



# コンピュータアニメーション特論

## 第10回 キャラクタアニメーション(3)

九州工業大学 情報工学研究院 尾下真樹

# 今日の内容

- 前回の復習、vecmathの復習
- 順運動学
- 人体形状変形モデル



# キャラクター・アニメーション

- CGにより表現された人体モデル(キャラクター)のアニメーションを実現するための技術
- キャラクター・アニメーションの用途
  - オフライン・アニメーション(映画など)
  - オンライン・アニメーション(ゲームなど)
    - どちらの用途でも使われる基本的な技術は同じ(データ量や詳細度が異なる)
    - 後者の用途では、インタラクティブな動作を実現するための工夫が必要になる
- 人体モデル・動作データの処理技術



# 全体の内容

- 人体モデル(骨格・姿勢・動作)の表現
- 人体モデル・動作データの作成方法
- サンプルプログラム、動作再生
- 順運動学、人体形状変形モデル
- 姿勢補間、キーフレーム動作再生、動作補間
- 動作接続・遷移、動作変形
- 逆運動学、モーションキャプチャ
- 動作生成・制御



# 今日の内容

- 前回の復習、vecmathの復習
- 順運動学
- 人体形状変形モデル





# 前回の復習

# 人体モデルの表現

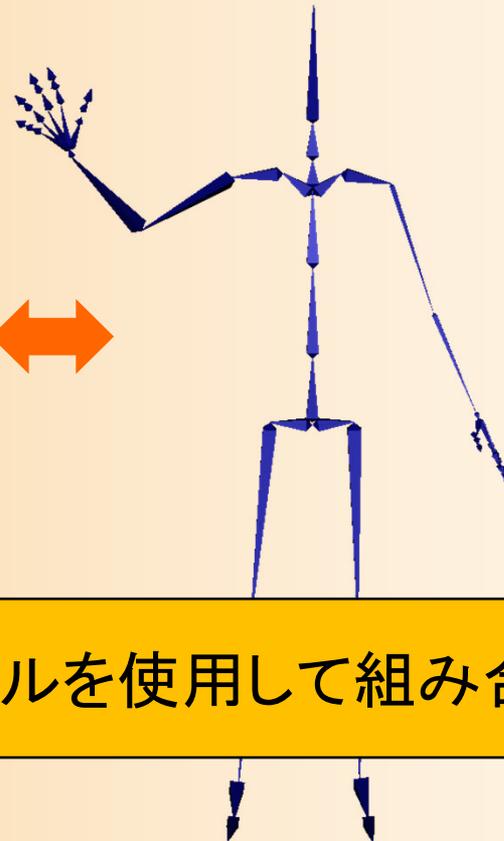
形状モデル  
(ポリゴンモデル)

骨格モデル  
(多関節体)

描画用



姿勢・動作の  
表現・処理用



形状と骨格に別のモデルを使用して組み合わせ



# 骨格モデルの表現

- 多関節体モデルによる表現

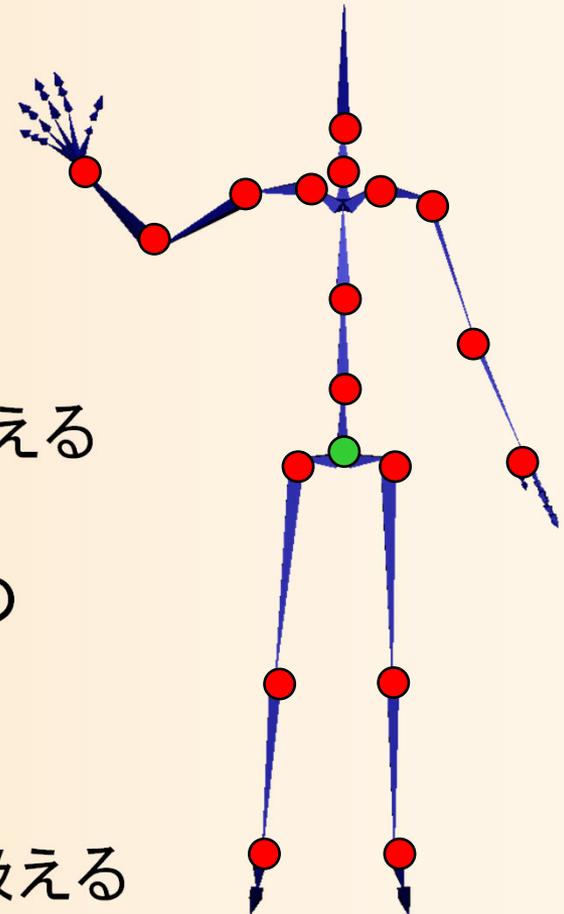
- 複数の体節(部位)が  
関節で接続されたモデル

- 体節

- 多関節体の各部位、剛体として扱える
- 複数の関節が接続されており、  
体節の長さや体節内での各関節の  
接続位置は固定

- 関節

- 2つの体節の間を接続、点として扱える
- 関節の回転により姿勢が変化する



# 骨格・姿勢の表現方法

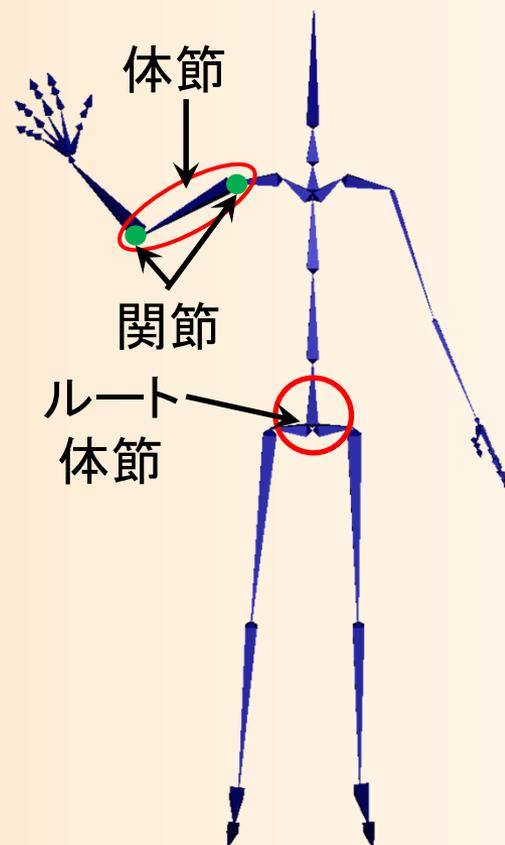
- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける

- 体節

- 複数の関節と接続
- 各関節の接続位置
  - 体節のローカル座標系

- 関節

- 2つの体節の間を接続
  - ルート側・末端側の体節



# 骨格・姿勢・動作のデータ構造

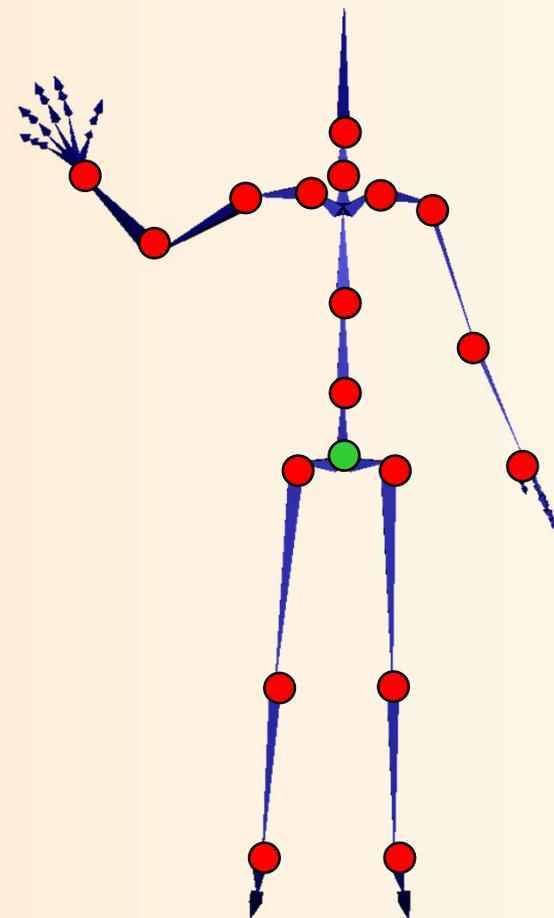
- 骨格・姿勢の構造体定義  
(SimpleHuman.h/cpp)

```
// 人体モデルの体節を表す構造体
struct Segment

// 人体モデルの関節を表す構造体
struct Joint

// 人体モデルの骨格を表すクラス
class Skeleton

// 人体モデルの姿勢を表すクラス
class Posture
```



# 骨格モデルのデータ構造(1)

## • 体節のデータ構造

```
// 人体モデルの体節を表す構造体
```

```
Struct Segment
```

```
{
```

```
    // 体節番号・名前
```

```
    int         index;
```

```
    string      name;
```

```
    // 体節の接続関節数
```

```
    int         num_joints;
```

```
    // 接続関節の配列 [接続関節番号]
```

```
    Joint **    joints;
```

```
    // 各接続関節の接続位置の配列(体節のローカル座標系)[接続関節番号]
```

