第7章 座標変換

7.1 頂点処理

7.1.1 図形表示の手順

OpenGL による図形表示では、正規化デバイス座標系上に設定されているクリッピング空間からはみ出た部分はクリッピングされ、画面には表示されません。したがって、任意の大きさの図形を表示するには、その図形をクリッピング空間に収まるように拡大縮小と平行移動を行う必要があります。第 6 章では、描画しようとする図形が単純な二次元図形であることから、これを場当たり的な方法で行っていました。

しかし、描画しようとする図形が複雑な構造を持っていたりすると、このようなアプローチではすぐに訳が分からなくなってしまいそうです。それに OpenGL は、本来は三次元図形の表示を行う API です。三次元の図形を二次元の空間である画面上に表示するには、さらに投影という処理も必要になります。

この投影を含めた三次元図形表示については、既に標準的な手順が考えられています。一般にグラフィックス表示の対象となる形状データは、複数の「部品」で構成されています。これを空間中に「配置」することによって、シーンを構成します (図 78)。この「部品の配置」のための座標変換をモデル変換と言います。次に、部品が配置されたシーンをカメラの位置から見たときの座標に変換します。これをビュー変換と言います。そして最後に投影変換を行って、シーンをスクリーンの正規化デバイス座標系上に投影します。

このように、座標変換はモデル変換→ビュー変換→投影変換の順に行われるので、これも一つの論理的なパイプラインと見なすことができます。このため、これをジオメトリパイプラインと呼ぶことがあります。かつては、この処理をジオメトリエンジンと名付けられた専用のハードウェアで行っていました (さらにその前は CPU によるソフトウェア処理でした)。

現在の GPU では、ジオメトリパイプラインをバーテックスシェーダ上にシェーダプログラムとして実装します (図 79)。バーテックスシェーダは頂点バッファに格納されている頂点属性を受け取り、これらの座標変換により正規化座標系上の座標値を求めて、それを次のステージに送ります。

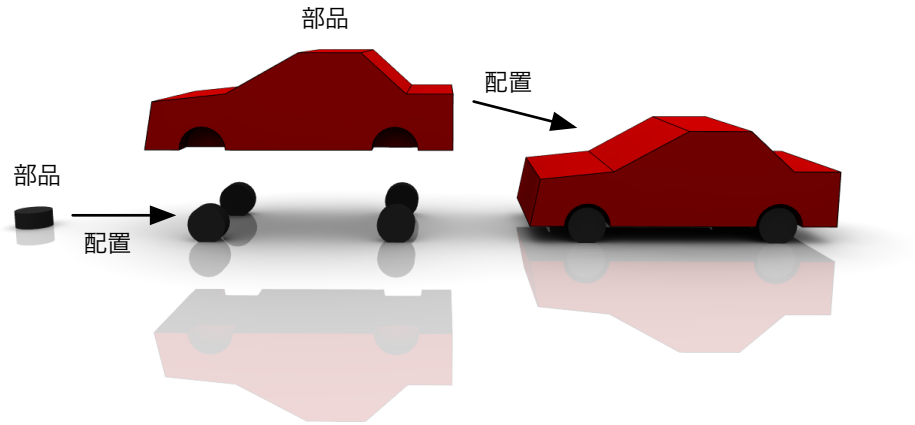

図 78 部品の配置

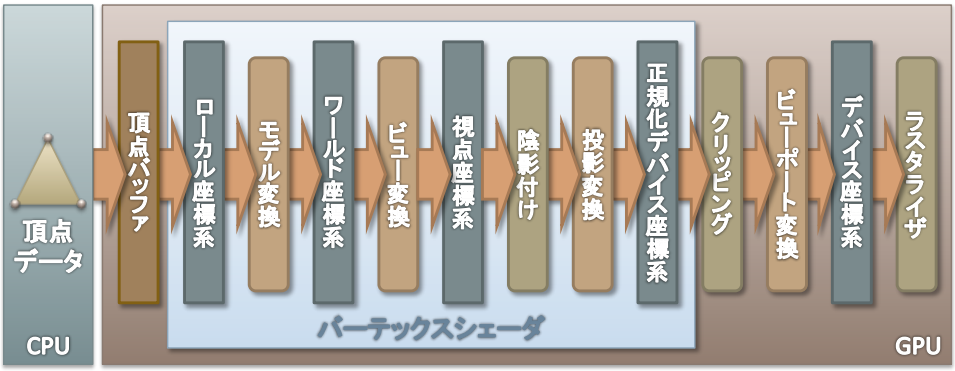

図 79 頂点処理のパイプライン

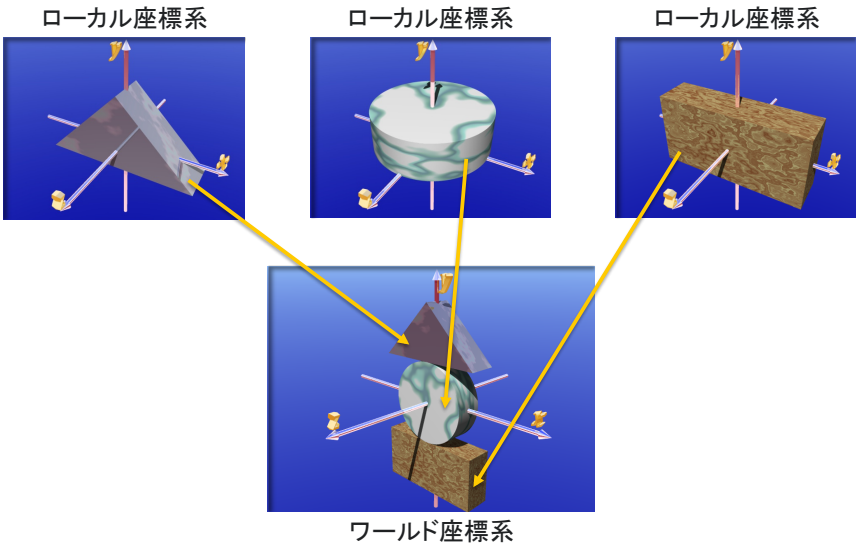

図 80 モデル変換

7.1.2　モデル変換

シーンを構成する部品 (オブジェクトと呼ばれます) は、個々に独自の座標系で定義されています。この座標系をローカル座標系といいます。シーンを構成するには、この部品のほか視点や光源などを、ワールド座標系と呼ばれる単一の座標系に配置します。このために行われるローカル座標系からワールド座標系への座標変換を、モデル変換といいます (図 80)。

モデル変換は部品ごとに設定します。部品間に骨格のような階層構造があれば、この変換も階層的に合成して実行します。モデル変換後は、すべての部品がワールド座標系上に存在します。

7.1.3　ビュー変換

ワールド座標系で構築されたシーンの視点位置から見た映像を作成するために、視点を基準とする視点座標系にシーン全体を座標変換します。このために行われるワールド座標系から視点座標系への変換をビュー変換といいます (図 81)。なお、モデル変換とビュー変換は通常ひとつの座標変換に合成されます。この合成変換をモデルビュー変換といいます。

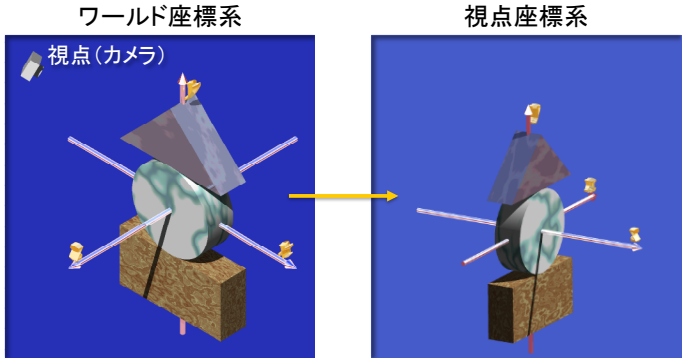

図 81 ビュー変換

ローカル座標系、ワールド座標系、視点座標系、およびスクリーンの関係を、図 82 に示します。

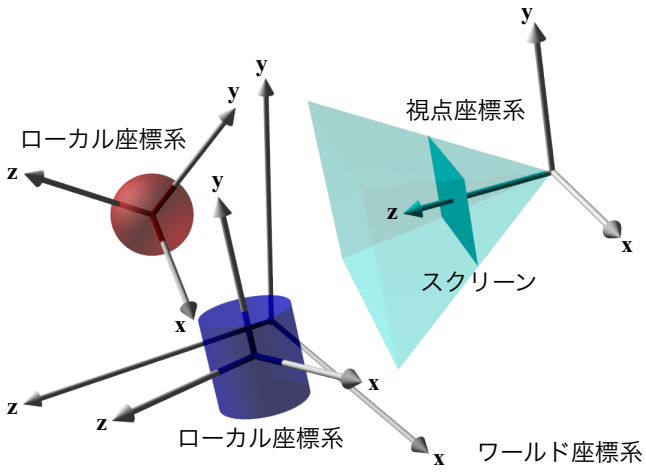

図 82 各座標系の関係

7.1.4　投影変換

視点座標系に配置されたシーンをスクリーンに投影する変換を投影変換といいます。ディスプレイの表示領域は有限ですから、そこに映るシーンの空間も限定されます。この空間をビューボリューム (View Volume) といいます。投影変換は、この空間を、中心が原点にあり一辺の長さが 2 の立方体の空間である標準ビューボリューム (Canonical View Volume) に変形します (図 83)。これは三次元のクリッピング領域です。

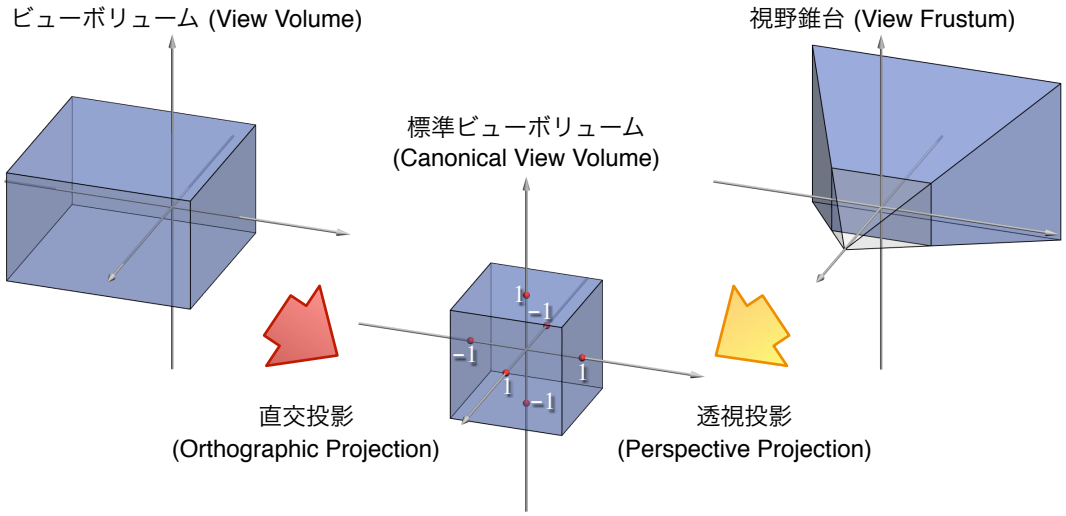

図 83 標準ビューボリューム

標準ビューボリュームへの変換は、変換前のビューボリュームに直方体を用いる場合と、四角錐台を用いる場合の二通りがあります。直方体を用いれば直交投影 (Orthogonal Projection) となります。一方、四角錐台を用いれば透視投影 (Perspective Projection) となります (図 19)。なお、この四角錐台のビューボリュームは、特にビューフラスタム (View Frustum, 視野錐台あるいは視錐台) と呼ばれます。

