Computer Graphics, Task 5 - Guidelines

Paweł Aszklar
P.Aszklar@mini.pw.edu.pl

Warsaw, May 31, 2016

1 Introduction

This document contains guidelines for Task 5 which consists of displaying a representation of a three dimensional scene in the program window. For drawing, only library functions for coloring individual pixels and drawing line segments can be used. All other functionality, described later in this document, including scene modeling, projecting those models onto the program window, and calculating colors of the resulting pixels needs to be implemented by yourself. In particular, you should not use such functionality build into 3D rendering libraries such as Direct3D and OpenGL.

2 Scene Modeling

Scene presented by the program will consist of several types of abstract objects. They need to be modeled in a way that will simplify creation of the final image. The scene may include following types of objects:

- **Models** — representing solids displayed on the screen. Each of them will be approximated as one or more triangle mesh positioned within a 3D space.
- **Light sources** — if present, they provide lighting for the scene. They are not drawn themselves, but may affect the colors of pixels on the surface of an object drawn on the screen.
- **Camera** — defines the position and orientation of the point of view from which the scene is being observed.

Creation of models for the solids, that are present in the scene, will often require the use of points and vectors describing positions and directions in a 3D space. Those points and vectors will often need to be expressed in and converted between different coordinate systems. It is then vitally important to always remember which coordinate system we are currently working in.

For the purpose of this task, we will assume that all coordinate systems used are right-handed (e.g. for the coordinate system of the camera, the X axis points to the right, the Y axis points upwards, and the Z axis points towards us, i.e. away from the screen — for reference in a left-handed coordinate system the Z axis would point into the screen instead).

2.1 Meshes

In this task drawn objects consist of one or more geometrical solids. For each of the solids we can find a triangular mesh approximating its surface.

A triangular mesh is a set of flat triangles positioned in 3D space. Each triangle is defined by its three vertices. Each vertex has its position and can have some additional attributes, such as a vector normal to the surface at that point, texture coordinates (discussed in a separate document), etc. In particular the normal vector is needed to calculate the color of a point on a surface depending on the lighting (see. section 5). Bearing in mind the representation of transformations described in section 3, when creating the mesh, it is better to already store points $\mathbf{p}$ and normal vectors $\mathbf{n}$ with four coordinates, as follows:

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}, \mathbf{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$$

It is worth considering, that even though some vertices in two neighboring triangles in the mesh, might share the same position, their respective normal vectors might be different (e.g. for a corner of a cuboid, the normal vector depends on the face which the vertex is a part of). In those cases it might be beneficial to store vertices as tuples $(\mathbf{p}, \mathbf{n})$ despite the increased memory cost of storing duplicated positions.

When modeling, the positions and vectors will be expressed in a so called *local* coordinate system in such a way that the definition of the mesh is as simple as possible.

Additionally, when defining a triangle as a set of three vertices, it might be useful to order them correctly. Since the meshes will be closed, each triangle will have the *visible* side facing the outside of the solid. If we make sure that for each triangle, if we look at it from its visible side, the order of vertices on the screen is clockwise, we can implement a simple optimization algorithm, that will ignore triangles facing away from the screen (as described in section 4.2).

For each modeled solid, besides the triangle mesh, we might need to define its material coefficients, needed to calculate surface color based on lighting. Material coefficients are described in section 5.2.

Last part of the definition of a model is a transformation from the local coordinate system to the coordinate system of the scene (sometimes referred to as *global* coordinate system). This defines position and orientation of the object in the scene. Transformations between coordinate systems are described in section 3.

Following subsections describe algorithms used to generate vertices and meshes for basic geometrical solids.

2.1.1 Cuboid mesh

Cuboid mesh will be defined in a coordinate system, whose origin is in the bottom, back, left corner, and whose axes are parallel to the edges of the solid. Width w of the cuboid will describe the length of the edge parallel to the X axis, height h will be the length of edge parallel to Y axis, and depth d will be the length of edge parallel to Z axis. As a remainder, the coordinate system is right-handed, so the greater z coordinate describes points "closer to us".

Cuboid has 8 corners, however, each face has a different normal vector. Hence we will model each face independently using 4 vertices and 2 triangles. The entire mesh will then consist of 24 vertices and 12 triangles.

Let us first consider the front face, which consists of vertices:

$$\mathrm{V}_i = (\mathbf{p}_i, \mathbf{n}_i)\,,\ i = 1, \ldots, 4$$

For a given face all normal vectors are the same and in case of the front face they are equal to:

$$\mathbf{n}_i = [0, 0, 1, 0]^T\,,\ i = 1, \ldots, 4$$

Vertex positions are defined as:

$$\begin{aligned}\mathbf{p}_1 &= [0, 0, d, 1]\\ \mathbf{p}_2 &= [w, 0, d, 1]\\ \mathbf{p}_3 &= [w, h, d, 1]\\ \mathbf{p}_4 &= [0, h, d, 1]\end{aligned}$$

Front face itself consists of triangles:

$$\mathrm{T}_1 = (\mathrm{V}_1, \mathrm{V}_2, \mathrm{V}_3)\,, \mathrm{T}_2 = (\mathrm{V}_1, \mathrm{V}_3, \mathrm{V}_4)$$

The remaining five faces can be defined analogously. As noted previously it is important to consider the proper ordering of vertices in each triangle.

