Lode's Computer Graphics Tutorial

Raycasting

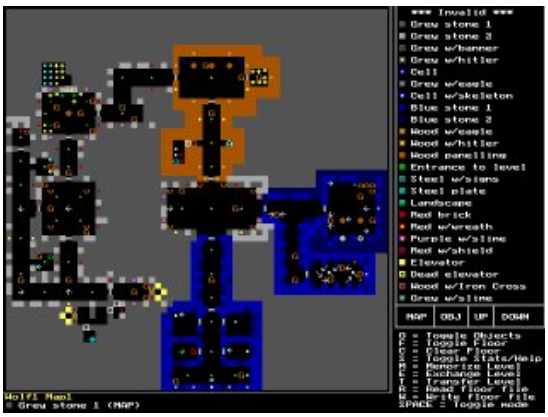
Table of Contents

Back to Index

Introduction

Raycasting is a rendering technique to create a 3D perspective in a 2D map. Back when computers were slower it wasn't possible to run real 3D engines in realtime, and raycasting was the first solution. Raycasting can go very fast, because only a calculation has to be done for every vertical line of the screen. The most well known game that used this technique, is of course Wolfenstein 3D.

The raycasting engine of Wolfenstein 3D was very limited, allowing it to run on a even a 286 computer: all the walls have the same height and are orthogonal squares on a 2D grid, as can be seen in this screenshot from a mapeditor for Wolf3D:



Things like stairs, jumping or height differences are impossible to make with this engine. Later games such as Doom and Duke Nukem 3D also used raycasting, but much more advanced engines that allowed sloped walls, different heights, textured floors and ceilings, transparent walls, etc... The

sprites (enemies, objects and goodies) are 2D images, but sprites aren't discussed in this tutorial for now.

Ray*casting* is not the same as ray*tracing*! Raycasting is a fast semi-3D technique that works in realtime even on 4MHz graphical calculators, while raytracing is a realistic rendering technique that supports reflections and shadows in true 3D scenes, and only recently computers became fast enough to do it in realtime for reasonably high resolutions and complex scenes.

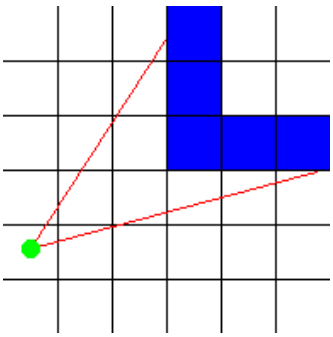
The code of the untextured and textured raycasters is given in this document completely, but it's quite long, you can also download the code instead:

raycaster_flat.cpp
raycaster_textured.cpp

The Basic Idea

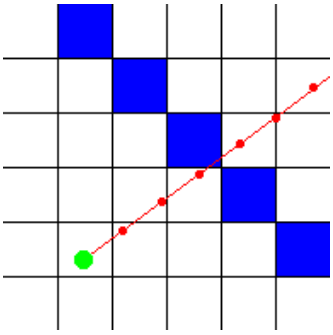
The basic idea of raycasting is as follows: the map is a 2D square grid, and each square can either be 0 (= no wall), or a positive value (= a wall with a certain color or texture).

For every x of the screen (i.e. for every vertical stripe of the screen), send out a ray that starts at the player location and with a direction that depends on both the player's looking direction, and the x-coordinate of the screen. Then, let this ray move forward on the 2D map, until it hits a map square that is a wall. If it hit a wall, calculate the distance of this hit point to the player, and use this distance to calculate how high this wall has to be drawn on the screen: the further away the wall, the smaller it's on screen, and the closer, the higher it appears to be. These are all 2D calculations. This image shows a top down overview of two such rays (red) that start at the player (green dot) and hit blue walls:

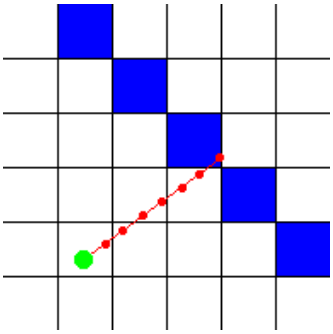


To find the first wall that a ray encounters on its way, you have to let it start at the player's position, and then all the time, check whether or not the ray is inside a wall. If it's inside a wall (hit), then the loop can stop, calculate the distance, and draw the wall with the correct height. If the ray position is not in a wall, you have to trace it further: add a certain value to its position, in the direction of the direction of this ray, and for this new position, again check if it's inside a wall or not. Keep doing this until finally a wall is hit.

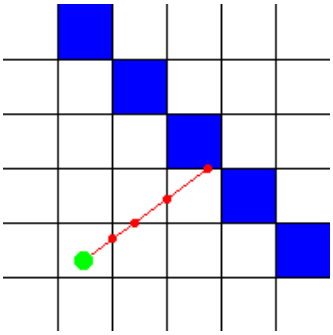
A human can immediatly see where the ray hits the wall, but it's impossible to find which square the ray hits immediatly with a single formula, because a computer can only check a finite number of positions on the ray. Many raycasters add a constant value to the ray each step, but then there's a chance that it may miss a wall! For example, with this red ray, its position was checked at every red spot:



As you can see, the ray goes straight through the blue wall, but the computer didn't detect this, because it only checked at the positions with the red dots. The more positions you check, the smaller the chance that the computer won't detect a wall, but the more calculations are needed. Here the step distance was halved, so now he detects that the ray went through a wall, though the position isn't completely correct:



For infinite precision with this method, an infinitely small step size, and thus an infinite number of calculations would be needed! That's pretty bad, but luckily, there's a better method that requires only very few calculations and yet will detect every wall: the idea is to check at every side of a wall the ray will encounter. We give each square width 1, so each side of a wall is an integer value and the places in between have a value after the point. Now the step size isn't constant, it depends on the distance to the next side:

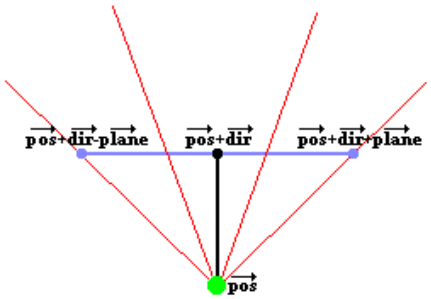


As you can see on the image above, the ray hits the wall exactly where we want it. In the way presented in this tutorial, an algorithm is used that's based on DDA or "Digital Differential Analysis". DDA is a fast algorithm typically used on square grids to find which squares a line hits (for example to draw a line on a screen, which is a grid of square pixels). So we can also use it to find which squares of the map our ray hits, and stop the algorithm once a square that is a wall is hit.

Some raytracers work with Euclidean angles to represent the direction of the player and the rays, and determinate the Field Of View with another angle. I found however that it's much easier to work with vectors and a camera instead: the position of the player is always a vector (an x and a y coordinate), but now, we make the direction a vector as well: so the direction is now determinated by two values: the x and y coordinate of the direction. A direction vector can be seen as follows: if you draw a line in the direction the player looks, through the position of the player, then every point of

the line is the sum of the position of the player, and a multiple of the direction vector. The length of a direction vector doesn't really matter, only its direction. Multiplying x and y by the same value changes the length but keeps the same direction.

