Ray Casting Tutorial

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.

(*This document was written in 1996)

A BRIEF HISTORY

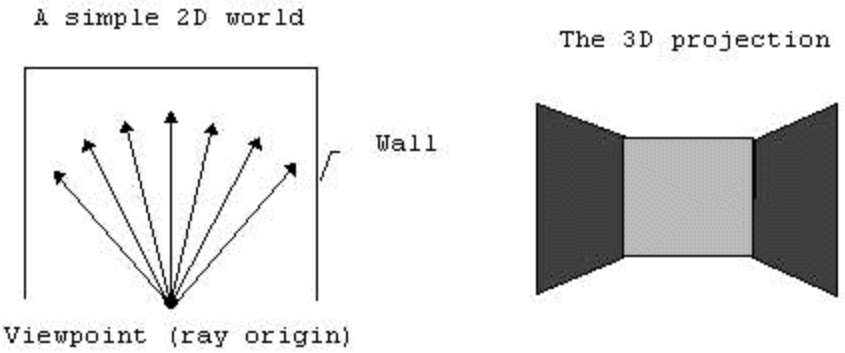
Ray-casting sensation began with the release of a game, Wolfenstein 3D (iD Software), in 1992 (seeFigure 3 on the next page). In Wolfenstein 3D, the player is placed on a three dimensional maze-like environment, where he/she must find an exit while battling multiple opponents. Wolfenstein 3D becomes 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?

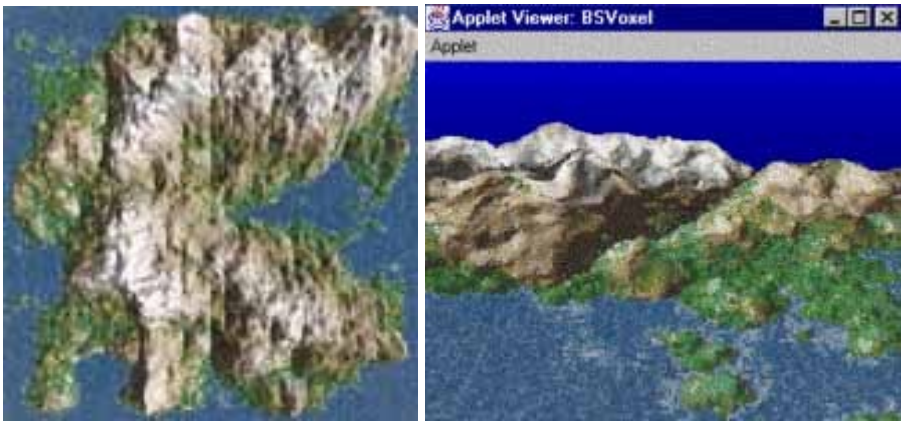
Ray-casting is a technique that transform a *limited* form of data (a very simplified map or floor plan) into a 3D projection by tracing rays from the view point into the viewing volume (LaMothe 942). For example, ray-casting transforms something like A into B in Figure 1.

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 (below) for instance. The important point to remember is that ray-casting "traces rays backward from viewer's eye to objects."

Figure 2
Here is an example of a rendering by subclass/variant of ray casting, known as height mappingor terrain mapping or voxel rendering. To learn more about voxel rendering, refer to the article by Peter Friese on More Tricks of Game Programming Gurus; or the book Black Art of 3D Game Programming by Andre La Mothe.



RAY-CASTING AND/VS RAY-TRACING

Like ray-casting, ray-tracing "determines the visibility of surfaces by tracing imaginary rays of light from viewer's eye to the object in the scene" (Foley 701).

From both definitions, it seems that ray-casting and ray-tracing is the same. Indeed, some books use both terms interchangeably. From game programmers perspective, however, ray-casting is regarded as a special implementation (subclass) of ray-tracing.

This distinctions because is made because in general, ray-casting is faster than ray-tracing. This is possible because ray-casting utilizes some geometric constraint to speed up the rendering process. For instance: walls are always perpendicular with floors (you can see this in games such as Doom or Wolfenstein 3D). If it were not for such constraints, ray-casting will not be feasible. We would not want to ray-cast arbitrary splines for instance, because it is difficult to find a geometrical constraints on such shapes.

Table 1 is a general comparison between ray-casting and ray-tracing. The main point to remember is that there are "less number of rays" to trace in ray-casting because of some "geometric constraints." Or, it can also be said that ray-casting is a special purpose implementation of ray-tracing.

TABLE 1: A COMPARISON BETWEEN RAY-CASTING AND RAY-TRACING (GAME PROGRAMMERS/GAME DEVELOPERS PERSPECTIVE)

RAY-CASTING	RAY-TRACING
Principle: rays are cast and traced*in groups* based on some geometric constraints. For instance: on a 320×200 display resolution, a ray-caster traces only 320 rays (the number 320 comes from the fact	**Principle**: each ray is traced*separately,* so that every point (usually a pixel) on the display is traced by one ray. For instance: on a 320×200 display resolution, a ray-tracer needs to trace

that the display has 320 horizontal pixel resolution, hence 320 vertical column).	320×200 (64,000) rays. (That is roughly 200 times slower than ray-casting.)
Formula: in most cases, inexact.	**Formula**: in most cases, exact.
Speed: very fast compared to ray-tracing; suitable for real time process.	**Speed**: slow; unsuitable for real time process (at least not until we got a 500Ghz machine).
Quality: resulting image is not very realistic. Often, they are blocky (Figure 3).	**Quality**: resulting image is very realistic – sometimes too realistic (Figure 4).
World: limited by one or more geometric constraints (simple geometric shapes).	**World**: almost any shape can be rendered.
Storage: small. Rendered images are not stored on disk. Normally, only the map is stored, and corresponding images are generated "on the fly."	**Storage**: Rendered images are stored on disk and loaded when needed. Presently, no hardware is fast enough for "on the fly" rendering.
Examples: Wolfenstein 3D (iD Software), Shadow Caster (Raven), Arena (Bethesda), Doom (iD Software), Dark Forces (LucasArts).	**Examples**: Examples: 7th Guest (Trilobyte), Critical Path (Mechadeus), 11th Hour (Trilobyte), Myst (Cyan), Cyberia (Xatrix).
Figure 3: Scene from Wolfenstein 3D (iD Software). Notice the blocky look. The objects (gun) and enemies (a dog) are just transparent bitmaps being scaled and blitted (i.e.: pasted) over the background.	**Figure 4**: Scene from the game 7th Guest (Virgin Software/Trylobyte). The result of the rendering is stunning. However, player's movement is restricted to a pre-determined path (because the amount of pre-rendered images are limited).

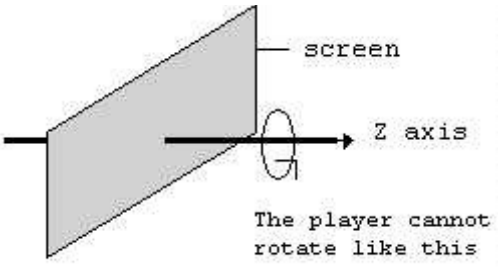
Before we begin, let's examine the limitation of ray casting.

LIMITATIONS OF RAY-CASTING

Ray casting is fast because it utilizes some geometric constraints. In most cases, walls are always at 90 degrees angle with the floor. (Note that we are not talking about the angle between walls and another walls, but the angle between walls and floor.)

Thus, a limitation that almost exists on a ray-casting game is that the viewpoint cannot be rotated along the Z axis (Figure 5). If this is allowed, then walls could be slanted and the benefit of drawing in vertical slices will be lost. This inability to rotate along the Z axis is one of the reason of why ray-casting environment is not regarded as a true three dimensional environment.

Figure 5: On a ray-casting environment, the player can move forward, backward, and turn left or right; but cannot rotate/swing around the Z-axis (this kind of Z-axis rotation is called a tilt).



RAY-CASTING STEP 1: CREATING A WORLD

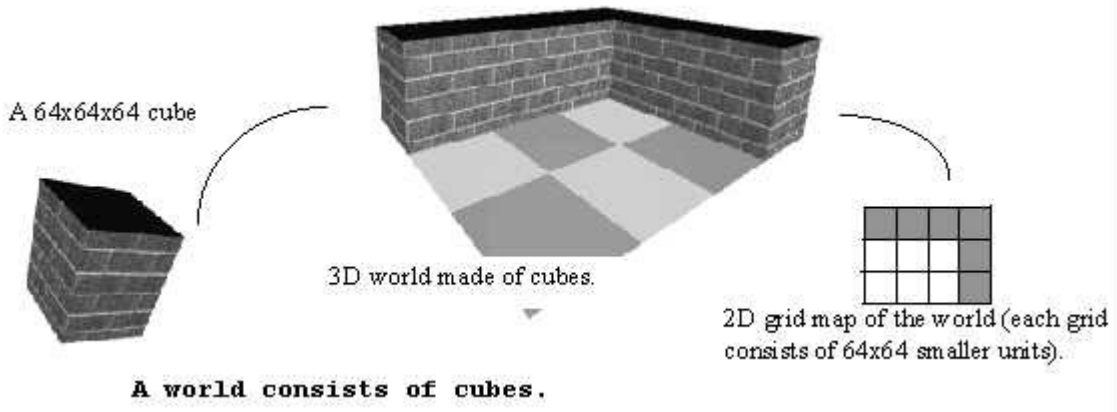
To illustrate the process of ray-casting, we will create a maze world that has the following geometric constraints:

- 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.

