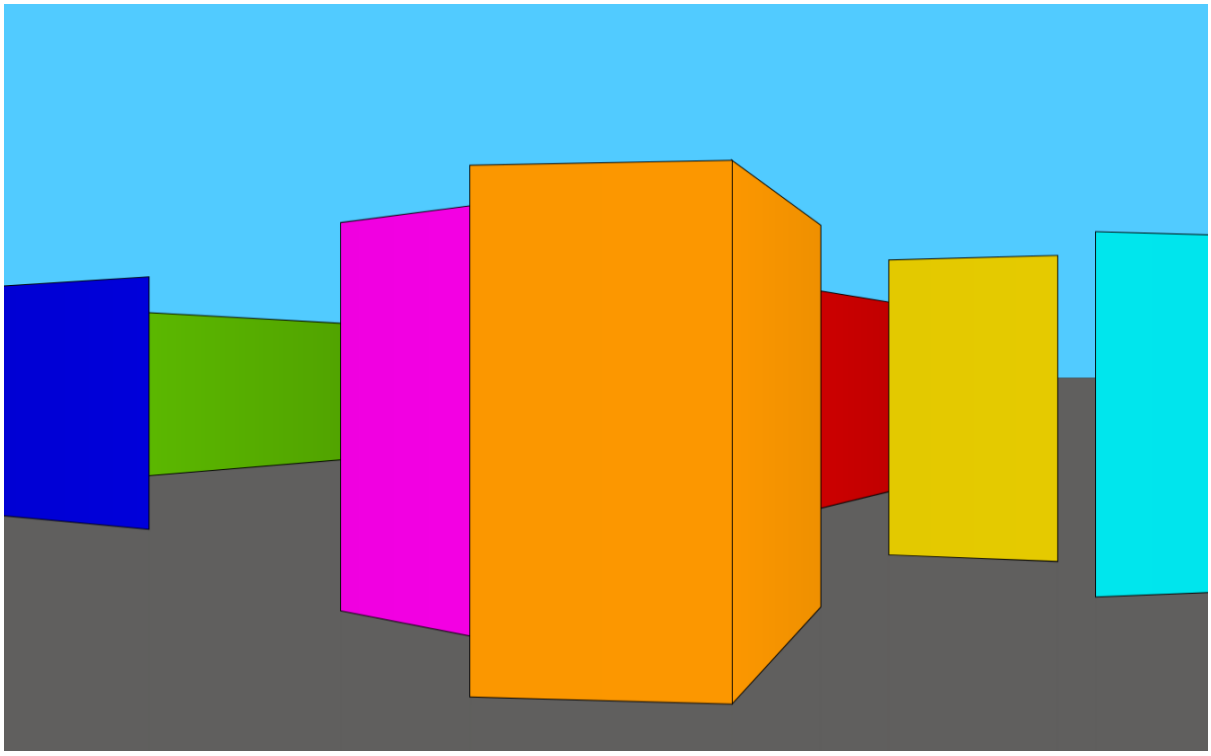
MAPR:
Marcus' Algorithm for Pseudo-3D Rendering

Creating a pseudo-3D render engine from scratch

Marcus Hamelink



24th march 2022

Contents

1 Introduction

After learning to program in Python in March 2020 at the beginning of confinement, I became fascinated with all things computer. I kept learning new programming concepts and, during the summer, I started learning Javascript to code websites. I was particularly interested in creating games.

At the beginning of my third year of high school, I discovered *Raycasting*, which is a "3D computer graphics technique"[1]. This is what made possible the first 3D video games like *Doom* or *Wolfenstein* 3D in the early 1990s. Based on a 2D space and not on a 3D world, the method allows to simulate a 3D perspective without the complex calculations of the third dimension. I will explain in more detail how this technique works in the development.

I then started to learn how to implement this method myself by following guides on the internet and that's when I decided to make it my graduation project. Initially, my plan was to create a game engine with a lot of features that would allow anyone to draw environments on a map and explore them in perspective. There would also be easy access to my code to program new games with my basics.

Soon after, I realized that the method I was using did not appeal to me visually and I wanted to exploit the capabilities of modern computers. So I decided to try to create my own technique following the principles of Raycasting. The concepts of vector geometry that I learned in my third year of high school math classes helped me tremendously in applying the math to my code.

After I turned in this graduation project, I continued to work on it in my spare time. This document presents the original explanations with mentions of the features I improved or added.

In this report, I will first detail how to access a demo of the final project and a guide to its use. I will then explain the principles of the current Raycasting methods. Finally, I will describe the new features that make up my method.

[1] Raycasting. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 22 juillet 2021 à 22:17. [Consulté le 2 octobre 2021]. Disponible à l'adresse : https://fr.wikipedia.org/wiki/Raycasting

2 Information about the project

2.1 Choice of programming language

I wrote the code for this project with *VSCode* which is a text editor for programming. It can be run by opening the file *index.html* with any internet browser *(Chrome, Firefox, Safari, etc.)*.

I chose to develop my project as a website that would then be accessible with any internet browser, including those found on a smartphone. This choice was motivated by the desire to easily share my creation with friends. A modern website is usually developed by combining 3 different programming languages.

- HTML** (HyperText Markup Language) to describe the static content (formatted text, images, etc.) included on the page.
- CSS** (Cascading Style Sheets) to customize the appearance of the content defined in the HTML part, taking into account the device on which the site is displayed To realize the importance of *CSS*, here is a comparison of the computer interface with and without style applied:

Project page with applied style[2]

Project page without style

- Javascript** A multi-purpose programming language that allows the programmer to dynamically manipulate the web page by modifying the displayed elements.

In order to facilitate the development in these three different languages, I chose to use the VSCode text editor, a very common choice for this type of project.

This project has been realized without no dependencies, that is to say that I did not use any downloaded libraries to help me.

2.2 Instructions for using the application

My project page is divided into three parts:

- The 3D projection of the environment from the player's first person perspective
- The bird's eye view of the interactive level
- The scrollable settings section allowing the user to customize the appearance of the projection

On a device with a keyboard, the player can be moved with the W, A, S and D keys. To direct the player's gaze from left to right or up and down, the arrow keys are used. On a touch screen where there is no keyboard, virtual keys are drawn on the 3D projection.

