第9章 形状の変形

9.1 バーテックスシェーダによる形状作成

バーテックスシェーダに in 変数によって与えられる頂点属性 (attribute) は、図形の頂点を識別できる情報であれば、必ずしも座標値である必要はありません。バーテックスシェーダにおいて組み込み変数 gl_Position に代入したものが、クリッピング空間における図形の頂点位置になります。したがって、入力された頂点属性をもとに gl_Position に代入する値をバーテックスシェーダ内で決定することにより、バーテックスシェーダによって形を作ることができます。

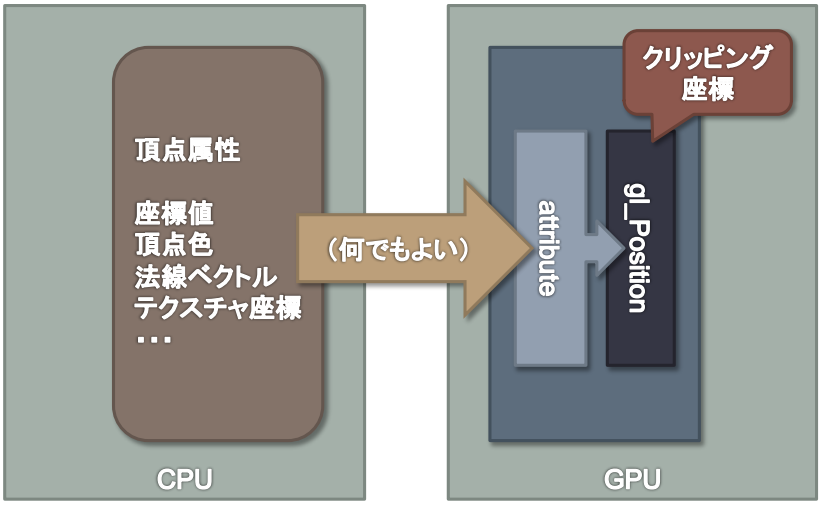


図 102 頂点属性の転送

例えば、頂点属性として図 103 (a) のように格子状に並んだ二次元の頂点データを準備したとします。

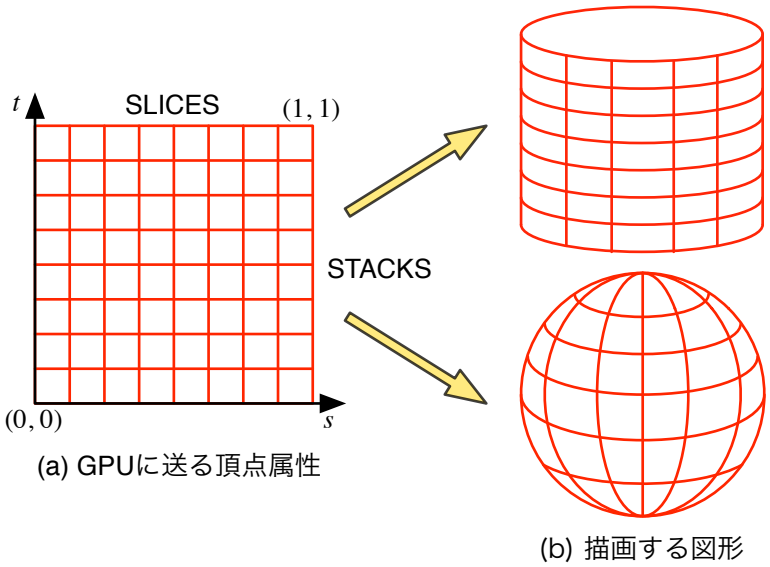


図 103 バーテックスシェーダによる形状の作成

このようなデータは、例えば次のプログラムで作成できます。

```
float parameter[(SLICES + 1) * (STACKS + 1)][2];

for (int j = 0; j <= STACKS; ++j)
{
  for (int i = 0; i <= SLICES; ++i)
  {
    int k = i + j * (SLICES + 1);
    float s = (float)i / (float)SLICES;
    float t = (float)j / (float)STACKS;

    attribute[k][0] = s;  // [0,1] の値
    attribute[k][1] = t;  // [0,1] の値
  }
}
```

