モーションスプライシング

http://www.tmps.org/index.php?%A5%E2%A1%BC%A5%B7%A5%E7%A5%F3%A5%B9%A5%D7%A5%E9%A5%A4%A5%B7%A5%F3%A5%B0

今回は，モーションスプライシング（注：管理人による呼称）と呼ばれる動作データ編集技術を実装してみました．スプライシング（splicing）という言葉は聞き慣れないかもしれませんが，「あるデータから一部分を取り除き、別のデータの一部分を結合する」といった操作を意味します．今回紹介する技術では「ある歩行動作から上半身の動作を取り除き，別の歩行動作の上半身動作を結合する」というスプライシングを実現します．例えば，旋回軌道の歩行動作と直線軌道の荷持ち歩行動作から，旋回軌道の荷持ち歩行動作を計算できます．このアルゴリズムは，下半身が周期的に運動するような動作データに適用が限られるなど制約も多いのですが，比較的応用範囲が広く実用的な技術だと思います．

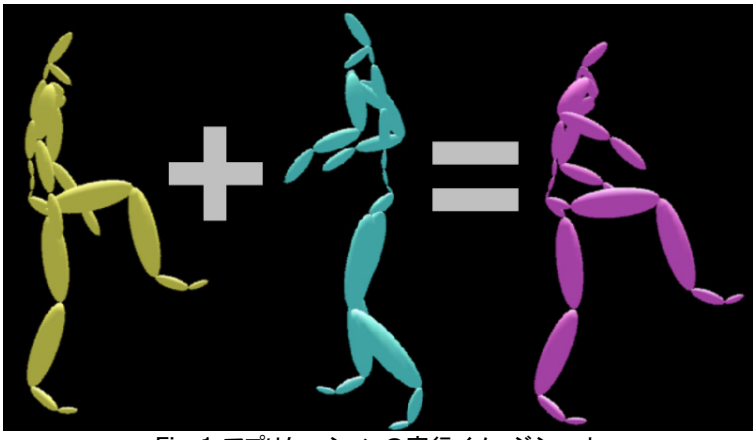


Fig.1 アプリケーションの実行イメージショット

サンプルプログラム [†]

サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update June 2006 の環境下で，MFC を用いて作成しており，コンパイルには GNU Scientific Library が必要です．アーカイブには，いくつかの歩行動作の BVH モーションキャプチャデータも含めてあります．今回もバイナリデータをアップしておきます．DirectX SDK (June 2006) もしくは DirectX End User Runtime (June 2006) をインストールして遊んでみてください．

サンプルソースコード(VC++2005, MFC, 360kb)

サンプルバイナリ(604kb)

遊び方 [†]

サンプルプログラムは特定のスケルトン構造を持つ MOCAP データファイルに特化して作成されています．そのため，ほとんどの場合正常動作しないと思われますが，あくまでもサンプルプログラムだということでご了承ください．

1. 実行形式ファイル（MotionSplicing.exe）とエフェクトファイル（Viewer.fx）を同じディレクトリに置きます
2. プログラムを起動して [ファイル]→[開く]から，下半身動作のサンプル→上半身動作のサンプルの順に，2つの BVH モーションデータファイルをロードします
3. [Motion]→[Temporal-align]で 2 つのサンプル動作を同期させます
4. [Motion]→[Spatial-align]で上半身姿勢を補正します
5. [Motion]→[Splicing]で上半身動作をコピーします
6. [Motion]→[Transfer]で最終的な補正を施します

- スプライシング結果は，[ファイル]→[名前を付けて保存]で BVH 形式で保存できます

アーカイブにはこの記事に掲載していないモーションデータも含めていますので，実際に遊んでスプライシングの効果を確かめてみてください．

参考文献

1. Rachel Heck, Lucas Kovar, and Michael Gleicher, Splicing Upper-Body Actions with Locomotion, Computer Graphics Forum (Eurographics 2006), vol.17, no.3-4, pp.219-227, 2006.
2. Berthold K.P. Horn, Closed-Form Solution of Absolute Orientation using Unit Quaternions", Journal of the Optical Society A, vol.4, no.4, pp.629–642, 1987.
3. Lucas Kovar and Michael Gleicher, "Flexible Automatic Motion Blending with Registration Curves", Symposium on Computer Animation 2003, pp.214-224, 2003.