[2]Démonstration du projet hébergé sur https://mapr.me/

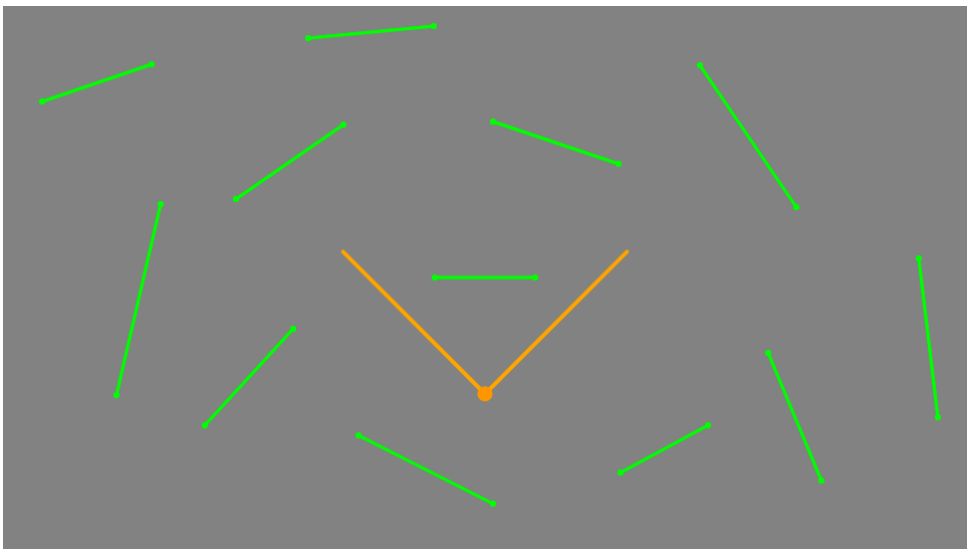
Obstacles can be drawn on the bird's eye view by clicking and dragging the mouse (or finger on touch screens). By holding the *Shift* key while drawing an obstacle, the cursor will automatically snap to the end of the nearest obstacle. This feature also exists on touchscreens, but it is done automatically when the endpoint is already placed.

The settings section contains everything you need to change the appearance of the obstacles (color, height position, etc.). However, a quick selection of different plans already created are available at the beginning of the section. These plans have been drawn to show the capabilities of my program. There are also several options that make it easier to show the project's features and that helped me to solve some display problems. If this section is too small on the page, there is a button in the bottom right corner that allows you to enlarge it.

3 Description of the Raycasting technique

I first discovered Raycasting thanks to a video by Daniel Shiffman, a creator on YouTube who runs *The Coding Train* channel, where in one of his series, *Coding challenge*, he does small projects in order to educate his listeners. His video on Raycasting[3] was a good introduction to the subject for me. I spent many hours understanding the principles so I could apply them in my own projects. I decided to make it my Matura work, because I was curious to know how I could create my own games with it.

The most common method and the one I coded first is the following: We start from a 2-dimensional plane where the player is represented by a point and two segments indicating the extent of his field of view (indicated below in orange). Around him are placed lines that represent obstacles (in green). The player can move and orient himself freely on the map.



2D shot of the player surrounded by obstacles

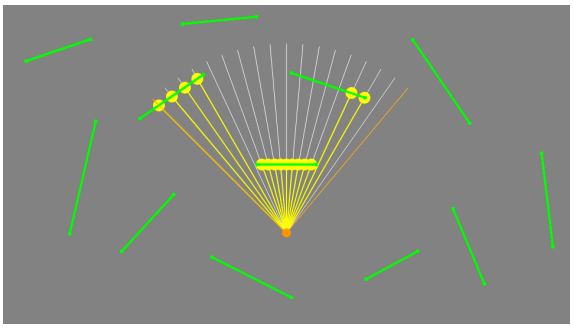
In my program, all the obstacles as well as the player are represented in my program as "Javascript objects". That is, each of them has properties that correspond to them individually. Such as the x, y coordinates on the plane, the orientation, the color, etc.

Then rays are generated from the player between the ends of his field of view. When the rays intersect with an obstacle, the point of intersection where the two segments meet is calculated[4].

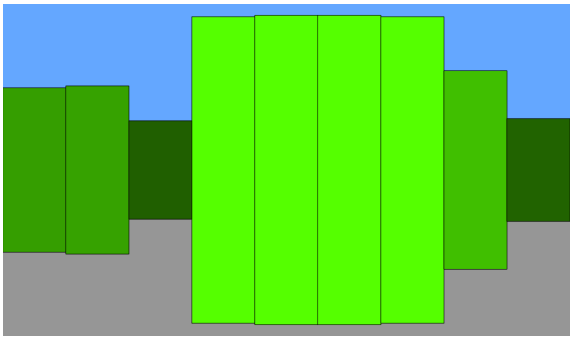
The 2D plane is only a representation of the calculations made for the 3D rendering. Thus, we can walk on the plane as if we were in the place of the point and we perceive all the obstacles with our own eyes on the ground.

[3] SHIFFMAN Daniel, 2019 Coding Challenge #146 : Rendering Raycasting [enregistrement vidéo]. YouTube [en ligne]. Disponible à l'adresse : https://youtu.be/vYgIKn7iDH8.

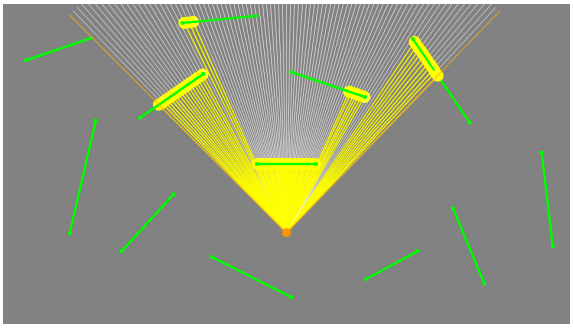
[4] Le développement de la formule mathématique de l'intersection se trouve en Annexe a)



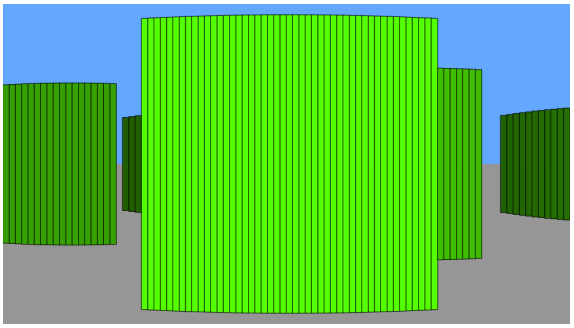
2D plan with 18 calculated intersection points



3D perspective calculated from the 2D plan



2D plan with 90 intersection points



Same perspective with more details

In the images above, we can see a representation (left) of the 2D plane and (right) the 3D perspective.