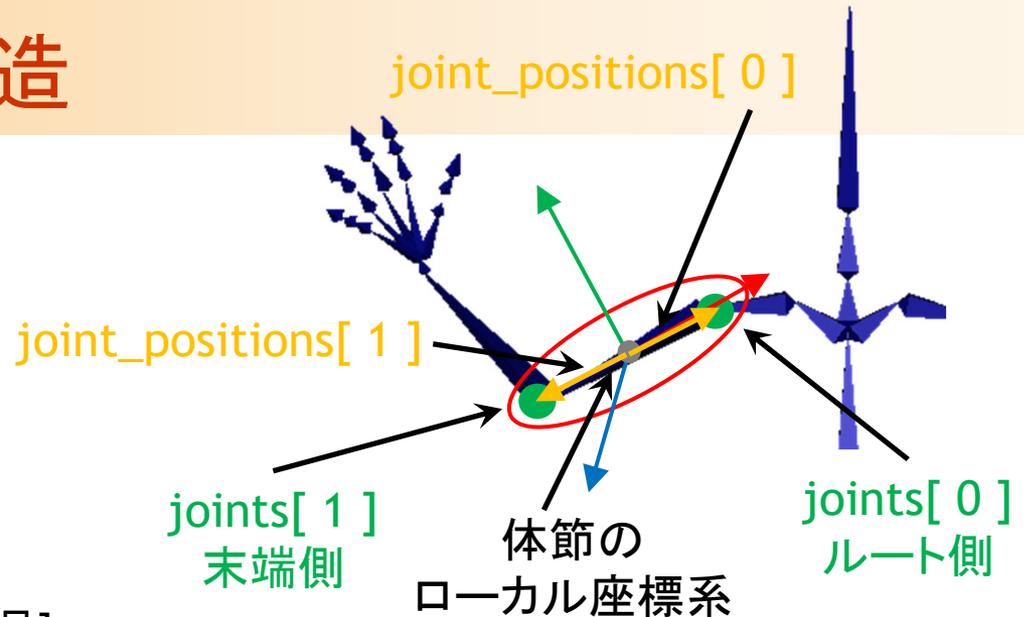
```
    Point3f *   joint_positions;
```

```
    // 体節の末端位置
```

```
    bool        has_site;
```

```
    Point3f     site_position;
```

```
};
```



複数の関節と接続  
ルート体節以外は、0番目の  
関節が、ルート側の関節とする

# 骨格モデルのデータ構造(2)

- 関節のデータ構造

```
// 人体モデルの関節を表す構造体
struct Joint
{
    // 関節番号・名前
    int         index;
    string      name;

    // 接続体節
    Segment *   segments[ 2 ];
};
```



2つの体節の間を接続  
0番目の体節が、ルート側の  
体節とする

# 骨格モデルのデータ構造(3)

- 骨格データ構造

```
// 人体モデルの骨格を表すクラス
struct Skeleton
{
    // 関節数
    int          num_segments;
    // 関節の配列 [関節番号]
    Segment **  segments;

    // 体節数
    int          num_joints;
    // 体節の配列 [体節番号]
    Joint **     joints;

    Skeleton( int s, int j );
    ~Skeleton();
};
```



# 姿勢のデータ構造

- 姿勢のデータ構造

```
// 人体モデルの姿勢を表すクラス
class Posture
{
public:
    const Skeleton * body;
    Point3f    root_pos;        // ルートの位置
    Matrix3f   root_ori;       // ルートの向き(回転行列表現)
    Matrix3f * joint_rotations; // 各関節の相対回転(回転行列表現)
                                   // [関節番号] ※ 関節数分の配列

public:
    // コンストラクタ
    Posture( Skeleton * b );
    // 初期化
    void Init( Skeleton * b );
};
```



# 動作のデータ構造

## • 動作のデータ構造

```
// 人体モデルの動作を表すクラス
class Motion
{
    // 骨格モデル
    const Skeleton * body;
    // フレーム数
    int num_frames;
    // フレーム間の時間間隔
    float interval;
    // 全フレームの姿勢 [フレーム番号]
    Posture * frames;

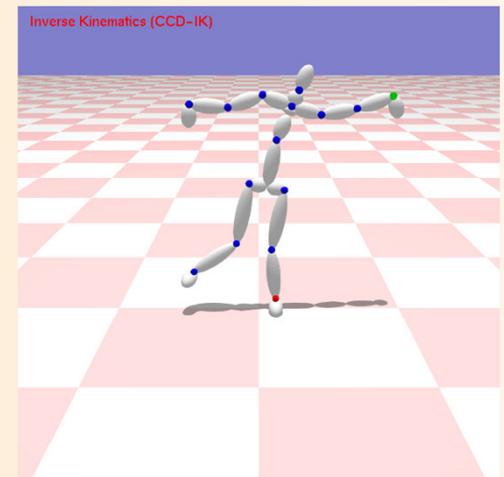
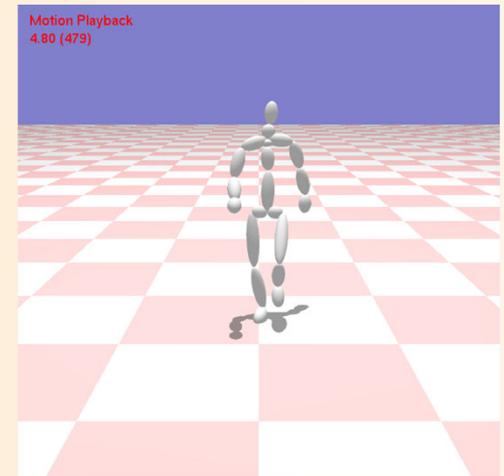
    // 姿勢を取得
    void GetPosture( float time, Posture & p ) const;
};
```

時刻を入力として、  
その時刻の姿勢を出力



# デモプログラム

- 複数のアプリケーションを含む
  - マウスの中ボタン or m キーで切り替え
- 動作再生
- キーフレーム動作再生
- 順運動学計算
- 姿勢補間
- 動作補間(2つの動作の補間)
- 動作接続・遷移
- 動作変形
- 逆運動学計算(CCD-IK)
- 上記以外の応用アプリケーションも含む



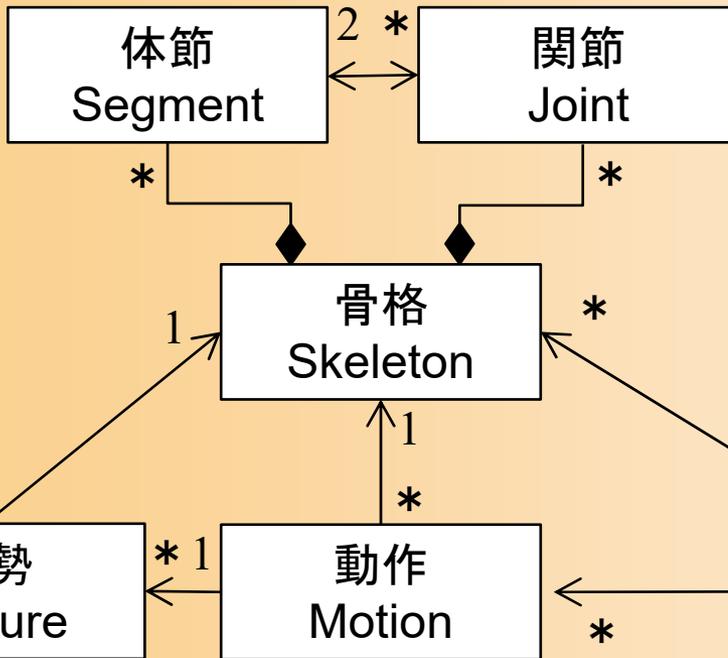
# サンプルプログラム

- デモプログラムの一部のサンプルプログラム
  - 骨格・姿勢・動作のデータ構造定義 (SimpleHumn.h/cpp)
  - BVH動作クラス (BVH.h/cpp)
  - アプリケーションの基底クラスとGLUTコールバック関数 (SimpleHumanGLUT.h/cpp)
    - アプリケーションの基底クラス GLUTBaseAppの定義・実装
      - 各イベント処理のためのメソッドの定義を含む
      - 本クラスを派生させて各アプリケーションクラスを定義
    - 複数のアプリケーションの管理と、現在のアプリケーションのイベント処理を呼び出すGLUTコールバック関数
  - メイン処理 (SimpleHumanMain.cpp)
  - 各アプリケーションの定義・実装 (???App.h/.cpp)
    - 主要な処理を各自で実装(レポート課題)



# クラス図

## クラス・構造体間の関係



グローバル関数の集まりで構成されるので、クラスではないが、ここでは一つのクラスと同様に記述

フレームワーク  
GLUTFramework

基底アプリ  
GLUTBase

基底アプリの  
集合として管理  
派生クラスの  
実装は意識しない

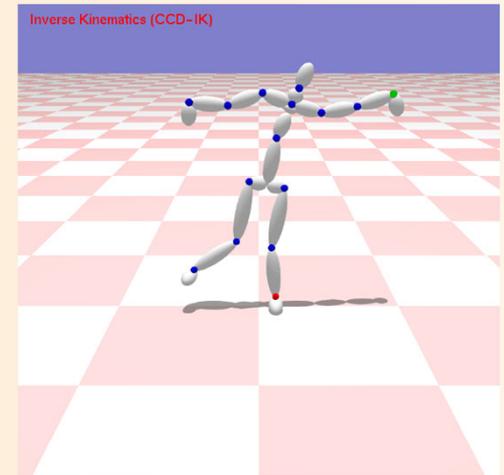
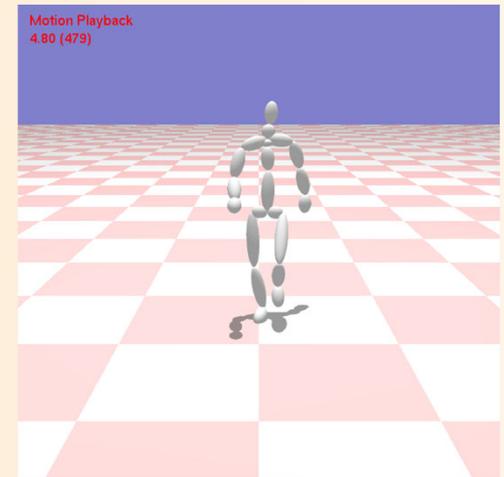
継承

