Assignment 2 – Venerba Mirco 872653 – Zannini Giosuè 873810

- **Design decisions:**
  - We have also entered the screen coordinates in the vertex object because maybe they are useful in the next task otherwise we will remove them in the next assignments.
  - We did not calculate the barycentric coordinates because we used other methods for the test of inclusion of points in the triangle and for the calculation of the depth coordinates. We have explained these techniques later in this documentation. In case they are useful in future assignments, we will create the related functions.
  - After calculing vertices triangles in normalized and screen coordinates, we order them using the screen coordinates in clockwise otherwise the technique of edge function to check if a point is into or not in the triangle doesn't work.
  - We add also the methods to manage the textures and other proprieties but we didn't interpolate them because it isn't necessary in this assignment, in fact the shader returns only the first decimal number of the depth, in case we will add the interpolation in the next assignment.
  - We didn't store the details of each point inside the triangles because it is not necessary and in case we will implement it in the future assignments.
  - For the problem "Primitives outside the range are dropped", in our project is very very simple its management in fact we used two nested for loop to elaborate only the pixel that are into the screen dimensions.

```
/*for each point in the matrix*/
for (int i = 0; i < HEIGHT; ++i) {
    for (int j = 0; j < WIDTH; ++j) {
```

  - All data types in our project are double (even coordinates) to maintain a high level of accuracy because we don't know how the project will evolve in future assignments, in case we change the data type of the coordinates to int.

```
/*i added also the screen coordinates variables to maintain a complete schema for the vertex,
 * in case i can eliminate them in the future assignments*/
double x, y, z, normalX, normalY, normalZ, screenX, screenY, screenZ, u, v;
```

  - Unlike the first assignment we have to implement the static shader, so for this the return type of the function is char and we deleted the polymorphism structure and the generic type "Target_t" from the shader.

```
class Shader {
public:
    char firstDecimalZbuffer(double val) {
        val -= static_cast<double>(static_cast<int>(val));
        return static_cast<int>(val * 10) + ASCII_CODE_0;
    }
};
```

- We used a specific file for all the constants and we decided on this strategy to make the project more dynamic, parametric and simpler variable. In fact if we would change the screen size we just change this file and the whole project changes automatically.

```
/*other details*/
const int NUMBER_PROJECTION_ARRAY = 6;
const int NUMBER_VERTICES_TRIANGLE = 3;
const int NUMBER_DIFFERENT_MEASURES = 3;
const int NUMBER_PROJECTION_MATRIX = 4;
const int NUMBER_NORMALIZED_MEASURES = 4;
const char PRINTED_CHARACTER_INVALID_CELL = '.';
const std::string SPACE_OUTPUT = "          ";
const int ASCII_CODE_0 = 48;
const int NUMBER_CHARACTERISTICS_VERTEX_TRIANGLE = 3;

/*file parameter*/
const std::string FILE_SOURCE = "cubeMod.obj";

/*parameter far and near parametric*/
const int FAR = 2;
const int NEAR = 1;

/*define the projection array as left, top, right, bottom, near,  far*/
const double projectionArray[NUMBER_PROJECTION_ARRAY] = {-1, -1, 1, 1, 1, 2};
/*define the view matrix*/
const double viewMatrix[NUMBER_PROJECTION_MATRIX * NUMBER_PROJECTION_MATRIX] = {0.5, 0.0, 0.0, 0.7,
                                                                                0.0, 0.5, 0.0, 0.7,
                                                                                0.0, 0.0, 0.5, 0.9,
                                                                                0.0, 0.0, 0.0, 1.0};

/*order triangles*/
const int FLAT_CORNER = 180;
const double PI = 3.14159;
const int FULL_CIRCLE = 360;
```

- We initialised the opportune variables and objects as in Moodle page and as in the email received from the prof to replicate the same example.
- We used many files in the project:
    - Main.cpp: this file is used to create the complete logic execution plan, in fact in the first part there are all variable definitions and initializations and in the last part there is the definition of the operation object (rasterizer) and the many calls to its functions.
    - OperationRasterizer.h: this file is used to define the object class that contains all required functions to solve this assignments and in fact the class used inside represents the rasterizer object, in fact from the main file we call this functions.
    - Triangle.h: this file is used to define the object class that contains the methods to create the triangle object with an array of three vertex class objects representing the three real vertices.

