コンピュータアニメーション特論

第9回　キャラクタアニメーション（2）

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
- BVH動作データの読み込みと再生
- サンプルプログラム
- 動作再生

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、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
- BVH動作データの読み込みと再生
- サンプルプログラム
- 動作再生

前回の復習

キャラクタ・アニメーションの実現方法

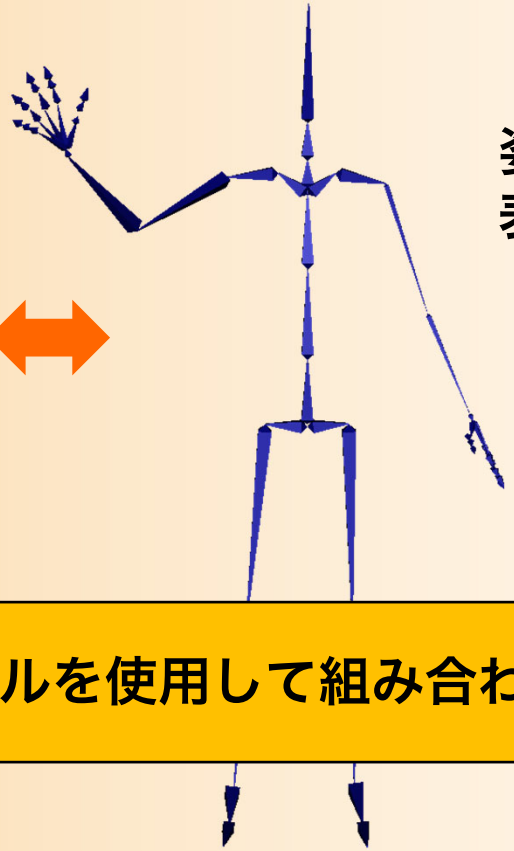
- オフライン・アニメーション制作
  - 通常は市販のアニメーション制作ソフトウェアを利用
- オンライン・アニメーション生成
  - ゲームエンジン（ミドルウェア）の利用
    - Unity, Unreal 等（市販のコンピュータゲーム等でも利用されている）
    - 基本的には、アニメーション制作ソフトウェアで作成されたキャラクタモデルや動作データを再生する機能を提供
    - 提供されている機能以上の高度な動作生成・変形は困難
  - 自分でソフトウェアライブラリを開発
    - 高度な処理も自由に追加できる
    - キャラクタモデルや動作データは、他のソフトウェアで作成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

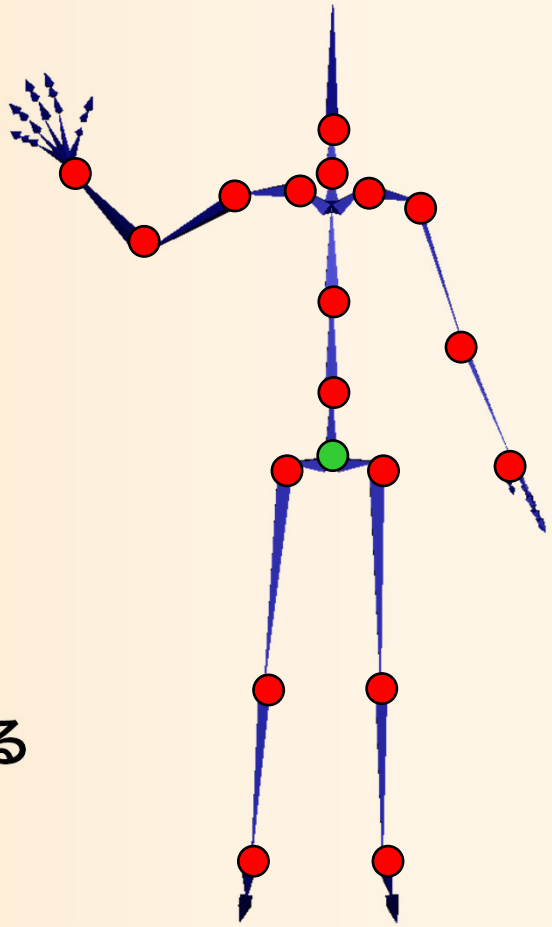


姿勢・動作の表現・処理用

形状と骨格に別のモデルを使用して組み合わせ

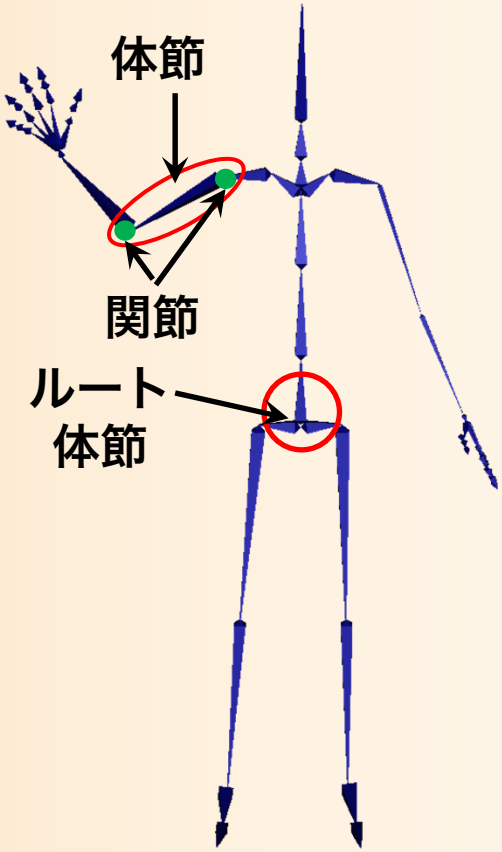
骨格モデルの表現

- 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

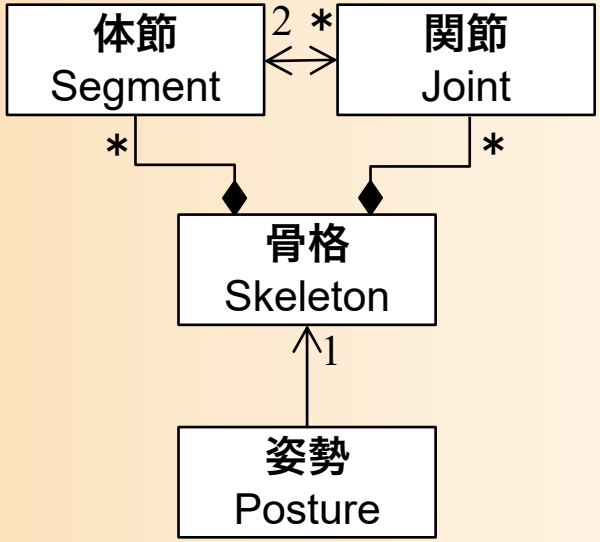
- 骨格情報と姿勢情報のデータ構造を分ける

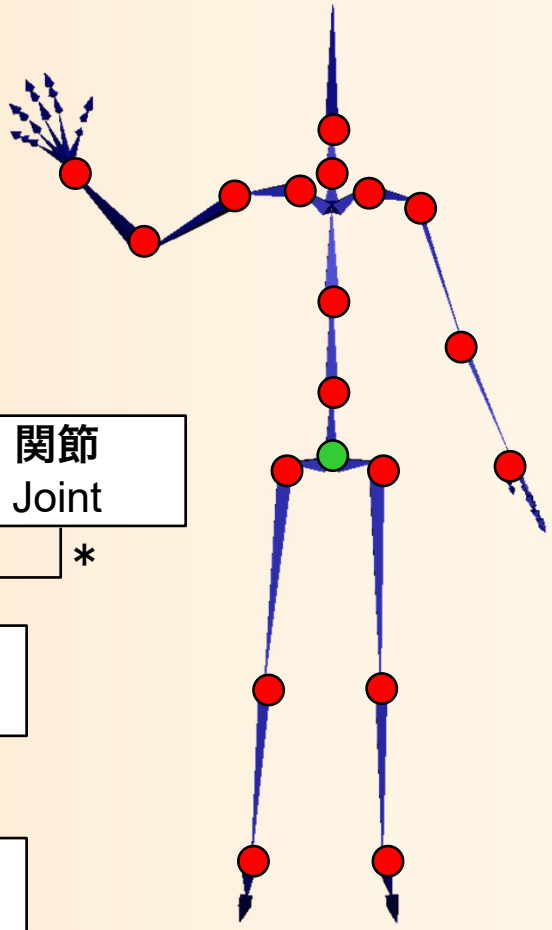
```
// 多関節体の姿勢を表す構造体
struct  Posture
{
    Skeleton *  body;
    Point3f       root_pos;          // ルートの位置
    Matrix3f      root_ori;          // ルートの向き（回転行列表現）
    Matrix3f *    joint_rotations;  // 各関節の回転（回転行列表現）
                                     // [関節番号] 関節数分の配列
};
```

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
  - 公開されているデータは少ない
  - 本科目のプログラミング演習用のBVHファイルを参照

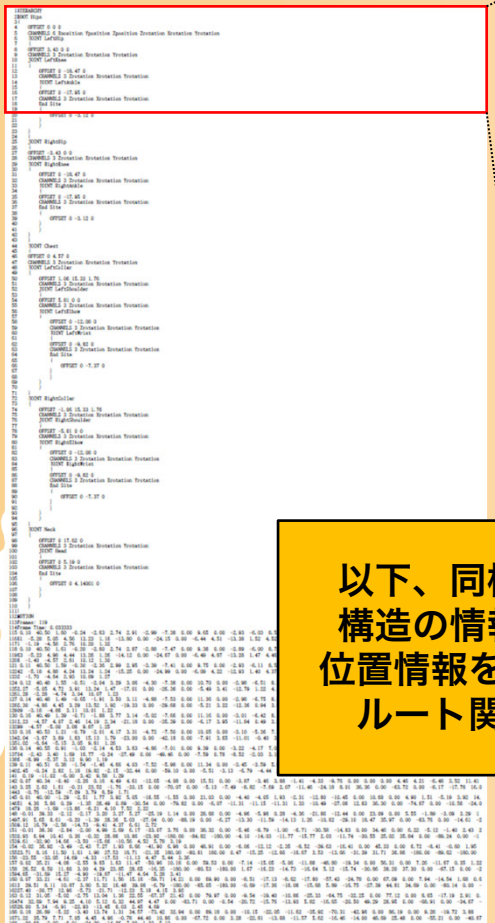
BVH形式の例（1）

- 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



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
    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（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

