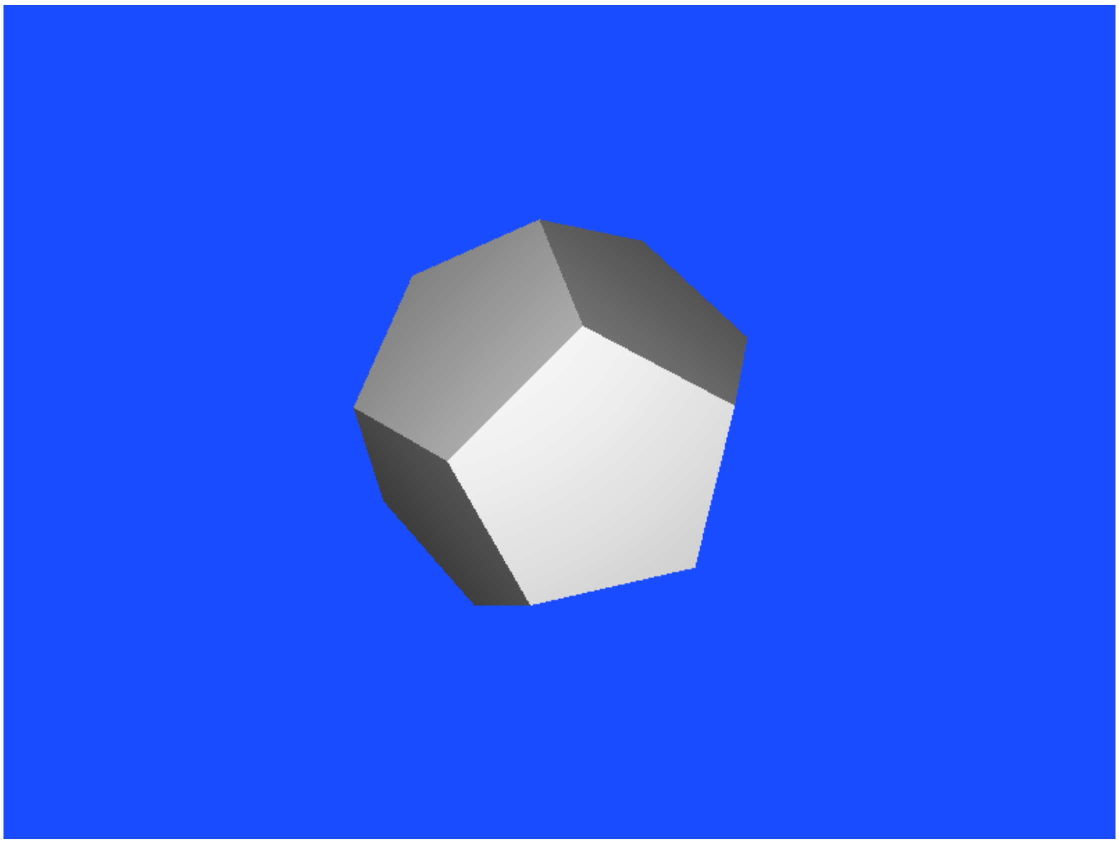

Assignment 3 — Going 3D

Deadline: 2016-12-06 23:59

To move on into the realm of 3D graphics, we need to compute the image of the projection of the objects in a virtual scene onto the image plane of a virtual camera. For scenes containing objects modeled using triangles, we simply project all the vertices of all the objects onto the image plane and draw the respective triangles. Visibility can be resolved using a Z-Buffer. Essential for a realistic effect is to apply some form of shading to color the objects.

Task 3 (20 points)

Extend the application «`task3`» such that it renders an image of a three-dimensional object.

