Christian Doppler Gymnasium, Zborovská 45, Prague 5

YEAR'S WORK

3D Simulation and Rendering in Microsoft Excel

Elaborated: Ondřej
Čopák Class: 8.M
School year: 2023/2024
Seminar: Seminar in Programming

I declare that I have written my year's work independently and exclusively using the sources cited. I agree to lend my thesis for study purposes.

Prague, 13 January 2024 Ondřej Čopák

Table of Contents

1. Home

The aim of this paper is to present the basic principles of rendering 3D objects in game engines. I chose this topic in order to extend the reach of the worldwide trend of *porting* (older) games, such as Doom, to different environments. Thus, I decided to implement the basics of graphical cube rendering, inspired by the computer game Minecraft, into the Microsoft Excel environment (hereafter referred to as *Excel*).

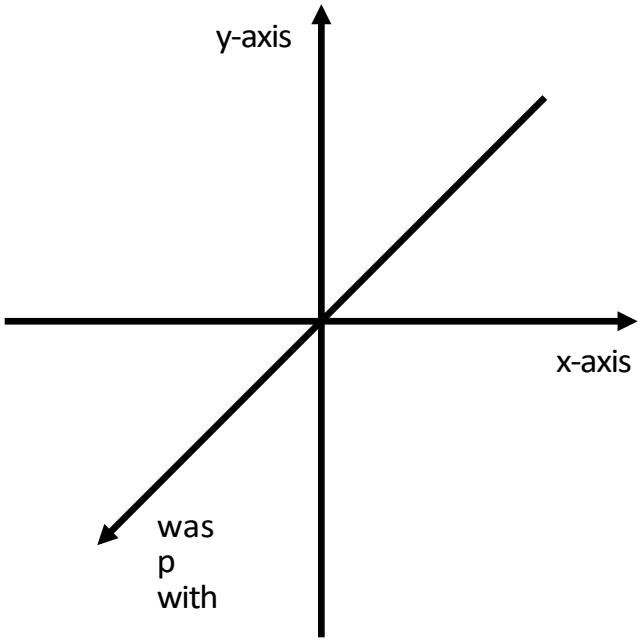
Emphasis is also placed on explaining how Excel can be seen as an environment for programming 3D games. Camera rotation, player movement, conversion to two dimensions and other aspects related to visualizations are analyzed. A large part of this is also the optimization of the code to minimize the number of computations in order to speed up the program. A basic understanding of how computer games work and a basic knowledge of working with coordinates are prerequisites for a good understanding of the work.

The whole thesis is divided into three parts (theoretical basis, implementation and optimization) and is further divided into subsections. The theoretical part deals with the algorithms and mathematics behind moving in three-dimensional space. The implementation section focuses on the concrete implementation in the Excel environment, and the last section of the paper deals with the limiting factors of the environment and further optimizations of the program. The thesis relies only on the literature listed at the end of the thesis and the official documentation of the programming language. All figures in this thesis are of my own creation.

Doom (1993, id Software)
Minecraft (2009, Mojang Studios)
Microsoft Excel (2016, Microsoft)

2. Theoretical basis of 3D environment

The chapter is devoted to the theoretical basis of mathematical operations - movement, rotation in 3D environment and subsequent conversion to 2D. The *X, Y* and *Z* axes are always right-handed throughout the thesis (Figure 2.1). They form the so-called Cartesian coordinate system, which determines the unique coordinates of points [1]. The term *player* throughout the thesis refers to the position from which the rest of the scene is observed. The term *camera*, in turn, refers to the direction from which *the player* is looking.



(Fig. 2.1 - right-hand *X, Y, Z* axes)

2.1 Movement in three-dimensional space

To minimize the complexity of the calculations, it is convenient to place the player at the starting point of the coordinate system, namely at the point [0, 0, 0]. This eliminates the need to work with the player position and the cube position simultaneously and simplifies the calculations. Each cube is therefore moved along each axis according to the original player position.

2.2 Rotation in three-dimensional space

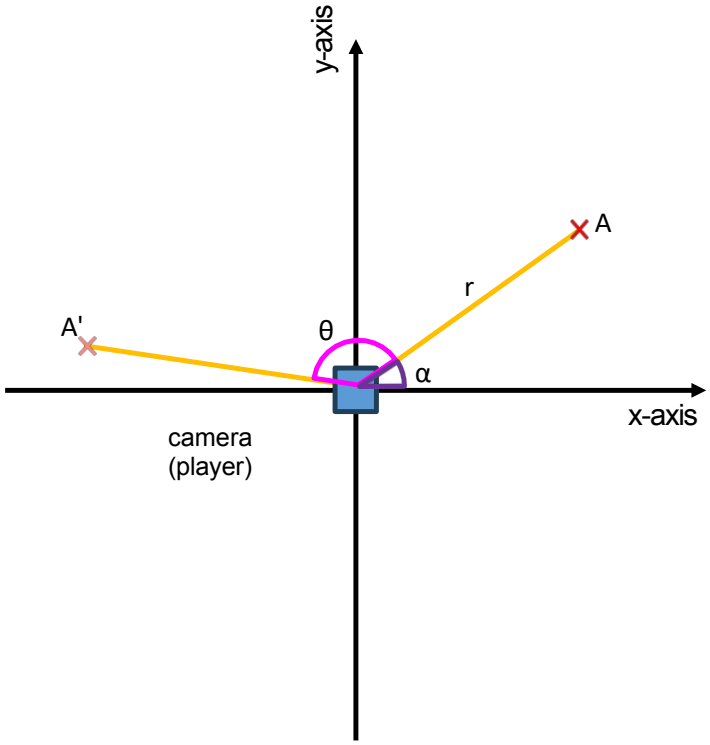
The camera, like the player, also rotates to be oriented in the initial direction [0, 0, 0]. Thus, all points are rotated to remain at the same distance from the player, but at a given camera angle. The program only allows camera rotation in two directions - *up/down* and

left/right. The original game does not allow the player to tilt his head in the direction

to the shoulders. For better visualization, this part of the work works in a 2D plane, where the player is at point [0, 0] and only those axes where the values change during a particular rotation are selectively displayed.

2.2.1 Deriving the formula for rotation in 2D

For accurate object placement and orientation, it is necessary to derive a formula for rotation in the 2D plane. This formula allows to calculate the new coordinates of a point after rotation around the origin. The goal is to calculate the new coordinates of the point [*x, y*] based on the angle of rotation (Figure 2.2).



(Figure 2.2 - rotation of point A by angle θ in the Cartesian coordinate system of the XY plane)

A: a point defined by the coordinates [*x, y*], which is rotated.
And' : the resulting point after rotation.
θ: angle of rotation (yaw, pitch).
α: the original angle between the *X-axis* and the point.
β: total angle of rotation ($\alpha + \theta$).
r: the distance from the origin to a point in the 2D plane.

The angle α is given by the arcsine of the y and x ratios and is also given by the quadrant in which the point is located.

$$\alpha = \arctan \frac{y}{x}$$

From this, the total angle of rotation of point A can be calculated.

$$\beta = \alpha + \theta$$

The new point coordinates are therefore:

$$x' = r \cdot \cos(\beta)$$

$$y' = r \cdot \sin(\beta)$$

In order to avoid the problems associated with division by zero (in the case of $x = 0$) and to take into account the quadrant in which the coordinates are located, we need to modify the formulas so that both coordinates are expressed using the two goniometric functions $\sin(\theta)$ + $\cos(\theta)$. We can therefore start the modifications by using the sum formula for $\tan(\alpha + \beta)$.

$$\tan(\alpha + \beta) = \frac{\tan(\alpha) + \tan(\beta)}{1 - \tan(\alpha) \cdot \tan(\beta)}$$

The value of $\tan(\arctan(n))$ is equal to n.

$$\tan(\beta) = \frac{\frac{y}{x} + \tan(\theta)}{1 - \frac{y}{x} \cdot \tan(\theta)}$$

$$\tan(\beta) = \frac{y + x \cdot \tan(\theta)}{x - y \cdot \tan(\theta)}$$

Next, β from the previous step is substituted into the formulas.

$$x' = r \cdot \frac{x - y \cdot \tan(\theta)}{\sqrt{x^2 + y^2}}$$

$$y' = r \cdot \frac{y + x \cdot \tan(\theta)}{\sqrt{x^2 + y^2}}$$

After making the adjustment, where $r = \sqrt{x^2 + y^2}$ comes out:

$$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)$$

$$y' = x \text{ - } \sin(\theta) + y \text{ - } \cos(\theta)$$

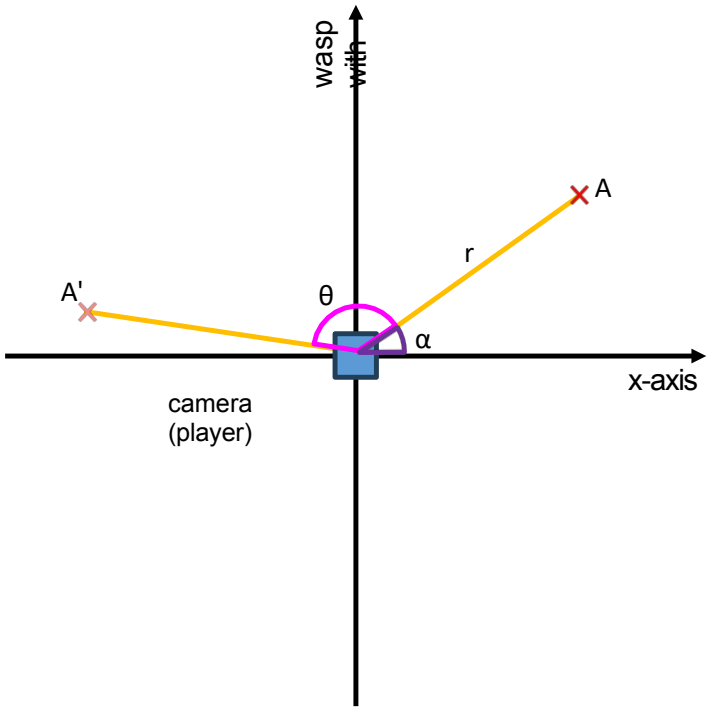
These formulas will be used in both rotations.

