One Approach to Real Time Texture Mapping

This first section just talks a little about myself. Mostly it describes the events leading up to my post on the comp.graphics.* newsgroups. If you couldn't care less or it bores you, skip it. The nitty-gritty stuff starts at the next heading.

About three years ago I rented a game called Ultima Underworld, written by ORIGIN Systems, Inc. It was the first virtual-reality type game I had ever seen and I thought it was so good I bought it. I had made a few feeble attempts a few years earlier at programs which would simulate a three-dimensional perspective, couldn't figure out the math, and gave it up. So I thought Underworld was pretty much magic.

Well, shortly after buying Underworld, I walked into the PC computer lab on campus and saw a guy playing a 3D type game and instinctively said, "Oh, that's Underworld." He said, "Nuh uh, it's called Castle Wolfenstein. It's shareware. If you want I'll make you a copy."

That night, I played all the way through the first episode of Wolfenstein in one sitting. I thought it was the best video game I had ever played. I couldn't believe how realistic it looked, how fast it was, how it fit on one diskette, and how the guys who wrote it were giving it away for free (at least the first episode). I pretty much forgot Underworld over the next few weeks as my friends and I etched a groove on my harddrive where Wolfenstein was installed.

Eventually Wolfenstein got old (I never got Spear of Destiny), and I got way too busy with school. I forgot about id and Wolfenstein almost altogether. Then for my last semester, I took the undergrad graphics course. Just after the semester started, a friend told me that id had a new game out. It was called DOOM, and it was a lot better than Wolfenstein. I said no way, I would have heard of it. He told me where to get it FTP. I got it and, well, let's just say I'm lucky I graduated. The graphics course was excellent, though, and I decided that I would really enjoy doing graphics programming. Especially video games.

Since I was already in Austin, I decided to apply to ORIGIN. I got an interview, told them what I knew, said I wanted to do 3D type programming, and got scheduled to see the 3D head-honcho guy. Well, the 3D head-honcho guy was 45 minutes late when I decided that he was too late for me. I didn't want to be working for a company like that so I took another offer which offered more money and less satisfaction. The 3D head-honcho guy called me four times over the next week trying to reschedule, but I never returned his call. So anyway, I ended up where I am now: working for a telecommunications company in Dallas wishing I was still in Austin. This was sometime in June, 1994.

Without very many acquaintances in this rather odd city, I have had a lot of time to experiment with the knowledge I got from my computer graphics course. Before long I decided to implement a DOOM-type graphics engine. As I started reading the comp.graphics.* news groups, I realized that a few hundred others (probably a lot more) had the same idea in mind. It wasn't until DOOM released (unofficially?) their WAD specs and I started seeing hundreds of new WADS being put on the net that I realized how popular DOOM had become. But over the next several months, discussion of these topics decreased steadily and it was as if all the fervor had never existed (I don't have access to the DOOM newsgroups). That brings us to the present. I have a working prototype which seems fast and it is not 100% optimized (eg. for pipelining, since I know nothing about it).

What follows is a description of the technique I am using to determine the horizontal index into a texture when mapping it onto a vertical (for lack of a better word) 3D surface when

projected onto a 2D projection plane. I am assuming you already have a basic understaning of 2D projections. Also I don't get into hidden surface removal in this document.

I am not a seasoned graphics expert so forgive me if I don't use the appropriate terminology in certain places. Also, I am not suggesting that I have discovered a revolutionary method either. It is undoubtedly thoroughly documented in some easily accessible text at your neighborhood bookstore. But for those of you who can't afford books, I hope it will help you with your own work on texture mapping. I know I would have liked it if something like this had been available when I first started out.

