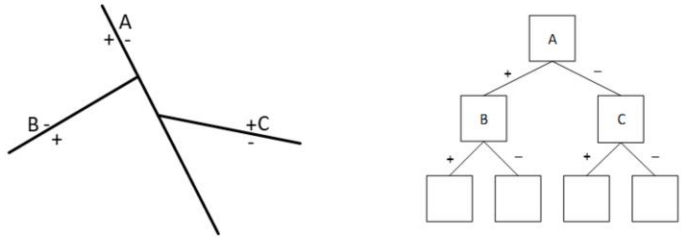
BSP Trees

jodavis42@gmail.com

BSP Tree

Recursively subdivides space with a hyperplane



Originally used for view-dependent rendering

Bsp (binary space partitioning) trees are a very general purpose spatial partition. Bsp trees can be used to represent oct/quad trees and k-d trees. A bsp-tree splits space in half using a splitting hyper-plane (a plane in 3d). Despite their simplicity, bsp trees are very versatile. Originally they were used to solve the back-to-front ordering issue of rendering by allowing the traversal of the tree from any view-point. This is now almost entirely antiquated because of graphics hardware (hierarchical z-buffering). Nevertheless, bsp-trees still have a lot of uses.

Bsp trees are unlike all other spatial partitions talked about so far as build times are expensive, hence they are better suited for static geometry. Keep in mind I will not be talking about them with the same interface in mind. In particular, we'll not have Insert/Update/Remove only a construction. We'll also mainly focus on ray-casting (although I'll mention other forms of casting briefly) and we'll not talk about pair tests.

A bsp-tree, much like a static aabb tree, is built from a set of input geometry. Most commonly bsp trees work with triangles as input (although polygons can also work), not bounding volumes. For this reason they tend to be used on a mesh directly.

Bsp Construction

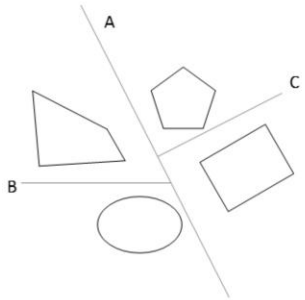
1. Pick a split plane
2. Partition geometry
3. Recurse until stop condition

Bsp-tree construction is typically broken up into 3 main steps. Basically we just keep splitting the geometry based upon some split plane until a termination condition is reached. The termination conditions can vary greatly but for our class we'll be stopping when a node only has 1 triangle.

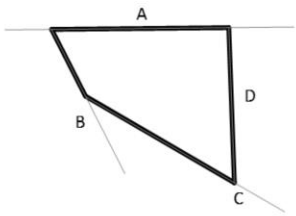
Split Plane

Two main methods of picking split planes

Arbitrary split planes



Auto-Partitioning



Picking a split plane can be broken up into two main categories: Auto-partitioning and arbitrary split planes. Auto-partitioning picks split planes from the input geometry. Arbitrary partitioning just chooses some other split plane based upon some heuristics. Auto-partitioning is often the primary method used to choose a split plane as it greatly limits the number of candidates, even though there are often much better choices available.

Partitioning geometry

Have to deal with 4 cases:

1. Positive side
2. Negative side
3. Straddling
4. Co-Planar

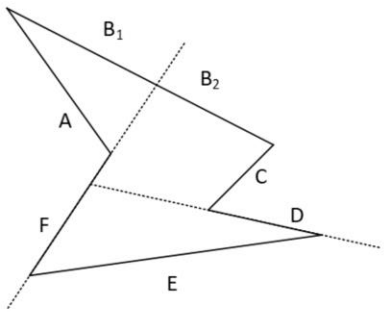
Where do we store co-planar geometry?

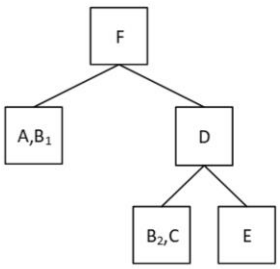
There are 4 cases to consider when partitioning input geometry. The first two cases are self-explanatory, if a polygon is completely on one side of the split plane then we pass it down that side. If a polygon is straddling, typically the polygon is split into two new polygons to send down both sides.

The tricky case is what to do about coplanar polygons. By definition, we will have at least one coplanar polygon when using auto-partitioning and it is likely that we'll get more than one. What we choose to do with any coplanar polygons determines what kind of a tree we make. There's 3 distinct classifications for Bsp trees: node-storing, leaf-storing, and solid-leaf.

Node Storing Tree

Co-planar objects stored on node





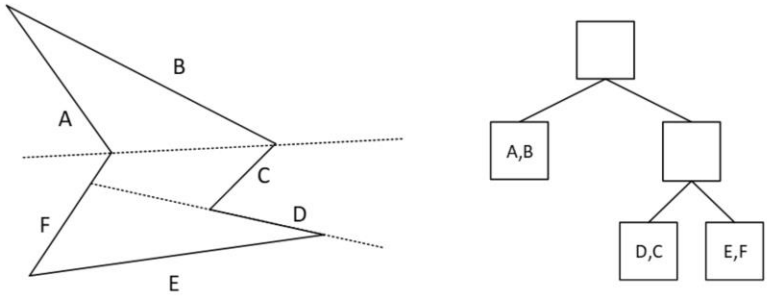
A node-storing bsp-tree is able to store geometry in the internal nodes as well as leaf nodes. In particular, when we have co-planar geometry we store them in the node itself. This is the tree we'll be constructing for our assignment.

This kind of tree isn't without its problems though, in particular it's not as well suited for collision tests. As we traverse down the tree to perform collision detection we have to check all polygons stored within a node. This can waste a lot of time if we determine further down the tree that we're outside.

Leaf Storing Tree

Push co-planar nodes down a side

don't allow again as a split plane



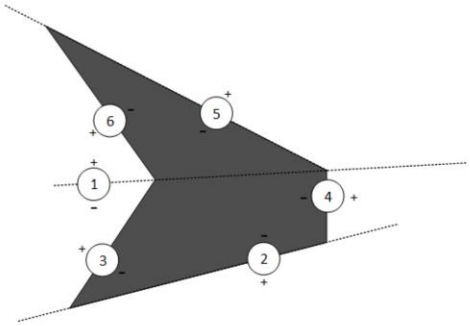
A leaf storing tree will not store any geometry in an internal node. When a coplanar node is encountered then the geometry is passed down one side and marked so it can't be selected again for a split plane. Which side it's passed down isn't particularly important, but one approach is to send it down the positive or negative side based upon the sign of the normal compared to the split plane's normal.

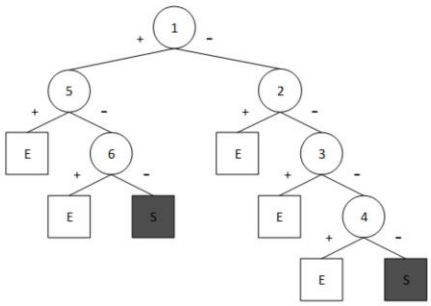
This tree works out a bit better for collision detection purposes as we can early out with less work since there's no geometry stored in an internal node.

Solid Leaf Tree

Doesn't store geometry

Represents boundary of a shape



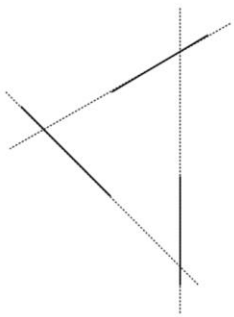


The final kind of bsp tree doesn't even store the geometry itself, but just marks leaf nodes as either empty or solid (E or S in the picture). With this kind of tree we can tell if a point ends up inside or outside a mesh. This kind of tree only makes sense when we use the bsp tree on water-tight meshes.