今回実装したシステムは，論文[1]をベースにしつつ，いくつかの計算を簡略化したものです．そのため，オリジナルほど良い結果を与えない場合があるかもしれません．また，このシステムの技術的なコアとなる論文が[2]と[3]です．論文[2]はモーションブレンディングの記事で既に紹介した Registration curves に関する論文です．また論文[3]は，2 つの点群間の距離の二乗総和を最小化するような回転トランスフォームを，Closed-form に計算するアルゴリズムについて述べています．

スプライシング処理

この技法の目的は，ある歩行動作の上半身動作を，別の歩行動作の上半身動作で置き換えることです．ここで，上半身とは骨盤直上の関節から頭頂もしくは手先にいたる部位を指し，下半身とはルートから足先に至る部位を指します．そして，一方のサンプルから下半身動作を，もう一方のサンプルから 骨盤関節以外の上半身動作をコピーすることで新しい動作を生成します．その際，上半身と下半身動作を継ぎ換えても違和感が生じないように（2 つのサンプル動作の特徴を失わないように），骨盤関節の新しい回転量を計算することがこの技法の焦点となります．

実装したシステムでは，次のような動作データのみをスプライシングできます．

- 下半身動作は（ある程度）周期的な歩行動作．上半身動作はどのような動作でもOK.
- 2つの歩行動作のステップ数やステップ順序は同一．

ここで，上半身動作のサンプル歩行動作データを A，下半身動作のサンプル歩行動作データを B と表記します．また，それぞれの上半身動作データを A_U と B_U，下半身動作データを A_L と B_L のように表します．このとき，単純なスプライシングによって生成される動作 C は，$C_U = A_U$ と$C_L = B_L$ と表すことができます．

ただし，単純に $C_U = A_U$，$C_L = B_L$ のようにデータをそのままコピーするだけでは不自然な結果を生じますので，いくつかの前処理や補正処理を施す必要があります．論文[1] では以下に示す 3 段階のアルゴリズムを提案しています．

