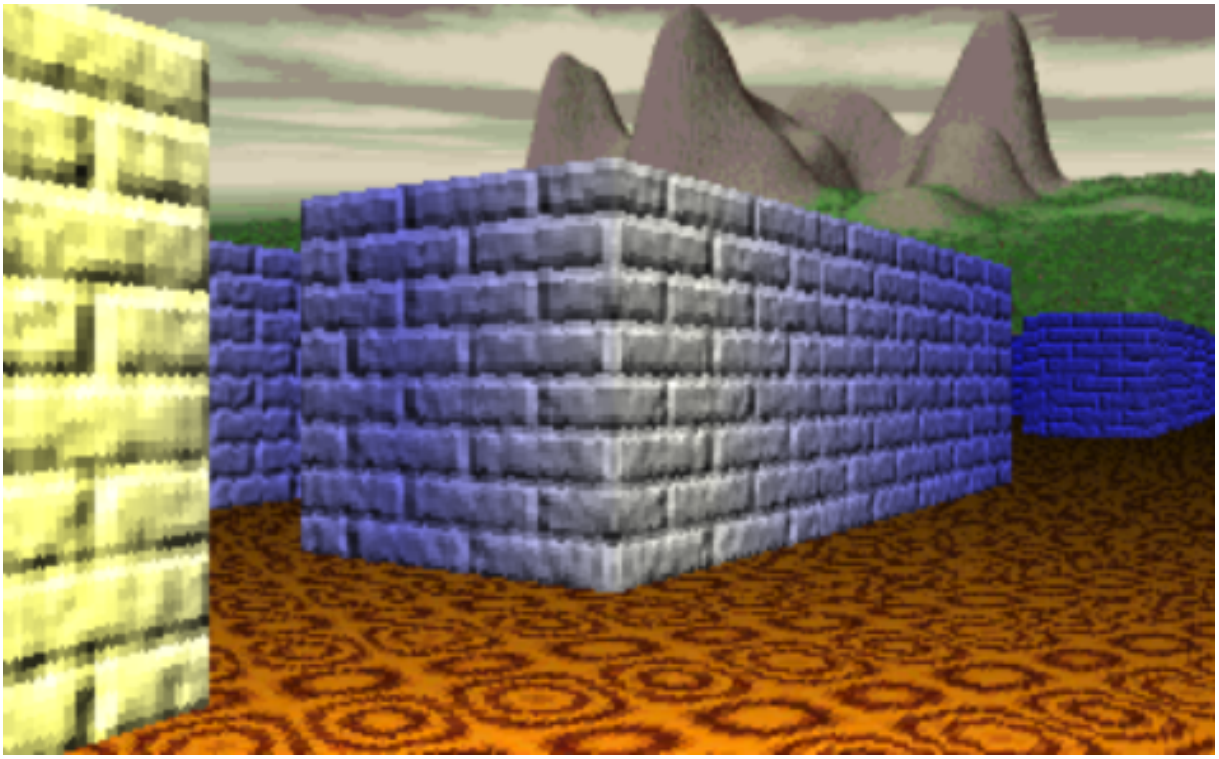
Ray-Casting Tutorial For Game Development

And Other Purposes



by F. Permadi

Sumário

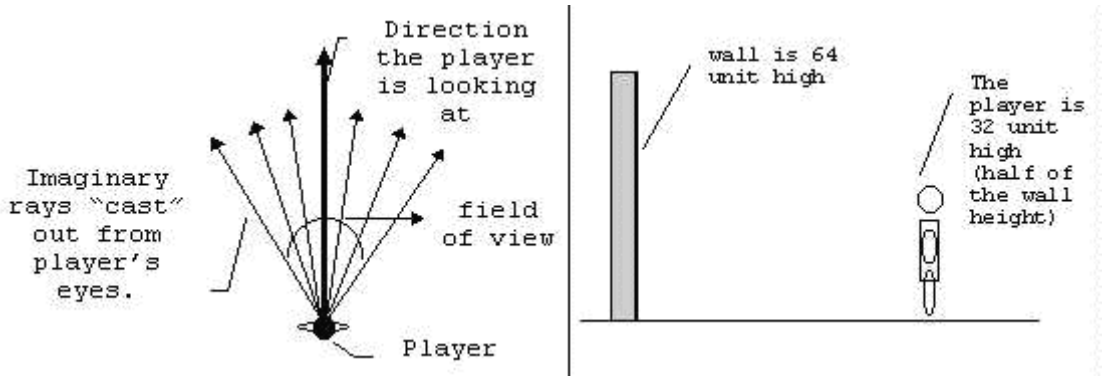
RAY-CASTING STEP 2: DEFINING PROJECTION ATTRIBUTES

