コンピュータアニメーション特論

第4回　キーフレームアニメーション（1）

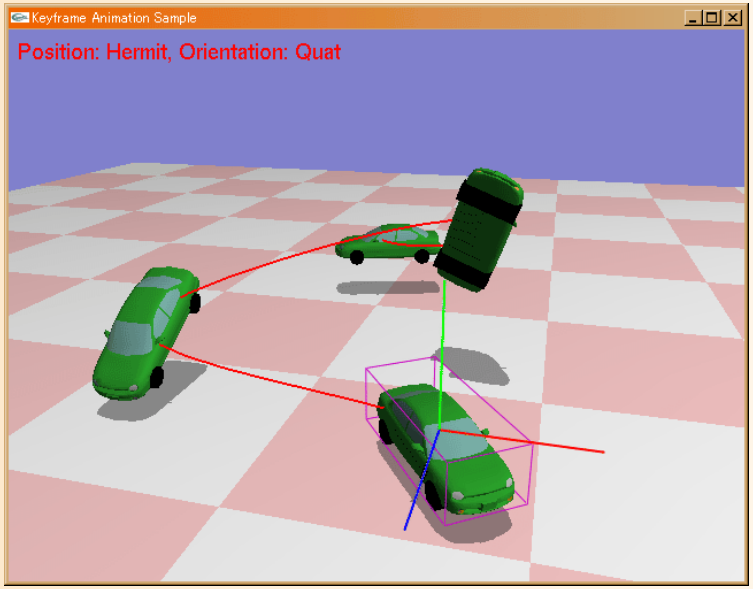
九州工業大学 情報工学研究院　尾下真樹

今日の内容

- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間
  - Hermite曲線
  - Bézier曲線
  - B-Spline曲線

キーフレームアニメーション

- 入力された複数のキーフレーム（時刻・状態の組）からアニメーションを生成
  - 少数のキーフレームの情報から、連続的なアニメーションを生成
  - 前後のキーフレームの状態（位置・向き）を補間して、キーフレーム間の任意時刻の状態を生成
    - 位置や向きの補間の計算が必要となる



全体の内容

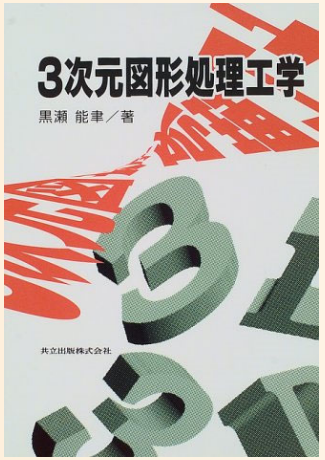
- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間、Hermite曲線、Bézier曲線、B-Spline曲線
- 向きの補間
  - 向きの表現と変換、オイラー角、四元数と球面線形補間
- アニメーションプログラミング
- レポート課題

今日の内容

- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間
  - Hermite曲線
  - Bézier曲線
  - B-Spline曲線

参考書

- 「3DCGアニメーション」
  栗原恒弥・安生健一 著、技術評論社、¥2,980
  - アニメーション技術全般を解説
- 3次元図形処理工学
  黒瀬 能聿 著、共立出版、¥2,600
  - 曲線・曲面について詳しく解説



- vecmathを理解するための数学
  平鍋 健児 著（四元数の詳しい解説）
  - http://www.objectclub.jp/download/vecmath1

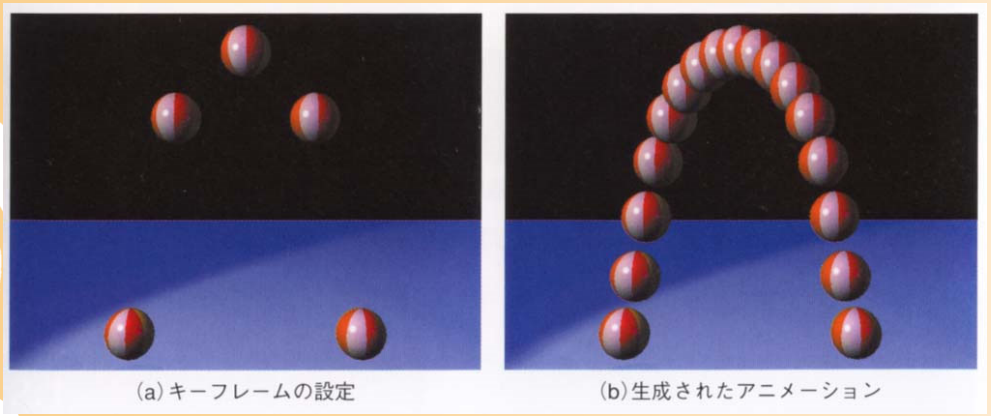
キーフレームアニメーションの基礎

アニメーションの原理

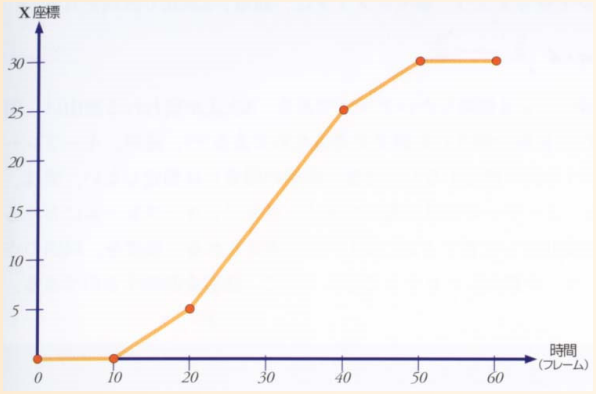
- 少しずつ変化する画像を連続して表示することでアニメーションとして見える
  - 1秒間に10枚～30枚毎程度（fps: frame per sec）
    - テレビ 30fps（60fps）、映画 24fps、アニメ 12fps、TV ゲーム 30 or 60 fps
  - 3次元アニメーションは、少しずつ物体の位置・向きを変えながら、連続して描画することで実現

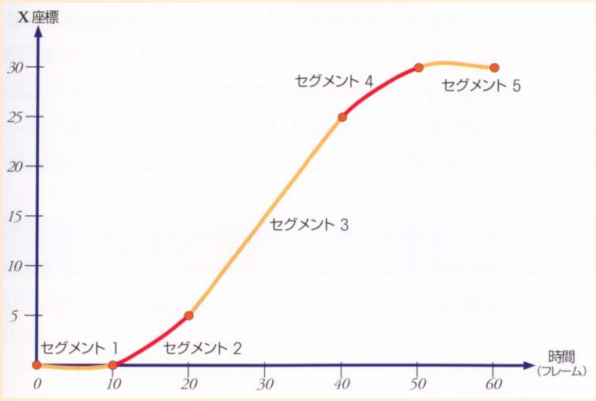
キーフレームアニメーション

- 動きのキーとなる（時刻＋状態）のデータ列から、全体の動きを生成
  - キーフレームの補間の方法には多くの種類がある
    - 線形補間
    - ベジェ曲線、スプライン曲線



参考書 図3.3





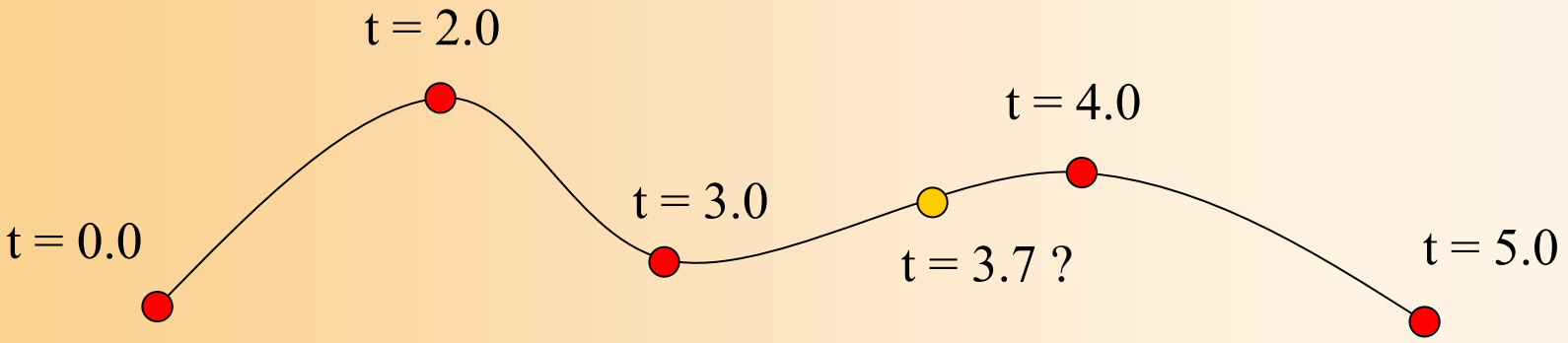
参考書 図3.9

動力学シミュレーション

- 物理法則に従ったアニメーション
  - 現実の物体は物理法則に従う
    - 重力、衝突、摩擦力、力を加えると運動する、落下、等
  - キーフレームアニメーションでは、これらの物理法則は考慮されないので不自然に見える可能性がある
- 動力学シミュレーションを使ったアニメーション生成
  - 初期状態を与えると、各フレームごとに運動方程式を数値的に解いていくことで、アニメーションが生成される
  - 望むような結果を得るような初期条件の設定が難しいという問題がある
  - キーフレームアニメーションとの使い分けが必要
  - 動力学シミュレーションについては、後日の授業で説明

