物理計算によるキーフレーム補間

http://www.tmps.org/index.php?%CA%AA%CD%FD%B7%D7%BB%BB%A4%CB%A4%E8%A4%EB%A5%AD%A1%BC%A5%D5%A5%EC%A1%BC%A5%E0%CA%E4%B4%D6

キーフレーム法によるキャラクタアニメーション制作では，キーフレームで指定された関節角度を線形関数やスプライン関数などによって補間することで，キーフレーム間の動作を自動的に計算できます．しかし，自然なキャラクタアニメーションを制作するためには経験やセンスが求められます．例えば，実際のキャラクタは質量を持っていますので，質量が軽い部分は速く動かせても，重い部分を加速／減速するためには大きな力が加わっているはずです．そうしたキャラクタの"重さ"を感じさせためには，キーフレームや補間カーブを試行錯誤的に編集しなければなりません．

そのような物理的な特性を反映したアニメーションの制作には，物理シミュレーション（キャラクタアニメーションでは特に動力学シミュレーション）の利用が適しています．しかし従来のシミュレーション技術では，キーフレームアニメーションのように「指定された姿勢」を「指定された時間」に満たす動作を計算するために，高度な最適化計算を利用する必要がありました．この記事では，こうした問題を解決する一手法として，物理シミュレーションによるキーフレーム補間手法を解説します．この方法はCGキャラクタのような多関節リンク構造体のキーフレームアニメーション制作のために提案されたもので，四肢各部の慣性モーメントを考慮した物理シミュレーションによって補間計算を行います（ただし重力の影響は反映できません）．なお，今回のサンプルプログラムは，最近普及が目覚しい物理計算エンジンである Open Dynamics Engine（ODE）を利用して作成しています．

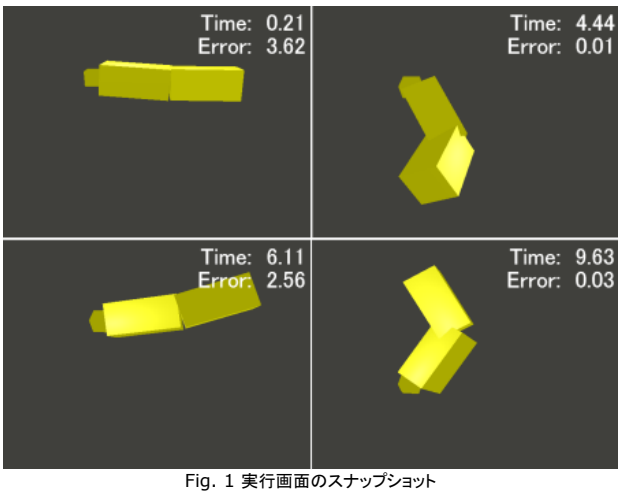

Fig. 1 実行画面のスナップショット

参考文献 [†]

1. Brian Allen, Derek Chu, Ari Shapiro and Petros Faloutsos, On the Beat! Timing and Tension for Dynamic Characters, ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2007, pp.239-247, 2007.
2. demura.net
   - ODEの導入から応用まで非常にわかりやすくまとめられています．同著者による簡単！実践！ロボットシミュレーション Open Dynamics Engineによるロボットプログラミングも ODE 初学者にとって素晴らしい良書です．

[↑]

サンプルプログラム [†]

サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update December 2006 + ODE 0.8 の環境下で MFC を用いて作成し，WinVista(64bit)上でのみ実行テストしています．ただし，バイナリデータは32bit版です．

VC++2005プロジェクト(45kb)
サンプルバイナリ(418kb)

[ファイル]→[シミュレーション]ボタンを押すと，適当に設定した 2 つのキーフレームを 時間間隔 5.0 ごとに交互に補間する，以下のようなアニメーションが表示されるはずです．

サンプルムービー(m1v形式，1.14mb)

[↑]

ODEの組み込み [†]

この記事では，すでに ODE が適切にインストールされていることを前提に解説を行います．ODE の基本的なセットアップや使用法については demura.net さんなどを参考にしてください．なお，ODE のヘッダファイルはインクルードパスが通ったディレクトリ下の"ode"フォルダにまとめられており，ライブラリ名は"ode.lib"であると仮定しています．

[↑]

シミュレーション名前空間 [†]

基本的な ODE のグローバル変数や初期化関数は NPhysics 名前空間としてまとめています．グローバル変数は上からシミュレーション環境のインデクス，衝突チェックのための空間構造のインデクス，そして検出された衝突発生部分を格納するデータ構造のインデクスとなっています．今回は唯一のシミュレーション仮想世界を使いますので，それぞれ 1 つのグローバル変数のみを定義しています．

シミュレーションを実行するためには，まず CreateWorld 関数でシミュレーション環境を初期化し，StepSimulation 関数によって spf ミリ秒だけシミュレーションを進めます．最後に，DestroyWorld 関数を呼び出して仮想環境を破棄するという流れになります．

```
namespace NPhysics
{
  extern dWorldID g_dWorld;              // 物理世界
  extern dSpaceID g_dSpace;              // 衝突検出空間
  extern dJointGroupID g_dContactGroup; // 衝突制約グループ

  // 物理世界の構築
  bool CreateWorld(void (* callback)(void *data, dGeomID o0, dGeomID o1));
  // 物理世界の破壊
  void DestroyWorld();

  // 物理世界の使用可否
  bool IsActiveWorld();

  // シミュレーションのステップ実行
  bool StepSimulation(double spf, void (* callback)(void *data, dGeomID o0, dGeomID o1));
};
```

次に，各関数の実装コードを示します．ほとんどdemura.net さんの ODE チュートリアルに沿って作成していますので，ここでは解説は省略します．なお，今回のシステムは重力成分を考慮したシミュレーションは行えないので，仮想環境は無重力に設定する必要があります．

```
namespace NPhysics
{
  dWorldID g_dWorld = NULL;
  dSpaceID g_dSpace = NULL;
  dJointGroupID g_dContactGroup = NULL;
};

// シミュレーションのステップ実行
bool NPhysics::StepSimulation(double spf, void (* callback)(void *data, dGeomID o0, dGeomID o1))
{
  dSpaceCollide(NPhysics::g_dSpace, 0, callback);
  dWorldStep(NPhysics::g_dWorld, spf);
  dJointGroupEmpty(NPhysics::g_dContactGroup);
  return true;
}
// 物理シミュレーション環境の構築
bool NPhysics::CreateWorld(void (* callback)(void *data, dGeomID o0, dGeomID o1))
{
  // すでに構築済みの場合はスキップ
  if (g_dWorld != NULL || g_dSpace != NULL)
    return false;

  // シミュレータ構築
  g_dWorld = dWorldCreate();
  // 重力の設定（無重力）
  dWorldSetGravity(g_dWorld, 0, 0, 0);
  // 衝突判定用ハッシュグリッド
  g_dSpace = dHashSpaceCreate(0);
  // 衝突判定用制約グループ
  g_dContactGroup = dJointGroupCreate(0);

  return true;
}

// 物理シミュレーション環境の破棄
void NPhysics::DestroyWorld()
{
  dJointGroupDestroy(g_dContactGroup);
  dSpaceDestroy(g_dSpace);
  dWorldDestroy(g_dWorld);

  g_dWorld = NULL;
  g_dSpace = NULL;
  g_dContactGroup = NULL;
}
// 物理シミュレータ環境の利用可否チェック
bool NPhysics::IsActiveWorld()
{
  return (g_dWorld != NULL) && (g_dSpace != NULL);
}
```

