Middle East Technical University | Department of Computer Engineering

CENG 477

Introduction to Computer Graphics

Fall '2019-2020
Assignment 2 - A Software Rasterizer implementing the Forward Rendering Pipeline

Due date: December 8th, 2019, Sunday, 23:59

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 similar to the first assignment. 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 implement a clipping algorithm of your choice **(Cohen-Sutherland or Liang-Barsky)** for wireframe mode only. You will also implement the backface culling (for both modes) for correct rendering of the scene. You will write your implementations in C++ language.

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

2 Specifications

1. Name of the executable will be "rasterizer".
2. Your executable will take an input file name from command line. 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. You will implement two types of projection transformations: **orthographic projection and perspective projection**.

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

7. 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).**

8. **You will also implement a clipping algorithm of your choice.** For example, you can choose an algorithm among algorithms that you saw on the lectures. You will do clipping for wireframe mode only.

9. There can be multiple camera configurations for each given scene.

10. **Here is the good news. You will NOT deal with input files.** Helper functions for reading inputs, helper functions for mathematical operations (e.g. normalization, matrix multiplication) are given to you in **"Scene.cpp"** file. It is **STRONGLY** recommended to inspect types in header files and helper functions in "Scene.cpp" file.

11. Since you will not deal with input reading, memory allocation for the image and implementing "main()" function, **you will only implement "forwardRenderingPipeline()" member function of the Scene class.** You can also write helper functions by yourself. **If you decide to implement additional functions, add declarations to "Scene.h" and implementations to "Scene.cpp" file.**

12. **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 Input File

You will not deal with reading input files, however, it is good for you to know what input file contains for testing purposes. Input files are in XML format and their structure is given below:

```
<Scene>
  <BackgroundColor>R G B</BackgroundColor>
  <Culling> 0 or 1 </Culling>
  <ProjectionType> 0 or 1 </ProjectionType>
  <Cameras>
    <Camera id="#">
      <Position>X Y Z</Position>
      <Gaze>X Y Z</Gaze>
      <Up>X Y Z</Up>
      <ImagePlane>
          Left Right Bottom Top Near Far HorRes VerRes
      </ImagePlane>
      <OutputName> <image_name>.ppm </OutputName>
    </Camera>
    ... more cameras ...
  </Cameras>

  <Vertices count="#">
    <Vertex id="1" position="X Y Z" color="R G B" />
    ... more vertices ...
  </Vertices>

  <Translations count="#">
    <Translation id="1" value="tx ty tz" />
    ... more translations ...
  </Translations>

  <Scalings count="#">
    <Scaling id="1" value="sx sy sz" />
    ... more scalings ...
  </Scalings>

  <Rotations count="#">
    <Rotation id="1" value="angle ux uy uz" />
    ... more rotations ...
  </Rotations>

  <Models count="#">

    <Model id="#" type="0 or 1">
      <Transformations count="#">
        <Transformation> <transformation_type> <transformation_id> </Transformation>
```

```
      ... more transformations ...
    </Transformations>

    <Triangles count="#">
      <Triangle> <vertex_id_1> <vertex_id_2> <vertex_id_3>  </Triangle>
      ... more triangles ...
    </Triangles>

  </Model>

  ... more models ...
 </Models>
</Scene>
```

3.1 Input File Sections

3.1.1 Background color:

Specifies the R, G, B values of the background.

3.1.2 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.**

3.1.3 ProjectionType Setting:

Specifies which projection transformation is applied. **If it is 0, that means you need to apply orthographic projection**, **If it is 1, you should apply perspective projection.**

3.1.4 Cameras:

- **Id** specifies the camera id.
- **Position** specifies the X, Y, Z coordinates of the camera.
- **Gaze** specifies the direction that the camera is looking at.
- **Up** specifies 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.
- **ImagePlane** specifies 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.
- **OutputName** is a string which is the name of the output image.

