BSPFaq

COLLABORATORS			
	TITLE : BSPFaq		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 14, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

BSPFaq

1.1 BSP Tree Frequently Asked Questions (FAQ)

1.2 Changes

* Changes

Date Section Change

08/21/96 "Online Resources" Added a reference to Paton Lewis's Java based BSP tree demo applet.

08/07/96 "Online Resources" Added a reference to the Win95 BSP Tree Demo Application.

07/24/96 "Online Resources" LINK http___www.qualia.com_cgi-bin_bspfaq_bsp?24.txt}Added reference to Michael Abrash's ftp site at Id.

07/11/96 "Online Resources" LINK http___www.qualia.com_cgi-bin_bspfaq_bsp?24.txt}Added reference to Andrea Marchini and Stefano Tommesani's BSP tree compiler page.

05/01/96 "General" The FAQ articles may now be annotated using the forum mechanism.

04/28/96 "Forum" Experimental new discussion area for BSP trees.

04/24/96 "General" Added "Next" and "Previous" links on each page of the FAQ.

04/17/96 "Whole FAQ" The web search engines have been pointing a lot of people at the entire listing version of the FAQ, rather than at the indexed version. This has led to significantly increased load on our server, and slow response times. As a

04/12/96 "Online Resources" Update on A.T. Campbell's resources

04/08/96 "Eliminating Recursion" Initial Draft with code example

03/25/96 "General" White pages

03/22/96 "Online Resources" A.T. Campbell's home page, Update Mel Slater's location

03/21/96 "Contribution" Corrected e-mail address

"Online Resources" Arcball FTP site Paper by John Holmes, Jr.

02/19/96 "Ray Tracing" Draft implementation notes

"Analytic Visibility" Draft contents

"Boolean Operations" Spelling corrections

--

Last Update: 08/21/96 23:08:36

1.3 About this document

* About this document

General

The purpose of this document is to provide answers to Frequently Asked Questions about Binary Space Partitioning (BSP) Trees. This document will be posted monthly to "comp.graphics.algorithms". It is also available via WWW at the URL:

http://www.qualia.com/bspfaq/

The most recent newsgroup posting of this document is available via ftp at the following URLs:

file://www.qualia.com/pub/bspfaq/bspfaq.txt

ftp://rtfm.mit.edu/pub/usenet/news.answers/graphics/bsptree-faq

Requesting the FAQ by mail

You can request that the FAQ be mailed to you in plain text and HTML formats by sending e-mail to "bspfaq@qualia.com" with a subject line of "SEND BSP TREE [what]". The "[what]" should be replaced with any combination of "TEXT" and "HTML". Respectively, t

Copyrights and distribution

This document is maintained by Bretton Wade, lead developer for Qualia, Incorporated, and a graduate of the Cornell University Program of Computer Graphics. This resource is provided as a service to the computing community in the interest of diss

Warranty and disclaimer

This article is provided as is without any express or implied warranties. While every effort has been taken to ensure the accuracy of the information contained in this article, the author/maintainer/contributors assume(s) no responsibility for errors or

The contents of this article do not necessarily represent the opinions of Qualia, Incorporated.

Conversion to AmigaGuide-format by Heikki Pora (pora@lut.fi) 09/26/96

--

Last Update: 05/01/96 11:13:53

1.4 Acknowledgements

* Acknowledgements"

Web Space

Thank you to Qualia, Incorporated for kindly providing the web space for this document free of charge.

About the contributors

This document would not have been possible without the selfless contributions and efforts of many individuals. I would like to take the opportunity to thank each one of them. Please be aware that these people may not be amenable to recieving e-mail on a r

Contributors

* Bruce Naylor (naylor@research.att.com)

* Richard Lobb (richard@cs.auckland.ac.nz)

* Dani Lischinski (danix@cs.washington.edu)

* Chris Schoeneman (crs@lightscape.com)

* Jim Arvo (arvo@graphics.cornell.edu)

* Kevin Ryan (kryan@access.digex.net)

* Lukas Rosenthaler (rosenth@foto.chemie.unibas.ch)

* Anson Tsao (ansont@hookup.net)

* Ron Capelli (capelli@vnet.ibm.com)

* Ian CR Mapleson (mapleson@cee.hw.ac.uk)

* Steve Larsen (larsen@sunset.cs.utah.edu)

* Timothy Miller (tsm@cs.brown.edu)

* Richard Matthias (richardm@cogs.susx.ac.uk)

* Ken Shoemake (shoemake@graphics.cis.upenn.edu)

* Seth Teller (seth@theory.lcs.mit.edu)

* Peter Shirley (shirley@graphics.cornell.edu)

* Michael Abrash (mikeab@idece2.idsoftware.com)

* Robert Schmidt (robert@idt.unit.no)

If I have neglected to mention your name, and you contributed, please let me know immediately!

--

Last Update: 05/01/96 11:13:53

1.5 How can you contribute?

* How can you contribute?

Please send all new questions, corrections, suggestions, and contributions "bspfaq@qualia.com"

--

Last Update: 05/01/96 11:13:53

1.6 About the pseudo C++ code

* About the pseudo C++ code

Overview

The general efficiency of C++ makes it a well suited language for programming computer graphics. Furthermore, the abstract nature of the language allows it to be used effectively as a psuedo code for demonstrative purposes. I will use C++ notation for all

In order to provide effective examples, it is necessary to assume that certain classes already exist, and can be used without presenting excessive details of their operation. Basic classes such as lists and arrays fall into this category.

Other classes which will be very useful for examples need to be presented here, but the definitions will be generic to allow for freedom of interpretation. I assume points and vectors to each be an array of 3 real numbers (X, Y, Z).

Planes are represented as an array of 4 real numbers (A, B, C, D). The vector (A, B, C) is the normal vector to the plane. Polygons are structures composited from an array of points, which are the vertices, and a plane.

The overloaded operator for a dot product (inner product, scalar product, etc.) of two vectors is the ’|’ symbol. This has two advantages, the first of which is that it can’t be confused with the scalar multiplication operator. The second is that precede

The code for BSP trees presented here is intended to be educational, and may or may not be very efficient. For the sake of clarity, the BSP tree itself will not be defined as a class.

--

Last Update: 05/01/96 11:13:53

1.7 What Is a BSP Tree?

* What is a BSP Tree?"

Overview

A Binary Space Partitioning (BSP) tree is a data structure that represents a recursive, hierarchical subdivision of n-dimensional space into convex subspaces. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane th

A "hyperplane" in n-dimensional space is an n-1 dimensional object which can be used to divide the space into two half-spaces. For example, in three dimensional space, the "hyperplane" is a plane. In two dimensional space, a line is used.