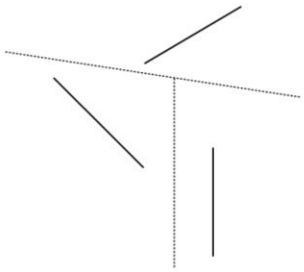
Auto vs. Arbitrary

Some times auto-partitioning can't choose well

1. Forcing splits that aren't necessary



auto-partitioning



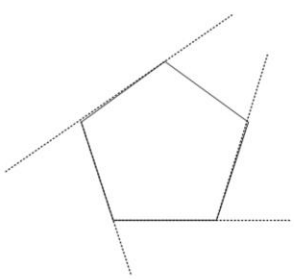
arbitrary splits

While auto-partitioning helps us in greatly reducing the number of split candidates, it sometimes creates poor splits. One simple scenario is pictured here. No matter which split plane is picked using auto-partitioning, some geometry will have to be split. If instead we were able to choose some other planes we could easily pick split planes that don't intersect anything.

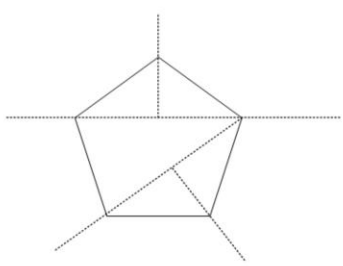
Auto vs. Arbitrary

Some times auto-partitioning can't choose well

2. Convex objects are worst case



auto-partitioning



arbitrary splits

The other common scenario where auto-partitioning fails (probably the more common scenario) is with convex objects. No matter how we choose split planes we'll create a completely unbalanced tree. If we could instead choose some arbitrary planes we could produce a balanced tree.

Arbitrary Split Possibilities

Pre-determined axes (cardinal)
- Test evenly space points
- Test polygon vertices

Plane from vertex through another edge

Hint planes

There are a few methods to limit possibilities for split planes. We have to figure out how to limit both the axis and position.

One of the most common methods of limiting the axis is to pick from some pre-determined ones, such as the cardinal axes. With this method we need to pick some split points. A simple choice is to try points evenly spread throughout the boundaries of the cell. Trying to split at a polygon's vertex is another good choice.

Another method of choosing an axis is to try one that goes from a vertex of one polygon through the edge of another. The idea with this method is to try to reduce splitting by forcing it to go through an edge.

Finally, manually placed hint planes can be a good choice. For example, if a model of digipen was made then hint planes could be placed at classroom borders.

Despite the flaws of auto-partitioning, it is much easier to build a bsp tree with one. It is also much quicker to build. For this class we'll be using auto-partitioning.

Choosing a split plane

Need some way to measure how good a split is

$$score = K * N_s + (1 - K) * Abs(N_f - N_b)$$

Ratio of splits to balance controlled by K

How many planes to test?

- All
- Fixed amount of random planes
- Fixed amount near median

No matter what method of picking split planes we use, we need some way of measuring how good a split is. The two easy values to measure is how evenly the tree will be balanced and how many polygons are split. Hence, a common scoring method is to take a weighted average of the two depending on what kind of tree we're after. In this equation N_s is the number straddling, N_F is the number in front, N_B is the number behind and K is a constant controlling the blending between the two.

So what kind of value should be picked for K? Well by creating a balanced tree we can limit worst-case performance, however this could cause us to create extra polygons by making more splits. Because of this a good value of K tends to be above 0.5. I recommend starting somewhere around 0.8 and tweaking it from there.

Now that we have a metric we just have to figure out which polygons to test. We could obviously test all of them and pick the best one, but that is very expensive (n^2 performance). If we're trying to get an even split then a good choice would be to pick some polygons near the median along some axis. This does require a sort though which puts us at $nlog(n)$. We can also just sample a random set of polygons and pick the best. This method tends to work well in practice with a relatively small number of picks (e.g. 5 random picks).

Polygon Classification

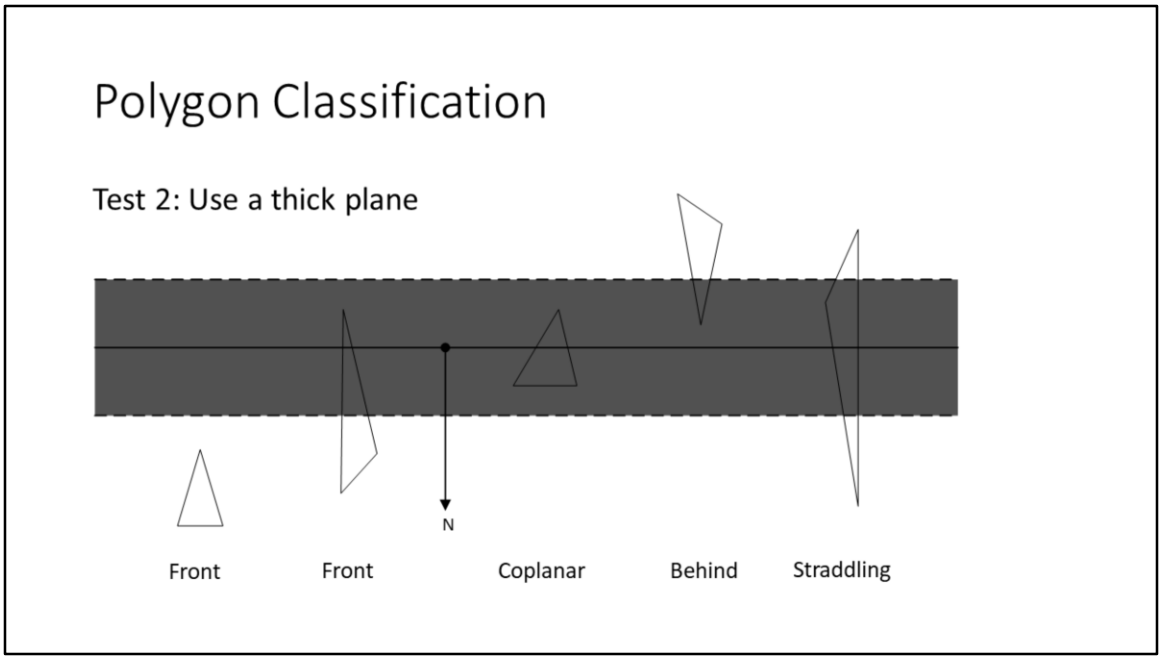
Test 1: Classify all points against the plane

Floating point can cause inaccuracies

Splits won't be correct

The simplest method of classifying a polygon is to just test all of the points against our split plane. If all points are on one side or the other then the polygon is wholly contained on that side. If all points are on the plane then it is coplanar. Otherwise we have a point on both sides so the polygon is straddling.

This test unfortunately doesn't work using the regular plane equation. Floating point numbers will cause a lot of inaccuracies that will screw up subsequent tests. For one we'll never get coplanar polygons. The other bigger issue is that if we split a polygon by a plane and re-classify it then we might get incorrect results. Event though we compute explicit points from a split that should produce a polygon that is in front of a split plane, numerical issues can make this not true.



The most practical method of dealing with numerical issues is to use a thick plane. With this any point that is within ϵ distance is classified as on the plane.
The classification now changes a bit with thick planes. If all points are coplanar then the polygon is still classified as coplanar. If all points are on the positive side or coplanar then the polygon is classified as in front. The same is true for the negative side. Now a polygon is straddling only if there are points on both sides.

So why is a thick plane important? First of all, this allows us to actually classify a polygon as coplanar. More importantly it'll fix polygon splitting issues. Now when we split we'll compute a point that will be classified as coplanar even if it ends up slightly behind the plane.

Polygon Splitting

Sutherland-Hodgman clipping algorithm

Previous Vertex (A)	Next Vertex (B)	Front Output	Back Output
InFront	InFront	B	None
Behind	InFront	I, B	I
InFront	Behind	I	I, B
Behind	Behind	None	B

The simple way to clip split polygons is to use a slight alteration of the Sutherland-Hodgman clipping algorithm. The algorithm proceeds one edge at a time keeping track of the start and end vertex's classification. Typically the algorithm discards anything that is clipped but we can alter it to output into one of two arrays.

