A Complete 3D Graphics Pipeline

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This project implements a complete 3D graphics pipeline including affine transformations, coordinate system conversions, camera orientation, and perspective projection. The implementation features transformation matrices for translation and rotation, world-to-view coordinate conversion, camera look-at functionality, and perspective projection with rasterization. The system renders 3D objects using Gouraud shading and handles camera animation along a circular path with both static and target-following camera behaviors. The project demonstrates fundamental computer graphics techniques using Python with NumPy for efficient matrix operations, providing insights into 3D transformation pipelines essential for computer graphics applications.**

***Index Terms*—computer graphics, affine transformations, perspective projection, camera modeling, coordinate transformations, 3D rendering, rasterization**

A. `translate` *function*

The `translate` function constructs a 4×4 affine transformation matrix that represents a translation in 3D space. It takes a translation vector `t_vec` of shape $(3,)$ or $(1,3)$ as input, which defines the displacement along the x, y, and z axes. The function initializes a 4×4 identity matrix and replaces the last column of the top three rows with the components of the translation vector, effectively encoding the translation into the matrix. The resulting matrix can then be used to transform 3D points or objects in homogeneous coordinates by shifting their position without altering their orientation or scale.

To verify the correctness of the `translate` function, a test was performed using the translation vector $\texttt{t\_vec} = [1, 2, 3]$. The resulting transformation matrix was:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This output matches the expected form of a 3D affine transformation matrix that performs a pure translation. The last column correctly reflects the input translation vector, while the rest of the matrix remains the identity matrix, indicating no rotation or scaling. Therefore, the function is working as intended.

Detailed code for this function can be found in the translate_func.py file.

This document was prepared by Dimitrios Karatis as part of the second assignment for the Computer Graphics course at Aristotle University of Thessaloniki.

B. `rotate` *function*

The `rotate` function constructs a 4×4 homogeneous transformation matrix that performs a rotation in 3D space. It takes as input a rotation axis (a 3-element vector), a rotation angle in radians, and an optional center of rotation (defaulting to the origin). The axis vector is first normalized, and the rotation matrix is computed using Rodrigues' rotation formula, which is suitable for rotating around an arbitrary axis. The resulting 3×3 rotation matrix is then embedded into the upper-left corner of a 4×4 identity matrix to create a homogeneous transformation. If a rotation center other than the origin is provided, the function applies a translation to move the space so that the rotation occurs about the specified center, and then translates it back.

To validate the `rotate` function, a test was conducted by rotating $\frac{\pi}{2}$ radians (90 degrees) about the z-axis, with the rotation centered at the point $(1, 0, 0)$. The resulting transformation matrix was:

$$\begin{bmatrix} 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix corresponds to a correct 90-degree counterclockwise rotation around the z-axis, while also accounting for the specified center of rotation. The rotation is clearly visible in the top-left 2×2 block, and the translation components in the last column adjust the position to rotate around $(1, 0, 0)$ instead of the origin. The result confirms that the `rotate` function is operating as expected.

Detailed code for this function can be found in the rotate_func.py file.

C. `compose` function

The `compose` function performs the composition of two 4×4 affine transformation matrices by matrix multiplication. It takes as input two transformation matrices, `mat1` and `mat2`, and returns their product, computed as `mat1 @ mat2`. This operation combines the effects of the two transformations into a single matrix. The order of multiplication follows the standard rules of matrix operations, meaning that `mat2` is applied first, followed by `mat1`. This is consistent with the right-to-left nature of transformation composition in homogeneous coordinates. The function includes a shape check to ensure that both input matrices are valid 4×4 affine transformations.

To verify the correctness of the `compose` function, a test was conducted by combining a translation and a rotation transformation. A translation matrix was defined to shift points by $(2, 3, 5)$, and a rotation matrix was defined for a $90°$ counterclockwise rotation about the z-axis. The composition was performed as `compose(translation, rotation)`, meaning the rotation is applied first, followed by the translation. The resulting matrix was:

$$\begin{bmatrix} 0 & -1 & 0 & 2 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix correctly reflects the expected sequence of transformations: the top-left 2×2 block corresponds to a $90°$ rotation about the z-axis, and the last column indicates the subsequent translation. Therefore, the output confirms that the `compose` function operates correctly and respects the proper order of transformation application.

