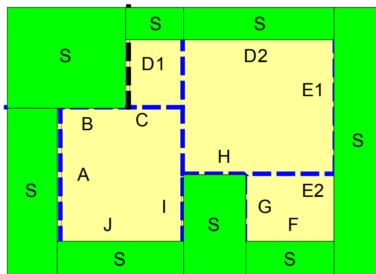


Binary Space Partitioning

parte I

Introducción a los árboles Bsp

Autor: Diego G. Ruiz

Versión: 1.0.1 (draft)

Tabla de Contenidos

Introducción 4
¿Qué es un BSP? 4
¿Para qué se utiliza un árbol BSP? 4
Creación de un árbol BSP 4
El primer paso 5
Como trabaja un BSP 5
Árbol BSP de nodos sólidos 8
Una rápida introducción a la detección de colisiones 11
El compilador BSP 11
Selección de un segmento divisor 11
Implementación del compilador 12
El método BuildBspTree 13
La función ClassifyPoint 13
La función ClassifyPoly 14
La función SplitPolygon 16
La función SelectBestSplitter 16
La función BuildBspTree 18
División de un polígono 21
Dependencias de métodos 24
Dibujando el árbol BSP 24

Binary Space Partitioning (BSP)

Introducción

El particionamiento binario del espacio (BSP) es una técnica muy empleada en juegos 3D desde los últimos 10 años. Si bien fue creada en el año 1969, aplicada a la generación de imágenes por computadora, díficilmente se hubiese pensado en aquel momento que algún día se utilizaría en juegos de computadora.

Sin embargo, John Carmack (Id Software) consideró que podría ser conveniente aplicarla en videojuegos para mejorar la eficiencia en el dibujado de entornos y decidió aplicar una versión 2d de la técnica en su nuevo juego: **Doom**. A partir de aquel momento nada sería lo mismo y otros desarrolladores, sorprendidos por el flamante juego de Id Software, decidieron aplicar el mismo algoritmo en sus juegos. Al día de hoy, la mayor parte de los motores utilizan de algún modo este algoritmo.

¿Qué es un BSP?

Un árbol BSP es una estructura de datos utilizada para ordenar y buscar polígonos en el espacio n-dimensional. El espacio es representado por el árbol, mientras que los nodos del mismo representan subespacios convexos.

Cada nodo almacena hiperplanos que dividen el espacio en dos, referencian a dos nodos y almacenan uno o más segmentos o polígonos (según las dimensiones del espacio utilizado).

Normalmente los árboles BSP representan espacios de dos dimensiones (juegos tipo Doom) y tres dimensiones (juego tipo Quake). En el primer caso los nodos almacenan segmentos y en el segundo los nodos almacenan polígonos (triángulos).

¿Para qué se utiliza un árbol BSP?

Inicialmente la idea fue utilizar el algoritmo para ordenar los polígonos del mundo en donde se recreaba el juego, debido a que en aquel momento no existía la aceleración de z-buffer por hardware y utilizar un z-buffer vía software era muy poco eficiente. Por lo tanto, había que buscar el modo de dibujar los polígonos en el orden adecuado para que todo se vea correctamente.

Hoy día, los detractores de este algoritmo proclaman que en esta área la técnica es obsoleta pues toda placa aceleradora, que se precie de tal, posee aceleración de z-buffer. Sin embargo, el poseer ordenados los polígonos que conforman en mundo es muy beneficioso para otros aspectos entre los cuales se destaca: la eliminación eficiente de polígonos no visibles, detección de colisiones, generación de protocolos de red de baja latencia, etc.

Pero no son todos beneficios, existen varios puntos negativos en la utilización de árboles BSP, principalmente porque los mismos son estáticos y es muy costoso recalcularlos en tiempo de ejecución, lo cual acarrea que la geometría de un mapa deba permanecer invariable en el desarrollo del juego.

Creación de un árbol BSP

Debido a que la generación de un árbol BSP es un proceso costoso, se realiza en tiempo de creación de mapas, usualmente mediante un trabajo por lotes a cargo de un programa

denominado compilador BSP. Un motor que acepte entornos descritos con este algoritmo sólo deberán recorrerlo e interpretarlo.

El primer paso

En la creación de un juego 3D tipo FPS, simplemente por definir algún género, tendremos:

Un mapa: que será un conjunto de polígonos que conformará el mundo en cual transcurre el juego.

Entidades: Personajes, objetos no estáticos, armas, powerups, etc.

Olvidemos por un momento las entidades, nuestro motor deberá ser capaz de dibujar el mapa del juego desde un punto de vista, en la cuál estará situada la cámara.

Si el mapa posee 10.000 polígonos (cantidad de polígonos promedio en un mapa de Quake 3) el motor deberá dibujar cada uno de esos polígnos de modo ordenado o utilizando z-buffer, pero recordemos que si existen objetos con algún grado de transparencia el z-buffer se torna inútil.

Si no deseamos utilizar z-buffer, entonces deberemos ordenar los polígonos a partir del punto de visualización en tiempo de ejecución antes de dibujarlos…naturalmente esto es impracticable. Agregando el cálculo de colisiones para todos los objetos (todos contra todos) torna la situación aún peor.

Si describimos nuestro mapa mediante un árbol BSP, tendremos almacenado en el árbol todos los polígonos del mapa de manera ordenada y listo para ser recorrido rápidamente por el motor en tiempo de ejecución.

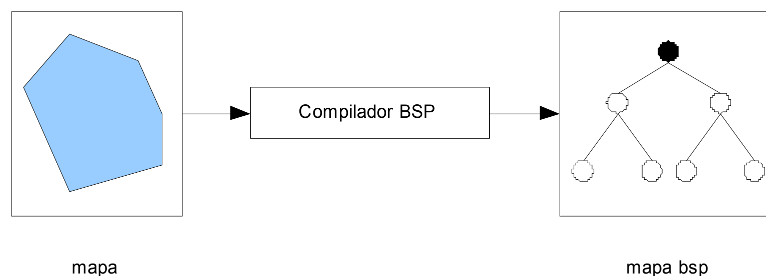


Figura 1. El compilador BSP tendrá el trabajo de ordenar todos los polígonos de un mapa

Pero no sólo utilizaremos un árbol BSP para dibujar los polígonos en el orden correcto, sino también para, y esto tal vez sea lo más importante, eliminar rápidamente polígonos no visibles. Esto es posible lograrlo analizando si un polígono se encuentra por detrás del volumen de visualización definido por el frustrum, si es así inmediatamente podemos inferir que todos los polígonos que se encuentren detrás el polígono testeado también lo estarán.

