SOFTWARE RENDERER

Recently, I made something like a video game. The video game isn't made and is basic. But from this project I understood that how video games could be made and how they work at the lower level. I started making the renderer without much research in the already existing ones and hence there could be better versions and ideas about the algorithm and the maths. I call it software renderer because, it doesn't use GPU. I wanted to keep it simple, straight and less messy and as a result it only uses the CPU for renderering.

What exactly is this software renderer actually?

It a program which can run on CPU producing images (or frames) of a 3D scence. It is like keeping a camera in a 3D world and showing the picture on the screen.

What format should the 3D scenes be?

The 3D scences can be made up of triangles. We might not use triangles, but I used it for the sake of trying nothing new and most possibly keeping it simple. The triangles are in a single plane making rendering easy to some extent.

How do image forms?

The 3D triangles which make the game, can be projected on the screen and arranged upon one on each other according to which is behind or front and what part is visible. I used rays from the camera which could fall on the triangles and take the color of it (or texture) and put that color on the frame. If another triangle comes which is in front of it, we can remove the older pixel and replace it. The image formations is done according to a camera which is placed in the origin of the scene looking at the positive z axis.

What library can we use to display the images?

I used SDL for displaying the images because it is simple and the most important thing it is working. SDL is no much messy and is straight forward. I only tried other APIs like windows 32 api's set pixel, but it was not enough fast for a game. I even tried the DOS BOX turbo c++ but again it was slow and hardly working.

How do we start?

We can start by keeping a camera towards positive y axis towards a 2D scene to generate a 1D image. Later we can render 3D scences also. We will start with projecting a point on the screen.

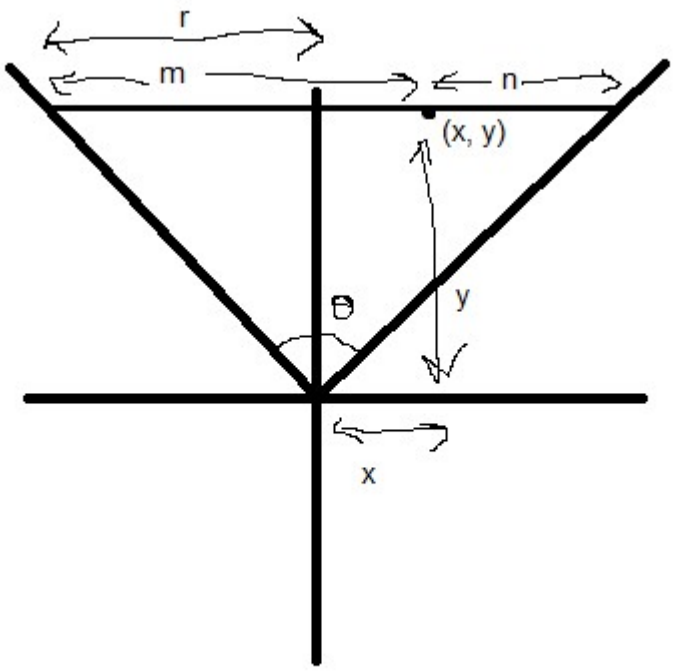

Fig 1

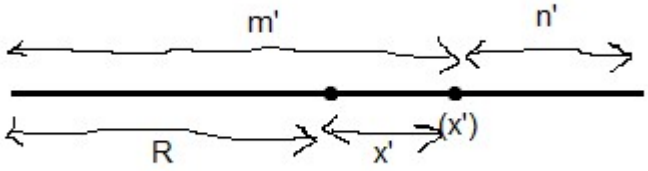

Fig 2

What is field of view?

The triangular structure which is made up of two lines is the field of view. This enables us to see the objects in sight. In case of 2D it is an area in where things are visible on the screen. And the fact we need to use that is, field of view and screen both keep the point in same ratio. For example if the point is just in a straight of the camera (the point is the center) then the point will always be in the center of the screen and have a ratio 1:1, no matter how much far it is. By looking at this observation we can write the equation about the figure 1 (scence) and figure 2 (screen) which is given.

$$\frac{m}{n} = \frac{m'}{n'}$$

$$\frac{r+x}{r-x} = \frac{R+x'}{R-x'}$$

$$tan(\theta/2) = \frac{r}{y}$$

$$ytan(\theta/2) = r$$

$$\frac{ytan(\theta/2)-x}{ytan(\theta/2)+x} = \frac{R-x'}{R+x'}$$

$$x' = \frac{x}{y}Rcot(\theta/2)$$

(Hence the relation between the point on the scene and the screen)

How is the field of view in case of 3D?

