**INTRODUCTION**

There has been an explosive growth in the personal computer market in the past few years\*. This growth, in part, is generated by excitement and curiosity for multimedia titles. This project is an attempt to obtain some knowledge and experience on the multimedia title development process. Specifically, we will take a closer look on issues related to the development of a three dimensional multimedia game.

(\*Side Note: This document was written in 1996.)

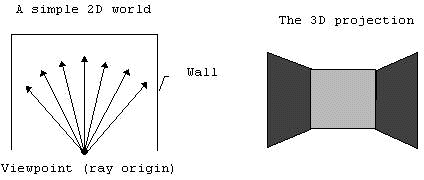
**A BRIEF HISTORY**

Ray-casting sensation began with the release of a game, Wolfenstein 3D (iD Software), in 1992 (see[Figure 3 on the next page](http://www.permadi.com/tutorial/raycast/rayc2.html#FIGURE3)). In Wolfenstein 3D, the player is placed on a three dimensional maze-like environment, where he/she must find an exit while battling multiple opponents. Wolfenstein 3D becomes an instant classic for its fast and smooth animation. What enables this kind of animation is an innovative approach to three dimensional rendering known as "ray-casting."

Wolfenstein 3D was developed and created by Id Software. Henceforth, Id's programmer, John Carmack, might well be the person who initiates the ray-casting sensation (Myers 5).

**WHAT IS RAY-CASTING?**

Ray-casting is a technique that transform a *limited* form of data (a very simplified map or floor plan) into a 3D projection by tracing rays from the view point into the viewing volume (LaMothe 942). For example, ray-casting transforms something like A into B in [Figure 1](http://www.permadi.com/tutorial/raycast/rayc1.html#FIGURE1).

**Figure 1A and B**  


Note that this is not the only application of ray-casting. Ray-casting can also be used to render terrain map such as in [Figure 2](http://www.permadi.com/tutorial/raycast/rayc1.html#FIGURE2) (below) for instance. The important point to remember is that ray-casting "traces rays backward from viewer's eye to objects."

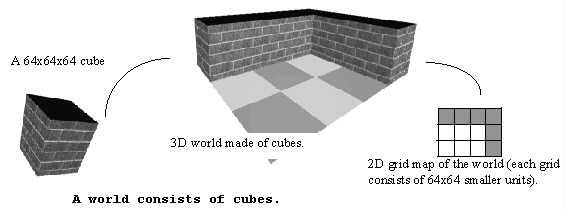
**RAY-CASTING STEP 1: CREATING A WORLD**

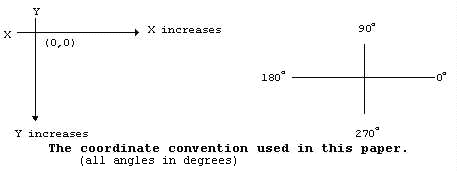
To illustrate the process of ray-casting, we will create a maze world that has the following geometric constraints:

1. Walls are always at 90° angle with the floor.  
2. Walls are made of cubes that have the same size.  
3. Floor is always flat.

For our purpose, each cube will have the size of 64x64x64 units. (The choice of 64 is arbitrary, but it will be useful to pick a number that is a multiple of 2; because we can perform some arithmetic shift operations on such number (shift operations are faster than multiplication or division). The larger the size of the cube, the blockier the world will look like, but smaller cube will make the rendering slower.)

Such a world is illustrated in [Figure 6](http://www.permadi.com/tutorial/raycast/rayc3.html#FIGURE6).

**Figure 6**  


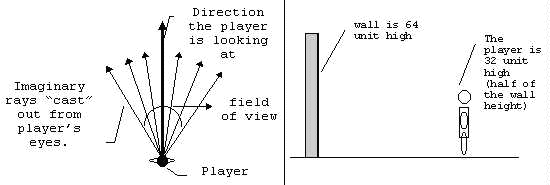
Before continuing, we will define our coordinate system so that there is no confusion. The coordinate system that we use is illustrated in [Figure 7](http://www.permadi.com/tutorial/raycast/rayc3.html#FIGURE7).   
**Figure 7**  
   
Note: Any kind of cartesian coordinate system would work just as well. However, you do have to be consistent (don't use the top-down coordinate system for one thing but then use the buttom-up coordinate for others). You'd be likely to confuse yourself if you do this - I did.

