Object-Environment Collision Detection using Onion BSPs

Stuart Golodetz

1 Introduction

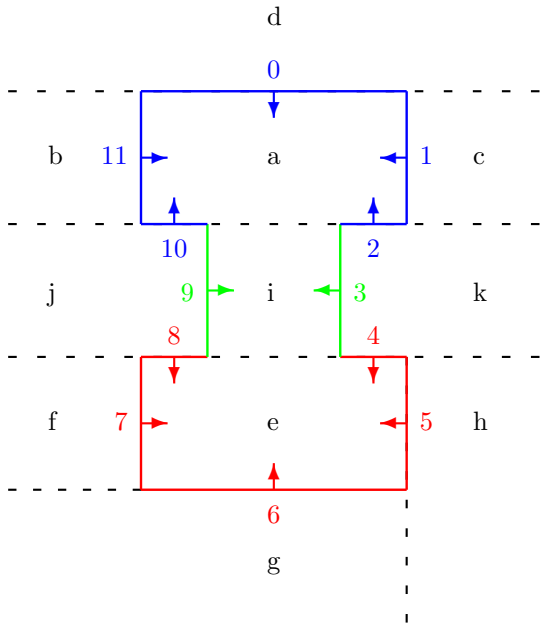
In my last article [7], I described how to automatically generate navigation meshes to support the navigation of agents around 3D environments (e.g. game worlds), as implemented in my homemade *hesperus* engine [5]. However, there is far more to such navigation than simply mesh generation: it remains to be shown how to determine where (if anywhere) an agent can be found on the mesh and how to make best use of the mesh when allowing both user-controlled and AI agents to move around the environment. Agent movement must necessarily interact with an implementation's physics system, since the navigation mesh only covers the *walkable* surfaces of the world and there is a need to ensure that agents are simulated correctly even when they are not on the mesh. In particular, any implementation needs to ensure that agents do not collide with either the world or each other, and that the effects of forces such as gravity are properly applied to them when not on the mesh. For that reason, before tackling the agent movement problem itself, it is important to take a step back and look at how the physics system in *hesperus* works.

As a first step, I want to focus this article on a way of detecting collisions between objects (including agents) and their environment, via the construction of a special binary space partitioning (BSP) representation of the world that I call an *onion BSP* (for reasons that will be explained). Onion BSPs are a simple extension of BSP trees for multiple configuration spaces, based on the ideas of van Waveren for *Quake III Arena* in [17]. The collisions (also known as *contacts*) that we detect can be fed to the rest of the physics system for later resolution. Future articles will focus on how to detect object-object collisions using a technique called Minkowski Portal Refinement [15], and how to combine the techniques into a rudimentary physics system, before we return to the original problem of agent movement. Readers who are interested in a more general look at games physics engine development are advised to take a look at the excellent (and aptly-named) book by Millington on the topic [13].

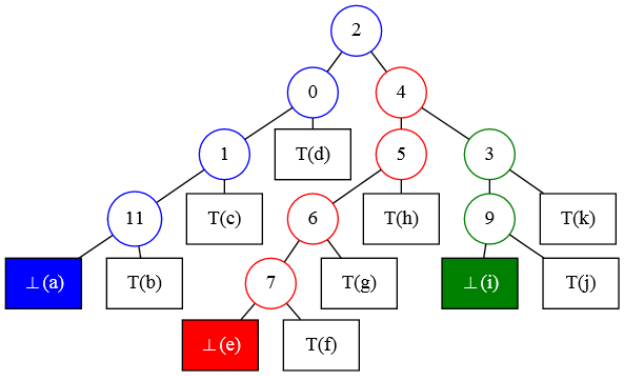
The organisation of this article is as follows: in §2, I briefly revisit the ideas behind binary space partitioning; in §3, I describe how to construct onion BSPs; in §4, I describe how to perform (swept) collision detection between objects and onion BSPs using an algorithm for finding the first point at which a half-ray crosses a wall in the world; and in §5 I discuss the limitations of this approach and briefly compare it to a related approach that achieves the same effect by moving the planes of a normal BSP at runtime.