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

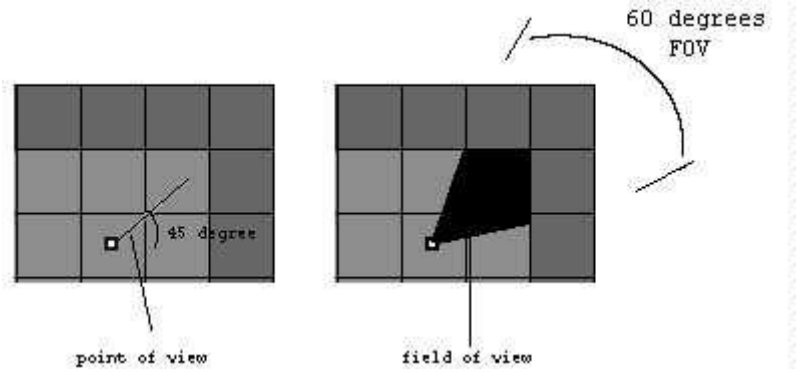


To put the player inside the world, we need to define the player's X coordinate, the player's Y coordinate, and the angle that the player is facing to. These three attributes forms the "point of view" of the player.

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 in Figure 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

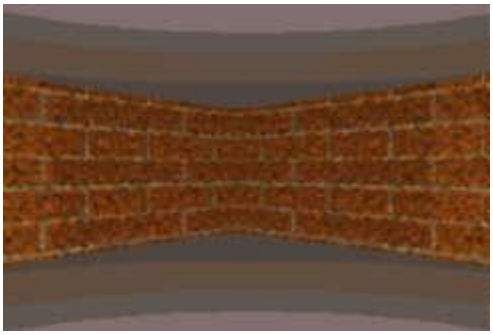
Player is in the middle of grid coordinate (1,2) or unit coordinate (96,160) with a viewing angle of 45 degrees and a field of view of 60 degrees.