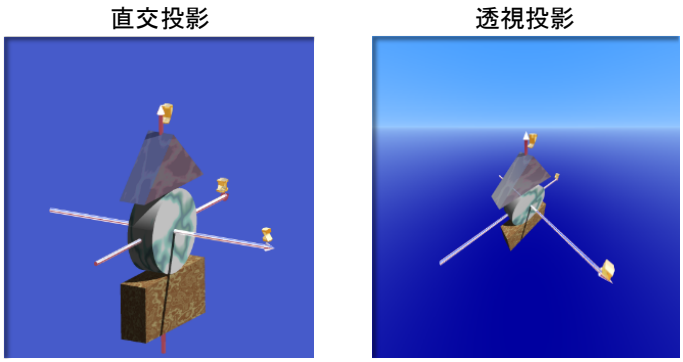

図 84 直交投影と透視投影

7.2 同次座標

7.2.1 アフィン変換

第 6 章で行っていた拡大縮小などの線形変換と平行移動の組み合わせは、一般にアフィン変換と呼ばれます。一次元のアフィン変換は次のように表すことができます。

$$x' = ax + b \tag{1}$$

これは x を a 倍して、b だけ平行移動します。同様に、二次元、三次元の場合は、それぞれ次のようになります。

$$\begin{aligned} x' &= a_{xx}x + a_{yx}y + b_x \\ y' &= a_{xy}x + a_{yy}y + b_y \end{aligned} \tag{2}$$

$$\begin{aligned} x' &= a_{xx}x + a_{yx}y + a_{zx}z + b_x \\ y' &= a_{xy}x + a_{yy}y + a_{zy}z + b_y \\ z' &= a_{xz}x + a_{yz}y + a_{zz}z + b_z \end{aligned} \tag{3}$$

行列を使うと、(3) 式を次のように書くことができます。

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} \\ a_{xy} & a_{yy} & a_{zy} \\ a_{xz} & a_{yz} & a_{zz} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} \tag{4}$$

7.2.2 同次座標の導入

アフィン変換は (4) 式に示すように、線形変換の結果を平行移動したものとして表すことができます。しかし、積と和の組み合わせでは、複数のアフィン変換の合成変換を表すことが面倒になります。そこで、x, y, z にもう一つ数を追加して、次のように表すことを考えてみます。

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{xx} & a_{yx} & a_{zx} & b_x \\ a_{xy} & a_{yy} & a_{zy} & b_y \\ a_{xz} & a_{yz} & a_{zz} & b_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5}$$

この x', y', z' も (3) 式と同じものになります。このように表せば、変換の合成を行列の積のみで表すことができます。

同次座標 (斉次座標) は、(5) 式のように、表現しようとする空間の次元より一つ多い数の組で座標 (点の位置) を表すものです。二次元なら三つの数、三次元なら四つの数で座標を表します。通常の三次元の座標 (以降は実座標と呼ぶことにします) を (x^*, y^*, z^*) と表すとき、それと、その同次座標 (x, y, z, w) との間には、同次座標の定義により次の関係があります。

$$\begin{aligned} x^* &= \frac{x}{w} \\ y^* &= \frac{y}{w} \\ z^* &= \frac{z}{w} \end{aligned} \tag{6}$$

したがって、実座標が (x, y, z) であれば、その同次座標は $(x, y, z, 1)$ となります。また、同次座標の四つ目の要素 w が 0 に近づけば、その実座標は (x, y, z) 方向の無限遠に向かって移動します。CG では、同次座標 $(x, y, z, 0)$ は (x, y, z) の方向を表すものとして用いられます。

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow (x, y, z) \text{の位置} \tag{7}$$

$$\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \Rightarrow (x, y, z) \text{の方向} \tag{8}$$

同次座標にスカラー値を掛けても、実座標は変わりません。

$$a \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} ax \\ ay \\ az \\ aw \end{pmatrix} \Rightarrow \left(\frac{ax}{aw}, \frac{ay}{aw}, \frac{az}{aw}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \tag{9}$$

CG では、ある点から別の点に向かう方向単位ベクトルを求めなければならないことがよくあります。いま、二点 $\mathbf{P}_0, \mathbf{P}_1$ の位置をそれぞれ同次座標 $\mathbf{P}_0 = (x_0, y_0, z_0, w_0), \mathbf{P}_1 = (x_1, y_1, z_1, w_1)$ で表すとき、$\mathbf{P}_0$ から $\mathbf{P}_1$ に向かうベクトルは、それぞれの実座標を求めて差を求めます。

$$\frac{\mathbf{P}_1}{w_1} - \frac{\mathbf{P}_0}{w_0} = \begin{pmatrix} x_1/w_1 \\ y_1/w_1 \\ z_1/w_1 \\ 1 \end{pmatrix} - \begin{pmatrix} x_0/w_0 \\ y_0/w_0 \\ z_0/w_0 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1/w_1 - x_0/w_0 \\ y_1/w_1 - y_0/w_0 \\ z_1/w_1 - z_0/w_0 \\ 0 \end{pmatrix} \tag{10}$$

これだと w_0 か w_1 のどちらかが 0 の時に計算できないので、この両辺に $w_0 w_1$ を掛けます。

$$w_0\mathbf{P}_1 - w_1\mathbf{P}_0 = \begin{pmatrix} w_0x_1 \\ w_0y_1 \\ w_0z_1 \\ w_0w_1 \end{pmatrix} - \begin{pmatrix} w_1x_0 \\ w_1y_0 \\ w_1z_0 \\ w_1w_0 \end{pmatrix} = \begin{pmatrix} w_0x_1 - w_1x_0 \\ w_0y_1 - w_1y_0 \\ w_0z_1 - w_1z_0 \\ 0 \end{pmatrix} \tag{11}$$

(11) 式は引き算により四つ目の要素が 0 になっているため、実座標ではなく方向のみを表します。しかし、この計算は $\mathbf{P}_0$ と $\mathbf{P}_1$ のどちらか一方の四つ目の要素が 0、すなわち無限遠にある場合も場合分けすること無く計算できます。また、この計算はこの時点まで割り算を必要としません。照明計算などで (11) 式の結果を方向として使う場合は、これを正規化します。実座標が必要なら、これを $w_0 w_1$ で割ります。

なお、同次座標の和は実座標の平均になります。

$$\sum_{i=1}^{N} \frac{\mathbf{P}_i}{w_i} = \sum_{i=1}^{N} \begin{pmatrix} x_i/w_i \\ y_i/w_i \\ z_i/w_i \\ 1 \end{pmatrix} = \begin{pmatrix} \sum x_i/w_i \\ \sum y_i/w_i \\ \sum z_i/w_i \\ N \end{pmatrix} \Rightarrow \left(\frac{\sum x_i}{N}, \frac{\sum y_i}{N}, \frac{\sum z_i}{N}\right) \tag{12}$$

7.3 変換行列

7.3.1 同次座標の座標変換

式 (5) のアフィン変換では、ベクトルや行列の一部に 0 や 1 の定数が入っていました。ここではより一般的な同次座標による座標変換について考えてみます。同次座標で表された点の位置 $\mathbf{v}$ の変換行列 $\mathbf{M}$ による座標変換 $\mathbf{v}' = \mathbf{M}\mathbf{v}$ は、式 (15) のように表すことができます。

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{13}$$

$$\mathbf{v}' = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \tag{14}$$

$$\mathbf{M} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \tag{15}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{16}$$

このベクトル $\mathbf{v}'$ の各要素 (x', y', z', w') は次式により求められます。

$$\begin{array}{lcccccccc} x' & = & m_0\,x & + & m_4\,y & + & m_8\,z & + & m_{12}\,w \\ y' & = & m_1\,x & + & m_5\,y & + & m_9\,z & + & m_{13}\,w \\ z' & = & m_2\,x & + & m_6\,y & + & m_{10}\,z & + & m_{14}\,w \\ w' & = & m_3\,x & + & m_7\,y & + & m_{11}\,z & + & m_{15}\,w \end{array} \tag{17}$$

したがって、ベクトル $\mathbf{v}'$ の各要素は、行列 $\mathbf{M}$ の各行を要素とするベクトルと $\mathbf{v}$ の内積になります。たとえば、$x' = (m_0\ m_4\ m_8\ m_{12}) \cdot (x\ y\ z\ w)^T$ です ((17) 式)。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{18}$$

● OpenGL の変換行列

式 (16) に示した 4 行 4 列の変換行列は OpenGL の座標変換の基本となるもので、GPU で座標変換を行うために、CPU 側から GPU 側に頻繁に渡されます。ただし、行列を配列変数に格納する際には、配列の要素の順序は行列の要素の順序を**見かけ上**転置したものになります。

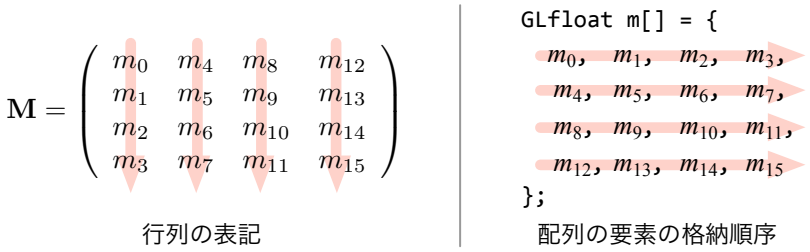

図 85 行列の表記と配列の要素の格納順序

7.3.2 平行移動

点の位置を、現在の位置から $\mathbf{t} = (t_x, t_y, t_z)$ 離れたところに平行移動する変換行列 $\mathbf{T}(t_x, t_y, t_z)$ は、次のようになります。

$$\mathbf{T}(\mathbf{t}) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19}$$

実座標 (x, y, z) に対するこの変換行列による変換は、次のようになります。

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix} \tag{20}$$

$\mathbf{T}(7, 8, 0)$ という変換は、図 86 のような平行移動になります。

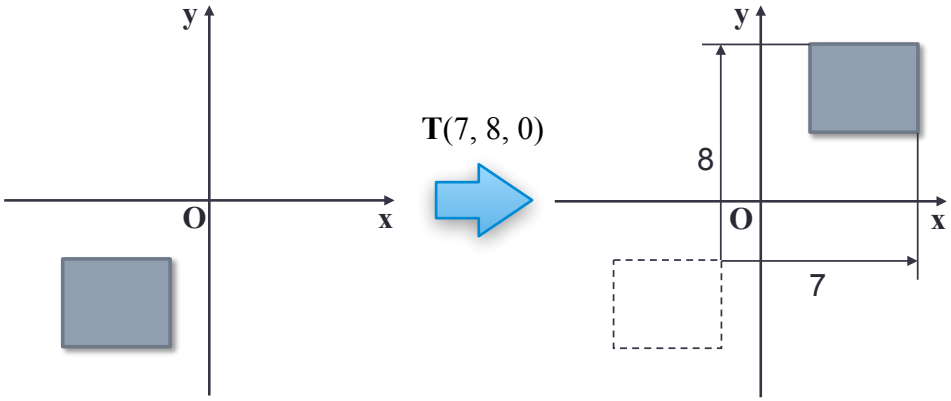

図 86 平行移動

同次座標の四つ目の要素が 0 の時は、この変換の影響を受けません。

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} \tag{20}$$

● 平行移動の変換行列を設定する関数

配列変数 m に平行移動の変換行列を設定する関数は、たとえば次のようになります。

```
// (x, y, z) 平行移動する変換行列を m に設定する
GLfloat *loadTranslate(GLfloat x, GLfloat y, GLfloat z, GLfloat *m)
{
  m[12] = x;
  m[13] = y;
  m[14] = z;
  m[ 1] = m[ 2] = m[ 3] =
  m[ 4] = m[ 6] = m[ 7] =
  m[ 8] = m[ 9] = m[11] = 0.0f;
  m[ 0] = m[ 5] = m[10] = m[15] = 1.0f;
  return m;
}
```

