1º Take the STL file and load to memory.

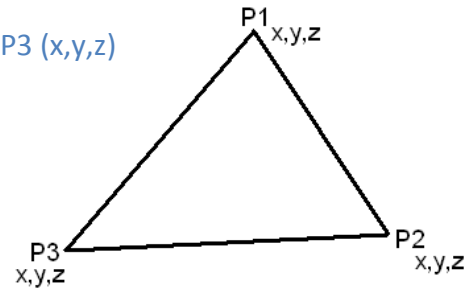
```
#define MAX(a,b) ((a > b) ? a : b)
#define MIN(a,b) ((a < b) ? a : b)
FILE *stl; // pointer to STL file
char *cache_stl; // pointer to STL in memory
// open the stl file
if((stl = fopen("c:\\bin.stl", "rb")) == NULL) { /* error opening the file stl... */ }
// get the size
fseek(stl, 0, SEEK_END);
unsigned long size = ftell(stl);
// allocates space in memory to the file
if((cache_stl =(char *) malloc(size)) == NULL) {  /* error while allocating memory... */ }
// load the file to memory
fseek(stl,0,SEEK_SET);
if(fread(cache_stl, 1, size, stl) != size) {  /* error reading the file... */ }
// close the file
fclose(stl);
```

2º Get the numbers of triangles.

```
unsigned int *ntt = (unsigned int *) &cache_stl[80]; // pointer to the numbers of triangles
```

3º Make a loop in all triangles to calculate the minimum point (X,Y,Z) and the maximum (X,Y,Z).

```
float maximum[3]; // maximum coordinated
float minimum[3]; // minimum coordinated
float *points; // pointer to the coordinates P1 (x,y,z), P2 (x,y,z) e P3 (x,y,z)
unsigned int x; // used in loops
for(x = 0; x < (*ntt-1); x++)
{
    points = (float *) &cache_stl[96+(x*50)];
    if(x == 0)
    {
        minimum[0] = points[0]; maximum[0] = points[0]; // x
        minimum [1] = points[1]; maximum[1] = points[1]; // y
        minimum [2] = points[2]; maximum[2] = points[2]; // z
    }
    else
    {
        if(minimum[0] > points[0]) minimum[0] = points[0]; // x
        if(minimum[1] > points[1]) minimum[1] = points[1]; // y
        if(minimum[2] > points[2]) minimum[2] = points[2]; // z
        if(maximum[0] < points[0]) maximum[0] = points[0]; // x
        if(maximum[1] < points[1]) maximum[1] = points[1]; // y
        if(maximum[2] < points[2]) maximum[2] = points[2]; // z
    }
    if(minimum[0] > points[3]) minimum[0] = points[3]; // x
    if(minimum[0] > points[6]) minimum[0] = points[6]; // x
    if(minimum[1] > points[4]) minimum[1] = points[4]; // y
    if(minimum[1] > points[7]) minimum[1] = points[7]; // y
    if(minimum[2] > points[5]) minimum[2] = points[5]; // z
    if(minimum[2] > points[8]) minimum[2] = points[8]; // z
    if(maximum[0] < points[3]) maximum[0] = points[3]; // x
    if(maximum[0] < points[6]) maximum[0] = points[6]; // x
    if(maximum[1] < points[4]) maximum[1] = points[4]; // y
    if(maximum[1] < points[7]) maximum[1] = points[7]; // y
    if(maximum[2] < points[5]) maximum[2] = points[5]; // z
    if(maximum[2] < points[8]) maximum[2] = points[8]; // z
```