ODEを用いた剛体リンク系の構築 [†]

サンプルプログラムで作成した，3 自由度関節によって接続された 3 リンクアームの構造を下図に示します．リンク L0 は仮想世界に固定されているので，実際に運動するのはリンク L1，L2 のみです．ダミーリンク LD0 と LD1 は 3 自由度関節構造を定義するために仮想的に導入した，無視できるほどの大きさと質量しか与えらず，物理計算や順運動学計算には全く影響しないリンクです．Universal 関節 U0，U1 はそれぞれ 2 つの回転軸を持つ関節で，本サンプルではリンクの向きに垂直に 2 つ回転軸を設定することで，子リンクの向きを操作します（スイング動作）．一方，Hinge 関節 H0，H1 はそれぞれ 1 つの回転軸を持つ関節で，本サンプルではリンクの向きに平行に設定しており，子リンクのひねり回転を操作します（ツイスト動作）．当初は，ODE でサポートされている 3 自由度ボール-ソケット（球状）関節を用いていましたが，球状関節モデルでは回転可動域を設定できないので，このような Universal + ダミーリンク + Hinge 関節の実装に落ち着きました．

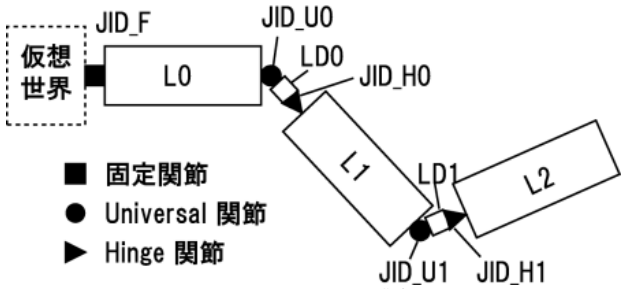

Fig.2 3 リンクアーム構造

リンクアームに関連した変数等の宣言部を示します．列挙子によってリンクと関節にインデクスを割り振っています(BodyID, JointID)．同様に，全ての関節自由度に対してもインデクスを与えています(AxisID)．ODE のリンクオブジェクトはそれぞれ m_vecBodies と m_vecGeoms 配列中に，与えられたインデクス順に格納されます．同様に ODE の関節オブジェクトは m_vecJoints 配列中に格納され，m_vecAxes には関節軸ベクトルが順次格納されます．

```
  .....

private:
  // リンクID (リンク L0, ダミーリンク LD0, リンク L1, ダミーリンク LD1, リンク L2)
  enum BodyID { BID_L0, BID_LD0, BID_L1, BID_LD1, BID_L2 };
  // 関節ID (固定関節 F, ユニバーサル関節 U0, ヒンジ関節 H0, ユニバーサル関節 U1, ヒンジ関節 H1)
  enum JointID { JID_F, JID_U0, JID_H0, JID_U1, JID_H1 };
  // 関節回転軸 (固定関節 F, ユニバーサル関節 U0-0, ヒンジ関節 H0, ユニバーサル関節 U1-0, ユニバーサル関節 U1-1, ヒンジ関節 H1)
  enum AxisID { AID_F, AID_U00, AID_U01, AID_H0, AID_U10, AID_U11, AID_H1 };
  //!<- 剛体リスト
  std::vector m_vecBodies;
  //! 衝突判定用ジオメトリリスト
  std::vector m_vecGeoms;
  //! 関節リスト
  std::vector m_vecJoints;
  //! 関節回転軸リスト
  std::vector m_vecAxes;
  //! シミュレーション実行スイッチ
  bool m_bDoSimulate;

private:
  // 3リンクアームの構築
  void InitializeArm();
  // アームの破棄
  void DestroyArm();

  .....
```