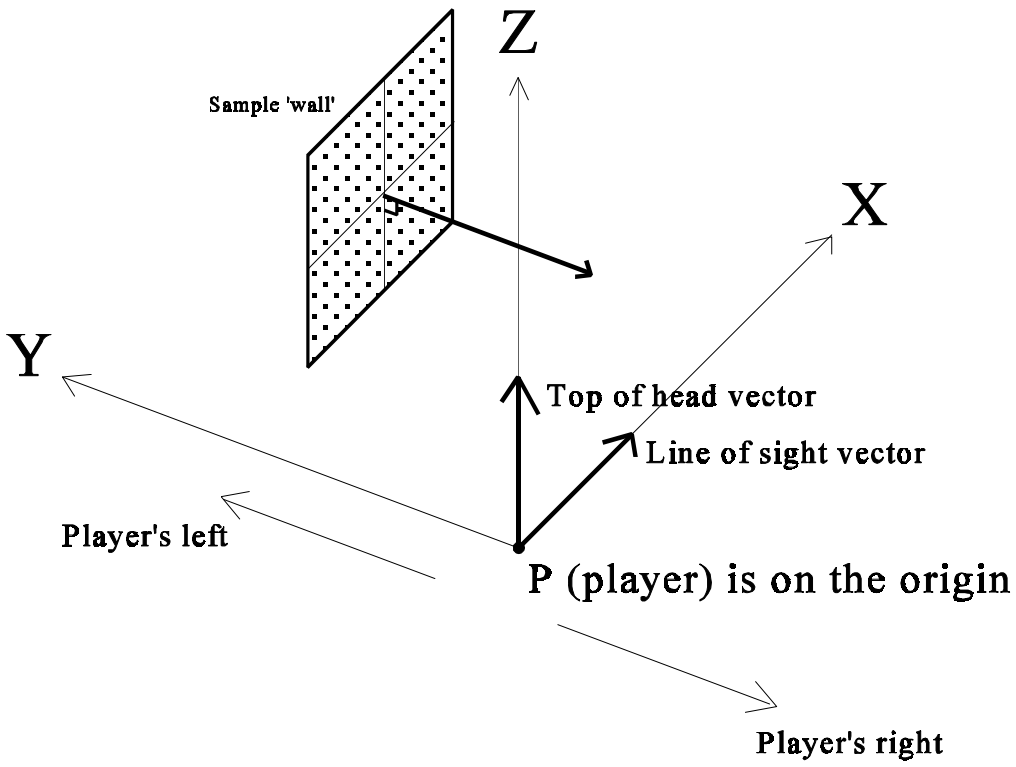
The Not So Nitty Gritty Stuff

In this section, I am going to discuss the method I finally settled on to map the textures. The method is really so simple that it might annoy you. You may have already thought of it. It may be common knowledge. It might even be one of the slower methods you can use. I really don't know for sure. All I know is that I had to figure it out for myself and it is plenty fast for me. If it, for some reason, bothers you immensely, please, let me know about it. I love to hear good criticism, I hear it so seldomly (wink).

Bill's Terminology:

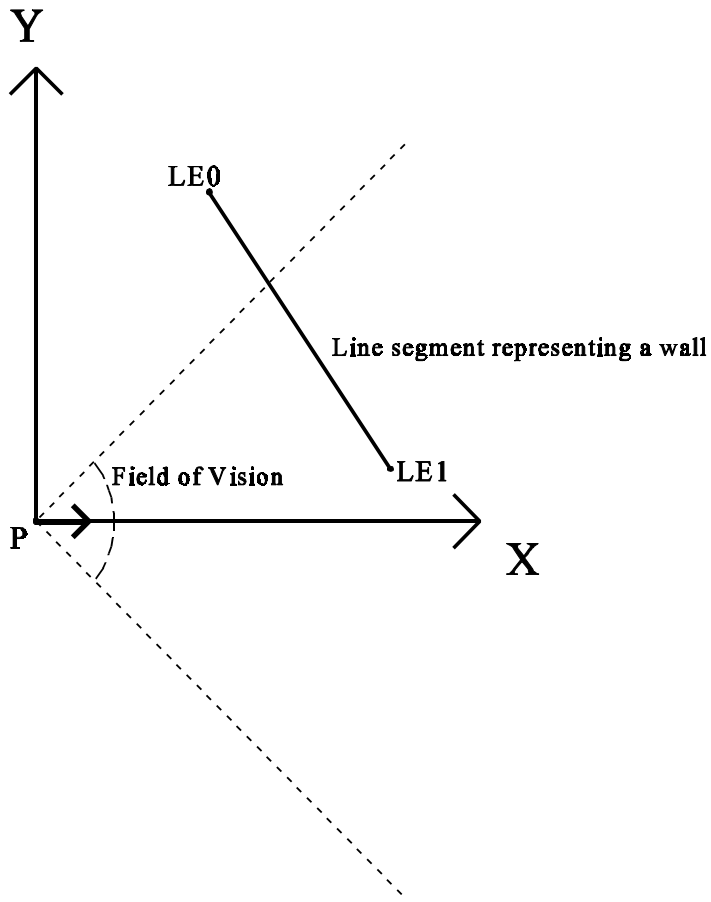
First off, it is important that there is a common understanding between you, the reader, and myself, the writer, when referring to our 3D world.

