



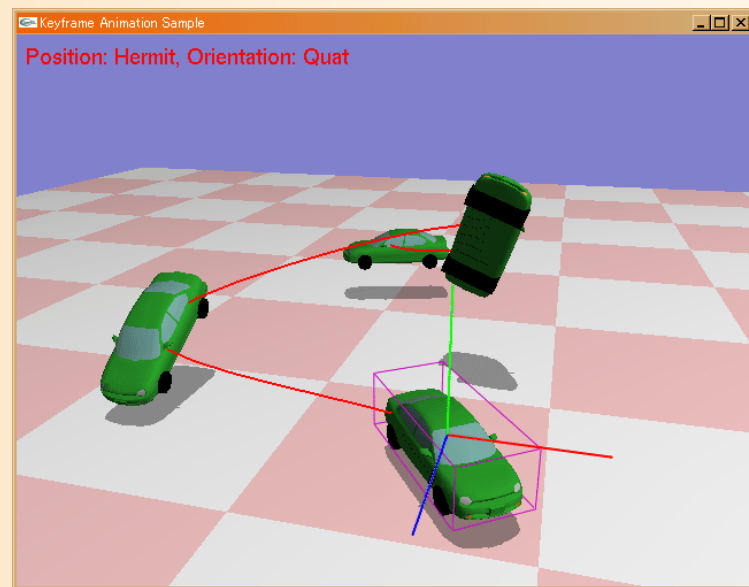
コンピュータグラフィックス特論Ⅱ

第6回 キーフレームアニメーション(1)

九州工業大学 尾下 真樹

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



今日の内容

- キーフレームアニメーション

- 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

- アニメーションプログラミング

- レポート課題



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 著、共立出版、¥2,600
– 曲線・曲面について詳しく説明
- 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>



参考書(続き)

- Computer Graphics Gems JP 2013/2014
「パラメトリックポーズブレンド」
 - 回転の補間方法についての詳しい解説





キーフレームアニメーションの基礎

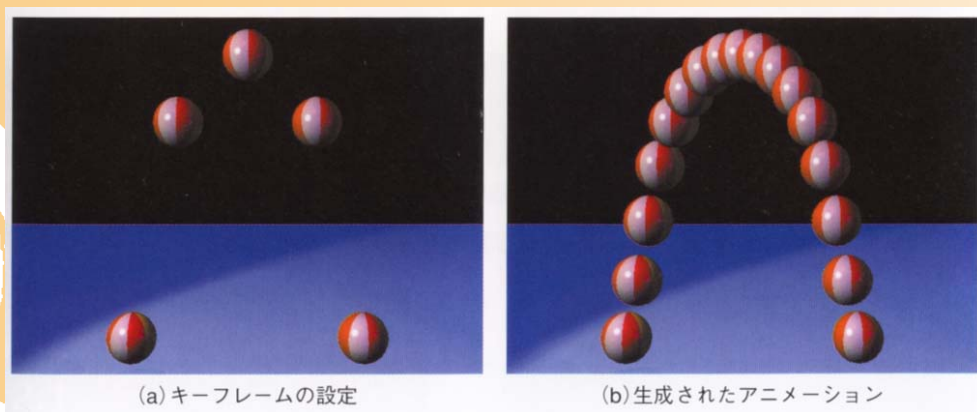
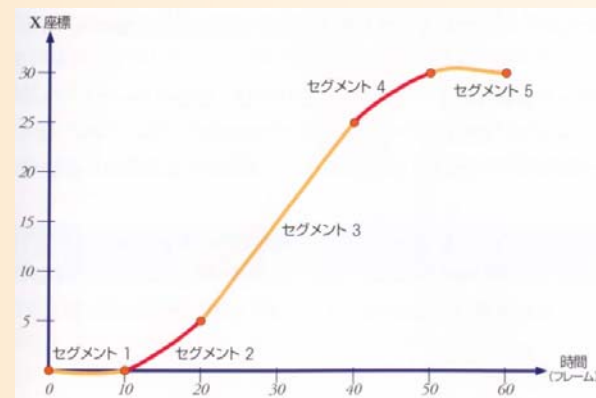
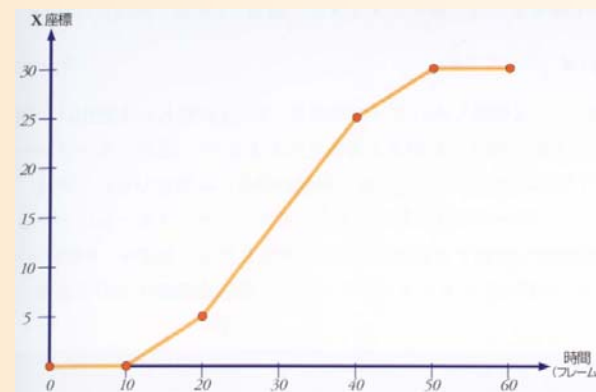
アニメーションの原理

