GLSL Programming/Vertex Transformations

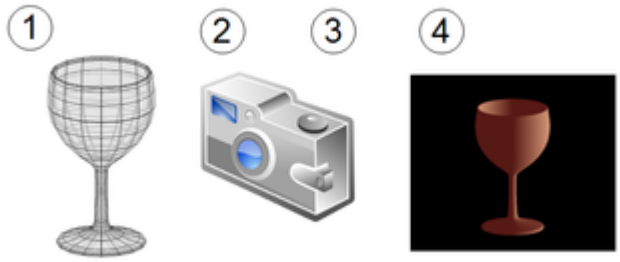
One of the most important tasks of the vertex shader and the following stages in the OpenGL (ES) 2.0 pipeline is the transformation of vertices of primitives (e.g. triangles) from the original coordinates (e.g. those specified in a 3D modeling tool) to screen coordinates. While programmable vertex shaders allow for many ways of transforming vertices, some transformations are performed in the fixed-function stages after the vertex shader. When programming a vertex shader, it is therefore particularly important to understand which transformations have to be performed in the vertex shader. These transformations are usually specified as uniform variables and applied to the incoming vertex positions and normal vectors by means of matrix-vector multiplications. While this is straightforward for points and directions, it is less straightforward for normal vectors as discussed in Section "Applying Matrix Transformations".

Here, we will first present an overview of the coordinate systems and the transformations between them and then discuss individual transformations.

Overview: The Camera Analogy

It is useful to think of the whole process of transforming vertices in terms of a camera analogy as illustrated to the right. The steps and the corresponding vertex transformations are:

1. positioning the model — modeling transformation
2. positioning the camera — viewing transformation
3. adjusting the zoom — projection transformation
4. cropping the image — viewport transformation



The camera analogy: 1. positioning the model, 2. positioning the camera, 3. adjusting the zoom, 4. cropping the image

The first three transformations are applied in the vertex shader. Then the perspective division (which might be considered part of the projection transformation) is automatically applied in the fixed-function stage after the vertex shader. The viewport transformation is also applied automatically in this fixed-function stage. While the transformations in the fixed-function stages cannot be modified, the other transformations can be replaced by other kinds of transformations than described here. It is, however, useful to know the conventional transformations since they allow to make best use of clipping and perspectively correct interpolation of varying variables.

The following overview shows the sequence of vertex transformations between various coordinate systems and includes the matrices that represent the transformations:

object/model coordinates — input to the vertex shader, i.e. position in attributes

↓ **modeling transformation**: model matrix $\mathbf{M}_{\text{object}\rightarrow\text{world}}$

world coordinates

↓ **viewing transformation**: view matrix $\mathbf{M}_{\mathbf{world}\rightarrow\mathbf{view}}$

view/eye coordinates

↓ **projection transformation**: projection matrix $\mathbf{M}_{\mathbf{projection}}$

clip coordinates — output of the vertex shader, i.e. `gl_Position`

↓ **perspective division** (by `gl_Position.w`)

normalized device coordinates

↓ **viewport transformation**

screen/window coordinates — `gl_FragCoord` in the fragment shader

Note that the modeling, viewing and projection transformation are applied in the vertex shader. The perspective division and the viewport transformation is applied in the fixed-function stage after the vertex shader. The next sections discuss all these transformations in detail.

Modeling Transformation

The modeling transformation specifies the transformation from object coordinates (also called model coordinates or local coordinates) to a common world coordinate system. Object coordinates are usually specific to each object or model and are often specified in 3D modeling tools. On the other hand, world coordinates are a common coordinate system for all objects of a scene, including light sources, 3D audio sources, etc. Since different objects have different object coordinate systems, the modeling transformations are also different; i.e., a different modeling transformation has to be applied to each object.

In effect, it 'pushes' the object away from the origin and optionally applies a rotation to it.

