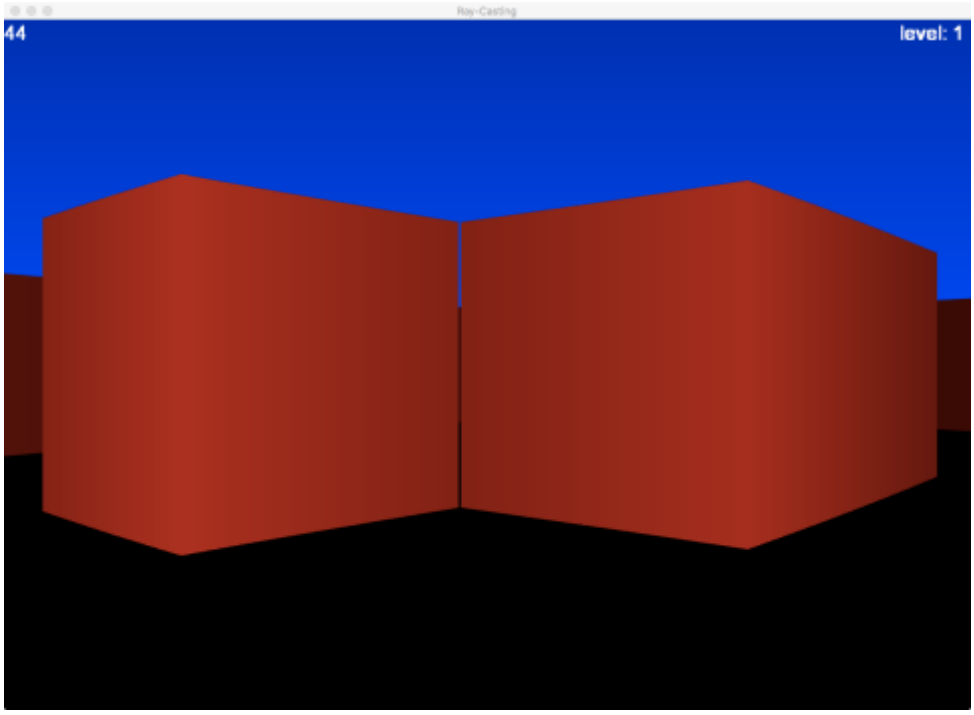
Ray-casting



Labyrinthe 2016, © D3M0T3p

Travail de maturité réalisé par

D3m0t3p

Table des matières

1. Introduction

Passionné d'informatique et de nouvelles technologies, j'ai toujours rêvé de créer mon propre jeu vidéo. Ce travail de maturité a été l'occasion de mener ce projet à bien.

La réalisation des jeux vidéo repose essentiellement sur des techniques de gestion d'images de synthèses. Parmi les nombreuses techniques existantes j'ai choisi d'implémenter la technique de ray-casting dans un jeu.
Ce nom ne parle pas à grand monde, et je ne le connaissais moi-même pas avant de lire un article[1] décrivant l'utilisation du ray-casting dans de vieux jeux comme Wolfenstein et Doom (1990). Suite à la lecture, j'ai eu envie de tester cette technique par moi-même. Dans mon travail, nous verrons comment mettre en place un moteur de ray-casting au sein d'un jeu.

Pour mener à bien ce travail, j'ai utilisé le C++ comme langage de programmation, car c'est avec lui que je suis le plus à l'aise. Bien que de nombreux guides d'apprentissage existent sur internet, l'ouvrage C++ Primer 5[2] reste la référence pour acquérir une bonne compréhension du langage. D'autre part le C++ est un langage très rapide à l'exécution et qui est multiplateforme, c'est à dire que le code écrit fonctionnera sur les principaux OS : Windows et les dérivés de Unix. Ce langage toujours très utilisé est ancien et ne permet pas de gérer nativement des interfaces graphiques comme les fenêtres, les boutons, etc… Il existe cependant plusieurs moyens de faire des interfaces graphiques en C++. Le premier est d'utiliser OpenGL qui permet d'accéder directement à la carte graphique et de créer sa propre fenêtre. Cependant ce moyen est très compliqué et de bas niveau.

J'ai choisi d'utiliser la bibliothèque SFML : Simple and Fast Multimedia Library. Nous n'aurons ainsi pas besoin de nous soucier d'OpenGL, ce qui est pratique car il faut de très bonnes connaissances sur le fonctionnement de l'ordinateur pour réussir à l'utiliser. Ainsi SFML nous offre une interface haut-niveau, ce qui signifie qu'elle s'occupe des détails nous lui dirons juste « fait une fenêtre de cette taille, dessine un rectangle là » et nous n'aurons pas à gérer chaque pixel. Remarquons que la SFML est plus limitée que OpenGL : elle ne permet pas de gérer nativement la 3D.

En plus de la bibliothèque SFML, j'ai eu besoin d'autres outils comme un IDE (Integrated Development Environment) pour écrire le programme et le compiler (transformer le code en un fichier exécutable par l'ordinateur). Dans mon cas, travaillant sur un mac book air 2014 avec seulement 4GB de mémoire, j'ai utilisé Xcode car il est fourni gratuitement par Apple. Il arrive souvent en programmant qu'on fasse une modification qui fasse que le programme ne fonctionne plus. Il est donc essentiel d'utiliser un gestionnaire de version afin de savoir exactement les changements effectués d'une version à l'autre. J'ai utilisé le très populaire logiciel de gestion de versions « git [3]». Pour éviter toute perte de données, j'ai synchronisé mon projet à chaque ajout de code sur github.com qui est une version de git dans le cloud public. Ainsi même si je devais perdre mon ordinateur chaque version serait en ligne et je pourrais récupérer chaque version depuis n'importe quel poste.

A l'aide du moteur de ray-casting que j'ai développé, j'ai créé un jeu de labyrinthe en 3D dans lequel il faut trouver la sortie. La jouabilité se veut minimaliste car l'intérêt principal de ce projet est de faire des rendus à l'aide du moteur de ray-casting.

---

1 https://zestedesavoir.com/articles/153/comment-doom-et-wolfenstein-affichaient-leurs-graphismes/ 22 septembre 2015

2 Stanley B Lippman, Josée Lajoie, Barbara E. Moo, C++ Primer 5th Edition, Westford (massachusetts), Addison Wesley, 2015

3 https://git-scm.com

2. Ray-tracing

Avant de rentrer dans les détails du ray-casting, je tiens à attirer votre attention sur le ray-tracing qui ressemble de nom mais est en vérité complètement différent. Le ray-tracing vise à réaliser une image très réaliste. Elle calcule le trajet de la lumière, sa réflexion sur tous les objets de la scène, l'absorbance de la lumière par les objets, la réfractions selon l'indice optique du milieu etc. Cela rend les calculs très complexes et même un ordinateur actuel très puissant peut prendre plusieurs heures de calcul selon le degré de réalisme pour réaliser une image comme sur la figure 1.

