

射影変換とモデルビュー変換

2015 年度 情報可視化論

陰山 聡

2015.06.23

射影変換と同次座標

クリップ座標

これまではクリップ座標の立方体の中だけで描画していた。つまりビューボリュームは正規化ビューボリューム

$$[-1, +1] \times [-1, +1] \times [-1, +1]$$

で固定であった。これは不便。

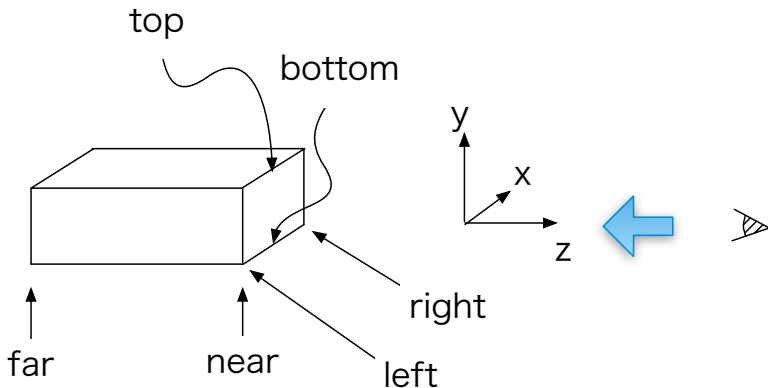
長さの単位（大きさ）を気にしないで、つまりワールド座標で物体を自由に定義したい。

射影変換

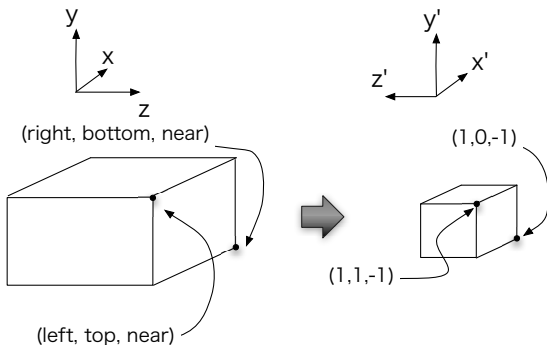
- 正射影
- 透視射影

正射影

- カメラは $z = \infty$ に位置し、 $-z$ 方向を向いている。
- 正規化ビューボリュームの座標系は左手系。



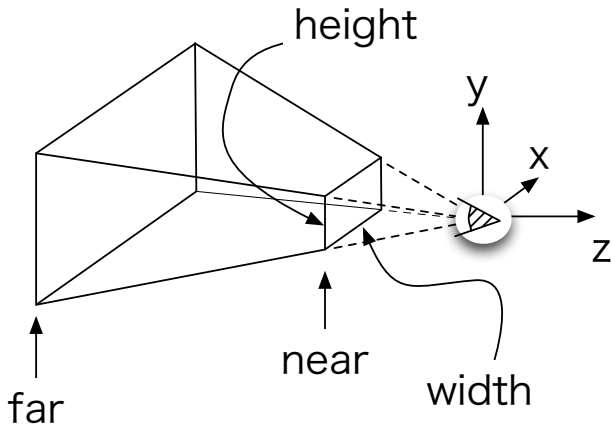
正射影



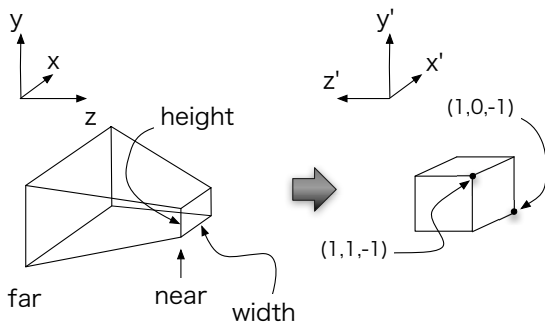
- 直方体から立方体（正規化ビューボリューム）への単純なスケール変換行列。
- 自分で行列を設定するのも簡単だが、次の関数が用意されている
- `mat4.ortho(left, right, bottom, top, near, far, projectionMatrix);`