No hace falta mencionar la diferencia de velocidad que podría existir entre un motor que transforme e ilumine 10.000 polígonos para mostrar una cierta área de un mapa contra otro que sólo procese, digamos 1.000 para mostrar la misma imagen.

Como trabaja un BSP

Para entender como trabaja un BSP nos basaremos en distintos ejemplos utilizando un espacio 2d, simplemente porque es más fácil visualizarlo en una hoja. Luego, en la práctica desarrollaremos mapas en dos y en tres dimensiones donde los elementos a divir son triángulos, que llamaremos de modo genérico **polígonos**, ya que, a fin de cuentas, para el algoritmo es indistinto (siempre y cuando todos los puntos del polígono se encuentren en el mismo plano).

Supongamos que el mapa de nuestro juego es el que muestra la figura 2. Una habitación sencilla delimitada por 10 paredes.

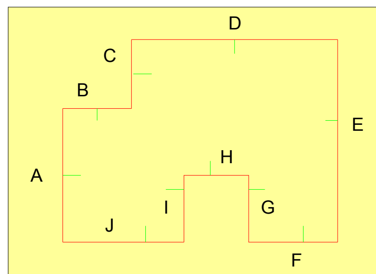


Figura 2. El mapa de nuestro juego

Cada pared, representada por un segmento color rojo posee adjunto otro pequeño segmento que indica su orientación. Esto es muy importante debido a que todos los segmentos (o polígonos) en un árbol BSP deben ser convexos, es decir, que todos deben estar mirándose mutuamente.

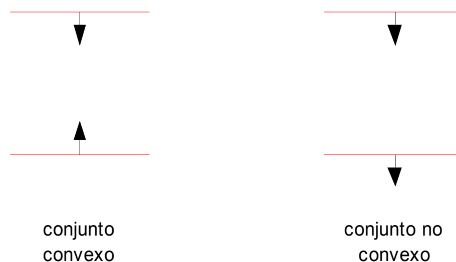


Figura 3. Conjuntos convexos y no convexos

En el apéndice A, se describe un sencillo algoritmo para determinar esto.

El primer paso en la construcción de un árbol BSP es la selección de una recta (plano, en las tres dimensiones) que divida nuestro mundo en dos. Naturlamente, existen infinitas rectas que podrían hacer esto, podríamos crear una de modo arbitrario, el asunto es que deseamos que nuestro árbol BSP se encuentre balanceado, o mejor dicho, lo más balanceado posible.

Definición: Un árbol se encuentra balanceado cuando diferencia de profundidad entre la rama izquierda y derecha no es mayor a uno.

La razón por la cual deseamos tener un árbol balanceado es para realizar la menor cantidad de divisiones posibles y que de este modo al motor de nuestro juego le lleve menos tiempo recorrerlo.

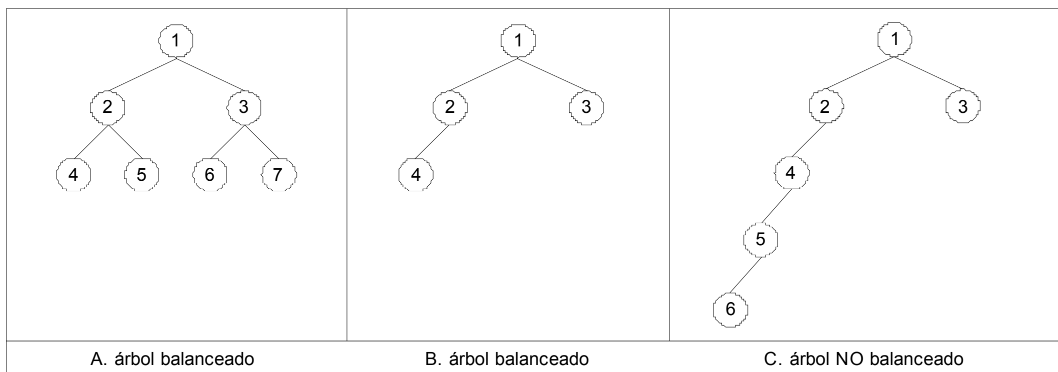


Figura 4. Árboles balanceados y no balanceados.

Usualmente no se crea un recta de división arbitraria sino que se selecciona un segmento que arbitre de divisor de nuestro mundo (así como en tres dimensiones se selecciona un polígono). Que segmento seleccionar es una de las tareas importantes que debe efectuar nuestro compilador BSP.

La idea es que el segmento divida al mundo de modo que a un lado y el otro resulten una cantidad similar de segmentos.

En nuestro ejemplo, seleccionaremos el segmento I. Por el momento, nuestra selección fue manual luego veremos un algoritmo que realice esto de modo automático.

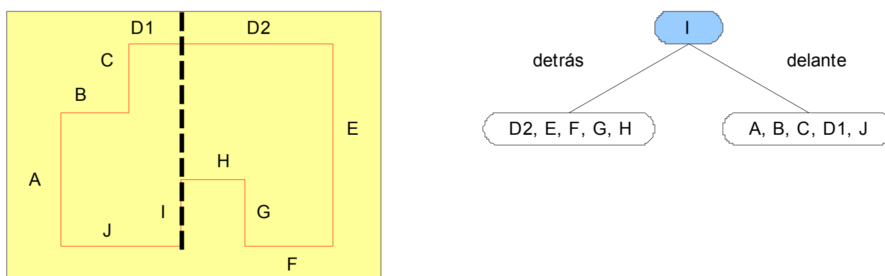


Figura 5

El segmento I, divide a nuestro mundo en dos. Pero notemos que la recta trazada divide al segmento D en dos. Este caso es muy común y será otra de las tareas que deberá efectuar el compilador BSP: dividir un polígono en partes de modo de establecer subconjuntos divisibles.

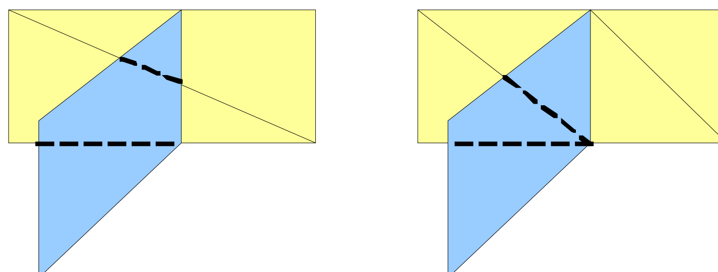


Figura 6. División de polígonos.

