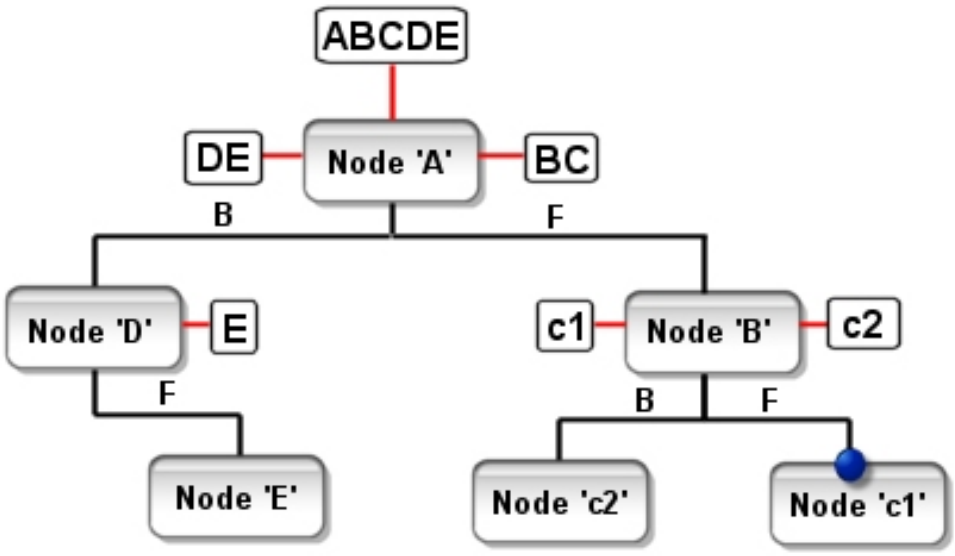
Chapter Sixteen

Spatial Partitioning III



Introduction

In this lesson we will continue our discussion of spatial partitioning by introducing Binary Space Partitioning trees. Binary Space Partitioning (BSP) is a technique that has been the cornerstone design element in some of the most powerful and successful 3D game engines on the market. When John Carmack and id software used BSP trees for Doom™ and later Quake™, BSP quickly became an industry buzzword. While these games have aged a bit over the years, you may be surprised to learn that virtually all of the latest first person shooters still continue to use BSP based engines in one form or another. With the more recent introduction of pixel shader programs that often perform many complex rendering passes per polygon, eliminating overdraw and reducing the number of polygons that need to pass through these shader programs has once again become the key to fast game performance -- an area in which the BSP trees (along with accompanying technologies) excel.

The reason this spatial tree was not covered in the previous lessons is because it really does deserve its own chapter due to the myriad ways in which it can be used and its importance in the field of computer science and game development. Having BSP trees at our disposal will allow us to perform perfect alpha sorting very efficiently, allow us to create spatial trees that have arbitrarily shaped leaves (leaves that are not always just boxes), and will allow us to divide the world into areas that describe whether that space is empty or solid. Empty space is space that is not currently occupied by any geometry and can be freely moved around in by the game characters, where as solid space is space that currently describes an area that contains a solid object (e.g., a wall). Because the BSP tree can be used to discriminate between solid and empty areas in a game world, it can be used to calculate a potential visibility set, which is the goal we will be working towards both in this lesson and the next.

A potential visibility set (PVS) is a pre-compiled set of data built by an application called a PVS Calculator that tells our application which leaves can be seen from every leaf in the tree. If leaf A is not visible from leaf B then leaf A's polygon data will not need to be rendered when the camera is in leaf B. Although this process seems similar to the visibility system we developed in the previous lesson, there is a significant difference -- a PVS calculator accounts for the occlusion of geometry when determining what is visible from a particular leaf. If your scene uses a large number of polygons, but the camera is currently located in a small room with no windows or doors, the only polygons that the visibility system would flag for rendering would be the walls, the floor, and ceiling of the room in which the camera is currently located. This would be true regardless of how many polygons were currently intersecting the view frustum. The polygons situated behind those walls are occluded by the walls and therefore would not be seen (and thus not rendered).

The BSP tree will play a crucial part in the construction of the PVS calculator we will create in the following lesson. It is this usage of the BSP tree and the PVS data that it aids in compiling that allowed games such as Quake™, Quake II™, Half Life™, and Medal of Honor™ (and the dozens of games built around those engines) to run at high frame rates despite the vastness of the levels. It is not overstating the point to say that the ability to calculate and process a potential visibility set is one of the most fundamentally important technologies in a 3D graphics engine; it represents a way to process and render only what is visible with virtually zero run-time processing, thus decoupling the performance of your game from the scene polygon count in the general case.

Later in this lesson we will study another BSP use case commonly seen in many 3D world editing packages, such as GILES™. The ability to carve one solid object from another or to fuse two objects into a single mesh are likely familiar to you if you have spent any significant time working with GILES™ or any other world editor package such as WorldCraft™. Indeed you can create very complex scenes by carving and merging simple mesh objects into more complex shapes. This technique is referred to as Constructive Solid Geometry (or sometimes, Geometric Boolean Operations). These techniques are once again made possible by the solid/empty space discrimination that can be made when the scene data is compiled into one or more BSP trees. In GILES™, each brush is compiled internally as a mini-BSP tree that describes which regions of the brush are in solid and empty space. When two brushes are merged into one (called a *union* operation), the polygons from each brush are sent down the other brush's BSP tree. Any polygons that end up in solid space (in either tree) are deleted, leaving two sets of remaining polygons (the non-deleted polygons from each brush). These can then be collected and added to a new single brush that is the union of the original two. BSP tree based Constructive Solid Geometry (CSG) techniques can be used to carve explosion damage into surrounding geometry, to merge a multi-brush level into a single static mesh, or to remove polygons that are not visible because they are embedded in other objects (e.g., removing the bottom face of a crate that is resting on the floor and would therefore never be seen). The latter process can help reduce the polygon count of our scenes and as you will see later, can be used to mold geometric data into a form that will describe solid and empty areas when compiled into a BSP tree.

At the end of this lesson we will have understood and implemented the following:

- **BSP Node Trees:** This tree type will be used for storing alpha polygons and rendering them in a pixel perfect back to front order. Using the BSP tree for this purpose will allow us to get perfect alpha blending results with all geometry configurations (something we have not yet been able to do in our framework). This tree will be constructed in Lab Project 16.1 to manage any alpha polygons that the scene may contain. The application will use this tree (after all non-transparent geometry has been rendered) to render its alpha list in correct back to front order. We will also develop a rendering solution for the BSP node tree that renders the alpha polygons it contains efficiently, trying to maximize batch rendering.

- **BSP Leaf Trees:** This is a tree that compiles in the same way as the node tree but differs in that it collects and stores the polygons at the terminal nodes (i.e., leaves) of the tree. This type of tree will be very similar to the trees we implemented in the previous lesson (with the kD-tree being its closest relative). Our implementation of such a tree can be derived from CBaseTree and can therefore use all of the same rendering functionality. The difference between a BSP leaf tree and a kD-tree is that the leaves are not necessarily box shaped, but can be arbitrarily sized and shaped.

- **Solid BSP Leaf Trees:** This is essentially the exact same tree as the BSP leaf tree with the exception that it expects to have geometry passed into it that meets certain standards. Such a tree (when supplied with the suitable geometry) will compile the geometry into arbitrarily shaped leaf nodes like the standard leaf tree, but this will be done in such a way that the tree will be able to tell us exactly which areas in the scene are considered solid (e.g., inside a wall) and which are considered empty. In the next lesson we will see that it is this property that will allow us to generate portals that are used by the PVS calculator to build the visibility sets for each leaf in the

tree. That is, it is vital that we have a way to compile a spatial tree that will tell us exactly what is solid and what is not. Using this information, our PVS calculator can calculate, from any leaf, exactly which leaves are occluded due to there being a solid obstruction in the way.

- **Constructive Solid Geometry (Boolean Operations):** It is important that we learn how to build a BSP tree that can provide us with the solid/empty information about the scene since we will need this information in our PVS calculator. As mentioned a moment ago, whether we can do so depends on whether the geometry we are being given meets certain standards that the BSP compiler will expect. The CSG techniques we implement in this lesson will allow us to write routines that attempt to mend data that would otherwise be considered unsuitable for the compilation of a solid/empty BSP tree. We shall see that CSG techniques will ultimately be used to generate the data needed to compile a PVS.

16.1 Introducing BSP Trees

Although the kD-tree we discussed in the previous lesson is in fact a tree that performs a binary spatial partition at each node (and could theoretically be referred to as a BSP tree in that sense), the more industry standard view of the term 'BSP tree' is a tree that partitions space at each node (into two halfspaces – thus, binary) using an arbitrarily oriented split plane. Most often, the term is used to describe a ***polygon-aligned*** BSP tree, where the split plane chosen at each node is not an axis-aligned plane (like we use in the kD-tree) but is taken from the polygon dataset. That is, each polygon input into the compilation process has its plane used as a split plane once. Using such a technique, the number of nodes in the tree will be equal to the number of polygons since each one's plane is used to split a node. Figure 16.1 shows how a polygon-aligned BSP tree would divide the space of a scene composed of five polygons (A through E).

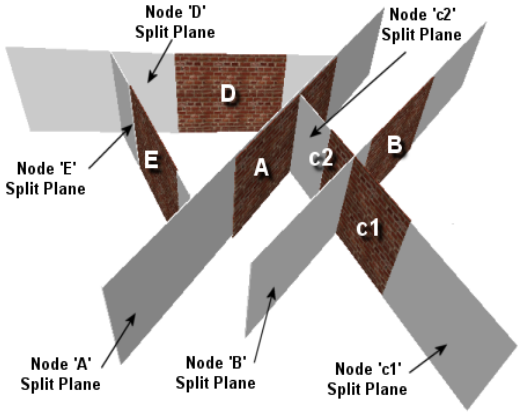


Figure 16.1 : Polygon Aligned BSP Tree

Since the scene is comprised of five polygons (forget about why polygon C is divided into two sub polygons at the moment), five nodes are created. The split plane stored at each node is the plane of the polygon that was selected from the polygon list at that node. As you can see, our scene is divided up into irregularly shaped regions, unlike the quad-tree, oct-tree and kD-tree we discussed in the previous lessons (which always carved the world into axis aligned box shaped regions). However, an important point to remember is that a BSP tree can be compiled using any arbitrarily oriented planes at each node; it need not use the planes of the polygons it has been passed for the compile process. As we will discuss in a moment however, there is great benefit in doing so under certain circumstances.

The compilation of a BSP tree is essentially the same as the building process of any of the other spatial trees we have developed thus far. Each node in the tree stores a split plane that is used to cut the space

represented by that node into two child nodes. While the kD-tree always used a split plane at each node that was aligned with one of the world axes, the BSP tree can use any arbitrarily oriented plane. The polygon list that makes it into that node is then divided into front and back lists, just as we saw with the kD-tree. Any polygons in the list that span the current node's plane are split and the relevant fragments added to both the front and back list. These lists are then passed into the two child nodes where a new plane is selected and the polygon data is further subdivided. This is all identical to the split trees we compiled in the previous lessons, and you will see that the code to both the BSP tree compiler and the kD-tree build function (for example) are almost identical. The only difference in our description so far is that the plane chosen to split each node is not necessarily chosen to be axis aligned; it can be arbitrarily oriented. As one might imagine, using arbitrary split planes at each node no longer divides the underlying region of space into a neatly organized stack of boxes.

There are two main flavors of BSP tree that we will develop in this lesson: node trees and leaf trees. A leaf tree works in the same way as the previous trees we have covered. The terminal nodes of the tree are the leaf nodes and they contain all the polygon data that belongs in that leaf (just as before). These are the polygons that are passed down to that terminal node during the build process. A node tree is similar, but not exactly the same. The only real difference between a leaf tree and a node tree is that the node tree does not contain the leaf structures that store polygon data at the terminal nodes. Instead a node tree might choose to store the polygon data in the nodes themselves or perhaps store no polygon data at all. With this small exception, the node tree is identical to the leaf tree.

In the next section we will introduce the BSP node tree, explain its properties, and see how it can be used for back to front rendering of polygon data. The BSP node tree we implement will generate its node planes from the polygon data compiled into the tree (i.e., a polygon-aligned BSP node tree). The polygons themselves will also be stored at the nodes. That is, each node will contain a split plane that was created by one of the polygons in the original set. That polygon and all others that share that plane will also be stored at the node. Although it might not be obvious why we would want to build a BSP tree that partitions the world using a set of polygon planes or why we would want to store polygons in the nodes at arbitrary levels of the tree (instead of at leaf nodes), all of this will be explained shortly.

16.2 Polygon-Aligned BSP Node Trees

In the early days of 3D game development, when end-users' systems were not equipped with hardware depth buffers and megabytes of video memory, one of the major tasks that faced any game engine programmer was how the scene was going to be rendered into the frame buffer such that polygons nearer the viewer obstructed polygons that were further away. These days, we take for granted the fact that the depth buffer will perform a per-pixel comparison before each pixel is rendered so that a pixel only gets rendered if its camera relative depth is less than the depth of a previous pixel already stored in the frame buffer at that same location. Prior to the introduction of 3D hardware acceleration and the inexpensive main memory found in the mid level PCs of today, 3D engine programmers did not have the luxury of throwing their polygons at the frame buffer in an arbitrary order, relying on the depth buffer to handle the occlusion of distant objects. On early systems, memory was very limited. It was barely possible to fit all the polygon data into memory, let alone allocate another large chunk of memory the size of the frame buffer just to store per-pixel depth information. Additionally, since these engines ran

completely in software, performing a per-pixel test prior to rendering each pixel was also something which could not be performed in real time on early microprocessors.

The 3D developer had bigger concerns than batch rendering by texture or material and in fact, was forced to use a technique that was mutually exclusive with respect to these concepts. The engine had to render the scene so that polygons further away from the viewer were rendered first and polygons nearer the viewer were rendered later. Remember, without a depth buffer, any polygon that was rendered would always be drawn over anything currently in the frame buffer. Even if the polygon that was rendered was further away from the camera position than the polygon currently occupying the same space in the frame buffer, that polygon would be overwritten. In other words, without any depth testing in the pipeline, rendering to the frame buffer was analogous to painting on a canvas. Regardless of the scene already painted on that canvas, anything you painted subsequently would be rendered over the top. Figure 16.2 shows a classic example of what can happen if no depth testing is available and the polygons are rendered in an arbitrary order. In this example we have a mesh of a corridor section consisting of five polygons. The long darker horizontal polygon is the polygon that forms the wall at the end of the corridor. Technically, it should be occluded by the left and right wall sections. However, because it was the last polygon that the artist added to the mesh, and we are simply rendering the faces of the mesh in the order in which they are stored, it is rendered to the frame buffer last, overwriting everything currently contained in the frame buffer in its overlapping pixel locations.

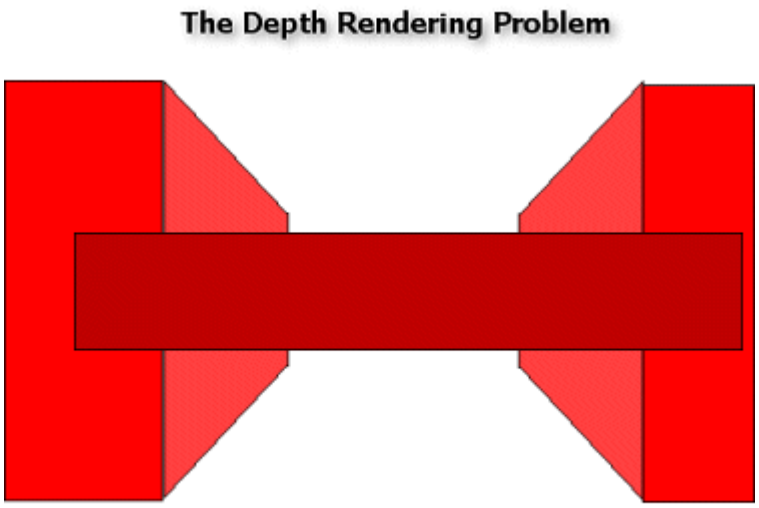


Figure 16.2

One of the early solutions to this problem was a rendering technique called the Painters Algorithm. The technique is so named because it is analogous to the way in which a painter builds up their scene on a canvas. Normally, a painter paints the background objects first and then later adds foreground objects. This ensures that the objects in the foreground are painted on top of the background objects, thus occluding them from view. The Painters Algorithm is a technique in which the polygons that are to be rendered are first sorted into a list based on their distance from the camera. The further a polygon is from the camera, the higher up in the polygon list it would be stored and the earlier it would get rendered to the frame buffer.

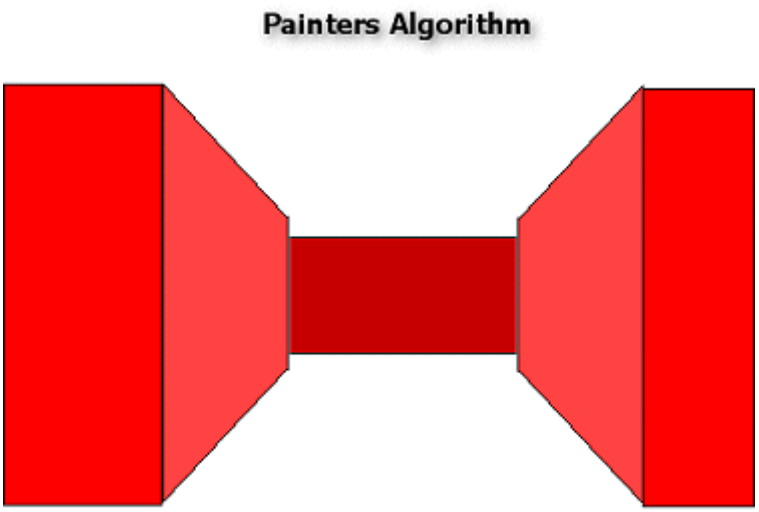


Figure 16.3

When the sorted polygon list was rendered from beginning to end, the polygons would be rendered in back to front order with respect to the camera. Nearer polygons that share the same screen space region as further polygons would automatically overwrite them, providing the correct depth based polygon occlusion.

In Figure 16.3 the same section of corridor is depicted as before, only this time the polygons have been distance sorted prior to being rendered. We can see that the first polygon to be rendered would now be the wall at the far end of the corridor. Since this one is rendered first, when the left and right wall sections are finally rendered, they automatically overwrite the correct sections of the far wall, giving the proper sense of depth.

So we see that in order to render a 3D scene correctly when no depth buffer support is available, we must sort our polygon data such that it is rendered in a back to front order with respect to the camera's current position. Keep in mind that the CPUs on these older machines were already constrained with just the actual rendering functionality; having to perform a back to front polygon sort before drawing each frame was certainly no help performance-wise. Of course, there are ways to improve depth sorting speed and efficiency. The hash table technique we implemented in Module I of this series was efficient enough, but it was far from being a perfect polygon sorter. The problem was not the technique itself, but the way that the distance to each polygon was originally calculated before being stored in that table.

Recall that in Chapter 7 of Module I, in order to sort the polygons (whether using a hash table or any other sorting technique) we had to use a point on that polygon as the sorting key. A common practice is to use the view space center position of the polygon such that the final list/table compiled describes a list of polygons sorted by the distance to their center positions from the camera. While using such a technique works quite well in a lot of circumstances, there are certain geometry configurations in which this technique will produce an erroneous render order (see Figure 16.4).

In Figure 16.4 we can see two polygons with a camera position represented by the sphere at the bottom of the image. The smaller polygon clearly overlaps the larger one from the viewer's perspective and therefore should be rendered into the frame buffer after the larger one. However, note the two arrows and their lengths representing the distance from the camera's position to the center of each polygon. We can clearly see that the camera is located closer to the center position of the larger polygon, even though the smaller polygon is in front of it. This means the method of sorting using the center point of each polygon would fail in this case and the smaller polygon would be rendered first, only to be partially overwritten by the larger polygon that is rendered afterwards. In this situation, the viewer would see depth artifacts.

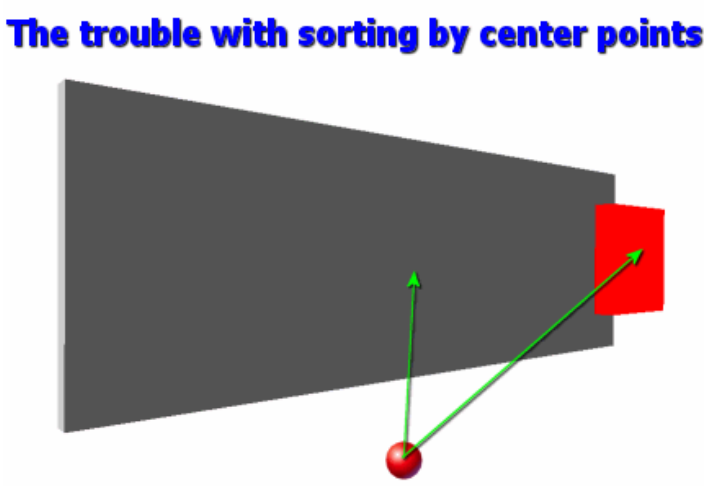


Figure 16.4

Even if a more thorough method was used to more accurately calculate the distance from the camera to the polygon and we could sidestep this particular problem, there are still certain geometric configurations for which a definitive draw order can not be ascertained. The configuration of the two polygon construct depicted in Figure 16.5 is so effortlessly handled by a depth buffer algorithm that it is easy to take depth buffers for granted.

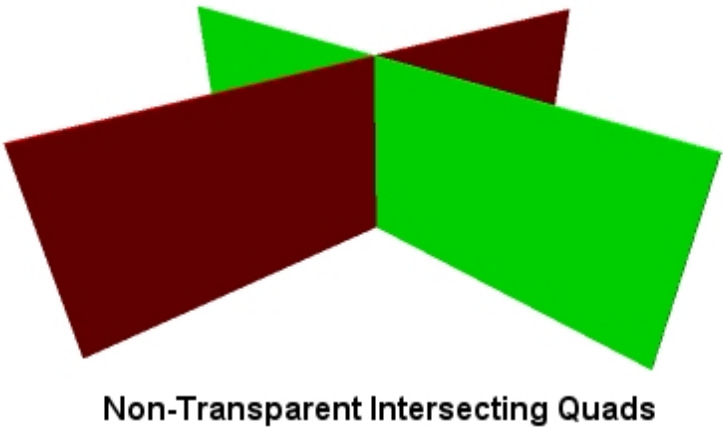


Figure 16.5

In this image we have two polygons that intersect each other to form an X shaped configuration when viewed from the camera's current location. The dark red polygon that starts at the bottom left of the image is both in front of and behind the green polygon that starts at the bottom right of the image. However, the green polygon is also both in front of and behind the red polygon.

With a depth buffer, the order in which these polygons are rendered is insignificant. If we render the red (right) polygon first, then the depth values for each of its pixels will be written the depth buffer first. When the green polygon is rendered, some of its pixels will have greater distances than those currently containing the red polygon's pixel and as such these pixels will not be rendered. We can see this in the top left of the image where, even though the green polygon was rendered second, some of its pixels are never rendered because they are found to lay behind some of the pixels of the red polygon. In the bottom right section of the image however, where we can see that the green pixels would have smaller depth values than the red pixels already contained at that location, the

red pixels are correctly overwritten by the green pixels. We have a situation where the two polygons both occlude and are occluded by each other. The depth buffer handles this case perfectly.

Looking at the above diagram one might wonder how such an arrangement could be rendered if the depth buffer was not at our disposal. Which polygon should be rendered first? As it happens, there is no solution to this problem using only polygon sorting. If the green polygon was rendered first then when the red polygon was rendered, it would overlap all the pixels in the green polygon sharing the same screen space locations, even if the red pixels were further away from the view point than the green pixels already drawn there. If the red polygon was rendered first, the same would happen in reverse and part of the red polygon would be overwritten by the green polygon in areas that should have been occluded by it. The only way this situation can be solved is by splitting the polygons into multiple sections such that the ambiguity is removed and a clear render list can be compiled for those polygon segments. This is one of the things that a BSP tree does when it compiles its data; it makes sure that when the tree is finally built, no situations like the one shown above exist in the geometry. Thus, a clear back to front rendering order can always be known at runtime.

16.2.1 When Depth Buffers Will Not Suffice

So far we have discussed the problems that faced the game engine developers of yesteryear when a depth buffer was not available for perfect depth sorting. While this discussion may have seemed nothing more than nostalgic now that we always have access to depth buffers, this is far from the case. In Chapter 7 we learned that we will need to render our alpha (i.e., transparent) polygons in a back to front order so that blending operations are carried out correctly. If we simply throw alpha polygons at the pipeline in an arbitrary order, artifacts can result since many common blending modes are order dependant.

Because we want transparent polygons to allow what is behind them to show through, we will want to render them after the main scene has been rendered. This way, the alpha polygon does not get its depth values written to the depth buffer and prevent geometry that is behind it (but should be seen through it) from being rendered. If we are looking at the outside of a house model, we should be able to see through its windows and into the room(s) on the inside. If the window was rendered before the room behind it and depth writing was being used to record the depth values of the alpha polygons in the depth buffer, when we tried to render the section of the room behind the window later, it would be rejected by the depth test. Thus, we will want to render our alpha polygons after all opaque polygons have been rendered to avoid this situation.

However, even if the alpha polygons are rendered in a separate pass after the rest of the scene has be constructed in the frame buffer, this situation can still occur if alpha polygon A is in front of alpha polygon B, but rendered first. The depth values for A will be written to the depth buffer first, and when alpha polygon B is rendered later, its pixels are depth rejected and never rendered behind polygon A. Since polygon A is supposed to be transparent, we are supposed to see polygon B behind it. Therefore B's pixels still should have been rendered. As we learned, we can stop this from happening by simply disabling Z writing when we render the alpha polygons. This will ensure that they are correctly obscured by the opaque polygon data already contained in the frame/depth buffer and that alpha polygons

(rendered with Z writing disabled) do not occlude one another and prevent each other from being rendered.

This would work fine were it not for the fact that the blending operations that we perform are often order dependant. If polygon A is a blue window and polygon B (behind it) is a red window, the red polygon should be rendered first and the blue polygon rendered second so that we see a red window that has been tinted blue. If the red window polygon was rendered second, we would have incorrect blending where the two polygons overlap (we would see a blue window tinted red). This would indicate that we are viewing the blue window through the red window instead of the other way around.

The only way to render our alpha polygons correctly is to render them in a second pass with a perfect back to front ordering. As we can no longer rely on the depth buffer, and must sort the polygons ourselves, we find ourselves with the same problem that faced the early game engine developers. How do we render a set of polygons in a perfect back to front order that gives correct blending results every time, regardless of how the polygon data is oriented? Admittedly, we only have to do this for our alpha polygons (instead of the entire scene), but even so, a solution is required.

The hash table technique accomplished most of these goals, but occasionally failed due to the fact that it used polygon center positions as its sorting criterion. In Figure 16.6 we see both the intersecting polygon problem and the blending order problem. We are using the same two polygons from Figure 16.5, but now made transparent. Compiling polygons into a back to front render list is not always possible (see Figure 16.5) and by using the same example again but with alpha polygons, we also see the blending errors that are produced.

For the set of polygons shown in Figure 16.6, there is no clear draw order. In this example we have decided that because no draw order can be determined, we will just render the red (left) polygon first and the green (right) polygon second.

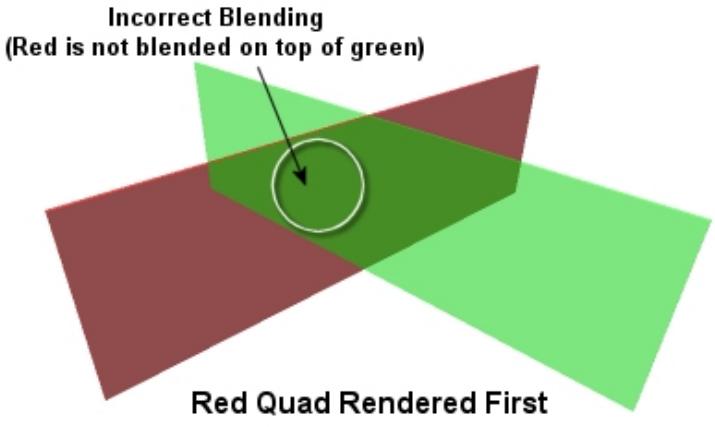


Figure 16.6

Because the green quad was rendered afterwards, it overwrites a portion of the red quad that is supposed to be in front of it. The rough area of green pixels that are supposed to appear to be behind the red polygon are highlighted by the circle in the diagram. Because the green polygon was rendered second, the blend order is wrong. The color of these pixels should produced by viewing a green pixel through a red polygon, but instead we are viewing red pixels through a green polygon. Of course, if we were to render the green polygon first, then the opposite error would occur, as shown in Figure 16.7.

This example does not look as obviously incorrect, but after some investigation you will see that the approximate area of pixels in the region of the white circle in the red polygon should be located behind the green polygon and as such, should have been produced by blending green on top of red instead of red on top of green.

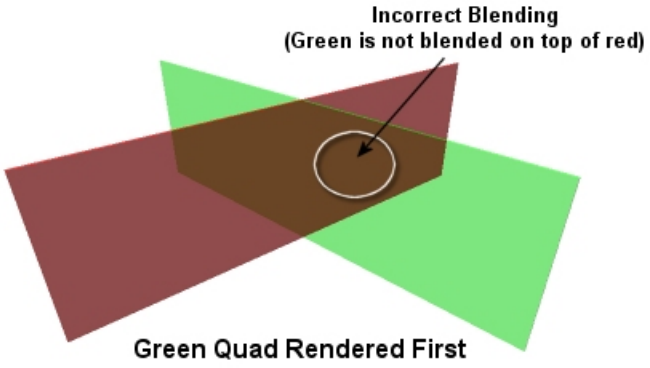


Figure 16.7

So even if we sort our polygons in a back to front rendering order so that the alpha polygons are blended in the correct order, we are still not fully covered in all cases. In a situation like this, there simply is no correct drawing order. Whichever polygon we render first, part of it will be overwritten by the second polygon in areas where it should not have been.

Figure 16.8 shows how we would like the two alpha polygons to be rendered. This is what compiling a polygon-aligned BSP node tree for our polygon data will allow us to do.

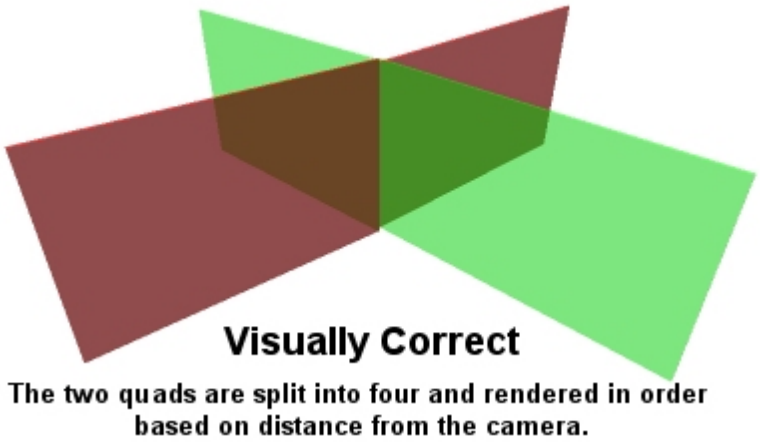


Figure 16.8

In this example, the BSP tree has used the plane of the green polygon to split the red polygon into two pieces and has used the plane of the red polygon to split the green polygon into two pieces. The original two polygons have now been replaced by four non-intersecting polygons which can be correctly ordered back to front.

In this example we can see that the back two quads (one red and one green) would be rendered first in the alpha pass. When the second two quads are rendered, the red quad is correctly blended over the back green quad and the green quad is correctly blended over the back red quad, giving us a perfect render order and perfect alpha blending.

In Lab Project 16.1, after every alpha polygon has been loaded and added to the BSP tree, the tree will be compiled. As we will discuss in the next section, this tree will allow us to render alpha data in a perfect back to front order from any camera position even though the tree is only built once. It should be

noted however that just because we are using it to sort alpha polygons, BSP trees could be used to sort the entire scene into a back to front rendering list. Indeed it was often the BSP tree that allowed the game engines of old to render polygons to the frame buffer with the correct depth occlusion without the aid of a hardware depth buffer. Thus, in the next section we will discuss building BSP trees from the more general perspective of compiling arbitrary polygon sets (not just alpha polygons).

16.2.2 Building our First BSP Tree

When we think about the spatial trees that we developed in the previous lessons, we can perceive a tree to be an arrangement of spatial relationships. Each node in the tree describes its position in the world relative to other nodes in the tree. In the case of a kD-tree for example, we know when we visit a node that the sub-tree of nodes attached to its back pointer are all representing space that has subdivided the space behind the current node's split plane. The sub-tree of nodes stored in its front list represent the space that subdivides the space in front of the current node's split plane. Thus we can think of the kD-tree as describing a collection of planes with an established relationship to one another. We know exactly which node is behind another node, and vice versa.

In order to render a list of polygons in a back to front order, we need to establish those same relationships between polygons. That is, we need to know if one polygon is behind another. Therefore, if the kD-tree tells us the relationships of planes to one another, we can use the exact same technique for our polygons. We can just use the planes of the polygons themselves as the split planes. Because we know that building a tree for these planes will describe their relationship to one another, it will therefore tell us the relationships of the polygons used to create those planes.

The process of building a BSP node tree is simple. The build procedure is passed the list of polygons that needs to be compiled into the tree. At each node, a polygon is selected from the list of polygons that made it into that node. The plane on which the selected polygon lies becomes the split plane for that node and (this is the important part) the polygon itself, and any others that may lie on the same plane, are removed from the list and stored at the node. The remaining list of polygons is then classified against the node plane and split into two polygon lists that belong in the front and back halfspace of the plane. The front list is then traversed, where it is used to create a branch of nodes in the same way as before. The back list is also traversed so that a series of back nodes are constructed, again in the same way. At the end of the process we will have compiled a BSP tree containing N nodes, where N is the number of polygons from the original dataset that lay on unique planes. Each node will store a split plane just like the kD-tree, although this split plane may not be axis aligned; it will instead be the plane of the polygon that was selected from the list at that node. Each node will also store one or more polygons that lay on that plane. Although we often refer to a tree of this type as having no leaves, what this really means is that all the polygons are not stored in the terminal nodes. If you wish, you can think of a node tree as being a tree where every node acts a leaf, because each node will store renderable data.

When the tree is built, we will be able to traverse the nodes of the tree and perform a classification test with the camera position at each node. The camera will be "pushed down the tree" so that it processes the nodes that are furthest from the viewer first. As each node is visited, the polygons stored at that node

(i.e., that lay on the node plane) are rendered. Therefore, while the tree is generated once and remains static, the order in which we visit the nodes of the tree depends on the camera position passed into the traversal function. The end result is a static tree that can be traversed in a back to front node order from any position in the scene. We will look at the rendering logic for the tree a little later; for now we will focus on the tree building process with a very simple example.

Figure 16.9 shows a level consisting of three polygons labeled A, B, and C. The camera position is not known or considered when building the BSP tree (i.e., it is view independent).

Using the example polygon set in Figure 16.9, the three polygons would be registered with our BSP tree at load time. After the polygons have been added to the tree, the BSP tree's Build function would be called to start the recursive compile process.

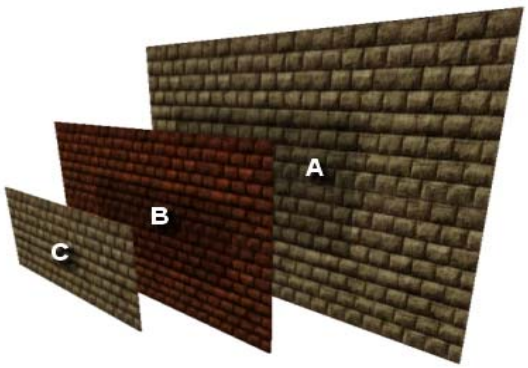


Figure 16.9

The first thing that would happen is the root node would be created and a polygon would need to be chosen from the list to use as the split plane for that node. The polygon we choose from the list at each node can be randomly selected from the list of polygons that made it into that node and the tree will still be successfully compiled (it can even be traversed successfully in the correct back to front order). However, we will learn later that there are tests we can perform at each node to find polygons which make better candidates for split planes higher up the tree, and are thus preferred. For now however, we will assume that polygon B has been chosen at random for the root node.

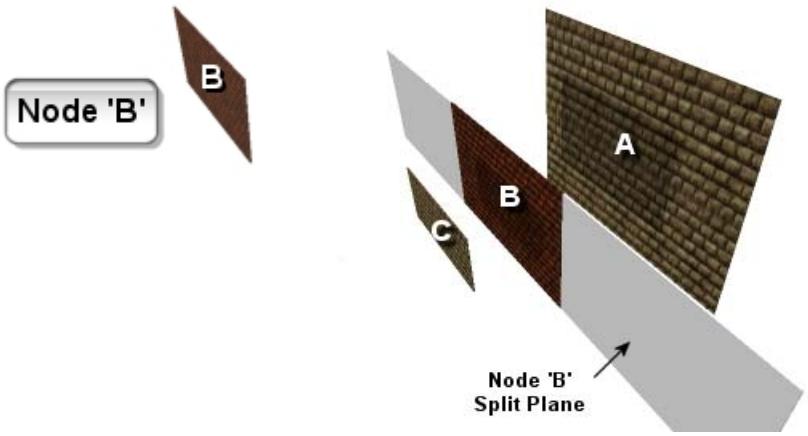


Figure 16.10

So polygon B is selected first and is removed from the list. The plane for polygon B will be calculated and stored in the root node (Node B) as the split plane. Polygon B will also be stored as the renderable data at that node.

At this point our tree has one node that stores a split plane, shown as the gray slab in Figure 16.10. Polygon B itself is also stored in the node, leaving only polygons A and C in the polygon list. This polygon list is then classified against the node plane to create two lists for both the front and back children of the root. We can see that polygon A, when classified against the split plane of node B, is found to be behind the plane and is added to the back list. Polygon C on the other hand is in front of the plane and is added to the front list. We have now done all the work we need to do at the root node. We have a split plane and a polygon stored at that node, and two polygon

lists describing the polygons that are both in front and behind the node plane. In this simple example there is currently only one polygon in the back list (A) and one polygon in the front list (C).

> **Note:** The split plane is infinitely extended in all directions, but we chose to display only the portions that match the vertical dimensions of the polygon for ease of viewing in the diagrams.

As the front and back polygon lists are not yet empty, we must recur down the front and back of the node and create additional child nodes. Two nodes are created and attached to the root node's front and back pointers. The function then recurs into each child passing in the respective list. In this example we will assume that we step into the back child first, passing in the back list which contains a single polygon (A).

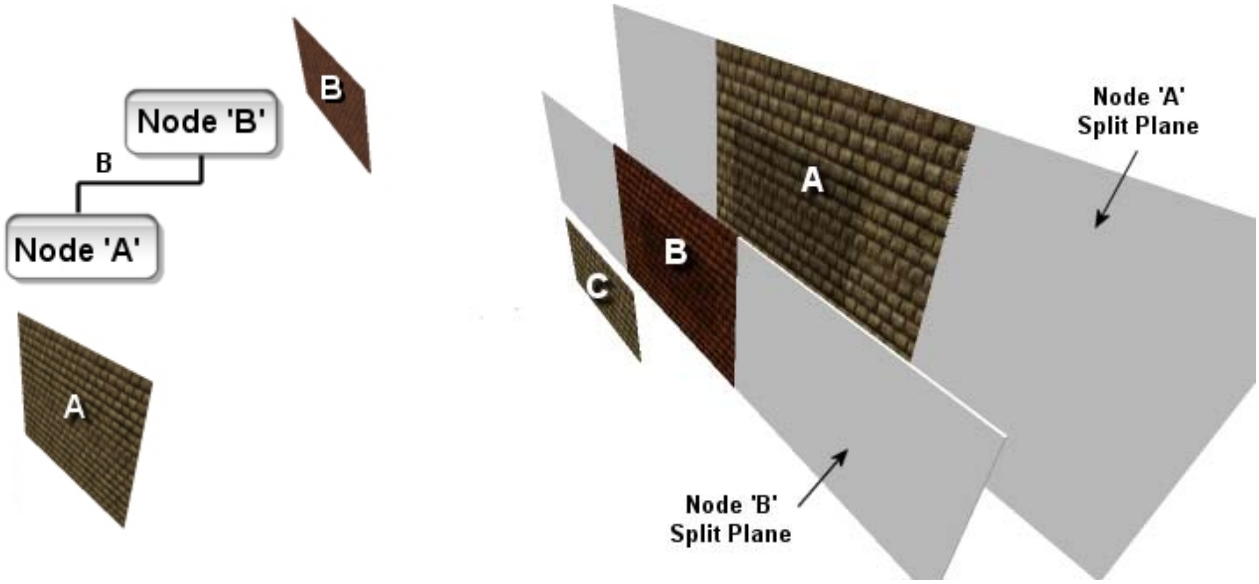


Figure 16.11

When we arrive at the back child of the root, the same process happens. A polygon has to be selected from the passed list so that its plane can be used as the split plane at that node. Since the polygon list passed into the back child contains only one polygon, Polygon A, the choice is easy. Polygon A is removed from the list and stored in the node. The polygon's plane is calculated and stored as the split plane for this node, which is called Node A in the diagram. As the polygon list is now empty, we have reached a terminal node. Because there is no polygon data left to subdivide, we no longer have to compile any front of back polygon lists for this node and we do not have to create any child nodes. With polygon A and its plane stored in the node, we return back up to the root.

At this point we have processed the root's back child but have not yet traversed into the front child with the front polygon list. This is done as the final step for this node.

As we step into the front child, we once again find ourselves at a new node that needs to have a plane selected for it from the passed polygon list. In this example, the polygon list passed into the front node contains a single polygon (C), and therefore this polygon is removed from the list and stored in the node along with its plane. As the polygon list is now empty, no child lists or nodes have to be compiled and

the function can step back up to the root. Once back at the root node, we return from the build process and hand program flow back to the application.

Figure 16.12 shows the final representation of our three polygon BSP tree after the front node has been visited and polygon C assigned to it.

