

Notes on Permadi Raycasting Tutorial

Joseph21, may 5, 2023

Disclaimer: For the first part (where I implemented along the Permadi tutorial chapters) the notes are pretty decent and structured. The second part (where I experimented with additional concepts in ray casting) the notes are not very structured (yet).

The Permadi tutorial series (explanation and examples) can be found here:

<https://permadi.com/1996/05/ray-casting-tutorial-table-of-contents/>

All my implementations using the PixelGameEngine can be found on my github:

<https://github.com/Joseph21-6147/Raycasting-tutorial-series---Permadi-inspired>

If you want to code along, you'll need to provide the sprite and texture files yourself.

Advice: The notes do not provide a complete description of the implementations. There are several sources you should combine with these notes to get the complete picture:

1. *The Permadi tutorial pages;*
2. *The differences between the current code file you are studying and the previous – use some difference analyzer like WinMerge to get the incremental changes in the code visible.*
3. *Each code file has its own header comment in which I summarize what was changed with respect to the previous code file.*
4. *I try to make the code readable by adding comments where appropriate. So in the end you'll just have to read code 😊*

Preface

- Tutorial notes¹ - none
- Implementation notes - none

Part 1 – Introduction

- Tutorial notes
 - Describes the history and the concept of ray casting
- Implementation notes - none

Part 2 – Ray casting vs Ray tracing

- Tutorial notes - none
- Implementation notes - none

¹ The tutorial notes refer to the parts of the above ray casting tutorial pages

Part 3 – Limitations of ray casting

- Tutorial notes
 - relevant theory starts here
- Implementation notes
 - each tile is split in 64x64 units, and each cube in 64³ units. I'm struggling how to model this, since keeping the tiles as units (like Javid does) is much more logical for the ray tracing. For now I only created some int nTileSize and nGridX, nGridY pair that are derived from nTileSize in combination with nMapX, nMapY. However, these variables aren't used anywhere (yet)²
 - I created a map (modeled as a string), defined a map size

Part 4 – Step 2 = Defining projection attributes, part 5

- Tutorial notes - none
- Implementation notes
 - I created class variables for the player position, it's viewing angle, its field of view and its height.
 - The viewing angle and field of view is in degrees

Part 6 – step 3 = Finding Walls, part 7

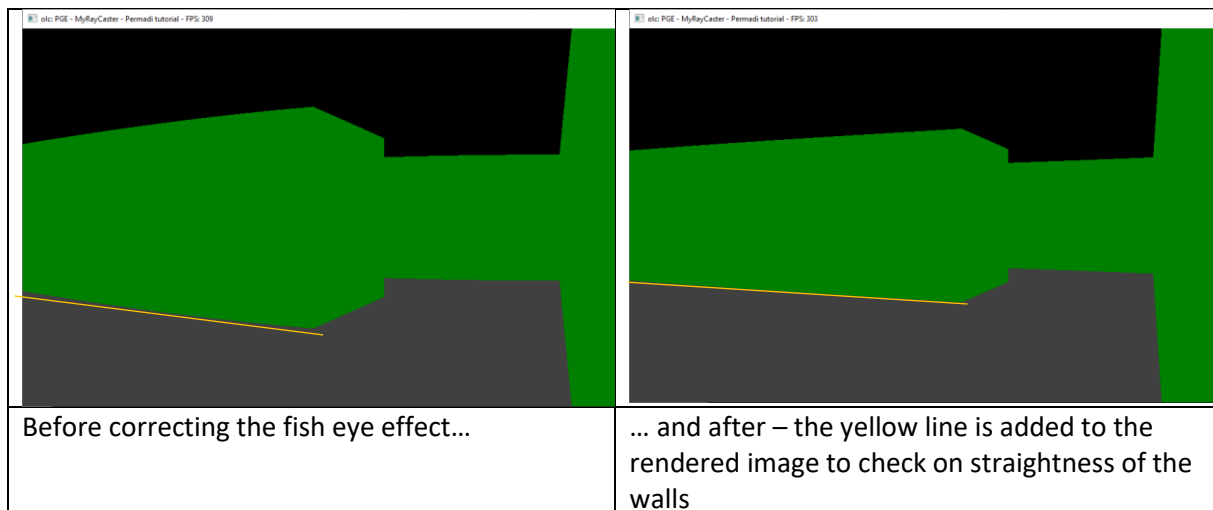
- Tutorial notes
 - this is where the Differential Data Analysis (DDA) algorithm comes in. Imo the explanation in the Permadi tutorial isn't particularly comprehensive, so I sought and found more explanation in for instance Javid's video on DDA³.
- Implementation notes
 - I adopted the DDA algo code from javid's video tutorial, and wrapped it in a function to use it in this tutorial
 - Later on I implemented my own DDA function, based on Javid's video. So I adopted the idea and the concept of the algorithm, but didn't copy the code.

Part 8 – step 4 = finding distance to walls

- Tutorial notes
 - Describes the fish eye effect and a way to correct that
- Implementation notes
 - This correction uses an additional cosf() call. This is only needed per screen slice. I might optimize performance using a lookup table in the future.

