7自由度リンク構造のための解析的IK

http://www.tmps.org/index.php?7%BC%AB%CD%B3%C5%D9%A5%EA%A5%F3%A5%AF%B9%BD%C2%A4%A4%CE%A4%BF%A4%E1%A4%CE%B2%F2%C0%CF%C5%AAIK

多くのインバースキネマティクスアルゴリズムは，スケルトンの構造に依存しないような計算法が提案されています．しかし，モデルの自由度が増えるにつれて姿勢の制御が複雑になりがちです． そのためインバースキネマティクスは，キャラクタの腕や脚といった単一の部位に対してのみ利用されることが多いと思います． 今回は，人体キャラクタの腕や脚に特化した解析的IK手法を解説します． モデルの構造は限定されますが，比較的簡単に関節可動域も設定でき，また解析計算なので非常に高速に計算できる方法です．

参考文献 †

今回の手法は，キャラクタアニメーション研究者にはご存知，Normal I. Badler先生が主催する Pennsylvania大学the Center for Human Modeling & Simulationの研究成果です（PublicationsでPDFが公開されています）．

```
D. Tolani, A. Goswami, and N. Badler,
"Real-time inverse kinematics techniques for anthropomorphic limbs",
Graphical Models 62(5), pp. 353-388, 2000.
```

同様なアプローチはSIGGRAPH1999でのJehee Lee先生の論文でも報告されています． この原稿を書きながら発見したのですが，Lee先生は昨年よりSeoul Natinal UniversityでMovement Research Labを立ち上げられています． この研究室の今後の動向にも注目していきたいと思います．（こちらもPublicationsでPDFとムービーが公開されています）

```
Jehee Lee and Sung Yong Shin,
"A Hierarchical Approach to Interactive Motion Editing for Human-like Characters",
SIGGRAPH 99, 39-48, 1999.
```

解析的IKの実装コードは，HMS内のIKANプロジェクトページでも提供されています． これはJACKのPlug-Inであり，商用利用ではライセンス使用料が発生するようですが， 研究用途などの非商用用途ではフリーで配布されています． オリジナルの実装に興味がある方は同ページから申し込みされるとよいでしょう．（ちなみに私は所有していません．オリジナルの実装に興味はあるのですが，面倒なので...）

7自由度モデル †

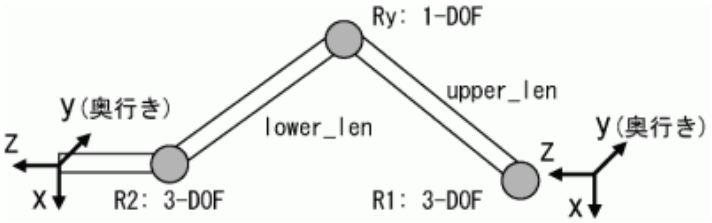

Fig.1 7自由度モデル

LimbIKの対象となる7自由度リンク系をFig.1に示します． モデルは3つのリンクを3つの関節（図中の灰色の丸）で連結することで構築されます． R1関節とR2関節が3自由度，Ry関節はY軸周りにのみ回転する1自由度関節で，リンク系の始点はR1関節に対応します． さらに，Ry関節はY軸周りの正方向にしか回転できない（0～180º）という拘束条件が設定されています． こうした構造は，一般的な人体キャラクタの腕や脚のモデルに近いことがわかると思います．

