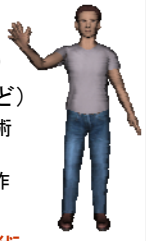
コンピュータグラフィックス特論 II

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

キャラクタ・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）のアニメーションを実現するための技術
- キャラクタ・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
- 逆運動学
- 姿勢補間
- 動作接続・遷移・補間
- 動作変形・生成・制御

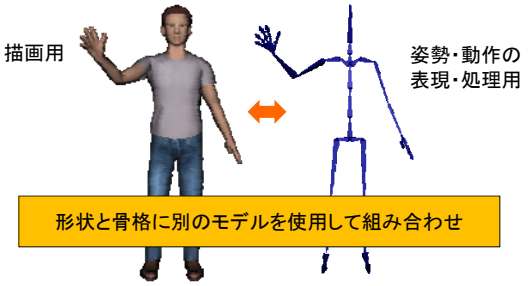
今日の内容

- 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
- 運動学
  - 順運動学
- ワンスキンモデル

前回の復習

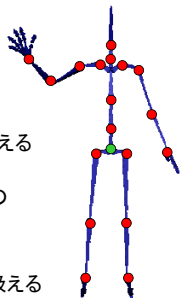
人体モデルの表現

形状モデル（ポリゴンモデル）　骨格モデル（多関節体）



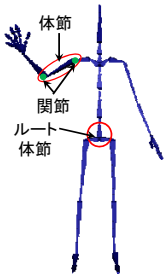
骨格モデルの表現

- 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、体節と関節を分ける

```
// 人体モデル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;
};

// 人体モデルの関節を表す構造体
struct  Joint
{
    // 接続体節
    Segment *              segments[ 2 ];
};
```

```
// 人体モデル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

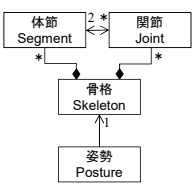
- 骨格情報と姿勢情報を分ける

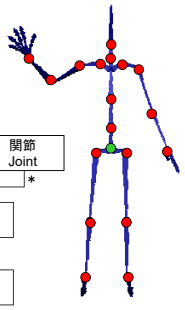
```
// 人体モデルの姿勢を表す構造体
struct  Posture
{
    Skeleton *  body;
    Point3f       root_pos;        // ルートの位置
    Matrix3f      root_ori;        // ルートの向き（回転行列表現）
    Matrix3f *  joint_rotations; // 各関節の回転（回転行列表現）
                                   // [リンク番号] リンク数分の配列
};
```

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
```





キャラクタ・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

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、先述の表現方法(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
    - 各関節が持つ回転軸（＋ルート関節の座標軸）を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列挙
- BVHのサンプルデータ例
  - Eyes, Japan http://www.mocapdata.com/

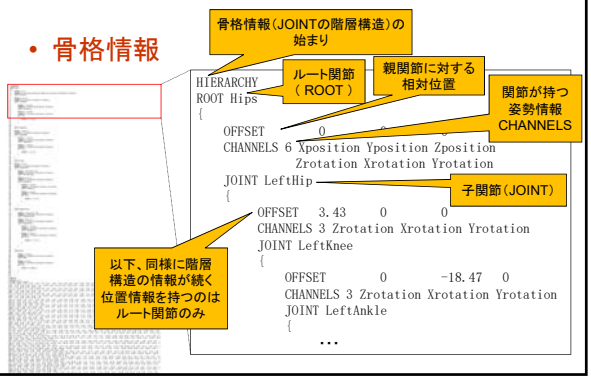
BVH形式の例（1）

- 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
            {
                ...
```

BVH形式の例（1）

- 骨格情報



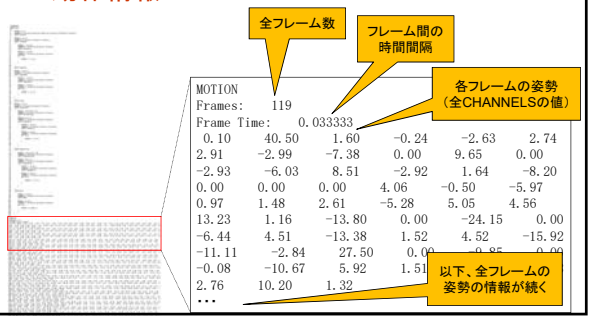
BVH形式の例（2）

- 動作情報