この関数は、引数 x, y, z で与えられた位置に平行移動する変換行列を、引数 m に与えられた 16 要素の配列変数に格納します。平行移動の変換行列は、式 (20) より式 (15) の変換行列の対角要素 m_0、m_5、m_{10}、m_{15} に 1、m_{12} に x、m_{13} に y、m_{14} に z を代入したものになります。なお、この関数は戻り値として、引数の配列変数 m のポインタを返します。

● **単位行列を設定する関数**

平行移動、あるいは次に述べる拡大縮小の変換行列を設定する関数を使って、単位行列を設定する関数を作ることができます。平行移動の関数を使うなら、移動量を 0 にします。

```
// 単位行列を m に設定する
GLfloat *loadIdentity(GLfloat *m)
{
  return loadTranslate(0.0f, 0.0f, 0.0f, m);
}
```

7.3.3 拡大縮小

点の位置を、原点を中心に $\mathbf{s} = (s_x, s_y, s_z)$ 倍する変換行列 $\mathbf{S}(s_x, s_y, s_z)$ は、次のようになります。

$$\mathbf{S}(\mathbf{s}) = \mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{21}$$

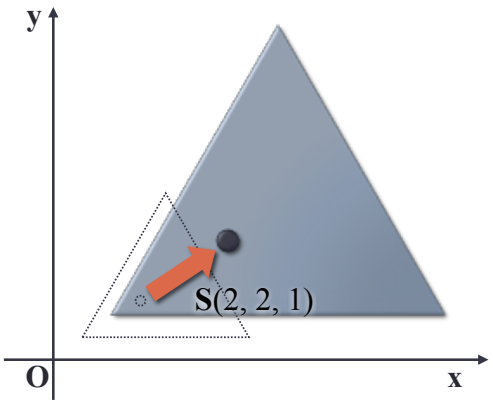

図 87 拡大縮小

拡大率を $s_x = s_y = s_z = a$ とした場合と w 要素の拡大率を $1/a$ とした場合とでは、実座標は等

しくなります。

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} ax \\ ay \\ az \\ 1 \end{pmatrix} \Rightarrow (ax,\ ay,\ az) \tag{22}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/a \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1/a \end{pmatrix} \Rightarrow (ax,\ ay,\ az) \tag{23}$$

● **拡大縮小の変換行列を設定する関数**

平行移動と同様に、拡大縮小の変換行列を次のようにして作ることができます。

```
// (x, y, z) 拡大縮小する変換行列を m に設定する
GLfloat *loadScale(GLfloat x, GLfloat y, GLfloat z, GLfloat *m)
{
  m[ 0] = x;
  m[ 5] = y;
  m[10] = z;
  m[ 1] = m[ 2] = m[ 3] =
  m[ 4] = m[ 6] = m[ 7] =
  m[ 8] = m[ 9] = m[11] =
  m[12] = m[13] = m[14] = 0.0f;
  m[15] = 1.0f;
  return m;
}
```

この関数は、引数 x, y, z で与えられた割合で x 軸方向、y 軸方向、z 軸方向に拡大縮小する変換行列を、引数 m に与えられた 16 要素の配列変数に格納します。拡大縮小の変換行列は、式 (23) より式 (15) の変換行列の対角要素の m_0 に x、m_5 に y、m_{10} に z、m_{15} に 1 を代入したものになります。この関数も戻り値として、引数の配列変数 m のポインタを返します。

7.3.4 せん断

せん断は形状の異なる部分に異なる方向に力をかけた時に生じる変形のことを言います。

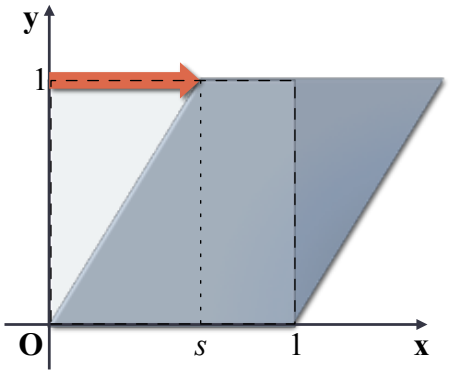

図 88 せん断

これは座標値のある軸の値に係数をかけたものを別の軸に加えることによって実現できます。

図 88 の場合は Y 座標値に s をかけたものを X 座標値に足しています。これは (24) 式で表されます。

$$\mathbf{H}_{xy}(s) = \begin{pmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{24}$$

この変換は x, y, z の各軸に対してそれぞれ二つずつ定義されるので、全部で六つあります。

$$\mathbf{H}_{xy}(s) = \begin{pmatrix} 1 & s & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{yz}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & s & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{zx}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{H}_{yx}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ s & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{zy}(s) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & s & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{H}_{xz}(s) = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(25)

7.3.5 回転

次の変換は、点の位置を、原点を中心に回転します。

- X 軸中心の回転

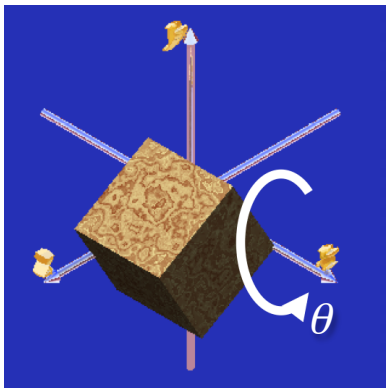

図 89 X 軸中心の回転

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{21}$$

● Y 軸中心の回転

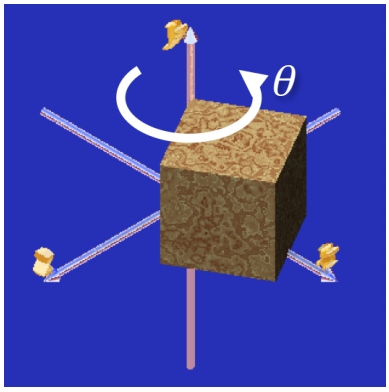

図 90 Y 軸中心の回転

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{22}$$

● Z 軸中心の回転

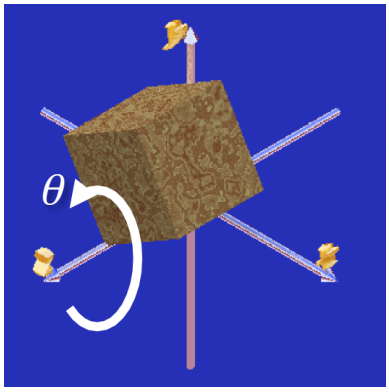

図 91 Z 軸中心の回転

$$\mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{23}$$

● 任意の軸中心の回転

方向余弦 (l, m, n) を軸として θ 回転する変換行列は、次式により求めることができます。

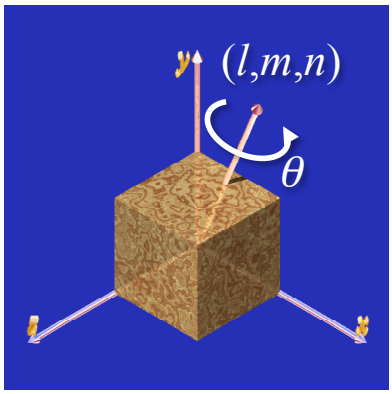

図 92 任意の軸中心の回転

$$\mathbf{R}(l,m,n,\theta) = \begin{pmatrix} l^2+(1-l^2)\cos\theta & lm(1-\cos\theta)-n\sin\theta & ln(1-\cos\theta)+m\sin\theta & 0 \\ lm(1-\cos\theta)+n\sin\theta & m^2+(1-m^2)\cos\theta & mn(1-\cos\theta)-l\sin\theta & 0 \\ ln(1-\cos\theta)-m\sin\theta & mn(1-\cos\theta)+l\sin\theta & n^2+(1-n^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (24)$$

● 任意の軸中心の回転を設定する関数

式 (24) の変換行列を求める関数の例を以下に示します。この関数は引数 x, y, z で与えられたベクトルを軸に引数 a だけ回転する変換行列を m に設定します。

```
// (x, y, z) を軸に a 回転する変換行列を m に設定する
GLfloat *loadRotate(GLfloat a, GLfloat x, GLfloat y, GLfloat z, GLfloat *m)
{
  const GLfloat d(sqrt(x * x + y * y + z * z));

  if (d > 0.0f)
  {
    const GLfloat l(x / d), m(y / d), n(z / d);
    const GLfloat l2(l * l), m2(m * m), n2(n * n);
    const GLfloat lm(l * m), mn(m * n), nl(n * l);
    const GLfloat c(cos(a)), c1(1.0f - c), s(sin(a));

    m[ 0] = (1.0f - l2) * c + l2;
    m[ 1] = lm * c1 + n * s;
    m[ 2] = nl * c1 - m * s;

    m[ 4] = lm * c1 - n * s;
    m[ 5] = (1.0f - m2) * c + m2;
    m[ 6] = mn * c1 + l * s;

    m[ 8] = nl * c1 + m * s;
    m[ 9] = mn * c1 - l * s;
    m[10] = (1.0f - n2) * c + n2;

    m[ 3] = m[ 7] = m[11] = m[12] = m[13] = m[14] = 0.0f;
    m[15] = 1.0f;
  }

  return m;
}
```

7.4 直交座標系の変換

互いに直交する三つの単位ベクトル **i**', **j**', **k**' を軸とする座標系上の点の位置 **p** を、この座標系と原点を共有する、別の直交する三つの単位ベクトル **i**, **j**, **k** を軸とする座標系に移す変換を求めます。この変換は回転の変換になります。それぞれの座標系における **p** の座標値を (*x*', *y*', *z*') と (*x*, *y*, *z*) とするとき、**p** は次のように表すことができます。

$$\mathbf{p} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} = x'\mathbf{i}' + y'\mathbf{j}' + z'\mathbf{k}' \tag{30}$$

この式は、行列を用いて次のように書くことができます。

$$\begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{31}$$

したがって (*x*, *y*, *z*) は、次式で求めることができます。

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{32}$$

ここで行列 **(i j k)**, **(i' j' k')** は直交行列ですから、これらの逆行列は転置行列と等しくなります。

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{T} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \tag{33}$$

しがたがって、次の変換行列 **M** は (*x*', *y*', *z*') を (*x*, *y*, *z*) に回転します。

$$\mathbf{M} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{pmatrix}^{T} \begin{pmatrix} \mathbf{i}' & \mathbf{j}' & \mathbf{k}' \end{pmatrix} = \begin{pmatrix} \mathbf{i}\cdot\mathbf{i}' & \mathbf{i}\cdot\mathbf{j}' & \mathbf{i}\cdot\mathbf{k}' \\ \mathbf{j}\cdot\mathbf{i}' & \mathbf{j}\cdot\mathbf{j}' & \mathbf{j}\cdot\mathbf{k}' \\ \mathbf{k}\cdot\mathbf{i}' & \mathbf{k}\cdot\mathbf{j}' & \mathbf{k}\cdot\mathbf{k}' \end{pmatrix} \tag{34}$$

ここで、この M について別の (ちょっとくどい) 説明をします。いま、互いに直交する三つの単位ベクトルを **r**, **s**, **t** とします。

$$\mathbf{r} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix},\ \mathbf{s} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix},\ \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{35}$$