This method with vectors also requires an extra vector, which is the camera plane vector. In a true 3D engine, there's also a camera plane, and there this plane is really a 3D plane so two vectors (u and v) are required to represent it. Raycasting happens in a 2D map however, so here the camera plane isn't really a plane, but a line, and is represented with a single vector. The camera plane should always be perpendicular on the direction vector. The camera plane represents the surface of the computer screen, while the direction vector is perpendicular on it and points inside the screen. The position of the player, which is a single point, is a point in front of the camera plane. A certain ray of a certain x-coordinate of the screen, is then the ray that starts at this player position, and goes through that position on the screen or thus the camera plane.

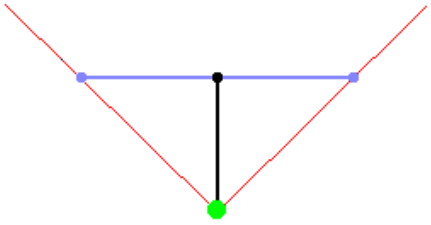


The image above represents such a 2D camera. The green spot is the position (vector "pos"). The black line, ending in the black spot, represents the direction vector (vector "dir"), so the position of the black dot is pos+dir. The blue line represents the full camera plane, the vector from the black dot to the right blue dot represents the vector "plane", so the position of the right blue point is pos+dir+plane, and the posistion of the left blue dot is pos+dir-plane (these are all vector additions).

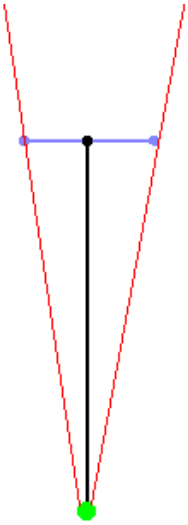
The red lines in the image are a few rays. The direction of these rays is easily calculated out of the camera: it's the sum of the direction vector of the camear, and a part of the plane vector of the camera: for example the third red ray on the image, goes through the right part of the camera plane at the point about 1/3th of its length. So the direction of this ray is dir + plane*1/3. This ray direction is the vector rayDir, and the X and Y component of this vector are then used by the DDA algorithm.

The two outer lines, are the left and right border of the screen, and the angle between those two lines is called the Field Of Vision or FOV. The FOV is determinated by the ratio of the length of the direction vector, and the length of the plane. Here are a few examples of different FOV's:

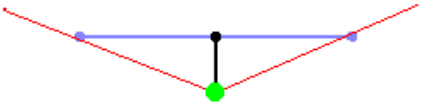
If the direction vector and the camera plane vector have the same length, the FOV will be 90°:



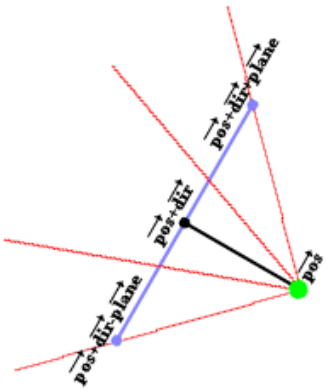
If the direction vector is much longer than the camera plane, the FOV will be much smaller than 90°, and you'll have a very narrow vision. You'll see everything more detailed though and there will be less depth, so this is the same as zooming in:



If the direction vector is shorter than the camera plane, the FOV will be larger than 90° (180° is the maximum, if the direction vector is close to 0), and you'll have a much wider vision, like zooming out:



When the player rotates, the camera has to rotate, so both the direction vector and the plane vector have to be rotated. Then, the rays will all automaticly rotate as well.



To rotate a vector, multiply it with the rotation matrix

```
[ cos(a) -sin(a) ]
[ sin(a)  cos(a) ]
```

If you don't know about vectors and matrices, try to find a tutorial with google, an appendix about those is planned for this tutorial later.

There's nothing that forbids you to use a camera plane that isn't perpendicular to the direction, but the result will look like a "skewed" world.

Untextured Raycaster

Download the source code here: raycaster_flat.cpp

To start with the basics, we'll begin with an untextured raycaster. This example also includes an fps counter (frames per second), and input keys with collision detection to move and rotate.

The map of the world is a 2D array, where each value represents a square. If the value is 0, that square represents an empty, walkthroughable square, and if the value is higher than 0, it represents a wall with a certain color or texture. The map declared here is very small, only 24 by 24 squares, and is defined directly in the code. For a real game, like Wolfenstein 3D, you use a bigger map and load it from a file instead. All the zero's in the grid are empty space, so basicly you see a very big room, with a wall around it (the values 1), a small room inside it (the values 2), a few pilars (the values 3), and a corridor with a room (the values 4). Note that this code isn't inside any function yet, put it before the main function starts.

```
#define mapWidth 24
#define mapHeight 24
#define screenWidth 640
#define screenHeight 480

int worldMap[mapWidth][mapHeight]=
{
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,2,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,3,0,0,0,3,0,0,0,1},
  {1,0,0,0,0,0,2,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,2,2,0,2,2,0,0,0,0,3,0,3,0,3,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,5,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,4,4,4,4,4,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
  {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
};
```

A first few variables are declared: posX and posY represent the position vector of the player, dirX and dirY represent the direction of the player, and planeX and planeY the camera plane of the player. Make sure the camera plane is perpendicular to the direction, but you can change the length of it. The ratio between the length of the direction and the camera plane determinates the FOV, here the direction vector is a bit longer than the camera plane, so the FOV will be smaller than 90° (more precisely, the FOV is 2 * atan(0.66/1.0)=66°, which is perfect for a first person shooter game). Later on when rotating around with the input keys, the values of dir and plane will be changed, but they'll always remain perpendicular and keep the same length.

The variables time and oldTime will be used to store the time of the current and the previous frame, the time difference between these two can be used to determinate how much you should move when a certain key is pressed (to move a constant speed no matter how long the calculation of the frames takes), and for the FPS counter.

```
int main(int /*argc*/, char */*argv*/[])
{
  double posX = 22, posY = 12;  //x and y start position
  double dirX = -1, dirY = 0; //initial direction vector
  double planeX = 0, planeY = 0.66; //the 2d raycaster version of camera plane
```

```
double time = 0; //time of current frame
double oldTime = 0; //time of previous frame
```

The rest of the main function starts now. First, the screen is created with a resolution of choice. If you pick a large resolution, like 1280*1024, the effect will go quite slow, not because the raycating algorithm is slow, but simply because uploading a whole screen from the CPU to the video card goes so slow.

```
screen(screenWidth, screenHeight, 0, "Raycaster");
```

After setting up the screen, the gameloop starts, this is the loop that draws a whole frame and reads the input every time.

```
while(!done())
{
```

Here starts the actual raycasting. The raycasting loop is a for loop that goes through every x, so there isn't a calculation for every pixel of the screen, but only for every vertical stripe, which isn't much at all! To begin the raycasting loop, some variables are delcared and calculated:

The ray starts at the position of the player (posX, posY).

cameraX is the x-coordinate on the camera plane that the current x-coordinate of the screen represents, done this way so that the right side of the screen will get coordinate 1, the center of the screen gets coordinate 0, and the left side of the screen gets coordinate -1. Out of this, the direction of the ray can be calculated as was explained earlier: as the sum of the direction vector, and a part of the plane vector. This has to be done both for the x and y coordinate of the vector (since adding two vectors is adding their x-coordinates, and adding their y-coordinates).