- 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,位置)の組を指定し、動きのデータを生成
 - 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
 - ベジエ曲線、スプライン曲線



参考書 図3.3

参考書 図3.9

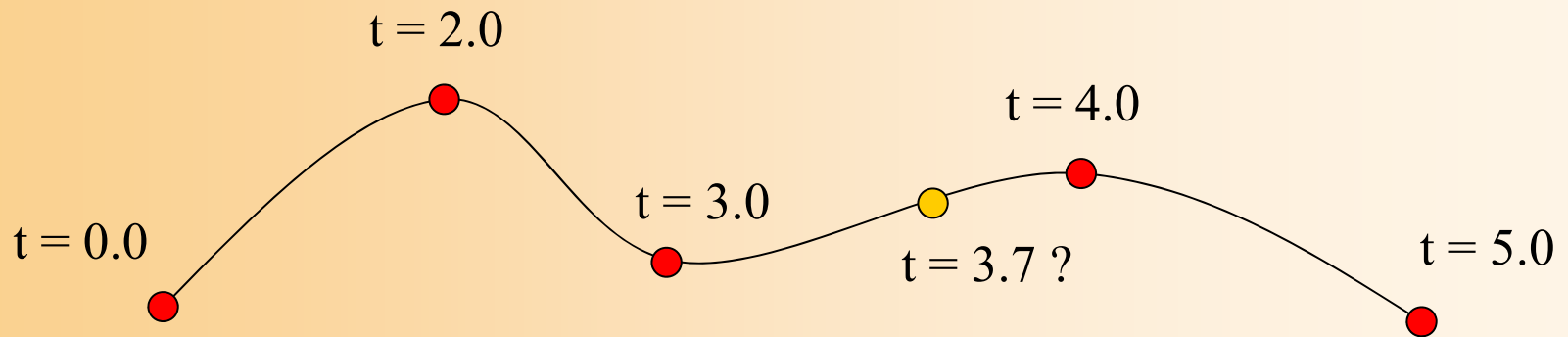
動力学シミュレーション

- 物理法則に従ったアニメーション
 - 実際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次回紹介



キーフレームアニメーションの方法

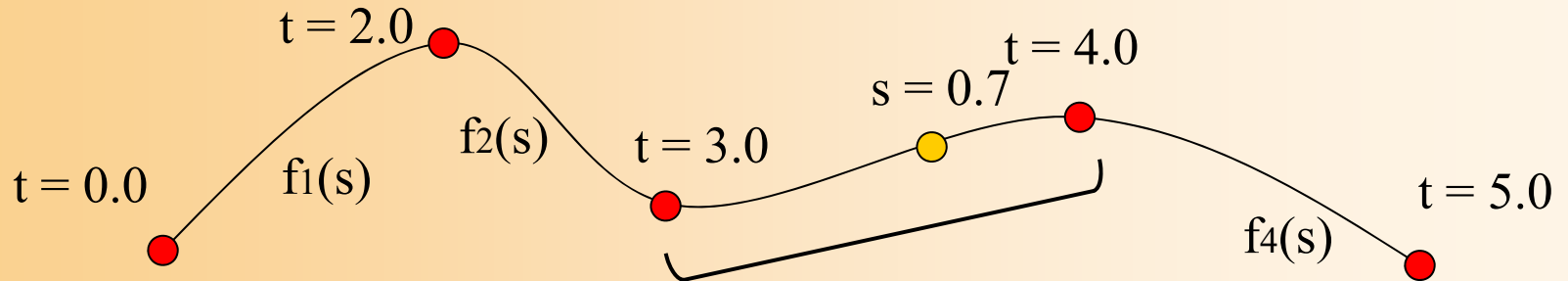
- キーフレームにおけるオブジェクトの位置・向きを補間して、全フレームの位置・向きを計算
- 位置・向きのそれぞれについて補間を行う必要がある
 - 位置と向きでは補間の方法が異なるため



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： $s = 0.0 \sim 1.0$ の範囲とする）



