



コンピュータグラフィックス特論Ⅱ

第11回 キャラクタアニメーション(2)

九州工業大学 尾下 真樹

2021年度

今日の内容

- 前回の復習
- BVH動作データの読み込みと再生
- サンプルプログラム
- 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
- 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
- BVH動作データの読み込みと再生
- サンプルプログラム
- 順運動学
- 人体形状変形モデル





前回の復習

キャラクター・アニメーションの実現方法

- オフライン・アニメーション制作
 - 通常は市販のアニメーション制作ソフトウェアを利用
- オンライン・アニメーション生成
 - ゲームエンジン(ミドルウェア)の利用
 - Unity, Unreal 等(市販のコンピュータゲーム等でも利用されている)
 - 基本的には、アニメーション制作ソフトウェアで作成されたキャラクターモデルや動作データを再生する機能を提供
 - 提供されている機能以上の高度な動作生成・変形は困難
 - 自分でソフトウェアライブラリを開発
 - 高度な処理も自由に追加できる
 - キャラクターモデルや動作データは、他のソフトウェアで作成されたファイルを読み込んで使用する必要がある



人体モデルの表現

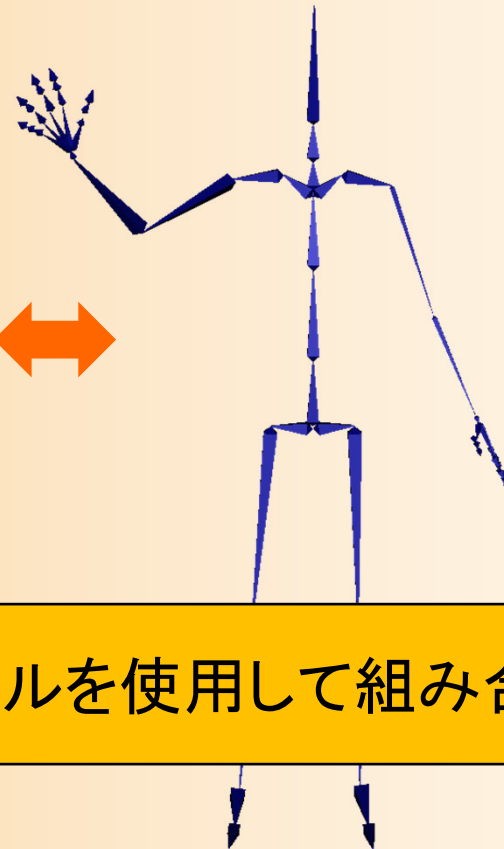
形状モデル
(ポリゴンモデル)

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

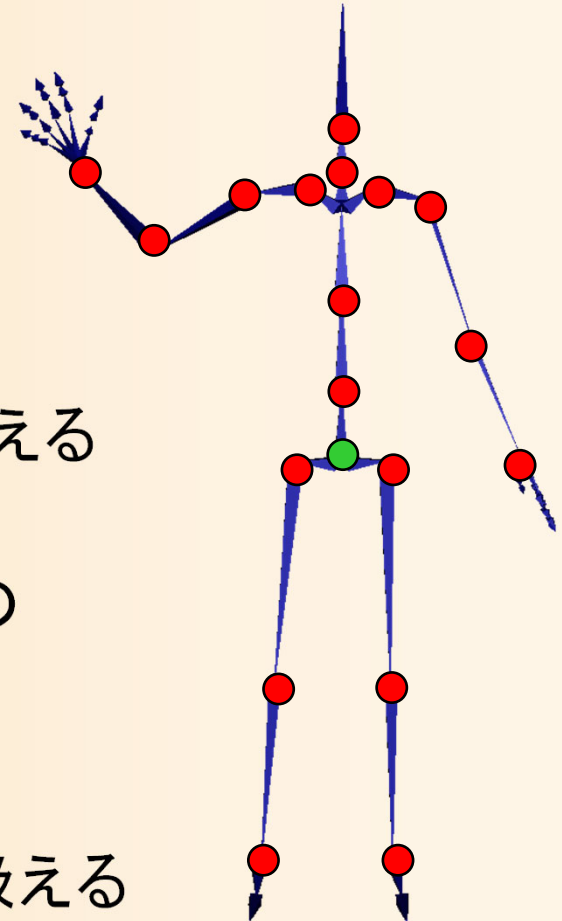
- 複数の体節(部位)が
関節で接続されたモデル

- 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

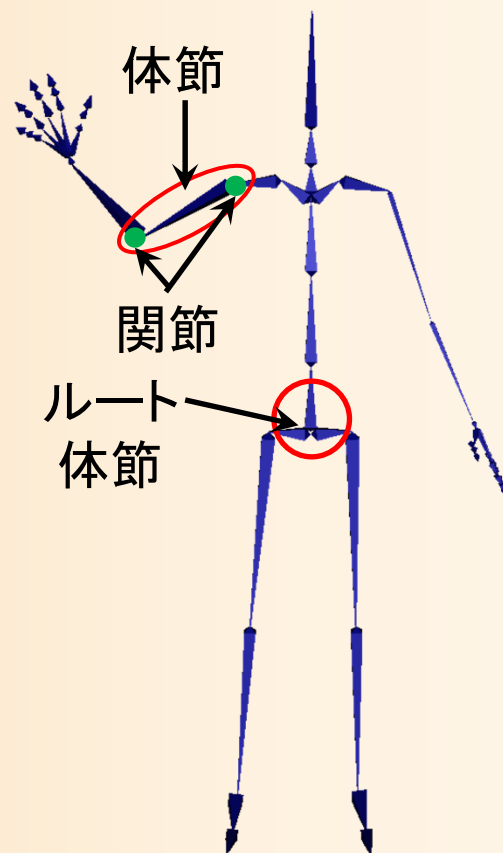
- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける

– 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



骨格モデルの表現方法

- 骨格情報の中で、体節と関節のデータ構造を分ける

```
// 多関節体の体節を表す構造体
struct Segment
{
    // 接続関節
    vector< Joint * >    joints;
    // 各関節の接続位置(体節のローカル座標系)
    vector< Point3f >   joint_positions;
};
```

```
// 多関節体の関節を表す構造体
struct Joint
{
    // 接続体節
    Segment *           segments[ 2 ];
};
```

```
// 多関節体の骨格を表す構造体
struct Skeleton
{
    // 体節・関節の配列
    vector< Segment * > segments;
    vector< Joint * >    joints;
};
```

姿勢の表現方法

- 骨格情報と姿勢情報のデータ構造を分ける

```
// 多関節体の姿勢を表す構造体
struct Posture
{
    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; // 各関節の回転(回転行列表現)
                                   // [関節番号] 関節数分の配列
};
```



骨格モデルの表現方法のまとめ

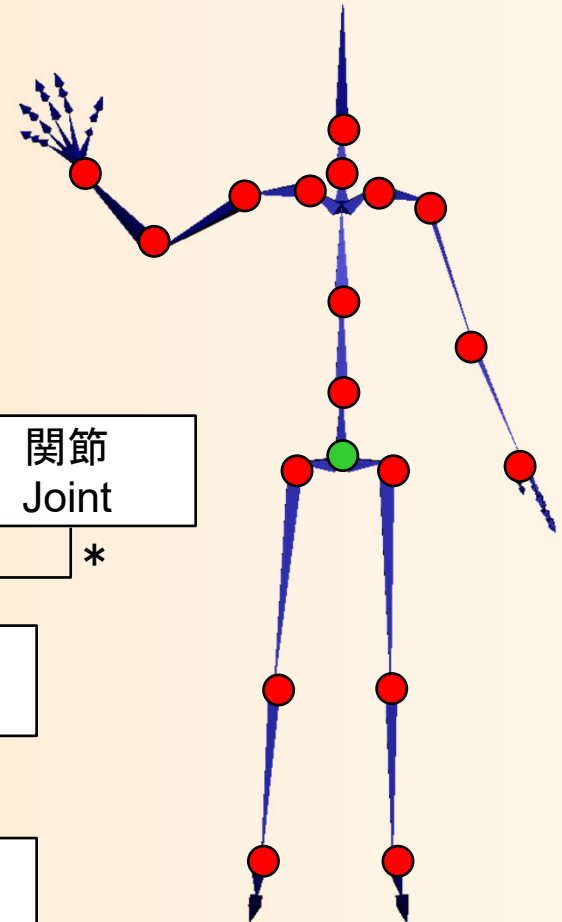
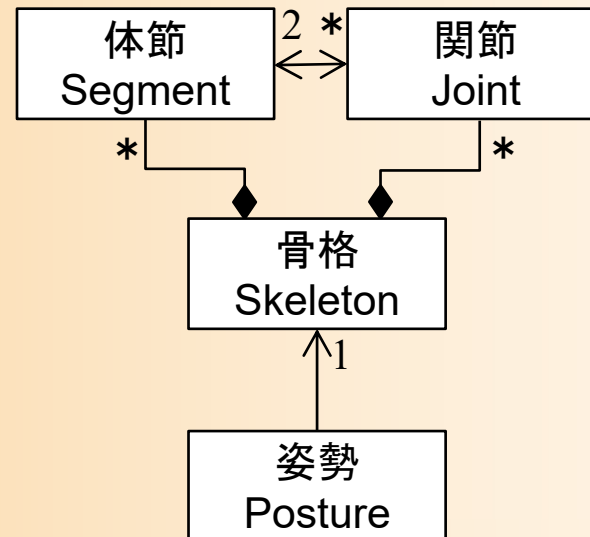
- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、体節と関節を分ける