Figure 1 scène produite avec le ray-tracing[4]

On comprend aisément qu'il n'est pas possible de faire un jeu avec cette méthode car un jeu doit pouvoir afficher plusieurs images par seconde pour avoir un rendu fluide. C'est pourquoi je me suis plutôt orienté vers le ray-casting qui permet de rendre des espaces en 3D en temps réels sans calculs trop lourds.

3. Ray-casting

Le ray-casting est une technique développée dans les années 1990 visant à dessiner des espaces 3D en temps réel sur des ordinateurs qui avaient peu de puissance. Il faut bien réaliser que les ordinateurs de l'époque avaient moins de puissance que nos téléphones portables actuels. Cette technique révolutionna l'industrie du jeu-vidéo. Doom qui utilisait cette technique fit dix millions de vente, ce qui était énorme pour l'époque.
Pour pouvoir fonctionner avec de faibles ressources, le ray-casting utilise des astuces qui permettent de réduire le temps de calcul. C'est pourquoi le ray-casting ne donne pas réellement de la 3D, mais s'apparente plus à un dessin ayant une perspective.

Pour limiter les calculs et être rapide, le ray-casting impose les contraintes suivantes:

1) Les éléments de murs ont tous la même taille (H=P=L).
2) Les murs sont plats et à 90° par rapport au sol.
3) Le joueur a la taille d'une moitié de mur.
4) Le joueur ne peut pas bouger la caméra sur l'axe y.

4 https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

A cause de ces restrictions les rendus ont des caractéristiques typiques, on remarque par exemple que les éléments de mur sont carrés, de la même taille et que rien n'est arrondi, les angles entre les murs sont de 90°. On peut juxtaposer des éléments afin de former un plus long mur.

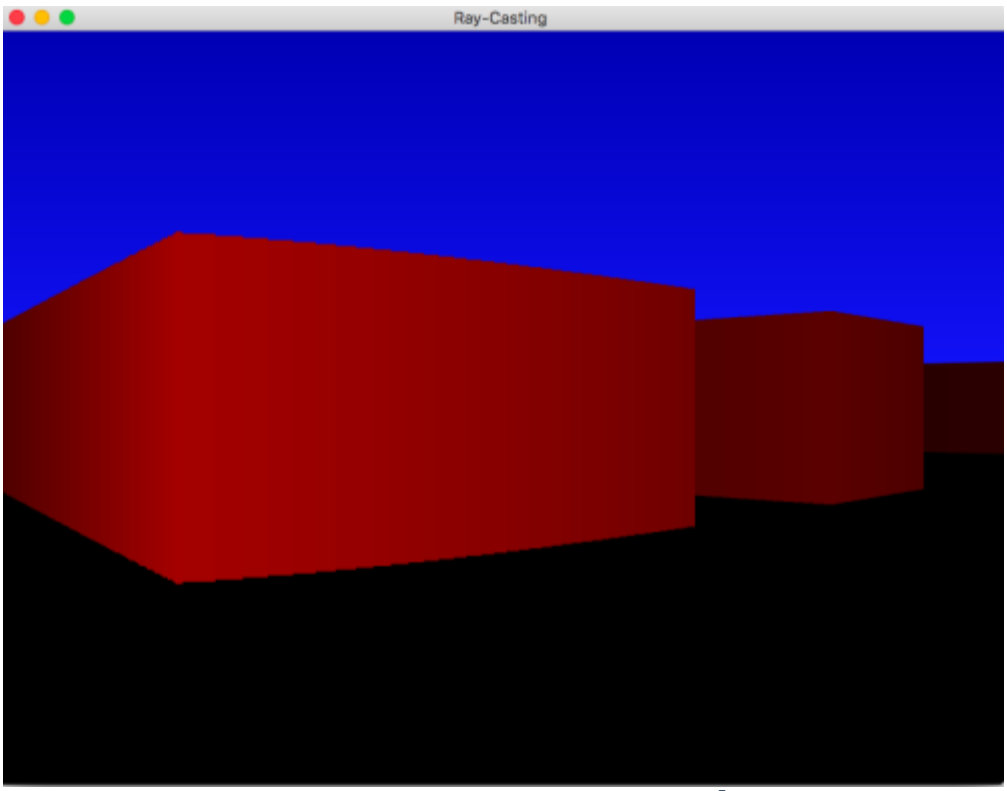


Figure 2 : caractéristiques du ray-casting[5]

Comme nous le voyons sur la figure 2, toutes ces caractéristiques se retrouvent dans mon travail.

4. Présentation des différents référentiels

L'utilisation de la SFML ajoute une contrainte supplémentaire présentée ci-dessous. La SFML utilise un système de repère qui ne respecte pas celui du cercle trigonométrique, il faut donc faire une conversion repère mathématique <-> repère de dessin.
La SFML est conçue comme ceci : le points 0;0 est en haut à gauche.

5 image du jeu

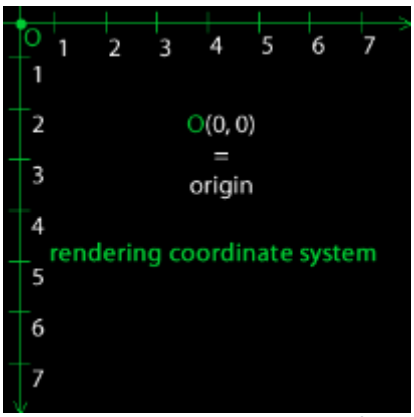


Figure 3 : repère de la SFML[6]

La carte du monde est comme sur la figure 4, elle stocke les informations comme une matrice, c'est à dire qu'on accède en premier lieu à la ligne puis à la colonne, les coordonnées d'un point sont (y ; x). Elle est faite de manière à s'accorder avec les coordonnées spéciales de la SFML.

+	---------- > x									
|	1	1	1	1	1	1	1	1	1	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	1	0	1	0	1
|	1	0	0	0	0	1	0	0	1	0	1
|	1	0	0	0	0	1	0	0	1	0	2
|	1	0	0	0	0	1	0	0	1	0	1
|	1	0	0	0	0	1	0	0	1	0	1
V	1	0	0	0	0	1	0	0	0	0	1
y	1	1	1	1	1	1	1	1	1	1	1

Figure 4 : exemple de carte

Les 1 sont des murs et les 0 du vide, chaque case de vide ou de mur est un espace de 64 [m] ou unité. J'attribue ainsi à des caractères des rôles, par exemple le 2 représente la sortie du labyrinthe.