キーフレーム3と4の間の区間の軌道を表す関数 $f_3(s)$



位置・向き of 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

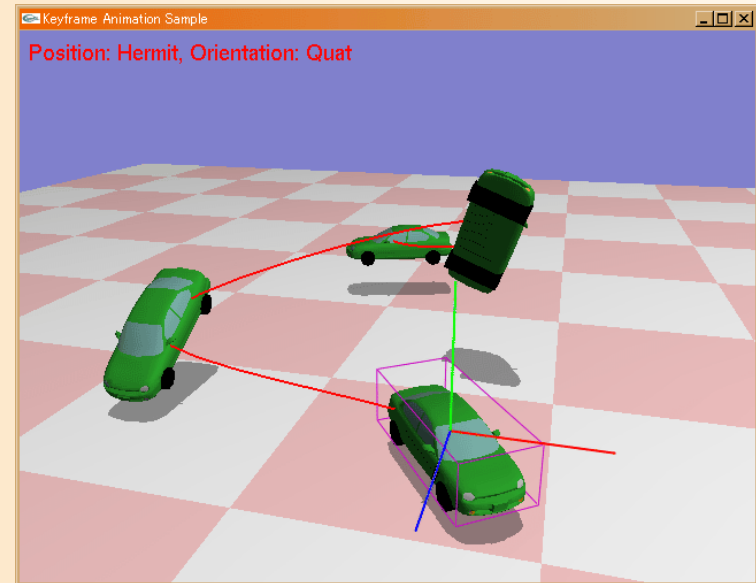




サンプルプログラム

デモプログラム

- キーフレームアニメーション
 - オブジェクトの配置、位置・向きの変更機能
 - オブジェクトの追加
 - マウス操作でオブジェクトの選択・移動・回転
 - アニメーション機能
 - 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



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 (keyframe_sample.cpp)
- 幾何形状の読み込み・描画関数 (Obj.h)
 - 第4・5回の授業で扱った内容
- オブジェクトの配置操作クラス (ObjectLayout クラス)
 - キー操作・マウス操作によって、キーフレームの物体の位置・向きの変更ができる
 - オブジェクトの数・位置・向きを取得するためのインターフェース(メソッド)を提供



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)
```

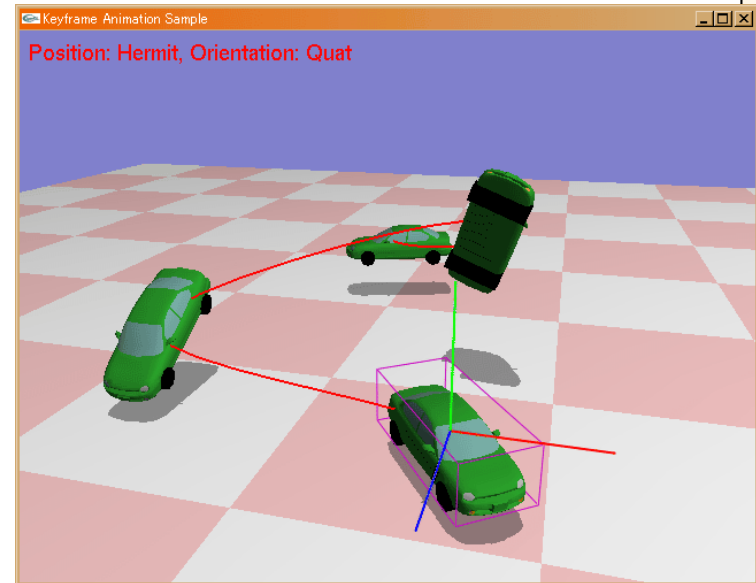
```
void RenderShadow( Obj * obj, float matrix[ 16 ], ... );
```

前回のレポート課題
(空欄に前回のレポート
課題の解答を記述すれ
ば、影が描画される)

オブジェクトの配置操作クラス(1)

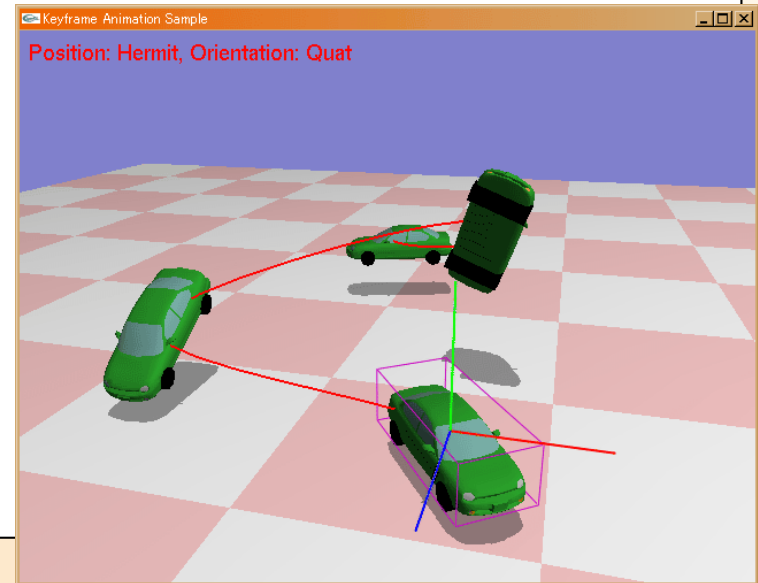
```
class ObjectLayout
{
    // オブジェクト情報を表す構造体
    struct Object
    {
        Point3f  pos; // 位置
        Matrix3f ori; // 向き
        Vector3f size; // サイズ(描画用)
    };
    // 全オブジェクトの情報
    vector< Object >  objects;

    // 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)