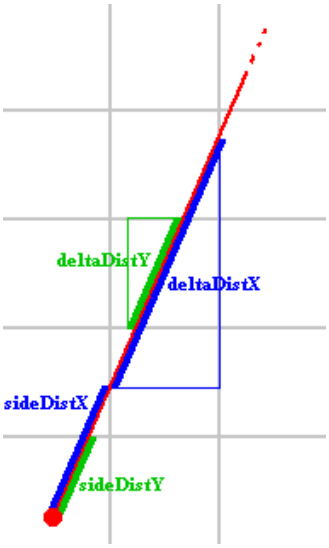
```
  for(int x = 0; x < w; x++)
  {
    //calculate ray position and direction
    double cameraX = 2 * x / double(w) - 1; //x-coordinate in camera space
    double rayDirX = dirX + planeX * cameraX;
    double rayDirY = dirY + planeY * cameraX;
```

In the next code piece, more variables are declared and calculated, these have relevance to the DDA algorithm:

mapX and mapY represent the current square of the map the ray is in. The ray position itself is a floating point number and contains both info about in which square of the map we are, and *where* in that square we are, but mapX and mapY are only the coordinates of that square.

sideDistX and sideDistY are initially the distance the ray has to travel from its start position to the first x-side and the first y-side. Later in the code they will be incremented while steps are taken.

deltaDistX and deltaDistY are the distance the ray has to travel to go from 1 x-side to the next x-side, or from 1 y-side to the next y-side. The following image shows the initial sideDistX, sideDistY and deltaDistX and deltaDistY:



When deriving deltaDistX geometrically you get, with Pythagoras, the formulas below. For the blue triangle (deltaDistX), one side has length 1 (as it is exactly one cell) and the other has length raydirY / raydirX because it is exaclty the amount of units the ray goes in the y-direction when taking 1 step in the X-direction. For the green triangle (deltaDistY), the formula is similar.

deltaDistX = sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX))
deltaDistY = sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY))

But this can be simplified to:

deltaDistX = abs(|rayDir| / rayDirX)
deltaDistY = abs(|rayDir| / rayDirY)

Where |rayDir| is the length of the vector rayDirX, rayDirY (that is sqrt(rayDirX * rayDirX + rayDirY * rayDirY)): you can indeed verify that e.g. sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX)) equals abs(sqrt(rayDirX * rayDirX + rayDirY * rayDirY) / rayDirX). However, we can use 1 instead of |rayDir|, because only the *ratio* between deltaDistX and deltaDistY matters for the DDA code that follows later below, so we get:

deltaDistX = abs(1 / rayDirX)
deltaDistY = abs(1 / rayDirY)

Due to this, the deltaDist and sideDist values used in the code do not match the lengths shown in the picture above, but their relative sizes all still match.

[thanks to Artem for spotting this simplification]

The variable perpWallDist will be used later to calculate the length of the ray.

The DDA algorithm will always jump exactly one square each loop, either a square in the x-direction, or a square in the y-direction. If it has to go in the negative or positive x-direction, and the negative or positive y-direction will depend on the direction of the ray, and this fact will be stored in stepX and stepY. Those variables are always either -1 or +1.

Finally, hit is used to determinate whether or not the coming loop may be ended, and side will contain if an x-side or a y-side of a wall was hit. If an x-side was hit, side is set to 0, if an y-side was hit, side will be 1. By x-side and y-side, I mean the lines of the grid that are the borders between two squares.

```
//which box of the map we're in
int mapX = int(posX);
int mapY = int(posY);
```

```
//length of ray from current position to next x or y-side
double sideDistX;
double sideDistY;

 //length of ray from one x or y-side to next x or y-side
double deltaDistX = (rayDirX == 0) ? 1e30 : std::abs(1 / rayDirX);
double deltaDistY = (rayDirY == 0) ? 1e30 : std::abs(1 / rayDirY);
double perpWallDist;

//what direction to step in x or y-direction (either +1 or -1)
int stepX;
int stepY;

int hit = 0; //was there a wall hit?
int side; //was a NS or a EW wall hit?
```

NOTE: If rayDirX or rayDirY are 0, the division through zero is avoided by setting it to a very high value 1e30. If you are using a language such as C++, Java or JS, this is not actually needed, as it supports the IEEE 754 floating point standard, which gives the result Infinity, which works correctly in the code below. However, some other languages, such as Python, disallow division through zero, so the more generic code that works everywhere is given above. 1e30 is an arbitrarily chosen high enough number and can be set to Infinity if your programming language supports assiging that value.

Now, before the actual DDA can start, first stepX, stepY, and the initial sideDistX and sideDistY still have to be calculated.

If the ray direction has a negative x-component, stepX is -1, if the ray direciton has a positive x-component it's +1. If the x-component is 0, it doesn't matter what value stepX has since it'll then be unused.
The same goes for the y-component.

If the ray direction has a negative x-component, sideDistX is the distance from the ray starting position to the first side to the left, if the ray direciton has a positive x-component the first side to the right is used instead.
The same goes for the y-component, but now with the first side above or below the position.
For these values, the integer value mapX is used and the real position subtracted from it, and 1.0 is added in some of the cases depending if the side to the left or right, of the top or the bottom is used.
Then you get the perpendicular distance to this side, so multiply it with deltaDistX or deltaDistY to get the real Euclidean distance.

```
//calculate step and initial sideDist
if (rayDirX < 0)
{
  stepX = -1;
  sideDistX = (posX - mapX) * deltaDistX;
}
else
{
  stepX = 1;
  sideDistX = (mapX + 1.0 - posX) * deltaDistX;
}
if (rayDirY < 0)
{
  stepY = -1;
  sideDistY = (posY - mapY) * deltaDistY;
}
else
{
  stepY = 1;
  sideDistY = (mapY + 1.0 - posY) * deltaDistY;
}
```

Now the actual DDA starts. It's a loop that increments the ray with 1 square every time, until a wall is

hit. Each time, either it jumps a square in the x-direction (with stepX) or a square in the y-direction (with stepY), it always jumps 1 square at once. If the ray's direction would be the x-direction, the loop will only have to jump a square in the x-direction everytime, because the ray will never change its y-direction. If the ray is a bit sloped to the y-direction, then every so many jumps in the x-direction, the ray will have to jump one square in the y-direction. If the ray is exactly the y-direction, it never has to jump in the x-direction, etc...

sideDistX and sideDistY get incremented with deltaDistX with every jump in their direction, and mapX and mapY get incremented with stepX and stepY respectively.

