http://archive.gamedev.net/archive/reference/articles/article872.html

**Advanced Raycasting Techniques**

Copyright © 1999 by Andreas Christian Seidel , All rights reserved!

**Abstract**

This tutorial is for all those coders out there who want to know how to get more out of raycasting. It is not needed to know something about the 'conventional' way of raycasting, but it may help you to understand what I'm talking about, if you already know how to do raycasting. You will learn how to produce a more complex world using an advanced raycasting technique. There will be no irritating words about BSP trees, portal rendering or something else you propably heard of before, when reading articles about programming a polygonal 3D-engine. This tutorial is dedicated to pure raycasting!

**Introduction**

The main idea behind raycasting is to follow a virtual ray of light on it's way through a virtual world made up by bits and bytes within your computers memory. The conventional way of raycasting has two main points that are different to the technique I'll explain in this tutorial:

the map is a 2D-array where a number > 0 represents a solid block and a zero stands for an empty space

the rays casted out are traced in very small steps to get a resolution that is high enough to produce smooth results

This leads to the following restrictions:

you can only have orthogonal walls

all walls have the same size, since every entry in the map is same-size

You will surely agree that these are restrictions that nobody is willing to tolerate in a 3D-engine any more. The following sections will show you how to do raycasting in another, better way to produce better results. Enjoy!

**Aims**

The aims of this new approach are obvious:

we want to produce levels with non-orthogonal walls

the walls should not all have the same size

How can we do this?

**First step**

The first step is to think about a new level-format. If our level should contain any type of walls a 2D-map as used in a conventional raycaster is no longer the right thing. We need a more variable format.

It is a good idea to save all the walls as lines. Every wall has a starting point (x1/y1) and an end point (x2/y2). The resolution of the map depends on how fine the graphical results should be. But more on that later.

But if you think about this, you'll propably get some problems. Let me help you: you think that it is impossible to create a level with hundreds of walls and render the view at acceptable speed, though? Well, if you tried to handle all those walls once per frame you'd propably wait for minutes to get one frame rendered! But there is a solution for that.

**Using sectors**

The easiest way to get around handling all the walls at once is to use sectors. You put all the walls of one room together to one sector. By doing that, your level will be completely divided into sectors consisting of a certain number of walls. I think it should be easy to understand, that the player is in one sector at level-startup and moves from one sector to the other during the game. Once you have the sector the player is in, you know all the walls that are possibly seen by the player. These are the walls you have to test for rendering first.

**Casting out rays through the level**

But how do I render those walls using raycasting? Well, here is the answer: Your walls are lines. And you can calculate the slope of a line, right? In case you do not know how, look at the following equation:

m = (y2 - y1) / (x2 - x1)

(where m is the lines slope and x1, x2, y1, y2 are the walls start- and end-points)

So all of your walls defined in your sectors have a slope. Now you define your first ray. It's a line, too. And has a slope of course. Since you don't know any x- or y-values of your ray, but you know it's angle, you can use the tan()-function to get to the ray's slope:

m(ray) = tan(angle\_of\_ray)

Once you have both of this values you can do some mathematical transformation and get to an equation which lets you find the intersection between your ray and a wall! And that's the whole trick. Now let's look at this 'magic' equation.

**Calculating intersections**

First of all you have to write down your line-equations for the wall and for the ray:

(wall) y = m1 \* x + b1

(ray) y = m2 \* x + b2

Since we're searching for a point of intersection we need to put those equations together:

m1 \* x + b1 = m2 \* x + b2

Well, there are a lot of unknown values. We know m1 and m2, but x, b1 and b2 are unknown at the moment. 3 variables and only one equation? That seems to be bad. But it's not that hard.

First we eliminate b2. Think about the following: If we need to calculate an intersection of two lines. So they must be within the same coordinate system. Since the player is moving, the rays are relative to a moving system, whereas the walls remain where they are. That leads to the following idea:

One of the two b-values is a fixed value and the other is changing. It's up to you which value you want to keep and which you want to calculate, I choosed b2 to be fixed. Simply because b2 is always 0, if you define it as being a fixed value. What does this mean to our coordinate system? It means that the coordinate system has it's origin in the player-position and moves around as the player moves!

