Tutorial de Computação Gráfica de Lode

Raycasting

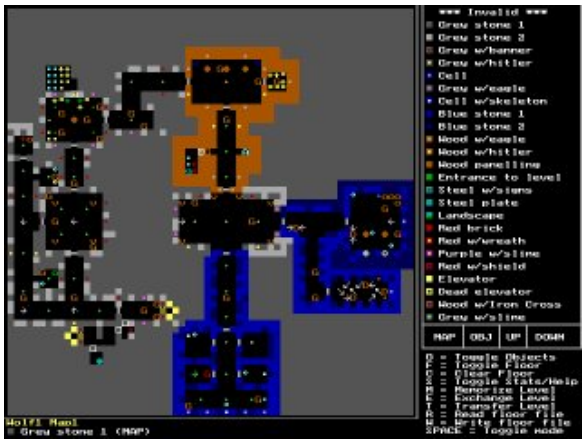
Índice

Voltar ao índice

Introdução

Raycasting é uma técnica de renderização para criar uma perspectiva 3D em um Mapa 2D. Quando os computadores eram mais lentos, não era possível executar motores 3D reais em tempo real, e o raycasting foi a primeira solução. O raycasting pode ser muito rápido, porque apenas um cálculo precisa ser feito para cada linha vertical da tela. O mais conhecido O motor de raycasting de Wolfenstein 3D era muito limitado, permitindo que

o motor de raycasting de Wolfenstein 3D fosse muito limitado, permitindo que o motor de raycasting de Wolfenstein 3D fosse muito limitado. ele para rodar em um computador 286: todas as paredes têm o mesmo altura e são quadrados ortogonais em uma grade 2D, como pode ser visto em esta captura de tela de um editor de mapas para Wolf3D:



Coisas como escadas, saltos ou diferenças de altura são impossíveis de fazer com este motor. Jogos posteriores, como Doom e Duke Nukem 3D também usava raycasting, mas motores muito mais avançados que permitiam paredes inclinadas, diferentes

alturas, pisos e tetos texturizados, paredes transparentes, etc... Os sprites (inimigos, objetos e guloseimas) são imagens 2D, mas os sprites não são discutidos neste tutorial por enquanto.

A*fundição* de raios não é a mesma coisa como*ray tracing*! Raycasting é uma técnica semi-3D rápida que funciona em tempo real mesmo em 4MHz calculadoras gráficas, enquanto o raytracing é uma renderização realista técnica que suporta reflexos e sombras em verdadeiras cenas 3D, e só recentemente os computadores se tornaram rápidos o suficiente para fazer isso em tempo real para resoluções razoavelmente altas e cenas complexas.

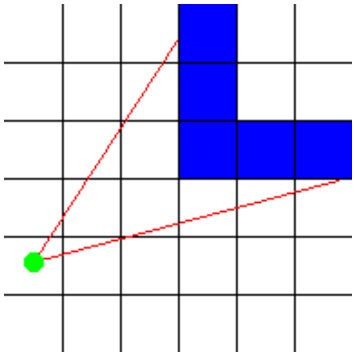
O código dos raycasters não texturizados e texturizados é dado neste documento completamente, mas é bastante longo, você também pode baixar o code em vez disso:

raycaster_flat.cpp
raycaster_textured.cpp

A ideia básica

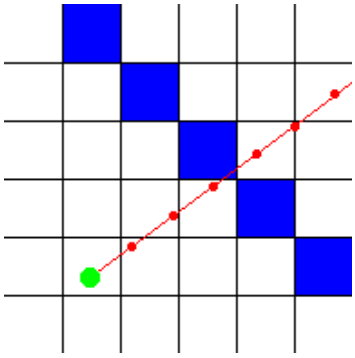
A ideia básica do raycasting é a seguinte: o mapa é um quadrado 2D grade, e cada quadrado pode ser 0 (= sem parede) ou um positivo valor (= uma parede com uma determinada cor ou textura).

