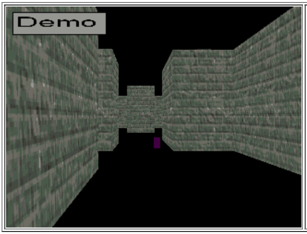
Binary Space Partitioning Tutorial

for Mr-GameMaker.com



Creating a Solid Node Based BSP Tree Compiler & Renderer

written by Gary Simmons

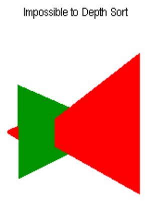
Introduction

Although Binary Space Partitioning has been around for many many years it really became a buzz word when it was discovered that John Carmack used BSP trees in Doom and Quake. Although the game Doom is now of some age virtually all the latest first person shooters still use BSP engines which really goes to show how this technique has stood the test of time. The aim of this tutorial is to explain the basics of how a BSP works as well as to write the code to a 'Solid Node Based BSP compiler' and Renderer.I hope to explain this in such a way that even somebody fairly new to 3D graphics will understand it as I feel most other resources on the net leave a lot of unanswered questions for the less experienced programmer.

In order to understand BSPs though it is an absolute necessity that you be well versed in the Dot Product and have a complete understanding of it.If you are yet to have any experience with the Dot Product it is highly recommended that you read our Dot Product tutorial before proceeding with this tutorial.It may also be worth you reading our Cross Product tutorial so you fully understand the generation of Polygon Normals.

What is a BSP tree ?

During the development of a 3D engine there comes a time when you have to decide how you are going to draw all the polygons in a frame from the current camera position in the right order. Lets assume that your level consists of 10000 polygons (an average quake level) , how are you going to draw them all in the right order each frame ? Ok you say, forget the order I will just use a Z Buffer. This would certainly work if your level has no translucent objects but what if it has.The Translucent objects must be rendered after the objects that are underneath it otherwise it will not blend correctly. Also performing a for/next loop to sort all your levels polygons every frame would result in a performance level where you could probably bake a cake in between frame updates.Add to that collision detection of all the polygons in your level (having to check if you have hit each one) and then you could probably ice the cake as well.Not only that, but sorting your polygons based on their distance from the camera does not solve the following case.

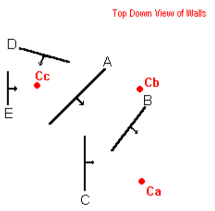


The picture above causes a real problem when trying to draw your polygons in a back to front order (using the painters algorithm) because the RED polygon is both further and closer to the camera than the green one.How do you sort this? Well a Z-Buffer sorts this problem of course but can be extremely slow if there is no Hardware Acceleration and this still does not solve the problem of having to draw your translucent objects in the right order but a BSP can sort this problem without the need for a Z Buffer and also , storing a level in a BSP Tree can provide many benefits in other areas (collision detection,polygon culling).

Creating a BSP tree of your level can solve all these problems by precalculating the order of your polygons at development time (not runtime) so all the polygon sorting is precalculated .This is the Job of the BSP Compiler that will take an input list of polygons and pre sort them.After they have been sorted they are saved to disk in the form of a BSP tree which can be loaded at runtime and quickly traversed.Not only can a BSP tree be used for drawing all the polygons in your scene in the right order (WITHOUT the need for a Z-Buffer) but it can also be used (in my opinion more importantly) for culling any unseen objects from the fustrum at lightning speed.This is because with a simple test you can check whether a given polygon is behind the fustrum, if it is, then all the polygons behind this polygon are also rejected in one go rejecting most probably most of the polygons in your game level. I dont have to tell you how much faster your engine will run if every frame you can reject a couple of thousand polygons with a few simple tests and stop them having to be transformed and lit.We will also cover how to perform fast collision detection using a BSP tree.For now though we are going to concentrate on the first order of business, how to draw the polygons in our scene in the right order. We are going to write a BSP compiler later so do not panic but first lets have a real good hard look at how a BSP tree works.

How do they Work

Take a look at Figure a) below.This shows a simple game level from the top looking down.The black lines are the WALLS of a simple game level viewed from the top down and the black arrows coming out of them describe the way the wall is facing.



Above Ca,Cb & Cc are examples of where the player in the game (or the camera) could be situated.In order to draw the level in the right order we need to be able to Draw the walls farthest from the camera first.We certainly do not want to draw a near wall and then have our engine render a wall that is supposed to be behind it on top of it.We can see that from the three camera positions we should draw the walls in the following order.

Camera Position	Correct Drawing Order
Ca	D or E in any order, A , C & B is Unclear
Cb	D or E in any order, A , B & C is Unclear
Cc	C & B is Unclear , A , D or E in any order

A BSP Tree stores all the polygons in a level so that they are either behind or infront or neigbouring polygons but can you see why we have a problem here with walls C & B.We can not properly describe it as being either Front or Back because it is in fact both.Part of wall C is infront of wall B and part is Behind.This causes a real problem for rendering because we can not find a correct drawing order that will work from all positions.Because we cant describe wall C as being either behind or infront of wall B the BSP Tree algorithm will not work and may get it wrong in some places.To deal with this our BSP Compiler must detect that a polygon is straddling another and split that polygon into two halves.One half can then be placed infront of the polygon and the other half placed behind.This may sound a little complicated at the moment but do not worry it will get clearer.

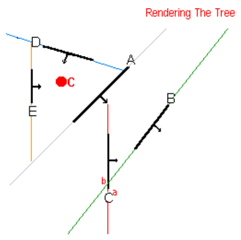
A BSP Tree stores its information by splitting the world into two halves (a bit like drawing a huge never ending line right through the game world).All the polygons are checked to see which side of the split they lie on (front or back) and are assigned to the relative half.If a polygon straddles the splitter then that polygon is split into two bits, the first bit that lies in front of the splitter and second bit that lies behind.These two new polygons are then assigned to the relative lists (front or back) of the splitter and the original polygon (before it was split) can be discarded and is no longer used in the BSP Tree.This is because two new polygons have been created to take its place.These two polygons that represent the first though can accurately be described as being either front or back of the splitter.

So our first split is now complete.We have two lists of polygons one that lies to the front of the splitter and one that lies to the back.What we do now is repeat that process over and over again creating new front and back lists until we have assigned each polygon a unique place in the tree.So for example after we have performed the first split we have a front and back list.We now go down the front list of polygons and split this list into two using a new splitter.After this we have a front and back list for the second spitter so we go down the front and back lists of this splitter and find yet another splitter and split these lists.Each time we are dividing the split list and making them smaller until there are no more polygons left to be assigned.(Dont worry if this is giving you a headache it will start to make sense in a minute).

Ok then, so we now know that the first thing we have to do is split the original list of polygons that describe the game world into two and and assign each polygon to either front or back of the splitter.But where does the Splitter come from.We can actually use the planes of the polygons in the levels as the splitters.

In case you dont know what a plane is , every polygon lies on an infinite plane.Get a piece of A4 paper and draw a small triangle in the middle of it.The triangle is the Polygon but the Paper is the Plane.The plane goes on forever (it has no edges like the paper) though but can you see that if you hold the paper up infront of you and start turning it to different angles the paper is always at the same angle as the triangle.In other words a plane is almost like a polygon but with NO edges around the outside.You can also see that if you are sat in your living room and the paper went on forever it would carve a line in the walls of your house.If you imagine that walls of your house are polygons you can see that after the wall has been split by the paper one half would lie to one side of the paper and the other half would lie the other side (front and back splitting).

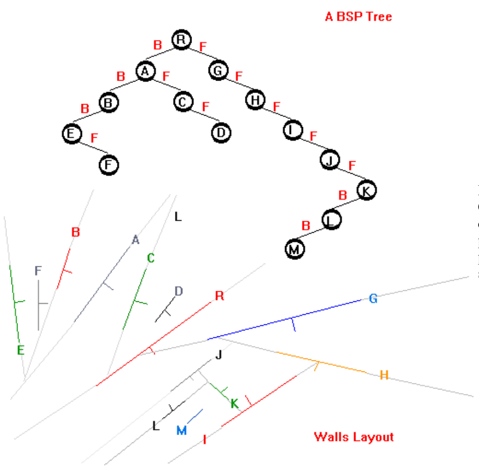
Well we will look at choosing a good splitting polygon later but for now lets just say that is does not matter and we can use any polygon we like.Another important thing is that the actual polygon used as the splitter is not added to any list (front or back).This means each time we split we eliminate a polygon and we keep on splitting until all polygons in the level has been used as a splitter so the front and back lists become empty.At this point our BSP tree is built.Lets now have a look at our game level and see how a BSP Compiler might split it up.Remember the order we are choosing the spitters is purely for the purposes of explanation at the moment.



Front List of split (A)	Back List of Split (A)
B , C	D , E
Front List of split (B)	Back List of Split (B)
Ca	Cb
Front List of split (D)	Back List of Split (D)
E	Nothing

So first we pass our compiler a list of 5 walls.First of all we choose wall A as the splitter.The GRAY line describes the plane polygon A lies on and carves the world up.You can imagine this line (plane) as going on forever (of infinite length) in all directions.So with A used to construct our splitter we loop through the polygons (ignoring A itself) and test all the polygons against the splitter (A) .We keep two lists (Front and Back) so it is easy to see that after this first split the front and back lists look like shown above (Front/Back list of Split A). Next we do exactly the same thing again but this time instead of using all the polygons in the world we use just A's Front list and then A's back List etc.We should see by this that a recursive function is needed to build the tree and it is suprising how small the code to a BSP compiler actually is because of this.We just call our BuildBSP funtion and pass in the original list of polygons and the BuildBSP function seperates the front and back polygons into front and back lists and then the function calls itself passing in first the front and then the back list.The thing to try and picture in your head is the cascade effect of this.The BuildBSP function calls itself with front and back lists which themselves get split into front and back lists and sent to the BuildBSP function which again divides these lists into front and back lists etc until all the polygons have been made into splitters.Lets have a look now at a slightly more complicated example and also show the tree how it would appear in memory.Each circle in the following diagram is called a NODE.A Node is just a structure defined by you that holds the splitter(which is also the polygon) and also pointers to a front and back child Node.We will look at this in much more detail in a moment but for now just make sure you understand how the polygons (which are also are the splitters in our example) are linked together in the BSP Tree.Below is a diagram showing the BSP Tree and the Layout of the walls.The walls are laballed alphabetically in the order they were choosen for being splitters and the letter F and B in the tree diagram represent two pointers Front & Back at each node that points to two other nodes.One to the front of the current node and one to the back.

Note:
The reason I keep reminding you that the polygons stored at each node are also the splitters is that it is possible to use any arbitrary planes as the splitters.For example, you may wish to devise a tree where all the splitters are planes constructed to be aligned to the principle world axis and in some instances this can be beneficial , but this usually causes more polygons to be split which raises the polygon count.Also its much easier to use the polygons as the splitting planes because we already have them.



First we choose polygon R as our splitter.This means that the polygons get divided into two lists (Front & Back).At this point the Front list contains polygons G,H,I,J,K,L,M and the Back list contains polygons A,B,C,D,E,F. So then first we take a look at splitter R's Front list.We then choose another polygon (G in our example) and make Polygon R point to polygon G with its front child pointer .We further split this list into front and back lists.There are no children to the back of polygon G so all the remaining polygons (H,I,J,K,L,M) are put into G's front list.We then choose another splitter (H) and point to this with Polygon G's front child pointer.When the front list is done for each splitter we then do the same with the back list.Study the BSP Tree diagram hard an make sure you can see exactly how the splitters are linked to each other.When you have this in your head carry on reading...

A typical structure for a BSP Node may look something like this.Remember this and get it in your head as you will need it to understand the rest of the tutorial:-

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

You can see that this structure represents what the Nodes in the above diagram would need as base properties.A splitter which is also the polygon to be rendered at this Node and pointers to a Node behind this node and a Node in front of this Node.

By the time our BSP compiler has finished compiling the above level each Node in the tree will contain a Polygon which is the splitter and will have a Front pointer pointing to a Node that is in front of it and a Back pointer pointing to a Node that is behind it.These nodes in turn will each have a polygon stored at the Node that is the splitter and front and back child pointer nodes pointing to a node that is infront and behind of it respectively.Note that not all Nodes will have valid Front & Back child pointers.For example in the above diagram G has no back children so we set G's back pointer to NULL.The same is true for H,I & J.Notice also that K & L have only back children so their FrontChild pointers would be set to NULL.Node M is a leaf node because there are no other polygons either in front or behind that have not been already used by the compiler so both its Front and Back child pointers would be set to NULL.

Ok then, we have had just about enough theory for the time being lets start to look at how we can actually implement this in code.Lets first have a look at the pseudo code to walk a BSP Tree and render it.Its so small you will not believe it.

