キャラクタのスケルトン構造

http://www.tmps.org/index.php?%A5%AD%A5%E3%A5%E9%A5%AF%A5%BF%A4%CE%A5%B9%A5%B1%A5%EB%A5%C8%A5%F3%B9%BD%C2%A4

本稿では，キャラクタスケルトン構造の実装方法の一例を紹介します．まず，スケルトン構造の基礎となる木構造（ツリー構造）を拡張し，関節ノード構造を定義します．そして，複数の関節ノードを接続することで，キャラクタのスケルトンを構築します．さらに，構築したスケルトンを描画し，順運動学に基づいて任意の関節の3次元座標を計算する方法についても説明します．なお，本実装はモーションキャプチャデータ（BVHファイルやASF/AMCファイル）を利用することを前提に作成していますので，他の目的においては冗長なコードが含まれることをご了承ください．また，本稿は順運動学のページとDirectX Graphicsによる3DCG描画をベースに記述していますので，それらも合わせてご参照ください．

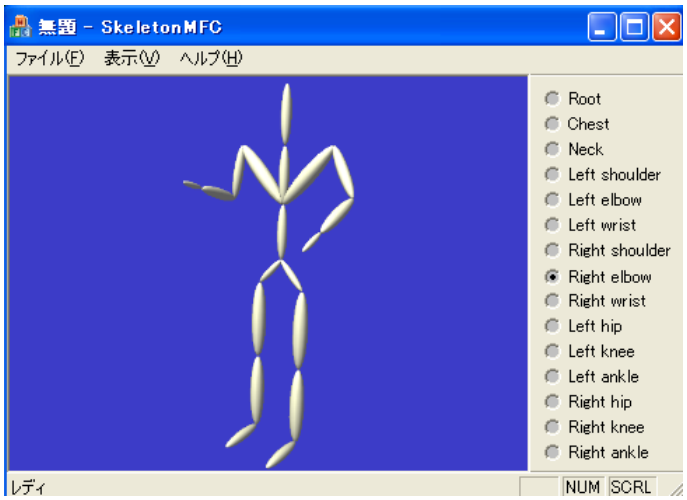

Fig.1 実行画面のスナップショット

サンプルコード [†]

サンプルコードは VisualC++2005 + DirectX SDK Update October 2005の環境下でビルドテストしています．なお，DXUT 版とMFC 版では，スケルトン構造クラスなど基本部分は同一ですが，MFC 版にはいくつかの操作インターフェイスが追加されています．本稿も MFC 版についてのみ記述していますので，DXUT 版はあくまでもオマケだとご理解ください．

DXUT 版(VC++2005, 288kb)
MFC 版(VC++2005, 72kb)

DXUT 版では，マウスの右ボタンドラッグで視点移動，マウスホイールでズームイン/アウトします．また，"Button1"ボタンを押すと，両肩が下方向に曲がります．

MFC 版では，左ボタンドラッグで視点回転，右ボタンドラッグで視点平行移動，中央ボタンドラッグとホイールでズームイン/アウトします．また，ダイアログバーで適当な関節を選択し，SHIFTキー+いずれかのマウスボタンを押しながらドラッグで，関節を回転します．なお，操作中の関節には赤球が表示されます．やっつけ仕事のインターフェイスですが，何卒ご容赦ください....

木構造: CTreeNodeクラス

順運動学のページでも述べたように，人体のスケルトン構造は木構造（ツリー構造）を基本とした，単純な階層構造によって表されます．本実装でも，シンプルな木構造（実際には木ノード）を扱うためのクラスを定義し，その拡張によってキャラクタの関節クラスを作成します．作成した CTreeNodeクラスでは，子ノードの追加，削除，親ノードの参照といった最小限の機能だけを定義しました．クラス宣言部のヘッダファイルを以下に示します．なお，クラス実装部分の説明は冗長でしょうから，ここでは割愛します．