この配列変数 parameter を頂点バッファオブジェクト (VBO) に詰めます。

```
GLuint parameterBuf;
glGenBuffers(1, &parameterBuf);
glBindBuffer(GL_ARRAY_BUFFER, parameterBuf);
glBufferData(GL_ARRAY_BUFFER, sizeof parameter, parameter, GL_STATIC_DRAW);
```

そして、頂点配列オブジェクトを介して図形の描画を行います。

```
// in (attribute) 変数 parameter のインデックスの検索
GLint parameterLoc = glGetAttribLocation(program, "p1");

// 描画に使う頂点配列オブジェクトの指定
glBindVertexArray(vao);

// 描画に使う頂点バッファオブジェクトの指定
glBindBuffer(GL_ARRAY_BUFFER, parameterBuf);
glVertexAttribPointer(parameterLoc, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(parameterLoc);
```

```
// 図形の描画
glDrawArrays(GL_POINTS, 0, (SLICES + 1) * (STACKS + 1));
```

このとき、シェーダプログラム program のバーテックスシェーダにおいて次のようなプログラムを実行すれば、この頂点属性から球面上の頂点位置を得ることができます。これは頂点属性で与えられた [0, 1] の範囲の二次元のパラメータの s 成分と t 成分を、それぞれ [0, 2π], [0, π] の範囲に変換し、さらにそれらを方位角と仰角に用いて単位球面上の位置を求めます。

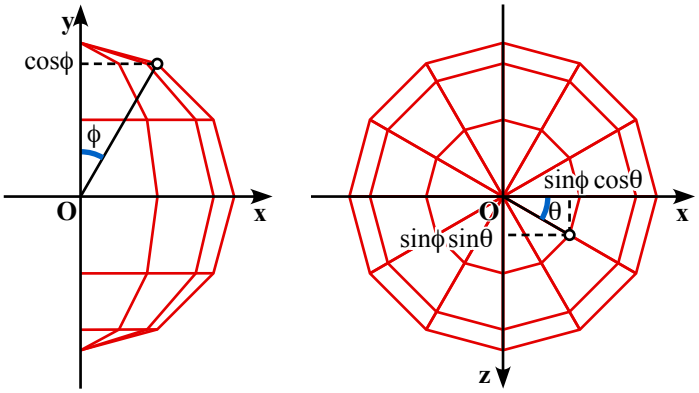


図 104 方位角・仰角と球面上の点の位置

```
#version 150 core
in vec2 parameter;       // CPU から与えられる頂点属性
uniform mat4 mc;         // 変換行列

void main(void)
{
  float th = 6.283185 * parameter.s;  // [0,1]→[0,2π]
  float ph = 3.141593 * parameter.t;  // [0,1]→[0,π]
  float r = sin(ph);

  vec4 p = vec4(r * cos(th), cos(ph), r * sin(th), 1.0);

  gl_Position = mc * p;
}
```

また、これを次のプログラムに書き換えれば、同じ頂点属性から、今度は円柱面上の頂点位置を得ることができます。これは頂点属性で与えられた [0, 1] の範囲の二次元のパラメータの s 成分と t 成分を、それぞれ [0, 2π], [-1, 1] の範囲に変換し、s 成分については方位角、t 成分については円柱の高さ方向の値を求めます。

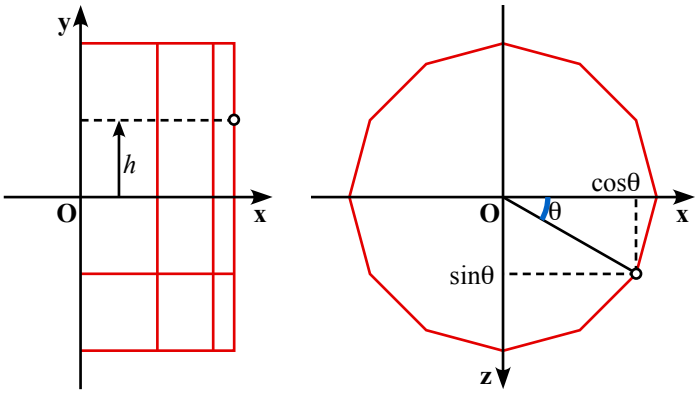


図 105 方位角・高さと円柱面上の点の位置

```
#version 150 core
in vec2 parameter;      // CPU から与えられる attribute
uniform mat4 mc;        // 変換行列

 void main(void)
 {
   float th = 6.283185 * parameter.s;  // [0,1]→[0,2π]
   float y = parameter.t * 2.0 - 1.0;  // [0,1]→[-1,1]

   vec4 p = vec3(cos(th), y, sin(th), 1.0);

   gl_Position = mc * p;
 }
```