Por un lado tendremos los segmentos A, B, C, D1 y J y por el otro D2, E, F, G y H. Ahora, aplicaremos de modo recursivo el mismo algoritmo que antes sólo que con cada uno de los subconjunto de segmentos que quedaron.

Siempre los segmentos que queden por delante del segmento divisor se colocarán a derecha mientras que los que se encuentren por detrás se colocarán a izquierda.

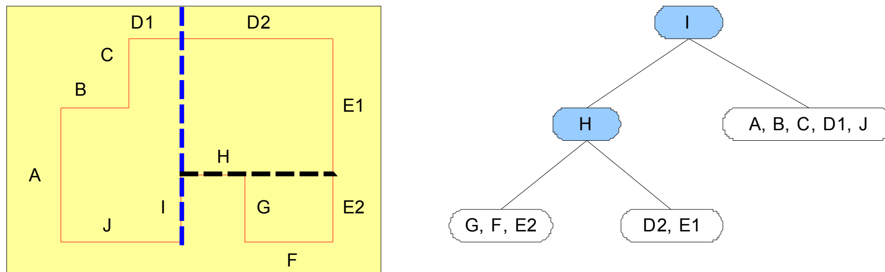


Figura 7

Tomando primero la rama izquierda del árbol, seleccionamos al segmento H como buen divisor del subconjunto se segmentos restantes.

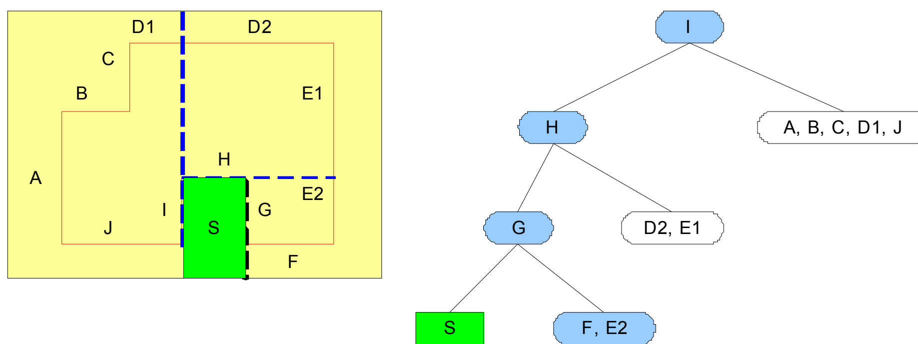


Figura 8

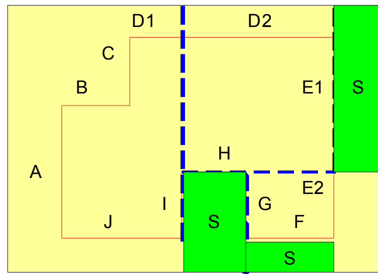
Siguiendo nuestro camino la rama izquierda del nodo H nos define un subconjunto de tres segmentos (también divide al segmento E en dos). No tenemos muchas opciones, elegimos arbitrariamente el segmento G que nos divide el mundo en dos: por un lado queda el segmento F y E2 y ¿por el otro?. Nuestro árbol BSP no dejará a polígonos como hojas de ninguna rama, debido a que deseamos contruir un árbol BSP de nodos **sólidos**, todo recorrido del árbol deberá terminar en un nodo especial que indicará si la resultante es un sólido o un espacio vacío. Por lo tanto, a izquierda de nuestro nodo G quedará determinado un sólido, ya que es un subespacio determinado por los lados internos de unas paredes. Más adelante volveremos a abordar este tema nuevamente.

Árbol BSP de nodos sólidos

No es estrictamente necesario colocar nodos "sólidos" y "vacíos" en nuestro árbol BSP. Veremos más adelante que esto nos facilitará los cálculos de visibilidad entre dos puntos del mapa pero bien podríamos no fijar este tipo de información en el árbol y de todos modos estaríamos creando un árbol BSP (clásico) que podría ser utilizado para ordenar polígonos y para que nos ayude en la eliminación de polígonos no visibles.

Área Sólida

En pocas palabras un "vacío" es una zona del mapa por donde el jugador puede caminar y "sólido" es una zona por donde nadie podría desplazarse como el interior de paredes.



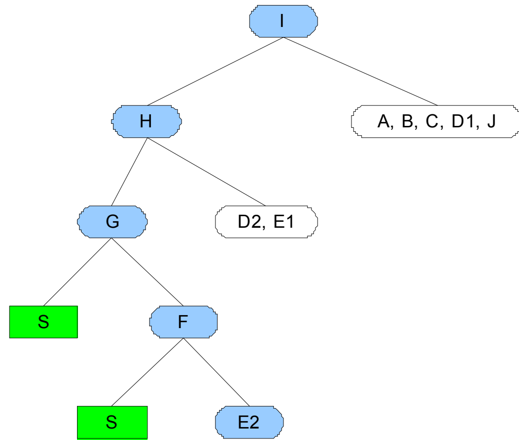
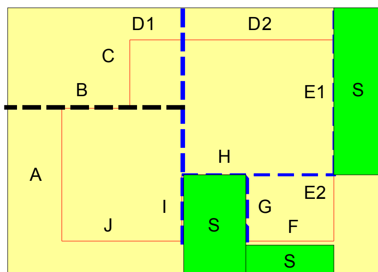


Figura 9

Terminando de clasificar los polígonos de una de las ramas del árbol continuamos con el subconjunto {D2, E1}. Procediendo de modo análogo con G, seleccionamos a E1 como segmento divisor obteniendo por un lado al segmento D2 y por el otro un sólido.



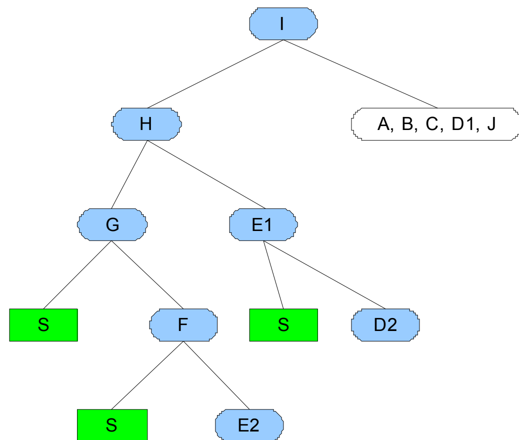


Figura 10

Ahora proseguimos con la rama derecha.