[6] https://dev.my-gate.net/2012/06/21/using-sfview/

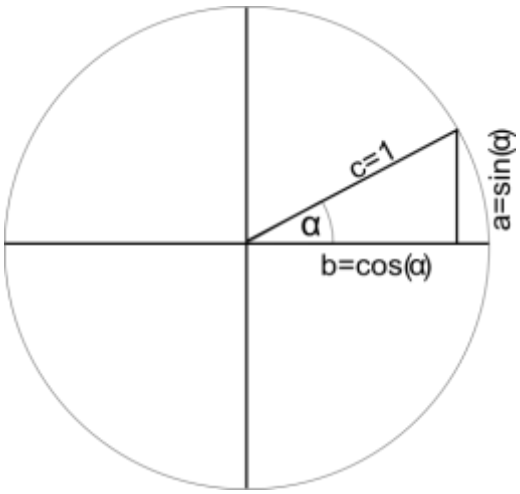


Figure 6 : sinus dans le cercle trigonométrique[7]

+	---------- > x									
|	1	1	1	1	1	1	1	1	1	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	1	0	0	1
|	1	0	0	0	0	1	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	0	1
|	1	0	0	0	0	0	0	0	2	1
V	1	0	0	0	0	0	0	0	0	1
y	1	1	1	1	1	1	1	1	1	1

Figure 5 : l'axe y augmente vers le bas

Il est très important de noter que l'axe y n'est pas orienté dans le même sens que dans le cercle trigonométrique, cela implique que lorsque je regarde « vers le bas » (angle de 270°) et avance, la valeur y augmente, tandis que sur le cercle trigonométrique elle diminue. Il faut donc inverser le signe du sinus à chaque utilisation. De plus les fonctions trigonométriques en C++ prennent des angles en radian ce qui implique que les angles en degré soient convertis en radian avant chaque utilisation.

Un exemple pour illustrer le problème :
Imaginons que le joueur avance avec un angle de 45°, le calcul qui s'impose est le suivant :

$$position.y = position.y + sin(45 * \pi/180) * distance$$

La position en y augmente alors que sur ma carte elle diminue, je soustrais donc à la place d'ajouter, et le calcul devient :

$$position.y = position.y - sin(45 * \pi/180) * distance$$

De cette manière la position dans la carte sera correcte. De plus, lorsque j'aurai un angle entre 180° et 360°, le sinus me donnera un nombre négatif que je soustrairai pour avoir au final un nombre positif, ce qui concorde bien avec ma carte.

5. Les calculs

Comme dit précédemment, nous pouvons comparer le ray-casting avec un dessin ayant une perspective. En effet plus un objet est loin, plus il est petit et inversement.

[7] http://blog.teamleadnet.com/2013/09/drawing-circle-and-calculating-sinus.html

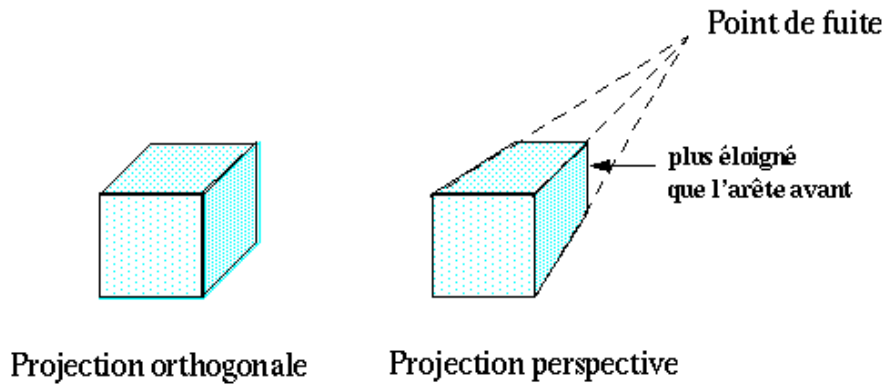


Figure 7 différentes perspective illustrée par un cube[8]

Le ray-casting se base sur cette même idée : les objets les plus éloignés sont les plus petits, comme tous les objets (mur) ont la même taille le travail nous est grandement simplifié. Il nous suffit donc de déterminer la distance entre la caméra (le joueur) et le mur qu'il regarde et d'ensuite afficher à l'endroit du mur un rectangle d'une hauteur inversement proportionnelle à sa distance.

Pour calculer la distance nous procédons comme ceci :
On se positionne sur le joueur et on avance de 2 mètres en suivant son regard, on teste si on est sur un mur, sinon on recommence jusqu'à tomber sur un mur. La fonction de calcul prend donc en paramètre la carte, l'angle de la direction du regard du joueur et sa position. Avec ces informations la fonction retourne la distance du joueur jusqu'au mur.

Un pseudo code de cette routine pourrait s'écrire comme ceci :

```
fonc trouverDistance(position, carte, directionVue)
{
	distance = 0
	si position dans carte == mur
	{
		retourne la distance
	}

	position.x += 2*cos(directionVue)
	position.y += 2*sin(directionVue)
	distance += 2                                   //simplification de Pythagore
}
```

Il faut bien sûr répéter cette routine pour chaque colonne de pixel affichée à l'écran.
Une fois la distance connue, on affiche un rectangle à l'endroit que l'on vient de tester et on lui donne la taille correspondant à la distance.

[8] https://www.irit.fr/~Jean-Denis.Durou/ENSEIGNEMENT/VISION/COURS/co04.html

Avec cette méthode de calcul, il se peut que des erreurs surviennent. Imaginons que nous ayons deux murs placés en diagonal.

J	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0

Le joueur est représenté par la lettre J, les cases rouges sont les endroits où le *rayon détecteur* passe, comme il avance de deux en deux il va peut-être sauter l'obstacle (les '1') et arriver dans la case jaune. Cela se traduit dans le jeu par un léger trou à l'intersection des murs, comme illustré dans la figure 8.

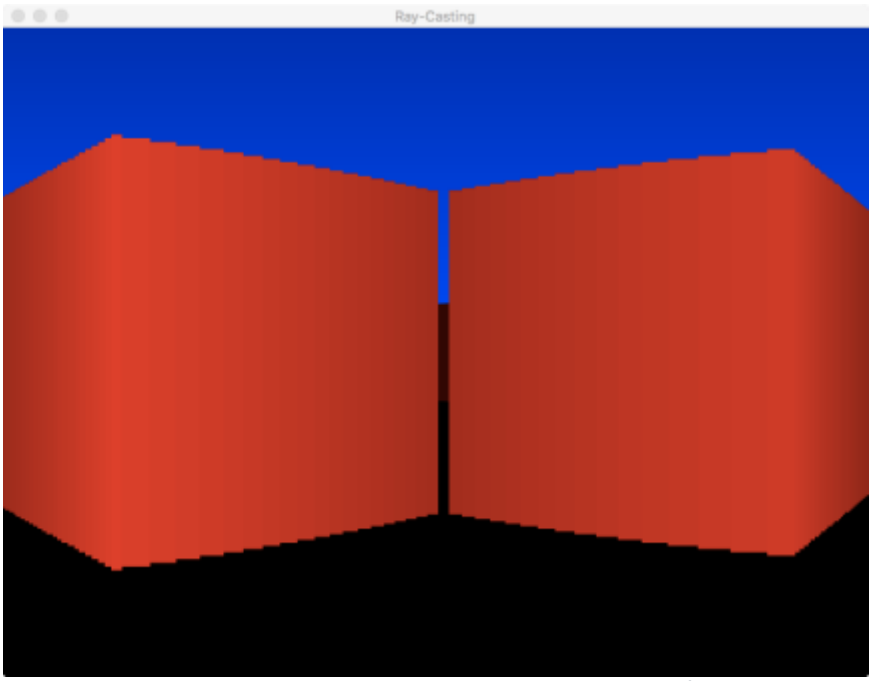


Figure 8 Exemple d'erreur de la méthode de calcul[9]