// 多関節体の体節を表す構造体
struct Segment

// 多関節体の関節を表す構造体
struct Joint

// 多関節体の骨格を表す構造体
struct Skeleton

// 多関節体の姿勢を表す構造体
struct Posture





BVH動作データの読み込みと再生

動作データのファイル形式の例

- 仕様が公開されているフォーマットは少ない
 - BVH、BVA、ASF-AMC、FBX (MotionBuilder)、VRML、X、COLLADA
- BVH形式
 - アスキー形式で可読性が高く、扱いやすい
 - 骨格情報と動作情報(各時刻の姿勢)を持つ
 - 姿勢はオイラー角表現
- ASF-AMC形式
 - 骨格情報(ASF形式) + 動作情報(AMC形式)
 - アスキー形式、姿勢はオイラー角表現



BVH形式

• BVH形式の仕様

- 詳しい仕様はネット上で探せば見つかる
- 骨格情報は、骨格情報の表現方法2に近い形式
 - 体節＋親側の関節をまとめて一つの関節(JOINT)として扱う
 - 各関節が持つ各回転軸(＋ルート関節の座標軸)をチャンネル(CHANNELS)として定義
- 動作情報は、各フレームの全チャンネルの値を順番に格納

• BVHのサンプルデータ例

- Eyes, Japan <http://www.mocapdata.com/>



BVH形式の例(1)

- 骨格情報

```
HIERARCHY
ROOT Hips
  OFFSET 0 0 0
  CHANNELS 6 Xposition Yposition Zposition
  Zrotation Xrotation Yrotation
  JOINT LeftHip
    OFFSET 3.43 0 0
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftKnee
      OFFSET 0 -18.47 0
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftAnkle
        ...
```

HIERARCHY
ROOT Hips

{

OFFSET 0 0 0
CHANNELS 6 Xposition Yposition Zposition
Zrotation Xrotation Yrotation

JOINT LeftHip

{

OFFSET 3.43 0 0
CHANNELS 3 Zrotation Xrotation Yrotation
JOINT LeftKnee

{

OFFSET 0 -18.47 0
CHANNELS 3 Zrotation Xrotation Yrotation
JOINT LeftAnkle

{

...



BVH形式の例(1)

骨格情報

```

HIERARCHY
ROOT Hips
{
  OFFSET 0 0 0
  CHANNELS 6 Xposition Yposition Zposition
  Zrotation Xrotation Yrotation
  JOINT LeftHip
  {
    OFFSET 3.43 0 0
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftKnee
    {
      OFFSET 0 0 -18.47 0
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftAnkle
      {
        ...
      }
    }
  }
}

```

骨格情報(JOINTの階層構造)の
始まり

ルート関節
(ROOT)

親関節に対する
相対位置

関節が持つ
姿勢情報
CHANNELS

子関節(JOINT)

以下、同様に階層
構造の情報が続く
位置情報を持つのは
ルート関節のみ

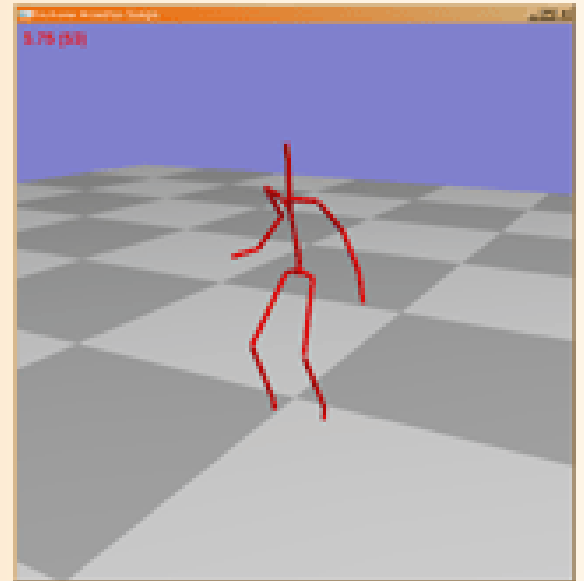
```

HIERARCHY
ROOT Hips
{
  OFFSET 0 0 0
  CHANNELS 6 Xposition Yposition Zposition
  Zrotation Xrotation Yrotation
  JOINT LeftHip
  {
    OFFSET 3.43 0 0
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftKnee
    {
      OFFSET 0 0 -18.47 0
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftAnkle
      {
        ...
      }
    }
  }
}

```


デモプログラム

- BVH動作の読み込みと再生 (BVH Player)
 - BVH動作ファイルを読み込んで再生
 - 骨格情報にもとづいて各フレームの姿勢を描画
- LキーでBVHファイルを選択
 - BVHファイルは講義のページには置いていないので、各自、ネット上で公開されているものなどを探して試すこと



Press 'L' key to Load a BVH file

BVH動作再生
BVH Player



サンプルプログラム(1)

- BVH動作の読み込みと再生 (bvh_player.cpp)
- BVHクラス (BVH.h/cpp)
 - BVHデータ構造の定義
 - なるべくBVH形式に近い形式のデータ構造を定義
 - 骨格情報 + 動作情報
 - BVHファイルの読み込み
 - 読み込んだBVH動作データの任意のフレーム番号の姿勢を描画
- GLUT + OpenGLを使用



BVHクラス

```
class BVH
{
public:
    // チャンネルの種類
    enum ChannelEnum
    {
        X_ROTATION, Y_ROTATION, Z_ROTATION,
        X_POSITION, Y_POSITION, Z_POSITION
    };
    struct Joint;

    // チャンネル情報
    struct Channel
    {
        // 対応関節
        Joint *    joint;

        // チャンネルの種類
        ChannelEnum    type;

        // チャンネル番号
        int            index;
    };

    // 関節情報
    struct Joint
    {
        // 関節名
        string        name;
        // 関節番号
        int            index;
    };
};
```

```
    // 関節階層(親関節)
    Joint *    parent;
    // 関節階層(子関節)
    vector< Joint * >    children;

    // 接続位置
    double        offset[3];

    // 末端位置情報を持つかどうかのフラグ
    bool        has_site;
    // 末端位置
    double        site[3];

    // 回転軸
    vector< Channel * >    channels;
};

/* 階層構造の情報 */
int        num_channel; // チャンネル数
vector< Channel * >    channels; // チャンネル情報 [チャンネル番号]
vector< Joint * >    joints; // 関節情報 [パーツ番号]
map< string, Joint * >    joint_index; // 関節名から関節情報へのインデックス

/* モーションデータの情報 */
int        num_frame; // フレーム数
double        interval; // フレーム間の時間間隔
double *    motion; // [フレーム番号][チャンネル番号]

// コンストラクタ
BVH( const char * bvh_file_name );
```


BVHクラス

```
class BVH
{
public:
    // チャンネルの種類
    enum ChannelEnum
    {
        X_ROTATION, Y_ROTATION, Z_ROTATION,
        X_POSITION, Y_POSITION, Z_POSITION
    };
    struct Joint;

    // チャンネル情報
    struct Channel
    {
        // 対応関節
        Joint * joint;

        // チャンネルの種類
        ChannelEnum type;

        // チャンネル番号
        int index;
    };

    // 関節情報
    struct Joint
    {
        // 関節名
        string name;
        // 関節番号
        int index;
```

BVHの骨格
情報の定義で
用いられる
CHANNELS,
JOINT(ROOT)
に対応する
構造体

```
        // 関節階層(親関節)
        Joint * parent;
        // 関節階層(子関節)
        vector< Joint * > children;

        // 接続位置
        double offset[3];

        // 末端位置情報を持つかどうかのフラグ
        bool has_site;
        // 末端位置
        double site[3];

        // 回転軸
        vector< Channel * > channels;
    };

    /* 階層構造の情報 */
    int num_channel; // チャンネル数
    vector< Channel * > channels; // チャンネル情報 [チャンネル番号]
    vector< Joint * > joints; // 関節情報 [パーツ番号]
    map< string, Joint * > joint_index; // 関節名から関節情報へのインデックス

    /* モーションデータの情報 */
    int num_frame; // フレーム数
    double interval; // フレーム間の時間間隔
    double * motion; // [フレーム番号][チャンネル番号]

    // コンストラクタ
    BVH( const char * bvh_file_name );
```

骨格情報
(JOINTの
階層構造)

動作情報

BVHファイルを読み込み

サンプルプログラム(2)