² Later I removed this code. My map / wall blocks are 1 x 1 x 1 in size. The initial height of the player is 0.5f

³ See: <https://www.youtube.com/watch?v=NbSee-XM7WA&t=1053s&pp=ygULamF2aWR4OSBkZGE%3D>

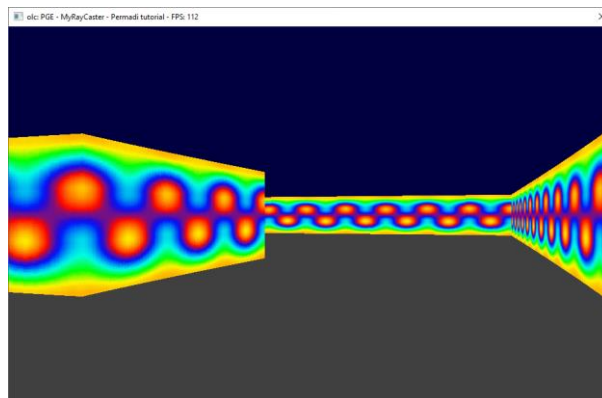


Part 9 – step 5 = Drawing walls

- Tutorial notes
 - In the tutorial a constant number of 277 is presented as the distance to the projection plane. This number is a result of the (choice of the) screen width (320 in the tutorial) in combination with the field of view angle (60 degrees, the cos of half of that is $\frac{1}{2} \sqrt{3} \approx 0.866$, so $320 * 0.866 \approx 277$ (rounded oc))
- Implementation notes
 - I implemented this distance to the projection plane as a derivation instead of a constant: Since the screen width and the field of view don't change during the game, this calculation is put in OnUserCreate()
 - [saved a copy of the source file – since I finally got the code rendering something]

Part 10 – Textured mapped walls

- Tutorial notes
 - None
- Implementation notes
 - Needed some debugging time to find out I missed a '&' in the parameter to GetDistanceToWall(). The y coordinate of the cell wasn't interpreted as a reference and didn't get any value due to this... Now it's working fine:



- I downloaded Permadi's texture files in the process. Not particularly special (and only 64x64 resolution), but handy (and colourful).

- [saved a copy of the source file]

Part 11 – Drawing floors, part 12 – Floor casting (continued)

- Tutorial notes
 - The way of thinking from the tutorial is pretty clear: the theory is comprehensible. You have to add a lot of your own stuff to get it implemented. The tutorial describes how to get the floor distance, and then you have to implement yourself how to work that out to the correct floor location (tile), and to the correct sample coordinates for sampling the floor sprite.
- Implementation notes
 - Success!! [although it takes a performance hit with 1 x 1 pixelsize 😊]
 - It's just the right amount of challenge. Not too easy, you got to figure things out for yourself.



Part 13 – Drawing ceilings

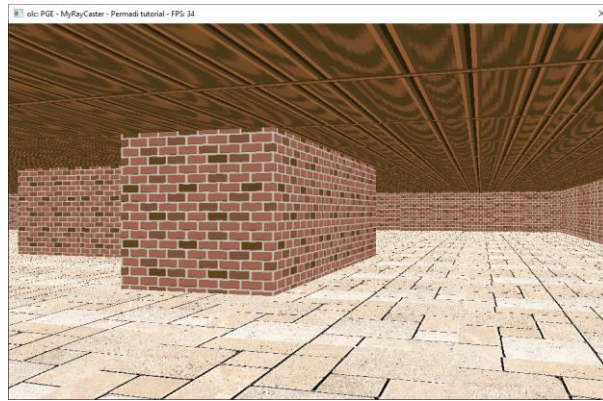
- Tutorial notes
 - This is a straightforward extension of part 11 and 12
- Implementation notes
 - Texturing the ceilings is a pretty straightforward variation on texturing the floors. You only have to take into account the relation of the y screen coordinate and the middle of the screen height...
 - For the floor:

```
// work out the distance to the location on the floor you are looking at through this pixel
// (the pixel is given since you know the x and y to draw to)
float fFloorProjDistance = ((fPlayerH / float( y - nHalfScreenHeight )) * fDistToProjPlane)
                          / cos( fViewAngle * PI / 180.0f );
```

- For the ceiling:

```
// work out the distance to the location on the ceiling you are looking at through this pixel
// (the pixel is given since you know the x and y screen coordinate to draw to)
float fCeilProjDistance = ((fPlayerH / float( nHalfScreenHeight - y )) * fDistToProjPlane)
                          / cos( fViewAngle * PI / 180.0f );
```

- Success!!



Intermezzo:

1. Got a reference from discord to: <https://lodev.org/cgtutor/raycasting.html> Check it out, looks usable. Javid refers to this site as well in his DDA video.
2. Performance is taking a hit since I use a lot of trig functions:
 - a. One `atan2f()` call for each wall pixel;
 - b. One `sin()` and one `cos()` call for each floor and each ceiling pixel;
 I can try to optimize with `sin()` and `cos()` as lookup values rather than function calls.

Part 14 – Variable height walls

- Tutorial notes
 - The explanation in the Permadi tutorial is not straightforward (for me at least). So I pretty much derived my own approach on this challenge. Refer to the separate file **Multilevel raycasting algorithm YYYYMMDD.pdf** for the explanation of this algorithm.
- Implementation notes
 - The DDA function that determines the first hit and the distance and location thereof had to be adapted to return a list of hits for each map cell that has a height > 0 . This list of hits is used in the rendering of walls.
 - Permadi's tutorial really wasn't that helpful for this topic. It took me a lot of time to figure this out...
 - First step was to enhance the function that calculates the distance to the nearest wall: it should return a list of all walls that are intersected (not just the first one), including their heights.
 - Second step is to build up each vertical line on the screen using that info. If a wall is hit, render it, else render either floor or ceiling. If another wall is behind the first one, and is higher, render the top part. This is very similar to what I worked out before I followed the permadi tutorial.

Algorithm overview

Step 1 - Regular ray casting stops upon finding the first hitpoint (or the boundaries of the map). For multilevel ray casting you have to create a list of hitpoints. So don't stop at the first hitpoint, but only stop at the boundaries of the map. Record all hitpoints where the height of the block before and after the hit point differs.