**How do we get b1?**

The formula for b1 is rather easy:

b1 = (playerx - x1) \* m1 + (playery - y1)

Once you have b1, it is possible to resolve the formula:

x = b1 / (m2 - m1)

And that's the whole thing. After you calculated the x-value of intersection it should be no problem to find the y-value. But in case you don't know how to handle that, look at the following equation:

y = y1 + (x - x1) \* m1

This is the y-value of the found intersection. So how do I calculate the intersection within my program?

Just do the following things:

inputs for this function are:

- the angle of the ray (m2)

- the wall, including the walls gradient (w, m1)

return values are:

- the x- and y-values of the intersection

declare local variable b1

b1 = (playerx - w.x1) \* m1 + (playery - w.y1)

x = b1 / (m2 - m1)

y = w.y1 + (x - w.x1) \* m1

return (x, y)

That's all. You'll have to handle some special cases where the gradients m1 or m2 get infinite or zero, but that shouldn't be a hard thing to do.

**Getting the distance**

The most important thing after calculating an intersection is to find out about it's distance to the player, since this value tells you how big your wall-slice will be and which scale you'll have to use for drawing it. We won't use the pythagorean theorem to calculate distances, since it's a rather slow algorithm. We'll use sine and cosine for calculating them.

If you have calulated an intersection you always know the angle of the ray you're currently casting. It's easy to get the sine- and cosine-value for that angle by using a look-up table or the slow trigonometric functions. Our traced ray is nothing else than a hypothenuse in a triangle. The sine of this hypothenuse is equal to the y-difference between playery and the y-value of the intersection (ys) and the cosine is equal to the x-difference between playerx and the x-value of the intersection (xs). It's now easy to get that x- and y-differences by applying a simple subtraction:

x\_diff = (playerx - xs)

y\_diff = (playery - ys)

**Now just divide them by cosine/sine and you get the length of the hypothenuse and so the correct distance!**

Note: You could use both values in any case. But it's a good idea to use the bigger value of x\_diff and y\_diff for the calculation since you'll get more precise results for your distance then.

Caution: If the cosine or sine gets zero you must use the other value, since you can't divide by zero!

Example:

if ( abs(x\_diff) > abs(y\_diff) )

distance = x\_diff / cos(angle); else

distance = y\_diff / sin(angle);

**Correcting the distance**

As you may have noticed we used trigonometric functions to calculate the distance. We worked with our coordinates as if we were in a polar-coordinate system. But we're using a cartesian-coordinate system for our map! The distances we calculate won't be the correct ones. We have to correct them before using them. This correction is done by multiplying the found distance with a correction-cosine. How do we get this 'correction-cosine'?

Imagine the player stands right in front of a wall. If you calculate the distance of the ray with the player-angle to this wall, you get the radius of a circle. Take this distance and change the angle in head! You get a circle! Remember that the height of a wall-slice is affected by it's distance to the player. If the player now stands in front of a straight wall the rays with another angle than the player-angle won't hit the wall, if they have the same distance as the first ray! That's a problem. In effect the rays must hit the wall. And the result is a bigger distance. That would result in a non-straight wall on-screen. And that's not what we want to have! But there's a solution for that...

You just have to find the 'correction-cosine' which is nothing else than the cosine of the difference-angle between the player-angle and the angle of the casted ray. This difference is zero for the ray with the player-angle and so the cosine is 1. In any other case this cosine has a different value. Just multiply this cosine with your calculated distance and you're done! You see that the multiplication does not affect the player-angle-ray since it's 1 there...

**Calculating the height of a wall-slice**

If a wall is closer to the player it has to be displayed bigger. That's not hard to know. But how do we implement that in our raycaster?

It's an easy thing. Just divide a certain value by the found distance to the wall. What you get is a value which is getting smaller as the distance gets bigger and vice versa. That's all you want, isn't it? It's up to you which value you choose. It depends on a lot of factors. You'll propably want to have walls with same height and width. So it may take some time until you find a more or less correct value for your engine. Just play around with that value until you get to the point...

