

# 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 is 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 notes<sup>1</sup> - none
- Implementation notes<sup>2</sup> - 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

---

<sup>1</sup> The tutorial notes refer to the Permadi tutorial

<sup>2</sup> The implementation notes refer to my implementation code files. As you can see upto part 9 of Permadi tutorial there are no implementations (yet), since it's all history, background and theory

### **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  $64^3$  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)<sup>3</sup>
  - 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, its 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 DDA<sup>4</sup>.
- 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.

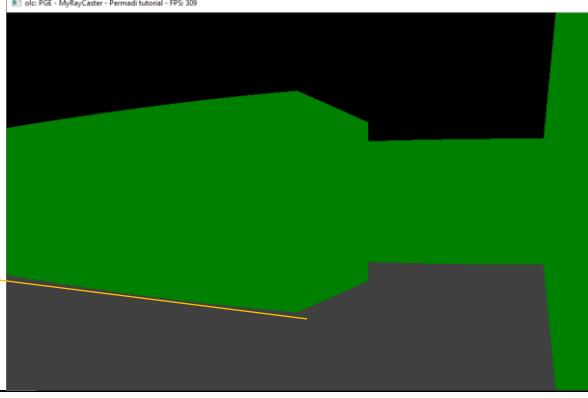
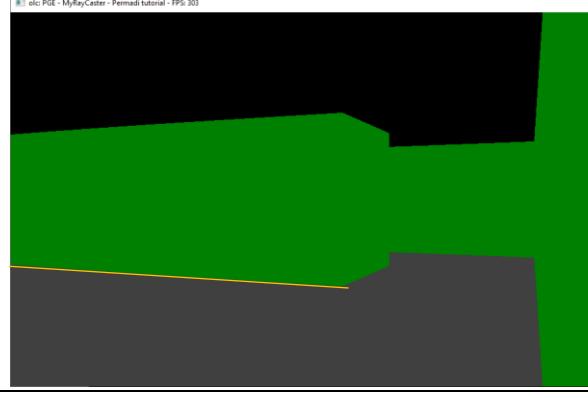
### **Part 8 – step 4 = finding distance to walls**

- 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.

---

<sup>3</sup> 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

<sup>4</sup> See: <https://www.youtube.com/watch?v=NbSee-XM7WA&t=1053s&pp=ygULamF2aWR4OSBkZGE%3D>

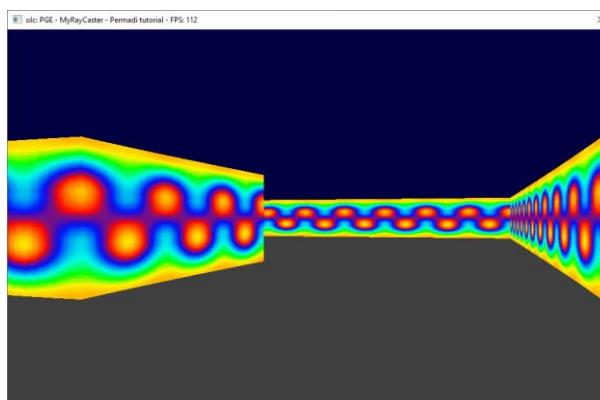
	
Before correcting the fish eye effect...	... and after – the yellow line is added to the rendered image to check on straightness of the walls

### 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  $\frac{1}{2}\sqrt{3} == 0.866$ , so  $320 * 0.866 == 277$  (rounded off)).
- 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 ]
  - [ saved an alternative of the source file using the DDA algorithm instead of ray marching ]

### 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

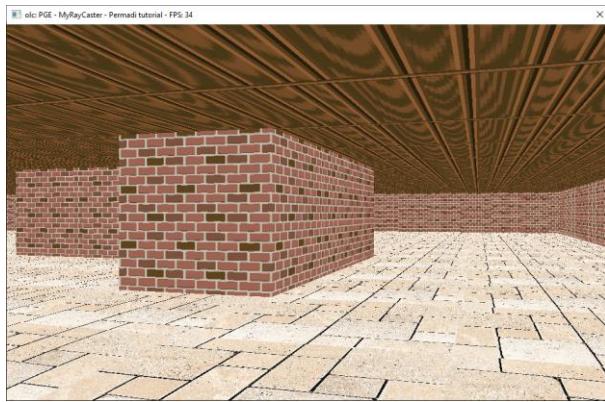
- Tutorial notes
  - This is a straightforward extension of part 11 and 12
- Implementation notes
  - Texturing the ceilings is a pretty straightforward variation on texturing the floors. You only have to take into account the relation of the y screen coordinate and the middle of the screen height...
  - 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<sup>5</sup>.

---

<sup>5</sup> I experimented with multi level raycasting in the past, as an extension of Javidx9's video's. The approach I followed in this Permadi tutorial 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 hit point (or the boundaries of the map). For multilevel ray casting you have to create a list of hit points. So don't stop at the first hit point, but only stop at the boundaries of the map or all hit points where the height of the block before and after the hit point differs.

This can typically be implemented in the ray casting code.

**Step 2** - Using Permadji you can calculate how a wall segment (or a block) projects onto the screen. This technique was already applied in regular (single level) ray casting. This step aims to extend this to point information with the info on how each hit point is projected onto the screen.

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

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

This is typically done in the rendering code.

**Step 1 – build intersection list**  
(Intersection list == hit point list)

**Step 2 – add projection info per hit point**

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 Permadji 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 collision list. This info is needed to render the roof of a block that is viewed from above.

$e_0 = \text{projected top height calculated with distance } d_0, \text{ so } d_0 = d_0, \text{ so the last } e \text{ value is meaningless}$

**Step 3 – render slice using the intersection list**

```
// y is the screen height value (vertical coordinate) of the pixel that is rendered
// values for b, c, d and e are as defined in previous slides

if (y > b)
    render floor
else if (b_i >= y > c_i)
    render wall (using distance d_i)
else if (c_i >= y > e_i)
    render roof
else { // c_i, e_i > 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 explored territory. If you're not, check them out.<sup>6</sup>
- 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 already done 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:

<sup>6</sup> See youtube: <https://www.youtube.com/watch?v=xW8skO7MFYw&t=1001s&pp=ygULamF2aWR4OSBmcHM%3D> and <https://www.youtube.com/watch?v=HEb2akswCcw&t=1820s&pp=ygULamF2aWR4OSBmcHM%3D>

