Majd Hamdan Nov 24, 2022

3D Graphics Display Model

1. Preface

This document illustrate the mathematical model used for the 3D graphics display model. In this document, I provide instructions to run the code, explain the assumptions and derive the equations used in the code.

To execute the code, please open "3D_Graphiscs_Wireframe.py" or "3D_Graphics_Shader.py" and specify the name of the input file (if it is in the same path as the code) in line 27 under the "file_name" variable. Please specify the path to the input file and the name of the file otherwise. To run the program, you need the **math**, **pygame**, **time**, and **copy** libraries installed. If the libraries are installed, you may run the programs from any IDE or execute it from the command line. To run it from the command line, please navigate to the folder where the programs are saved then type:

```
python .\3D_Graphiscs_Wireframe.py"
```

Please follow the same instructions to run both python files. As a default, the input file is specified as "cube.txt" and is expected to be in the same folder as the programs.

You can run "D_Graphics_Colored_faces.py" without filling in the faces of the triangles by commenting line 325 and uncommenting line 330. This will function like "3D_Graphiscs_Wireframe.py" but with only showing visible triangles and following the correct order of drawing.

2. Assumptions and Definitions

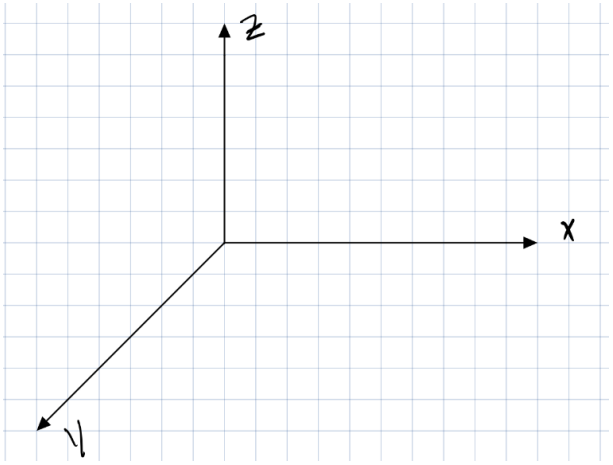
The assumption made in this submission are:

- The observer is an infinite distance from the canvas
- For shading, we assume light is coming from the direction of the viewer. We assume the light source is large compared to the display (light is coming from every direction-not a single point)

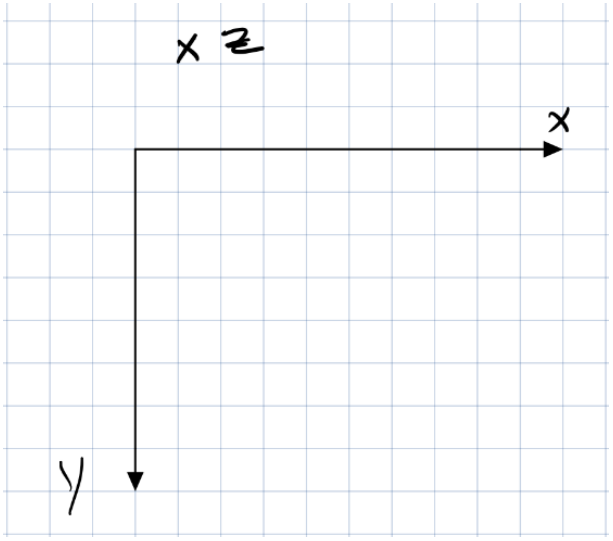
We define a vector by specifying three coordinates. We define a triangle by specifying three vertices. The order at which the vertices are specifies is **clockwise**. The order is important to define the forward face of the triangle (unify the directions of the triangle norms). We discuss the reason for computing the triangles norm in further sections. Clockwise order results in norms pointing away from the z-axis.

We define the world 3D space as shown in the following figure. The positive X-axis is pointing horizontally to the right. The positive Y-axis is pointing towards us. The positive Z-axis is pointing vertically upwards. The center is at (0,0,0).

3D Graphics Display Model



We define the display 2D plane as shown in the following figure. The positive X-axis is pointing horizontally to the right. The positive Y-axis is pointing vertically upward. The positive Z-axis is pointing out of the plane of the window toward the observer. • The origin is at the center of the display.