各デモのアプリ  
???App



# サンプルプログラム

- 複数のアプリケーションを含む
  - マウスの中ボタン or m キーで切り替え
- 動作再生
- キーフレーム動作再生
- 順運動学計算
- 姿勢補間
- 動作補間(2つの動作の補間)
- 動作接続・遷移
- 動作変形
- 逆運動学計算(CCD-IK)





# vecmathの復習

# vecmath

- vecmath C++版

- <http://www.objectclub.jp/download/vecmath1>

- テンプレートライブラリ

- Vector3f, Vector3d など、float と double 両方に対応
- リンクの必要がなく、インクルードするだけで使える

- 点 (Point3) とベクトル (Vector3) の使い分け

- 変換行列をかけると、Point3 には平行移動も適用されるが、Vector3 は回転のみ適用される (オーバーロードの機能により、型によって判断)

- 一通りの機能があり便利



# vecmathの利用方法(1)

- サンプルプログラムでは、vecmath を使用
- 主要な使用クラス(float型の例)
  - Point3f …… 点(3次元ベクトル)
  - Vector3f …… ベクトル(3次元ベクトル)
  - Matrix3f …… 回転行列(3×3行列)
  - Matrix4f …… 座標変換行列(4×4行列)
  - Quat4f …… 四元数(4次元ベクトル)
  - AxisAngle4f …… 回転軸+回転角度
  - Color3f, Color4f …… 色(3 or 4次元ベクトル)



# vecmathの利用方法(2)

- vecmathのインストール
  - 適当な場所にコピー、インクルードディレクトリを設定
- vecmathのヘッダファイルをインクルードして利用
- 最新の Visual Studio では、プロジェクト設定の変更が必要(開発環境の設定方法の資料を参照)
- サンプルプログラムでは、vecmathクラスを引数として OpenGL 関数を呼び出すための関数を定義している(vecmath\_gl.h)
  - 詳細はヘッダファイルの中身を参照



# vecmathの利用方法(3)

- 行列・ベクトルのメンバ変数

- 3次元ベクトル(Poin3f, Vector3f, Color3f等)は、  
x, y, z のメンバ変数を持つ

- 4次元ベクトル(Quat4f, Color4f等)は、x, y, z, w  
のメンバ変数を持つ

- 3×3行列は、m00, m01, ...,  $\begin{pmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{pmatrix}$ ,  
m22 のメンバ変数を持つ

- 4×4行列は、m00, m01, ...,  $\begin{pmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{pmatrix}$ ,  
m33 のメンバ変数を持つ



# vecmathの利用方法(4)

- 行列・ベクトルへの値の取得・設定・変換

- set()、get()メソッドにより、値の取得・設定・変換ができる

```
Vector3f v; // 3次元ベクトル
Matrix3f r; // 3×3行列(回転行列)
Matrix4f m; // 4×4行列(回転・移動行列)
Quat4f q; // 四元数ベクトル

v.set( 1.0f, 2.0f, 3.0f ); // v に (1.0, 2.0, 3.0) の値を設定
m.get( &v ); // m の並行移動成分(3次元ベクトル)を取得
m.get( &r ); // m の回転成分(3×3行列)を取得

r.rotY( 0.25 * M_PI ); // r に Y軸周りに 1/4 π 回転する回転行列を設定
q.set( m ); // 回転行列 r を四元数 q に変換

m.set( 1.0f,0.0f,0.0f,0.0f, 0.0f,1.0f,0.0f,0.0f, 0.0f,0.0f,1.0f,0.0f,
0.0f,0.0f,0.0f,1.0f ); // m に単位行列を設定
m.setIdentity(); // m に単位行列を設定
```



# vecmathの利用方法(5)

## • 行列・ベクトルの演算

- add()、sub()、mult()メソッドなどにより、演算が行える
- + - \* などの演算子も利用可能(処理速度は遅くなる)
- 同一の型の変数同士以外、これらの2項演算は適用不可

```
Vector3f v, v1, v2; Matrix3f m, m1, m2; float s;
```

```
v.add( v1, v2 ); // v1 と v2 の和を v に代入
```

```
v = v1 + v2; // 同じく、v1 と v2 の和を v に代入
```

```
v = v1.cross( v2 ); // v1 と v2 の外積(ベクトル)を v に代入
```

```
s = v1.dot( v2 ); // v1 と v2 の内積(スカラー)を s に代入
```

```
v.scaleAdd( 0.5f, v1, v2 ); // v1 の 0.5 倍を v2 に加えたものを、v に代入
```

```
m.mul( m1, m2 ); // m1 と m2 の積を m に代入  $M = M_1 M_2$ 
```

```
m = m1 * m2; // 同じく、m1 と m2 の積を m に代入  $M = M_1 M_2^t$ 
```

```
m.mulTransposeRight( m1, m2 ) // m1 と m2 の転置の積を m に代入
```

```
m.invert( m1 ); // m1 の逆行列を m に代入  $M = M_1^{-1}$ 
```



# vecmathの利用方法(6)

## • 行列・ベクトルの演算(続き)

- 行列クラスの transform() メソッドにより、行列とベクトルの掛け算(座標変換の適用)を行える
  - Matrix4f(回転+移動)と Matrix3f(回転) のどちらを適用するかや、Point3f(座標)と Vector3f(ベクトル)のどちらに対して適用するかによって、型に応じた適切な計算が行われる

```
Point3f p; // 3次元位置
Vector3f v; // 3次元ベクトル
Matrix3f r; // 3×3行列(回転行列)
Matrix4f m; // 4×4行列(回転・移動行列)

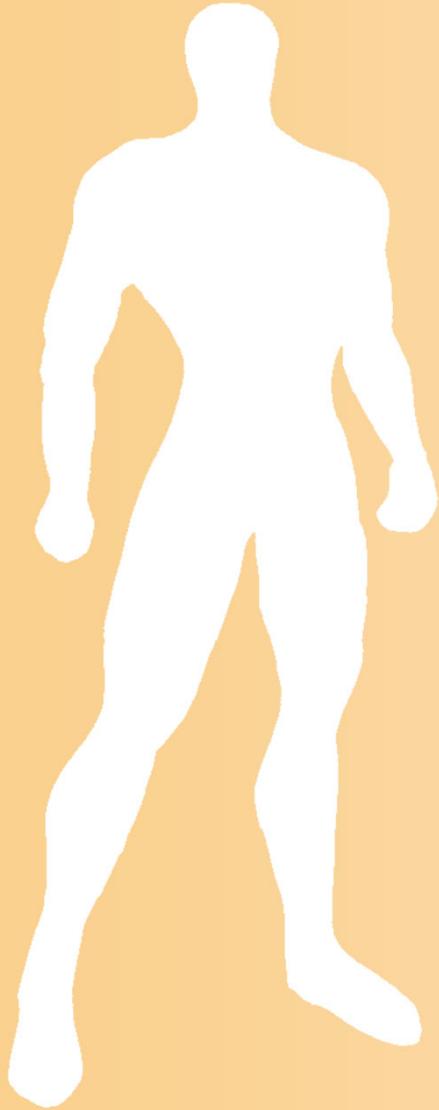
m.transform( &p ); // p に m の回転+移動成分をかけたものを p に代入
m.transform( &v ); // v に m の回転成分をかけたものを v に代入
r.transform( &v ); // v に回転行列 r をかけたものを v に代入
```



# 今日の内容

- 前回の復習、vecmathの復習
- 順運動学
- 人体形状変形モデル





# 順運動学

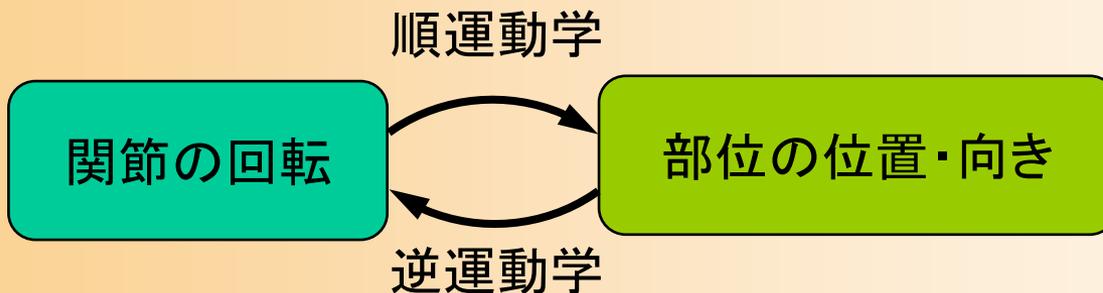
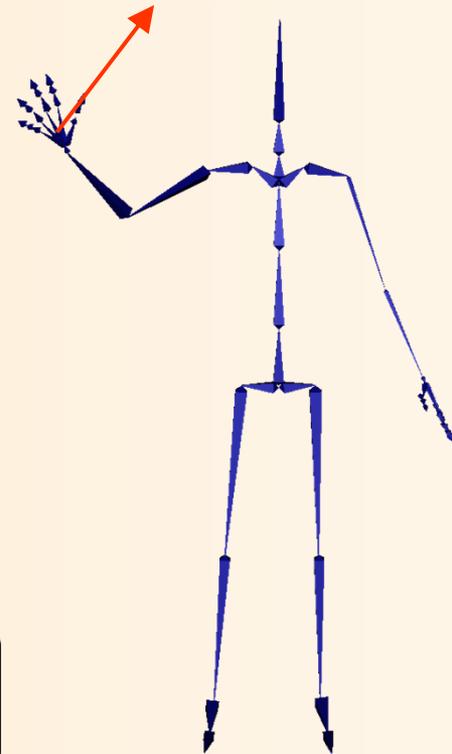
# 順運動学

