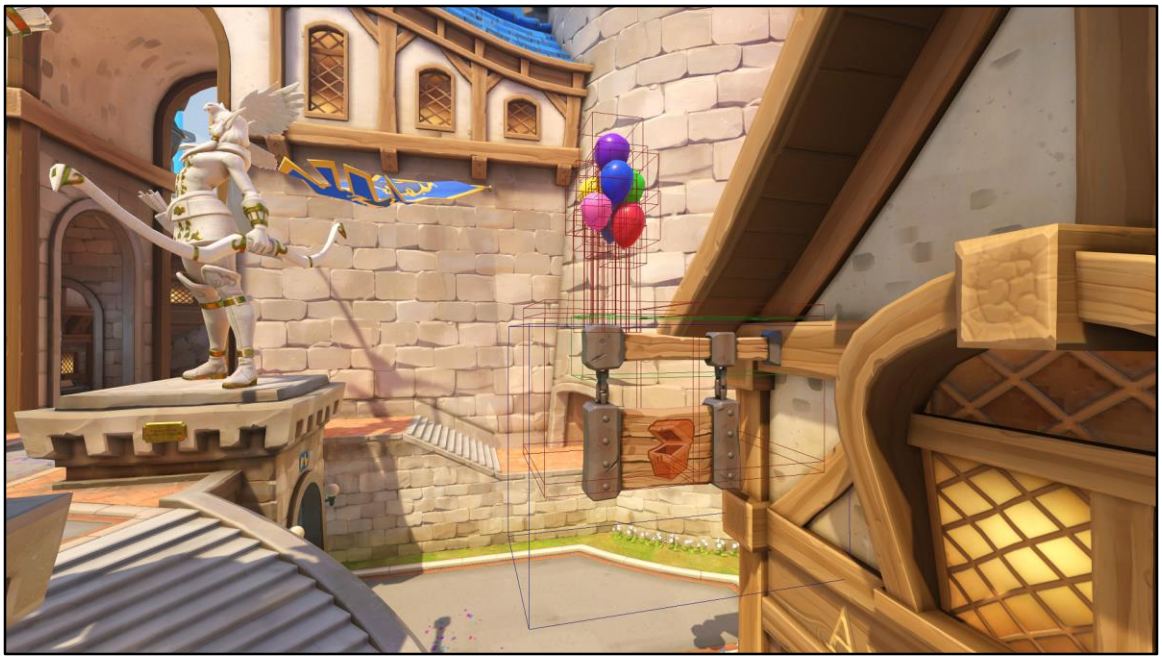
Dynamic Bounding Volume Hierarchies

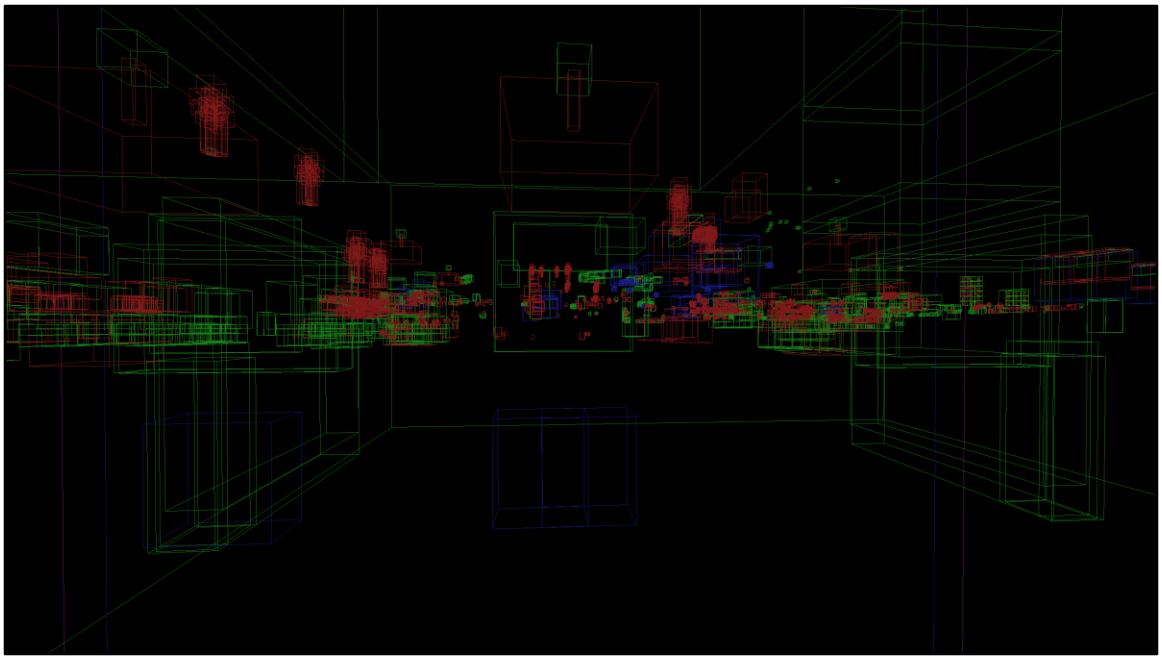
Erin Catto, Blizzard Entertainment

This is one of my favorite Overwatch maps: BlizzardWorld. This is the spawn area inside the Hearthstone Tavern.

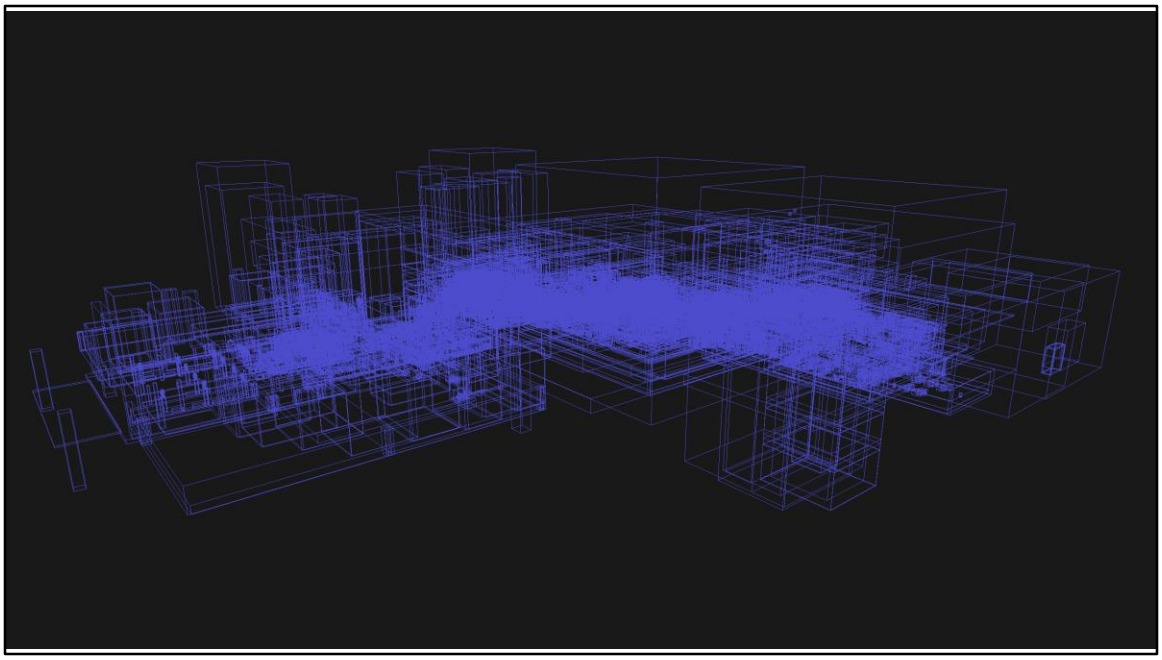
All the objects in the map, the floors, the walls, the chairs are objects that are enclosed in axis aligned bounding boxes. This is done to accelerate collision detection.



Even the balloons and their strings have separating bounding boxes.

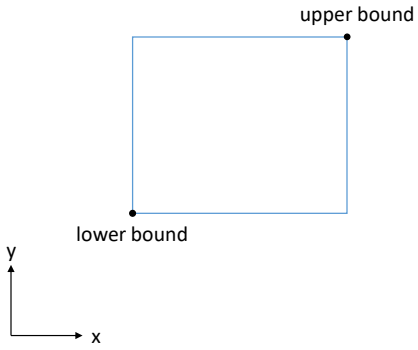


There is almost 9000 separate collision objects in the editor. Green boxes are static objects, blue kinematic, and red dynamic.



Here is a zoomed out view of all the bounding boxes in the map.

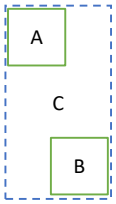
Axis Aligned Bounding Box (AABB)




```
struct AABB
{
    Vec3 lowerBound;
    Vec3 upperBound;
};
```

Here is the definition of an a bounding box that I will use.

Union of two AABBs



$C = A \cup B$

```
AABB Union(AABB A, AABB B)
{
    AABB C;
    C.lowerBound = Min(A.lowerBound, B.lowerBound);
    C.upperBound = Max(A.upperBound, B.upperBound);
    return C;
}
```

Given two bounding boxes we can compute the union with min and max operations. These can be made efficient using SIMD.