Structure of the Model Matrix

The modeling transformation can be represented by a 4×4 matrix, which we denote as the model matrix $\mathbf{M}_{\mathbf{object}\rightarrow\mathbf{world}}$. Its structure is:

$$\mathbf{M}_{\mathbf{object}\rightarrow\mathbf{world}} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{with } \mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{and } \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$\mathbf{A}$ is a 3×3 matrix, which represents a linear transformation in 3D space. This includes any combination of rotations, scalings, and other less common linear transformations. $\mathbf{t}$ is a 3D vector, which represents a translation (i.e. displacement) in 3D space. $\mathbf{M}_{\mathbf{object}\rightarrow\mathbf{world}}$ combines $\mathbf{A}$ and $\mathbf{t}$ in one handy 4×4 matrix. Mathematically spoken, the model matrix represents an affine transformation: a linear transformation together with a translation. In order to make this work, all three-dimensional points are represented by four-dimensional vectors with the fourth coordinate equal to 1:

$$P = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix}$$

When we multiply the matrix to such a point P, the combination of the three-dimensional linear transformation and the translation shows up in the result:

$$\mathrm{M}_{\text{object}\to\text{world}}\ P = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & t_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & t_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1}p_1 + a_{1,2}p_2 + a_{1,3}p_3 + t_1 \\ a_{2,1}p_1 + a_{2,2}p_2 + a_{2,3}p_3 + t_2 \\ a_{3,1}p_1 + a_{3,2}p_2 + a_{3,3}p_3 + t_3 \\ 1 \end{bmatrix}$$

Apart from the fourth coordinate (which is 1 as it should be for a point), the result is equal to

$$\mathrm{A} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

Accessing the Model Matrix in a Vertex Shader

The model matrix $\mathrm{M}_{\text{object}\to\text{world}}$ can be defined as a uniform variable such that it is available in a vertex shader. However, it is usually combined with the matrix of the viewing transformation to form the modelview matrix, which is then set as a uniform variable. In some versions of OpenGL (ES), a built-in uniform variable `gl_ModelViewMatrix` is available in the vertex shader. (See also Section “Applying Matrix Transformations”.)

Computing the Model Matrix

Strictly speaking, GLSL programmers don't have to worry about the computation of the model matrix since it is provided to the vertex shader in the form of a uniform variable. In fact, render engines, scene graphs, and game engines will usually provide the model matrix; thus, the programmer of a vertex shader doesn't have to worry about computing the model matrix. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, the model matrix has to be computed. (OpenGL before version 3.2, the compatibility profiles of newer versions of OpenGL, and OpenGL ES 1.x provide functions to compute the model matrix.)

The model matrix is usually computed by combining 4×4 matrices of elementary transformations of objects, in particular translations, rotations, and scalings. Specifically, in the case of a hierarchical scene graph, the transformations of all parent groups (parent, grandparent etc.) of an object are combined to form the model matrix. Let's look at the most important elementary transformations and their matrices.

The 4×4 matrix representing the translation by a vector $\mathbf{t} = (t_1, t_2, t_3)$ is:

$$\mathrm{M}_{\text{translation}} = \begin{bmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4×4 matrix representing the scaling by a factor s_x along the x axis, s_y along the y axis, and s_z along the z axis is:

$$\mathbf{M}_{\text{scaling}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 4×4 matrix representing the rotation by an angle α about a normalized axis (x, y, z) is:

$$\mathbf{M}_{\text{rotation}} = \begin{bmatrix} (1-\cos\alpha)x\,x + \cos\alpha & (1-\cos\alpha)x\,y - z\sin\alpha & (1-\cos\alpha)z\,x + y\sin\alpha & 0 \\ (1-\cos\alpha)x\,y + z\sin\alpha & (1-\cos\alpha)y\,y + \cos\alpha & (1-\cos\alpha)y\,z - x\sin\alpha & 0 \\ (1-\cos\alpha)z\,x - y\sin\alpha & (1-\cos\alpha)y\,z + x\sin\alpha & (1-\cos\alpha)z\,z + \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Special cases for rotations about particular axes can be easily derived. These are necessary, for example, to implement rotations for Euler angles. There are, however, multiple conventions for Euler angles, which won't be discussed here.

A normalized quaternion (w_q, x_q, y_q, z_q) corresponds to a rotation by the angle $2\arccos(w_q)$. The direction of the rotation axis can be determined by normalizing the 3D vector (x_q, y_q, z_q).