- 運動学
- 順運動学
- デモプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ



# 運動学

- 運動学(キネマティクス)
  - 多関節体の姿勢表現の基礎となる考え方
  - 人間の姿勢は、全関節の回転により表現できる
  - 関節の回転と各部位の位置・向きとの関係を計算するための手法



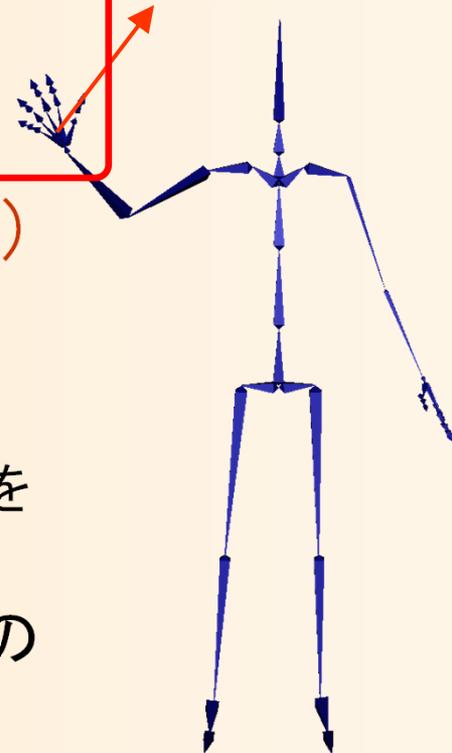
# 順運動学と逆運動学

- 順運動学 (フォワード・キネマティクス)
  - 多関節体の関節回転から、各部位の位置・向きを計算
  - 回転・移動の変換行列の積により計算
- 逆運動学 (インバース・キネマティクス)
  - 指定部位の目標の位置・向きから、多関節体の関節回転の変化を計算
    - 手先などの移動・回転量が与えられた時、それを実現するための関節回転の変化を計算する
  - 姿勢を指定する時、関節回転よりも、手先の位置・向きなどを使った方がやりやすい
  - ロボットアームの軌道計画等にも用いられる



# 順運動学と逆運動学

- 順運動学 (フォワード・キネマティクス)
  - 多関節体の関節回転から、各部位の位置・向きを計算
  - 回転・移動の変換行列の積により計算
- 逆運動学 (インバース・キネマティクス)
  - 指定部位の目標の位置・向きから、多関節体の関節回転の変化を計算
    - 手先などの移動・回転量が与えられた時、それを実現するための関節回転の変化を計算する
  - 姿勢を指定する時、関節回転よりも、手先の位置・向きなどを使った方がやりやすい
  - ロボットアームの軌道計画等にも用いられる



# 順運動学

- 運動学
- 順運動学
- サンプルプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ

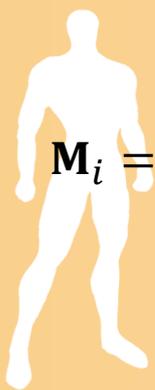


# 順運動学

- 順運動学 (フォワード・キネマティクス)

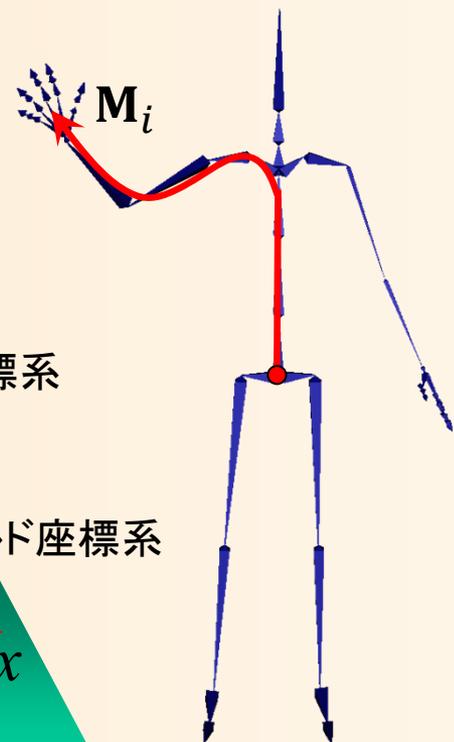
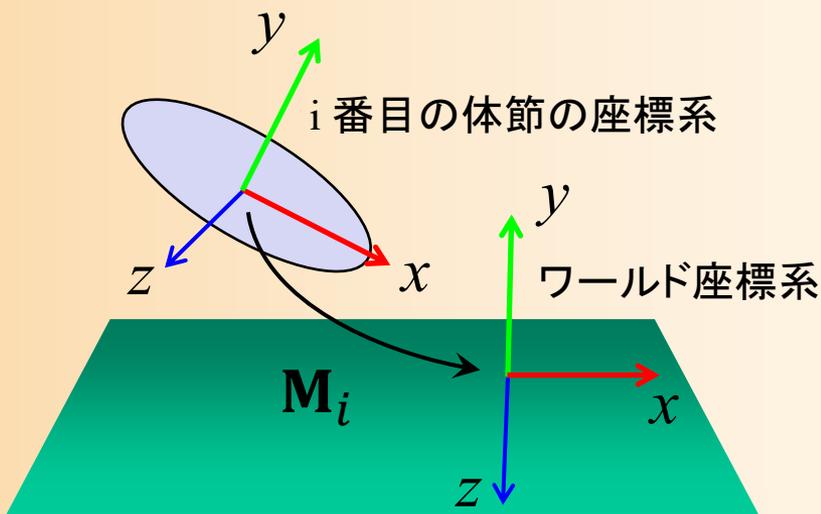
- 骨格と姿勢から、全体節 (and 関節) の位置・向きを表す変換行列を計算

- $i$  番目の体節のローカル座標系からワールド座標系への変換行列  $M_i$



$M_i =$

向き			位置
$Rx_x$	$Rx_y$	$Rx_z$	$x$
$Ry_x$	$Ry_y$	$Ry_z$	$y$
$Rz_x$	$Rz_y$	$Rz_z$	$z$
0	0	0	1



# 順運動学の計算方法

- 順運動学 (フォワード・キネマティクス)

- 全体節 (and 関節) の位置・向きを表す変換行列を計算

- $i$  番目の体節のローカル座標系からワールド座標系への変換行列  $M_i$

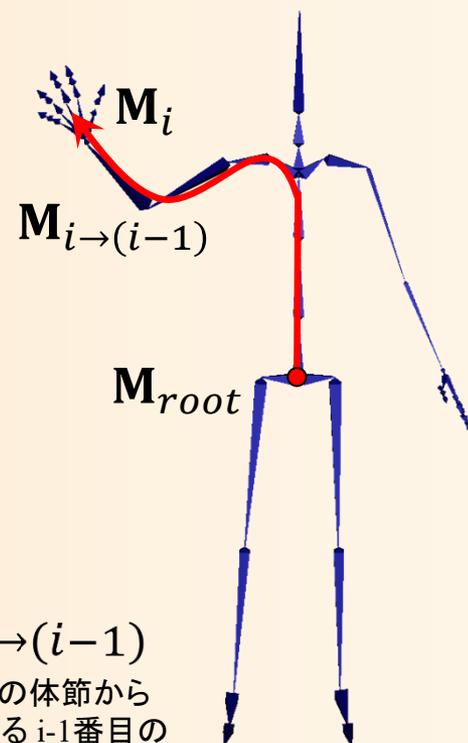
- 変換行列の計算方法

- ルート体節から  $i$  番目の体節に向かって順番に隣接する体節への変換行列をかけることで計算できる

$$M_i = M_{root} M_{1 \rightarrow root} \cdots M_{(i-1) \rightarrow (i-2)} M_{i \rightarrow (i-1)}$$

ルート体節の  
位置・向き

$i$  番目の体節から  
隣接する  $i-1$  番目の  
体節への変換行列



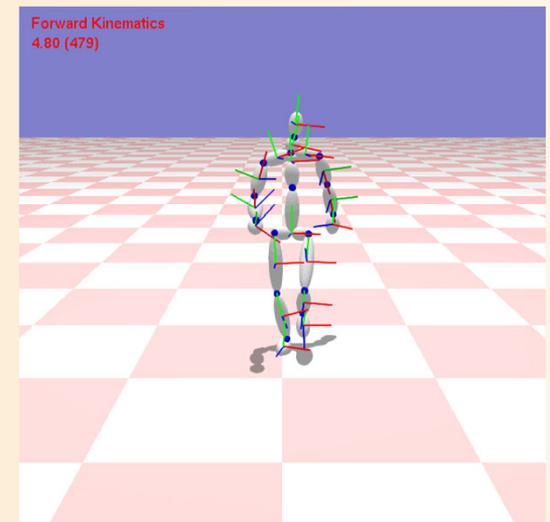
# 順運動学

- 運動学
- 順運動学
- サンプルプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ



# デモプログラム

- 順運動学計算アプリケーション
  - 動作再生中の各姿勢から順運動学計算
  - 順運動学計算
  - 各関節の位置を可視化
    - 青の球で描画
  - 各体節の位置・向きを可視化
    - 局所座標系のX軸・Y軸・Z軸方向を、赤・青・緑の線分で描画



