SECONDARY SCHOOL VOCATIONAL ACTIVITIES

Area No. 18: Computer science

Rendering Cubes in 3D Spreadsheet Environment

Ondřej Čopák

The capital city of Prague

Capital City of Prague, 2024

SECONDARY SCHOOL VOCATIONAL ACTIVITIES

Area No. 18: Computer science

Rendering Cubes in 3D Spreadsheet Environment

Rendering Cubes in the 3D Environment of a Spreadsheet

Name: Ondřej Čopák
School: Gymnasium Christian Doppler, Zborovská 621/45, Prague
Region: Hlavní město Praha
Consultant: No consultant

Capital City of Prague, 2024

Statement

I declare that I have prepared my SOT work independently and have used only the sources and literature listed in the bibliographic entries. All images are of my own creation.

I declare that the printed version and the electronic version of the competition work are identical.

I have no serious reason against making this work available in accordance with Act No. 121/2000 Coll., on Copyright, on Rights Related to Copyright and on Amendments to Certain Acts (Copyright Act), as amended.

Prague, 26 March 2024 . . . . . .

Ondřej Čopák

Acknowledgements

I would like to thank my colleagues and especially my teacher Bc. Petr Vincen for his help with the formal and professional part of the thesis. Furthermore, I would like to thank the online community of porting the computer game Doom to various game engines, mainly the moderator Wojtek Graj, for his support and consultation regarding the theory of raycasting.

Abstract

The thesis deals with the display of textured cubes in the Microsoft Excel spreadsheet environment. For the purpose of this work I have implemented a VBA program that can render cubes depending on the position and rotation of the player's camera. Furthermore, the thesis focuses on the function of Excel as a game engine and the mathematical part of the 3D-to-cell display using perspective projection. Finally, I discuss the optimization of the program to render individual frames as fast as possible. The result is the ability to render a frame in an acceptable time.

Keywords

Perspective projection; 3D environment; Microsoft Excel; Textures; Optimization

Abstract

The thesis is about displaying textured cubes in the spreadsheet environment of Micro- soft Excel. For this purpose, I have developed a VBA program capable of rendering cubes according to the position and rotation of the player's camera. Additionally, this thesis focuses on the function of Excel as a game engine and the mathematical aspects of pro- jecting 3D cubes onto cells using perspective projection. In conclusion, emphasis is placed on optimizing the program to achieve faster rendering of individual frames. The result is the ability to render a frame in an acceptable time.

Keywords

Perspective projection, 3D environment; Microsoft Excel; Textures; Optimization

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[1], to different environments. Thus, I decided to implement the basics of graphical cube rendering, inspired by the computer game Minecraft[2], into the Microsoft Excel[3] (hereafter referred to as *Excel*) environment.

Emphasis is also placed on explaining how Excel can be perceived as an environment for 3D game design. Camera rotation, player movement, conversion to 2D and other aspects related to visualizations are analyzed. A large part of this is also optimizing the code to minimize the number of calculations in order to speed up the program. A basic understanding of how computer graphics works is a prerequisite for a good understanding of the work.

The whole thesis is divided into three main 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 second part focuses on the concrete implementation in the Excel environment as well as the user interaction with the program, and the last part of the paper deals with limiting environmental factors and further code optimizations. The thesis then concludes (and discusses) by discussing various possible improvements to the program and its specific uses.

Throughout the work, I have set 5 points to guide the program

- **Readability** - The code should be as clear, explained and commented as possible.
- **Speed** - The resulting rendering should be as fast as possible.
- **Controls** - Controlling the camera, player and settings should be as simple as possible.
- **All in VBA** - The entire program must be written in VBA (in Excel).
- **Easy to modify** - It should be easy for other developers to understand how to add new textures, objects, make optimizations and other modifications.

[1] Doom (1993, id Software) [2] Minecraft (2009, Mojang Studios)

3Microsoft Excel (2016, Microsoft)

2 Theoretical 3D BASE environment

The chapter is devoted to the theoretical basis of mathematical operations - movement, rotation in 3D environment and subsequent conversion to 2D in linear representation. 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 the points [1].

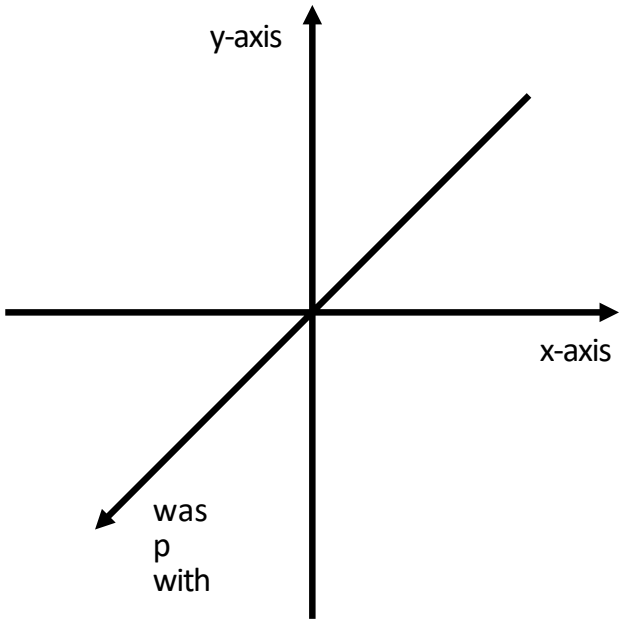


Figure 2.1: right-hand axis *X*, *Y*, *Z*

2.1 Shift in 3D

To minimize the complexity of the calculations, it is convenient to place the player at the origin 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 location. Thus, instead of moving the player, the world will move (by opposite

values of x, y, z)

2.2 ROTATION IN 3D

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*. In the original game, the head cannot be tilted towards 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 the axes where the values change during a particular rotation are shown.

2.2.1 Deriving the formula for rotation in 2D

To accurately position and orient the objects, 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).

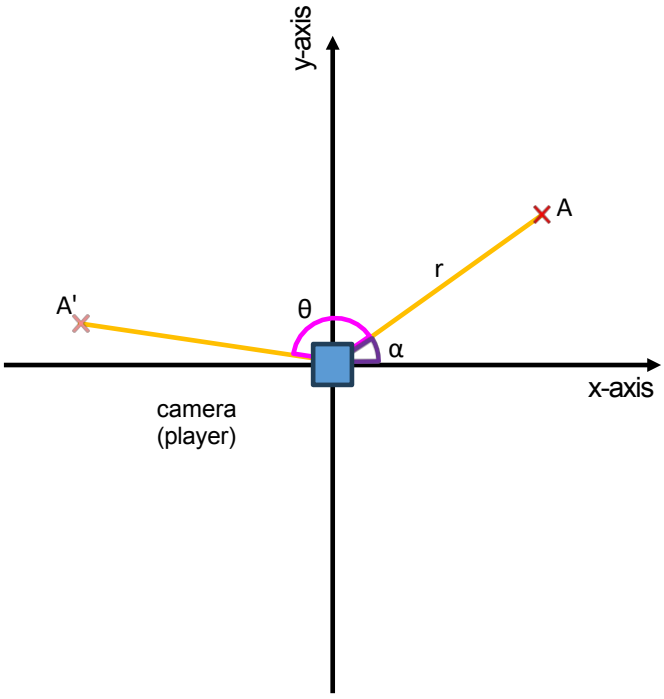


Figure 2.2: rotation of point A by angle θ in the Cartesian coordinate system of the XY plane

A: a point defined by the coordinates $[x, y]$, which is rotated.
A': result point after rotation.
θ: angle of rotation (yaw, pitch).
α: the original angle between the X-axis and the point.
β: total rotation angle $(\alpha + \theta)$.

r: the distance from the origin to a point in the 2D plane.

For a better understanding, we need to derive the formula for 2D rotation based on (Figure 2.2) The angle α is given by the arcsine of the ratio y/x and is also given by the quadrant in which the point is located.

$$\alpha = \arctan\left(\frac{y}{x}\right)$$

From this, the total angle of rotation of the 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 $x = 0$) 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 $\tan(\alpha + \theta)$.

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

The value of $\tan(\arctan(\frac{y}{x}))$ equals $\frac{y}{x}$ and β equals $\alpha + \theta$.

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

Next, the fraction is expanded by the expression $x \cdot \cos(\theta)$.

