Project document_RayCaster

1. **Personal information**

   3D visualisation

   Sara Storbjörk

   1004762

   Datateknik, Kandidat inom teknikvetenskaper (SCI)

   1. year

   3.5.2022

2. **General description**

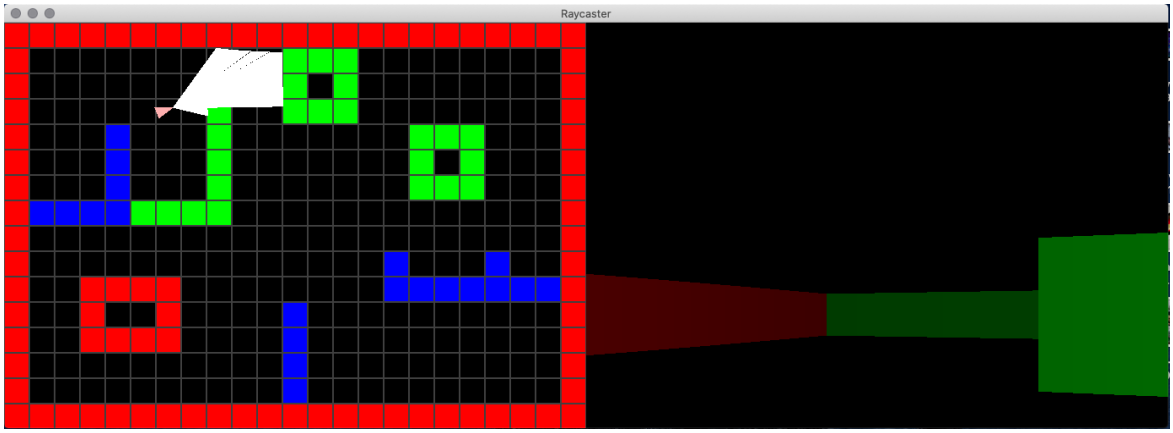
   The topic for this project was 3D-visualisation and the final program is a “Wolfenstein 3D” - inspired 3D scene rendering program. The algorithm that is used to render the scene is called **ray casting** and renders the view of a labyrinth with walls and floors based on an array that is read from a json file within the program. The GUI consists of two side by side windows that the user can interact with through the keyboard. The two windows both represent the same scene, where the left one is the top-down view and right one the rendered image of what the layer can see when looking out into the scene. The vertical lines that draws the 3D image are dependent on the length of the rays that are cast from the player’s position into the scene and stops when a wall is hit.

   There wasn’t a difficulty level in the project description, but I would say the final project was made at an **intermediate** level since some things were left out of the program that was included in the technical plan, which was planned on the demanding difficulty level.

3. **User interface**

   You start the program by going to the object called **Raycaster** in src/main/scala and then clicking “Run ‘Raycaster’” or by clicking the green play-button that should appear in the beginning of the code. The library that was chosen for this project is the Scala Swing Library, which I chose because it had been introduced to me briefly in older programming assignments. Other possible UI libraries would have been i.e scalaFX, but since this programs user interface only consists of drawn images I thought Scala Swing seemed most natural to use. The program is closed by interrupting it in the user’s IDE of choice or by walking into a wall.

   When starting the program two windows appear. The window to the left is the top-down view of the scene (can be seen of as a map) with a player drawn as a triangle with white lines drawn from the player out into the map. The white lines represent the player’s field of view (FOV) of the map. The right window represent what the player sees if it was standing in the map and looking out into the scenery, which would be walls of different heights, colors and brightness based of the distance to the wall.



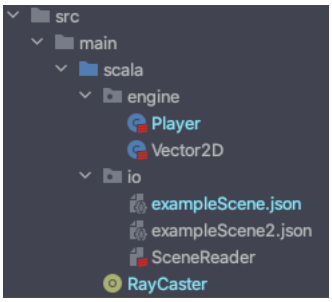
The program’s graphical user interface

   The player can move forwards (UP KEY) and backwards (DOWN KEY). When the player moves forward it does so by a certain moving speed, which in this implementation can’t be changed. If the player walks into a wall the program closes with a closing message. The player can rotate to the left (LEFT KEY) and rotate to the right (RIGHT KEY). The key inputs

