COMP37111: Advanced Computer Graphics

Spatial Enumeration and Culling

Steve Pettifer

December 2020

Department of Computer Science
The University of Manchester

1 Spatial Enumeration

The final sections of this course unit deal with techniques for improving rendering performance so that things can be drawn in real-time. These techniques are a fundamental part of the way in which modern computer games and other interactive 'virtual environments' work.

We'll look at a series of techniques that all fall into the category of 'spatial enumeration' (also called 'spatial indexing'). They all address the same problem of forming a an efficient link between the data structures used by a particular application (say, a world full of monsters, if its a computer game) and the data structures used to draw the 3D scenes (polygons, meshes, objects, textures and so forth). The basic issue is that the kind of data structures that are sensible for representing the application data are almost certainly not the same ones that make for efficient rendering. You can think of spatial enumeration as playing a similar role to 'indexing' a database: in a database you want to represent your data in a way that is semantically correct, to avoid redundancy and inconsistency – but this isn't necessary the optimal way of storing it for fast querying. So you build indexes that map between the shape of data expected back from common queries, and the more 'truth and beauty' version of the data held in the normalised schema. The same kind of effect happens in computer graphics: there are a set of common 'queries', such as 'which object intersects first with this ray', or 'give me all the objects that are visible from this eyepoint, sorted in order of distance from the eyepoint', and we need these to be fast in order to create interactive 3D scenes. The same spatial enumeration structures that we'll be looking at for these interactive purposes are also useful in speeding up the performance of 'offline' rendering algorithms such as Ray Tracing, Radiosity and Volume Rendering.

In fact, the first of these techniques is one that we've already explored; we just didn't give it a name at the time.

1.1 The Gridcell

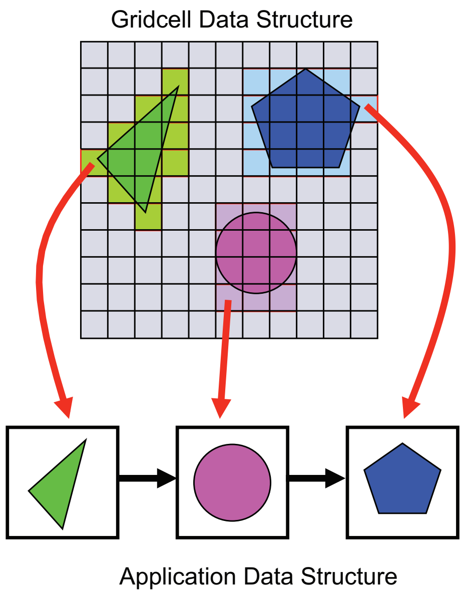


Figure 1: A 2D version of the 'gridcell' structure, showing pointers from the objects in 'world space' back down into the underlying objects in the application itself. Of course, in reality this structure is a regular 3D array rather than a 2D one, so the cells are cubes – but the principle is identical.

The idea of a gridcell structure is simple: you subdivide your 3D space into a large number of small cubes. Then you place your 3D objects into this space. Every cell that is intersected by a part of the objects gets a pointer to the underlying data structure that represents the object, so that if you choose a point in 3D space, you can easily determine which cell contains it, and then follow any of the references from that cell to find out which objects you may have intersected with.

A simplified 2D version of this is shown in Figure 1.

Creating an empty gridcell structure is trivial; it's just a 3D array where each cell contains a list of pointers to objects. Populating it is a bit more tricky, since this means that as you position an object in 3D space, you have to work out which of the gridcell's cells the object touches, and make sure that each of them gets a pointer back to that object. The decision as to whether you only do this for cells that intersect with the surface of the object, or for all cells including those 'inside' the object will depend on the application. So the cost of populating the gridcell is fairly high, as is the spatial complexity (which is $O(n^3)$, where n is the number of cells in each dimension). But what is the cost of 'querying' the gridcell? Let's say we want to work out which objects are intersected if we fire a ray into a particular cell (x, y, z); answering this is extremely cheap! We can index into our 3D array to immediately find the appropriate cell, and then follow any pointers in that cell to tell us which objects we've 'hit'. So 'querying' the gridcell is extremely cheap – $O(1)$ in fact. With the gridcell we have excellent query time; but very poor use of memory (if we make an accurate, fine grained gridcell, it'll cost lots; if we make a course grained one with few cells, then we'll get a lot of false positives when we query). You'll probably have spotted by now that the 'voxel structure' we talked about in the Volume Rendering section is basically a gridcell, where the underlying objects are values from our original sample data.

