Return to initial translation

Tutoriel d'infographie de Lode

Raycasting

Table des matières

Retour à l'index

Introduction

Le Raycasting est une technique de rendu permettant de créer une perspective 3D dans une carte 2D. À l'époque où les ordinateurs étaient plus lents, il n'était pas possible de faire fonctionner de vrais moteurs 3D en temps réel, et le raycasting était la première solution. Le Raycasting peut aller très vite, car il suffit de faire un calcul pour chaque ligne verticale de l'écran. Le jeu le plus connu qui utilise cette technique est bien sûr Wolfenstein 3D.

Le moteur de raycasting de Wolfenstein 3D était très limité, lui permettant de fonctionner même sur un ordinateur 286 : tous les murs ont la même hauteur et sont des carrés orthogonaux sur une grille 2D, comme on peut le voir sur cette capture d'écran d'un mapeditor pour Wolf3D :

Des choses comme les escaliers, les sauts ou les différences de hauteur sont impossibles à faire avec ce moteur. Des jeux ultérieurs tels que Doom et Duke Nukem 3D utilisaient également le raycasting, mais des moteurs beaucoup plus avancés qui permettaient des murs inclinés, des hauteurs différentes, des sols et plafonds texturés, des murs transparents, etc... Les sprites (ennemis, objets et goodies) sont des images 2D , mais les sprites ne sont pas abordés dans ce didacticiel pour le moment.

Le ray *casting* n'est pas la même chose que le ray *tracing* ! Le Raycasting est une technique semi-3D rapide qui fonctionne en temps réel même sur des calculatrices graphiques à 4 MHz, tandis que le Raytracing est une technique de rendu réaliste qui prend en charge les reflets et les ombres dans de vraies scènes 3D, et ce n'est que récemment que les ordinateurs sont devenus assez rapides pour le faire en temps réel pour des durées raisonnablement élevées. résolutions et scènes complexes.

Le code des raycasters non texturés et texturés est complètement donné dans ce document, mais il est assez long, vous pouvez aussi télécharger le code à la place :

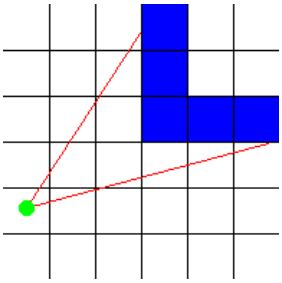
raycaster_flat.cpp
raycaster_textured.cpp

L'idée de base

L'idée de base du raycasting est la suivante : la carte est une grille carrée 2D, et chaque carré peut être soit 0 (= pas de mur), soit une valeur positive (= un

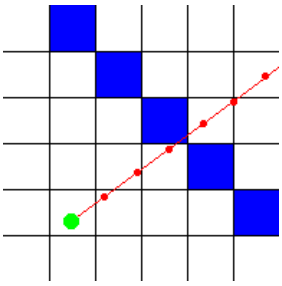
L'idée de base du raycasting est la suivante : la carte est une grille carrée 2D, et chaque carré peut être soit 0 (= pas de mur), soit une valeur positive (= un mur avec une certaine couleur ou texture).

Pour chaque x de l'écran (c'est-à-dire pour chaque bande verticale de l'écran), envoyez un rayon qui commence à l'emplacement du joueur et avec une direction qui dépend à la fois de la direction de regard du joueur et de la coordonnée x de l'écran. Ensuite, laissez ce rayon avancer sur la carte 2D, jusqu'à ce qu'il touche un carré de la carte qui est un mur. S'il heurte un mur, calculez la distance de ce point de vie au joueur, et utilisez cette distance pour calculer la hauteur à laquelle ce mur doit être dessiné à l'écran : plus le mur est éloigné, plus il est petit à l'écran, et plus il est proche , plus il semble être élevé. Ce sont tous des calculs 2D. Cette image montre un aperçu de haut en bas de deux de ces rayons (rouges) qui partent du joueur (point vert) et frappent les murs bleus :

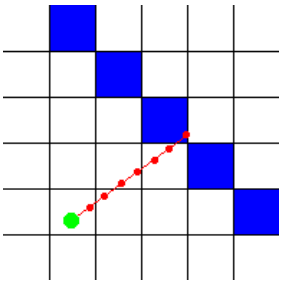


Pour trouver le premier mur qu'un rayon rencontre sur son chemin, il faut le laisser démarrer à la position du joueur, puis tout le temps, vérifier si le rayon est ou non à l'intérieur d'un mur. Si c'est à l'intérieur d'un mur (coup), alors la boucle peut s'arrêter, calculer la distance et dessiner le mur avec la bonne hauteur. Si la position du rayon n'est pas dans un mur, il faut le tracer plus loin : ajoutez une certaine valeur à sa position, dans le sens de la direction de ce rayon, et pour cette nouvelle position, vérifiez à nouveau s'il est à l'intérieur d'un mur ou non . Continuez ainsi jusqu'à ce qu'un mur soit finalement touché.

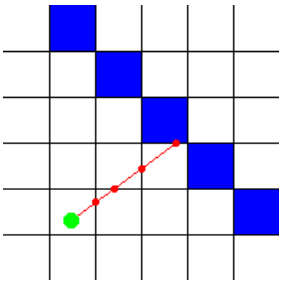
Un humain peut immédiatement voir où le rayon touche le mur, mais il est impossible de trouver quelle case le rayon touche immédiatement avec une seule formule, car un ordinateur ne peut vérifier qu'un nombre fini de positions sur le rayon. De nombreux raycasters ajoutent une valeur constante au rayon à chaque étape, mais il y a une chance qu'il manque un mur ! Par exemple, avec ce rayon rouge, sa position a été vérifiée à chaque point rouge :



comme vous pouvez le voir, le rayon traverse directement le mur bleu, mais l'ordinateur ne l'a pas détecté, car il n'a vérifié qu'aux positions avec le rouge points. Plus vous vérifiez de positions, moins il y a de chances que l'ordinateur détecte un mur, mais plus il faut de calculs. Ici, la distance de pas a été réduite de moitié, alors maintenant il détecte que le rayon a traversé un mur, bien que la position ne soit pas



Pour une précision infinie avec cette méthode, il faudrait un pas infiniment petit, et donc un nombre infini de calculs ! C'est plutôt mauvais, mais heureusement, il existe une meilleure méthode qui ne nécessite que très peu de calculs et qui détectera pourtant tous les murs : l'idée est de vérifier de chaque côté d'un mur que le rayon rencontrera. Nous donnons à chaque carré une largeur de 1, donc chaque côté d'un mur est une valeur entière et les endroits intermédiaires ont une valeur après le point. Maintenant, la taille du pas n'est pas constante, elle dépend de la distance jusqu'au côté suivant :

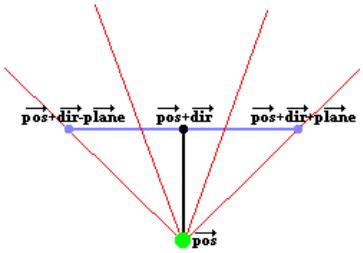


Comme vous pouvez le voir sur l'image ci-dessus, le rayon frappe le mur exactement où nous le voulons. De la manière présentée dans ce tutoriel, un algorithme est utilisé qui est basé sur DDA ou "Digital Differential Analysis". DDA est un algorithme rapide généralement utilisé sur les grilles carrées pour trouver les cases auxquelles une ligne correspond (par exemple pour tracer une ligne sur un écran, qui est une grille de pixels carrés). Nous pouvons donc également l'utiliser pour trouver les carrés de la carte touchés par notre rayon et arrêter l'algorithme une fois qu'un carré qui est un mur est touché.

