ラグドールシミュレーション

http://www.tmps.org/index.php?%A5%E9%A5%B0%A5%C9%A1%BC%A5%EB%A5%B7%A5%DF%A5%E5%A5%EC%A1%BC%A5%B7%A5%E7%A5%F3

Movie 1. 落下シミュレーション

Movie 2. キーボードによるインタラクション

XNA GSからのODEの利用では，XNA Game Studio 2.0 から Open Dynamics Engine（ODE） の API を利用する方法をまとめました．本記事ではその応用として，簡単なキャラクタの物理シミュレーションを行うプログラムを紹介します．ただし，キャラクタは外力に従って受動的に振舞うだけで能動的に動けない，いわゆるラグドールを扱います．例えば，Movie.1 では重力に逆らうことなく後ろ向きに落下しており，Movie.2 では与えられた外力によって四方八方へ吹き飛ばされています．このように，ラグドールは何のリアクションを起こすこともなく，力を加えられられるがままに振舞います．

シミュレーション自体は ODE を用いて比較的簡単に実現できますので，本記事では XNA 向けのクラス作成を通じて各 API の利用手順と，簡単なキャラクタモデルの構築手順を中心に紹介します．

参考文献，関連記事

- Open Dynamics Engine
  - いわずもがな．オープンソースの物理演算ライブラリです．
- demura.net
  - ODE に関する日本の総本山です．各 API の詳細についてはこちらをご参照ください．
- XNA GSからのODEの利用
  - 本記事のベースです．
- 物理計算によるキーフレーム補間
  - ODEで多リンク構造を定義し，物理シミュレーションする方法を紹介しています（サンプルはC++）．いくつかのクラスはこちらから移植しました．

サンプルプログラム

サンプルプログラムは Visual Studio 2005 Professional と XNA Game Studio 2.0 を用いて作成しました．また，ODE の DLL 名は "ode32.dll" と仮定しています．なお，"ode32.dll" は SourceForge からダウンロードしたソースファイル（ode-src-0.9.zip）から構築した，x86 版の DLL です．

Visual Studio 2005 + XNA GS 2.0プロジェクト（44KB）

プログラムを実行すると，10個の剛体で構成された人型のラグドールが落下します．また，キーボードの十字キーを操作することで，胴体ボディに外力を加えることができます．「↑」「↓」キーで視線方向への力，「←」「→」キーで水平方向への力を加えられます．

基本クラスの作成

本記事では，ODE の API をいくつかのクラスでカプセル化し，多関節・多リンクキャラクタを構築します．まず，シミュレーション全体に関するクラス，立方体や球などの基本プリミティブを定義するクラス，そしてキャラクタの各ボーンを定義するクラスを作成します．そして，複数のボーンを関節で接続することでキャラクタの全身を定義します．

物理空間クラス

全てのシミュレーションに共通の ODE の初期化処理などは，仮想物理空間を管理する Physics クラスとしてまとめます．仮想物理空間を複数定義することは少ないと判断，つまりこのクラスのインスタンスを複数作成する場合は まれ だと判断したので，静的クラスとして作成しました．

まず，クラスフィールドとそれに対応するプロパティを示します．フィールドはそれぞれ物理空間のID（world），衝突検出空間のID（space），接触ジョイントグループのID（contactGroup），そして衝突検出時のデフォルトの動作を記述したコールバックデリゲート（defaultCollisionCallback）で構成されます．また，読み取り専用のプロパティとして，Gravity プロパティも用意しました．読んで字のごとく，仮想物理空間の重力ベクトルを取得するために利用します．

```
1  #region field
2  private static WorldID world = WorldID.Zero;                     // ODE - ワールドID
3  private static SpaceID space = SpaceID.Zero;                     // ODE - スペースID
4  private static JointGroupID contactGroup = JointGroupID.Zero;    // ODE - 接触ジョイントグループID
5  private static Ode.NearCallback defaultCollisionCallback = null; // 衝突判定コールバック
6  #endregion
7
8  #region property
9  public static WorldID World
10 {
11   get { return world; }
12   private set { space = value; }
13 }
14 public static SpaceID Space
15 {
16   get { return space; }
17   private set { space = value; }
18 }
19 public static JointGroupID ContactGroup
20 {
21   get { return contactGroup; }
22   private set { contactGroup = value; }
23 }
24 public static Vector3 Gravity
25 {
26   get
27   {
28     Ode.Vector3 g;
29     Ode.WorldGetGravity(world, out g);
30     return new Vector3(g.X, g.Y, g.Z);
31   }
32 }
33 public static Ode.NearCallback DefaultCollisionCallback
34 {
35   get { return defaultCollisionCallback; }
36 }
37 #endregion
```