CTreeNodeクラスヘッダファイル

```
 1  class CTreeNode
 2  {
 3  private:
 4    CTreeNode *m_pParent;                          // 親ノードへのポインタ
 5    std::vector m_vecChildren;          // 子ノードへのポインタ列
 6
 7  public:
 8    CTreeNode(void);                               // デフォルトコンストラクタ
 9    virtual ~CTreeNode();                          // デストラクタ
10
11  private:
12    virtual void Destroy(void);                    // 子ノードも含むノードの破棄
13
14  public:
15    size_t GetChildCount(void) const               // 子ノード数取得
16      { return m_vecChildren.size(); }
17
18  public:
19    void SetParent(const CTreeNode *parent)        // 親ノード設定
20      { m_pParent = const_cast(parent); }
21    CTreeNode* GetParent(void)                     // 親ノード取得
22      { return m_pParent; }
23    const CTreeNode* GetParent(void) const         // 親ノード取得
24      { return m_pParent; }
25    CTreeNode* GetRoot(void);                      // ルートノード取得
26    CTreeNode* GetChild(size_t nid) const;         // 子ノード取得
27
28  public:
29    size_t FindChild(const CTreeNode *node) const;// 子ノード検索
30    void AddChild(CTreeNode *child);               // 子ノード追加
31    bool RemoveNode(size_t nid);                   // ノード削除
32    bool RemoveNode(const CTreeNode *node)         // ノード削除
33      { return RemoveNode(FindChild(node)); }
34  };
```

関節ノード: CJointクラス

木ノードクラスを継承し，スケルトンの関節ノードクラスを作成します．関節ノードは次の4つの情報を格納します．

- 関節オフセット
  - 親関節ノードからみた，関節ノードの3次元位置を表す位置ベクトル．
  - 通常，構築時に設定した値に固定され，時間経過に応じて変化しない．
  - ルートノードに対しては，ワールド座標系からの初期オフセット成分を与える．
- 関節回転量
  - 関節の曲げ角＝親ボーンからみた子ボーンの回転量．
  - 時間経過に応じて変化．

    - ルートノードの場合は全身の方向を与える.
- 関節位置
    - ワールド座標におけるルートノードの3次元位置を示す.
    - 通常，ルートノードに対してのみ設定する.
    - 時間経過に応じて変化.
- 関節初期トランスフォーム
    - スケルトン描画用.
    - 単位立方体を関節オフセットベクトル，すなわちボーンの大きさに一致させるためのトランスフォーム成分（Fig.2 (b)）
    - ボーンの初期方向を(0, 0, +1.0)とみなす場合の値.

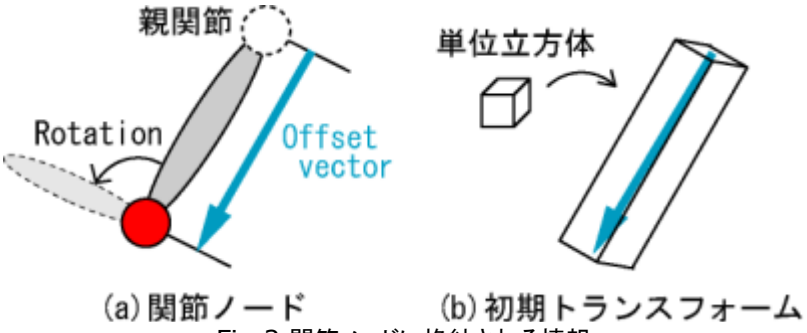

Fig.2 関節ノードに格納される情報

作成したCJointクラスは，これらの情報へのアクセスインターフェイスを中心に，いくつかのメンバにより構成されます．クラス宣言部のヘッダファイルを以下に示します．

