Notes – 3DSage: Let's Program Doom – part 1

Joseph21 – 2022-08-16

Youtube: https://youtu.be/huMO4VQEwPc

- First the Raycaster series
- Now the DOOM series
- Maybe the Quake engine after this one;[1]

The Duke Nukem build engine used a portal system to break up the levels and draw them in the correct order. The DOOM engine used a BSP (Binary Space Partitioning) for that;

Original 3DSage code is written in C and OpenGL

All we want to do in this part 1 tutorial is isolate *this* (see image – yellow parts). They are called *sectors* – just a simple object defined by walls, projected into three dimensions, drawn with vertical lines, that we can move around in 3d space:
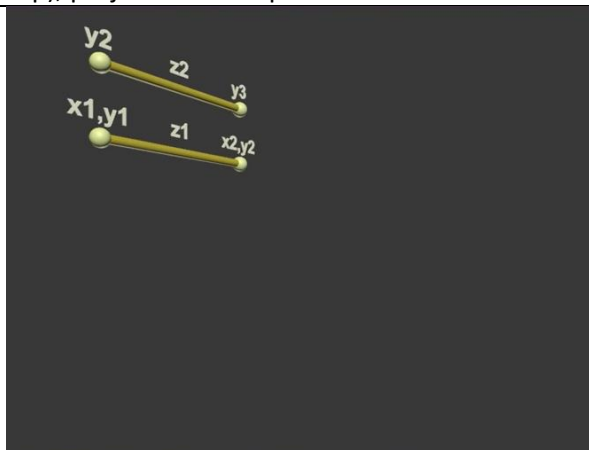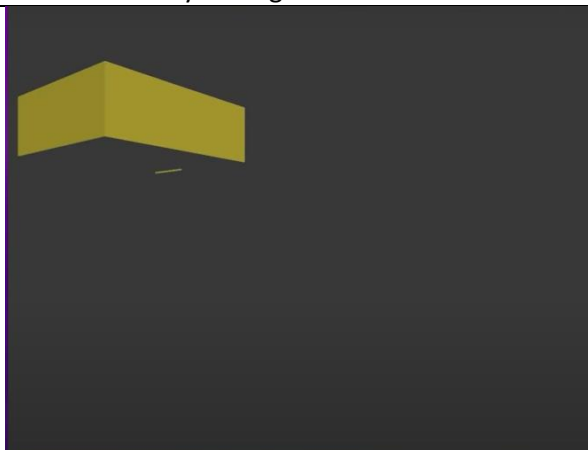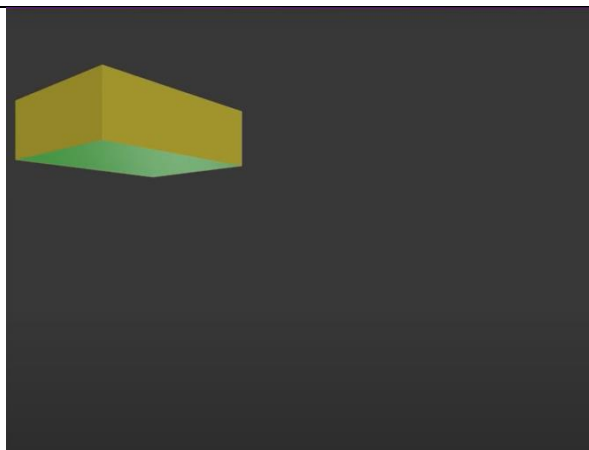

We are not gonna move the player around in the 3d world, but we are gonna move the world around the player:

So when the player moves in the positive right x direction, we are actually moving all the points in the level to the left, to give the same look. And same for the y and the z (up) direction.

---
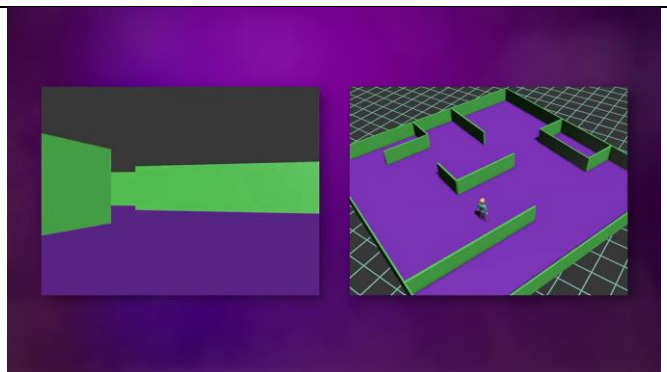
[1] All made by idSoftware in the '90s

Every wall is gonna have two lines (bottom and top), projected in 3d space …[2]	… and we can use a for loop to draw vertical lines all the way through:
And we can use vertical lines to draw the bottom or top surface of a sector	So our perspective is slightly gonna be off, but it's good enough for an early nineties 3d game engine