When the ray has hit a wall, the loop ends, and then we'll know whether an x-side or y-side of a wall was hit in the variable "side", and what wall was hit with mapX and mapY. We won't know exactly where the wall was hit however, but that's not needed in this case because we won't use textured walls for now.

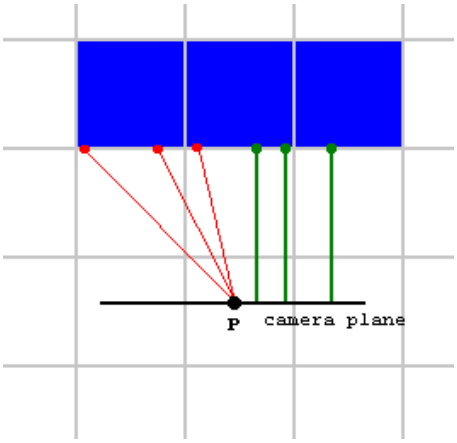
```
//perform DDA
while (hit == 0)
{
  //jump to next map square, either in x-direction, or in y-direction
  if (sideDistX < sideDistY)
  {
    sideDistX += deltaDistX;
    mapX += stepX;
    side = 0;
  }
  else
  {
    sideDistY += deltaDistY;
    mapY += stepY;
    side = 1;
  }
  //Check if ray has hit a wall
  if (worldMap[mapX][mapY] > 0) hit = 1;
}
```

After the DDA is done, we have to calculate the distance of the ray to the wall, so that we can calculate how high the wall has to be drawn after this.

We don't use the Euclidean distance to the point representing player, but instead the distance to the camera plane (or, the distance of the point projected on the camera direction to the player), to avoid the fisheye effect. The fisheye effect is an effect you see if you use the real distance, where all the walls become rounded, and can make you sick if you rotate.

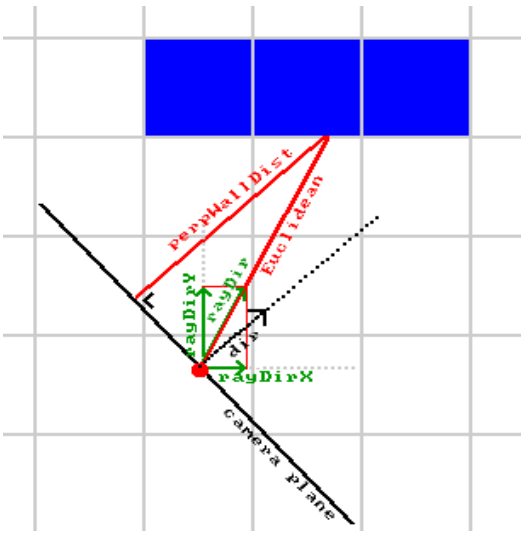
The following image shows why we take distance to camera plane instead of player. With P the player, and the black line the camera plane: To the left of the player, a few red rays are shown from hitpoints on the wall to the player, representing Euclidean distance. On the right side of the player, a few green rays are shown going from hitpoints on the wall directly to the camera plane instead of to the player. So the lengths of those green lines are examples of the perpendicular distance we'll use instead of direct Euclidean distance.

In the image, the player is looking directly at the wall, and in that case you would expect the wall's bottom and top to form a perfectly horizontal line on the screen. However, the red rays all have a different lenght, so would compute different wall heights for different vertical stripes, hence the rounded effect. The green rays on the right all have the same length, so will give the correct result. The same still apllies for when the player rotates (then the camera plane is no longer horizontal and the green lines will have different lengths, but still with a constant change between each) and the walls become diagonal but straight lines on the screen. This explanation is somewhat handwavy but gives the idea.



Note that this part of the code isn't "fisheye correction", such a correction isn't needed for the way of raycasting used here, the fisheye effect is simply avoided by the way the distance is calculated here. It's even easier to calculate this perpendicular distance than the real distance, we don't even need to know the exact location where the wall was hit.

This perpenducular distance is called "perpWallDist" in the code. One way to compute it is to use the formula for shortest distance from a point to a line, where the point is where the wall was hit, and the line is the camera plane:



However, it can be computed simpler than that: due to how deltaDist and sideDist were scaled by a factor of |rayDir| above, the length of sideDist already almost equals perpWallDist. We just need to subtract deltaDist once from it, going one step back, because in the DDA steps above we went one step further to end up inside the wall.

Depending on whether the ray hit an X side or Y side, the formula is computed using sideDistX, or sideDistY.

```
//Calculate distance projected on camera direction (Euclidean distance would give fisheye effect!)
if(side == 0) perpWallDist = (sideDistX - deltaDistX);
else          perpWallDist = (sideDistY - deltaDistY);
```

A more detailed derivation of the perpWallDist formula is depicted in the image below, for the side == 1 case.

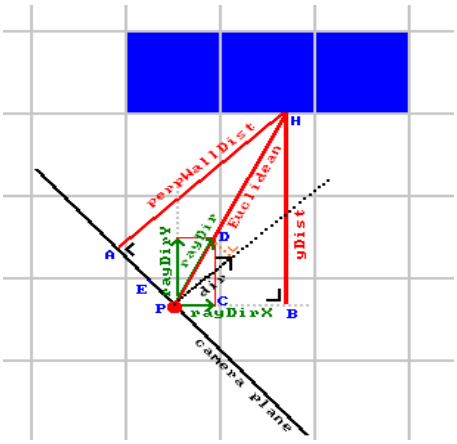
Meaning of the points:

- P: position of the player, (posX, posY) in the code

- H: hitpoint of the ray on the wall. Its y-position is known to be mapY + (1 - stepY) / 2
- yDist matches "(mapY + (1 - stepY) / 2 - posY)", this is the y coordinate of the Euclidean distance vector, in world coordinates. Here, (1 - stepY) / 2) is a correction term that is 0 or 1 based on positive or negative y direction, which is also used in the initialization of sideDistY.
- dir: the main player looking direction, given by dirX,dirY in the code. The length of this vector is always exactly 1. This matches the looking direction in the center of the screen, as opposed to the direction of the current ray. It is perpendicular to the camera plane, and perpWallDist is parallel to this.
- orange dotted line (may be hard to see, use CTRL+scrollwheel or CTRL+plus to zoom in a desktop browser to see it better): the value that was added to dir to get rayDir. Importantly, this is parallel to the camera plane, perpendicular to dir.
- A: point of the camera plane closest to H, the point where perpWallDist intersects with camera plane
- B: point of X-axis through player closest to H, point where yDist crosses the world X-axis through the player
- C: point at player position + rayDirX
- D: point at player position + rayDir.
- E: This is point D with the dir vector subtracted, in other words, E + dir = D.
- points A, B, C, D, E, H and P are used in the explanation below: they form triangles which are considered: BHP, CDP, AHP and DEP.