2 Binary Space Partitioning

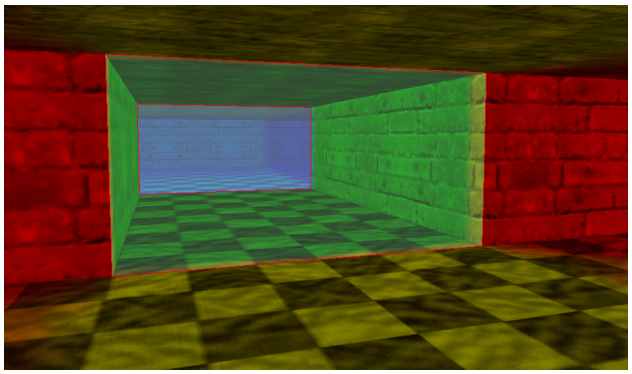
Binary space partitioning is a technique for representing n-dimensional space as a binary tree (known as a BSP tree) by recursively dividing it into two using *hyperplanes* (the n-dimensional generalisation of planes). It was originally introduced by Fuchs et. al. [4] in 1980, and saw widespread use in first-person games of the *Quake* era (e.g. see [1]) as a way of representing 3D polygonal game worlds, most notably because it provided a way of rendering a world's polygons in either back-to-front or front-to-back order [4, 9] without the need for a z-buffer on the graphics card (z-buffers once used to be quite costly). As graphics cards have matured, commercial games have moved away from binary space partitioning as a rendering approach, because traversing a BSP tree is relatively slow in comparison to simply throwing



(a)



(b)



(c)

Figure 1: A BSP example for a simple 2D world with two rooms, connected by a corridor: (a) shows a top-down view of the world, where the arrows represent the facings of the world polygons and the dashed lines represent the split planes chosen when constructing the BSP in (b); (b) shows the BSP tree that is constructed for the world based on the chosen split planes; (c) shows what a 3D version of the world looks like in *hesperus*, with portals (doorways) rendered as translucent polygons to illustrate the boundaries between the empty leaves (a, e and i) of the BSP. (Note that the 3D version actually has additional floor and ceiling polygons, but we ignore that here for the purposes of explanation.)

large numbers of triangles at the graphics card and letting the z-buffer handle the rendering order, but BSP trees remain interesting as a basis for collision detection and constructive solid geometry techniques [3, 11].

An example BSP tree is shown in Figure 1. Each node of the tree represents a convex subspace of the world being partitioned; moreover, the leaves of the tree represent a *partition* of the entire space, i.e. they are mutually disjoint and their union is equal to the space. Each branch node has an associated split plane (a line with facing in 2D) that divides the subspace represented by the node in two. Each leaf node contains the polygons (line segments in 2D) that fall within the subspace it represents, and carries a flag that indicates whether the subspace represented by the leaf is empty (i.e. navigable by an agent, denoted as $\bot$) or solid (non-navigable, denoted as $\top$). The BSP tree as a whole can be used to decide whether or not any given point in the world lies in empty or solid space in $O(h)$ time, where h is the height of the tree, by the simple means of classifying the point against the split planes in the tree, starting from the root, and recursing down the relevant side of the tree at each stage until hitting a leaf.

Constructing a BSP tree for a polygonal world is also done recursively, starting from the set of all the polygons in the world. At each recursive step, one of the current set of polygons whose plane has not been used further up the tree is chosen as the *split polygon*, and its plane is used to split the other polygons into two sets, one of polygons that are in front of the plane and one of those that are behind it. (If no suitable split polygon can be found, then we create a leaf of the tree to contain the current set of polygons and return.) If a polygon straddles the plane, it is split, with its two halves being placed in the appropriate sets. If a polygon lies on the plane, it is put into either the front or back set based on the orientation of its normal with respect to the plane. The two sets of polygons are then processed recursively to construct the subtrees of the current node. Finally, a branch node is constructed from the split plane and the two subtrees.

An extremely detailed description of BSP construction, together with diagrams that clarify precisely how the process works, can be found in [8]; readers may additionally wish to take a look back at a previous article I wrote for *Overload* [6].

3 Onion BSPs

As mentioned in the previous section, standard BSP trees can be used to determine whether individual points are in empty or solid space; moreover, this extends to line segments – there is a relatively straightforward BSP algorithm that will allow us to find the first transition point at which a line segment crosses from empty to solid space (e.g. see [2, 10, 16]). This can form the basis for a simple collision detection scheme for *point-based* agents – at each frame, we can test the line segment representing an agent's proposed movement for that frame against the tree, taking the first transition point as the point of collision if the agent tries to walk into a wall.

Unfortunately, however, most agents in 3D games are not point-based, so we need a way to handle objects with extent. The way I describe here is due to van Waveren [17] and uses the notion of configuration spaces I described in [7]. An alternative, similar approach, that works by modifying a normal BSP at runtime, is discussed in §5. Both of the approaches described are based on the same principle – that performing collision detection between an object with extent and the world is equivalent to performing collision detection between a point at the centre of the object and a copy of the world that has been suitably expanded in accordance with the size of the object.

