モーションブレンディング

http://www.tmps.org/index.php?%A5%E2%A1%BC%A5%B7%A5%E7%A5%F3%A5%D6%A5%EC%A5%F3%A5%C7%A5%A3%A5%F3%A5%B0

本稿では，時間長や運動速度が異なる複数の動作をブレンディングする方法を説明します．この技法を使うことで，ゆっくりな動作と素早い動作から，その中間的な速さの動作を合成することができます．まず，一般的なモーションブレンディングの処理手順を説明し，複数の動作の時間長やタイミングを同期させるための技法を説明します．モーションデータの同期処理は，モーションデータベース検索の基礎手法でもあり，これまでにもいくつかのアルゴリズムが報告されています．本稿では「registration curves」と呼ばれる，動的時間伸縮法（Dynamic Timewarping）と呼ばれる波形マッチング法を，キャラクタのモーションデータに応用したアルゴリズムを紹介します．

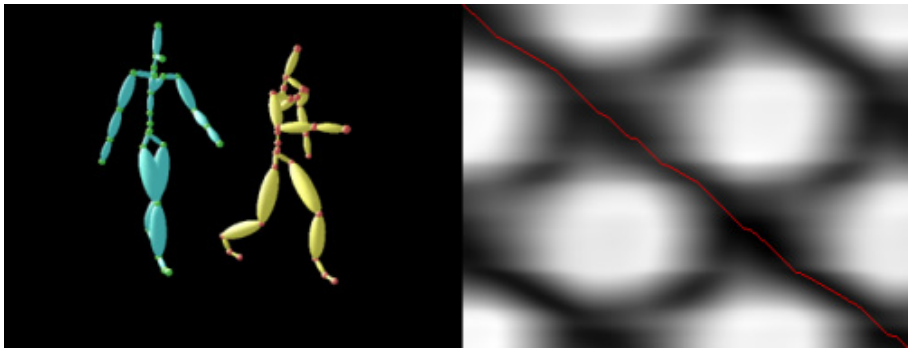


Fig.1 実行画面のスナップショットと registration curve

サンプルプログラム

動作のフィルタリングのプログラムを拡張し，モーションブレンディングを行うためのプログラムを作成しました．サンプルプログラムは，VisualC++ 2005 + DirectX SDK October 2006 の環境下で，MFCを用いて作成しています．なお，処理例で用いた BVH モーションキャプチャデータもアーカイブに含めています．

MFC 版(VC++2005, 209kb)

また，いくつかの機能を除き，DXUTを用いて作成したサンプルも残しておきます．

DXUT 版(VC++2005, 442kb)

参考文献

1. Armin Bruderlin and Lance Williams, "Motion Signal Processing", SIGGRAPH 1995, pp.97-104, 1995.
2. Lucas Kovar, Michael Gleicher, and Fred Pighin, "Motion Graphs", ACM Transactions on Graphics (SIGGRAPH 2002), vol.21, no.3, pp. 473-482 , 2002.
3. Lucas Kovar and Michael Gleicher, "Flexible Automatic Motion Blending with Registration Curves", Symposium on Computer Animation 2003, pp.214-224, 2003.
4. Lucas Kovar and Michael Gleicher, "Automated Extraction and Parameterization of Motions in Large Data

Sets", ACM Transactions on Graphics (SIGGRAPH 2004), vol.23, no.3, pp. 559-568 , 2004.

まず論文(1)の中で，動的時間伸縮法のモーションブレンディングへの応用が提案されました．今回紹介する registration curves は，2002 年の SIGGRAPH で発表された手法(3)を基礎とし，2003 年の SCA で発表されました(3)．さらに registration curves は，モーションデータベースからの類似動作クリップの自動抽出アルゴリズム(4)に応用されています．

モーションブレンディングの概要

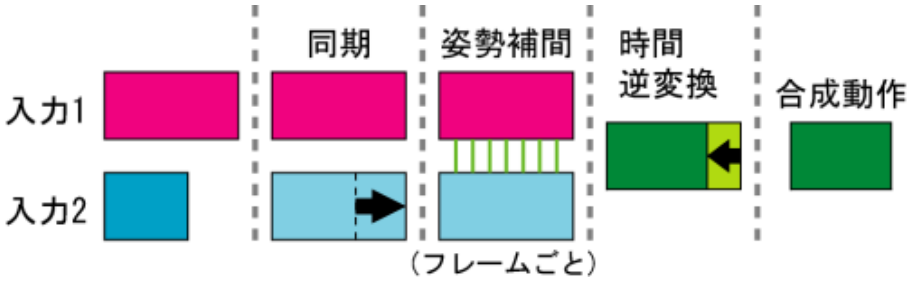


Fig.2 基本的なモーションブレンディング処理手順

一般的なモーションブレンディングの処理手順を Fig.2 にまとめます．まず，複数のサンプル動作を同期させ，例えば歩行動作における接地や足を上げるタイミングを一致させます．次に，同期した動作の姿勢を各時間フレームごとに補間します．そして，補間された動作を時間的に逆変換することで，最終的な動作を合成します．この時間逆変換の処理は，時間長が違う動作をブレンドしたときに，それらの中間的な長さの動作を合成するための処理になります．なお，姿勢の補間処理は動作のフィルタリングで説明した方法をそのまま利用できますので，ここでは複数の動作の同期処理と時間逆変換処理を中心に説明します．

動的時間伸縮法：Dynamic Timewarping

動的時間伸縮法(Dynamic Timewarping，以後は DTW と略記)は，データ長やピーク値をとるタイミングが一致するように 2 つの波形データの同期を取るアルゴリズムです(非線形時間伸縮法とも呼ばれるようです)．Fig.3 に示す例では，DTWを用いて青線の波形を赤線の波形に時間的に一致するように変形していますが，波形の時間長が一致し，ほぼ同じタイミングでピーク値をとるように変換されることがわかります．