1.2 The Octree

The Octree is a relatively simple evolution of the gridcell idea, which exploits what's known as 'spatial coherence'; the property that in most scenes, 'stuff tend to cluster together'. Look at the world around you; it typically conists of big empty spaces, and then bunches of things near one another. This works at multiple levels: a galaxy is mostly sparse, with things clustered into solar systems and lots of space between; a city has lots of empty space above street level, with things clustered into buildings; chairs cluster around tables; things on the desk around you end up in piles. And so on. The gridcell approach doesn't take this into account, so you typically end up with large proportions of the cells being 'empty': this isn't so much the case with Volume Rendering, because you're probably focussing your visualisation on a blob of data that already contains 'interesting stuff'; but for a scene in a computer game (which usually looks a bit like the real world), you can see that there are likely to be fairly big empty spaces. The idea of the Octree is to start off with one large cuboid space that encompasses the whole scene, and then to break that down into 8 smaller spaces, which will contain smaller subsets of the scene. . . and to keep on doing that until each cell (called an 'octant') contains fewer than a predetermined number of objects. This is an 'adaptive' technique, meaning that if one of the top level cells happens to fall in a fairly empty part of the scene, you don't subdivide it any further, and instead focus the used of CPU and memory on bits of the world that need it. The 2D analogue of this approach is called a 'quadtree', and is shown in Figure 2.

1.3 Hierarchical Bounding Volumes

The problem that we're faced with over and over in graphics is that of dealing with lots of polygons. Although, for example, interesecting a ray with a polygon is relatively painless we went through this in a fair amount of detail), most scenes consist of a very large number of polygons,

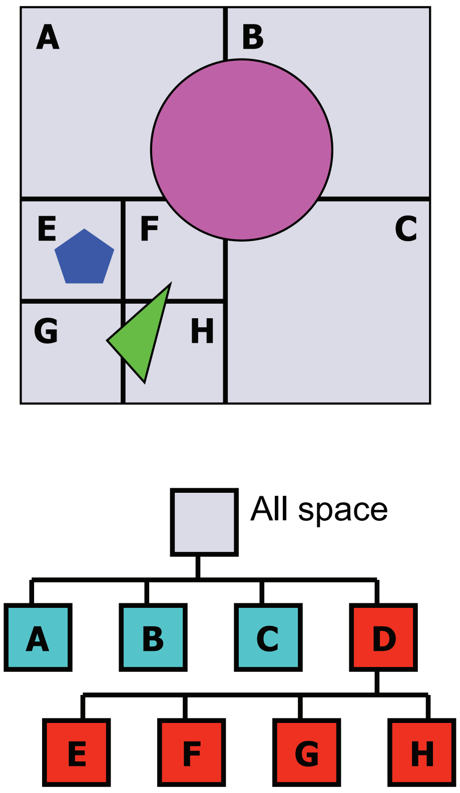


Figure 2: A 'quadtree' is the two dimensional version of a 3D octree. In this figure, the overall space is broken into four square quadrants ('octants' in 3D, which are cubes), which are then sub-divided further until only one polygon exists in each.

so repeating all the calcualtions becomes quite painful. On the other hand, the ray/sphere intersection calculation is very cheap—but not very many things, apart from spheres, are spherical. The idea behind bounding volumes is to use 'cheap' shapes (like spheres) to approximate more computationally expensive ones (such as polygonal meshes), in an attempt to eliminate some parts of the scene from our enquiries early on. Imagine you have a polygonal object—say, for the sake of argument—a cow, and we put a notional sphere around the cow (we don't draw the sphere, it's just there in the algorithm). Now if we try to do a ray/sphere intersection and *miss the sphere*, we can immediately be sure that we don't need to bother wasting time testing the individual cow polygons to see if any of those have been hit: we know they haven't because the ray didn't hit the surrounding sphere. If we *do* hit the sphere, though, we then need to test the individual cow polygons for intersections. If we hit a cow polygon, then we've 'wasted' one extra calculation (the intersection with the sphere), but that's no big deal since we've got a 'hit' result. If don't hit a cow polygon, then we've wasted much more time, which is a bad thing. On average though, if we're casting loads of rays into the scene, and if we assume that our scene is reasonably sparse so more rays miss the cow-in-a-sphere than hit it, then we gain an enormous win, since most of our rays can tell they haven't possibly hit the cow by just testing against the very simple sphere bounding volume.