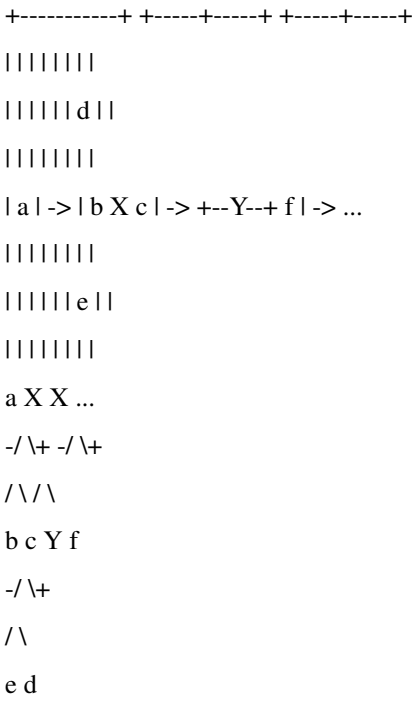
BSP trees are extremely versatile, because they are powerful sorting and classification structures. They have uses ranging from hidden surface removal and ray tracing hierarchies to solid modeling and robot motion planning.

Example

An easy way to think about BSP trees is to limit the discussion to two dimensions. To simplify the situation, let's say that we will use only lines parallel to the X or Y axis, and that we will divide the space equally at each node. For example, given a s

[Image]

or the ASCII art version:



Other space partitioning structures

BSP trees are closely related to Quadtrees and Octrees. Quadtrees and Octrees are space partitioning trees which recursively divide subspaces into four and eight new subspaces, respectively. A BSP Tree can be used to simulate both of these structures.

--

Last Update: 05/01/96 11:13:53

1.8 How do you build a BSP Tree?

* How do you build a BSP Tree?

Overview

Given a set of polygons in three dimensional space, we want to build a BSP tree which contains all of the polygons. For now, we will ignore the question of how the resulting tree is going to be used.

The algorithm to build a BSP tree is very simple:

* Select a partition plane.

* Partition the set of polygons with the plane.

* Recurse with each of the two new sets.

Choosing the partition plane

The choice of partition plane depends on how the tree will be used, and what sort of efficiency criteria you have for the construction. For some purposes, it is appropriate to choose the partition plane from the input set of polygons. Other applications m

In any case, you want to evaluate how your choice will affect the results. It is desirable to have a balanced tree, where each leaf contains roughly the same number of polygons. However, there is some cost in achieving this. If a polygon happens to span t

Partitioning polygons

Partitioning a set of polygons with a plane is done by classifying each member of the set with respect to the plane. If a polygon lies entirely to one side or the other of the plane, then it is not modified, and is added to the partition set for the side

When to stop

The decision to terminate tree construction is, again, a matter of the specific application. Some methods terminate when the number of polygons in a leaf node is below a maximum value. Other methods continue until every polygon is placed in an internal no

Pseudo C++ code example

Here is an example of how you might code a BSP tree:

```
struct BSP_tree
{
plane partition;
list polygons;
BSP_tree *front,
*back;
```

This structure definition will be used for all subsequent example code. It stores pointers to its children, the partitioning plane for the node, and a list of polygons coincident with the partition plane. For this example, there will always be at least on

```
void Build_BSP_Tree (BSP_tree *tree, list polygons)
{
polygon *root = polygons.Get_From_List ();
tree->partition = root->Get_Plane ();&
tree->polygons.Add_To_List (root);&
list front_list,
back_list;
polygon *poly;
while ((poly = polygons.Get_From_List ()) != 0)
{
int result = tree->partition.Classify_Polygon (poly);&
switch (result)
{
case COINCIDENT:
tree->polygons.Add_To_List (poly);&
break;
case IN_BACK_OF:
backlist.Add_To_List (poly);
break;
case IN_FRONT_OF:
frontlist.Add_To_List (poly);
```

```
break;
case SPANNING:
polygon *front_piece, *back_piece;
Split_Polygon (poly, tree->partition, front_piece, back_piece);&
backlist.Add_To_List (back_piece);
frontlist.Add_To_List (front_piece);
break;
}
}
if ( ! front_list.Is_Empty_List ())
{
tree->front = new BSP_tree;&
Build_BSP_Tree (tree->front, front_list);&
}
if ( ! back_list.Is_Empty_List ())
{
tree->back = new BSP_tree;&
Build_BSP_Tree (tree->back, back_list);&
}
}
```

This routine recursively constructs a BSP tree using the above definition. It takes the first polygon from the input list and uses it to partition the remainder of the set. The routine then calls itself recursively with each of the two partitions. This im

One obvious improvement to this example is to choose the partitioning plane more intelligently. This issue is addressed separately in the section, "How can you make a BSP Tree more efficient?".

--

Last Update: 05/01/96 11:13:53

1.9 How do you partition a polygon with a plane?

* How do you partition a polygon with a plane?

Overview

Partitioning a polygon with a plane is a matter of determining which side of the plane the polygon is on. This is referred to as a front/back test, and is performed by testing each point in the polygon against the plane. If all of the points lie to one si

The basic algorithm is to loop across all the edges of the polygon and find those for which one vertex is on each side of the partition plane. The intersection points of these edges and the plane are computed, and those points are used as new vertices for

Implementation notes

Classifying a point with respect to a plane is done by passing the (x, y, z) values of the point into the plane equation, Ax + By + Cz + D = 0. The result of this operation is the distance from the plane to the point along the plane's normal vector. It wi

For those not familiar with the plane equation, The values A, B, and C are the coordinate values of the normal vector. D can be calculated by substituting a point known to be on the plane for x, y, and z.

Convex polygons are generally easier to deal with in BSP tree construction than concave ones, because splitting them with a plane always results in exactly two convex pieces. Furthermore, the algorithm for splitting convex polygons is straightforward and

Pseudo C++ code example

Here is a very basic function to split a convex polygon with a plane:

```
void Split_Polygon (polygon *poly, plane *part, polygon *&FRONT, POLYGON *&BACK)
{
int count = poly->NumVertices (),&
out_c = 0, in_c = 0;
point ptA, ptB,
outpts[MAXPTS],
inpts[MAXPTS];
real sideA, sideB;
ptA = poly->Vertex (count - 1);&
sideA = part->Classify_Point (ptA);&
for (short i = -1; ++i
{
ptB = poly->Vertex (i);&
sideB = part->Classify_Point (ptB);&
if (sideB > 0)&
{
if (sideA
{
// compute the intersection point of the line
// from point A to point B with the partition
// plane. This is a simple ray-plane intersection.
vector v = ptB - ptA;
real sect = - part->Classify_Point (ptA) / (part->Normal () | v);&
outpts[out_c++] = inpts[in_c++] = ptA + (v * sect);
}
outpts[out_c++] = ptB;
}
else if (sideB
{
if (sideA > 0)&
{
// compute the intersection point of the line
// from point A to point B with the partition
// plane. This is a simple ray-plane intersection.
```

vector v = ptB - ptA;

real sect = - part->Classify_Point (ptA) / (part->Normal () | v);&

outpts[out_c++] = inpts[in_c++] = ptA + (v * sect);

}

inpts[in_c++] = ptB;

}

else

outpts[out_c++] = inpts[in_c++] = ptB;

ptA = ptB;

sideA = sideB;

}

front = new polygon (outpts, out_c);

back = new polygon (inpts, in_c);

A simple extension to this code that is good for BSP trees is to combine its functionality with the routine to classify a polygon with respect to a plane.