The actual derivation:

- 1: Triangles PBH and PCD have the same shape but different size, so same ratios of edges
- 2: Given step 1, the triangles show that the ratio yDist / rayDirY is equal to the ratio Euclidean / |rayDir|, so now we can derive perpWallDist = Euclidean / |rayDir| instead.
- 3: Triangles AHP and EDP have the same shape but different size, so same ratios of edges. Length of edge ED, that is |ED|, equals length of dir, |dir|, which is 1. Similarly, |DP| equals |rayDir|.
- 4: Given step 3, the triangles show that the ratio Euclidean / |rayDir| = perpWallDist / |dir| = perpWallDist / 1.
- 5: Combining steps 4 and 2 shows that perpWallDist = yDist / rayDirY, where yDist is mapY + (1 - stepY) / 2) - posY
- 6: In the code, sideDistY - deltaDistY, after the DDA steps, equals (posY + (1 - stepY) / 2 - mapY) * deltaDistY (given that sideDistY is computed from posY and mapY), so yDist = (sideDistY - deltaDistY) / deltaDistY
- 7: Given that deltaDistY = 1 / |rayDirY|, step 6 gives that yDist = (sideDistY - deltaDistY) * |rayDirY|
- 8: Combining steps 5 and 7 gives perpWallDist = yDist / rayDirY = (sideDistY - deltaDistY) / |rayDirY| / rayDirY.
- 9: Given how cases for signs of sideDistY and deltaDistY in the code are handled the absolute value doesn't matter, and equals (sideDistY - deltaDistY), which is the formula used



[Thanks to Thomas van der Berg in 2016 for pointing out simplifications of the code (perpWallDist

could be simplified and the value reused for wallX).
[Thanks to Roux Morgan in 2020 for helping to clarify the explanation of perpWallDist, the tutorial was lacking some information before this]
[Thanks to Noah Wagner and Elias for finding further simplifications for perpWallDist]

Now that we have the calculated distance (perpWallDist), we can calculate the height of the line that has to be drawn on screen: this is the inverse of perpWallDist, and then multiplied by h, the height in pixels of the screen, to bring it to pixel coordinates. You can of course also multiply it with another value, for example 2*h, if you want to walls to be higher or lower. The value of h will make the walls look like cubes with equal height, width and depth, while large values will create higher boxes (depending on your monitor).

Then out of this lineHeight (which is thus the height of the vertical line that should be drawn), the start and end position of where we should really draw are calculated. The center of the wall should be at the center of the screen, and if these points lie outside the screen, they're capped to 0 or h-1.

```
      //Calculate height of line to draw on screen
      int lineHeight = (int)(h / perpWallDist);

      //calculate lowest and highest pixel to fill in current stripe
      int drawStart = -lineHeight / 2 + h / 2;
      if(drawStart < 0)drawStart = 0;
      int drawEnd = lineHeight / 2 + h / 2;
      if(drawEnd >= h)drawEnd = h - 1;
```

Finally, depending on what number the wall that was hit has, a color is chosen. If an y-side was hit, the color is made darker, this gives a nicer effect. And then the vertical line is drawn with the verLine command. This ends the raycasting loop, after it has done this for every x at least.

```
      //choose wall color
      ColorRGB color;
      switch(worldMap[mapX][mapY])
      {
        case 1:  color = RGB_Red;  break; //red
        case 2:  color = RGB_Green;  break; //green
        case 3:  color = RGB_Blue;   break; //blue
        case 4:  color = RGB_White;  break; //white
        default: color = RGB_Yellow; break; //yellow
      }

      //give x and y sides different brightness
      if (side == 1) {color = color / 2;}

      //draw the pixels of the stripe as a vertical line
      verLine(x, drawStart, drawEnd, color);
    }
```

After the raycasting loop is done, the time of the current and the previous frame are calculated, the FPS (frames per second) is calculated and printed, and the screen is redrawn so that everything (all the walls, and the value of the fps counter) becomes visible. After that the backbuffer is cleared with cls(), so that when we draw the walls again the next frame, the floor and ceiling will be black again instead of still containing pixels from the previous frame.

The speed modifiers use frameTime, and a constant value, to determinate the speed of the moving and rotating of the input keys. Thanks to using the frameTime, we can make sure that the moving and rotating speed is independent of the processor speed.

```
    //timing for input and FPS counter
    oldTime = time;
    time = getTicks();
```

```
    double frameTime = (time - oldTime) / 1000.0; //frameTime is the time this frame has taken, in seconds
    print(1.0 / frameTime); //FPS counter
    redraw();
    cls();

    //speed modifiers
    double moveSpeed = frameTime * 5.0; //the constant value is in squares/second
    double rotSpeed = frameTime * 3.0; //the constant value is in radians/second
```

The last part is the input part, the keys are read.

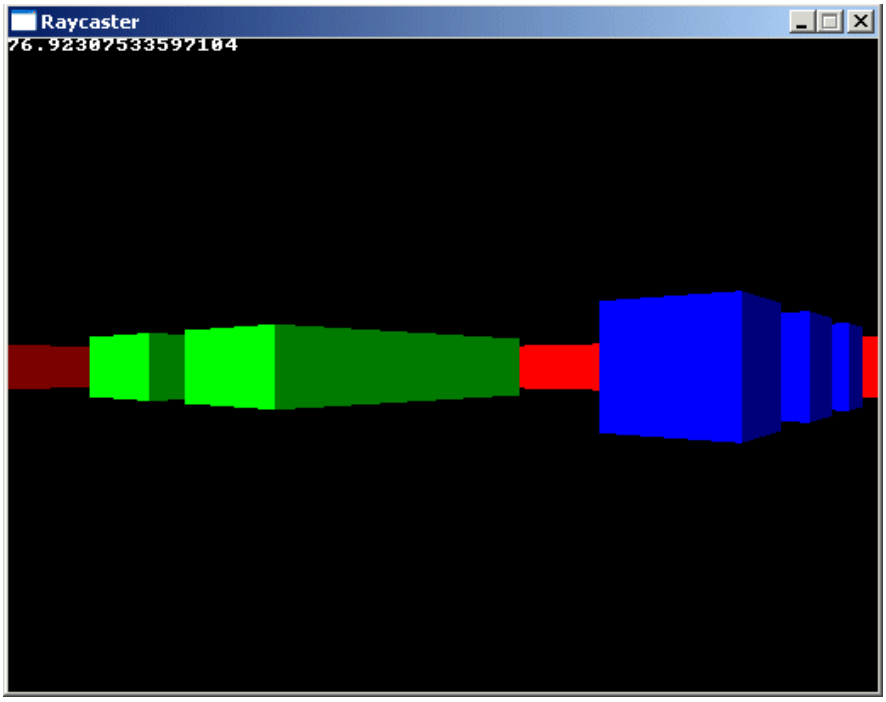
If the up arrow is pressed, the player will move forward: add dirX to posX, and dirY to posY. This assumes that dirX and dirY are normalized vectors (their length is 1), but they were initially set like this, so it's ok. There's also a simple collision detection built in, namely if the new position will be inside a wall, you won't move. This collision detection can be improved however, for example by checking if a circle around the player won't go inside the wall instead of just a single point.