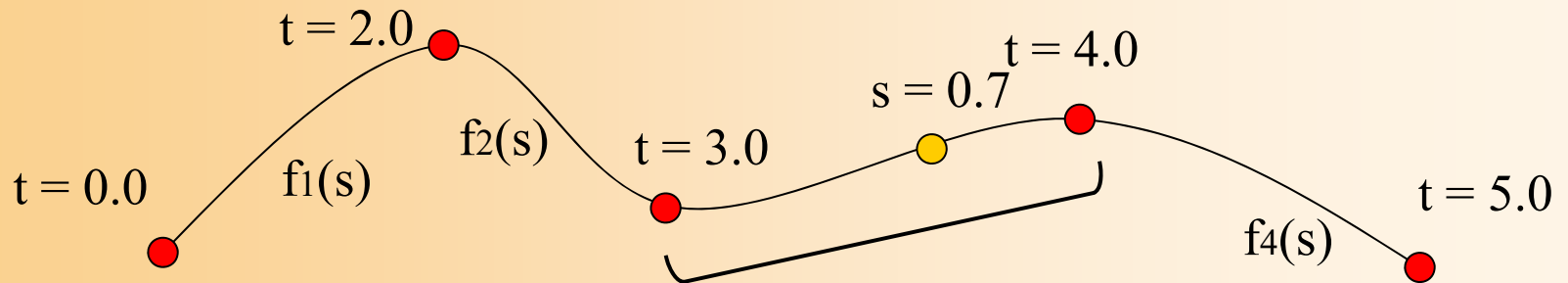
```
.....  
// オブジェクト位置・視点の変更を通知  
void Update();  
// マウス移動時の処理  
void OnMouseMove( int mx, int my );  
// マウスのボタンが押された時の処理  
void OnMouseDown( int mx, int my );  
// マウスのボタンが離された時の処理  
void OnMouseUp( int mx, int my );  
  
// オブジェクトの個数を取得  
size_t GetNumObjects();  
// 各オブジェクトの位置・向きを取得  
Point3f & GetPosition( int no );  
Matrix3f & GetOrientation( int no );  
Matrix4f & GetFrame( int no );  
}
```



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： $s = 0.0 \sim 1.0$ の範囲とする）



キーフレーム3と4の間の区間の軌道を表す関数 $f_3(s)$



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
};
```

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)

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

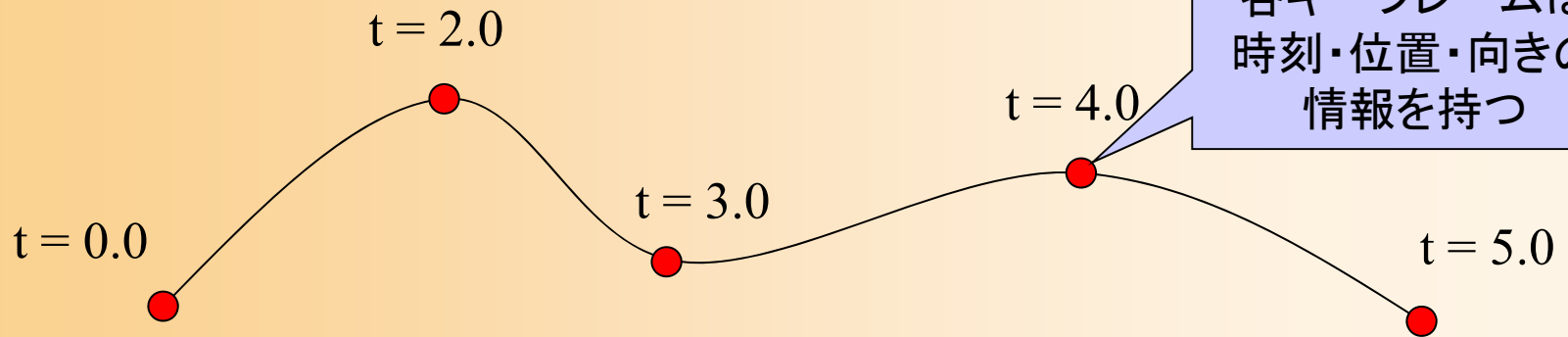
```
// 使用する位置・向き補間方法
PositionInterpolationEnum    pos_method = PI_LINEAR;
OrientationInterpolationEnum ori_method = OI_EULAR;
```

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;
```



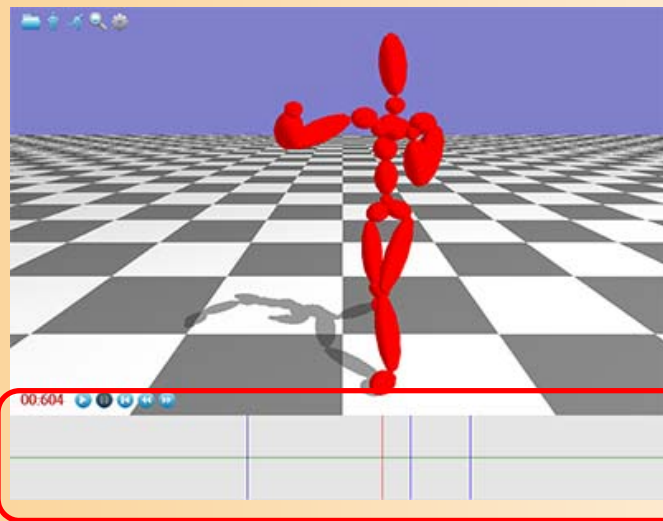
キーフレーム情報の設定

- 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、キーフレームとして使用する
- 今回のプログラムでは、各キーフレームの時間は固定とする
 - $t = 1.0$ 秒, 2.0 秒, 3.0 秒, ... n 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間は固定とする ($t = 1, 2, 3, \dots, n$)
 - タイムラインを使ったキーフレームの時間操作