9.2 変形のアニメーション

これまで述べてきた座標変換によるアニメーションは、形の変形を伴わない剛体変換はもとより、拡大縮小やせん断変形を含め、いずれも glDrawArrays() や glDrawElements() などによって一度に描画される形状、すなわち描画単位を対象にしていました。このとき、変換行列は描画に先立って描画途中に変化することの無い uniform 変数に設定されるため、描画単位に含まれるすべての頂点には同じ変換が適用されます。

しかし、人体の動きや表情の変化のように、対象形状の部位ごとに異なる変形は、このような描画単位ごとの均一な変換では実現することができません。このような変形を行うためには、変換行列によらずに頂点位置を決定する必要があります。このため、従来このような変形は、アプリケーション (CPU) 側で実行されていました。

プログラマブルシェーダの導入により、このような頂点ごとに異なる処理も GPU 側で実行可能になりました。この場合、アプリケーションは基本となる頂点属性を GPU 側に送り、変形の処理はシェーダプログラムで実行します。

9.2.1 モーフィング

表情の変化のように、対象形状の部位によって異なる変形を行う場合には、**モーフィング**が用いられます。これは対象形状の変形前と変形後の形状を用意しておき、それらの間で対応する頂点の位置を線形補間により求めて、中間の形状を決定します。

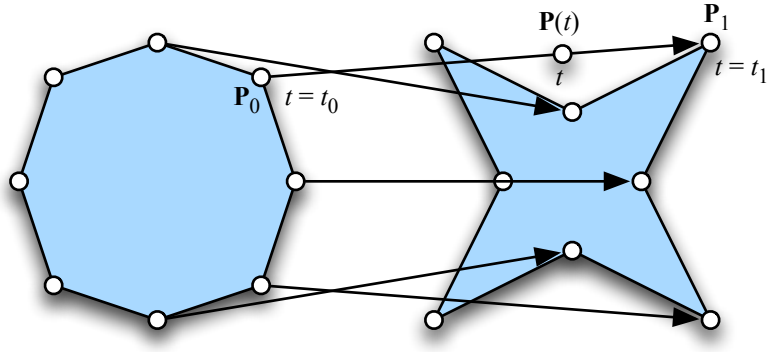


図 106 モーフィング

図 106 の二つの図形の頂点の数や接続関係は一致しており、頂点の位置のみが異なっています。いま、時刻 t_0 における形状の頂点 $\mathbf{P}_0$ に対して時刻 v ける形状の頂点 $\mathbf{P}_1$ が対応付けられているとき、この二点の位置の時刻 t における位置 $\mathbf{P}(t)$ は、次式により求めることができます。

$$\begin{aligned} t &\in [t_0, t_1] \\ s(t) &= \frac{t - t_0}{t_1 - t_0} \\ \mathbf{P}(t) &= (1 - s(t))\mathbf{P}_0 + s(t)\mathbf{P}_1 \end{aligned} \tag{72}$$

このとき、$0 \leq s(t) \leq 1$ です。いま、$t_0 = 0$, 時刻 $t_1 = 1$ として、これをバーテックスシェーダに実装すると、次のようなプログラムになります。関数 mix(p0, p1, t) は GLSL の組み込み関数で、p0 * (1 – t) + p1 * t を計算します。

```
#version 150 core
in vec4 p0, p1;         // 変形前と変形後の点の位置
uniform float t;        // 時刻
uniform mat4 mc;        // 座標変換行列
void main(void)
{
  gl_Position = mc * mix(p0, p1, t);
}
```

