DoomTechniques

COLLABORATORS			
	TITLE : DoomTechniques		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 24, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 DoomTechniques **1**

1.1 main 1

Chapter 1

DoomTechniques

1.1 main

Doom 3D Engine Techniques
By Brian 'Neuromancer' Marshall
(Email: brianm@vissci.demon.co.uk)

Right now that that little lot is out of the way I will start this document proper....

1: Definition of Terms

Throughout this document I will be making use of many graphical terms using my understanding of them as they apply to this algorithm. I will explain all the terms below. Feel free to skip this part....

Texture:
A texture for the purpose of this is a square image.

U and V:

U and V are the equivelants of x and y but are in texture space.
ie They are the the two axies of the two dimensional texture.

Screen:
For my purposes 'screen' is the window we wish to fill: it doesn't have to be the whole screen.

Affine Mapping:
A affine mapping is a texture map where the texture is sampled in a linear fashion in both U and V.

Biquadratic Mapping:
A biquadratic mapping is a mapping where the texture is sampled along a curve in both U and V that approximates the perspective transform. This gives almost proper forshortening.

Projective Mapping:
A projective mapping is a mapping where a changing homogenous coordinated is added to the texture coordinateds to give (U,V,W) and a division is performed at every pixel. This is the mathematically and visual correct for of texture mapping for the square to quadrilateral mappings we are using.
(As an aside it is possible to do a projective mapping without the divide (or 3 multiplies) but that is totally unrelated to the matter in hand...)

Ray Casting:
Ray Casting in this context is back-firing 'rays' along a two dinesional map. The rays do however follow heights... more on that later

Sprite:
A Sprite is a bitmap that is either a monster or an object. To put it another way it is anything that is not made out of wall or floor sectins.

Sprite Scaling:
By this I mean scaling a bitmap in either x or y or both.

Right... Now thats over with onto the foundation:

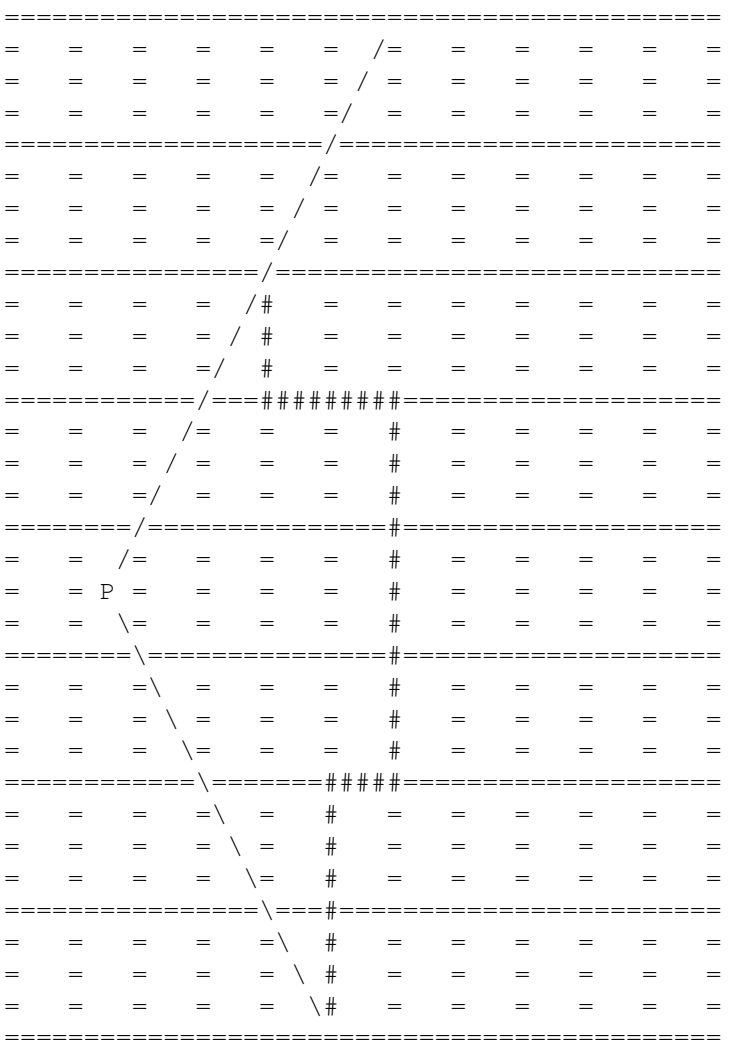
2: Two Dimensional Ray Casting Techniques

In order to make this accessible to anyone I will start by explaining 2d raycasting as used in Wolfenstein 3d style games.