Further elementary transformations exist, but are of less interest for the computation of the model matrix. The 4×4 matrices of these or other transformations are combined by matrix products. Suppose the matrices $\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{M_3}$ are applied to an object in this particular order. ($\mathbf{M_1}$ might represent the transformation from object coordinates to the coordinate system of the parent group; $\mathbf{M_2}$ the transformation from the parent group to the grandparent group; and $\mathbf{M_3}$ the transformation from the grandparent group to world coordinates.) Then the combined matrix product is:

$$\mathbf{M}_{\text{combined}} = \mathbf{M_3}\mathbf{M_2}\mathbf{M_1}$$

Note that the order of the matrix factors is important. Also note that this matrix product should be read from the right (where vectors are multiplied) to the left, i.e. $\mathbf{M_1}$ is applied first while $\mathbf{M_3}$ is applied last.

Viewing Transformation

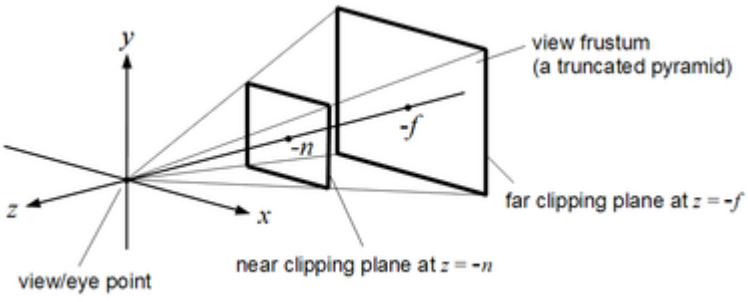


Illustration of the view coordinate system.

The viewing transformation corresponds to placing and orienting the camera (or the eye of an observer). However, the best way to think of the viewing transformation is that it transforms the world coordinates into the view coordinate system (also: eye coordinate system) of a camera that is placed at the origin of the coordinate system, points to the **negative** z axis and is put on the xz plane, i.e. the up-direction is given by the positive y axis.

This step rotates the entire world towards the camera, which is always looking at a fixed position from the origin.

Accessing the View Matrix in a Vertex Shader

Similarly to the modeling transformation, the viewing transformation is represented by a 4×4 matrix, which is called view matrix $\mathbf{M}_{\text{world}\to\text{view}}$. It can be defined as a uniform variable for the vertex shader; however, it is usually combined with the model matrix $\mathbf{M}_{\text{object}\to\text{world}}$ to form the modelview matrix $\mathbf{M}_{\text{object}\to\text{view}}$. (In some versions of OpenGL (ES), a built-in uniform variable `gl_ModelViewMatrix` is available in the vertex shader.) Since the model matrix is applied first, the correct combination is:

$$\mathbf{M}_{\text{object}\to\text{view}} = \mathbf{M}_{\text{world}\to\text{view}}\mathbf{M}_{\text{object}\to\text{world}}$$

(See also Section “Applying Matrix Transformations”.)

Computing the View Matrix

Analogously to the model matrix, GLSL programmers don't have to worry about the computation of the view matrix since it is provided to the vertex shader in the form of a uniform variable. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, it is necessary to compute the view matrix. (In older versions of OpenGL this is usually achieved by a utility function called `gluLookAt`.)

Here, we briefly summarize how the view matrix $\mathbf{M}_{\text{world}\to\text{view}}$ can be computed from the position **t** of the camera, the view direction **d**, and a world-up vector **k** (all in world coordinates). The steps are straightforward:

1. Compute (in world coordinates) the direction **z** of the z axis of the view coordinate system as the negative normalized **d** vector:

$$\mathbf{z} = -\frac{\mathbf{d}}{|\mathbf{d}|}$$

2. Compute (again in world coordinates) the direction **x** of the x axis of the view coordinate system by:

$$\mathbf{x} = \frac{\mathbf{d}\times\mathbf{k}}{|\mathbf{d}\times\mathbf{k}|}$$

3. Compute (still in world coordinates) the direction **y** of the y axis of the view coordinate system:

$$\mathbf{y} = \mathbf{z}\times\mathbf{x}$$

Using **x**, **y**, **z**, and **t**, the inverse view matrix $\mathbf{M}_{\text{view}\to\text{world}}$ can be easily determined because this matrix maps the origin (0,0,0) to **t** and the unit vectors (1,0,0), (0,1,0) and (0,0,1) to **x**, **y**,, **z**. Thus, the latter vectors have to be in the columns of the matrix $\mathbf{M}_{\text{view}\to\text{world}}$:

$$\mathbf{M}_{\text{view}\to\text{world}} = \begin{bmatrix} x_1 & y_1 & z_1 & t_1 \\ x_2 & y_2 & z_2 & t_2 \\ x_3 & y_3 & z_3 & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