ここで注意しなければならないのは、gl_Position に代入すべき頂点位置を決定するのに、二つの頂点属性 p0, p1 が必要になる点です。

いま、p0 には既に時刻 $t_0 = 0$ における形状 (元の形状) の頂点属性を保持する頂点バッファオブジェクト (VBO) が割り当てられているとします。そこで、これに加えて時刻 $t_1 = 1$ における形状 (変形後の形状) の頂点属性を保持する頂点バッファオブジェクト準備します。ここで p1 は変形後頂点の三次元位置を格納した配列変数であるとし、vertices にはその頂点の数が格納されているとします。

```
// p1 の頂点バッファオブジェクトの作成
GLuint p1Buf;
glGenBuffers(1, &p1Buf);
glBindBuffer(GL_ARRAY_BUFFER, p1Buf);
glBufferData(GL_ARRAY_BUFFER,
  sizeof (GLfloat[3]) * vertices, p1, GL_STATIC_DRAW);
```

また、シェーダプログラム program のバーテックスシェーダにおいて、頂点属性の入力に用いる in 変数 p1 のインデックスを求めます。

```
// in (attribute) 変数 p1 のインデックスの検索（見つからなければ -1）
GLint p1Loc = glGetAttribLocation(program, "p1");
```

描画の際には、描画に用いる頂点配列オブジェクトを結合します。これには既に p0 の頂点バッファオブジェクトが組み込まれているとします。これに p1 の頂点バッファオブジェクトを追加し、それを in 変数 p1 のインデックスに割り当てます。そして、図形の描画を行います。lines には描画する図形の頂点インデックスの数が格納されているとします。

```
// 描画に使う頂点配列オブジェクトの指定
glBindVertexArray(vao);

// 頂点バッファオブジェクトを in (attribute) 変数 p1 から参照できるようにする
glBindBuffer(GL_ARRAY_BUFFER, p1Buf);
glVertexAttribPointer(p1Loc, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(p1Loc);

// 図形の描画
glDrawElements(GL_LINES, lines, GL_UNSIGNED_INT, 0);
```

9.2.2 モーフターゲット

人の表情やリップシンク (音声に会わせて口を動かすこと、口パク) などでは、元の形状に対して、変形後の形状が複数存在する場合があります。モーフィングにおいて目標となる複数の形状を、モーフターゲットと呼びます。このとき、複数の形状の間で、形状が滑らかに切り替わる必要があります。

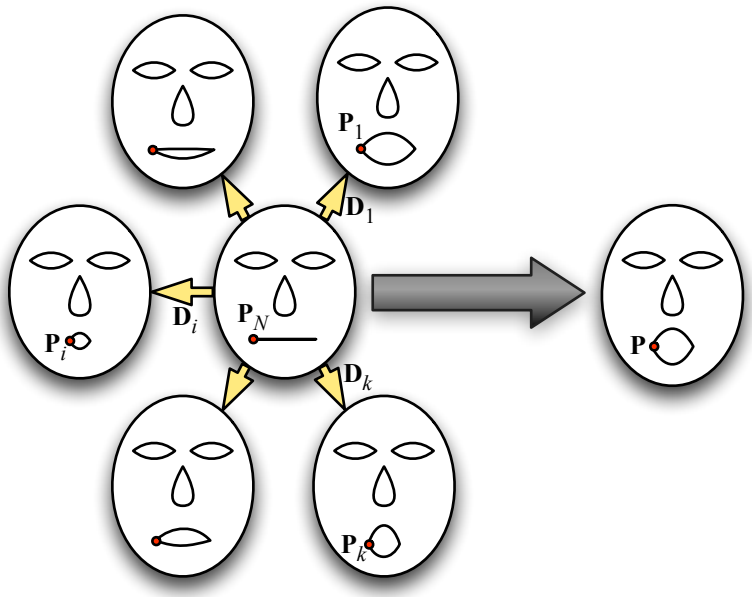


図 107 モーフターゲット

そこで、まず基準となる形状を決定します。その形状の頂点の位置を $\mathbf{P}_N$ とします。また、目標の形状 i の対応する頂点の位置を $\mathbf{P}i$ とします。そして、これと基準となる形状の頂点位置と

