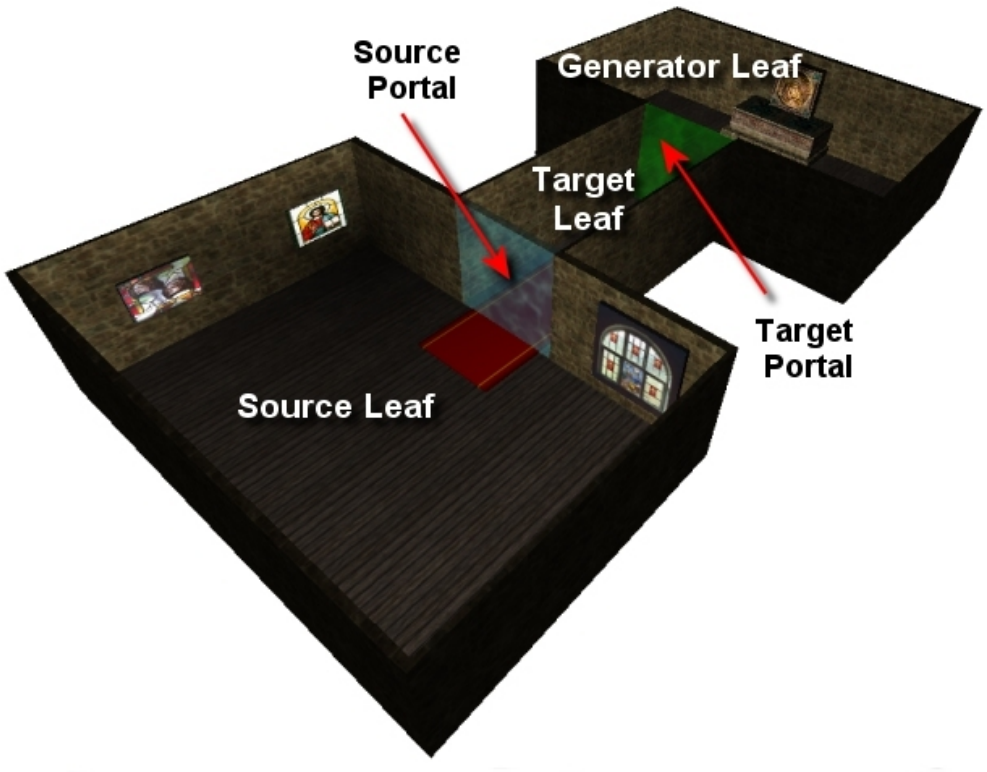
# Workbook Seventeen:
# Potential Visibility Sets



# Introduction

In this lesson we will continue to add to the lab project implemented in the previous lesson (lab projects 16.2 and 16.3) to facilitate the calculation and rendering of geometry using potential visibility sets. It is imperative that you have read the accompanying text book before continuing with this work book. PVS and portal generation theory will not be discussed in this work book as its intention is to focus on the implementation of these processes into our compiler tool. It is expected that have read the text book and fully understand the theory and placeholder code described there.

In lab project 17.1 we will evolve the development time compiler tool initially implemented in lab project 16.2 by adding two new processing modules to it that collectively make PVS calculation for the input geometry set possible. In addition to the hidden surface removal, BSP leaf tree compile and T-junction repair modules added to our tool in the previous lesson, we will now add a portal generation processing module and a PVS calculation processing module. This will complete the construction of our compiler tool allowing it to compile and save geometry out to an IWF file along with its leaf based PVS data.

Lab project 17.2 will involve a very minor evolution from lab project 16.3 requiring only a few lines of code to be changed in order for it to render geometry using a PVS. As you have no doubt guessed, this lab project (just like 16.3) is being used to represent the run-time component demonstrating how the pre-calculated PVS data can be loaded from file and rendered efficiently by a rendering engine.

# Lab Project 17.1: Adding a PVS Calculator to the BSP Compiler

In the first implementation of our BSP compiler tool in the previous chapter, we saw how the compiler itself was constructed as a series of separate processes that were glued together by the CCompiler class. The CCompiler class itself had no idea how to compile a BSP tree, perform HSR or remove T-Junctions from compiled geometry but did know the order in which these processes had to be carried out. The CCompiler class itself had the simple task of loading the geometry from an IWF file and passing this data to the individual processes in a very specified order. The various processes were all implemented in their own classes such as CProcessHSR, CProcessTJR and CBSPTree and as such, as long as CCompiler had knowledge of these objects it could invoke their ::Process methods to instruct them to perform their specific tasks on the data set. With the exception of the application's **main** function having to initially inform the CCompiler object of the IWF file that was to be loaded, all it had to do was issue a call to the CCompiler::CompilerScene function to set this chain of events in motion. When this function returned program flow back to the application's **main** function, the CCompiler class had all the information it needed stored in its compiled BSP leaf tree. The application would finally issue a call to the CCompiler::SaveScene method which would take the information stored in the BSP tree and save it out to file.

The CCompiler::CompileScene function was essentially nothing more than a series of calls to functions that invoked the various processes. We saw that it would first invoke the CProcessHSR module to

perform hidden surface removal on the data set in an attempt to remove hidden surfaces and any illegal geometry that may exist in the data set. After this process returned and had performed its task, the CBSPTree module would be invoked to compile the leaf BSP tree. After the BSP tree had been compiled and program flow had once again returned back to the CCompiler::CompileScene method, the next and final module to be invoked was CProcessTJR which would iterate through the polygons in the compiled BSP tree and would fix any T-Junctions that were found to exist. This would involve T-Junctions that were either introduced via the myriad of clipping operations that were performed via BSP tree construction or that existed in the original loaded dataset.

With the need to now introduce two new processes into the compiler, the CProcessPRT and CProcessPVS modules will be introduced in this lesson and added to the list of processes the compiler must perform. These processes will handle the generation of portals for the data set and the calculation of its leaf based PVS respectively. The CCompiler class will need to be updated to be aware of these new modules and provide a means for storing and passing the compilation options for these modules into the modules themselves.

Because of the modular design of the compiler, the additions to the CCompiler class will be extremely light. The CCompiler::CompileScene method will be updated to invoke additional processes. The order of scene compilation will now be as follows:-

1. Load in the IWF data from file
2. Perform 'Hidden Surface Removal' on loaded geometry to removal illegal surface fragments
3. Compile BSP leaf tree
4. **Generate portals for compiled tree**
5. **Calculate PVS for BSP tree using portals generated in step 4**
6. Perform T-Junction repair on final BSP tree geometry
7. **Construct mesh from BSP geometry and save to file along with PVS and portal Data**

Steps 4 and 5 in the above list are clearly the additions to lab project 16.2 and will be the focus of this work book. These processes will be wrapped in the CProcessPRT (PRT is short for Portal) and CProcessPVS classes respectively.

Step 7 is not actually invoked by the CCompiler::CompileScene method but is activated by the application's main function called the CCompiler::SaveScene method. This function is clearly not a new function and its code is actually unchanged from the previous lesson. However, you will recall that this simple function uses our custom CFileIWF class to perform its IWF file loading and saving. It is the code to this class (in particular its WriteTree method) which has been modified so that it now knows how to write out the additional PVS and portal information that will be stored in the tree.

> **Note:** Although there is no need to save the portals out to file it can sometimes be useful to have them around. You can decide whether you will need them or not and alter the saving code accordingly.

Finally, before we start looking at the code to the new classes and the modifications to previously existing classes, it should be noted that the structures of our CBSPTree class will need to be modified to store both portal information and PVS data. Remember from the text book that all this information needs to be stored in the BSP tree. That is, we need the BSP tree to maintain a list of all the portals that were generated by the portal generation process. Furthermore, we need each of these portals to store in which

leaves they were found to reside. We also need each leaf in the BSP tree (each CBSPLeaf structure) to store an array of portal indices into the tree's portal array describing exactly which portals were found to reside in that leaf. With respect the calculation of the PVS set, the CBSPTree class must also have a place to store the compiled PVS data (the compressed visibility bit set for each leaf in the tree) and each leaf in the tree must be modified to also store a PVS index. The leaf's PVS index will describe the location in the tree's main PVS array where its visibility information will begin. You are reminded that after the PVS has been calculated, the visibility information for every leaf in the tree will be stored in this one master PVS data array which is why each leaf needs to know where within that array its visibility information is located.

We will start our discussion of the lab project 17.1 source code by examining the changes to the top level object that binds everything together, the CCompiler class.

# The CCompiler Class - Updated

The CCompiler class has changed very little from its initial conception in lab project 16.2. The complete class declaration is shown below with any new members highlighted in bold. Remembering that the CCompiler class stores a set of compilation options for each process that it invokes, we can see that it now has two new member variables PRTOPTIONS and PVSOPTIONS. These are structures that contain the compilation options for the portal generator module and the PVS calculator module respectively. Notice, how two new methods have also been added called PerformPRT and PerformPVS. These are the methods called from the CCompiler::CompileScene method to invoke the portal generation and PVS calculation modules. This is not a new concept to us as we discussed how each process invoked by the compiler has a matching member function that performs this task (such as PerformHSR and PerformTJR etc).

***Excerpt from CCompiler.h***

```
class CCompiler
{
public:

    // Constructors & Destructors for This Class.
             CCompiler();
    virtual ~CCompiler();


    // Public Functions for This Class.
    bool            CompileScene     ( );
    void            Release          ( );
    void            SetFile          ( LPCTSTR FileName );
    void            SetOptions       ( UINT Process, const LPVOID Options );
    void            GetOptions       ( UINT Process, LPVOID Options ) const;
    void            SetLogger        ( ILogger * pLogger ) { m_pLogger = pLogger; }
    CBSPTree       *GetBSPTree       ( ) const { return m_pBSPTree;   }
    bool            SaveScene        ( LPCTSTR FileName );

    void            PauseCompiler    ( );
    void            ResumeCompiler   ( );
    void            StopCompiler     ( );
    bool            TestCompilerState( );

    COMPILESTATUS   GetCompileStatus ( ) const { return m_Status; }
```

```
    void            SetCompileStatus ( COMPILESTATUS Status ) { m_Status = Status; }



    // Public Variables for This Class.
    vectorMesh      m_vpMeshList;       // A list of all meshes loaded.
    vectorTexture   m_vpTextureList;    // A list of all textures loaded.
    vectorMaterial  m_vpMaterialList;   // A list of all materials loaded.
    vectorEntity    m_vpEntityList;     // A list of all entities loaded.
    vectorShader    m_vpShaderList;     // A list of all shaders loaded.

private:

    // Private Functions for This Class.
    bool            PerformHSR( );      // Hidden Surface Removal
    bool            PerformBSP( );      // Binary Space Partition Compilation

    bool            PerformPRT( );      // Portal Compilation
    bool            PerformPVS( );      // Potential Visibility Set Compilation

    bool            PerformTJR( );      // T-Junction Repair


    // Private Variables for This Class.
    HSROPTIONS      m_OptionsHSR;       // Hidden Surface Removal Options
    BSPOPTIONS      m_OptionsBSP;       // BSP Compilation Options

    PRTOPTIONS      m_OptionsPRT;       // Portal Compilation Options
    PVSOPTIONS      m_OptionsPVS;       // PVS Compilation Options

    TJROPTIONS      m_OptionsTJR;       // T-Junction Repair Options

    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    LPTSTR          m_strFileName;      // The file used for compilation
    COMPILESTATUS   m_Status;           // The current status of the compile run
    ULONG           m_CurrentLog;       // Current logging channel for messages.

    CBSPTree       *m_pBSPTree;         // Our compiled BSP Tree.
};
```

We will see later that the portal processing module has no real options that influence the way in which the portals are compiled. Therefore, the PRTOPTIONS structure contains a single Boolean which describes whether the user would like the compiler to perform the portal generation process. The structure is defined in CompilerTypes.h and is shown below.

***Excerpt from CompilerTypes.h***

```
typedef struct _PRTOPTIONS {            // Portal Compilation Options
    bool            Enabled;            // Process Enabled ?
} PRTOPTIONS;
```

If the Enabled Boolean is set to false in the CCompiler::m_OptionsPRT structure then the compiler will not call the CCompiler::PerformPRT method and the portal generation module will not be invoked.

The PVSOPTIONS structure has three members which will influence the way in which the PVS calculator will generate its final PVS data. The PVSOPTIONS structure is also declared in the file CompilerType.h and is shown below.

***Excerpt from CompilerTypes.h***

```
typedef struct _PVSOPTIONS {            // PVS Compilation Options
    bool            Enabled;            // Process Enabled ?
    bool            FullCompile;        // Perform Full PVS Compile
    unsigned char   ClipTestCount;      // Number of portal clip tests to perform
} PVSOPTIONS;
```

The ways in which these members influence the PVS calculator module are described below.

**bool Enabled;**

This member is no stranger to us as the options structure for each process has this member. In the case of this structure, it describes to the compiler whether PVS compilation should be invoked when compiling the data. It is possible that you might just want to compile a regular leaf tree and may not be interested in PVS calculation. When this is the case, this member can be set to false.

> **Note:** If the portal generation module has been disabled then the PVS calculator will not be invoked even if this member has been set to true. The PVS calculator absolutely needs the portal information to have been generated and stored in the BSP tree in order to perform its task. Enabling the PVS process without enabling the portal generation process will simply cause the PVS module to return prematurely as soon as it is invoked. That is, the PVS module will exit immediately when it discovers there are no portals for it to work with.

**bool FullCompile;**

This is a very handy option to use when you are developing your tool and do not want to wait potentially hours for your level to compile each time you wish to run your code. You will recall from the text book that the first task performed in calculating the PVS is to generate a very course visibility set for each portal based on nothing more than portal orientations to one another. A temporary list is built for each portal describing which portals could possibly have a flow between them. This is then use to perform a flood fill through the leaves of the tree from the portal currently being processed to very quickly retrieve an array of leaf visibility bits for that portal. Any bits set to zero in this 'Possible Visibility Array' represent leaves that can not possibly be seen from the portal in question. This is done for each portal so that prior to the main clipping process beginning, we have a very rough guide as to which leaves may be visible from a given portal. The main clipping process simply refines this leaf visibility information for each portal by performing anti-penumbra clipping and setting other leaves in the 'Possible Visibility Set' to zero as they are found to exist outside any valid anti-penumbra originating at the current source portal. The result at the end of the PVS process is the 'Potential Visibility Array' for each portal. This is a much refined version of the 'Possible Visibility Array' that was initially calculated at the start of the process and will typically contain much fewer leaves in its visibility set.

Having said all this, it is the calculation of the 'Potential Visibility Array' for each portal that takes so much time due to the exceptionally recursive nature of the procedure and the grotesque amount of clipping that will need to be done to the anti-penumbra planes. It is this process that makes sure that we have the tightest potential visibility set possible. However, this process takes hours to complete on complex data sets and you certainly don't always need to have a full PVS compile performed when testing your application or level geometry. Imagine for example that your artist had just added a new asset to the scene and wanted to see it being rendered in-game. The entire scene would have to be

compiled again forcing the development team to wait several hours before the new level was compiled and able to be loaded into the PVS aware game render engine.

Often, at the development time we are not nearly so picky when testing things such as new assets or new light placement within the scene that our game is actually running with the most optimal PVS data set and would gladly trade off the hours of compile time for a PVS data set that compiles in seconds even if the resulting PVS set generated halved the frame rate of the application. After all, you can save the full-blown PVS compile for when the game is about to burned and shipped.

The calculation of the 'Possible Visibility Array' for each portal is compiled just prior to the main clipping process to aid in the speed up of that process. This array can be thought of as a very over generous visibility set for each portal. As mentioned, the only thing taken into consideration when calculating this array for each portal is the orientations of the portals with respect to one another. No occlusion is considered. However, this data is still in PVS format and can be used as the PVS data instead of calculating the main data set via the core clipping procedure. It takes seconds/minutes to compile instead of hours and is ideal for performing a quick PVS build in order to test some other assets in your game engine. Therefore, if this option is not set to true, a full compile will not be performed and instead, the PVS data calculated will contain the leaf visibility information returned from the simple flood fill for each portal. The main anti-penumbra clipping procedure will not have been invoked to further refine this set. If this option is set to true, the full blown calculator will be invoked and a tight potential visibility set will be laboriously calculated.

**unsigned char ClipTestCount;**

In the text book we examine how when performing the anti-penumbra clipping process, we could further refine the amount of geometry that made it into the anti-penumbra's volume by building it four times. This would allow us to trim the size of the source and generation portals that got passed into the next recursion by making sure we have the tightest clip planes possible.

In the first step the anti-penumbra planes are built from the source portals vertices to the target portal edges and the generator portal is clipped to it. In the second step we would then reverse the order by building an anti-penumbra from each vertex in the target portal to each edge in the source portal. Once again the generator portal would be clipped to these planes. We showed in the accompanying text book that it was entirely possible for tighter planes to be generated from target to source instead of from source to target and vice versa. Therefore, we build an anti-penumbra in both directions to make sure we clip the maximum possible from the generator portal and reduce its size as much as possible for the next recursion. Obviously, if the generator portal does not survive the any of the two clipping stages described above, the portal is considered not to be visible from the current source portal and we do not recur through it into the neighbor leaf.

In the third test we build an anti-penumbra from what is left of the generator portal and the target portal to create an anti-penumbra with which we can clip the source portal too. We also reverse the plane order and clip the source portal to the anti-penumbra planes generated from the target to the generator portal in a $4^{th}$ clip test. Once again, if at any point the source portal is entirely clipped away it means that the generator portal can not see it and therefore, no line of sight can possibly exist from the source portal through the generator portal. The temporarily clipped source portal is also passed into the next recursion

with the clipped generator portal so that as we step through the process for a given source portal, the source, target and generator portals becomes smaller and smaller allowing us to generate smaller and tighter anti-penumbras in future recursions and hopefully reject more generator leaves from being recurred into and added to the source portal visibility set.

So we have seen that if all these clipping operations are to be performed, we essentially have to build four anti-penumbras and clip four portals to those anti-penumbras for each generator portal that we visit during the core recursive clipping process. As this clipping is obviously a laborious process to perform each time, the number of these clips that we perform can greatly influence the speed of PVS calculation. When generating the commercial data set it is recommended that you activate all four clipping stages to get the tightest possible PVS data set generated. However, during development tests and debug runs you can perform fewer clip tests. This will speed up PVS compile time at the cost of a more generous PVS in the typical case.

While having an option that allows us to toggle the number of clips performed from between 1 to 4 may seem like a trivial speed up option, it is not the case. Each anti-penumbra test performed means first constructing the anti-penumbra. This involves two loops that iterate through each vertex in the source portal and each edge in the target portal. A plane in constructed for each combination and the source and target portals are classified against it to determine if it is a separating plane. Two more loops enter the fray here as we have to test each vertex in each of the portals against the plane to determine its separation status. Only if the plane has the source and target portal in opposite half spaces will the plane be added to the anti-penumbras clip list. So as we can see, just the generation of the anti-penumbra generated four nested loops. Then of course, we have to loop through each anti-penumbra plane clipping the generator portal to each one which can involve memory allocations and de-allocations when the portal gets clipped. That is an awful lot of work to perform four times for each generator portal that we visit when calculating the visibility set for each source portal.

The ClipTestCount option of this structure can be set from 1 to 4 to determine how many of these clips we would like to perform at each generator portal. Performing only one clip will generate the PVS data array much faster but will typically have a larger number of leaves flagged as visible in each leaf's potential visibility set.

So we have seen that these two new CCompiler member structures contain compilation options for the portal generation process and the PVS process. Although we will set the compile options for each process in the CCompiler constructor, the options for each process can also be set by the application via the CCompiler::SetOptions function. We will look at the modified code to this method in a moment.

Let us now examine the changes to the CCompiler source code staring with the constructor.

## Constructor - CCompiler

The constructor of CCompiler is used to set up the default compile options for each of its modules. The compile options for every module can be altered and retrieved by the application once the CCompiler object has been created via its SetOptions and GetOptions methods respectively.

We saw in the previous lab project how the default options were configured for the hidden surface removal, BSP compile and T-Junction repair modules in this constructor. Now we have added code to also set the default compile options for the portal generation and PVS calculator modules.

```
CCompiler::CCompiler()
{
    // Set up default HSR options
    m_OptionsHSR.Enabled            = true;

    // Set up default BSP options
    m_OptionsBSP.Enabled            = true;
    m_OptionsBSP.TreeType           = BSP_TYPE_SPLIT;
    m_OptionsBSP.SplitHeuristic     = 3.0f;
    m_OptionsBSP.SplitterSample     = 60;
    m_OptionsBSP.RemoveBackLeaves   = true;
    m_OptionsBSP.AddBoundingPolys   = false;

    // Set up default Portal Compile Options
    m_OptionsPRT.Enabled            = true;

    // Set up default PVS Options
    m_OptionsPVS.Enabled            = true;
    m_OptionsPVS.FullCompile        = true;
    m_OptionsPVS.ClipTestCount      = 2;

    // Set up default TJR Options
    m_OptionsTJR.Enabled            = true;

    // Reset Vars
    m_strFileName = NULL;
    m_pBSPTree    = NULL;
    m_pLogger     = NULL;
    m_Status      = CS_IDLE;

}
```

As you can see, the hidden surface removal module has a single compiler option which dictates whether it should be invoked or not. By default, all modules are enabled. As mentioned though, the application can change these settings via CCompiler methods prior to calling the CCompiler::CompileScene method.

The default parameters for the BSP tree build are also unchanged. We enable the process and specify that we would like to build a tree in which the resulting polygons are split/clipped to the leaf nodes in which they reside. We set the splitter choosing heuristic to 3.0 so that we weight the reduction of splits as three times more important than tree balance during splitter selection. We also set the splitter sample to 60 so that in order to speed up BSP tree compile, only the first 60 polygons in the list that make it into each node are tested as split plane candidates. We also specify that we would like any geometry that makes it into back leaves (such as illegal geometry fragments causes by floating polygon rounding errors during the clipping process) to be deleted so that accurate solid/empty space information can be maintained. By default, we also specify that we would not like the BSP compiler to seal our level by surrounding it in an inward facing cube prior to being compiled.

The portal generation options structure has only a single Boolean member describing whether or not this process should be enabled. As mentioned, this is set to true because all modules are enabled by default. The same is true for the T Junction removal options structure.

By default we also enable the PVS calculator and set it to perform a full compile (instead of a quick portal flood) to generate its PVS data. We also set the number of clips to perform at each generator portal to 2 by default. The minimum is one and the maximum is four

## SetOptions - CCompiler

The CCompiler::SetOptions function is also not a new function but has now been modified to allow the application to set the compilation options of the two new processes. You will recall that this function accepted as its first parameter the numerical ID of the process which is to have its properties set. The numerical IDs are defined in the file Common.h. We have now added two more numerical defines called PROCESS_PRT and PROCESS_PVS which are used to signify the portal generation module and the PVS calculation modules respectively. Shown below is our current list of numerical defines.

***Excerpt from Common.h***

```
#define PROCESS_HSR         0   // Hidden Surface Removal
#define PROCESS_BSP         2   // Binary Space Partition
#define PROCESS_PRT         3   // Portals
#define PROCESS_PVS         4   // Potential Visibility Set
#define PROCESS_TJR         5   // T-Junction Repair
```

The second parameter to the CCompiler::SetOptions function is a void pointer which the application can use to pass the relevant options structure for the process it wishes to alter the compilation parameters for. Here is the modified version of the function.

```
void CCompiler::SetOptions( UINT Process, const LPVOID Options )
{
    switch (Process)
    {
        case PROCESS_HSR:
            m_OptionsHSR = *((HSROPTIONS*)Options);
            break;


        case PROCESS_BSP:
            m_OptionsBSP = *((BSPOPTIONS*)Options);
            break;

        case PROCESS_PRT:
            m_OptionsPRT = *((PRTOPTIONS*)Options);
            break;

        case PROCESS_PVS:
            m_OptionsPVS = *((PVSOPTIONS*)Options);
            break;

        case PROCESS_TJR:
            m_OptionsTJR = *((TJROPTIONS*)Options);
            break;

    } // End Switch
```

```
}
```

As you can see, two more case statements have been added to cast the void pointer to either a PRTOPTIONS structure or a PVSOPTIONS structure based on the passed process ID. In each case we copy the contents of the passed structure into the relevant member structure to set the compile options for the passed process. For example, the application could set the options for the PVS process to compile only a 1 clip anti-penumbra PVS by using the following code.

```
CCompiler Compiler;
CompilerSetFile("SomeExampleFile.iwf")

PVSOPTIONS pvsOptions;
pvsOptions.Enabled            =     true;
pvsOptions.FullCompile        =     true;
pvsOptions.ClipTestCount      =     1;

Compiler.SetOptions( PROCESS_PVS , (void*) &pvsOptions );

… Set other process options here …

Compiler.CompileScene ();
```

This is just a simple example but shows how an application using the compiler can configure the compile options of the various processes prior to calling the CompileScene method.

## GetOptions - CCompiler

As one might expect, the CCompiler class also exposes a member function to allow the application to fetch the compiler options for a given process. The function takes two parameters with the first being the numerical index of the process the application would like to retrieve the compile time options for. The second parameter is a void pointer to the options structure for the applicable process that will be cast to the correct type inside the function and filled with the compiler settings for that process as shown below.

```
void CCompiler::GetOptions( UINT Process, LPVOID Options ) const
{
    switch (Process)
    {
        case PROCESS_HSR:
            *((HSROPTIONS*)Options) = m_OptionsHSR;
            break;



        case PROCESS_BSP:
            *((BSPOPTIONS*)Options) = m_OptionsBSP;
            break;

        case PROCESS_PRT:
            *((PRTOPTIONS*)Options) = m_OptionsPRT;
            break;

        case PROCESS_PVS:
            *((PVSOPTIONS*)Options) = m_OptionsPVS;
```

```
            break;

        case PROCESS_TJR:
            *((TJROPTIONS*)Options) = m_OptionsTJR;
            break;

    } // End Switch
}
```

Once again, this is not a new method to us but has had two new cases added to the switch statement that allow us to retrieve the compiler options for the portal generation and PVS generation modules.

## CompileScene - CCompiler

As we discussed in the previous lesson, it is the CCompiler::CompileScene method that glues the various processes together into a geometry compilation pipeline. In the previous lesson, we saw that this method was responsible for instantiating a CFileIWF object and using it to import the IWF information into its internal mesh, material, texture and entity arrays. This data is then copied from the CFileIWF data vectors into the compiler's own vectors where data that is not related to the BSP compile process (such as entities and materials for example) can be stored and then written back out to the resulting compiled IWF file. None of this is new and is shown below.

```
bool CCompiler::CompileScene( )
{
    CFileIWF IWFFile;

    // Validate Data
    if (!m_strFileName) return false;

    // Retrieve the filename portion only of the string
    LPTSTR FileName = NULL;
    FileName = _tcsrchr( m_strFileName, _T('\\') );
    if (!FileName) FileName = _tcsrchr( m_strFileName, _T('/') );
    if (!FileName) FileName = m_strFileName;
    if (FileName[0] == _T('\\') || FileName[0] == _T('/')) FileName++;

    try
    {
        // We Are starting the process
        m_Status = CS_INPROGRESS;

        // Load the specified file
        IWFFile.Load( m_strFileName );

        // Obtain ownership of the objects loaded
        m_vpMeshList     = IWFFile.m_vpMeshList;
        m_vpMaterialList = IWFFile.m_vpMaterialList;
        m_vpTextureList  = IWFFile.m_vpTextureList;
        m_vpEntityList   = IWFFile.m_vpEntityList;
        m_vpShaderList   = IWFFile.m_vpShaderList;

        // Clear out the IWF's object vectors (don't destroy)
        IWFFile.m_vpMeshList.clear();
        IWFFile.m_vpMaterialList.clear();
        IWFFile.m_vpTextureList.clear();
        IWFFile.m_vpEntityList.clear();
        IWFFile.m_vpShaderList.clear();
```

```
        // Write load success
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                                 0,
                                 true,
                                 _T("Successfully loaded geometry from file '%s'"),
                                 FileName );

        } // End if Logger


    } // End try Block

    catch (HRESULT & e)
    {
        // Write Load Failure
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                        LOGF_ERROR,
                        true,
                       _T("Failed to load geometry from file '%s' with error code '0x%x'"),
                       FileName, e );

        } // End if Logger

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
    catch (...)
    {
        // Write Load Failure
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_GENERAL,
                                 LOGF_ERROR,
                                 true,
                                 _T("Failed to load geometry from file '%s'"),
                                 FileName );

        } // End if Logger

        IWFFile.ClearObjects();
        Release();
        return false;

    } // End Catch Block
```

At this point the CFileIWF file data has all been copied into the compiler's internal vectors so the CFileIWF data objects can be released as they are no longer needed.

Now we will start the actual compilation process by invoking the various modules one at a time and passing the results of each module onto the next in the chain. As we saw in the previous lesson, the hidden surface removal module is invoked first to remove any illegal geometry and then the BSP leaf tree compiler is invoked to compile the BSP tree. In the previous lesson, the next and final step prior to saving the compiled data out to disk was to perform T-Junction repair on the final compiled polygon

data. However, we now have to processes that must be inserted after the BSP compile process and before the T-Junction repair process. That is, after the BSP tree has been compiled we will then invoke the portal generator to generate the portals for the compiled geometry. These portals will be stored in the BSP tree. The next module to be invoked will be the PVS calculator which will use the portals stored in the tree to generate the final compressed PVS for the BSP tree. This final data will also be stored in the BSP tree in the form of a single compressed data array. Each leaf in the tree will also contain a numerical index into this master array describing where in the array its PVS data begins.

