UNIVERSITATEA “ALEXANDRU IOAN CUZA” IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

propusă de

Drăghici Constantin-Dănuț

Sesiunea: februarie, 2022

UNIVERSITATEA “ALEXANDRU IOAN CUZA” IAȘI

FACULTATEA DE INFORMATICA

3D rendering in real time with limited resources

Drăghici Constantin-Dănuț

Abstract

Although in today's age everyone has in their pocket a device that can easily render complex 3d scenes in real time, things were different at the beginning of the 1990s. Computers had limited graphics capability and for the vast majority of computer programs, user interaction was made using simple 2d interfaces and text. Despite all that, in 1992 Id Software launched the first 3d video game that acquired mainstream success, Wolfenstein 3D, featuring impressive graphics for the hardware it was running on. The goal of this project is to create a demo application implementing a different real-time rendering method showcasing a variety of additional features.

Sesiunea: februarie, 2022

Contents

Chapter 1
Introduction

1.1 Context

Real-time 3d rendering is part of the field of computer graphics and deals with creating 2d images starting from a scene in real time. The goal of computer graphics is the generation of images, which are also called frames, and one of the characteristics of computer graphics is the number of frames generated every second. Real-time graphics differ from traditional computer graphics when it comes to the number of frames generated, which is often over 30 frames every second.

Traditional computer graphics usually use ray tracing, a process where a large number of rays are cast from the virtual camera towards the 3d scene, thus generating a very detailed image. This process is very time consuming and the rendering of a single image can take anywhere from a few hours to a few days.

Real-time graphics must render a frame in less than 1/30 of a second and therefore the ray tracing process is too slow. The process used in real-time graphics rasterization, a process where every object from the scene is split in simple shapes, usually triangles or squares. Every shape is positioned, rotated and scaled on the screen and special hardware (GPU) is used to generate the pixels inside the shapes. This GPU can work with a very large number of shapes and is capable of generating complex effects like shadows, motion blur and reflections.

1.2 Motivation

One of the most prominent domains in which real-time 3d rendering is used is the video game industry, where the images on the screen must react in real-time to the user's inputs.

Most personal computers were capable of generating simple 2d images like lines and polygons in real time. However, real-time 3d graphics proved to be a very difficult task for traditional computers based on the Von Neumann architecture. In spite of these limitations, in 1992 Id Software launched a game called "Wolfenstein 3D" that had an interactive 3d world.

Today most games have at their core a game engine, like Unity, Unreal engine 4 and many others, that does all the work related to rendering the image. A game engine has the advantage of making the game development faster but often requires dedicated hardware. The goal of this project is to develop a real-time 3d engine that can run without dedicated hardware and using little resources. Likewise, the problems encountered during the development of this engine and the implemented solutions will be highlighted and the methods used to create the image will be compared with the ones used in the game Wolfenstein 3D.

1.3 Contributions

The list of own contributions that the paper contains are:

1. The storage method of the 3d scenes – Chapter 4
2. The creation of the real-time 3d rendering process – Chapter 4
3. The creation of the lighting effect – Chapter 4
4. The creation of a demo game that uses this rendering process – Chapter 4
5. Advantages and disadvantages compared to the Wolfenstein 3d engine– Chapter 5

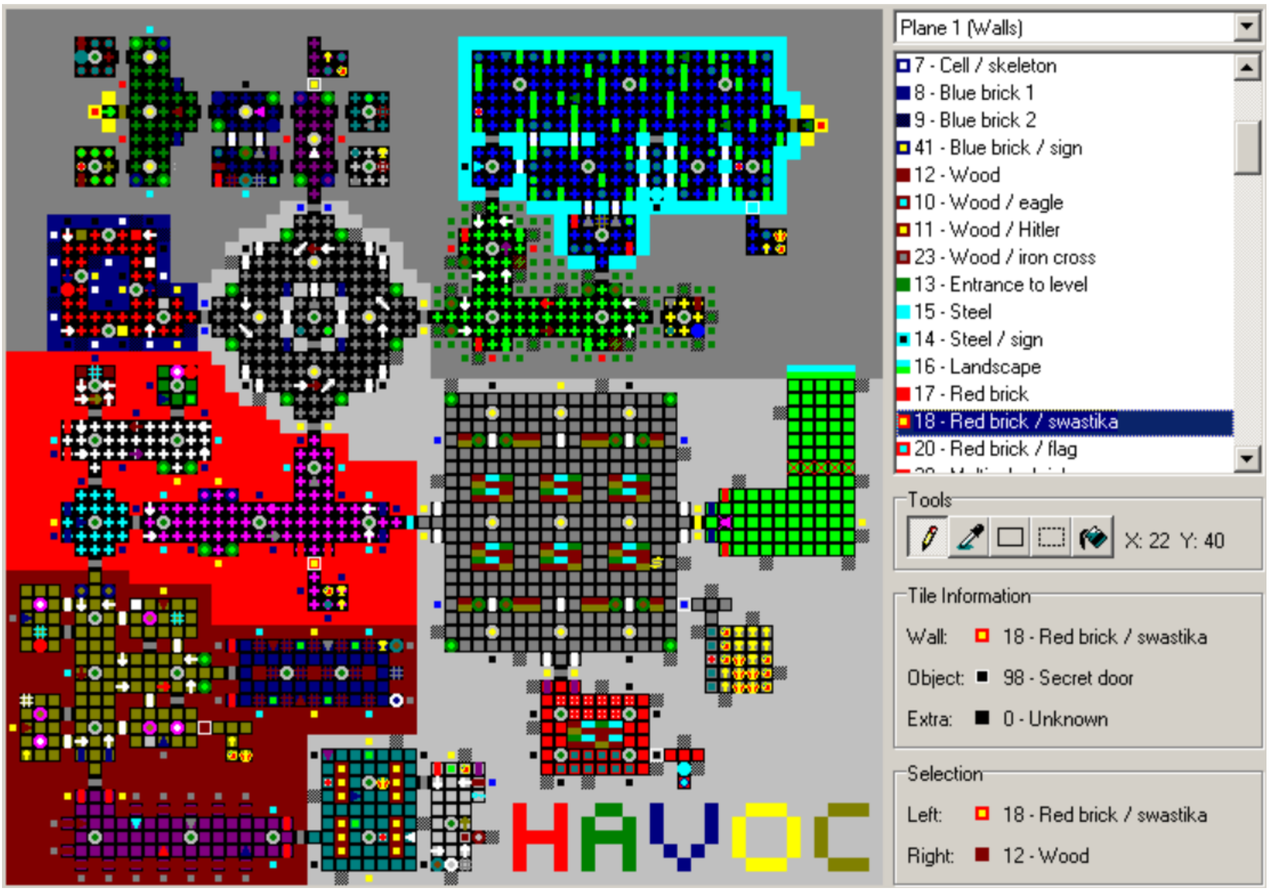
Chapter 2

Similar contributions

2.1 Ray casting in Wolfenstein 3D

2.1.1 Representation of the scene in memory

In Wolfenstein 3D the user can explore a world from a 3d perspective but in reality, this world is a 2d scene represented in memory using a bidimensional matrix that stores whole numbers. Each position in the matrix represents an object inside the scene and the type of the object is encoded using the value of the number stored at that position (E.g.: 0 - Empty, 8 – Blue rock, 12 – Wood, 15 – Steel, etc.).



Tile Information

Wall: 18 - Red brick / swastika

Object: 98 - Secret door

Extra: 0 - Unknown

Selection

Left: 18 - Red brick / swastika

Right: 12 - Wood

Figure 2.1.1.1: A map from Wolfenstein 3d opened in an editor

This scene representation is very efficient when it comes to the storage space used but at the same time it comes with a set of problems. First of all, because the scene is stored in memory in a 2d matrix there can't exist more objects at the same position. This means that there can't exist different areas on top of each other, like bridges or buildings with multiple floors. Furthermore, all the objects from the scene have a square shape, which makes it impossible for a smooth diagonal structure to exist.

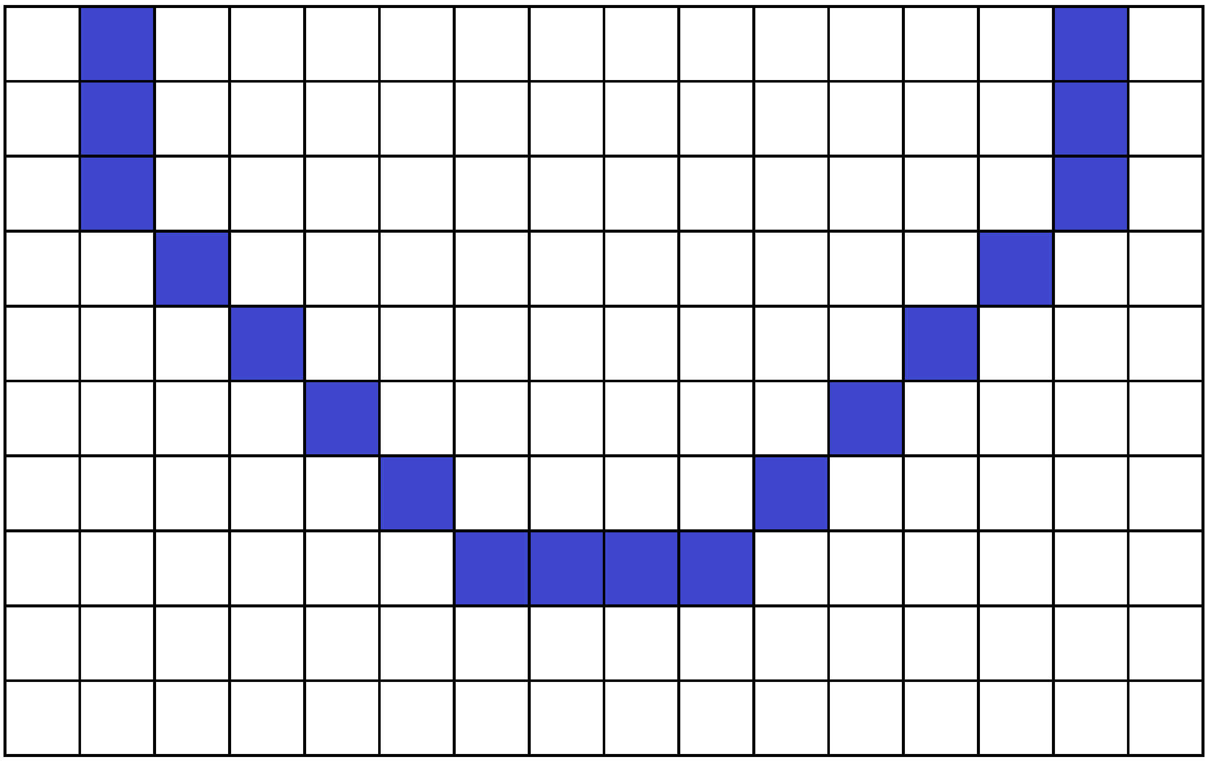


Figure 2.1.1.2: Diagonal structure with jagged edges

2.1.2 Image rendering

Ray casting is a technique that creates a 3d perspective starting from a 2d map. Because the scene is 2d, the field of view will take the shape of a triangle instead of a pyramid.

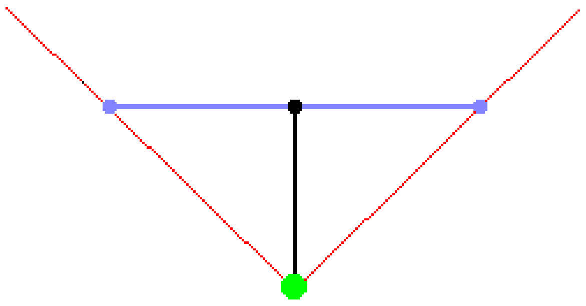
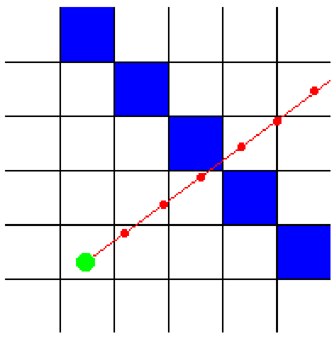


Figure 2.1.2.1: 2d representation of the field of view

According to Lode Vandevenne in Lode's Computer Graphics Tutorial, in order to build the image, the field of view is split in a large number of parts of equal size, each of them representing a vertical line of the final image. A ray is cast from the position of the virtual camera towards the middle of each of these parts that stops the moment it collides with an object.

To check if the ray intersects with an object a point is chosen on the ray at a fixed interval and check the value of the square that contains that point. If the value is equal to 0 then that square is empty and we chose a new point to check. If the value is not 0 then the ray collides with an object and we stop checking for new points.

Choosing the interval isn't a trivial matter, the bigger the interval, the more likely it is to miss an object, and the smaller the interval, the slower it will find a collision. As we can see in the figure 2.1.2.2, by using a big interval when choosing our points, the object was missed. In the figure 2.1.2.3 the interval used is smaller and therefore the collision was successfully detected. However, no matter how small the interval is, there will always be a possibility that a collision will be missed.



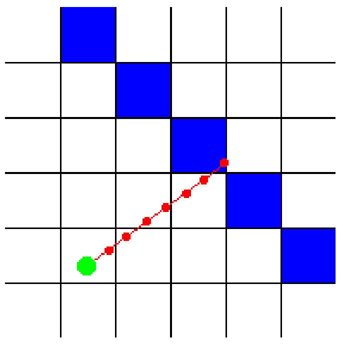


Figure 2.1.2.2: Check with a big interval

Figure 2.1.2.3: Check with a small interval

However, thanks to the 2d matrix representation of the scene, a more efficient method can be used. Because all the objects are arranged on a grid it is possible to check only the points where the ray intersects the grid, in other words the points that have an integer as their x or y coordinate (Fig. 2.1.2.4). Using this method, the point where the ray collides with an object can be found with the minimum number of steps.