```
MOTION
Frames:    119
Frame Time:    0.033333
 0.10     40.50     1.60     -0.24     -2.63     2.74
2.91     -2.99     -7.38     0.00     9.65     0.00
-2.93     -6.03     8.51     -2.92     1.64     -8.20
0.00     0.00     0.00     4.06     -0.50     -5.97
0.97     1.48     2.61     -5.28     5.05     4.56
13.23     1.16     -13.80     0.00     -24.15     0.00
-6.44     4.51     -13.38     1.52     4.52     -15.92
-11.11     -2.84     27.50     0.00     -9.85     0.00
-0.08     -10.67     5.92     1.51     -1.19     -4.58
2.76     10.20     1.32
...
```

BVH形式の例(2)

- 動作情報



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
  - BVH動作ファイルに記述されている骨格を表示
  - 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

サンプルプログラム

- 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;

      // 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

骨格情報（JOINTの階層構造）

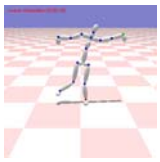
動作情報

BVHファイルを読み込み

サンプルプログラム

デモプログラム

- 複数のデモを含むプログラム
  - 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
- 逆運動学(CCD-IK)
- 姿勢補間
- 動作補間(2つの動作の補間)
- 動作接続・遷移



サンプルプログラム

- デモプログラムの一部(human_sample.cpp)
  - 骨格・姿勢のデータ構造定義(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(GLUTBaseApp.h/cpp)
    - 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
  - メイン処理、コールバック関数(GLUTBaseApp.cpp)
    - 全アプリケーションを管理・切替、イベント呼び出し
  - 各デモの主要な処理は各自で実装
    - 全アプリケーションの実装は

サンプルプログラム

- デモプログラムの一部(human_sample.cpp)
  - 骨格・姿勢のデータ構造定義(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(GLUTBaseApp.h/cpp)
    - 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
  - メイン処理、コールバック関数(GLUTBaseApp.cpp)
    - 全アプリケーションを管理、切替
    - 実行中のアプリケーションのイベント処理を呼び出し
  - 各デモの主要な処理は各自で実装

サンプルプログラム

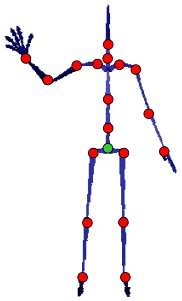
- デモプログラムの一部
  - 骨格・姿勢のデータ構造定義(SimpleHumn.h/cpp)
  - BVH動作クラス(BVH.h/cpp)
  - アプリケーションの基底クラス(GLUTBaseApp.h/cpp)
    - 各イベント処理のためのメソッドの定義を含む
    - 本クラスを派生させて各アプリケーションクラスを定義
  - メイン処理、コールバック関数(GLUTBaseApp.cpp)
    - 全アプリケーションを管理、切替
    - 実行中のアプリケーションのイベント処理を呼び出し
  - 各アプリケーションの実装(human_sample.cpp)
    - 主要な処理を各自で実装(レポート課題)
      - 課題提出をしやすいように全アプリケーションの実装を一つのファイルに集約

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)

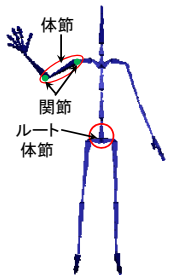
```
// 多関節体の体節を表す構造体
struct Segment

// 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動作クラス(BVH.h/cpp)



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

```
// 人体モデル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;
};

// 人体モデルの関節を表す構造体
struct  Joint
{
    // 接続体節
    Segment *             segments[ 2 ];
};

// 人体モデルの骨格を表す構造体
struct  Skeleton
{
    // 体節・関節の配列
    int                   num_segments;
    Segment **            segments;
    int                   num_joints;
    Joint **              joints;
};
```