The van Waveren approach is an offline method designed for brush-based 3D environments (that is, environments built up by combining simple, convex polyhedra). Agents are represented by axis-aligned bounding boxes (AABBs); each class of agent may have multiple AABBs for different poses (e.g. standing or crouching). At level compilation time, the brushes of the environment are expanded for each AABB and the faces of the expanded brushes are unioned together to form an expanded world for that AABB. Each of these expanded worlds can be compiled into a BSP tree, allowing us to perform collision detection for an object represented by the corre-

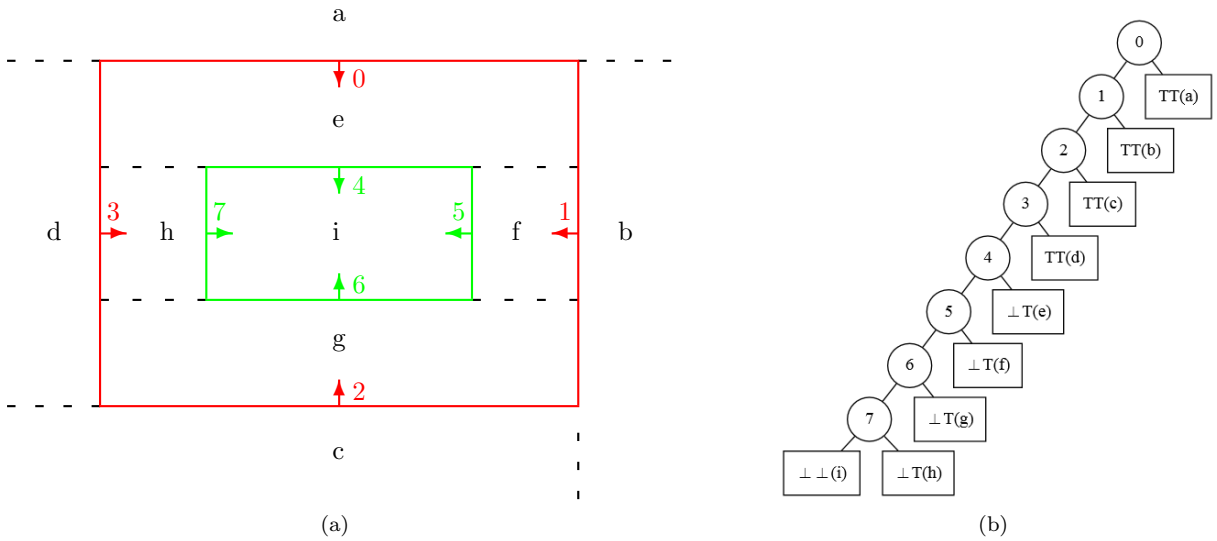


Figure 2: An example 2D world (a) and one possible onion BSP for it (b). The coloured lines (red and green) denote two separate configuration spaces. Their polygons (shown as numbered, oriented line segments) are compiled into the same onion BSP as shown. Individual leaves (labelled with letters) can be empty ($\bot$) in one space and solid ($\top$) in another, e.g. leaf e is empty in the red space but solid in the green one.

sponding AABB. However, maintaining multiple BSP trees is inconvenient because then objects that are in the same physical location but have different sizes cannot be resolved to a leaf in any particular tree – we would much prefer to have a single tree that represents all of the information available.

We can achieve this by constructing a different type of BSP tree that I call an *onion BSP*[1]. Onion BSPs are a generalisation of standard BSPs in which we replace the empty/solid flag in each leaf node with a vector of flags indicating whether the leaf is empty/solid in each configuration space associated with an AABB. Figure 2 shows two configuration spaces generated for an example world (the original, unexpanded world is not shown) and an onion BSP that might be generated for it.

[1]For the interested reader, the name 'onion BSP' comes from the idea that the expanded worlds look rather like the layers of an onion when superimposed in an image. This analogy is not strictly accurate, because the various different AABBs will not in general nest inside each other, but the name is nevertheless both convenient and suggestive.

3.1 The Compilation Process

Onion BSP compilation is in principle much the same as BSP compilation (see §2), but slightly trickier because we have to test the solidity of each leaf in each configuration space rather than getting it for free as part of the compilation process. An explanation of this testing process is deferred to the next section, but it also has an impact on the main part of the compilation. In particular, the test involves checking an arbitrary point in the leaf for solidity in each configuration space (the solidity of any point in the leaf is guaranteed to be the same as that of the entire leaf), so we will need (a) a way of determining an arbitrary point in a leaf, and (b) a way of testing a point for solidity in a configuration space. As will be seen, determining an arbitrary point in a leaf will involve knowing the set of split planes on the path

Listing 1 Building an Onion Tree

```
function build_tree
:  (polys: Vector<Polygon>) → OnionBSPTree

var nodes: Vector<Node>;
var ancestors: Vector<Plane>;
var polyIndices: Vector<PolyIndex>
  := {(i,true) | 0 ≤ i < |polys|};
build_subtree(ref polys, polyIndices,
  ref nodes, ref ancestors);
return make_onion_bsp(nodes);

class PolyIndex
  var index: int;
  var splitCandidate: boolean;
```

from the root of the tree to the leaf, so these should be maintained during compilation.