3.1.5 Vertices:

- **Count** specifies how many vertices are in the scene.
- **Id** specifies the vertex id.
- **Position** specifies the position of the vertex in X Y Z.
- **Color** specifies the color of each vertex in R G B.

3.1.6 Translations:

- **Count** specifies how many translation types are in the scene.
- **Id** specifies the translation id.
- **Value** specifies t_x, t_y, and t_z, which are the translation parameters, i.e., translation amounts along the major axes.

3.1.7 Scalings:

- **Count** specifies how many scaling types are in the scene.
- **Id** specifies the scaling id.
- **Value** specifies s_x, s_y, and s_z, which are the scaling parameters, i.e., scaling level in the corresponding coordinate axes.

3.1.8 Rotations:

- **Count** specifies how many rotation types are in the scene.
- **Id** specifies the rotation id.
- **Value** specifies α, u_x, u_y, and u_z, which 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).

3.1.9 Models:

- **Count** specifies how many models are in the scene.
- **Id** specifies the model id.
- **Type** specifies the type of the model. It is either 0 or 1. **0 for wireframe, 1 for solid rendering.**
- **Transformation count** specifies how many transformations are applied to this model. Each transformation has transformation type and transformation id parameters. The 't' indicates a translation transformation, 's' indicates a scale transformation, and 'r' indicates a rotation transformation. The transformation id ranges from [1 ... translation count] for type 't' transformations, and similarly for the other type of transformations.

- **Triangle count** specifies how many triangles are there for this model. 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.

4 Sample input/output

A sample camera and a scene file are provided below:

```
<Scene>
        <BackgroundColor>255 255 255</BackgroundColor>
        <Culling>1</Culling>     <------- backface culling is enabled
        <ProjectionType>1</ProjectionType>     <------- perspective projection
        <Cameras>
                <Camera id="1">
                        <Position>-10 -40 -26</Position>
                        <Gaze>1 1 -0.2</Gaze>
                        <Up>0 1 0</Up>
                        <ImagePlane>-1 1 -1 1 2 1000 700 700</ImagePlane>
                        <OutputName>filled_box_1.ppm</OutputName>
                </Camera>
                <Camera id="2">
                        <Position>-20 -28 -23</Position>
                        <Gaze>1 0.5 -0.2</Gaze>
                        <Up>0 1 0</Up>
                        <ImagePlane>-1 1 -1 1 2 1000 700 700</ImagePlane>
                        <OutputName>filled_box_2.ppm</OutputName>
                </Camera>
        </Cameras>

        <Vertices count="8">
                <Vertex id="1" position="1.0 1.0 -1.0" color="100 100 100" />
                <Vertex id="2" position="-1.0 1.0 -1.0" color="255 0 0" />
                <Vertex id="3" position="-1.0 1.0 1.0" color="0 255 0" />
                <Vertex id="4" position="1.0 1.0 1.0" color="0 0 255" />
                <Vertex id="5" position="1.0 -1.0 -1.0" color="0 0 255" />
                <Vertex id="6" position="-1.0 -1.0 -1.0" color="0 255 0" />
                <Vertex id="7" position="-1.0 -1.0 1.0" color="255 0 0" />
                <Vertex id="8" position="1.0 -1.0 1.0" color="100 100 100" />
        </Vertices>

        <Translations count="2">
                <Translation id="1" value="0.0 10.0 0.0" />
                <Translation id="2" value="3.0 -3.0 -6.0" />
        </Translations>

        <Scalings count="1">
```