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

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

※ IK計算などで、体節・関節
を順番に辿りながら処理をする
ときのために、ルートがどちら
側を判断するためのルールや
情報があった方が良い

姿勢の表現方法

```
// 人体モデル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 *  link_rotations; // 各リンクの相対回転（回転行列表現）
                                // [リンク番号] リンク数分の配列
};
```

動作の表現方法

```
// 人体モデルの動作を表す構造体
struct  Motion
{
    // 骨格モデル
    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

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
  - BHVデータ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
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, 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、ルートのBVH関節からは生成しない
      - BVH関節数＝体節数＝関節数＋1 となる

動作データの生成

- BVH動作の動作情報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

アプリケーションの基底クラス

- GLUTBaseApp

```
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 );
};
```

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;

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

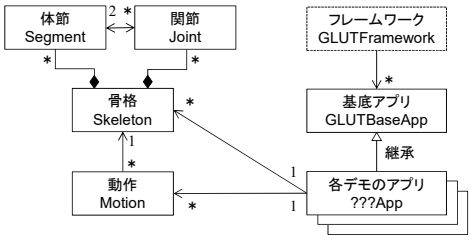
上のリストの中の、現在実行中のアプリケーションを表す

- 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();
   ・・・
}
```

クラス図

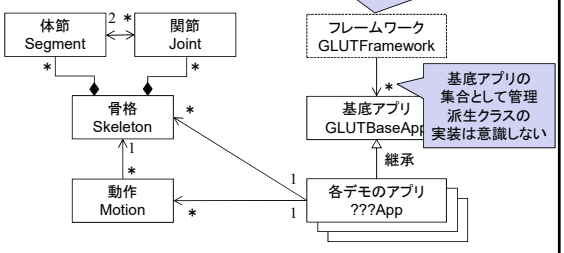
- クラス・構造体間の関係



クラス図

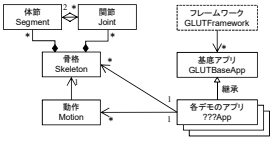
- クラス・構造体間の関係

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



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

動作再生アプリケーション

- MotionPlaybackApp
  - 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

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 delta 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

レポート課題

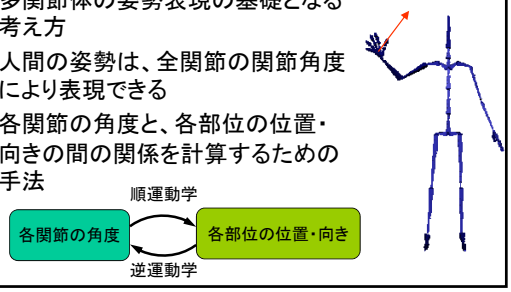
- キャラクタ・アニメーション基礎技術
  - サンプルプログラム（human_sample.cpp）の未実装部分を作成

1. 順運動学計算
2. 姿勢補間
3. キーフレーム動作再生
4. 動作接続・遷移
5. 動作補間
6. 逆運動学計算（CCD法）

順運動学

運動学

- 運動学（キネマティックス）
  - 多関節体の姿勢表現の基礎となる考え方
  - 人間の姿勢は、全関節の関節角度により表現できる
  - 各関節の角度と、各部位の位置・向きの間の関係を計算するための手法



順運動学と逆運動学

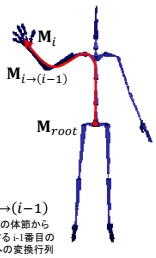
- 順運動学（フォワード・キネマティックス）
  - 多関節体の関節角度から、各部位の位置・向きを計算
  - 回転・移動の変換行列の積により計算
- 逆運動学（インバース・キネマティックス）
  - 指定部位の位置・向きから、多関節体の関節角度を計算
    - 手先などの移動・回転量が与えられた時、それを実現するための関節角度の変化を計算
  - 動きを指定する時、関節角度よりも、手先の位置・向きなどを使った方がやりやすい
  - ロボットアームの軌道計画等にも用いられる