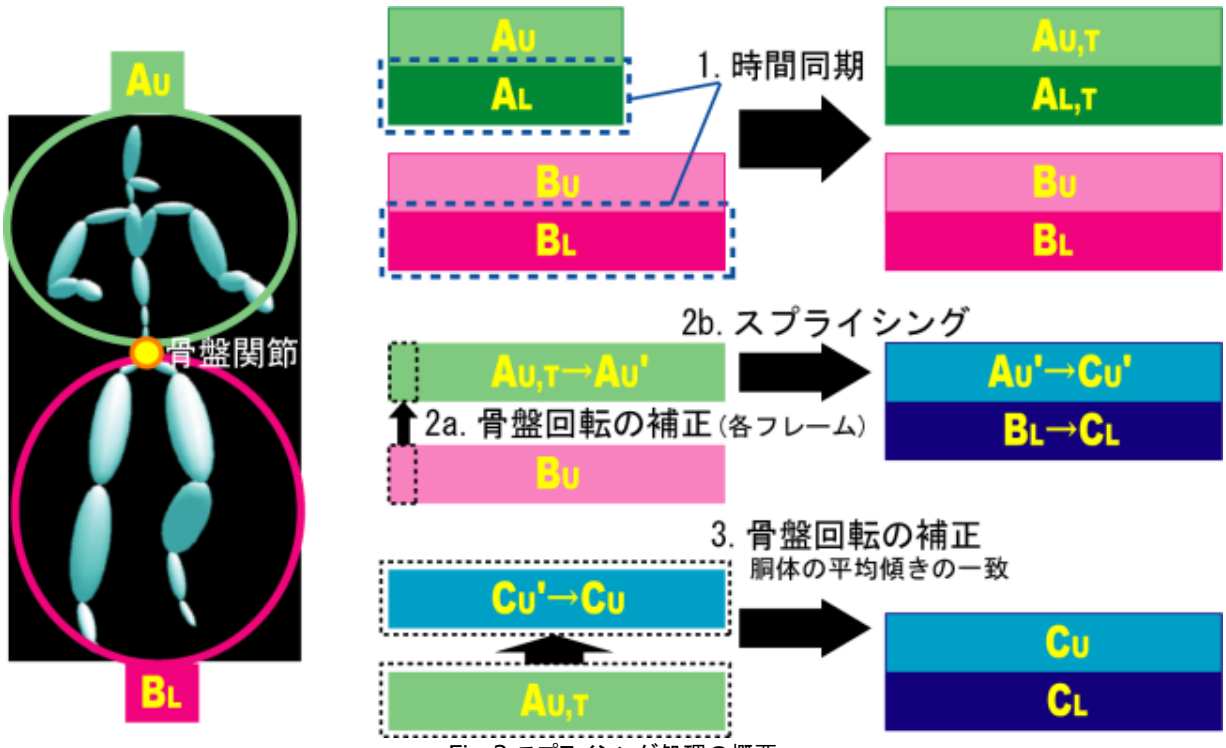


Fig.2 スプライシング処理の概要

1. 下半身動作のみを用いた動作の時間的正規化
   - 接地のタイミングなどを同期させることで，2 つの歩行動作の時間長や動作のタイミングを正規化します．
2. フレーム単位での上半身姿勢の補正とスプライシング
   - サンプル動作における上半身と下半身の連動関係を，スプライシング動作に反映するための処理です．各フレームにおいて，コピー先の上半身姿勢 B_U の胴体の傾きにコピー元 A_U の胴体の傾きが一致するように，A_U の骨盤関節の回転を補正します．そして，補正した上半身動作を $A'_U \rightarrow C'_U$，下半身動作を $B_L \rightarrow C_L$ のようにコピーします．
3. 動作全体にわたる大域的な上半身姿勢の補正
   - 動作全体にわたるサンプル上半身姿勢の特徴を，スプライシング動作に反映するための処理です．スプライシング後の上半身動作 C'_U における胴体の平均傾きが，オリジナル A_U の胴体の平均傾きに一致するように，C'_U の骨盤関節の回転を補正します．

下肢動作のみを用いた歩行動作の同期:Temporal aligning [†]

ほとんどの場合，動作 A と B は時間長や動作のタイミングは異なります．そのため，単純に $C_U = A_U$, $C_L = B_L$ のようにデータをコピーするだけでは，時間長の不一致などの不具合を生じます．例えば，通常の歩行動作では左手と右足，右手と左足が同時に前後しますが，単純なスプライシングではこれらの位相がずれてしまい，右手と右足，左手と左足が連動するような不自然な（緊張した人のような？）歩行になってしまいます．そこで，モーションブレンディングの記事で解説した動的時間伸縮法（DTW 法）を用いて A を時間変換し，2 つの動作を同期させます．

ただしこのアルゴリズムでは，上半身の動作は無視して下半身の動作だけが同期するように時間変換します．これは，見た目が大きく異なる 2 つの上半身動作を無理に同期させようとすると，接地などの重要なタイミングを考慮できなくなるためです．また，歩行動作における上半身と下半身の連動関係を考えれば，下半身動作さえ同期すれば上半身動作もそれに従うと考えられるからです．

下半身運動にもとづく同期処理の実装コードを以下に示します．モーションブレンディングで実装した NMotionSignal::Synchronize 関数とほぼ同一のコードですが，2 姿勢間の距離の算出関数を NFigureUtil::PoseDistance 関数から NFigureUtil::PoseDistanceLowerBody に変更しています．

```
 1  bool NMotionSignal::SynchronizeLowerBody(CMotionData &dest, CVector& iitw,
 2                       const CMotionData &src, const CMotionData &base, const CFigure &fig)
 3  {
 4    CFigure figs = fig, figb = fig;
 5
 6    // 下半身のみを対象とする距離行列の計算
 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::PoseDistanceLowerBody(figs, figb);
13      }
14    }
15
16    std::vectorint, int>> path = dmat.CalcRegistrationCurve();
17    CTimewarpMatrix twm(path);
18    dest = twm.Transform(src);
19
20    iitw.Initialize(src.NumFrames());
21    for (size_t f = 0;f < iitw.Size();++f)
22      iitw[f] = static_cast<double>(f);
23    iitw = twm.Transform(iitw);
24
25    for (size_t f = 0;f < iitw.Size() - 1;++f)
26      iitw[f] = iitw[f + 1] - iitw[f];
27    iitw[iitw.Size() - 1] = 1.0;
28
29    return true;
30  }
```

NFigureUtil::PoseDistanceLowerBody 関数では，下半身の各関節間のユークリッド距離の二乗総和によって，下半身姿勢の姿勢距離を算出します．

```
1  float NFigureUtil::PoseDistanceLowerBody(const CFigure &fig0, const CFigure &fig1)
2  {
3    std::vector pc0 = fig0.GetPointCloudLowerBody();
4    std::vector pc1 = fig1.GetPointCloudLowerBody();
```

```
  return calc_pose_distance(pc0, pc1);
}
```