Here's an example of the formula:

if (distance == 0) distance = 1; // prevent the formula from dividing by zero

height = certain\_value / distance; // get the height by just dividing

It is that easy! Most of the time goes up for finding the right values.

One word about different heights: if you assign a different 'certain\_value' to every sector, you have walls with different heights without having too much trouble!

**Scaling the wall-slices**

You propably want to put some textures onto your walls?! You have to scale a column of a texture onto one wall-slice. This is done by using a rather simple scaling algorithm that can be implemented in C or assembler. If you want to do it really fast, use assembler, otherwise C does a good job too. The idea is to draw the slice pixel by pixel and advance a certain number of texels within the texture. Since this value won't be an integer-value, we need to use fixed-point arithmetic here, since floating-point math is simply too slow for that job.

Please refer to some other article to learn something on fixed-point arithmetic, I don't want to explain it at this point, sorry.

You get your scaling-factor by just dividing the texture-y-dimension by the height of the wall-slice.

Look at this example for better understanding:

long scale;

scale = (texture.y\_dim << 16) / height; // use 16 bits fixed-point value here

We just take the y-dimension of the texture, reserve 16 bits for the decimal part of the scaling-factor and divide this new value by the height of the wall-slice. Now, we can draw the whole wall-slice by starting at texture-index 0 and advance by the scaling-factor whenever a new pixel is drawn.

Look at the following loop:

int i; // loop-variable

long index\_frac; // holds the 16 bits fixed-point version of texture-index

int tex\_index; // the 'normal' texture-index without decimal part

index\_frac = 0; // reset the fixed-point-version of the texture-index to zero

tex\_index = 0; // set texture-index to the first texel

for (i = 0; i < height; i++)

{

tex\_index += (index\_frac >> 16); // cut off the decimal part and add the whole part of

// index\_frac to the texture index

// place pixel here using tex\_index

index\_frac += scale; // advance index by scale-factor

index\_frac &= 66535; // mask out the whole part and just keep the decimal part of the

// texture-index

}

I hope you can get this. If not, try to find some other article on scaling-algorithms. Maybe you can find one. I used this algorithm in my engine and it produces good results!

Keep in mind to use as much bits as possible for your decimal part. This improves accuracy. On the other hand you need enough place for the whole part. So 16 bits is a good middle-way.

**Handling sectors**

One thing is different to 'conventional raycasting'. You're working with sectors. You need to do that to reduce the number of intersections you have to calculate. You need to mark all walls that connect one sector with another as transparent walls. For these walls your not drawing a wall but display another sector instead. You should do that recursively.

So it may look like this:

draw\_column\_for(sector)

{

-> get\_intersection\_to\_wall

-> is the wall transparent?

+ Yes, draw\_column\_for(next\_sector)

- No, draw\_wall\_slice

}

You see that this algorithm is recursively drawing columns for a certain sector. This is the best way to deal with that.

**Finding the right intersections within one sector**

One thing is important when trying to find intersections to walls within one sector. You will always find two walls that intersect with a ray. That's just because a ray is a straight line. It can intersect with a wall in front and with one wall behind the player! Ok, and it's up to you to consider which wall should be drawn and which not. Since the two found intersection will always lie in different quadrants of your cartesian coordinate-system, it's easy though to decide which one is the right for you. Just find out in which quadrant the ray's angle is in, and your intersection must occur in the same quadrant! Want to know how?

Here it is:

int get\_quadrant(int angle)

{

if ((angle >= 0) && (angle < 90)) return(1); // angle in first quadrant

if ((angle >= 90) && (angle < 180)) return(2); // angle in second quadrant

if ((angle >= 180) && (angle < 270)) return(3); // angle in third quadrant

if ((angle >= 270) && (angle < 360)) return(4); // angle in fourth quadrant

}

That's how to get the quadrant for the ray. Not too hard. But we have to find out about the quadrant of that intersection too. We just use the x- and y-differences again, to find out about it!