The 3D perspective screen is divided horizontally by the number of rays generated. Thus, for each of these rays, a part of this 3D screen is associated and the distance between the player and the intersection with an obstacle is used to draw a rectangle with a height proportional to the distance. That is, if the distance is greater, the intersection is further away from the player, so the height of the rectangle will be smaller and vice versa if the distance is smaller. This simulates the perspective we see with our own eyes.

The images above are a demonstration of this concept with on one side what happens on the 2D plane and on the other side what we can see after the image calculations. To calculate the height of the rectangle, there is another way that I apply in my new method for these 3D image calculations.

There are also other methods to code a renderer with Raycasting, like starting from a grid to present the obstacles as cubes [5], the basic principle is the same. It is from this principle explained before that I created a new method that I liked better.

[5] Il existe un guide complet pour réaliser cette méthode, Raycasting, Lode Vandevenne, 2004-2020, Disponible à l'adresse : https://lodev.org/cgtutor/raycasting.html

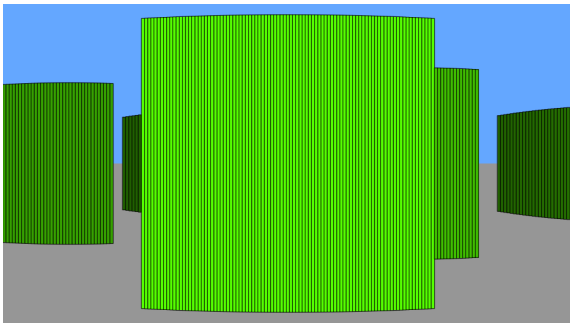
4 The principles of my new method

4.1 Concept

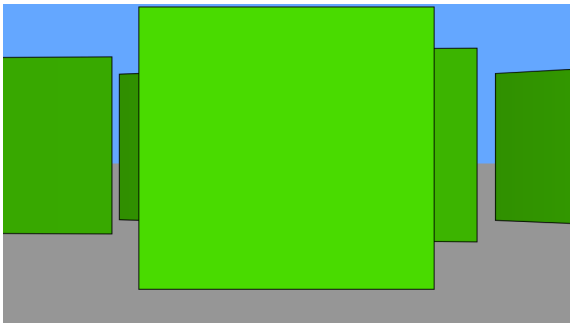
What I didn't like with the initial method was the pixelated aspect of the 3D rendering. When the player moves in another direction, we see a change in the rectangles that are drawn, breaking the perspective that we perceive with our own eyes.

On the other hand, the more rays we use, the more rectangles are drawn. However, an increase in the number of calculations per image quickly slows down the game experience and limits the possibilities, especially if, in addition to the number of rays, the number of obstacles is also increased. Moreover, you have less freedom if you want, for example, to draw an outline around the obstacles (since there is no single shape, but a multitude of rectangles, each with its own outline).

So I thought I could use polygons to draw these obstacles, because no matter what the orientation of the obstacles is, the resulting perspective will always be a polygon with four corners. The problem is that with the previous method, we don't know if two rays intersect with the same obstacle. This could have been detected eventually, but another problem arises when the corner of an obstacle is between two rays. If the player moves a little bit, the corner of this obstacle in the 3D perspective will not be at the same place.



3D perspective of the old method 180 calculated rays[^oldvsnew]



3D perspective of my new method using 10 rays

It was these little details that made me realize that if I wanted to change my method, I had to remove this generation of all these rays. Keeping this idea of drawing polygons, we can deduce a new way to detect obstacles.

In this section I will present the different functions used by my program. These functions are executed on each of the obstacles.

4.2 Detecting obstacles in the field of view

To start, we need to group all the obstacles that are in the player's field of view. The function `isInsideFOV()` (which means "inside the field of view" in English) first assigns two new variables `p` and `h` which are two vectors that respectively connect the player's position to each of the obstacle's extremities (`p` which means position and corresponds to the vector of the first point of the obstacle and `h` which means header, also known as the obstacle's header)

Next, we perform the verification by repeatedly using `isClockwiseOrder()`[6], a function that takes two vectors and returns true if the first is oriented clockwise with respect to the second.

[6] Voir point b) de l'Annexe pour une explication détaillée de cette fonction

In my program, if one end of the obstacle is both to the right of the left field-of-view vector and also to the left of the right field-of-view vector, then the function will return true. This is verified for both ends. In addition, if the player observes an obstacle at close range and the extremities go outside the field of view, the function also checks if there is an intersection[7] between the obstacle and each of the field of view vectors.