The basic algorithm's outputs are displayed above. In this table the first two columns are the states of our two vertices on the current edge. Depending on these two states we'll output 0 to 2 vertices into either the front or back list. Note that the output 'I' is the intersection point of the current edge with the split plane.

This table doesn't deal with thick planes though and hence needs to be altered.

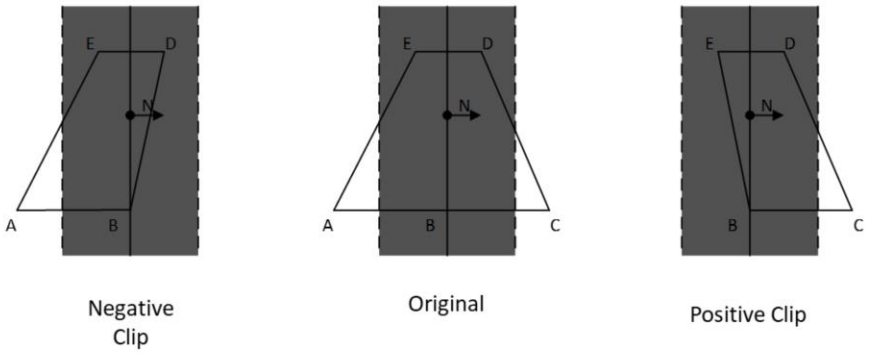
Polygon Splitting

Previous Vertex (A)	Next Vertex (B)	Front Output	Back Output
InFront	InFront	B	None
Behind	InFront	I, B	I
InFront	Behind	I	I, B
Behind	Behind	None	B
Any	CoPlanar	B	B

To fix the coplanar issue from before we should only have to deal with end points that are coplanar.

Polygon Splitting

Clipped geometry can overlap



Negative Clip | Original | Positive Clip

Unfortunately this can cause an issue where clipped geometry overlaps because we only handle the transitions that ended with a coplanar point.

Polygon Splitting

Full Clipping Table

Previous Vertex (A)	Next Vertex (B)	Front Output	Back Output
InFront	InFront	B	None
CoPlanar	InFront	B	None
Behind	InFront	I, B	I
InFront	CoPlanar	B	None
CoPlanar	CoPlanar	B	None
Behind	CoPlanar	B	B
InFront	Behind	I	I, B
CoPlanar	Behind	None	A, B
Behind	Behind	None	B

To properly handle all edge cases we have to take care of all 9 cases (3 x 3 states). Note: we can't just arbitrarily run this clip table as it'll duplicate vertices when a polygon is not straddling a clip plane. We only run the clipping algorithm if a polygon has some vertices in front and behind the thick plane.

Edge Intersection

How do we find the intersection point?

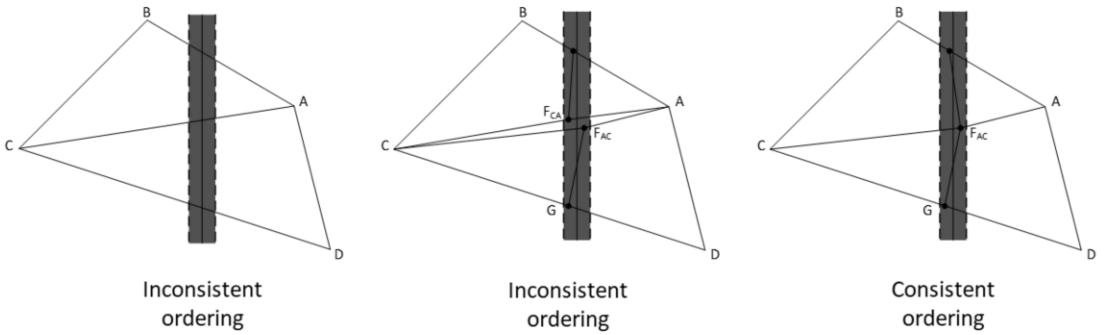
- Use Ray vs. Plane to get t
- Verify $0 \leq t \leq 1$ for segment

We haven't yet covered segment vs. plane so that we can find the intersection point. Luckily this is just re-using ray vs. plane. As long as the result of t is in the range $0 \leq t \leq 1$ then the segment intersects the plane. To find the intersection point just find the point along the ray at the given t value.

Robustness

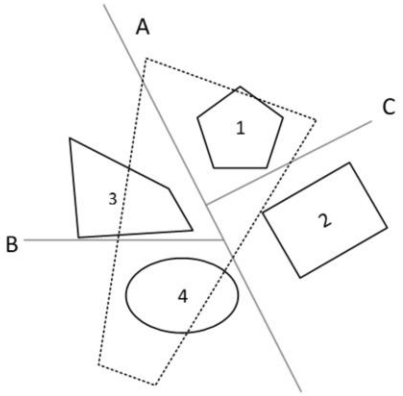
Edge ordering matters

Clip edges with consistent ordering to avoid holes



There is one more minor issue when it comes to robust clipping. When we clip a quad made up of triangles ABC and BCD, the edge BC will be clipped twice. There is a small difference between the clipping when it comes to the two triangles; in one it will be edge BC and in the other it'll be edge CB. Unfortunately there will be minor floating point inaccuracies that cause the two intersection points with the split plane to differ slightly, causing either minor cracks or overlaps in the mesh. Luckily, to deal with this is easy. We only have to ensure the correct ordering going from the front of the plane to the back. This can be achieve by just ensuring an edge is always clipped in the same order. As we often are iterating through vertices in an array, one method is to traverse the edge always from smaller to larger vertex index.

View Dependent Rendering



Far side objects can't obstruct near side

Bsp-trees were originally created to solve the ordering issues with rendering. If we render a bunch of triangles back-to-front then we don't need to do any form of z-buffering to get the correct image. We could also render back-to-front to properly deal with transparency. We can properly control the rendering order from any viewpoint using bsp-trees.

To render back to front with a bsp-tree simply requires you to identify which side the camera's position is on (doesn't depend on what direction the camera is facing). Let's call these the near and far sides. No geometry on the far side can obstruct any on the near side from view, hence it is always safe to render the far side first. Likewise no geometry that is contained in the node can obstruct any on the near side. Hence by recursively rendering far->coplanar->near we can render back to front.

Point query

Simply recurse down the side the point is on

```
void BspPointQuery(BspTree::BspNode* node, const Vector3& point, float planeEpsilon)
{
  // Classify the point
  IntersectionType::Type pointSide = PointPlane(point, node->mPlane.mData, planeEpsilon);
  // Check for front/coplanar
  if(pointSide == IntersectionType::Inside || pointSide == IntersectionType::Coplanar)
    BspPointQuery(node->mInsideNodes, point, planeEpsilon);
  // Check for back/coplanar
  if(pointSide == IntersectionType::Outside || pointSide == IntersectionType::Coplanar)
    BspPointQuery(node->mOutsideNodes, point, planeEpsilon);
}
```

As mentioned before, bsp-trees can be used to represent the boundary of an object. With this any point can be classified as inside or outside a mesh as long as it is water-tight, even if it is concave. To do this the bsp-tree needs to be constructed as a solid-leaf tree (or at least one who's leaf nodes represent the boundary of the mesh).

Simply iterate through the side the point is on until a leaf node is reached. If that leaf node is on the positive side of the mesh's surface then we were in an empty node and there's no intersection. If we're on the negative side then we're in a solid node and the point is contained. The only tricky part to deal with is if the point is coplanar at some point. To properly deal with this both sides of the tree have to be traversed. If the results down both sides are the same then the point is that classification, otherwise it is coplanar.

Ray Query

Ray-casts are a lot trickier than you'd think!

Basic Idea:

- Go down the side the ray is on
- Check triangles in plane
- Go down the opposite side

Ray queries on a Bsp are a lot trickier than they seem. I'll slowly build up to all we need to deal with (as it's quite a lot) but I'll start with something really simple.