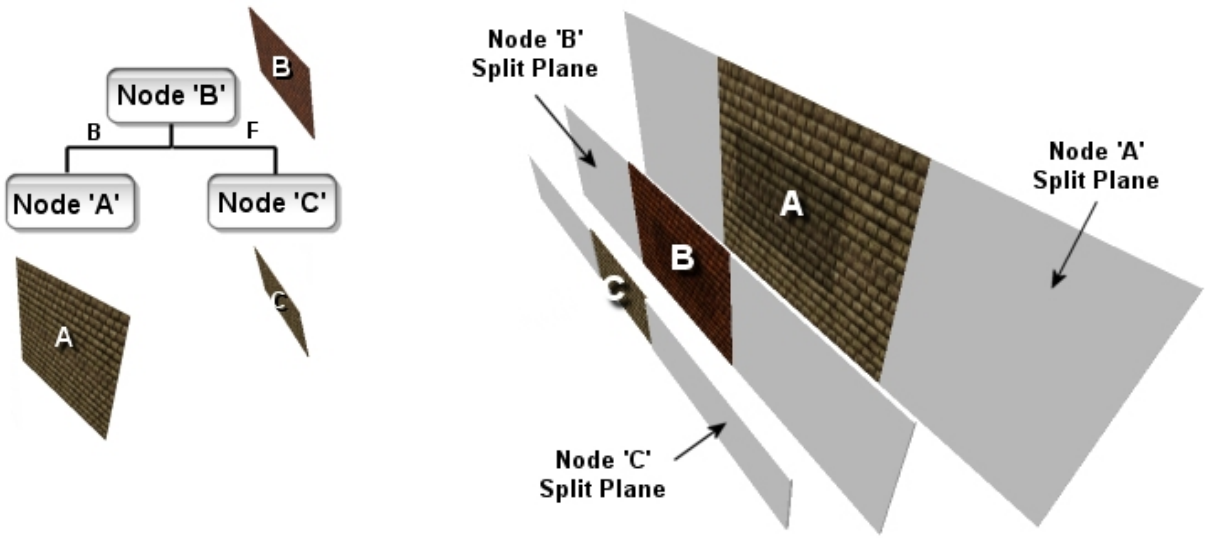


Figure 16.12

Looking at the tree diagram on the right hand side of Figure 16.2, you can see that we have created a three node tree from our list of polygons and we have stored a polygon at each node.

16.2.3 Rendering Our First BSP Tree

In this next section we will examine how to render a BSP node tree in perfect back to front order using the simple example tree discussed in the last section. We will look at two examples using two different camera positions so that we can see that while the tree is compiled once at startup and never changes, it can be traversed such that the polygon draw order is dynamically created in a back to front order from any camera position.

The rendering of the BSP node tree is a typical traversal which, as we would now expect, will be implemented using a recursive function. The application will render the tree by calling this function for the root node (passing a root node pointer and a camera position). The function will visit each node and perform a fast and simple test to determine which child should be visited first. This will depend on which halfspace of the current node's plane the camera is currently contained in.

For each node visited during the render traversal, the basic logic performed is as follows:

- Classify camera position against node plane

- If camera position is in frontspace of plane
  - Step into back child
  - Render polygon stored at the current node
  - Step into front child

- Else if camera position is in backspace of plane
  - Step into front child
  - Render polygon stored at the current node
  - Step into back child

This traversal logic at each node makes a lot of sense. We are essentially stating at each node, if the camera is in front of the plane, render the stuff behind it first, then render the polygon stored at this plane, and finally render the stuff in front of the plane. Alternatively, if the camera is behind the current node plane being tested, render everything in front of the plane first, render the polygon data stored at the node, and then finally render the data behind the plane. This traversal logic means that at each node the choice of which order in which to visit each child node is dependant on the camera position. Thus, when the entire tree has been visited, the node's polygons will have been rendered in a back to front order with respect to the position of the camera.

Imagine that the node structure for our BSP tree looks something like this:

```
struct BSPNode{
  POLYGON *splitter;
  BSPNode *FrontChild;
  BSPNode *BackChild;
};
```

Given the above, the following semi-pseudo code shows how our recursive BSP tree rendering function might look. This function would be called by the application just once each frame and passed the root node of the BSP tree and the current camera position.

```
void RenderBSP (BSPNode *CurrentNode , D3DXVECTOR3 &CameraPosition)
{
  int Result = ClassifyPoint( CameraPosition, CurrentNode->Plane );

  if (Result == Front || Result == OnPlane)
  {
      if (CurrentNode->BackChild != NULL)  RenderBSP (CurrentNode->BackChild );

      DrawPolygon(CurrentNode->Polygon);

      if (CurrentNode->FrontChild != NULL) RenderBSP (CurrentNode->FrontChild);
  }
  else
  {
      if (CurrentNode->FrontChild != NULL) RenderBSP (CurrentNode->FrontChild);
      if (CurrentNode->BackChild != NULL)  RenderBSP (CurrentNode->BackChild );
```

```
    }
}
```

In this example, the DrawPolygon function is assumed to be a function that renders the passed polygon to the frame buffer. Notice that there is no need to render the polygon data stored at the node if the camera is located in its back space. This would obviously mean the camera is looking at the back of the polygon, which would be back face culled by the pipeline anyway.

Rendering Example 1

In this first example which uses our three polygon BSP tree from the previous section we are given a camera position as seen in Figure 16.13. We can clearly see that in order for the polygons in the tree to be rendered in the correct back to front order we should render polygon A first, followed by polygon B and then finally polygon C. What is also worth noting is that the orientation of the camera is not used in the traversal logic at all, which may at first seem strange. Only the camera position is used and classified against the node planes to determine the correct drawing order.

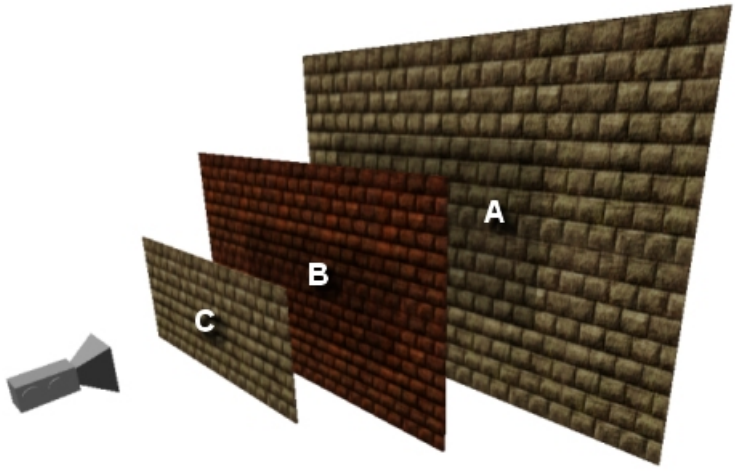


Figure 16.13

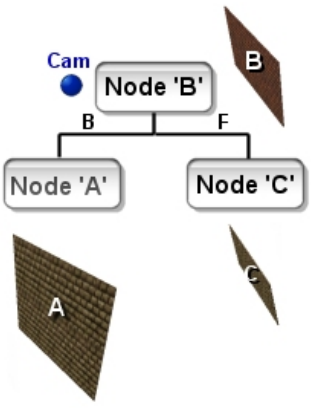


Figure 16.14

In Figure 16.14 we start our traversal of the tree by passing our camera position into the root node. This is shown as the blue dot labeled ‘Cam’ in the image. As discussed and shown in the previous pseudo code section, the first thing we do is classify the camera position against the plane stored at Node B. In Figure 16.13 we can see that the camera is currently in front of the split plane stored at Node B

Note: Remember, the split plane stored at each node is the plane that the polygon lies on. Therefore, we can see that the camera position would be in front of Node B because it is in front of polygon B, which is just a subset of the same plane.

As the camera is currently in front of node B, our logic tells us that we must visit the back child first before rendering the polygon stored at this node. So in the next step, we traverse into the back child of node B and arrive at Node A.

As Figure 16.15 shows, when the camera arrives at node A, it is once again classified against the split plane stored there and we determine that it is in front of node A. This would normally mean that we

would step into the back child first before rendering the polygon stored at that node. However, as no back list exists, we skip that part and simply render the polygon stored in the node (A).

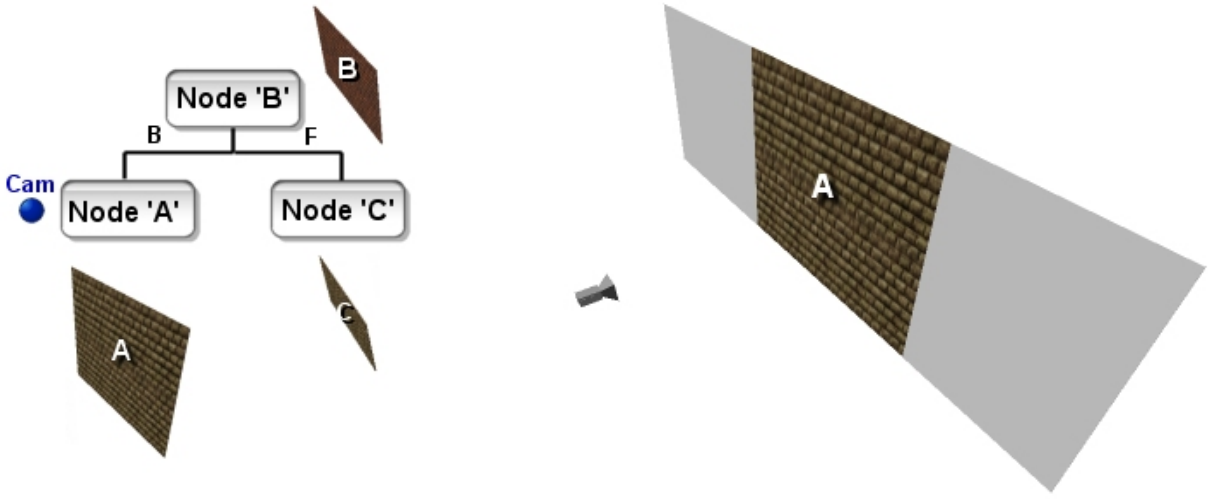


Figure 16.15

After rendering the polygon at node A we would step into the front child next, but this node is a terminal node that has no front list, so we return from the current instance of the recursive function and pop back up to the root node just after the point where the back child was stepped into.

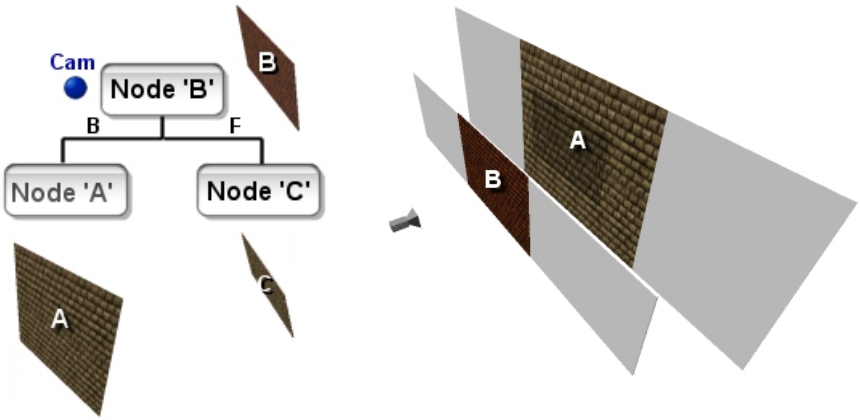


Figure 16.16

Before we stepped into the back child of the root node, we determined that the camera position was in front of the root node. This determined that we had to render the back child first, then render the polygon stored at the node and then traverse into the front child. We have now completed the first step and have stepped into and have returned from the back child. Therefore, our next job is to render the polygon stored at the node. The polygon stored at the root node B is polygon B, so we render this polygon next, as shown in Figure 16.16. Since this polygon is closer to the camera than polygon A, some sections of polygon A will be overwritten by polygon B in the frame buffer. This is absolutely correct because, from the camera's view point, polygon B should partly occlude polygon A.

We have now performed two of the three steps we had to perform at the root node (we rendered the back list and rendered the polygon stored at the node). All that is left to do now is step into the front child of the root node, where we visit node C.

Note: Remember that the order in which we decide to traverse into the front and back lists is dependant on the camera position. If we are in front of the plane, we step into the back child first. If we are behind the plane, we step into the front child first. This ensures that we always visit the nodes first that are in the opposite halfspace of the current node's split plane with respect to the camera position.

As Figure 16.17 shows, when we visit the front child we find that it has no children, so we simply render the polygon stored at the node (i.e., polygon C). As polygon C was rendered last, it will overwrite sections of polygon B in the frame buffer, which again is absolute correct. Polygon C is closer to the camera and therefore should occlude polygon B to some extent.

Figure 16.17

After polygon C is rendered, we return from node C back up to the root. At this point we learn that the root node has performed all its tasks, so the root node call returns and program flow passes back to the application. Although this was a simple example, we have just seen how to use a BSP tree to render our small scene in back to front order. Again, the same function can be used to render the same tree in a perfect back to front order even when the camera position changes. Stepping through one more rendering traversal using the same BSP tree with a different camera location will solidify our understanding of the traversal process.

Render Example 2

In this second example we will use the same BSP tree, but move the camera such that it is situated between polygons B and A as shown in Figure 16.18. Once again you should take note that although the camera in Figure 16.18 is facing in a given direction (in fact, almost opposite the earlier orientation), this is not factored during the traversal. Rendering order is determined using only the camera position.

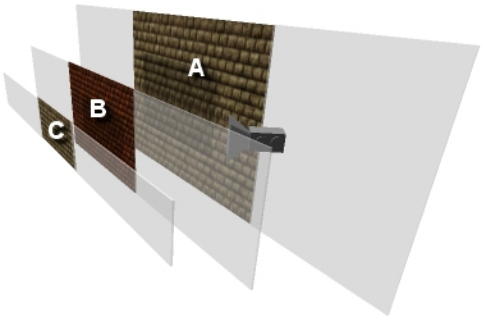


Figure 16.18

As before, we start off with nothing rendered and send the camera position into the root node to begin the traversal process.

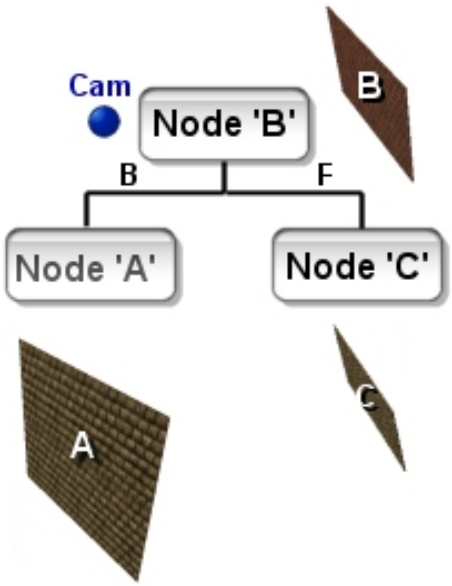


Figure 16.19

At the root node we classify the camera position against the plane of polygon B and find that it is located in the plane's backspace. This differs from our previous example where the camera was located in the root node's frontspace. As such, the order in which we traverse into the front and back children must be reversed. Because the camera is now located in the backspace of plane B we know that the more distant polygons must be located in the frontspace of B, and therefore should be visited and rendered first.

The order in which we do things at the root node will now be to step into the front child first and then render the polygon stored at the node (polygon B) before finally stepping into the back child. As we are currently at the root node, our first task is to step into the front node (Node C).

At Node C we find that we are visiting a terminal node and as such, there is no front or back child to traverse into. Therefore we just render the polygon stored there (see Figure 16.20). However, because the camera is behind the polygon it would be back face culled, so we would normally only render the polygon data stored at any node if the camera is in the plane's frontspace. But in this particular example, we will go ahead and render it for the purposes of demonstrating the correct traversal order being executed. At this point we have rendered C as our first polygon and when the function returns, we pop back up to the root node.

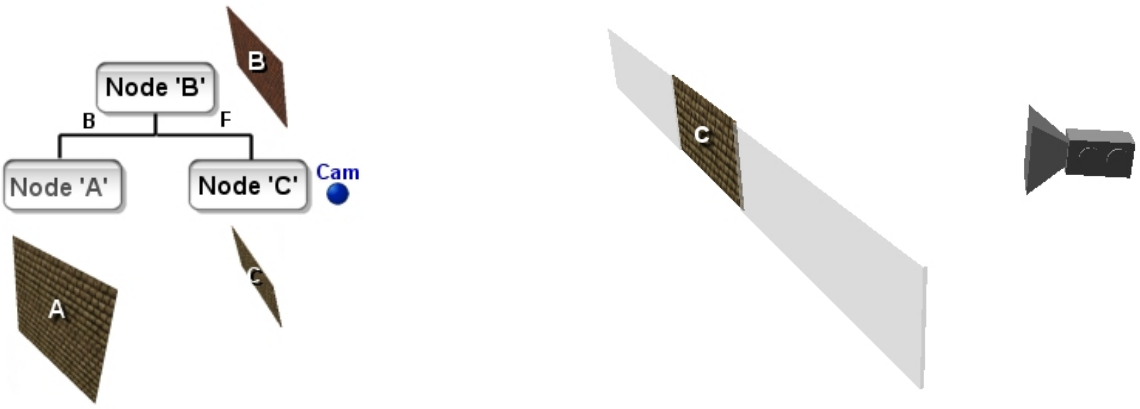


Figure 16.20

Once we are back at the root node, having traversed and (potentially) rendered the front tree of Node B (which in this example contains just Node C), we then render the polygon stored at Node B. However, once again, as the camera is in the backspace of node B, there is actually no need to render the data stored there as it would ultimately be back face culled. This means polygon B becomes the second candidate for rendering this time around (see Figure 16.21). Again, for this particular example, we will render the polygon anyway (at least in our images) so that you can see the order in which the polygons would be rendered if they were actually facing the camera.

Figure 16.21

Having rendered the polygon at the root node, we then step into its back child where we arrive at terminal Node A. This node then has its polygon rendered, as shown in Figure 16.22.

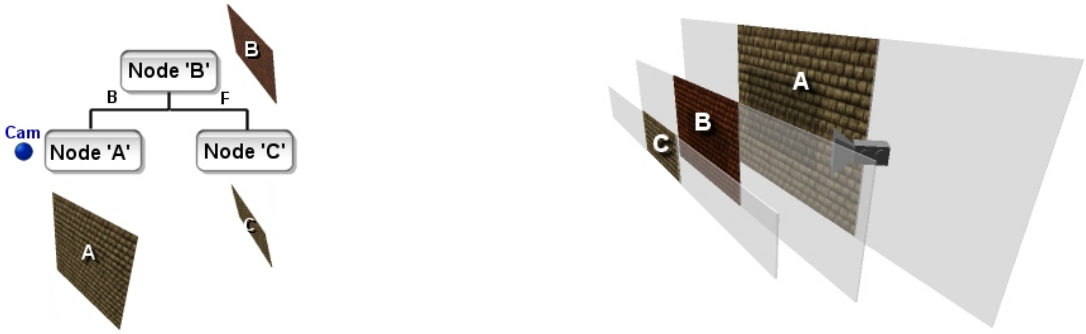


Figure 16.22

Just remember that in reality, polygon A would be the only polygon rendered in this scene as it is the only one in which the camera is located in its front space. At this point the polygon has been rendered and we step back up to the root node and return program flow back to the application.

We have now seen two examples of rendering the same BSP tree using different camera locations. It should be clear to you that even though the tree was only compiled once (hopefully in an offline process, although that would not be necessary for our simple example), it can be dynamically traversed at run time to generate a perfect back to front drawing order for polygons.

16.2.4 Perfect Back to Front Ordering with BSP trees.

In the previous section we examined a very simple example of a three polygon scene compiled into a BSP tree which we rendered using a specific set of traversal logic to determine a back to front draw order. Of course, the example BSP tree we used was deliberately simplistic to teach the fundamentals of BSP tree construction and traversal. All we have really done so far is introduce another technique to essentially give us the same results as our hash table (or any other sorting technique). After all, the polygons did not intersect each other and cause the ambiguous polygon order scenario. If our above example had contained such polygons, the technique we discussed would have failed since the same

problems still exist. So what if we were to discover a situation during the build process where polygon A is both behind and in front of polygon B? Should it be added to the node as a back child, a front child, or split by the node plane and have each fragment added to the relevant lists? The answer is that we must split polygons that span planes during compilation. Indeed it is this very concept that solves the geometric ambiguity issue for intersecting polygons.

In order to understand the full BSP tree building process, we will use a slightly more complex set of data that causes some spanning polygons that will need to be split during the build process.

In Figure 16.23 we see five polygons labeled A through E. We will assume that the camera is looking at the front of all the polygons here. That is, the polygon normals all face in the general direction of the camera. Looking at this collection of polygons, we have to try to distinguish which polygons should be rendered first and which should be rendered last based on the camera position.

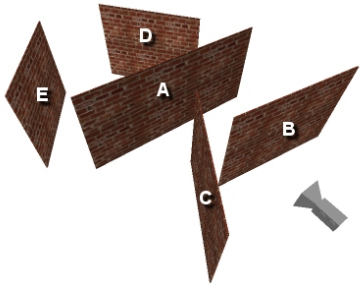


Figure 16.23

We can easily see that the first two polygons to be rendered should be polygons E and D, followed by polygon A. But which polygon should be rendered last, C or B? C is spanning the plane on which polygon B lies and is therefore both in front and behind polygon B. The order in which to render these polygons is now unclear and cannot be immediately determined. We can also imagine that such a geometric arrangement would be problematic in the build process of the tree using the techniques described above. Imagine for example that polygon B was chosen as the root node of the tree. We know that we then must classify the other polygons against this node plane to send them into the front or back list. We can see that polygons A, D and E would all be assigned to the back list of B but which list would polygon C belong to if it exists in both the front and back halfspace of node B's plane?

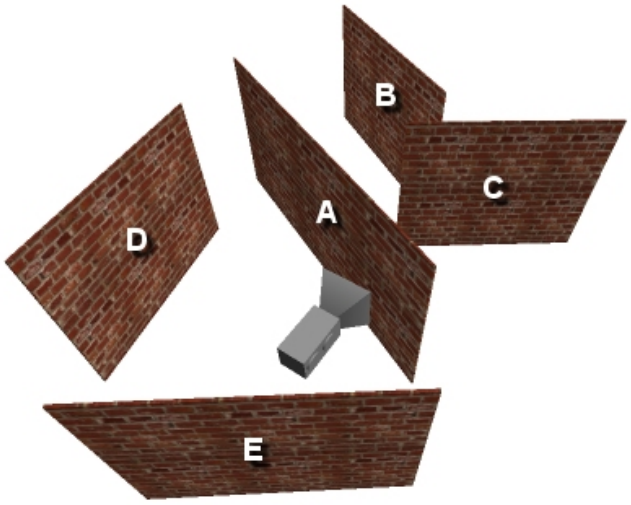


Figure 16.24

Figure 16.24 shows the same scene with the camera in a different position. This arrangement more clearly shows the problems that occur when one polygon spans the plane of another polygon.

Imagine what would happen if polygon C was chosen to create the split plane at the root node. Both polygons A and D would be spanning this plane, and this is not something we can allow to happen. For the BSP tree to provide a correct back to front polygon collection routine, every polygon must be either in front, behind, or on plane with another polygon. As soon as we have a polygon that spans a node plane anywhere in the tree, we lose the relationship between the polygons that tell us precisely

which polygon must be visited first.

The solution to this problem is simple. When a node is created, any polygons remaining in the list that made it into the node are classified against the node plane to decide whether they belong in the front or back list. Any polygon that is spanning the node is split, creating two new polygons which replace the original polygon in the list. Essentially, any polygon that spans a node plane, and is therefore causing us a problem, is split into smaller components that can be accurately described as belonging in either the back list or the front list of that node.

Without further ado, let us look at an example of building a polygon-aligned BSP node tree with the splitting logic introduced. With splitting in place, our BSP tree will be able to compile any geometry and render that geometry in a perfect back to front order from any camera position in real-time.

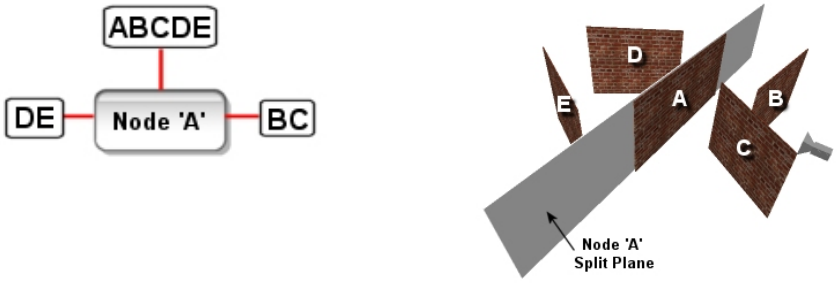


Figure 16.25

In this example we will use the same five polygons shown in the previous diagrams. For the sake of demonstration we will choose the split planes at each node in alphabetical order, but as discussed, the BSP tree will build and work correctly choosing the splitters in any order.

In the first step shown in Figure 16.25 the recursive build function is passed a root node and a list of five polygons labeled A through E. In this example, polygon A is chosen to be the splitter at the root node and is therefore removed from the list and stored in the node. In Figure 16.25 we have depicted the plane as the gray slab extending out from polygon A. As we can see, the root node now splits the space of the scene into two nearly equal sub-spaces. Polygon A is now stored at the node and the polygon list contains polygons B through E. In the next step, these polygons are classified against the plane of polygon A in order to construct front and back lists that will be passed down to the respective child nodes. If we look at Figure 16.25, we can see that D and E get assigned to the back list and polygons B and C get assigned to the front list. As we have non-empty front and back lists at the root node, it means we must create front and back child nodes and attach them to the root.

In this example we will recur into the front node first where the input list includes polygons B and C. A new split plane has to be chosen at the child node and, because we are going alphabetically, polygon B is selected from the list. The split plane is generated and stored in the node, along with polygon B. With polygon B now extracted from the list and stored in the node, the polygon list at node B now contains only one polygon: polygon C

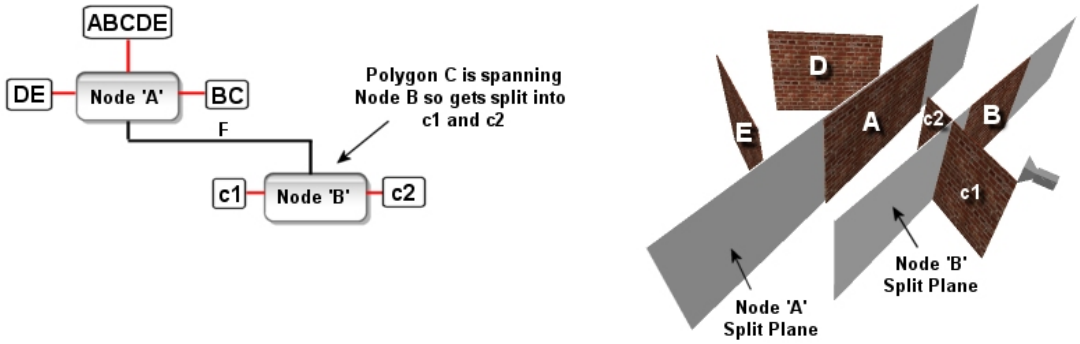


Figure 16.26

.

At this point, our job is to classify the remaining polygons in the list against the node's plane. In this instance, the polygon list contains just one polygon, so we would classify polygon C against the plane of Node B to determine whether it should be sent to the back or front of the node.

When polygon C is classified against the plane it is found to be spanning that plane and therefore polygon C must be broken into two child polygons. When we split a polygon against the plane of a node, the clipped fragment that is situated behind the plane is added to the node's back list and the fragment in the node's frontspace is added to the front list. After splitting polygon C into two child polygons (c1 and c2) and deleting the original polygon, we now have a front list and a back list compiled at node B. Each list contains a single polygon. The back list at this point would contain polygon c2 and the front list would contain polygon c1 (see Figure 16.26).

Because Node B still has polygons in its front and back lists, we will need to create both front and back child nodes. We then have to step into each of these children to continue the recursive building process. We will step down into the front child of Node B first.

When we enter the front child of Node B we are passed a polygon list that contains the single polygon c1. This final polygon is selected as the splitter at the new node, creating the Node c1 shown in Figure 16.27. At this point there are no more polygons in the list, so we return from Node c1.

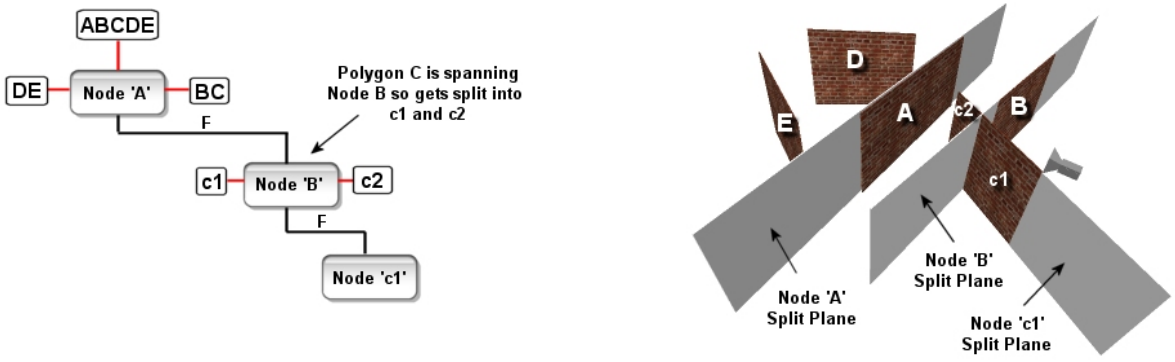


Figure 16.27

After we return from node c1 we find ourselves back at node B having built its front branch. As this node also has a polygon still in its back list (c2), we step into the back child of node B next and build its back tree.

As with node c1, when we step into the back child of B, we are passed a single polygon (c2), so the polygon to use as the split plane at this node is obvious. Polygon c2 is removed from the list and stored in the node labeled c2 in Figure 16.28

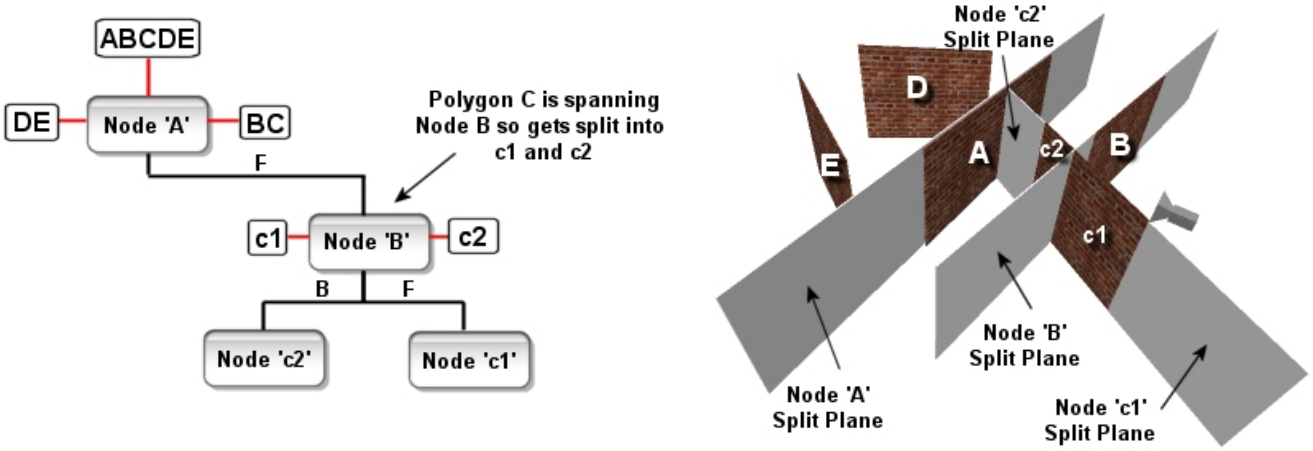


Figure 16.28

After storing the split plane and polygon at Node c2 we know we have reached a terminal node because there are no polygons remaining in the input list. In this case, the recursive procedure returns and program flow steps back up to Node B. At Node B we also realize that we have built its front and back sub-trees at this point, so there is no more work left to do at this node either. Thus, we return from Node B, popping up the tree and back to the root node A. At Node A we have only processed its front sub-tree, so it is now time to recur into its back child with the back list that was compiled at this node. The back list compiled for Node A contained polygons D and E, so these are the polygons that are passed into the newly created back child node of the root. When we step into the back child we are passed polygons D and E and one of them must be removed from the list and used as a splitter. In keeping with our alphabetical scheme, we will assume that polygon D gets selected and stored in the node as shown in Figure 16.29

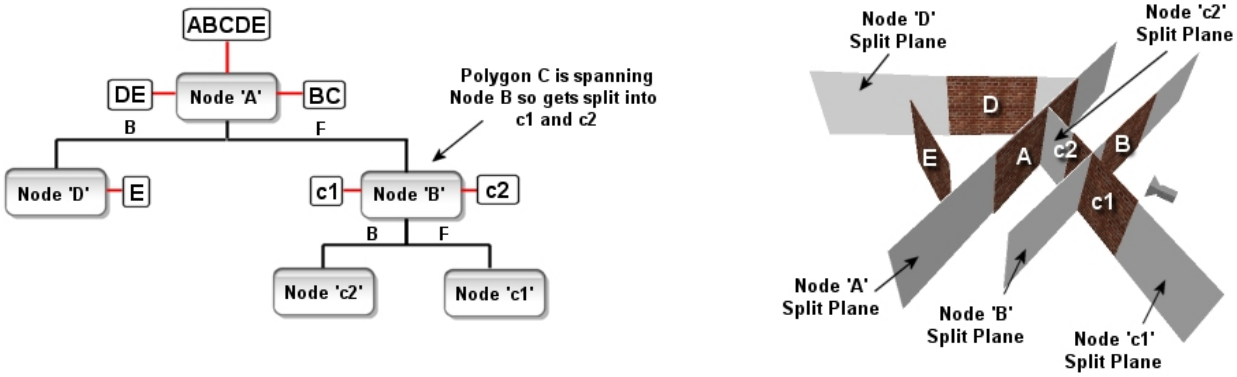


Figure 16.29

After the split plane for polygon D has been stored in the new node (called Node D), the remaining polygons in the list are classified into front and back lists. In this example, only polygon E remains in the list. Since it is found to be in front of Node D, it will be added to the front list. As there is no back

list compiled for Node E we do not have to create a back child node. However, there is still a single polygon in its front list, so we must create a new front child node and step into it.

In the front child of Node D we find only polygon E as its input data, so this becomes the splitter for the final node down this branch of the tree (see Figure 16.30).

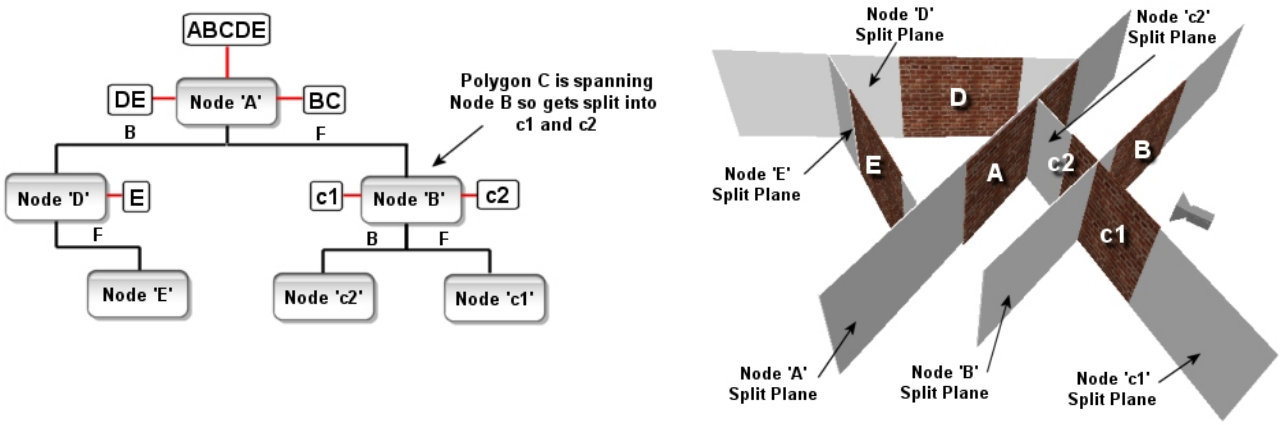


Figure 16.32

We return from Node E back to Node D where, after finding no back child has been created for this node, our task is done and we step back to Node A. When we unwind the call stack all the way back up the back branch of the tree and arrive back at the root node, we find that we have created the front and back trees of the root node and thus built the entire tree. At this point we return from the root node and the build process is complete.

16.2.5 An Overview of the Build Procedure

We have now built a BSP tree that has correctly split overlapping and intersecting polygons so that no rendering order ambiguity exists. It also worth noting that in our example the geometry set did not contain any polygons that shared planes. That is, we did not encounter a situation such that when classifying the polygon list against a node plane, we found a polygon that shares the plane with the node.

In such situations there are several things that can be done and they will all work with varying degrees of efficiency. Below we discuss possible strategies for dealing with the on plane case during the polygon/node classification step.

- You could test the normal of the polygon and add it to the front or back list depending on whether its normal is facing in the same direction as the node plane's normal or not. This would mean that each of the polygons that share the plane will have their own nodes created in the tree. Although this will work fine and still produce a perfectly valid tree, it does mean that we will have several nodes in the tree that essentially describe the same plane. As the BSP tree is being used to ascertain depth order based on view space plane distance, this redundancy is a little pointless. We know for example, that all polygons that share the same plane can be rendered

together since they cannot be occluding each other (they all describe the same depth slice of the scene).

- We could store any polygon that exists on the same plane as the node plane in the node itself, even if the normal of the polygon faces into the opposing halfspace. That way, even if 100 polygons spread throughout the level existed on the same plane (even if facing in opposing directions) only one node would be added to the tree and all 100 polygons would be stored at this node. The problem with this approach is that, because we now have multiple polygons stored at a node which may be facing in opposing directions, we must always render the polygons stored at that node even if the camera position is located in the plane's backspace. We mentioned earlier that during the rendering traversal we only render the data stored at the node if the camera is in front of the node and can thus potentially see the front side of the polygon stored there. However, with this technique, even if the camera is located in the backspace of a node plane, this does not mean that it will not contain polygons that are facing into the backspace. Therefore, we will have to render the polygon list stored there, whether the camera is behind or in front of the plane. This could mean that at a given node we render many back facing polygons that will be back face culled by the pipeline only after they have been needlessly transformed.

- The third option is to use a slight modification of the second strategy discussed above. That is, when we select a polygon whose plane is to become the node splitter, we find any remaining polygons in the list that share the same plane (even if facing in opposing directions) and store them at the node. Where this differs from the previous strategy is that the node structure itself will now have two polygon lists. One list will contain on plane polygons that face in the same direction as the plane and the other will contain the on plane polygons that were found to be facing into its backspace. With this technique, we get the best of both worlds with only a slight adjustment to our rendering strategy. Now, when the camera is located in the frontspace of the plane we only render the list at that node containing the polygons that face in the same direction as the node plane normal. If the camera is located in the backspace, we render only the list of polygons stored at the node that have opposing face normals. This technique is desirable because it means we create only one node in the tree for a given plane even if there are hundreds of polygons located on that plane. The result is a shallower tree that is more efficient to traverse. By storing many polygons at the nodes, we also have the ability to render them with a single call, speeding up rendering. Finally, as we have two lists of polygons stored at the node (of which only one is ever rendered depending on the camera's location with respect to the plane), we automatically perform back face culling with zero overhead.

- This final strategy is a mix of the ones described above and is the one we are going to implement in Lab Project 16.1's BSP tree compiler. We go back to the concept of a node storing just a single polygon list that is only rendered when the camera is in the node's front space. When a node is created and a polygon is removed from the list to create its plane, we will also remove any polygons that share the same plane and have same facing normals. We will store them at the node as discussed. Polygons that are found to be on plane with the node having opposing normals will not be stored at that node -- they will be added to the back list. This means, they will be used later to create additional nodes. That is, a node will contain a list of polygons that share the same plane and are facing in the same direction (the node's front space). These

polygons can all be rendered together when the node is visited and is found to have the camera in its front space.

One might wonder why we decided to opt for the fourth strategy instead of the third, when it seems to create more nodes. In truth, while both techniques provide good performance, our decision was slightly biased because the final strategy is the one that we ***must*** use during the building of a BSP leaf tree (discussed in the next section). Rather than having two different build strategies for the leaf and node trees, we felt that it would be better to unify the processing of the on plane case across BSP tree types. This will also help our transition into the BSP leaf tree discussion.

The following notes demonstrate how the BSP tree we build in Lab Project 16.1 will construct a node.

- Enter Node N.
- Select a polygon from the input list and store its plane at the node.
- For each polygon in the list.
    - Classify polygon against node plane
        - Front : Add to front list.
        - Back : Add to back list.
        - Spanning : Split polygon and add each fragment to front and back lists.
        - On Plane
            - Same facing normal : Store at node.
            - Opposing facing normal : Add to back list.
- If (Front list)
    - Create new front node and attach to current node.
    - Recur into front node passing in the front list of polygons.

- If (Back list)
    - Create new back node for the current node.
    - Recur into back node passing in back list of polygons.
- Return.

If you look at the CBSPNodeTree::Build function in the CBSPNodeTree.cpp source file that ships with Lab Project 16.1, you will see that these are the exact steps taken to construct each node. The source code to the entire class will be explained in the accompanying workbook

We now know how to compile a BSP node tree, and in the last build example we used a slightly more complex set of geometry which had intersecting polygons (or polygons with intersecting planes). The splitting process that occurs in the BSP tree build procedure resolved any ambiguity by clipping polygon C to the plane at Node B, thus creating two separate polygons which no longer span the plane. Let us now look at one final rendering example using the example BSP tree we have just built so that we are sure that we fully understand why the clipping of polygon C during the build process means that we can now calculate a perfect draw order.

Render Example 3

In this example we are using the tree that we used in the last build example. We start off by visiting the root node with the camera. To the left in Figure 16.33 we see our currently compiled BSP tree. The blue dot in this image represents the location of the node we are currently visiting as we step through the traversal. As you can see, the blue dot is currently at the root node since this is the first node we will visit with the camera. Although nothing has been rendered yet, the right hand side of Figure 16.33 shows the node planes that the tree represents and the way in which the space has been carved up. We can also see the location of our camera as currently being in front of nodes B and c1. All node planes in this example are assumed to be oriented in the general direction of the camera. It is this camera location we are now going to traverse the tree with to determine the correct draw order.