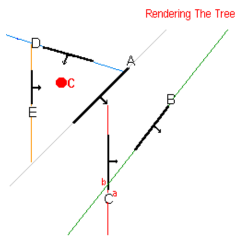
```
void RenderBSP (NODE * CurrentNode)
{
int Result;
Result=ClassifyPoint(CameraPosition,CurrentNode->Polygon);
if (Result==Front)
{
if (CurrentNode->BackChild!=NULL) RenderBSP (CurrentNode->BackChild);
DrawPolygon(CurrentNode->Polygon);
if (CurrentNode->FrontChild!=NULL) RenderBSP (CurrentNode->FrontChild);
}
else
{
if (CurrentNode->FrontChild!=NULL) RenderBSP (CurrentNode->FrontChild);
DrawPolygon(CurrentNode->Polygon);
if (CurrentNode->BackChild!=NULL) RenderBSP (CurrentNode->BackChild);
}
}
```

Thats it!! That little snip of recursive code will call itself repeatedly until the whole BSP Tree (level of our game) is drawn.We first of all call this function and pass in the Root Node that contains the first polygon that was used as a splitter in our compiler.The ClassifyPoint function above is a function that we will write later that tells us whether the camera is currently on the front side or the back side of a polygon.You can see that we test the polygon stored at the Current node (our first splitter) and if we are infront of it we draw the polygons behind it first (by making this function call itself but this time passing in the BackChild pointer), then draw the polygon actually stored at the node (the splitter) and then draw the polygons in front of the Current Node.Otherwise we are behind the current node so we reverse the order and draw the polygons infront of the current wall first etc.This allows us to always draw walls that are the farthest away first so they will always render correctly where ever we may be standing in the level.

If you are new to recursive code (functions that call themselves) just try and think of it as each time it calls itself it's really just like calling another completely seperate function because each time the function calls itself all the local data up to that point is stored on the stack so when the current instance of the function finally regains control again (because the previous function returned) all the data is still in the same state as when the function first called itself and the function resumes executing from that point as if the call had simply been a call to a printf command for example. Eventually when the recursive process ends because we have hit a NODE with no front or back children the function returns which gives control back to the previous instance of the function which made the call which returns to the previous function which made the call etc until we are right back at square one again and all the data has finally been poped off the stack and we are back in the first instance of the function which we manually called ourselves.Cool eh?.

Ok then lets check this works with a diagram that we can manually step through in our heads executing the above code.Below once again the walls are laballed in the order they became the splitters by the compiler.This of course defines the way they are all linked together.Below the RED dot 'C' is the camera position.The view direction of the camera is not important.

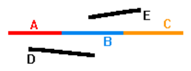
We first call our function passing in our ROOT Node A. Are we in front of Node A? No we are behind it so examine Node A's front child.This points to node B.Is the camera in front of Node B ? No so check Node B's front Child.This is Node Ca(remember our compiler had to split this line because it straddled B's split line) of which there are NO front or back child Nodes so we render Polygon Ca.We then return to the function that was processing Node B.We draw Node B's polygon and then check Node B's Back child pointer which points to node Cb.This node has no front or back Child Nodes so we render polygon Cb which returns to Node B's function which itself is now finished with its work.So node B's function returns and we are now back in the original function call we made to A.We have rendered all the walls in front of A so now we Render Polygon A.We now step back and examine Node A's Back Child pointer which points to Node D.The camera is in front of Node D so we draw the Back Nodes first but there are none so we simply Draw Node D and then Check for any Nodes in front of D.There is one, Node E.Is the camera in front of E? Yes so draw back wall first but there aren't any so simply draw Polygon E and then check for Nodes that are in front of wall D.There aren't any so we are done and our level has been drawn in the correct order.WOW.



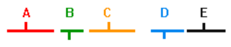
Congratulations you have just rendered a BSP Tree in your head.I hope by now you can really understand the rendering order.If we are in front of a polygon.Draw all polygons behind it first then draw the polygon itself then draw the polygons in front of it.If we are behind the polygon then do all that in the reverse order.You should also be able to see that each Polygon (also the splitter) also points to at most TWO other splitters one in front of it and one behind it.What we have in the end is a level where every polygon is linked in a way that is relative to neighbouring polygons.All we have to do at each splitter is a cheap test (ClassifyPoint) to discover which side of the Splitter our camera is on and take the appropriate actions by either walking down the back tree or the front tree at that Node.

Making a Solid Tree

Now one thing may be starting to confuse you a little.When talking about compiling a BSP tree so far we have only discussed what happens to polygons that are to the front and back of a splitter (they go in the splitters front and back lists) and what happens when a polygon straddles a splitter.But what happens when a polygon is on exactly the same plane as the splitter.The picture below shows an example of this and it happens very often with games like Quake and Unreal where there are a great number of long passage ways etc.



As you can see polygons A,B and C are on the same dividing plane so what should we do.Well my first implementation of the BSP Compiler decides that if any polygons shared a plane with the splitter those polygons would NOT be added to any list and instead would ALL be stored in the same node.Because these polygons are on the same plane and BSP Levels are depth sorted using planes we know that even if these polygons are over completely different sides of the game level from each other they can all be drawn at the same time.So in the above example the compiler would use for example Polygon A as a splitter but would link polygons B & C to polygon A in the node also.Only polygons D and E would be stored in the appropriate lists.This speeds up rendering for a number of reasons.Because each polygon sharing a plane no longer needs its own Node the tree has less nodes so is quicker to traverse.Also imagine in the above example that we are stood in font of polygon A which is the Root Node in the tree.The rendering algorithm would be changed slightly like this.Are we in front of Node A (YES).Render the Back Nodes of A first and then render ALL the polygons stored in Node A in one batch.Then draw front walls of node A.The great thing here is we can render ALL the planes polygons in one Node saving function call overhead etc.There is a problem though.Polygons can share a Plane but be FACING in opposite directions.Look at the picture below.



This is no problem either we just have two linked lists in our Node structure.In other words in the above example we would have a pointer at the Node containing the polygon used for splitting as usual but then we would have two lists one called 'SameFacing' which would be a linked list of all the polygons sharing the Node that are facing the same way as the Splitter polygon and another list called 'Opposite Facing' which would be a pointer to a list containing polygons facing the opposite direction to the Nodes splitter polygon.When you render the node you would use a function like so.

```
void WalkBspTree(NODE *Node,D3DVECTOR *pos)
{
POLYGON *shared;
int result=ClassifyPoint(pos,Node-> Splitter);

if (result==CP_FRONT)
{
shared=Node-> Splitter->SameFacingShared;
if (Node-> Back!=NULL) WalkBspTree(Node-> Back,pos);
lpDevice-> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,D3DFVF_LVERTEX,&Node-> Splitter-> VertexList[0],Node-> Splitter-> NumberOfVertices,&Node-> Splitter->Indices[0],Node-> Splitter-> NumberOfIndices,NULL);

while (shared!=NULL)
{
lpDevice-> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,D3DFVF_LVERTEX,&shared-> VertexList[0],shared-> NumberOfVertices,&shared-> Indices[0],shared-> NumberOfIndices,NULL);
shared=shared-> SameFacingShared;
}

if (Node->Front!=NULL) WalkBspTree(Node->Front,pos);
return ;
}

// this means we are at back of node
shared=Node->Splitter->OppositeFacingShared;
if (Node->Front!=NULL) WalkBspTree(Node->Front,pos);

while (shared!=NULL)
{
lpDevice-> DrawIndexedPrimitive(D3DPT_TRIANGLELIST,D3DFVF_LVERTEX,&shared-> VertexList[0],shared-> NumberOfVertices,&shared-> Indices[0],shared-> NumberOfIndices,NULL);
shared=shared-> OppositeFacingShared;
}

if (Node-> Back!=NULL) WalkBspTree(Node->Back,pos);
return;
}
```

I have highlighted the lines of interest.If we are in front of the current Node then we Render the Back Tree first and then the Nodes polygon and also render all polygons that are facing the same way.If we are Behind this node then we render the front Tree but we DO NOT bother rendering the actual Nodes polygon because we know we are behind so will be back face culled anyway.Then we render ALL polygons that are facing the opposite way to the Node(in other words they are front facing because the node itself is back facing.

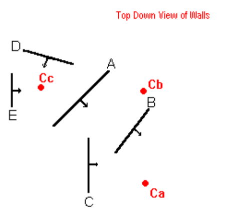
I had this up and running in my first BSP Demo and it did speed up rendering and BSP traversal by a small amount, but whilst I was actually writing this tutorial I discovered a way much better than the above way.In other words we are going to use a different and easier way than above.Although the above technique is faster for rendering it is not that great for collision detection.Actually that is not true, it can give you very accurate collision detection at good speed but often you do not need to know exactly what polygon (a wall for example) you have hit you only need to know that you have hit something.I did write a collision detection routine for the type of tree above and if you want it you can contact the webmaster and ask for it but what we are going to do now is use a slightly different type of BSP tree called a Solid Tree.If we compile our level into a Solid tree we can at lightning speed find out not only if we have hit a wall for example but also find out whether two points have a Line of Sight of each other.(COOL).So we are going to write our BSP compiler to compile a 'Node Based Solid BSP Tree'.

Solid BSP Tree ? What's the Difference

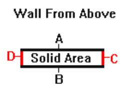
There are hardly any differences between a Solid BSP and a Normal BSP tree but the differences are important.This is them:-

1. The First difference comes in the way we deal with the last polygon in a Front or Back list during compilation.Up until now if a polygon has no front or back child Nodes we just set these pointers to NULL.What we will do now is if a Node has no front or back child we will create an extra node and attach it to the parents front or back list.This node though will have no polygon or splitter stored in it but instead we will add two more fields to our Node structure called 'IsLeaf' and 'IsSolid'.We will set 'IsLeaf' to true letting us know that NO polygons live in this node and if this new node is to the front of the parent (the parent being the last poly in the current poly list) we will set 'IsSolid' to false meaning that we are in an empty space (a space we can walk in).If this new node is to the back of the parent then we will set is 'IsSolid' to true.This means that the space behind a wall is bounded by walls so is solid and cannot be walked through.Now if you are totally confused you should be which is why I am going to show some pictures to expain it.

   The typical level we have been looking at and using as an example up until now has not been very realistic.I mean look at the picture below.When have you ever walked around a Quake level and just seen a load of walls without any BACKS to them.



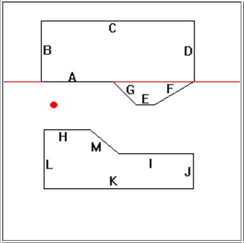
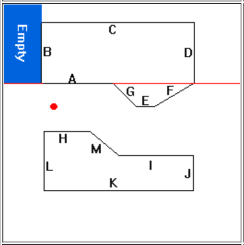
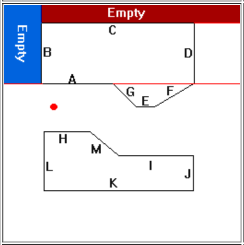
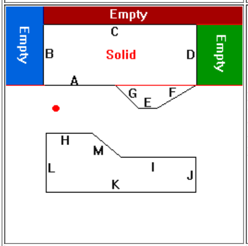
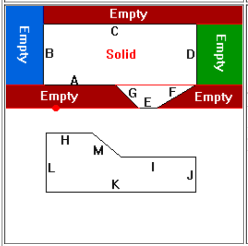
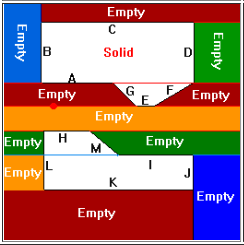
Because remember that the backs of walls can never been seen because they are back face culled.In other words if you walked behind wall A in the above picture and then turned around 180 degrees you would not see the back of wall A.Levels are designed in such a way that if you wanted a wall to stand alone in the middle of a level it would actually be made up of 4 or 5 faces like so:-



Can you see that the faces of the wall are all facing outwards from the center.This means that if we are Behind all those walls we are in a solid space.An we should not be there.We will look at more examples of this in a moment.

2. The second difference in a Node based Solid tree is in the dealing of polygons that share the plane with the splitter.Remember above we went through all the pain of keeping two linked lists at each Node containing same facing and opposite facing shared polygons.Well we dont have to do that now.All our compiler will do if we find a polygon in the List that is sharing the plane with the splitter is send it down the splitters Front List.This may sound strange but it is important to do this so we can correctly identify Solid and Empty Spaces.In other words if you are in a space that is bounded by outward facing polygons then you are in solid space and probably cast in stone. Anyway its a lot easier than the first way we looked at. You may be thinking that if we feed shared planed polygons down the front list then we are creating extra nodes and possibly extra splits and you would be correct.However the speed increase in performing collision detection and Line of Sight determination far far outweighs the traversal of a few extra nodes.

Lets run through in our heads then with the aid of a couple of pictures what our compiler will do .Below shows a picture of a typical piece of corridor section from a game level.Do not worry about the order that I choose the splits for now it is just for demonstation.The solid spaces are enclosed by outward facing walls and the walls are lablelled in the order we will choose them as splitters .To clarify it further the red dot shows a typical position where the player could stand.All the white space outside the two solid areas are empty space that the player can walk about in.The enclosed blocks represent solid areas as if they were made out of solid brick.We are now going to compile this level into a Solid BSP tree in our heads step by step.This should make understanding the compiler code simple after that.

