3D Graphics Engine

Team: Mykhailo-Taras Sobko, Anna-Alina Bondarets, Oleksandra Stasiuk

The source code of the project can be found on the GitHub:
https://github.com/MykhailoSobko/3d_graphics_rendering

1. Introduction

This project is about rendering 3D objects. Given one object representation in 3D, the engine gives the opportunity to view it projected onto a 2D screen with the transformations given from the user, such as rotations, zooming, translation, and changing light direction. This will be useful not only for the study purposes, but also as a fundamental step to developing the real game engine.

The project is focused around the real-time computer graphics, or real-time rendering, conceptions. It provides real-time object analysis with an interactive interface through keyboard and mouse. Using the different techniques for rendering, such as ray-tracing and rasterization, the user can respond to rendered images in real time, producing an interactive experience.

2. Rendering

Rendering is the final process of creating the actual 2D image or animation from the prepared scene. This can be compared to taking a photo or filming the scene after the setup is finished in real life. Several different, and often specialized, rendering methods have been developed. These range from the distinctly non-realistic wireframe rendering through polygon-based rendering, to more advanced techniques such as: scanline rendering, ray tracing, or radiosity. Rendering may take from fractions of a second to days for a single image/frame. In general, different methods are better suited for either photorealistic rendering, or real-time rendering.

Rendering for interactive media, such as games and simulations, is calculated and displayed in real time, at rates of approximately 20 to 120 frames per second. In real-time rendering, the goal is to show as much information as possible as the eye can process in a fraction of a second (a.k.a. "in one frame": In the case of a 30 frame-per-second animation, a frame encompasses one 30th of a second).

The primary goal is to achieve an as high as possible degree of photorealism at an acceptable minimum rendering speed (usually 24 frames per second, as that is the minimum the human eye needs to see to successfully create the illusion of movement). In fact, exploitations can be applied in the way the eye 'perceives' the world, and as a result, the final image presented is not necessarily that of the real world, but one close enough for the human eye to tolerate.

The rapid increase in computer processing power has allowed a progressively higher degree of realism even for real-time rendering. Real-time rendering is often polygonal and aided by the computer's GPU.

Polygonal modeling is an approach for modeling objects by representing or approximating their surfaces using polygon meshes. The basic object used in mesh modeling is a vertex, a point in three-dimensional space. Two vertices connected by a straight line become an edge. Three vertices, connected to each other by three edges, define a triangle, which is the simplest polygon in Euclidean space. A group of polygons, connected to each other by shared vertices, is generally referred to as an element. Each of the polygons making up an element is called a face.

The flat nature of triangles makes it simple to determine their surface normal. Surface normals are useful for determining light transport in ray tracing, and are a key component of the popular shading models (and the one we use). Note that every triangle has two face normals, which point in opposite directions from each other. In many systems only one of these normals is considered valid – the other side of the polygon is referred to as a backface, and can be made visible or invisible depending on the programmer's desires.

3. Technologies

We decided to develop a real-time rendering model, because of the linear algebra needed in the real-time calculations behind interactions with the object. We used the most popular triangle polygonal approach, which is well-documented and acknowledged in the industry (although voxel rendering, which utilizes regular grids, i. e., cubes, instead of triangles, is becoming more and more popular nowadays) and a flat shading technique, which is relatively simple in technical way compared to others, but utilizes linear algebra methods.

Professional graphic engines may utilize more complex polygons which consist not only of triangles and smarter shading methods, but the aim of this project is to apply linear algebra that is used in the core of the concept of 3D rendering, not some fancy pre-built technologies.

In our engine, meshes are static (they can not change their form in real time using animation), but can be moved (translated, rotated and scaled) relative to the camera. Also, we used a technique with monochrome meshes, where the meshe's single color is chosen at the start, but each polygon is shaded depending on the direction of the light (which, in our realization, can be rotated horizontally in real time). Adding colors to separate polygons or texturing would complicate the project technically (at least, at this stage), but would not bring in more mathematical value.

4. Linear algebra applications: theory and implementation

To render a three-dimensional scene on a two-dimensional display screen, we need to determine where on the screen each vertex of the object should be drawn. 3D graphics systems use homogeneous coordinates, or projective coordinates, to project vertices in four-dimensional space.

4.1. Homogeneous coordinates

If homogeneous coordinates of a point are multiplied by a non-zero scalar then the resulting coordinates represent the same point. Since homogeneous coordinates are also given to points at infinity, the number of coordinates required to allow this extension is one more than the dimension of the projective space being considered. Thus, we should use four homogeneous coordinates $(x, y, z, w)$to specify a point in the projective plane.

To apply the transformations, we should cast a point $(x_0, y_0, z_0)$from the Euclid coordinates to the homogeneous one by assigning the fourth coordinate to the value $w_0 = 1$. When we want to draw the point, we should transfer it back to the Euclid coordinates by simply normalizing all the coordinates by the w_0. Thus, when $w_0 \neq 0$, we achieve a point $(\frac{x_0}{w_0}, \frac{y_0}{w_0}, \frac{z_0}{w_0})$ to be drawn after all the transformations and projections.

4.2. Projection matrix

The projection matrix, the same as all the transformation matrices we will be using, is a 4×4 matrix P, determined as follows:

$$P = \begin{pmatrix} a \cdot \frac{1}{\tan(0.5 \cdot fov)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(0.5 \cdot fov)} & 0 & 0 \\ 0 & 0 & \frac{far}{far - near} & 1 \\ 0 & 0 & \frac{-far \cdot near}{far - near} & 0 \end{pmatrix},$$

where:

$a = \frac{height}{width}$ is the aspect ratio of the screen dimensions;

$fov = \frac{1}{\tan(0.5 \cdot fov_{rad})}$ is the view frustum with fov_{rad}being the horizontal field-of-view value given in radians;

far is the value defining where to display the object in the back by the z (distance from the camera to the far plane);

$near$ is the value defining the front by the z (distance from the camera to the near plane);

4.3. Transformation matrices

All the transformation matrices are represented in the four-dimensional homogeneous coordinate system. Then, the transformation applies with the following steps to each single point of a triangle:

1. Cast 3D vector to homogeneous case
2. Multiply transformation matrix by this vector
3. Cast transformed point to 3D

There are the following transformations implemented in the engine:

1. Translation: $\begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
@staticmethod
def get_translation_matrix(T_x: float, T_y: float, T_z: float) -> np.array:
    return np.array([[1.0,  0.0,  0.0,  T_x],
                     [0.0,  1.0,  0.0,  T_y],
                     [0.0,  0.0,  1.0,  T_z],
                     [0.0,  0.0,  0.0,  1.0]])
```

2. Rotation by x: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
@staticmethod
def get_X_rotation_matrix(degree: float):
    phi = np.deg2rad(degree)
    return np.array([[1.0,        0.0,           0.0,       0.0],
                     [0.0,  np.cos(phi),  -np.sin(phi),  0.0],
                     [0.0,  np.sin(phi),   np.cos(phi),  0.0],
                     [0.0,        0.0,           0.0,       1.0]])
```

3. Rotation by y: $\begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
@staticmethod
def get_Y_rotation_matrix(degree: float):
    phi = np.deg2rad(degree)
    return np.array([[ np.cos(phi),  0.0,  np.sin(phi),  0.0],
                     [      0.0,      1.0,       0.0,      0.0],
                     [-np.sin(phi),  0.0,  np.cos(phi),  0.0],
                     [      0.0,      0.0,       0.0,      1.0]])
```

4. Rotation by z: $\begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 & 0 \\ \sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
@staticmethod
def get_Z_rotation_matrix(degree: float):
    phi = np.deg2rad(degree)
    return np.array([[np.cos(phi),  -np.sin(phi),  0.0,  0.0],
                     [np.sin(phi),   np.cos(phi),  0.0,  0.0],
                     [     0.0,            0.0,       1.0,  0.0],
                     [     0.0,            0.0,       0.0,  1.0]])
```

5. Scale: $\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

```
@staticmethod
def get_scale_matrix(S_x: float, S_y: float, S_z: float) -> np.array:
    return np.array([[S_x,  0.0,  0.0,  0.0],
                     [0.0,  S_y,  0.0,  0.0],
                     [0.0,  0.0,  S_z,  0.0],
                     [0.0,  0.0,  0.0,  1.0]])
```

4.4. Depth

To draw a correctly projected image of a 3D object, we should care about the order of the triangles to draw. Also, we should not consider the triangles which describe not visible planes to reduce the rendering process.

To find out whether the triangle is visible, we implemented the following steps:

1. Find normal vector of the triangle
2. Calculate the dot product of the norm and a camera-triangle direction
3. If it is positive, then the triangle is not visible

```
@staticmethod
def get_normal(triangle: np.array):
   edge_1 = triangle[1] - triangle[0]
   edge_2 = triangle[2] - triangle[0]

   normal = np.cross(edge_1, edge_2)

   return normal / np.linalg.norm(normal)

normal = self.get_normal(triangle)

if np.dot(normal, triangle[0] - self.camera) > 0:
   return None
```

Moreover, we can use the value of the dot product dp of triangle normal and a light direction to define the depth of the planes. Because vectors of the dot product are normalized, the value of dp will be in the range between -1 and 1. Thus, we can use its value to define the color scaler, by multiplying on which we can achieve darker colors of the far planes:

```
@staticmethod
def get_color_scaler(dp: float):
    return (1.5 + dp) / 2.5
```

With that filter, we get the value in the range from 0.2 to 1, and will use this number to multiply each component of the original RGB color. By doing that, the less value becomes, the darker color will be. The lower bound is 0.2 and not less for not making is look completely dark (0 stands for black), while the max value of 1 gives us the original color in the light.

4.5. Light direction rotation

Considering the original vector of a light direction $(0, 0, -1)$, we can multiply it from left by any transformation matrix to give a new source of light. Considering the case with the rotation matrix, we implemented the horizontal light rotation around the object.

5. Pipeline

We have implemented everything we covered in the document, yet it is still not flawless. You can check out the demo in GIF formats on GitHub:

- a cube: https://github.com/MykhailoSobko/3d_graphics_rendering/blob/main/demo/demo_cube.gif
- a spaceship: https://github.com/MykhailoSobko/3d_graphics_rendering/blob/main/demo/demo_plane.gif

Yet the cube demo seems to work perfectly, on a more sophisticated object like spaceship there appeared to be some problems, so all of the rest time we will spend debugging the existing code and trying to fix it.

Adding the textures and more control on light manipulations could be a final point on the project, but it doesn't seem to bring us more mathematical knowledge, so we will only prettify the existing results.