2.2.2 Rotation about the Y axis (yaw)

For rotation of the *XZ* plane (rotating the camera left and right) you can use the already calculated formula, where *Y* remains the same and *X* and *Z* change depending on the *yaw* angle (Figure 2.3). I will use the cyclic substitution formula from Section 2.2.1.

θ is the *yaw* angle.

$$x' = x \text{ - } \cos(\theta) \text{ - } z \text{ - } \sin(\theta)$$
$$y' = y$$
$$z' = x \text{ - } \sin(\theta) + z \text{ - } \cos(\theta)$$



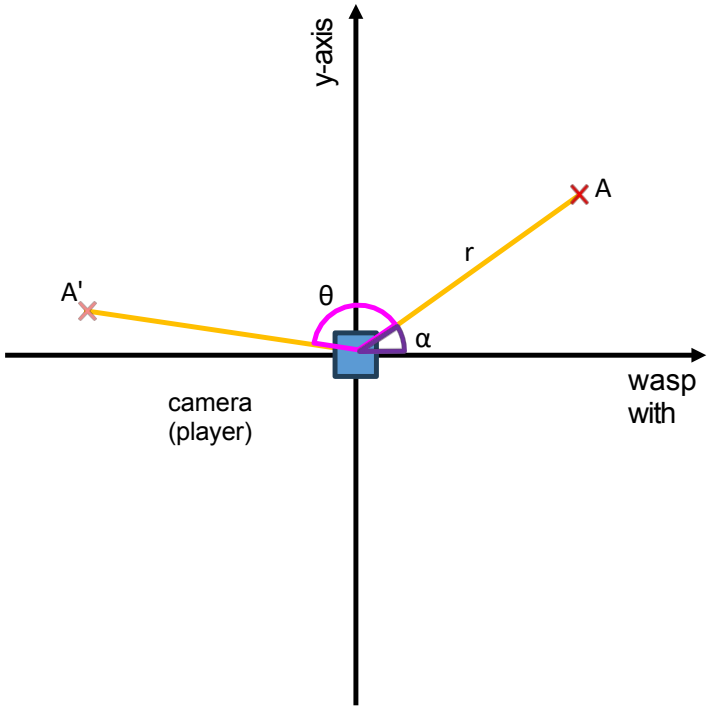
(Figure 2.3 - rotation of point *A* by angle θ (*yaw*) in the case of a camera rotating about the *Y* axis)

2.2.3 Rotation around the X axis (pitch)

For the rotation of the *Y Z* plane (rotating the camera up and down) I will again use the derived formula. This time, the cyclical swapping keeps the *X* coordinate the same, where the *Y* and *Z* values again change depending on the *pitch* angle.

θ corresponds to the *pitch* angle.

$$x' = x$$
$$y' = y \text{ - } \cos(\theta) \text{ - } z \text{ - } \sin(\theta)$$
$$z' = y \text{ - } \sin(\theta) + z \text{ - } \cos(\theta)$$



(Fig. 2.4 - rotation of point *A* by angle θ (*pitch*) in case of camera rotation around *X* axis)

2.3 Display in 2D plane

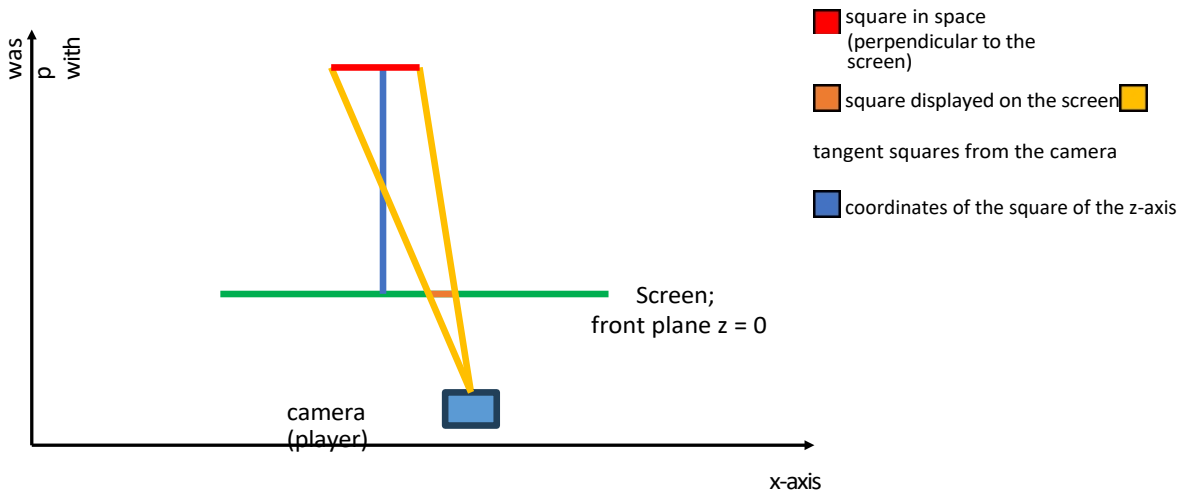
The principle of *perspective projection* is used to convert 3D space into a 2D plane in the system. This principle provides an in-depth visualization of spatial objects on the screen surface [2]. Perspective projection works with so-called normalized coordinates, their principle is explained in the previous part of the thesis (see Section 2.1 and 2.2). The normalized *x, y* and *z* coordinates, obtained by looking at the player in the *z-axis* direction, can then be converted to the corresponding 2D coordinates.

nice screens.

$$X_{2D} = \frac{X3D}{Z_{3D}}, \qquad Y_{2D} = \frac{Y3D}{Z_{3D}}$$

These formulas exploit the similarity of triangles (*red side* with camera and *orange side* with camera), where the *orange side* was created as a line between points that are the intersections of tangents (from the camera to the original *red* square) with the screen plane [3]. The similarity coefficient is based on the distance from the XY plane (z coordinate) (Figure 2.5).

As the distance of the sides of the cube from the screen increases, the content of the cube (the displayed shape on the screen) will decrease. Thus, the program calculates positions based on the proportions in the screen axes with the distance of the *z* coordinate. If the cube passes through the screen, the point is shifted 1 unit in the *z-axis* direction to avoid a division by zero problem and allow the user to see that side on the screen (code 2.6; line 2).



(Fig. 2.5 - perspective view of squares on the screen, top view - converted to 2D for better orientation)

```
1  If intersection Point (2) == 0 Then
2       intersection Point = Array ( intersection Point (0), intersection Point (1),
3  1) End If
4
5  projected X = intersection Point (0) / intersection Point (2)
6  projected Y = intersection point (1) / intersection point (2)
```

(code 2.6 - perspective view of points; modified)

3. Programme implementation

In this part I will focus on the concrete implementation of the whole program in the Excel environment using the Visual Basic for Applications (*VBA*) language. I will then detail the key algorithms, functions and classes necessary for the program to function. The program is written in English to make it accessible to the widest possible range of users. The source code together with comments is available in the appendix of the thesis.

3.1 Excel environment

In Excel, cells act as pixels that are organized into a table that acts as a screen, where they create the resulting image (snapshot).

The Excel file consists of three sheets. The first sheet serves as the main screen, which displays the game scene. The second sheet is used for settings, statistics listing and other controls. The last sheet is used to store all the textures. Each texture is always divided into three groups of 8x8 cells. The first group contains the top texture of the cube, the middle group represents all the sides of the cube, and the third group contains the bottom texture of the cube.

3.2 VBA (Visual Basic for Applications)

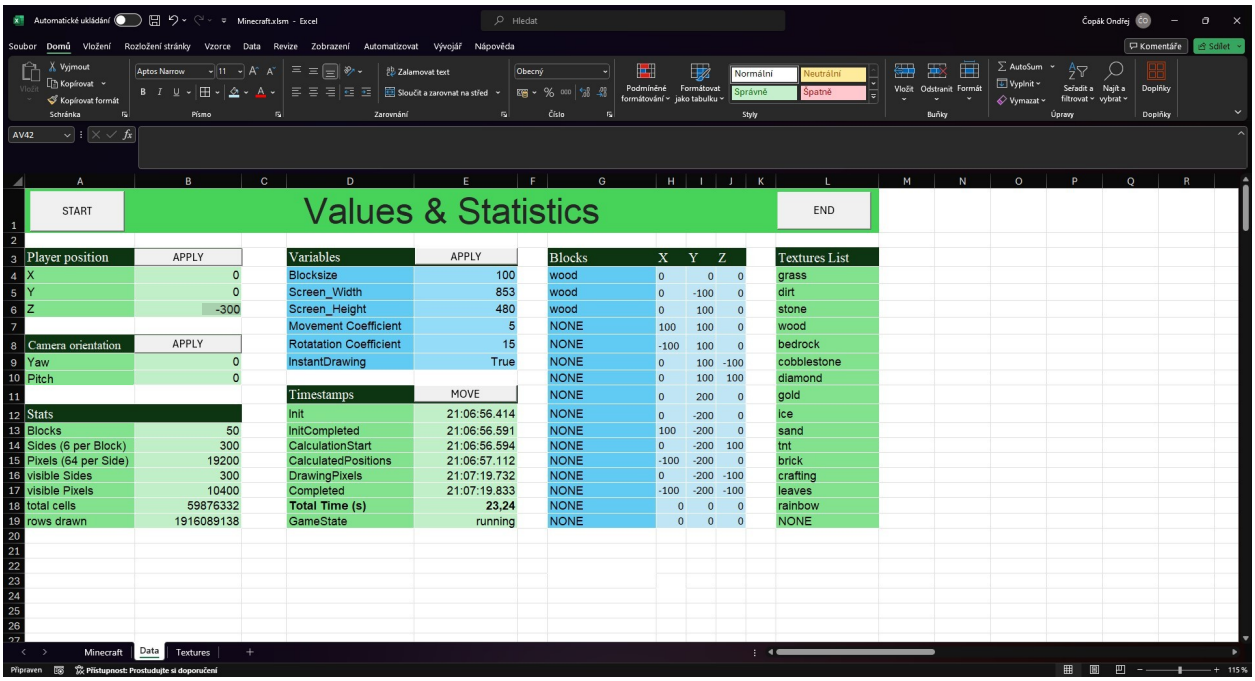
For the implementation of the program I chose the VBA programming language because it is the only language that Excel supports. This choice has its positives - all documentation for functions is created only for this language, but it also has its negatives, because the user does not have the possibility to choose a better optimized language.