- Create a simple, three-dimensional object like, e. g., a platonic solid.[1]
  - Store the vertex data in one or more buffer objects.
  - Set up an appropriate vertex array object describing your vertex data layout.
- Write a vertex shader that
  - maps the object to the screen according to a given model, view, and projection matrix as well as
  - outputs suitable normal vectors for shading computations in the fragment shader.
  - Use uniform variables to pass information such as, e. g., matrices to your shaders.
- Write a fragment shader that
  - implements simple lambertian shading with
  - light coming from a point light source located at the position of the camera.
- Set up a camera (view and projection matrix) looking at your object.
- Demonstrate the use of the model matrix by either
  - animating the object (e. g. have the object spin or move around), or
  - rendering the object multiple times in different locations and orientations.
- Draw your object.

The following sections are intended to provide you with a short recap of the basics relevant to this exercise. A more detailed discussion of the mathematical background can be

[1] `http://en.wikipedia.org/wiki/Platonic_solid`

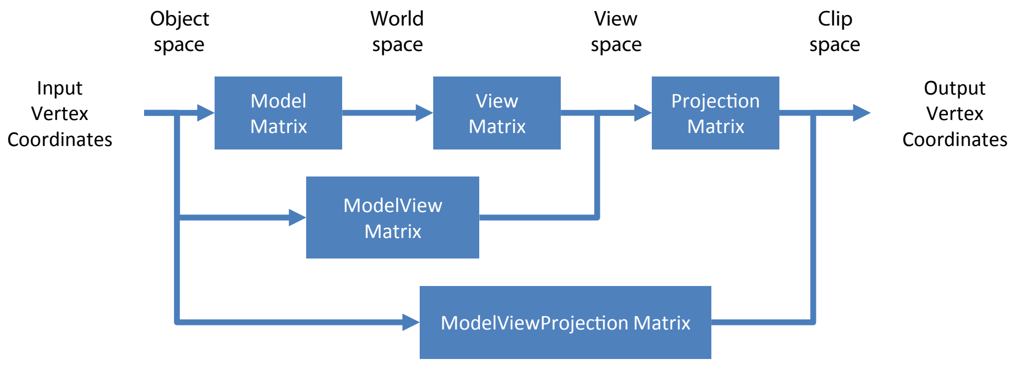

Figure 1: The sequence of transformations typically implemented in a vertex shader used to draw three-dimensional objects, the coordinate systems involved, and the matrix transforms that relate them.

found, e. g., in the book by Lengyel [**lengyel2004mathematics**]. The results important for solving this task are found in equations (5), (6), (3), (4), and (7), as well as in section 3.1.3.

3.1 3D Transforms

A typical 3D scene is generally composed of a number of individual objects. Rendering an image of such a scene as seen from a certain point of view requires us to project the scene onto the image plane of a virtual camera.

In real-time graphics, we usually represent objects by modeling their surface using triangles (a so-called *boundary representation*). One of the many nice properties of triangles is that the perspective projection of a triangle onto a plane is still a triangle. Therefore, simply projecting each vertex onto the image plane will yield the projection of the whole scene.

Figure 1 shows the sequence of transformations commonly employed to implement the mapping of object vertices onto the image plane. An object is usually modeled in its own *local coordinate system* also called *object space*. A *model matrix* (often also called *world matrix*) describes how an object is placed in the scene, it transforms local object coordinates into global *world coordinates*. Ultimately, we need to find the coordinates of the object's vertices' projection on the screen. Since we generally want to place our camera in the scene freely, the first step towards this goal is to compute the coordinates of all points as they appear relative to the camera. In other words: we need to transform the vertex positions from world space into *view space*, the local coordinate system of the camera (where the camera is located at the origin, looking down the negative z-axis). This transformation is described by the *view matrix*. The *projection matrix* then finally maps points from view space onto the image plane.

Long story short: An object's model matrix relates the object to the world, the view matrix relates the world to the camera, and the projection matrix maps from the camera to the screen. In practice, these three matrices are usually *concatenated*, i. e., multiplied together, to form a single combined *model-view-projection matrix* that directly maps an object to the screen. This matrix is then passed to the vertex shader where it is applied to each vertex.