The resulting main compilation process is shown in Listings 1 and 2. The key thing to note is the way in which a set of split planes is maintained in order to facilitate solidity testing – we add the current split plane to the set before each recursive call to `build_subtree` and remove it again afterwards, so that whenever we reach a leaf it will contain precisely those split planes on the path from the root of the tree to the leaf. Note that orientation is important, so the current split plane must be reversed when recursing into the right-hand subtree.

3.2 Determining Leaf Solidity

A *solidity descriptor* for a leaf in an onion BSP is a vector of flags indicating whether the leaf is empty or solid in each configuration space for which we compiled the BSP. It is common for a leaf to be empty in one configuration space and solid in another – for example, a leaf might be empty in the configuration space corresponding to the crouch pose of an agent, but solid in the configuration space corresponding to the standing pose, indicating that the agent can traverse the leaf whilst crouching but not whilst standing (e.g. think of a low tunnel). To determine a leaf's solidity descriptor, we find an arbitrary point in the leaf and check its solidity in each configuration space in turn; the resulting empty/solid results are combined to form the full solidity descriptor. To test

Listing 2 Building an Onion Subtree

```
function build_subtree
:  (polys: ref Vector<Polygon>;
    polyIndices: Vector<PolyIndex>;
    nodes: ref Vector<Node>;
    ancestors: ref Vector<Plane>) → Node

var splitter: Plane
  := choose_splitter(polyIndices);

// If there were no suitable split candidates,
// this is a leaf.
if splitter = null then
  var solidity: DynamicBitset
    := determine_leaf_solidity(ancestors);
  var indicesOnly: Vector<int>
    := {i | (i,_) ∈ polyIndices};
  nodes.push_back(
    Leaf(|nodes|, solidity, indicesOnly)
  );
  return nodes.back();

var backPolys, frontPolys: Vector<PolyIndex>;
for each pi@(index, splitCandidate) ∈ polyIndices
  var poly: Poly := polys[index];
  switch classify_against_plane(poly, splitter)
    case CP_BACK:
      backPolys.push_back(pi);
      break;
    case CP_COPLANAR:
      if splitter.norm().dot(poly.norm()) > 0 then
        frontPolys.push_back((index, false));
      else
        backPolys.push_back((index, false));
      break;
    case CP_FRONT:
      frontPolys.push_back(pi);
      break;
    case CP_STRADDLE:
      (back, front) := split_poly(poly, splitter);
      polys[index] := back;
      polys.push_back(front);
      backPolys.push_back(pi);
      frontPolys.push_back(
        (|polys| - 1, splitCandidate)
      );
      break;

ancestors.push_back(splitter);
var left: Node
  := build_subtree(frontPolys, nodes, ancestors);
ancestors.pop_back();

ancestors.push_back(splitter.flipped());
var right: Node
  := build_subtree(backPolys, nodes, ancestors);
ancestors.pop_back();

var subRoot: Node
  := Branch(|nodes|, splitter, left, right);
nodes.push_back(subRoot);
return subRoot;
```

Listing 3 Determining Leaf Solidity

```
function determine_leaf_solidity
:  (ancestors: Vector<Plane>) → DynamicBitset

// Assumed available from elsewhere:
// * mapTrees: Vector<BSPTree>

// Find an arbitrary point within the leaf with the
// specified ancestor planes.
var p: Vec3 := arbitrary_leaf_point(ancestors);

// Classify the point against each map tree to
// determine the solidity descriptor for the leaf.
var solidity: DynamicBitset(|mapTrees|);
for each mt_i ∈ mapTrees
  var leaf: BSPLeaf := mt_i.find_leaf(p);
  solidity[i] := leaf.is_solid();

return solidity;
```

points' solidity in a configuration space, we build a normal BSP tree (called a *map tree* in the code) for the space at the start of the compilation process and later classify any relevant points with regard to it. The top-level process to determine leaf solidity is shown in Listing 3.

3.2.1 Finding an Arbitrary Leaf Point

To find an arbitrary point in a leaf, recall that each leaf represents a convex subspace of the world. Our first intuition might be to create an explicit representation of the leaf as a convex polyhedron and then compute the average of the midpoints of the polyhedron's faces as our point. This does in fact work perfectly for fully-bounded leaves (see Figure 3(a)), but unfortunately fails for unbounded ones (see Figure 3(b)). Fortunately, in practice, there is an easy solution to this problem: we can simply stipulate that the world we are representing is bounded by an inward-facing box, thereby ensuring that every leaf is bounded (see Figure 3(c)). This is clearly a reasonable assumption in the context of representing a 3D world, since it would not be meaningful for such a world to be infinite. (The interested reader may wish to take a look at [14], where a similar approach is taken to deal with unboundedness in a related linear programming problem.)