Para cada x da tela (ou seja, para cada faixa vertical do), envie um raio que começa no local do jogador e com uma direção que depende tanto da direção de visão do jogador, e a coordenada x da tela. Então, deixe este raio avançar no mapa 2D, até atingir um quadrado do mapa que é uma parede. Se acertar uma parede, calcule a distância desse ponto de vida até o jogador e Use essa distância para calcular a altura que essa parede deve ser desenhada na tela: quanto mais longe a parede, menor ela fica tela, e quanto mais perto, mais alto parece ser. Estes são todos Cálculos 2D. Esta imagem mostra uma visão geral de cima para baixo de dois desses raios (vermelho) que começam no jogador (ponto verde) e atingem o azul paredes:

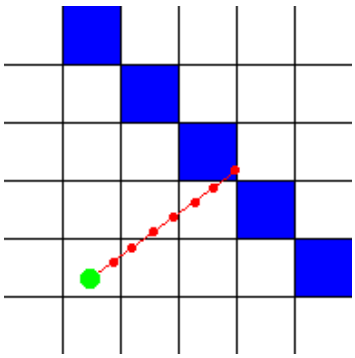


Para encontrar a primeira parede que um raio encontra em seu caminho, você tem para deixá-lo começar na posição do jogador, e então o tempo todo, Verifique se o raio está ou não dentro de uma parede. Se estiver dentro de um parede (batida), então o loop pode parar, calcular a distância e Desenhe a parede com a altura correta. Se a posição do raio não for em uma parede, você tem que rastreá-lo ainda mais: adicione um certo valor a sua posição, na direção da direção desse raio, e Para esta nova posição, verifique novamente se está dentro de uma parede ou não. Continue fazendo isso até que finalmente uma parede seja atingida.

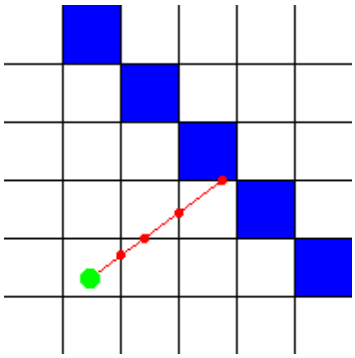
Um humano pode ver imediatamente onde o raio atinge a parede, mas é impossível encontrar qual quadrado o raio atinge imediatamente com um fórmula única, porque um computador só pode verificar um número finito de posições no raio. Muitos raycasters adicionam um valor constante a raio a cada passo, mas então há uma chance de que ele possa perder um parede! Por exemplo, com este raio vermelho, sua posição foi verificada em cada mancha vermelha:



Como você pode ver, o raio atravessa direto a parede azul, mas O computador não detectou isso, porque apenas verificou no posições com os pontos vermelhos. Quanto mais posições você verificar, mais menor a chance de o computador não detectar uma parede, mas o mais cálculos são necessários. Aqui a distância do passo foi reduzida pela metade, então agora ele detecta que o raio atravessou uma parede, embora o posição não está completamente correta:



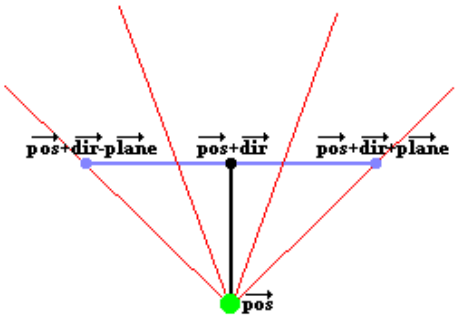
Para precisão infinita com este método, um passo infinitamente pequeno tamanho e, portanto, um número infinito de cálculos seria necessário! Isso é muito ruim, mas, felizmente, existe um método melhor que requer apenas poucos cálculos e, no entanto, detectará todas as paredes: a ideia é verificar em todos os lados de uma parede o raio encontrar. Damos a cada quadrado largura 1, então cada lado de uma parede é um valor inteiro e os lugares intermediários têm um valor após o ponto. Agora o tamanho do passo não é constante, depende da distância para o próximo lado:



Como você pode ver na imagem acima, o raio atinge a parede exatamente onde queremos. Da maneira apresentada neste tutorial, um algoritmo é usado com base em DDA ou "Diferencial Digital Análise". DDA é um algoritmo rápido normalmente usado em grades quadradas para descobrir quais quadrados uma linha atinge (por exemplo, para desenhar uma linha em um tela, que é uma grade de pixels quadrados). Portanto, também podemos usá-lo para descobrir quais quadrados do mapa nosso raio atinge e parar o algoritmo uma vez que um quadrado que é uma parede é atingido.