Certains traceurs de rayons fonctionnent avec des angles euclidiens pour représenter la direction du joueur et des rayons, et déterminent le champ de

vision avec un autre angle. J'ai trouvé cependant qu'il est beaucoup plus facile de travailler avec des vecteurs et une caméra à la place : la position du joueur est toujours un vecteur (une coordonnée x et ay), mais maintenant, nous faisons aussi de la direction un vecteur : donc la direction est maintenant déterminée par deux valeurs : les coordonnées x et y de la direction. Un vecteur de direction peut être vu comme suit : si vous tracez une ligne dans la direction dans laquelle le joueur regarde, passant par la position du joueur, alors chaque point de la ligne est la somme de la position du joueur et un multiple de la direction vecteur. La longueur d'un vecteur de direction n'a pas vraiment d'importance, seule sa direction.

Cette méthode avec des vecteurs nécessite également un vecteur supplémentaire, qui est le vecteur du plan de la caméra. Dans un vrai moteur 3D, il y a aussi un plan caméra, et là ce plan est vraiment un plan 3D donc deux vecteurs (u et v) sont nécessaires pour le représenter. Cependant, le raycasting se produit dans une carte 2D, donc ici le plan de la caméra n'est pas vraiment un plan, mais une ligne, et est représenté par un seul vecteur. Le plan de la caméra doit toujours être perpendiculaire au vecteur de direction. Le plan de la caméra représente la surface de l'écran de l'ordinateur, tandis que le vecteur de direction lui est perpendiculaire et pointe à l'intérieur de l'écran. La position du joueur, qui est un point unique, est un point devant le plan de la caméra. Un certain rayon d'une certaine abscisse de l'écran, est alors le rayon qui commence à cette position du joueur,

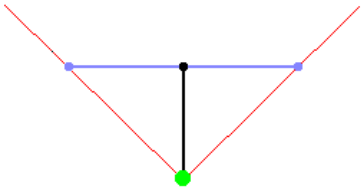


L'image ci-dessus représente une telle caméra 2D. La tache verte est la position (vecteur "pos"). La ligne noire, se terminant par le point noir, représente le vecteur de direction (vecteur "dir"), donc la position du point noir est pos+dir. La ligne bleue représente le plan complet de la caméra, le vecteur du point noir au point bleu droit représente le vecteur "plan", donc la position du point bleu droit est pos+dir+plane, et la position du point bleu gauche est pos+dir-plane (ce sont tous des ajouts vectoriels).

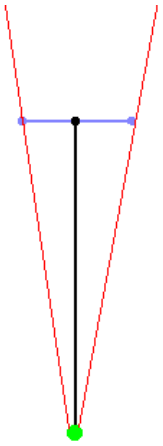
Les lignes rouges dans l'image sont quelques rayons. La direction de ces rayons se calcule facilement hors de la caméra : c'est la somme du vecteur direction de la caméra, et d'une partie du vecteur plan de la caméra : par exemple le troisième rayon rouge sur l'image, passe par la partie droite du plan de la caméra au point d'environ 1/3 de sa longueur. Donc la direction de ce rayon est dir + plan*1/3. Cette direction de rayon est le vecteur rayDir, et les composantes X et Y de ce vecteur sont ensuite utilisées par l'algorithme DDA.

Les deux lignes extérieures sont les bordures gauche et droite de l'écran, et l'angle entre ces deux lignes s'appelle le champ de vision ou FOV. Le FOV est déterminé par le rapport entre la longueur du vecteur de direction et la longueur du plan. Voici quelques exemples de différents FOV :

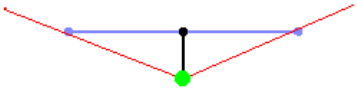
Si le vecteur de direction et le vecteur du plan de la caméra ont la même longueur, le champ de vision sera de 90 ° :



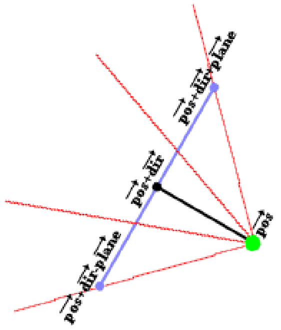
si le vecteur de direction est beaucoup plus long que le plan de la caméra, le champ de vision sera beaucoup plus petit que 90 °, et vous aurez un champ de vision très vision étroite. Vous verrez tout plus détaillé cependant et il y aura moins de profondeur, donc c'est la même chose qu'un zoom avant :



Si le vecteur de direction est plus court que le plan de la caméra, le FOV sera supérieur à 90° (180° est le maximum, si le vecteur de direction est proche de 0), et vous aurez une vision beaucoup plus large, comme un zoom arrière :



lorsque le joueur tourne, la caméra doit tourner, donc le vecteur de direction et le vecteur plan doivent être tournés. Ensuite, les rayons tourneront tous automatiquement.



Pour faire pivoter un vecteur, multipliez-le par la matrice de rotation

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

Si vous ne connaissez pas les vecteurs et les matrices, essayez de trouver un tutoriel avec google, une annexe à ce sujet est prévue pour ce tutoriel plus tard .

Rien ne vous interdit d'utiliser un plan de caméra qui n'est pas perpendiculaire à la direction, mais le résultat ressemblera à un monde "incliné".

Raycaster non texturé

Téléchargez le code source ici : raycaster_flat.cpp

Pour commencer par les bases, nous allons commencer avec un raycaster non texturé. Cet exemple comprend également un compteur de fps (images par seconde) et des touches d'entrée avec détection de collision pour se déplacer et pivoter.

La carte du monde est un tableau 2D, où chaque valeur représente un carré. Si la valeur est 0, ce carré représente un carré vide et accessible, et si la valeur est supérieure à 0, il représente un mur avec une certaine couleur ou texture. La carte déclarée ici est très petite, seulement 24 par 24 carrés, et est définie directement dans le code. Pour un vrai jeu, comme Wolfenstein 3D, vous utilisez une carte plus grande et la chargez à partir d'un fichier à la place. Tous les zéros de la grille sont des espaces vides, donc en gros vous voyez une très grande pièce, avec un mur autour (les valeurs 1), une petite pièce à l'intérieur (les valeurs 2), quelques piliers (les valeurs 3), et un couloir avec une chambre (les valeurs 4). Notez que ce code n'est pas encore à l'intérieur d'une fonction, placez-le avant le démarrage de la fonction principale.

```
#define mapLargeur 24
#define mapHauteur 24
#define screenWidth 640
#define screenHeight 480

int mappemonde[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}
};
```

Quelques premières variables sont déclarées : posX et posY représentent le vecteur position du joueur, dirX et dirY représentent la direction du joueur, et planeX et planeY le plan caméra du joueur. Assurez-vous que le plan de la caméra est perpendiculaire à la direction, mais vous pouvez en modifier la longueur. Le rapport entre la longueur de la direction et le plan de la caméra détermine le FOV, ici le vecteur de direction est un peu plus long que le plan de la caméra, donc le FOV sera inférieur à 90° (plus précisément, le FOV est 2 * atan(0.66 /1.0)=66°, ce qui est parfait pour un jeu de tir à la première personne). Plus tard, lors de la rotation avec les touches d'entrée, les valeurs de dir et plane seront modifiées, mais elles resteront toujours perpendiculaires et garderont la même longueur.

Les variables time et oldTime seront utilisées pour stocker l'heure de l'image actuelle et de l'image précédente, la différence de temps entre ces deux peut être utilisée pour déterminer combien vous devez vous déplacer lorsqu'une certaine touche est enfoncée (pour vous déplacer à une vitesse constante, peu importe combien de temps le calcul des images prend), et pour le compteur FPS.

```
int main(int /*argc*/, char */*argv*/ [])
{
  double posX = 22, posY = 12 ;  //x et y position de départ
   double dirX = -1, dirY = 0; //vecteur direction initial
   double planeX = 0, planeY = 0.66; //la version 2d raycaster du plan caméra

  double time = 0; //heure de la trame courante
   double oldTime = 0; //heure de l'image précédente
```

Le reste de la fonction principale commence maintenant. Tout d'abord, l'écran est créé avec une résolution de choix. Si vous choisissez une grande résolution, comme 1280*1024, l'effet sera assez lent, non pas parce que l'algorithme de raycating est lent, mais simplement parce que le téléchargement d'un écran entier du processeur vers la carte vidéo est si lent.