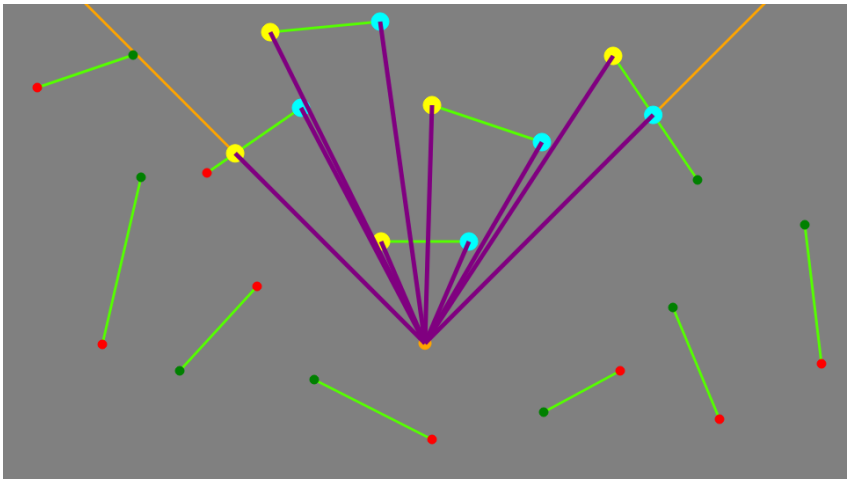
This function allows the next calculations to be performed only on the obstacles that matter.

```
function isInsideFOV() {
// Creating vector going from player to wall's first vertex
    this.p = {
        'x': this.pos.x - player.pos.x,
        'y': this.pos.y - player.pos.y
    };

// Creating vector going from player to wall's first vertex
    this.h = {
        'x': (this.pos.x + this.dir.x) - player.pos.x,
        'y': (this.pos.y + this.dir.y) - player.pos.y
    };

    this.p.dist = Math.sqrt((this.p.x) ** 2 + (this.p.y) ** 2);
    this.h.dist = Math.sqrt((this.h.x) ** 2 + (this.h.y) ** 2);

    if ((isIntersectionFovW(player.fov.v1, this) ||
            isIntersectionFovW(player.fov.v2, this)
        ) ||
        (isClockwiseOrder(player.fov.v1.dir, this.p) &&
            !isClockwiseOrder(player.fov.v2.dir, this.p) &&
            isClockwiseOrder(player.fov.v1.dir, this.h) &&
            !isClockwiseOrder(player.fov.v2.dir, this.h) &&
            (this.p.dist <= player.fov.v1.dir.length*2 &&
                this.p.dist <= player.fov.v2.dir.length*2) &&
            (this.h.dist <= player.fov.v1.dir.length*2 &&
                this.h.dist <= player.fov.v2.dir.length*2)
        )
    ) return true;
}
```

4.3 Arrange the vector data for each obstacle

The `processFOV()` function, which is executed only after verifying that the obstacle is in the field of view, modifies the $\vec{p}$ and $\vec{h}$ vectors to make the data easier to process later.

[7] La fonction *isIntersection()* dont l'explication se trouve dans l'Annexe a)



Visualization of obstacles intersecting with the field of view vectors (yellow for the left end of the obstacle and blue for the right end)

First we make sure that $\vec{p}$ is to the left of $\vec{h}$; if not, we reverse the data. Then we check if the obstacle crosses one of the vectors of the field of view. If it does, we define again one of the vectors so that we have only the part of the obstacle which is inside the field of view.

```
function processFOV() {
    // make sure v1 is always to the left of v2
    if (!isClockwiseOrder(this.p, this.h)) {
        let vtemp = this.p;
        this.p = this.h;
        this.h = vtemp;
    }

    //  redefine wall if it intersects with LEFT fov ray
    if (!isClockwiseOrder(player.fov.v1.dir, this.p) &&
            isClockwiseOrder(player.fov.v1.dir, this.h)) {
        this.p = intersectionFovW(player.fov.v1, this);
        this.p.x -= player.pos.x;
        this.p.y -= player.pos.y;
    }

    //  redefine wall if it intersects with RIGHT fov ray
    if (!isClockwiseOrder(player.fov.v2.dir, this.p) &&
            isClockwiseOrder(player.fov.v2.dir, this.h)) {
        this.h = intersectionFovW(player.fov.v2, this);
        this.h.x -= player.pos.x;
        this.h.y -= player.pos.y;
    }
    return true;
}
```

5 Implementation

5.1 Projection calculations from 2D to 3D

Now that we have all the necessary data, we can finally perform the calculations that will lead to a 3D perspective. We will take the vectors $\vec{p}$ et $\vec{h}$ as well as the director vector of the player (the one which indicates his orientation) to then normalize them in order to be able to work with their unit values and not have to worry about their norm.

We will then apply the formula for the angle between two vectors. The angle `v1xangle` corresponds to the angle between the vector $\vec{p}$ and the player's direction vector. The angle `v2xangle` corresponds to the angle between $\vec{p}$ and $\vec{h}$. The angles are calculated in this way so that we have values going from negative to positive. We have a relative angle between the vectors $\vec{p}$ and $\vec{h}$, which we denote respectively $\angle p$ and $\angle h$. So we have $\angle p$ which is relative to the player's director vector as well as $\angle h$ which is just an addition of $\angle p$ and the relative angle between the two.

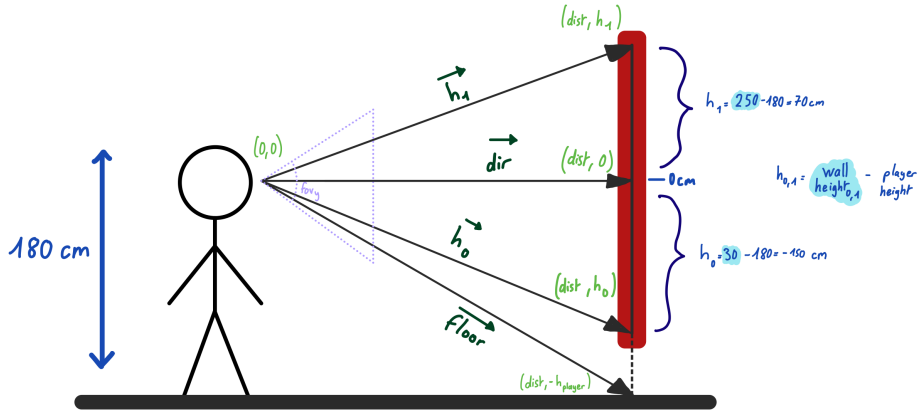
Finally, we define the variables for each of the corners of the polygon that will be generated to draw the obstacle. We calculate `x1` and `x2` with a proportion between the previously calculated angle and its relative place on the screen and `h1` and `h2` are calculated thanks to the `calculateHeight()` function which I will talk about in the next section.

```
function calculate3D() {
    const fovamount = player.fov.xamount;

    this.p = vectorNormalize(this.p);
    this.h = vectorNormalize(this.h);
    const dir = vectorNormalize(player.dir,
                    Math.sqrt((player.dir.y) ** 2 + (player.dir.x) ** 2));
    let v1xangle = Math.acos(vectorDotProduct(this.p, dir));
    let v2xangle = Math.acos(vectorDotProduct(this.h, this.p));

    // correct sign depending on side of v1/v2
    if (!isClockwiseOrder(dir, this.p)) v1xangle = -v1xangle;
    if (!isClockwiseOrder(this.p, this.h)) v2xangle = -v2xangle;

    v2xangle = v2xangle + v1xangle; // make v2 relative to v1

    this.p.dist *= Math.cos(v1xangle);
    this.x1 = degrees(v1xangle) * canvas.width / fovamount;
    this.h1 = this.calculateHeight(this.p);

    this.h.dist *= Math.cos(v2xangle);
    this.x2 = degrees(v2xangle) * canvas.width / fovamount;
    this.h2 = this.calculateHeight(this.h);

    if (this.h1 > 10000) this.h1 = 10000;
    if (this.h2 > 10000) this.h2 = 10000;
}
```

5.2 Calculate the relative size of an obstacle in the field of view

After calculating the distance between the player and a point on an obstacle, we can then determine the height that this slice of obstacle will take on the 3D perspective.