Fase 0 – initial setup

A few notes on getting started:

- For the keys he[3] uses < and > for strafe left and right. I use *arrow keys* left and right.
- I let PGE handle the pixelscale for me, not gonna do it in the code. I suspect I don't need GLSW and GLSH, but we'll see;
- He times using glut ticks (integers representing 1/1000 of a second) and waits for 50 ticks to start a new frame, I time using the PGE float (seconds), and wait for 0.05f seconds.
- The Keys struct is not needed, since I can query the PGE key state.
- He has the glut environment set up to have the screen origin *in the bottom left corner*. I adapted myPixel() to achieve the same result: myPixel(x, y, col) will draw at screen coordinate (x, SH – 1 – y), where SH is screen height;
- It actually renders at 20 frames per second, while the OnUserUpdate() iterates at much higher rates.

NOTE – I suspect that

---

[2] Note that: the horizontal lines share the same z-coordinate, and the vertical lines share the same x coordinate
[3] Everywhere I use he or his or him, I'm referring to 3DSage, the creator and publisher of the video.

- init() is a mandatory function in glut that has the same meaning as OnUserCreate();
- display() is a mandatory function in OpenGL that has the same meaning as (parts of) OnUserUpdate();

Having these definitions for *screen/window size and resolution*…

```
#define res        1                  // 1 = 160x120, 2 = 320x240, 4 = 640x480
#define SW         160*res            // screen width
#define SH         120*res            // screen height
#define SW2        (SW/2)             // half of screen width
#define SH2        (SH/2)             // half of screen height
#define pixelScale 4/res             // pixel scale
//#define GLSW      (SW*pixelScale)   // window width
//#define GLSH      (SH*pixelScale)   // window height
```

… I construct the screen/window as

```
Construct( SW, SH, pixelScale, pixelScale )
```

So the screen will have 160 x 120 logical pixels, and each logical pixel is built up from res x res physical pixels. Note that constants GLSW and GLSH are needed in a glut environment but not in the PGE.

Result of Fase 0
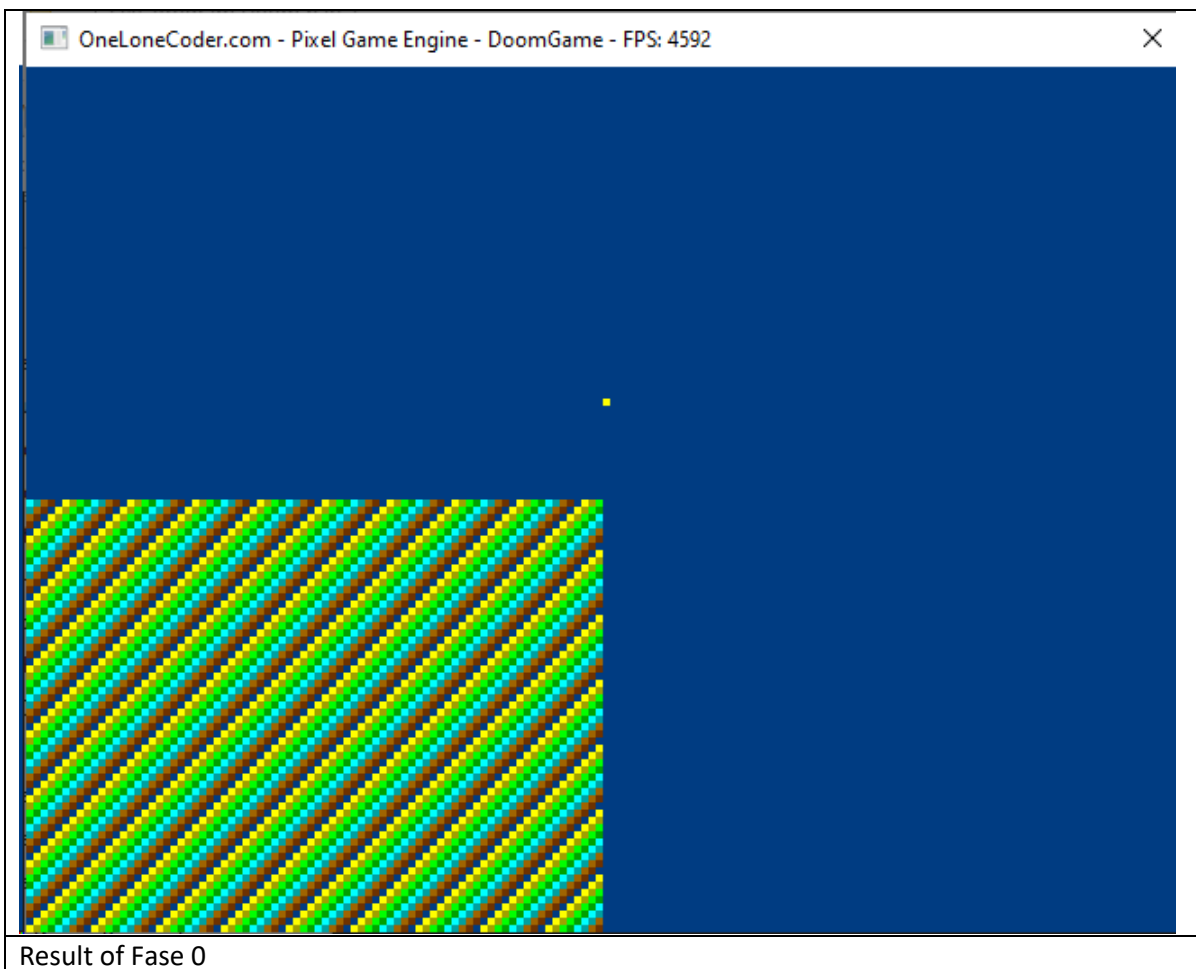
Fase 1a – basic rotation and movement