また、X 軸、Y 軸、Z 軸のそれぞれの軸方向の単位ベクトル **x**, **y**, **z** は次のように表されます。

$$\mathbf{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix},\ \mathbf{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix},\ \mathbf{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{36}$$

r, **s**, **t** を **x**, **y**, **z** に一致させる回転の変換行列を **M** とします。

$$\begin{cases} \mathbf{x} = \mathbf{Mr} \\ \mathbf{y} = \mathbf{Ms} \\ \mathbf{z} = \mathbf{Mt} \end{cases} \tag{37}$$

これを行列で表すと、次のようになります。

$$\begin{pmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{r} & \mathbf{s} & \mathbf{t} \end{pmatrix} \tag{38}$$

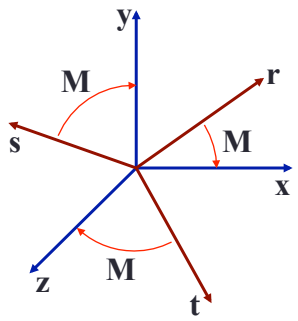

図 93 座標軸の回転

この左辺は単位行列になります。

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \mathbf{M} \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix} \tag{39}$$

したがって、この場合の $\mathbf{M}$ は次のようにして求めることができます。

$$\mathbf{M} = \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix}^{-1} = \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix}^{T} = \begin{pmatrix} r_x & r_y & r_z \\ s_x & s_y & s_z \\ t_x & t_y & t_z \end{pmatrix} = \begin{pmatrix} \mathbf{r}^T \\ \mathbf{s}^T \\ \mathbf{t}^T \end{pmatrix} \tag{40}$$

7.5 変換の合成

7.5.1 複数の変換の組み合わせ

複数の変換の合成は、変換行列の積で表すことができます。たとえば、図 94 のように x 方向に 4 移動した後、y 方向に 3 移動する変換は、(41) 式の単一の変換行列になります。

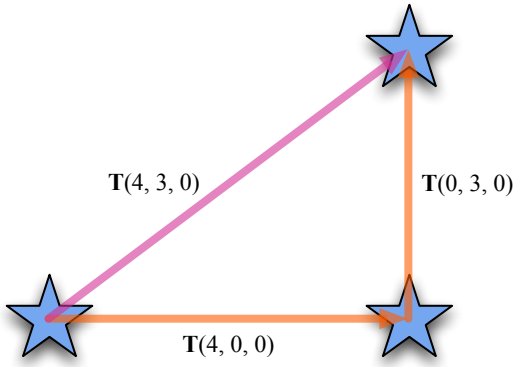

図 94 変換の合成

座標変換がこの程度の平行移動だけなら、座標値に直接移動量を加えるという実装でも実現できます。しかし、グラフィックス処理において座標変換が平行移動だけで済むということはまずないので、このように変換行列として表現した方が処理を単純化でき、結果的に全体の処理量の削減につながります。

$$\begin{aligned}\mathbf{T}(0,3,0)\mathbf{T}(4,0,0) &= \begin{pmatrix}1&0&0&0\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1&0&0&4\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}\\ &= \begin{pmatrix}1&0&0&4\\0&1&0&3\\0&0&1&0\\0&0&0&1\end{pmatrix} = \mathbf{T}(4,3,0)\end{aligned} \tag{41}$$

● **行列の積を求める関数**

変換の合成に用いる行列の積を求める関数は、たとえば次のようになります。OpenGL では行列が配列変数に見かけ上転置した形で格納されていることを考慮しています。

```
// 変換行列 m0 と m1 の積を m に求める
GLfloat *multiply(const GLfloat *m0, const GLfloat *m1, GLfloat *m)
{
  for (int j = 0; j < 4; ++j)
  {
    for (int i = 0; i < 4; ++i)
    {
      const int ji(j * 4 + i);

      m[ji] = 0.0f;
      for (int k = 0; k < 4; ++k)
        m[ji] += m0[k * 4 + i] * m1[j * 4 + k];
    }
  }

  return m;
}
```

あるいは、このように三重のループになっているのが気に食わなければ、たとえば次のように書くこともできます。

```
//変換行列 m0 と m1 の積を m に求める
GLfloat *multiply(const GLfloat *m0, const GLfloat *m1, GLfloat *m)
{
  for (int ji = 0; ji < 16; ++ji)
  {
    const int i(ji & 3), j(ji & ~3);

    m[ji] = m0[i] * m1[j] + m0[4 + i] * m1[j + 1]
      + m0[8 + i] * m1[j + 2] + m0[12 + i] * m1[j + 3];
  }

  return m;
}
```

7.5.2 剛体変換

平行移動と回転は、立体の形状に影響を与えません。そのため、この二つを組み合わせた変換は剛体変換と呼ばれます。

$$\mathbf{M} = \mathbf{T}(\mathbf{t})\mathbf{R}(l,m,n,\theta) = \begin{pmatrix} r_{00} & r_{10} & r_{20} & t_x \\ r_{01} & r_{11} & r_{21} & t_y \\ r_{02} & r_{12} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{42}$$

M が剛体変換の変換行列であれば、その各要素は回転の変換行列 $\bar{\mathbf{R}}$ と平行移動のベクトル **t** で構成されています。

$$\bar{\mathbf{R}} = \begin{pmatrix} r_{00} & r_{10} & r_{20} \\ r_{01} & r_{11} & r_{21} \\ r_{02} & r_{12} & r_{22} \end{pmatrix},\ \mathbf{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \Rightarrow \mathbf{M} = \begin{pmatrix} \bar{\mathbf{R}} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{43}$$

7.5.3 任意の点を中心にした回転

7.3.5 で示した回転の変換は、いずれも原点を通る軸を中心としたものでした。任意の点を通る軸を中心に回転するには、一旦その点を原点に移す平行移動を行い、回転した後に元の位置に戻します。この変換は次の三つの変換の合成変換 **M = T(p) R$_z$(θ) T(−p)** になります。

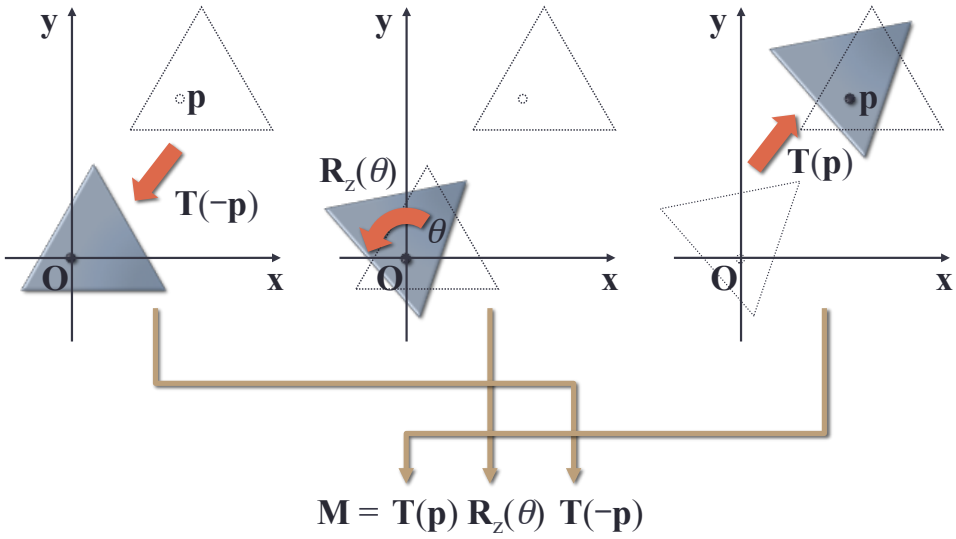

図 95 任意の点を中心とする回転

7.5.4 任意の点を中心にした拡大縮小

✦ で示した拡大縮小は、原点を基準にしています。そのため、拡大縮小を行うとする図形が原点から離れていると、図形全体の位置も変ってしまいます。

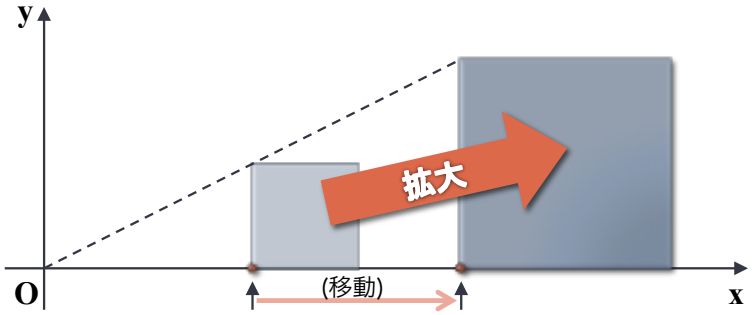

図 96 拡大縮小に伴う図形の移動

図形を、その図形の位置で拡大するためには、拡大縮小を行う際の基準点を図形の位置に移す必要があります。これは基準点が原点になるように図形を平行移動し、その後に拡大縮小して、元の位置に戻すことで実現します。この変換は次の三つの変換の合成変換 **M = T(p) S(s) T(−p)** になります。

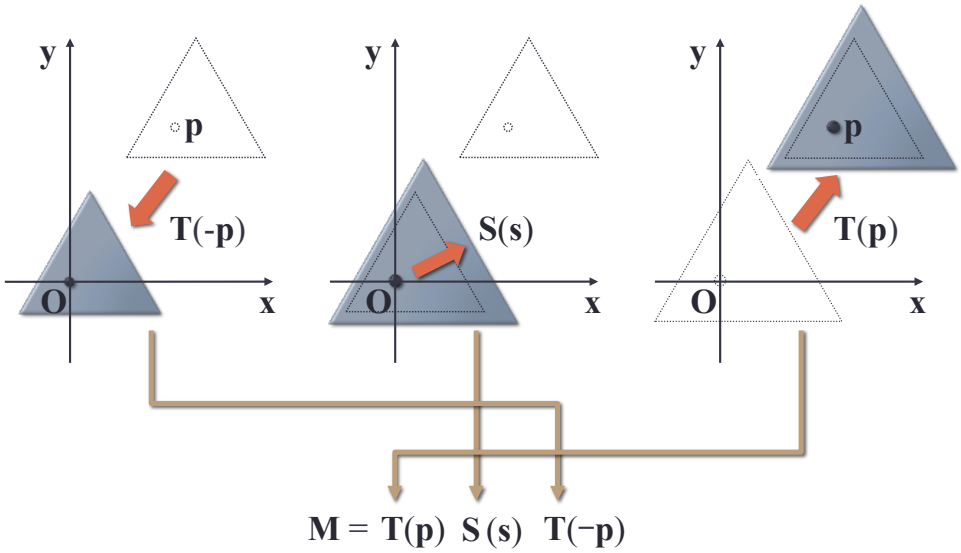

図 97 任意の基準点による拡大縮小

7.5.5 特定方向への拡大縮小

✦ で示した拡大縮小は、また X、Y、Z のそれぞれの軸方向に対する拡大縮小しか行うことができません。任意の方向に対して拡大縮小を行うためには、その方向を一旦 X、Y、Z 軸の方向に合わせてから拡大縮小を行い、それから元の向きに戻します。いま、軸ベクトル **(x y z)** をそれぞれ **(r s t)** 方向に回転する変換行列を **F**、s_r、s_s、s_t をそれぞれ X、Y、Z 軸方向の拡大率とする拡大縮小の変換行列 **S** とするとき、この変換は $\mathbf{M} = \mathbf{FSF}^T$ となります。

$$\mathbf{F} = \begin{pmatrix} \mathbf{r} & \mathbf{s} & \mathbf{t} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{44}$$

$$\mathbf{S} = \begin{pmatrix} s_r & 0 & 0 & 0 \\ 0 & s_s & 0 & 0 \\ 0 & 0 & s_t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{45}$$