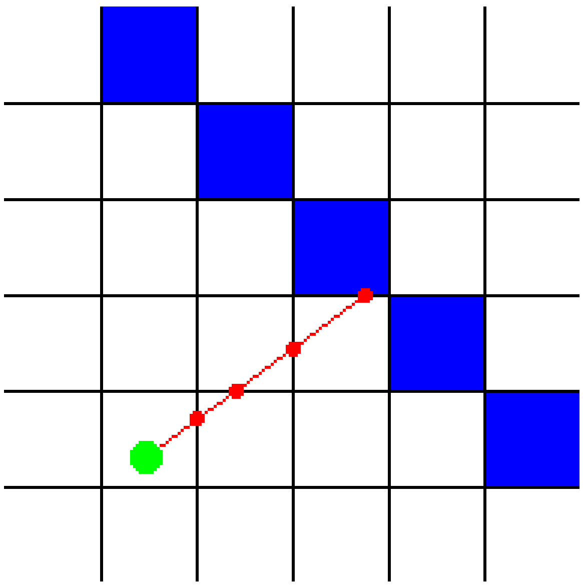


Figure 2.1.2.4: Checking points on grid lines

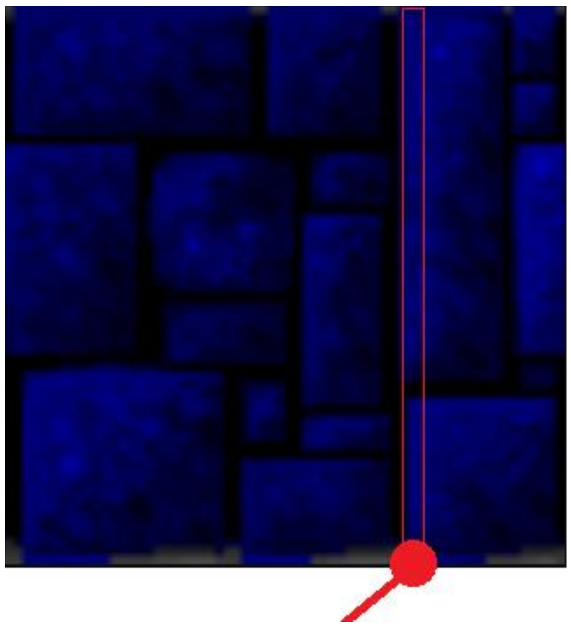


Figure 2.1.2.5: Selected column

The object's type is determined by checking the value of the number saved at the object's location, each type having an associated texture saved in memory. The next step is determining what part of the texture needs to be drawn, in other words determining what part of the object the ray collided with. This is done using the fractional part of the non-integer coordinate of the collision point. In the previous figure the fractional part of the non-integer coordinate is 0.7 and by multiplying it with 64, which is the horizontal resolution of the textures, we get the number of the column of pixels that should be drawn on the screen. The next step is calculating the distance from the collision point to the virtual camera and to scale the column of pixels accordingly. This makes the objects further away to appear smaller and the objects closer to appear bigger.

To render the image, this column of pixels is drawn on screen at the position of its respective part of the field of view. By doing the same procedure for all the other parts of the field of view a final image is created.

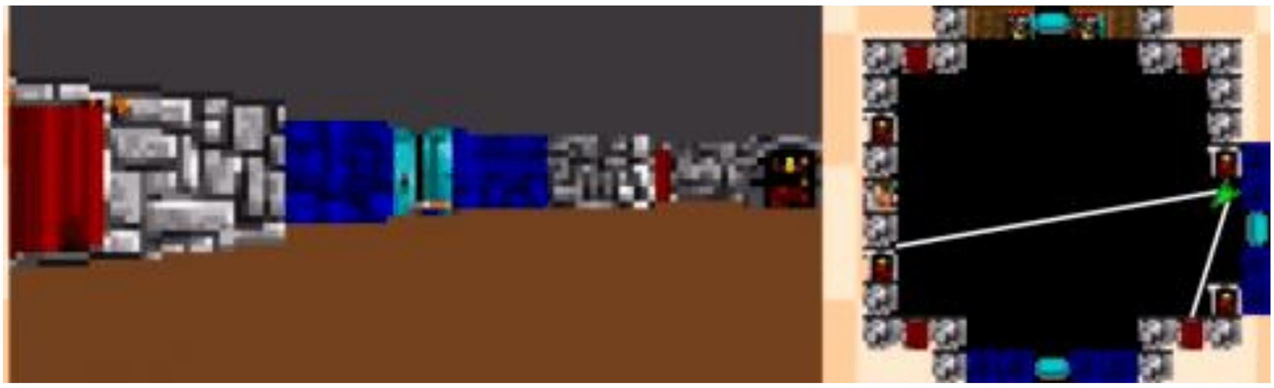


Figure 2.1.2.6: Image created using ray casting

2.1.3 Advantages and disadvantages

Advantages

- High efficiency – Due to the simple way the image is created, this process is quick and efficient.
- Scenes are easy to make – All it takes to create a scene is a bidimensional matrix of integers
- The scenes use very little storage space – The only thing stored about the objects inside the scene is an integer value that determines their type. There is no need to store the position of objects inside the scene because it is given by the position inside the matrix.

Disadvantages

- The walls can only be at straight angles – This is due to the way the objects are stored in memory. Because all the objects must fit inside a grid, the available positions they can be in are limited
- The objects all have the same height
- It is impossible for a scene to contain 2 areas on top of each other – This is a consequence of the way the scenes are stored and the way the image is drawn.

2.2 Id Tech 1

Id Tech 1 is the engine used by the game Doom, made by the same company that also made the engine for Wolfenstein 3D. Although this engine works with the same hardware limitations as the previous engine, it solved some of the problems present before.

According to Fabiem Sanglard in Game engine black book: Doom, this new engine has the capability to work with height differences, stairs, jumps and lights. That being said, one of the biggest limitations of the old engine still remains. Because the scenes are actually 2d, the scene cannot contain areas stacked on top of each other, making structures like multiple floors building impossible.

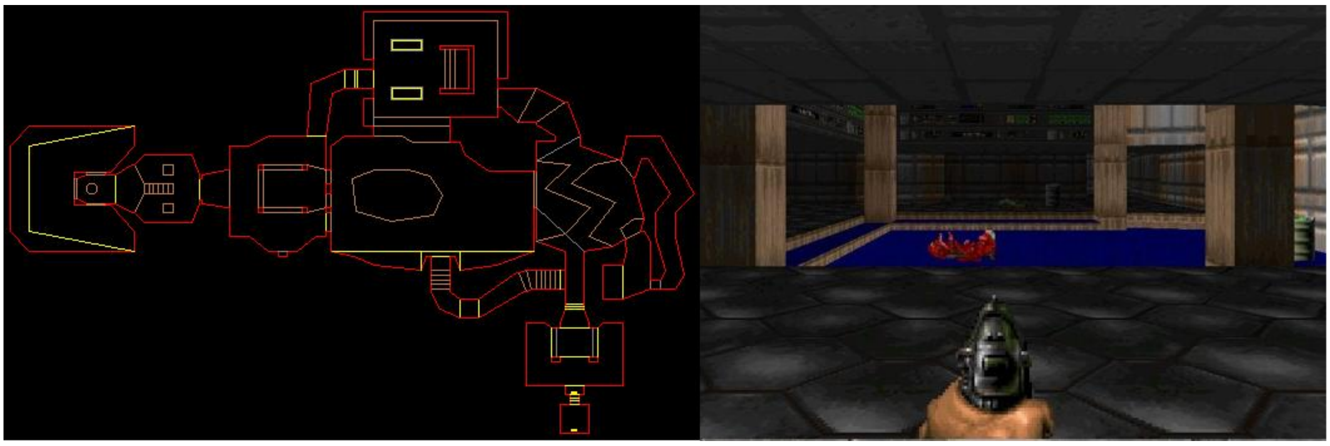


Figure 2.2: First map from Doom and its 2d representation in memory

Id Tech 1 creates the image by drawing simple polygons on the screen, in contrast with Wolfenstein 3d which creates the image using vertical lines of pixels. This allows for the objects to be drawn at any coordinates on the screen, therefore creating the illusion of different elevation levels. As we can see in figure 2.2, each object is represented in memory by a straight-line segment, making it possible to create more advanced scenes. Similar methods will be implemented in the creation of the demo application.

2.3 Conclusions

While Doom is a big advancement in real-time rendering when compared to Wolfenstein 3d, it still has its fair share of limitations. The image is rendered with a maximum of 256 colors, which was increased to over 16 million colors in the demo application.

Likewise, the engine could render with a maximum framerate of 35 frames/second and the logic was tied to the framerate. Therefore, in the case when the game was running on a computer that could not render all the 35 frames fast enough, the animation and the logic would happen in slow-motion. In order to make the animations and the logic independent from the framerate in the demo application, when rendering a new frame, the new positions of the objects are computed using the time passed since the last drawn frame instead of a fixed amount of time. As a consequence, the demo application will run at the proper speed regardless of the framerate.

Another drawback of Doom is that it runs at a fixed resolution of 320x200, making it unsuitable for the screens in use today. The demo application fixes this problem by scaling the positions of the objects with the resolution of the application window before drawing the objects on the screen, therefore being able to run at any screen resolution.

Chapter 3
Used technologies

3.1 OpenGL[1]

OpenGL is an API that can run on a large variety of platforms and that is widely adopted for programming the 2d and 3d graphics components of computer programs. Some of the fields that use this API are computer assisted graphics (CAD), virtual reality, flight simulations and video games.

In contrast with other APIs in which the programmer only needs to describe the scene and let the API do the rest, OpenGL is a low-level API that requires the programmer to give exact instructions in order to render an image. The base function of OpenGL is to accept simple shapes like points, lines and polygons and to convert them into pixels. This requires that the programmer has a good understanding of the graphics pipeline but at the same time offers the freedom to implement new rendering algorithms.

By using a library like Mesa 3D, OpenGL is able to run using only the processor, without requiring dedicated hardware.

3.2 Gloss library

Gloss is a library for the functional programming language Haskell that has OpenGL at its core and that provides simple functions for the manipulation of graphics elements. The only graphical functions used in the developed rendering engine are the ones for drawing a line or a polygon.

[1] More information about the library OpenGL can be found at the following address: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf

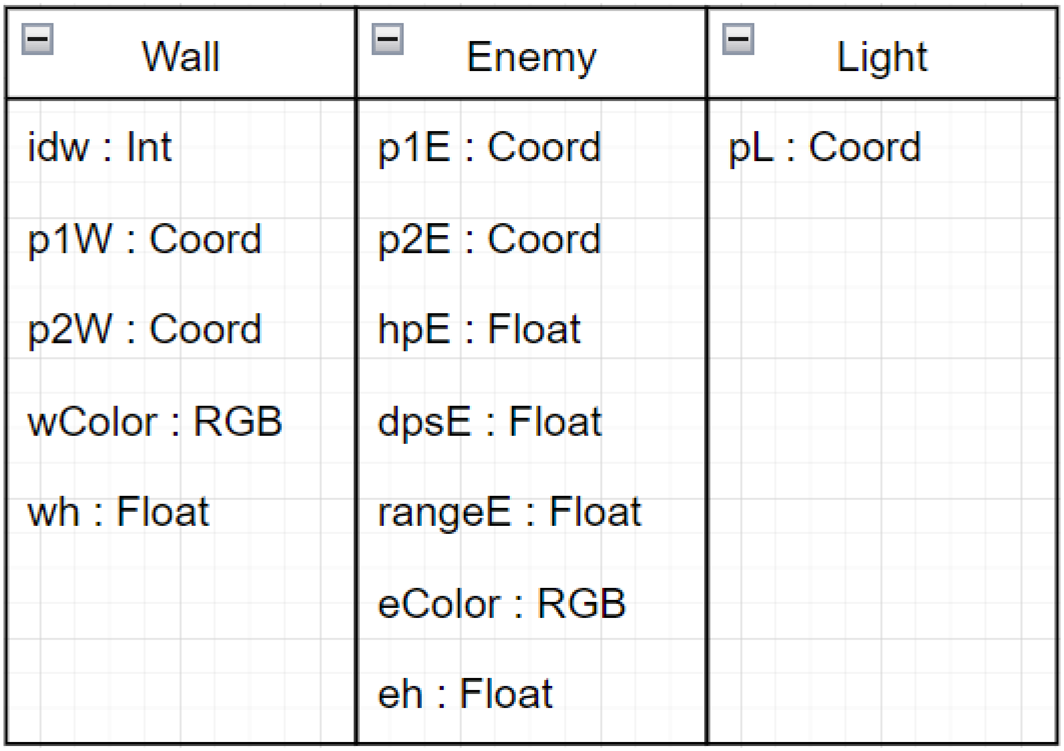
Chapter 4

Creating the demo application

4.1 Scene representation in memory

The scene is represented using 3 different lists, one for the walls, one for the enemies and one for the light sources.

Besides their position inside the scene, which is present in all the data types, the walls and enemies contain other additional data which makes possible the implementation of new features which will be discussed in the following chapters.



Wall	Enemy	Light
idw : Int	p1E : Coord	pL : Coord
p1W : Coord	p2E : Coord	
p2W : Coord	hpE : Float	
wColor : RGB	dpsE : Float	
wh : Float	rangeE : Float	
	eColor : RGB	
	eh : Float	

Figure 4.1.1: Diagram of the data structures

This way of representing a scene uses more storage space compared to the bidimensional matrix used by Wolfenstein 3D but it allows for more complex scenes and fixes most of its limitations.

One of the biggest advantages that this way of storing the scene offers is the possibility of having objects at any angle. This is because unlike the 2d matrix used by Wolfenstein 3D that limits the objects to a grid, this representation stores the coordinates of each individual object. Therefore, as we can see in figure 4.1.2, all the objects are smooth, unlike in the figure 2.1.1.2 where the diagonal objects were jagged.

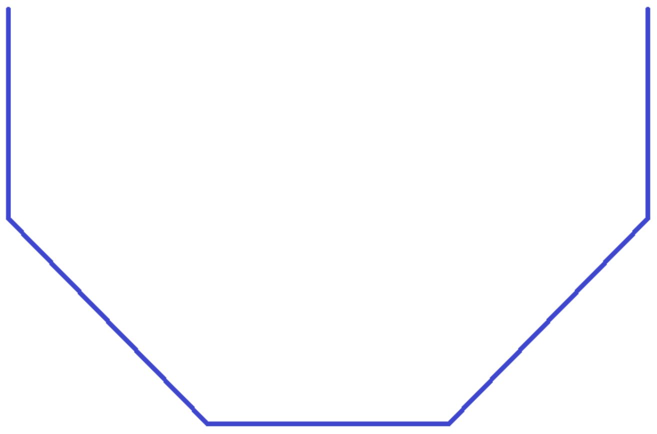


Figure 4.1.2: Smooth diagonal objects

Also, another big advantage of this representation is that the objects can gave any lengths, which offers a lot more flexibility when creating a scene. This is a consequence of the fact that each object's position is defined by a pair of points in the scene that give the object its start and end. A good example is the figure 4.1.3 where we can observe very long objects representing the exterior walls of a room and very short objects used to give the impression of thickness of the interior walls.