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

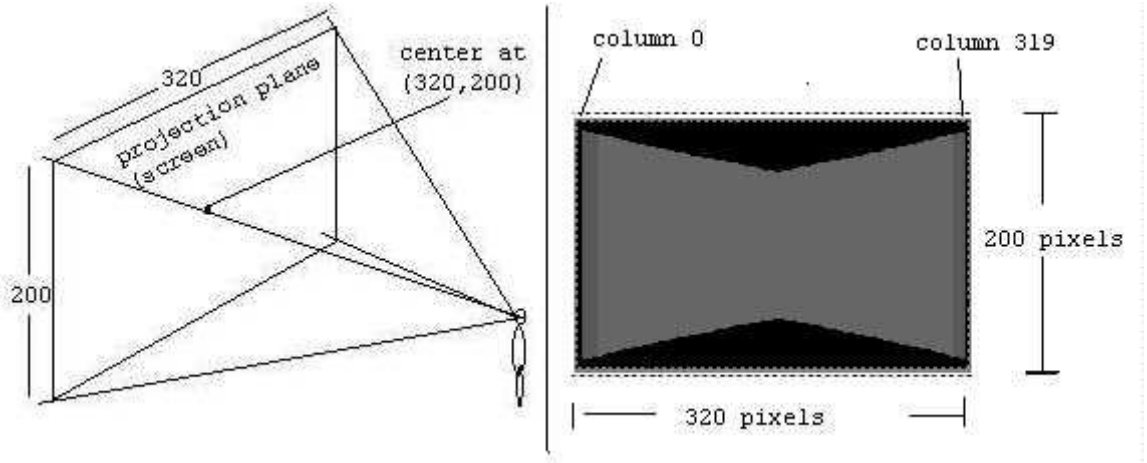


May 17, 1996 By permadi

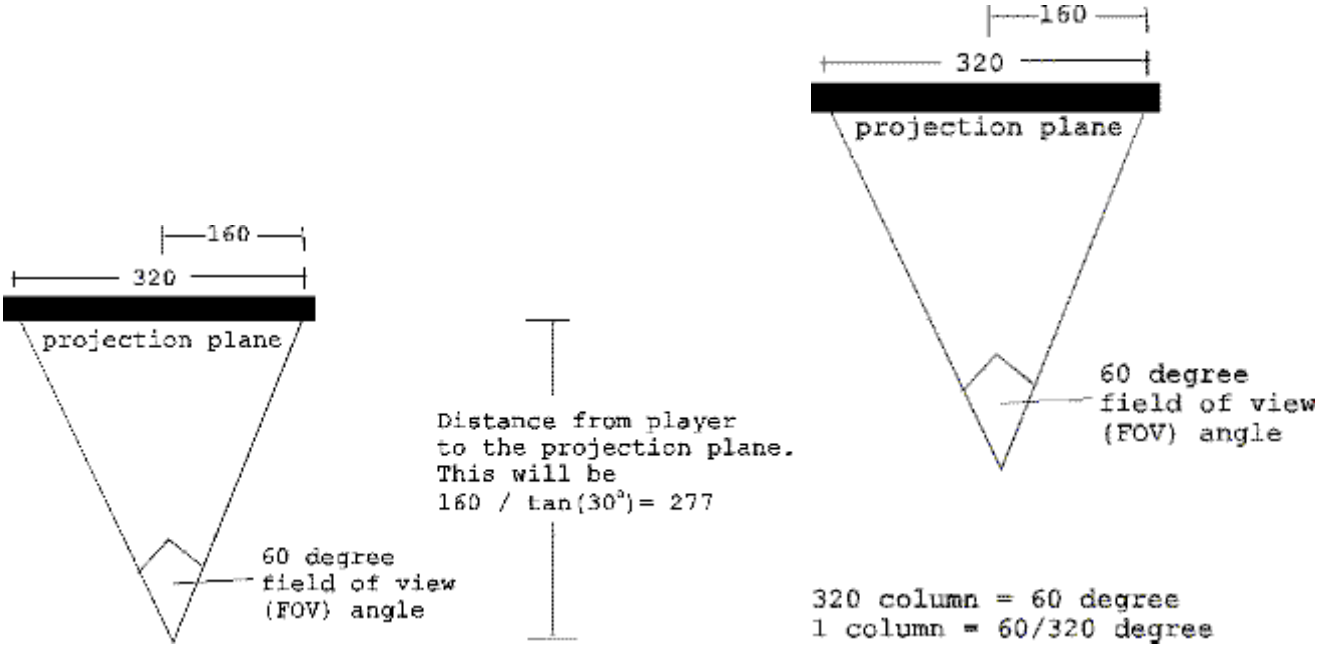
By knowing the ***field of view (FOV)*** and the ***dimension of the projection plane***, we can calculate ***the angle between subsequent rays*** and ***the distance between the player and the projection plane***. These steps are illustrated in[Figure 