タイムライン



位置・向き of 補間

- 位置・向き of 補間 (UpdateModelMat() 関数)
 - 入力: 時刻
出力: 4×4 変換行列 (位置・向き of 情報を含む)
 - 処理手順
 - 入力時刻が、何番目 of 区間に相当するかを判定
 - 現在 of 区間での位置・向きを計算するための制御点 of 情報を、キーフレーム情報から取得
 - 制御点と補間関数にもとづき、入力時刻における位置・向きを求める
 - サンプルプログラムでは一部 of 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 of 補間関数

```
void UpdateModelMat( 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_HERMITE )
        // 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
```

時刻を入力すると、その時刻での位置・向きを補間により計算し、4×4行列として返す

現在は i 番目のキーフレームの時刻が i と仮定

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)

物体の更新と描画

```
// 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( animation_time, model_mat );  
}
```

現在時刻の位置・向きを表す4×4行列を計算

```
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 );
```

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)

影の描画関数を呼び出し



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の関数は全て配列として渡すようになっており、行列・ベクトルの表現は決まっていない
- 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

- 座標 (Point3) とベクトル (Vector3) の使い分け

- 変換行列をかけると、Point3 には並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
 - テンプレートライブラリ、高速
 - 任意次元の行列・ベクトルの表現・演算
 - 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



サンプルプログラムでの 行列・ベクトルの扱い

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



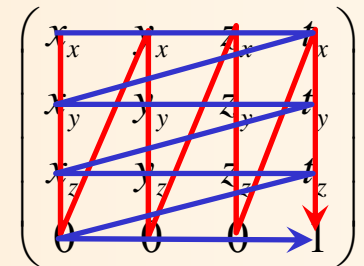
vecmathの利用方法

- 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の設定
- vecmathのヘッダファイルをインクルード
- 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)



行列の扱いに際しての注意

- 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
- 右手系 or 左手系
 - OpenGLでは右手系
- 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



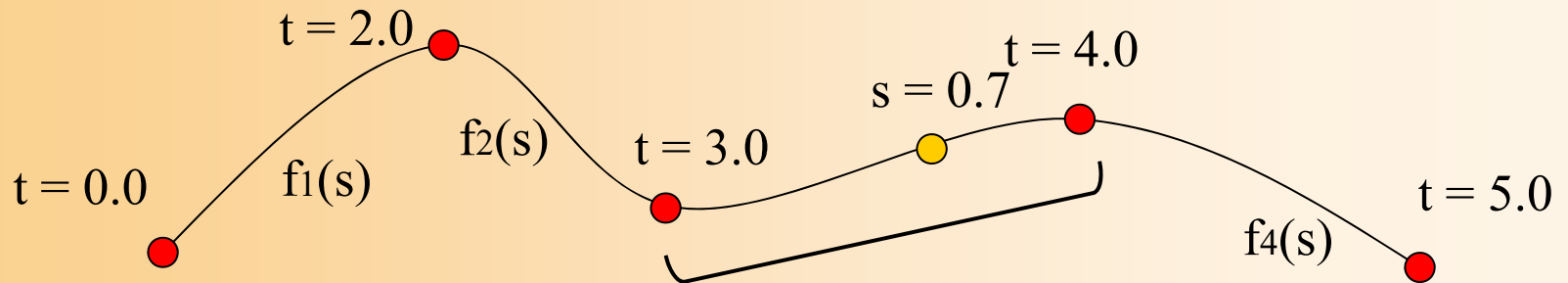


位置補間

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: $s = 0.0 \sim 1.0$ の範囲とする)