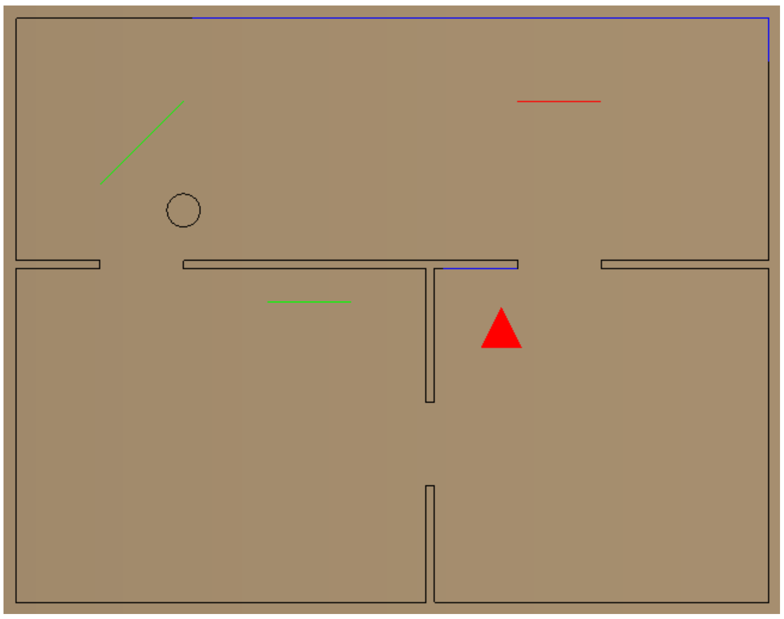
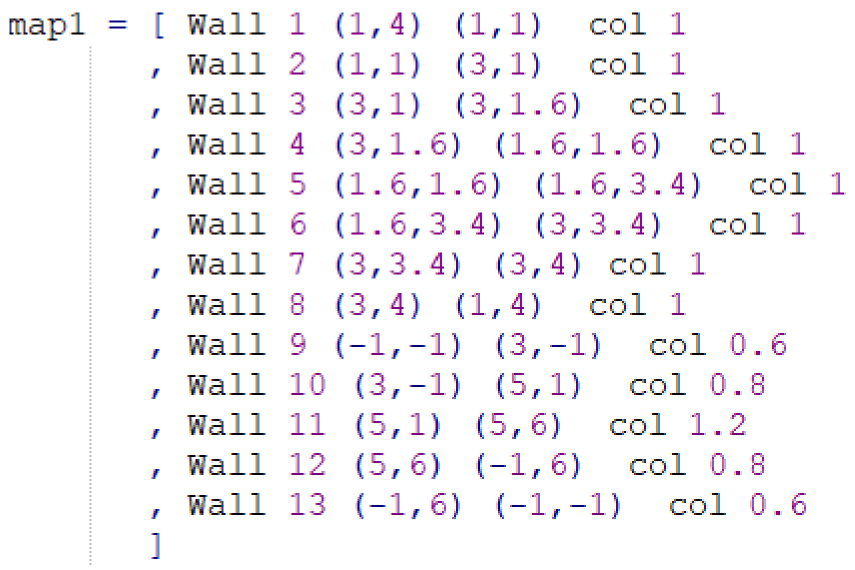


Figure 4.1.3: Scene of a house

4.2 Creating the scenes

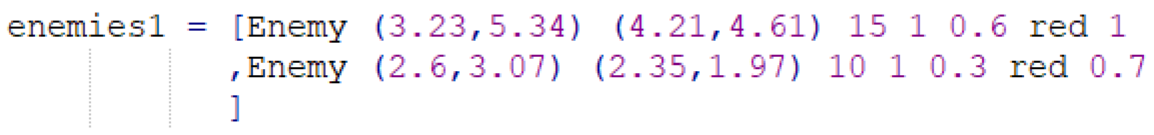
Each scene represents a game stage and contains 3 lists: one for the walls, one for the enemies and one for the lights. The list containing the walls is defined as follows:



```
map1 = [ Wall 1 (1,4) (1,1)  col 1
       , Wall 2 (1,1) (3,1)  col 1
       , Wall 3 (3,1) (3,1.6)  col 1
       , Wall 4 (3,1.6) (1.6,1.6)  col 1
       , Wall 5 (1.6,1.6) (1.6,3.4)  col 1
       , Wall 6 (1.6,3.4) (3,3.4)  col 1
       , Wall 7 (3,3.4) (3,4) col 1
       , Wall 8 (3,4) (1,4)  col 1
       , Wall 9 (-1,-1) (3,-1)  col 0.6
       , Wall 10 (3,-1) (5,1)  col 0.8
       , Wall 11 (5,1) (5,6)  col 1.2
       , Wall 12 (5,6) (-1,6)  col 0.8
       , Wall 13 (-1,6) (-1,-1)  col 0.6
       ]
```

Each wall has, in order, the following: an unique id, a pair of coordinates for the first point, a pair of coordinates for the second point, a color and a height. It should be noted that even if in the example above all the walls have the same color "col", each wall can have a different color.

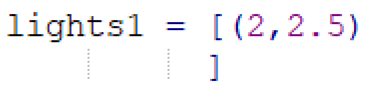
The list with the enemies is defined in a similar way:



```
enemies1 = [Enemy (3.23,5.34) (4.21,4.61) 15 1 0.6 red 1
           ,Enemy (2.6,3.07) (2.35,1.97) 10 1 0.3 red 0.7
           ]
```

Each enemy has, in order, the following: a pair of coordinates for the first point, a pair of coordinates for the second point, health points, attack points, the distance at which they can hurt the player, the color and the height.

The list containing the lights is simple compared to the others because the only information stored about the light is its position in the scene:



```
lights1 = [(2,2.5)
          ]
```

The other stages are created in a similar way.

4.3 Drawing the HUD

During the game it is important that the user has access to certain information like his health points, remaining ammo, the location of the middle of the field of view and a map of the scene.

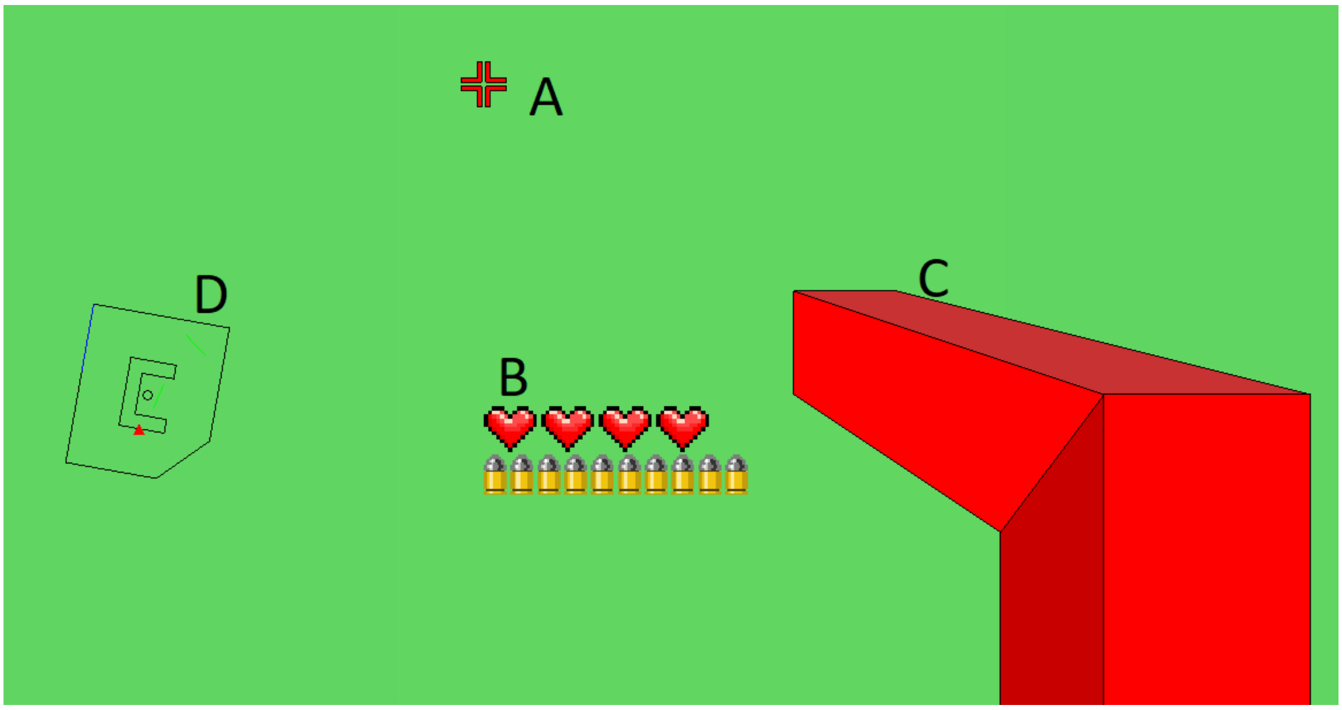


Figure 4.3.1

The middle of the field of view is highlighted by a red target in the middle of the screen (figure 4.3.1 – A)

The images of the ammo and health points are drawn using small square polygons and are drawn multiple times in order to show the current number of points (figure 4.3.1 – B).

A decorative gun drawn using 4 polygons (figure 4.3.1 – C)

The map of the scene must convey in an easy-to-understand manner the player's position, the direction the player is looking, the walls, the enemies and the light sources. In order to create the map for each object inside the scene a segment is drawn, using black for walls and green for enemies. This is also when the player and the lights are drawn, using a red triangle and black circles respectively (figure 4.3.1 – D).

4.4 Drawing the image

The field of view represents all the elements from the scene that can be visible on the screen at a given time. The field of view starts from the virtual camera and extends towards the scene in the shape of a pyramid.

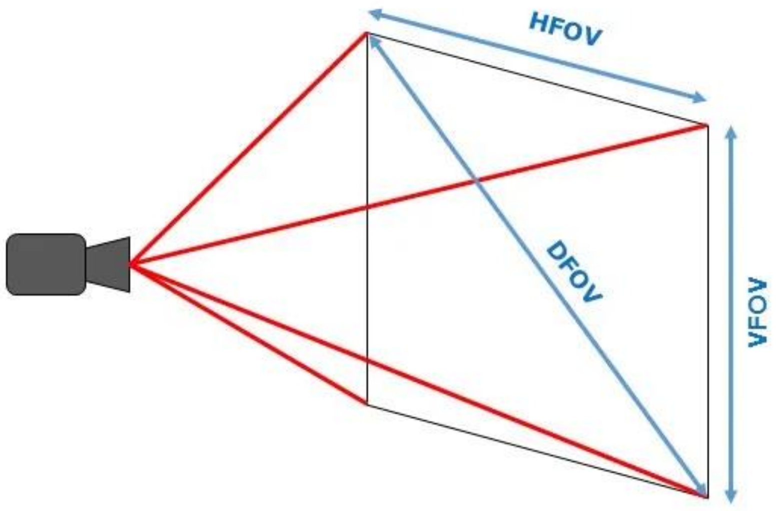


Figure 4.4.1: Field of view of a virtual camera

The position of the camera and the direction in which the field of view extends are controlled by the user, making possible the exploration of a scene in real time.

Since the position of an element in the scene is determined by a pair of points, in order to draw that element on the screen the point in the scene must be transformed into points on the screen.

The horizontal position of a point on the screen is calculated as the slope of the line from the virtual camera towards the point in the 2d plane of the scene. The virtual camera is situated at (0,0) and therefore the formula for the slope is simplified to just a simple division.

A ray is cast from the camera towards the point in the scene. The length of the projection of that ray onto the camera plane must be calculated in order to compute the y coordinate of that point on the screen. If the distance from the virtual camera towards the point is used instead, it will create a fish-eye effect. It should be noted that for each point in the 2d scene there must be two points on the screen at different vertical positions in order for the image to be 3d. The y coordinates of the two points depend on the height of the object and the length of the ray's projection. In order to draw the object on the screen an OpenGL function that draws a polygon is called with the computed points as parameters (fig 4.4.3).