The algorithm itself is shown in Listing 4. It is

Listing 4 Finding an Arbitrary Leaf Point

```
function arbitrary_leaf_point
:  (ancestors: Vector<Plane>) → Vec3

// Step 1: Make an inward-facing convex polyhedron
// around the leaf.

// Make an array of possible bounding planes: these
// are the ancestor planes themselves, plus the
// planes that bound the 3D world. The planes are
// specified as ax + by + cz - d = 0.
const HALFWORLDBOUND: double := 100000;
var planes: Vector<Plane> := ancestors;
planes.push_back(Plane((1,0,0), -HALFWORLDBOUND));
planes.push_back(Plane((-1,0,0), -HALFWORLDBOUND));
planes.push_back(Plane((0,1,0), -HALFWORLDBOUND));
planes.push_back(Plane((0,-1,0), -HALFWORLDBOUND));
planes.push_back(Plane((0,0,1), -HALFWORLDBOUND));
planes.push_back(Plane((0,0,-1), -HALFWORLDBOUND));

var faces: Vector<Poly>;
for each p_i : Plane ∈ planes
  // Build a large initial face on each plane.
  var face: Poly := make_large_poly(p_i);

  // Clip it to the other planes.
  var discard: bool := false;
  for each p_j : Plane ∈ planes
    if j = i then continue;
    switch classify_against_plane(face, p_j)
      case CP_BACK:
        // Face entirely out of leaf.
        discard = true;
        break;
      case CP_COPLANAR:
        // Shouldn't happen: ancestors are unique.
        throw "Unexpected duplicate plane";
      case CP_FRONT:
        // Face entirely in leaf.
        continue;
      case CP_STRADDLE:
        // Part of face in leaf, part not.
        (_,front) := split_poly(face, p_j);
        face := front;
        break;

    if discard then break; // early out

  // Add surviving faces to the array.
  if not discard then faces.push_back(face);

// Step 2:  Compute the average of the polyhedron
// face midpoints.
var denom: int := 0;
var p: Vec3(0,0,0);
for each face: Poly ∈ faces
  for each v ∈ face.vertices()
    p := p + v;
    denom := denom + 1;

assert denom ≠ 0;
p := p / denom;
return p;
```