VBA is one of the interpreted languages [4]. Code written in VBA is translated into proprietary executable code called *Microsoft P-code* and then stored as source code in an Excel document [4]. VBA allows, for example, Excel control and cell manipulation, which can be used to implement a game environment.

3.3 User interface (program control)

The second workbook *Data* contains all the controls for the entire program. Here the user can set the player position, camera orientation, texture and cube positions, or the image settings, and can also view various statistics on when and how many calculations the program has performed on each frame.

This unit takes a closer look at the different parts of the user interface. All values are set by changing the text value of a cell and then confirming it (by clicking on the *Apply* button) (Figure 3.1).



(Fig. 3.1 - user interface of the program in the Excel environment)

3.3.1 Top rail

The top bar contains a title that informs the user that they are in the game controls and stats section. On the left is the *Start* button, which when clicked by the player will trigger the Init function, which initializes the entire game. On the right is the button to exit the game.

3.3.2 Player position

Here the user can set the current position of the player in *x, y, z* coordinates.

3.3.3 Camera orientation

Similar to the player position section, here the user sets the camera angles. The program also updates the current camera values after the movement function is executed.

3.3.4 Stats

The Stats section includes statistics, including the total number of cubes in the game environment, the total number of sides (number of cubes multiplied by 6), and the number of different colored texture units (Pixel - the number of sides multiplied by 64, which is the number of pixels in the texture). In addition, this section provides information about how many positions of sides, pixels and cells the program has calculated. This data can be used as an indicator of success in optimizing the program.

3.3.5 Variables

Here the user can change the default values of the graphic output. *Blocksize* specifies the side length of the cubes and thus a unique coordinate scale. *Screen Width* and *Screen Height* predict the screen size in number of cells. *Movement* and *Rotation Coefficient* specify scales of player movement in specific directions and camera rotation in degrees along specific axes. *InstantDrawing* influences whether individual cells are painted sequentially, which can be used to graphically visualize the entire imaging process, which is also more time consuming. At the same time, all cells will be colored without continuously rendering the program, which significantly speeds up the plotting process.

3.3.6 Blocks

The user is offered to change the texture and position of the centres of the individual cubes. The default texture value is *NONE*, which means that the cube will not be included in the program. To insert a cube into the program, the user must select one of the 16 predefined textures (name) and then select its coordinates, which should always be unique, and multiples of the cube's side size (*Blocksize*) so that the cubes do not overlap.

3.3.7 Texture List

This is a list that allows the user to see what different cube textures are available.

3.3.8 Timestamps

The last section serves as a program speed indicator. Each cell contains the exact time of the operation to the millisecond. In the first line, the user sees the time when the program was started, and the following lines show when the various functions for calculating the image were completed. The last line shows the total time of the snapshot calculation from the execution of the user input to the completion of the cell display. This indicator can again be used to optimize and test the program.

3.4 Classes (Classes)

Classes make it easier to organize the code and contribute to the modularity and readability of the program. The classification of data and functions into logically related units facilitates the management and clarity of the programming environment, which is crucial when developing complex systems such as game environments. In total, the program uses 8 custom classes.

3.4.1 Gaming environment

3.4.1.1 Player

The Player class is used to store important information about the player in the game environment. These values include the player's current position in three dimensions (*x, y, z*) and camera angles (*yaw, pitch*). The use of this class makes the code cleaner because instead of using 5 different variables, variables of one class are used. In addition, the use of classes has the advantage of better scalability of the program, for example when adding more players.

3.4.1.2 Game

The function of the Game class is to store all the various scene settings and game controls. This function includes all the settings related to graphics processing that are listed in the 3.3.4. As with the Player class, the Game class is used to make the program clearer. At the same time, this class could be used to store different profiles of game settings.

3.4.1.3 Calculations

The Calculations class optimizes calculations of goniometric functions (cosine, sine, tan- gens) in a game environment. It contains pre-calculated values of these functions for angles from 0 to 359 degrees, eliminating the need for repeated calculations during program execution. By using this class, angles can be processed more efficiently, minimizing the burden associated with calculating goniometric functions when computing cube positions.

3.4.1.4 Textures

Textures is a class that contains functions for loading and saving colored textures. The LoadInput method loads textures from a specified range of cells in a sheet and stores them in an internal collection (sheet) according to the specified texture name. Each texture is represented by an 8x8 matrix of colors.

The GetColorCollection method allows you to retrieve a stored collection of colors associated with a specific texture, which speeds up access to textures for rendering game objects.

3.4.1.5 Stats

The Stats class is used to store essential statistics about the game environment in a specific *frame*. It stores information about the number of cubes (*Blocks*), visible pixels (*VisiblePixels*), visible sides of cubes (*VisibleSides*), number of rendered cells (*Cells*) and rendered rows (*RowsDrawn*). This data can be used to evaluate optimization functions, such as how many sides of cubes the program did not need to count because they are out of the player's field of view. As with the Player and Game classes, this class is used to make the code clearer.

3.4.2 Geometric objects

3.4.2.1 Block

The Block class stores information about individual cubes. It contains variables such as *distance*, which represents the distance of the center of the cube from the player, *point*, which is the position of the center of the cube, and *sides*, a collection representing all six sides of the cube.

3.4.2.2 Side

Side is a class that, like Block, contains information about the sides of a cube. It contains data such as *distance*, which indicates the distance of the center of the side from the player, *texture*, which indicates the texture of the side (Texture class), *midPoint*, which is the position of the center of the side, and *orientation*, which indicates the orientation of the side to (*Up/Down/Left/Right/Top/Bottom*). It also contains the vertices of the page, labeled *a, b, c*, and *d*.

3.4.2.3 Pixel

The last Pixel class is similar to the Side class. However, unlike this class, it does not include center and distance from the player. Instead, it contains a specific color based on the pixel texture. Each instance of the Side class always corresponds to 64 instances of the Pixel class.

3.5 Procedures (Sub, Functions)

The whole program is divided into five modules (files), which thematically group functions that perform different tasks when calculating a snapshot or interacting with the Excel environment. As with the classes, the modules help to make the code clearer.

There are two types of procedures used in this environment: the *Sub* and the *Function*. The Sub (*Sub- process*) procedure, unlike *Function*, cannot return a value. However, any changes made to the values of variables in this procedure will be preserved after the procedure is terminated. Sub procedures can also be assigned to buttons and keyboard shortcuts. *Functions* are usually used to perform mathematical operations and must always return a pre-declared value. *Sub* procedures are often used to interact with Excel, i.e. work with cells. In this thesis, the term *function* includes both procedures (functions and sub procedures), I will focus on selected functions that are most essential to understanding the operation of the program. In more detail work devotes to optimization algorithms in Section 4.3.

3.5.1 Main

This module contains the most important functions for visualizing the game environment. The main part contains procedures that numerically process input values and create an image from them.

3.5.1.1 Init (Sub)

The process will only start when the button is clicked. When clicked, all textures, cube positions, player positions, camera settings and goniometric function values are loaded. Next, the functions for calculating the current frame and creating keyboard shortcuts are called.

3.5.1.2 Move (Sub)

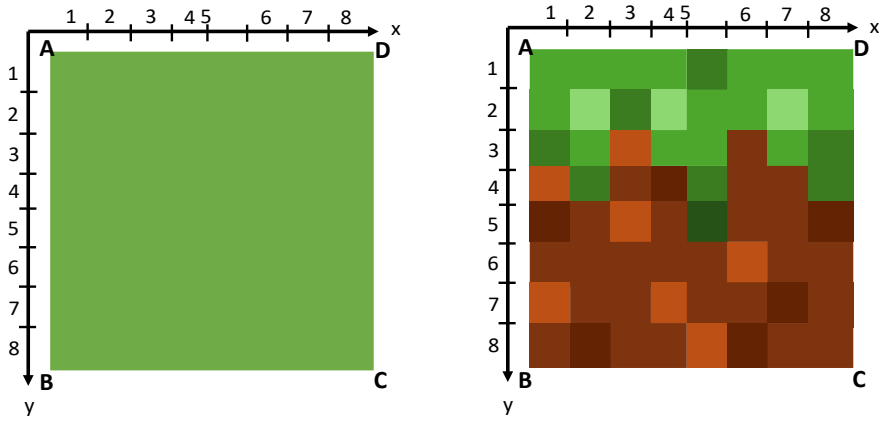
First, the previous frame is erased by recolouring it with the same colour. After that, all functions that ensure the recording of statistics and program execution time are triggered here. Next, three functions are called in turn - CalculatePositions, ApplyTexture and ConvertDraw2D, which I cover in detail in Chapters 3.5.1.3 to 3.5.1.5. The result of these functions is a collection of all the pixels that make up the resulting image.