**RAY-CASTING STEP 2: DEFINING PROJECTION ATTRIBUTES**

Now that we have the world, we need to define some attributes before we can project and render the world. Specifically, we need to know these attributes:

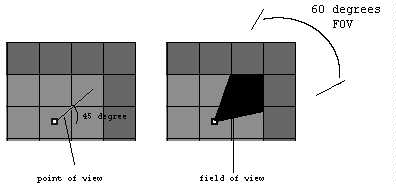
1. Player/viewer's height, player's field of view (FOV), and player's position.   
2. Projection plane's dimension.  
3. Relationship between player and projection plane.

The player should be able to see what is in front of him/her. For this, we will need to define a field of view (FOV). The FOV determines how wide the player sees the world in front of him/her (see [Figure 8](http://www.permadi.com/tutorial/raycast/rayc4.html#FIGURE8)). Most humans have a FOV of 90 degrees or more. However, FOV with this angle does not look good on screen. Therefore, we define the FOV to be 60 degrees through trial and experimentation (on how good it looks on screen). The player's height is defined to be 32 units because this is a reasonable assumption considering that walls (the cubes) are 64 units high.

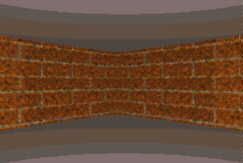
**Figure 8**  


To put the player inside the world, we need to define the player's X coordinate, the player's Y coordinate, and the angle that the player is facing to. These three attributes forms the "point of view" of the player.

Suppose that the player is put somewhere in the **middle** of **grid coordinate** (1,2) at a viewing angle of 45 degrees relative to the world, then the player's *point of view* and FOV will be like in [Figure 9](http://www.permadi.com/tutorial/raycast/rayc4.html#FIGURE9). (One grid consist is 64 x 64 units. Thus, we can also say that the player is in **unit coordinate**(96,160)).

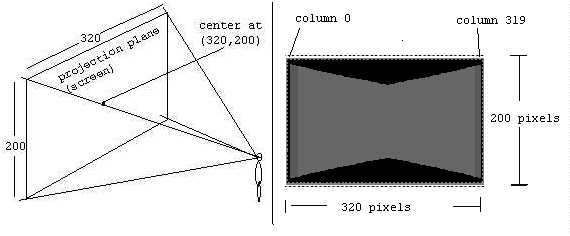
**Figure 9  
Player is in the middle of grid coordinate (1,2) or unit coordinate (96,160) with a viewing angle of 45 degrees and a field of view of 60 degrees.**  


We need to define a projection plane so that we can project what the player sees into the projection plane. A projection plane of 320 units wide and 200 units high is a good choice, since this is the resolution of most VGA video cards. (Video resolution is usually referred in pixels, so think of 1 pixel as equal to 1 unit.)   
When the player's point of view is projected into the projection plane, the world should look like the scene in [Figure 10](http://www.permadi.com/tutorial/raycast/rayc4.html#FIGURE10) below.

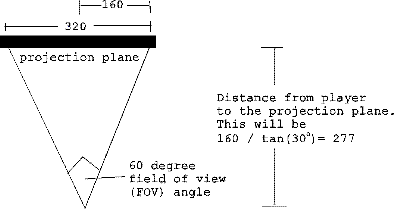
**Figure 10  
**

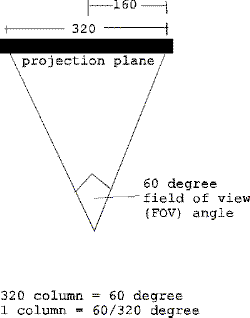
By knowing the ***field of view (FOV)*** and the ***dimension of the projection plane***, we can calculate ***the angle between subsequent rays*** and ***the distance between the player and the projection plane***. These steps are illustrated in[Figure 11](http://www.permadi.com/tutorial/raycast/rayc5.html#FIGURE11) (Many books define these last two values arbitarily, without telling the reader where the values come from, here is the justification.)

