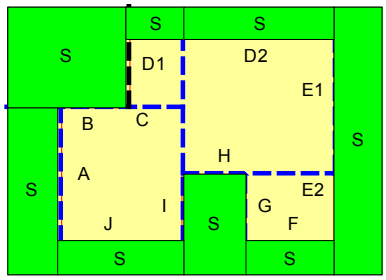


Binary Space Partitioning

parte III

Construcción del árbol y partición de polígonos

Autor: Diego G. Ruiz

Versión: 1.0 (draft)

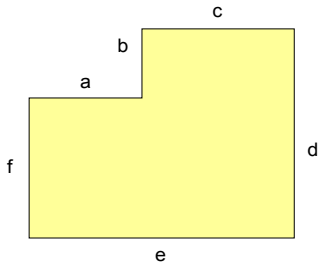
Tabla de Contenidos

Binary Space Partitioning (BSP)

parte III - Construcción del árbol y partición de polígonos

Introducción

Veamos como funciona el algoritmo de generación de un árbol BSP de nodo sólido, así como la detección de colisiones para el siguiente mapa:



La clase BspTree

Según el documento **bsp01.doc** tenemos implementado un conjunto de funciones que introducimos en una clase llamada BspTree, entre ellas podemos encontrar:

int BspTree::ClassifyPoint(Vector3 * pos, BspPolygon * poly);

Recibe un punto en R3 y un polígono que representa un plano y retorna una de las siguientes constantes en función de la ubicación de dicho punto respecto al plano.

CP_FRONT : El punto se encuentra al frente del plano.

CP_BACK : El punto se encuentra detrás del plano.

CP_ONPLANE : El punto se encuentra sobre el plano.

int BspTree::ClassifyPoly(BspPolygon * pol, BspPolygon * polTestigo);

Recibe dos polígonos, el primero se testea contra el segundo para verificar si el mismo se encuentra delante o detrás. Para esto se verfica cada uno de los vértices del primer polígono contra el plano representado por el segundo. Los valores de retorno posible son:

CP_FRONT : El polígono se encuentra al frente del plano.

CP_BACK : El polígono se encuentra detrás del plano.

CP_ONPLANE : El polígono se encuentra sobre el plano.

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

BspPolygon * BspTree::SelectBestSplitter(BspPolygon * pPolyList);

Recibe una lista enlazada de polígonos y retorna el puntero al polígono que se considera mejor divisor del conjunto. Para esto se basa en la siguiente ecuación:

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

Teniendo en cuenta que mejor divisor será quien obtenga **menor** puntaje, la ecuación penaliza la cantidad de splits multiplicándola por un número arbitrario (en este caso 8), es posible ajustar dicho valor y reejecutar la construcción de un determinado árbol para analizar los distintos resultados. Sin embargo, mejores resultados para un mapa particular podría significar peores resultados para otro mapa.

void BspTree::SplitPolygon(BspPolygon * poly, BspPolygon * plane, BspPolygon * polyFrontSplit, BspPolygon * polyBackSplit);

Recibe un polígono y un plano expresado otro polígono, entrega en sus últimos dos parámetros una partición del primer polígono originada por el plano recibido.

void BspTree::DrawBspTree(BspNode * pNode, Vector3 * pos);

Recorre el árbol Bsp y dibuja cada uno de sus nodos en un orden determinado.

bool BspTree::IsSolid(BspNode * pNode, Vector3 * pos);

Recorre el árbol y retorna si la posición recibida como parámetro se encuentra en un nodo sólido o vacío.

void BspTree::BuildBspTree(BspNode * pCurrentNode, BspPolygon * pPolyList);

Construye el árbol Bsp haciendo uso de los otros métodos de la clase.