次に，Physics クラスのメソッドをまとめて示します．重力ベクトルを引数として仮想物理空間を初期化する Create，それを破棄する Destroy，仮想物理空間の初期化状態を判定する IsActive，そしてステップ時間 [sec/frame] を指定してシミュレーションを進める Step メソッドで構成されます．各メソッドは基本的な ODE-API のみを呼び出すだけですので，実装の説明は省略します．

```
1  public static void Create(Vector3 gravity)
2  {
3    if (world != WorldID.Zero || space != SpaceID.Zero)
4      return;
5    Ode.InitODE();
6
7    world = Ode.WorldCreate();
8    Ode.WorldSetGravity(world, gravity.X, gravity.Y, gravity.Z);
9
10   space = Ode.HashSpaceCreate(SpaceID.Zero);
11   contactGroup = Ode.JointGroupCreate(0);
12
13   defaultCollisionCallback = new Ode.NearCallback(CollisionCallback);
14 }
15
16 public static void Destroy()
17 {
18   Ode.JointGroupDestroy(contactGroup);
19   Ode.SpaceDestroy(space);
```

```
20 |    Ode.WorldDestroy(world);
21 |
22 |    contactGroup = JointGroupID.Zero;
23 |    space = SpaceID.Zero;
24 |    contactGroup = JointGroupID.Zero;
25 |
26 |    Ode.CloseODE();
27 ! }
28
29    public static bool IsActive()
30 {
31 |    return (world != WorldID.Zero) && (space != SpaceID.Zero);
32 ! }
33
34    public static void Step(Real spf, Ode.NearCallback proc)
35 {
36 |    Ode.SpaceCollide(space, IntPtr.Zero, proc);
37 |    Ode.WorldStep(world, spf);
38 |    Ode.JointGroupEmpty(contactGroup);
39 ! }
```

仮想物理空間のデフォルトの衝突検出コールバックメソッドを以下に示します．まず，9～12行目で，関節で接続された隣合う剛体同士の接触は無視します．衝突モデルは弾性衝突モデルとし(15行目)，反発係数は 0.5(18行目)，反発が発生する最小速度は 0.1 m/s(19行目)とします．また，物体間に働く摩擦の大きさは無限大(16，17行目．実際は単精度浮動小数の最大値)としました．

```
1   private static void CollisionCallback(IntPtr data, GeomID geom0, GeomID geom1)
2  {
3  |   const int N = 20;
4  |   Ode.ContactGeom[] contacts = new Ode.ContactGeom[N];
5  |
6  |   BodyID b0 = Ode.GeomGetBody(geom0);
7  |   BodyID b1 = Ode.GeomGetBody(geom1);
8  |
9  |   if (b0 != BodyID.Zero && b1 != BodyID.Zero && Ode.AreConnected(b0, b1) == true)
10 |     return;
11 |   if (b0 != BodyID.Zero && b1 != BodyID.Zero && Ode.AreConnectedExcluding(b0, b1, Ode.JointType.Contact) == true)
12 |     return;
13 |
14 |   Ode.Contact contact = new Ode.Contact();
15 |   contact.surface.mode = Ode.ContactFlags.Bounce;
16 |   contact.surface.mu = Real.MaxValue;
17 |   contact.surface.mu2 = Real.MaxValue;
18 |   contact.surface.bounce = 0.5f;
19 |   contact.surface.bounce_vel = 0.1f;
20 |
21 |   int n = Ode.Collide(geom0, geom1, N, contacts, Ode.ContactGeom.SizeOf);
22 |   for (int i = 0; i < n; ++i)
23 |   {
24 |       contact.geom = contacts[i];
25 |       JointID c = Ode.JointCreateContact(world, contactGroup, ref contact);
26 |       Ode.JointAttach(c, b0, b1);
27 !   }
28 ! }
```

基本プリミティブクラス [†]

球，直方体，円筒などの基本的な物体プリミティブのクラスを作成します．まず，それらの共通部分をまとめた抽象クラス PhysicalPrimitive を作成します．クラスの宣言と，フィールドの定義を以下に示します．

```
1   abstract public class PhysicalPrimitive : IPhysicalObject, IDisposable
2  {
3  |   protected BodyID body = BodyID.Zero;    // ODE - ボディ
4  |   protected GeomID geom = GeomID.Zero;    // ODE - ジオメトリ
5  |   protected Model model = null;           // XNA - 表示モデル
6  |   protected Vector3 scale = Vector3.Zero; // 単位サイズ基本プリミティブの表示スケール
7  |   protected float weight = 0.0f;          // 質量 [kg]
```

PhysicalPrimitive クラスは，ODE のボディ:シミュレーション対象の剛体(質点)モデルや，ジオメトリ:接触判定用の形状モデルと併せて，表示用のXNAのモデル，単位球や単位立方体からのスケール，そして物体の質量をフィールドに持ちます．