2.1.2 Cylinder mesh

Cylinder mesh will be defined in a coordinate system whose origin is in the center of bottom base, whose X and Z axes define a plane containing the bottom base, and whose Y axis points towards the top base. Let h be

the height of the cylinder and r be the radius of bases. The surface of a cylinder can not be represented exactly as a set of flat triangles. Hence we will approximate its shape using a right regular n-sided prism.

Due to the difference in normal vectors, we will consider bases independently of the side faces. Each base will be modeled as an n-sided regular polygon consisting of $(n+1)$ vertices (one for each corner and one for the center) and n triangles. Cylinder side will be approximated using $2n$ vertices and $2n$ triangles. In total the mesh will contain $4n+2$ vertices and $4n$ triangles.

For all vertices of the top base:

$$\mathrm{V}_i = (\mathbf{p}_i, \mathbf{n}_i)\,,\ i = 0, \ldots, n$$

normal vectors are equal to:

$$\mathbf{n}_i = [0, 1, 0, 0]^T\,,\ i = 0, \ldots, n$$

Position of the vertex in the center of top base is:

$$\mathbf{p}_0 = [0, h, 0, 1]^T$$

and positions of the vertices on the sides of top base are:

$$\mathbf{p}_i = \left[r\cos\left(\frac{2\pi}{n}\,(i-1)\right), h, r\sin\left(\frac{2\pi}{n}\,(i-1)\right), 1\right],\ i = 1, \ldots, n$$

Finally the top base is composed of triangles:

$$\mathrm{T}_i = \left(\mathrm{V}_0, \mathrm{V}_{(i+2)\bmod(n+1)}, \mathrm{V}_{i+1}\right),\ i = 0, \ldots, n-1$$

Similarly for the bottom base:

$$\mathrm{V}_i = (\mathbf{p}_i, \mathbf{n}_i)\,,\ i = 3n+1, \ldots, 4n+1$$

$$\mathbf{n}_i = [0, -1, 0, 0]^T\,,\ i = 3n+1, \ldots, 4n+1$$

$$\mathbf{p}_{4n+1} = [0, 0, 0, 1]^T$$

$$\mathbf{p}_i = \left[r\cos\left(\frac{2\pi}{n}\,(i-1)\right), 0, r\sin\left(\frac{2\pi}{n}\,(i-1)\right), 1\right],\ i = 3n+1, \ldots, 4n$$

$$\mathrm{T}_i = (\mathrm{V}_{4n+1}, \mathrm{V}_{i+1}, \mathrm{V}_{i+2})\,,\ i = 3n, \ldots, 4n-2$$

$$\mathrm{T}_{4n-1} = (\mathrm{V}_{4n+1}, \mathrm{V}_{4n}, \mathrm{V}_{3n+1})$$

Cylinder side will be approximated by the side faces of the prism, however, normal vectors will be calculated based on the shape of the original surface. Therefore, vertices with the same position on the side of the cylinder will also share normal vectors, removing the need for duplicated vertices.

Since we don't have to consider each side face of the prism separately, that part of the mesh can be defined by vertices:

$$\mathrm{W}_i = (\mathbf{p}_i, \mathbf{n}_i)\,,\ i = n+1, \ldots, 3n$$

where:

$$\mathbf{p}_i = \begin{cases} \mathbf{p}_{i-n} & i = n+1, \ldots, 2n \\ \mathbf{p}_{i+n} & i = 2n+1, \ldots, 3n \end{cases}$$

$$\mathbf{n}_i = \left[\frac{p_{i,x}}{r}, 0, \frac{p_{i,z}}{r}, 0\right]^T\,,\ i = n+1, \ldots, 3n$$

The side surface consists of triangles:

$$\mathrm{T}_i = \begin{cases} (\mathrm{V}_{i+1}, \mathrm{V}_{i+2}, \mathrm{V}_{i+1+n}) & i = n, \ldots, 2n-2 \\ (\mathrm{V}_{2n}, \mathrm{V}_{n+1}, \mathrm{V}_{3n}) & i = 2n-1 \\ (\mathrm{V}_{i+1}, \mathrm{V}_{i+2-n}, \mathrm{V}_{i+2},) & i = 2n, \ldots, 3n-2 \\ (\mathrm{V}_{3n}, \mathrm{V}_{n+1}, \mathrm{V}_{2n+1}) & i = 3n-1 \end{cases}$$

2.1.3 Sphere mesh

Sphere mesh will be defined in a coordinate system whose origin is in the center of the sphere. The sphere will be approximated by subdividing it akin to how meridians and parallels subdivide the globe. Using this analogy, the X and Z axes will define a plane containing the equator, and the Y axis will point to the north pole. The vertices of the mesh will be placed where meridians and parallels intersect. Let r be the radius of the sphere, m the number of meridians and n the number of parallels in the subdivision.

Similarly to how we defined a mesh for the side surface of a cylinder, a sphere can not be represented exactly using flat triangles so the resulting mesh will be just an approximation. However, the normal vectors in each vertex is defined in relation to the original surface, so vertices with the same position will share normal vectors, removing the need for vertices with duplicated position. As a result sphere mesh will consist of $mn + 2$ vertices (intersections of m meridians and n parallels plus two poles) and $2mn$ triangles.