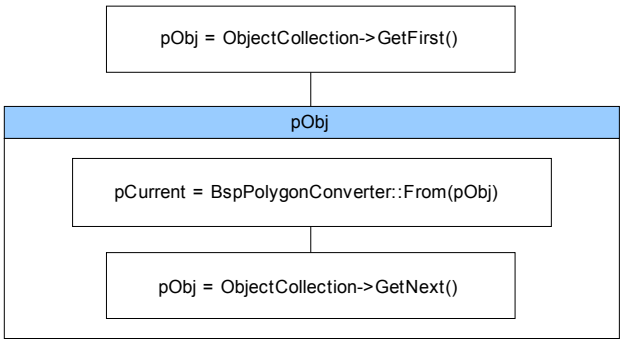
Por otro lado, la clase posee como propiedad el nodo raíz del árbol Bsp a generar:

BspNode m_rootNode;

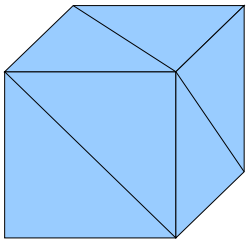
Debido a que los métodos **DrawBspTree**, **IsSolid** y **BuildBspTree** son recursivos, es conveniente crear otros métodos sin el parámetro **BspNode*** para ser invocados desde otros objetos, ellos se encargarán de realizar la llamada correspondiente introduciendo el nodo raíz, propiedad de la clase.

La construcción de árbol

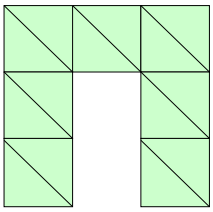
Para poder invocar el método **BuildBspTree** deberemos contar con una lista enlazada de polígonos sin ningún orden en particular. En implementación adjunta al documento se realiza la carga de un mapa en formato 3ds y se lo conviente a **BspPolygon**.



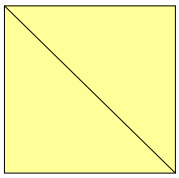
En primer lugar se recorre cada uno de los objetos de la colección de objetos 3ds del mesh del mapa cargado. Cada objeto 3ds posee un conjunto de vértices que definen lo que en el **Discreet 3d Studio Max** fue un objeto seleccionable. Pero nuestros polígonos tendrán una restricción: pueden poseer una cantidad no limitada de vértices (aunque en la implementación adjunta se limita a 4) pero **todos** deben encontrarse en el mismo plano y formar un sólido.



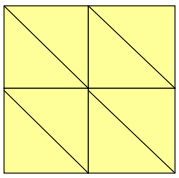
a. Conjunto total de vértices en distintos planos



b. Conjunto total de vértices en el mismo plano pero no forman un sólido



c. Conjunto total de vértices en el mismo plano formando un sólido



d. Conjunto total de vértices en el mismo plano formando un sólido

Una implementación válida sería tomar como un BspPolygon simplemente a cada triángulo del modelo, sin embargo si abundaran las construcciones como la citada en el ejemplo **d** el árbol sería más profundo de manera innecesaria (más profundidad -> más cantidad de nodos -> mayor uso de memoria -> mayor tiempo empleado en recorrer el árbol).

En la implementación adjunta se toma como un polígono triángulos y figuras como la **c** citada en la tabla anterior.

La función de conversión se encuentra colocada como método estático en una clase llamada **BspPolygonConverter**. La idea aquí es crear un tipo de dato del tipo **BspPolygon** y agregarlo a la lista enlazada (el último nodo de la lista es recibido como el segundo parámetro).

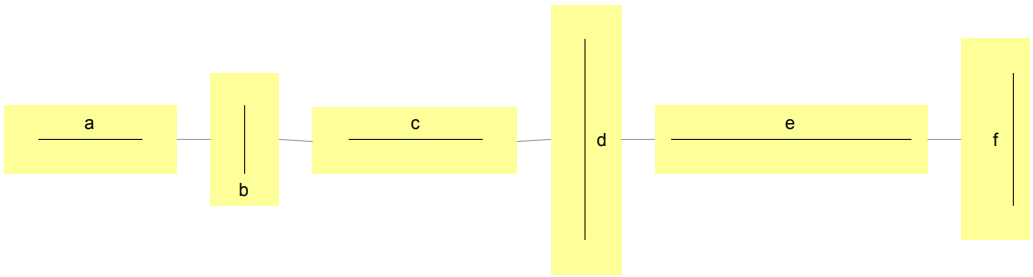
Luego se itera por todos los índices del mesh y se analiza antes de crear un nuevo BspPolygon si el objeto anterior posee tres vértices y el actual también, si existe coincidencia de dos índices y el tercer vértice se encuentra en el mismo plano que el primero del objeto anterior, entonces es un objeto tipo "Plane" de 3d Studio y se lo adjunta al BspPolygon anterior, evitando la creación de un nuevo objeto.