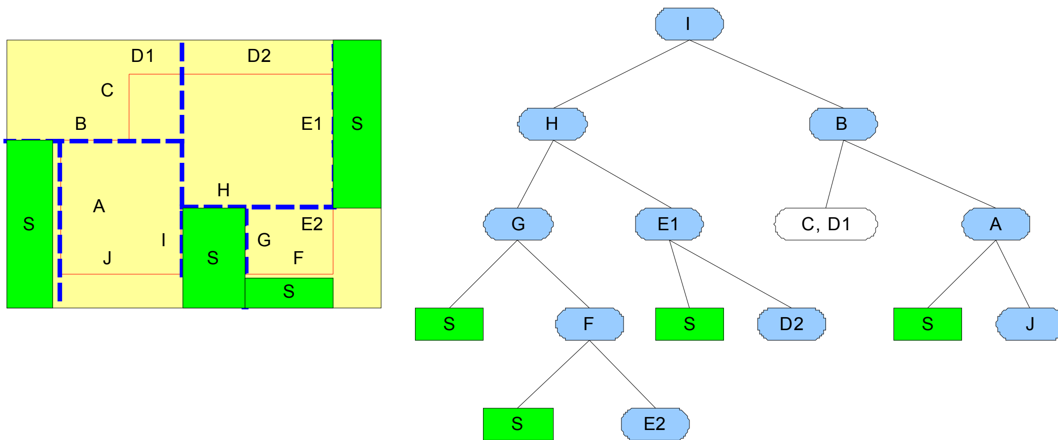


Figura 11

Con la misma política utilizada anteriomente continuamos dividiendo nuestro mundo en dos, especificando sólidos cuando nos parezca conveniente.

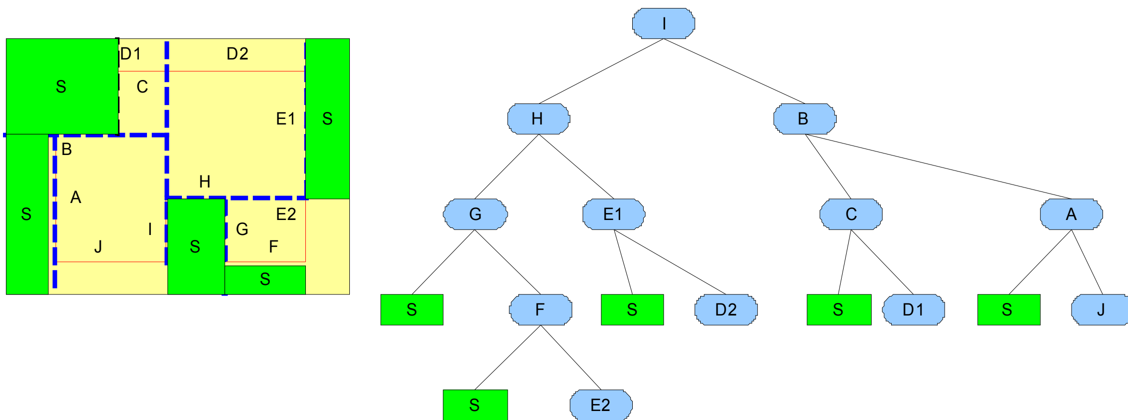
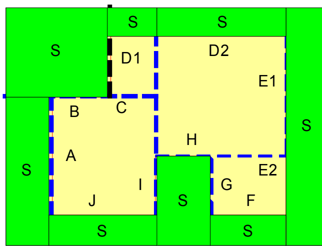


Figura 12

Finalmente, llegó el momento de especificar las hojas del último nivel de nuestro árbol. Habíamos dicho que no dejaríamos polígonos sueltos como hojas. Lo que haremos será especificar que divide el último polígono, según la orientación del mismo colocará sólido en un lado y vacío en el otro.



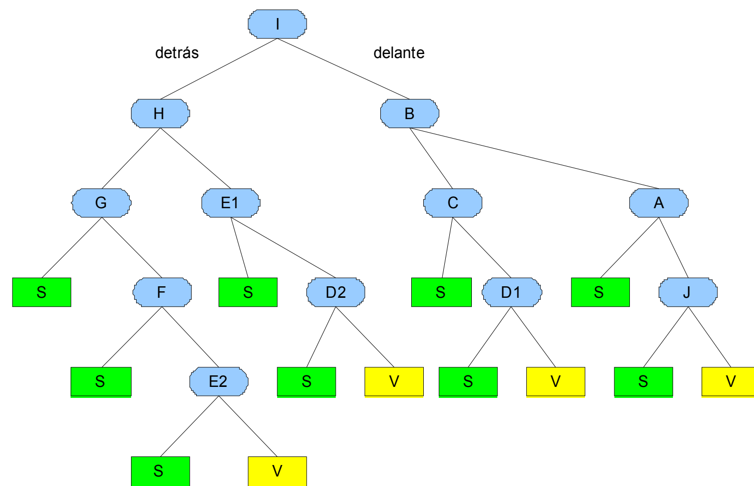


Figura 13

Una rápida introducción a la detección de colisiones

Habíamos comentado que el árbol BSP nos ayudaría mucho en la detección de colisiones. Supongamos que tenemos un nivel que cuenta con 5.000 polígonos, nuestro árbol BSP de nodos sólidos poseerá aproximadamente 10.000 nodos. Ahora imaginemos que deseamos verificar si el jugador colisiona contra una pared ¿cómo procederíamos en estos casos? Usualmente deberíamos verificar la posición de nuestro jugador contra todo polígono que exista en el mapa, pero esto es muy costoso y depende linealmente del tamaño y complejidad del mapa. ¿Cómo sería la solución utilizando BSPs? Primero deberíamos verificar si nuestro jugador se encuentra delante o detrás del polígono raíz, fuera cual fuese el resultado estaremos eliminando la mitad de los polígonos del mapa con esa sencilla verificación, luego continuaremos verficando la posición del jugador testeando contra cada uno de los nodos, finalmente quedaremos posicionados sobre un nodo sólido o vacío, habiendo obtenido el resultado que estábamos buscando. Con aproximadamente 15 verificaciones habremos recorrido un árbol de 10.000 nodos.

El compilador BSP

Visto un ejemplo sencillo de modo muy general, analicemos cuales son las dos tareas que remarcamos debe hacer el compilador BSP:

- Seleccionar un segmento (un polígono en 3D) que divida en el mapa en dos de modo que la cantidad de segmentos a los lados sea similar (idealmente igual).
- Dividir segmentos (polígonos en 3D) cuando un segmento divisor corte algún segmento del mapa. Los nuevos segmentos tendrán las mismas propiedades que el segmento que les dio origen.