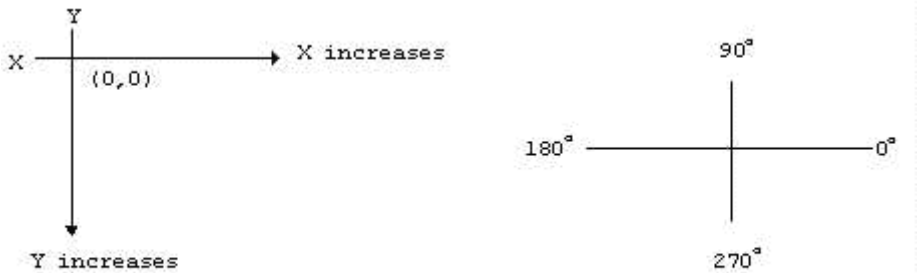
Figure 6



A world consists of cubes.

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.

Figure 7



The coordinate convention used in this paper.
(all angles in degrees)

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

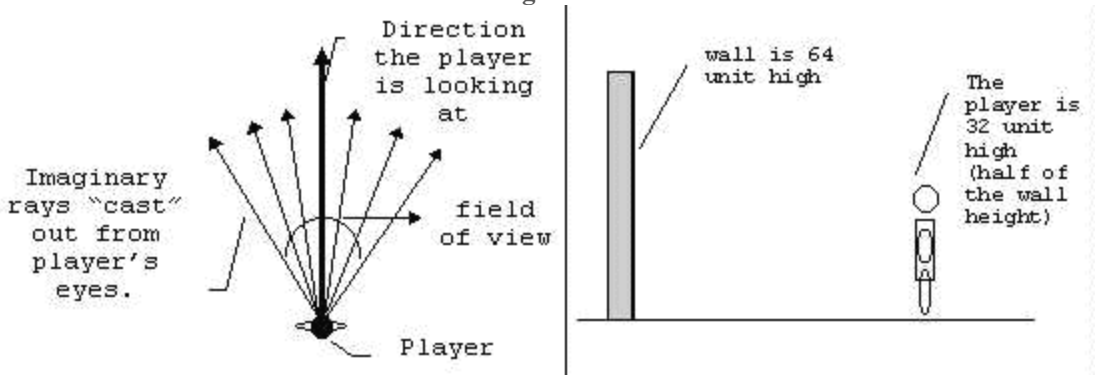
Now that we have the world, we need to define some attributes before we can project and render the world. Specifically, we need to know these attributes:

- 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). 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.

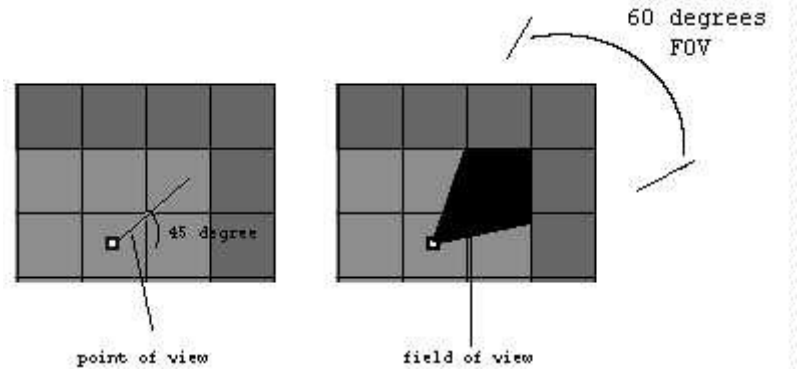
Figure 8



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.

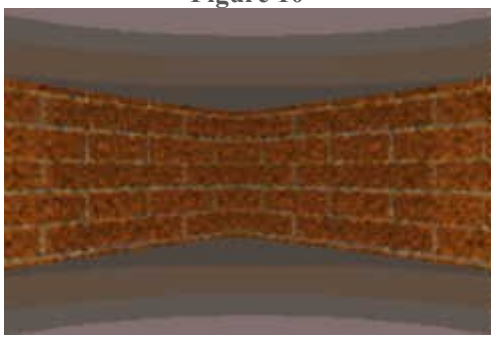
Suppose that the player is put somewhere in the **middle** of **grid coordinate** (1,2) at a viewing angle of 45 degrees relative to the world, then the player's *point of view* and FOV will be like inFigure 9. (One grid consist is 64 x 64 units. Thus, we can also say that the player is in **unit coordinate** (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.



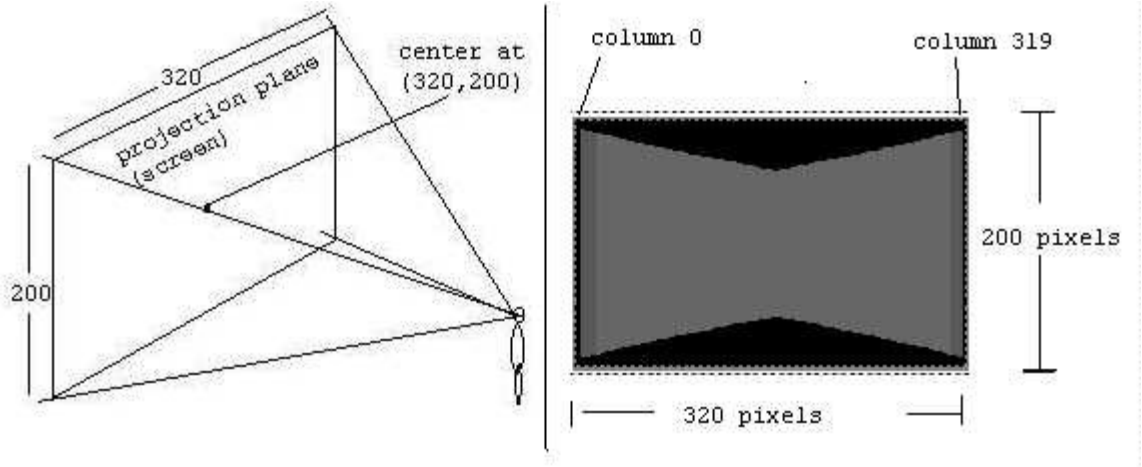
We need to define a projection plane so that we can project what the player sees into the projection plane. A projection plane of 320 units wide and 200 units high is a good choice, since this is the resolution of most VGA video cards. (Video resolution is usually referred in pixels, so think of 1 pixel 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 below.

Figure 10

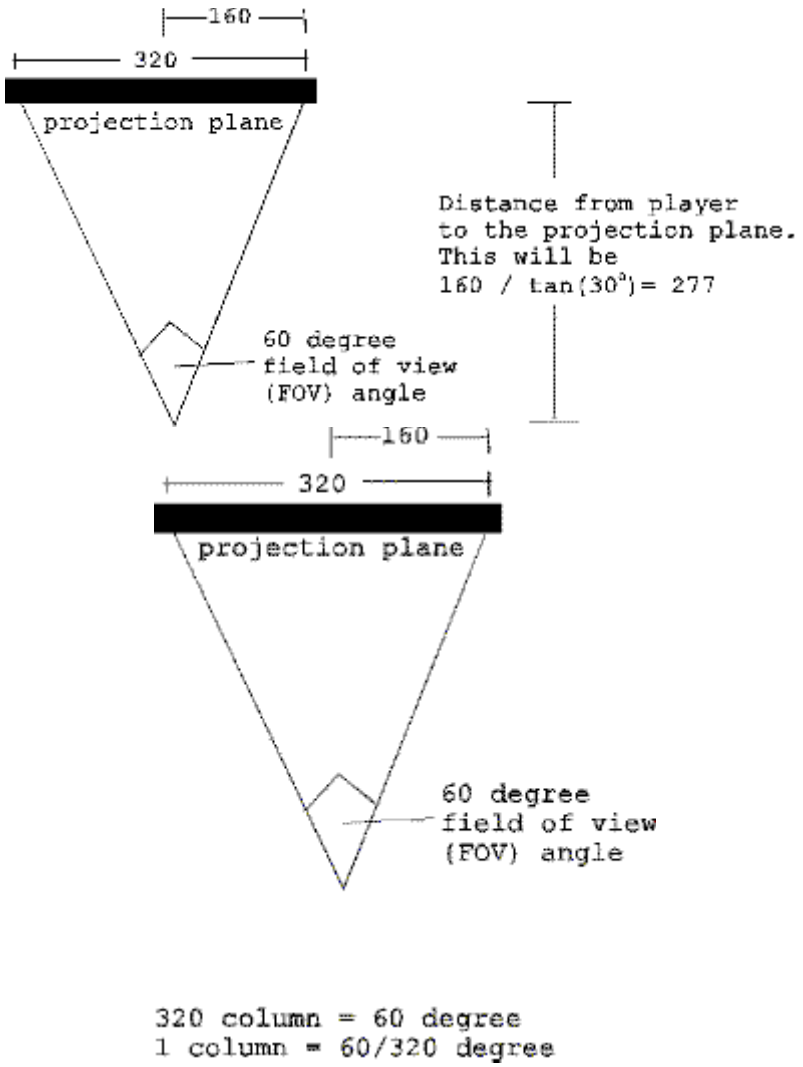


By knowing the ***field of view (FOV)*** and the ***dimension of the projection plane***, we can calculate ***the angle between subsequent rays*** and ***the distance between the player and the projection plane***. These steps are illustrated inFigure 11 (Many books define these last two values arbitarily, 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 (most of these are high school level math, I recommend brushing up on Trigonometry/Pythagorean theorem if you don't understand):



So now we know:

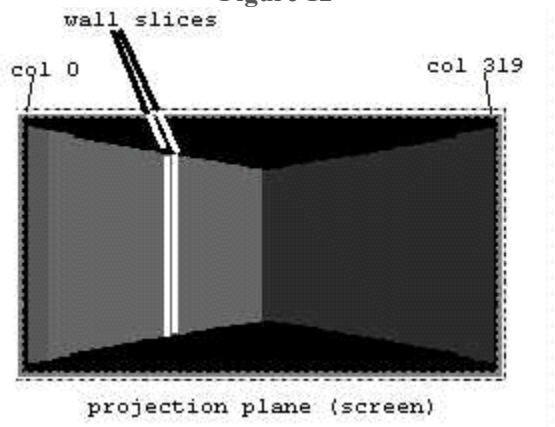
- Dimension of the Projection Plane = 320 x 200 units
- Center of the Projection Plane = (160,100)
- Distance to the Projection Plane = 277 units
- Angle between subsequent rays = 60/320 degrees

(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

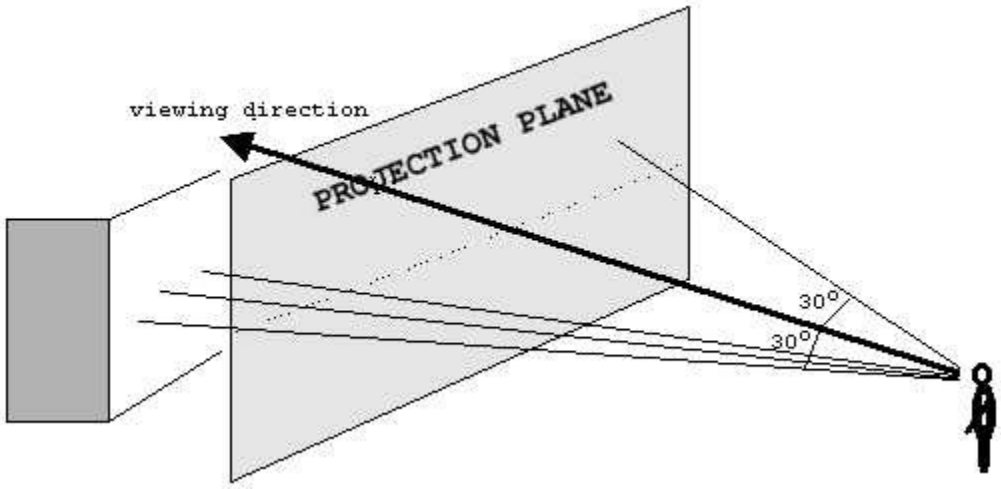
Notice from the previous image (Figure 11), that the wall can be viewed as collection of 320 vertical lines (or 320 wall slices).

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.

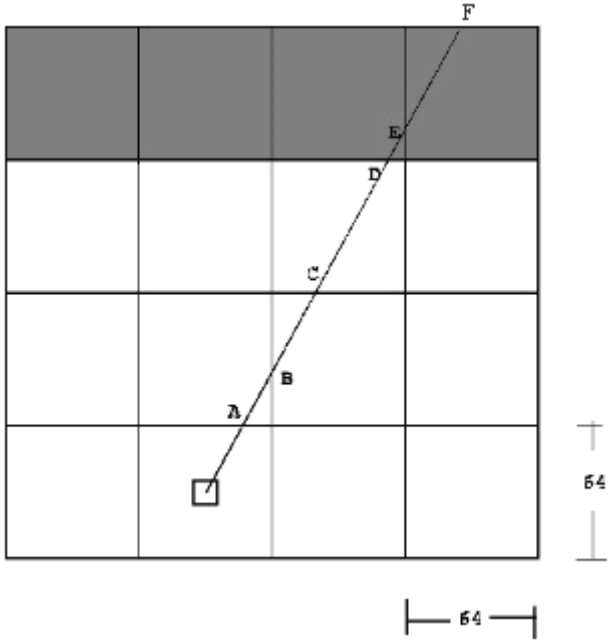


Therefore, to render such scene, we can simply trace 320 rays starting from left to right. This can be done in a loop. The following illustrates these steps:

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.
3. Record the distance to the wall (the distance is equal to the length of the ray).
4. 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).
5. Repeat step 2 and 3 for each subsequent column until all 320 rays are cast.

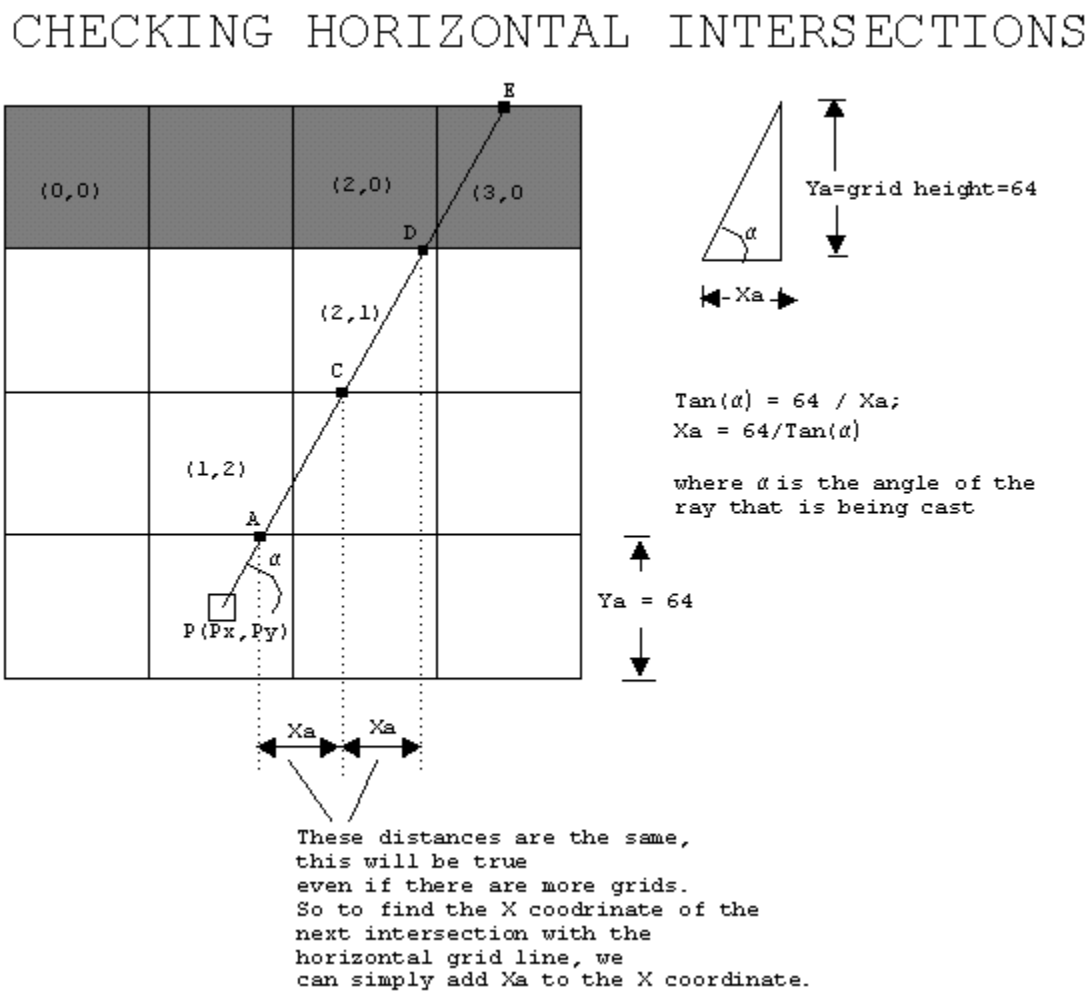
The trick to **step 2A** is that instead of checking each pixels, we only have to check each grid. This is because a wall can only appear on a grid boundary. Consider a ray being traced as in Figure 14. To check whether this ray has hit a wall or not, it is sufficient to check the grid intersection points 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 find walls, 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.

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 the 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), 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).

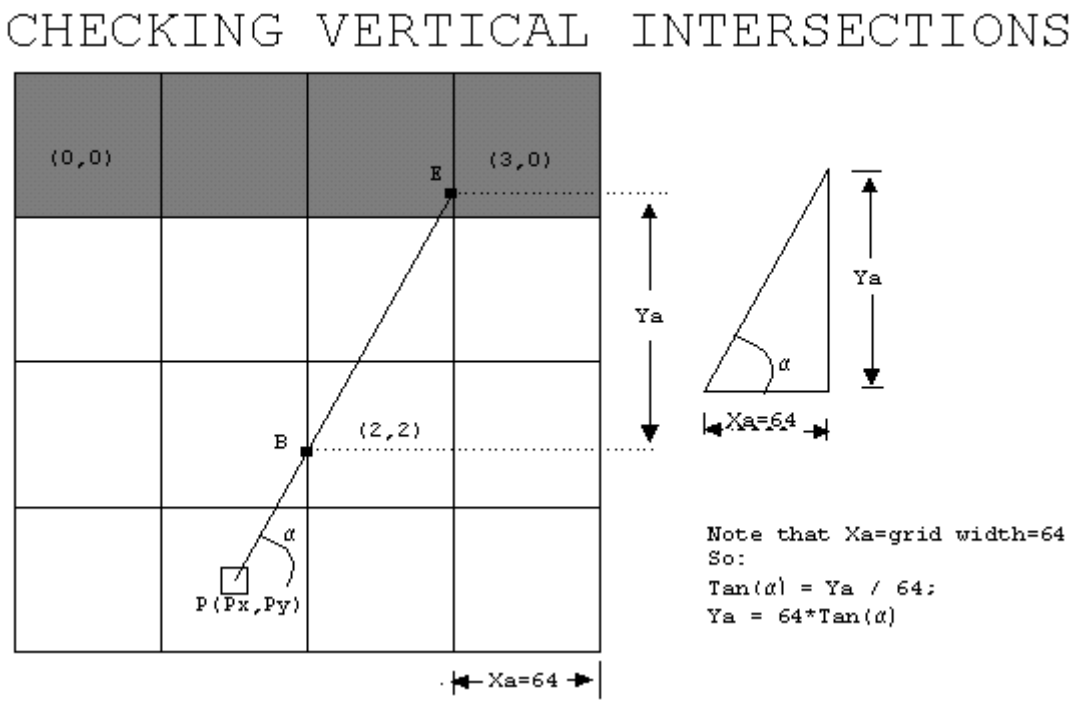
5. Grid (2,1) is checked.
   Again, there is no wall, so the ray is extended
   to D.