It is a cone. The extra information which I would keep is the angle of the point on scence present in the plane of the cone is same as the angle of the point on the screen (with horizontal or vertical). Suppose if the object is in the horizontal line of the plane of the cone, then the point will in the horizontal of the circular screen too. And yes the screen is circular with some radius.

In 3D the cone is made up of triangular field of views with different angle together forming the volume of the cone and also it also says that we can use the older equation just by adding an information the angle and other things.

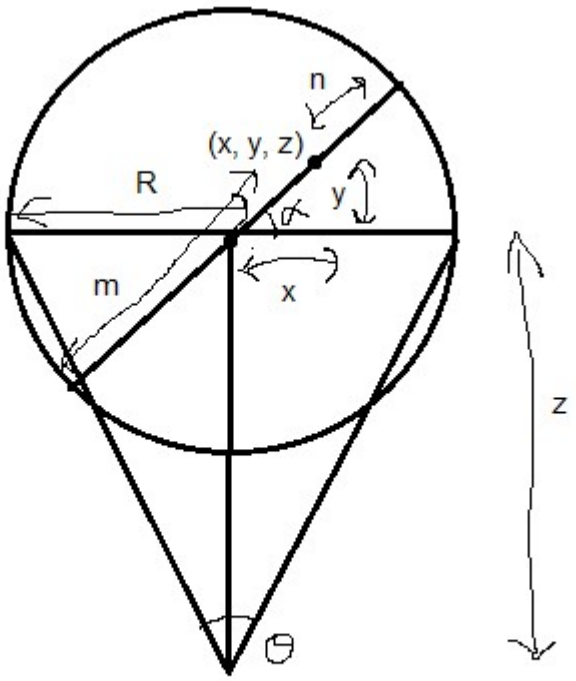

Fig 3 (3D scene)

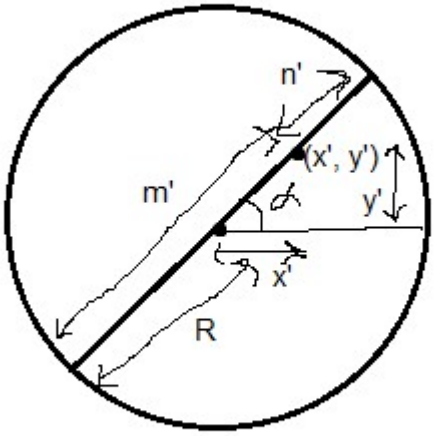

Figure 4 (Screen)

$$\sqrt{x'^2 + y'^2} = \frac{\sqrt{x^2 + y^2}}{z} Rcot(\theta/2)$$

Be careful about the axis considerations, the camera is facing towards the positive z axis.

$$tan(\alpha) = \frac{y}{x} = \frac{y'}{x'}$$

$$\frac{y}{x} x' = y'$$

$$\sqrt{\frac{x'^2 y^2}{x^2} + x'^2} = \frac{x' \sqrt{x^2 + y^2}}{x} = \frac{\sqrt{x^2 + y^2}}{z} Rcot(\theta/2)$$

$$x' = \frac{x}{z} Rcot(\theta/2)$$

$$y' = \frac{y}{z} Rcot(\theta/2)$$

Now we would be able to project a point and even draw a ray from the camera (all points in the ray would have the same projection).

The next plan is take all possible rays and reflect from any surface (surface of the 3D triangle) back by taking its color (color of triangle or color of the texture pixel).

How do I project a 3D triangle on the screen?

```
For all pixels (we can exclude unwanted pixels if we want)

      Find the plane on which the triangle lies

      Draw a ray to the plane

      Find the intersection point between the plane and the ray
      in the 3D space

      If point is inside the triangle

            Draw the pixel by analyzing what should be the color
            of it according to that triangle's color or texture
```

Now in our renderer the triangle's coordinates would be given and we have to somehow draw it on the screen. There may be other techniques to do it, but this trick which I invented should work.