He's working with degrees and converts them to radians where necessary using a lookup table approach

World transform operations [although he doesn't state it, that's what's happening here]

```
void draw3D() {
    int wx[4], wy[4], wz[4];                    // local x, y, z values for the wall
    float CS = M.cos[P.a], SN = M.sin[P.a];     // local copies of sin and cos of player angle
    // offset bottom 2 points by player
    int x1 = 40 - P.x, y1 =  10 - P.y;          )  1
    int x2 = 40 - P.x, y2 = 290 - P.y;          )
    // world X position
    wx[0] = x1 * CS - y1 * SN;                   }
    wx[1] = x2 * CS - y2 * SN;                   }
    // world Y position (depth)                  }  2
    wy[0] = y1 * CS + x1 * SN;                   }
    wy[1] = y2 * CS + x2 * SN;                   }
    // world Z position (height)
    wz[0] = 0 - P.z + ((P.l * wy[0]) / 32.0f);  ]
    wz[1] = 0 - P.z + ((P.l * wy[1]) / 32.0f);  ]  3
    // screen x, screen y position
    wx[0] = wx[0] * 200 / wy[0] + SW2; wy[0] = wz[0] * 200 / wy[0] + SH2;  )
    wx[1] = wx[1] * 200 / wy[1] + SW2; wy[1] = wz[1] * 200 / wy[1] + SH2;  )  4
    // draw points (unless it is off screen)
    if (wx[0] > 0 && wx[0] < SW && wy[0] > 0 && wy[0] < SH) { myPixel( wx[0], wy[0], 0 ); }
    if (wx[1] > 0 && wx[1] < SW && wy[1] > 0 && wy[1] < SH) { myPixel( wx[1], wy[1], 0 ); }
}
```

Sub 1 - The initial sector bottom line is put at points (40, 10) and (40, 290). However since the world rotates and translates around the player, we offset these coordinates by the players location: (40 – P.x, 10 – P.y) and (40 – P.x, 290 – P.y). The effect is that the sector is located so that the world's origin is at the player's location.

Sub 2 - Then we rotate it around the player to simulate the effect that the player itself is rotated. This is done using the 2d rotation matrix approach…

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

… but without matrix multiplication:

- For the x values this is wx[0] = x1 * CS – y1 * SN;
- For the y values this is wy[0] = y1 * CS + x1 * SN;

Sub 3 - Then we do the z-translation: wz[0] = 0 – P.z;   (comparable to the x and y translations w.r.t. the player). [Since z is not used until now, we could do this at the same time where x and y are offset…]

Depth calculation [in draw3D()]

Sub 4 - The further the point is away from the player, the closer it should be to the center of the screen. All we have to do is divide the worlds x and z position by the y value (the depth in this case)

```
        // screen x, screen y position
        wx[0] = wx[0] * 200 / wy[0] + SW2; wy[0] = wz[0] * 200 / wy[0] + SH2;
        wx[1] = wx[1] * 200 / wy[1] + SW2; wy[1] = wz[1] * 200 / wy[1] + SH2;
```

"I'm timesing it by 200 to affect the field of view, but you can change it if you want" – this apparently is some empirically determined number representing the distance to the projection plane ig.[4]

SW2 and SH2 are half of screen width resp. screen height. "without the SW2 and SH2 offsets, it would be scaled to our origin (which is the bottom left corner of the screen). We want it to scale to the center of the screen."

Simulate looking up or down [in draw3D()]

Before:

```
        // world Z position (height)
        wz[0] = 0 - P.z;
        wz[1] = 0 - P.z;
```

After:

```
        // world Z position (height)
        wz[0] = 0 - P.z + ((P.l * wy[0]) / 32.0f);
        wz[1] = 0 - P.z + ((P.l * wy[1]) / 32.0f);
```

Multiply our depth (wy[]) with the variable P.l, and divide by 32 to keep it in scale[5]. Basically this value (i.g. the part after the '+') will be much higher the further the point is.