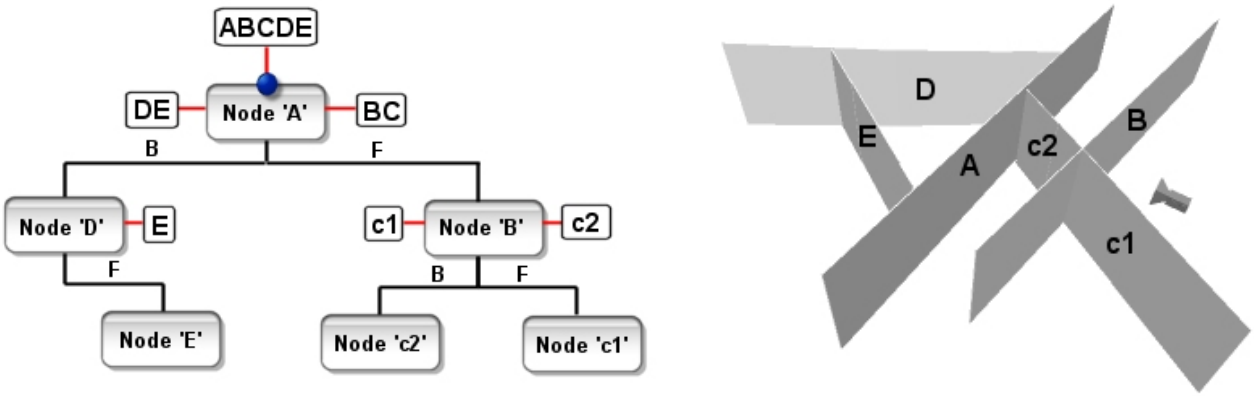


Figure 16.33

At the root node, the camera position is classified against node plane A and is found to be contained in its frontspace. This means we must step down the back of Node A first. When we step down the back of Node A we arrive at Node D (see Figure 16.34).

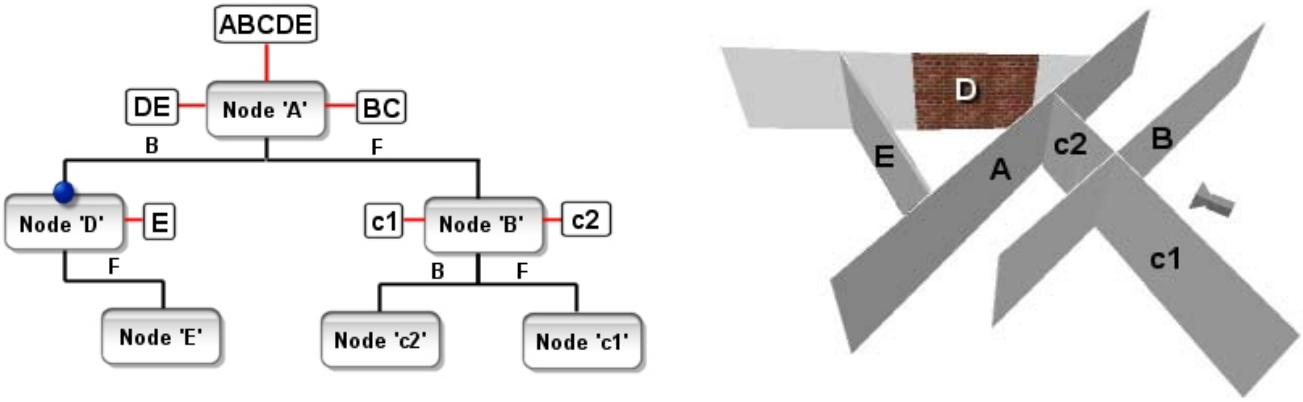


Figure 16.34

At Node D we find that our camera position is located in its frontspace, so we must render its back child first before we render the data stored at Node D. As Node D has no back child, we skip that step and just

render the polygon data stored there. This renders our first polygon so far (polygon D) as shown in Figure 16.34.

After rendering the data stored at Node D, we then step into its front child (E). Here we find a terminal node with no front or back children, so we simply render the polygon stored there, making polygon E the second polygon we render (see Figure 16.35).

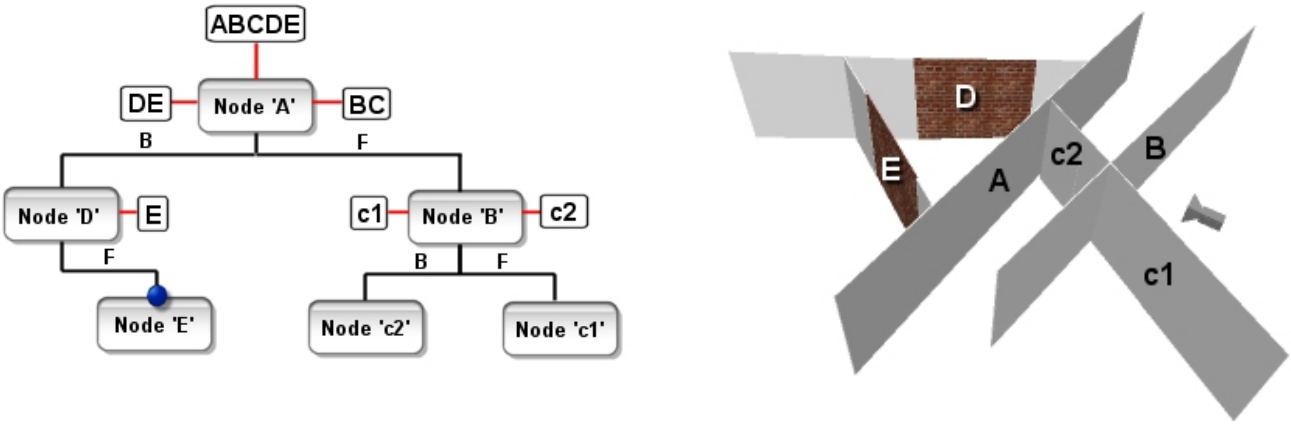


Figure 16.35

After rendering the polygon at Node E we pop back up to Node D and find that we have performed all the tasks necessary to process that node (having rendered its data and traversed its child list), so we return from that node and find ourselves all the way back up at the root node.

At the root node we have currently performed only one of the three tasks we must perform. As the camera was found to be in the node's frontspace we had to traverse into the back child first; we have now done that. The next thing we must do prior to stepping into the front list is render the polygon data stored at this node. This makes polygon A the third polygon to be rendered (see Figure 16.36). As we can see, so far, our scene is rendering in a perfect back to front order with respect to the camera position.

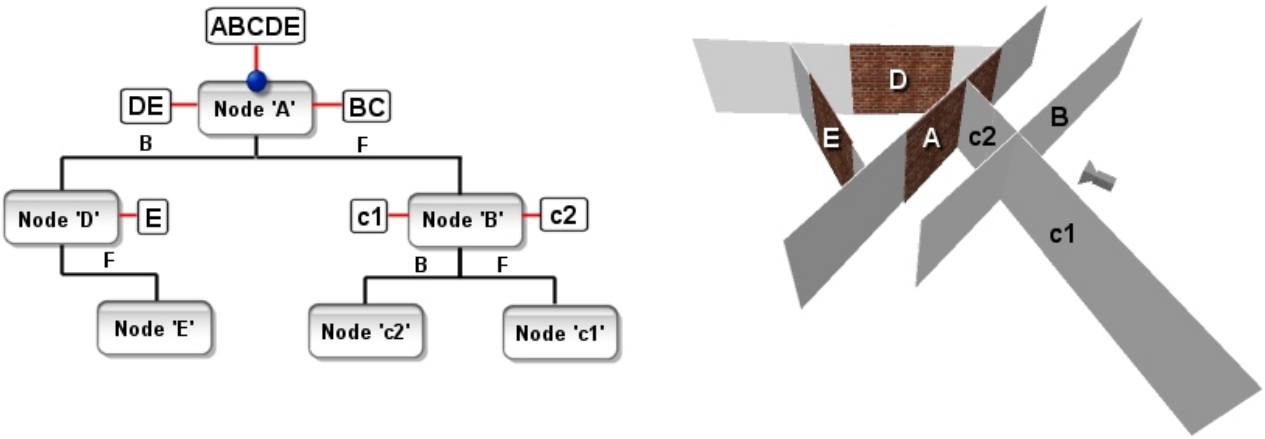


Figure 16.36

At this point we have rendered the data at Node A and should finally step into its front child where we arrive at Node B (see Figure 16.37).

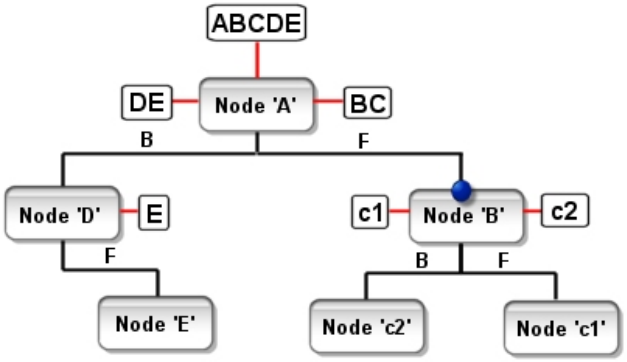


Figure 16.37

At Node B we classify the camera position against its plane and find that it is located in its frontspace. So we must step into its back child first, render the data stored at Node B, and then step into its front child. We step down into the back child of Node B first, where we arrive at Node c2. Node c2 is a terminal node, so there are no children to step into. As we now know, when this is the case we simply render the polygon data stored at the node (polygon c2 in this case).

As Figure 16.38 shows, polygon c2 becomes the fourth polygon we render, and we are still maintaining a perfect back to front draw order. This polygon did not exist in the original dataset sent into the tree; it was created from a larger polygon that originally spanned Node B. We can see now however, that polygon c2 can be perfectly described as being behind Node B.

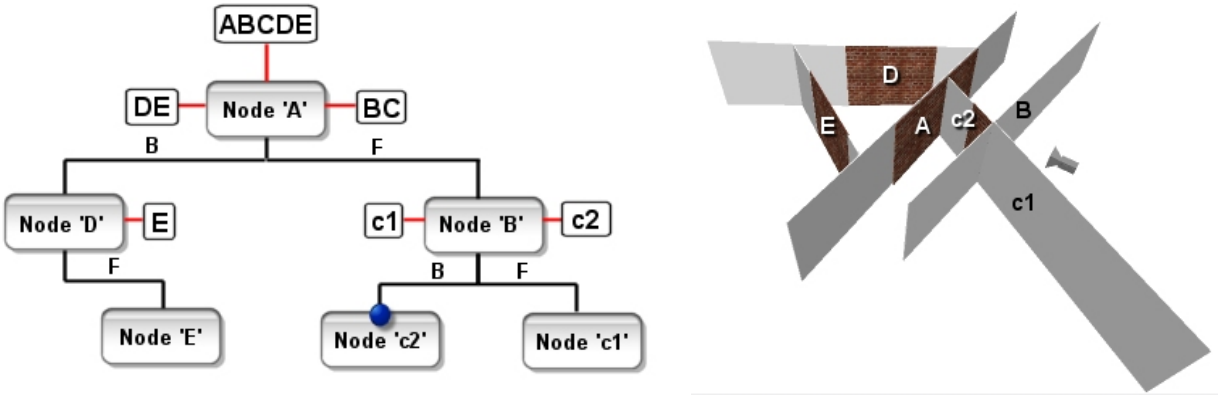


Figure 16.38

With the polygon data at node c2 rendered, we return and find ourselves back at Node B having processed its back child. Now we render the polygon data stored at Node B, which in this example is polygon B. Polygon B becomes the fifth polygon we render (see Figure 16.39).

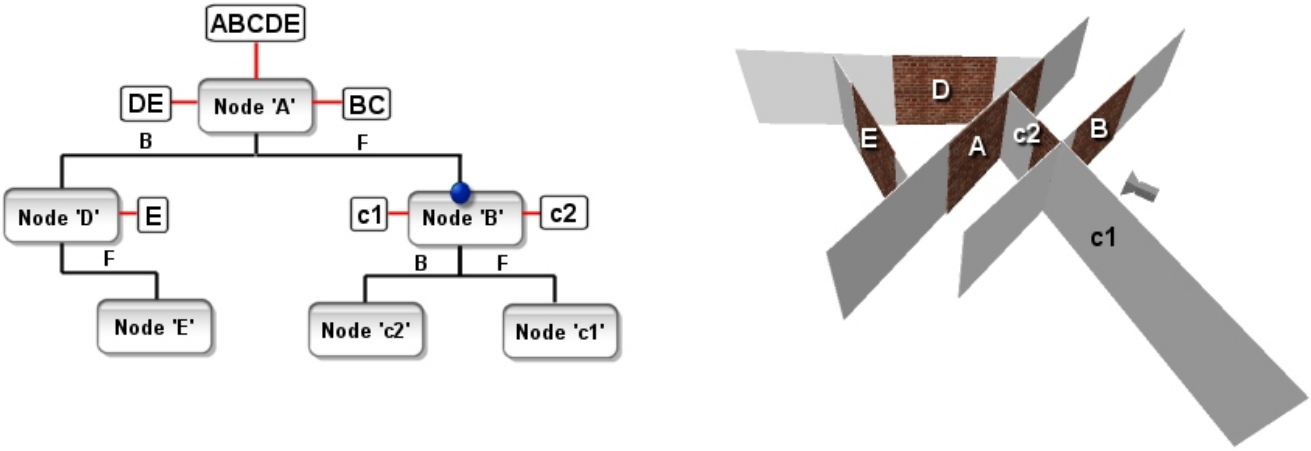


Figure 16.39

Having processed the back child of Node B and having rendered the data contained at Node B, our final task in the processing of Node B is to step into its front child where we arrive at Node c1. Node c1 is a terminal node, so we have to do nothing other than render the polygon data that is stored there. In this example that is polygon c1 (a child split of the original polygon C). This makes c1 the sixth and final polygon to be rendered, as shown in Figure 16.40.

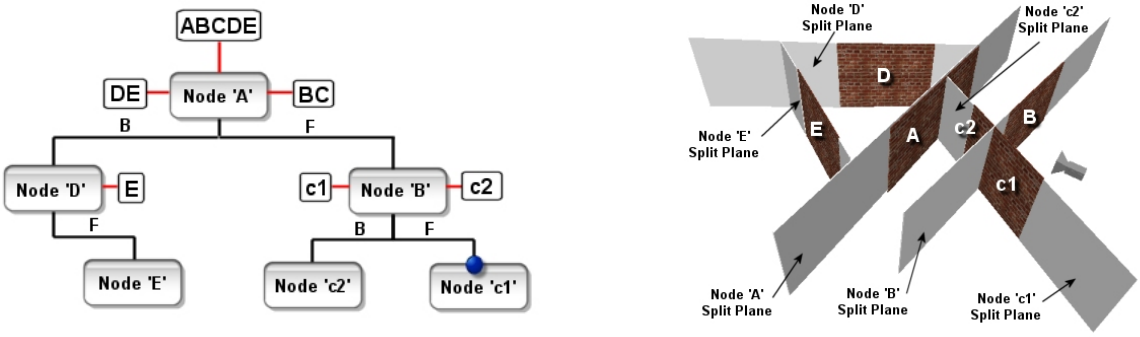


Figure 16.40

So by clipping the polygons to the node planes, we can generate a tree from an arbitrary polygon soup that can always be traversed in a perfect back to front order. This allows us to render polygons correctly such that nearer polygons will occlude more distant ones, even when we have no depth buffer support. While it is very unlikely given modern hardware that we would ever want to render our entire scene's geometry database using this technique, it certainly gives us a good solution for rendering our alpha polygons. This approach will ensure that blending always occurs in the correct order, even if multiple alpha polygons are layered on top of each other from the camera's perspective.

Since we will be building our tree such that on plane polygons that share the same normal direction as the split plane are stored at that node, the logic executed at each node during the rendering traversal is shown below:

- Enter node N
- Classify camera position again node plane
    - In Front or On Plane
        - Step into back child.
        - Render all polygons stored at node N
        - Step into front child
    - Behind or On Plane
        - Step into front child
        - Step into back child
- Return

Your attention is once again directed to the fact that if the camera is behind the current node, then the polygons stored at that node cannot be seen and are not rendered. If the camera position is in front of the plane, or situated on the plane, we render the polygon data.

One topic we have been deliberately steering clear from so far is the selection of a node's polygon as a split plane during the building process. While we can choose any polygon from the list passed into the node as the node's split plane and still get a BSP tree that works perfectly, there are some choices that are better than others when selecting a node's plane from a list of candidate polygons.

16.2.6 Choosing a Split Plane

We know already that our BSP compiler will recursively call itself with sets of polygons and that during iteration it will choose a polygon out of that list to become the splitter for the current node. All of the other polygons are then assigned to front or back lists, which themselves will be split, and so on until every polygon has eventually been chosen as a splitter. The last polygon left in a list will not actually split anything and will be assigned to the terminal node. The question is, during node construction how do we decide which polygon in the list should be used to create the split plane for that node? Is there some set of properties or some heuristic that we should consider to determine a preference?

While we could randomly pick a splitter each time from the list, there are actually some important distinctions that can be made between the candidates. That is, we can categorize polygons as either potentially good splitters or bad ones depending on certain criteria.

Every time we select a new polygon as a node's plane to divide a given subspace, any polygons in the list straddling the split plane (i.e., the splitter polygon's plane) will have to be split into two pieces. If you selected a splitter that intersected every other polygon in the list, all of them would have to be split into two pieces in order to be assigned to the front and back lists of the node. So on your first call to your compiler function you have just *doubled* the polygon count. This is definitely not a desirable outcome. Obviously the process continues as you traverse the new front and back lists and the polygon count (and therefore the node count) might quickly grow to a dangerous level with respect to

maintaining real-time rendering performance. It will also slow down the tree compilation time which, while not as onerous since it is an offline process, still impacts your development schedule.

While this example is a bit extreme, the point is that there are going to be some polygons that do not make great candidates as splitters early on in the tree creation process because they cause too many splits in the remaining dataset. So to build a more optimal BSP tree, we really want to consider candidate polygons that cause the fewest number of splits to occur in the remaining dataset.

Unfortunately, there is no way to build every potential BSP tree that would result from every potential splitter being accounted for at any given level in the tree. The number of permutations we would be talking about is a factorial based on the number of polygons (N!), so this approach is completely out of the question, even on a modern supercomputer. In case you were curious, or are unfamiliar with factorials, a tiny 50 polygon level would have $50! \approx 3x10^{64}$ possible BSP trees that would have to be built and examined for splitter preference.

So we can obviously forget about the possibility of building the perfect BSP tree, and instead settle for a very good one. One solution which gives good results is the following:

Each time we create a new node during the build process we will loop through each polygon in the list and process it. By 'processing' it, we mean that we will take its plane and test every other polygon in the list against it and record the number of polygons in the list that span that plane and would need to be split *if* the current candidate was used as the split plane. Whenever a polygon is processed which causes fewer splits than the previous minimum, we record the polygon and the split count. After every polygon in the list has been tested for a given node, we will have found the polygon in the list that will cause the fewest splits in the remaining dataset at that node and therefore is a great choice for the node's split plane. The process of polygon selection for a node's split plane is shown below. We might imagine that this is wrapped in a function called SelectBestSplitter which can be called by the node to select a splitter from its passed polygon list.

- Best Score = Infinitely High
- For each polygon in list A
    - Score = Splits = 0;
    - For each polygon in list B
        - Polygon B[*i*]->ClassifyPlane(Polygon A->Plane)
        - If (Spanning) Splits++
    - End for loop B
    - Score = Splits
    - if (Score < BestScore)
        - BestScore = Score;
        - pSelectedFace = Polygon A

- End for loop A
- Return pSelectedFace

As you can see, we first set the Best Score variable to a very high number. We will use this variable to search for the polygon that gets the lowest score (i.e., fewest splits) when tested as a split plane. We then loop through the polygons. The outer loop tracks the polygon being used as the split plane candidate. Inside this loop we set the current split count and score for this polygon to zero as we have not tested it against the other polygons yet. We then loop through every polygon in the list and classify it against the plane of polygon A. If polygon B spans polygon A then we know that choosing A as the splitter will cause this polygon (B) to be split, so we increase the split count. At the end of the inner loop we have tested every polygon (B) against the current split plane candidate (A) and have stored (in the variable Score) the total number of polygons that will get split immediately if this polygon is chosen as the split plane for the current node being constructed. If this score is lower than the best score we have recorded so far (for other polygons in the A loop) we record the score and the polygon which generated it. At this point we have stored the polygon which has generated the lowest number of splits so far. Once the outer loop has completed, every polygon in the list will have been tested as a split plane candidate and pSelectedFace will store the polygon that the current node should use as the splitter.

Although at first this might sound like the perfect tree compilation algorithm, keep in mind that choosing a splitter that causes very few splits at one level (node) of the tree could actually cause more splits further down the tree than might have been the case had we chosen a splitter which had initially split more polygons higher up in the tree. While we are content to work with this less than perfect approach to splitter selection, there is one other concept which we can factor into our choice of splitter, which is not new to us.

Although choosing a polygon that creates the fewest splits is arguably the most important criteria in splitter selection, there is also tree balance to consider. A perfectly balanced BSP tree (which is almost impossible to create without excessive splitting) is one with exactly the same number of nodes in both the back and front branches of the root (i.e., an even distribution of nodes). We discussed in the previous lessons the benefit of having a balanced tree and how it helps to achieve shallower trees and maintain consistent traversal times. Sometimes frame rate consistency is more important than pure speed. For example it is better to have a game engine run at 30fps consistently throughout the level than have it run at 90fps in some places and drop to 7fps in others. This is where tree balance becomes a factor.

Unfortunately, the downside to balancing a BSP tree is often the creation of more splits in the list. So we are going to have to find some middle ground here and adjust our split plane selection logic to also factor in the tree imbalance that will be introduced by choosing a particular polygon. As our split plane selection logic essentially has to classify every polygon against the candidate plane to record the number of splits, it is a small step to also record the number of polygons that are found to be completely in front or completely behind the candidate plane. If a candidate plane has the same number of polygons in its front list as in its back list, it means that the node splits its space in a perfectly balanced manner. The larger the disparity between the front and back face counts, the more imbalanced the node (and most likely the entire tree) will be. Therefore, we might imagine that the score for a given candidate polygon

would be the sum of the number of splits it creates and the absolute difference between the number of polygons found to be contained in both its halfspaces. This would make our polygon score calculation:

Score = abs(frontfaces - backfaces) + splits

So we will subtract the number of back faces from the number of front faces and take the absolute value. We then add to that the number of splits, giving us a score for the polygon that will be at its lowest when the polygon creates a perfectly balanced partition (frontfaces=backfaces) and causes no splits.

We are almost there, except that at the moment we are probably not giving enough influence to the number of splits. In most cases, reducing the number of splits will be our primary goal, so we will need a way of letting the selection process know that we would like to place more weight on the split count than on the imbalance score. Therefore, our SelectBestSplitter function will also take a split heuristic (a weight value) parameter which is multiplied by the split count to create the score for each candidate plane:

Score = abs(frontfaces - backfaces) + (splits * SplitHeuristic)

As you can see we now multiply the split count by the passed split heuristic so that we can give more or less priority to reducing splits by passing a higher or lower split heuristic value respectively.

Our new version of the SelectBestSplitter function will loop through the list of polygons passed in, choose a different candidate split plane each time and then test it against the rest of the polygons in the list. Each time a splitter has been considered, we will end up with the number of splits it caused and the number of front and back polygons that would end up in the front and back lists of the respective node. We can then calculate our score using the above formula. If the score is lower than any previous score we encountered during this call then we will keep track of this polygon. When the function ends, it will return a pointer to the polygon in the list with the lowest score. This polygon can then be used by the node to create its split plane.

- Parameter = “Split Heuristic” (single weight value)
- Best Score = Infinitely High
- For each polygon in list ‘A’
    - Score = Splits = Back Faces = Front Faces = 0;
    - For each polygon in list ‘B’
        - Polygon ‘B’->ClassifyPlane(Polygon A->Plane)
        - In Front :
            - Front Faces ++
        - Behind :
            - Back Faces ++
        - Spanning :
            - Splits++
    - End for loop ‘B’

    - Score = abs(Front Faces + Back Faces) + (Splits*Split Heuristic)

    - if (Score < BestScore)
        - BestScore = Score;
        - pSelectedFace = Polygon A;

- End for loop A

- Return pSelected

The logic to the code shown above is contained inside the CBSPNodeTree::SelectBestSplitter method in Lab Project 16.1. We will cover the actual source code to the BSP compiler to Lab Project 16.1 in the accompanying workbook.

16.2.7 BSP Node Trees Conclusion

We have now completed our introduction to the standard BSP node tree, using polygon-aligned planes to carve the geometry into pieces that can be rendered in a perfect back to front order. Although the node tree is the simplest of the BSP tree incarnations, you will be pleased to know that the leaf tree we develop in the next section is essentially exactly the same tree. The only real difference is some extra logic introduced during the build phase to collect convex clumps of polygons at the terminal nodes of the tree.

Covering the BSP node tree first was a good way to ease ourselves into the subject matter that we will need to explore both in this lesson and in the following lesson. However, let us not write off the BSP node compiler as simply an academic exercise with no real world application. Indeed, as we have discovered, even on today's latest cutting edge hardware, alpha polygons must be rendered in a back to front order to maintain proper blending. The BSP node tree is now going to be our tool of choice for handling alpha polygons. There are many other areas where BSP node trees can be used, such as in the realm of texture consolidation (efficiently packing multiple images onto a single texture surface). Also keep in mind that many applications it has in the area of geometry repair and constructive solid geometry.

Before moving on to the next BSP tree type, now would be a good time to open up the source code for Lab Project 16.1 and examine (along with the accompanying workbook) the node based BSP tree compiler.

16.3 Introducing BSP Leaf Trees

In this section we will examine a slight variation of the BSP tree building technique which will allow us to introduce leaves into our BSP tree. The BSP leaf tree is actually a lot more like the trees we have built in previous lessons because the polygons are not stored at the individual nodes but are passed down the tree during the build process and collected at the terminal nodes (i.e., the leaf nodes) of the tree. This is much more akin to the strategy used to build the quad-tree, oct-tree and kD-tree in the previous lessons. Although it may seem like we would now have to examine another form of tree, the good news is that this is not the case. Whether we build a leaf tree or a node tree, the tree itself is constructed in exactly the same way with respect to how the node planes are selected and the way that the space of the scene is subdivided. In fact, one might say that our node tree already has leaves; we are just not using them for anything yet. Closer examination of the BSP node tree that we constructed in the previous section will highlight the fact that although we decided to store the polygons in the nodes of the tree, the planes of the BSP tree still carve the scene into convex areas which can be navigated to by traversing the tree to the terminal nodes. Unlike the other leaf trees we have constructed however (quad-tree, oct-tree and kD-tree), the leaves of the BSP are not axis aligned bounding boxes, but are arbitrary convex regions bounded by the node planes which intersect to form that region. Yet just like any of our previous tree types, we can send objects or polygons down the tree and assign them to the terminal nodes in which they are eventually found to reside. We will see that these terminal nodes represent a convex region of the scene and what is considered to be a 'leaf'.

This all sounds a little abstract at the moment, so let us take the node tree that we looked at in the previous section and examine what would be involved in turning it into a leaf tree. Figure 16.41 shows the BSP node tree we constructed in the previous section.

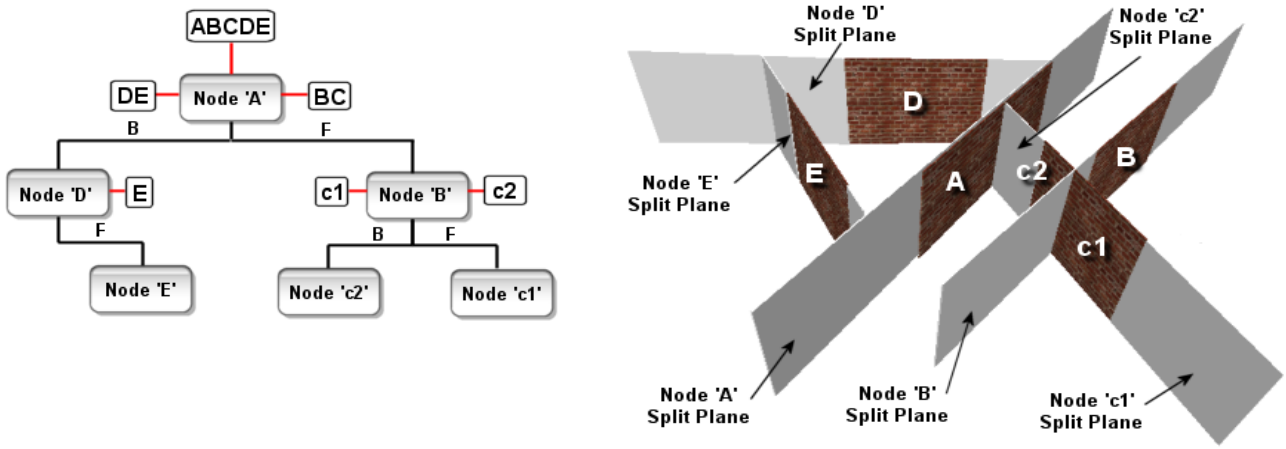


Figure 16.41

The node tree is referred to by its name simply because the polygons are stored at the nodes. Recall that this was done to solve a very specific problem related to alpha sorting. By realizing that the planes of the tree could be traversed in a back to front order and that the planes themselves were created from the polygons, every polygon in the scene will lay on one of these planes. It is logical to assume then, that we

can render the polygons in a perfect back to front order by storing them at the nodes they helped create and render them when those nodes are visited during an ordered walk of the tree. This is obviously a very useful strategy to use when perfect back to front order rendering is required. However, it would be extremely inefficient to also store our non-alpha polygons in this way due to the fact that we essentially process one polygon at a time. It would be much nicer to have a BSP tree at our disposal which works much like the quad-tree for example, where the polygons are collected at the leaf nodes and can be rendered in a batch when the leaf is found to be visible. Furthermore, although we already have trees at our disposal that allow us to do just this, you will see later that the leaf BSP tree will become one of the most important technologies we will implement in this course as it will allow us to construct potential visibility sets, do hidden surface removal, and perform a host of other really useful and essential tasks. Therefore, while the BSP leaf tree may seem like just another leaf tree (which it is), we will see later that because the polygon data is used to construct the node split planes, the tree is built with very important information about which areas of the scene are considered to be empty space and which areas are considered to be solid space. This is the only spatial tree we have covered so far that provides us with this information. This is precisely the information we will need in the following lesson to build a potential visibility set compiler for our scene data and accelerate rendering by an order of magnitude.

To understand where the leaves are located in a BSP tree and what they represent we will first highlight the similarities between the BSP tree and the other tree types. For example, we know that we can build an empty quad-tree by simply dividing empty space and assigning no polygon data to it. The result is a tree of empty bounding boxes. Such a technique can be useful if a game uses purely dynamic objects and would like to benefit from hierarchical spatial partitioning. The quad-tree starts with a large root node bounding box and subdivides that box down to a certain level. When the tree has finally been constructed, there are no polygons stored in the tree, but the leaves are still there -- they are just empty. We still have the ability to traverse the tree and find visible leaves and we will still have the ability to send volumes down the tree and assign them to the leaf nodes in which they are found to reside. The same is essentially true with our BSP node tree. That is, we have subdivided the scene into convex areas using a series of planes and as such, navigating to the terminal nodes allows us to describe ourselves as being in one of those areas. In the case of the node tree, we are just choosing to store nothing at the terminal nodes that represent those areas. Therefore, although the node tree did not store any polygon data in the leaves of the tree, the leaves still exist and are implied by the nature of spatial partitioning.

Let us imagine that we decide to build the same node tree illustrated in Figure 16.41 but decided not to store the polygons at the nodes. We will simply discard them after they have been used to create a node plane (along with any co-planar/same facing polygons). That is, after a polygon has been selected as a splitter and used to create a node plane, that polygon, along with any others that are co-planar and same facing, are deleted from the list and are never considered again during tree creation. The result would be an empty BSP tree just like that empty quad-tree we just discussed. The polygon data passed into the tree in this example was simply used to determine which split plane to use at each node. In fact, one can imagine how the BSP compiler could even be modified **not** to take a list of polygons but instead a list of planes in such a scenario. The resulting tree would be the same (see Figure 16.42). The only difference now is that we have not stored the polygons at the nodes. We have an empty BSP tree just like that empty quad-tree we discussed above, just carving up space using a different set of criteria.

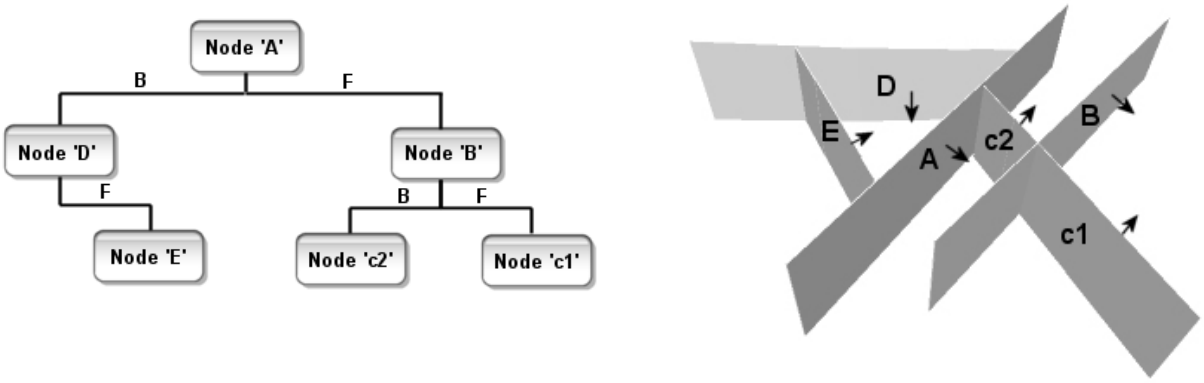


Figure 16.42

In Figure 16.42 the plane normals are shown as the black arrows in the rightmost image so that we can see the direction in which the planes are facing. When looking at the rightmost image and remembering that planes are infinite we can see that the planes divide the scene up into a series of convex areas.

> **Note:** Although planes are technically mathematically infinite, when drawing the planes of a BSP tree, the planes only carve up to the parent node space. This is because the plane is only used to carve up the designated halfspace of the parent node. Node D for example is assumed to carry on infinitely to the right of the image but stops at node A because it was selected to only partition node A's back space.

We will examine why using the polygon planes as the node planes of the tree divides the scene into a series of convex areas a little later, but for now just know that this is the case. For example, we can see that planes A, D, and E bound a triangular region of the scene. This region exists behind node A, in front of node D and in front of node E. Therefore, we can see from the tree diagram on the left hand side of this image, that if an object is found to be in front of node E during a tree traversal, it must be in this area and therefore, this area is the front leaf of node E. What structure we use to represent this area is not important for the moment (we might imagine that we use a structure that can contain a list of polygons and dynamic objects that have been passed through the tree and found to exist in that area). Furthermore, we can also see that if an object was found to be in front of node c2, it is obviously in the area of space bounded by planes A, c2, and B labeled 'leaf 3' in Figure 16.43 since node planes A and B were also navigated to reach node c2.

The conclusion we can draw from this discussion is that every terminal node in our tree is a plane which has a region down both its front and back sides. That is, each terminal node has a back leaf and a front leaf which represents one of the areas making up the scene. In fact, a leaf exists wherever a node has no front or back child. In this instance, we are referring to a leaf as a region of the scene that is bounded by the planes that have been traversed to reach that area of the tree. In Figure 16.43 we have color coded

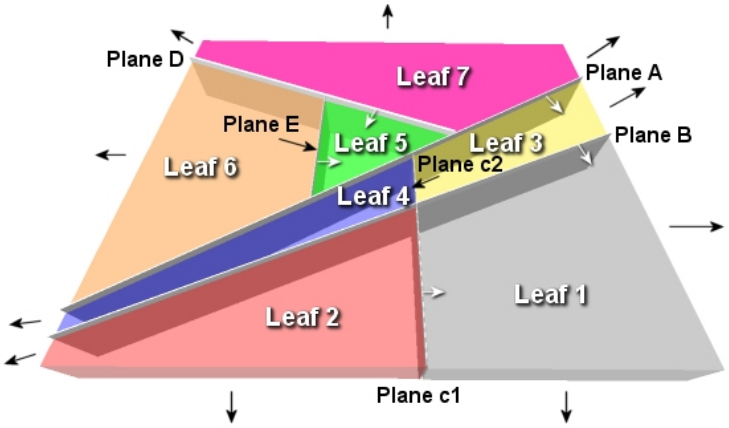
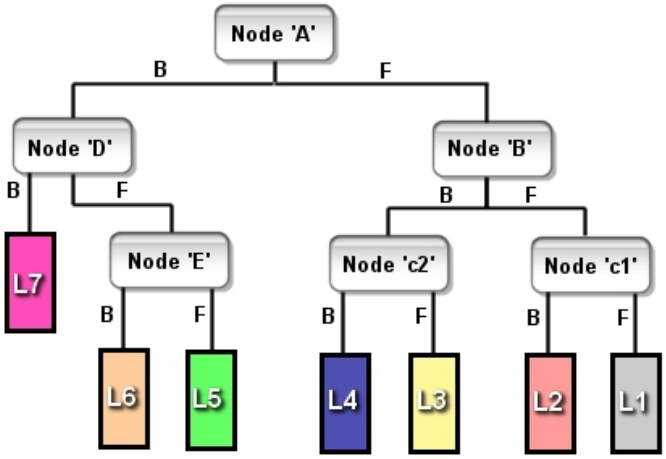


Figure 16.43

the convex areas of the scene and have also illustrated in the accompanying tree diagram where these areas exist.

Looking at the tree diagram for Figure 16.43 we can see that we have added leaves to the tree where a node has no front and back child node. To be clear, these leaves (labeled L1 to L7) are all empty and are labeled here to illustrate which area of the scene is reached by stepping down the front or back of a childless node. The boxes in the tree diagram might represent a leaf structure where polygons and dynamic objects are stored. The important point being made here is simply that whenever we step down the side of a node and it has no child node, we are in a leaf. This is a convex area described and bounded by the planes above it in the tree.



> **Note:** The leaves L1 through L7 could be represented as simple structures used to contain any objects or polygons that get assigned to that area. The important point is that we recognize this is really no different in principle from the node tree. The tree is exactly the same and the regions represented by these leaves always existed, the only difference now is that we are assuming the use of a leaf structure to catch and store any objects that get passed down the tree and eventually end up getting passed down the front or back of a node for which no child node exists.

Study the diagram and try out some test traversals of your own to see if you understand why the leaves are located in the tree in the locations that they are. Do not worry if you are still finding this a little confusing, we will step through some examples.

Figure 16.43 shows us that the BSP tree carves the scene up into regions that we call leaves and as such, is not unlike the quad-tree, oct-tree and kD-trees that we built in previous lessons. There are two big differences: First, a leaf is no longer an axis aligned bounding box, but is defined by the planes that bound the region (the planes above it in the tree). Second, not all leaves are fully surrounded by planes and as such, are assumed to carry in infinitely in the unbounded direction (this is only true for exterior leaves). In our simple example scene, all leaves except leaf 5 are exterior leaves, as they exist around the exterior of the scene. For example, we can determine the exact volume of leaf 5 since it is fully bounded by planes A, D and E. These are the parent planes of the leaf that need to be traversed in order to make it into that region. However, leaf 6 which is located behind node E is only bounded by planes on three sides and is therefore assumed to have infinite volume to the right of the image. We can also see that leaf 1 would also continue infinitely down and to the right of the image as the leaf is only bounded by planes on its left and top sides.

Let us now use some example locations which we will send through the tree and test that our theory is correct. In Figure 16.44 we have placed a red sphere in a region of the scene that visually places it in leaf 3. That is, it is located behind plane B and in front of planes A and c2. We might imagine that this red sphere represents the position of our camera in the scene and we would like to know in which leaf within the BSP tree it is currently located so that we can render any dynamic objects that have also been assigned to that leaf (the structure attached to the front of node c2). Finding what leaf a given position vector is in is no different from the kD-tree case. We simply send it down the tree starting at the root node and classify it against each node plane we visit, sending it down the front or back of the node depending on the result. Figure 16.45 shows us sending the query position shown in Figure 16.44 through our BSP tree.