ここで，LimbIKがターゲットとする逆運動学問題を明らかにしておきます．

1. R2関節の位置のみを指定し，R1関節とRy関節の回転量を計算．R2関節は扱わない．
2. R2関節の位置と先端リンクの方向を指定し，R1関節，Ry関節，R2関節の全ての回転量を計算

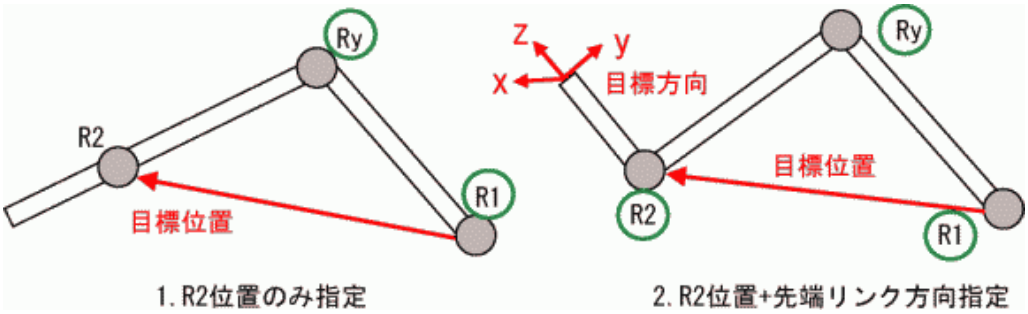

Fig.2 LimbIKの問題設定

腕の例でいうと，1.手首の位置（3自由度）のみを指定して肩とひじの回転量（計4自由度）を計算し，手首の曲げ方はユーザが決定する，というケースと，2.手首の位置と手のひらの方向，指先の方向（計6自由度）を指定し，肩，ひじ，手首の回転量（計7自由度）を求めるケースの2通りを扱います．ここで，いずれのケースにおいても1自由度の冗長性が残ることがわかります．LimbIKでは，この1自由度が「R1関節とR2関節の目標位置を結んだ軸周りの，Ry関節の回転量」となるように定式化し，その回転量はユーザが指定するものとしています．この回転は「swivel角」と呼ばれ，例えば，手首と肩の位置を固定した状態でひじを動かす際のパラメータとなります．

次に，リンク系の座標系の設定について解説します．まず，Z軸はリンクが伸びていく方向に設定されます．Y軸は，その軸周りの正回転がRy関節の回転方向に一致するように決定され，X軸はZ軸とY軸方向の外積から求められます．人体の腕を例に説明すると，右腕を水平に伸ばした状態で，腕が伸びていく方向がZ軸，鉛直下向き方向がY軸，身体前面に向かう方向がX軸になります．人体キャラクタにLimbIKを適用する場合の，各部位における座標系の設定例をFig.2に示します．

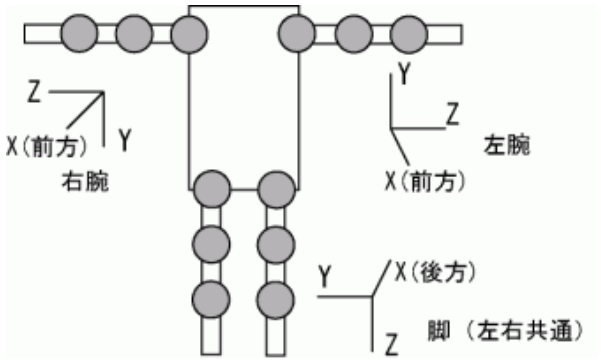

Fig.3 人体モデルにおける基準座標系設定例

LimbIK Solver [†]

詳細は参考文献に記述されていますので，ここではアルゴリズムのポイントをソースコードを通して説明していきます．まず，実装したLimbIK Solverクラスの宣言部を示します．

```
 1  class CLIKSolver
 2  {
 3  public:
 4    //! デフォルトコンストラクタ
 5    CLIKSolver(void);
 6    //! 初期化子付コンストラクタ
 7    CLIKSolver(float fUpperBone, float fLowerBone);
 8    //! デストラクタ
 9    ~CLIKSolver(void);
10
11    //! リンク長x2の設定
12    void InitStructure(float fUpperBone, float fLowerBone);
13
14    //! 目標R2位置を満たす解のチェック
15    bool IsSolution(const D3DXVECTOR3 &vGoal);
16    //! 目標R2座標系を満たす解のチェック
17    bool IsSolution(const D3DXMATRIX &mGoal);
18
19    //! 目標R2位置からの姿勢計算
20    bool Solve(D3DXMATRIX &mR1, D3DXMATRIX &mRy, const D3DXVECTOR3 &vGoal, float fSwivel);
21    //! 目標R2座標系からの姿勢計算
22    bool Solve(D3DXMATRIX &mR1, D3DXMATRIX &mRy, D3DXMATRIX &mR2, const D3DXMATRIX &mGoal, float fSwivel);
23
24    //! 目標R2位置とRy関節の位置からSwivel角の計算
25    bool GetSwivel(float &fSwivel, const D3DXVECTOR3 &vGoal, const D3DXVECTOR3 &vRyPos);
26    //! 目標R2座標系とRy関節の位置からSwivel角の計算
27    bool GetSwivel(float &fSwivel, const D3DXMATRIX &mGoal, const D3DXVECTOR3 &vRyPos);
28
29  private:
30    //! 第1リンク長
```

```
31 !    float m_fUpperBone;
32 |    //! 第2リンク長
33 !    float m_fLowerBone;
34 ! };
```