CJointクラスヘッダファイル

```
 1  class CJoint : public CTreeNode
 2  {
 3  private:
 4    D3DXVECTOR3 m_vOffset;                    // ノードオフセット
 5    D3DXQUATERNION m_qRotation;               // 関節回転量
 6    D3DXVECTOR3 m_vPosition;                  // 関節位置
 7    D3DXMATRIX m_mInit;                       // ボーンの初期トランスフォーム
 8
 9  public:
10    CJoint(const D3DXVECTOR3 &offset, const CJoint *parent); // 初期化子付コンストラクタ
11    virtual ~CJoint();                        // デストラクタ
12
13  public:
14    void Initialize(const D3DXVECTOR3 offset, const CJoint *parent); // 初期化関数
15
16  public:
17    D3DXMATRIX GetInitialTransform() const;   // ボーンの初期トランスフォームの取得
18    void SetOffset(const D3DXVECTOR3 &offset);// 関節オフセットベクトルの設定
19    D3DXVECTOR3 GetOffset() const             // 関節オフセットベクトルの取得
20      { return m_vOffset; }
21    D3DXMATRIX GetOffsetMatrix() const;       // 関節オフセット行列の取得
22    void SetRotation(const D3DXQUATERNION &q);// 回転クォータニオンの設定
23    D3DXQUATERNION GetRotation() const        // 回転クォータニオンの取得
24      { return m_qRotation; }
25    D3DXMATRIX GetRotationMatrix() const;     // 回転行列の取得
26    void SetPosition(const D3DXVECTOR3 &pos); // 位置ベクトルの設定
27    D3DXVECTOR3 GetPosition() const           // 位置ベクトルの取得
28      { return m_vPosition; }
29    D3DXMATRIX GetPositionMatrix() const;     // 位置行列の取得
30    D3DXMATRIX GetTransformMatrix() const;    // トランスフォーム行列の取得
31
32  public:
33    void Destroy();                           // ノードの破棄
34    CJoint* GetParent()                       // 親関節ノードの取得
35      { return static_cast(CTreeNode::GetParent()); }
36    void SetParent(const CJoint *node);       // 親ノードの設定
```

```
37 |    CJoint* GetChild(size_t jid)              // 子関節ノードの取得
38 |      { return dynamic_cast(CTreeNode::GetChild(jid)); }
39 |    void AddChild(CJoint *node);              // 子ノードの追加
40 |    CJoint* GetRoot()                         // ルートノードの取得
41 |      { return dynamic_cast(CTreeNode::GetRoot()); }
42 ! };
```

CJointクラス実装部 †

単純なデータアクセスインターフェイス関数を除いた，いくつかのメンバ関数について簡単な説明を加えます．

まず，SetOffsetメンバ関数では，与えられた関節オフセットベクトルから，初期関節トランスフォームを計算します．前述の通り，全てのボーンの初期状態は+Z方向の単位ベクトルであると定義していますので，この単位ベクトルを関節オフセットベクトルに一致させるためのトランスフォームを計算します．

そのため，まず+Z方向の単位ベクトルと関節オフセットベクトルについて，(1)2ベクトル間の内積から回転量，(2)外積から回転主軸を計算しておきます．次に，(3)オフセットベクトル長とボーンの太さをスケーリング成分とする行列を作成し，最後にそれらの乗算によってトランスフォーム行列を求めます．

```
 1   void CJoint::SetOffset(const D3DXVECTOR3 &offset)
 2 ┼ {
 3 |   m_vOffset = offset;
 4 |   float offset_len = D3DXVec3Length(&offset);
 5 |
 6 ┼   if (offset_len <= 0) {
 7 |     D3DXMatrixIdentity(&m_mInit);
 8 ┼   } else {
 9 |     D3DXVECTOR3 vz(0, 0, 1), vaxis(0, 0, 0);
10 ┼     // (1) 回転量
11 !     float angle = acosf(offset.z / offset_len);
12 ┼     // (2) 回転主軸
13 !     D3DXVec3Cross(&vaxis, &vz, &offset);
14 ┼     // (3) ボーン長，太さ
15 !     D3DXMatrixScaling(&m_mInit, 0.2f * offset_len, 0.2f * offset_len, offset_len);
16 |     m_mInit._43 += offset_len / 2.0f;
17 |
18 |     D3DXMATRIX tmp;
19 |     D3DXMatrixRotationAxis(&tmp, &vaxis, angle);
20 |     m_mInit *= tmp;
21 !   }
22 ! }
```