If we want to optimise things a bit more though, we should look at the relationship between the shape of our arbitrary object, and a sphere. We can see there's a lot of 'wasted space' between the two shapes since they are not a good fit for one another. The more wasted space, the more 'false positives' we'll get which lead us to test against the cow's polygons, so ideally we'd like a shape that minimises this.

We could try a bounding cube; the cost of intersecting with a cube is the same as that of intersecting with 6 polygons (i.e. the quads forming the faces of the cube), which compared to the cow is going to be cheap, but compared with the sphere a bit more expensive. The problem here is that a cube is generally not a much better fit to most objects than a sphere, so a balance between 'cost of intersection' and 'false positives caused by wasted space' needs to be drawn again. In the extreme, of course, the best fitting bounding volume for a cow would be, well, a bounding cow. But the cost of testing against this fairly specialist bounding volume is the same as the underlying cow, which would just be silly.

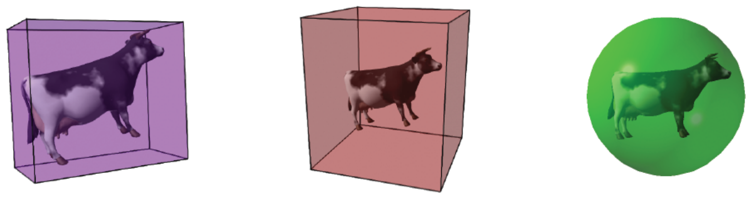


Figure 3: Three different bounding volumes: a cuboid, a cube and a sphere.

It turns out that a reasonable compromise in many cases is to used a bounding cuboid, i.e. rather than a regular cube, we just allow the pairs of opposite faces to be different sizes. Here the cost is almost identical to that of a regular cube, but because we can vary the size in three dimensions, we can often get a better fit to the objects we're enclosing. Figure 3 shows the cow enclosed in the three bounding shapes we've discussed.

The obvious next step is to allow bounding volumes to contain other bounding volumes; again exploiting the spatial coherene of most scenes. The process is conceptually simple; enclose complex polygonal objects in bounding shapes (let's say cuboids), then enclose any bounding shapes that end up being near one another in a bigger bounding shape, and repeat the process until you have a suitable hierarchy of objects. You can think of this as being a bit like the octree process in reverse, but this time you end up with the nodes in the tree being spatially scattered around the scene wherever clusters of objects appear, rather than being arranged regularly. Hierarchical Bounding Volumes (HBV) is an example of an 'irregular' spatial enumeration technique; by contrast, the gridcell and octree techniques are considered 'regular' because they split space up into equally sized regions. A partitioning of an environment using bounding 'spheres' (circles in this 2D illustration) is shown in Figure 4.

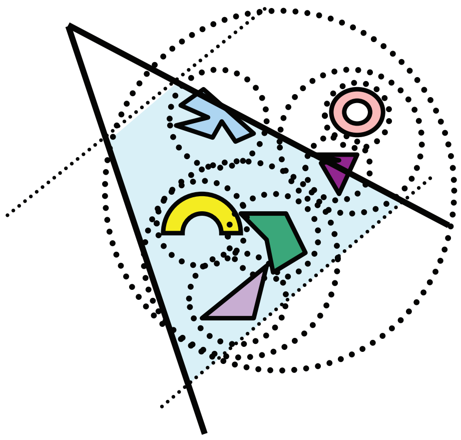


Figure 4: A hierarchy of bounding circles (these would be spheres in 3D) surrounding clusters of objects in a scene. Note that this is an irregular partitioning of space; although all the bounding objects are circles/spheres, they are positioned based on the location of objects, and not according to a pre-determined regular pattern.

1.4 Binary Space Partitioning

If we think a little more carefully about the queries we are really asking of our spatial enumeration techniques, we realise that they often boil down to one specific kind of question: "what is the closest object that intersects with this ray/vector/line". This is true whether you are ray tracing, or firing a laser from your virtual gun at a virtual bad guy; or even if you are just selecting an object with a mouse. The regular spatial enumeration techniques (gridcell and octree) can answer this question fairly easily, because they are tightly coupled to the co-ordinates of the 3D environment and so preserve the concept of in-front/behind, above/below, right-of/left-of fairly well; but the more anarchic hiearachical bounding volume mechanism, while being more efficient in its use of space, loses this relationship; it simply represents 'collections of things near each other'.