Each vertex of the mesh:

$$\mathrm{V}_i = (\mathbf{p}_i, \mathbf{n}_i)\,,\ i = 0, \ldots, mn+1$$

has a normal vector equal to:

$$\mathbf{n}_i = \mathbf{p}_i / r = \left[\frac{p_{i,x}}{r}, \frac{p_{i,y}}{r}, \frac{p_{i,z}}{r}, 0\right]^T\,,\ i = 0, \ldots, mn+1$$

Position of the top and bottom pole is respectively equal to:

$$\mathbf{p}_0 = [0, r, 0, 1]^T$$

$$\mathbf{p}_{mn+1} = [0, -r, 0, 1]^T$$

Other vertices are positioned as follows:

$$\mathbf{p}_{im+j} = \begin{bmatrix} r\cos\left(\frac{2\pi}{m}\left(j-1\right)\right)\sin\left(\frac{\pi}{n+1}i\right) \\ r\cos\left(\frac{\pi}{n+1}i\right) \\ r\sin\left(\frac{2\pi}{m}\left(j-1\right)\right)\sin\left(\frac{\pi}{n+1}i\right) \\ 1 \end{bmatrix},\ i = 1, \ldots, n;\ j = 1, \ldots, m$$

Top and bottom "lid" (that contain the poles) consist of triangles:

$$\mathrm{T}_i = \begin{cases} (\mathrm{V}_0, \mathrm{V}_{i+2}, \mathrm{V}_{i+1}) & i = 0, \ldots, m-2 \\ (\mathrm{V}_0, \mathrm{V}_1, \mathrm{V}_m) & i = m-1 \end{cases}$$

$$\mathrm{T}_{2(n-1)m+i} = \begin{cases} \left(\mathrm{V}_{mn+1}, \mathrm{V}_{(n-1)m+i+1}, \mathrm{V}_{(n-1)m+i+2}\right) & i = 0, \ldots, M-2 \\ \left(\mathrm{V}_{mn+1}, \mathrm{V}_{mn}, \mathrm{V}_{(n-1)m+1}\right) & i = m-1 \end{cases}$$

and rings making up the strips between parallels consist of triangles (for $i = 0, \ldots, n-2$):

$$\mathrm{T}_{(2i+1)m+j-1} = \begin{cases} \left(\mathrm{V}_{im+j}, \mathrm{V}_{im+j+1}, \mathrm{V}_{(i+1)m+j+1}\right) & j = 1, \ldots, m-1 \\ \left(\mathrm{V}_{(i+1)m}, \mathrm{V}_{im+1}, \mathrm{V}_{(i+1)m+1}\right) & j = m \end{cases}$$

$$\mathrm{T}_{(2i+2)m+j-1} = \begin{cases} \left(\mathrm{V}_{im+j}, \mathrm{V}_{(i+1)m+j+1}, \mathrm{V}_{(i+1)m+j}\right) & j = 1, \ldots, m-1 \\ \left(\mathrm{V}_{(i+1)m}, \mathrm{V}_{(i+1)m+1}, \mathrm{V}_{(i+2)m}\right) & j = m \end{cases}$$

2.2 Light sources

There are three types of light sources that can be placed in the scene. Each light source will have its color (usually white). Other parameters depend on the light source type:

- **Point light** defined by its position - a point in the global coordinate system
- **Directional light** defined by its direction - a vector in the global coordinate system
- **Spot light** defined by position, direction (expressed in global coordinate system) and a coefficient describing the size of the cone of light.

Description of those parameters and how they affect the lighting in the scene can be found in section 5.1.

2.3 Camera

Camera describes the point-of-view (position and orientation) used to observe the scene (i.e. how the image plane of the window is positioned in relation to the rest of the scene). To properly display the scene on the screen we need to transform all objects from the coordinate system of the scene (i.e. the global coordinate system) to the coordinate system of the camera (see section 3.2). Another transformation connected to the camera i projection, which will allow us to project points from the 3D space to the surface of a 2D image plane (see subsection 3.3).

3 Transformations

In the previous section two types of transformations where mentioned: transformation between two 3D coordinate systems and projection. To represent those transformations in an uniform manner, each three-dimensional point **p** an vector **v** needs to be expressed using so called *affine*, aka. *homogeneous*, coordinates. Using this representation, each point and vector has four elements, first three are equal to its coordinates in 3D, and the fourth one is equal to 1 for points and 0 for vectors.

$$\mathbf{p} = [p_x, p_y, p_z, 1]^T$$
$$\mathbf{v} = [v_x, v_y, v_z, 0]^T$$

This will allow us to express all transformations as 4×4 matrices. Transformation of a point or vector will be performed by multiplying such matrix by a 4-element vector:

$$v' = Mv$$

Warning! Even though all transformations can be expressed in the same manner, only points are subject to projection. Vectors are never projected and due to the way projection matrix is constructed, that kind of operation has no geometrical meaning.