This can typically be implemented in the ray tracing function/code.

Step 2 - Using Fermat you can calculate how a wall segment (or a block) projects onto the screen (this was already done in regular ray casting). Use the same approach to extend the hitpoint information with the info on how each hitpoint is projected onto the screen.

You can put this in a separate function, or integrate it in the rendering function.

Step 3 - When rendering the slices on screen, you use the hitpoint list to determine how to render each pixel on the screen.

This is typically done in the rendering function / code.

Algorithm step 1 – build intersection list

Intersection #	Distance to player	Height becomes at intersection
0	d_0	2
1	d_1	1
2	d_2	0
3	d_3	1
4	d_4	4
5	d_5	0

Algorithm step 2 – add projection info per hit point

Black content – initial table content, filled after ray casting
Red content – added using projection calculations

Intersection #	Distance to player	Height becomes at intersection	Projected bottom	Projected top (front)	Projected top (back)
0	d_0	2	b_0	c_0	e_0
1	d_1	1	b_1	c_1	e_1
2	d_2	0	b_2	c_2	e_2
3	d_3	1	b_3	c_3	e_3
4	d_4	4	b_4	c_4	e_4
5	d_5	0	b_5	c_5	e_5

Use the Fermat formula to project block bottom and top onto the screen. For the bottom, only the projected front of the block (at intersection) is relevant. For the top of the block, not only the front but also the projected back of that block is relevant and is stored in the extended column list.

e_n = projected top height calculated with distance d_n , so d_n so the last e value is meaningless

Algorithm step 3 – render slice using the intersection list

```
// y is the screen height value (vertical coordinate) of the pixel that is rendered
if (y > bi)
    render floor
else if (bi >= y > ci)
    render wall (using distance di)
else if (ci >= y > ei)
    render roof
else { // ci, ei > y
    Try next point (i + 1) from intersection list with same criteria
    If (no next point available) → ceil
}
```

Please check the notes in the separate file **Multilevel raycasting algorithm YYYYMMDD.pdf** for an explanation of the multilevel ray casting algo

Part 15 – Horizontal movement

- Tutorial notes
 - If you are familiar with Javidx9's video's on first person shooter, this stuff is known territory (if you're not, check them out)
- Implementation notes
 - I implemented these movements already in the first version of the code (the implementation of part 9), since there wouldn't be much to look at or to test otherwise... So the implementation is there at the start (part 9).
 - Besides rotation and forward and backward movement, I also implemented strafing left and right.
 - Experimented and defined movement speeds for rotation, strafing and forward / backward.

Part 16 – Vertical Motion: looking up and down

- Tutorial notes
 - This is a feature that Javidx9 didn't explain.
- Implementation notes
 - This is quite trivial to implement. I added a class variable fLookUp, which is 0.0f initially, and which is controllable with the up and down arrow keys. Although it expresses pixels (screen space), the variable must be a float to fractionally (i.e. smoothly) increase or decrease the value. I let the value of the vertical half of the screen depend on it:

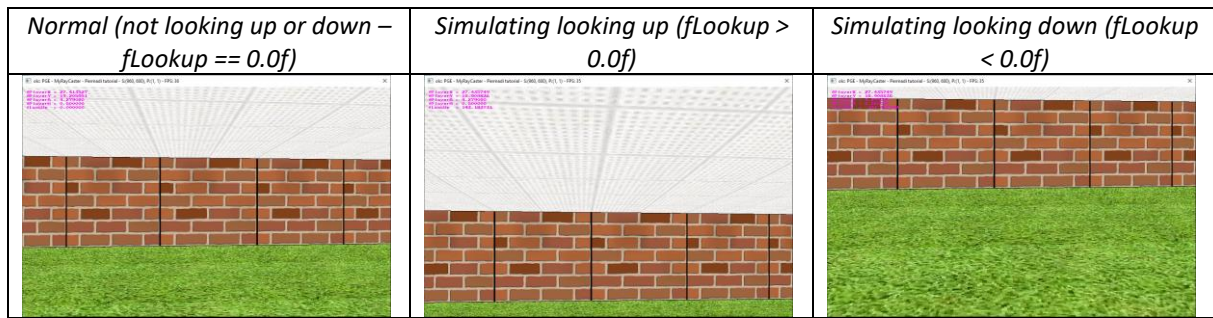
```
int nHalfScreenWidth = ScreenWidth() / 2;

int nHalfScreenHeight = ScreenHeight() / 2 + (int)fLookUp;
```

- This is where that value is applied:

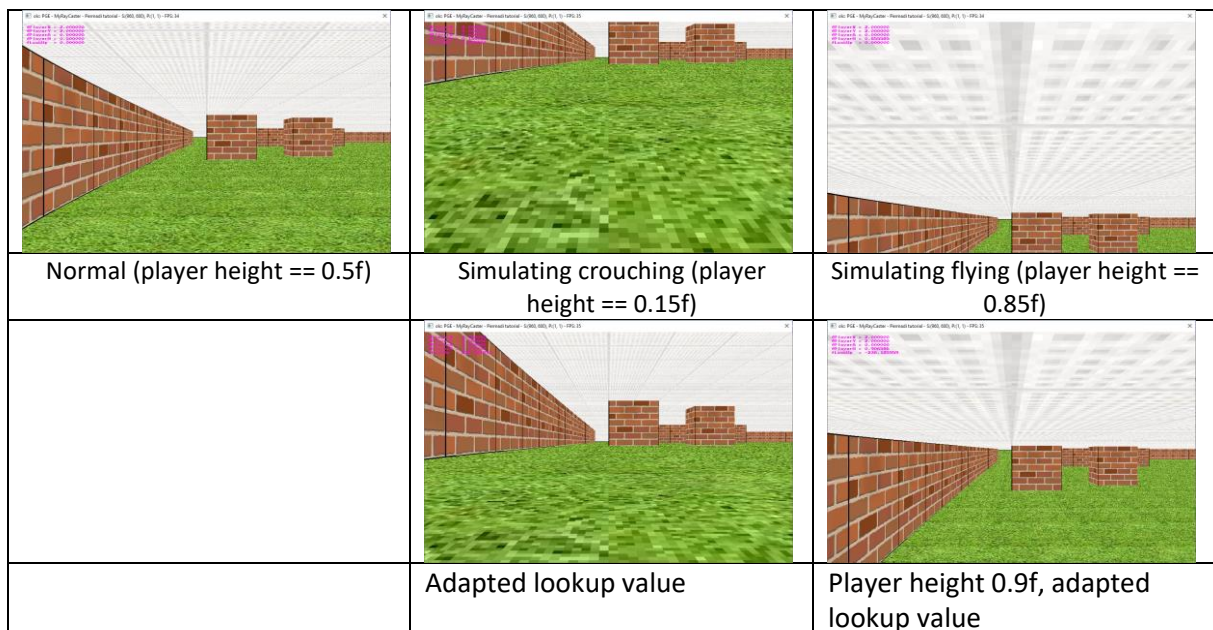
main.cpp 418 int nHalfScreenHeight = ScreenHeight() / 2 + (int)fLookUp	Calculation of value
main.cpp 435 float fCeilProjDistance = ((fPlayerH / float(nHalfScreenHeight - py)) * fDistToProjPlane) / cos(fViewAngle * PI / 180.0f)	Sampling of the ceiling
main.cpp 450 float fFloorProjDistance = ((fPlayerH / float(py - nHalfScreenHeight)) * fDistToProjPlane) / cos(fViewAngle * PI / 180.0f)	Sampling of the floor
main.cpp 481 nWallCeil = nHalfScreenHeight	Default value if no wall

- The results are identical to Permadi's example illustrations⁴:



Part 17 – Vertical Motion: flying and crouching

- Tutorial notes
 - This is a feature that Javidx9 didn't explain in his FPS series.
- Implementation notes
 - For a single layer (non multilevel walls) environment, I got it working after quite some experimentation:



- Note that the horizon is shifting due to changing the player height. If you don't want this, you can compensate for this, by adapting the look up value. This is what is done in the second row of illustrations above. After these experiments I built this into the code to compensate for height changes of the horizon by altering the `fLookUp` value.
- So for simple single level maps this works fine, and the player height is even clamped within the open range `<0.0f, 1.0f>`.
- For multilevel maps the flying and crouching works correct, but the rendering is not sufficient. I already made some corrections to check for the horizon height:

```
//
if (y >= nWallFloor) {
    nDrawMode = FLOOR_DRAWING;
    nDrawMode = (y <= nHorizonHeight) ? CEIL_DRAWING : FLOOR_DRAWING;
} else ...
```

⁴ I googled a couple of sprites to really resemble the Permadi example 😊

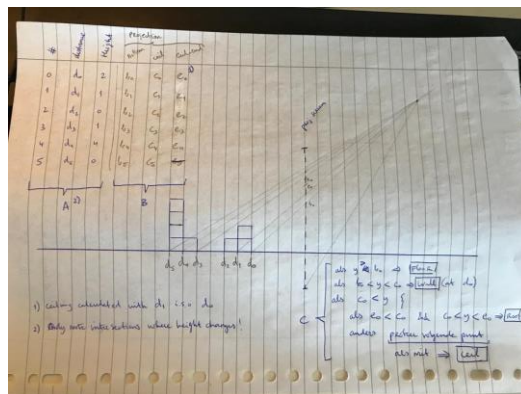
```
//
    } else {
        nDrawMode = CEIL_DRAWING;
        nDrawMode = (y <= nHorizonHeight) ? CEIL_DRAWING : FLOOR_DRAWING;
    }
}
```

After implementing flying and crouching I found that the roofing (the top faces) of the wall blocks isn't correctly rendered yet:

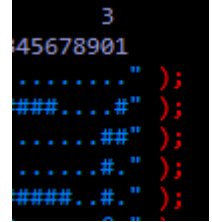
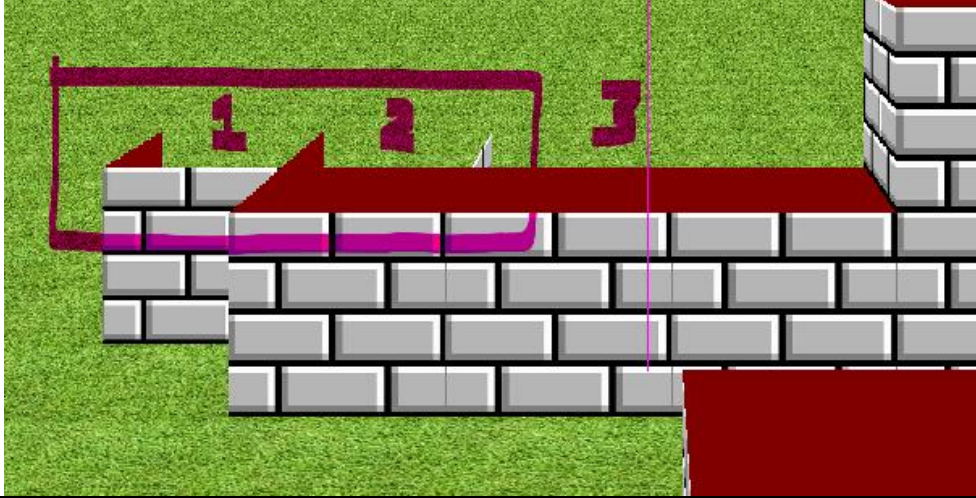