Providing a logging object has been assigned to the CCompiler class we first output to the general channel that the compile process is about to begin. The first parameter to the LogWrite method specifies that this message should be output to the general channel and the second parameter describes this message as being a normal status message so that the default ink color is used for the text (only when being used with a GUI logger). The third parameter instructs the logger that this is the start of a new message so that prior to being printed the logger inserts a line feed to move the text output cursor to the start of a new line.

```
    // Write Log Info
    if ( m_pLogger )
    {
        m_pLogger->LogWrite(LOG_GENERAL,
                            0,
                            true,
                            _T("Beginning compilation run \t\t\t- ") );

    } // End if Logger
```

In the next section we step through each possible module that can be performed and test its options structure to test that its **Enabled** Boolean is set to true. If it is then this means the application would like the compiler to perform that process (assuming the compiler has not been placed into a state where the user has cancelled the compile). As you can see, we first call the CCompiler::PerformHSR function which will invoke the hidden surface removal module and instruct it to removal illegal geometry from the compiler's data set. The second process to be performed is the BSP compiler which is activated via a call to the CCompiler::PerformBSP method. Both of these methods were discussed in the previous lesson.

```
    // Start compiling by removing all hidden surfaces
    m_CurrentLog = LOG_HSR;
    if ( m_OptionsHSR.Enabled && m_Status != CS_CANCELLED) PerformHSR();

    // Build the BSP Tree if requested
    m_CurrentLog = LOG_BSP;
    if ( m_OptionsBSP.Enabled && m_Status != CS_CANCELLED) PerformBSP();
```

Out of the final three modules that are activated (shown below), the first two are new to this application and show two new methods of CCompiler which will be used to invoke the portal generation processor and the PVS calculator modules respectively. The third and final process to be activated is the T-Junction repair module via a call to the CCompiler::PerformTJR method.

```
    // Build the portals if requested
```

```
    m_CurrentLog = LOG_PRT;
    if ( m_OptionsPRT.Enabled && m_Status != CS_CANCELLED) PerformPRT();

    // Build the PVS if requested
    m_CurrentLog = LOG_PVS;
    if ( m_OptionsPVS.Enabled && m_Status != CS_CANCELLED) PerformPVS();

    // Repair any T-Juncs if requested
    m_CurrentLog = LOG_TJR;
    if ( m_OptionsTJR.Enabled && m_Status != CS_CANCELLED) PerformTJR();

    // Clean up if required
    if ( m_Status == CS_CANCELLED ) Release();

    // Processing Run Completed
    m_Status = CS_IDLE;

    // Write end of compilation message (We use warning just to make it blue ;)
    if ( m_pLogger ) m_pLogger->LogWrite( LOG_GENERAL,
                                          LOGF_WARNING | LOGF_ITALIC,
                                          false,
                                          _T("Success") );

    // Success!
    return true;
}
```

As you can see, when the T-Junction removal method returns, the level data will have been compiled and all that is left to do is output the success message (providing the compiler was not cancelled mid way through the chain of events).

The two new sections of code are highlighted in bold above and show the calls to the PerformPRT and PerformPVS methods. These are simple methods that invoke the new modules we will develop in this work book. The code to the PerformPRT method is discussed next.

## PerformPRT - CCompiler

The CCompiler::PerformPRT method is called by the CCompiler::CompileScene method after the BSP tree has been compiled. Its task is to initialize and invoke the portal generation module. Although we have not yet discussed the code to the CProcessPRT class (the portal generation module), we can see in the following code how the module is initialized and used. You will notice that this module shares the same interface methods as the other modules making the way in which the compiler interacts with each module consistent for the most part. Each module for example has methods that allow us to set its options, set the logging class to be used and to pass a pointer to the parent CCompiler object that is invoking the module. The ProcessPRT module is only used during the portal generation process and once its Process method has returned, the portal information will all be stored in the BSP tree. Therefore, the CProcessPRT object can be instantiated on the stack as shown in the following code.

```
bool CCompiler::PerformPRT()
{
    // One time compile process
    CProcessPRT ProcessPRT;

    // Set our process options
    ProcessPRT.SetOptions( m_OptionsPRT );
```

```
    ProcessPRT.SetLogger( m_pLogger );
    ProcessPRT.SetParent( this );
```

Now that we have allowed the CProcessPRT object to store a pointer to the logger class our application is using, we will now write some information to the logger stating that the portal generation module is about to start. We clear the logger's channel that has been reserved for portal generation output (LOG_PRT) and output some copyright information and information instructing the user that the portal compilation process is about to begin.

```
    // Write Log Information
    if ( m_pLogger )
    {
        m_pLogger->Clear( LOG_PRT );
        m_pLogger->LogWrite( LOG_PRT,
                             LOGF_WARNING | LOGF_BOLD  ,
                             false,
                              _T("\r\nPortal Processor v1.0.0\r\n"));

        m_pLogger->LogWrite( LOG_PRT,
                             LOGF_WARNING | LOGF_ITALIC,
                             false,
                             _T("Copyright © 2005 GameInstitute.com.
                                 All Rights Reserved.\r\n"));

        m_pLogger->LogWrite( LOG_PRT,
                             0,
                             true,
                             _T("Beginning portal compilation process."));

    } // End if Logger Available
```

As the CProcessPRT object will need access to the BSP tree that it is to generate portals for, its 'Process' method accepts a pointer to a BSP tree as its only parameter. It is this function that is the top level function for the entire portal generation process. That is, when the CProcessPRT::Process function returns, every portal will have been created and will be stored in the BSP tree's portal array. The leaves of the BSP tree will also contain portal indices describing which portals they contain and every portal will contain a two element leaf array describing the indices of the leaves in which they reside.

Here is the remainder of the function that calls the 'Process' method to generate the portals and finally outputs a completion method prior to returning.

```
    // Compile the Portal set
    ProcessPRT.Process( m_pBSPTree );

    // Write Log Information
    if ( m_pLogger )
    {
        if ( m_Status != CS_CANCELLED )
            m_pLogger->LogWrite( LOG_PRT,
                                 0,
                                 true,
                                 _T("Portal compilation completed successfully."));
        else
            m_pLogger->LogWrite( LOG_PRT, 0, true, _T("Portal compilation cancelled."));
```

```
    } // End if Logger Available

    // Success!!
    return true;
}
```

If the process was cancelled for some reason during the portal compile then a cancellation message is output to the portal process logging channel instead.

## PerformPVS - CCompiler

This method is almost a duplicate of the previously discussed function with the exception that it instantiates and invokes the CProcessPVS module. We will look at the code to the CProcessPVS class later in the lesson but for now just know that its Process method will build the master compressed PVS data array for the PBS tree.

A CPerformPVS object is instantiated on the stack and the PVS options structure is passed into its SetOptions method. We also inform the module of the logging object we are using and pass a pointer to the CCompiler object that is invoking the module. We then clear the PVS logging channel and output copyright information about the PVS module to that channel.

```
bool CCompiler::PerformPVS()
{
    // One time compile process
    CProcessPVS ProcessPVS;

    // Set our processor options
    ProcessPVS.SetOptions( m_OptionsPVS );
    ProcessPVS.SetLogger( m_pLogger );
    ProcessPVS.SetParent( this );

    // Write Log Information
    if ( m_pLogger )
    {
        m_pLogger->Clear( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS,
                             LOGF_WARNING | LOGF_BOLD,
                             false,
                             _T("\r\nPVS Processor v1.0.0\r\n"));

        m_pLogger->LogWrite( LOG_PVS,
                             LOGF_WARNING | LOGF_ITALIC,
                             false,
                             _T("Copyright © 2005 GameInstitute.com.
                                 All Rights Reserved.\r\n"));

        m_pLogger->LogWrite( LOG_PVS,
                             0,
                             true,
                             _T("Beginning visibility determination process."));

    } // End if Logger Available
```

After informing the user that the PVS calculator is beginning its process, we call the CProcessPVS::Process method to invoke the top level PVS processing method. When this function returns, the passed BSP tree will have had the master PVS data array stored within it and each leaf will contain and index into this master array describing where its visibility information begins.

```
    // Begin the PVS Process
    ProcessPVS.Process( m_pBSPTree );

    // Write Log Information
    if ( m_pLogger )
    {
        if ( m_Status != CS_CANCELLED )
            m_pLogger->LogWrite( LOG_PVS,
                                 0,
                                 true,
                                 _T("Visibility determination completed successfully."));
        else
            m_pLogger->LogWrite( LOG_PVS,
                                   0,
                                   true,
                                   _T("Visibility determination cancelled."));

    } // End if Logger Available

    // Success!!
    return true;

}
```

As you can see, after the CProcessPVS::Process method returns, we test the status of the compiler and output either a completion message or a cancellation message to the PVS logging channel depending on the outcome.

We have now discussed all the code changes to the CCompiler class and as we have seen, the modular design of the compiler itself makes adding future modules extremely easy. Of course, most of this work book will be dedicated to examining the code to these two new modules (CProcessPRT and CProcessPVS) but before we do, we must look at other classes and structures that will need to be changed to accommodate the storage of the information these two modules will provide our tree.

In the next section we will examine the changes to the CPolygon class and will examine a new CPolygon derived class called CBSPortal. The CBSPPortal object will be used by the CProcessPRT and CBSPTree objects to generate and store the portals of the tree respectively. As discussed in the accompanying text book, a portal is simply a polygon with some additional information packaged with it (such as which leaves in the BSP tree they reside in).

# The CPolygon Class - Updated

When discussing the generation of portals in the accompanying text book, we learned that the first step in generating a portal is to create a large initial polygon on the plane of the node (currently having a portal generated for it) that is large enough to at least fill the root node's bounding box along that plane. It was this initial portal that was passed into the BSP tree and clipped to the nodes of the tree so that any

portal fragments that existed in solid space were clipped away. The result of this clipping process was a portal that described the exact shape and size of the 'doorway' between two leaves.

Generating an initial portal on the node plane that was large enough to fill the root node's bounding box was discussed in detail in the text book and requires two inputs, the plane on which the polygon/portal should be constructed an a bounding box describing how large it should be on the plane. As it is pretty useful in other situations to build a polygon on a certain plane that we know will fill some bounding box, we have decided to build this functionality straight into the CPolygon class so that all derived classes (including CBSPPortal) expose it. The modified CPolygon declaration is shown below with the new method highlighted in bold.

***Excerpt from CompilerTypes.h***

```
class CPolygon
{
public:
        // Constructors & Destructors
                  CPolygon( );
         virtual ~CPolygon( );


        // Public Variables for This Class
        CVertex              *Vertices;             // Polygon vertices
        unsigned long   VertexCount;                // Vertices in this poly


        // Public Functions for This Class
        long             AddVertices( unsigned long nVertexCount = 1 );
        long             InsertVertex( unsigned long nVertexPos );
        void             ReleaseVertices();


        // Public Virtual Functions for This Class
        virtual HRESULT Split( const CPlane3& Plane,
                               CPolygon * FrontSplit,
                               CPolygon * BackSplit,
                               bool bReturnNoSplit = false );

       virtual bool    GenerateFromPlane( const CPlane3& Plane,
                                          const CBounds3& Bounds );

};
```

The GenerateFromPlane method accepts two parameters. The first is the plane on which the polygon should generate its vertices and the second parameter describes a bounding box which the polygon should at least fill (it may be bigger). As the CBSPPortal class used by the portal generator is derived from this class it too will expose this method. This means, the portal generator can simply call this function to generate the initial portal on the node plane by passing in the node that is currently being processed and the bounding box of the root node. This portal can then be passed down the tree and clipped at solid leaves.

## GenerateFromPlane - CPolygon

As discussed in the text book, generating a polygon on a plane of a specific size involves first projecting the center of the bounding box onto the node plane to calculate the center of the polygon we are creating. Projecting the center of the bounding box onto the plane is a simple matter of classifying the bounding box center position against the plane to get the distance to the plane from the center point along the plane normal. We can then move the bounding box center point along the plane normal to position it on the plane.

Figure 17.1 shows the center of the bounding box labeled CB and shows how moving it along the plane normal by the distance from CB to the plane creates the point CP. CP is the new center point of our polygon and the point from which the four vertices of the portal quad will be placed relative too.

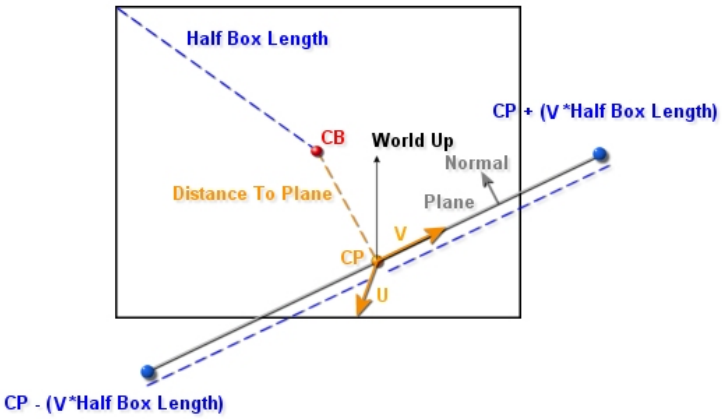



**Figure 17.1**

After CP has been calculated we have to generate the tangent and bi-normal vectors of the plane shown as vectors U and V in figure 17.1.

Generating the U and V vectors is simple with the cross product at our disposal. We first find any vector which is not identical to the plane normal. This is important because by crossing this vector with the normal we will get vector U, a vector that is perpendicular to the plane normal and thus tangent to the plane. If the vector we choose to cross with the normal is identical to the normal the two input vectors to the cross product will be the same and the resulting vector is undefined.

In figure 17.1 the world up vector is used but any vector can be used as long as it is not the same as the normal vector. In our code we test the world X, Y and Z axis vectors and choose the one that is least aligned with the plane normal. This assures that our cross product will not have any epsilon issues if the two input vectors are nearly equivalent. Once vector U is generated we can simply cross it with the plane normal to get vector V. At this point, these vectors should be unit length.

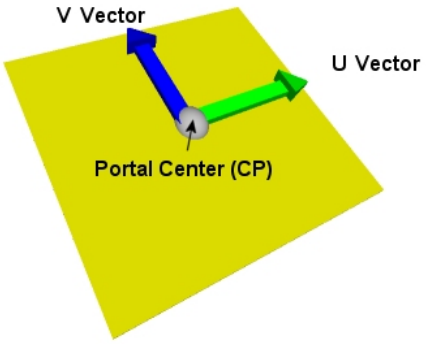



**Figure 17.2**

Next we get the half length vector of the bounding box by subtracting from the maximum box extent vector, the center position of the box. We then retrieve the length of this resulting half vector. Once we have the length we can use this to scale the U and V tangent vectors so that they can be used in combination with the polygon center point to describe the four corner vertex positions if the polygon. These four vertices are then generated and stored in the polygons vertex array.

> **Note:** It is clear that as this method generates the vertices of the polygon, it should only be called for CPolygon's (and derived objects) which currently contain no vertex data. For example,

```
CPolygon Poly;
Poly.GenerateFromPlane ( SomePlane , SomeBox);
```

The code to this method is fairly short and is shown below. We first calculate the center of the polygon by calculating the distance from the bounding box center point to the plane and then moving the point along the reverse plane normal to locate a point on the plane (CP).

```
bool CPolygon::GenerateFromPlane( const CPlane3& Plane, const CBounds3& Bounds )
{
        CVector3 CB, CP, U, V, A;

        // Calculate BBOX Centre Point
        CB = Bounds.GetCentre();

        // Calculate the Distance from the centre of the bounding box to the plane
        float DistanceToPlane = CB.DistanceToPlane( Plane );

        // Calculate Centre of Plane
        CP = CB + (Plane.Normal * -DistanceToPlane );
```

Next we need to generate the tangent vector and in order to do this we need a vector to cross with the plane normal which must not be identical to the plane normal. The vector A is calculated by analyzing the plane normal components and generating an axis aligned vector that is least aligned to the normal.

```
        // Calculate Major Axis Vector
        A = CVector3(0.0f,0.0f,0.0f);

         if( fabs(Plane.Normal.y) > fabs(Plane.Normal.z) ) {
             if( fabs(Plane.Normal.z)  < fabs(Plane.Normal.x) ) A.z = 1; else A.x = 1;
        } else {
             if (fabs(Plane.Normal.y) <= fabs(Plane.Normal.x) ) A.y = 1; else A.x = 1;
    } // End if
```

We then cross vector A with the plane normal to generate vector U and then cross the normal with vector U to generate vector V. These vectors are then both normalized.

```
        // Generate U and V vectors
        U = A.Cross(Plane.Normal);
        V = U.Cross(Plane.Normal);
        U.Normalize(); V.Normalize();
```

In the next step we calculate the length of the vector from the center of the bounding box to the box corner which describes the furthest the edges of the box could ever be from the polygon. We then scale the unit tangent vectors by this amount.

```
        float Length = (Bounds.Max - CB).Length();

        // Scale the UV Vectors up by half the BBOX Length
        U *= Length; V *= Length;
```

With these two vectors and the polygon center point we can now combine them to generate the position vectors of the four vertices of the quad polygon we wish to create on this plane which are stored in a temporary 3D vector array called P in the following code.

```
        CVector3 P[4];
        P[0] = CP + U - V; // Bottom Right
        P[1] = CP + U + V; // Top Right
        P[2] = CP - U + V; // Top Left
        P[3] = CP - U - V; // Bottom Left
```

With these corner positions computed we then allocate space in the polygon's vertex array to store these four vertices and then copy them into the vertices one by one. The normal of each vertex is assigned the normal of the plane.

```
        // Allocate new vertices
        if (AddVertices( 4 ) < 0) return false;

        // Place vertices in poly
        for ( int i = 0; i < 4; i++)
        {
             Vertices[i] = CVertex(P[i]);
             Vertices[i].Normal = Plane.Normal;
         } // Next vertex

    // Success!
    return true;
}
```

If you are feeling a little rusty on this procedure, please refer back to the accompanying text book where we describe this process in more detail. This completes our coverage of the modified CPolygon object. As you can see, we simply added a new member function.

# The CBSPPortal Class

The portal generation module will need a structure with which it can use to represent portal data within the BSP tree. Although a portal is just a polygon (geometrically speaking), we can not use CPolygon to represent them as we need to package additional information with each portal. Such information includes storing the indices of leaves in which the portal is found to reside and the number of leaves in which the portal resides. As discussed in the text book, each valid portal will always reside in exactly two leaves so this leaf count member would be used during portal generation to delete any portal fragments that are found not to exist in two. The portal must also store information about the node in the tree on whose plane it has been created. This is also important information to have during the portal generation process as it allows us to identify the two leaves under that node which are the valid leaves in which the portal should reside. If the portal is found to pop out in a leaf which is above the portals owner node, it is a rogue fragment and can be discarded.

> **Note**: It is vitally important that you have read the accompanying text book before progressing through this work book. Portal and PVS generation is a complex subject and theory of these processes will not be rehashed again in this work book. In short, if you have not studied the text book you find you understand very little of the discussions that follow.

The process that clips the portal to the BSP tree will at some point through the process will have split the initial portal in a list of portal fragments. As these fragments must be grouped together and passed

through the tree also, the portal structure will also have a **NextPortal** member that will allow us to string multiple CBSPPortal structures together in a linked list.

The CBSPPortal class is declared in CBSPTree.h and is shown below. Notice that it is derived from CPolygon so we do not have the reinvent the wheel with respect to its vertex management and member functions. This means the CBSPPortal will also expose the 'CreateFromPlane' member previously discussed which will be used by the portal generator to build the initial portal quad on the node plane.

***Excerpt from CBSPTree.h***

```
class CBSPPortal : public CPolygon
{
public:

    // Constructors / Destructors for this Class
    CBSPPortal( );

   // Public Variables for This Class
    CBSPPortal     *NextPortal;         // Linked List Next Portal Item
    unsigned char   LeafCount;          // In how many leaves this portal resides
    unsigned long   OwnerNode;          // Node that created this portal
    unsigned long   LeafOwner[2];       // Front / Back Leaf Owner Array

   // Virtual Public Functions for This Class
   virtual HRESULT Split( const CPlane3& Plane,
                          CBSPPortal * FrontSplit,
                          CBSPPortal * BackSplit );

};
```

Notice that we have added four new members to those inherited from CPolygon which allow us to store the indices of the leaves in which the portal is eventually found to reside and the node the node on which the portal was generated. Notice however that we also implement a new Split function that accepts CBSPPortal pointers. Do not worry! We do not have to implement a polygon splitting function all over again. As you will see in a moment, the Split function simple calls the base class's Split function and then copies the extra portal information into the two resulting front and back splits. Let us list those member variables and describe their purpose.

**unsigned long OwnerNode;**
As we know, all the nodes of the BSP tree are stored in its master node array. We also know that an initial portal will need to be generated on every node that does not have solid space behind it. This portal will then be passed down the tree and any portions of the portal that exist in solid space will be clipped away. This member contains the index of the node in the tree's node array whose plane this portal was created on. This is used during portal generation to make sure that when a portal ends up in a leaf, it is a leaf that is below the owner node in the tree. If this is not the case then this is a rogue fragment and can be deleted.

**unsigned long LeafOwner[2];**
Every valid portal will always exist in exactly two leaves as a portal by its very nature is a polygon that represents a door way between leaves and as such, has a leaf on either of its sides. Each valid portal will contain the indices of the two leaves in which it resides in this array. Therefore, this array tells us the two leaves for which this portal forms a doorway.

**unsigned char LeafCount;**
This member is used during portal generation to keep track of how many leaf indices we have currently added to the above array. This member will start of at zero when the initial portal is first fed into the root node of the tree and will be incremented each time the portal pops out in a leaf. This will never be higher that 2 as a portal can only ever possibly exist in two leaves. However, that does not mean that the initial portal will not end up in many valid portals being created which all exist in a different set of leaves throughout the level. Remember, when a split plane is chosen for BSP creation, the entire geometry set beneath that node in the tree is split. That split may have caused multiple convex leaves to be created across the entire level and therefore, therefore, multiple portals may exist on this node plane to bridge the gaps between those pairs of leaves.

**CBSPPortal *NextPortal;**
As we showed in the text book, the portal/BSP tree clipping process will require keeping track of al the fragments that a portal gets split into via linked lists. This member allows us to storing multiple CBSPPortal fragments together into a linked list so that we can pass the entire list down the tree by simply passing a pointer to the head of the list through the recursive process.

## Constructor - CBSPPortal

The constructor of CBSPPortal simply initializes the portals members to their invalid default values. Each element in the LeafOwner array is set to zero and so is the leaf count. The NextPortal member is set to NULL and the owner node index is set to -1.

```
CBSPPortal::CBSPPortal()
{
    // Initialise any class specific items
    LeafOwner[0]    = 0;
    LeafOwner[1]    = 0;
    NextPortal      = NULL;
    LeafCount       = 0;
    OwnerNode       = -1;
}
```

When the portal object is first created we can see that it is not a valid portal. The portal generation process, after creating a new CBSPPortal object will first construct its geometry via a call to its (inherited) GenerateFromPlane method. This will build the polygon on the chosen node plane and will fill the CBSPPortal with geometry representing a large quad on the node plane. This CBSPPortal will then be passed down and clipped to the BSP tree where the above members will eventually get assigned their final values.

## Split - CBSPPortal

Every time a portal is split during the portal generation process, we must not only split the geometry into two child portals but must also carry over the information recorded in the parent portal into the two children. It is vitally important that the journey of the parent is inherited by the children before the parent is deleted for the portal generation process to work. For example, if it has already been

determined that the parent portal was found to exist in leaf 10, we know that both the children will also exist in leaf 10 assuming they are not found to exist in solid space further down the tree and are deleted. Therefore, we must make sure that this information is carried over into the split fragments.

As you can see in the code that follows, the CBSPortal::Split method simply wraps a call to the CPolygon::Split method which additional code to copy the portal information over into the two child splits.

```
HRESULT CBSPPortal::Split( const CPlane3& Plane,
                                CBSPPortal * FrontSplit,
                                CBSPPortal * BackSplit)
{
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;

    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->LeafCount    = LeafCount;
        FrontSplit->OwnerNode    = OwnerNode;
        FrontSplit->LeafOwner[0] = LeafOwner[0];
        FrontSplit->LeafOwner[1] = LeafOwner[1];

    } // End If

    if (BackSplit)
    {
        BackSplit->LeafCount     = LeafCount;
        BackSplit->OwnerNode     = OwnerNode;
        BackSplit->LeafOwner[0]  = LeafOwner[0];
        BackSplit->LeafOwner[1]  = LeafOwner[1];
    } // End If

    // Success
    return BC_OK;
}
```

Notice that we can pass FrontSplit and BackSplit into the CPolygon::Split method even though they are of type CBSPPortal because CBSPPortal is derived from CPolygon.

# The CBSPTree Class - Updated

The CBSPTree class will have to be slightly updated in this lab project so that it can now accommodate the portal and PVS information that it must now also store. The BSP tree will now maintain a vector of all the valid CBSPPortals that were generated via the portal generation process. That is, once the portal generation module has found a valid two leaf portal, it will add its pointer the tree's master portal array.

The CBSPTree class will also need to have some mechanism of storing the final PVS data array that will be generated by the PVS module. As discussed in the text book, the PVS data whether compressed or uncompressed will be represented as a byte array and therefore, the BSP tree will now have an unsigned char pointer that points to this block of data. It will also need a member variable to the store the size of the PVS data block and a Boolean specifying whether the data has been compressed. The compression

technique used is called ‘zero run length encoding’ which compresses runs of zero bytes up to 255 bytes in length into two bytes. This compression technique is discussed in the text book and lecture.

The CBSPTree class is declared in the file, CBSPTree.h. We will not show the whole class declaration here as it is getting rather large and we have only added four new members. Therefore, below we show just the members that have been added in this lab project to facilitate the storage of the portal and PVS data that will be generated by the CProcessPRT and CProcessPVS modules respectively.

***Excerpt from CBSPTree.h***

```
class CBSPTree
{
public:

    // Public Variables for This Class.
    UCHAR          *m_pPVSData;             // PVS Data set (array)
    unsigned long   m_lPVSDataSize;         // Size of the PVS data set
    bool            m_bPVSCompressed;       // Is the PVS data compressed

private:
    // Private Functions for This Class.
    vectorBSPPortal m_vpPortals;            // Portals built by the CProcessPRT compiler.
};
```

These new member variables are described below.

**UCHAR *m_pPVSData;**
After the PVS calculation module has generated a PVS for every leaf in the tree and merged them together into a single array, this pointer will be assigned to point at that master PVS data block. This is the PVS data that will be saved out to file and utilized by the run time component.

This member pointer is assigned to the PVS data by the CProcessPVS module via a call to a new CBSPTree member function called SetPVSData. This function will be passed a UCHAR pointer to the PVS data, the size of the PVS data array and a Boolean describing whether the data is in compressed format. We will look at the code to the SetPVSData method in a moment.

**unsigned long m_lPVSDataSize;**
This member will be assigned its value via the CBSPTree::SetPVSData method which will be invoked by the CProcessPVS module after the PVS has been calculated. It will describe the size of the UCHAR array of PVS data stored in the above member.

**bool m_bPVSCompressed;**
This member will be assigned a value of true or false by the CBSPTree::SetPVSData method which is invoked from the CProcessPVS module. It describes whether the PVS data array is in compressed (ZRLE) format or whether it has been stored as an uncompressed bit set. Obviously, if compiler options have been set such that the data is not compressed, the run time component will need to know this so that it iterates through the PVS data at render time in the correct manner.

**vectorBSPPortal m_vpPortals;**
This is an STL vector that will be used to store pointers to all the valid CBSPPortal structures generated by the portal generation module. The CBSPTree interface also exposes a SetPortal method which the

portal generation module can use to store a portal to this vector once it has been validated as being valid two leaf portal. The portals in this array will all be two way portals which will later be used by the CProcessPVS module to clone a set of one way portals for PVS calculation.

Because we now have an array (vector) of portals stored in the BSP tree, methods will need to be added that allow us to reserve space in this vector every time we wish to add new portals to the tree. Methods will also be needed to allow us to retrieve the number of portals in this vector and as mentioned, a method will be added to allow another modules to place portal pointers in this vector. Furthermore, the CBSPTree object will also have a function that a calling module can use to allocate the memory for a new CBSPPortal. Let us have a look at these new methods now which will give us an idea of the functions that will be called by the portal generation module, to allocate a new portal, add it to the tree's portal array and retrieve information about that portal.