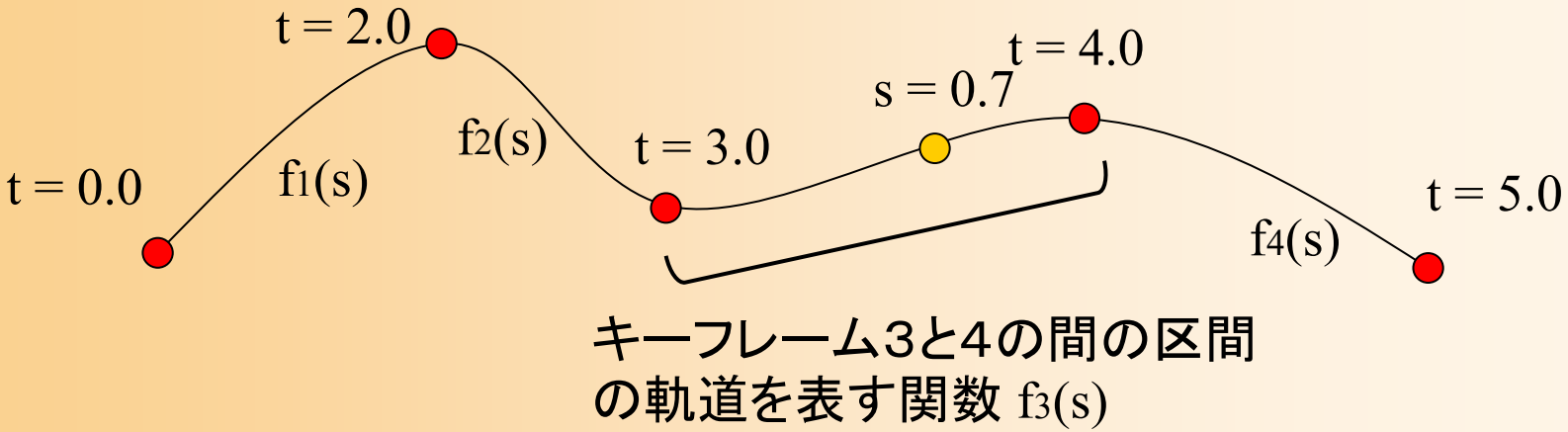
キーフレームアニメーションの実現方法

- キーフレームにおけるオブジェクトの位置・向きを補間して、全フレームの位置・向きを計算
- 位置・向きのそれぞれについて補間を行う必要がある
  - 位置と向きでは補間の方法が異なるため



補間の考え方

- 補間関数
  - 軌道全体を各キーフレーム間の区間に分ける
  - 各区間の軌道を何らかの関数により表現
    - 通常は、区間の前後の制御点をもとに、関数を決定
  - 全体の時刻から、現在の区間内のローカル時間を計算（例：s = 0.0 ～ 1.0 の範囲とする）



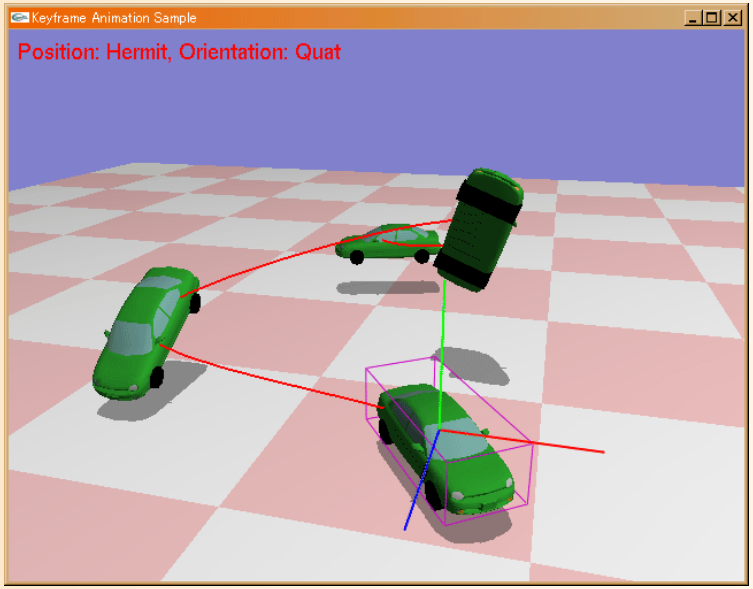
位置・向きの補間

- 位置の補間方法
  - 位置の表現方法
    - 位置ベクトルによる表現
  - 位置の補間方法
    - 線形補間、Hermite曲線、Bézier曲線、B-Spline曲線
- 向きの補間方法
  - 向きの表現方法
    - 回転行列、オイラー角、回転軸と回転角度、四元数
  - 向きの補間方法
    - オイラー角、四元数

サンプルプログラム

デモプログラム

- キーフレームアニメーション
  - レイアウトモード
    - キーフレームを表すオブジェクトの追加・削除
    - マウス操作による、オブジェクトの選択・移動・回転
  - アニメーションモード
    - 位置補間方法の切り替え
      - 線形補間、Hermite補間、Bezier補間、B-Spline補間
    - 向き補間方法の切り替え
      - オイラー角、四元数補間
    - 軌道表示機能




Keyframe Animation
Layout Mode
Position: Linear, Orientation: Eular
キーフレームアニメーション
Keyframe Animation

サンプルプログラムの構成

- デモプログラムの一部のサンプルプログラム（keyframe_sample.cpp）
- 幾何形状の読み込み・描画関数（Obj.h/cpp）
  - 過去の授業で扱った内容、影の描画も含む
- 物体の配置操作クラス（ObjectLayout.h/cpp）
  - キー操作・マウス操作によって、キーフレームを表す物体の位置・向きの変更ができる
- vecmath 補助関数（vecmath_gl.h）
  - 行列・ベクトルを扱うための vecmath ライブラリの補助関数（詳細は後述）

幾何形状の読み込み・描画関数

```
// 幾何形状データ(Obj形式用)
struct  Obj
      定義は省略(第4回目の講義資料を参照)

// Objファイルの読み込み
Obj *  LoadObj( const char * filename );

// Mtlファイルの読み込み
void  LoadMtl( const char * filename, Obj * obj );

//幾何形状モデルのスケーリング(スケーリング後のサイズを返す)
void  ScaleObj( Obj * obj, float max_size, ・・・ );

//幾何形状モデル(Obj形状)の描画
void  RenderObj( Obj * obj );

//幾何形状モデル(Obj形状)の影の描画(ポリゴン投影による影の描画)
void  RenderShadow( Obj * obj, float matrix[ 16 ], ・・・ );
```

過去の授業のサンプル
プログラムに追加

参考:影の表現

- レンダリング画像の現実感(リアリティ)を出す上で、影の描画は不可欠
  - 影の有無は、画面の自然さに大きく影響
  - 特に空中に浮いている物体を描画するようなときには、影があると、高さが把握しやすい
- 影の描画の技術
  - いくつかの方法が利用されている
  - 高度な描画技術が必要となる
  - 詳しくは後日の授業で紹介

参考：ポリゴン投影による影の描画

- 物体を構成する各ポリゴンを、地面に投影して、灰色（半透明）で描画
  - 物体の形状を反映した影を描画できる
  - 単純計算で、2倍の量のポリゴンを描画する必要がある

