

# 3D グラフィックスにおけるリアルタイムな ニューラルスタイル変換

指導教員 西村まどか  
C1201517 須藤琢郎

2022 年 12 月 12 日

# 目次

<b>第 1 章</b>	<b>はじめに</b>	2
1.1	背景・目的 . . . . .	2
1.2	描画プロセス . . . . .	2
1.3	使用したライブラリ . . . . .	2
<b>第 2 章</b>	<b>ニューラルスタイル変換とは</b>	3
2.1	Gatys のニューラルスタイル変換 . . . . .	3
2.2	Johnson のニューラルスタイル変換 . . . . .	5
<b>第 3 章</b>	<b>3D モデルの描画</b>	7
3.1	モデルが描画されるまでの流れ . . . . .	7
3.2	メッシュの読み込み . . . . .	7
3.3	頂点情報の読み込み . . . . .	7
3.4	テクスチャ読み込み・SRV 作成 . . . . .	8
3.5	インデックスバッファ・バーテックスバッファ作成 . . . . .	8
3.6	頂点変換・コンスタントバッファ . . . . .	9
3.7	バーテックスシェーダー . . . . .	21
3.8	ピクセルシェーダー . . . . .	21
<b>第 4 章</b>	<b>変換ネットワークへの入力</b>	23
4.1	描画結果の取得 . . . . .	23
4.2	ネットワークへの入力 . . . . .	24
4.3	出力結果の描画 . . . . .	24
4.4	スタイル変換後の描画結果 . . . . .	25
<b>第 5 章</b>	<b>結論</b>	26
<b>参考文献</b>		27
<b>付録 A</b>	<b>使用した 3D モデルとバイナリ変換用プログラム</b>	28

# 第1章

## はじめに

### 1.1 背景・目的

畳み込みニューラルネットワーク (CNN) は、主に画像分類などで使われる機械学習のモデルの一つであり、調べていくうちにニューラルスタイル変換という CNN を使用して画像を加工する手法があることを知った。その手法は、Descriptive Methods と Generative Methods の二つに分類される。Gatys のニューラルスタイル変換では、Descriptive Methods が使われており、これは画像内の画素を反復的に直接更新する。Johnson のニューラルスタイル変換では、Generative Methods が使われており、これは生成モデルを反復的に最適化し順伝播により画像を生成する。Johnson のニューラルスタイル変換は、リアルタイムに画像を処理することが可能であり、3D グラフィックスにおいても実装することができるのではないかと考えた。

### 1.2 描画プロセス

1. RT(レンダーターゲット) と呼ばれる描画領域に 3D モデルを描画
2. RT からピクセル情報を取得
3. 取得したピクセル情報をあらかじめ訓練しておいた変換ネットワークに入力する。
4. モデルから出力されたピクセル情報をテクスチャに格納し、RT に上書きする形で描画。

### 1.3 使用したライブラリ

- Directx11(C/C++ 言語)  
Microsoft が提供しているゲーム・マルチメディア処理用の API(ライブラリ)。3D 描画や算術周りの機能が用意されており、今回 3D グラフィックスを処理するために使用する。
- Torch(Lua 言語)  
オープンソースの機械学習ライブラリであり、Lua プログラミング言語に基づくスクリプト言語が使用されている。Johnson の変換ネットワークを学習するために使用する。
- OpenCV  
インテルが開発・公開したオープンソースのコンピュータビジョン向けライブラリ。画像処理・画像解析および機械学習等の機能を持つ。Torch で学習した変換ネットワークを読み込むために使用した。

## 第2章

# ニューラルスタイル変換とは

### 2.1 Gatys のニューラルスタイル変換

ニューラルスタイル変換は、2015年 Leon Gatys 他によって提唱されたディープラーニングベースの画像加工のひとつ。ターゲット画像のコンテンツを維持した上で、リファレンス画像のスタイルをターゲット画像に適用する。



図 2.1 ニューラルスタイル変換の例

図 2.1 では、ゴッホの「星月夜」、葛飾北斎の「神奈川沖浪裏」がスタイルとみなされ、狼の写真がコンテンツとみなされる。画像分類では交差エントロピー誤差などの損失関数を使ってモデルの出力値と正解の値との解離を求め、その値を最小化するようにパラメーターを更新(逆伝播)した。一方で Gatys のスタイル変換では、元の画像のコンテンツを維持した上で、リファレンス画像のスタイルを取り入れるように損失関数を定義し、その値を最小化するように画像自体を更新する。損失関数は次のように定義される。

$$\begin{aligned} loss = & \text{distance}(\text{style(referenceimage)} - \text{style(generatedimage)}) \\ & + \text{distance}(\text{content(originalimage)} - \text{content(generatedimage)}) \end{aligned} \quad (2.1)$$

この作業では、VGG ネットワークと呼ばれる高性能の畳み込みニューラルネットワークによって学習された特徴表現を使用し、コンテンツとスタイルを個別に処理する。

$$L_{total} = \alpha L_{content} + \beta L_{style}$$

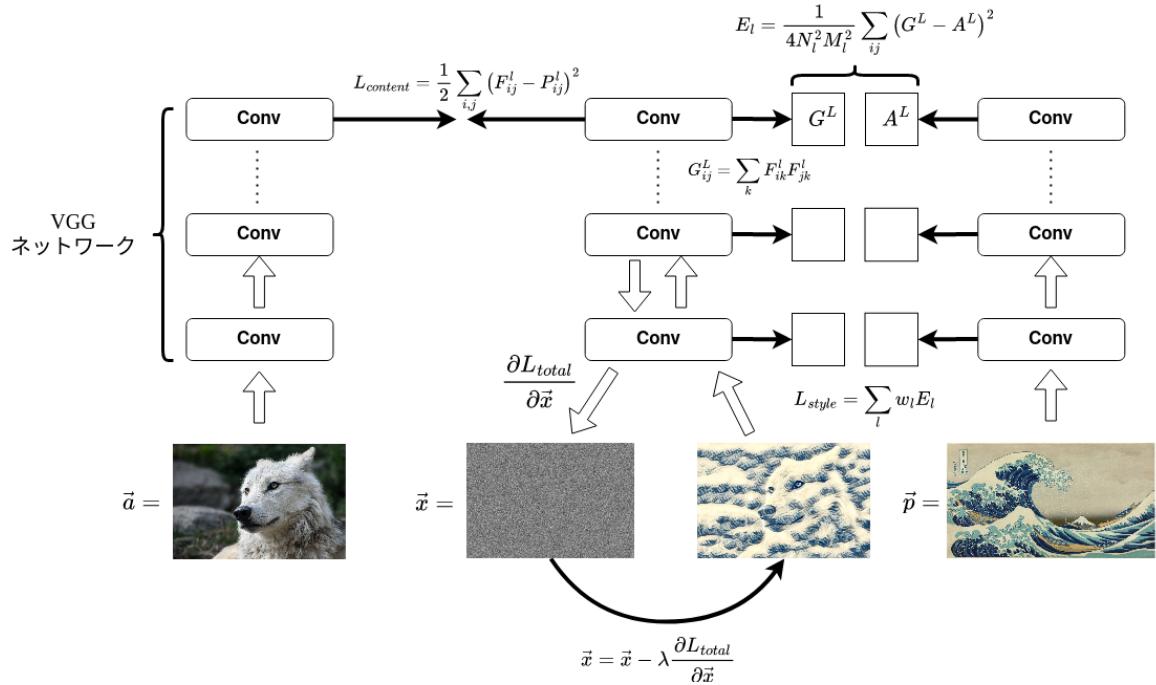


図 2.2 Gatys のニューラルスタイル変換

### 2.1.1 コンテンツの損失関数

ネットワークの入力側に近い層の活性化には、画像に関する「局所的な」情報が含まれていて、出力側に近くほど「大域的で抽象的な」情報が含まれるようになる。コンテンツ損失関数では、深い層(出力側)の表現を1層抜き出し、生成された画像との平均二乗誤差をとる。 $l$  層における生成画像を  $F_{ij}^l$ 、コンテンツ画像を  $P_{ij}^l$  とすると、

$$L_{content} = \frac{1}{2} \sum_{i,j} (F_{ij}^l - P_{ij}^l)^2 \quad (2.2)$$

これにより、CNN の出力側の層から見た場合に、生成された画像が元のターゲット画像と同じように見えることが保証される。

### 2.1.2 スタイルの損失関数

コンテンツの損失関数が使用するのはひとつの層だけだが、スタイルの損失関数は CNN の複数の層を使用する。個々の層でフィルタを適用した出力結果についてそれぞれの間で相関を取ることで、特徴ごとの類似度、つまりその画像らしさを表現できる。相関を取るには2つの特徴マップどうしの内積を取る。これはグラ

ム行列と呼ばれる。

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \quad (2.3)$$

グラム行列の結果とスタイル元画像の差分を取り平均二乗誤差を求める。特徴マップの数を  $N_l$ 、特徴マップのサイズを  $M_l$  とすると、

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (G_{ij}^l - A_{ij}^l)^2 \quad (2.4)$$