See hand written notes below. I created an algorithm where

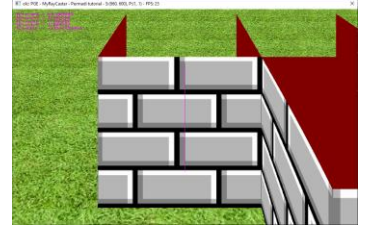
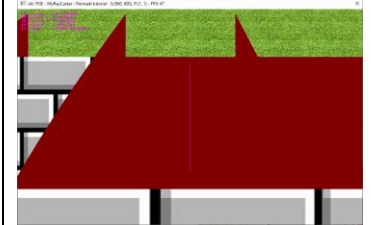
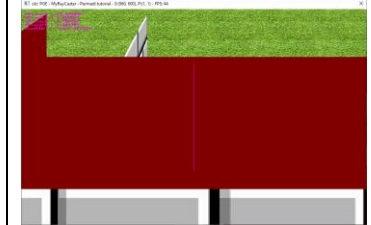
- The adapted DDA function generates a complete list of all the raster points where the height of the heightmap differs (before I took the raster points where the height was > 0);
- After the list is composed along the ray, it is extended with the projected heights of bottom, ceiling and possibly back-ceiling (in order to render roofs).
- This info is used to render floor, wall, roof and/or ceiling per slice.



This works OK, but not at the boundaries of the map. Something's going wrong there...

	
Map detail: '.' = height 0, '#' = height 1	Rendering of that piece of the map. You're looking at the east boundary of the map. Situation 1 corresponds with map location (31, 1), situation 2 with (31, 2) etc

Situation 3 is rendered as expected, but situations 1 and 2 are not. What's happening here?

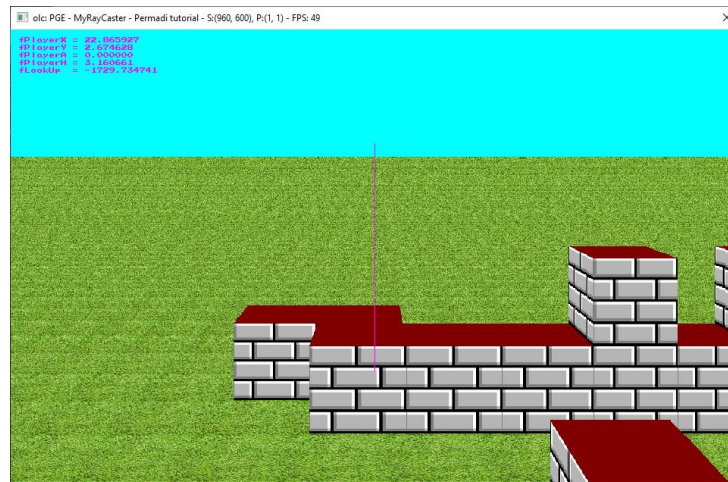
Test set 1:	Test set 2:	Test set 3:
		
I wrote some test code that prints (upon pressing T) the intersection list at the slice denoted by the magenta line		

Output of intersection lists:

<pre>// Test set 1 intersection list: fIntersectX: 31, fIntersectY: 1.5355 , nMapCheckX: 31, nMapCheckY: 1, fDistance: 1.97075 , nHeight: 1, bottom_front: 553, ceil_front: 132, ceil_back: 132 fIntersectX: 31, fIntersectY: 1.5355 , nMapCheckX: 31, nMapCheckY: 1, fDistance: 1.97075 , nHeight: 0, bottom_front: 553, ceil_front: 553, ceil_back: 553 // Test set 2 intersection list: fIntersectX: 30, fIntersectY: 2.58958, nMapCheckX: 30, nMapCheckY: 2, fDistance: 0.970755, nHeight: 1, bottom_front: 1356, ceil_front: 500, ceil_back: 132 fIntersectX: 31, fIntersectY: 2.58958, nMapCheckX: 31, nMapCheckY: 2, fDistance: 1.97075 , nHeight: 0, bottom_front: 553, ceil_front: 553, ceil_back: 553 // Test set 3 intersection list: fIntersectX: 30, fIntersectY: 3.42653, nMapCheckX: 30, nMapCheckY: 3, fDistance: 0.970755, nHeight: 1, bottom_front: 1356, ceil_front: 500, ceil_back: 132 fIntersectX: 31, fIntersectY: 3.42653, nMapCheckX: 31, nMapCheckY: 3, fDistance: 1.97075 , nHeight: 0, bottom_front: 553, ceil_front: 553, ceil_back: 553 fIntersectX: 31, fIntersectY: 3.42653, nMapCheckX: 31, nMapCheckY: 3, fDistance: 1.97075 , nHeight: 0, bottom_front: 553, ceil_front: 553, ceil_back: 553</pre>

Observations:

- The fDistance value of the last point is not correct: because the cached version of the distance was stored into the list, the distance to the back of the cell was reported identical to the distance to the front of the cell. I fixed this and the rendering problem was solved instantly 😊
- For situations where a cell doesn't cross the boundary, an additional entry is created in the list. This is not needed, but it's not harmful for the rendering, so I leave it as it is. → later I used a more clever condition, that an additional entry is only inserted if the list is not empty and the current height > 0. This prevents the unnecessary insertion of additional nodes.