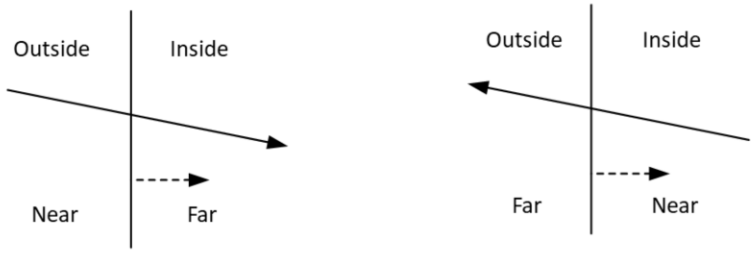
The very basic idea is that we need to traverse down the tree depending on which side the ray is on. If the ray cross the plane then we have to traverse both.

Ray Query

First some terminology:

The near side is the one that contains the ray start

The far side is the other side



*Near/Far is independent of Inside/Outside

Ray queries on a Bsp are a lot trickier than they seem. I'll slowly build up to all we need to deal with (as it's quite a lot) but I'll start with something really simple.

The very basic idea is that we need to traverse down the tree depending on which side the ray is on. If the ray cross the plane then we have to traverse both. To avoid confusion, I'll be referring to the two sides of the plane here as the near and far side. The near side is whichever side the ray starts on while the far side is the other. Note that this is independent of which side is the front or back.

So the simplest traversal we can write is this:

Determine Near/Far sides
Recurse down the near side
If we hit the split plane:
  Check the triangles in the plane. If we hit one then update t (if closer) and return true;
  Otherwise check the far side

Ray Query

Basic traversal is just like view dependent rendering (in reverse)

```
bool BspRayCastSimple(BspTree::BspNode* node, const Ray& ray, float& t)
{
  // Determine near/far
  // Recurse down the near side
  //   If we hit a triangle on the near side then return true
  // If we hit the plane
  //   Check the triangles in the plane, if we hit any update t and return true
  //   Otherwise check the far side and return its result
}
```

Will this cull anything though?

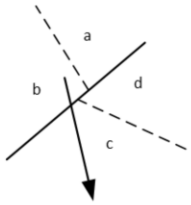
Now we can write the most basic of traversals. The basic idea here is the same as with view dependent rendering. The order we can possibly hit anything is near, coplanar, far. If we hit anything in an early stage then by definition the later stages will produce a later t-value so why even bother checking.

This is of course a very simple version of ray casting that needs improvement. The first thing to look at here is will this test actually cull any of the triangles.

Ray Query

For this to be worth it we need better that $O(n)$

Half-spaces behind the ray start will be culled



Is there something that should be culled but isn't?

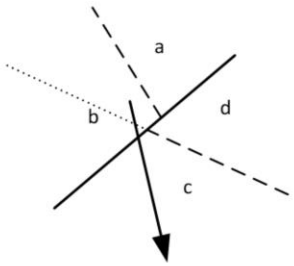
Now let's look at how good of a ray-cast this actually produces. Remember we could easily just check all of the original triangles without the bsp-tree, so this needs to out-perform the linear pass to be worth it (especially since we might produce more triangles when splitting).

The easiest to see is that since we have a tree, if a plane is not hit by the ray then it will be culled. This happens when a ray either travels away from the plane or is parallel. So with any luck we could cull a large portion of space. For instance, in the above picture the entire half-space of 'a' is culled an not tested (including all children).

This does beg the question, what other scenarios should cull something?

Ray Query

We need to not visit nodes that should be clipped by our parent



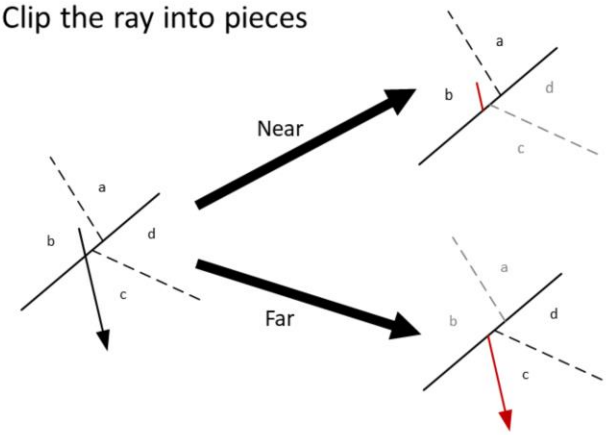
The ray doesn't intersect the region of 'd' but traverses into it

The answer is simple: bsp-trees recursively divide space in half, but planes are infinite. There are scenarios where a ray can hit a plane (remember it's infinite) and traverse down both sides when it should have been clipped by its parent. In fact, the previous picture I gave shows this. I added another version of the same picture to illustrate this.

The ray hit's the split plane between c and d but on the opposite side of the parent. Clearly the ray doesn't actually intersect any of d so why are we traversing into it?

Ray Query

Solution 1: Clip the ray into pieces

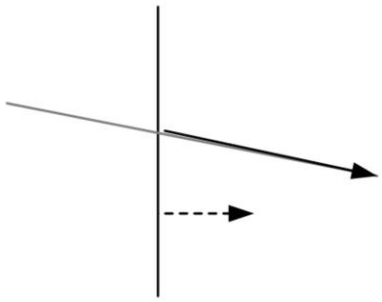


The basic answer is quite simple. We have a half-infinite ray and an infinite plane. It's not too easy to try and clip a plane to determine what it's valid ranges are (within its parents). It is easy however to clip a ray into line segments. Simply compute the t-value of the ray with the split plane and then clip it into two segments, sending each one down its respective side.

Ray Query

Problem 1: Numerical Robustness

The clipped point can slightly drift from the original line



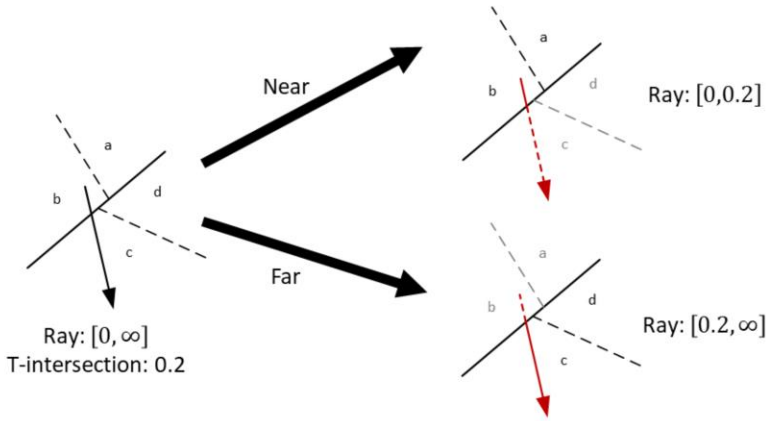
Unfortunately floating point numbers cause problems with this solution. Each time you clip a segment to get a new segment the calculated position can drift slightly off the original line. That means the further down the tree you go the worse your results can get. This means you can actually miss triangles with this method that would be hit by the original ray.

As a small example try these numbers:
Ray: start(1, 0, 0), direction(1, 1, 0) normalized
Plane: normal(1, 1, 1) normalized, point (1.1, 0, 0)

If you compute the t intersection you should get t = 0.0707106814.
If you use this t to get a point on the ray you can then test that point's distance from the ray start, (essentially the t-value). In this case you'll get 0.0707106441 which is just slightly different.
While this might not seem like much this can compound and get worse the further away from the origin you move (try a plane at 1,000,000 and the error goes up to 0.06).

Ray Query

Solution 2: “Clip” the ray into pieces with t-values (tMin/tMax)



We still need to clip the ray and small errors are ok, the biggest problem is the compounding error terms. The solution to this is luckily really easy. Instead of directly clipping the ray into line segments, we can implicitly clip the ray using t-values. Now the most error we can occur is from one sampling of the line, but no errors will build up as we recurse through the tree.

Ray Query

So how do we efficiently cull half-spaces?

