コンピュータアニメーション特論 プログラミング演習資料
第4・5回 キーフレームアニメーション

九州工業大学 情報工学研究院　尾下 真樹

2022 年度

1　サンプルプログラム

キーフレームアニメーションのサンプルプログラム（keyframe_sample.cpp）をもとに、4 通りの位置補間方法（線形補間、Hermite 補間、Bézier 補間、B-Spline 補間）と 2 通りの向き補間方法（オイラー角補間、四元数補間）を実現するプログラムを作成する。キーボードのスペースキーで、アニメーションモードとレイアウトモードの切り替えを行う。p キーで、位置の補間方法を、線形補間 → Hermite 補間 → Bézier 補間 → B-Spline 補間、の順に変更する。o キーで、向きの補間方法を、オイラー角補間 → 四元数補間、の順に変更する。f キーで、表示モードを、通常表示 → 軌道表示 → 全フレーム表示、の順に変更する。レイアウトモード中は、オブジェクトの中心をクリックし、操作軸上でドラッグすることでオブジェクトを移動・回転できる。オブジェクトをクリックする度に、移動・回転モードが、ワールド座標系での移動→ローカル座標系での回転→ローカル座標系での移動→ワールド座標系での回転、の順に切り替わる。a キーでオブジェクトを末尾に追加し、d キーで末尾のオブジェクトを削除する。

本サンプルプログラムは、keyframe_sample.cpp, obj.h, obj.cpp, ObjectLayout.h, ObjectLayout.cpp, vecmath_gl.h の 6 つのファイルから構成される。外部ライブラリとして、OpenGL と GLUT に加えて、行列・ベクトルなどを扱うための vecmath C++ ライブラリを使用する。最初に、メインとなるサンプルプログラム（keyframe_sample.cpp）のソースコード全体を示す。その後、サンプルプログラムの主要な処理や他のソースコードを説明する。

ソースコード 1: keyframe_sample.cpp

```
1  //
2  //  コンピュータアニメーション特論
3  //  キーフレームアニメーション サンプルプログラム
4  //
5
6
7  // 基本的なヘッダファイルのインクルード
8  #ifdef  _WIN32
9     #include <Windows.h>
10    #include <mmsystem.h>
11 #endif
12
13 #include <string.h>
14 #include <vector>
15
16 // GLUTヘッダファイルのインクルード
17 #include <GL/glut.h>
18
19 // vecmathヘッダファイルのインクルード
20 #include <Vector3.h>
21 #include <Point3.h>
22 #include <Matrix3.h>
23 #include <Matrix4.h>
```

```
24 #include <Quat4.h>
25 #include <Color4.h>
26 #include "vecmath_gl.h"
27
28 // 幾何形状オブジェクト、及び、読み込み・描画関数
29 #include "Obj.h"
30
31 // 複数オブジェクトの位置・向きをマウスで操作するためのクラス
32 #include "ObjectLayout.h"
33
34 // 標準算術関数・定数の定義
35 #define  _USE_MATH_DEFINES
36 #include <math.h>
37
38
39
40 //
41 //  カメラ・GLUTの入力処理に関するグローバル変数
42 //
43
44 // カメラの回転のための変数
45 static float    camera_yaw = 15.0f; // 30.0;      // Y軸を中心とする回転角度
46 static float    camera_pitch = -20.0f; // -30.0;   // X軸を中心とする回転角度
47 static float    camera_distance = 5.0f; // 15.0; // 中心からカメラの距離
48
49 // マウスのドラッグのための変数
50 static int      drag_mouse_r = 0; // 右ボタンがドラッグ中かどうかのフラグ（1:ドラッグ中,
       0:非ドラッグ中）
51 static int      drag_mouse_l = 0; // 左ボタンがドラッグ中かどうかのフラグ（1:ドラッグ中,
       0:非ドラッグ中）
52 static int      drag_mouse_m = 0; // 中ボタンがドラッグ中かどうかのフラグ（1:ドラッグ中,
       0:非ドラッグ中）
53 static int      last_mouse_x, last_mouse_y; // 最後に記録されたマウスカーソルの座標
54
55 // ウィンドウのサイズ
56 static int      win_width, win_height;
57
58
59 //
60 //  オブジェクトの配置・表示に関するグローバル変数
61 //
62
63 // 複数オブジェクトの位置・向きをマウスで操作するためのモジュール
64 ObjectLayout *  layout = NULL;
65
66 // 表示用の幾何形状オブジェクト
67 Obj *  object;
68 Vector3f  object_size;
69
70 // 点光源の位置（影の投影方向）
71 Vector3f  light_pos( 0.0f, 10.0f, 0.0f );
72
73 // 影の色
74 Color4f   shadow_color( 0.2f, 0.2f, 0.2f, 0.5f );
75
76
77
78 //
79 //  位置・向きの補間に関するグローバル変数
80 //
81
82 // 位置補間方法を表す列挙型
83 enum  PositionInterpolationEnum
84 {
```

```
85      PI_LINEAR,
86      PI_HERMIT,
87      PI_BEZIER,
88      PI_BSPLINE,
89      NUM_PI_METHOD
90  };
91
92  // 向き補間方法を表す列挙型
93  enum  OrientationInterpolationEnum
94  {
95      OI_NONE,
96      OI_EULER,
97      OI_QUAT,
98      NUM_OI_METHOD
99  };
100
101 // 位置補間方法の名前を表す文字列（表示用）
102 const char *  pi_name[] = {
103     "Linear", "Hermit", "Bezier", "B-Spline" };
104
105 // 向き補間方法の名前を表す文字列（表示用）
106 const char *  oi_name[] = {