層ごとの損失の線形和をスタイルの合計損失とする。

$$L_{style} = \sum_l w_l E_l \quad (2.5)$$

入力側の層と出力側の層の両方で活性化の「相関関係」を同じに保つことでスタイルを維持する。

## 2.2 Johnson のニューラルスタイル変換

Gatys のスタイル変換ではイテレーションごとに最適化と更新が行われるため処理時間が遅いという短所がある。ゲームなどの入力とほぼ同時に処理が行われ、描画結果がフレーム毎に異なるような場面で Gatys のスタイル変換を行うのは適切でない。そこで Johnson のスタイル変換を使う。Johnson のスタイル変換は Gatys のスタイル変換の欠点である処理速度を大きく改善している。Gatys と Johnson の手法の違いは、Gatys のスタイル変換では損失関数を最小化するように画像自体を更新するのに対して、Johnson のスタイル変換では変換ネットワークと、損失ネットワークの2つのネットワークから構成され、損失ネットワークで抜き出したスタイル、コンテンツ、生成画像の特徴を基に損失関数を用意し、その値が最小化するように変換ネットワークの重みを更新する。推論では変換ネットワークに対して入力を順伝播させるだけなので Gatys のスタイル変換よりも処理速度が高速になる。Gatys のニューラルネットワークと Johnson のニューラルネットワークを比べた結果、 $712 \times 474$  の画像でのスタイル変換では Gatys のスタイル変換が 10 回のイテレーションで 770 秒かかったのに対して Johnson のスタイル変換 1 回の順伝播で約 3 秒で変換できた。

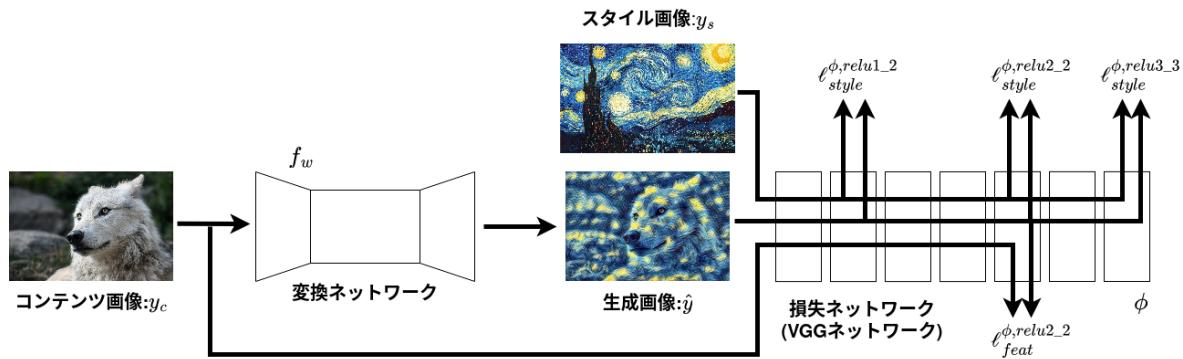


図 2.3 Johnson のニューラルスタイル変換

### 2.2.1 損失ネットワーク

基本的なアイディアは Gatys の手法と同じで損失ネットワーク  $\phi$  のコンテンツ損失  $l_{feat}^\phi$  とスタイル損失  $l_{style}^\phi$  の合計を最終的な損失とする。

### コンテンツ損失

高次元の一層からコンテンツ画像と生成画像の特徴マップを取り出し、平均二乗誤差をとる。入力  $\hat{y}$  に対する  $j$  番目の特徴マップを  $\phi_j(\hat{y})$ 、 $j$  番目の特徴マップのチャンネル数、高さ、横幅をそれぞれ  $C_j, H_j, W_j$  とする。

$$l_{feat}^{\phi} = \frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y_c)\|_2^2 \quad (2.6)$$

### スタイル損失

各層ごとにスタイル画像と生成画像のグラム行列を取って損失とする。

グラム行列:

$$G_j^{\phi}(x)_{c,c'} = \frac{1}{C_j H_j W_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \phi_j(x)_{h,w,c} \phi_j(x)_{h,w,c'} \quad (2.7)$$

スタイル損失:

$$l_{style}^{\phi} = \|G_j^{\phi}(\hat{y}) - G_j^{\phi}(y_s)\|_F^2 \quad (2.8)$$

スタイル損失では Frobenius ノルム  $F$  を取っている。Frobenius ノルムは行列の全成分の二乗和のルートを取ることで求めることができ、行列の大きさを表す。行列の全成分を一列に並べてベクトルとみなしたときのベクトルの長さ (L2 ノルム) と考えることもできる。

## 第3章

# 3D モデルの描画

### 3.1 モデルが描画されるまでの流れ

3D モデルはポリゴン(三角形・四角形・五角形などの多角形)をつなぎ合わせたメッシュによって作成される。一般的な3Dゲームモデルではポリゴン数が5千ポリゴンや10万ポリゴンという数になり、OBJやFBXなどの3Dファイルから頂点情報を抽出しポリゴンを形成する。ファイルの中にはモデルの作成・シェーディング(陰影をつけ立体感を与える技法)に必要な頂点座標・頂点番号・法線・接線・uv座標・マテリアル・テクスチャなどの情報が入っている。今回はモデルの表示に最低限必要な頂点座標・頂点番号・uv座標・テクスチャを使用する。

### 3.2 メッシュの読み込み

ソースコード 3.1 メッシュ読み込み

---

```

1 std::ifstream in(filename, std::ios::in | std::ios::binary);
2 in.read(reinterpret_cast<char*>(&m_numMesh), sizeof(m_numMesh));
3 in.read(reinterpret_cast<char*>(&m_numMaterial), sizeof(m_numMaterial));
4
5 mesh.resize(m_numMesh);
6 material.resize(m_numMaterial);
7
8 in.read(reinterpret_cast<char*>(&mesh[0]), mesh.size() * sizeof(Mesh));

```

---

はじめに ifstream 関数を使ってファイルを読み込む。第三引数にバイナリモードを指定しているのは読み込みを高速化するためにあらかじめ3Dファイルをバイナリ化したためである。次に read 関数を使ってメッシュ数・マテリアル数を読み込む。読み込んだメッシュ数・マテリアル数の分だけ mesh と material という変数名の動的配列を resize(メモリ領域確保)する。最後の read 関数では mesh 配列にメッシュ情報を詰め込んでいる。

### 3.3 頂点情報の読み込み

ソースコード 3.2 頂点情報読み込み

```

1 for (int i = 0; i < m_numMesh; i++)
2 {
3     piFaceBuffer[i].resize(mesh[i].m_numTriangles * 3);
4     pVertexBuffer[i].resize(mesh[i].m_numTriangles * 3);
5     in.read(reinterpret_cast<char*>(&piFaceBuffer[i][0]), piFaceBuffer[i].size() *
6         sizeof(int));
7     in.read(reinterpret_cast<char*>(&pVertexBuffer[i][0]), pVertexBuffer[i].size()
8         * sizeof(Vertex));
9 }

```

メッシュの数だけループを回し、piFaceBuffer と pVertexBuffer という二次元動的配列を resize しながらそれぞれ頂点番号と頂点情報を格納している。

### 3.4 テクスチャ読み込み・SRV 作成

ソースコード 3.3 テクスチャ読み込み・SRV 作成

```

1 for (int i = 0; i < m_numMaterial; i++)
2 {
3     if (LoadTexture(material[i].textureName, device, filepath, i) == false)
4     {
5         MessageBox(0, L"テクスチャ読み込み失敗", NULL, MB_OK);
6     }
7
8 }

```

マテリアルの数だけループを回し、material 配列に格納されたテクスチャのファイル名とモデルのパスが LoadTexture という自作関数に渡され、SRV が作成される。CPU 側で作成されたリソースはそのままでは使用することができず、View を介して GPU 側のメモリにコピーすることで描画することができる。SRV(Shader Resource View) も View のひとつでテクスチャを GPU に渡す役割をする。

### 3.5 インデックスバッファ・バーテックスバッファ作成

ソースコード 3.4 インデックス・バーテックスバッファ作成

```

1 D3D11_SUBRESOURCE_DATA sub_resource;
2 sub_resource.pSysMem = piFaceBuffer[i].data();
3 CreateBuffer(&buffer_desc, &sub_resource, &m_indexBuffer[i]);
4
5 D3D11_SUBRESOURCE_DATA sb;
6 sb.pSysMem = pVertexBuffer[i].data();
7 CreateBuffer(&bd, &sb, &m_vertexBuffer[i]);

```

頂点番号・頂点情報をそれぞれインデックスバッファ・バーテックスバッファに格納する。インデックスバッファはポリゴンを構成する頂点の順番を格納するためのバッファである。四角形は二つの三角形ポリゴン

で構成され六つの頂点が必要になるがインデックスバッファによって複数のポリゴンが共有する頂点を複数用意する必要がなくなる（四角形が4つの頂点で構成できる）。

## 3.6 頂点変換・コンスタントバッファ

格納した頂点番号・頂点情報がGPU側に渡ったとしてもそのままでは描画されない。頂点に対して座標変換を施す必要がある。モデルの持つローカル座標に対して、ワールド行列・ビュー行列・プロジェクション行列を作用することで、クリッピング座標に変換され3Dモデルとして描画される。

