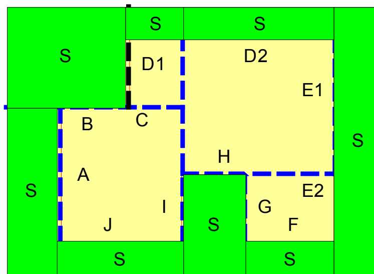

Binary Space Partitioning

parte II

Árboles Bsp basados en hojas y PVS

Autor: Diego G. Ruiz

Versión: 1.0 (draft)

Tabla de Contenidos

Introducción.. 4
Árboles BSP basados en hojas.. 4
Diferencias en el algoritmo de construcción del árbol BSP.. 4
Árbol BSP basado en hojas de nodos sólidos: lo mejor de los dos mundos.. 6
Dibujando nuestro mapa utilizando z-buffer y PVS.. 6
Conjuntos de polígonos potencialmente visibles (PVS).. 7
Inconvenientes de implementación.. 9
El algoritmo Zero Run Length.. 9
Creando el PVS.. 10
Generación de portales.. 10
Frustum culling, la última frontera.. 11

Binary Space Partitioning (BSP)

parte II

Introducción

En este documento veremos como se pueden utilizar los árboles BSP para eliminar polígonos no visibles.

Sabemos que todo polígno que no se encuentre dentro de la escena visualizada será finalmente recortado (clipeado) por Direct3D, sin embargo para que Direct3D haga esto deberá primero transformar cada uno de los vértices que conforma el objeto, si nosotros podemos establecer mediante un algoritmo sencillo que un objeto no será visualizado en escena podemos ahorrar todo este tiempo de procesamiento inútil y dedicarselo a colocar una mayor cantidad de polígonos en pantalla para lograr una mejor calidad gráfica.

La idea será combinar un árbol BSP de geometría sólida como el visto en la parte I del documento con información de conjuntos potencialmente visibles (PVS; Potential Visibility Sets). De este modo podremos descartar prácticamente toda la geometría no visible.

De este modo también independizaremos, en gran medida, el tamaño del mapa cargado con la velocidad de dibujado del frame.

Árboles BSP basados en hojas

Los árboles BSP basados en hojas (en inglés: Leafy BSP Trees o Leaf based BSP Trees) es el tipo de árbol BSP que utilizan juegos como el Quake y Half-Life.

Esta variedad de árbol BSP agrupa polígonos que finalmente no dejan detrás a ningún otro, es decir, llegado un momento deberemos seleccionar un splitter para la división de un subconjunto de polígonos, eventualmente descubriremos que todo polígono que se tome como splitter deja delante a todos los demás, por lo tanto esto quiere decir que no existen inconvenientes en el orden de dibujado de dicho subconjunto ya que ningún polígono oculta a ningún otro, finalmente no tiene sentido seguir generando nodos y todos estos polígonos quedan agrupados en una hoja.

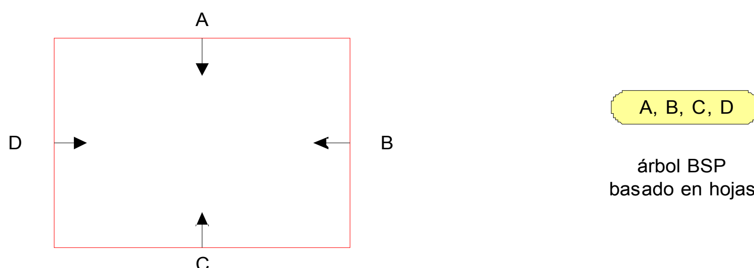

Figura 1. Ejemplo de un árbol BSP basado en hojas

Diferencias en el algoritmo de construcción del árbol BSP

Además de la agrupación de polígonos citada en el párrafo anterior, existe una diferencia en el modo de procesar nodos al que veníamos empleando. Anteriormente cada uno de los nodos donde se colocaba un splitter, poseía el polígono que actuaba como tal y era uno del subconjunto de polígonos que había ingresado a nuestra función de construcción; dicho polígono una vez que era seleccionado como splitter era quitado de la lista de polígonos y luego se volvía a invocar la función de construcción con el subconjunto izquierdo por un lado y el subconjunto derecho por otro.