次に，PhysicalPrimitive クラスのプロパティを示します．なお，各フィールドに単に読み取りアクセスするためのプロパティ(Body，Geom等)は省略します．

```
1   public Vector3 Position
2  {
3  |   get {
4  |     Ode.Vector3 v;
5  |     Ode.BodyCopyPosition(body, out v);
6  |     return new Vector3(v.X, v.Y, v.Z);
7  !   }
8  |   set { Ode.BodySetPosition(body, value.X, value.Y, value.Z); }
```

```
 9 ! }
10   public Matrix Orientation
11 ┼ {
12 ┼   get {
13 |     Ode.Matrix3 r;
14 |     Ode.BodyCopyRotation(body, out r);
15 |
16 |     Matrix m = Matrix.Identity;
17 |     m.M11 = r.M11;
18 |     m.M12 = r.M12;
19 |     m.M13 = r.M13;
20 |     m.M21 = r.M21;
21 |     m.M22 = r.M22;
22 |     m.M23 = r.M23;
23 |     m.M31 = r.M31;
24 |     m.M32 = r.M32;
25 |     m.M33 = r.M33;
26 |
27 |     return m;
28 !   }
29 ┼   set {
30 |     Ode.Matrix3 r;
31 |     r.M11 = value.M11;
32 |     r.M12 = value.M12;
33 |     r.M13 = value.M13;
34 |     r.M41 = 0;
35 |     r.M21 = value.M21;
36 |     r.M22 = value.M22;
37 |     r.M23 = value.M23;
38 |     r.M42 = 0;
39 |     r.M31 = value.M31;
40 |     r.M32 = value.M32;
41 |     r.M33 = value.M33;
42 |     r.M43 = 0;
43 |
44 |     Ode.BodySetRotation(body, ref r);
45 !   }
46 ! }
47   public Matrix Coordinate
48 ┼ {
49 ┼   get {
50 |     Vector3 pos = Position;
51 |     Matrix rot = Orientation;
52 |     rot.M41 = pos.X;
53 |     rot.M42 = pos.Y;
54 |     rot.M43 = pos.Z;
55 |     return rot;
56 !   }
57 ┼   set {
58 |     Vector3 pos = new Vector3(value.M41, value.M42, value.M43);
59 |     Matrix rot = BasicGeometry.Orthogonalize(value);
60 |     Position = pos;
61 |     Orientation = rot;
62 !   }
63 ! }
```

Position プロパティは，ODE の剛体位置取得，設定APIを隠匿します．XNA の Vector3 型を入出力データとして，Ode.BodyCopyPosition／Ode.BodySetPosition を呼び出すことで剛体の位置を取得／設定します．Orientation プロパティも同様に，XNA の Matrix 型を入出力として，剛体の回転量を取得／設定します．位置と回転を含めたトランスフォームの取得／設定は，Matrix 型の Coordinate プロパティを通じて行います．

次に，コンストラクタと Dispose メソッドを示します．PhysicalPrimitive クラスコンストラクタは，全てのプリミティブに共通のフィールドのみを初期化します．一方，body と geom フィールドは各プリミティブに依存して設定する必要がありますので，継承クラスでそれぞれ初期化／設定することになります．

```
 1   public PhysicalPrimitive(float weight, Vector3 scale, Model model)
 2 ┼ {
 3 |   this.weight = weight;
 4 |   this.scale = scale;
 5 |   this.model = model;
 6 ! }
 7
 8   public virtual void Dispose()
 9 ┼ {
10 |   if (body != BodyID.Zero)
11 |     Ode.BodyDestroy(body);
12 |   body = BodyID.Zero;
13 |
14 |   if (geom != GeomID.Zero)
15 |     Ode.GeomDestroy(geom);
16 |   geom = GeomID.Zero;
17 ! }
```

球プリミティブ [†]

球プリミティブ PhysicalSphere クラスは密度が一様な球を定義します．密度分布が一様なので，慣性テンソルなどの全ての物理特性は，球の半径と質量

から一意に決定します. そのため, フィールドには球半径 radius のみが追加されます. Radius プロパティを除いた PhysicalSphere クラスの定義を以下に示します.

```
1  class PhysicalSphere : PhysicalPrimitive
2  {
3    public PhysicalSphere(float weight, Model model, float radius)
4      : base(weight, new Vector3(radius, radius, radius), model)
5    {
6      private readonly float radius = 0; // 球の半径 [m]
7
8      public PhysicalSphere(float weight, Model model, float radius)
9        : base(weight, new Vector3(radius, radius, radius), model)
10     {
11       this.radius = radius;
12
13       Ode.Mass mass;
14       Ode.MassSetZero(out mass);
15       Ode.MassSetSphereTotal(out mass, weight, Radius);
16
17       body = Ode.BodyCreate(Physics.World);
18       Ode.BodySetMass(body, ref mass);
19
20       geom = Ode.CreateSphere(Physics.Space, radius);
21       Ode.GeomSetBody(geom, body);
22     }
23   }
24 }
```

Ode.MassSetSphereTotal 関数によって, 密度一様な球の慣性テンソルを計算し, Ode.CreateSphere 関数によって球ジオメトリを生成します. なお, XNA 表示用モデルには, 半径 1.0 の単位球モデルが渡されることが期待されます.