Note that this code is not robust, since numerical stability may cause errors in the classification of a point. The standard solution is to make the plane "thick" by use of an epsilon value.

--

Last Update: 05/01/96 11:13:53

1.10 How do you remove hidden surfaces with a BSP Tree?

* How do you remove hidden surfaces with a BSP Tree?

Overview

Probably the most common application of BSP trees is hidden surface removal in three dimensions. BSP trees provide an elegant, efficient method for sorting polygons via a depth first tree walk. This fact can be exploited in a back to front "painter's algo

BSP trees are well suited to interactive display of static (not moving) geometry because the tree can be constructed as a preprocess. Then the display from any arbitrary viewpoint can be done in linear time. Adding dynamic (moving) objects to the scene is

Painter's algorithm

The idea behind the painter's algorithm is to draw polygons far away from the eye first, followed by drawing those that are close to the eye. Hidden surfaces will be written over in the image as the surfaces that obscure them are drawn. One condition for

+------+

| |

+---------------| |--+

| | | |

| | | |

| | | |

| +--------| |--+

| | | |

+--| |--------+ |

| | | |

| | | |

| | | |

+--| |---------------+

| |

One reason that BSP trees are so elegant for the painter’s algorithm is that the splitting of difficult polygons is an automatic part of tree construction. Note that only one of these two polygons needs to be split in order to resolve the problem.

To draw the contents of the tree, perform a back to front tree traversal. Begin at the root node and classify the eye point with respect to its partition plane. Draw the subtree at the far child from the eye, then draw the polygons in this node, then draw

Scanline hidden surface removal

It is just as easy to traverse the BSP tree in front to back order as it is for back to front. We can use this to our advantage in a scanline method method by using a write mask which will prevent pixels from being written more than once. This will repres

The trick to making a scanline approach successful is to have an efficient method for masking pixels. One way to do this is to maintain a list of pixel spans which have not yet been written to for each scan line. For each polygon scan converted, only pixe

The scan line spans can be represented as binary trees, which are just one dimensional BSP trees. This technique can be expanded to a two dimensional screen coverage algorithm using a two dimensional BSP tree to represent the masked regions. Any convex pa

Implementation notes

When building a BSP tree specifically for hidden surface removal, the partition planes are usually chosen from the input polygon set. However, any arbitrary plane can be used if there are no intersecting or concave polygons, as in the example above.

Pseudo C++ code example

Using the BSP_tree structure defined in the section, "How do you build a BSP Tree?", here is a simple example of a back to front tree traversal:

```
void Draw_BSP_Tree (BSP_tree *tree, point eye)
{
real result = tree->partition.Classify_Point (eye);&
if (result > 0)&
{
Draw_BSP_Tree (tree->back, eye);&
tree->polygons.Draw_Polygon_List ();&
Draw_BSP_Tree (tree->front, eye);&
}
else if (result
{
Draw_BSP_Tree (tree->front, eye);&
tree->polygons.Draw_Polygon_List ();&
Draw_BSP_Tree (tree->back, eye);&
}
else // result is 0
{
// the eye point is on the partition plane...
```

Draw_BSP_Tree (tree->front, eye);&

Draw_BSP_Tree (tree->back, eye);&

}

If the eye point is classified as being on the partition plane, the drawing order is unclear. This is not a problem if the Draw_Polygon_List routine is smart enough to not draw polygons that are not within the viewing frustum. The coincident polygon list

It is possible to substantially improve the quality of this example by including the viewing direction vector in the computation. You can determine that entire subtrees are behind the viewer by comparing the view vector to the partition plane normal vecto

Front to back tree traversal is accomplished in exactly the same manner, except that the recursive calls to Draw_BSP_Tree occur in reverse order.

--

Last Update: 05/01/96 11:13:53

1.11 How

* How do you compute analytic visibility with a BSP Tree?

Overview

Analytic visibility is a term which describes the list of surfaces visible from a single point in a scene. Analytic visibility is important to the architectural community because it may be necessary to obtain a visible lines only view of a building for o

BSP trees can be used to compute visible fragments of polygons in a scene in at least two different ways. Both methods involve the use of a bsp tree for front to back traversal, and a second tree which describes the visible space in the viewing volume.

Screen partitioning

This method uses a two dimensional BSP tree to partition the viewing plane into regions which have and have not been covered by previously rendered polygons. Whenever a polygon is rendered, it is inserted into the screen tree and clipped to the currently

Beam tree

This method clips each polygon drawn to a beam tree which defines the viewable area. The beam tree originates as a description of the viewing frustum, and is in fact a special kind of BSP tree. When a new polygon is rendered, it is first passed through th

First DRAFT.

--

Last Update: 05/01/96 11:13:53

1.12 How do you accelerate ray tracing with a BSP Tree?

* How do you accelerate ray tracing with a BSP Tree?

Overview

Ray tracing with a BSP tree is very similar to hidden surface removal with a BSP tree. The algorithm is a simple forward tree walk, with a few additions that apply to ray casting. See Jim Arvo's voxel walking algorithm for ray tracing excerp

Implementation notes

Probably the biggest difference between ray tracing and other applications of BSP trees is that ray tracing does not require splitting of primitives to obtain correct results. This means that the hyperplanes can, and should, be chosen strictly for tree ba

Because ray tracing is a spatial classification problem, balancing is the key to performance. Most spatial partitioning schemes for accellerating ray tracing use a criteria called "occupancy", which refers to the number of primitives residing in each part

Balancing is discussed elsewhere in this document.

MORE TO COME

--

Last Update: 05/01/96 11:13:53

1.13 How do you perform boolean operations on polytopes with a BSP Tree?

* How do you perform boolean operations on polytopes with a BSP Tree?

Overview

There are two major classes of solid modeling methods with BSP trees. For both methods, it is useful to introduce the notion of an in/out test.

An in/out test is a different way of talking about the front/back test we have been using to classify points with respect to planes. The necessity for this shift in thought is evident when considering polytopes instead of just polygons. A point can not be

Incremental construction

Incremental construction of a BSP Tree is the process of inserting convex polytopes into the tree one by one. Each polytope has to be processed according to the operation desired.