Ahora el polígono que actuó como splitter NO es extraido de la lista de polígonos sino que se agregado a la lista de polígonos que se encuentran delante del splitter. Y además poseerá una propiedad adicional que indicará que dicho polígono ya fue un splitter y NO podrá volver a serlo en futuras particiones del espacio.

División de un segmento que fue splitter
Es posible que un polígono que fue splitter sea dividido en dos por medio de otro polígono en algún momento de la construcción del árbol. En dicho caso la propiedad que indica "este polígono ya fue splitter" debe ser heredada a los dos nuevos polígonos creados.

Veamos un ejemplo:

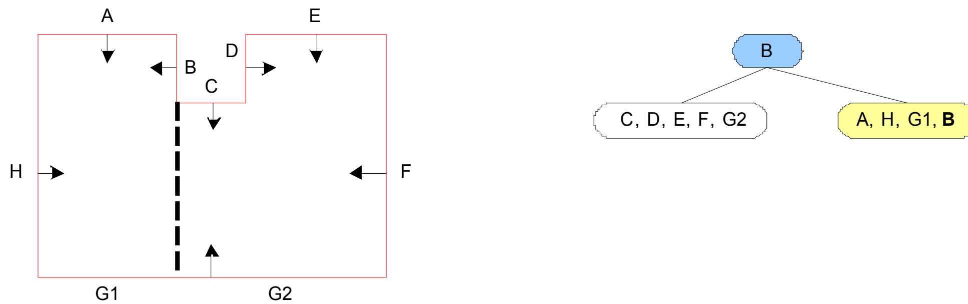

Figura 2. El polígono B es incluido en el subconjunto de polígonos "delante".

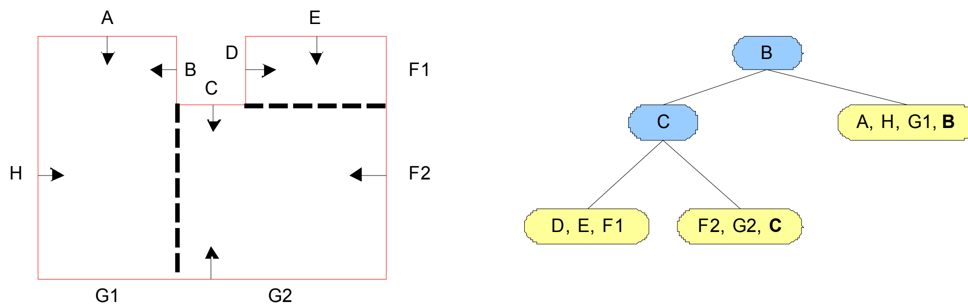

Figura 3. El polígono C es incluido en el subconjunto de polígonos "delante"

En el ejemplo anterior nuestro árbol BSP basado en hojas quedó conformado por tres hojas, dichas hojas poseen subconjuntos de polígonos que puede dibujarse en cualquier orden. Veamos como hubiese quedado el mismo mapa creado por medio de un árbol BSP de nodo sólido:

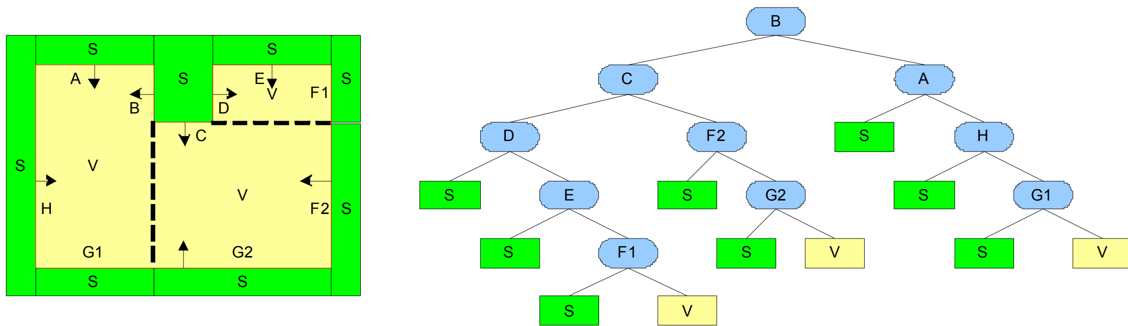

Figura 4. El mismo mapa creado por medio de un árbol BSP de nodo sólido

Como se puede apreciar el árbol BSP basado en hojas posee menos nodos (y menos profundidad) que el árbol BSP de nodo sólido ¡y esto es bueno! Siempre deseamos que nuestro árbol BSP sea pequeño debido a que de este modo será más rápido su recorrido y por lo tanto se realizará en menor tiempo.

Sin embargo, existían varias razones que justificaban la creación de árbol BSP de nodo sólido, el mismo lo utilizaríamos para verificar colisiones y verificar si dos objetos dentro del mapa se veían entre sí.

Pero el árbol BSP basado en hojas nos será útil para el cálculo de los PVS ¿Qué hacer entonces?

Árbol BSP basado en hojas de nodos sólidos: lo mejor de los dos mundos

Realizaremos una mezcla de los dos tipos de árboles y crearemos un árbol basado en hojas de nodo sólido (en inglés: Solid Leaf BSP Tree). Veamos como queda nuestro ejemplo:

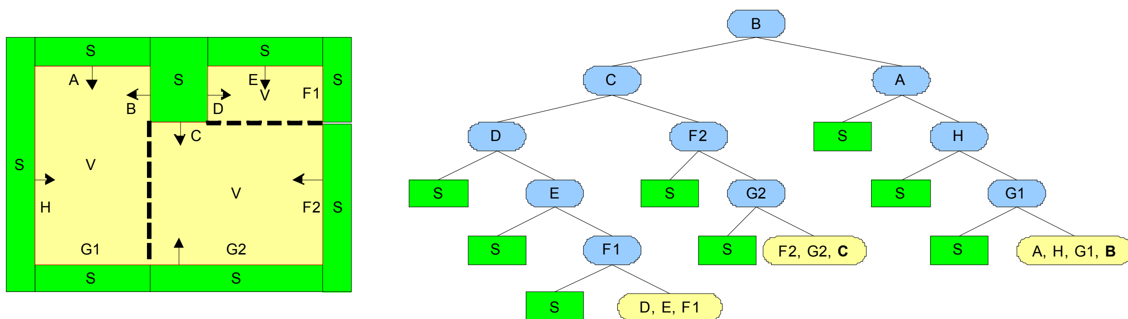

Figura 5. Nuestro flamante árbol BSP basado en hojas de nodos sólidos

Como se puede apreciar en la figura 5 la información de "vacíos" ya no forma parte de nuestro árbol y la misma fue reemplazada por el conjunto de polígonos que conforman un subespacio convexo (polígonos que no se tapan entre sí).

De este modo nuestro árbol sigue siendo útil para el cálculo de colisiones y también lo es para el cálculo de PVS.

Dibujando nuestro mapa utilizando z-buffer y PVS

Habíamos comentado que inicialmente el árbol BSP era utilizado para ordenar polígonos y dibujar una escena de atrás hacia delante debido a que las placas de video de aquella época no

poseían aceleración por z-buffer. Sin embargo, existiendo z-buffer es conveniente realizar lo contrario, es decir, dibujar desde adelante hacia atrás (con exepción de polígonos semitransparentes) debido a que de este modo haremos un uso conveniente del z-buffer ya que los polígonos rechazados por el z-buffer no serán rasterizados.