下半身の各関節位置を示す点群は，CFigure::GetPointCloudLowerBody 関数で求められます．ただし，オリジナルの論文[1]ではルート位置と両ひざ位置のみを用いていますが，ここでは下半身を構成する全ての関節位置を点群として扱っています．これは，オリジナルの方法ではうまく同期できなかったためですが，何か別の理由（実装ミス？）によるかもしれません．また，下半身の始点関節ノード名はソースコード中で指定していますので，別形式の動作データファイルを扱う場合にはプログラムの変更が必要になります（これは単なる手抜きです）．

```
std::vector CFigure::GetPointCloudLowerBody() const
{
  std::vector pc;
  LPD3DXMATRIXSTACK pMatStack;
  const CJoint *pRoot = GetJoint( FindNode("sacroiliac") );

  if (FAILED(D3DXCreateMatrixStack(0, &pMatStack)))
    return pc;

  pMatStack->LoadIdentity();
  pMatStack->MultMatrixLocal(&pRoot->GetTransformMatrix());
  for (size_t j = 0;j < pRoot->NumChildren();j++)
    GetPointCloudSub(pc, pRoot->GetChild(j), pMatStack);
  pMatStack->Release();

  return pc;
}
```

フレーム単位の上半身姿勢の補正とスプライシング：Spatial aligning and splicing

歩行動作における上半身の姿勢は，下半身動作に連動して変化すると考えられます．そのため単純なデータコピーによるスプライシングでは，上半身と下半身の連動関係が失われてしまいます．例えば Fig.3 の黄色スケルトンに示す忍び足歩行を見ると，足を前に踏み出す際の全身バランスを保つため，上半身は後方へと傾いています．しかし，この忍び足の下半身動作に平常時の歩行動作の上半身動作を単純に接続すると，足の動きに関係なく胴体は常に直立してしまうことになります．こうした不具合を解消するためには，少なくともコピー先 B_U の胴体とコピー元 A_U の胴体の方向が一致しなければなりません．姿勢補正処理では，こうした連動関係を満たすための骨盤関節の回転角を計算します．

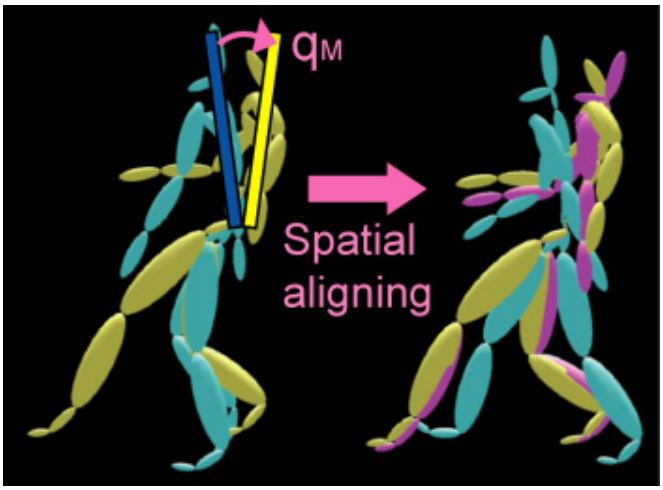


Fig.3 姿勢補正処理

姿勢補正処理では，各時間フレームごとに新しい骨盤関節の回転角を計算します．まず，コピー先 B_U の上半身関節位置を表す点群とコピー元 A_U の点群のユークリッド距離の二乗和を最小化するような，コピー元の骨盤回転 q_a への補正回転量 q_m を計算します．そして，補正を加えた回転量 $q_a q_m$ を新しい骨盤関節の回転量とします．

上半身の関節位置を表す点群は，骨盤関節と胴体中央の関節，そして両肩の 4 つの位置から構成されます．実装コードは次の通りです．

```
std::vector CFigure::GetPointCloudUpperBody() const
{
  std::vector pc(4);
  pc[0] = GetWorldPosition( FindNode("vl5") );
  pc[1] = GetWorldPosition( FindNode("vt6") );
  pc[2] = GetWorldPosition( FindNode("r_shoulder") );
```

```
  pc[3] = GetWorldPosition( FindNode("l_shoulder") );
  return pc;
}
```