動作情報の始まり

全フレーム数

フレーム間の時間間隔

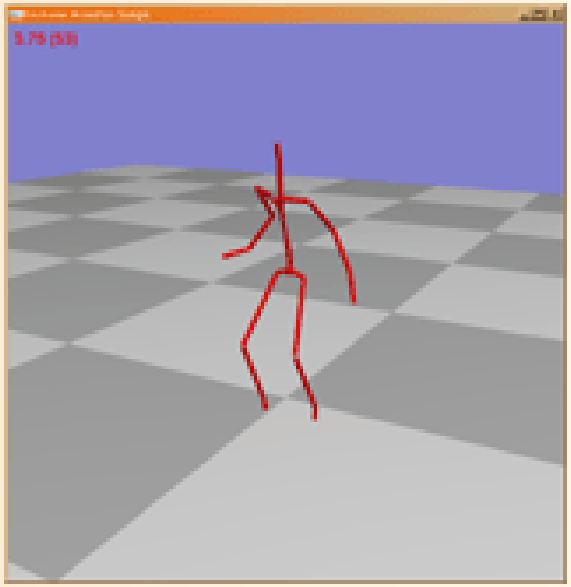
各フレームの姿勢データ
１行が１フレームを表す
骨格情報の全チャンネルの値の組

```
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
...
```

以下、全フレームの姿勢データが続く

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




BVH Player Sample
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;

      // 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（2）

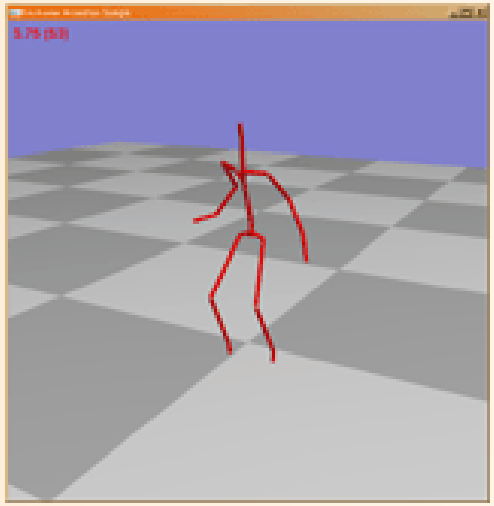
- BVH動作の読み込み処理
  - BVH::BVH(const char * bvh_file_name) 関数
  - iostream を使用
  - 骨格情報の読み込み
    - １行ずつファイルから文字列を読み込み
    - タグに応じて、関節オブジェクトの生成と属性の設定
  - 動作情報の読み込み
    - １行ずつファイルから文字列を読み込み
    - 各フレームの姿勢データの設定

サンプルプログラム（3）

- BVH動作の読み込みと再（bvh_player.cpp）
  - グローバル変数の定義
    - 視点操作のための変数や、BVH動作を表す変数、アニメーション再生の時刻を表す変数
  - メイン関数
    - OpenGL + GLUTの初期化
  - GLUTのコールバック関数
    - キーボードコールバック
      - BVHファイルの読み込み
    - アニメーションコールバック
      - アニメーションの時間を進める
    - 画面描画コールバック
      - BVH動作データの現在時刻の姿勢を描画

サンプルプログラム（4）

- 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
      - BHV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
- 動作再生

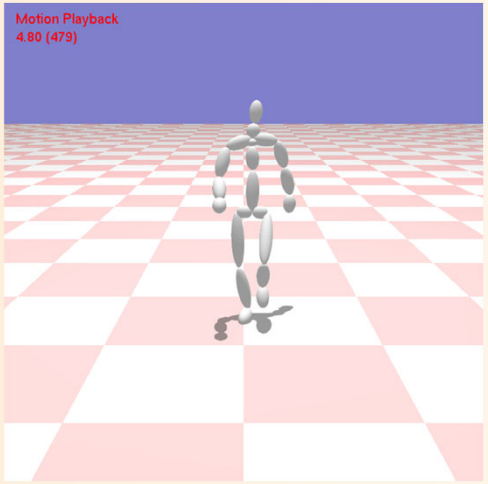
サンプルプログラム

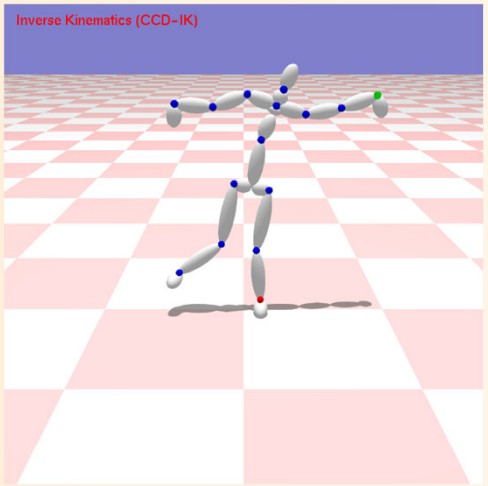
サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

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





キャラクタアニメーション

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）

サンプルプログラムの解説

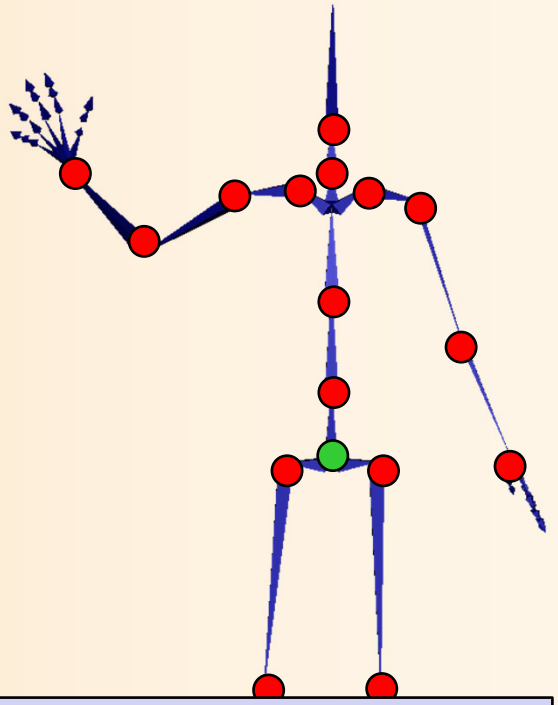
- 以降、サンプルプログラムの基礎を解説
- 各アプリケーションの詳細は、次回以降の講義で解説
- ソースコードを含む詳しい解説は、講義のウェブページの演習資料を参照
  - 今回はサンプルプログラムの規模が大きいので、演習資料を参照しながら、全体を理解する

サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

骨格・姿勢・動作のデータ構造

- 骨格・姿勢・動作の
  構造体・クラスの定義
  （SimpleHuman.h/cpp）

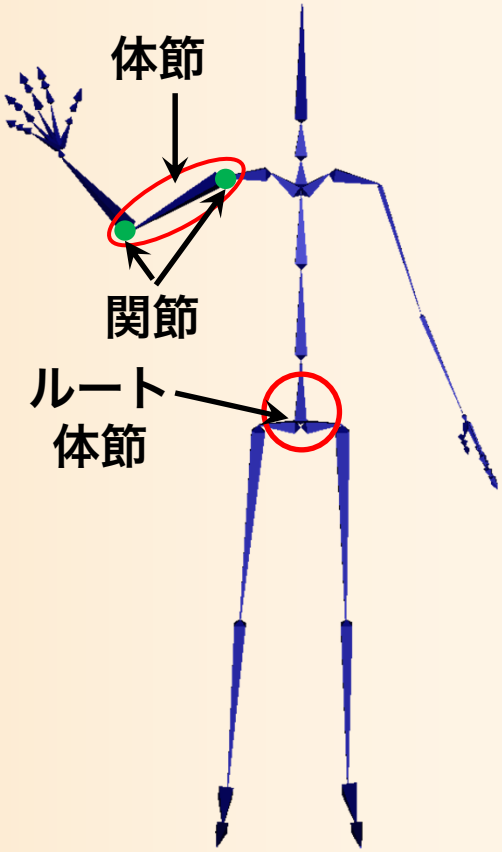


```
// 人体モデルの体節を表す構造体
struct  Segment

// 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

// 人体モデルの動作を表すクラス
class  Motion
```


基本的には構造体の扱い（メンバ変数は public とする）だが、コンストラクタやメソッドが定義できた方が便利であるため、クラスとしている

骨格・姿勢の表現方法

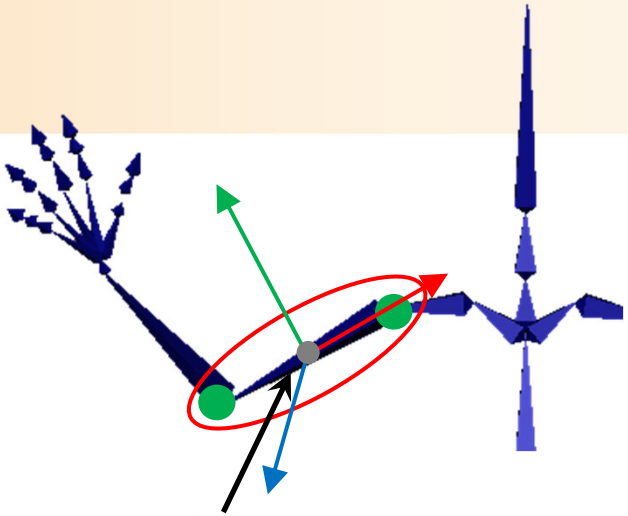
- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける
  - 体節
    - 複数の関節と接続
    - 各関節の接続位置
      - 体節のローカル座標系
  - 関節
    - ２つの体節の間を接続
      - ルート側・末端側の体節