Il existe plusieurs solutions pour éviter cela. La première consiste à affiner la recherche des murs en mettant un incrément plus petit, par exemple un incrément de 0.5 à la place de 2 mais le temps de calcul quadruplerait et l'erreur ne serait pas éliminée dans 100% des cas. Seul un saut infinitésimal avec une précision infinie et un temps de calcul tendant vers l'infini, fonctionnerait, mais cette solution n'est pas réaliste.
Une astuce qui ne demande aucun calcul supplémentaire est de simplement rajouter un mur afin de combler le vide, le rayon passera peut-être les murs placés en « diagonal » mais détectera à coups sûr le mur derrière. Cela ne règle pas le problème et cela impose une nouvelle contrainte sur la carte.
La dernière solution est de complètement changer le système de calcul des distances.
A la place d'avancer par petit pas en espérant tomber sur un mur nous pouvons calculer l'emplacement de chaque changement de case et regarder si la case qui suit est, ou non, un mur. Cela peut être fait à l'aide de l'algorithme DDA (Digital Differential Analyzer).

[9] image du jeu

Cette technique est plus difficile à mettre en œuvre, et je ne l'ai pas implémentée. Par conséquent je ne la détaillerai pas plus.

6. Les problèmes rencontrés

6.1 Conversions implicites

Le C++ est un langage dit permissif, c'est à dire que si le programmeur écrit un calcul qui n'a aucun sens mais est juste au niveau de la syntaxe, le compilateur ne dira rien, même si le programme n'aura pas le comportement voulu. C'est un peu ce qui m'est arrivé, j'ai fait des conversions implicites dans certains calculs et le compilateur ne me l'a pas signalé. J'ai eu ce problème pour la boucle de rendu qui s'exécute soixante fois par seconde. Dans la boucle je fais le calcul 1/60 pour savoir combien de temps doit être affiché mon image mais comme les opérants sont des entiers, j'ai eu le résultat en entier (donc arrondi à 0) ce qui fait que la boucle ne fonctionnait pas comme je le souhaitais.

Un autre problème auquel j'ai dû faire face était la conversion entre les radians et les degrés. En effet je ne savais pas que les fonctions sinus et cosinus en C++ prenaient des angles en radian et non en degré ce que j'ai mis longtemps à découvrir et qui a été la cause de nombreux résultats erronés.

6.2 Fish-eyes

Les problèmes exposés précédemment sont dus à une mauvaise connaissance du C++, mais j'ai aussi eu à faire face à un souci provenant de la méthode de calcul. En effet si l'on se place perpendiculairement à un mur et que l'on calcul les distances, on remarque que les distances sur les côtés sont plus grandes, ce qui est vrai mais dans le jeu le mur sera plus petit et ça ne correspond pas à une vision réaliste.
Sur la figure 9 on peut voir l'effet du fish-eye lorsque on regarde un mur perpendiculairement et sur la figure 10, lorsqu'il est corrigé.

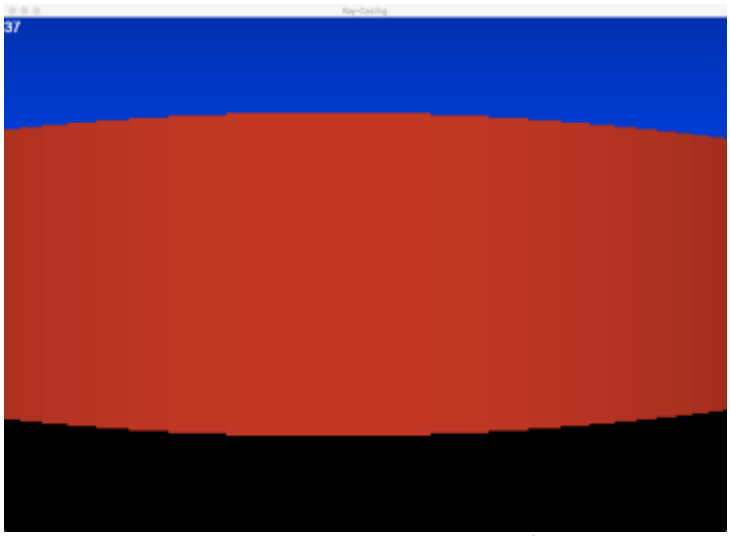


Figure 9 Exemple de fish-eyes[10]

[10] image du jeu

Figure 10: Exemple de fish-eyes corrigé[11]

La raison pour laquelle les Humains ne voient pas les murs arrondis est due à notre cristallin. C'est un composé de cellules fibreuses et transparentes qui agit comme une lentille et est situé dans l'œil. C'est grâce à lui que cette déformation est corrigée.

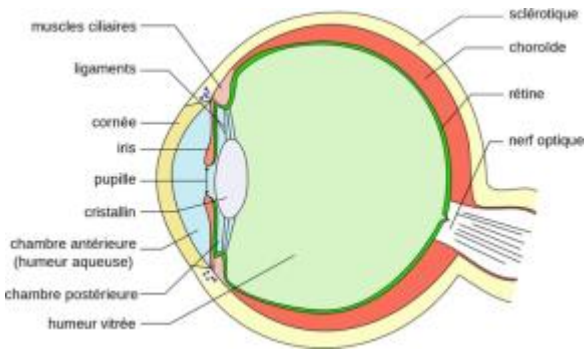


Figure 11: Œil humain[12]

Le nom de fish-eyes provient du fait que les poissons n'ont pas de cristallin ce qui fait qu'ils ont une vision déformée qui ressemble aux rendus qu'on obtient avec le ray-casting.

Pour corriger ce problème, j'ai dû simuler l'action de la lentille. La formule est simple : il suffit de multiplier la distance distordue par le cosinus de l'angle, angle entre le rayon distordu et le rayon correcte.
Dans la boucle où j'effectue les rendus, la variable *i* va de [player.angle-30 ; player.angle+ 30]. Il faut récupérer la valeur de *i* puis soustraire player.angle, ainsi nous avons l'angle entre le rayon distordu et le rayon correcte il ne reste plus qu'à multiplier par le sinus de l'angle.
La formule finale en notation C++ ressemble à cela :

$$\text{distance} \ *= \ \text{cosf}((\text{i} - \ \text{player.angle}) \ * \ 3.141592/180);$$

Je multiplie par $\frac{3.141592}{180}$ pour transformer les angles de degrés en radians.

Ainsi nous avons vu la majorité des problèmes rencontrés au cours du projet.