```

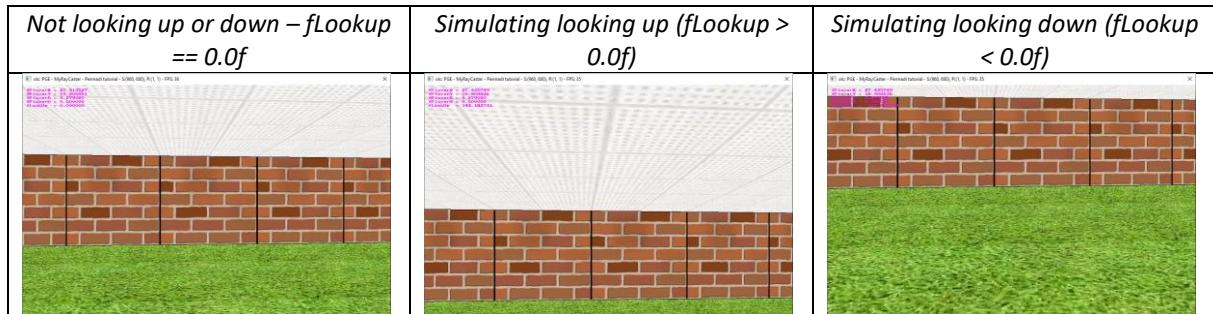
int nHalfScreenWidth = ScreenWidth() / 2;
int nHalfScreenHeight = ScreenHeight() / 2 + (int)fLookUp;

```

- This is where that value is applied:

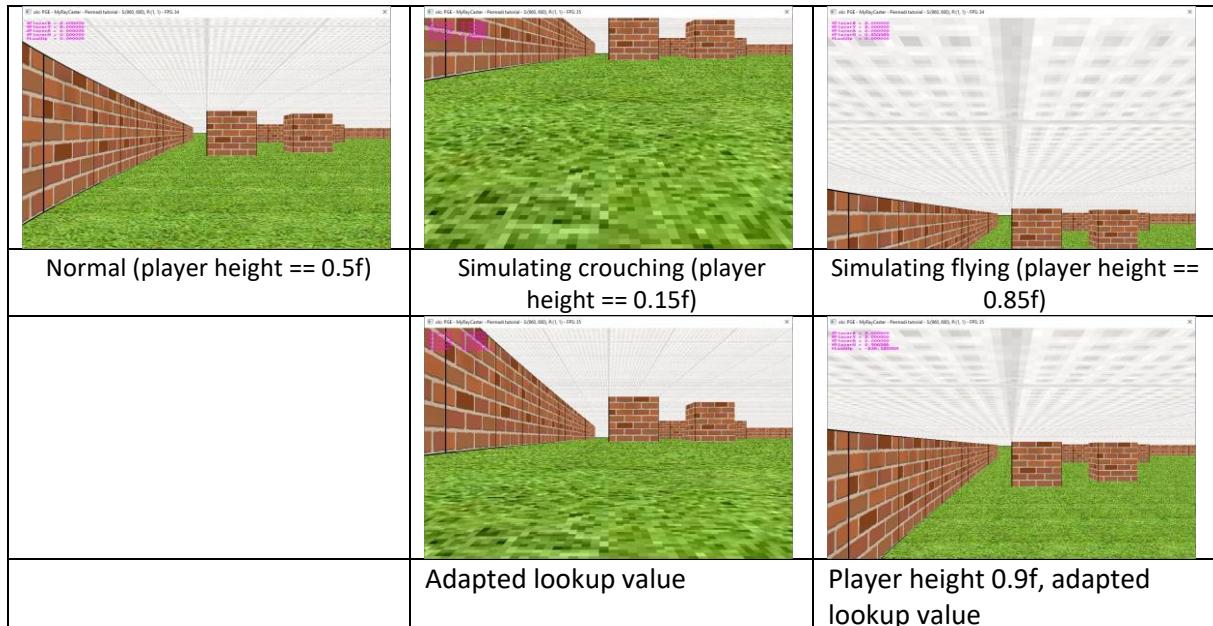
Code file / line / code	What happens there?
main.cpp 418 int nHalfScreenHeight = ScreenHeight() / 2 + (int)fLookUp;	Calculation of value
main.cpp 435 float fCeilProjDistance = ((fPlayerH / float(nHalfScreenHeight - py)) * fDistToProjPlane) / cos(fViewAngle * PI / 180.0f)	Sampling of the ceiling
main.cpp 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
main.cpp 482 nWallFloor = nHalfScreenHeight	Default value if no wall

- The results are identical to Permadi's example illustrations<sup>7</sup>:



## 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.

<sup>7</sup> I googled a couple of sprites to really resemble the Permadi example 😊

- 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 correctly, but the rendering is not sufficient. I already made some corrections to check for the horizon height:

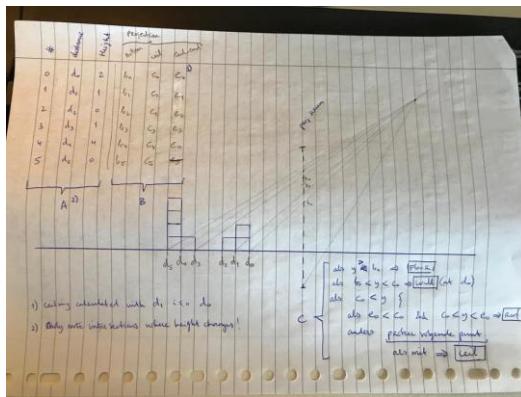
```
//  
    if (y >= nWallFloor) {  
        nDrawMode = FLOOR_DRAWING;  
        nDrawMode = (y <= nHorizonHeight) ? CEIL_DRAWING : FLOOR_DRAWING;  
    } else ...  
  
//  
    } else {  
        nDrawMode = CEIL_DRAWING;  
        nDrawMode = (y <= nHorizonHeight) ? CEIL_DRAWING : FLOOR_DRAWING;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

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



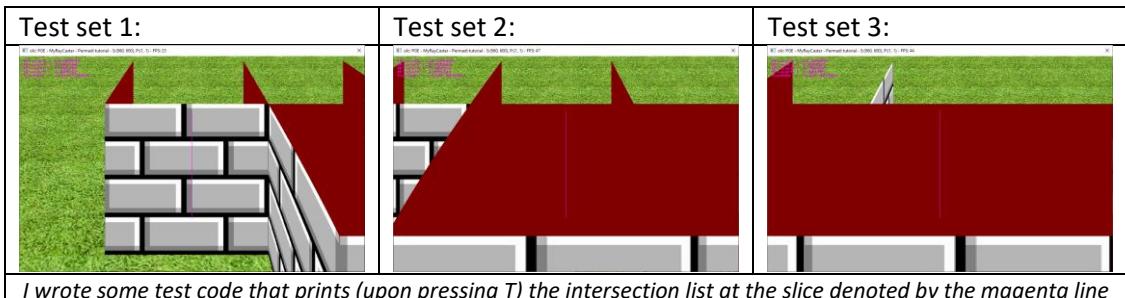
See hand written notes below. I created an algorithm where

- The adapted DDA function generates a complete list of all the raster points where the height of the heightmap differs (before I simply 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.



<pre> 3 45678901 ..... ); ##.## ); .....## ); .....#. ); ####.## ); .....); </pre>	
Map detail: '.' = height 0, '#' = height 1	<p>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</p>

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



Output of intersection lists:

```

// 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

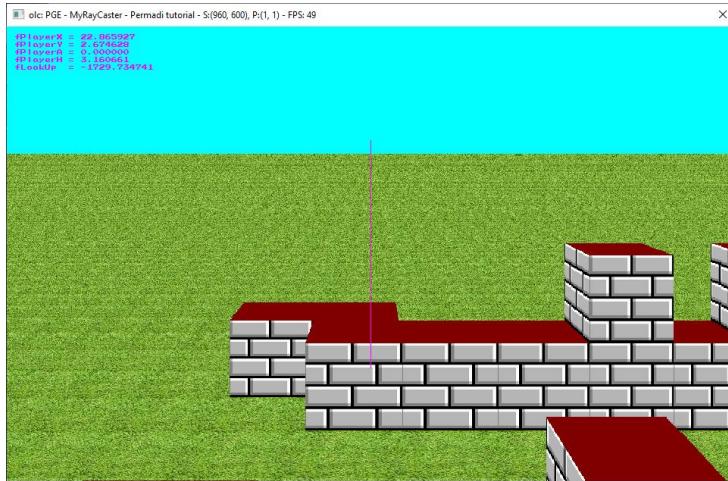
// Test set 2 intersection list:
fIntersectX: 30, fIntersectY: 2.58958, nMapCheckX: 30, nMapCheckY: 2, fDistance: 0.970755, nHeight: 1, bottom_front: 1356, ceil_front: 500, ceil_back: 132
fIntersectX: 31, fIntersectY: 2.58958, nMapCheckX: 31, nMapCheckY: 2, fDistance: 1.97075 , nHeight: 0, bottom_front: 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

```

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.