順運動学計算  
Forward Kinematics



# 順運動学

- 運動学
- 順運動学
- サンプルプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ



# 順運動学の計算方法

- フォワード・キネマティクス(順運動学)
  - 姿勢(腰の位置・向き、全関節の回転)から、全体節・関節の位置・向きを計算
  - 繰り返し計算
    - ルートから末端に向かって繰り返し
      - 複数の子関節がある場合は各方向に分岐
      - 再帰呼び出しを使うと実装しやすい
    - 前の体節の位置・向きを表す変換行列に、
      1. 次の関節への移動(・回転)
      2. 関節の回転
      3. 次の体節への移動(・回転)を順番、に右側にかける適用



# 順運動学の計算方法

- 繰り返し計算

- ルートから末端に向かって繰り返し

- 複数の子関節がある場合は各方向に分岐

- 再帰呼び出しを使うと実装しやすい

- 複数の末端に向かっての枝分かれにも対応できる

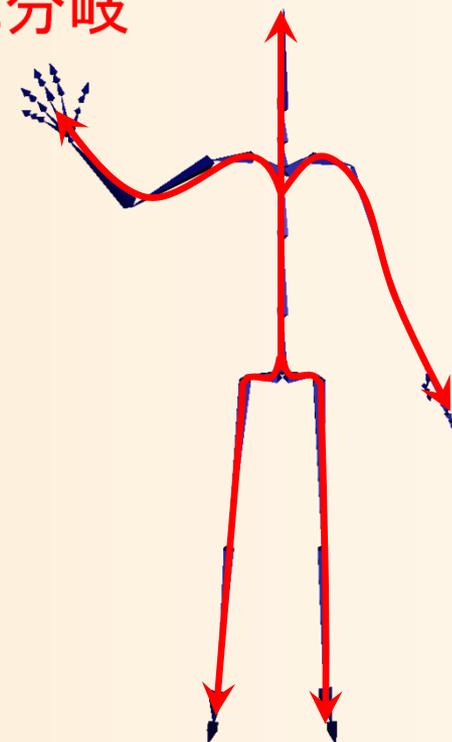
- 前の体節の位置・向きに、

1. 次の関節への移動・回転

2. 関節の回転

3. 次の体節への移動・回転

を順番に適用



# 順運動学の計算方法

- 繰り返し計算

- ルートから末端に向かって繰り返し

- 複数の子関節がある場合は各方向に分岐
- 再帰呼び出しを使うと実装しやすい

- 前の体節の位置・向きに、 $\mathbf{M}_{i-1}$

1. 次の関節への移動(・回転)

2. 関節の回転  $\mathbf{R}_j \quad \mathbf{T}_{(i-1) \rightarrow j}$

3. 次の体節への移動(・回転)

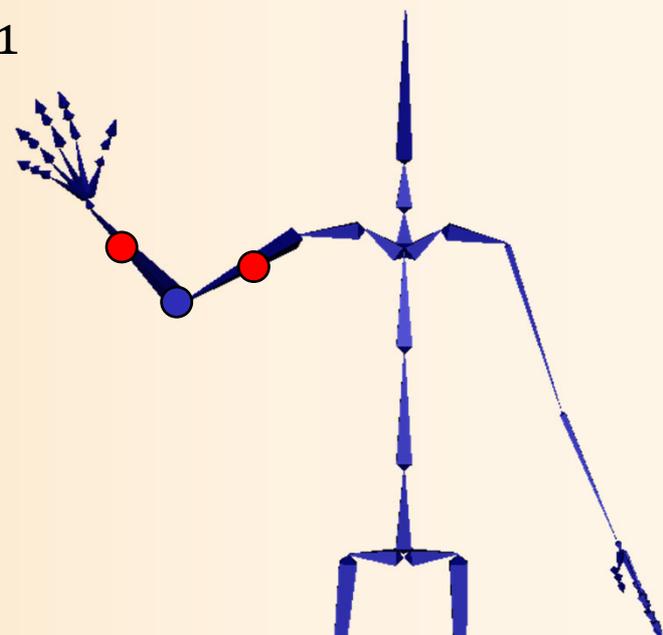
を順番に適用  $\mathbf{T}_{j \rightarrow i}$

$$\mathbf{M}_i = \mathbf{M}_{i-1} \mathbf{T}_{(i-1) \rightarrow j} \mathbf{R}_j \mathbf{T}_{j \rightarrow i}$$

①

②

③



# 順運動学の計算方法

- 繰り返し計算

- ルートから末端に向かって繰り返し

- 複数の子関節がある場合は各方向に分岐
- 再帰呼び出しを使うと実装しやすい

骨格情報  
から取得

姿勢情報  
から取得

骨格情報  
から取得

前の体節の位置・向きに、 $M_{i-1}$

1. 次の関節への移動(・回転)

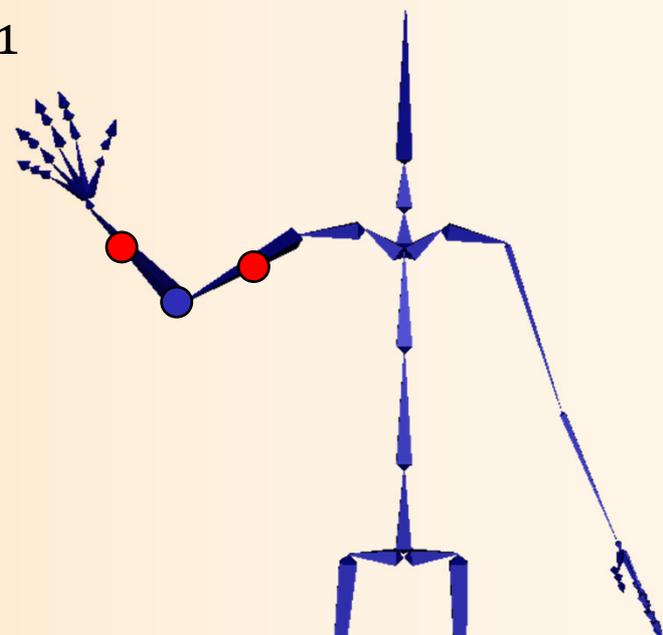
2. 関節の回転  $R_j$   $T_{(i-1) \rightarrow j}$

3. 次の体節への移動(・回転)

を順番に適用  $T_{j \rightarrow i}$

$$M_i = M_{i-1} T_{(i-1) \rightarrow j} R_j T_{j \rightarrow i}$$

①      ②      ③



# 順運動学の計算方法

- 繰り返り計算

- ルートから末端に向かって繰り返り

- 複数の子関節がある場合は各子関節に岐
    - 再帰呼び出しを使うと実装し

- 前の体節の位置・向きに、 $M_{i-1}$

- 1. 次の関節への移動(・回転)

- 2. 関節の回転  $R_j$   $T_{(i-1) \rightarrow j}$

3. 関節 → 次の体節の中心への平行移動 (次の体節の座標系)

2. 関節回転 (姿勢)

関節の位置

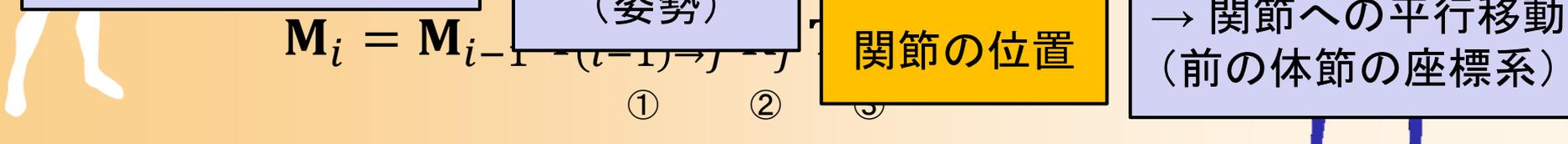
1. 前の体節の中心 → 関節への平行移動 (前の体節の座標系)

$$M_i = M_{i-1} \cdot T_{(i-1) \rightarrow j}$$

①

②

③



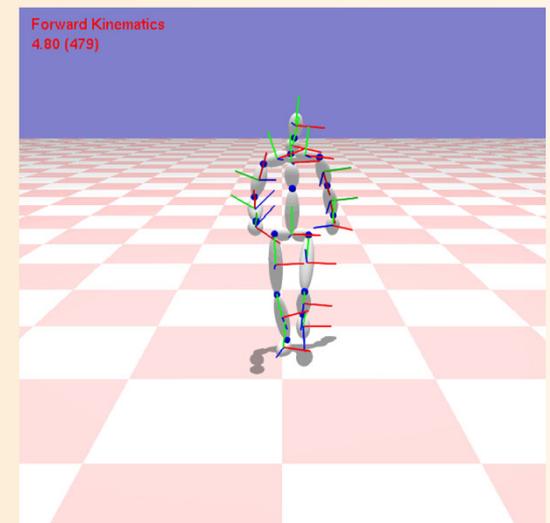
# 順運動学

- 運動学
- 順運動学
- サンプルプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ



# プログラミング演習

- サンプルプログラム (ForwardKinematicsApp.cpp) の未実装部分を作成
- 順運動学計算アプリケーション
  - 動作再生中の各姿勢から順運動学計算
  - 各関節の位置を可視化
    - 青の球で描画
  - 各体節の位置・向きを可視化
    - 局所座標系のX軸・Y軸・Z軸方向を赤・青・緑の線分で描画
  - 順運動学計算 (各自実装)



# 順運動学計算アプリケーション(1)

- ForwardKinematicsApp (一部未実装)
  - MotionPlaybackApp から派生
    - 動作再生処理は、基底クラスを利用
  - 動作再生中に順運動学計算を呼び出して、現在姿勢での全体節の位置・向き(座標系)、全関節の位置を計算して描画
    - 順運動学計算結果の描画処理は実装済み
  - 順運動学計算(各自実装)
    - MyForwardKinematicsIteration関数の一部を作成