[11] image du jeu
[12] http://www.gralon.net/articles/sante-et-beaute/medecine/article-l-oeil-humain---schema-et-fonctionnement-9921.htm

7. Conclusions

J'ai donc réussi à créer un moteur de ray-casting en C++, mais presque un an a passé et ma vision du problème a évolué. Les choix pris au commencement me restreignent désormais. Par exemple la méthode de calcul utilisée pour trouver la distance au mur m'empêche de leur appliquer une texture. Si je commençais le projet aujourd'hui, je ferais des choix de conception différents. J'aurais directement commencé par implémenter la recherche de mur à l'aide de l'algorithme DDA et non à tâtons comme actuellement. De plus je ferais l'implémentation des deux modes de jeu (en pause ou en jeu) de façon orienté objet. J'aurais dû faire des classes qui correspondent à l'état du jeu. Ainsi je ne devrais pas tout modifier pour ajouter un autre mode de jeu, la création d'une nouvelle classe suffirait.
La dernière chose que je souhaiterais modifier est ma manière de travailler. En effet les développeurs utilisent des tests unitaires ce que j'aurais fait pour mon projet. Un test unitaire est un code qui vise à prouver le bon fonctionnement d'une partie du programme, en ayant une multitude de tests on peut ainsi vérifier que chaque partie du programme fonctionne individuellement comme prévu. Grâce à cela, on détecte plus tôt les bogues et le développement est accéléré. Je n'en ai pas fait car j'aurai dû apprendre à utiliser boost.test qui est un outil pour faire des tests unitaires, ce que je n'ai pas voulu faire. En effet, j'aurais été noyé sous la masse de nouvelles informations à assimiler.

8. Bilan personnel

Ce projet a été très enrichissant à plusieurs égards. Tout d'abord, je n'avais jamais réalisé de développements informatiques d'une telle d'envergure et sur une aussi longue période de temps. En faisant l'usage d'outils tels github/git, j'ai découvert les méthodologies utilisées dans les grands projets open source. Je me sens maintenant en mesure de pouvoir participer à ces derniers. J'ai aussi pu découvrir par moi-même le bien-fondé des règles qu'il faut suivre pour bien programmer.

Je suis très content d'avoir eu l'occasion de faire ce travail et je pense que grâce à ce projet mes futurs programmes seront mieux architecturés. J'essayerai à l'avenir d'utiliser des tests unitaires pour assurer la qualité du code que j'écris.

Finalement ce travail de maturité m'a conforté dans l'envie de continuer mes études dans l'informatique.

9. Bibliographie

9.1 Liens internet

- L'association Zeste de Savoir, zestedesavoir.com [en ligne], https://zestedesavoir.com/articles/153/comment-doom-et-wolfenstein-affichaient-leurs-graphismes/ (site mis à jour le mardi 19 janvier 2016 à 11h08)

- F. Permadi, permadi.com [en ligne], http://permadi.com/1996/05/ray-casting-tutorial-1/ (site mis à jour le 17 mai 1996)

- Wikipédia, https://en.wikipedia.org [en ligne], https://en.wikipedia.org/wiki/Camel_case (site visité le 27 octobre 2016)
- Linus Torvalds, https://git-scm.com [en ligne] https://git-scm.com (site visité le 15 mars 2016)

9.2 Livres

- Stanley B Lippman, Josée Lajoie, Barbara E. Moo, C++ Primer 5th Edition, Westford (massachusetts), Addison Wesley, 2015
- Jan Haller, Henrik Vogelius Hansson, Artur Moreira, SFML Game Development, Birmingham, Packt Publishing, 2013

9.3 Images

- « Glasses, pitcher, ashtray and dice », Gilles Tran, https://en.wikipedia.org/wiki/Ray_tracing_(graphics) (image consultée le 22 septembre 2015)

- « Calculating sinus », Adam Horvath, http://blog.teamleadnet.com/2013/09/drawing-circle-and-calculating-sinus.html (image consultée le 15 novembre 2015)

- « Exemple d'un cube », Jean-Denis Durou, https://www.irit.fr/~Jean-Denis.Durou/ENSEIGNEMENT/VISION/COURS/co04.html (image consultée le 3 mai 2016)

- « Schéma d'un œil humain », Audrey Vautherot, http://www.gralon.net/articles/sante-et-beaute/medecine/article-l-oeil-humain---schema-et-fonctionnement-9921.htm (image consultée le 24 mars)

10. Remerciements

Je tiens à remercier tout particulièrement M. XXXX qui a su me conseiller et me soutenir tout au long de mon travail. Il m'a particulièrement aidé à comprendre la partie mathématique de la méthode de ray-casting.
J'aimerais aussi remercier les développeurs de la SFML qui font un incroyable travail et mettent une bibliothèque de qualité à la portée de tous.

11. Déclaration d'authenticité

L'élève :

Prénom NOM : …D3M0T3P.....

Groupe : ...40X.....

Maître accompagnant : ..... XXXXX .... XXXX….....

Atteste avoir conçu et rédigé personnellement, dans son style propre, le travail de maturité ci-joint ;
Atteste notamment ne pas avoir eu recours au plagiat et avoir systématiquement et clairement mentionné tous les emprunts faits à autrui.

Lieu, date et signature :

…….....

12. Annexes

12.1 Main.cpp

```
// Here is a small helper for you ! Have a look.
#include "Game.hpp"
#include "SFML/Graphics.hpp"
#include <iostream>
#include "ResourcePath.hpp"

int main(int, char const**)
{
        Game gm{};

        gm.run();



    return EXIT_SUCCESS;
}
```

12.2 Player.hpp

```
//
//  Player.hpp
//  Ray-Casting
//
//

#pragma once
#include "SFML/Graphics.hpp"
#include <iostream>
#include <cmath>



class Player {
public:    //var
           sf::Vector2f position;
           float angle;
           float speed; //30 pixeel/s


           bool isMovingUp;
           bool isMovingDown;
           bool isMovingRight;
           bool isMovingLeft;




public:    //function
           Player();
           void move(const sf::Time &deltaTime);
           sf::Vector2f futurMove(const sf::Time &deltaTime);

};
```

12.3 Player.cpp

```
//
//  Player.cpp
//  Ray-Casting
//
//

#include "Player.hpp"

Player::Player():
	position(70,600),
	speed(75.0),
	angle(0),
	isMovingUp(false),
	isMovingDown(false),
	isMovingRight(false),
	isMovingLeft(false)

{

}


void Player::move(const sf::Time &deltaTime){
	if(isMovingDown){
	//DOWN
		position.y += speed*sinf(angle*3.14/180)*deltaTime.asSeconds();		//met - car sinus
		position.x -= speed*cosf(angle*3.14/180)*deltaTime.asSeconds();
	}



	if(isMovingUp){                                                                                 //UP
		position.y -= speed*sinf(angle*3.14/180)*deltaTime.asSeconds();
		position.x += speed*cosf(angle*3.14/180)*deltaTime.asSeconds();
	}

	if(isMovingRight){
		//UP
		position.y += speed*sinf((angle+90)*3.14/180)*deltaTime.asSeconds(); //met + car devrait etre -
mais on rajoute un - car c'est un sinus => +
		position.x -= speed*cosf((angle+90)*3.14/180)*deltaTime.asSeconds();
	}
	if(isMovingLeft){
		//UP
		position.y += speed*sinf((angle-90)*3.14/180)*deltaTime.asSeconds(); //met + car devrait etre -
mais on rajoute un - car c'est un sinus => +
		position.x -= speed*cosf((angle-90)*3.14/180)*deltaTime.asSeconds();
	}
}


sf::Vector2f Player::futurMove(const sf::Time &deltaTime){
	double x,y;
	x = position.x;
	y=position.y;
	if(isMovingDown){
	//DOWN
		y += speed*sinf(angle*3.14/180)*deltaTime.asSeconds();		//met - car sinus
		x -= speed*cosf(angle*3.14/180)*deltaTime.asSeconds();
	}
```

```
	if(isMovingUp){                                                                 //UP
		y -= speed*sinf(angle*3.14/180)*deltaTime.asSeconds();		//met + car devrait etre - mais on rajoute un - car c'est un sinus => +
		x += speed*cosf(angle*3.14/180)*deltaTime.asSeconds();
	}

	if(isMovingRight){
		//UP
		y += speed*sinf((angle+90)*3.14/180)*deltaTime.asSeconds();		//met + car devrait etre - mais on rajoute un - car c'est un sinus => +
		x -= speed*cosf((angle+90)*3.14/180)*deltaTime.asSeconds();
	}
	if(isMovingLeft){
		//UP
		y += speed*sinf((angle-90)*3.14/180)*deltaTime.asSeconds(); //met + car devrait etre - mais on rajoute un - car c'est un sinus => +
		x -= speed*cosf((angle-90)*3.14/180)*deltaTime.asSeconds();
	}
	return sf::Vector2f(x,y);
}
```