Alguns traçadores de raios trabalham com ângulos euclidianos para representar o direção do jogador e dos raios, e determinar o Campo de Vista com outro ângulo. Descobri, no entanto, que é muito mais fácil trabalhar com vetores e uma câmera: a posição do jogador é sempre um vetor (uma coordenada x e uma y), mas agora, fazemos o direção um vetor também: então a direção agora é determinada por Dois valores: as coordenadas x e y da direção. Uma direção vetor pode ser visto da seguinte forma: se você desenhar uma linha na direção o jogador olha, através da posição do jogador, então cada ponto da reta é a soma da posição do jogador, e um múltiplo do vetor de direção. O comprimento de um vetor de direção realmente não importa, apenas sua direção. Multiplicando x e y por o mesmo valor altera o comprimento, mas mantém o mesmo direção.

Este método com vetores também requer um vetor extra, que é o vetor do plano da câmera. Em um verdadeiro mecanismo 3D, há também uma câmera avião, e aí este plano é realmente um plano 3D, então dois vetores (u e v) são obrigados a representá-lo. Raycasting acontece em um mapa 2D No entanto, aqui o avião da câmera não é realmente um avião, mas uma linha, e é representado com um único vetor. O plano da câmera deve seja sempre perpendicular no vetor de direção. O plano da câmera representa a superfície da tela do computador, enquanto a direção O vetor é perpendicular a ele e aponta para dentro da tela. O posição do jogador, que é um único ponto, é um ponto em frente do plano da câmera. Um certo raio de uma certa coordenada x da tela, é então o raio que começa nesta posição do jogador, e passa por essa posição na tela ou assim a câmera avião.

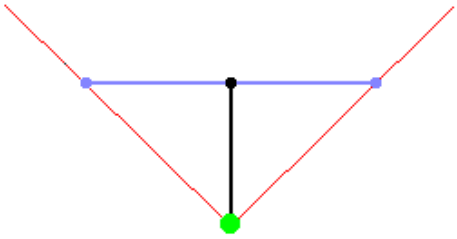


A imagem acima representa uma câmera 2D. O ponto verde é o posição (vetor "pos"). A linha preta, terminando na mancha preta, representa o vetor de direção (vetor "dir"), então a posição de O ponto preto é POS+DIR. A linha azul representa a câmera completa plano, o vetor do ponto preto para o ponto azul direito representa o vetor "plano", então a posição do azul direito ponto é pos+dir+plane, e a posição do ponto azul esquerdo é pos+dir-plane (todas essas são adições vetoriais).

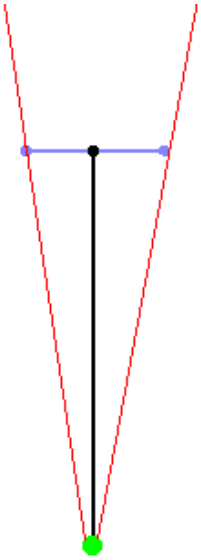
As linhas vermelhas na imagem são alguns raios. A direção desses raios é facilmente calculado fora da câmera: é a soma dos raios vetor de direção do camear, e uma parte do vetor plano de a câmera: por exemplo, o terceiro raio vermelho na imagem, vai através da parte direita do plano da câmera no ponto cerca de 1/3 de seu comprimento. Portanto, a direção desse raio é dir + plano*1/3. Essa direção do raio é o vetor rayDir e os componentes X e Y deste vetor são então usados pelo algoritmo DDA.

As duas linhas externas são a borda esquerda e direita da tela, e o ângulo entre essas duas linhas é chamado de Campo de Visão ou FOV. O FOV é determinado pela razão entre o comprimento do vetor de direção e o comprimento do plano. Aqui estão alguns exemplos de FOVs diferentes:

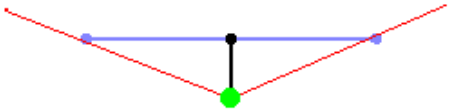
Se o vetor de direção e o vetor do plano da câmera tiverem o mesmo comprimento, o FOV será de 90°:



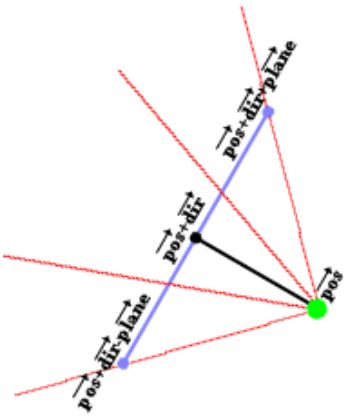
Se o vetor de direção for muito maior que o plano da câmera, o O FOV será muito menor que 90°, e você terá um visão estreita. Você verá tudo mais detalhado e lá será menos profundidade, então isso é o mesmo que aumentar o zoom:



Se o vetor de direção for menor que o plano da câmera, o FOV será maior que 90° (180° é o máximo, se o direcionamento é próximo de 0) e você terá um vetor muito mais amplo visão, como diminuir o zoom:



quando o jogador gira, a câmera tem que girar, então tanto o O vetor de direção e o vetor plano devem ser girados. Em seguida, o todos os raios também girarão automaticamente.



Para girar um vetor, multiplique-o pela matriz

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

Se você não sabe sobre vetores e matrizes, tente encontrar um tutorial com o Google, um apêndice sobre isso está planejado para este tutorial mais tarde.

Não há nada que o proíba de usar um plano de câmera que não seja perpendicular à direção, mas o resultado será semelhante a um mundo "distorcido".

Raycaster não texturizado

Baixe o código-fonte aqui: [raycaster_flat.cpp]

Para começar com o básico, começaremos com um raycaster sem textura. Este exemplo também inclui um contador de fps (quadros por segundo) e teclas de entrada com detecção de colisão para mover e girar.

O mapa do mundo é uma matriz 2D, onde cada valor representa um quadrado. Se o valor for 0, esse quadrado representa um vazio, quadrado passível de passo e, se o valor for maior que 0, ele representa uma parede com uma determinada cor ou textura. O mapa declarado aqui é muito pequeno, apenas 24 por 24 quadrados, e é definido diretamente no código. Para um jogo real, como Wolfenstein 3D, você usa um mapa maior e carregue-o de um arquivo. Todos os zeros no grade são espaços vazios, então basicamente você vê uma sala muito grande, com um parede ao redor (os valores 1), uma pequena sala dentro dele (os valores 2), alguns pilares (os valores 3) e um corredor com uma sala (o valores 4). Observe que este código ainda não está dentro de nenhuma função, coloque antes que a função principal comece.

```
#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}
};
```

As primeiras variáveis são declaradas: posX e posY representam o vetor de posição do jogador, dirX e dirY representam o direção do jogador, e planeX e planeY o plano da câmera de o jogador. Certifique-se de que o plano da câmera esteja perpendicular ao direção, mas você pode alterar o comprimento dela. A razão entre o comprimento da direção e o plano da câmera determinam o FOV, aqui o vetor de direção é um pouco mais longo que a câmera plano, de modo que o FOV será menor que 90° (mais precisamente, o FOV é 2 * atan (0,66 / 1,0) = 66 °, o que é perfeito para uma primeira pessoa jogo de tiro). Mais tarde, ao girar com as teclas de entrada, Os valores de dir e plane serão alterados, mas eles sempre permanecem perpendiculares e mantêm o mesmo comprimento.

As variáveis time e oldTime serão usadas para armazenar o tempo de o quadro atual e o anterior, a diferença de tempo entre Esses dois podem ser usados para determinar o quanto você deve se mover quando uma determinada tecla é pressionada (para mover uma velocidade constante, não importa como longo o cálculo dos quadros leva), e para o FPS balcão.

```
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
```

O resto da função principal começa agora. Primeiro, a tela é criada com um resolução de escolha. Se você escolher uma resolução grande, como 1280*1024, o efeito irá completamente lento, não porque o raycating algoritmo é lento, mas simplesmente porque o upload de uma tela inteira de a CPU para a placa de vídeo vai tão lento.