透視射影

- カメラは原点に位置し、 $-z$ 方向を向いている。
- 物体を回転・移動させればいつでも $-z$ 方向に見えるようにできる。

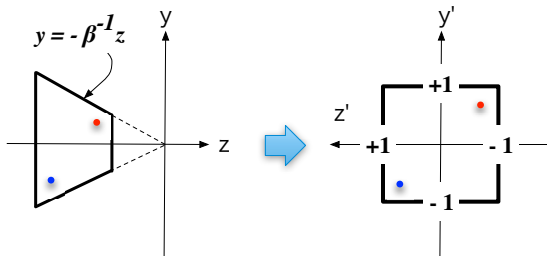


透視射影



- 視錐台形から立方体への変換。

透視射影



$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta x/z \\ -\beta y/z \\ c_1 z + c_2 \end{pmatrix}$$

$\beta > 0, c_1, c_2$ は定数。 x と y に関しては $-\beta/z$ 倍のスケール変換。

(このスケール係数は、たとえば、視錐台の上面 $y = -\beta^{-1}z$ が $y' = +1$ に変換されることで確認できる。)

(復習)

平行移動変換

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix}$$

は同次座標を使うと行列演算で書けた。

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

では、透視射影

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta x/z \\ -\beta y/z \\ c_1 z + c_2 \end{pmatrix}$$

も工夫すれば行列で書けるか？

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

… できない。

では、透視射影

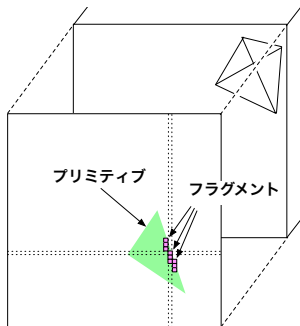
$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta x/z \\ -\beta y/z \\ c_1 z + c_2 \end{pmatrix}$$

も工夫すれば行列で書けるか？

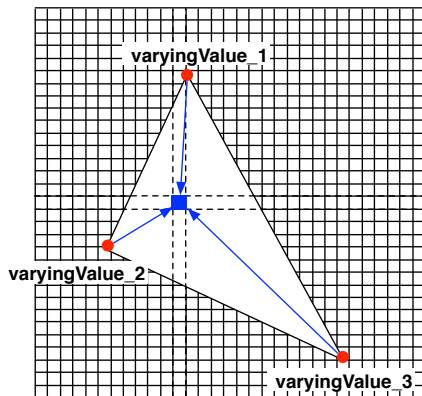
$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

z 補間の問題

- 透視射影の場合、もう一つ注意しなければならない問題がある。
- 復習：フラグメントシェーダに渡される時の varying 値の自動線形補間

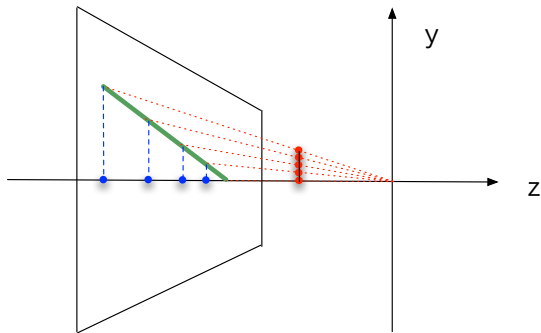


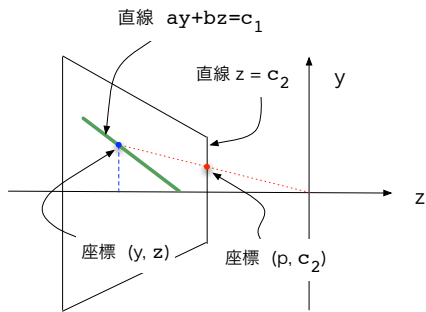
z 補間の問題



z 補間の問題

- ピクセル（赤）が等間隔に並んでいても
- プリミティブ（緑）上の補間点（青）の z 値は非等間隔。
- これは都合が悪い。



等間隔ピクセル p は等間隔の $1/z$ に対応

$$(ap/c_2 + b)z = c_1$$

$$\frac{1}{z} = \frac{a}{c_2 c_1} p + \frac{b}{c_1}$$

そこで z の代わりに

$$z \Rightarrow z' = -\alpha - \gamma/z$$

という座標変換して z' をフラグメントシェーダ（プリミティブ合成ステージ）に渡せば、 z 方向にも自然に等間隔の線形補間になる。

（負の z に対して、 $\gamma > 0$ なら z' は z の単調増加関数である。）

$-1 \leq z' \leq +1$ となるように α と γ を調整する。

—

そういえば … 透視射影の x, y 成分も z の除算が入っていた：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} -\beta x/z \\ -\beta y/z \end{pmatrix}$$