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 the 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.

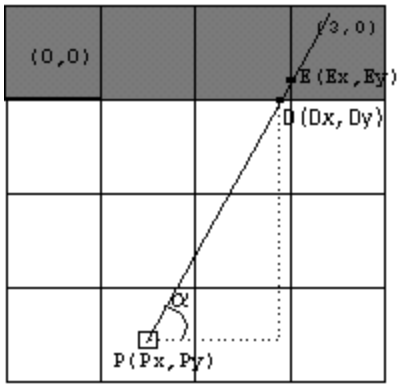
In this example, point D is closer than E. So the wall slice at D (not E) will be drawn.

There's a Java applet example that that illustrated the steps described on this page. I included the source which you can find in the applet link or here

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.



Two ways of finding distance:

$PD=\text{square root}((Px-Dx)^2+(Py-Dy)^2$

$PE=\text{square root}((Px-Ex)^2+(Py-Ey)^2$

or

$PD=ABS(Px-Dx)/\cos(\alpha) = ABS(Py-Dy)/\sin(\alpha)$

$PE=ABS(Px-EX)/\cos(\alpha) = ABS(Py-Ey)/\sin(\alpha)$

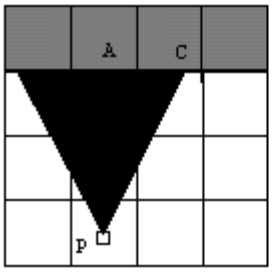
(where ABS=absolute value)

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

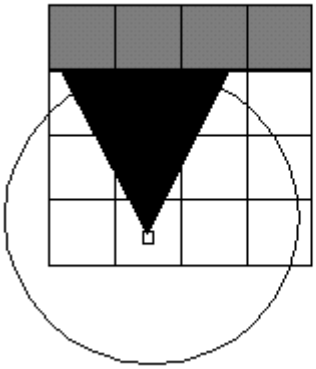
Wall slices that are farther from the center (point A in the figure), appear shorter. This is because rays that are farther from the center of projection have longer distance. For instance, the ray PC is longer than PA. (Like in real life, the farther the wall, the smaller the wall appears. However, the computer screen, unlike the spherical human eyes, is flat. Therefore we must somehow counter this effect.)

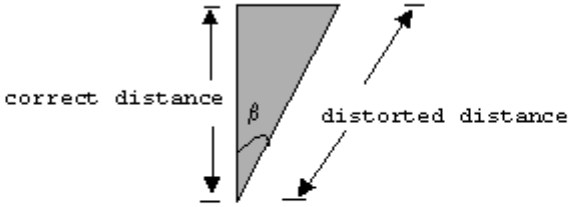


Result of projection without removing the distortion.

Figure 19

How to remove the distortion.





To get the correct distance from ditored distance fist notice that

$\cos(\beta)$ =correct distance/distorted distance

so

correct distance = distorted distance * $\cos(\beta)$

Thus to remove the viewing distortion, the resulting distance obtained from equations in Figure 17must 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.

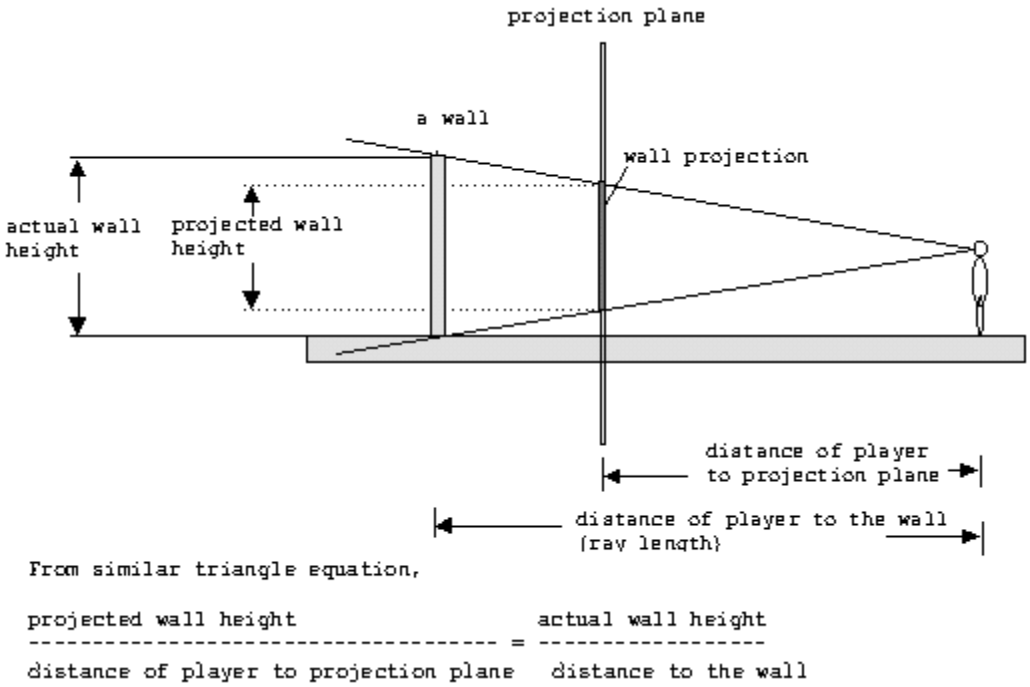
RAY-CASTING STEP 5: DRAWING WALLS

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.



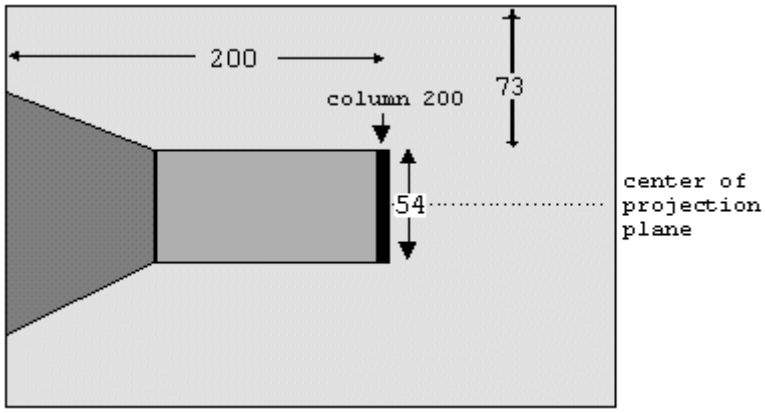
Our world consist cubes, where the dimension of each cube is 64x64x64 units, so the wall height is 64 units. We also already know the distance of the player to the projection plane (which is 277). Thus, the equation can be simplified to:
Projected Slice Height = 64 / Distance to the Slice * 277

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.

Figure 21: A partly rendered view.



DEMO WITH SOURCE CODE: https://permadi.com/tutorial/raycast/demo/1/

TEXTURE MAPPED WALLS

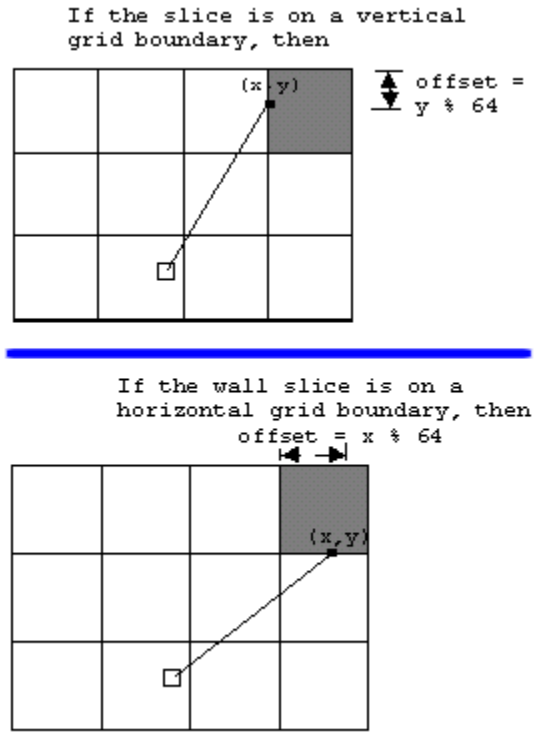
To make the walls more attractive, the walls can be painted with texture (bitmap) using a technique known as texture mapping. (Texture mapping in general refers to a technique of painting a bitmap/texture onto a surface.) For the cube world, we use bitmaps that have the size of 64 by 64 pixels. This 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 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 illustrates the process of finding the offset.

Figure 23: Finding the offset of bitmap.



Link: Some seamless texture map samples to download and use.

SEE DEMO WITH SOURCE CODE: https://permadi.com/tutorial/raycast/demo/2/

DRAWING FLOORS

To draw floors, we can perform floor-casting (floor-casting refers to a techique of rendering floors). Note however, that it would be wasteful to perform floor-casting without texture mapping or shading. In other words, if the floor is not to be textured or shaded (shading will be explored later), then we can simply paint the floor with a solid color and we are done. Keeping that in mind, let us explore what is required to do floor-casting.