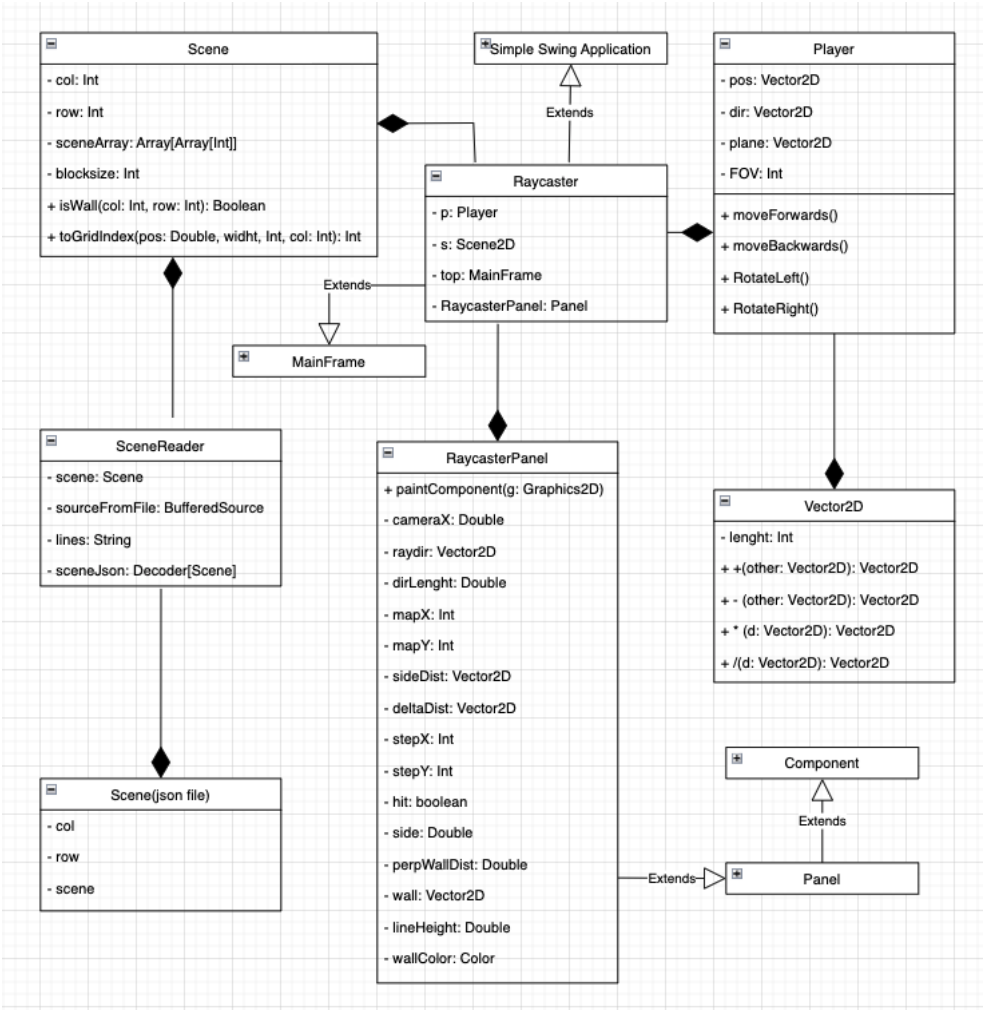
repaints both windows so the user can see how the map and view changes when moving around and gives the illusion of walking around a labyrinth.

4. **Program structure**

   The program consists of a total of five classes and two json files that are grouped into the packages **engine** and **io** and then the GUI (**RayCaster**) by itself. Package engine consists of the keystones of this program: the classes Player and Vector2D. Package io consists of the classes that will read the files with the scene to be rendered, that is class **SceneReader**, case class **Scene** and the files to be read: *exampleScene.json* and *exampleScene2.json*.



The Raycaster and RaycasterPanel class together makes up the GUI, which is a SimpeSwingApplication. The most important algorithms (ray casting and DDA) are implemented in the RaycasterPanel class, that is an extension of the Scala Swing abstract class Panel.



UML-diagram of the program.