- Vertex.h: this file is used to define the object class that contains the methods to represent the vertex object and this object has many fields as input coordinates, normalized coordinates, u and v parameter and the last are screen coordinates that we added because perhaps they are useful in the next assignment otherwise we will remove them.
- ValueProperties.h: this file is used to mantain organized all costants and their value in a single file to make the project more dynamic, parametric and simpler variable.
- Shader.h: this file is used to define the object class that contains the method to create the static shader class to get the first decimal number of all depth of each point inside all triangles.
- ReadFile.h: this file is used to create all usefull functions to read and verify the correctness of the input file that rappresents the object, composed of only many triangles, that we want to draw in the screen

```
INPUT FILE:
v)  0.5 -0.5 1
v)  0.5 -0.5 2
v)  -0.5 -0.5 2
v)  -0.5 -0.5 1
v)  0.5 0.5 1
v)  0.499999 0.5 2
v)  -0.5 0.5 2
v)  -0.5 0.5 1
vt) 1 0.333333
vt) 1 0.666667
vt) 0.666667 0.666667
vt) 0.666667 0.333333
vt) 0.666667 0
vt) 0 0.333333
vt) 0 0
vt) 0.333333 0
vt) 0.333333 1
vt) 0 1
vt) 0 0.666667
vt) 0.333333 0.333333
vt) 0.333333 0.666667
vt) 1 0
vn) 0 -1 0
vn) 0 1 0
vn) 1 0 0
vn) -0 0 1
vn) -1 -0 -0
vn) 0 0 -1
f)  2/1/1 3/2/1 4/3/1
f)  8/1/2 7/4/2 6/5/2
f)  5/6/3 6/7/3 2/8/3
f)  6/8/4 7/5/4 3/4/4
f)  3/9/5 7/10/5 8/11/5
f)  1/12/6 4/13/6 8/11/6
f)  1/4/1 2/1/1 4/3/1
f)  5/14/2 8/1/2 6/5/2
f)  1/12/3 5/6/3 2/8/3
f)  2/12/4 6/8/4 3/4/4
f)  4/13/5 3/9/5 8/11/5
f)  5/6/6 1/12/6 8/11/6
```

- cubeMod.obj: this file is used to describe, using the coordinates, normal coordinates, texture coordinates and their mapping function, the whole object that we want to draw in the screen space.

- The method that we used to check if a point is into the triangle is the edge function, a simple method to know if a point (x,y) is below or above of the segment created by the two input parameter vertices checking that the result is >=0 and then recall it also in the other edges.

```
/*function used to create the edge function, a simple method to know if a point (x,y)
 * is below or above of the segment created by the two input parameter vertices */
inline double edgeFunction(Vertex vertex1, Vertex vertex2, double x, double y) {
    return (vertex2.getScreenCoordinateX() - vertex1.getScreenCoordinateX()) *
           (y - vertex1.getScreenCoordinateY()) -
           (vertex2.getScreenCoordinateY() - vertex1.getScreenCoordinateY()) *
           (x - vertex1.getScreenCoordinateX());
}
```

- The method that we used to calculate the depth for each point is using the plan equation, in fact we can think a triangle as a plan passing between the three vertices. After find the plan equation we can substitute the x and y coordinates of the actual point to find the remaining z coordinate. In fact in the next image we can observe that in the input parameters there are the three vertices coordinates and the two coordinates of the actual point and after elaborate the complex instruction this function return the remaining z coordinate.

```
/*function used to calculate the plan equation between three input point and it returns the z-coordinate of another
 * point(x,y,unknown) , into this plan, that are known only the x and y coordinates */
inline double planEquationTriangle(double x0, double y0, double z0, double x1, double y1, double z1, double x2, double y2,
                                   double z2, double x, double y) {
    /*in this function parameters i pass the double coordinates because if i pass the complete vertex (more nice),
     * i must call the getter method of all coordinates and the formula becomes illegible*/
    return (((x - x0) * (y1 - y0) * (z2 - z0) + (y - y0) * (z1 - z0) * (x2 - x0) -
            (y - y0) * (x1 - x0) * (z2 - z0) -
            (x - x0) * (z1 - z0) * (y2 - y0)) / ((-1) * (x1 - x0) * (y2 - y0) + (y1 - y0) * (x2 - x0))) + z0;
}
```

- We used the character '.' for the empty cells as in Moodle page but we can modify it using the PRINTED_CHARACTER_INVALID_CELL constant in the ValueProperties.h file.

- **Usage of the pipeline:**

  - The first thing that the person, that execute the program, must do is change the path/name of the file .obj in the valuesProprierties file.

  - The first thing is create and initialize all necessary variables and objects as screen matrix, zbuffer, projection matrix, projection array, view matrix, shader, the read file methods to read and compute the content of the input file .obj.

  - Then we created the Operation class object with Target_t=char, also setting their fields passing the references of screen matrix, zbuffer, projection matrix.

  - Then we decided to not create the read file object because the methods doesn't rappresent a real object but different usefull functions to read and verify the obj input file.

- Then using the readfile methods we create a dynamic vector of triangles using data inside the .obj input file and in each for loop we use the size method of the vector class to get the correct number of triangles that are describe in the obj input file. If the file isn't in the correct format or the number of parameters are wrong the program print in output the error description and it will stop the execution.

```
ERRORE: Numbero parametri errato nelle texture del vertice
```

- Then we compute/transform the input coordinates and normal cooordinates using the view matrix that the prof sent by email.
- Then for each triangle calculate the normalized vertex coordinates, and the relative screen vertex coordinates.
- Then for each triangle and for each point in the screen from (0,0) to (49,149) (thinking screen as in the Moodle example 50x150) check it using the edge function if it is into the triangle and in case calculate the depth using the plan equation and finally compare it with the zbuffer and in case it is smaller update zbuffer so to make visible only the near objects.
- The last thing is to use the static shader to calculate the first decimal of the depth to save it in the screen matrix for the final step that it is the screen print function.

- **Example of execution**
  - To compile the whole project: g++ *.cpp *.h -o def
  - To execute the project: ./def
  - We can observe an example of the execution in the "Esempio di output.txt"