### 3.6.1 ワールド行列

ワールド行列は、ローカル座標系（モデル座標系）からワールド座標系へモデルを配置する際に使う行列である。通常座標の3次元にオブジェクトを平行移動するために必要な同時座標が含まれ、 $4 \times 4$ 行列になる。スケーリング、回転、移動の成分で構成されており、それぞれを  $S, R, T$  で表すと、

$$W = SRT \quad (3.1)$$

の順番で合成される。

#### スケーリング行列

スケーリング行列は、ローカル座標の任意の点を「原点を中心」として引き伸ばしたり縮めたりする。 $P(x, y, z)$  をローカル座標の位置、 $S$  をスケーリング行列とすると、

$$\begin{aligned} P'(x', y', z', 1) &= P \cdot S \\ &= (x, y, z, 1) \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= (S_x x, S_y y, S_z z, 1) \end{aligned} \quad (3.2)$$

ローカル座標の任意の点が定数倍されている。一般に、デフォルトのオブジェクトの大きさを変化するために使われる。

#### 回転行列

回転行列には、ローカル座標にある点を一つの軸を中心軸として回転させるという作用があり、どの軸で回転させるかによって行列の形が異なる。 $x, y, z$  それぞれの軸の行列は次のようになる。

$$\begin{aligned} P'(x', y', z', 1) &= P \cdot R_x \\ &= (x, y, z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= (x, y \cos \theta - z \sin \theta, y \sin \theta + z \cos \theta, 1) \end{aligned} \quad (3.3)$$

$$\begin{aligned}
 P'(x', y', z', 1) &= P \cdot R_y \\
 &= (x, y, z, 1) \begin{pmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
 &= (x \cos \theta + z \sin \theta, y, -x \sin \theta + z \cos \theta, 1)
 \end{aligned} \tag{3.4}$$

$$\begin{aligned}
 P'(x', y', z', 1) &= P \cdot R_z \\
 &= (x, y, z, 1) \begin{pmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
 &= (x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta, z, 1)
 \end{aligned} \tag{3.5}$$

### 移動行列

移動行列は、ローカル座標空間の任意の点をワールド空間内で平行移動させる作用を持つ。移動行列を  $T$  とすると、

$$\begin{aligned}
 P'(x', y', z', 1) &= P \cdot T \\
 &= (x, y, z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix} \\
 &= (x + T_x, y + T_y, z + T_z, 1)
 \end{aligned} \tag{3.6}$$

プログラム上では次のようにワールド行列を作っている。

ソースコード 3.5 ワールド行列作成

---

```

1 SimpleMath::Matrix m_world = XMMatrixScaling(x_scaling, y_scaling, z_scaling) *
    XMMatrixRotationX(XMConvertToRadians(x_rotation)) * XMMatrixRotationY(
    XMConvertToRadians(y_rotation)) * XMMatrixRotationZ(XMConvertToRadians(z_rotation)) *
    XMMatrixTranslation(x_translation, y_translation, z_translation);

```

---

### 3.6.2 ビュー行列

ビューベクトルは、ワールド座標に置かれたあらゆる物をカメラ座標系に変換させる行列である。DirectX の場合、カメラは原点にあって  $Z$  軸方向を向いていると定義されている（左手座標系）。例として、ワールドの  $(10, 20, 5)$  の位置にカメラを置き、 $Z$  軸方向を向かせているとする。これを行列で表現すると、

$$M_{cam} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 10 & 20 & 5 & 1 \end{pmatrix} \tag{3.7}$$

ビュー行列はこのカメラを原点に移動させて Z 軸方向を向かせればよいため、 $(-10, -20, -5)$  だけ平行移動させればよい。

$$V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -10 & -20 & -5 & 1 \end{pmatrix} \quad (3.8)$$

お互いを掛けると、

$$M_{cam} \cdot V = I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.9)$$

カメラ行列とビュー行列がお互いに逆行列の関係にある。つまり、カメラ自身のワールド行列が分かれれば、その逆行列を求めれば、ビュー行列になる。ビュー行列を求めるためにカメラのワールド行列を作成する。まずカメラの座標軸を作成する。カメラの座標軸を作成するためにはカメラの位置・注視点・上方向のベクトルが必要になる。例としてカメラ座標  $(100, 100, 100)$ 、注視点  $(250, 250, 250)$ 、上方向ベクトル  $(0, 1, 0)$  のカメラを原点とする座標軸の作成を行う。

$z$  軸ベクトル

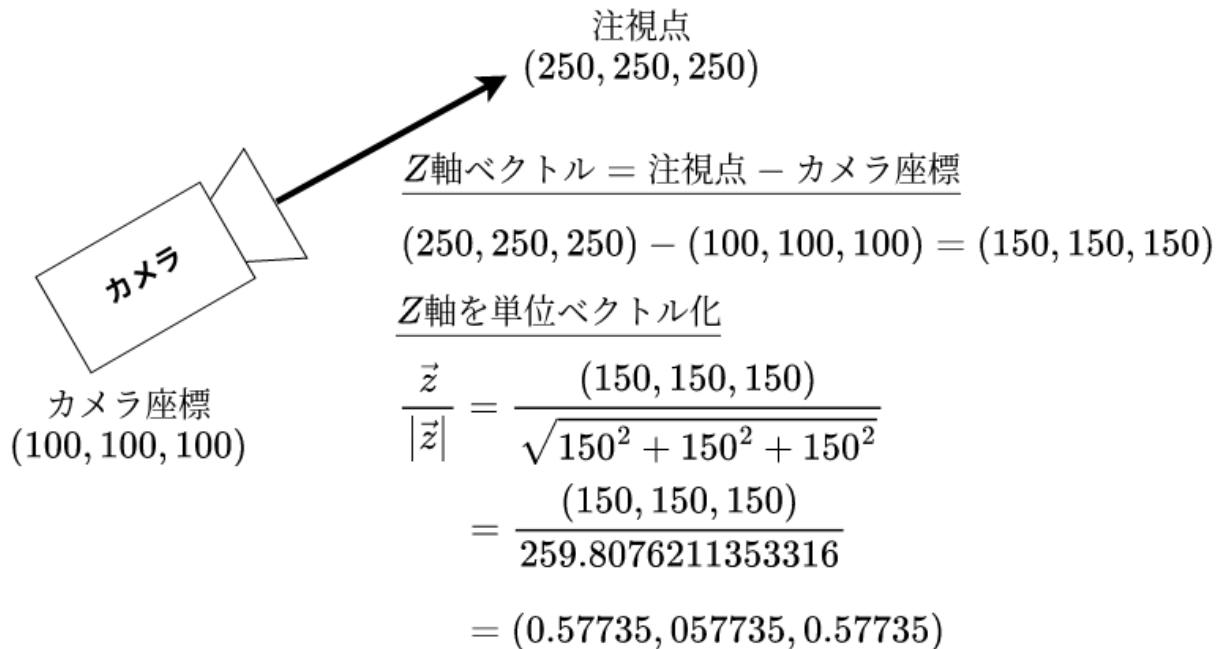
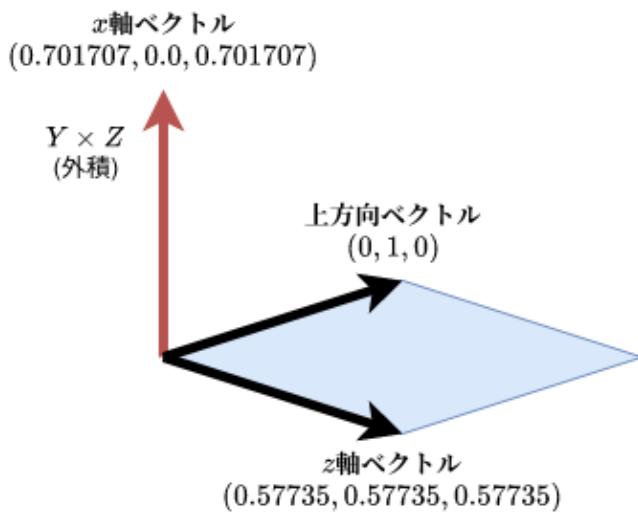


図 3.1  $z$  軸ベクトルの求め方

注視点の座標からカメラ座標を引いた結果を  $z$  軸として使用する。



$$\underline{x\text{軸ベクトル} = \text{上方向ベクトル} \times z\text{軸ベクトル}}$$

$$\vec{x} = \vec{Y} \times \vec{Z} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0.57735 \\ 0.57735 \\ 0.57735 \end{pmatrix} = \begin{pmatrix} 1 \times 0.57735 - 0 \times 0.57735 \\ 0 \times 0.57735 - 0 \times 0.57735 \\ 0 \times 0.57735 - 1 \times 0.57735 \end{pmatrix} = \begin{pmatrix} 0.57735 \\ 0 \\ -0.57735 \end{pmatrix}$$

#### x軸ベクトルを単位ベクトル化

$$\frac{\vec{x}}{|\vec{x}|} = \frac{(0.57735, 0, -0.57735)}{\sqrt{0.57735^2 + 0 + (-0.57735)^2}} = \frac{(0.57735, 0, -0.57735)}{0.8164962002361064} = (0.701707, 0, -0.701707)$$

図 3.2 x 軸ベクトルの求め方

#### × 軸ベクトル

カメラの上方向ベクトルと z 軸のベクトルの外積を求めてその結果を x 軸とする。二つのベクトルの外積を求めることで直行する（垂直に交わる）ベクトルを求めることができる。