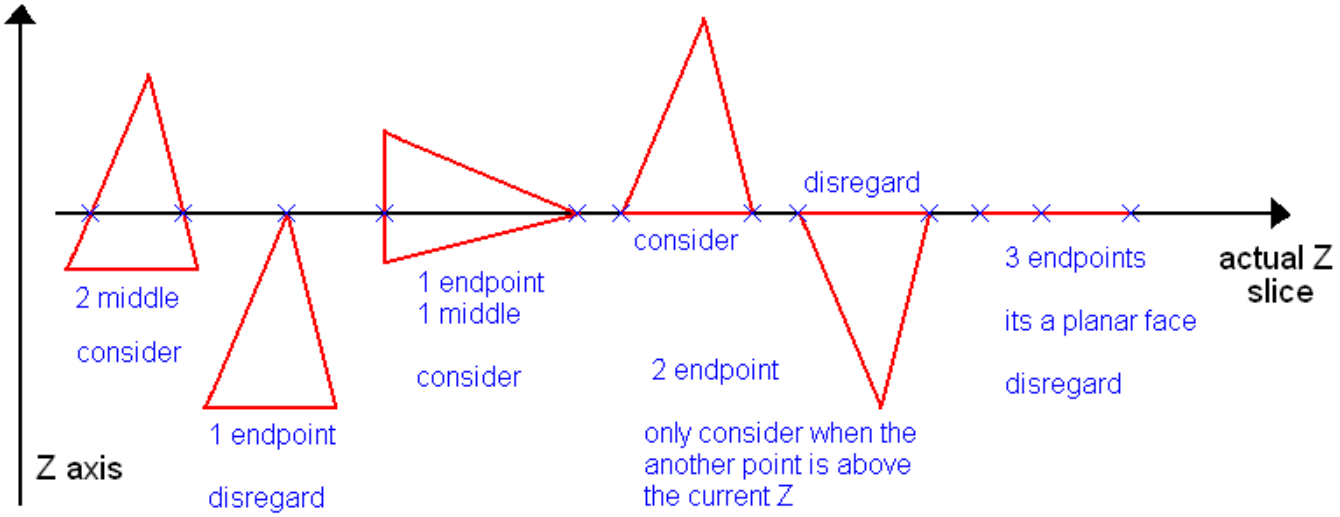


```
}
```

4º Make a loop between the minimum and maximum Z, and increment is the thickness of the slice.

```
double z; // current position in Z of the slice
float increment = 1; // thickness of the slice
for(z = minimum[2]; z < maximum[2] ; z += increment) // each loop is a slice
{
```