Notar que aquí hacemos uso de la convención que posee el 3d Studio en la descripción de figuras, por esa razón utilizamos índices específicos de objetos.

```
BspPolygon * BspPolygonConverter::From(Object3ds * pObj, BspPolygon * pFirst, int & iObjs)
{
	// El puntero pasado como parámetro debe apuntar a un nodo válido
	if (!pFirst)
		return 0;

	BspPolygon * pCurrentNode = pFirst;
	BspPolygon * pNode;

	bool bVertexDifOnPlane = false;

	// Recorro los índices del objeto 3ds
	for (int i=0; i<pObj->m_iIdxCount / 3; i++)
	{
		// ¿Existe un polígono definido anteriormente
		// con tres vértices?
		if (pCurrentNode->m_iNumberOfVerts == 3)
		{

			bVertexDifOnPlane = false;

			// Hardcode powa!
			if (pObj->m_psIdxs[1] == pObj->m_psIdxs[5] &&
				pObj->m_psIdxs[2] == pObj->m_psIdxs[4])
			{
				if (pObj->m_pVerts[0].nx == pObj->m_pVerts[3].nx &&
					pObj->m_pVerts[0].ny == pObj->m_pVerts[3].ny &&
					pObj->m_pVerts[0].nz == pObj->m_pVerts[3].nz)
				{
					// Agrego el nodo al polígono anterior
					pNode->m_iNumberOfIdxs = 6;
					pNode->m_iNumberOfVerts = 4;
					pNode->m_verts[3] = pObj->m_pVerts[pObj->m_psIdxs[3]];
					pNode->m_idxs[3] = 1;
					pNode->m_idxs[4] = 3;
					pNode->m_idxs[5] = 2;

					bVertexDifOnPlane = true;
				}

			}

		}
```

```
                if (!bVertexDifOnPlane)
                {
                        pNode = new BspPolygon;

                        pNode->m_iNumberOfIdxs = 3;
                        pNode->m_iNumberOfVerts = 3;

                        pNode->m_verts[0] = pObj->m_pVerts[pObj->m_psIdxs[i*3]];
                        pNode->m_verts[1] = pObj->m_pVerts[pObj->m_psIdxs[i*3+1]];
                        pNode->m_verts[2] = pObj->m_pVerts[pObj->m_psIdxs[i*3+2]];

                        pNode->m_idxs[0] = 0;
                        pNode->m_idxs[1] = 1;
                        pNode->m_idxs[2] = 2;

                        pNode->m_iMaterialIdx = pObj->m_iMaterialIdx;

                        pCurrentNode->m_pNext = pNode;
                        pCurrentNode = pNode;

                        iObjs++;
                }
        }

        // Marca el último nodo agregado como último de la lista
        pCurrentNode->m_pNext = NULL;

        return pCurrentNode;
}
```