Por lo tanto, nuestro árbol BSP nos será muy útil pero lo utilizaremos de un modo distinto al empleado en los primeros juegos que lo implemetaron (como el **Doom** de **Id Software**).

Conjuntos de polígonos potencialmente visibles (PVS)

Hemos comentado que los PVS nos ayudarían a pasar por alto vértices que formen parte de objetos no visibles en una escena. De este modo aumentaremos el frame rate debido a que no deberemos transformar vértices que finalmente será clippeados.

Inicialmente podemos pensar que la estructura PVS se trata simplemente de un array de booleanos situado en cada hoja de nuestro árbol que nos indica que región del mismo dibujar y que región no dibujar. Más adelante veremos como generar este array, pero ahora detengámonos a analizar este array.

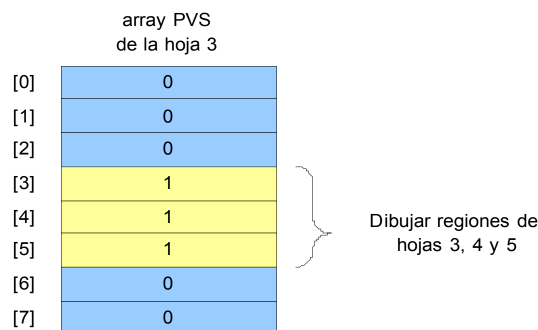

Figura 6. Un array PVS

Ahora, ya no dibujaremos el árbol BSP entero sino que llegada a la hoja donde se encuentra posicionada la cámara, realizaremos un lookup en el array PVS y dibujaremos sólo las hojas que correspondan ¡sin ningún otro procesamiento adicional!

El pseudocódigo de consulta será el siguiente:

```
void DrawTree(long lCameraLeaf)
{
   long lOffset;
   for (int i=0; i<iNumberOfLeafs; i++)
   {
      lOffset = lCameraLeaf * iNumberOfLeafs;
      if (PVSData[lOffset + i])
         RenderLeaf(i);
```

```
  }
}
```

En el código anterior asumimos que existen un array global o accesible desde la función **DrawTree** con la información PVS para todas las hojas.

Es importante notar que la información de PVS es independiente de la vista, es decir que en principio dibujaremos todas las regiones que se podrían ver desde otra en cualquier posición de la misma, efectivamente es posible que esto termine no siendo así y más adelante veremos como solucionar este inconveniente.

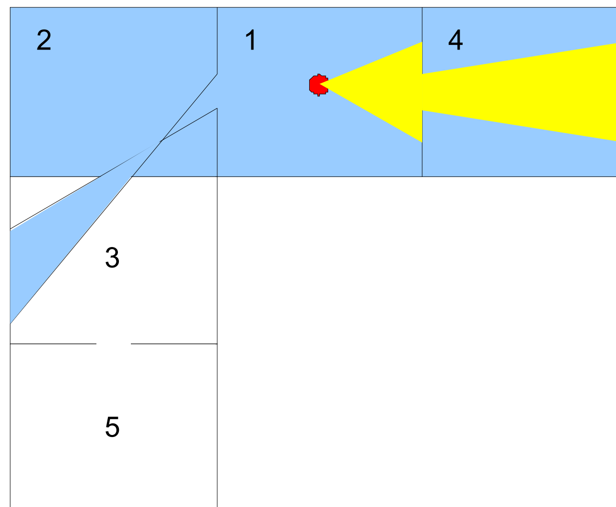

Figura 7. Un mapa con PVS

En la figura anterior podemos observar un mapa donde las áreas azules indican que es posible ver desde cualquier posición de la región 1, mientras que las áreas amarillas indican que es posible ver desde la posición de la cámara. Por lo tanto, si bien desde la posición de la cámara sólo se ve la región 1 y 4, nuestro PVS indica que debemos dibujar la región 1, 2, 3 y 4.