When I say 'wall' I am referring to a rectangular polygon defined in 3D world coordinates with a perpendicular vector that is parallel with the X/Y plane. When I say 'player', I am really talking about the 'eye' or the point of perspective. Also, I think I stray a bit from the standard model for defining a 3D coordinate system (or maybe I don't). But you will see why as we go on. In DOOM, as in my engine, everything can really be thought of in a 2D sense. For this reason, I chose to make the axis plane which is parallel to the ceiling and floor the X/Y plane. So the 'map' of your 3D world then occupies the standard Cartesian plane. When I discuss rendering a scene, I am assuming that the player is always on the origin facing the positive X axis. For you picky people (like me), I mean the player's head, not his feet (actually his cyclops eyeball). Here is a graphic which attempts to depict this important concept:



So before using any data for rendering perspective purposes it needs to be translated so P (player) is at the origin. Then rotated so the Line Of Sight vector is collinear with the positive X axis. In my data scheme, I use two points to describe a line in the X/Y plane (novel). Then by associating a Z bottom and a Z top value with it, this line can then represent a 'wall' or polygon in 3-space. That is how all of the walls in my 3D environment are defined.

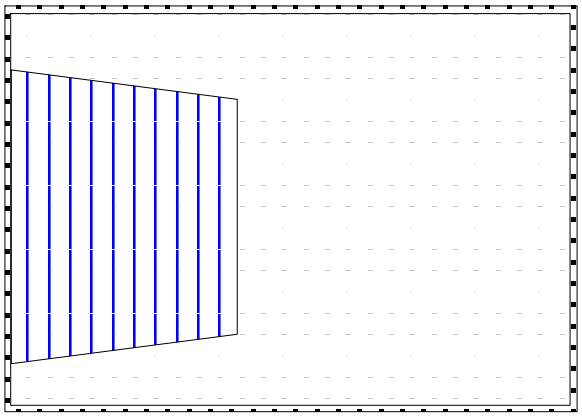
OK, let's assume all the data has been translated and rotated so the player is at the origin looking down the X axis. (If you don't know how to do this, then reading this document is probably a little premature.) Now imagine you are above the player looking down.



In the above diagram, the positive Z axis is coming straight out of the page. The two dotted lines connected with the dotted arc represent the player's left-to right field of view. They are at right angles to each other. The upper dotted line is collinear with the line defined by X = Y. The lower is collinear with the line defined by X = -Y. LE0 and LE1 are the endpoints of the pictured line segment. They have only X and Y coordinates, no Z. Instead an upper and a lower Z coordinate are given to the line as a whole to make it a 3D rectangle. Once again, this is what I am calling a wall in 3-space. It is this wall to which we want to apply a texture.

I hope you are still with me so far. If you aren't, I suggest playing DOOM, getting a copy of dmspec13.zip and trying out a doom WAD editor such as DEU. All of these things are available through FTP at infant2.sphs.indiana.edu. This should help familiarize you with the concepts being discussed thus far.

OK, with that out of the way, let's get down to the fun part. To transform the four points of our polygon into perspective, we divide the Y's and the Z's by their corresponding X's right? Right. Now we have projected the four corners of the wall onto our two-dimensional screen. The result would look something like this:



Notice that the left part of the wall is out of the player's field of view and consequently off the left edge of the screen. It would have needed clipping prior to rendering. Right, now let's assume the line defining this wall has a length of 128 units. Let's also assume that we defined the line's upper and lower Z coordinates to be 64 and -64 respectively. That makes the wall a 128x128 vertical (there's that word again), flat surface in 3-space. OK, let's also assume that the texture we want to 'paste' on this wall is, that's right you guessed it, 128 rows by 128 columns.