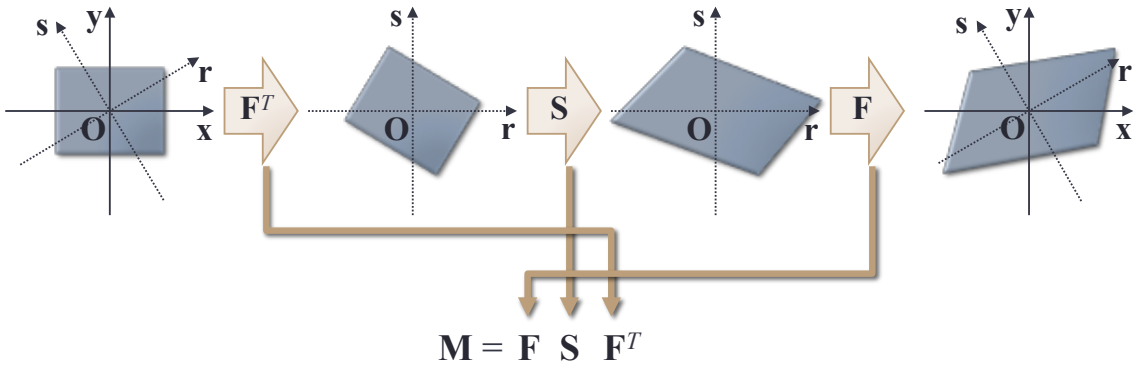

図 98 特定方向への拡大縮小

7.5.6 変換の順序

変換の合成は行列の積で表されますが、行列の積は交換の法則が成り立ちません。したがって、同じ変換行列を合成した場合でも、その順序が異なれば、合成変換の結果は異なります。

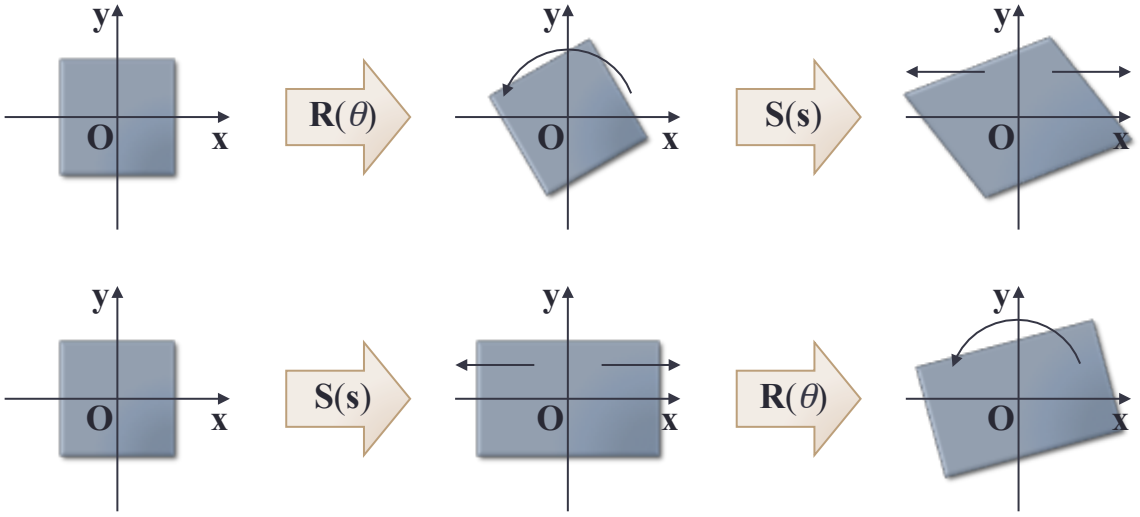

図 99 変換の順序による合成変換の結果の違い

7.6 オイラー変換

7.6.1 オイラー角とオイラー変換

原点を中心とした任意の回転は、X, Y, Z の三つの軸中心の回転を合成して表すことができます。この回転を表す各軸中心の回転角の組を**オイラー角**と呼び、それによって表される回転の変換をオイラー変換と呼びます。

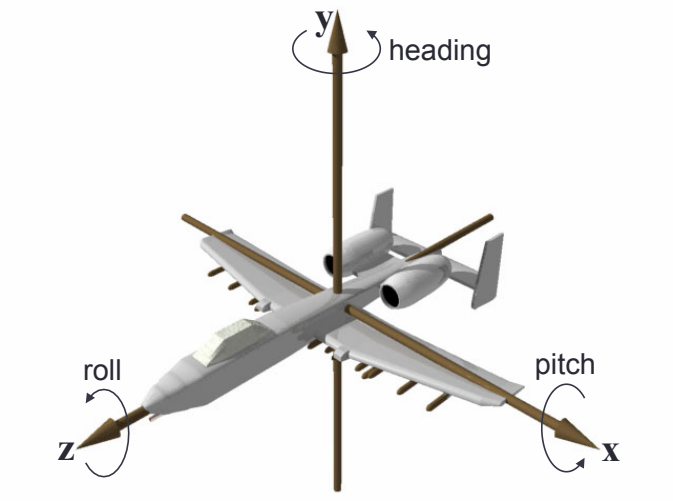

図 100 オイラー角

合成変換は変換行列の積で表されますが、行列の積には交換法則が成り立たちません。したがってオイラー変換の結果は、各軸中心の回転を合成する順序によって変化します。ここでは、Z 軸中心の回転 (roll, bank) → X 軸中心の回転 (pitch) → Y 軸中心の回転 (heading, yaw) の順に変換するものとします。また、それぞれの角度を r, p, h で表します。

- r: roll, bank (Z 軸中心の回転角)
- p: pitch (X 軸中心の回転角)
- h: heading, yaw (Y 軸中心の回転角)

このとき、オイラー変換 $\mathbf{E}(h, p, r)$ は次式で表されます。

$$
\begin{aligned}
\mathbf{E}(h,p,r) &= \mathbf{R}_y(h)\mathbf{R}_x(p)\mathbf{R}_z(r) \\
&= \begin{pmatrix} \cos h & 0 & \sin h & 0 \\ 0 & 1 & 0 & 0 \\ -\sin h & 0 & \cos h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos p & -\sin p & 0 \\ 0 & \sin p & \cos p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos r & -\sin r & 0 & 0 \\ \sin r & \cos r & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (46) \\
&= \begin{pmatrix} \sin h \sin p \sin r + \cos h \cos r & \sin h \sin p \cos r - \cos h \sin r & \sin h \cos p & 0 \\ \cos p \sin r & \cos p \cos r & -\sin p & 0 \\ \cos h \sin p \sin r - \sin h \cos r & \cos h \sin p \cos r + \sin h \sin r & \cos h \cos p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

7.6.2 ジンバルロック

オイラー角のうち、$p = \pi / 2$ の場合を考えます。このとき、$\sin p = 1, \cos p = 0$ となりますから、(46) 式は次のようになります。

$$
\mathbf{E}(h, \pi/2, r) = \begin{pmatrix} \sin h \sin r + \cos h \cos r & \sin h \cos r - \cos h \sin r & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \cos h \sin r - \sin h \cos r & \cos h \cos r + \sin h \sin r & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad (47)
$$

これは、加法定理により、次のように書き換えられます。

$$\mathbf{E}(h,\pi/2,r)=\begin{pmatrix}\cos(h-r) & \sin(h-r) & 0 & 0\\ 0 & 0 & -1 & 0\\ -\sin(h-r) & \cos(h-r) & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix} \tag{48}$$

この回転は **(*h* - *r*)** という単一の角度で決定される、一つの軸による回転です。このため、*r* と *h* のどちらを変化させても同じ回転になり、自由度が一つ減ってしまいます。*p* = π / 2 の回転によって *r* の回転軸 (Z 軸) が *h* の回転軸と一致します。この現象を**ジンバルロック**と呼びます。

7.6.3 回転変換行列からオイラー角の算出

ある回転変換行列 **M** がオイラー変換 **E**(*h*, *p*, *r*) にもとづくものであったとします。

$$\mathbf{M}=\begin{pmatrix}m_0 & m_4 & m_8 & m_{12}\\ m_1 & m_5 & m_9 & m_{13}\\ m_2 & m_6 & m_{10} & m_{14}\\ m_3 & m_7 & m_{11} & m_{15}\end{pmatrix}=\mathbf{E}(h,p,r) \tag{49}$$

m_1 と m_5 の関係から、*r* を求めることができます。

$$\left.\begin{aligned}m_1&=\cos p\sin r\\ m_5&=\cos p\cos r\end{aligned}\right\}\ \rightarrow\ r=\mathtt{atan2}(m_5,m_1) \tag{50}$$

同様に、m_8 と m_{10} の関係から、*h* を求めることができます。

$$\left.\begin{aligned}m_8&=\sin h\cos p\\ m_{10}&=\cos h\cos p\end{aligned}\right\}\ \rightarrow\ h=\mathtt{atan2}(m_{10},m_8) \tag{51}$$

p は m_9 から求めることができます。

$$m_9=-\sin p\rightarrow\ p=\mathtt{asin}(-m_9) \tag{52}$$

ただし、$m_1 = m_5 = 0$ のときは $\cos p = 0$ なので、*p* = ± π / 2 となります。これはジンバルロックが発生している状態であり、(50) 式や (51) 式で *r* や *h* を求めることができません。その場合は、例えば $h = 0$ とし、m_0 と m_{14} の関係から、*h* を求めることができます。

$$\left.\begin{aligned}m_0&=\cos(h\mp r)\\ m_4&=\sin(h\mp r)\end{aligned}\right\}\ \rightarrow\ h=0,r=-\mathtt{atan2}(m_0,m_4) \tag{53}$$

7.7 ビュー変換

7.7.1 視点の位置の移動

視点の位置を三次元空間中の別の場所に移して、そこから見た図形を描くことを考えてみます。ビュー変換は図形を視点の位置と方向からみた座標系 (視点座標系) に移す座標変換です。

ここで現在の状態、すなわち二次元の図形を表示している状態で、視点の位置がどこにあるか考えてみます。画面上では右方向に x 軸が伸び、上方向に y 軸が伸びています。z 軸は x 軸と

y 軸の両方と直交していますから、画面に対して垂直に伸びているはずです。OpenGL では右手系 (図 101) の座標系を用いますから、z 軸は画面から見ている人の方に向かって伸びています。

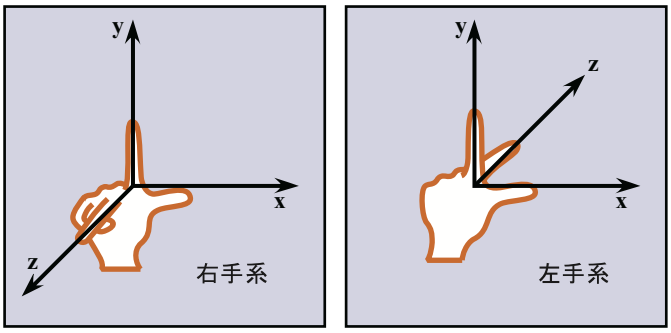

図 101 右手系と左手系

これは、言い換えれば、視点は原点にあり、z 軸の負の方向を向いていることになります。ビュー変換は三次元空間中に置かれた視点の位置を原点とし、視線を z 軸の負の方向としたときの、図形の位置を求める処理になります。