Test data is outputted upon pressing T along the magenta marked screen slice

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
```

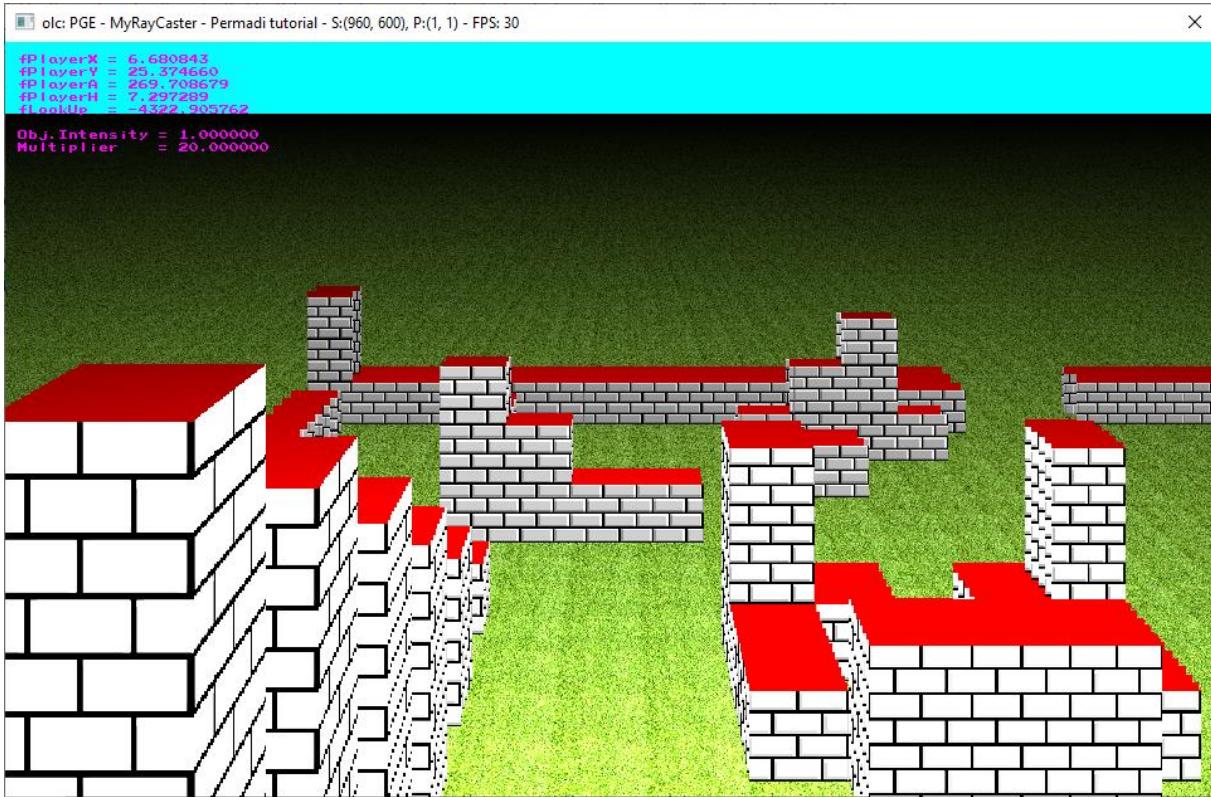
Saved a copy of the working code including test code.

## 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 a constant 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:



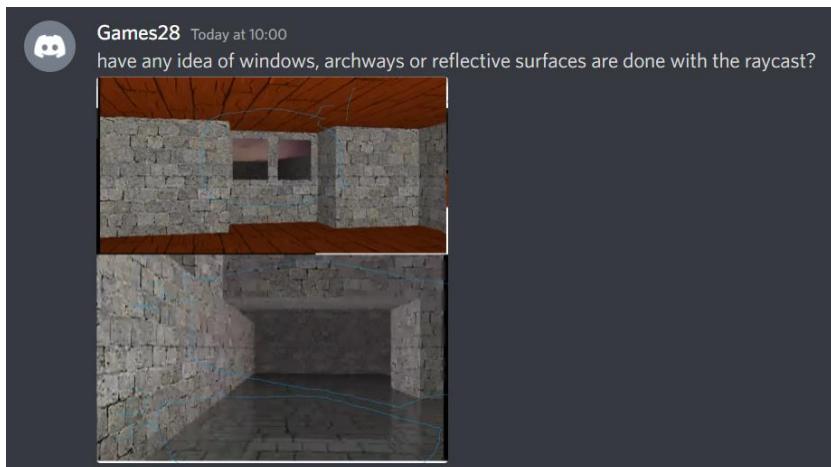
- I found that the background was too dark where the foreground was too light.  
Solved this by clamping the shading factor between two values (e.g. [0.1f, 1.0f]).

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, 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 forward this is implemented in 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 forward this is implemented in 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 forward this is implemented in part 23a

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

Note 2022-06-07:



 Games28 Today at 10:00  
have any idea of windows, archways or reflective surfaces are done with the raycast?

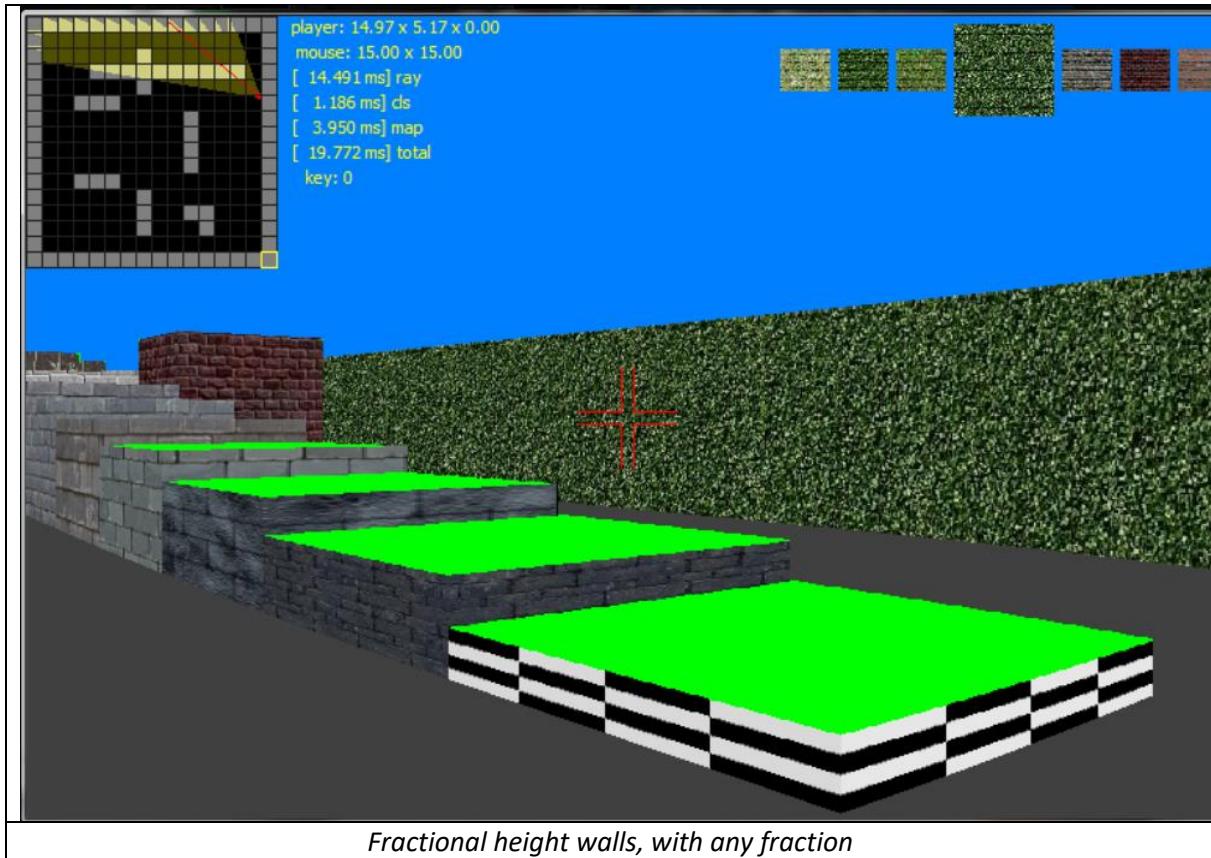
 Joseph21 Today at 10:43  
That's raycasting for experts ig - if its raycasting at all...  
1. Windows in a raycasted environment is probably not that difficult to do using texturing techniques, or simply rendering the window on top of the raycast result.  
2. Archways should be possible with my permadi end result and some (or a lot 😊) elaboration on that. You'd have to define the raycast map on multiple floors for that ig. I'm having multiple height walls, but they're still only defined as a single floor map.  
3. Reflective surfaces - this could maybe be done using a combination of blending and texturing for the reflective surface, where you use the upper part of the screen as the sprite to sample from.  
Thanks for the suggestions, I'll put these on my backlog for follow up of the permadi series 😊

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



## Part 20a – fractional wall heights

Inspired by the following examples:



I implemented this one in part 20a

## Part 20b-c – performance improvements

How to probe OnUserUpdate() – start with the following break down, it probably will suffice, otherwise you can detail on this:

1. UserInput
2. Render loop x
  1. DDA part
  2. Collecting ray hit point data
  3. Render loop y
    1. Collecting data for rendering this pixel
    2. Actually rendering this pixel
3. Rendering minimap
4. Rendering debuginfo

Results (contents of **profiling part 20 for performance improvement 20230417.txt**):