この点群間のユークリッド距離の二乗和を最小化するような補正回転量は，参考文献[2]の論文で述べられているアルゴリズムで求められます．ここでの実装コードは，点群 pc0 と 補正後の pc1 間の距離 $\left|pc0-q_m(pc1)\right|$ を最小化するような回転量 q_m を計算します．私は「おまじない」として実装したコードですが，適当な行列を作って固有値分解すると目的のクォータニオンが得られるという，とてもスマートなアルゴリズムです．

```
D3DXQUATERNION NFigureUtil::CalcRotationBetweenPointClouds(const std::vector &pc0,
                                                           const std::vector &pc1)
{
  const float fnum = static_cast<float>(pc0.size());
  D3DXVECTOR3 cent0(0, 0, 0), cent1(0, 0, 0);
  for (size_t i = 0;i < pc0.size();++i)
  {
    cent0 += pc0[i];
    cent1 += pc1[i];
  }
  cent0 /= fnum;
  cent1 /= fnum;

  float sxx =0, sxy =0, sxz =0, syx =0, syy =0, syz =0, szx =0, szy =0, szz =0;
  for (size_t i = 0;i < pc0.size();++i)
  {
    sxx += (pc1[i].x - cent1.x) * (pc0[i].x - cent0.x);
    sxy += (pc1[i].x - cent1.x) * (pc0[i].y - cent0.y);
    sxz += (pc1[i].x - cent1.x) * (pc0[i].z - cent0.z);
    syx += (pc1[i].y - cent1.y) * (pc0[i].x - cent0.x);
    syy += (pc1[i].y - cent1.y) * (pc0[i].y - cent0.y);
    syz += (pc1[i].y - cent1.y) * (pc0[i].z - cent0.z);
    szx += (pc1[i].z - cent1.z) * (pc0[i].x - cent0.x);
    szy += (pc1[i].z - cent1.z) * (pc0[i].y - cent0.y);
    szz += (pc1[i].z - cent1.z) * (pc0[i].z - cent0.z);
  }

  CMatrix mN(4, 4);
  mN(0, 0) = sxx + syy + szz;
  mN(0, 1) = syz - szy;       mN(1, 0) = mN(0, 1);
  mN(0, 2) = szx - sxz;       mN(2, 0) = mN(0, 2);
  mN(0, 3) = sxy - syx;       mN(3, 0) = mN(0, 3);
  mN(1, 1) = sxx - syy - szz;
  mN(1, 2) = sxy + syx;       mN(2, 1) = mN(1, 2);
  mN(1, 3) = szx + sxz;       mN(3, 1) = mN(1, 3);
  mN(2, 2) = -sxx + syy - szz;
  mN(2, 3) = syz + szy;       mN(3, 2) = mN(2, 3);
  mN(3, 3) = -sxx - syy + szz;

  CMatrix evecs;
  CVector evals;
  NStatistics::EVD(evecs, evals, mN);
  gsl_eigen_symmv_sort(evals.m_pvVector, evecs.m_pmMatrix, GSL_EIGEN_SORT_VAL_DESC);

  D3DXQUATERNION nq, q(evecs(1, 0), evecs(2, 0), evecs(3, 0), evecs(0, 0));
  return *D3DXQuaternionNormalize(&nq, &q);
}
```

姿勢補正処理の実装コードは次の通りです．まず，コピー先 B_U とコピー元 A_U のワールド座標系における上半身の関節位置点群を求めます．次に，骨盤関節のローカル座標系がワールド座標系の原点座標系に一致するように，点群の全ての位置を座標変換します．そして，これらの点群間の距離を最小化するような回転量 q_mを求め，A_U の骨盤関節回転量に加えて C_U に格納します．

この実装コードは改良の余地がある(始めから骨盤関節のローカル座標系における点群を計算する等)のですが，あくまでもオリジナルの論文に沿ったコードを掲載しています．

```
void CSceneView::OnMotionSpatialAlign()
{
  const size_t tid = m_pFigure[0]->FindNode("vl5");
  for (size_t frm = 0;frm < m_pMotion[0]->NumFrames();++frm) {
    m_pFigure[0]->SetPose(m_pMotion[0], frm);
    m_pFigure[1]->SetPose(m_pMotion[1], frm);

    std::vector pc0 = m_pFigure[0]->GetPointCloudUpperBody();
```

```
 9 |     std::vector pc1 = m_pFigure[1]->GetPointCloudUpperBody();
10 |
11 |     // pelvis のローカル座標系をワールド座標系原点に一致させるトランスフォーム
12 !     D3DXMATRIX lp0, lp1;
13 |     D3DXMatrixInverse(&lp0, NULL, &m_pFigure[0]->GetWorldCoordinate(tid));
14 |     D3DXMatrixInverse(&lp1, NULL, &m_pFigure[1]->GetWorldCoordinate(tid));
15 |     for (size_t i = 0;i < pc0.size();++i) {
16 |       D3DXVECTOR3 vt;
17 |       pc0[i] = *D3DXVec3TransformCoord(&vt, &pc0[i], &lp0);
18 |       pc1[i] = *D3DXVec3TransformCoord(&vt, &pc1[i], &lp1);
19 !     }
20 |
21 |     // rot(pc1) ≒ pc0 なる回転トランスフォーム qm
22 !     D3DXQUATERNION qm = NFigureUtil::CalcRotationBetweenPointClouds(pc0, pc1);
23 |     // qm により AU の pelvis 回転を補正し CU の pelvis にコピー
24 !     D3DXQUATERNION qc = m_pMotion[1]->GetRotation(frm, tid) * qm;
25 |     m_pMotion[2]->SetRotation(frm, tid, qc);
26 !   }
27 |
28 |   for (size_t i = 0;i < 3;++i)
29 |     m_pFigure[i]->SetPose(m_pMotion[i], m_frame);
30 |   SendMessage(WM_USER_RENDER_SCENE);
31 ! }
```