There are several ways to do floor-casting. However, all of them use a similar technique. The technique is explained below.

1. Find an intersection with the floor.

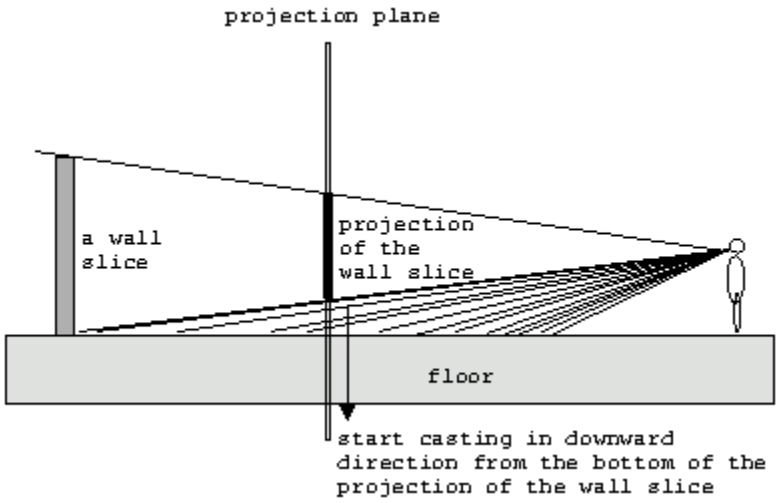
2. Determine the world coordinate of the floor that had been intersected.

3. Calculate the distance between the player and the floor intersection.

4. Project the floor intersection onto the projection plane.

Note that it is not necessary to draw all the floors. We should only draw floors that are not covered by walls. For that reason, we should start the casting from the **bottom** of the wall slices. From the bottom of the slices, we then scan every pixels on the projection plane in **downward**direction (i.e.: cast rays subsequently in downward direction). This time, however, instead of looking for intersection with walls, the ray looks for intersection with the floor.

Figure 24: Floor casting.

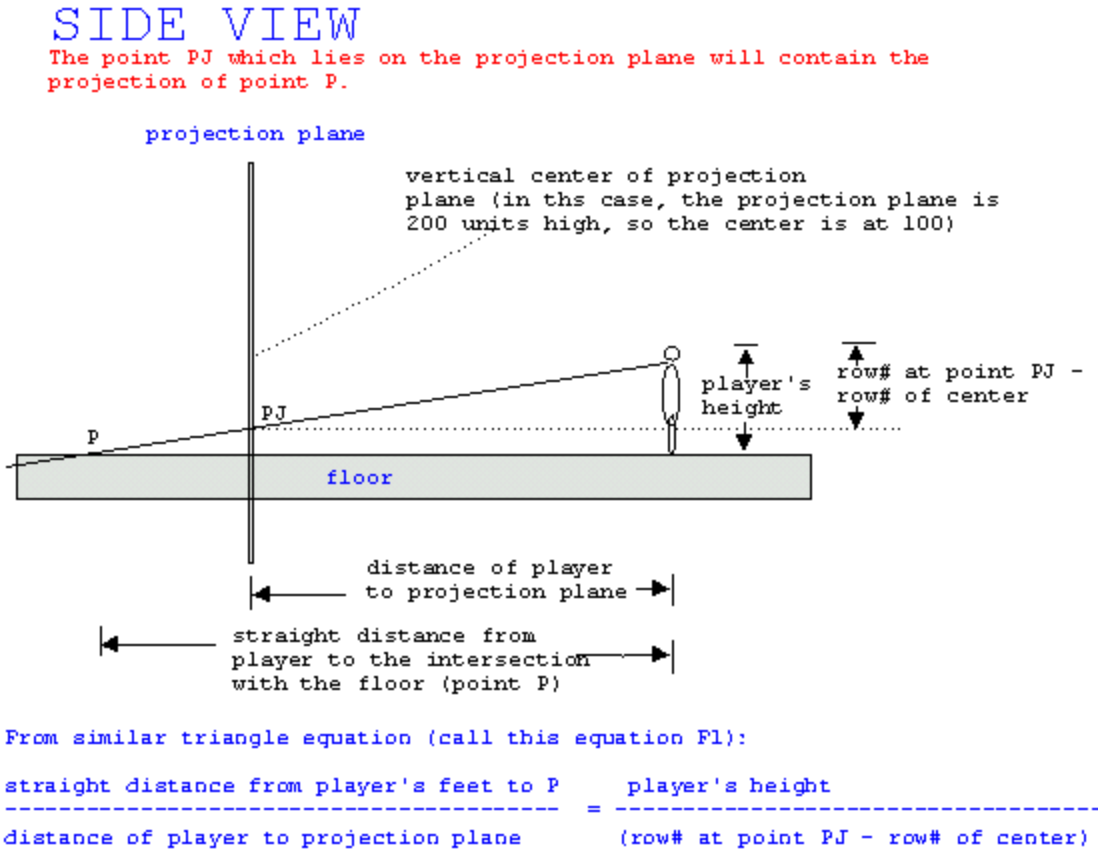


Remember, the we do not need to cast beyond the projection plane. (Ie: cast from the bottom of the wall slice, row by row in downward direction; stop when the bottom of the projection plane is reached.)

FLOOR CASTING (Continued)

The math behind floor-casting is explained in the Figure 25 below.

Figure 25: The math behind floor-casting.

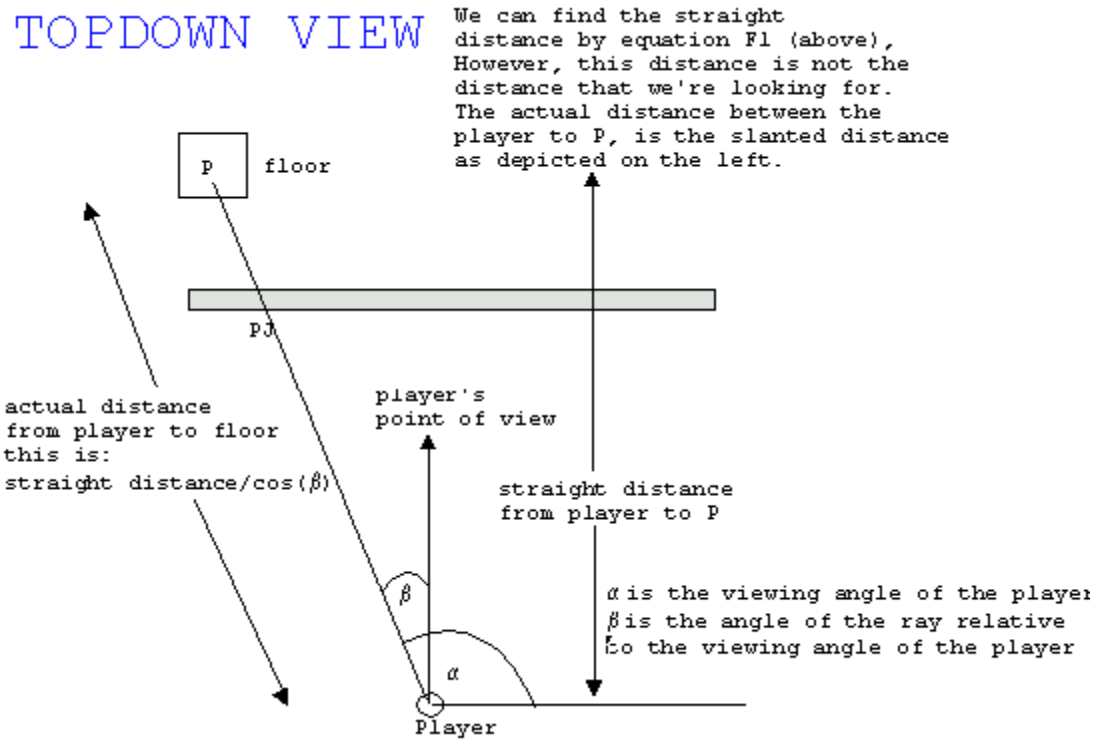


To reiterate, take a look at the illustration while reading these steps:

* Start from the bottom of the wall slice.

1. Take the pixel position (you have this value when you did the wall casting).
2. Draw a line (a ray) from the pixel to the viewers eye.
3. Extends the line so that it intersect the floor.
4. The point where the line “intersect” the floor is the point on the texture map that is being hit by the ray.
5. Take the pixel value of that point on the texture map (see the next figure to see how this can be done) and draw it on the screen.

* Repeat 1-5 until the bottom of the screen is reached.

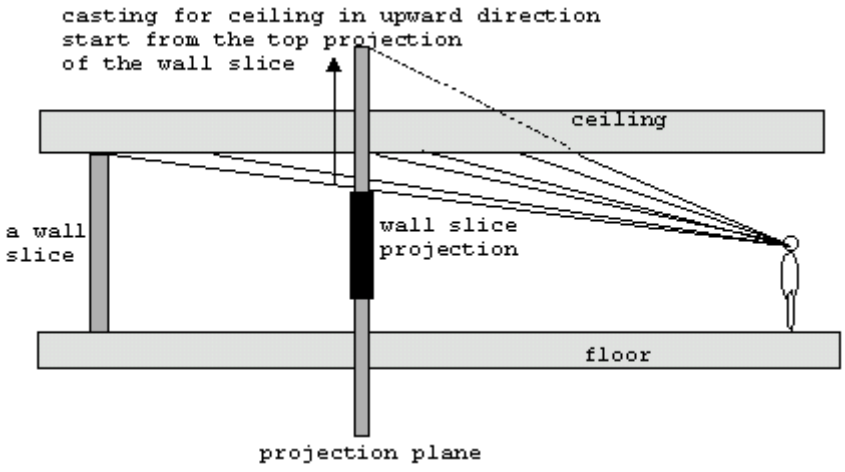


Demo with source code: https://permadi.com/tutorial/raycast/demo/4/

DRAWING CEILINGS

To draw the ceiling, the floor-casting process can be reversed. Instead of tracing rays from the **bottom** of a wall slice in **downward** direction, trace the ray from the **top** of the wall in the **upward** direction. This is actually pretty straightforward once the theory behind floor-casting has been grasped.