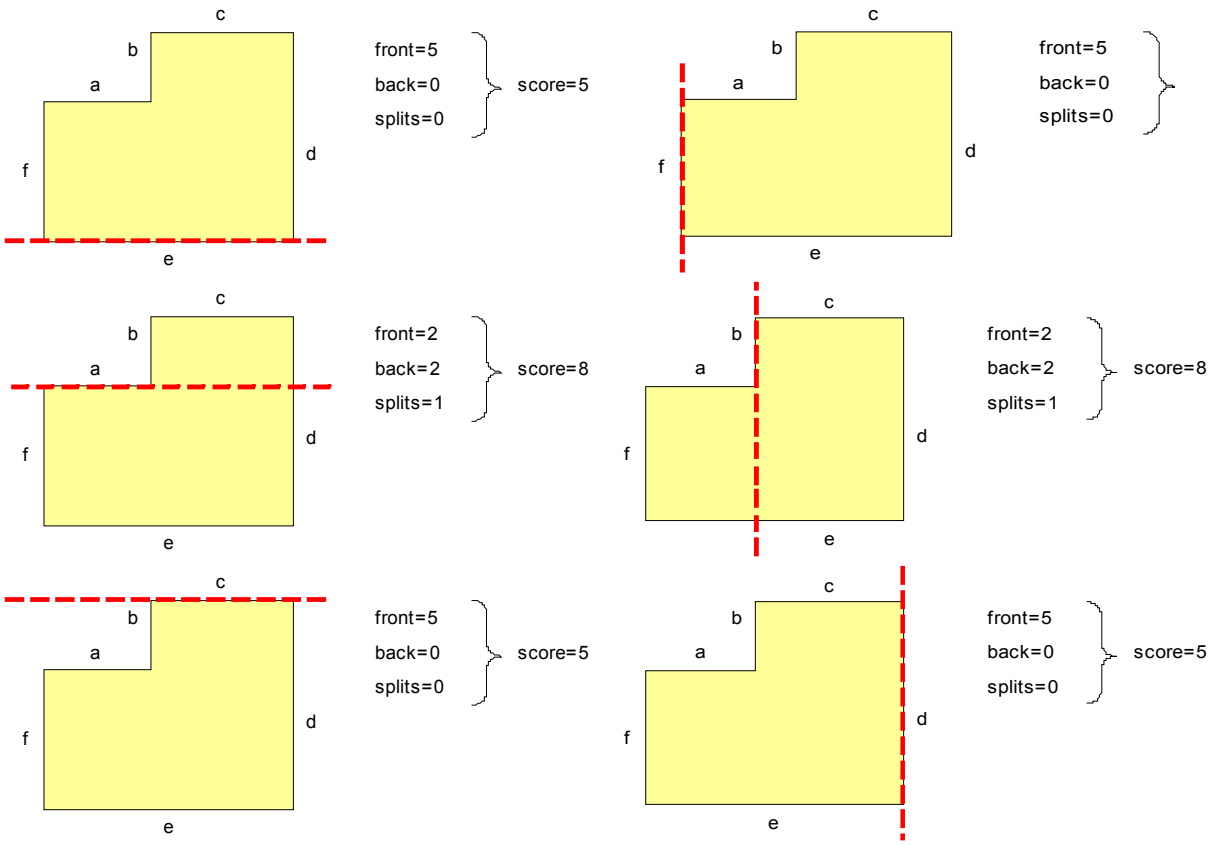
Una vez convertido el mesh 3ds a lista enlazada de polígonos tipo **BspPolygon** podremos invocar el método **BuildBspTree** de la clase **BspTree**.



Lista enlazada de polígonos

Lo primero que hará el método **BuildBspTree** será buscar un spliter "bueno" para el conjunto de polígonos recibidos. Para esto invoca el método **SelectBestSplitter** que, como mencionamos anteriormente, calcula un puntaje para cada polígono evaluado y elije el mejor.

Nuestro método con la ecuación que estamos utilizando clasificará los polígonos del siguiente modo:



Como podemos observar para nuestro mapa se penaliza "demasiado" el realizar un split. Existiendo sólo seís polígonos, se prefiere un split que no particione el subconjunto a uno que divida un polígono en dos.

Por lo tanto , tomará como splitter el polígono **e**, **f**, **c**, ó **d** (en función de cómo estén ordenados).

Tomado el splitter ingresará a un bucle donde se procesarán todos los polígonos del conjunto. Se evaluará cada uno de ellos contra el splitter (se tiene especial cuidado de uno comparar el splitter contra sí mismo), de esta evaluación se determinará si el polígono está:

- delante o en el mismo plano
- detrás
- el splitter divide al polígono en cuestión

Si el polígono se encuentra delante o en el mismo plano, lo que se hará será agregarlo a una lista de polígonos "delante".

Si el polígono se encuentra detrás, lo que se hará será agregarlo a una lista de polígonos "detrás".