$\vec{p}, \vec{q}, \vec{r}$ are the position vectors of the corners of the 3D triangle, then we can find the equation of the plane where the 3D triangle lies.

$(\vec{q} - \vec{p}) \times (\vec{r} - \vec{p}) = \vec{n} = a\hat{i} + b\hat{j} + c\hat{k}$ is the equation for the vector normal from the plane.

$a(x - p_x) + b(y - p_y) + c(z - p_z) = 0$ This would be equation of the plane if one

point in the plane and the normal vector is given.

$$ax + by + cz = ap_x + bp_y + cp_z$$

$$ax + by + cz = \overrightarrow{p} \cdot \overrightarrow{n}$$

We found the equation of the plane, now we have find its intersection point with the ray of light which would see the color.

$$a\frac{zx'}{Rcot(\theta/2)} + b\frac{zy'}{Rcot(\theta/2)} + cz = \overrightarrow{p} \cdot \overrightarrow{n}$$

$$z = \frac{\overrightarrow{p} \cdot \overrightarrow{n}}{\frac{ax'+by'}{Rcot(\theta/2)} + c}$$

$\frac{x'}{y'} = \frac{x}{y}$ (From the ray equations)

$$\frac{ayx'}{y'} + by + \frac{cyRcot(\theta/2)}{y'} = \overrightarrow{p} \cdot \overrightarrow{n}$$

$$y = \frac{\overrightarrow{p} \cdot \overrightarrow{n}}{\frac{ax'+cRcot(\theta/2)}{y'} + b}$$

$$ax + \frac{bxy'}{x'} + \frac{cxRcot(\theta/2)}{x'} = 0$$

$$x = \frac{\overrightarrow{p} \cdot \overrightarrow{n}}{\frac{by'+cRcot(\theta/2)}{x'} + a}$$

We finally found the intersection point, next step would be checking if the point is inside the triangle or not. This can be done by measuring areas, but at first we need to find how can we find the area of a triangle in 3D.

If $\overrightarrow{p}, \overrightarrow{q}, \overrightarrow{r}$ are the position vector of the corners of the triangle and if

$(\overrightarrow{q} - \overrightarrow{p}) \times (\overrightarrow{r} - \overrightarrow{p}) = a\hat{i} + b\hat{j} + c\hat{k}$ then

$\frac{\sqrt{a^2+b^2+c^2}}{2}$ is the area of the triangle

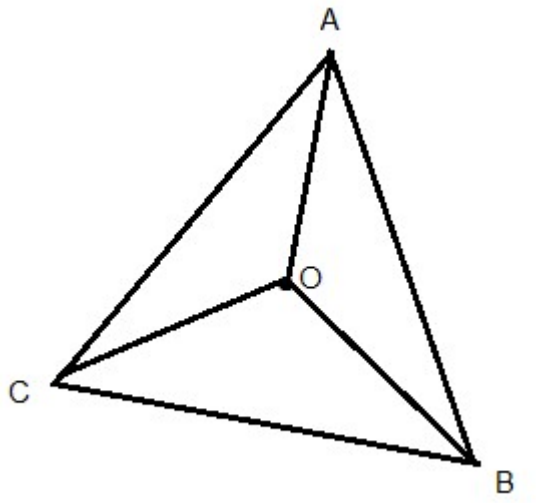

Fig 5 (3D triangle)

If

$ar(ABC) = ar(AOC) + ar(AOB) + ar(BOC)$ then the point O would lie inside the triangle. We can find if the point is inside or not by calculating different four areas.