However, we require the matrix $\mathbf{M}_{\text{world}\to\text{view}}$; thus, we have to compute the inverse of the matrix $\mathbf{M}_{\text{view}\to\text{world}}$. Note that the matrix $\mathbf{M}_{\text{view}\to\text{world}}$ has the form

$$\mathbf{M}_{\text{view}\to\text{world}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

with a 3×3 matrix $\mathbf{R}$ and a 3D vector $\mathbf{t}$. The inverse of such a matrix is:

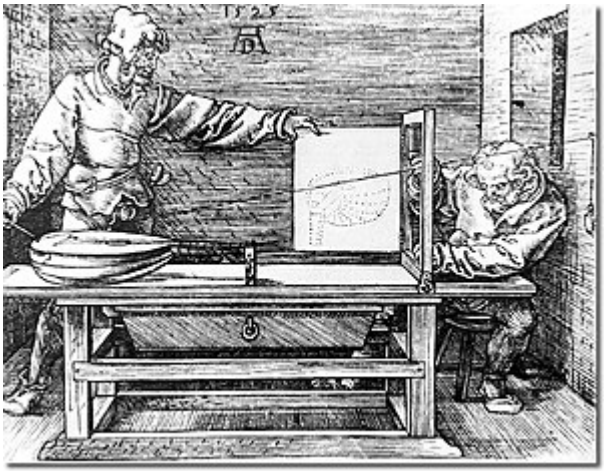
$$\mathbf{M}_{\text{view}\to\text{world}}^{-1} = \mathbf{M}_{\text{world}\to\text{view}} = \begin{bmatrix} \mathbf{R}^{-1} & -\mathbf{R}^{-1}\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

Since in this particular case the matrix $\mathbf{R}$ is orthogonal (because its column vectors are normalized and orthogonal to each other), the inverse of $\mathbf{R}$ is just the transpose, i.e. the fourth step is to compute:

$$\mathbf{M}_{\text{world}\to\text{view}} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \text{ with } \mathrm{R} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix}$$

While the derivation of this result required some knowledge of linear algebra, the resulting computation only requires basic vector and matrix operations and can be easily programmed in any common programming language.

Projection Transformation and Perspective Division



Perspective drawing in the Renaissance: “Man drawing a lute” by Albrecht Dürer, 1525

First of all, the projection transformations determine the kind of projection, e.g. perspective or orthographic. Perspective projection corresponds to linear perspective with foreshortening, while orthographic projection is an orthogonal projection without foreshortening. The foreshortening is actually accomplished by the perspective division; however, all the parameters controlling the perspective projection are set in the projection transformation.

Technically spoken, the projection transformation transforms view coordinates to clip coordinates. (All parts of primitives that are outside the visible part of the scene are clipped away in clip coordinates.) It should be the last transformation that is applied to a vertex in a vertex shader before the vertex is returned in `gl_Position`. These clip coordinates are then transformed to normalized device coordinates by the **perspective division**, which is just a division of all coordinates by the fourth coordinate. (Normalized device coordinates are named as such because their values are between -1 and +1 for all points in the visible part of the scene.)

This step translates the 3d positions of object vertices to 2d positions on the screen.

Accessing the Projection Matrix in a Vertex Shader

Similarly to the modeling transformation and the viewing transformation, the projection transformation is represented by a 4×4 matrix, which is called projection matrix $\mathbf{M}_{\text{projection}}$. It is usually defined as a uniform variable for the vertex shader. (In some versions of OpenGL (ES), a built-in uniform variable `gl_Projection` is available in the vertex shader; see also Section “Applying Matrix Transformations”.)