3.5.1.3 CalculateSides (Function)

The function first creates a collection of Block classes and assigns them a texture and position based on user input. It then creates a new collection of Side classes for each cube with six sides assigned. It calculates new positions for all the vertices of the sides based on the player's position and the camera's orientation using the CalculateCoordinates function. Finally, it ranks all sides by distance from the player from farthest to closest, so that the player cannot see through the sides.

3.5.1.4 ApplyTexture (Function)

The ApplyTexture function receives a Side collection as input and returns a Pixel collection. For each side, the function creates 64 smaller sides (a Pixel object). These pixels are defined in the same way as the sides - by the four vertices *a, b, c, d, which* are created as $\sum_{k=1}^{8}$ (for *k* 1 to 8) times the vectors $\overrightarrow{ab}$ and $\overrightarrow{ad}$. (see Figure 3.2, code 3.3; line 5). Then apply the texture according to the orientation (code 3.3; line 16) and check that all vertices are in the plane in front of the player. If so, the pixel is saved.



(fig. 3.2 - visualization of texture application - 1x Side on the left, 64x Pixel on the right)

```
Function Apply Texture ( allSides Pre As Collection ) As Collection
    For Each s Index In allSides Pre
        For x = 0 To 7
            For y = 0 To 7
                ad = (( s.d (0) - s.a (0)) / 8 , (s.d (1) - s.a (1)) / 8 , (s.d (2) - s.a (2)) /
                    8)
                ab = (( s.b (0) - s.a (0)) / 8 , (s.b (1) - s.a (1)) / 8 , (s.b (2) - s.a (2)) /
                    8)

                a3 = (s.a + ( ad (0) * x + ad (1) * x + ad (2) * x) + ( ab (0) * y + ab (1) * y
                    + ab(2) * y))

                b3 = (s.a + ( ad (0) * x + ad (1) * x + ad (2) * x) + ( ab (0) * (y + 1) + ab
                    (1) * (y + 1) + ab (2) * (y + 1))

                c3 = (s.a + ( ad (0) * (x + 1) + ad (1) * (x + 1) + ad (2) * (x + 1)) + ( ab
                    (0) * (y + 1) + ab (1) * (y + 1) + ab (2) * (y + 1))

                d3 = (s.a + ( ad (0) * (x + 1) + ad (1) * (x + 1) + ad (2) * (x + 1)) + ( ab
                    (0) * y + ab (1) * y + ab (2) * y))

                If s. orientation = " up" Then
                    col = texture Color(y + 1 + 18)(8 - x)
                Else If s. orientation = " down " Then
                        col = texture Color(y + 1)(8 - x)
                Else
                        col = texture Color(y + 1 + 9)(8  - x)
                End If

                 If Is PointInside FOV (a3 , b3 , c3 , d3 ) = TRUE Then
                    Set new Pixel = New pixel
                    New Pixel. Initialize a3 , b3 , c3 ,  d3 , col
                    AllSquares . Add new Pixel
                End If
            Next y
        Next x
    Next sIndex
Set Apply Texture = allSquares
```

(code 3.3 - texture application of each side; modified)

3.5.1.5 ConvertDraw2D (Function)

The last function ConvertDraw2D aims to convert all Pixels from 3D to 2D and display them in specific cells. For each Pixel, the coordinates are calculated according to the algorithm described in Section 2.3. The previous part of the program prevents the input of any coordinates that have a negative z coordinate.

The problem of division by 0 when the coordinate $z = 0$ is solved by setting it to $z = 1$, so the player will see the object. In the last part of the loop, these points are rescaled according to the screen size setting (code 3.4; line 8).

Then, for each quadrilateral (Pixel), based on all 4 vertices, all the coordinates of each side are found using *Bresenham's algorithm* (see Section 3.5.2.3). The function then determines which coordinates on the screen lie inside the quadrilateral, thus obtaining all the coordinates of each Pixel.

```
1  If Int( intersection Point (2)) = 0 Then
2      intersection Point = Array ( intersection Point (0), intersection Point (1), 1)
3  End If
4
5  projected X = intersection Point (0) / intersection Point (2)
6  projected Y = intersection point (1) / intersection point (2)
7
8  If G. screen Height < G. screen Width Then
9      screen X = Int(( projected X + 1) * 0.5 * G. screen Height + (G. screen Width – G.
           screen Height) / 2)
10     screen Y = Int ((1 - projected Y ) * 0.5 * G. screen Height)
11 Else
12     screen X = Int(( projected X + 1) * 0.5 * G. screen Width )
13     screen Y = Int ((1 - projected Y ) * 0.5 * G. screen Width + (G. screen Height - G.
           screen Width ) / 2)
14 End If
```

(code 3.4 - coordinate conversion from 3D to 2D; modified)

3.5.2 Geometry

The Geometry module contains all functions that deal with coordinate calculation. There are 3 essential features.

3.5.2.1 CalculateCoordinates (Function)

This function is an implementation of the algorithm already mentioned in Section 2.1 and 2.2. First, the coordinates of each side must be shifted by the opposite coordinates of the player's position, and then the order of rotation around the X and Y axes does not matter.

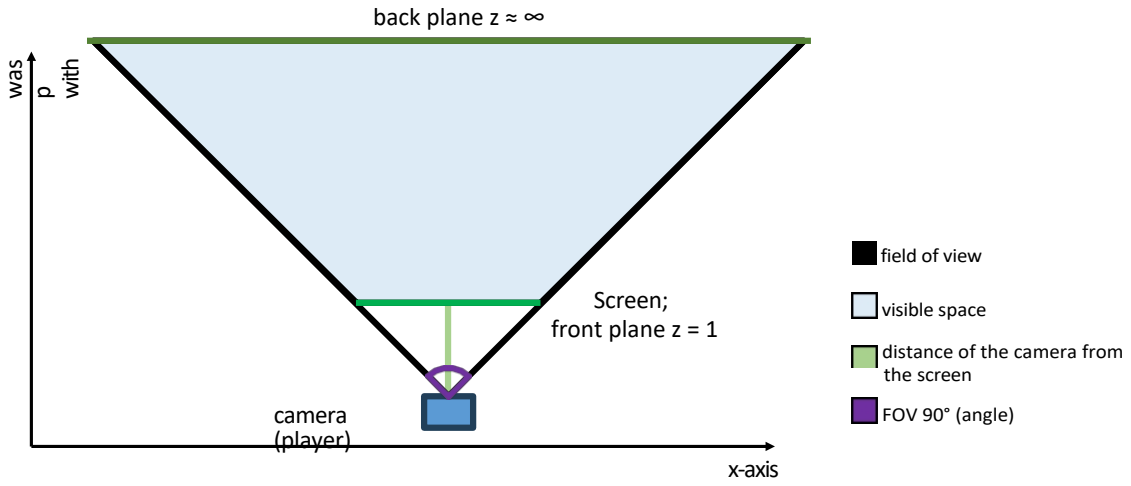
3.5.2.2 IsPointInsideFOV (Function)

The purpose of the procedure is twofold. To verify that the point is in the plane in front of the player and also to verify that the point is in the player's field of view. The first condition is satisfied if the coordinate $z > 0$.

The second condition is based on the mathematical solution of the 2D conversion in Section 2.3. The player's field of view resembles a comoving quadrilateral needle, where the upper base lies in the player's screen plane ($z = 0$) and the lower base runs towards the infinite *z-coordinate* (Figure 3.5) [5]. V

each plane $z = k$, $k \in (0, \infty)$, the intersection of the given kth plane with the field of view is a rectangle, so that

just calculate if the point (based on the position of z) is in it. First we need to calculate the scale of the rectangle. This also depends on the *z-coordinate*. The larger the *z-coordinate*, the larger the rectangle will be. The scale is calculated using the goniometric function tangent (Figure 3.5 and code 3.6). The function finds the lengths of the smaller side and then calculates the length of the other side of the rectangle depending on the aspect ratio of the screen (code 3.6; line 2). If both conditions are met, the function returns true.



(fig. 3.5 - player's field of view, top view - converted to 2D for better orientation)

```
1  radius = C. tan (90 / 2) * point (2)
2  If G. screen Width > G. screen Height Then
3      If ( radius * G. screen Width / G. screen Height) - projected Point (0) >= 0 And radius -
           projected Point (1) >= 0 Then
4          result = TRUE
5      End If
6  Else
7      If radius - projected Point (0) >= 0 And ( radius * G. screen Height / G. screen Width ) -
           projected Point (1) >= 0 Then
8          result = TRUE
9      End If
10 End If
```

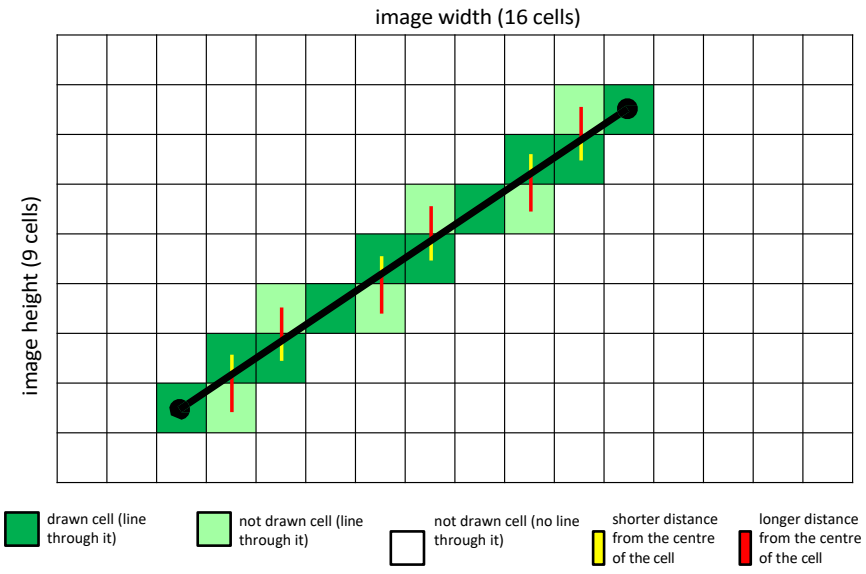
(code 3.6 - check if a point is within the field of view; modified)

3.5.2.3 GetLinePixels (Sub)

The final function of the Geometry module is to calculate all 2D coordinates of a line segment between two points. The result can be achieved using *Bresenham's algorithm*.