The **vector2D** classes methods consists of simple vector arithmetics (computation of the length of a vector and the four basic vector operations: addition, subtraction, multiplication and division of two vectors or a vector and a scalar). Class **Player** was in the technical plan called ‘Camera’ but since only one of the three core vectors in the class can be called a camera I changed the name. The key variables are the three vectors that the whole program is built upon: *position vector*, *direction vector* and *plane (camera) vector*. The key methods are the ones for moving the player around: *MoveForwards()*, *moveBackwards(), RotateLeft()* and *RotateRight()* which moves/rotates the player with a certain amount of pixels/angles.

The case class **Scene’s** most important methods are method *toGridIndex* and *isWall.* Method toGridIndex takes as constructor parameters the player’s position (x or y) and converts it into a whole integer that corresponds to a tile the player is standing on in the map. Based on this method can then the method isWall determine if the position the player is in corresponds to a wall or floor tile.

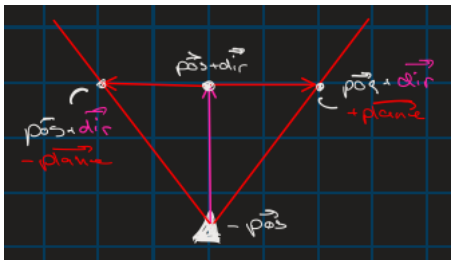
The reason Scene was made a case class is for it to be used in for decoding of the json file that stores the information of the scene (amount of rows and columns and the specific values in the arrays) that is needed for the scene to be rendered. The reading of the JSON-file is done in object **SceneReader** that uses the library Circe (a JSON library for Scala). The key method in this class is the custom decoder *SceneDecoder* that is used to decode the json into a case class Scene in Scala code.

These above mentioned classes makes up the core so that the raycasting algorithm then can be calculated and rendered in the class **RaycasterPanel**, which is a subclass of the abstract classes **Panel** and **Component**. The key methods/variables in these classes are *perpWallDist* from which we get *lineHeight* (calculated as distance from player position to a wall in that direction). The method *wallColor* designates a color to the wall the ray hit and the method *normalized* decides the shade of the color (also dependent on the distance to the player) .

5. **Algorithms**

The algorithm I chose to render the 3D-model is, as mentioned before, a **ray casting** algorithm. We place a player onto the scene, which is a two-dimensional square grid, made up of tiles that are either walls or floors. A ray is cast out into the direction the player is looking and continues on in the same direction until a wall tile is hit. When a wall is hit, the distance to that wall is calculated and then used to decide the height of the line that will be drawn vertically on the screen. Since we want to model everything the player sees in a certain field of vision (FOV) multiple rays is cast out both to the left and right side of the general direction. The two outer rays makes up how much of the scene the player will see when looking in the general direction.

Vectors and a camera is used to cast the rays and calculate the length of them. The general direction-vector consists of the player’s position added to the direction the player is looking. To get the width of the FOV a camera plane (line) is added perpendicularly to the general direction-vector and its’ values goes from -1 (left side) to 1 (right side). The camera plane can be viewed as the computer screen and the general direction-vector as a vector pointing into the screen. The rays cast to the left of the general direction is **pos+dir-plane** and to the right is **pos+dir+plane**.

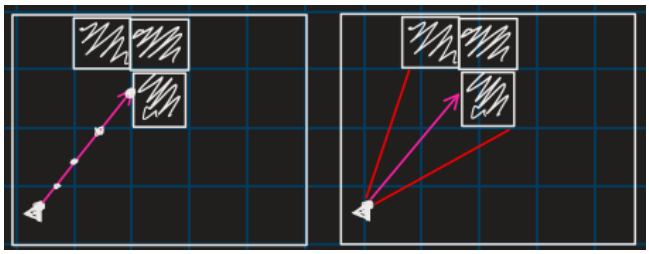


The “camera” represented by vectors. The pink vector represents the general direction the player is watching. The red vectors that are cast perpendicularly to the sides from the general direction vectors represent the camera plane. The length of the plane vectors determines the players FOV.