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

Consequently, β can be expressed from this equation.

$$\beta = \arctan\left(\frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}\right)$$

The following is the substitution of β from the previous step into the formulas.

$$x' = r \cdot \cos\left(\arctan\left(\frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}\right)\right)$$

$$y' = r \cdot \sin\left(\arctan\left(\frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}\right)\right)$$

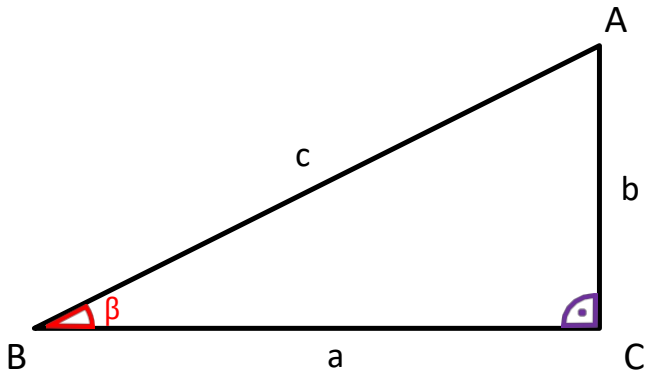


Figure 2.3: right triangle with angle β at vertex B

To complete the adjustments, the values cos(arctan($\frac{y \cdot \cos(\theta)+x \cdot \sin(\theta)}{x \cdot \cos(\theta)-y \cdot \sin(\theta)}$)) a sin(arctan($\frac{y \cdot \cos(\theta)+x \cdot \sin(\theta)}{x \cdot \cos(\theta)-y \cdot \sin(\theta)}$))
From previous calculations:

$$\beta = \arctan\left(\frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}\right)$$

And at the same time, for a general angle θ must hold (Figure 2.3):

$$\tan(\beta) = \frac{b}{a}$$

This can be converted to:

$$\tan(\beta) = \tan\left(\arctan\left(\frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}\right)\right) = \frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}$$

Both equations imply that

$$\frac{b}{a} = \frac{y \cdot \cos(\theta) + x \cdot \sin(\theta)}{x \cdot \cos(\theta) - y \cdot \sin(\theta)}$$

We can express both sides of a, b from this ratio.

$$b = y \cdot \cos(\theta) + x \cdot \sin(\theta)$$

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

The functions $\sin(\beta)$ and $\cos(\beta)$ require the side c to be calculated. That side can be expressed using the Pythagorean theorem ~~from~~ (Figure 2.3) as $c = \sqrt{a^2 + b^2}$.

$$\sin(\beta) = \frac{b}{c} = \frac{b}{\sqrt{a^2 + b^2}}$$

$$\cos(\beta) = \frac{a}{c} = \frac{a}{\sqrt{a^2 + b^2}}$$

After substituting after a, b and into the original system of equations, this is modified to:

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

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

Multiplying brackets.

$$x' = r - \frac{x - \cos(\theta) - y - \sin(\theta)}{\sqrt{y^2 \cos(\theta)^2 + xy \cos(\theta) \sin(\theta) + x^2 \sin(\theta)^2 + xx^2 \cos(\theta)^2 - xy \cos(\theta) \sin(\theta) + y^2 \sin(\theta)^2}}$$

$$y' = r - \frac{y - \cos(\theta) + x - \sin(\theta)}{\sqrt{y^2 \cos(\theta)^2 + xy \cos(\theta) \sin(\theta) + x^2 \sin(\theta)^2 + xx^2 \cos(\theta)^2 - xy \cos(\theta) \sin(\theta) + y^2 \sin(\theta)^2}}$$

Subtract $x - y \cos(\theta) \sin(\theta)$.

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

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

Printing x^2 and y^2 .

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

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

Adjustment by $\sin(\theta)^2 + \cos(\theta)^2 = 1$.

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

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

Expressing r as the distance of a point from the origin of the coordinates.

$$x' = \sqrt{x^2 + y^2} - \frac{x - \cos(\theta) - y - \sin(\theta)}{\sqrt{x^2 + y^2}}$$

$$y' = \sqrt{x2} + y2 - \frac{y\_\cos(\theta) + x\ \sin(\theta)}{\sqrt{x2 + y2}}$$

-

Shortening and finishing to final shape:

$$'x = x - \cos(\Theta) - y - \sin(\Theta)$$

$$'y = x - \sin(\Theta) + y - \cos(\Theta)$$

These formulas will be used in both rotations.

2.2.2 Rotation about the Y axis (yaw)

For the rotation of the XZ plane (rotation of the camera to the left and right) we can use the already calculated formula, where Y remains the same and X and Z change depending on the *yaw* angle (Figure 2.4). I use the cyclic substitution formula from Section 2.2.1.

Θ is the *yaw* angle.

$$'x = x - \cos(\Theta) - z - \sin(\Theta)$$
$$'y = y$$
$$'z = x - \sin(\Theta) + z - \cos(\Theta)$$

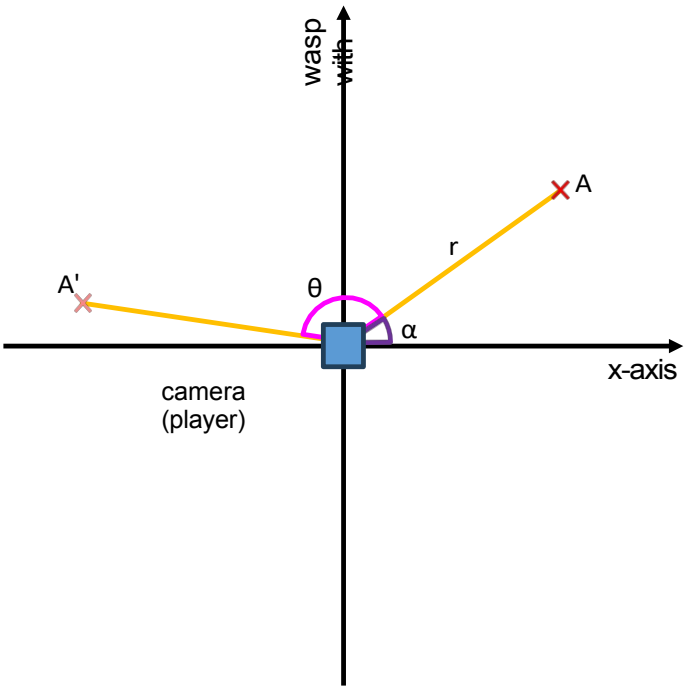


Fig. 2.4: rotation of point A by angle Θ (*yaw*) in the case of a camera rotating around axis Y

2.2.3 Rotation about the X axis (pitch)

For the rotation of the YZ plane (rotation of the camera up and down) I will again use the derived formula. This time, the cyclic substitution leaves the same coordinate X, where the values of Y and Z again change depending on the *pitch* angle (Figure 2.5).

θ corresponds to the *pitch* angle.

$$
\begin{aligned}
x' &= x \\
y' &= y \cdot \cos(\theta) - z \cdot \sin(\theta) \\
z' &= y \cdot \sin(\theta) + z \cdot \cos(\theta)
\end{aligned}
$$

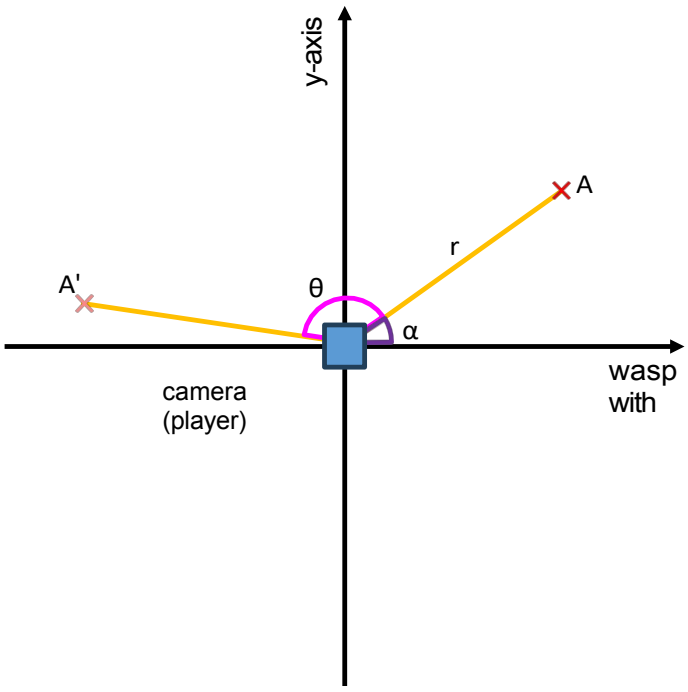


Figure 2.5: rotation of point A by angle θ (*pitch*) in case of camera rotation around axis X

2.3 Transformation MATRIX IN 3D

In the program (code 2.3), the coordinates of individual points are calculated using the equations derived in the previous sections (Sections 2.1, 2.2), because the VBA language is not optimized for working with matrices (Section 4.1).

```
1  ' Translate the point relative to the player's position
2  trans Vertex (0) = point (0) - P.x
3  trans Vertex (1) = point (1) - P.y
4  trans Vertex (2) = point (2) - P.z
5
6  ' Rotate the point around the player's yaw ( axe Y)
7  rotVertex Y (0) = trans Vertex (0) * c. cos (P. yaw ) - trans Vertex (2) * c. sin (P. yaw )
8  rotVertex Y (1) = trans Vertex (1)
9  rotVertex Y (2) = trans Vertex (0) * c. sin (P. yaw ) + trans Vertex (2) * c. cos (P. yaw )
10
11 ' Rotate the point around the player's pitch ( axis X) rotVertex
12 P (0) = rotVertex Y (0)
13 rotVertex P (1) = rotVertex Y (1) * c. cos (P. pitch ) - rotVertex Y (2) * c. sin (P. pitch
14 )
   rotVertex P (2) = rotVertex Y (1) * c. sin (P. pitch ) + rotVertex Y (2) * c. cos (P. pitch )
```

Code 2.1: perspective view of points; variable names modified

Camera movement and rotation can be combined and expressed using the so-called Transform Matrix. Therefore, these derived equations can be transformed into matrices. The resulting position of a point after applying both displacement and both rotations will correspond to the product of these matrices, because rotation about two different axes can be converted by stacking the rotations to rotation first about the first axis and then about the second axis.