12.4 RayCasting.hpp

```
//
//  RayCasting.h
//
//
//
//
#pragma once
#include "SFML/Graphics.hpp"
#include "utility.hpp"
class RayCasting {



private:
	std::map<float,float> _cosTable;
	std::map<float,float> _sinTable;
	const double _PI ;

public:
	enum class Algo{
		LINEAR,
		DDA
	};

public:
	RayCasting();
	float rayCasting(const sf::Vector2f& playerPosition,float angle,const std::vector<std::vector<int>> &labyrinth,int &blockID, sf::Vector2f& hitPosition, const Algo algo)const;		//retourn la distance jusque au mur rencontré
private:
	sf::Vector2f computeLineCoo(const sf::Vector2f& position,float angle, const std::vector<std::vector<int>>& labyrinth) const;

	sf::Vector2f computeColumnCoo(const sf::Vector2f& position, float angle, const std::vector<std::vector<int>>& labyrinth) const;

};
```

12.5 RayCasting.cpp

```
//
//  RayCasting.cpp
//
//
//
//

#include "RayCasting.hpp"
#include <cmath>
#include <iostream>
#include <algorithm>

                                //definit dans utility.cpp
extern int sizeOfBlock;
                                        //pareil



RayCasting::RayCasting():
        _PI(3.14159265358979323846)
{

}
sf::Vector2f RayCasting::computeLineCoo(const sf::Vector2f &position, float angle, const
std::vector<std::vector<int> > &labyrinth) const{

        float x = position.x;
        float y = position.y;
        sf::Vector2f A;

        if(std::abs(static_cast<int>(floor(angle))%360) <180){  //[0; π]
                angle = angle * _PI / 180;

                A.y = floor( x/sizeOfBlock) * 64 - 1;
                A.x = x + ((y-A.y) / tanf(angle));
                while (true) {
                        if(labyrinth.at(floor(A.x/sizeOfBlock)).at(floor(A.y/sizeOfBlock)) != 0){
                                return sf::Vector2f(A.x, A.y);

                        }

                        A.x += sizeOfBlock/tan(angle);
                        A.y -= 64;
                }
        }
        else{   // ]π; 2π[
                angle = angle * _PI / 180;
                A.y = floor(x/64)*64+64;
                A.x = x + ((A.y - y)/tan(angle));

                while (true) {
                        if(labyrinth.at(floor(A.x/sizeOfBlock)).at(floor(A.y/sizeOfBlock)) != 0){
                                return sf::Vector2f(A.x, A.y);

                        }

                        A.x += sizeOfBlock/tan(angle);
                        A.y += 64;
                }
        }
}

sf::Vector2f RayCasting::computeColumnCoo(const sf::Vector2f &position, float angle, const
std::vector<std::vector<int> > &labyrinth) const {

        auto radTanAngle = -tan(angle*_PI/180);
```

```
	float x = position.x;
	float y = position.y;
	sf::Vector2f A;

	if(std::abs(static_cast<int>(floor(angle))%360) > 270 || std::abs(static_cast<int>(floor(angle)) % 360) <90){
	//
		//angle [270;360] U [0;90]
		angle = angle * _PI / 180;
		A.x = floor(x/sizeOfBlock)*sizeOfBlock + sizeOfBlock;
		A.y = y - radTanAngle * (A.x - x);

		while (true) {
			if(labyrinth.at(floor(A.y/sizeOfBlock)).at(floor(A.x/sizeOfBlock)) != 0){
				return sf::Vector2f(A.x, A.y);
			}

			A.x += 64;
			A.y += radTanAngle * 64;
		}
	}
	else{	//angle 90 < angle <270

		angle = angle * _PI / 180;
		A.x = floor(x/64) * 64 - 1;
		A.y = y - radTanAngle * (A.x - x);

		while ( /*A.x < labyrinth.size()*64 && A.x >0 && A.y < labyrinth.size()*64 && A.y*/true) {

	if(labyrinth.at(static_cast<size_t>(floor(A.y/sizeOfBlock))).at(static_cast<size_t>(floor(A.x/sizeOfBlock)))
!= 0){
				return sf::Vector2f(A.x, A.y);
			}

			A.x -= 64;                                                                    // - car
le repère est	--->x
			A.y += radTanAngle * 64;

	//                                              y|
		}
	//                                                  |
	}
	//                                                  V
}
	//

//calcule la distance depius un certaine position avec un cerain angle jusque au mur le plus proche
//retourn la distance jusque au mur

float RayCasting::rayCasting(const sf::Vector2f& playerPosition,float angle,const std::vector<std::vector<int>>
&labyrinth,int &blockID, sf::Vector2f& hitPosition, const Algo algo) const{
	float x = playerPosition.x;
	float y = playerPosition.y;

	if (algo == Algo::LINEAR) {

		float d{0.0f};
		angle = angle*_PI/180;
		float cosAngle = cosf(angle);
		float sinAngle = -sinf(angle);
		while (true) {
			try {
				if(labyrinth.at(floor(y/sizeOfBlock)).at(floor(x/sizeOfBlock)) == 1){
					blockID = 1;
					hitPosition.x = x;
					hitPosition.y = y;
					return d; //return distance au mur
				}
```

```
                                        else if (labyrinth.at(floor(y/sizeOfBlock)).at(floor(x/sizeOfBlock)) ==2){
                                                blockID = 2;
                                                hitPosition.x = x;
                                                hitPosition.y = y;
                                                return d;
                                        }

                                        x += 0.5*cosAngle;
                                        y += 0.5*sinAngle;
                                        d += 0.5;                       //simplification de l'equation de pythagor


                                } catch (std::out_of_range& ec) {

                                        return EXIT_FAILURE;
                                }

                        }


        }
        else{           //DDA algo
                //auto line = computeLineCoo(playerPosition, angle, labyrinth);
                auto column = computeColumnCoo(playerPosition, angle, labyrinth);

                //float distanceLine = sqrtf( powf(line.x - x, 2) + pow(line.y - y,2));
                float distanceColumn = sqrtf( powf(column.x - x, 2) + pow(column.y - y,2));
                return distanceColumn;
//              float dist;
//
//              if(distanceColumn < distanceLine){
//                      dist = distanceColumn;
//
//                      if(labyrinth.at(floor(column.y/sizeOfBlock)).at(floor(column.x/sizeOfBlock)) ==1)
//                              blockID = 1;
//                      else
//                              blockID = 2;
//              }
//              else{
//                      dist = distanceLine;
//                      if(labyrinth.at(floor(column.y/sizeOfBlock)).at(floor(column.x/sizeOfBlock)) ==1)
//                              blockID = 1;
//                      else
//                              blockID = 2;
//              }
                return 0;
        }


        return 0; //ne devrait jamais arriver ici car la map est entourée de mur
}
```