2.1: Wolfenstien 3D Style Techniques...

Wolfenstein 3d was a game that rocked the world (well me anyway!). It used a technique where you fire a ray accross a 2d grid based map to find all its walls and objects. The walls were then drawn vertically using sprite scaling techniques to simulate texture mapping.

The tracing accross the map looked something like this;



(#’s are walls, = is the grid....)

This is just a case of firing a ray for each vertical
line on the screen. This ray is traced accross the map to
see where it crosses a grid boundry. Where it crosses a
boundry you cjeck to see if there is a wall there we see how
far away it it and draw a scaled vertical line from the texture
on screen. The line we draw is selected from the texture by
seeing where the line has intersected on the side of the square it
hit.
This is repeated with a ray for each vertical line on the
screen that we wish to display.
This is a very quick explaination of how it works missing
out how the sprites are handled. If you want a more detailed
explaination then I suggest getting acksrc.zip from
ftp.funet.fi in /pub/msdos/games/programming

This is someone’s source for a Wolfenstien engine written
in Borland C and Assembly language on the Pc.
Its is not the fastest or best but has good documentation
and solves similiar sprite probelms, distance probelms and has
some much better explaination of the tracing technique tahn I have

put here. I recommend to everyone interested taht you get a copy
and have a thorough play around with it.
(Even if you don't have a Pc: Everything but the drawing and video
 mode setting is done in 'C' so it should not be too hard to port
 ....)

2.2 Ray Casting in the Doom Environment

When you look at a screen from Doom you see floors, steps
walls and lots of other trappings.
You look out of windows and accross courtyards and you
say WOW! what a great 3d game!!
Then you fire your gun a baddie who's in line with you but
above you and bang! he's a corpse.
Then you climb up to the level where the corpse is and look
out the window to where you were and you say Gosh! a 3d game!!

Hmmm....

Stop gawping at the graphics for a minute and look at the map
screen. Nice line vectors. But isn't the map a bit simple???
Notice how depite colours showing you that there are different
heights. Then notice that despite the fact that there is NEVER a
place where you can exist on two different levels. Smelling a little
2d yet???
  Look where there are bridges (or sort of bridges) : managed to
see under them yet??

  The whole point to this is that Doom is a 2D games just like
its ancestor Wolfenstein but it has rather more advanced raycasting
which does a very nice job of fooling the player into thinking its a
3d game that shifting loads of polygons and back-culling, depth
sorting etc...

Right the explaination of how you turn a 2d map into the 3d
doom screen is complex so if you are having difficulty try reading
it a few times and if all else fails mail me....

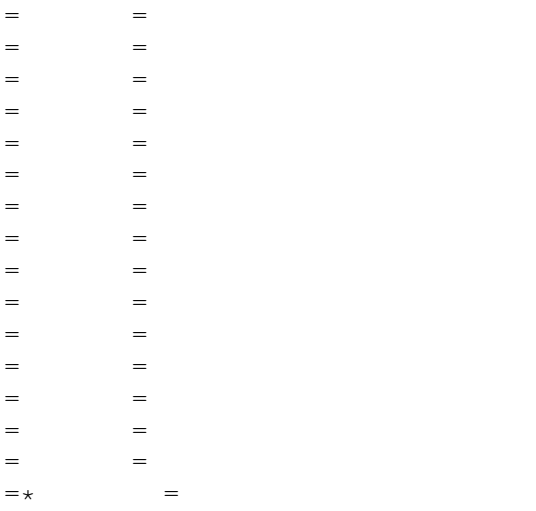
2.3 What is actually done!

  Right to start with the raycasting is started in the same
way as Wolfenstien. That is find out where the player is in the 2d
map and get a ray setup for the first vertical line on the screen.

Now we have an extra stage from the Wolfenstein I described
whcih involves a data srtucture that we will use later to actually
draw the screen.

In this data structure we start the ray off as at the bottom
of the screen. This is shown in the diagram below;

```
=================================
=       =
```



Where the '=' show the boundry of the screen and '*' is the virtual
position of the ray.

Note: the Data structure is really two structures:
One which is a set of list for each vertical 'scanline' and
One which is a corresponding list for horizontal scanlines.