の差 $\mathbf{D}i$ を求めます。

$$\mathbf{D}_i = \mathbf{P}_i - \mathbf{P}_N \tag{73}$$

このとき、目標の形状を合成して得ようとする目的の形状の頂点の位置 $\mathbf{P}$ を、w_i を目標の形状 i に近づく度合いを表す重みとして、次式により求めます。

$$\mathbf{P} = \mathbf{P}_N + \sum_{i=1}^{k} w_i \mathbf{D}_i \tag{74}$$

これにより、基準の形状に目標の形状への差分の重み付け和を加えることによって、ある目標の形状から、別の目標の形状までの変形を滑らかにアニメーションすることができます。

なお、モーフターゲットでは、基準の形状に加えて目標の形状の数だけ、頂点属性を用意する必要があります。

9.3 骨格の変形

人体やロボットのように、骨格に沿った変形のアニメーションは CG やゲームなどでは非常にポピュラーなものです。特に、人体のように表面はひと続きになっている形状を骨格によって変形する手法は、スキニングと呼ばれます。

9.3.1 関節を動かす

ロボットのアニメーションでは、関節部分は複数のパーツの組み合わせで表現するのが一般的です。この場合、パーツごとにアニメーションの変換行列 (剛体変換) を割り当てます。図 108 では、二つのパーツのそれぞれに $\mathbf{M}_1, \mathbf{M}_2$ の変換行列が割り当てられています。

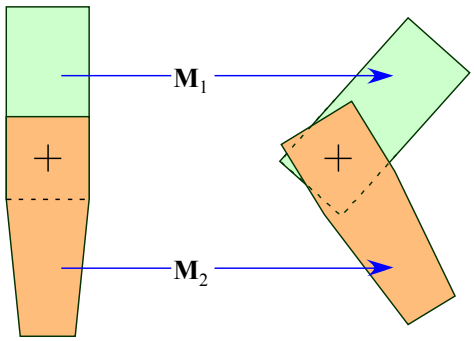


図 108 関節のアニメーション

しかし、この例では関節部分はやはり二つのパーツが別々に動いているように見えるため、人体のようにひと続きの形状としては知覚されません。この例では、接合部分は二つのパーツが重なり合っているものの、人間の「ひじ」のようには見えません。このような部分は単一のパーツで表現すべきですが、これは静的な (形状が変形しない) パーツでは解決できません。

関節部分を柔軟性のあるパーツで接合すれば、全体として一つのパーツであると捉えることが可能になります。柔軟性のあるパーツとは、例えば異なるパーツ同士の頂点の接続関係だけを定義し、それ自身は独自の頂点を持たないようなものです。

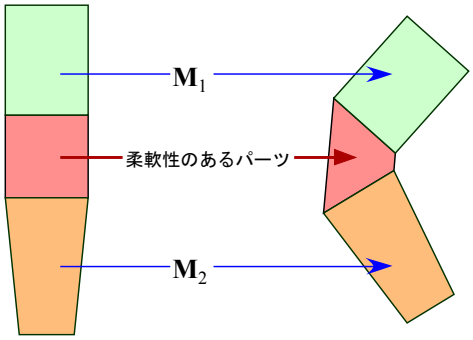


図 109 柔軟性のあるパーツによる接続

柔軟性のあるパーツの頂点の座標は、その頂点を共有しているパーツの変換の影響を受けます。しかし、柔軟性のあるパーツ固有の頂点を持たないため、変換によっては形状がつぶれるなどの問題が発生します。ひと続きの形状なのに、部位によって性質が異なるパーツを組み合わせることには無理があります。

9.3.2 バーテックスブレンディング

バーテックスブレンディングは、一つの頂点に対する複数の変換の結果を合成して、頂点の位置を決定する方法です。これにより、ひと続きのパーツに対して、骨格に沿った変形を行います。