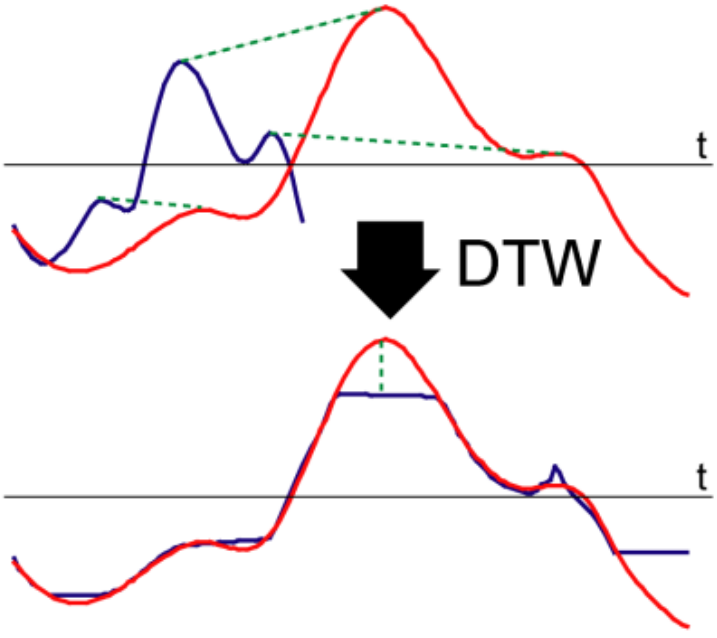


Fig.3 動的時間伸縮法: Dynamic Timewarping

一般的に，DTW は次の手順で計算されます．

- 入力:波形 1 $F(m)$, $m \in \{1 \cdots M\}$ と，波形 2 $G(n)$, $n \in \{1 \cdots N\}$
- 出力: 波形 1 に同期した波形 2 $G'(t)$, $t \in \{1 \cdots M\}$

1. 距離行列 D の計算
   - サイズ $M \times N$ の行列 D を生成し，各成分 $D(m.n)$ に $F(m)$ と $G(n)$ の非類似度(ex. 2乗差分 $|F(m)-G(n)|^2$など)を格納します．
   - Fig.3 の 2 つの波形から計算される距離行列を Fig.4 に示します．ピクセルが白いほど距離が大きいことを表します．

2. 距離行列上のコスト最小経路の計算
   - 距離行列の始点 (1.1) から終点 $(M.N)$ に至る経路のうち，経路上の成分の総和 $\sum D(m.n)$ が最小となるような経路を計算します．
   - 最適経路探索アルゴリズムには動的計画法が利用されます．
   - Fig.4 では赤線がコスト最小経路を示します．
3. コスト最小経路に沿った時間変換
   - コスト最小経路にあたる位置を $(i.j)$ とするとき，$G'(i)=G(j)$ のように時間変換．

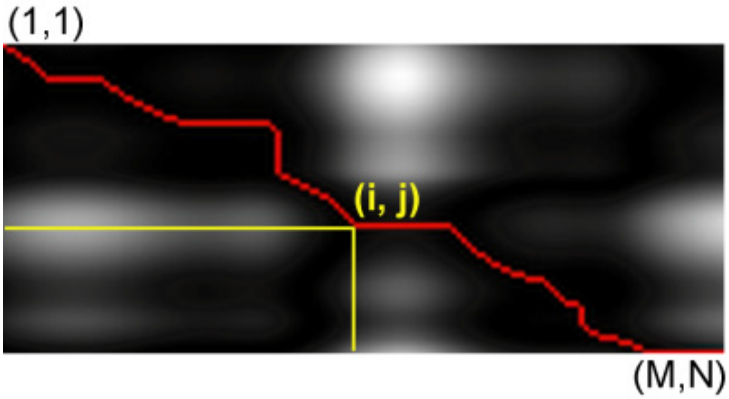


Fig.4 距離行列とコスト最小経路

このアルゴリズムをモーションデータに関して考えると，$F(m)$ が基準となるモーションデータ，$G(n)$ は時間変換の対象となるモーションデータ，距離行列に格納される値は 2 つの姿勢の非類似度に対応します．そして，距離行列上のコスト最小経路は，参考文献 3 のなかで Registration curve（以後，R-curveと略記）と呼ばれています．

また，モーションデータに DTW を適応する際には，キャラクタの姿勢の非類似度を適切に計算することが重要です．ここでは姿勢間の非類似度を計算するための，代表的な 2 つの方法について説明します．

姿勢距離測度

DTW は一次元波形についてのアルゴリズムですが，データ間の非類似度をスカラ値で算出できれば，モーションデータのような多次元データにも適用できます．そこで，2 つのキャラクタの姿勢の「見た目」の差異を定量化するような，姿勢距離測度 d_p と呼ばれる指標を導入します．これまでに提案されている姿勢距離の計算方法には，①関節角度を利用する方法と，②関節位置などの身体部位位置を利用する方法の，大きく分けて 2 つのアプローチがあります．

関節角度に基づく姿勢距離

2 つのキャラクタの対応する関節角度 q_i と q'_i の差分 $dist(q_i.q'_i)$ をもとに距離を計算する方法です．

$$d_p=\sum_i w_i dist^2(q_i.q'_i)$$

$dist(q_i.q'_i)$ にはいくつかの定義がありますが，たとえば Interactive Control of Avatars Animated with Human Motion Data では，差分クォータニオン $dist(q_i.q'_i)=\|\log(q'^{-1}_i q_i)\|$ を用いています

ただし，この姿勢距離を利用する際には，各関節の重要度を示す加重係数 w_i を，各関節運動の見た目への影響を考慮して適切に設定する必要があります．例えば，$w_i=1.0$ のように全ての関節に同一の重みを設定すると，上半身全体を移動させる腰の曲げと，手先だけに影響する手首の曲げを同じ重要度で扱うことになります．そうした加重係数の一般的な設定方法は確立していませんので，経験的な試行錯誤が必要になります．

身体部位の位置に基づく姿勢距離

対応する 2 つの身体各部位の位置 p_i と p'_i のユークリッド距離をもとに計算する方法です[参考文献 2].

$$d_p = \sum_i w_i \| p_i - p'_i \|^2$$

この方法では，重み係数の設定はあまり考えなくても問題ありません．もちろん，注目部位の重みを大きくすることも考えられますが，ほとんどの研究では w_i は利用されていないようです．また「見た目の違い＝姿勢距離」という点では，関節角度を利用するよりも，身体部位の位置から計算する方法が自然だと思います．したがって本サンプルプログラムでは，身体位置に基づき，加重係数 w_i を考慮しない姿勢距離の計算コードのみを示しています．