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

Depois de configurar a tela, o gameloop é iniciado, este é o loop que desenha um quadro inteiro e lê a entrada todas as vezes.

```
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).

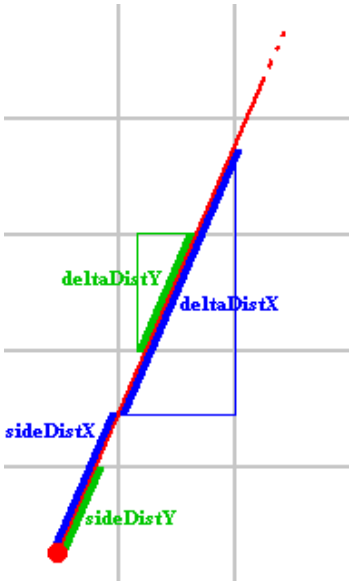
```
  for(int x = 0; x < w; x++)
  {
    //calculate ray position and direction
    double cameraX = 2 * x / double(w) - 1; //x-coordinate in camera space
    double rayDirX = dirX + planeX * cameraX;
    double rayDirY = dirY + planeY * cameraX;
```

No próximo código, mais variáveis são declaradas e calculadas, estes têm relevância para o algoritmo DDA:

mapX e mapY representam o quadrado atual do mapa em que o raio é em. A posição do raio em si é um número de ponto flutuante e contém ambas as informações sobre em qual quadrado do mapa estamos, e *onde* estamos nesse quadrado, mas mapX e mapY são apenas as coordenadas desse quadrado.

sideDistX e sideDistY são inicialmente a distância que o raio tem que Viaje de sua posição inicial para o primeiro lado X e o primeiro lado y. Mais tarde no código, eles serão incrementados enquanto as etapas são executadas.

deltaDistX e deltaDistY são a distância que o raio tem que percorrer Vá de 1 lado X para o próximo lado X, ou de 1 lado Y para o próximo lado y. A imagem a seguir mostra o sideDistX inicial, sideDistY e deltaDistX e deltaDistY:



Ao derivar deltaDistX geometricamente, você obtém, com Pitágoras, as fórmulas abaixo. Para o triângulo azul (deltaDistX), um lado tem comprimento 1 (pois é exatamente uma célula) e o outro tem comprimento raydirY / raydirX porque é exatamente a quantidade de unidades em que o raio entra a direção y ao dar 1 passo na direção X. Para o triângulo verde (deltaDistY), o fórmula é semelhante.

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

Mas isso pode ser simplificado para:

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

Onde |rayDir| é o comprimento do vetor rayDirX, rayDirY (ou seja, sqrt(rayDirX * rayDirX + rayDirY * rayDirY)): você pode realmente verificar que, por exemplo, sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX)) é igual a abs(sqrt(rayDirX * rayDirX + rayDirY * rayDirY) / rayDirX). No entanto, podemos usar 1 em vez de |rayDir|, porque apenas a *razão* entre deltaDistX e deltaDistY importa para o código DDA que segue mais adiante, então obtemos:

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

Devido a isso, os valores deltaDist e sideDist usados no código não correspondem aos comprimentos mostrados na figura acima, mas seus tamanhos relativos ainda correspondem.

[obrigado a Artem por detectar essa simplificação]

A variável perpWallDist será usada posteriormente para calcular o comprimento do raio.

O algoritmo DDA sempre saltará exatamente um quadrado a cada loop, um quadrado na direção x ou um quadrado na direção y. Se tiver que ir na direção x negativa ou positiva, e o A direção y negativa ou positiva dependerá da direção de o raio, e esse fato será armazenado no stepX e no stepY. Aqueles As variáveis são sempre -1 ou +1.

Finalmente, hit é usado para determinar se o loop de vinda é ou não pode ser terminado, e o lado conterá se um lado x ou um lado y de um parede foi atingida. Se um lado x foi atingido, o lado é definido como 0, se um lado y foi atingido, o lado será 1. Por lado x e lado y, quero dizer as linhas de a grade que são as bordas entre dois quadrados.