GetTransformMatrixメンバ関数は，関節オフセット，回転量，位置情報の全てを含むトランスフォーム行列を計算します．順運動学や逆運動学で述べた計算手順のように，リンクの親子関係にしたがって (子ボーンの回転量)×(関節オフセット)×(関節の位置)で求められます．なお，オフセット行列と位置行列は，どちらも平行移動成分しか含まないので乗算の順番は逆でも問題ありません．

```
 1   D3DXMATRIX CJoint::GetTransformMatrix() const
 2 ┼ {
 3 |     return GetRotationMatrix() * GetOffsetMatrix() * GetPositionMatrix();
 4 ! }
```

スケルトン構造 †

キャラクタのスケルトンは，複数の関節ノードを組み合わせることで構築されます．本実装では，あらかじめ関節ノードの階層構造を定義し，そのルートノードを引数として渡すことでスケルトンを構築するクラスを作成しました．CFigureクラス宣言部のヘッダファイルを以下に示します．

CFigureクラスヘッダファイル †

```
class CFigure
{
private:
  CJoint *m_pRoot;                                  // ルートノード
  std::vector m_vecJoints;                  // 関節ノードリスト

public:
  CFigure(CJoint *root);                            // 初期化子付コンストラクタ
  virtual ~CFigure(void);                           // デストラクタ

private:
  void ListingJoint(CJoint *root);                  // スケルトン木構造からのノードリストの作成

public:
  size_t GetJointCount() const;                     // 関節総数の取得

public:
  void SetRootPosition(const D3DXVECTOR3 &pos);     // ルート位置の設定
  D3DXVECTOR3 GetRootPosition() const;              // ルート位置の取得
  void SetRootOrientation(const D3DXQUATERNION &rot); // ルート方向の設定
  D3DXQUATERNION GetRootOrientation() const;        // ルート方向の取得
  void SetJointRotation(size_t jid, const D3DXQUATERNION &rot);    // 関節回転量の設定
  D3DXQUATERNION GetJointRotation(size_t jid) const;  // 関節回転量の取得
  CJoint* GetRoot();                                // ルートノードへのポインタの取得

public:
  D3DXVECTOR3 GetWorldPosition(size_t jid) const;     // 関節ノードのワールド座標ベクトルの取得
};
```

CFigureクラス実装部 [†]

CFigureクラスについて，単純なデータアクセスインターフェイス以外のメンバ関数について説明を加えます．

まず，SetRootOrientation，GetRootOrientationメンバ関数の関数名
が，SetRoot"Rotation"，GetRoot"Rotation"でない理由は，3D空間における回転の表現形式をご参照ください（といいつつ，GetJointRotation(0)でルート方向が取得できる事実は無視してください...）

ListingJointメンバ関数では，木構造として与えられた関節ノードへのポインタを，全て一次元配列(m_vecJoints)に格納します．これは，ある特定の関節ノードを操作するときに，毎回木構造のルートから子ノードを探索するのは冗長なので，ランダムアクセス可能な形式でノードへのポインタを格納し，計算量の改善を図るためです．

```
void CFigure::ListingJoint(CJoint *root)
{
  m_vecJoints.push_back(root);
  for (size_t i = 0; i < root->GetChildCount(); ++i)
    ListingJoint(root->GetChild(i));
}
```

GetWorldPositionメンバ関数は，最もシンプルなフォワードキネマティクスの関数です．ワールド座標系における任意の関節位置を計算します．

```
D3DXVECTOR3 CFigure::GetWorldPosition(size_t jid) const
{
  D3DXVECTOR3 vTmp(0, 0, 0), vWorldPos(0, 0, 0);
  D3DXMATRIX mTrans;

  mTrans = m_vecJoints[jid]->GetOffsetMatrix();
  D3DXVec3TransformCoord(&vWorldPos, &vTmp, &mTrans);

  for (CJoint *pNode = m_vecJoints[jid]->GetParent();pNode != NULL;pNode = pNode->GetParent())
  {
    mTrans = pNode->GetTransformMatrix();
    vTmp = vWorldPos;
    D3DXVec3TransformCoord(&vWorldPos, &vTmp, &mTrans);
  }
  return vWorldPos;
```

```
16 ! }
```