107     "None", "Euler", "Quat" };
108
109 // 使用する位置・向き補間方法
110 PositionInterpolationEnum       pos_method = PI_LINEAR;
111 OrientationInterpolationEnum    ori_method = OI_EULER;
112
113
114 //
115 //  キーフレーム情報に関するグローバル変数
116 //
117
118 // キーフレーム情報
119 struct Keyframe
120 {
121     float     time; // 時刻
122     Point3f   pos;  // 位置
123     Matrix3f  ori;  // 向き
124 };
125
126 // 設定されている全キーフレーム情報（可変長配列）
127 vector< Keyframe >   keyframes;
128
129
130 //
131 //  アニメーション関連のグローバル変数
132 //
133
134 // アニメーション中かどうかを表すフラグ
135 bool   on_animation = false;
136
137 // 全フレーム描画モード・軌道描画モード
138 bool   on_draw_frames = false;
139 bool   on_draw_trajectory = true;
140
141 // アニメーションの再生時間
142 float  animation_time = 0.0f;
143
144 // 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
145 float  model_mat[ 16 ];
146
147
148
```

```
149 //
150 //  キーフレームアニメーションのための処理
151 //
152
153
154 //
155 //  オブジェクト配置にもとづいて全キーフレーム情報を更新
156 //
157 void  UpdateKeyframes()
158 {
159     Keyframe  key;
160
161     // キーフレーム数を設定
162     int  num_keyframes = layout->GetNumObjects();
163     keyframes.resize( num_keyframes );
164
165     // 各キーフレームの情報を設定
166     for ( int i=0; i<num_keyframes; i++ )
167     {
168         // i番目のキーフレームの時刻を i秒とする
169         key.time = (float) i;
170
171         // 位置・向きを設定
172         key.pos = layout->GetPosition( i );
173         key.ori = layout->GetOrientation( i );
174
175         // キーフレームの情報を設定
176         keyframes[ i ] = key;
177     }
178 }
179
180
181 //
182 //  回転行列からオイラー角への変換（yaw → pitch → roll の順の場合）（
        vecmathの行列を引数とする）
183 //
184 void  ConvMatToEuler( const Matrix3f & m, float & yaw, float & pitch, float & roll )
185 {
186     Vector3f  y_axis, z_axis;
187     m.getColumn( 1, &y_axis );
188     m.getColumn( 2, &z_axis );
189
190     yaw = atan2( z_axis.x, z_axis.z );
191
192     float  cos_yaw = cos( yaw );
193     pitch = atan2( -z_axis.y, sqrt( z_axis.x * z_axis.x + z_axis.z * z_axis.z ) );
194
195     float  sin_yaw = sin( yaw );
196     float  cos_pitch = cos( pitch );
197     roll = atan2( cos_pitch * ( sin_yaw * y_axis.z - cos_yaw * y_axis.x ), y_axis.y );
198 }
199
200
201 //
202 //  物体の位置・向きを更新
203 //  （キーフレーム数、キーフレーム配列、時刻を入力として、その時刻における位置・向きを表
        す変換行列を出力）
204 //
205 void  UpdateModelMat( int num_keyframes, const Keyframe * keyframes, float time, float
        mat[ 16 ] )
206 {
207     if ( num_keyframes <= 1 )
208         return;
209
```

```
210     // 指定時刻に対応する区間の番号と区間内での正規化時間（0.0～1.0）
211     int   seg_no = -1;
212     float   t = 0.0f;
213
214     // 指定時刻に対応する区間の番号を取得
215     for ( int i=0; i<num_keyframes-1; i++ )
216     {
217         // 指定時刻が i番目の区間に対応するかを判定
218         if ( ( time >= keyframes[ i ].time ) && ( time <= keyframes[ i+1 ].time ) )
219         {
220             seg_no = i;
221
222             // 区間内での正規化時間を計算
223             t = ( time - keyframes[ i ].time ) / ( keyframes[ i+1 ].time  - keyframes[ i
                    ].time );
224
225             break;
226         }
227     }
228     if ( seg_no == -1 )
229     {
230         // 最初のキーフレームより前の時刻が指定されたら、最初の区間の開始時刻を使用
231         if ( time < keyframes[ 0 ].time )
232         {
233             seg_no = 0;
234             t = 0.0f;