ただし，姿勢距離を算出する際には，ワールド座標系における身体位置は直接利用できません．これは，例えば反対方向へ歩行する2つのキャラクタの姿勢距離を計算する場合，その歩行姿勢が非常に似ていたとしても，キャラクタが離れるにしたがって姿勢距離が大きくなるような不自然な結果が生じるためです．したがって，各部位間の距離を計算する前に，キャラクタの位置や向きをなるべく一致させるような正規化処理が必要とあります．例えば Motion Graphs では，次の手順で身体各部位に対応する点群の座標を正規化したうえで，姿勢距離を計算しています．

- 入力: 点群1 $\{ p_1(x_1 y_1 z_1) \cdots p_n(x_n y_n z_n) \}$, 点群2 $\{ p'_1(x'_1 y'_1 z'_1) \cdots p'_n(x'_n y'_n z'_n) \}$

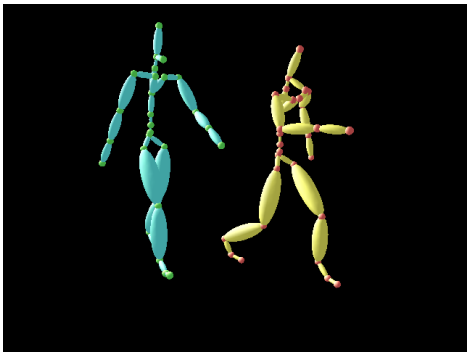
1. 座標正規化行列の計算
   - 点群1と点群2のユークリッド距離を最小化する座標正規化行列: $T_n = \mathrm{argmin} \sum_i w_i \| p_i - T_n p'_i \|^2$
   - y 軸（鉛直軸）周りの回転量 θ と，x, z方向への平行移動成分 x_0, z_0 をそれぞれ次式によって計算
     - $\theta = \arctan \dfrac{\sum_i w_i (x_i z'_i - x'_i z_i) - (\bar{x}\bar{z}' - \bar{x}'\bar{z}) / \sum_i w_i}{\sum_i w_i (x_i x'_i + z_i z'_i) - (\bar{x}\bar{x}' + \bar{z}\bar{z}') / \sum_i w_i}$
     - $x_0 = (\bar{x} - \bar{x}' \cos(\theta) - \bar{z}' \sin(\theta)) / \sum_i w_i$
     - $z_0 = (\bar{z} - \bar{x}' \sin(\theta) - \bar{z}' \cos(\theta)) / \sum_i w_i$
     - ただし，$\bar{x} = \sum_i w_i x_i$（他の変数も同様）
   - $T_n = RotateY(\theta) * Translate(x_0 . 0 . z_0)$ によって変換行列を計算
2. 姿勢距離 d_p の計算
   - $d_p = \sum_i w_i \| p_i - T_n p'_i \|^2$

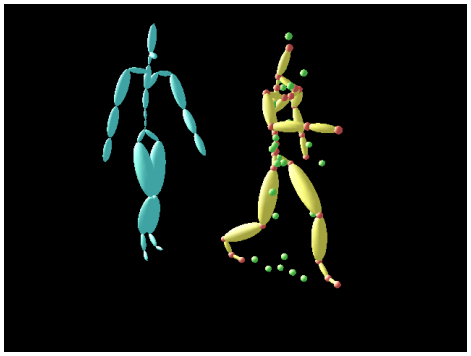
全ての重みを 1.0 に固定した場合，座標正規化行列の計算プログラムは次のようになります．STL::vector として入力される2つの点群から，T_n を計算しています．

```
 1  D3DXMATRIX NFigureUtil::CalcTransformBetweenFigures(const std::vector &pc1, const std::vector &pc2)
 2  {
 3    // top_left: thetaを計算する式の分子の左の項（以下同様）
 4    float top_left = 0, top_right = 0, bottom_left = 0, bottom_right = 0;
 5    float ax1 = 0, ax2 = 0, az1 = 0, az2 = 0;
 6    float tx, tz, ty = 0, ry;
 7
 8    for (size_t i = 0;i < pc1.size();++i)
 9    {
10      top_left += pc1[i].x * pc2[i].z - pc2[i].x * pc1[i].z;
11      bottom_left += pc1[i].x * pc2[i].x + pc1[i].z * pc2[i].z;
12
13      ax1 += pc1[i].x;
14      az1 += pc1[i].z;
15      ax2 += pc2[i].x;
16      az2 += pc2[i].z;
17
18      ty += pc1[i].y - pc2[i].y;
19    }
20    top_right = (ax1 * az2 - ax2 * az1) / static_cast<float>(pc1.size());
21    bottom_right = (ax1 * ax2 + az1 * az2) / static_cast<float>(pc1.size());
22
23    ry = std::atan2f(top_left - top_right, bottom_left - bottom_right);
24    tx = (ax1 - ax2 * cosf(ry) - az2 * sinf(ry)) / static_cast<float>(pc1.size());
25    tz = (az1 + ax2 * sinf(ry) - az2 * cosf(ry)) / static_cast<float>(pc1.size());
26    ty /= static_cast<float>(pc1.size());
27
```

```
28 |    D3DXMATRIX mr, mt;
29 |    D3DXMatrixRotationY(&mr, ry);
30 |    D3DXMatrixTranslation(&mt, tx, ty, tz);
31 |    return mr * mt;
32 ! }
```

このプログラムを用いた点群の座標正規化の結果を示します．入力として，左側のような2種類の歩行動作を与えたとき，それぞれのキャラクタの関節位置に対応する点群が計算されます．そして，この例では水色のスケルトンの点群を，黄色のスケルトンの点群との距離が最小になるように座標正規化しています．ムービーの中ごろで正規化された点群が不可思議な動きをしますが，そういうものなのでしょう(もしかしたらバグかもしれませんが．．．)





入力歩行動作と点群(m1vムービー，206KB)　点群の座標正規化(m1vムービー，266KB)

また，姿勢距離の算出プログラムは次のようになります．

```
1   float calc_pose_distance(const std::vector &pc1, const std::vector &pc2)
2   {
3     D3DXMATRIX trans = NFigureUtil::CalcTransformBetweenFigures(pc1, pc2);
4
5     D3DXVECTOR3 diff;
6     float dist =0.0f;
7     for (size_t j = 0;j < pc1.size();++j) {
8       D3DXVec3TransformCoord(&diff, &pc2[j], &trans);
9       diff = pc1[j] - diff;
10      dist += D3DXVec3LengthSq(&diff);
11    }
12    return sqrtf(dist / static_cast<float>(pc1.size()));
13  }
```