Binary Space Partitioning is an approach that attempts to retain the 'relative positioning' property, whilst also being efficient in terms of space and compute complexity. There are two main variants, one of which is easy to understand but not hugely efficient, and another which is a bit more tricky to make sense of on first contact, but which has better performance in most cases. We'll look at the former first, as a way of leading up to the latter.

Axis Aligned Binary Space Partitioning attempts to split the world recursively into two sections, where the boundary is drawn such that it aligns with one of the world axes. The process is shown in Figure 5. It starts by dividing the space vertically into two sections, shown by line 0. The next step then divides both these spaces horizontally in two (shown by lines 1a and 1b). These in turn are divided vertically, then horizontally and so on, until some termination condition is reached (say, that no more than 2 objects are left in any one region). The advantage of this approach is that it preserves the relative position of the objects in the different regions, i.e. to the left and right of vertical divisions, and in-front/behind the horizontal ones. This means that by traversing the tree that's created, and knowing what 'decision' was made at each sub-division, you can determine the relative position of any object with respect to any viewpoint (which means you can work out an ordering of things, say to select the 'nearest' intersection straight off, rather than having to get all possible intersecting objects and then sorted them into order along the ray).

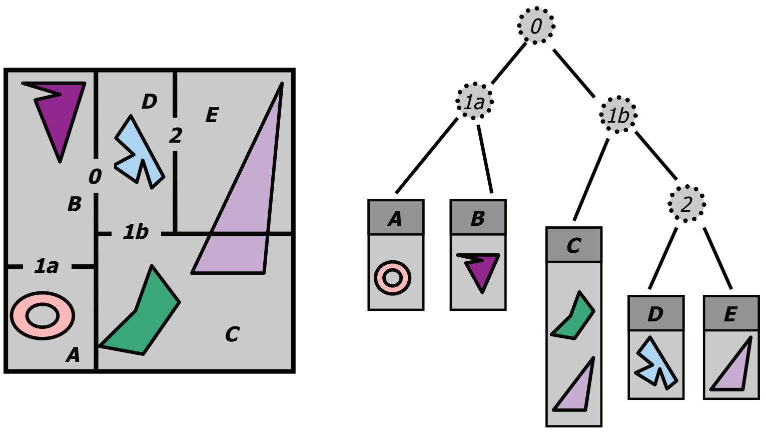


Figure 5: An example of an axis-aligned BSP containing 5 objects

A relatively minor tweak that makes BSP more efficient is to allow the partitioning to take place along arbitrary axes, rather than being constrained to world axes, and in Polygon Aligned Binary Space Partitioning, the planes are frequently (but not always) chosen to coincide with the planes defined by polygons in the scene.

1.5 Generating BSP Trees

(The following text is taken from the Wikipedia entry on ***Binary Space Partitioning***^{w}*, and is modified only slightly. The diagrams are by* ***Chrisjohnson***^{w}*)*

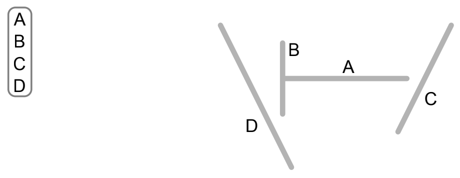
The canonical use of a BSP tree is for rendering polygons (that are double-sided, that is, without back-face culling) with the **Painter's algorithm**^{w}. Such a tree is constructed from an unsorted list of all the polygons in a scene. The recursive algorithm for construction of a BSP tree from that list of polygons is:

1. Choose a polygon P from the list.
2. Make a node N in the BSP tree, and add P to the list of polygons at that node.
3. For each other polygon in the list:
   (a) If that polygon is wholly in front of the plane containing P, move that polygon to the list of nodes in front of P.
   (b) If that polygon is wholly behind the plane containing P, move that polygon to the list of nodes behind P.
   (c) If that polygon is intersected by the plane containing P, split it into two polygons and move them to the respective lists of polygons behind and in front of P.
   (d) If that polygon lies in the plane containing P, add it to the list of polygons at node N.
4. Apply this algorithm to the list of polygons in front of P.

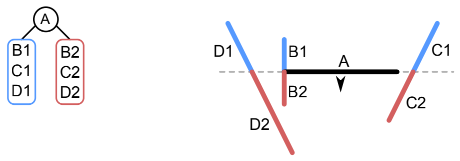
5. Apply this algorithm to the list of polygons behind P.