235         }
236         // 最後のキーフレームより後の時刻が指定されたら、最後の区間の終了時刻を使用
237         else
238         {
239             seg_no = num_keyframes - 2;
240             t = 1.0f;
241         }
242     }
243
244
245     // 指定時刻におけるオブジェクトの位置・向き
246     Vector3f   p;
247     Matrix3f   o;
248
249     // 位置を線形補間により計算
250     if ( pos_method == PI_LINEAR )
251     {
252         // 区間の両端点の位置を取得
253         const Point3f &   p0 = keyframes[ seg_no ].pos;
254         const Point3f &   p1 = keyframes[ seg_no + 1 ].pos;
255
256         // 両端点を線形に補間
257         p.scaleAdd( t, p1 - p0, p0 );
258
259         // 両端点を線形に補間（下記の書き方でも可）
260 //      p = t * ( p1 - p0 ) + p0;
261     }
262
263     // 位置をエルミート曲線により計算
264     else if ( pos_method == PI_HERMIT )
265     {
266         // 区間の両端点の位置を取得
267         const Point3f &   p0 = keyframes[ seg_no ].pos;
268         const Point3f &   p1 = keyframes[ seg_no + 1 ].pos;
269
270         // 区間の両端点の傾きを取得
271         Vector3f   v0, v1;
272         const Matrix3f &   o0 = keyframes[ seg_no ].ori;
```

```
273         const Matrix3f &  o1 = keyframes[ seg_no + 1 ].ori;
274         o0.getColumn( 2, &v0 );
275         o1.getColumn( 2, &v1 );
276         v0.negate();
277         v1.negate();
278 
279         // Hermite関数の値を計算
280         // 各自実装（式・プログラムは講義資料を参照）
281 
282         // ※レポート課題
283 
284         // p = ???;
285     }
286 
287     // 位置をエルミート曲線により計算
288     else if ( pos_method == PI_BEZIER )
289     {
290         // 指定時刻に対応するBezier補間の区間の番号と区間内での正規化時間（0.0～1.0）
291         int  bezier_seg_no = -1;
292         float  s = 0.0f;
293 
294         // Bezier補間の区間番号を計算
295         // 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
296         // 区間数×3＋1個のキーフレームが必要
297         bezier_seg_no = ( seg_no == 0 ) ? 0 : (int) floor( seg_no / 3 );
298 
299         // 最後の区間でキーフレーム数が4つに足りない場合は、前の区間の最後の時刻を使用
300         if ( ( bezier_seg_no + 1 ) * 3 + 1 > num_keyframes )
301         {
302             bezier_seg_no --;
303             s = 1.0f;
304         }
305         // 区間内での正規化時間（0.0～1.0）を計算
306         else
307         {
308             s = ( time - keyframes[ bezier_seg_no * 3 ].time ) / ( keyframes[
                    bezier_seg_no * 3 + 3 ].time - keyframes[ bezier_seg_no * 3 ].time );
309         }
310 
311         // 一つも区間が存在しない場合（キーフレーム数が3個以下の場合）は、最初のキーフレ
                ームの位置を出力
312         if ( num_keyframes < 4 )
313         {
314             p = keyframes[ 0 ].pos;
315         }
316         // Bezier補間を計算
317         else
318         {
319             // 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
320             const Point3f &  p0 = keyframes[ bezier_seg_no * 3 ].pos;
321             const Point3f &  p1 = keyframes[ bezier_seg_no * 3 + 1 ].pos;
322             const Point3f &  p2 = keyframes[ bezier_seg_no * 3 + 2 ].pos;
323             const Point3f &  p3 = keyframes[ bezier_seg_no * 3 + 3 ].pos;
324 
325             // Bezier関数の値を計算
326             // 各自実装（式は講義資料を参照）
327             // ※ 媒介変数は、t ではなく s を使うことに注意
328 
329             // ※レポート課題
330 
331             // p = ???;
332         }
333     }
334 
```

```
335     // 位置をB-Spline曲線により計算
336     else if ( pos_method == PI_BSPLINE )