If the x-difference is > 0 it can only be the 1st or the 4th quadrant, otherwise the 2nd or 3rd. If the y-difference is > 0 it can only be the 3rd or the 4th quadrant, otherwise the 1st or 2nd.

If you put these two conditions together you have it! Look at this piece of source:

int get\_intersection\_quadrant(int xs, ys) // xs, ys => intersection-point

{

int x\_diff, y\_diff;

x\_diff = playerx - xs;

y\_diff = playery - ys;

if (x\_diff > 0)

{

if (y\_diff > 0) return(4); else return(1);

} else {

if (y\_diff > 0) return(3); else return(2);

}

}

Well, and it should be easy now to figure out, how a function could look like, that tells you whether an intersection is correct, isn't it?

int correct\_intersection(int xs, ys, angle)

{

if (get\_intersection\_quadrant(xs, ys) == get\_quadrant(angle)) return(1); else return(0);

}

**Moving around in your level**

To move your player around in your level you have to modify his position. Your player always walks into one direction. That can be the player-angle if he walks forward, or any other angle, if he walks e.g. to the left or the right.

The first thing you have to do is to find out about the angle the player is moving. After that you need to calculate the x-unit and the y-unit of his movement. To get the x-unit you take the cosine of the angle, for the y-unit you take the sine. Once you have those basic units you multiply them by a number which represents the length of the player's movement.

If you want to move your player by 256 map-units, just multiply the x-unit and the y-unit by 256. That's all you have to do.

void move\_player(int length, int direction)

{

int x\_unit, y\_unit;

x\_unit = cos(direction) \* length;

y\_unit = sin(direction) \* length;

playerx += x\_unit;

playery += y\_unit;

}

Well, your player is moving now. But you will notice that he can walk to whereever he wants to! He can just leave a sector and you won't even know about it!

**Blocking the player**

If you don't want the player to walk through your walls you need to test how far the player is away from the next wall. Just calculate the distance to the wall a ray hits with the angle that is equal to the players moving direction. If the distance is smaller than a certain value leave the move\_player-function and stop him from walking through the wall!

The code for that is as follows:

void move\_player(int length, int direction)

{

int x\_unit, y\_unit;

long distance;

distance = get\_distance(direction); // get the distance to a wall hit by the ray with angle 'direction'

if (distance < 256) return; // leave the function if distance is too small

x\_unit = cos(direction) \* length;

y\_unit = sin(direction) \* length;

playerx += x\_unit;

playery += y\_unit;

}

Well, you should not try to block the player if you hit a transparent wall! Just test wether the wall you hit is transparent and decide to return afterwards.

**Sector-transition**

Ok, your player can now move around in one sector and can't walk through your walls. But if you'd allow him to just walk through a transparent wall, you'll get a black screen! Remember: The player is always in one sector the 'player\_sector'. You check all the walls within that player\_sector for possible intersection with your rays. But you don't test the walls of all the other sectors! So if you move the player outside the player\_sector WITHOUT changing the player\_sector, you'll get no correct intersections any more and you'll just see a black screen!

The only thing to do, is to test which sector the player is running into and change the value player\_sector to the new sector-number. That's all. If you organize your level-data in the right way, you'll save the number of the next sector with every transparent wall, because it's easy then to get the needed information fast.

**Conclusion**

Well, this is the first part of my tutorial on advanced raycasting. I hope that you understood the main concept and that you're able now to program your own simply sector-based raycaster. In the next parts I will tell you how to implement floor- and ceiling-textures, lighting, doors, jumping, looking up and down and so on. I'll try to give you all the information you need to create an engine like REX3D which is the name of my 3D-engine. If you want to look at it to see how much you can get out of raycasting, please visit my homepage at http://www.phryzer.de/rex3d

You can get the source-code of an older version of REX3D, if you mail me. My address is  
webmaster@phryzer.de

So if you have any suggestions, questions or whatever, just tell me.

Date this article was posted to GameDev.net: 12/2/1999

(Note that this date does not necessarily correspond to the date the article was written)