The values we need are : *dist* the distance computed earlier using the `intersection()` function, h_0 the difference between the bottom of the obstacle and the height of the player, h_1 the difference between the top of the obstacle and the height of the player, floor the difference between the height of the player and the ground, $canvas_{height}$ the height in pixels of the screen in 3D perspective, and FOV_y the maximum vertical field of view value. To better illustrate these points, we will represent them as vectors, but we only need the vertical values, because the horizontal value is the same. The height of the player and the height of the top and bottom of each obstacle are predefined and can be changed freely.



Schematic representation of values

Thus, we can calculate the proportion between the height of these points and the relative height of the screen in 3D perspective using the following formulas:

$$\text{floor}_{\text{height}} = \frac{\text{floor}_y \cdot \text{canvas}_{\text{height}}}{\tan(FOV_y) \cdot dist} \quad \text{h}_{0\text{height}} = \frac{\text{h}_{0y} \cdot \text{canvas}_{\text{height}}}{\tan(FOV_y) \cdot dist} \quad \text{h}_{1\text{height}} = \frac{\text{h}_{1y} \cdot \text{canvas}_{\text{height}}}{\tan(FOV_y) \cdot dist}$$

Then, we can use a proportion between the angle and the height of the 3D perspective screen. The function *calculateHeight()* finally returns these values calculated by this proportion.

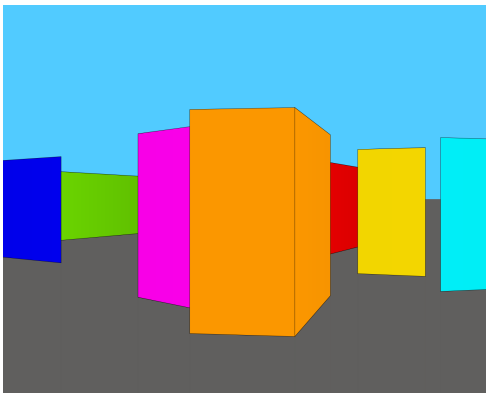
At the time of handing in this project as a Maturité assignment, this part of my program was more complex than it needed to be, as I tried to find measurements through the angles. This gave a projection that at first sight looked good, but one soon realized that something was wrong. Finally, the proportionality between the different heights gives a simpler calculation and heights that look like what we would perceive with our own eyes

```
function calculateHeight(v) {
    const h0 = vectorCreate(v.dist, this.height0 - player.height);
    const h1 = vectorCreate(v.dist, this.height1 - player.height);
    const floor = vectorCreate(v.dist, -player.height);

    return {
        'floor': (floor.y * canvas.height)
```

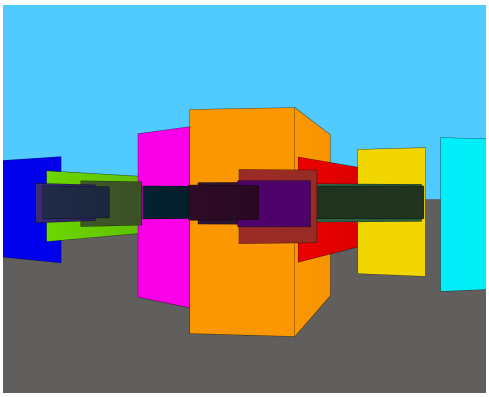
```
                    / (Math.tan(radians(player.fov.yamount)) * v.dist),
        'h0':     (h0.y * canvas.height)
                    / (Math.tan(radians(player.fov.yamount)) * v.dist),
        'h1':     (h1.y * canvas.height)
                    / (Math.tan(radians(player.fov.yamount)) * v.dist),
        'sat': 1,
        'dist': v.dist
    };
}
```

5.3 Sorting the obstacles

Currently, we have a list of obstacles that we have to draw in a random order. Without processing this list, we can end up with a rendering like the following.



Rendering with obstacle sorting

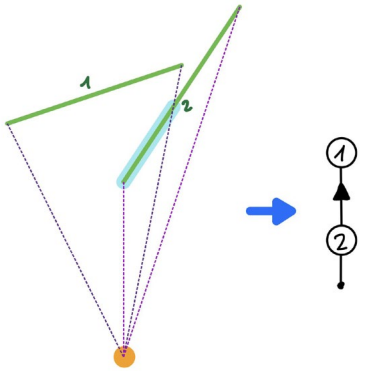


Rendering without obstacle sorting

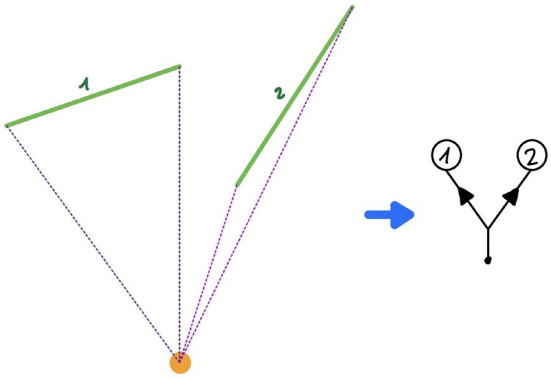
We therefore need to sort these obstacles according to their distance from the player. This takes into account that we need to be able to determine whether an obstacle is behind or in front of another one in relation to the player. There is also no single value to compare, as each obstacle has two coordinates x and y and everything must be calculated relative to the player.

A totally ordered set is a set whose elements are all comparable to each other. For example, a set of numbers that two by two can all be compared to order them from small to large.