	Ok then, our compiler function is passed all the polygons A through M to compile.It choose splitter A as the first split and stores it in the Root Node.We then test all the polygons against polygon A so that they are put into the respective Front and Back Lists.Node A's front List will contain Polygons E through M and A's back list will contain polygons B through D.Lets just concentrate on polygon A's back list for now.The Compiler function creates a new node and calls itself with this new node and poygon A's back list B,C,D.The compiler then chooses another wall as a splitter from this list .Wall B in our example.This wall is then stored in the new Node we passed in to the function and linked to Node A's Back Pointer.
	Now with Node B stored we then test the remaining walls in the list (C & D) against wall B and as you can see they both go in wall B's back list.There are no walls in front of Node B so instead of just setting it to NULL we create New node and add it to Node B's Front Pointer.This Node has no polygon though because it is a leaf.Instead we just set the variable 'IsLeaf' to true and because this leaf is in front of its parent node (B) we set 'IsSolid' to false.This is an empty space that can be walked in.If we traverse the tree and end up in this leaf then this means we are somewhere in the Blue box shown opposite.There are walls however in Node B's front list so the compiler function calls itself again once again passing in the Front list (walls C&D) and a newly created Node.
	This time through the compiler function has a choice of either wall C or wall D as a splitter.It chooses wall C and stores it in the newly passed node (which was attached to B's Back pointer).There are no walls in front of C so once again a new leaf node is created and added to Node C's Front Pointer .Once again 'IsSolid' is set to false because it is in front of C and 'IsLeaf' is set to true so we know that this is leaf and no polygon is stored here.Node C does have a back wall though (D) so once again the function called itself passing in wall D as the polygon list and also creates a new node that is attached to C's Back Pointer.This is passed to the function also and will end up containing D as the splitter because it is the only one left in the list.
	Now we choose splitter D because it is the only one left in the list.There are no front walls so once again a new leaf Node is created and attached to Node D's front pointer once again setting 'IsSolid' to false.If we end up in this leaf then we are somewhere within the green box shown opposite.However, there are no walls behind Node D either so once again we create a new leaf node and attach it to Node D's back list but because this leaf is BEHIND node D we set 'IsSolid' to true.If we end up in this leaf then we are in the soild area bounded by walls A,B,C,D and this is not allowed. Its important to realize how a leaf is bounded by its parent Nodes to create an atomic space.(convex Hull in other words).The function returns to Node C's function which in turn returns to Node B's function which ends up back at A (Root node and the first instance of the function that we called) where we have only processed the back list.Now we process Node A's front list which if you remember consited of walls E through M.
 Empty Empty C B Solid D Empty A Empty G E F Empty H M L I J K	The compiler function now recursivly calls itself again with Node A's front list.We choose wall E as the splitter,store it in a new node and again attach it to Node A's front pointer.The remaining polygons are tested against wall E and we end up with a back list of polygon F & G and a front list for E containing polygons H through M.Lets have a look at the back list first.The compiler function calls itself passing in the back list of E and a new node and this time wall F is selected and stored in the new node and the new node is conected to Node E's Back pointer.There are no walls in front of wall F so once again a new leaf is created and added to Node F's Front pointer once again setting 'IsLeaf' to true and because this leaf node is in front of wall F once again 'IsSolid' is set to false because this is an empty space.If we end up in this leaf we are in the dark red box to the right of wall F. G is the last wall sent which has no front or back walls.Once again a front leaf is created as being empty and attached to G's front Pointer and a Solid leaf is created and attached to G's back pointer.This leaf represents the solid are bounded by walls E,F & G.You can now see the top solid area is now completely represented in the tree.
 Empty Empty C B Solid D Empty A Empty G E F Empty Empty Empty H M Empty Empty L I J K Empty Empty	Here is this section finished being compiled.You step through the other splits in your head and make sure that you fully understand the THEORY of whats going on even if you do not know how to code it yet.The most important point to remember is that a splitter is not in ANY of its own lists.For example wall A may look like it is behind wall B but remember that wall B is a child of A and can only divide wall A's back subspace.Wall A is not in B's lists at all.Just like wall C is a child of B so wall C can not SEE B because it can only divide wall B's back sub space.This is the very core process to spacial sub division other wise the compiler could not section off bits of space.

Lets now have a look at how our Node structure will look.Remember a BSP tree is nothing more than a mass of Nodes pointing to each other in an order that makes sense.

```
struct NODE
{
POLYGON * Splitter;
NODE * Front;
NODE * Back;
BOOL IsLeaf;
BOOL IsSolid;
};
```

Nothing much to explain here we have just added two variables to signal if this is a leaf.We need to know this so we do not try and render a polygon at this node because 'Splitter' will be set to NULL for a leaf.We have discussed what 'IsSolid' does above.

Other Benefits of a Solid BSP Tree

Now I do not want to get into collision detection with a BSP tree to heavily yet but lets just have a look at some of the benefits.Imagine the above level was much larger and instead of there only being two or three walls either side of the root splitter A imagine instead there were 5000 polygons (therefore 10000 Nodes in our BSP Tree.We will just forget about the extra polygons created from splits for now).Imagine we want to check if we have are about to move into a wall. Without the BSP you would have to test each polygon all 10000 of them to see if you are about to intersect with one.Not only that but polygon collision detection normally involves a lot of cpu intensive maths like square roots etc.Well its just not going to happen is it.Now lets have a look at this the BSP way.We have the point we are about to move to.We now start testing it against the polygons (Nodes) in the BSP.Is it to the front or back of Node A. If its to the front of Node A then every single polygon behind Node A is rejected after just one test because we can not possibly hit them.We have just rejected 5000 polygons after one simple test.Because we are in front of A we go down A's front tree doing the same at each Node.At each Node we are rejecting half that nodes polygons (this is assuming a perfectly balanced tree which is hardly ever the case but serves us for this example).Rejecting half a Nodes polygons at every node will make you eventually pop out at a leaf after about 15-25 Nodes even in a big level.Once you pop out at a leaf (assuming we are using a Solid BSP as described above as we will be) we simply check that leafs 'IsSolid' variable and if it is set to true we can not walk that way.The great thing about this though is the fact that we are loosing HALF each nodes polygons at every test.Not only that but because we have a solid tree with leaves that tell us they are solid we do not even have to do any complicated ray intersection.This is lightning fast.Not only that but exactly the same technique can be used for detecting Line if Sight.Later on once we have written our compiler we will write a collision detection/line if sight function to use the tree the way we have described. Now that does sound tempting does it not?

Note:
The above technique is a simplified version of how collision detection and line of sight works.Ofcourse it does not allow for the fact that the point you are about to move to is empty but there may be something in between your current position and the position you are about to move to.We will cover this in detail later when we write our LineOfSight function and its still very easy to solve but I do not want to confuse you any more than you probably already are at the moment.

I hope by now you have a firm grasp on the whole BSP Tree theory and how it sub divides space.Hopefully when you see the actual code for the Compiler and Renderer all the un answered questions will start to fall into place.The theory lesson is now over and it is time to look at some raw code.We are now going to go step by step through the lines of code used in my BSP Demo.This demo is downloadable at the bottom of this page and is a fully working 3D BSP compiler that compiles a list of polys into a BSP Tree and then lets you walk around the level it creates.

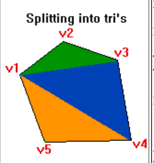
A Slight Diversion - Setting up the Geometry for our BSP Demo

Many BSP Tree resources on the web only provide you with theory or little bits of code and you are left worrying how exactly you work this code into your own program.To hopefully stop that happening this time we are (together) going to write a fully fledged BSP Demo.This demo will actually Compile the BSP Tree and then render it and let you walk around it just like in a real game.You can then use this code as a basis for your program. However in order for us to write a BSP Compiler we must have something for it to compile otherwise how will we know if it works.What we are now going to do is take a slight detour away from BSP Trees for just a couple of minutes while I explain how the polygons for the demo are created and stored.The reason I am explaining this and not just letting you download the source for a look (which you can do at the bottom of this page anyway) is because the way the Polygons are stored is of major importance to HOW we write our compiler.So we are going to take a look at a few initialization routines that create the polygons and link them altogether in a Linked List to send to the BSP Compiler.

The first thing for us to look at however is the structure we will use to hold Polygon Data.Each polygon structure will represent one polygon.It looks like this.

```
struct POLYGON
{
D3DLVERTEX VertexList[10];
D3DVECTOR Normal;
WORD NumberOfVertices;
WORD NumberOfIndices;
WORD Indices[30];
POLYGON * Next;
};
```

The polygon structure holds an array of D3DLVERTEX structures which describes the vertices in the polygon.The actual Polygon will internally be rendered using Indices into the vertex array.This is what the Indcies array is for in the structure.This will allow us to simply pass any shape (convex) polygons to our AddPolygon function (we will write this in a minute) containing any number of vertices and our AddPolygon function will break each polygon into multiple triangles so it can be passed to the DrawIndexedPrimitive function.If you are not familiar with Indices it may well be worth looking them up in the SDK docs.Indices allow you to build multiple triangles using shared vertices .Below shows an example of such a polygon.



You can see that this polygon is made up of 5 vertices but has to be broken into three triangles in order to be rendered.Although this shape has 5 vertices it will need 9 indices to describe it to the renderer because the renderer needs to render triangles and each triangle needs 3 points.The nine indices would be as follows v1,v2,v3,v1,v3,v4,v1,v4,v5.You can calculate the number of indices needed with the following equation:-

NumberOfIndices=(NumberOfVertices-2)*3;

This of course has nothing whatsoever to do with BSP Trees but I am explaining how our BSP Compiler expects to see the polygons represented.
This polygon can be rendered using the following line:-

```
lpDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,D3DFVF_LVERTEX,pVertexList, 5 ,pIndices, 9 ,NULL);
```

As you can see the above shape may render three triangles and that would normally mean 9 vertices have to be transformed and lit.However because we are using indices to describe the triangle only five vertices actually get transformed and lit and are re used by faces that share them.We will look at the code responsable for breaking up the polygons into multiple triangles later when we write our AddPolygon function which will add polygons to the scene.

Once you understand the BSP Compiler it will be easy to modify it to take any polygon format you want.What makes our POLYGON structure weird above is that it is multi purpose because in our example we compile and render the tree all at runtime (because the level is not very big).Also in our POLYGON structure you can see that there is a pointer to a POLYGON called 'Next'.This member is used in the compilation process as we will send all the polygons in our scene to our BSP Compiler in the form of a Linked list.This means that polygon 1 will have a Next pointer that points to Polygon 2 and polygon 2's Next pointer will point to Polygon 3 etc.Eventually the last polygon in the list will have its Next pointer set to NULL.The AddPolygon function we will write will also handle the linking process.We simply pass in the new polygon and the previous polygon and it will link the previous polygons Next pointer to point at the newly constructed polygon. One point worthy of note is that this pointer is only used in BSP compilation and is not used once the tree is compiled .That is what I meant by a multi purpose structure that is used for compiling and rendering.If you saved your BSP to disk and loaded it back in you could ommit this field from the POLYGON structure.You may also notice that the POLYGON structure has a Normal member which is just a vector that is perpendicular to the polygon and describes the direction the polygon is facing.Once again our AddPolygon function will generate the Normal for the polygon automatically and stuff it in the polygon structure.(Hopefully you have read our Cross Product & Dot Product tutorials and the word Normal is not new to you.If you do not know what a normal is then stop reading this now and read our Cross Product and Dot Product tutorials or none of this will make any sense to you).

We now know how we are going to store the polygons and pass them to the BSP Compiler but how are we going to create the polygons in the first place.We certainly are not ready to talk about importing levels from various editors which is just as well because I know nothing about the various file formats of the various BSP Level editors (although you can find some info on how Quake levels are stored at www.flipcode.com which also happens to be a bloody excellent site, well done kurt).

Setting up the Polygons ready for Compilation

In my BSP demo (that can be down loaded at the bottom of this page) I needed a way to quickly knock up some polygons to test my compiler with.I defined an array like so in my code.Below shows just a fragment of the array as staring at lots of zeros and ones does not exactly make great reading unless you are a Binar.(ST:TNG series 1 I think, well Tasha Yar was still alive anyway)

```
BYTE BSPMAP []=
{0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,2,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0,
0,2,0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,
1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,1,
0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,
0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
0,1,1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,3,1,
0,2,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,
1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,
0,1,0,0,0,0,1,2,0,0,0,1,0,0,0,1,0,0,0,1,
0,1,0,0,0,1,2,0,0,0,0,1,1,0,0,0,0,0,0,1,
0,1,0,0,0,1,0,0,0,0,0,3,1,0,0,0,0,0,0,1,
0,1,0,1,1,2,0,0,0,0,0,0,1,0,0,0,0,0,0,1,
1,2,0,0,0,0,0,0,1,0,0,0,1,1,1,1,0,0,0,1,
1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,
1,0,0,1,2,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,
1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
```

With this array we are going to build a function that will loop through each element of the array.One entry in the array represents 1 unit squared of 3D World Space so when a 1 is encountered with zeros all around it 4 polygons are created for the four walls of a cube all facing outwards. If a number other than zero is to any side of the 1 then a wall is not built for that side because it is covered up by an adjacent wall.So in other words each 1 digit represent a 1x1 cube in world space and we will texture the four walls (n,s,e & w) with some brick wall texture.The 2 and 3's in the map were just put in there by me at the last minute to give the map a few angles.A 2 digit represents a NE facing wall and a 3 represents a NW facing wall.I didn't bother to create SW or SE but you can do that yourself if you want.Just build them the same as the NW and NE walls but reverse the winding order of the vertices.The polygons are created in WORLD space with the center of space 0,0,0 being in the middle of the map and each digit as I said represents 1x1 unit in space.The full grid is 20x40 in dimensions (not all shown here).

Note:-

For those of you reading this and thinking what a dodgy way to make a level you are of course right.But all I needed was an easy way to create lots of polygons to throw at the BSP Compiler.Also the last thing we need right now is a tutorial on how to import levels from various editors which actually is just as well because I have not got a clue.