Computing the Projection Matrix

Analogously to the modelview matrix, GLSL programmers don't have to worry about the computation of the projection matrix. However, when developing applications in modern versions of OpenGL and OpenGL ES or in WebGL, it is necessary to compute the projection matrix. In older versions of OpenGL this is usually achieved with the functions `gluPerspective`, `glFrustum`, or `glOrtho`.

Here, we present the projection matrices for three cases:

- standard perspective projection (corresponds to `gluPerspective`)
- oblique perspective projection (corresponds to `glFrustum`)
- orthographic projection (corresponds to `glOrtho`)

The **standard perspective projection** is characterized by

- an angle θ_{fovy} that specifies the field of view in y direction as illustrated in the figure to the right,
- the distance n to the near clipping plane and the distance f to the far clipping plane as illustrated in the next figure,
- the aspect ratio a of the width to the height of a centered rectangle on the near clipping plane.

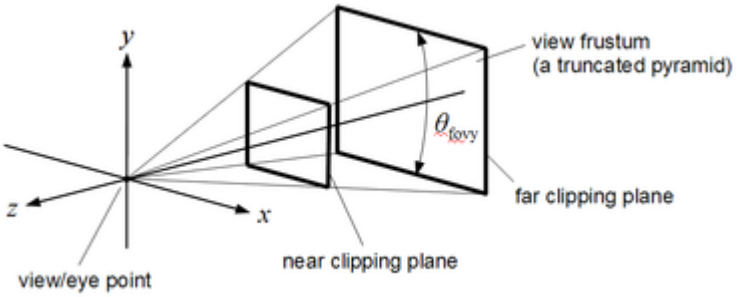


Illustration of the angle θ_{fovy} specifying the field of view in *y* direction.

Together with the view point and the clipping planes, this centered rectangle defines the view frustum, i.e. the region of the 3D space that is visible for the specific projection transformation. All primitives and all parts of primitives that are outside of the view frustum are clipped away. The near and front clipping planes are necessary because depth values are stored with a finite precision; thus, it is not possible to cover an infinitely large view frustum.

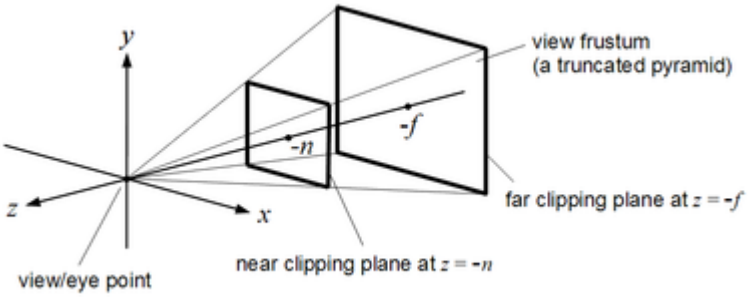


Illustration of the near and far clipping planes at $z = -n$ and $z = -f$

With the parameters θ_{fovy}, a, n, and f, the projection matrix $\mathbf{M}_{\text{projection}}$ for the perspective projection is:

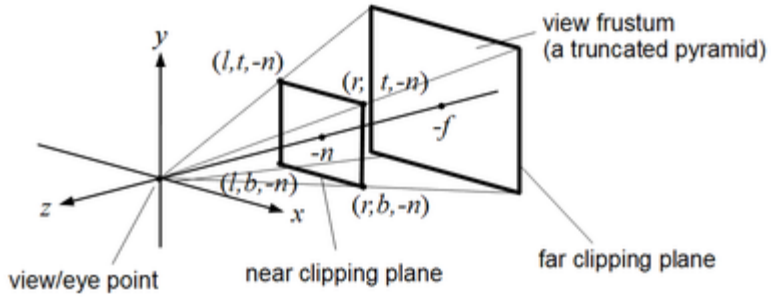
$$\mathbf{M}_{\text{projection}} = \begin{bmatrix} \frac{d}{a} & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{with } d = \frac{1}{\tan(\theta_{\text{fovy}}/2)}$$

The **oblique perspective projection** is characterized by

- the same distances n and f to the clipping planes as in the case of the standard perspective projection,
- coordinates r (right), l (left), t (top), and b (bottom) as illustrated in the corresponding figure. These coordinates determine the position of the front rectangle of the view frustum; thus, more

view frustums (e.g. off-center) can be specified than with the aspect ratio a and the field-of-view angle θ_{fovy}.