メンバ関数は大別して・解の存在のチェック，・解の計算，・与えられた姿勢におけるswivel角の計算，の3つです．コメントで「目標R2座標系」と記述しているのは，リンク系始点座標系からみた，R2関節の目標位置と目標方向（すなわち座標系）を表す行列を指します．以下に，個別の実装を示していきます．

まず，コンストラクタ，デストラクタ，リンク構造の初期化関数を示します．ここではメンバ変数を初期化するだけです．

```
 1    CLIKSolver::CLIKSolver(void)
 2 |  {
 3 |    m_fUpperBone = 0.0f;
 4 |    m_fLowerBone = 0.0f;
 5 !  }
 6    CLIKSolver::CLIKSolver(float fUpperBone, float fLowerBone)
 7 |  {
 8 |    m_fUpperBone = fUpperBone;
 9 |    m_fLowerBone = fLowerBone;
10 !  }
11    CLIKSolver::~CLIKSolver()
12 |  {
13 !  }
14    void CLIKSolver::InitStructure(float fUpperBone, float fLowerBone)
15 |  {
16 |    m_fUpperBone = fUpperBone;
17 |    m_fLowerBone = fLowerBone;
18 !  }
```

次に，逆運動学解の存在をチェックする関数を示します．ここでは，R1関節の位置からR2関節の目標位置への距離が，2つのリンク長の合計の範囲内に収まっているかチェックします．また，安定に計算するために，目標位置がR1関節の位置にあまりに近いときにも計算不可能としています．

```
 1    bool CLIKSolver::IsSolution(const D3DXVECTOR3 &vGoal)
 2 |  {
 3 |    // R1関節とR2関節の目標位置との距離によって，解の存在を判定
 4 !    float fDistGoal = D3DXVec3Length(&vGoal);
 5 |    if (fDistGoal >= m_fUpperBone + m_fLowerBone || fDistGoal <= 1.0e-5f)
 6 |      return false;
 7 |    return true;
 8 !  }
 9
10    bool CLIKSolver::IsSolution(const D3DXMATRIX &mGoal)
11 |  {
12 |    // 座標系を表す行列から，位置成分のみを取り出す
13 !    D3DXVECTOR3 vGoal(mGoal._41, mGoal._42, mGoal._43);
14 |    return IsSolution(vGoal);
15 !  }
```

次に，R2関節の目標位置のみを指定する場合のLimbIKの解析計算部分を示します．Fig.4に示すように，1.Ry関節の回転量計算→2.swivel角を変化させたときにRy関節が描く軌跡（円）の計算→ 3.swivel角からRy関節の位置の計算→4.R1関節の回転量の算出，の手順に沿って，解析幾何の計算が実行されます．なお，この実装での変数名は参考文献に沿っていますので，原著を読む際の参考になればと思います．