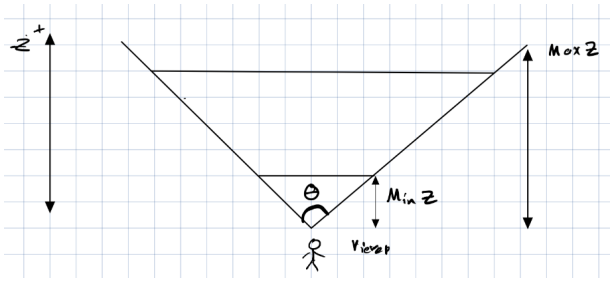


3D Graphics Display Model

3. 3D_Graphiscs_Wireframe.py Model

3.1 Perspective Projection

We take the display as a rectangle object with width w and height h. We define the aspect ration, $a = \frac{h}{w}$. The 2D screen is depicting the field of view of the viewer. The field of view is shown in the following figure. The field of view scope is defined with an angle θ. The furthest distance the viewer can see is defined as Z_{max}. Z_{min} is defined as the distance between the viewer and the screen.



When projecting any point (x, y, z) in the 3D plane on the screen, we must scale x, y, and z according to the above figure.

Both x and y must be scaled by the aspect ratio a to be displayed on the screen in a way that reflects their position in 3D. Additionally, we must take into account that the closer the object is to the user the bigger it is and vice versa. If we increase the field of view by increasing θ, the viewer can see more objects and vice versa. This effect is reminiscent of zooming. We also observe that the projection must take into account that the further a moving object is a way from the viewer, their movement appear smaller. For the z coordinate, we must offset the point in the 2D plane by the assumed distance from the screen. We also must scale it to fit in the $[Z_{min}, Z_{max}]$ interval. These steps are described by the following equations:

$$x_{2D} = \frac{a}{\tan(\frac{\theta}{2})z} x_{3D}$$
$$x_{2D} = \frac{1}{\tan(\frac{\theta}{2})z} y_{3D}$$
$$z_{2D} = \frac{1}{Z_{max} \cdot Z_{min}} (Z_{max} z_{3D} - (Z_{max} - Z_{min}))$$

Here, we assume for simplicity that $y_{screen} = 1$. Multiplying x_{3D} by a normalize x accordingly. The $\frac{1}{\tan(\frac{\theta}{2})}$ represents the effect of zooming. Dividing both x and y by z takes care of reducing movement as the object is further away from the viewer (another way of saying that the

derivative of x and y are proportional to $1/z$: $\dot{x} \propto \frac{x}{z}$, $\dot{y} \propto \frac{y}{z}$). $\frac{Z_{max}}{Z_{max}-Z_{min}}$ is scaling z_{3D} within the view and $\frac{Z_{max} \cdot Z_{min}}{Z_{max}-Z_{min}}$ is taking care of the offset.
Taking these operation to matrix form, we have:

$$\begin{bmatrix} x_{2D} \\ y_{2D} \\ z_{2D} \end{bmatrix} = \begin{bmatrix} \frac{a}{\tan(\frac{\theta}{2})} & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})z} & 0 \\ 0 & 0 & \frac{Z_{max}}{Z_{max}-Z_{min}} \end{bmatrix} \begin{bmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \end{bmatrix}$$

We apply the offset translation to z by extending the dimensions of the matrix to 4:

$$\begin{bmatrix} x_{2D} \\ y_{2D} \\ z_{2D} \\ w \end{bmatrix} = \begin{bmatrix} \frac{a}{\tan(\frac{\theta}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})z} & 0 & 0 \\ 0 & 0 & \frac{Z_{max}}{Z_{max}-Z_{min}} & 0 \\ 0 & 0 & -\frac{Z_{max} \cdot Z_{min}}{Z_{max}-Z_{min}} & 0 \end{bmatrix} \begin{bmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \\ 1 \end{bmatrix}$$

We still need to divide x and y by z. To still conserve the value of z before the matrix operation, we add 1 to the (4, 3) element of the transition matrix to obtain:

$$\begin{bmatrix} x_{2D} \\ y_{2D} \\ z_{2D} \\ z_{3D} \end{bmatrix} = \begin{bmatrix} \frac{a}{\tan(\frac{\theta}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})z} & 0 & 0 \\ 0 & 0 & \frac{Z_{max}}{Z_{max}-Z_{min}} & 1 \\ 0 & 0 & -\frac{Z_{max} \cdot Z_{min}}{Z_{max}-Z_{min}} & 0 \end{bmatrix} \begin{bmatrix} x_{3D} \\ y_{3D} \\ z_{3D} \\ 1 \end{bmatrix}$$

In code, we simply extract z_{3d} from the result and divide x_{2D} and y_{2D} by it to obtain the correct result. **Both "3_Graphiscs_Vertices.py" and "3D_Graphics_Shader.py" are using this matrix to obtain the projection coordinates**. This is done in the **ProjectTriangle()** function in both files.