We can remove also some pixels into consideration by doing this. This would increase the speed of the program.

```
x_max = (RADIUS * 2) - 1
x_min = 0
y_max = (RADIUS * 2) - 1
y_min = 0
if all the three points of the triangle doesnot have z as zero
(to prevent division by zero)
      x_0 = RADIUS + (RADIUS * FOV * t_pt[0].X)/t_pt[0].Z;
      x_1 = RADIUS + (RADIUS * FOV * t_pt[1].X)/t_pt[1].Z;
      x_2 = RADIUS + (RADIUS * FOV * t_pt[2].X)/t_pt[2].Z;
      y_0 = RADIUS - (RADIUS * FOV * t_pt[0].Y)/t_pt[0].Z;
      y_1 = RADIUS - (RADIUS * FOV * t_pt[1].Y)/t_pt[1].Z;
      y_2 = RADIUS - (RADIUS * FOV * t_pt[2].Y)/t_pt[2].Z;
      x_max = (int) round(MAXIMUM_VALUE(x_0, x_1, x_2))
      x_min = (int) round(MINIMUM_VALUE(x_0, x_1, x_2))
      y_max = (int) round(MAXIMUM_VALUE(y_0, y_1, y_2))
      y_min = (int) round(MINIMUM_VALUE(y_0, y_1, y_2))
      if x_min < 0: x_min = 0
      if x_max < 0: x_max = 0
      if x_min >= (RADIUS * 2): x_min = (RADIUS * 2) - 1
      if x_max >= (RADIUS*2): x_max = (RADIUS * 2) - 1
      if y_min < 0: y_min = 0
      if y_max < 0: y_max = 0
      if y_min >= (RADIUS * 2): (RADIUS * 2) - 1
      if y_max >= (RADIUS*2): y_max = (RADIUS * 2) - 1
      For i x_min to x_max
            For j y_min to y_max
                  Make the ray from the pixel and so on....
```

For understand how I removed unwanted pixels from the code, we need at first understand the difference between the coordinates of displaying screen and the calculated screen.

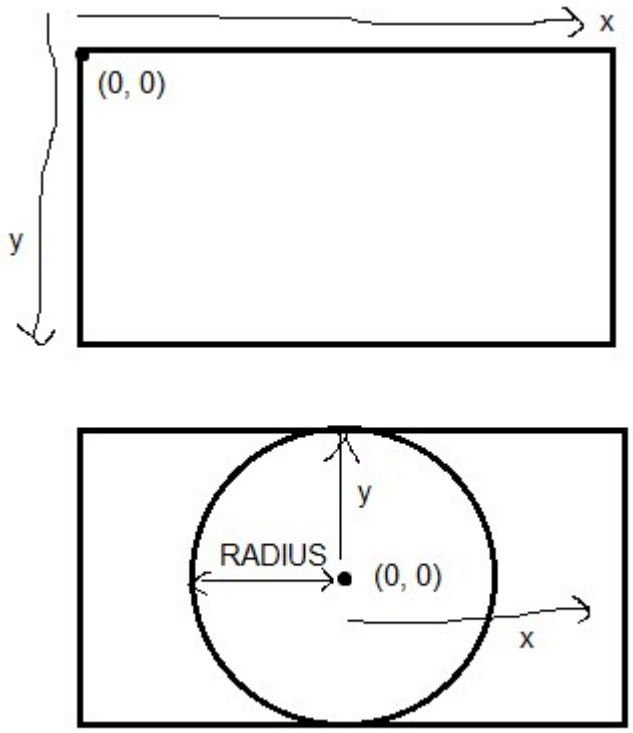

Fig 6 (First one is displaying screen and second one is the calculated screen)

From calculated (x, y) to displaying screen (x', y')

```
x' = RADIUS + x
```

```
y' = RADIUS - y
```

From displaying screen (x', y') to calculated (x, y)

```
x = x' - RADIUS
```

```
y = RADIUS - y'
```

We are doing the things on a square screen of dimensions

```
RADIUS * 2 X RADIUS * 2
```

If z is safe (any part of the triangle is not behind the camera because projection formula doesn't work when this case is there) then we project all the three points and the screen and iterate from the smallest to the largest pixel in both x and y axis. Actually by doing this we only iterating over a small part of pixels and rasterizing (drawing 2D shapes on screen) the triangle within a squaure.

If due to some case the points are outside the screen we will only check pixels inside the

screen and for this I used some inequalities.

How do draw multiple triangles and really form the rendered image?

So now we have enough tools project a triangle and also for rendering. We will talk about colors and texture little after, we will talk about the z buffer algorithm which we can use for rendering.