The following sequence illustrates the use of this algorithm in converting a list of lines or polygons into a BSP tree. At each of the eight steps, the algorithm above is applied to a list of lines, and one new node is added to the tree.

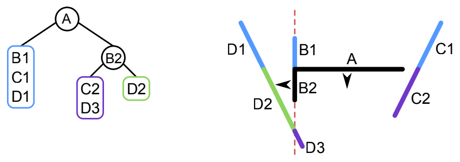
0. Start with a list of lines, (or in 3D, polygons) making up the scene. In the tree diagrams, lists are denoted by rounded rectangles and nodes in the BSP tree by circles. In the spatial diagram of the lines, direction chosen to be the 'front' of a line is denoted by an arrow.



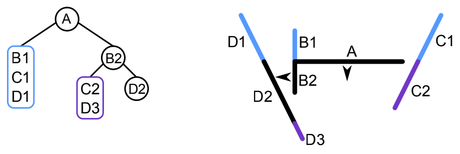
1. Following the steps of the algorithm above i) Choose a line, A, from the list. ii) add it to a node. iii) Split the remaining lines in the list into those which lie in front of A (i.e. B2, C2, D2), and those which lie behind (B1, C1, D1). iv) Process first the lines lying in front of A (in steps iiv), v) followed by those behind it (in steps 47).



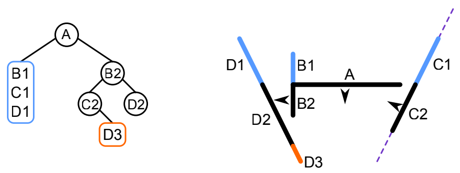
2. Apply the algorithm to the list of lines in front of A (containing B2, C2, D2). We choose a line, B2, add it to a node and split the rest of the list into those lines that are in front of B2 (D2), and those that are behind it (C2, D3).



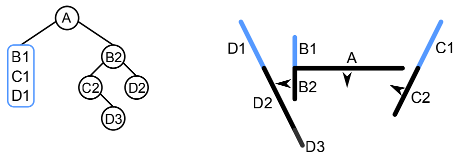
3. Choose a line, D2, from the list of lines in front of B2. It is the only line in the list, so after adding it to a node, nothing further needs to be done.



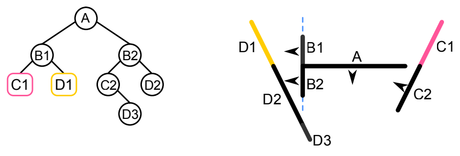
4. We are done with the lines in front of B2, so consider the lines behind B2 (C2 and D3). Choose one of these (C2), add it to a node, and put the other line in the list (D3) into the list of lines in front of C2.



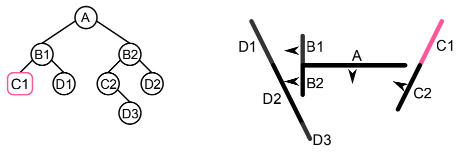
5. Now look at the list of lines in front of C2. There is only one line (D3), so add this to a node and continue.



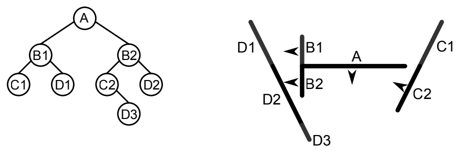
6. We have now added all of the lines in front of A to the BSP tree, so we now start on the list of lines behind A. Choosing a line (B1) from this list, we add B1 to a node and split the remainder of the list into lines in front of B1 (i.e. D1), and lines behind B1 (i.e. C1).



7. Processing first the list of lines in front of B1, D1 is the only line in this list, so add this to a node and continue.



8. Looking next at the list of lines behind B1, the only line in this list is C1, so add this to a node, and the BSP tree is complete.



The final number of polygons or lines in a tree will often be larger (sometimes much larger) than that in the original list, since lines or polygons that cross the partitioning plane must be split into two. It is desirable that this increase is minimised, but also that the final tree remains reasonably balanced. The choice of which polygon or line is used as a partitioning plane (in step 1 of the algorithm) is therefore important in creating an efficient BSP tree.