12.6 Utility.hpp

```
//
//  utility.h
//  Ray-Casting
//
//

#pragma once
#include <vector>
#include <array>
#include <string>
#include <fstream>
#include <iostream>


bool load_from_file(std::vector<std::vector<int>>& labyrinth, const unsigned int labNum=1);
void show(const std::vector<std::vector<int>>& labyrinth);
```

12.7 Utility.cpp

```
//
//  utility.cpp
//
//
//
//

#include "utility.hpp"
#include "ResourcePath.hpp"


//##############                    the labs            ##############


int sizeOfBlock{64};



bool load_from_file(std::vector<std::vector<int>>& labyrinth,const unsigned int labNum){

        /*

         format :

         1,1,1,1,1,1,1,1
         1,0,0,0,0,0,0,1
         1,0,0,0,0,0,0,1
         1,0,0,0,0,0,0,1
         1,0,0,0,0,0,0,1
         1,0,0,0,0,0,0,1
         1,1,1,1,1,1,1,1

         1 = wall

         */
        std::string extension{".txt"};
        std::ifstream labFile{};

        labFile.open(resourcePath() + "lab" + std::to_string(labNum) + extension);
                std::string line{};
                size_t count{0};


                if(labFile.is_open()){

                        while (std::getline(labFile,line)) {
                                labyrinth.push_back(std::vector<int>());        //nouvelle ligne 2d

                                for (size_t i{0}; i<line.size(); i+=2) {  //saute le caractère ',' et va au prochain
nombre

                                        if(line.at(i)== '1')                                    //utilise  '1' car
.at() retourn un char et ne peux pas etre comparé avec int
                                        {
                                                labyrinth.at(count).push_back(1);
                        //collonne
                                        }
                                        else if(line.at(i) == '0'){
                                                labyrinth.at(count).push_back(0);
                                        }
                                        else if(line.at(i) == '2'){
                                                labyrinth.at(count).push_back(2);
                                        }

                                }
```

```
                                        ++count;
                                }
                                return true;
                        }

                        else{
                                return false;
                        }


}


void show(const std::vector<std::vector<int>>& labyrinth){

        for(auto& i:labyrinth){
                for(auto&j :i){
                        std::cout <<j<<" ";
                }
                std::cout<<std::endl;
        }
}
```

12.8 Game.hpp

```
//
//  Game.h
//
//
//
//
#pragma once
#include <vector>
#include "SFML/Audio.hpp"
#include "RayCasting.hpp"
#include "Player.hpp"



class Game{

        sf::RenderWindow _window;
        sf::Music _music;
        RayCasting _rcEngine;
        Player _player;
        unsigned int _levelID;
        std::vector<std::vector<int>> _labyrinth;
        void (Game:: *statPlayed)(sf::Clock &clock, sf::Time& timeSinceLastUpdate);
        RayCasting::Algo _algo;



private:                                                    //fontion membre
        void processPlayEvent();
        void update(const sf::Time &deltaTime);
        void render();
        void loadLevel(const unsigned int levelID);
        void handleKeyboardInput(sf::Keyboard::Key key, bool isPressed);
        void renderSky();
        void renderFloor();
        void play(sf::Clock &clock,sf::Time& timeSinceLastUpdate );


        void pause(sf::Clock &clock,sf::Time& timeSinceLastUpdate);
        void handlePauseEvent();
        void renderPause();

        sf::Time _deadLine;

public:
        Game();
        void run();


};
```