After we have set up our 3D environment (initialized D3D etc) the first function we call is our InitPolygons Function which looks like so:-

```
void InitPolygons(void)
{
D3DLVERTEX VERTLIST[4][4];
PolygonList=NULL;// this is a GLOBAL pointer
POLYGON *child=NULL;
int direction[4];
for (int y=0;y< 40;y++)
{
for (int x=0;x< 20;x++)
{
ZeroMemory(direction,sizeof(int)*4);
int offset=(y*20)+x;
// check what the digit is in the current map location
if (BSPMAP[offset]!=0)
{
```

```
if (BSPMAP[offset]==2)// North East Wall
{
VERTLIST[0][0]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,0,0);
VERTLIST[0][1]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE( 255, 255, 255),0,1,0);
VERTLIST[0][2]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE( 255, 255, 255),0,1,1);
VERTLIST[0][3]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,0,1);
direction[0]=1;
}
if (BSPMAP[offset]==3)// North West Wall
{
VERTLIST[0][0]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE( 255, 255, 255),0,0,0);
VERTLIST[0][1]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,1,0);
VERTLIST[0][2]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,1,1);
VERTLIST[0][3]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE( 255, 255, 255),0,0,1);
direction[0]=1;
}

if (BSPMAP[offset]==1)// Its a Standared wall
{
if (x > 0)
{
if (BSPMAP[offset-1]==0)// if theres nothing to the left add a left facing wall
{
VERTLIST[0][0]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,0,0);
VERTLIST[0][1]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE(255,255,255),0,1,0);
VERTLIST[0][2]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE(255,255,255),0,1,1);
VERTLIST[0][3]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,0,1);
direction[0]=1;
}
}
if (x < 19)
{
if (BSPMAP[offset+1]==0)// if there is nothing to the right add a right facing wall
{
VERTLIST[1][0]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE(255,255,255),0,0,0);
VERTLIST[1][1]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,1,0);
VERTLIST[1][2]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,1,1);
VERTLIST[1][3]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE(255,255,255),0,0,1);
direction[1]=1;
}
}
if (y > 0)
{
if (BSPMAP[offset-20]==0)// if there is nothing south add a south facing wall
{
VERTLIST[2][0]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,0,0);
VERTLIST[2][1]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,1,0);
VERTLIST[2][2]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE(255,255,255),0,1,1);
VERTLIST[2][3]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)+0.5f),RGB_MAKE( 255, 255, 255),0,0,1);
direction[2]=1;;
}
}
if(y < 39)
{
if (BSPMAP[offset+20]==0)// if there is nothing to the north add a north facing wall
{
VERTLIST[3][0]=D3DLVERTEX(D3DVECTOR(x-10.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE(255,255,255),0,0,0);
VERTLIST[3][1]=D3DLVERTEX(D3DVECTOR(x-9.5f,3.0f,(20.0f-y)-0.5f),RGB_MAKE(255, 255, 255),0,1,0);
VERTLIST[3][2]=D3DLVERTEX(D3DVECTOR(x-9.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,1,1);
VERTLIST[3][3]=D3DLVERTEX(D3DVECTOR(x-10.5f,0.0f,(20.0f-y)-0.5f),RGB_MAKE( 255, 255, 255),0,0,1);
direction[3]=1;;
}
}
}// end for if offset==1

// build the polygons

for (int a=0;a<4;a++)
{
if (direction[a]!=0)
{
if (PolygonList==NULL)
{
PolygonList=AddPolygon(NULL,&VERTLIST[a][0],4);
child=PolygonList;
}
else
{
child=AddPolygon(child,&VERTLIST[a][0],4);
}
}//
}////
}// end for if offset!=0
}
}
BSPTreeRootNode=new NODE;
BuildBspTree(BSPTreeRootNode,PolygonList);

}
```

With the exception of the last two line which we will cover later, this function does not take much explanation.All it does is loops through the BSPMAP array,reads in the digits and sets up some vertices for the walls.Notice how each wall has four vertices described in a clockwise manner and how the position into the map is used to place each vertices in world space.At each digit that returns '1' we check the area to the left,right,north and south of the digit if there are any zeros at that location it means this wall is an external wall and can be seen by the player so vertices are setup at that location defined in a way that is clockwise when you are looking at it.We then loop through the Direction array and for each element set to 1 we know some vertices have been set up for a wall so we pass these four vertices to our AddPolygon function (see this in a minute).The AddPolygon function takes a pointer to a PARENT polygon, array of vertices and the number of vertices in the array.Notice above that the PolygonList pointer is a Global pointer that will point to the ROOT of our linked list of polygons.If this polygon==NULL then this is the first polygon and had no parent so we just pass NULL as the parent.For every other polygon bar the first though we pass in the previously created polygon as the new polygons parent.The AddPolygon function returns a pointer to a POLYGON structure that has had its Normal generated,Has been split into multiple triangles and has been linked to the parent passed into the function by the parents 'Next' pointer.This new polygon then becomes the Parent in the call to the next polygon to be created.In other words, the first polygons 'Next' pointer points to the second polygons 'Next' pointer and so on.

By the time we have reached the end of the for/next loops that loop through the rows and columns of the BSPMAP we have all the POLYGONS generated and linked together in a linked list.All we need is a pointer to the ROOT of this list so we can pass it to the BSP Compiler.We have this in the global pointer 'PolygonList' which we used to store the POLYGON structure returned from the FIRST call to AddPolygon.We will later pass this List of polygons to our BSP Compiler that will compile a BSP Tree out of them.

Also notice above that the vertices are of type D3DLVERTEX and the diffuse color is just set to bright white,specular to zero and the texture coordinates are set so any texture loaded when the polygons are rendered will be stretched over the entire polygon.This is of course because our texture coordinates range from 0,0 (top left) to 1,1(bottom right).Texture coordinates are in the range of 0 to 1 so this means map the entire texture over the wall like a table cloth.In our Demo I have used a texture of a brick wall to map onto all the walls so I just set this texture at start up in texture stage 0 and leave it there so all walls are rendered using it.

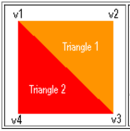
The obvious missing piece to the puzzle in the above function 'InitPolygons' is the call to the AddPolygons function.This function actually does quite a bit of work.It generates a Normal for the Polygon , but more importantly it constructs a polygon using multiple triangles to represent our desired shape.For example, we are passing in square walls with four vertices so this function has to break this up so that it can be rendered as two triangles.But this is just standard D3D stuff anyway.Lets have a look at it a couple of lines at a time.

```
POLYGON * AddPolygon(POLYGON* Parent,D3DLVERTEX *Vertices,WORD NOV)
{
int loop;
POLYGON * Child=new POLYGON;
Child-> NumberOfVertices=NOV;
Child-> NumberOfIndices=(NOV-2)*3;
Child-> Next=NULL;
for (loop=0;loop< NOV;loop++)
{
Child-> VertexList[loop]=Vertices[loop];
}
```

The first thing this function does is create a new POLYGON structure to hold the polygon that is about to be made by this function.We then copy over the number of vertices in the polygon (NOV) and also calculate how many Indices we will need.Remember we looked at this little formula earlier but for all the walls in our demo (which contain 4 vertices) this will be equal to (4-2)*3=6 indices.This is because a square wall will have to be broken down into two triangles.Each triangle is made up of 3 points 3*2=6 obviously.We set all the other pointers in the POLYGON structure to NULL for now .We then copy all the vertices from the array passed into this function over into the polygon structure.Lets have a look at the next bit.This is the bit where we are going to fill in the indices array with the six points that make up the two triangles that make up our original square wall.Actually you should make sure you understand this code as this code will break any complex convex polygon into simple triangles and will probably be used many times throughout your applications development.For example the X File format stores its polygons in complex format so if you load X files in you will have to use code like this to break the faces of an object into triangles.Remember that in the following listing we are calculating the contents of the Indices array.Each element in this a ray is a numerical reference to a vertex in the vertex list.Heres the code.

```
//calculate indices
WORD v0,v1,v2;
for (loop=0;loop< Child-> NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
Child-> Indices[loop*3]=v0;
Child-> Indices[(loop*3)+1]=v1;
Child-> Indices[(loop*3)+2]=v2;
}
```

Not very long this bit is it when you think how clever it is.Can you see whats happening each time through the loop? Get a pen and paper and try it drawing a square with four vertices (or any convex shape) and have a look at the indices list you get.Actually forget the pen and paper I will fire up my paint package and show you exactly what is happening to our wall (square polygon) in this loop.

	You can see that the two triangles are created out of the square yet only four Vertices are needed to describe them.The Indices list for our wall would hold 6 entries like so:- v1,v2,v3,v1,v3,v4 Three Indicies a triangle so two triangles need six Indices.

So our AddPolygon function has now set up the POLYGON structure and filled in the Indices array, the vertex Array and filled in the Number of vertices and Indices etc.Our next task is to Generate a Normal for the polgon using two edges of the polygon and the Cross Product.(If you are unfamiliar with the Cross Product please read our Tutorial).As I hope you know at this point a Normal is a vector that describes the way the polygon is facing.Although we do not use this pre calculated Normal to render the polygon we will need it in many places to calculate Dot Product operations later on.

```
// generate polygon normal
D3DVECTOR * vec0=(D3DVECTOR *) &Child->VertexList[0];
D3DVECTOR * vec1=(D3DVECTOR *) &Child->VertexList[1];
```

```
D3DVECTOR * vec2=(D3DVECTOR *) &Child->VertexList[Child->NumberOfVertices-1];// the last vert
D3DVECTOR edge1=(*vec1)-(*vec0);
D3DVECTOR edge2=(*vec2)-(*vec0);
Child->Normal=CrossProduct(edge1,edge2);
Child->Normal=Normalize(Child->Normal);
```

As you can see the generated Normal is stored in the POLYGON structures 'Normal' field which is just a Vector.

Our last job is to link this child to the parent (previous polygon we created) which was passed into this function as a parameter.This allows each new polygon we create (by calling this function) to LINK to the previous polygon created using the previous polygons 'Next' pointer.We must check that the Parent passed in is not NULL though because remember the first time we call this function the first polygon will not have a parent so we will pass in null.Also notice that this function returns a pointer to the child which can then be used as the parent the next time this function is called to create a new polygon. The code in the previous function 'InitPolygons' should now make a lot more sense as you should now see how the linked list is created.Heres the last bit of code that assigns the parent.

```
if (Parent!=NULL)
{
Parent->Next=Child;
}
return Child;
}
```

Just for completeness here is the above code to AddPolygon in its entirety.

```
POLYGON * AddPolygon(POLYGON* Parent,D3DLVERTEX *Vertices,WORD NOV)
{
int loop;
POLYGON * Child=new POLYGON;
Child-> NumberOfVertices=NOV;
Child-> NumberOfIndices=(NOV-2)*3;
Child-> Next=NULL;
for (loop=0;loop< NOV;loop++)
{
Child-> VertexList[loop]=Vertices[loop];
} //calculate indices
WORD v0,v1,v2;
for (loop=0;loop< Child-> NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
Child-> Indices[loop*3]=v0;
Child-> Indices[(loop*3)+1]=v1;
Child-> Indices[(loop*3)+2]=v2;
} // generate polygon normal
D3DVECTOR * vec0=(D3DVECTOR *) &Child->VertexList[0];
D3DVECTOR * vec1=(D3DVECTOR *) &Child->VertexList[1];
D3DVECTOR * vec2=(D3DVECTOR *) &Child->VertexList[Child->NumberOfVertices-1];// the last vert
D3DVECTOR edge1=(*vec1)-(*vec0);
D3DVECTOR edge2=(*vec2)-(*vec0);
Child->Normal=CrossProduct(edge1,edge2);
Child->Normal=Normalize(Child->Normal);
if (Parent!=NULL)
{
Parent->Next=Child;
}
return Child;
}
```

Ok, we have waffled on forever here when this stuff has nothing to do with compiling the actual BSP Tree but it was important (I feel) that you understood the way the polygons were represented in order to understand the BSP Compiler that will work on them.Just remember that each Polygon structure points to the next one with its 'Next' pointer and we have a pointer to the root (First poly in the List) .

Lets Build a BSP Compiler Function

Ok then, lets get back on track.We have a list of polygons and we want a function that is going to build a BSP Tree. Before we call our function we have to allocate memory for a Single Node.You may have noticed earlier in our InitPolygons routine that this is where I allocated the memory for the Node .Remember that Node structure is exactly what I showed you earlier.It has just five fields.A pointer to a POLYGON structure , two pointers to front and back nodes and two boolean variables stating whether this node is a leaf and if its solid or not.We pass this newly created Node (still empty as none of its fields have been set up yet) to our BSP Compiler function along with our list of Polygons.When the function returns the BSP Tree will be built and the Node we passed into the function will be the Root node in the tree.This is the Node we will pass to our render and line of sight functions that will take care of traversing down the tree to all the other Nodes in the correct order. Just in case you do not feel like scrolling back up to the InitPolygons function the last two lines of that function were as follows.

```
BSPTreeRootNode=new NODE;
BuildBspTree(BSPTreeRootNode,PolygonList);
```

That is all we have to do to setup the tree once our BuildBspTree function is written.

Before we write the main Compiler function (which is actually quite small) we need a few helper functions that the compiler function will need to call.The first function we will write is a small function called 'ClassifyPoint'.You may remember that we will also need this function when we render the tree also because it checks which side of a Plane a point lies on.A point is said to lay behind a plane if it is on the opposite side of the plane to the side the plane Normal is facing.In other words if a plane faces left (because the Normal faces left) a point on the right side of the plane will be Behind the plane.If it is on the right side of this plane then the point is said to lay In Front of the plane.If the point lay exactly on the plane then it is said to be 'Coincident' with the plane and is neither in front or behind the plane .This means we need a function that will take a Point in 3D space and a Plane and return one of three results.The result I have defined as global integers are CP_BACK,CP_FRONT & CP_ONPLANE. The correct name for the 3rd integer is really CP_COINCIDENT but I think CP_ONPLANE is much easier to understand.This means our 'ClassifyPoint' function will need to return one of these three integers as a result.