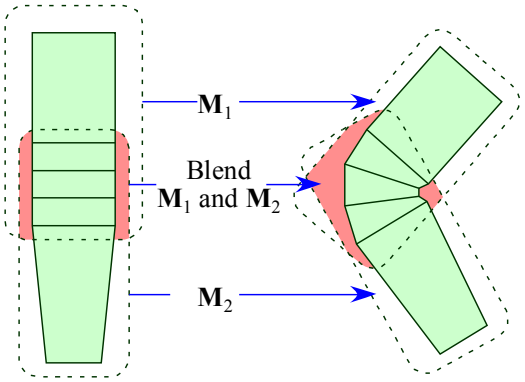


図 110 バーテックスブレンディング

バーテックスブレンディングでは、対象の形状の全体を、柔軟性のある単一のパーツで表します。また、二つの変換 $\mathbf{M}_0, \mathbf{M}_1$ について、それぞれの影響範囲を決定します。そして、影響範囲が重なる部分の頂点の座標値は，$\mathbf{M}_0, \mathbf{M}_1$ による変換の結果を合成して求めます。

以降でバーテックスブレンディングの手順を説明します。この変形の基準となる仮想的な骨格のパーツを**ボーン**と呼びます。ボーンは、そのローカル座標系の Z 軸に沿った、長さ 1 の線分です。ただし、説明する際には線分では識別しづらいので、図 111 では八面体で表しています。

ボーンを、このボーンを骨格として変形しようとするワールド座標系上の初期形状に沿って配置します。図 111 の $\mathbf{M}_0, \mathbf{M}_1$ は、この配置に用いたモデリング変換 (行列) です。また、同様にこのボーンの変形後の形状をワールド座標系に配置します。この配置に用いたモデリング変換を、それぞれ $\mathbf{B}_0(t), \mathbf{B}_1(t)$ とします。アニメーションであれば、この変換は時刻 t の関数です。

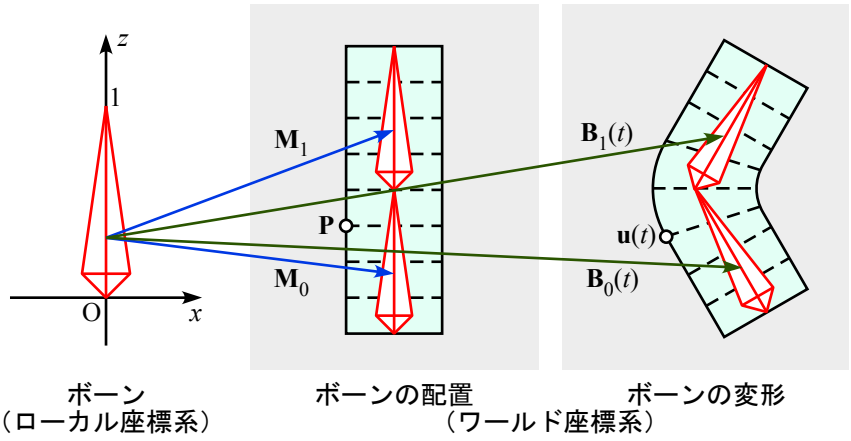


図 111 ボーンの配置

初期形状の頂点の位置 **P** の、変形後の頂点の位置 **u**(*t*) は、以下の手順により求めます。まず、この点 **P** の各ボーンに対する相対位置、すなわち、その点のボーンのローカル座標系における位置を求めます。これには、初期形状に沿って配置したボーンのモデリング変換の逆変換 $\mathbf{M}_0^{-1}$, $\mathbf{M}_1^{-1}$ を用います (図 112)。

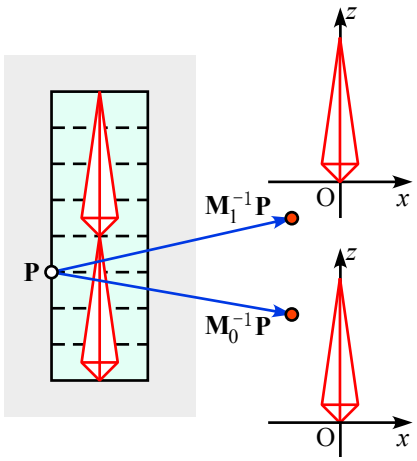


図 112 ボーンに対する相対位置の算出

次に、これらの点の位置に対して、ボーンを変形後の位置に配置するモデリング変換 $\mathbf{B}_0(t)$, $\mathbf{B}_1(t)$ を適用します。その結果、初期形状の頂点の位置 **P** の、それぞれのボーンに沿った変形後の位置が得られます (図 113)。