1.6 Traversal

A BSP tree is traversed in a linear time, in an order determined by the particular function of the tree. Again using the example of rendering double-sided polygons using the painter's algorithm, for a polygon P to be drawn correctly, all the polygons which are behind the plane in which P lies must be drawn first, then polygon P must be drawn, then finally the polygons in front of P must be drawn. If this drawing order is satisfied for all polygons in a scene, then the entire scene is rendered in the correct order. This procedure can be implemented by recursively traversing a BSP tree using the following algorithm. From a given viewing location V, to render a BSP tree:

1. If the current node is a leaf node, render the polygons at the current node.
2. Otherwise, if the viewing location V is in front of the current node:
   (a) Render the child BSP tree containing polygons behind the current node
   (b) Render the polygons at the current node
   (c) Render the child BSP tree containing polygons in front of the current node
3. Otherwise, if the viewing location V is behind the current node:
   (a) Render the child BSP tree containing polygons in front of the current node
   (b) Render the polygons at the current node
   (c) Render the child BSP tree containing polygons behind the current node
4. Otherwise, the viewing location V must be exactly on the plane associated with the current node. Then:
   (a) Render the child BSP tree containing polygons in front of the current node
   (b) Render the child BSP tree containing polygons behind the current node

Applying this algorithm recursively to the BSP tree generated above results in the following steps:

- The algorithm is first applied to the root node of the tree, node A. V is in front of node A, so we apply the algorithm first to the child BSP tree containing polygons behind A

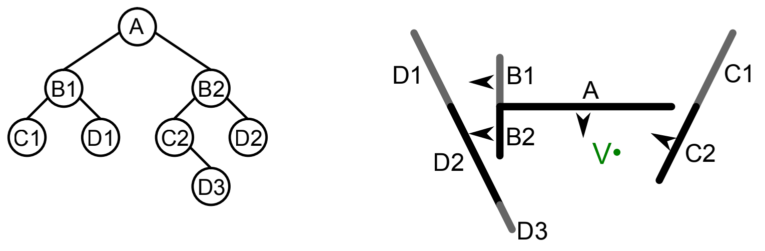


Figure 6: An example BSP tree constructed for a simple scene containing four polygons

- This tree has root node B1. V is behind B1 so first we apply the algorithm to the child BSP tree containing polygons in front of B1:
  * This tree is just the leaf node D1, so the polygon D1 is rendered.
- We then render the polygon B1.
- We then apply the algorithm to the child BSP tree containing polygons behind B1:
  * This tree is just the leaf node C1, so the polygon C1 is rendered.
- We then draw the polygons of A
- We then apply the algorithm to the child BSP tree containing polygons in front of A
- This tree has root node B2. V is behind B2 so first we apply the algorithm to the child BSP tree containing polygons in front of B2:
  * This tree is just the leaf node D2, so the polygon D2 is rendered.
- We then render the polygon B2.
- We then apply the algorithm to the child BSP tree containing polygons behind B2:
  * This tree has root node C2. V is in front of C2 so first we would apply the algorithm to the child BSP tree containing polygons behind C2. There is no such tree, however, so we continue.
  * We render the polygon C2.
  * We apply the algorithm to the child BSP tree containing polygons in front of C2
    · This tree is just the leaf node D3, so the polygon D3 is rendered.

The tree is traversed in linear time and renders the polygons in a far-to-near ordering (D1, B1, C1, A, D2, B2, C2, D3) suitable for the painter's algorithm.

1.7 Questions

1. Why is the BSP Tree approach inappropriate for Volume Rendering?
2. What are the space and time complexities of the various spatial enumeration techniques?
3. How do the various techniques perform if the contents of the scene are moving?

1.8 Spatial Enumeration Ponderings

1. How might you optimally combine the techniques for scenes with static and moving content?
2. Would would be the implications of applying the octree approach to Volume Rendering?
3. Which of the techniques might be helpful in improving the performance of Ray Tracing? or for Radiosity?

2 Culling

For most graphical scenes it is likely that there's much more detail or content in the underlying model than is visible in any one rendered view; this is particularly true for interactive applications with dynamic content where objects and the viewpoint vary. There's an obvious performance advantage to be had if you can work out which bits of the scene aren't going to contribute to the rendered view (i.e. if you can't see them, don't waste time drawing them!). This might seem like an obvious thing to try to achieve, but it's not trivial to achieve, since you have to make sure that the cost of excluding something from being rendered doesn't exceed the cost of just rendering it. We'll look at a number of 'culling' techniques designed to achieve this.