直方体プリミティブ

PhysicalBox クラスは, 密度一様な直方体を定義します. PhysicalSphere のように, 直方体各辺の長さと質量が与えられると, 直方体の物理特性が一意に決定します. Size プロパティを除いた PhysicalBox クラスの定義を示します.

```
1  class PhysicalBox : PhysicalPrimitive
2  {
3    public PhysicalBox(float weight, Model model, Vector3 size)
4      : base(weight, size, model)
5    {
6      private readonly Vector3 size = Vector3.Zero; // 直方体の各辺の長さ [m]
7
8      public PhysicalBox(float weight, Model model, Vector3 size)
9        : base(weight, size, model)
10     {
11       this.size = size;
12
13       Ode.Mass mass;
14       Ode.MassSetZero(out mass);
15       Ode.MassSetBoxTotal(out mass, weight, size.X, size.Y, size.Z);
16
17       body = Ode.BodyCreate(Physics.World);
18       Ode.BodySetMass(body, ref mass);
19
20       geom = Ode.CreateBox(Physics.Space, size.X, size.Y, size.Z);
21       Ode.GeomSetBody(geom, body);
22     }
23   }
24 }
```

Ode.MassSetBoxTotal 関数によって, 密度一様な直方体の慣性テンソルを計算し, Ode.CreateBox 関数によって直方体ジオメトリを生成します. なお, XNA 表示用モデルには, 各辺長さ 1.0 の単位立方体モデルが渡されることが期待されます.

円筒プリミティブ

PhysicalCylinder クラスは, 密度一様な円筒を定義します. こちらも円筒長さと円半径, 質量が与えられると, 円筒の物理特性が一意に決定します. Length, Radius プロパティを除いた PhysicalCylinder クラスの定義を示します.

```
1  class PhysicalCylinder : PhysicalPrimitive
2  {
3    public PhysicalCylinder(float weight, Model model, float length, float radius)
4      : base(weight, new Vector3(radius, radius, length), model)
5    {
6      private readonly float length; // 長さ [m]
7      private readonly float radius; // 半径 [m]
8
9      public PhysicalCylinder(float weight, Model model, float length, float radius)
10       : base(weight, new Vector3(radius, radius, length), model)
11     {
```

```
12 |        this.length = length;
13 |        this.radius = radius;
14 |
15 |        Ode.Mass mass;
16 |        Ode.MassSetCylinderTotal(out mass, weight, 3, radius, length);
17 |
18 |        body = Ode.BodyCreate(Physics.World);
19 |        Ode.BodySetMass(body, ref mass);
20 |
21 |        geom = Ode.CreateCylinder(Physics.Space, radius, length);
22 |        Ode.GeomSetBody(geom, body);
23 !      }
24 !    }
25 ! }
```

Ode.MassSetCylinderTotal 関数によって，密度一様な直方体の慣性テンソルを計算し，Ode.CreateCylinder 関数によって直方体ジオメトリを生成します．なお，XNA 表示用モデルには，円筒長 1.0，半径 1.0 の単位円筒モデルが渡されることが期待されます．

地面プリミティブ

特殊なプリミティブとして，地面のジオメトリを扱うプリミティブクラスを定義します．こちらは PhysicalPrimitive を継承しないクラスで，ODE のボディや XNA の表示モデルを扱わず，フィールドには地面の法線と ODE のジオメトリのみを持ちます．そのため，コンストラクタでは Ode.CreatePlane を用いて指定した法線を持つ平面ジオメトリを生成しています．

```
1   class PhysicalGround : IDisposable
2  {
3  |   private readonly Vector4 planeParam = Vector4.Zero; // 平面パラメータ (a, b, c, d), ax + by + cz + d = 0
4  |   protected GeomID geom = GeomID.Zero;                // ODE - 地面ジオメトリ
5  |
6  |   public PhysicalGround(Vector4 planeParam)
7  |   {
8  |     this.planeParam = planeParam;
9  |     geom = Ode.CreatePlane(Physics.Space, planeParam.X, planeParam.Y, planeParam.Z, planeParam.W);
10 !   }
11 |
12 |   public virtual void Dispose()
13 |   {
14 |     if (geom != GeomID.Zero)
15 |       Ode.GeomDestroy(geom);
16 |     geom = GeomID.Zero;
17 !   }
18 ! }
```