- BVH動作の読み込み処理
 - BVH::BVH(const char * bvh_file_name) 関数
 - iostream を使用
 - 骨格情報の読み込み
 - 1行ずつファイルから文字列を読み込み
 - タグに応じて、関節オブジェクトの生成と属性の設定
 - 動作情報の読み込み
 - 1行ずつファイルから文字列を読み込み
 - 各フレームの姿勢データの設定



サンプルプログラム(3)

• BVH動作の再生

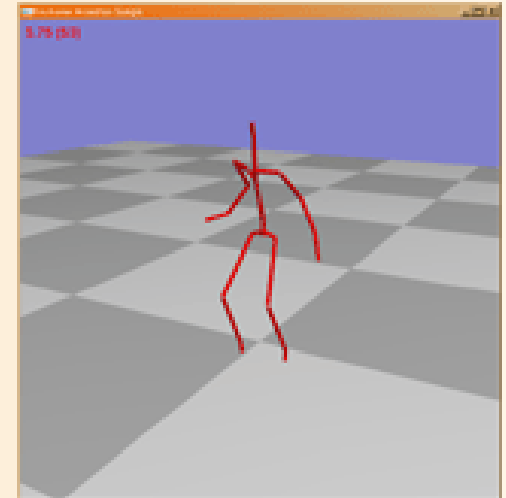
– 動作データ中の指定フレームの姿勢を描画

- `BVH::RenderFigure(int frame_no, float scale)`

- 何フレーム目の姿勢を描画するかを指定
- BVHファイルによって、座標の単位が異なるため、描画時の比率も指定できるようになっている

– 各体節を円柱を使って描画

- GLUTの描画関数を使用
- 順運動学計算(後述)により、各体節の位置・向きを計算
 - 骨格・姿勢情報から計算
 - OpenGLの視野変換行列を使って計算



今日の内容

- 前回の復習
- BVH動作データの読み込みと再生
- サンプルプログラム
- 順運動学
- 人体形状変形モデル

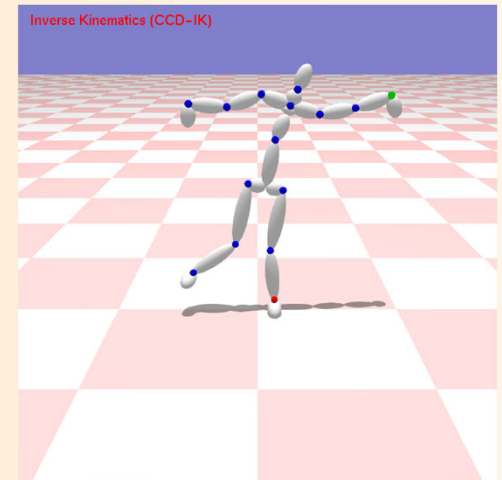
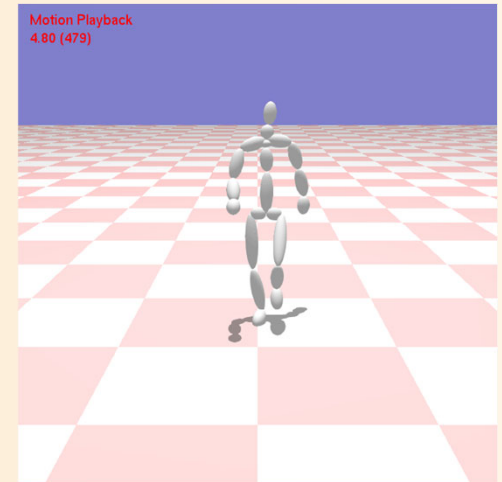




サンプルプログラム

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)



Motion Playback

0.10 (10)

キャラクターアニメーション Human Animation



サンプルプログラム

- デモプログラムの一部

- 骨格・姿勢・動作のデータ構造定義 (SimpleHuman.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 (GLUTBaseApp)
 - 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
- コールバック関数 (SimpleHumanGLUT.h/cpp)
 - GLUTBaseAppの定義・実装、全アプリケーションを管理・切替
 - 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)



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)

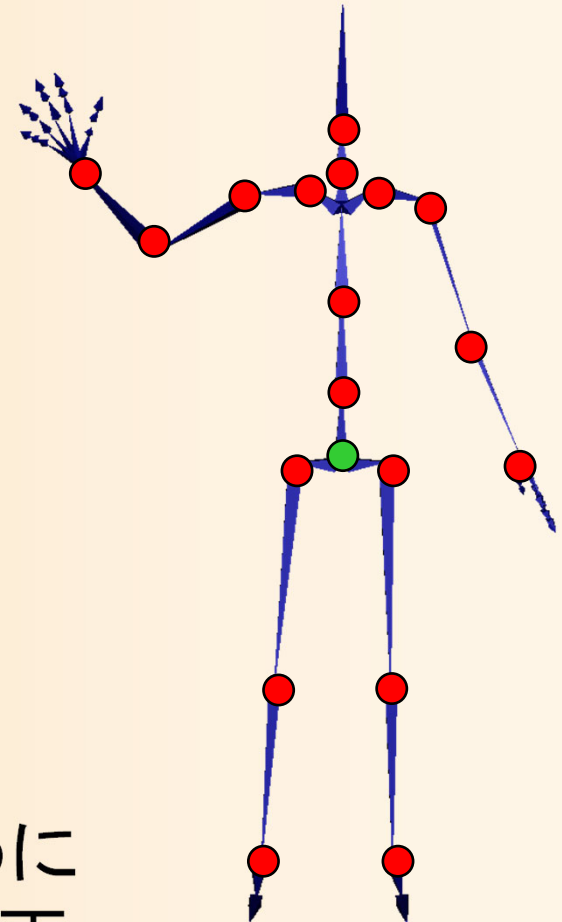
```
// 人体モデルの体節を表す構造体
struct Segment

// 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
```

- プログラムから利用しやすいように
前回のデータ構造から細部を変更



骨格・姿勢の表現方法

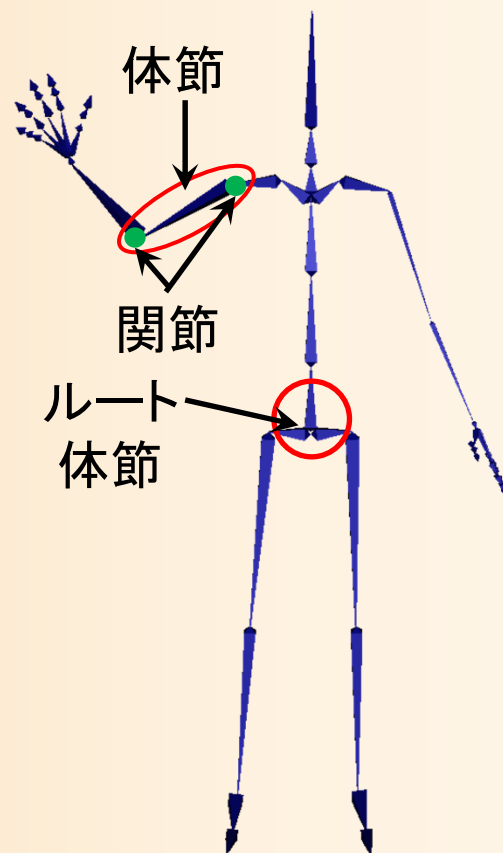
- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける

- 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



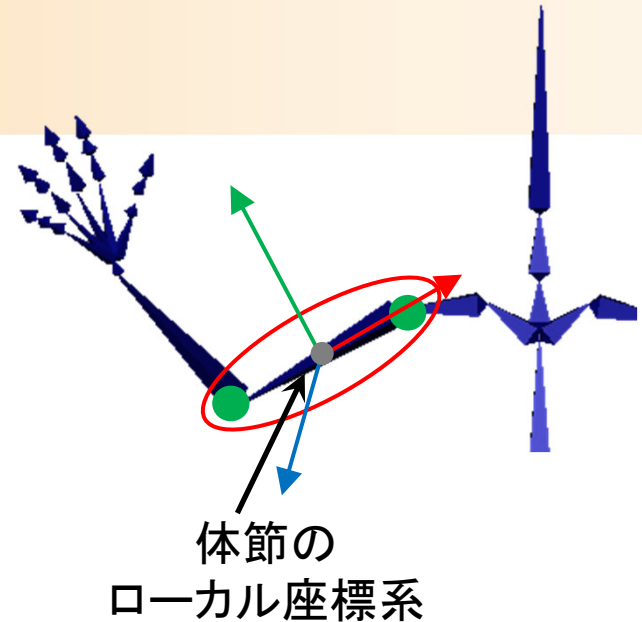
骨格モデルのデータ構造(1)