The use of matrices can greatly speed up calculations because all trans-formation matrices are first multiplied with rotation matrices and then this one matrix is applied to all points.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} c\text{-}\cos(P.\text{yaw}) & 0 & -c\text{-}\sin(P.\text{yaw}) \\ 0 & 1 & 0 \\ c\text{-}\sin(P.\text{yaw}) & 0 & c\text{-}\cos(P.\text{yaw}) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\text{-}\cos(P.\text{pitch}) & -c\text{-}\sin(P.\text{pitch}) \\ 0 & c\text{-}\sin(P.\text{pitch}) & c\text{-}\cos(P.\text{pitch}) \end{bmatrix} \begin{bmatrix} x - P.x \\ y - P.y \\ z - P.z \end{bmatrix}$$

(3D point transformation matrix for yaw, pitch and translation)

The sequence of rotations and translations is unmistakable. This follows from the property of non-commutativity of ma- tic. Otherwise, different coordinates would be calculated.

2.4 View TO THE 2D plane

The principle of *perspective projection is* used to convert 3D space into a 2D plane. This ensures in-depth visualization of spatial objects on the screen surface [2].

The perspective projection works with the so-called normalized coordinates, the principle of which is explained in the previous part of the thesis (see Section 2.1 and 2.2). The normalized coordinates x, y and z, obtained by looking at the player in the direction of the z-axis, can then be converted into the corresponding 2D coordinates of the screen.

$$X_{2D} = \frac{X3D}{Z3D}, \quad Y_{2D} = \frac{Y3D}{Z3D}$$

These formulas exploit the similarity of the triangles (*red side* with camera and *orange side* with camera), where the *orange side* was created as a line between points that are 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 2.6).

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 coordinate distance z. If the cube passes through the screen, the point is shifted 1 unit in the direction along the z-axis so that there is no division by zero problem and the user can see that side on the screen (code 2.4; line 2).

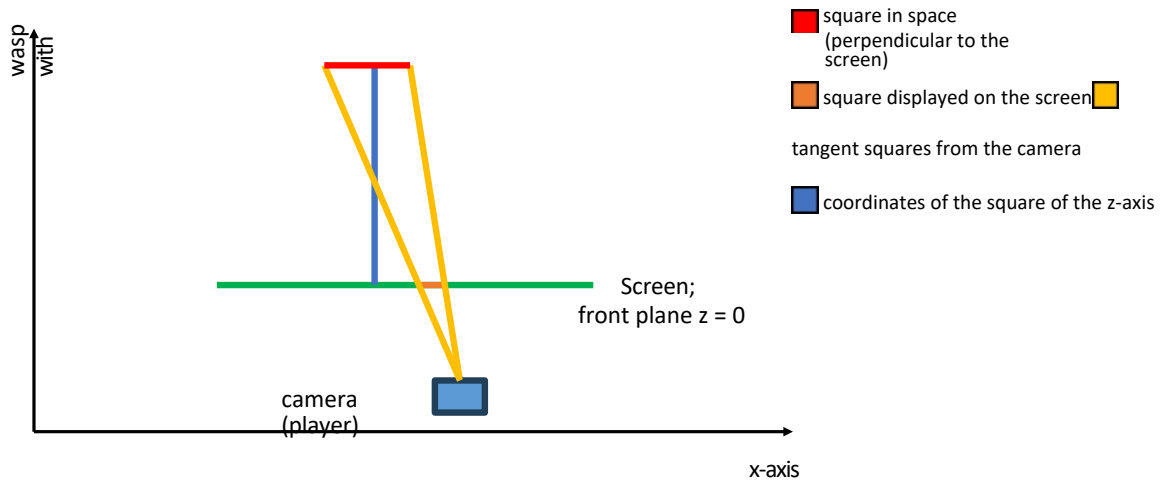


Fig. 2.6: perspective view of squares on the screen, top view - converted to 2D for better orientation

```
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 2.2: 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 discuss in 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 range of users. The source code along with with comments is available in the appendix of the thesis.

3.1 Environme EXCEL nt

In Excel, cells act as pixels that are organized into a table that acts as a screen, where they form 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 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 a 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 (control PROGRAM)

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 screen settings, and also has the possibility to see various statistics, when and how many calculations the program has performed for 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).

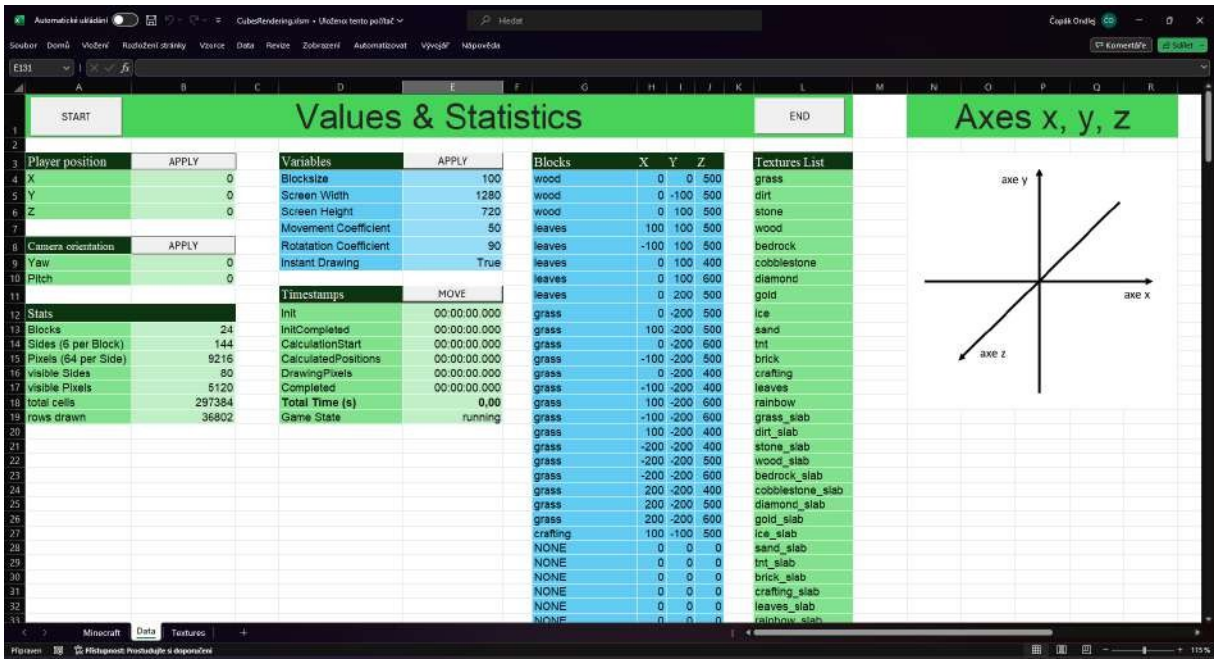


Fig. 3.1: user interface in Excel environment

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 a button to exit the game.

3.3.2 Player position

Here the user can set the current position of the player in the coordinates *x*, *y*, *z*. After executing the movement function here, the program automatically updates the values of

the current player positions.

3.3.3 Camera orientation

Similar to the player position section, here the user sets the camera rotation angles. The program also updates the current camera values after the motion 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 texture color 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* represent 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 computationally intensive. At the same time, all cells will be colored without continuously rendering the program, thus significantly speeding 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. The textures themselves are then on the 3rd *Textures* sheet (Figure 3.2).

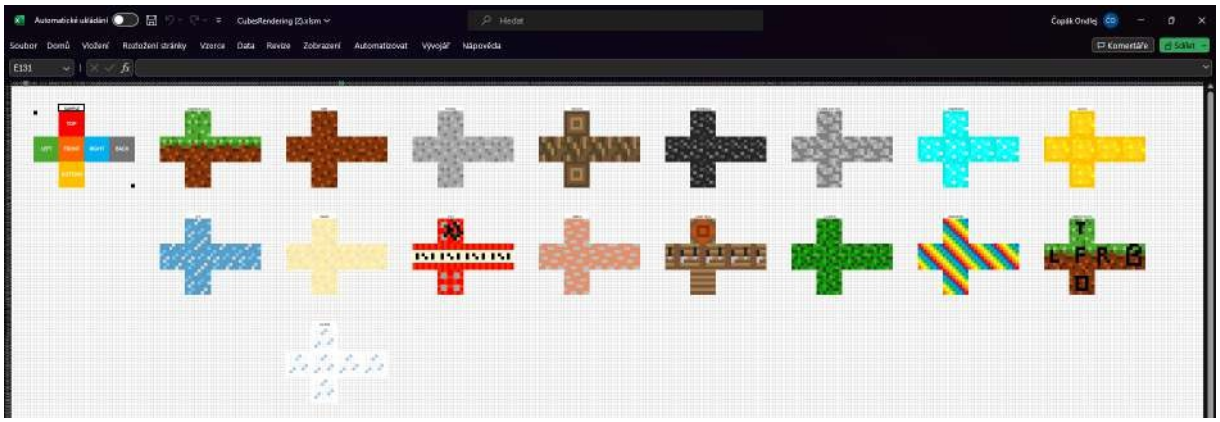


Figure 3.2: list of cube textures in Excel workbook 3

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 row, the user sees the time when the program was started, and the following rows 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 Class (CLASSES) es

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 the game environment. 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*).

Using this class makes the code clearer because instead of using 5 different variables,

variables of one class are used. In addition, using 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 3.3.5. 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 various profiles of game settings or other settings of the environment.

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 outside the field of view of the player. As with the Player and Game classes, this class is used to make the code cleaner.

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 specifies the texture of the side (Texture class), *MiddlePoint*, which is the position of the center of the side, and *Orientation*, which indicates the orientation of the side to (*Top/Bottom/Left/Right/Front/Back*). It also contains the vertices of the side, 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 variable values in this procedure are 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 the declared value. *Sub* procedures are often used to interact with Excel, i.e. work with cells.

In this paper, the term *function* includes both procedures (functions and subprocesses),

and I will focus on selected functions that are most important for understanding the basic operation of the program. The thesis discusses optimization algorithms in more detail 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 the input values and create a 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. Then the functions for calculating the current frame and creating 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 camera rotation 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 sides - by the four vertices *a, b, c, d,* which arise as$^{k}_{8}$ (for k 1 to 8) times

vectors →-*ab* and $\overrightarrow{ad}$. (Figure 3.3), code 3.5.1.4; line 5). Subsequently, the texture is applied according to
orientation (code 3.5.1.4; line 16) and check that all vertices are in
plane in front of the player. If so, the Pixel will be saved.

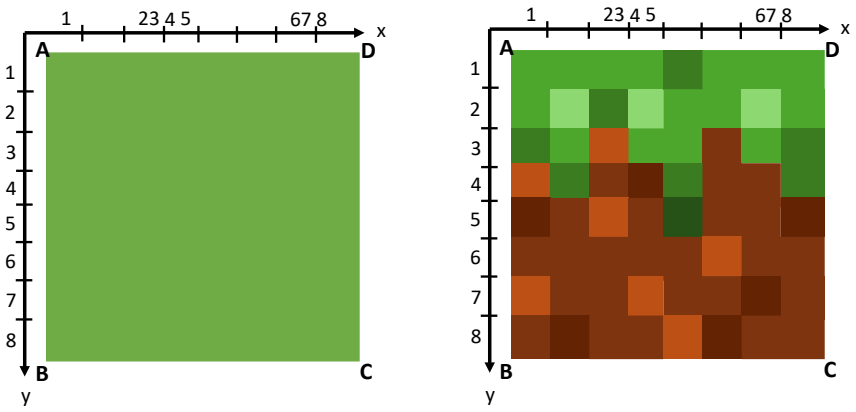


Fig. 3.3: visualization of texture application - 1x Side on the left, 64x Pixel on the right