R-curve と時間変換行列

姿勢距離の導入により，2 つのモーションデータの距離行列とその R-curve が求められ ます．そして，DTWアルゴリズムで説明したように，R-curve を順に辿ることでモーション全体を時間変換できます．ただし，動的計画法によって求められる R-curve は，部分的に縦もしくは横方向に直線状に連続することがあります．こうした直線部分は，変換された動作の急激な変化，もしくは動作が一時的に静止するような不具合となって表れます．そのため参考文献 3では，B-spline 近似によって R-curveを平滑化するなどの対応をとっています．ただし B-spline 近似の実装が面倒だったで，ここでは畳み込みフィルタを用いて R-curve を平滑化する方法を紹介します．この平滑化処理によって，もちろん距離行列上のコスト最小性は失われ，歩行動作における接地のタイミングがズレるなどの副作用も生じますが，運動の滑らかさという意味では生成動作の品質を向上できます．さらに，R-curve を用いた時間変換処理を行列演算によって実現する方法を紹介します．

まず，R-curve の急激な変化を抑えるために，動的計画法の探索対象となる領域を制限します．これは制限したい範囲内の距離行列の要素に，十分に大きな値を加算することで 達成できます．ここでは，帯状の制限マスクを計算するコードを以下に示します．この関数では，探索対象領域の大きさを示すパラメータ ratio にしたがって，DBL_MAX/10.0 を加算する制限マスクを生成します．パラメータ ratio による探索対象領域を変化を Fig.5 に示します．

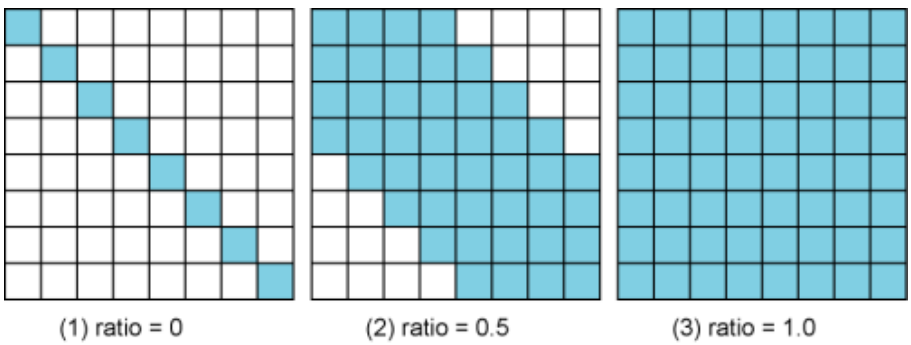


Fig.5 R-curve計算対象領域の制限

```
CMatrix gen_restriction_mask_belt(size_t row, size_t col, float ratio)
{
  int width = static_cast<int>(row * ratio);
  CMatrix mask(row, col, DBL_MAX / 10.0);
  for (int c = 0;c < col;++c) {
    int cr = static_cast<int>(static_cast<double>(row) / static_cast<double>(col) * c);
    for (int r = cr - width;r <= cr + width;++r) {
      if (r < 0 || r >= row)
        continue;
      mask(r, c) = 0;
    }
  }
  return mask;
}
```

次に，探索対象領域内で (1.1) から $(M.N)$ に至るコスト最小経路を動的計画法によって計算します(Fig.6(1))．そして，列方向に直線的に連続している部分の中間位置を算出し(Fig.6(2))，移動平均カーネルを用いた畳み込みフィルタリングを施します(Fig.6(3))．最後に，平滑化された中間座標をもとに R-curve を再計算します(Fig.6(4))．スマートな方法を思いつかなかったので，今回はこのように少し回りくどい手順を踏んでいます．

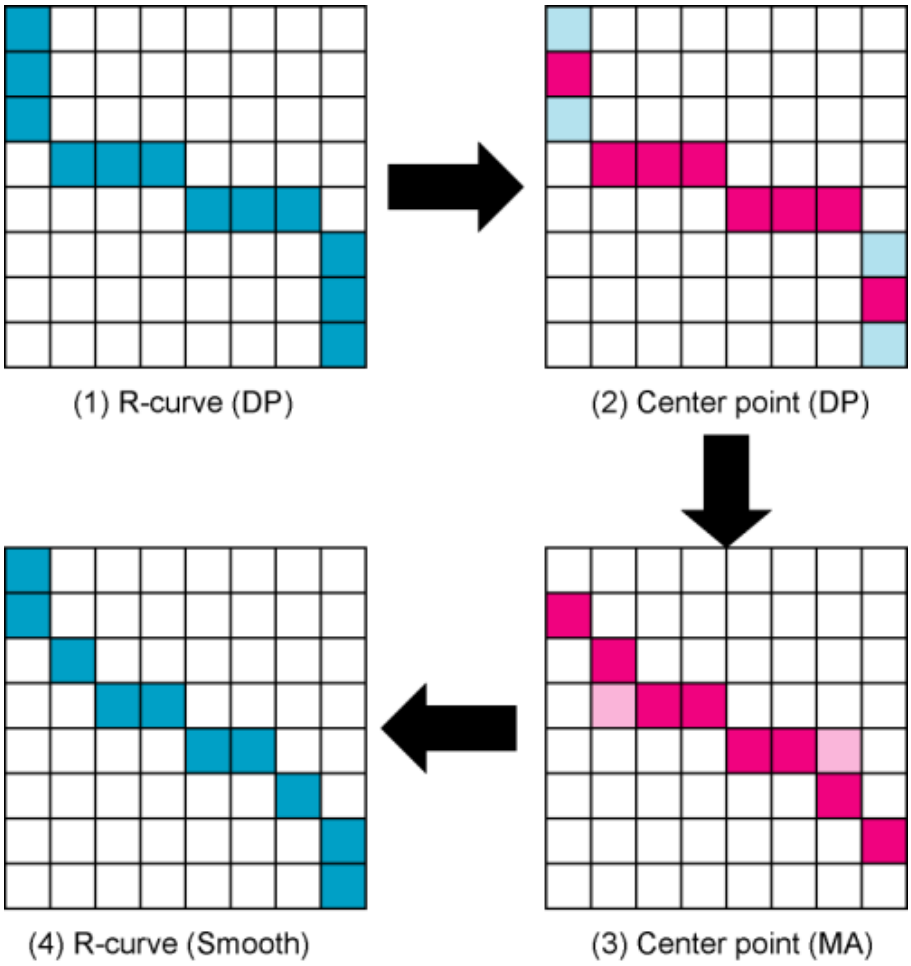
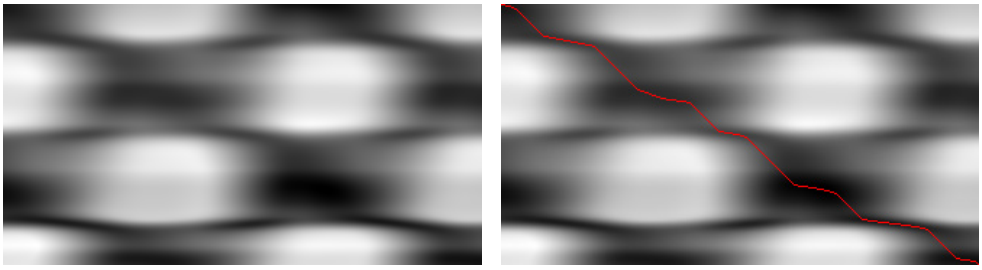
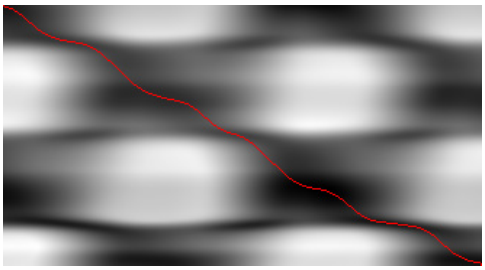