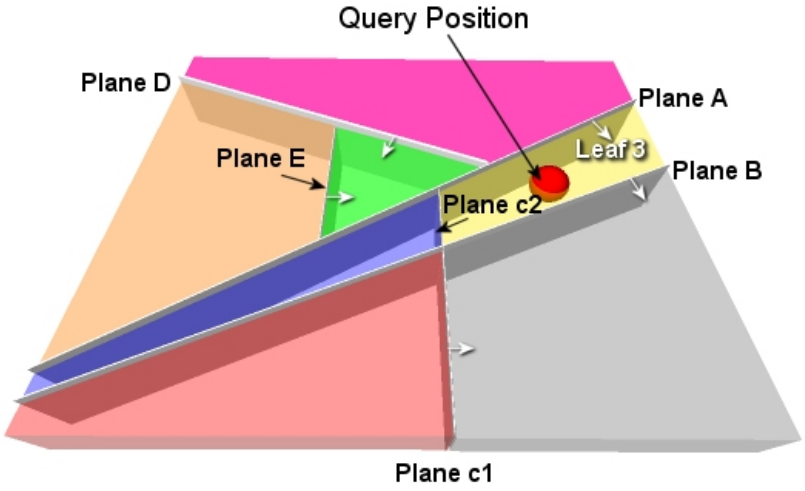


Figure 16.44

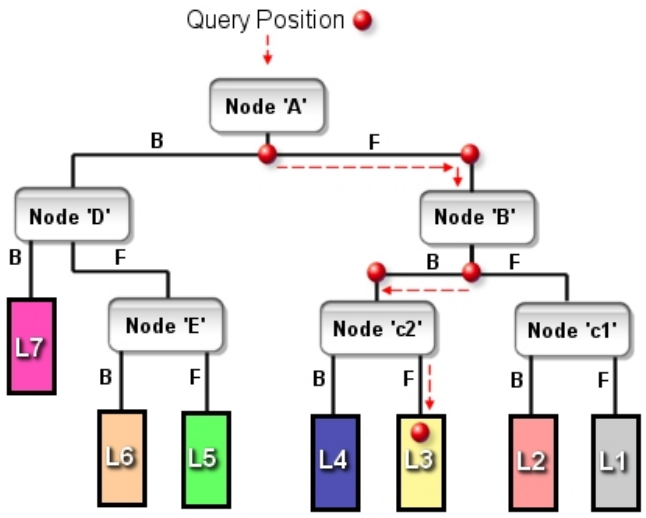
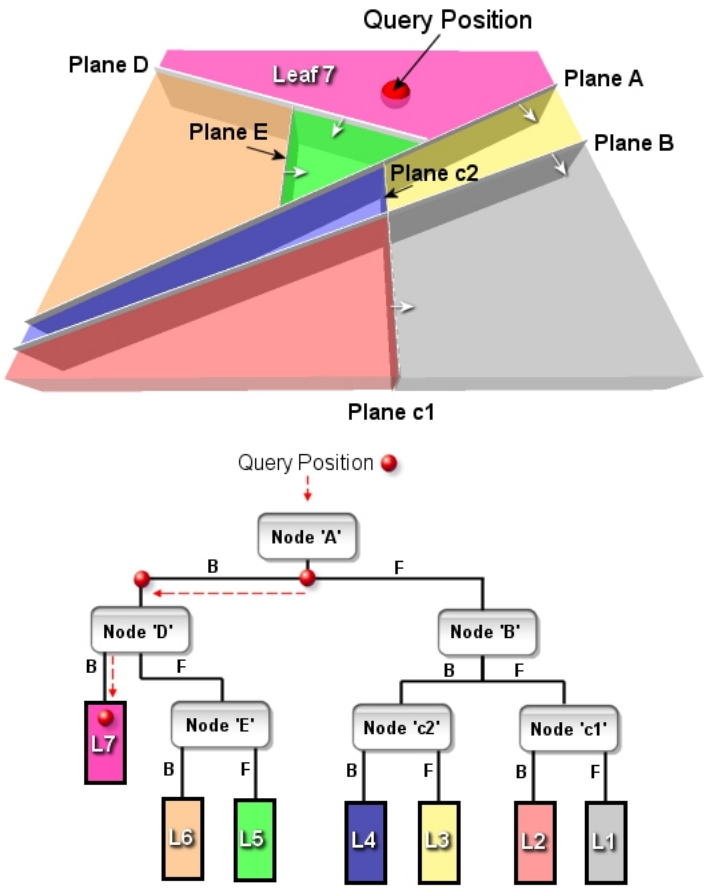


Figure 16.45

In Figure 16.45 we show the position in Figure 16.44 being passed through the tree. You should reference both of these images during this next examination of the traversal process.

At node A we classify the query position against the node and find that it is in front of plane A (as can be seen in Figure 16.44), therefore, it is passed down the front of A where it reaches node B. At node B the query position is found to be behind the plane stored there and is passed down the back of node B where it enters node c2. At node c2 the classification is performed once again and this time the query position is found to be located in the front halfspace of the terminal node c2 which places it in leaf 3, the region of space bounded by planes A, B, and c2. Leaf L3 might be a structure that contains all the polygon data that has been passed down the tree and clipped to the nodes and found to reside in c2's front halfspace. These polygons would also need to be rendered if leaf 3 was found to exist inside the view frustum.

In Figure 16.46 we see another example of traversing the BSP tree with a query position and further solidify our understanding of the regions of space represented by the leaves of the BSP tree.



The query position is shown in Figure 16.46 to exist in leaf 7. Looking at the topmost image we can see that this is a region of space that is bounded by planes A and D. It is located behind both of these planes. Thus, if we know that a given position is located behind planes A and plane D then it must be in the region of space we have labeled leaf 7. We can see that this is the case in the topmost image and furthermore, can see in the bottommost image that when we traverse the BSP tree with this query position, these are exactly the tests that are performed. The query position is fed in at the root node where it is found to be located behind node plane A. Because of this, it is passed down the back tree of A where it reaches node D. The query position is once again classified against node D where it is found to lay behind that node also. As node D has no back child we know that this is the end of the road and we have reached the convex area that we have labeled leaf 7. Finally, we will show one more example where the query position is located in leaf 5 accompanied by the tree diagram that shows the query position being passed down the tree. Eventually we will pop out in the front space of node E.

Figure 16.46

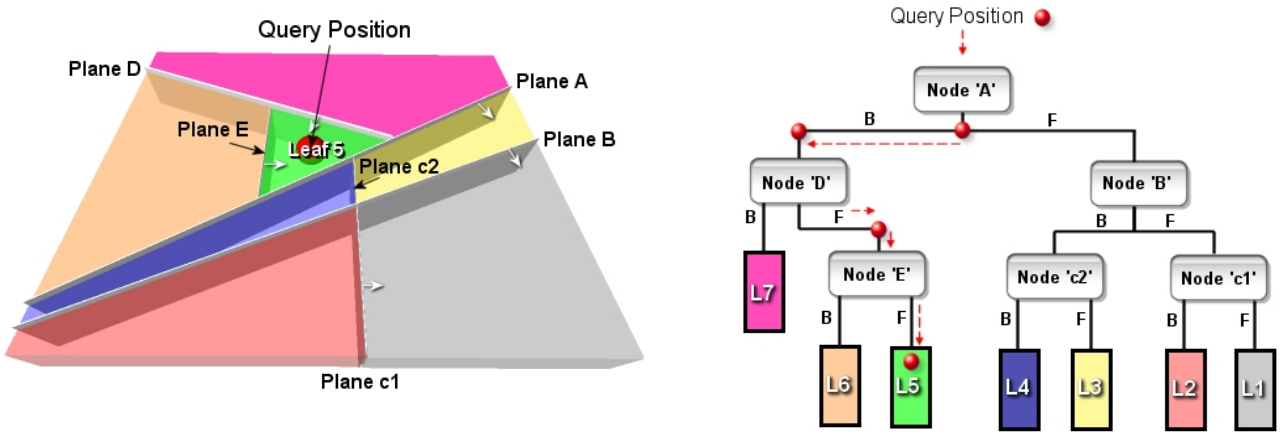


Figure 16.47

When we look at the rightmost image in Figure 16.47 we can see that if we were to describe this position in English we would say that it is in the region of space behind plane A and in front of planes E and D. However, notice that in the rightmost image this is exactly what we are doing when we traverse the BSP tree. We are testing the query position against these planes of the tree and eventually find that our query position is located in front of node E, in front of node D and behind node A.

16.3.1 Populating the BSP Leaf Tree

Now that we understand where the leaves of the BSP tree exist and what areas they represent, it is the next logical step to determine how we would compile the leaf tree and collect the polygon data at the leaves of the tree instead of storing them at the nodes. Before we look at how to modify the BSP build process to collect the polygons at the leaf nodes during the compile process, we will first perform the polygon population of the tree in a separate pass. This will give us the opportunity to become more familiar with the rules of passing polygons down the BSP tree before we merge this polygon passing and collection logic into the core build process. Therefore, in these next examples we will assume that the BSP tree was built as in our previous example and is an empty tree initially. That is, during tree construction, after a polygon was used as a splitter, it was simply deleted from the list and considered no further. As we saw in the previous section, this creates a tree of separating planes only. Once we have this tree we will then send the polygons that were used to create the tree into the root node one at a time and will track their progress as they descend through the tree into the leaf nodes in which they belong. This will demonstrate the polygon passing logic that we will eventually merge into the core build process.

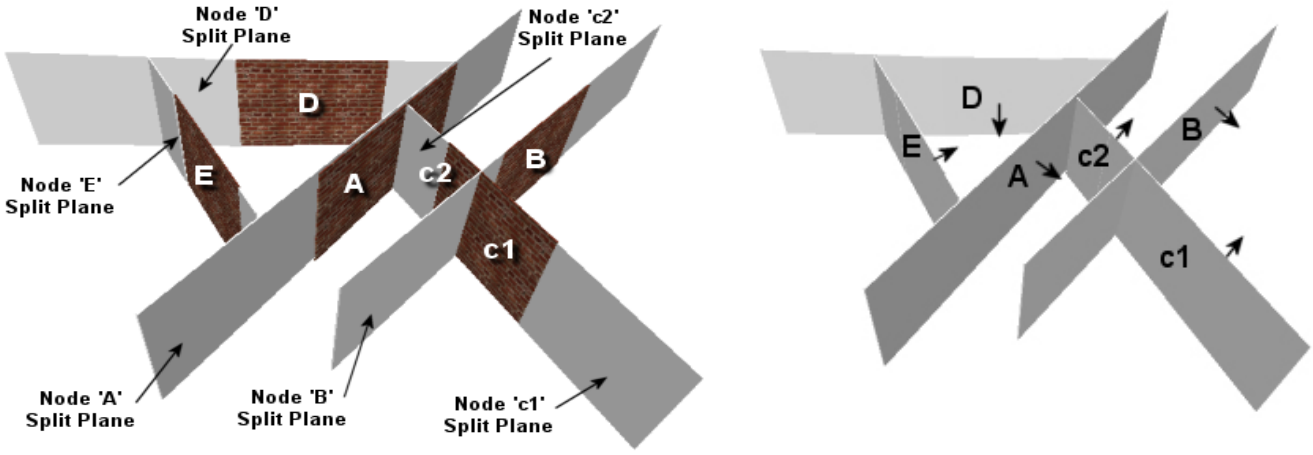


Figure 16.48

The leftmost image shows the polygons that were initially passed into the tree and also shows the node plane created by each. The rightmost image shows the BSP tree after compilation with no polygon data assigned to it. As you can see however, the polygon data was still used to determine the node planes to be used.

Assuming that our tree has been constructed as shown above, what we will now do in a second pass is loop through polygons A through E and send each one down the tree starting at the root node. At each

node, the polygon will be classified against the plane and sent down the front or back child depending on the classification result. If a polygon is found to be spanning a node plane then it will be split by that plane and the two child fragments dispatched down the front and back of that node respectively. If the polygon is found to lie on the plane and face in the same direction as the node plane it will be passed down the front of the node, if it is co-planar but faces in the opposite direction to the node plane it will be passed down the back of the node.

> **Note**: Later in this lesson we will learn why dealing with the co-planar classification case as described above is so important. With the leaf tree, we must pass co-planar same-facing polygons down the front of a node and co-planar opposite-facing polygons down the back of a node. Take this on face value for the time being as the reason for this will not become clear until we wish to exploit the solid/empty space determination properties of the tree. However, if we do not handle co-planar polygons in this exact way, we will lose the ability to query the solid/empty space properties of the geometry stored within the tree.

Figure 16.49 shows us that polygon A will be the first to be passed down the tree. Figure 16.50 shows the journey of this polygon through the tree where it eventually gets clipped into two fragments that exist in leaves 3 and 4 respectively. The first plane it is tested against is node A. As it is co-planar with this plane and pointing into the same frontspace, it is passed down the front to node B. It is found to lie behind node B so is passed down the back to node c2. Polygon A spans node c2 so is split into polygon fragments a1 and a2 and the original polygon A is deleted. Polygons a1 and a2 are passed down the front and back of node c2 respectively. The two fragments end up in leaves 3 and 4.

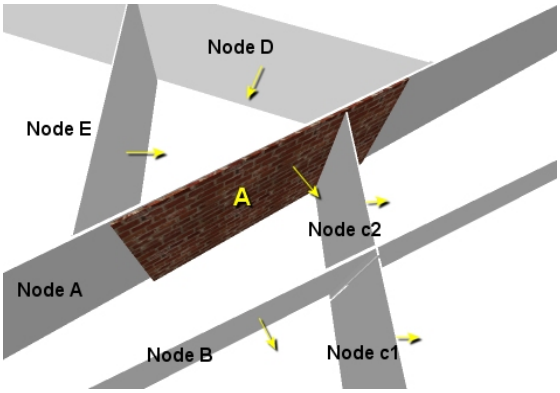
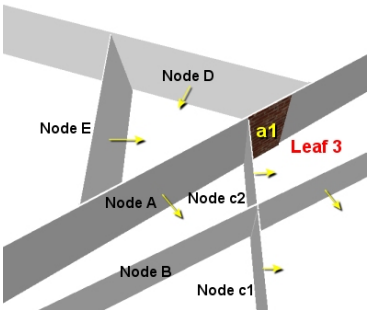
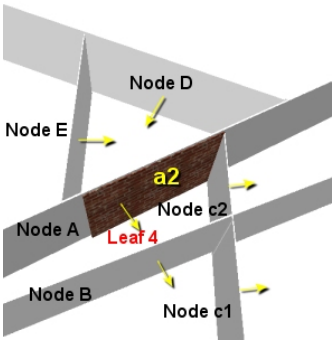


Figure 16.49





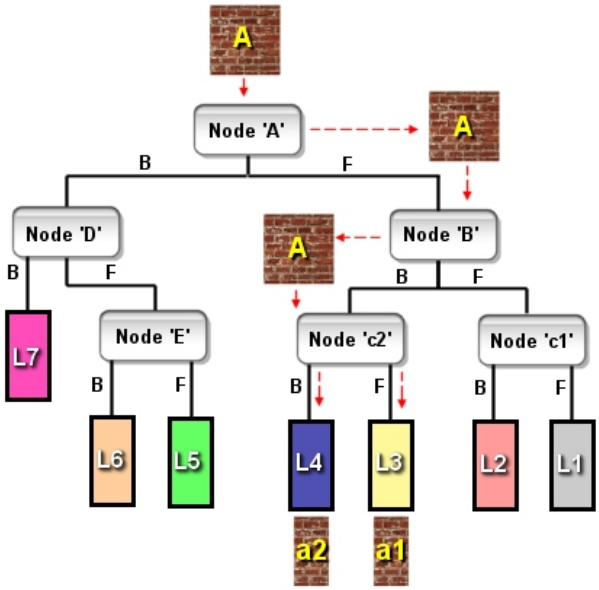


Figure 16.50

At this point then, we have two polygons stored in our tree, a1 and a2, which were both originally created from polygon A. These polygons will have been added to leaves L3 and L4. Next we send polygon B through the tree. Its journey is shown in Figure 16.51 where we see that it ends up in leaf 1.

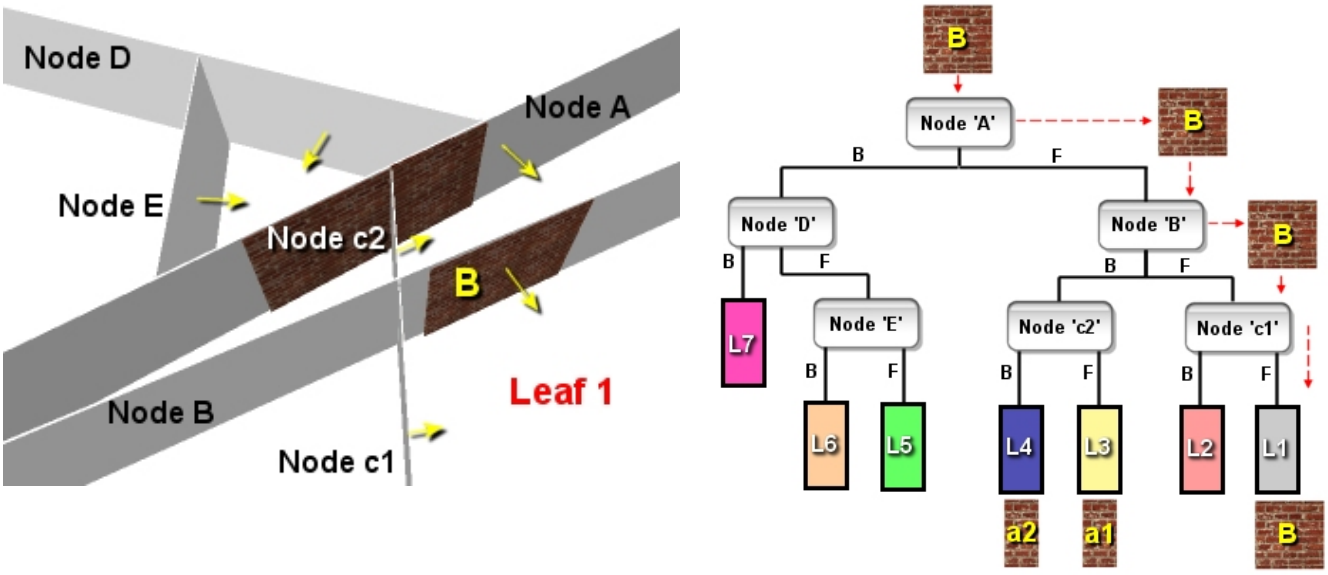


Figure 16.51

We would like to draw your attention to a very important trait of the BSP tree which we are already starting to see manifest itself, even with the two polygons that we have added to the tree thus far. In the quad-tree, the oct-tree and kD-tree, although the leaf nodes represented convex regions within the scene (in these instances the region was an axis aligned bounding box), the polygons were assigned to these regions as a soup. That is, the leaf nodes contained a soup of polygons which filled up the space of the leaf node to some extent. However, this is not the case with the BSP tree.

Because the polygons themselves are used to create the split planes we know for a fact that every polygon in the tree will lay on one of those node planes. As we also know that the node planes always form the boundary of one or more leaves, it stands to reason that the polygons assigned to a leaf will actually bound the region represented by that leaf and will not be contained in the middle of the leaf's space. We can clearly see that this is the case in all of our diagrams to date. In Figure 16.51 for example, we can see that although polygon B was assigned to leaf 1 (the leaf it faces into), it is located on the boundary plane of that leaf. This is a very important point to bear in mind as we go forward.

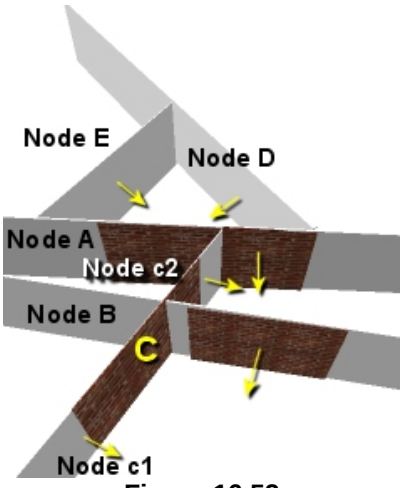


Figure 16.52

In the next example we will feed polygon C into the BSP tree. The original polygon C is shown in Figure 16.52 and we can already see that it will be split into two fragments and will therefore live in two leaves. Figure 16.53 shows the polygon's journey through the BSP tree.

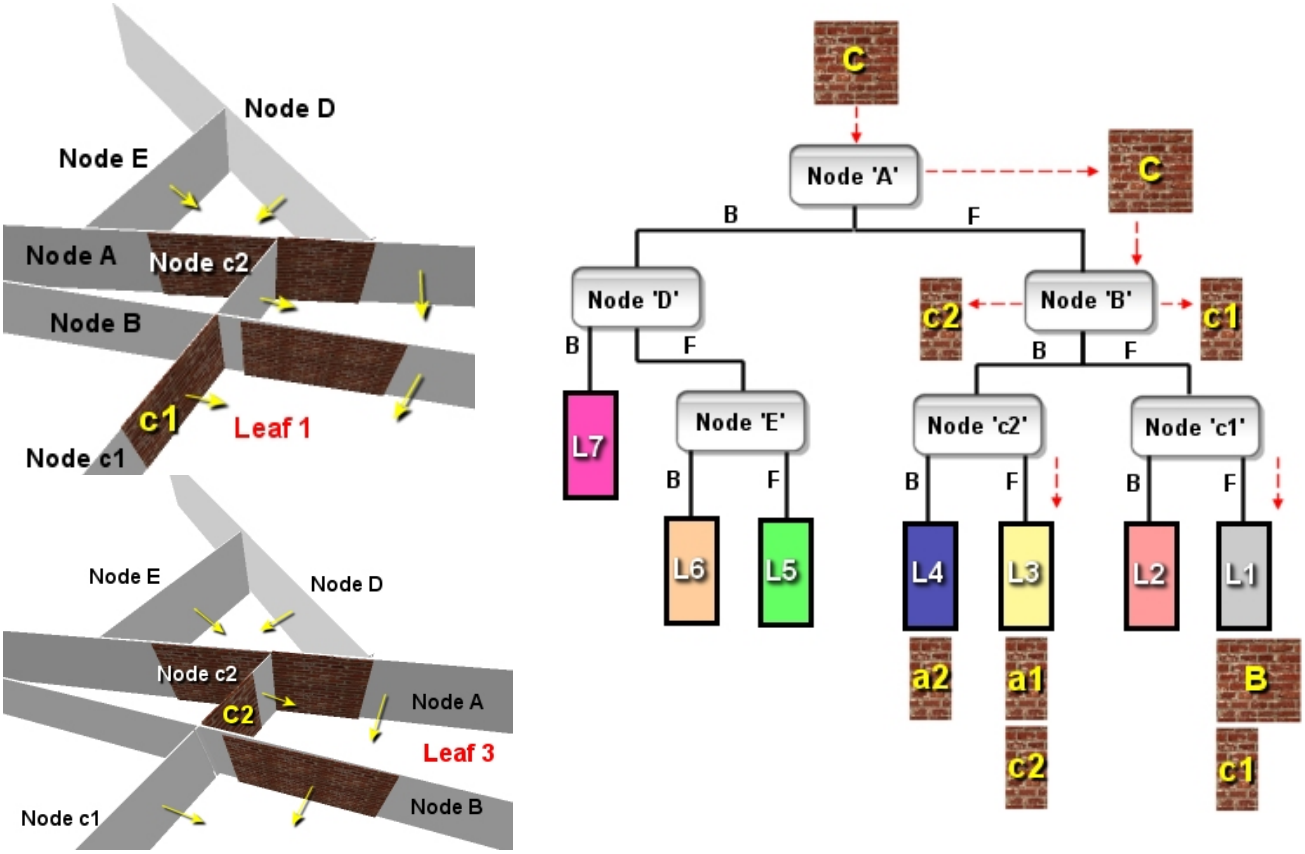


Figure 16.53

As the rightmost image in Figure 16.53 shows, polygon C is fed into the tree and is classified against the root node. Polygon C is found to be located entirely in the front space of node A so is dispatched down the front side to node B. At node B a similar classification takes place between the polygon and the node plane and this time polygon C is found to be spanning node B. This means polygon C will have to be split into two new fragments, c1 and c2 and the original polygon C will be deleted. Split fragment c1 is in the front halfspace of B so we will follow that fragment first.

As c1 is passed down the front of node B it arrives at node c1. It is obvious in this example that c1 is on plane with node c1 as this was the polygon that was originally selected to create the node's plane. Remember, we have built the tree and populated it in two different passes in this example. As polygon c1 is co-planer with node c1 and has a normal oriented into the node's front space it is passed down the front of node c1. However, node c1 is a terminal node so we know that to the front of this node must exist a leaf and in this example, that leaf is leaf 1. Polygon c1 is added to leaf 1's polygon list where it joins polygon B that was added to the leaf a moment ago. Polygon c1 has been fully processed and assigned to its leaf node so we unwind back up to node B again where we still have to process split fragment c2, which is located in its back space. Polygon c2 is passed down the back of B where it is found to be co-planar with node c2 and is therefore added to leaf 3 where it joins polygon a1.

With polygons A,B and C assigned to leaves in the tree, next we pass in polygon D whose original location is shown in the leftmost image in Figure 16.54. The rightmost image shows its route of classifications as it is passed down the back of A to node D. At node D it is found to be co-planar and

facing into the same frontspace and is therefore passed down the front where it arrives at node E. Polygon D is also located in node E's frontspace so is finally passed down the front and stored in leaf 5.

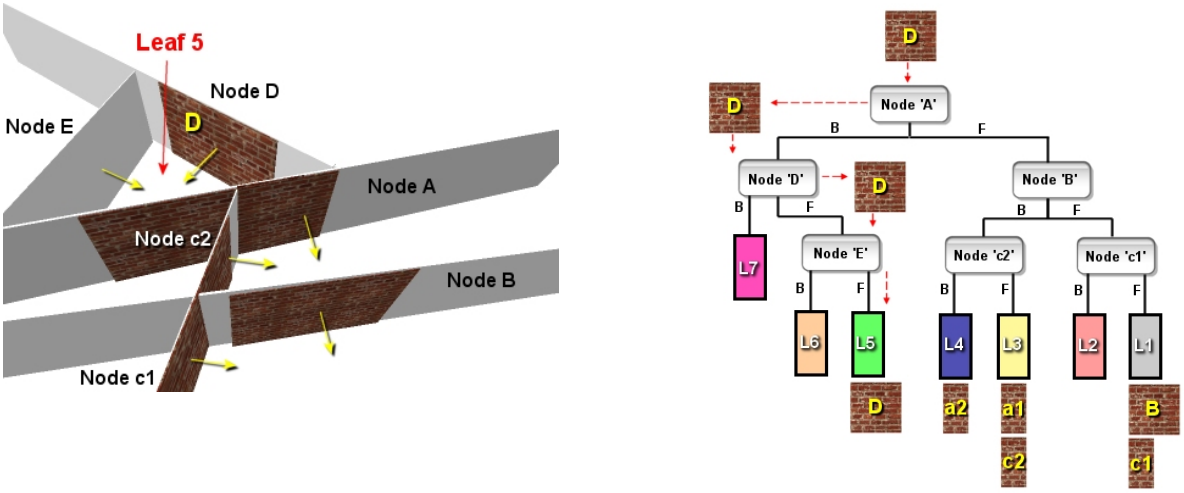


Figure 16.54

Finally, we have one polygon left to process, polygon E. It takes an identical route through the tree as polygon D (see Figure 16.55). It too ends up being assigned to leaf 5 where it joins polygon D.

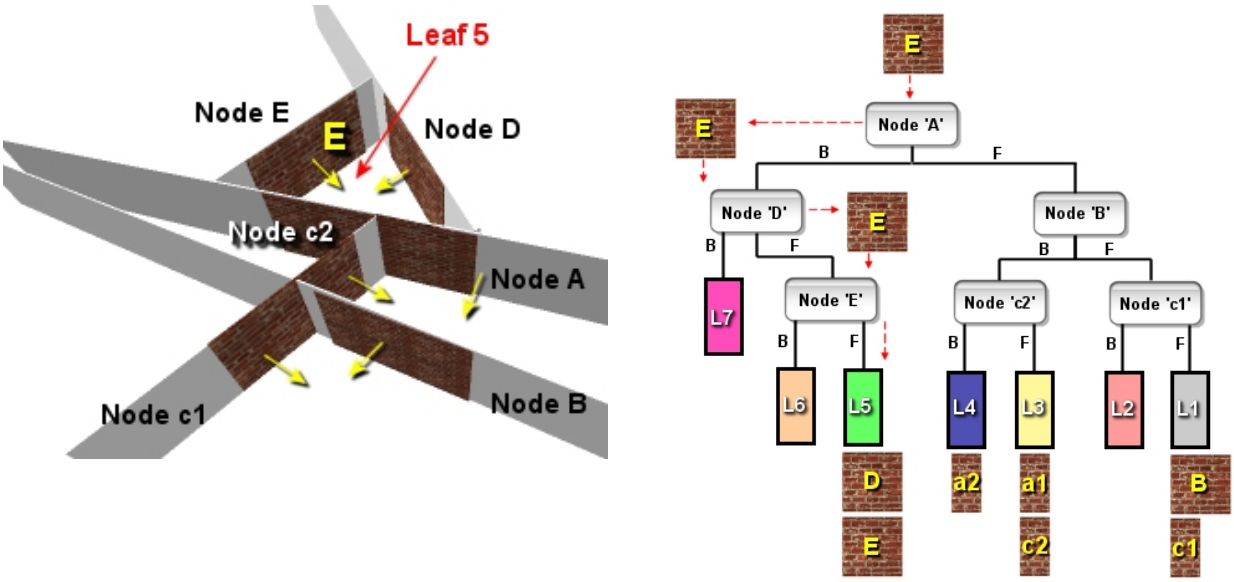


Figure 16.55

To clarify the process, we have performed the BSP compile in two different passes. Hopefully this has highlighted for you just how much the BSP tree is like any other tree we have developed so far. It has also shown us that the node tree also is the exact same tree as the leaf tree with the exception that we chose not to use the leaf information that was implied by the partitioning scheme. Our original node tree example now has been converted into a leaf tree and the final result is shown in Figure 16.56

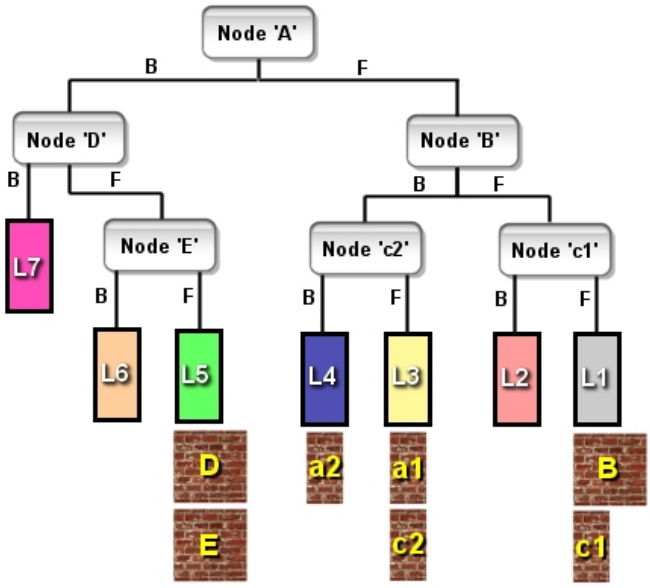


Figure 16.56

Notice that some leaves have had no polygon data assigned to them (leaves 2, 6 and 7). Also notice in all the previous diagrams that because of the way that we send a polygon that is co-planar and same facing down the front of that node, a polygon will always be assigned to the leaf that it is facing into. It might seem strange to think of the camera being in leaf 5 and not being able to see polygon A whose plane also bounds that region but polygon A would be back-facing and therefore would be back face culled by the pipeline. ***The only polygons assigned to a leaf are the ones that exist on one of the leaf's bounding planes and that have normals that face into that leaf.***

This would seem to cause a bit of problem. If the camera was located in leaf 5 for example, we would expect to see polygon A would we not? This is true, but this whole problem is caused by the fact that we have used as our example (quite deliberately) a completely infeasible data set. That is, we have developed data where it is possible for the player to see the backs of polygons, which is something you would never be allowed to see in a real scene. As we know, back face culling means that if we were to ever be allowed to walk around a polygon and view its back side, it would essentially disappear in front of our very eyes causing a complete breakdown in the solid integrity of the level. You will see shortly that when provided with proper legal geometry, this situation will never arise and any leaf which represents empty space (the space in which the player is allowed to be) will be bounded only by polygons that face into the space of that leaf.

In the above examples we built a tree of planes first and then pumped the polygons through the tree in a second pass to populate the leaf nodes. While this method certainly works it is a little redundant to perform a second pass when we already have the polygon data available during the first pass. Later, we will show some example code of how to merge these two processes together to create a function that will compile a BSP tree from an input polygon list and populate the leaf nodes with the polygon data as it is being constructed. While this might sound quite complicated, it really is not. When you think about it, this is exactly how our quad-tree, oct-tree and kD-tree were built in the previous lesson. There are some added complications for sure due to the fact that the polygon list is not only being passed down the tree and collected at the leaf nodes but is also being selected to construct node planes at each step, but they are trivial and can be resolved with a line or two of conditional logic. However, for the time being we will stick with the two pass approach which will help us more clearly demonstrate certain topics over the coming sections.

16.3.2 BSP Trees and Convex Areas

In this section we will briefly discuss how and why the BSP tree compiler always manages to break the most complex level into a series of simple convex areas. This will also highlight the BSP tree's ability to represent any complex mesh internally as a series of convex areas that can be easily queried for point/ray/etc. containment. This is important because we will often wish to determine whether a point is contained within a non-convex object, which is much more expensive than testing against a simple convex volume. It should be noted that you do not need to know this information in order to build a BSP tree but may find it an interesting read and helpful to really understanding the inner workings of the BSP compilation process.

Perhaps the best first example of how binary space partitioning using the polygon planes works to break the scene down into convex areas can be shown by examining the process of breaking a non-convex polygon into a convex one. All the technologies we have developed thus far are designed to work with convex polygons and therefore, this information may become useful to you if your scene has been designed in such a way that it contains non-convex polygons. In such an instance, you will have to isolate these polygons and break them down into a series of convex ones. This is not something you will usually ever have to do as most level editors and modeling applications will enforce the exportation of convex polygons and perform the convexity process for you. However, imagine if you were writing your own world editor like GILES™ and you allowed the user to edit a polygon at the vertex level. It is quite possible that the vertices of the polygon could be manipulated in such a way as to form a non-convex polygon. It is important that the editor break this up into convex polygons as our collision routines and point in polygon tests rely on convex polygons being used.

Figure 16.57 shows a non-convex polygon. A useful definition of a convex primitive is one that states that the infinite planes which form the boundary of an object must **never** intersect that same object's interior space. If we were to apply that same definition to a polygon, this would mean that if we were to iterate through each of the polygon edges and construct planes from them, at no point should the polygon itself be found to be spanning any of those edge planes if it is to be considered convex. In Figure 16.57 an edge is highlighted red which clearly shows that its infinite plane would cut through the polygon's interior.

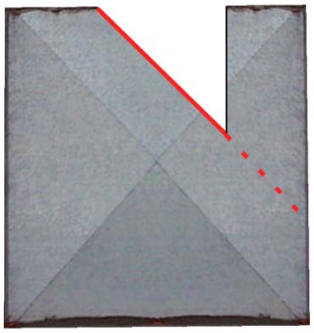


Figure 16.57

Using such a primitive would completely break our point in polygon intersection tests because we can no longer test whether a planar point is interior to a polygon simply by testing if the point is contained behind all of its edge planes (as we can in the convex case). It is quite possible for a point to be situated within the interior of the polygon shown in Figure 16.57 but still be located in front of one of its planes. Our point in polygon test would reject this point as soon as it was found to be in an edge plane's frontspace and would assume that the point is not interior to the polygon. You should be able to easily find a region in this polygon where a point could be located in front of the edge plane highlighted red in Figure 16.57.

Imagine that we wished to test if this polygon was convex or not. We could loop through each edge of the polygon and construct a plane from it (we learned how to do this in earlier lessons). For each edge

plane we would classify the points of the polygon against it. If we find an edge plane that has vertices in both of its halfspaces, we know this is a non-convex polygon and thus we split the polygon along that plane into two child polygons which are hopefully convex. Figure 16.58 illustrates that once the red edge plane in Figure 16.57 has been located, we could split the polygon by this plane thus creating two convex polygons which can be used in its place. The original non-convex polygon can then be discarded.

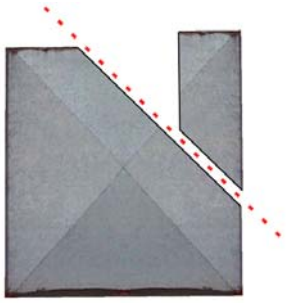


Figure 16.58

In this particular example our task would be complete because by finding this spanning plane and splitting the polygon, the result is two convex polygons which can be used instead of the original. We can now see that we can determine if a point is inside any of these two convex polygons simply by testing if a point is contained behind all of their edge planes (or in front of the edge planes depending on the directions you choose for your edge plane normals).

You may also notice that we could have easily solved this problem using the other spanning edge plane shown in Figure 16.59. As you can see, had the edges been tested in a different order, a different spanning edge may have been found first and used to split the polygon into two convex pieces. In this case, the results are different but the overall goal has still been achieved. That is, the two resulting polygons are different from those generated from the edge plane used in Figure 16.58 but the two resulting pieces are still convex. This is not unlike the BSP tree compiler where, regardless of the order in which we choose the split planes, the level will always be broken into convex regions. The shape and size of those regions may be totally different and dependant on the order in which polygons were selected as split planes, but the overall goal of convexity is achieved regardless of the splitter selection order.

Figure 16.59

Of course, it is entirely possible that even after the non-convex polygon has been split by a spanning edge plane, that the two resulting pieces may themselves still not be convex. Therefore, after the non-convex polygon has been split into two child polygons, the same process must be repeated on both of those children to test for convexity. If we find while classifying one of the child polygons against each of its edge planes that a spanning case arises, we know that the child must be split by this plane into two further child polygons. The process repeats itself until we finally test the edges of the child polygons and find no edges which intersect the interior of the polygons.

In Figure 16.60 we see another example of a convex polygon which cannot be resolved into two convex components using a single edge plane split. In the leftmost image we show the original non-convex

polygon and we have highlighted the first edge plane we found that the polygon vertices are spanning. We split the polygon by this edge plane resulting in the two new polygons shown in step 2. The rightmost polygon in step 2 is clearly convex and when we test its edges we find that none of them intersect the interior of the child polygon. However, the leftmost polygon in step 2 does still have edges which intersect the polygon and therefore this child polygon will need to be further subdivided into two new children.

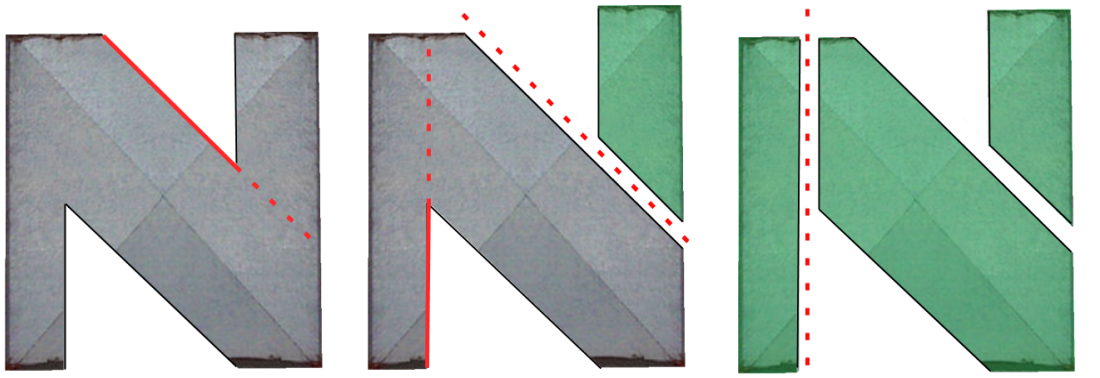


Figure 16.60

In step 2 we highlight the edge in red which we find in the leftmost polygon to be intersecting its interior and once again perform a split. This divides the leftmost child polygon into to new convex polygons shown in step 3. When we test the edges of these children we find that none of their edges intersect the interior space of the polygon to which they belong and thus, we have successfully broken the complex non-convex polygon into three convex ones.

When we examine the images in Figure 16.60 it becomes clear that using the edges of the polygons as the split planes will eventually always carve the polygon up into convex areas. It is a simple process of eliminating all spanning edges until they are removed. A convex object has no spanning edges so when we finally achieve this goal, we have logically eliminated everything from the original polygon that made it non-convex.

What also starts to become clear is that this is essentially how our BSP tree compiler subdivides our level geometry by using the planes of the polygons that comprise the level. For example, if we imagine that the polygon shown in step 1 of Figure 16.60 was actually a top down view through the roof of a building, we can see that the building would be broken up into convex areas using this technique. Our BSP tree compiler does exactly the same thing -- it makes sure that every polygon in the level is used as a split plane and when a plane is found which does intersect the geometry in the level, the geometry is split and assigned to polygon lists that are passed down either side of that node.

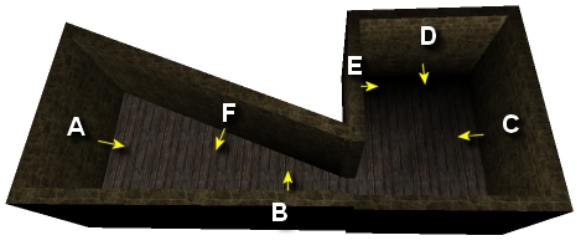


Figure 16.61

In Figure 16.61 we see a top-down view of a piece of 3D geometry. The geometry is a single room, but that room is non-convex. We will now see how the BSP compiler would break this non-convex geometry into two convex leaves. ***Again, each leaf always represents a convex area in the BSP tree.***

> **Note:** In Figure 16.61 and the images that follow the polygons are assumed to all be facing in towards the interior of the room. Back face culling has been disabled to show a better illustration of the geometry being compiled. Furthermore, the polygons in these diagrams have been given some degree of thickness to aid the clarity of the diagram, but as we know, in reality these polygons would be infinitely thin. Finally, although we have shown the room as having a floor polygon, we will ignore this polygon for now to help simplify the number of nodes and operations that would need to be done. The floor is simply there to aid the 3D perspective of the image. We will compile only the walls.

We will now step through the process that would be involved in building a BSP tree using this level. We will use this example to get more comfortable with the idea of passing the polygons down the tree into the leaf nodes during BSP compilation. As this is a fairly simply level to compile we will not accompany each image with a tree diagram. All we are trying to accomplish here is an understanding of why the BSP tree carves up the space of the scene into convex regions. The polygons in these images have been labeled A through F, which indicates the order they are to be selected as splitters during the compile process.

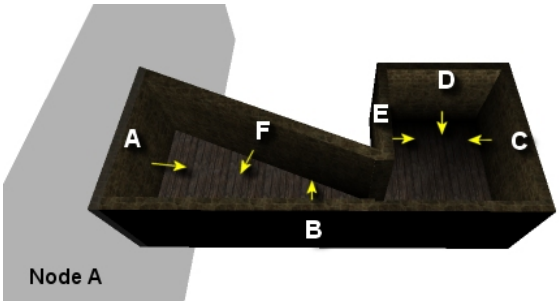


Figure 16.62

In Figure 16.62 polygon A is selected as the root node and marked as having been used as a splitter so that it will not be selected again. All the polygons (A through F) are classified against node A and are found to exist in its front halfspace, so they are compiled into a front list and passed down the front.

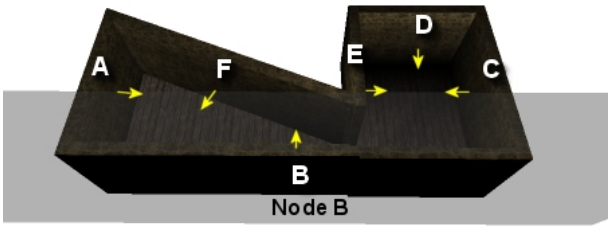


Figure 16.63

Polygon B is selected from the list at A's front child and is marked as having been used as a splitter. All the polygons (A through F) are in node B's front space so are passed down the front.