参考:影の描画処理

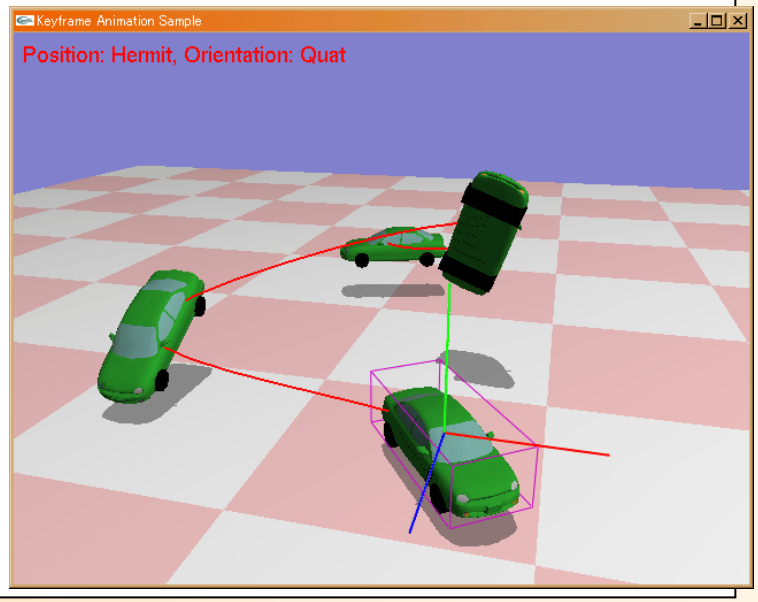
- ポリゴン投影による影の描画処理
  - 入力として以下の情報を渡す
    - 幾何形状モデル(Objクラスのオブジェクト)
    - 物体の位置・向き(4×4変換行列)
    - 光源の方向(3次元ベクトル)
    - 影の色(RGB)

```
//幾何形状モデル(Obj形状)の影の描画(ポリゴン投影による影の描画)
void    RenderShadow( const Obj * obj,
                const float obj_matrix[ 16 ], const Vector & light_dir,
                float color_r, float color_g, float color_b, float color_a );
```

オブジェクトの配置操作クラス(1)

```
class  ObjectLayout
{
      // オブジェクト情報を表す構造体
      struct  Object
      {
            Point3f   pos;  // 位置
            Matrix3f  ori;  // 向き
            Vector3f  size; // サイズ(描画用)
      };
      // 全オブジェクトの情報
      vector< Object >  objects;

      // オブジェクトの追加
      int  AddObject();
      // オブジェクトの削除
      int  DeleteObject( int no );
      // オブジェクトの情報設定
      ........
```

オブジェクトの位置・向きを表す構造体の可変長配列により、全てのオブジェクトの状態を管理

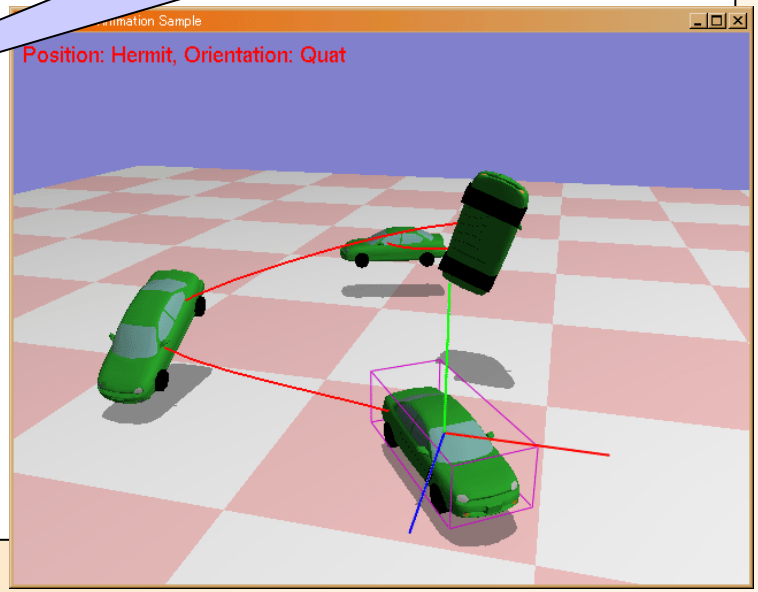


オブジェクトの配置操作クラス(2)

```
.....
// オブジェクト位置・視点の変更を通知
void  Update();
// マウス移動時の処理
void  OnMoveMouse( int mx, int my );
// マウスのボタンが押された時の処理
void  OnMouseDown( int mx, int my );
// マウスのボタンが離された時の処理
void  OnMouseUp( int mx, int my );

// オブジェクトの個数を取得
size_t  GetNumObjects();
// 各オブジェクトの位置・向きを取得
Point3f &  GetPosition( int no );
Matrix3f &  GetOrientation( int no );
Matrix4f &  GetFrame( int no );
}
```

マウス操作処理のメンバ関数
マウス関連のコールバック
関数の中から呼び出し

オブジェクトの数・位置・向きを
取得するためのメンバ関数
キーフレームの設定に使用



メインプログラム

- メインプログラム（keyframe_sample.cpp）
- 位置・向きの補間方法の切り替え機能
  - 位置・向きの補間方法を表す列挙型・変数を定義
  - キーボード入力により変更（KeyboardCallback（）関数）
- 幾何形状の読み込みと描画
- キーフレームの位置・向きの配置操作
  - MouseClickCallback（）, MouseDragCallback（）, MouseMotionCallback（）関数から、オブジェクト配置クラスの処理を呼び出し
- キーフレームアニメーション（IdleCallback（）関数）
  - 現在時刻に応じて、キーフレームの位置・向きを補間

位置・向きの補間方法の定義

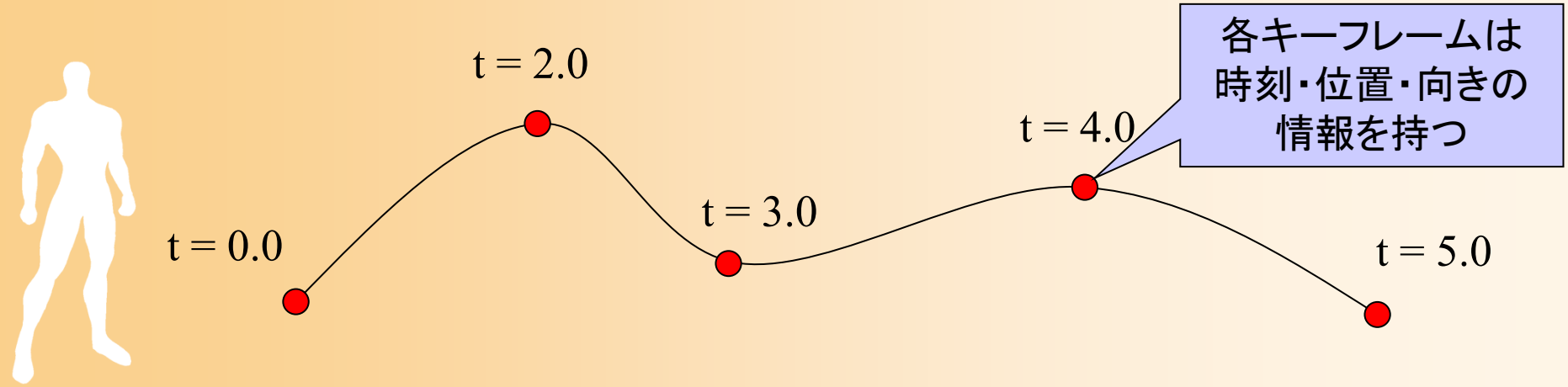
```
// 位置補間方法を表す列挙型
enum  PositionInterpolationEnum
{
        PI_LINEAR, PI_HERMIT, PI_BEZIER, PI_BSPLINE, NUM_PI_METHOD
};

// 向き補間方法を表す列挙型
enum  OrientationInterpolationEnum
{
        OI_NONE, OI_EULER, OI_QUAT, NUM_OI_METHOD
};

// 使用する位置・向き補間方法
PositionInterpolationEnum      pos_method = PI_LINEAR;
OrientationInterpolationEnum   ori_method = OI_EULER;
```

列挙型の総数を表す定数
（配列を確保したりするときに便利）

適当な初期値を設定

キーフレーム情報の定義

```
// キーフレーム情報
struct Keyframe
{
        float       time; // 時刻
        Point3f     pos;  // 位置
        Matrix3f    ori;  // 向き
};

// 設定されている全キーフレーム情報（可変長配列）
vector< Keyframe >   keyframes;
```