3.1 Transformation between coordinate systems

Transformation between two coordinate systems is expressed by a so called *affine* transformation matrix in the form of:

$$M = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

First three columns define the change of orientation of the axes. The last column describes the shift of point of origin. This explains the choice of 1 as

the fourth coordinate for points and 0 for vectors, since coordinates of vectors are only affected by the change in orientation of axes, the coordinates of points however, are also modified by the translation of the origin, described by the last column.

3.1.1 Geometric interpretation of affine transformation

Elements of affine transformation matrix can be interpreted as follows:

$$M = \begin{bmatrix} X_x^{src} & Y_x^{src} & Z_x^{src} & 0_x^{src} \\ X_y^{src} & Y_y^{src} & Z_y^{src} & 0_y^{src} \\ X_z^{src} & Y_z^{src} & Z_z^{src} & 0_z^{src} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{X}^{src}$, $\mathbf{Y}^{src}$ i $\mathbf{Z}^{src}$ define coordinates of unit vectors of the axes of the source coordinate system expressed in destination coordinate system. Similarly $\mathbf{0}^{src}$ describes coordinates of point of origin of the source coordinate system expressed in the destination one.

3.1.2 Construction of affine transformation matrix

Construction of transformation matrix, where the axes and the origin of the source coordinate system are given in the destination one is trivial. Often however, we encounter the opposite case where axes $\mathbf{X}^{dst}$, $\mathbf{Y}^{dst}$, $\mathbf{Z}^{dst}$ and the origin $\mathbf{0}^{dst}$ of the destination coordinate system are expressed using source coordinates. Those can be used to construct the reverse transformation M^{-1}:

$$M^{-1} = \begin{bmatrix} X_x^{dst} & Y_x^{dst} & Z_x^{dst} & 0_x^{dst} \\ X_y^{dst} & Y_y^{dst} & Z_y^{dst} & 0_y^{dst} \\ X_z^{dst} & Y_z^{dst} & Z_z^{dst} & 0_z^{dst} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To calculate the desired transformation M, we need to invert the above matrix.

That matrix can be decomposed to:

$$M_{-1} = TO$$

where:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0_x^{dst} \\ 0 & 1 & 0 & 0_y^{dst} \\ 0 & 0 & 1 & 0_z^{dst} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$O = \begin{bmatrix} X_x^{dst} & Y_x^{dst} & Z_x^{dst} & 0 \\ X_y^{dst} & Y_y^{dst} & Z_y^{dst} & 0 \\ X_z^{dst} & Y_z^{dst} & Z_z^{dst} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If both coordinate systems are orthonormal (i.e. axes vectors in each are unit length and perpendicular), then:

$$T^{-1} = -T$$

$$O^{-1} = O^T$$

and as a result:

$$M = \left(M^{-1}\right)^{-1} = (TO)^{-1} = O^{-1}T^{-1} = O^T\,(-T)$$

Multiplying the two matrices gives us:

$$M = \begin{bmatrix} X_x^{dst} & X_y^{dst} & X_z^{dst} & \mathbf{X}^{dst} \cdot \mathbf{0}^{dst} \\ Y_x^{dst} & Y_y^{dst} & Y_z^{dst} & \mathbf{Y}^{dst} \cdot \mathbf{0}^{dst} \\ Z_x^{dst} & Z_y^{dst} & Z_z^{dst} & \mathbf{Z}^{dst} \cdot \mathbf{0}^{dst} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{A} \cdot \mathbf{B}$ is a three-dimensional dot product of two vectors.

3.1.3 Combinations of basic transformation

Positioning an object (whose position and orientation is defined by a matrix of transformation from the local coordinate system of the model to the global coordinate system of the scene) using the above method may not be the most convenient. Often a more intuitive approach would be to position the object using a series of basic transformations, such as rotations around axes, and translations (shifting of position). We can easily define affine transformation matrices for those basic operations. Combination of those operations will be equivalent to multiplication of their respective matrices.

Rotation matrices We will define three variants of rotation matrix, one for rotation around each of the axes. For each the parameter α will define the angle of rotation.

$$R_X\,(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_Z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation matrix Translation, usually represented by a translation vector $\mathbf{t} = [t_x, t_y, t_z]^T$, can also be expressed as an affine matrix:

$$T(\mathbf{t}) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combination of transformations Combination of a series of consecutive transformations (in order first to last) $M_1, \ldots, M_n$ can be expressed by a single matrix M obtained by multiplying matrices of those transformations as follows:

$$M = M_n \cdots M_1$$

In that case the first transformation applied will be M_1, the second will be M_2, etc. *Warning!* Combination of transformation, similarly to matrix multiplication is not commutative, and thus the order of operations is important.