• 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;
};
```



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]

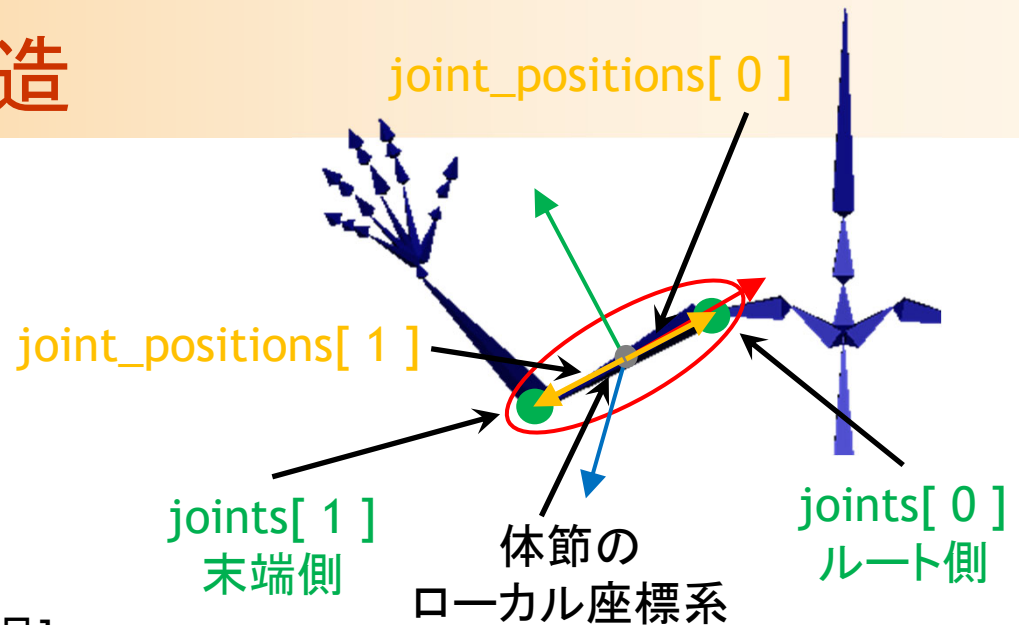
Point3f * joint_positions;

// 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;

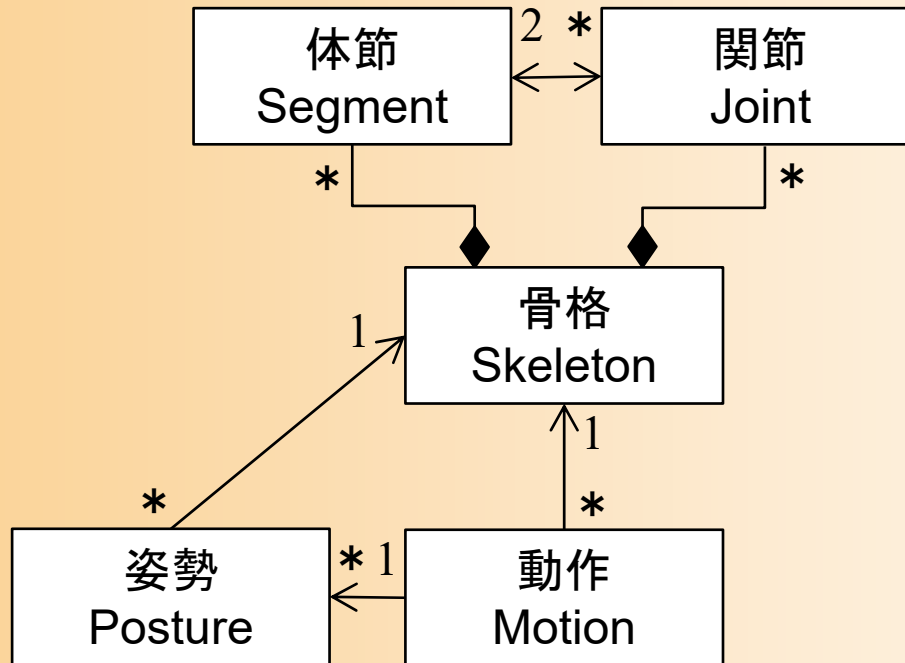
    // 姿勢を取得
    void GetPosture( float time, Posture & p ) const;
};
```

時刻を入力として、
その時刻の姿勢を出力



クラス図

- クラス・構造体間の関係



骨格・動作の読み込み

- 骨格モデルや動作データが必要
- BVH形式の動作データを読み込み、骨格モデル・動作データに変換する
 - BVH動作を扱うためのクラス(BVH)や読み込み処理は、前に説明した通り
 - BVH動作のままでは扱いづらいため、変換する



```
// BVH動作から骨格モデルを生成
```

```
const Skeleton * CoustructBVHSkeleton( class BVH * bvh );
```

```
// BVH動作から動作データ(+骨格モデル)を生成
```

```
Motion * CoustructBVHMotion( class BVH * bvh, const Skeleton * b );
```

骨格モデルの生成

- BVH動作の骨格情報から骨格モデルを生成
 - BVH形式の関節は、体節＋親側の関節を組み合わせたもの(骨格モデルの表現方法2)
 - サンプルプログラムで使用する骨格モデル(体節・関節を分ける)に合わせて変換が必要
 - 各BVH関節から一つの体節を生成
 - 全接続関節の中心を基準とする体節のローカル座標系での各接続関節の位置を計算
 - 各BVH関節から一つの関節を生成
 - ただし、ルート of BVH関節からは生成しない
 - $BVH\text{関節数} = \text{体節数} = \text{関節数} + 1$ となる



動作データの生成

- BVH動作から動作データを生成
 - 一定間隔動作データを生成
 - BVH形式では関節回転がオイラー角で表現されている
 - サンプルプログラムで使用する姿勢表現に合わせて、回転行列への変換が必要
 - BVH関節が持つチャンネルの情報にもとづいて、各軸周りの回転行列を順番に掛けることで、回転行列を計算する



描画処理

- 姿勢描画

```
// 姿勢の描画(スティックフィギュアで描画)
void DrawPosture( const Posture & posture );

// 姿勢の影の描画(スティックフィギュアで描画)
void DrawPostureShadow( const Posture & posture,
                        const Vector3f & light_dir, const Color4f & color );
```

- 内部で順運動学計算(後述)を呼び出し
- 各体節を楕円体として描画
 - 楕円体の大きさは、体節内の接続関節位置から計算
 - 楕円体の描画には、OpenGLの関数を使用



アプリケーションの基底クラス

- GLUTBaseApp (SimpleHumanGLUT.h/cpp)

```
class GLUTBaseApp
{
protected:
    // 視点操作のための変数
    // マウス入力処理のための変数
    // アプリケーション状態の変数
public:
    // イベント処理インターフェース
    virtual void Initilize();
    virtual void Start();
    virtual void Display();
    ...
    virtual void Animation( float delta );
};
```



アプリケーションの基底クラス

- GLUTBaseApp (SimpleHumanGLUT.h/cpp) のイベント処理インターフェース
 - 初期化 (最初に一度だけ呼ばれる)
 - 開始処理 (アプリケーション切替時に呼ばれる)
 - 画面描画
 - アニメーション (アイドル処理から呼ばれる)
 - 前回の処理からの経過時刻が引数として渡される
 - ウィンドウサイズ変更
 - 入力処理
 - マウスクリック、マウスドラッグ、マウス移動
 - キー押下、特殊キー押下
- ※ 各アプリケーションで必要な処理を実装



GLUTメイン処理

- 複数のアプリケーションを管理・切り替え

```
// 全アプリケーションのリスト  
vector< GLUTBaseApp * >  applications;
```

初期化時に、全アプリケーションを生成・登録

```
// 現在実行中のアプリケーション  
GLUTBaseApp *  app = NULL;
```

上のリストの中の、現在実行中のアプリケーションを表す

- GLUTコールバック関数から、現在のアプリケーションのイベント処理を呼び出し

```
//void DisplayCallback( void )  
{  
    app->Display();  
    ...  
}
```

基底クラスで定義されたインターフェースを通じて、各派生クラスで実装された処理が呼ばれる。オブジェクト指向の多態(ポリモーフィズム)の応用。



プログラムのメイン処理

- main関数 (SimpleHumanMain.cpp)
 - 全てのアプリケーションの配列を渡して開始

```
int main( int argc, char ** argv )
{
    // 全アプリケーションのリスト
    vector< class GLUTBaseApp * >    applications;

    // 全アプリケーションを登録
    applications.push_back( new MotionPlaybackApp() );
    ...

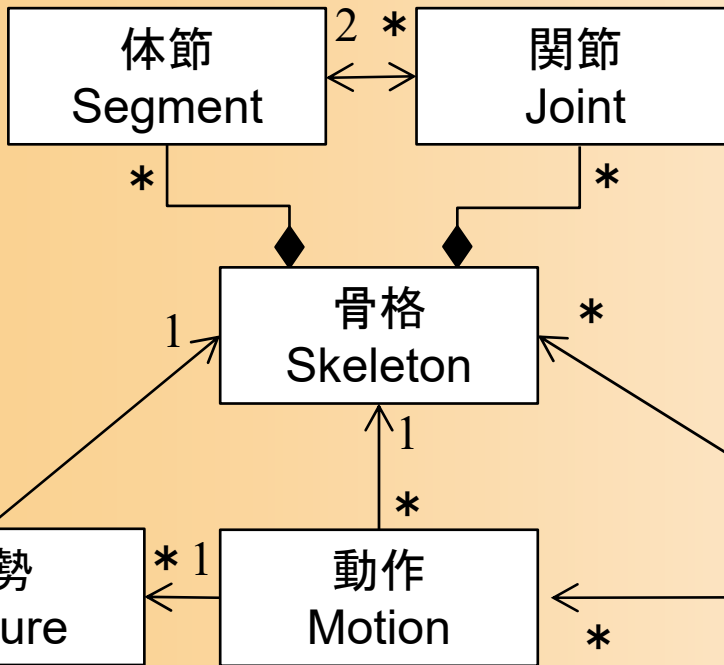
    // GLUTフレームワークのメイン関数を呼び出し
    SimpleHumanGLUTMain( applications, argc, argv );
};
```