2.1 Detail culling

The first, and most simple of these techniques is called 'detail culling'. The idea here is to not bother drawing things that are simply too small to have a perceivable effect on the final scene. This could be a small object reasonably close up, or a huge one in the far distance. All we need to determine here is the size of the object when projected onto the view plane – this could just be a simple bit of trigonometry based on the bounding cubious of the object (and culling out large objects in the distance is particularly satisfying, since spending a lot of effort drawing 1000s of textured polygons, only to find out that the result is a single pixel in the final image is obviously wasteful). This approach is particularly effective for moving scenes, since the viewer is unlikely to notice tiny bits of lost detail.

2.2 Backface culling

It's fairly obvious that for most objects you can't see all of them all at the same time; the chances are that some bits of the object are facing you and are therefore visible, and that some parts are on the 'other side', hidden from view. This is certainly true for opaque convex objects, where the front facing surface hides the back surface; it's a bit more complex for 'open' or concave objects (for example, you can see the back face of an open box). What this does mean though is that for a lot of things we could save time by simply not rendering the polygons that are facing away from the viewer. Figure 7 shows a simple model, first rendered with filled polygons, then with all its polygons visible, and finally with only those poylgons that have a surface normal facing the viewplane drawn. You can see there is a considerable saving in terms of the number of polygons shown; not surprisingly, somewhere in the region of 50% in this case. But how do you determine whether a polygon is facing towards or away from the viewer? There are two considerations to take into account here; first, it has to be a cheap calculation (otherwise we might as well just draw all the polygons and let the z-buffer take care of hiding the ones behind), and second it has to ideally be something we can do early on in the rendering pipeline (otherwise, if we've done most of the hard work transforming a polygon, we might as well just draw it anyway and again let the z-buffer do the occlusion work for us).

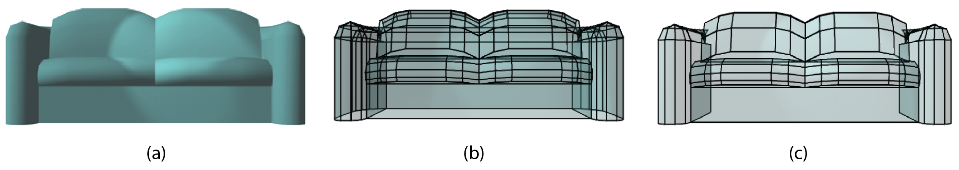


Figure 7: A model of a sofa rendered (a) with filled polygons, (b) showing the wireframe of its polygonal mesh, and (c) showing only those polygons that face the viewer.

We can do a simple test on polygons in object-space by calculating the dot product of the view vector and the polygon's surface normal, and looking at the sign of the result. If it's negative, the polygon faces away from the viewer and can be discarded; if it's positive, it's facing towards us and should be drawn (we are essentially testing wether the polygon is angled more than 90 degrees away from perpendicular to the view vector). The down side of this approach is that it does require a bit of trigonometry (recall the dot product of two vectors will require a cosine calculation) – but modern hardware often has this calculation available as a GPU instruction, so this isn't too painful (Figure 8.

There is one alternative to doing the calculation in world space, which is to look at the winding of the polygon as it is rendered onto the view plane; assuming that our model has a consistent polygon winding, then resulting polygons wound one way when projected on to the viewplane will be facing towards the viewer, and those wound the other will be facing away from the viewer.

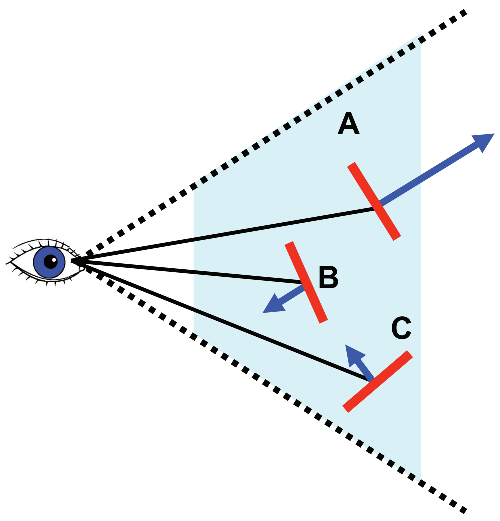


Figure 8: Backface culling in object-space. Polygons with normals within 90 degrees either side of the surface normal are facing towards the viewer.

2.3 Frustum Culling

The viewing frustum is a geometric representation of the volume visible to the virtual camera. Naturally, objects outside this volume will not be visible in the final image, so they are discarded. Often, objects lie on the boundary of the viewing frustum. These objects are cut into pieces along this boundary in a process called clipping, and the pieces that lie outside the frustum are discarded as there is no place to draw them. The various spatial enumeration techniques described in Section 1 make it possible to quickly determine which objects are candidates for culling.