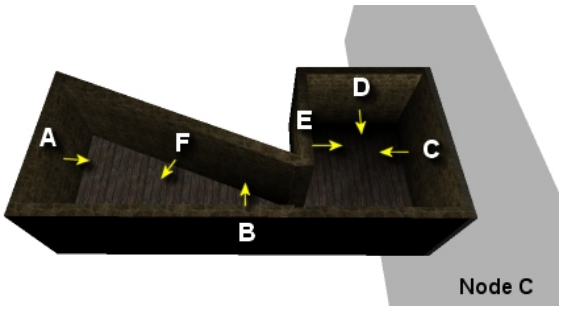


Figure 16.64

Node C is selected next to be the child node of B (Figure 16.64). Polygon C is marked as having been used as a splitter and all the polygons are classified against node C and found to exist in its frontspace. Once again, they are all added to the front list of the current node and sent down the front of node C where a new child node will be created.

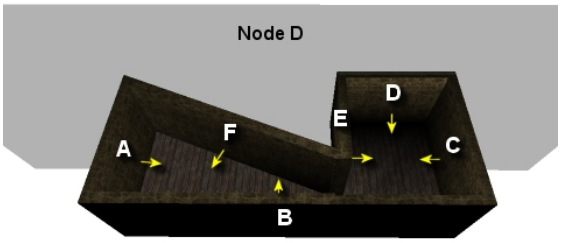


Figure 16.65

In Figure 16.65 we can see that polygon D is selected as a split plane next and the polygon is marked as having been used as a splitter. All the polygons in the list that were passed into node C (polygons A through F) are classified against node D and are once again found all to exist in its frontspace. They are packaged into the front list and passed down the front of node D.

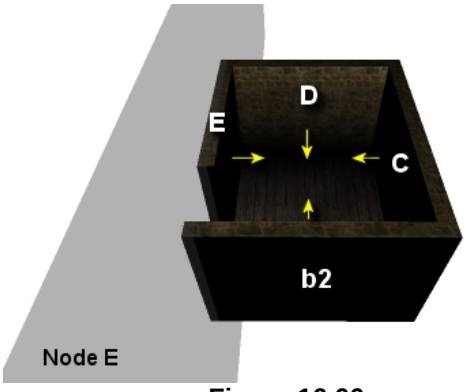


Figure 16.66

Down the front of D the BSP compiler chooses polygon E as the next split plane. This is the interesting bit. Plane E is the plane that intersects the interior of the geometry and therefore the geometry in the list will need to be split. When the polygons are classified against node plane E, polygons E, D and C are found to exist in its frontspace and are added to the front list, while polygons A and F exist in its backspace and are therefore added to the back list. Polygon B however is spanning the plane and is split into fragments b1 and b2 with b2 being the fragment located in the front halfspace of node B. b1 is added to the back list and b2 is added to the front list. What is vitally important to know is that when polygon B, which has already been used as a split plane, is split, we carry over the status of its 'BeenUsedAsSplitter' Boolean into the child splits b1 and b2. This will stop b1 and b2 from being used as split planes later since their parent polygon has already been used. Otherwise we would have redundant split planes which would create more nodes in the tree.

The front list at node E will contain polygons E, D, C, and b2 and these will be passed down the front of node E as shown in Figure 16.66. When we get down the front of node E however, we find that we have no more polygons that have not yet been used as splitters and therefore, our job is done. At this point we must have a group of polygons whose planes represent a convex volume and therefore we have determined that there is a leaf in front of node E. Furthermore, by clipping and passing the polygons down the tree during the build process, we have also collected the polygons for this leaf at the same time. As you can see in Figure 16.66, the first leaf we create is comprised of four polygons which all face into the center of the leaf. If we query the tree for a position and find that the query position is located in this leaf, we now it is located in the section of the original room shown in Figure 16.66.

Note: A leaf is created whenever we have a list of polygons passed into a node have all been used as splitters. Unlike the quad-tree, oct-tree and kD-tree which can have many stop codes which determine when a leaf is created, with the BSP tree we keep creating child nodes until we find that a node is passed a list of polygons which have all be used to create split planes. No other stop code is necessary. As soon as no split candidates remain, a leaf is created and the planes of the list polygons passed into that node are guaranteed to form a convex region that bounds the leaf.

With the leaf in front of node E created we must now process node E's back list which contained polygons A, b1, and F. When we step down the back tree of node E we find that only one polygon remains in the list which has not yet been used as a split plane. This is polygon F which is used to create the back child node of E. Polygons A, b1 and F are classified against node F and are all found to exist in the front space. However, as polygons A, b1 and F have all been used as splitters we do not create a new front child for node F but instead attach a leaf structure to its front containing these three polygons. We now have our second convex leaf. Our level as now been divided into two convex leaves as shown below in Figure 16.68.

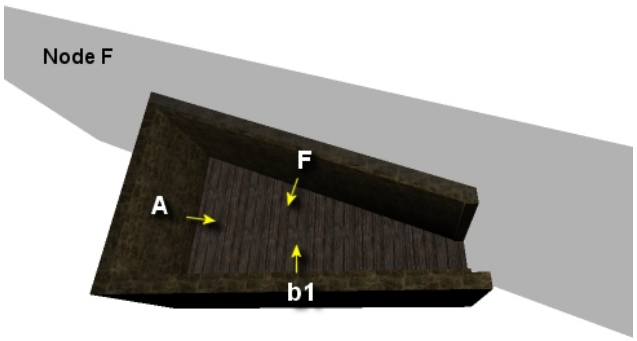


Figure 16.67

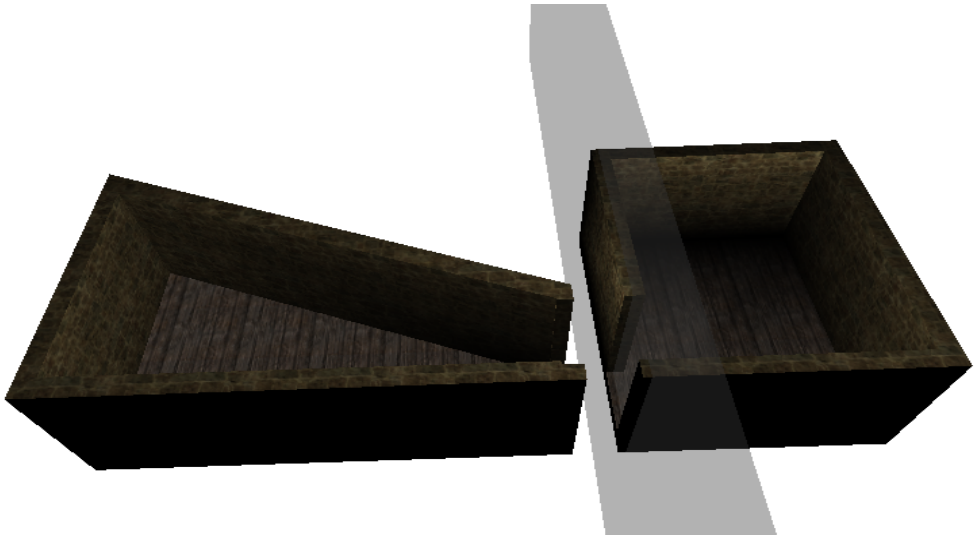


Figure 16.68

We have now seen exactly why the BSP tree compilation technique breaks the world into a series of convex areas and we have also seen how much it is like the polygon examples we examined earlier. In this case, it was node E's plane that was found to be the spanning plane and responsible for dividing the initial polygon list into two child lists in front and behind the split plane. We have also seen how a leaf is created whenever we reach a point in the recursive process where every polygon in the list passed into a node has already been used as a splitter, making it impossible for another node plane to be selected. It is also worthy of note that whenever no polygon data gets passed down one side of a node, an empty leaf is created there. Figure 16.69 shows the final resulting planes of the BSP tree that was just compiled and we can see that it has carved the world up into seven convex leaves. Only two of the leaves contain polygon data and those are the only two leaves that have all their bounding plane normals facing into the interior of the leaf.

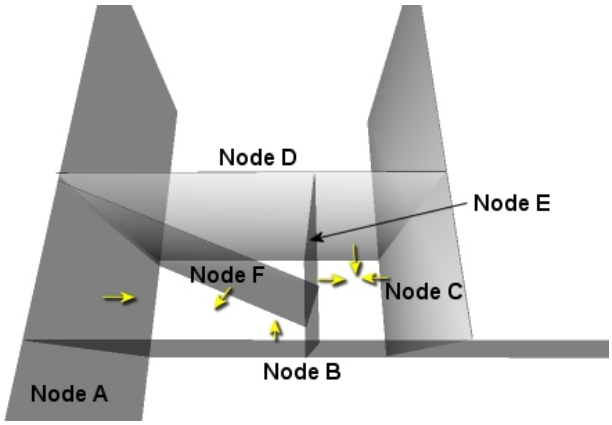


Figure 16.69

16.3.3 Solid / Empty BSP Trees

The title of this section might lead you to believe that we are going to discuss a different type of BSP tree that can encode solid and empty information. However, the current tree we have will already do that if we send it geometric data in an ordered and sensible format. Providing that we send the compiler geometry that obeys certain conditions (i.e., *legal geometry*), a level will be compiled such that polygon data will only ever be stored in front leaves. Back leaves will always be empty of polygon data and therefore, represent areas of solid space within the game level.

When we refer to a BSP tree as a 'solid' BSP tree, it simply means that because of the way the input data has been constructed, we can easily deduce which areas of the scene are empty of obstructions and which areas of the scene are assumed to be solid space (e.g., the space in the middle of a brick wall). Solid space is space that the player cannot see through and is a region where the player should never be allowed to move. We will see shortly that assuming we send the geometry to the BSP compiler in the correct format, every leaf attached to the back of a node will represent a solid area (i.e., a leaf of solid space). Every leaf attached to the front of a node will represent an empty space leaf. That is, a leaf in which the area of space is assumed to be empty and that the player can both see and navigate through. The polygons of the level will always be assigned to empty leaves (front leaves), which may sound strange at first as we normally consider the polygons to be the solid objects that we cannot navigate through. This is still the case of course in theory. However, as discussed earlier, the polygons will always lay on the boundary planes of a leaf and therefore, it is the empty space between those polygons in which the player is allowed to navigate. Furthermore, because of the way we handle the on-plane case during BSP tree construction, polygons will always end up in leaves which their normals face into. This means, when the player is located in empty space, the polygons assigned to that leaf will be facing into the leaf, and therefore, facing into the same halfspace as the player.

Although this might all sound a little abstract (until we look at some examples of geometry that will compile a solid BSP tree), assuming that this is correct, why is it important that we have access to this solid/empty space information? The reasons are actually numerous and quite important.

First, if we can guarantee that every back leaf represents solid space, then performing line of sight tests becomes trivial. If we wish to determine whether point A can see point B, then we would create a ray from point A to B and send it through the tree. At each node the ray would be sent down the front or back of the tree depending on the ray/plane classification result at that node. If the ray is found to span the plane then the intersection point C with the plane is calculated and the ray is split into two by creating two child rays A->C and C->B. These two ray fragments are passed down their respective sides of the node. This process continues until the ray fragments pop out in the leaf nodes. We performed an identical process to this in lesson 14 when we discussed passing a ray through the kD-tree (CollectLeavesRay). However, for a line of sight test, if at any point we find that a fragment of the ray has been passed into a back leaf (i.e., a region of solid space), we know that no line of sight can possibly exist between these two points. That is because part of the ray passes through an area of space that is solid, such as a section of wall or floor for example. When this is the case we can immediately return false for the line of sight test. Performing line of sight tests in this way is much faster than testing each polygon for intersection with the ray. Using the BSP tree, we have no point in polygon tests to perform at all.

Another important reason for needing solid and empty space determination will become clear when we calculate a potential visibility set in the following and final lesson. A potential visibility set is a block of data that describes to us which leaves are visible from any other leaf within the tree. If we have a potential visibility set at our disposal, we can simply traverse the tree each frame to find the leaf in which the camera is currently located and fetch the visibility set for that leaf. This will instantly tell us which leaves are visible from the current leaf we are in and we can render each of these leaves. The potential visibility set will not calculate the visibility information based on leaf inclusion within the camera frustum, as with our previous rendering systems. Instead, it will be calculated at development time and will take occlusion into account. This means that if a leaf represents a small room with a small open doorway, the only polygons visible to that leaf will be the polygons forming the walls, floor, and ceiling of the room itself and the small number of polygons that can be seen through the open doorway from within that room. All other leaves that lay behind the walls will not be in the leaf's visibility set and thus we typically render only a small number of polygons and reduce overdraw significantly. Essentially, instead of rendering everything that is inside the frustum, we render only what is actually visible from that location. Polygons that are located well within the frustum but are occluded by nearer geometry will not be in the visibility set for that leaf and will not have to be processed in any way. The potential visibility set for each leaf takes a very long time to compile (sometimes many hours) so it is done as a development time process. The information is then saved out to disk and loaded into the game at runtime and used. In the following lesson we will write a PVS (Potential Visibility Set) calculator and one of the chief pre-requisites for being able to calculate the PVS for a game level is that the level be compiled into a solid BSP tree. Take a look at Figure 16.70 to see why this is the case.

Here we see the two leaves that the non-convex room was broken into in our previous example. In this image we have deliberately separated the leaves so that we can see the gaps in the polygon data were the two leaves connect. We can think of this gap as the doorway between the two leaves. The plane that created the split into two leaves is also shown. You can imagine how in a really complex level thousands of leaves will be created which all have similar doorways that lead into one another. In order to calculate the visibility for a leaf, we will need to know the size of each of these doorways so that we can build view volumes out of them which will describe everything that a leaf can see through its doorways (that lead into other leaves). Do not worry about how this is done as this will be the entire focus of the following lesson. The problem we have is that we need to know the size of these doorways which is information we currently do not have. We know the planes on which these doorways will lie because they are the node planes that split the geometry (the gray plane shown in Figure 16.70) but we do not know the size of the doorway. We will need to build temporary polygons (called portals) that will fit these doorways exactly. Once we have calculated all the portal polygons for a given leaf we will have created polygons that fit these doorways. We will then be able to construct view volumes from these portals which will describe what can be seen from the leaf, through those portals, and into other leaves. This will tell us which other leaves are visible from the current leaf having its visibility set calculated. If we know which areas of the world are solid space (such as located in a wall for example) and which areas are empty, we can build an initially huge polygon on that split plane and send it through the BSP tree. Any fragment of the polygon that ends up in a solid leaf (a back leaf) can be deleted as it is obviously situated inside a solid object or is outside the exterior walls of the level. At the end of the process we will end up with a polygon fragment that ends up in empty space and this is the portal(s) to the leaves that were subdivided by that split plane.

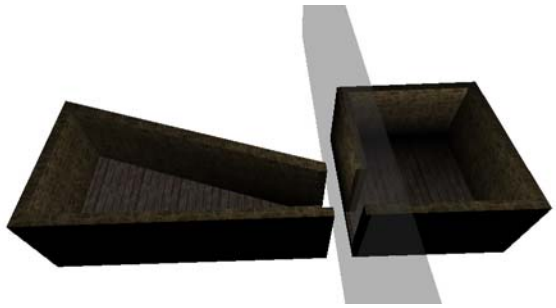


Figure 16.70

The calculation of a PVS will be discussed in the following lesson but what has become very apparent is the need of the PVS calculator to be able to determine during the portal creation phase whether or not a portal fragment has ended up in solid or empty space. Thus, we need to make sure we provide the PVS calculator with a BSP tree which has been compiled with geometry such that every back leaf can be considered solid space and every front leaf can be considered empty space. This is all about the geometry we send into the tree and not about changing any of the BSP compiler code in any way, as we will soon examine.

Another reason we will need to be able to compile a solid tree will be when performing CSG operations which will be discussed in the final section of this lesson. CSG operations allow us to carve one object from another or union two objects together into a single mesh. We will see later that the union operation is especially important to us as it allows us to fuse all the meshes comprising the level into a single static mesh which can be compiled into a solid BSP tree. The union operation will remove hidden surfaces, which are essentially polygons from one object embedded (or partially embedded) inside the interior space of another object. Geometry that has such arrangements of polygons is considered illegal geometry as it breaks the rules laid down by the solid BSP tree. The tree compiled from such data will contain invalid solid/empty space information thus causing PVS calculation and any future CSG operations on that geometry to fail.

Note: Such data is referred to as 'illegal geometry' from the BSP tree's perspective and will result in an invalid BSP tree. We will discuss illegal geometry in some detail later on in this lesson and will discuss ways to correct such problems in the geometry. CSG operations allow us to remove these hidden surfaces that would cause the BSP tree created to be invalid and is another process that relies on solid/empty information.

So we have seen that the need to be able to build a BSP tree which contains solid/empty information is important especially to future processes that we will undertake. We have also learned that a solid BSP tree is fundamentally no different from a normal BSP tree with respect to the code -- it all comes down to the geometry that we send our compiler. If we send legal geometry to our BSP compiler, a level will be created which will always have empty back leaves and polygons stored in the front leaves. Those back leaves will always represent solid space and the front leaves will always represent empty space.

This may all seem incredibly hard to imagine at the moment. Indeed it seems odd that by just supplying geometry constructed in a certain way we will get this solid/empty information for free. However, it is absolutely true and is the exact reason that artists that are developing scenes for a game engine that uses BSP/PVS technology will use world editors such as WorldCraft™ or GILES™. These editors allow the artist to construct worlds from a series of simple solid primitives using CSG operations to carve and union these simple shapes into more complex objects. The editors force (for the most part) the artist to build scenes that are considered legal geometry just by the very way that they are constructed. This will all make more sense when we cover exactly what solid/empty space is and what is considered legal/illegal geometry by the solid leaf BSP compiler.

In our node tree discussion we used an example level that was extremely unrealistic and made reference to the fact that because a polygon lay on of the node planes, it may actually lie on the boundaries of a front and back leaf. Figure 16.71 highlights such an example from this level. We can see for example that polygon B has been assigned to the leaf in its node plane's frontspace and therefore this polygon would be assigned to leaf 1. However, notice that this polygon (and its plane), also form a boundary for the leaf that lies behind the plane. If our camera was located in the leaf bounded by planes A, B and c2 for example, we would be looking at the back of polygon B, which would be back face culled. This would allow us to see right through this polygon into leaf 1 which would completely destroy the visual integrity of the level.

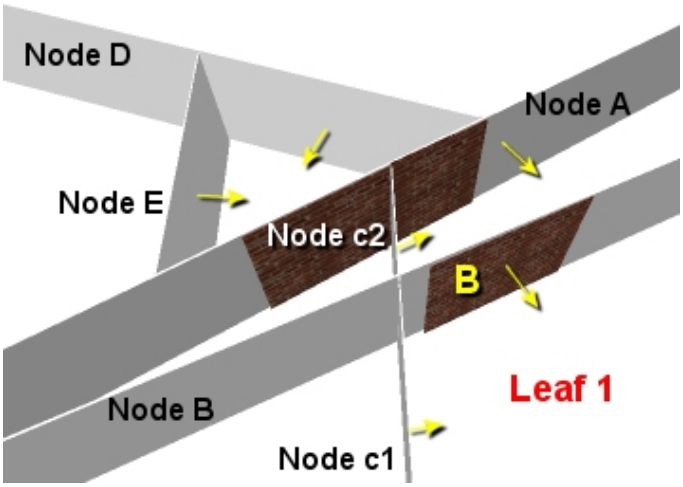


Figure 16.71

Of course, this never happens in a real game since our artists would not include a wall that can be viewed from all sides as a single polygon. As Figure 16.72 shows, if we were to represent a section of wall as a single polygon it would only be visible to the camera when viewed from its front side. If the player was allowed to navigate the camera such that it could view the polygon from behind, it would be back face culled and would therefore be invisible.

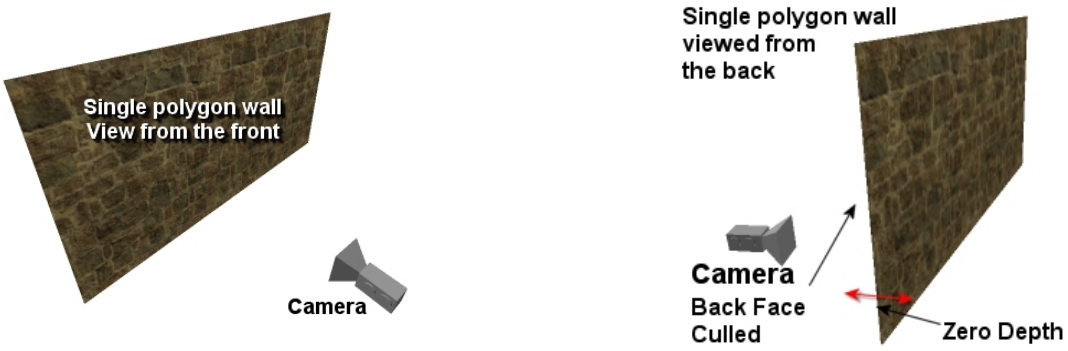


Figure 16.72

Representing a wall in such a way would only be acceptable if that wall formed one of the exterior walls of your interior scene and as such, the camera could never navigate around the back of it. In fact, even when viewed from the side or top we compromise our belief that this is supposed to be a solid wall as it would become clear that this polygon is nothing but a thin sliver of texture.

World editors such as GILES™ aid in helping the artist avoid such mistakes by forcing the placement of solid objects. For such a wall section, a cube would be placed and scaled to fit the desired dimensions. The cube has 6 faces instead of 1 and can be viewed from all sides. Regardless of the side you are viewing the wall from, there will always be faces with normals oriented towards the view direction. This obviously means that 6 faces get added to the scene instead of 1, but it also means that we now have a wall that is not only viewable from all directions, but also has a thickness when viewed from the sides, above or beneath, as shown in Figure 16.73.

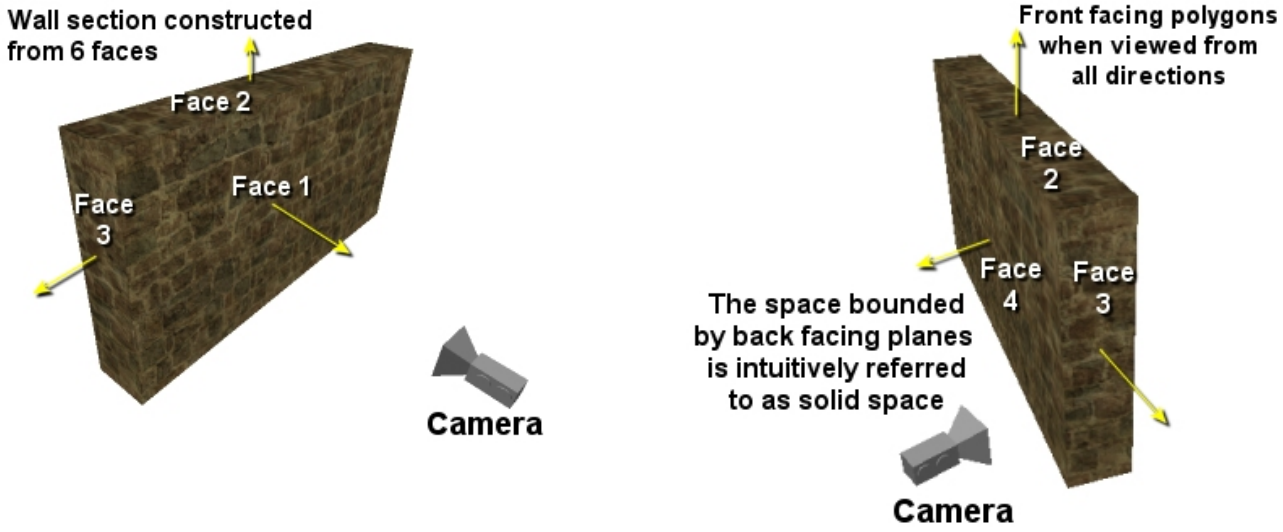


Figure 16.73

Assuming that this wall was the only geometry in our scene, let us see what happens when we compile it into a BSP tree. For the rest of this example we will simplify to a 2D top down perspective and will ignore the top and bottom faces. That is, we will assume that we are compiling a 4 faceted cube instead of a 6 and are viewing that cube from above.

Figure 16.74 shows the compilation of this 2D wall section (the cube). Polygons pA, pB, pC and pD are the outward orientated faces of the wall and the gray arrows show their face normals. The polygons are assumed to be selected as splitters in alphabetical order during the build process, creating node planes A, B, C and D. To the right we see the BSP tree that is generated from such data.

As you can see, node A is selected first and all of the polygons are classified against it. pA is co-planar and same facing which means it is sent down the front of node A. As the only polygon in the front list is polygon pA, which has already been used as a splitter, a leaf is created and polygon pA is assigned to it. The rest of the polygons are added to the back list and sent down the back of node A.

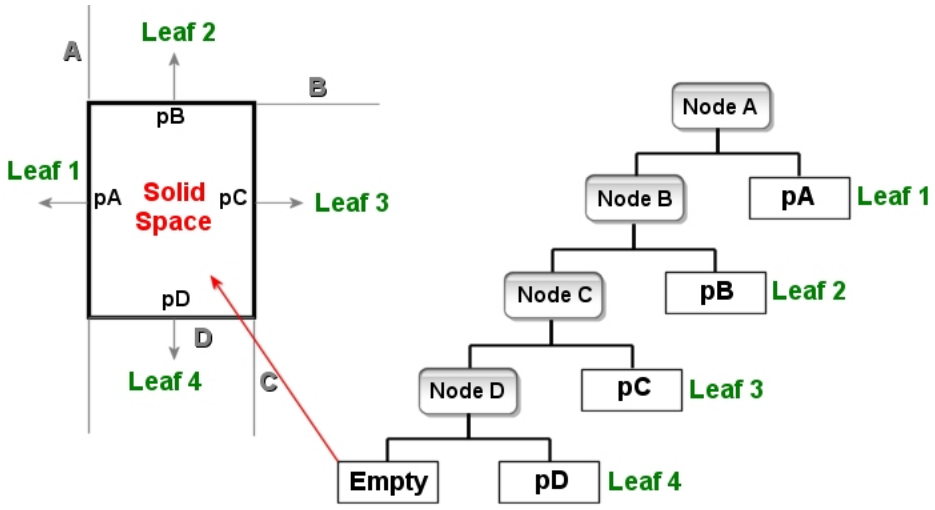


Figure 16.74

Down the back of node A polygon pB is selected as the next splitter which creates node B. Polygons pC and pD are added to the back list and polygon pB is co-planar and same facing which means it gets added to the front list. As there is only one polygon in the front list for node B (polygon pB) which has already been used as a splitter, a new leaf (leaf 2) is created down the front side of node B and polygon pB assigned to it. As we continue down the tree we see that polygons pC and pD end up getting assigned to their own leaves down the front of nodes C and D respectively. At node D however, there are no polygons in the back list so an empty leaf is created. As Figure 16.74 shows however, this empty leaf represents the area of solid space behind each of the polygons.

> **Note:** Whenever we compile a convex object with outward facing normals into a BSP tree we get a one sided tree as shown in Figure 16.74 where the nodes are all back children. If we were to reverse the plane normals so that all the polygons faced inward, empty space would be in the middle of the cube, solid space would be all around the outside of the cube, and the tree would be a one way tree down the front side instead of the back.

When we examine Figure 16.74 it starts to become clear why we can rely on a back leaf always being empty and representing solid space. When a polygon is co-planar and same facing, it is passed down the front of the tree, which means it will always end up in a leaf down the side of the tree which its normal is facing into. If the polygon is co-planar but not same facing, it is sent down the back list where it will later be used to create its own node. The polygon is obviously going to be found to be co-planar and same facing with this new node so is passed down its front. Thus, every polygon will eventually get added to a front leaf. Conversely, because this is the case, no polygons will ever be added to back leaves. Since all polygons always end up facing into the empty space of the leaf to which they are

assigned, the solid leaves represent the space behind those polygons and thus will never contain polygons of their own. Examining Figure 16.74 again, we can see that if we pass a query position through the tree that ends up in this empty back leaf, it will be located behind polygons pA, pB, pC and pD and therefore be located in center of the wall (i.e., in solid space).

Although this is probably about as simple a level as you could possibly compile, we will examine more complex levels later and show that the same holds true for our entire scene as long as we obey certain level creation rules. That is, provided we supply the BSP compiler with legal geometry, every solid region in the scene will be represented as a back leaf and empty space will always be represented as a front leaf. We will examine exactly what constitutes illegal geometry a bit later and see why it causes the solid/empty space relationship to break down.

The addition of solid and empty information to our BSP leaf tree is the primary reason behind the importance of ensuring that the on-plane case is handled correctly during construction. Recall that if a polygon is co-planar with the current node, then we add that polygon to the front list and ensure that it is not used as a splitter again. That is, we flag this polygon as used *only* if it points in the same direction. If a polygon's vertices are co-planar but its normal points in the opposite direction, then we add it to the back list, but we **do not** remove it from consideration as a future node plane candidate. This will ensure that the polygon will be selected as a split plane later, which is vitally important to the solid/empty integrity of our level

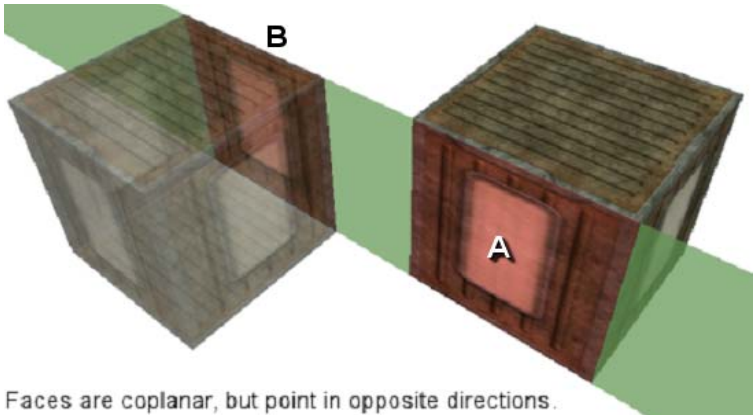


Figure 16.75

Consider the situation outlined in Figure 16.75. The example shown produces an arrangement unlike that of any of our previous test geometry. Here we have a scenario in which we find two polygons that lie on the same plane as one another but point in opposite directions. Imagine that we were to select the closest polygon as the candidate for the separating node plane depicted here. When we compare the polygon list against this plane, this candidate polygon is added to the front list as expected. Due to the fact that the co-planar polygon that is furthest away from the camera faces into the back halfspace of the node plane, it is added to the back list. As discussed in the previous section, we would not usually mark the co-planar polygon that was added to the back list as used. Yet, it may not be clear at this point exactly why this particular step is so important.

Contrary to our current building concept, if we were to remove **both** of these polygons from consideration by flagging them as used at this node, we would end up with a tree structure similar to that shown in Figure 16.77. Even though we are focusing on one particular node here and showing only the subsection of the tree that would be generated by the two cubes in the above example, we can clearly see that if *both* polygons 'A' and 'B' were flagged as used when node A was created, we would end up with a situation in which polygon 'B' has been added to a leaf that falls behind the node.

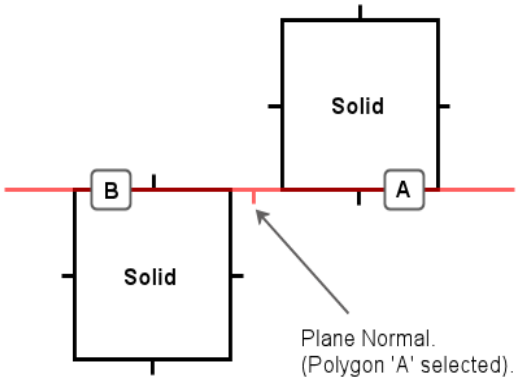


Figure 16.76

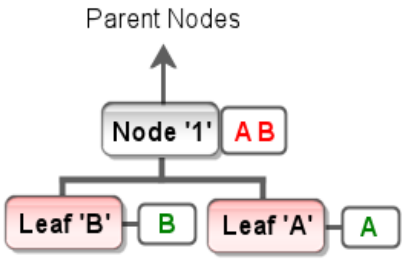


Figure 16.77

Note: Figures 16.76 and 16.77 do not represent a full and valid BSP leaf tree. These diagrams are merely intended to draw your attention to one particular portion of a larger tree.

In this particular example we can observe that the front side of polygon 'B' faces into a leaf contained in the back halfspace of the node. This conflicts with the key property that allows us to identify solid and empty areas in our tree because we now find that a leaf which should clearly describe empty space falls *behind* a node plane.

This is the reason why it is vital that we implement the on-plane case in the manner discussed in previous sections. By **not** marking polygon 'B' as 'used' during the classification step for the first node, we allow our compiler to select that polygon for node creation at a point further down the tree structure. As we can see in Figures 16.78 and 16.79, in this case we have allowed the compiler to generate a new node for each polygon which points in opposing directions.

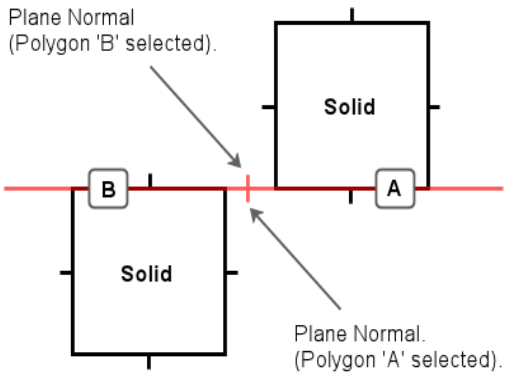


Figure 16.78

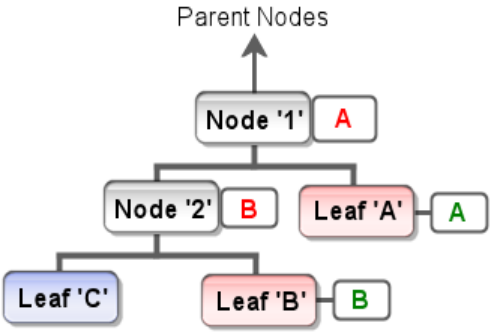


Figure 16.79

Once we allow polygon 'B' to create its own node, we can see that we have restored the ability to classify a leaf as solid if it is contained within the back halfspace of its parent node, or empty if it exists in front. Notice that polygons A and B both eventually end up getting assigned to front leaves.

Referring back to our single wall section example again and assuming that the wall section we have just compiled into a BSP tree is currently the only geometry in the scene, you will see that the BSP tree automatically provides us with the ability to perform line of sight tests in an extremely efficient manner.

In Figure 16.75 we show two positions that exist in empty space labeled J and K. Position J is located in leaf 1, where the wall polygon pA could be viewed. Position K is also in empty space (leaf 3) and from this position the wall polygon pC can be viewed. However, position J cannot see position K because the wall polygons are in the way.

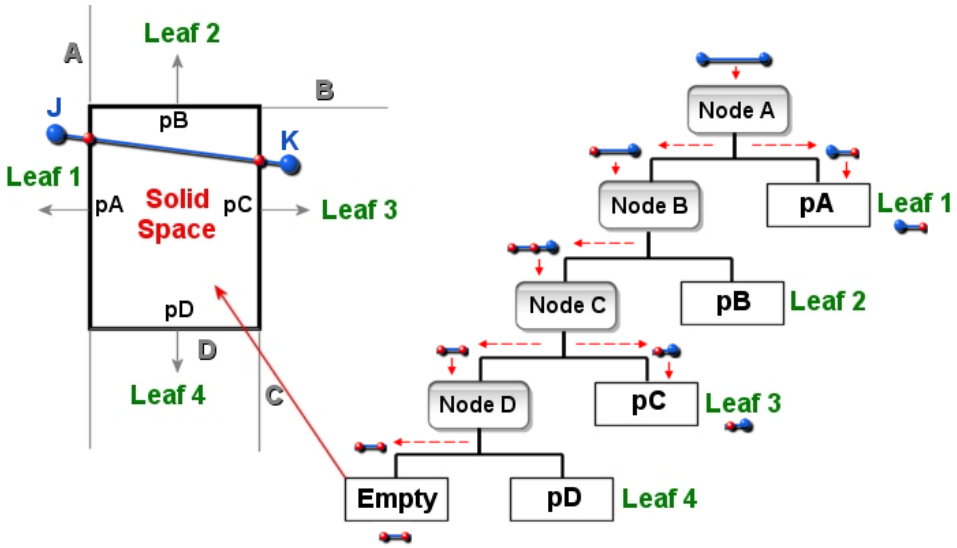


Figure 16.80

Usually, to determine if line of sight exists between points J and K we would have to perform expensive ray/polygon tests between the ray and each polygon in the vicinity of the ray. This could be a very large number of expensive tests in a complex level. However, we can see in this simple example that we can test very efficiently if line of sight exists between two points in a solid BSP tree by simply sending the ray down the tree and returning false as soon as a fragment of the ray ends up in solid space. If all fragments of the ray end up in empty space then a line of sight does exist. Note that this is determined without testing the actual polygon data.

As shown in Figure 16.80, the ray is sent in at the root node. A ray/plane test determined that the ray spans the node, so the intersection point with the plane is calculated. That ray is then split such that this intersection point becomes the end point for the child in one halfspace and the start point of the child in the other. In this example, the front split of the ray is sent down the front of node A where it arrives in Leaf 1, which is empty space. This is the section of the ray shown in Figure 16.80 between point J and the red dot on the plane of node A. The back split of the ray is passed down the back of node A where it is classified against node B. The ray fragment is completely behind node B so is passed down the back into node C. At node C the ray is found to span the plane and is split, creating two child rays. The ray fragment in the front halfspace of node C is shown in the diagram as the segment starting at the red dot on node C and ending at point K. This is sent down the front of node C where it lands in leaf 3 which is empty space. The ray fragment that was found to lie in the back space of node C can be seen in the diagram as the line that joins the two red dots on nodes A and C. This is passed down the back of node C where it arrives at node D. It is found to lay in node D's back halfspace and is passed down the back of node D into a back leaf. Since the ray has entered a back leaf this must mean that this fragment is in solid space (as we can clearly see in the top down view of the cube). Therefore, we return false from this node, and eventually the function, indicating that no line of sight exists between these two points.

It is important to understand that just because we want to have a solid BSP tree does not mean that every wall, floor or ceiling in the level has to be constructed using six sided cubes. It simply means that the geometry must be constructed such that empty space is always bounded by front facing polygons and

solid space is always bounded by back facing polygons. For example, if we were to invert the normals of the polygons in the previous example and compile a BSP tree, we would get a different, but still perfectly valid, tree.

Figure 16.81 shows the result of compiling the same polygons but with negated normals. Now the polygon normals all face into the center of the cube which creates only one empty leaf. Now, there are four back leaves behind nodes A, B, C and D which combined represent the solid space around the outside of the cube. This is still a perfectly valid solid tree. In fact, we might imagine that the inward facing polygons represent the walls of a room and leaf 1 is the empty space inside that room in which the player is allowed to walk. This is still perfectly valid because empty space is still bounded by inward facing polygons and solid space is still separated from empty space by the backs of polygons.

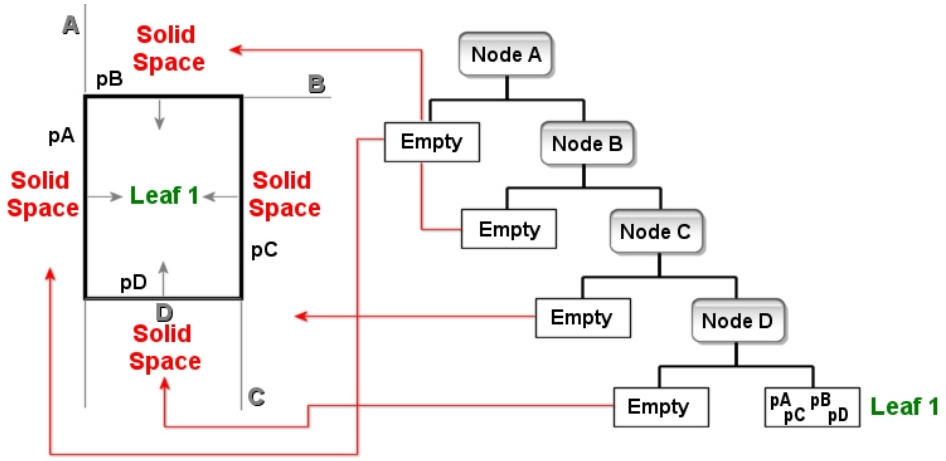


Figure 16.81

To illustrate the fact that our BSP tree code has not changed in any way, let us compile the non-convex room we looked at earlier. Remember when looking at Figure 16.82 that the walls of the room are supposed to be single inward facing polygons. In this diagram the walls have been given artificial thickness to better aid the visual demonstration. Back face culling has also been disabled so that polygon B is still rendered even though we are technically viewing it from its back side.

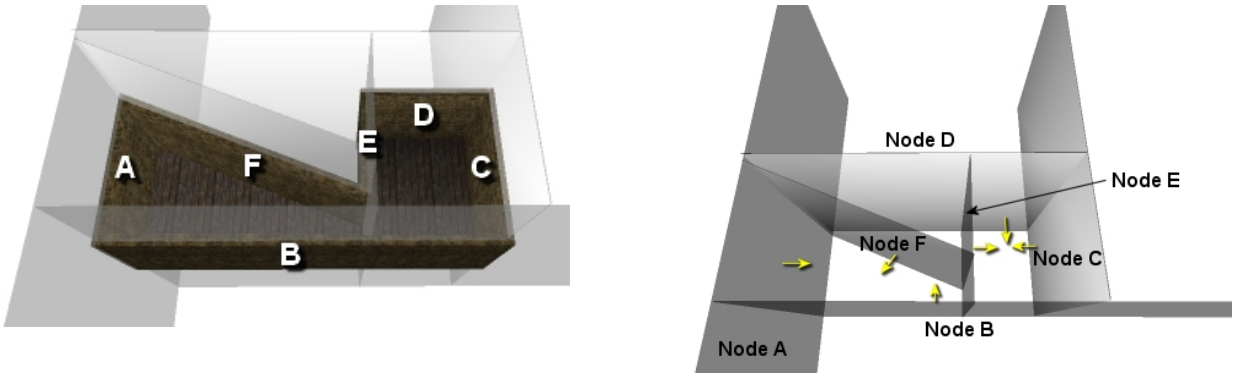


Figure 16.82

The leftmost image shows the original polygons that were used to compile the BSP tree. The letters assigned to them describes the order in which they were used to create node planes. Since we are now very familiar with the BSP tree building process, we will not show the construction process of the tree. The rightmost image shows the node planes that were created during compilation. The plane normal directions are depicted by the yellow arrows. Again, to keep the tree simple, we will pretend the floor polygon does not exist and has been rendered here only to aid with the visual component of the diagram.