キーフレーム3と4の間の区間の軌道を表す関数 $f_3(s)$



位置・向き of 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

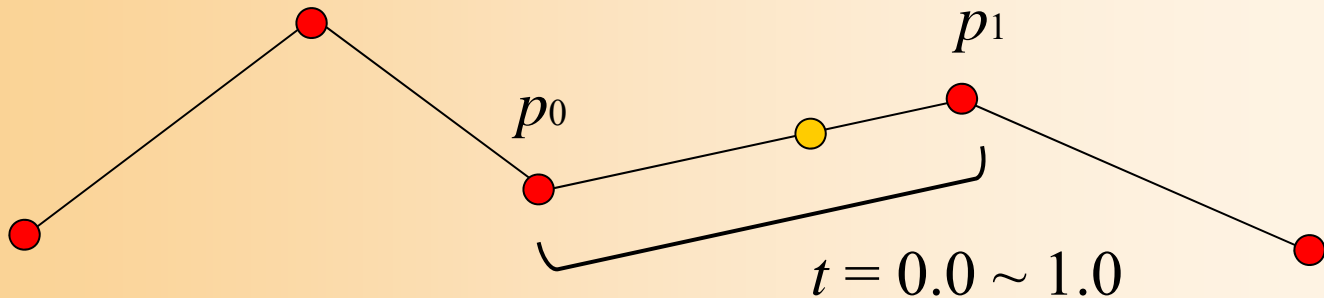


線形補間

- 区間の両端点間を線形に補間

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

- 直線的な動き
- キーフレームで動きが急に変わる



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.scaleAdd( t, p1 - p0, p0 );
```

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



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計算

- p はベクトルでも良い (xyz のそれぞれを補間)

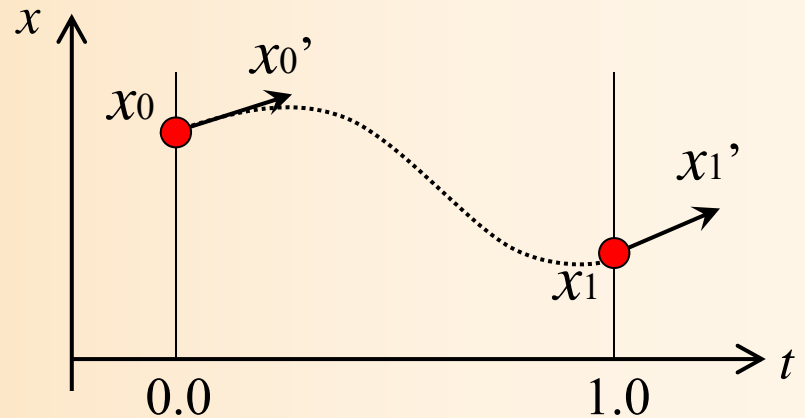
$$p(t) = H_0(t) \cdot p_0 + H_1(t) \cdot p_1 + h_0(t) \cdot p'_0 + h_1(t) \cdot p'_1$$

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

$$H_1(t) = -2t^3 + 3t^2 = t^2 (3-2t)$$

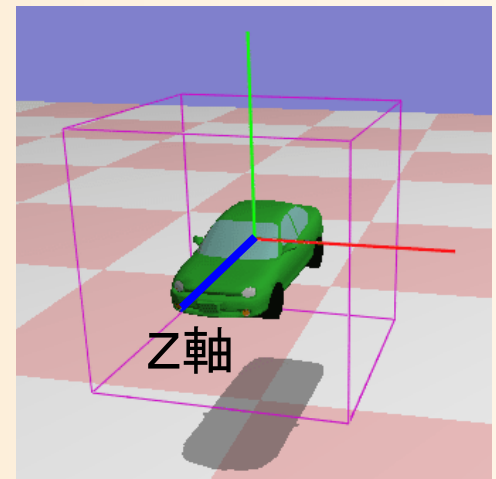
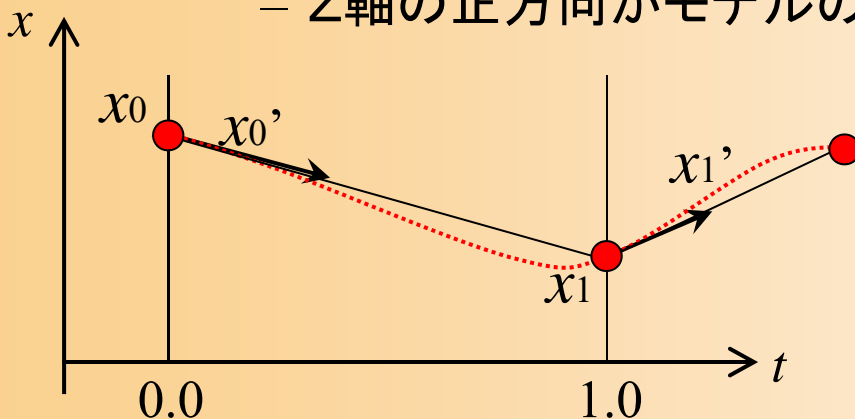
$$h_0(t) = t^3 - 2t^2 + t = (t-1)^2 t$$

$$h_1(t) = t^3 - t^2 = (t-1)t^2$$