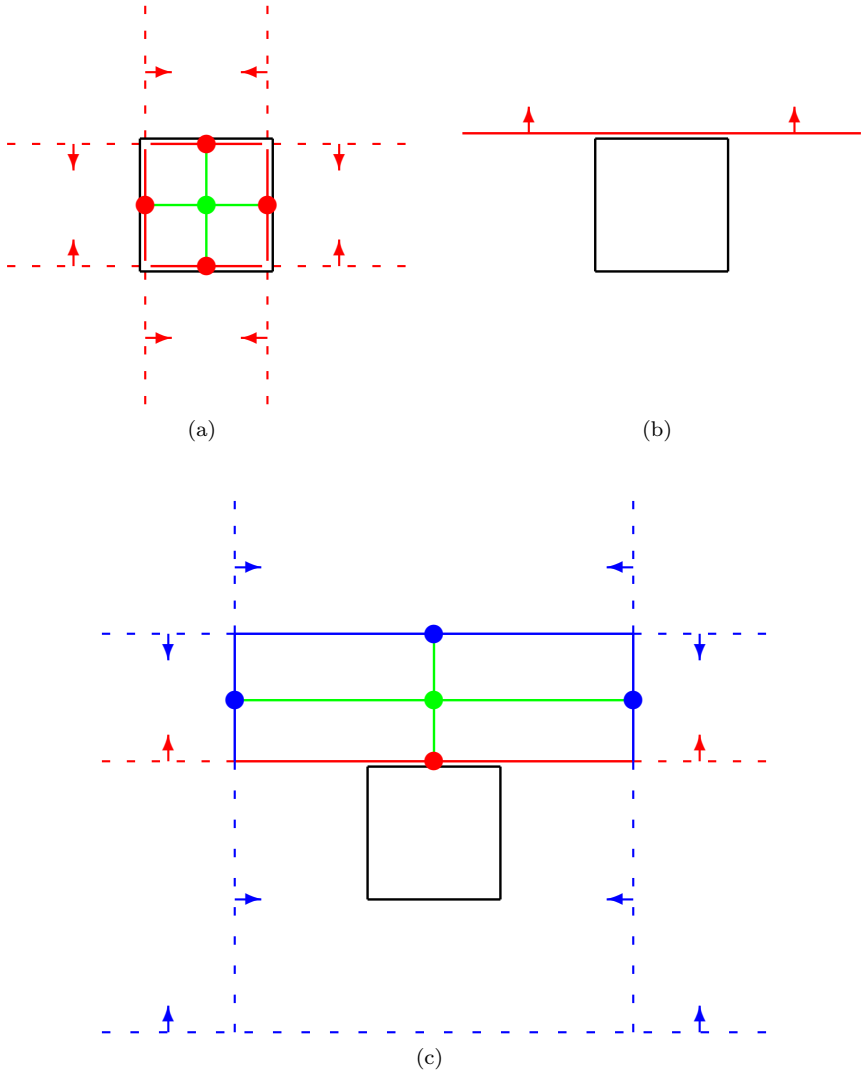


Figure 3: Finding an arbitrary point in a leaf of a simple 2D world with a single room (shown in black). In (a), we can successfully build a convex polyhedron for the bounded leaf representing the room itself and then average the midpoints of the polyhedron's faces (shown in red) to find a suitable point (shown in green). In (b), the same procedure fails for the unbounded leaf behind the room's topmost wall. In (c), we rectify the problem by adding bounding planes (shown in blue) around the world as a whole. This ensures that all of the leaves are bounded, allowing the method to work.

called with the set of ancestor planes leading down to the given leaf in the tree (recall that these are maintained as part of the top-level compilation process). These are augmented with the planes of the inward-facing box that we are assuming bounds the world. We then construct an extremely large polygon on each of the planes in turn, and clip it to the other planes (see [5] for the implementation details). The set of polygons that results forms a convex polyhedron representing the (bounded) leaf. As previously stated, we finally compute the midpoint of each face of the polyhedron and average them to produce an arbitrary point that is guaranteed to be inside the leaf.

4 Collision Detection

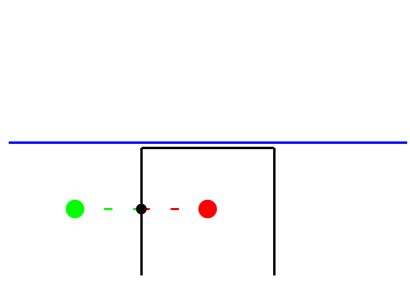
Recall that our goal is to detect collisions between moving objects of various sizes and a stationary world. The desired output of our collision detection approach is a set of collisions (or contacts), each of which is specified by a *collision point* (a first point at which the moving object touches the world), a *collision normal* (the normal of the surface that is hit by the moving object) and a *collision time* (a number in the range $[0, 1]$ indicating at what point during the movement the collision occurs).

Having constructed an onion BSP for the world, it is now possible to perform collision detection against it for objects with a specific AABB, using a variant of the 'find first transition' algorithm mentioned in §3 (see Listing 5). The key difference from the version for normal BSPs is that the leaf solidity test at the top of the `fft_sub` function is performed for a specific configuration space (e.g. one corresponding to an agent's crouching pose); in all other respects the two are essentially the same.

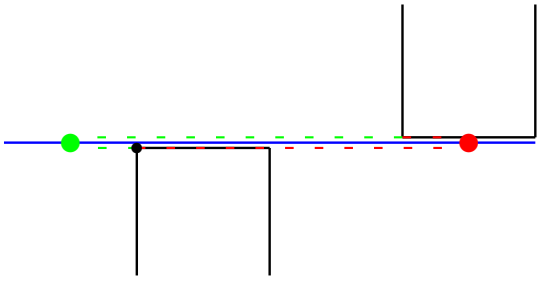
The algorithm is initially called on the movement ray (which is just a line segment) of an agent for the current frame and the root node of the onion BSP, and proceeds recursively, ultimately producing a 'transition' to indicate its result (transitions are either (a) `RAY_E`, indicating that the entire movement ray is in empty space, (b) `RAY_S`, indicating that the entire movement ray is in solid space, or (c) a triple

Listing 5 Find First Transition (Onion BSP Version)