最後に，骨盤関節を除く上半身動作データ A_U を C_U にコピーします．

```
 1   void CSceneView::OnMotionSplicing()
 2 | {
 3 |   const size_t eid = m_pFigure[1]->FindNode("sacroiliac");
 4 |   const size_t tid = m_pFigure[1]->FindNode("vl5");
 5 |
 6 |   for (size_t frm = 0;frm < m_pMotion[1]->NumFrames();++frm)
 7 |   for (size_t jid = 1;jid < eid;++jid)
 8 |   {
 9 |     if (jid != tid)
10 |     {
11 |       D3DXQUATERNION qc = m_pMotion[1]->GetRotation(frm, jid);
12 |       m_pMotion[2]->SetRotation(frm, jid, qc);
13 !     }
14 !   }
15 |
16 |   for (size_t i = 0;i < 3;++i)
17 |     m_pFigure[i]->SetPose(m_pMotion[i], m_frame);
18 |   SendMessage(WM_USER_RENDER_SCENE);
19 ! }
```

動作単位の上半身姿勢の補正：Posture transfer

姿勢補正処理によって下半身動作 B_L と上半身動作 A_U のスプライシングの違和感を低減できますが，副作用として新しい問題が生じます．例えば，猫背歩行の上半身動作と通常走行の下半身動作のスプライシングでは，走行動作に合わせて背筋を伸ばすことが姿勢補正処理に対応するため，見た目に重要な特徴である猫背姿勢が失われる可能性があります．

そこで，上半身動作 A_U の大域的な傾きの特徴を，スプライシング後の動作に反映するための補正処理を施します．まず，オリジナル上半身動作 A_U とスプライシング後の上半身動作 C_U の関節位置点群の時間平均をそれぞれ求めます．次に，これら平均点群間のユークリッド距離の二乗和を最小化するような補正回転量 q_t を求めます．最後に，全フレームの骨盤関節の回転量に q_t を加えます．