5º Make a loop in all triangles and see which intersect the current Z.




```
unsigned int lines = 0; // number of lines not horizontal that was intersected
float *cache_tmp = NULL; // pointer to memory of intersected lines of actual slice
double diff, factor; // used in some calculations

for(x = 0; x < *ntt; x++)
{
    points = (float *) &cache_stl[96+(x*50)];
    float final[3][2]; // store the points where was intersected
    int cnt=0, cntpos=0;
    if(points[2] == z) // endpoint P1
    {
        final[cnt][0] = points[0];
        final[cnt][1] = points[1];
        cnt++; cntpos += 1;
    }
    if(points[5] == z) // endpoint P2
    {
        final[cnt][0] = points[3];
        final[cnt][1] = points[4];
        cnt++; cntpos += 2;
    }
    if(points[8] == z) // endpoint P3
    {
        final[cnt][0] = points[6];
        final[cnt][1] = points[7];
        cnt++; cntpos += 4;
    }
    if((points[2] > z && points[5] < z) || (points[5] > z && points[2] < z)) // middle between P1-P2
    {
        diff = MAX(points[2],points[5]) - MIN(points[2],points[5]);
        factor = (z-MIN(points[2],points[5]))/diff;
        if(points[2] < points[5])
```

```
    {
        final[cnt][0] = (((points[3] - points[0]) * factor) + points[0]);
        final[cnt][1] = (((points[4] - points[1]) * factor) + points[1]);
    }
    else
    {
        final[cnt][0] = (((points[0] - points[3]) * factor) + points[3]);
        final[cnt][1] = (((points[1] - points[4]) * factor) + points[4]);
    }
    cnt++;
}
if((points[2] > z && points[8] < z) || (points[8] > z && points[2] < z)) // middle between P1-P3
{
    diff = MAX(points[2],points[8]) - MIN(points[2],points[8]);
    factor = (z-MIN(points[2],points[8]))/diff;
    if(points[2] < points[8])
    {
        final[cnt][0] = (((points[6] - points[0]) * factor) + points[0]);
        final[cnt][1] = (((points[7] - points[1]) * factor) + points[1]);
    }
    else
    {
        final[cnt][0] = (((points[0] - points[6]) * factor) + points[6]);
        final[cnt][1] = (((points[1] - points[7]) * factor) + points[7]);
    }
    cnt++;
}
if((points[8] > z && points[5] < z) || (points[5] > z && points[8] < z)) // middle between P2-P3
{
    diff = MAX(points[8],points[5]) - MIN(points[8],points[5]);
    factor = (z-MIN(points[8],points[5]))/diff;
    if(points[8] < points[5])
    {
        final[cnt][0] = (((points[3] - points[6]) * factor) + points[6]);
        final[cnt][1] = (((points[4] - points[7]) * factor) + points[7]);
    }
    else
    {
        final[cnt][0] = (((points[6] - points[3]) * factor) + points[3]);
        final[cnt][1] = (((points[7] - points[4]) * factor) + points[4]);
    }
    cnt++;
}
if(cnt == 1 && cntpos > 0) // 1 endpoint
{
    // disregard
}
if(cnt == 2) // line
{
    if(cntpos == 0) // 2 middle
    {
        if(final[0][1] != final[1][1]) // if the line is horizontal, will disregard (rule of ScanLine-Fill)
        {
            float *p; // pointer temporary
            if((p = (float *) realloc (cache_tmp, (lines+1)*16)) == NULL) { /* Error (re)allocating memory */ }
```

```
                cache_tmp = p;
                cache_tmp[(lines*4)+0] = final[0][0];
                cache_tmp[(lines*4)+1] = final[0][1];
                cache_tmp[(lines*4)+2] = final[1][0];
                cache_tmp[(lines*4)+3] = final[1][1];
                lines++;
            }
        }
        else if((cntpos == 1) || (cntpos == 2) || (cntpos == 4)) // 1 endpoint 1 middle
        {
            if(final[0][1] != final[1][1]) // if the line is horizontal, will disregard (rule of ScanLine-Fill)
            {
                float *p; // pointer temporary
                if((p = (float *) realloc (cache_tmp, (lines+1)*16)) == NULL) { /* Error (re)allocating memory */ }
                cache_tmp = p;
                cache_tmp[(lines*4)+0] = final[0][0];
                cache_tmp[(lines*4)+1] = final[0][1];
                cache_tmp[(lines*4)+2] = final[1][0];
                cache_tmp[(lines*4)+3] = final[1][1];
                lines++;
            }
        }
        else // 2 endpoints
        {
            if(final[0][1] != final[1][1]) // if the line is horizontal, will disregard (rule of ScanLine-Fill)
            {
                if((points[2] > z) || (points[5] > z) || (points[8] > z))
                {
                    float *p; // pointer temorary
                    if((p = (float *) realloc (cache_tmp, (lines+1)*16)) == NULL) { /*Error (re)allocating memory */ }
                    cache_tmp = p;
                    cache_tmp[(lines*4)+0] = final[0][0];
                    cache_tmp[(lines*4)+1] = final[0][1];
                    cache_tmp[(lines*4)+2] = final[1][0];
                    cache_tmp[(lines*4)+3] = final[1][1];
                    lines++;
                }
            }
        }
    }
    if(cnt == 3 && cntpos == 7) // face plane in Z
    {
        // disregard
    }
} // end of loop of triangles
if(lines == 0) // if no lines intersected, go to the next slice,and print a blank
{
    // PRINT A BLANK SLICE
    continue;  // go to the next slice
}
if(lines == 1) // if found only one line, something is wrong
{
    // PRINT A BLANK SLICE
    free(cache_tmp); // deallocate from memory the lines generated when sliced
    continue; // go to the next slice
```

```
}
```