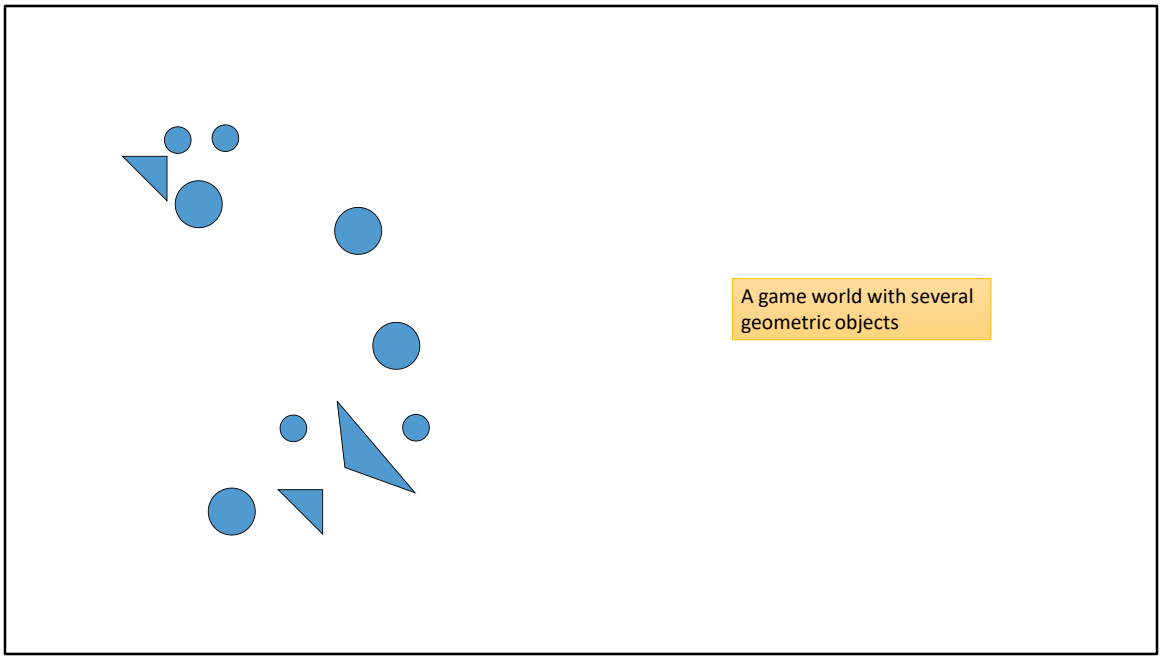
Notice the cup notation. You will see it again.

Surface area of an AABB

```
float Area(AABB A)
{
    Vec3 d = A.upperBound - A.lowerBound;
    return 2.0f * (d.x * d.y + d.y * d.z + d.z * d.x);
}
```

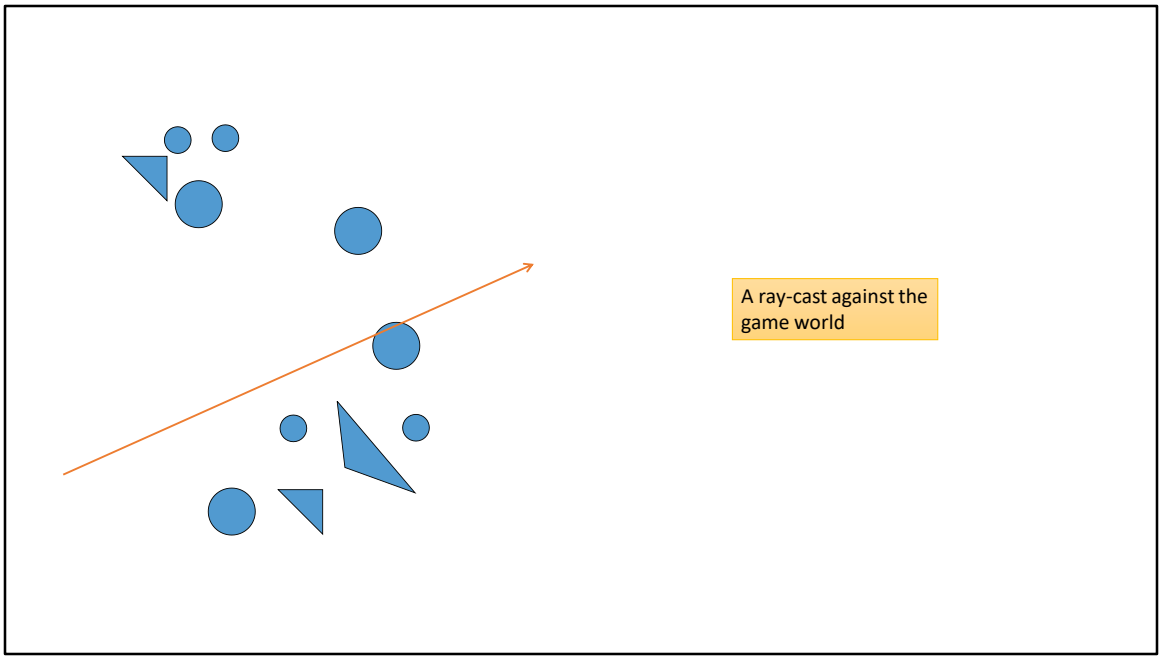
$SA(A)$

I will also need to compute the surface area. Notice the SA notation. I will use that as well.

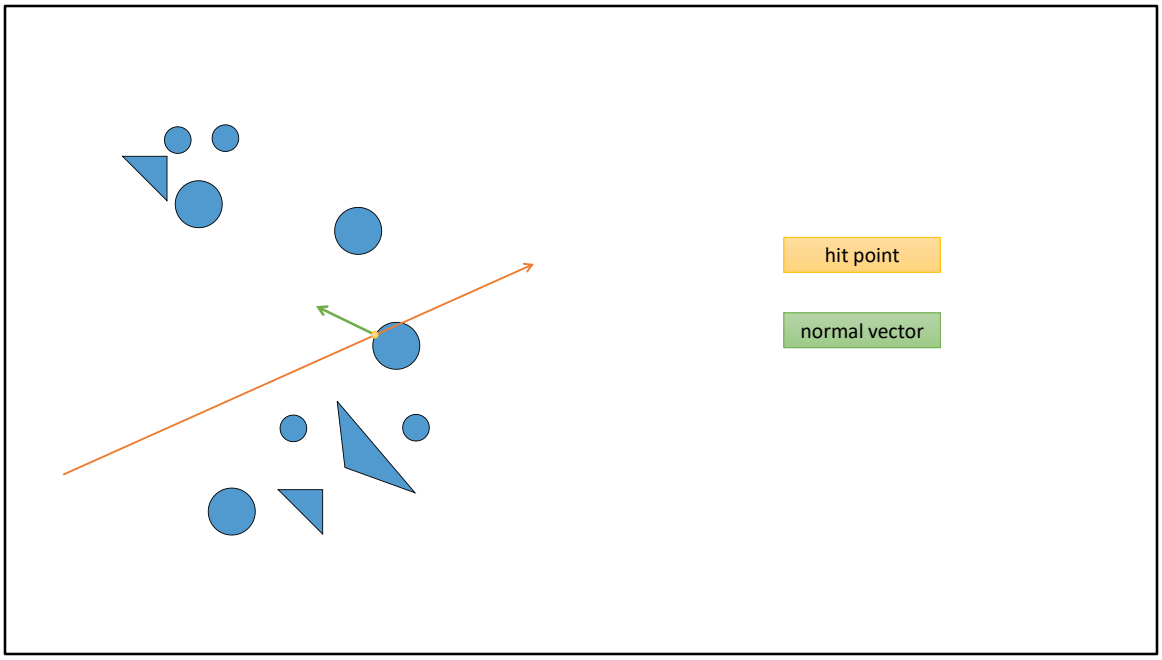


Game worlds often have many objects. Players, rigid bodies, wall, floors, etc.

This is an abstract example of a game world with several geometric objects.