In case you are starting to panic that you do not know how to construct a plane you do not have to.Every polygon lies on an infinite plane and as long as you know the Normal for the plane (the polygon Normal will do) and we also know the position of a point that is known to be on the plane (any vertex in the Polygon will do) we have a Plane and can use the DotProduct to find the results.This means our function will take a Point in 3D space and a Polygon as its parameters.The function will construct the plane from the polygon we pass in .Here is the code:-

```
int ClassifyPoint(D3DVECTOR *pos,POLYGON * Plane)
{
float result;
D3DVECTOR *vec1=(D3DVECTOR *)&Plane-> VertexList[0];
D3DVECTOR Direction=(*vec1)-(*pos);
result=DotProduct(Direction,Plane->Normal);
if (result< -0.001) return CP_FRONT;
if (result> 0.001) return CP_BACK;
return CP_ONPLANE;
}
```

This is just standard Dot Product stuff so you should know this by now so ill just go over it briefly.First we calculate the direction vector from the position in space to a Vertex in the polygon.We then pug this into the Dot Product along with the polygons Normal and we have returned the distance to the plane.In this code above if the return value is negative (< 0) then the point is in front of the plane.If the result is a positive number then we are Behind the polygons plane.If the result is zero the point is on the plane (just like one of the polygons vertices would be).Notice that because of the accuracy problems with floating point numbers (rounding errors) it is possible for the point to be on the plane but the result not be EXACTLY zero.For example 0.00001.Because of this problem we make the Plane Thicker by using a Fuzzy compare.That's what 0.001 and the -0.001 is all about.It is a compare that has tolerance for floating point rounding errors.

That's our first helper function done.This function will be used both in our compiler function and our Render function.For example, in our render function we will call ClassifyPoint at each Node in the BSP Tree and pass in the Nodes polygon and the current Camera position.If we are in front of the current nodes polygon(splitter) we will go down the Nodes Back tree first and if we are Behind the Nodes polygon we will go down the current Nodes front tree first.

The next function we are going to write will allow us to test an entire polygon against a plane.This function will be called 'Classify poly' and will step through each vertex in the input polygon and classify that point with the plane.This will be used by our BSP Compiler function for finding out if a Polygon needs to be split because it straddles another polygons Infinite plane.If all the points in the polygon are behind the plane (splitter) then the polygon does not need to be split and can be added to the Nodes back list.If all the points are in front of the plane then the polygon once again does not straddle the plane and can be added un altered to the Node front list of polygon.If however some of the points are behind and some of the points are in front of the plane then the polygon will have to be split into two and the front bit gets added to the front list and the back bit gets added to the back bit.We will talk a little later about what happens when the polygon is actually sharing the plane with another polygon as we will make a special case for this.However if you wanted to you could just choose to send it down any list , front or back.

Here is the code to 'ClassifyPolygon' which just does what Classify Point does but for every point in the polygon.I could have made ClassifyPolygon call ClassifyPoint for each vertex but the code is not that big so I duplicated it in the ClassifyPolygon routine as this will save on function call over head.

Note:-

There is something a little bit strange about OUR 'ClassifyPoly' routine.Normally a function like this would return either CP_FRONT,CP_BACK,CP_SPANNING or CP_ONPLANE.The first three it will return pointing out that the polygon is either to the Front or Back of the plane or is spanning the plane and needs to be split.However you may remember that Rule 1 for a Solid BSP was that if a polygon shares a Splitters plane then it is to be treated like a Front face and added to the front list.Therefore the following code returns CP_FRONT instead of CP_ONPLANE.If you are NOT making a Solid BSP you would return CP_ONPLANE and make it share the node as shown earlier.

```
int ClassifyPoly(POLYGON *Plane,POLYGON * Poly)
{
int Infront=0;
int Behind=0;
int OnPlane=0;
float result;
D3DVECTOR *vec1=(D3DVECTOR *)&Plane->VertexList[0];
for (int a=0;aNumberOfVertices;a++)
{
D3DVECTOR *vec2=(D3DVECTOR *)&Poly->VertexList[a];
D3DVECTOR Direction=(*vec1)-(*vec2);
result=DotProduct(Direction,Plane->Normal);
if (result> 0.001)
{
Behind++;
```

```
                                   }
                                  else
                           if (result< -0.001)
                                   {
                                Infront++;
                                   }
                                  else
                                   {
                                OnPlane++;
                                Infront++;
                                Behind++;
                                   }
                                   }
if (OnPlane==Poly-> NumberOfVertices) return CP_FRONT;// this would nomrally be CP_ONPLANE
                  if (Behind==Poly-> NumberOfVertices) return CP_BACK;
                 if (Infront==Poly-> NumberOfVertices) return CP_FRONT;
                           return CP_SPANNING;
                                   }
```

Not a lot to explain here.It just classifies each point in the Polygon with the Plane (which is passed in as a polygon itself) and for each point behind,in front or on the plane a counter is kept.At the end of the function if all the vertices are not on the plane and all the vertices are not in front of the plane and all the vertices are not behind the plane then it means that vertices must be on opposing sides of the plane meaning a split will have to be performed.In this case the function returns CP_SPANNING.This will tell our compiler function that the polygon needs to be split.

There are two more helper functions that need to be written also.One of them is 'SplitPolygon' which our compiler will call to split a polygon if it straddles a Nodes Polygon(splitter).In other words if ClassifyPolygon returns CP_SPANNING.The SplitPolygon routine though we will look at AFTER we have written the BSP Compiler but just be aware of what it does.Its prototype is as follows:-

```
void SplitPolygon(POLYGON *Poly,POLYGON *Plane,POLYGON *FrontSplit,POLYGON *BackSplit);
```

This function takes to Polygon and splits the first polygon against the Plane of the second polygon.The function splits the first polygon into two new and unique polygons and pointers are returned by the function in the FrontSplit and BackSplit pointers.As I said we will look at this function later on after we have written the compiler function because I want to concentrate on the BSP issue first.

The last helper function we will need to write that will be used by our BSP Compiler function will be called 'SelectBestSplitter' and will be passed a group of polygons and decide which one of these will make the best choice to become the Next splitter.Once the splitter is chosen it will be placed in a new Node and all the remaining polygons in the list will be assigned to either its front or back list .

How Do We Choose the the Best Splitter

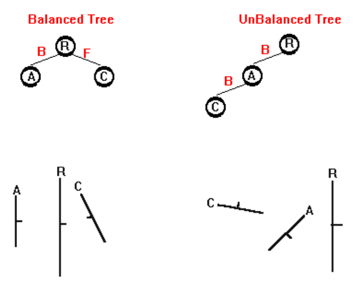
As you have no doubt grasped by now our compiler will call itself repeatedly with sets of polygons and each time will choose a polygon out of that list to become the Splitter for that node.All the other polygons are assigned to front or back lists which themselves will be split and so on until every polygon has been choosen as a splitter.The last polygon left in a list will not actually split anything but will just create two LEAF nodes that it will add to its own front and back Pointers .The back leaf will be a Solid leaf and the front leaf will be an Empty leaf.This Node will point to NO other nodes.But how do we decide each time our BSP Compiler function is called which single polygon should be used next in the list to split the others by.We could actually randomly pick a splitter each time from the list but there are good choices and bad choices.

Every time we select a new Splitter to divide a given subspace any polygons in the list straddling over the splitter (Splitter Polygons Plane) will have to be split into two.That means all of a sudden one polygon becomes two polygons.Imagine for example that you choose a splitter that intersected every other polygon in the list.Every polygon in the list would have to be split in to two.If this was your entire polygon set then with the first call to our function we have just DOUBLED the polygon count.If you have not caught on yet THIS IS BAD.Now imagine that those split polygons have now been assigned to front and back list of the splitter and the BSP Compiler function calls itself to find another splitter to divide the sub lists.Image the function chooses a splitter in the front list that once again intersects every other polygon in the front list and all those polygons which are themselves splits from the original list will have to be split doubling the size of this list etc.Imagine the same for the original back list and within only TWO calls to our BSP Compiler function the polygon count of your 3D world has just been QUADRUPLED. Then remember that the BSP Compiler function will call itself repeatedly until no polygons remain in front or back list (in other words the function will call itself a lot of times for a 10000 polygon level) and you can imagine all the splits taking place at each level in the tree would leave you with a polygon count that would only allow frame updates Sundays.

Now of course the above example is taking it to the extreme but the point is that some of the polygons in your levels will need to be split into two and these splits may have to be further split into two pieces further down the tree.So to build the ultimate BSP Tree then we need our function to calculate which order the Splitters should be chosen that creates the least amount of splits. You would think this would be a case of building a BSP tree in every single combination at every single node but you would not believe the number of combinations that would exist in a simple 5000 set of polygons.If you set your computer to figure it out now you would have died of old age before it had finished (I'm not joking).This means clearly we are going to have to forget about building the perfect BSP Tree and just try and get a good one.What we will do is each time the BSP Compiler function repeatedly calls itself with polygon lists we will just loop through and choose each one as a splitter in turn and record how many polygons it splits.In other words we will decide at each level in the tree which polygon splits the least amount of polygons in the immediate list.Although at first this may sound the same as the perfect tree you must remember that choosing a splitter that causes not many splits at one level of the tree could actually cause more splits further down the tree than perhaps a splitter which first of all split more polygons. However there is not a lot we can do about this so this is the technique we shall use. All sorted then ? Not quite.

What is the Balance of a Tree?

Although choosing a splitter that creates the least amount of splits is the property most important in my opinion there is also the Balance of the tree to consider.A perfectly balanced tree (which is hardly ever possible to create) is a bsp tree with exactly the same number of nodes behind it and in front of it.The diagram below shows a balanced and an unbalanced tree .

Example of a Balanced and Unbalanced tree showing both the Tree and Wall layout  Balanced Tree UnBalanced Tree B R F A C B R B A C A R C C A R	As you can see the left tree has exactly one node behind the root and one node in front of the root.The right most tree has no nodes in front of it but has two nodes behind it.One very important thing to notice here is that the left most tree has a depth of 1 but the right most tree actually has a depth of 2.By the depth I mean in the diagram the number of nodes vertically descending from the root node.The deeper a tree (the more nodes you have to traverse before you reach a node) the more time it takes to traverse the tree.Sometimes frame rate consistency is more important than out right speed.For example it is better to have a game engine run at 30 fps consistently throughout the level than have a level run at 90fps in some places and drop to 7fps in others.This is where the Balance of the tree comes into play.Although the balance of the tree does not effect OUR rendering routines because we will be using the painters algorithm to draw back to front which means every node will be visited anyway (until we implemement Fustrum rejection bounding boxes.more on this later), the balance will effect the collision detection routines we write later.Look at the right most example in the above picture.If we are in front of R we no longer have to check any walls because there are no front walls in Rs list.However if we are behind wall R we will have to descend into the tree and visit the nodes down the back list.Now imagine this simple example being converted to thousands of polygons and you can see that the frame rate would drop if we were behind wall R and speed up in front of wall R.The balanced tree above would be consistent because it would be equally deep at each leaf node.

There is a problem though because a more balanced splitter option may cause more splits in the list.What I suggest is this.We will loop through each polygon in the polygon list an score the wall with the following formula.The wall with the lowest score in the list wins and becomes the splitter for that node.Here's the formula:-

score=abs(frontfaces-backfaces)+(splits*8)

You can see that we subtract the front faces with the back faces and get the absolute value which will be higher the more un balanced the tree will be with this polygon and we then add on to that the number of splits that would take place and then multiply this by an amount you think gives more importance to splitting.I have used a value of eight here because I think the number of splits should be a much higher priority than the balance of the tree.

The actual function our compiler will call to choose the best splitter in a given list is quite easy to understand.We have already written 'ClassifyPoint' and 'ClassifyPolygon' which does the hard work for us.This loops through the list of polygons passed in and each time chooses a different splitter and tests it against the rest of the polygons in the list.Each time a splitter has been considered we end up with the number of splits and the number of front and back polygons in the respective lists.We then build the score using the above formula and if the score is lower than any previous then we remember this polygon.When the function ends it returns a pointer to the polygon in the list with the lowest score.Heres the code:-

```
          POLYGON * SelectBestSplitter(POLYGON *PolyList)
                              {
                   POLYGON* Splitter=PolyList;
                   POLYGON* CurrentPoly=NULL;
unsigned long BestScore=100000;// just set to a very high value to begin
                  POLYGON * SelectedPoly=NULL;
                     while (Splitter!=NULL)
                              {
                      CurrentPoly=PolyList;
          unsigned long score,splits,backfaces,frontfaces;
                score=splits=backfaces=frontfaces=0;
                    while (CurrentPoly!=NULL)
                              {
                    if (CurrentPoly!=Splitter)
                              {
          int result=ClassifyPoly(Splitter,CurrentPoly);
                         switch ( result)
                              {
                         case CP_ONPLANE:
                             break;
                          case CP_FRONT:
                           frontfaces++;
                             break;
                          case CP_BACK:
                            backfaces++;
                             break;
                        case CP_SPANNING:
                            splits++;
                             break;
default:
break;
}
}
CurrentPoly=CurrentPoly-> Next;
}// end while current poly
score=abs(frontfaces-backfaces)+(splits*8);
if (score< BestScore)
{
BestScore=score;
SelectedPoly=Splitter;
}
Splitter=Splitter-> Next;
}// end while splitter == null
return SelectedPoly;
}
```

The BuildBSPTree function

The time has come to build the actual BSP Compiler function now that we have written all the little helper functions (except SplitPolygon which we will write later) that the Compiler Needs.So here goes a couple of line at a time.

```
void BuildBspTree(NODE * CurrentNode,POLYGON * PolyList)
{
POLYGON *polyTest=NULL;
POLYGON *FrontList=NULL;
POLYGON *BackList=NULL;
POLYGON *NextPolygon=NULL;
POLYGON *FrontSplit=NULL;
POLYGON *BackSplit=NULL;
D3DVECTOR vec1,vec2;
CurrentNode-> Splitter=SelectBestSplitter(PolyList);
polyTest=PolyList;
```

As you can see the function is passed a list of polygons and an empty node.Remember that each node has a pointer to a polygon called Splitter.Aboce you can see that we fill this Nodes splitter with a pointer returned from the SelectBestSplitter function.This is the function we have just written so at this point the new Node now has a pointer to a splitter that has been selected rfom the polygon list passed into the function. We will se what these other tempory polygon pointers are for in a moment.The last thing we do above is assign a tempory pointer (polyTest) to PolyList.PolyList after all is the pointer to the polygon list so we do not want to manipulate it directly or we will never be able to get back to the root of the polygon list.