Fase 1b – basic wall

Now we draw the bottom two points, let's connect it with a line. The function drawWall() will eventually draw the top line too and connect it to the bottom line. Note that the parameters to drawWall() are in screen space.

In the loop to draw a line: "I added the addition of 0.5 value to help for rounding issues"

If we draw the bottom line, now also draw the top line. The top line has the same x, y values in world space, it's just pushed up in the z value[6]

If we have a bottom y point and a top y point, we can use a for loop to draw between the two.

Fase 1c – clipping screen boundaries

"We don't wanna ever draw behind the player."

Clipping against the screen boundaries is not difficult. Just check if the pixel coordinates are in [0.0f, ScreenWidth – 1 resp. ScreenHeight – 1]. That's all.

Result of fase 1c:

---

[4] In real 3D you after the projection you also scale into view: the projection operation leaves the screen x and y coordinates in the domain [-1.0f, +1.0f] and you want to translate that to [0.0f, +1.0f]. So in this 200 there's a bit of aspect ratio, and a bit of screen size (width in this case), and the FOV will have some role…

[5] What is this scale he speaks about at this point?

[6] You have to keep your spaces straight for this video – in *world space* x and y axis cover the horizontal plane, and the z axis elevates up and down, whereas in *screen space* the x and y axis cover the vertical (projection) plane. This could easily lead to misunderstanding.
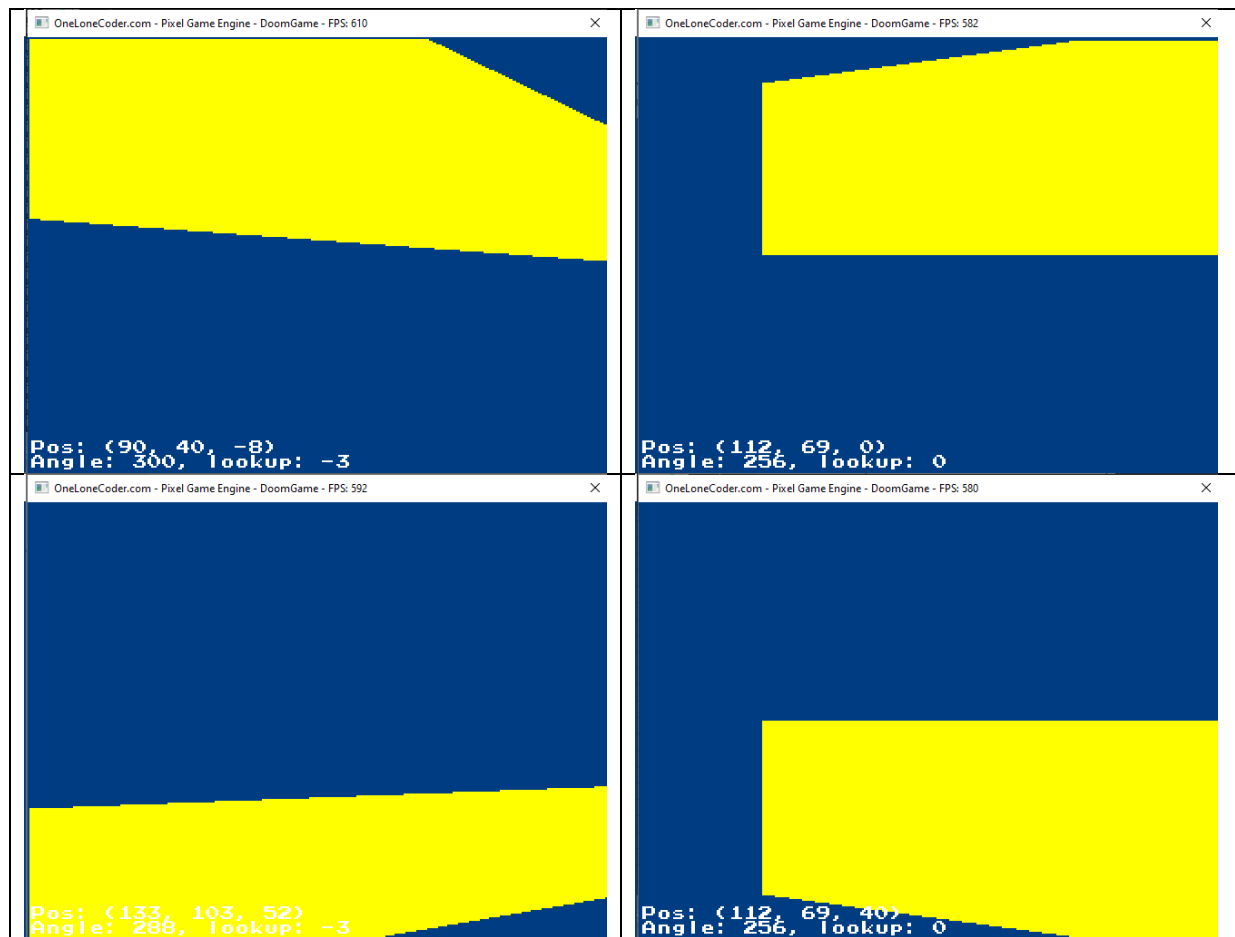
It's not very visible, but there are background pixels at the left, right and upper, lower window boundary, indicating that the clipping is working correctly.