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

Après avoir configuré l'écran, la boucle de jeu démarre, c'est la boucle qui dessine une image entière et lit l'entrée à chaque fois.

```
  tandis que (! fait ())
  {
```

Ici commence le raycasting proprement dit. La boucle de raycasting est une boucle for qui passe par tous les x, il n'y a donc pas de calcul pour chaque pixel de l'écran, mais seulement pour chaque bande verticale, ce qui n'est pas grand-chose du tout ! Pour commencer la boucle de raycasting, certaines variables sont supprimées et calculées :

Le rayon commence à la position du joueur (posX, posY).

cameraX est la coordonnée x sur le plan de la caméra que représente la coordonnée x actuelle de l'écran, de cette manière de sorte que le côté droit de l'écran obtienne la coordonnée 1, le centre de l'écran obtienne la coordonnée 0 et le côté gauche de l'écran obtient la coordonnée -1. À partir de cela, la direction du rayon peut être calculée comme expliqué précédemment : comme la somme du vecteur de direction et d'une partie du vecteur plan. Cela doit être fait à la fois pour les coordonnées x et y du vecteur (puisque l'ajout de deux vecteurs consiste à ajouter leurs coordonnées x et à ajouter leurs coordonnées y).

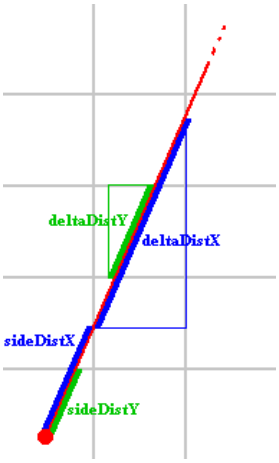
```
    for(int x = 0; x < w; x++)
    {
      //calculer la position et la direction
       du rayon double cameraX = 2 * x / double(w) - 1; //coordonnée x dans l'espace caméra
       double rayDirX = dirX + planeX * cameraX;
      double rayonDirY = dirY + planeY * caméraX ;
```

Dans le morceau de code suivant, d'autres variables sont déclarées et calculées, celles-ci sont pertinentes pour l'algorithme DDA :

mapX et mapY représentent le carré actuel de la carte dans laquelle se trouve le rayon. La position du rayon elle-même est un nombre à virgule flottante et contient à la fois des informations sur dans quel carré de la carte nous sommes, et *où* dans ce carré nous nous trouvons, mais mapX et mapY ne sont que les coordonnées de ce carré.

sideDistX et sideDistY sont initialement la distance que le rayon doit parcourir depuis sa position de départ jusqu'au premier côté x et au premier côté y. Plus tard dans le code, ils seront incrémentés au fur et à mesure des étapes.

deltaDistX et deltaDistY sont la distance que le rayon doit parcourir pour passer du côté x au côté x suivant, ou du côté y au côté y suivant. L'image suivante montre les sideDistX, sideDistY et deltaDistX et deltaDistY initiaux :



lors de la dérivation géométrique de deltaDistX, vous obtenez, avec Pythagoras, les formules ci-dessous. Pour le triangle bleu (deltaDistX), un côté a la longueur 1 (car il s'agit exactement d'une cellule) et l'autre a la longueur raydirY / raydirX car c'est exactement le nombre d'unités que le rayon va dans la direction y en prenant 1 pas dans la direction X. Pour le triangle vert (deltaDistY), la formule est similaire.

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

Mais cela peut être simplifié en :

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

Où |rayDir| est la longueur du vecteur rayDirX, rayDirY (c'est-à-dire sqrt(rayDirX * rayDirX + rayDirY * rayDirY)) : vous pouvez en effet vérifier que par exemple sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX)) est égal à abs(sqrt (rayDirX * rayDirX + rayDirY * rayDirY) / rayDirX). Cependant, nous pouvons utiliser 1 au lieu de |rayDir|, car seul le *rapport* entre deltaDistX et deltaDistY compte pour le code DDA qui suit plus loin ci-dessous, nous obtenons donc :

deltaDistX = abs(1 / rayonDirX)
deltaDistY = abs(1 / rayonDirY)

Pour cette raison, les valeurs deltaDist et sideDist utilisées dans le code ne correspondent pas aux longueurs indiquées dans l'image ci-dessus, mais leurs tailles relatives correspondent toujours.

[merci à Artem d'avoir repéré cette simplification]

La variable perpWallDist sera utilisée plus tard pour calculer la longueur du rayon.

L'algorithme DDA sautera toujours exactement d'un carré à chaque boucle, soit un carré dans la direction x, soit un carré dans la direction y. S'il doit aller dans la direction x négative ou positive, et la direction y négative ou positive dépendra de la direction du rayon, et ce fait sera stocké dans stepX et stepY. Ces variables sont toujours soit -1 ou +1.

Enfin, hit est utilisé pour déterminer si la boucle à venir peut être terminée ou non, et side contiendra si un côté x ou un côté y d'un mur a été touché. Si un côté x a été touché, le côté est défini sur 0, si un côté y a été touché, le côté sera 1. Par côté x et côté y, j'entends les lignes de la grille qui sont les frontières entre deux carrés .

```
//dans quelle case de la carte nous sommes
 int mapX = int(posX);
int mapY = int (posY);

//longueur du rayon de la position actuelle au côté x ou y suivant
 double sideDistX ;
double sideDistY ;

 //longueur du rayon d'un côté x ou y au côté x ou y suivant
 double deltaDistX = (rayDirX == 0) ? 1e30 : std::abs(1 / rayDirX);
double deltaDisY = (rayDirY == 0) ? 1e30 : std::abs(1 / rayDirY);
double perpWallDist ;

//dans quelle direction aller dans la direction x ou y (soit +1 ou -1)
 int stepX ;
int stepY ;

int hit = 0 ; // y a-t-il eu un coup de mur ?
côté int ; // est-ce qu'un mur NS ou EW a été touché ?
```

REMARQUE : Si rayDirX ou rayDirY sont 0, la division par zéro est évitée en la définissant sur une valeur très élevée 1e30. Si vous utilisez un langage tel que C++, Java ou JS, ce n'est pas vraiment nécessaire, car il prend en charge la norme à virgule flottante IEEE 754, ce qui donne le résultat Infinity, qui fonctionne correctement dans le code ci-dessous. Cependant, certains autres langages, tels que Python, interdisent la division par zéro, donc le code plus générique qui fonctionne partout est donné ci-dessus. 1e30 est un nombre suffisamment élevé choisi arbitrairement et peut être défini sur Infinity si votre langage de programmation prend en charge l'attribution de cette valeur.

Maintenant, avant que le DDA réel puisse démarrer, les premiers stepX, stepY et les sideDistX et sideDistY initiaux doivent encore être calculés.

Si la direction du rayon a une composante x négative, stepX est -1, si la direction du rayon a une composante x positive, c'est +1. Si le composant x est 0, la valeur de stepX n'a pas d'importance puisqu'elle sera alors inutilisée.
Il en va de même pour la composante y.