Fig.6 R-curveの平滑化

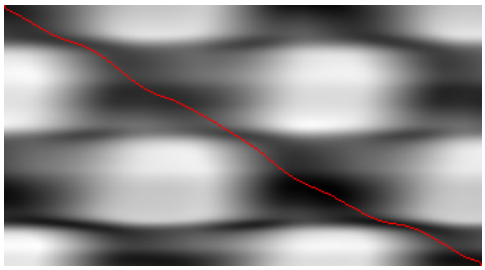
平滑化による R-curve の変化と，アニメーションの同期結果の変化を下表にまとめます．平滑化を行わない場合にはガタついた R-curve が得られますが，フィルタ幅を大きくするにつれて滑らかさが増す様子が確認できます．



入力歩行動作(m1vムービー, 206KB)



平滑化なし(m1vムービー, 223KB)



フィルタ幅=10(m1vムービー, 224KB)

フィルタ幅=20(m1vムービー, 224KB)

そして, $M \times N$ の距離行列 D を, Fig.7に示すような $M \times N$ の時間変換行列 W と呼ばれる行列に変換します(※管理者による呼称). 詳細は後述しますが, 変換元のフレーム番号を順番に格納したベクトルと時間変換行列の乗算によって, 変換後のフレーム番号を格納した次元のベクトルが求められます.

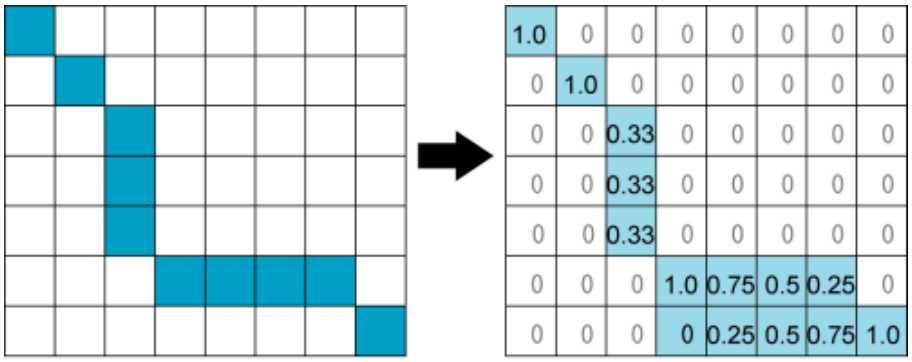


Fig.7 R-curveと時間変換行列

時間変換行列の各成分は R-curve の勾配に基づいて, 以下に示すプログラムによって計算されます. R-curve が距離行列上で水平方向もしくは垂直方向に連続する回数をカウントし, カウント数に基づいて変換行列の各成分を決定します. なお, STL::pair の成分はそれぞれ pair::first が行, pair::second が列に対応します.

```
// path: R-curve
bool CTimewarpMatrix::Initialize(const std::vector<pair<int, int>> &path)
{
  CMatrix::Initialize(path.back().first + 1, path.back().second + 1, 0.0);
  size_t radj =1, cadj =1;

  for (std::vector<pair<int, int>>::const_iterator it = npath.begin();it != npath.end();++it) {
    if (it != npath.end() - 1 && it->first == (it + 1)->first) {
      radj++;
      cadj = 1;
    }
    else if (it != npath.end() - 1 && it->second == (it + 1)->second) {
      radj = 1;
      cadj++;
    } else {
      if (radj == 1 && cadj == 1) {
        Data(it->first, it->second) = 1.0;
      } else if (cadj > 1) {
        for (size_t r = 0;r < cadj;r++)
          Data(it->first - r, it->second) = 1.0 / static_cast<double>(cadj);
      } else if (it == npath.end() - 1) {
        for (size_t c = 1;c <= radj;c++)
          Data(it->first, it->second - radj + c) = 1.0;
      } else {
        for (size_t c = 1;c <= radj;c++) {
          Data(it->first + 1, it->second - radj + c) = static_cast<double>(c - 1) / static_cast<double>(radj);
          Data(it->first, it->second - radj + c) = 1.0 - Data(it->first + 1, it->second - radj + c);
        }
      }
      radj = 1; cadj = 1;
    }
  }
  return true;
}
```

次に，時間変換対象となるモーションデータのフレーム番号を順番に格納したベクトル $s_m=[1.2.\cdots.M]$ を作成します．最後に，ベクトル s_m と時間変換行列 W の積により，時間変換結果のフレーム番号を格納した N 次元のベクトル $s=s_mW$ が求まります．Fig.7 の例では，s_m=[1 2 3 4 5 6 7]なるベクトルを時間変換行列で変換すると，s=[1 2 4 6 6.25 6.5 6.75 7] が得られます．この結果より，変換後のモーションの3フレーム目には元の動作の4フレーム目の姿勢が，4フレーム目には元の動作の6フレーム目の姿勢が格納されることがわかります．