Next up we have to step through each Polygon in the the Polygon list (now pointed at by polyTest) and call 'ClassifyPoly' for each polygon to see if it is to the back,front or spanning the splitter(Polygon we have just selected and stored in the Node).If polyTest equalls NULL then we are at the end of the polygon list so we have to check for this.We also use the 'NextPolygon' pointer to point to the next polygon to be tested in the list.You will see in a moment that after a polygon has been tested as assigned to the relative group (front or back etc) that polygons 'Next' pointer is altered so that it points to the Next polygon in the front or back list and NOT the next polygon in the Global list passed into the function.Thats why we have to make a copy of it before it is altered so we can still loop through all the polygons in the global list.Also notice the check to make sure the the Current Polygon being tested against the splitter is not the Splitters itself.If this is the case then the polygon is ignored and skip over it performing no action.This makes sure that the splitter NEVER ends up in any of its own lists and it what makes the polygon lists finally empty because each time through this function we are loosing a polygon from the list (the splitter).

```
while (polyTest!=NULL)
{
NextPolygon=polyTest-> Next;// remember because polytest-> Next will be altered
if (polyTest!=CurrentNode-> Splitter)
{
switch (ClassifyPoly(CurrentNode-> Splitter,polyTest))
{

case CP_FRONT:
polyTest-> Next=FrontList;
FrontList=polyTest;
break;

case CP_BACK:
polyTest-> Next=BackList;
BackList=polyTest;
break;
```

Nothing special with the first two cases.If the polygon is behind or front of the splitter it is added to the appropriate lists.Just in case you are confused by the switching around I will explain it.In the case of CP_FRONT above, the Current Polygon being tested has its next pointer set to point at the 'FrontList' pointer.This will be NULL the first time a polygon is assigned to the front list.Then the FrontList pointer is altered to point at the newly assigned polygon.In other words the new polygon is added to the top of the list and the polygons 'Next' pointer is set to point at whatever FrontList pointed at before so we still have a linked list.Notice that we have changed the 'Next' Pointer of the polygon which we would normally use to look at the next polygon in the global polygon list.We cannot do this now because it only points to the back list.This is why we made a copy of this pointer above so that we can still step through and access the rest of the polygons.

Lets have a look at what happens if the ClassifyPoly function returns CP_SPANNING.This section of code also ties up all the while loops we just started.

```
case CP_SPANNING:
FrontSplit=new POLYGON;
BackSplit=new POLYGON;
ZeroMemory(FrontSplit,sizeof(POLYGON));
ZeroMemory(BackSplit,sizeof(POLYGON));
SplitPolygon(polyTest,CurrentNode-> Splitter,FrontSplit,BackSplit);
delete polyTest;
FrontSplit-> Next=FrontList;
FrontList=FrontSplit;
BackSplit-> Next=BackList;
BackList=BackSplit;
break;
default:
break;
}
}// end if polypoint!=CurrentNodesplitter
polyTest=NextPolygon;
}// end while loop
```

If the current polygon being tested does straddle the splitter then we create two new polygons and zero the memory.We then send these pointers along with the polygons that needs to be split and the Splitter it needs to be split against to the 'SplitPolygon' function which will when returns will have split the polygon into two halves and they will be pointed to by the FrontSplit and BackSplit pointers.Interestingly enough we no longer need the orginal polygon so we free up the memory.This may sound strange at first but remember a polygon isn't in the tree until it is used as a splitter and I may have been split many times by different splitters at this point.If a polygon is split and then those to splits are split the first two splits not only are not used by the tree anymore but we also loose pointers to them in memory so we must free them up here while a valid temporary pointer to the unsplit polygon still exists.The FrontSplit and BackSplit pointers which now point to the two newly created splits are added to the front list and back list respectively.After this polyTest is then reloaded with the orginal 'Next' pointers value which of course is the next polygon in the input list.We do this loop for every polygon in the input list (ignoring the splitter itself of course so this does not get added to any list).

At this point we have divided all the polygons in the list to either front or back of the Nodes Splitter.If there are NO polygons in the front list then a new Leaf NOde is created and attached to the current nodes Front Pointer.This leaf Node is set to be Empty (space you CAN walk about in).If there are NO walls in the Back List then once again a new node is created and attached to the current nodes Back pointer.Because this node is Behind the current Node it is set to be a solid leaf.
If there are polygons in the front list a new Node is created and and attched to the Current Nodes Front pointer.This function then calls itself sending in the new node and the current node and the Front list as the polygon list.Exactly the same happen if there are polygons in the Back list except the newly created node is attached to the current nodes Back pointer instead of the front obviously.Heres the last bit of the function.

```
if (FrontList==NULL)
{
NODE *leafnode=new NODE;
ZeroMemory(leafnode,sizeof(leafnode));
leafnode-> IsLeaf=true;
leafnode-> IsSolid=false;
CurrentNode-> Front=leafnode;
}
else
{
NODE * newnode=new NODE;
ZeroMemory(newnode,sizeof(newnode));
newnode-> IsLeaf=false;
CurrentNode-> Front=newnode;
BuildBspTree(newnode,FrontList);
}
if (BackList==NULL)
{
NODE *leafnode=new NODE;
ZeroMemory(leafnode,sizeof(leafnode));
leafnode-> IsLeaf=true;
leafnode-> IsSolid=true;
CurrentNode-> Back=leafnode;;
}
else
{
NODE * newnode=new NODE;
ZeroMemory(newnode,sizeof(newnode));
newnode-> IsLeaf=false;
CurrentNode-> Back=newnode;
BuildBspTree(newnode,BackList);
}
}// end function
```

And that is the compiler totally built.For completeness here is the BSP Compiler function in its entirety:-

```
void BuildBspTree(NODE * CurrentNode,POLYGON * PolyList)
{
POLYGON *polyTest=NULL;
POLYGON *FrontList=NULL;
POLYGON *BackList=NULL;
POLYGON *NextPolygon=NULL;
POLYGON *FrontSplit=NULL;
POLYGON *BackSplit=NULL;
D3DVECTOR vec1,vec2;
CurrentNode-> Splitter=SelectBestSplitter(PolyList);
polyTest=PolyList;

while (polyTest!=NULL)
{
NextPolygon=polyTest-> Next;// remember because polytest-> Next will be altered
if (polyTest!=CurrentNode-> Splitter)
{
switch (ClassifyPoly(CurrentNode-> Splitter,polyTest))
{

case CP_FRONT:
polyTest-> Next=FrontList;
FrontList=polyTest;
break;

case CP_BACK:
polyTest-> Next=BackList;
BackList=polyTest;
break;

case CP_SPANNING:
FrontSplit=new POLYGON;
BackSplit=new POLYGON;
ZeroMemory(FrontSplit,sizeof(POLYGON));
ZeroMemory(BackSplit,sizeof(POLYGON));
SplitPolygon(polyTest,CurrentNode-> Splitter,FrontSplit,BackSplit);
delete polyTest;
FrontSplit-> Next=FrontList;
FrontList=FrontSplit;
BackSplit-> Next=BackList;
BackList=BackSplit;
break;
default:
break;
}
}// end if polypoint!=CurrentNodesplitter
polyTest=NextPolygon;
}// end while loop

if (FrontList==NULL)
{
NODE *leafnode=new NODE;
ZeroMemory(leafnode,sizeof(leafnode));
leafnode-> IsLeaf=true;
leafnode-> IsSolid=false;
CurrentNode-> Front=leafnode;
}
else
{
NODE * newnode=new NODE;
ZeroMemory(newnode,sizeof(newnode));
newnode-> IsLeaf=false;
CurrentNode-> Front=newnode;
BuildBspTree(newnode,FrontList);
}
if (BackList==NULL)
{
NODE *leafnode=new NODE;
ZeroMemory(leafnode,sizeof(leafnode));
leafnode-> IsLeaf=true;
leafnode-> IsSolid=true;
CurrentNode-> Back=leafnode;;
}
else
{
NODE * newnode=new NODE;
ZeroMemory(newnode,sizeof(newnode));
newnode-> IsLeaf=false;
CurrentNode-> Back=newnode;
BuildBspTree(newnode,BackList);
}
}// end function
```

Not very big is it when you consider what it is doing?The recursive nature of this type of function can confuse some people but just remember that the functions calls itself at each node for both the front and back lists.Look through it and make sure you really now whats going on.Remember we only call this function once from our InitPolygons routine passing in the Complete list of polygons for the game world and a newly created Root Node. When the function returns the Root Node contains a valid pointer to the entire BSP Tree.

Now of course there is a call to 'SplitPolygon' in the above function which we have not yet addressed.Lets address it now and get it out of the way and then we will look at some more exciting stuff like rendering the tree and performing Line of Sight/Collision detection with the BSP Tree.

Splitting a Polygon with a Plane

In our Dot Product tutorial (which I hope you have either read or fully understand) we wrote a function to intersect a Ray with a Plane.This function will be needed by our SplitPolygon routine.This is because to intersect a Polygon with a plane we simply treat each edge of the polygon as a seperate ray.Although I am not going to explain the 'GetIntersect' routine here because it is fully explained in our Dot Product tutorial I will show you the code for the function for completeness.The following function is slightly different from the one in our DotProduct tutorial as it takes an extra parameter.The function accepts six parameters.The start and End of the Ray in world coordinates and a Point on the plane that needs to be tested against the ray and the plane normal.The last two parameters are filled by the function on return.If the function returns true then an intersection between the start point and the end point has occured with the plane.If this is the case on return then 'intersection' will point to a vector containing the point in world coordinates that the ray hit the plane and 'percentage' is the new parameter we have added since the Dot Product tutorial.This will point to a float that represents how far down the ray from start point the intersection has happened as a percentage of the overall ray length.This value is between 0 (0%=line start) and 1(100%=line end).This reason we need this percentage returned is because when we carve our polygon up we will need to carve up and add new texture coordinates as well.We will see how this value is used then.

```
bool Get_Intersect (D3DVECTOR *linestart,D3DVECTOR *lineend,D3DVECTOR *vertex,D3DVECTOR *normal,D3DVECTOR * intersection, float *percentage)
{
D3DVECTOR direction,L1;
float linelength,dist_from_plane;

direction.x=lineend->x-linestart->x;
direction.y=lineend->y-linestart->y;
direction.z=lineend->z-linestart->z;

linelength=DotProduct(direction,*normal);

if (fabsf(linelength)<0.0001)
{
return false;
}

L1.x=vertex->x-linestart->x;
L1.y=vertex->y-linestart->y;
L1.z=vertex->z-linestart->z;

dist_from_plane=DotProduct(L1,*normal);
*percentage=dist_from_plane/linelength;

if (*percentage<0.0f)
{
return false;
}
else
if (*percentage>1.0f)
{
return false;
}

intersection->x=linestart->x+direction.x*(*percentage);
intersection->y=linestart->y+direction.y*(*percentage);
intersection->z=linestart->z+direction.z*(*percentage);
return true;
}
```

Like I said earlier, if the above code looks a little strange to you our Dot Product tutorial will take you through it Line by Line.

Now lets write that SplitPolygon routine.You might imagine it would be difficult but with our GetIntersect function above we simply have to step through each edge of the test polygon and treat it as a ray and call GetIntersect above to retrieve the intersection point.Remember that we are testing a polygon against the Splitter at each node so the when we cal GetIntersect above we pas in the two vertices that make up the current edge of the polygon, then we need to pass the GetIntersect routine some information about the Plane.This ofcourse is the plane that our Splitter polygon lays on so we simply pass in any vertex in the Splitter polygon (vertex[0] will do?) and the Normal of the splitter.We have all this information stored in the splitter so thats no problem.

Lets now look at the 'SplitPolygon' routine a couple of lines at a time.

```
void SplitPolygon(POLYGON *Poly,POLYGON *Plane,POLYGON *FrontSplit,POLYGON *BackSplit)
{
```

Just to refresh your memory, this function is called from our BuildBSPTree function when a polygon has been tested against the current splitter and is found to be spanning it.Parameter 1 (Poly) is the polygon that needs to be split into two and parameter 2(plane) is the Splitter polygon that needs to divide the polygon in two.The last two parameters are empty at the moment and will be filled in by this function and will return two new polygons that are the result of the split.