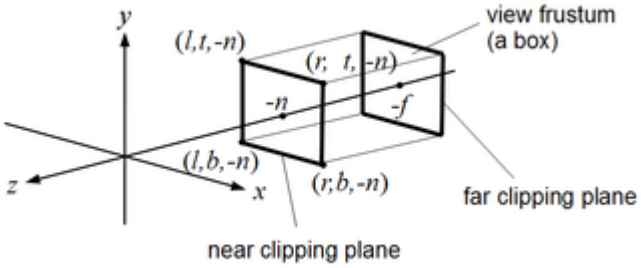
Given the parameters n, f, r, l, t, and b, the projection matrix $\mathbf{M}_{\text{projection}}$ for the oblique perspective projection is:



Parameters for the oblique perspective projection.

$$\mathbf{M}_{\text{projection}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

An **orthographic projection** without foreshortening is illustrated in the figure to the right. The parameters are the same as in the case of the oblique perspective projection; however, the view frustum (more precisely, the view volume) is now simply a box instead of a truncated pyramid.



Parameters for the orthographic projection.

With the parameters n, f, r, l, t, and b, the projection matrix $\mathbf{M}_{\text{projection}}$ for the orthographic projection is:

$$\mathbf{M}_{\text{projection}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewport Transformation

The projection transformation maps view coordinates to clip coordinates, which are then mapped to normalized device coordinates by the perspective division by the fourth component of the clip coordinates. In normalized device coordinates (ndc), the view volume is always a box centered around the origin with the coordinates inside the box between -1 and +1. This box is then mapped to screen coordinates (also called

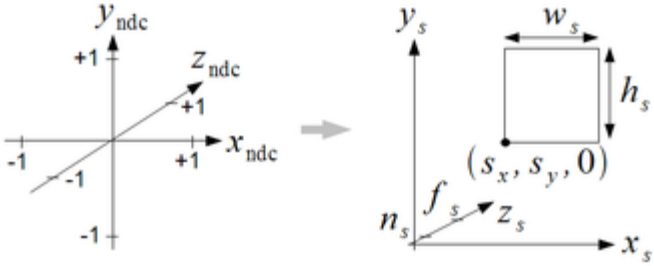


Illustration of the viewport transformation.

window coordinates) by the viewport transformation as illustrated in the corresponding figure. The parameters for this mapping are the coordinates s_x and s_y of the lower, left corner of the viewport (the rectangle of the screen that is rendered) and its width w_s and height h_s, as well as the depths n_s and f_s of the front and near clipping planes. (These depths are between 0 and 1). In OpenGL and OpenGL ES, these parameters are set with two functions:

`glViewport(GLint` s_x`, GLint` s_y`, GLsizei` w_s`, GLsizei` h_s`);`

`glDepthRangef(GLclampf` n_s`, GLclampf` f_s`);`

The matrix of the viewport transformation isn't very important since it is applied automatically in a fixed-function stage. However, here it is for the sake of completeness:

$$\mathrm{M}_{\mathrm{viewport}} = \begin{bmatrix} \frac{w_s}{2} & 0 & 0 & s_x + \frac{w_s}{2} \\ 0 & \frac{h_s}{2} & 0 & s_y + \frac{h_s}{2} \\ 0 & 0 & \frac{f_s - n_s}{2} & \frac{n_s + f_s}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Further Reading

The conventional vertex transformations described here are defined in full detail in Section 2.12 of the "OpenGL 4.1 Compatibility Profile Specification" available at the Khronos OpenGL web site (http://www.khronos.org/opengl/).

A more accessible description of the vertex transformations is given in Chapter 3 (on viewing) of the book "OpenGL Programming Guide" by Dave Shreiner published by Addison-Wesley. (An older edition is available online (http://www.glprogramming.com/red/chapter03.html)).

< GLSL Programming