As far as I have been able to figure out, there are three basic techniques you can use to draw the texture on the wall. I will briefly discuss two of them and then give a more detailed description of the one I use.

Method 1.

If we were facing the wall head on (not at an angle like we are) then the projection of the wall would look square on the 2D screen. Mapping the texture would be a synch. Just calculate the scale between the texture's dimensions and the dimensions of the projected polygon and interpolate with a DDA algo. It's linear, it's quick, and the result looks beautiful. It only has one, albeit major, drawback; it only works when your Line of Sight vector is orthogonal to the surface to be mapped.

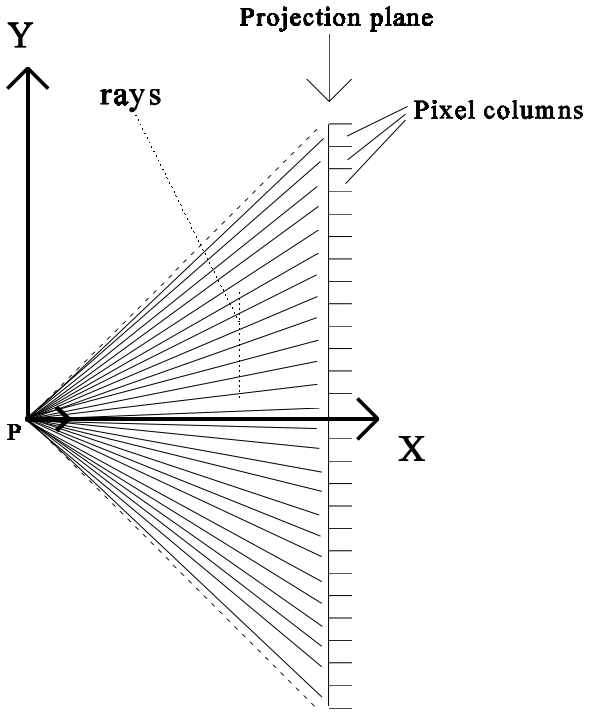
Actually, you can still use this method if you aren't facing the surface head on. You could use the same horizontal interpolation and interpolate each vertical pixel column individually since they would have different lengths (illustrated in the figure above). The problem is that, due to the workings of the human eye and the nature of light rays, the texture appears more and more distorted as the angle between the Line of Sight vector and the vector perpendicular to the wall

decreases (it's 180 when you are facing head on).

Believe me, without giving it much thought, I tried this method first. It didn't take me long to realize that it does not offer very realistic results. I realized, however, that the problem was only in the horizontal indexing. Because the horizontal interpolation is linear in this method, the middle of the texture remains in the middle of the projected polygon when it should, in fact, approach the side of the polygon representing the edge of the wall furthest from the player as the aforementioned angle decreases. But the vertical pixel columns still look fine. After some thinking, I realized that this phenomena is due to the fact that one column of the projected polygon represents a vertical line segment in 3-space which has an obvious property; every point on that line has the same X value (and Y value for that matter). Since we are using linear projection in the first place (dividing the Z by X), linear interpolation along a vertical column of pixels is justifiable. In fact, it's correct. So I just had to work on the horizontal indexing. Hmm... After reading a book on ray-tracing, I realized what the problem was, and reorganized my code for method number two.

Method 2.

This method involves what is commonly referred to as ray-casting. Much like a ray-tracer, you use parametric equations to determine the intersection of a ray and a plane. First, let's define what I mean by ray. For the purposes of this discussion, rays are defined as 2D directional vectors whose bases are at P (player). They will have only X and Y components. This because we only need them for horizontal indexing purposes. There needs to be one ray for each pixel column of your screen or viewport. Any point on the X/Y plane in world coordinates which lies along this ray would be mapped to that pixel column when projected. Each ray will have an angle relative to the X axis in world coordinates. Here is another fine graphic to help you visualize this:



This figure has the same perspective as the last one. The projection plane is pictured on the right side of the figure as seen from above. The hash marks represent the integer pixel columns which would result from projection. The rays are originating at P and extending towards the pixel columns; one for each pixel column. Each ray has an angle associated with it; namely the angle between it and the positive X axis. So the X and Y vector components of each ray are the COS and SIN of that ray's angle.