In our case, the set of obstacles is not of total order, because there can be two pairs, where neither is ahead of the other. In reality, this set is called a *Partially Ordered Set* (or *POSET*). To sort, we use what is called a *graph*. This is a way of representing the hierarchy of relationships between objects. Each point, called a *vertex*, represents an object, and each of these vertices is linked by what is called an *edge*. By relating each obstacle in a graph, the presentation of the graph looks like a kind of tree.

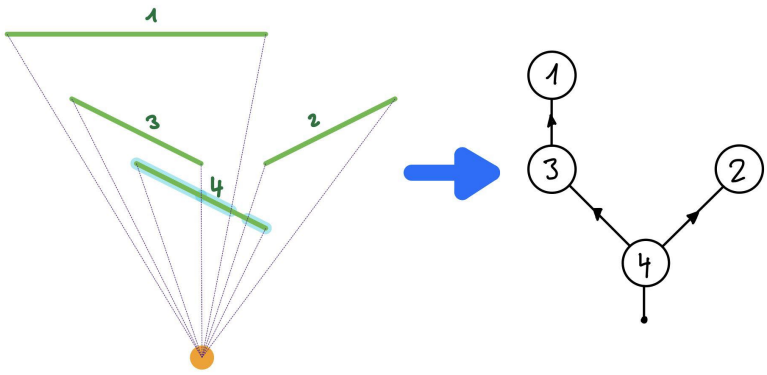


Two obstacles that have a relationship with the player



Two obstacles that have no relation to each other with respect to the player

In the images above are two examples with a pair of obstacles and their representation as a graph. representation in the form of a graph. On the left, obstacle 2 appears in front of obstacle 1, so it is placed before the first one in the graph. On the right, neither of them is in front of or behind the other, so we will represent them by two diverging edges, called branches, on the graph.



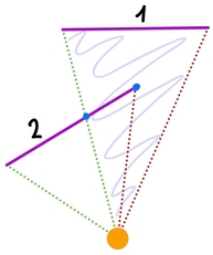
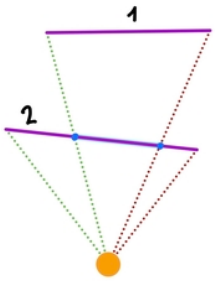
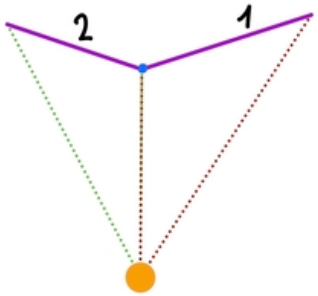
Case of a graph with several branches

In the image above, the two cases are merged. We notice that the obstacle 4 appears in front of all the others, without any relation between them.

5.3.1 Creation of the graph

5.3.1.1 Comparison criteria between two obstacles In order to create the graphs, we need a function that can compare two obstacles and determine if they are one in front of the other or not. This function is called `v1HigherThanv2()`. The comparison must be done in such a way that all possibilities of obstacle placement are taken into account.

The current version takes into account three cases.

Case 1	Case 2	Case 3
		
One of the ends of an obstacle in front of the other	One obstacle hiding the other	Obstacles glued, but not superimposed

Case 1) First, there is a check to see if one end of obstacle 2 is in front of the other. This can be done by considering that the two ends of obstacle 1 and the point where the player is located form a triangle. Then, if the point of obstacle 2 is indeed inside the triangle, then we know that obstacle 2 is in front of 1.

Case 2) We also check for intersections between obstacle 2 and the vectors connecting the player to obstacle 1. If this is the case, it means that obstacle 2 is necessarily in front of 1.

Case 3) Finally, we check if the obstacles share extremities between them, because this case does not return true in the other checks. When implementing this part of the code, I had difficulties in comparing the coordinates of two points, because there is a comparison of two decimal numbers.

5.3.1.2 Accuracy errors of floating points

The computer stores these numbers, which are also called *floating points*, in a scientific notation in base 2. Although this way offers us the possibility of working with decimals, it also comes with some problems which are the precision errors. Indeed, the addition of two floating points $0.1 + 0.2$ which would normally be 0.3, does not give the same result in a program. This is due to the representation of 0.1 and 0.2 which are, depending on the computer, equivalent to, for example, 0.100000001 or 0.20000012. These numbers are roundings that are unfortunately necessary, because a computer cannot represent these numbers with such precision.

```
const EPSILON = 0.01
function isSame(v1, v2) {
    return  Math.abs(v1.x - v2.x) < EPSILON &&
            Math.abs(v1.y - v2.y) < EPSILON
}
```

To counter this problem, when we want to check the equality between two floating points, we take the difference between them and check if it is smaller than a certain constant. In my program, I include this comparison under the function `isSame()` which uses a constant called *Epsilon* which is 0.01. This function returns true if the difference between the coordinates of the two points gives a number smaller than the constant. This overcomes the problem of precision, because a certain margin of error is included.

```
function v1HigherThanv2(w1, w2) {
    // CAS 3
    if (isSame(vectorMult(w1.p, w1.p.dist), vectorMult(w2.h, w2.h.dist)) ||
        isSame(vectorMult(w1.h, w1.h.dist), vectorMult(w2.p, w2.p.dist))) {
        return false
```

```
    }

    // CAS 1 et 2
    return  isIntersectionVectors(PltoW1P, W2ptoW2h, w1.index, w2.index) ||
            isIntersectionVectors(PltoW1H, W2ptoW2h, w1.index, w2.index) ||
            ptInTriangle(W2p, player.pos, W1p, W1h, w1.index, w2.index) ||
            ptInTriangle(W2h, player.pos, W1p, W1h, w1.index, w2.index)
}
```

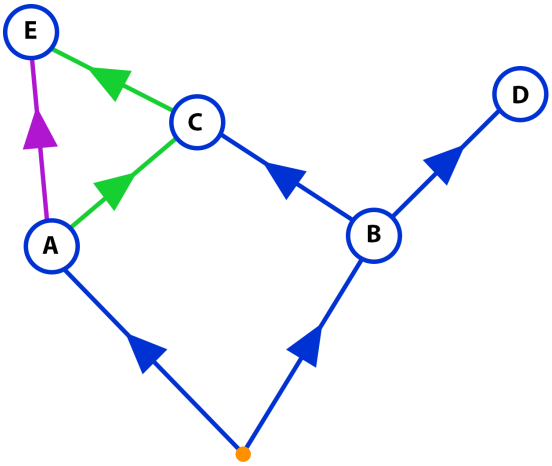
5.3.2 Compare all the obstacles together