```
      //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?
```

NOTA: Se rayDirX ou rayDirY forem 0, a divisão até zero será evitada pela configuração para um valor muito alto 1e30. Se você estiver usando uma linguagem como C++, Java ou JS, isso não é realmente necessário, pois suporta o padrão de ponto flutuante IEEE 754, que dá o resultado Infinity, que funciona corretamente no código abaixo. No entanto, alguns outros como Python, não permitem a divisão até zero, portanto, o código mais genérico que funciona em todos os lugares é dado acima. 1e30 é um número suficientemente alto e escolhido arbitrariamente pode ser definido como Infinity se sua linguagem de programação suportar assiging esse valor.

Agora, antes que o DDA real possa ser iniciado, primeiro stepX, stepY e o sideDistX e sideDistY iniciais ainda precisam ser calculados.

Se a direção do raio tiver um componente x negativo, stepX é -1, se A direção do raio tem um componente X positivo é +1. Se o x-component é 0, não importa qual valor stepX tem desde então não será usado.
O mesmo vale para o componente y.

Se a direção do raio tiver um componente x negativo, sideDistX será o distância da posição inicial do raio para o primeiro lado para o esquerda, se a direção do raio tiver um componente X positivo, o primeiro lado à direita é usado em seu lugar.
O mesmo vale para o componente y, mas agora com o primeiro lado acima ou abaixo da posição.
Para esses valores, o valor inteiro mapX é usado e o valor real subtraída e 1,0 é adicionado em alguns dos casos dependendo se o lado à esquerda ou à direita, da parte superior ou do inferior é usado. Então você obtém a distância perpendicular a isso lado, então multiplique-o com deltaDistX ou deltaDistY para obter o real Distância euclidiana.

```
//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;
}
```

Agora o DDA real começa. É um loop que incrementa o raio com 1 quadrado de cada vez, até que uma parede seja atingida. Cada vez, ou salta um quadrado na direção x (com stepX) ou um quadrado na y (com stepY), ele sempre salta 1 quadrado de uma vez. Se o A direção de Ray seria a direção X, o loop só terá para saltar um quadrado na direção x todas as vezes, porque o raio vai Nunca mude sua direção y. Se o raio estiver um pouco inclinado para o y, então a cada tantos saltos na direção x, o raio

terá que pular um quadrado na direção y. Se o raio for exatamente na direção y, ele nunca precisa pular na direção x, etc...

sideDistX e sideDistY são incrementados com deltaDistX a cada salto em sua direção, e mapX e mapY são incrementados com stepX e stepY, respectivamente.

Quando o raio atinge uma parede, o loop termina e então saberemos se um lado X ou Y de uma parede foi atingido na variável "lado", e qual parede foi atingida com mapX e mapY. Nós não saberemos exatamente onde a parede foi atingida, no entanto, mas isso não é necessário em este caso porque não usaremos paredes texturizadas por enquanto.

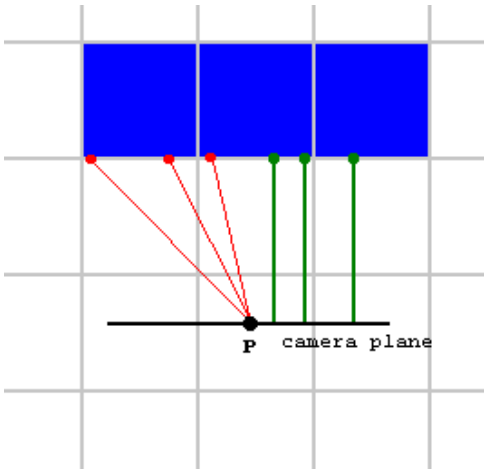
```
//perform DDA
while (hit == 0)
{
  //jump to next map square, either in x-direction, or in y-direction
  if (sideDistX < sideDistY)
  {
    sideDistX += deltaDistX;
    mapX += stepX;
    side = 0;
  }
  else
  {
    sideDistY += deltaDistY;
    mapY += stepY;
    side = 1;
  }
  //Check if ray has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

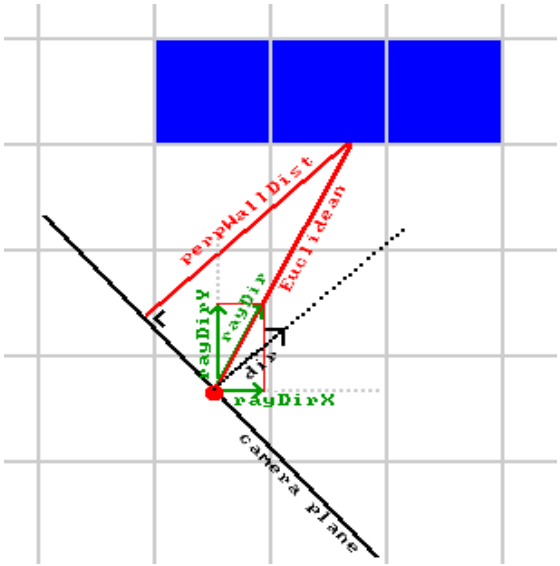
The following image shows why we take distance to camera plane instead of player. With P the player, and the black line the camera plane: To the left of the player, a few red rays are shown from hitpoints on the wall to the player, representing Euclidean distance. On the right side of the player, a few 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.