SimpleHumanGLUT.h/cpp で定義・実装



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

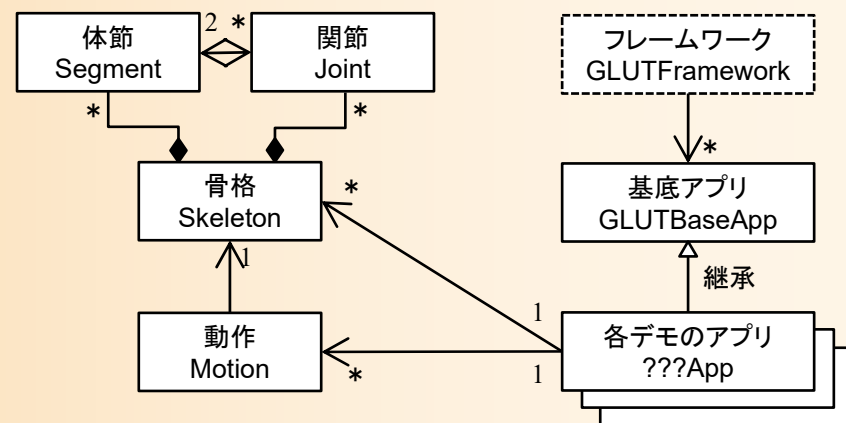
各デモのアプリ
???App



参考: UML

- 前スライドのクラス図は、UMLに基づく記述
- Unified Modeling Language (UML)
 - オブジェクト指向に基づくソフトウェア設計を図で記述するときの描き方

- ソフトウェア開発で広く使われている
- クラス図以外にも、シーケンス図、ユースケース図などがある
- UMLを知っていれば、コミュニケーションを円滑に進められる



参考：デザインパターン

- **デザインパターン**

- オブジェクト指向によるソフトウェア設計では、複数のクラス同士が連携して大きな機能を実現する
- よく使われるクラスの役割の組み合わせをパターンとしてまとめたものをデザインパターンと呼ぶ
 - 23種類の GOFパターンが代表的なデザインパターン
- デザインパターンを知っていれば、ソフトウェア設計に応用でき、コミュニケーションを円滑に進められる

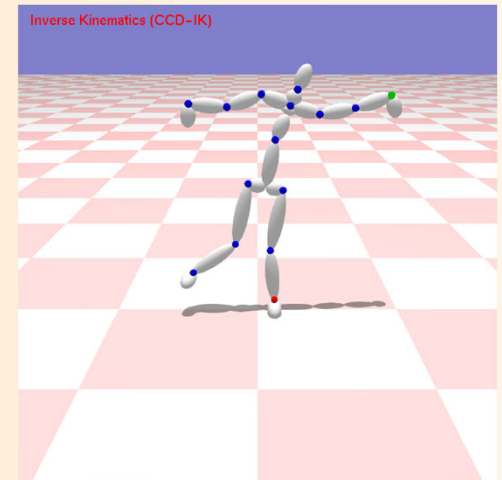
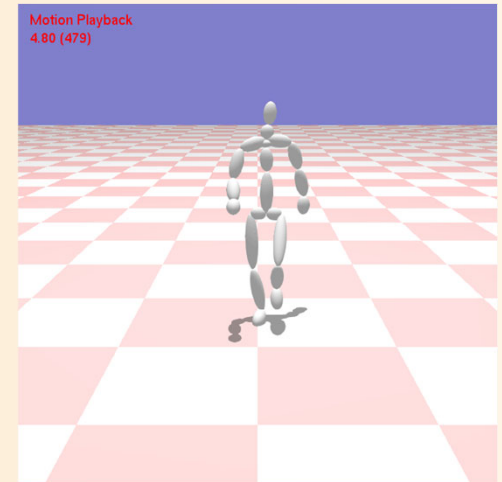
- **本プログラムは、テンプレートパターンに基づく**

- アプリケーションの処理手順の枠組みを規定して、その枠組みにそった複数の異なるアプリケーションを実装



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)



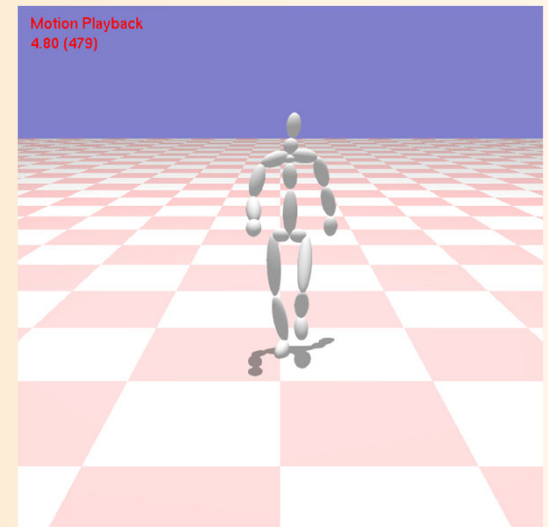
動作再生アプリケーション

- 動作再生アプリケーション

- BVH動作を読み込んで再生

- Sキーで、一時停止・再生
 - 一時停止中にP・Nキーで、前・次のフレーム
 - Wキーで、再生速度を変更
 - Lキーを押すと、読み込むBVHファイルを選択

- 最初に自動的に読み込むBVHファイルは、プログラム中で指定



Motion Playback

0.10 (10)

動作再生

Motion Playback



動作再生アプリケーション

- MotionPlaybackApp (実装済み)
 - GLUTBaseAppを派生
 - 初期化処理 (Initialize関数) で、骨格や動作を読み込み (BVH動作を読み込んで変換)
 - 開始処理 (Start関数) で、再生時刻をリセット
 - アニメーション処理 (Animation関数)
 - 経過時間 δt に応じてアニメーション時間を進める
 - 動作から現在時刻の姿勢を取得
 - 描画処理 (Display関数)
 - 現在姿勢を描画、時刻・フレーム数を描画



動作再生アプリケーション

- クラス定義・実装 (MotionPlaybackApp.h/cpp)

```
class MotionPlaybackApp : public GLUTBaseApp
{
protected:
    // キャラクタの骨格
    const Skeleton * body;
    // キャラクタの姿勢
    Posture * curr_posture;
    // 動作データ
    Motion * motion;
    ...
public:
    // イベント処理インターフェース
    ....
};
```

実装の詳細の説明は省略
(各自でソースファイルを確認
すること)



レポート課題(予告)

- キャラクタ・アニメーション
 - サンプルプログラムの未実装部分を作成
 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
 5. 動作接続・遷移
 6. 動作変形
 7. 逆運動学計算(CCD法)



今日の内容

- 前回の復習
- BVH動作データの読み込みと再生
- サンプルプログラム
- 順運動学
- 人体形状変形モデル

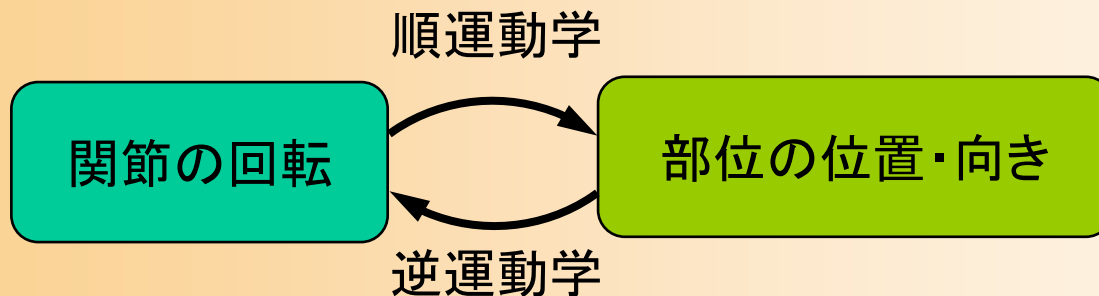
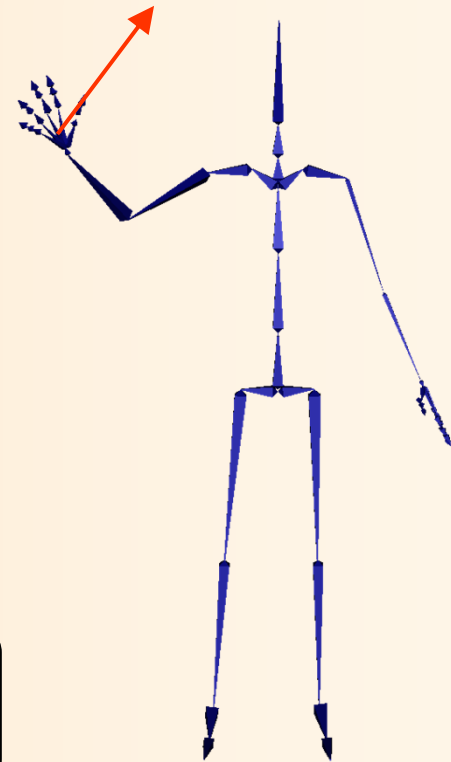