11](#) (Many books define these last two values 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.)

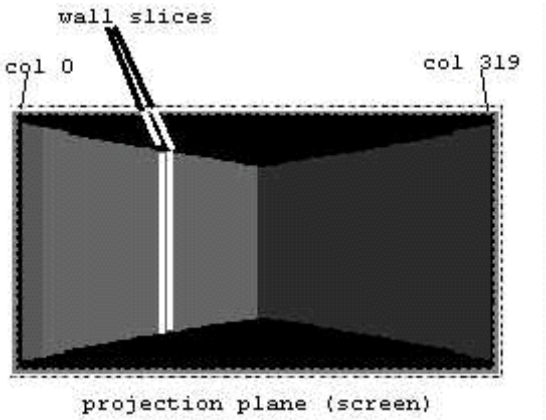
Ray Casting Tutorial – Part 6

May 17, 1996 By permadi

RAY-CASTING STEP 3: FINDING WALLS

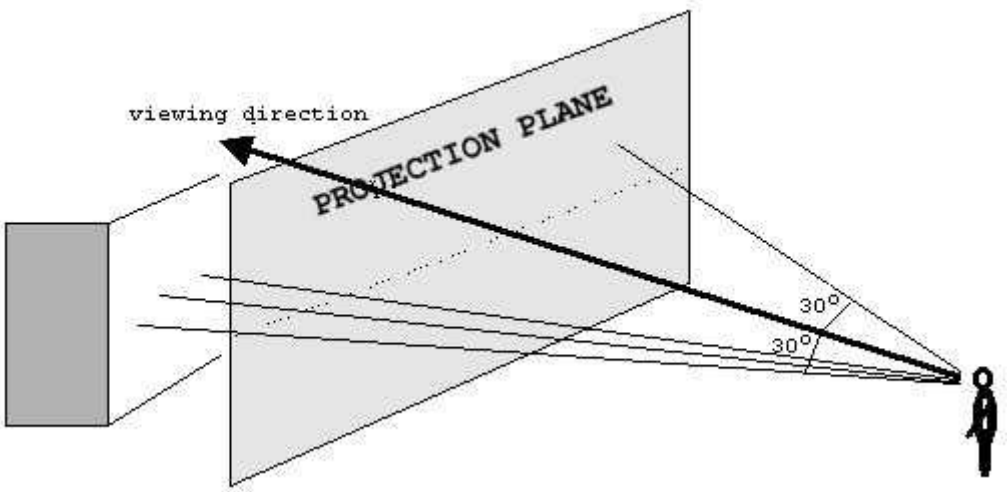
Notice from the previous image (Figure 11), that the wall can be viewed as collection of 320 vertical lines (or 320 wall slices).