#### y 軸ベクトル

Y 軸も X 軸ベクトルを求めた時と同じように、外積で求める。X 軸のベクトルと z 軸の外積を求めてその結果を Y 軸とする。

x 軸、y 軸、z 軸のベクトルを以下のように定めると、行列 R は次のようになる。

$$\begin{aligned} X \text{ 軸ベクトル} &= X = (X_x, X_y, X_z), \\ Y \text{ 軸ベクトル} &= Y = (Y_x, Y_y, Y_z), \\ Z \text{ 軸ベクトル} &= Z = (Z_x, Z_y, Z_z) \end{aligned}$$

とすると、

$$R = \begin{pmatrix} X_x & X_y & X_z & 0 \\ Y_x & Y_y & Y_z & 0 \\ Z_x & Z_y & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

この方法で作られるのは回転とスケーリングのみ含まれる行列で移動成分はまだ入っていないがひとまずこの行列の逆行列を作る。カメラのワールド行列は軸ベクトルから作られおり、各ベクトルは互いに直行していて、各行ベクトルの長さは 1 になっている。このような行列は直行行列と呼ばれ、その転置行列は逆行列になることが分かっている。転置行列を求めて普通に逆行列を求めるよりも計算コストを節約することができる。

$$R^T = \begin{pmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.10)$$

次にカメラの位置から移動行列を作る。カメラの位置を  $P(P_x, P_y, P_z)$  とすると、移動行列  $T$  は、

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -P_x & -P_y & -P_z & 1 \end{pmatrix} \quad (3.11)$$

位置を戻す必要があるため  $P$  をマイナスにする。 $R$  を転置した行列と移動行列  $T$  を合成すれば、ビュー行列  $V$  になる。

$$\begin{aligned} V = TR^T &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -P_x & -P_y & -P_z & 1 \end{pmatrix} \begin{pmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ -P'_x & -P'_y & -P'_z & 1 \end{pmatrix} \end{aligned} \quad (3.12)$$

プログラム上では `XMMatrixLookAtLH` 関数にカメラ座標・注視点・上方向ベクトルを引数として渡してビュー行列を作っている。

---

#### ソースコード 3.6 ビュー行列作成

---

```
1 SimpleMath::Matrix m_view = XMMatrixLookAtLH(eye, target, up);
```

---

### 3.6.3 プロジェクション行列

カメラが物体を映し出す空間領域をビューボリュームという。プロジェクション行列の目的は、ビューボリュームをクリップボリュームに潰すことである。クリップボリュームは、 $x$  成分の幅が -1 から 1、 $y$  成分の幅も -1 から 1、 $z$  成分は 0 から 1 の範囲である直方体である。

プロジェクション変換による投影方法には平行投影と透視投影の二つがある。

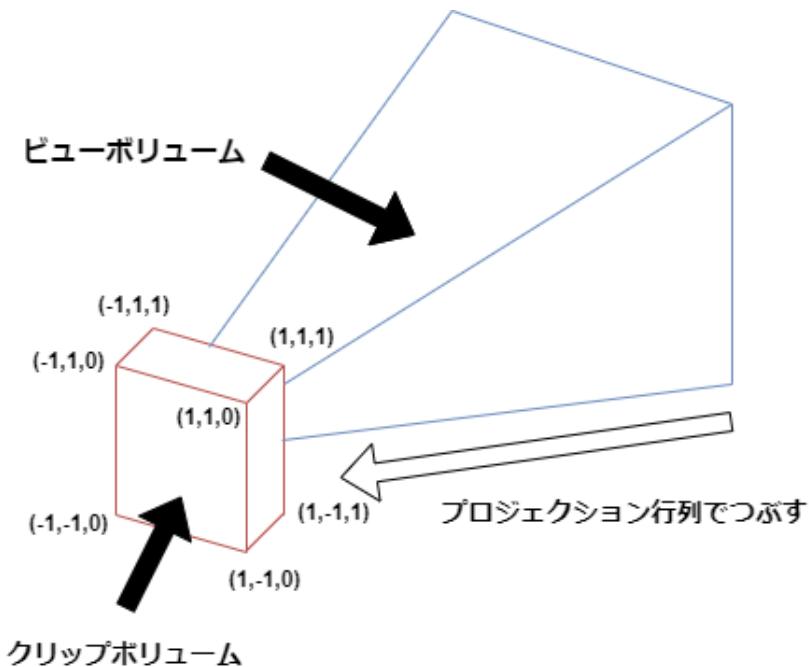


図 3.3 プロジェクション行列による投影

### 平行投影

平行投影のビューポリュームは直方体になる。投影線が平行なため、パースがかからず（遠近感がつかない）、遠くのものが小さくなったりせず、物体はその位置に関わらずに常にその物体のサイズで投影される。平行投影のビューポリューム内に点  $P(x, y, z)$  があるとする。変換される  $P$  点を  $P'(x', y', z')$  とする。 $P'$  を  $P$  とボリュームのパラメーターで表現できれば、変換行列を作ることができる。ビューポリュームのニアクリップ面（カメラ側の断面）の左 X 座標が  $L$ 、右 X 座標を  $R$ 、上 y 座標を  $T$ 、下 y 座標を  $B$ 、ニアクリップ面までの  $z$  軸上の距離を  $N$ 、ファークリップ面（一番奥の面）までの距離を  $F$  としてプロジェクション行列を作っていく。

ビューポリューム内の  $P$  点の  $x$  座標は、 $L \leq x \leq R$  と書ける。すべての項から  $L$  を引くと、

$$0 \leq x - L \leq R - L \quad (3.13)$$

続けて式を変形していく。

すべての項を  $R-L$  で割る。

$$0 \leq \frac{x - L}{R - L} \leq 1 \quad (3.14)$$

すべての項に 2 を掛ける。

$$0 \leq \frac{2x - 2L}{R - L} \leq 2 \quad (3.15)$$

すべての項から 1 を引く。

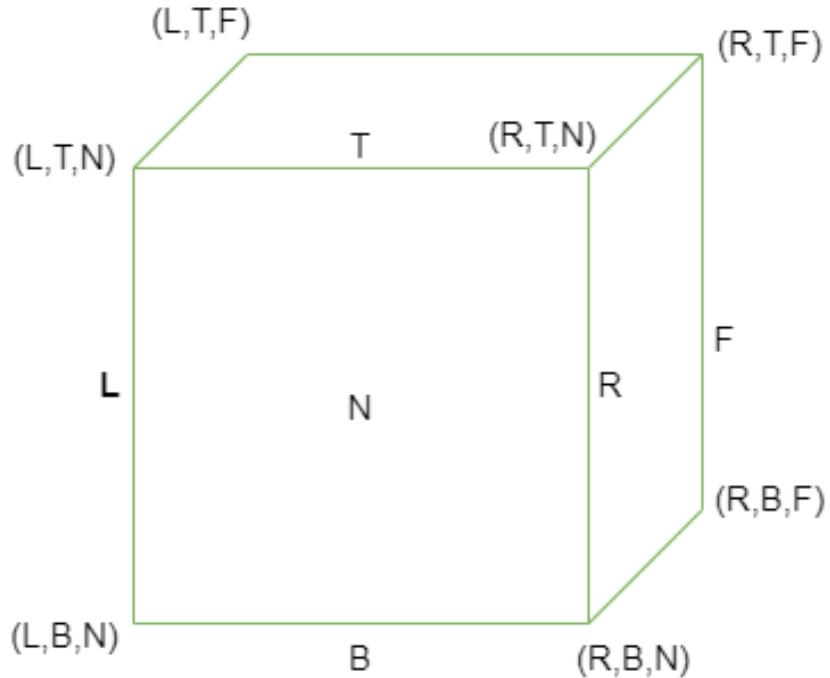


図 3.4 平行投影のビューポリューム

$$-1 \leq \frac{2x - 2L}{R - L} - 1 \leq 1 \quad (3.16)$$

クリップボリュームの範囲と同じになる。さらに式を整理する。

$$-1 \leq \frac{2x}{R - L} - \frac{R + L}{R - L} \leq 1 \quad (3.17)$$

この式より、クリップボリュームに変換された  $P'$  点の  $x'$  を次のように表せる。

$$x' = \frac{2x}{R - L} - \frac{R + L}{R - L} \quad (3.18)$$

$y'$  についても同様の手順で、表せる。

$$y' = \frac{2y}{T - B} - \frac{T + B}{T - B} \quad (3.19)$$

$z'$  はクリップボリュームの範囲が  $[0, 1]$  のため、 $x', y'$  と手順が異なる。 $N \leq z \leq F$  から始めて、すべての項から  $N$  を引く。

$$0 \leq z - N \leq F - N \quad (3.20)$$

$F - N$  で割る。

$$0 \leq \frac{z - N}{F - N} \leq 1 \quad (3.21)$$

クリップボリュームの範囲と同じになる。式を整理する。

$$0 \leq \frac{z}{F-N} - \frac{N}{F-N} \leq 1 \quad (3.22)$$

この式より、

$$z' = \frac{z}{F-N} - \frac{N}{F-N} \quad (3.23)$$

を得る。

これにより  $P'(x', y', z')$  を次のように表現できる。

$$P'(x', y', z') = \left( \frac{2x}{R-L} - \frac{R+L}{R-L}, \frac{2y}{T-B} - \frac{T+B}{T-B}, \frac{z}{F-N} - \frac{N}{F-N} \right) \quad (3.24)$$