Si la direction du rayon a une composante x négative, sideDistX est la distance entre la position de départ du rayon et le premier côté à gauche, si la direction du rayon a une composante x positive, le premier côté à droite est utilisé à la place.
Il en va de même pour le composant y, mais maintenant avec le premier côté au-dessus ou en dessous de la position.
Pour ces valeurs, la valeur entière mapX est utilisée et la position réelle en est soustraite, et 1.0 est ajouté dans certains cas selon que le côté gauche ou droit, du haut ou du bas est utilisé. Ensuite, vous obtenez la distance perpendiculaire à ce côté, alors multipliez-la avec deltaDistX ou deltaDistY pour obtenir la distance euclidienne réelle.

```
//calculer le pas et la sideDist initiale
 si (rayDirX < 0)
{
  stepX = -1 ;
  sideDistX = (posX - mapX) * deltaDistX ;
}
autre
{
  étapeX = 1 ;
  sideDistX = (mapX + 1.0 - posX) * deltaDistX ;
}
```

```
si (rayDirY < 0)
{
  étapeY = -1 ;
  sideDistY = (posY - mapY) * deltaDistY ;
}
autre
{
  étapeY = 1 ;
  sideDistY = (mapY + 1.0 - posY) * deltaDistY ;
}
```

Maintenant, le DDA proprement dit commence. C'est une boucle qui incrémente le rayon de 1 carré à chaque fois, jusqu'à ce qu'un mur soit touché. A chaque fois, qu'il saute d'un carré dans la direction x (avec stepX) ou d'un carré dans la direction y (avec stepY), il saute toujours 1 case à la fois. Si la direction du rayon était la direction x, la boucle n'aurait qu'à sauter d'un carré dans la direction x à chaque fois, car le rayon ne changerait jamais sa direction y. Si le rayon est un peu incliné dans la direction y, alors tous les sauts dans la direction x, le rayon devra sauter d'un carré dans la direction y.

Si le rayon est exactement dans la direction y, il n'a jamais à sauter dans la direction x, etc. .

Lorsque le rayon a touché un mur, la boucle se termine, et nous saurons alors si un côté x ou y d'un mur a été touché dans la variable "side", et quel mur a été touché avec mapX et mapY. Cependant, nous ne saurons pas exactement où le mur a été touché, mais ce n'est pas nécessaire dans ce cas car nous n'utiliserons pas de murs texturés pour le moment.

```
//exécute DDA
 tant que (appuyez sur == 0)
{
  //passe au carré suivant de la carte, soit dans la direction x, soit dans la direction y
   if (sideDistX < sideDistY)
  {
    sideDistX += deltaDistX ;
    carteX += étapeX ;
    côté = 0 ;
  }
  autre
  {
    sideDistY += deltaDistY ;
    cartes + = stepY ;
    côté = 1 ;
  }
  //Vérifier si le rayon a touché un mur
   if (worldMap[mapX][mapY] > 0) hit = 1 ;
}
```

Une fois le DDA terminé, nous devons calculer la distance du rayon au mur, afin de pouvoir calculer la hauteur du mur à dessiner après cela.

Nous n'utilisons pas la distance euclidienne au point représentant le joueur, mais plutôt la distance au plan de la caméra (ou, la distance du point projeté sur la direction de la caméra au joueur), pour éviter l'effet fisheye. L'effet fisheye est un effet que vous voyez si vous utilisez la distance réelle, où tous les murs s'arrondissent et peuvent vous rendre malade si vous tournez.

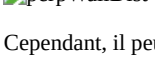
L'image suivante montre pourquoi nous prenons la distance au plan de la caméra au lieu du joueur. Avec P le joueur et la ligne noire le plan de la caméra : à gauche du joueur, quelques rayons rouges sont affichés à partir des repères sur le mur jusqu'au joueur, représentant la distance euclidienne. Sur le côté droit du joueur, quelques rayons verts sont affichés allant des points de vie sur le mur directement au plan de la caméra au lieu du joueur. Ainsi, les longueurs de ces lignes vertes sont des exemples de la distance perpendiculaire que nous utiliserons à la place de la distance euclidienne directe.

Dans l'image, le joueur regarde directement le mur, et dans ce cas, vous vous attendez à ce que le bas et le haut du mur forment une ligne parfaitement horizontale sur l'écran. Cependant, les rayons rouges ont tous une longueur différente, il faudrait donc calculer différentes hauteurs de mur pour différentes bandes verticales, d'où l'effet arrondi. Les rayons verts à droite ont tous la même longueur, ils donneront donc le bon résultat. La même chose s'applique toujours lorsque le joueur tourne (alors le plan de la caméra n'est plus horizontal et les lignes vertes auront des longueurs différentes, mais toujours avec un changement constant entre chacune) et les murs deviennent des lignes diagonales mais droites sur l'écran. Cette explication est un peu vague mais donne l'idée.


perpWallDist

Notez que cette partie du code n'est pas une "correction fisheye", une telle correction n'est pas nécessaire pour le mode de raycasting utilisé ici, l'effet fisheye est simplement évité par la façon dont la distance est calculée ici. Il est encore plus facile de calculer cette distance perpendiculaire que la distance réelle, nous n'avons même pas besoin de connaître l'endroit exact où le mur a été touché.

Cette distance perpendiculaire est appelée "perpWallDist" dans le code. Une façon de le calculer consiste à utiliser la formule de la distance la plus courte entre un point et une ligne, où le point est l'endroit où le mur a été touché et la ligne est le plan de la caméra :


perpWallDist

Cependant, il peut être calculé plus simplement que cela : en raison de la façon dont deltaDist et sideDist ont été mis à l'échelle par un facteur de |rayDir| ci-dessus, la longueur de sideDist est déjà presque égale à perpWallDist. Nous avons juste besoin de soustraire deltaDist une fois, en reculant d'un pas, car dans les étapes DDA ci-dessus, nous sommes allés un peu plus loin pour nous retrouver à l'intérieur du mur.

Selon que le rayon touche un côté X ou un côté Y, la formule est calculée à l'aide de sideDistX ou de sideDistY.

```
//Calculer la distance projetée dans la direction de la caméra (la distance euclidienne donnerait un effet fisheye !)
 if(side == 0) perpWallDist = (sideDistX - deltaDistX);
sinon perpWallDist = (sideDistY - deltaDistY);
```

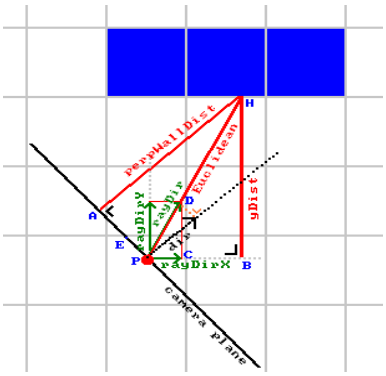
Une dérivation plus détaillée de la formule perpWallDist est illustrée dans l'image ci-dessous, pour le côté == 1 cas.

Signification des points :

- P : position du joueur, (posX, posY) dans le code
- H : repère du rayon sur le mur. Sa position y est connue pour être mapY + (1 - stepY) / 2
- yDist correspond à "(mapY + (1 - stepY) / 2 - posY)", c'est la coordonnée y du vecteur de distance euclidienne, en coordonnées universelles. Ici, (1 - stepY) / 2) est un terme de correction qui est 0 ou 1 basé sur la direction y positive ou négative, qui est également utilisé dans l'initialisation de sideDistY.
- dir : la direction de recherche du joueur principal, donnée par dirX, dirY dans le code. La longueur de ce vecteur est toujours exactement 1. Cela correspond à la direction du regard au centre de l'écran, par opposition à la direction du rayon actuel. Il est perpendiculaire au plan de la caméra et perpWallDist est parallèle à celui-ci.
- ligne pointillée orange (peut être difficile à voir, utilisez CTRL + molette de défilement ou CTRL + plus pour zoomer dans un navigateur de bureau pour mieux le voir): la valeur qui a été ajoutée à dir pour obtenir rayDir. Surtout, ceci est parallèle au plan de la caméra, perpendiculaire à dir.
- A : point du plan de la caméra le plus proche de H, point d'intersection de perpWallDist avec le plan de la caméra
- B : point de l'axe X passant par le joueur le plus proche de H, point où yDist croise l'axe X du monde passant par le joueur
- C : point à la position du joueur + rayDirX
- D: point at player position + rayDir.
- E : C'est le point D avec le vecteur dir soustrait, en d'autres termes, E + dir = D.
- les points A, B, C, D, E, H et P sont utilisés dans l'explication ci-dessous : ils forment des triangles qui sont considérés : BHP, CDP, AHP et DEP.