Si por debajo de la región 5 se extendiesen 100 regiones más las mismas serían rechazadas inmediatamente sin procesar un solo polígono de manera inmediata.

Por lo tanto el PVS reduce la cantidad de polígonos que procesaremos, aunque es claro que su efectividad dependerá en gran medida de cómo se encuentre construido el mapa. Siempre existen recomendaciones de diseño que favorezcan de algún modo la cantidad de vértices a procesar, tengamos en cuenta el siguiente ejemplo:

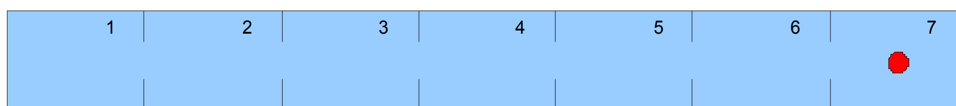

Figura 8

Desde la habitación 7 es posible ver todas las demás, por lo tanto deberíamos procesar todos los polígonos. Favorezcamos ahora la generación de información de visibilidad mediante un pequeño cambio:

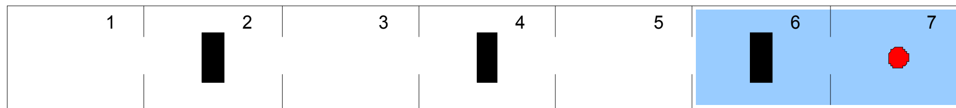

Figura 9

Ahora, evitando que desde la habitación 7 se vea la 5 (desde cualquier posición) por medio de una pared situada en el centro de la habitación 6 mejoramos muchísimo nuestra información de visibilidad.

Inconvenientes de implementación

La estructura PVS es una gran ayuda para nuestro motor a la hora de descartar polígonos que finalmente no serán visibles. Sin embargo, hagamos una cuenta sencilla, en un árbol BSP con 6000 hojas, el array PVS como es un todos contra todos debería tener 6000 * 6000 elementos, es decir 36 millones de elementos, si cada elemento es un byte (debido a que simplemento es 1 o 0) el espacio total ocupado por nuestro PVS es de poco más de 4.5 Mb, creo que no hace falta mencionar que esto es demasiado.

Lo que haremos entonces será utilizar algún método de compresión de datos en memoria, como es el Zero Run Length.

El algoritmo Zero Run Length

El algoritmo Zero Run Length es muy beneficioso para nuestros fines debido a que si el mapa es muy grande (el caso de un mapa con gran cantidad de hojas) es muy posible que desde una habitación sólo podamos ver un subconjunto muy pequeño de habitaciones, esto quiere decir que nuestro array estará poblado de ceros.

Un algoritmo del tipo RL se basa en colocar la cantidad de repeticiones de un elemento y luego el elemento en cuestión, por lo tanto si existe una repetición considerable de muchos elementos el tamaño final de la estructura de datos utilizada será considerablemente menor. El ZRL, que es un algoritmo particular basado en RL, se basa en este mismo principio sólo que para el número 0, como nuestro array estará poblado de ceros, entonces el porcentual de compresión total debería ser un número alto. Veamos un ejemplo para entender como funciona este algoritmo:

Nuestro PVS sin comprimir:

4	3	7	8	0	0	0	0	0	0	0	0	3	5	0	0	0	0	0	0

Cada celda de la tabla es un byte, es decir 8 bits formados por 1 y 0 indicando si la región es visible o no.

Nuestro PVS comprimido mediante ZRL:

4	3	7	8	0	8	3	5	0	6

¿Qué hemos hecho? Como se puede apreciar las zonas sin ceros del array permanecen inalteradas, la compresión se realiza cuando existe una gran cantidad de ceros adyacentes, en

dicho caso se coloca UN cero (tomando como una especie de caracter de escape, algo así como la barra invertida (\ ) dentro de un texto encerrado en comillas en lenguaje C/C++) y luego el siguiente número se interpreta como la CANTIDAD de ceros que deberían seguir.