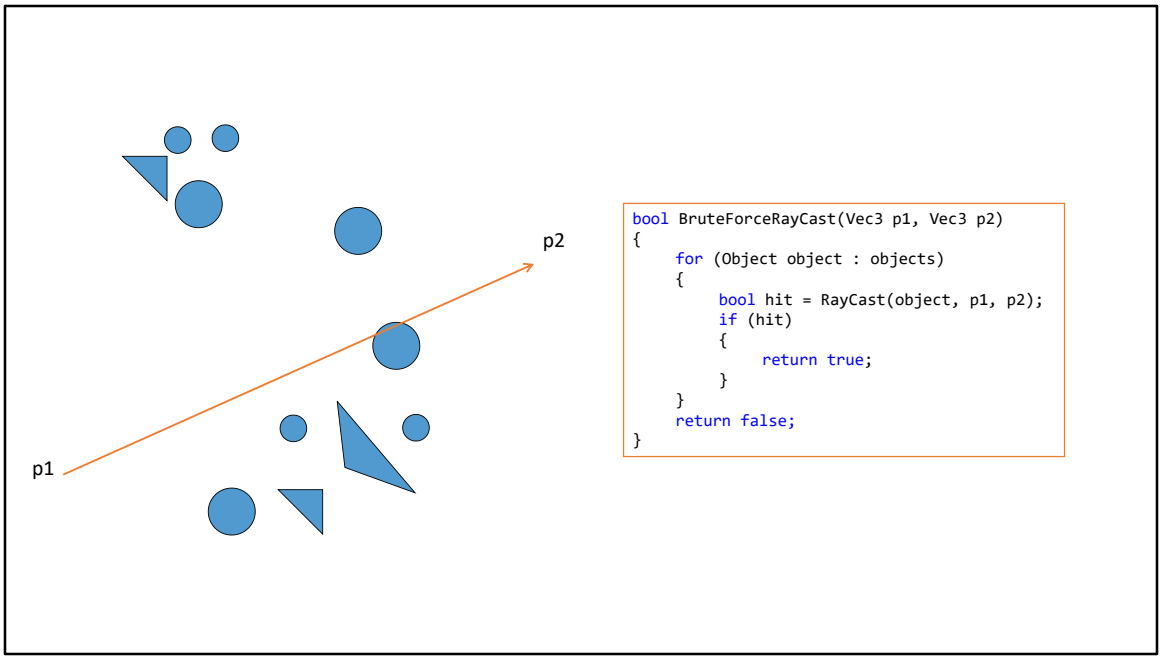


Often in games we need to ray cast against the scene. To shoot a weapon, check visibility, find the ground, etc.

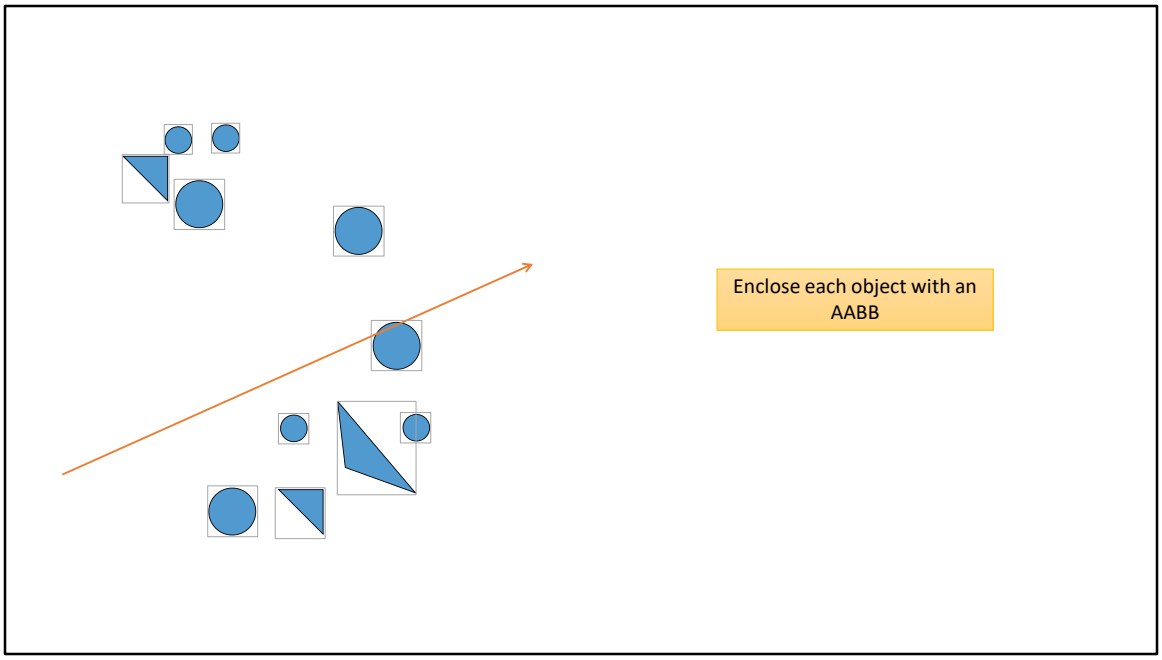


Typically we want the hit point, normal vector, and some way of identifying the object that was hit. Detailed ray cast results are not in the scope of this presentation.

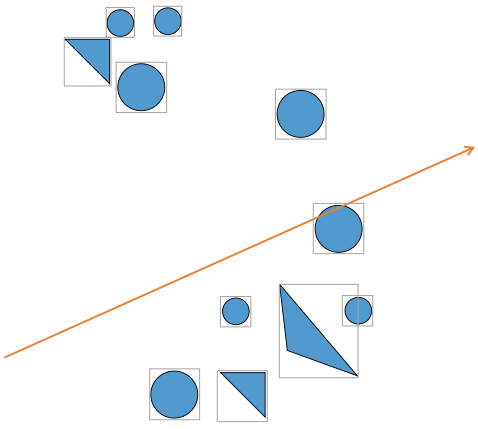
However, I am going to discuss ways to make ray casting faster.



The simplest way to perform the ray cast is to check each object. Brute force can be slow if there are many objects. Sometimes cache friendly is not enough.



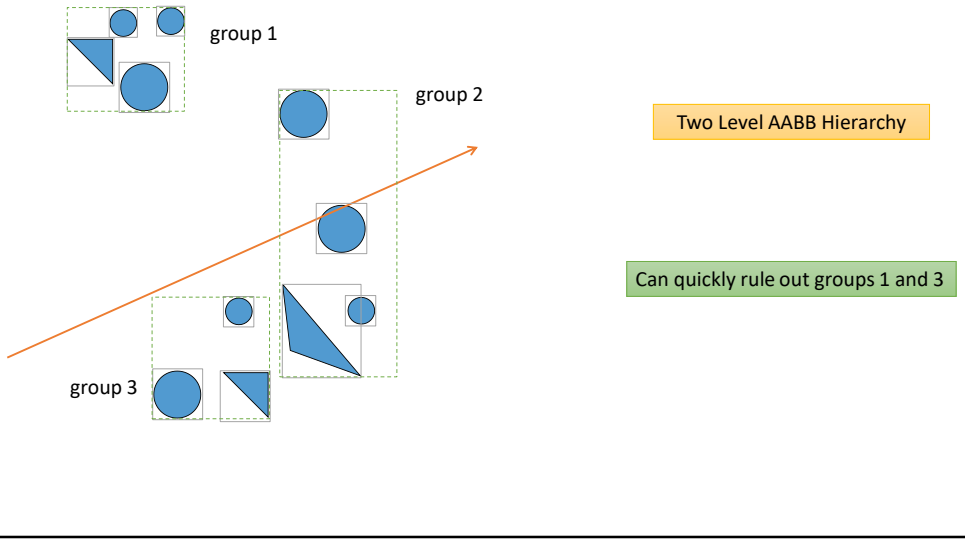
Suppose the shapes are complex. Then it might be worth checking whether the ray intersects the AABB before testing the more complex object. Also the AABB test can be faster because we don't need the hit point or normal. We just a need to know if the ray overlaps the AABB.



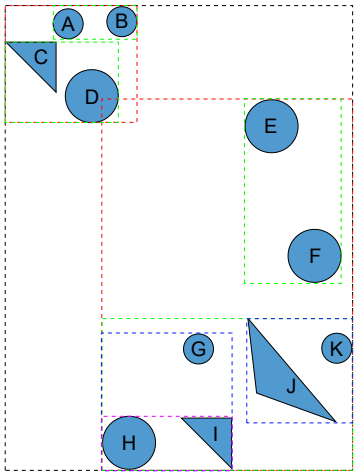
```
bool RayCastTestAABB(Vec3 p1, Vec3 p2)
{
      for (Object object : objects)
      {
            if (TestOverlap(object.aabb, p1, p2) == false)
            {
                  continue;
            }

            bool hit = RayCast(object, p1, p2);
            if (hit)
            {
                  return true;
            }
      }
      return false;
}
```

Here is the algorithm for going over every object, but first testing if the ray overlaps the bounding box. Again, I don't have time to get into the details of the detailed ray cast or the bounding box overlap test.



We can try grouping some objects together inside larger bounding boxes. Then we can skip whole groups in many cases.