Output after the fix of the same three test sets of intersection lists:

```
// Test set 1 intersection list:
fIntersectX: 23, fIntersectY: 1.7184, nMapCheckX: 23, nMapCheckY: 1, fDistance: 0.134073, nHeight: 1, bottom_front: 19763, ceil_front: 13563, ceil_back: 601
fIntersectX: 27, fIntersectY: 1.7184, nMapCheckX: 27, nMapCheckY: 1, fDistance: 4.13407, nHeight: 0, bottom_front: 802, ceil_front: 802, ceil_back: 489
fIntersectX: 31, fIntersectY: 1.7184, nMapCheckX: 31, nMapCheckY: 1, fDistance: 8.13407, nHeight: 1, bottom_front: 489, ceil_front: 387, ceil_back: 363
fIntersectX: 31, fIntersectY: 1.7184, nMapCheckX: 31, nMapCheckY: 1, fDistance: 9.13407, nHeight: 0, bottom_front: 454, ceil_front: 454, ceil_back: 454

// Test set 2 intersection list:
fIntersectX: 30, fIntersectY: 2.62545, nMapCheckX: 30, nMapCheckY: 2, fDistance: 7.13407, nHeight: 1, bottom_front: 533, ceil_front: 417, ceil_back: 363
fIntersectX: 31, fIntersectY: 2.62545, nMapCheckX: 31, nMapCheckY: 2, fDistance: 9.13407, nHeight: 0, bottom_front: 454, ceil_front: 454, ceil_back: 454

// Test set 3 intersection list:
fIntersectX: 30, fIntersectY: 3.2822, nMapCheckX: 30, nMapCheckY: 3, fDistance: 7.13407, nHeight: 1, bottom_front: 533, ceil_front: 417, ceil_back: 387
fIntersectX: 31, fIntersectY: 3.2822, nMapCheckX: 31, nMapCheckY: 3, fDistance: 8.13407, nHeight: 0, bottom_front: 489, ceil_front: 489, ceil_back: 454
fIntersectX: 31, fIntersectY: 3.2822, nMapCheckX: 31, nMapCheckY: 3, fDistance: 9.13407, nHeight: 0, bottom_front: 454, ceil_front: 454, ceil_back: 454
```

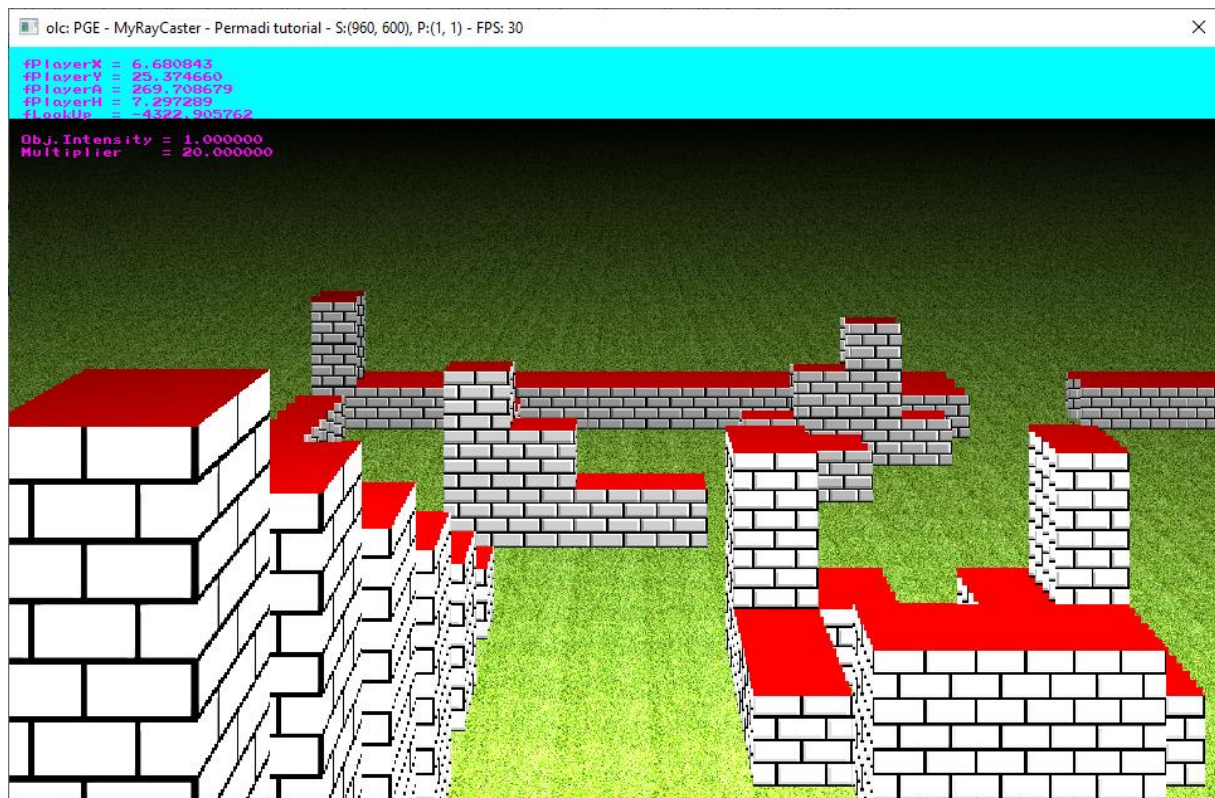
Saved a copy of the working code including test code.