次にリンクアームの作成部の実装コードを示します．リンク L0 は立方体，リンク L1, L2 は同サイズの直方体で，全てのリンクは同質量に設定しています．また，全てのリンクの質量分布は一様と仮定しており，重心位置は各リンクの中心位置に設定し，慣性テンソルは ODE 内部で自動的に解析値に設定されます．一方，ダミーリンクは微小な球に微量の質量を与えて構築しています．詳細を書き出すと ODE の API の説明になってしまいますので，構造図と実装コードを見比べることで読み取ってみてください．

```
// 3 リンクアームの構築
void CSceneView::InitializeArm()
{
  DestroyArm();

  dMass m;
  dBodyID b;
  dGeomID g;
  dJointID j;

  //
  // リンク L0 (地面に固定) の構築
  b = dBodyCreate(NPhysics::s_dWorld);
  dMassSetBoxTotal(&m, 10.0, 5.0, 5.0, 5.0);
  dBodySetMass(b, &m);
  g = dCreateBox(NPhysics::s_dSpace, 5.0, 5.0, 5.0);
  dGeomSetBody(g, b);
  dBodySetPosition(b, 0, 0, -2.5);
  m_vecBodies.push_back(b);
  m_vecGeoms.push_back(g);
  //
  // リンク L1 の構築
  b = dBodyCreate(NPhysics::s_dWorld);
  dMassSetSphereTotal(&m, 0.0001, 0.001);
  dBodySetMass(b, &m);
  dBodySetPosition(b, 0, 0, 0);
  m_vecBodies.push_back(b);
  m_vecGeoms.push_back(NULL);
  //
  // ダミーリンク LD1 の構築
  b = dBodyCreate(NPhysics::s_dWorld);
  dMassSetCapsuleTotal(&m, 10.0, 3, 5.0, 20.0);
  dBodySetMass(b, &m);
  g = dCreateCapsule(NPhysics::s_dSpace, 5.0, 20.0);
  dGeomSetBody(g, b);
  dBodySetPosition(b, 0, 0, 10.0);
  m_vecBodies.push_back(b);
  m_vecGeoms.push_back(g);
  //
  // リンク L2 の構築
  b = dBodyCreate(NPhysics::s_dWorld);
  dMassSetSphereTotal(&m, 0.001, 0.001);
  dBodySetMass(b, &m);
  dBodySetPosition(b, 0, 0, 20.0);
  m_vecBodies.push_back(b);
  m_vecGeoms.push_back(NULL);
  //
  // ダミーリンク LD1 の構築
  b = dBodyCreate(NPhysics::s_dWorld);
  dMassSetCapsuleTotal(&m, 10.0, 3, 5.0, 20.0);
```

```
51 |    dBodySetMass(b, &m);
52 |    g = dCreateCapsule(NPhysics::s_dSpace, 5.0, 20.0);
53 |    dGeomSetBody(g, b);
54 |    dBodySetPosition(b, 0, 0, 30.0);
55 |    m_vecBodies.push_back(b);
56 |    m_vecGeoms.push_back(g);
57 |    //
58 |    // 固定関節 F : 地面 - リンク L0 間の関節
59 !    j = dJointCreateFixed(NPhysics::s_dWorld, 0);
60 |    dJointAttach(j, NULL, m_vecBodies[BID_L0]);
61 |    dJointSetFixed(j);
62 |    m_vecJoints.push_back(j);
63 |    m_vecAxes.push_back(D3DXVECTOR3(0, 0, 0));
64 |    //
65 |    // ユニバーサル関節 U0 : リンク L0 - ダミーリンク LD0 間の関節
66 !    j = dJointCreateUniversal(NPhysics::s_dWorld, 0);
67 |    dJointAttach(j, m_vecBodies[BID_L0], m_vecBodies[BID_LD0]);
68 |    dJointSetUniversalAnchor(j, 0, 0, 0);
69 |    dJointSetUniversalAxis1(j, 1.0, 0, 0);
70 |    dJointSetUniversalAxis2(j, 0, 1.0, 0);
71 |    m_vecJoints.push_back(j);
72 |    m_vecAxes.push_back(D3DXVECTOR3(1.0f, 0, 0));
73 |    m_vecAxes.push_back(D3DXVECTOR3(0, 1.0f, 0));
74 |    //
75 |    // ヒンジ関節 H0 : ダミーリンク LD0 - リンク L1 間の関節
76 !    j = dJointCreateHinge(NPhysics::s_dWorld, 0);
77 |    dJointAttach(j, m_vecBodies[BID_LD0], m_vecBodies[BID_L1]);
78 |    dJointSetHingeAnchor(j, 0, 0, 0.001);
79 |    dJointSetHingeAxis(j, 0, 0, 1.0);
80 |    m_vecJoints.push_back(j);
81 |    m_vecAxes.push_back(D3DXVECTOR3(0, 0, 1.0f));
82 |    //
83 |    // ユニバーサル関節 U1 : リンク L1 - ダミーリンク LD1 間の関節
84 !    j = dJointCreateUniversal(NPhysics::s_dWorld, 0);
85 |    dJointAttach(j, m_vecBodies[BID_L1], m_vecBodies[BID_LD1]);
86 |    dJointSetUniversalAnchor(j, 0, 0, 20.0);
87 |    dJointSetUniversalAxis1(j, 1.0, 0, 0);
88 |    dJointSetUniversalAxis2(j, 0, 1.0, 0);
89 |    m_vecJoints.push_back(j);
90 |    m_vecAxes.push_back(D3DXVECTOR3(1.0f, 0, 0));
91 |    m_vecAxes.push_back(D3DXVECTOR3(0, 1.0f, 0));
92 |    //
93 |    // ヒンジ関節 H1 : ダミーリンク LD1 - リンク L2 間の関節
94 !    j = dJointCreateHinge(NPhysics::s_dWorld, 0);
95 |    dJointAttach(j, m_vecBodies[BID_LD1], m_vecBodies[BID_L2]);
96 |    dJointSetHingeAnchor(j, 0, 0, 20.001);
97 |    dJointSetHingeAxis(j, 0, 0, 1.0);
98 |    m_vecJoints.push_back(j);
99 |    m_vecAxes.push_back(D3DXVECTOR3(0, 0, 1.0f));
100 ! }
```

アームの破棄コードは次の通りです. ODE の API を呼び出して各オブジェクトと格納配列をクリアしています.

```
1  // アームの破棄
2  void CSceneView::DestroyArm()
3  {
4    while (!m_vecJoints.empty()) {
5      if (m_vecJoints.back() != NULL)
6        dJointDestroy(m_vecJoints.back());
7      m_vecJoints.pop_back();
8    }
9    while (!m_vecGeoms.empty()) {
10     if (m_vecGeoms.back() != NULL)
11       dGeomDestroy(m_vecGeoms.back());
12     m_vecGeoms.pop_back();
13   }
14   while (!m_vecBodies.empty()) {
15     if (m_vecBodies.back() != NULL)
16       dBodyDestroy(m_vecBodies.back());
17     m_vecBodies.pop_back();
18   }
19   m_vecAxes.clear();
20 }
```

DirectX グラフィクスを用いたリンクアームの表示 [†]

リンクアーム表示部分の実装コードを示します. まず, リンクに与えられたインデクスを引数として, 各リンクの大きさや姿勢を取得するための関数を作成しました. CSceneView::GetBodySize 関数では, 各リンクの形状に対応した API によってリンクサイズを取得します. CSceneView::GetBodyCoord ではリンク中央のワールド座標と回転を取得し, リンク姿勢に対応したトランスフォーム行列を計算しています.

```
1  // 剛体サイズの取得
2  D3DXVECTOR3 CSceneView::GetBodySize(size_t bid) const
3  {
4    D3DXVECTOR3 v;
5    switch (bid)
6    {
7    case BID_L0:
8      dVector3 lv;
9      dGeomBoxGetLengths(m_vecGeoms[bid], lv);
10     v = D3DXVECTOR3(static_cast<float>(lv[0]), static_cast<float>(lv[1]), static_cast<float>(lv[2]));
11     break;
12   case BID_L1:
13   case BID_L2:
14     dReal r, l;
15     dGeomCapsuleGetParams(m_vecGeoms[bid], &r, &l);
16     v = D3DXVECTOR3(static_cast<float>(r * 2.0), static_cast<float>(r * 2.0), static_cast<float>(l));
17     break;
18   default:
19     v = D3DXVECTOR3(0, 0, 0);
20     break;
21   }
```

```
22 |    return v;
23 ! }
24 |
25 - // 剛体座標トランスフォーム行列の取得
26 ! D3DXMATRIX CSceneView::GetBodyCoord(size_t bid) const
27 - {
28 |    const dReal *pos = dBodyGetPosition(m_vecBodies[bid]);
29 |    const dReal *rot = dBodyGetRotation(m_vecBodies[bid]);
30 |
31 |    D3DXMATRIX m;
32 |    D3DXMatrixIdentity(&m);
33 |    for (size_t i = 0;i < 12;++i)
34 |      m[(i % 4) * 4 + (i / 4)] = static_cast<float>(rot[i]);
35 |    m._41 = static_cast<float>(pos[0]);
36 |    m._42 = static_cast<float>(pos[1]);
37 |    m._43 = static_cast<float>(pos[2]);
38 |    m._44 = 1.0f;
39 |
40 |    return m;
41 ! }
```