3.1.1 Homogeneous coordinates

In three-dimensional euclidean space, some important coordinate transformations like translation and perspective projection are not linear and, therefore, cannot be described by a matrix. However, when we extend our euclidean space into a real projective 3-space, adding a fourth dimension, these transformations can be expressed as linear mappings in so called *homogeneous coordinates*. Three-dimensional euclidean space then corresponds to the subspace where the fourth coordinate, w, is one. The homogeneous coordinates of a point are therefore given by the mapping

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \mapsto \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}. \tag{1}$$

Vice versa, dividing all coordinates by the w-coordinate leads to the projection of general homogeneous coordinates ($w \in \mathbb{R} \setminus \{0\}$) back to a point with $w = 1$, i. e., euclidean space:

$$\begin{bmatrix} p'_x \\ p'_y \\ p'_z \\ p'_w \end{bmatrix} \mapsto \frac{1}{p'_w} \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}. \tag{2}$$

Rasterization hardware The vertex shader output `gl_Position` is interpreted by the GPU as the vertex' homogeneous coordinates in so called *clip space*.[2] When projected according to (2), clip space coordinates lead to *normalized device coordinates*, where the unit cube $[-1,1]^3$ corresponds to the current viewport—more specifically: $\begin{bmatrix}-1 & 1 & \cdot\end{bmatrix}^T$ marks the bottom left and $\begin{bmatrix}1 & 1 & \cdot\end{bmatrix}^T$ the top right corner, the z-coordinate is the depth used for depth buffering. Anything that projects outside the unit cube will be clipped away. The GPU takes the vertex shader output, performs primitive clipping, and projects clip space coordinates to normalized device coordinates, which are finally mapped to the viewport, resulting in *window coordinates*, i. e., pixel coordinates in the framebuffer.

[2]The coordinate space where clipping is performed, hence the name.

Note: Clip space is basically the only coordinate system the GPU understands natively. The GPU is completely unaware of any higher-level concepts such as world space, a camera, or an object. It just sees a bunch of primitives and draws them. It is up to us to construct these primitives in such a way, that they form images of three-dimensional objects on the screen.[3] The sequence of model, view and projection transformations described above is nothing but one approach to do so in a humanly meaningful way.

3.1.2 Projection transform

Given a point $\mathbf{p} = \begin{bmatrix} p_x & p_y & p_z & 1 \end{bmatrix}^T$ in view space, we would like to compute the coordinates of the perspective projection of that point onto the image plane located at $z = -z_n$. As illustrated in Figure 2, the x and y coordinates of the projected point $\mathbf{p}'$ are given by

$$p''_x = z_n \frac{p_x}{-p_z} \qquad p''_y = z_n \frac{p_y}{-p_z} \qquad p''_z = -z_n,$$

which can be expressed in homogeneous coordinates as

$$\mathbf{p}'' = \begin{bmatrix} z_n p_x \\ z_n p_y \\ z_n p_z \\ -p_z \end{bmatrix}.$$

However, since the $[-1,1]^3$ unit cube in normalized device coordinates corresponds to the viewport, we need to not only project points to our virtual projection screen, but also scale the x and y-coordinates such that the projection screen is mapped to the $[-1,1]^2$ domain. Further, we need to map the z-coordinates to the $[-1,1]$ range to get suitable depth values for depth buffering.

As can be seen in Figure 2, the view frustum is completely defined by the *near clipping plane* $z = z_n$, the *far clipping plane* $z = z_f$, and the width w and height h of our projection screen in view space. Instead of specifying the screen width and height in view space directly, a more intuitive measure is usually used: the vertical *field of view* angle β. Given β, we can compute h:

$$h = 2 z_n \tan \frac{\beta}{2}$$

and from the *aspect ratio*

$$a = \frac{w}{h}, \tag{3}$$

[3] As a consequence, the GPU can, of course, be used to draw not just shapes of three-dimensional origin, but any kind of shape we can make out of the available primitives.