```
For Each s 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 = " top " Then
                col = texture Color(y + 1 + 18)(8 - x)
            Else If s. orientation = " bottom " 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
                AllSquares . Add Pixel. Initialize a3 , b3 , c3 , d3 , col
            End If
        Next y
    Next x
Next with
```

Code 3.1: application of texture to 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 2.4. 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 $z = 1$, so that the player will see the object. In the last part of the loop, these points are scaled according to the screen size setting (code 3.5.1.5; 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  ' Converts all 4 vertixes from 3 D to 2 D
2  For Each intersection Point In Array ( Side .a3 , Side .b3 , Side .c3 , Side . d3 )
3
4      ' Projects 3 D points to 2 D points
5      If Int( intersection Point (2)) = 0 Then
6          intersection Point = Array ( intersection Point (0), intersection Point (1), 1)
7      End If
8
9      ' Applies Perspective Projection
10     projected X = intersection Point (0) / intersection Point (2)
11     projected Y = intersection Point (1) / intersection Point (2)
12
13     ' Multiplies the projected points by the screen size
14     If G. screen Height < G. screen Width Then
15         screen X = Int(( projected X + 1) * 0.5 * G. screen Height + (G. screen Width - G.
               screen Height) / 2)
16         screen Y = Int ((1 - projected Y ) * 0.5 * G. screen Height)
17     Else
18         screen X = Int(( projected X + 1) * 0.5 * G. screen Width )
19         screen Y = Int ((1 - projected Y ) * 0.5 * G. screen Width + (G. screen Height - G.
               screen Width ) / 2)
20     End If
21
22 Next intersection Point
```

Code 3.2: coordinate conversion from 3D to 2D; modified

3.5.2 Geometry

The Geometry module contains all functions that deal with coordinate calculation.

3.5.2.1 CalculateCoordinates (Function)

This function is an implementation of the algorithm already mentioned in Sections 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 axes X and Y does not matter.

3.5.2.2 IsPointInsideFOV (Function)

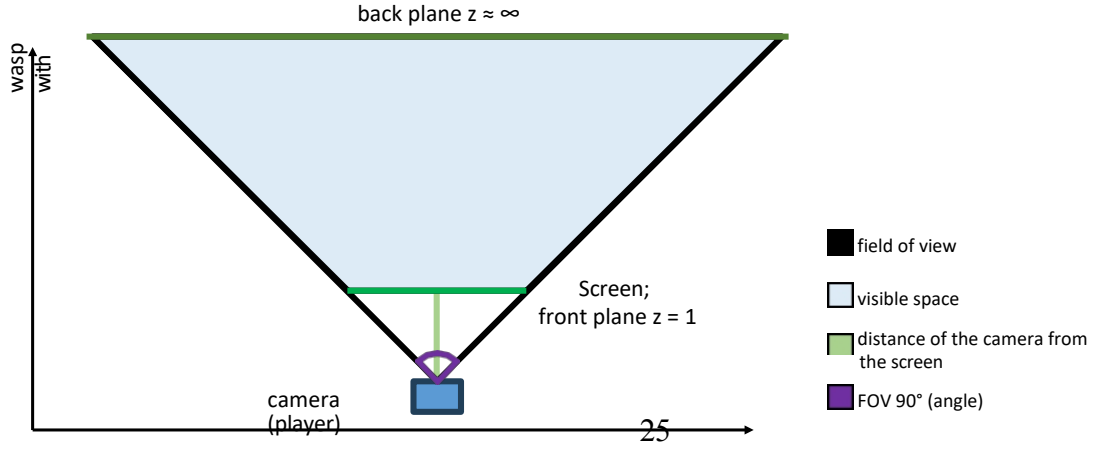
The purpose of the procedure is twofold. To verify that the point is in the plane in front of the player and also to verify that the point is in the player's field of view. The first condition is satisfied if $z > 0$.

The second condition is based on the mathematical solution of the 2D conversion in Section 2.4. The player's field of view resembles a comoving tetrahedral needle, where the upper base lies in the plane of the player's screen (plane $z = 0$) and the lower base runs towards the infinite coordinate z (Figure 3.4) [5].

In each plane $z = k$, $k \in (0, \infty)$, the intersection of a given k-th plane with the field of view is a rectangle,

so we just need to 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 coordinate z. The larger the coordinate z, the

the bigger the rectangle gets. The scale is calculated using the goniometric function tangent (code 3.5.2.2). 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 3.5.2.2; line 2). If both conditions are met, the function returns true.



x-axis

Figure 3.4: 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 3.3: 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*. This 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 traverses the points in between and selects the appropriate pixels (Figure 3.5) [6].

Furthermore, as part of the program optimization, the lowest and highest values of x are stored in the dictionary according to the column key y. This process is performed for each point of the quadrilateral. The result is a set of pixels in the form of a square, where each y contains points from the lowest x to the highest

x. This is used when plotting cells row by row in Excel, where setting the colour of one cells takes as long as setting up multiple cells in a rectangular range [7].

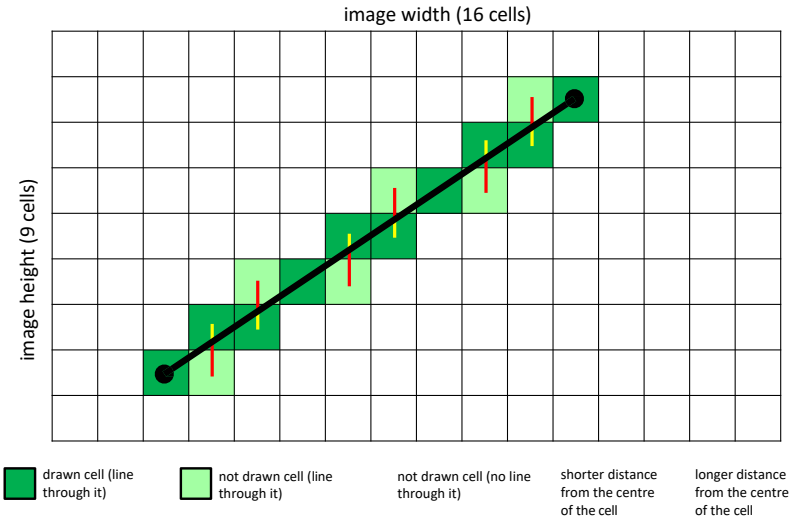


Figure 3.5: Bresenham's algorithm for line-based cell retrieval

3.5.3 Functions

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

3.5.3.1 RemoveDuplicateSides (Sub)

This optimization function removes all sides that have the same center. If 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. Whenever the program finds a center that already exists it is stored in a new sheet and the pages with those centers are discarded.