**FIGURE 11  
Here is what we know:**



**Here is what we can calculate (brush up on your Trigonometry if you don't understand, most of these are high school level math):**





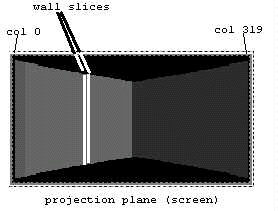
So now we know:

* Dimension of the Projection Plane = 320 x 200 units
* Center of the Projection Plane = (160,100)
* Distance to the Projection Plane = 277 units
* Angle between subsequent rays = 60/320 degrees

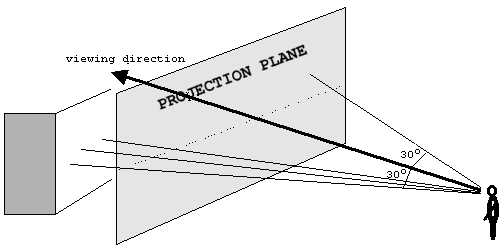
(We will occasionally refer the "angle between subsequent rays" as the "angle between subsequent columns." Later, this angle will be used to loop from column to column. The distance between player to the projection plane will be used for scaling.)

**RAY-CASTING STEP 3: FINDING WALLS**

Notice from the previous image ([Figure 11](http://www.permadi.com/tutorial/raycast/rayc5.html#FIGURE11)), that the wall can be viewed as collection of 320 vertical lines (or 320 wall slices).

**Figure 12**  


This is precisely a form of geometrical constraints that will be suitable for ray-casting. Instead of tracing a ray for every pixel on the screen, we can trace for only every vertical column of screen. The ray on the extreme left of the FOV will be projected onto column 0 of the projection plane, and the right most ray will be projected onto column 319 of the projection plane.

**Figure 13: Rays looking for walls.**  


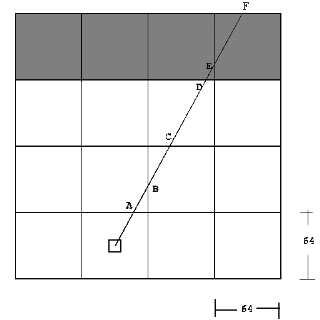
Therefore, to render such scene, we can simply trace 320 rays starting from left to right. This can be done in a loop. The following illustrates these steps:

1. Based on the viewing angle, subtract 30 degrees (half of the FOV).  
2. Starting from column 0:

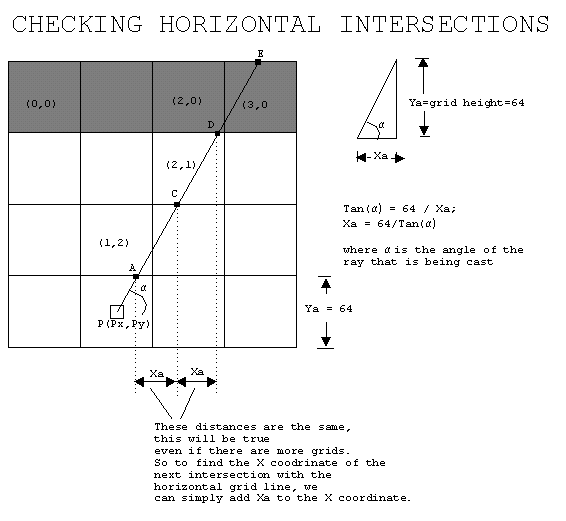
A. Cast a ray. (The term "cast" is a bit confusing. Imagine the player as a wizard who can "cast" rays instead of spells. The ray is just an "imaginary" line extending from the player.)  
B. Trace the ray until it hits a wall.  
C. Record the distance to the wall (the distance is equal to the length of the ray).

3. Add the angle increment so that the ray moves to the right (we know from Figure 10 that the value of the angle increment is 60/320 degrees).  
4. Repeat step 2 and 3 for each subsequent column until all 320 rays are cast.

The trick to **step 2A** is that instead of checking each pixels, we only have to check each grid. This is because a wall can only appear on a grid boundary. Consider a ray being traced as in [Figure 14](http://www.permadi.com/tutorial/raycast/rayc6.html#FIGURE14). To check whether this ray has hit a wall or not, it is sufficient to check the grid intersection points at A, B, C, D, E, and F. 