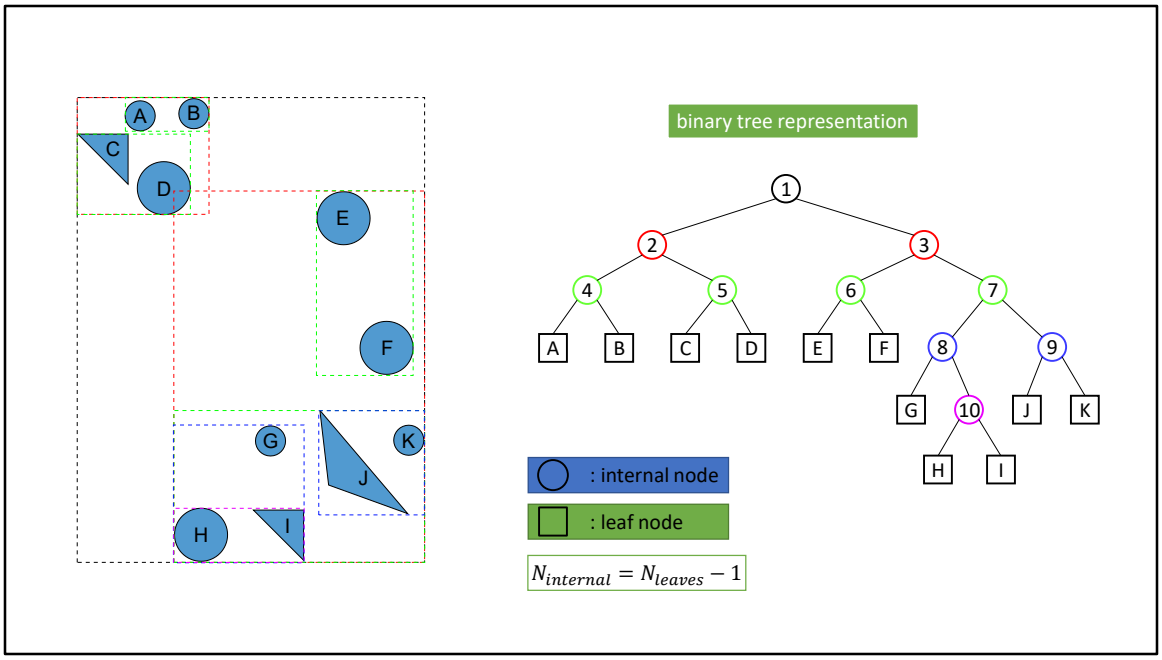


Bounding Volume Hierarchy (BVH)

The bounding volumes are AABBs

Other shapes are possible

Once you use more than one level of bounding boxes, you might as well make a whole tree. This is a bounding volume hierarchy. Here the bounding volumes are AABBs. Other shapes are possible, such as bounding spheres.



I'm using a binary tree for the BVH. This is efficient and easy to implement.

The tree consists of internal nodes and leaf nodes. The leaf nodes are the collision objects and the internal nodes only exist to accelerate collision queries.

Recall that the number of internal nodes can be computed from the number of leaf nodes, regardless of the tree structure.

```
struct Node
{
    AABB box;
    int objectIndex;
    int parentIndex;
    int child1;
    int child2;
    bool isLeaf;
};
```

```
struct Tree
{
    Node* nodes;
    int nodeCount;
    int rootIndex;
};
```

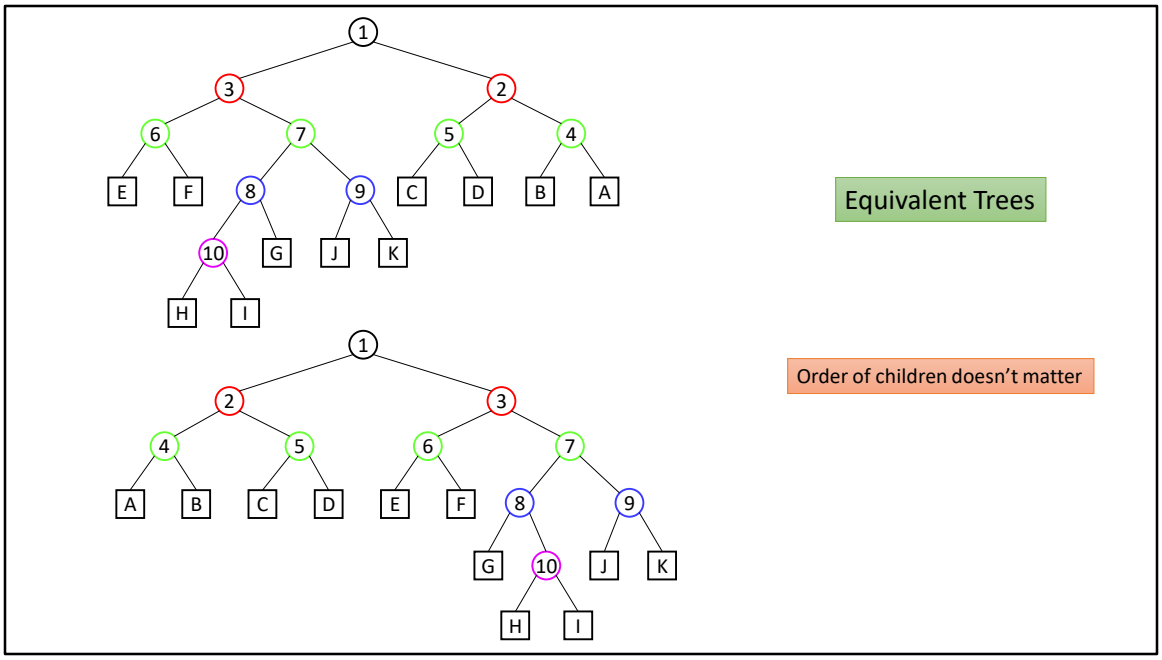
The code for the binary tree consists of a node structure and a tree structure. The node has an object index that relates it to the game object it contains.

```
bool TreeRayCast(Tree tree, Vec3 p1, Vec3 p2)
{
    Stack<int> stack;
    Push(stack, tree.rootIndex);
    while (IsEmpty(stack) == false)
    {
        int index = Pop(stack);
        if (TestOverlap(tree.nodes[index].box, p1, p2) == false)
        {
            continue;
        }

        if (nodes[index].isLeaf)
        {
            int objectIndex = tree.nodes[index].objectIndex;
            if (RayCast(objects[objectIndex], p1, p2))
            {
                return true;
            }
        }
        else
        {
            Push(stack, tree.nodes[index].child1);
            Push(stack, tree.nodes[index].child2);
        }
    }
    return false;
}
```

Here's an example of ray casting against a binary tree BVH. This code uses a local stack instead of using functional recursion. Children get pushed onto the stack until no more candidates are left.

This is just an example. It leaves out several important optimizations.



The binary tree can take a number of forms, depending on the how the BVH is built. For a BVH the ordering of nodes is not relevant. We can swap left and right children at any level and still have the same BVH.

For example, these two trees are equivalent. This is different than other binary trees, such as Red Black trees, where the order matters.

Dynamic AABB Tree

- Moving objects
- Object creation and destruction
- Streaming

Insertion Algorithm

Key algorithm for dynamic bounding volume hierarchies

The key algorithm for dynamic bounding volume hierarchies is the algorithm for inserting leaves. So I'm going to spend a lot of time on this. Leaf removal is straight forward and is not covered.

```
void InsertLeaf(Tree tree, int objectIndex, AABB box)
{
    int leafIndex = AllocateLeafNode(tree, objectIndex, box);
    if (tree.nodeCount == 0)
    {
        tree.rootIndex = leafIndex;
        return;
    }

    // Stage 1: find the best sibling for the new leaf

    // Stage 2: create a new parent

    // Stage 3: walk back up the tree refitting AABBs
}
```

Here is the structure of the insertion algorithm.

```
// Stage 1: find the best sibling for the new leaf

int bestSibling = 0;
for (int i = 0; i < m_nodeCount; ++i)
{
    bestSibling = PickBest(bestSibling, i);
}
```

Stage 1 descends the tree, looking for the best option for a sibling. I'll be talking a lot about how to find the best sibling.

```
// Stage 2: create a new parent
int oldParent = tree.nodes[sibling].parentIndex;
int newParent = AllocateInternalNode(tree);
tree.nodes[newParent].parentIndex = oldParent;
tree.nodes[newParent].box = Union(box, tree.nodes[sibling].box);

if (oldParent != nullIndex)
{
      // The sibling was not the root
      if (tree.nodes[oldParent].child1 == sibling)
      {
            tree.nodes[oldParent].child1 = newParent;
      }
      else
      {
            tree.nodes[oldParent].child2 = newParent;
      }

      tree.nodes[newParent].child1 = sibling;
      tree.nodes[newParent].child2 = leafIndex;
      tree.nodes[sibling].parentIndex = newParent;
      tree.nodes[leafIndex].parentIndex = newParent;
}
else
{
      // The sibling was the root
      tree.nodes[newParent].child1 = sibling;
      tree.nodes[newParent].child2 = leafIndex;
      tree.nodes[sibling].parentIndex = newParent;
      tree.nodes[leafIndex].parentIndex = newParent;
      tree.rootIndex = newParent;
}
```

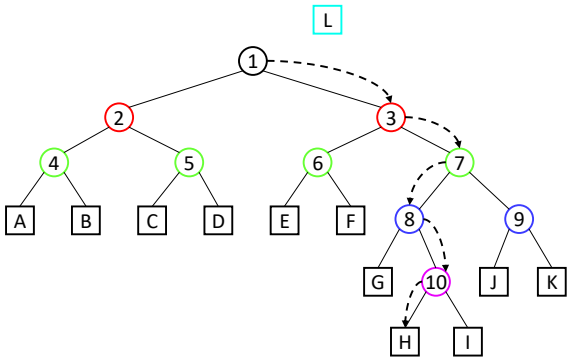
Stage 2 deals with all the details of modify the tree after a sibling has been chosen. Edge cases must be handled.

```
// Stage 3: walk back up the tree refitting AABBs
int index = tree.nodes[leafIndex].parentIndex;
while (index != nullIndex)
{
     int child1 = tree.nodes[index].child1;
     int child2 = tree.nodes[index].child2;

     tree.nodes[index].box = Union(tree.nodes[child1].box, tree.nodes[child2].box);
     index = tree.nodes[index].parentIndex;
}
```

Stage 3 adjusts the AABBs of the new leaf's ancestors. This is called *refitting*.