Fase 1d – clipping near plane

Then we still need to fix that divide by 0 issue.
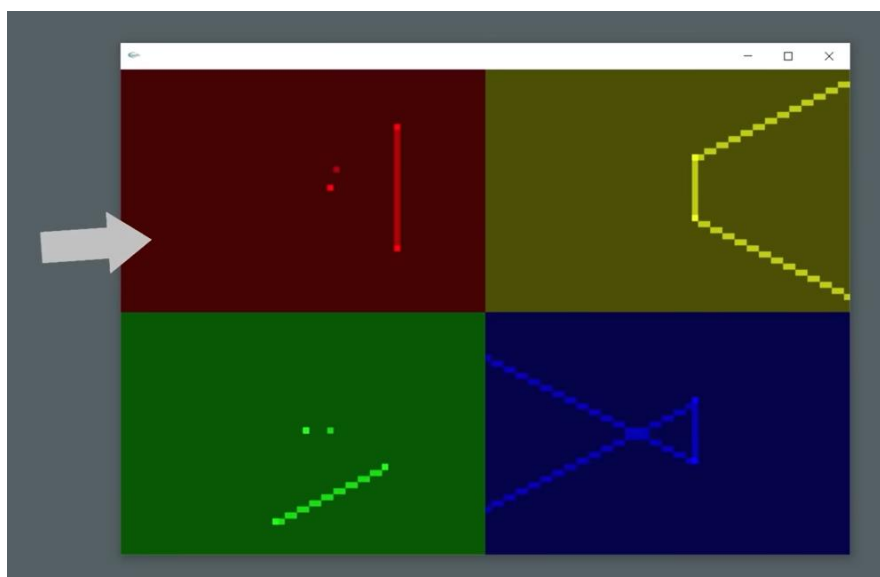
- Upper left square – what you think should happen in a video game, where the player moves around the world (upper right is what we expect to see)
- Lower left square – what is actually happening: the world is rotating around the player (lower right – this can give all kinds of weird errors)[7]

The magic apparently is in the clipBehindPlayer() function. Note that I replaced the pointer to int parameters with reference to int parameters:

```
// clip line
void clipBehindPlayer( int &x1, int &y1, int &z1, int x2, int y2, int z2 ) {

    float da = y1;                              // distance plane -> point a
    float db = y2;                              // distance plane -> point b
    float d = da - db; if (d == 0) { d = 1; }
    float s = da / (da - db);                   // intersection factor (between 0 and 1)
    x1 = x1 + s * (x2 - x1);
    y1 = y1 + s * (y2 - y1); if (y1 == 0) { y1 = 1; }   // prevent divide by zero
    z1 = z1 + s * (z2 - z1);
}
```

"So the line is partially in front of us and partially behind us. We need to convert that player's position in the middle to a value from 0.0f to 1.0f, and then times that value by the difference in the x values, y values and z values. That will give us the exact cut point.

References are used to return the values that are calculated by the function. And a divide by zero is prevented. "[8]

Important note - The position of the player is assumed to be at the origin ("in the middle").

I altered the code later on, to have the check on the d value have effect, and renamed s into t (since t is the conventional parameter name when lerping in [0, 1]:

```
// clip line
void clipBehindPlayer( int &x1, int &y1, int &z1, int x2, int y2, int z2 ) {
    float da = y1;            // distance plane -> point a
    float db = y2;            // distance plane -> point b
    float d = da - db; if (d == 0) { d = 1; }            // prevent division by zero
    float t = da / d;         // intersection factor (between 0 and 1)
    x1 = x1 + t * (x2 - x1);
    y1 = y1 + t * (y2 - y1); if (y1 == 0) { y1 = 1; }   // prevent division by zero
    z1 = z1 + t * (z2 - z1);
}
```

Fase 2a – basic sectors

Struct walls[9] is defined, and an array of these structs is created. It doesn't contain z coordinates since walls are grouped and associated with sectors[10], and the sectors contain the z coordinates.

A sector is a clearly defined object, built up of certain walls, has a bottom height (i.e. floor height) and a top height (i.e. ceiling height). It could be a box, but also have multiple walls like pentagon or octagon

---

[7] Note how this resembles the Bisqwit prototype code.