We have: ray start, ray direction, tMin, tMax, tPlane

*I highly recommend labeling tPlane different than the result t

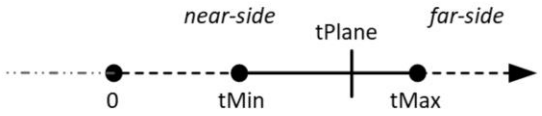
Using tMin, tMax, and tPlane we can come up with 4 cases

Now that we've determined how to clip our ray so we can cull half-spaces the question is how do we use the information at hand to do so? We technically have 5 pieces of information: the ray start, ray direction, the split plane, tMin, and tMax. With the ray and plane we can get two more pieces of information: the near and far sides and the t-value of the plane.
With the near and far sides labeled, we can actually determine which sides have to be traversed by just using tMin, tMax, and tPlane. If we look at it we have 4 cases to consider.

Do note that I highly recommend keeping a separate variable for the tPlane, that is not mixing it with the final result t-value. This was a big problem I saw last semester

Ray Query

Case 1: $tMin \leq tPlane \leq tMax$



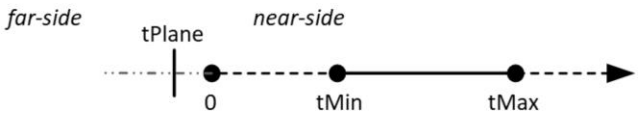
The clipped ray hits the split plane:

- Visit the near and far side
- Check geometry in the plane
- Don't forget to update the tMin/tMax during recursion

The first case is the most obvious: the computed t-value against the plane is within our clipped ray. In this case the clipped ray hits the split plane and we have to traverse down both sides. We also need to check the geometry in the node since the ray touches the plane. In this case we also have to send each side the correct tMin/tMax values. For the near side it gets [tMin, tPlane] and the far side gets [tPlane, tMax].

Ray Query

Case 2: t$Plane < 0$



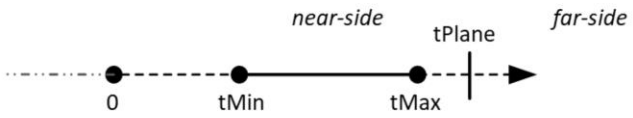
The split plane is behind the ray:

Only traverse the near side

The next case is one we've already seen. If the t-value of the split plane is negative then that means the ray starts on the opposite side and we only traverse the near side. The values of tMin and tMax don't change.

Ray Query

Case 3: $tMax < tPlane$



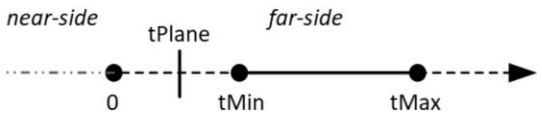
The clipped ray can't reach the split plane:

Only traverse the near side

The next case is that our clipped ray can't quite reach the split plane. In this case we only traverse the near side while tMin and tMax stay the same.

Ray Query

Case 4: $0 < tPlane < tMin$



The split plane is behind the ray:

Only traverse the far side

The last case is probably the hardest to visualize. If the t-value of the plane is between 0 and tMin then the ray start is on the opposite side of the split plane than the clipped ray. In this case we only traverse the far side as shown by the picture. Remember the near and far side are based solely upon which side the ray start is on. Once again, the t-value range doesn't change.

Ray Query – Edge Cases

Unfortunately there's 3(ish) major edge cases
These are almost entirely caused by thick planes

If we ignore them we'll miss triangles we should hit!

Quick Note:
Only visit both side and a node's geometry if the ray hits the plane

This ray-cast is nice and simple, but unfortunately there's a lot of edge cases we need to deal with. If you write the presented tests there will be several cases where the wrong t-value will be presented. You might be wondering how any cases could be missed. With the previously shown split planes there are not really any edge cases, but we've constructed a different bsp-tree than shown.

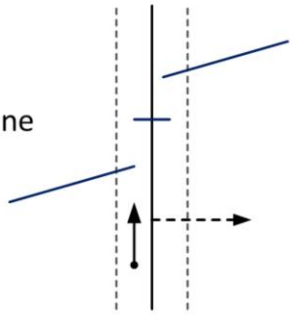
In particular one thing breaks everything: thick planes. When we built our bsp-tree we allowed some "incorrect" triangles in our tree because we have a thick plane. This means that we could cull out half-spaces (and hence triangles) that we shouldn't if we don't consider the thick planes.

One quick note before continuing with the edge cases: the only time we should ever check the geometry in a node is if the "clipped" ray hits the thick plane. It's also only in this case that both sides should be visited. This means that only 1 of the previous 4 cases should check the geometry in the plane (case 1).

Ray Query – Edge Cases

Case 1: The ray start is coplanar

Visit both sides and the geometry in the plane



**triangles are blue*

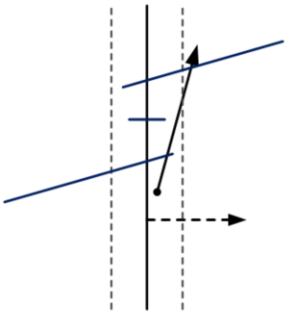
With the 4 major cases that we covered we've ignored thick planes. With the addition of thick planes the first thing we have to consider is what happens when the ray start is coplanar? In this case it doesn't matter what the t-value with the plane is, we have to check everything. Since the ray touches the thick plane we have to check the triangles in the plane, but we also have to recurse into both sides.

When recursing we need to pass down tMin/tMax values for each side. Since we're coplanar we can just pass the same tMin/tMax down both sides. The reason we don't compute the plane's t-value will become clear after a few more edge cases.

In the above picture we'd hit a triangle on the outside of the plane first, but it's not hard to construct a scenario where we hit a triangle that is coplanar or on the inside first.

Ray Query – Edge Cases

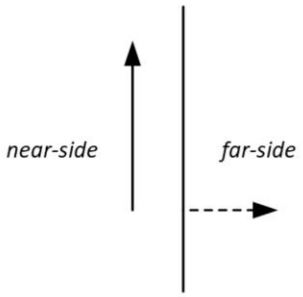
Note that the ray can still hit triangles
on the opposite side of the non-thick plane



It's important to see why we still have to traverse both sides of the thick plane. As triangles are classified as in outside split plane even if one point is in front (but within epsilon) there's a chance that a triangle can be hit on the opposite side it was categorized on. This is the primary reason we have to traverse both side even though the ray start is coplanar.

Ray Query – Edge Cases

Case 2: The ray is parallel to the plane
Determined by RayPlane returning false



Only recurse down the near side
*Assumes we've already dealt with coplanar cases

The next edge case to consider is what happens if RayPlane returns false?
There's 2 possibilities here:

1. The plane is behind the ray
2. The ray is parallel to the plane.

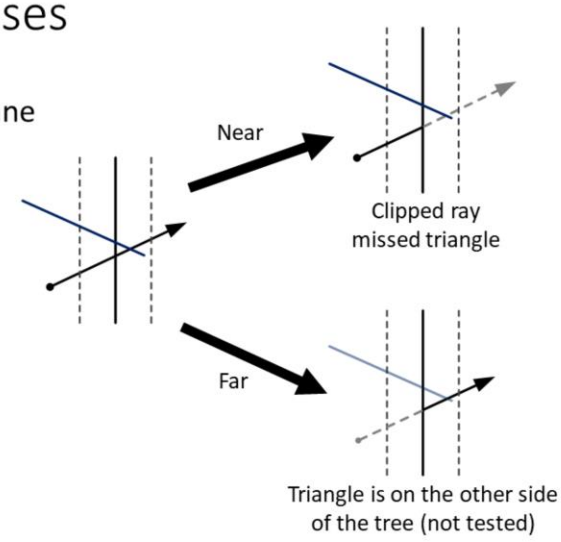
In case 1 we want to only recurse down the near side (evident from regular case 2). In case 2 we can't compute a t-value, but we still need to recurse down the near side since there could be more things hit there. This means in either case we simply recurse down the near side.

Note: this assumes that we've already checked for the ray start being coplanar, otherwise we'd have to visit both sides.

