Assignment 4

University of Genoa

Master In Computer Science - Computer Graphics - 109186

A.A 2023-2024

Gabriele Francesco Berruti

20 November 2024

Index

Chapter 1

Ex.1

1.1 Rasterization Basics

In this exercise, you will implement a rendering system for triangulated 3D models based on rasterization.

Using Eigen

In all exercises, you will need to do operations with vectors and matrices. To simplify the code, you will use Eigen.

Preparing the Environment and Submission

Follow the instructions on the General instructions document to set up what you need for the assignment. Note: For the purpose of exercises 1 to 2, the world space, camera space, and canonical viewing volume will be one and the same.

1.2 Understand the Structure of the Assignment

Assignment 4 Directory Structure

The **Assignment 4** directory contains the following:

- **data/**: Input data files (e.g. JSON configurations) different for each exercise (ex. bunny_ex1 , bunny_ex2 ecc...)
- **img/**: Images, including input graphics or output visualizations.
- **Report/**: Documents describing the analysis or results.
- **src/**: Source code organized in subdirectories (e.g., `ex1`, `ex2`), each containing:
  - `main.cpp`: Main program file for the exercise.
  - `raster.h`, `raster.cpp` , `attributes.cpp`: Header and implementation files.

Additionally, the root directory includes a `run.sh` script to automate compilation and execution.

Functionality of the `run.sh` Script

The `run.sh` script performs the following tasks:

1. **Argument Validation:** Checks if an exercise identifier (e.g., `ex1`, `ex2`) is provided.
2. **File Configuration:** Updates `CMakeLists.txt` to point to the correct source files for the selected exercise.
3. **Build Setup:**
   - Deletes and recreates the `build/` directory.
   - Runs `cmake` in `RELEASE` mode and compiles the project.
4. **Execution:** Runs the compiled binary (`assignment4`) with a predefined data file, depending on the exercise (e.g., `bunny_ex4.json` for `ex4`).

Example of using the script :

./run.sh ex1
./run.sh ex2 f
./run.sh ex3 v
./run.sh ex4 w

1.3 Ex.1: Load and Render a 3D model

Order of execution and main functions:

1. `main`
   `int main(int argc, char *argv[])`
   The entry point of the program. It initializes parameters like the framebuffer dimensions (`width`, `height`), sets up shaders, and loads the scene from a JSON file. Key operations include:
   - Initializing the `Framebuffer` to store rasterized output.
   - Configuring the `VertexShader`, `FragmentShader`, and `BlendingShader`.
   - Loading the scene and invoking the rasterization pipeline.
2. `load_scene`
   `void load_scene(const std::string& filename, Program& program, UniformAttribut uniform, FrameBuffer& frameBuffer)`
   This function parses the JSON scene file and sets up:
   - Ambient light, and camera parameters (e.g., projection type, field of view, near/far planes).
   - Create Projection and camera matrices using helper functions like `create_camera_matrix` and `perspective_matrix`.
   - Read Materials, lights, and objects in the scene.

For each object/Mesh , vertices are loaded, transformations specific to the object (model matrix) are applied, and triangles are rasterized using `rasterize_triangles`.

3. `load_off`
   `void load_off(const std::string& filename, vector<VertexAttributes>& v)`
   This function reads vertex and facet data from an `.off` file. Each triangle's vertices are transformed into `VertexAttributes` and stored for rasterization inside the v variable .

4. `create_model_matrix`
   `Matrix4f create_model_matrix(const Vector3f& translation, float angle_x, float angle_y, float angle_z, const Vector3f& scale_factors)`
   Computes the model transformation matrix, combining translation, rotation, and scaling transformations. This matrix maps object coordinates to world coordinates.

5. `create_camera_matrix`
   `Matrix4f create_camera_matrix(Vector3f& eye, Vector3f& target, Vector3f& up)`
   Generates the camera transformation matrix, which converts world coordinates into camera coordinates. This is based on the camera's position, orientation, and target position observed .

6. `ortographic_matrix` and `perspective_matrix`
   `Matrix4f ortographic_matrix(Vector3f top_right_corner, float near, float far, float aspectRatio)`
   `Matrix4f perspective_matrix(float near, float far, float fov, float aspectRatio)`
   Both functions generate projection matrices:

   - `ortographic_matrix` performs an orthogonal projection, mapping camera space to the canonical view volume.
   - `perspective_matrix` applies a perspective transformation, preserving depth perception.

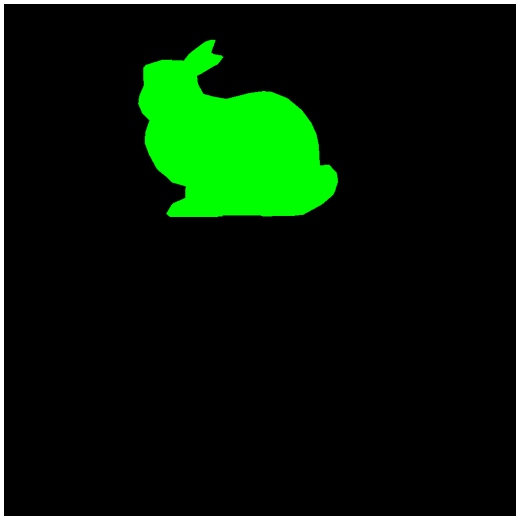
7. `rasterize_triangles`
   This function processes triangle vertices to rasterize them directly onto the framebuffer using the specified shaders. Each triangle is rendered with uniform color.

8. `framebuffer_to_uint8` and `stbi_write_png`
   Converts the framebuffer's data into an 8-bit format and writes it as a PNG image file, producing the final rendered output.

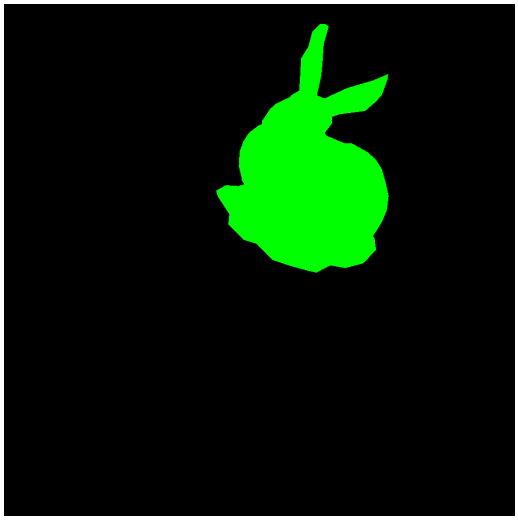
Shader Functions

- `VertexShader`: Transforms vertices from object space to clip space. Applies model, camera, and projection transformations in sequence and trasform the position in Homogeneous coordinate .

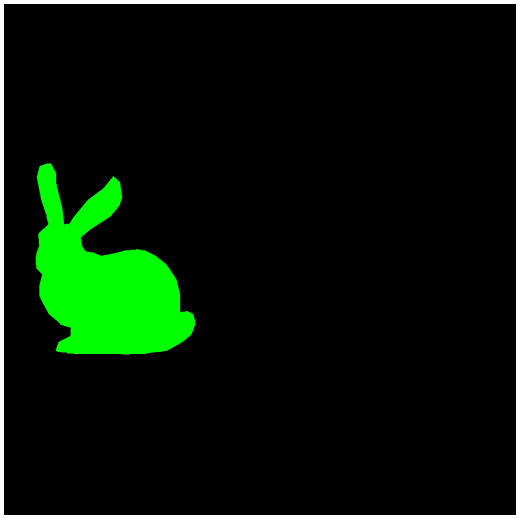
- `FragmentShader`: Assigns a fixed color to each fragment.
- `BlendingShader`: Converts color values to the $[0, 255]$ range for rendering.



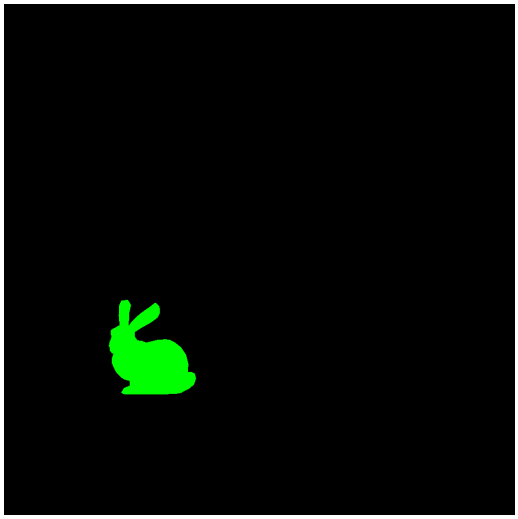
(a) Example Base Scene orthographic with green color.



(b) Example Base Scene orthographic rotated.



(c) Example Base Scene Prospective.



(d) Example Base Scene Prospective in distance

Figure 1.1: Examples on orthographic and prospective Projection .

Chapter 2

Ex.2

2.1 Shading

In this exercise, we extend the code to implement different shading options for the triangles, including the addition of a light source. The scene will render in three modes:

Shading Mode

1. **Wireframe:** Draws only the edges of the triangles.
2. **Flat Shading:** Each triangle is rendered with a uniform normal, corresponding to the plane containing it. The wireframe is overlaid on top.
3. **Per-Vertex Shading:** Normals are assigned to each vertex, and colors are interpolated across the triangle's interior.

Additional Methods and Modifications with respect to Ex1 :

- Introduced the `ShadingMode` enum to manage rendering modes (`WIREFRAME`, `FLAT_SHADING`, `PER_VERTEX_SHADING`).
- Updated vertex and fragment shaders to compute normals and apply lighting based on the active shading mode.
- Added functions for computing face normals (`compute_face_normals`) and vertex normals (`compute_per_vertex_normals`).

```
enum ShadingMode { WIREFRAME, FLAT_SHADING, PER_VERTEX_SHADING };
ShadingMode shadingMode = WIREFRAME; // Default mode

// Computes face normals for flat shading
void compute_face_normals(const MatrixXf &vertices,
                          const MatrixXi &facets,
                          MatrixXf &face_normals,
                          Vector3f center_of_model);

// Computes vertex normals for per-vertex shading
```

```
void compute_per_vertex_normals(const MatrixXf &vertices,
                                const MatrixXi &facets,
                                 const MatrixXf &face_normals,
                                 MatrixXf &vertex_normals);
```

Compute Vertex Normal , Face Normal and Compute center of the 3D Model

Compute Center of the 3D Model

This function computes the geometric center of the 3D model by averaging the coordinates of all vertices. The resulting center serves as a reference point for computing normal .
To obtain it we sum of the 3 vertex point and average it so we find the face center . After that we average all the face center . Pay attention we do not take care of the area of each face to compute the center .

Face Normal

The face normal computation involves determining the normal vector perpendicular to each triangular face. To compute a correct normal first of all we compute the vector that connect the point to the center of the model in Object coordinate System . In case the normal computed point in a different direction with respect to the vector that point to the center of the object so the normal make sense .
If it's not the case it's necessary to re-compute the normal to point in the opposite direction and we assert that this is true .

Compute Vertex Normal

This function calculates the normal vector at each vertex of the 3D model. It averages the normals of the adjacent faces that the vertex touch . We iterate on the faces and sum up the face normal . After for each vertice we divide with the number of facets that it touch and we normalize .

Main Program Modifications:

- Integrated shading selection via command-line arguments (`wire`, `flat`, `vertex`).
- Applied normals in the vertex shader and manage the normal computed before to pass to the BlendingShader :
- And compute the transformation of the point in the Projection space which has to be returned to Camera Space in the Blending Shader .

```
if (shadingMode == PER_VERTEX_SHADING) {
    FinalNormal = va.vertex_normal;
    out.vertex_normal = FinalNormal;
} else {
    FinalNormal = va.face_normal;
    out.face_normal = FinalNormal;
```

```
}
```

- Extended the fragment shader to calculate illumination using the Phong model (ambient, diffuse, and specular) as in Assignment 2/3 .

Scene Loader Updates:

- Modified the `load_scene` function to handle new rendering modes:

```
if (shadingMode == WIREFRAME) {
    rasterize_lines(...);
} else if (shadingMode == FLAT_SHADING) {
    rasterize_triangles(...);
    rasterize_lines(...); // Overlay wireframe
} else {
    rasterize_triangles(...); // Per-vertex shading
}
```

- Implemented the OFF loader to store vertex attributes for each shading mode. For example when implementing FLAT SHADING with WIREFRAME over it we need to store the WIREFRAME vertex attributes and the FLAT SHADING vertex attributes. And the size of the wireframe vertex vector will be double the dimension of the flat vertex attributes to rasterize it .

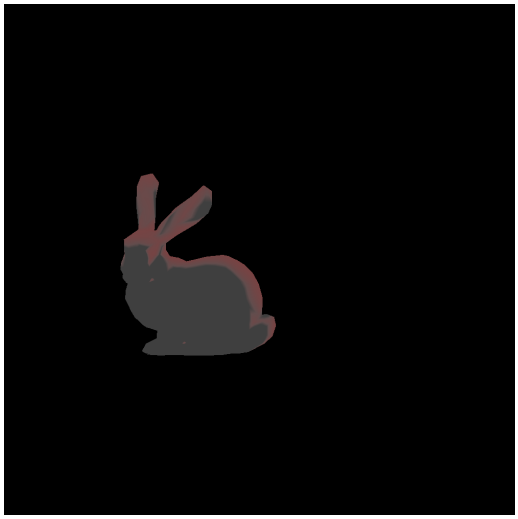
Output:

Images are saved to the `../img/ex2/` folder for each shading mode.

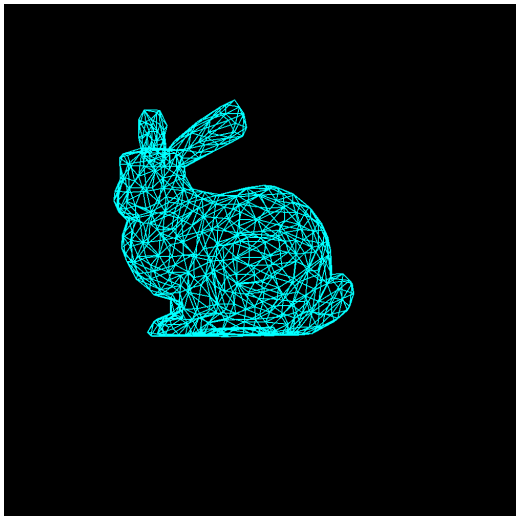
(a) Example Base Scene orthographic with changed color.

(b) Example Base Scene orthographic chnage in Theta.

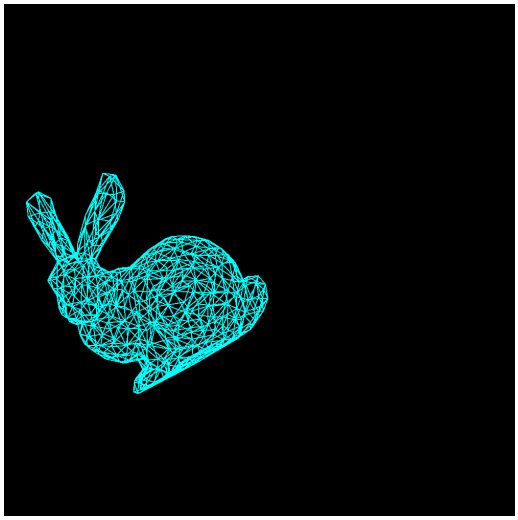
(c) Example Base Scene Ortographic Per Vertex Rotated



(d) Example Base Scene Prospective Per Vertex in distance



(e) Example Base Scene Ortographic Wireframe



(f) Example Base Scene Prospective Wireframe Rotated

Figure 2.1: Examples Ex2 on orthographic and prospective Projection on FLAT , WIREFRAME and PER VERTEX shading

Chapter 3

Ex.3

3.1 Object Transformation

The primary goal for this exercise is to add transformations to the vertex shader. Specifically:

1. Rotate the object around its barycenter.
2. Translate the object toward the camera.
3. Modify the normal based on the transformation applied

The additional logic in the vertex shader to achieve these transformations is shown below in the following functions:

Rotate Point around the barycenter of the Object

```
Matrix4f compute_rotation_around_center(const Vector3f& center,
float frame,
float rotation_speed_along_axis,
float translation_speed_toward_camera,
const Matrix4f& model_matrix)
```

This function rotates an object about its center by computing a transformation matrix. Based on the current frame and rotation speed, it rotates the object around the Z-axis after first translating it to and from its center. The object is then moved in the direction of the camera using a translation. Lastly, it returns the full transformation matrix after combining the resultant transformations with the object's model matrix.

Compute Transformed Normal

```
Vector4f compute_transformed_normal(const Vector4f& input_normal,
                                    const Matrix4f& model_transform,
                                    const Matrix4f& camera_matrix)
```

The altered normal vector is calculated using this function. To appropriately account for non-uniform scaling, it first applies the inverse transpose of the model

transformation matrix to the input normal. Next, it uses the transpose of the camera's rotation matrix to convert the normal into camera space. Since it represents a normal rather than a point, the resultant normal is normalized and returned as a 4D vector with the fourth component set to 0.

GIF Rendering

To demonstrate the object transformations, three GIF animations are generated, one for each shading type:

- **Wireframe Rendering**
- **Flat Shading**
- **Per-Vertex Shading**

The animation process captures a sequence of frames with rotation and translation applied. These frames are saved to a GIF using the `gif.h` library, as described in the code:

```
const int num_frames = 25;  // Number of frames
const float animation_speed = 0.1; // Speed of rotation
GifWriter g;
GifBegin(&g, fileName, frameBuffer.rows(), frameBuffer.cols(), delay);
```

Output:

Images are saved to the `../img/ex3/` folder for each shading mode.

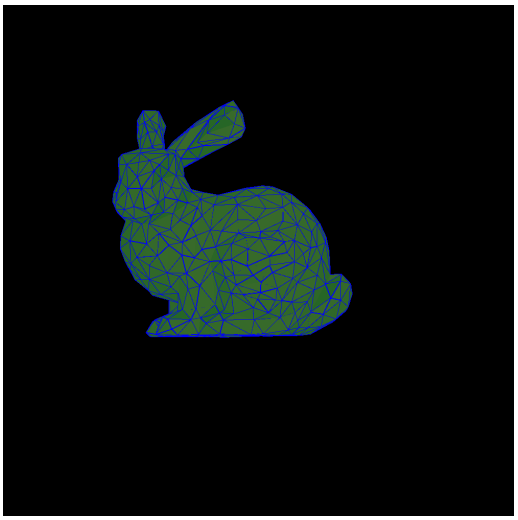


Figure 3.1: Flat GIF

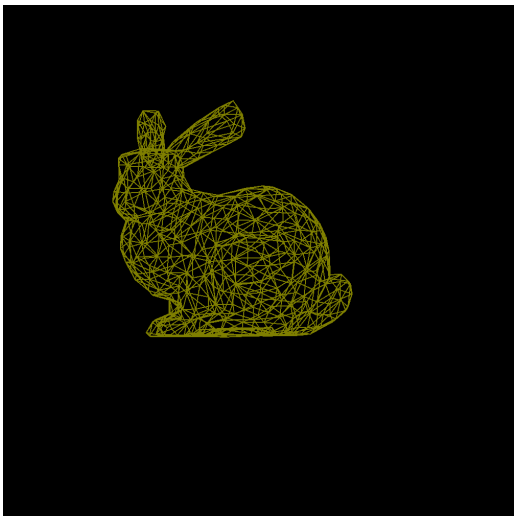


Figure 3.2: Wire GIF

Figure 3.3: Vertex GIF

Chapter 4

Ex. 4

4.1 Implement a perspective camera and add support for multiple resolutions

Perspective Camera and Framebuffer Adaptation

- Adjusted the aspect ratio dynamically based on framebuffer size to avoid image distortion during resizing.

```
int height=1000;
int width=500;
.....
// Manage  the aspect Ratio in Prospective Projection
if (aspectRatio < 1)
        MProj(0,0) *= aspectRatio;
else
        MProj(1,1) *= (1/aspectRatio);
```

Lighting equation need to be computed in the camera space

Lighting Equation in Camera Space

- Ensured shading correctness by calculating the lighting equation in the camera space (or eye space).
  To do it it's necessary to return to the camera space in the Blending Shader with the Inverse Projection Matrix just to compute the illumination equation using the Phong model .

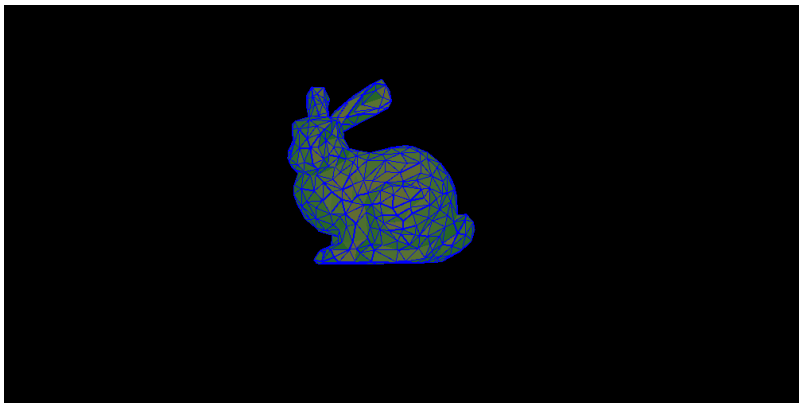


Figure 4.1: Flat GIF

Figure 4.2: Wire GIF

Figure 4.3: Vertex GIF