## IncreasePortalCount - CBSPTree

This method is called by the CProcessPRT module every time it wishes to add space for a new CBSPPortal pointer to the end of the BSP tree's portal array. So that the vector is not being continually resized for every valid portal that we find and add to the tree, we use a resizing threshold which should not be new to you. The array resize threshold is set to 100 by default and is assigned the definition BSP_ARRAY_THRESHOLD. Here is the code to the function that allows us to make sure there is at least enough room at the end of the BSP tree's portal array to add a new portal pointer.

```
bool CBSPTree::IncreasePortalCount()
{
    try
    {
        // Resize the vector if we need to
        if (m_vpPortals.size() >= (m_vpPortals.capacity() - 1))
        {
            m_vpPortals.reserve( m_vpPortals.size() + BSP_ARRAY_THRESHOLD );

        } // End If

        // Push back a NULL pointer (will already be allocated on storage)
        m_vpPortals.push_back( NULL );

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return false;

    } // End Catch

    // Success
    return true;
}
```

The first thing we do in the above code is fetch the size of the vector. This tells us how many portal pointers are currently stored in that vector. We compare this against the capacity of the vector which describes how many portals can be stored in that vector before it is considered full. As the purpose of this function is to assure that the capacity is at least 1 greater than the current size so that there is room

to add another portal pointer, a compare between the two is performed. If the size of the vector is greater or equal to the capacity then it means the vector is full and we must resize it. However, instead of simply resizing the vector by 1 to make room for the a new portal, we resize by our threshold value which by default will resize the vector making room for 100 more portal pointers. Why do we do this? Because array resizes are expensive and we do not want to be performing one for every single portal that we add. This way, we make sure that we only cause a resize every 100 portals even if that means at the end of the process we have a little unused capacity in the vector.

You can see that if the capacity is full we reserve more space so that the vector is large enough to contain its current data (size) plus the 100 (BSP_ARRAY_THRESHOLD) new elements. Notice how we push a NULL pointer on the back of the array which forces the size of array to be increased by 1. You will see why this is necessary in a moment as.

## GetPortalCount - CBSPTree

This simple function allows a calling module to inquire about how many portals are currently contained in the BSP tree's portal array. It simply returns the size of the vector.

```
unsigned long   GetPortalCount( ) const { return m_vpPortals.size(); }
```

To understand how this might be needed, imagine that we have a CBSPPortal called pMyPortal that we would like to add to the BSP tree's portal array after finding that it is a valid portal. We would first fetch the current portal count of the array as this will also describe the index of the portal we wish to add to the end like so.

```
// Get the current number of portals stored in the tree's array
int PortalIndex = pTree->GetPortalCount();

// The capacity of the array is such that there is enough room to store new portal
pTree->IncreasePortalCount();

// Store the portal at the end of the array
pTree->SetPortal ( PortalIndex , pMyPortal);
```

You will see later that this is exactly the steps taken by the portal generation module each time it wishes to add a new portal to the BSP trees portal array.

## AllocBSPPortal - CBSPTree

This function should be used by all modules that wish to allocate a new CBSPPortal. In keeping with our previous strategy we are placing all memory allocation responsibility on the BSP tree object via a series of AllocBSP…method, for all objects that are defined in its header file. This method simply allocates a new CBSPPortal structure and returns it to the caller.

```
CBSPPortal * CBSPTree::AllocBSPPortal( )
{
    CBSPPortal * NewPortal = NULL;
```

```
    try
    {
        // Allocate new portal
        NewPortal = new CBSPPortal;

        // Note : VC++ new does not throw an exception on failure (easily ;)
        if (!NewPortal) throw std::bad_alloc();

    } // End Try

    catch (...) { return NULL; }

    // Success!
    return NewPortal;
}
```

Note that this function does not add the allocated portal to the portal array in any way. It is simply a helper function that wraps the allocation and handles the throwing of an exception if an error occurs.

## SetPortal - CBSPTree

When discussing the GetPortalCount method a moment ago, we showed some example code that demonstrated how to add a new portal to the end of the tree's portal array. This protocol involved fetching the current portal count, increasing the size of the portal array by 1 and then setting the portal at the specified position. Here we see the code to the SetPortal function that is used to store the portal pointer in the BSP tree's portal array at the specified position.

The function takes two parameters. The first in the array index where the caller would like the portal to be stored in the array and the second is a pointer to the CBSPPortal structure that is to be stored in the array.

```
void CBSPTree::SetPortal ( unsigned long Index, CBSPPortal * pPortal )
{
  if (Index < m_vpPortals.size()) m_vpPortals[Index] = pPortal;
}
```

Providing that the passed index is within the current size of the array, the value of that array element is replaced with the passed pointer. You might be wondering why the passed index is compared against the size of the vector and not the capacity. To be safe, we only allow the SetPointer method to assign values to elements that are within the current size of the array even if the array has a much larger capacity. As we know that we will always be adding portals to this array one at a time and in order, this just introduces a safety net that would stop us storing portals in non-linear addresses within the array. However, this now clearly demonstrates why in the IncreasePortalCount method we pushed a NULL on the back of the array and forced the size of the array to be increased by one. Were we not to do this we would not be able to use SetPortal to add a new portal to the end of the array. By adding a NULL to the back of the list initially, we create this portal position in the array first and then fill it later when we call the SetPortal method.

## GetPortal - CBSPTree

For completeness, whenever there is a Set function there is usually a Get function that performs the reverse operation. The CBSPTree::GetPortal method accepts a single parameter describing the location of the element within the BSP tree's portal array for which the caller would like to fetch the portal pointer. This is fetched from the array/vector and returned to the caller.

```
CBSPPortal* CBSPTree::GetPortal( unsigned long Index ) const
{
   return (Index < m_vpPortals.size()) ? m_vpPortals[Index] : NULL;
}
```

If the passed index is outside the range of the current number of portals stored in the array (<size), NULL is returned.

## SetPVSData - CBSPTree

This new member function will be called by the CProcessPVS module to store the compiled PVS data and accompanying information in the BSP tree. The first parameter to this function is where the unsigned char array of PVS data for the entire tree will be passed. The second parameter will describe the number of bytes in this array and the third parameter will describe whether compression was enabled for the CProcessPVS module. This information will be copied and stored in the tree's member variables.

The first thing the function does is delete any PVS data that the tree may already be pointing to with its m_pPVSData pointer as this will now be used to allocate a new block to contain a copy of the passed PVS data. A new byte array is allocated of the correct size (described by the second parameter) and is pointed to by the tree's m_pPVSData pointer.

```
HRESULT CBSPTree::SetPVSData( UCHAR PVSData[], unsigned long PVSSize, bool PVSCompressed )
{
    // Release any previous data
    if (m_pPVSData) delete[] m_pPVSData;

    try
    {
        // Allocate the PVS Set
        m_pPVSData = new UCHAR[ PVSSize ];
        if (!m_pPVSData) throw std::bad_alloc(); // VC++ Compat

        // Copy over the data
        memcpy( m_pPVSData, PVSData, PVSSize );

    } // End Try Block

    catch ( std::bad_alloc )
    {
        return BCERR_OUTOFMEMORY;

    } // End Catch Block

    // Store Values
    m_lPVSDataSize      = PVSSize;
```

```
    m_bPVSCompressed    = PVSCompressed;

    // Success
    return BC_OK;
}
```

After the new array has been allocated the PVS data is copied over from the passed array into the tree's m_pPVSData array. We also copy over the size and compression status of the data into the m_lPVSDataSize and m_bPVSCompressed member variables.

After this function has been called by the CProcessPVS module, the BSP tree will contain all relevant PVS information. This new BSP tree information will also be written out to file when the scene is saved.

That covers all the changes to the CBSPTree class so we will now look at some minor modifications that have been made to the CBSPLeaf class.

# The CBSPLeaf Class - Updated

With the introduction of the portal generation and PVS calculation process in this lab project, two new member variables will be added to our leaf structure. Each leaf will now need to store a ULONG array of indices into the tree's portal array describing the portals in that array that reside in that leaf. You will recall from the accompanying text book that in addition to each portal storing the indices of the leaves in which it belongs, each leaf will store the indices of the portals (which index in to the BSP tree's portal array/vector) that reside in that leaf. We also discussed in the text book how because of the fact that the PVS data for every leaf will be combined into a single PVS data block when stored in the BSP tree (as we have seen), each leaf will need to store the index of the BYTE in the tree's m_pPVSData array where its PVS data set begins. Here is the updated class declaration for CBSPLeaf contained in the file, CBSPTree.h

***Excerpt from CBSPTree.h***

```
class CBSPLeaf
{
public:

    // Constructors & Destructors
             CBSPLeaf( );
    virtual ~CBSPLeaf( );

    // Public Functions for This Class
    bool            BuildFaceIndices( CBSPFace * pFaceList );
    bool            AddPortal( unsigned long PortalIndex );

    // Public Variables for This Class
    std::vector<long>   FaceIndices;    // Indices to faces in this leaf

    std::vector<long>   PortalIndices;  // Indices to portals in this leaf
    unsigned long       PVSIndex;       // Index into the master PVS array

    CBounds3            Bounds;         // Leaf Bounding Box
};
```

As you can see, there is also a new method called ‘AddPortal’ which is a simple helper function that allows the caller (in this application the portal generation module) to add a portal index to the leaf’s PortalIndices array. Let us have a look at those two new member variables first.

**std::vector<long> PortalIndices;**
This is an array/vector of portal indices describing the portals that reside in the leaf. Each element in this vector is an index into the CBSPTree::m_vpPortals vector. This array will be filled during the portal generation process.

**unsigned long PVSIndex;**
This single unsigned long member describes a byte offset into the CBSPTree::m_pPVSData where the leaf’s visibility bits begin. This is calculated and stored in each leaf during the PVS calculation process.

## Constructor - CBSPLeaf

The leaf constructor now simply sets the PVSIndex of the leaf to -1 initially indicating that either no PVS data exists for the tree or that it is has not yet been calculated.

```
CBSPLeaf::CBSPLeaf()
{
    // Initialise anything we need
    PVSIndex        = -1;
}
```

## AddPortal - CBSPLeaf

The CBSPLeaf::AddPortal method is called to add a portal index to the leaf’s PortalIndices array. This method is called during the portal generation process when a portal is found to exist in a leaf.

The function uses the same BSP_ARRAY_THRESHOLD strategy to minimize the number of array capacity resizes that must be performed during the portal generation process. As you can see in the following code, if the vector size reaches the vector capacity then the capacity of the vector is resized to make room for N more indices (were N is the current resize threshold). The passed portal index is then added to the end of the array.

```
bool CBSPLeaf::AddPortal( unsigned long PortalIndex )
{
    try
    {
        // Resize the vector if we need to
        if (PortalIndices.size() >= (PortalIndices.capacity() - 1))
        {
            PortalIndices.reserve( PortalIndices.size() + BSP_ARRAY_THRESHOLD );
        } // End If

        // Finally add this portal index to the list
```

```
        PortalIndices.push_back( PortalIndex );

    } // End Try Block

    catch (...)
    {
        // Clean up and bail
        PortalIndices.clear();
        return false;
    } // End Catch

    // Success
    return true;
}
```

We have finally covered all the changes to the BSP tree and the CCompiler class and are now ready to start looking at the source code to the CProcessPRT class. This is the module whose 'Process' function is used to generate the portals for the BSP tree.

# The Portal Generator Module

The portal generation module is the first of the two new modules we will introduce in this lab project. It is vitally important that you have read the accompanying text book and especially the section that pertains to portal generation before continuing with this section. The portal generation code is highly recursive and hard to follow if you have not achieved a grasp of the theory. The theory will not be explained in this work book and it is assumed that you have at least a basic understanding of the portal generation algorithm we are using when viewing this section.

As we have discussed, the portal generation module is contained in a class called CProcessPRT. That is, this module contains all the functions that CCompiler will call to generate the portals for the BSP tree. We saw earlier, that the portal generation process is invoked from a function in CCompiler called PerformPRT which is called from CCompiler::CompileScene should portal generation be enabled for the current compile. The PerformPRT method calls a handful member functions of the CProcessPRT object that are common to all our modules. These include such trivial tasks as informing the module of the BSP tree that is being used, the logger object that error/status reports should be sent to and informing the module of the parent CCompiler object that is invoking the process. This information is all stored inside the CProcessPRT module prior to the CProcessPRT::Process method being called. This same strategy is common across all the modules. That is, for each module, we set up some of its member variables prior to calling its Process method. It is the Process method of each module that kick starts the actual process. In the case of the CProcessPRT module, it is the Process method that will generate all the portals for the tree. On function return, all portals will have been compiled and stored in the BSP tree's portal array. All leaves in the tree will contain an array of portal indices that describe the portals that were found to reside in those leaves. Finally, each portal will contain a 2 element array of leaf indices describing the two leaves that each portal forms the doorway between. We will now examine the code to this module.

# The CProcessPRT Class

The CProcessPRT class declaration is contained in the file ProcessPRT.h and is also shown below. The public interface of this class should be instantly familiar from other modules. It comprises of four methods that are exposed by other modules. The SetOptions, SetLogger and SetParent methods are common to all modules and allow the module to be configured prior to the Process method being invoked. The options structure, logging object pointer and CCompiler parent pointer passed into these methods are all stored in private members variables (you can see that they are inline functions). The private member variables are also the same as the other modules with the exception that a PRTOPTIONS structure is used to contain the portal compilation options.

***Excerpt from ProcessPRT.h***

```
class CProcessPRT
{
public:
    // Constructors & Destructors for This Class.
             CProcessPRT();
    virtual ~CProcessPRT();

    // Public Functions for This Class.
    HRESULT         Process( CBSPTree * pTree );
    void            SetOptions( const PRTOPTIONS& Options ) { m_OptionSet = Options; }
    void            SetLogger ( ILogger * pLogger )         { m_pLogger = pLogger; }
    void            SetParent ( CCompiler * pParent )       { m_pParent = pParent; }

private:
    // Private Functions for This Class.
    CBSPPortal     *ClipPortal          ( unsigned long Node, CBSPPortal * pPortal );
    bool            FindLeaf            ( unsigned long Leaf, unsigned long Node );
    unsigned long   ClassifyLeaf        ( unsigned long Leaf, unsigned long Node );
    HRESULT         AddPortals          ( CBSPPortal * PortalList );

    // Private Variables for This Class.
    PRTOPTIONS      m_OptionSet;        // The option set for portal Compilation.
    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    CCompiler      *m_pParent;          // Parent Compiler Pointer
    CBSPTree       *m_pTree;            // The tree used to compile the portal set.

};
```

The Process method is also no stranger to us and is the method that all modules expose to actually carry out their task. This class also has four private member functions which will be called by the public Process method to carry out its task of generating the portals and storing them in the tree.

## Process - CProcessPRT

This method invoked by CCompiler to perform the portal generation. With the help of four of the class's private methods, this function is responsible for the entire process tasked to this module. That is, when this function has completed, the two way portals will all have been generated and stored in the BSP tree's portal array. Each portal in this array will also contain the indices of the two leaves in which they reside and each leaf in the BSP tree will contain an array of portal that live in that leaf.

**Note:** Recall from text book that while a single portal can never exist in more than two leaves, a single leaf may have many portals that reside within it.

The function takes a single parameter when called from the CCompiler object. It is passed a pointer to the BSP tree which is to have its portals generated. The passed BSP tree pointer is copied into the module's member variable (m_pTree) so that we have access to the BSP tree throughout all its functions. We then output information to the logging object via its LogWrite function displaying the message that portal compilation is about to commence. We also set a rewind marker and progress range within the logger so that every time the progress indicator is updated, we can return to the cursor position set by the rewind marker and overwrite the old progress value with the current one in the loggers output window. The initial progress value is set to zero and the range of the progress indicator is set to the number of nodes in the tree. Therefore, each time we process a node and (potentially) generate a portal for it, we can increase the current progress and have the logging object return to the rewind marker (the cursor position in the output channel where the first digit of the progress percentage will be displayed) and will update the current progress percentage value.

```
HRESULT CProcessPRT::Process( CBSPTree * pTree )
{
    HRESULT      ErrCode;
    CBounds3     PortalBounds;
    CBSPPortal * InitialPortal = NULL;
    CBSPNode   * CurrentNode   = NULL;
    CBSPNode   * RootNode      = NULL;
    CPlane3    * NodePlane     = NULL;
    CBSPPortal * PortalList    = NULL;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Store tree for compilation
    m_pTree = pTree;

    try
    {
        // *************************
        // * Write Log Information *
        // *************************
        if ( m_pLogger )
        {
            m_pLogger->LogWrite( LOG_PRT,
                                 0,
                                 true,
                                 _T("Compiling scene portal information \t\t- " ) );

            m_pLogger->SetRewindMarker( LOG_PRT );
            m_pLogger->LogWrite( LOG_PRT, 0, false, _T("0%%" ) );
            m_pLogger->SetProgressRange( pTree->GetNodeCount() );
            m_pLogger->SetProgressValue( 0 );
        }
        // *************************
        // *    End of Logging     *
        // *************************
```

In the next section of the code we first test that we can retrieve the root node (node index 0) from the node array and if not then we return error (safety incase we are trying to compile portal data for a BSP

tree object that has not yet compiled its data). We fetch the root node because its bounding box will be used to create the initial portal on each node plane. We then loop through each node in the node array.

Let us now examine the contents of this node loop.

```
        // Store required values ready for use.
        if (!(RootNode = m_pTree->GetNode(0))) throw BCERR_BSP_INVALIDTREEDATA;

        // Create a portal for each node
        for (unsigned long i = 0; i < pTree->GetNodeCount(); i++)
        {
            // Update progress
            if ( m_pParent && !m_pParent->TestCompilerState()) return BC_CANCELLED;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );
```

The first thing we do inside this loop is test that the parent compiler has not been put into a cancelled state by the user. If so, we simple return BC_CANCELLED because the user has obviously aborted the process mid compile. Provided this is not the case however, you can see that we instruct the logging object to update its progress as we are about to process another node.

In the next section of code we fetch the current node structure we are processing from the BSP tree's node array. We store a pointer to this node structure in the local node pointer **CurrentNode**. We then fetch from this structure the index of the node plane stored at that node so that we can fetch the node's plane from the BSP tree's plane array and store its pointer in the local variable **NodePlane.**

```
            // Store required values ready for use.
            if (!(CurrentNode = m_pTree->GetNode(i))) throw BCERR_BSP_INVALIDTREEDATA;

            if (!(NodePlane   = m_pTree->GetPlane(CurrentNode->Plane)))
                                                    throw BCERR_BSP_INVALIDTREEDATA;
```

Now that we have the node structure and the plane structure of the node we are currently processing, let us first test whether a portal could possibly exist on this node. As we know, a portal can only exist on a node plane if that node has **not** got solid space behind it. If it has then this node does not have leaves in both its half spaces and therefore, no portal generated on this node could ever bridge two leaves. When this is the case we simply skip processing this node any further and continue on to the next iteration of the loop and the next node waiting to be processed.

```
            // Skip any that have solid space behind them
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) continue;
```

If we get this far without skipping to the next iteration then it means the current node we are processing has leaves on both sides and it stands a very good chance of generating a real portals. Of course, we do not know this for sure yet, but we certainly know that we will have to create a new portal on the node plane that is as large as the root node's bounding box and will have to send this portal down the tree clipping away any fragments that exist in solid space. If anything survives, then we do have a portal or multiple portal fragments that can be added to the BSP tree's portal array.

The first step is to allocate a new CBSPPortal structure using the CBSPTree::AllocBSPPortal function which we described earlier. This portal will be initially empty but we want it to represent a quad that is

located in the current node's plane and is large enough to fill the root node's bounding box. Fortunately, we have already written the CPolygon::GenerateFromPlane method that will construct such a portal given the node plane and the root nodes bounding box. Therefore, in the next section of code, you can see that after we allocate a new portal structure and retrieve the root node's bounding box, we then pass this information into the GenerateFromPlane method to generate the initial portal on that plane. We also store in the portal the index of the node on which it was initially generated in its OwnerNode member.

```
            // Allocate a new initial portal for clipping
            if (!(InitialPortal = CBSPTree::AllocBSPPortal())) throw BCERR_OUTOFMEMORY;

            // Initial Portal should fill root node
            PortalBounds = RootNode->Bounds;

            // Generate the portal polygon for the current node
            InitialPortal->GenerateFromPlane( *NodePlane, PortalBounds );
            InitialPortal->OwnerNode = i;
```

At this point we are ready to drop that initial portal in at the root node of the tree and clip it to the solid space of the tree as it makes its way down to the leaf nodes. The ClipPortal method is used for this. It is a recursive function that will call itself repeatedly until either the initial portal has been completely clipped away (in which case PortalList will be assign NULL on function return) or until it has correctly calculated the valid portal fragments in which case, they will be returned in a linked list. PortalList will point to the head of this valid portal list on function return. As the first parameter to the ClipPortal method we pass in the index of the node we would like to start clipping from which will be the root node (node zero). As the second parameter we pass our initial portal that is to be passed through the tree and clipped.

```
            // Clip the portal and obtain a list of all fragments
            PortalList = ClipPortal(0, InitialPortal );

            // Clear the initial portal value, we no longer own this
            InitialPortal = NULL;
```

Notice in the above code how when the ClipPortal method returns, we simply set the InitialPortal pointer to NULL instead of releasing it. That is because this portal will have been clipped and deleted by the ClipPortal method as it is passed through the tree and split into child fragments. As soon as we pass the initial portal into the ClipPortal method it is the responsibility of the ClipPortal method to clean up its memory when it gets split.

At this point, if PortalList is not NULL then it contains a list of one or more valid portals that have been generated on the current node plane. If this is the case these portals should be added to the BSP tree's portal array. The CProcessPRT::AddPortals method is invoked to perform this task. Contrary to the way we normally do things, we will look at the simple AddPortals method prior to examining the ClipPortal code. This will show us how the portal information returned from ClipPortal gets added to the tree and the leaves of that tree first.

```
            // Add any valid fragments to the final portal list
            if (PortalList)
            {
                if (FAILED(ErrCode = AddPortals( PortalList ))) throw ErrCode;
            } // End If PortalList
```

```
        } // Next Node

    } // End Try

    catch ( HRESULT& Error )
    {
        // If we dropped here, something failed
        if ( InitialPortal )      delete InitialPortal;
        if ( m_pLogger )          m_pLogger->ProgressFailure( LOG_PRT );
        return Error;

    } // End Catch

    // Success
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PRT );
    return BC_OK;

}
```

We can see that after the AddPortals method has been called we see the closing brace to the node loop such that at the bottom of the function, every portal in the level will have been created and added to the tree's portal list. Before returning success, we output to the logging object that the mission has been a success. When the function returns, the portal generation process is over and all portals have been created and stored in the tree.

## AddPortals - CProcessPRT

The AddPortals method is called from the Process method to add a list of valid portals returned from the ClipPortal method for a given node to the BSP tree's node array. The function is passed a CBSPPortal pointer that points to the head of this list. First we set up a loop to iterate through every portal in the list. The Iterator local pointer is used to step through the elements in the list.

```
HRESULT CProcessPRT::AddPortals( CBSPPortal * PortalList )
{
    unsigned long PortalIndex = 0;
    CBSPPortal  * Iterator;
    CBSPLeaf * Leaf = NULL;

    // Validate
    if (!PortalList) return BCERR_INVALIDPARAMS;

    // Iterate through the list, obtaining valid portals
    Iterator = PortalList;
    while ( Iterator != NULL )
    {
```

Inside this loop we first fetch the current portal count from the BSP tree as this will tell us the location within the BSP tree's master portal array where the next portal should be placed (at the end of the currently stored portals).

```
        // Store new portal index
        PortalIndex = m_pTree->GetPortalCount();
```

As every portal will at this point store in its LeafOwner array the indices of the two leaves in which it was found to reside during the ClipPortal function, we next loop through each of these elements and store the index of this portal (in the BSP tree's portal array) in the PortalIndices array of each leaf.

```
        // Add this portal to each leaf
        for ( int i = 0; i < 2; i++ )
        {
            Leaf = m_pTree->GetLeaf( Iterator->LeafOwner[i] );
            if (!Leaf) return BCERR_BSP_INVALIDTREEDATA;
            Leaf->AddPortal( PortalIndex );

        } // Next Leaf
```

As you can see in the above code, in each iteration of the loop we fetch the leaf index from the portal's LeafOwner array and then use that to retrieve the relevant leaf structure from the tree. Once we have a pointer to the leaf in which this portal should have its index stored, we then call the CBSPLeaf::AddPortal method (which we looked at earlier) which will add the passed index to the leaf's PortalIndices array. Remember that although we have not yet added the current portal being processed in the passed list to the BSP tree's portal array, PortalIndex describes the location of where it will be stored as this describes the current number of portals in the list prior to this portal being added.

With the portal index now stored in the two leaves in which it was found to reside, we next instruct the BSP tree to make sure there is enough space in its portal array to add this new portal pointer.

```
        // We are adding a new portal
        m_pTree->IncreasePortalCount();
```

We then finish processing the current portal by using the CBSPTree::SetPortal method to store the current portal's pointer at that index in the tree's master portal array.

```
        // Set the portal
        m_pTree->SetPortal( PortalIndex, Iterator );
```

We then assign the current portal's NextPortal member to Iterator so that in the next iteration of the loop, if NextPortal is not NULL, Iterator will point to the next portal in the passed list to be added to the tree's portal array.

```
        // Move onto next portal
        Iterator = Iterator->NextPortal;

    } // End While

    return BC_OK;
}
```

When this function returns, every portal in the passed list will have been added to the BSP tree's portal array and the leaves in which these portals reside will have had the portal indices added to their PortalIndices array.

## ClipPortal - CProcessPRT

As discussed in the accompanying text book, the ClipPortal method really is the portal generation engine. It is the function that is called from the Process method and passed an initial portal that is to be clipped to the tree. When the function is first called it will visit the root node and will then recursively call itself until the portal has either been complete deleted, or until it has a list of valid portal fragments to return. Valid portals are fragments of the initial portal passed in the root that ended up in empty space and were found to reside in two leaves. This is a rather huge function which was explained in detail in the text book. As this version of the function is almost identical we will make our way through it quite quickly so that you can see the version of the function that is used by our compiler.

The function is passed a node index (which will be the root node the first time it is called from the Process method) and a portal. The first time this function is called this portal will be the initial portal that was created on the node inside the Process method, for nodes further down the tree, this portal may be a fragment of that initial portal.

```
CBSPPortal * CProcessPRT::ClipPortal( unsigned long Node, CBSPPortal * pPortal )
{
    // 52 Bytes including Parameter list (based on __thiscall declaration)
    CBSPPortal   * PortalList     = NULL, *FrontPortalList = NULL;
    CBSPPortal   * BackPortalList = NULL, *Iterator        = NULL;
    CBSPPortal   * FrontSplit     = NULL, *BackSplit       = NULL;
    CBSPNode     * CurrentNode    = NULL;
    CPlane3      * CurrentPlane   = NULL;
    unsigned long  OwnerPos, LeafIndex;

    // Validate Requirements
    if (!pPortal || !m_pTree) throw BCERR_INVALIDPARAMS;

    // Store node for quick access
    if (!(CurrentNode  = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
    if (!(CurrentPlane = m_pTree->GetPlane(CurrentNode->Plane)))
                                                    throw BCERR_BSP_INVALIDTREEDATA;
```

As we can see in the above section of code, we first use the passed node index to fetch the node structure from the BSP tree. This is the node that we are currently visiting with the passed portal/fragment. We then use the node's Plane index to fetch the node's plane structure from the BSP tree's master plane array.

The rest of the function is essentially just a switch statement which deals with the result of classifying the portal against the plane. The following code classifies the vertices of the polygon against the node plane. We use the CPlane3::ClassifyPoly function for this task. The function will return either CLASSIFY_FRONT, CLASSIFY_BACK, CLASSIFY_ONPLANE or CLASSIFY_SPANNING.

```
    // Classify the portal against this nodes plane
     switch (CurrentPlane->ClassifyPoly(      pPortal->Vertices,
                                              pPortal->VertexCount,
                                              sizeof(CVertex))    )
    {
```

The front and back cases are small and simple to deal with but the spanning and on plane cases are considerably more complex. We will look at the CLASSIFY_ONPLANE case first.

```
        case CLASSIFY_ONPLANE:

            // The Portal has to be sent down Both sides of the tree and tracked.
            // Send it down front first but DO NOT delete any bits that end up in solid
            // space, just ignore them.
            if (CurrentNode->Front < 0 )
            {
                        // The Front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER)
                {
                    // This portal is added straight to the front list
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    FrontPortalList              = pPortal;
                    pPortal->LeafCount++;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End If no leaf found

            } // End if child leaf
            else
            {
                    // Send the Portal Down the Front List and get returned
                    // a list of PortalFragments that survived the Front Tree
                    FrontPortalList = ClipPortal(CurrentNode->Front, pPortal);

             } // End If child node
```

The above code shows the first section of dealing with the on plane case. When a portal is on plane we need to send that portal down the front tree so that it can be clipped to the front tree of the node. This will return a linked list of any portal fragments that survive the front tree. Each portal in this list should then be passed down and clipped to the back tree of the node. Any fragment of the portal passed into this function that survives both the front and back trees of the node can then be compiled into a linked list and returned from the function.