多関節リンククラス

基本プリミティブをボーンとし，それらを関節によって接続するモデルを作成します．キャラクタのスケルトン構造で解説したように，キャラクタのボーンは 1 つの親関節と複数の子関節を持つような，階層的なツリー構造を構成します．今回作成した ArticulateBodyNode クラスも単一の親ボーンを参照し，かつ複数の子ボーンリストを持つような，キャラクタモデルの階層ツリー構造における各ノードを表します．したがって，クラスはツリー操作に関するフィールドと，キャラクタの関節やボーンに関するフィールドを持つことになります（本来であればこれらは別クラスに分解すべきでしょうが，面倒だったので...）．ArticulatedBodyNode クラスの宣言とフィールドを以下に示します．

```
1   public class ArticulatedBodyNode : IDisposable
2  {
3  |   private PhysicalPrimitive bone = null;                                  // ボーン
4  |   private ArticulatedBodyNode parent = null;                              // 親ノード
5  |   private List children = new List(); // 子ノード
6  |   private JointID joint = JointID.Zero;                                    // 関節（親ノードへ接続）
7  |   private Ode.JointType jointType = Ode.JointType.None;                    // 関節の種類
```

bone フィールドはボーンの形状や物理特性を保持する基本プリミティブです．parent と children がツリー構造に関するフィールドで，それぞれ親ノード＝親ボーンと子ノード＝子ボーンリストを表します．一方，joint は親ボーンとの間の関節，jointType はその関節の種類（後述）を表すような ODE の変数型です．

次に，ArticulatedBodyNode クラスのコンストラクタと Dispose メソッドを示します．

```
1   public ArticulatedBodyNode(PhysicalPrimitive bone)
2  {
3  |   this.bone = bone;
4  ! }
5
6   public virtual void Dispose()
7  {
8  |   for (int i = 0; i < NumChildren; ++i)
9  |     children[i].Dispose();
10 |   children.Clear();
11 |
12 |   if (joint != JointID.Zero)
13 |     Ode.JointDestroy(joint);
14 |   joint = JointID.Zero;
```

```
15 |
16 |    if (bone != null)
17 |      bone.Dispose();
18 |    bone = null;
19 |
20 |    jointType = Ode.JointType.None;
21 |    parent = null;
22 ! }
```

コンストラクタはボーンのプリミティブのみを引数にとります．つまり，デフォルトではどの関節にも接続されないボーンが生成されます．それらを以下のメソッドによって関節接続することで，任意の多関節・多リンク構造体を構築します．

- 球ジョイント
  - 球ジョイント(またはボール・ソケット:ball-and-socket関節)はあらゆる方向への回転やひねりを許容する関節です．物理的なイメージは ODE 公式ドキュメントの球ジョイントを参照してください．なお，球ジョイントには関節可動域を設定できません．
  - このメソッドは，親ボーンへの参照 parent と，グローバル座標系における関節の中心座標 anchor を引数にとります．

```
 1   public virtual void ConnectByBall(ArticulatedBodyNode parent, Vector3 anchor)
 2 + {
 3 |   if (joint != JointID.Zero || this.parent != null)
 4 |     throw new ApplicationException();
 5 |
 6 |   joint = Ode.JointCreateBall(Physics.World, JointGroupID.Zero);
 7 |   Ode.JointAttach(joint, parent.Bone.Body, bone.Body);
 8 |   Ode.JointSetBallAnchor(joint, anchor.X, anchor.Y, anchor.Z);
 9 |
10 |   jointType = Ode.JointType.Ball;
11 |
12 |   parent.AddChild(this);
13 ! }
```

- ヒンジジョイント
  - ヒンジジョイント(Hinge 関節)は，1 つの回転軸を持つ関節です．物理的なイメージは ODE 公式ドキュメントのヒンジジョイントを参照してください．ヒンジジョイントには回転軸周りの回転量に関する関節可動域を設定できます．
  - メソッドは親ボーンへの参照 parent，グローバル座標系における関節の中心座標 anchor，回転軸ベクトル axis，そして可動域の下限と上限値 limit(= float[2] {下限値 rad，上限値 rad})を引数にとります．

```
 1   public virtual void ConnectByHinge(ArticulatedBodyNode parent, Vector3 anchor, Vector3 axis, float[] limit)
 2 + {
 3 |   if (joint != JointID.Zero || this.parent != null)
 4 |     throw new ApplicationException();
 5 |
 6 |   joint = Ode.JointCreateHinge(Physics.World, JointGroupID.Zero);
 7 |   Ode.JointAttach(joint, parent.Bone.Body, bone.Body);
 8 |   Ode.JointSetHingeAnchor(joint, anchor.X, anchor.Y, anchor.Z);
 9 |   Ode.JointSetHingeAxis(joint, axis.X, axis.Y, axis.Z);
10 |
11 |   Ode.JointSetHingeParam(joint, (int)Ode.JointParam.LoStop, limit[0]);
12 |   Ode.JointSetHingeParam(joint, (int)Ode.JointParam.HiStop, limit[1]);
13 |
14 |   jointType = Ode.JointType.Hinge;
15 |   parent.AddChild(this);
16 ! }
```