La dérivation proprement dite :

- 1 : Les triangles PBH et PCD ont la même forme mais des tailles différentes, donc les mêmes rapports d'arêtes
- 2 : Étant donné l'étape 1, les triangles montrent que le rapport yDist / rayDirY est égal au rapport Euclidien / |rayDir|, donc maintenant nous pouvons dériver perpWallDist = Euclidien / |rayDir| Au lieu.
- 3 : Les triangles AHP et EDP ont la même forme mais des tailles différentes, donc les mêmes rapports d'arêtes. La longueur du bord ED, c'est-à-dire |ED|, est égale à la longueur de dir, |dir|, qui est 1. De même, |DP| est égal à |rayDir|.
- 4 : Étant donné l'étape 3, les triangles montrent que le rapport euclidien / |rayDir| = perpWallDist / |dir| = perpWallDist / 1.
- 5 : La combinaison des étapes 4 et 2 montre que perpWallDist = yDist / rayDirY, où yDist est mapY + (1 - stepY) / 2) - posY
- 6 : Dans le code, sideDistY - deltaDistY, après les étapes DDA, est égal à (posY + (1 - stepY) / 2 - mapY) * deltaDistY (étant donné que sideDistY est calculé à partir de posY et mapY), donc yDist = (sideDistY - deltaDistY) / deltaDisTY
- 7 : Sachant que deltaDistY = 1 / |rayDirY|, l'étape 6 donne que yDist = (sideDistY - deltaDistY) * |rayDirY|
- 8 : La combinaison des étapes 5 et 7 donne perpWallDist = yDist / rayDirY = (sideDistY - deltaDistY) / |rayDirY| /rayDirY.
- 9 : Étant donné la façon dont les cas pour les signes de sideDistY et deltaDistY dans le code sont traités, la valeur absolue n'a pas d'importance et est égale à (sideDistY - deltaDistY), qui est la formule utilisée



[Merci à Thomas van der Berg en 2016 pour avoir signalé des simplifications du code (perpWallDist pourrait être simplifié et la valeur réutilisée pour wallX).
[Merci à Roux Morgan en 2020 pour avoir aidé à clarifier l'explication de perpWallDist, le tutoriel manquait d'informations avant cela]
[Merci à Noah Wagner et Elias pour avoir trouvé d'autres simplifications pour perpWallDist]

Maintenant que nous avons la distance calculée (perpWallDist), nous pouvons calculer la hauteur de la ligne qui doit être tracée à l'écran : c'est l'inverse de perpWallDist, puis multiplié par h, la hauteur en pixels de l'écran, pour amener aux coordonnées en pixels. Vous pouvez bien sûr aussi le multiplier par une autre valeur, par exemple 2*h, si vous voulez que les murs soient plus hauts ou plus bas. La valeur de h fera ressembler les murs à des cubes de hauteur, largeur et profondeur égales, tandis que de grandes valeurs créeront des boîtes plus hautes (selon votre moniteur).

Ensuite, à partir de cette lineHeight (qui est donc la hauteur de la ligne verticale qui doit être tracée), la position de début et de fin de l'endroit où nous devons vraiment dessiner est calculée. Le centre du mur doit être au centre de l'écran, et si ces points se trouvent en dehors de l'écran, ils sont plafonnés à 0 ou h-1.

```
//Calculer la hauteur de la ligne à dessiner à l'écran
 int lineHeight = (int)(h / perpWallDist);

// calcule le pixel le plus bas et le plus haut pour remplir la bande actuelle
 int drawStart = -lineHeight / 2 + h / 2;
si(drawStart < 0)drawStart = 0 ;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h)drawEnd = h - 1;
```

Enfin, en fonction du numéro du mur touché, une couleur est choisie. Si un côté y a été touché, la couleur est rendue plus sombre, cela donne un effet plus agréable. Et puis la ligne verticale est dessinée avec la commande verLine. Cela termine la boucle de raycasting, après avoir fait cela pour chaque x au moins.

```
//choisissez la couleur du mur
 ColorRGB color;
switch(worldMap[mapX][mapY])
```

```
      switch(worldMap[mapX][mapY])
      {
        cas 1 : couleur = RGB_Red ; Pause; //
         cas rouge 2 : couleur = RGB_Green ; Pause; //
         cas vert 3 : color = RGB_Blue ; Pause; //
         cas bleu 4 : couleur = RGB_White ; Pause; //blanc
         par défaut : couleur = RGB_Yellow ; Pause; //jaune
       }

      // donne aux côtés x et y une luminosité différente
       if (side == 1) {color = color / 2;}

      //dessine les pixels de la bande sous la forme d'une ligne verticale
       verLine(x, drawStart, drawEnd, color);
    }
```

Une fois la boucle de raycasting terminée, le temps de l'image actuelle et de l'image précédente est calculé, le FPS (images par seconde) est calculé et imprimé, et l'écran est redessiné de sorte que tout (tous les murs et la valeur du fps compteur) devient visible. Après cela, le backbuffer est effacé avec cls(), de sorte que lorsque nous dessinons à nouveau les murs de l'image suivante, le sol et le plafond seront à nouveau noirs au lieu de contenir encore des pixels de l'image précédente.

Les modificateurs de vitesse utilisent frameTime et une valeur constante pour déterminer la vitesse de déplacement et de rotation des touches d'entrée. Grâce à l'utilisation de frameTime, nous pouvons nous assurer que la vitesse de déplacement et de rotation est indépendante de la vitesse du processeur.

```
    //minutage pour l'entrée et le compteur FPS
     oldTime = time ;
    temps = getTicks();
    double frameTime = (heure - oldTime) / 1000.0 ; //frameTime est le temps que cette image a pris, en secondes
     print(1.0 / frameTime); // compteur FPS
     redessiné();
    cls();

    // modificateurs de vitesse
     double moveSpeed = frameTime * 5.0 ; //la valeur constante est en carrés/seconde
     double rotSpeed = frameTime * 3.0; //la valeur constante est en radians/seconde
```

La dernière partie est la partie d'entrée, les clés sont lues.

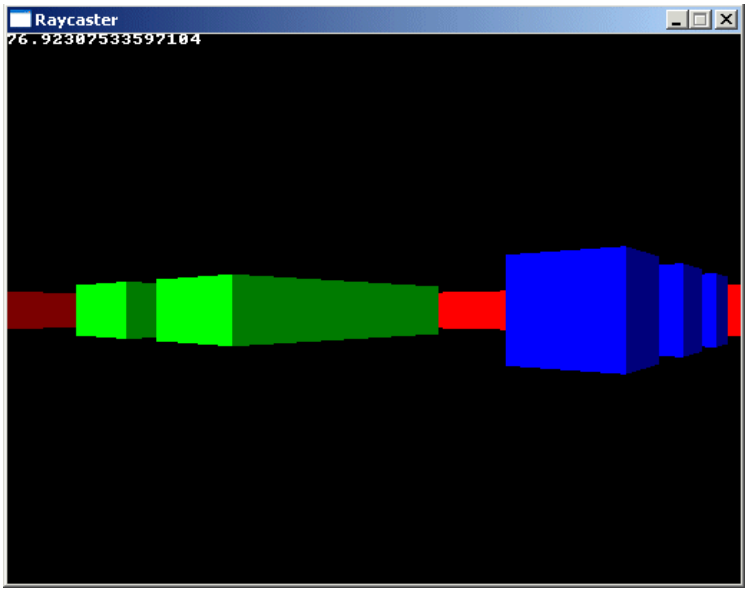
Si la flèche vers le haut est enfoncée, le joueur avancera : ajoutez dirX à posX et dirY à posY. Cela suppose que dirX et dirY sont des vecteurs normalisés (leur longueur est de 1), mais ils ont été initialement définis comme ceci, donc ça va. Il y a aussi une simple détection de collision intégrée, à savoir si la nouvelle position sera à l'intérieur d'un mur, vous ne bougerez pas. Cette détection de collision peut cependant être améliorée, par exemple en vérifiant si un cercle autour du joueur n'ira pas à l'intérieur du mur au lieu d'un seul point.