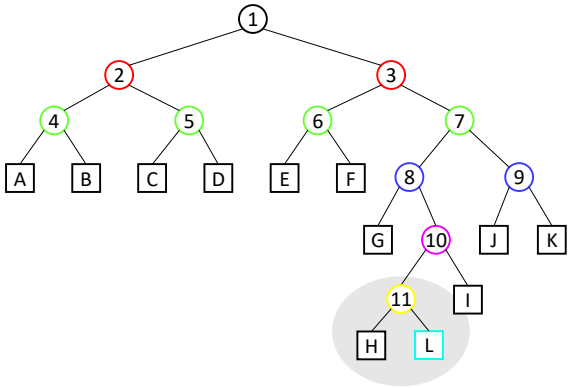
Stage 1: Look for best sibling



H found to be the best sibling for new leaf L

Here is what a sibling search might look like in Stage 1.

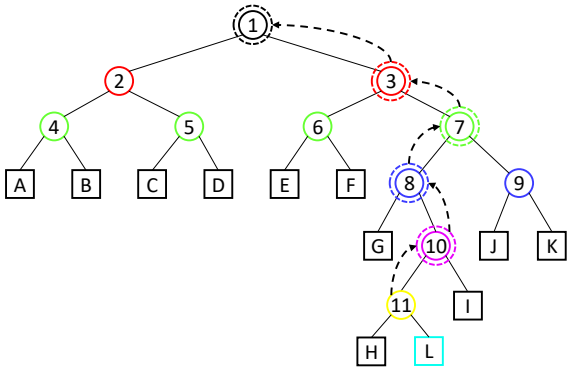
Stage 2: Create new internal node



Create internal node 11, parent for H and L

Stage 2 handles the creation of a new node and hooking the nodes together.

Stage 3: Refit ancestor AABBs



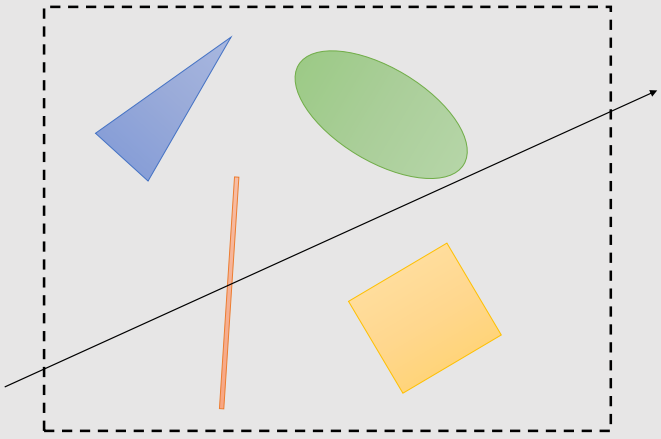
Ensure all nodes enclose their children

Stage 3 walks back up the tree and refits the parent bounding boxes.

Picking a good sibling

Goal: faster ray casts

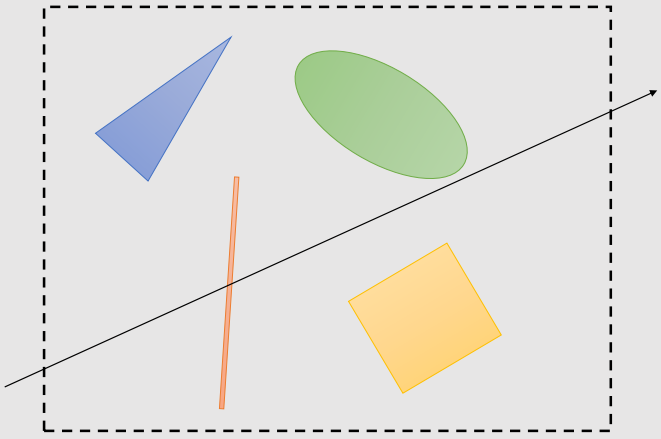
Surface Area Heuristic (SAH)



The probability of a ray hitting a convex object is proportional to the surface area

The surface area heuristic is a powerful metric that can be used to drive the construction of a BVH. The idea is that the probability of a ray hitting an object is proportional to the surface are of the object.

Surface Area Heuristic (SAH)



The probability of a ray hitting a convex object is proportional to the surface area

We can use this to build good BVHs

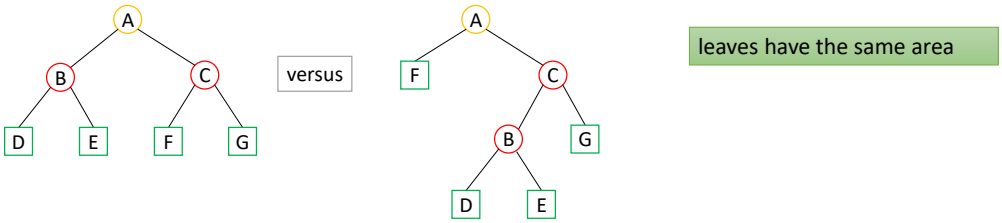
Cost function of a tree

$$C(T) = \sum_{i \in Nodes} SA(i)$$

```
float ComputeCost(Tree tree)
{
    float cost = 0.0f;
    for (int i = 0; i < tree.nodeCount; ++i)
    {
        cost += Area(tree.nodes[i].box);
    }
    return cost;
}
```

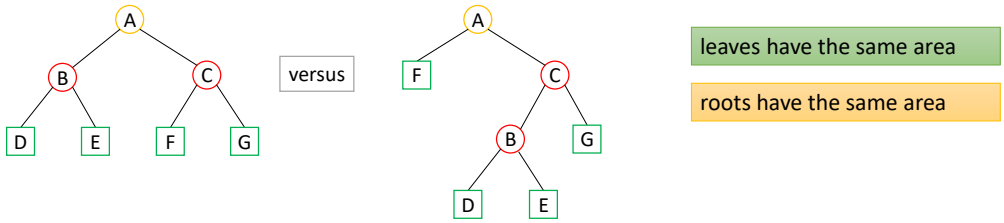
Using the surface area function we can compute a cost metric for any tree.

How shall we compare trees?



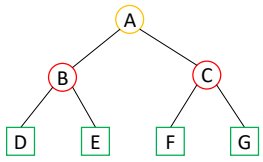
We want to a way to compare trees using the surface area heuristic. That way we can see which one is better.

How shall we compare trees?

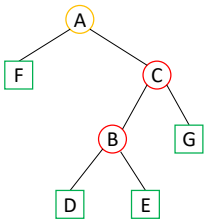


Many trees can be built from the same set of leaves. The surface area of the leaves is the same and the surface area of the root node is the same. Only the surface area of the internal nodes varies.

How shall we compare trees?



versus



leaves have the same area

roots have the same area

C has a different area!

$SA(C) = SA(F \cup G)$

$SA(C) = SA(B \cup G)$

Many trees can be built from the same set of leaves. The surface area of the leaves is the same and the surface area of the root node is the same. Only the surface area of the internal nodes varies.

Revised cost function

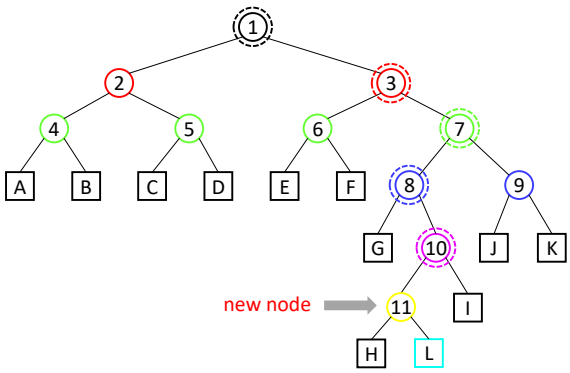
$$C(T) = \sum_{i \in Inner\ Nodes} SA(i)$$

```
float ComputeCost(Tree tree)
{
    float cost = 0.0f;
    for (int i = 0; i < tree.nodeCount; ++i)
    {
        if (tree.nodes[i].isLeaf == false)
        {
            cost += Area(tree.nodes[i].box);
        }
    }
    return cost;
}
```

Leaf area doesn't matter

The cost of a tree is the total surface area of the internal nodes. This gives us an objective way to compare the quality of two trees.

SAH insertion cost



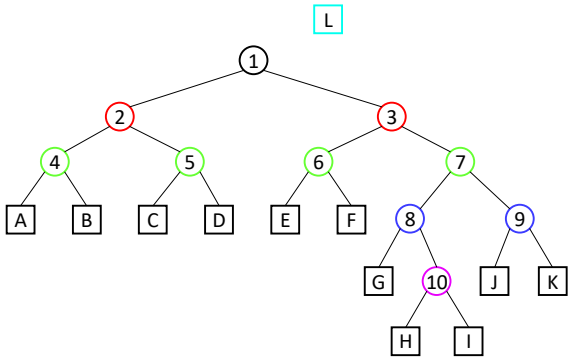
The cost of choosing H as the sibling for L

$$C_H = SA(11) + \Delta SA(10) + \Delta SA(8) + \Delta SA(7) + \Delta SA(3) + \Delta SA(1)$$

$$\Delta SA(node) = SA(node \cup L) - SA(node)$$

Going back to the earlier example where I inserted L into the tree and created node 11.
SAH gives us a way to compute the cost of inserting L. The cost is the area of the new parent node 11 plus the increased surface area of all the ancestors.
This is the surface area added to the tree.