この変換を行列にする。 $P(x, y, z) \times \text{行列} = P'(x', y', z')$  となるようなプロジェクション行列を作ればいいため、

$$\begin{pmatrix} \frac{2}{R-L} & 0 & 0 & 0 \\ 0 & \frac{2}{T-B} & 0 & 0 \\ 0 & 0 & \frac{1}{F-N} & 0 \\ \frac{R+L}{R-L} & -\frac{T+B}{T-B} & -\frac{N}{F-N} & 1 \end{pmatrix} \quad (3.25)$$

この行列はさらに簡単にすることができる。 $R$  と  $L$  は符号が逆で大きさが同じため、足すとゼロになる。 $T$  と  $B$  についても同様である。また、 $R-L$  はビューボリュームの横幅、 $T-B$  は縦幅になる。それらを  $w$  と  $h$  で表すと、

$$\begin{pmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{F-N} & 0 \\ 0 & 0 & -\frac{N}{F-N} & 1 \end{pmatrix} \quad (3.26)$$

となる。

### 透視投影

透視投影では、ビューボリュームは錐台になる。パースがかかるためリアルに映る。平行投影と同じようにビューボリューム内の点を  $P(x, y, z)$ 、クリップボリュームに変換される  $P$  点を  $P'(x', y', z')$ 、ビューボリュームのニアクリップ面の左 X 座標を  $L$ 、右 X 座標を  $R$ 、上 Y 座標を  $T$ 、下 Y 座標を  $B$ 、ニアクリップ面までの  $z$  軸上の距離を  $N$ 、ファークリップ面までの距離を  $F$  として変換行列を作る。

透視投影では  $z$  値の違いによって、 $x, y$  座標が変化するため変化の比率を求める。ビューボリューム内の点  $P$  から視点までを結ぶ直線と、ニアクリップ面との交点を点  $P'$  とする。 $P$  と  $P'$  それぞれから、 $z$  軸に垂線を下ろすと相似な三角形が 2 つできる。

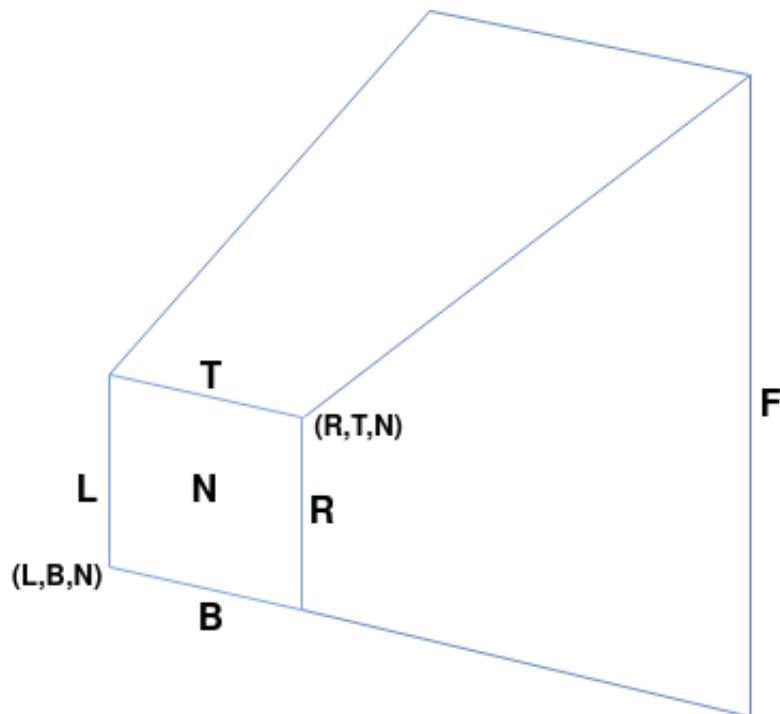


図 3.5 透視投影のビュー・ボリューム

長い方の垂線  $L_1$  は次のように書ける。

$\vec{OP} = p, \vec{OZ} = z$  とおくと、

$$\begin{aligned}
 \vec{L_1} &= p - z \\
 &= (x, y, z) - (0, 0, z) \\
 &= (x, y, 0) \\
 |\vec{L_1}| &= \sqrt{x^2 + y^2}
 \end{aligned} \tag{3.27}$$

相似な三角形であるため、辺の長さは比率で導くことができる。そのため短い方の垂線  $L_2$  は次のように書ける。

$$\begin{aligned}
 L_2 &= \frac{N}{z} L_1 = \frac{N}{z} \sqrt{x^2 + y^2} \\
 &= \sqrt{\left(\frac{N}{z}\right)^2 (x^2 + y^2)} \\
 &= \sqrt{\left(\frac{Nx}{z}\right)^2 + \left(\frac{Ny}{z}\right)^2}
 \end{aligned} \tag{3.28}$$

$L_1$  と  $L_2$  の最終式を見比べると、

$$L_1 = \sqrt{x^2 + y^2}, L_2 = \sqrt{\left(\frac{Nx}{z}\right)^2 + \left(\frac{Ny}{z}\right)^2} \tag{3.29}$$

となっていて、この2つから  $P'$  の  $x' = \frac{Nx}{z}, y' = \frac{Ny}{z}$  となることがわかる。

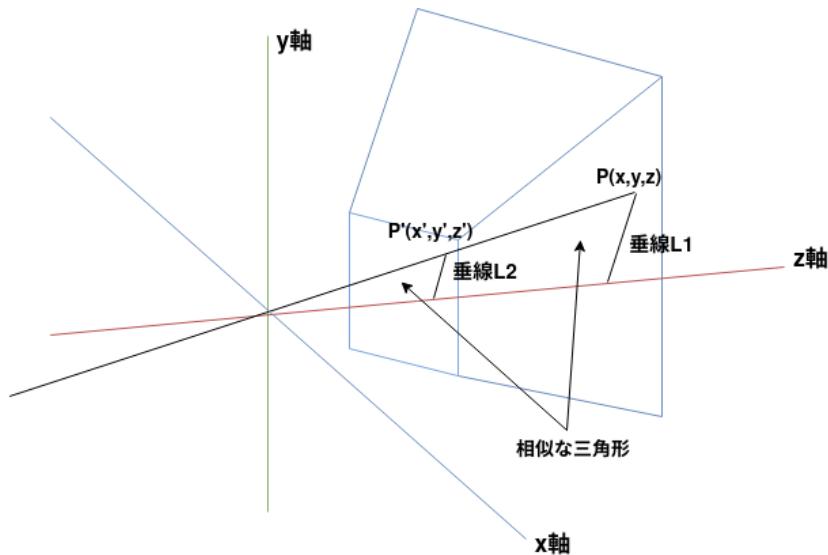


図 3.6 ビューボリューム内の相似な三角形

$x, y$  座標の変化分は  $\frac{N}{z}$  という単純な比率で表せる。透視投影は平行投影での変換  $+x, y$  の変化率になる。  
平行投影での

$$\begin{aligned} x' &= \frac{2x}{R-L} - \frac{R+L}{R-L} \\ y' &= \frac{2y}{T-B} - \frac{T+B}{T-B} \end{aligned} \quad (3.30)$$

の中での  $x$  を  $\frac{N}{z}x$  に、 $y$  を  $\frac{N}{z}y$  に置き換える。

$$\begin{aligned} x' &= \left(\frac{2}{R-L}\right)\frac{N}{z}x - \frac{R+L}{R-L} \\ y' &= \left(\frac{2}{T-B}\right)\frac{N}{z}y - \frac{T+B}{T-B} \end{aligned} \quad (3.31)$$

両辺に  $z$  を掛ける。

$$\begin{aligned} x'z &= \frac{2N}{R-L}x - \frac{R+L}{R-L}z \\ y'z &= \frac{2N}{T-B}y - \frac{T+B}{T-B}z \end{aligned} \quad (3.32)$$

残る  $z'z$  は  $z'z = az + b$  という式から求めることができる。未知数が  $a$  と  $b$  の二個なので、2つの式を見つけることができれば、 $a$  と  $b$  が求まる。

$z = N$  のときに  $z' = 0$

$z = F$  のときに  $z' = 1$

となることが分かっているためこの2つの状態をそれぞれ  $z'z = az + b$  に代入する。

$$\begin{aligned} zz' &= az + b(z' = 0, z = N) \\ 0N &= aN + b \\ 0 &= aN + b \end{aligned} \tag{3.33}$$

$$\begin{aligned} zz' &= az + b(z' = 1, z = F) \\ 1F &= aF + b \\ F &= aF + b \end{aligned} \tag{3.34}$$

$$\begin{aligned} b &= -aN \\ F &= aF - aN \\ F &= a(F - N) \\ a &= \frac{F}{F - N} \\ b &= -\frac{FN}{F - N} \end{aligned} \tag{3.35}$$

$a$  と  $b$  を  $z' = az + b$  に代入する。

$$z'z = \frac{F}{F - N}z - \frac{FN}{F - N} \tag{3.36}$$

この時点で 3 つの式が揃ったため、

$$P(x, y, z, 1) \times \text{行列} = P'(x'z, y'z, z'z, w'z) \tag{3.37}$$

となる行列を作る。

$$\begin{pmatrix} \frac{2N}{R - L} & 0 & 0 & 0 \\ 0 & \frac{2N}{T - B} & 0 & 0 \\ 0 & 0 & \frac{F}{F - N} & 1 \\ 0 & 0 & -\frac{FN}{F - N} & 0 \end{pmatrix} \tag{3.38}$$