337     {
338         // 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
339         int  k0, k1, k2, k3;
340         k0 = ( seg_no > 0 ) ? ( seg_no - 1 ) : seg_no;
341         k1 = seg_no;
342         k2 = seg_no + 1;
343         k3 = ( seg_no + 2 == num_keyframes ) ? ( seg_no + 1 ) : ( seg_no + 2);
344         const Point3f &  p0 = keyframes[ k0 ].pos;
345         const Point3f &  p1 = keyframes[ k1 ].pos;
346         const Point3f &  p2 = keyframes[ k2 ].pos;
347         const Point3f &  p3 = keyframes[ k3 ].pos;
348
349         // B-Spline 関数の値を計算
350         // 各自実装（式は講義資料を参照）
351
352         // ※レポート課題
353
354         // p = ???;
355     }
356
357
358     // 向きの補間なし
359     if ( ori_method == OI_NONE )
360     {
361         o = layout->GetOrientation( seg_no );
362     }
363
364     // 向きをオイラー角の線形補間により計算
365     else if ( ori_method == OI_EULER )
366     {
367         // 区間の両端点の向きを取得
368         const Matrix3f &  o0 = keyframes[ seg_no ].ori;
369         const Matrix3f &  o1 = keyframes[ seg_no + 1 ].ori;
370
371         // オイラー角に変換
372         float  y0, p0, r0;
373         float  y1, p1, r1;
374         ConvMatToEuler( o0, y0, p0, r0 );
375         ConvMatToEuler( o1, y1, p1, r1 );
376
377         // 各回転角度を線形補間
378         float  y, p, r;
379         if ( y0 < y1 - M_PI )
380             y0 += 2.0f * M_PI;
381         else if ( y0 > y1 + M_PI )
382             y0 -= 2.0f * M_PI;
383         y = ( y1 - y0 ) * t + y0;
384         p = ( p1 - p0 ) * t + p0;
385         r = ( r1 - r0 ) * t + r0;
386
387         // 行列に変換
388         Matrix3f  rot;
389         o.rotY( y );
390         rot.rotX( p );
391         o.mul( o, rot );
392         rot.rotZ( r );
393         o.mul( o, rot );
394     }
395
396     // 向きを四元数の球面線形補間により計算
397     else if ( ori_method == OI_QUAT )
398     {
```

```
399         // 区間の両端点の向きを取得
400         const Matrix3f &  o0 = keyframes[ seg_no ].ori;
401         const Matrix3f &  o1 = keyframes[ seg_no + 1 ].ori;
402 
403         // 四元数を使って球面線形補間を計算
404         // 各自実装（式は講義資料を参照）
405         // 球面線形補間は、vecmath の Quat4fクラスの interpolate関数で計算できる
406 
407         // ※レポート課題
408 
409         // q = ???;
410     }
411 
412     // オブジェクトの位置・向きを表す変換行列を配列にコピー
413     Matrix4f  f;
414     f.set( o, p, 1.0f );
415     f.transpose();
416     memcpy( mat, &f.m00, sizeof( float ) * 16 );
417 }
418 
419 
420 
421 //
422 //  以下、プログラムのメイン処理
423 //
424 
425 
426 //
427 //  あらかじめ定義されたオブジェクト配置を設定
428 //
429 void  SetupScene( int no )
430 {
431     if ( !layout )
432         return;
433 
434     Matrix3f  ori, rot;
435 
436     if ( no == 1 )
437     {
438         layout->DeleteAllObjects();
439         layout->AddObject();
440         layout->SetObjectSize( 0, object_size );
441         layout->SetObjectPos( 0, Point3f( -1.0f, 0.5f, -1.5f ) );
442         ori.rotY( M_PI * 0.6f );
443         layout->SetObjectOri( 0, ori );
444 
445         layout->AddObject();
446         layout->SetObjectSize( 1, object_size );
447         layout->SetObjectPos( 1, Point3f( 0.0f, 1.0f, 0.0f ) );
448         ori.rotY( M_PI * 1.2f );
449         rot.rotX( M_PI / 3.0f );
450         ori.mul( ori, rot );