Selección de un segmento divisor

Como habíamos comentado antes, en teoría podríamos elegir cualquier segmento como dividor dentro del subconjunto de segmento que posee el mapa (o parte de él) en un determinado momento, sin embargo es conveniente seleccionar el segmento que divida "mejor" el mapa, es decir, que divida al mapa de modo de dejar a ambos lados una cantidad de segmentos similar.

Es importante establecer que el segmento utilizado como divisor no formará parte de ninguna de las dos listas que se generen. Con esto queremos decir que cada vez que hagamos una división estaremos eliminado un polígono de la lista entrante y seguiremos realizando divisones hasta que absolutamente todos los segmentos de la lista original hayan sido utilizados como divisores (si un segmento debe partirse en dos, lo dicho se aplica a cada una de sus partes). Llegado a este punto habremos creado el árbol BSP, sólo bastará agregar los vacíos y sólidos.

Implementación del compilador

El compilador deberá recibir como entrada una lista de polígonos para esto podremos crear una estructura que nos permita especificar uno con todas las propiedades que necesitaremos tener en él:

```
struct Polygon
{
	TexNormVertex m_verts[4];
	short m_idxs[6];
	int m_iNumberOfVerts;
	int m_iNumberOfIdxs;
	Polygon * m_pNext;
};
```

Luego, el compilador hará su trabajando y comenzará a crear nodos del árbol BSP que poseerán en siguiente formato:

```
struct BspNode
{
	Polygon * m_pSplitter;
	BspNode * m_pFront;
	BspNode * m_pBack;
	bool m_bIsLeaf;
	bool m_bIsSolid;
};
```

Finalmente, la estructura que contenga los nodos, el árbol en sí mismo, será simplemente un puntero al nodo raíz del mismo.

El método BuildBspTree

Este método recibirá como parámetro un nodo del árbol y una lista de polígonos. En primer lugar el nodo del árbol será el nodo raíz y el conjunto de polígonos será el conjunto completo de polígonos de nuestro mapa.

Pero antes de codificar este método, que será el principal del compilador BSP, necesitaremos algunas funciones de ayuda como una que dado un punto y un plano nos indique si el mismo está delante o detrás del mismo.

La función ClassifyPoint

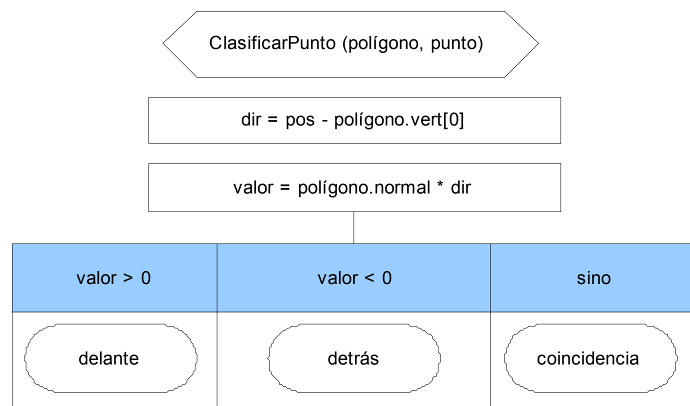
La función recibirá un punto (Vector3) y un polígono (Polygon) y podrá retornar un valor entre las siguientes constantes:

CP_BACK: El punto se encuentra detrás del polígono.

CP_FRONT: El punto se encuentra delante del polígono.

CP_ONPLANE: El punto se encuentra sobre el mismo plano que el polígono.

Diagrama de flujo de la función:



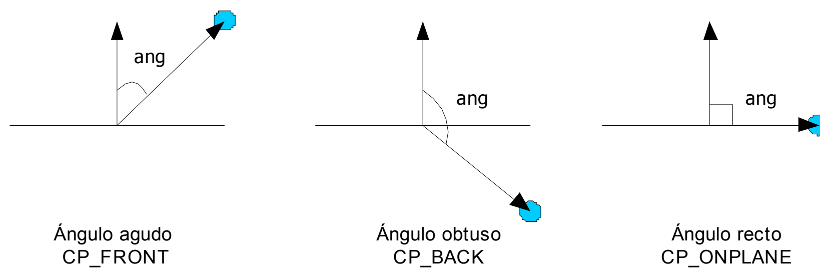
La función se basa en el principio que el producto escalar (punto) de dos vectores retorna un número:

> 0, si el ángulo formado entre dichos vectores es agudo.

< 0, si el ángulo formado entre dichos vectores es obtuso.

= 0, si el ángulo formado entre dichos vectores es de 90 grados.

Si el ángulo formado entre el vector dirección y el vector normal es agudo, el punto se encuentre frente al polígono, mientras que si el ángulo es obtuso sucede el caso contrario y finalmente si el ángulo es de 90 grados el punto se encuentra sobre el mismo plano que el polígono.



El código:

```
int ClassifyPoint(Vector3 * pos, Polygon * poly)
{
	// Creo un vector dirección (utilizo cualquier punto del polígono
	// ya que todos están en el mismo plano).
	Vector3 dir;
	dir.x = pos->x - poly->m_verts[0].x;
	dir.y = pos->y - poly->m_verts[0].y;
	dir.z = pos->z - poly->m_verts[0].z;

	// Multiplico el vector dirección con la normal del polígono
	float result = D3DXVec3Dot(&dir, (Vector3 *) &poly->m_verts[0].nx);

	// En función del resultado de la operación anterior
	// retorno la clasificación del punto.
	if (result > 0.001f)
		return CP_FRONT;
	else if (result < -0.001f)
		return CP_BACK;
	else
		return CP_ONPLANE;
}
```

Notemos que las comparaciones no son contra 0 sino contra números cercanos a él, debido a pequeños errores que se pueden introducir en el uso de números tipo float.

La función ClassifyPoly

La función **ClassifyPoly** es muy similar a **ClassifyPoint**, de hecho podría hacer uso de la misma (no lo hacemos simplemente para poder optimizar un poco la función, recordemos que esta será invocada en muchas ocasiones), la idea aquí es probar cada uno de los vértices de un polígono para conocer si se encuentra delante o detrás de otro.

Esta función será utilizada por el compilador para saber si un polígono se encuentra en parte delante y en parte detrás de otro pasado como parámetro, siendo este el caso el polígono testeado deberá ser dividido en dos. O en otro caso, si se encuentra totalmente delante o totalmente detrás saber en el que rama del árbol agregarlo.

Existirá el caso especial donde todos los vértices de un polígono se encuentren en el mismo plano que el polígono testigo, veremos luego que hacer aquí.