It is useful to examine the construction process in two dimensions. Consider the following figure:

A B

+-------------+

| |

| |

| E | F

| +-----+-------+

| | | |

| | | |

| | | |

+-------+-----+ |

D | C |

| |

| |

+-------------+

H G

Two polygons, ABCD, and EFGH, are to be inserted into the tree. We wish to find the union of these two polygons. Start by inserting polygon ABCD into the tree, choosing the splitting hyperplanes to be coincident with the edges. The tree looks like this af

AB

-/ \+

/ \

/ *

BC

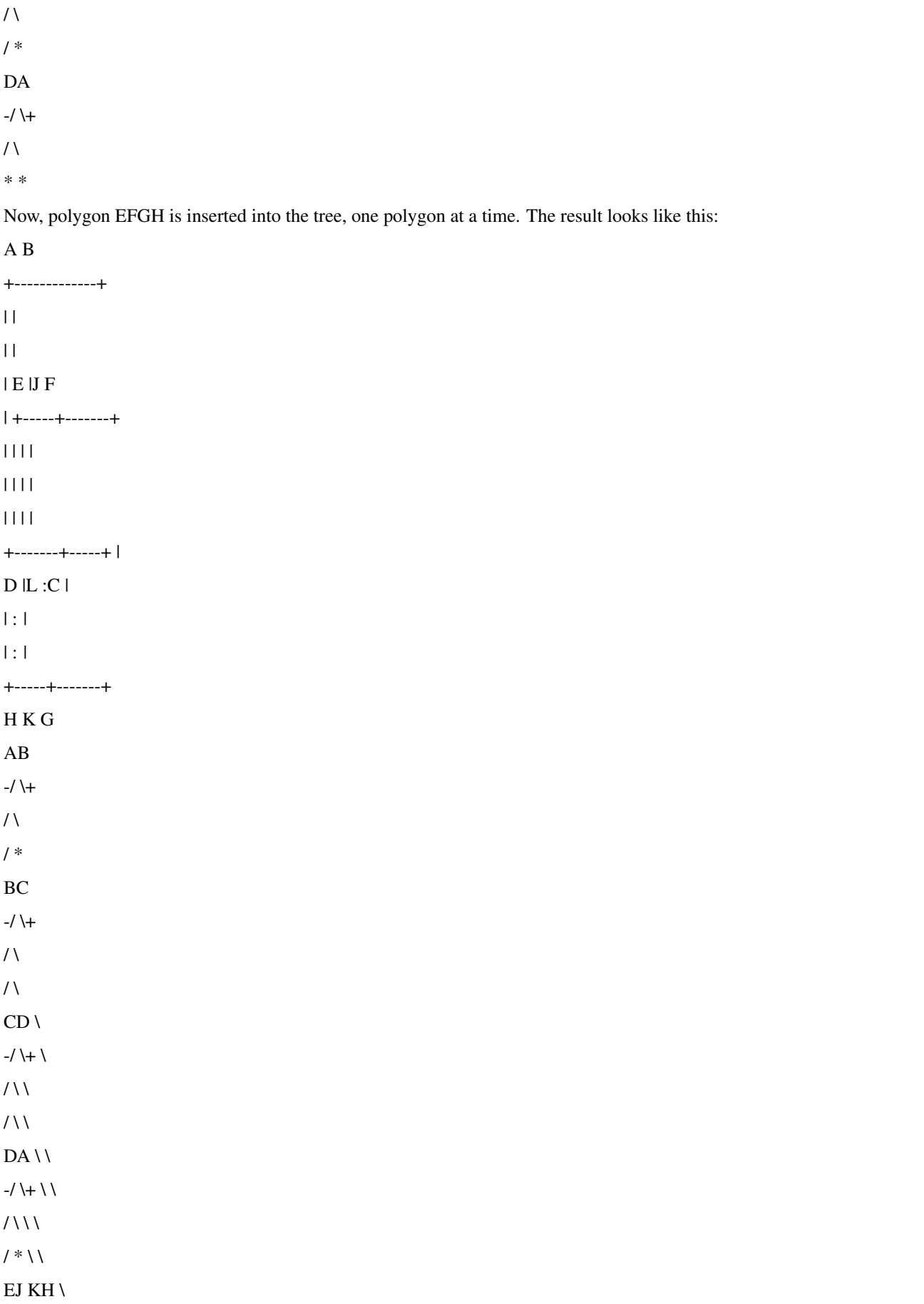
-/ \+

/ \

/ *

CD

-/ \+



Now, polygon EFGH is inserted into the tree, one polygon at a time. The result looks like this:

-/ \+ -/ \+ \

/ \ / \ \

/ * / * \

LE HL JF

-/ \+ -/ \+ -/ \+

/ \ / \ / \

* * * * FG *

-/ \+

/ \

/ *

GK

-/ \+

/ \

* *

Notice that when we insert EFGH, we split edges EF and HE along the edges of ABCD. this has the effect of dividing these segments into pieces which are inside ABCD, and outside ABCD. Segments EJ and LE will not be part of the boundary of the union. We cou

Our tree now looks like this:

A B

+--------------+

| |

| |

| |J F

| +-------+

| | |

| | |

| | |

+-------+-----+ |

D |L :C |

| : |

| : |

+-----+-------+

H K G

AB

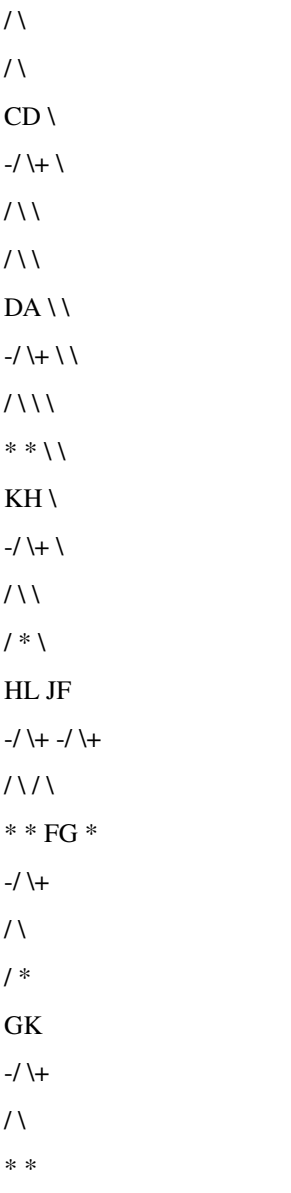
-/ \+

/ \

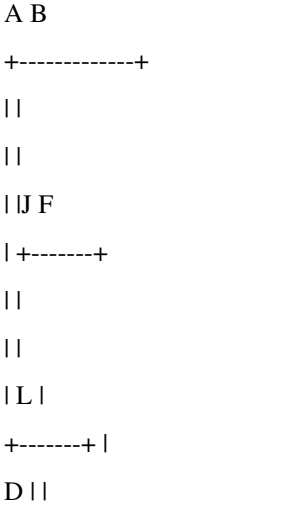
/ *

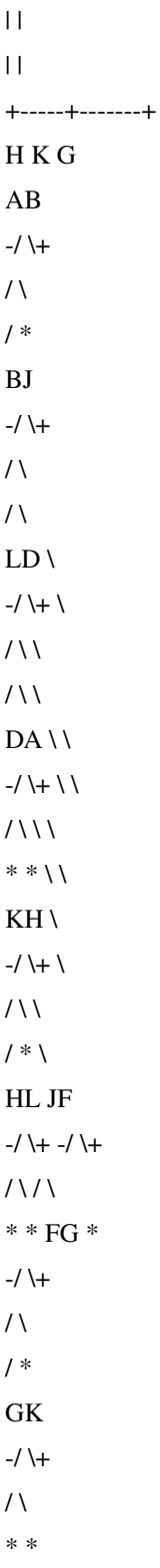
BC

-/ \+



Now, we would like some way to eliminate the segments JC and CL, so that we will be left with the boundary segments of the union. Examine the segment BC in the tree. What we would like to do is split BC with the hyperplane JF. Conveniently, we can do this





As you can see, the result is the union of the polygons ABCD and EFGH.