順運動学

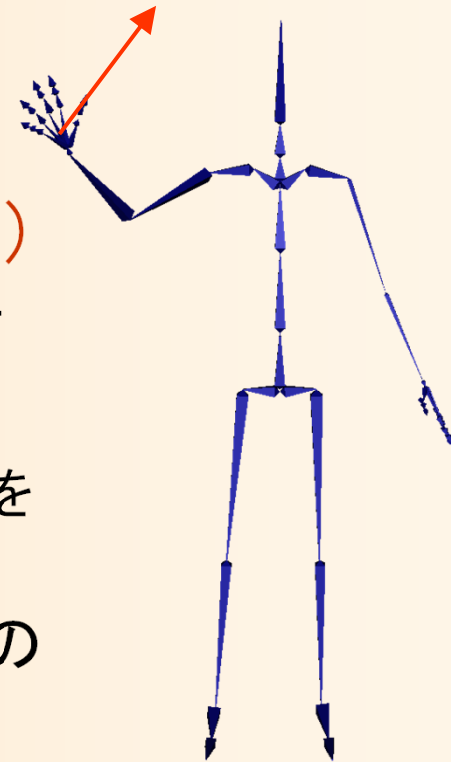
運動学

- 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

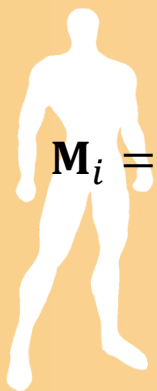


順運動学の計算方法

- 順運動学 (フォワード・キネマティクス)

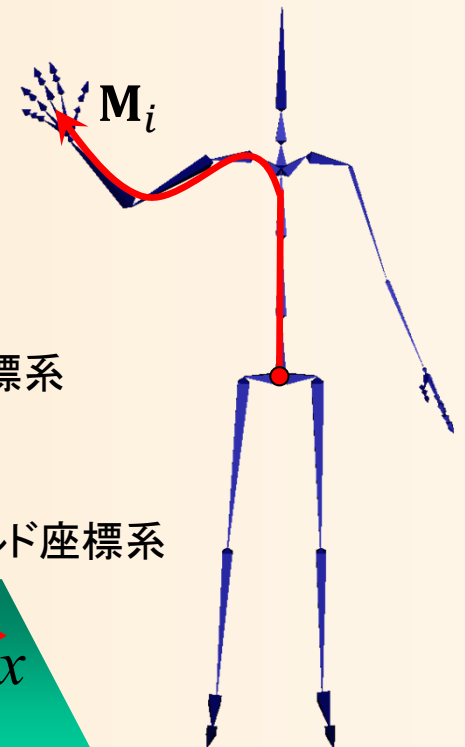
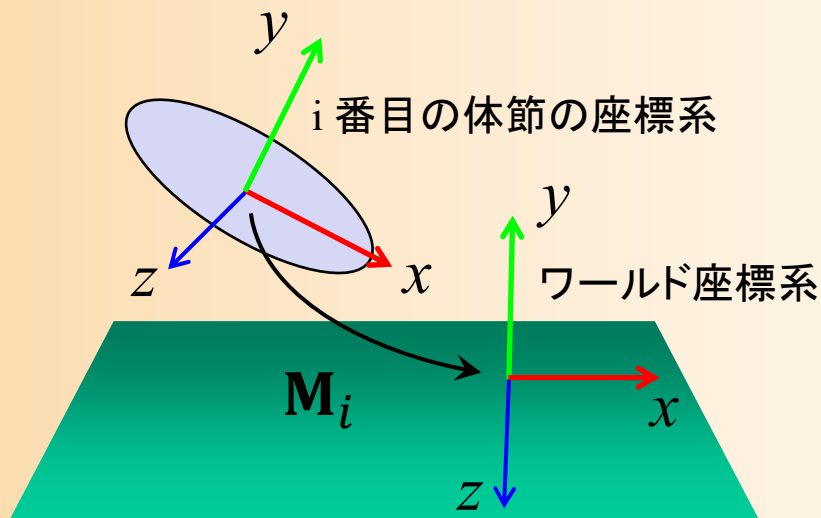
- 全体節 (and 関節) の位置・向きを表す変換行列を計算

- 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



$$M_i = \begin{pmatrix} \begin{matrix} R_{xx} & R_{xy} & R_{xz} \\ R_{yx} & R_{yy} & R_{yz} \\ R_{zx} & R_{zy} & R_{zz} \end{matrix} & \begin{matrix} x \\ y \\ z \end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

向き (Rotation matrix) and 位置 (Position vector) are indicated by dashed boxes around the rotation and translation parts respectively.



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

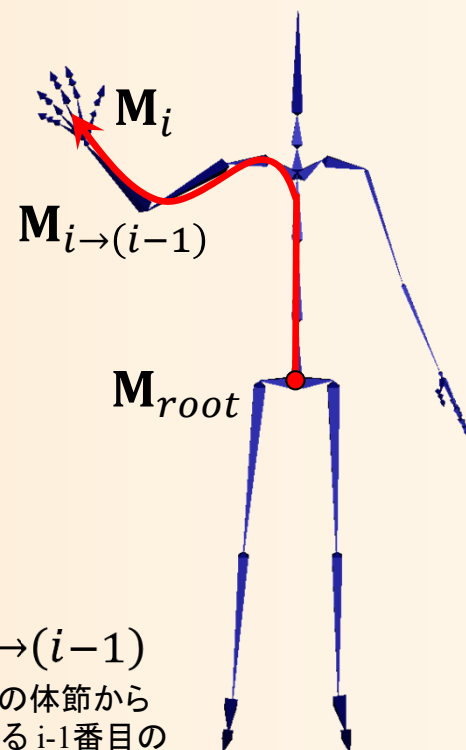
- 変換行列の計算方法

- 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

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

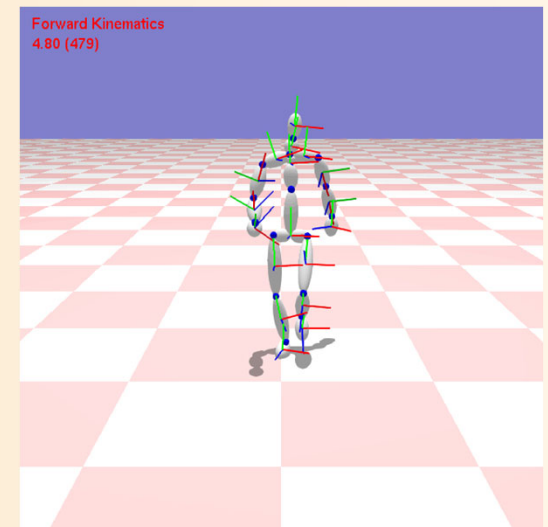
ルート体節の
位置・向き

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



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

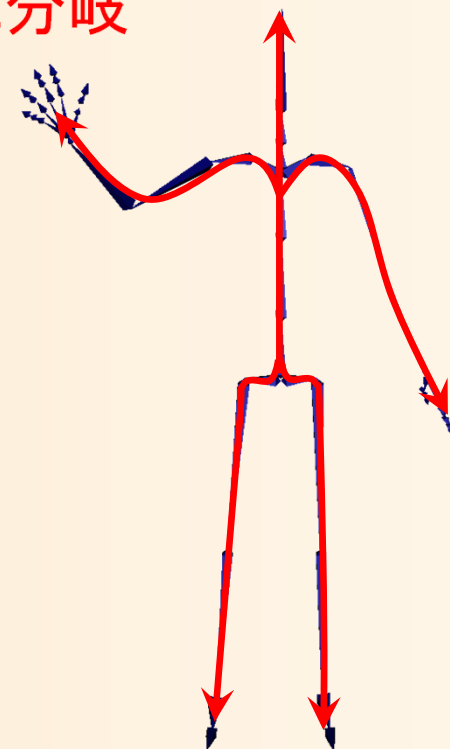
- 前の体節の位置・向きに、

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)