[8] What I don't understand here is that the d value is checked on zero, and set to 1 if zero, but the d value is never applied anywhere after that. Shouldn't it be applied as the denominator for calculating s? Like float s = da / d; // instead of da / (da – db);

[9] This is called a LINEDEF (icw SIDEDEF) in the DOOM code.

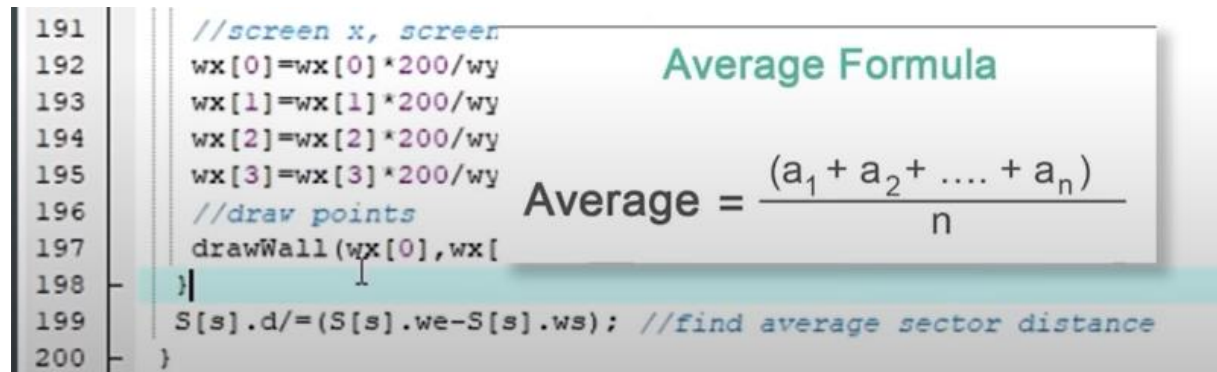[10] In the DOOM code these are called SECTORs as well.

Note: the sectors share their bottom and top height (z-coordinates). That's why it isn't stored in the walls array. The variable d will hold the distance, that's needed to draw the sectors in order (in the implementation of painters algorithm)

Think of z1 as the *floor* value of the sector and z2 as the *roof* (or ceiling) of it

In draw3D() we're adding two more (nested) for loops to cycle through each of the sectors, and within each sector cycle through each of the walls within that sector.

For calculating the distance, we use the average of the two x positions and the average of the two y positions. At the end/ outside the loop the average over all the walls is taken:

```
191        //screen x, screen
192        wx[0]=wx[0]*200/wy
193        wx[1]=wx[1]*200/wy
194        wx[2]=wx[2]*200/wy
195        wx[3]=wx[3]*200/wy
196        //draw points
197        drawWall(wx[0],wx[
198      }
199      S[s].d/=(S[s].we-S[s].ws);  //find average sector distance
200    }
```

Average Formula

$$\text{Average} = \frac{(a_1 + a_2 + \dots + a_n)}{n}$$

"Now I want to create an array that will store all the values for this sector":

```
std::vector<int> loadSectors = {
    // wall start, wall end, z1 height, z2 height
     0,   4,   0,  40,   // sector 1
     4,   8,   0,  40,   // sector 2
     8,  12,   0,  40,   // sector 3
    12,  16,   0,  40,   // sector 4
};
```

The first sector is gonna load walls 0 up to (but not including) 4 (so its going to load 0, 1, 2, 3), and has floor value 0 and top value 40, etc

I had to put constant values in the array declarations to get it to compile, like this:

```
int loadSectors[numSect * 4] = {

int loadWalls[numWall * 5] = {
```

NOTE: Fixed that on second thought by using std::vector<int> instead of int array.

The wall info is initialized as another std::vector<int>:

```
std::vector<int> loadWalls = {
    // x1, y1, x2, y2, color index
     0,   0,  32,   0,  0,
    32,   0,  32,  32,  1,
    32,  32,   0,  32,  0,
     0,  32,   0,   0,  1,

    64,   0,  96,   0,  2,
    96,   0,  96,  32,  3,
    96,  32,  64,  32,  2,
    64,  32,  64,   0,  3,
```

Etc. Each group of four walls got a different color so we could tell each sector apart.