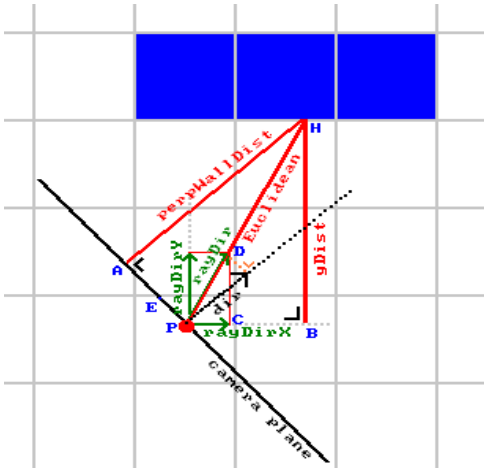
Meaning of the points:

- 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.

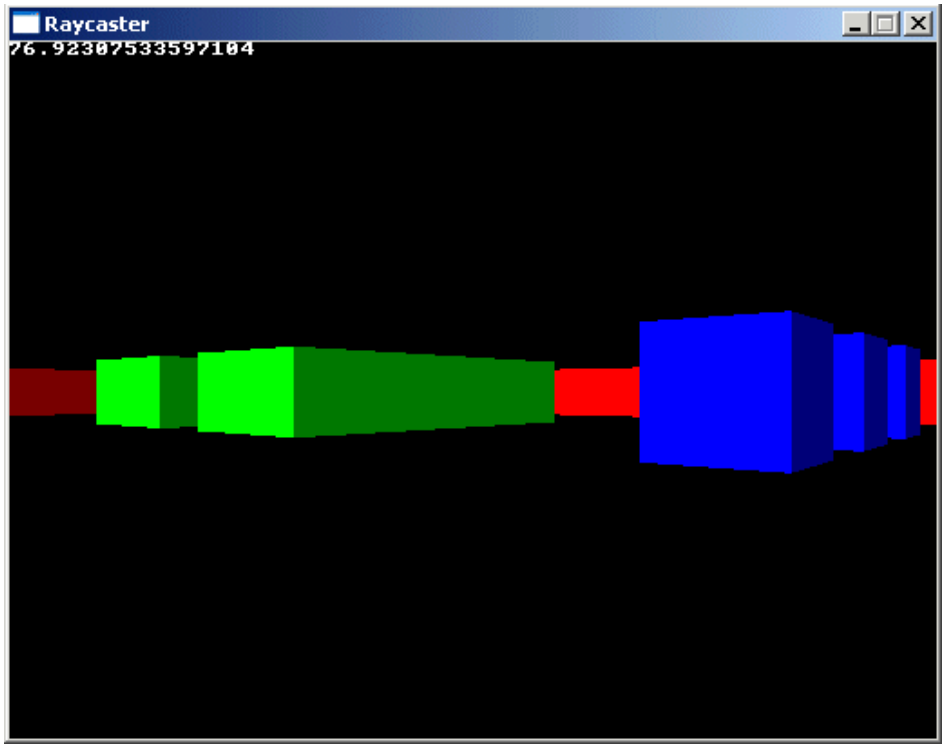
If the up arrow is pressed, the player will move forward: add dirX to posX, and dirY to posY. This assumes that dirX and dirY are normalized vectors (their length is 1), but they were initially set like this, so it's ok. There's also a simple collision detection built in, namely if the new position will 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.

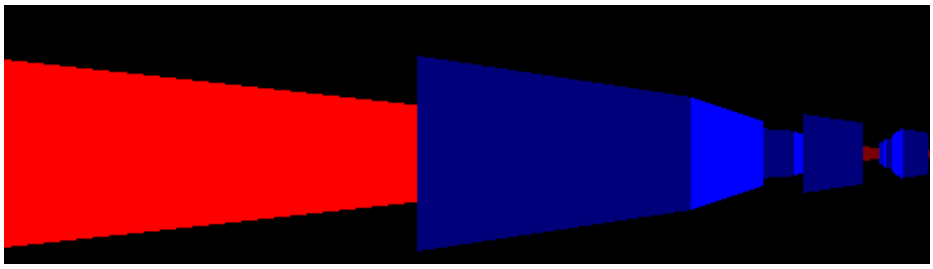
To rotate, if the left or right arrow is pressed, both the direction vector and plane vector are rotated by using the formulas of multiplication with the rotation matrix (and over the angle rotSpeed).

```
    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]=
{
  {4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,7,7,7,7,7,7,7,7},
  {4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},
  {4,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},
  {4,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},
  {4,0,3,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},
  {4,0,4,0,0,0,0,5,5,5,5,5,5,5,5,7,7,0,7,7,7,7,7},
  {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},
  {4,4,4,4,4,4,0,4,4,4,6,0,6,2,2,2,2,2,2,2,3,3,3,3},
  {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;
  }
}
```