In the above section of code we show the portion of the on plane case that deals with sending the portal down the front of the node first. If the node's Front member contains a negative number then we know that there is a leaf to the front of this node (empty space) and as such, the portal should record the index of this leaf in the portal. This is one of the leaves the portal has been found to reside in. Therefore, when this is the case we add 1 to the node's Front value and ABS it so that we have an index into the BSP tree's leaf array of the leaf in which the portal has landed. We then call the ClassifyLeaf function which will return either NO_OWNER, FRONT_OWNER or BACK_OWNER indicating whether this leaf is located in the front or back half space of the node. If the function returns NO_OWNER then it means that the leaf the portal has landed in does not exist beneath the portal's owner node in the tree and therefore this is a rogue portal (rogue portals are discussed in the text book). When this is the case we simply delete the portal as this portal is not valid for the current node being generated. However, if

FRONT_OWNER or BACK_OWNER is returned then we store the leaf index in the portal's LeafOwner array. Notice that FRONT_OWNER and BACK_OWNER are also used as the array indexes so that the leaf that exists in the front space of a portal will always be contained in the first element of the portal's LeafOwner array and the second element will always contain the index of the leaf contained in the back half space of the node. You can also see in the above code that if the portal does land in a leaf, we increase the portal's LeafCount member to reflect the fact that we have just added a leaf index to the portal's leaf array. We also assign the FrontPortalList local variable to point at this portal which will be used in a moment. We also make sure that the portal's next pointer is set to NULL.

Finally notice at the bottom of the above code, how if a leaf does not exist down the front of this node, but a child node exists instead, the ClipPortal function is called recursively to send the portal down the front tree of the current node. This function will either return a linked list of all the fragments of this portal that survived the front tree, the head of which is assigned to the FrontPortalList local variable, or will return NULL if none of the portal survived being clipped to the front tree.

At this point, FrontPortalList either points to the single portal that made it into a leaf to the front of this node, a list of fragments that survived the portal being clipped to the front tree of the node, or NULL if either the portal made it into a leaf that was not beneath the owner node in the tree (rogue portal fragment) or if the portal was clipped to the front tree and was found to be contained completely in solid space.

In the next section of code we see that if FrontPortalList equals NULL then there is nothing more to do at this node. The portal passed into this node has been completely deleted so we return NULL.

```
            // If nothing survived return here.
            if (FrontPortalList == NULL) return NULL;
```

However, if there are portals in our front list then we know each will have to be clipped to the back tree of the node next. If the node has no back child (solid space behind it) then we simply return the front list of portal fragments.

```
            //// If the back is solid, just return the front list
            if ( CurrentNode->Back == BSP_SOLID_LEAF ) return FrontPortalList;
```

Now we will loop through each portal in the front portal list and will send each one in the list into the ClipPortal function to clip it to the front tree of the node. Each time we call the ClipPortal function to send the current front list portal being processed down the back tree, we will take the returned list of portals and add them to a larger list that is being compiled. This larger list will contain, at the end of the next section of the code, any fragments that survived both the front and back trees of the node.

```
            // Loop through each front list fragment and send it down the back branch
            pPortal = FrontPortalList;

              while ( pPortal != NULL )
              {
                 CBSPPortal * NextPortal = pPortal->NextPortal;
                 BackPortalList          = NULL;
```

At the head of the loop we cache the next portal in the front portal list so that when the current portal we are processing gets sent down the back tree and potentially deleted, we do not loose access to the next portal in the front portal list to be processed. The variable BackPortalList will be used to retrieve any fragments of the current front portal fragment that is sent down the back tree of the node.

> **Note**: Remember that the current portal being processed in this loop (pPortal) is a portal fragment from the front portal list that is now about to be clipped to the back tree.

First we test the Back member of the current node and if found to be negative it means an empty space leaf exists to the back of this node. Here we are adding support for empty back leaves that can occur in very special cases which will be discussed later in the series. As discussed in the text book however, a level provided for solid BSP compilation, will not ever contain populated back leaves. Still, we will add support for populated back leaves now.

As we did in the case of an immediate front leaf, if a leaf does exist to the back of this node, it means the portal has landed in this leaf. As such, we convert the node's Back member into a valid leaf index and fetch the appropriate leaf structure from the BSP tree's leaf array. We then classify this leaf against the portal's owner node to determine in which half space of the portal this leaf exists. The leaf indices are then stored in the portal depending on the outcome.

```
                // Empty leaf behind?
                if ( CurrentNode->Back < 0 )
                {
                    // The back is a Leaf, determine which side of the node it fell
                    LeafIndex   = abs(CurrentNode->Back + 1);
                    OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                    // Found the leaf below?
                    if ( OwnerPos != NO_OWNER )
                    {
                        // Attach it to the back list
                        pPortal->LeafOwner[OwnerPos] = LeafIndex;
                        pPortal->NextPortal          = BackPortalList;
                        BackPortalList               = pPortal;
                        pPortal->LeafCount++;

                    } // End if leaf found
                    else
                    {
                        // Delete the portal, but continue to the next fragment
                        delete pPortal;
                        continue;

                    } // End If no leaf found

                } // End if child leaf

                else

                {
                    // Send the Portal Down the back and get returned a list of
                    // PortalFragments that survived the Front Tree
                    BackPortalList      = ClipPortal(CurrentNode->Back, pPortal);

                } // End If child node
```

Notice that if a leaf does not exist immediately to the back of this node it means a child node must exist there instead. When this is the case we send the portal down the back of the node to clip it to the node's back tree. Any surviving fragments are returned in a linked list, the head of which is assigned to the BackPortalList pointer.

At this point, BackPortalList contains only the list of fragments for a single fragment that survived the front tree of the node and as discussed, we must collect all the BackPortalList's generated from each portal in FrontPortalList and stitch them together into a single linked list that can be returned from the function. The PortalList local variable will be used to point at the head of this combined list which will be returned from the function.

In the next section of code we can see that assuming that BackPortalList is not null, we must add them to the current list we have compiled so far. We do this by first iterating to the tail of the BackPortalList.

```
                // Anything in the back list?
                if (BackPortalList != NULL)
                {
                    // Iterate to the end to get the last item in the back list
                    Iterator = BackPortalList;

                    while ( Iterator->NextPortal != NULL) Iterator = Iterator->NextPortal;
```

At this point Iterator will point to the last element in BackPortalList. We now assign the NextPortal member of this final portal in the back portal list to point at 'PortalList' which currently points to the head of the list of portal fragments we have collected so far. What we are doing is adding the back portal list to the front of the portal list of all the fragments we have collected so far. Here is the remaining code of the on plane case.

```
                    // Attach the last fragment to the first fragment
                    // from the previous iteration.
                    Iterator->NextPortal = PortalList;

                    // Portal List now points at the current complete
                    // list of fragments collected so far
                    PortalList = BackPortalList;

                } // End if BackPortalList is not empty


                // Move on to next portal
                pPortal = NextPortal;

            } // End While Portal != NULL

            // Return the full list
            return PortalList;
```

As you can see after we have assigned the Iterator to point at PortalList (the current head of the list of all fragments we have collected so far), we the reassign PortalList to point at BackPortalList so that it now points to the complete list of fragments we have collected so far, including those contained in BackPortalList. Finally, we can see that at the bottom of main loop that iterates through each portal in FrontPortalList, we assign pPortal to point at NextPortal. Next Portal is where we stored a pointer to the

next portal in FrontPortalList that will need to be clipped to the back tree of the node in the next iteration.

Outside the loop and in the very last line of code shown above, PortalList will contain all the fragments of the portal passed into the function that survived both the front and back trees of the current node being visited. This linked list is returned from the function.

With the on plane case covered, we will next look at what happens if the portal passed into the function is found to be contained entirely in the front half space of the current node being visited.

If there is a leaf immediately attached to the front of this node then it means the portal has made it into a leaf. When this is the case we see that familiar piece of code that fetches the leaf from the BSP tree's leaf array and classifies it against the owner node of the portal. Depending on which side of the owner node's plane the leaf is found to reside, the leaf index is recorded in the appropriate position in the portals LeafOwner array and the portals leaf count is increased so that we know how many leaves this portal has been found to exist in at this point. Below we show the entire CLASSIFY_INFRONT case.

```
            case CLASSIFY_INFRONT:

             // Either send it down the front tree or add it to the portal
             // list because it has come out in Empty Space
            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER )
                {
                    // This is just returned straight away, it's in an empty leaf
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    pPortal->LeafCount++;
                    return pPortal;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End if leaf not found

            } // End if child leaf
            else
            {
                // Pass down the front
                PortalList = ClipPortal(CurrentNode->Front, pPortal);
                return PortalList;

            } // End If child node

            break;
```

Above we can see that if a leaf does not exist immediately to the front of this node it means a child node exists and as such, the portal is passed down the front tree of the node and clipped to any solid space that may exist there. The ClipPortal method will return a list of one or more fragments of this portal that survived being clipped to the front tree which are then returned from the function to the parent instance of the function.

The CLASSIFY_BEHIND case is almost the same except with a very important difference. If solid space exists to the back of the node then the portal has to be deleted and NULL returned. It is this case during the recursive process that is responsible for clipping away the parts of a portal that land in solid space.

```
            case CLASSIFY_BEHIND:

            // Test the contents of the back child
            if (CurrentNode->Back == BSP_SOLID_LEAF )
            {
                // Destroy the portal
                delete pPortal;
                return NULL;

            } // End if solid leaf
```

However, if solid space does not exist down the back of this node then two other conditions may be true. Either an empty space leaf exists down the back of this node in which case we store the leaf index in the portal in the normal way, or a child node exists down the back of the current node in which case the portal must be clipped to the back tree and the resulting fragment list returned as shown below.

```
            else

            if (CurrentNode->Back < 0 )
            {
                // The back is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Back + 1);
                OwnerPos    = ClassifyLeaf( LeafIndex, pPortal->OwnerNode );

                // Found the leaf below?
                if ( OwnerPos != NO_OWNER )
                {
                    // This is just returned straight away, it's in an empty leaf
                    pPortal->LeafOwner[OwnerPos] = LeafIndex;
                    pPortal->NextPortal          = NULL;
                    pPortal->LeafCount++;
                    return pPortal;

                } // End if leaf found
                else
                {
                    delete pPortal;
                    return NULL;

                } // End if leaf not found

            } // End if child leaf
            else
            {
                // Pass down the back
```

```
                PortalList = ClipPortal(CurrentNode->Back, pPortal);
                return PortalList;

            } // End If child node

            break;
```

The final case we must deal with I this function is the case where the portal is spanning the plane. When this is the case we must split the portal into two child fragments and clip the back fragment to the back tree of the node and the front fragment to the front tree of the node. Any surviving fragments from the front split portal and the back split portal are joined together into a single linked list which is then returned from the function. We will look at this case a section at a time.

```
            case CLASSIFY_SPANNING:

            // Allocate new front fragment
            if (!(FrontSplit = CBSPTree::AllocBSPPortal())) throw BCERR_OUTOFMEMORY;

            // Allocate new back fragment
            if (!(BackSplit  = CBSPTree::AllocBSPPortal())) { delete FrontSplit;
                                                              throw BCERR_OUTOFMEMORY; }

            // Portal fragment is spanning the plane, so it must be split
            if (FAILED( pPortal->Split(*CurrentPlane, FrontSplit, BackSplit)))
            {
                delete FrontSplit; delete BackSplit;
                throw BCERR_OUTOFMEMORY;
            } // End If

            // Delete the ORIGINAL portal fragment
            delete pPortal;
            pPortal = NULL;
```

As the above code shows, because we know the portal is spanning the plane it will have to be split into two children so we first allocate two new empty CBSPPortal structures. These are then passed into the parent portal's Split method along with the plane of the current node we are visiting. When the split function returns, FrontSplit will contain the fragment of the portal that exists in the front half space of the passed plane and BackSplit will contain the fragment that exists in the back half space. The original portal (pPortal) that was passed into the function can now be deleted as it has been replaced by these two splits.

Our next task is to deal with the front split first by sending it down the front tree of the node. If a leaf exists immediately to the front of the node then the front split portal obviously exists in this leaf. When this is the case the leaf index is recorded in the portal as we have seen many times before.

```
            // There is another Front NODE ?
            if (CurrentNode->Front < 0 )
            {
                // The front is a Leaf, determine which side of the node it fell
                LeafIndex   = abs(CurrentNode->Front + 1);
                OwnerPos    = ClassifyLeaf(LeafIndex, FrontSplit->OwnerNode );

                // Found the leaf?
                if ( OwnerPos != NO_OWNER)
                {
```

```
                FrontSplit->LeafOwner[OwnerPos] = LeafIndex;
                FrontSplit->NextPortal          = NULL;
                FrontPortalList                 = FrontSplit;
                FrontSplit->LeafCount++;

            } // End if leaf found
            else
            {
                delete FrontSplit;

            } // End If no leaf found

        } // End if child leaf
        else
        {
                FrontPortalList = ClipPortal(CurrentNode->Front, FrontSplit);

        } // End If child node
```

Notice in the **else** case however that if a child node exists here instead, the front split portal is clipped to the front tree of the node with any surviving fragments being assigned to the FrontPortalList local variable. Notice that even if the portal makes it into the leaf, we also assign FrontPortalList to point at it so that regardless of whether a node or a leaf exists to the front of this node, we know that FrontPortalList will point to one or more portals that have survived the front tree of the node at this point.

With the front split being clipped to the front tree dealt with, we will now send the back split portal down the back tree of the node. If there is solid space behind this node then the back split has landed in solid space and is therefore simply deleted. Alternatively, if a leaf exists to the back of this node we see that familiar code that classifies the leaf against the node and stores the leaf index in the appropriate location in the portals LeafOwner array.

```
        // There is another back NODE ?
        if ( CurrentNode->Back == BSP_SOLID_LEAF )
        {
            // We ended up in solid space
            delete BackSplit;

        } // End if solid leaf
        else if (CurrentNode->Back < 0 )
        {
            // The back is a Leaf, determine which side of the node it fell
            LeafIndex   = abs(CurrentNode->Back + 1);
            OwnerPos    = ClassifyLeaf(LeafIndex, BackSplit->OwnerNode );

            // Found the leaf?
            if ( OwnerPos != NO_OWNER)
            {
                BackSplit->LeafOwner[OwnerPos] = LeafIndex;
                BackSplit->NextPortal          = NULL;
                BackPortalList                 = BackSplit;
                BackSplit->LeafCount++;

            } // End if leaf found
            else
            {
                delete BackSplit;
```

```
            } // End If no leaf found

        } // End if child leaf
```

Finally, if a child node exists to the back of this node instead of a leaf, we clip the back split portal to the back tree of the node using the BackPortalList local pointer to point to any surviving fragments on function return.

```
    else
        {
            BackPortalList = ClipPortal(CurrentNode->Back, BackSplit);

        } // End If child node
```

Notice that even in the case where the portal makes it into a back leaf, we assign BackPortalList to point at the portal so regardless of whether or not it landed in a leaf or was clipped to the back of the tree, BackPortalList will contain any fragments of the back split portal that survived the back node of the tree.

At this point we have FrontPortalList and BackPortalList which can potentially contain the fragments of the front split portal and the back split portal that survived being sent down the front and back of the node respectively. Our final step is two join these two portal lists together into a single list before returning this combined list from the function.

The following code shows that if there are portals in FrontPortalList then we iterate through the list to get a pointer to the last portal in that list. If BackPortalList isn't NULL then we assign the NextPortal member of that last portal in FrontPortalList to point to the first portal in the back portal list which is then returned from the function.

```
        // Find the End of the front list and attach it to Back List
        if (FrontPortalList != NULL)
        {
                // There is something in the front list
                Iterator = FrontPortalList;

                while (Iterator->NextPortal != NULL) Iterator = Iterator->NextPortal;

                if (BackPortalList != NULL) Iterator->NextPortal = BackPortalList;
                    return FrontPortalList;

        } // End if front list
```

If there are no portals in the front portal list then alternatively we just return either the back portal list or NULL if no portals exist in this the back portal list either as shown below.

```
        else
        {
            // There is nothing in the front list simply return the back list
            if (BackPortalList != NULL) return BackPortalList;
            return NULL;

        } // End if no front list
```

```
            // If we got here, we are fresh out of portal fragments so simply return NULL.
            return NULL;

    } // End switch

    return NULL;
}
```

The ClipPortal method is certainly an intimidating function on first appearance although, examining the various cases in isolation has shown that this is really just a special case CSG function. Most of the code that seems to make the function look overly complex is actually trivial linked list manipulation and management code. This function is the core of the portal generation process. As we have seen, it is called by the CProcessPRT::ClipPortal method for each initial portal that is created on a node plane.

Whenever a portal is found to exist in a leaf in the above code, we record the index of that leaf in the portal. Whether the leaf index is stored in the first element of the portals LeafOwner array or the second depends on whether the leaf exists in the front or back half space of the node respectively. To make this determination the ClassifyLeaf method is used. This method also takes care of identifying rogue portal fragments if the leaf can not be found down either the front or back tree of the current portals owner node. Let us have a look at the code to this function next.

## ClassifyLeaf - CProcessPRT

When this function is called from ClipPortal it is passed the index of the leaf in which the portal fragment has been found to reside and is also passed the index of the portal's owner node. Recall that the owner node is the node for which the initial portal was created and sent into the first instance of the ClipPortal method.

The function first uses the passed node index to fetch the node structure from the BSP tree's node array.

```
unsigned long CProcessPRT::ClassifyLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // Validate Requirements
    if (!m_pTree) throw BCERR_INVALIDPARAMS;
    if (!(CurrentNode = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
```

If the node has a negative number in its Front member then it means a leaf exists down the front of the node. In this case we convert it to a valid leaf index and perform an equality test with the passed leaf index. If they are equal then we have located the passed leaf as being attached immediately to the front of the node. This means the leaf we are looking for is in the front half space of the portal so we return FRONT_OWNER.

```
    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1) == Leaf ) return FRONT_OWNER;
    }
```

If the node's Front member is non negative then it contains the index of the front child node. When this is the case we use the CProcessPRT::FindLeaf function (discussed in a moment) to traverse the front tree of the node looking for the leaf. If the function returns true then the leaf was located down the front tree of the node so we can also return FRONT_OWNER. If the function did not return true then we were unable to locate the leaf in the front tree so we will have to search the back tree of the node instead.

```
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return FRONT_OWNER;
    } // End If
```

Provided that the node's Back member is non negative it means a child node exists there so we should traverse down the back tree of the node searching for the leaf using the FindLeaf function once again. If the function returns true the leaf was located in the back tree which means this is the leaf that exists in the back space of the portal thus we return BACK_OWNER. If none of these cases are true then it means the portal fragment obviously landed in a leaf that is not beneath the portal's owner node in the tree and as such is obviously a rogue fragment that should be deleted. When this is the case we reach the bottom of the function where NO_OWNER is returned.

```
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf(Leaf, CurrentNode->Back )) return BACK_OWNER;
    } // End If

    return NO_OWNER;
}
```

When NO_OWNER is returned from this function back to the ClipPortal method, the portal fragment is deleted.

## FindLeaf - CProcessPRT

This is the recursive leaf searching function that was called from the previously discussed function to search for a given leaf down a given sub-tree. As we saw in the previous function, the first parameter to this function is where the index of the leaf we wish to search for should be passed. The second parameter is where the index of the node we would like to start searching from should be passed. As we saw in the previous function, this is the back or front child of the portal's owner node for which the leaf search is being performed.

The function first uses the passed node index to fetch the relevant node structure from the tree's node array.

```
bool CProcessPRT::FindLeaf( unsigned long Leaf, unsigned long Node )
{
    CBSPNode * CurrentNode = NULL;

    // Validate Requirements
    if (!m_pTree) throw BCERR_INVALIDPARAMS;
    if (!(CurrentNode = m_pTree->GetNode( Node ))) throw BCERR_BSP_INVALIDTREEDATA;
```

If a leaf exists down the front of this node then we test to see if that leaf has the same index as the leaf we are searching for. If so, we have found our leaf so we return true.

```
    // Check to see if the front is this leaf
    if ( CurrentNode->Front < 0 )
    {
        if ( abs(CurrentNode->Front + 1) == Leaf ) return true;
    }
```

If a leaf does not exist down the front of the current node we are visiting but a child node exists there instead, we will recur down the front tree of this node searching for the leaf.

```
    else
    {
        if (FindLeaf( Leaf, CurrentNode->Front )) return true;
    } // End If
```

If we get this far then it means the leaf could not be found down the front tree of the current node so we will search the back tree of the current node instead and return true if it is located.

```
    // Iterate down the back if it's a node
    if ( CurrentNode->Back >= 0 )
    {
        if (FindLeaf( Leaf, CurrentNode->Back )) return true;
    } // End If

    return false;
}
```

If we reach the bottom of the function without returning true then it means the leaf could not be found down the front and back tree of the current node so we return false.

# Portal Generation Conclusion

This completes our coverage of the portal generation module and as we have seen, it is implemented via a handful of functions. We have also seen that all our CCompiler object had to do was invoke the CProcessPRT::Process method to populate the BSP with portal data.

We are at the mid-way point in this lab project having implemented one of the two modules necessary to add potential visibility set calculation to our compiler application. In the next section we will discuss the implementation of the CProcessPVS class which uses the portal data now present in the tree to perform the final calculation of the PVS data. It should be noted that the portal generation module must be enabled for the PVS calculation module to work. As the PVS calculation module can not possibly perform its task with our portal data being present in the tree, the module will terminate immediately if no portal data is found to be present in the BSP tree.

# The PVS Calculator

It is crucial that you have read the PVS section in the accompanying text book before continuing with this section of the work book. In this section we will not re-cover the theory of PVS calculation and anti-penumbra generation and clipping. It is assumed you have read and understood the text book and are now interested in seeing how this technique pertains to this particular application and our application structures and classes.

Our PVS calculator is contained in the module CProcessPVS. It is this module's Process method which is called from CCompiler post portal generation. The PVS calculation module itself has all its data and structures contained in the project source files ProcessPVS.h and ProcessPVS.cpp.

In ProcessPVS.h we have a compiler define named PVS_COMPRESSDATA which can be set to zero or one to control whether the resulting PVS data should be zero run length encoded by the PVS generation module prior to being stored in the BSP tree.

```
#define PVS_COMPRESSDATA        1       // 1 = ZRLE Compress, 0 = Don't Compress
```

By default we set this to 1 and it is unlikely that you will want to change this. PVS data sets for complex levels can be quite large if no compression is used and while memory is fairly abundant on today's end user systems, we do not want to waste it considering the typical huge number of other resources that may have to be stored. Also, we discussed in the text book how zero run length encoding our data actually helps speed up the rendering of a given leaf's PVS by allowing us to skip past entire runs of non-visible leaves with a single byte increment of the PVS data pointer.

Before we examine the code to the CProcessPVS module, there are several support structures used by this process that we must first discuss. For example, we know that the two-way portals stored in the BSP tree will have to be duplicated into a number of one way portal structures that have added member variables that pertain to the PVS calculation process. We also discussed in the text book how in order to reduce memory allocation and fragmentation during the recursive clipping process, each of these portals will have the ability to share its underlying geometry with other portals.

# The CPVSPortal Class

The CPVSPortal class is the object that will be used to store the one way portals used by the PVS calculation module. For each original two-way portal stored in the BSP tree at the end of the portal generation process, two CPVSPortal objects will be created. Each of the two one-way portals will share and represent the same geometry of the two-way portal from which they were cloned but each will point into an opposing half space. In the text book we had a section devoted to the need for us having a strict portal flow during the PVS clipping process and that is what these one-way portals provide us.

Each one-way portal which is temporarily generated for the PVS calculation module stores much more additional information than a normal two-way portal it was cloned from. We have to store in that portal the index of its neighbor leaf. The neighbor leaf is the leaf in which the one-way portal does not reside but has its normal facing into. That is to say, a one-way portal's visibility flows out from its owner leaf

into the neighbor leaf. We will also need to store in the one-way portal the visibility bit sets that are being compiled for that portal. Recall from the text book, that the PVS for the tree is actually generated by first calculating the PVS of each portal. Therefore, each of these portals will need to have a member where we can store its PVS. Each leaf's PVS is then calculated by simply accumulating the visibility information of each portal that resides within that leaf.

The CPVSPortal structure is shown below followed by a description of its members.

***Excerpt from CPVSPortal.h***

```
class CPVSPortal
{
public:
    // Constructors / Destructors for this Class
             CPVSPortal( );
    virtual ~CPVSPortal( );

    // Public Variables for This Class
    UCHAR             Status;               // The compilation status of this portal
    UCHAR             Side;                 // Which direction does this portal point
    long              Plane;                // The plane on which this portal lies
    long              NeighbourLeaf;        // The leaf into which this portal points
    long              PossibleVisCount;     // The size of the PossibleVis array
    UCHAR            *PossibleVis;          // "Possible" visibility information
    UCHAR            *ActualVis;            // "Actual" visibility information
    bool              OwnsPoints;           // Does this own the points ??
    CPortalPoints    *Points;               // The vertices making up this portal

};
```

**UCHAR Status;**

This member is used to store the current status of the portal. This allows us to determine whether this portal has had its PVS calculated yet, whether it is still waiting to have its PVS calculated or whether it is currently in the process of having its visibility information calculated. During the calculation of the PVS, each portal can be set to one of the following #defines from the file CProcessPVS.h

***Excerpt from CProcessPVS.h***

```
#define PS_NOTPROCESSED     0       // Portal has not yet been processed
#define PS_PROCESSING       1       // Portal is currently being processed
#define PS_PROCESSED        2       // Portal has been processed
```

These status flags are used in a number of places. Firstly, as the PVS calculation process is one of essentially looping through each portal and calculating its PVS, we need to know which portals have been processed already so that we do not try and calculate its PVS more than one. Now this would not be necessary if we were to simply loop through the array of one-portals and calculate the PVS for each one in that order however, as discussed in the text book, the PVS calculator can be speeded up slightly by processing less complex portals first. That is, portals which have a lower 'Possible Visibility Count'. The possible visibility count is calculated prior to the core clipping process and describes the number of leaves that had the potential to be visible as determined by a simple flood fill. By choosing portals with the lowest possible visibility count first and which have not yet be processed (obviously), we have a situation where during the main clipping process, the calculation time of visibility information for more complex portals can be reduced by using the PVS already calculated for each of the lesser complex

portals it can see. There will only ever be one CPVSPortal with a status of PS_PROCESSING at any one time during the PVS calculation procedure.

**long Plane;**
In this member we will store the index of the plane in the BSP tree's plane array on which the portal was created. This is the index of the owner node of the two way portal from which this portal was cloned. As two one-way portals will be generated from a single two-way portal, both of these portals will share the same plane index. It might seem strange that these two one-way portals would share the same plane when they face into opposing half spaces of that plane but we will see that it is precisely for this reason the 'Side' member of this structure (discussed next) is used.

The plane is stored in the one-way portal because it will be needed for clipping. For example, we discussed in the text book how if during the core clipping process the generator portal is found to span the source portal's plane, the generator portal should be clipped to that plane such that any fragment of the generator portal that lay in the back space of that plane is discarded. This helps narrow the anti-penumbra and is also more vital than a simple optimization. If this was not done then when we recur through that generator portal such that it becomes the target portal in the next recursion, the anti-penumbra generated could become twisted or inverted causing erroneous clipping and endless recursive loops to be formed.

The problem one might see at the moment is that both the one-way portals duplicated from a single two-way portal share the same plane while the portals themselves are pointing into opposing half spaces of that plane. Therefore, this would only work correctly for the one-way portal that shares the same front space with the plane. For the other one-way portal, clipping the generator portal to its plane would actually remove the section of the generator portal that is in the front space of the portal instead of in its back space. This would obviously be completely incorrect which is why the 'Side' member (discussed next) is used. The Side member tells the PVS processing module on which side of the plane the portal's normal is assumed to be pointing into. If the Side member indicates that the one-way portal is actually pointing into the opposing half space of its stored plane, then the plane is flipped temporarily on the fly prior to the clip being performed such that in both cases, we always remove the section of the generator portal that is in the back space of the portal.

**UCHAR Side;**
As discussed above, this member is used to indicate whether the portal is pointing into the same front space as its stored node plane. If this member is set to FRONT_OWNER then it is and any generator portals can be clipped against this plane without any problems. As polygon clipping functions typically remove the polygon fragment that lay in the back space of the plane, this is correct and the fragment of the generator portal that lay in the back space of the portal will be discarded. If this member is set to BACK_OWNER then it means the stored node plane is actually facing into the opposing half space to that of the plane normal and as such, the plane direction will need to be flipped prior to the clip.

**long NeighbourLeaf;**
This member stores the index of the leaf that the portal flows into. As discussed in the text book, portal visibility flow happens from a source leaf, through the back of its contained portals and into their neighbor leaves. This stores the neighbor leaf of the portal; the leaf that you will arrive in if you step through the back of the portal from its owner leaf.

**UCHAR *PossibleVis;**
To optimize the core clipping process as much as possible, prior to the main anti-penumbra clipping processor being invoked, a very approximate PVS will be calculated using a simple flood fill technique controlled by the rules of one-way portal flow. This process will be very quick to perform and will generate a leaf visibility bit set for each portal. Any bits set to zero in this bit set represent leaves that could not possibly ever be visible from the portal and as such, we know during the core clipping process not to visit these leaves unnecessarily and perform wasteful expensive clipping operations to essentially arrive at a non visible result.

After the portal flow procedure has been perform for this portal (prior to the core clipping procedure being invoked for this portal), this unsigned char pointer will point to an array of bytes that contain the 'Possible' visibility set for the portal. Each bit in the array will represent a leaf and as such, each byte element in this array represents the visibility information for eight leaves with respect to this portal. This visibility set will be very approximate and vastly over generous but will help optimize the core recursive clipping procedure.