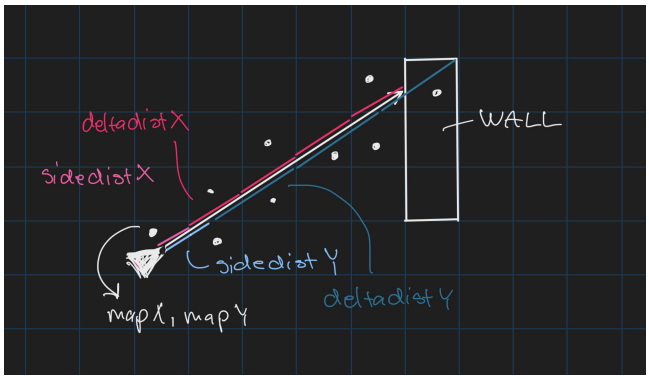
The ratio between the general direction vector and the camera plane decides the size of the FOV. If the direction vector is longer than the camera plane, the FOV is narrow and vice versa. Narrowing the FOV would create the illusion of zooming in, and widening FOV then creates the illusion of zooming out. When a player looks to the side the whole camera has to rotate to either left or right. This is done by multiplying the vectors with the rotation matrix:

$$\begin{matrix} cos(a) & -sin(a) \\ sin(a) & cos(a) \end{matrix}$$

To calculate the distance until a wall is hit, an algorithm based on **Digital Differential Analyzer** is used that will check at every side of a tile in the ray direction if the side belongs to a wall or not. If the side belongs to a floor tile the distance (*deltadistance*) to the next tile border is added to the ray. This distance is dependent on if the ray hit a x-side (vertical line) or a y-side (horizontal line). This algorithm is more efficient then i.e checking at every pixel in the ray direction since only the essential points on the grid is checked, tile borders.



The left image shows the ray cast out in the general direction from the player, with the step size of the ray shown. The right image shows the field of view from the player where the red lines represent the outer rays.



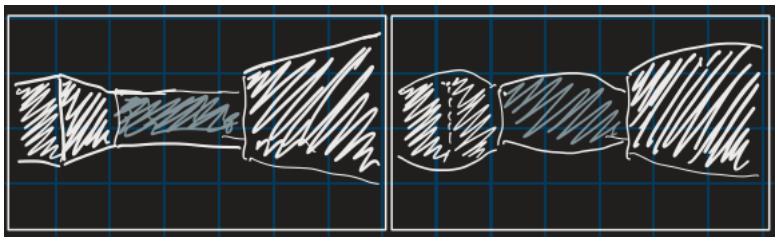
Side distance and delta distance shown in the DDA-algorithm. The white dots represent the current tile (mapX, mapY) the ray is in at the moment. The loop continually checks which of the vectors are longer and determines to either go a square in x or y-direction. Here is also illustrated how the ray goes one distance too far in x-direction since mapX and mapY will always be one loop behind. This is corrected when calculating the final distance: perpWallDIst. In this situation a y-wall was hit, which is why we use deltaDistY to calculate perpWallDist.

```
Ray casting loop //for every x-pixel on screen:

  // ray direction in current x-coordinate
    var rayDir: Vector2D
  // length of ray form current position inside a tile to closest x or y-border in ray direction
    var sideDistance: Vector2D
  // calculate length of ray from one x or y-border to next x or y-border.
  // Based on the Pythagorean theorem, but can be simplified since only the ratio between deltadistance for x and y matters in DDA
    var deltaDistance: Vector2D = abs( 1 / raydir)

perform DDA // Go through every border on the tile in the ray direction until a wall is hit.
  // while ray hasn't hit a wall:
    // check if sideDistance.x or sideDistance.y is longer
      // add deltadistance to sideDistance (either x or y)
      // jump to next map tile, either in x or y-direction
      // decide if border belongs to x or y-wall
      // finally, check if a wall was hit.

  // calculate distance to wall, that is, projected distance from wall to camera plane (perpWallDist)
  // check if x or y-border of wall was hit
  // since we are adding the deltadistance before checking if we hit a wall in the DDA, that distance has to be subtracted here.
  val perpWallDist = sideDistance - deltaDistance

  // calculate height of line to draw on screen

  // decide wallColor based on corresponding array integer value

  // draw vertical line
```