実際のリンクアーム描画関数は次のコードとなります．リンクサイズ→位置・回転→全体としてのスケーリング の順で合成されたトランスフォーム行列を用いて，単位立方体メッシュを表示しています．

```
1  - // アームの描画
2  ! void CSceneView::DrawArm()
3  - {
4  |    D3DXMATRIX s, m;
5  |    D3DXVECTOR3 sv;
6  |
7  |    sv = GetBodySize(BID_L0);
8  |    D3DXMatrixScaling(&s, sv.x, sv.y, sv.z);
9  |    m = s * GetBodyCoord(BID_L0);
10 |    m *= *D3DXMatrixScaling(&s, 0.01f, 0.01f, 0.01f);
11 |    DrawMeshSub(m_pCubeMesh, 0, m);
12 |
13 |    sv = GetBodySize(BID_L1);
14 |    D3DXMatrixScaling(&s, sv.x, sv.y, sv.z);
15 |    m = s * GetBodyCoord(BID_L1);
16 |    m *= *D3DXMatrixScaling(&s, 0.01f, 0.01f, 0.01f);
17 |    DrawMeshSub(m_pCubeMesh, 0, m);
18 |
19 |    sv = GetBodySize(BID_L2);
20 |    D3DXMatrixScaling(&s, sv.x, sv.y, sv.z);
21 |    m = s * GetBodyCoord(BID_L2);
22 |    m *= *D3DXMatrixScaling(&s, 0.01f, 0.01f, 0.01f);
23 |    DrawMeshSub(m_pCubeMesh, 0, m);
24 ! }
```

リンク系の物理特性の計算

ある関節回転軸にトルクを与えてリンクアームを回転させるためには，接続するリンクの質量や慣性モーメントだけでなく，末端に至る全ての子リンクの物理特性も考慮する必要があります．例えば，ヒンジ関節 H0 と H1 に同じ大きさのトルクを与えた場合には，ヒンジ H0 に与えられたトルクはリンク L1 と L2 を運動させるために消費されるのに対し，H1 に与えられたトルクは リンク L2 を運動させるためだけに消費されます．したがって，当然リンク L1 よりも L2 の回転量のほうが大きくなります．さらにその回転量の差は，トルクを与える瞬間のリンクアームの姿勢によっても変化します．こうした，各リンクの物理特性とアーム全体の姿勢をもとに，アーム全体の重心位置と慣性テンソル，そして各関節回転軸に関する慣性モーメントを求めるアルゴリズムを実装します．

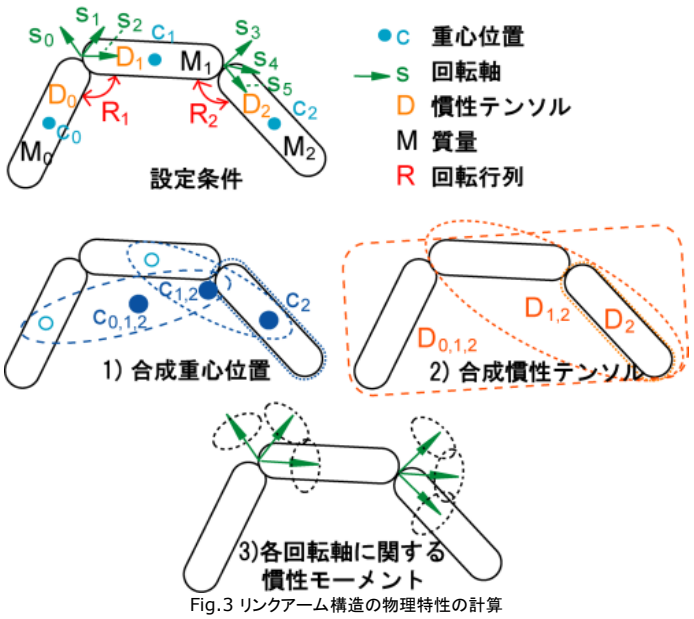

Fig.3 リンクアーム構造の物理特性の計算

まず，リンクアームに与えられる物理特性は，各リンクの質量 M，慣性テンソル D，重心位置 c と，リンク間の回転行列 R，そしてその関節回転軸 s です．次に，これらの条件のもとで，各リンクから末端に至るリンク全体の(1)合成重心位置と(2)合成慣性テンソルを計算します．最後に，これらの合成値を利用して各関節回転軸に関する(3)慣性モーメントを計算するという手順になります．これらの手順はニュートン・オイラー(Newton-Euler)法の一部としても知られており，O(リンク数)の効率的な計算オーダーであることからロボットシミュレーション分野では広く利用されています．

重心位置の合成

まず，リンクアームの合成重心座標を計算します．合成重心座標とは，リンク構造全体としての重心位置を示すもので，アームを構成する各リンクそれぞれの重心位置の質量加重平均に

よって計算できます．これは下図に示すようにリンク構造の姿勢によって変化します．CSceneView::CalcCompositeCOG 関数では，全ての子リンクの質量と姿勢を考慮した，各リンクの合成重心座標を計算します．つまり，リンク L0 に対してはリンク L0，L1，L2 の全ての重心の合成(res[0])，L1 に対しては L1，L2 の合成(res[1])，そして L2 は L2のみの重心位置(res[2])を計算します．