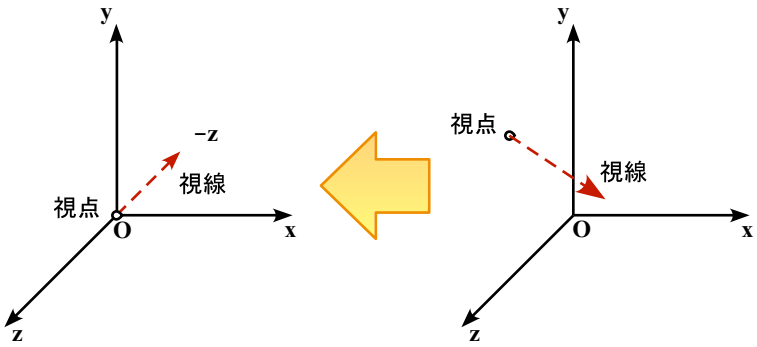

図 102 ビュー変換

7.7.2 ビュー変換行列

● 視点の設定

この変換行列を求める関数を作ります。視点の位置を $\mathbf{e} = (e_x, e_y, e_z)$、視線上にある目標点の位置を $\mathbf{g} = (g_x, g_y, g_z)$ とします。このほか、この視点の「上方向」のベクトルを $\mathbf{u} = (u_x, u_y, u_z)$ とします。上方向を y 軸方向とするなら、これに (0, 1, 0) を設定します。

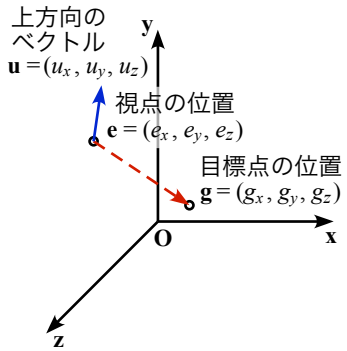

図 103 視点の設定

● **視点を原点に平行移動**

まず、視点 $\mathbf{e} = (e_x, e_y, e_z)$ を原点 $\mathbf{O} = (0, 0, 0)$ に平行移動する変換行列 $\mathbf{T}_v$ を作ります。

$$\mathbf{T}_v = \begin{pmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{54}$$

● **視点座標系の軸 r, s, t**

視点の位置 $\mathbf{e}$ を原点に移すと、目標点の位置は目標点の位置は $\mathbf{g} - \mathbf{e}$ になります。これと上方向のベクトル $\mathbf{u}$ から、視点を原点とした座標系である視点座標系の軸ベクトルを求めます。これを $(\mathbf{r}\ \mathbf{s}\ \mathbf{t})$ とします。

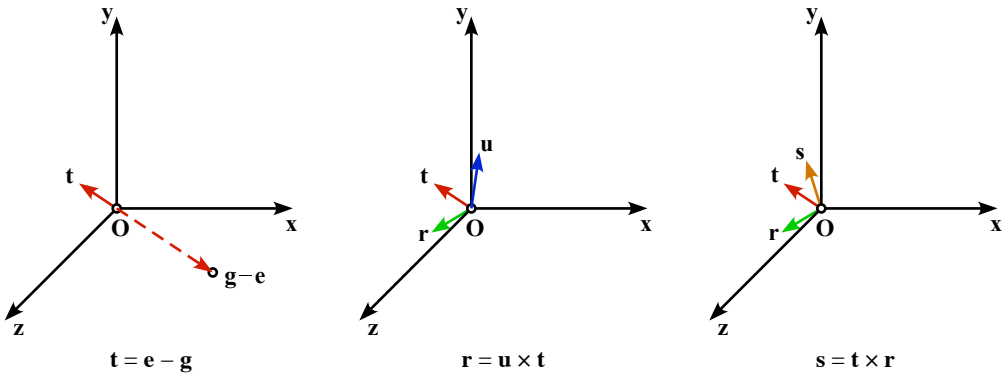

図 104 視点座標系の軸ベクトル (r s t)

視点座標系における z 軸に相当するベクトル $\mathbf{t} = (t_x, t_y, t_z)$ は、視線方向の逆ベクトル $\mathbf{e} - \mathbf{g}$ になります。視点座標系の x 軸に相当するベクトル $\mathbf{r} = (r_x, r_y, r_z)$ は、上方向のベクトル $\mathbf{u}$ と $\mathbf{t}$ との外積 $\mathbf{u} \times \mathbf{t}$ により求めます。視点の座標系の y 軸に相当するベクトル $\mathbf{s} = (s_x, s_y, s_z)$ を、ベクトル $\mathbf{t}$ とベクトル $\mathbf{r}$ の外積 $\mathbf{t} \times \mathbf{r}$ で求めます。

$$\begin{aligned} \mathbf{t} = \mathbf{e} - \mathbf{g} &= \begin{pmatrix} e_x - g_x \\ e_y - g_y \\ e_z - g_z \end{pmatrix} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \\ \mathbf{r} = \mathbf{u} \times \mathbf{t} &= \begin{pmatrix} u_y t_z - u_z t_y \\ u_z t_x - u_x t_z \\ u_x t_y - u_y t_x \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} \\ \mathbf{s} = \mathbf{t} \times \mathbf{r} &= \begin{pmatrix} t_y r_z - t_z r_y \\ t_z r_x - t_x r_z \\ t_x r_y - t_y r_x \end{pmatrix} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix} \end{aligned} \tag{55}$$

● **視線を目標点に向けて回転**

得られた視点座標系の軸ベクトル $(\mathbf{r}\ \mathbf{s}\ \mathbf{t})$ をそれぞれ正規化し、式 (40) にもとづいて式 (56) に示す回転の変換行列 $\mathbf{R}_v$ を作ります。

$$\mathbf{R}_v = \begin{pmatrix} \dfrac{\mathbf{r}^T}{|\mathbf{r}|} & 0 \\ \dfrac{\mathbf{s}^T}{|\mathbf{s}|} & 0 \\ \dfrac{\mathbf{t}^T}{|\mathbf{t}|} & 0 \\ 0 \quad 0 \quad 0 & 1 \end{pmatrix} = \begin{pmatrix} \dfrac{r_x}{|\mathbf{r}|} & \dfrac{r_y}{|\mathbf{r}|} & \dfrac{r_z}{|\mathbf{r}|} & 0 \\ \dfrac{s_x}{|\mathbf{s}|} & \dfrac{s_y}{|\mathbf{s}|} & \dfrac{s_z}{|\mathbf{s}|} & 0 \\ \dfrac{t_x}{|\mathbf{t}|} & \dfrac{t_y}{|\mathbf{t}|} & \dfrac{t_z}{|\mathbf{t}|} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{56}$$

まず、ベクトル $\mathbf{s}$ の長さを調べます。これが 0 でなければ、ベクトル $\mathbf{r}$ とベクトル $\mathbf{t}$ の長さは、ともに 0 ではありません。このとき、これらを正規化して、$4 \times 4 = 16$ 要素の配列変数 rv の左上の 3×3 の要素に格納します。

● ビュー変換行列

平行移動の変換行列 $\mathbf{T}_v$ に回転の変換行列 $\mathbf{R}_v$ を乗じて、ビュー変換行列を求めます。

$$\mathbf{M}_v = \mathbf{R}_v \mathbf{T}_v \tag{57}$$

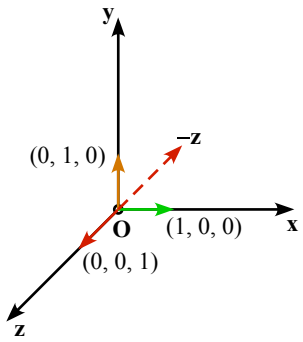

図 105 回転の変換適用後の視点座標系の座標軸

● ビュー変換行列を設定する関数

```
// ビュー変換行列を m に設定する
GLfloat *loadLookat(
  GLfloat ex, GLfloat ey, GLfloat ez,   // 視点の位置
  GLfloat gx, GLfloat gy, GLfloat gz,   // 目標点の位置
  GLfloat ux, GLfloat uy, GLfloat uz,   // 上方向のベクトル
  GLfloat *m)
{
  // 平行移動の変換行列
  GLfloat tv[16];
  loadTranslate(-ex, -ey, -ez, tv);

  // t 軸 = e - g
  const GLfloat tx(ex - gx);
  const GLfloat ty(ey - gy);
  const GLfloat tz(ez - gz);

  // r 軸 = u x t 軸
  const GLfloat rx(uy * tz - uz * ty);
  const GLfloat ry(uz * tx - ux * tz);
  const GLfloat rz(ux * ty - uy * tx);
```

```
  // s 軸 = t 軸 x r 軸
  const GLfloat sx(ty * rz - tz * ry);
  const GLfloat sy(tz * rx - tx * rz);
  const GLfloat sz(tx * ry - ty * rx);

  // s 軸の長さのチェック
  const GLfloat s2(sx * sx + sy * sy + sz * sz);
  if (s2 == 0.0f) return m;

  // 回転の変換行列
  GLfloat rv[16];

  // r 軸を正規化して配列変数に格納
  const GLfloat r(sqrt(rx * rx + ry * ry + rz * rz));
  rv[ 0] = rx / r;
  rv[ 4] = ry / r;
  rv[ 8] = rz / r;

  // s 軸を正規化して配列変数に格納
  const GLfloat s(sqrt(s2));
  rv[ 1] = sx / s;
  rv[ 5] = sy / s;
  rv[ 9] = sz / s;

  // t 軸を正規化して配列変数に格納
  const GLfloat t(sqrt(tx * tx + ty * ty + tz * tz));
  rv[ 2] = tx / t;
  rv[ 6] = ty / t;
  rv[10] = tz / t;

  // 残りの成分
  rv[ 3] = rv[ 7] = rv[11] = rv[12] = rv[13] = rv[14] = 0.0f;
  rv[15] = 1.0f;

  // 視点の平行移動の変換行列に視線の回転の変換行列を乗じる
  return multiply(rv, tv, m);
}
```

7.8 投影変換

7.8.1 標準視体積

二次元の図形表示では、画面上の表示領域 (ビューポート) に正規化デバイス座標系のクリッピング領域内のものが表示されます。この領域には**深度** (奥行き) 方向にも表示可能な範囲が設定されており、これは図 106 に示す各軸 [-1, 1] の範囲の立方体の空間です。これを**標準視体積** (Canonical View Volume) といいます。画面には、この xy 平面上への投影像が表示されます。

このため、三次元空間の任意の領域に置かれた物体を画面に表示するには、表示しようとする領域を、この標準視体積に収める必要があります。これは投影変換により行います。

投影変換の方法には、遠くも近くも同じ大きさで表示される**直交投影**と、遠いものほど小さく表示される**透視投影**があります。これら以外にも、たとえば遠いものほど大きく表示したり、魚眼レンズのように歪ませて表示したりする投影方法などもありますが、ここでは取り扱いません。

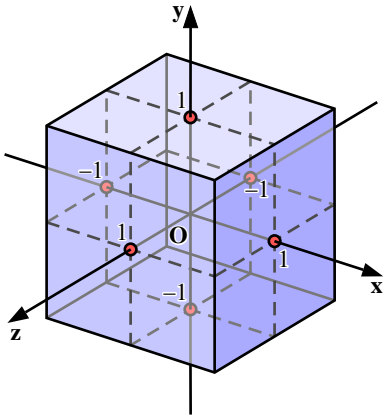

図 106 標準視体積

7.8.2 直交投影

● **直交投影の手順**

三次元空間中の任意の領域を標準視体積に収めるために、直交投影ではワールド座標系上の 2 点 (*left*, *bottom*) および (*right*, *top*) を対角の頂点とする矩形の表示領域の他に、深度方向の範囲 (*near*, *far*) も指定します。

いま、視点が原点にあり、視線が xy 平面に垂直な z 軸上の負の方向を向いている (右手系) とします。*near* は表示を行うもっとも視点に視線に垂直な平面の深度 (xy 平面からの距離) であり、*far* は最も遠方にある平面の深度です。これらによって決まる直方体の領域を、**視体積** (View Volume) といいます。