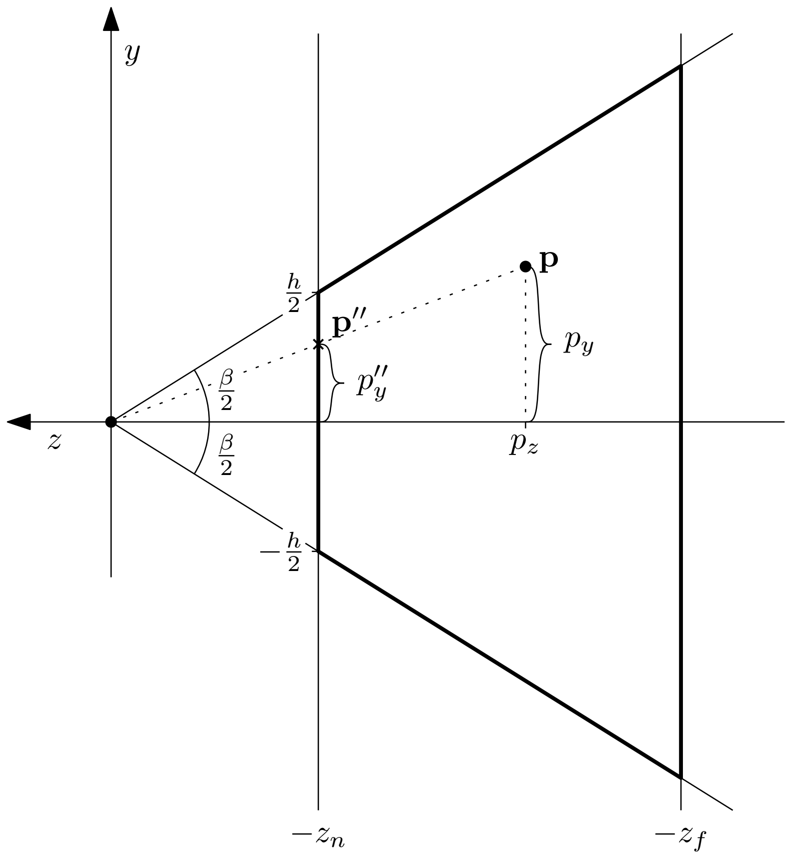

Figure 2: Perspective projection of a point $\mathbf{p}$ onto the near plane located at $z = -z_n$.

of the current viewport, we can then compute w:

$$w = ah.$$

All that is left to do then, is to scale p_x by $\frac{1}{w}$ and p_y by $\frac{1}{h}$.

Since we have to divide by p_z to achieve perspective projection, i. e., $p'_w = p_z$ is a given, we cannot use a simple scaling and offset to map the z-coordinate to the $[-1,1]$ range directly. We need to find a linear combination[4]

$$p'_z = m_1 p_z + m_2$$

that, when divided by $p'_w = p_z$, will lead to a monotonic (necessary for the depth test to work) mapping where $z_n \mapsto -1$ and $z_f \mapsto 1$. It can be shown that such a mapping is achieved by

$$p'_z = -\frac{z_f + z_n}{z_f - z_n} p_z - \frac{2 z_f z_n}{z_f - z_n}.$$

Implementing everything discussed above, we arrive at the projection matrix

$$\mathbf{P} = \begin{bmatrix} \frac{1}{a \tan\frac{\beta}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\beta}{2}} & 0 & 0 \\ 0 & 0 & -\frac{z_f+z_n}{z_f-z_n} & -\frac{2z_f z_n}{z_f-z_n} \\ 0 & 0 & -1 & 0 \end{bmatrix}. \qquad (4)$$

3.1.3 Model transform

The model matrix $\mathbf{M}$ places an object in the world. It is often constructed as a sequence of rotations, translations, and other basic transformations. For example, a rotation $\mathbf{R}_y(\varphi)$ by angle φ about the y-axis, and a translation $\mathbf{T}(t_x, t_y, t_z)$ by offset $\begin{bmatrix} t_x & t_y & t_z \end{bmatrix}^T$ would be given by

$$\mathbf{R}_y(\varphi) = \begin{bmatrix} \cos\varphi & 0 & \sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T}(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