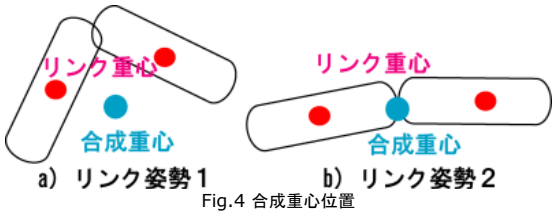

Fig.4 合成重心位置

```
// 各リンク座標系における合成重心座標の計算
std::vector CSceneView::CalcCompositeCOG() const
{
  std::vector res(3);
  D3DXVECTOR3 p, psum(0, 0, 0), cog;
  D3DXMATRIX m, mi;
  float msum = 0;

  dMass pmass, cmass;
  const dReal *pos;

  // ワールド座標系における L2 重心位置
  dBodyGetMass(m_vecBodies[BID_L2], &pmass);
  pos = dBodyGetPosition(m_vecBodies[BID_L2]);
  cog = D3DXVECTOR3(static_cast<float>(pos[0]), static_cast<float>(pos[1]), static_cast<float>(pos[2]));
  psum += cog * static_cast<float>(pmass.mass);
  msum += static_cast<float>(pmass.mass);
  // L2リンク座標系における L2 重心位置　＝リンク中心位置
  m = GetBodyCoord(BID_L2);
  D3DXMatrixInverse(&mi, NULL, &m);
  D3DXVec3TransformCoord(&res[2], &cog, &mi);

  // ワールド座標系における L1 重心位置
  dBodyGetMass(m_vecBodies[BID_L1], &pmass);
  pos = dBodyGetPosition(m_vecBodies[BID_L1]);
  p = D3DXVECTOR3(static_cast<float>(pos[0]), static_cast<float>(pos[1]), static_cast<float>(pos[2]));
  psum += p * static_cast<float>(pmass.mass);
  msum += static_cast<float>(pmass.mass);
  cog = psum / msum;
  // L1リンク座標系における L2 & L1 重心位置
  m = GetBodyCoord(BID_L1);
  D3DXMatrixInverse(&mi, NULL, &m);
  D3DXVec3TransformCoord(&res[1], &cog, &mi);

  // ワールド座標系における L0 重心位置
  dBodyGetMass(m_vecBodies[BID_L0], &pmass);
  pos = dBodyGetPosition(m_vecBodies[BID_L0]);
  p = D3DXVECTOR3(static_cast<float>(pos[0]), static_cast<float>(pos[1]), static_cast<float>(pos[2]));
  psum += p * static_cast<float>(pmass.mass);
  msum += static_cast<float>(pmass.mass);
  cog = psum / msum;
  // L0 リンク座標系における L2 & L1 & L0 重心位置
  m = GetBodyCoord(BID_L0);
  D3DXMatrixInverse(&mi, NULL, &m);
  D3DXVec3TransformCoord(&res[0], &cog, &mi);

  return res;
}
```

慣性テンソルの合成 [†]

慣性テンソルとはおおざっぱに言うと，全ての回転軸に関するリンクの"回転させにくさ"を行列（2次のテンソル）形式で表現したものです．例えば，長くて細い円柱を中心軸に沿ってひねるためには小さなトルクで十分ですが(Fig. a)，円柱の端を持って振り回すには比較的大きなトルクが必要です(Fig. b)．

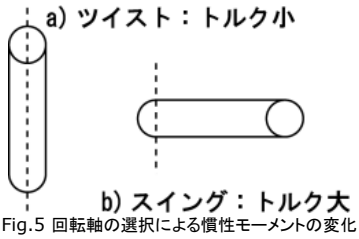

Fig.5 回転軸の選択による慣性モーメントの変化

リンクアーム全体の慣性テンソルは，各リンクの慣性テンソルとリンクアームの姿勢を合成することで計算されます．例えば，下図 a) のように各回転軸に対してリンクアームが均一に分布している場合には，いずれの軸の"回転させにくさ"もほぼ同じですが，図b)のように一方の軸に沿った姿勢をとる場合には，同じ角度だけ回転させるためのトルクに大きな差が生じます．

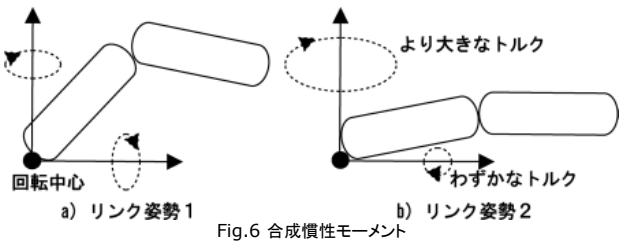

Fig.6 合成慣性モーメント

こうした合成慣性テンソルを計算するためには，子リンクの局所座標系で定義されている慣性テンソルを，親リンクの局所座標系からみた値に変換する必要があります．すなわち，次のような数式によって，子リンクの中心位置を原点とする慣性テンソルを，親リンクの中心を原点とし，かつ 2 つのリンク間の回転を考慮した慣性テンソルに変換する必要があります．

$$D' = R^T DR - M([p][cR] + [cR][p] + [p][p])$$

ここで, D と D' はそれぞれ子リンクの慣性テンソルと, 親リンクとそれに接続する子リンクの合成慣性テンソルを表します. M は子リンクの質量, R は子リンクから親リンクへの回転行列, p は子リンクの局所座標系から見た親リンク中心の位置ベクトル, c は子リンクの局所座標系からみた子リンクの重心位置の位置ベクトルを表します.

また, $[\cdot\cdot\cdot]$ はベクトルから 歪対称行列(skew-symmetric matrix)を計算することを表します. 歪対称行列の定義は次の通りです.

$$[x\ y\ z] = \begin{pmatrix} 0 & -z & y \\ z & 0 & -x \\ -y & x & 0 \end{pmatrix}$$