順運動学の計算方法

- フォワード・キネマティックス（順運動学）
  - 各体節（or 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 $\mathbf{M}_i$
  - 変換行列の定義・計算方法
    - i 番目の体節からルート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

$$\mathbf{M}_i = \mathbf{M}_{root}\,\mathbf{M}_{1\to root}\cdots\mathbf{M}_{(i-1)\to(i-2)}\,\mathbf{M}_{i\to(i-1)}$$

（$\mathbf{M}_{root}$：ルート体節の位置・向き／$\mathbf{M}_{i\to(i-1)}$：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）



順運動学の計算方法

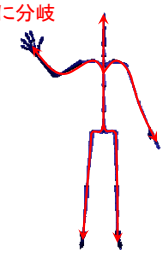
- フォワード・キネマティックス（順運動学）
  - 姿勢（腰の位置・向き、各関節の回転）から、各体節・関節の位置・向きを計算
  - 繰り返し計算
    - ルートから末端に向かって繰り返し
      - 複数の子関節がある場合は各方向に分岐
      - 再帰呼び出しを使うと実装しやすい
    - 前の体節の位置・向きを表す変換行列に、
      1. 次の関節への移動（・回転）
      2. 関節の回転
      3. 次の体節への移動（・回転）

      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
  - 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

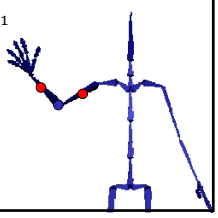


順運動学の計算方法

- 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）$\mathbf{T}_{(i-1)\to i}$
    2. 関節の回転 $\mathbf{R}_j$
    3. 次の体節への移動（・回転）$\mathbf{T}_{(i-1)\to i}$

    を順番に適用

$$\mathbf{M}_i = \mathbf{M}_{i-1}\,\underset{①}{\mathbf{T}_{(i-1)\to i}}\,\underset{②}{\mathbf{R}_j}\,\underset{③}{\mathbf{T}_{(i-1)\to i}}$$



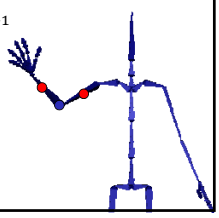
順運動学の計算方法

- 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）$\mathbf{T}_{(i-1)\to i}$
    2. 関節の回転 $\mathbf{R}_j$
    3. 次の体節への移動（・回転）$\mathbf{T}_{(i-1)\to i}$

    を順番に適用

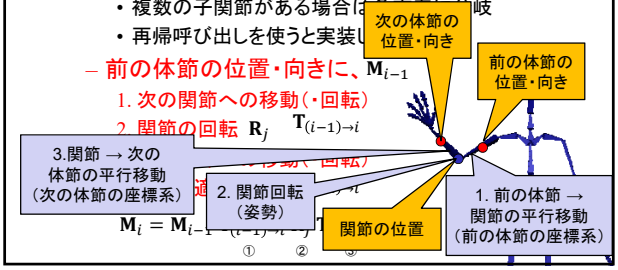
骨格情報から取得
姿勢情報から取得
骨格情報から取得

$$\mathbf{M}_i = \mathbf{M}_{i-1}\,\underset{①}{\mathbf{T}_{(i-1)\to i}}\,\underset{②}{\mathbf{R}_j}\,\underset{③}{\mathbf{T}_{(i-1)\to i}}$$



順運動学の計算方法

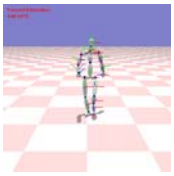
- 繰り返し計算
  - ルートから末端に向かって繰り返し
    - 複数の子関節がある場合は各々を分岐
    - 再帰呼び出しを使うと実装しやすい
  - 前の体節の位置・向きに、$\mathbf{M}_{i-1}$
    1. 次の関節への移動（・回転）
    2. 関節の回転 $\mathbf{R}_j$ $\mathbf{T}_{(i-1)\to i}$