指定した関節の回転量はその関節の位置に影響を及ぼさないので，オフセット成分のみを乗算します．あとは，ルートノードまで順次，親関節のトランスフォーム行列を乗算することで所望の座標を得ます．この計算の基礎については，逆運動学のページも合わせてご参照ください．

利用例

作成したクラス群を利用して下図に示すスケルトン構造を構築し，DirectX Graphicsにより描画します．本サンプルは，HLSL - High Level Shader Languageで示したビュークラスを拡張して作成しています．本稿では，主にそれらの拡張部分について説明します．

本稿で作成したスケルトンの階層構造を以下に示します．

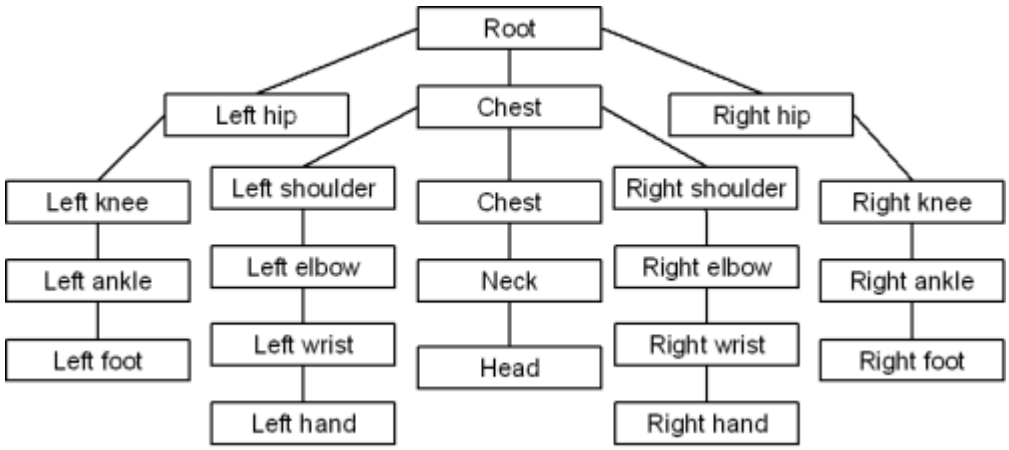

Fig.3 スケルトンの階層構造

スケルトンの構築

上記の階層構造に対応したスケルトンを構築するためのコードは，次のようになります．関節オフセットの設定→関節ノードオブジェクトの生成→親関節との対応付けという一連の処理を，全ての関節について行っています．

```
 1  void CSceneView::CreateSkeleton()
 2  {
 3    D3DXVECTOR3 v;
 4
 5    v.x = 0; v.y = 0; v.z = 0;
 6    CJoint *root = new CJoint(v, NULL);
 7    v.x = 0; v.y = 20.0f; v.z = 0.0f;
 8    CJoint *waist = new CJoint(v, root);    root->AddChild(waist);
 9    v.x = 0; v.y = 20.0f; v.z = 0.0f;
10    CJoint *neck = new CJoint(v, waist);    waist->AddChild(neck);
11    v.x = 0; v.y = 20.0f; v.z = 0.0f;
12    CJoint *head = new CJoint(v, neck);     neck->AddChild(head);
13
14    v.x = 0; v.y = 20.0f; v.z = 20.0f;
15    CJoint *lshold = new CJoint(v, waist);  waist->AddChild(lshold);
16    v.x = 0; v.y = 0.0f; v.z = 20.0f;
17    CJoint *lelbow = new CJoint(v, lshold); lshold->AddChild(lelbow);
18    v.x = 0; v.y = 0.0f; v.z = 20.0f;
19    CJoint *lwrist = new CJoint(v, lelbow); lelbow->AddChild(lwrist);
20    v.x = 0; v.y = 0.0f; v.z = 10.0f;
21    CJoint *lhand = new CJoint(v, lwrist);  lwrist->AddChild(lhand);
22
23    v.x = 0; v.y = 20.0f; v.z = -20.0f;
24    CJoint *rshold = new CJoint(v, waist);  waist->AddChild(rshold);
25    v.x = 0; v.y = 0.0f; v.z = -20.0f;
26    CJoint *relbow = new CJoint(v, rshold); rshold->AddChild(relbow);
27    v.x = 0; v.y = 0.0f; v.z = -20.0f;
28    CJoint *rwrist = new CJoint(v, relbow); relbow->AddChild(rwrist);
29    v.x = 0; v.y = 0.0f; v.z = -10.0f;
```

```
30 |    CJoint *rhand = new CJoint(v, rwrist);  rwrist->AddChild(rhand);
31 |
32 |    v.x = 0; v.y = -10.0f; v.z = 10.0f;
33 |    CJoint *lhip = new CJoint(v, root);     root->AddChild(lhip);
34 |    v.x = 0; v.y = -30.0f; v.z = 0.0f;