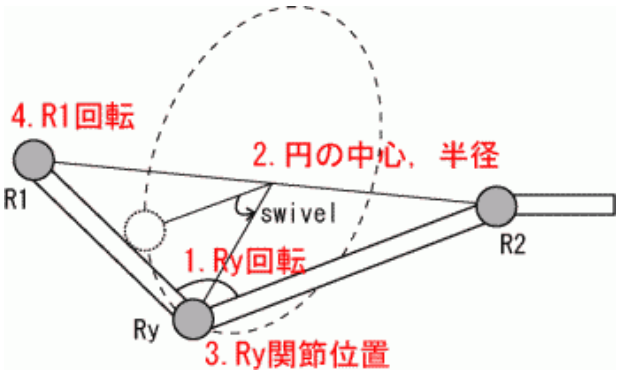

Fig.4 LimbIKアルゴリズム

```
 1    bool CLIKSolver::Solve(D3DXMATRIX &mR1, D3DXMATRIX &mRy, const D3DXVECTOR3 &vGoal, float fSwivel)
 2 |  {
 3 |    //
 4 |    // vN: R2関節からR2関節の目標位置に向かう単位ベクトル
```

```
    // vU: 目標位置がR1関節に近づくときに，Ry関節が移動する方向(基準座標系)
    // vV: nV と vUの外積
    // vC: swivelを変化させたときにRy関節が描く円の中心座標
    // vE: Ry関節の位置
    // vT: 作業用
    //
    D3DXVECTOR3 vN, vU, vV, vC, vE, vT;
    // R1関節の回転成分を表す行ベクトルx3
    D3DXVECTOR3 vR1X, vR1Y, vR1Z;
    // R1関節の位置とR2関節の目標位置との距離
    float fDistGoal;
    // Ry関節の回転量
    float fRy;
    //
    //  fCosAlpha: vNとvEの間の角度
    //  fCircleRadius : Ry関節軌跡円の半径
    //
    float fCosAlpha, fCircleRadius;
    // fDdistGoalが短くなる時にRy関節を移動させる方向（基準座標系）
    D3DXVECTOR3 vA(-1.0f, 0.0f, 0.0f);

    // 目標までの距離
    fDistGoal = D3DXVec3Length(&vGoal);

    // 幾何学的に到達不可能な場合は解析失敗
    if (fDistGoal >= m_fUpperBone + m_fLowerBone)
      return false;
    // 目標があまりにも近い場合は計算が発散するので，とりあえず解析失敗としている
    if (fDistGoal <= 1.0e-5f)
      return false;

    // リンク系始点から目標へ向かう単位ベクトル
    D3DXVec3Normalize(&vN, &vGoal);

    // 余弦定理より，Ry関節の回転量を算出
    fRy = acosf((fDistGoal * fDistGoal - m_fUpperBone * m_fUpperBone - m_fLowerBone * m_fLowerBone)
      / (2.0f * m_fUpperBone * m_fLowerBone));
    if (fRy > 0)
      fRy += D3DX_PI;
    D3DXMatrixRotationY(&mRy, fRy);

    // 同じく余弦定理より，(R1→Ry関節)ベクトルと(R1→R2関節)ベクトルのなす角を計算
    fCosAlpha = (fDistGoal * fDistGoal + m_fUpperBone * m_fUpperBone - m_fLowerBone * m_fLowerBone)
      / (2.0f * fDistGoal * m_fUpperBone);
    // Ry関節軌跡円の中心と半径を計算
    vC = fCosAlpha * m_fUpperBone * vN;
    fCircleRadius = sqrtf(1.0f - fCosAlpha * fCosAlpha) * m_fUpperBone;

    // swivel角の入力はDegreeを想定
    fSwivel = D3DXToRadian(fSwivel);

    // swivel=0のときの，軌跡円中心からRy関節への方向ベクトル
    vU = vA - D3DXVec3Dot(&vA, &vN) * vN;
    D3DXVec3Normalize(&vU, &vU);
    D3DXVec3Cross(&vV, &vN, &vU);

    // Ry関節の位置
    vE = vC + fCircleRadius * (cosf(fSwivel) * vU + sinf(fSwivel) * vV);

    // R1関節の回転量の計算
    D3DXVec3Normalize(&vR1Z, &vE);
    vR1X = vGoal - D3DXVec3Dot(&vGoal, &vR1Z) * vR1Z;
    D3DXVec3Normalize(&vR1X, &vR1X);
    D3DXVec3Cross(&vR1Y, &vR1Z, &vR1X);

    D3DXMatrixIdentity(&mR1);
    mR1(0, 0) = vR1X.x;
    mR1(0, 1) = vR1X.y;
    mR1(0, 2) = vR1X.z;
    mR1(1, 0) = vR1Y.x;
    mR1(1, 1) = vR1Y.y;
    mR1(1, 2) = vR1Y.z;
    mR1(2, 0) = vR1Z.x;
    mR1(2, 1) = vR1Z.y;
    mR1(2, 2) = vR1Z.z;

    return true;
}
```

先端リンクの方向も指定する場合は，まずR2関節を目標位置に一致させ，その時の先端リンクの方向を順運動学により計算します． そして，目標方向からの回転量の差分を求め，R2関節の回転量とします． 実装はもう少し効率化されていますが，基本的な考え方は同じです．

```
bool CLIKSolver::Solve(D3DXMATRIX &mR1, D3DXMATRIX &mRy, D3DXMATRIX &mR2, const D3DXMATRIX &mGoal, float fSwivel)
{
  D3DXMATRIX mGoalRot = mGoal;
  mGoalRot._41 = 0;
  mGoalRot._42 = 0;
  mGoalRot._43 = 0;

  D3DXVECTOR3 vGoal(mGoal._41, mGoal._42, mGoal._43);
  if (!Solve(mR1, mRy, vGoal, fSwivel))
    return false;

  D3DXMATRIX mT;
  D3DXMatrixTranspose(&mT, &mR1);
  mR2 = mGoalRot * mT;
  D3DXMatrixTranspose(&mT, &mRy);
  mR2 *= mT;
  return true;
}
```