**Figure 14: This ray intersects the grids at points A,B,C,D,E, and F.**  


To do this, we need to check any grid intersection points that are encountered by the ray; and see if there is a wall on the grid or not. The best way is to check for horizontal and vertical intersections separately. When there is a wall on either a vertical or a horizontal intersection, the checking stops. The distance to both intersection points is then compared, and the closer distance is chosen. This process is illustrated in the following two [figures](http://www.permadi.com/tutorial/raycast/rayc7.html#FIGURE15).

Figure 15    


Steps of finding intersections with horizontal grid lines:

1. Find coordinate of the first intersection (point A in this example).  
2. Find Ya. (Note: Ya is just the height of the grid; however, if the ray is facing up, Ya will be **negative**, if the ray is facing down, Ya will be **positive**.)  
3. Find Xa using the equation given above.  
4. Check the grid at the intersection point. If there is a wall on the grid, stop and calculate te distance.  
5. If there is no wall, extend the to the next intersection point. Notice that the coordinate of the next intersection point -call it (Xnew,Ynew) is Xnew=Xold+Xa, and Ynew=YOld+Ya.

As an example the following is how you can get the point A:

**Note: remember the Cartesian coordinate is**

**increasing downward (as in** [page 3](http://www.permadi.com/tutorial/raycast/rayc3.html)**), and**

**any fractional values will be rounded down.**

======Finding horizontal intersection ======

1. Finding the coordinate of A.

If the ray is facing up

A.y = rounded\_down(Py/64) \* (64) - 1;

If the ray is facing down

A.y = rounded\_down(Py/64) \* (64) + 64;

(In the picture, the ray is facing up, so we use

the first formula.

A.y=rounded\_down(224/64) \* (64) - 1 = 191;

Now at this point, we can find out the grid

coordinate of y.

However, we must decide whether A is part of

the block above the line,

or the block below the line.

Here, we chose to make A part of the block

above the line, that is why we subtract 1 from A.y.

So the grid coordinate of A.y is 191/64 = 2;

A.x = Px + (Py-A.y)/tan(ALPHA);

In the picture, (assume ALPHA is 60 degrees),

A.x=96 + (224-191)/tan(60) = about 115;

The grid coordinate of A.x is 115/64 = 1;

So A is at grid (1,2) and we can check

whether there is a wall on that grid.

There is no wall on (1,2) so the ray will be

extended to C.

2. Finding Ya

If the ray is facing up

Ya=-64;

If the ray is facing down

Ya=64;

3. Finding Xa

Xa = 64/tan(60) = 36;

4. We can get the coordinate of C as follows:

C.x=A.x+Xa = 115+36 = 151;

C.y=A.y+Ya = 191-64 = 127;

Convert this into grid coordinate by

dividing each component with 64.

The result is

C.x = 151/64 = 2 (grid coordinate),

C.y = 127/64 = 1 (grid coordinate)

So the grid coordinate of C is (2, 1).

**(C programmer's note: Remember we always round down,**

**this is especially true since**

**you can use right shift by 8 to divide by 64).**

5. Grid (2,1) is checked.

Again, there is no wall, so the ray is extended

to D.

6. We can get the coordinate of D as follows:

D.x=C.x+Xa = 151+36 = 187;

D.y=C.y+Ya = 127-64 = 63;

Convert this into grid coordinate by

dividing each component with 64.

The result is

D.x = 187/64 = 2 (grid coordinate),

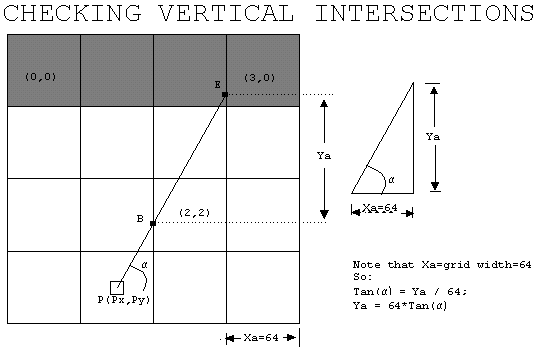
D.y = 63/64 = 0 (grid coordinate)

So the grid coordinate of D is (2, 0).

6. Grid (2,0) is checked.

There is a wall there, so the process stop.

(Programmer's note: You can see that once we have the value of Xa and Ya, the process is very simple. We just keep adding the old value with Xa and Ya, and perform shift operation, to find out the grid coordinate of the next point hit by the ray.)