これらの数式をもとに, 各リンクの合成慣性テンソルを計算するコードを次に示します. まず, リンク L2 の慣性テンソルを計算し res[2] に格納します. 次に, リンク L1 の慣性テンソルを pi に計算し, リンク L2 の慣性テンソルを変換した値を加えることで, L1 と L2 の合成慣性テンソルを res[1] に格納します. 同様に, リンク L0 の慣性テンソルに res[1] に格納されている慣性テンソルを変換して加えることで, L0, L1, そして L2 の合成慣性テンソルを res[0] に計算しています.

```
// スキュー・対称行列の計算
D3DXMATRIX skew_symmetric_matrix(const D3DXVECTOR3 &v)
{
  D3DXMATRIX m;
  D3DXMatrixIdentity(&m);

  m._11 = 0;    m._12 = -v.z; m._13 = v.y;
  m._21 = v.z;  m._22 = 0;    m._23 = -v.x;
  m._31 = -v.y; m._32 = v.x;  m._33 = 0;
  m._44 = 0;
  return m;
}

// 各リンク座標系における合成慣性テンソルの計算
std::vector CSceneView::CalcCompositeInertiaTensor() const
{
  std::vector res(3);

  dMass pmass, cmass;
  D3DXMATRIX pc, pr, pi;
  D3DXMATRIX cc, cr;
  D3DXMATRIX sst, ssc, tmp, rot;
  D3DXVECTOR3 trans, centr, cent;

  //
  // リンク L2 の慣性テンソル on リンク L2 座標系
  dBodyGetMass(m_vecBodies[BID_L2], &pmass);
  pc = GetBodyCoord(BID_L2);
  pr = NBasicGeometry::Orthogonalize(pc);
  D3DXMatrixScaling(&pi, static_cast<float>(pmass.I[0]), static_cast<float>(pmass.I[5]), static_cast<float>(pmass.I[10]));
  pi._44 = 0;
  res[2] = pi;

  //
  // リンク L2 & L1 の慣性テンソル on リンク L1 座標系
  cc = pc;  cr = pr;
  dBodyGetMass(m_vecBodies[BID_L1], &cmass);
  pc = GetBodyCoord(BID_L1);
  pr = NBasicGeometry::Orthogonalize(pc);
  D3DXMatrixScaling(&pi, static_cast<float>(pmass.I[0]), static_cast<float>(pmass.I[5]), static_cast<float>(pmass.I[10]));
  pi._44 = 0;

  rot = *D3DXMatrixInverse(&tmp, NULL, &cr) * pr;
  trans = *reinterpret_cast(&pc._41) - *reinterpret_cast(&cc._41);
  cent = D3DXVECTOR3(static_cast<float>(cmass.c[0]), static_cast<float>(cmass.c[1]), static_cast<float>(cmass.c[2]));
  D3DXVec3TransformCoord(&r, &, &rot);
  sst = skew_symmetric_matrix(trans);
  ssc = skew_symmetric_matrix(centr);
  res[1] = pi + *D3DXMatrixTranspose(&tmp, &rot) * res[2] * rot  -  static_cast<float>(cmass.mass) * (sst * ssc + ssc * sst + sst * sst);

  //
  // リンク L2 & L1 & L0 の慣性テンソル on リンク L0 座標系 = ワールド座標系
  cc = pc;  cr = pr;
  dBodyGetMass(m_vecBodies[BID_L0], &cmass);
  pc = GetBodyCoord(BID_L0);
  pr = NBasicGeometry::Orthogonalize(pc);
  D3DXMatrixScaling(&pi, static_cast<float>(pmass.I[0]), static_cast<float>(pmass.I[5]), static_cast<float>(pmass.I[10]));
  pi._44 = 0;

  rot = *D3DXMatrixInverse(&tmp, NULL, &cr) * pr;
  trans = *reinterpret_cast(&pc._41) - *reinterpret_cast(&cc._41);
  cent = D3DXVECTOR3(static_cast<float>(cmass.c[0]), static_cast<float>(cmass.c[1]), static_cast<float>(cmass.c[2]));
  D3DXVec3TransformCoord(&r, &, &rot);
  sst = skew_symmetric_matrix(trans);
  ssc = skew_symmetric_matrix(centr);
  res[0] = pi + *D3DXMatrixTranspose(&tmp, &rot) * res[1] * rot  -  static_cast<float>(cmass.mass) * (sst * ssc + ssc * sst + sst * sst);

  return res;
}
```

各関節回転軸に関する慣性モーメントの算出

算出された合成重心位置と合成慣性テンソルにもとづき, 各関節回転軸に関する子リンク系全体の慣性モーメントを次式によって計算します.

$$m_i = s_i D_{i\cdots n} s_i + M_{i\cdots n} \left| (c_{i\cdots n} - d_i) \times s_i \right|^2$$

s_i は各関節回転軸方向を表すベクトル, $D_{i\cdots n}$ と $c_{i\cdots n}$, $M_{i\cdots n}$ はそれぞれリンク i から末端リンクまでに至る合成慣性テンソルと合成重心位置, 合成質量をを示します. また, d_i はリンクの中心位置から親リンクと接続する関節位置へのベクトルを表します. この式の実装コードを以下に示します. d_i は全てリンク中心からリンク始端点に至るベクトルなので, 各リンクの長さから計算しています.

```
// 各関節軸回転に対する慣性モーメントの計算
std::vector<double> CSceneView::CalcMomentOfInertia() const
{
    D3DXMATRIX it;
    dMass mass0, mass1;
    dBodyGetMass(m_vecBodies[BID_L1], &mass0);
    dBodyGetMass(m_vecBodies[BID_L2], &mass1);

    dReal r, l0, l1;
    dGeomCapsuleGetParams(m_vecGeoms[BID_L1], &r, &l0);
    dGeomCapsuleGetParams(m_vecGeoms[BID_L2], &r, &l1);

    std::vector<double> moi(6);
    std::vector ccog = CalcCompositeCOG();
    std::vector ccit = CalcCompositeInertiaTensor();
    D3DXVECTOR3 dv, vt;
    D3DXVECTOR4 vt4;


    dv = D3DXVECTOR3(0, 0, -static_cast<float>(l1 * 0.5));
    dv = ccog[2] - dv;

    D3DXVec3Transform(&vt4, &m_vecAxes[AID_U10], &ccit[2]);
    moi[3] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_U10]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_U10]);
    moi[3] += mass1.mass * powf(D3DXVec3Length(&vt), 2.0f);

    D3DXVec3Transform(&vt4, &m_vecAxes[AID_U11], &ccit[2]);
    moi[4] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_U11]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_U11]);
    moi[4] += mass1.mass * powf(D3DXVec3Length(&vt), 2.0f);

    D3DXVec3Transform(&vt4, &m_vecAxes[AID_H1], &ccit[2]);
    moi[5] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_H1]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_H1]);
    moi[5] += mass1.mass * powf(D3DXVec3Length(&vt), 2.0f);


    dv = D3DXVECTOR3(0, 0, -static_cast<float>(l0 * 0.5));
    dv = ccog[1] - dv;

    D3DXVec3Transform(&vt4, &m_vecAxes[AID_U00], &ccit[1]);
    moi[0] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_U00]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_U00]);
    moi[0] += (mass0.mass + mass1.mass) * powf(D3DXVec3Length(&vt), 2.0f);

    D3DXVec3Transform(&vt4, &m_vecAxes[AID_U01], &ccit[1]);
    moi[1] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_U01]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_U01]);
    moi[1] += (mass0.mass + mass1.mass) * powf(D3DXVec3Length(&vt), 2.0f);

    D3DXMatrixIdentity(&it);
    for (size_t i = 0;i < 3;++i)
      it[i * 5] = static_cast<float>(mass0.I[i * 5]);
    it._44 = 0;
    D3DXVec3Transform(&vt4, &m_vecAxes[AID_H0], &it);
    moi[2] = D3DXVec3Dot(reinterpret_cast(&vt4), &m_vecAxes[AID_H0]);
    D3DXVec3Cross(&vt, &dv, &m_vecAxes[AID_H0]);
    moi[2] += (mass0.mass + mass1.mass) * powf(D3DXVec3Length(&vt), 2.0f);

    return moi;
}
```