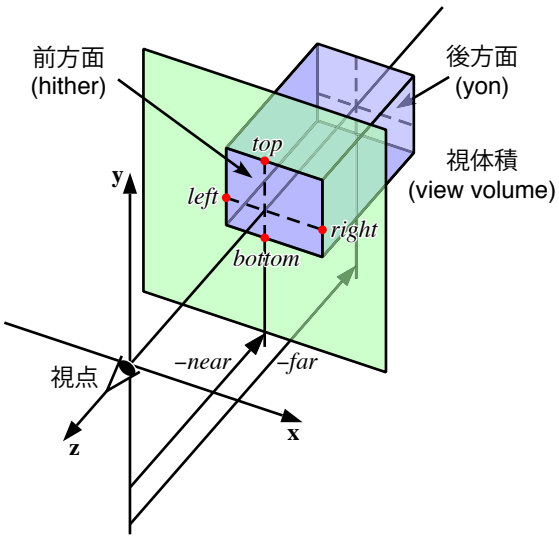

図 107 視体積

この視点に近いほうの平面を**前方面** (hither) 、遠い方の面を**後方面** (yon) といいます。視野変換によって視点は z 軸方向の負の方向に向いていますから、前方面と後方面の z 値は、それぞれ *−near*、*−far* になります。

この視体積内の図形を標準視体積に収めるには、視体積の中心が原点になるように平行移動し、一辺の長さが 2 になるように拡大縮小します。

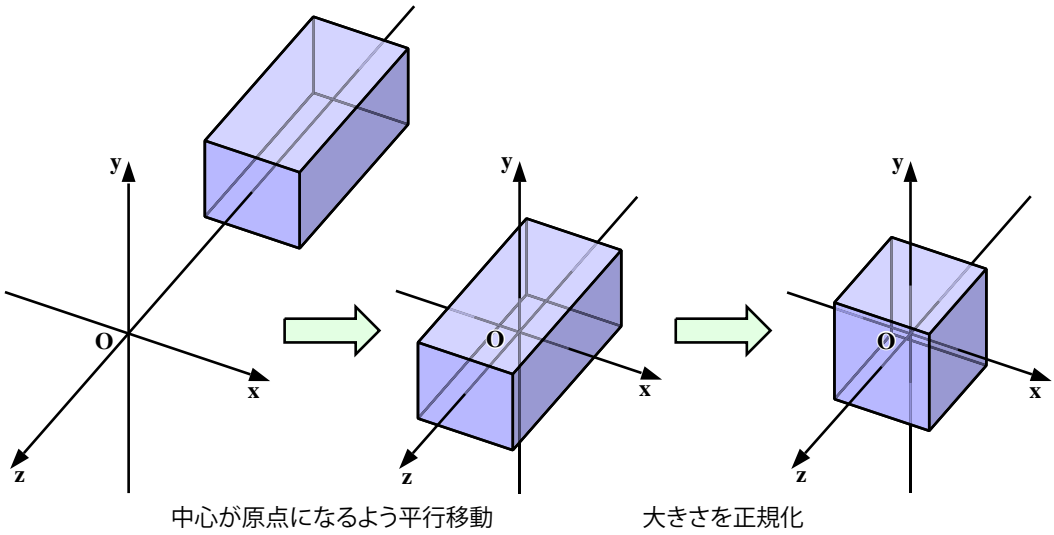

図 108 直交投影の手順

● 中心が原点になるよう平行移動

まず、視体積の中心の位置を求め、そこが原点になるように平行移動する変換行列を求めます。視体積の中心は $((right + left) / 2, (top + bottom) / 2, -(far + near) / 2)$ です。

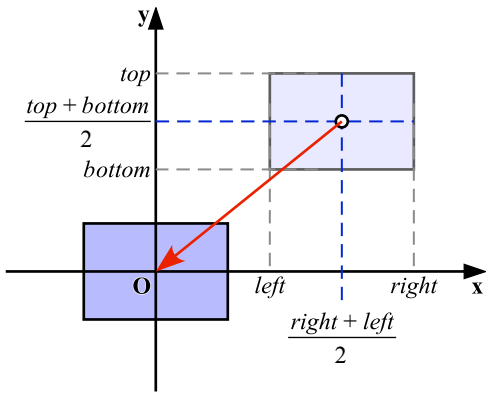

図 109 中心が原点となるように平行移動

この変換行列は平行移動の変換行列 (21) より、(58) 式のようになります。*far* と *near* に負号が付いているので、Z に関しては逆に負号が無くなっています。

$$\mathbf{M}_{centering} = \begin{pmatrix} 1 & 0 & 0 & -\dfrac{right + left}{2} \\ 0 & 1 & 0 & -\dfrac{top + bottom}{2} \\ 0 & 0 & 1 & \dfrac{far + near}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{58}$$

● 大きさを正規化

視体積の幅、高さ、および深度を求め、それらの長さが 2 になるように拡大縮小する変換行列を求めます。視体積の幅、高さ、深度は、それぞれ *right* – *left*、*top* – *bottom*、–(*far* – *near*) です。

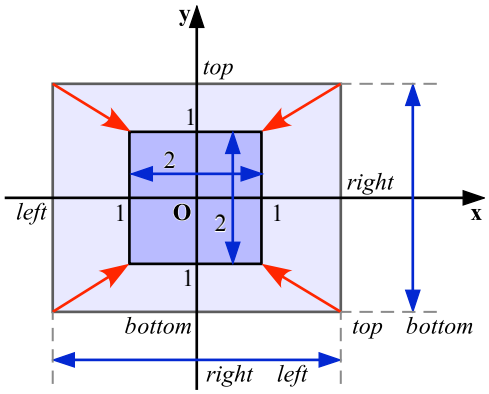

図 110 一辺の長さが 2 になるよう拡大縮小

この変換行列は拡大縮小の変換行列 (21) より (59) 式のようになります。これも *far* と *near* に負号が付いているので、Z に関しては負号が付いています。

$$\mathbf{M}_{scaling} = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & -\frac{2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{59}$$

● 直交投影変換行列

直交投影変換行列 $\mathbf{M}_o$ は、この二つの変換行列 $\mathbf{M}_{centering}$ と $\mathbf{M}_{scaling}$ を乗じたものです。

$$\begin{aligned} \mathbf{M}_o &= \mathbf{M}_{scaling}\mathbf{M}_{centering} \\ &= \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{60}$$

● 直交投影変換行列を設定する関数

配列変数 m に直交投影の変換行列を設定する関数は、たとえば次のようになります。

```
// 直交投影変換行列を m に設定する
GLfloat *loadOrthogonal(GLfloat left, GLfloat right,
  GLfloat bottom, GLfloat top, GLfloat zNear, GLfloat zFar, GLfloat *m)
{
  const GLfloat dx(right - left);
```

```
  const GLfloat dy(top - bottom);
  const GLfloat dz(zFar - zNear);

  if (dx != 0.0f && dy != 0.0f && dz != 0.0f)
  {
    m[ 0] = 2.0f / dx;
    m[ 5] = 2.0f / dy;
    m[10] = -2.0f / dz;
    m[12] = -(right + left) / dx;
    m[13] = -(top + bottom) / dy;
    m[14] = -(zFar + zNear) / dz;
    m[15] = 1.0f;
    m[ 1] = m[ 2] = m[ 3] = m[ 4] = m[ 6] = m[ 7] = m[ 8] = m[ 9] = m[11] = 0.0f;
  }

  return m;
}
```

7.8.3 透視投影

透視投影も直交投影と同様に、前方面上の 2 点 (*left*, *bottom*) および (*right*, *top*) を対角の頂点とする矩形の表示領域と、深度方向の範囲 (*near*, *far*) を指定します。ただし透視投影の場合は、視点を頂点とする四角錐を前方面で切り落とした空間 (錐台) を標準視体積に収めます。この空間を**視錐台** (View Frustum) といいます。

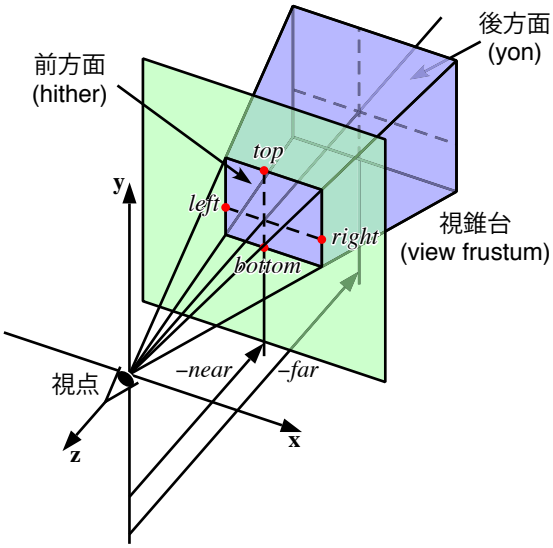

図 111 視錐台

この視錐台内の図形を標準視体積に収めるには、視錐台の中心軸が視線 (z 軸) と一致するようにせん断変形し、それを透視変換した後、中心が原点になるように平行移動し、最後に一辺の長さが 2 になるように拡大縮小します。

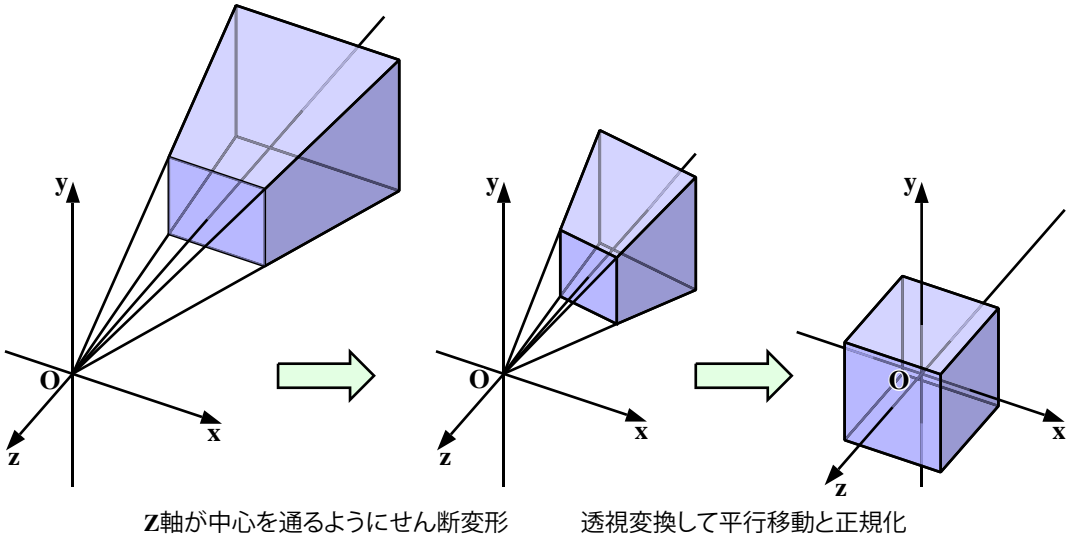

図 112 透視投影の手順

● せん断変形

透視投影の場合は、前方面上の矩形の表示領域の中心を z 軸が通るようにするために、視錐台全体をせん断変形します。前方面は *–near* の位置にあり、その上の表示領域の中心は ((*right* + *left*) / 2, (*top* + *bottom*) / 2) にありますから、z = –1 における移動量は ((*right* + *left*) / 2*near*, (*top* + *bottom*) / 2 *near*) になります。

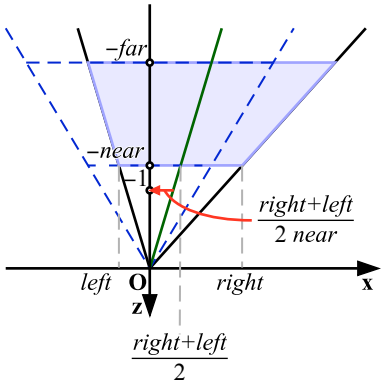

図 113 せん断変形変換

$$\mathbf{M}_{shear} = \begin{pmatrix} 1 & 0 & \dfrac{right+left}{2\,near} & 0 \\ 0 & 1 & \dfrac{top+bottom}{2\,near} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{61}$$