Earlier, I mentioned calculating the intersection of these rays with a plane. This plane is our wall. But since we are concerned with only the X and Y dimensions, this plane is actually reduced down to a line again with the two (X, Y) endpoints. I am still going to refer to this line as the wall for simplicity sake.

Now suppose that we have the X and Y vector components of each ray (one for every pixel column) precalculated and stored in a table. Now mapping the texture onto the projected polygon is simple. I will give the algorithm in pseudo-code otherwise known as English:

FOR each pixel column which is partially or wholly occupied by the projected polygon
1) get that pixel column's corresponding ray
2) setup parametric intersection equations for that ray and the wall
3) solve for the parameter representing the wall
4) use that parameter as the horizontal index into the texture
5) find the texture column at that index
6) scale it onto the pixel column of the screen using linear interpolation
ENDFOR

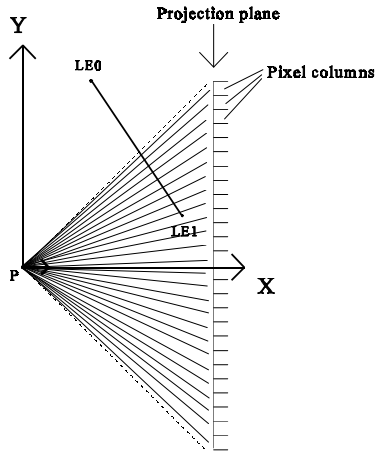
The parametric equations for the ray and wall are:

Ray:

RayX = **s** * (X component of ray)
RayY = **s** * (Y component of ray)

Wall:

WallX = LE0[x] + **t** * LE1[x]
WallY = LE0[y] + **t** * LE1[y]



Now you have four linear equations. We know that RayX=WallX and that RayY=WallY so we can easily solve for **t** (which ranges from 0.0 to 1.0). Once you get **t**, multiply it by the width of the wall (128) and you have your horizontal index into the texture. Ideally, you would set the equations up so that **t** would already be scaled by the wall width.

This method worked for me, however, it had its deficiencies. For one thing, I seemed to lose too much precision using fixed-point arithmetic (16 bit mantissa). Secondly, it seemed slow. I attributed this to math required to set up the equations and to the division required to solve it. This probably wouldn't be noticeable if you were only mapping a few polygons, but once you start rendering 100 or greater, it has to take a toll. So I thought long and hard about how I could reduce the amount of math necessary to calculate the horizontal texture index. Finally, with the help of it all clicked in my head. The answer was right there the whole time in simple geometry and my understanding was confirmed in id's data structures. Method 3 was the result of my revelation and is described in the next section.

Method 3

I am running short on time, and this document is already way too long, so I am just going to tell you how it works. It should be obvious why it works. Hey, I figured it out, so it can't be too hard (wink). Hint: remember than tangent equals sine divided by cosine.

Just as in method 2 you need a ray for each pixel column on the screen. Except you don't need to store the X and Y vector components of each ray. Instead you need to store the angle of the ray. In addition to this RayTable, you will need a table containing the value of the tangent for every angle between 90 degrees and -90 degrees (TanTable). The resolution of angle measurement is up to you. I am using a 16-bit word to store angles, so I have 65536 discreet angle values, which means my TanTable contains 32768 tangent values. (That's why the tables are so damn big. I'm using 4-byte double-words for fixed-point values so my table is 128k. Try declaring an array that size in your real-mode compiler.) That might be overkill, though. You could probably get by with one fourth of those values.

Anyway, as before, all we really need to do is get the horizontal index into the texture for a particular pixel column. The rest is just scaling and clipping. Here is how you get the horizontal index into a texture for a specific pixel column using method 3:

You must know a few things first. Here is a list of these.

1.) **A**. The angle of the wall (or the line representing the wall) in world coordinates. You will need to make a **vector** out the wall to give it direction. Always make the endpoint farthest left in the player's field of view the base of the vector. Then measure the angle between this vector and the positive X axis (0 degrees).