そこで

ある点 $(x, y, z)^t$ に対して、

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} -\beta x/z \\ -\beta y/z \\ -\alpha - \gamma/z \end{pmatrix}$$

をプリミティブ組み立てステージに渡せばよい。(3成分全てに z の除算が入っていることに注意。) これを OpenGL では、次の2段階に分けて実行する。

透視射影変換の二つのステップ

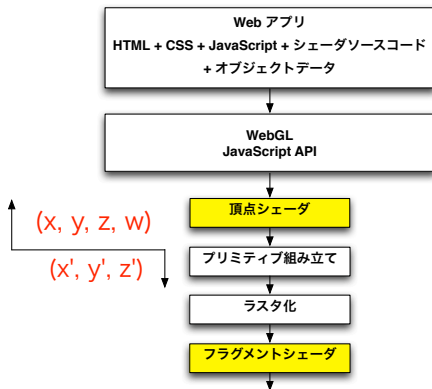
(1) 同次座標の行列演算。(この行列は一意には決まらない。ここで挙げるのは一つの例である。)

$$\begin{pmatrix} x^\dagger \\ y^\dagger \\ z^\dagger \\ w^\dagger \end{pmatrix} = \begin{pmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \alpha & \gamma \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(2) w 座標での割り算。これを透視除算という。プリミティブ組み立て時に実行される。

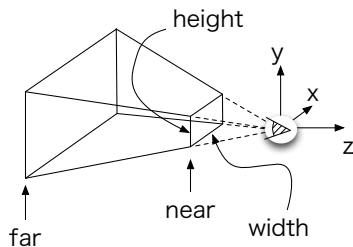
$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x^\dagger/w^\dagger \\ y^\dagger/w^\dagger \\ z^\dagger/w^\dagger \end{pmatrix}$$

パイプラインでの座標の処理



- $(x', y', z') = (x/w, y/w, z/w)$
- GPU が自動的に実行

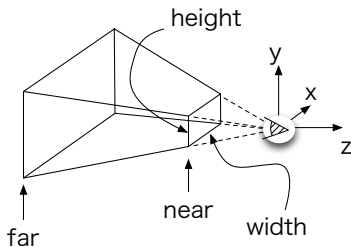
透視射影行列の設定



$$\begin{pmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \alpha & \gamma \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

パラメータ α, β, γ の決定 \Rightarrow 計算するのは簡単。

透視射影行列の設定



`mat4.perspective(fovy, aspect, near, far, projectionMatrix);`

`fovy` (field of view) は視野の高さ (y) 方向の角度。 `aspect` は縦横比。

上下と左右に非対象な frustum (視錐台) を作る場合は、

`mat4.frustum(left, right, bottom, top, near, far, projectionMatrix);`

(ステレオ画像を作る時には、右目用の画像と左目用の画像が必要である。このとき、視錐台は左右非対称。)