平行投影のときと同じく、シンプルな形にすることができる。

$$\begin{pmatrix} \frac{2N}{w} & 0 & 0 & 0 \\ 0 & \frac{2N}{h} & 0 & 0 \\ 0 & 0 & \frac{F}{F - N} & 1 \\ 0 & 0 & -\frac{FN}{F - N} & 0 \end{pmatrix} \tag{3.39}$$

プログラム上では `XMMatrixPerspectiveFovLH` 関数に画角、アスペクト比、ニアクリップ面までの距離、ファークリップ面までの距離を引数として渡してプロジェクション行列を作っている。

### ソースコード 3.7 プロジェクション行列作成

---

```
1 SimpleMath::Matrix m_proj = XMMatrixPerspectiveFovLH(XM_PIDIV4, aspect, 1, 10000.0f);
```

---

### 3.6.4 コンスタントバッファ

ワールド行列、ビュー行列、プロジェクション行列が用意出来たら行列を GPU 側に送るためにコンスタントバッファを作る。コンスタントバッファは行列に限らず、ベクトルやスカラーなどの定数を GPU に送ることができる。コンスタントバッファを作成する前に行列を格納する構造体を作成する。

ソースコード 3.8 構造体作成

---

```

1     struct Constant_Buffer
2     {
3         XMFLOAT4X4 world;
4         XMFLOAT4X4 view;
5         XMFLOAT4X4 proj;
6     };

```

---

コンスタントバッファ作成の際には構造体のサイズ、GPU への書き込みを行うのか読み込みかを行うのかの設定、定数の更新設定などを指定する。今回は GPU への書き込みで毎フレームごとに定数の中身も変えたいので下記のような設定にした。

ソースコード 3.9 コンスタントバッファ作成

---

```

1 //定数バッファの作成
2 D3D11_BUFFER_DESC cb;
3 cb.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
4 cb.ByteWidth = sizeof(Constant_Buffer);
5 cb.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
6 cb.MiscFlags = 0;
7 cb.StructureByteStride = 0;
8 cb.Usage = D3D11_USAGE_DYNAMIC;
9 (device->CreateBuffer(&cb, NULL, &pConstantBuffer));

```

---

GPU に対してコンスタントバッファの内容を書き込んでいく。ワールド行列、ビュー行列、プロジェクション行列の転置を取り、構造体に格納する。行列の転置を取り意味はシェーダーでの行列は列優先になり要素やベクトルの掛け合わせの順番が変わるからである。Map 関数を実行し、データを GPU のメモリへ転送する許可を取得する。memcpy や memcpy\_s などのメモリコピー関数を使ってビデオカード上のメモリにデータをコピーする。

ソースコード 3.10 GPU へ書き込み

---

```

1 D3D11_MAPPED_SUBRESOURCE msr;
2 Constant_Buffer cb;
3 XMStoreFloat4x4(&cb.world, XMMatrixTranspose(world));
4 XMStoreFloat4x4(&cb.view, XMMatrixTranspose(view));
5 XMStoreFloat4x4(&cb.proj, XMMatrixTranspose(proj));
6
7 context->Map(pConstantBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &msr);
8 memcpy_s(msr.pData, msr.RowPitch, (void*)&cb, sizeof(cb));
9 context->Unmap(pConstantBuffer, 0);

```

---

シェーダー側 (hlsl) でコンスタントバッファを受け取る。

ソースコード 3.11 コンスタントバッファ受け取り

```
1 cbuffer cb : register(b0)
2 {
3     float4x4 gWorld;
4     float4x4 gView;
5     float4x4 gProj;
6 }
```

### 3.7 バーテックスシェーダー

コンスタントバッファによって受け取った行列を使い全頂点に対して座標変換を行う。座標変換をするにはバーテックスシェーダーを用いる。バーテックスシェーダーでは頂点の位置、色、テクスチャ座標などを決定する。`mul` 関数を使って頂点座標に対して、ワールド行列、ビュー行列、プロジェクション行列の順番で乗算を行う。

ソースコード 3.12 座標変換

```
1 output.pos = mul(pos, gWorld);
2 output.pos = mul(output.pos, gView);
3 output.pos = mul(output.pos, gProj);
```

バーテックスシェーダーによって加工された頂点情報はピクセルシェーダーに送られる。

### 3.8 ピクセルシェーダー

ピクセルシェーダーでは CPU 側からもらったテクスチャや定数などの情報をもとに最終的な色を決定するシェーダーである。今回は SRV を介してテクスチャを転送している。また、テクスチャからピクセルを取り出すためのサンプラーも用意する。

ソースコード 3.13 テクスチャ・サンプラー受け取り

```
1 Texture2D Texture : register(t0);
2 SamplerState Sampler : register(s0);
```

uv 座標をもとに、テクスチャからサンプリングを行い、取得したピクセルをそのまま最終的な出力の色とする。

ソースコード 3.14 サンプリング

```
1 float4 texel = Texture.Sample(Sampler, input.uv);
2 return texel;
```



図 3.7 描画結果

## 第4章

# 変換ネットワークへの入力

### 4.1 描画結果の取得

3D モデルを描画することができたので、描画結果を取得してこれを変換ネットワークへ入力する。画面に何らかの絵を表示するとき、直接表画面に絵は描画されずバックバッファという画面の裏側を表す領域に書き込みが行われる。バックバッファを表画面（フロントバッファ）に切り替えることでちらつきが防止され、表示が滑らかになる。バックバッファに書き込まれた情報を取得するためにはフロントバッファとバックバッファの切り替えを制御するスワップチェーンから `GetBuffer` 関数を呼び出す。

---

```
1 swapchain->GetBuffer(0, __uuidof(ID3D11Texture2D), reinterpret_cast<void**>(&SourceTex));
)
```

---

読み取り専用テクスチャを作成し、バックバッファの内容をコピーする。

ソースコード 4.1 テクスチャへバックバッファの内容をコピー

---

```
1 ID3D11Texture2D* pCopyTexture = NULL;
2
3 D3D11_TEXTURE2D_DESC description;
4 inputTex->GetDesc(&description);
5 description.BindFlags = 0;
6 description.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
7 description.Usage = D3D11_USAGE_STAGING;
8 device->CreateTexture2D(&description, NULL, &pCopyTexture);
9
10 context->CopyResource(pCopyTexture, inputTex);
```

---

`Map` 関数を実行して GPU からの読み取り許可を取得する。OpenCV が提供する `cv::Mat` クラスを利用し、ピクセル情報をテクスチャからコピーする。

ソースコード 4.2 ピクセル取得

---

```
1 D3D11_MAPPED_SUBRESOURCE ResourceDesc;
2 context->Map(pCopyTexture, NULL, D3D11_MAP_READ, 0, &ResourceDesc);
3
4 cv::Mat img = cv::Mat::ones(720, 1280, CV_8UC4);
5 size_t buffer_size = ResourceDesc.RowPitch * height;
```

---

```

6 CopyMemory(&img.data[0], ResourceDesc.pData, buffer_size);
7
8 context->Unmap(pCopyTexture, 0);

```

---

## 4.2 ネットワークへの入力

Torch で学習させた変換ネットワークを読み込む。推論処理で GPU を使いたいため、CUDA バックエンドを有効化する。

ソースコード 4.3 ネットワークの読み込み

---

```

1 cv::String modelPathN = "Data/models/eccv16/starry_night.t7";
2 cv::String modelPathW = "Data/models/eccv16/the_wave.t7";
3 cv::String modelPathM = "Data/models/eccv16/la_muse.t7";
4 cv::String modelPathV = "Data/models/eccv16/composition_vii.t7";
5 net = cv::dnn::readNetFromTorch(modelPathN);
6 net.setPreferableBackend(cv::dnn::DNN_BACKEND_CUDA);
7 net.setPreferableTarget(cv::dnn::DNN_TARGET_CUDA);

```

---

テクスチャから取得した情報を変換ネットワークへ入力できるようにする。まず、チャンネル数（カラー数）を 4(RGBA) から 3(BGR) に変更する。続いて、blobFromImage 関数を使用して OpenCV の Mat 型を dnn モジュールの入力として使用できるように型変換を行う。

ソースコード 4.4 入力できる形へ変換

---

```

1 cv::cvtColor(img, bgr_img, cv::COLOR_RGBA2BGR);
2 blob = cv::dnn::blobFromImage(bgr_img, 1.0, cv::Size(width, height), mean, false, false
);

```

---

変換ネットワークに入力をセットして、順伝播を行う。

ソースコード 4.5 順伝播

---

```

1 net.setInput(blob);
2 cv::Mat prob = net.forward();

```

---

## 4.3 出力結果の描画

出力されたデータを元の型へ変換し、チャンネル数を 3(BGR) から 4(RGBA) へ変更する。Map 関数を実行し、出力用のテクスチャへ書き込みを行う。

ソースコード 4.6 変換内容をテクスチャへコピー

---

```

1 cv::dnn::imagesFromBlob(prob, result);
2 D3D11_MAPPED_SUBRESOURCE msr;
3 context->Map(pTexture.Get(), 0, D3D11_MAP_WRITE_DISCARD, 0, &msr);
4
5 cv::cvtColor(result[0], copyimg, cv::COLOR_BGR2RGBA);