Ray Query – Edge Cases

Case 3: The ray hits the thick plane

A ray can hit a triangle on
the opposite side of the plane

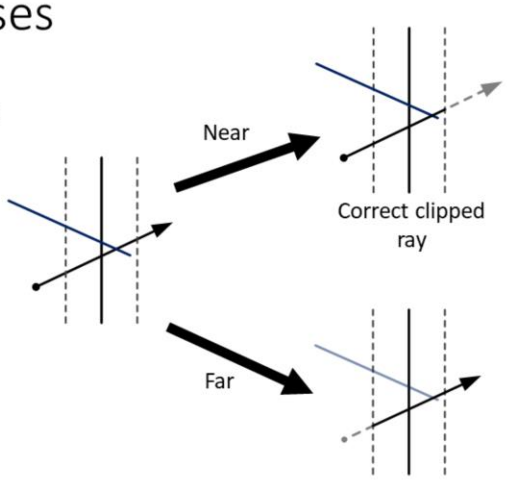


Now for the biggest case. This case technically affects case 2 as well but it's easier to view it independently as it can happen without a coplanar ray start.

The most important thing to realize about this case is that we need to account for the plane being thick in all areas of the test. The first thing to notice is that a ray can easily hit a triangle that was coplanar but extended to the other side of the plane. In this case if we improperly clipped the ray then we'll miss the triangle.

Ray Query – Edge Cases

A properly clipped ray extends to the edge of the thick plane

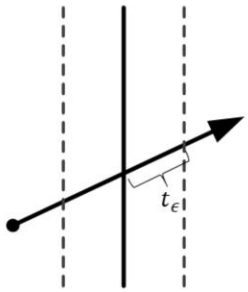


To fix this case we just need to realize that we can have geometry from any side in the region of the thick plane hence we have to clip our ray with this in mind. That means we need to actually clip our ray such that it always overlaps the entirety of the thick plane.

Ray Query – Edge Cases

How do we compute the proper clipped ray?

Easiest way is to compute a t-epsilon:

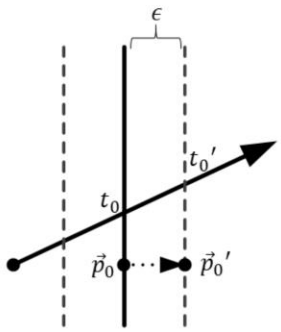


There's a few different ways to do this but I personally believe the easiest is to somehow compute what the plane's epsilon value means when it comes to the t's ray value. This can be thought of as the half length of the ray that would be clipped to be coplanar to the plane.

Ray Query – Edge Cases

Computing t_ϵ method 1:

Compute the intersection time of the ray with the plane offset by ϵ



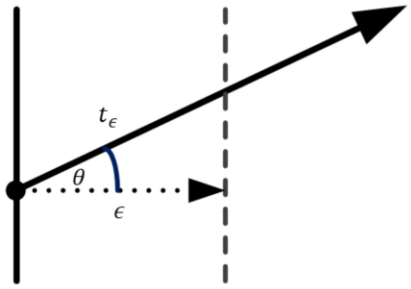
$$\vec{p}_0' = \vec{p}_0 + \epsilon\hat{n}$$
$$t_\epsilon = t_0' - t_0$$

There's two methods that are equally easy to compute t_ϵ. The first method is to realize that we can compute a t-value of our plane offset by an epsilon in the direction its normal. This simply amounts to getting a point on the plane (given $\hat{n}$ and d we can do this as $d\hat{n}$) and then offsetting in the direction of the normal by a fixed amount. We can then use that point and the old normal to compute the new plane.

Note that as a shortcut the only thing that should change is d which will be computed as $Dot(\vec{p}_0', \hat{n})$ which can be expanded to $Dot(\vec{p}_0 + \epsilon\hat{n}, \hat{n})$. This further simplifies to $Dot(\vec{p}_0, \hat{n}) + Dot(\epsilon\hat{n}, \hat{n})$ which is also equivalent to just saying $d' = d + \epsilon$.

Ray Query – Edge Cases

Computing t_ϵ method 2: Trig



$$\cos(\theta) = \frac{\epsilon}{t_\epsilon}$$

$$t_\epsilon = Abs\left(\frac{\epsilon}{\cos(\theta)}\right)$$

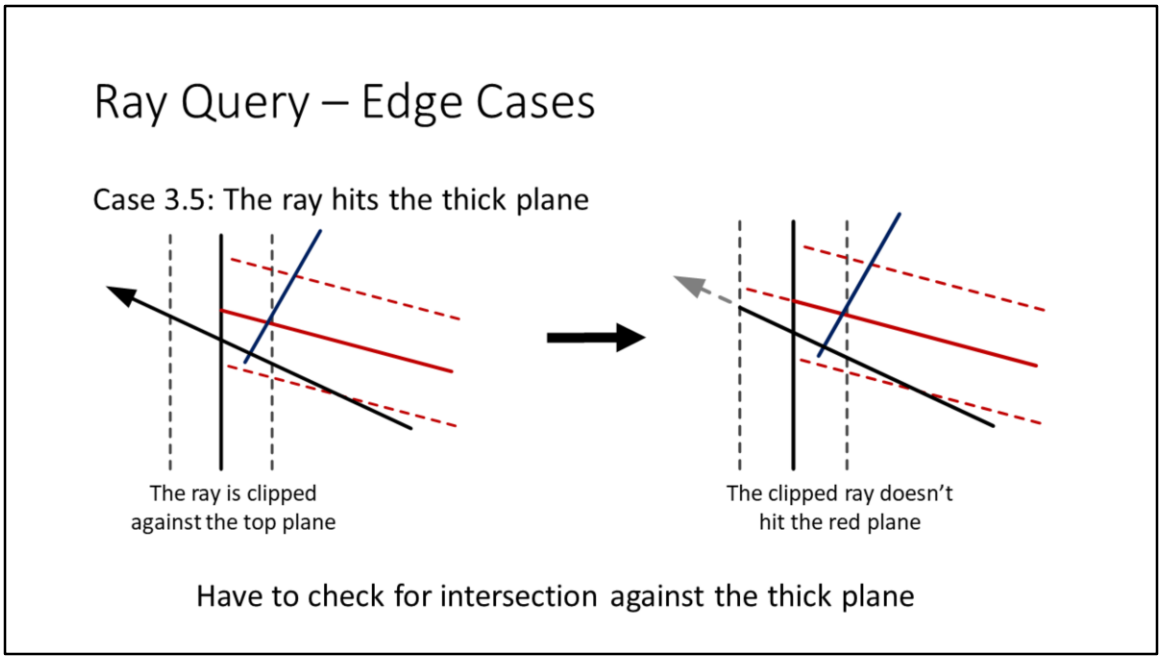
The second method is equally as easy although cheaper. We just have to realize that the plane normal and the ray direction will make a triangle with a right angle where it meets the epsilon plane. We can then use simple trig identities to compute t_ϵ and we can easily compute cos with the dot-product. Be careful to abs the value though as the ray could be pointing the opposite direction of the plane normal.

Ray Query – Edge Cases

Case 3: The ray hits the thick plane

Will expanding tMin and tMax take care of all cases?

Now are there any more edge cases where we can miss triangles even though we expand tMin and tMax? The short answer is yes (obviously or I wouldn't be asking it). This case is much hard to come up with but it can happen.



This picture is a bit of a mess but it's the simplest case I could come up with. Image we have a tree with the black plane at the root, with the red plane (it's also thick) underneath, and then finally a triangle that belongs to the top side of the red plane. When we clip the ray against the black plane we'll have a new tMin and tMax that we use when checking the ray against the children split planes. In this case though the clipped ray doesn't hit the red plane and we won't traverse down the far side of the plane.