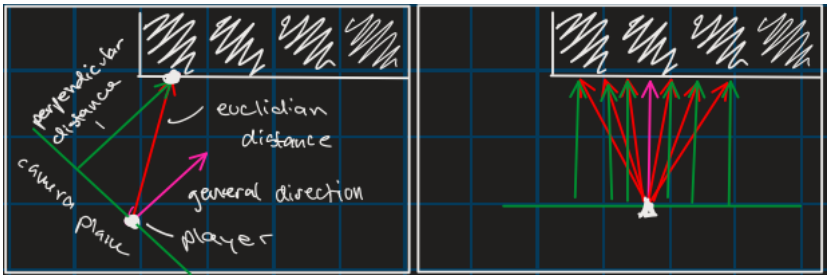
When calculating the distance to the walls it has to be taken into consideration that if we take the euclidian distance from the player to the wall the and then draw the vertical lines, the top and bottom parts of the wall would be rounded, even if the player was looking straight at a wall. This is because the rays that would be cast out to the sides form the general direction will have different lengths and therefore the drawn lines would be of different height and create a rounded effect a.k.a fish-

eye effect. To avoid this the perpendicular distance from the wall to the camera plane is used instead, where the camera plane crosses the position of the player (green line in pictures below).

It would have been possible to use trigonometry instead of vectors to represent the general direction and the rays and another angle to determine the angle for the field of view. But with vectors and a camera the fish-eye problem “fixes itself” and we don’t have to work with angles.



Left window represent the image we get using the perpendicular distance to the camera plane. The right window represent the image we would get if used the euclidian distance between the player position and the wall, i.e the fish-eye effect.



Situation when the ray cast out from the player’s position has hit a wall. The green vectors represent the perpendicular vector to the camera plane, which will be used to calculate the distance from the wall to the player. The red vectors represent the euclidian distance from the player to the wall and the pink vector the general direction. Especially on the right side photo can be seen that the red vectors are of different length in comparison to the green vectors.

The final formula that is used in this project is to normalise the *lineHeight* from the screen height to fit into the RGB range (0 to 255):

$$newvariable = (oldvalue - min)/(max - min)$$

6. **Data structures**

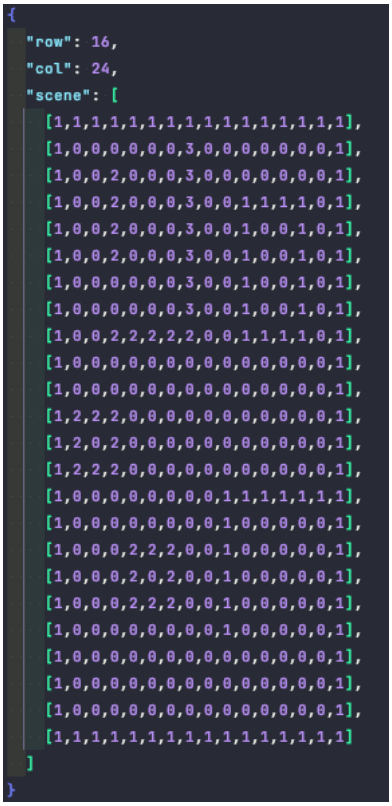
   The main data structure and main building block for the program is a two dimensional array that makes up the walls and floors of the scene. In my program I chose to use an immutable collection since the contents of the array doesn’t change - it’s only read. The array is a simple data structure to use and since I didn’t need to change its substance I didn’t feel the need to use a buffer. If however textures where to be added, a buffer would be more suitable. All other values are stored as integers, doubles or colors.

7. **Files and Internet access**

   The program reads a json-file which consists of a single Json object with three Json objects inside: The array that builds the scene and additional constants for row and column numbers. The json-files can be found under the package io and can be switched by simply change the file path in the SceneReader class (row 23) and then restart the application. If the user would want to make their own scene they can do so by using the two examples that was provided in the code and add a new file based on that information in the same package with the other files.

```
21  object SceneReader {
22
23    private val sourceFromFile = Source.fromFile("src/main/scala/io/exampleScene.json")
```

To change between different scenes you can change the file path to the file name of your scene in SceneReader.



Example of how a json-file used in this program can look like.

The json files has three key value pairs: row - amount of rows (integer), col - amount of columns (integer) and scene - the two dimensional scene array of integers. The integer values goes from 0 to 3, where 0 represents walls and the values 1 to 3 represents walls of different colors (Red, Green, Blue). I found having the row and column amounts hardcoded into the file made the program less prone of IndexOutOfBounds-exceptions. Every column and row of the scene has to include the same amount of integers.