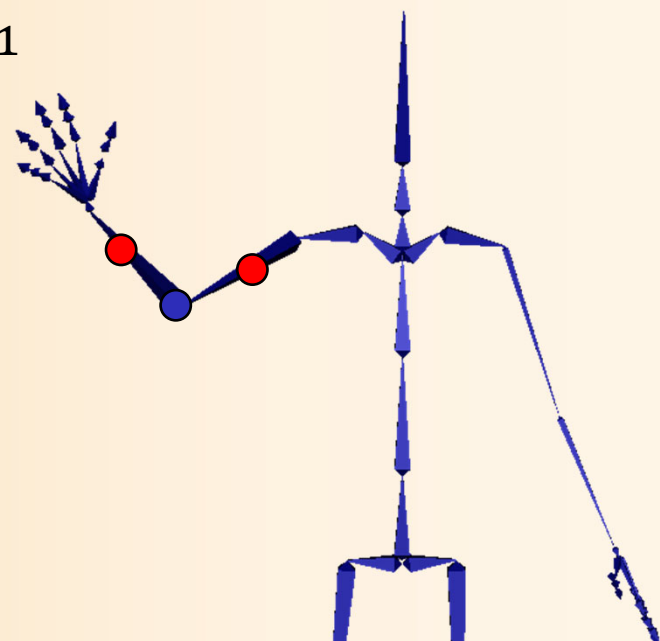
を順番に適用 $\mathbf{T}_{j \rightarrow i}$

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

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)

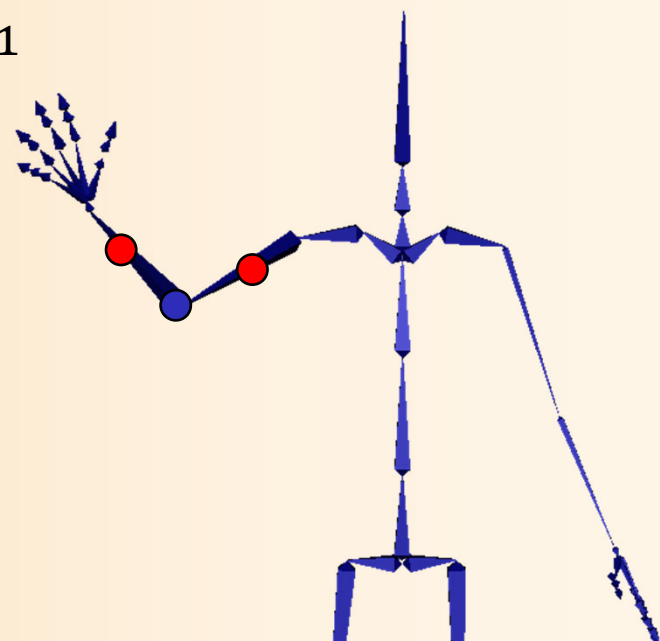
2. 関節の回転 R_j $T_{(i-1) \rightarrow j}$

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)

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

①

②

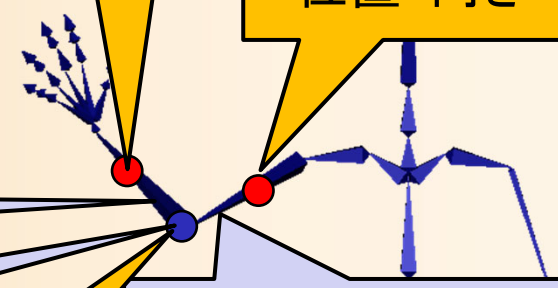
関節の位置

1. 前の体節の中心 → 関節への平行移動(前の体節の座標系)

2. 関節回転(姿勢)

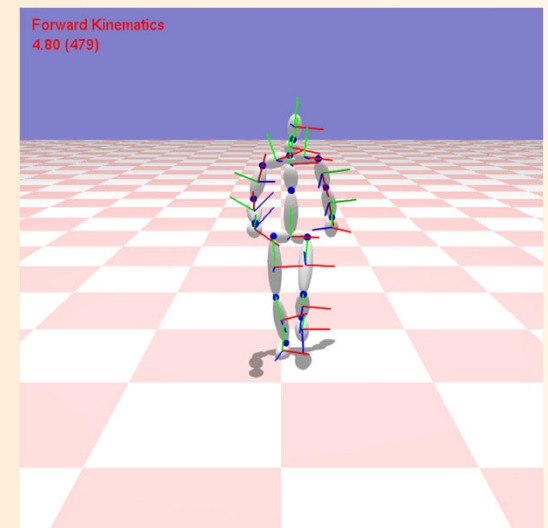
次の体節の位置・向き

前の体節の位置・向き



プログラミング演習

- サンプルプログラム (human_sample.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

- 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



順運動学計算のプログラミング

- 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。
 - 現在の体節(の中心)の位置・向きを取得 ①
 - 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
 - 次の関節の回転をかける ③
 - 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
 - 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 4×4 変換行列 **frame**
 - 計算用の 4×4 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(3×3 行列)を、 4×4 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

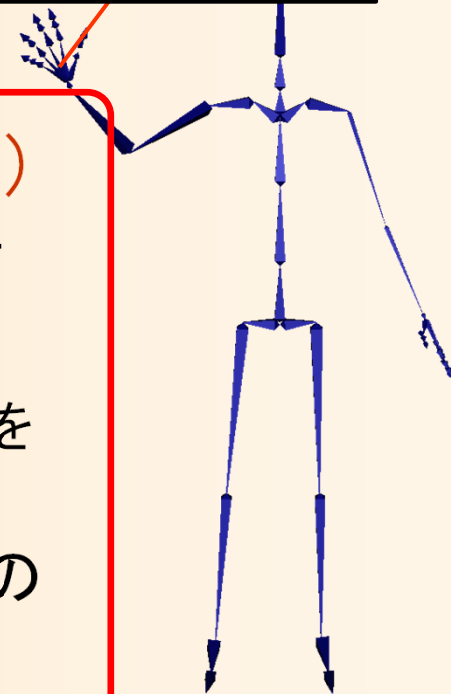
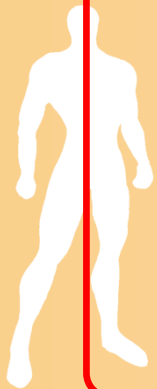
- 順運動学(フォワード・キネマティクス)

- 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)

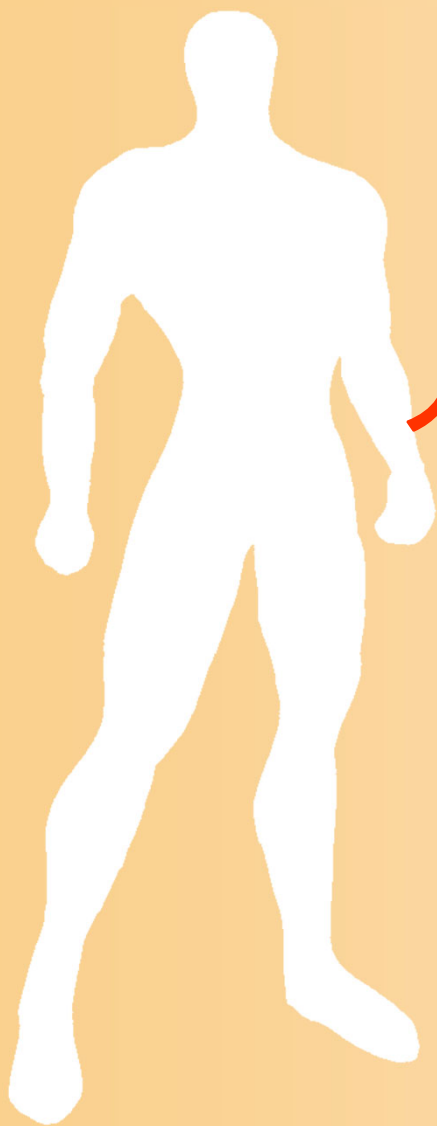
- 指定部位の目標の位置・向きから、多関節体の関節回転の変化を計算
 - 手先などの移動・回転量が与えられた時、それを実現するための関節回転の変化を計算する
- 姿勢を指定する時、関節回転よりも、手先の位置・向きなどを使った方がやりやすい
- ロボットアームの軌道計画等にも用いられる