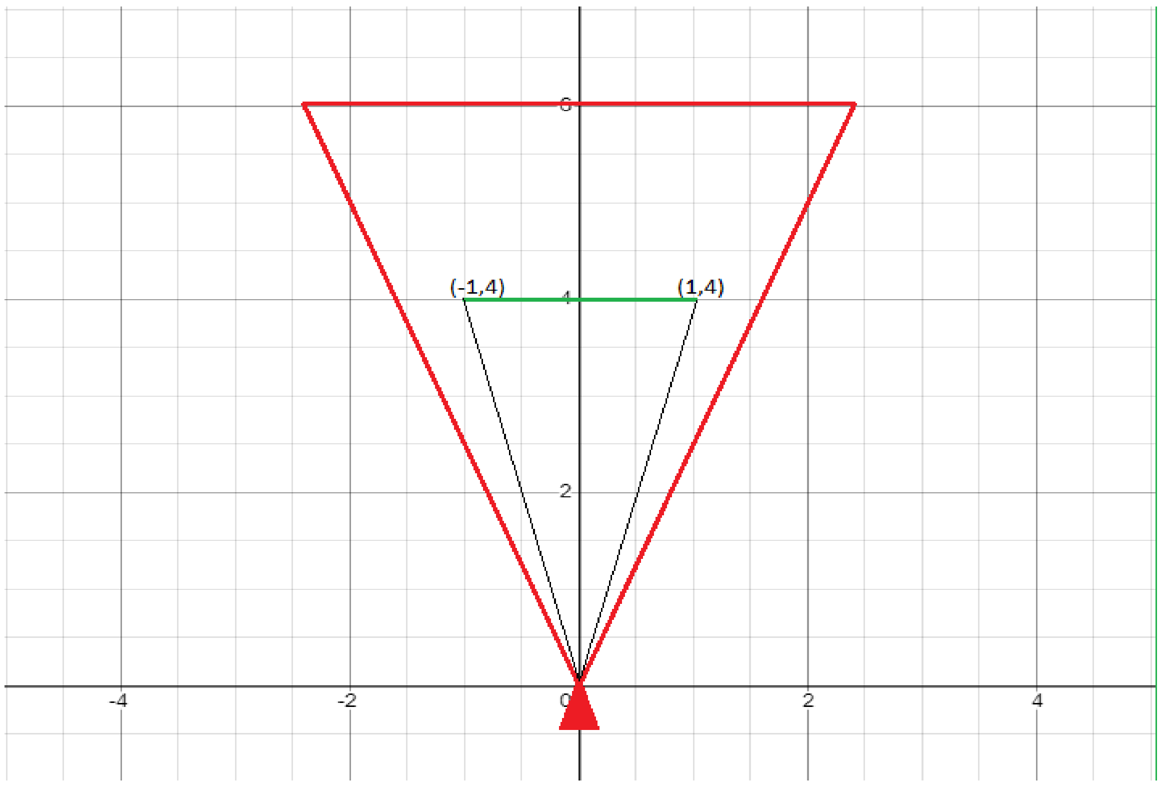


Figure 4.4.2: An object inside the field of view

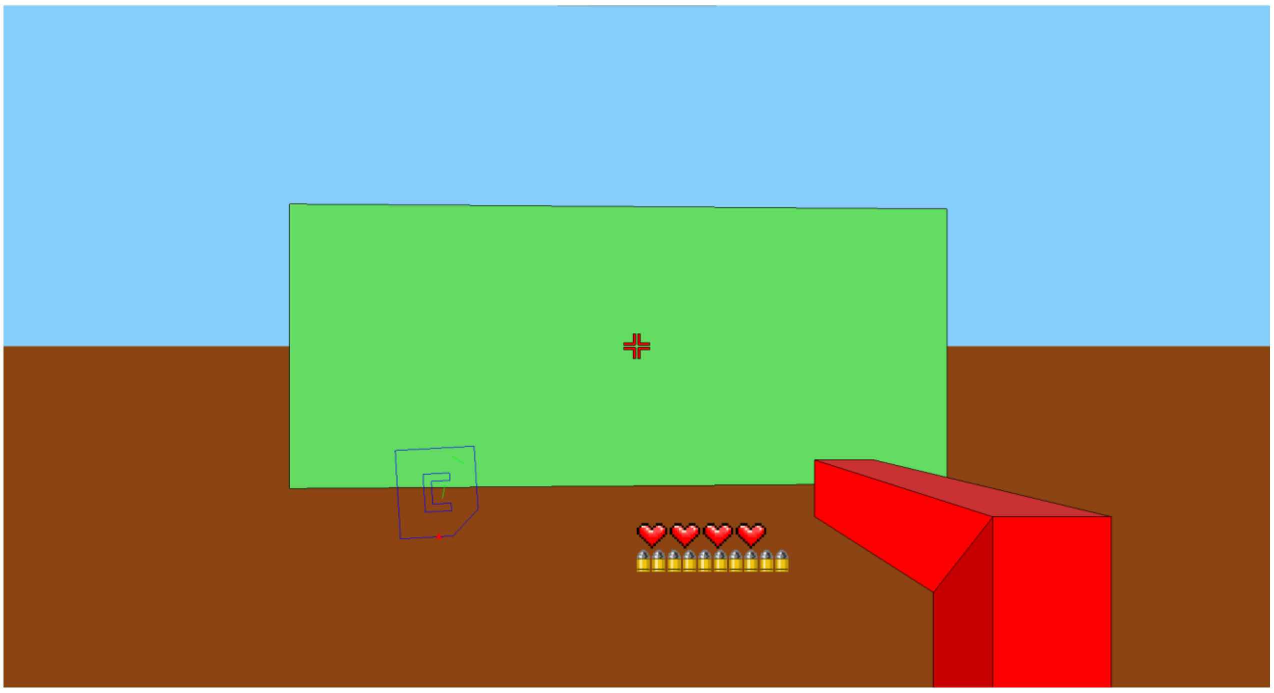


Figure 4.4.3: The image created on the screen

Because every object has a specific height that is taken into account when it is drawn, it is possible to create a scene with objects of different heights (fig 4.4.4) which is not possible in Wolfenstein 3D.

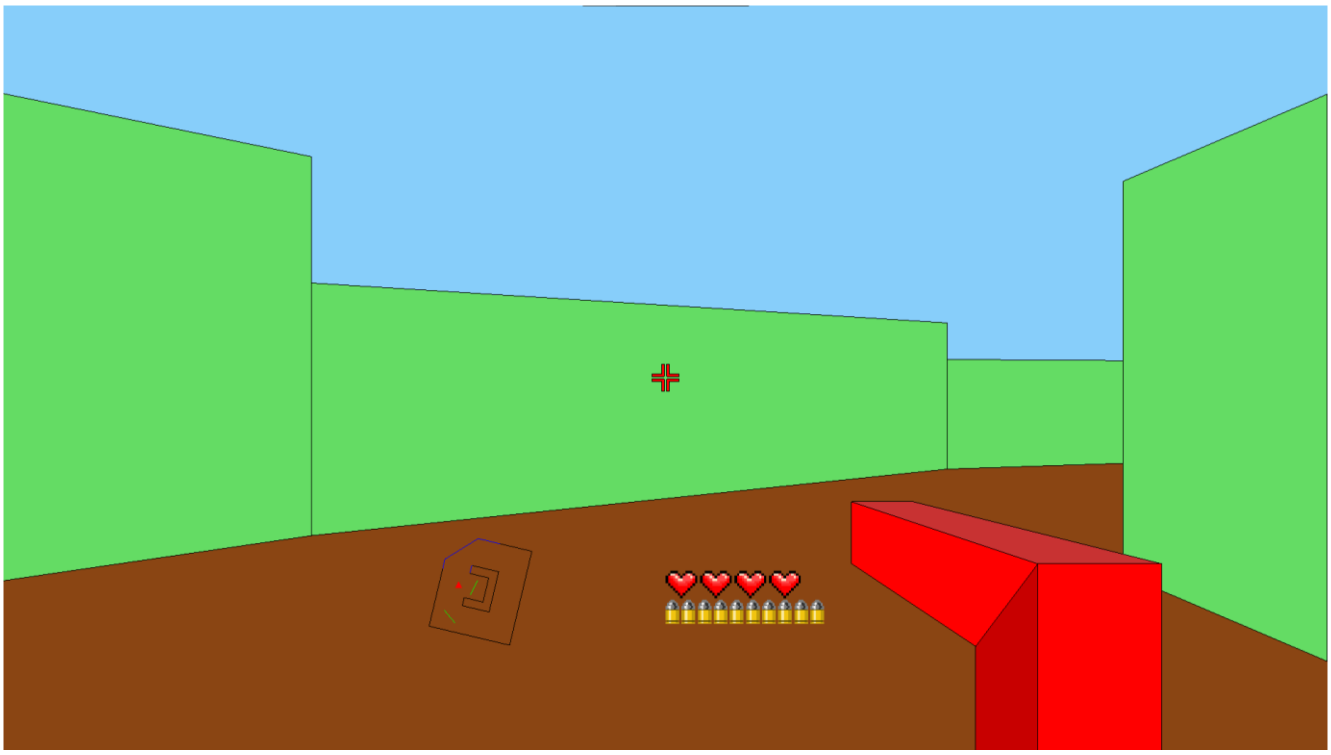


Figure 4.4.4: Walls of different heights

The next step in rendering a scene is drawing all the objects. However, if the objects aren't drawn in a particular order the process will result in an image that isn't properly rendered (fig 4.4.5).

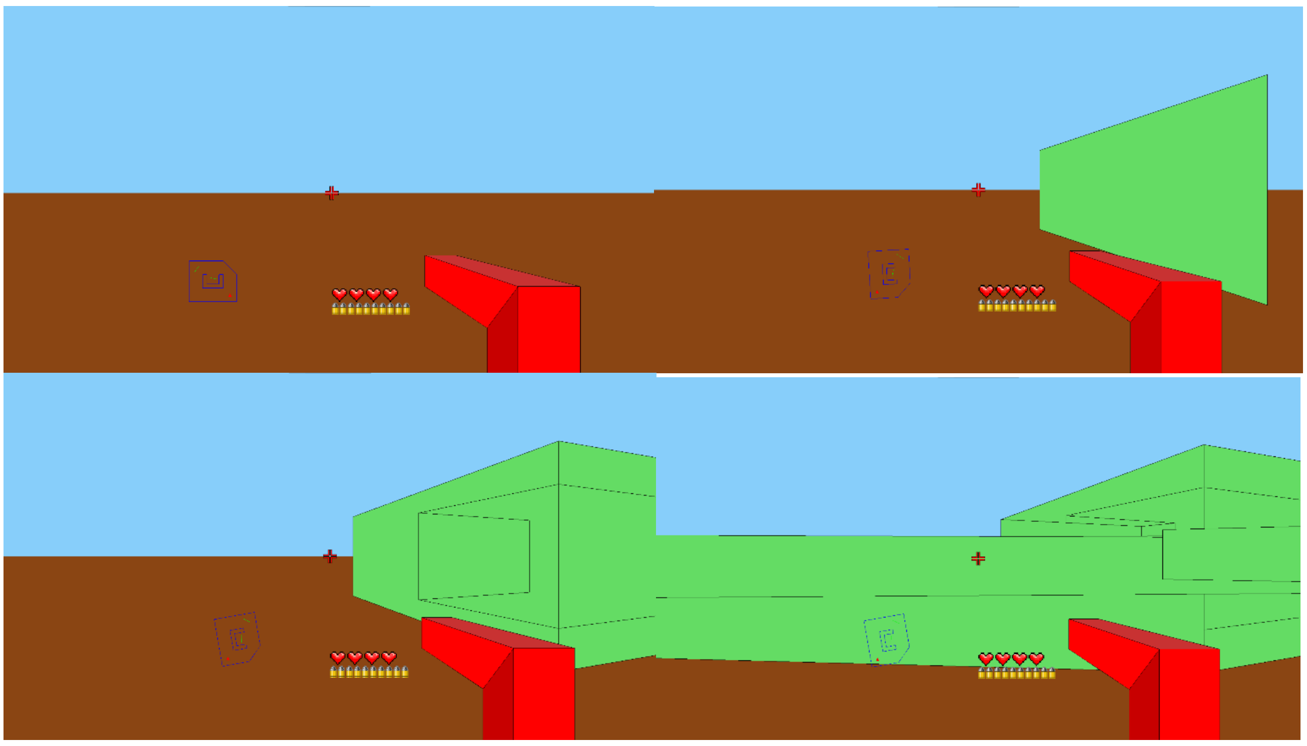


Figure 4.4.5: Render of all the elements in the list

As it can be seen in the previous figure, the order in which the objects are drawn is important. Without a proper order in which the objects are drawn the objects from further away could be drawn after the objects that are closer to the virtual camera.

In order to fix this problem, we need an algorithm that sorts all the objects from the scene based on the order in which they must be drawn so the image renders correctly.

A good start is calculating the distance from the virtual camera to every object from the scene. Since the objects are represented as a segment, the formula for the distance from a point to a line segment given by 2 points can be used:

$$\operatorname{distance}(P_1, P_2, (x_0, y_0)) = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}.$$

The objects from the scene will be ordered based on the distance to the virtual camera. However, there are situations in which a more distant object must be drawn before a closer object (figure 4.4.6).

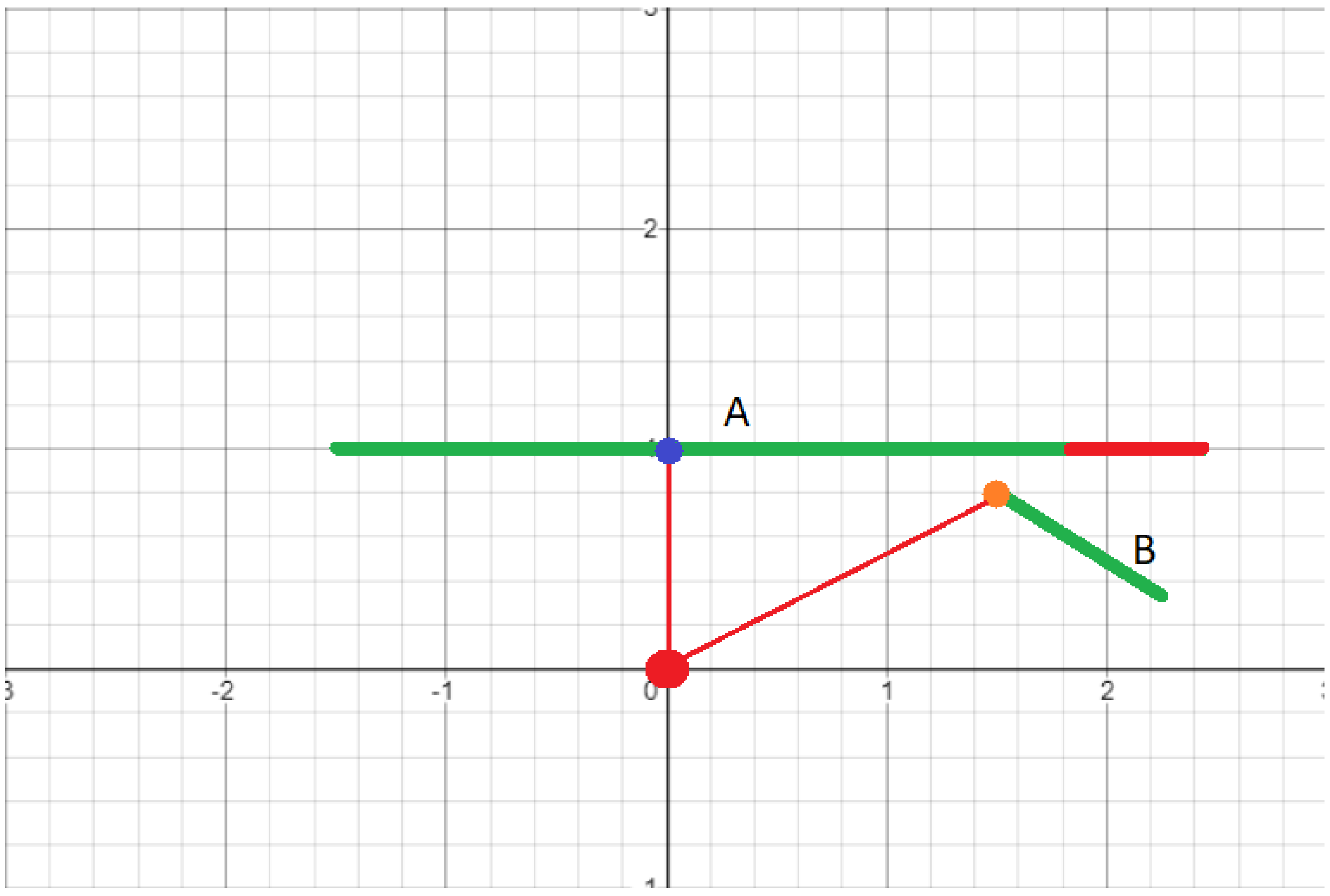


Figure 4.4.6

As we can see in the previous figure, the segment A is partially covered up by the segment B, meaning that the segment B must be drawn after drawing the segment A. Therefore, a simple sort based on distance is not sufficient for a correct rendering. For the image to render properly, the objects that are covered up by other objects must be drawn before the segments that cover them up are drawn.

The first step in checking if an object is covered up is picking a number of points along the object, preferable at small and equal intervals.