The same is done if you press the down arrow, but then the direction is subtracted instead.

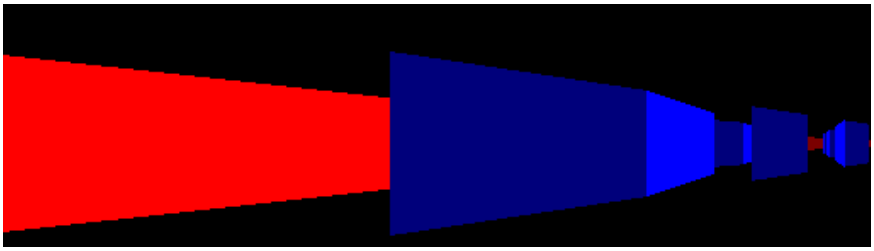
To rotate, if the left or right arrow is pressed, both the direction vector and plane vector are rotated by using the formulas of multiplication with the rotation matrix (and over the angle rotSpeed).

```
    readKeys();
    //move forward if no wall in front of you
    if (keyDown(SDLK_UP))
    {
      if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
      if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
    }
    //move backwards if no wall behind you
    if (keyDown(SDLK_DOWN))
    {
      if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
      if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
    }
    //rotate to the right
    if (keyDown(SDLK_RIGHT))
    {
      //both camera direction and camera plane must be rotated
      double oldDirX = dirX;
      dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
      dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
      double oldPlaneX = planeX;
      planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
      planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
    }
    //rotate to the left
    if (keyDown(SDLK_LEFT))
    {
      //both camera direction and camera plane must be rotated
      double oldDirX = dirX;
      dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
      dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
      double oldPlaneX = planeX;
      planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
      planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
    }
  }
}
```

This concludes the code of the untextured raycaster, the result looks like this, and you can walk around in the map:



Here's an example of what happens if the camera plane isn't perpendicular to the direction vector, the world appears skewed:



Textured Raycaster

Download the source code here: raycaster_textured.cpp

The core of the textured version of the raycaster is almost the same, only at the end some extra calculations need to be done for the textures, and a loop in the y-direction is required to go through every pixel to determinate which texel (texture pixel) of the texture should be used for it.

The vertical stripes can't be drawn with the vertical line command anymore, instead every pixel has to be drawn seperately. The best way is to use a 2D array as screen buffer this time, and copy it to the screen at once, that goes a lot faster than using pset.

Of course we now also need an extra array for the textures, and since the "drawbuffer" function works with single integer values for colors (instead of 3 separate bytes for R, G and B), the textures are stored in this format as well. Normally, you'd load the textures from a texture file, but for this simple example some dumb textures are generated instead.

The code is mostly the same as the previous example, the bold parts are new. Only new parts are explained.

The screenWidth and screenHeight are now defined in the beginning because we need the same value for the screen function, and to create the screen buffer. Also new are the texture width and

height that are defined here. These are obviously the width and height in texels of the textures.

The world map is changed too, this is a more complex map with corridors and rooms to show the different textures. Again, the 0's are empty walkthrougable spaces, and each positive number corresponds to a different texture.

```
#define screenWidth 640
#define screenHeight 480
#define texWidth 64
#define texHeight 64
#define mapWidth 24
#define mapHeight 24

int worldMap[mapWidth][mapHeight]=
{
  {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,7,7,7,7,7,7,7,7},
  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},
  {4,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},
  {4,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},
  {4,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},
  {4,0,4,0,0,0,0,5,5,5,5,5,5,5,5,5,7,7,0,7,7,7,7,7},
  {4,0,5,0,0,0,0,5,0,5,0,5,0,5,0,5,7,0,0,0,7,7,7,1},
  {4,0,6,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,0,0,0,8},
  {4,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,7,7,1},
  {4,0,8,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,0,0,0,8},
  {4,0,0,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,7,7,7,1},
  {4,0,0,0,0,0,0,5,5,5,5,0,5,5,5,5,7,7,7,7,7,7,7,1},
  {6,6,6,6,6,6,6,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6,6},
  {8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
  {6,6,6,6,6,6,0,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6,6},
  {4,4,4,4,4,4,0,4,4,4,6,0,6,2,2,2,2,2,2,3,3,3,3},
  {4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,0,0,0,2},
  {4,0,0,0,0,0,0,0,0,0,0,0,6,2,0,0,5,0,0,2,0,0,0,2},
  {4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,2,0,2,2},
  {4,0,6,0,6,0,0,0,0,4,6,0,0,0,0,0,5,0,0,0,0,0,0,2},
  {4,0,0,5,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,2,0,2,2},
  {4,0,6,0,6,0,0,0,0,4,6,0,6,2,0,0,5,0,0,2,0,0,0,2},
  {4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,0,0,0,2},
  {4,4,4,4,4,4,4,4,4,4,1,1,1,2,2,2,2,2,2,3,3,3,3,3}
};
```

The screen buffer and texture arrays are declared here. The texture array is an array of std::vectors, each with a certain width * height pixels.

```
int main(int /*argc*/, char */*argv*/[])
{
  double posX = 22.0, posY = 11.5;  //x and y start position
  double dirX = -1.0, dirY = 0.0; //initial direction vector
  double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane

  double time = 0; //time of current frame
  double oldTime = 0; //time of previous frame

  Uint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline
  std::vector texture[8];
  for(int i = 0; i < 8; i++) texture[i].resize(texWidth * texHeight);
```

The main function now begins with generating the textures. We have a double loop that goes through every pixel of the textures, and then the corresponding pixel of each texture gets a certain value calculated out of x and y. Some textures get a XOR pattern, some a simple gradient, others a sort of brick pattern, basicly it are all quite simple patterns, it's not going to look all that beautiful, for better textures see the next chapter.

```
  screen(screenWidth,screenHeight, 0, "Raycaster");

  //generate some textures
```

```
for(int x = 0; x < texWidth; x++)
for(int y = 0; y < texHeight; y++)
{
  int xorcolor = (x * 256 / texWidth) ^ (y * 256 / texHeight);
  //int xcolor = x * 256 / texWidth;
  int ycolor = y * 256 / texHeight;
  int xycolor = y * 128 / texHeight + x * 128 / texWidth;
  texture[0][texWidth * y + x] = 65536 * 254 * (x != y && x != texWidth - y); //flat red texture with black cross
  texture[1][texWidth * y + x] = xycolor + 256 * xycolor + 65536 * xycolor; //sloped greyscale
  texture[2][texWidth * y + x] = 256 * xycolor + 65536 * xycolor; //sloped yellow gradient
  texture[3][texWidth * y + x] = xorcolor + 256 * xorcolor + 65536 * xorcolor; //xor greyscale
  texture[4][texWidth * y + x] = 256 * xorcolor; //xor green
  texture[5][texWidth * y + x] = 65536 * 192 * (x % 16 && y % 16); //red bricks
  texture[6][texWidth * y + x] = 65536 * ycolor; //red gradient
  texture[7][texWidth * y + x] = 128 + 256 * 128 + 65536 * 128; //flat grey texture
}
```