漸次的時間逆変換：Incremental Inverse Timewarping

同期処理を経てブレンディングされた動作は，時間逆変換処理によって適切な時間長やタイミングに変換されます．例えば，$G(n)$ に同期した $F'(t)$ は，$G(n)\rightarrow F(m)$ への時間変換と同一の処理手順によって時間逆変換されます．

ただし，モーションブレンディングにおける時間逆変換には，漸次的時間逆変換（Incremental Inverse Timewarping）というアルゴリズムが用いられます．このアルゴリズムでは，動作のブレンド率の変化に応じて合成動作の時間長を簡単に変化させることができます．一方，非漸次的なアルゴリズムでは，合成動作に時間的な逆戻りが生じるなどの不具合が発生することが指摘されています[参考文献 3]．時間変換行列を用いる場合の漸次的時間変換アルゴリズムは次のような手順で処理されます．

- 入力：2 つの動作 $F(m)$，$G(n)$ と，$G(n)$ に同期した動作 $F'(t)$
- 出力：$F'(t)$の時間逆変換結果 $F^*(m)$

1. $F(m)$ から $G(n)$ への時間変換 $s=s_mW$ の計算
2. $d[n]=s[n+1]-s[n]$．ただし，$d[N]=1.0$．
3. $F^*(1)=F(1)$，$t=1$ を初期値とし，$F^*(t+d[n])=F'(n)$，$t=t+d[n]$の処理を $n\leq N$ の間繰り返す．

s ではなく，そのフレーム差分情報である d を用いる点が異なります．このとき，合成動作の時間長 T は $T=\sum_n d[n]$ で与えられます．なお，上記のアルゴリズムでは時間変換した動作を元の長さに逆変換するだけですが，実際のブレンディングでは，それぞれのサンプル動作に対応する d_i の加重和 $d=\sum w_i d_i$ によって時間逆変換することになります．

時間変換行列から漸次的時間変換関数 d を計算するコードを以下に示します．

```
 1  CVector CTimewarpMatrix::GetIITW() const
 2  {
 3    CVector iitw(Row());
 4    for (size_t f = 0;f < iitw.Size();++f)
 5      iitw[f] = static_cast<double>(f);
 6    iitw = Transform(iitw);
 7
 8    for (size_t f = 0;f < iitw.Size() - 1;++f)
 9      iitw[f] = iitw[f + 1] - iitw[f];
10    iitw[iitw.Size() - 1] = 1.0;
11    return iitw;
12  }
```

アルゴリズムのまとめ

モーションブレンディングの具体的な処理手順を以下にまとめます．

- 入力：X 個のサンプル動作 $Y_x(t_x)$ と 補間カーネル k_x．ただし $x\in[1\cdots X]$，$\sum_x k_x=1.0$
- 出力：合成動作 $Z(t)$

1. 同期処理
   - $Y_0(t_0)$ を基準動作とする．動作：$Y_(t_0)\rightarrow Y'_0(i)$，$i\in[1\cdots T_0]$．基準動作の時間変換：$s_0=[1\cdots T_0]$
   - 他のサンプルの同期処理．動作：$Y_x(t_x)\rightarrow Y'_x(i)$．時間変換：$s_x[i]$．$i\in[1\cdots T_0]$
   - 漸次的時間変換の計算．$d_x[i]=s_x[i+1]-s_x[i]$
2. 補間処理
   - 各時間フレームごとに姿勢を補間：$Z'(i)=\sum_x k_x Y'_x(i)$．
   - 漸次的時間変換を補間：$d[i]=\sum_x k_x d_x[i]$
3. 時間逆変換
   - 補間された漸次的時間変換 d を用いて，補間動作を時間逆変換 $Z'(i)\rightarrow Z(t)$．合成動作の時間長

$$T = \sum_i d[i] = \sum_i \sum_x k_x d_x[i]$$

実装

動作のフィルタリングのプログラムを拡張することで，モーションブレンディングのサンプルプログラムを作成しています．追加したクラスや名前空間，関数は主に次の通りです．

- NFigureUtil 名前空間の追加
  - CalcTransformBetweenFigures 関数
    - 点群の座標正規化行列の計算
  - PoseDistance 関数
    - 身体部位の位置による姿勢距離の計算
- NMotionSingal 名前空間への関数の追加
  - Synchronize 関数
    - 動作の同期処理
  - IncrementalTimewarp 関数
    - 漸次的時間変換
  - Blend 関数
    - モーションブレンディング
- CDistanceMatrix クラス
  - 距離行列と R-Curve
- CTimewarpMatrix クラス
  - 時間変換行列と(非漸次的)時間変換

以下に，主要な関数のコードを示します．

動的計画法によるコスト最小経路探索

格子状にノードが配置される場合の動的計画法の計算です．まず，距離行列に探索範囲制限マスクを加算し，その結果に対して動的計画法を適用しています．STL::pair を使っていますが，pair::first が行，pair::second が列に対応しています．なお，始端から終端へ向う R-curve を取得するために，後退代入によって得られる経路情報を最後に反転しています(51行)．

```
 1  std::vector< std::pair<int, int> > CDistanceMatrix::CalcRegistrationCurve(float ratio) const
 2  {
 3    std::vectorint, int>> path;
 4    CDistanceMatrix tmp(*this);
 5    tmp += gen_restriction_mask_belt(tmp.Row(), tmp.Col(), ratio);
 6
 7    // DP: Forward
 8    for (int type = 0;type < 2;++type) {
 9      int loopmax = type == 0 ? Row() : Col();
10      for (int loop = 1;loop < loopmax;++loop) {
11        int r = type == 0 ? loop : Row() - 1;
12        int c = type == 0 ? 0 : loop;
13        while (IsRange(r, c)) {
14          double mincost = DBL_MAX;
15          if (IsRange(r - 1, c) && tmp(r - 1, c) < mincost)
16            mincost = tmp(r - 1, c);
17          if (IsRange(r, c - 1) && tmp(r, c - 1) < mincost)
18            mincost = tmp(r, c - 1);
19          if (IsRange(r - 1, c - 1) && tmp(r - 1, c - 1) < mincost)
20            mincost = tmp(r - 1, c - 1);
21          tmp(r, c) += mincost;
22          --r; ++c;
23        }
24      }
25    }
26
27    // DP: Backward
28    std::pair<int, int> pos(Row() - 1, Col() - 1);
29    path.push_back(pos);
30    while (path.back().first != 0 || path.back().second != 0) {
```

```
31 |      double mincost = DBL_MAX;
32 |      pos = path.back();
33 |      path.push_back(pos);
34 |      if (IsRange(pos.first - 1, pos.second) && tmp(pos.first - 1, pos.second) < mincost) {
35 |        mincost = tmp(pos.first - 1, pos.second);
36 |        path.back().first = pos.first - 1;
37 |        path.back().second = pos.second;
38 |      }
39 |      if (IsRange(pos.first, pos.second - 1) && tmp(pos.first, pos.second - 1) < mincost) {
40 |        mincost = tmp(pos.first, pos.second - 1);
41 |        path.back().first = pos.first;
42 |        path.back().second = pos.second - 1;
43 |      }
44 |      if (IsRange(pos.first - 1, pos.second - 1) && tmp(pos.first - 1, pos.second - 1) < mincost) {
45 |        mincost = tmp(pos.first - 1, pos.second - 1);
46 |        path.back().first = pos.first - 1;
47 |        path.back().second = pos.second - 1;
48 |      }
49 |    }
50 |
51 |    // 末端から格納されている経路情報の反転
52 |    std::reverse(path.begin(), path.end());
53 |    return path;
54 |  }
```