Notice in the rightmost image that we have also created no node for this floor polygon (this is just a choice to keep the tree as small as possible during the examples).

Although you did not necessarily know it when we first examined this piece of geometry, this is perfectly legal geometry that will compile into a solid BSP tree. In this example, we will once again demonstrate our method using a two step approach. We will assume that the nodes of the tree were generated in an initial pass and that the polygons were then passed down the tree and collected in the leaf nodes in a second pass. The geometry depicted here obeys all the rules for keeping solid and empty space separate. The empty space is the space within the room itself and solid space is assumed to be all around the outside. Assuming that the BSP tree has already been compiled, let us now send the polygons (used to create the nodes in the first pass) into the tree and see where they end up. If this geometry is legal, we should find that all the polygons get assigned to leaves down the front of nodes and all leaves created behind nodes should represent solid space. This will also be our final example of populating the BSP tree with polygon data in a second pass. It will allow us to solidify exactly what happens to each polygon as it is passed down the tree before we merge this process into the node construction process and write our final solid BSP leaf tree compiler.

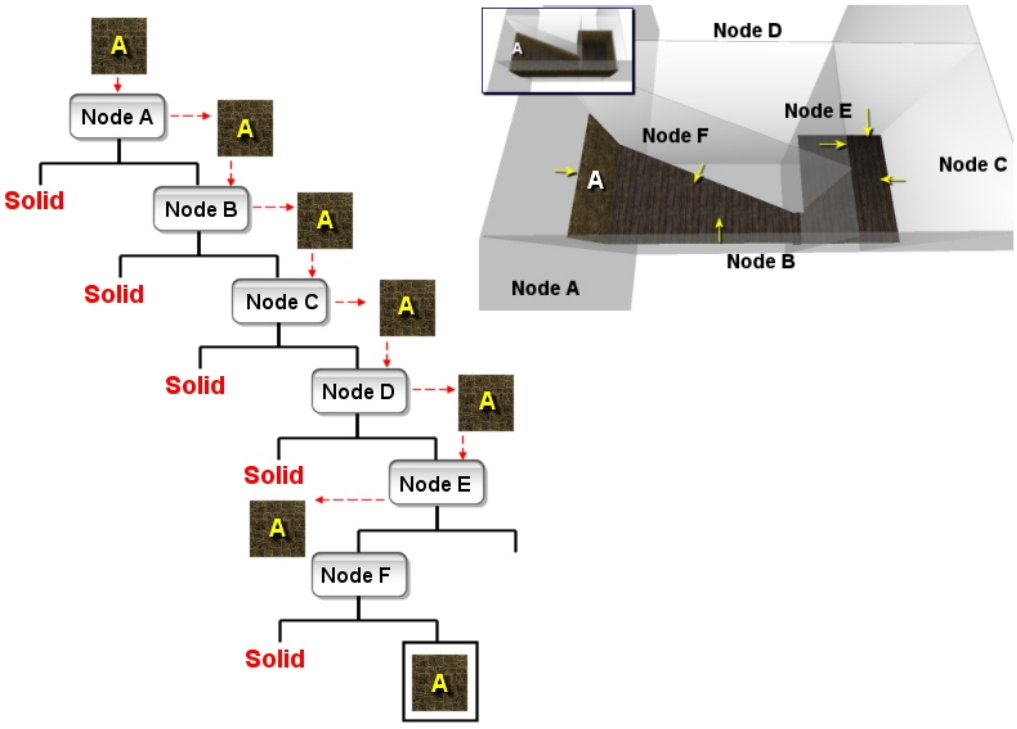


Figure 16.83

As Figure 16.83 shows, polygon A is sent into the tree first. At this point, you should be able to understand why the nodes of the tree are arranged in the order in which they are depicted. Polygon A is classified against node A where it is found to be co-planar and same facing and is therefore passed down the front into node B. We find that the polygon is also contained in the front halfspace of nodes B, C and D so it gets passed down the front at each step until it gets to node E. When classified against node E it is found to exist in its backspace and is passed down the back where it enters node F. As we can see in the diagram, polygon A is located entirely in node F's frontspace so is passed down the front of F. As no

more nodes exist down the front of node F it must mean that a leaf exists here to which polygon A is assigned.

The next polygon we pass down the tree is polygon B. At node A it is found to exist in the frontspace so is passed down the front into the node B. At node B it is obviously found to be co-planar and same facing (as this is the polygon that created this node) so is passed down the front into node C.

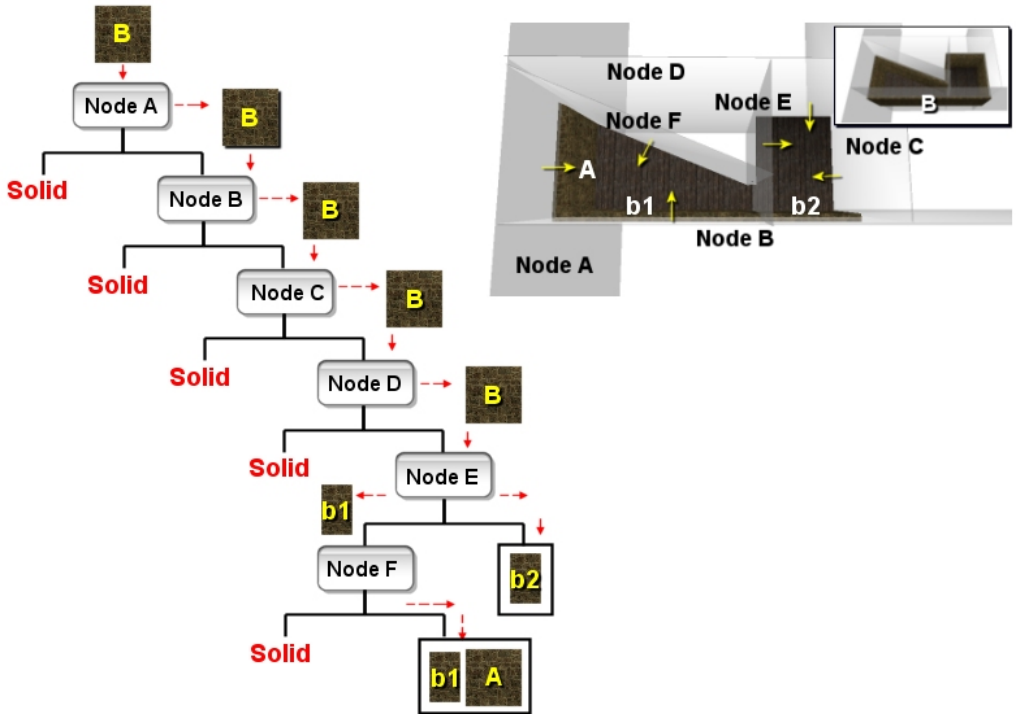


Figure 16.84

Looking at the diagram we can see that polygon B is also contained in node C's frontspace so is passed down the front into node D. Once again, node D has polygon B contained in its frontspace, so polygon B is passed into the front child of node D, which is node E.

At node E polygon B is found to be spanning the plane and is therefore split into polygon fragments b1 and b2. The original polygon B is deleted. Polygon b2 exists in the frontspace of node E so is passed into its front child. As no front child exists however this must mean that polygon b2 has entered an empty leaf attached to the front of node E. Polygon b1 is passed down the back of node E where it arrives at node F. Polygon b2, when classified against node F is found to be contains entirely in its frontspace and is therefore passed down the front of F where it gets added to the leaf that exists there. Polygons A and b1 now both exist in the empty space leaf attached to the front of node F.

Next we pass polygon C into the tree.

As Figure 16.85 clearly shows, as polygon C is passed down the tree it is found to exist in the front halfspaces of nodes A, B, D and E. This means it eventually gets added to the leaf attached to the front of node E.

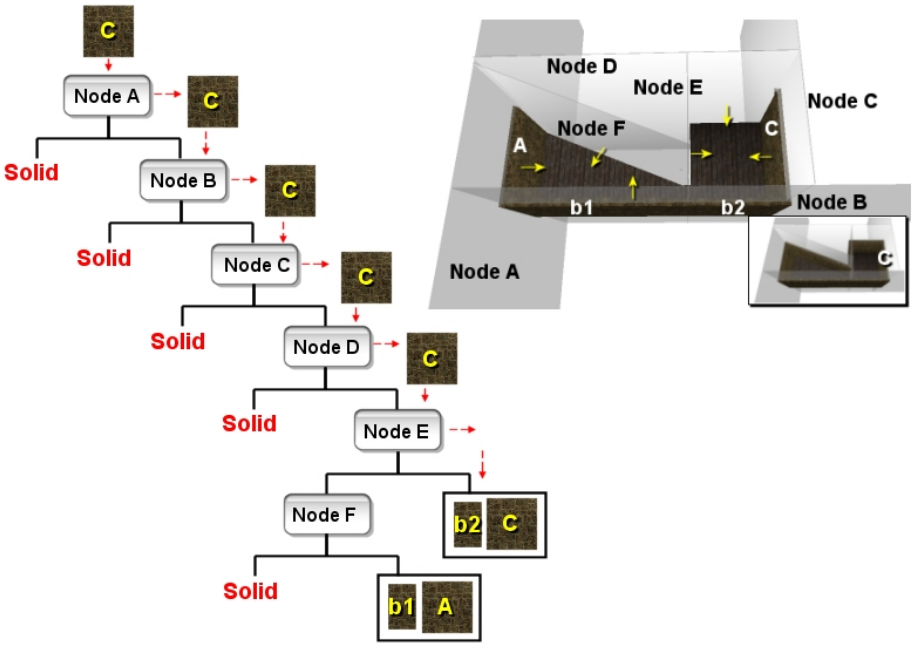


Figure 16.85

Things are looking very good at this point. No polygons have been added to back leaves and so far every polygon we have added has been added to one of two leaves. This is in keeping with what we discovered about this level when we discussed the convexity properties of the BSP tree. You will recall we used this example geometry and showed how the BSP tree would carve the geometry up into two leaves. This certainly seems to be the case. This does not mean that the tree only has two leaves; it means that it will only have two leaves that contain geometry (i.e., two leaves that are situated in front of a node).

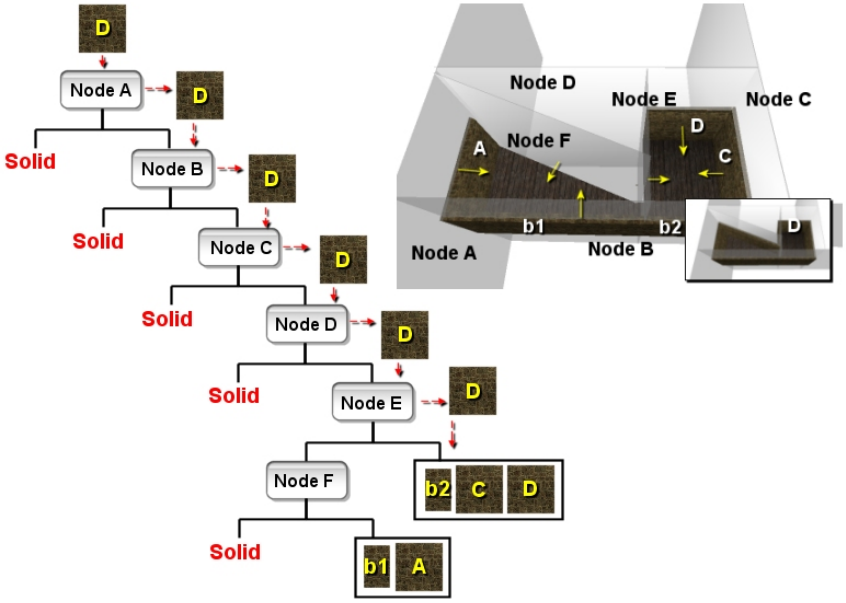


Figure 16.86

Passing polygon D down the tree we see an identical route being taken. By looking at the geometry in Figure 16.86 we can clearly see that polygon D is in the frontspace of planes A, B, C, D and E which results in the polygon being added to the leaf in the frontspace of node E. In the tree diagram we can see that this is the exact classification that happens when polygon D is passed down the tree. Polygon D is added to a front leaf which already contains polygons b2 and C.

Figure 16.87 shows the route that the penultimate polygon (E) takes though the tree. It is classified in the frontspace of nodes A, B, C, D, and E. Of course, node E is the node that was created from this polygon during the building phase and as such, we know it will get passed down the front of such a node. This also clearly demonstrates how this forces polygon E to

eventually be assigned to an empty space leaf that the node plane normal (and the polygon normal) is facing into.

We can see at this stage that the front leaf of E is complete and contains all the polygons that bound that leaf and face into it. All the polygons for this leaf are shown in the square inset in Figure 16.87.

Finally we have polygon F to send through the tree whose route and eventual leaf placement is depicted in Figure 16.88.

Polygon F is found to be in the frontspace of nodes A, B, C and D and is therefore passed down the tree into node E. At node E the polygon is found to be in the backspace of the node, so is passed down the back of E into node F.

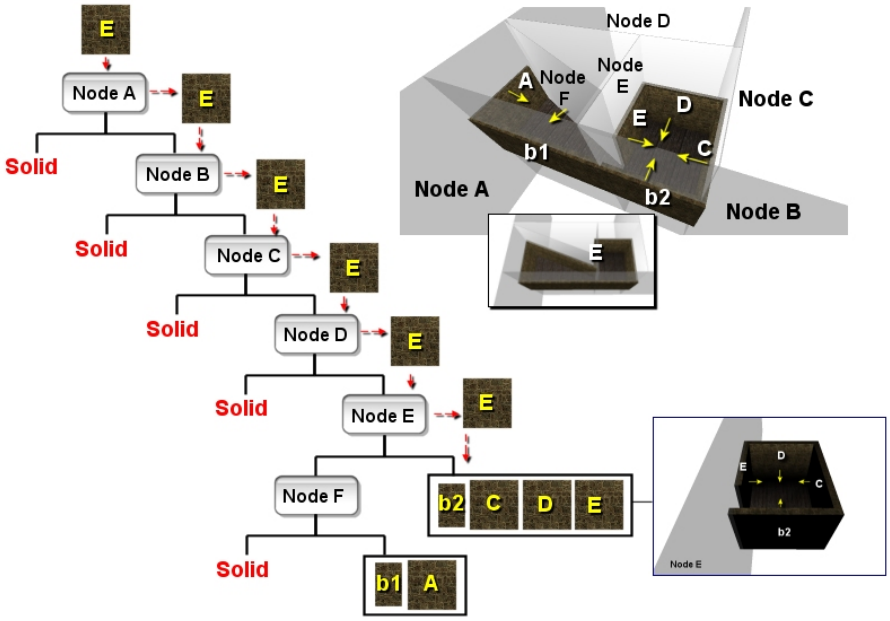


Figure 16.87

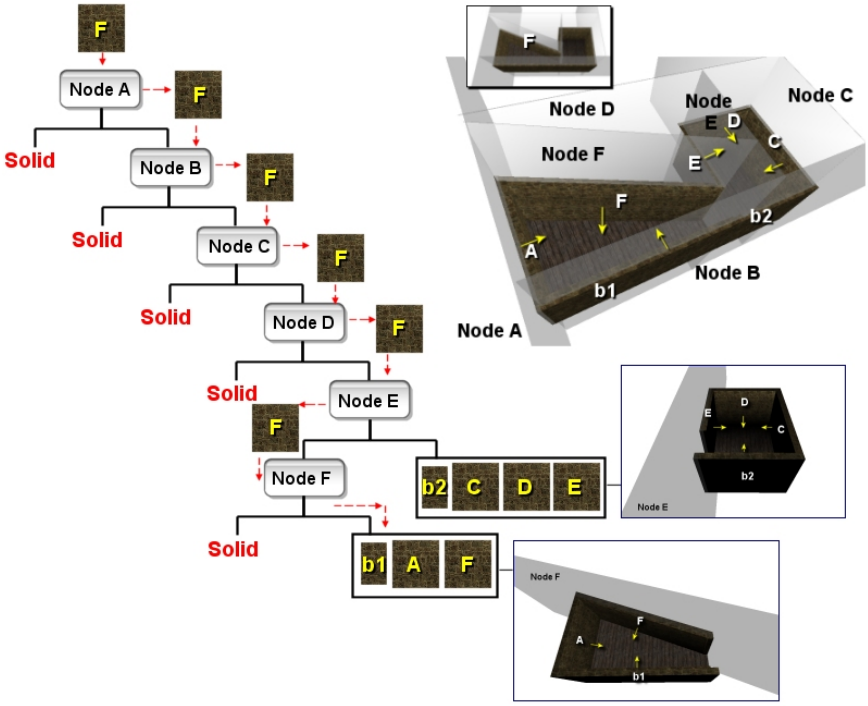


Figure 16.88

At node F, polygon F is clearly going to be found to be co-planar and same facing with the node since this was the polygon that was used to create this node. As we know, a polygon is always passed down the front of a node whose plane it was used to create, ensuring that the polygon ends up in an empty leaf which the node plane normal is facing into. We can see that this is the case in Figure 16.88. At node F, polygon F is passed down the front where it is added to the leaf that exists there. This leaf contains polygons A, b1, and now F, forming the second convex area that the non-convex geometry has been broken into.

Figure 16.88 shows our final tree which has been compiled such that if ever we traverse into a back leaf, we know we have reached solid space.

Figure 16.89 depicts our compiled tree along with three positions labeled A, B, and C. We can see in the top right image that points A and B are in solid space because they lie behind the polygons and have no line of sight with the front of any polygons in the level. Point C is in empty space as it is contained within the rightmost populated leaf. The tree diagram demonstrates what happens when we drop these three points through the tree and track their progress as they are classified against each node plane and sent down the front or back of each node depending on the classification result. As we can see, the tree gives us the correct result and verifies not only that two of these points are in solid space and one of them is in empty space, but it also tells us exactly in which solid and empty regions of the scene these points reside.

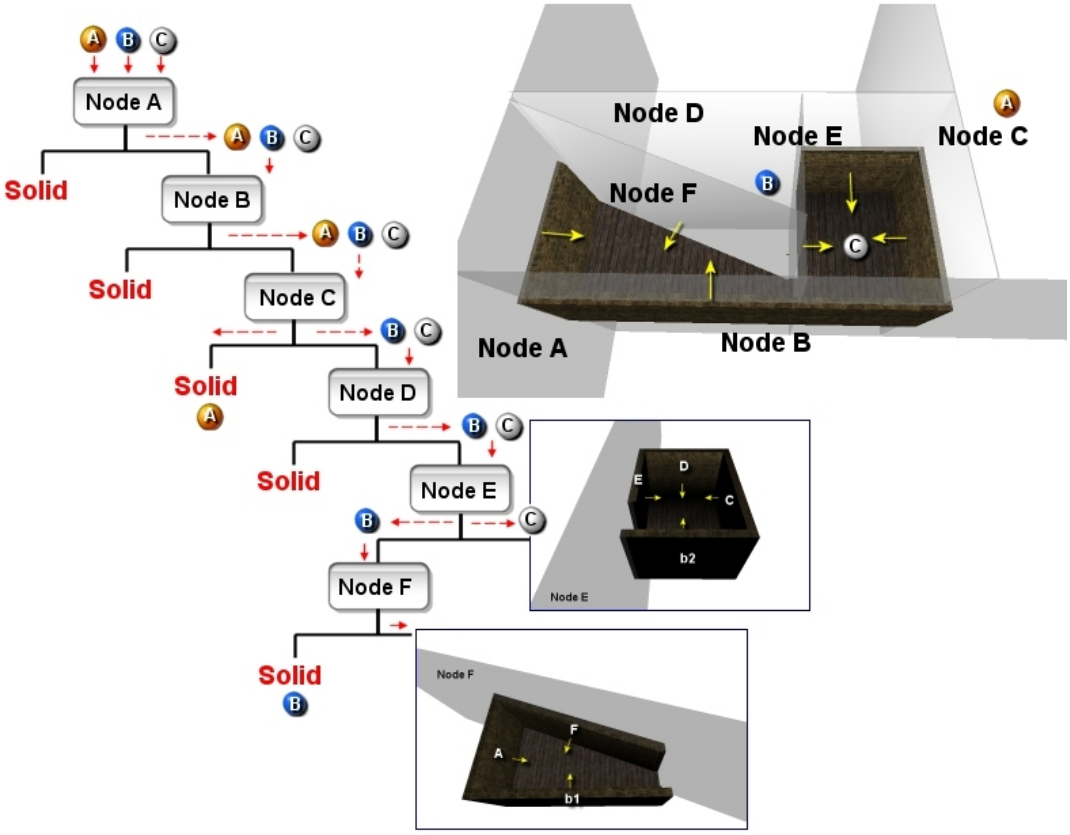


Figure 16.89

You should now be able to understand the above diagram and understand the reasons for each point's journey through the tree. As the tree shows, there are only two empty regions in this scene in which the player would normally be allowed to be located -- the empty leaf bounded by inward facing polygons b2, C, D and E (down the front of node E) and the empty leaf in front of node F bounded by inward facing polygons A, b1, and F. We can think of the polygons as providing the barrier between empty space and solid space and the means by which we stop one flooding into the other. If illegal geometry is supplied to the compiler that is exactly what will happen. We will end up with areas of space that are ambiguous because they exhibit traits that would classify them as being both solid and empty space. We

will look at some examples of illegal geometry in a short while and examine how it corrupts the solid/empty integrity of the tree.

16.3.4 Building the BSP Solid Leaf Tree Compiler

In our discussion so far we have discussed (for educational purposes) compilation of a leaf tree in two different stages. In the first stage we used the input polygon list to build the tree of nodes in the exact the same way we did when constructing our BSP node tree earlier in the lesson. However, we have assumed that once the polygon has been selected for a node it is discarded from the list. At the end of the first stage, we will have built an empty BSP tree (i.e., a BSP tree which contains no polygon data in its leaves). We then looked at how we could collect these polygons at the leaves during a second pass, where each polygon is sent down the tree and clipped to the nodes. The polygons eventually pop out at leaf nodes and these are the nodes to which the polygons are assigned. Of course, there is no reason to do this all in two separate passes as doing so would be inefficient. When constructing the tree, we have the polygon data at our disposal, so there is no reason why we cannot pass the polygons down the tree, clip them to the nodes, and collect them at the leaf nodes during construction. This allows us to implement the entire tree building process along with its static data storage in a single pass. This is how a BSP compiler usually operates and is how the BSP compiler that we will create in Lab Project 16.2 will operate.

Collecting the polygons at the leaf nodes during the build process has much in common with the way we constructed our quad-tree, oct-tree and kD-tree in the previous lessons. When constructing such trees, we generated an axis aligned split plane (or multiple split planes) at each node and divided the polygon list passed into that node into 2, 4, or 8 sub-lists depending on the type of tree being constructed. In the case of the kD-tree for example, at each node all of the polygons would be divided into two lists to be passed into the front and back child nodes. This same process continued until we reached a terminal node and all of the polygons that make it into that node were stored in a leaf structure that was attached to that node.

The same is going to be true with the BSP tree, although we have other things to consider, since the polygon list that makes it into each node is also being used to select split planes. In a nutshell, once a polygon has been selected as a splitter and a node plane created from it, it must still be passed down the tree, classified and potentially split against the remaining nodes in the tree (i.e., what was our ‘second pass’ earlier). The polygon that was used to create the root node for example, might be passed down the tree and split into multiple fragments, each assigned to a different leaf node. This is no different from how a polygon is passed down the kD-tree and repeatedly clipped to the nodes of the tree resulting in multiple fragments during the build process. However, what we must be mindful of is that once a polygon has been used as a split plane, it must never be selected as a splitter further down the tree by another node. This was not an issue in the node tree because as soon a polygon was selected to create a node plane, it was removed from the list and stored in the node. However, because we are continuing to pass the polygon data down the tree, we must make sure that it is not selected again. Furthermore, we must also make sure that if a polygon has been used as a node plane and later on down the tree it gets split by another node plane into two new child fragments, those child fragments must also not be used as splitters. This is obviously not very difficult to achieve; we can simply add a Boolean member to our

CPolygon structure called ‘BeenUsedAsSplitter’ (for example) which is set to true once the polygon has been used as a node plane. When this is passed down the tree into child nodes, each child node, when selecting a polygon from its input list to use as a split plane, will ignore any polygons in the list that have their ‘BeenUsedAsSplitter’ Booleans already set to true. This will prevent the polygon from ever being used as a split plane again until it eventually pops out in an empty leaf and is stored. If the polygon gets split further down the tree, we can simply copy the ‘BeenUsedAsSplitter’ status of the parent polygon into the two new child polygons so that they too will not be used as splitters as they continue their route down the tree. In fact, we must make sure that is the case for *all* co-planar same-facing polygons. That is, if a polygon is selected as a node plane, we must flag any polygons in the list that are co-planar and same-facing as having been used as a splitter too, so that they are not used to create additional planes down the tree. Since all co-planar polygons would generate the same split plane, it would be redundant to do otherwise.

In our final BSP compilation example we will bring these processes together to show how a piece of legal geometry can be compiled into a solid BSP leaf tree in a single pass.

In the leftmost image in Figure 16.90 we see a top down view of a simple scene (a room containing two triangular pillars). Once again, back face culling has been disabled so that we can see all the faces in the level and the yellow arrows depict the normals of each polygon and the node planes that they will create. The four outer walls form the walls of the room itself. These are facing into the room and as such, space behind these walls is considered solid space. In the center top section of the room there are two triangular constructs that are supposed to be simple pillars. These polygons all face out from the center of the pillar into the empty space of the room and as such, the areas bounded by the back faces of these pillars is naturally considered solid space. In the leftmost image we have placed a red ‘S’ to depict where solid space is assumed to be. In the rightmost image we show what this room might look like were the camera to be placed inside and we can see a portion of the two solid pillars at the far end of the room. A floor polygon has also been added in the rightmost image to give a better idea of what the room would ideally look like. However, to simplify the number of nodes we have to draw in our diagrams, we will ignore the floor and ceiling polygons and compile only the walls.

Figure 16.90

Each polygon in the leftmost image is labeled A through H, which will use to demonstrate the order in which each polygon will be selected to create a node plane at each step during the compilation process. As we have discussed, the polygons can be selected in any order and a valid tree will still be created. Of course, the number of leaves and polygons that result in the tree will be different, but the solid and

empty space will be retained and each leaf will represent a convex area described by the parent planes that intersect to form the boundary of that region.

Before looking at the tree diagram and following the compilation process step-by-step, Figure 16.91 demonstrates how the scene will be carved if the nodes are selected in the above order. It also shows where the empty leaves will be located, and can be used as a reference when we discuss the tree diagram that follows it.

Figure 16.92 depicts every step of compiling this level into a solid BSP tree and collecting the polygon data at the leaf nodes in a single phase.

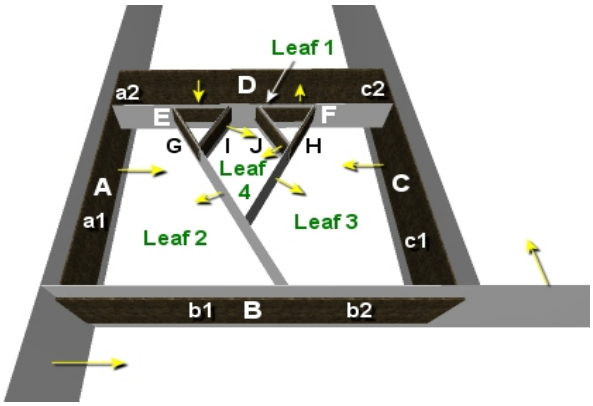


Figure 16.91

The polygon list passed into to the compiler contains polygons A through J. Polygon A is selected as the splitter first and its plane stored in the root node. Polygon A (along with any co-planar polygons) is marked as having been used as a splitter. Polygons that have already been used as splitters are highlighted red in Figure 16.92.

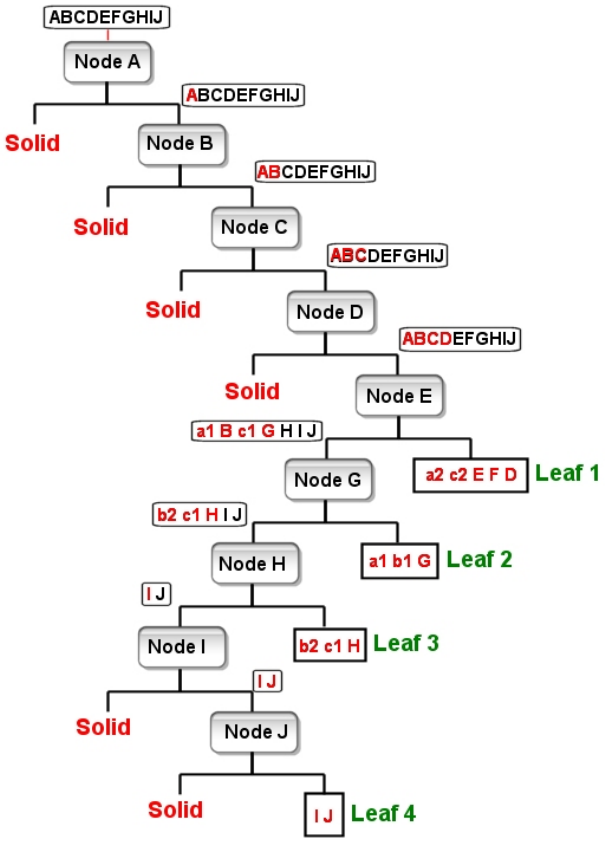


Figure 16.92

Every polygon classified against node A is found to exist in its front space, added to the front list, and passed down the front of the node. As no polygons made it into the back list of node A an empty leaf is created there, which we know will represent solid space. At the front child of A, a splitter is selected so a choice is made from polygons B through J. Polygon A is ignored by the splitter selection process as it has already been used. Polygon B is selected in this example and is marked as having been used as a splitter. All the polygons (**including A**) are classified against node B and are added to the front list. No polygons exist in the back list of node B so an empty (solid) leaf is created there. The polygons in the front list prompt a new child node to be created down the front of B and this time the selection for a splitter ignored polygons A and B as they have already been used. This process continues down to node E where we can see at this point, nodes A,B,C,D and E have been used as splitters. At node E, polygons A through J are classified against the node plane and two lists are created. Furthermore, polygons A and C span node plane E and are therefore split into polygons a1,a2 and c1,c2 respectively. The back list of node E contains polygons a1, B, c1, G, H, I and J and the front list contains polygons a2, c2, E, F, and D. Notice however that because polygons A and C have already

been used as splitters before they were split, the status of their parents is carried over into the children. Polygons a1, a2, c1 and c2 are all highlighted red, indicating they have already been used as splitters and should not be selected again. We can see in Figure 16.92 that when the front list is passed down the front of node E, every polygon in this list has already been used as a splitter and therefore, we have no more nodes to create down this side of the tree. When this is the case we know it is time to create a leaf and that every polygon in this list must lay on the boundary of a convex area. These polygons are added to Leaf 1 which is attached to the front of node E.

The back list of node E contains polygons a1, B, c1, G, H, I and J. These have not yet all been used as splitters so a new child node still needs to be created down the back this node. From this list, only polygons G, H, I, and J are considered as node plane candidates as they are the only ones that have not yet been used as splitters. In this example, node G is chosen next which splits polygon B into b1 and b2 as shown in Figure 16.94. When all the polygons are classified against G we end up with a front list containing polygons a1, b1 and G and a back list containing polygons b2, c1, H, I and J. As all the polygons in the front list have been used as splitters a new leaf is created down the front of node G and polygons a1, b1 and G are added to it.

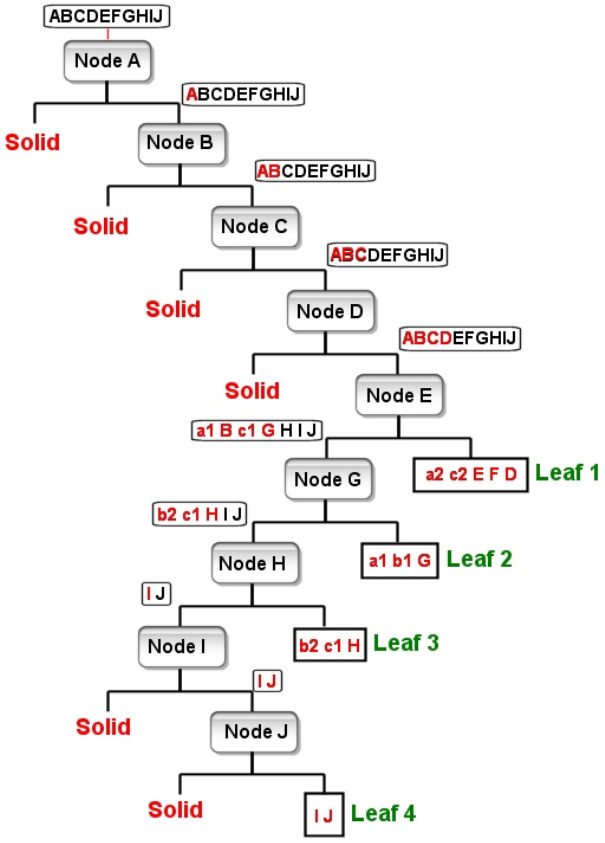


Figure 16.93

In the back list of node G we have three polygons which have not yet been used as splitters: H, I and J. Polygon H is selected next. When the remaining polygons (b2, c1, H, I and J) are classified against node H we end up with a front list containing polygons b2, c1, and H which have all been used as splitters. This means that these polygons are added to a new leaf (leaf 3 in the diagram) which is attached to the front of node H.

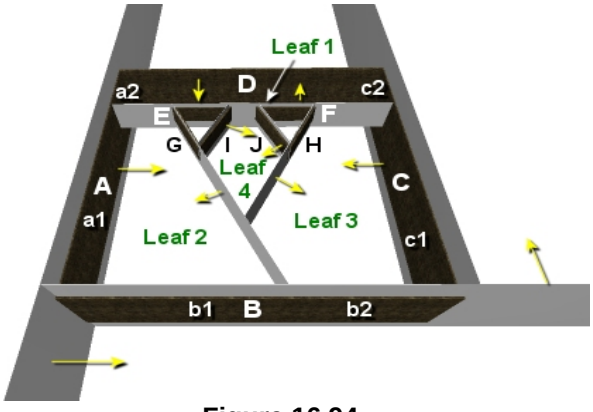


Figure 16.94

The back list of node H contains polygons I and J which have not yet been used as splitters. Polygon I is selected next to create node I. Both of these polygons exist in the frontspace of node I, so a solid leaf is created down the back of node I. The polygon list passed into the front of node I contains polygons I and J and only J has not yet been used as a splitter. Thus J is selected next and marked as having been used. These polygons are then classified against node J and both found to exist in the front halfspace. This means the creation of a solid leaf behind node J. As the polygons in the front list have now all been used as splitters we have no

more nodes to create and polygons I and J are added to a new empty leaf which is attached to the front of node J.

We have now just compiled the complete solid leaf BSP tree on paper. Before moving on, study Figures 16.93 and 16.94 and make sure you fully understand how the tree was constructed and where the solid and empty areas are. Notice how the polygons contained in the leaves in Figure 16.94 marry up with the polygons that are shown in the front leaves in Figure 16.93. Notice as well that empty space is always the region of space that has the polygon normals facing into it and how solid space is always separated from empty space by the backs of polygon along its boundaries.

Let us now examine some placeholder code that could be used to compile a BSP tree. For the exact code, please consult the accompanying workbook and lab project. The code shown here is just to demonstrate the basic concepts and logic that must be employed in a solid leaf BSP compiler.

This demonstration code assumes that a class exists called CBSPTree and that we are showing only the details of the compilation functions. It is assumed that before the CBSPTree::CompileTree function is called by the application, all static polygon data that should be compiled into the tree has been registered with the tree and added to the tree's linked list of CBSPPolygon structures pointed at by the member variable m_pFaceList. The CBSPPolygon structure is assumed to have a 'UsedAsSplitter' member that will be set to false for all input polygons prior to the commencement of the compilation process. As discussed, this Boolean member is set to true when a polygon is used as a node plane or if it is found to be co-planar and same facing as a node plane.

Finally, it should be noted that just because the leaves of the BSP tree (and the nodes) no longer represent regions that can tightly fit into an axis aligned bounding box, we can still store AABBs at each node and use the same hierarchal bounding box tests for hierarchical visibility determination and collision querying as implemented in the previous lesson. It should also be noted that because the leaves are no longer box shaped volumes, the bounding box we store at the node will no longer tightly bound the leaf in the typical case.

Imagine for example a leaf that was shaped like a view frustum (see Figure 16.95). We can see that the bounding box that was compiled for the leaf would be a loose fit. It is possible during a collision query for example, that the collision volume would intersect the bottom near corner of the leaf box (the minimum world extents corner of the leaf's bounding box). In such a case the leaf itself would not be inside the query volume (only its box would) but it would return true for a collision. In such a case, the polygons in the leaf would be tested needlessly for collision. However, this is certainly preferable to testing all the polygons in all the leaves and fits in with the visibility system and the collision querying system we have used for the other tree types.

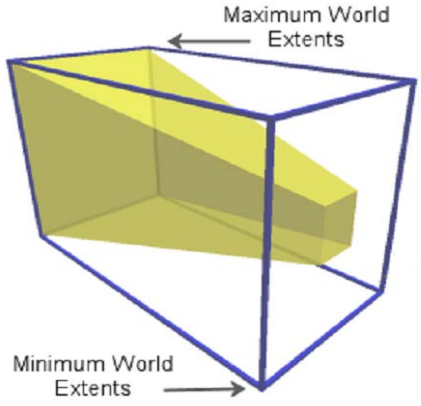


Figure 16.95

The top level function in this example is called CompileTree which simply creates the root node and invokes the recursive compilation process by calling the BuildBSPTree function. This function is passed the newly allocated root node and a pointer to the head of the polygon linked list.

```
HRESULT CBSPTree::CompileTree( )
{
    // Validate values
    if (!m_pFaceList) return BCERR_INVALIDPARAMS;

    m_pRootNode = new CBSPNode;

    // Compile the BSP Tree
    BuildBSPTree( &m_pRootNode, m_pFaceList));

    // Success
    return BC_OK;
}
```

The BuildBSP tree function has the task of populating the node with a separating plane that will be chosen from the list of polygons passed into the node. Only polygons in the list that have not yet been used as a splitter will be considered as a separating plane for this node.

```
HRESULT CBSPTree::BuildBSPTree( CBSPNode *pNode, CBSPPolygon * pFaceList )
{

    CBSPPolygon     *TestFace  = NULL, *NextFace      = NULL;
    CBSPPolygon     *FrontList = NULL, *BackList      = NULL;
    CBSPPolygon     *FrontSplit = NULL, *BackSplit    = NULL;
    CBSPPolygon     *Splitter   = NULL;
    CLASSIFY      Result;
    int           v;

    // Select the best splitter from the list of faces passed
    Splitter = SelectBestSplitter( pFaceList,
                                   DEF_SplitterSample,
                                   DEF_SplitHeuristic);

    if (!Splitter) { Error has occurred so do cleanup here }
```

We saw earlier in the node tree discussion that the SelectBestSplitter function is passed a list of polygons from which it will select one that should be used to create the node plane for the current node. The final two parameters to this function control how many polygons should be sampled in this test and the ‘splits versus balance’ heuristic that should be used when selecting the polygon. In the above code, the final two parameters to this function are assumed to be variables or #defines that you would like used for these values. The SelectBestSplitter function will have been slightly modified from the node tree version in that it must now ignore any polygons in the passed list which have previously been used as splitters (i.e., have their ‘UsedAsSplitter’ Booleans set to true). However, they will still be classified against each potential splitter candidate and contribute to the scoring of the balance versus splits heuristic. After all, just because a polygon has been used as a split plane does not mean that it will not get split into multiple fragments by node planes selected further down the tree during the building process. We will not show the modified code here to the SelectBestSplitter function as the full source

code will be discussed in the accompanying workbook. However, just know that all we have done is modified the outer polygon loop so that any polygons that have been used as splitters are not considered for a new split plane.

The function returns a polygon. In this example code we will assume that each CBSPPolygon also stores the polygon's plane and as such we can simply fetch the plane from the polygon and store it in the current node being visited, as shown below. We also set the returned polygon's (Splitter) UsedAsSplitter Boolean to true so it, or any fragments it gets split into, will never be used to create another node.

```
    // Flag as used, and store plane
    Splitter->UsedAsSplitter = true;
    pNode->Plane = Splitter->Plane;
```

Now that we have the separating plane stored at the node our next task is to loop through every polygon in the list and classify it against this plane and add it to the front or back list for this node depending on the classification result. Any polygons that span this plane will be replaced by two split fragments that exist in each halfspace and will be added to their respective lists.

In the following code we first set up a loop to iterate through each polygon in the linked list. The polygon we are current testing against the node will be stored in the local variable 'TestFace'. We also store its plane in 'Plane' for easy access. Notice that (just as we did in the node tree) we also store a temporary pointer to the next polygon in the list so that we still have access to the next polygon to be processed even when the current polygon being processed is removed from the list. This is important as currently the only access we have to the next polygon in the list is via the current polygon's next pointer (TestFace->Next). If TestFace gets deleted from the list during a split operation or a leaf assignment, we will still need to access the next polygon.

```
    // Classify faces
    for (  TestFace = pFaceList;
           TestFace != NULL; TestFace = NextFace, pFaceList = NextFace )
    {
        // Store plane for easy access
        CPlane Plane = TestFace->Plane;

        // Store next face, as 'TestFace' may be modified / deleted
        NextFace = TestFace->Next;
```

Now it is time to classify the current polygon (TestFace) against the node plane to find out whether it is in the front or back halfspace, spanning, or on-plane with the node. If the current polygon's plane is the same as the plane of the polygon that was selected as a splitter, then we know we have a polygon that is on-plane. Otherwise, we classify the polygon against the plane as we normally do by using a function called ClassifyPolygon which accepts the plane we wish to classify against (which in this case is the plane that was used as the node plane) and the vertices of the current polygon being tested.

```
        // Classify the polygon
        if ( TestFace->Plane == Splitter->Plane )
        {
            Result = CP_ONPLANE;
```

```
        } // End if uses same plane
        else
        {
            Result = ClassifyPolygon( SplitterPlane ,
                                      TestFace->Vertices,
                                      TestFace->VertexCount,
                                      sizeof(CVertex));

        } // End if differing plane
```

Now we have the classification result of TestFace stored in the Result local variable. This will contain CP_ONPLANE, CP_SPANNING, CP_FRONT, or CP_BACK. Next we will enter a switch statement and take the appropriate action in each case.