La función recibirá dos parámetros: el polígono a probar y el polígono usado como base o testigo para realizar la prueba. Los valores posibles de retorno son:

CP_FRONT: El polígono se encuentra totalmente delante del testigo.

CP_BACK: El polígono se encuentra totalmente detrás del testigo.

CP_ONPLANE: El polígono se encuentra en el mismo plano del polígono testigo.

CP_SPANNING: El polígono se encuentra en parte delante y en parte detrás del testigo.

El código:

```
int ClassifyPoly(Polygon * pol, Polygon * polTestigo)
{
	int iFront = 0; int iBehind = 0; int iOnPlane = 0;
	Vector3 dir;
	float result;

	for (int i=0; i<pol->m_iNumberOfVerts; i++)
	{

		// Obtengo el vector dirección
		dir.x = pol->m_verts[i].x - polTestigo->m_verts[0].x;
		dir.y = pol->m_verts[i].y - polTestigo->m_verts[0].y;
		dir.z = pol->m_verts[i].z - polTestigo->m_verts[0].z;

		// Realizo la multiplicación escalar del vector dirección con la normal
		// del polígono testigo.
		result = D3DXVec3Dot(&dir, (Vector3 *) &polTestigo->m_verts[0].nx);

		if (result > 0.001f)
			iFront++;
		else if (result < -0.001f)
			iBehind++;
		else
			iOnPlane++;

		if (iFront > 0 && iBehind > 0)
			return CP_SPANNING;
		else if (iOnPlane > 1)
			return CP_ONPLANE;
	}
```

```
        if (iFront > 0)
                return CP_FRONT;
        else
                return CP_BACK;
}
```

La función SplitPolygon

La función **SplitPolygon** recibirá dos polígono como entrada: el polígono a dividir y el polígono que representa el plano de división y dos parámetros de salida que serán dos polígonos que forman una partición del polígono original.

```
void SplitPolygon(Polygon * poly, Polygon * plane, Polygon * polyFrontSplit, Polygon * polyBackSplit)
{
        // ...
}
```

La función SelectBestSplitter

La función **SelectBestSplitter** retornará dado un conjunto de polígonos cuál de ellos es el mejor para realizar la partición del mapa. Como habíamos comentado antes, es importante seleccionar un buen polígono divisor en cada partición para obtener un bueno árbol BSP (lo más balanceado posible) sin embargo es muy díficil obtener "el mejor" árbol BSP ya que un buen splitter de primer nivel podría crear subconjuntos donde los splitters necesarios para ellos no sean tan bueno como en otros casos, por lo tanto la selección del mejor splitter para un nivel debería tener en cuenta los splitter posibles de niveles inferiores, pero esto no es factible de implementar ya que nos llevaría muchísimo tiempo, entonces nos conformaremos con que nuestro algoritmo elija un buen splitter solamente para el nivel analizado.

¿Qué polígono es considerado un buen splitter?

Un buen splitter deberá tener en cuenta dos cosas:

- Que los subconjuntos de polígonos sean numéricamente similares.
- Que la cantidad de particiones de polígonos generada sea baja.

El balance del árbol es muy importante, pero no lo es menos la cantidad de divisiones de polígonos que podría generar nuestro splitter. Supongamos el peor caso: un splitter divide en dos a absolutamente todos los polígonos del nivel; esto causaría que cada rama de nuestro árbol volviese a poseer la cantidad de polígonos original (menos el polígono utilizado como splitter, claro). La consecuencia de esto hecho generará árbol de mayor profundidad, lo cual a su vez generará consultas al mismo mas extensas porque se deberán recorrer mas nodos.

Desgraciadamente nuestros dos objetivos pueden no darse simultáneamente, podríamos tener por un lado un splitter que divida al mapa en partes iguales pero divida muchos polígonos en

dos mientras que otro divide al mapa en partes bastantes desiguales pero no generar polígonos extra ¿cuál elegir entonces?

Lo que haremos será crear una ecuación que retorne un puntaje en función de los resultados del splitter y luego nos quedaremos con el splitter de menor puntaje. Esta ecuación tendrá en cuenta el balance (subconjuntos similares en cantidad) y la profundidad (pocas particiones de polígonos).

score = | cantidad_polígonos_delante - cantidad_polígonos_detrás | + (cantidad_splits * 8)

Como puede observarse en la fórmula le estamos dando mayor importancia a la cantidad de splits. Una vez armado nuestro algoritmo creador del árbol podremos jugar con la ecuación para ver los distintos árboles que genera y poder hacer un ajuste más fino (aunque naturalmente mejorar el árbol BSP a partir de un conjunto de polígonos podría emperar el árbol BSP que se genere a partir de otros).

```
Polygon * SelectBestSplitter(Polygon * pPolyList)
{
	int iFrontFaces, iBackFaces, iSplits;
	long lScore, lBestScore = -1;

	// Me posiciono en el primer polígono
	BspPolygon * pSplitter = pPolyList;
	BspPolygon * pCurrent;
	BspPolygon * pSelectedPoly = NULL;
	int result;

	while (pSplitter)
	{
		iFrontFaces = iBackFaces = iSplits = 0;
		pCurrent = pPolyList;
		while (pCurrent)
		{
			if (pCurrent != pSplitter)
			{
				result = ClassifyPoly(pCurrent, pSplitter);

				if (result == CP_FRONT)
					iFrontFaces++;
				else if (result == CP_BACK)
					iBackFaces++;
				else if (result == CP_SPANNING)
					iSplits++;
			}
			pCurrent = pCurrent -> m_pNext;
```

```
		}

		lScore = abs(iFrontFaces - iBackFaces) + iSplits * 8;

		if (lScore < lBestScore || lBestScore == -1)
		{
			lBestScore = lScore;
			pSelectedPoly = pSplitter;
		}
		pSplitter = pSplitter->m_pNext;
	}
	return pSelectedPoly;
}
```

La función BuildBspTree

Finalmente hemos llegado a tratar la función principal que construirá el árbol de manera recursiva.

```
void BuildBspTree(BspNode * pCurrentNode, Polygon * pPolyList)
{
	int result;
	BspPolygon * pFrontList = NULL;
	BspPolygon * pBackList = NULL;
	BspPolygon * pNextPoly = NULL;

	// Obtengo el mejor splitter
	pCurrentNode->m_pSplitter = SelectBestSplitter(pPolyList);

	// Selecciono el primer polígono de la lista entrante
	BspPolygon * pPolyTest = pPolyList;

	// Mientras que queden polígonos en la lista entrante
	while (pPolyTest)
	{
		// Fijo cual es el próximo polígono a probar
		pNextPoly = pPolyTest->m_pNext;

		if (pPolyTest != pCurrentNode->m_pSplitter)
		{
			// Clasifico el polígono
			result = ClassifyPoly(pPolyTest, pCurrentNode->m_pSplitter);
			if (result == CP_FRONT)
```