骨格モデルのデータ構造（1）

- 体節のデータ構造

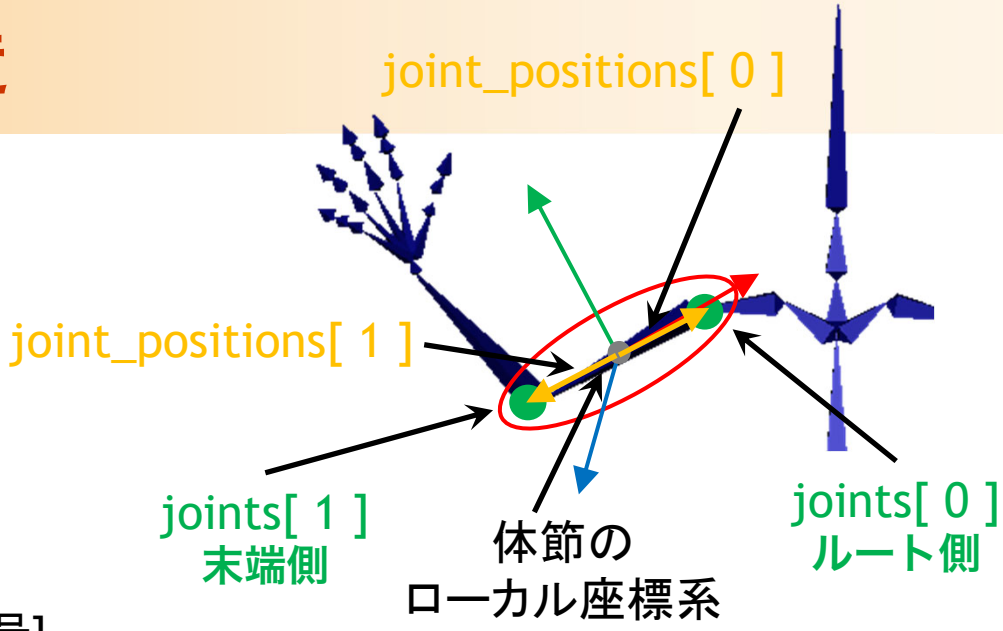
```
// 人体モデルの体節を表す構造体
struct  Segment
{
    // 体節番号・名前
    int             index;
    string          name;

    // 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

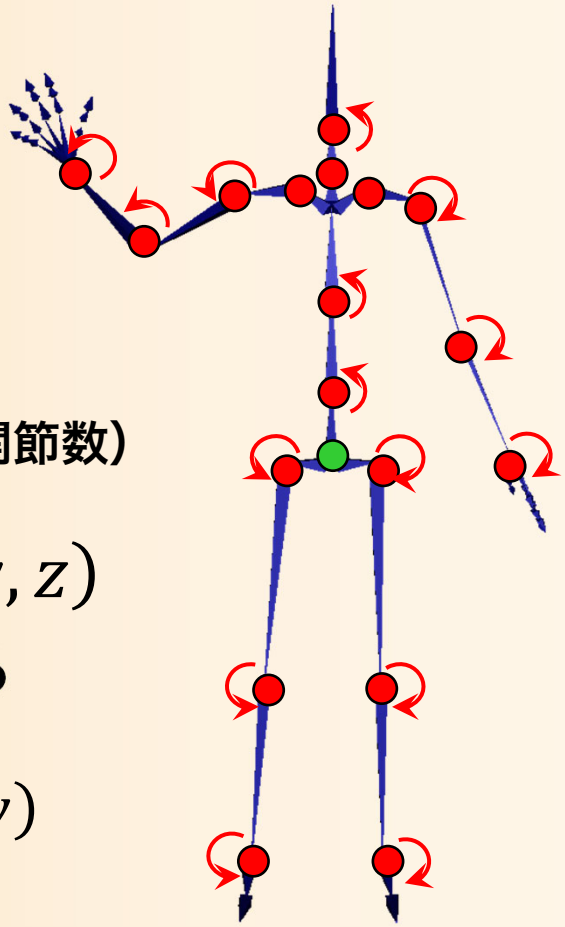
```
// 人体モデルの骨格を表すクラス
class  Skeleton
{
    // 関節数
    int             num_segments;
    // 関節の配列 [関節番号]
    Segment **      segments;

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

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

姿勢データの表現

- 多関節体の姿勢の表現
  - 各関節の回転（40~自由度）
  - 腰の位置・向き（６自由度）

$$\mathbf{p} = (\underbrace{\mathbf{v}_r,\ \mathbf{o}_r,}\ \underbrace{\mathbf{r}_1, \cdots, \mathbf{r}_n})$$

腰の位置・向き　全関節の回転（n は関節数）



  - 位置は３次元ベクトルで表現 (x, y, z)
  - 向き・回転は複数の表現方法がある

$$\begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{pmatrix} \quad (\theta_1, \theta_2, \theta_3) \quad (x, y, z, w)$$

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( Sleleton * b );
    // 初期化
    void  Init( Skeleton * b );
};
```

動作データの表現方法

- **一定間隔動作データ**
  - 一定間隔の全フレームの姿勢データを持つ方法
  - **姿勢データの配列**により表現可能
    - 30〜120 fps 程度の多数の姿勢データが必要

$$\mathbf{p} = \left(\underbrace{\mathbf{v}_r,\ \mathbf{o}_r}_{\text{腰の位置・向き}},\ \underbrace{\mathbf{r}_1, \cdots, \mathbf{r}_n}_{\text{全関節の回転}} \right)$$

$$\mathbf{m} = \left(\mathbf{p}_1, \cdots, \mathbf{p}_m \right) \quad m \text{ はフレーム数}$$

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

動作データの表現方法

- キーフレーム動作データ
  - キーフレームの姿勢データのみを持ち、中間の姿勢は補間によって求める方法
  - （時刻、姿勢データ）の組の配列により表現可能

$$\mathbf{p} = \left(\underbrace{\mathbf{v}_r,\ \mathbf{o}_r}_{\text{腰の位置・向き}},\ \underbrace{\mathbf{r}_1, \cdots, \mathbf{r}_n}_{\text{全関節の回転}} \right)$$

$$\mathbf{m} = \left((t_1, \mathbf{p}_1), \cdots, (t_k, \mathbf{p}_k) \right)$$

k はキーフレーム数

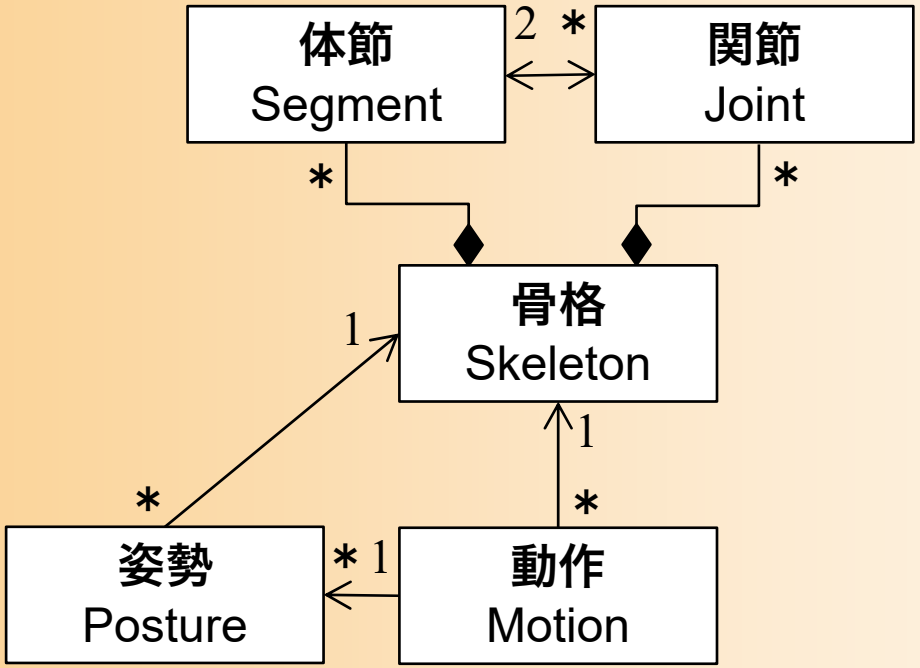
キーフレーム動作のデータ構造

- キーフレーム動作のデータ構造
  - 詳細は後の講義で説明

```
// 人体モデルのキーフレーム動作を表すクラス
class  KeyframeMotion
{
    // 骨格モデル
    const Skeleton *  body;
    // キーフレーム数
    int                 num_keyframes;
    // 各キー時刻の配列［キーフレーム番号］
    float  *            key_times;
    // 各キー姿勢の配列［キーフレーム番号］
    Posture *           key_poses;

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

クラス図

- クラス・構造体間の関係



サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

基本処理関数

- 骨格・姿勢・動作に対する主要な基本処理関数（SimpleHuman.h/cpp）
  - BVH動作から骨格モデルを生成（ConstructBVHSkeleton関数）
  - BVH動作から動作データ（+骨格モデル）を生成（ConstructBVHMotion関数）
  - BVHファイルを読み込んで動作データ（＋骨格モデル）を生成（LoadAndConstructBVHMotion関数）
  - 姿勢の描画（DrawPosture関数）
  - 姿勢の影の描画（DrawPosture関数）

骨格・動作の読み込み

- 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
  - 骨格・動作のデータ構造へ変換

```
// 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 );

// BVHファイルを読み込んで動作データ(＋骨格モデル)を生成
Motion *  LoadAndCoustructBVHMotion( const char * bvh_file_name,
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（ConstructBVHSkeleton関数）
  - 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動作から動作データを生成（ConstructBVHMotion関数）
  - 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、回転行列への変換を行う（GetBVHPosture関数）
    - 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

姿勢の描画

- 姿勢＋影の描画

```
// 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（DrawBone関数）
    - 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
  - 地面に投影した影を描画（光源方向と色を指定）

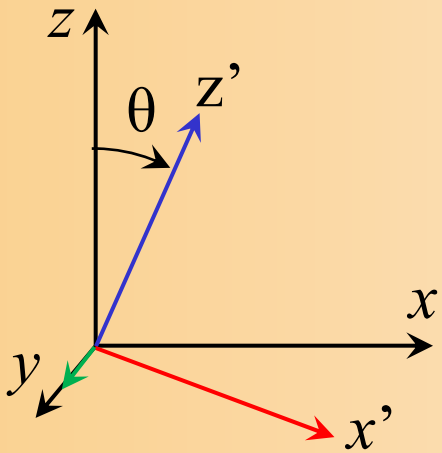
水平方向の向きの変換

- 3自由度の向き（回転行列）と水平方向の向き（1自由度の回転角度）の間の変換
  - Matrix3f 型と float 型の間の変換
  - 水平方向のみ移動や回転がよく用いられるため
    - 具体例は、後日の講義（動作接続など）で説明

```
// 変換行列の水平向き(方位角)成分を計算
// (Z軸の正の方向を0とする時計回りの角度を -180～180の範囲で求める)
float  ComputeOrientationAngle( const Matrix3f & ori );

// 水平回転を表す変換行列を計算
// (Z軸の正の方向を0とする時計回りの角度を -180～180の範囲で指定する)
void  ComputeOrientationMatrix( float angle, Matrix3f & ori );
```