3.5.3.2 ReverseCollection (Function)

This is one of the basic functions that had to be manually implemented. This function returns a new list 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. Pro- cedura uses its own 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 ones 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

concerned with re-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, the default blue, which represents the sky.

3.5.4.2 FillCellsInRange (Sub)

This function receives as input the coordinates of the top left corner, bottom right corner and color for the given cell range. This function then verifies that all cells are located 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 time frame starts.

3.5.5 Keys (Controls)

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 variables *moveBy* and *rotateBy* determine the value of 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 will set up 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 keyboard shortcuts used to control the game.

3.5.5.3 MoveTop/Bottom (Sub)

Keyboard shortcuts: *space/x*
Movement along the *Y* axis

Unlike the movement along the *X* or *Z* axes, 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 ' for Top
2  P.y = P.y - G. move By ' for Bottom
```

Code 3.4: player movement up, down; modified

3.5.5.4 MoveLeft/Right (Sub)

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

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

```
1   dx = C . sin (P. yaw ) * G. move By
2   dz = C . cos (P. yaw ) * G. move By
3
4   ' for Left -; +
5   ' for Right +; -
6  P.x = P.x -+ dz
7  P.z = P.z +- dx
```

Code 3.5: player movement left, right; modified

3.5.5.5 MoveFront/Back (Sub)

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

As with the left and right movement, the up and down movement again works with coordinates relative to the player, not the axes. Therefore, it is also necessary to account for camera rotation 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  ' for Front +; +
5  ' for Back -; -
6  P.x = P.x +- dx
7  P.z = P.z +- dz
```

Code 3.6: player movement forward, backward; modified

3.5.5.6 LookTop/Bottom (Sub)

Keyboard shortcuts: *r/f*
Camera rotation according to *X* axis

The rotation value along the *X* (*pitch*) axis in the range (-90; 90) degrees expresses the angle of rotation
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  ' for Top >; + ; -
4  ' for Bottom <; -; +
5  If P. pitch >( <) + -90 Then
6      P. pitch = -+90
7  End If
```

Code 3.7: rotate camera up, down; modified

3.5.5.7 LookLeft/Right (Sub)

Keyboard shortcuts: *q/e*
Camera rotation according to *Y* axis

In the case of rotation (of the *player's entire body*) along the *Y* (*yaw*) axis in the range (0; 360), which is given in degrees, the program allows the player to rotate about 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
goniometric functions, which facilitates calculations within the game.

```
1  P. yaw  = P. yaw +- G. rotate By
2
3  '  for   Left >=; 360; +
4  '  for   Right <=; 0; -
5  If P. yaw >=( <=) 360(0) Then
6        P.yaw = P. yaw +- 360
7   End If
```

Code 3.8: rotate camera left, right; modified

3.6 OPTIONS cubes

The program works with quadrilaterals as basic elements. All shapes, especially cubes, are created based on the four vertices, which are expressed analytically.

3.6.1 Cube

The cube is the basic object for which the program was mainly created. Each cube has 6 sides and each cube is 8x8 pixels.

3.6.2 Halfway

Instead of a cube, the resulting solid is a cube that is exactly half the height. A normal texture (the same as for a cube) can be applied to this object. The top and bottom walls remain the same as for the cube, and the bottom portions of the 8x4 texture are applied to the side walls.

3.6.3 Adding other shapes

The program is designed to make it easy for users to add their own solids. The first thing to do is to add construction methods to the InitBlock (Main) function that define the centers of the walls and their orientation relative to the center of the solid so that textures can be applied later. Next, in the initialization function (Side), the user must define the position of the quadrilateral vertices relative to a given point. Here, for example, it is possible to create triangles instead of squares, which are used to compose all objects in conventional 3D engines, if the user sets the same coordinates for two different points (thus reducing the number of vertices from four to three).

The problem with using triangles would be the subsequent application of textures, as the texture would be more intense at the connected vertex than on the opposite side. This problem would need to be solved in the last section used to add new shapes (*ApplyTexture* function in Main). Here the user specifies how the texture will be applied (and which parts).

3.6.4 Transparency

It is not possible to set the transparency of the cell fill color in Microsoft Excel. Instead, the user can enter a decimal number from 0 to 1 into the texture cells, which represents the intensity of *opacity of* the color in the texture (0 = completely transparent, 1 = completely opaque). The number detection is done in the Textures class (*LoadInput*). If the cell contains no text, the *opacity is* set to 1.

During normal rendering, individual cells are redrawn. In the case of a transparent texture, it first determines what color is in the cell and renders the new color in proportion to the transparency of the new color. This process is incompatible with row-by-row rendering because the newly created colors may not be the same (if the cell colors are different before the transparent texture is rendered), so transparent textures must always be rendered cell by cell (code 3.6.4).