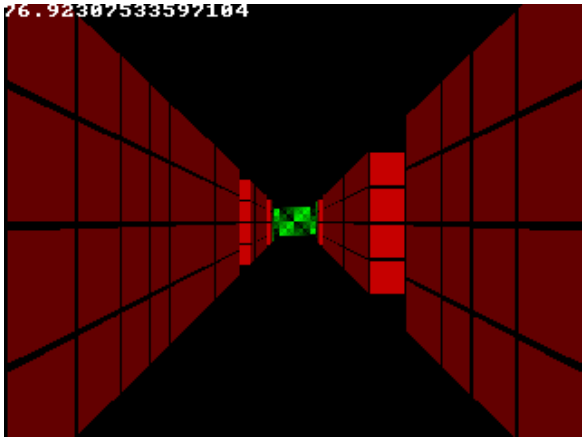
Now the buffer still has to be drawn, and after that it has to be cleared (where in the untextured version we simply had to use "cls". Ensure to do it in scanline order for speed thanks to memory locality for caching). The rest of this code is again the same.

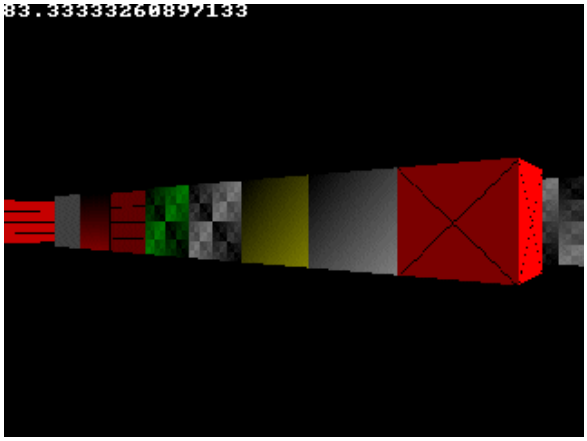
```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of cls()
//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds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

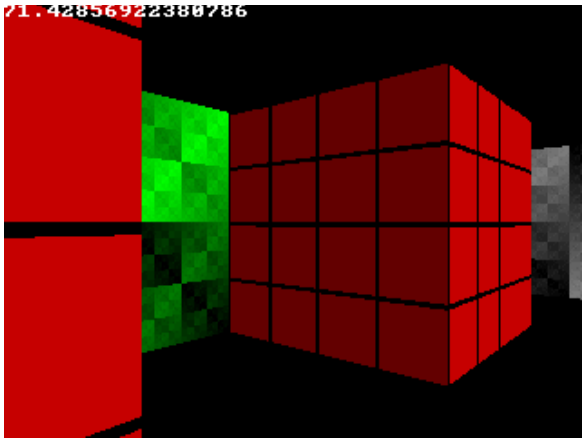
And here's again the keys, nothing has changed here either. If you like you can try to add strafe keys (to strafe to the left and right). These have to be made the same way as the up and down keys, but use planeX and planeY instead of dirX and dirY.

```
    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]);
```

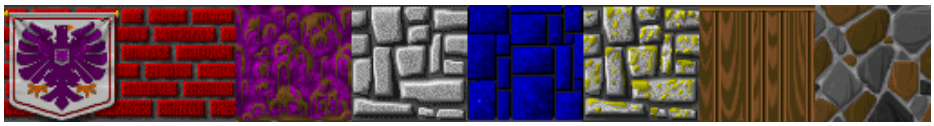
Or just swap X and Y where the textures are generated, but in many cases after loading an image or getting a texture from other formats you'll have it in scanlines anyway and have to swap it this way.

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