2.) **Pdist**. The shortest distance (in world coordinates) between the player and the wall to be drawn. This distance is only in terms of the X/Y plane and can be simply calculated with a cross-product since you know the angle of the wall (see Appendix). Its value is always positive because... if you were on the other side of the wall, the direction of the wall **vector** would change.

3.) **BP** (base point). The point along the wall **vector** closet to the player. This is a simple intersection problem since, again, you know the angle of the wall and **Pdist**. Note that **BP** may not lie on the wall itself.

4.) **BtoBP**. The distance (in world coordinates) between the base of the wall vector and the **BP**. Again, this distance is in the X/Y plane only. This too can be calculated with a simple cross-product. It should be thought of in terms of the vector direction. So a negative value means the **BP** is 'behind' the base of the wall vector.

All of the above variables can and should be calculated on a 'per-wall' basis. You definitely want these calculations outside of your surface-rendering loop. Many of these calculations can be done prior to translating and rotating the data to the player's perspective too. This, in particular, allows you to calculate the angles for all of your walls and store them as part of the data for that wall when you are generating your 3D environment.

OK, here it is. This is how you get the horizontal index for one pixel column of a wall projected onto the 2D projection plane. You need to repeat it for each pixel column which your polygon occupies. You should probably draw this on paper as you follow it.

1) Get the angle of the ray associated with this pixel column. Call it **RayA**.
2) Add 90 degrees to **A**. Then subtract **RayA** from that quantity. This becomes the index into your table of tangents.

TanIndex = (**A** + 90 degrees) - **RayA**

3) Get the value from your tangent table at index **TanIndex**. Call it **Tan**.

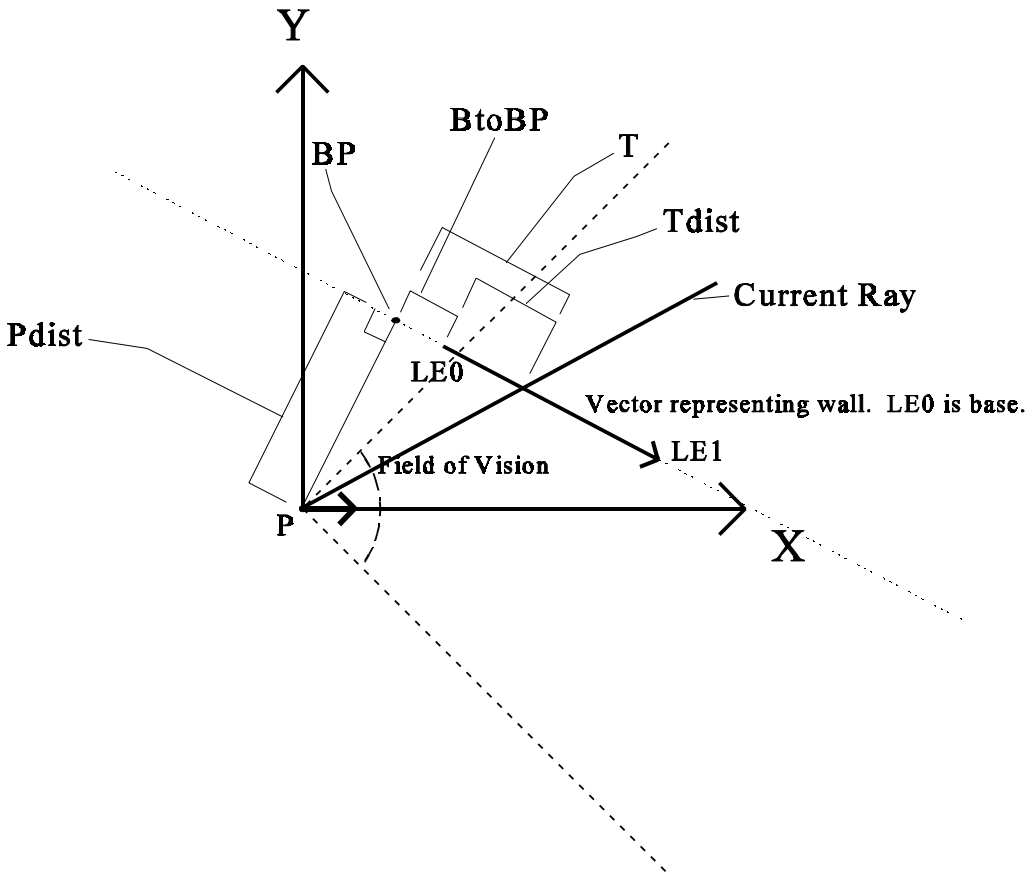
Tan = TanTable[**TanIndex**]