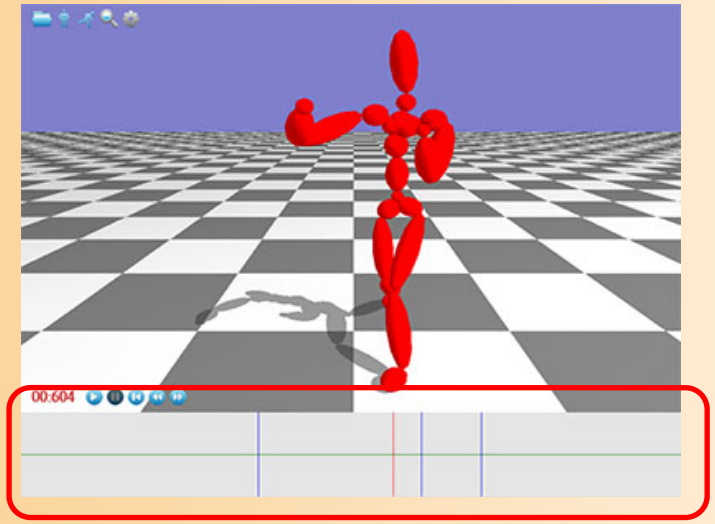


キーフレーム情報の設定

- キーフレーム設定（UpdateKeyframes関数）
  - オブジェクトの配置操作機能（ObjectLayout オブジェクト）から取得した、各オブジェクトの位置・向きを、各キーフレームの位置・向きとして使用する
  - 今回のプログラムでは、各キーフレームの時刻は固定とする
    - t = 0.0秒, 1.0 秒, 2.0 秒, 3.0 秒, …. n-1 秒
    - 一般のアニメーションソフトでは、タイムラインを使ってキーフレームの時刻を指定できるようになっている

補足:キー時刻の操作

- 一般的なキーフレームアニメーション作成では、各キーフレームの時刻の操作も必要
  - 今回のプログラムでは、簡略化のため、各キーフレームの時刻は固定とする(t = 0, 1, 2, …. n-1)
  - タイムラインを使ったキーフレームの時間操作



位置・向きの補間

- 位置・向きの補間（UpdateModelMat（）関数）
  - 入力：キーフレーム配列、時刻
    出力：4×4変換行列（位置・向きの情報を含む）
  - 処理手順
    - 入力時刻が、何番目の区間に相当するかを判定
    - その区間における位置・向きを計算するための制御点の情報を、キーフレーム配列から取得
    - 制御点と補間関数にもとづき、入力時刻における位置・向きを計算する
  - サンプルプログラムでは一部の補間方法しか記述されていないので、残りは各自で作成する

位置・向きの表現

- 変換行列による位置＋向きの表現
  - 物体のローカル座標系からワールド座標系への変換行列（４×４行列）により表現
    - ３次元ベクトルによる位置の表現と、
    - ３×３行列による向きの表現、の組み合わせ
      - 向きには他の表現方法もある（詳しくは次回説明）
    - 位置＋向きをまとめて表現できる

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} & x \\ R_{21} & R_{22} & R_{23} & y \\ R_{31} & R_{32} & R_{33} & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

位置・向きの補間の関数

```
void  UpdateModelMat( int num_keyframes, const Keyframe * keyframes,
                      float time, float mat[ 16 ] )
{
    // 区間番号と区間内でのローカル時刻(0.0～1.0)を計算
    int  seg_no;   float  t;
    ・・・・・・・・

    // 位置を計算
    if ( pos_method == PI_LINEAR )
        // 線形補間
    else if ( pos_method == PI_HER
        // エルミート補間
    ・・・・・・・・

    // 向きを計算
    if ( ori_method == OI_NONE )
        // 補間なし
    else if ( ori_method == OI_EULAR )
        // 向きをオイラー角で補間
```

キーフレーム数、キーフレーム配列、時刻を入力すると、その時刻での位置・向きを求めて、4×4の変換行列として返す

変数で指定された補間方法によって計算方法を選択
（各計算の実現方法は、以降で説明）

キーフレームアニメーション再生

```
// アニメーション中のオブジェクトの位置・向きを表す変換行列
float  model_mat[ 16 ];
void  IdleCallback()
{
      // アニメーション中のオブジェクトの位置・向きを更新
      UpdateModelMat( keyframes.size(), &keyframes.front(),  animation_time,
                              model_mat );
}

void  DisplayCallback( void )
{
      // アニメーション中のオブジェクトを描画
      glPushMatrix();
            glMultMatrixf( model_mat );
            RenderObj( object );
      glPopMatrix();

      // オブジェクトの影を描画
      RenderShadow( object, model_mat );
```

現在時刻での位置・向きを表す4×4変換行列を求める

変換行列を直接設定
（現在設定されているワールド座標系からカメラ座標系への変換行列に、右からかける）

オブジェクトの描画関数を呼び出し

オブジェクトの影の描画関数を呼び出し

今日の内容

- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間
  - Hermite曲線
  - Bézier曲線
  - B-Spline曲線

行列・ベクトルを扱うプログラミング

行列・ベクトルを扱うプログラミング

- C/C++ での行列・ベクトルの扱い
  - 全て配列として扱う方法
    - 渡された配列を行列・ベクトルとみなして、各種演算を行うような関数を作成
  - 行列・ベクトルなどのクラスを作成する方法
- どちらの方法を使うとしても、標準的な方法はないので、自分で作成する or 既存のライブラリを選択する必要がある

行列・ベクトルのライブラリ

- OpenGL
  - OpenGLの関数は全て配列を渡すようになっており、行列・ベクトルの構造体は存在しない
- DirectX
  - 行列・ベクトルの構造体・演算関数が、ユーティリティとして提供されている
    - 他の環境で使うのは難しいため、一般的ではない
- Java3D
  - vecmath というクラスライブラリがある
  - vecmath の非公式 C++ 版も存在する

vecmath

- vecmath C++版
  - http://www.objectclub.jp/download/vecmath1
  - テンプレートライブラリ
    - Vector3f, Vector3d など、float と double 両方に対応
    - リンクの必要がなく、インクルードするだけで使える
  - 点（Point3）とベクトル（Vector3）の使い分け
    - 変換行列をかけると、Point3 には平行移動も適用されるが、Vector3 は回転のみ適用される（オーバーロードの機能により、型によって判断）
  - 一通りの機能があり便利

一般的な行列計算ライブラリ

- 一般的な行列計算のための C++ ライブラリ
  - Eigen、BLAS、LAPACK など
  - 一般的な行列・ベクトルの演算
    - 任意次元の行列の扱い、疎行列の扱い
    - 逆行列計算（LU分解）、特異値分解、コレスキー分解など
  - 3次元空間の行列・ベクトルの表現や計算にも使うことができる
  - テンプレートライブラリになっているものもある
  - 高機能な分、使い方はやや難しくなる

Eigen

- 行列・ベクトル計算の C++ ライブラリ
  - 任意次元の行列・ベクトルの表現・演算
  - テンプレートライブラリ、高速
  - 3次元空間の行列・ベクトル・四元数にも対応
    - コンピュータグラフィックスでの利用に適している
  - 行列演算に対応、疎行列にも対応
  - 最近広く使われているが、日本語の資料はまだ少ない

vecmathの利用方法(1)

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

vecmathの利用方法(2)

- vecmathのインストール
  - 適当な場所にコピー、インクルードディレクトリを設定
- vecmathのヘッダファイルをインクルードして利用
- 最新の Visual Studio では、プロジェクト設定の変更が必要(開発環境の設定方法の資料を参照)

- サンプルプログラムでは、vecmathクラスを引数として OpenGL 関数を呼び出すための関数を定義している(vecmath_gl.h)
  - 詳細はヘッダファイルの中身を参照

vecmathの利用方法(3)

- 行列・ベクトルのメンバ変数
  - 3次元ベクトル(Poin3f, Vector3f, Color3f 等)は、x, y, z のメンバ変数を持つ
  - 4次元ベクトル(Quat4f, Color4f等)は、x, y, z, w のメンバ変数を持つ
  - 3×3行列は、m00, m01, … , m22 のメンバ変数を持つ

$$\begin{pmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{pmatrix}$$

  - 4×4行列は、m00, m01, … , m33 のメンバ変数を持つ

$$\begin{pmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{pmatrix}$$

vecmathの利用方法(4)

- 行列・ベクトルへの値の取得・設定・変換
  - set()、get()メソッドにより、値の取得・設定・変換ができる

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

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

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

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

vecmathの利用方法(5)