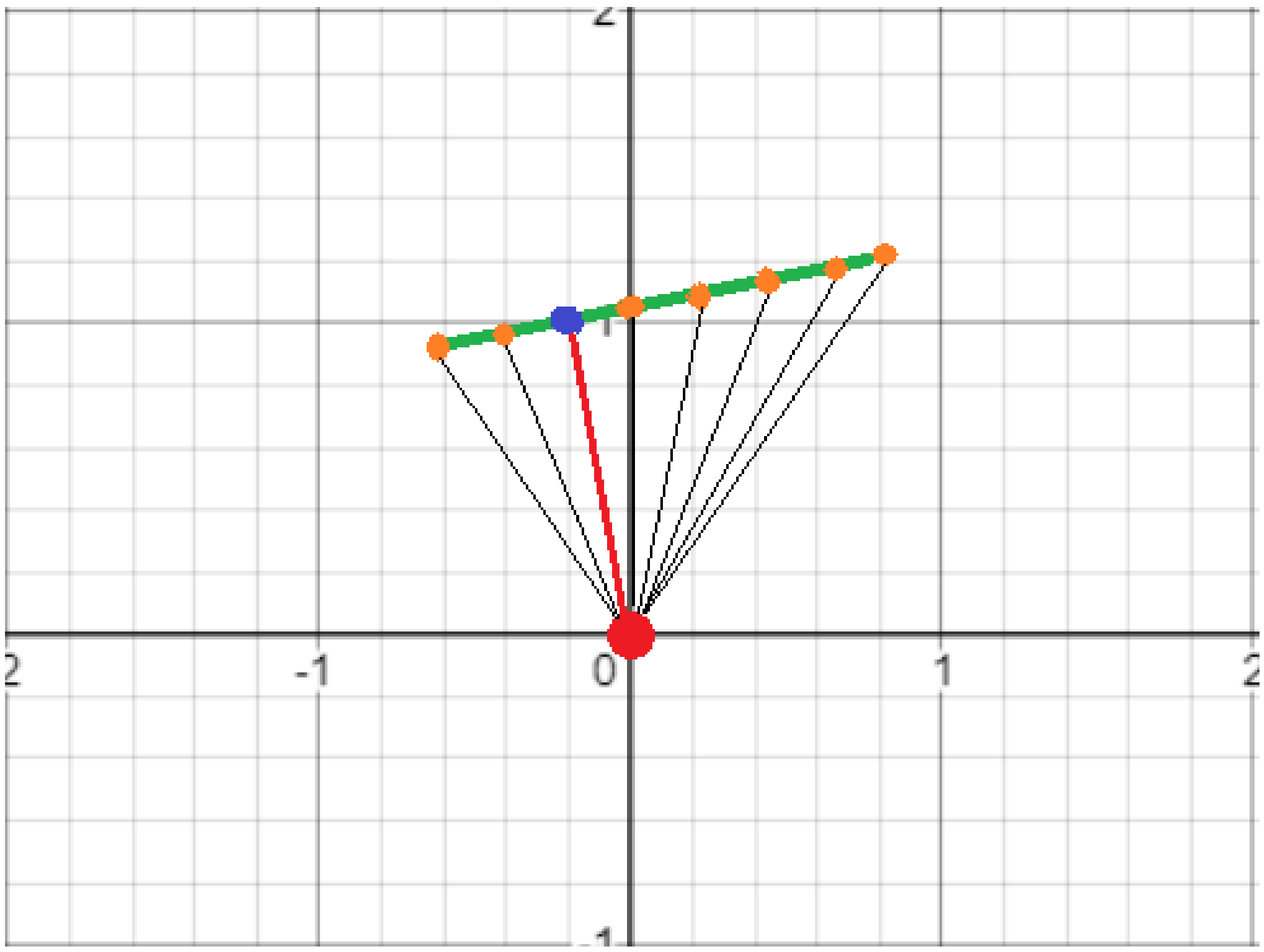


Figure 4.4.7: Points chosen along an object

For each of the chosen points a line segment is created from it and the virtual camera. If this line segment intersects any other object in the scene, then it means that the point is covered up.

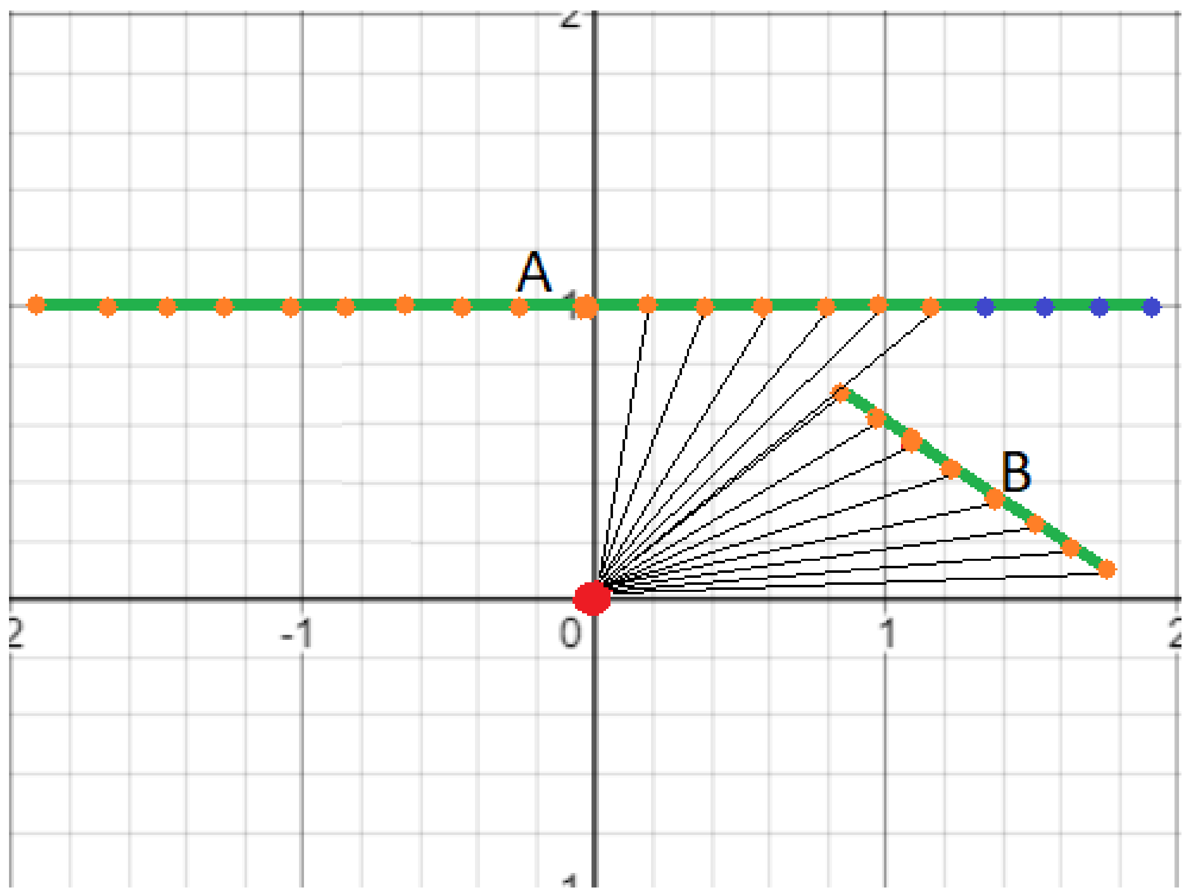


Figure 4.4.8

As we can see in figure 4.4.8, the line segment A contains points that are covered up by the line segment B, and the line segment B doesn't have any points covered up. Therefore, line segment A should be drawn before line segment B.

We create a list where we will store the objects from the scene in the order that they should be drawn. The objects that are not covered up by any other object are added to the list, in ascending order based on the distance. After all the objects that are completely visible are added to the list we will check again if the objects from the scene are completely visible, without taking into consideration the objects that are already part of the list. This step is repeated until all the objects from the scene are inside the list. This ensures that all the objects covered up by an object are drawn before the object covering them.

In the above figure, object A is covered up by object B, meaning that object B will be added to the list. Since not all the objects from the scene are inside the list, the check will repeat, without taking into consideration the object B. Without the object B, the object A will be completely visible, so it will be added to the list.

In order to render the final image, we simply draw all the objects in the reverse order they were added to the list. The end result is an image correctly rendered (figure 4.4.9).

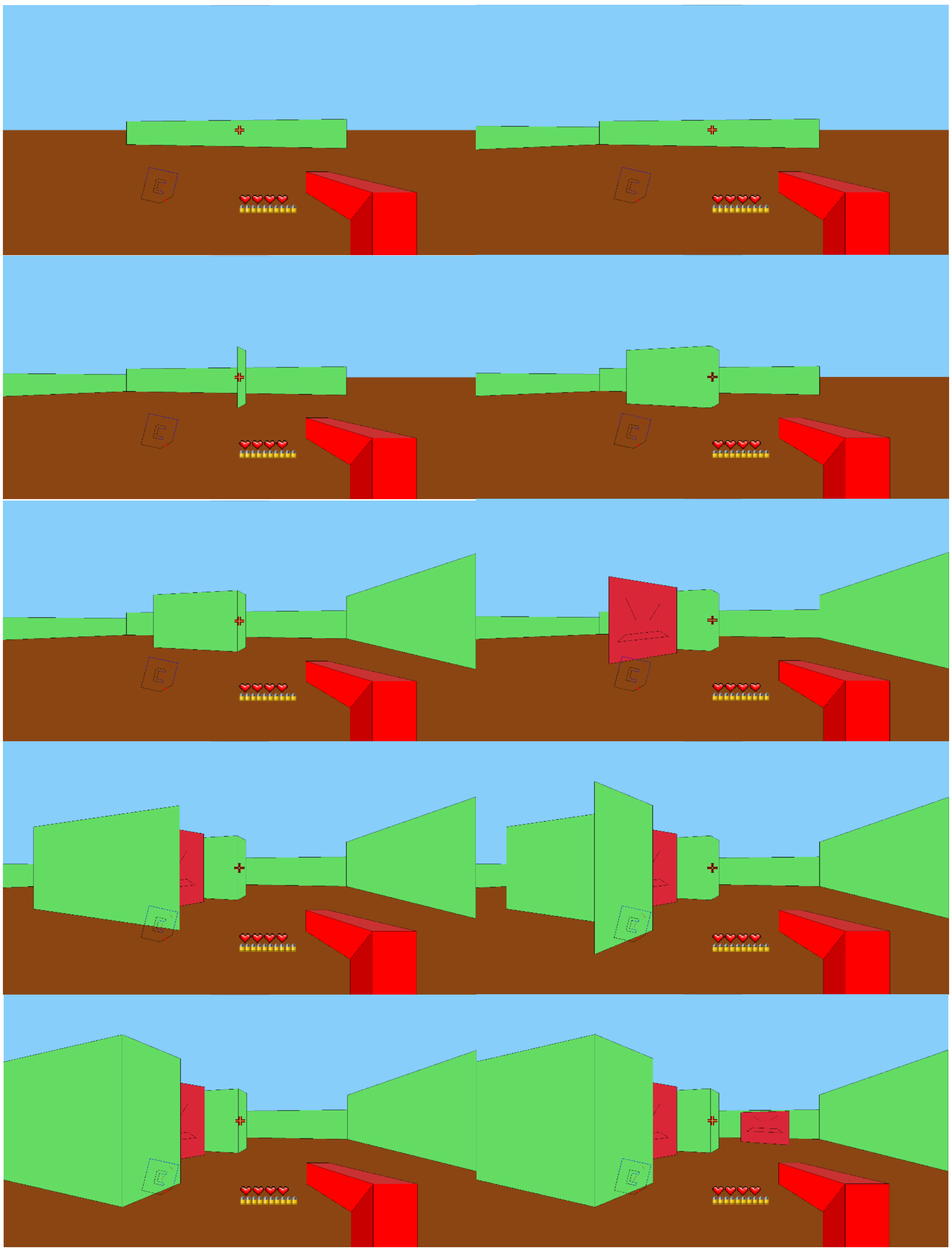


Figure 4.4.9: The complete rendering of an image

Even if the final image is correct, this rendering method is inefficient because it isn't aware if an object will be visible or not in the final image. Therefore, it can waste time processing the objects that are outside the field of view or that are completely covered by other objects.

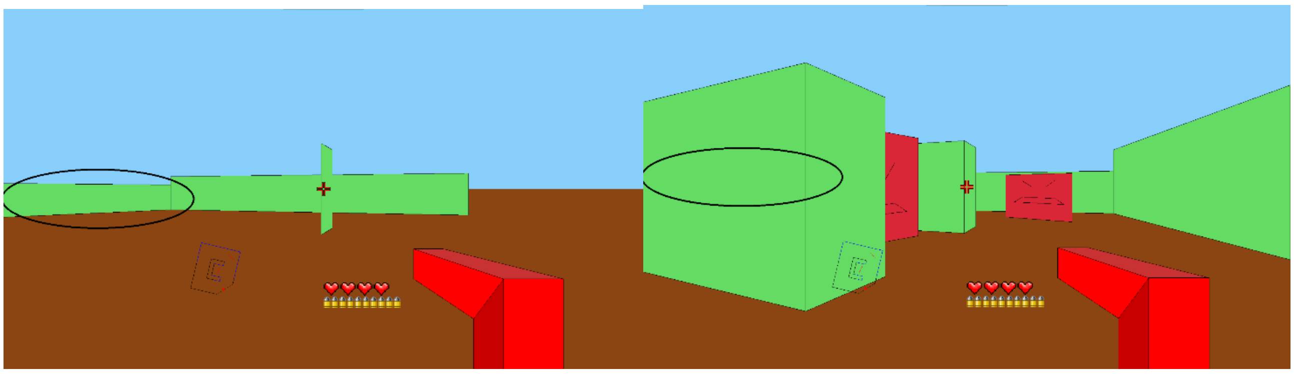


Figure 4.4.10: A wall that is not visible in the final image

As we can see in figure 4.4.10, the circled wall is drawn on the screen but is later completely covered by another wall. In order to avoid unnecessarily wasting resources, it is required that the objects that will not be visible in the final image are eliminated.