実装したコードを次に示します．まず，各時間フレームにおいてワールド座標系における A_U と C_U の関節位置点群を算出します．次に，骨盤関節をワールド座標系の原点に一致させ，かつ腰の横方向の表すベクトルをワールド座標系の+X軸に一致させるように，これらの点群を座標変換します．この操作を全ての時間フレームについて繰り返し，それぞれれの点群の時間平均位置を算出します．そして，この平均点群間のユークリッド距離の二乗和を最小化するような q_t を計算し，C_U の各時間フレームの骨盤関節の回転量にそれぞれ加えます．

```
 1   void CSceneView::OnMotionTransfer()
 2 | {
 3 |   const size_t tid = m_pFigure[1]->FindNode("vl5");
 4 |   const float fnum = static_cast<float>(m_pMotion[1]->NumFrames());
 5 |
 6 |   std::vector pc1, pc2;
 7 |   for (size_t frm = 0;frm < m_pMotion[1]->NumFrames();++frm) {
```

```
 8 |     m_pFigure[1]->SetPose(m_pMotion[1], frm);
 9 |     m_pFigure[2]->SetPose(m_pMotion[2], frm);
10 |
11 |     // ルート位置
12 !     D3DXVECTOR3 rp1 = m_pFigure[1]->GetRootPosition();
13 |     D3DXVECTOR3 rp2 = m_pFigure[2]->GetRootPosition();
14 |     // ルート方向
15 !     D3DXQUATERNION rq1 = m_pFigure[1]->GetRootOrientation();
16 |     D3DXQUATERNION rq2 = m_pFigure[2]->GetRootOrientation();
17 |     D3DXMATRIX rm1, rm2;
18 |     D3DXMatrixRotationQuaternion(&rm1, &rq1);
19 |     D3DXMatrixRotationQuaternion(&rm2, &rq2);
20 |
21 |     // ワールド座標系における腰横方向ベクトルの算出
22 !     D3DXVECTOR3 vx(1, 0, 0), hv1, hv2;
23 |     D3DXVec3TransformCoord(&hv1, &vx, &rm1);
24 |     D3DXVec3TransformCoord(&hv2, &vx, &rm2);
25 |
26 |     // 腰横方向ベクトルをワールド+X軸に一致させるための回転トランスフォーム行列
27 !     D3DXMATRIX am1 = vector_aligning_rotation(hv1, vx);
28 |     D3DXMATRIX am2 = vector_aligning_rotation(hv2, vx);
29 |
30 |     std::vector tpc1 = m_pFigure[1]->GetPointCloudUpperBody();
31 |     std::vector tpc2 = m_pFigure[2]->GetPointCloudUpperBody();
32 |
33 |     if (pc1.empty()) {
34 |       pc1.resize(tpc1.size(), D3DXVECTOR3(0, 0, 0));
35 |       pc2.resize(tpc2.size(), D3DXVECTOR3(0, 0, 0));
36 !     }
37 |
38 |     D3DXVECTOR3 vt, va;
39 |     for (size_t i = 0;i < tpc1.size();++i) {
40 |       vt = tpc1[i] - rp1;
41 |       pc1[i] += *D3DXVec3TransformCoord(&va, &vt, &am1);
42 |       vt = tpc2[i] - rp2;
43 |       pc2[i] += *D3DXVec3TransformCoord(&va, &vt, &am2);
44 !     }
45 !   }
46 |   for (size_t i = 0;i < pc1.size();++i) {
47 |     pc1[i] /= fnum;
48 |     pc2[i] /= fnum;
49 !   }
50 |
51 |   D3DXQUATERNION qt = NFigureUtil::CalcRotationBetweenPointClouds(pc1, pc2);
52 |   for (size_t frm = 0;frm < m_pMotion[2]->NumFrames();++frm)
53 |   {
54 |     D3DXQUATERNION qc = m_pMotion[2]->GetRotation(frm, tid) * qt;
55 |     m_pMotion[2]->SetRotation(frm, tid, qc);
56 !   }
57 |
58 |   for (size_t i = 0;i < 3;++i)
59 |     m_pFigure[i]->SetPose(m_pMotion[i], m_frame);
60 |   SendMessage(WM_USER_RENDER_SCENE);
61 ! }
```

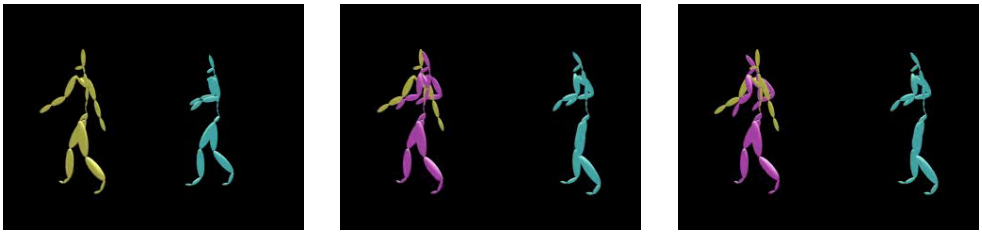
ここで,「腰横方向ベクトルをワールド+X軸に一致させるための回転トランスフォーム行列」を求める関数の実装でハマったので,失敗例も載せておきます.まず,失敗例では Axis-angle による解法を用いました.元ベクトル v_m と一致させたいベクトル v_o の外積により回転主軸を求め,内積により回転量を算出しています.そして,Axis-angle→クォータニオン→回転行列と変換して目的のトランスフォーム行列を得ています.この方法でも確かに変換後の v_m は v_o に一致するのですが,Y 軸や Z 軸方向が過大に変化することがあるので全く使いものになりませんでした.そこで,成功例に示すようなオイラー角への分解アルゴリズムに切り替えたところ,問題なく動作するようになりました.

```
 1 |   #if 0 //失敗 ver.
 2 |   D3DXMATRIX vector_aligning_rotation(const D3DXVECTOR3 &src, const D3DXVECTOR3 &target)
 3 |   {
 4 |     D3DXVECTOR3 av, nav;
 5 |     D3DXVec3Cross(&av, &src, &target);
 6 |     D3DXVec3Normalize(&nav, &av);
 7 |
 8 |     D3DXQUATERNION aq;
 9 |     float angle = acos(D3DXVec3Dot(&src, &target));
10 |     if (fabsf(angle) < 1.0e-6f)
11 |       D3DXQuaternionIdentity(&aq);
12 |     else
13 |       D3DXQuaternionRotationAxis(&aq, &nav, angle);
```

```
14 |
15 |    D3DXMATRIX am;
16 |    return *D3DXMatrixRotationQuaternion(&am, &aq);
17 ! }
18    #else //成功 ver.
19    D3DXMATRIX vector_aligning_rotation(const D3DXVECTOR3 &src, const D3DXVECTOR3 &target)
20 {
21 |    D3DXVECTOR3 src2, target2;
22 |    D3DXMATRIX yms, ymt, tmp;
23 |
24 |    D3DXMatrixRotationY(&yms, atan2f(src.z, src.x));
25 |    D3DXVec3TransformCoord(&src2, &src, &yms);
26 |    D3DXMatrixRotationZ(&tmp, -atan2f(src2.y, src2.x));
27 |    yms *= tmp;
28 |
29 |    D3DXMatrixRotationY(&ymt, atan2f(target.z, target.x));
30 |    D3DXVec3TransformCoord(&target2, &target, &ymt);
31 |    D3DXMatrixRotationZ(&tmp, -atan2f(target2.y, src2.x));
32 |    ymt *= tmp;
33 |    return yms * *D3DXMatrixInverse(&tmp, NULL, &ymt);
34 ! }
35    #endif
```