Bresenham's algorithm uses incremental decision making to decide which pixel to select for rendering. It starts by calculating the differences between the coordinates of the endpoints, then it traverses the points in between, selects the appropriate pixels, and updates the decision variables (Figure 3.7) [6].

Further, the program optimization only stores the lowest and highest row (x) value in the dictionary according to the column (y) key. Since this process is done for each point of the quadrilateral, and since the resulting output is a set of pixels that lie in a square, for each y, all points from the lowest x to the highest x must also be an element of the quadrilateral. This information can then be used when plotting cells row by row, since in Excel it takes the same amount of time to set the color of a single cell as it does to set the same color of multiple cells in a rectangular range [7].



(Figure 3.7 - Bresenham's algorithm for line-based cell retrieval)

3.5.3 Functions

Here you can find functions that compare collections (mostly cube sides) according to given criteria.

3.5.3.1 RemoveDuplicateSides (Sub)

This optimization function removes any duplicate pages that have the same center. When all cubes are at a distance of multiples of the side lengths, each center can only have sides with the same orientation. At the same time, unless there are multiple cubes in one location, there can be at most two sides in each side center.

The function achieves the result by converting the collection (leaf) to a center-indexed library and then back to the collection. Since each index in the library contains only one value, the result is always a leaf where the side with the original cube center farther from the player remains. An explanation of this behavior can be found in Section 4.3.4.3.

3.5.3.2 ReverseCollection (Function)

This is one of the basic functions that had to be manually implemented. This function returns a new sheet that is ordered from the last element to the first.

3.5.3.3 SortByDistance (Function)

The output of this function is a sorted hand in descending order of the distance of the sides from the player. The pro- cedure uses a custom implementation of *QuickSort* number ordering.

3.5.3.4 QuickSort (Sub)

QuickSort incrementally divides a sheet (collection) of numbers into smaller sheets using a selected element called a *pivot*. The elements are then arranged so that elements smaller than the *pivot* are on the left and larger elements are on the right [8]. This process is repeated recursively until the sheet is sorted.

3.5.4 Visual

The penultimate file contains functions that communicate with Excel. The functions are primarily concerned with coloring cells, writing values, and retrieving settings.

3.5.4.1 ClearScreen (Sub)

The purpose of this procedure is to clear the last frame of all displayed cubes. The function does this by setting the color of all cells to a solid color, by default blue, which represents the sky.

3.5.4.2 FillCellsInRange (Sub)

This function receives as input the coordinates of the upper left corner, lower right corner and color for the given cell range. This function then verifies that all cells are on the screen and colors them with the selected color (based on the textures).

3.5.4.3 SetPlayer/Variables (Sub)

Both functions will load information regarding player positions/game environment settings from the 2nd sheet (settings) into the program memory. Both functions are always called before the calculation of the current frame starts.

3.5.5 Keys

The last module takes care of controlling the game using keyboard shortcuts. Controlling the game using the keyboard is an important element to simplify the control of the game, instead of clicking buttons (on the screen) or manually adjusting the player's numerical positions.

Excel does not allow you to assign individual keys to trigger functions, so the *left shift* key has been assigned to all shortcut keys. Therefore, instead of moving forward with the *w* key, the game is controlled with *shift+w*. In the following functions, shift is omitted from the shortcut keys for clarity.

When each function is triggered, it first adjusts the player position or camera rotation relative to the hotkey and then triggers the function to calculate the current frame. The *moveBy* and *rotateBy* variables determine how much the player moves and rotates and can be changed in the settings workbook. In the thesis, the +- symbol indicates whether the value is positive or negative depending on the direction the player is moving.

3.5.5.1 BindKeys (Sub)

When you click the *Start* button, this function starts. It sets all the keyboard shortcuts used to control the game from the procedures listed below.

3.5.5.2 FreeKeys (Sub)

Unlike BindKeys, this feature deletes all shortcuts used to control the game.

3.5.5.3 MoveUp/Down (Sub)

Keyboard shortcuts: *space/x*
Y-axis movement

Unlike movement along the *X* or *Z* axis, it does not depend on the rotation of the camera, so when the player moves up and down, it only changes along the *Y* axis.

```
1  P.y = P.y +- G. move By
```

(code 3.8 - player moving up, down; modified)

3.5.5.4 MoveLeft/Right (Sub)

Keyboard shortcuts: *a/d*
Movement along *X, Z* axes

Here we need to take into account the rotation of the player's camera along the *Y* axis. If the player is at point [*0, 0, 10*], the rotation of the camera *y* would be 0, so the player should move only along the *X* axis. If the player were standing at point [*10, 0, 0*] and the camera was $y = -90^{\circ}$, then the player must move along the *Z-axis*. Therefore, it is necessary to use goniometric functions depending on the rotation of the camera.

```
1  dx = C. sin (P. yaw ) * G. move By
2  dz = C. cos (P. yaw ) * G. move By
3
4  P.x = P.x -+ dz
5  P.z = P.z +- dx
```

(code 3.9 - player moves left, right; modified)

3.5.5.5 MoveFront/Back (Sub)

Keyboard shortcuts: *w/s*
Movement along *X, Z* axes

As with the left and right movement, the up and down movement again works with coordinates relative to the player, not to the axes. Therefore, it is also necessary to account for the rotation of the camera along the Y-axis using the same reasoning.

```
1  dx = C. sin (P. yaw ) * G. move By
2  dz = C. cos (P. yaw ) * G. move By
3
4  P.x = P.x +- dx
5  P.z = P.z +- dz
```

(code 3.10 - moving the player forward, backward; modified)

3.5.5.6 LookUp/Down (Sub)

Keyboard shortcuts: *r/f*
Rotate the camera according to the *X* axis

The X-axis rotation value (*pitch*) in the range of (-90; 90) degrees expresses the angle of the player's head and limits the possibility of looking up and down. These constraints correspond to the real physical capabilities of human head movement.

```
1  P. pitch = P. pitch +- G. rotate By
2
3  If P. pitch >( <) + -90 Then
4        P. pitch = -+90
5  End If
```

(code 3.11 - rotate camera up, down; modified)

3.5.5.7 LookLeft/Right (Sub)

Keyboard shortcuts: *q/e*
Rotate the camera according to the *Y* axis

In the case of rotation (of the *player's whole body*) along the *Y* axis (*yaw*) in the range (0; 360), which is given in degrees, the program allows the player to rotate around his own axis. If the player exceeds the upper limit of this range, the rotation value is decreased or increased by one period (360°). This ensures that the program can use pre-calculated values of the goniometric functions, which facilitates calculations within the game.

```
1  P. yaw = P. yaw +- G. rotate By
2
3  If P. yaw >=( <=) 360(0) Then
4        P. yaw = P. yaw +- 360 End
5  If
```

(code 3.12 - camera rotation left; right, modified)

4. Program optimization

The last part of the thesis deals with analyzing the program and looking for possible optimizations.

4.1 VBA limitations

When working in a 3D environment, VBA is not the optimal language. Unlike the computer game Minecraft, the Excel environment is not designed to handle large amounts of calculations in a short amount of time. VBA is primarily designed for automation in Microsoft applications [9]. The language is not able to execute multiple parts in the code at the same time (called *multithreading*) and also does not have the ability to use the *GPU*, a graphics processor designed to process large amounts of data [9].

Another problem is the use of static type checks (variables are not bound to a specific data type, but can be changed during the program). This slows down the code significantly due to the need to dynamically adapt to different data types [9].

4.2 Functional complexities

This section deals with the so-called *Landau notation* (*Big O Notation*), which determines the complexity of the calculation depending on the growth of the parameter. The highest order notation is always preferred when evaluating the complexity of each function [10]. This is especially true when the function combines multiple subfunctions that are not interdependent within cycles. Complexity is denoted in this paper by $O(f(x))$.

4.2.1 Init

This is a linear $O(n)$ notation that depends on the number of textures or the number of values of the gonio-metric functions.

4.2.2 CalculatePosition

The function behaves as $O(n)$, where n is the number of cubes. For each cube, 30 positions are computed (6 faces, 4 vertices and a center) and then the faces are ordered by distance using *QuickSort* (Section 3.5.3.4), which is $O(n \log n)$ on average [8].

4.2.3 ApplyTexture

This function also exhibits linear complexity $O(n)$. It traverses all sides and divides them into 64 parts. The coefficient tends to be quite high, in the order of tens, because of all the counting functions, but remains constant depending on the number of sides.

4.2.4 ConvertDraw2D

This function has the largest amount of calculations. For each side, the 2D coordinates of all four vertices are calculated. The program must then determine which cells to set to a given color based on the position of all points lying in the quadrilateral, which is done using *Bresenham's algorithm* (Section 3.5.2.3).

The overall complexity of this function is difficult to determine, and can be as high as $O(n \cdot p^3)$, where n is
is the number of sides and p is the average side length (in cells), which can reach values close to infinity in the vicinity of the player.