La même chose est faite si vous appuyez sur la flèche vers le bas, mais la direction est alors soustraite à la place.

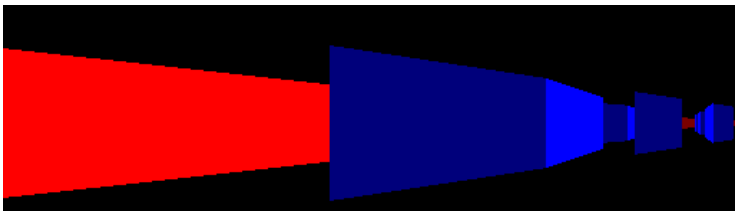
Pour faire pivoter, si la flèche gauche ou droite est enfoncée, le vecteur de direction et le vecteur plan sont pivotés en utilisant les formules de multiplication avec la matrice de rotation (et sur l'angle rotSpeed).

```
    readKeys();
    // avancer si pas de mur devant soi
     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 ;
    }
    // recule si aucun mur derrière toi
     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 ;
    }
    //tourne vers la droite
     si (keyDown(SDLK_RIGHT))
    {
      // la direction de la caméra et le plan de la caméra doivent être tournés
       double oldDirX = dirX ;
      dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
      dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
      double ancienPlanX = planX ;
      planX = planX * cos(-rotSpeed) - planY * sin(-rotSpeed);
      planY = ancienPlanX * sin(-rotSpeed) + planY * cos(-rotSpeed);
    }
    //tourne vers la gauche
     si (keyDown(SDLK_LEFT))
    {
      // la direction de la caméra et le plan de la caméra doivent être tournés
       double oldDirX = dirX ;
      dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
      dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
      double ancienPlanX = planX ;
      planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
      planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
    }
  }
}
```

Ceci conclut le code du raycaster non texturé, le résultat ressemble à ceci, et vous pouvez vous promener dans la carte :



Voici un exemple de ce qui se passe si le plan de la caméra n'est pas perpendiculaire au vecteur de direction, le monde apparaît de travers :



Raycaster texturé

Téléchargez le code source ici : raycaster_textured.cpp

Le noyau de la version texturée du raycaster est presque le même, seulement à la fin quelques calculs supplémentaires doivent être faits pour les textures, et une boucle dans la direction y est nécessaire pour aller à travers chaque pixel pour déterminer quel texel (pixel de texture) de la texture doit être utilisé pour cela.

Les bandes verticales ne peuvent plus être dessinées avec la commande de ligne verticale, à la place, chaque pixel doit être dessiné séparément. La meilleure façon est d'utiliser un tableau 2D comme tampon d'écran cette fois, et de le copier à l'écran immédiatement, ce qui va beaucoup plus vite que d'utiliser pset.

Bien sûr, nous avons maintenant également besoin d'un tableau supplémentaire pour les textures, et puisque la fonction "drawbuffer" fonctionne avec des valeurs entières uniques pour les couleurs (au lieu de 3 octets séparés pour R, G et B), les textures sont également stockées dans ce format . Normalement, vous chargeriez les textures à partir d'un fichier de texture, mais pour cet exemple simple, des textures stupides sont générées à la place.

Le code est essentiellement le même que l'exemple précédent, les parties en gras sont nouvelles. Seules les pièces neuves sont expliquées.

ScreenWidth et screenHeight sont maintenant définis au début car nous avons besoin de la même valeur pour la fonction screen et pour créer le tampon d'écran. Les autres nouveautés sont la largeur et la hauteur de la texture qui sont définies ici. Ce sont évidemment la largeur et la hauteur en texels des textures.

La carte du monde est également modifiée, il s'agit d'une carte plus complexe avec des couloirs et des pièces pour montrer les différentes textures. Encore une fois, les 0 sont des espaces vides et chaque nombre positif correspond à une texture différente.

```
#define screenWidth 640
#define screenHeight 480
#define texWidth 64
#define texHeight 64
#define mapLargeur 24
#define mapHauteur 24

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

```
  {4,0,0,0,0,0,0,5,0,0,0,0,0,0,5,7,0,0,0,0,0,0,0},
  {4,0,0,0,0,0,0,5,0,0,0,0,0,0,5,7,0,0,0,7,7,7,1},
  {4,0,0,0,0,0,0,5,5,5,5,0,5,5,5,5,7,7,7,7,7,7,1},
  {6,6,6,6,6,6,6,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6},
  {8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},
  {6,6,6,6,6,6,0,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6},
  {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}
} ;
```

Le tampon d'écran et les tableaux de texture sont déclarés ici. Le tableau de texture est un tableau de std :: vectors, chacun avec une certaine largeur * hauteur pixels.

```
int main(int /*argc*/, char */*argv*/ [])
{
  double posX = 22,0, posY = 11,5 ;  //x et y position de départ
   double dirX = -1.0, dirY = 0.0; //vecteur de direction initial
   double planeX = 0.0, planeY = 0.66; //la version 2d raycaster du plan caméra

  double time = 0; //heure de la trame courante
   double oldTime = 0; //heure de l'image précédente

  Uint32 buffer[screenHeight][screenWidth] ; // coordonnée y en premier car cela fonctionne par ligne de balayage
   std :: vectortexture[8] ;
  for(int i = 0; i < 8; i++) texture[i].resize(texWidth * texHeight);
```

La fonction principale commence maintenant par générer les textures. Nous avons une double boucle qui traverse chaque pixel des textures, puis le pixel correspondant de chaque texture obtient une certaine valeur calculée à partir de x et y. Certaines textures ont un motif XOR, d'autres un simple dégradé, d'autres une sorte de motif de briques, en gros ce sont tous des motifs assez simples, ça ne va pas être si beau, pour de meilleures textures voir le chapitre suivant.

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

  // génère des textures
   pour (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); //texture plate rouge avec texture croix noire
     [1][texWidth * y + x] = xycolor + 256 * xycolor + 65536 * xycolor; //
     texture inclinée en niveaux de gris[2][texWidth * y + x] = 256 * xycolor + 65536 * xycolor; //
     texture dégradé jaune incliné[3][texWidth * y + x] = xorcolor + 256 * xorcolor + 65536 * xorcolor; //xor
     texture en niveaux de gris[4][texWidth * y + x] = 256 * xorcolor; //
     texture verte xor[5][texWidth * y + x] = 65536 * 192 * (x % 16 && y % 16); //
     texture de briques rouges[6][texWidth * y + x] = 65536 * ycolor; //dégradé rouge
    texture[7][texWidth * y + x] = 128 + 256 * 128 + 65536 * 128 ; //texture grise plate
   }
```