> **Note**: We discussed earlier how one of the compile time options for our CProcessPVS module is the PVSOPTIONS::FullCompile Boolean. If set to false, our PVS calculator will not bother performing the core clipping procedure at all and will simply return the PVS data based into the possible visibility information stored in this byte array for each portal. That is, the PVS calculated for each leaf will be the accumulation of the PossibleVis arrays of each portal contained in that leaf. While this will generate an very over generous PVS with very approximate visibility, it will compile extremely quickly which might be useful during development time when you wish to test the compiler but not wait hours for the compiler to calculate the actual potential visibility set.

**long PossibleVisCount;**
This member will be calculated during the initial flood of this portal and the calculation of its PossibleVis array. It will describe the number of possibly visible leaves. That is, the number of bits in the PossibleVis array that have been set to 1. As discussed a moment ago, to speed up compilation we will wish to calculate the PVS for the least complex portals first. This member describes the portals complexity and therefore, describes the order in which it should be chosen for full PVS calculation. The lower this number, the earlier in the process this portal will be chosen to have its PVS calculated.

**UCHAR *ActualVis;**
The job of the core clipping process of this module is to take the PossibleVis array of a portal and refine it using anti-penumbra clipping to generate the tightest potential visibility set possible. The resulting (actual) PVS for the portal will be stored in this array. That is, after this portal has been processed, the ActualVis array will contain the 'real' visibility information for this portal and the visibility information that will ultimately contribute to its owner leaf's PVS at the end of the process.

**CPortalPoints *Points;**
This member is of a type we have not yet discussed. The CPortalPoints structure is a specialized structure derived from CPolygon and as such is actually used to store the geometry of the portal itself. We can think of the CPortalPoints class as being a CPolygon object with a few extra member variables that pertain to the PVS clipping process.

**bool OwnsPoints;**
To save memory, we try to maximize the re-use of geometry among the portals. For example, we know that when we generate two one-way portals from a two-way portal that it would be a waste to allocate two CPortalPoints objects for each one-way portal. These polygons and their geometry will essentially be exactly the same as each other and will contain the exact same vertex data. Therefore, we can allocate one CPortalPoints structure (remember this is just a CPolygon derived object) and can have both one-way portals point at it. Therefore, in order to make sure that when each one-way portal is deleted we do not try and delete this shared CPortalPoints structure twice, only one of the one-way portals will have its OwnsPoints set to true. This will be assumed to be the portal that owns the structure and the one that will take care of releasing it when it is deleted. When the other one-way portal is deleted, which has this member set to false, it will not attempt to delete the CPortalPoints structure that it references as it knows that another object will handle its clean up.

Let us now have a look at the methods of this one way portal class for which there is only a constructor and a destructor.

## Constructor - CPVSPortal

The constructor simply initializes all members to zero, NULL or false and sets the status of this portal to its default state of PS_NOTPROCESSED. That is, this portal has not yet had its PVS data (ActualVis array) calculated.

```
CPVSPortal::CPVSPortal()
{
    // Initialise any class specific items
    Status              = PS_NOTPROCESSED;
    Plane               = -1;
    NeighbourLeaf       = -1;
    PossibleVisCount    = 0;
    PossibleVis         = NULL;
    ActualVis           = NULL;
    Points              = NULL;
    OwnsPoints          = false;
}
```

## Destructor - CPVSPortal

The destructor is simple also but sheds some light on the OwnsPoints member that we discussed a moment ago.

The portal contains three possible memory allocations that it may be responsible for releasing. If its ActualVis and PossibleVis arrays have been allocated then they will need to be deleted. Also, if this portal is the owner of the CPortalPoints object that it references then it should deleted that too.

```
CPVSPortal::~CPVSPortal()
{
    // Clean up after ourselves
    if (ActualVis && PossibleVis != ActualVis) delete []ActualVis;
    if (PossibleVis) delete []PossibleVis;
    if (Points && OwnsPoints ) delete Points;

    // Empty pointers
    PossibleVis = NULL;
    ActualVis   = NULL;
    Points      = NULL;
}
```

As you can see in the above code, we only release the CPortalPoints object if the portal's OwnsPoints Boolean is set to true.

Let us now look at the CPortalPoints object which is derived from CPolygon and contains the actual geometry of the portal referenced by this CPVSPortal object.

# The CPortalPoints Class

The CPortalPoints class essentially encapsulates a portal polygon. It is derived from CPolygon and as such inherits its geometry members and methods. This class adds some of its own members and implements its own versions of the Split and Clip functions. Once again, do not worry we do not have to write polygon splitting and clipping functions all over again. These functions are simple wrappers around their base class counterparts that facilitate the copying of the extra data into the child polygons resulting from the split/clip.

This class is declared in CProcessPRT.h and is shown below. It will be followed by a discussion of its members and an examination of its member functions.

```
class CPortalPoints : public CPolygon
{
public:

    // Constructors / Destructors for this Class
             CPortalPoints( );
             CPortalPoints( const CPolygon * pPolygon, bool Duplicate = false );
    virtual ~CPortalPoints( );

    // Public Functions for This Class
    CPortalPoints *     Clip( const CPlane3& Plane, bool KeepOnPlane );
    virtual HRESULT     Split( const CPlane3& Plane,
                               CPortalPoints * FrontSplit,
                               CPortalPoints * BackSplit);

    // Public Variables for This Class
    bool                OwnsVertices;           // Do we own the vertices stored here ?
    CPVSPortal         *OwnerPortal;            // Pointer to this points parent portal ;)
};
```

As you can see, we have added only two member variables to that of those inherited from CPolygon.

**bool OwnsVertices;**
This member is analogous to the OwnsPoints member of the CPVSPortal structure. It allows multiple CPortalPoints structures to share the same underlying vertex data. For example, when two CPVSPortals are first created (from a given two-way portal) we have seen that one CPortalPoints structure is created which is shared by both. Furthermore, as the geometry of this portal is identical to that of the two-way portal stored in the tree, we can simply assign its vertex pointer to the vertex array stored in the two-way portal in the BSP tree. That is, for each one-way portal that we initially create, its CPortalPoints structure will not have allocated its own vertex data but will point to the vertex array of the CBSPPortal from which it was cloned.

This may all sound a little over cautious but it is quite necessary for both the compiler's performance and addressing memory footprint issues. For example, we know that if portal gets clipped then we will have to allocate a new set of vertex data for the child split fragments. There is nothing we can do about that. However, there may be many times during the process where a generator portal does not get clipped at all and as such, we can happily use the vertex data that was originally created for the CBSPPortal version of the portal.

This Boolean lets the CPortalPoints destructor know whether or not the vertex array store here is owed by (was allocated for) this portal specifically (such as if this polygon was the result of a clip operation) or if its vertex pointer is assigned to the vertex array of another object in which case the memory should not be released. As discussed, when each one-way portal is created, each of their CPortalPoints objects will not contain there own vertex data but alias the vertex data stored in the portals of the BSP tree.

**CPVSPortal *OwnerPortal;**
This member is used to point at the CPVSPortal that owns this object and will be responsible for its clean up. For each pair of one-way portals that we generate, only one of them will own the CPortalPoints structure that they both alias (phew, that is a lot to keep track of).

## Constructor - CPortalPoints

There are two constructors for this class. The default constructor simply sets the owner portal pointer to NULL and sets the OwnsVertices Boolean to false by default as shown below.

```
CPortalPoints::CPortalPoints()
{
    // Initialise any class specific items
    OwnsVertices = false;
    OwnerPortal  = NULL;
}
```

The second constructor is a copy constructor that can be used to create and populate a new CPortalPoints object from the data stored in a passed CPolygon object.

The first parameter to the copy constructor is a pointer to the CPolygon we would like this object to copy or alias the vertex data of. The second parameter is a Boolean which specifies whether we would

like this object to allocate its own vertex array and copy the vertex data over from the passed CPolygon, or whether we would like to simply alias the vertex data by assigning the vertex data pointer to point at the vertex array of the passed CPolygon. In the later case, the CPortalPoints object will not own the vertex data and should no delete it within its destructor. We inform the destructor of whether or not the vertex data of this object should be released during object deletion by setting the OwnsVertices member to true or false respectively.

Here is the code:

```
CPortalPoints::CPortalPoints( const CPolygon * pPolygon, bool Duplicate )
{
    // Initialise any class specific items
    OwnsVertices = false;
    OwnerPortal  = NULL;
    if (!pPolygon) return;

    // Store or duplicate verts
    if ( Duplicate )
    {
        // Duplicate the vertices
        if (AddVertices( pPolygon->VertexCount ) < 0) throw BCERR_OUTOFMEMORY;
        memcpy( Vertices, pPolygon->Vertices, VertexCount * sizeof(CVertex) );
        OwnsVertices = true;

    } // End if Duplicate
    else
    {
        // Simply store a copy of the pointer info
        Vertices     = pPolygon->Vertices;
        VertexCount  = pPolygon->VertexCount;
        OwnsVertices = false;

    } // End if !Duplicate
}
```

As you can see, if the Duplicate Boolean parameter is set to true then we do indeed allocate a vertex array for this object and copy over the vertex data from the passed CPolygon. We then set the OwnsVertices member to true which will instruct the destructor to release this memory on object de-allocation. If the Boolean is set to false then we simply copy over the vertex count from the passed polygon and assign the vertex pointer to point at the passed CPolygon vertex array. We also set the OwnsVertices member to false so that we do not try to delete this vertex array on object destruction. These are not our vertices to delete.

## Destructor - CPortalPoints

The destructor only deletes the vertex array of the CPortalPoints object if it owns them. That is, if the OwnsVertices Boolean is set to true.

```
CPortalPoints::~CPortalPoints()
{
    // Clean up after ourselves only if required
    if (OwnsVertices) ReleaseVertices();
```

```
    // Simply NULL our vertex values
    Vertices    = NULL;
    VertexCount = 0;
}
```

It will become more apparent why we try to share as much data as possible during the core clipping process.

## Split - CPortalPoints

The CPortalPoints object implements it own polygon splitter. As with the base class version of this function it takes three parameters with the first being the split plane. The final two parameters however are now of type CPortalPoints.

As CPortalPoints is derived from CPolygon we can use the base class version of the function to perform that actual splitting of the geometry into the front and back children.

```
HRESULT CPortalPoints::Split( const CPlane3& Plane,
                                    CPortalPoints * FrontSplit,
                                    CPortalPoints * BackSplit)
{
    // Call base class implementation
    HRESULT ErrCode = CPolygon::Split( Plane, FrontSplit, BackSplit );
    if (FAILED(ErrCode)) return ErrCode;
```

As we have created two new polygons by copying over portions of vertex data from the parent polygon, each one will own its own vertices so we must set the OwnsVertices member of both the front and back split (if they exist) to true.

```
    // Copy remaining values
    if (FrontSplit)
    {
        FrontSplit->OwnsVertices = true;
    } // End If

    if (BackSplit)
    {
        BackSplit->OwnsVertices  = true;
    } // End If

    // Success
    return BC_OK;
}
```

As we have seen, this function is a simple wrapper around a call to the base class version of the function with the added logic of making sure that the child split fragments understand that they have had there own unique vertex data generated by the clipping process. This is especially true as we would not want the vertex data of the children to be deleted when the parent polygon is deleted because as we know, one of the first things we do after splitting a polygon into two children is delete the original. Our split routines (as we have seen) will always create polygon fragments that own there own vertex data.

## Clip - CPortalPoints

The Clip method simply classifies the polygon represented by this object against the plane passed in as the first parameter and removes the portion of the polygon that is found to be behind the plane. The second parameter indicates whether we would like the polygon to be clipped or kept if it is found to exist on the plane itself. This will be used later during the anti-penumbra clipping process. The function returns a pointer to the new clipped polygon.

First we classify the polygon against the plane and then enter a switch statement that chooses the outcome based on the classification result. Notice at the top of the function how the NewPoints local pointer is allocated. This will be used to point to the clipped fragment and will be the pointer that is returned from the function.

```
CPortalPoints * CPortalPoints::Clip( const CPlane3& Plane, bool KeepOnPlane )
{
    CPortalPoints * NewPoints = NULL;

    try
    {
        // Classify the points
        CLASSIFYTYPE Location = Plane.ClassifyPoly( Vertices, VertexCount, sizeof(CVertex) );

        // What location ?
        switch ( Location )
        {
```

If the polygon is found to be contained in the front space of the plane then nothing is to be clipped. Therefore, we simple assign the NewPoints pointer to point at the current polygon. This means, the method will return a pointer to the CPortalPoints structure for which it was invoked. That is, the object will just return a pointer to itself.

```
            case CLASSIFY_INFRONT:
                // All were in front, simply return this
                NewPoints = this;
                break;
```

If the polygon is found to exist entirely in the back space of the clip plane then the polygon should be totally clipped away. When this is the case we return NULL indicating that none of the polygon should survive. The caller can then choose to delete the object if it so chooses.

```
            case CLASSIFY_BEHIND:
                // Nothing was in front
                NewPoints = NULL;
                break;
```

We will see later when we cover the core clipping function of the PVS calculation how we sometimes want to clip away a polygon even if it is located on the plane. For example, if the generator portal is located on the same plane as the target portal then the target portal can not possibly see through the generator portal and should be completely clipped away.

You can see below that in the on plane case, if the KeepOnPlane Boolean parameter is set to true, we just assign the NewPoints pointer to the 'this' pointer allowing the object to return a pointer to itself. Otherwise, we break and NULL will be returned at the bottom of the function signifying to the caller that the portal should be deleted.

```
            case CLASSIFY_ONPLANE:
                // Should we keep the onplane case ?
                if ( KeepOnPlane ) NewPoints = this;
                break;
```

Finally, if the polygon is spanning the plane, we create a new CPortalPoints object which is passed into the Split function to retrieve the front fragment. NULL is passed as the back split as we do not wish to retrieve the back fragment of the polygon as this is the fragment that should be discarded.

```
            case CLASSIFY_SPANNING:

                // Allocate a new set of points
                NewPoints = new CPortalPoints;
                if (!NewPoints) throw std::bad_alloc();

                // Clip the current portal points
                if ( FAILED(Split( Plane, NewPoints, NULL ))) throw BCERR_OUTOFMEMORY;
                break;

        } // End Switch

    } // End try block

    catch (...)
    {
        return NULL;

    } // End catch block

    // Success!!
    return NewPoints;
}
```

# The CProcessPVS Class

With the initial support structures covered, we will now look at the code to the CProcessPVS module. It is declared in ProcessPVS.h and is shown below. First four public methods of the object's interface should be familiar as they form the method set common to all our modules. They include the Process method that is called by CCompiler to invoke the PVS calculator module and the methods to set the modules Logger, parent and options. There are also two additional public methods called 'GetPVSPortalCount' and 'GetPVSPortal' which can be used to fetch the one-way portal information stored in the module. There are also many private functions which are used by the Process method to accomplish its task which we will examine in a moment.

```
class CProcessPVS
{
public:
    // Constructors & Destructors for This Class.
             CProcessPVS();
    virtual ~CProcessPVS();

    // Public Functions for This Class.
    HRESULT         Process( CBSPTree * pTree );
    void            SetOptions( const PVSOPTIONS& Options ) { m_OptionSet = Options; }
    void            SetLogger ( ILogger * pLogger )         { m_pLogger = pLogger; }
    void            SetParent ( CCompiler * pParent )       { m_pParent = pParent; }

    unsigned long   GetPVSPortalCount( ) const
                                { return (unsigned long)m_vpPVSPortals.size(); }

    CPVSPortal     *GetPVSPortal( unsigned long Index ) const
                                { return (Index < m_vpPVSPortals.size()) ?
                                          m_vpPVSPortals[Index] : NULL; }

private:
    // Private Functions for This Class.
    HRESULT         GeneratePVSPortals( );
    HRESULT         InitialPortalVis( );
    HRESULT         CalcPortalVis( );
    void            PortalFlood( CPVSPortal * SourcePortal,
                                 unsigned char PortalVis[],
                                 unsigned long Leaf );

    HRESULT         ExportPVS( CBSPTree * pTree );

    void            GetPortalPlane( const CPVSPortal * pPortal, CPlane3& Plane );

    ULONG           CompressLeafSet ( UCHAR MasterPVS[],
                                      const UCHAR VisArray[],
                                      ULONG WritePos);

    ULONG           GetNextPortal();

    HRESULT         RecursePVS( ULONG Leaf, CPVSPortal * SourcePortal, PVSDATA & PrevData );

    CPortalPoints * ClipToAntiPenumbra( CPortalPoints * Source,
                                        CPortalPoints * Target,
                                        CPortalPoints * Generator,
                                        bool ReverseClip );

    // Private Static Functions for This Class.
```

```
    static CPortalPoints *  AllocPortalPoints( const CPolygon * pPolygon, bool Duplicate );
    static bool             GetPVSBit( UCHAR VisArray[], ULONG DestLeaf );
    static void             SetPVSBit( UCHAR VisArray[], ULONG DestLeaf, bool Value = true );
    static void             FreePortalPoints( CPortalPoints * pPoints );

    // Private Variables for This Class.
    PVSOPTIONS      m_OptionSet;        // The option set for PVS Compilation.
    ILogger        *m_pLogger;          // Logging interface used to log progress etc.
    CCompiler      *m_pParent;          // Parent Compiler Pointer

    CBSPTree       *m_pTree;            // The tree used to compile the PVS.
    ULONG           m_PVSBytesPerSet;   // Number of Bytes required to
                                        // describe a single leaf's visibility
    vectorPVSPortal m_vpPVSPortals;     // Vector storage of pointers to CPVSPortal objects
};
```

We will first examine the member variables of this object before examining each of its methods.

**PVSOPTIONS m_OptionSet;**
This member holds the compilation options for the PVS module and is set by CCompiler via the CProcessPVS::SetOptions method. We looked at this structure earlier and saw that it contained members to instruct the module to either do a full or fast compile (where fast simply use the PossibleVis array and does not perform the anti-penumbra clipping procedure) and describes the number of anti-penumbra clip tests (1 to 4) that should be carried out for each Source/Generator portal combination encountered during the recursive clipping procedure.

**ILogger *m_pLogger;**
This member will point to the logging object whose interface will be used by this module to output status reports and compilation errors or warnings. This is set by the CCompiler object prior to the Process function being called via the CporcessPVS::SetLogger method.

**CCompiler *m_pParent;**
This member is set by the CCompiler object via the CProcessPVS::SetParent method. It stores a pointer to the CCompiler object that invoked it.

**CBSPTree *m_pTree;**
This member points to the BSP tree that is having its PVS data calculated and that contains the portal information generated by the previously discussed module. This pointer is set by CCompiler via the parameter to the CProcessPVS::Process method.

**ULONG m_PVSBytesPerSet;**
This member will be calculated by this module at the start of the Process method and will contain the number of bytes needed to store the visibility information for a single leaf. As each byte contains 8 bits a single byte will contain the information for 8 leaves.

**vectorPVSPortal m_vpPVSPortals;**
This is an STL vector will store CPVSPortal structures. It is in this vector that the pointers of all the one-way portals generated by this module will be stored.

## Constructor - CProcessPVS

The constructor of this module simply initializes all members to zero or NULL.

```
CProcessPVS::CProcessPVS()
{
    // Reset / Clear all required values
    m_PVSBytesPerSet    = 0;
    m_pLogger           = NULL;
    m_pTree             = NULL;
    m_pParent           = NULL;
}
```

## Destructor - CProcessPVS

The destructor of this object must release the one-way portals (CPVSPortal structures) that it allocated to complete its task and must also empty the vector that contained these pointers.

```
CProcessPVS::~CProcessPVS()
{
    ULONG i;

    // Clean up after ourselves
    for ( i = 0; i < GetPVSPortalCount(); i++ ) if ( GetPVSPortal(i) )
                                                    delete GetPVSPortal(i);

    // Clear Vectors
    m_vpPVSPortals.clear();
}
```

## AllocPortalPoints - CProcessPVS

There will be many times throughout the PVS calculation process that we will need to allocate new CPortalPoints object. In keeping with the allocation strategy we have used for other modules, this function can be used to allocate a new object of this type. The function simply wraps the allocation call.

The function takes two parameters. The first is a pointer to the CPolygon object (or derived object) that has the geometry we would like this new CPortalPoints object to represent and the second parameter is a Boolean that specifies whether we would like the CPortalPoints structure to copy the polygon data into its own vertex array or simply alias it. These parameters are simply passed into the CPortalPoints copy constructor which we have already covered.

```
CPortalPoints * CProcessPVS::AllocPortalPoints( const CPolygon * pPolygon, bool Duplicate )
{
    CPortalPoints * NewPoints = NULL;

    try
    {
        // Attempt to allocate a new set of points
```

```
        NewPoints = new CPortalPoints( pPolygon, Duplicate );
        if (!NewPoints) throw std::bad_alloc();

    } // End try block

    catch (HRESULT)
    {
        // Constructor throws HRESULT
        if (NewPoints) delete NewPoints;
        return NULL;

    } // End Catch

    catch ( std::bad_alloc )
    {
        // Failed to allocate
        return NULL;

    } // End Catch

    // Success!!
    return NewPoints;
}
```

## FreePortalPoints - CProcessPVS

This module also has a method that can be used to free a CPortalPoints structure. It takes a single parameter, a pointer to the CPortalPoints structure that is to be released.

```
void CProcessPVS::FreePortalPoints( CPortalPoints * pPoints )
{
    // Validate Parameters
    if (!pPoints) return;

    // We are only allowed to delete NON-Owned point sets
    if ( pPoints->OwnerPortal == NULL ) delete pPoints;
}
```

Notice that this method will only physically delete the CPortalPoints object if it is not owned by a parent portal. If it is owned by a portal then the portal should be responsible for its clean up inside the portals destructor.

## SetPVSBit - CProcessPVS

There will be several times throughout the PVS compilation procedure when we will need to set or clear a bit in a given portal's visibility bit set (be that its ActualVis array or its PossibleVis array). This method is a utility method that allows us to pass a bit set, a leaf whose visibility bit is to be altered in that set and a Boolean describing whether that leaf's bit should be set or cleared in the passed bit set. The function wraps calculating the bit that needs to be set in the passed bit array and setting that bit accordingly.

```
void CProcessPVS::SetPVSBit( UCHAR VisArray[], ULONG DestLeaf, bool Value /* = true */ )
{
    // Set / remove bit depending on the value
    if ( Value == true )
    {
        VisArray[ DestLeaf >> 3 ] |=  (1 << ( DestLeaf & 7 ));
    }
    else
    {
        VisArray[ DestLeaf >> 3 ] &= ~(1 << ( DestLeaf & 7 ));

    } // End if Value
}
```

The bit shifting logic is explained in the text book so refer back if you are feeling a little rusty. Recall that DestLeaf>>3 just divides it by eight which tells us the byte in which this leaf's visibility bit resides. By ANDing the leaf index with 7 (binary 00000111) we also get the bit within that byte that needs to be set (0 through 7). Therefore, we can shift a value of one by this amount to create a byte that has only that bit set (or unset) and then OR it with the byte in the bit set to toggle that leaf's bit on or off.

## GetPVSBit - CProcessPVS

We also have a function that uses the same bit shifting logic to retrieve the visibility status of a leaf in the passed bit set.

```
bool CProcessPVS::GetPVSBit( UCHAR VisArray[], ULONG DestLeaf )
{
    return (VisArray[ DestLeaf >> 3 ] & (1 << ( DestLeaf & 7))) != 0;
}
```

With these utility functions out of the way, let us now look at the function that makes it all happen. The CProcessPVS::Process method.

## Process - CProcessPVS

This function is the parent function of the PVS calculation process that calls the various sub-processes in order to calculate the final PVS for the passed BSP tree. We will look at the code a section at a time.

The first thing we do in this function is test that the passed tree has portal data generated for it. If this is not the case we return immediately as we can not possibly calculate PVS data for this tree. Otherwise, we copy the passed BSP tree pointer into the module's member variable.

```
HRESULT CProcessPVS::Process( CBSPTree * pTree )
{
    HRESULT hRet;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Validate Input Data
    if ( pTree->GetPortalCount() == 0 ) return BCERR_BSP_INVALIDTREEDATA;
```

```
    // Store tree for compilation
    m_pTree = pTree;
```

The next thing we do is calculate the module's m_PVSBytesPerSet member such that it describes the number of bytes needed to hold the visibility bit set for a single leaf. As discussed in the text book, although there are 8 bits per byte with each bit representing a leaf's visibility, we can not just divide the total leaf count of the tree by 8 to calculate this size. If we did it would erroneously calculate the number of bytes to be allocated to store a 9 leaf tree as 9/8 = 1 (integer math). As we know, we would actually need two bytes to represent this information with the first 8 leaves having their bits in byte one and the $9^{th}$ leaf with its bit in the first position in byte 2. Therefore, to cope with the integer truncation we add 7 to the leaf count first and then divide this by 8. Therefore, if we had a 9 leaf tree, the number of bytes we would need to represent a bit for each leaf would be (9+7)/8 = 16/8=2.

```
    // Calculate Number Of Bytes needed to store each leafs
    // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    m_PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;
```

For reasons that will become clear in the core clipping process we also wish to pad this size to the nearest four byte boundary. This will allow us to access the visibility byte arrays using a long pointer and iterate through 4 bytes (32 leaves) at a time. For example, imagine that we allocated three bytes and then tried to write to the first byte with a four-byte pointer (long *). In this case, we would accidentally write to the fourth byte, overstepping the bounds of the array and writing to invalid memory. Therefore, if we need 22 bytes, we will allocate 24 bytes instead; the two bytes at the end will serve as padding only and will never be used by us.

```
    // 32 bit align the bytes per set to allow for our early out long conversion
    m_PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;
```

This m_PVSBytesPerSet member now describes the size that we will need to allocate each portal's PossibleVis and ActualVis arrays later in the process. These arrays will be padded with extra unused bytes at the end of the array if necessary to make sure we can access the array four bytes at a time without overflowing the array. The remainder of the function calls four member functions and clearly shows the four sub-process involved in generating the PVS data for the tree.

First we call the GeneratePVSPortals method. This is the method that will fetch each two-way portal from the tree and will generate two one-way CPVSPortal structures from it. It is the CPVSPortals that will be used by the PVS calculator to control flow visibility as we recur through the level.

```
    // Retrieve all of our one way portals
    hRet = GeneratePVSPortals();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

After the above function returns, this object's m_vpPVSPortals vector will be filled with all the one-way portals needed to perform PVS calculation. Because of their one-way nature, there will be twice as many CPVSPortals in this array as there are CBSPPortals stored in the BSP tree.

With the one-way portals generated, next we call the InitialPortalVis function. It is this function that will perform the initial flood fill through the level and calculate the PossibleVis array for each portal. You will recall that the PossibleVis array stored in each portal is a very approximate leaf visibility array for that portal which will be used to speed up the core clipping process when the portal's ActualVis array is calculated.

```
    // Calculate initial portal visibility
    hRet = InitialPortalVis();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

When the above function returns each CPVSPortal will contain its PossibleVis array of approximate leaf visibility information. Next we enter the core clipping process with a call to CalcPortalVis. This function is the function that will calculate the ActualVis array for each portal. That is, the function that will generate the leaf visibility information for each portal, the portals' PVS.

```
    // Perform actual full PVS calculation
    hRet = CalcPortalVis();
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;
```

At this point in the function the PVS for each portal will have been calculated so we next call the ExportPVS method. This method will use the portal PVS's to calculate the PVS for each leaf in the tree. Recall that a leaf's PVS is simply the accumulation of the PVS's of each portal residing in that leaf. After the PVS for each leaf has been calculated this information will be compressed and stored in the BSP tree in a single byte array. The leaves of the tree will also have their PVSIndex members pointing to the correct location in this array describing the starting location of the leaf's visibility information within the final PVS data block.

```
    // Export the visibility set to the final BSP Tree master array
    hRet = ExportPVS( pTree );
    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Success
    return BC_OK;
}
```

After the ExportPVS method returns, all that is left to do is return from the function as the PVS information has been calculated, optionally compressed using zero run length encoding and stored in the BSP tree. Our module at this point has completed its task and can return program flow back to CCompiler.

> **Note:** Recall that after the PVS module has returned after completing its process, CCompiler will then activate the T-junction repair module as a final step before saving the BSP tree out to disk in the form of an IWF file.

We will now cover the methods called from the above function in order and get a real understanding of where and when everything happens throughout the PVS calculation process. The first function to be called is the GeneratePVSPortals method so we will examine the code to this function first.

## GeneratePVSPortals - CProcessPVS

This method has a very simple task. It has to loop through each CBSPPortal (two-way portal) stored in the BSP tree and generate two CPVSPortals (one-way portals) for it. Each one-way portal will have a different neighbor leaf selected from the two leaves in which the two-way portal resides. This will dictate the direction in which the one-way portals are assumed to facing.

The first section of the code fetches the number of portals stored in the BSP tree. We then set up a loop to allocate twice this many CPVSPortals and store their pointers in the CProcessPVS::m_vpPVSPortals vector.

```
HRESULT CProcessPVS::GeneratePVSPortals( )
{
    ULONG i, p, PortalCount = m_pTree->GetPortalCount();

    // Allocate enough PVS portals to store one-way copies.
    try
    {
        m_vpPVSPortals.resize( PortalCount * 2 );
        for ( i = 0; i < PortalCount * 2; i++ )
        {
            // Allocate a new portal
            m_vpPVSPortals[i] = new CPVSPortal;
            if (!m_vpPVSPortals[i]) throw std::bad_alloc();

        } // Next Portal

    } // Try vector ops

    // Catch Failures
    catch (...)
    {
        return BCERR_OUTOFMEMORY;

    } // End Catch
```

At this point we have allocated the correct number of one-way portals and have their pointers stored in the module's one-way portal vector. However, these portals are still un-initialized as we have not yet populated them with geometry.