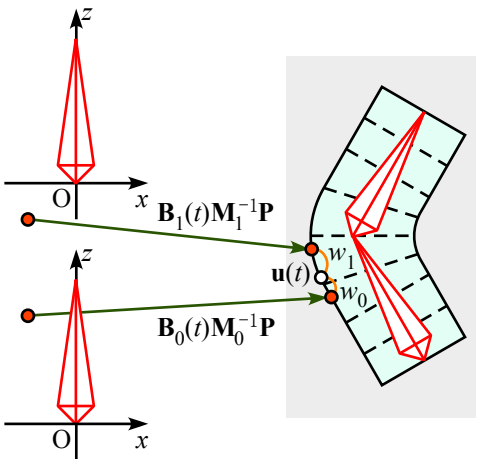


図 113 変形後の頂点位置のブレンド

ただし、この位置は変形に用いたボーンの数だけ存在します。そこで、それぞれの変換の重みを w_0, w_1 として、その頂点の変形後の位置 $\mathbf{u}(t)$ を、それぞれの重み付け和により求めます。

$$\mathbf{u}(t) = w_0\mathbf{B}_0(t)\mathbf{M}_0^{-1}\mathbf{P} + w_1\mathbf{B}_1(t)\mathbf{M}_1^{-1}\mathbf{P} \tag{75}$$

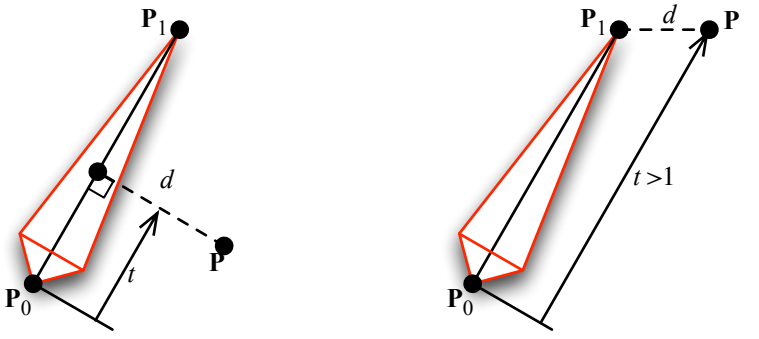
骨格が n 本のボーンで構成されるときは、$\mathbf{u}(t)$ はそのすべてのボーンに対して変形後の位置を求めて、それらの重み付け和により計算します。

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i\mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{P} \tag{76}$$

ここで重み w_i には、次の条件が与えられます。

$$\sum_{i=0}^{n-1} w_i = 1,\ w_i \geq 0 \tag{77}$$

この重み w_i は、初期形状の頂点 $\mathbf{P}$ と個々のボーン i との距離や、影響範囲などによって決定します。影響範囲の設定方法には様々な方法がありますし、手続き的な処理が用いられる場合もあります。ここでは $\mathbf{P}$ と個々のボーン i との距離をもとに決定する方法について解説します。



(a) 点とボーンの軸との距離　　(b) 点とボーンの端点との距離

図 114 ボーンと点との距離

図 114 (a) においてボーンの両端点の位置をそれぞれ $\mathbf{P}_0, \mathbf{P}_1$ とするとき、この二点を通る直線

と任意の点 **P** との距離 d は、次式により求めることができます。

$$\begin{aligned}&\mathbf{V}_1 = \mathbf{P}_1 - \mathbf{P}_0,\ \ \mathbf{V}_2 = \mathbf{P} - \mathbf{P}_0\\&t = \frac{\mathbf{V}_1 \cdot \mathbf{V}_2}{\mathbf{V}_1^2}\\&d = |\mathbf{V}_2 - \mathbf{V}_1 t|\end{aligned} \tag{78}$$