Now that we have established a comparison between two obstacles, we must be able to associate each of them together and thus create a graph. The function `wallsToGraph()` will associate each obstacle with another so that each pair is unique. This is a combination whose order is not important, because the comparison with the `v1HigherThanv2()` function is done in both directions in order to know in which order to place the obstacles in the graph. If they are unrelated, they will be added to the graph, but as individual vertices rather than an edge in one direction.

```
function wallsToGraph(w) {
    if (w.length < 2) return [];

    const g = new Graph();
    for (let i = 0; i < w.length - 1; i++) {
        for (let j = i+1; j < w.length; j++) {
            if (v1HigherThanv2(w[i], w[j])) {
                g.addEdge(j, i);
            } else if (v1HigherThanv2(w[j], w[i])) {
                g.addEdge(i, j);
            } else {
                g.addVertex(i);
                g.addVertex(j);
            }
        }
    }
    let final = g.topologicalSort().reverse()
    return final;
}
```

5.3.3 Interpretation of the graph

Once the graph has been created, it must be interpreted in order to draw the obstacles from the farthest to the closest. To order this graph, we will use a traversal algorithm called *Depth First Search* to access all the vertices in their order in the graph. We will be able to form a list of the obstacles that we have to draw in the order of the farthest from the player to the closest. This traversal consists in starting from each point and traversing all the other vertices to find out the farthest we can go from this first point. In other words, each point will be assigned a distance value that represents the longest distance that can be covered to reach the end of the graph. This technique is possible because the graph is acyclic. That is to say that by following the order of the edges, we will not be able to return to the vertices we have traversed.



Example of a path from a vertex

Here, starting from vertex A, there are two possible paths to the end of the graph. By going to E, the distance is 1, while by passing through C and then reaching the end, the distance is 2. In this example, we assign a distance of 2 to vertex A. Thus, we can draw the obstacles in the order of their distance as previously evaluated.

5.4 Drawing the obstacles

To finally draw the obstacles on the 3D perspective, we take each obstacle in the order of the list created by the sorting in the previous section. We use the values calculated previously x_1, x_2, h_1 and h_2 which define the corners of the final polygon. The color is determined by a variable `this.hex` (hexadecimal value of the color) which is specific to the obstacle. We use a quadratic function that takes as abscissa the distance of the vectors `p` and `h` (vectors going from the player to each of the ends of the obstacle) in order to calculate the level of darkness that we will apply by a gradient between the two variations of the initial color. This gives an air of depth, because the further away from the player a part of the obstacle is, the darker it will be.

```
function display3D() {
    const maxl = 0;
    const minl = -0.75;
    const a = canvas2D.width/1.2;
    const n = 2;

    let L1 = -((this.p.dist / a) ** n);
    if (L1 < minl) L1 = minl;
    if (L1 > maxl) L1 = maxl;

    let L2 = -((this.h.dist / a) ** n);
    if (L2 < minl) L2 = minl;
    if (L2 > maxl) L2 = maxl;

    const grd = ctx.createLinearGradient(
                this.x1 + canvas.width / 2, canvas.height / 2,
                this.x2 + canvas.width / 2, canvas.height / 2);
    grd.addColorStop(0, shadeHexColor(this.hex, L1));
    grd.addColorStop(1, shadeHexColor(this.hex, L2));
```

```
    ctx.fillStyle = grd;

    polygon([this.x1 + canvas.width / 2, this.h1.h0 + canvas.height / 2,
    this.x1 + canvas.width / 2, this.h1.h1 + canvas.height /2,
    this.x2 + canvas.width / 2, this.h2.h1 + canvas.height /2,
    this.x2 + canvas.width / 2, this.h2.h0 + canvas.height /2
    ], `grd`, 2);
}
```

6 Conclusion

To summarize, the implementation of my new method for a 3D renderer with Raycasting has been successful. I started with the idea of creating a game engine from the method used in the first 3D video games like *Wolfenstein 3D* or *Doom*. After being dissatisfied with the look of the 3D rendering, I pushed the limits that would have been impassable with the computing capabilities of the computers of the time and I was able to modernize the technique while keeping the simplicity of the concept. I have created a new method that is based on a simple idea of the old one, but that fully exploits the capabilities of today's devices.

Through the realization of the project, I learned to apply vector geometry concepts that I was taught in school in parallel. I also studied graph theory to understand sorting algorithms and POSETs. Thanks to this project, I learned to confront problems outside of my knowledge and look for solutions independently.

The creation of the graphical interface is also a part of the final version that I am very proud of. I didn't have much experience in this area and I learned techniques that will be extremely useful for my future projects. I had to figure out how to create a dynamic interface that is guaranteed to scale to the size of the screen as well as the ability to access the same functionality on both the computer and the touch screen.

The final version of my project looks exactly like I imagined it would when I started working on it a year and a half ago. Because of this, I get great satisfaction from being able to show my website from my phone to my friends and family.

After spending several hundred hours working on this project, I'm extremely proud of the result and look forward to using it as a springboard to explore new programming techniques. The lessons I learned in the process are very useful for the future. Now you can really dive into this world that seems to be a 3D space, but really isn't.

7 Appendix

7.1 a) Find the intersection of two segments

Let Px, Py be the coordinates of the point of intersection of the two segments defined by (x_1 , y_1) and (x_2 , y_2) for the first and (x_3 , y_3) and (x_4 , y_4) for the second. Thus, we pose t, u such that :

$$t = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2) * (y_3 - y_4) - (y_1 - y_2) * (x_3 - x_4)} \quad \text{and} \quad u = \frac{(x_1 - x_3)(y_1 - y_2) - (y_1 - y_3)(x_1 - x_2)}{(x_1 - x_2) * (y_3 - y_4) - (y_1 - y_2) * (x_3 - x_4)}$$

Note that the denominator is the same for the calculation of u and t.

These two formulas are the result of a development of a calculation of determinants [8] (notion associated to matrices).

Before calculating the point P_x and P_y, where the intersection of the two segments is located, we can check if this intersection exists.