This is again the start of the gameloop and initial declarations and calculations before the DDA algorithm. Nothing has changed here.

```
//start the main loop
while(!done())
{
  for(int x = 0; x < w; x++)
  {
    //calculate ray position and direction
    double cameraX = 2*x/double(w)-1; //x-coordinate in camera space
    double rayDirX = dirX + planeX*cameraX;
    double rayDirY = dirY + planeY*cameraX;

    //which box of the map we're in
    int mapX = int(posX);
    int mapY = int(posY);

    //length of ray from current position to next x or y-side
    double sideDistX;
    double sideDistY;

    //length of ray from one x or y-side to next x or y-side
    double deltaDistX = sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX));
    double deltaDistY = sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY));
    double perpWallDist;

    //what direction to step in x or y-direction (either +1 or -1)
    int stepX;
    int stepY;

    int hit = 0; //was there a wall hit?
    int side; //was a NS or a EW wall hit?

    //calculate step and initial sideDist
    if (rayDirX < 0)
    {
      stepX = -1;
      sideDistX = (posX - mapX) * deltaDistX;
    }
    else
    {
      stepX = 1;
      sideDistX = (mapX + 1.0 - posX) * deltaDistX;
    }
    if (rayDirY < 0)
    {
      stepY = -1;
      sideDistY = (posY - mapY) * deltaDistY;
    }
    else
    {
      stepY = 1;
      sideDistY = (mapY + 1.0 - posY) * deltaDistY;
    }
```

This is again the DDA loop, and the calculations of the distance and height, nothing has changed here either.

```
//perform DDA
while (hit == 0)
{
  //jump to next map square, either in x-direction, or in y-direction
  if (sideDistX < sideDistY)
  {
    sideDistX += deltaDistX;
    mapX += stepX;
    side = 0;
  }
  else
  {
    sideDistY += deltaDistY;
    mapY += stepY;
    side = 1;
  }
  //Check if ray has hit a wall
  if (worldMap[mapX][mapY] > 0) hit = 1;
}

//Calculate distance of perpendicular ray (Euclidean distance would give fisheye effect!)
if(side == 0) perpWallDist = (sideDistX - deltaDistX);
else          perpWallDist = (sideDistY - deltaDistY);

//Calculate height of line to draw on screen
int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current stripe
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0) drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h) drawEnd = h - 1;
```

The following calculations are new however, and replace the color chooser of the untextured raycaster.
The variable texNum is the value of the current map square minus 1, the reason is that there exists a texture 0, but map tile 0 has no texture since it represents an empty space. To be able to use texture 0 anyway, substract 1 so that map tiles with value 1 will give texture 0, etc...

The value wallX represents the exact value where the wall was hit, not just the integer coordinates of the wall. This is required to know which x-coordinate of the texture we have to use. This is calculated by first calculating the exact x or y coordinate in the world, and then substracting the integer value of the wall off it. Note that even if it's called wallX, it's actually an y-coordinate of the wall if side==1, but it's always the x-coordinate of the texture.

Finally, texX is the x-coordinate of the texture, and this is calculated out of wallX.

```
//texturing calculations
int texNum = worldMap[mapX][mapY] - 1; //1 subtracted from it so that texture 0 can be used!

//calculate value of wallX
double wallX; //where exactly the wall was hit
if (side == 0) wallX = posY + perpWallDist * rayDirY;
else           wallX = posX + perpWallDist * rayDirX;
wallX -= floor((wallX));

//x coordinate on the texture
int texX = int(wallX * double(texWidth));
if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;
if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;
```

Now that we know the x-coordinate of the texture, we know that this coordinate will remain the same, because we stay in the same vertical stripe of the screen. Now we need a loop in the y-direction to

give each pixel of the vertical stripe the correct y-coordinate of the texture, called texY.

The value of texY is calculated by increasing by a precomputed step size (which is possible because this is constant in the vertical stripe) for each pixel. The step size tells how much to increase in the texture coordinates (in floating point) for every pixel in vertical screen coordinates. It then needs to cast the floating point value to integer to select the actual texture pixel.

NOTE: a faster integer-only bresenham or DDA algorithm may be possible for this.

NOTE: The stepping being done here is affine texture mapping, which means we can interpolate linearly between two points rather than have to compute a different division for each pixel. This is not perspective correct in general, but for perfectly vertical walls (and also perfectly horizontal floors/ceilings) it is, so we can use it for raycasting.

The color of the pixel to be drawn is then simply gotten from texture[texNum][texX][texY], which is the correct texel of the correct texture.

Like the untextured raycaster, here too we'll make the color value darker if an y-side of the wall was hit, because that looks a little bit better (like there is a sort of lighting). However, because the color value doesn't exist out of a separate R, G and B value, but these 3 bytes sticked together in a single integer, a not so intuitive calculation is used.

The color is made darker by dividing R, G and B through 2. Dividing a decimal number through 10, can be done by removing the last digit (e.g. 300/10 is 30: the last zero is removed). Similarly, dividing a binary number through 2, which is what is done here, is the same as removing the last bit. This can be done by bitshifting it to the right with >>1. But, here we're bitshifting a 24-bit integer (actually 32-bit, but the first 8 bits aren't used). Because of this, the last bit of one byte will become the first bit of the next byte, and that screws up the color values! So after the bitshift, the first bit of every byte has to be set to zero, and that can be done by binary "AND-ing" the value with the binary value 011111110111111101111111, which is 8355711 in decimal. So the result of this is indeed a darker color.

Finally, the current buffer pixel is set to this color, and we move on to the next y.

```
      // How much to increase the texture coordinate per screen pixel
  double step = 1.0 * texHeight / lineHeight;
  // Starting texture coordinate
  double texPos = (drawStart - h / 2 + lineHeight / 2) * step;
  for(int y = drawStart; y<drawEnd; y++)
  {
    // Cast the texture coordinate to integer, and mask with (texHeight - 1) in case of overflow
    int texY = (int)texPos & (texHeight - 1);
    texPos += step;
    Uint32 color = texture[texNum][texHeight * texY + texX];
    //make color darker for y-sides: R, G and B byte each divided through two with a "shift" and an "and"
    if(side == 1) color = (color >> 1) & 8355711;
    buffer[y][x] = color;
  }
}
```

Now the buffer still has to be drawn, and after that it has to be cleared (where in the untextured version we simply had to use "cls". Ensure to do it in scanline order for speed thanks to memory locality for caching). The rest of this code is again the same.