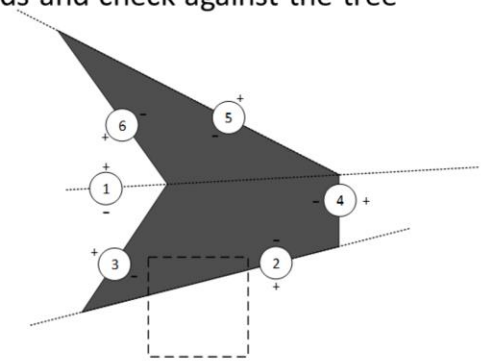
It should be easy now to see what our error was, we shouldn't be checking to see if we hit the plane, we need to see if we hit the thick plane. Luckily this is easy to do as we only have to expand the ranges in the $tMin \leq tPlane \leq tMax$ check to $tMin - t_\epsilon \leq tPlane \leq tMax + t_\epsilon$.

With all of this in place we can finally catch all cases in ray vs. bsp-tree.

Object Query

Aabb vs. Bsp Test 1:

Split aabb into quads and check against the tree

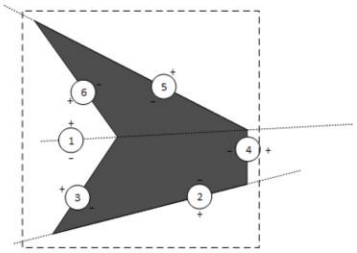


Generic object queries can be performed on a bsp-tree, but they aren't as easy to get right. To start with let's look at checking an aabb against a bsp tree. If we have a solid-leaf representation (or just a way to know if a leaf is the positive or negative side) then we can split the aabb into 6 polygons. We can send each polygon down the tree and if any part ends up in a solid leaf then there is intersection.

Object Query

2 Problems:

1. Slow to test all 6 planes
2. Doesn't work if aabb contains entire tree



Can fix #2 by making a bsp for the aabb

There's 2 fundamental problems with this test though. The first one, and the harder one to address, is that it's slow. We have to send 6 polygons to get clipped against the tree for an aabb. A more complicated shape would have to send all of its faces down making even more checks. The second problem is that we're only checking if a query shape's polygon ends up in a solid leaf. If the query shape completely contains the tree (or an isolated mesh within the tree) then it will not report any intersection. Luckily this is easy to work around. We can build a bsp tree for the query shape and intersect the two trees against each other. The basic idea though is to just perform set operations (intersection) and see if they're non-empty. This happens to be what we'll be talking about in the next part.

Object Query

Fixing the performance problem is harder

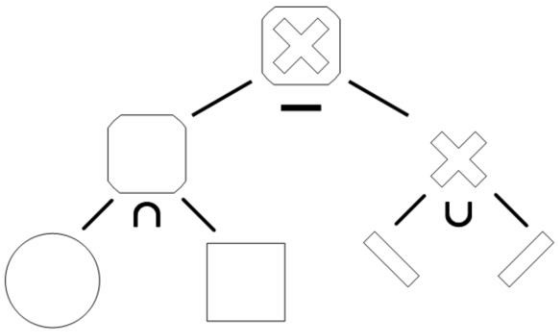
Expand bsp by the aabb (Minkowski sums)

See orange book for more details

Fixing the performance issue unfortunately is not too easy and requires something we haven't talked about yet. The basic idea is to using Minkowski sums to expand the bsp-tree by the query shape and then check a point against it. This requires more planes to be inserted into the bsp tree though. For more details check out the section in Real-Time Collision Detection (under Using the Bsp Tree).

Constructive Solid Geometry

Build complex shapes out of Boolean operations



One really cool thing that we can do with bsp trees is CSG operations. CSG stands for Constructive Solid Geometry, which is creating complex geometry by using Boolean operations to combine shapes. The three main operations used in CSG are union (or), Intersection (and), and subtraction.

BSP Functions for CSG

Need to implement 3 functions for CSG operations:

1. AllTriangles
2. Invert
3. ClipTo

To perform CSG operations with bsp trees we just need to add 3 functions: Invert, ClipTo, and AllTriangles.

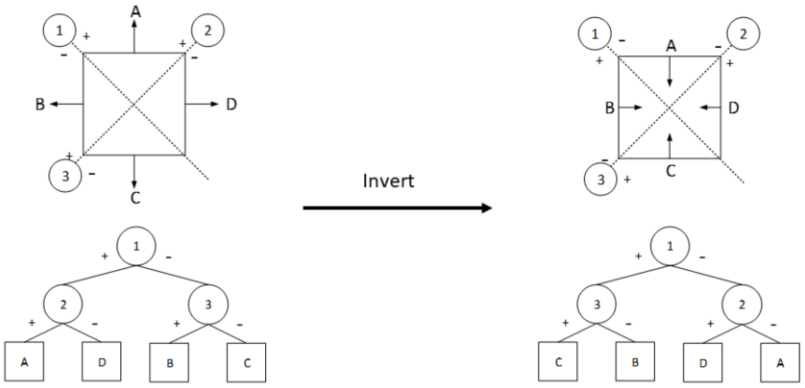
1.AllTriangles Function

Append all triangles in the tree to a list

AllTriangles is the simplest function needed. This doesn't really serve any geometric purpose, but just collects all triangles in the bsp-tree into a list. This function is needed to combine the results of various operations into one list of triangles (our new shape).

2.Invert Function

Flips positive and negative space



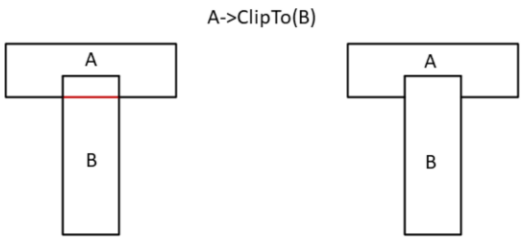
The invert function simply flips all positive and negative spaces in the tree.

This entails 3 steps for each node in the bsp tree:
1. Flip a split plane. This is just multiplying the plane by -1.
2. Flip any contained geometry. With a triangle this amounts to just swapping two points (effectively flipping the sign of the triangle's normal).
3. Swap the front and back-side pointers. When the tree is inverted what was the positive sub-tree becomes the negative sub-tree.

These steps are performed recursively on each node.

3.ClipTo Function

Removes all geometry of A inside B



A->ClipTo(B) will remove all geometry of A that is inside B.

ClipTriangle

1. Recursively split down the tree
2. Remove any triangle in a solid leaf

Coplanar Triangles?

Group with front triangles if the normal points the same way as the split plane

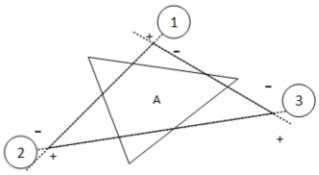
Before going into the finer details of the ClipTo function, we need to understand how to clip a triangle against a tree.

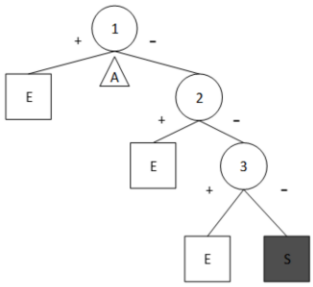
We need to recursively split a triangle against all nodes in the tree. Any triangles on the front side of a split plane are recursively clipped to the front children and vice versa for the back side. Any triangle that ends up in a solid leaf is fully contained within the bsp-volume and is deleted. How do we determine if a triangle ends up in a solid leaf? Even if we don't explicitly construct a solid-leaf tree we can classify any geometry as being in a solid leaf if it ends up on the back side of a split plane that has no back side children.

There's one extra caveat here: How do we deal with coplanar triangles? Group them with either the front or back triangles depending on the direction they are facing. If a triangle's normal faces the same way as the split plane's normal then group it with the front triangles, otherwise the back triangles.

ClipTriangle

Clip triangle A against the tree

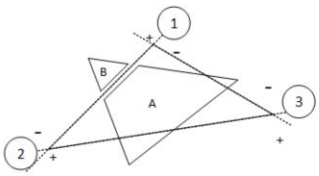


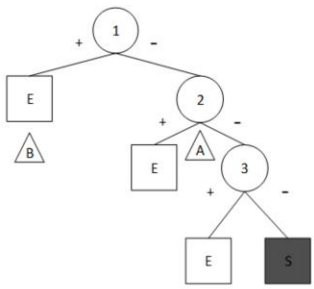


Now for an example. We have a bsp-tree representing a triangle (because it's simpler). We are clipping the triangle A against this tree.