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軸の向きを傾きとする

プログラム例(2)

- vecmath を使った計算方法の例

.....

// 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;

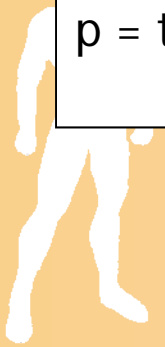
c = v0;

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



Bézier曲線

- Bézier曲線 (ベジエ曲線)

- 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 ($p_3=p_0'$)

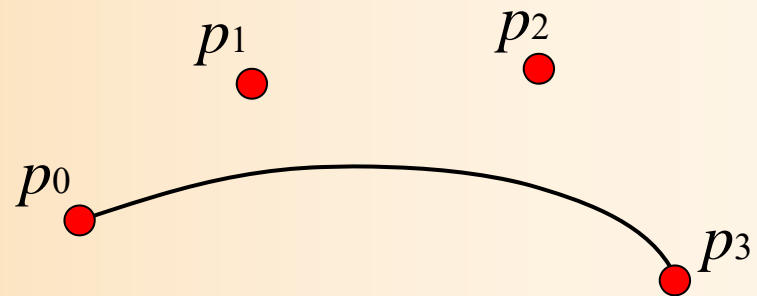
$$p(t) = X_0(t) \cdot p_0 + X_1(t) \cdot p_1 + X_2(t) \cdot p_2 + X_3(t) \cdot p_3$$

$$X_0(t) = (t-1)^3$$

$$X_1(t) = 3(t-1)^2 t$$

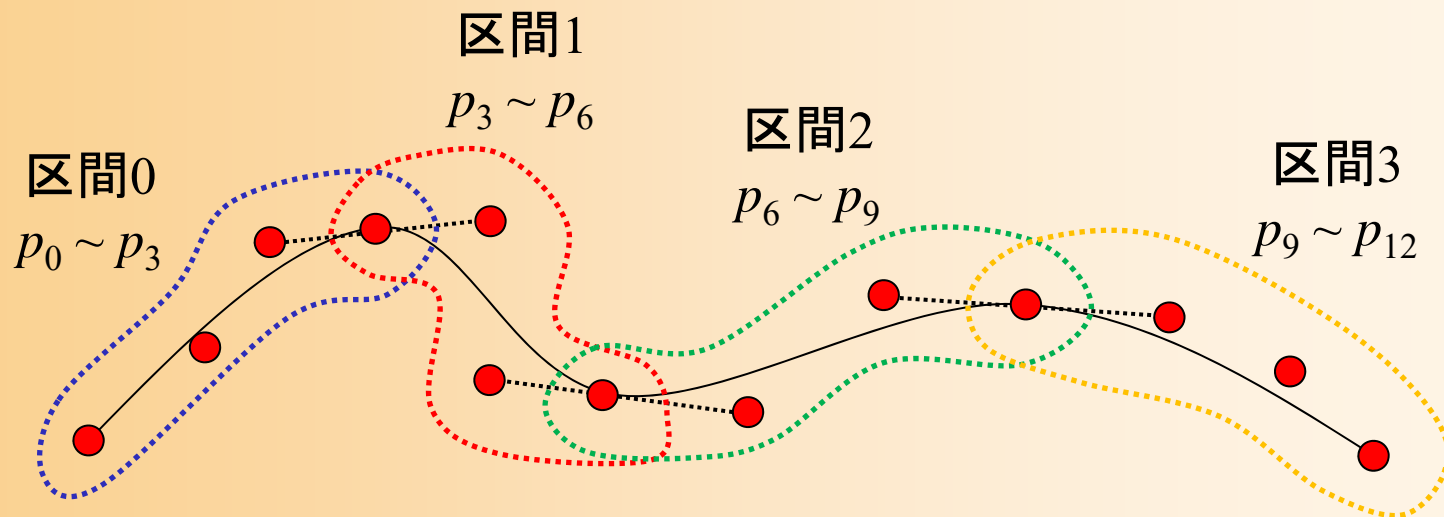
$$X_2(t) = 3(t-1)t^2$$

$$X_3(t) = t^3$$



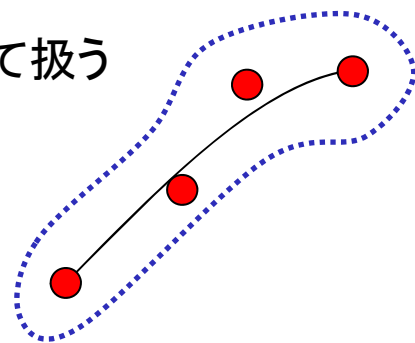
Bézier曲線の区間

- 連続する4点を1つの区間とする
 - $i \times 3 \sim i \times 3 + 3$ 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
 - 隣接区間を滑らかに接続するためには、境界の前後の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 );
```