First we set up some local variables that will be used throughout the function.FrontList and BackList are arrays of vertices to contains the vertices for the newly created front and back polygon splits.As each point is classified as point to the front or the back of the plane it will be added to these lists.

```
D3DLVERTEX FrontList[20],BackList[20],FirstVertex;
D3DVECTOR PlaneNormal,IntersectPoint,PointOnPlane,PointA,PointB;
WORD FrontCounter=0,BackCounter=0,loop=0,CurrentVertex=0;
float percent;

// Find any vertex on the plane to use later in plane intersect routine
PointOnPlane=*(D3DVECTOR *)&Plane->VertexList[0];
```

The first thing we do above is store the first vertex of the splitter polygon in the PointOnPlane vector.The line may look a bit weird with all its casting etc but remember that the first part of a D3DLVERTEX structure is just a vector so can be cast as such.We will use this PointOnPlane variable later when we call the GetIntersect routine.

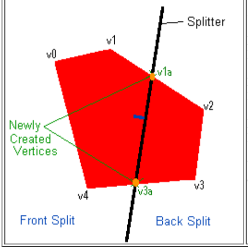
The first thing we must do is take a look at the First Vertex in the polygon (not the Splitter polygon, the one we are trying to split) and find which side of the Splitter this point is on and add that vertex to the respective vertex array (Frontlist or Backlist).This means that if a point is in front of the Splitter then that vertex will be in the Front polygon created.Vice versa for the Back list.If the vertex is on the plane (sharing the plane with Splitter) then the same vertex is added to BOTH lists because it will be needed in both polygons:-

```
// first we find out if the first vertex belongs in front or back list
FirstVertex=Poly->VertexList[0];
switch (ClassifyPoint( (D3DVECTOR *)&FirstVertex,Plane))
{
case CP_FRONT:
FrontList[FrontCounter++]=FirstVertex;
break;
case CP_BACK:
BackList[BackCounter++]=FirstVertex;
break;
case CP_ONPLANE:
BackList[BackCounter++]=FirstVertex;
FrontList[FrontCounter++]=FirstVertex;
break;
default:
break;
}
```

As you have probably gathered the whole purpose of this function is just to classify edges of the polygon to the splitter and assigned them to the relative lists.Once we have done this we we have two lists which we can then construct the polygons out of.

Now that we have the first vertex in the correct list(or lists if on plane) we now have to loop through the rest of the vertex.We start though at the second vertex in the polygon and compare it to the previous vertex.If they are on opposing sides of the splitter then an intersection with the plane has occurred and a new vertex needs to be created and added to both lists.Lists have a look at a picture and we will have a look at the splitting technique in more detail.

Splitting a Polygon with a Plane



We have already assigned V0 to the front list in our example with the preceeding code.Now we loop through vertices 1 though 0(again) and each edge being tested is the Current Vertex with the previous.So for example we test point 1 & 0 then point 2 & 1 etc.
This is how it works using the example opposite.We test v1 and v0 and discover they are both on the front side of the splitter so v1 gets added to the Front list (which already contains v0) and so all is done.Next we compare v2 with v1 and discover they are on opposing sides of the splitter so we call GetIntersect which will return the position of the new vertex on the split line (v1a opposite).This intersection is added to both lists and v2 is added to the back list.

Front List so far contains : v0,v1,v1a
Back List so far contains : v1a,v2.

Now next time through the loop we test v3 with v2 and they are both behind the plane so v3 gets added to the back list.We then loop round to compare v4 with v3 and find another intersection.The intersection (v3a) is added to the back list and the front list and v4 is added to the front list also.

Front List so far contains : v0,v1,v1a,v3a,v4
Back List so far contains : v1a,v2,v3,v3a

We then loop through again but roll back over to test zero and the previous vertex (v4) there is no split they are both on the same side of the plane but because this is vertex zero which is already in the list it does not get added to any list and we are done.However had there been a split between v4 & v0 (lets call it v4a) then v4a would have been added to both front and back lists.

So we are now going to loop through from vertices one (not zero because its already in the list) through zero putting in a bit of logic to make sure that we roll back over to zero after the last vertex.

```
for (loop=1;loop<Poly->NumberOfVertices+1;loop++)
{
if (loop==Poly->NumberOfVertices)
{
CurrentVertex=0;
}
else
{
CurrentVertex=loop;
}
PointA=*(D3DVECTOR *)&Poly->VertexList[loop-1];
PointB=*(D3DVECTOR *)&Poly->VertexList[CurrentVertex];
```

As you can see we loop through the vertices of the polygon that needs to be split.The current vertex to be worked on is equal to the current value of the loop unless it we have rolled over the last vertex which means we set the current vertex to zero. At this point we now know the two vertices that make up the edge we are going to test.PointA a takes the previous vertex in the polygon (which would have already been assigned to a list) and casts it to a D3DVECTOR so that we can use it with all our functions and PointB casts the current vertex (not in any list yet) to a D3DVECTOR.In other words, the first time through the loop PointA will hold vertex zero and PointB will hold vertex one.This ofcourse makes up the first edge.

Next up we classify PointB with the plane to get which side of the plane it is on.If it is actually ON the plane then this vertex is added to both lists. `PlaneNormal=Plane->Normal;`

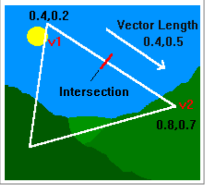
```
int Result=ClassifyPoint(&PointB,Plane);
if (Result==CP_ONPLANE)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
else
{
```

If PointB is not on the plane then we test for an intersection with the plane.We call GetIntersect passing in PointA,PointB and the Ray start and end and also pass in the Point on the plane we got earlier (first vertex in the splitter) and the Plane Normal(splitters Normal).If this function returns 'true' IntersecPoint will hold the X,Y,Z of the new vertex created by the split that is on the plane (v1a or v3a in the above example).We can create a new vertex out of this vector.Also if the function returns 'true' the float 'percentage' will hold how far down the ray from the start (as a percentage between 0 and 1) the intersection occurred.We will use this for generating a NEW texture coordinate for the NEW vertex.

```
if (Get_Intersect(&PointA,&PointB,&PointOnPlane,&PlaneNormal,&IntersectPoint, &percent)==true)
{
```

If we pass the test and a collision has occurred we already have the new vertex position in 'IntersectPoint'.When I First wrote my compiler the level was not texture mapped.I simply created a new vertex out of the IntersectPoint returned and fely very happy with my self until I tried to to texture the level and all the textures screwed up.I forgot that the new vertex is going to need a new Texture Coordinate.This was quite easy to overcome (after a bit of thought) which is why the 'GetIntersect' function was modified to also return a percentage.Lets see how it works.(I assume you are familiar with the Normalized Texture Coordinates that D3D uses).

Interpolating the New Texture Coordinate



The picture opposite shows how a triangle polygon may be mapped to a texture.You can see the texture coordinates for v1 and v2.The red line shows where an intersection with the plane has occurred with this polygon and infact shows the point that a new texture coordinate needs to be created for.First of all we subtract the first vertex texture coordintes from the second and we end up with the Vector Length of the line between v1 and v2 on the texture.You can see opposite that <0.8,0.7>-<0.4,0.2>=<0.4,0.5> which is the direction and the length between texture coordinate 1 & 2.Now the great thing is that our GetIntersect function returned a percentage from the start of the line that the intersect occurred.We can reuse this value for texture coordinates as well.For example imagine that the percentage returned by GetIntersect is 0.5 meaning that the plane intersect the polygon exactly half way between vertex 1 & vertex 2.If we multiply the Vector opposite (vector length) by the percentage (0.5 in this example) we get a vector of

<0.4*0.5 , 0.5 * 0.5>.=<0.2 , 0.25>

Now just add this vector so it is relative to the start point (the start of the ray or in our case the first vertices texture coordinates)

New Texture Coordinates=<0.4 , 0.2> + <0.2 , 0.25>=<0.6 , 0.45>

This same interpolation could also be used for calculating a Vertex light value as well.

Here's the code that calculates the new texture coordinates and creates a new vertex.

```
float deltax,deltay,texx,texy;
deltax=Poly->VertexList[CurrentVertex].tu-Poly->VertexList[loop-1].tu;
deltay=Poly->VertexList[CurrentVertex].tv-Poly->VertexList[loop-1].tv;
texx=Poly->VertexList[loop-1].tu+(deltax*percent);
texy=Poly->VertexList[loop-1].tv+(deltay*percent);
D3DLVERTEX copy=D3DLVERTEX(IntersectPoint,RGB_MAKE(255,255,255),0,texx,texy);
```

Now we have the new Vertex and we know we have to add it to both front and back polygon list (because it belongs in both the front split and the back split polygons) but we also have to put the Current Vertex into the correct list also.If PointB is in front of the splitter then we add the New Vertex to Both lists and then add PointB to the Front List AS LONG AS the current vertex is NOT vertex zero because this one would already be in the list.The same is true if PointB is behind the splitter but the other way around obviously.

```
if (Result==CP_FRONT )
{
BackList[BackCounter++]=copy;
FrontList[FrontCounter++]=copy;
if (CurrentVertex!=0)
{
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
}

if (Result==CP_BACK)
{
FrontList[FrontCounter++]=copy;
BackList[BackCounter++]=copy;
if (CurrentVertex!=0)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
}
}

}// end if intersection (get intersect==true)

else
```

Ok then thats the code taken care of for if an edge intersects the splitter , but if this is not the case then the Current Vertex and the Last previous vertex are both to one side if the plane.If this is the case we simply stick the Current Vertex into the appropriate lists.Once again though only if the Current Vertex is NOT vertex zero because this was put in the list at the start of the function

```
. {
if (Result==CP_FRONT)
{
if (CurrentVertex!=0)
{
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
}

if (Result==CP_BACK)
{
if (CurrentVertex!=0)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
}
}
}

}
}//end loop through each edge
```

At this point in the function we all the loops have ended and we have two lists of vertices.One for the front split polygon and one for the back split polygon.The remainder of the function simply builds the polygons using exactly the same technique as we used earlier in our AddPolygon routine.It copies the Vertex Lists in to the Front and Back split polygon structures, Calculates the indices (because remember even if a triangle is split with a plane one of the splits will now have four vertices) and cuilds the indices for each polygon.Then as a last step it generates the polygon Normals for the two polygons.Heres the rest of the function:-

```
//OK THEN LETS BUILD THESE TWO POLYGONAL BAD BOYS

FrontSplit->NumberOfVertices=0;
BackSplit->NumberOfVertices=0;

for (loop=0;loop<FrontCounter;loop++)
{
FrontSplit->NumberOfVertices++;
FrontSplit->VertexList[loop]=FrontList[loop];
}

for (loop=0;loop<BackCounter;loop++)
{
BackSplit->NumberOfVertices++;
BackSplit->VertexList[loop]=BackList[loop];
}

BackSplit->NumberOfIndices=(BackSplit->NumberOfVertices-2)*3;
FrontSplit->NumberOfIndices=(FrontSplit->NumberOfVertices-2)*3;

// Fill in the Indices Array

WORD v0,v1,v2;
for (loop=0;loop<FrontSplit->NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
FrontSplit->Indices[loop*3]=v0;
FrontSplit->Indices[(loop*3)+1]=v1;
FrontSplit->Indices[(loop*3)+2]=v2;
}

for (loop=0;loop<BackSplit->NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
BackSplit->Indices[loop*3]=v0;
BackSplit->Indices[(loop*3)+1]=v1;
BackSplit->Indices[(loop*3)+2]=v2;
}

// calculate polygon Normals

D3DVECTOR edge1,edge2;
edge1=*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[1]]-*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[0]];

edge2=*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[FrontSplit->NumberOfIndices-1]]-*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[0]];

FrontSplit->Normal=CrossProduct(edge1,edge2);
FrontSplit->Normal=Normalize(FrontSplit->Normal);

edge1=*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[1]]-*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[0]];

edge2=*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[BackSplit->NumberOfIndices-1]]-*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[0]];

BackSplit->Normal=CrossProduct(edge1,edge2);
BackSplit->Normal=Normalize(BackSplit->Normal);
}
```