To perform other boolean operations, the process is similar. For intersection, you discard segments which land in exterior nodes instead of internal ones. The difference operation is special. It requires that you invert the polytope before insertion. For

Tree merging

--

Last Update: 05/01/96 11:13:53

1.14 How do you perform collision detection with a BSP Tree?

* How do you perform collision detection with a BSP Tree?

Overview

Detecting whether or not a point moving along a line intersects some object in space is essentially a ray tracing problem. Detecting whether or not two complex objects intersect is something of a tree merging problem.

Typically, motion is computed in a series of Euler steps. This just means that the motion is computed at discrete time intervals using some description of the speed of motion. For any given point P moving from point A with a velocity V, it's loca

Consider the case where T = 1, and we are computing the motion in one second steps. To find out if the point P has collided with any part of the scene, we will first compute the endpoints of the motion for this time step. P1 = A + V, and P2 = A + (2 * V)

Two approaches are possible. The first is commonly used in applications like games, where speed is critical, and accuracy is not. This approach is to recursively divide the motion segment in half, and check the midpoint for containment by some object. Typ

The second approach, which is more accurate, but also more time consuming, is to treat the motion segment as a ray, and intersect the ray with the BSP Tree. This also has the advantage that the motion resulting from the impact can be computed more accurat

--

Last Update: 05/01/96 11:13:53

1.15 How do you handle dynamic scenes with a BSP Tree?

* How do you handle dynamic scenes with a BSP Tree?

Overview

So far the discussion of BSP tree structures has been limited to handling objects that don't move. However, because the hidden surface removal algorithm is so simple and efficient, it would be nice if it could be used with dynamic scenes too. Faster anima

The BSP tree hidden surface removal algorithm can easily be extended to allow for dynamic objects. For each frame, start with a BSP tree containing all the static objects in the scene, and reinsert the dynamic objects. While this is straightforward to imp

If a dynamic object is separated from each static object by a plane, the dynamic object can be represented as a single point regardless of its complexity. This can dramatically reduce the computation per frame because only one node per dynamic object is i

Implementation notes

Inserting a point into the BSP tree is very cheap, because there is only one front/back test at each node. Points are never split, which explains the requirement of separation by a plane. The dynamic object will always be drawn completely in front of the

A dynamic object inserted into the tree as a point can become a child of either a static or dynamic node. If the parent is a static node, perform a front/back test and insert the new node appropriately. If it is a dynamic node, a different front/back test

An alternative when inserting a dynamic node is to construct a plane whose normal is the vector from the point to the eye. This plane is used in front/back tests just like the partition plane in a static node. The plane should be computed lazily and it is

Cleanup at the end of each frame is easy. A static node can never be a child of a dynamic node, since all dynamic nodes are inserted after the static tree is completed. This implies that all subtrees of dynamic nodes can be removed at the same time as the

Caveats

Recent discussion on "comp.graphics.algorithms" has demonstrated some weaknesses with this approach. In particular, an object modelled as a point may not be rendered in the correct order if the actual object spans a partitioning plane.

Advanced methods

Tree merging, "ghosts", real dynamic trees... MORE TO COME

--

Last Update: 05/01/96 11:13:53

1.16 How do you compute shadows with a BSP Tree?

* How do you compute shadows with a BSP Tree?

Overview

--

Last Update: 05/01/96 11:13:53

1.17 How do you extract connectivity information from BSP Trees?

* How do you extract connectivity information from BSP Trees?

Overview

--

Last Update: 05/01/96 11:13:53

1.18 How are BSP Trees useful for robot motion planning?

* How are BSP Trees useful for robot motion planning?

Overview

--

Last Update: 05/01/96 11:13:53

1.19 How are BSP Trees used in DOOM?

* How are BSP Trees used in DOOM?

Overview

Before you can understand how DOOM uses a BSP tree to accelerate its renderingprocess, you have to understand how the world is represented in DOOM. Whensomeone creates a DOOM level in a level editor they draw linedefs in a 2dspace. Yes, that's right, DOOM

When the level is saved by the editor some new information is createdincluding the BSP tree for that level. Before the BSP tree can be created, allthe sectors have to be split into convex polygons known as sub-sectors. If youhad a sector that was a square

Given a point on the 2d map, the renderer (which isn't discussed here) wantsa list of all the segs that are visible from that viewpoint in closest firstorder. Because of the restrictions placed on the DOOM world, the renderer caneasily tell when the scree

Each node in the BSP tree defines a partition line (this does not have be alinedef in the world but usually is) which is the equivalent to the partitionplane of a 3d BSP tree. It then has left and right pointers which are eitheranother node for further su

During the display update process the BSP tree is searched starting from thenode containing the sub-sector that the player is currently in. The searchmoves outwards through the tree (searching the other half of the current nodebefore moving onto the other

In case you're thinking that it is inefficient to dump a whole sub-sectorsworth of segs into the renderer at once, the segs in a sub-sector can beback-face culled very quickly. DOOM stores the angle of linedefs (of whichsegs are part). When the angle of t

--

Last Update: 05/01/96 11:13:53

1.20 How can you make a BSP Tree more robust?

* How can you make a BSP Tree more robust?

Overview

--

Last Update: 05/01/96 11:13:53

1.21 How efficient is a BSP Tree?

* How efficient is a BSP Tree?

Space complexity

For hidden surface removal and ray tracing accelleration, the upper bound is O(n ^ 2) for n polygons. The expected case is O(n) for most models. MORE LATER

Time complexity

For hidden surface removal and ray tracing accelleration, the upper bound is O(n ^ 2) for n polygons. The expected case is O(n) for most models. MORE LATER

--

Last Update: 05/01/96 11:13:53

1.22 How can you make a BSP Tree more efficient?

* How can you make a BSP Tree more efficient?

Bounding volumes

Bounding spheres are simple to implement, take only a single plane comparison, using the center of the sphere.

Optimal trees

Construction of an optimal tree is an NP-complete problem. The problem is one of splitting versus tree balancing. These are mutually exclusive requirements. You should choose your strategy for building a good tree based on how you intend to use the tree.

Minimizing splitting

An obvious problem with BSP trees is that polygons get split during the construction phase, which results in a larger number of polygons. Larger numbers of polygons translate into larger storage requirements and longer tree traversal times. This is undesi

Bear in mind that minimization of splitting requires pre-existing knowledge about all of the polygons that will be inserted into the tree. This knowledge may not exist for interactive uses such as solid modelling.