C'est à nouveau le début de la boucle de jeu et des déclarations et calculs initiaux avant l'algorithme DDA. Rien n'a changé ici.

```
  //démarre la boucle principale while
   (!done())
  {
    for(int x = 0; x < w; x++)
    {
      //calculer la position et la direction
       du rayon double cameraX = 2*x/double(w)-1 ; //coordonnée x dans l'espace caméra
       double rayDirX = dirX + planeX*cameraX ;
      double rayonDirY = dirY + planY*caméraX ;

      //dans quelle case de la carte nous sommes
       int mapX = int(posX);
      int mapY = int (posY);

      //longueur du rayon de la position actuelle au côté x ou y suivant
       double sideDistX ;
      double sideDistY ;

      //longueur du rayon d'un côté x ou y au côté x ou y suivant
       double deltaDistX = sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX));
      double deltaDistY = sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY));
      double perpWallDist ;

      //dans quelle direction aller dans la direction x ou y (soit +1 ou -1)
       int stepX ;
      int stepY ;

      int hit = 0 ; // y a-t-il eu un coup de mur ?
      côté int ; // est-ce qu'un mur NS ou EW a été touché ?
```

```
//calculer le pas et la sideDist initiale
 si (rayDirX < 0)
{
  stepX = -1 ;
  sideDistX = (posX - mapX) * deltaDistX ;
}
autre
{
  étapeX = 1 ;
  sideDistX = (mapX + 1.0 - posX) * deltaDistX ;
}
si (rayDirY < 0)
{
  étapeY = -1 ;
  sideDistY = (posY - mapY) * deltaDistY ;
}
autre
{
  étapeY = 1 ;
  sideDistY = (mapY + 1.0 - posY) * deltaDistY ;
}
```

C'est encore la boucle DDA, et les calculs de la distance et de la hauteur, rien n'a changé ici non plus.

```
//exécute DDA
 tant que (appuyez sur == 0)
{
  //passe au carré suivant de la carte, soit dans la direction x, soit dans la direction y
   if (sideDistX < sideDistY)
  {
    sideDistX += deltaDistX ;
    carteX += étapeX ;
    côté = 0 ;
  }
  autre
  {
    sideDistY += deltaDistY ;
    cartes + = stepY ;
    côté = 1 ;
  }
  //Vérifier si le rayon a touché un mur
   if (worldMap[mapX][mapY] > 0) hit = 1 ;
}

//Calculer la distance du rayon perpendiculaire (la distance euclidienne donnerait un effet fisheye !)
 if(side == 0) perpWallDist = (sideDistX - deltaDistX);
sinon perpWallDist = (sideDistY - deltaDistY);

//Calculer la hauteur de la ligne à dessiner à l'écran
 int lineHeight = (int)(h / perpWallDist);

// calcule le pixel le plus bas et le plus haut pour remplir la bande actuelle
 int drawStart = -lineHeight / 2 + h / 2;
si(drawStart < 0) drawStart = 0 ;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h) drawEnd = h - 1;
```

Les calculs suivants sont cependant nouveaux et remplacent le sélecteur de couleur du raycaster non texturé.
La variable texNum est la valeur du carré de carte courant moins 1, la raison en est qu'il existe une texture 0, mais le carreau de carte 0 n'a pas de texture puisqu'il représente un espace vide. Pour pouvoir utiliser la texture 0 de toute façon, soustrayez 1 pour que les tuiles de carte avec la valeur 1 donnent la texture 0, etc...

La valeur wallX représente la valeur exacte où le mur a été touché, pas seulement les coordonnées entières du mur. Ceci est nécessaire pour savoir quelle coordonnée x de la texture nous devons utiliser. Ceci est calculé en calculant d'abord la coordonnée x ou y exacte dans le monde, puis en soustrayant la valeur entière du mur. Notez que même s'il s'appelle wallX, c'est en fait une coordonnée y du mur si side==1, mais c'est toujours la coordonnée x de la texture.

Enfin, texX est la coordonnée x de la texture, et celle-ci est calculée à partir de wallX.

```
//calculs de texturation
 int texNum = worldMap[mapX][mapY] - 1; //1 en est soustrait pour que la texture 0 puisse être utilisée !

//calculer la valeur de wallX
 double wallX ; //où exactement le mur a été touché
 if (side == 0) wallX = posY + perpWallDist * rayDirY;
sinon murX = posX + perpWallDist * rayDirX ;
murX -= sol((murX));

//coordonnée x sur la 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;
```

Maintenant que nous connaissons l'abscisse de la texture, nous savons que cette coordonnée restera la même, car nous restons dans la même bande verticale de l'écran. Nous avons maintenant besoin d'une boucle dans la direction y pour donner à chaque pixel de la bande verticale la coordonnée y correcte de la texture, appelée texY.

La valeur de texY est calculée en augmentant d'une taille de pas précalculée (ce qui est possible car elle est constante dans la bande verticale) pour chaque

pixel. La taille de pas indique de combien augmenter les coordonnées de texture (en virgule flottante) pour chaque pixel dans les coordonnées verticales de l'écran. Il doit ensuite convertir la valeur à virgule flottante en nombre entier pour sélectionner le pixel de texture réel.

REMARQUE : un algorithme de Bresenham ou DDA plus rapide uniquement en nombre entier peut être possible pour cela.

REMARQUE : Le pas à pas effectué ici est un mappage de texture affine, ce qui signifie que nous pouvons interpoler linéairement entre deux points plutôt que d'avoir à calculer une division différente pour chaque pixel. Ce n'est pas une perspective correcte en général, mais pour des murs parfaitement verticaux (et aussi des sols/plafonds parfaitement horizontaux), nous pouvons donc l'utiliser pour le raycasting.

La couleur du pixel à dessiner est alors simplement obtenue à partir de texture[texNum][texX][texY], qui est le bon texel de la bonne texture.

Comme le raycaster non texturé, ici aussi, nous rendrons la valeur de couleur plus sombre si un côté en y du mur a été touché, car cela semble un peu mieux (comme s'il y avait une sorte d'éclairage). Cependant, comme la valeur de couleur n'existe pas à partir d'une valeur R, G et B séparée, mais ces 3 octets collés ensemble dans un seul entier, un calcul pas si intuitif est utilisé.

La couleur est assombrie en divisant R, G et B par 2. La division d'un nombre décimal par 10 peut être effectuée en supprimant le dernier chiffre (par exemple 300/10 est 30 : le dernier zéro est supprimé). De même, diviser un nombre binaire par 2, ce qui est fait ici, revient à supprimer le dernier bit. Cela peut être fait en le décalant vers la droite avec >>1. Mais, ici, nous effectuons un décalage de bits sur un entier 24 bits (en fait 32 bits, mais les 8 premiers bits ne sont pas utilisés). Pour cette raison, le dernier bit d'un octet deviendra le premier bit de l'octet suivant, et cela bousille les valeurs de couleur ! Ainsi, après le décalage de bits, le premier bit de chaque octet doit être mis à zéro, et cela peut être fait en binaire "AND-ing" la valeur avec la valeur binaire 011111110111111101111111, qui est 8355711 en décimal. Le résultat est donc bien une couleur plus foncée.