Figure 27: Casting to find the ceiling.



Later, we will explain how to simulate the illusion of looking up, looking down, flying, and crouching. If the programmer does not wish to simulate these, it is possible to draw the floor and the ceiling at the same time. This is because the distance of the player's eyes to the floor and ceiling is equal/symetrical. (Floors and ceilings are symmetrical since the player's eyes is exactly at the midpoint between floors and ceilings.)

Demo with source code: https://permadi.com/tutorial/raycast/demo/6/

VARIABLE HEIGHT WALLS

So far, all the walls in our world have the same height. With some innovations, we can actually use walls of different height. This makes the world more interesting as illustrated in the next figure.

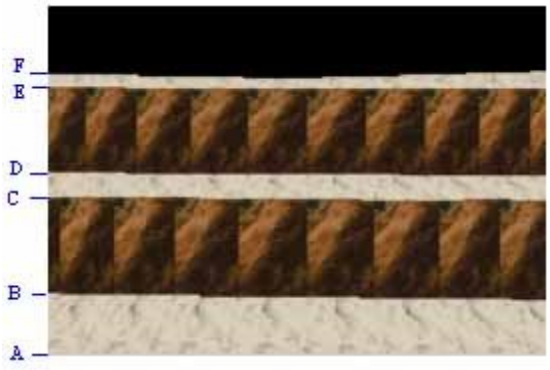
Figure 28: Variable height walls

The easiest way to conceptualize variable height walls is to consider walls as floors. That is, imagine walls as floors that are **raised**. We will need an array to hold the height of each floor grid to make this work. The basic method to render the scene is this:

1. Start from the leftmost column of the projection plane.
2. Find the height of the floor that the player is currently standing on. (Call this CURRENT_HEIGHT.)
3. Cast a ray and check intersections as before.
4. If the ray hits a floor that has different height than the CURRENT_HEIGHT, then that floor is either raised/sunk. (A raised floor is just a wall.)
5. If it is raised, then it will be visible. Project it, and render it. (Figure 30 below illustrates the math behind this.)
6. If it is sunk, then we don't need to project it because it will not be visible.
7. Draw the floor from the point of where the height changes occurs until the point where the **top of the last wall slice** is projected onto. (Initially, the **top of last wall slice** will be the bottom of the projection plane.)
8. Repeat until the ray extends pass the limit of the world map.
9. Repeat step 2 to 8 for all subsequent columns.

To clarify this process, consider the process of rendering the scene in Figure 29 below.

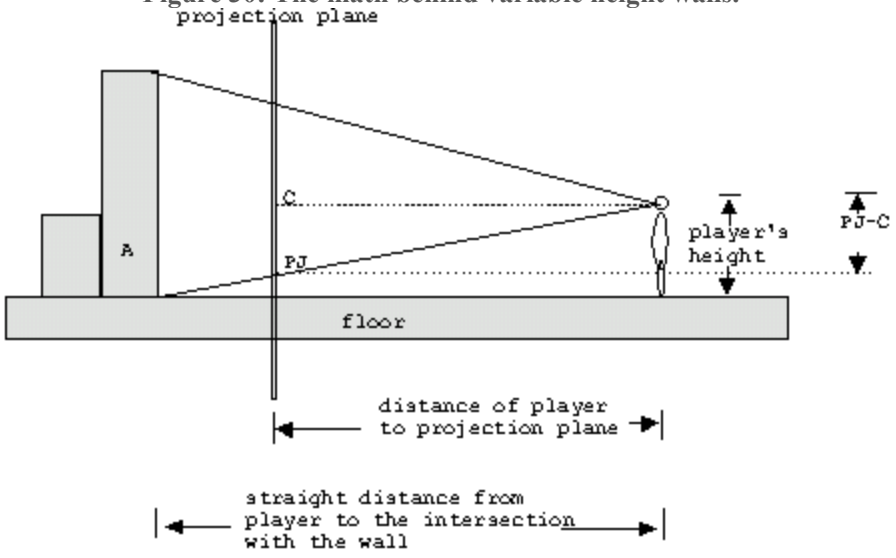
Figure 29



Start by tracing the ray that hits the point A (the bottommost row of the projection plane). As the ray is moved along the projection plane in the upward direction, the wall at point B is hit, so slice BC is drawn. Knowing that there is no height change from A to B, the floor from point A to B is drawn. The ray is then extended. It detects a height change at point D. Therefore, slice DE is drawn. Knowing that there is no height change from point C to D, the floor from point C to D is drawn. The ray is then extended again. At point F, the edge of the map is reached. Since there can be no more height changes, the floor from point F to E is drawn and the process is repeated until the whole screen is rendered.

The main drawback of using variable height walls is that the rendering process will be considerably slower. This is because the rays no longer stops when the closest wall is hit. One way to speed this up is to set a visibility distance and simply ignore anything beyond that distance.

Figure 30: The math behind variable height walls.



From similar triangle equation:

$$\frac{\text{straight distance from player's feet to intersection with wall}}{\text{distance of player to projection plane}} = \frac{\text{player's height}}{\text{(PJ-C)}}$$

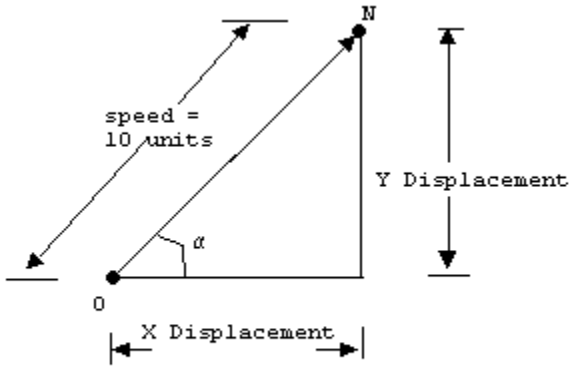
HORIZONTAL MOTION

The player should be able to move at least in three ways: forward, backward, and turning. The player's position is defined by a coordinate and a viewing angle. To allow motion, two more attributes are needed. They are the **player's movement speed**, and the **player's turning speed**. The player's movement speed defines how many units the player should move when he/she is moving forward or backward. The player's turning speed (measured in angle) defines how many angle to be added or subtracted when the player is turning. We will discuss each how we use these attributes to allow motion.

A. Moving forward and backward.

- We define the player's movement speed to be 10 units. (Generally, this can be any number, but the larger number, the less smooth the movement will appear.) The process of finding the x and y displacement is illustrated below. If the player is moving forward, we add the XDisplacement to the current player's X coordinate; and add Ydisplacement to the current player's Y coordinate. If the player is moving backward, we subtract the XDisplacement to the current player's X coordinate; and subtract Ydisplacement to the current player's Y coordinate. (Always check for world/wall boundaries so that the player won't go outside the map or walk through a wall.)

Figure 31: Finding the displacement based on player's speed (in this case, the player's speed in 10 units).

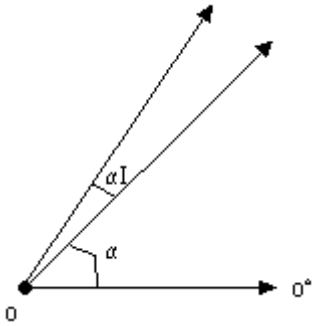



```
XDisplacement = cos(α)*10
YDisplacement = sin(α)*10
```

B. Turning.

The process of turning is very simple to implement. All we need to do is to add or subtract an angle increment (aI) to the current player's viewing angle (wrap around whenever the turn goes to a full circle). Again, larger angle increment will cause the movement appear less smooth.

Figure 32.



SEE DEMO: https://permadi.com/tutorial/raycast/demo/3/

VERTICAL MOTION: LOOKING UP AND DOWN

It is possible to simulate the illusion of looking up and down, as well as flying and crouching on a ray-casting environment. However, note that -and this is important- the trick that is about to be explained in here does not always follow the correct three dimensional projection theories. **Ie: These are tricks, they're not the correct way to do a "realistic" simulation.**

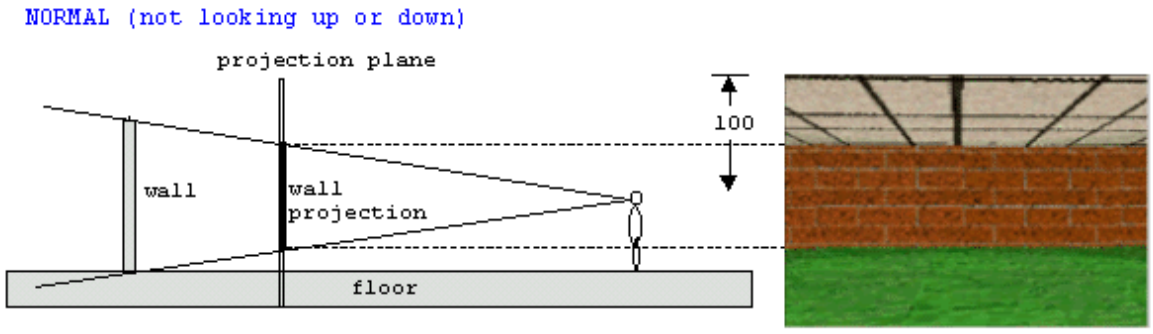
A. Looking up and down.