4) Multiply **Tan** times **Pdist**. Call this **T**.

T = Tan *Pdist

5) Subtract **BtoBP** rom **T**. The result is your horizontal index.

Tdist = T - **BtoP**.



That's all, folks. This doc turned out a lot longer than I had expected. I think I went into too much detail at the beginning and probably not enough at the end. But that is probably good. If you understand methods 1 and 2 you should understand method 3. See? I told you it was simple. For those of you more interested in the use of BSP trees, I refer you to dmspec13.zip. It describes how they are organized and gives a basic algorithm for creating them. It even has some nifty ASCII figures for illustration. As for traversing the tree for hidden surface removal, e-mail me if you can't figure it out.

I am hereby inviting everyone to submit their opinion about this document. If you have anything to say, whether it's about the three methods, about making this a better document, about anything I didn't say which I should have, about my use (or misuse) of terminology, about other aspects of graphics programming, or about anything whatsoever, please e-mail me at **doughty@bnr.ca**. Please include the word graphics somewhere in the subject. I can't promise a response, because a whole lot of people have mailed me already, but I can promise to do my best.

ACKNOWLEDGEMENTS

I would like to say thanks to the following people for various things.

Dr. Fussell, Graphics Prof/Wizard, University of Texas at Austin. I learned more in his class than in any other I took.

id, inc., my inspiration. These guys have my utmost respect. Doing what I wanna do and doing a damn good job of it. I hope you didn't patent your WAD format...

Matt Fell, Author of Doom Spec, a wonderful service to any doom fan. I could have never conceptualized the necessary data structures without it. I recommend it to any beginning graphics or game programmer.

Brendan Wyber and Raphaël Quinet, Authors of DEU. The only WAD editor I've ever needed. It has been invaluable to me as a test tool. My program first used (and basically still uses) WAD files.

Tran (a.k.a. Thomas Pytel), Author of PMODE, the best free dos extender library around.

Matt Pritchard, Author of **THE** complete Mode X library.

John McCarthy, for his 100% bugless extension of Matt Pritchard's library into Tran's PMODE.

All my family and friends. Without them, who knows where I'd be... crazy, probably.

DISTRIBUTION

This document can be distributed freely without modifications by any means electronic or otherwise. Do not incorporate it into any other documents without the author's prior consent. All rights reserved by the author.

November 1994.
William D. Doughty.

Appendix - Calculating distance with cross product

In two dimensions, the cross product can be used to calculate the shortest distance between a point **P** and a line. You must know the angle **A** of the line and one of the endpoints **LE0**. With this you can construct a unit vector which represents the line. The endpoint of the line being the vector base, and the cosine and sine of the line's angle being the offset from the base to the vector head.
The second vector has the same base and the point **P** as its head. To find the distance, translate the vectors to the origin and take the cross product.

$$\mathbf{v}1 = \cos\mathbf{A} * i + \sin\mathbf{A} * j$$
$$\mathbf{v}2 = (\mathbf{P}[x] - \mathbf{LE0}[x]) * i + (\mathbf{P}[y] - \mathbf{LE0}[y]) * j$$

$$\mathbf{d} = \mathbf{v}1 \times \mathbf{v}2$$

$$\mathbf{d} = \cos\mathbf{A} * (\mathbf{P}[y] - \mathbf{LE0}[y]) - \sin\mathbf{A} * (\mathbf{P}[x] - \mathbf{LE0}[x])$$

The absolute value of d is the distance. The sign of d determines which side of the unit vector (**v**1) the point **P** lies on. Negative means **P** is on the right side of **v**1. Positive means left.