- ユニバーサルジョイント
  - ユニバーサルジョイント(Universal 関節)は，2 つの回転軸(一般的には 2 軸は直交)を持つ関節です．物理的なイメージは ODE 公式ドキュメントのユニバーサルジョイントを参照してください．ユニバーサルジョイントには，それぞれの回転軸周りの回転量に関する関節可動域を設定できます．
  - メソッドは親ボーンへの参照 parent，グローバル座標系における関節中心座標 anchor，1つめ(親ボーン側)の回転軸ベクトル axis1 と可動域 limit1，そして 2 つめ(子ボーン側)の回転軸ベクトル axis2 と可動域 limit2 を引数にとります((limit1/limit2 = float[2] {下限値 rad，上限値 rad}))を引数にとります．

```
 1   public virtual void ConnectByUniversal(ArticulatedBodyNode parent, Vector3 anchor, Vector3 axis1, float[] limit1,
 2                                                                                     Vector3 axis2, float[] limit2)
 3 + {
 4 |   if (joint != JointID.Zero || this.parent != null)
 5 |     throw new ApplicationException();
 6 |   if (limit1.Length < 2 || limit2.Length < 2)
 7 |     throw new ApplicationException();
 8 |
 9 |   joint = Ode.JointCreateUniversal(Physics.World, JointGroupID.Zero);
10 |   Ode.JointAttach(joint, parent.Bone.Body, bone.Body);
11 |   Ode.JointSetUniversalAnchor(joint, anchor.X, anchor.Y, anchor.Z);
12 |
13 |   Ode.JointSetUniversalAxis1(joint, axis1.X, axis1.Y, axis1.Z);
14 |   Ode.JointSetUniversalAxis2(joint, axis2.X, axis2.Y, axis2.Z);
15 |
16 |   Ode.JointSetUniversalParam(joint, (int)Ode.JointParam.LoStop, limit1[0]);
17 |   Ode.JointSetUniversalParam(joint, (int)Ode.JointParam.HiStop, limit1[1]);
18 |   Ode.JointSetUniversalParam(joint, (int)Ode.JointParam.LoStop2, limit2[0]);
19 |   Ode.JointSetUniversalParam(joint, (int)Ode.JointParam.HiStop2, limit2[1]);
```

```
20 |
21 |    jointType = Ode.JointType.Universal;
22 |    parent.AddChild(this);
23 ! }
```

- 固定ジョイント
  - 固定ジョイントは，その名の通り 2 つのボーンを固定して接続するジョイントです．
  - メソッドは親ボーンへの参照のみを引数にとります．

```
 1   public virtual void ConnectByFix(ArticulatedBodyNode parent)
 2 { 
 3 |   if (joint != JointID.Zero || this.parent != null)
 4 |     throw new ApplicationException();
 5 |
 6 |   joint = Ode.JointCreateFixed(Physics.World, JointGroupID.Zero);
 7 |   Ode.JointAttach(joint, parent.Bone.Body, bone.Body);
 8 |   Ode.JointSetFixed(joint);
 9 |
10 |   jointType = Ode.JointType.Fixed;
11 |   parent.AddChild(this);
12 ! }
```

基本クラスを利用したメインプログラムの作成

仮想物理空間の構築と破棄

Physics クラスのメソッドを呼び出すことで，仮想物理空間を構築，破棄します．まず，構築メソッドはグラフィクスデバイスの初期化状態に依らないので，XNA フレームワークの Initialize メソッドに記述しました．一方，破棄メソッドは UnloadContent メソッド内に記述しています．Initialize メソッドとの対応関係を考えると OnExiting メソッドに記述するべきかもしれませんが，デフォルトで用意されるメソッドの利用を優先しました．

```
 1   protected override void Initialize()
 2 { 
 3 |   Physics.Create(new Vector3(0, -9.8e2f, 0));
 4 |   base.Initialize();
 5 ! }
 6
 7   protected override void UnloadContent()
 8 { 
 9 |     Physics.Destroy();
10 ! }
```