12.9 Game.cpp

```
//
//  Game.cpp
//
//
//
//
#include "Game.hpp"
#include "utility.hpp"
#include "ResourcePath.hpp"

Game::Game()
:
	_rcEngine(),
	_player(),
	_levelID(1),
	_music(),
	_algo(RayCasting::Algo::LINEAR),
	_deadLine()

{
	sf::ContextSettings settings;
	settings.antialiasingLevel =8;

	_window.create(sf::VideoMode(1200,900), "Ray-Casting",sf::Style::Default,settings);

	load_from_file(_labyrinth,1);
	 statPlayed = &Game::pause;



}



void Game::play(sf::Clock &clock, sf::Time& timeSinceLastUpdate){


	const sf::Time frameTime = sf::seconds(static_cast<float>(1.0/60.0));

	processPlayEvent();
	timeSinceLastUpdate += clock.restart();

	while (timeSinceLastUpdate > frameTime){
		timeSinceLastUpdate -= frameTime;
		processPlayEvent();
		update(frameTime);
	}
	render();

	_deadLine -= sf::seconds(1.0/60);

}



void Game::pause(sf::Clock &clock, sf::Time& timeSinceLastUpdate){
	handlePauseEvent();

	_window.clear(sf::Color::Black);
	renderPause();
	_window.display();
}

void Game::renderPause(){
	_window.clear();
```

```
	sf::Font font;
	if(!font.loadFromFile(resourcePath()+"arial.ttf"))
		return;
	sf::Text txt;
	txt.setFont(font);
	txt.setCharacterSize(50);
	txt.setPosition(_window.getSize().x/2 -txt.getGlobalBounds().width/2, _window.getSize().y/2 -
txt.getGlobalBounds().height/2);
	txt.setString("Pause");

	_window.draw(txt);
	_window.display();

}
void Game::handlePauseEvent(){

	sf::Event event;

	while (_window.pollEvent(event)) {
		switch(event.type){

			case sf::Event::EventType::KeyPressed :

				if(event.key.code == sf::Keyboard::Key::Escape){
					statPlayed = &Game::play;
				}
				break;
			case sf::Event::EventType::KeyReleased:
				handleKeyboardInput(event.key.code, false);
			default:

				;
				break;
		}
	}
}

void Game::run(){

	_deadLine = sf::seconds(60);

	sf::Clock clock;
	sf::Time timeSinceLastUpdate = sf::Time::Zero;

	while (_window.isOpen()){

			(this->*statPlayed)(clock, timeSinceLastUpdate);

	}



}
void Game::processPlayEvent(){

	sf::Event event{};
	while (_window.pollEvent(event)) {

		switch (event.type) {

			case sf::Event::Closed:
				_window.close();
				break;
			case sf::Event::KeyPressed:
				handleKeyboardInput(event.key.code, true);
				break;
			case sf::Event::KeyReleased:
```

```
                                        handleKeyboardInput(event.key.code, false);
                                        break;
                                case sf::Event::Resized:
                                        //std::cout<< event.size.width<<" "<<event.size.height<<'\n';
                                        break;
                                default:
                                        ;
                                        break;
                        }
                }

}


void Game::update(const sf::Time &deltaTime){



                if (_labyrinth.at(floor(_player.position.y/64)).at(floor(_player.position.x/64)) ==2){
                        loadLevel(++_levelID);
                        _player.position = sf::Vector2f(100,100);
                         _player.angle = 360;

                         show(_labyrinth);std::cout<<"\n\n\n\n";
                 }
                auto posi = _player.futurMove(deltaTime);
                if(_labyrinth.at(floor(posi.y/64)).at(floor(posi.x/64)) != 1){
                        _player.move(deltaTime);
                }




}


void Game::handleKeyboardInput(sf::Keyboard::Key key, bool isPressed){

        //############movement#############
        if (key == sf::Keyboard::W) {_player.isMovingUp = isPressed;}
        if (key == sf::Keyboard::S) {_player.isMovingDown = isPressed;}
        if (key == sf::Keyboard::A) {_player.isMovingLeft = isPressed;}
        if (key == sf::Keyboard::D) {_player.isMovingRight = isPressed;}

        //##########angle###############
        if(key == sf::Keyboard::Left){
                _player.angle += 3;

        }
        if(key == sf::Keyboard::Right){
                _player.angle -= 3;
        }

        //########         speed    #######
        if(key == sf::Keyboard::Up)                     _player.speed+=3;
        if(key == sf::Keyboard::Down)         _player.speed-=3;



        if(key == sf::Keyboard::N and !isPressed){

                if(_levelID != 4)
                        loadLevel(++_levelID);
```

```
		}

		if(key == sf::Keyboard::Key::L and !isPressed){

			loadLevel(_levelID);
		}

		if (key == sf::Keyboard::P and !isPressed) {
			if(_levelID != 1)
				loadLevel(--_levelID);

		}
		if(key == sf::Keyboard::Escape && isPressed){
			statPlayed = &Game::pause;
		}
		/*if(key == sf::Keyboard::Space && isPressed){
			_algo = (_algo == RayCasting::Algo::LINEAR) ? RayCasting::Algo::DDA :
RayCasting::Algo::LINEAR;
			std::cout <<"changes algo\n";

		}*/

		if(key == sf::Keyboard::Key::T){
			_deadLine.asSeconds() < 0 ? _deadLine = sf::seconds(10) : _deadLine += sf::seconds(10);
		}
		if(key == sf::Keyboard::Key::G)
			_deadLine -= sf::seconds(10);

		if(key == sf::Keyboard::H)
			std::cout<< _player.position.x<<" "<<_player.position.y<<"\t|\t"<< _player.angle<<'\n';




}


void Game::render(){
		sf::Font font;
		if(!font.loadFromFile(resourcePath()+"arial.ttf"))
			return;
		sf::Text time,level;
		time.setFont(font);
		level.setFont(font);
		time.setCharacterSize(25);
		level.setCharacterSize(25);
		level.setString(std::string("level: ")+std::to_string(_levelID));
		level.setPosition(_window.getSize().x- level.getGlobalBounds().width*1.2,time.getPosition().y);
		_deadLine.asSeconds() <= 0 ? time.setString("Time's up !") : time.setString
(std::to_string(static_cast<int>(_deadLine.asSeconds()))) ;

		_window.clear();
		auto sizeWin = _window.getSize();
		unsigned int barCount{1};
		unsigned int nbRect{400};
		renderSky();
		//renderFloor();

		for (float i= _player.angle - 30 ; i < _player.angle + 30; i+= 60.0/nbRect) {
			int blockID;
			sf::Vector2f hitPosition;
			float distance = _rcEngine.rayCasting(_player.position, i, _labyrinth, blockID, hitPosition, _algo);

			if(distance < .5){
				distance +=1.1f;
			}
```

```
                distance *= cosf((i- _player.angle) * 3.141592/180);    //correct fish-eyes

                sf::RectangleShape bar{sf::Vector2f( sizeWin.x/nbRect , (64/distance) */*692*/
_window.getSize().x/(2*tanf(30*3.1415/180)))};          //cstr prends la taille de l'objet comme argument


                /*

                 le 692 est la distance du joueur jusque au plan de projection definit comme win.x/2*tanj(30)

                 */

                bar.setPosition((nbRect-barCount)* sizeWin.x/nbRect, sizeWin.y/2 - bar.getSize().y/2);
                if(blockID ==1){
                        bar.setFillColor(sf::Color(

static_cast<sf::Uint8>(floor(((1.6*255.0)/distance)*64)) >=sf::Uint8(250) ?
250:static_cast<sf::Uint8>(floor(((1.6*255.0)/distance)*64)),
                                                                                        0,
                                                                                        0));


                        std::cout <<static_cast<sf::Uint8>(floor((1.6*255.0)/(distance)*64))<<'\n';
                }
                else if (blockID ==2){
                        bar.setFillColor(sf::Color(204,127,49));

                }



                _window.draw(bar);
                _window.draw(time);
                _window.draw(level);
                ++barCount;


        }
        _window.display();

}




void Game::loadLevel(const unsigned int levelID){

        if(levelID == 5 or levelID == 0)
                return;

        _labyrinth.clear();
        load_from_file(_labyrinth,levelID);
        show(_labyrinth);
        std::cout << "\nlevel "<<levelID << "loaded";

        _deadLine = sf::seconds(60);

}




void Game::renderSky(){
```

```
	/*
	 format du vertex array;

	 1                              2


	 0                              3

	 */

	sf::VertexArray sky{sf::Quads,4};
	sky[0].position = sf::Vector2<float>(0, _window.getSize().y/2);
	sky[0].color = sf::Color(20,20,255);

	sky[1].position = sf::Vector2f(0,0);
	sky[1].color = sf::Color(0,0,255,180);


	sky[2].position = sf::Vector2f(_window.getSize().x,0);
	sky[2].color = sf::Color(0,0,255,180);


	sky[3].position = sf::Vector2f(_window.getSize().x, _window.getSize().y/2);
	sky[3].color = sf::Color(20,20,255);




	_window.draw(sky);

}

void Game::renderFloor(){
	/*

	 format est le même que dans void Game::renderSky();


	 */

	sf::VertexArray floor{sf::Quads,4};

	floor[0].position = sf::Vector2<float>(0,_window.getSize().y);
	floor[0].color = sf::Color::Green;


	floor[1].position = sf::Vector2f(0,_window.getSize().y/2);
	floor[1].color = sf::Color::Green;



	floor[2].position = sf::Vector2f(_window.getSize().x,_window.getSize().y/2);
	floor[2].color = sf::Color::Green;



	floor[3].position = sf::Vector2f(_window.getSize().x, _window.getSize().y);
	floor[3].color = sf::Color::Green;




	_window.draw(floor);
}
```