水平方向の向きの変換

- 3自由度の向き（回転行列）と水平方向の向き（1自由度の回転角度）の間の変換
  - Matrix3f 型と float 型の間の変換
  - 水平方向のみ移動や回転がよく用いられるため
    - 具体例は、後日の講義（動作接続など）で説明



相互変換

$$\mathbf{M} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}$$

水平方向の回転行列

$$\theta = \tan^{-1}\frac{\sin\theta}{\cos\theta}$$

水平方向の回転角度

その他の基本処理

- 体節・関節探索、順運動学計算、姿勢補間など
  - 後日の講義で説明

```
// 骨格モデルから体節･関節を名前で探索
int  FindSegment( const Skeleton * body, const char * segment_name );
int  FindJoint( const Skeleton * body, const char * joint_name );

// 順運動学計算
void  ForwardKinematics( const Posture & posture, vector< Matrix4f > &
seg_frame_array, vector< Point3f > & joi_pos_array );

// 姿勢補間（2つの姿勢を補間）
void  PostureInterpolation( const Posture & p0, const Posture & p1,
float ratio, Posture & p );
```

サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

アプリケーションの基底クラス（1）

- 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  Initialize();
    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（2）

- GLUTBaseAppクラス（SimpleHumanGLUT.h/cpp）のメンバ変数
  - アプリケーション情報
    - アプリケーション名（アプリケーション切替メニューに表示）
  - 視点操作のための変数
    - 視点の方位角・仰角、注視点との距離、注視点の位置
  - マウス入力処理のための変数
    - 各ボタンの押下状態、前回のマウスカーソル位置
  - 画面描画に関する変数
    - 光源位置、影の方向・色
  - アプリケーション状態の変数
    - ウィンドウサイズ、初期化フラグ、視点更新フラグ

アプリケーションの基底クラス（3）

- GLUTBaseAppクラス（SimpleHumanGLUT.h/cpp）のイベント処理インターフェース
  - 初期化（Initialize関数）
  - 開始処理（Start関数）
  - 画面描画（Display関数）
  - アニメーション（Animation関数）
  - ウィンドウサイズ変更（Resize関数）
  - マウスクリック（MouseClick関数）、マウスドラッグ（MouseDrag関数）、マウス移動（MouseMotion関数）
  - キー入力（Keyboard関数）、特殊キー入力（KeyboardSpecial関数）

※ 各アプリケーションで必要な処理を実装

アプリケーションの基底クラス（4）

- GLUTBaseAppクラス（SimpleHumanGLUT.h/cpp）のイベント処理インターフェース（1）
  - 初期化（Initialize関数）
    - 最初に一度だけ呼ばれる
    - アプリケーションに必要な変数の生成・初期化を行う
  - 開始処理（Start関数）
    - アプリケーション切替時やリセット時に呼ばれる
    - アニメーション再生開始などの変数の初期化を行う
  - 画面描画（Display関数）
    - キャラクタを含むシーンの描画と表示メッセージの描画を行う
    - 視点操作に応じた視野変換行列の設定、光源の設定、格子模様の床の描画の処理が実装済み

アプリケーションの基底クラス（5）

- GLUTBaseAppクラス（SimpleHumanGLUT.h/cpp）のイベント処理インターフェース（2）
  - アニメーション（Animation関数）
    - 何も操作が行われていない状態でも不定期のタイミングで呼ばれる（GLUTのアイドルコールバック関数から呼ばれる）
    - 前回の処理からの経過時間（秒）が引数として渡される
  - ウィンドウサイズ変更（Resize関数）
    - ウィンドウサイズが変更されたときに呼ばれる
    - OpenGLの射影行列やビューポートの設定の処理が実装済み
    - 通常は派生クラスで処理を追加する必要はない

アプリケーションの基底クラス（6）

- GLUTBaseAppクラス（SimpleHumanGLUT.h/cpp）のイベント処理インターフェース（3）
  - マウスクリック（MouseClick関数）、マウスドラッグ（MouseDrag関数）、マウス移動（MouseMotion関数）
    - マウス操作が行われたときに呼ばれる
    - 現在のマウスカーソルの位置や、マウスクリック関数ではボタンの種類や状態が入力として渡される
    - 基本的な視点操作のための処理が実装済み
  - キー入力（Keyboard関数）、特殊キー入力（KeyboardSpecial関数）
    - キー入力が行われたときに呼ばれる
    - 入力されたキーや、現在のマウスカーソルの位置が渡される

サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

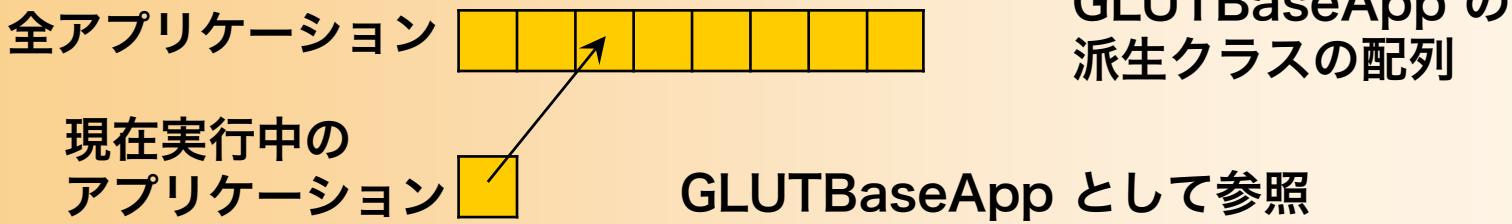
複数アプリケーションの管理（1）

- 複数のアプリケーションの管理

```
// 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;
```

初期化時に、全アプリケーションを生成・登録

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



複数アプリケーションの管理（2）

- 実行するアプリケーションの切り替え
  - 何番目のアプリケーションを実行するかを指定

```
//  実行アプリケーションの切替
void  ChangeApp( int app_no );
```

- 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。

GLUTのコールバック関数

- 画面描画（DisplayCallback関数）
- アニメーション（AnimationCallback関数）
- ウィンドウサイズ変更（ResizeCallback関数）
- マウスクリック（MouseClickCallback関数）
- マウスドラッグ（MouseDragCallback関数）
- マウス移動（MouseMotionCallback関数）
- キー入力（KeyboardCallback関数）
- 特殊キー入力（KeyboardSpecialCallback関数）
  - 現在のアプリケーションのイベント処理を呼び出し
  - アプリケーションの切替やリセットのための操作インターフェースの実装を含む

プログラムの開始処理

- GLUTフレームワークのメイン関数
  - OpenGL・GLUTの初期化
  - 開始時に実行するアプリケーションの初期化
  - アプリケーションの切替メニューの初期化
  - GLUTのメインループに処理を移す

```
//  GLUTフレームワークのメイン関数
//（実行するアプリケーションのリストを指定）
int  SimpleHumanGLUTMain(
  const vector< class GLUTBaseApp * > & app_lists, ... );

//  GLUTフレームワークのメイン関数
//（実行する一つのアプリケーションを指定）
int  SimpleHumanGLUTMain( class GLUTBaseApp * app, ... );
```

単一のアプリケーションを実行する場合に使用できる

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

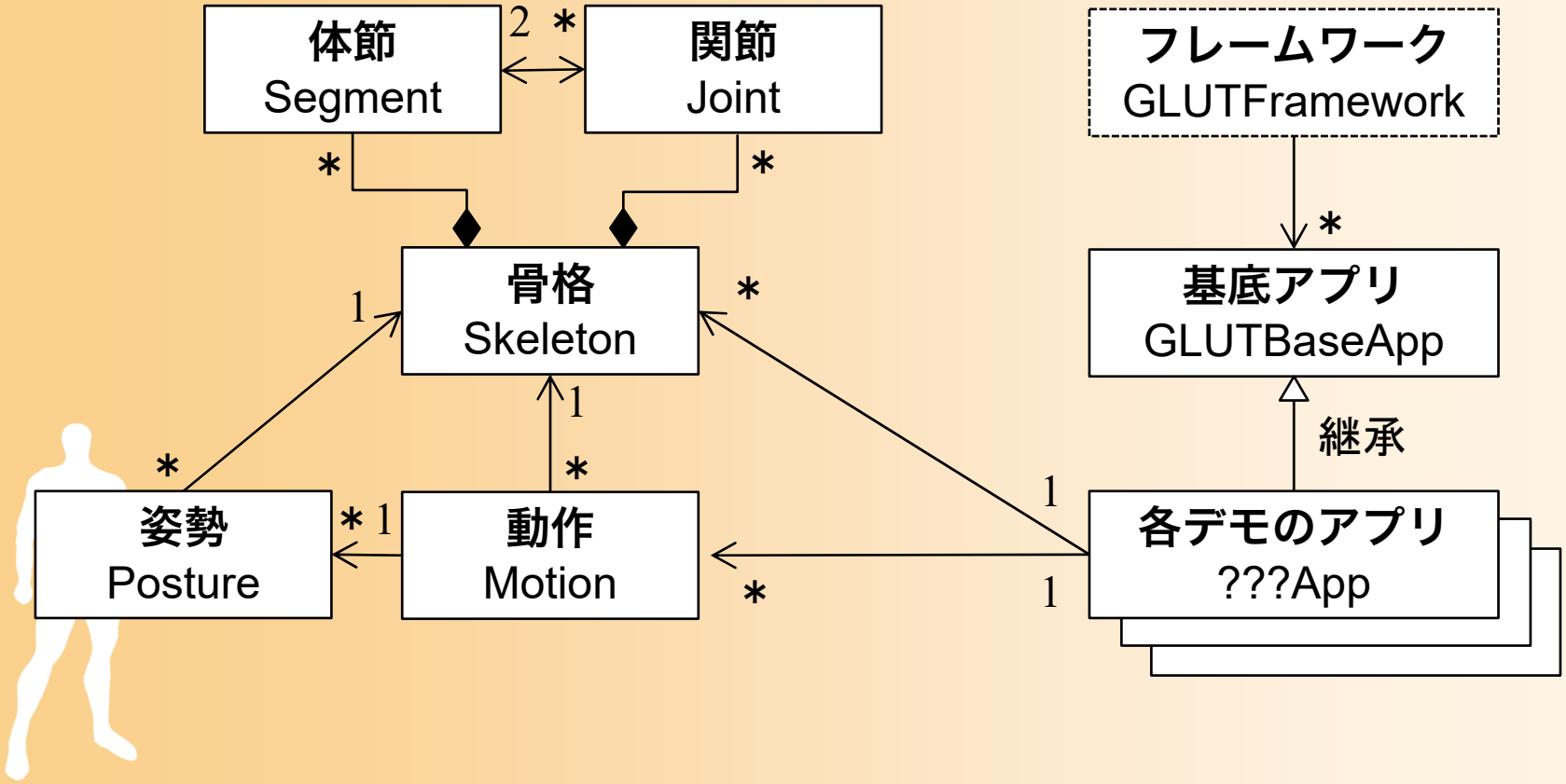
```
int  main( int argc, char ** argv )
{
  // 全アプリケーションのリスト
  vector< class GLUTBaseApp * >    applications;

  // 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 );
};
```

サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

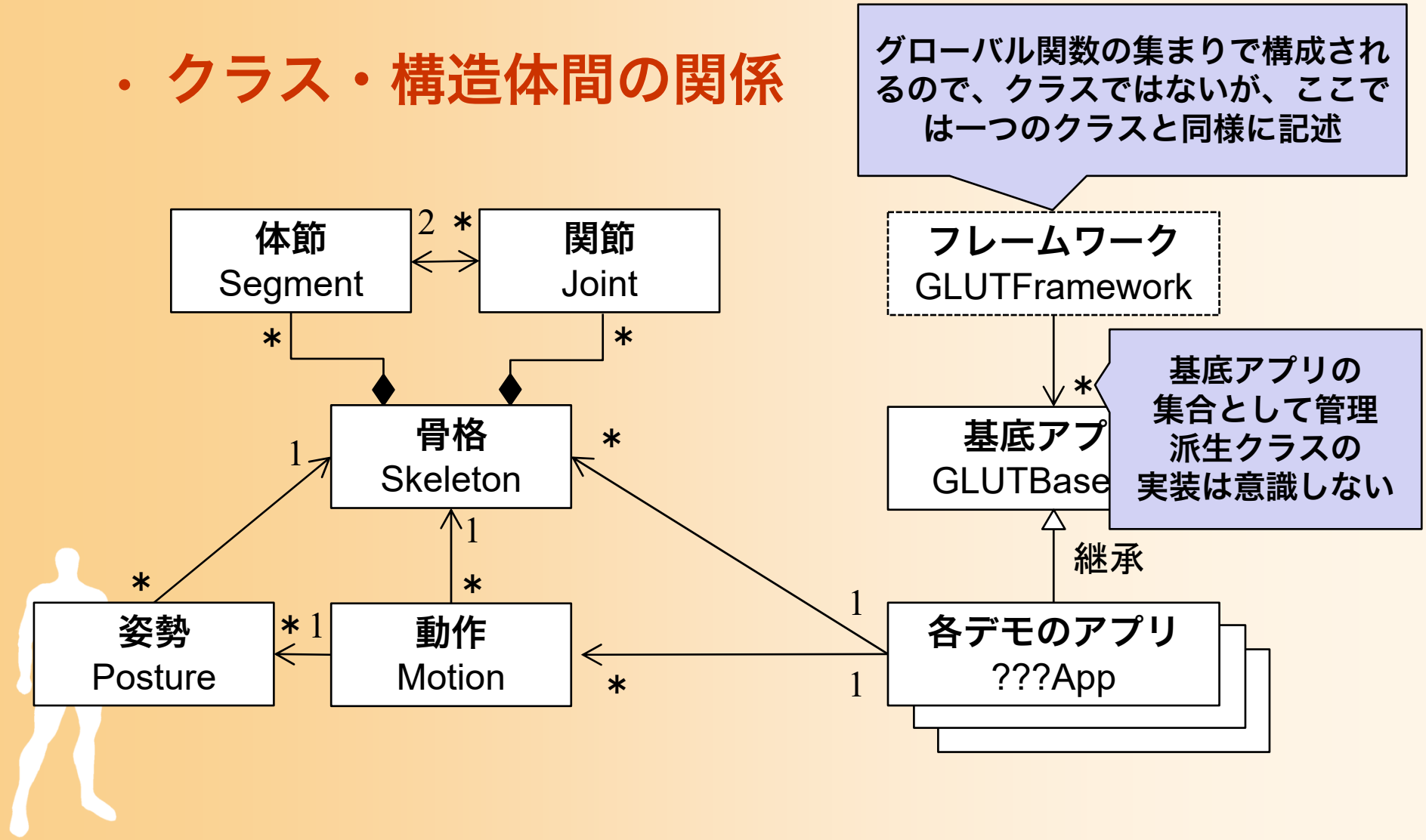
クラス図

- クラス・構造体間の関係



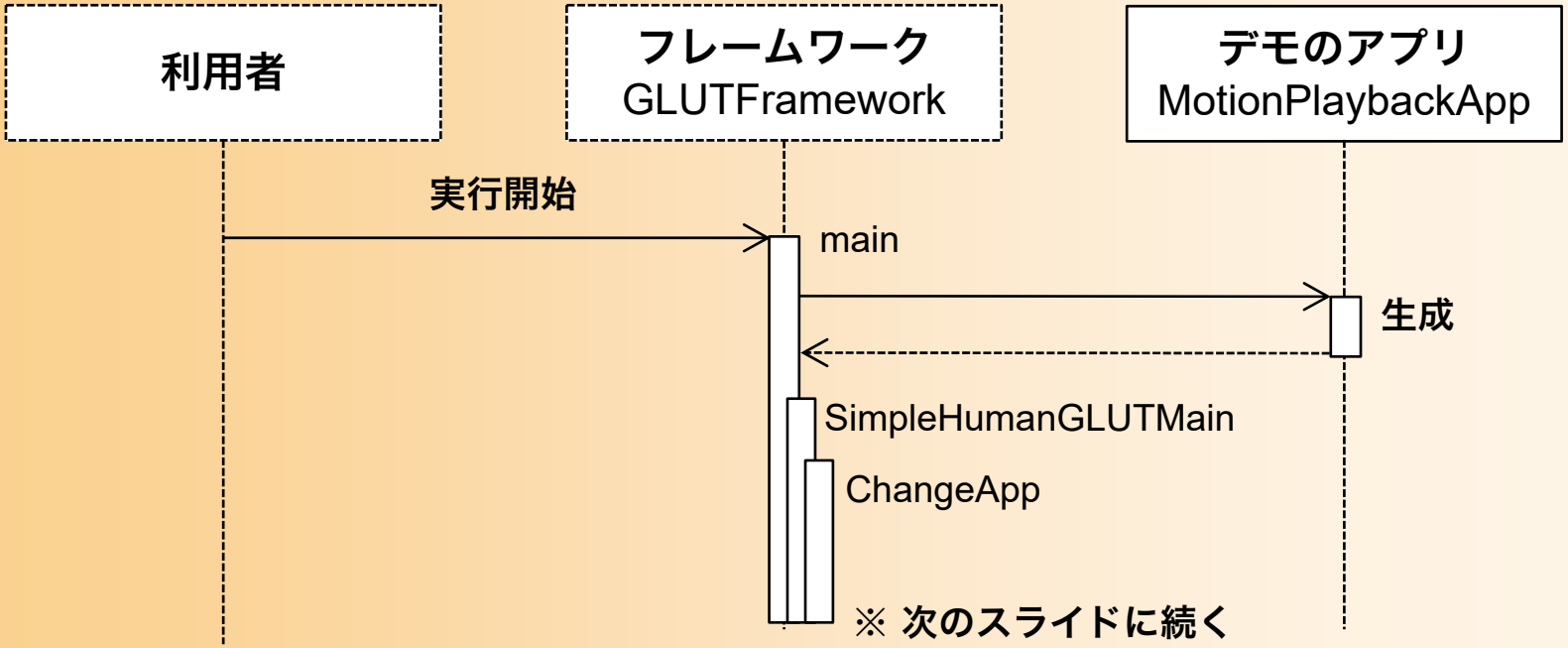
クラス図

・クラス・構造体間の関係

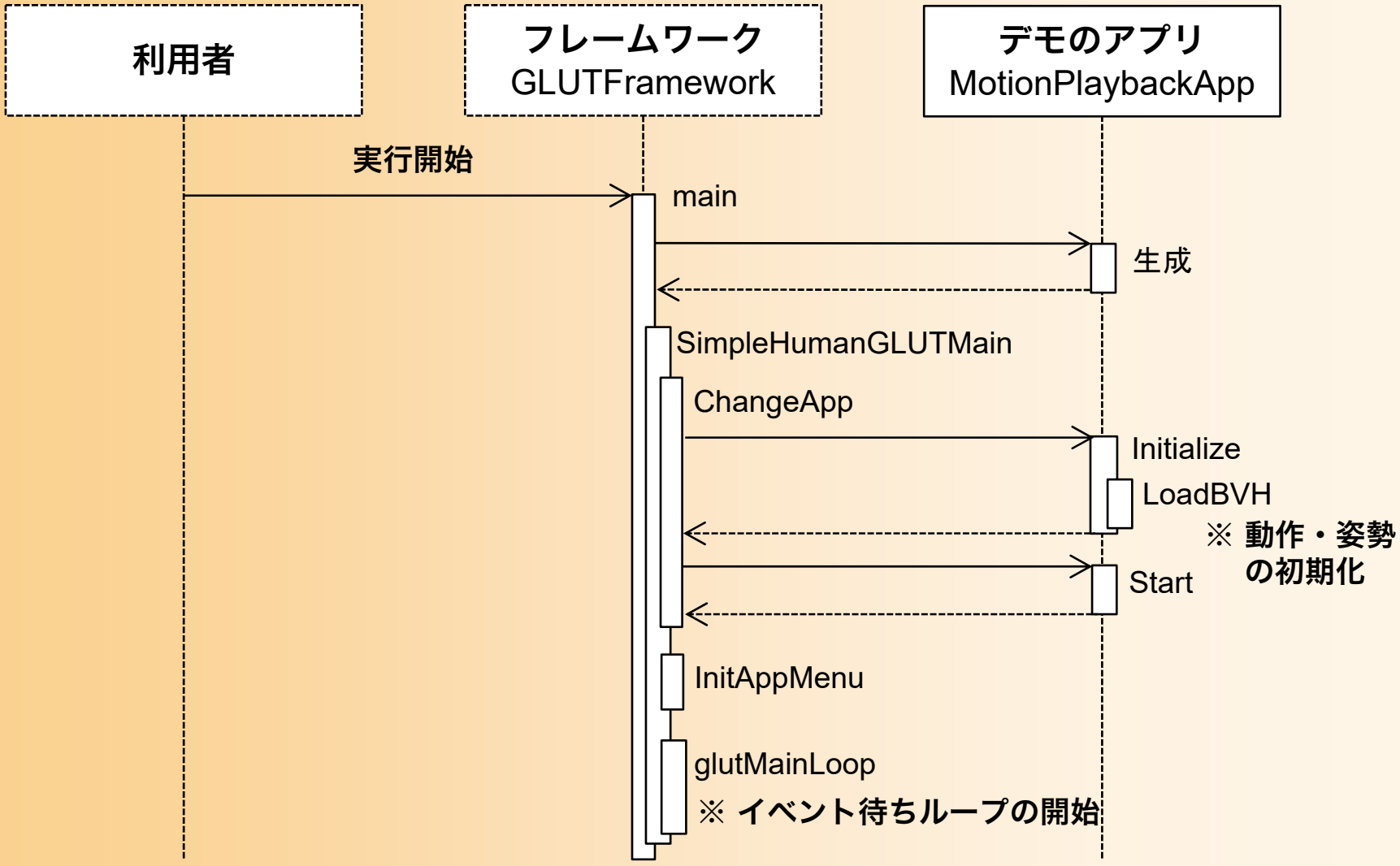


シーケンス図（1）

- プログラム開始時の処理の流れ
  - 処理の流れ（上から下）に従って、どのオブジェクトのどのメンバ関数が呼ばれるかを表す

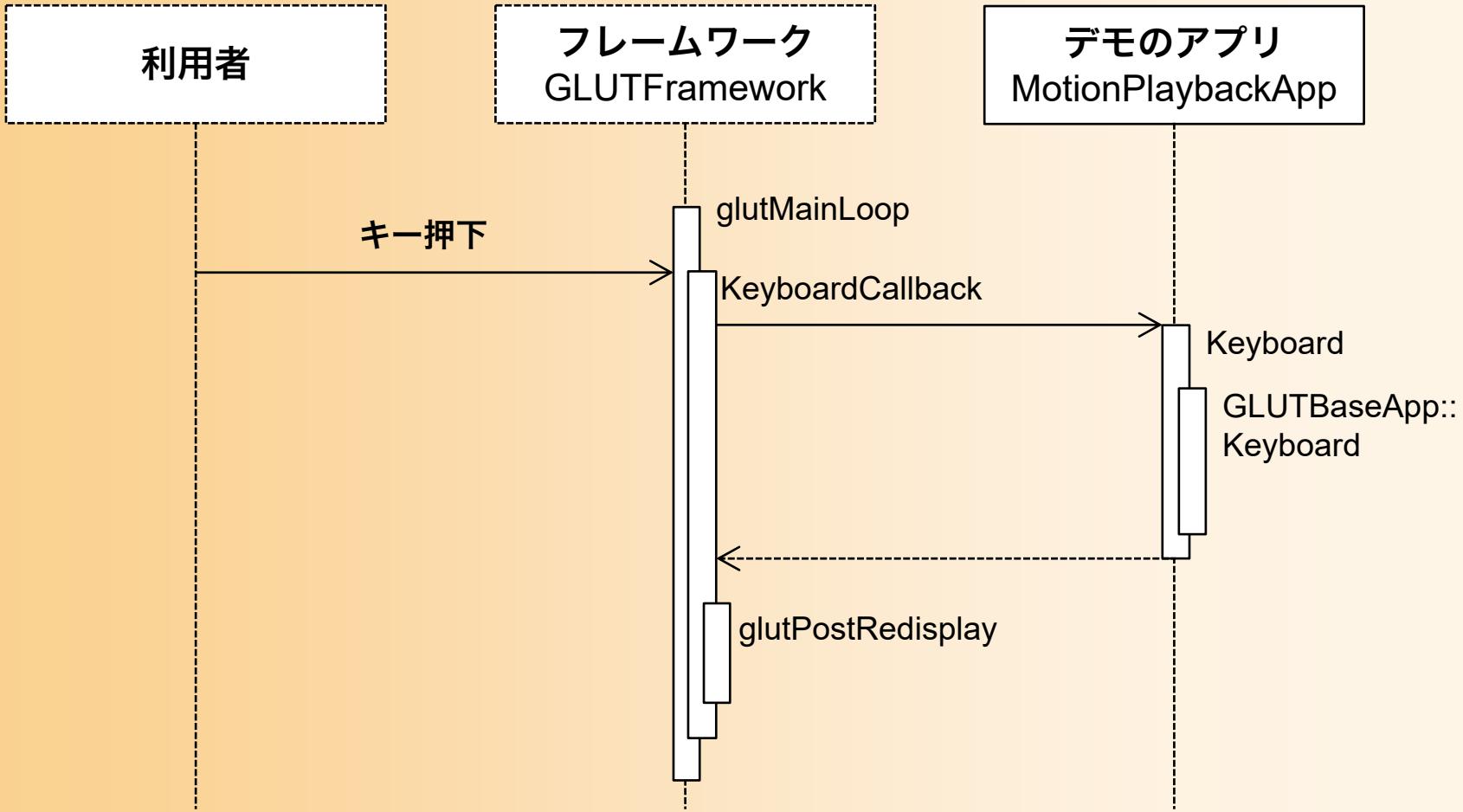


シーケンス図（2）


利用者
フレームワーク
GLUTFramework
デモのアプリ
MotionPlaybackApp
実行開始
main
生成
SimpleHumanGLUTMain