Detailed code for this function can be found in the compose_func.py file.

D. `world2view` function

The `world2view` function transforms 3D points from world coordinates into camera view coordinates, simulating the perspective of a virtual camera in a 3D scene. It takes three inputs: `pts`, a $3 \times N$ matrix (or alternatively an $N \times 3$ matrix) representing N 3D points in world coordinates; `R`, a 3×3 rotation matrix that defines the orientation of the camera; and `c0`, a 3D vector specifying the position of the camera in world coordinates. Internally, the function reshapes and verifies the input data as necessary and then applies the transformation formula:

$$\texttt{view\_coords} = R \cdot (\texttt{world\_coords} - \texttt{c0}),$$

which translates the points relative to the camera's position and then rotates them into the camera's reference frame. The output is returned as an $N \times 3$ matrix of points in camera coordinates.

To test the `world2view` function, three points in world coordinates were transformed using a camera rotation of $90°$ about the z-axis and a camera positioned at $(1, 1, 1)$. The input points, represented as a 3×3 matrix, were translated by subtracting the camera position and then rotated by the camera's orientation matrix. The resulting view coordinates were:

$$\begin{bmatrix} -3 & 0 & 6 \\ -4 & 1 & 7 \\ -5 & 2 & 8 \end{bmatrix}$$

This output correctly reflects the expected transformation, as the rotation rotates points counterclockwise around the z-axis, and the translation accounts for the camera position. The function correctly converts the points from world coordinates into the camera's view frame, confirming its proper operation.

Detailed code for this function can be found in the world2view_func.py file.

E. `lookat` function

The `lookat` function computes the camera's view transformation given the camera's position (`eye`), the up direction vector (`up`), and the target point (`target`) the camera is looking at. It returns a rotation matrix R and a translation vector t that together transform points from world coordinates to camera coordinates using the formula:

$$\text{camera\_coords} = R \times (\text{world\_coords} - t).$$

The function first calculates the camera's forward axis as the normalized vector pointing from the camera position to the target. Then, it computes the right axis as the normalized cross product between the forward vector and the up vector, ensuring orthogonality. The true up vector is recalculated as the cross product of the right and forward vectors to maintain an orthonormal basis. These three vectors form the rows of the rotation matrix R, which aligns the world coordinate system with the camera's coordinate system. The translation vector t represents the camera's position in world coordinates.

To test the correctness of the `lookat` function, the camera was positioned at coordinates $(2, 3, 5)$, looking towards the origin $(0, 0, 0)$ with the up direction aligned to the world's y-axis $(0, 1, 0)$. The function returned a rotation matrix R and a translation vector t, representing the camera's orientation and position, respectively. Transforming the world coordinate of the target point (the origin) into camera coordinates using the formula $\mathbf{p}_{camera} = R(\mathbf{p}_{world} - t)$ produced the coordinates:

$$[-2.937, -5.054, -1.957].$$

This output correctly reflects the position of the origin relative to the camera's viewpoint, confirming that the function properly constructs an orthonormal camera coordinate system and correctly transforms points from world space to camera space.

Detailed code for this function can be found in the lookat_func.py file.

F. `perspective_project` *function*

The `perspective_project` function performs the projection of 3D points onto a 2D image plane using the pinhole camera model. It takes as inputs a set of 3D points in world coordinates (`pts`), the camera's focal length (`focal`), a rotation matrix R that transforms points from world to camera coordinates, and a translation vector t representing the camera's position. The function first converts the 3D points to the camera coordinate system by applying the rotation and translation using the `world2view` function. It then extracts the depth values (distance along the camera's viewing axis) for each point. Finally, it computes the 2D projected coordinates on the image plane by scaling the x and y camera coordinates with the focal length and normalizing by the depth, thereby implementing the perspective projection. The function returns the 2D image coordinates as a $2 \times N$ matrix along with the corresponding depth values as a $1 \times N$ array.

To verify the correctness of the `perspective_project` function, a set of sample 3D points was projected onto a 2D image plane using a focal length of 1.0. The camera was assumed to be aligned with the world coordinate axes (identity rotation matrix) and positioned at the origin (zero translation vector). The function computed the projected 2D points and their corresponding depths along the camera's viewing direction. The resulting projected points,

$$\begin{bmatrix} 0.143 & 0.25 & 0.333 \\ 0.571 & 0.625 & 0.667 \end{bmatrix},$$

correctly represent the normalized image coordinates obtained by dividing the x and y world coordinates by the depth values $[7, 8, 9]$. This outcome matches the expected behavior of the pinhole camera model, confirming that the function accurately performs perspective projection.