今日の内容

- 前回の復習
- BVH動作データの読み込みと再生
- サンプルプログラム
- 順運動学
- 人体形状変形モデル

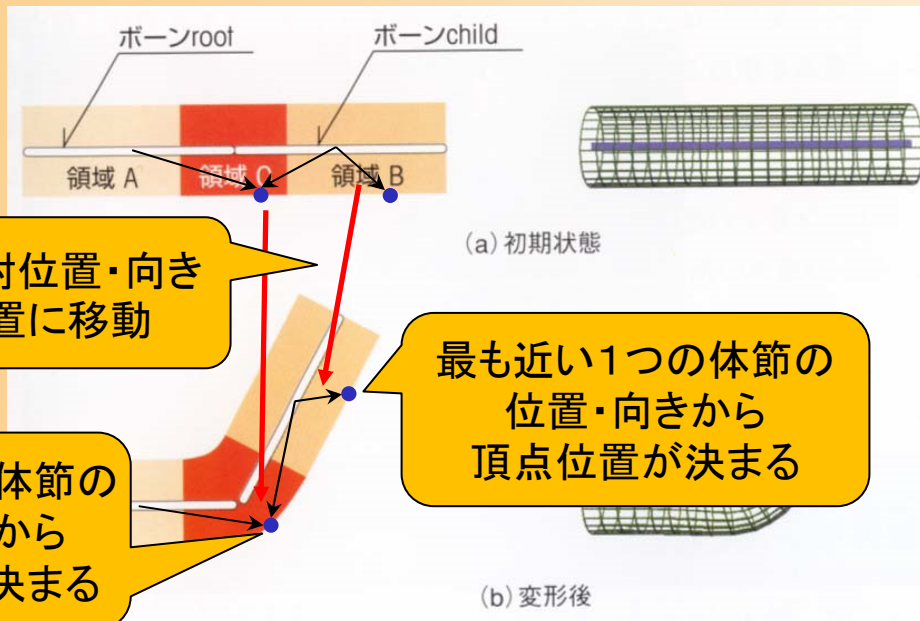




人体形状変形モデル

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

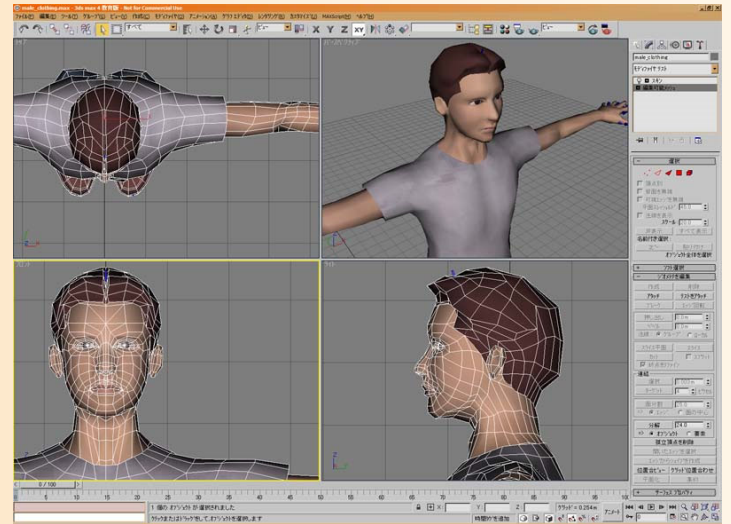
```
// ワンスキンモデルを表す構造体
struct OneSkinModel
{
    // 骨格情報
    Skeleton * skeleton;
    // 幾何形状情報
    Obj * skin_shape;
    // 変形のためのウェイト情報
    float ** weights; // [頂点番号][体節番号] の2次元配列
    // 初期姿勢での各体節の変換行列の逆行列
    Matrix4f * init_seg_frames; // [体節番号]
};
```

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



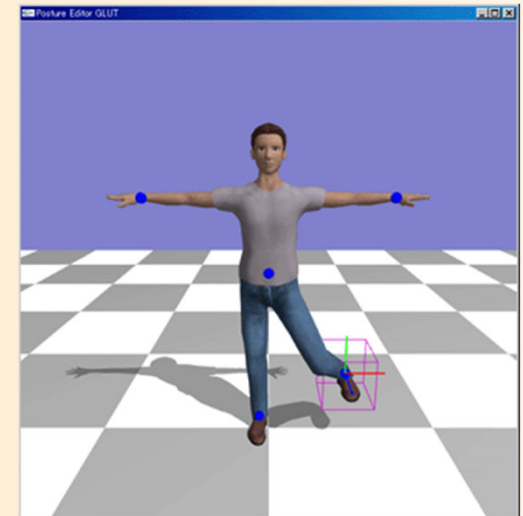
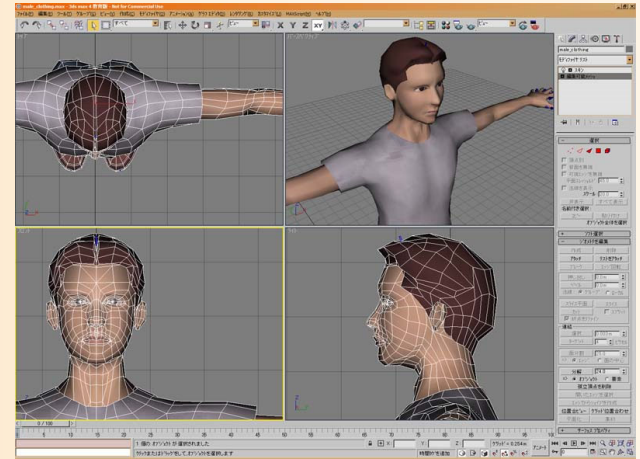
形状変形モデルの出力

- 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

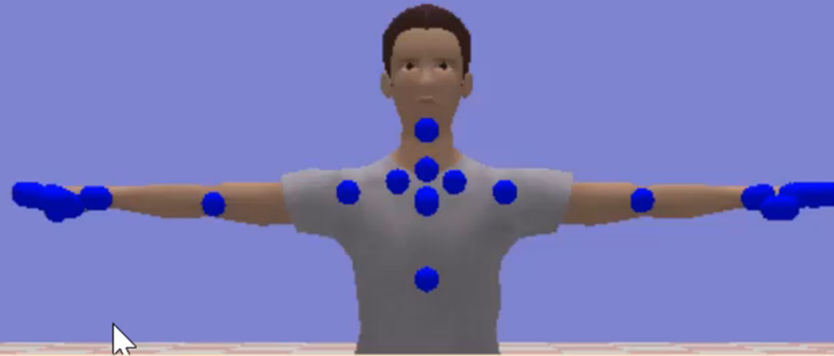


デモプログラム

- 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

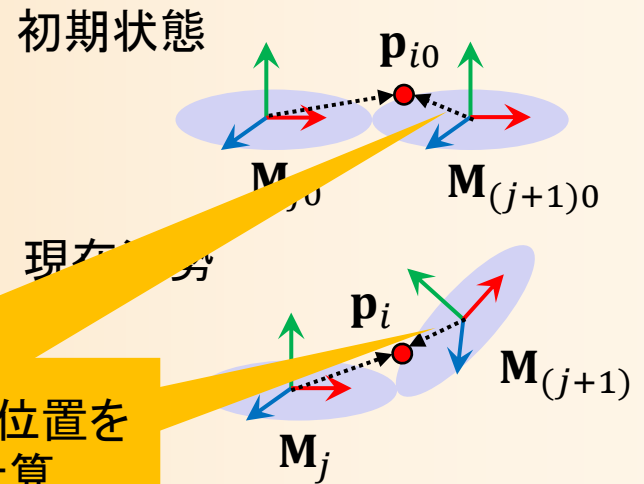
- 幾何形状データに対応する、骨格構造＋初期姿勢の情報が必要
 - 両者の位置を合わせる必要がある
 - 初期姿勢における各体節の位置・向きが必要
 - 形状変形の計算では、各体節の初期姿勢での位置・向きを表す変換行列の、逆行列を使用
 - 順運動学計算により求める



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 (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

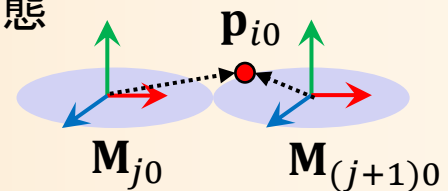
\mathbf{M}_j 各体節の変換行列(現在の姿勢から計算)

w_{ij} 各頂点が各体節から受ける重み

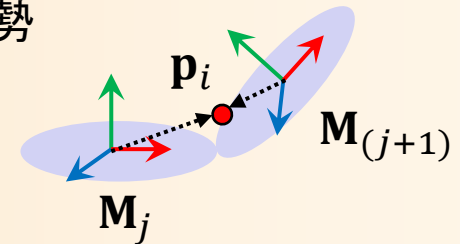
\mathbf{p}_{i0} 初期状態での各頂点の位置

\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

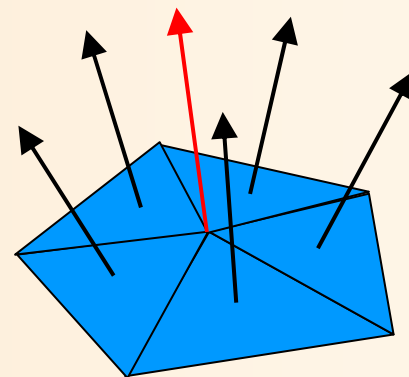


形状変形モデルの変形方法(2)

- 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
- BVH動作データの読み込みと再生
- サンプルプログラム
- 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. 動作接続・遷移

6. 動作変形

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
- 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