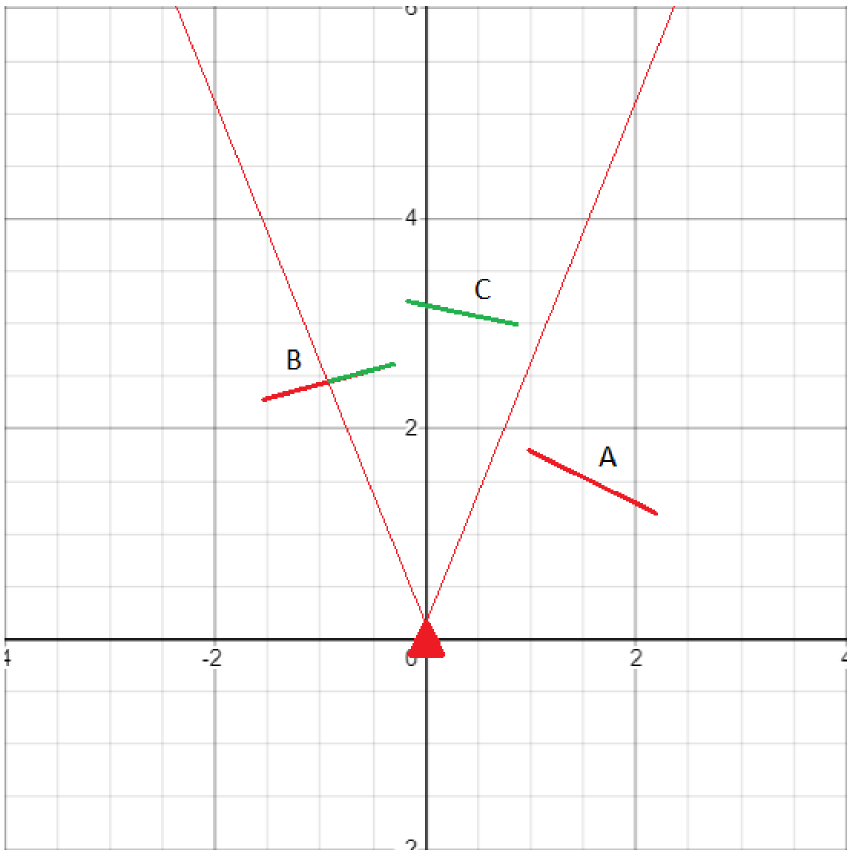


Figure 4.4.11: Filtering of objects outside the field of view

Firstly, the objects that are outside the field of view will be filtered out. Each object can either be completely inside, completely outside or partially outside of the field of view. Each of these kinds of objects will be processed in a different way. The distinction between these three different kinds of objects is done like in the following way:

- If the points that define the object are not inside the field of view and the object doesn't intersect with the edges of the field of view then the object is outside the field of view (object A in figure 4.4.11). This object will be deleted from the list because it will not be visible in the final image.
- If the points that define the object are inside the field of view and the object doesn't intersect with the edges of the field of view then the object is completely inside the field of view (object C in figure 4.4.11). This object will remain in the list without undergoing any changes.
- If the objects intersect with the edges of the field of view, then the object is partially inside the field of view. Since some parts of the object will be outside the field of view, drawing the entire object would be inefficient. For this reason, the object will be split at the points where it intersects the field of view and only the inside part will remain in the list (object B from figure 4.4.11).

After this processing of the objects in the list, only the ones inside the field of view will remain. The next step is filtering the objects that are completely covered up by other objects from the scene, as they will not be visible in the final image. This way we can avoid situations like the one exemplified in figure 4.4.10 where a wall is drawn only to be completely covered later in the rendering process.

A number of points are chosen at fixed intervals on each object from the list (figure 4.4.12). In order to decide with an acceptable grade of certainty it is enough to conclude that if none of the points chosen on an object are visible, then the object is also not visible.

This approach is not perfect as there can exist an object that has visible parts but all the points chosen are not visible, and therefore the object being considered not visible (object A from figure 4.4.13). However, if the interval between the point is small then even if the object has a visible part that wasn't detected, it would be a very small one and it wouldn't be noticeable in the final image.

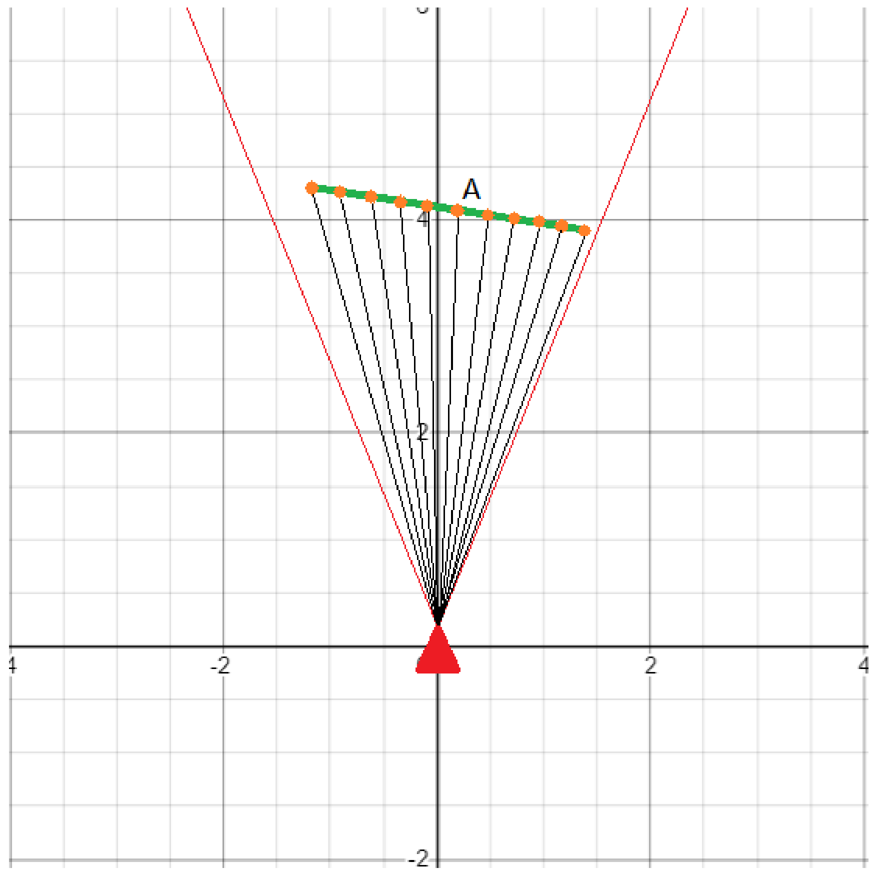


Figure 4.4.12: Point chosen on an object

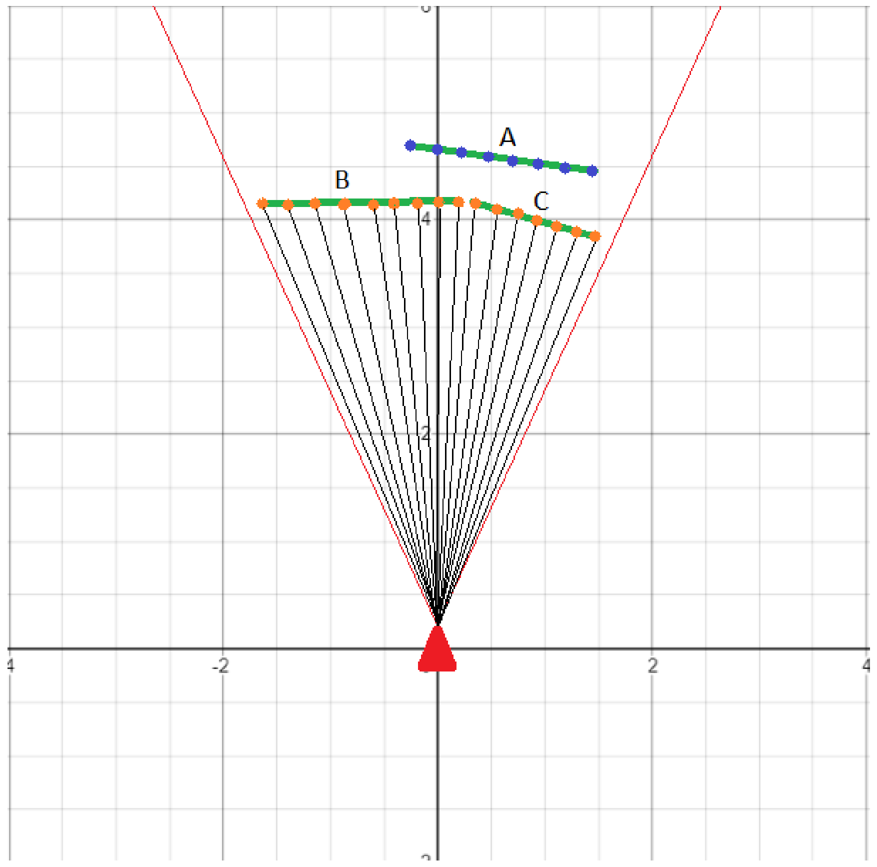


Figure 4.4.13

In order to check if a point is visible, we check if the segment defined by the point and the virtual camera intersects an object from the list. If all the points of an object are labeled as not visible then the object will be deleted from the list of objects that will be drawn on the screen.

If figure 4.4.14 all the points on the objects A and B are visible, some of the points on the object C are visible and none of the points on the object D are visible. Therefore, the objects A, B and C will remain in the list and object D will be deleted.

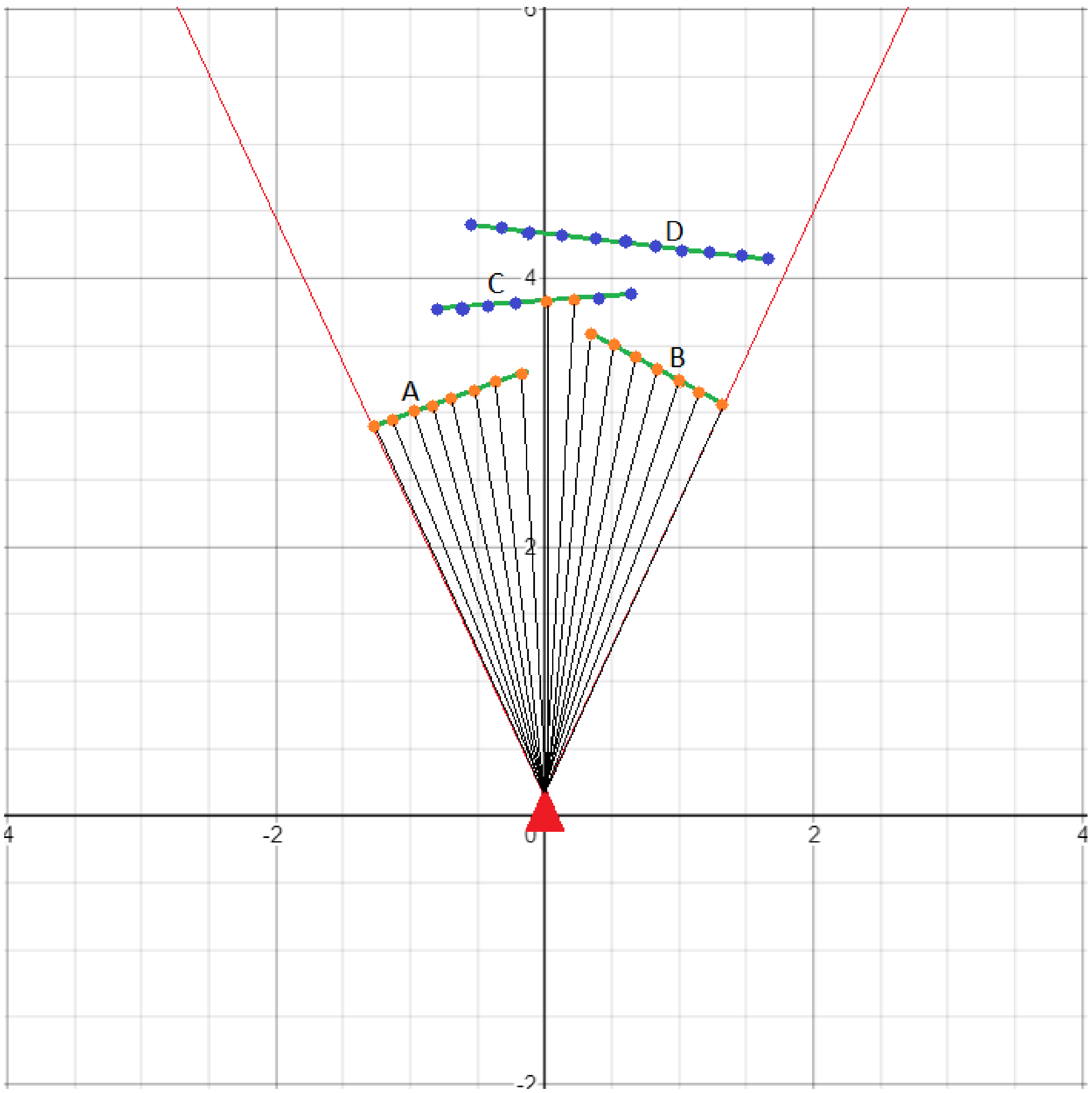


Figure 4.4.14: Object visibility check

At the end of the process the list will contain only the objects that will be visible in the final image, in the order that they must be drawn on screen. The final image is rendered by iterating over the list and drawing each element.

The main goal of this rendering method is to draw the image quickly, in less than 1/30 of a second. However, real-time rendering is often used in applications where the scene is dynamic and where the user can interact with the scene. For this reason, it is important to keep in mind how these changes will affect the rendering algorithm.

This rendering algorithm makes use of a multitude of mathematical formulas to check the visibility of objects and the order in which they must be drawn. The most used formula is the one for determining if two-line segments intersect which is used multiple times for each object when determining its visibility. In all the previous examples the virtual camera has the coordinates (0,0), which greatly simplifies the formula. However, If the user interacts with the scene and the virtual camera’s position is changed, the simplified formula can no longer be used.

Because of the large number of times this formula is used, when the user interacts with the scene it is more efficient to change the position of all the objects in the scene instead of changing the position of the virtual camera. This way the virtual camera will always have the coordinates (0,0) and the simplified formula can always be used.

4.5 User interaction

The player will always be in the same position as the virtual camera (0,0). Similar to the enemies, the player has health points, attack points, a distance from which he can hurt the enemies and, in addition, a movement direction and speed. The movement direction and speed are actually the direction and magnitude of the movement vector used for interacting with the scene.

To stop the player from walking through objects we check for a collision between the movement vector and the objects from the scene, using the formula for the intersection of two-line segments. If such an intersection exists then the player is trying to move through an object, therefore the movement will not be allowed.

Another way the player can interact with the scene is by getting attacked by the enemies. In order for the player to be attacked, two conditions must be met: the player must be in the attack range of the enemy and the enemy must not be blocked by another object. The first step is to determine the distance from the player to all the enemies from the scene. Since each enemy can have a different attack range and only the enemies at a closer distance than their attack range can attack, we must check if the distance to the enemy is smaller than the attack range. In the second step we check if the enemies close enough to attack are blocked by other objects. This is done by checking if the ray from the player to the enemy intersects any other object from the scene. If the ray doesn't intersect with anything then the enemy can attack the player.