Detailed code for this function can be found in the perspective_project_func.py file.

G. `rasterize` *function*

The `rasterize` function converts 2D points given in camera plane coordinates into corresponding pixel coordinates on a discrete image grid. It takes as input the 2D points in a continuous coordinate system centered at the camera plane, the physical width and height of the camera plane in world units, and the desired resolution of the output image in pixels (width and height). The function computes scale factors to map world units to pixels and shifts the points so that the origin of the camera plane aligns with the center of the image. Since the camera plane uses an upward positive y-axis while image pixel coordinates typically have the origin at the bottom-left with y increasing upwards, the function flips the y-axis accordingly. Finally, it rounds the coordinates to the nearest integer pixel indices and clips them to ensure they lie within the image bounds. The output is a $2 \times N$ matrix of integer pixel coordinates suitable for image rasterization.

To verify the correctness of the `rasterize` function, a set of 2D points in camera coordinates was mapped onto pixel coordinates of an image with resolution 640×480 pixels. The camera plane dimensions were set to 2.0 meters in width and 1.5 meters in height. The function correctly scaled and translated the input points so that the origin in the camera plane corresponds to the image center. For example, the point at $(0, 0)$ in camera coordinates maps to the pixel $(320, 240)$, which is the center of the image. Similarly, positive and negative coordinates were converted appropriately, respecting the axis directions and image boundaries. The output pixel coordinates,

$$\begin{bmatrix} 320 & 480 & 160 \\ 240 & 144 & 304 \end{bmatrix},$$

demonstrate that the function correctly performs the rasterization process, confirming its accuracy.

Detailed code for this function can be found in the rasterize_func.py file.

H. `render_object` *function*

This function implements the complete 3D rendering pipeline using a pinhole camera model to render a textured 3D object from a given camera viewpoint. It accepts as input the 3D vertex positions (`v_pos`), texture coordinates (`v_uvs`), triangle connectivity (`t_pos_idx`), and camera parameters including position (`eye`), orientation (`up`), and target point (`target`). The rendering process begins by computing the camera's view matrix through the `lookat` function, which yields the rotation and translation required to transform world coordinates to camera space. Next, the 3D vertices are projected onto the image plane using `perspective_project`, producing 2D coordinates and associated depth values. These are then converted to discrete pixel coordinates via the `rasterize` function. The 2D pixel positions are combined with depth to form a complete vertex representation, which is passed to `render_img`—a function that handles triangle rasterization and texture mapping using UV coordinates. The final result is a 512×512 RGB image depicting the textured object from the camera's perspective.

Detailed code for this function can be found in the render_object_func.py file.

I. `make_video_from_frames` function

The `make_video_from_frames` function compiles a sequence of image frames into an MPEG-4 video file. It takes as input the folder containing the individual frame images, the desired output video name, and the frame rate (defaulted to 25 frames per second). The function assumes the frames are named sequentially as `frame_000.png`, `frame_001.png`, ..., `frame_124.png`, and loads them into a list using the `imageio` library. Once all frames are collected, it uses `imageio.mimsave` to encode and save the video in MP4 format under the specified output name. This utility is used to generate animation demos from the rendered frames of the 3D scenes, fulfilling the requirement of producing 5-second simulations at 25 FPS as stated in the project description.

Detailed code for this function can be found in the make_video_from_frames_func.py file.

J. `generate_forward_demo` function

The `generate_forward_demo` function simulates and renders a dynamic 3D scene where a camera, mounted on a car moving along a circular path, always looks in the forward direction of motion. It begins by loading all necessary scene data from a file (`hw2.npy`) including object geometry, camera intrinsics, and trajectory parameters. The car moves along a circle of known center and radius, and its position for each frame is computed based on angular velocity derived from the given linear speed. The camera is rigidly attached to the car with a fixed offset and is oriented to look along the tangent of the circular path, simulating forward-facing behavior. For each frame of the animation, the camera's world-space position and target direction are calculated. Then, the `render_object` function is called to render the textured object (a pyramid) from the current camera viewpoint using a perspective pinhole model. The rendered image is saved to a dedicated output folder with sequential filenames. After generating all 125 frames (for a 5-second animation at 25 FPS), the function calls `make_video_from_frames` to compile the image sequence into a video file named `demo_forward_video.mp4`.