To place an object rotated about its local y-axis by $\frac{\pi}{6}$ at position $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^T$ we would now use the model matrix

$$\mathbf{M} = \mathbf{T}(1,2,3)\mathbf{R}_y\left(\frac{\pi}{6}\right).$$

[4]keep in mind that $p_w = 1$

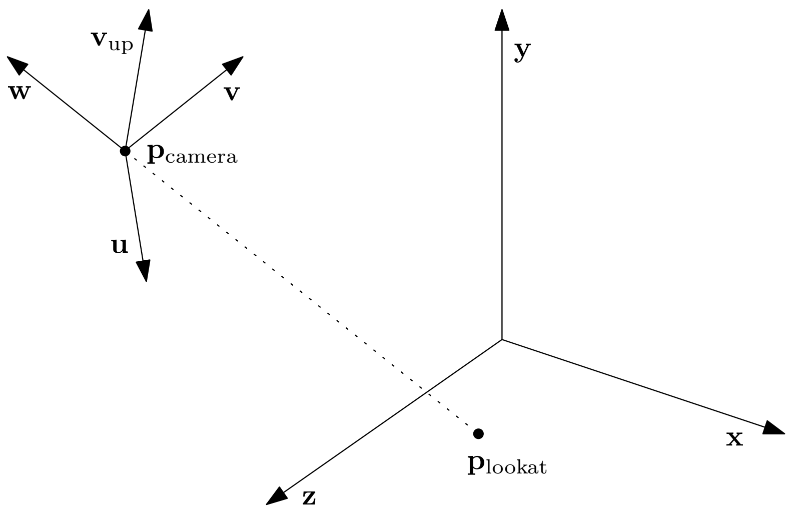

Figure 3: The coordinate frame of the camera is related to world coordinates by the base vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$, and the camera position $\mathbf{p}_{\text{camera}}$. Location and orientation of the camera are usually specified in terms of camera position, the look at point $\mathbf{p}_{\text{lookat}}$, and an up vector $\mathbf{v}_{\text{up}}$ (defines roll of the camera around the viewing direction).

Transforming surface normals When a surface is transformed into a new coordinate system by means of a linear transformation $\mathbf{M}$, the coordinates of the surface normals do not simply also change according to $\mathbf{M}$. Instead, the coordinates of surface normals transform according to the upper 3×3 submatrix of the inverse transpose ${\mathbf{M}^{-1}}^T$.

3.1.4 View transform

The view matrix transforms coordinates of points in world space to their coordinates relative to the camera. Figure 3 shows a typical setup. The location and orientation of the camera is usually specified in terms of the camera position $\mathbf{p}_{\text{camera}}$, the look at point $\mathbf{p}_{\text{lookat}}$, and an up vector $\mathbf{v}_{\text{up}}$. The up vector points in the direction that is to be considered upwards for the camera, its purpose is to define the roll of the camera along the view direction. The base vectors can be derived from these values:

$$\mathbf{w} = \frac{\mathbf{p}_{\text{camera}} - \mathbf{p}_{\text{lookat}}}{\|\mathbf{p}_{\text{camera}} - \mathbf{p}_{\text{lookat}}\|} \qquad \mathbf{u} = \frac{\mathbf{v}_{\text{up}} \times \mathbf{w}}{\|\mathbf{v}_{\text{up}} \times \mathbf{w}\|} \qquad \mathbf{v} = \mathbf{w} \times \mathbf{u}. \tag{5}$$

And the view matrix can then be computed as

$$\mathbf{V} = \begin{bmatrix} u_x & u_y & u_z & -\langle \mathbf{p}_{\text{camera}}, \mathbf{u} \rangle \\ v_x & v_y & v_z & -\langle \mathbf{p}_{\text{camera}}, \mathbf{v} \rangle \\ w_x & w_y & w_z & -\langle \mathbf{p}_{\text{camera}}, \mathbf{w} \rangle \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6}$$

Note Another way to think about the view matrix is as the inverse of the model matrix one would use to place the camera in the scene if it were an object. A model matrix transforms from a local to the global coordinate system, while the view matrix does the reverse, it transforms from the global to a local coordinate system.