  Now we start tracing the ray. We skip accross the 2d map until
we hit something interesting. By something interesting I mean something
that is an actual wall or florr section edge.
Right we have hit the edge of either a floor or wall section.
We have several things to do know. These are;

If it was a wall we hit:

1: Find out how 'high' of screen this section of wall should be
   due to the distance it is accross the 2d map.
2: Find out at what 'virtual height' it is: This is so that we can see
   where in the vertical scanline in comes for testing where to insert
   it and for clipping it.
3: Test in our structure to see if you draw it or not.
   (This is done so that you can look through windows : how this works
    will become apparent later.)
4: If any of the wall segment is visible then we find out where along
   the texture we have hit it and write into the structure the area of
   the screen it takes up as well as the texture, the point where we
   have hit the texture and the size it should be on screen. (This is
   so that we can draw it correctly even if the whole span is not on
   screen.

If it was a floor section that we hit:

1: Find out where on the vertical line we are working the floor section
   that the ray has hit is. (We know the height of the the floor in the
   virtual map (2d) and we know the height of the player and the distance
   of the floor square from the player so it is easy).
   As a side effect of this we now know the U,V value where the ray has

hit the floor square.

2: Trace Accross the floor square till we hit the far edge of the floor square : we then workout where this is on the vertical scanline using the same technique as above. We now know the vertical span of the floor section, and where on the span it is.

3: We check to see if the span is visible on the vertical span. If it is or part of it is used then we mark that part of the vertical scanline as used.
We also have to make use of the horizontal buffer I mentioned. We insert into this in 2 places. The first is the x coordinate of where we hit the floor square into the y line where we where on the screen. Phew got that bit?? We also insert here the U,V value which we knew from the tracing. (I told you we'd need it later...)

As you can see there's a little more to hiting a floor segment than a wall segment. Also note that a you exit a floor segment you may also hit a wall segment.

Tracing the individual ray is continued until we hit a special kind of wall. This wall is marked as a wall that connects to the ceiling. This is one place to stop tracing this ray. However we can stop tracing early if we have found enough to fill the whole vertical scanline then we can stop whenevr we have done this.

Next come a trick. I said we were tracing along a 2d map. Well I lied a bit. There are (In my implementation at least..) TWO 2d maps. One is basically from the floor along including all the 'floor' walls and everything up to and including the walls that join onto the ceiling. The other map is basically the ceiling (with anything coming down from the ceiling on it if you are doing this: this makes life a little more complex as I'll explain below..)
Now when we have traced along the bottom map and hit a wall that connects to the ceiling then we go back and trace along the ceiling from the start to fill in the gaps. There is a problem with this however. The problem is when you have things like a monolith or something else built out of walls jutting down from the ceiling. you have to decide whether to draw it or draw whatever was already in the scanline structure. This means either storing extra information in the buffer ie z coordinates or tracing along both the ceiling and floor at the same time.... for most people I would suggest just not having anything jutting down from the ceiling.
Also you could trace backwards instead of starting a new ray. This would be fasterfor many cases as you wouldn't be tracing through lots of floor squares that aren't on screen. By tracing backwards you can keep going up the vertical scanline and you know that you are on the screen. As soon as something goes off the top of the screen you can handle that and then stop tracing.

Phew. has everyone got that???

Now we just go back and fire rays up the rest of the vertical scanlines. Easy!!???

At the end of this lot we have the necessary data in the two buffers to go back and draw the screen background.

(There is one more thing done while tracing but I'll explain that later...)

Oh... one other thing... you have may want to change the raycasting a bit to subdivide the map... it helps with speed.
And don't forget the added complexity that walls aren't all at 90 degrees to each other...

3: Drawing the walls and Why it works!!

If you are familiar with Wolfenstein then please still read this as it is esential background to understanding the floor routine.

As all of you probably know the walls are drawn by scaling the line of the texture to the correct size for the screen. The information in the vertical buffer makes this easy. What you probably don't know is why this creates texture mapping that is good enough to fool us.

The wall function is a Affine texture mapping. (well almost) Now affine texture mappings look abysmal unless you do quite a lot of subdivision (The amount needed varies according to the angle the projected square is at.). So why does the Doom technique work??