R-curveの平滑化 [†]

R-curveを移動平均カーネルを用いて平滑化する処理を以下に示します. 引数 filter を半径とする移動平均カーネルを利用して R-curve を平滑化します.

```
 1  std::vector<int, int>> CDistanceMatrix::CalcSmoothRegistrationCurve(float ratio, size_t filter) const
 2  {
 3    std::vector<float> kernel(filter * 2 + 1, 1.0f / static_cast<float>(filter * 2 + 1));
 4    std::vector<int, int>> org = CalcRegistrationCurve(ratio);
 5    std::vector<int> opnts, npnts;
 6
 7    // 各列についてR-curveの中点を計算
 8    int back = 0;
 9    for (std::vector<int, int>>::const_iterator cit = org.begin();cit != org.end(); ) {
10      int back = cit->second;
11      int min = cit->first, max = cit->first;
12      while (++cit, cit != org.end() && cit->second == back)
13        max = cit->first;
14      opnts.push_back((min + max) / 2);
15    }
16
17    // 中点系列に対する平滑化
18    for (int c = 0;c < opnts.size();++c) {
19      float v = 0;
20      for (int k = 0;k < kernel.size();++k) {
21        int p = c + (k - kernel.size() / 2);
22        v += kernel[k] * opnts[p < 0 ? 0 : p >= opnts.size() ? opnts.size() - 1 : p];
23      }
24      npnts.push_back(static_cast<int>(v));
25    }
26
27    // 平滑化された中点系列からのR-curveの再計算
28    std::vector<int, int>> path;
29    back = 0;
30    for (int r = 0;r < npnts.size() - 1;++r) {
31      int next = (npnts[r] + npnts[r + 1] + 1) / 2;
32      for (int i = back;i <= next;++i) {
33        if (path.size() < 2 || (path.end() - 2)->first != i - 1 || (path.end() - 2)->second != r - 1)
34          path.push_back(std::pair<int, int>(i, r));
35        else {
36          path.back().first = i;
37          path.back().second = r;
38        }
39      }
40      back = next;
41    }
42    for (int i = back;i <= org.back().first;++i) {
43      if (path.size() < 2 || (path.end() - 2)->first != i - 1 || (path.end() - 2)->second != npnts.size() - 2)
44        path.push_back(std::pair<int, int>(i, npnts.size() - 1));
45      else {
46        path.back().first = i;
47        path.back().second = npnts.size() - 1;
```

```
48 !      }
49 !    }
50 |    return path;
51 ! }
```

同期処理 [†]

同期処理を行う NMotionSignal::Synchronize 関数では，距離行列の計算（7～14行），（R-curve の取得，17行），モーションデータの時間変換（18～19行），そして漸次的時間逆変換を計算します（22～29行）．

```
 1    bool NMotionSignal::Synchronize(CMotionData &dest, CVector &iitw,
 2                         const CMotionData &src, const CMotionData &base, const CFigure &fig)
 3  {
 4 |    CFigure figs = fig, figb = fig;
 5 |
 6 |    // 距離行列の計算
 7 !    CDistanceMatrix dmat(src.NumFrames(), base.NumFrames());
 8 |    for (size_t i = 0;i < src.NumFrames();++i) {
 9 |      figs.SetPose(&src, i);
10 |      for (size_t j = 0;j < base.NumFrames();++j) {
11 |        figb.SetPose(&base, j);
12 |        dmat(i, j) = NFigureUtil::PoseDistance(figs, figb);
13 !      }
14 !    }
15 |
16 |    // registration curveの計算
17 !    std::vector<int, int>> path = dmat.CalcSmoothRegistrationCurve(0.7f, 10);
18 |    // timewarp行列への変換
19 !    CTimewarpMatrix twm(path);
20 |    // 時間変換
21 !    dest = twm.Transform(src);
22 |    // 漸次的時間逆変換関数
23 !    iitw = twm.GetIITW();
24 |
25 |    return true;
26 ! }
```

漸次的時間変換 [†]

漸次的時間変換はオンライン計算を想定して考案されたものですが，このサンプルではあらかじめ合成動作の時間長を計算し，全てのフレームの姿勢を計算しています．

```
 1    bool NMotionSignal::IncrementalTimewarp(CMotionData &dest, const CMotionData &src, const CVector &itw)
 2  {
 3 |    double frm = 0;
 4 |    for (size_t i = 0;i < itw.Size();++i)
 5 |      frm += itw[i];
 6 |    dest.Initialize(static_cast(frm), src.NumJoints());
 7 |
 8 |    // 初期フレームの姿勢はそのままコピー
 9 !    dest.SetPosition(0, src.GetPosition(0));
10 |    for (size_t j = 0;j < src.NumJoints();j++)
11 |      dest.SetRotation(0, j, src.GetRotation(0, j));
12 |
13 |    double back = 0, pos = itw[0];
14 |    for (size_t i = 1;i < itw.Size();++i) {
15 |      size_t f0 = static_cast(ceil(back));
16 |      size_t f1 = static_cast(floor(pos));
17 |      for (size_t f = f0;f <= f1;++f) {
18 |        float t = (static_cast<float>(f) - static_cast<float>(back)) / static_cast<float>(pos - back);
19 |
20 |        D3DXVECTOR3 v1 = src.GetPosition(i - 1), v2 = src.GetPosition(i);
21 |        dest.SetPosition(f, v1 * (1.0f - t) + t * v2);
22 |        for (size_t j = 0;j < src.NumJoints();++j) {
23 |          D3DXQUATERNION q, q1 = src.GetRotation(i - 1, j), q2 = src.GetRotation(i, j);
24 |          D3DXQuaternionSlerp(&q, &q1, &q2, t);
25 |          dest.SetRotation(f, j, q);
26 !        }
27 !      }
28 |      if (f1 - f0 >= 0)
29 |        back = pos;
30 |      pos += itw[i];
31 !    }
32 |    for (size_t f = static_cast(ceil(back));f < dest.NumFrames();++f) {
```

```
33 |      dest.SetPosition(f, src.GetPosition(src.NumFrames() - 1));
34 |      for (size_t j = 0;j < dest.NumJoints();++j)
35 |        dest.SetRotation(f, j, src.GetRotation(src.NumFrames() - 1, j));
36 !    }
37 |    return true;
38 ! }
```