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:
    1. 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.)
    2. 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.

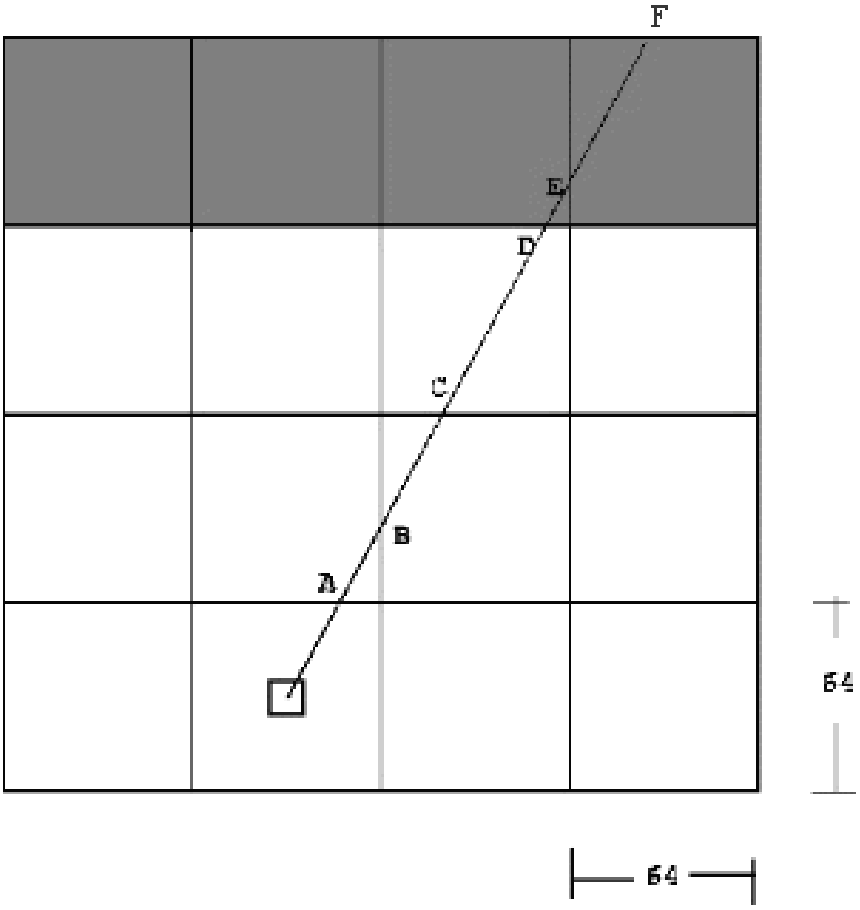


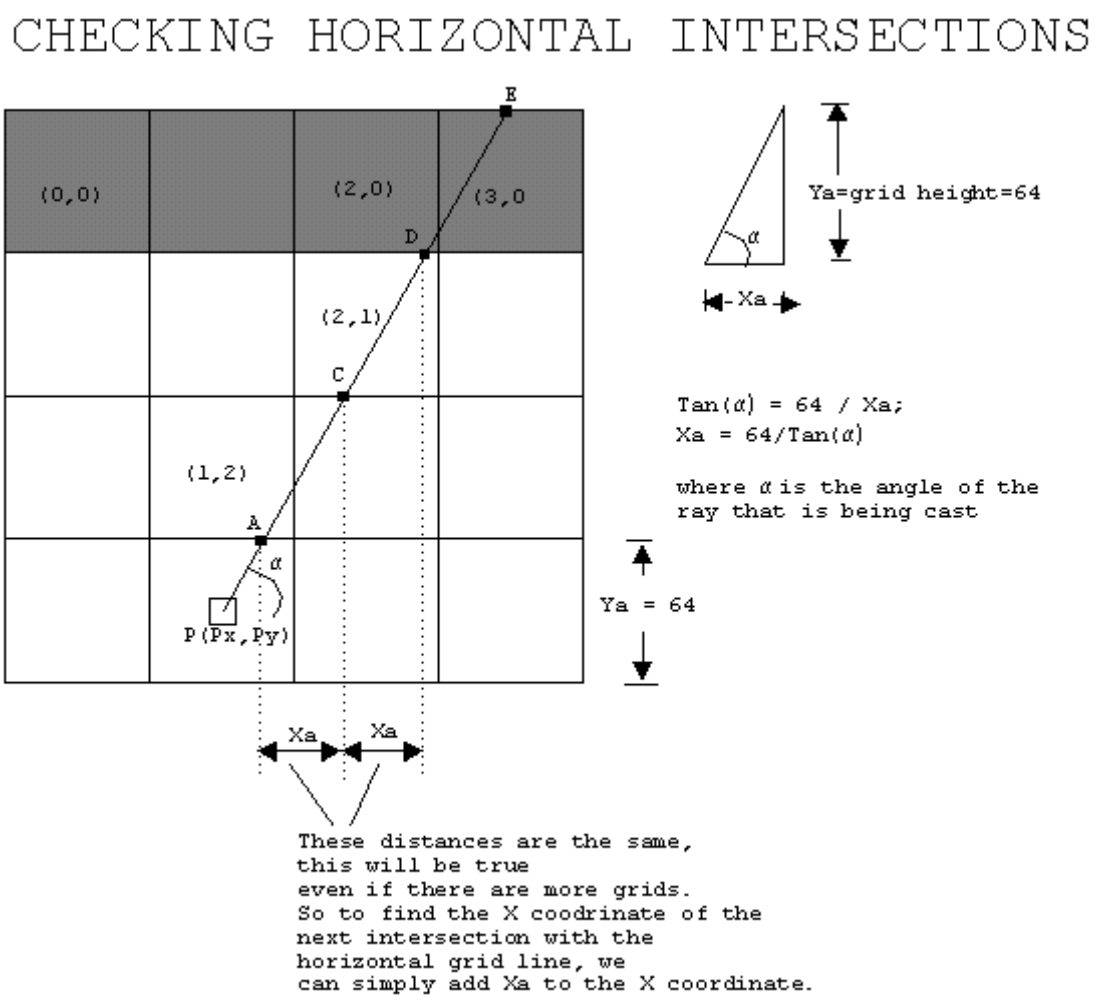
Figure 14: This ray intersects the grids at points A,B,C,D,E, and F.

Ray Casting Tutorial – Part 7

May 17, 1996 By permadi

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:

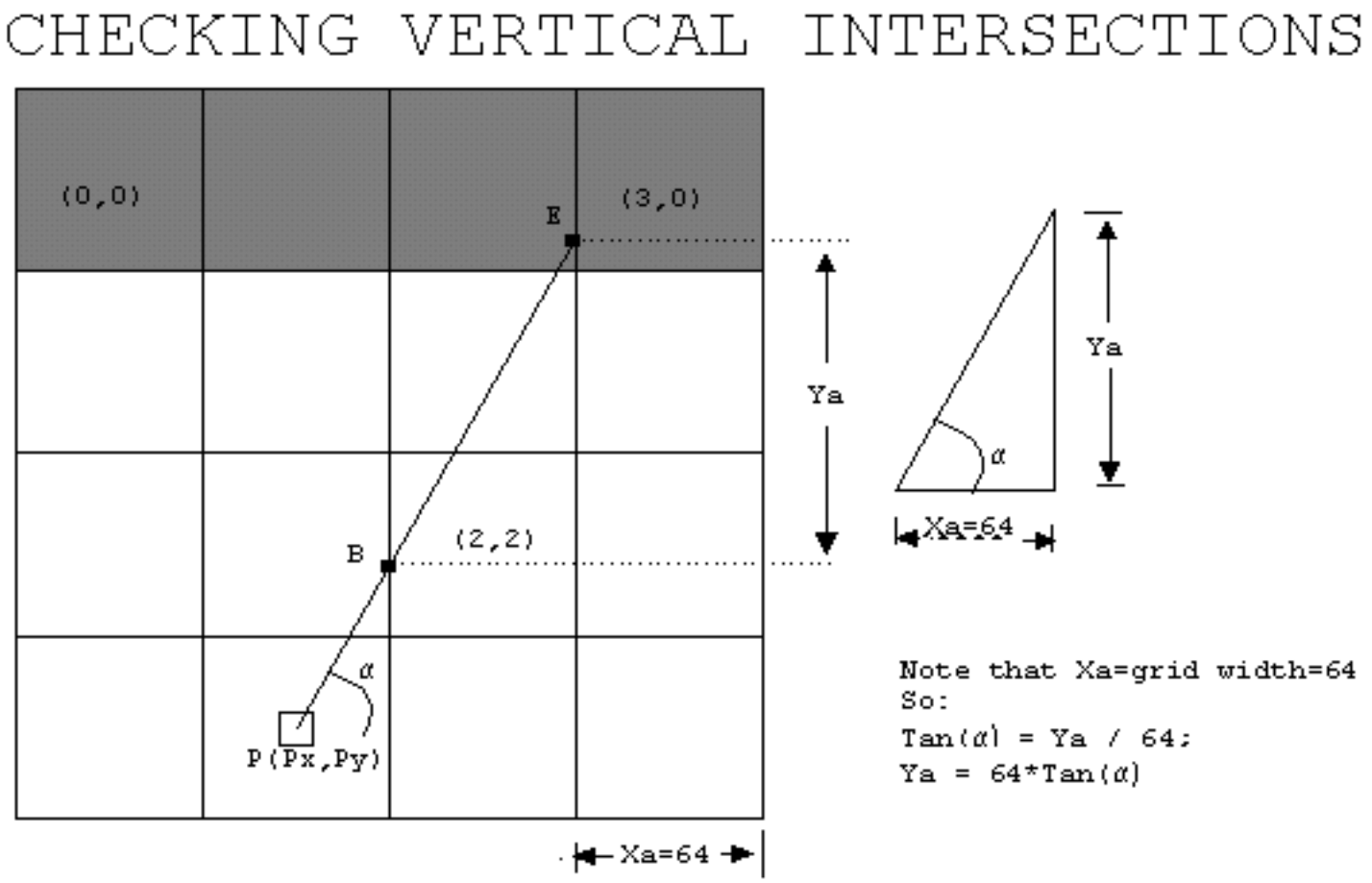
Note: remember the Cartesian coordinate is increasing downward (as in page 3), and any fractional values will be rounded down.

```
======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.

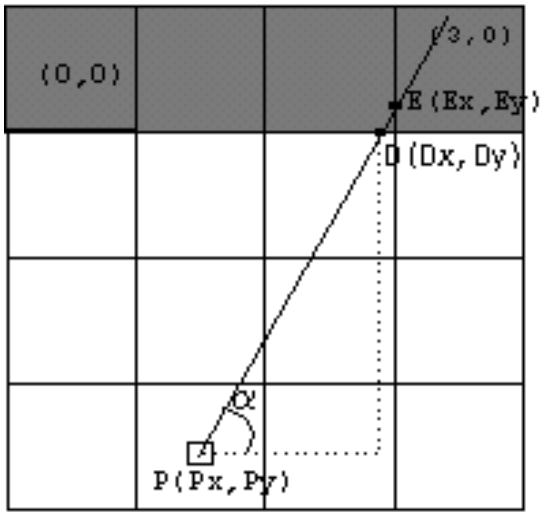
In this example, point D is closer than E. So the wall slice at D (not E) will be drawn.

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)

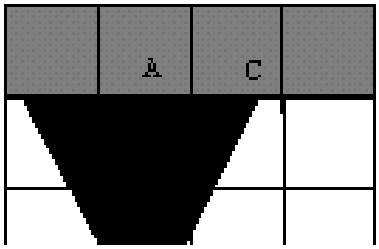
The sine or cosine functions are cheaper to implement because they can be pre-computed and put into tables. This can be done because ALPHA (player's POV) has to be between 0 to 360 degrees, so the number of possibilities are limited (the square root method has a virtually unlimited possible values for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

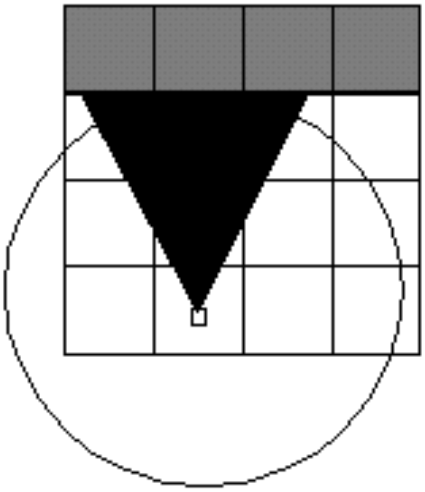
Figure 18

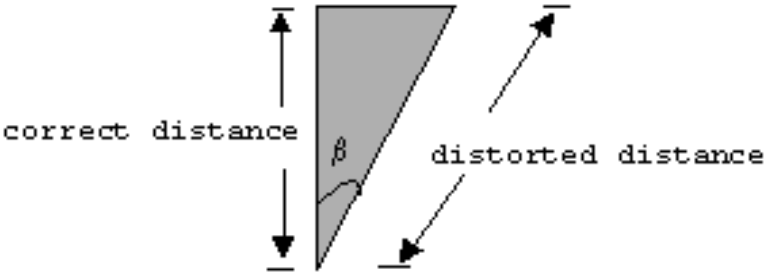