```
1  ' Get the current RGB color
2  originalCol = originalCell. Interior. Col
3  ' Calculate the new color
4  new Color = RGB (
5        Int ((( opacity ) * ( originalCol % 256) + opacity * ( colTexture % 256)) + 0.5),
6        Int ((( opacity ) * (( originalCol \ 256) % 256) + opacity * (( colTexture \ 256) %
           256)) + 0.5),
7        Int ((( opacity ) * ( originalCol \ 65536) + opacity * ( colorTexture \ 65536)) + 0.5)
8  )
9  originalCell. Interior. Color = new Col
```

Code 3.9: transparent cell application; 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 significantly slows down code due to the need to dynamically adapt to different data types [9].

4.2 COMPOSITIONS Functions

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. In this paper, the complexity is denoted by $O(f(x))$.

4.2.1 Init

This is a linear notation $O(n)$, which 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 goes through all the sides and divides them into 64 parts. The coefficient tends to be quite high, in the order of tens, because of all the numerical 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 $O(n - p^3)$, where n is the number of sides and p is the average side length (in cells), which can reach values approaching 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 (at a distance from the player of 1 to 30 blocks with random rotation with respect to the camera), these functions are negligible in terms of time complexity compared to the complexity of the third level.

4.3 IMPLEMENTATI BY optimizing ON

To run smoothly, a computer game should be able to render one frame per $\frac{1}{60}$ second (*fps)*. The goal is to reduce the number of computations in order to speed up. This can be achieved in VBA by finding more efficient algorithms and omitting 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 parame- trs 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 optimizations (Section 4.3.4). Next, I will focus on the relationship between the growth of the number of cubes and the number of rendered cells with the performance of the program.

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 distributed 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 4.1).

Figure 4.1: dependence of the 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 calculation 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 calculation 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 4.1: 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. 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 4*KUHD*(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 4.2). 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 4.2).

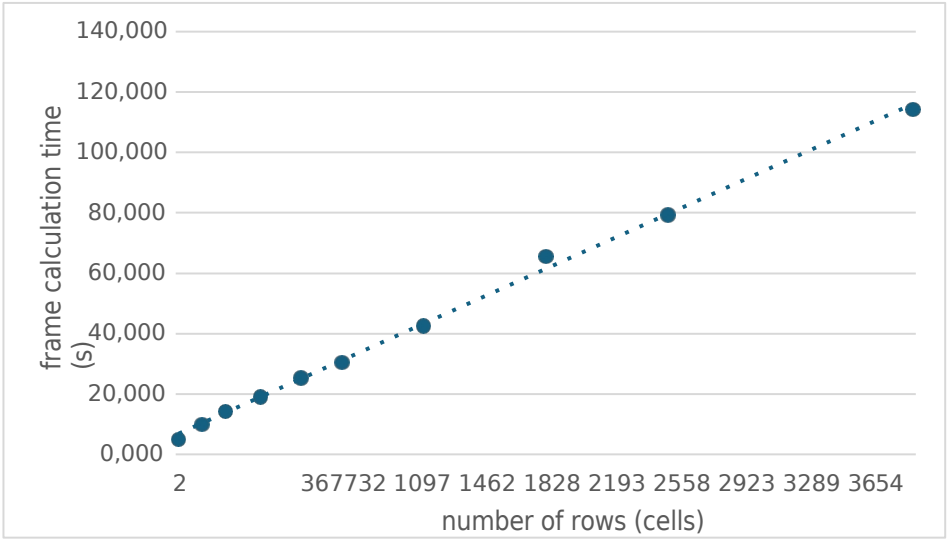


Figure 4.2: dependence of the computation time per frame on the number of rendered cells (optimized code)

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 calculation 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]	34,938	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 4.2: 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 4.3.

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]. If the program was unable to render a frame due to timing reasons, a *dash* is shown in the table instead. The measured values are given in the appendix of this paper 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 in the vicinity of the player without any calculations, only pixel scaling. (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 4.3). 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 biggest improvement
performance is evident in the *first set of tests*, where the programme improved by **68%** on average (Table 4.4). A similar improvement is also visible in the test with *50 randomly placed cubes*, especially when the player is in multiple cubes simultaneously (Table 4.4).

This optimization does not have a significant effect when tested 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 4.4). 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 a single 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 4.5). However, compared to the program without any optimizations, this speedup is only **8%** (Table 4.4). *The inside of the cube* is do- end faster than when using the overall optimization (Table 4.5), 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 discards both sides, because 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 greatest improvement is seen in the test where all *27 cubes* are touching. Compared to the

program without optimization, the code is on average **42%** faster **(**Table 4.4) and **60%** faster compared to the overall optimization (Table 4.5).

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. It takes the same amount of time to color a single cell as it does to color multiple cells (a rectangle of cells) with the same color. Therefore, this optimization improves performance, especially in the *50 cube* test set *with plotting*, by **95% compared to the** overall optimization (Table 4.4) and **25%** relative to the program without optimizations (Table 4.4). 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 4.4). The average frame in a given test set takes **15 seconds** instead of the original **491 seconds** (Table 4.3). 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 at small distances with high ranks.

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 could be further optimized, but
due to the nature of the environment, higher rendering speeds close to $\frac{1}{60}$ seconds cannot be achieved.

average values [time / frame]	Inside one cubes	27 adjacent cubes	50 random cubes	50 kr. without Plot	50 crores. 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 4.3: average image rendering time in seconds

average values [time / frame]	Inside one cubes	27 adjacent cubes	50 random cubes	50 kr. without Plot	50 kroner. only the scope.	Averag e
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 4.4: ratio of optimizations to the program with no optimizations relative to no optimizations in percentage

average values [time / frame]	Inside one cubes	27 adjacent cubes	50 random cubes	50 kr. without Plot	50 kroner. only the scope.	Averag e
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**	**33,501**

Table 4.5: ratio of total optimization to specific optimization relative to total optimization in percentage

4.4 Summary PROGRAMME structures

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. It checks to see if they are in its field of view and converts these same colored parts into 2D to specific cells, these are converted into commands (coloring lines) and checks to make sure all the cells lie in the screen. After the display, statistics are entered into the program and the program waits for further input (Figure 4.3)(Figure 4.4).

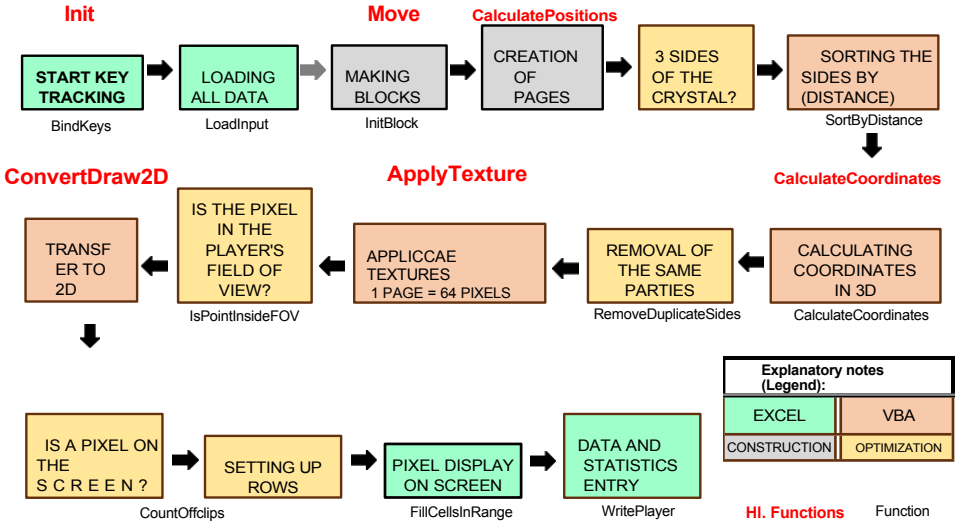


Figure 4.3: diagram of the basic structure of the whole program

Figure 4.4: example of rendering an image using the program

5 Discussion

This penultimate part of the thesis looks at how to improve the program, how to build on this work, and how to apply the program.

5.1 OPTIMIZING