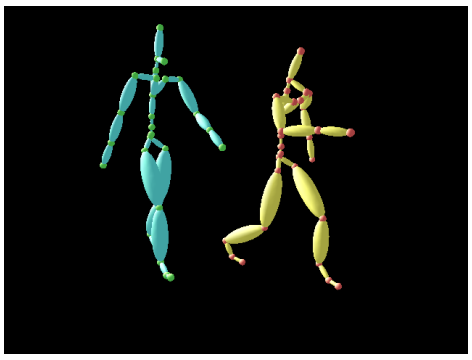
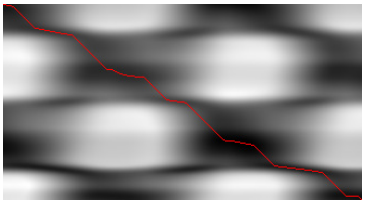
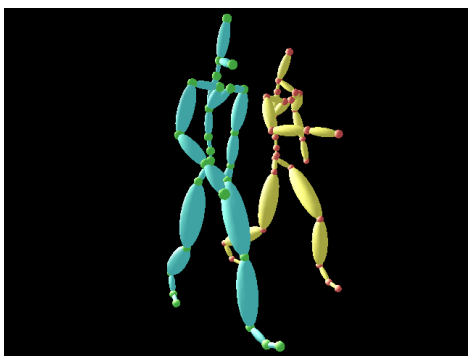
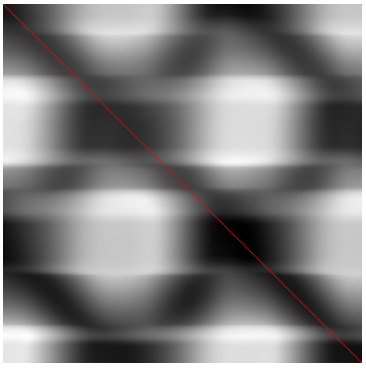
ブレンディング [t]

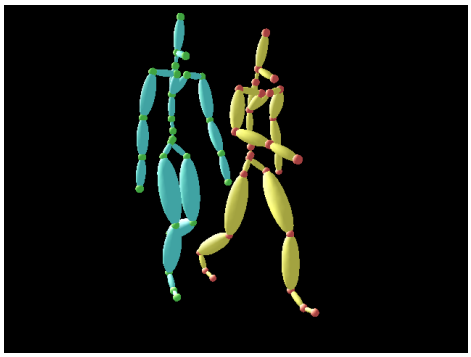
ブレンディング処理の本体になります. std::vector で渡された先頭のサンプル動作を基準として, 他の動作の同期処理と漸次的時間変換を計算し(9～13行), 各時間フレームにおける姿勢の補間(15行), 補間動作の時間逆変換(16行)の順で処理します.

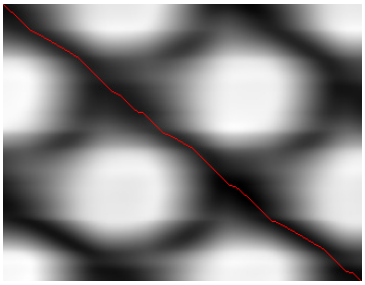
```
 1    bool NMotionSignal::Blend(CMotionData &dest, const std::vector &srcs,
 2                              const std::vector<float> &kernel, const CFigure &fig)
 3  {
 4 |  CMotionData blended;
 5 |  std::vector mtmp;
 6 |  CVector vtmp, iitw(srcs.front()->NumFrames(), kernel.front());
 7 |
 8 |  mtmp.push_back(srcs.front());
 9  |  for (size_t i = 1;i < srcs.size();++i) {
10 |    mtmp.push_back(new CMotionData);
11 |    Synchronize(*mtmp.back(), vtmp, *srcs[i], *srcs[0], fig);
12 |    iitw += vtmp * kernel[i];
13 !  }
14 |
15 |  Interpolate(blended, mtmp, kernel);
16 |  IncrementalTimewarp(dest, blended, iitw);
17 |
18  |  while (mtmp.size() > 1) {
19 |    delete mtmp.back();
20 |    mtmp.pop_back();
21 !  }
22 |  return true;
23 ! }
```

処理例:歩行動作のブレンディング [t]

3歩分の普通の歩行動作と, 忍び歩き動作のブレンディング結果を示します. 表の1行目はオリジナルの歩行動作, 2行目は普通の歩行動作を基準として忍び歩き動作を同期させた結果, そして 3 行目は 50:50 で 2 つの動作をブレンディングした結果です.

3歩目の接地時の姿勢	R-curve	ムービー
		入力歩行動作(m1v, 206KB)
		同期処理(m1v, 223KB)





ブレンディング結果(m1v, 215KB)

まとめ

一般的なモーションブレンディングの処理手順を説明しました. また, 姿勢距離測度や距離行列, registration curve など, 他の動作編集技法にとっても重要な技術も簡単に説明しました. 今回紹介したアルゴリズムは実装が比較的簡単で, パラメータチューニングも不要な使い勝手の良い手法だと思います. ただし, 距離行列の作成が計算速度の大きなボトルネックになるので, この点は改善の余地がありそうです.

Last-modified: 2006-12-16 (土) 20:04:21 (2690d)

Site admin: cherub