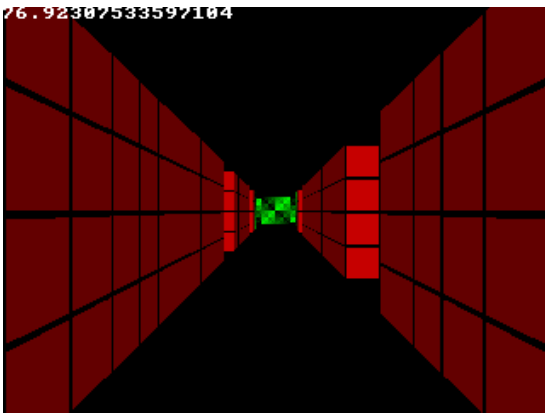
```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of cls()
//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds
print(1.0 / frameTime); //FPS counter
```

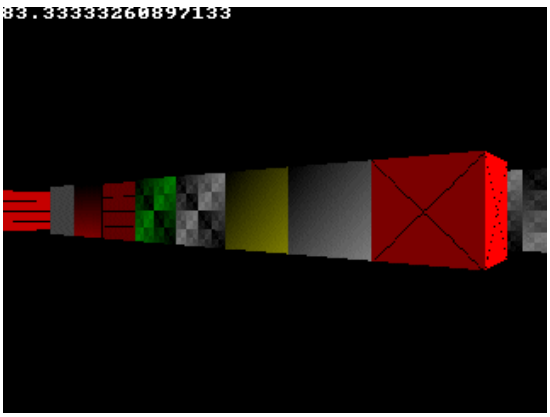
```
    redraw();

    //speed modifiers
    double moveSpeed = frameTime * 5.0; //the constant value is in squares/second
    double rotSpeed = frameTime * 3.0; //the constant value is in radians/second
```

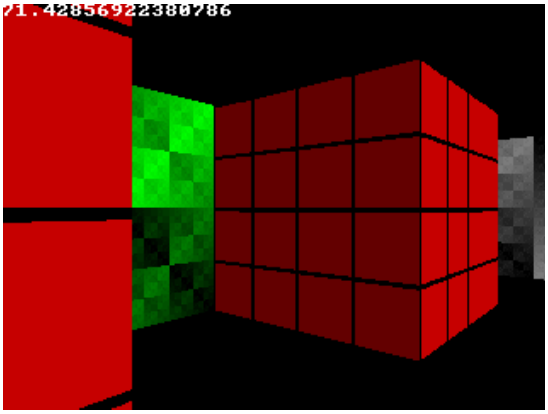
And here's again the keys, nothing has changed here either. If you like you can try to add strafe keys (to strafe to the left and right). These have to be made the same way as the up and down keys, but use planeX and planeY instead of dirX and dirY.

```
    readKeys();
    //move forward if no wall in front of you
    if (keyDown(SDLK_UP))
    {
      if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
      if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
    }
    //move backwards if no wall behind you
    if (keyDown(SDLK_DOWN))
    {
      if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
      if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
    }
    //rotate to the right
    if (keyDown(SDLK_RIGHT))
    {
      //both camera direction and camera plane must be rotated
      double oldDirX = dirX;
      dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
      dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
      double oldPlaneX = planeX;
      planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
      planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
    }
    //rotate to the left
    if (keyDown(SDLK_LEFT))
    {
      //both camera direction and camera plane must be rotated
      double oldDirX = dirX;
      dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
      dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
      double oldPlaneX = planeX;
      planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
      planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
    }
  }
}
```

Here's a few screenshots of the result:







Note: Usually images are stored by horizontal scanlines, but for a raycaster the textures are drawn as vertical stripes. Therefore, to optimally use the cache of the CPU and avoid page misses, it might be more efficient to store the textures in memory vertical stripe by vertical stripe, instead of per horizontal scanline. To do this, after generating the textures, swap their X and Y by (this code only works if texWidth and texHeight are the same):

```
//swap texture X/Y since they'll be used as vertical stripes
for(size_t i = 0; i < 8; i++)
for(size_t x = 0; x < texSize; x++)
for(size_t y = 0; y < x; y++)
std::swap(texture[i][texSize * y + x], texture[i][texSize * x + y]);
```

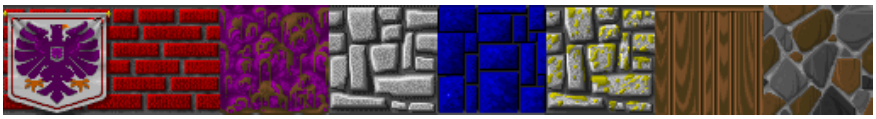
Or just swap X and Y where the textures are generated, but in many cases after loading an image or getting a texture from other formats you'll have it in scanlines anyway and have to swap it this way.

When getting the pixel from the texture then, use the following code instead:

```
Uint32 color = texture[texNum][texSize * texX + texY];
```

Wolfenstein 3D Textures

Instead of just generating some textures, let's load a few from images instead! For example the following 8 textures, which come from Wolfenstein 3D and are copyright by ID Software.



Just replace the part of the code that generates the texture patterns with the following (and make sure those textures are in the correct path). You can download the textures here.

```
//generate some textures
unsigned long tw, th;
loadImage(texture[0], tw, th, "pics/eagle.png");
loadImage(texture[1], tw, th, "pics/redbrick.png");
loadImage(texture[2], tw, th, "pics/purplestone.png");
loadImage(texture[3], tw, th, "pics/greystone.png");
loadImage(texture[4], tw, th, "pics/bluestone.png");
loadImage(texture[5], tw, th, "pics/mossy.png");
loadImage(texture[6], tw, th, "pics/wood.png");
loadImage(texture[7], tw, th, "pics/colorstone.png");
```

In the original Wolfenstein 3D, the colors of one side was also made darker than the color of the other side of a wall to create the shadow effect, but they used a seperate texture every time, a dark and a light one. Here however, only one texture is used for each wall and the line of code that divided R, G and B through 2 is what makes the y-sides darker.

Performance Considerations

On a modern computer, when using high resolution (4K, as of 2019), this software raycaster will be slower than some much more complex 3D graphics get rendered on the GPU with a 3D graphics card.

There are at least two issues holding back speed of the raycaster code in this tutorial, which you can take into account if you'd like to make a super fast raycaster for very high resolutions:

- Raycasting works with vertical stripes, but the screen buffer in memory is laid out with horizontal scanlines. So drawing vertical stripes is bad for memory locality for caching (it is in fact a worst case scenario), and the loss of good caching may hurt the speed more than some of the 3D computations on modern machines. It may be possible to program this with better caching behavior (e.g. processing multiple stripes at once, using a cache-oblivious transpose algorithm, or having a 90 degree rotated raycaster), but for simplicity the rest of this tutorial ignores this caching issue.
- This is using software blitting with SDL (in QuickCG, in redraw()), which is slow for large resolutions compared to hardware rendering. Likely QuickCG's usage of SDL itself is not optimal and e.g. using OpenGL (even for software rendering) may be faster, so that may be fixable behind the scenes. Since this CG tutorial is about software rendering this issue is ignored here as well.

Next Part

Go directly to part II

Last edited: 2020