3.2 Construction of camera (view) matrix

Camera matrix (aka. view matrix) can be created in many different ways, depending on the parameters used to control it. In this example we will show a formula for camera matrix defined by:

- **cPos** — position of the camera expressed as a point in the coordinate system of the scene
- **cTarget** — position of camera target (i.e. a point the camera is "looking at") expressed in coordinate system of the scene
- **cUp** — auxiliary vector pointing "up" also expressed in coordinate system of the scene, which defines camera orientation (*Warning!* It should not be parallel to a line going through points **cPos** and **cTarget**)

Based on those parameters we can define an orthonormal local coordinate system of the camera (expressed in the coordinate system of the scene):

$$\mathbf{cZ} = \frac{\mathbf{cPos} - \mathbf{cTarget}}{\|\mathbf{cPos} - \mathbf{cTarget}\|}$$

$$\mathbf{cX} = \frac{\mathbf{cUp} \times \mathbf{cZ}}{\|\mathbf{cUp} \times \mathbf{cZ}\|}$$

$$\mathbf{cY} = \frac{\mathbf{cZ} \times \mathbf{cX}}{\|\mathbf{cZ} \times \mathbf{cX}\|}$$

where $\mathbf{A} \times \mathbf{B}$ is a three-dimensional cross product of two vectors and $\|\mathbf{A}\|$ is vectors length.

cPos is the origin of the camera coordinate system, and its axes are defined by unit length perpendicular vectors **cX**, **cY**, **cZ**. Using the method described in 3.1.2 we can calculate the matrix as follows:

$$M = \begin{bmatrix} cX_x & cX_y & cX_z & \mathbf{cX} \cdot \mathbf{cPos} \\ cY_x & cY_y & cY_z & \mathbf{cY} \cdot \mathbf{cPos} \\ cZ_x & cZ_y & cZ_z & \mathbf{cZ} \cdot \mathbf{cPos} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.3 Perspective projection matrix

The goal of perspective projection is to project positions of vertices from three-dimensional space onto two-dimensional image plane of the screen in accordance with perspectivity, i.e. the further the point is from the camera, the closer its projection is to the center of the image.

3.3.1 Construction of perspective projection matrix

The following perspective projection will be defined independently of the pixel dimensions of the image, distance of the user from the screen, etc. For that, we will assume that we are observing the 3D scene through a virtual window $[-1, 1] \times [-1, 1]$ on an image plane perpendicular to **Z** axis. The distance of the image plane from the camera is such that the window provides a horizontal field of view of θ.

Since our square $[-1, 1] \times [-1, 1]$ window will be stretched over the inside of rectangular program window, we need to take into account its aspect ratio:

$$a = \frac{screen_{height}}{screen_{width}}$$

Projection matrix should contain one additional operation. Besides the onscreen coordinates of the projected vertex position, we often need to preserve a value related to its z coordinate. We choose to map z coordinates to a range of $[0, 1]$ (lower values indicate points closer to the camera). The mapping should be monotonic, so it can be used in implementation of Z-buffer algorithm (see 4.4.1), and also non-linear, so it will help with interpolation of other vertex attributes (e.g. normal vectors, texture coordinates, etc.) inside projected triangles (see 4.3.1). Fitting all possible z values into $[0, 1]$ range would cause numerical errors, hence we will limit the scene drawn on the screen by so called *near* and *far* clipping plane, which are parallel to the image plane and their distances from the camera are equal to n and f respectively. Points, that don't lie between these planes will not be drawn on the screen.

Perspective projection matrix that fulfills all of those requirements, parameterized by values θ, a, n and f is given by formula:

$$\mathbf{P} = \begin{bmatrix} \operatorname{ctg}\frac{\theta}{2} & 0 & 0 & 0 \\ 0 & \frac{\operatorname{ctg}\frac{\theta}{2}}{a} & 0 & 0 \\ 0 & 0 & \frac{-f}{f-n} & \frac{-fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Exact derivation of the above formula can be found here: http://bit.ly/20AkEEe - *Warning!* Due to a difference in the interpretation of matrix and vector multiplication, the formula for a matrix derived on that website is a transposition of the one presented here.

3.3.2 Interpretation of projection results

Multiplying a point:

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

by a projection matrix $\mathbf{P}$ results in the following:

$$\mathbf{p}'' = \begin{bmatrix} x'' \\ y'' \\ z'' \\ w'' \end{bmatrix} = \mathbf{P}\mathbf{p} = \begin{bmatrix} x\operatorname{ctg}\frac{\theta}{2} \\ \frac{y}{a}\operatorname{ctg}\frac{\theta}{2} \\ \frac{f(-z-n)}{f-n} \\ -z \end{bmatrix}$$

In general, that will not be the final result, because the resulting point expressed in affine coordinates is not normalized (i.e. the fourth coordinate is not equal to 1). An additional step after projection should be to normalize the point by dividing its coordinates by the fourth one:

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \mathbf{p}''/w'' = \begin{bmatrix} -\frac{x}{z}\operatorname{ctg}\frac{\theta}{2} \\ -\frac{y}{az}\operatorname{ctg}\frac{\theta}{2} \\ -\frac{f(-z-n)}{z(f-n)} \\ 1 \end{bmatrix}$$

4 Scene drawing algorithm

This section will present an algorithm for drawing a single mesh. In a scene containing multiple meshes, each of them can be drawn independently (bearing in mind the visibility problem).