# 順運動学計算アプリケーション(2)

- クラス定義 (ForwardKinematicsApp.h/cpp)
  - MotionPlaybackApp から派生
    - 動作再生処理は、基底クラスを利用
  - 順運動学計算結果を格納するためのメンバ変数

```
// 順運動学計算アプリケーションクラス
class ForwardKinematicsApp : public MotionPlaybackApp
{
protected:
    // 全体節の位置・向き
    vector< Matrix4f > segment_frames;
    // 全関節の位置
    vector< Point3f > joint_positions;
    ....
}
```



# 順運動学計算アプリケーション(3)

- メンバ関数 (ForwardKinematicsApp.h/cpp)
  - 基本的には基底クラスの処理を呼び出し
  - 開始処理で、計算結果配列の初期化
  - アニメーション処理で、順運動学計算の呼び出し
  - 画面描画で、計算結果の描画

```
class ForwardKinematicsApp : public MotionPlaybackApp  
{
```

```
.....
```

```
// 開始・リセット
```

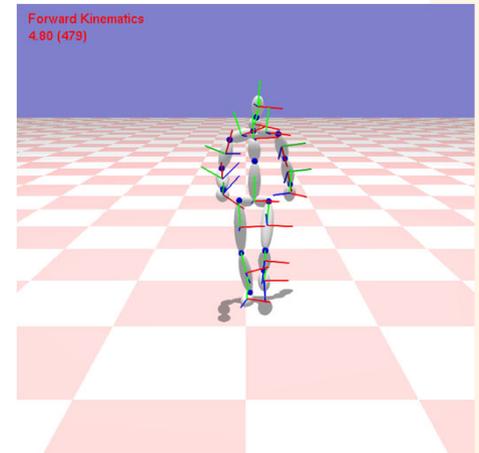
```
virtual void Start();
```

```
// 画面描画
```

```
virtual void Display();
```

```
// アニメーション処理
```

```
virtual void Animation( float delta );
```



# 順運動学計算の関数定義(1)

- 順運動学計算

- 入力: 姿勢 (各関節の回転 + ルートの位置・向き)
  - 姿勢への参照を読み取り専用 (const) で渡す  
(値渡しにすると、コピーが発生して、効率が悪い)
- 出力: 全体節の位置・向き + 全関節の位置
  - STLの可変長配列を使用
  - 出力を格納できるように参照渡し
  - 関節は向きを持たないと考えて、位置のみを求める

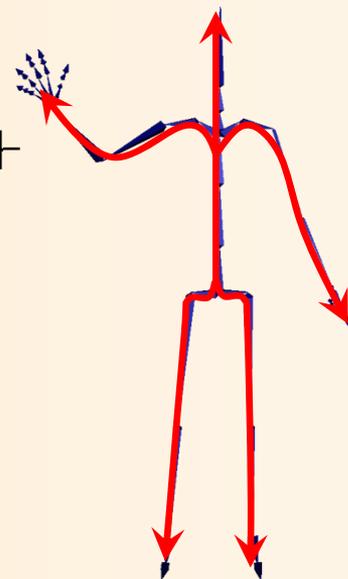
```
// 順運動学計算
```

```
void MyForwardKinematics( const Posture & posture,  
    vector< Matrix4f > & seg_frame_array,  
    vector< Point3f > & joi_pos_array );
```



# 順運動学計算の関数定義(2)

- 順運動学計算の繰り返し計算
  - 入力: 現在の体節と前の体節を入力+  
順運動学計算関数と同じ入出力
  - 次の体節に対して再帰呼び出し
    - ルート体節(腰)から各末端体節に向かつて、再帰呼び出し



```
// 順運動学計算のための反復計算
// (ルート体節から末端体節に向かつて繰り返し再帰呼び出し)
void MyForwardKinematicsIteration( const Segment * segment,
    const Segment * prev_segment, const Posture & posture,
    Matrix4f * seg_frame_array, Point3f * joi_pos_array);
```



# 順運動学計算のプログラミング

```
// 順運動学計算
void MyForwardKinematics( const Posture & posture,
    vector< Matrix4f > & seg_frame_array,
    vector< Point3f > & joi_pos_array )
{
    // 配列初期化
    seg_frame_array.resize( posture.body->num_segments );
    joi_pos_array.resize( posture.body->num_joints );

    // ルート体節の位置・向きを設定
    seg_frame_array[ 0 ].set( posture.root_ori, posture.root_pos, 1 );

    // Forward Kinematics 計算のための反復計算
    ForwardKinematicsIteration(
        posture.body->segments[ 0 ], NULL, posture,
        &seg_frame_array.front(), &joi_pos_array.front() );
}
```



# 順運動学計算のプログラミング

```
// 順運動学計算
void MyForwardKinematics( const Posture & posture,
    vector< Matrix4f > & seg_frame_array,
    vector< Point3f > & joi_pos_array )
{
    // 配列初期化
    seg_frame_array.resize( posture.body->num_segments );
    joi_pos_array.resize( posture.body->num_joints );
    // ルート体節 (root joint) の初期化
    seg_frame_array[0] = Matrix4f::Identity();
    joi_pos_array[0] = posture.root_pos;
    // Forward Kinematics 計算のための反復計算
    ForwardKinematicsIteration(
        posture.body->segments[ 0 ], NULL, posture,
        &seg_frame_array.front(), &joi_pos_array.front() );
}
```

最初の体節 (ルート体節) と前の体節 (なし) を引数として呼び出し、再帰呼び出しによる反復計算を開始



# 順運動学計算のプログラミング

```
// 順運動学計算のための反復計算
// (ルート体節から末端体節に向かって繰り返し再帰呼び出し)
void MyForwardKinematicsIteration( const Segment * segment,
    const Segment * prev_segment, const Posture & posture,
    Matrix4f * seg_frame_array, Point3f * joi_pos_array)
{
    // 現在の体節に隣接する各関節に対して繰り返し
    for ( int i=0; i<segment->num_joints; i++ )
    {
        // 次の体節・関節を取得、前の体節側(ルート側)の関節はスキップ

        // 次の体節の変換行列+次の関節の位置を計算
        // 前の体節の変換行列と関節の回転(姿勢より取得)から計算

        // 次の体節に対して繰り返し(再帰呼び出し)
        ForwardKinematicsIteration( ... );
    }
}
```



# 順運動学計算のプログラミング

```
// 順運動学計算  
// (ルート体節か
```

現在の体節と一つ前の体節+現在姿勢を入力  
全体節の位置・向きの変換行列と全関節の位置の配列も  
計算結果を格納するための引数として渡す

```
void MyForwardKinematicsFunction( const Segment * segment,  
    const Segment * prev_segment, const Posture & posture,  
    Matrix4f * seg_frame_array, Point3f * joi_pos_a
```

```
{
```

```
    // 現在の体節に隣接する各関節に対して繰り返し  
    for ( int i=0; i<segment->num_joints; i++ )
```

末端側の隣接する  
関節・体節に向かっ  
て繰り返し

```
{
```

```
    // 次の体節・関節を取得、前の体節側(ルート側)の関節はスキップ
```

再帰呼び出し  
による繰り返し

次の体節の変換行列+次の関節の位置を計算

```
    // 次の体節の変換行列と関節の回転(姿勢)から
```

```
    // 次の体節  
    ForwardKinematicsFunction( ... );
```

末端体節に到達  
したら戻る

```
    ... );
```

前のスライドの  
計算方法に従って計算、  
結果は引数として渡され  
た配列に格納

```
}
```



# 順運動学計算の繰り返し処理

- 現在の体節に隣接する全ての関節(次の関節)に対して、以下の処理を繰り返す。ただし、引数で指定された一つ前の体節の方向へは、繰り返しは行わない。
  1. 現在の体節(の中心)の位置・向きを取得 ①
  2. 現在の体節(の中心)から次の関節への平行移動をかける(現在の体節の座標系での平行移動) ②
  3. 次の関節の回転をかける ③
  4. 次の関節から次の体節(の中心)への平行移動をかける(次の体節の座標系での平行移動) ④
  5. 次の体節に対して再帰呼び出し

$$\mathbf{M}_i = \mathbf{M}_{i-1} \mathbf{T}_{(i-1) \rightarrow j} \mathbf{R}_j \mathbf{T}_{j \rightarrow i}$$

①                      ②                      ③                      ④



# 順運動学計算のプログラミング

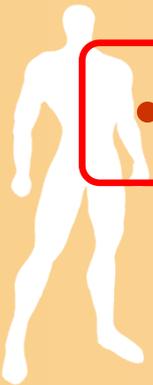
- 繰り返し処理での座標変換の計算
  - 以下の変数を使用(いずれも Matrix4f 型)
    - 次の体節の位置・向き(下式の左辺)を表す  $4 \times 4$ 変換行列 **frame**
    - 計算用の  $4 \times 4$ 変換行列 **mat**
  - $i$  番目の体節から、末端方向に隣接する次の  $j$  番目の体節に対して、以下の式を計算
    - **frame** を ① で初期化する
    - ①～③ に対応する変換行列を **mat** に代入して、**frame** に対して順番に右側からかける
      - 体節内の並行移動(3次元ベクトル)や関節の回転( $3 \times 3$ 行列)を、 $4 \times 4$ 変換行列に変換(代入)してから、かける


$$\mathbf{M}_i = \mathbf{M}_{i-1} \mathbf{T}_{(i-1) \rightarrow j} \mathbf{R}_j \mathbf{T}_{j \rightarrow i}$$