The z buffer algorithm

```
Fill the depth buffer with positive infinite values

Fill the frame buffer with background color

For all triangles

      For all pixels

            If the pixel can have the projection of the triangle

                  If the new z coordinate of the intersection is
                  lesser than depth buffer's stored z coordinate
                  of the previous intersection then

                        Update the new z coordinate in the z
                        buffer and fill the color in the frame
                        buffer
```

We already talked something about this algorithm in the introduction. It is just drawing pixels in the frame buffer if the intersection point's z coordinate is smaller than the initital when drawing the projected pixel for every triangle. So we keep a depth buffer for storing the z coordinate and a frame buffer for the generated output.

What color to fill the frame buffer?

We can fill plain color of the triangle which can be the attribute on it. This idea is simple. But what if we want to put textures on the models? We can paste the texture on the surface and we will be doing it by considering the fact that even if we paste the texture on somewhere, the distance between the 2D pixels of the texture would remain same. We can try doing with this idea.

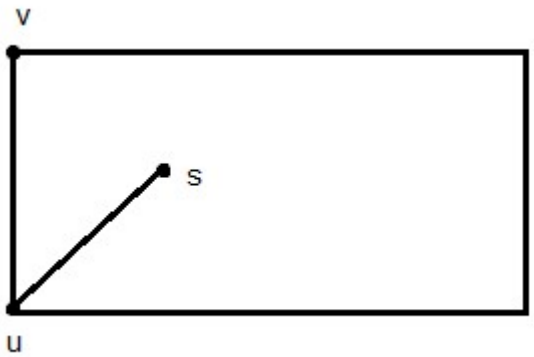

Fig 7 (A texture)

We can take the two position vectors of corners of the triangle on which we are drawing the texture. We can put textures in two triangles and draw rectangular textures which is sometimes more desirable.

The two position vector of corners $\overrightarrow{u}, \overrightarrow{v}$

The vector $\overrightarrow{s}$ represent the position vector of the intersecting point

$$\frac{\overrightarrow{UV} \cdot \overrightarrow{US}}{|\overrightarrow{UV}||\overrightarrow{US}|} = \cos\theta$$

$$x = |\overrightarrow{US}| \sin\theta$$

$$y = |\overrightarrow{US}| \cos\theta$$

Here x and y would the coordinates of the 2D texture image. We can take the color and put it in the frame buffer. If the value of $|\overrightarrow{US}|$ is larger then we can take the remainder of the of the found x and y coordinates with the dimensions of the texture.

The final algorithm in pseudo code

```
Fill the depth buffer with infinity and frame buffer by
background color

For all triangles

      If all the points of the triangle has negetive z value
      then go to the next triangle because the camera can't see
      whats behind it

      if all the values of the z is positive

            Change the range of the pixels to be interated by
            using the projection formula and also never consider
            pixels outside the screen

            For all displaying screen x axis values (in the
            given range)

                  For all displaying screen y axis values (in
                  the given range)

                        Take the triangle and find the plane in
                        which it is in

                        Draw a ray from the pixel and intersect
                        it with the plane

                        If the intersection pixel is inside the
                        triangle then
```

```
If according to depth buffer the
new z coordinate of the
intersection point is less then

      Fill the color of texture or
      the plain color in the
      triangle on the (x, y) pixel
      of the frame buffer

      Update the depth buffer also
      with the new z value
```

But how can I code the renderer logic?

For that we need a C complier and SDL library. We would also need the files for textures to load.

C complier and SDL library

Setting up SDL with code blocks and MinGW

1) First thing you need to do is download SDL headers, libary and binaries. You will find them on the SDL website. Since Code::Blocks comes with the MinGW compiler by default, odds are you'll want to download the MinGW development libraries.

Open the gzip archive and there should be a tar archive. Open up the tar archive and the should be a folder called SDL2-2.something.something. In side of that folder there should be a bunch of folders and files, most importantly i686-w64-mingw32 which contains the 32bit library and x86_64-w64-mingw32 which contains the 64bit library. This is important: most compilers still compile 32bit binaries by default to maximize compatibility. We will be using the 32bit binaries for this tutorial set. It doesn't matter if you have a 64bit operating system, since we are compiling 32bit binaries we will be using the 32bit library. Inside of i686-w64-mingw32 are the include, lib, and bin folders which contain everything we need compile and run SDL applications. Copy the contents of i686-w64-mingw32 to any directory you want. I recommend putting it in a folder that you dedicate to holding all your development libraries for MinGW. I created C:\mingw_dev_lib

2) Start up Code::Blocks and create a new empty project.