3.2 Shading

To keep things simple, we will only apply lambertian shading to our object. A perfectly diffuse reflector evenly reflects incoming light into all directions. Therefore, the apparent brightness of such a surface is independent of a viewer's position. It only depends on the angle of incidence at which light hits the surface. Such a reflector is also called a *lambertian reflector*, after Johann Heinrich Lambert. The lambertian BRDF simply is

$$f_r = \frac{\mathbf{c}_d}{\pi}.$$

The radiance L_o coming from a point on the surface of such reflector, when illuminated by a single light source, is thus given by

$$L_o(\mathbf{n},\mathbf{l}) = \frac{\mathbf{c}_d}{\pi} \circ \mathbf{E}_L \max(\mathbf{n}\cdot\mathbf{l}, 0) \tag{7}$$

where $\mathbf{n}$ is the surface normal, $\mathbf{l}$ is the direction from the point on the surface to the light source (both vectors are assumed to be normalized), $\mathbf{c}_d$ is the diffuse reflectance (an RGB vector corresponding to the surface color), and $\mathbf{E}_L$ the irradiance from the light source (an RGB vector corresponding to the color of the light).

Submission format

Hand in your solution in the form of a ZIP archive using our submission system.[5] Once uploaded, your submission will automatically be built and run on our reference system. You receive an email notification as soon as the results of this automatic test run are available.

Your submission will be unpacked over an unmodified version of the framework. Therefore, only files you modified or added[6] need to be included. Make sure to keep the directory structure in your submitted archive the same as in the framework (root directory: «`rtg`»), otherwise, automated unpacking and reassembly will fail. You can assume, e. g., for the sake of loading assets from files, that the initial working directory of your application is the «`rtg`» root directory.

[5] `https://courseware.icg.tugraz.at`

[6] If you modify any of the CMake scripts, remember to also include them in your submission, otherwise the build will fail.

Words of wisdom

- Keep track of which coordinate system you are in. Computations involving coordinate vectors—like, e. g., shading computations—are only meaningful, if all coordinates are relative to the correct (which usually means: the same) coordinate system. For example, computing the dot product of a light vector given in view space and a surface normal given in object space is entirely possible, but does not produce a meaningful result.

- Be aware of the fact that the data structure of matrices in our in-house math library is **row major**, while OpenGL expects **column major** by default. Either convert your finished matrices before passing to GLSL or indicate the **row major** layout for the compiler in the GLSL source code.

- When setting up your camera, be careful to specify the up vector such that it is not collinear to the viewing direction. Otherwise, the cross product in equation (5) will degenerate to a zero vector and your view matrix collapses.

- The mapping from view space z-coordinates to depth values is highly nonlinear. A general rule of thumb is, to place the far plane as close as possible and the near plane as far away as possible to make optimal usage of the limited depth precision and avoid artifacts.

- Don't let integer division bite you when computing the aspect ratio of your viewport.

- Take advantage of the excellent OpenGL online manual[7].

- Make sure to check out the tutorial on error handling

- Do not forget to display your rendered image using the `swapBuffers()` method of the `context`.

- Consulting online tutorials, you may often see the `glm` library referenced. This library is not installed on the test server and must not be used for these assignments. The in-house math framework already supports many vector and matrix operations. However, in contrast to `glm`, it does not provide methods for setting up a particular matrix, e.g. for projection. This is intentional, since the assignments are in part about understanding the composition and the effects of these matrices, and how their manipulation affects the eventual image.

[7] `http://www.opengl.org/sdk/docs/man/`