Wall slices that are farther from the center (point A in the figure), appear shorter. This is because rays that are farther from the center of projection have longer distance. For instance, the ray PC is longer than PA. (Like in real life, the farther the wall, the smaller the wall appears. However,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



Result of projection without removing the distortion.

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 * $\cos(\beta)$

Figure 19

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:

$$\text{Projected Slice Height} = \frac{\text{Actual Slice Height}}{\text{Distance to the Slice}} * \text{Distance to Projection Plane}$$

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

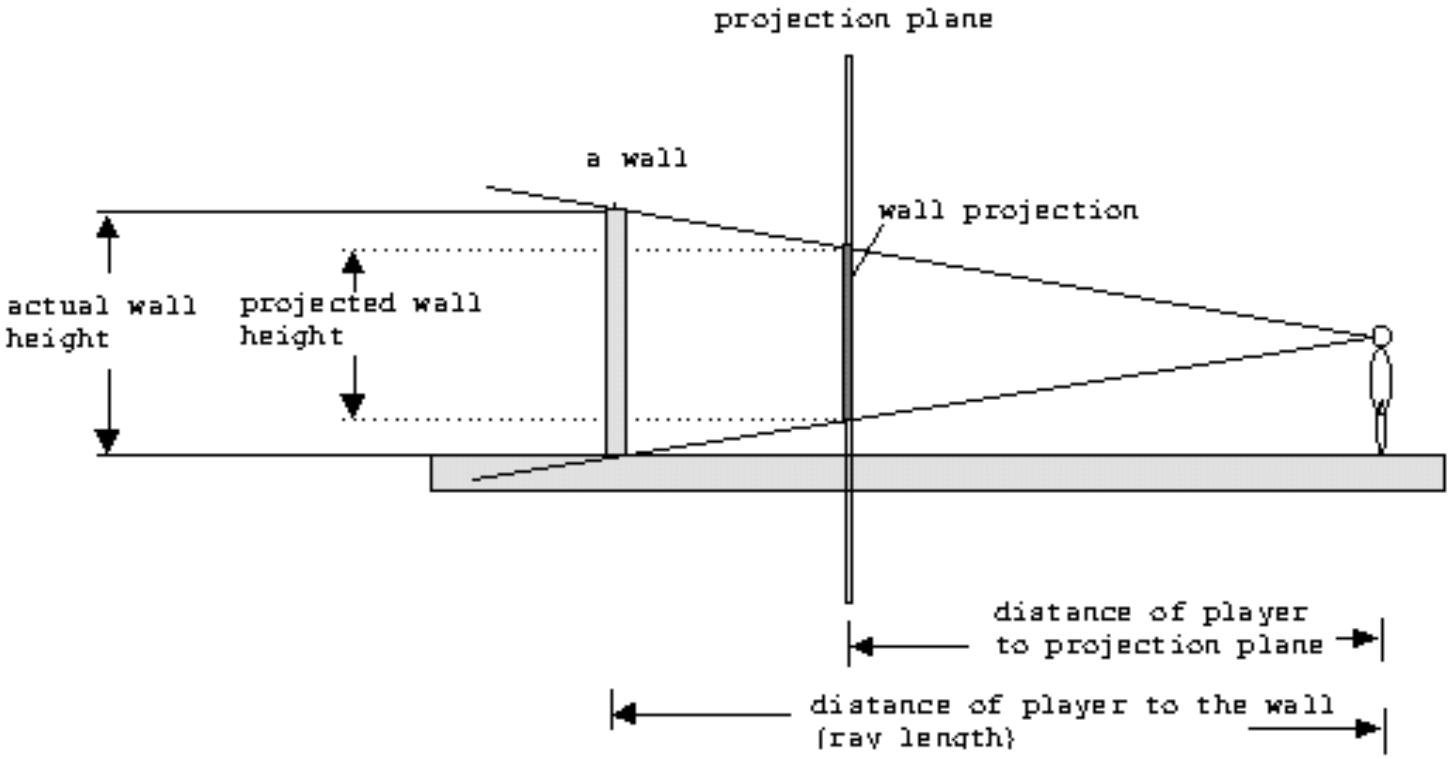
Figure 20: The math behind wall scaling.

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).



From similar triangle equation,