```

Profiling data without any key activity (just starting position)
* test ran for about 1 minute
* in that time the PGE called OnUserUpdate() 1199 times
* the "loop x" frequency corresponds with a screen width of 1200 pixels
* the "loop x+y" frequency corresponds with a screen height of 720 pixels
* > 95% of the time is in the pixel rendering part, maybe zoom in on that
* FPS is about 19.66 on the average (= 1199 frames / 60.987 sec)

profiling stats for OnUserUpdate() and peripheral PGE time
-----
Probe nr: 0 name: user input frequency: 1199 cum. musec: 818768 mean musec: 682.875732 ( 1.681551 % )
Probe nr: 1 name: rndr loop x - DDA part frequency: 1438800 cum. musec: 915937 mean musec: 0.636598 ( 1.881112 % )
Probe nr: 2 name: rndr loop x - coll. hit point data frequency: 1438800 cum. musec: 563082 mean musec: 0.391355 ( 1.156434 % )
Probe nr: 3 name: rndr loop x+y - coll. data f. cur pixel frequency: 1438800 cum. musec: 25275 mean musec: 0.017567 ( 0.051909 % )
Probe nr: 4 name: rndr loop x+y - rendering cur pixel frequency: 1035936000 cum. musec: 46368180 mean musec: 0.044760 ( 95.228996 % )
Probe nr: 5 name: rndr minimap frequency: 1199 cum. musec: 0 mean musec: 0.000000 ( 0.000000 % )
Probe nr: 6 name: rndr debuginfo frequency: 1199 cum. musec: 0 mean musec: 0.000000 ( 0.000000 % )
Probe nr: 7 name: engine time frequency: 1199 cum. musec: 0 mean musec: 0.000000 ( 0.000000 % )

total means (musec): 683.966003
(msec) : 0.683966
( 100.000000 % )
total mean fps : 1462.060913

Process returned 0 (0x0) execution time : 60.987 s
Press any key to continue.

After a couple of optimizations, I got the FPS pulled up to about 32.16 on the average. Afaic this suffices for the moment:
profiling stats for OnUserUpdate() and peripheral PGE time
-----
Probe nr: 0 name: user input frequency: 2012 cum. musec: 1431982 mean musec: 711.720703 ( 3.372966 % )
Probe nr: 1 name: rndr loop x - DDA part frequency: 2414400 cum. musec: 1553313 mean musec: 0.643354 ( 3.658756 % )
Probe nr: 2 name: rndr loop x - coll. hit point data frequency: 2414400 cum. musec: 1304507 mean musec: 0.540303 ( 3.072705 % )
Probe nr: 3 name: rndr loop x+y - coll. data f. cur pixel frequency: 2414400 cum. musec: 46925 mean musec: 0.019435 ( 0.110530 % )
Probe nr: 4 name: rndr loop x+y - rendering cur pixel frequency: 1738368000 cum. musec: 37318984 mean musec: 0.021468 ( 87.903107 % )
Probe nr: 5 name: rndr minimap frequency: 2012 cum. musec: 0 mean musec: 0.000000 ( 0.000000 % )
Probe nr: 6 name: rndr debuginfo frequency: 2012 cum. musec: 625944 mean musec: 311.105377 ( 1.474382 % )
Probe nr: 7 name: engine time frequency: 2012 cum. musec: 173025 mean musec: 85.996521 ( 0.407552 % )

total means (musec): 1110.047119
(msec) : 1.110047
( 100.000000 % )
total mean fps : 900.862671