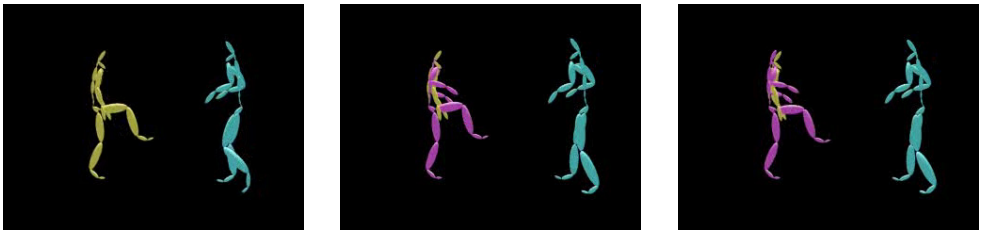
結果

荷運び歩行の上半身動作を歩行動作やはしご登り動作にスプライシングしてみました．3 つのムービーはそれぞれサンプル動作，同期処理・姿勢補正処理・スプライシングを施した結果，そしてさらに Posture Transfer を施した結果に対応します．なお，黄色スケルトンが下半身のサンプル，青色スケルトンが上半身のサンプル，そして桃色スケルトンがスプライシングされた動作を表します．



サンプル(m1v, 155KB) スプライシング(m1v, 179KB) 最終結果(m1v, 179KB)

(a) 荷運び歩行上半身＋通常歩行下半身動作



サンプル(m1v, 155KB) スプライシング(m1v, 179KB) 最終結果(m1v, 179KB)

(b) 荷運び歩行上半身＋障害物乗り越え歩行下半身動作

サンプル(m1v, 155KB) スプライシング(m1v, 179KB) 最終結果(m1v, 179KB)

(c) 荷運び歩行上半身＋はしご登り下半身動作

まとめ

今回は「ある歩行動作から上半身の動作を取り除き，別の歩行動作の上半身動作を結合する」という歩行動作のスプライシング技術を解説しました．このアルゴリズムは歩行動作をターゲットにしてはいますが，下半身がある程度周期的に動作していれば他にもさまざまに応用可能だと思われます．規則的なステップを踏むダンスへの応用などもおもしろいですね．

ただし，Spatial Alignment の処理は大幅な改良の余地があると思われます．オリジナルの論文では「どのような座標系のもとで上半身点群を算出するべきか？」をじっくり議論していますが，結局は骨盤関節のローカル座標系での座標値を算出しています．しかしこのアプローチでは，骨盤関節の回転量が大きく異なる動作において破綻を生じる原因になると思います．とはいえ，確かにこれは難しい問題なので，これからの発展に期待したいところです．

また，今回のサンプルを見るとわかるように，単純な DTW ではスムーズな時間変換が行われないことが多々あります．これは，オリジナル論文のように時間変換曲線を B-Spline 近似することでスムージングするか，より高性能な時間同期処理アルゴリズムを用いることで改善できると思われます．

Last-modified: 2006-10-13 (金) 18:09:12 (2754d)

Site admin: cherub