Enfin, le pixel de tampon actuel est défini sur cette couleur et nous passons au y suivant.

```
        // De combien augmenter la coordonnée de texture par pixel d'écran
     double step = 1.0 * texHeight / lineHeight;
    // Coordonnée de texture de départ
     double texPos = (drawStart - h / 2 + lineHeight / 2) * step;
    for(int y = drawStart; y<drawEnd; y++)
    {
      // Convertit la coordonnée de texture en entier, et masque avec (texHeight - 1) en cas de débordement
       int texY = (int)texPos & (texHeight - 1);
      texPos += étape ;
      Uint32 color = texture[texNum][texHeight * texY + texX] ;
      // rend la couleur plus sombre pour les côtés y : les octets R, G et B sont divisés chacun par deux avec un "décalage" et un "et"
       if(side == 1) color = (color >> 1) & 8355711;
      buffer[y][x] = couleur ;
    }
  }
```

Maintenant, le tampon doit encore être dessiné, et après cela, il doit être effacé (là où dans la version non texturée, nous devions simplement utiliser "cls". Assurez-vous de le faire dans l'ordre des lignes de balayage pour plus de rapidité grâce à la localité mémoire pour la mise en cache). Le reste de ce code est à nouveau le même.

```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //efface le tampon au lieu de cls()
 //timing pour l'entrée et le compteur FPS
 oldTime = time;
temps = getTicks();
double frameTime = (heure - oldTime) / 1000.0 ; //frametime est le temps que cette image a pris, en secondes
 print(1.0 / frameTime); // compteur FPS
 redessiné();

// modificateurs de vitesse
 double moveSpeed = frameTime * 5.0 ; //la valeur constante est en carrés/seconde
 double rotSpeed = frameTime * 3.0; //la valeur constante est en radians/seconde
```

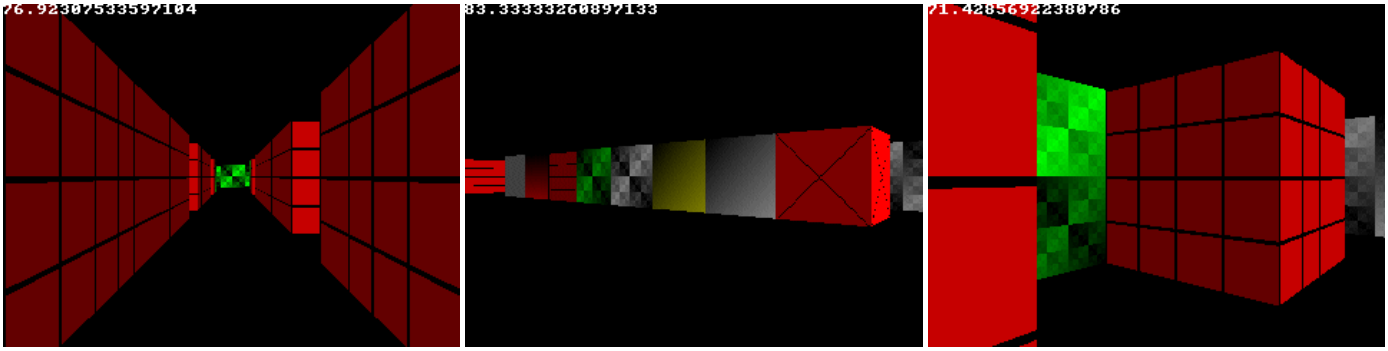
Et revoici les clés, rien n'a changé ici non plus. Si vous le souhaitez, vous pouvez essayer d'ajouter des touches de mitraillage (pour mitrailler à gauche et à droite). Celles-ci doivent être faites de la même manière que les touches haut et bas, mais utilisez planeX et planeY au lieu de dirX et dirY.

```
readKeys();
// avancer si pas de mur devant soi
 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 ;
}
// recule si aucun mur derrière toi
 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 ;
}
//tourne vers la droite
 si (keyDown(SDLK_RIGHT))
{
  // la direction de la caméra et le plan de la caméra doivent être tournés
   double oldDirX = dirX ;
  dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
  dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
  double ancienPlanX = planX ;
  planX = planX * cos(-rotSpeed) - planY * sin(-rotSpeed);
  planY = ancienPlanX * sin(-rotSpeed) + planY * cos(-rotSpeed);
```

```
    }
    //tourne vers la gauche

     si (keyDown(SDLK_LEFT))
    {
      // la direction de la caméra et le plan de la caméra doivent être tournés
       double oldDirX = dirX ;
      dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
      dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
      double ancienPlanX = planX ;
      planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
      planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
    }
  }
}
```

Voici quelques captures d'écran du résultat :



Remarque : Habituellement, les images sont stockées par des lignes de balayage horizontales, mais pour un raycaster, les textures sont dessinées sous forme de bandes verticales. Par conséquent, pour utiliser de manière optimale le cache du CPU et éviter les sauts de page, il pourrait être plus efficace de stocker les textures en mémoire bande verticale par bande verticale, plutôt que par ligne de balayage horizontale. Pour cela, après avoir généré les textures, échangez leur X et Y par (ce code ne fonctionne que si texWidth et texHeight sont identiques) :

```
// échange la texture X/Y car elles seront utilisées comme bandes verticales
 for(size_t i = 0; i < 8; i++)
for(size_t x = 0; x < texSize; x++)
pour(taille_t y = 0; y < x; y++)
std::swap(texture[i][texSize * y + x], texture[i][texSize * x + y]);
```

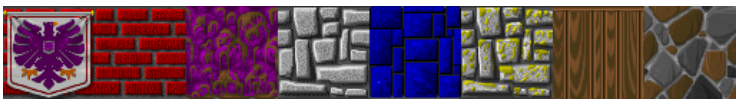
Ou échangez simplement X et Y là où les textures sont générées, mais dans de nombreux cas, après avoir chargé une image ou obtenu une texture à partir d'autres formats, vous l'aurez de toute façon dans les scanlines et devrez l'échanger de cette façon.

Lorsque vous obtenez le pixel à partir de la texture, utilisez plutôt le code suivant :

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

Textures Wolfenstein 3D

Au lieu de simplement générer des textures, chargeons-en quelques-unes à partir d'images ! Par exemple les 8 textures suivantes, qui proviennent de Wolfenstein 3D et sont copyright ID Software.



Remplacez simplement la partie du code qui génère les motifs de texture par ce qui suit (et assurez-vous que ces textures sont dans le bon chemin). Vous pouvez télécharger les textures ici .

```
// génère des textures
 non signées longues 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" );
```

Dans le Wolfenstein 3D original, les couleurs d'un côté étaient également rendues plus sombres que la couleur de l'autre côté d'un mur pour créer l'effet d'ombre, mais ils utilisaient une texture distincte à chaque fois, une sombre et une claire. Ici cependant, une seule texture est utilisée pour chaque mur et la ligne de code qui divise R, G et B par 2 est ce qui rend les côtés y plus sombres.

Considérations relatives aux performances

Sur un ordinateur moderne, lors de l'utilisation d'une haute résolution (4K, à partir de 2019), ce raycaster logiciel sera plus lent que certains graphiques 3D beaucoup plus complexes rendus sur le GPU avec une carte graphique 3D.

Il y a au moins deux problèmes qui freinent la vitesse du code raycaster dans ce tutoriel, que vous pouvez prendre en compte si vous souhaitez créer un raycaster super rapide pour des résolutions très élevées :

- Raycasting fonctionne avec des bandes verticales, mais le tampon d'écran en mémoire est agencé avec des lignes de balayage horizontales. Ainsi, dessiner des bandes verticales est mauvais pour la localité de la mémoire pour la mise en cache (c'est en fait le pire des cas), et la perte d'une bonne mise en cache peut nuire à la vitesse plus que certains des calculs 3D sur les machines modernes. Il peut être possible de programmer cela avec un meilleur comportement de mise en cache (par exemple, traiter plusieurs bandes à la fois, utiliser un algorithme de transposition sans cache ou avoir un raycaster tourné à 90 degrés), mais pour plus de simplicité, le reste de ce tutoriel ignore ce problème de mise en cache.
- Cela utilise le blitting logiciel avec SDL (dans QuickCG, dans redraw()), qui est lent pour les grandes résolutions par rapport au rendu matériel. L'utilisation probable de QuickCG de SDL lui-même n'est pas optimale et, par exemple, l'utilisation d'OpenGL (même pour le rendu logiciel) peut être plus rapide, ce qui peut être corrigé dans les coulisses. Étant donné que ce didacticiel CG concerne le rendu logiciel, ce problème est également ignoré ici.

Partie suivante

Aller directement à la partie II

Dernière édition : 2020