Process returned 0 (0x0) execution time : 62.553 s
Press any key to continue.

```

## Part 21 – sprites / objects added

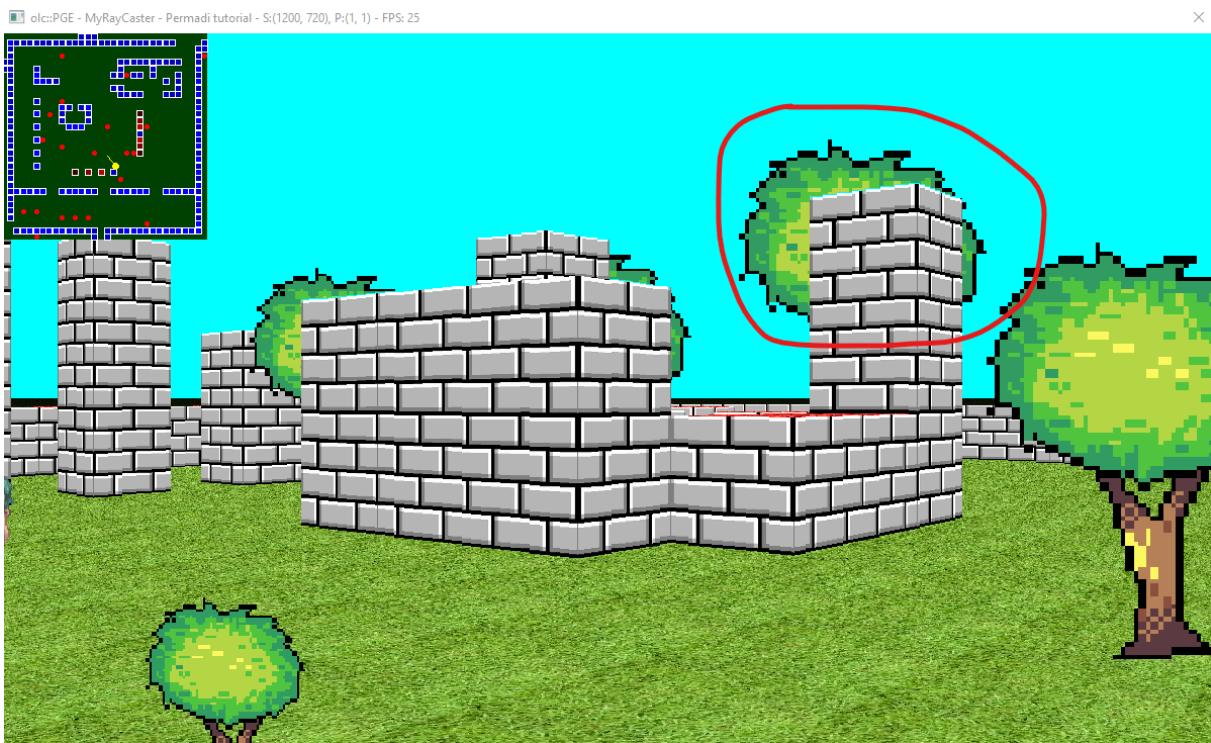
Adaptations to depth buffer (in comparison with the approach Javidx9 uses in his FPS vid part 2):

Line	Text
131	float *fDepthBuffer = nullptr; <span style="color:red">1</span>
235	fDepthBuffer = new float[ ScreenWidth() ]; <span style="color:red">2</span>
748	fDepthBuffer[x] = fCurDistance; <span style="color:red">3</span>
938	if (pSample != olc::BLANK && fDepthBuffer[nObjColumn] >= fObjDist) { <span style="color:red">4</span>
940	fDepthBuffer[nObjColumn] = fObjDist;

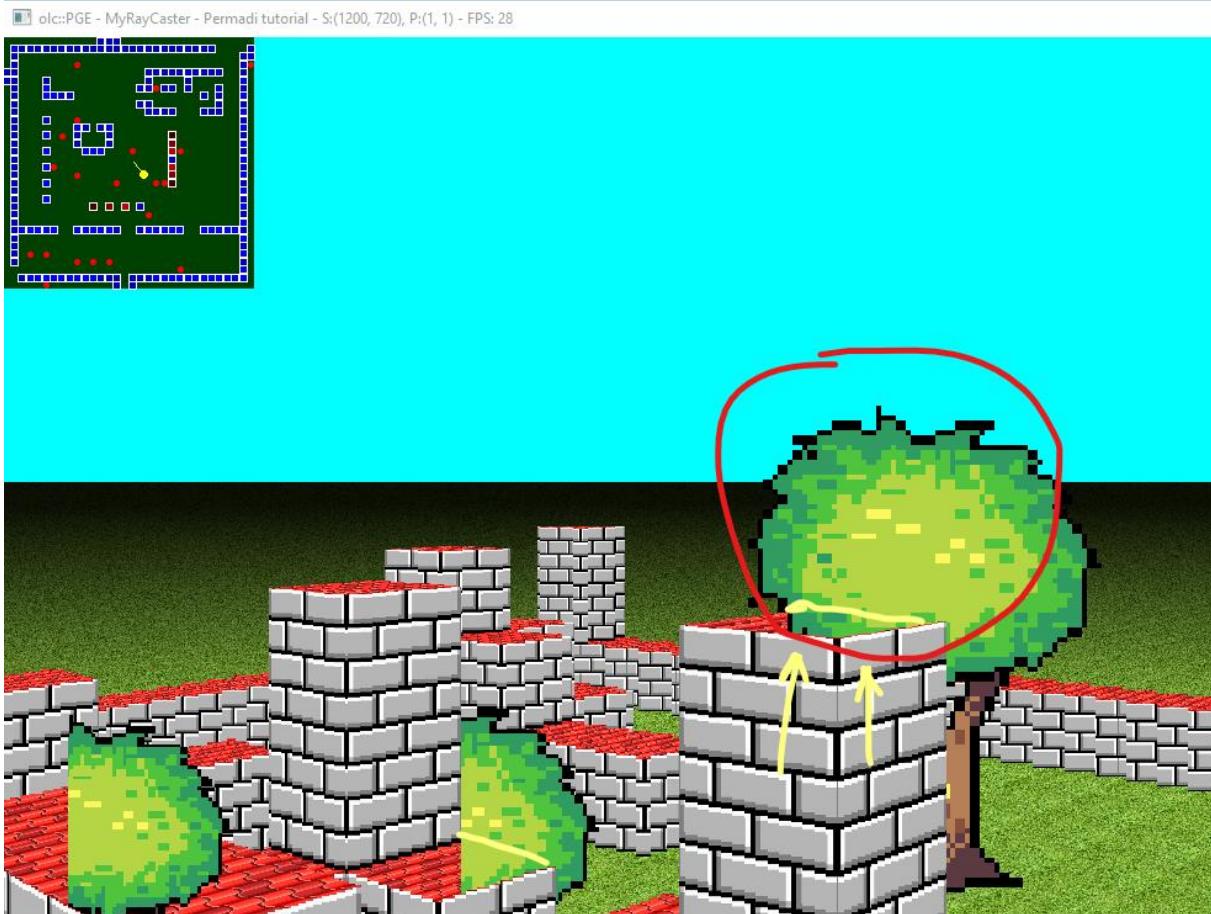
1. Declaration of pointer to depth buffer – no changes needed
2. Initialization – must be altered to make it 2D
3. Filling / updating of the depth buffer must be altered
4. Application of the depth buffer must be altered

After implementation it appears ok, but there are some strange artifacts.

Note this large tree behind the 3-level wall:



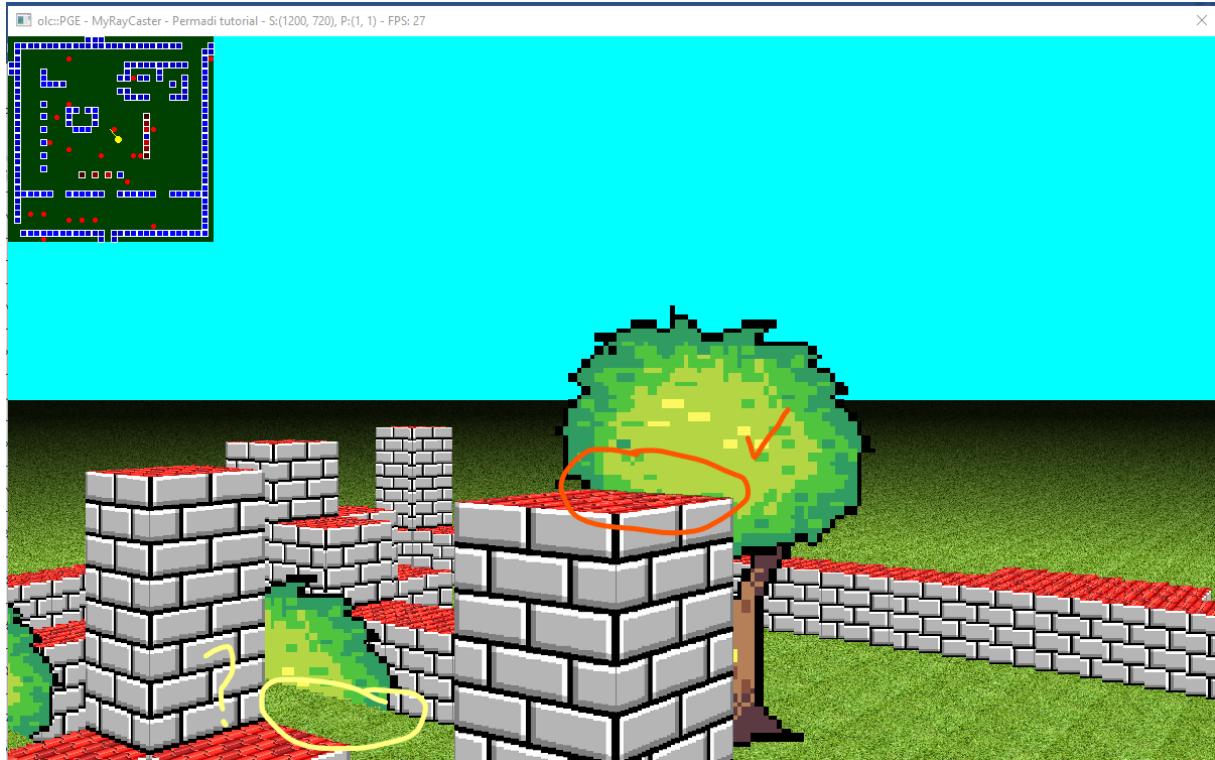
Look what happens if I elevate the player:



It doesn't work properly in combination with roof rendering.