8. **Testing**

   The testing process followed the testing plan well since I had planned to do most of my testing via the GUI where I can compare the both windows I have of the scene with each other. The other kind of testing I've done is by printing out the values for specific variables in the terminal, i.e when implementing the DDA and ray caster algorithm I used that method. I feel like using the GUI worked really well since my program is almost only graphical.

   I didn't include any particular tests for the file management because I only needed for the json file to be read and converted from Json to Scala. In that stage of the project I used print-methods and the terminal to see what was going on during the implementation.

9. **Known bugs and missing features**

   The biggest defect that will be noticed in the program is that the ray casting algorithm isn't exact. This isn't all too noticeable if you only watch the right side window (the rendered scene), but since I also included the top-down view of the scene this impreciseness is kind of screaming at the user. Sometimes the ray looks right through a wall or kind of makes the wall "jump" a tile to the side.

   This defect is most noticeable when the scene is more complex and crowded. I deliberately let the first scene (exampleScene.json) have more walls and then the second one (exampleScene2.json) more spacious. When comparing the algorithm in these two scene it is way nicer to look around in the one with more space.

   I couldn't find the reason for this defect and I tried lots of other options, but the final code is what seemed to be the most precise. Since it's more noticeable on the left window might have something to do with the way I drew the FOV in 2D (the white rays), but also something fundamental in the ray casting logic. Maybe it can have something to do with when I'm converting the pixel position of the player to the tile position, and that would be what makes the rays "jump" sometimes. Or it could be because of something completely else.

   Also a small bug that exists is when one of the direction rays is 0 (happens when the player is positioned at a 90 degree angle) is that the line in the middle of the picture is drawn wrong. An example of this bug is if you look at the rendered

image when starting the program the first time that there is a blue line in the middle of the picture, even though it should be green.

To fix this it would be possible to handle these cases seperately but when trying to do so it still had the wrong color. Another missing feature that I had included in the technical plan is zooming, but I decided to leave it out completely since I had struggles with the algorithm and also later when trying to add threading.

10. **3 best sides and 3 weaknesses**

    I think the best sides in the final program is the file reading, since it both work well in different array sizes and the code is simple, concise and reliable. If you use a smaller array for the scene the whole window will be smaller but the internal logic still works the same. This also means that you can't make an array too big since it won't fit on the screen.

    The next best side is the usage of vectors in this program since it made it the whole implementation process easier and clearer than using trigonometry, except for the rotation movement where the rotation matrix was used. Lastly I like that I included the top-down view of the map since I feel like it gives more to the program and user experience overall.

    On the other hand, the top down view in the final program is also a weakness, or more exactly the rays that makes up the FOV, since the execution isn't as clean as I would have wanted it to be, as explained above. Another weakness is that the only available colors in this program on the walls are red, green and blue in different brightness. For example a customizable color palette or textures would have been nice a nice addition.

    Lastly, the overall performance of the program isn't as good as it could have been and to fix this I was encouraged to implement threading since the rays are calculated and drawn independently to each other. So I spent the last week trying research and implement this in the program but the best result I got only painted about 1/5 of the rays onto the screen. If I were to continue with this program I would like to get that to work.

11. **Deviations from the plan, realized process and schedule**

    I didn't manage to stick to the implementation schedule and I did end up requesting a one week deadline extension since I planned to not implement threading into the project, but I changed my mind when the deadline was coming up in less than a week. In the end I didn't get the threading to work as I wanted so it still got left out of this project.

    I spent most my time on this project on the first two weeks and in the last three weeks. The order of the way things were implemented went almost exactly according to the plan, except for the struggles I had in the beginning with the file reading that made me have to leave it for a while and instead to start working on the other parts of the program. Also the addition of threading made me have to work a bit extra around and after wappu, which accounted for in the original plan and sadly I decided to let it go since I didn't get it to work.