The intersection of the two segments exists if $0.0 \leq t \geq 1.0$ et $0.0 \leq u \geq 1.0$. This allows to ignore the result and only know if there is an intersection in order to optimize the program.

On the other hand, we can still compute P_x and P_y such that:

$$(P_x, P_y) = (x_3 + u(x_4 - x_3), y_3 + u(y_4 - y_3))$$

In my code, I have two functions related to intersection: *Intersection()* which determines the coordinates of an intersection point of two segments if it exists, and *isIntersection()* which does the same without the same without calculating the coordinates.

```
function isIntersection(w1, w2) {
    const x1 = w1.pos.x;
    const y1 = w1.pos.y;
    const x2 = x1 + w1.dir.x * w1.dir.length;
    const y2 = y1 + w1.dir.y * w1.dir.length;
    const x3 = w2.pos.x;
    const y3 = w2.pos.y;
    const x4 = x3 + w2.dir.x;
    const y4 = y3 + w2.dir.y;
    const den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
    if (den == 0) return false;
// Si le dénominateur est égal à 0, le calcul engendrerait une erreur
// donc l'intersection n'existe forcément pas
    const t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / den;
    const u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / den;

// Si l'intersection existe, on retourne vrai ou faux.
    return u >= 0 && u <= 1 && t >= 0 && t <= 1;
}
```

[8] Line-line intersection. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 6 octobre 2021 à 21:04. [Consulté le 9 octobre 2021]. Disponible à l'adresse: https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection

```
function intersection(w1, w2) {
// On n'effectuera aucun calcul si aucune intersection n'existe!
    if (!isIntersection(w1, w2)) return;

    const x1 = w1.pos.x;
    const y1 = w1.pos.y;
    const x2 = x1 + w1.dir.x * w1.dir.length;
    const y2 = y1 + w1.dir.y * w1.dir.length;
    const x3 = w2.pos.x;
    const y3 = w2.pos.y;
    const x4 = x3 + w2.dir.x;
    const y4 = y3 + w2.dir.y;

    const den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
    const u = ((x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)) / den;

    const xint = x3 + u * (x4 - x3);
    const yint = y3 + u * (y4 - y3);
    const intersection = {
        'x': xint,
        'y': yint,
            // Formule pour calculer la distance entre deux points
        'dist': Math.sqrt((y1 - yint) ** 2 + (x1 - xint) ** 2)
    };
    return intersection;
}
```

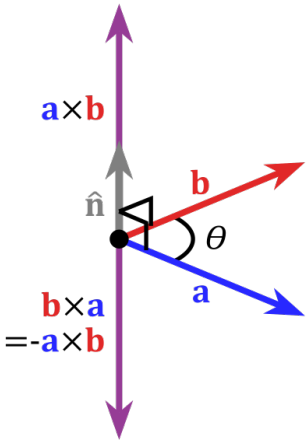
7.2 b) Find if two vectors are clockwise

A function used a lot in my program is called *isClockwiseOrder()*. This function returns true if the first inserted vector is oriented in such a way that it comes before the second vector in a clockwise direction. This allows me to compare the orientation of two vectors without worrying about their relative angle. The operation behind this function is the vector product.

As we are on a 2D plane we will apply the vector product $a \times b$ with $a_z = 0$ and $b_z = 0$:

$$a \times b = \begin{pmatrix} a_x \\ a_y \\ 0 \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ 0 \end{pmatrix} = \begin{pmatrix} a_y \cdot 0 - 0 \cdot b_y \\ 0 \cdot b_x - a_x \cdot 0 \\ a_x \cdot b_y - a_y \cdot b_x \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ a_x \cdot b_y - a_y \cdot b_x \end{pmatrix}$$

Ceci simplifie les calculs à effectuer et réduit le résultat à une valeur seulement. Cette valeur représente un certain vecteur perpendiculaire au plan 2D dont la hauteur est déterminée par le calcul. Ainsi, par la règle d'anticommutativité du produit vectoriel, (c'est-a-dire que $a \times b = -b \times a$), on peut facilement déterminer si deux vecteurs sont orientés dans le sens des aiguilles d'une montre.



The anticommutativity of the cross product[9]

In conclusion, if the result of $a_x \cdot b_x - a_y \cdot b_x$ is positive, then a and b are clockwise. If it is negative, then a and b are counterclockwise. If it is 0, then the vectors are parallel. In my code, I consider this case as equivalent to the first one by simplification.

```
function isClockwiseOrder(v1, v2) {
    return v1.x*v2.y - v1.y*v2.x <= 0;
}
```

[9]Représentation de l'anticommutativité du produit vectoriel, (auteur inconnu), 2008, Disponible à l'adresse: https://commons.wikimedia.org/wiki/File:Cross_product_vector.svg

8 Bibliographie

8.1 Images

- Page de titre : Vue d'obstacles multicolores en perspective ; Image générée à partir de mon code, HAMELINK Marcus 2021
- The cross product with respect to a right-handed coordinate system (auteur inconnu), 2008, Disponible à l'adresse : https://en.wikipedia.org/wiki/Cross_product#/media/File:Cross_product_vector.svg
- Représentation schématique des valeurs ; Schéma que j'ai dessiné, HAMELINK Marcus 2021
- Schémas pour illustrer la comparaison entre deux obstacles ; Schémas que j'ai dessinés, HAMELINK Marcus 2021

8.2 Sources

- Raycasting. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 22 juillet 2021 à 22:17. [Consulté le 10 octobre 2021]. Disponible à l'adresse : https://fr.wikipedia.org/wiki/Raycasting
- SHIFFMAN Daniel, 2019. Coding Challenge #146 : Rendering Raycasting [enregistrement vidéo]. Youtube [en ligne]. Disponible à l'adresse : https://youtu.be/vYgIKn7iDH8
- Line-line intersection. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 6 octobre 2021 à 21:04. [Consulté le 14 octobre 2021]. Disponible à l'adresse : https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection
- VANDEVENNE Lode, 2004-2021. Lode's Computer Graphics Tutorial : Raycasting [en ligne]. Dernière modification de la page en 2020. [Consulté le 15 janvier 2021] Lode Vandevenne, 2004-2020. Disponible à l'adresse : https://lodev.org/cgtutor/raycasting.html