In the code I called these variables `ceil_front` and `ceil_back`, and I erroneously assumed they limited a surface that was seen from below...

So that was pretty easy to solve. But there are other artifacts still:



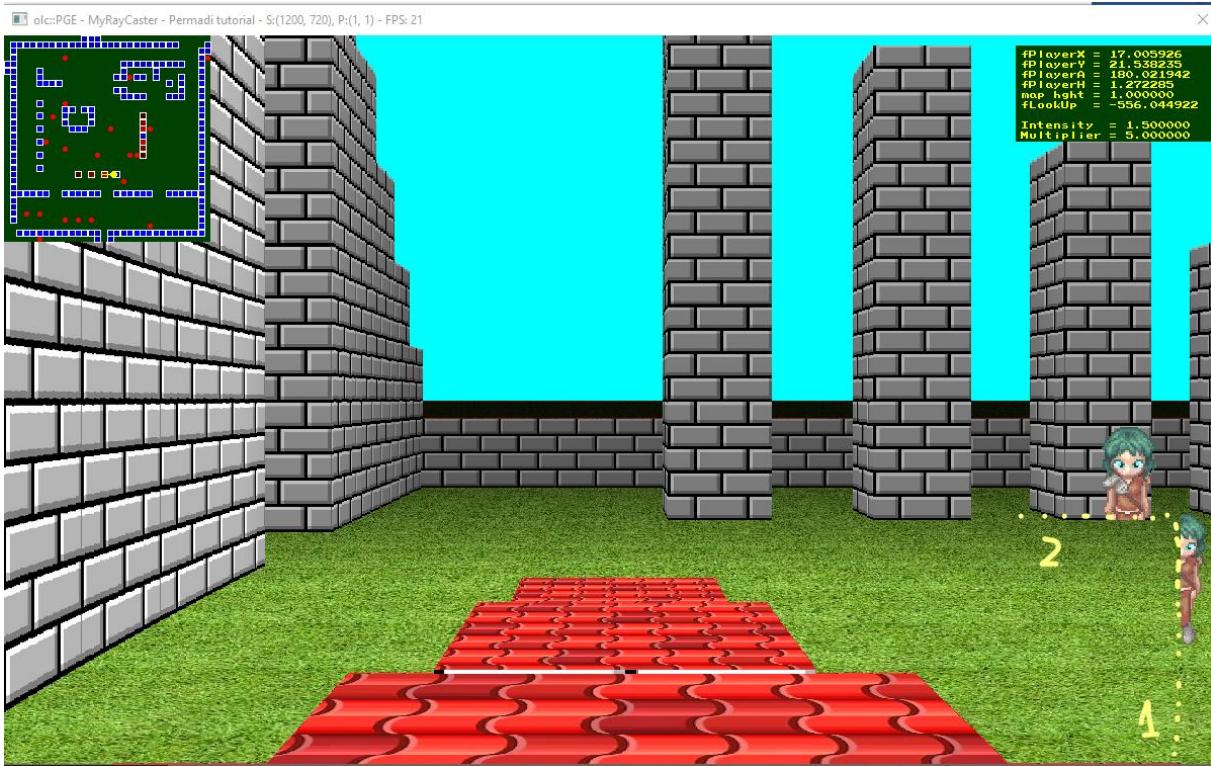
Why isn't the yellow part rendered properly?



These three objects (I tipped them yellow in the minimap) are rendered ok if I'm in front of the fractional wall...



... and get truncated when I'm above or behind the fractional wall



Check this scene:

- the player is right above the height = 1.0f block (see info HUD');

- the forward of the two objects is only half rendered, along the boundary of said block (noted 1);
- the distant of the two objects is only half rendered, along the boundary of the wall rendering behind it (noted 2);

It looks like the depth buffer is filled there with the distance of the level 1 block, even thought it isn't even visible.

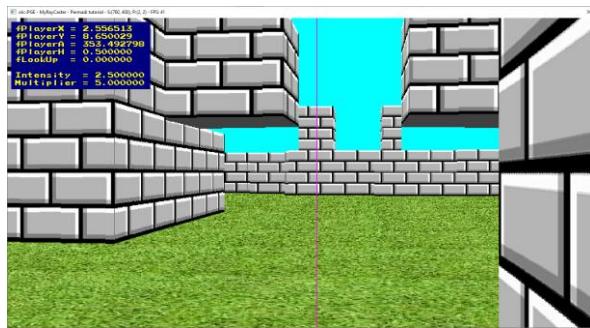
[ Dunno how I fixed this, but I did 😊 ]

## Part 22 – gaps, holes, bridges, overhangs (after refactoring of map representation)

To fix in multilevel ray caster experiment

### 1. Get hit lists per level (instead of just the first hit)

Consider this scene:



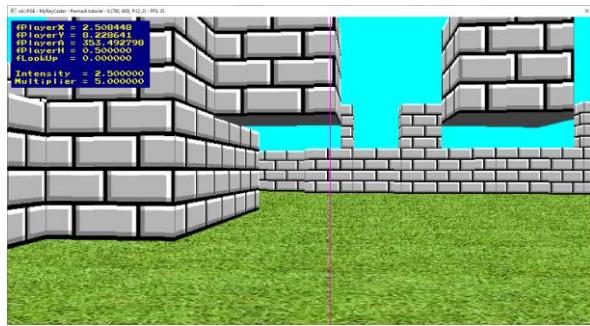
With hit list