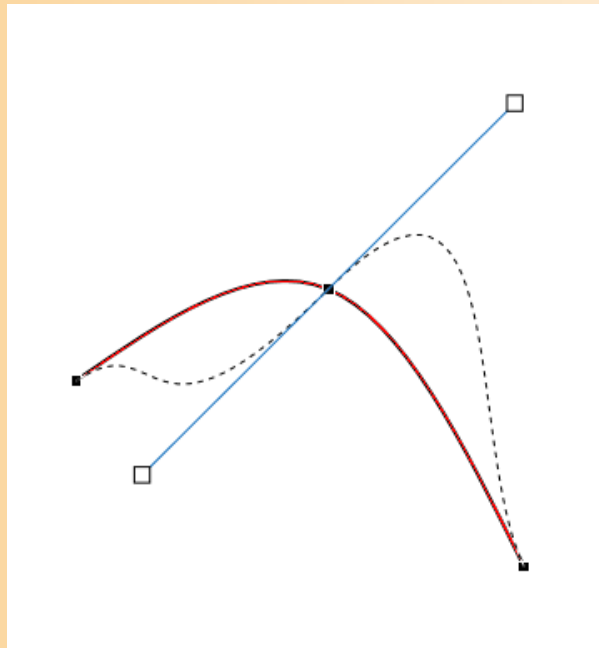
```
// Bezier補間の区間の4つの制御点(両端点と、中間の2つの点)の位置を取得
const Point3f & p0 = keyframes[ bezier_seg_no * 3 ].pos;
const Point3f & p1 = keyframes[ bezier_seg_no * 3 + 1 ].pos;
const Point3f & p2 = keyframes[ bezier_seg_no * 3 + 2 ].pos;
const Point3f & p3 = keyframes[ bezier_seg_no * 3 + 3 ].pos;
```

```
// 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

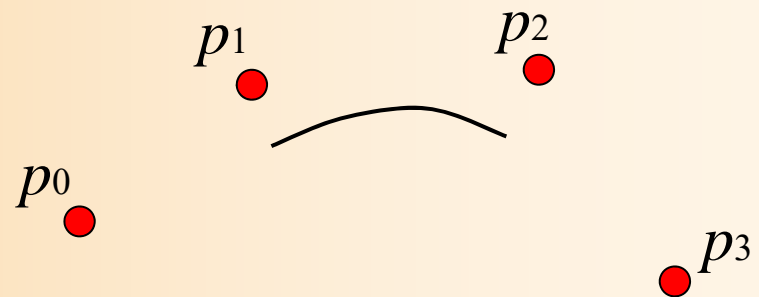
- B-Spline曲線 (Bスプライン曲線)
 - 4点の位置を使用
 - 接続が非常に容易
 - 前の区間の p_1, p_2, p_3 に、次の点 p_4 点を加えて次の区間にすれば、前後の区間がなめらかにつながる

$$X_0(t) = \frac{1}{6}t^3 + \frac{1}{2}t^2 + \frac{1}{2}t + \frac{1}{6} \quad p(t) = X_0(t) \cdot p_0 + X_1(t) \cdot p_1 + X_2(t) \cdot p_2 + X_3(t) \cdot p_3$$

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

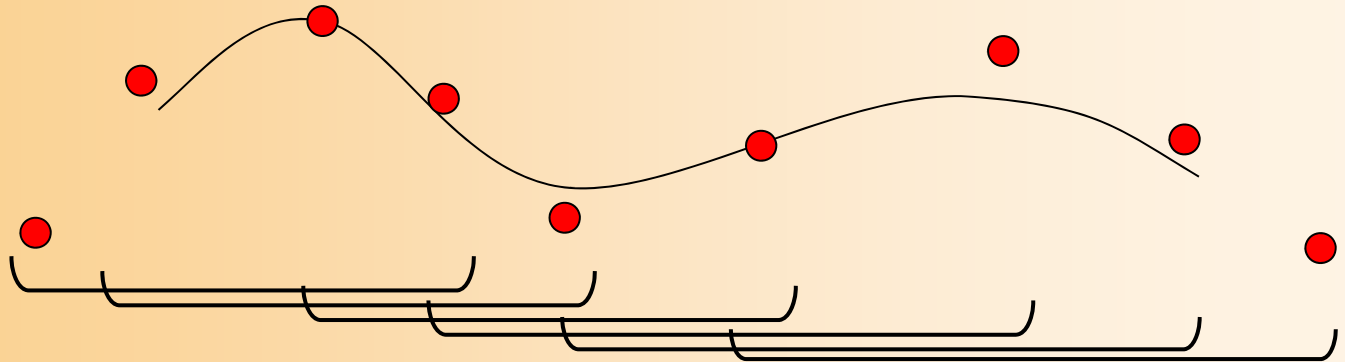
$$X_2(t) = -\frac{1}{2}t^3 + \frac{1}{2}t^2 + \frac{1}{2}t + \frac{1}{6}$$

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



B-Spline曲線の区間

- 連続する4点を1つの区間とする
 - $i - 1 \sim i + 3$ 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



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)

- 与えられた複数の点を通るような B-Spline曲線の制御点 (キーフレーム) 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」参照 (4.8節)



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ように、最適な制御点を計算により求める
- 対話的に軌道を変形したい時
 - Bézier曲線 or B-Spline曲線



今日の内容

- キーフレームアニメーション

- 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

- 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
 - 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
 - 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、Bézier、B-Spline 曲線
- 向き補間: 球面線形補間