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

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

v.add( v1, v2 ) ; // v1 と v2 の和を v に代入
v = v1 + v2; // 同じく、v1 と v2 の和を v に代入
v = v1.cross( v2 );  // v1 と v2 の外積(ベクトル)を v に代入
s = v1.dot( v2 );  // v1 と v2 の内積(スカラ)を s に代入
v.scaleAdd( 0.5f, v1, v2 ); // v1 の 0.5 倍を v2 に加えたものを、v に代入

m.mul( m1, m2 );  // m1 と m2 の積を m に代入    M = M1  M2
m = m1 * m2;  // 同じく、m1 と m2 の積を m に代入                M = M1  M2^t
m.mulTransposeRight( m1, m2 )  // m1 と m2の転置 の積を m に代入
m.invert( m1 );  // m1 の逆行列を m に代入      M = M1^-1
```

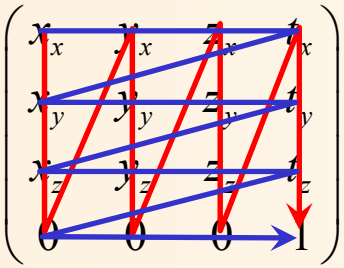
vecmathの利用方法(6)

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

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

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

行列の扱いに際しての注意(1)

- ライブラリ内部の表現方法によっては、OpenGLに渡すときに、表現方法を変換する必要がある
- 右手座標系 or 左手座標系
  - OpenGLでは右手座標系
- どちらから行列をかけるか
  - OpenGLでは左からかける
  - 異なる方式の間では行列の転置が必要
- 行→列表現か、列→行表現か
  - OpenGLでは、列→行表現
  - 異なる方式の間では行列の転置が必要



行列の扱いに際しての注意(2)

- ライブラリ内部の表現方法によっては、OpenGLに渡すときに、表現方法を変換する必要がある
- vecmath から OpenGL への変換行列の受け渡し
  - vecmath では、OpenGL と同様、右手座標系で左からかけるが、行→列表現のため、OpenGL に渡すには、転置が必要

```
void  glMultMatrixf( const Matrix4f & m )
{
   Matrix4f  mat;
   mat.transpose( m );
   glMultMatrixf( &mat.m00 );
}
```

vecmathの４×４行列を、現在の変換行列にかける

引数の４×４行列を、コピーして転置

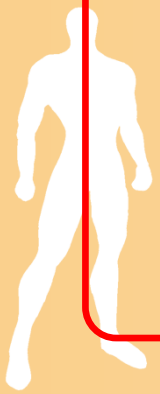
OpenGLの関数に、行列の先頭要素のアドレスを渡す

プログラミング演習問題

- 今回も、Moodle のプログラミング演習問題も受験する
  - VPL (Virtual Programming Lab) の機能を利用
  - 指定された処理を実現するように、与えられたプログラムの空欄を記述する
  - VPL 上で、コンパイル・実行、評価を行う
  - 評価まで行い、正しく実行されることを確認

今日の内容

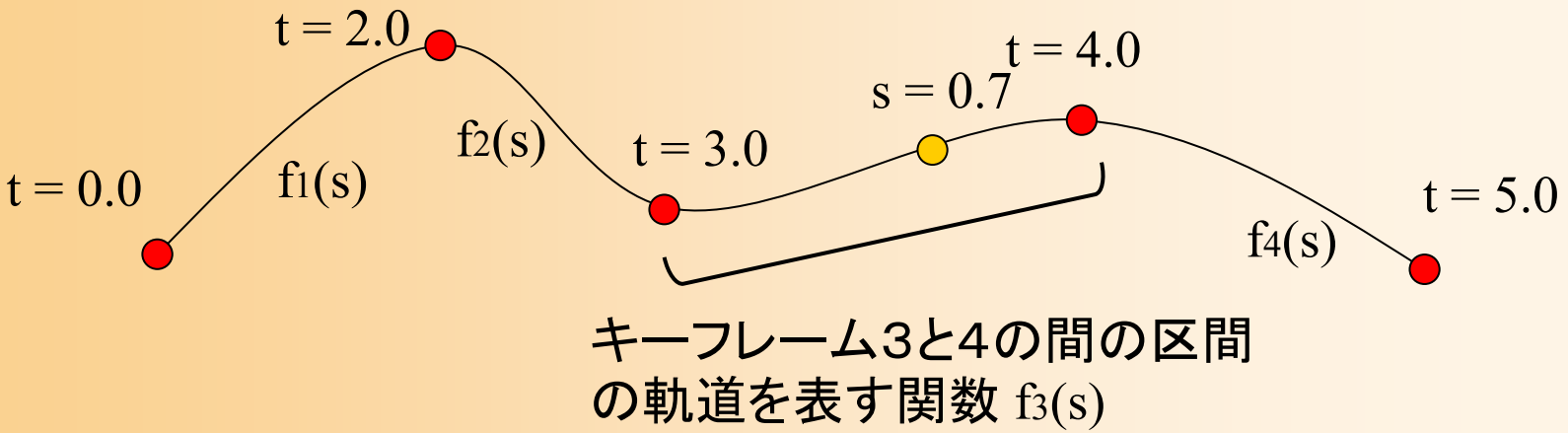
- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間
  - Hermite曲線
  - Bézier曲線
  - B-Spline曲線



位置補間

補間の考え方（確認）

- 補間関数
  - 軌道全体を各キーフレーム間の区間に分ける
  - 各区間の軌道を何らかの関数により表現
    - 通常は、区間の前後の制御点をもとに、関数を決定
  - 全体の時刻から、現在の区間内のローカル時間を計算（例：s = 0.0 ～ 1.0 の範囲とする）



位置・向きの補間（確認）

- 位置の補間方法
  - 位置の表現方法
    - 位置ベクトルによる表現
  - 位置の補間方法
    - 線形補間、Hermite曲線、Bézier曲線、B-Spline曲線
- 向きの補間方法
  - 向きの表現方法
    - 回転行列、オイラー角、回転軸と回転角度、四元数
  - 向きの補間方法
    - オイラー角、四元数

位置補間

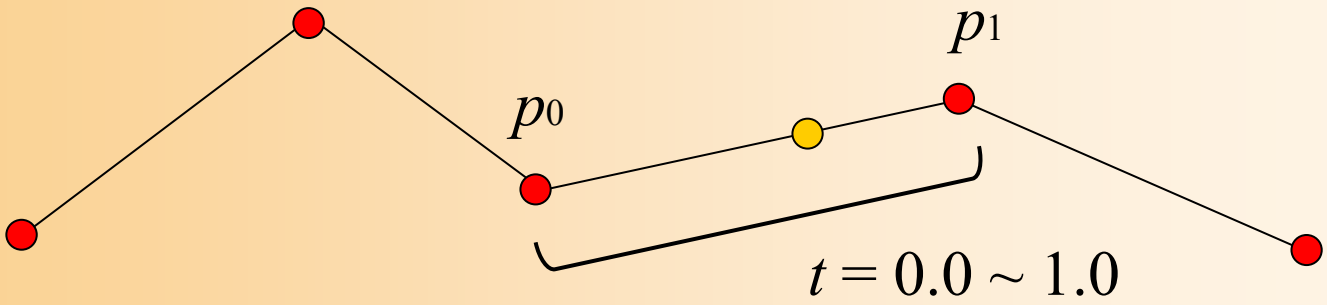
- 位置の表現方法
  - 位置ベクトル（x, y, z）による表現
  - 各座標値を独立に補間すれば良い
- 位置補間の方法
  - 線形補間
  - 曲線補間
    - Hermite曲線
    - Bézier曲線
    - B-Spline曲線

線形補間

- 区間の両端点間を線形に補間

$$\mathbf{p} = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

  - 区間内での速度が一定と仮定
  - 直線的な動き、キーフレームで動きが急に変化



プログラム例

```
// 求める位置を格納する変数
Vector3f  p;

// 現在の区間の両端点の位置を取得
const Point3f &  p0 = keyframes[ seg_no ].pos;
const Point3f &  p1 = keyframes[ seg_no + 1 ].pos;
```

- vecmath を使った計算方法の例
  - どちらの書き方でも可