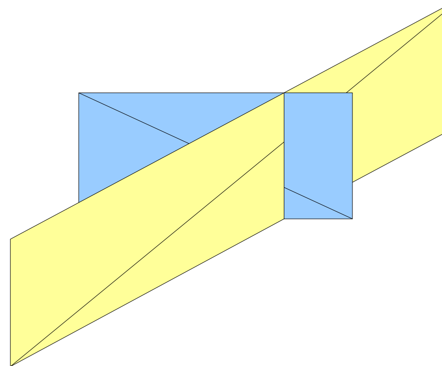
```
			{
				// Agrego a la cabeza el polígono clasificado
				pPolyTest->m_pNext = pFrontList;
				pFrontList = pPolyTest;
			}
			else if (result == CP_BACK)
			{
				// Agrego a la cabeza el polígono clasificado
				pPolyTest->m_pNext  = pBackList;
				pBackList = pPolyTest;
			}
			else if (result == CP_SPANNING)
			{
				// Debo dividir el polígono en dos

				// Creo dos BspPolygon y les fijo 0 en todas
				// sus propiedades
				BspPolygon * pFrontSplit = new BspPolygon;
				BspPolygon * pBackSplit = new BspPolygon;
				memset((void *) pFrontSplit, 0, sizeof(BspPolygon));
				memset((void *) pBackSplit, 0, sizeof(BspPolygon));

				// Parto el polígono
				SplitPolygon(pPolyTest, pCurrentNode->m_pSplitter, pFrontSplit, pBackSplit);

				// Eliminano el polígono original
				delete pPolyTest;

				// Agrego un polígono a la lista delantera
				// y el otro a la trasera
				pFrontSplit-> m_pNext  = pFrontList;
				pFrontList = pFrontSplit;
				pBackSplit-> m_pNext  = pBackList;
				pBackList = pBackSplit;
			}
		}

		pPolyTest = pNextPoly;
	} // end while

	if (!pFrontList)
	{
		// Creo un nuevo nodo y lo fijo en cero
		BspNode * pLeafNode = new BspNode;
```

```
		memset((void *) pLeafNode, 0, sizeof(BspNode));

		// Doy valor a algunas de sus propiedades
		pLeafNode->m_bIsLeaf = true;
		pLeafNode->m_bIsSolid = false;
		pCurrentNode->m_pFront = pLeafNode;
	}
	else
	{
		// Creo un nuevo nodo y lo fijo en cero
		BspNode * pNewNode = new BspNode;
		memset((void *) pNewNode, 0, sizeof(BspNode));

		// Doy valor a algunas de sus propiedades
		pNewNode->m_bIsLeaf = false;
		pCurrentNode->m_pFront = pNewNode;

		// Invoco nuevamente el método
		BuildBspTree(pNewNode, pFrontList);

	}

	if (!pBackList)
	{
		// Creo un nuevo nodo y lo fijo en cero
		BspNode * pLeafNode = new BspNode;
		memset((void *) pLeafNode, 0, sizeof(BspNode));

		// Doy valor a algunas de sus propiedades
		pLeafNode->m_bIsLeaf = true;
		pLeafNode->m_bIsSolid = true;
		pCurrentNode->m_pBack = pLeafNode;
	}
	else
	{
		// Creo un nuevo nodo y lo fijo en cero
		BspNode * pNewNode = new BspNode;
		memset((void *) pNewNode, 0, sizeof(BspNode));

		// Doy valor a algunas de sus propiedades
		pNewNode->m_bIsLeaf = false;
		pCurrentNode-> m_pBack  = pNewNode;

		// Invoco nuevamente el método
```

```
                    BuildBspTree(pNewNode, pBackList);
          }
}
```

División de un polígono

Veamos ahora la función **SplitPolygon** de manera detallada. Por lo que sabemos, poseemos un polígono que está cortando a otro (aunque en realidad el plano al cual pertence un polígono estaría cortando a otro).



Para crear la función **SplitPolygon** nos basaremos en otra función que denominaremos **GetIntersect** y que nos dará el punto de intersección entre un rayo y un polígono, veamos su prototipo:

```
bool GetIntersect(
Vector3 * pv3LineStart,
Vector3 * pv3LineEnd,
Vector3 * pv3Vertex,
Vector3 * pv3Normal,
Vector3 * pv3Intersection,
float * fPercentage);
```

Los primeros dos parámetros describen el rayo (punto de comienzo y punto de fin), el tercer y cuarto parámetro especifican el plano contra el cual calcular la intersección (punto + normal). Los dos últimos parámetros son de salida: el primero retorna el punto de intersección y el segundo es el porcentaje de que tan lejos está la intersección de pv3LineStart (0%: el punto coincide con pv3LineStart, 100%: el punto coincide con pv3LineEnd), esta última propiedad nos será útil para recalcular las coordenadas de textura.

Código de la función:

```
bool GetIntersect(Vector3 * pv3LineStart, Vector3 * pv3LineEnd, Vector3 * pv3Vertex, Vector3 * pv3Normal, Vector3 *
pv3Intersection, float & fPercentage)
{
          Vector3 v3Dir = *pv3LineEnd - *pv3LineStart;
```

```
	float fLineLength = D3DXVec3Dot(&v3Dir, pv3Normal);

	// Si la dirección del rayo es perpendicular al plano, significa que ambos corren
	// paralelamente y jamás se cortarán.
	if (fLineLength < 0.001f && fLineLength > -0.001f)
		return false;

	Vector3 v3L1 = *pv3Vertex - *pv3LineStart;

	fPercentage = D3DXVec3Dot(&v3L1, pv3Normal) / fLineLength;

	if (fPercentage < 0.0f || fPercentage > 1.0f)
		return false;

	// Calculo el punto de intersección
	*pv3Intersection = *pv3LineStart + v3Dir * fPercentage;

	return true;
}
```

Ahora crearemos la función **SplitPolygon**, de la cual ya habíamos adelantado el prototipo. La idea aquí será testear cada borde del polígono tomado como un rayo por medio de la función **GetIntersect** para obtener el punto de intersección.