The player can also attack an enemy when the enemy is inside the attack range of the player, the enemy is not blocked by other objects and it is in the middle of the field of view. When the user presses the attack button a ray with size equal to the attack range of the player is cast from the virtual camera towards the middle of the field of view. If the first object the ray collides with is an enemy, then the player attacks the enemy. When attacked, the enemy health points decrease, and if they decrease below 0 then the enemy is removed from the scene. When all the enemies are removed the player advances to the next stage.

The direction in which the user wants to move, also known as the movement vector, is represented in figure 4.5.1 with a black arrow. To create the movement, instead of moving the camera in the direction of the movement vector, all the objects will move in the opposite direction. The virtual camera's position relative to the objects will be the same as if the camera was the one that changed its position. As we can see in the figure 4.5.1, objects A and B moved in the opposite direction of the movement vector and the virtual camera's position remains unchanged (0,0).

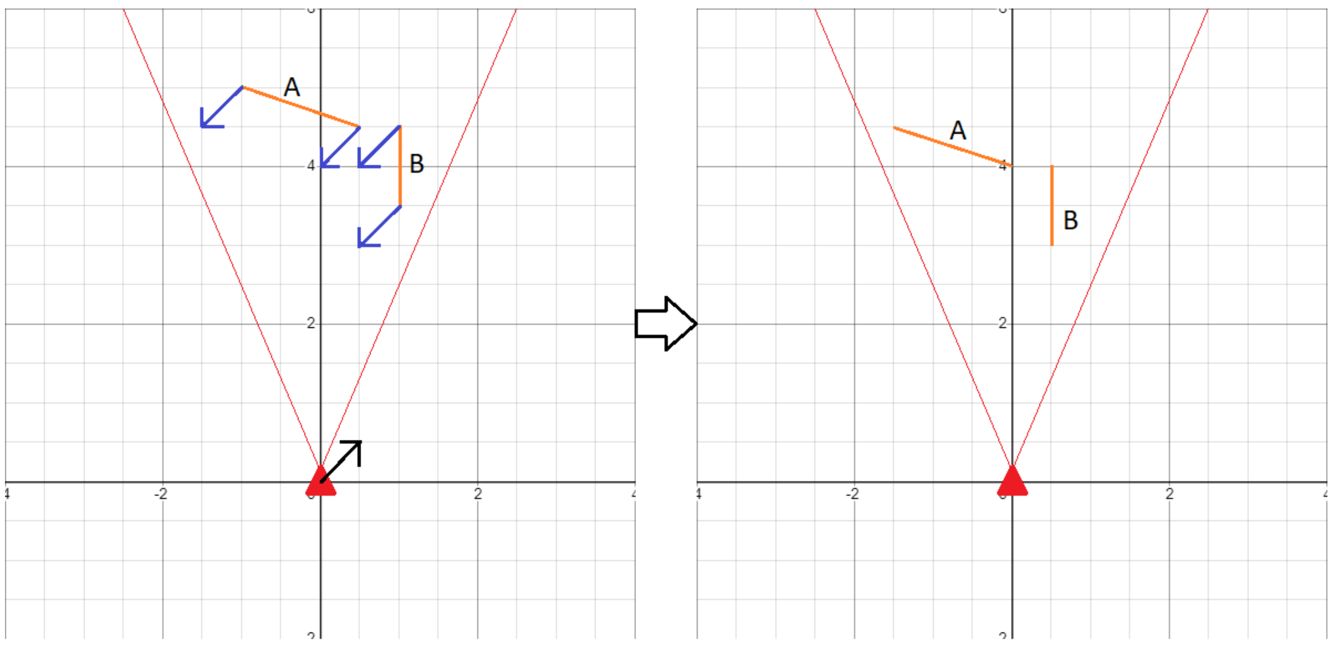


Figure 4.5.1: Movement

When the direction of the field of view must change, the objects will be rotated around the virtual camera instead of rotating the virtual camera (figure 4.5.2). Since the objects must rotate around the origin (0,0), rotating an object can be done using simple formulas. Since the field of view direction will remain the same after rotating, the direction of a new movement vector will not depend on the direction of the field of view and therefore computing the new positions of the objects after they move will be faster. Another benefit is that the slope of the field of view's edges will not change, making it easier to filter the objects outside the field of view.

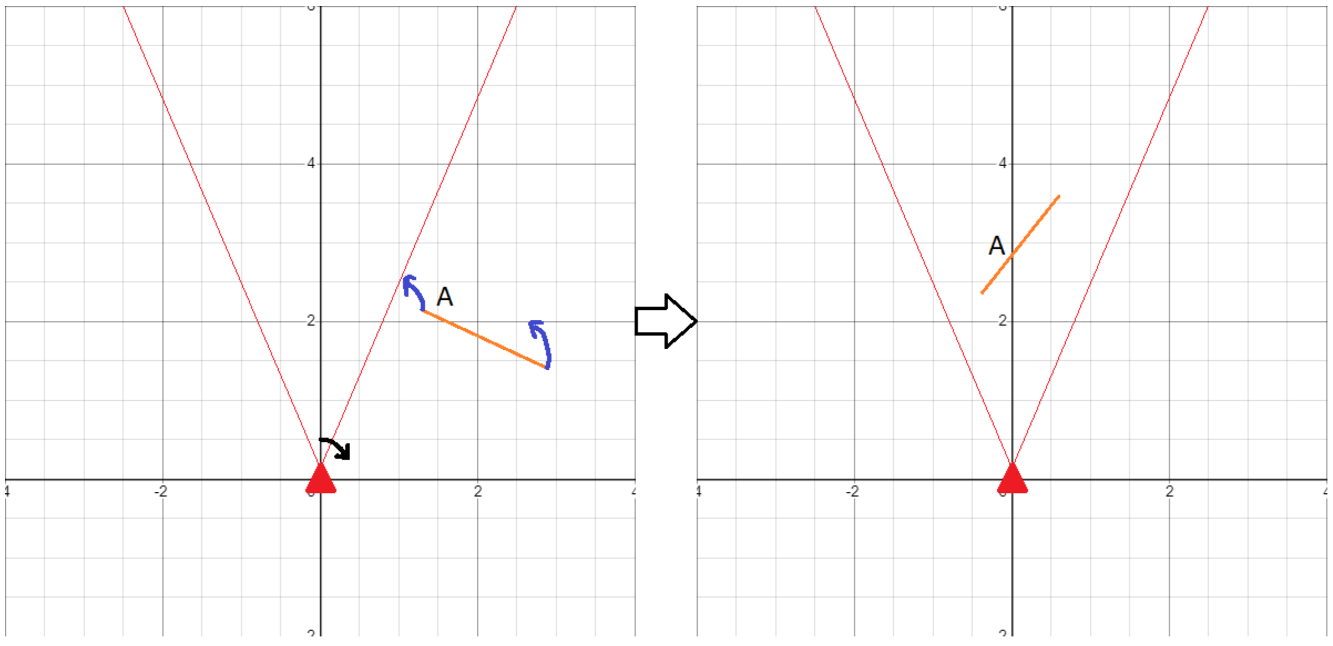


Figure 4.5.2: Rotation

4.6 Lighting

One of the biggest improvements that Doom brought to the table is in the lighting department, featuring ambient lighting that reacted to the player's input (Joe Rybicki, Top 5 Innovations of DOOM (1993))

The scene can contain light sources and therefore the color of the objects must change depending on how much light hits them. To create a convincing effect the following must be true:

1. Light can't pass through objects.
2. The light level of an object is determined by the closest visible light source.
3. The object's color must depend on their light level, the objects with a lower light level being darker.

For each object a ray will be cast from the object towards all the light sources present in the scene. By using the formula for the intersection of two-line segments we can conclude which light sources are able to illuminate the object. If a ray towards a light source intersects an object from the scene, then that light source is blocked by an object and will not be taken into account (1).

In order to compute the light level, we calculate the distance to all the unobstructed light sources and choose the minimum of these distances (2).

To get the new color of the object we compute the weighted average of the original color of the object and the color black. The percentage of black increases with the distance from the light source, thus the objects near a light source will have a color close to their original color and the objects far from a light source will have a color close to black (3).

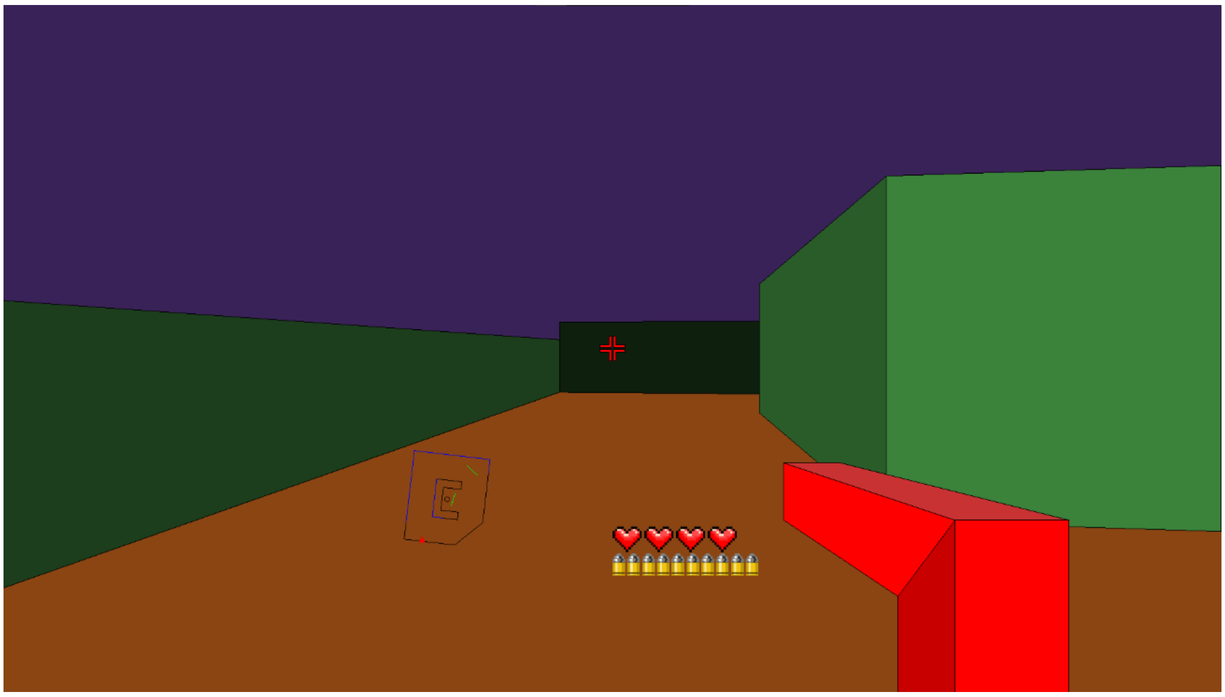


Figure 4.6.1: Walls with different light levels

In the previous figure, the light source is in the same position as the virtual camera and therefore the walls closer to the camera have a color close to their original color, green, and the walls away from the camera are darker. However, this implementation remains to be desired. As we can see, the wall on the left side of the screen has an uniform color, meaning that both the parts close to the camera and the parts away from the camera have the same color.

To create more convincing lighting effect, the objects must have a color gradient. As we can see in figure 4.6.2, this gradient makes the lighting look more realistic.

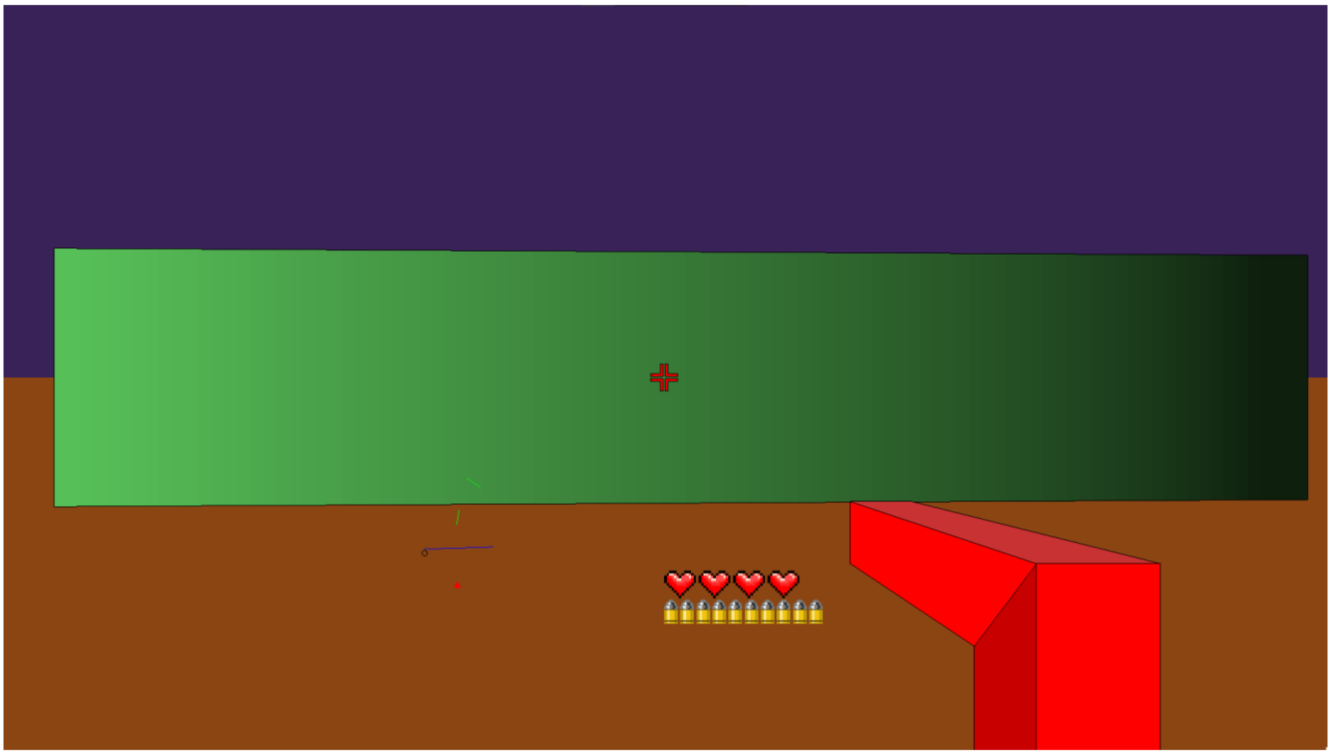


Figure 4.6.2: Wall with light coming from the left

The first step towards a color gradient is splitting the object in multiple segments. In order to have the same level of detail for all the objects from the scene regardless of their size, the number of segments is proportional with the object's size.