Tree balancing

Tree balancing is important for uses which perform spatial classification of points, lines, and surfaces. This includes ray tracing and solid modelling. Tree balancing is important for these applications because the time complexity for classification is b

For the hidden surface problem, balancing doesn't significantly affect runtime. This is because the expected time complexity for tree traversal is linear on the number of polygons in the tree, rather than the depth of the tree.

Balancing vs. splitting

If balancing is an important concern for your application, it will be necessary to trade off some balance for reduced splitting. If you are choosing your hyperplanes from the polygon candidates, then one way to optimize these two factors is to randomly se

Reference Counting

Other Optimizations

--

Last Update: 05/01/96 11:13:53

1.23 How can you avoid recursion?

* How can you avoid recursion?

Overview

A BSP tree resembles a standard binary tree structure in many ways. Using the tree for a painter's algorithm or for ray tracing is a "depth first" traversal of the tree structure. Depth first traversal is traditionally presented as a recursive operation,

Implementation Notes

Depth first traversal is a means of enumerating all of the leaves of a tree in sorted order. This is accomplished by visiting each child of each node in a recursive manner as follows:

```
void Enumerate (BSP_tree *tree)
{
if (tree->front)
Enumerate (tree->front);
if (tree->back)
Enumerate (tree->back);
```

To eliminate the recursion, you have to explicitly model the recursion using a stack. Using a stack of pointers to BSP_tree, you can perform the enumeration like this:

```
void Enumerate (BSP_tree *tree)
{
Stack stack;
while (tree)
{
if (tree->back)
stack.Push (tree->back);
if (tree->front)
stack.Push (tree->front);
tree = stack.Pop ();
}
```

On some processors, using a stack will be faster than the recursive method. This is especially true if the recursive routine has a lot of local variables. However, the recursive approach is usually easier to understand and debug, as well as requiring less

--

Last Update: 05/01/96 11:13:53

1.24 What is the history of BSP Trees?

* What is the history of BSP Trees?

Overview

Neophyte: How did the BSP-Tree come to be?

Sage: Long ago in a small village in Nepal, a minor godling gave a specialnut to the priests at an out of the way temple. With the nut, was aprophecy: When a group of three gurus, two with red hair, and theother who was not what he seemed, came to the te

N: no! No! NO! The TRUE story.

S: OK.

Long ago (by computer industry standards) in a rapidly growing sunbeltcity in Texas, a serendipitous convergence of unusual talents andpersonalities occurred. A brief burst of graphics wondermentsappeared, and the convergence diverged under its own explo

N: ...No! The facts!

S: Huh? Oh you want FACTS. Boring stuff?

Henry Fuchs started teaching at an essentially brand new campus, theUniversity of Texas at Dallas, in January 1975. He returned to Utahto complete his PhD the following summer. He returned to Dallas andtaught for the 1975-76, 1976-77 and 1977-78 academi

Zvi Kedem joined this faculty in the fall of 1975. He was (and still isI suppose) a "theory person," but a special theory person. He is goodat applying theory to practical problems.

Bruce Naylor had a bachelors degree from the U of Texas (Austin - "thereal one"), in philosophy if I recall correctly. He had talked hisway into a job at Texas Instruments in Dallas. (Something aboutphilosophy makes you good in logic, which is really th

Graphics, of course, is dazzling and wonderful. Henry was (is) fullof energy and enthusiasm. It was natural that he attracted lots ofthe grad students. Kedem was a perfect complement, providing not onlythe formal rigor and proofs, but also the impetus

The result was two SIGGRAPH papers and Naylor's PhD dissertation, and thelaunch of BSP-trees into the world. The two papers are

Fuchs, Kedem and Naylor, "Predeterming Visibility Priority in 3-D Scenes"SIGGRAPH '79, pp 175-181. (subtitled "preliminary report")

and

Fuchs, Kedem and Naylor, "On Visible Surface Generation by A Priori TreeStructures" SIGGRAPH '80, pp124-133.

The first paper isn't really BSP-trees, but rather the preliminary work whichled to BSP-trees as the solution. The second paper is the "classic" startingpoint referenced in texts, etc.

Both reference Schumacker, Brand, Gilliland and Sharp, "Study forApplying Computer-Generated Images to Visual Simulation"AFHRL-TR-69-14, US AF Human Resources Lab, 1969

and the description of this algorithm more easily found in

Sutherland, Sproull and Schumacker, "A Characterization of Ten HiddenSurface Algorithms", ACM Computing Surveys, v 6, no 1, pp 1-55.

Naylor finished in 1981 (?) and went to Georgia Tech, and later toBell Labs. He has continued to work on related and similar ideas witha variety of students and collaborators. Others have also taken theideas in new directions.

--

Last Update: 05/01/96 11:13:53

1.25 Where can you find sample code and related online resources?

* Where can you find sample code and related online resources?

BSP tree FAQ companion code

The companion source code to this document is available via FTP at:

* file://ftp.qualia.com/pub/bspfaq/

or, you can also request that the source be mailed to you by sending e-mail to "bspfaq@qualia.com" with a subject line of "SEND BSP TREE SOURCE". This will return to you a UU encoded copy of the sample C++ source code.

Also in this directory is the BSP tree demo application for MacOS and Win95. This demo shows how to use BSP trees for basic Boolean modelling and hidden surface removal, and includes an implementation of Ken Shoemake's ArcBall interface. The MacOS package

Information about the ArcBall controller can be found in Graphics Gems IV (see below). The original paper and demo application for 68k based Apple Macintosh computers is available via WWW or FTP at:

* http://www.cis.upenn.edu/ftpbin/ftphtml.pl?/pub/graphics/arcball

* ftp://ftp.cis.upenn.edu/pub/graphics/arcball/

Graphics Gems

The Graphics Gems archive is available via FTP at:

* file://ftp.princeton.edu/pub/Graphics/GraphicsGems/

* ftp://ftp-graphics.stanford.edu/pub/Graphics/GraphicsGems/

It is an invaluable resource for all things graphical. In particular, there are some BSP tree references worth looking over.

Peter Shirley and Kelvin Sung have C sample code for ray tracing with BSP trees in "Graphics Gems III"}

Norman Chin has provided a wonderful resource for BSP trees in "Graphics Gems V". He provides C sample code for a wide variety of uses.

Other BSP tree resources

The Ray Tracing News is available at:

* ftp://ftp-graphics.stanford.edu/pub/Graphics/RTNews/html/index.html

It has many good articles which involve the use of BSP trees or compares them to other data structures for ray tracing efficiency. It's definitely worth a look.

The DejaNews research service is a search engine for newsgroup postings. Conduct a search on "BSP tree" to read over recent, as well as old, discussions about BSP trees. DejaNews is available at:

* http://www.dejanews.com/

Pat Fleckenstein and Rob Reay have put together a FAQ on 3D graphics, which includes a blurb on BSP trees, and an FTP site with some sample code:

* http://www.csh.rit.edu/~pat/misc/3dFaq.html

* file://ftp.csh.rit.edu/pub/3dfaq/

"Dr. Dobbs Journal" has an article in their July '95 issue about BSP trees, By Nathan Dwyer. It describes the construction of BSP trees for visible surface processing, how to split polygons with planes, and how to dump the tree to a file. There is C++ sou

* http://www.ddj.com/ddj/1995/1995.07/dwyer.htm

Michael Abrash's columns in the DDJ Sourcebooks are an excellent introduction to the concept of BSP trees, and the practical details of implementing them for games like Doom and Quake. The source code for these articles is distributed over several sites.