Global search for optimum



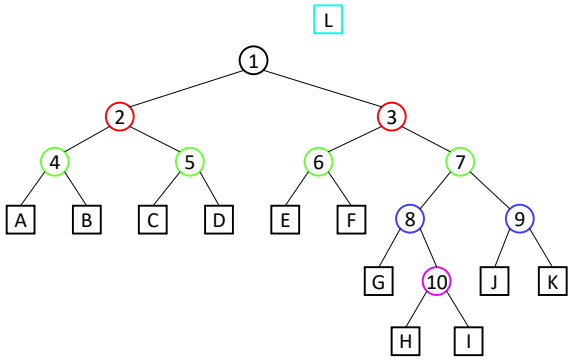
Sibling choices:
- internal nodes 1-10
- leaf nodes A-K

2N-1 choices

Every node in the tree is a potential sibling for leaf node L. Each choice adds a different surface area to the tree.

I would like to find the sibling that adds the least surface area to tree.

Global search for optimum



Sibling choices:
- internal nodes 1-10
- leaf nodes A-K

2N-1 choices

Expensive!

Unfortunately it is expensive to evaluate the cost of every potential sibling.

Branch and Bound Algorithm

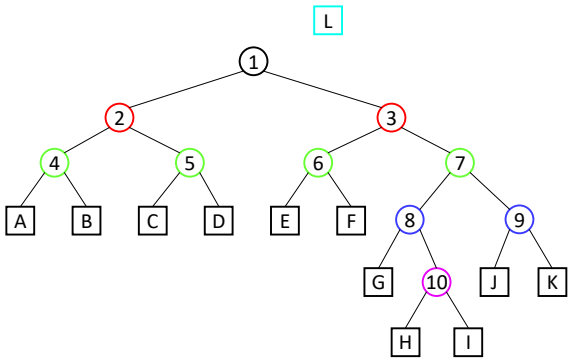
A faster global search

Branch and bound is a powerful algorithm that makes the global search faster.

Main idea of branch and bound

- Search through tree recursively
- Skip sub-trees that cannot possibly be better

Branch and Bound



Recurse through tree, looking for lowest cost sibling S for L

Use a priority queue Q to explore best candidates first

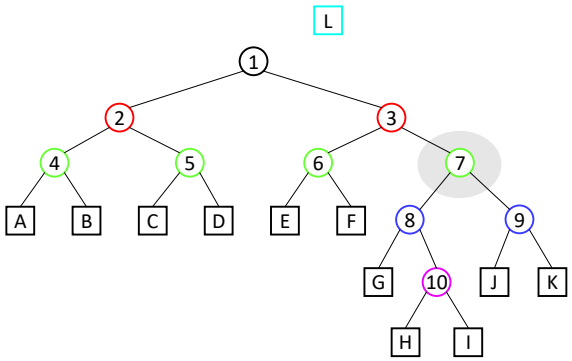
Initialize: $S_{best} = 1$

$C_{best} = SA(1 \cup L)$

$Q = \{1\}$

Here is a example of how branch and bound works.

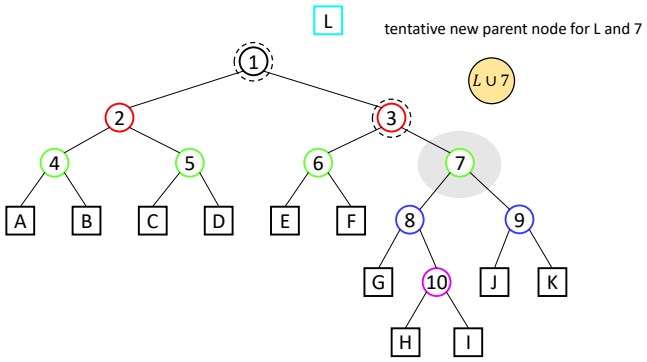
Branch and Bound



Suppose the search reaches node 7

Suppose we are exploring this tree, looking for the best sibling. We find our way to node 7 and want to determine if node 7 has the best cost. We also want to determine if it is worthwhile to explore the children of node 7.

Branch and Bound



Cost for choosing node 7

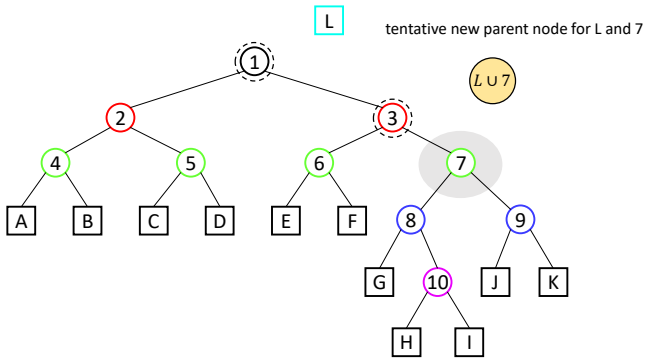
$$C_7 = SA(L \cup 7) + \Delta SA(3) + \Delta SA(1)$$

direct cost inherited cost

The cost of node 7 is the sum of the direct cost and the inherited cost.

The direct cost is the surface area of the new internal node that will be created for the siblings.
The inherited cost is the increased surface area caused by refitting the ancestor's boxes.

Branch and Bound



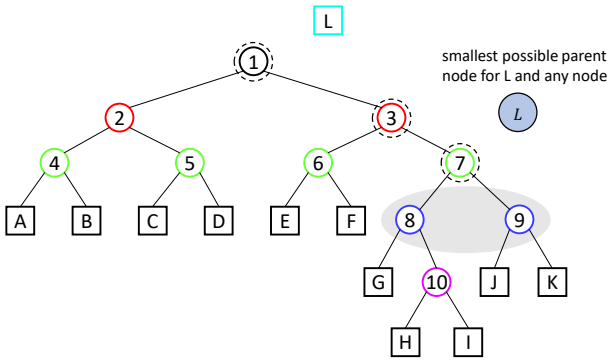
Cost for choosing node 7

$$C_7 = SA(L \cup 7) + \Delta SA(3) + \Delta SA(1)$$

$$if\ C_7 < C_{best}\ then\ C_{best} = C_7$$

If the cost of node 7 is better than the best cost then update the best cost.

Branch and Bound



Consider pushing 8 and 9 onto the queue

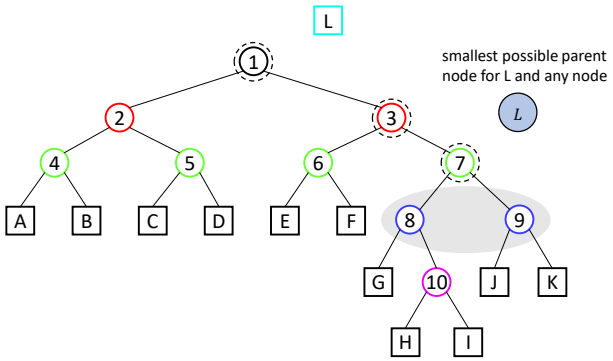
$$C_{low} = SA(L) + \Delta SA(7) + \Delta SA(3) + \Delta SA(1)$$

lower bound cost for nodes 8 and 9

Is it worthwhile to explore the sub-tree of node 7?

A lower bound for children of node 7 is the surface area of L plus the inherited cost (including 7).

Branch and Bound

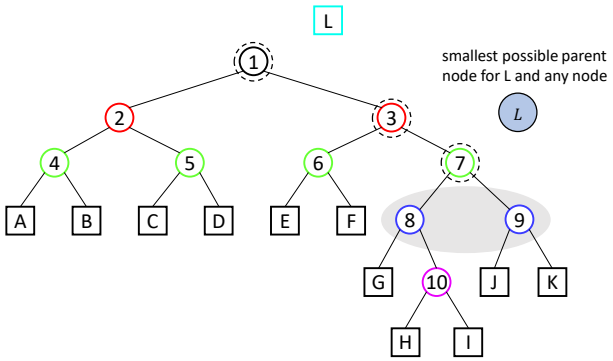


$$C_{low} = SA(L) + \Delta SA(7) + \Delta SA(3) + \Delta SA(1)$$

$$if\ C_{low} < C_{best}\ then\ push(8)\ and\ push(9)$$

If the lower bound cost for the children is lower than the best cost, then it is worth exploring those sub-trees and they are pushed onto the priority queue.

Branch and Bound



$$C_{low} = SA(L) + \Delta SA(7) + \Delta SA(3) + \Delta SA(1)$$

$if\ C_{low} < C_{best}\ then\ push(8)\ and\ push(9)$

Otherwise we can prune the sub-tree at node 7 from the search

Otherwise can prune the whole sub-tree rooted at node 7 from the search. This drastically improves performance.