In the next step we will loop through each of the portals we just calculated two at a time. We process the one-way portals in pairs in the following loop as each pair of one-way portals is a child (to use the term very loosely) of a single one-way portal in the BSP tree. So you can see that we set up the following loop to step through the one-way portal list two portals at a time. Heee is the first section of the loop code.

```
    // Loop through each portal, creating the duplicate points
    for ( i = 0, p = 0; i < PortalCount; i++, p+=2 )
    {
        // Retrieve BSP Portal for easy access
        CBSPPortal * pBSPPortal = m_pTree->GetPortal(i);
```

```
        CPortalPoints *pp = AllocPortalPoints( pBSPPortal, false );
        if ( !pp ) return BCERR_OUTOFMEMORY;
```

Inside the loop we fetch a pointer to the original two-way portal from the BSP tree whose geometry we are going to copy/alias using the pair of one-way portals we are currently processing. pBSPPortal points to the original portal in the BSP tree which we will use to set up two one-way portals. We then allocate a new CPortalPoints object. Remember, this is the polygon that will be stored in the two CPVSPortals we are about to populate.

Notice how we clone the CPortalPoints object from the BSP portal. Although we are creating two one way portals, we only need to create one CPortalPoints objects as each one-way portal will share the same physical polygon data. Only one of the CPVSPortals we are about to populate will actually own the CPortalPoints structure and will be responsible for deleting it within its destructor. The other CPVSPortal object will simply reference it.

Another important point to notice is how we also pass false into the copy constructor so that the CPortalPoints object does not have its own array of vertex data allocated, but simply aliases the vertex array stored in the original BSP portal. Therefore, we will create two CPVSPortal structures that share the same CCPortalPoints object and that CPortalPoints object will share its vertex data with the original BSP portal. It is clear then that the two CPVSPortals we are about to populate share the same vertex data, the vertex data in the original BSP polygon.

Here is the remainder of the loop (and the function) that sets up each CPVSPortal which we discuss beneath it.

```
        // Create link information for front facing portal
        m_vpPVSPortals[p]->Points        = pp;
        m_vpPVSPortals[p]->Side          = FRONT_OWNER;
        m_vpPVSPortals[p]->Status        = PS_NOTPROCESSED;
        m_vpPVSPortals[p]->Plane         = m_pTree->GetNode( pBSPPortal->OwnerNode )->Plane;
        m_vpPVSPortals[p]->NeighbourLeaf = pBSPPortal->LeafOwner[ FRONT_OWNER ];
        m_vpPVSPortals[p]->OwnsPoints    = true;

        // Store owner portal information (used later)
        pp->OwnerPortal = m_vpPVSPortals[p];

        // Create link information for back facing portal
        m_vpPVSPortals[p + 1]->Points   = pp;
        m_vpPVSPortals[p + 1]->Side     = BACK_OWNER;
        m_vpPVSPortals[p + 1]->Status   = PS_NOTPROCESSED;
        m_vpPVSPortals[p + 1]->Plane    = m_pTree->GetNode( pBSPPortal->OwnerNode )->Plane;

        m_vpPVSPortals[p + 1]->NeighbourLeaf = pBSPPortal->LeafOwner[ BACK_OWNER ];
        m_vpPVSPortals[p + 1]->OwnsPoints    = false;

    } // Next Portal

    // Success!!
    return BC_OK;
}
```

Notice how each one-way portal we populate points to the same CPortalPoints structure but only the first portal we create has its OwnsPoints Boolean set to true. The first one-way portal is therefore the owner of the polygon itself and is responsible for the cleanup of that polygon in its destructor.

Notice also that because we know that the BSP portal has the index of the leaf in its front space stored in element zero (FRONT_OWNER) in its LeafOwner array and the leaf that is located in its back space located in element 1 (BACK_OWNER) in the portal's LeafOwner array, we can easily set up portals that correctly point into the relevant neighbor leaves. For example, the first portal we set up is assigned the Side value of front owner which means we intend this portal to be the one that faces in the same direction as the owner node's plane. When this is the case we know that its neighbor leaf (the leaf its normal should point into should a normal actually exist) is in the front space of the node plane and as such, we assign to its NeighborLeaf member the leaf index stored in the BSP portal's LeafOwner array at element FRONT_OWNER. Notice that we simply flip this logic to set up the 2nd one-way portal so that its side is set to BACK_OWNER. This means its normal is assumed to flow into the back space of the leaf and thus, it faces in the opposite direction to that of the original node plane.

Finally, notice how we also store the plane of the portal's owner node (the portals plane) in the CPVSPortals. As discussed earlier, although the portals are supposed to be facing in opposing directions we store the same clip plane in both. The Side member will instruct the core clipping process to flip the plane orientation of the back facing portal prior to clipping anything against it. This may seem a little unclear at the moment which is why we will jump ahead temporarily and look at a function that will be used later during the main clipping process, the GetPortalPlane function.

## GetPortalPlane - CProcessPVS

This function is used by the PVS calculator when it wishes to retrieve the plane of a CPVSPortal object. This is often required when generator portals need to be clipped against the plane of the source portal and vice versa. As our clip functions will always remove portions of a polygon/portal that lay in the back space of a clip plane, and considering that the two one-way way portals that exist on the same plane and were duplicated from the same two way polygon both store the same plane, it is obvious that just using this clip plane for the back facing portal (Side = BACK_OWNER) would cause errors. That is, as the plane normal is facing into the opposite half space as the portal, clipping anything away that lies behind this plane would actually clip away anything that lay in the front half space of the portal causing obvious PVS errors. To fix this problem the GetPortalPlane method is used to retrieve the plane of a portal. If the portal whose plane is being retrieved has its side set to BACK_OWNER, then the plane orientation is flipped prior to it being returned. This will make sure that the returned plane always faces into the same front space as the portal flows.

```
void CProcessPVS::GetPortalPlane( const CPVSPortal * pPortal, CPlane3& Plane )
{

    // Store plane information
    Plane = *m_pTree->GetPlane( pPortal->Plane );

    // Swap sides if necessary
    if ( pPortal->Side == BACK_OWNER )
    {
        Plane.Normal   = -Plane.Normal;
```

```
        Plane.Distance = -Plane.Distance;

    } // End if Swap Sides
}
```

## InitialPortalVis - CProcessPVS

After the one-way portal array has been created, the Process method next calls the InitialPortalVis method. It is this method that performs the flood fill through the leaves of the tree and constructs a very crude visibility array (PossibleVis) for each portal. As we have discussed, this array will be used to optimize the main PVS calculation procedure (the anti-penumbra clipping process).

The function first sends some information to the PVS logging channel specifying that the initial portal flow is about to be calculated and the progress range of the logger is set to the total number of portals in the one-way portal array. Once we have processed each portal in this array and calculated its PossibleVis array, we will have completed the initial portal flow procedure.

```
HRESULT CProcessPVS::InitialPortalVis()
{
    CPortalPoints  *pp;
    ULONG           p1, p2, i;
    CPlane3         Plane1, Plane2;
    UCHAR          *PortalVis = NULL;
    CPVSPortal     *pPortal1, *pPortal2;

    // *************************
    // * Write Log Information *
    // *************************
    if ( m_pLogger )
    {
        m_pLogger->LogWrite( LOG_PVS,
                             0,
                             true,
                             _T("Calculating initial PVS portal flow \t\t- " ) );

        m_pLogger->SetRewindMarker( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
        m_pLogger->SetProgressRange( GetPVSPortalCount() );
        m_pLogger->SetProgressValue( 0 );
    }
    // *************************
    // *    End of Logging     *
    // *************************
```

The rest of the function is essentially just a loop through each portal that performs a flood fill and is comprised of two different stages.

In the first stage we calculate which portals in the level could conceivably have portal flow with the current portal being processed. This allows us to compile a temporary byte array large enough to store a byte for each portal. Each byte in this array is set to either 1 or 0 depending on whether that portal can be seen from the current portal being processed. Portals that have their bytes set to zero in this array will essentially stop the flood for this portal. Once we have this portal visibility array compiled for the current portal being processed, it will be passed into a recursive flood filling function and used to

calculate the PossibleVis array for the current portal. Remember, the PossibleVis array of the portal is a bit set containing the 'possible' leaf visibility array of the portal. That is, which leaves in the tree have the potential to be visible during the anti-penumbra clipping process.

So, the first thing we must construct inside the loop that iterates through each portal, is the array of bytes that describes which portals can be seen by the current portal being processed.

We loop through each of the other portals and perform two tests on it to determine whether its corresponding byte in the array should be set to zero or one for the current source portal.

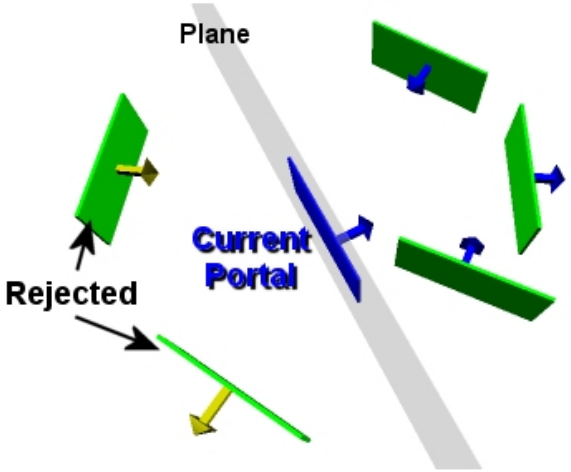


Figure 17.3

Firstly, as portal flow passes through the back of a portal out into its neighbor leaf, we know that any portals that are located behind the plane of the current portal being processed can not possibly be visible to the current portal as shown in figure 17.3. We can see that in the initial test we find two portals located behind the current portal's plane. This means visibility flow can not exist out of the front of the current portal and through the back of these two portals. That is, the current portal can only see into its front space and therefore can not possible see through portals in its back space. As such, the two portals in the example would have their bytes set to zero in the current portals array.

As flow can only exist out of the front of one portal and through the back of another, we can also see that for portals in the front space of the current portal which face towards the current portal no flow can exist also.

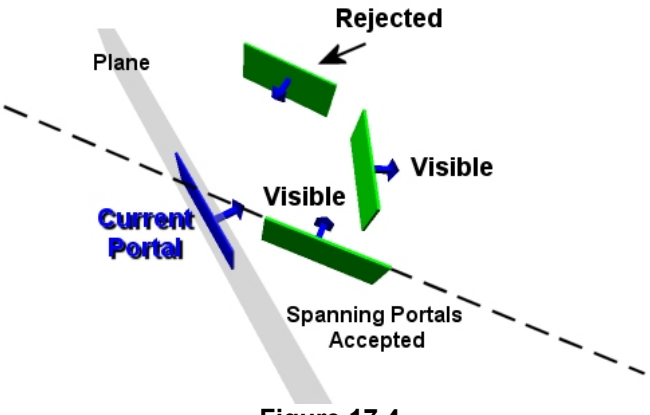


Figure 17.4

In figure 17.4 we can see that in this second test another portal is also rejected by the visibility test because its faces into the opposing half space to the current portal. That is, the text portal and the current portal are facing each other.

We can see that the middle portal on the right hand side of the diagram is not rejected from the visibility test because it faces into the same half space as the current portal. Therefore, we can clearly see that the back side of this portal is clearly visible from the current portal and as such, portal flow can happen between these two portals. The bottom portal of the three test portals is also accepted even though its plane spans the current portal. We will have to sort this problem out later in the main recursive process and clip such portals to each others plane, but for now it is accepted as at least some of the current portal lay in the test portals back space and can see through the back of that portal. Therefore, portal flow does occur between these portals to some degree.

Let is now see the code that performs these tests and compiles the temporary portal/portal visibility buffer. This should obviously be large enough to store a byte for each portal.

```
    try
    {
        // Allocate temporary visibility buffer
        if ( !(PortalVis = new UCHAR[ GetPVSPortalCount() ])) throw std::bad_alloc();

        // Loop through the portal array allocating and checking
        // portal visibility against every other portal
        for ( p1 = 0; p1 < GetPVSPortalCount(); p1++)
        {
            // Update progress
            if (!m_pParent->TestCompilerState()) break;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );

            // Retrieve first portal for easy access
            pPortal1 = GetPVSPortal( p1 );

            // Retrieve portal's plane
            GetPortalPlane( pPortal1, Plane1 );

            // Allocate memory for portal visibility info
            if (!(pPortal1->PossibleVis = new UCHAR[m_PVSBytesPerSet]))
                  throw std::bad_alloc();

            ZeroMemory( pPortal1->PossibleVis, m_PVSBytesPerSet );

            // Clear temporary buffer
            ZeroMemory( PortalVis, GetPVSPortalCount() );
```

Starting at the top of the above section of code we can see that we allocate the byte array that will be temporarily used by each portal we process to contain its portal/portal visibility information. We allocate it large enough to store a byte for each one-way portal.

We then set up a loop to iterate through each portal and compute its possible visibility. After updating the logger's progress we fetch the pointer to the portal that is to be process p1. We then fetch the portals plane using the GetPortalPlane method. Recall from earlier that this function will take care of returning a plane which always faces into the portals neighbor leaf. As this portal (pPortal1) is about to have its PossibleVis array calculated we allocate this array next. Notice that it is allocated to m_PVSBytesPerSet in size. The value of this variable was calculated at the beginning of the Process method and contains how many bytes (including 4 byte alignment padding) we must allocate to have an array where we can represent a single visibility bit for each leaf in the tree. We then zero this array initially so that all leaves are considered invisible to this portal by default. We also zero the PortalVis buffer which is the buffer we will use temporarily in this loop to calculate the portal/portal visibility information (byte for each portal).

Now that we have all the information for the current portal that we are processing, we will set up a loop to iterate through all the other portals in the scene and will perform the two visibility tests illustrated in figures 17.3 and 17.4. At the head of this inner loop we obviously skip the tests if the test portal is equal to the current portal being processed. If this is not the case however, then we fetch a pointer to the test portal and its plane.

```
            // For this portal, loop through all other portals
            for ( p2 = 0; p2 < GetPVSPortalCount(); p2++)
            {
                // Don't test against self
                if (p2 == p1) continue;

                // Retrieve second portal for easy access
                pPortal2 = GetPVSPortal( p2 );

                // Retrieve portal's plane
                GetPortalPlane( pPortal2, Plane2 );
```

For the first test we loop though each of the test portals vertices and classify their positions against the plane of the current portal. If any vertex is found to exist in the front space of the current portal then some portal flow may exist so we break. If we do not break from the loop prematurely however, then it means we must have found that the test portal is contained completely in the back space of the current portal's plane. When this is the case the loop variable 'i' will equal the vertex count of the test portal on loop exit which we can test and skip any further processing if this is the case. That is, if the test portal is found to be in the back space of the current portal then it can not possibly be visible and no portal flow can exist between them so we skip any further tests and never set its byte to one in the PortalVis array.

```
                // Test to see if any of p2's points are in front of p1's plane
                pp = pPortal2->Points;
                for ( i = 0; i < pp->VertexCount; i++)
                {
                   if ( Plane1.ClassifyPoint( pp->Vertices[i] ) == CLASSIFY_INFRONT ) break;

                } // Next Portal Vertex

                // If the loop reached the end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;
```

If we get this far then it means the test portal (or a fragment of it) must be located in front of the current portal. Our next test is to see if the portals are facing into each other because if this is the case, no portal flow can exist between them also.

As the portals do not contain normals we perform this test by classifying each vertex in the current portal against the plane of the test portal. If portal flow exists between these portals then some of the vertices of the current portal must be located in the back space of the test portal. As soon as we find a vertex that is contained in the back space of the test portal we break from the loop.

```
                // Test to see if any of p1's portal points are Behind p2's plane.
                pp = pPortal1->Points;

                for ( i = 0; i < pp->VertexCount; i++)
                {
                    if ( Plane2.ClassifyPoint( pp->Vertices[i] ) == CLASSIFY_BEHIND ) break;

                } // Next Portal Vertex

                // If the loop reached the end, there were no points in front so continue
                if ( i == pp->VertexCount ) continue;
```

If the loop does not exit prematurely then it means the current portal is in the front space of the test portal and therefore, these portals must be facing each other. When this is the case, loop variable 'i' will be equal to the current portal's vertex count so we can perform this comparison and skip this portal if this is the case. That is, the back of the test portal is not visible to the current portal so no portal flow can exist between them.

If the test portal passes both these tests then it means portal flow does exist between the current portal and the test portal so we set the test portal's byte to 1 in the PortalVis buffer.

```
                // Fill out the temporary portal visibility array
                PortalVis[p2] = 1;

            } // Next Portal 2
```

After the above inner loop has completed, PortalVis will contain a byte set to 1 for every portal that is potentially visible from the current portal. All we have to do now is perform the flood fill using this information.

```
            // Now flood through all the portals which are visible
            // from the source portal through into the neighbour leaf
            // and flag any leaves which are visible (the leaves which
            // remain set to 0 can never possibly be seen from this portal)
            pPortal1->PossibleVisCount = 0;
            PortalFlood( pPortal1, PortalVis, pPortal1->NeighbourLeaf );

        } // Next Portal
```

As you can see, before performing the flood fill we set the current portals PossibleVisCount to zero as we have not yet found any leaves to be visible, this is what the PortalFlood method will determine. We then call the PortalFlood function (which we will discuss next) to perform the flood. We pass in the current portal which is to have its PossibleVis array calculated, the PortalVis buffer which describes where the flood is blocked by non-visible portals and as the third parameter we pass in the neighbor leaf index of the current portal. It is in this leaf the portal flood will begin. When the outer portal loop exits, every portal will have had a chance to be the current portal and will have had its PossibleVis array calculated using the PortalFlood function.

Before we continue we next test that the compiler has not had a notification to cancel the compilation and if it has received such a notification we return. (Remember that the compiler is running in its own thread).

```
        // If we're cancelled, clean up and return
        if ( m_pParent->GetCompileStatus() == CS_CANCELLED )
        {
            if ( PortalVis) delete[]PortalVis;
            return BC_CANCELLED;

        } // End if Cancelled.

    } // End Try Block

    // Catch Bad Allocations
```

```
    catch ( std::bad_alloc )
    {
        // Clean up and return (Failure)
        if (PortalVis) delete []PortalVis;
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End Catch Block
```

At the bottom of the function we delete the temporary portal/portal visibility buffer and inform the logger that the process has been a success before returning from the function.

```
    // Clean up
    if (PortalVis) delete []PortalVis;

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

As was evident from look at the above code, a call is made to the PortalFlood function for each portal it processes to generate that portals 'possible' visibility set. Let us have a look at that function next.

## PortalFlood - CProcessPVS

This recursive function is passed a source portal which needs to have its PossibleVis array populated with leaf visibility information. It is also passed a byte array describing, for the passed portal, which other portals in the level are visible. As the third parameter the neighbor leaf of the portal is passed. This is the leaf that is located immediately in front of the passed portal and is the leaf at which the flood fill for the portal will begin. This function will start from the neighbor leaf and will recursively flood out through its portals into neighboring leaves. The flood stops in any given direction when we enter an area where its portals are no longer considered visible from the passed source portal. That is, when we reach a portal that has its corresponding byte in the PortalVis buffer set to zero.

For each leaf we recur into via a portal we set its visibility bit in the source portals PossibleVis array to 1. Figure 17.5 demonstrates this process.

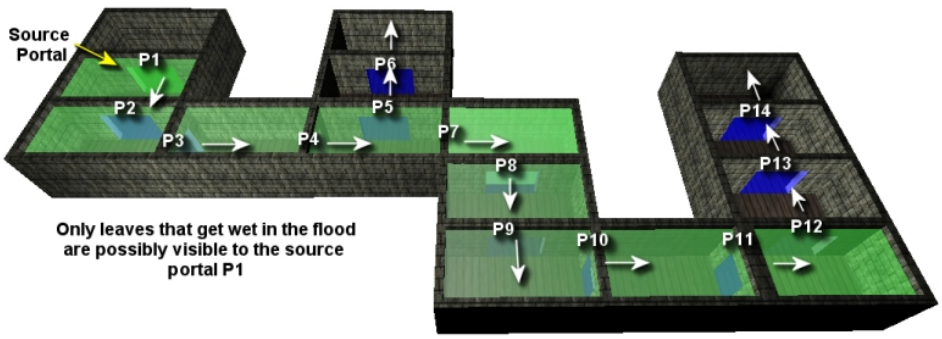



**Figure 17.5**

We can see that the flood starts in the neighbor leaf of portal P1 and flows through every portal for which portal flow exists and marks that portal's neighbor leaf as visible. The pattern repeats until we run out of visible portals to flood through. You can see in figure 17.5 that we do not flood through portals P5 or P2 as these portals were found to have no portal flow with the source portal (P1) in the previous method. That is, we only flow through a portal and mark its neighbor leaf as visible if its corresponding

byte in the passed PortalVis array is set to 1. In figure 17.5 the leaves that are highlighted green were marked as possibly visible by this flood filling process.

In the first section of the function we use the passed neighbor leaf index to fetch the leaf structure from the BSP tree's leaf array. As this is the leaf we are currently visiting it means the flood has made it into this leaf and therefore this leaf must be visible from the portal in question. However, because we do not wish to ever get stuck in a recursive loop we must make sure that we have not visited this leaf before in the flood so that we do not flow through its portals again. You can see that we use the GetPVSBit method to fetch the visibility bit in the portals PossibleVis array for the current leaf and if it is already set to 1, it means we have already visited this leaf before and have marked it as visible. When this is the case we simply return. If this is not the case however, we use the SetPVSBit method to mark the current leaf as visible in the portals PossibleVis array.

```
void CProcessPVS::PortalFlood( CPVSPortal * SourcePortal,
                               unsigned char PortalVis[],
                               unsigned long Leaf )
{
    CBSPLeaf * pLeaf = m_pTree->GetLeaf( Leaf );

    // Test the source portals 'Possible Visibility' list
    // to see if this leaf has already been set.
    if ( GetPVSBit( SourcePortal->PossibleVis, Leaf ) ) return;

    // Set the possible visibility bit for this leaf
    SetPVSBit( SourcePortal->PossibleVis, Leaf );

    // Increase portals 'Complexity' level
    SourcePortal->PossibleVisCount++;
```

Notice in the above code that after we have marked the leaf as visible to the portal we increment the portal's PossibleVisCount so that when the flood is complete, this member will contain all the visible leaves that were found. That is, the number of leaves that got wet by the flood fill. As described earlier, this PossibleVisCount member will be used as a measure of the portal's potential complexity and will be used by the core clipping process to determine which portal should have its PVS calculated next. Portals with a smaller PossibleVisCount will be selected for PVS calculation first.

Now that we have identified that the current leaf is visible it is now time to loop through all the portals stored in that leaf. If the portal is visible to the current source portal we will recur through that portal into its neighbor leaf. Otherwise we will skip the portal. Here is the remainder of the function code.

```
    // Loop through all portals in this leaf (remember the portal numbering
    // in the leaves match up with the originals, not our PVS portals )
    for ( ULONG i = 0; i < pLeaf->PortalIndices.size(); i++)
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;

        // If we can't pass through this portal then continue to next portal
        if ( !PortalVis[ PortalIndex ] ) continue;

        // Flood fill out through this portal
        PortalFlood( SourcePortal, PortalVis, GetPVSPortal( PortalIndex )->NeighbourLeaf );
```

```
    } // Next Leaf Portal
}
```

Notice that because we are accessing the portal information from the leaf of the BSP tree which stores one two-way portal instead of a pair of one-way portals, we need to multiply the leaf's portal index by 2 to get the index into the one-way portal array of the first in the pair of one-way portals that were generated for it. If the first one-way portal in the pair has a neighbor leaf equal to the leaf we are already in then this is obviously the one-way portal that points back into this leaf. This is obviously not the one-way portal that flows out of the neighbor leaf and therefore, the portal we are after must be the second in the pair so we increment the leaf index so that it addresses the correct one-way portal. Next we test to make sure this portal is visible by testing its corresponding byte in the PortalVis array, and if it is visible, the function calls itself to recur into the neighbor leaf of this portal.

We have now covered two of the processes that are invoked from the main Process method of this module. At this point, program flow will have been returned back to the Process method and not only will the one-way portals have been created and stored, each of these portals will contain a PossibleVis array describing a very approximate leaf PVS for that portal. The next method invoked from the Process method is the CalcPortalVis function. This is the function that is the doorway to the core PVS calculation process and the function that kick starts the anti-penumbra clipping function that ultimately calculates the ActualVis array (the real PVS) for each portal.

## The CalcPortalVis - CProcessPVS

Before we look at the CalcPortalVis method we must look at a structure that will be used to pass data from one recursion of the function to the next when performing the recursive clipping process. Because we will need to recur through the RecursePVS function many times (covered in a moment), it will be helpful to have a data structure where we can pass information from the previous recursion to the next. For example, when the RecursePVS function calls itself, the current generator portal needs to be passed into the next recursion as the target portal. We also need to be able to access the source portal at all times, regardless of what instance of the function we are in, so that we can set its visibility bits for each visible leaf we encounter. The structure is shown below.

***Excerpt from CProcessPVS.h***

```
typedef struct _PVSDATA                 // Structure to hold pvs processing data
{
        CPortalPoints   *SourcePoints;   // Current source portals points
        CPortalPoints   *TargetPoints;   // Current target portals points
        CPlane3          TargetPlane;    // The target's plane
        UCHAR           *VisBits;        // Visible Bits being calculated
} PVSDATA;
```

This structure is the transport mechanism with which to pass data about the generator portal that was selected in a previous iteration of the recursive function onto the next recursion. Since the previous generator portal becomes the target portal in the next recursion and is used to build a new anti-penumbra, this allows us to find a generator portal, fill out the information about it in one of these structures, and then pass on this structure to the next recursion so that we can access and use it as the

target portal during anti-penumbra generation. Let is discuss what the four members will be used for by the core PVS calculation process.

**CPortalPoints *SourcePoints;**
As we discuss in the text book, during the recursive clipping process the source portal will be the portal that is currently having is PVS calculated (ActualVis array populated). As we step recursively through generator portal after generator portal into new neighbor leaves, we need access to the source portals polygon data so that it can be used to create the anti-penumbra with the target portal at each step. When the RecursePVS function is first called, this will be a pointer to the source portal's polygon data.

However, we also discussed in the text book how at each step we build anti-penumbras between not only the source and target portals (allowing us to cull/clip generator portals) but also between the generator portal and the target portal which is used to clip a temporary copy of the source portal's polygon. By clipping the source portal and the generator portal at each step as small as possible, we assure that in the next recursive step a smaller anti-penumbra will be built allowing us to cull more generator portals from having to be visited. During the recursive process, this member will contain a copy of the source portal in its currently clipped state. That is, when at a given step in the process the source portal is clipped to the anti-penumbra and a new smaller source portal polygon is generated, that polygon is stored in this member so that it can be passed into the next recursion as the source portal geometry.

This is always a temporary copy of the source portal polygon from the first point it gets clipped. That is, although we set this to the CPortalPoints member of the source portal at the start of the process, we never want to clip or delete the original source portal as this will be needed for the calculation of other portals later in the process. Our hope is that as we step from leaf to leaf, the portal stored in this member will become smaller and smaller due to the ongoing clipping in each recursion until it generates a very small anti-penumbra in which no generator portals are contained and stopping the portal flow along that given path.

**CPortalPoints *TargetPoints;**
This member will be used to pass the geometry of the generator portal that has been stepped though in one recursion onto the next recursion where it will become the target portal and used to build the anti-penumbra planes.

As was the case with the source portal transport mechanism described above, a generator portal selected in one recursion of the function will be clipped to the current anti-penumbra in an attempt to make that generator as small as possible (or cull it completely) for the next recursion. The clipped copy of the generator portal is then stored in this member variable and passed into the next recursion of the function where it will be used as the target portal. By making both the source and target portals smaller and smaller with each generator portal that we step through, we will get a tighter and more accurate PVS calculated and raise the chances of being able to abort a from given path of portals earlier.

When the RecursePVS method is first called, this will be set to NULL as no target portal will have yet been selected. This allows us to identify the special case in the RecursePVS function where we have entered it for the very first time and we do yet have two portals with which to construct an anti-penumbra. In this first special case, no anti-penumbra clipping is performed. A generator portal is

simply selected and stored in the TargetPoints member. We then recur through that portal into its neighbor leaf where we enter the clipping process proper.

**CPlane3 TargetPlane;**
This will be used to store the plane of the target portal when input into the RecursePVS function. That is, this will be the plane of the generator portal that was selected in the previous recursion and the plane of the portal polygon stored in the TargetPoints member. As discussed in the accompanying text book, we will need this as we will need to clip the generator portal to the target portal's plane to make sure that a spanning case does not occur that could corrupt the anti-penumbra in the next recursion. The source portal will also be clipped to the generator portal's plane so all spanning case are eliminated. Remember, the only portion of the generator portal that should be visible is the portion that is contained in the front space of the source/target portal and vice versa.

The first time the RecursePVS function is called this will be set to the plane of the source portal. As we described above, the first call is a special case where a target portal has not yet been selected. This will allow us to make sure that the generator portal we selected (which will become the first target portal in the next recursion) will be clipped to the plane of the source portal as once again, the only portion of the target portal that should be visible to the source portal is the section that is contained in the source portal's front space.

**UCHAR *VisBits;**
This array will be used to accumulate and pass into the current recursion all leaves that can possibly be seen by the source portal given the current combination of portals we have stepped through up to this point. This allows us to very efficiently determine if there are any leaves to process down this path which might be visible to the source portal and if not, we can terminate this path of flow immediately. We know for example that at the start of the process each portal will have a bitset (PossibleVis) describing which leaves may be visible. However, we also know that this set can be refined by ANDing the bitsets of all portals we have stepped through up until this point. For example, imagine that in a 10 leaf level the PossibleVis array of the source portal has the following leaf bits set to 1.

Source Portal Vis - 1111100000

We can see at the start that only the first five leaves of the level could possibly be visible to the source portal as this was determined by the portal flood we performed earlier. Let us also assume that we next choose a target portal with which to build the anti-penumbra which had the following PossibleVis array calculated for it during the portal flood.

Target Portal Vis - 1110011100

We can see that the source portal has the possibility of being able to see into leaves 1 through 5 whilst the target portal has the possibility of seeing into leaves 1 through 3 and 6 through 8. Therefore, when we build an anti-penumbra between these two portals we are actually asking the question, "How much can we see from the source portal through the target portal" which at an approximate level can be determined by ANDing the visibility sets

Current Vis - 1111100000 + 1110011100 = 1110000000

As you can see, before we even perform any clipping we can tell that there can only possibly be 3 leaves that we need to visit and can be seen through the combination of portals we are currently looking through, leaves 1 through 3. It doesn't stop there though.

For example, imagine that we next choose a generator portal that has the following PossibleVis array

Generator Vis - 0011111111

When we combine its visibility set with the current visibility information we have collected so far we get:-

Current Vis - 111000000
AND
Generator Vis - 001111111

New Current Vis - 001000000

As you can see, this tells us that there is only one leaf we are interested in visiting at this point in the process. That is, there is only one leaf that is visible from the source portal when looking through the target and generator portals. This process is ongoing. As we step from recursion to recursion we combine the possible visibility sets of each portal we step through so we can very quickly determine whether there are any leaves to visit down this current path of portals through which we have traveled. This allows us to bail from the path early instead of having to perform the anti-penumbra clipping for generator portals which we have already determined have neighbor leaves that could not possibly be visible through the combination of portals we have traveled to get to this point.

The VisBits member of this structure allows us to pass this combined bitset into the next recursion where it will be combined with the next generator portals PossbleVis array and so on. That is, it inputs into the RecursePVS function a bitset that describes what leaves could possibly be visible given the portals we have traveled through to that point. When the RecursePVS function is first called, this will point to the source portal's PossibleVis array as we have not yet selected any target portals. For all future recursions this will contain the combination (the bitwise ANDing) of the source portals PossiblVis array with the PossibleVis arrays of all the portals we have walked through to get to the current leaf.

With this structure explained we can now look at the code to the CalcPortalVis function a section at a time and see how this structure is initially setup and passed into the first instance of the RecursePVS function.

In the first section of the code we test the compiler options for this module. If the option has not be set to enable a full PVS compile then it means the user does not wish us to perform the core time consuming anti-penumbra clipping tests at all and as such we should just use the PossibleVis arrays calculated for each portal in the portal flood fill as the actual PVS for each portal and return.