Since we are assuming the height of the screen is 1, it is important to note that the result of this matrix multiplication will be normalized to 1. We must scale the x and y results so the figure is large enough to be seen in the display. This is done within the **DrawTriangle()** function in the code. The scaling is such that the max coordinate occupies 1/4 of the screen.

3.2 Rotation

We mapped every 1 unit of mouse movement to a corresponding rotation angle as described in the following equations:

$$\theta_x = 2\pi \frac{y_{units}}{w}$$
$$\theta_y = 2\pi \frac{x_{units}}{h}$$

where θ_x is the rotation angle around the x-axis, y_{units} are the mouse movements units in the y direction, θ_y is the rotation angle around the y-axis, and x_{units} are the mouse movements units in the x direction. This is done in the **driver code section of the code**.

To apply these rotations, we simply use the rotation transition matrices defined as:

$$\text{x rotation matrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{y rotation matrix} = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & \cos\theta_x & 0 & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

These rotations are accomplished using **YRotation()** and **XRotation()** functions in my code.

This concludes the modeling used in n "3D_Graphiscs_Wireframe.py".

4. 3D_Graphics_Shader.py Model

The model in "3D_Graphiscs_Wireframe.py is displaying all triangles of the models. However, in real life, shapes closer to the viewer block shapes that are positioned behind them. The following subsection illustrate the method I used to find triangles that shouldn't be displayed.

4.1 Triangle norms

Given that we defined the order at which we specify the vertices of a triangle as clockwise, if we were to find the norm of a triangle, the norm will be pointing away from the z-axis (except when the triangle is perpendicular to the z-axis). Therefore, triangles with a face towards the viewer will have a negative norm (since the positive z-axis direction is into the page). Otherwise, the norm will be positive. That being said, even if the normal of a shape is negative, we are unable to see shapes with normal perpendicular to our line of sight (when the shape is viewed on edge). We can avoid such surfaces by taking the dot product of the normal with a line from the viewer eye to any point on the triangle. In our code, we assumed the user is at the origin for simplicity. But we defined the user's location using a variable called **vcamera** that can be adjusted to place the user at a desired location. If the dot product is non-zero and negative, it means that the shape is viewable by the viewer. We only project triangles that satisfy the previous condition. This is achieved between lines 180 and 212 of the **ProjectTriangle()** function.

4.2 Triangle Coloring

To achieve the shading effect, we incorporate a light source in our model. As indicated previously, we assume light is coming from the direction of the viewer and that the light source is large compared to the display so that light rays appear to be coming from every direction. A normalized light ray vector may then be defined as:

$$v_{\text{ray}} = 0x + 0y + 1z$$

To compute how much light reach a certain shape, we make use of the triangles norms. First, we don't have to consider the effect of light on triangles with positive norms since they won't be painted. For painted triangles, the more the norm of the triangle align with the direction of the light, the more light will reach the triangle. Another way of saying this is the larger the dot product between the light ray and the norm of a triangle is, the bluer the triangle will be. The dot product will be negative since the two vectors are pointing opposite to each other. To avoid having a negative result for the dot product, we define the ray vector as:

$$v_{\text{ray}} = 0x + 0y + -1z$$

to simplify the code. This is accomplished within **ProjectTriangle()** function in the code.

The dot product will give a value between 0 and 1 for all visible triangles. We map this value to a color between #00005F and #0000FF where #0000FF represent a value of 1 for the dot product. here are 160 values between #00005F and #0000FF. For simplification, we will only use 100 of them by taking the dot product to two significant figure. This is done using the **ConvertToColor()** function in the code.

4.3 Order of Painting

Lastly, for the case of two visible triangles with one of the triangle closer to the user but not completely covering the other triangle, we may arrive at an issue with the current code. If the further triangle was to be drawn first, then the second triangle was drawn. The second triangle will simply cover the first triangle and we will have no issues. But if the further triangle was drawn last, it will cover the closer triangle and show parts of the shape that shouldn't otherwise be viewed. To overcome this challenge, we rank the triangles by how far they are from the viewers and draw the triangles further away first. We use the average of the z values of all of a triangle vertices to describe how far a triangle is from the user. This done within **DrawTriangles()** function in the code.

5. Final Considerations

One important assumption we made throughout our code is that a triangle is specified by a clockwise ordering of its vertices. The input file may not follow this assumption and instead

contains triangles that are specified in anticlockwise order. In this case, the triangles that should be displayed will be hidden and vice versa. To solve this, we had to re-arrange the triangles such that they all follow a clockwise order. We do so by finding the angle between the norm of a triangle and the z axis. If the angle is larger than 90, then the norm is pointing towards the z-axis instead of away from it. We re-arrange any triangle with an angle larger than 90 with the z-axis. This is done in line 197 in the code.