4.1 Vertex transformation

Fist step of the algorithm is to transform the vertices of a mesh. Fist we need to find the position and normal vector coordinates of the vertex expressed in the global coordinate system. For that they should be multiplied by the local-to-global transformation matrix of the mesh. Resulting position $\mathbf{p}^G$ and normal vector $\mathbf{n}^G$ in the coordinate system of the scene should be stored for the purpose of lighting calculation (see Phong illumination model in section 5). Next position should be transform to the camera coordinate system and multiplied by projection matrix. Result of that multiplication $\mathbf{p}''$ should also be stored in the vertex for the purpose of interpolation of vertex attributes (see section 4.3.1). Finally we can normalize the result of projection (see section 3.3.2) obtaining $\mathbf{p}'$ that describes the position of the projection in the image window. Ultimately, after all transformations each vertex should be described with four components:

$$\mathrm{V} = \left(\mathbf{p}^G, \mathbf{n}^G, \mathbf{p}'', \mathbf{p}'\right)$$

4.2 Back-face culling

After projection we can find out which triangles face towards and which face away from the camera. If the meshes are closed, the triangles facing backwards will be covered by those facing forward, so drawing them is not necessary. If we introduced consistent ordering of vertices in triangles (see section 2.1), triangles facing forward should have projections of their vertices ordered clockwise. Assuming a triangle is defined by vertices (after projection):

$$V_1 = \left(\mathbf{p}_1^G, \mathbf{n}_1^G, \mathbf{p}_1'', \mathbf{p}_1'\right),\ V_2 = \left(\mathbf{p}_2^G, \mathbf{n}_2^G, \mathbf{p}_2'', \mathbf{p}_2'\right),\ V_3 = \left(\mathbf{p}_3^G, \mathbf{n}_3^G, \mathbf{p}_3'', \mathbf{p}_3'\right)$$

the order of their projected positions:

$$\mathbf{p}_1' = \begin{bmatrix} x_1' \\ y_1' \\ z_1' \\ 1 \end{bmatrix},\ \mathbf{p}_2' = \begin{bmatrix} x_2' \\ y_2' \\ z_2' \\ 1 \end{bmatrix},\ \mathbf{p}_3' = \begin{bmatrix} x_3' \\ y_3' \\ z_3' \\ 1 \end{bmatrix}$$

can be easily checked by calculating a cross product of:

$$\begin{bmatrix} x_2' - x_1' \\ y_2' - y_1' \\ 0 \end{bmatrix} \times \begin{bmatrix} x_3' - x_1' \\ y_3' - y_1' \\ 0 \end{bmatrix}$$

If the third component of the result is positive, triangle is front facing, otherwise it faces away from the camera and should be ignored.

4.3 Clipping

Some triangles of the mesh might be partially or entirely invisible, if one or more of triangle vertices lie outside of view frustum defined by the camera position, view window, and far and near clipping plane. Before we attempt to draw the triangles, we need to clip them to only the visible part.

Vector $\mathbf{p}'$ obtained as a result of projection described above:

$$\mathbf{p}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

contains coordinates x' and y' describing its position within image window, and value z' corresponding to the distance of the point before projection from camera. As described before, the projection matrix is constructed so that visible points should fall within $[-1, 1] \times [-1, 1] \times [0, 1]$. Hence each triangle should be clipped to that cuboid, using a three-dimensional variant of any of the known clipping algorithms.

4.3.1 Interpolation along projected line segments

When a clipping algorithm clips an edge of the projected polygon, introduces a new vertex and calculates its projected position. However, for the purpose of illumination calculations or texturing we also need to calculate other vertex attributes for that new vertex. Unfortunately, linear interpolation between projected point does not correspond to linear interpolation in the 3D space before projection, as demonstrated by figure 1.

Let $\mathrm{V}_1, \mathrm{V}_2$ be the vertices of an edge subdivided by a clipping algorithm:

$$\mathrm{V}_1 = (\mathbf{p}_{W1}, \mathbf{n}_{W1}, \mathbf{p}'_1, \mathbf{p}''_1)$$

$$\mathrm{V}_2 = (\mathbf{p}_{W2}, \mathbf{n}_{W2}, \mathbf{p}'_2, \mathbf{p}''_2)$$

$$\mathbf{p}''_1 = \begin{bmatrix} x''_1 \\ y''_1 \\ z''_1 \\ w''_z \end{bmatrix}, \ \mathbf{p}'_1 = \begin{bmatrix} x'_1 \\ y'_1 \\ z'_1 \\ 1 \end{bmatrix}$$

$$\mathbf{p}''_2 = \begin{bmatrix} x''_2 \\ y''_2 \\ z''_2 \\ w''_z \end{bmatrix}, \ \mathbf{p}'_2 = \begin{bmatrix} x'_2 \\ y'_2 \\ z'_2 \\ 1 \end{bmatrix}$$

Let the edge be subdivided at point $\mathbf{p}'_t$ obtained by linear interpolation between $\mathbf{p}'_1$ and $\mathbf{p}'_2$ for a given value of coefficient $t \in (0,1)$ as follows:

$$\mathbf{p}'_t = \begin{bmatrix} x'_t \\ y'_t \\ z'_t \\ 1 \end{bmatrix} = (1-t)\,\mathbf{p}'_1 + t\mathbf{p}'_2$$