PDパラメータ制御によるキーフレーム補間

ここまでで解説した技術はさほど目新しいものではなく，ロボット工学などの分野では既に広く使われているものです．On the beat の技術的なコアは，フィードバック制御を利用した物理計算によって，任意の時間に任意の角度をとる多リンクアームアニメーションを計算する点です．そのため，この論文では PD 制御におけるフィードバックゲインを解析的に計算するアプローチを取っています．

PDフィードバック制御

フィードバックコントローラを用いたロボットアーム制御の目的の一つは，なるべき早い時間内に，指定した「姿勢」と「速度」を満たすような関節駆動トルクを計算することです．つまり，関節角度の目標角度からの差と，関節角速度の目標角速度からの差を全て 0 にするように，それぞれの差の大きさに応じて最適なトルクを算出することが求められます．その計算法の一つに，PD 制御と呼べる技術があります．これは，目標からの角度差に比例したトルクを与える P 制御(P: Proportional=比例)に，角速度差に比例したトルクを与える D 制御(D: Derivative=微分 ← 角速度は角度の微分)を組み合わせた制御法です．また，PD 制御は力学的キャラクタアニメーション法で使われているバネ-ダンパモデルそのもので，P 項がバネ，D 項がダンパにそのまま対応します．ちなみに，制御系分野では角変位量の時間総和も考慮した PID 制御(I: Integral=積分←角変位の時間積分)も使われますが，CGではあまり目にしません(なぜでしょうか…？)．

ある時刻におけるリンクアームの関節角度と角速度を θ と $\dot{\theta}$，目標角度と角速度をそれぞれ θ_d と $\dot{\theta}_d$ とするとき，PD 制御では次式によってトルク τ を計算します．

$$\tau = k_s(\theta_d - \theta) + k_d(\dot{\theta}_d - \dot{\theta})$$

目標値からの角度差と角速度差に適当なフィードバックゲイン k_s，k_d を乗算して和を取ることでトルクを求めます．つまり，目標との差が大きい場合は大きな力を加えて高速に目標値に近づけ，目標に近くなったら小さなトルクを与えます．ただし，フィードバックゲイン k_s，k_d の設定によっては，目標に収束せずに発散したり収束に多大な時間がかかることがあります．したがって，安定かつ高速な制御を実現するためには，フィードバックゲインを手作業で入念にチューニングする必要がありました．収束に要する時間を指定することは，なおのこと難しい問題です．

この問題に対し，On the Beatでは フィードバックゲインを次式によって計算します．

$$k_s = \frac{m}{\lambda^2}$$

$$k_d = 2\frac{m}{\lambda}$$

ここで，$\lambda = \Delta t / n$ かつ $n = -1/\ln(f)$ で，m はある関節回転軸に関するリンクアームの慣性モーメント，Δt はキーフレーム間の時間間隔を表します．なお，手動パラメータ f は 0.9 に設定することが推奨されています．

親関節運動を考慮したトルク補正

上式では，エフェクタに至る全ての子リンクの姿勢が変化しないと暗に仮定しています．しかし実際は，ある関節に加えられたトルクは直続するリンクだけでなく全ての子リンクに加速度を与えるため，結局は全ての子関節も回転させます．したがって上式によって算出されたトルクをそのまま用いると，キーフレーム補間計算に若干の誤差が生じることになります．つまり，キーフレーム補間の精度を向上させるためには，ある関節のトルクを算出する際に，ルートに至る全ての親関節に加えられているトルクもしくは加速度を考慮しなければなりません．

まず，各親関節に加えられているトルクによって，子リンクに発生する加速度の大きさを計算します．このとき，加速度の大きさはワールド座標系において計算します．

$$a_i = \sum_{j=0}^{i-1} \frac{\tau_j}{m_j} G_j^w s_j$$

ここで，a_i はリンク i に発生する加速度，τ_j と m_j は親関節 j に加えられるトルク と慣性モーメント，s_j は親関節の回転軸ベクトル，トランスフォーム行列 G_j^w は 関節 j の局所座標系をワールド座標系に変換するためのトランスフォーム行列です．つまりこの式では，各親関節のトルクによって発生する加速度の大きさをワールド座標系において計算し，全ての親リンクについての総和を計算しています．

次に，算出された加速度を子関節の局所座標系に座標変換し，トルク補正項として算出します．

$$a_i \cdot (G_i^w s_i)$$

この補正項を PD フィードバック式に加えることで，最終的なトルク算出計算式となります．以上の全ての計算をまとめることで，次式が得られます．

$$\tau_i = \frac{m_i}{\lambda^2}(\theta_d - \theta) + 2\frac{m_i}{\lambda}(\dot{\theta}_d - \dot{\theta}) + a_i \cdot (G_i^w s_i)$$