If the result is CP_ONPLANE then we know that the vertices of the test polygon lay on the node plane but we do not know yet whether this polygon faces into the same frontspace as the node or has an opposing normal. As discussed several times throughout this lesson, this is important because if the normal faces in the opposite direction, it should be added to the back list. Otherwise, it should be added to the front list. In the next section of code we assume the implementation of a function called 'SameFacing' which compares the two normals passed in as parameters and returns true if they are facing into the same halfspace.

```
        // Classify the polygon against the selected plane
        switch ( Result )
        {
            case CP_ONPLANE:

                // Test the direction of the face against the plane.
                if ( SameFacing( Splitter->Plane.Normal , pPlane->Normal ) )
                {
                    // Mark matching planes as used
                    if (!TestFace->UsedAsSplitter)
                    {
                        TestFace->UsedAsSplitter = true;

                    } // End if !UsedAsSplitter

                    TestFace->Next  = FrontList;
                    FrontList       = TestFace;
                }
                else
                {
                    TestFace->Next  = BackList;
                    BackList        = TestFace;
                } // End if Plane Facing
                break;
```

As you can see, we pass in the node plane normal and the normal of the test polygon and if found to be equal we know we have a co-planar same-facing polygon. When this is the case we set its

UsedAsSplitter Boolean to true since using this polygon as a splitter further down the tree would be redundant as the scene has already been divided by the same plane. We also add the polygon to the front list by assigning the test face's next pointer to point at the current head of the list and then reassign the pointer to the head of the list to point at the test face. Essentially, we are just adding the polygon to the head of the front list. Notice in the above code however that if the normals are not facing in the same direction in the on-plane case, the polygon is added to the back list and its UsedAsSplitter status is unaltered.

The CP_FRONT and CP_BACK cases are delightfully simple. If the polygon is located entirely in the frontspace of the node then we just add its pointer to the front list of polygons being compiled. If the polygon is located in the backspace of the current node then we add it to the head of the back list currently being compiled for this node.

```
            case CP_FRONT:
                      // Pass the face straight down the front list.
                      TestFace->Next            = FrontList;
                      FrontList                 = TestFace;
                      break;

            case CP_BACK:
                      // Pass the face straight down the back list.
                      TestFace->Next            = BackList;
                      BackList                  = TestFace;
                      break;
```

When a polygon is found to be spanning the node it must be split into two child fragments just as was the case with the node tree. However, after we have created the two new child fragments, we must set their 'UsedAsSplitter' Booleans equal to the parent polygon status prior to the parent polygon being deleted. This will make sure that if the parent polygon has already been used as a splitter, this state is inherited by the children so that they are not selected as splitters later. As discussed, this would be redundant and would create many unnecessary nodes. Shown below is the CP_SPANNING case and the final section of the polygon classification loop.

```
            case CP_SPANNING:

                // Allocate new front fragment
                FrontSplit          = CBSPPolygon;
                FrontSplit->Next    = Frontlist;
                FrontList           = FrontSplit;

                // Allocate new back fragment
                BackSplit           = new CBSPPolygn;
                BackSplit->Next     = BackList;
                BackList            = BackSplit;

                // Split the polygon
                TestFace->Split( Splitter->Plane, FrontSplit, BackSplit);

                // Copy over status of parent into children
                FrontSplit->UsedAsSplitter = TestFace->UsedAsSplitter;
                BackSplit->UsedAsSplitter  = TestFace->UsedAsSplitter;
```

```
                // Remove original polygon
                delete TestFace;

                      break;

            } // End Switch

      } // End while loop
```

As you can see in the above code, two new polygons are created, FrontSplit and BackSplit. These are fed into the parent polygon's Split method along with the node plane. When the split function returns, FrontSplit will contain the polygon fragment that exists in the frontspace of the node plane and BackSplit will contain the fragment that exists in the backspace. Before deleting the parent polygon, we copy the value of its UsedAsSplitter Boolean into each of the children.

When the while loop shown above exists, we will have compiled a front list and a back list of polygons describing exactly which polygons should be passed down the front and back tree of the current node.

The first thing we do is test to see if the back list has any polygons in it and whether any of them have still not yet been selected as splitters. If there are still polygons in the back list which have not been selected then all is well and we know we have to create a new back child node and keep on creating nodes from this list. If the back list has a list of polygons which have all been used as splitters then we have a real problem. Under normal circumstances (were we compiling a basic leaf tree) we would just store the polygons in a back leaf, and we could certainly do that. However, we have also discussed that if we have been passed legal geometry, this situation should never occur. That is, back leaves will always be empty and we should never be in a situation where we have a list of polygons that have all been used as splitters existing at the back of a node. When this happens, we have been given illegal geometry and we run the very real risk that the solid/empty information will be corrupt. Now, normally if the artist has created the level without regard to the technology being used, then the level will be highly illegal and there is not much you can do except modify the source level to get rid of the offending geometry (we will discuss such techniques in a moment). However, it is sometimes the case that the geometry was defined in such a way that it is technically legal but due to floating point accumulation errors or perhaps slightly sloppy object placement by the artist, a polygon's normal may face into a solid leaf. When this is the case, it often means this small sliver of polygon which has ended up in a back leaf can simply be deleted as it exists within the solid space of another object. This is the approach we take in this example. If any polygons end in back leaves, we will assume it is a floating point accumulation error and simply delete the offending fragments. However, while this works very well for compiling geometry that is essentially legal but with a few minor issues, geometry that has been assembled without any concern for solid/empty legality cannot be fixed by this simple solution. If the geometry is extremely illegal then this method could well delete half the level.

> **Note:** It is important that your project artist be aware of the rendering technology you are using and design the artwork in a compliant manner. Although we will discuss techniques for correcting illegal geometry in the next section, as well as gain a greater understanding as to what causes it, the artist must take responsibility for developing assets that can be used by the rendering solution you ultimately decide to employ.

In the following code we remove any illegal fragments that end up facing into back leaves. This assumes the implementation of a function called 'DeletePolyList' that, when passed the back list, will delete all the polygons contained within.

```
     // If No potential splitters remain, free the back list
     if ( BackList && CountSplitters( BackList ) == 0 )
     {
        // Remove illegal faces
        DeletePolyList (BackList);
        BackList = NULL;

     } // End if No Splitters
```

We can really not stress enough that the above section of code is absolutely no substitute for correct solid/empty level design. It fixes problems in situations where floating point inaccuracies introduced by the myriad of clipping operations that can be done on a polygon during the compile process cause its vertices to drift off the original plane and end up in solid space. But it is not a magic wand that can be thrown at any polygon soup with the expectation of compiling a perfect tree.

Now that we have the front and back lists of the current node compiled, we will construct a bounding box to be stored at that node that will bound all the polygons that exist down its front and back side. In the following code it is assumed that the CBSPNode structure has a function called CalculateBounds that, when passed two lists of polygons, will construct its bounding box to be large enough to contain the vertices of all polygons stored in those lists. Although this bounding box will not fit the geometry as tightly as the nodes of a quad-tree or an oct-tree (where the polygons have been clipped into box space regions), they can still be used to coarse cull entire branches of the tree during visibility and collision queries.

```
    // Calculate the nodes bounding box
    pNode->CalculateBounds( FrontList, BackList );
```

In the next step we test to see if any of the polygons in the front list are yet to be selected as splitters. If this is not the case then all the polygons in the front list have already been selected as splitters further up the tree and we have no more nodes to select. We also know that the polygons in this list will lie on the boundary planes of a convex region of empty space (an empty leaf). The polygons will also be facing into this empty leaf so our task is simple: create a new leaf, add the polygons to it, and attach it to the node's leaf member. If there are still polygons in the front list that have not yet been selected as splitters then more subdivision must occur down the front of this node. When this is the case we create a new node, attach it to the current node's Front pointer and recur into that node with the front list.

```
    // If all splitters are used in frnt list create front leaf
    if ( CountSplitters( FrontList ) == 0 )
    {
        // Add a new leaf and store the resulting faces
        CBSPLeaf * FrontLeaf = new CBSPLeaf;
        pNode->Leaf = FrontLeaf;
        CBSPLeaf->AddPolygons ( FrontList );
    }
    else
```

```
    {
        // Allocate a new node and step into it
        CBSPNode *pFrontNode  = new CBSPNode;
        pNode->Front          = pFrontNode;
        BuildBSPTree( pNode->Front, FrontList );

    } // End If FrontList

    // Front list has been passed off, we no longer own these
    FrontList = NULL;
```

When we reach the bottom of the above code, all children down the front of the node will have been created and all the polygons in this node's front list will have had their pointers stored in leaves. This means we should not delete the polygons stored in the front list since their pointers have been copied into other locations; we can simply set the front list pointer to NULL.

What may seem strange in the above code is that we assign the front leaf to a node member called 'Leaf". As a node can have both front and back leaves, should this not be called 'FrontLeaf" instead? Well, we could, but since we know for a fact that our BSP tree will never store polygon data in back leaves, why bother creating empty leaf structures and attaching them to nodes when nothing will ever be stored in them? Instead, we will simply make the node's Back pointer, which normally points to a back child node, dual purpose. If this member is not NULL then it means there is a child node attached to the back of the current node. If this pointer is NULL, it must mean a leaf exists down the back of this node. However, all we need to know is that a solid space leaf is represented by a NULL pointer down the back of a node and we have all the information we need. Therefore, the only leaves we ever actually have to create and store information for are front leaves, which will be pointed to by the node's Leaf member. If traversing the tree we find that we need to traverse down the back of a node that has its back node pointer set to NULL, we know that we have traversed into solid space. Here is the following and final section of the compilation function.

```
    // If the back list is empty flag this as solid
    // otherwise push the back list down the back of this node
    if ( !BackList )
    {
        // Set the back as a solid leaf
        pNode->Back = BSP_SOLID_LEAF;
    }
    else
    {
            // Allocate a new node and step into it
            CBSPNode *pBackNode = new CBSPNode;
            pNode->Back = pBackNode;
            BuildBSPTree( pNode->Back, BackList);

     } // End If BackList

    // Back list has been passed off, we no longer own these
    BackList = NULL;


    // Success
    return BC_OK;
```

```
}
```

As you can see, if there are no polygons in the back list we know there must be a solid leaf behind this node so we set the back pointer to NULL. Otherwise, we need to further subdivide the space behind this node and we create a new node, attach it to the current node's Back pointer, and recur into the back child with the back list.

We have just examined everything involved in writing a solid leaf BSP compiler. This code will generate a BSP leaf tree which can be traversed like any other tree to perform visibility and collision queries. The fact that we have stored bounding boxes at each node also allows us to use the same AABB hierarchical queries on the tree as before, although we may sometimes end up traversing into leaves where the geometry does not intersect the query volume. As discussed earlier, this is because the bounding box of a node will not be a tight fit around its geometry. Therefore, it is possible that a ray for example, may be determined to intersect the bounding box of a leaf (or node) even if the ray does not intersect the convex region of the leaf. Of course, this simply means that we may end up testing polygons in a leaf that are not actually contained in the query volume in some circumstances. The same frustum culling traversal can also be performed on the tree although once again, because of the loose fitting bounding boxes around the nodes, we may find that we traverse into nodes and render some leaves that are not actually contained within the frustum.

When we consider that the polygon aligned solid leaf BSP tree will on average create much larger trees than quad-trees or kD-trees for example, one might imagine why this tree would be favored over our previous tree types. To be clear, if all we were going to do is plug our solid leaf BSP tree into the leaf bin rendering system we developed in the previous lesson, this would be a valid point. The BSP tree would be outperformed by those other trees in the typical case. However, because the BSP tree contains the solid/empty information, it makes CSG operations possible (discussed later in this lesson) and allows for very efficient line of sight tests to be carried out on large scene databases. Furthermore, and by far the most important feature, is that it allows us to construct a potential visibility set for our level which will increase the performance of our current rendering systems by an order of magnitude (in the case of a highly occluded level). Therefore, in the next lesson we will see how the BSP tree, coupled with the potential visibility set, will be our rendering technology of choice going forward and one of the core technologies that is carried forward into Module III when we start to construct our game engine.

16.4 Illegal Geometry and Hidden Surface Removal

It should be apparent at this stage that the 'solid' aspect of a solid BSP leaf tree is entirely dependant upon the geometry that is built by the artist and ultimately fed into the compiler. If the artist generates scenery in which the various different areas of the scene are not properly closed or bounded, then our ability to determine the correct nature of the space described by each leaf will be lost. While the geometry that is compiled is still a very important factor in whether or not the integrity of the solid/empty information is retained, there are solutions we can employ to ensure that this situation is less of an issue. In this section we will discuss exactly what illegal geometry is and how it can be avoided during the asset development phase. Illegal geometry is caused by a condition that can be summed up as follows:

"When the front of one or more polygons (or some part of them) can see the back of another polygon, we have illegal geometry. This means that the boundaries between solid and empty space have leaked into one another, making solid/empty space determination impossible."

Figure 16.96 shows the simplest case of illegal geometry. We have two isolated polygons positioned such that the front of polygon can 'see' the back of polygon B. If we look at the tree diagram to the right we can see that the following happens:

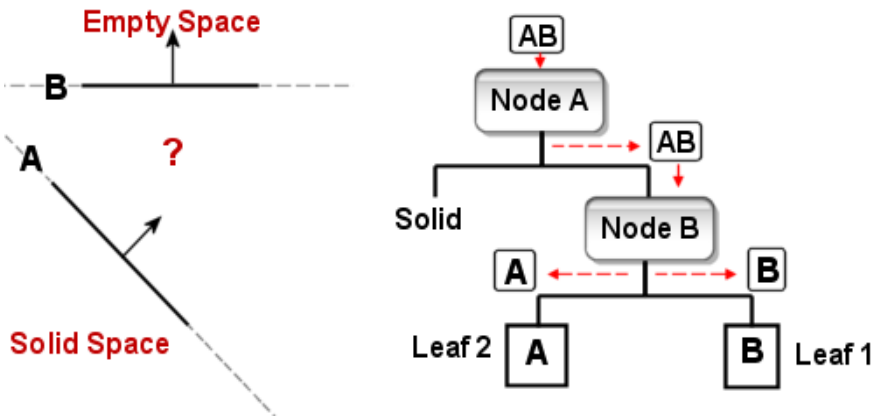


Figure 16.96

At the root node polygon A is selected as the splitter and when classified against this node plane, polygons A and B are sent down the front where B is selected as the next node. Behind node A no polygons exist in the back list so this is correctly identified as a solid leaf. Remember, solid space is the space behind the polygons. At node B we classify the polygons and find that polygon A is added to node B's back list and polygon B is added to the front list. As polygon B has already been used as a splitter this means that to the front of node B is a leaf containing polygon B. This space is correctly identified as empty space. Remember, empty space is the space located in front of the polygons. However, polygon A is passed down the back of node B but has been used as a splitter, so what should we do with it? We could delete it but that would make one of our polygons disappear and would describe the region of space between A and B as being solid space. This would be incorrect however as this space is in front of polygon A and therefore, should be empty space as it is the only space from which polygon A can be viewed. We could alternatively make this a back leaf and assign polygon B to it, but leaves with polygons assigned represent empty space therefore this would identify the region between polygons A and B as empty space. This is not correct either as in this space the viewer can see the back of polygon B, which would be removed during back face culling. The player should never be allowed inside regions of space where the backs of polygons can be seen. As you can see, this situation cannot be resolved and the region of space between polygons A and B cannot be added to any category. This is illegal geometry.

Of course, the above scenario is a little too simplistic as it is highly unlikely that any professional artist would ever construct a level which such glaring geometrical errors. However, there are ways to place geometry in a level that seem very sensible but which still cause illegal geometry, as shown next.

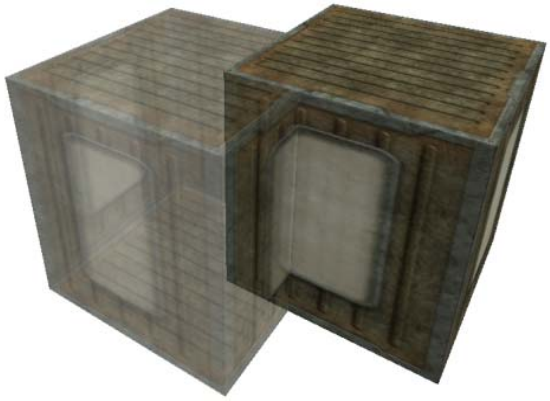


Figure 16.97

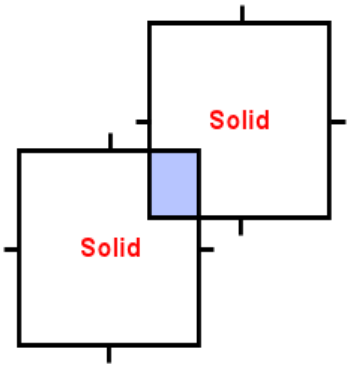


Figure 16.98

Imagine the scenario depicted in Figure 16.97 in which the artist has accidentally (or perhaps on purpose) overlapped two simple box shaped objects. The closest box has been rendered with alpha blending enabled so that we can more easily see that there are portions of the neighboring box intersecting its interior. Figure 16.98 shows the particular problem that our solid BSP leaf tree compiler would have in resolving this type of scenario during compilation. We know by experience that the space inside the two cubes should be defined as solid. However, due to the fact that some of the polygons intersect the interior space of each cube, we find a situation in which that same space which we know to be solid, exists in the front halfspace of one or more polygon fragments. That is, some portions of the polygons making up the boxes are front facing into the solid space of the other crate. This situation is one that we would commonly refer to as a *leak*. This is where solid and/or empty space – as defined by our tree and leaf layout – has 'leaked' into an area in which it should never have existed.

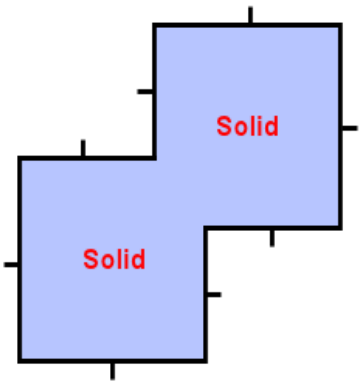


Figure 16.99

Figure 16.99 demonstrates the only possible arrangement of geometry in which this situation can be resolved. We can see here that the fragments of any polygons which are embedded in the solid space of either of the cubes have been removed. In doing, so we remove the ambiguity introduced by the erroneous front faces in solid space.

The process by which we go about removing these intersecting polygon fragments is called *hidden surface removal*. Put simply, this process is intended to remove any polygons that can never possibly be seen because they are contained within the solid space of other primitives in the scene. Not only can this remove a significant amount of overdraw in our application, it also takes care of the problematic situations outlined in this section. Something that you may have heard discussed in the past is the term 'Constructive Solid Geometry' (or 'Boolean Operations'). This term covers a multitude of topics, but in principal these techniques provide us with the means to perform various geometric operations on multiple objects. One example is the ability to merge the volumes of two primitives together in the cases where they might intersect.

In the next section we will see how we can use one of the techniques known as the 'union' operation to resolve many such types of problematic geometric situations prior to BSP compilation. This union operation is analogous to the hidden surface removal process depicted above and will help to ensure that we are able to create an accurate and very stable solid leaf BSP tree. Thus, even if the geometry does

contain such intersecting objects, the hidden surface removal techniques we will implement in this lesson will allow us to remove any polygon fragments that are found to reside in the solid space of another object. The GILES™ world editor can perform hidden surface removal on the entire level if you chose to select all brushes and then perform the union CSG operation on them. This will merge all the meshes in the scene into a single mesh, removing any polygon fragments that lie in solid space. However, if you are developing your levels in a package like 3D Studio MAX™, which does not employ the same HSR-oriented world building techniques, you may find that you create levels with many such hidden surfaces that would cause a BSP compiler to fail. Therefore, the BSP compiler we create in Lab Project 16.2 will also have an HSR processor that can be invoked as a pre-compile step to remove all such hidden surfaces and merge all meshes contained in the file into a single mesh which is then fed into the BSP compiler.

Although HSR techniques can remove hidden surfaces from within the solid space of an object, the individual meshes/brushes comprising the scene must be constructed from legal geometry. As long as all of the individual meshes comprising the scene can themselves be compiled into a solid leaf tree individually, we can perform a union operation to merge all these meshes into a single mesh with all hidden surfaces removed. However, if the individual components making up the scene contain intersecting/hidden surfaces, HSR will fail to fix the problem and the only cure is for the artist to fix this geometry manually.

Before we move into the final section of this lesson and discuss constructive solid geometry (a superset of hidden surface removal) we will look at some examples of legal geometry so that we are clear exactly how easy it is to create. Experience has shown that virtually every problem that a student has getting the BSP tree to operate and query correctly comes down to it being fed illegal geometry, and a misunderstanding of the various situations that cause illegal geometry during object placement in a level. By looking at some common examples of illegal geometry and how they can often be cured with the union CSG operation supported by many world editors and modelling packages, we will get a good feel for what to do and what not to do during the level building process.

Figure 16.100 depicts a scene created in the GILES™ world editor. It is a very simple level comprised of a room containing some pillars. It all looks pretty inoffensive until we examine how the level was created. The room itself was constructed by hollowing out a cube so that in the center of the cube we can paint the inward facing walls, floor and ceiling polygons with textures. All is well so far; the solid space of the room is contained between the polygons that form the interior and exterior sides of each wall. The outward facing polygons of the room can obviously not be seen in this image as we would have to walk outside for this to be the case, but it does not matter for this discussion. The room is perfectly valid geometry. It is a single mesh that respects the solid and empty space rules of construction.

What makes this geometry illegal is the placement of the four pillar meshes. Each pillar is constructed from three meshes: two square blocks that press against the floor and ceiling and a central cylindrical column. The cylinder is also resting up against the blocks. The blocks and the cylinder columns are closed meshes, so let us now see why this is a problem.

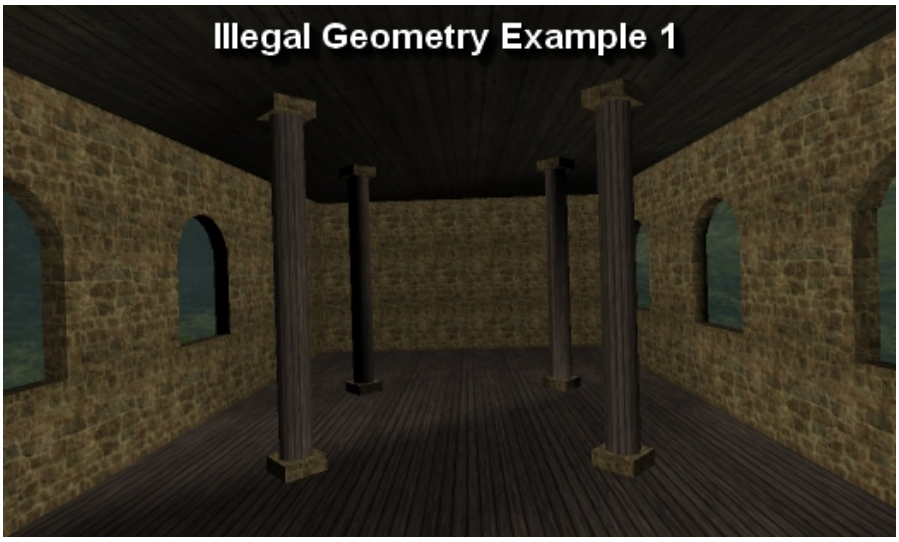


Figure 16.100

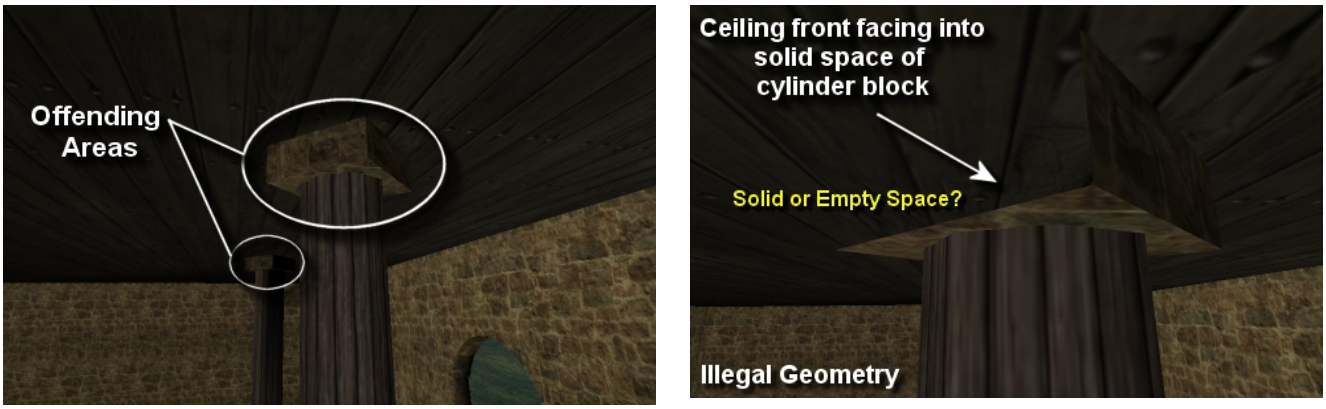


Figure 16.101

Figure 16.101 highlights the first problematic area of this scene. The pillar blocks themselves are pressed up against the ceiling, as shown in the rightmost image. As the top face of the pillar block and the roof polygons are co-planar, we have a situation where a section of the roof polygon is facing into the solid space of the pillar blocks. This is shown in the image on the right where we have removed a side face of the pillar block so that we can see what is happening inside. As you can see, the ceiling polygon is front facing into the solid space of the pillar block, which we know to be exactly what causes our solid BSP tree compiler to fail. Although it cannot be seen here, the top face of the pillar block would also be facing up into the solid space behind the ceiling polygons, so we have illegal geometry on two counts. Obviously, this will be the same situation for every cylinder block in the roof. The cylinder blocks at the bottom of each pillar will also have the same conflict with the floor polygon.

One very inelegant way to deal with this problem is to move the cylinder blocks ever so slightly away from the roof and floors so that empty space can flow between them. You may get away with this, but it certainly is not the answer we are looking for. Apart from being tedious for the artist to manually adjust everything like this, it may also happen that the user of our game will notice that the pillars are floating

in mid air. No, this definitely is not the way to go since this is a limitation we just can not live with. Luckily the hidden surface removal routines we will develop will come to the rescue by performing a union operation between the pillar block and the mesh of the room (which contains the roof polygon). The block will become part of the room mesh and the offending surfaces will be removed.

In Figure 16.102 we show the result of a union operation between the pillar block and the room mesh, which removes these hidden surfaces. As you can see, the section of the ceiling polygon that was contained inside (or co-planar) with the solid space of the pillar block has been carved away and although it cannot be seen here, so too is the top face of the pillar block that was pressed up against the ceiling.

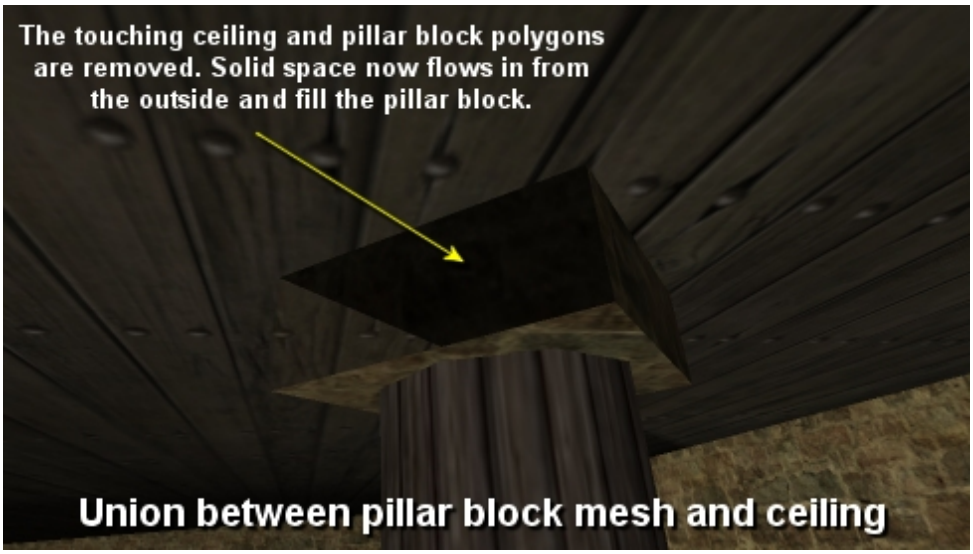


Figure 16.102

The pillar block and the room mesh have now been merged into a single mesh and the hidden surfaces have been removed. Solid space will now flow thorough this hole and fill up the inside of the block. Note that in this image we have removed the left face of the block so that we may observe what is happening inside. Obviously, if we were not to do this we would not notice any difference in the level. It is the *hidden surfaces* that have been removed by the union operation and since these polygons are hidden, we would not be able to visibly see that they have been removed.

Figure 16.103 illustrates where problems still exist. In this image, the front and back faces of the cylinder invisible so that we can see what is happening inside the solid space of the cylinder. As we can see, we have the same situation. This time, the bottom face of the block is co-planar with the top face of the cylinder (not seen here because of back face culling) which means the bottom face of the block is front facing into the solid space of the cylinder and vice versa. This problem can be resolved again by performing a union operation between the column mesh and the room mesh (which now contains the cylinder block after the previous union operation).

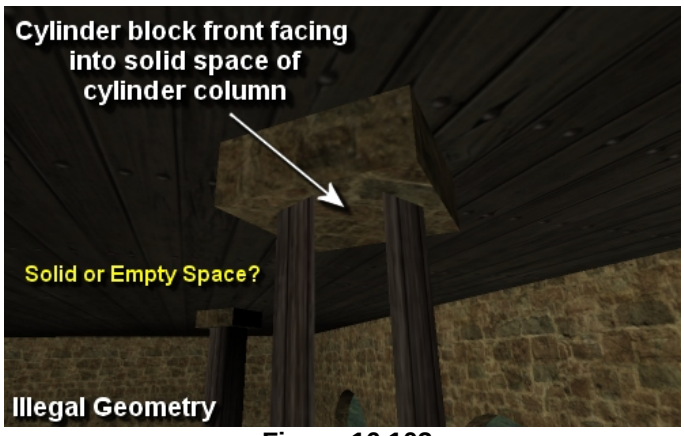


Figure 16.103

The results of this second union operation can be seen in Figure 16.104 where we note that the section of the bottom face of the pillar block that is facing into the cylinder's solid space is removed, and so too is the top face of the cylinder block. The cylinder block, the cylinder column, and the room geometry (walls, floors, ceiling, etc.) have now all been merged into one mesh with hidden surfaces removed.

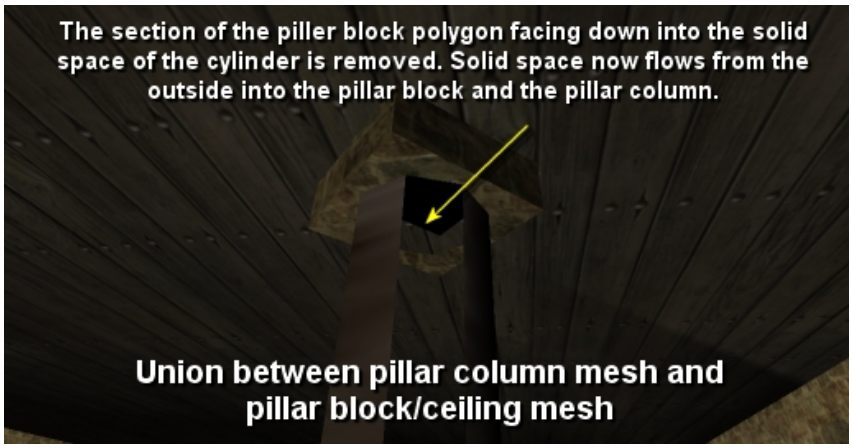


Figure 16.104

Once again we have made the front faces of the cylinder invisible so that we can see what has happened inside its solid space. Notice that the union operation has clipped a hole in the bottom face of the block which matches the dimensions of the cylinder column. Although it cannot be seen here, the top face of the cylinder has also been removed so that solid space flows from the outside, through the cylinder block and into the pillar columns. Solid space is still perfectly bounded by the backs of polygons and cannot leak out into empty space.

As Figure 16.105 shows, even if we select the all the cylinders and the room meshes and perform a union on them all to remove the hidden surfaces and correct these problems, illegal geometry can be caused by simply placing two crates in a room stacked on top of each other. This seems pretty harmless, but of course, if hidden surface removal (union) is not applied to these crates, the exact same geometry problems arise.

Figure 16.105

By making the polygons of the crates transparent in the rightmost image, the problem becomes immediately obvious. The top face of the bottom crate and the bottom face of the top crate are co-planar but facing in opposite directions. The top face of the bottom crate is facing into the solid space of the top crate and vice versa. Therefore, our HSR routines should also merge these two brushes/meshes together using a union operation to remove the offending hidden sections of the surfaces.

Figure 16.106 shows the result of performing a union on these two crate meshes into a single mesh.

We can see by making the faces of the top crate transparent that the entire section of the bottom crate's top face that was pointing into the solid space of the top crate has been clipped away and vice versa. These two crates are now a single mesh which is filled with solid space and whose outward facing polygons bound that solid space.

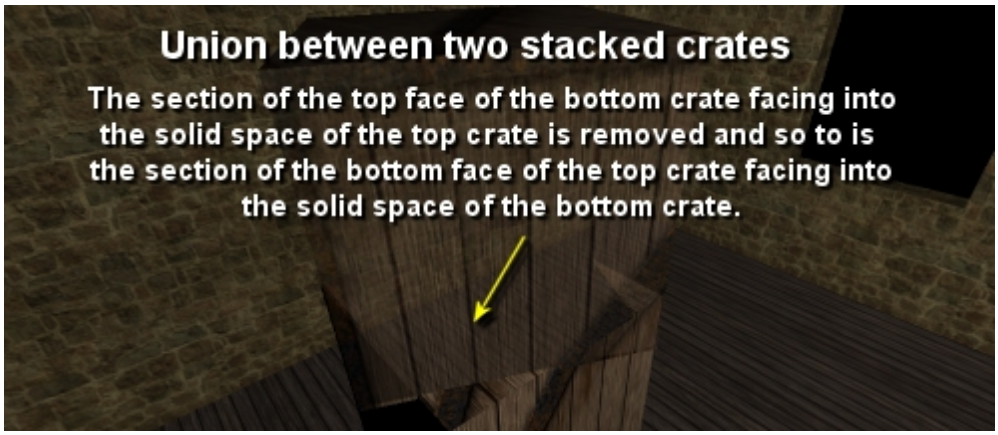


Figure 16.106

Of course, the same problem still exists between the bottom face of the bottom crate and the floor polygon of the room mesh on which it is resting. As Figure 16.107 illustrates, performing a union operation on our two crate mesh and the mesh of the room itself will remove the section of the floor that faces into the solid space of the crates and will remove the bottom face of the crate that faces into the solid space behind the floor.

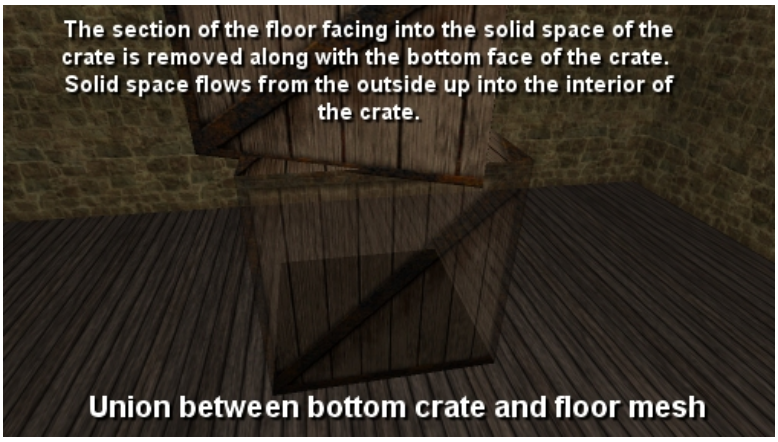


Figure 16.107

So we have seen how easy it is to create illegal geometry but we have also shown how in all these cases, HSR (union) comes to the rescue. The union operation is one of merging all the meshes that comprise the scene into a single mesh removing all hidden surfaces in the process. If the world editor/3D modeler of your choice does not support CSG and the union operation in particular, do not worry. We will discuss how to perform a union operation on your scene in the next section of this lesson and will also implement an HSR module in our BSP compiler in Lab Project 16.2. This HSR module will be invoked prior to the BSP compile to remove any hidden surfaces that exist in a multi-mesh environment.

Finally, before discussing CSG it is important to take a moment to strongly recommend that you spend a good deal of time thinking about which objects in your scene should be considered BSP geometry. Remembering that nearly every polygon you pass into the compiler will be used to create a node, we can imagine how placing a 10,000 polygon sphere in the middle of a room would create 10,000 nodes in the tree, each of which would split the polygons of the level into many tiny slivers. This would make for an extremely large tree that is slow to traverse and high on memory usage. Furthermore, it would raise the polygon count of the scene significantly.

It is quite common for only the core geometry of the level to be compiled into the BSP tree (e.g., walls, floors, ceilings, etc.) and for high polygon objects and room accessories such as furniture meshes and

décor to be assigned to the tree as dynamic/detail objects. For example, GILES™ has a property that allows you to flag a mesh as being a detail object which our BSP compiler will not compile into the tree -- it will simply write it back out to the final compiled file. When our BSP file is loaded, these detail objects can be loaded also and passed down the tree and assigned to the leaves in which they reside. This produces a much smaller and faster tree and is highly recommended in many situations.

> **Important Note:** To keep the tree small and stable, you should only be compiling your core geometry into the level. High polygon models used to populate the scene with furniture or décor should not be compiled and should be assigned to the tree as detail objects at run time.

Bearing this in mind, we might imagine how we could sidestep the entire illegal geometry problem in the previous examples by compiling only the original room meshes into a BSP tree. Once the BSP tree of the empty room has been compiled, the crate and pillar meshes could simply be assigned as detail objects to leaves in which their bounding volumes are found to reside. When these leaves are considered to be visible, the detail objects assigned to those leaves are also flagged as visible and rendered just as was the case with our previous tree types. If we were to take this approach, the only geometry from the previous example that would be compiled into the BSP tree is shown in Figure 16.108.

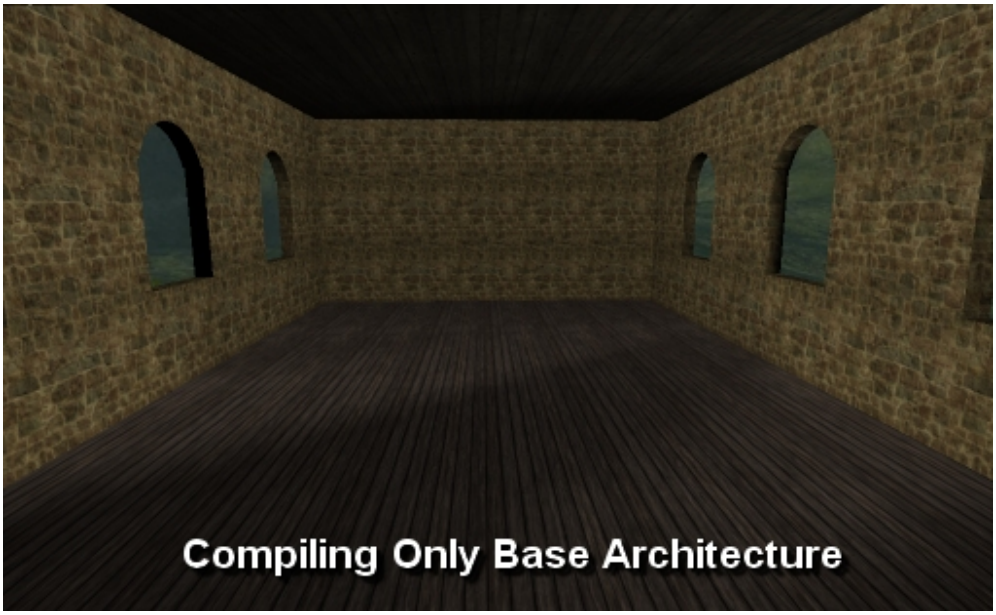


Figure 16.108

Of course, it is not always convenient or efficient to have to update the leaf assignments for objects that you know are static, so for simple static objects like crates and pillars, you may decide that you want to compile them into the tree, which means the HSR routines demonstrated above will have to be discussed and implemented. Fortunately, that is exactly what we are going to discuss in the next section.

16.5 Constructive Solid Geometry

In this section we will be examining the process known as Constructive Solid Geometry (CSG). Over the next several pages we will look at how to apply the BSP tree knowledge we have gained thus far when performing CSG and hidden surface removal (HSR). Our approach to CSG will require that the geometry we send through the compiler is *brush based*. This means that our level data must be made up of a set of enclosed hulls or primitives (e.g. cubes, cylinders, teapots, etc. – basically anything that is completely *solid*). This is the way that many of the most popular level editors work (e.g. WorldCraft™ or GILES™). As we alluded to earlier on in this chapter, hidden surface removal is used to ensure that our geometry is legal and is actually a logical extension of CSG. In fact, we basically get this support for free when we implement CSG functionality using our BSP Compiler.