4.2.5 Total

Since these functions are executed sequentially and only once each, the overall complexity is governed by the notation of the ConvertDraw2D function, which has the highest time complexity.

The other functions have a time complexity mostly around $O(n)$. When processing a large number of coordinates, which is already the case when computing a single cube (within a reasonable distance from the player), these functions are negligible in terms of time complexity compared to the third-order complexity.

4.3 Optimizing

For a smooth experience, a typical computer game should be able to render one frame per $\frac{1}{60}$ seconds. This is called the number of frames per second (*fps*).

The goal of this section is to reduce the number of calculations so that the program is as close as possible to this value
he approached. This can only be achieved in the case of VBA by finding faster algorithms and omitting calculations of objects that the player cannot see.

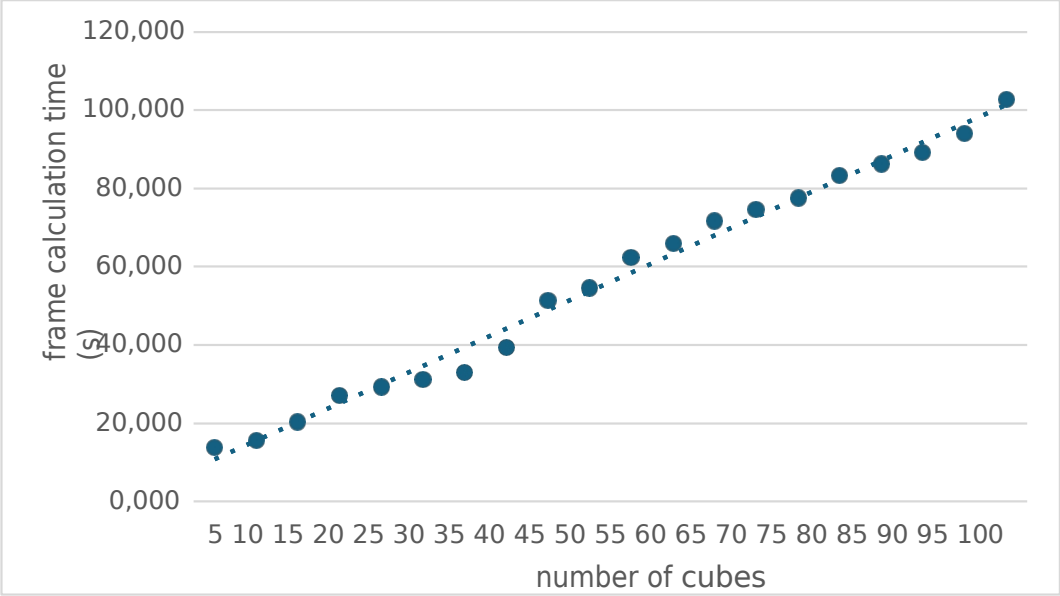
4.3.1 Power dependence

In the first segment of the optimization, I will look in more detail at how the growth of some of the para-meters can affect the overall speed of the program. This analysis is based on measurements from tests that were performed on the overall optimized code. Specifically, this is a program that integrates all of the optimization measures described in Section 4.3.4.

Next, I will look at the relationship between the growth of the number of cubes and the number of rendered cells with the performance of the program, with the goal of understanding how each parameter affects its behavior.

4.3.1.1 Number of cubes

The test investigated the dependence of image rendering time on the increasing number of cubes. The tests were performed on a set from 5 to 100 cubes with a step size of 5 cubes. The cubes were randomly scattered in the vicinity of the player. The graph (Figure 4.1) shows that the dependence is linear, and as the number of cubes increases, the rendering time per cube approaches one second. This means that it takes the program about 1 second to render one cube (Table 4.2).



(Figure 4.1 - dependence of computation time per frame on the total number of rendered cubes (optimized code))

Number of cubes	5	10	15	20	25	30	35	40	45	50
Frame time (s)	13,715	15,616	20,354	27,087	29,257	31,233	33,010	39,448	51,343	54,532
Time/cube (s)	**2,743**	**1,562**	**1,357**	**1,354**	**1,170**	**1,041**	**0,G43**	**0,G86**	**1,141**	**1,0G1**

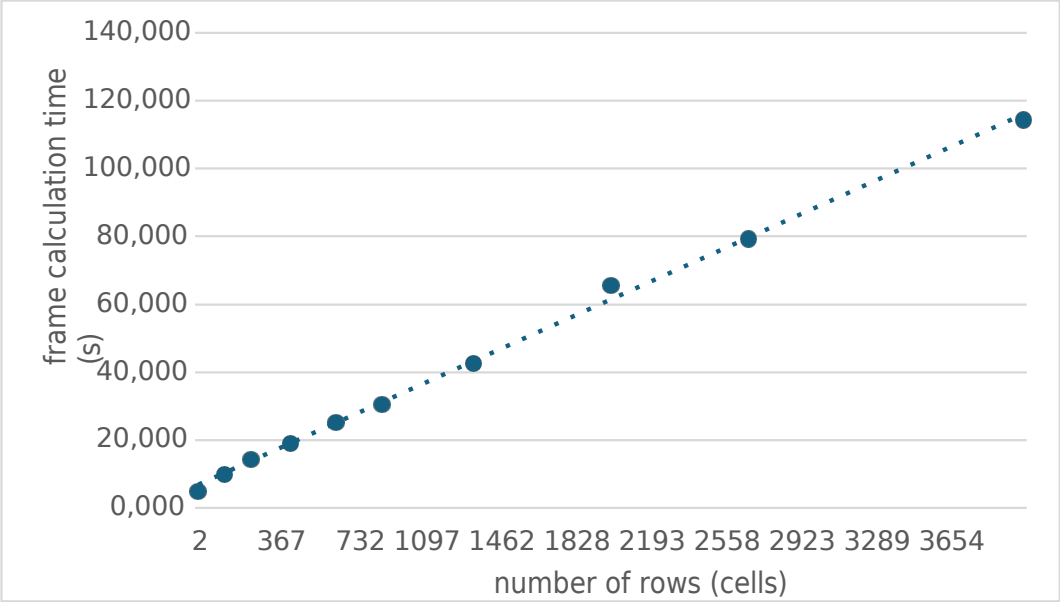
Number of cubes	55	60	65	70	75	80	85	G0	G5	100
Frame time (s)	62,370	65,750	71,733	74,558	77,524	83,243	86,227	89,180	94,052	102,588
Time/cube (s)	**1,134**	**1,0G6**	**1,104**	**1,065**	**1,034**	**1,041**	**1,014**	**0.GG1**	**0,GG0**	**1,026**

(Table 4.2 - computation time per frame as a function of the total number of cubes (optimized code))

4.3.1.2 Number of cells

The test investigated the dependence of image rendering time on the increasing number of cells forming the image. The tests were performed ranging from a line height of 9 to 2160 cells in a sequence of commonly used values from *nHD*(360) to *4KUHD*(2160) with a screen ratio of 16 : 9 [11].

From the results, it can be concluded that the computation time per cell decreases significantly as the number of cells increases. This rule holds even when the time is related to the number of rows (*cells*) (Figure 4.3). This means that the optimized code below actually speeds up the rendering of cells on a row-by-row basis. From the tests, it looks like the limiting value for a row will be close to 0.03 seconds per row (Table 4.4).



(Fig. 4.3 - dependence of computation time per frame on the number of rendered cells (optimized code))

Total number of cells	144	G216	36864	102400	230400	40G600	G21600	2073600	3686400	82G4400
Image width (cell)	16	128	256	427	640	853	1280	1920	2560	3840
Image height (cells	9	72	144	240	360	480	720	1080	1440	2160
Frame time (s)	5,031	9,804	14,196	19,055	25,375	30,391	42,555	65,640	79,297	114,359
Time / 1000 cells (s)	**34,G38**	**1,064**	**0,385**	**0,186**	**0,110**	**0,074**	**0,046**	**0,032**	**0,022**	**0,014**
Time / number of lines (s)	**0,314**	**0,077**	**0,055**	**0,045**	**0,040**	**0,036**	**0,033**	**0,034**	**0,031**	**0,030**

(Table 4.4 - computation time per frame depending on the total number of rendered cells (optimized code))

4.3.2 Test kits

All optimizations were tested against 5 sets of tests based on 10 different measurements. Each set of tests was designed to be temporal in nature and used the same randomly placed cubes across measurements. The average temporal improvements for each optimization are recorded in Table 4.5.

Measurement results may vary depending on the device on which they were taken. For example, measurements of the rendering time of the same frames vary due to the prioritization of the processor, which may be prioritizing other tasks at any given time, the amount of *RAM* available, and more [12]. The measured values are given in the appendix of this thesis and the program read these values over 16 hours.

- Player in the center of one cube, random camera rotation.
- 50 randomly placed cubes near the player. (Doesn't have to be in the player's line of sight, doesn't count towards the average)
- 50 randomly placed cubes near the player without drawing pixels.
- 50 randomly placed cubes near the player without any calculations, just pixel plotting. (Determined as the difference of the previous two measurements)
- View of 27 adjacent cubes with constant position and rotation parameters.

4.3.3 Without optimization

Without any optimizations, the program had a problem with some of the side calculations near the player, and even the program could not calculate these values due to an *overflow* error for some collections. Calculating 50 cubes took the program around **20 minutes** on average, and the average time for all tests was around **8 minutes** (Table 4.5). This time is significantly long and needs to be reduced.