キャラクタモデルの作成

基本プリミティブで構成されたボーンを任意の関節で接続することで，キャラクタの全身モデルを構築します．ArticulateBodyNode クラスは，最初にボーンを配置した後に関節を設定，という処理手順を想定していますので，まず各ボーンを空間に配置し，隣接するボーンを関節で接続するという流れになります．これらのボーンや関節の位置は全てグローバル座標系における絶対的な値を指定する必要がありますので，あらかじめ初期姿勢における全ての座標を与えなければなりません．

今回作成したキャラクタの設計を Fig.1 に示します．赤丸が各ボーンの中心座標，青丸が関節の中心座標を示します．

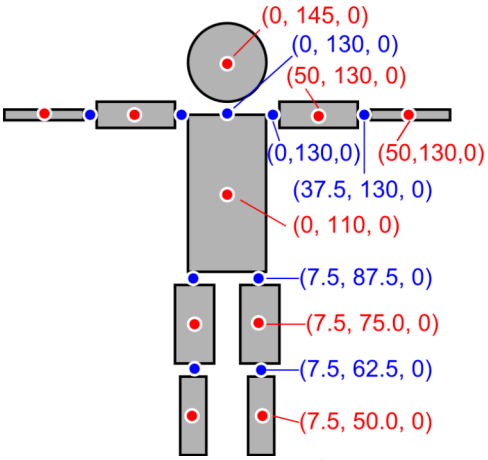

Fig.1 キャラクタモデル

これに対応するモデル作成メソッド CreateModel のコードを以下に示します．なお，今回は簡単化のために胴体や肩，股関節は 2 自由度(ユニバーサルジョイント)，ひじやひざは 1 自由度(ヒンジジョイント)に設定し，関節可動域は適当に与えています．

```
 1   protected void CreateModel()
 2 { 
```

```
    ground = new PhysicalGround(new Vector4(0, 1.0f, 0, 0));

    PhysicalBox boxBody = null;
    PhysicalSphere sphereBody = null;
    ArticulatedBodyNode parent = null, child = null;

    // 胴体
    boxBody = new PhysicalBox(10.0f, cubeModel, new Vector3(20.0f, 40.0f, 10.0f));
    boxBody.Position = new Vector3(0, 110.0f, 0);
    ragdollRoot = new ArticulatedBodyNode(boxBody);

    // 頭
    sphereBody = new PhysicalSphere(5.0f, sphereModel, 10.0f);
    sphereBody.Position = new Vector3(0, 145.0f, 0);
    child = new ArticulatedBodyNode(sphereBody);
    child.ConnectByUniversal(ragdollRoot, new Vector3(0, 132.5f, 0),
            Vector3.UnitX, new float[2] { -MathHelper.Pi / 3.0f, MathHelper.Pi / 3.0f },
            Vector3.UnitZ, new float[2] { -MathHelper.Pi / 3.0f, MathHelper.Pi / 3.0f });

    // 左上腕
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(20.0f, 5.0f, 5.0f));
    boxBody.Position = new Vector3(25.0f, 130.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByUniversal(ragdollRoot, new Vector3(12.5f, 130.0f, 0),
            Vector3.UnitY, new float[2] { -MathHelper.Pi / 4.0f, MathHelper.Pi * 0.7f },
            Vector3.UnitZ, new float[2] { -MathHelper.Pi * 0.5f, MathHelper.Pi * 0.5f });
    parent = child;

    // 左前腕
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(20.0f, 4.0f, 4.0f));
    boxBody.Position = new Vector3(50.0f, 130.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByHinge(parent, new Vector3(37.5f, 130.0f, 0),
            Vector3.UnitY, new float[2] { 0, MathHelper.Pi * 0.7f});

    // 右上腕
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(20.0f, 5.0f, 5.0f));
    boxBody.Position = new Vector3(-25.0f, 130.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByUniversal(ragdollRoot, new Vector3(-12.5f, 130.0f, 0),
            Vector3.UnitY, new float[2] { -MathHelper.Pi * 0.7f, MathHelper.Pi / 4.0f },
            Vector3.UnitZ, new float[2] { -MathHelper.Pi * 0.5f, MathHelper.Pi * 0.5f });
    parent = child;

    // 右前腕
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(20.0f, 4.0f, 4.0f));
    boxBody.Position = new Vector3(-50.0f, 130.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByHinge(parent, new Vector3(-37.5f, 130.0f, 0),
            Vector3.UnitY, new float[2] { -MathHelper.Pi * 0.7f, 0 });

    // 左大腿
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(10.0f, 20.0f, 10.0f));
    boxBody.Position = new Vector3(7.5f, 75.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByUniversal(ragdollRoot, new Vector3(7.5f, 87.5f, 0),
            Vector3.UnitX, new float[2]{-MathHelper.Pi * 0.1f, MathHelper.Pi * 0.8f},
            Vector3.UnitZ, new float[2]{-MathHelper.Pi * 0.1f, MathHelper.Pi * 0.4f});
    parent = child;

    // 左すね
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(8.0f, 20.0f, 8.0f));
    boxBody.Position = new Vector3(7.5f, 50.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByHinge(parent, new Vector3(7.5f, 62.5f, 0),
            Vector3.UnitX, new float[2] { -MathHelper.Pi * 0.7f, 0 });

    // 右大腿
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(10.0f, 20.0f, 10.0f));
    boxBody.Position = new Vector3(-7.5f, 75.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByUniversal(ragdollRoot, new Vector3(-7.5f, 87.5f, 0),
            Vector3.UnitX, new float[2] { -MathHelper.Pi * 0.1f, MathHelper.Pi * 0.8f },
            Vector3.UnitZ, new float[2] { -MathHelper.Pi * 0.4f, MathHelper.Pi * 0.1f });
    parent = child;

    // 右すね
    boxBody = new PhysicalBox(5.0f, cubeModel, new Vector3(8.0f, 20.0f, 8.0f));
    boxBody.Position = new Vector3(-7.5f, 50.0f, 0);
    child = new ArticulatedBodyNode(boxBody);
    child.ConnectByHinge(parent, new Vector3(-7.5f, 62.5f, 0),
            Vector3.UnitX, new float[2] { -MathHelper.Pi * 0.7f, 0 });
}
```