There are many reasons why we would want to use CSG operations. One of the main advantages for anyone writing a BSP Compiler is the fact that we can perform Hidden Surface Removal (HSR) using the **union** operation. If you have ever tried to create a reasonably complex level using a package such as 3DStudio MAX™ and tried to compile it into a BSP tree, you will likely have learned that illegal geometry is quite commonplace. HSR takes care of most types of illegal geometry which crop up when designing a level. It simply clips away any parts of intersecting brushes which could cause the compiler to fail. The other advantage we achieve by adding support for CSG into our compiler or world editor is the ability to carve one brush from another using the **difference** operation. This allows us to easily carve doors or windows from a wall, or perhaps create other hollow objects (e.g. coffee cups, etc.) that retain their solid geometric properties.

We will discuss HSR in more detail a little later, but in order to do so we must first understand the basic principles involved in each of the 3 primary CSG operations.

16.5.1 CSG Operation Primer

Constructive Solid Geometry (often referred to as a Boolean Operation) is the process of building solid objects from other solids. The three CSG operators most often encountered are known as the Union, Intersection, and Difference operators. Each of these operators acts upon two objects and produce a single object result. By combining multiple levels of CSG operators, complex objects can be produced from any number of simple primitives. Due to the fact that we are basing our shape's 'solid' and 'empty' space on the information constructed by the BSP tree, we need not even ensure that these objects are convex to begin with.

The following set of diagrams illustrates the three CSG operations mentioned. As mentioned, although we are using simple primitives here the same principles apply with any shape we can create.

CSG Operations

We start off with our two brushes: a box and a cylinder. We can combine these two brushes in a number of ways in order to obtain various different results. You may have seen these operations elsewhere defined using their Boolean Operator names, so for future reference we have denoted these alternate names beside each operation heading.

Union (XOR)

This is the **union** of our two brushes. While it may seem as though they are simply placed on top of one another, in fact the union operation removes all polygons that fall inside the solid space of either brush. If we were to move the camera inside either of the brushes, we would see that any polygons (or fragments of polygons), which ended up in solid space are discarded. This operation represents the core of our HSR (Hidden Surface Removal) system. The **XOR** is basically the same as the Exclusive-OR operator seen in many programming languages. It simply keeps any area which does not share the same space (i.e. the parts of the box which lay outside the cylinder and the parts of the cylinder which lay outside the box).

Difference (NOT)

The **difference** operation is probably the most widely used of the three. Its purpose is to carve one brush from another. The difference (or carve) operation will result in different sets of polygons depending which brush you perform the operation with. For example, in this diagram we use the cylinder as the *cutting brush*, which basically subtracts the cylinder from the box. Only the parts of the box **NOT** inside the cylinder remain untouched. However if we turned the box into a cutting brush, it would leave behind the top half of the cylinder (since the bottom half lies inside the box, and is carved away).

Intersection (AND)

Here we see the **intersection** of our two brushes. The resulting geometry of the intersection operation describes only the area of the two brushes which were intersecting (i.e. only the parts of the box **AND** the cylinder which were intersecting remain untouched). This operation is not as widely used as the union or the difference operations, but is an alternative method of creating certain geometry if the other operations are not suitable. Some editors choose not to implement this operation at all.

16.5.2 CSG Principles

With the aid of the solid BSP leaf tree, CSG is not nearly as complicated as it may first appear. In this section we will discuss the theory behind the various CSG operators outlined above before we move on to examine how we might implement a complete CSG processor.

Figure 16.109

The first thing to realise when adding CSG to our applications, is that these methods will only work when the geometry is represented as *Brushes* (or *Solids*). What this basically means is that our level must initially be made up of various primitives such as cubes, cylinders, wedges, spikes, cones, spheres and so on. This is not as limiting as it first sounds. When we refer to brushes or solids we essentially mean anything which can be defined as having a solid space. That is, it is completely enclosed such that solid space cannot leak out into empty space. Thus any primitive that fits these criteria (such as the object shown in figure 16.109) can be a considered as valid input for each operation. The reason for this requirement will become clear when we begin to discuss how CSG actually works. It should be instantly apparent why the BSP tree is an ideal candidate for this job. Due to the fact that the very foundation of constructive solid geometry is based on the determination of solid space within any particular object, the solid / empty information provided by our BSP tree implementation will be of immense benefit to us.

As mentioned, the key to successfully implementing CSG, whether we have the aid of a BSP tree or not, is being able to determine which areas of our brush are solid and which are not. There are many problems associated with making this determination when we are not using a BSP tree, but fortunately for us, this will not be a concern.

In a short while, we will be describing how the various CSG operations actually work. But before we do this, it is vital that you understand exactly what we are describing when we talk about difference (NOT), union (XOR) and intersection (AND). So what we will do is have a brief refresher on bit manipulation which shows how the various operators (used in most programming languages) work on simple numbers. This should help us to better visualize the definitions we will be using throughout the remainder of this chapter.

Although we will be discussing the effect that these various operations have on bits and not on actual geometry at this point, it is helpful to imagine that each bit that is set to 1 is an area of space within the solid interior of the brush it is describing. The tables that follow will show how the areas of both of our brushes after the operation has been performed contributed to the construction of the final combined brush.

> **Note:** If you have any trouble understanding the bit operations, it may be helpful to start up the calculator that comes with Windows (or any scientific calculator which can handle binary numbers) and follow them through for yourself. Make sure that the calculator is running in scientific mode (View/Scientific for the Windows calculator).

The XOR (^) Operation (Union Equivalent)

Figure 16.110

The XOR operation can be a little tricky to understand at first. XOR compares each bit of the first set against the corresponding bit of the second set. Any bits that are equal in both sets (i.e. both equal to 0 or both equal to 1) will result in the corresponding bit in the output being set to 0. However, if either of the corresponding bits equals 1 while the other is 0, then the output bit will be set to 1.

The following bit tables should make this clear. We will be defining the bits as if they were areas of space occupied by a cylinder and a cube. What remains after the XOR operation will be the areas of space taken from the original brushes and used in the resulting brush (i.e. if a bit is set to 0 then the parts of either of the brushes which occupied this space will be discarded and not used in the resulting brush)

Cylinder Brush

1	**1**	**1**	**1**	**0**	**0**	**1**	**0**

XOR

Cube Brush

0	**0**	**0**	**1**	**1**	**1**	**1**	**1**

=

Resulting Brush

1	**1**	**1**	**0**	**1**	**1**	**0**	**1**

Keeping in mind that a 1 in the table above corresponds with an area of space occupied by the solid space in the brush it was describing, you can see that the areas of the cylinder which were sharing the same space as that of the cube will be removed in the resulting brush. It is important to take into account the fact that the result in these examples will describe how the two brushes are to be merged. So if the result contained a 0, then neither the section from the cube brush or the cylinder brush which occupied this particular space will be used in the final brush. This means that any part of the cube inside the cylinder and any parts of the cylinder inside the cube will not be used in the final brush (they are discarded).

Although it may not be clear at the moment, it may be worthwhile to know that this operator is the basis of the hidden surface removal techniques we will be discussing shortly.

The AND (&) Operation (Intersection Equivalent)

Figure 16.111

The AND operation is the simplest of the three. Put simply, if *both* of the corresponding bits in *both* of the sets equal 1, then the corresponding bit in the result will also equal 1. In any other case a 0 is the result. In figure 16.111 we see that only the parts of the two brushes which occupied the same space have remained.

As before, what remains after the AND op will be the parts of the original brushes used in the resulting brush.

Cylinder Brush

1	1	1	1	0	0	1	0

AND

Cube Brush

0	0	0	1	1	1	1	1

=

Resulting Brush

0	0	0	1	0	0	1	0

We can see that in the AND operation, the majority of the space described by each source brush has been discarded. Only the areas which overlapped and shared the same space now remain.

The NOT (&~) Operation (Difference Equivalent)

Figure 16.112

The NOT operator is not quite the same as the AND, OR, or XOR operators. When we use a logical NOT in a programming language it usually returns 0 if the value passed in is non-zero, and 1 if the value passed in is zero. The same logic applies to a bit-wise NOT. All of the bit values are switched from 1 to 0 and vice versa. Recall from our earlier definition of the difference operator that the resulting output of this operation depends on which brush was used as the carving brush. In this case, the space and polygons described by this brush need to be inverted such that we retain the portions of the carving brush that falls into the solid space of the opposing brush, and likewise retain the parts of the second brush that fall into the *empty* space of the carving brush. Because of this need to reverse the situation found in the intersection operation, this procedure basically performs a NOT operation on the carving brush first (but not on the brush we are carving from) and then an AND op on these two sets of bits. This means that we are essentially describing a NOT-AND operation (or NAND).

Cylinder Brush (Carving Brush)

1	1	1	1	0	0	1	0

NOT

Cylinder Brush after NOT

0	0	0	0	1	1	0	1

AND

Cube Brush (Carved Brush)

0	0	0	1	1	1	1	1

=

Resulting Brush

0	0	0	0	1	1	0	1

Here we see that the resulting brush contains only those parts of the original cube which were not occupied by the cylinder. If we wanted to carve the cube from the cylinder instead, we would apply the NOT to the cube rather than the cylinder.

Hopefully you now better understand what each of the CSG operators do. One thing that you will find about constructive solid geometry as we move forward is that that understanding what each operation does is actually harder than implementing the operators themselves. Since we will of course be working with real geometry, the 'area of space = a bit' analogy we used in the previous examples does not ultimately describe the techniques we are going to implement even though they clearly demonstrate the results we should expect.

In the next section we will begin to look specifically at how these CSG operations will function in real world situations, using physical geometry as input.

> **Note:** If you would like to see a practical demonstration of each of these operations in a real world application it is recommended that you obtain and install a copy of the GILES™ world editor from your class CD or download area. In this application you will be able to try out each of the CSG operators described above, using any manner of varying brush shapes that can be created. If you need help with using the GILES™ application, a full manual is included in the installation package, available from both the start and help menus.

16.5.3 Performing CSG with Geometry

At this point we are already well versed in BSP theory and we should know exactly how they subdivide the space within an object into solid and empty areas. Although when working with CSG we will usually be compiling only small subsets of our level at any given point, remember that the compilation principles used are exactly the same as those employed when compiling a multi-thousand polygon scene.

Before we begin to process any geometry for the purposes of our chosen operation, we first need to be able to determine which areas of space within each of the brushes in our scene (cube, cylinder, etc.) are solid and which are empty. In order to achieve this, we will first need to compile a unique BSP tree for each brush that we would like to take into consideration during the operation. Although it is not strictly necessary for us to compile each of these trees in advance – and in fact it is probably a good idea to only compile each tree as and when we need it – we must ensure that this information is available to us at some point during the procedure.

With this solid and empty spatial information to hand, we are able to implement each of our CSG operators using a simple series of clipping and removal operations.

16.5.4 Implementing the CSG Operators

As the basic principles behind the implementation of each of the CSG operators are the same, we will begin by examining the **union** (**XOR**) operation which we will use as the foundation for understanding how the other two remaining operators function.

The Union Operation

As we know, the union operator is intended for to be utilized in cases where we would like to merge two brushes together. In order to achieve this we must first clip each of the brushes' polygons to the others compiled BSP tree, and remove any fragments of those polygons that may fall into the solid areas of either brush's tree.

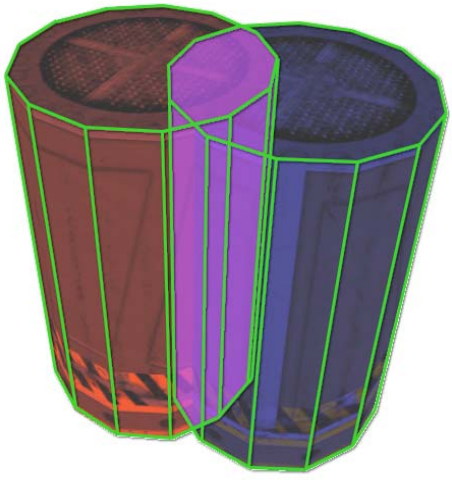


Figure 16.113

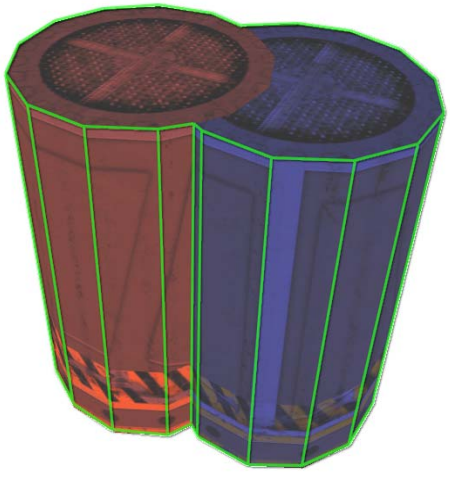


Figure 16.114

Figure 16.113 shows two intersecting cylinders. We can clearly see that we have begun this process using two separate convex hulls (the cylinders themselves) both of which have their own solid space (the red and blue areas). We can also see that where the cylinders overlap, we have an area of space that is shared by both hulls (the purple area). Figure 16.114 shows the outcome of the union operation, where the sections of each hull that lay inside the others' solid space has been removed. The result is a single solid shape which has had any and all interior polygons split and removed. One thing that you may have noticed about the resulting brush in the above figure is that the top faces of the cylinder seem to have remained untouched. Obviously if we were to remove both pieces of the top polygon that fall inside each other's solid space we would be left with a hole in our new solid. The specifics about how to handle this case are covered shortly, but for now just concentrate on how the boundaries of the objects (shown as green wire frame) have been merged together to form one continuous mesh.

Given the basic concept of this operation, let us take this background knowledge and apply it to two simple cubes using the BSP Tree. The first step is of course to build up a BSP Tree for each of the cubes (we will ignore the top and bottom polygons of this cube in order to keep the examples clear, but the same principles apply). Figure 16.115 depicts a top down view of a cube brush and figure 16.116 shows the resulting compiled solid BSP leaf tree.

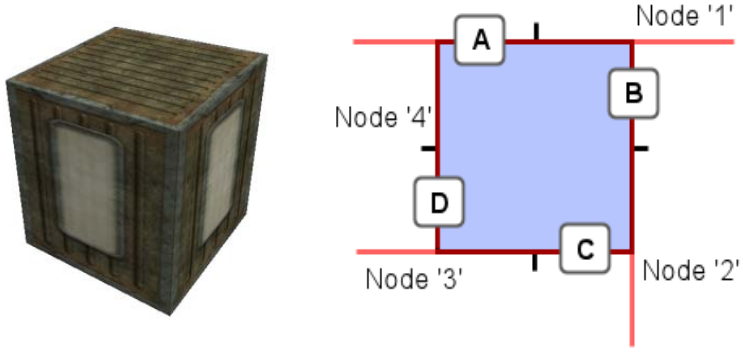


Figure 16.115

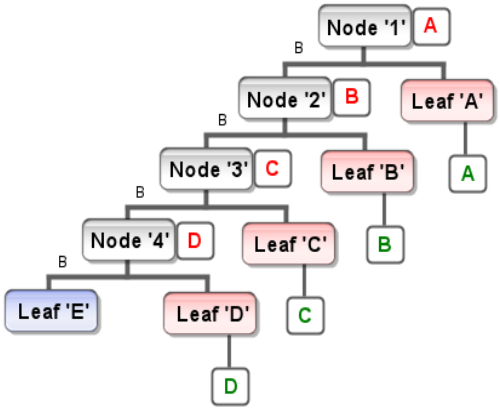


Figure 16.116

The process used to construct this tree should already be familiar at this point so we will not go into great detail about how we ended up with the hierarchy shown. This tree is quite unbalanced and certainly very simple, but it will benefit us to start small. As in each of the prior examples, the letters in figure 16.115 denote the original polygons that bound the solid space inside the cube. As illustrated in figure 16.116, we can see that we end up with just one solid leaf at the bottom of the tree with each of the remaining empty leaves attached to the front of each node. With this information available, we can start to actually perform the union operation. We will use a second identical cube to perform the op with, since the BSP tree constructed for each cube will be the same.

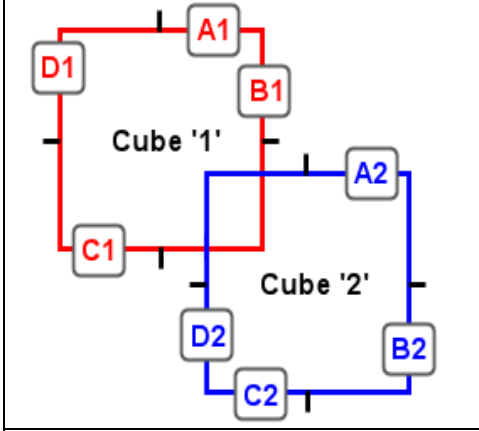


Figure 16.117

Figure 16.117 shows our two identical cubes and how they intersect. We will need to perform the same operation on both cubes (i.e. removing the areas of the first from the solid area of the second and vice versa), but we will begin by clipping cube 2's polygons against those of cube 1. To perform the union operation, we must pass every polygon contained within the first cube through the tree of the second cube, splitting the polygons against node planes as we recurse. Whenever a polygon fragment ends up in a leaf which describes solid space, it will be discarded. Once these steps have been completed for the first cube, we reverse the process and perform exactly the same operation with the second cubes polygons. Let us walk through the example step by step. We will refer to the cube 1's polygons as A1, B1, C1 and D1, and likewise the cube 2's polygons as A2, B2, C2 and D2. In the first iteration of this process we do not actually use the polygons of cube 1 at any point. This is the key to this entire process. At this point we are simply classifying and splitting the polygons of cube 2, against the nodes of the first cube's tree as we traverse.

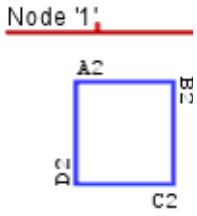


Figure 16.118

First we classify all of the polygons of cube 2 against the root node (node '1') of the tree we are clipping against. As we loop through each polygon, we can see that as we test them individually, each one is found to be completely behind this node. Just as with the actual BSP compilation process, each of these polygons should therefore be added to a *virtual* back list for this node (i.e. it is not physically attached to the node – we keep two separate lists called 'Front' and 'Back' and store each polygon in this list depending on where it lies in relation to the nodes plane. We will see why we do this in a moment). Once all of the polygons of the cube have been classified against this plane and added to the relevant list, we then step to the next node in the tree, passing the appropriate list of polygons. Since there are no polygons in the front list at this time, there is no need to step into the node/leaf indicated by node 1's front pointer. We do however have polygons stored in the back list so we pass this list down the tree to node 1's back node/leaf. (We will explain what happens if this is a leaf shortly).

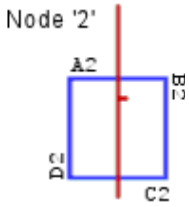


Figure 16.119

We have now passed all of our polygons to the node described by Node 1's back pointer, which happens to be node 2 in this case. As before, we start classifying polygons against the current node's plane. Starting with polygon A2, we find that this polygon is spanning the node plane and should be split. This gives us two separate polygon fragments -- one behind the plane (A2a) and one in front of the plane (A2b). We then add each respective split fragment to the relevant front or back list and discard the original polygon. Now we move on to polygon B2 which is completely in front of this node. Thus, it is added to the front list. Polygon C2, like polygon A2, is also spanning the plane and again must be split with the resulting fragments being added to the relevant list. Again, the original polygon is deleted. Finally we classify poly D2 and find that it is totally behind the plane. It is also therefore added to the back list.

At this node, we actually have polygons in both our front and back lists. We will deal with the front list first. Our front list contains part of the original A2 polygon (labelled A2b), part of the original C2 polygon (named C2b), and the complete polygon B2. As before, we need to pass this front list into

whatever node 2's front pointer contains, which in this case is an empty leaf. Because the union operation only discards polygons which end up in solid space, all of the polygons in this front list have survived the operation and as such, can be added to the final resulting brush at this point.

Our back list contains the fragments A2a, C2a and the in-tact polygon D2. This list is then passed down the back of node 2 and into node 3. We begin the process yet again. In figure 16.120, our split polygon fragments are shown in green.

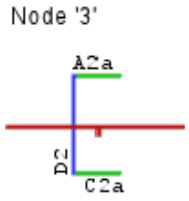


Figure 16.120

> **Note:** Remember that we are only testing against the polygons which actually survived to this point and were passed down to this node.

We start at the beginning of our list and find that the fragment A2a is completely behind the plane and is therefore added to the back list. Then C2a is added to the front list. D2 is spanning node 3's plane so it is split into two fragments (D2a and D2b) where D2a is added to the front list and D2b to the back.

As before, we have some polygons in our front list, so we pass this to whatever is in the front pointer of node 3. This happens to be an empty leaf, so whatever is in the front list has survived and can be added to the final resulting brush as before. Our back list also contains some surviving polygons, so it will be passed down the back of node 3 which directs us to node 4.

In node 4 we now have only two poly fragments that remain (A2a and D2b). We classify each of these and find they are both behind the plane and added to the back list. As in the example of node 1, we find there are no polygons in our front list, so there is no need to go recurse into the front. We do however have these two remaining polygon fragments in our back list, so we pass it down the back of node 4. However, remembering that we can distinguish solid leaves as being any attached to the back of a node this time around we find that we fall into a *solid* leaf. Since the union operation discards any polygons which make it into solid space, these polygons will *not* be added to the final brush.

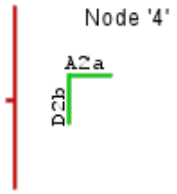


Figure 16.121

Figure 16.122 depicts the polygons that actually made it into the final brush. You will notice that the original polygon C2 has been split, and was added as two separate fragments (even though neither was deleted). Unfortunately there is no way to avoid this during the traversal process, but there are many techniques we can employ to reduce the number of splits in our final brush after processing has been completed. These will be discussed as we cover the implementation of our final CSG processor.

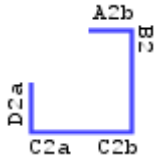


Figure 16.122

As mentioned, we have to perform this process for both brushes. So the next time around we will have to pass all of the ***second*** cube's polygons through the ***first*** cube's mini-BSP Tree in the same way as we have done with the first. But how can we do this now that we have clipped and removed most of the polygons? Actually this will not be a problem. Remember that we have already compiled the Mini-BSP trees for each brush, before the CSG process begins. This means that even though the polygons of cube 1's tree have been altered, the tree of this cube is still completely intact in its previous state. For the purposes of CSG, the polygon information that is stored in the tree is essentially irrelevant, all we need to pay attention to are the node planes, and the leaves themselves. While we will not step through the

process again for the second cube (as you should now be able to do it for yourself) the following figure illustrates the process.

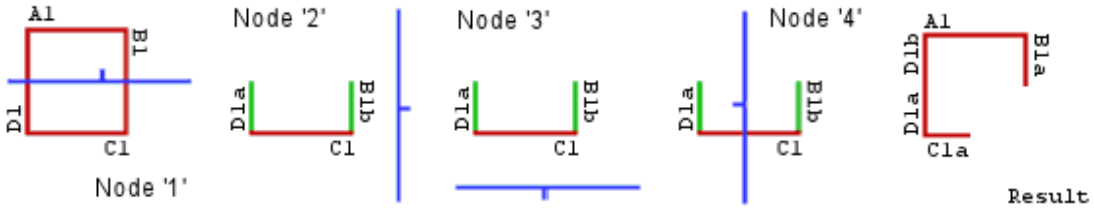


Figure 16.123

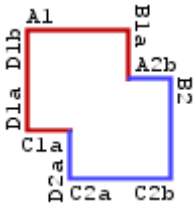


Figure 16.124

We see now that the final results of both clip operations yielded the hull shown above. We can also see exactly where each polygon came from. Although this jumble of nodes, planes and polygons may be a little confusing at first, by referring back to Figure 16.124 and following the process through step-by-step for each brush, you should start to understand exactly what is going on at each stage through the tree traversal.

We are fortunate in that we are able to reuse much of the functionality of our BSP tree code when performing CSG – for instance, our polygon classification and splitting routines. This means that with a little bit of planning, we can actually add our complete BSP tree clipping routine in just one function thanks to recursion. While some people operate under the impression that recursion is a large performance penalty, the reality is that most C++ compilers can generate much faster code than the average programmer trying to implement a manual stack. We do not really have to face the typical problems like stack overflows, because the depth of the Mini-BSP Trees will be relatively small.

One thing that we did not encounter in the previous scenario is what happens when we find a coplanar test case. As it turns out, the on-plane case does present us with a small problem when performing a CSG operation. Let us take a look at this problem now.

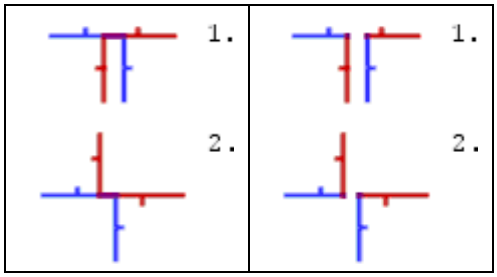


Figure 16.125 **Figure 16.126**

Figures 16.125 and 16.126 demonstrate the ‘before’ and ‘after’ of two separate cases which include polygons that are coplanar with the nodes of the tree we are testing against. In the top case (case 1), there are two polygons which share the same plane and in this case both face in the same direction. In the bottom case (case 2), the polygons are on the same plane but are this time facing in opposite directions. Let us examine case 1 first. By referring to figure 16.126, we can see the outcome if we were to automatically pass these on plane cases down the back of the node with which it is sharing a plane.

The reason this would happen is that, as we know, eventually these polygons would make their way into a solid leaf and would never be added to the final resulting brush. If you look at the colours of the polygons in case 1 in Figure 16.126, you will realise that once the CSG processor has run its course, the two lower polygons will eventually have been removed because they reside in solid space, but the gap still remains. So we end up with a hole in our hull where polygons were previously overlapping. This is clearly not a good situation, and will more than likely result in our final BSP compile failing due to illegal geometry (i.e. space leakage).

However, in case 2, we find that passing the coplanar fragments down the back of the tree is correct. If you refer back to the union operation, you will realise that the two areas of solid space have been merged and that this is still a totally valid hull (all hidden surfaces have been removed). This is a crucial part of the HSR algorithm which will take place before final BSP compilation, so we are actually fortunate in this way to be able to simply test the normal of the poly against the plane of the node. If it is facing in the opposite direction to the node, then we can simply add this poly straight into our back list as we would do for any other poly which is behind the plane. This is virtually identical to the method we used to handle the coplanar cases during the construction of both the node and leaf BSP tree earlier in this chapter. Recall that we test to see if the polygon normal is facing in the opposite direction to the node or not. This also works for the other operators (such as difference and intersection) but we will discuss those a little later in the lesson.

So the only case we have to deal with when handling the coplanar cases is that in which both the polygon normal and the node's plane are facing in the same direction.

The solution to this problem of the hole in the resulting hull is relatively simple. We already know that we perform the clipping first on one brush, and then on the other. All we have to do is to allow the clipping routine to pass these cases (where both are facing in the same direction) down the back list for one brush, but when we reverse the operation and clip the other brush, make sure that it is passed down the front list instead. Again, refer back to case 1 in Figure 16.126 where we can see the result if we were to send both fragments, from both brushes, down the back of the clipping node. In that case, both fragments are removed. But if only one of the fragments is removed, by passing only one of them down the back and the other down the front, we can see that the one sent down the front will be added to the resulting brush and the gap will be closed. Earlier in this section we demonstrated the result of the union of two cylinders in figure 16.114, in which only one of the top faces was clipped, and the other remained in-tact to fill the hole. This is an ideal demonstration of this process in action.

Fortunately, as mentioned, this same process applies to all of the CSG operators that we cover in this chapter. This is nice because it means that for these other operators we can follow exactly the same steps we have outlined here. For the most part, we need only to decide whether we want to discard the polygons which end up in solid space or those that end up in empty space.

For more information on how best to handle the on-plane case during implementation, it would be a good idea to refer to the workbook for this lesson to see the approach that we have used.

The Intersection Operation

Due to the fact that the implementation process is the same between each type of operator, we can simply now discuss how we might adapt the techniques we learned in the union operator for use with the **intersection (AND)** operation.

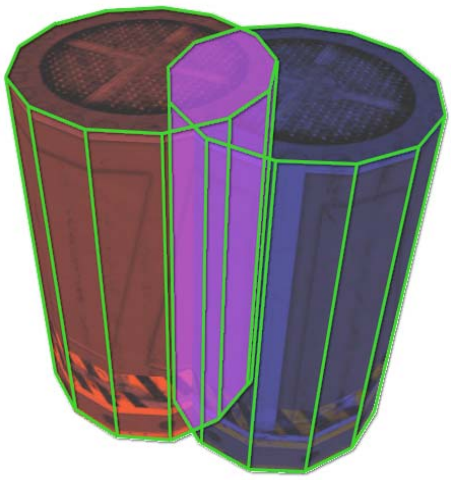


Figure 16.127

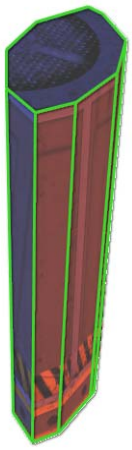


Figure 16.128

The intersection operator works in much the same way as the union operation did. In Figure 16.128 we can see that in this situation, everything that ended up in solid space has survived as opposed to being removed. There is not any significant difference in the way this works: all the on-plane cases remain the same, and everything is still split and sent down the relevant side of the clipping node as before. The only difference is that in order to produce the result we are looking for we need to discard any polygons which end up in empty space rather than in solid space as we did previously. Thus only polygons that end up in solid space are added to the resulting brush. Just as before, we perform the operation with one brush's polygons against the other brush's BSP Tree, and then reverse the process using the other brush's polygons against the first brush's BSP Tree.

For completeness the following diagram shows the result of this operation using our cube example from the last section. The step-by-step diagrams would be identical to those we saw in the union case and have therefore not been included here.

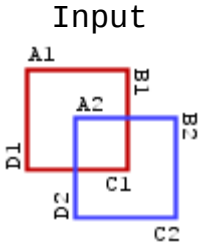


Figure 16.129

Figure 16.130

The Difference Operation

The final operator we will discuss is the **difference (NOT)** operator. It works in a slightly different manner than the other two operators, due to the fact that it is dependant on which brush we want to use to perform the operation with.

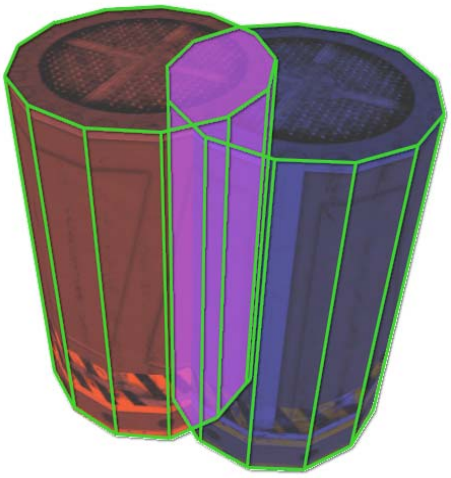


Figure 16.131

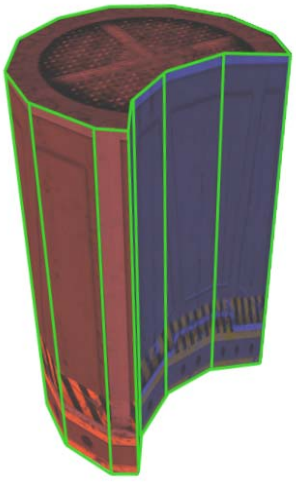


Figure 16.132

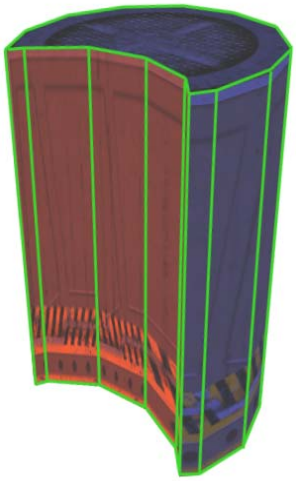


Figure 16.133

There are actually a number of aliases by which this operation is known, and you may encounter them in other contexts. Very often you will hear the difference operation referred to as a NOT, NAND, carve or subtract operation. We can see that Figure 16.132 shows the result after the blue hull from figure 16.131 has been subtracted or carved from the red hull. By the same token, Figure 16.133 shows the result after the red hull from figure 16.131 has been subtracted or carved from the blue hull.

It is clear then that the outcome of this operation does indeed depend on which brush we carve from the other. If you look back to our earlier definition of ***NOT***, we can also see that in our literal definition of the NOT / NAND operation, the outcome depended on which set of bits we swapped using the bit-wise NOT operation.

The interesting part is how we 'NOT' the brush we want to carve with, in the same way we did with the bit manipulation we performed earlier. Just as the NOT inverted the bits, we will invert all of the polygons and their normals before compiling the mini-BSP tree for this brush. In doing so, solid space becomes empty space and empty space becomes solid space. We will then perform the AND operation as shown in the intersection operation previously discussed (i.e. keep anything which ends up in solid space in either of the brushes).

Although this operation is very similar to the intersection op shown previously, we will step through the entire process as we did with the union since it is very important that you fully understand the process. Keep in mind that the BSP tree has been built in a different fashion this time around (i.e. it is inverted). To help illustrate this, the new tree which was built from the *inverted* brush polygons is shown next.

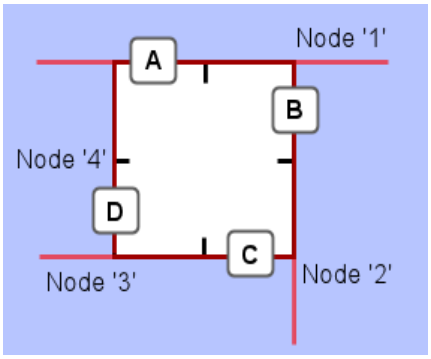


Figure 16.134

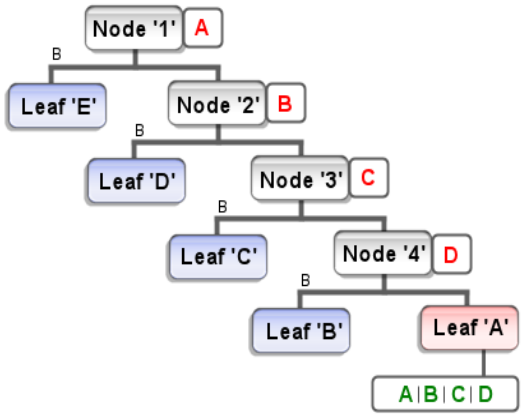


Figure 16.135

Figure 16.135 shows our inverted tree structure in which you can see that each of the brush polygons have been. Just remember that we cannot generally get away with inverting only the normal; we also have to invert the winding order of the polygons themselves if we want these polygons to render correctly when attached to the final resulting brush. We will explain how to reverse the winding order in the accompanying workbook for this chapter.

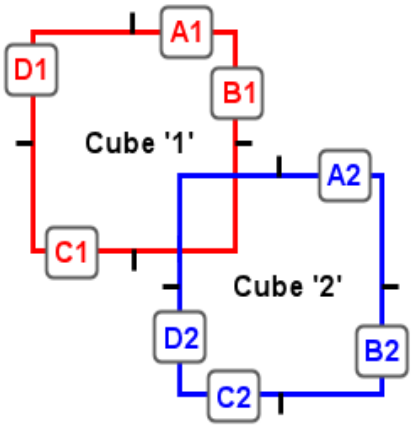


Figure 16.136

As before, we are going to use the same two cube example. In this example scenario we are going to be carving cube 1, from cube 2. Remember that cube 1 has been inverted and its mini-BSP tree has been built from the polygons in their inverted state. This means that the node planes will point in the opposite directions as shown in the figures above. Also remember that we have done nothing to the second cube just yet. We do not need to invert the polygons of any brush we are carving *from*, only the brush we are carving *with*. You can refer back to figures 16.135 and 16.136 for the definition of the second cube's tree if needed. To perform the difference operation properly, we need to do exactly the same thing as we would when performing an intersection (AND) operation assuming we have built this inverted tree. This means we will delete everything that ends up in *empty* space. This will be achieved in the same way in which we performed the clipping in our step by step union example with the exception of the differing polygon discard rules. For future reference, it does not particularly matter which brush we start with, but this example will focus only on clipping the second cube, to the tree of the inverted cube '1'. Let us now examine the rest of the process.

Looping through in alphabetical order, we find that all of our polygons which are being tested against node ‘1’ are in front of the plane, and as a result they are all added to the front list. There is nothing currently in the back list so we do not need to pass anything down the back of this node. Thus, we continue down the front, passing our newly built front list.

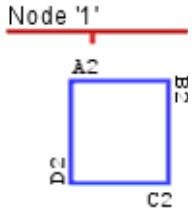


Figure 16.137

The front leads to node 2 (see figure 16.135) and we again classify our polygons against the node. We find that polygons A2 and C2 need to be split and their fragments are added to the relevant front/back lists. (Remember that we delete the original poly at this point). B2 is placed in the back list and D2 also goes in the front. Although our function would usually traverse down the front of the node before dealing with the back list, we can do it now to save confusion due to the recursive nature. The back list (containing A2b, B2 and C2b) is passed down the back of the current node and we find ourselves in a solid leaf. Since the difference operation keeps any polygons or fragments of polygons which end up in solid space, these polygons can be added to our final brush. The front list is passed down the front of the node which leads us to node 3.

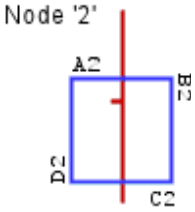


Figure 16.138

The diagram above shows the polygons that were passed to node 3. Again we classify all of the polygons and find that C2a is added to the back list and D2 is split into two fragments on either side of the plane. D2a is therefore added to the back list and D2b is added to the front list along with A2a which is also in front of the plane. As before, the back list is sent down the back, and again we find a solid leaf and add these surviving polygons from that back list to our final brush. The front list goes down the front of the node, which takes us finally to node 4.

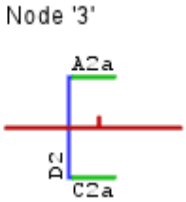


Figure 16.139

Classifying the remaining polygons, we find that no polygons are added to our back list and that both are added to the front. Our recursive function then attempts to pass this list down the front of the node, but we find that there is an empty leaf here. Since this operation discards any polygons which end up in **empty** space, these final two poly fragments are deleted.

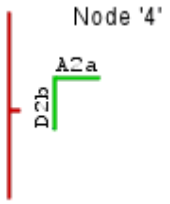


Figure 16.140

The final resulting brush is shown in the figure 16.141. We can see that, as we are clipping the second cube, this figure depicts exactly the same resulting fragments as those from cube 2 that survived during our union operation. This clearly demonstrates that swapping the rule about whether we delete what is in empty space or in solid space really does work as expected.

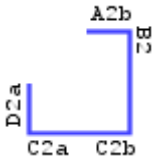


Figure 16.141

While this part of the difference operation was fairly obvious, the next bit is where the main differences lies. We can see the steps involved in clipping the first cube (the inverted brush) against cube 2 in figure 16.142. You may notice that the diagrams are identical to those we saw with the union operation. You should be able to work your way through the process with the aid of the following illustrations without much trouble. Be sure that you understand exactly why we get the results that we do.

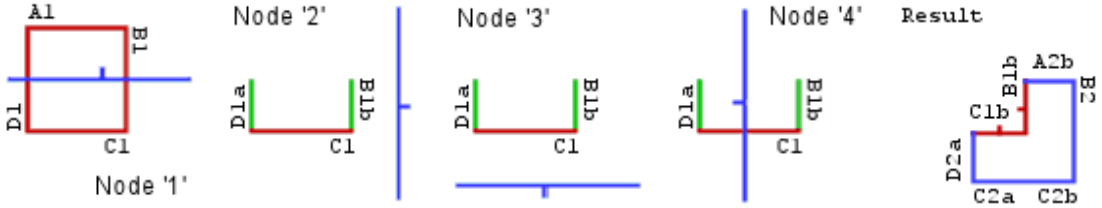


Figure 16.142

If you carried out this process on paper, you will hopefully have arrived at the result shown above. The red polygons will be all that remain of cube 1. Due to the fact that we inverted cube 1's polygons before we built the mini-BSP tree for that brush, you can see that they are now automatically pointing in the correct direction. They also close up the gap which would have been left cube 1's exterior hull.

> **Note:** Bugs in CSG processors are very difficult to track down. If there is a problem during implementation, it is very rare that the problems you see in the resulting geometry will have any bearing on the problem inherent in the code. For this reason it is important that you understand the operations completely.

Additional Notes about the Difference Operator

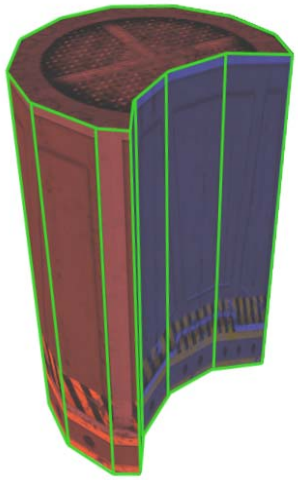


Figure 16.143

In an ideal world, the difference operator would fit neatly into the concepts we discussed in our 'area of space = a bit' examples. However, in some cases we may not be able to invert the brush data before compiling the mini-BSP tree. For future reference, we will look at one possible solution for this case. When we are performing the difference operation, we obviously still have to clip one brush to the other and vice versa in the same way we always have. For this explanation we will use Figure 16.143 as the target result we would like to obtain.

In this case we will assume that we are not able to invert the brush data before rendering, or for whatever reason, cannot invert the BSP Tree data after it is built. The solution we can employ is that when we clip the red hull's polygons using the blue hull's BSP Tree, we **delete** any polygons which end up in **solid space**. However, when we then clip the blue hull's polygons against the red hull's BSP Tree, we **retain** any polygons which end up in the red hull's **solid space**, and **delete** anything which ended up in **empty space**. There is one final step: the polygons that remained of the blue hull (which we can see in Figure 16.143) must be inverted before being added to the final brush so that they point outwards and seal up the hull. By switching the rule between the two traversal / clipping procedures in this way, we are essentially getting

the same result as inverting the tree itself. In theory however, this is not as clean as simply performing a NOT on the entire carving brush. Although either solution is a valid one, it is really more a matter of preference than necessity in many cases.

16.5.5 Conclusion

We covered a good amount of new information in this section that should be fully understood before continuing on. It would be a good idea to try some examples out on your own: perhaps an 8-sided cylinder against a cube. You can also experiment with these operations in the GILES™ world editing program. Try some of the operations on cubes, cylinders, and spheres and see what sorts of results are generated by the CSG processor under different circumstances. It will help to have a visual understanding of what it is we are aiming for as GILES™ itself uses all of the operations we have discussed here in this chapter.