4.3.4 Specific optimization

4.3.4.1 Player's field of view

This optimization significantly reduces the number of computations for all points that the player cannot see on the image, based on his field of view, which in this program is 90° . This leads to a reduction in computations for the number of 2D coordinates near the player. The largest improvement in performance is seen in the *first set of tests*, where the program improved by **68%** on average (Table 4.6). A similar improvement is also seen in the test with *50 randomly placed cubes*, especially when the player is in multiple cubes simultaneously (Table 4.6). Conversely, this optimization does not have a significant effect in the test with *27 adjacent cubes* already in the player's field of view. Similarly, the improvement is not noticeable in the test with *50 cubes without plotting*, because the optimization does not apply to the actual drawing of the cell colors in the workbook (Table 4.6). The implementation of the algorithm is in Section 3.5.2.2.

4.3.4.2 Number of sides of the cube

This is an algorithm that reduces the number of sides displayed for each cube. If the player is looking at a cube and is not inside but outside the cube, he can see a maximum of three different sides (from one point) at any one time. Comparing the original data where all optimizations were tested with the data where this optimization was omitted, a **40%** speedup can be observed when testing the *display of 27 cubes* (Table 4.7). However, compared to the program without any optimizations, this speedup is only **8%** (Table 4.6). *Inside the cube*, it is even faster than when using the overall optimization (Table 4.7), which is due to the deviation mentioned above.

4.3.4.3 Duplicate pages

This optimization eliminates parties that are close to each other. The function removes only one of them for better continuous visualization. This optimization could be improved to discard both sides, since in the case where the cubes are touching, neither of the touching sides can be seen (assuming the cubes are placed in multiples of their sides). The most significant improvement is seen in the test where all *27 cubes* are touching. Compared to the program without optimization, the code is on average **12%** faster (Table 4.6) and **60%** faster compared to the overall optimization (Table 4.7).

4.3.4.4 Rendering by rows

Line-by-line rendering is an optimization where the program colors cells line-by-line according to the current texture color instead of coloring each cell individually. This optimization improves performance, especially in the *50 cube* test set *with plotting*, by **95%** compared to the overall optimization (Table 4.6) and **25%** relative to the program without optimizations (Table 4.6). In addition, there is a reduction in the complexity of plotting cells in the workbook from $O(n^2)$ to $O(n)$.

4.3.5 All optimizations

The combination of all five optimizations results in a significant increase in pro- gram efficiency and a reduction in rendering time of **96%** on average (Table 4.6). The average frame in a given test set takes **15 seconds** instead of the original **491 seconds** (Table 4.5). Despite these improvements, the most challenging process remains the calculation of specific cells based on the vertices of quadrilaterals (Section 4.2.4).

4.3.6 Other options

Other ways to make the program more efficient include optimizing the algorithm for calculating the number of cells in trapezoids. The current algorithm must traverse all cells between all vertices, which can be inefficient, especially for small distances with high

values.

A significant optimization could be to limit the calculation of cube positions and sides that are not visible to the player behind other cubes. It might also be considered to stop showing the textures of the cubes the player is in. Overall, the program can be further optimized, but due to the po

higher rendering speeds close to $\frac{1}{60}$ seconds cannot be achieved in this environment.

⌀ values (time/frame)	Inside one cubes	27 adjacent cubes	50 random cubes	50 cubes, without plotting	50 cubes, only the scope.	Average
Without optimizations (s)	**83,581**	**462,3G1**	**1420,584**	**1144,G63**	**275,621**	**4G1,63G**
Player's field of view (s)	26,709	439,961	701,132	429,356	271,776	**2G1,G51**
Number of sides of the cube (s)	78,*337*	422,852	1369,481	1099,550	269,931	**467,667**
Duplicate pages (s)	78,013	404,532	1384,568	1114,410	270,158	**466,778**
Rendering. by rows (s)	49,766	427,275	1322,370	1117,569	204,801	**44G,853**
Total Optimization (s)	**5,437**	**22,185**	**31,125**	**28.1GG**	**2,G27**	**14,687**

(Table 4.5 - average image rendering time in seconds)

⌀ values (% opt. / without opt.)	Inside one cubes	27 adjacent cubes	50 random cubes	50 cubes, without plotting	50 cubes, only the scope.	Average
Without optimizations (%)	0,000	0,000	0,000	0,000	0,000	**0,000**
Player's field of view (%)	68,044	4,851	50,645	62,500	1,395	**34,1G7**
Number of sides of the cube (%)	6,275	8,551	3,597	3,966	2,065	**5,214**
Duplicate pages (%)	6,663	12,513	2,535	2,668	1,982	**5,G57**
Rendering. by rows (%)	40,458	7,594	6,914	2,393	25,695	**1G,035**
Total Optimisation (%)	**G3,4G5**	**G5,202**	**G7.80G**	**G7,537**	**G8,G38**	**G6,2G3**

(Table 4.6 - ratio of optimizations to program with no optimizations relative to no optimizations in percent)

⌀ values (% of total opt. / opt.)	Inside one cubes	27 adjacent cubes	50 random cubes	50 cubes, without plotting	50 cubes, only the scope.	Average
Without optimizations (%)	0,000	0,000	0,000	0,000	0,000	**0,000**
Player's field of view (%)	90,440	1,091	95,674	96,059	23,886	**52,86G**
Number of sides of the cube (%)	-3,663	43,891	39,093	37,906	48,569	**31,676**
Duplicate pages (%)	2,369	61,657	13,579	7,704	46,433	**2G,541**
Rendering. by rows (%)	83,921	36,824	68,309	-2,938	95,868	**53,41G**
Total Optimisation (%)	**G3,4G5**	**G5,202**	**G7.80G**	**G7,537**	**G8,G38**	**G6,2G3**

(Table 4.7 - ratio of total optimization to specific optimization relative to total optimization in percentage)

4.4 Summary of the programme structure

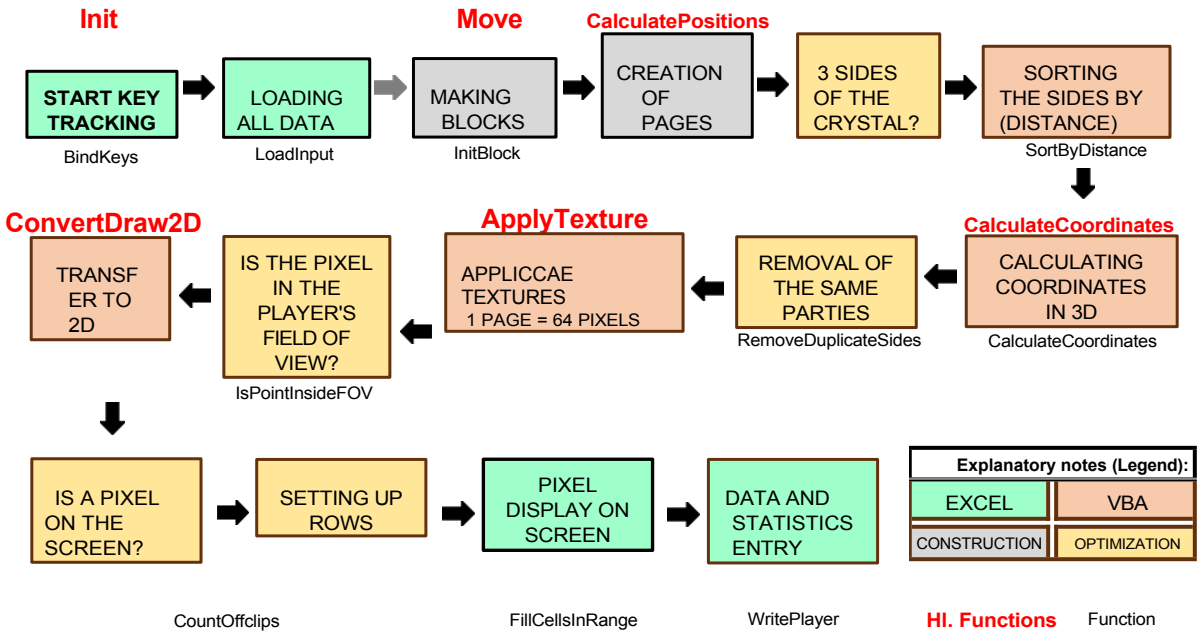
This section focuses on the core structure of the programme.

The frame count can be triggered using Init, which then triggers Move, or by using the keys used to move the player, which also triggers Move. Once all the environment variables and parameters have been loaded, the Block and Side data structures are produced. Then the 3

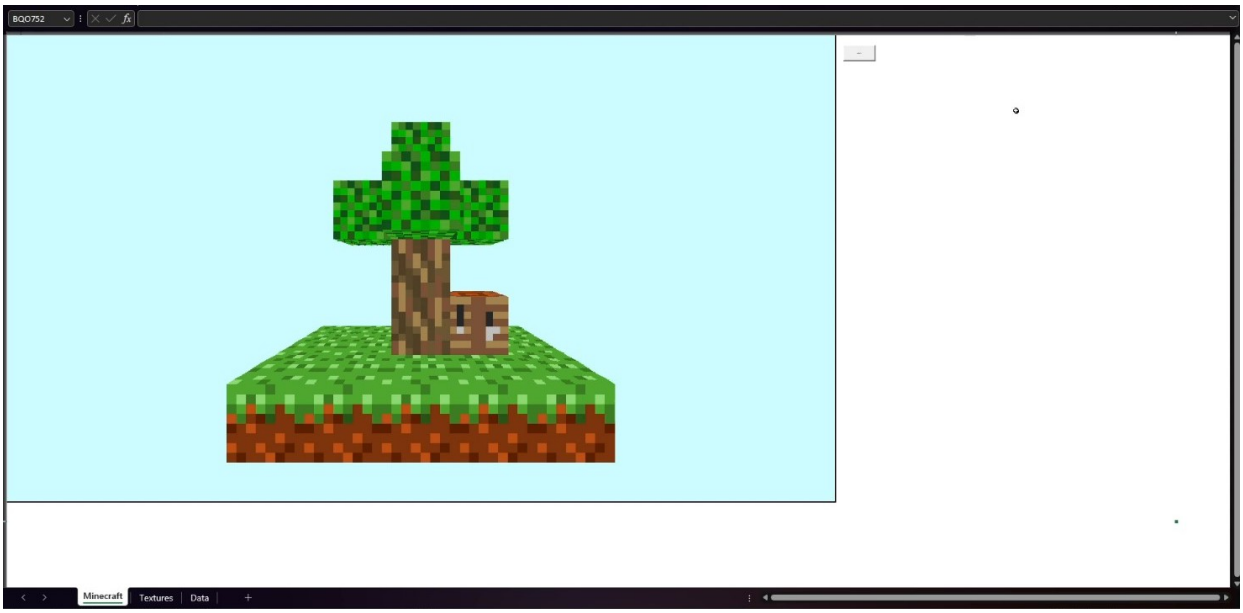
closest sides of each cube are always displayed (if the player is outside the cube) and all sides are ordered by distance from farthest to closest.