```

```
6 size_t img_size = msr.RowPitch * height;
7 CopyMemory(msr.pData, copyimg.data, img_size);
8
9 context->Unmap(pTexture.Get(), 0);
```

このテクスチャを貼り付けるための四角形ポリゴンを用意し、頂点変換を施すことでスタイル変換された描画結果が表示される。

#### 4.4 スタイル変換後の描画結果

次はそれぞれ、ゴッホの「星月夜」、葛飾北斎の「神奈川沖浪裏」をスタイル画像として学習させた変換モデルの出力結果である。

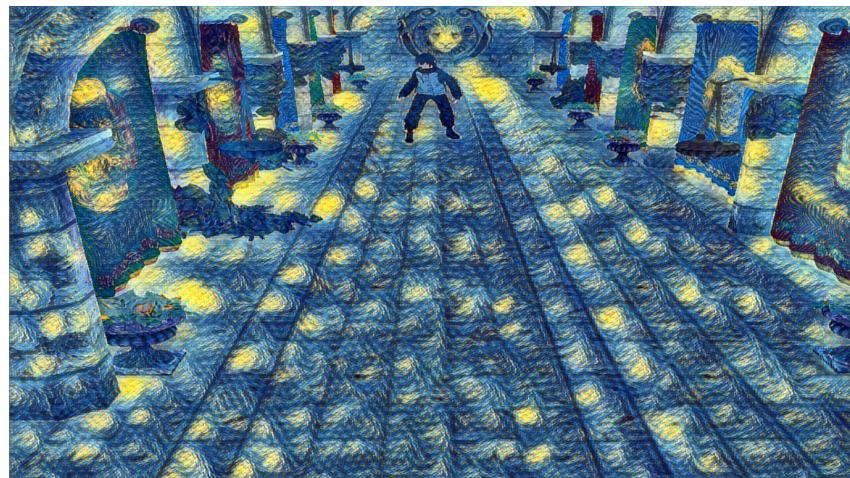


図 4.1 星月夜をスタイル画像として学習させた変換ネットワークの出力結果



図 4.2 神奈川沖浪裏をスタイル画像として学習させた変換ネットワークの出力結果

## 第 5 章

### 結論

機械学習を用いることで 3D グラフィックスの表現の幅を広げることができる実感した。それと同時に多くの課題も発見した。リアルタイムでのニューラルスタイル変換を実装するため、ある程度コード数を減らすようにしたが、テクスチャの解像度、GPU の性能などによって FPS(1 秒あたりのコマ数) が左右される。これを解消するためには変換ネットワークの精度を保ったままサイズを小さくできるような新しいアルゴリズムを考える必要がある。また、多くの重みを読み込むためメモリ使用率が高くなってしまうという問題もあった。

今後は、姿勢推定・深度推定・超解像など 3D グラフィックで応用がききそうなモデルを試す。特に姿勢推定はカメラからの映像や動画から人間の関節の場所を特定することができ、これを 3D モデルに適用することができればメタバースなどの 3D 仮想空間でのさらなる自己投影につながるのではないか。

## 参考文献

- [1] Leon A.Gatys. ImageStyle Transfer Using Convolutional Neural Networks. 2016. p2414-2423.  
[https://www.cv-foundation.org/openaccess/content\\_cvpr\\_2016/papers/Gatys\\_Image\\_Style\\_Transfer\\_CVPR\\_2016\\_paper.pdf](https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf). 2022.
- [2] Justin Johnson. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. 2016.  
<https://cs.stanford.edu/people/jcjohns/papers/eccv16/JohnsonECCV16.pdf>. 2022.
- [3] François Chollet. Python によるディープラーニング. マイナビ出版. 2022.
- [4] 鎌田茂雄. SHADER GURU with Direct3D10/11. Northbrain. 2022
- [5] 宮川雅巳. 水野真治. 矢島安敏. 経営工学の数理 II. 朝倉書店. 2022

## 付録 A

# 使用した 3D モデルとバイナリ変換用プログラム

3D グラフィックスの実装では Sponza.obj と character.fbx という二つの 3D モデルを使った。Sponza.obj の中には頂点座標・uv 座標・法線・ポリゴン情報・マテリアルなどの情報が含まれている。それらは下記のようにそれぞれ v・vt・vn・f・mtlib という文字の後に格納されている。

ソースコード A.1 頂点座標

---

```

1 v -273.471222 223.719650 141.117310
2 v -299.607239 219.111008 167.253311
3 v -238.697922 219.111008 167.254211
4 v -238.697922 223.736984 141.018967
5 v -238.697922 215.930191 178.787857
6 v -311.552643 215.857513 179.198746
7 v -299.411072 217.998474 167.057144
8 v -238.697922 217.998474 167.058044
9 v -311.749939 216.969849 179.396011
10 v -238.697922 217.042725 178.984024
11 v -164.461884 219.111008 167.254211
12 v -164.461884 223.699966 141.228943
13 v -164.461884 217.998474 167.058044
14 v -164.461884 215.856140 179.207840
15 v -164.461884 216.968658 179.404007
16 v -238.697922 222.624466 140.822784
17 v -238.696808 228.704407 106.342903
18 v -233.027420 229.704117 100.673523
19 v -164.461884 229.704117 100.672203
20 v -164.461884 222.587433 141.032776
21 v -233.223587 230.816635 100.869667
22 v -164.461884 230.816635 100.868362

```

---

ソースコード A.2 uv 座標

---

```

1 vt 0.552029 0.021754
2 vt 0.571469 0.052047

```

---

---

```

3 vt 0.574728 0.021754
4 vt 0.594963 0.052047
5 vt 0.597388 0.021754
6 vt 0.618577 0.052047
7 vt 0.541693 0.082340
8 vt 0.566380 0.082340
9 vt 0.590908 0.082340
10 vt 0.615278 0.082340
11 vt 0.639567 0.082340
12 vt 0.585025 0.112632
13 vt 0.610547 0.112632
14 vt 0.635672 0.112632
15 vt 0.660597 0.112632
16 vt 0.576994 0.142925
17 vt 0.604107 0.142925
18 vt 0.630464 0.142925

```

---

## ソースコード A.3 法線

---

```

1 vn -0.0293 0.0108 -0.9995
2 vn -0.0399 0.0422 -0.9983
3 vn -0.1330 0.1068 -0.9853
4 vn 0.0067 -0.0091 -0.9999
5 vn -0.0309 0.0075 -0.9995
6 vn -0.0169 0.0380 -0.9991
7 vn -0.1026 0.1136 -0.9882
8 vn 0.0163 -0.0160 -0.9997
9 vn -0.0287 0.0035 -0.9996
10 vn -0.0687 0.1156 -0.9909
11 vn -0.0577 0.0190 -0.9982
12 vn -0.0226 0.0108 -0.9997
13 vn -0.0051 0.0036 -1.0000
14 vn -0.0005 0.0003 -1.0000
15 vn -0.0151 0.0051 -0.9999
16 vn -0.0011 0.0010 -1.0000
17 vn -0.0000 0.0000 -1.0000
18 vn 0.0098 -0.0025 -0.9999
19 vn 0.0001 -0.0000 -1.0000
20 vn 0.0090 -0.0031 -1.0000
21 vn 0.0478 -0.0143 -0.9988
22 vn 0.0032 -0.0024 -1.0000
23 vn 0.0191 -0.0096 -0.9998

```

---

## ソースコード A.4 ポリゴン情報

---

```

1 f 4756/6371/3849 4504/6316/3555 4503/6372/3850
2 f 4758/6373/3851 4760/6374/3852 4761/6375/2814

```

```

3 f 4761/6375/2814 4763/6376/3853 4762/6377/2977
4 f 4764/6378/3854 4766/6379/2817 4767/6380/2818
5 f 4767/6380/2818 4769/6381/2819 4768/6382/2820
6 f 4770/6383/3855 4772/6384/3856 4773/6385/3857
7 f 4775/6386/2827 4777/6387/2828 4774/6388/2829
8 f 4778/6389/3858 4780/6390/3859 4781/6391/3860
9 f 4783/6392/2836 4785/6393/2837 4782/6394/2838
10 f 4786/6395/3861 4788/6396/3862 4789/6397/3863
11 f 4791/6398/2845 4793/6399/2846 4790/6400/2847
12 f 4794/6401/3864 4796/6402/3865 4797/6403/3410
13 f 4799/6404/2854 4801/6405/2855 4798/6406/2856
14 f 4802/6407/3866 4804/6408/3867 4805/6409/3868
15 f 4807/6410/2863 4809/6411/2864 4806/6412/2865
16 f 4810/6413/3869 4812/6414/3870 4813/6415/3871
17 f 4815/6416/2872 4817/6417/2873 4814/6418/2872
18 f 4818/6419/2881 4820/6420/3872 4821/6421/3873
19 f 4822/6422/2890 4824/6423/3874 4825/6424/3875
20 f 4826/6425/2898 4828/6426/3876 4829/6427/3877
21 f 4830/6428/2906 4832/6429/3878 4833/6430/3879
22 f 4834/6431/2915 4836/6432/3880 4837/6433/3881
23 f 4838/6434/2924 4840/6435/3882 4841/6436/3883