```
// 両端点を線形に補間
p = t * ( p1 - p0 ) + p0;
```

```
// 両端点を線形に補間
p.sub( p1 - p0 );
p.scaleAdd( t, p, p0 );
```

2つ目の引数（ベクトル）を、1つ目の引数（実数）倍して、3つ目の引数（ベクトル）に加えたものを、計算結果として格納する

曲線を使った補間

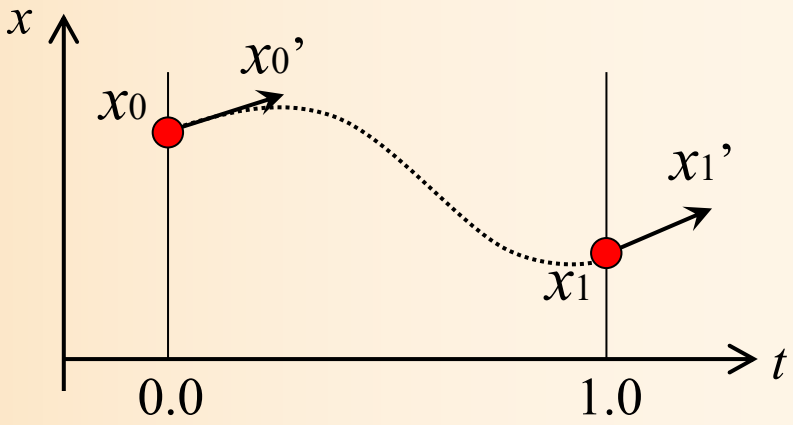
- いろいろな曲線の種類がある
  - Hermit曲線
    - ２点の位置・速度から補間関数を計算
  - Bézier曲線、B-Spline曲線
    - ４点の位置から補間関数を計算

Hermite曲線

- Hermite曲線（エルミート曲線）
  - 現在の区間の両端の２点の位置・速度から補間関数を決定
    - p はベクトルでも良い（x, y, z のそれぞれを補間）

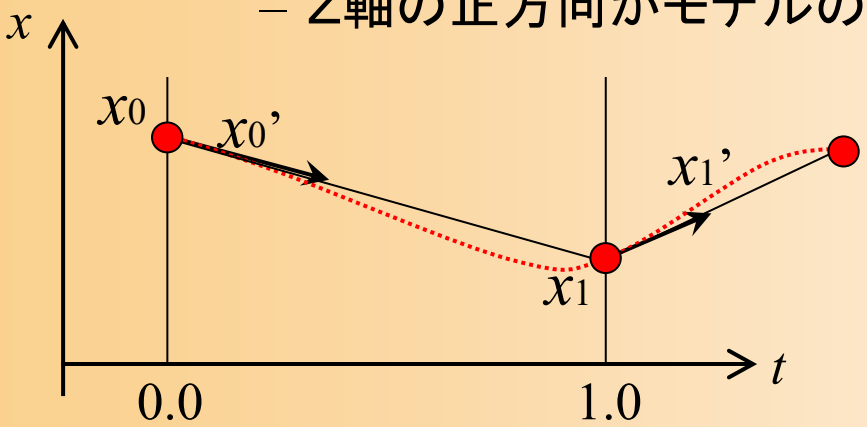
$$p(t) = H_0(t) \cdot p_0 + H_1(t) \cdot p_1 + h_0(t) \cdot p_0' + h_1(t) \cdot p_1'$$

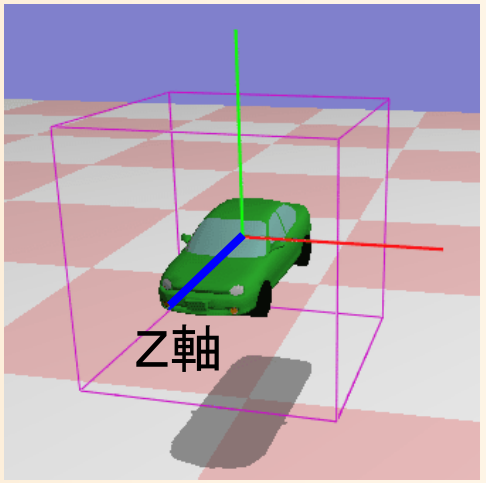
$$H_0(t) = 2t^3 - 3t^2 + 1 = (t-1)^2(2t+1)$$
$$H_1(t) = -2t^3 + 3t^2 = t^2(3-2t)$$
$$h_0(t) = t^3 - 2t^2 + t = (t-1)^2 t$$
$$h_0(t) = t^3 - t^2 = (t-1)t^2$$



Hermite曲線の使用

- キーフレームでの速度ベクトルの計算方法
  - 普通は位置・向きしか指定されないことが多いため、その場合は、何らかの方法で速度を計算
  - 速度ベクトル（方向+大きさ）の計算方法
    - 方法1：次のキーフレームとの差から計算
    - 方法2：向きの方向ベクトルから計算
      - Z軸の正方向がモデルの正面とする





プログラム例(1)

- vecmath を使った計算方法の例

```
// 求める位置を格納する変数
Vector3f  p;

// 区間の両端点の位置を取得
const Point3f &  p0 = keyframes[ seg_no ].pos;
const Point3f &  p1 = keyframes[ seg_no + 1 ].pos;

// 区間の両端点の傾きを取得
Vector3f  v0, v1;
const Matrix3f &  o0 = keyframes[ seg_no ].ori;
const Matrix3f &  o1 = keyframes[ seg_no + 1 ].ori;
o0.getColumn( 2, &v0 );
o1.getColumn( 2, &v1 );
v0.negate();
v1.negate();
```

モデル座標系の－Z軸を速度ベクトルの向きとする
速度ベクトルの大きさは1とする

プログラム例(2)

- vecmath を使った計算方法の例(続き)

```
// Hermite関数の係数を計算
float  a, b, c, d;
a = 2.0f * t * t * t - 3.0f * t * t + 1;
b = -2.0f * t * t * t + 3.0f * t * t;
c = t * t * t - 2.0f * t * t + t;
d = t * t * t - t * t;

// Hermite関数の計算
p.scale( a, p0 );
p.scaleAdd( b, p1, p );
p.scaleAdd( c, v0, p );
p.scaleAdd( d, v1, p );
```

前のスライドの式を記述

最後の式は、オーバーロードされた演算子を使用して、このように記述することもできる

```
// Hermite関数の計算
p = a * p0 + b * p1 + c * v0 + d * v1;
```

プログラム例(3)

- vecmath を使った計算方法の例(別方法)

```
// Hermite関数の係数を計算
Vector3f  a, b, c, d;
a =  2.0f * p0 - 2.0f * p1 +        v0 + v1;
b = -3.0f * p0 + 3.0f * p1 - 2.0f * v0 - v1;
c =                                  v0;
d =         p0;