The biggest room for improvement is in the speed of the programme. Despite the implemented optimizations, the program does not reach the required standard rendering speed close to $\frac{1}{60}$ seconds (60 frames per second). There are other methods to speed up, but they require external programs outside Excel. Because of the goal of this paper, which is to render (render) using only VBA in Excel, these options are set aside and may serve for future development.

5.1.1 GPU

The main limitation of the VBA language is the inability to use GPUs [9]. This would allow to speed up coordinate calculations, as the GPU is directly designed for this purpose. There are external libraries that allow communication between VBA and GPUs using interfaces such as OpenCL[1], optimized to work with 3D rendering[2].

A concrete implementation could work as follows: First, the VBA in Excel would collect all the data from the individual cells, which would be passed via a text file or as parameters to a (not yet implemented) C program using *OpenCL*. This program would then perform all the calculations, analogous to what is already programmed in VBA. It would then return the results either in the form of a return value or via a text file containing a list of individual cells to be colored back into the original program. This implementation would allow the time of all numerical operations to be reduced to milliseconds (for a single frame)(Figure 5.1).

[1] OpenCL (1992, Khronos Group)

2External library (2023, github.com/Excel-lent/ClooWrapperVBA, Excel-lent)

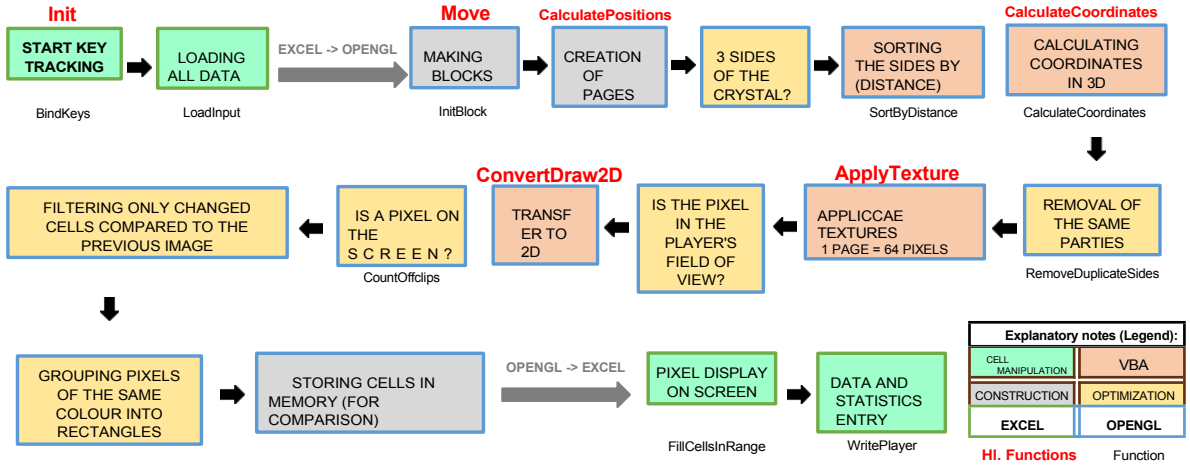


Figure 5.1: diagram of the basic program structure with *OpenCL* calculations

5.1.2 Plotting cells

The amount of computation is only one of the two main components of the program's runtime. The other essential part is working with the graphical interface - with cells. The operation of plotting a single cell (or a rectangle of cells) is a more time-consuming operation than mathematical operations. In the current version of the program, it is possible to turn off the GUI update during cell plotting and then turn it on after the plotting is finished. However, in Excel, this transition takes longer as the number of plotting operations increases, because each change requires updating the GUI and plotting new cells. This can lead to delays, especially when working with large amounts of data or complex cell formatting operations. This behavior can even lead to application crashes due to overflow of numbers or random access memory (*RAM*).

The program currently draws the sides that are not visible, which leads to repeated pixel color changes during rendering of a single image. This can be solved, for example, by creating a 2D array to write these colors to. The program could then plot the colors from this array into the cells after all the colors have been rendered. Such an approach would reduce rendering inefficiencies and minimize repeated pixel color changes during a given frame.

However, implementing this feature would mean a speed degradation due to the inability to render cells in rectangles instead of individual cells. Therefore, we need to implement a function to group individual pixels into rectangles (Figure 5.2).

Similar to some video formats that effectively store data by saving only the first frame and then only the pixel changes from the previous frames (Figure 5.3), there is a significant reduction in file size. This process can also speed up the rendering of frames, as the program will not have to re-render cells.

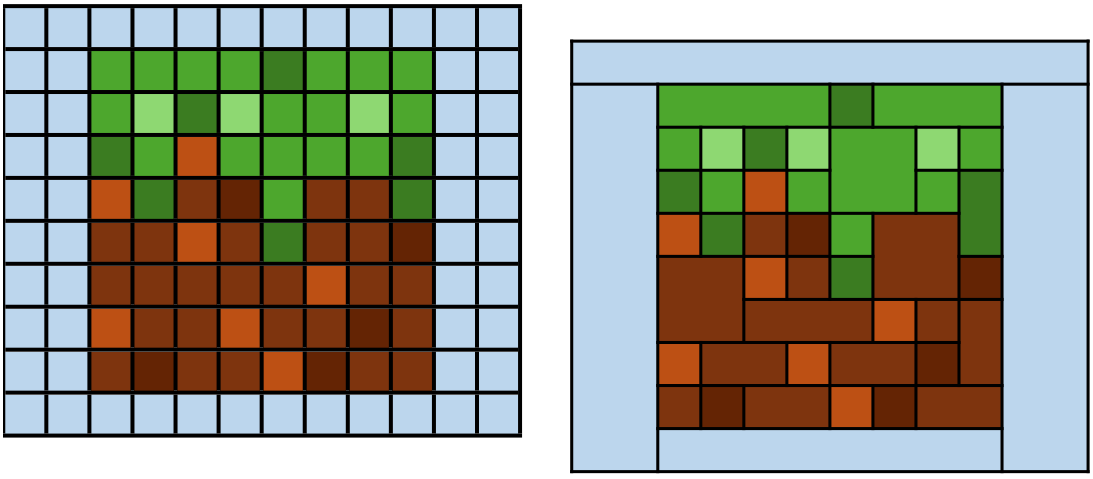
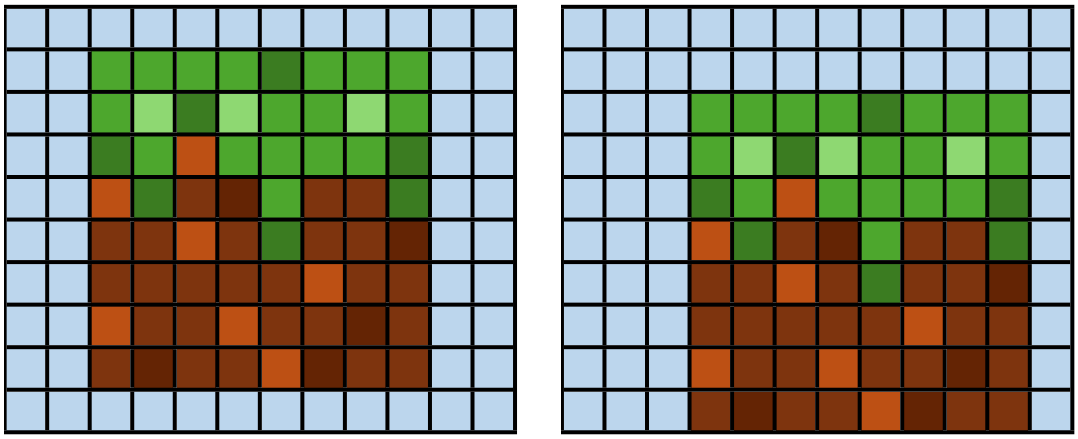
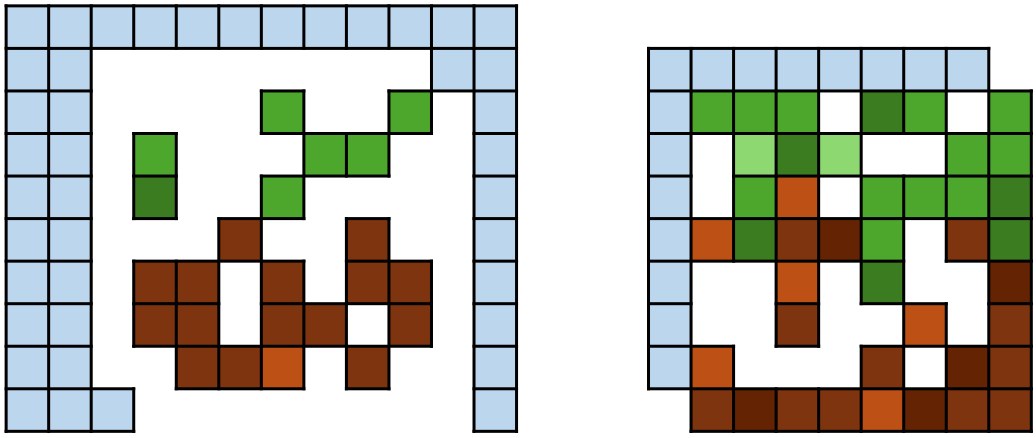


Figure 5.2: cell colouring - on the left by individual cells, on the right by rectangles



Previous image

Rendered image



Intersection of the same cells of both frames

Cell difference of frames (what will be plotted)

Fig. 5.3: colouring only changed cells - bottom right

5.2 Usage

5.2.1 Educational

One of the most important uses of labor is for education. This whole work documents and explains the principles of moving and rotating using linear representation in 3D system, converting coordinates to 2D using perspective projection, explaining the most important parts of the code and their functions and optimizing the program. The work contains many key concepts ideas and things to watch out for when working in a 3D engine.

5.2.2 3D engine

Another major contribution of the work is the implementation of a 3D environment that can work with textured squares, and it is relatively easy to rewrite the program to be able to work with and triangles, as is common with most 3D engines.

5.2.3 Animation

The third main way to use the work is for animation. The program includes a package that allows automating the process of moving and rotating around an object or modifying the coordinates of cubes. The output is sets of images that represent individual frames of the animation and can then be combined using external programs into .gif files (to create the animation itself). The individual outputs are included in the appendix along with tools and instructions for using them (examples: Figure 5.4, Figure 5.5 and Figure 5.6).

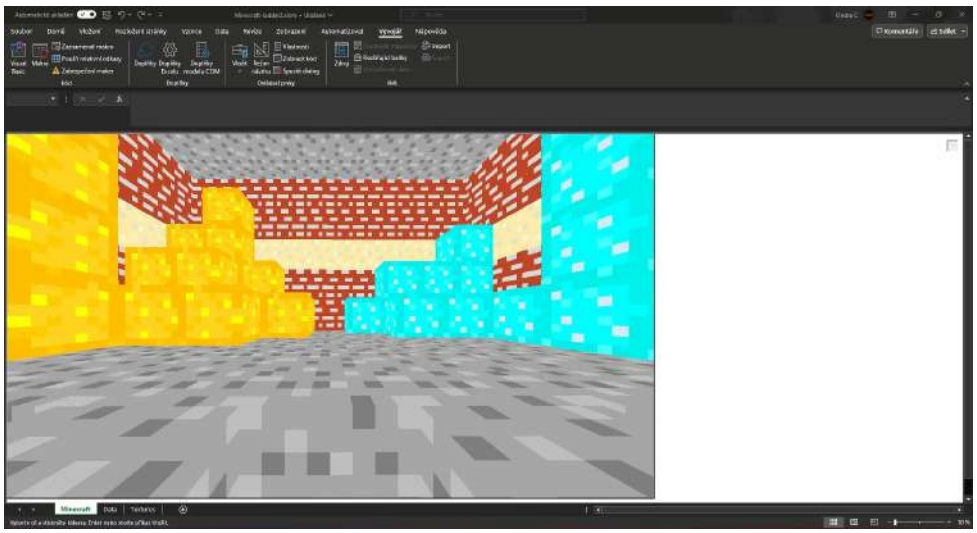


Fig. 5.4: animation example - 1x frame from the animation of climbing up in a building

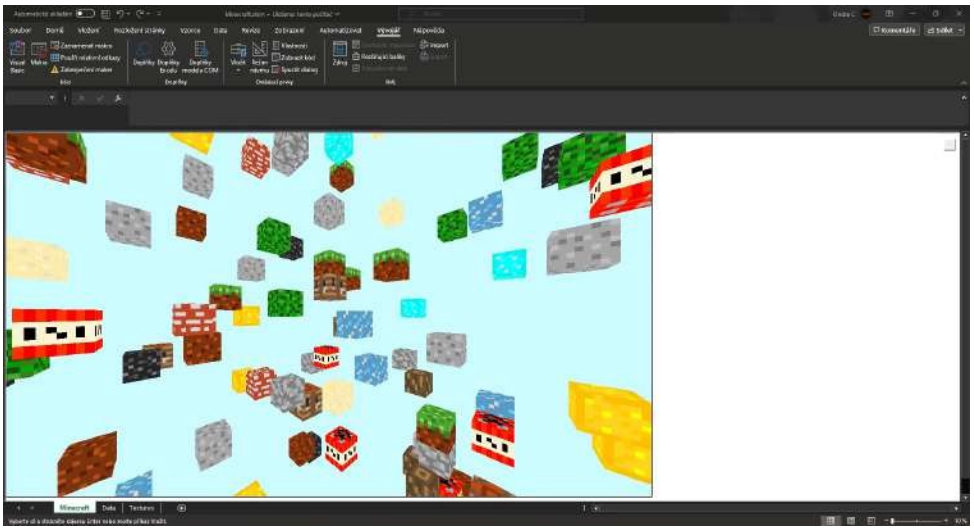


Fig. 5.5: animation example - 1x frame from animation of falling player around cubes

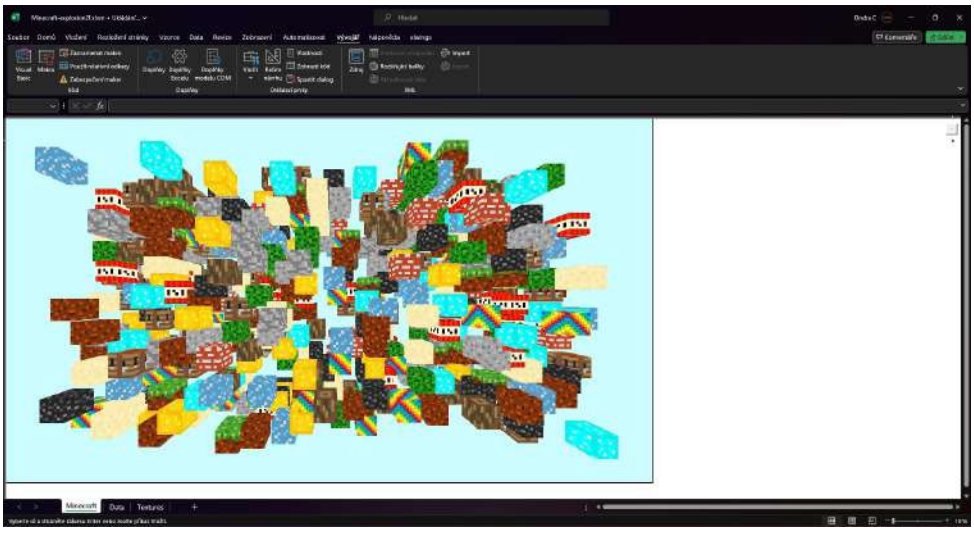


Fig. 5.6: example animation - 1x frame from animation of cube shattering into space and back

5.3 OPTIONS Continued from

Since this is a 3D cube display skeleton, it offers the possibility of continuing the output on the port for the computer game Minecraft. However, there are a few issues that need to be resolved before doing so.

To program the basics of this game, you first need to create a few game functions. This includes simulating simple physics such as detecting collisions with blocks, implementing gravity, and simulating light. Such an approach would mean that the rendering of frames would no longer be contingent on player movement, but frames would be counted continuously. Therefore, the primary problem to be solved is the stability of Excel. Another basic game mechanic is inventory and item production, which could be implemented using buttons and other key

abbreviations. Laying and extracting blocks should also be an important part. The program is already ready for this implementation, as it stores the distances of each cube from the player sorted by size, and it would only be necessary to check if the block is in the middle of the screen.

Last but not least, it would make it easier for users to add new point parameterization shapes. Section 3.6.3 explained how to add new solids, but this requires changes and the addition of more options to the conditions in three different parts of the program. Due to the inability to format the collection clearly (in Visual Basic for Application), it would be necessary to use an external file, for example in *JSON* format, where all the parameters for each block would be stored. However, creating a function that would be able to retrieve individual data types from a *JSON* file brings with it a large number of problems due to the nature of the object-oriented VBA language.