451         rot.rotZ( M_PI / 4.0f );
452         ori.mul( ori, rot );
453         layout->SetObjectOri( 1, ori );
454 
455         layout->AddObject();
456         layout->SetObjectSize( 2, object_size );
457         layout->SetObjectPos( 2, Point3f( -2.0f, 0.5f, 1.0f ) );
458         ori.rotY( M_PI );
459         rot.rotX( - M_PI / 6.0f );
460         ori.mul( ori, rot );
461         layout->SetObjectOri( 2, ori );
462 
```

```
463         layout->AddObject();
464         layout->SetObjectSize( 3, object_size );
465         layout->SetObjectPos( 3, Point3f( 0.0f, 0.4f, 1.5f ) );
466         ori.rotY( M_PI * 1.2f );
467         layout->SetObjectOri( 3, ori );
468
469         layout->AddObject();
470         layout->SetObjectSize( 4, object_size );
471         layout->SetObjectPos( 4, Point3f( 1.0f, 0.35f, 1.75f ) );
472         ori.rotY( M_PI * -0.4f );
473         rot.rotX( M_PI / 12.0f );
474         ori.mul( ori, rot );
475         layout->SetObjectOri( 4, ori );
476
477         layout->AddObject();
478         layout->SetObjectSize( 5, object_size );
479         layout->SetObjectPos( 5, Point3f( 1.5f, 0.6f, 0.0f ) );
480         ori.rotY( 0.0f );
481         layout->SetObjectOri( 5, ori );
482
483         layout->AddObject();
484         layout->SetObjectSize( 6, object_size );
485         layout->SetObjectPos( 6, Point3f( 1.0f, 1.0f, -1.0f ) );
486         ori.rotY( M_PI / 3 );
487         layout->SetObjectOri( 6, ori );
488
489         camera_yaw = 15.0f;
490         camera_pitch = -20.0f;
491         camera_distance = 6.0f;
492     }
493     else if ( no == 2 )
494     {
495         layout->DeleteAllObjects();
496
497         layout->AddObject();
498         layout->SetObjectSize( 0, object_size );
499         layout->SetObjectPos( 0, Point3f(-1.0f, 0.5f, 0.0f ) );
500         ori.rotY( M_PI * 0.6f );
501         layout->SetObjectOri( 0, ori );
502
503         layout->AddObject();
504         layout->SetObjectSize( 1, object_size );
505         layout->SetObjectPos( 1, Point3f( 1.0f, 0.5f, 0.0f ) );
506         ori.rotY( M_PI * 1.2f );
507         rot.rotX( M_PI / 3.0f );
508         ori.mul( ori, rot );
509         rot.rotZ( M_PI / 4.0f );
510         ori.mul( ori, rot );
511         layout->SetObjectOri( 1, ori );
512
513         camera_yaw = 0.0f;
514         camera_pitch = -20.0f;
515         camera_distance = 5.0f;
516     }
517
518     // オブジェクト配置にもとづいて全キーフレーム情報を更新
519     UpdateKeyframes();
520 }
521
522
523 //
524 //  格子模様の床を描画
525 //
526 void  DrawFloor( float tile_size, int num_x, int num_z, float r0, float g0, float b0,
```

```
    float r1, float g1, float b1 )
{
   int  x, z;
   float  ox, oz;

   glBegin( GL_QUADS );
   glNormal3d( 0.0, 1.0, 0.0 );

   ox = - ( num_x * tile_size ) / 2;
   for ( x=0; x<num_x; x++ )
   {
      oz = - ( num_z * tile_size ) / 2;
      for ( z=0; z<num_z; z++ )
      {
         if ( ( (x + z) % 2 ) == 0 )
            glColor3f( r0, g0, b0 );
         else
            glColor3f( r1, g1, b1 );

         glTexCoord2d( 0.0f, 0.0f );
         glVertex3d( ox, 0.0, oz );
         glTexCoord2d( 0.0f, 1.0f );
         glVertex3d( ox, 0.0, oz + tile_size );
         glTexCoord2d( 1.0f, 1.0f );