図 114 (b) のように点 **P** が線分 $\mathbf{P}_0\mathbf{P}_1$ の鉛直方向にないときは、d にはボーンの **P** に近いほうの端点との距離を用います。したがって、d は t の値により場合分けして求めます。

$$d = \begin{cases} |\mathbf{V}_2| & (t \le 0) \\ |\mathbf{V}_2 - \mathbf{V}_1 t| & (0 < t < 1) \\ |\mathbf{V}_2 - \mathbf{V}_1| & (t \ge 1) \end{cases} \tag{79}$$

ここで t を [0, 1] の範囲でクランプ ($t < 0$ なら $t = 0$, $t > 1$ なら $t = 1$) すれば、この場合分けは不要になります。GLSL の組み込み関数 clamp(*x*, *min*, *max*) を用いれば、これは clamp(*t*, 0, 1) で求めることができます。

この距離 d をもとに、このボーンによる重み w_i を次式により求めます。

$$w_i = (d + 1)^{-c} \tag{80}$$

d に 1 を加えているのは、(...) 内が 0 にならないようにするためです。この指数 c には 16 や 64 などの値を用います。c が大きいほど、頂点 **P** が離れたボーンから受ける影響が急激に小さくなります。一方、複数のボーンからの距離がほぼ等しい場合には、それらの影響が拮抗する位置に頂点が移動します。

なお、w_i は (77) 式の条件を満たす必要がありますが、**P** が同次座標で表されており、かつ、$\mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{P}$ の w 要素が 1 であれば、(76) 式によって $\mathbf{u}(t)$ の w 要素は w_i の総和になります。したがって、$\mathbf{u}(t)$ の実座標において w_i は (77) 式の条件を満たします。

$$\mathbf{u}(t) = \sum_i w_i \begin{pmatrix} x_i \\ y_i \\ z_i \\ 1 \end{pmatrix} = \begin{pmatrix} \sum_i w_i x_i \\ \sum_i w_i y_i \\ \sum_i w_i z_i \\ \sum_i w_i \end{pmatrix} \Rightarrow \begin{pmatrix} \sum_i \frac{w_i}{\sum_i w_i} x_i \\ \sum_i \frac{w_i}{\sum_i w_i} y_i \\ \sum_i \frac{w_i}{\sum_i w_i} z_i \end{pmatrix} \tag{81}$$

● **シェーダプログラム**

バーテックスブレンディングを実装したシェーダプログラムの例を以下に示します。下線部が重み w_i に相当します。

```
#version 150 core
in vec4 position;                       // 点の位置（ローカル座標）
uniform mat4 modelViewMatrix;           // モデルビュー変換行列
uniform mat4 projectionMatrix;          // 投影変換行列

// ボーンのデータ
const int MAXBONES = 8;
uniform int numberOfBones;              // ボーンの数
uniform vec4 p0[MAXBONES];              // ボーンの根元の位置（ワールド座標）
uniform vec4 p1[MAXBONES];              // ボーンの先端の位置（ワールド座標）
uniform mat4 blendMatrix[MAXBONES];     // Bi * inverse(Mi)（CPU で計算しておく）
const float exponent = -16.0;           // 重みの指数
```

```
void main()
{
  vec4 p = modelViewMatrix * position, u = vec4(0.0);
  for (int i = 0; i < numberOfBones; ++i)
  {
    vec4 v1 = p1[i] - p0[i], v2 = p - p0[i];
    float l = dot(v1, v1);
    if (l > 0.0) v2 -= v1 * clamp(dot(v1, v2) / l, 0.0, 1.0);  // v2 - v1 t
    u += pow(length(v2) + 1.0, exponent) * blendMatrix[i] * p; // 重み付け和
  }
  gl_Position = projectionMatrix * u;
}
```

● **組み込み関数**

今回提示したシェーダのソースプログラムでは、以下の組み込み関数を使っています。

sin(x), cos(x)

三角関数, x (radian) の正弦、余弦

pow(x, y)

指数関数, x^y

mix(v1, v2, t)

v1 と v2 を t で比例配分、v1 * (1 - t) + v2 * t

dot(v1, v2)

ベクトル v1, v2 の内積、外積は cross(v1, v2)

length(v)

v の絶対値／長さ

clamp(v, min, max)

クランプ、v < min なら min, v > max なら max, それ以外は v