ChangeApp
Initialize
LoadBVH
※ 動作・姿勢の初期化
Start
InitAppMenu
glutMainLoop
※ イベント待ちループの開始

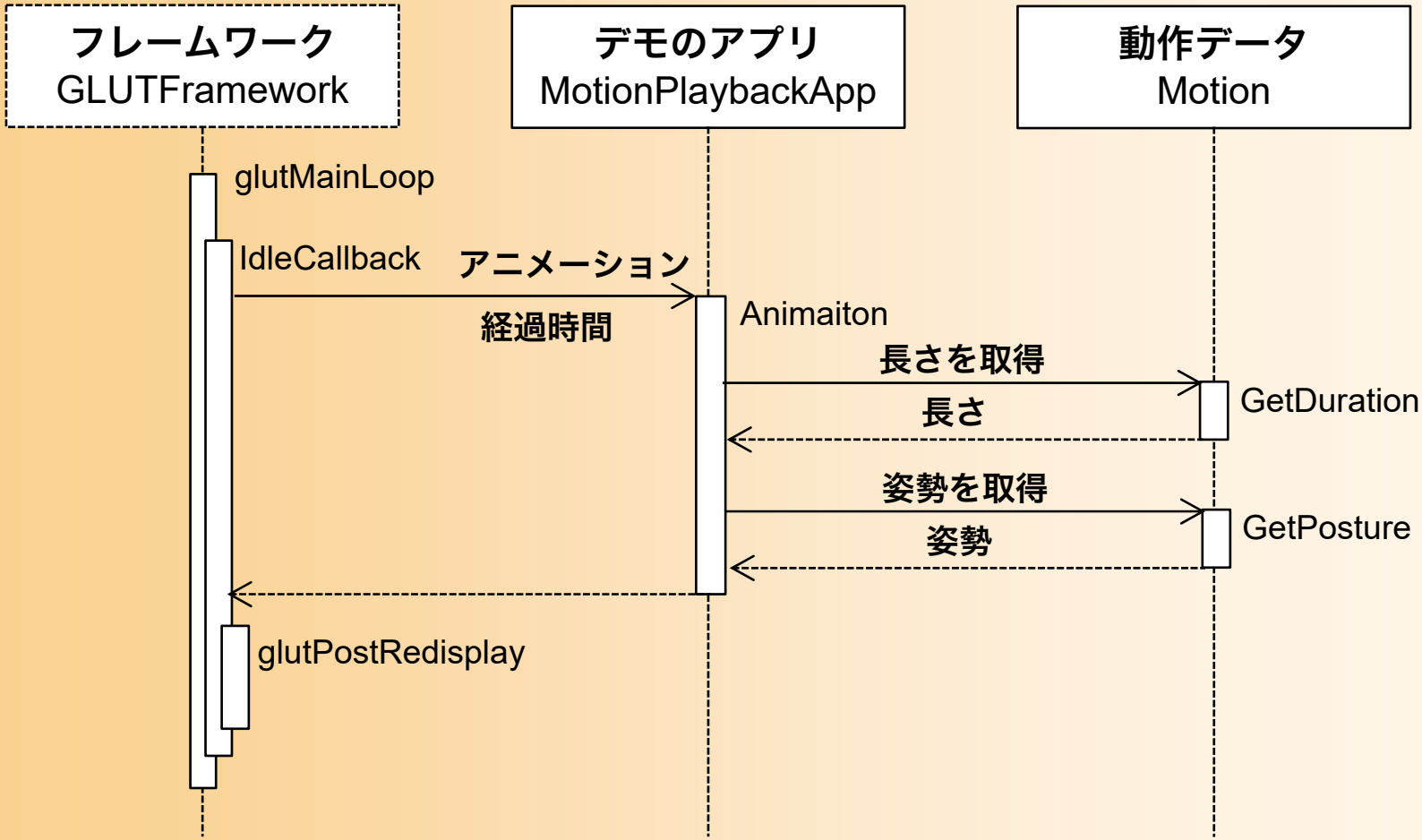
シーケンス図（3）

- イベント発生時（キー入力）の流れ



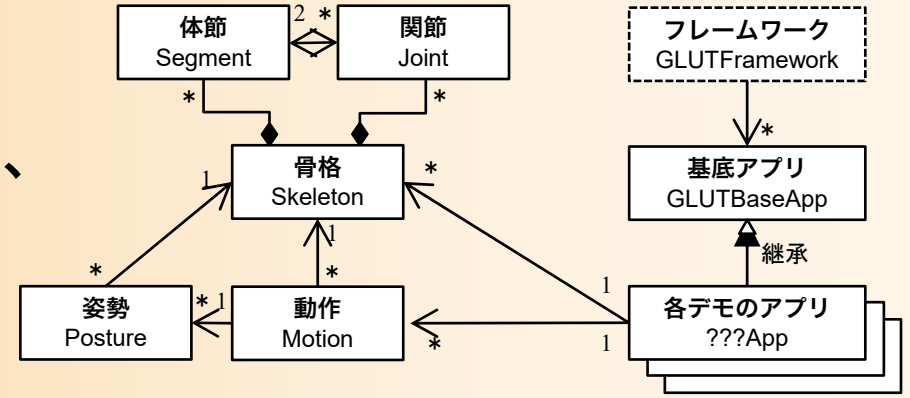
シーケンス図（4）

- イベント発生時（アニメーション処理）の流れ



参考：UML

- ここまでに出てきたクラス図やシーケンス図は、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
    - 他にも、ユースケース図、状態図、などがある
    - UMLの知識があれば、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