```
HRESULT CProcessPVS::CalcPortalVis()
{
    ULONG   i;
    HRESULT hRet;
    PVSDATA PVSData;

    // If we want to perform a quick vis (not at all accurate) we can
    // simply use the possible vis bits array as our pvs bytes.
     if ( !m_OptionSet.FullCompile )
     {
        for ( i = 0; i < GetPVSPortalCount(); i++ )
        {
            CPVSPortal * pPortal = GetPVSPortal( i );
             pPortal->ActualVis = pPortal->PossibleVis;

        } // Next Portal

        // We are finished here
        return BC_OK;

    } // End if !FullCompile
```

As you can see in the above code, if a full compile is not required we just loop through each portal and copy its PossibleVis array into its ActualVis array. As we know, the ActualVis array is where our compiler will expect the real PVS data of the portal to exist after the core clipping process has been performed. As we do not wish to perform this core clipping process we simply copy over the PossibleVis arrays into the ActualVis arrays of each portal so that the bitset is stored in the portal where the PVS exporter will expect to find it. We then return because our job is already done.

> **Note**: This option is not to be used for commercial output. The PVS set generated using such an option will be extremely over generous and run time application performance will suffer. This option is really for the developers of the project to perform a very quick compile to test the various assets of their game without having to wait long periods for PVS calculation every time they update the geometry of their level.

Now it is time to start the core PVS calculation process and as such we output to the logging device that PVS calculation is about to being. As the PVS process is essentially one of just looping through each portal and calculating its ActualVis array (that made it sound deceptively easy) we set the range of the logger to the portal count so that it can be increased with each portal we process.

```
    // *************************
    // * Write Log Information *
    // *************************
    if ( m_pLogger )
    {
        m_pLogger->LogWrite( LOG_PVS, 0, true, _T("Full PVS compile progress \t\t\t- " ) );
        m_pLogger->SetRewindMarker( LOG_PVS );
        m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
        m_pLogger->SetProgressRange( GetPVSPortalCount() );
        m_pLogger->SetProgressValue( 0 );
    }
    // ************************
    // *    End of Logging    *
    // ************************
```

Next we make sure that the PVSData structure that we pass into the initial call to the RecursePVS function is initialized to zero and then we set up a loop to iterate through each portal.

```
    try
    {
        // Clear out our PVSData struct
        ZeroMemory( &PVSData, sizeof(PVSDATA) );

        // Lets process those portal bad boys!! ;)
        for ( i = 0; i != -1; i = GetNextPortal() )
        {
            CPVSPortal * pPortal = GetPVSPortal( i );

            // Update Progress
            if (!m_pParent->TestCompilerState()) throw BC_CANCELLED;
            if (m_pLogger) m_pLogger->UpdateProgress();
```

Notice that we use the GetNextPortal method to choose the next portal in the list to have its PVS calculated. We will have a look at this function in a moment but as discussed earlier, we wish to process least complex portals first so this function simply returns the index of a portal that has not yet been processed and has the smallest number of visible leaves in its PossibleVis array. We then use this index to fetch the next portal to be processed (pPortal) which will become the source portal in the next call to the RecursePVS function. We also update the logger's progress as a new portal is about to have its PVS calculated.

Our next task is to populate the members of the PVSData structure that will be passed into the first recursion of the function. As explained above, we should initially set the SourcePoints member to point to the source portal's polygon geometry (its CPortalPoints structure) and should set the VisBits pointer to point at the PossibleVis array of the source portal. The Plane of the source portal is also retrieved and stored in the TargetPlane member. Although this last step seems a little strange, by storing the source portal's plane in the TargetPlane member for the first call to the RecursePVS function, it means the target portal that we do select in that function will be clipped to the source portals plane so that any spanning cases are eliminated.

```
            // Fill our our initial data structure
            PVSData.SourcePoints    = pPortal->Points;
            PVSData.VisBits         = pPortal->PossibleVis;
            GetPortalPlane( pPortal, PVSData.TargetPlane );
```

Notice that we do not set the TargetPoints member of the PVSData structure in the above section of code. This is left at NULL so that we can detect in the RecursePVS function that it is the first recur of the function and as such, no anti-penumbra has to be created. We can just choose a generator portal and recur into its neighbor leaf.

Because the current portal we are processing is about to have its ActualVis array calculated, we had better allocate the memory for it now so that it is large enough to store a bitset that represents the visibility information for each leaf and we should initially set all the bits in this array to zero so that they are all initially invisible. It will be the RecursePVS function that will set the bits to 1 in this array when it finds a leaf that is visible.

```
            // Allocate the portals actual visibility array
            pPortal->ActualVis = new UCHAR[ m_PVSBytesPerSet ];
            if (!pPortal->ActualVis) throw std::bad_alloc(); // VC++ Compat

            // Set initial visibility to off for all leaves
            ZeroMemory( pPortal->ActualVis, m_PVSBytesPerSet );
```

Now we call the RecursePVS function to calculate the PVS for this portal. We pass in the neighbor leaf index as the first parameter as this is the leaf for which the source portal first leads into and is where the recursive procedure should begin for this portal. As the second parameter we pass in the source portal itself. This will be needed by the RecursePVS function so that it can set bits to 1 in its ActualVis array each time it locates a visible leaf. As the final parameter we pass in the PVSData structure. When this function returns, the PVS for the portal will be stored in its ActualVis array so we set the status of this portal to PS_PROCESSED. This will assure that the GetNextPortal method used to select the next portal to be processed in the loop will not select this portal again.

```
            // Step in and begin processing this portal
            hRet = RecursePVS( pPortal->NeighbourLeaf, pPortal, PVSData );
            if ( FAILED( hRet ) ) throw hRet;

            // We've finished processing this portal
            pPortal->Status = PS_PROCESSED;

        } // Next Portal

    } // End Try Block
```

At this point the PVS for each portal has been calculated and is stored in their ActualVis arrays. All that is let to do is inform the logger to output success before returning from the function.

```
    // Catch all failures
    catch (std::bad_alloc)
    {
        // Failed to allocate
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End if

    catch ( HRESULT& e )
    {
        // Arbitrary Error
        if ( m_pLogger && FAILED(e) ) m_pLogger->ProgressFailure( LOG_PVS );
        return e;

    } // End if

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

Before we look at the RecursePVS function which will undoubtedly require heavy discussion, we will first examine the GetNextPortal method of this module that was used in the above code to select the next portal to have its PVS calculated.

## GetNextPortal - CProcessPVS

This small function has the simple task of determining which portal in the list should have its PVS calculated next. The function loops through each portal in the list searching for the portal with the lowest PossbleVisCount that has not yet had its PVS calculated. That is, it is looking for the least complex portal in the list whose status is still PS_NOTPROCESSED. Once the portal is located, its status is set to PS_PROCESSING prior to its index being returned. This is the portal that will have its PVS calculated next and is therefore the portal that we are currently processing. The complete code is shown below.

```
ULONG CProcessPVS::GetNextPortal( )
{
    CPVSPortal * pPortal;
    long PortalIndex = -1, Min = 999999, i;

    // Loop through all portals
    for ( i = 0; i < (signed)GetPVSPortalCount(); i++ )
    {
        pPortal = GetPVSPortal(i);

             // If this portal's complexity is the lowest and it has
        // not already been processed then we could use it.
        if ( pPortal->PossibleVisCount < Min && pPortal->Status == PS_NOTPROCESSED)
        {
                    Min = pPortal->PossibleVisCount;
                    PortalIndex = i;
        } // End if Least Complex

     } // Next Portal

     // Set our status flag to currently being worked on =)
     if ( PortalIndex > -1) GetPVSPortal( PortalIndex )->Status = PS_PROCESSING;

    // Return the next portal
    return PortalIndex;
}
```

## RecursePVS - CProcessPVS

We finally get to the function that contains the real meat of the process. This function calls itself recursively until the PVS set of the source portal has been calculated. The function takes three parameters. The first is the index of the leaf that we have currently stepped into which will be the neighbor leaf of the source portal the first time it is called. This leaf will be marked as visible in the source portals PVS. As we have stepped into this leaf it is obviously visible from the source portal. The second parameter is the source portal itself which is in the process of having its PVS calculated by this function. The third parameter is a PVSData structure which in the normal case will contain information about the target portal that is to be used in this function (the generator portal selected in the previous recursion) and the visibility buffer of all the portals we have stepped through so far which will be used for early-out determination.

```
HRESULT CProcessPVS::RecursePVS( ULONG Leaf, CPVSPortal * SourcePortal, PVSDATA & PrevData )
{
    ULONG           i,j;
    bool            More;
    ULONG          *Test, *Possible, *Vis;

    PVSDATA         Data;
    CPVSPortal     *GeneratorPortal;
    CPlane3         ReverseGenPlane, SourcePlane;
    CPortalPoints  *SourcePoints, *GeneratorPoints, *NewPoints;

    // Store the leaf for easy access
    CBSPLeaf * pLeaf = m_pTree->GetLeaf( Leaf );

    // Mark this leaf as visible
    SetPVSBit( SourcePortal->ActualVis, Leaf );
```

The first thing the function does is used the passed leaf index to fetch the leaf structure from the BSP tree. It then sets this leaf's bit to one in the source portal's ActualVis array. This leaf is visible from the source portal and has now been added to its PVS.

Notice in the above code that we instantiate on the stack a PVSData structure (in addition the one passed into the function as the third parameter). This new PVSData structure (imaginatively named Data) will be used to carry the generator portal information and combined visibility bitset from this function into the next recursion. That is, the Data local variable will be what we pass into the next call to the function as the third parameter and is the structure whose data we must fill out before we do so.

First, we allocate the structure a new VisBits array large enough to hold a bit for each leaf. This array will be used to store the result of combining the VisBits array passed into the function (PrevData.VisBits) with the PossibleVis array of the generator portal that we select in this function.

```
    // Allocate our current visibility buffer
    Data.VisBits = new UCHAR[ m_PVSBytesPerSet ];
    if (!Data.VisBits) throw std::bad_alloc(); // VC++ Compat
```

For ease of access, we also assign some local variables to structure members we are going to need to access frequently. We can see in the following code that the Possible local variable is used to point at the Data.VisBits array we just allocated and the Vis local variable is used to point at the PVS set (ActualVis array) of the source portal. This allows us to access these two buffers in short hand.

```
    // Store data we will be using inside the loop
    Possible    = (ULONG*)Data.VisBits;
    Vis         = (ULONG*)SourcePortal->ActualVis;
    GetPortalPlane( SourcePortal, SourcePlane );
```

Notice in the above code how we then fetch the plane of the source portal and store it in the SourcePlane local variable. Remember once again why the GetPortalPlane method is used for this task. It takes care of flipping the direction of the portal's node plane (the returned plane) if it faces into an opposing half space to the portals neighbor leaf.

The rest of the function is contained inside a loop that iterates through every portal in the current leaf and processes it for generator portal candidacy. Remember that the leaf structure in the BSP tree (which

we just retrieved above) will contain the indices of the two-way portals that reside in that leaf. As we did before, we must multiply this index by 2 so that we get the index of the first in the pair of one-way portals that were generated by that two-way portal. We then test to see if the first portal in the pair does not have a neighbor leaf equal to the leaf we are currently in because if it does, this is the one-way portal that flows into the current leaf and not out of it. When this is the case we know that the next portal in the array must be the other portal in the pair which faces out of the current leaf and into the neighbor leaf we may be interested in recurring into if this generator portal is not completely clipped away later in the function.

```
    // Check all portals for flow into other leaves
    for ( i = 0; i < pLeaf->PortalIndices.size(); i++ )
    {
        // Find correct portal index (the one IN this leaf (not Neighbouring))
        ULONG PortalIndex = pLeaf->PortalIndices[ i ] * 2;
        if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == Leaf ) PortalIndex++;

        // Store the portal for easy access
        GeneratorPortal = GetPVSPortal( PortalIndex );
```

Now that we have the generator portal we can see how the PrevData.VisBits array that is passed into this function is used. This contains a leaf bitset that has been generated by ANDing all the PossibleVis arrays of the portals we have traveled through to get to this current leaf. Therefore, only leaves with their bits set to one in this array are leaves that are visible from the source portal when looking through all the portals we have stepped through to get to this leaf. Therefore, we test to see if the generator portal's neighbor leaf has its bit set to one in this bitset. If it does not, it means the leaf on the opposing side of this portal is not possibly visible from the source portal so we have no interest in processing this generator portal any further. As you can see, when this is the case we simply skip this generator portal and continue to the next iteration of the generator portal loop.

```
        // We can't possibly recurse through this portal if it's neighbour
        // leaf is set to invisible in the target portals PVS
        if ( !GetPVSBit( PrevData.VisBits, GeneratorPortal->NeighbourLeaf ) ) continue;
```

Now it is time to combine the PrevData.VisBits array with the visibility array of the generator portal so that we can store the resulting bit set in Data.VisBits (aliased via the 'Possible' pointer) and send that onto a future recursion. It is this section of code that performs the bitset accumulation we have discussed in each recursion. What we are going to do is take the visibility set we have been passed into this function (PrevData.VisBits) and further refine it by the generator portal's PossibleVis bits. But wait!!! If the generator portal has already had its PVS generated such that its status is equal to PS_PROCESSED, we can refine this bit set even further by using its ActualVis array instead of its PossibleVis array which will allow us to hopefully carve of a great many more visible leaves out of the equation. Therefore, in the next section of code we first test the status of the generator portal and if it has been processed, we assign the local Test pointer to point at its ActualVis array. If not, then Test is assigned to point at its PossibleVis array.

```
        // If the portal can't see anything we haven't already seen, skip it
        if ( GeneratorPortal->Status == PS_PROCESSED )
            Test = (ULONG*)GeneratorPortal->ActualVis;
        else
            Test = (ULONG*)GeneratorPortal->PossibleVis;
```

Now we will loop through PrevData.VisBits four bytes at a time (using a long pointer which is why we padded these visibility arrays to a 32 bit boundary) and will AND it with the visibility buffer of the generator portal 'Test'. We will store the result in 'Possible' which is a pointer to Data.VisBits which we know is the PVSData structure that will be passed into the next recur of this function. As we AND these arrays together four bytes at a time and store them in 'Possible', we will also perform a bitwise AND with the NOT of the source portals PossibleVis array (Aliased by the local Vis pointer). This allows us to quickly detect if there are any bits set in Possible which are also set in Vis and therefore means there are still leaves in Possible that might be visible from the source portal and will need to be visited. When this is the case we set the More local Boolean variable to true

```
        More = false;
        // Check to see if we have processed as much as we need to
        // this is an early out system. We check in 32 bit chunks to
        // help speed the process up a little.
        for ( j = 0; j < m_PVSBytesPerSet / sizeof(ULONG); j++ )
        {
            Possible[j] = ((ULONG*)PrevData.VisBits)[j] & Test[j];
            if ( Possible[j] & ~Vis[j] ) More = true;

        } // Next 32 bit Chunk

        // Can we see anything new ??
        if ( !More ) continue;
```

The little loop above can be a bit of a brain teaser at first but it is essentially just creating the combined VisBits array that will be sent into the next recursion and at the same time determining whether there is anything left to see by continuing down this path. Outside the loop you can see that if More is not set to true, if means there are no bits set to one in the Possible array that are also set to one in the source portal's PossibleVis array and as such, there is nothing of interested on the other side of this generator portal so there is not need to recur through it into the neighbor leaf. When this is the case we simply skip any further processing of the generator portal and continue to the next iteration of the loop.

In the next step we fetch the plane of the generator portal and store it in the Data.TargetPlane so that it can be passed to the next recursion as the target portal's clip plane. Remember, the generator portal selected in this recursion will become the next target portal when we recur through it into its neighbor leaf.

```
        // The current generator plane will become the next recursions target plane
        GetPortalPlane( GeneratorPortal, Data.TargetPlane );
```

Our next test is to make sure that the generator portal is not ON_PLANE with the target portal that has been passed into this function in the PrevData parameter. If it is, then they cannot see each other and we can skip this generator portal. For this test we are using a generator plane with a reversed normal and are comparing this normal against the normal the normal of the target portal (PrevData.TargetPlane). We do

this because we need to check that the portal we have just entered this leaf through is not on the same plane as any we are about to leave this leaf through. A target portal can not see a generator portal that is on the same plane as it and stepping through such a portal could carry us back into the leaf that we just exited in the previous recur.

```
        // We can't recurse out of a coplanar face, so check it
        ReverseGenPlane.Normal   = -Data.TargetPlane.Normal;
        ReverseGenPlane.Distance = -Data.TargetPlane.Distance;
        if ( ReverseGenPlane.Normal.FuzzyCompare( PrevData.TargetPlane.Normal, 0.001f ))
              continue;
```

Our next task is to clip the generator portal to the plane of the source portal so that any portion of the generator portal that is contained in the back space of the source portal is removed. If this operation completely clips away the generator portal we skip this generator portal as there seems to be no valid portion of it in front of the source portal.

```
        // Clip the generator portal to the source. If none remains, continue.
        GeneratorPoints = GeneratorPortal->Points->Clip( SourcePlane, false );
        if ( GeneratorPoints != GeneratorPortal->Points )
             FreePortalPoints( GeneratorPortal->Points );

        if (!GeneratorPoints) continue;
```

In this next section of code we see some special case code which is only executed the first time the RecursePVS function is called (when PrevData.TargetPoints will equal NULL). When this is the case we do not wish to perform any anti-penumbra clipping as we are really just trying to select the first target portal. As you can see, in this instance we simply copy over the source points passed into the function (PrevData.SourcePoint) into the PVSData structure that will be passed into the next recursion (Data.SourcePoints) and also copy over the generator portal into the TargetPoints member of this structure also so that it will become the target portal in the next recursion. We then call the RecursePVS function to recur into the neighbor leaf of the generator portal (which is really the first target portal in this instance) and on function return skip to the next generator portal in the leaf to be processed.

```
        // The second leaf can only be blocked if coplanar
        if ( !PrevData.TargetPoints )
        {
            Data.SourcePoints = PrevData.SourcePoints;
            Data.TargetPoints = GeneratorPoints;
            RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );
            FreePortalPoints( GeneratorPoints );
            continue;

        } // End if Previous Points
```

Remember that the above conditional code is only executed in the first instance of the function when no previous target portals exist and need to be found.

The next thing we do is clip the generator portal to the target portal's plane. The section of the generator portal that should be visible to the source portal is that section located in the front space of the target portal and as such, can be seen when looking through the back of the target portal. If none of the generator portal survives the clip then it is not visible to the source portal and we do not have to step

through it into its neighbor leaf. When this is the case we process the generator portal no further and simply skip to the next generator portal in the loop that needs to be processed.

```
        // Clip the generator portal to the previous target. If none remains, continue.
        NewPoints = GeneratorPoints->Clip( PrevData.TargetPlane, false );
        if ( NewPoints != GeneratorPoints ) FreePortalPoints( GeneratorPoints );
        GeneratorPoints = NewPoints;
        if (!GeneratorPoints) continue;
```

The source portal should also be clipped to the generator portals plane. However, we need to make sure that the only portion of the source portal that survives is the portion that is located behind the plane of the generator portal and not in front as it usually the case with a clipping routine. The source portal should only ever be able to see through the back of a generator portal and a source portal that spans the generator portals plane clearly violates that. We need to flip our clipping operation so that is clips away any portion of the source portal that is located in the front space of the generator portal's plane. Fortunately, we already calculated a reversed generator plane above so we can re-use it here. By passing in the reversed generator plane into our clip function we will remove the section of the source portal that is in the back space of this plane, which is really the section that is in the front space of the real generator portal's plane. As you can see, we make a new copy of the source portal geometry from the source portal geometry passed into the function (PrevData.SourcePoint) which my have been clipped many times before and we then clip it to the reversed generator plane.

```
        // Make a copy of the source portals points
        SourcePoints = new CPortalPoints( PrevData.SourcePoints, true );

        // Clip the source portal
        NewPoints = SourcePoints->Clip( ReverseGenPlane, false );
        if ( NewPoints != SourcePoints ) FreePortalPoints( SourcePoints );
        SourcePoints = NewPoints;

        // If none remains, continue to the next portal
        if ( !SourcePoints ) { FreePortalPoints(  GeneratorPoints ); continue; }
```

As the above code shows, if none of the source portal survived the clip it means the source portal can see no portion of the generator portal's back face and as such no portal flow can exist. When this is the case we skip any further processing of this generator portal and continue to the next iteration of the loop where we will test the next generator portal in this leaf which needs to be tested.

At this point we know we have a source portal and a generator portal which although have been clipped to each others plane, have portal flow between them in the loosest sense. Our next task is to perform the clipping of the source and generator portals to the anti-penumbra. The ClipToAntiPenumbra function is the method that creates the anti-penumbra and performs the clipping of the passed portal to its planes. As discussed earlier, there are a possibly four anti-penumbras that we can build and clip to and how many we perform depends on the number of clip tests that have been enabled in the PVSOPTIONS structure. The clip tests are performed as shown below:-

- Clip Test 1 : Build Anti-Penumbra from **source portal** to **target portal** and clip the **generator portal** to its planes.
- Clip Test 2 : Build Anti-Penumbra from **target portal** to **source portal** and clip the **generator portal** to its planes.
- Clip Test 3 : Build Anti-Penumbra from **generator portal** to **target portal** and clip the **source portal** to its planes.
- Clip Test 4 : Build Anti-Penumbra from **target portal** to **generator portal** and clip the source portal to its planes.

After each clipping operation, if none of the portal that was clipped survived, the source and generator portals can not see each other through the target portal meaning we need process this generator portal no further and should not recur through it into the neighbor leaf. Instead, we simply continue on to the next iteration of the loop and process the next generator portal in the leaf. Here is the code that performs the four clip tests. We will look at the ClipToAntiPenumbra method next.

```
        // Lets go Clipping :)
        if ( m_OptionSet.ClipTestCount > 0 )
        {
            GeneratorPoints = ClipToAntiPenumbra( SourcePoints,
                                                  PrevData.TargetPoints,
                                                  GeneratorPoints,
                                                  false );

            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 1 Clip Test

        if ( m_OptionSet.ClipTestCount > 1 )
        {
            GeneratorPoints = ClipToAntiPenumbra( PrevData.TargetPoints,
                                                  SourcePoints,
                                                  GeneratorPoints,
                                                  true );

            if (!GeneratorPoints) { FreePortalPoints( SourcePoints ); continue; }

        } // End if 2 Clip Tests

        if ( m_OptionSet.ClipTestCount > 2 )
        {
            SourcePoints = ClipToAntiPenumbra( GeneratorPoints,
                                               PrevData.TargetPoints,
                                               SourcePoints,
                                               false );

            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 3 Clip Test

        if ( m_OptionSet.ClipTestCount > 3 )
        {
            SourcePoints = ClipToAntiPenumbra( PrevData.TargetPoints,
```

```
                                        GeneratorPoints,
                                        SourcePoints,
                                        true );

            if (!SourcePoints) { FreePortalPoints( GeneratorPoints ); continue; }

        } // End if 4 Clip Test
```

The first two parameters to the ClipToAntiPenumbra function are the portals between which the anti-penumbra will be created. The third parameter is where the portal that should be clipped is passed.

The fourth parameter to the function is a Boolean that describes whether the portal should be clipped to the back of the anti-penumbra planes or the front. This is necessary because the direction of the plane normals being built from source to target will be opposite to that of the normals of the plane constructed when going from target to source as is shown in figure 17.6.

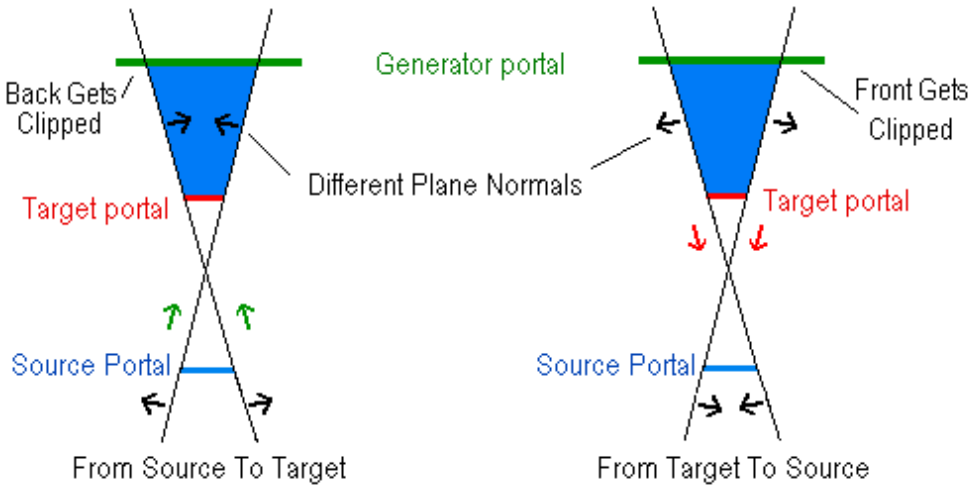



**Figure 17.55**

The leftmost image shows a 2D representation of the planes generated when the anti-penumbra is being constructed from the vertices of the source portal to the edges of the target portal. We can see that on the opposite side of the target portal the visible region is bounded by planes that face inwards and as such, any portion of the portal that is to be clipped that is located in the back space of any anti-penumbra plane is clipped away. In the right most image we see that when the direction of the anti-penumbra is reversed and is instead constructed from the target portal to the source portal, the visible region on the opposite side of the target portal is now bounded by outward facing planes and as such, the function should clip away any section of the generator portal that is located in the front space of any of these planes. That is why we flick this Boolean switch with each call in the above section of code. We are informing the ClipToAntiPenumbra method as to whether the portal should be clipped to the back or front spaces of the anti-penumbra's clip planes respectively.