2.4 Occlusion Culling

The Z-Buffer techniques takes care of the problem of objects near to the viewplane being 'drawn over' those further away; but it does rely on all the objects being drawn, and happens very late on in the rendering pipeline, so doesn't help performance. Ideally we'ld like to arrange for objects that are entirely behind other opaque objects to be excluded from list of things to be drawn early on — but this isn't trivial to arrange (imagine a very complicated polygonal object hidden by a simple single quadrilateral wall—it would be enormously wasteful to draw all the polygons only to later find that they are overwritten in the final image by pixels contributed by the single polygon wall). Approaches have been proposed that use techniques based on the Z-Buffer approach to solve the general problem, e.g. (Greene, Kass, and Miller, 1993).

At one (absurd!) extreme, you could imagine pre-computing all possible configurations of a scene, and remembering which objects are occluded from various viewpoints, but this of course isn't practical, and is especially problematic for scenes with moving content. One compromise is to create Potentially Visible Sets of objects; essentially dividing the scene in to regions and pre-computing their visibility (again, only really plausible for scenes with static or pre-determined movement).

Another approach to solving this problem is to exploit specific features of the scene being rendered. It's common, for example, that indoor scenes consist of enclosed spaces of one kind or another, with portals—doors and windows—through which other enclosed spaces are potentially visible (Figure 9. From any given viewpoint, then, its likely that you can potentially see things in the current room, and in any room to which there is a direct line of sight through 'portals' (Luebke and Georges, 1995). The 'portal culling' approach works like roughly like this:

1. From the current viewpoint, identify any portals that are partly or wholly visible in the current view frustum.
2. For each portal, cast a rays against the perimeter of the portal into other rooms.
3. For each room that a ray enters, identify portals that are visible in the view frustum, and repeat the ray casting process.
4. Each room that a ray passes through has contents that are potentially visible from the original viewpoint; create a new viewing 'frustum' (the portal need not be square, so it's not always technically a frustum) from the eyepoint that takes into account the frame of the portal, and repeat the process from Step 1 until all visible portals have been included.

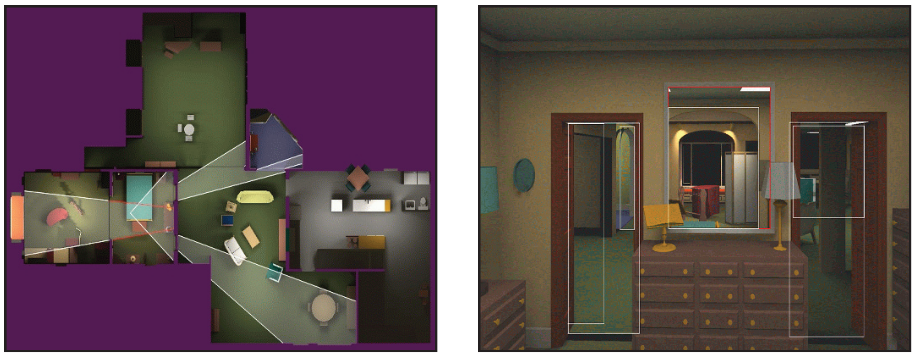


Figure 9: (a) A plan view of a series of rooms and their interconnecting portals, and (b) the view from one of the rooms, showing the outlines of the portals and the content seen through them.

2.5 Culling Ponderings

- Why is the paper by Luebke and Georges, 1995 called 'Portals and Mirrors'?

- Why is backface culling of polygons in screen-space (i.e. based on the winding of the resultant polygon) of little value in terms of performance?
- How could the portal-culling technique be modified to speed up the rendering of a city-like environment, consisting of densly built tall buildings when viewed from street level?
- Why might a model of an industrial plant consisting of lots of extremely long pipes pose problems for culling algorithms?

References

Greene, Ned, Michael Kass, and Gavin Miller (1993). "Hierarchical Z-buffer visibility". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '93. Anaheim, CA: ACM, pp. 231–238. ISBN: 0-89791-601-8. DOI: `10.1145/166117.166147`.

Luebke, David and Chris Georges (1995). "Portals and mirrors: simple, fast evaluation of potentially visible sets". In: *Proceedings of the 1995 symposium on Interactive 3D graphics*. I3D '95. Monterey, California, United States: ACM, 105–ff. ISBN: 0-89791-736-7. DOI: `10.1145/199404.199422`.