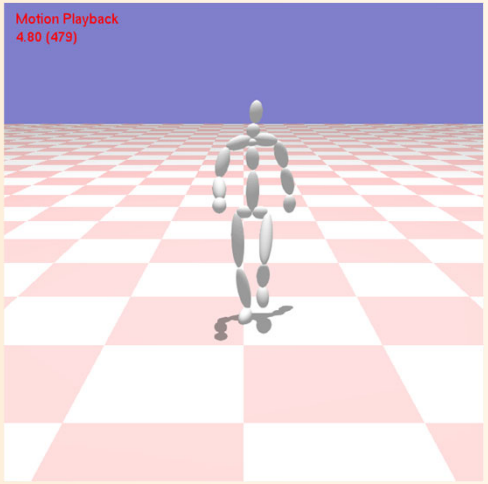
- デザインパターン
  - オブジェクト指向によるソフトウェア設計では、複数のクラス同士が連携して大きな機能を実現する
  - よく使われるクラスの役割の組み合わせをパターンとしてまとめたものをデザインパターンと呼ぶ
    - 23種類の GOFパターンが代表的なデザインパターン
  - デザインパターンを知っていれば、ソフトウェア設計に応用でき、コミュニケーションを円滑に進められる
- 本プログラムは、テンプレート パターンに基づく
  - アプリケーションの処理手順の枠組みを規定して、その枠組みにそった複数の異なるアプリケーションを実装

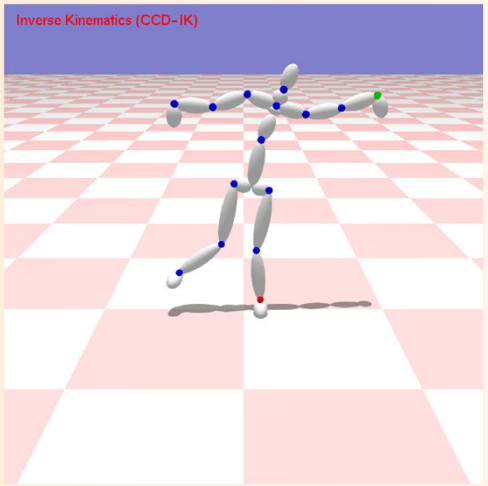
サンプルプログラム

- サンプルプログラムの概要
- 骨格・姿勢・動作のデータ構造
- 骨格・姿勢・動作の基本処理関数
- アプリケーションの基底クラス
- GLUTコールバック関数とメイン関数
- プログラムの設計のまとめ

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）





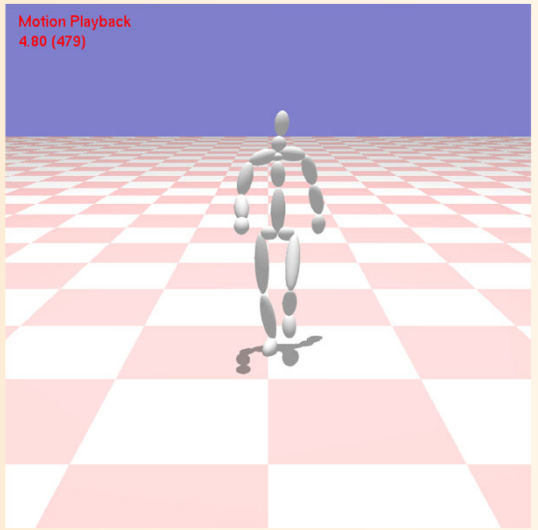
今日の内容

- 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
- 動作再生

動作再生

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

- クラス定義・実装（MotionPlaybackApp.h/cpp）
  - GLUTBaseAppを派生

```
class  MotionPlaybackApp : public GLUTBaseApp
{
  protected:
    // 動作データ
    Motion *  motion;
    // 現在の姿勢
    Posture *  curr_posture;
    ...
  public:
    // イベント処理インターフェース
    ....
};
```

動作再生アプリケーション（2）

- MotionPlaybackApp クラスのメンバ変数
  - 動作再生の入力情報
    - 動作データ（Motion * motion）
      - 人体モデルの骨格情報（Skeleton * body）を含む
  - 動作再生のための変数
    - 現在の姿勢（Posture * curr_posture）
    - アニメーション再生中かどうかを表すフラグ
    - アニメーションの再生時間
    - アニメーションの再生速度
    - 現在の表示フレーム番号

動作再生アプリケーション（3）

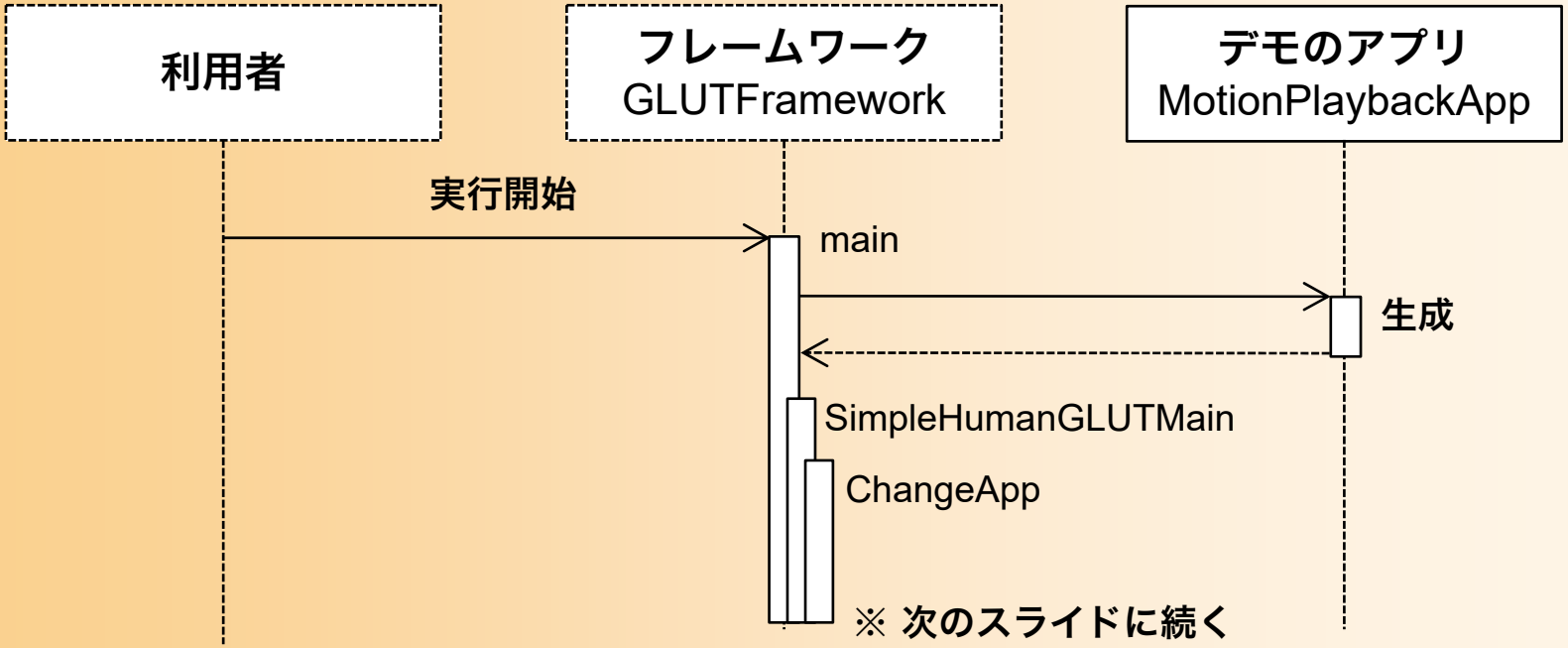
- 初期化処理（Initialize関数）
  - 動作データ（+骨格モデル）の読み込みと生成
    - LoadBVHメンバ関数 →
      LoadAndCoustructBVHMotion関数を呼び出し
- 開始処理（Start関数）
  - アニメーションの再生時刻をリセット
- キー入力関数（Keybaord関数）
  - アニメーションの再生・停止・前後フレーム
  - 別の動作データの読み込み（LoadBVHメンバ関数）

動作再生アプリケーション（4）

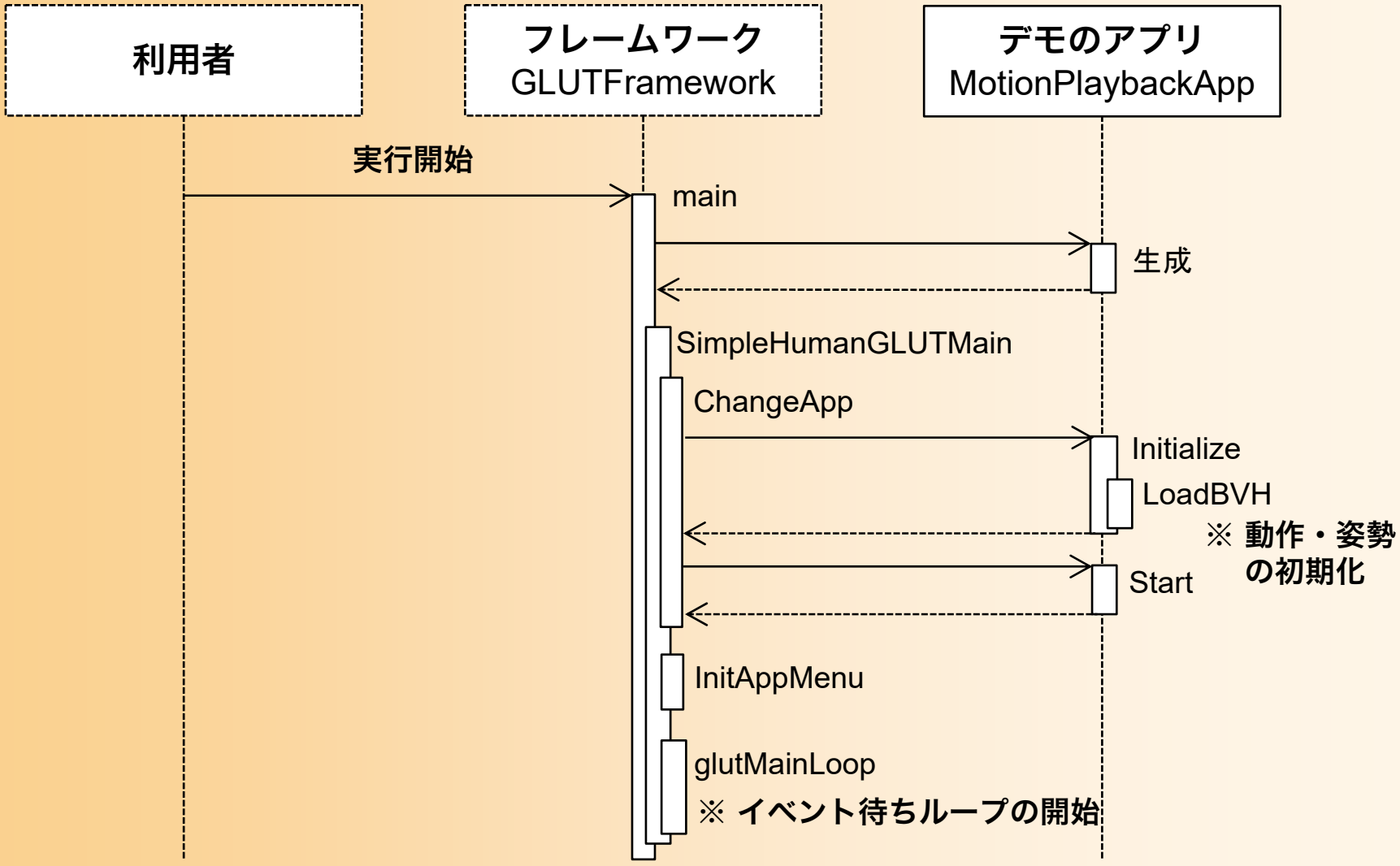
- アニメーション（Animation関数）
  - 前フレームからの経過時間 delta に応じて、アニメーションの再生時間を進める
  - 動作から現在時刻の姿勢を取得
    - Motionクラスの GetPosture メンバ関数を利用
    - 現在時刻の姿勢をメンバ変数 curr_posture に格納
- 画面描画（Display関数）
  - 現在姿勢（curr_posture）を描画
    - DrawPosture関数、DrawPostureShadow関数を利用
  - 時刻・フレーム数を表す文字列を描画