If we survive the clip tests performed in the previous section of code then it means the source portal can clearly see the generator portal and as such, we should recur through the generator portal into its neighbor leaf. Before doing that we store the new clipped source portal polygon and the clipped generator portal polygon in the PVSData structure (Data) and then pass it into the RecursePVS function to recur into the next leaf.

Below we show the remainder of the function that performs this task and shows the clean up of the temporarily clipped source and generator portals inside the bottom of the loop.

```
        // Store data for next recursion
        Data.SourcePoints = SourcePoints;
        Data.TargetPoints = GeneratorPoints;

        // Flow through it for real
        RecursePVS( GeneratorPortal->NeighbourLeaf, SourcePortal, Data );

        // Clean up
        FreePortalPoints( SourcePoints );
        FreePortalPoints( GeneratorPoints );

    } // Next Portal

    // Clean up
    if (Data.VisBits) delete []Data.VisBits;

    // Success
    return BC_OK;
}
```

When we reach the bottom of this loop we will have processed every generator portal in this leaf and will have calculated its contribution to the source portal's ActuaVis array. Before returning we release the Data.VisBits array which we allocated earlier.

## ClipToAntiPenumbra - CProcessPVS

The ClipToAntiPenumbra function has the task of finding all separating planes that divide the source and target portals into opposing half spaces. Such a plane is a valid anti-penumbra plane which is used to clip the generator portal.

> **Note:** In this function we are using the term generator portal to refer to the portal passed in as the third parameter to this function. However, as we saw in the above code, this may be a pointer to either the actual source or generator portal depending on which clip test is being performed.

This function first sets up a loop to iterate through every edge in the source portal as shown below.

```
CPortalPoints * CProcessPVS::ClipToAntiPenumbra( CPortalPoints * Source,
                                                 CPortalPoints * Target,
                                                 CPortalPoints * Generator,
                                                 bool ReverseClip )
{
    CPlane3         Plane;
    CVector3        v1, v2;
    float           Length;
    ULONG           Counts[3];
    ULONG           i, j, k, l;
    bool            ReverseTest;
    CPortalPoints  *NewPoints;

    // Check all combinations
    for ( i = 0; i < Source->VertexCount; i++ )
    {
```

```
        // Build first edge
        l = ( i + 1 ) % Source->VertexCount;
        v1 = Source->Vertices[l] - Source->Vertices[i];
```

Loop variable ‘i’ describes the index of the first vertex in the edge we are testing and the ‘l’ local variable describes the index of the next vertex forming the edge. This is calculated as being i+1 with a modulus performed with the vertex count of the source polygon so that ‘l’ will wrap around to the first vertex for the last edge of the portal. Vector v1 is then calculated by subtracting vertex position l from i thus generating the current edge vector in the source portal we wish to process.

Now that we have an edge in the source portal we need to create a plane with that edge and every vertex in the target portal. Next we set up a loop to loop through each vertex in the target portal and create an additional vector v2. This is a vector from the one of the source vertices in the edge and the current vertex in the target portal we are processing. Vectors v1 and v2 are now vectors tangent to a plane generated from the source to the target portal so we will next perform the cross product between these two vectors to retrieve the normal of that plane.

```
        // Find a vertex belonging to the generator that makes a plane
        // which puts all of the vertices of the target on the front side
        // and all of the vertices of the source on the back side
        for ( j = 0; j < Target->VertexCount; j++ )
        {
            // Build second edge
            v2 = Target->Vertices[ j ] - Source->Vertices[ i ];
            Plane.Normal = v1.Cross( v2 );
```

Next we record the length of the returned normal (which has not yet been normalized) and if the length is found to be zero (with tolerance) it means we have an invalid plane so will continue to the next iteration of the loop where we will process the next target vertex.

```
            // If points don't make a valid plane, skip it
            Length = Plane.Normal.x * Plane.Normal.x +
                     Plane.Normal.y * Plane.Normal.y +
                     Plane.Normal.z * Plane.Normal.z;
            if ( Length < 0.1f ) continue;
```

If we get this far it means that we have a valid plane and as such we will normalize the plane normal and will calculated the plane’s distance from the origin by dotting the plane normal with the target vertex (which is a point known to be on the plane).

```
            // Normalize the plane normal
            Length = 1 / sqrtf( Length );
            Plane.Normal *= Length;

            // Calculate the plane distance
            Plane.Distance = -Target->Vertices[ j ].Dot( Plane.Normal );
```

At this point we have a valid plane so our next step is to test if this is a separating plane. That is, if it has the source portal completely contained in one half space and the target portal in the other. If not then this is not a valid an anti-penumbra plane. Figure 17.7 reminds us of what a separating plane looks like by

showing us both a separating plane and a non-separating plane. Only the plane generated in the leftmost image is a valid anti-penumbra clip plane and will be used to clip the generator portal.

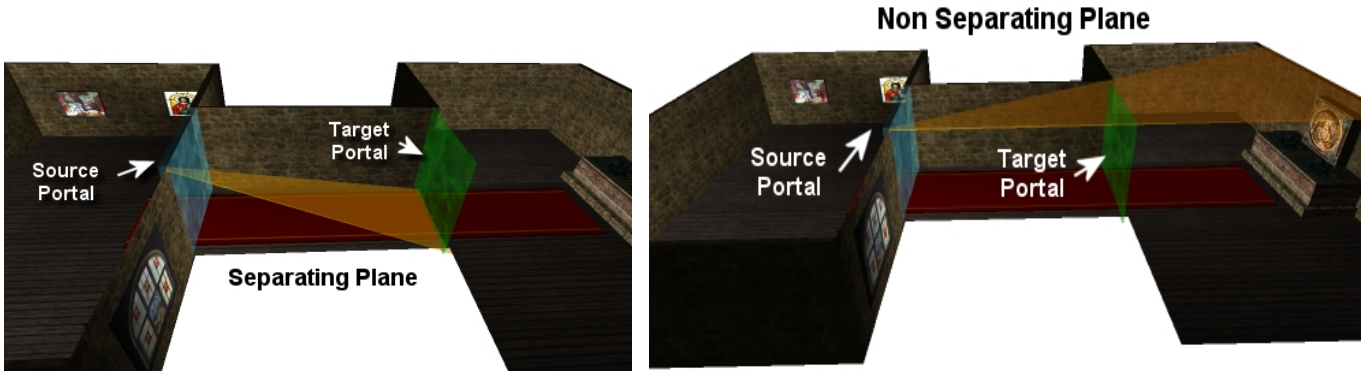



**Figure 17.7**

Our first task in achieving this goal is to loop through each vertex in the source portal and classify it against our plane. As it was one of the source portal's vertices that was used to generate this plane, it is impossible to have vertices in the source portal located in both half spaces of the plane. Therefore, in this loop we are searching for the first vertex that is found in either the front or back half spaces at which point we can break. We know that if one of the vertices is contained in the front space of the plane then the entire polygon must be and likewise for the back half space. If the source portal is found to be located in the back half space of the plane we set the ReverseTest Boolean to false before breaking so that we know that when we test the target portal we are wishing it to be contained in the front half space. Alternatively, this Boolean is set to true if the source portal is found to be located in the front space of the plane meaning we wish to find the target portal located in the back space.

```
            // Find out which side of the generated separating plane has the source portal
            ReverseTest = false;
            for ( k = 0; k < Source->VertexCount; k++ )
            {
                // Skip if it matches other verts
                if ( k == i || k == l ) continue;

                // Classify the point
                CLASSIFYTYPE Location = Plane.ClassifyPoint( Source->Vertices[ k ] );
                if ( Location == CLASSIFY_BEHIND )
                {
                    // Source is on the negative side, so we want all pass
                    // and target on the positive side.
                    ReverseTest = false;
                    break;

                } // End If Behind
                else if ( Location == CLASSIFY_INFRONT )
                {
                    // Source is on the positive sode, so we want all pass
                    // and target on the negative side.
                    ReverseTest = true;
                    break;

                } // End if In Front

            } // Next Source Vertex
```

As the above loop has no conditional to deal with the on-plane case, if all its vertices are co-planar with the candidate plane then the loop will be allowed to carry out to its conclusion. This means outside the loop, the loop variable ‘k’ will be equal to the vertex count of the source portal. This tells us that the source portal is co-planar and as such, this plane could not possibly be a separating plane so we should skip it and continue on to test the next vertex in the target portal.

```
            // Planar with the source portal ?
            if ( k == Source->VertexCount ) continue;
```

In the next section we will loop through each vertex in the target portal and will classify it against the plane. Before doing so we test the value of the ReverseTest local Boolean and flip the direction of the plane normal if it is set to true. This allows us to treat the source portal as having existed in the back space of the plane (regardless of where it was actually located) which means in this code we are searching for a portal that has its vertices contained in the front space of the plane.

```
            // Flip the normal if the source portal is backwards
            if ( ReverseTest ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }

            // If all of the pass portal points are now on the positive
            // side then this is the separating plane.
            ZeroMemory( Counts, 3 * sizeof(ULONG) );
            for ( k = 0; k < Target->VertexCount; k++ )
            {
                // Skip if the two match
                if ( k == j ) continue;

                // Classify the point
                CLASSIFYTYPE Location = Plane.ClassifyPoint( Target->Vertices[ k ] );
                if ( Location == CLASSIFY_BEHIND )
                    break;
                else if ( Location == CLASSIFY_INFRONT )
                    Counts[0]++;
                else
                    Counts[2]++;

            } // Next Target Vertex

            // Points on the negative side ?
            if ( k != Target->VertexCount ) continue;

            // Planar with separating plane ?
            if ( Counts[0] == 0 ) continue;
```

As you can see, as soon as we find a vertex that is situated behind the plane it means this portal is in the same half space of the plane as the source portal (pay attention to the flipping of the plane depending on the result of ReverseTest that allows us to make that determination). This means we break instantly as this is not a separating plane. If any vertices are found to exist in the front space of the plane then we increase the value of Counts[0].

Outside the loop we can see that if loop variable ‘k’ is not equal to the vertex count of the target portal it means we broke from the loop early after finding a vertex in the same half space of the source portal. This means this plane can not possibly separate the source and target portals into two half spaces so we continue on to the next iteration of the loop and the next plane to test. We can also see that if Counts[0]

equal zero, it means we never incremented it in the loop meaning no vertex was every found to be in the front space of the plane either. This must mean the target portal is located on the candidate plane and as such, this can not possibly be a separating plane. Once again, if this is the case we skip any further processing of this plane and skip to the next iteration of the loop.

If we reach this point it means we have found a separating plane and the generator portal passed into the function should be clipped to it. Before doing so we test the value of the ReverseClip parameter remembering that if this is set to true then the caller would like us to clip away portal fragments that are in the front space of the separating plane instead of in the back space as is usually the case. We address this by simply flipping the direction of the clip plane prior to clipping the portal.

As the next and final section of this function shows, we clip the generator portal to the clip plane and if we find that a new portal was created by the clip procedure, the original portal (Generator) is released. We then assign the passed Generator portal pointer to point at the new clipped polygon returned from the clipping function instead. In the last line of the loop for this plane, we test that the Generator has not been assigned a value of NULL and if so, it means we must have completely clipped away the generator portal with this plane and as such, the generator portal is outside the anti-penumbra and NULL is returned.

```
            // Flip the normal if we want the back side
            if ( ReverseClip ) { Plane.Normal = -Plane.Normal;
                                 Plane.Distance = -Plane.Distance; }

            // Clip the target by the separating plane
            NewPoints = Generator->Clip( Plane, false );
            if ( NewPoints != Generator ) FreePortalPoints( Generator );
            Generator = NewPoints;

            // Target is not visible ?
            if (!Generator) return NULL;

        } // Next Target Vertex

    } // Next Source Vertex

    // Success!!
    return Generator;
}
```

The bottom of the function is only ever reached if some portion of the generator portal survived being clipped to all the separating planes generated in the above loops. This new clipped generator portal is returned from the function back to the caller.

# Where Are We?

At this point we have covered the core PVS generation procedure that was invoked from the CProcessPVS::Process method via a call to the CalcPortalVis method. When the CalcPortalVis method returns program flow back to the Process method, every portal will have had its PVS generated. Let us have another look at the CProcessPVS::Process method to remind us of what is let to do.

```
HRESULT CProcessPVS::Process( CBSPTree * pTree )
{
    HRESULT hRet;

    // Validate values
    if (!pTree) return BCERR_INVALIDPARAMS;

    // Validate Input Data
    if ( pTree->GetPortalCount() == 0 ) return BCERR_BSP_INVALIDTREEDATA;

    // Store tree for compilation
    m_pTree = pTree;

    // Calculate Number Of Bytes needed to store each leafs
        // vis array in BIT form (i.e 8 leafs vis per byte uncompressed)
    m_PVSBytesPerSet = (pTree->GetLeafCount() + 7) / 8;

    // 32 bit align the bytes per set to allow for our early out long conversion
    m_PVSBytesPerSet = (m_PVSBytesPerSet * 3 + 3) & 0xFFFFFFFC;

    // Retrieve all of our one way portals
    hRet = GeneratePVSPortals();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Calculate initial portal visibility
    hRet = InitialPortalVis();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Perform actual full PVS calculation
    hRet = CalcPortalVis();

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Export the visibility set to the final BSP Tree master array
    hRet = ExportPVS( pTree );

    if ( FAILED( hRet ) ) return hRet;
    if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    // Success
    return BC_OK;
}
```

As you can see, we have covered all methods in the process with the exception of the final one called ExportPVS. It is this method, which we will look at next, that is responsible for combining the PVS data

for each portal that resides in a given leaf to generate the PVS for that leaf in the BSP tree. This method will also compress the data using zero run length encoding and will store in the leaves of the BSP tree the index into which each leaf's visibility data begins in the master PVS data block

## ExportPVS - CProcessPVS

At the time this function is called each portal will contain its own PVS in its ActualVis array. However, our PVS rendering code will not be interested in what each of the portals can see and in fact, there is no need for the portals to even be saved to file. Our PVS renderer wants to know what each leaf can see and it is this function that takes care of building the master PVS data block that will be stored in the BSP tree and will describe the visibility sets for each leaf instead of each portal.

Now that we have the visibility sets for each portal, determining what each leaf can see is trivial. The PVS of a leaf is simply the accumulation of the PVS's of each portal that resides in that leaf and as such can be calculated with a simple loop at each leaf.

Let us cover the function a section at a time. The first section of the function writes out logging information to the PVS channel describing whether it is going to create a compressed leaf PVS or whether it will calculate the leaf PVS for the BSP tree as an uncompressed bit set (this is controller by a #define).

```
HRESULT CProcessPVS::ExportPVS( CBSPTree * pTree )
{
    UCHAR * PVSData = NULL;
    UCHAR * LeafPVS = NULL;
    ULONG   PVSWritePtr = 0, i, p, j;

    try
    {
        // ************************
        // * Write Log Information *
        // ************************
        if ( m_pLogger )
        {

            #if ( PVS_COMPRESSDATA )
                m_pLogger->LogWrite( LOG_PVS,
                                     0,
                                     true,
                                     _T("ZRLE compressing PVS data for export \t\t- " ) );
            #else
                m_pLogger->LogWrite( LOG_PVS,
                                     0,
                                     true,
                                     _T("Building final PVS data for export \t\t- " ) );
            #endif

            m_pLogger->SetRewindMarker( LOG_PVS );
            m_pLogger->LogWrite( LOG_PVS, 0, false, _T("0%%" ) );
            m_pLogger->SetProgressRange( pTree->GetLeafCount() );
            m_pLogger->SetProgressValue( 0 );
        }
        // ************************
        // *    End of Logging     *
```

```
        // ************************
```

Notice that the progress range of the logger in the above code is set to the number of leaves in the passed BSP tree. That is because this process will be complete once we have looped through each leaf in the BSP tree and have calculated its PVS and added it to a master PVS array.

Our next step is to allocate some memory that will be used to store the master PVS data array. This must be large enough to store the visibility for every leaf in the tree in compressed format. One might imagine that if m_PVSBytesPerSet contains the number of bytes needed to store a leaf's visibility information in uncompressed format, then we can allocate an array large enough to store LeafCount*m_PVSBytesPerSet bytes. This should always be large enough to store everything we need as it is large enough to store a visibility set for each leaf in uncompressed format. Is that correct?

Well, in practice yes that will almost definitely be the case and in fact this array will be much larger than we actually need once the data is compressed. That doesn't matter though because after we have compressed the data we can resize the array to its correct actual size. However, look at how we allocate this array and initialize its memory.

```
        // Reserve Enough Space to hold every leafs PVS set
        PVSData = new UCHAR[pTree->GetLeafCount() * (m_PVSBytesPerSet*2)];
        if (!PVSData) throw std::bad_alloc();

        // Set all visibility initially to off
        ZeroMemory( PVSData, pTree->GetLeafCount() * (m_PVSBytesPerSet*2));
```

Why are we multiplying m_PVSBytesPerSet by 2 before multiplying it with the tree's leaf count? Although incredibly unlikely, there is very slim chance that compressing our data using ZRLE could actually make it larger than in its uncompressed format. Imagine for example we had 6 bytes in each of our visibility sets and the 2nd, 4th and 6th bit were set to zero.

Leaf Set = N , 0 , N , 0 , N , 0

Imagining that N is some placeholder for a non zero byte in a leaf's PVS. We know that ZRLE encoding will try to compress runs of zero bytes by collapsing those runs in to two bytes. The first is the zero itself and the second is the byte that tells us how many bytes of zero the run represents. However, in the above scenario there are no runs of zeros and as such, an additional byte (the run byte) would be inserted after each zero describing a run of 1. Obviously this provides no benefit and actually would increase the size of our 6 byte visibility array to a 9 byte array.

Leaf Set = N , 0 , 1 , N , 0 , 1 , N , 0 , 1

Therefore we can see that in the most unusual circumstances we may generate a compressed PVS that is larger than its uncompressed counterpart and in the most appalling of circumstances, compressing the PVS could actually double its size.

> **Note:** it should be noted that the above scenario is very unlikely as the PVS by its very nature in an occluded environment will generate leaf based PVS sets where most of the other leaves of not visible and as such, huge runs of zeros will be compressed into only two bytes providing us with huge memory

savings in nearly every case. However, as it is ***possible*** that we could double its size by compressing it, we had better allocate the PVS data block large enough to handle it so that our compiler does not crash and burn mid way through the compression procedure.

With the master PVS data array allocated we will now allocate a temporary buffer that will be used by each leaf to accumulate and collect the combined ActualVis arrays of each portal contained in that leaf. This buffer should be large enough to store an uncompressed bit set for each leaf. That is, this buffer should be equal in size to the portal's ActualVis array.

```
        // Allocate enough memory for a single leaf set
        LeafPVS = new UCHAR[ m_PVSBytesPerSet ];
        if (!LeafPVS) throw std::bad_alloc();
```

Now we will loop through each leaf in the tree. Inside the loop we will use the loop variable to fetch from the BSP tree the structure of the leaf we are currently processing. We also update the progress of the logger which is incremented on a per leaf bases.

```
        // Loop round each leaf and collect the vis info
        // this is all OR'd together and ZRLE compressed
        // Then finally stored in the master array
        for ( i = 0; i < pTree->GetLeafCount(); i++ )
        {
            CBSPLeaf * pLeaf = pTree->GetLeaf(i);

            // Update progress
            if (!m_pParent->TestCompilerState()) break;
            if ( m_pLogger ) m_pLogger->UpdateProgress( );
```

In the next step we will initialize the temporary LeafPVS buffer to zero prior to collecting the visibility sets of each of its portals into it. We also set the PVSIndex member of the leaf to that of the value stored in the PVSWritePtr local variable. We will see in a moment that this variable will be increased each time we compress the PVS data for a leaf and add it to the master PVS data block and as such, it will always describe the index into this array where the compressed PVS data of the leaf we are about to process will be placed into this array. We also set the visibility bit in the LeafPVS buffer for the current leaf we are processing as it can obviously see itself.

**Note:** PVSWritePtr is zero the first time this loop executes which describes leaf 0's PVS as starting at the very beginning of the master PVS data array

```
            // Clear Temp PVS Array Buffer
            ZeroMemory( LeafPVS, m_PVSBytesPerSet );
            pLeaf->PVSIndex = PVSWritePtr;

            // Current leaf is always visible
            SetPVSBit( LeafPVS, i );
```

Now we loop through every portal in this leaf and use the familiar methods to retrieve the index of the one-way portal that it represents.

```
            // Loop through all portals in this leaf
            for ( p = 0; p < pLeaf->PortalIndices.size(); p++ )
            {
```

```
                // Find correct portal index (the one IN this leaf)
                ULONG PortalIndex = pLeaf->PortalIndices[ p ] * 2;
                if ( GetPVSPortal( PortalIndex )->NeighbourLeaf == i ) PortalIndex++;
```

At this point we have a pointer to the current portal so we will loop through each byte in its ActualVis array and will bitwise OR it with the current contents of the LeafPVS buffer so that the necessary visibility bits are enabled.

```
                // Or the vis bits together
                for ( j = 0; j < m_PVSBytesPerSet; j++ )
                {
                        LeafPVS[j] |= GetPVSPortal( PortalIndex )->ActualVis[j];

                } // Next PVS Byte

            } // Next Portal
```

At this point the LeafPVS buffer contains the complete uncompressed PVS set for the current leaf that we are processing so our next task is to store this in the master PVSData array. If compression has not been enabled then we literally just copy the information stored in LeafPVS into the master data array at the location contained in PVSWritePtr and then increase the write pointer by the size of the LeafPVS array. This is done so that when we process the next leaf, it will contain an index to the location where its data should be copied to in the master PVS data array. However, if we have chosen to compress the data set then the data is added to the master PVSData array using a function called CompressLeafSet as shown below.

```
            #if ( PVS_COMPRESSDATA )

                // Compress the leaf set here and update our master write pointer
                PVSWritePtr += CompressLeafSet( PVSData, LeafPVS, PVSWritePtr );

            #else

                // Copy the data into the Master PVS Set
                memcpy( &PVSData[ PVSWritePtr ], LeafPVS, m_PVSBytesPerSet );
                PVSWritePtr += m_PVSBytesPerSet;

            #endif

        } // Next Leaf
```

The CompressLeafSet method will be discussed in a moment but for now just know that it is passed the master PVSData array as its first parameter as this is where the function will need to copy the compressed data into. As the second parameter we will pass the LeafPVS buffer which contains the uncompressed PVS for the current leaf. This is the data that the function will compress and copy into the PVSData array. As the third parameter we pass the write pointer index value which describes the starting location in the PVSData array where the new compressed data should be written to. As this function returns the size that the passed LeafPVS data array was ultimately compressed to, we can use this value to increment the write pointer value on function return so that it contains the index of the first byte after the block of data we have just added. This will be the starting location for the next leaf's compressed data and the byte at which its PVS data will begin in the master array.

At this point we will have compressed every leaf's PVS and stored it in the PVSData array and each leaf in the tree will also have had its PVSIndex member set so that it describes the index of the first byte in this array where its visibility information begins. We can now delete the LeafPVS buffer as it is no longer needed and can call the BSP tree's SetPVSData method (discussed earlier) so that it can make a copy of the PVSData array we haven just compiled.

```
        // Clean up after ourselves
        delete []LeafPVS;
        LeafPVS = NULL;

        // Pass this data off to the BSP Tree (data, size, compressed)
        if (FAILED(pTree->SetPVSData( PVSData,
                                      PVSWritePtr,
                                      PVS_COMPRESSDATA ))) throw std::bad_alloc();

        // Free our PVS buffer
        delete []PVSData;

        // If we're cancelled, bail
        if ( m_pParent->GetCompileStatus() == CS_CANCELLED ) return BC_CANCELLED;

    } // End Try Block

    catch (...)
    {
        // Clean up and return (Failure)
        if ( LeafPVS ) delete []LeafPVS;
        if ( PVSData ) delete []PVSData;
        if ( m_pLogger ) m_pLogger->ProgressFailure( LOG_PVS );
        return BCERR_OUTOFMEMORY;

    } // End Catch Block

    // Success!!
    if ( m_pLogger ) m_pLogger->ProgressSuccess( LOG_PVS );
    return BC_OK;
}
```

Recall that the CBSPTree::SetPVSData method will make a copy of the passed PVS data array and by passing in PVSWritePtr as the second parameter we also inform it of the final size of the compressed data. As it uses this to allocate its PVS array this means that whilst the PVSData array allocated in this function may have been allocated much larger than necessary, the actual array stored in the PVS tree will be the correct size. The CBSPTree::SetPVSData method copies over all the PVS data into its own array and therefore, on function return we can delete the local PVSData array as it is no longer needed. The third parameter to the SetPVSData method simply informs the function that the data is compressed which the run time component will need to know when reading the PVS data at render time.

## CompressLeafSet - CProcessPVS

This method is passed as its first parameter a master PVS data buffer that the compressed data will be copied into and as its 3rd parameter the location within this array where we should start writing the compressed data. As the second parameter the uncompressed PVS of a single leaf of passed. This is the data that is to be compressed.

The function first sets up a loop to iterate through every byte in the passed PVS set (VisArray).

```
ULONG CProcessPVS::CompressLeafSet ( UCHAR MasterPVS[],
                                     const UCHAR VisArray[],
                                     ULONG WritePos)
{
    ULONG   RepeatCount;
    UCHAR  *pDest = &MasterPVS[ WritePos ];
    UCHAR  *pDest_p;

    // Set dynamic pointer to start position
    pDest_p = pDest;

    // Loop through and compress the set
    for ( ULONG j = 0; j < m_PVSBytesPerSet; j++ )
    {
        // Store the current 8 leaves
        *pDest_p++ = VisArray[j];

        // Don't compress if all bits are not zero
        if ( VisArray[j] ) continue;
```

pDest_p is used to point at the current byte in the master PVS data array (passed as the first parameter) that we are currently copying information into. As you can see in the first line inside the loop in the above code, we copy the contents of the current byte being processed in the leaf set (VisArray) into the master PVS data array. In the bottom line in the above code we can see that after copying over this byte we test to see if it was zero or not. If it isn't zero then this byte has visible leaves in its bitset and can not be compressed. This means we can just skip to the next byte in the buffer having copied over this byte into the master array.

If we make it past the last line in the above code however, it means the current byte we have just copied over is a zero byte and therefore we must set up a loop to see how many zero bytes follow it. Once we have counted the run of zeros, we can simply insert this run length byte into the master PVS data array just after the zero byte we just copied over. Here is the remainder of the function.

```
        // Count the number of 0 bytes
        RepeatCount = 1;
        for ( j++; j < m_PVSBytesPerSet; j++ )
        {
            // Keep counting until byte != 0 or we reach our max repeat count
            if ( VisArray[j] || RepeatCount == 255) break; else RepeatCount++;

        } // Next Byte

        // Store our repeat count
        *pDest_p++ = (UCHAR)RepeatCount;
```

```
        // Step back one byte because the outer loop
        // will increment. We are already at the correct pos.
             j--;

    } // Next Byte

    // Return written size
    return pDest_p - pDest;
}
```

As you can see, we loop from our current byte position through the bytes that follow counting how many zero bytes we encounter in a continuous run. The RepeatCount is set to 1 to begin with as before this loop we only know of one zero that exists, the zero byte we just copied over. In the middle of this inner loop you can see that we break from the loop as soon as a non-zero byte is encountered or if the repeat count reaches 255 which is the highest run length value we can store in a byte. Otherwise, for each consecutive zero byte we find in the run we increment the repeat count.

Outside the inner loop, RepeatCount will contain the length of the run of zeros that were found starting from the original zero byte we copied into the master array. We then write this run length into the PVS data buffer (pDest_p).

Finally, we return from the function the total size of the data we managed to compress by subtracting from pDest_p, which contains the address of the last byte we have just written to the PVS data array, the value of pDest which contains the address of the first byte that was written by this function. That is, we return the number of bytes of compressed data that we have written to the PVS data array.

# Conclusion

Writing a portal and PVS compiler has perhaps been our most challenging task to date. However, our compiler tool has now evolved into something quite significant. You are certainly not expected to grasp every nuance of the code we have written in this chapter after a single read through the work book however, using this book as an aid to help you navigate the source code will have you up to speed on exactly how all this stuff works in no time at all.

It is nice that the ultimate work book in this course was dedicated to writing such a useful and reusable tool and that the data generated by such a tool will certainly be used in GP3 where it will be used in conjunction with a more complete graphics engine. When using complex pixel shaders with multiple passes (as we will be in GP3), reducing overdraw has once again become a premier design goal. The more wonderful and complex these programmable pixel shaders are, the more strain is placed on the 3D hardware for each pixel that it renders. Using a PVS we can make sure that our 3D hardware does not grind to a halt spending most of its time performing multiple passes on pixels that are occluded and can not even be seen. If it has not become apparent to you yet, let me just say that by developing this tool and the means to render its data, we have overcome a huge obstacle in game engine development. The ability to render only the small region of the scene that is currently visible from the camera. Our games can now spend that GPU processing power rendering high detail objects in the immediate vicinity instead of rendering an abundance of low detail objects that can never been seen but are rendered anyway.

The lab projects in this course have certainly become rather large at times but we have many re-usable modules in place now and have acquired many techniques that were necessary to learn before we could even start to construct a real graphics engine. We are certainly ready for Graphics Programming with DirectX Part III.