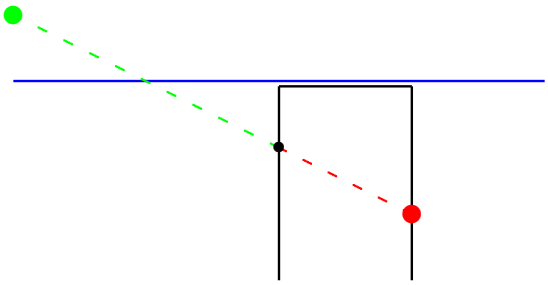
```
function fft_sub: (src: Vec3; dest: Vec3;
  node: Node) → Transition

// Assumed available throughout:
// * cSpace: int [the configuration space index]

var leaf: Leaf := node.as_leaf();
if leaf ≠ null then
  return leaf.is_solid(cSpace) ? RAY_S : RAY_E;

var br: Branch := node.as_branch();
var left, right: Node := br.left(), br.right();
var splitter: Plane := br.splitter();
var cpSrc, cpDest: PlaneClassifier;
switch classify_against_plane
(src, dest, splitter, ref cpSrc, ref cpDest)
  case CP_BACK:
    return fft_sub(src, dest, right);
  case CP_COPLANAR:
    var trLeft: Transition :=
      fft_sub(src, dest, left);
    var trRight: Transition :=
      fft_sub(src, dest, right);
    if trLeft.class = trRight.class then
      switch trLeft.class
        case RAY_E|RAY_S:
          return trLeft;
        default:
          var dLeft: double := |src − trLeft.loc|^2;
          var dRight: double := |src − trRight.loc|^2;
          return dLeft < dRight ? trLeft : trRight;
    else if trLeft.class = RAY_E2S|RAY_S2E then
      return trLeft;
    else if trRight.class = RAY_E2S|RAY_S2E then
      return trRight;
    else return RAY_E;
  case CP_FRONT:
    return fft_sub(src, dest, left);
  default: // case CP_STRADDLE
    var mid: Vec3 :=
      intersect(src, dest, splitter);
    (near,far) := cpSrc = CP_FRONT ?
      (left,right) : (right,left);
    var trNear: Transition :=
      fft_sub(src, mid, near);
    if trNear.loc ≠ null then return trNear;
    var trFar: Transition :=
      fft_sub(mid, dest, far);
    switch trFar.class
      case RAY_E:
        return trNear.class = RAY_E ? RAY_E :
          Transition(RAY_S2E, mid, splitter);
      case RAY_S:
        return trNear.class = RAY_S ? RAY_S :
          Transition(RAY_E2S, mid, splitter);
      case RAY_E2S:
        return trNear.class = RAY_E ? trFar :
          Transition(RAY_S2E, mid, splitter);
      default: // case RAY_S2E
        return trNear.class = RAY_S ? trFar :
          Transition(RAY_E2S, mid, splitter);
```



(a) The movement ray is entirely on one side of the plane (shown in blue): recurse down that side.



(b) The movement ray lies in the plane itself: recurse down both sides and combine the results. In this case, both sides have a transition, so we take the nearer one.



(c) The movement ray straddles the plane: split it and recurse down the near side first. In this case, the near half is entirely in empty space, so we then recurse down the far side to find the transition point.

Figure 4: A few of the recursive cases for the 'find first transition' algorithm, illustrated on a movement ray from an agent's current position in empty space (shown in green) to its attempted position in solid space (shown in red).

(`RAY_E2S` or `RAY_S2E`, `point`, `splitter`), indicating that the movement ray first transitions from empty to solid, or solid to empty, space at the specified point on the specified split plane). At each recursive step, the relevant segment of the movement ray (initially, all of it) is classified against the split plane of the current node, and appropriate action is taken based on the result. If the movement ray segment is entirely on one side of the plane, we recurse down that side of the tree. If the movement ray segment is on the plane (the coplanar case), we pass it down both sides of the tree and subsequently combine the results. If the movement ray segment straddles the plane, we split it and pass the half of it near the start of the segment down the corresponding side of the tree. If this yields a non-trivial transition, we return it; otherwise, we pass the other half down the far side of the tree, and subsequently derive the result as shown in Listing 5. When we eventually reach a leaf, we return a transition based on the solidity of the leaf in the configuration space in which we are interested (this can be specified as an additional parameter to the algorithm, or provided by some other means). A few of the recursive cases are illustrated in Figure 4.

As mentioned, the ultimate result of the 'find first transition' algorithm is either a trivial transition (the

movement ray is entirely in empty or solid space) or a non-trivial one; in the latter case, the point and the normal of the split plane found can be used directly as the *collision point* and *collision normal* for a detected collision. The *collision time* can be calculated using simple ratios as follows. Denote the source and destination endpoints of the movement ray as $\mathbf{s}$ and $\mathbf{d}$ respectively, and the collision point as $\mathbf{c}$. Then the collision time is given by:

$$t = \sqrt{\frac{|\mathbf{c} - \mathbf{s}|^2}{|\mathbf{d} - \mathbf{s}|^2}}$$

The case of a trivial transition that lies entirely in solid space needs special handling to ensure robustness. In practical terms, this can very occasionally happen due to rounding errors when the source endpoint of the movement ray is on an empty/solid boundary. A simple way of dealing with the issue is to repeat the find first transition call with a source endpoint that is moved back from the boundary by a small amount (i.e. $\mathbf{s}' = \mathbf{s} - \epsilon(\mathbf{d} - \mathbf{s})$).