We need to find the corresponding vertex:

$$\mathrm{V}_t = \left(\mathbf{p}_t^G, \mathbf{n}_t^G, \mathbf{p}'_t, \mathbf{p}''_t\right)$$

$$\mathbf{p}''_t = \begin{bmatrix} x''_t \\ y''_t \\ z''_t \\ w''_t \end{bmatrix}$$

where

$$\mathbf{p}'_t = \mathbf{p}''_t / w''_t = \begin{bmatrix} \frac{x''_t}{w''_t} \\ \frac{y''_t}{w''_t} \\ \frac{z''_t}{w''_t} \\ 1 \end{bmatrix}$$

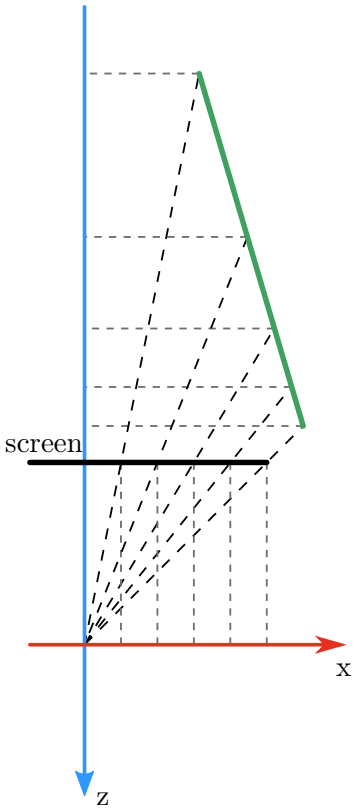

Figure 1: Linear interpolation of vertex projections

V_t lies on the edge$\mathrm{V}_1\mathrm{V}_2$, hence we know there exist such value $u \in (0,1)$, so that:

$$\mathbf{p}''_t = (1-u)\,\mathbf{p}''_1 + u\mathbf{p}''_2$$

$$\mathbf{p}_{Wt} = (1-u)\,\mathbf{p}_{W1} + u\mathbf{p}_{W2}$$

$$\mathbf{n}_{Wt} = (1-u)\,\mathbf{n}_{W1} + u\mathbf{n}_{W2}$$

Value of coefficient u can be derived from equation:

$$z'_t = \frac{z''_t}{w''_T} = \frac{(1-u)\,z''_1 + uz''_2}{(1-u)\,w''_1 + uw''_2}$$

which solved for u produces formula:

$$u = \frac{z'_t w''_1 - z''_1}{z'_t\,(w''_1 - w''_2) - (z''_1 - z''_2)}$$

Analogously u can be derived from equations:

$$x'_t = \frac{x''_t}{w''_t}$$

$$y'_t = \frac{y''_t}{w''_t}$$

Above formulas are in general ill-conditioned, therefore we should use that equation where the difference in corresponding coordinates of the edge vertices is the greatest. In addition, if the edge is parallel to the image plane, then $u = t$.

4.4 Triangle filling

Next step involves drawing the projected and clipped polygon onto the screen. Coordinates (x', y') need to be mapped from the window $[-1, 1] \times [-1, 1]$ to the actual pixel coordinates of the bitmap on which the image is drawn, so that point $(-1, -1)$ is mapped to the bottom-left corner, and point $(1, 1)$ to top-right corner of the bitmap. The polygon can then be filled using any of the presented variants of the scanline algorithm. It is important however, to properly interpolate other vertex attributes (normal vectors, texture coordinates) along the polygon edges and along each scanline using the same approach as described in section 4.3.1.

4.4.1 Z-buffer

If the scene consists of a single convex triangle mesh and as long as back-face culling is performed, no two triangle projections drawn on the screen should overlap. Otherwise an additional algorithm should be used to solve the visibility problem.

To ensure correct visibility of triangle mesh faces regardless of the order in which individual triangles are drawn the solution presented here makes use of a so called *Z-Buffer*, which is a two-dimensional array of the same size as the bitmap used for drawing the scene. It will be used to store z' values of the pixels drawn on the screen.

The values of z' are real numbers, however, due to non-linearity of the mapping between z coordinate in the camera coordinate system and z', for objects that are far from the camera, these values will be very close to 1. Storing those values as `floats` not provide enough precision to differentiate two distinctly different values of z'.

Thanks to the fact, that values of z' for points visible on screen are within the range of $[0, 1]$, we can create the Z-Buffer as an array of unsigned integers, and when storing values, remap the range $[0, 1]$ to $[0, \texttt{UINT\_MAX}]$. Using this representation we trade some precision for values close to 0, where it is less important, for increased precision close to 1.

When filling a polygon, each time a pixel would be drawn, first the z-buffer test needs to be performed. If the z' value for the pixel is larger than the one stored in the corresponding Z-Buffer element, the pixel is not drawn.

Otherwise, the pixel should be filled and the corresponding element of the Z-Buffer should be updated with z' value of the pixel.

4.4.2 Coloring pixels

If the Phong illumination model is used (as described in section 5), the color of the pixels within a triangle face is not constant. To calculate the color of a pixel the following parameters need to be determined for a given pixel:

- $\mathbf{p}^G$ - coordinates of a point on the mesh corresponding to the given pixel, expressed in the global coordinate system
- $\mathbf{n}^G$ - coordinates of a normal vector at a point on the mesh corresponding to the given pixel, expressed in the global coordinate system (it is advised to normalize the normal vectors before further calculations, since interpolation can produce vectors that are not unit length).
- Material coefficients of the mesh (see section 5.2)
- Positions and/or directions (expressed in global coordinate system) and colors of all the light sources in the scene (see section 5.1)
- Position of the camera expressed in the global coordinate system

5 Phonga illumination model

What follows is an approximated model of interaction of surfaces with light, known as Phong illumination model. It allows us to find the intensity (color) of the light reflected by the surface at the given point in a given direction to the observer.