最後に，与えられたリンク系の姿勢からのswivel角の計算部分を示します．まず，逆運動学計算の際と同様に，Ry関節が描く軌跡円の中心位置，半径を計算します．次に，swivel角が0°のときのRy関節の位置を計算します．そして，実際のRy関節の位置に移動させるための回転量としてswivel角を計算します．

```
bool CLIKSolver::GetSwivel(float &fSwivel, const D3DXVECTOR3 &vGoal, const D3DXVECTOR3 &vRyPos)
{
  D3DXVECTOR3 vN, vU, vV, vC, vE, vT;
  float fDistGoal, fCosAlpha, fCircleRadius;
  D3DXVECTOR3 vA(-1.0f, 0.0f, 0.0f);

  fDistGoal = D3DXVec3Length(&vGoal);
  if (fDistGoal >= m_fUpperBone + m_fLowerBone)
    return false;

  D3DXVec3Normalize(&vN, &vGoal);
  vU = vA - D3DXVec3Dot(&vA, &vN) * vN;
  D3DXVec3Normalize(&vU, &vU);
  D3DXVec3Cross(&vV, &vN, &vU);

  fCosAlpha = (fDistGoal * fDistGoal + m_fUpperBone * m_fUpperBone - m_fLowerBone * m_fLowerBone)
    / (2.0f * fDistGoal * m_fUpperBone);

  vC = fCosAlpha * m_fUpperBone * vN;
  fCircleRadius = sqrtf(1.0f - fCosAlpha * fCosAlpha) * m_fUpperBone;
  //
  // ここまではIKと全く同じ処理

  D3DXVECTOR3 vP, vPs;
  vP = vRyPos - vC;
  vPs = vP - D3DXVec3Dot(&vP, &vN) * vN;

  D3DXVec3Cross(&vT, &vPs, &vU);
  fSwivel = atan2f(D3DXVec3Length(&vT), D3DXVec3Dot(&vPs, &vU));
  vE = vC + fCircleRadius * (cosf(fSwivel) * vU + sinf(fSwivel) * vV);

  vE -= vRyPos;

  if (D3DXVec3Length(&vE) < 1.0f)
    fSwivel = -fSwivel;
  fSwivel = D3DXToDegree(fSwivel);
  return true;
}

bool CLIKSolver::GetSwivel(float &fSwivel, const D3DXMATRIX &mGoal, const D3DXVECTOR3 &vRyPos)
{
  D3DXVECTOR3 vGoal(mGoal._41, mGoal._42, mGoal._43);
  return GetSwivel(fSwivel, vGoal, vRyPos);
}
```

DirectX Graphicsにおける実装例

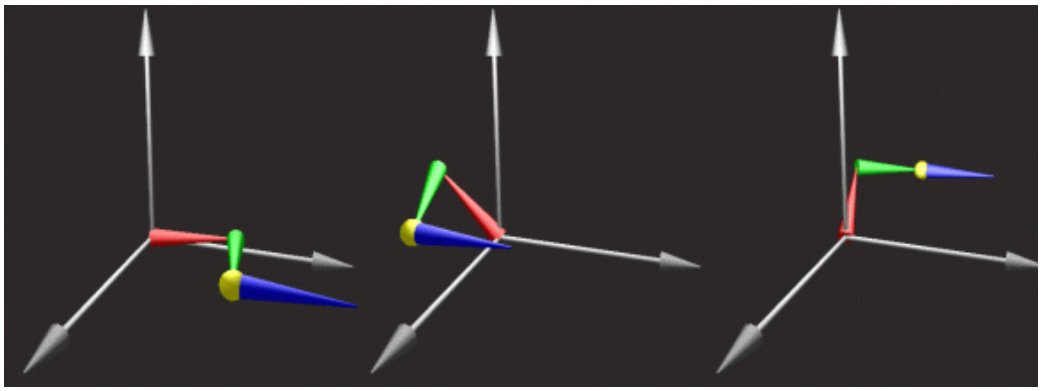

Fig.5実行画面スナップショット

今回のサンプルは，LimbIKにより7自由度リンク系を操作するプログラムです． Fig.5に示すように黄色の球にR2関節が追従して移動し，かつ先端リンクは一定方向に拘束されます．