Steps of finding intersections with vertical grid lines:

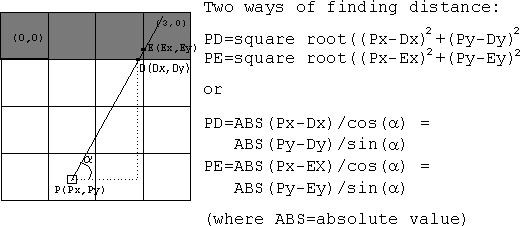
1. Find coordinate of the first intersection (point B in this example).  
The ray is facing right in the picture, so B.x = rounded\_down(Px/64) \* (64) + 64.  
**If the ray had been facing left B.x = rounded\_down(Px/64) \* (64) - 1.**  
A.y = Py + (Px-A.x)\*tan(ALPHA);  
2. Find Xa. (Note: Xa is just the width of the grid; however, if the ray is facing right, Xa will be **positive**, if the ray is facing left, Ya will be**negative**.)  
3. Find Ya using the equation given above.  
4. Check the grid at the intersection point. If there is a wall on the grid, stop and calculate te distance.  
5. If there is no wall, extend the to the next intersection point. Notice that the coordinate of the next intersection point -call it (Xnew,Ynew) is just Xnew=Xold+Xa, and Ynew=YOld+Ya.

In the picture, First, the ray hits point B. Grid (2,2) is checked. There no wall on (2,2) so the ray is extended to E. Grid (3,0) is checked. There is a wall there, so we stop and calculate the distance.

In this example, point D is closer than E. So the wall slice at D (not E) will be drawn

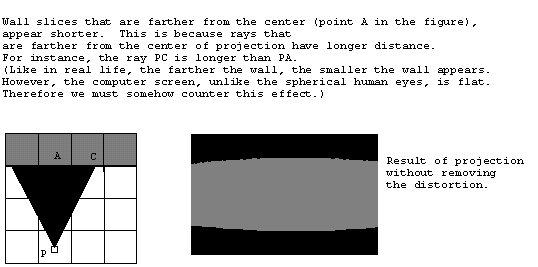
**RAY-CASTING STEP 4: FINDING DISTANCE TO WALLS**

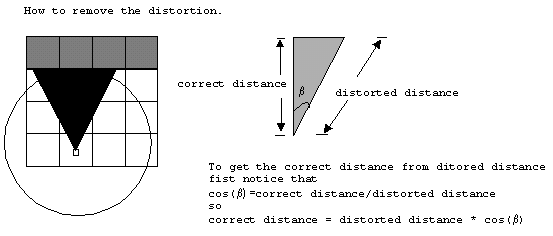
There are several ways to find the distance from the viewpoint (player) to the wall slice. They are illustrated below.

**Figure 17: Finding distance to a wall slice.**    


The sine or cosine functions are cheaper to implement because they can be pre-computed and put into tables. This can be done because ALPHA (player's POV) has to be between 0 to 360 degrees, so the number of possibilities are limited (the square root method has a virtually unlimited possible values for the x's and y's).

Before drawing the wall, there is one problem that must be taken care of. This problem is known as the "fishbowl effect." Fishbowl effect happens because ray-casting implementation mixes polar coordinate and Cartesian coordinate together. Therefore, using the above formula on wall slices that are not directly in front of the viewer will gives a longer distance. This is not what we want because it will cause a viewing distortion such as illustrated below.

**Figure 18**  


**Figure 19**  


Thus to remove the viewing distortion, the resulting distance obtained from equations in [Figure 17](http://www.permadi.com/tutorial/raycast/rayc8.html#Figure 17) must be multiplied by cos(BETA); where BETA is the angle of the ray that is being cast relative to the viewing angle. On the figure above, the viewing angle (ALPHA) is 90 degrees because the player is facing straight upward. Because we have 60 degrees field of view, BETA is 30 degrees for the leftmost ray and it is -30 degrees for the rightmost ray.