Si el splitter divide al polígono, lo que se hará será crear dos polígonos nuevos, invocar al método **SplitPolygon** que completa la información de estos dos polígonos nuevos con las propiedades del polígono que les dá origen, partiendo al mismo en el lugar que indica el splitter. El polígono original se destruye y los dos polígonos nuevos va a parar a las listas "delante" y "detrás" respectivamente.

Procesados todos los polígonos, nos ha quedado las listas "delante" y "detrás".

En primer lugar tomamos la lista "delante", si está vacía creamos un nuevo nodo "hoja" y lo agregamos al nodo actual. Si no está vacía, creamos un nuevo nodo le indicamos que NO es una hoja y reinvocamos la función BuildBspTree con el nuevo nodo creado como actual y la lista de polígonos "delante" parámetro.

Luego, tomamos la lista "detrás", si está vacía creamos un nuevo nodo "hoja" y lo agregamos al nodo actual. Si no está vacía, creamos un nuevo nodo le indicamos que NO es una hoja y reinvocamos la función BuildBspTree con el nuevo nodo creado como actual y la lista de polígonos "detrás" como parámetro.

Eventualmente la llamada recursiva terminará debido a que procesados todos los polígonos del subconjunto al menos una de la listas "delante" o "detrás" quedará vacía, se creará una hoja y se retornará sin realizar ninguna llamada.

Partiendo polígonos

Es muy factible que es ciertas ocasiones la función SelectBestSplitter seleccione un polígono que divida a otro en algún punto. Cuando éste era el caso invocábamos a **SplitPolygon**, veremos ahora como trabaja este método:

void BspTree::SplitPolygon(BspPolygon * poly, BspPolygon * plane, BspPolygon * polyFrontSplit, BspPolygon * polyBackSplit);

El método **SplitPolygon** recibe como primer parámetro el polígono a dividir, como segundo parámetro un polígono que representa el plano que partirá al primer polígono. El tercer y cuarto parámetro son los polígonos que conforman la partición del primero.

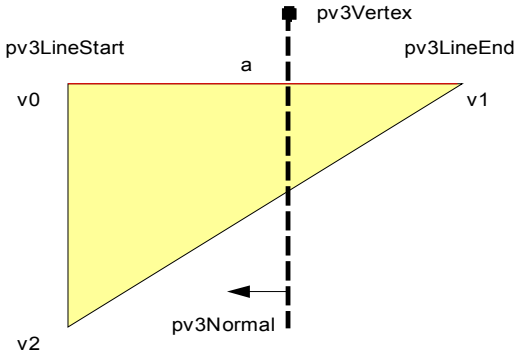
Lo que hará el método SplitPolygon será utilizar otro método llamado GetIntersect y que posee el siguiente prototipo:

bool BspTree::GetIntersect(Vector3 * pv3LineStart, Vector3 * pv3LineEnd, Vector3 * pv3Vertex, Vector3 * pv3Normal, Vector3 * pv3Intersection, float & fPercentage);

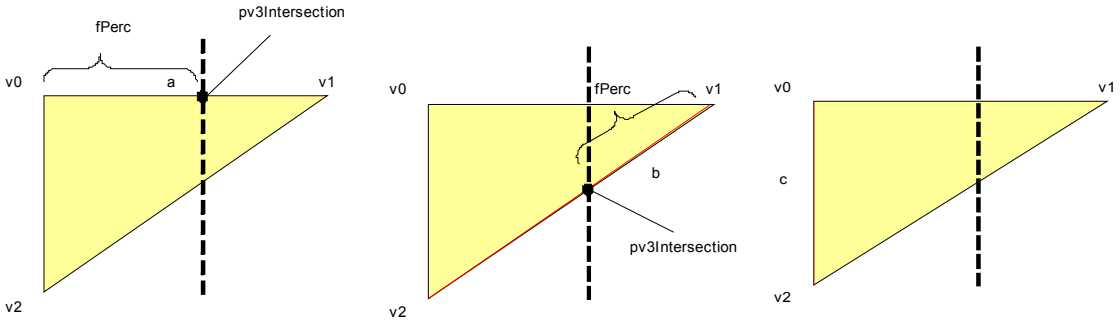
GetIntersect recibirá un segmento y un plano y retorna el punto dentro del segmento donde el plano interseca el segmento así como el porcentaje (número entre 0 y 1) a partir del primer vértice donde se produce la intersección.