18m30: "in the next video we'll be creating our level editing software"

The information contained in loadWalls[] and loadSectors[] is used in init() to populate the global array of sectors (S) and the global array of walls (W)[11]

In OnUserUpdate() I put a little code to check on the player values:

```
305
306            // display test info on the player
307            DrawString( 2, SH - 18, "Pos: (" + std::to_string( P.x ) + ", "        + std::to_string( P.y ) + ", " + std::to_string( P.z ) + ")" );
308            DrawString( 2, SH - 10, "Angle: " + std::to_string( P.a ) + ", lookup: " + std::to_string( P.l ) );
309
```

Fase 2b – painters algo

In Draw3D() the sectors are sorted (bubble sort) in decreasing distance to implement painters algorithm.

| Result of part 2a	Result of part 2b
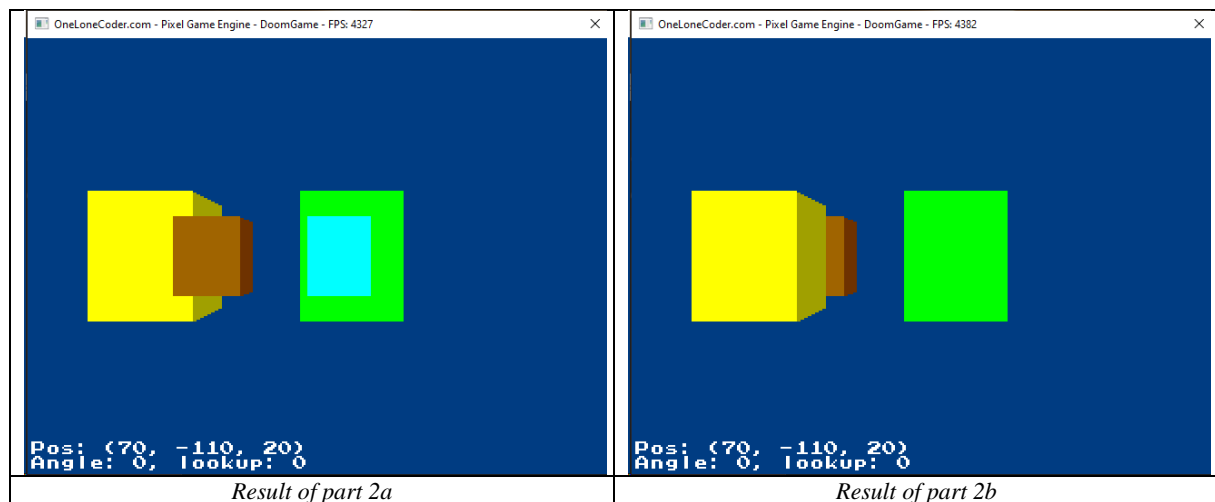
Fase 2c – multiple surfaces

Back-face culling is a method in computer graphics programming which **determines whether a polygon of a graphical object is visible**. If not visible, the polygon is "culled" from rendering process, which increases efficiency by reducing the number of polygons that the hardware has to draw.

All the walls in each sector are drawn twice, first the back walls (the flipped faces), and then the correct front facing walls
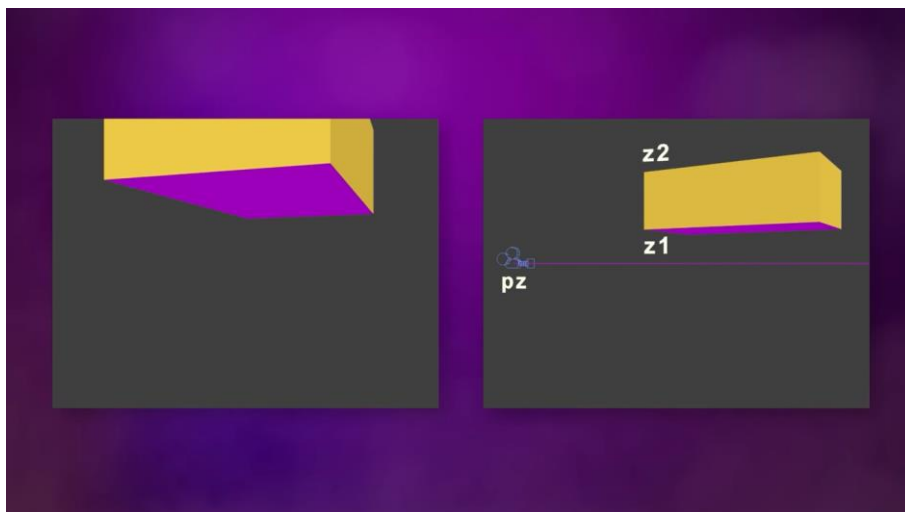
Fase 2d – top and bottom surface