// Hermite関数の計算
p = t*t*t * a + t*t * b + t*c + d;
```

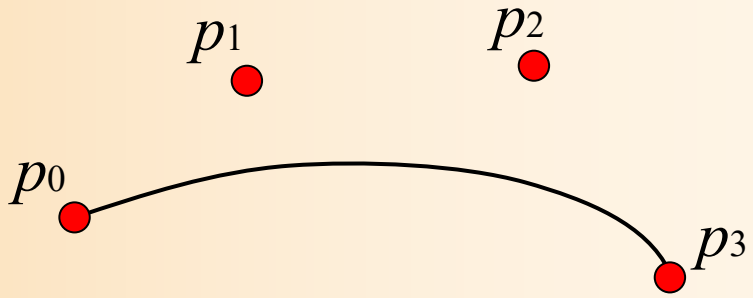
前のスライドの式を記述
t の3次式の各係数の値に変形して計算
(この例では毎回計算しているが、あらかじめ各区間ごとに計算して記録しておくと、処理を効率化できる)

Bézier曲線

- Bézier曲線（ベジェ曲線）
  - ４点の位置を補間
  - 接続が難しい
    - 隣接区間を滑らかにするには、前の区間の p_2-p_3 と次の区間の p_0'-p_1' を直線とする必要がある（$p_3 = p_0$'）

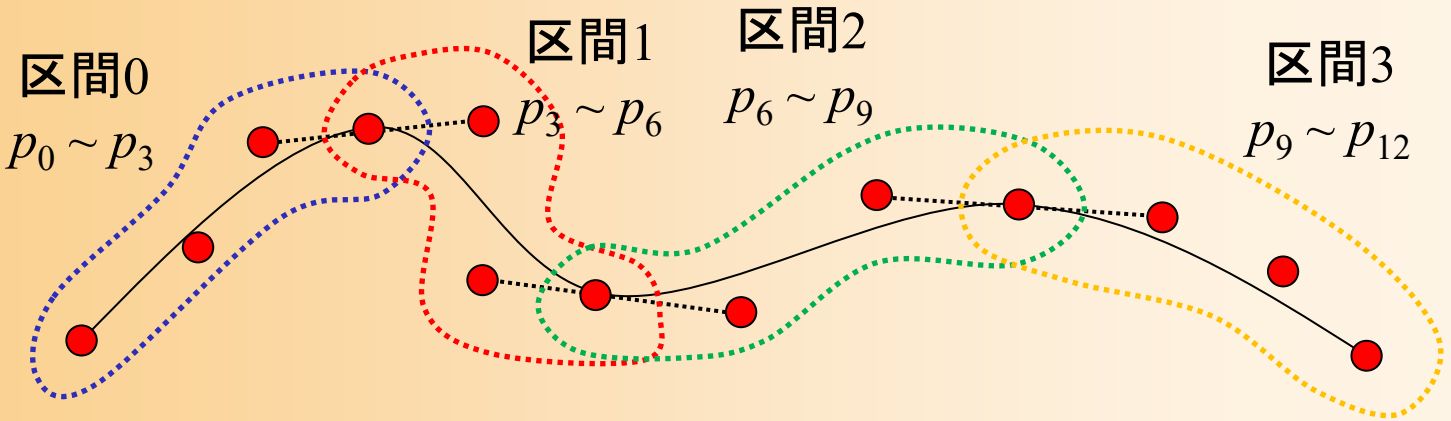
$$p(t) = X_0(t) \cdot p_0 + X_1(t) \cdot p_1 + X_2(t) \cdot p_2 + X_3(t) \cdot p_3$$

$$X_0(t) = (t-1)^3$$
$$X_1(t) = 3(t-1)^2 t$$
$$X_2(t) = 3(t-1)t^2$$
$$X_3(t) = t^3$$



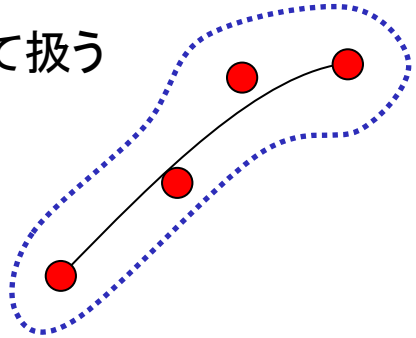
Bézier曲線の区間

- 連続する4点を1つの区間とする
  - i×3 ～ i×3+3 番目の点 で i番目の区間を定義
    - 各区間内での正規化時間（0～1）をもとに位置を補間
    - 隣接する区間が、1つの点を共有
  - 隣接区間を滑らかに接続するためには、共有点と前後の点の 3点が直線に並ぶようにする



プログラム例

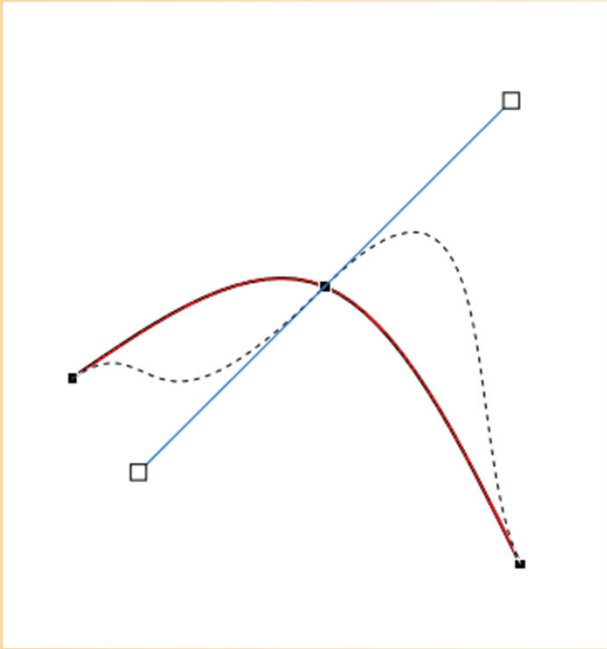
```
// 区間番号と正規化時間を計算
// 連続する4つのキーフレーム（3区間）をまとめて1つの区間として扱う
int  bezier_seg_no = (int) floor( seg_no / 3 );
float  s = ( time - keyframes[ bezier_seg_no * 3 ].time ) /
           ( keyframes[ bezier_seg_no * 3 + 3 ].time -
             keyframes[ bezier_seg_no * 3 ].time );

// Bezier補間の区間の4つの制御点（両端点と、中間の2つの点）の位置を取得
const Point3f &  p0 = keyframes[ bezier_seg_no * 3 ].pos;
const Point3f &  p1 = keyframes[ bezier_seg_no * 3 + 1 ].pos;
const Point3f &  p2 = keyframes[ bezier_seg_no * 3 + 2 ].pos;
const Point3f &  p3 = keyframes[ bezier_seg_no * 3 + 3 ].pos;

// Bezier関数の値を計算
p = ・・・;
```



前のスライドの式を記述
関数の引数には、t ではなく、
上で求めた s を使うことに注意

Bézier曲線の応用

- 2次元図形の編集機能を持つソフトウェアで広く使われている
  - 例：MS Office（Word, Power Point）、Illustrator



Power Point での
曲線編集画面の例

B-Spline曲線

- B-Spline曲線（Bスプライン曲線）
  - ４点の位置を使用
  - 接続が非常に容易
    - 前の区間の p_1, p_2, p_3 に、次の点 p_4 点を加えて次の区間にすれば、前後の区間がなめらかにつながる

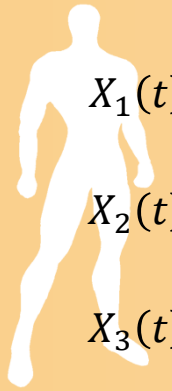
$$p(t) = X_0(t) \cdot p_0 + X_1(t) \cdot p_1 + X_2(t) \cdot p_2 + X_3(t) \cdot p_3$$

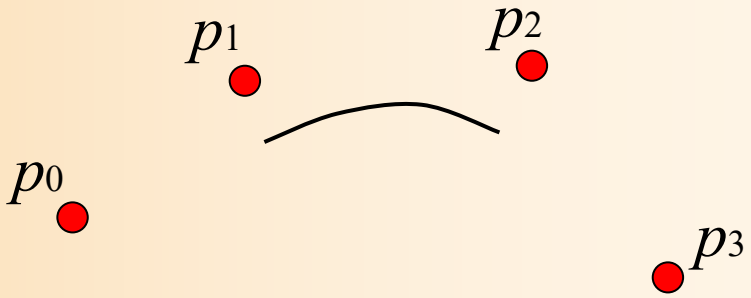
$$X_0(t) = -\frac{1}{6}t^3 + \frac{1}{2}t^2 - \frac{1}{2}t + \frac{1}{6}$$

$$X_1(t) = \frac{1}{2}t^3 - t^2 + \frac{2}{3}$$

$$X_2(t) = -\frac{1}{2}t^3 + \frac{1}{2}t^2 + \frac{1}{2}t + \frac{1}{6}$$

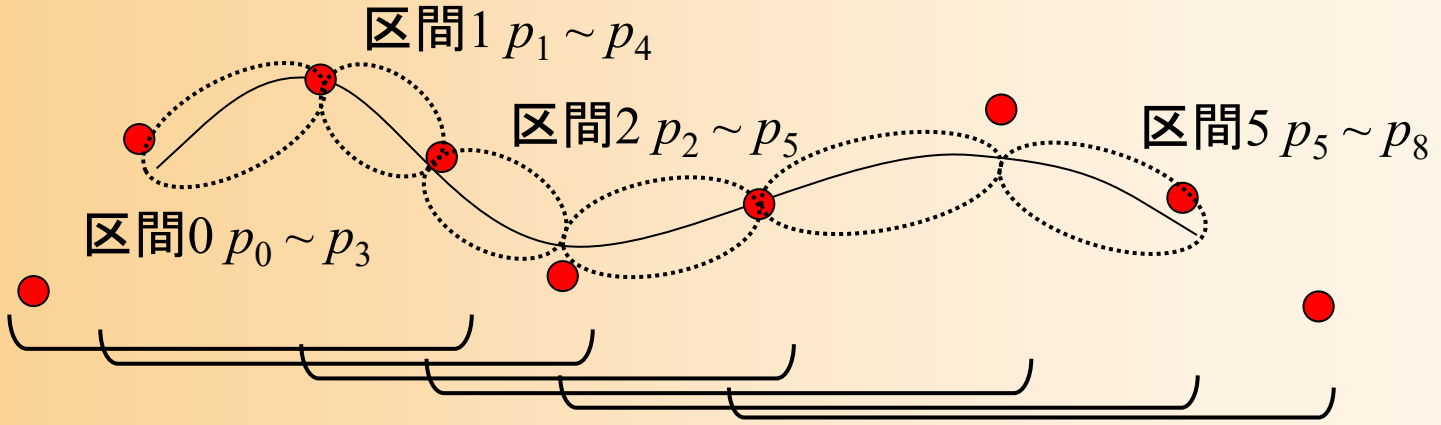
$$X_3(t) = \frac{1}{6}t^3$$





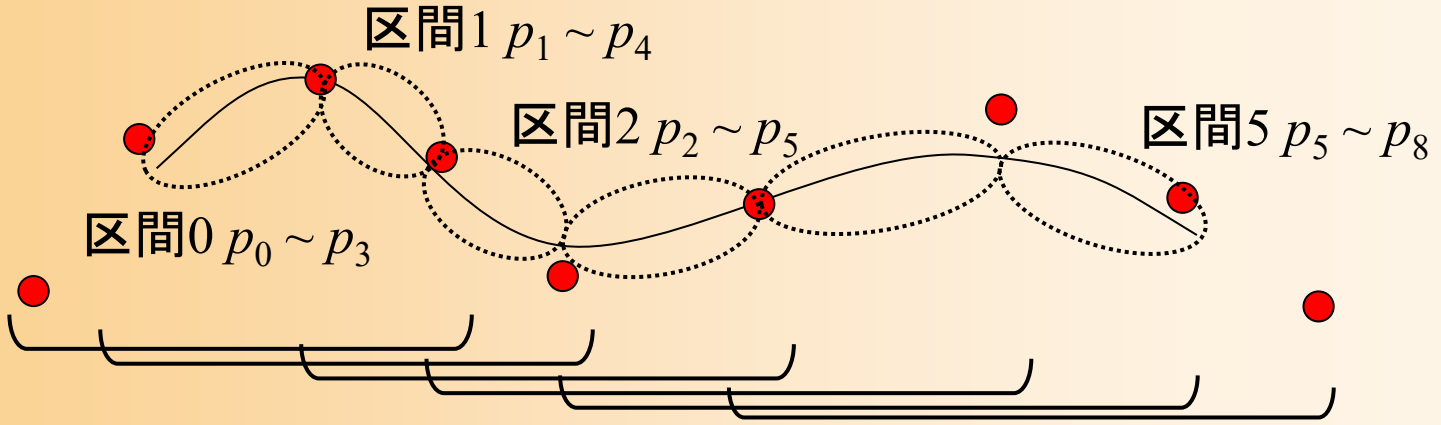
B-Spline曲線の区間

- 連続する4点を1つの区間とする
  - i －1 ～ i＋3 番目の点 で i 番目の区間を定義
  - 順番に点をずらしながら区間を定義することで、隣接する区間を連続的につなぐことができる
    - 最初と最後の2点間は、対応する曲線は存在しない
    - 制御点の数を n とすると、区間の数は n -3 となる



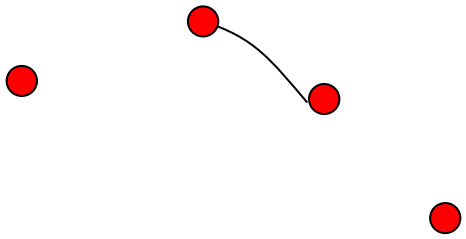
B-Spline曲線の区間

- 連続する4点を1つの区間とする
  - i －1 ～ i＋3 番目の点 で i 番目の区間を定義
  - 順番に点をずらしながら区間を定義することで、隣接する区間を連続的につなぐことができる
    - 最初と最後の2点間は、対応する曲線は存在しない
    - 制御点の数を n とすると、区間の数は n -3 となる



プログラム例