モデルビュー変換と射影変換

座標変換の実際

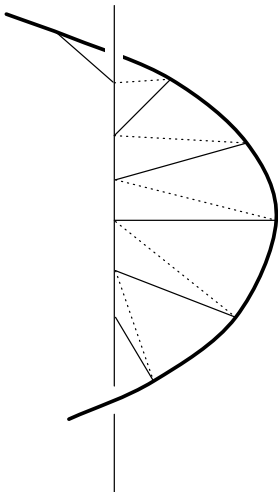
- モデルビュー変換
- 射影変換

について、あるオブジェクトに少しずつ座標変換をかけながらその効果を見る。

サンプル 3D 物体

らせん状の物体を考える

中心軸とそれにまきつく螺旋の間を面でつなぐ



webgl_sample_spiral_00.html

オブジェクトの構成部分

```
var dz = 0.1;
var pitch = 1.0;
var pof = 0;      // positionOffsetInFloats
var cof = 12;     // colorOffsetInBytes

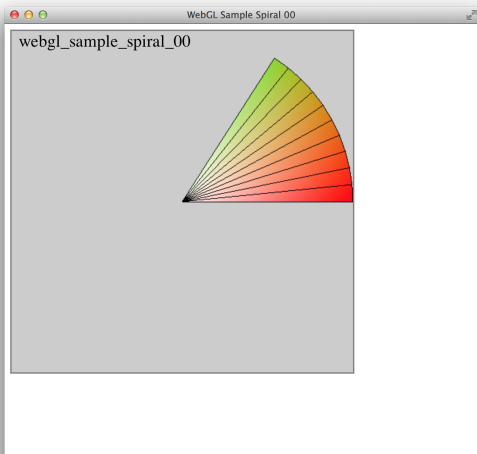
for (var k=0; k<Nz; k++) {
  var z = k*dz;
  positionView[ pof] = 0.0; // x
  positionView[1+pof] = 0.0; // y
  positionView[2+pof] = z;   // z
  colorView[ cof] = 255;    // R
  colorView[1+cof] = 255;   // G
  colorView[2+cof] = 255;   // B
  colorView[3+cof] = 255;   // A
  pof +=vertexSizeInFloats;
```

```
cof +=vertexSizeInBytes ;

phase = pitch*z;
positionView [  pof] = Math.cos(phase);           // x
positionView [1+pof] = Math.sin(phase);           // y
positionView [2+pof] = z;                          // z
colorView [   cof] = 255*Math.cos(phase);          // R
colorView [1+cof] = 255*Math.sin(phase);           // G
colorView [2+cof] = 255*Math.sin(phase*0.2);      // B
colorView [3+cof] = 255;                           // A
pof +=vertexSizeInFloats ;
cof +=vertexSizeInBytes ;
}
```

結果

何も変換していないので、見にくい。正規化ビューボリュームから外れている部分が見えない。



演習

- `webgl_sample_spiral_00.html` の `dz` (156 行目) を変えてその効果を見よう。

モデル変換

webgl_sample_spiral_01.html

モデルを移動・縮小・回転する行列 `modelViewMatrix` をつくり、頂点シェーダに送る。これも頂点属性 (`attribute`)。

```
mat4.identity(modelViewMatrix);
mat4.translate(modelViewMatrix, [0.8, -0.5, -0.8]);
mat4.scale(modelViewMatrix, [0.3, 0.3, 0.3]);
mat4.rotateX(modelViewMatrix, -Math.PI/3);
mat4.rotateY(modelViewMatrix, -Math.PI/3);
gl.uniformMatrix4fv(uniLocation,
                    false,
                    modelViewMatrix);
```

```
//--</new>--
```

```
gl.clear(gl.COLOR_BUFFER_BIT);
```

頂点シェーダ

modelViewMatrix も頂点の attribute

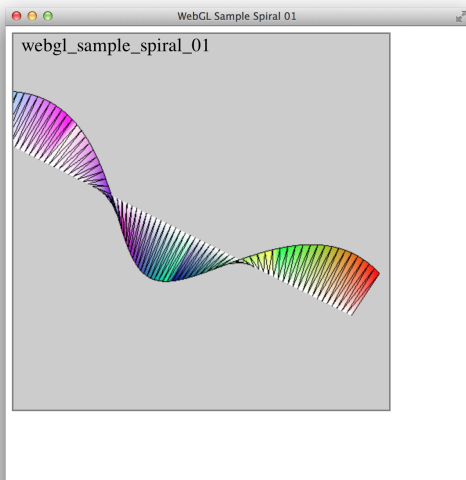
復習：全ての頂点に共通した attribute 値は uniform

```
<! -- new -->
```

```
<script id="shader-vs" type="x-shader/x-vertex">
  attribute vec3 aVertexPosition;
  attribute vec4 aVertexColor;
  uniform mat4 uMVMMatrix; //<--new
  varying vec4 vColor;

  void main() {
    vColor = aVertexColor;
    gl_Position = uMVMMatrix * vec4(aVertexPosition, 1.0); //<
    --new
  }
```