35 |    CJoint *lknee = new CJoint(v, lhip);    lhip->AddChild(lknee);
36 |    v.x = 0; v.y = -30.0f; v.z = 0.0f;
37 |    CJoint *lankle = new CJoint(v, lknee);  lknee->AddChild(lankle);
38 |    v.x = 20.0f; v.y = 0.0f; v.z = 0.0f;
39 |    CJoint *lfoot = new CJoint(v, lankle);  lankle->AddChild(lfoot);
40 |
41 |    v.x = 0; v.y = -10.0f; v.z = -10.0f;
42 |    CJoint *rhip = new CJoint(v, root);     root->AddChild(rhip);
43 |    v.x = 0; v.y = -30.0f; v.z = 0.0f;
44 |    CJoint *rknee = new CJoint(v, rhip);    rhip->AddChild(rknee);
45 |    v.x = 0; v.y = -30.0f; v.z = 0.0f;
46 |    CJoint *rankle = new CJoint(v, rknee);  rknee->AddChild(rankle);
47 |    v.x = 20.0f; v.y = 0.0f; v.z = 0.0f;
48 |    CJoint *rfoot = new CJoint(v, rankle);  rankle->AddChild(rfoot);
49 |
50 |    m_pFigure = new CFigure(root);
51 ! }
```

スケルトンの描画 [†]

スケルトンを描画するために，以下の2つの関数を作成しました．まず，DrawSkeleton関数では，(1)行列スタックを作成し，(2)スケルトン全体のスケーリング成分を設定します．そして，(3)ルートノードのトランスフォーム成分を行列スタックにセットし，(4)子ノードの描画関数を呼び出します．そして，全ての子ボーンを描画した後，最後に(5)行列スタックオブジェクトを解放します．

```
 1   void CSceneView::DrawSkeleton()
 2 { {
 3 |   HRESULT hr;
 4 |   LPD3DXMATRIXSTACK pMatStack;
 5 |   CJoint *pRoot;
 6 |
 7 |   m_pEffect->SetTechnique("PhongShader");
 8 |   m_pEffect->SetVector("g_vAmbiColor", &D3DXVECTOR4(0.6f, 0.6f, 0.6f, 1.0f));
 9 |   m_pEffect->SetVector("g_vSurfColor", &D3DXVECTOR4(0.6f, 0.6f, 0.6f, 1.0f));
10 |
11 |   // (1) 行列スタックの作成，初期化
12 !   if (hr = D3DXCreateMatrixStack(0, &pMatStack), FAILED(hr))
13 |     return;
14 |   pMatStack->LoadIdentity();
15 |   // (2) スケルトン全体のスケーリング成分の設定
16 !   pMatStack->Scale(0.02f, 0.02f, 0.02f);
17 |
18 |   pRoot = m_pFigure->GetRoot();
19 |   pMatStack->Push();
20 |     // (3) ルートノードのトランスフォームを設定
21 !     pMatStack->MultMatrixLocal(&pRoot->GetTransformMatrix());
22 |     // (4) 子ノードの描画
23 !     for (size_t j = 0;j < pRoot->GetChildCount();j++)
24 |       DrawSkeletonSub(pRoot->GetChild(j), pMatStack);
25 |   pMatStack->Pop();
26 |
27 |   // (5) 行列スタックの解放
28 !   pMatStack->Release();
29 ! }
```