①                      ①                      ②                      ③

# 順運動学

- 運動学
- 順運動学
- サンプルプログラム
- 順運動学の計算方法
- 順運動学のプログラミング
- 順運動学のまとめ



# 順運動学と逆運動学

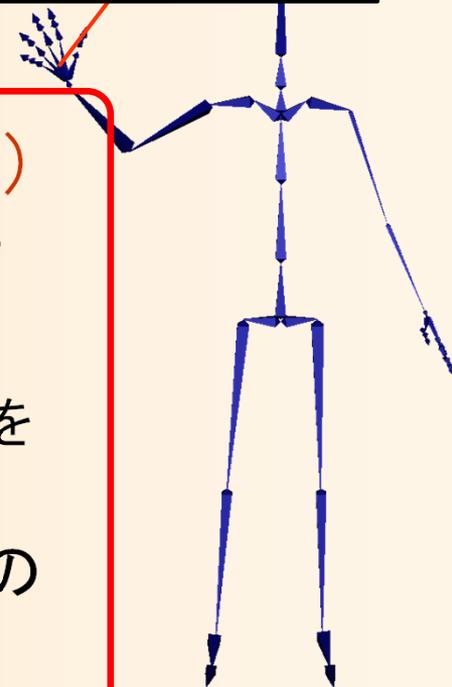
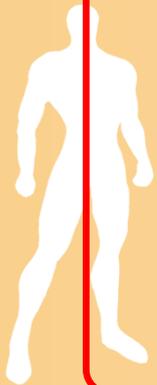
- 順運動学(フォワード・キネマティクス)

- 多関節体の関節回転から、各部位の位置・向きを計算
- 回転・移動の変換行列の積により計算

逆運動学は後日の講義で説明

- 逆運動学(インバース・キネマティクス)

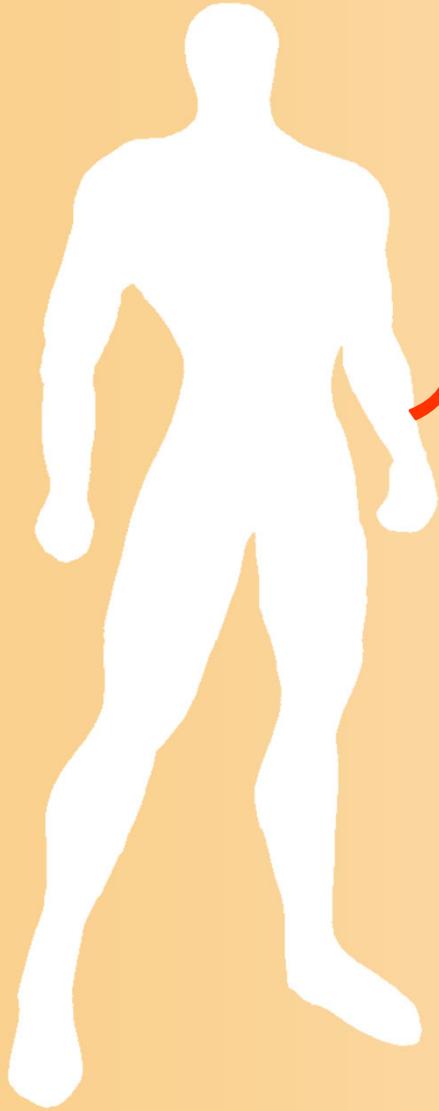
- 指定部位の目標の位置・向きから、多関節体の関節回転の変化を計算
  - 手先などの移動・回転量が与えられた時、それを実現するための関節回転の変化を計算する
- 姿勢を指定する時、関節回転よりも、手先の位置・向きなどを使った方がやりやすい
- ロボットアームの軌道計画等にも用いられる



# 今日の内容

- 前回の復習、vecmathの復習
- 順運動学
- 人体形状変形モデル

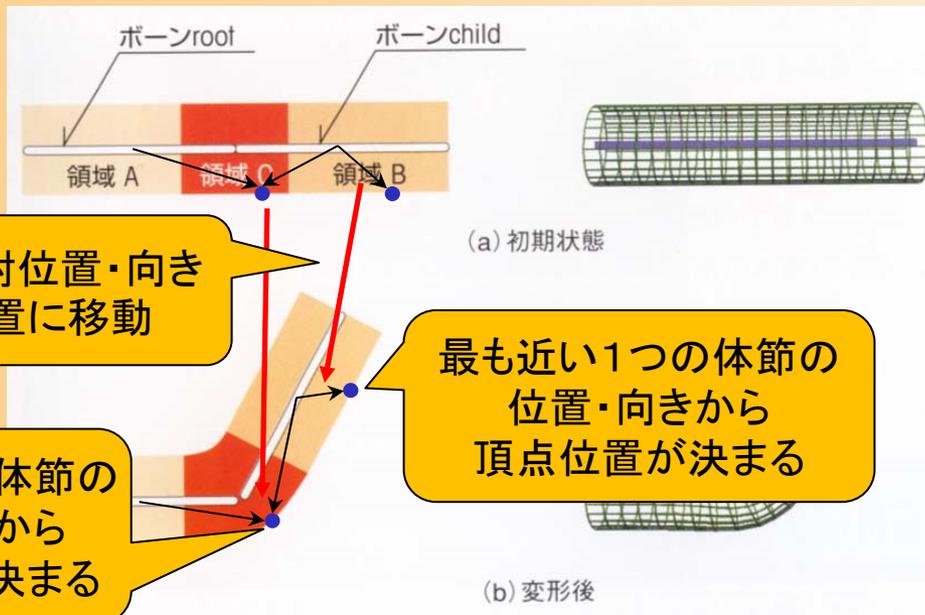




# 人体形状変形モデル

# 形状変形モデル(復習)

- 人間の形状を全身で1つのポリゴンモデルとして作成
- 骨格モデルの変形に応じてポリゴンモデルの各頂点を移動



「3DCGアニメーション」図4.16

# 形状モデルの表現(復習)

- キャラクタの形状変形モデルに必要な情報

- 骨格構造の情報
- 全身の幾何形状データ
- 骨格構造の各リンクから幾何形状の各頂点へのウェイト

- $m \times n$  の行列データ  
(リンク数 $m$ 、頂点数 $n$ )



- 通常はアニメーションソフトを使って作成したデータを利用

# 形状モデルの表現方法(復習)

- 変形のためのウェイト情報は、行列(2次元配列)により表現できる

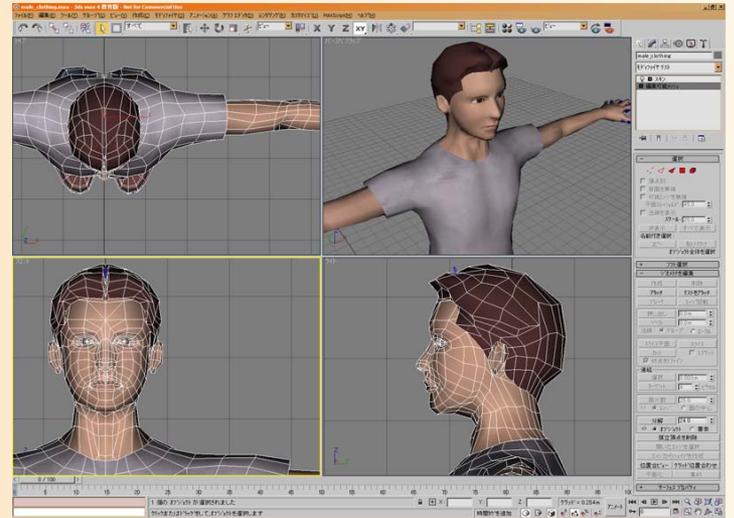
```
// ワンスキンモデルを表す構造体
struct OneSkinModel
{
    // 骨格情報
    Skeleton * skeleton;
    // 幾何形状情報
    Obj * skin_shape;
    // 変形のためのウェイト情報
    float ** weights; // [頂点番号][体節番号] の2次元配列
    // 初期姿勢での各体節の変換行列の逆行列
    Matrix4f * init_seg_frames; // [体節番号]
};
```

初期状態の姿勢  
から計算



# 人体モデルの作成方法(復習)

- 人体モデル  
(=骨格+形状モデル)  
の作成方法
- 市販のアニメーション  
制作ソフトウェアを使用  
してデザイナーが作成
- 自分のプログラムで使用するときには、ア  
ニメーション制作ソフトウェアから出力したファ  
イルを読み込んで使用



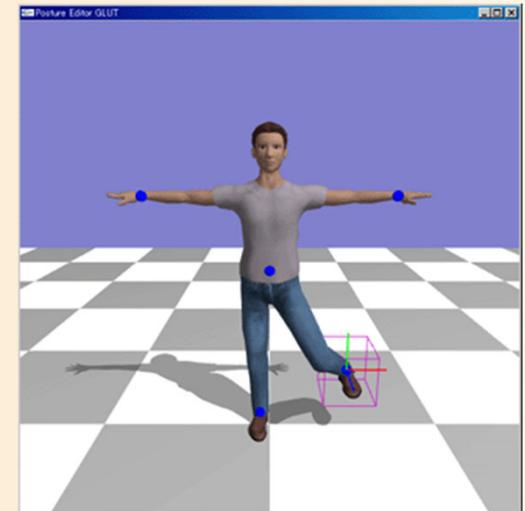
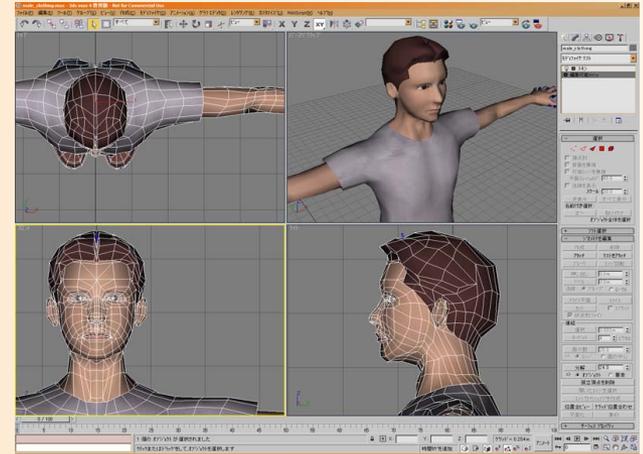
# 形状変形モデルの出力