Object Movement

Object movement strategies

- Refit ancestors

Object movement strategies

- Refit ancestors
  - leads to low quality trees

Object movement strategies

- Refit ancestors
  - leads to low quality trees
- Rebuild subtrees

Object movement strategies

- Refit ancestors
  - leads to low quality trees
- Rebuild subtrees
  - expensive (similar to garbage collection)

Object movement strategies

- Refit ancestors
  - leads to low quality trees
- Rebuild subtrees
  - expensive (similar to garbage collection)
- Remove/re-insert

Object movement strategies

- Refit ancestors
  - leads to low quality trees
- Rebuild subtrees
  - expensive (similar to garbage collection)
- Remove/re-insert
  - also expensive, but spread out

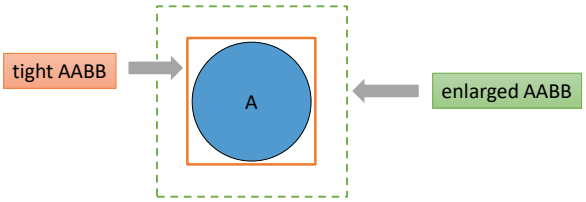
Object movement strategies

- Refit ancestors
  - leads to low quality trees
- Rebuild subtrees
  - expensive (similar to garbage collection)
- Remove/re-insert
  - also expensive, but spread out

choose remove/re-insert

Enlarged AABBs

- At 60Hz objects often don't move far per frame
- So use an enlarged AABB in the BVH
- Only update a leaf if the tight fitting AABB moves outside of the enlarged AABB



There are many schemes for enlarging the AABB. Choose one that works well for your game.

Problem: sorted input

A problem remains. Sorted input can wreck the dynamic tree.

Sorted input

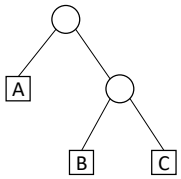
Imagine we have several game objects in a row. They are inserted into the tree in sorted order. We cannot disallow this because this is what the game may need to do.

Sorted input

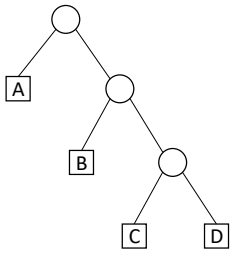
Here's how that process looks.

Sorted input

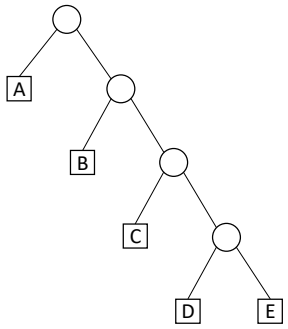
Sorted input



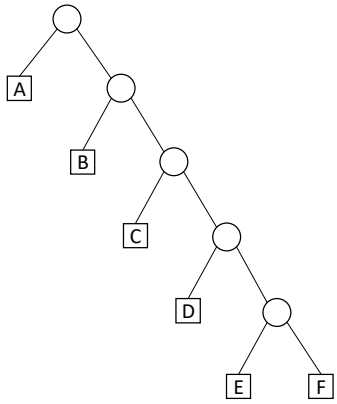
Sorted input



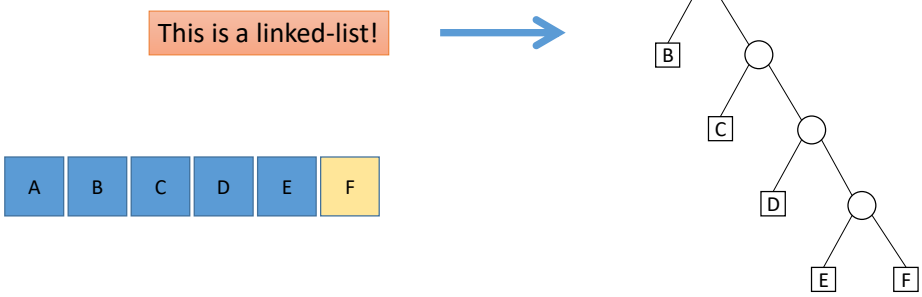
Sorted input



Sorted input



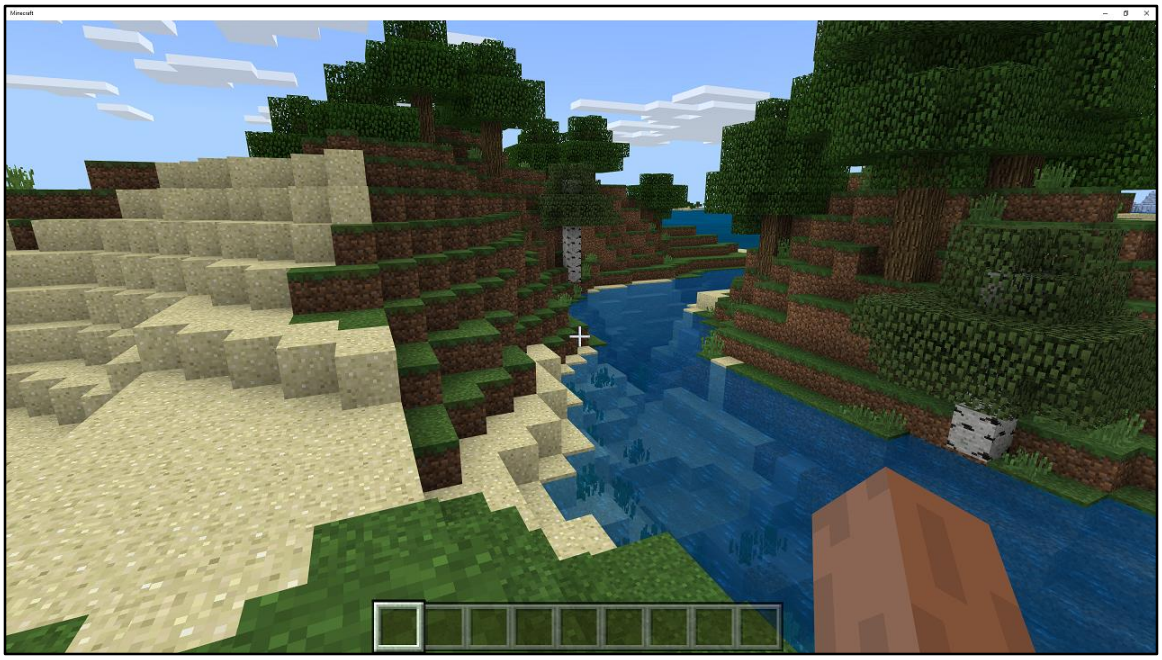
Sorted input



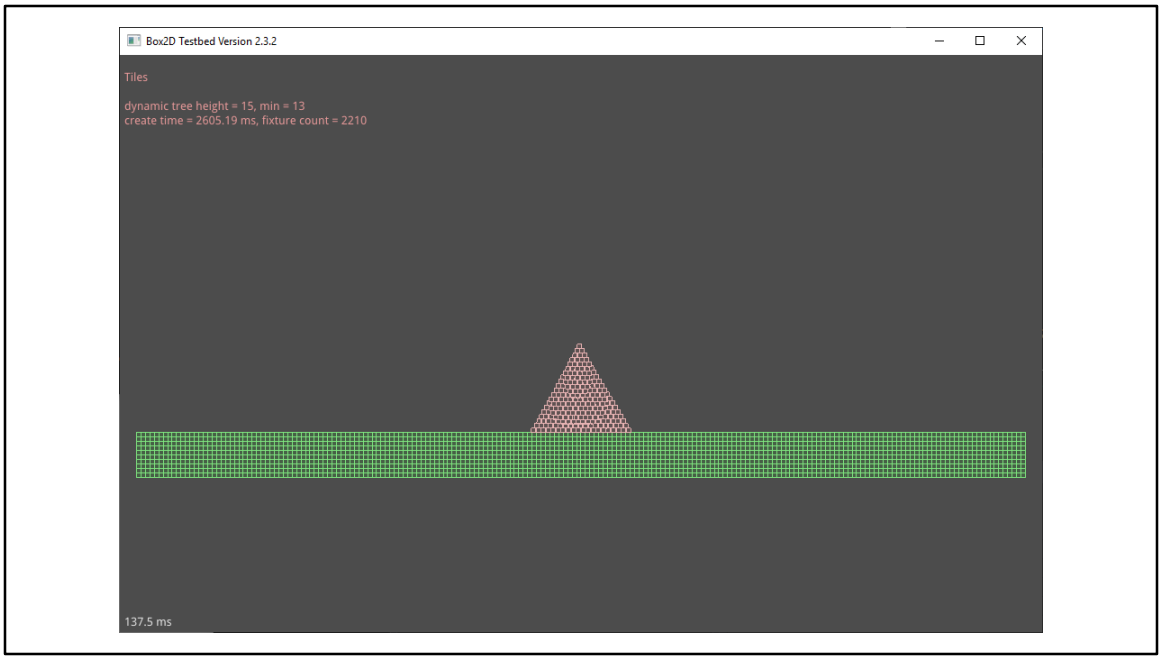
In this case the incremental SAH fails to provide a good tree. This is nothing new, this was known by the guys who invented the SAH back in 1987 (Goldsmith and Salmon).

To be honest, it is hard to imagine a reasonable cost metric that wouldn't fail.

Is sorted input an edge case?