Lo bueno de todo esto es que el algoritmo de compresión es tan sencillo, que para realizar una consulta en un nuestro "nuevo" array PVS no deberemos realizar una descompresión explíticita a otra zona de memoria, sino que trabajeremos directamente sobre el mismo.

¿Y por que con el cero?

Imaginemos una casa de muchos pisos poblada de habitaciones, si usted arma en una hoja de papel el array PVS se dará cuenta que está poblado de ceros ¿por qué? Porque desde una habitación sólo podrá ver una cuantas de toda la casa, por ejemplo, si la casa tuviese 50 habitaciones es probable que usualmente de una habitación no vea más de dos o tres. Entonces el algoritmo ZRL reduciría la cantidad de memoria necesaria considerablemente.

¿Y si estuviésemos en exteriores?

En exteriores se da la particularidad que existen pocas oclusiones, es decir, todo se ve de todos lados. Por lo tanto, utilizar PVS e incluso BSPs puede no ser la mejor opción en esos casos o al menos no obtendremos tantos beneficios como cuando estamos en interiores.

Creando el PVS

Para crear un PVS deberemos primero realizar la generación de portales.

Generación de portales

Para poder calcular que hoja se ve con que hoja deberemos conocer que hay entre medio de ambas, ya que en principio nuestro árbol estará compuesto por hojas con conjuntos de polígonos describiendo subespacios convexos ¿pero que región se ve con cual? Para esto deberemos generar un conjunto temporal de polígonos que describan el "agüjero" que queda en una región mirando a otra y a esta estructura se la denomina **portal**.

Los polígonos que forman el portal no son dibujados, de hecho ni bien terminemos de calcular el PVS serán eliminados.

En la siguiente figura se puede apreciar cuales serán los portales del mapa; los mismos están en la "unión" de los distintos subespacios convexos.

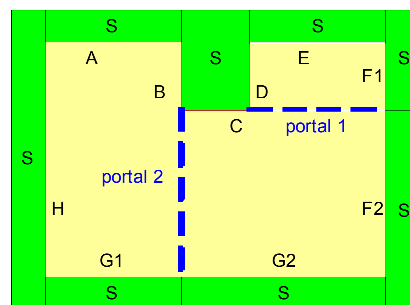

Figura 10. Los portales de nuestro mapa

Para crear un portal deberemos proceder del siguiente modo:

1. Primero deberemos crear un gran polígono por cada splitter de nuestro árbol, de modo que este polígono abarque los polígonos de todas las hojas que posee delante de sí. Haremos esto por cada nodo, este polígono irá "bajando" por el árbol y reduciéndose en tamaño hasta llegar a ser el portal que deseábamos construir.

2. Los portales que terminen en un sólido serán eliminados ya que no puede existir una ventana mirando a una pared. La idea de cómo trabajaremos será la siguiente:

Evaluaremos el portal en cada nodo para verificar si se encuentra delante, detrás o en el mismo plano que el plano splitter de dicho nodo:

Delante o en plano: Si el portal (o potencial portal) se encuentra delante del splitter entonces mandaremos al mismo hacia delante (esto significa pasarla la estructura de datos como parámetro a una llamada recursiva). Todo portal válido debe terminar en dos hojas, es decir que de describen de a pares, si un portal no posee su par en otra hoja el mismo es eliminado.

Detrás: El portal está detrás del splitter, por lo tanto lo enviamos por dicha rama. A no ser que el nodo que se encuentre detrás sea un sólido.

Splitting: El portal es partido por otro plano, por lo tanto hay que dividirlo y enviar cada parte hacia el nodo que corresponda.

Frustum culling, la última frontera

Como hemos comentado, el PVS retorna la información de visibilidad desde cualquier parte de la región, sin embargo es posible que realmente no estemos viendo todo lo que dicha estructura nos señala, para evitar procesar vértices demás la última técnica que nos queda es el **frustum culling** de cada región. [frustumculling.doc]

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