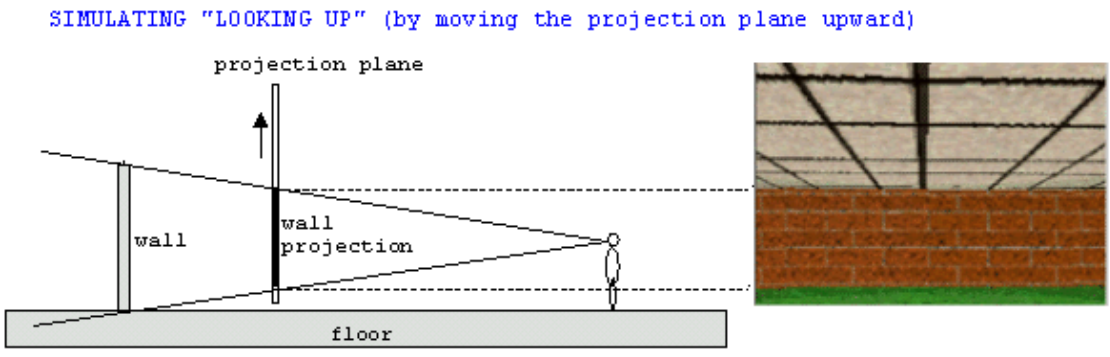
Recall that the projection plane is 200 units high. And up to this point, we always set the vertical center of the projection plane to be exactly in the middle (that is, at point y=100). Thus the midpoint of any wall slice will be drawn at projection point y=100. It turns out that the effect of looking up or down can be simulated simply by changing this value.

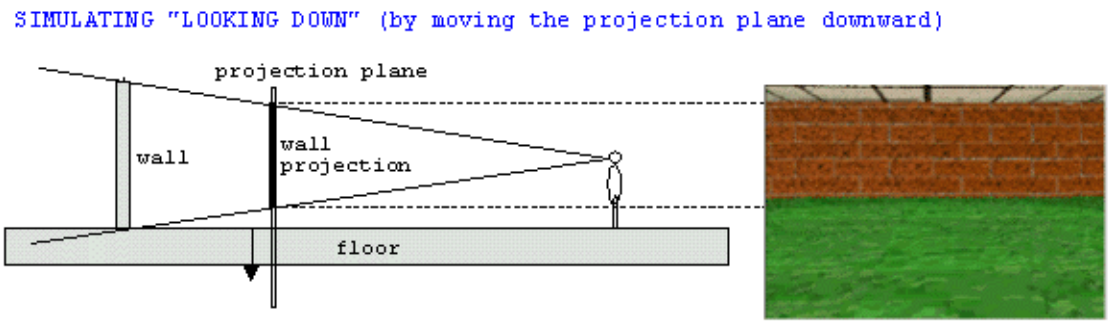
That is, to simulate **looking up**, instead of putting the center of the vertical slice at y=100, we put it at a point where y>100 (this is similar to moving the projection plane upward).

Similarly, to simulate **looking down**, instead of putting the center of the vertical slice at y=100, we put it at a point where y<100 (this is similar to moving the projection plane downward).

And why does this trick work at all? Hopefully, the following illustrations explains it.







If you're confused, imagine holding a mirror with a wall behind you while standing straight. When the mirror is moved up or down, different part of the wall is shown. The mirror is the projection plane. (Take a moment to imagine this before continuing.)

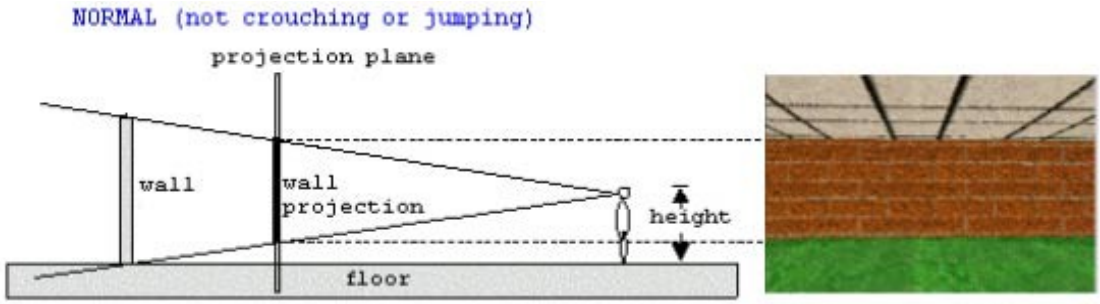
Demo with source code: https://permadi.com/tutorial/raycast/demo/7/

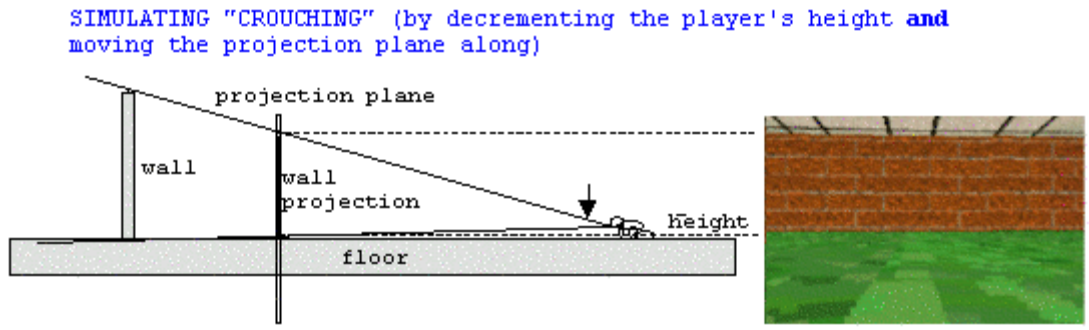
B. Flying and crouching.
Recall that the player's height is set to be 32 units. This means that the player's eyes (imagine the player's eyes are exactly on top of the player's head) are looking straight at the walls at point 32. Since 32 is one half of the walls' height, having the player's height at 32 makes the player's eyes halfway between the floor and the ceiling (see next figure).

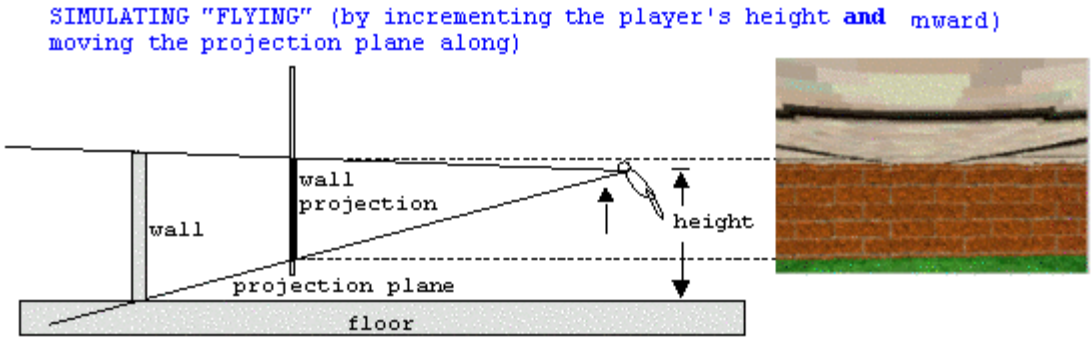
What if we change this value? Surprisingly (or maybe not), the walls will shift either upward or downward depending on whether the player's height is increased or decreased.

Thus, to make the player as if he/she is **flying** (or leaping), we can simply**increase** the player's height. Similarly, to make the player as if she/he is**crouching**, we can **decrease** the player's height. The height should not be allowed to be less than 0 or greater than walls' height, because doing that will make the player go over the ceiling or sink into the floor.

The next figure shows why this method works.







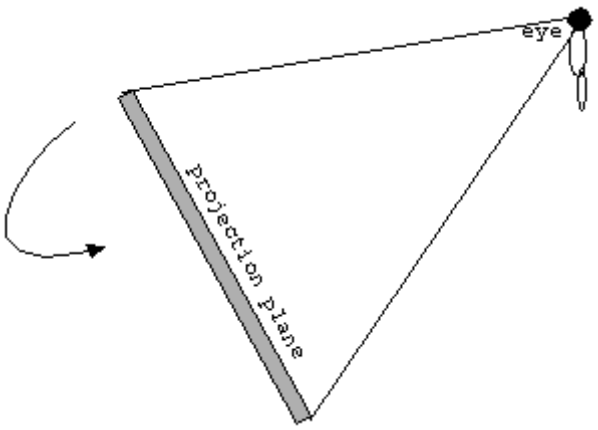
If you're confused, again we use the mirror method to clarify how this works. Imagine that that you are standing straight, holding a mirror on a small room. Stand facing away from the wall. Position the mirror so that it's in front of the eye (i.e.: you do not have to turn your head to see the mirror). Now, imagine what happen if you squat and see what is in the mirror. In the mirror, you should see different part of the wall and more floor area… like 2nd image on this page …. hope you got the idea.

The mirror is the projection plane, and the eye position is the player's height.