姿勢の更新 [†]

ODE のシミュレーションステップは，XNA の Update メソッド内で進行させます．Update メソッドは基本的に 1/60 sec 毎に XNA フレームワークより呼

び出されますが，シミュレーションは 1/100 sec ずつ進行させています．したがって，現実時間に比べるとアニメーションはゆっくりと再生されます．なお，衝突検出時の挙動は，前述の Physics クラスデフォルトのコールバックメソッドによって計算されます．

また，キーボードの十字キー入力に応じて，キャラクタに意図的に外力を加える処理も追加しました．押されたキーに従ってキャラクタのルート（胴体ボーン）に水平方向＋多少の浮力が加えられます．

```
protected override void Update(GameTime gameTime)
{
  // 押されたキーに応じて、胴体ボディに力を加える（多少の浮力も加える）
  KeyboardState keyboardState = Keyboard.GetState();
  Keys[] pressedKeys = keyboardState.GetPressedKeys();
  foreach (Keys key in pressedKeys)
  {
    switch (key)
    {
      case Keys.Up:
        Ode.BodyAddForce(ragdollRoot.Bone.Body, 0, 1.0e2f, -2.0e5f);
        break;
      case Keys.Down:
        Ode.BodyAddForce(ragdollRoot.Bone.Body, 0, 1.0e2f, 2.0e5f);
        break;
      case Keys.Right:
        Ode.BodyAddForce(ragdollRoot.Bone.Body, 2.0e5f, 1.0e2f, 0);
        break;
      case Keys.Left:
        Ode.BodyAddForce(ragdollRoot.Bone.Body, -2.0e5f, 1.0e2f, 0);
        break;
    }
  }

  Physics.Step(1.0e-2f, Physics.DefaultCollisionCallback);

  effects.Parameters["viewProj"].SetValue(camera.View * camera.Projection);
  base.Update(gameTime);
}
```

キャラクタの描画 [†]

最後に，キャラクタモデルの描画メソッドを示します．キャラクタの階層的ツリー構造の処理には再帰呼び出しが適していますが，ここでは Stack ジェネリクスを使って処理してみました．全てのボーンについて，ルートから親関節に至るトランスフォームを合成し，それによってボーンに関連づけられたモデルとその平面シャドウを描画しています．Draw モデルの実装については，サンプルプロジェクト内のソースコードを参照してください．

```
protected override void Draw(GameTime gameTime)
{
  graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

  Stack stack = new Stack();

  stack.Push(ragdollRoot);
  while (stack.Count > 0)
  {
    ArticulatedBodyNode node = stack.Pop();
    Matrix wm = node.Bone.ScaleTransform * node.Bone.Coordinate;

    effects.CurrentTechnique = effects.Techniques["PhongShader"];
    DrawModel(node.Bone.Model, wm);
    effects.CurrentTechnique = effects.Techniques["PlaneShadowShader"];
    DrawModel(node.Bone.Model, wm);

    for (int i = 0; i < node.NumChildren; ++i)
      stack.Push(node.GetChild(i));
  }

  base.Draw(gameTime);
}
```

まとめ [†]

XNA と ODE の組み合わせ利用の応用例として，簡単なラグドールシミュレーションを行うサンプルプログラムを紹介しました．ほとんど ODE の API をそのまま使っているだけですが，クラス作成例を通じてそれぞれの使用法が伝われば幸いです．