Next, calculate all the coordinates of the vertices and the centres of each side. The sides that share the same centre are removed. Each of the sides is divided into 64 smaller sides that already have a specific color based on the texture. They are checked to see if they are in front of the player and in their field of view, and these same colored portions of the sides are converted to 2D.

The last row is converted (based on the four vertices of the given quadrilaterals) to specific cells, these are converted to commands (coloring the rows), and it is checked that all cells lie in the screen. Once all sides are displayed, statistics are entered into the program and the program waits for further input (Figure 4.8, Figure 4.9).



(fig. 4.8 - graph of the basic structure of the whole program)



(Fig. 4.9 - example of drawing an image using the program)

5. Conclusion

The aim of this year's work was to implement a program for rendering 3D cubes using 2D views into cells in Microsoft Excel using Visual Basic for Applications.

The first part of the thesis dealt with the theory of coordinate computation based on camera rotation and player movement. The next part was the actual implementation of the problem, where I described in detail each function behind the program. The last part dealt with the problems of the Ex-cell environment, which prevented satisfactory results regarding the speed of frame calculation. This problem was then solved by implementing algorithms that served to optimize the program and reduce the necessary computations.

In conclusion, Excel is not a suitable environment to implement any 3D-based pro- gram. Overall, the information in this paper can serve as an insight for users into the basics of how three-dimensional environments work. This program has great room for improvement, both in terms of optimizations and user interface.

Literature

[1] Jarmila Robová et al. (2010): *Coordinate system in space*. Department of Didactics of Mathematics, Faculty of Mathematics and Physics, Charles University in Prague, Available from: [https://www.karlin.mff.cuni.cz/~portal/analyticka_geometrie/souradnice.php?chapter=soustavaSouradnicnicP]

[2] Rostislav Horčík (2009): *Perspective projections*. Institute of Computer Science, Academy of Sciences of the Czech Republic, Available from: [https://uivty.cs.cas.cz/~horcik/Teaching/applications/ node4.html]

[3] Martin Tichota (2009): *Perspective as a mathematical model of the lens*. West Bohemian University in Pilsen, Faculty of Applied Sciences, Department of Informatics and Computer Science, 2, Available from: [https://home.zcu.cz/~mikaMM/Galerie%20studentskych%20praci%20MM/2009/Tichota-Perspektiva%20v%20PG.pdf]

[4] Ivanov O et al. (2011): *Applications and Experiences of Quality Control. InTech. Research about New Predictive-Maintenance Methodology using VBA for Marine Engineering Applications*. José A. Orosa, Angel M. Costa and Rafael Santos, University of A Coruña, Spain, 399-410, Available from: [https://cdn.intechopen.com/pdfs/14853.pdf]

[5] Unity Technologies (2023): *Understanding the View Frustum*. Unity Documentation, Available from: [https://docs.unity3d.com/Manual/UnderstandingFrustum.html]

[6] Matúš Lipa, Pavel Rajmic (2019): *Rasterization of a line segment using Bresenham's algorithm - applet*. Department of Telecommunications, FEKT, Brno University of Technology, Available from: [https://www.utko.fekt.vut.cz/~rajmic/applets/Bresenham_line/line.html]

[7] Mark Johnson (2018): *9 quick tips to improve your VBA macro performance*. Microsoft Tech Community, Part 6, Available from: [https://techcommunity.microsoft.com/t5/excel/9-quick-tips-to-improve-your-vba-macro-performance/m-p/173687]

[8] Hoare, C. A. R (1962): *Quicksort*. The Computer Journal, 11-12, Available from: [https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf]

[9] Martin Král (2012): *Excel VBA*. Computer Press, 11-16, 442, 478, Available from: [https://books.google.cz/books?id=khu2DwAAQBAJ]

[10] P. Danziger (2016): *Big O Notation*. 1-6 Available from: [https://www.cs.ryerson.ca/

~mth210/Handouts/PD/bigO.pdf]

[11] Komal Munawar (2021): *Video Aspect Ratios: A Definitive Guide*. Available from: [https:
//motioncue.com/video-aspect-ratios]

[12] Michael Haavertz (2023): *Does CPU Affect FPS*. Kingston College London, Available from: [https://kingstoncollege.org/does-cpu-affect-fps]

Attachments

The source code is freely available at: [https://github.com/ProfiPoint/minecraft-excel

Measured values of tests (measured depending on the total optimization always with the optimization omitted (in seconds); in red wrong or negative):

	Inside one cube	27 adjacent cubes	50 random	cubes50 cubes without plotting	50 cubes. range only.
1	83,421	462,797	3193,812	2512,327	681,485
2	85,039	461,758	945,505	721,516	223,989
3	82,496	461,221	1141,775	915,820	225,955
4	83,691	464,545	912,871	718,750	194,121
5	83,843	462,612	failed; overflow	failed; overflow	-
6	83,211	461,657	1274,027	987,548	286,479
7	82,793	461,680	955,753	751,949	203,804
8	83,687	462,976	2154,002	1885,363	268,639
G	82,981	462,675	1362,336	1152,305	210,031
10	84,652	461,990	845,172	659,086	186,086

(without optimization (Section 4.3.3))

	Inside one cube	27 adjacent	cubes50 random cubes	50 cubes without a range	50 cubic meters. Range only.
1	56,656	22,500	1410,147	1397,542	12,605
2	57,226	22,540	652,652	646,417	6,235
3	56,735	22,359	761,587	759,532	2,055
4	56,367	22,875	failed; overflow	failed; overflow	-
5	57,039	22,305	487,726	491,552	-3,826
6	56,867	22,258	568,121	554,235	13,886
7	56,672	22,367	455,133	462,042	-6,909
8	57,133	22,399	1163,438	1152,523	10,915
G	57,219	22,304	647,773	654,610	-6,837
10	56,805	22,390	328,485	322,004	6,481

(field of le players (section 4.3.4.1))
view

	Inside one cube	27 adjacent cubes	50 random cubes	50 cubes without plotting	50 cubic meters. Range only.
1	5,297	39,547	79,816	65,411	14,405
2	5,230	39,430	47,521	35,606	11,915
3	5,043	39,352	45,598	40,851	4,747
4	5,363	38,945	39,344	35,164	4,180
5	5,293	38,977	51,234	45,817	5,417
6	5,160	40,078	39,282	35,418	3,864
7	5,449	40,797	59,891	57,078	2,813
8	5,330	38,890	43,863	41,153	2,710
G	5,105	39,500	55,637	55,543	0,094
10	5,179	39,876	48,844	42,086	6,758

(number of sides of the cube (Section 4.3.4.2))

	Inside one cube	27 adjacent	cubes50 random cubes	50 cubes with no plotting	50 cubes. range only.
1	5,375	57,617	53,391	45,273	8,118
2	6,211	57,586	28,875	26,375	2,500
3	5,289	58,000	34,446	34,274	0,172
4	5,856	57,938	25,281	21,899	3,382
5	5,486	57,922	41,750	35,453	6,297
6	6,180	58,391	27,758	23,969	3,789
7	5,185	57,625	46,461	38,648	7,813
8	5,732	57,687	29,711	26,484	3,227
G	5,168	57,984	39,250	29,062	10,188
10	5,207	57,843	33,234	24,086	9,148

(duplicate pages (section 4.3.4.3))

	Inside one cube	27 adjacent	cubes50 random cubes	50 cubes without plotting	50 cubic meters. Range only.
1	33,648	35,113	153,578	40,625	112,953
2	33,402	35,082	75,414	22,899	52,515
3	34,453	35,035	92,812	27,789	65,023
4	33,930	35,118	59,055	19,328	39,727
5	34,082	35,082	125,023	31,812	93,211
6	33,664	35,097	75,523	21,109	54,414
7	33,778	35,180	140,414	34,368	106,046
8	33,516	35,035	85,375	22,757	62,618
G	34,025	35,152	106,274	29,039	77,235
10	33,655	35,265	68,672	24,211	44,461

(line-by-line rendering (section 4.3.4.4))

	Inside one cube	27 adjacent	cubes50 random cubes	50 cubes without plotting	50 cubic meters. Range only.
1	5,251	22,769	46,754	42,180	4,574
2	5,494	22,945	25,953	23,922	2,031
3	5,884	22,719	24,723	22,986	1,737
4	5,484	22,889	22,539	20,321	2,218
5	5,748	21,707	40,305	34,313	5,992
6	5,052	21,726	24,372	22,242	2,130
7	5,419	21,769	39,367	35,804	3,563
8	5,204	21,773	25,782	24,250	1,532
G	5,429	21,780	33,250	30,593	2,657
10	5,405	21,772	28,207	25,375	2,832

(all optimizations (section 4.3.5))