```

hit pt: (14, 7.66329) map: (14, 7) dist: 11.4816, hght: 0, lvl: 0 bot - front: 228, back 226 top - front: 171, back 174
hit pt: (14, 7.66329) map: (14, 7) dist: 11.4816, hght: 0, lvl: 1 bot - front: 171, back 174 top - front: 114, back 122

from Y: 399 to Y: 229 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: FLOOR
from Y: 228 to Y: 200 hit point: (14, 7.66329) map location: (14, 7) distance: 11.4816, type: WALL
from Y: 199 to Y: 171 hit point: (14, 7.66329) map location: (14, 7) distance: 11.4816, type: WALL
from Y: 170 to Y: 114 hit point: (14, 7.66329) map location: (14, 7) distance: 11.4816, type: WALL
from Y: 113 to Y: 0 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: SKY
  
```

Clearly there's a far block behind the floating block, but it isn't rendered properly:



Hit list:

```

hit pt: (14, 7.54024) map: (14, 7) dist: 11.4955, hght: 0, lvl: 0 bot - front: 228, back 226 top - front: 171, back 174
hit pt: (7, 7.95957) map: (7, 7) dist: 4.49311, hght: 0, lvl: 1 bot - front: 127, back 140 top - front: -19, back 21

from Y: 399 to Y: 229 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: FLOOR
from Y: 228 to Y: 200 hit point: (14, 7.54024) map location: (14, 7) distance: 11.4955, type: WALL
from Y: 199 to Y: 171 hit point: (14, 7.54024) map location: (14, 7) distance: 11.4955, type: WALL
from Y: 170 to Y: 141 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: SKY
from Y: 140 to Y: 127 hit point: (7, 7.95957) map location: (7, 7) distance: 4.49311, type: CEIL
  
```

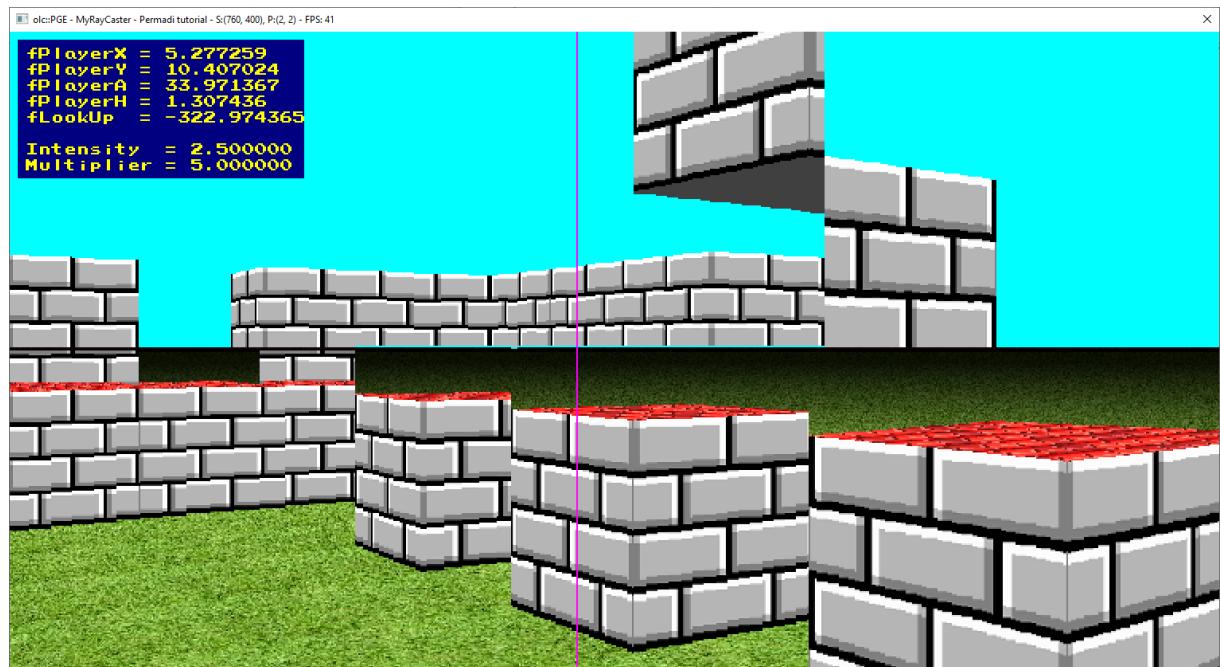
```
from Y: 126 to Y: 0 hit point: (7, 7.95957) map location: (7, 7) distance: 4.49311, type: WALL
```

This is caused by only taking one (actually two) hit point(s) per level. Since the ray on level 2 already found the floating block, it doesn't look further, and the far block behind it is not reported hence not rendered. I need to rebuild the DDA algo to get hit lists per level (again) iso just the first hit.

Note that the hitlists are consistent however 😊

## 2. Next problem

Consider this scene and hitlist:



```
hit pt: (9.41456, 13) map: (9, 13) dist: 4.88004, hght: 0, lvl: 0 bot - front: 375, back 354 top - front: 241, back 236  
hit pt: (12.6057, 15) map: (12, 15) dist: 8.64408, hght: 0, lvl: 1 bot - front: 223, back 222 top - front: 147, back 150  
  