```

---

ソースコード A.5 マテリアル

---

```
1 mtllib sponza.mtl
```

---

読み取りを高速化するために `obj_to_bin.cpp` というソースコードを作成し、`obj` ファイルをバイナリファイルに変換した。ソースコード全体の流れは `OBJ` ファイルから情報を読み取り、それらを配列に格納し、バイナリファイルへと書き込みを行うという形になる。

---

ソースコード A.6 格納用の配列とカウント用の変数

---

```

1 struct Vertex
2 {
3     XMFLOAT3 pos;
4     XMFLOAT3 normal;
5     XMFLOAT2 uv;
6 };
7
8 struct Material
9 {
10     CHAR name[256];
11     CHAR normaltextureName[256];
12     CHAR textureName[256];
13     XMFLOAT4 Ka;
14     XMFLOAT4 Kd;
15     XMFLOAT4 Ks;
16 };

```

```

17
18 struct Mesh
19 {
20     DWORD m_numTriangles;
21     DWORD faceCount;
22     CHAR materialname[200];
23 };
24
25 std::vector<Mesh> mesh;
26 std::vector<Material> material;
27 std::vector<std::vector<Vertex>> pVertexBuffer;
28 std::vector<std::vector<int>> piFaceBuffer;
29
30 DWORD vertCount = 0;
31 DWORD vnormalCount = 0;
32 DWORD vuvCount = 0;
33 DWORD faceCount = 0;
34 DWORD meshCount = 0;

```

---

はじめにモデルの情報を格納するための配列と頂点・法線・uv・ポリゴンの数をカウントする変数を用意する。

ソースコード A.7 カウント

```

1 FILE* fp = NULL;
2 fopen_s(&fp, "sponza.obj", "rt");
3
4 while (!feof(fp))
5 {
6     //キーワード読み込み
7     fscanf_s(fp, "%s ", key, sizeof(key));
8     //メッシュ
9     if (strcmp(key, "o") == 0)
10    {
11        m_numMesh++;
12    }
13    //頂点
14    if (strcmp(key, "v") == 0)
15    {
16        vertCount++;
17    }
18    //法線
19    if (strcmp(key, "vn") == 0)
20    {
21        vnormalCount++;
22    }
23    //テクスチャ座標

```

```

24     if (strcmp(key, "vt") == 0)
25     {
26         vuvCount++;
27     }
28 }
```

---

OBJ ファイルを開き、while 文の中で 1 行ずつ文字を読み取る。その文字の中から v・vn・vt というキーワードがあればカウント用変数をインクリメント（増加）させる。

ソースコード A.8 メモリ領域確保

```

1 mesh.resize(m_numMesh);
2 XMFLOAT3* pCoord = new XMFLOAT3[vertCount];
3 XMFLOAT3* pNormal = new XMFLOAT3[vnormalCount];
4 XMFLOAT2* pUV = new XMFLOAT2[vuvCount];
```

---

頂点情報を格納する mesh 配列と頂点座標・法線・uv 座標を一時的に格納しておく配列を先程カウントした分だけメモリ領域を確保する。

ソースコード A.9 頂点情報格納

```

1 fseek(fp, SEEK_SET, 0);
2 vertCount = 0;
3 vnormalCount = 0;
4 vuvCount = 0;
5 faceCount = 0;
6 meshCount = 0;
7
8 while (!feof(fp))
9 {
10     //キーワード読み込み
11     ZeroMemory(key, sizeof(key));
12     fscanf_s(fp, "%s ", key, sizeof(key));
13
14     //頂点 読み込み
15     if (strcmp(key, "v") == 0)
16     {
17         fscanf_s(fp, "%f %f %f", &x, &y, &z);
18         pCoord[vertCount].x = x;
19         pCoord[vertCount].y = y;
20         pCoord[vertCount].z = z;
21         vertCount++;
22     }
23
24     //法線読み込み
25     if (strcmp(key, "vn") == 0)
26     {
27         fscanf_s(fp, "%f %f %f", &x, &y, &z);
```

```

28         pNormal[vnormalCount].x = x;
29         pNormal[vnormalCount].y = y;
30         pNormal[vnormalCount].z = z;
31         vnormalCount++;
32     }
33
34     //テクスチャ座標読み込み
35     if (strcmp(key, "vt") == 0)
36     {
37         fscanf_s(fp, "%f %f", &x, &y);
38         pUV[vuvCount].x = x;
39         pUV[vuvCount].y = 1 - y;
40         vuvCount++;
41     }
42 }
```

---

ファイルの読み取り位置、カウント用変数を 0 で初期化する。頂点情報・法線・uv 座標をあらかじめ用意しておいた配列に一時的に格納する。

ソースコード A.10 最終的な格納先の配列のメモリ確保

```

1 piFaceBuffer.resize(m_numMesh);
2 pVertexBuffer.resize(m_numMesh);
3
4 for (int i = 0; i < m_numMesh; i++)
5 {
6     piFaceBuffer[i].resize(mesh[i].m_numTriangles * 3);
7     pVertexBuffer[i].resize(mesh[i].m_numTriangles * 3);
8 }
```

---

`piFaceBuffer` と `pVertexBuffer` という二次元動的配列をメモリ確保する。まず、`resize` でメッシュの数だけメモリ確保する。つぎにメッシュの数だけループを回し、そのメッシュを構成するポリゴンの数だけメモリ確保する。`m_numTriangles` に 3 を掛けている理由はポリゴンが三角形で作られていることが保証されているからである。

ソースコード A.11 最終的な格納先へ頂点番号・頂点情報を格納

```

1 while (!(feof(fp)))
2 {
3     //キーワード読み込み
4     ZeroMemory(key, sizeof(key));
5     fscanf_s(fp, "%s ", key, sizeof(key));
6
7     //メッシュ
8     if (strcmp(key, "o") == 0)
9     {
10         meshCount++;
11         faceCount = 0;
```

```
12     }
13
14     //フェイス読み込み
15     if (strcmp(key, "usemtl") == 0)
16     {
17         fscanf_s(fp, "%s ", key, sizeof(key));
18         for (DWORD i = 0; i < m_numMaterial; i++)
19         {
20             if (strcmp(key, material[i].name) == 0)
21             {
22                 boFlag = true;
23                 materialIndex = i;
24                 strcpy_s(mesh[meshCount - 1].materialname, key);
25                 break;
26             }
27             else
28             {
29                 boFlag = false;
30             }
31         }
32     }
33     if (strcmp(key, "f") == 0 && boFlag == true)
34     {
35         if (strlen(material[materialIndex].textureName) > 0)
36         {
37             fscanf_s(fp, "%d/%d/%d %d/%d/%d %d/%d/%d", &v1, &vt1, &vn1, &
38             v2, &vt2, &vn2, &v3, &vt3, &vn3);
39         }
40         else
41         {
42             fscanf_s(fp, "%d/%d %d/%d %d/%d", &v1, &vn1, &v2, &vn2, &v3, &
43             vn3);
44         }
45         //インデックスバッファ
46         piFaceBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3] = mesh[
47             meshCount - 1].faceCount * 3;
48         piFaceBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 1] =
49             mesh[meshCount - 1].faceCount * 3 + 1;
50         piFaceBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 2] =
51             mesh[meshCount - 1].faceCount * 3 + 2;
52         //頂点構造体に代入
53         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3].pos =
54             pCoord[v1 - 1];
55         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3].normal =
56             pNormal[vn1 - 1];
57         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3].uv = pUV
```

```

51         [vt1 - 1];
52         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 1].pos
53             = pCoord[v2 - 1];
54         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 1].
55             normal = pNormal[vn2 - 1];
56         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 1].uv =
57             pUV[vt2 - 1];
58         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 2].pos
59             = pCoord[v3 - 1];
60         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 2].
61             normal = pNormal[vn3 - 1];
62         pVertexBuffer[meshCount - 1][mesh[meshCount - 1].faceCount * 3 + 2].uv =
63             pUV[vt3 - 1];
64     }
65 }

```

キーワードが f(ポリゴン) かつマテリアルが使われていたら頂点番号と頂点情報を読み取って格納していく。

---

ソースコード A.12 書き込み

---

```

1 ofstream out("Sponza-Atrium/sponza.bin", ios::out | ios::binary);
2 if (!out) return 1;
3
4 int sumMesh = m_numMesh;
5 int sumMaterial = m_numMaterial;
6
7 out.write(reinterpret_cast<const char*>(&sumMesh), sizeof(sumMesh));
8 out.write(reinterpret_cast<const char*>(&sumMaterial), sizeof(sumMaterial));
9 out.write(reinterpret_cast<const char*>(&mesh[0]), mesh.size() * sizeof(Mesh));
10
11 for (int i = 0; i < m_numMesh; i++)
12 {
13     out.write(reinterpret_cast<const char*>(&piFaceBuffer[i][0]), piFaceBuffer[i].
14         size() * sizeof(int));
15     out.write(reinterpret_cast<const char*>(&pVertexBuffer[i][0]), pVertexBuffer[i].
16         size() * sizeof(Vertex));
17 }
18 out.write(reinterpret_cast<const char*>(&material[0]), material.size() * sizeof(
19 Material));
20 out.close();

```

---

sponza.bin という名前のバイナリファイルにメッシュ数・マテリアル数・メッシュ情報・メッシュごとの頂点番号・頂点情報、マテリアル情報を書き込んでいく。