**RAY-CASTING STEP 5: DRAWING WALLS**

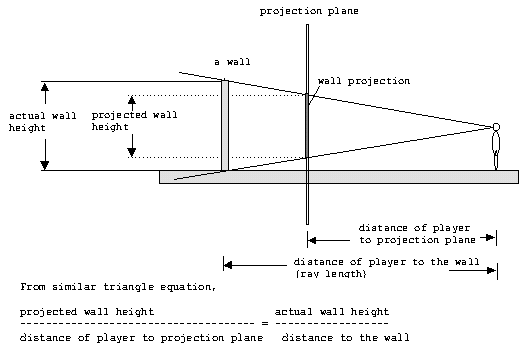
In the previous steps, 320 rays are casts, when each ray hits a wall, the distance to that wall is computed. Knowing the distance, the wall slice can then be projected onto the projection plane. To do this, the height of the projected wall slice need to be found. It turns out that this can be done with a simple formula:

Actual Slice Height

Projected Slice Height= --------------------- \* Distance to Projection Plane

Distance to the Slice

The logic behind this formula is explained in the Figure 20 below.

**Figure 20: The math behind wall scaling.**  


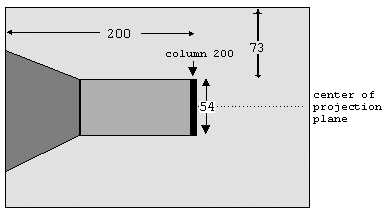
Our world consist cubes, where the dimension of each cube is 64x64x64 units, so the wall height is 64 units. We also already know the distance of the player to the projection plane (which is 277). Thus, the equation can be simplified to:

**Projected Slice Height = 64 / Distance to the Slice \* 277**

In an actual implementation, several things can be considered:

* For instance, 64/277 can be pre-computed, since this will be a constant value. Once this is calculated, the wall slice can be drawn on the screen. This can be done by simply drawing a vertical line on the corresponding column on the projection plane (screen).
* Remember where the number 277 came from?  This number can actually be deviated a bit without causing any huge impact.  In fact, it will save time to use the value of **255** because the programmer can use shift operator to save computing time (shift right by 3 to multiply, shift left to divide).

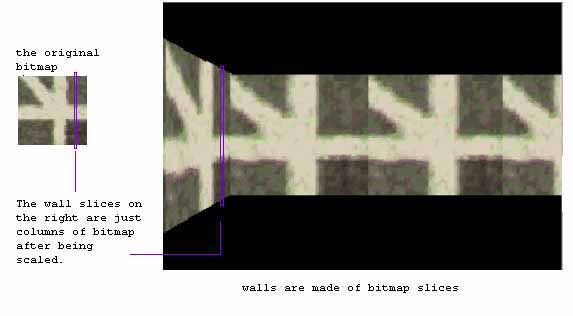
For example, suppose the ray at column 200 hits a wall slice at distance of 330 units. The projection of the slice will be 64 / 330 \* 277 = 54 (rounded up).   
Since the center of the projection plane is defined to be at 100. The middle of the wall slice should appear at this point. Hence, the top position where the wall slice should be drawn is 100-27=73. (where 27 is one half of 54). Finally, the projection of the slice will look something like the next [figure](http://www.permadi.com/tutorial/raycast/rayc9.html#FIGURE21).

**Figure 21: A partly rendered view.**  


## TEXTURE MAPPED WALLS

To make the walls more attractive, the walls can be painted with texture (bitmap) using a technique known as texture mapping. (Texture mapping in general refers to a technique of painting a bitmap/texture onto a surface.) For the cube world, we use bitmaps that have the size of 64 by 64 pixels. This size is chosen because 64 by 64 is also the size of the cube facets that we are using in our world. It is possible to use different size bitmaps, but using the same size simplifies the texture mapping process.

If we are to map a texture onto an arbitrary polygon, the texture mapping process will be complicated. Fortunately, on the ray-casting world that we are creating, texture mapping is just a matter of scaling a slice (a column) of bitmap (see [Figure 22](http://www.permadi.com/tutorial/raycast/rayc10.html#FIGURE22) below).

**Figure 22: Putting textures on walls.**  


When the ray is looking for the wall intersection, the offset (position of the ray relative to the grid) can be found easily. This offset can then be used to determine which column of the bitmap is to be drawn as the wall slice. The following [figure](http://www.permadi.com/tutorial/raycast/rayc10.html" \l "FIGURE23)illustrates the process of finding the offset.

**Figure 23: Finding the offset of bitmap.**  