If the player is in between the top and bottom of the wall, then we don't need to draw the surfaces at all
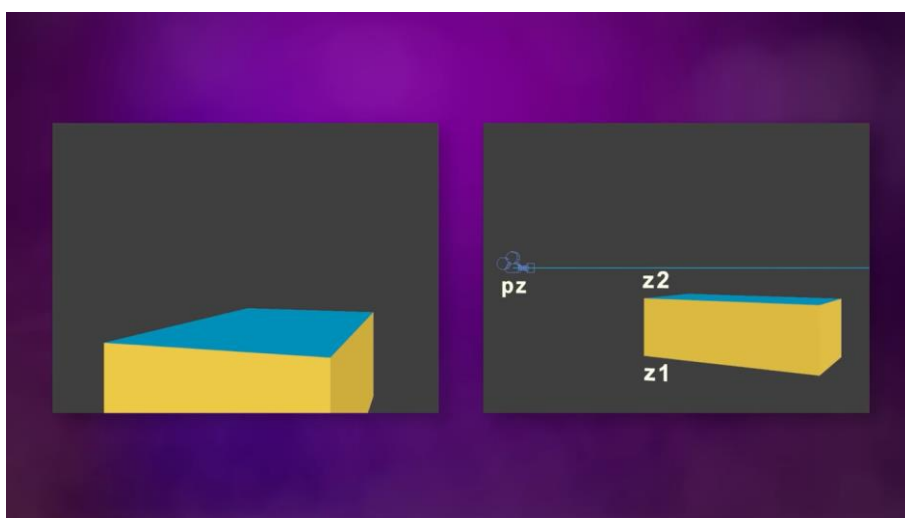
---

[11] 3DSage has a tendency to use variable names consisting of 1 character. I personally think that's not very descriptive and could be done much better.
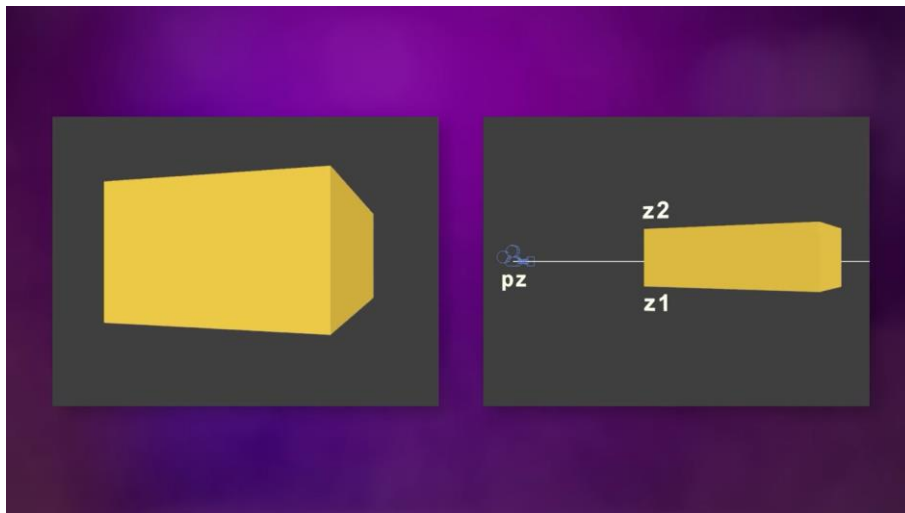
It's only if the player is beneath the bottom that we need to draw the floor:

Or if the player's height is above the top of the wall, then we can draw the top surface:

If we draw the top surface, let's first save the surrounding perimeter of the back walls. And then we draw the front walls, and with a for loop we can draw the surface as a vertical line up to that point [and inverted if it's the bottom]
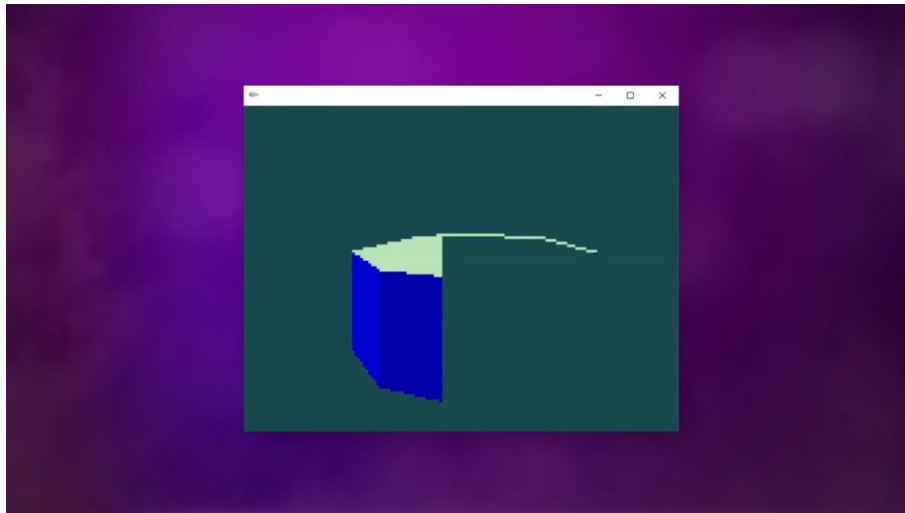
Finished product:

24m38s: "In the next video I'm gonna teach how to make software editing software. […] And off course we'll add textures and collision detection […] We're gonna add so much more in the next few video's […]"