6 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 motion. 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- cel environment, which prevented satisfactory results regarding the speed of frame calculation. This problem was subsequently 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 program. Overall, the information in this paper can serve as an insight for users into the basics of how three-dimensional environments work. The program also has room for improvement, both in terms of optimizations and user interface.

LITERATURE

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

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

[3] TICHOTA, M. (2009). *Perspective as a mathematical model of the lens*. University of West Bohemia in Pilsen, Faculty of Applied Sciences, Department of Computer Science and Computing, 2. [online]. Available from: https://home.zcu.cz/~mikaMM/Gallery%20studentskych%20work%20MM/2009/Tichota-Perspective%20in%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 [online]. Available from: https://cdn.intechopen.com/ pdfs/14853.pdf.

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

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

[7] JOHNSON, M . (2018). *9 quick tips to improve your VBA ma- cro performance*. Microsoft Tech Community, part 6 [online]. Availablez: 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 [online].

Available from: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/ 050010.pdf.

LITERATURE

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

[10] DANZIGER, P. (2016). *Big O Notation*. 1-6 [online]. Available from: https://www.cs.ryerson.ca/~mth210/Handouts/PD/bigO.pdf.

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

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

[13] Wikipedia contributors (2021). *Linear views [online]*. Wikipedia: the open encyclopedia. [online]. [cited 2024 Mar 22]. Available from: https://cs.wikipedia.org/w/index.php?title=Line%C3%A1rn%C3%AD_displayed%C3%AD&oldid=20724346.

[14] Wikipedia contributors (2023). *Perspective screening*. Wikipedia. [online]. [cited 2 0 2 4 Mar 22] Available f r o m : https: //en.wikipedia.org/w/index.php?title=Perspective%C3%AD_prom%C3%ADt%C3%A1n%C3%AD&oldid=23155156.

List of terms

- **Spreadsheet processor** - Application for manipulation with tabular data (Microsoft Excel).
- **3D** - Three-dimensional space formed by X, Y, Z axes.
- **2D** - Two-dimensional space defined by a pair of axes (for example X, Y).
- **Player** - The position from which the rest of the scene is observed.
- **Camera** - The direction from which the player is looking.
- **VBA** - Visual Basic for Applications (a programming language used in Microsoft Office applications to automate tasks, for example in Excel to interact with cells).
- **Snapshot** - One static instance of a scene made up of coloured cells.
- **Block** - Designation for an object (in basic composition a cube that has a center and 6 sides).
- **Side** - Designation for one side, which consists of 4 vertices (in basic composition a square) and forms a Block.
- **Pixel** - A designation for a texture unit with a uniform color that makes up the individual sides (1 Side 8x8 pixels in the basic composition).
- **Cell** - A unit in a spreadsheet containing a color and a value, together forming a screen.
- **Collection** - A group of related objects or elements; in VBA, a label for a sheet or dictionary.
- **Linear representation (Linear transformation)** - A representation that preserves vector operations of addition and multiplication by scalar between vector spaces X and Y [13].
- **Perspective Projection** - A method of projection where the projection lines emanate from a common point (the camera) that must not lie in projection, and the parallelism of the rays emanating from the camera is not maintained [14].
- **Landau notation (Big O notation)** - Notation for estimating the complexity of algorithms.
- **Game engine** - A software system for developing and operating video games.
- **GPU** - Graphics Processing Unit, specialized hardware for processing graphics and speeding up computations.

LIST OF pictures

LIST OF terms

LIST OF tables

LIST Excerp codes ts

Attachments

1. **Source Code** - A folder that contains pure source code.
    - **Classes:**
        1. **Block.cls**
        2. **Calculations.cls**
        3. **Game**
        4. **Pixel.cls**
        5. **Player.cls**
        6. **Side.cls**
        7. **Stats.cls**
        8. **Textures.cls**
    - **Modules:**
        1. **Functions.bas**
        2. **Geometry.bass**
        3. **Keys.bass**
        4. **Main.bass**
        5. **Visual.bass**
2. **Animations** - A folder containing 5 resulting video animations and a module of functions:
    1. **cubeRotation.mp4** - Rotation of one cube according to its *Y-axis*.
    2. **explosion.mp4** - 300 cubes coming from one point along *X, Y* axes
    3. **fallingPitAround.mp4** - Moving the player down the *Y* axis around the cubes.
    4. **islandRotation.mp4** - Rotation of the island model according to its *Y* axis.
    5. **ladder.mp4** - Moving the player up the *Y* axis, with a model of the building in front of him.
    6. **_addToMain.bas** - File containing instructions and functions used for animation.
3. **sourcecode.pdf** - the whole source code in one .pdf file.
4. **CubesRendering.xlsm** - The resulting Microsoft Excel work file capable of drawing snapshots. The source code is part of the file.

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

Tables

6 tables of measured rendering times. 10 measurements of 5 sets for each optimization. Measurements were always made using all optimizations (Section 4.3.4) with the given optimization omitted, **in seconds [s]**; red for erroneous or negative).

	Inside one cube	27 adjacent cubes	50 random cubes	50 cubic metres without plotting	50 cubic meters. 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	-	-	-
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

Table 6.1: without optimisation (Section 4.3.3)

	Inside one cube	27 adjacent cubes	50 random cubes	50 cubic metres without plotting	50 cubic meters. Range only.
1	56,656	22,5	1410,147	1397,542	12,605
2	57,226	22,54	652,652	646,417	6,235
3	56,735	22,359	761,587	759,532	2,055
4	56,367	22,875	-	-	-
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	654,61	-6,837
10	56,805	22,39	328,485	322,004	6,481

Table 6.2: player's field of view (section 4.3.4.1)

	Inside one cube	27 adjacent cubes	50 random cubes		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

Table 6.3: number of sides of the cube (Section 4.3.4.2)

	Inside one cube	27 adjacent cubes	50 random cubes		50 cubic meters. 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

table 6.4: duplicate pages (section 4.3.4.3)

	Inside one cube	27 adjacent cubes	50 random cubes		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

Table 6.5: line-by-line rendering (section 4.3.4.4)

	Inside one cube	27 adjacent cubes	50 random cubes		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

Table 6.6: all optimizations (Section 4.3.5)