Part 18 – Vertical Motion: combined effects

- Tutorial notes
 - The theory brings nothing new, it's just combining what was already achieved from earlier parts
- Implementation notes
 - I already implemented this. Flying and crouching is controlled with page up resp. page down. Looking up and down is controlled with arrow up resp. down key. Flying and crouching is done with built in functionality to keep the horizon at the same level (by adapting the looking up and down angle).
 - So all these combined effects are implemented in part 17 already.

Part 19 – Shading

- Tutorial notes
 - Simple description on colour theory, and a formula to use for fast shading
- Implementation notes
 - Example shading:



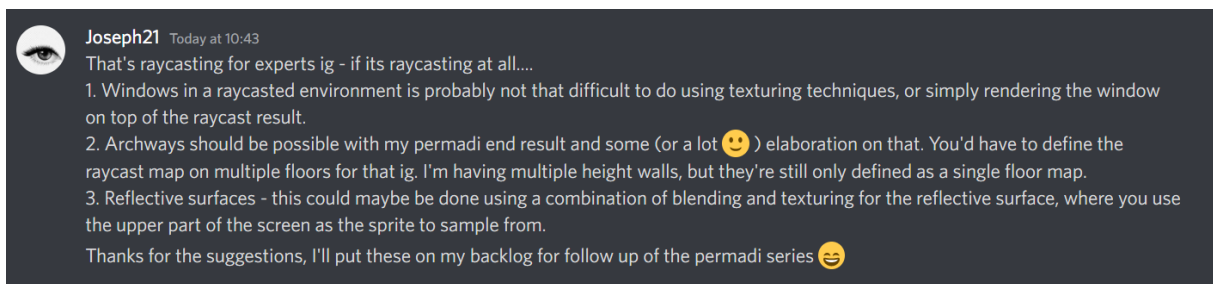
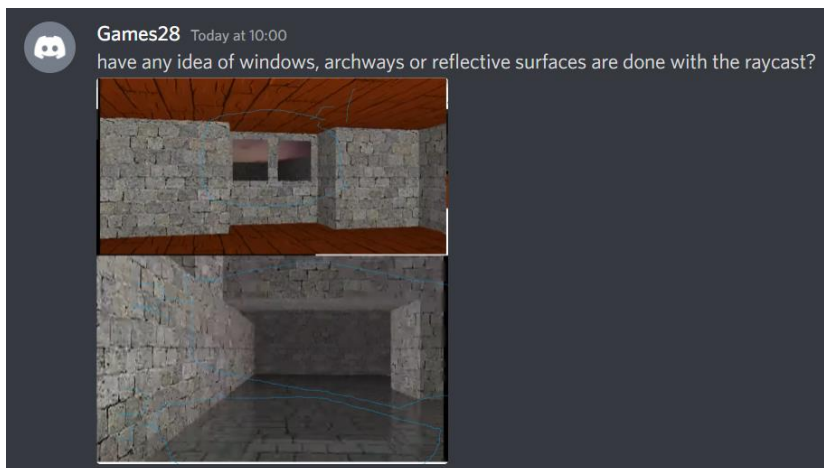
- I found that the background was too dark where the foreground was too light. Solved this by clamping the shading factor between two values (e.g. [0.1f, 1.0f]).

Still to do:

1. There's some nasty bug with the rendering on the boundaries of the map that I have to sort out.	solved, see notes on part 17
2. According to the tutorial there's only shading to cover. I want to do some experimenting with that as well.	done work in progress, see notes on part 19
3. For now I've only done textured ceilings with single level maps. I'm planning on trying to get textured ceilings to work in combination with multiple level maps.	NOTE: looking back this is implemented in the parts 22a-22d
4. And I want the texturing to be more flexible, so i'm looking for a way to give each face of each block its own texture (Sprite *).	NOTE: looking back this is implemented in the parts 23b-23c
5. Want to improve the DDA algorithm still, and see if I can get some performance improvements going - multilevel rendering takes a lot of processing power...	Done, don't believe there's much possible still
6. Add billboard rendering for objects and actors in my ray cast world...	NOTE: looking back this is implemented in part 23a

I think I'm gonna code other stuff than ray casting for a while...

Note 2022-06-07:

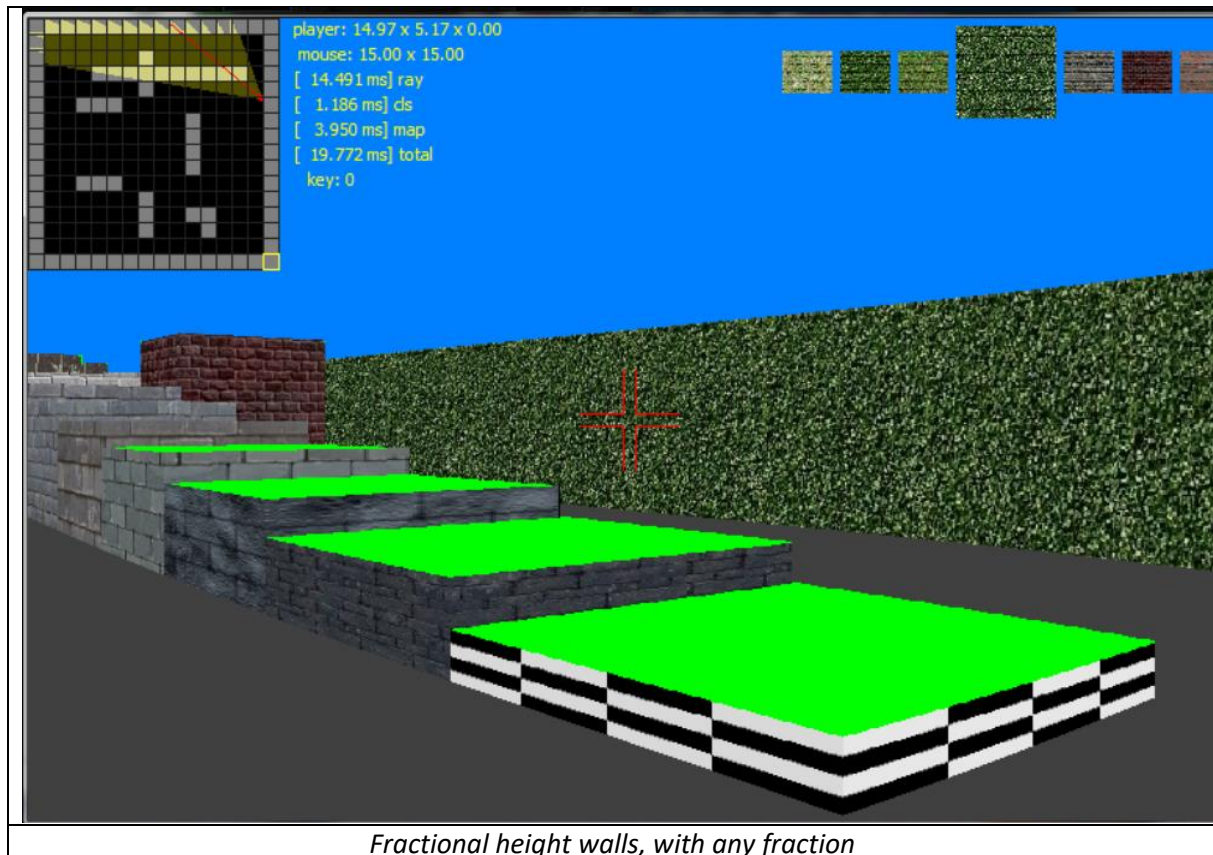


Example – window + reflective floor [the window is fake, it's not see-through]:



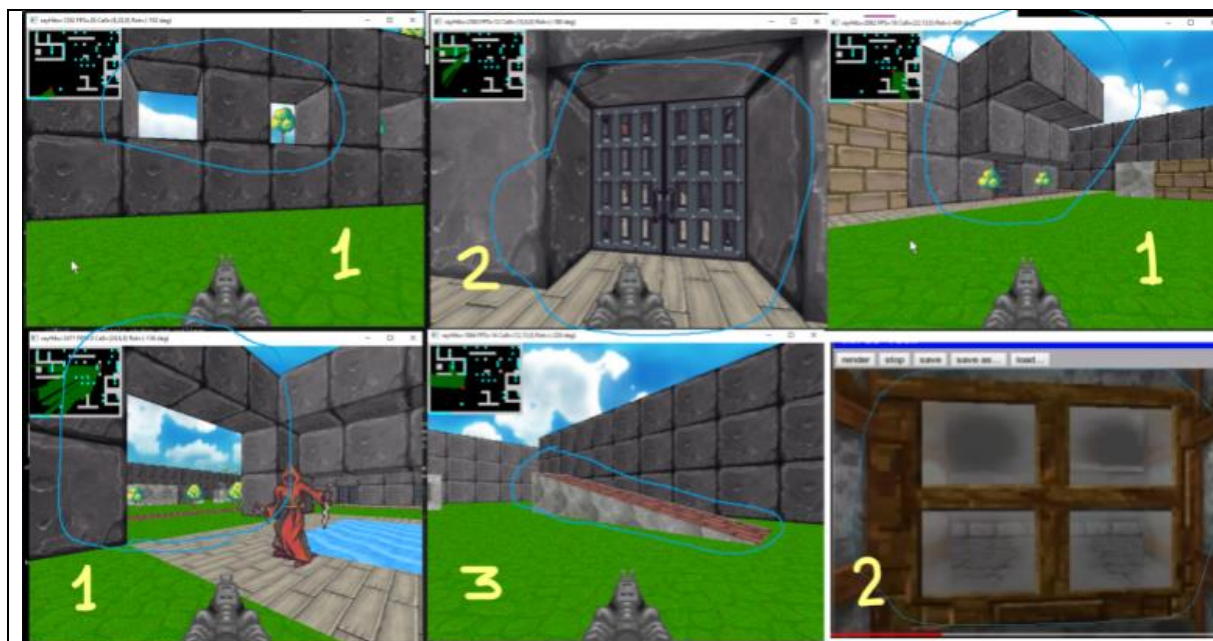
Ambitions

Inspired by the following examples:



I implemented this one in Permadi part 20

And



Marked 1 – gaps, bridges and overhanging walls;

Marked 2 – Doors, fences or windows with partial see through Marked 3 - Ramps
--

Sub Marked 1 – I did an attempt, but couldn't manage it first time

After multiple attempts I got this working ok now 😊 (part 22)

Sub Marked 2 – This may be feasible by rendering these doors, fences or windows as objects (i.e. *after* the background scene) and apply some alpha blending where necessary

After just one attempt I got something working, which needed a bit of fixing of the DDA algorithm – if you hit a transparent block you must record it as a hit point, but not let it influence the DDA.

Sub part 3 – this is future work, I have no idea atm how to implement this in ray casting

Idea 4 – fractional blocks that don't start at relative level height 0.0f

Idea 5 – blocks moving up & down

Idea 6 - Animated wall objects, like opening doors - Implemented April 28, 2023

Idea 7 – improvement on player physics