● 透視変換

透視投影では、物体の見かけの大きさが距離 (深度, z 値) に反比例します。(*x*, *y*, *z*) の位置にある点は *–near* の位置にある前方面の (*–near x* / *z*, *–near y* / *z*) の位置に投影されます。z 値に関しては、✦ の**透視深度**を参照してください。

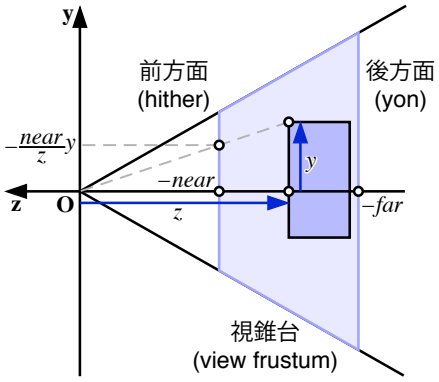

図 114 透視変換

$$\mathbf{M}_{perspective} = \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & \frac{far + near}{2} & far\,near \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{62}$$

● 透視投影変換行列

透視投影変換行列 $\mathbf{M}_p$ は、この二つの変換行列 $\mathbf{M}_{perspective}$ と $\mathbf{M}_{shear}$ の積に、大きさを正規化する変換行列 $\mathbf{M}_{scaling}$ を乗じたものです。

$$\begin{aligned} \mathbf{M}_p &= \mathbf{M}_{scaling}\mathbf{M}_{perspective}\mathbf{M}_{shear} \\ &= \begin{pmatrix} \frac{2\,near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2\,near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2\,far\,near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \end{aligned} \tag{63}$$

● 透視投影変換行列を設定する関数

配列変数 m に透視投影の変換行列を設定する関数は、たとえば次のようになります。

```
// 透視投影変換行列を m に設定する
GLfloat *loadFrustum(GLfloat left, GLfloat right,
  GLfloat bottom, GLfloat top, GLfloat zNear, GLfloat zFar, GLfloat *m)
{
  const GLfloat dx(right - left);
  const GLfloat dy(top - bottom);
  const GLfloat dz(zFar - zNear);

  if (dx != 0.0f && dy != 0.0f && dz != 0.0f)
  {
    m[ 0] = 2.0f * zNear / dx;
    m[ 5] = 2.0f * zNear / dy;
    m[ 8] = (right + left) / dx;
    m[ 9] = (top + bottom) / dy;
```

```
      m[10] = -(zFar + zNear) / dz;
      m[11] = -1.0f;
      m[14] = -2.0f * zFar * zNear / dz;
      m[ 1] = m[ 2] = m[ 3] = m[ 4] = m[ 6] = m[ 7] = m[12] = m[13] = m[15] = 0.0f;
    }

    return m;
  }
```

7.8.4 透視深度

透視投影では、投影像の大きさが深度 (z 値) に反比例します。透視投影後の深度にも、元の深度の逆数を用います。これにより、透視変換後の位置の精度は前方面に近いほど高く、前方面から離れるほど低くなります。このような深度を**透視深度**と言います。

いま図 115 において、xz 平面上の点 $A = (x_a, z_a)$ および $B = (x_b, z_b)$ が $z = -near$ の位置にある前方面に投影された位置を、それぞれ C および D とします。

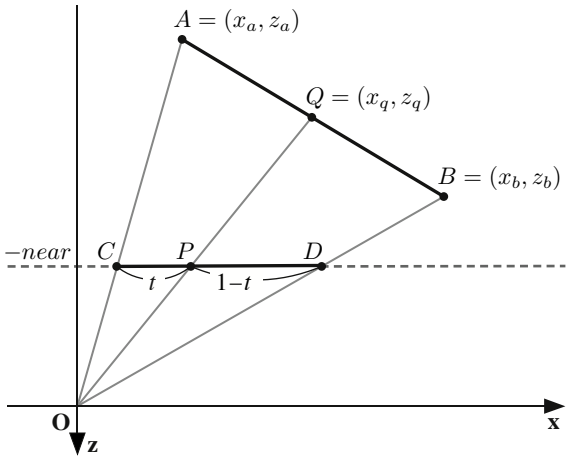

図 115 投影面上の線形補間

この線分 CD を $t : 1-t$ で内分する点を P とするとき、P を AB 上に逆透視投影した点の位置 $Q = (x_q, z_q)$ は、式 (64) により求めることができます。

$$\begin{cases} x_q = \dfrac{\dfrac{x_a}{z_a}(1-t) + \dfrac{x_b}{z_b}t}{\dfrac{1}{z_a}(1-t) + \dfrac{1}{z_b}t} \\ z_q = \dfrac{1}{\dfrac{1}{z_a}(1-t) + \dfrac{1}{z_b}t} \end{cases} \tag{64}$$

式 (64) の Q の z 成分 z_q は、A の z 成分 z_a と B の z 成分 z_b の**逆数を線形補間したものの逆数**になります。ラスタ化処理では画面上の画素の位置における深度を求める必要がありますが、これは透視深度を線形補間したものの逆数として得られます。

Q の x 成分 x_q の分子は、この点の z = 1 の投影面上での位置を線形補間により求めたものです。一方、分母は Q の透視深度、すなわち z_q の逆数です。

ただし逆数そのままだと、前方面に近いほど大きく遠くなるにつれて小さくなってしまうので、隠面消去処理の際に都合が悪くなってしまいます。また、この深度を標準視体積に収める必要もあります。

そこで、深度 (z 値) の逆数に $-far\ near$ を乗じ、これから $(far - near) / 2$ を減じて 0 点が中央に来るように平行移動します。そして最後に式 (59) の $\mathbf{M}_{scaling}$ により $-2 / (far - near)$ 倍して、[-1, 1] の範囲に正規化します。これによって「視点から遠いほど大きな値」になります。

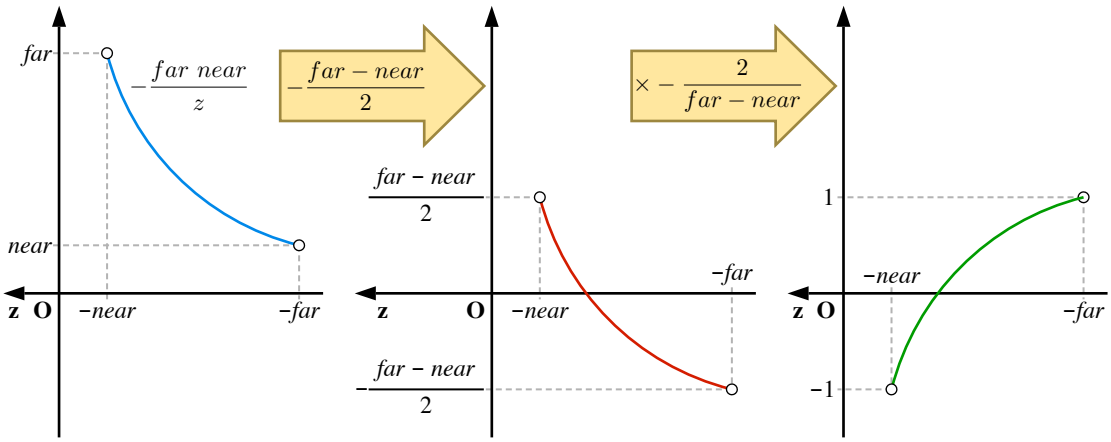

図 116 透視深度の正規化

以上により、この透視変換は式 (62) で表されます。

$$\begin{aligned} x^* &= -\frac{near}{z}x \\ y^* &= -\frac{near}{z}y \\ z^* &= -\frac{far\,near}{z} - \frac{far + near}{2} \end{aligned} \tag{65}$$

この変換を行列で表すと、式 (62) の $\mathbf{M}_{perspective}$ になります。実座標 (x, y, z) にこの変換を適用すれば、式 (66) より式 (67) に示す同時座標が得られます。

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} near & 0 & 0 & 0 \\ 0 & near & 0 & 0 \\ 0 & 0 & \frac{far + near}{2} & far\,near \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{66}$$

$$\begin{aligned} x' &= near\,x \\ y' &= near\,y \\ z' &= \frac{far + near}{2}z + far\,near \\ w' &= -z \end{aligned} \tag{67}$$

これから実座標 (x^*, y^*, z^*) を求めれば、式 (65) が得られます。

7.8.5　画角と縦横比をにもとづく透視投影変換行列

視錐台の指定による透視投影変換行列の設定は、汎用性は高いものの、直感的には理解しづらいものがあります。そこで、カメラのレンズの y 方向の**画角** (*fovy*) と画面の**縦横比** (*aspect ratio*) を使って透視投影変換行列を設定する方法がしばしば用いられます。

式 (68) の f は画角 *fovy* で上下端が±2 の投影面に投影した時の、投影面までの距離 (焦点距離) です。したがって $z = -near$ の位置にある前方面上の表示領域は、$-bottom = top = near / f$、$-left = right = aspect\ near / f$ となります。これを式 (63) に代入すれば、式 (69) が得られます。

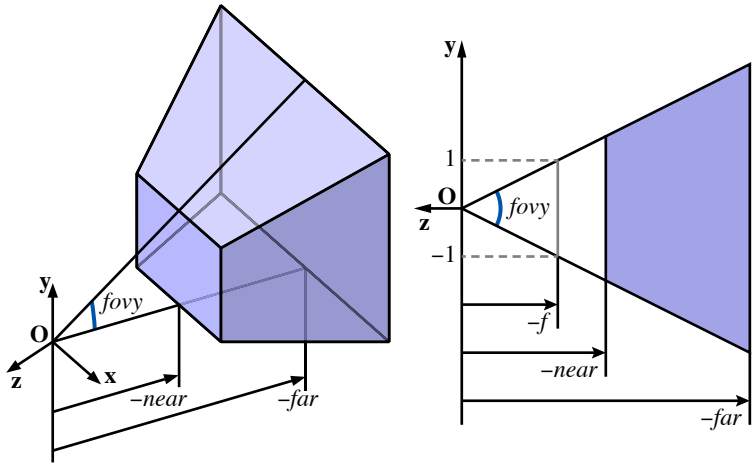

図 117 画角と縦横比にもとづく透視投影変換行列

$$f = \frac{1}{\tan\left(\frac{fovy}{2}\right)} = \cot\left(\frac{fovy}{2}\right) \tag{68}$$

$$\mathbf{M}_p = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2\,far\,near}{far - near} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{69}$$

● **画角を使って透視投影変換行列を設定する関数**

画角を指定して透視投影の変換行列を配列変数 m に設定する関数の例を以下に示します。

```
// 画角を指定して透視投影変換行列を m に設定する
GLfloat *loadPerspective(GLfloat fovy, GLfloat aspect,
  GLfloat zNear, GLfloat zFar, GLfloat *m)
{
  const GLfloat dz(zFar - zNear);

  if (dz != 0.0f)
  {
    m[ 5] = 1.0f / tan(fovy * 0.5f);
    m[ 0] = m[5] / aspect;
    m[10] = -(zFar + zNear) / dz;
```

```
    m[11] = -1.0f;
    m[14] = -2.0f * zFar * zNear / dz;
    m[ 1] = m[ 2] = m[ 3] = m[ 4] =
    m[ 6] = m[ 7] = m[ 8] = m[ 9] =
    m[12] = m[13] = m[15] = 0.0f;
  }

  return m;
}
```