The intensity is described by a three-component vector $[r, g, b]^T$, representing red, green, and blue component of the reflected light. For the purpose of following calculations, we assume that each component is a real value in range of $[0, 1]$, and vector $[1, 1, 1]$ represents white color.

Firstly, we will introduce parameters and coefficients defining the light sources and reflective properties of materials the surfaces are made of.

5.1 Light sources

Phong illumination model describes three types of light sources: point, directional, and spot lights. Each light is described by its color. In addition to that point light is defined by its position and illuminates objects equally in all directions. Directional light is not attached to any given point, but instead is defined by a direction, meaning, that for any point in the scene, the light comes from the same direction. Spot lights are described by both

position and direction. Points illuminated the most lie on a halfline starting from its position and going in its direction. The further the point is from that line, the less illuminated it is by this light source.

For the i^{th} light source, let:

- $\mathbf{p}_i$ be light source position expressed in global coordinate system (available only for point and spot lights)
- $\mathbf{d}_i$ be light source direction vector, expressed in global coordinate system (available only for directional and spot lights)
- I_i^p be light source base intensity (color — available for all light sources)
- r_i be spotlight focus coefficient (available only for spot lights; recommended values $\sim 10 - 100$).

5.2 Materials

In the Phong illumination model, the light reflected from a surface has three components. The first one is ambient reflection which is constant, emits equally in all directions and is independent of the relative positions of the surface, light sources and the observer. It represents light scattered uniformly throughout the whole scene. The second component is diffuse reflection. It consist of light coming from the light source that is reflected by the surface equally in all directions. The amount of light reflected depends on the angle between the normal vector and the vector pointing to the light source. It imitates matte surfaces. Third component, called specular reflection, where the intensity of the emitted light is the strongest in the direction of the oncoming light reflected of the surface. It imitates shiny surfaces.

For each different type of reflection, material of the surface should contain a three-element vector of coefficients in range of $[0, 1]$, representing the amount of red, green, and blue light reflected by the surface. In addition, for the specular reflection another scalar coefficient is needed, that describes the focus of this reflection. For a given surface let:

- $\mathbf{k}_a = \left[k_a^r, k_a^g, k_a^b\right]^T$ be a vector of ambient reflection coefficients
- $\mathbf{k}_d = \left[k_d^r, k_d^g, k_d^b\right]^T$ be a vector of diffuse reflection coefficients
- $\mathbf{k}_s = \left[k_s^r, k_s^g, k_s^b\right]^T$ be a vector of specular reflection coefficients
- m be specular focus coefficient

5.3 Calculating point color

Let us assume that we have a point $\mathbf{p}$ in global coordinates that lies on a surface with material coefficients as described in section 5.2 and that the vector normal to the surface at $\mathbf{p}$ is equal to $\mathbf{n}$. To calculate the point color as observed by the camera we need to know:

- $\mathbf{I}_a$ - intensity (color) of ambient light
- $\mathbf{p}_c$ - position of the camera in global coordinates

Formula for the color involves calculating the sum of ambient reflection, and, for each light source, the diffuse and specular reflections, as follows:

$$\mathbf{I} = \mathbf{I}_a\mathbf{k}_a + \sum_i \left(\mathbf{k}_d\mathbf{I}_i \max\left(\langle\mathbf{n}, \mathbf{l}_i\rangle, 0\right) + \mathbf{k}_s\mathbf{I}_i \max\left(\langle\mathbf{v}, \mathbf{r}_i\rangle, 0\right)^m\right)$$

where vector $\mathbf{v}$ defines a direction from the point on the surface to the camera:

$$\mathbf{v} = \frac{\mathbf{p}_c - \mathbf{p}}{\|\mathbf{p}_c - \mathbf{p}\|}$$

vector $\mathbf{l}_i$ defines direction from the point on the surface to the i^{th} light source:

- for point and spot lights

$$\mathbf{l}_i = \frac{\mathbf{p}_i - \mathbf{p}}{\|\mathbf{p}_i - \mathbf{p}\|}$$

- for directional lights

$$\mathbf{l}_i = -\mathbf{d}_i$$

vector $\mathbf{r}_i$ is a direction the light is coming from reflected by the surface at the given point:

$$\mathbf{r}_i = 2\langle\mathbf{n}, \mathbf{l}_i\rangle\mathbf{n} - \mathbf{l}_i$$

and $\mathbf{I}_i$ is the intensity of the light coming to the point from the i^{th} light source:

- for point and directional lights

$$\mathbf{I}_i = \mathbf{I}_i^p$$

- for spot lights

$$\mathbf{I}_i = \mathbf{I}_i^p \max\left(\langle-\mathbf{d}_i, \mathbf{l}_i\rangle, 0\right)$$