- 一般的に仕様が公開されている形状変形モデルのファイル形式は少ないため、適当な独自形式を使うこともある
  - FBX、Collada、Xなどは、形状変形モデルも表現可能
- 各情報を個別に出力して読み込むことも可能
  - 骨格構造＋初期姿勢の情報
    - BVH形式で出力可能
  - 全身の幾何形状データ
    - Obj形式などで出力可能
  - 骨格モデルの各体節から形状モデルの各頂点への重み
    - ソフトウェアによってはテキスト形式で出力可能

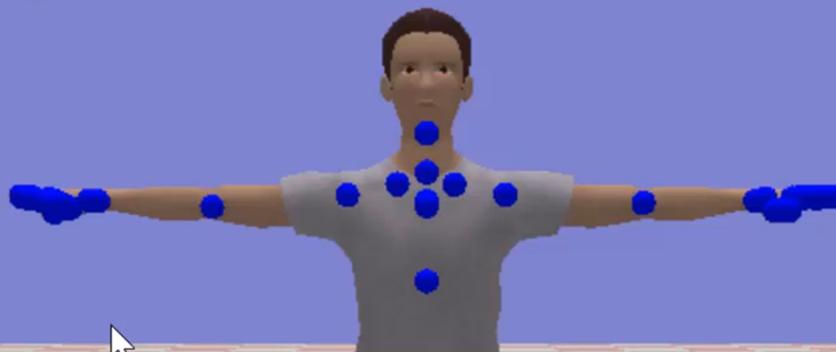


# デモプログラム

- 形状変形アプリケーション
  - 形状変形モデルの読み込み
    - 市販のアニメーション制作ソフトウェア (3ds max) で作成したキャラクターのデータを独自形式でエクスポート
    - 独自形式ファイルの読み込み
  - 姿勢変形
    - 関節点をマウスで選択してドラッグすると、姿勢を変形
    - 逆運動学計算 (後述) を使用
  - 形状モデルの変形・描画



Skin Deformation



人体形状变形

Body Shape Deformation



# 形状変形モデルの情報

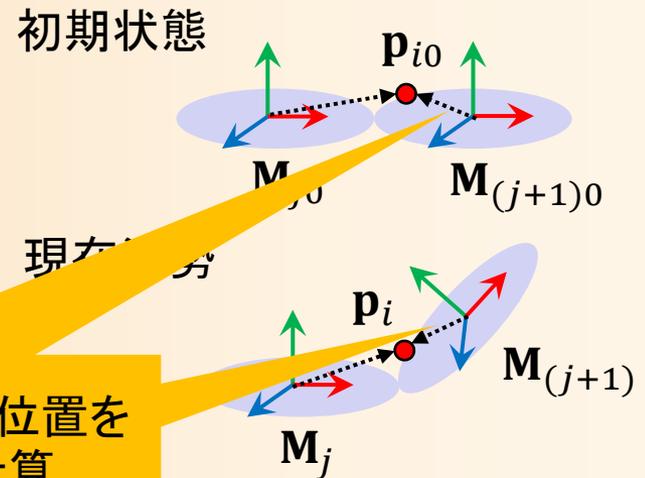
- 幾何形状データに対応する、骨格構造＋初期姿勢の情報が必要
  - 両者の位置を合わせる必要がある
  - 初期姿勢における各体節の位置・向きが必要
    - 形状変形の計算では、各体節の初期姿勢での位置・向きを表す変換行列の、逆行列を使用
    - 順運動学計算により求める



# 形状変形モデルの変形方法(1)

- 各頂点の位置を、各体節の変換行列(位置・向き)とウェイトから計算

$$\mathbf{p}_i = w_{ij} \sum_j \mathbf{M}_j \mathbf{M}_{j0}^{-1} \mathbf{p}_{i0}$$



- $\mathbf{p}_i$  各頂点の位置を計算 (体節のローカル座標系での位置を保つように頂点位置を計算)
- $\mathbf{M}_j$  各体節の変換行列 (現在の姿勢から計算)
- $w_{ij}$  各頂点が各体節から受けるウェイト (順運動学計算により姿勢から計算)
- $\mathbf{p}_{i0}$  初期状態での各頂点の位置
- $\mathbf{M}_{j0}$  初期状態での各体節の変換行列



# 形状変形モデルの変形方法(1)

- 各頂点の位置を、各体節の変換行列(位置・向き)とウェイトから計算

$$\mathbf{p}_i = w_{ij} \sum_j \mathbf{M}_j \mathbf{M}_{j0}^{-1} \mathbf{p}_{i0}$$

$\mathbf{p}_i$  各頂点の位置

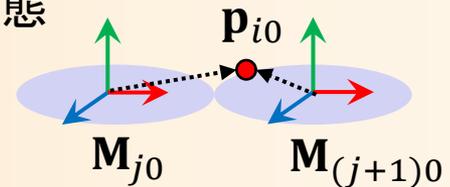
$\mathbf{M}_j$  各体節の変換行列(現在の姿勢から計算)

$w_{ij}$  各頂点が各体節から受ける重み

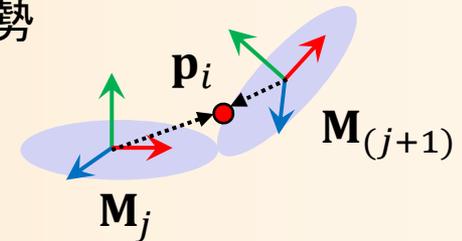
$\mathbf{p}_{i0}$  初期状態での各頂点の位置

$\mathbf{M}_{j0}$  初期状態での各体節の変換行列

初期状態



現在姿勢

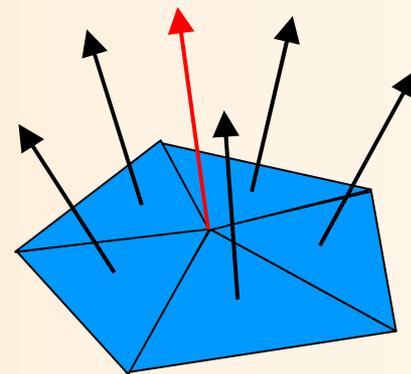


# 形状変形モデルの変形方法(2)

- 形状変形に合わせて、各頂点の法線ベクトルも計算する必要がある
- 方法1: 頂点位置と同様の方法で計算
  - 回転行列のみ適用、長さが1になるように正規化

$$\mathbf{n}_i = w_{ij} \sum_j \mathbf{R}_j \mathbf{R}_{j0}^{-1} \mathbf{n}_{i0}$$

- 方法2: ポリゴンモデルから計算
  - 頂点を共有する面の法線を平均
  - 長さが1になるように正規化



# 形状変形モデルの変形処理

```
// ワンスキンモデルの変形計算
void DeformSkinModel( const OneSkinModel * model, const Posture & posture,
                      Point3f * vertices, Vector3f * normals )
{
    // 現在姿勢での各体節の変換行列を計算

    // 各頂点の位置・法線を計算
}
```

ワンスキンモデル情報と  
現在姿勢を入力  
各頂点の位置を計算して出力  
(頂点座標と法線ベクトルの配列の  
先頭アドレスを引数として受け取る)

```
// ワンスキンモデルの描画
// 幾何形状モデル(Obj形状)の描画(頂点座標の配列を入力)
void RenderDeformedObj( const Obj * obj,
                        const Point3f * vertices, const Vector3f * normals )
{
    // 幾何形状モデルが持つ頂点位置の配列の代わりに、引数として渡された
    // 頂点位置・法線ベクトルの配列を使用して描画
    // ポリゴンや素材の情報は、幾何形状モデルが持つ情報を使用
}
```

# GPUを使った変形処理の実現

- 実際のアプリケーションでは、GPUを使った変形計算が用いられる
- Vertex Shader を使った変形計算
  - 描画時に動的に頂点位置・法線ベクトルを計算
  - 各頂点に対する各体節からの重みの情報は、別途パラメタとして与える
    - 実際には大部分の重みは0であり、各頂点に影響を与える体節の数は少ないため、そのことを利用して、コンパクトな形式で与えることができる
      - 例えば、頂点ごとに2つの4次元ベクトルを使用して、最大4つの体節番号と重みの情報を与える、など



# まとめ

- 前回の復習、vecmathの復習
- 順運動学
- 人体形状変形モデル



# レポート課題

- キャラクタ・アニメーション(1)

- サンプルプログラムの未実装部分(前半)を作成

1. 順運動学計算

2. 姿勢補間

3. キーフレーム動作再生

4. 動作補間

- サンプルプログラムの未実装部分(後半)は次の課題

5. 動作変形

6. 動作接続・遷移

7. 逆運動学計算(CCD法)

残りの課題は  
次回以降説明



# 次回予告

- 人体モデル(骨格・姿勢・動作)の表現
- 人体モデル・動作データの作成方法
- サンプルプログラム、動作再生
- 順運動学、人体形状変形モデル
- 姿勢補間、キーフレーム動作再生、動作補間
- 動作接続・遷移、動作変形
- 逆運動学、モーションキャプチャ
- 動作生成・制御