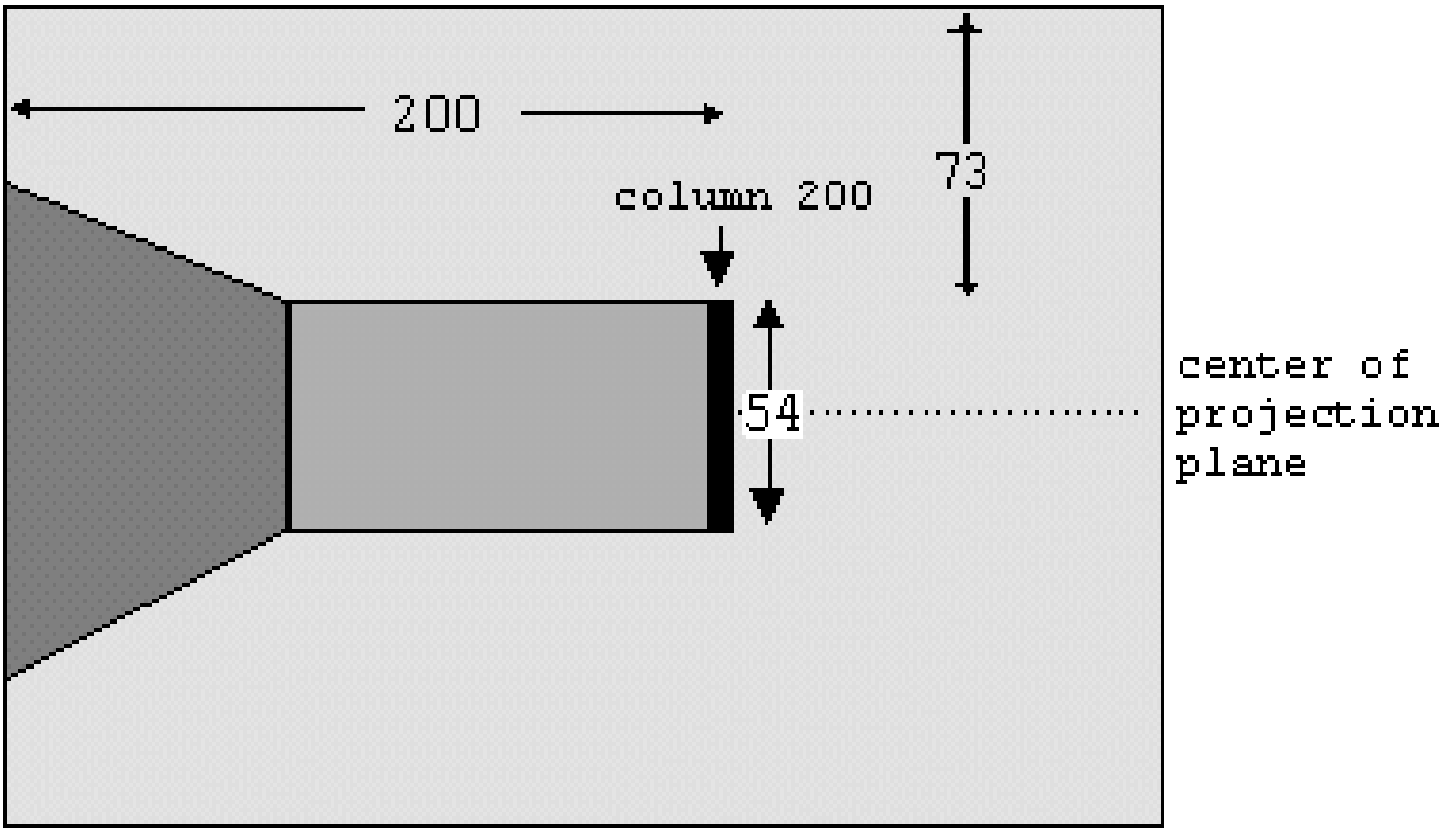
$$\frac{\text{projected wall height}}{\text{distance of player to projection plane}} = \frac{\text{actual wall height}}{\text{distance to the wall}}$$

- 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/

This document explores the fundamental theory behind ray-casting, a pseudo 3-dimensional rendering technique that are very popular in game development arena in the 90s. In general, this document does not bother with implementation and coding detail. The discussion will be mainly about concepts, the implementation is up to the reader. For a casual reader, the knowledge of the *Pythagorean theorem* and knowledge of high-school level math are assumed.

This document was written in the year of 1996, and although ray-casting has been supplanted by newer and more powerful techniques (and hardware!), the reader can hopefully still benefit from the technique.