The collisions we generate are fed into the physics system for later resolution. I will explain how this works in a future article.

5 Discussion

The approach that I have described thus far is a (comparatively) simple and effective way of detecting collisions between moving objects and a stationary world, but unsurprisingly it does have some limitations. One potential drawback is that it is designed to work for a small number of object sizes that are known at level compilation time: this was not a problem for games such as *Quake III*, but it makes the technique less suitable for games that want to contain a wide variety of differently-shaped characters, since compiling large numbers of different configuration spaces into an onion BSP would severely bloat the tree and lead to slow level compilation times and poor performance. Another drawback is that it only works for objects that do not rotate: games that want to support more realistic physical simulation have to use more complicated approaches (e.g. see [3, 13]).

A related approach that eliminates the first of these limitations, whilst still providing acceptable performance, was presented by Melax in [12]. The details can be found in that article, but the essence of the approach is to replace the 'find first transition' algorithm with a variant that dynamically moves the planes of a normal BSP for the world during ray testing so as to simulate the configuration spaces for different types of object. This avoids the need to know the sizes of the objects up-front, at the cost of making ray testing somewhat more costly. The approach was used successfully in the BioWare game *MDK2*.

6 Conclusions

In this article, I have described a simple and effective technique (due to van Waveren) for detecting collisions between moving objects and their surrounding 3D environment. While there are important limitations to this technique in the context of realistic physical simulation (most notably the fact that it only works for non-rotating objects), it has proved useful in a games context because it can detect collisions for translating objects quickly and accurately. The alternative approach (due to Melax) mentioned in §5 builds upon this technique by allowing large numbers of differently-shaped characters to be handled without bloating the tree.

It remains to be shown how to detect inter-object collisions and how to build a working physics system, which I hope to address in future articles. We can then return to our original problem of agent movement, using the physics system and the environment's navigation mesh in tandem.

7 Acknowledgements

I would particularly like to thank the editorial team for the effort that has gone into typesetting this article for publication. Many thanks also to the rest of the Overload team for reviewing this article and suggesting ways in which to improve it.

References

[1] Michael Abrash. *Michael Abrash's Graphics Programming Black Book*. Coriolis Group Books, special edition, 1997.

[2] Jim Arvo. Linear-Time Voxel Walking for Octrees. *Ray Tracing News*, 1(12), March 1988.

[3] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[4] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On Visible Surface Generation by A Priori Tree Structures. *Computer Graphics*, 14(3):124–133, 1980.

[5] Stuart Golodetz. The *hesperus* 3D game engine. Source code available online at: `https://github.com/sgolodetz/hesperus2`.

[6] Stuart Golodetz. Divide and Conquer: Partition Trees and Their Uses. *Overload*, 86:24–28, August 2008.

[7] Stuart Golodetz. Automatic Navigation Mesh Generation in Configuration Space. *Overload*, 117:22–27, October 2013.

[8] Stuart M Golodetz. A 3D Map Editor. Undergraduate thesis, Oxford University Computing Laboratory, May 2006.

[9] Dan Gordon and Shuhong Chen. Front-to-Back Display of BSP Trees. *IEEE Computer Graphics and Applications*, 11(5):79–85, 1991.

[10] Frederik W Jansen. Data structures for ray tracing. In Laurens R A Kessener, Frans J Peters, and Marloes L P van Lierop, editors, *Data Structures for Raster Graphics*, pages 57–73. Springer-Verlag Berlin Heidelberg, 1986.

[11] Mikola Lysenko, Roshan D'Souza, and Ching-Kuan Shene. Improved Binary Space Partition Merging. *Computer-Aided Design*, 40(12):1113–1120, 2008.

[12] Stan Melax. Dynamic Plane Shifting BSP Traversal. *Graphics Interface*, 2000:213–220, 2000.

[13] Ian Millington. *Game Physics Engine Development*. Morgan Kaufmann, 2007.

[14] Raimund Seidel. Small-Dimensional Linear Programming and Convex Hulls Made Easy. *Discrete & Computational Geometry*, 6(1):423–434, 1991.

[15] Gary Snethen. XenoCollide: Complex Collision Made Simple. In Scott Jacobs, editor, *Game Programming Gems 7*, pages 165–178. Charles River Media, 2008.

[16] Kelvin Sung and Peter Shirley. Ray Tracing with the BSP Tree. In David Kirk, editor, *Graphics Gems III*, pages 271–274. Morgan Kaufmann, 1992.

[17] Jean Paul van Waveren. The Quake III Arena Bot. Master's thesis, Delft University of Technology, 2001.