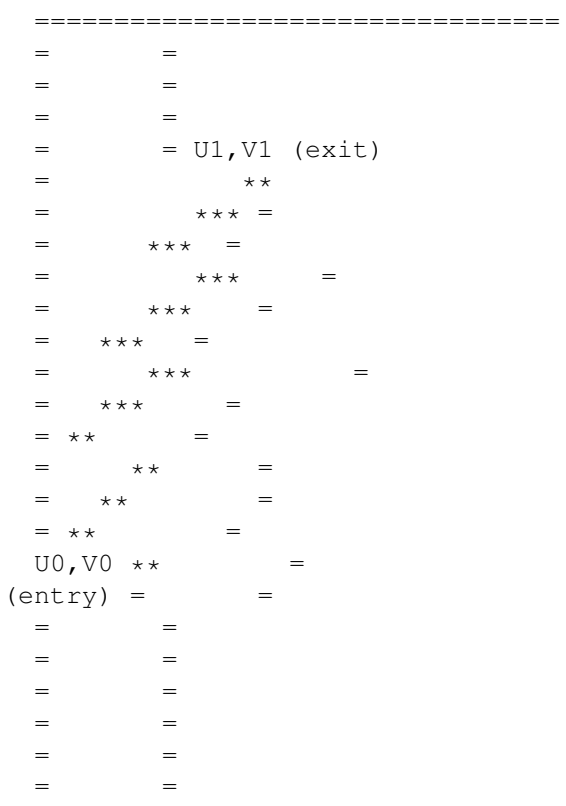
Well when we traced the rays we found out exactly where along the side of the square we hit we were in relation to the width of the texture. This means that the top and bottom pixels of the scaled wall piece are calculated correctly. This means that we have effecively subdivided the texture along vertical scanlines and as the effective subdidvisons are calculated exactly with proper forshortening as a result of the tracing. So the ray casting has made the texture mapping easy for us.
(We have enough subdivision by this scanline effect as the wall only rotates about one axis and we have proper foreshortening.)

This knowlege helps us understand how to do the floors and why that works.

We can now draw all the wall segments by just looking at the buffer and drawing the parts marked as walls.(Skiping where we put in the bits used by the floor/ceiling bits: we draw them later.)

4: Drawing the Floor/Ceiling and why it works!

If you have grasped why the walls work then you have just about won for the floors.
We have the information needed to draw the floors from the horizontal buffer.
All we have to do is look at the horizontal spans in the buffer and draw them in all.
Each of these spans has 2 end coordinates for which we have exact texture coorinates. This tells us which line across the texture we have to step along to do an Affine or linear mapping.
This is shown below;



(apologies for the wonky line: it should be straight!!)

Now...as the end coordinates are correct and the axis along which forshortening takes place is not involved (this is a fudge) we can step linearly along this line across the texture to approximate the mapping. (This is far easier than a proper texture map).

This is effectivly a wall lying on its side which works as the texture coordinates at the ends of the span have been calculated correctly. This is a benefit of the raycasting we used to find everything.

Easy huh??

5: Sprites

The Sprites are really quite easy to do. The basic technique is the same as used in Wolfenstein 3d.

This is done as follows:

When you enter a 'square' on the floor map you test to see if there are any sprites in the square. If there are you flag that sprite as visible and add it to a list of visible sprites.

When you have finished tracing and drawing the walls and floor you depth sort the sprites and draw them from the back to the front. (painters algorithm). The only complication in drawing them is that you have to check buffer that has the walls in, in order to clip the sprites correctly.

(If you're interested in Doom you can occasionally see large explosions (ie BFG) slip partially behind a wall segment.)

On possibly faster way of handling the sprites would be to mark them like wall segments as you find them in the buffer. The only (ONLY!) complication to this approach is that sprites can have holes in them. By this I mean things like the gap between an arm and a leg which should be the background colour.

6: Lighting and Depth Cueing

Lighting and Depth Cueing fits nicely in with the way that we have prepared the screen ready for drawing.

All we have to do is see how far away we are when we found either the floor or wall section and set the light level according to the distance.

The other thing that is applied is a light level. This is taken from the map at the edges where you have hit something. As the map is 2D it is easy to manage lighting, flickering etc.

For things like pools of light on the floor all you have to do is subdivide that patch of floor so that you can set the bit under the skylight to a lighter colour. Its also very easy to frig this for the lighting goggles.

7: Controlling the Baddies

This is pretty easy: all you have to think about is moving and reacting on a 2d map. the only complications are things like the monsters looking through windows and seeing a player but this all degenerates into a simple 2d problem. Things like deciding whether the player has been hit or has he/she hit a monster is just another case of firing a ray. (Or do it another way...)

8: Where next???

Thats all folks... hopefully a useful and intersting insight into my Doom engine works.

As to the question where next... well I already have some enhancements to my Doom enigine and others are in the works...

Some of what you may eventually see are:

- Proper lighting (I have done this already...its easier than you think)
- Non-Vertical walls (i.e. Aliens style corridors...)
- Orgranic Walls (i.e. Curved like the Aliens nest...)
- Fractal Landscapes (This one is still very much a theory but how about being able to go outside and walk up and down hills etc??)

If there are bits people are really shaky about I may post a new version of this... but I cannot get into implimentation issues as all implementation work is under copyright...

By the way if anyone out there implements this I'd love to here how you get on...

Anyone got any comments or any other interesting algorithms???