VisualC++.NET2003プロジェクトファイル(34Kb)

目標位置とswivel角は，キーボードのテンキーで操作できます．

- [7][1]: 目標のX軸方向の移動
- [8][2]: 目標のY軸方向の移動
- [9][3]: 目標のZ軸方向の移動
- [6][4]: swivel角の操作

サンプルではリンク系を描画する関数内で，LimbIK Solverの初期化，目標位置の指定，リンク系の姿勢計算の全ての処理を含めています．かなり冗長な実装ですが，それでもインタラクティブなスピードを実現しています． なお，いつもながらエラーチェックは全く行っていないので，解を計算できない場合はリンク系そのものが消失してしまいます．

```
HRESULT CSceneView::DrawLinks(void)
{
    LPD3DXMESH pMesh;
    LPD3DXMATRIXSTACK pMatStack;
    D3DXMATRIX r1, ry, r2;

    // LimbIKソルバの初期化
    CLIKSolver iks(1.0f, 1.0f);
    // 目標位置，方向，swivel角を指定してR1関節，Ry関節，R2関節の回転量を算出
    D3DXMATRIX mTarget;
    D3DXMatrixTranslation(&mTarget, m_vTarget.x, m_vTarget.y, m_vTarget.z);
    iks.Solve(r1, ry, r2, mTarget, m_fSwivel);

    // エフェクト，メッシュ，行列スタックの初期化
    m_pEffect->SetTechnique(m_hTechPhong);
    m_pEffect->SetVector(m_hAmbiColor, &D3DXVECTOR4(1.0f, 1.0f, 1.0f, 1.0f));
    D3DXCreateCylinder(m_pDeviceD3D, 0.1f, 0.01f, 1.0f, 10, 10, &pMesh, NULL);
    D3DXCreateMatrixStack(0, &pMatStack);

    // R1関節
    pMatStack->LoadIdentity();
    pMatStack->MultMatrixLocal(&r1);
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    m_pEffect->SetVector(m_hSurfColor, &D3DXVECTOR4(0.5f, 0.0f, 0.0f, 1.0f));
    DrawMeshSub(pMesh, *pMatStack->GetTop());

    // Ry関節
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    pMatStack->MultMatrixLocal(&ry);
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    m_pEffect->SetVector(m_hSurfColor, &D3DXVECTOR4(0.0f, 0.5f, 0.0f, 1.0f));
    DrawMeshSub(pMesh, *pMatStack->GetTop());

    // R2関節
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    pMatStack->MultMatrixLocal(&r2);
    pMatStack->TranslateLocal(0.0f, 0.0f, 0.5f);
    m_pEffect->SetVector(m_hSurfColor, &D3DXVECTOR4(0.0f, 0.0f, 0.5f, 1.0f));
    DrawMeshSub(pMesh, *pMatStack->GetTop());

    pMesh->Release();

    D3DXCreateSphere(m_pDeviceD3D, 0.12f, 10, 10, &pMesh, NULL);
    pMatStack->LoadIdentity();
    pMatStack->Translate(m_vTarget.x, m_vTarget.y, m_vTarget.z);
```

```
46 |    m_pEffect->SetVector(m_hSurfColor, &D3DXVECTOR4(0.5f, 0.5f, 0.0f, 1.0f));
47 |    DrawMeshSub(pMesh, *pMatStack->GetTop());
48 |
49 |    pMesh->Release();
50 |    pMatStack->Release();
51 |
52 |    return S_OK;
53 ! }
```

まとめ

今回は，キャラクタの腕部や脚部の構造に特化した解析的IK手法を紹介しました． 原著は英語ですが，図と数式だけで理解できると思いますので，基本的な解析幾何の知識があれば原著にあたることをお勧めします． LimbIKは提案手法の一部分であり，他にも関節可動域を導入する方法や最適化計算によるIK手法など，非常に有用な手法が解説されています．

次回は，前々回説明不足の感があった順運動学を補完する意味も込めて，簡単なキャラクタモデルを構築してみます． 余裕があれば，実際にLimbIKを適用してみたいとも思います．