DrawSkeletonSub関数は，スケルトンの木構造に従って再帰的に呼び出される関数で，ボーンを実際に描画し，子ノードを描画するための再帰呼び出しを行います．まず，(1)スケルトン構造の末端部位でないかチェックし，末端部位でないならば(2)初期トランスフォーム成分を行列スタックにセットしてボーンを描画します．なお，この初期トランスフォーム成分は子ノードには影響しませんので，Push()～Pop()関数によって囲まれています．そして，(3)自身の関節トランスフォーム成分を行列スタックにセットし，子ノードの描画関数を再帰的に呼び出します．

```
void CSceneView::DrawSkeletonSub(CJoint *pJoint, LPD3DXMATRIXSTACK &pMatStack)
{
  // (1) スケルトン末端かチェック
  if (pJoint == NULL)
    return;

  pMatStack->Push();
  (2) 初期トランスフォームの乗算
  pMatStack->MultMatrixLocal(&pJoint->GetInitialTransform());
    DrawMeshSub(m_pMeshSphere, *pMatStack->GetTop());
  pMatStack->Pop();

  pMatStack->Push();
    // (3) 子ノードの描画を再帰的に呼び出し
    pMatStack->MultMatrixLocal(&pJoint->GetTransformMatrix());
    for (size_t j = 0;j < pJoint->GetChildCount();j++)
      DrawSkeletonSub(pJoint->GetChild(j), pMatStack);
  pMatStack->Pop();
}
```

すこしややこしいかもしれませんので，多関節リンク系の描画例をも合わせて参照していただきたい思います．

フォワードキネマティクスによる関節ノード座標の計算

FK計算が正常に動作することを確認するため，CFigureクラスのGetWorldPositionメンバ関数を利用し，指定された関節ノード位置に赤球を描画しています（MFC版のみ）．(1)スケルトン全体のスケーリングと，(2)球の大きさを指定するためのスケーリングを設定するため，コードが若干長くなっています．

```
void CSceneView::DrawJointMarker()
{
  D3DXVECTOR3 pos = m_pFigure->GetWorldPosition(m_nActiveJoint);
  D3DXMATRIX world, tmp;

  // (1)
  D3DXMatrixScaling(&world, 0.02f, 0.02f, 0.02f);
  D3DXMatrixTranslation(&tmp, pos.x, pos.y, pos.z);
  world = tmp * world;
  // (2)
  D3DXMatrixScaling(&tmp, 5.0f, 5.0f, 5.0f);
  world = tmp * world;

  m_pEffect->SetTechnique("PhongShader");
  m_pEffect->SetVector("g_vAmbiColor", &D3DXVECTOR4(1.0f, 0.2f, 0.2f, 1.0f));
  m_pEffect->SetVector("g_vSurfColor", &D3DXVECTOR4(1.0f, 0.2f, 0.2f, 1.0f));

  DrawMeshSub(m_pMeshSphere, world);
}
```

まとめ

人体キャラクタアニメーションの基礎となるスケルトン構造について，木構造をベースとした実装法の一例について紹介しました．本実装では，特にモーションキャプチャデータの利用を前提とした実装法を作成しています．もちろん，物理シミュレーションを利用する場合にはボーンの慣性モーメント情報などを追加しなければなりませんが，木構造がベースとなる点は全てのモデルに共通すると思います．

木構造を基本としたスケルトン構造では，サンプルコードに示すように(1)FK計算のコードの簡易化，(2)再帰関数を利用したボーン描画など，コーディングが比較的シンプルにまとまりやすい利点があります．もちろん，スキンメッシュを利用する際にもこれらは大きな利点となります．

順運動学と木構造はキャラクタアニメーションの基礎となりますので，サンプルコードを改変しつつ理解していただければと思います．

Last-modified: 2008-01-30 (水) 19:10:31 (2280d)

Site admin: cherub