The light ever of each of these segments will be determined with the same method previously used to determine the light level of the entire object. It should be noted that by using this method each segment of the object must be drawn individually, therefore increasing the number of polygons drawn on screen. A rendered image using this method can be seen in figure 4.6.3.

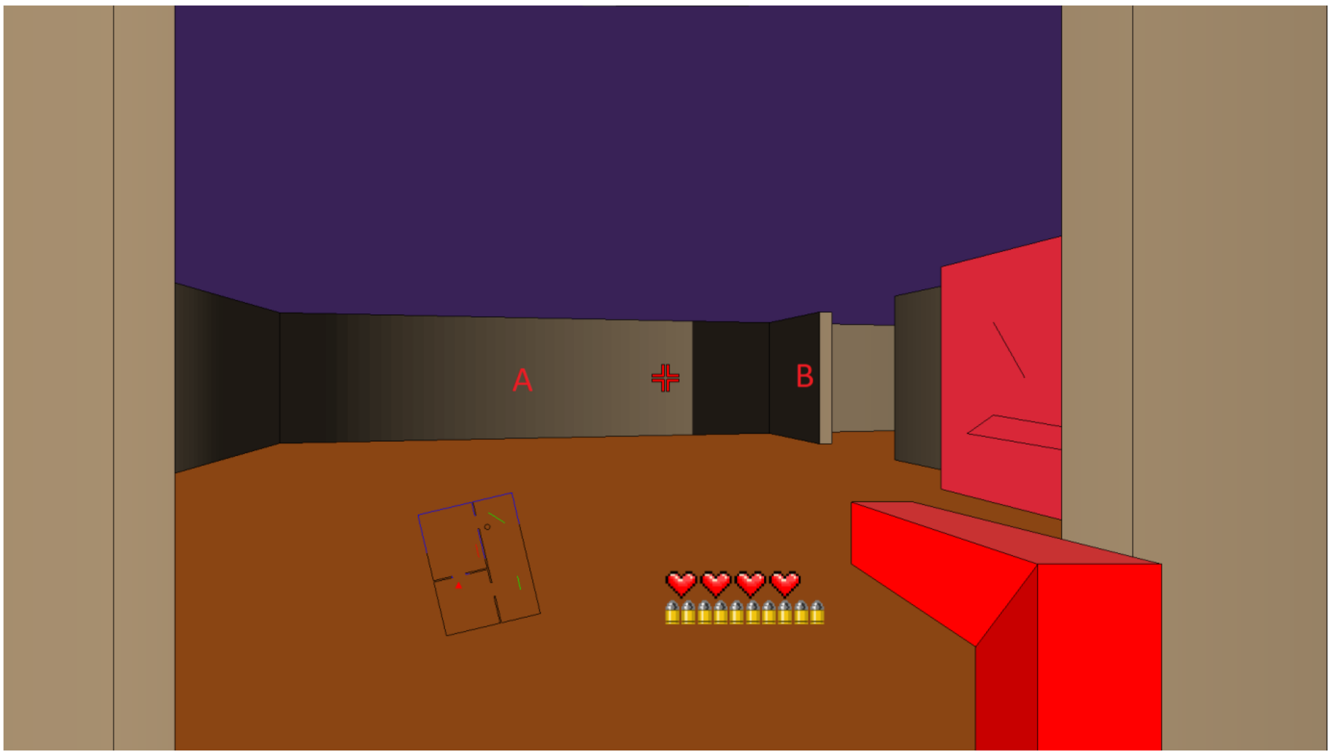


Figure 4.6.3: Final lighting effect

As we can see in the previous figure, The lighting behaves in a realistic manor. On the right there is a door leading to a room where we can find a light. Object A is illuminated by the light and is colored accordingly to the light level it receives. Also, since Object B blocks the light coming from the light source, the part of object A that is behind Object B is dark.

Chapter 5

Advantages and disadvantages compared to the Wolfenstein 3D engine

5.1 Advantages

1 The rendering process offers more flexibility when it comes to the dimensions an object can have and allows the scenes to be designed more efficient.

Because of the way the scenes were stored for the Wolfenstein 3D engine, all objects are composed of one or more cubes. Therefore, the objects have a minimum size, namely the size of a single cube, meaning it is impossible to create thin walls. At the same time, in order to create an object of a large size it must be composed from a large number of smaller objects, meaning that even for simple scenes, the number of objects can be very large.

In contrast to the Wolfenstein 3D engine, the developed rendering engine can work with objects of any size. Since the object's position in the scene is given by the coordinates of two points, the object can have any size and does not require it to be composed of smaller objects. Therefore, the majority of scenes have a smaller number of objects compared to the scenes used by the Wolfenstein 3D engine.

A good example is a simple scene of a rectangular room. In the developed rendering process the room can be composed of only 4 objects (walls), no matter the size of the room. In the Wolfenstein 3D engine however, the number of objects depends on the room's dimensions, and, in addition, the room has a minimum size. As we can see in figure 5.1, the number of objects increases with the size of the camera (Camera 1 -> 4 objects, Camera 2 -> 8 objects, Camera 3 -> 30 objects).

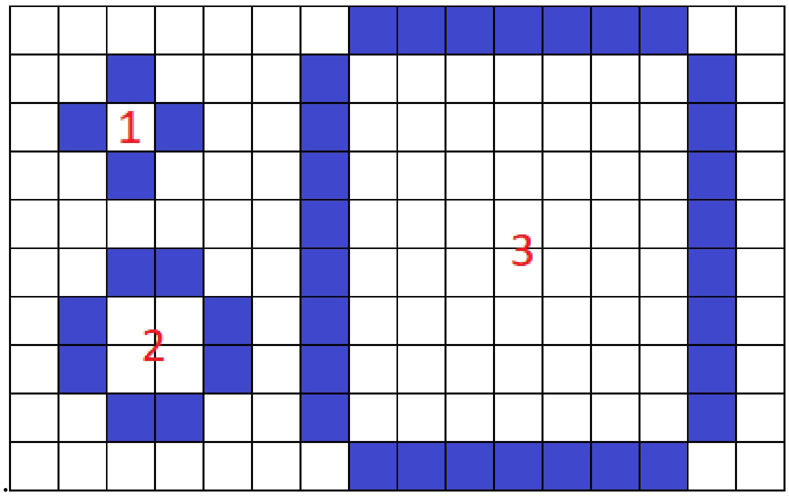


Figure 5.1: Rooms with different sizes

2. The angle of the objects is not restricted by a grid.

Because of the fact that the scenes used by the Wolfenstein 3D engine are stored as a 2d matrix and the position of an object is given by its position in the matrix, all the objects from the scene are restricted to a grid of squares. Therefore, the only possible angles between objects are either 90°, 180° or 270° (E.g., angles A, B and C from figure 5.2), meaning that the scenes can only contain simple geometry.

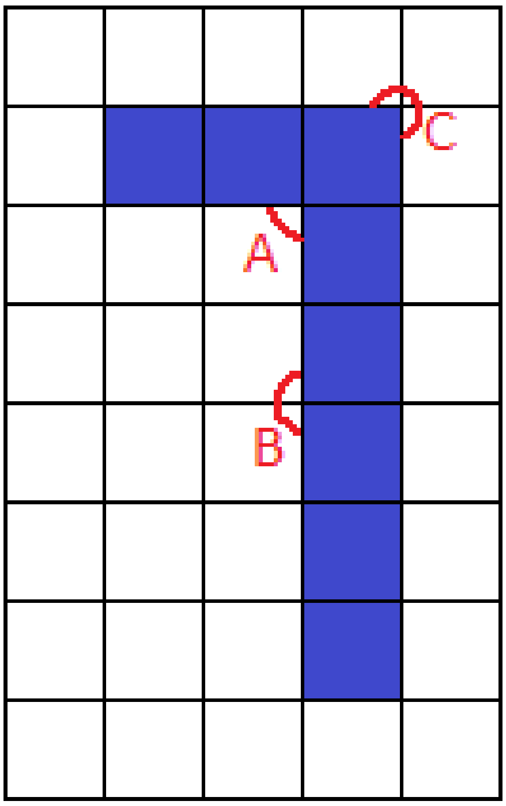


Figure 5.2: Possible angles in Wolfenstein 3D

The developed rendering process uses scenes that store the coordinates of the objects, therefore making it possible for the objects to have any angle between them. As a consequence, the scenes can be more complex, the objects not being restricted to a grid.

3. The walls can have different heights.

The Wolfenstein 3D engine draws the image by casting rays towards the objects in the scene and drawing a column of pixels for each ray. However, when drawing the pixels, it only takes into account the distance to that object and the height of all the objects is considered to be the same. Therefore, it is not possible for the objects to have different heights. In contrast, the developed rendering engine can draw objects of different heights, allowing for a more complex scene.

However, because of the 2d representation of the scene and the way in which the objects are filtered before drawing, this implementation of height still has a few drawbacks. The most notable one is that the height of the object is not taken into account when checking the object’s visibility, meaning that even if a tall object should be visible behind a short object, the tall object will be considered invisible and will not be rendered.

4. Lighting.

In spite of the fact that Wolfenstein 3D doesn't have the capability to work with lights, a similar but crude effect can be achieved using multiple similar textures. In order to create the illusion of lighting multiple types of objects are created, each with a similar texture but with slight variation of the colors. However, this workaround has its drawbacks:

- Each new object contributes to the hard limit of 256 object types, which imposes that this kind of lighting effect is used sparingly.
- The object types cannot change, meaning that the object will always have the same texture and as a consequence, the same light level. Therefore, it is impossible to create dynamic lighting that reacts to the environment.
- An object cannot have more than one light level.

In the developed engine however, the light level of the objects is computed in real time. Therefore, the light level of an object can change according to its environment, the light can be blocked by objects and an object can have multiple levels of light.

5.2 Disadvantages

1. Performance.

The Wolfenstein 3D engine uses a simple and efficient rendering method allowing it to run on very low-end systems, the minimum requirements being a single core CPU with a frequency of 10 MHz and 1 MB of RAM. This great performance comes with the drawback of the small number of features it offers. Most of the missing features, like variable object height, flexible scenes and dynamic lighting, are available in the developed engine. However, these features come with a big performance hit, the minimum system requirements for rendering 30 frames every second being a single-core CPU with a frequency of 400 MHz and 40 MB of RAM. It should be noted that these requirements can be lowered by decreasing the number of light zones for each object.

2. Storage space:

The storage method used to store the scenes by Wolfenstein 3D is a simple one, each scene being a bidimensional matrix of whole numbers. The position of each element from the scene is determined his position inside the matrix and the type of the object is determined by the value of the number stored at that position inside the matrix. In addition to the scenes, a texture for each object type is stored as an image with the dimensions of 64x64.

Because the scenes are stored separately from the textures, when creating a scene, it is impossible to use new object types or new textures. Instead, only the predefined object types can be used and their respective textures can be used. The total number of such predefined object types is 110.

However, a big advantage of this approach is the reduced storage space used by the scene. Because of the fact that the scenes don't contain any textures, when rendering a scene, a look-up table is used to find the correct texture for the objects. This allows the scenes to use the same textures, meaning that they must only be stored once. Another advantage of storing the scene as a bidimensional matrix is that the object's position is determined by his position in the matrix, and therefore not requiring any additional storage space. This makes it possible for a 50x50 scene to be stored using only 2.5 KB.

The storage method used by the developed rendering engine is more complex and more costly. In order for the object to not be restrained by a grid it must store the exact position in space of the objects. In addition, it also stores the height and color of the objects. Therefore, depending on the object's type, it can use anywhere from 23 bytes to 35 bytes to store a single object.

The most important distinction between the two storage methods is that for the developed method the storage space is proportional with the number of objects inside the scene and for the Wolfenstein 3D engine it is proportional with the size of the scene. This is a consequence of the fact that by storing the scene as a matrix, even the positions where an object doesn't exist must be stored. Therefore, while in most cases the scenes stored for Wolfenstein 3D use less storage space, if the scene is large but has a low number of objects the storage method for the developed rendering method uses less storage space.

Chapter 6

Conclusions

Using this rendering method, it is possible for the objects to have any position and orientation inside the scene and to have different heights, colors and light levels. However, the main disadvantage of the scene being 2d still remains, namely the impossibility of areas with the same position but different height levels (E.g., Buildings with multiple floors, bridges, etc.).

This rendering method is not as efficient as the one used by the Wolfenstein 3D engine, however because of the technological advances that took place, most of the electronics have sufficient processing power to use this process in order to display simple real-time 3d graphics either to create an animation or to create an interactive scene for the users.

Bibliography

1. Phill Cameron. How id built Wolfenstein 3D using Commander Keen tech, 2019
2. K. Thor Jensen. The Complete History Of First-Person Shooters, Pcmag 2017
3. Fabiem Sanglard. Doom engine code review, fabiensanglard.net, 2010
4. Fabiem Sanglard. Game engine black book: Doom, fabiensanglard.net, 2018
5. Fabiem Sanglard. Game engine black book: Wolfenstein 3D, fabiensanglard.net, 2018
6. Matthew S. Fell. The unofficial Doom specs, gamrs.com, 1994
7. Lode Vandevenne. Lode's Computer Graphics Tutorial – Raycasting, lodev.org, 2020
8. Hayden Smith. The History of Game Engines, prezi.com 2014
9. David Eberly. 3D Game Engine Design. Morgan Kaufmann Publishers, San Francisco, 2000.
10. David Barr. Upgraded Command Line First Person Shooter (FPS) Engine, onelonecoder, 2018
11. David Barr. Fast Ray Casting in Tiled Worlds using DDA, onelonecoder, 2021
12. David Barr. Line Of Sight or Shadow Casting in 2D, onelonecoder, 2018
13. F. Permadi. Ray-Casting Tutorial For Game Development And Other Purposes, permadi, 1996
14. Matthias Teschner. Computer Graphics Ray Casting, University of Freiburg, 2021
15. Graham Singer. The History of the Modern Graphics Processor, techspot, 2021
16. Leif Johnson. 1993's 'Doom' Wasn't the 3D Game We Think It Was, vice.com, 2016
17. Matt Godbolt. Wolfenstein 3D's map renderer, Youtube, 2018
18. Sjoerd de Jong. An In-Depth Look at Real-Time Rendering, Unreal Engine- Online learning courses, 2019
19. Joe Rybicki, Top 5 Innovations of DOOM (1993), Bethesda Slayer’s club, 2019