Phew, that function was larger than the actual BSP Compiler function.But it is the last function needed to make our BSP Compiler function work.Once again then , here is the Split Polygon Function in its entirety.

```
void SplitPolygon(POLYGON *Poly,POLYGON *Plane,POLYGON *FrontSplit,POLYGON *BackSplit)
{
D3DLVERTEX FrontList[20],BackList[20],FirstVertex;
D3DVECTOR PlaneNormal,IntersectPoint,PointOnPlane,PointA,PointB;
WORD FrontCounter=0,BackCounter=0,loop=0,CurrentVertex=0;
float percent;

// Find any vertex on the plane to use later in plane intersect routine
PointOnPlane=*(D3DVECTOR *)&Plane->VertexList[0];

// first we find out if the first vertex belongs in front or back list
FirstVertex=Poly->VertexList[0];
switch (ClassifyPoint( (D3DVECTOR *)&FirstVertex,Plane))
{
case CP_FRONT:
FrontList[FrontCounter++]=FirstVertex;
break;
case CP_BACK:
BackList[BackCounter++]=FirstVertex;
break;
case CP_ONPLANE:
BackList[BackCounter++]=FirstVertex;
FrontList[FrontCounter++]=FirstVertex;
break;
default:
break;
}

for (loop=1;loop<Poly->NumberOfVertices+1;loop++)
{
if (loop==Poly->NumberOfVertices)
{
CurrentVertex=0;
}
else
{
CurrentVertex=loop;
}
PointA=*(D3DVECTOR *)&Poly->VertexList[loop-1];
PointB=*(D3DVECTOR *)&Poly->VertexList[CurrentVertex];

PlaneNormal=Plane->Normal;
int Result=ClassifyPoint(&PointB,Plane);
if (Result==CP_ONPLANE)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
else
{

if (Get_Intersect(&PointA,&PointB,&PointOnPlane,&PlaneNormal,&IntersectPoint, &percent)==true)
{

float deltax,deltay,texx,texy;
deltax=Poly->VertexList[CurrentVertex].tu-Poly->VertexList[loop-1].tu;
deltay=Poly->VertexList[CurrentVertex].tv-Poly->VertexList[loop-1].tv;
texx=Poly->VertexList[loop-1].tu+(deltax*percent);
texy=Poly->VertexList[loop-1].tv+(deltay*percent);
D3DLVERTEX copy=D3DLVERTEX(IntersectPoint,RGB_MAKE(255,255,255),0,texx,texy);

if (Result==CP_FRONT )
{
BackList[BackCounter++]=copy;
FrontList[FrontCounter++]=copy;
if (CurrentVertex!=0)
{
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
}

if (Result==CP_BACK)
{
FrontList[FrontCounter++]=copy;
BackList[BackCounter++]=copy;
if (CurrentVertex!=0)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
}
}

}// end if intersection (get intersect==true)

else
{
if (Result==CP_FRONT)
{
if (CurrentVertex!=0)
{
FrontList[FrontCounter++]=Poly->VertexList[CurrentVertex];
}
}

if (Result==CP_BACK)
{
if (CurrentVertex!=0)
{
BackList[BackCounter++]=Poly->VertexList[CurrentVertex];
}
}
}

}
}//end loop through each edge


//OK THEN LETS BUILD THESE TWO POLYGONAL BAD BOYS

FrontSplit->NumberOfVertices=0;
BackSplit->NumberOfVertices=0;

for (loop=0;loop<FrontCounter;loop++)
{
FrontSplit->NumberOfVertices++;
FrontSplit->VertexList[loop]=FrontList[loop];
}

for (loop=0;loop<BackCounter;loop++)
{
BackSplit->NumberOfVertices++;
BackSplit->VertexList[loop]=BackList[loop];
}

BackSplit->NumberOfIndices=(BackSplit->NumberOfVertices-2)*3;
FrontSplit->NumberOfIndices=(FrontSplit->NumberOfVertices-2)*3;

// Fill in the Indices Array

WORD v0,v1,v2;
for (loop=0;loop<FrontSplit->NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
FrontSplit->Indices[loop*3]=v0;
FrontSplit->Indices[(loop*3)+1]=v1;
FrontSplit->Indices[(loop*3)+2]=v2;
}

for (loop=0;loop<BackSplit->NumberOfIndices/3;loop++)
{
if (loop==0)
{
v0=0;
v1=1;
v2=2;
}
else
{
v1=v2;
v2++;
}
BackSplit->Indices[loop*3]=v0;
BackSplit->Indices[(loop*3)+1]=v1;
BackSplit->Indices[(loop*3)+2]=v2;
}

// calculate polygon Normals

D3DVECTOR edge1,edge2;
edge1=*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[1]]-*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[0]];

edge2=*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[FrontSplit->NumberOfIndices-1]]-*(D3DVECTOR *)&FrontSplit->VertexList[FrontSplit->Indices[0]];

FrontSplit->Normal=CrossProduct(edge1,edge2);
FrontSplit->Normal=Normalize(FrontSplit->Normal);

edge1=*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[1]]-*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[0]];

edge2=*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[BackSplit->NumberOfIndices-1]]-*(D3DVECTOR *)&BackSplit->VertexList[BackSplit->Indices[0]];

BackSplit->Normal=CrossProduct(edge1,edge2);
BackSplit->Normal=Normalize(BackSplit->Normal);
}
```

If you have never done any sort of polygon clipping before then that was probably pretty hard going for you but the same function can be reused in other applications.For example the same function could be used to clip polygons to the plane of the View Fustrum although you would want to optimize it to make it faster .However all our splitting is done at the start of the application (and would normally be done in development) so it's not a problem.

If you have stayed with me up to here then first of all give youself a pat on the back.We now have created all the functions to build a BSP tree.All we have to do is call BuildBSPTree and pass in an empty root node and a list of polygon and whent he function returns all the polygons will be split and assigned to the appropriate nodes in the tree.You are now ready to start using that BSPTree to render your level and perform Line of Sight/Collision determination.

We have a BSP Tree but How do we Render It

Rendering is easy.We will now build a function that will be passed the Root node to the tree and will render the entire level in Back to Front order.Note that you can also render in Front to Back order if a Z-Buffer is being used (or some other VSD method).Using a Z-Buffer and drawing front to back will reduce overdraw and speed up drawing but will not cater for Translucent objects which must still be rendered back to front.There for our function will render back to front so a Z-Buffer is not needed and Translucent objects will display correctly (windows,glass, etc).The function is so small that changing it from a Back to Font renderer to a Front to Back renderer should be a piece of cake for you.Here is the function that will be called every frame to render the entire BSP Tree.It is called WalkBspTree and I just know how impressed you are going to be at how small this recursive function is.

```
void WalkBspTree(NODE *Node,D3DVECTOR *pos)
{
if (Node->IsLeaf==true) return;
int result=ClassifyPoint(pos,Node->Splitter);
if (result==CP_FRONT)
{
if (Node->Back!=NULL) WalkBspTree(Node->Back,pos);
lpDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,D3DFVF_LVERTEX,&Node->Splitter->VertexList[0],Node->Splitter->NumberOfVertices,&Node->Splitter->Indices[0],Node->Splitter->NumberOfIndices,NULL);
if (Node->Front!=NULL) WalkBspTree(Node->Front,pos);
return ;
} // this happens if we are at back or on plane
if (Node->Front!=NULL) WalkBspTree(Node->Front,pos);
if (Node->Back!=NULL) WalkBspTree(Node->Back,pos);
return;
}
```

Thats it!!. I hope by now this makes sense to you and you can see what is happening.We pass in to the function the Root Node to the tree and Camera Position.If the current Node is a leaf it has no polygons so returns.If we are in front of the current node we draw all the walls behind the node , then draw the node itself and then draw the nodes in front of the current node.This is what makes it a Back to Front renderer.Notice that if we are not infront of the node then we are behind it (or maybe on it) and we reverse the drawing order.

Believe it or not to make this a Front to Back renderer (instead of a Back to Front) all you have to do is change two lines.

if (result==CP_FRONT) should be changed to
if (result==CP_BACK).

and the line that actually draws the polygon (DrawIndexedPrimitve) should be moved from its current location down the listing so it is sandwiched between the 'if (Node->Front!=NULL)' and 'if(Node->Back!=NULL)' so it is situated two lines up from the last return command.

Please remember though that if you do decide to change the code to make it a Front to Back renderer you will need a Z-Buffer (or some other form of VSD method) to correctly render the world.Otherwise walls that are further away would be draw over walls that are nearer to the camera.

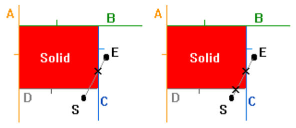
Line of Sight and Collision Detection

Well I did mention that I was going to show you how to do some cheap collision detection with a Solid Bsp Tree and although I was going to leave this subject until the Second Tutorial in the series I have since changed my mind and have decided to discuss it now.

Because the space in our game world has now been divided up into ares that are either Solid or Empty it becomes very simple to determine if a given position in the world can be stood in or is in fact a solid wall.We are going to develop a function called 'LineOfSight' that will take a Start point and a End point and return true if there is NO obstruction in between them.This can be used for example to determine weather a monster in your game world can SEE the player and start to chase him and even better is that it can also be used for collision detection.

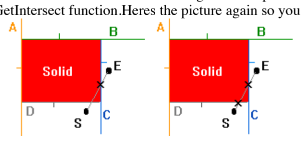
In order to use the function for collision detection you just pass in the Current Camera (or monster) position and the position you are about to move into and the function will return 'TRUE' is the path is clear. Now I bet you are thinking that this is going to be some massive function but in fact its very very small thanks to its (you guessed it) Recursive Nature.

The problem with Recursion is that it is difficult to Draw Pictures of whats going on but I am going to have to explain this function to you somehow so I will try Diagram approach again.The picture of the world below contains 4 walls facing outward that bound a solid area.The wall as labelled in the order they are linked in the tree with the orange line A being the Root Node.Remember if a Node has no Remaining walls either behind it or infront of it then it points to leafs that are either Solid or Empty respectively.You can see that the RED area below is behind all walls so is Solid space.The two point (S & E) below describe two points in 3D space (start and end).The current camera position could be described as S for example and the position the camera is about to move in to could be described as E.This is how it works.



First we call our function with the Root node and pass in the start and End points we wish to check.If both points are in front of the Node then the function calls itself passing in the two points(S & E) and the Nodes front pointer.We also send them down the Front node if both points lay on the plane of the Node.If both points are behind the plane then we do the same but send both points down the Back Node instead.This is all standared the sort of standard traversal we have been doing throughout this tutorial.The function calls itself recursively and if both points pop out in a leaf and the leaf is empty then the function returns True indicating that there is Line of sight.If both points pop out in a solid leaf then the function returns false.
The real majic happens however if the the start and end points are on opposing sides of the Node being tested.Lets step through the above diagram in our heads.

First we check Node A and both points are behind it so the function calls itself passing in Node A's 'Back' pointer (which points to Node B) as the Node parameter.This time we check against Node B and again both points are behind Node B.So the function calls itself again this time passing in Node B's 'Back' pointer (which points to Node C) as the Node Parameter.This time though the two points are on opposing sides of Node C so we computer the intersection between Start & End with Node C (the X in the first picture) using our GetIntersect function.Heres the picture again so you can look at it while you are reading.



The function now calls itself twice , once for the Back split (S to X) and Once for the Front split (X to E).If both these functions return true then this function returns true (and there is line of sight.).In other words in the above example the function would call itself passing in (Start & X) as the start and end points and would send the current Nodes Back pointer as the Node to be tested which would be C.At the Same time we would send End & X down the Front Pointer but would pop out at and empty leaf because there are no Front Nodes infront of C so this instance of the function would return true because there are no obstructions bewteen x and E.
Remember though that we sent S & X down Node C's back pointer which means these two point get tested against Node D. There was an intersection against S and X so a new intersection is generated on Node D.Now the function checks both (S & bottomX) and (X & topmostX).S & bottomMost X are sent down Node D's front pointer.There are no walls in front of Node D so we pop out in an empty leaf so true is returned meaning there are no obstructions between S and bottomX.We the send both bottomX and topmostX down Node D's Back pointer.There are No walls here so both these points are in a Solid leaf meaning Line of Sight is obstructed.Because this instance of the function returns false all the other one do to because both segments of the ray have to return true for the function to return true.

If that has confused you a little bit perhaps looking at the code may help.It is very small and I like it a lot because every time both points are in one side of a Node every single Node on the other side of that node is eliminated from the testing process also. Dont you just love Hierarchies.

```
bool LineOfSight (D3DVECTOR *Start,D3DVECTOR *End, NODE *Node)
{
float temp;
D3DVECTOR intersection;
if (Node->IsLeaf==true)
{
return !Node->IsSolid;
}

int PointA=ClassifyPoint(Start,Node->Splitter);
int PointB=ClassifyPoint(End,Node->Splitter);

if (PointA==CP_ONPLANE && PointB==CP_ONPLANE)
{
return LineOfSight(Start,End,Node->Front);
}

if (PointA==CP_FRONT && PointB==CP_BACK)
{
Get_Intersect (Start,End,(D3DVECTOR *) &Node->Splitter->VertexList[0],&Node->Splitter->Normal,&intersection,&temp);
return LineOfSight(Start,&intersection,Node->Front) && LineOfSight(End,&intersection,Node->Back) ;
}

if (PointA==CP_BACK && PointB==CP_FRONT)
{
Get_Intersect (Start,End,(D3DVECTOR *) &Node->Splitter->VertexList[0],&Node->Splitter->Normal,&intersection,&temp);
return LineOfSight(End,&intersection,Node->Front) && LineOfSight(Start,&intersection,Node->Back) ;
}


// if we get here one of the points is on the plane
if (PointA==CP_FRONT || PointB==CP_FRONT)
{
return LineOfSight(Start,End,Node->Front);
}
else
{
return LineOfSight(Start,End,Node->Back);
}
return true;
}
```

With a little help from the GetIntersect function the above code sorts out the whole collision detection thing in just a couple of lines of code by just recursively calling itself until every section of the reay ends up in a leaf (either solid or empty).It's quick too.

End of Part I

Well this brings us to the End of the first in a series (hopefully) of tutorials using BSP Trees.We have learned how to Compile Solid Node Based BSP Tree and Render it either in Back to Front or Front to Back order.As a the final iceing on the cake we have written a Line of Sight routine that will allow us to walk our camera around the scene and accurately collide with the walls,floors and ceilings of our world. This Tutorial has taken me So Long to write I hope it has been some use to you.It has taken me about four weeks to finish this tutorial because up until then I had never implemented a BSP Tree myself.

Part II Coming shortly

I have already started work on the second part of this tutorial.We will look at making a LEAFY tree to improve tree traversal and storage and we will also take a look at how to use Axis Aligned Bounding Boxes to peform very quick Fustrum Rejection of entire sections of the tree speeding up rendering by a very large amount.We will also take a look at saving a BSP Tree to a disk file.

If you have any questions please either post them on the message board or email me at gary@mr-gamemaker.com.

BSP Compiler Demo and Source Code

The following Link will allow you to download the BspCompiler demo and the source code of the Demo containing all the functions we have written throughout this tutorial.The demo allows you walk around a Game Level of castle walls using the keyboard.

Download BspDemo and Source Code

Home Page

Thanks For Visiting Mr-GameMaker.com