シーケンス図（1）

- プログラム開始時の処理の流れ
  - 処理の流れ（上から下）に従って、どのオブジェクトのどのメンバ関数が呼ばれるかを表す

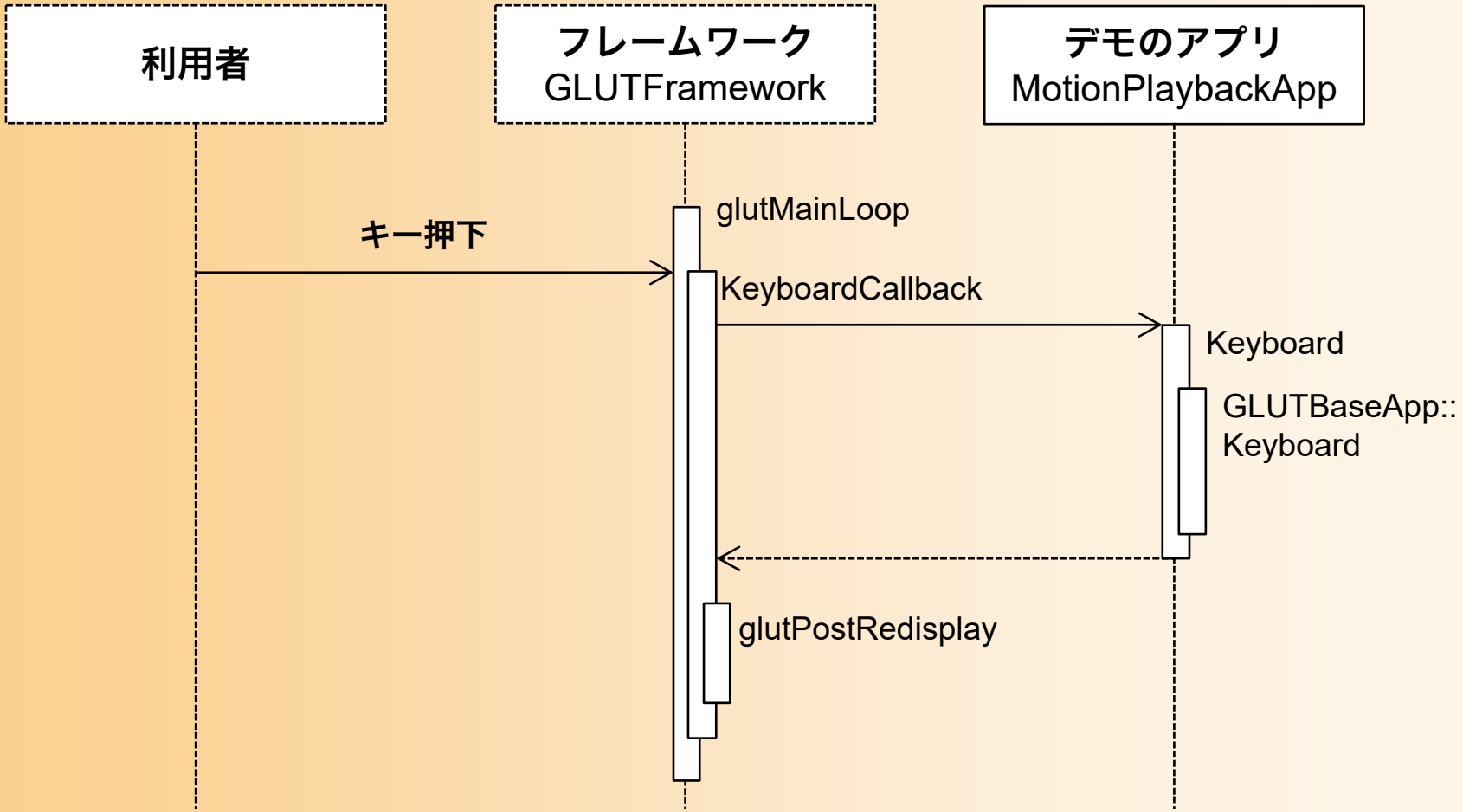


シーケンス図（2）


利用者
フレームワーク
GLUTFramework
デモのアプリ
MotionPlaybackApp
実行開始
main
生成
SimpleHumanGLUTMain
ChangeApp
Initialize
LoadBVH
※ 動作・姿勢の初期化
Start
InitAppMenu
glutMainLoop
※ イベント待ちループの開始

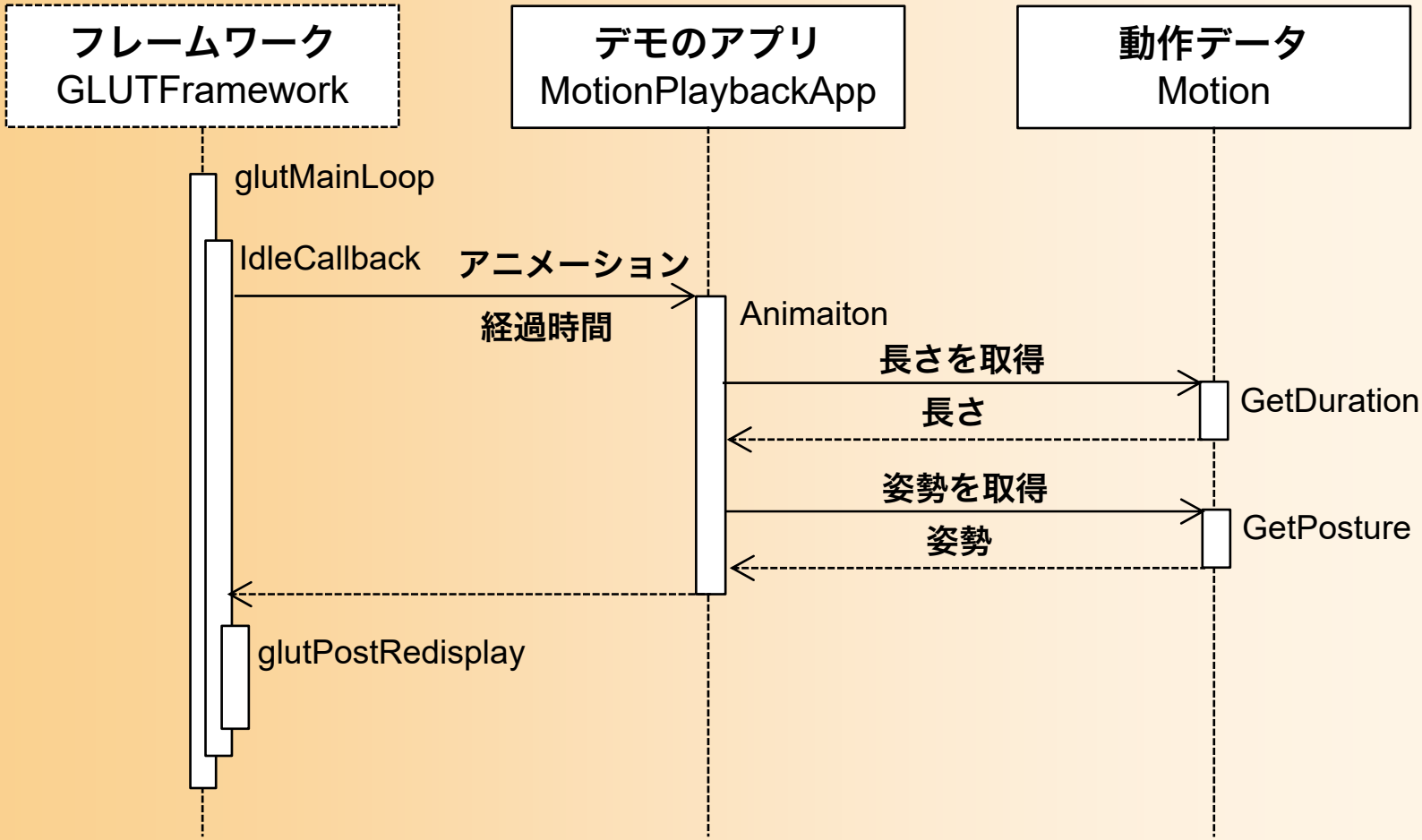
シーケンス図（3）

- イベント発生時（キー入力）の流れ



シーケンス図（4）

- イベント発生時（アニメーション処理）の流れ



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、時刻・フレーム数を描画

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
- 動作再生

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. 動作変形
    6. 動作接続・遷移
    7. 逆運動学計算（CCD法）

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