* ftp://ftp.mv.com/pub/ddj/1995/1995.cpp/asc.zip

* ftp://ftp.idsoftware.com/mikeab/

Ekkehard Beier has made available a generic 3D graphics kernel intended to assist development of graphics application interfaces. One of the classes in the library is a BSP tree, and full source is provided. The focus seems to be on ray tracing, with the

* ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/GENERIC/

Eddie Edwards wrote a commonly referenced text which describes 2D BSP trees in some detail for use in games like DOOM. It includes a bit of sample code, too.

* file://x2ftp.oulu.fi/pub/msdos/programming/theory/bsp_tree.zip

Mel Slater has made available his C source code for computing shadow volumes based on BSP trees. There is also a paper which compares three shadow volume algorithms. These are available via FTP at:

* ftp://ftp.dcs.qmw.ac.uk/people/mel/BSP/

The SPHIGS distribution that accompanies the famous Foley, et al. Computer Graphics textbook contains a BSP tree implementation and can be obtained at:

* file://ftp.cs.brown.edu/pub/sphigs.tar.Z

John Holmes, Jr. has a paper about BSP trees and boundary representations online. This paper touches on many aspects of BSP trees in a scientific manner, and is available at:

* http://www.cis.ufl.edu/~jch/is-final.ps

A.T. Campbell III has placed his doctoral dissertation online. This document describes the use of BSP trees for surface meshing in a radiosity algorithm. He has also made available source code in both C and C++ for many BSP tree tasks, including an implem

* http://www.arlut.utexas.edu/~atc/

Andrea Marchini and Stefano Tommesani have made available a BSP tree compiler and viewing program as part of the "Purple Haze" project. There is some nice accompanying documentation of the process of BSP tree construction, as well. The web page is at:

* http://www.ce.unipr.it/~tommesa/

Paton Lewis has built an excellent demonstration of BSP trees using Java. If you have had problems visualizing the tree contruction process, you should take a look at this tool. The user draws lines in an interactive construction space, and three other vi

* http://www.cs.brown.edu/people/pjl/bsptreedemo/bsptreedemo.html

General resources for computer graphics programming

If you are interested in game programming, check out the rec.games.programmer.faq: http://www.io.com/~paulhart/game/FAQ/rgp_FAQ You can also look at Paul's game programming page, http://www.io.com/~paulhart/game/

The computational geometry web page contains lots of information and pointers to related computational geometry tools and algorithms: http://www.cs.berkeley.edu/~jeffe/compgeom.html. Another computational geometry page to look at is @

--

Last Update: 08/21/96 23:05:45

1.26 References

* References

A partial listing of textual info on BSP trees.

* Abrash, M., BSP Trees, Dr. Dobbs Sourcebook, 20(14), 49-52, may/jun 1995.

* Dadoun, N., Kirkpatrick, D., and Walsh, J., The Geometry of Beam Tracing, Proceedings of the ACM Symposium on Computational Geometry, 55--61, jun 1985.

* Chin, N., and Feiner, S., Near Real-Time Shadow Generation Using BSP Trees, Computer Graphics (SIGGRAPH '89 Proceedings), 23(3), 99--106, jul 1989.

* Chin, N., and Feiner, S., Fast object-precision shadow generation for area light sources using BSP trees, Computer Graphics (1992 Symposium on Interactive 3D Graphics), 25(2), 21--30, mar 1992.

* Chrysanthou, Y., and Slater, M., Computing dynamic changes to BSP trees, Computer Graphics Forum (EUROGRAPHICS '92 Proceedings), 11(3), 321--332, sep 1992.

* Naylor, B., Amanatides, J., and Thibault, W., Merging BSP Trees Yields Polyhedral Set Operations, Computer Graphics (SIGGRAPH '90 Proceedings), 24(4), 115--124, aug 1990.

* Naylor, B., Interactive solid geometry via partitioning trees, Proceedings of Graphics Interface '92, 11--18, may 1992.

* Naylor, B., Partitioning tree image representation and generation from 3D geometric models, Proceedings of Graphics Interface '92, 201--212, may 1992.

* Naylor, B., {SCULPT} An Interactive Solid Modeling Tool, Proceedings of Graphics Interface '90, 138--148, may 1990.

* Gordon, D., and Chen, S., Front-to-back display of BSP trees, IEEE Computer Graphics and Applications, 11(5), 79--85, sep 1991.

* Ihm, I., and Naylor, B., Piecewise linear approximations of digitized space curves with applications, Scientific Visualization of Physical Phenomena (Proceedings of CG International '91), 545--569, 1991.

* Vanecek, G., Brep-index: a multidimensional space partitioning tree, Internat. J. Comput. Geom. Appl., 1(3), 243--261, 1991.

* Arvo, J., Linear Time Voxel Walking for Octrees , Ray Tracing News, feb 1988.

* Jansen, F., Data Structures for Ray Tracing, Data Structures for Raster Graphics, 57--73, 1986.

* MacDonald, J., and Booth, K., Heuristics for Ray Tracing Using Space Subdivision, Proceedings of Graphics Interface '89, 152--63, jun 1989.

* Naylor, B., and Thibault, W., Application of BSP Trees to Ray Tracing and CSG Evaluation, Tech. Rep. GIT-ICS 86/03, feb 1986.

* Sung, K., and Shirley, P., Ray Tracing with the BSP Tree, Graphics Gems III, 271--274, 1992.

* Fuchs, H., Kedem, Z., and Naylor, B., On Visible Surface Generation by A Priori Tree Structures, Conf. Proc. of SIGGRAPH '80, 14(3), 124--133, jul 1980.

* Paterson, M., and Yao, F., Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling, Discrete and Computational Geometry, 5(5), 485--503, 1990.

--

Last Update: 05/01/96 11:13:53

1.27 Linear-Time Voxel Walking for Octrees

* Linear-Time Voxel Walking for Octrees

Jim Arvo - 1988

Here is a new way to attack the problem of "voxel walking" in octrees (at least I think it's new). By voxel walking I mean identifying the successive voxels along the path of a ray. This is more for theoretical interest than anything else, though the al

For this discussion assume that we have recursively subdivided a cubical volume of space into a collection of equal-sized voxels using a BSP tree -- i.e. each level imposes a single axis-orthogonal partitioning plane. The algorithm is much easier to desc

Assuming that the leaf nodes form an NxNxN array of voxels, any given ray will pierce at most O(N) voxels. The actual bound is something like 3N, but the point is that it's linear in N. Now, suppose that we use a "re-traversal" technique to move from

In this note I propose a new "voxel walking" algorithm for octrees (call it the "partitioning" algorithm for now) which has a worst case time complexity of O(N) under the conditions outlined above. In the best case of finding a hit "right away" (after

BEST CASE: O(1) voxels WORST CASE: O(N) voxels

searched before a hit. searched before a hit.