関節トルクの算出とフォワードダイナミクス

PD フィードバック制御の実装コードを以下に示します．キーフレーム姿勢は配列 target に指定しており，各キーフレームにおいて全ての関節角速度が 0 となるように指定しています．なお，あまりに過大なトルクが算出された場合には，適当な大きさに打ち切っています．ODE を用いた実装では，dJointGet～Angle API によって各関節角度，dJointGet～AngleRate API によって各関節角速度を取得し，目標値からの誤差に応じてトルクを算出しています．なお，トルク補正項は関節 U1 と H1 についてのみ，U0 と H0 のトルクを考慮した補正量を計算しています．これは，ユニバーサル関節とヒンジ関節をあわせて 1 つの 3 自由度関節とみなしているためです．ただ，実際には別物として扱うのが正しいと思います(つまり，関節 H0 のトルク計算には関節 U0 を考慮した補正項を導入すべきでしょう)．

```
1   void CSceneView::StepSimulation()
2   {
3     // キーフレーム×2
4     static const double target[2][6] = {
5       {D3DX_PI / 4.0, D3DX_PI / 4.0, D3DX_PI / 4.0, D3DX_PI / 4.0, D3DX_PI / 4.0, D3DX_PI / 4.0},
6       {-D3DX_PI / 4.0, -D3DX_PI / 4.0, -D3DX_PI / 4.0, -D3DX_PI / 4.0, -D3DX_PI / 4.0, -D3DX_PI / 4.0}
7     };
8     static size_t sim_count = 0;
9
10    // ラムダ項の計算（キーフレーム間隔 5.0 固定なので定数として算出）
11    const double lambda = 5.0 / (-1.0 / log(0.9));
12    double theta, omega, torque[3], err = 0;
13    std::vector<double> moi = CalcMomentOfInertia();
14    D3DXMATRIX m;
15    D3DXVECTOR3 acc = D3DXVECTOR3(0, 0, 0), ts;
16    // 往路/復路スイッチ
17    size_t key = (sim_count / 500) % 2;
18
19    // 関節 U0 - 1 の PD フィードバック
20    m = NBasicGeometry::Orthogonalize(GetBodyCoord(BID_L1));
21    theta = -dJointGetUniversalAngle1(m_vecJoints[JID_U0]);
22    omega = -dJointGetUniversalAngle1Rate(m_vecJoints[JID_U0]);
23    torque[0] = (target[key][0] - theta) * moi[0] / (lambda * lambda) + 2.0 * moi[0] / lambda * (0 - omega);
24    err += pow(target[key][0] - theta, 2.0);
25
26    // 関節 U0 - 2 の PD フィードバック
27    theta = -dJointGetUniversalAngle2(m_vecJoints[JID_U0]);
28    omega = -dJointGetUniversalAngle2Rate(m_vecJoints[JID_U0]);
29    torque[1] = (target[key][1] - theta) * moi[1] / (lambda * lambda) + 2.0 * moi[1] / lambda * (0 - omega);
30    err += pow(target[key][1] - theta, 2.0);
31
32    // 関節 H0 の PD フィードバック
33    theta = -dJointGetHingeAngle(m_vecJoints[JID_H0]);
34    omega = -dJointGetHingeAngleRate(m_vecJoints[JID_H0]);
35    torque[2] = (target[key][2] - theta) * moi[2] / (lambda * lambda) + 2.0 * moi[2] / lambda * (0 - omega);
36    err += pow(target[key][2] - theta, 2.0);
37
38    // 過大なトルクの打ち切り
39    for (size_t i = 0;i < 3;++i)
40      torque[i] = (torque[i] < 0 ? -1.0 : 1.0) * (fabs(torque[i]) > 5.0e5 ? 5.0e5 : fabs(torque[i]));
41
42    // トルク付与
43    dJointAddUniversalTorques(m_vecJoints[JID_U0], -torque[0], -torque[1]);
44    dJointAddHingeTorque(m_vecJoints[JID_H0], -torque[2]);
45
46    // 関節 U0, H0 によって発生した加速度の総和計算
47    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_U00], &m);
48    acc += static_cast<float>(torque[0] / moi[0]) * ts;
49    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_U01], &m);
50    acc += static_cast<float>(torque[1] / moi[1]) * ts;
51    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_H0], &m);
52    acc += static_cast<float>(torque[2] / moi[2]) * ts;
53
54    // 関節 U1 - 1 の PD フィードバック
55    m = NBasicGeometry::Orthogonalize(GetBodyCoord(BID_L2));
56    theta = -dJointGetUniversalAngle1(m_vecJoints[JID_U1]);
```

```
57 |    omega = -dJointGetUniversalAngle1Rate(m_vecJoints[JID_U1]);
58 |    torque[0] = (target[key][3] - theta) * moi[3] / (lambda * lambda) + 2.0 * moi[3] / lambda * (0 - omega);
59 |    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_U10], &m);
60 |    torque[0] += D3DXVec3Dot(&acc, &ts);
61 |    err += pow(target[key][3] - theta, 2.0);
62 |
63 |    // 関節 U1 - 2 の PD フィードバック
64 !    theta = -dJointGetUniversalAngle2(m_vecJoints[JID_U1]);
65 |    omega = -dJointGetUniversalAngle2Rate(m_vecJoints[JID_U1]);
66 |    torque[1] = (target[key][4] - theta) * moi[4] / (lambda * lambda) + 2.0 * moi[4] / lambda * (0 - omega);
67 |    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_U11], &m);
68 |    torque[1] += D3DXVec3Dot(&acc, &ts);
69 |    err += pow(target[key][4] - theta, 2.0);
70 |
71 |    // 関節 H1 の PD フィードバック
72 !    theta = -dJointGetHingeAngle(m_vecJoints[JID_H1]);
73 |    omega = -dJointGetHingeAngleRate(m_vecJoints[JID_H1]);
74 |    torque[2] = (target[key][5] - theta) * moi[5] / (lambda * lambda) + 2.0 * moi[5] / lambda * (0 - omega);
75 |    D3DXVec3TransformCoord(&ts, &m_vecAxes[AID_H1], &m);
76 |    torque[2] += D3DXVec3Dot(&acc, &ts);
77 |    err += pow(target[key][5] - theta, 2.0);
78 |
79 |    // 過大なトルクの打ち切り
80 !    for (size_t i = 0;i < 3;++i)
81 |      torque[i] = (torque[i] < 0 ? -1.0 : 1.0) * (fabs(torque[i]) > 5.0e5 ? 5.0e5 : fabs(torque[i]));
82 |
83 |    // トルク付与
84 !    dJointAddUniversalTorques(m_vecJoints[JID_U1], -torque[0], -torque[1]);
85 |    dJointAddHingeTorque(m_vecJoints[JID_H1], -torque[2]);
86 |
87 |    // シミュレーションを1ステップ進める
88 !    NPhysics::StepSimulation(0.01, callback_collision);
89 |    ++sim_count;
90 |    m_sMessage.Format("Time: %5.2f¥nError: %5.2f", sim_count / 100.0f, err);
91 ! }
```

結果

キーフレーム時間間隔の変化と，リンクの物理特性の変化によってアニメーションがどのように変化するのか比較してみました．

まず，リンク L1 と L2 を同じ質量に設定し，キーフレーム間隔を 5.0 とした場合のアニメーションです．この結果を基準とします．

均一質量-キーフレーム間隔 5.0 (741kb)

次に，キーフレーム間隔を 2.0 に短縮した結果です．キーフレーム付近で計算が破綻してリンクがバタついていますが，なんとか収束しています．

均一質量-キーフレーム間隔 2.0 (353kb)

次に，リンク L1 の質量を 4 分の 1 に軽量化した場合の結果です．

L1リンク軽量化-キーフレーム間隔 5.0 (745kb)

最後に，リンク L2 の質量を 10 分の 1 に軽量化した場合の結果です．

L2リンク軽量化-キーフレーム間隔 5.0 (734kb)

このように，あまり目だった差は見られません．．．物理特性によってアニメーションが変化してくれると期待していたので，少し残念な結果となりました．

まとめ

物理計算によって多リンクアームのキーフレームアニメーションを計算する技術を紹介しました．重力の影響を考慮できないので真の意味での物理シミュレーションには利用できませんが，物理計算っぽいアニメーションを手軽に作れる点で実用的な技術だと思います．なお，オリジナルの論文ではモーションキャプチャデータのトラッキング＋シミュレーション応用なども示してあります．まだまだ改善の余地は多いのですが，これからの発展が期待できそうです．