Minecraft

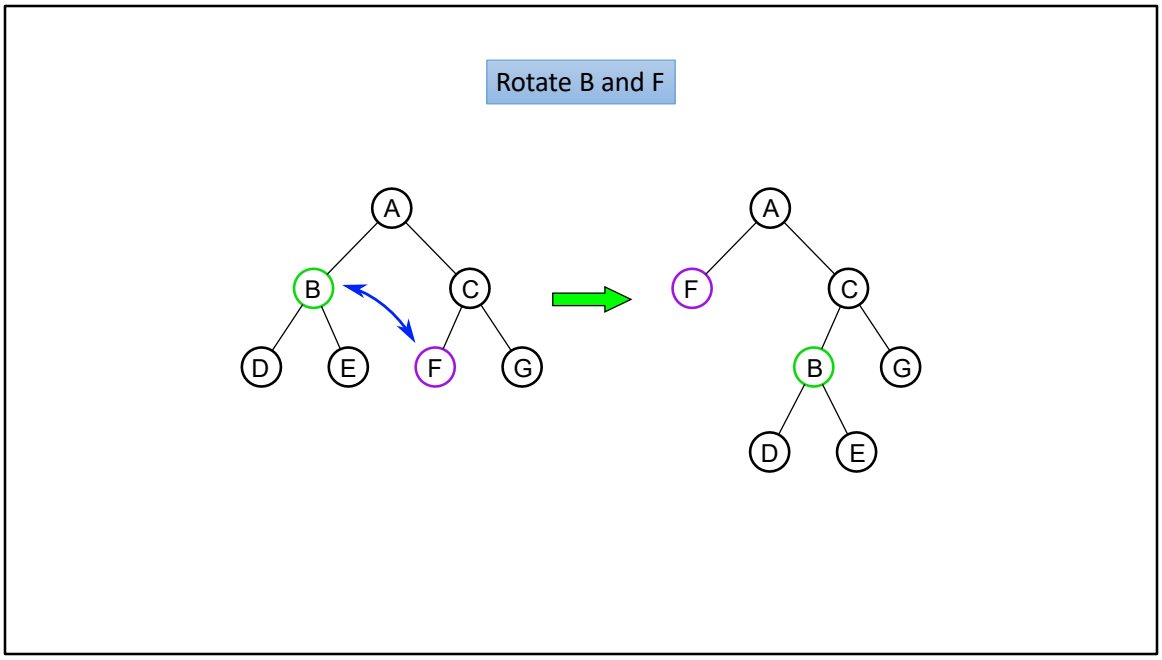

Fallen Star
Life: 100/100
Mana


Box2D Testbed Version 2.3.2
Tiles
dynamic tree height = 15, min = 13
create time = 2605.19 ms, fixture count = 2210
137.5 ms

Tree rotations

Re-arranging a tree to reduce the SAH cost

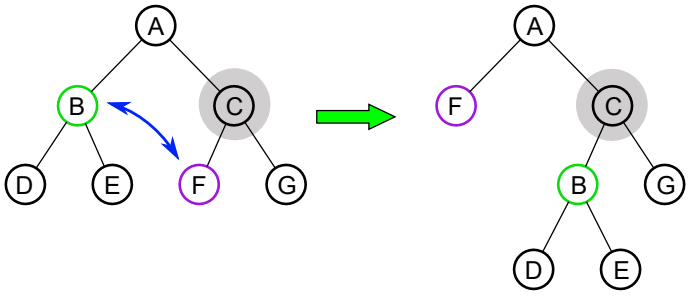
Tree rotations are used for AVL trees to keep them balanced. They can also be used for bounding volume hierarchies to reduce the surface area and to mitigate the problems introduced by sorted input.



In the tree on the left, A has four grand children. We can swap B and F to reconfigure the tree.

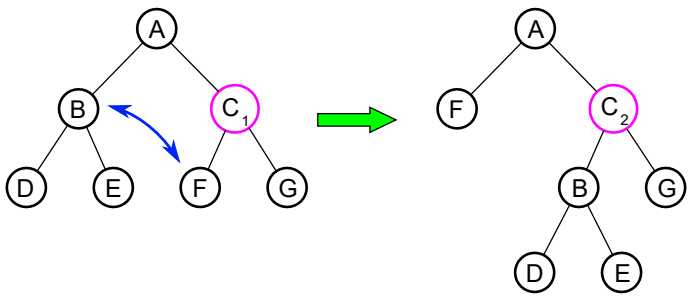
This is a local operation. Node A may be the child of some other node. Also, D, E, F, and G may have children.
This tree rotation does not affect the ancestors of A or the descendants of D, E, F, and G.

Rotate B and F



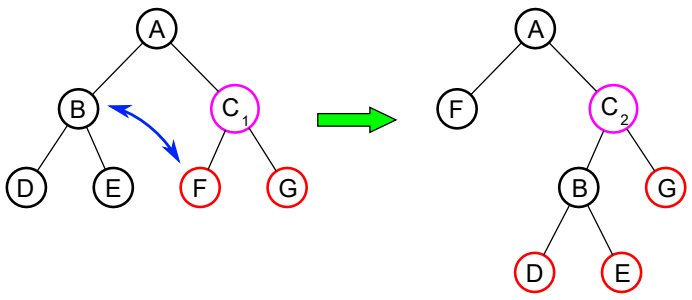
Only the surface areas of C differ

Rotate B and F



Which is better?

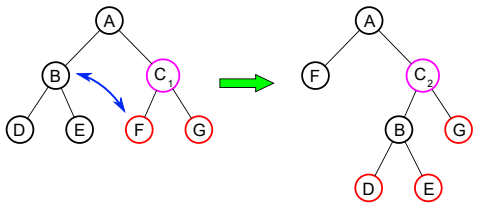
Rotate B and F

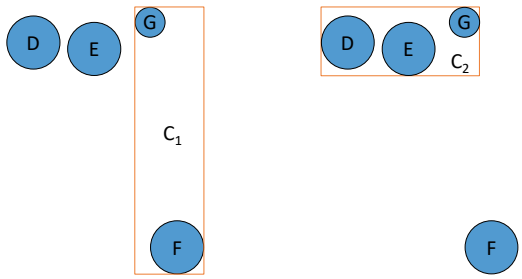


$SA(C_1) = SA(F \cup G)$

$SA(C_2) = SA(D \cup E \cup G)$

Rotate B and F

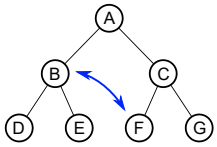


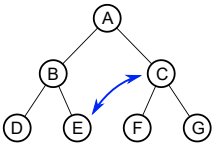


$SA(C_2) < SA(C_1)$

Four possible rotations

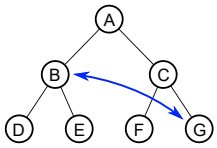
$B \Leftrightarrow F$

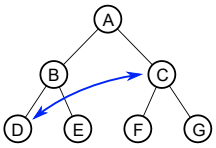




$C \Leftrightarrow E$

$B \Leftrightarrow G$





$C \Leftrightarrow D$

Sorted input

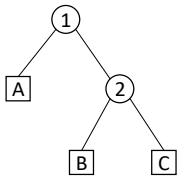
Now I will show how to use the tree rotation to resolve the sorted input problem.

Sorted input

A

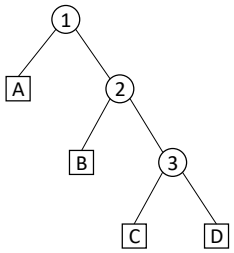
Sorted input

Sorted input

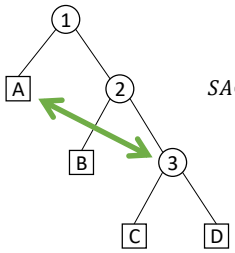


Can rotate A and C or A and B
Does not reduce the area of node 2

Sorted input



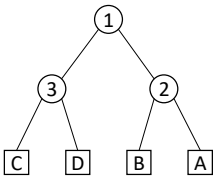
Sorted input



$SA(2) = SA(B \cup C \cup D)$

Rotate 3 and A

Sorted input



$SA(2) = SA(B \cup A)$

Rotate 3 and A

```
// Stage 3a: walk back up the tree refitting AABBs and applying rotations
int index = tree.nodes[leafIndex].parentIndex;
while (index != nullIndex)
{
     int child1 = tree.nodes[index].child1;
     int child2 = tree.nodes[index].child2;

     tree.nodes[index].box = Union(tree.nodes[child1].box, tree.nodes[child2].box);

     Rotate(index);

     index = tree.nodes[index].parentIndex;
}
```

The rotation operation can be installed in stage 3. It is a local operation that optimizes the tree as the ancestor AABBs are refitted.

References

- J. Goldsmith (1987) - Automatic Creation of Object Hierarchies
- S. Omohundro (1989) - Five Balltree Construction Algorithms
- A. Kensler (2008) - Tree Rotations for Improving Bounding Volume Hierarchies
- J. Bittner (2015) - Incremental BVH Construction for Ray Tracing
- N. Presson, btDbvt, Bullet Physics Engine, 2008

Thanks!

- box2d.org
- github.com/erincatto
- @erin_catto