+---+

| |

Re-traveral | O(log N) O(N Log N) |

| |

Partitioning | O(log N) O(N) |

| |

+---+

The new algorithm proceeds by recursively partitioning the ray into little line segments which intersect the leaf voxels. The top-down nature of the recursive search ensures that partition nodes are only considered ONCE PER RAY. In addition, the voxels

Below is a pseudo code description of the "partitioning" algorithm. It is the routine for intersecting a ray with an environment which has been subdivided using a BSP tree. Little things like checking to make sure the intersection is within the appropri

Node - A BSP tree node which comes in two flavors -- a partition node or a leaf node. A partition node defines a plane and points to two child nodes which further partition the "positive" and "negative" half-spaces. A leaf node points to a list

P - The ray origin. Actually, think of this as an endpoint of a 3D line segment, since we are constraining the "ray" to be of finite length.

D - A unit vector indicating the ray direction.

len - The length of the "ray" -- or, more appropriately, the line segment. This is measured from the origin, P, along the direction vector, D.

The function "Intersect" is initially passed the root node of the BSP tree, the origin and direction of the ray, and a length, "len", indicating the maximum distance to intersections which are to be considered. This starts out being the distance to the f

FUNCTION Intersect(Node, P, D, len) RETURNING "results of intersection"

IF Node is NIL THEN RETURN("no intersection")

IF Node is a leaf THEN BEGIN

intersect ray (P,D) with objects in the candidate list

RETURN("the closest resulting intersection")

END IF

dist := signed distance along ray (P,D) to plane defined by Node

near := child of Node in half-space which contains P

IF 0 < dist < len THEN BEGIN /* the interval intersects the plane */

hit_data := Intersect(near, P, D, dist)

IF hit_data <> "no intersection" THEN RETURN(hit_data)

Q := P + dist * D /* 3D coords of point of intersection */

far := child of Node in half-space which does NOT contain P

RETURN(Intersect(far, Q, D, len - dist))

END IF

ELSE RETURN(Intersect(near, P, D, len))

END

As the BSP tree is traversed, the line segments are chopped up by the partitioning nodes. The "shrinking" of the line segments is critical to ensure that only relevent branches of the tree will be traversed.

The actual encodings of the intersection data, the partitioning planes, and the nodes of the tree are all irrelevant to this discussion. These are "constant time" details. Granted, they become exceedingly important when considering whether the algorithm

A naive (and incorrect) proof of the claim that the time complexity of this algorithm is O(N) would go something like this:

The voxel walking that we perform on behalf of a single ray is really

just a search of a binary tree with voxels at the leaves. Since each

node is only processed once, and since a binary tree with k leaves has

k - 1 internal nodes, the total number of nodes which are processed in

the entire operation must be of the same order as the number of leaves.

We know that there are O(N) leaves. Therefore, the time complexity

is O(N).

But wait! The tree that we search is not truly binary since many of the internal nodes have one NIL branch. This happens when we discover that the entire current line segment is on one side of a partitioning plane and we prune off the branch on the othe

Suppose we were to pick N random voxels from the N**3 possible choices, then walk up the BSP tree marking all the nodes in the tree which eventually lead to these N leaves. Let's call this the subtree "generated" by the original N voxels. Clearly this i

This is in fact the case. To prove it I found that all I needed to assume about the voxels was connectedness -- provided I made some assumptions about the "niceness" of the BSP tree. To give a careful proof of this is very tedious, so I'll just outline

* Two voxels are "connected" (actually "26-connected") if they meet at a face, an edge, or a corner. We will say that a collection of voxels is connected if there is a path of connected voxels between any two of them.

* "regular" BSP tree is one in which each axis-orthogonal partition divides the parent volume in half, and the partitions cycle: X, Y, Z, X, Y, Z, etc. (Actually, we can weaken both of these requirements considerably and still make the proof work. If we'

Here is a sequence of little theorems which leads to the main result:

* A ray pierces O(N) voxels.

* The voxels pierced by a ray form a connected set.

* Given a collection of voxels defined by a "regular" BSP

tree, any connected subset of K voxels generates a unique

subtree with O(K) nodes.

* The "partitioning" algorithm visits exactly the nodes of

the subtree generated by the voxels pierced by a ray.

Furthermore, each of these nodes is visited exaclty once

per ray.

* The "partitioning" algorithm has a worst case complexity

of O(N) for walking the voxels pierced by a ray.

Theorems 1 and 2 are trivial. With the exception of the "uniqueness" part, theorem 3 is a little tricky to prove. I found that if I completely removed either of the "regularity" properties of the BSP tree (as opposed to just weakening them), I could con

Some Practical Matters

Since log N is typically going to be very small -- bounded by 10, say -- this whole discussion may be purely academic. However, just for the heck of it, I'll mention some things which could make this a (maybe) competative algorithm for real-life situatio

First of all, it would probably be advisable to avoid recursive procedure calls in the "inner loop" of a voxel walker. This means maintaining an explicit stack. At the very least one should "longjump" out of the recursion once an intersection is found.

The calculation of "dist" is very simple for axis-orthogonal planes, consisting of a subtract and a multiply (assuming that the reciprocals of the direction components are computed once up front, before the recursion begins).

A nice thing which falls out for free is that arbitrary partitioning planes can be used if desired. The only penalty is a more costly distance calculation. The rest of the algorithm works without modification. There may be some situations in which this

Sigh. This turned out to be much longer than I had planned...

Here is a slightly improved version of the algorithm in my previous mail. It turns out that you never need to explicitly compute the points of intersection with the partitioning planes. This makes it a little more attractive.

-- Jim

FUNCTION BSP_Intersect(Ray, Node, min, max) RETURNING "intersection results"

BEGIN

IF Node is NIL THEN RETURN("no intersection")

IF Node is a leaf THEN BEGIN /* Do the real intersection checking */

intersect Ray with each object in the candidate

list discarding those farther away than "max."

RETURN("the closest resulting intersection")

END IF

dist := signed distance along Ray to plane defined by Node

near := child of Node for half-space containing the origin of Ray

far := the "other" child of Node -- i.e. not equal to near.

IF dist > max OR dist < 0 THEN /* Whole interval is on near side. */

RETURN(BSP_Intersect(Ray, near, min, max))

ELSE IF dist < min THEN /* Whole interval is on far side. */

RETURN(BSP_Intersect(Ray, far , min, max))

ELSE BEGIN /* the interval intersects the plane */

hit_data := BSP_Intersect(Ray, near, min, dist) /* Test near side */

IF hit_data indicates that there was a hit THEN RETURN(hit_data)

RETURN(BSP_Intersect(Ray, far, dist, max)) /* Test far side. */

END IF

END

It should be noted that the recursive traversal algorithm introduced independently by Frederik Jansen in 1986 is a different implementation of the same idea.