There's one counter intuitive aspect of this vertical-motion method, which is this:

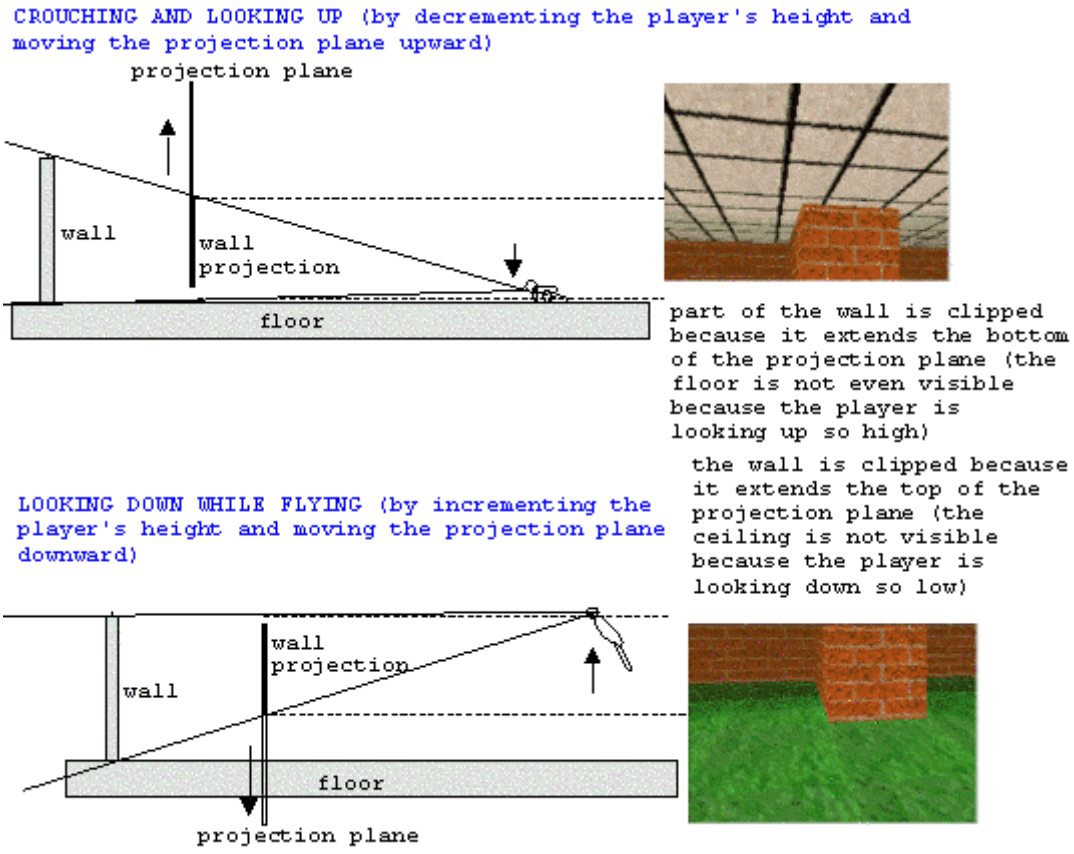
the projection plane must always be perpendicular with the player's eyes. (That is: the projection plane must always be parallel to the walls – they cannot be skewed in any way.) The best way to conceptualize this is to imagine a person "aiming" through a camera lens. The person always aims in forward direction at 90 degrees angle; even when he/she is crouching or standing on top of a table.

The reason for this is that when using this method, we can not skew the projection plane like in the next figure; because if we rotate the projection plane to follow the "normal" eye direction, then the walls will be slanted (no longer parallel with the projection plane); and the rendering process must then take this into account. That means, more complex calculation will be required, and the rendering process will become terribly slow.



This is how a more realistic *looking down* is supposed to happen:
when you look down, you move not only your eye, but **also** your head, so that the perspective is different (compare with the previous 6 pictures). The technique descibed on this page uses tricks earlier to "simulate" this.

C. Combined effects.
The effects explained above can be combined to create even more interesting motions such as illustrated below.



A castle viewed from the sky

SHADING

When an object is farther away from the viewer, the object should appear less/more bright. To accomplish this, a shading effect is needed. But first, we need to know about how colors are represented.

The standard 256 color VGA mode registers contains three numbers between 0 to 63 for every color in the palette which are called the RGB (RedGreenBlue) values. For example, full red color has RGB components of (63,0,0); full green has (0,63,0); and full blue has (0,0,63). Color such as full yellow, can be obtained by mixing full red and full green so that (63,63,0) is yellow.

To change the brightness of the red, green, or blue component of a color, the number representing the color component must be increased or decreased. For instance, to decrease the intensity of a color that have an RGB components (50,10,10) by one half, multiply each component by 0.5. The resulting color will be (25,5,5).

This is quite simple, but how do we know what intensity to use on what distance? The first option is to use an exact light intensity formula which goes something like this:

Intensity = (kI/(d+do))*(N*L)

From a game programmer perspective, this formula is too complicated and will be terribly slow, so we are not going to even bother with it. Our main goal will be to make a shading effect that **looks-right** (or at least reasonable). We do not particularly care whether the formula that we are using is the correct **text-book** formula or not.

(Side note.: For **game** programming, I tend to agree to this principle:
it's better to have something that is fast and look-resonably-right; that to have *something that is exactly-right, but slow*.)

Hence, the following formula is used instead (Lampton 406).

Intensity = Object Intensity/Distance * Multiplier

Here, **Object Intensity**is the intensity that the programmer wish to use (it should be between 0 and 1)**.** This is actually quite simple conceptually. It basically says that as objects gets farther, the intensity of the object gets smaller. **Multiplier** is a number to prevent Intensity from falling off to fast with distance. This calculation can still be expensive in real time, therefore a distance table such as the following table can be used:

Distance to object	Intensity

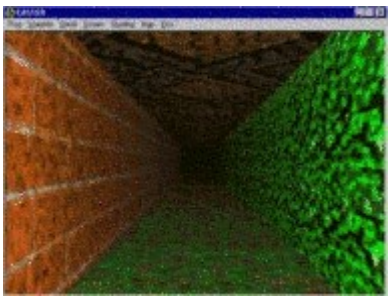
0 to 500	1
501 to 1000	0.75
1001 to 1500	0.50

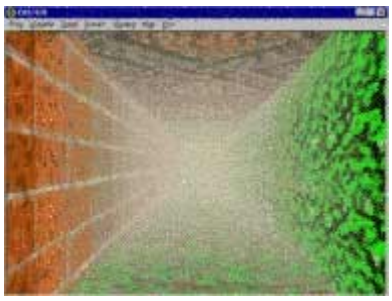
Ray-casting process lends itself nicely here because when we cast a ray, we also obtain the distance to the object to be rendered. In an actual implementation, we need to take into account also the number of available colors. Since most games can only use 256 colors, some acrobatics will be needed to make sure that the palette contains the correct color range. A possible solution for this is to use a color matching algorithm and map the result into an intensity table. When rendering, we simply fetch the correct color value from the appropriate table. (This is quite fast because a particular wall slice will have the same intensity for all of its pixels. So we only have to switch table between wall slices.)

Distance to object	Intensity	Palette Mapping Table Index
0 to 500	1	1
501 to 1000	0.75	2
1001 to 1500	0.50	3
…	…	…

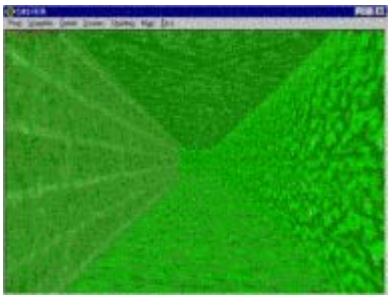
Normally, as the intensity of an object approaches zero, the object will appear darker. However, this does not have to be always the case. We can create in interesting effect, such as fog or underwater effect by altering the "target color." For instance, to create a fog effect, we can make the palette converges to white.

NIGHT EFFECT	FOG EFFECT

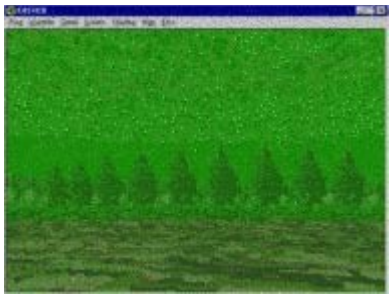




OOZE UNDERWATER EFFECT



UNDERSEA EFFECT



Demo with source code: https://permadi.com/tutorial/raycast/demo/5/

BIBLIOGRAPHY

- Abrash, Michael. **Zen of Graphics Programming**. Scotsdale, AZ: The Coriolis Group, 1995.
- Anderson, Greg, et al. **More Tricks of the Game Programming Gurus**. Indianapolis, Sams Publishing, 1995.
- Finegan, James. "Implementing Games for Windows." **Dr. Dobbs Sourcebook** 239 (1995): 42-47.
- Foley, James D.,et al. **Computer Graphics: Principles and Practice**. 2nd ed. New York: Addison Wesley, 1995.
- Hecker, Chris. "Changing the Rules for Transparent BLTs." **Game Developer** Feb./Mar. 1995: 12-22.
- LaMothe, Andre. **Black Art of 3D Game Programming** . Corte Madera, CA: Waite Group Press, 1995.
- LaMothe, Andre, et al. **Tricks of the Game Programming Gurus**. Indianapolis, Sams Publishing, 1994.
- Lampton, Christopher. **Garden of Imaginations**. Corte Madera, CA: Waite Group Press, 1994.
- Lyons, Eric R. **Black Art of Windows Game Programming**. Corte Madera, CA: Waite Group Press, 1994.
- Myers, Lary L. **Amazing 3-D Games Adventure Set**. Scotsdale, AZ: The Coriolis Group, 1995.
- Perry, Paul. **Multimedia Developer's Guide**. Indianapolis, Sams Publishing, 1994.
- Walnum, Clayton. **Dungeons of Discovery**. Indianapolis: Que, 1995.

NOTES

This document is adapted from a paper written by F. Permadi for a graduate course. The instructor of the class was Professor J.P. Abello. (1996)

THANKS TO

J.P. Abello
Susanto Kolim
Garrett Girod
Doug Ierardi