6º Calculate the minimum point (X,Y) and the maximum (X,Y) of the slice.

```
float _minimum[2]; // coordinated minimum of the generated slice
float _maximum[2]; // coordinated maximim of the generated slice
for(x = 0; x < (lines*2); x++)
{
    points = &cache_tmp[x*2];
    if(x == 0)
    {
        _minimum[0] = points[0]; _maximum[0] = points[0];
        _minimum[1] = points[1]; _maximum[1] = points[1];
    }
    else
    {
        if(_minimum[0] > points[0]) _minimum[0] = points[0];
        if(_minimum[1] > points[1]) _minimum[1] = points[1];
        if(_maximum[0] < points[0]) _maximum[0] = points[0];
        if(_maximum[1] < points[1]) _maximum[1] = points[1];
    }
}
```

7º With the lines intersecteds, make a algorithm called ScanLine-Fill (SCANLINE-FILL, that part that i did not know, after some reaserch in pdfs about image raster, that helped to finish the project).

Make a loop between the minimum Y and maximum Y, and increment is the resolution output of the image in Y.

```
double y; // current position of scanline-fill
float dpiy = 300; // resolution output in Y of the image
for(y = _minimum[1]; y < _maximum[1]; y += (25.4/dpiy)) // start the scanline-fill
{
    unsigned int lines_slf = 0; // numbers of points in X intersected
    float *cache_tmpx = NULL; // pointer to values of X

    for(x = 0; x < lines; x++)
    {
        points = &cache_tmp[x*4];
        if(MIN(points[1],points[3]) <= y && MAX(points[1],points[3]) > y)
        {
            diff = MAX(points[1],points[3]) - MIN(points[1],points[3]);
            factor = (y-MIN(points[1],points[3]))/diff;
            if(points[1] < points[3])
                diff = (((points[2] - points[0]) * factor) + points[0]);
            else
                diff = (((points[0] - points[2]) * factor) + points[2]);
            float *p; // ponteiro temporário
            if((p = (float *) realloc (cache_tmpx, ((lines_slf+1)*4))) == NULL) { /* Error (re)allocating memory */ }
            cache_tmpx = p;
            cache_tmpx[lines_slf] = diff;
            lines_slf++;
        }
    }
    // in the scanline-fill the numbers of lines must be even, if is odd you never know where is inside and
    outside
    if(lines_slf % 2)
    {
```

```
            // error the number of lines is odd...
            free(cache_tmpx);
            continue;
        }
        8º With the lines intersecteds ordering then in increasing order of X
        for(x = 0; x < lines_slf; x++)
        {
            if(x == (lines_slf-1)) break; // to fix a strange bug in my compiler
            float origin = cache_tmpx[x];
            unsigned long pmin = x;
            for(unsigned int i = (x+1); i < lines_slf; i++)
            {
                if(cache_tmpx[x] > cache_tmpx[i])
                {
                    cache_tmpx[x] = cache_tmpx[i];
                    pmin = i;
                }
            }
            if(pmin != x)
            {
                cache_tmpx[pmin] = origin;
            }
        }
        // write the horizontal lines in the image, separating  all X intersected in pairs
        for(x = 0; x < (lines_slf/2); x++)
        {
            float xse[2]; // x start <....> end
            xse[0] = cache_tmpx[x*2]; // start
            xse[1] = cache_tmpx[1+(x*2)]; // end
            Trace a line  between the X start to the X end on the image using the Y actual.
            // BMP_Line(((xse[0]-minimum[0])/(25.4/dpix)),((xse[1]-minimum[0])/(25.4/dpix)));
        }
        free(cache_tmpx); // deallocate from memory the X values generated on scanline-fill
    } // end of scanline-fill
    The image was generated, PRINT.
    free(cache_tmp); // deallocate from memory the lines generated when sliced
  } // end of slice
  free(cache_stl); // deallocate from memory the stl file
} // end of program
```