スナップショット webgl_sample_spiral_01.html



演習

- `webgl_sample_spiral_01.html` の回転軸と回転角を変更してその効果を見よう。

射影変換

webgl_sample_spiral_02.html

常に正規化ビューボリュームのサイズを意識してモデルを移動・縮小・回転を設定するよりも、ワールド座標（CG世界）の中にカメラを設定する、と考えた方が自然。

⇒ 射影変換。

mat4.perspective で projectionMatrix を作る

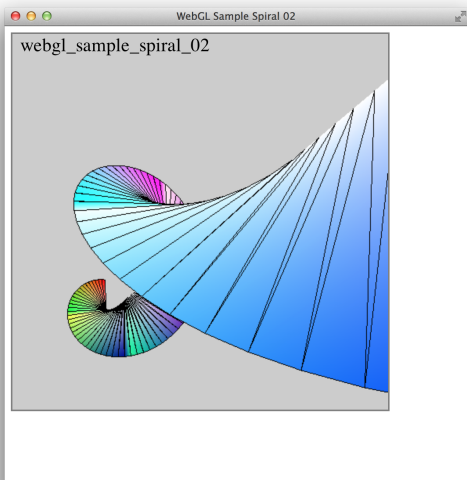
projectionMatrix を uniform として頂点シェーダに送る

シェーダではこの二つの行列をかける

```
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
```

このサンプルプログラムのこれ以外のポイントとして、デプステストとポリゴンオフセットがあるが説明は省略。

スナップショット webgl_sample_spiral_02.html



演習

- `webgl_sample_spiral_02.html` の射影行列の設定 `mat4.perspective(...)` を変更してその効果を見よう。

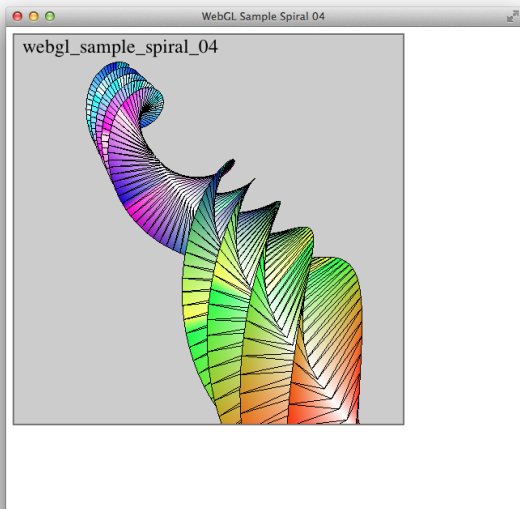
複数の変換行列の管理

描画する物体が複数の要素から構成されているときなど、カメラ（射影行列）は固定していて、モデルビュー行列は頻繁に更新してロードする場合には、モデルビュー行列のスタックを使うのが便利。（以前の OpenGL（OpenGL 1.x）には `glPushMatrix()`, `glPopMatrix()` があったが、今の OpenGL にはない。）

JavaScript には配列に `push` と `pop` のメソッドが備わっているのでこれを使えばいい。

練習

螺旋型の面を位相をずらして5枚描く



変換行列の push と pop

webgl_sample_spiral_04.html

```
}  
  modelViewMatrix = modelViewMatrixStack.pop();  
}  
  
//--new function  
function uploadModelViewMatrixToShader() {  
  gl.uniformMatrix4fv(uniLocation[1],  
    false,  
    modelViewMatrix);  
}
```

変換行列のシェーダへのロード

webgl_sample_spiral_04.html

```
function uploadProjectionMatrixToShader() {  
    gl.uniformMatrix4fv(uniLocation[0],  
                        false,  
                        projectionMatrix);  
}  
  
//--new function--  
function draw_a_spiral() {  
    // Draw triangles
```


描画

webgl_sample_spiral_04.html

}

```
function startup() {  
  canvas = document.getElementById("myGLCanvas");  
  gl = createGLContext(canvas);  
  
  shader_program = create_shader_program();
```

演習

- `webgl_sample_spiral_04.html` を自由に変更してみよう。