El segmento se encuentra definido por dos vértices descritos en los dos primeros parámetros, el plano se encuentra descrito mediante un punto y una normal mediante el tercer y cuarto parámetro respectivamente. Los dos últimos parámetros son de salida y especifican el punto de intersección y el porcentaje.

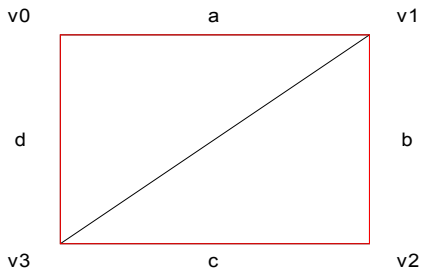
SplitPolygon tomará los bordes del polígono a partir e invocará para cada uno de ellos el método **GetIntersect** como indica la siguiente figura:



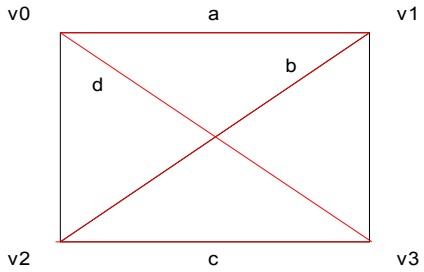
Al retornar el método **GetIntersect** se obtendrá en cada caso el punto de intersección del segmento enviado y de este modo obtendremos todos los puntos de intersección de los bordes de nuestro polígono.



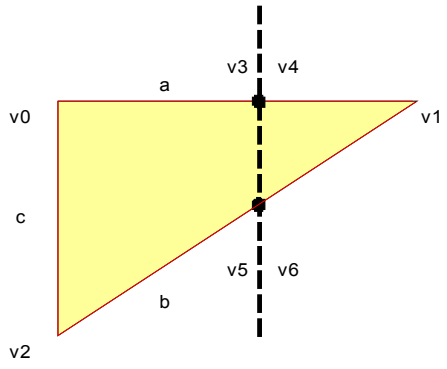
Es importante notar que si nuestro polígono posee cuatro lados, los vértices de los mismos deben estar en el siguiente orden:

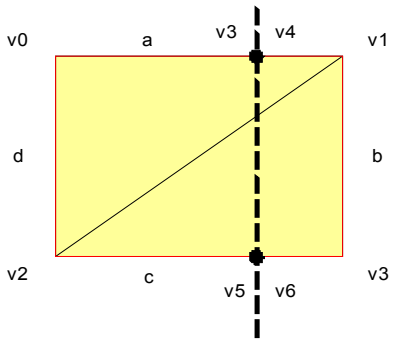


De modo que la unión de los vértices v0-v1, v1-v2, v2-v3 y v3-v0 siempre marquen los borden de la figura y no como el caso que se expresa en la siguiente figura:



Descritos los vértices correctamente un ejemplo de los nuevos vértices que se generarían serían los siguientes:





La idea ahora será agregar vértices a los nuevos polígonos, para el caso del triángulo el polígono polyFrontSplit quedará conformado por los vértices {v0, v3, v5, v2} y el polígono polyBackSplit quedará formado por {v4, v1, v6}. Mientras que el rectángulo quedará formado por polyFrontSplit = {v0, v3, v5, v2} y polyBackSplit = {v4, v1, v3, v6}.

Deberemos además especificar las nuevas coordenadas de texturas para los vértices nuevos, conociendo el porcentaje de intersección esto será muy sencillo:

```
nuevo_vert.tu = vert0.tu + (vert1.tu - vert0.tu) * fPerc;
nuevo_vert.tv = vert0.tv + (vert1.tv - vert0.tv) * fPerc;
```

Finalmente, creados los nuevos vértices con sus coordenadas de texturas, deberemos especificar los índices para formar los nuevos triángulos y que quede del siguiente modo: