Middle East Technical University North Cyprus Campus Computer Engineering Program

CNG 477

Introduction to Computer Graphics

Fall '2018-2019

Assignment 2 - A Software Rasterizer implementing the Forward Rendering Pipeline

Due date: January 4, 2019, Friday, 23:55

1 Objectives

In this assignment, you are going to implement Modeling Transformation, Viewing Transformation, and Rasterization stages of the Forward Rendering Pipeline. Specifically, given a set of triangles and the position and the color attributes of all of their vertices, your goal is to render the scene as a two-dimensional (2D) image. The parameters of a perspective view camera will be specified. You will transform all of the vertices to the viewport and then use line drawing and triangle rasterization algorithms to display the triangles in wireframe or solid modes. You will also implement the backface culling (for both modes) for correct rendering of the scene. You will write your implementations in C/C++ language.

Keywords: *forward rendering pipeline, modeling transformations, viewing transformations, line drawing, triangle rasterization, interpolation, backface culling*

2 Specifications

1. Name of the executable will be "rasterizer".
2. Your executable will take two .txt files as argument: one that describes the scene and one that describes the cameras (e.g. "sample_scene.txt sample_camera.txt"). Input file names are not static, they can be anything.
3. The scene will only be composed of instances of triangles. A set of triangles will comprise a model with a sequence of several transformations (translation, rotation, scaling) you will be able to move, rotate, or resize a model (i.e., all of the triangles in the model). Transformations will be applied to the models in the order specified in the input file.
4. You will not implement any lighting computations in this assignment. The color of each vertex will be provided as input and your goal will be to interpolate color along the edges and the surfaces of the triangles in wireframe and solid modes, respectively.

5. Use the midpoint algorithm to draw triangle edges and use the barycentric coordinates based algorithm to rasterize triangle faces.

6. In both wireframe and solid modes, triangles whose backface is facing the viewer will be culled. Backface culling should be enabled/disabled according to the setting in input file. **Default value for backface culling setting is 0 (disabled).**

7. You will not implement any clipping algorithm and you may assume that the camera is positioned so that the whole scene is within the viewing volume.

8. The camera description file may contain multiple camera configurations. Your program should generate a separate output for each configuration to the output files specified by the input file.

9. **Here is the good news. You will NOT deal with input files.** Helper functions for reading inputs, input reading sections of the "rasterizer.c"/"rasterizer.cpp" file, helper functions for mathematical operations (e.g. normalization, matrix multiplication) are given to you in assignment file. It is **STRONGLY** recommended to inspect types and helper functions inside "hw2_types.h", "hw2_file_ops.c"/"hw2_file_ops.cpp" and "hw2_math_ops.c"/"hw2_file_ops.cpp" files.

10. Since you will not deal with input reading, memory allocation for the image and implementing "main()" function, **you will only implement "forwardRenderingPipeline()" function** inside the code. You can also write helper functions by yourself. If you decide to implement additional functions, add them to "rasterizer.c"/"rasterizer.cpp" file.

11. **Last important note: You will NOT implement depth buffer algorithm in this homework.** Models will be given to you in back-to-front order. So, when you draw a model, you can assume that next model is closer to the camera than previous models.

3 Camera File

- **Camera Count** Integer
- **Camera "Cid"** id
  - · **Position** X Y Z
  - · **Gaze** X Y Z
  - · **Up** X Y Z
  - · **ImagePlane** Left Right Bottom Top Near Far HorRes VerRes
  - · **OutputName** <image_name>.ppm

1. Camera Count
Number of cameras that will be used for producing images of the scene.

2. Camera

1. **Position** defines the X, Y, Z coordinates of the camera.
2. **Gaze** defines the direction that the camera is looking at.
3. **Up** defines the up vector of the camera. The up vector is not necessarily given as orthogonal to the gaze vector. Therefore the camera's x-axis is found by a cross product of the gaze and up vectors, then the up vector is corrected by a cross product of gaze and x-axis vectors of the camera.
4. **ImagePlane** defines: the coordinates of the image plane in **Left**, **Right**, **Bottom**, **Top** parameters; distance of the image plane to the camera in **Near** and distance of the far plane to the camera in **Far** parameters, and the resolution of the final image in **HorRes** and **VerRes** parameters. All values are floats except **HorRes** and **VerRes**, which are integers.
5. **OutputName** is a string which is the name of the output image.

4 Scene File

- **Background Color** R G B
- **Backface Culling Setting** {0: disabled **(default)**, 1: enabled}
- **#Vertices**
- **Number of vertices** Integer
- #Colors
  - **Color of vertex #i R G B**
- #Positions
  - **Position of vertex #i X Y Z**
- **#Translations**
- **Translation Count** Integer
  - t_x t_y t_z
- **#Scalings**
- **Scaling Count** Integer
  - s_x s_y s_z
- **#Rotations**
- **Rotation Count** Integer
  - α u_x u_y u_z
- **#Models**
- **Number of models** Integer
- **Model ID** Integer
  - **Model Type** {0: wireframe, 1: solid}
  - **Number of transformations** Integer
    - **Transformation_type Transformation_id**
  - **Number of triangles** Integer
    - **Vertex index$_1$ Vertex index$_2$ Vertex index$_3$**

1. Background Color
Specifies the R, G, B values of the background.

2. Backface Culling Setting
Specifies whether culling is applied or not. **If it is 0, there will be no culling**, just draw all triangles. **If it is 1, culling should be done when triangle's backface is facing to the viewer.**

3. Number of vertices
Specifies how many vertices are in the scene.

4. Colors
Specifies the color of each vertex in R G B starting with the vertex id 1.

5. Positions
Specifies the position of each vertex in X Y Z starting with the vertex id 1.

6. Translation Count
Specifies how many translations are defined in the scene file.

7. Translation parameters
t_x, t_y, and t_z are the translation parameters, i.e., translation amounts along the major axes.

8. Scaling Count
Specifies how many scale transformations are defined in the scene file.

9. Scaling parameters
s_x, s_y, and s_z are the scaling parameters, i.e., scaling level in the corresponding coordinate axes.

10. Rotation Count
Specifies how many rotations are defined in the scene file.

11. Rotation parameters
α, u_x, u_y, and u_z are the rotation parameters, i.e., the object is rotated α degrees around the rotation axis which pass through points (u_x, u_y, u_z) and $(0, 0, 0)$. The positive angle of rotation is given as the counter-clockwise rotation along the direction (u_x, u_y, u_z).

12. Number of models
Specifies how many models (i.e., sets of vertices) are in the scene file.

13. Model ID

1. Model type of the model. **0 for wireframe, 1 for solid.**
2. Number of transformations.
3. The 't' indicates a translation transformation, 's' indicates a scale transformation, and 'r'

indicates a rotation transformation. The transformation id ranges from [1 ... Number of Translations] for type 't' transformations, and similarly for the other type of transformations.

4. Number of triangles.

5. Each triangle is given as a list of vertex ids in counter clockwise order when faced from the front side. Vertex ids start from 1.

5 Sample input/output

A sample camera and a scene file are provided below:
sample_camera.txt:

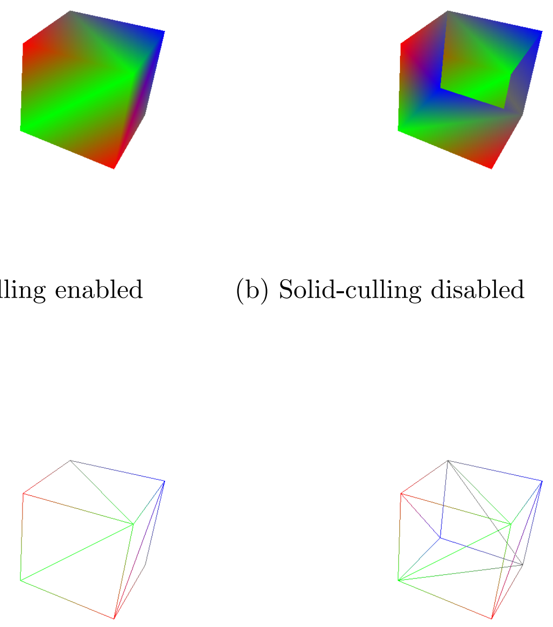
```
1
#Camera 1
0 5 0
0.1 -0.3 -0.5
0 1 0
-1 1 -1 1 2 1000 700 700
sample.ppm
```

sample_scene.txt:

```
255 255 255
1  <-----  backface culling is enabled
#Vertices
8
#Colors
100 100 100
255 0 0
0 255 0
0 0 255
0 0 255
0 255 0
255 0 0
100 100 100
#Positions
1.0 1.0 -1.0
-1.0 1.0 -1.0
-1.0 1.0 1.0
1.0 1.0 1.0
1.0 -1.0 -1.0
-1.0 -1.0 -1.0
-1.0 -1.0 1.0
1.0 -1.0 1.0
#Translations
2
0.0 10.0 0.0
```

```
3.0 -3.0 -6.0
#Scalings
1
5.2 5.2 5.2
#Rotations
3
45 0.0 1.0 0.0
60 0.8 0.6 0.0
20 1.0 0.0 0.0
#Models
1
1
1 <----- model is drawn in solid mode
3
r 1
t 2
s 1
12
7 8 4
7 4 3
8 5 1
8 1 4
6 3 2
6 7 3
3 4 1
3 1 2
6 2 5
2 1 5
5 8 6
7 6 8
```

You can see outputs of this example in the next page.



(a) Solid-culling enabled (b) Solid-culling disabled

(c) Wireframe-culling enabled (d) Wireframe-culling disabled

Figure 1: Same model, different modes of rendering

6 Hints & Tips

1. Start early!

2. Note that in the wireframe mode only the edges of the triangle will be drawn. For example, if there are two front facing triangles with the small triangle behind the large triangle, the small triangle will be visible in the wireframe mode, but will not be visible in the solid mode.

3. A makefile will be provided to you. You can use that file to compile your codes.
   For C:

   ```
   >_ make rasterizer_c
   >_ ./rasterizer <scene_file_name> <camera_file_name>
   ```

   For C++:

   ```
   >_ make rasterizer_cpp
   >_ ./rasterizer <scene_file_name> <camera_file_name>
   ```

If you plan to compile the rasterizer by yourself, make sure it is compiled and run successfully by running following commands:
For C:

```
>_ gcc -o rasterizer hw2_math_ops.c hw2_file_ops.c rasterizer.c -lm
>_ ./rasterizer <scene_file_name> <camera_file_name>
```

For C++:

```
>_ g++ -o rasterizer hw2_math_ops.cpp hw2_file_ops.cpp rasterizer.cpp
>_ ./rasterizer <scene_file_name> <camera_file_name>
```

4. For debugging purposes, consider using simple scenes. Also it may be necessary to debug your code by tracing what happens for a single triangle (always simplify the problem when debugging).

5. You can reach different inputs and desired images in “inputs” folder. You will see that different camera angles, output of different rendering modes are provided to you to understand 3d scene and rendering modes completely.

6. You can also find scripts that are used for drawing flags. Scripts and explanations are provided in case you want to play with them.

7 Regulations

1. **Programming Language:** C/C++

2. **Late Submission:** Late submission is not allowed.

3. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0. You can discuss algorithmic choices, but sharing code between each other or using third party code is strictly forbidden. To prevent cheating in this homework, we also compare your codes with online ray tracers and previous years' student solutions. In case a match is found, this will also be considered as cheating. Even if you take a “part” of the code from somewhere/somebody else - this is also cheating. Please be aware that there are “very advanced tools” that detect if two codes are similar. So please do not think you can get away with by changing a code obtained from another source.

4. **Announcements:** You must follow the ODTU-Class announcements for possible updates on the assignment.

5. **Submission:** Submission will be done via ODTU-Class. Create a .zip file named **“rasterizer.zip”** that contains your **“rasterizer.c”/“rasterizer.cpp” code ONLY**. The .zip file should not include any subdirectories. **Indicate group members' student ids as comment at the beginning of your source file.**

6. **Evaluation:** Your codes will be evaluated based on several input files including, but not limited to the test cases given to you as example. Rendering all scenes correctly will get you 100 points.

7. **Last note:** Your output format will be .ppm file, that is fixed. You will naturally want to view your outputs after running. You may face issues when you want to view your output file. You can open .ppm files in Ubuntu; however, you can not open .ppm files in Windows directly. You need a converter to do that. **ImageMagick** is a tool for displaying various image formats. It can also do conversion between formats. See the steps below:

   - If you are using **Ubuntu** and running command **"convert -version"** on terminal gives version information, ImageMagick is installed on your system.
   - If you are using **Windows** and running command **"magick convert -version"** on terminal gives version information, ImageMagick is installed on your system.
   - Otherwise, you need to install it from **here**. After installation, use following lines according to your operating system:

     Default function call, doesn't do conversion, no harm:

     ```
     convertPPMToPNG(cameras[i].outputFileName, 99);
     ```

     For conversion on Ubuntu:

     ```
     convertPPMToPNG(cameras[i].outputFileName, 1);
     ```

     For conversion on Windows:

     ```
     convertPPMToPNG(cameras[i].outputFileName, 2);
     ```

   Now you can view your outputs on Ubuntu and Windows, in PNG format.