On the left we have the geometry that we're actually splitting and on the right we have a diagram of the tree. Note that any small triangles on the diagram on the right are where various triangles are in the tree traversal.

ClipTriangle

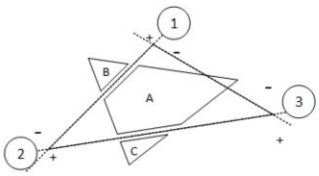


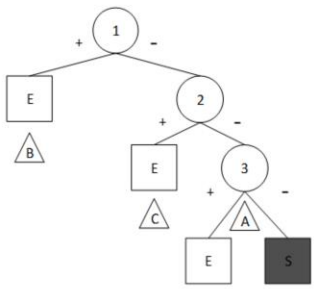


First we split A by plane 1. B is on the positive side of the split plane so it traverses down the positive side of the tree. Likewise A recurses down the negative side. Note that B reaches an empty leaf here, this means that B will not get clipped and will end up in the final shape.

Also note that I'm not bothering to turn A from a polygon to a triangle. This algorithm can either work with polygons or you can re-triangulate at each step.

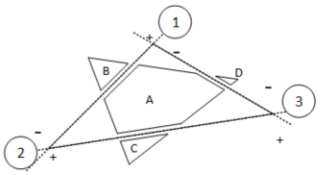
ClipTriangle

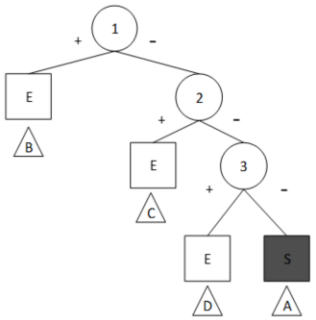




Now we continue to recurse with A, this time splitting against plane 2. Triangle C ends up in an empty leaf so it's is finished recursiing. A needs to continue down the tree though.

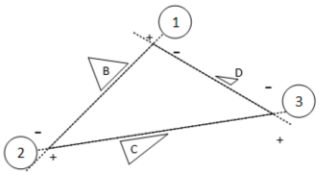
ClipTriangle

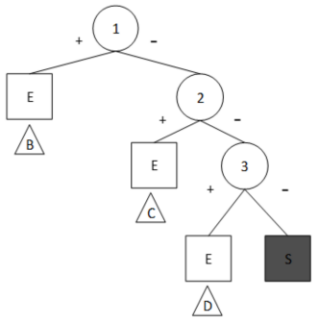




Now we split against plane 3. Finally we end up with A being in a solid leaf node, meaning it is fully contained inside the mesh of the bsp tree.

ClipTriangle





Now we can remove A and get the resultant shape after clipping. In this case we get 3 disjoint triangles, but we could've gotten pieces that were still connected.

3.ClipTo Function

Recursively clip all triangles against another tree

```
void BspNode::ClipTo(BspNode* node)
{
  mTriangles = node->ClipTriangles(mTriangles);
  // Recursively clip all children nodes
  if(mFront != NULL)
    mFront->ClipTo(node);
  if(mBack != NULL)
    mBack->ClipTo(node);
}
```

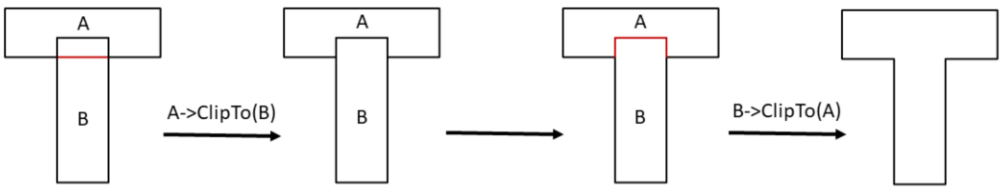
*ClipTriangles() is the same as ClipTriangle but with a list

To implement the ClipTo function now we just need to clip all of the triangles of tree A against tree B. We could do this by collecting all triangles of A using the AllTriangles function that then clip each one, but we want to preserve the bsp structure of tree A. Because of this we can just recurse through tree A and clip any triangles against B.

Note that we can't do any optimizations by clipping a triangle in A against a portion of B; we have to send each triangle in A down the root of tree B.

Union

Remove all of A inside B and vice-versa



To perform a union we want remove all geometry of A that is inside B and vice-versa. This is now easy to do with the ClipTo function.

Union

```
void Union(Node* nodeA, Node* nodeB, TriangleList& results)
{
  nodeA->ClipTo(nodeB);
  nodeB->ClipTo(nodeA);
  nodeA->AllTriangles(results);
  nodeB->AllTriangles(results);
}
```

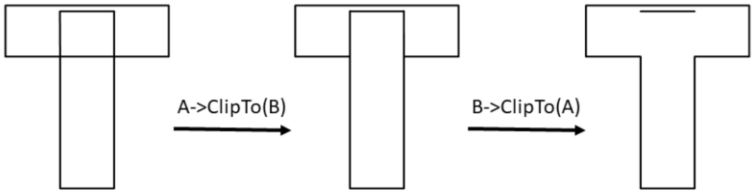
How does this work for coplanar faces?

To finish of the Union function we just have to take the results from each clip operation and merge them together using the AllTriangles function.

There is one extra thing to consider here though, what happens to coplanar faces?

Union: Coplanar Faces

Coplanar faces will be left behind



The current union algorithm will not work correctly for coplanar faces. We'll end up with a duplicate copy of geometry.

Union: Coplanar Faces

Remove any geometry in B left in A by inverting

```
void Union(Node* nodeA, Node* nodeB, TriangleList& results)
{
  nodeA->ClipTo(nodeB);
  nodeB->ClipTo(nodeA);
  // Remove coplanar faces
  nodeB->Invert();
  nodeB->ClipTo(nodeA);
  nodeB->Invert();
  nodeA->AddAllTriangles(results);
  nodeB->AddAllTriangles(results);
}
```

Luckily this is easy to fix: we can simply invert B and then clip it again to A. Any geometry in B that was outside of A will not be affected, only geometry that was inside and previously left alone: aka coplanar geometry.

Boolean Logic

Can relate functions to Boolean operations

$$Invert(A) \equiv \sim A \qquad Union(A, B) \equiv A \mid B$$

Build up other Boolean functions with the “Not” and “Or” operators

So how can we build up the other CSG operations? The answer is to use Boolean logic!

If we think about it, a union is equivalent to an “Or” operator, that is a point is in the union of A and B if it is in A or it is inside B.
Likewise we can think of the invert function as being a “Not” operator as it reverses a point test.

Now we can build up more complicated Boolean operations by using these building blocks.

Ideally we can build up Intersection and Subtraction by relating them to Boolean operations. Intersection is easy enough as it’s just the “And” operator, that is a point is in the intersection of two shapes only if it is inside A and inside B.

So how do we use Boolean logic to make an “And operator”?

Intersection

$Intersection(A, B) \equiv A \,\&\, B$

	F, F	T, F	F, T	T, T
$A \mid B$	F	T	T	T
$A\&B$	**F**	**F**	**F**	**T**
$\sim A \mid \sim B$	T	T	T	F
$\sim(\sim A \mid \sim B)$	F	F	F	T

$A \,\&\, B \equiv \sim(\sim A \mid \sim B)$

Using a truth table is the easiest way to see how we can represent $A\&B$.
Now we can just write the intersection using the Invert and Union functions.

Subtraction

$Subtraction(A, B) \equiv A \ \& \sim B$

Likewise we can write $A - B$ using Boolean operations. This one is a bit harder to visualize at first. One method is to just realize that a subtraction is equivalent to saying we want all of A that is not in B, or $A\&\sim B$.

Questions?