$\mathbf{M}_i = \mathbf{M}_{i-1}$ ① ② ③

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の実装
  - 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・緑の線分で可視化
  - 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）
  - 出力：各体節の位置・向き ＋ 各関節の位置
    - STLの可変長配列を使用
    - 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() );
}
```

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
void  MyForwardKinematicsIteration( 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
    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
        ForwardKin                    ・・・ );
    }
```

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・向きの配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

末端側の隣接する
関節・体節に向かっ
て繰り返し

再帰呼び出し
による繰り返し

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

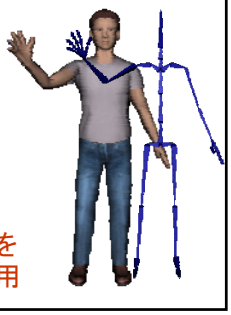
順運動学計算の繰り返し処理

- 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
  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)
  3. 次の関節の回転をかける
  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)
  5. 次の体節に対して再帰呼び出し

ワンスキンモデル

形状モデルの表現(復習)

- 人体形状(ワンスキンモデル)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×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モデルを利用



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]
};
```

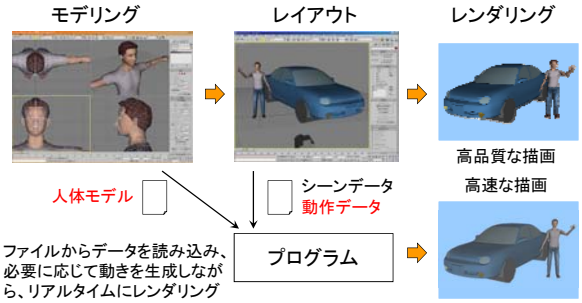
初期状態の姿勢
から計算

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
ソフトウェアを使用して
デザイナが作成
- 自分のプログラムで使用するときには、アニメーションソフトウェアからファイルに出力したものを読み込んで使用

市販ソフトウェアの利用（復習）

- 既存ソフトウェアと組み合わせたプログラミング



ワンスキンモデルの出力

- 一般的に仕様が公開されているワンスキンモデルのファイル形式は少ないので、適当な独自形式を使うのが一般的
  - FBX、Collada、X などは、ワンスキンモデルも表現可能
- 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
  - 骨格構造の各リンクから幾何形状の各頂点へのウェイト
    → ソフトによってはテキスト形式で出力可能

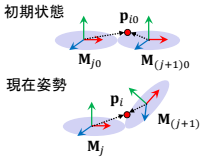
ワンスキンモデルの情報

- 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

ワンスキンモデルの変形方法

- 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

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

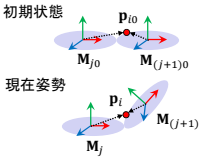


$\mathbf{p}_i$ 各頂点の位置
$\mathbf{M}_j$ 各体節の変換行列（現在の姿勢から計算）
w_{ij} 各頂点が各体節から受けるウェイト
$\mathbf{p}_{i0}$ 初期状態での各頂点の位置
$\mathbf{M}_{j0}$ 初期状態での各体節の変換行列

ワンスキンモデルの変形方法

- 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
$\mathbf{p}_{i0}$ 初期状態での各頂点
$\mathbf{M}_{j0}$ 初期状態での各体節の変換行列

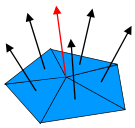
順運動学計算により姿勢から計算

ワンスキンモデルの変形方法

- 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

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

- 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



ワンスキンモデルの変形処理

```
// 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
}
```

デモプログラム

- ワンスキンモデルデモ
  - 形状モデルの変形・描画
    - 市販のアニメーションソフト（3ds max）で作成したキャラクタ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
    - ワンスキンモデルの変形・描画
  - 姿勢変形
    - 関節点をマウスでドラッグすると姿勢を変形
    - 逆運動学計算（後述）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座標・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

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
- 運動学
  - 順運動学
- ワンスキンモデル

レポート課題

- キャラクタ・アニメーション基礎技術
  - サンプルプログラム（human_sample.cpp）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. 動作接続・遷移
5. 動作補間
6. 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
- 逆運動学
- 姿勢補間
- 動作接続・遷移・補間
- 動作変形・生成・制御