3) Go to project properties.

4) Now we have to tell Code::Blocks to search for header files in the library folder we just extracted. Go to build options.

In the Search Directories, we need to add a new compiler directory. Click add, Select the SDL2 folder inside of the include directory from the folder we extracted. Say no when it asks you whether you want it to be a relative path. Now Code::Blocks knows where to find the SDL 2 header files.

If you get an error where the compiler says it can't find SDL.h, it means you messed up this step.

5) Next we are going to tell Code::Blocks to search for library files in the SDL folder we just

extracted. All you have to is go to the linker tab and add the lib directory from the folder we extacted to the linker search directories.If you get an error where the linker complains it can't find -lSDL2 or -lSDL2main, it means you messed up this step.

6) In order to compile SDL 2 code, we have to tell the compiler to link against the libraries. Go under Linker Settings and paste `-lmingw32 -lSDL2main -lSDL2` into the other linker options field and click OK.

If you get an error where the linker complains about a bunch of undefined references, it means you messed up this step.

7) Go back to the project properites and under Build Targets select the build type.

You can set it to GUI Application if you don't want console output, and Console Application if you do want console output.

8) When our SDL 2 application runs, the operating system needs to be able to find the dll file.

Go find the SDL 2 folder you extracted and from the bin folder inside copy SDL2.dll and put it either where your executable will run, or inside of the system directory. C:\WINDOWS \SYSTEM32 is the 32bit windows system directory and C:\Windows\SysWOW64 is the 64bit system directory of 32bit applications. If you get an error when you run the program where it complains that it can't find SDL2.dll, it means you messed up this step.

The basic codes for the projects to put images, load files and so on

```
//...

int main(int argc, char *argv[]) {
    SDL_Event event;
    SDL_Renderer *renderer;
    SDL_Window *window;

    SDL_Init(SDL_INIT_VIDEO);
    SDL_CreateWindowAndRenderer(RADIUS*2, RADIUS*2, 0,
&window, &renderer);
    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
    SDL_RenderClear(renderer);

    //...

    //Load texture
    char *texture = malloc(SIZE_OF_TEXTURE);
    FILE *fPtr;

    if ((fPtr = fopen("filename.ppm", "rb")) == NULL){
      printf("Error\n");
      return 0;
    }

    fread(texture, SIZE_OF_TEXTURE, 1, fPtr);

    //...
```

```
    while (1) {
        //...
        SDL_Delay(100); //Frame rate
        if (SDL_PollEvent(&event) && event.type == SDL_QUIT)
            break;
    }
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}

//...
//Draw the frame buffer
void drawBuffer(SDL_Renderer *renderer, struct COLOR_MIX
*frame_buffer){

    SDL_Texture *Tile = SDL_CreateTexture(renderer,
SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_STREAMING,
     RADIUS*2, RADIUS*2);

    unsigned char *bytes;
    int pitch;

    SDL_LockTexture(Tile, NULL, (void **)&bytes, &pitch);

    int i, j;
    for (i=0; i<RADIUS*2; ++i){
      for (j=0; j<RADIUS*2; ++j){
           bytes[(i*RADIUS*8+j*4)] = 255;
           bytes[(i*RADIUS*8+j*4) + 1] = frame_buffer[j*RADIUS*
2+i].BLUE;
           bytes[(i*RADIUS*8+j*4) + 2] = frame_buffer[j*RADIUS*
2+i].GREEN;
           bytes[(i*RADIUS*8+j*4) + 3] = frame_buffer[j*RADIUS*
2+i].RED;
      }
    }

    SDL_UnlockTexture(Tile);
    SDL_Rect destination = {0, 0, RADIUS*2, RADIUS*2};
    SDL_RenderCopy(renderer, Tile, NULL, &destination);
    SDL_RenderPresent(renderer);
}
```

We can use ppm files for reading the texture. We can convert normal image files to ppm format enabling our C program to read the texture file. The format of ppm files is like this.

```
P6
3 2
255
# The part above is the header
# "P3" means this is a RGB color image in ASCII
# "3 2" is the width and height of the image in pixels
# "255" is the maximum value for each color
# Below is the binary data of the image
```

The full code

https://github.com/SwastikMajumder/cpurenderer/tree/main/engine_test