```
// 区間の両端点と、さらにその隣の点（もしあれば）の位置を取得
int  k0, k1, k2, k3;
k0 = ( seg_no > 0 ) ? ( seg_no - 1 ) : seg_no;
k1 = seg_no;
k2 = seg_no + 1;
k3 = ( seg_no + 2 == num_keyframes ) ? ( seg_no + 1 ) : ( seg_no + 2);
const Point3f &  p0 = keyframes[ k0 ].pos;
const Point3f &  p1 = keyframes[ k1 ].pos;
const Point3f &  p2 = keyframes[ k2 ].pos;
const Point3f &  p3 = keyframes[ k3 ].pos;

// B-Spline 関数の値を計算
p = ・・・;
```

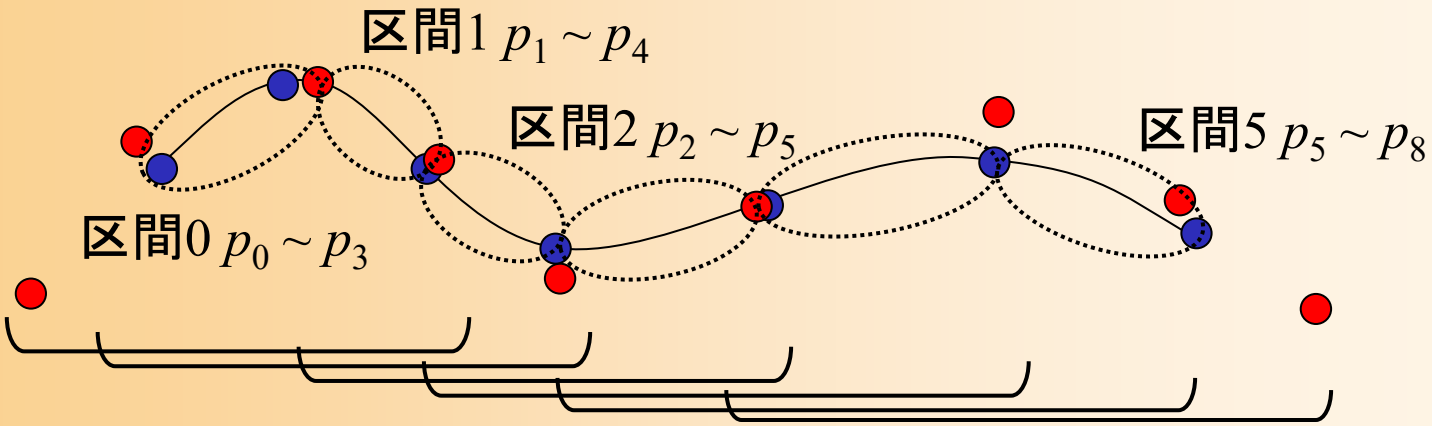


B-Spline曲線の特徴

- B-Spline曲線の特徴
  - 滑らかな軌道が得られる
    - 必ずしもキーフレームを通るとは限らない
    - 望む軌道を得るためには、キーフレームの位置を大きく調整する必要がある
      → 制御が難しい
  - 接続性が良い（Bézier曲線と比較して）
  - 与えられた複数の目標点を通るような B-Spline 曲線の制御点を計算する方法もある
    - 繰り返し処理により最適な制御点を計算
    - 参考書「3次元図形処理工学」（4.8節）参照

目標点を通るB-Spline曲線(1)

- 与えられた複数の目標点を通るような、B-Spline 曲線の制御点を計算する方法
  - 与えられた n-2個の目標点を通るような、B-Spline曲線の n個の制御点を計算する
    - n-3個の区間が定義される



目標点を通るB-Spline曲線(2)

- 繰り返し処理により、最適な制御点を計算
  - 目標点の位置を $\mathbf{P}_i\ (i = 1 \sim n-1)$ とする
  - 制御点の位置を $\mathbf{Q}_i\ (i = 0 \sim n)$ とする
  1. 最初に $\mathbf{Q}_i = \mathbf{P}_i$ で初期化（$\mathbf{Q}_0 = \mathbf{P}_1, \mathbf{Q}_n = \mathbf{P}_{n-1}$ ）
  2. 誤差が小さくなるように、繰り返し $\mathbf{Q}_i$ を更新
     - 全ての誤差 δ_i が一定値以下になれば終了

$$\delta_i = \mathbf{P}_i - \mathbf{Q}_i + \frac{1}{2}\left\{\mathbf{P}_i - \frac{1}{2}(\mathbf{Q}_i + \mathbf{Q}_{i+1})\right\}$$

$$\mathbf{Q}'_i = \mathbf{Q}_i + \delta_i$$

  - 参考書「3次元図形処理工学」(4.8節) 参照

曲線の利用

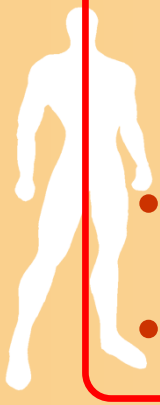
- キーフレームの位置・速度が決まっている時
  - Hermite曲線を利用
- キーフレームの位置だけ決まっている時
  - B-Spline曲線
  - キーフレームの位置を満たす制御点を、最適化計算により求める
- 対話的に軌道を変形したい時
  - Bézier曲線 or B-Spline曲線

全体の内容

- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間、Hermite曲線、Bézier曲線、B-Spline曲線
- 向きの補間
  - オイラー角、四元数と球面線形補間、相互変換
- アニメーションプログラミング
- レポート課題

次回予告

- キーフレームアニメーションの基礎
- サンプルプログラム
- 行列・ベクトルを扱うプログラミング
- 位置補間
  - 線形補間、Hermite曲線、Bézier曲線、B-Spline曲線
- 向きの補間
  - 向きの表現と変換、オイラー角、四元数と球面線形補間
- アニメーションプログラミング
- レポート課題



レポート課題（予告）

- 位置・向き補間の実装
  - サンプルプログラム（keyframe_sample.cpp）をもとに作成したプログラムを提出
    - UpdateModelMat()の空欄を埋めるプログラムを作成
  - 位置補間：Hermite曲線、Bézier曲線、B-Spline曲線
  - 向き補間：四元数の球面線形補間