Recordemos que estamos testeando un polígono contra un splitter (que es en realidad otro polígono) por lo tanto llamaremos la función pasando dos vértices del borde del polígono y cierta información del splitter que conformará el plano (la normal del splitter será la normal del plano y el resto de la información es un punto del plano para el cual bastará algún vértice del splitter).

El prototipo de la función es:

```
void SplitPolygon(Polygon * poly, Polygon * plane, Polygon * polyFrontSplit, Polygon * polyBackSplit);
```

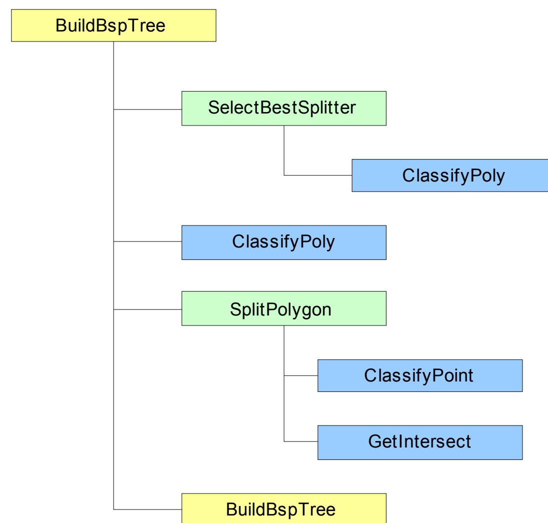
Recordemos que el primer polígono es el polígono a dividir y el segundo es el splitter; los dos últimos parámetros son de salida y representan el conjunto de polígonos en los cuales queda dividido el polígono original.

```
void SplitPolygon(Polygon * poly, Polygon * plane, Polygon * polyFrontSplit, Polygon * polyBackSplit)
{
	TexNormVertex v3FrontList[4]; TexNormVertex v3BackList[4];
	int iFrontCounter = 0;
	int iBackCounter = 0;

	// verifico si el primer punto del polígono a partir está delante
```

```
// o detrás del polígono usado como plano
int result = ClassifyPoint((Vector3 *) &poly->m_verts[0], plane);

if (result == CP_FRONT)
        // Agrego el vértice a la lista delantera
        v3FrontList[iFrontCounter++] = poly->m_verts[0];
else if (result == CP_BACK)
        // Agrego el vértice a la lista trasera
        v3BackList[iBackCounter++] = poly->m_verts[0];
else if (result == CP_ONPLANE)
{
        // Si está en el mismo plano agrego el vértice a las dos listas
        v3FrontList[iFrontCounter++] = poly->m_verts[0];
        v3BackList[iBackCounter++] = poly->m_verts[0];
}

Vector3 PointA, PointB;
int iNextVertex;
Vector3 v3IntersectPoint;
float fPerc;
for (int i=0; i<=poly->m_iNumberOfVerts; i++)
{
        PointA.x = poly->m_verts[i].x;
        PointA.y = poly->m_verts[i].y;
        PointA.z= poly->m_verts[i].z;

        if (i == poly->m_iNumberOfVerts)
                iNextVertex = 0;
        else
                iNextVertex = i + 1;

        PointB.x = poly->m_verts[iNextVertex].x;
        PointB.y = poly->m_verts[iNextVertex].y;
        PointB.z = poly->m_verts[iNextVertex].z;

        result = ClassifyPoint(&PointB, plane);
        if (result == CP_ONPLANE)
        {
                v3FrontList[iFrontCounter++] = poly->m_verts[iNextVertex];
                v3BackList[iBackCounter++] = poly->m_verts[iNextVertex];
        }
        else
        {
                if (GetIntersect(&PointA, &PointB, &plane->m_v3VertList[0], &v3IntersectPoint, fPerc)(
```

```
                {
                        // ...
                }
        }
    }

}
```

Dependencias de métodos



Dibujando el árbol BSP

La renderización de un árbol BSP es muy sencilla, y esto es bueno ya que es lo que deberá hacer el motor constantemente. Es posible utilizar un árbol BSP para dibujar los polígonos de atrás hacia delante (lo usual) o también de adelante hacia atrás (en caso de estar utilizando un Z-Buffer y no requirir el dibujado de objeto traslúcidos).

La función será llamada una vez por frame y dibujará el árbol completo.

```
void DrawBspTree(BspNode * pNode, Vector3 * pos)
{
        // Si el nodo es una hoja, retorno
        if (pNode->m_bIsLeaf)
                return;

        // Verifico si el punto está por delante o detrás del polígono
        int result = ClassifyPoint(pos, pNode->m_pSplitter);
```

```
// ¿Está la posición detrás?
// [Cambiar CP_BACK por CP_FRONT si se desea
//  revertir el orden de dibujado]
if (result == CP_BACK)
{
        // Si la lista "detrás" existe, la dibujo
        if (pNode->m_pBack)
                DrawBspTree(pNode->m_pBack, pos);

        // [Traer "snip01" si se desea revertir
        //  el orden de dibujado]

        // Si la lista "delante" existe, la dibujo
        if (pNode->m_pFront)
                DrawBspTree(pNode->m_pFront, pos);

        return;
}
else
{

        // ¿Está la posición detrás o en plano?
        if (pNode->m_pFront)
                DrawBspTree(pNode->m_pFront, pos);

        // ------------------------------------------------------------
        // "snip01"
        // Dibujo el polígono del nodo actual
        // ------------------------------------------------------------
        if (pNode->m_pSplitter->m_iMaterialIdx != -1)
        {
                g_renderer.SetMaterial(
                        g_matCol.GetMaterialNum(pNode->m_pSplitter->m_iMaterialIdx));
                g_renderer.BindTexture(
                        g_matCol.GetTextureIdFrom(pNode->m_pSplitter->m_iMaterialIdx));
        }
        else
                g_renderer.BindTexture(-1);

        g_renderer.ProcsIdx(pNode->m_pSplitter->m_verts, pNode->m_pSplitter->m_iNumberOfVerts,
                        pNode->m_pSplitter->m_idxs, pNode->m_pSplitter->m_iNumberOfIdxs,
                        AGL_TRIANGLE_LIST);
        // ------------------------------------------------------------
```

```
		if (pNode->m_pBack)
			DrawBspTree(pNode->m_pBack, pos);
	}

	return;

}
```

Como se puede apreciar en el código de la función, dibujamos siempre los polígonos que se encuentra detrás primero, luego los polígonos relacionados al nodo en el cual estamos y finalmente los polígonos que se encuentran delante.

Bibliografía

Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering - Samuel Ranta-Eskola

Core Techniques and Algorithms in Game Programming - Daniel Sánchez, Crespo Dalmau