Detailed code for this function can be found in the demo_forward_func.py file.

K. `generate_target_demo` function

The `generate_target_demo` function creates a dynamic 3D animation in which a camera, mounted on a moving car that follows a circular trajectory, continuously looks at a fixed point in space defined by `k_cam_target`. After loading the necessary scene data from the file `hw2.npy`—including object geometry, UV coordinates, triangle indices, and camera parameters—the function calculates the car's position for each frame based on its angular velocity around the circle. The camera's position is determined by adding a fixed offset to the car's position, simulating a rigid mounting. Unlike the forward-looking demo, here the camera does not align with the car's direction of motion, but instead always points toward the static target point, providing a distinct viewing experience. The function then renders each frame of the animation using the `render_object` function, which handles camera transformation, projection, rasterization, and texture mapping. Each frame is saved as a PNG image in an output folder named after the `demo_id`. Once all 125 frames (5 seconds at 25 FPS) are generated, the function calls `make_video_from_frames` to compile them into a video file named `demo_target_video.mp4`, showcasing the effect of constant target tracking during circular motion.

Detailed code for this function can be found in the demo_target_func.py file.

L. `test_load` function

The `test_load` function serves as a utility for inspecting the contents of the `hw2.npy` file, which stores all necessary data for the 3D rendering task. It uses NumPy's `load` function with `allow_pickle=True` to deserialize the file into a Python dictionary. It then prints all the available keys in the dataset along with the data type and shape of each corresponding value. This diagnostic tool is especially useful for understanding the structure of the input data, verifying that it has been loaded correctly, and checking that all required parameters—such as vertex positions, triangle indices, camera intrinsics, and animation parameters—are present and properly formatted before being used in the rendering pipeline.

The `hw2.npy` file contains a Python dictionary with 14 key-value pairs that provide all necessary parameters and geometry for the rendering task. The keys and their respective types and shapes are as follows: camera configuration includes `k_cam_up` (3×1 `ndarray`), `k_sensor_height` (int), `k_sensor_width` (int), and focal length `k_f` (int); vehicle and animation parameters include `car_velocity` (float), `k_road_radius` (float), `k_road_center` (3D `ndarray`), `k_cam_car_rel_pos` (3D `ndarray`), `k_duration` (int), and `k_fps` (int). The geometry of the 3D object is described by `v_pos` (3×16 `ndarray` of vertex positions), `v_uvs` (16×2 `ndarray` of UV coordinates), and t_pos_idx (6×3 triangle index array of type trimesh.caching.TrackedArray). Additionally, the key k_cam_target provides a static 3D target point as a (3×1 ndarray).

Detailed code for this function can be found in the test_load_func.py file.

FUNCTIONS FROM THE PREVIOUS ASSIGNMENT

M. vector_interp function

The vector_interp function performs linear interpolation of vectors between two known endpoints along either the x-axis or y-axis, as specified by the dim parameter. Given two 2D points p1 and p2, and their associated vector values V1 and V2, the function computes an interpolation factor $t \in [0, 1]$ based on the position of a specified coordinate within the interval defined by p1 and p2. It then linearly blends the vectors using this factor to return the interpolated result. This function is essential in the scanline rasterization process, as it enables the smooth interpolation of attributes such as UV coordinates and vertex colors across triangle edges and horizontal lines. Special care is taken to avoid division by zero in the case of vertically or horizontally aligned points.

Detailed code for this function can be found in the vector_interp_func.py file.

N. t_shading function

The t_shading function performs rasterization and shading of a single triangle using scanline rendering with Gouraud shading and texture mapping. It takes as input the triangle's vertex positions in screen space (including depth), corresponding UV texture coordinates, vertex colors, and the texture image. The vertices are first sorted by their y-coordinate to facilitate scanline traversal. For each horizontal scanline intersecting the triangle, the function computes the left and right endpoints by linearly interpolating along triangle edges using the helper function vector_interp. It then linearly interpolates the UV coordinates and vertex colors across the scanline. At each pixel within the scanline, the UV coordinate is mapped to the texture to sample the corresponding texel color. This texel color is then multiplied by the interpolated vertex color to produce the final shaded pixel. This approach results in smooth color transitions (Gouraud shading) and detailed surface appearance via texture mapping, yielding visually realistic rendering of the triangle.