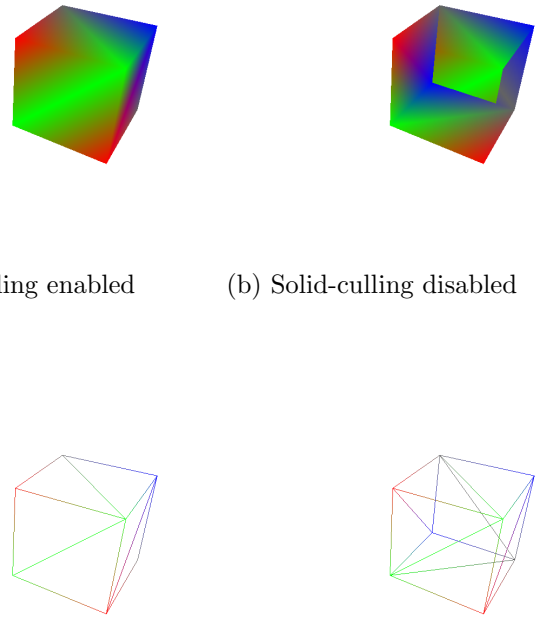
```
        <Scaling id="1" value="5.2 5.2 5.2" />
    </Scalings>

    <Rotations count="3">
        <Rotation id="1" value="45 0.0 1.0 0.0" />
        <Rotation id="2" value="60 0.8 0.6 0.0" />
        <Rotation id="3" value="20 1.0 0.0 0.0" />
    </Rotations>

    <Models count="1">

        <Model id="1" type="1">   <------- model is drawn in solid mode
            <Transformations count="3">
                <Transformation>r 1</Transformation>
                <Transformation>t 2</Transformation>
                <Transformation>s 1</Transformation>
            </Transformations>
            <Triangles count="12">
                <Triangle>7 8 4</Triangle>
                <Triangle>7 4 3</Triangle>
                <Triangle>8 5 1</Triangle>
                <Triangle>8 1 4</Triangle>
                <Triangle>6 3 2</Triangle>
                <Triangle>6 7 3</Triangle>
                <Triangle>3 4 1</Triangle>
                <Triangle>3 1 2</Triangle>
                <Triangle>6 2 5</Triangle>
                <Triangle>2 1 5</Triangle>
                <Triangle>5 8 6</Triangle>
                <Triangle>7 6 8</Triangle>
            </Triangles>
        </Model>
    </Models>
</Scene>
```

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

5 Hints & Tips

1. Start early!
2. Note that in the **wireframe** mode **only the edges of the triangle will be drawn**.
3. 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).
4. You can reach different inputs in **"culling_disabled_inputs"** and **"culling_enabled_inputs"** folders and desired images in **"culling_disabled_outputs"** and **"culling_enabled_outputs"** folders. 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 Regulations

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

2. **Late Submission:** You can submit your codes up to 3 days late. Each late day will be deduced from the total 7 credits for the semester. However, if you fail to submit even after 3 days, you will get 0 regardless of how many late credits you may have left. If you submit late and still get zero, you cannot claim back your late days. You must e-mail your assistants if you want your submission to not be evaluated (and therefore preserve your late day credits).

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. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

5. **Submission:** Submission will be done via OdtuClass. You can team-up with another student. Create a .zip file that contains your **source files and Makefile**. The .zip file should not include any subdirectories. **The .zip file name will be:**

   - If a student works with a partner student:

     `<partner_1_student_id>_<partner_2_student_id>_rasterizer.zip`

   - If a student works alone:

     `<student_id>_rasterizer.zip`

   - For example:

     ```
     e1234567_e2345678_rasterizer.zip
     e1234567_rasterizer.zip
     ```

   Make sure that when below command is executed, your executable file is ready to use:

   ```
   >_ make rasterizer_cpp
   >_ ./rasterizer <input_file_name>
   ```

   Therefore you HAVE TO provide a Makefile in your submissions.

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 working with our department's inek machines, you don't need additional installation. **ImageMagick is installed on ineks.**
   - If you are using **Ubuntu** and running command **"convert -version"** on terminal gives version information, that means ImageMagick is installed on your system.
   - If you are using **Windows** and running command **"magick convert -version"** on terminal gives version information, that means 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(scene->cameras[i]->outputFileName, 99);
     ```

     For conversion on Ubuntu:

     ```
     convertPPMToPNG(scene->cameras[i]->outputFileName, 1);
     ```

     For conversion on Windows:

     ```
     convertPPMToPNG(scene->cameras[i]->outputFileName, 2);
     ```

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