from Y: 399 to Y: 376 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: FLOOR  
from Y: 375 to Y: 241 hit point: (9.41456, 13) map location: (9, 13) distance: 4.88004, type: WALL  
from Y: 240 to Y: 236 hit point: (9.41456, 13) map location: (9, 13) distance: 4.88004, type: ROOF  
from Y: 235 to Y: 200 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: FLOOR  
from Y: 199 to Y: 241 hit point: (9.41456, 13) map location: (9, 13) distance: 4.88004, type: WALL  
from Y: 240 to Y: 226 hit point: (9.41456, 13) map location: (9, 13) distance: 4.88004, type: ROOF  
from Y: 235 to Y: 224 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: SKY  
from Y: 223 to Y: 147 hit point: (12.6057, 15) map location: (12, 15) distance: 8.64408, type: WALL  
from Y: 146 to Y: 0 hit point: (-1, -1) map location: (-1, -1) distance: 22.6274, type: SKY
```

Something's really going wrong with the assembly of the render list here. The green part is built OK, but

- the first yellow line (type FLOOR) shouldn't be there,
- the cyan line has its A and B values switched for some reason, and
- the second yellow line is a duplicate (of three lines up)

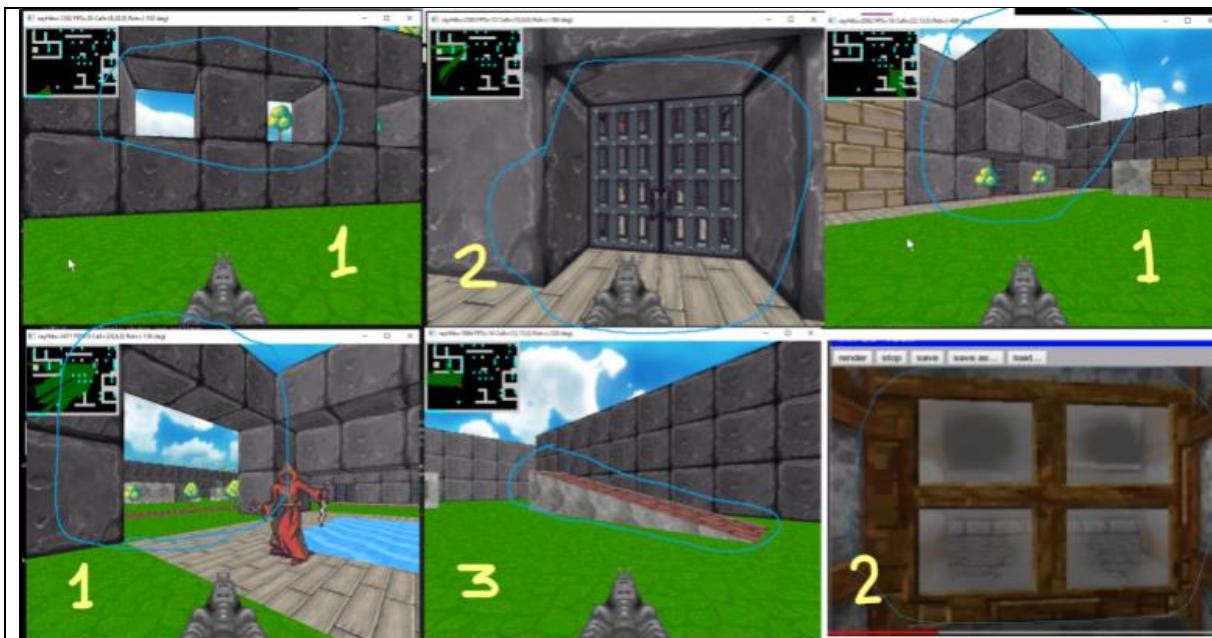
What's going wrong here? And why does it happen here, but not elsewhere?

Summary – I got some fixes, but in the end I ran down due to complexity and adding more bugs than I fixed. Stored this not working result as main - part 21 (holes and bridges) - FAILED ATTEMPT.cpp.

I have to give this a retry in steps:

1. First refactor the code to work with the new map modelling method, where each level is defined separately to make holes and bridges possible. But don't put them in yet!
2. You can build up the height per level by combining both the level and the height value in that level. This way it should be possible to create fractional walls on upper levels.
3. Now introduce the ray casting per level technique like before, differentiating hits only when the level alters along the ray track. This prevents very long ray casting traces.
4. An alternative would be what I did for Games raycaster – just store the info at every intersection, and let the rendering sort out the ray trace. The DDA will be simpler, the rendering possibly a bit more complex.
5. If all this works well, introduce holes, bridges and floating blocks (in that order) and test thoroughly.

## Part 22 – different map representation to support gaps, bridges, floating and overhanging blocks



The above images were given to me as inspiration:

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.

Implemented this in part 23e

## Idea 1

I think depth buffer doesn't always give the right effect, since the height differences are not taken into account. This can cause problems because in the depth buffer walls can have smaller distances than the roof that is on top of that wall (and looked upon).

The fix can be to calculate the angle looking down or up using the screen pixel you are looking through, the horizon height and the field of view:

Let  $f\text{AnglePerPixelRatio} = f\text{FoV} / \text{ScreenWidth}()$

Apply this on the screen y coordinate to get the angle looking up or down. Note that this assumes that pixels are evenly spaced horizontally AND vertically, but that's a safe assumption.

Multiply the cosine of this angle to the distance you pass in DrawDepth(), not only for floors, but also for walls and objects. This should compensate for additional distance due to height differences, and thus yield a better approximation of the actual distance / depth of the pixel drawn (consequently removing the erroneous effects).

*Implemented this, but still have glitches in rendering of ceilings and roofs in some places. It only occurs in certain conditions, but didn't find how to fix this properly yet.*

Look at this scene:



The rendering of roofs is buggy where there's a transparent block. It is OK where there's a regular block.

Under the big “bridge” it's failing where there are fractional blocks... What is going wrong??

## Idea 2

I could get rid of the atan2f() call for wall rendering, by letting the DDA function calculate and return the face direction that was hit. This should be relatively easy, since you have the x-step and y step (both are either positive or negative) and you know if the hit point is on a horizontal or vertical grid line.

Look at below table. When hitting a vertical grid line, you can determine EAST or WEST by looking at the x-step value. When hitting a horizontal grid line, you can determine NORTH or SOUTH by looking at the y-step value:

		<i>hit point is on ... grid line</i>	
		<i>horizontal</i>	<i>vertical</i>
<i>x-step</i>	<i>&lt; 0</i>		EAST
	<i>&gt; 0</i>		WEST
	<i>==</i>		
	<i>0</i>		-
<i>y-step</i>	<i>&lt; 0</i>	SOUTH	
	<i>&gt; 0</i>	NORTH	
	<i>==</i>		
	<i>0</i>	-	

In the DDA algorithm all this info is available, and it comprises max. two easy comparisons per hitpoint. I expect the saving of the atan2f() call to be beneficial for performance.

Upon introducing block associated objects, this only gains in relevance...

*Implemented this, and it works fine. It doesn't provide much performance gain though.*

## Ambitions



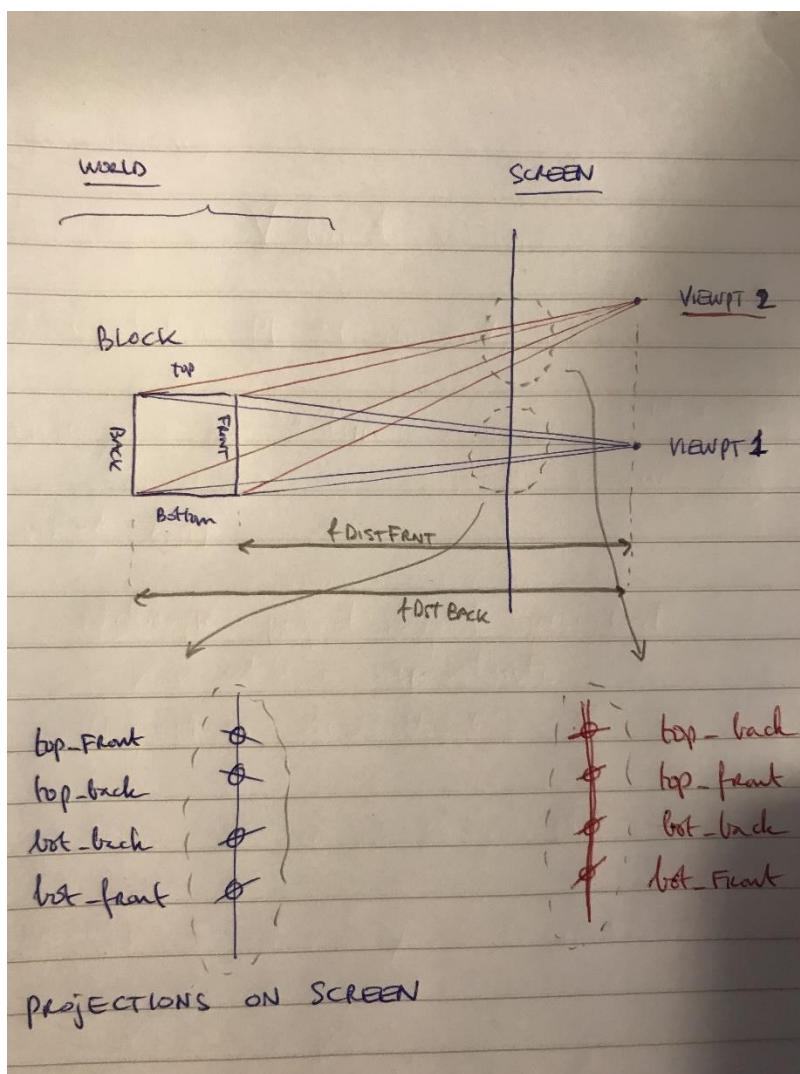
More ideas:

1. Sub part 3 – this is future work, I have no idea atm how to implement this in ray casting
2. Idea 4 – fractional blocks that don't start at relative level height 0.0f
3. Idea 5 – blocks moving up & down
4. Idea 6 - Animated wall objects, like opening doors - Implemented April 28, 2023
5. Idea 7 – improvement on player physics
6. Have static non transparent wall face objects be animated as well (so that you can animate lava)
7. Have columns of blocks grow and shrink gradually i.e. have their height be dynamically changing (by f.i.  $0.1f * fElapsedTime$  to raise or shrink 1 block per every 10 seconds) → have been working on this in recent version (after 23g)
8. For the scaling –
  - a. go back to the initial object / sprite rendering conform Javids part 2 fps vid.
  - b. Set scaling on the difference between bottom and ceiling after calculating it
  - c. Determine what must be change to compensate for player height.

## Annex – Explanation of projection of wall bottom and top onto screen slice

 Games28 Today at 01:46  
could you tell me how these variables top\_back, bot\_back, top\_front, bot\_front, fDistfrnt, fDistback works. like how you did with the calculatebottomandtop function?  
its was super useful

 Joseph21 Today at 05:48  
Each block in the map has a top and a bottom side, and has a front and back side (it also has a left and right side, but that's not relevant since we are rendering the block in slices) (edited)  
These four variables denote the projections of the blocks corner points onto the slice: they are the y coordinates in pixel space:  
\* top\_back – the projection (only the y coordinate) of the point at the top and at the back of the block onto the screen slice  
\* bot\_back – the same for the point at the bottom and at the back of the block  
\* top\_front, bot\_front – the same but for the points at the top and front resp. at the bottom and front of the block (edited)  
Like this:





Joseph21 Today at 05:57

^^ This example show what these variables mean - I drew two player viewpoints and detailed below how the projections on the screen work out:

Looking at viewpoint 1 , the order of these variables along the slice is

1. top\_front
2. top\_back
3. bot\_back
4. bot\_front (edited)

Since top\_back is projected lower on the screen than top\_front, the top side of the block (i also call it the roof) is not visible and must **not** be rendered. The same holds similarly for the ceiling (the bottom side of the block). Along the slice the distance between top\_front and bot\_front must be rendered with wall texture. (edited)

Now consider viewpoint 2, the order of the variables along the slice is different (since the viewpoint is above the block):

1. top\_back
2. top\_front
3. bot\_back
4. bot\_front

The ceiling (bottom face) is still not visible and should not be rendered. However, the top side of the block (aka the roof) is visible, since top\_back is higher on the slice than top\_front. So part of the slice between top\_back and top\_front must be rendered with roof texture.

As before, the part between top\_front and bot\_front must be rendered with wall texture. (edited)

The fDistFrnt and fDistBack variables contain the distances to the front resp the back of the block, from the players viewpoint. I denoted this in the diagram above. It is used to draw pixels using the depth buffer



Joseph21 Today at 06:10

As an exercise - work out for yourself what the order of these variables would be if the viewpoint is *below* the block, and what that order means for the rendering.