Detailed code for this function can be found in the t_shading_func.py file.

O. render_img function

The render_img function is responsible for rasterizing and shading the 3D mesh onto a 2D image. It receives as input the mesh triangle indices (faces), vertex screen-space positions with depth (vertices), per-vertex colors (vcolors), UV texture coordinates (uvs), and a texture image (texImg). The function begins by calculating the average depth of each triangle and sorting them in descending order to ensure proper back-to-front rendering using the Painter's Algorithm. Each triangle is then processed individually: its vertex positions, colors, and UV coordinates are extracted, and the t_shading function is invoked to apply texture mapping and shading using perspective-correct barycentric interpolation. The function ultimately returns a 512×512 RGB image that represents the fully textured and shaded 3D object from the current camera view.

Detailed code for this function can be found in the render_img_func.py file.

RESULTS

The output of the project is organized into two folders: demo_forward and demo_target. Each folder contains 125 rendered frames in PNG format, corresponding to 5 seconds of animation at 25 frames per second, as required by the assignment. Additionally, each folder includes a compiled video—demo_forward_video.mp4 and demo_target_video.mp4—which visually demonstrate the two camera configurations described in the problem statement. To generate these outputs, the user simply runs the corresponding Python scripts using python3 demo_forward.py or python3 demo_target.py. These scripts handle the rendering of individual frames and the construction of the final video.

Shown below are two indicative frames from each demo folder: one from the beginning and one from the end of the animation, specifically the first and last frames of both the demo_forward and demo_target sequences:

DEMO_FORWARD

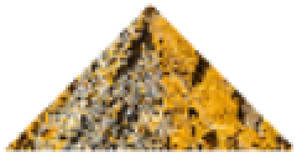
Fig. 1: demo_forward first frame

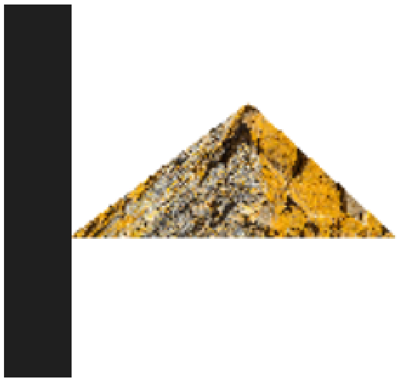
Fig. 2: demo_forward last frame

DEMO_TARGET

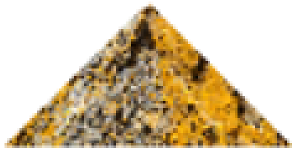
Fig. 3: demo_target first frame

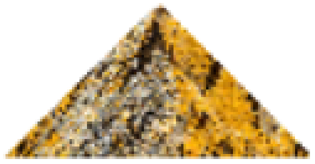
Fig. 4: demo_target last frame

OBSERVATIONS AND EVALUATION

Upon inspecting the generated videos, we can make the following observations. In the `demo_forward_video`, the object (the textured pyramid) appears to start in the center of the frame and gradually moves to the left side as the animation progresses. Additionally, a noticeable distortion in the shape of the object is observed during the motion. This suggests that the camera is moving along the circular trajectory with the car and looking in the direction of motion, as expected. According to the problem statement, in this scenario the camera should be fixed relative to the car and always oriented forward, meaning that the camera's optical axis (i.e., $\hat{z}_c$) should remain aligned with the instantaneous velocity vector of the vehicle. Despite the visual artifacts, the overall behavior is consistent with this intended setup.

In contrast, the `demo_target_video` demonstrates a different behavior: the object remains visually stable and centered throughout the duration of the video. This indicates that the camera is dynamically adjusting its orientation so as to continuously point toward a fixed target in world space, namely the point `k_cam_target` provided in the dataset. This behavior aligns precisely with what the `demo_target` should do based on the assignment, which specifies that as the car moves along the circular road, the camera may rotate and must always look at the given static target point.