    The first two weeks I got most of the top-down view of the scene implemented and I had a player drawn and moving. Unfortunately I had quite some struggles in the beginning. I had never built a GUI before so it took about 10 hours of reading on using the Scala Swing library before getting even a rectangle drawn on the screen. Also figuring out how to repaint the scene took some time since i didn't realise I need to focus the panel and also then call to it when repainting. Another issue that I had was that the rotation matrix that I used was using radians instead of degrees, so I thought something was wrong in my implementation,

    Then the player didn't recognise walls at all or walked half-way through the wall before stopping, which was because I hadn't figured how to accurately convert the player position on the map into a certain tile in the scene array. I also spent an overwhelming amount of time figuring out how to read json files using the circe-library, which made it hard for me to move on to the main part of the project. It ended up me hardcoding the scene I wanted to use unto the GUI so I could move on forward on the project even though the file management wasn't working. (25h)

    For the second two weeks things were going forward slowly on the project, but I ended up getting all my issues from the earlier two weeks fixed so I then could start form a clean plate and start implementing the raycaster. (10h)

    The next two weeks with the implementation of the ray caster and DDA algorithm went fairly smoothly, except for the bug/implementation error with the FOV that I never figured out. I also struggled with what I thought was user input. The origin to this problem was actually that I used two separate panels for the two windows instead of a single one, which meant that only one of the panels was 'in focus' could listen to user input at once, which then meant that only one of the windows got repainted. But after realizing my mistake I fixed the issue by collecting all the code into the RayCasterPanel class and moved on to add shading. (25h)

    On the last week I tried to fix some bugs and added the feature that the application closes if you walk into a wall. I did also try to implement threading. The best try I had consisted of using Futures in the loop and having it calculate everything until the perpendicular wall distance. Then I wanted to use a callback (onComplete) to do the rest of the calculations but my code only drew about 1/5 of the rays, which apparently could have been because of synchronization issues. (20h)

The final time I spent in this project (with he time I spent on this document included) is about 100h in total, which is about what I expected since I've never coded a whole program before and there is bound to be "stupid" errors that will be made.

12. **Final evaluation**

    The final quality of the program is a bit lacking in comparison from what I had hoped when I started this project. The reason for this is that the most important algorithm isn't working properly, which has it's effects on the overall user experience, as explained in section 9 and 10. Also the fact that this program is just a player walking around a map is a bit boring by itself, but I can see the potential of making it into a game (like Wolfenstein) or a building visualisation program. On the other hand am I proud of what's been made since I've never coded a complete program by myself before.

    The biggest improvement/change that would have to be made to the code is if, for example, textures where to be added. In that case the raycasting algorithm would need to be even more precise in where exactly on the wall the ray hit it for the corresponding texture coordinate to be rendered. The GUI could be more user friendly with buttons for starting and exiting the program instead of just two drawn pictures. Also information such as player position, direction (a compass perhaps) could be shown on the GUI. It would also be nice if the user could change the layout of the scene when using the program, which means that the file management needs to be developed further so that the original files gets written over and saved as a json file, which would include encoding of the array/buffer that the user designs into json values. I think the overall structure is good for further development, but the GUI is a bit messy since it contains a bit too much code that could be seperated into different classes/methods.

    If I started the project over again I would start the same way because I feel like I did prepare well for the project, but I would look into more different possibilities on know to solve the problems related to the implementation instead of choosing the first alternative I can find. I would also move the raycaster algorithm into its own method and class, because, as mentioned above, the GUI is a bit messy and crowded with both the core algorithms and GUI code in one single class. It would also probably make the implementation of threading easier if the heavy computations where done in a single method. I would also ask for help earlier instead of deadpanning my way to a solution.

    Despite the struggles I've had during the program it feels rewarding to have done it and also to get a glimpse into the field of computer graphics. I'm now even more amazed by graphic/game engines like Unity and Blender than ever before and might even have gotten an interest in this particular field of computer science.

13. **References**

    Ray casting:

    https://en.wikipedia.org/wiki/Ray_casting

    https://lodev.org/cgtutor/raycasting.html#The_Basic_Idea_https://github.com/vinibiavatti1/RayCastingTutorial/wiki/RayCastin
    v=gYRrGTC7GtA&t=306s

    https://www.techopedia.com/definition/21614/ray-casting

    https://fi.wikipedia.org/wiki/Wolfenstein_3D

    DDA: https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)

    Rotation matrix: https://en.wikipedia.org/wiki/Rotation_matrix

    Normalization: https://www.stathelp.se/sv/center_sv.html

    Scala Swing/ Graphics 2D: https://docs.oracle.com/javase/tutorial/uiswing/components/index.htmlhttps://index.scala-
    lang.org/scala/scala-swing/scala-swing/3.0.0?target=_3

    https://www.scala-lang.org/api/2.11.12/scala-swing/index.html#scala.swing.package

    https://otfried.org/scala/gui.html

    https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html

    Circe/json:

    https://circe.github.io/circe/

    https://www.baeldung.com/scala/circe-json

    https://manuel.bernhardt.io/2015/11/06/a-quick-tour-of-json-libraries-in-scala/

    https://mungingdata.com/scala/read-write-json/

    Concurrency:

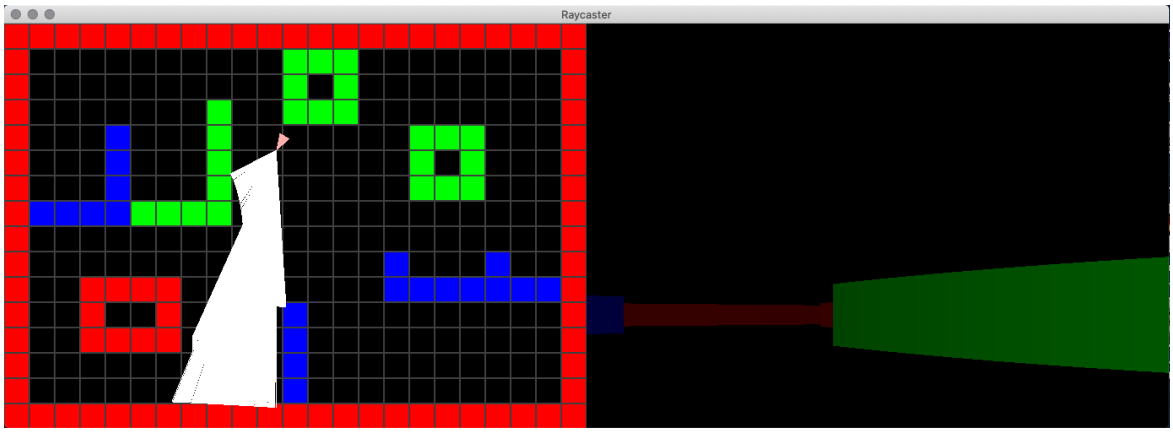
    https://docs.oracle.com/javase/tutorial/uiswing/concurrency

https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

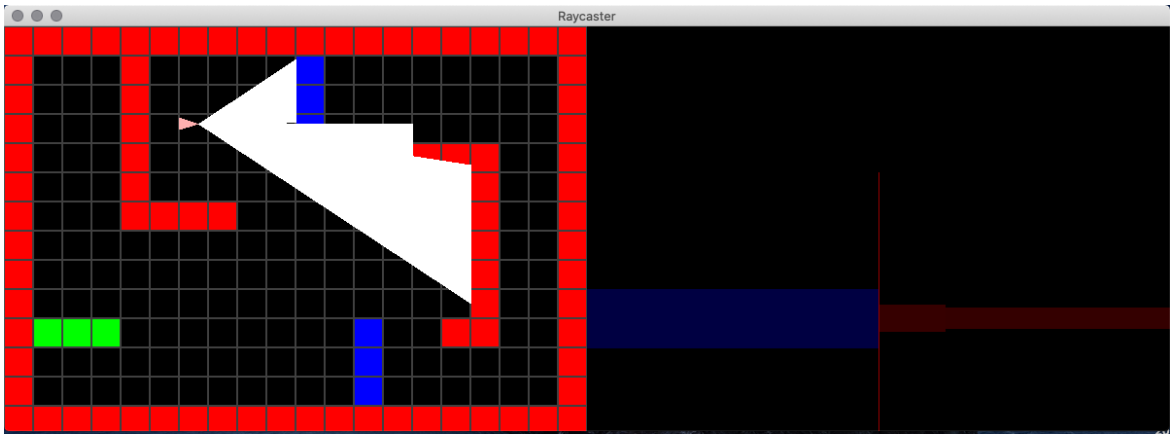
https://alvinalexander.com/scala/concurrency-with-scala-futures-tutorials-examples/

14. **Appendixes**

    The most important project appendix is **the full source code of the project**.



Example situation from ‘exampleScene.json’.



Example of the scene from file ‘exampleScene2.json’ where the bug where one of the ray direction vectors are 0 and therefore paints the wrong line in the middle of the picture. Also the FOV on the left image are not precise and are going through some of the walls.