         glVertex3d( ox + tile_size, 0.0, oz + tile_size );
         glTexCoord2d( 1.0f, 0.0f );
         glVertex3d( ox + tile_size, 0.0, oz );

         oz += tile_size;
      }
      ox += tile_size;
   }
   glEnd();
}


//
//  幾何形状モデル（Obj形状）の影の描画
//
void  RenderShadow( Obj * obj, const float matrix[ 16 ] )
{
   RenderObjShadow( obj, matrix, light_pos.x, light_pos.y, light_pos.z, shadow_color.x,
       shadow_color.y, shadow_color.z, shadow_color.w );
}

void  RenderShadow( Obj * obj, Matrix4f & mat )
{
   Matrix4f  frame( mat );
   frame.transpose();
   RenderObjShadow( obj, &frame.m00, light_pos.x, light_pos.y, light_pos.z,
       shadow_color.x, shadow_color.y, shadow_color.z, shadow_color.w );
}


//
//  テキストを描画
//
void  DrawTextInformation( int line_no, const char * message )
{
   if ( message == NULL )
      return;

   // 射影行列を初期化（初期化の前に現在の行列を退避）
   glMatrixMode( GL_PROJECTION );
```

```
588     glPushMatrix();
589     glLoadIdentity();
590     gluOrtho2D( 0.0, win_width, win_height, 0.0 );
591
592     // モデルビュー行列を初期化（初期化の前に現在の行列を退避）
593     glMatrixMode( GL_MODELVIEW );
594     glPushMatrix();
595     glLoadIdentity();
596
597     // Zバッファ・ライティングはオフにする
598     glDisable( GL_DEPTH_TEST );
599     glDisable( GL_LIGHTING );
600
601     // メッセージの描画
602     glColor3f( 1.0, 0.0, 0.0 );
603     glRasterPos2i( 16, 40 + 24 * line_no );
604     for ( int i=0; message[i]!='\0'; i++ )
605        glutBitmapCharacter( GLUT_BITMAP_HELVETICA_18, message[i] );
606
607     // 設定を全て復元
608     glEnable( GL_DEPTH_TEST );
609     glEnable( GL_LIGHTING );
610     glMatrixMode( GL_PROJECTION );
611     glPopMatrix();
612     glMatrixMode( GL_MODELVIEW );
613     glPopMatrix();
614 }
615
616
617 //
618 //  画面描画時に呼ばれるコールバック関数
619 //
620 void  DisplayCallback( void )
621 {
622     // 画面をクリア
623     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
624
625     // 変換行列を設定（ワールド座標系→カメラ座標系）
626     glMatrixMode( GL_MODELVIEW );
627     glLoadIdentity();
628     glTranslatef( 0.0,  0.0, - camera_distance );
629     glRotatef( - camera_pitch, 1.0, 0.0, 0.0 );
630     glRotatef( - camera_yaw, 0.0, 1.0, 0.0 );
631     glTranslatef( 0.5,  0.0, 0.0 ); // ワールド座標系での注視点（適当な位置を設定）
632
633     // 光源の位置を更新
634     float  light0_position[] = { light_pos.x, light_pos.y, light_pos.z, 1.0 };
635     glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
636
637     // 格子模様の床を描画
638     DrawFloor( 1.0f, 50, 50, 1.0f, 1.0f, 1.0f, 1.0f, 0.8f, 0.8f );
639
640     // オブジェクトの軌道を描画
641     if ( on_draw_trajectory )
642     {
643        // オブジェクトの軌道を描画
644        float  mat[ 16 ];
645        glDisable( GL_LIGHTING );
646        glLineWidth( 2.0 );
647        glColor3f( 1.0f, 0.0f, 0.0f );
648        glBegin( GL_LINE_STRIP );
649        for ( float t=0.0f; t<=layout->GetNumObjects()-1.0f+0.001f; t+=0.1f )
650        {
651           UpdateModelMat( keyframes.size(), &keyframes.front(), t, mat );
```

```
652             glVertex3f( mat[12], mat[13], mat[14] );
653         }
654         glEnd();
655         glEnable( GL_LIGHTING );
656 
657         // キーフレームのオブジェクトを描画
658         if ( on_animation )
659         {
660             for ( int i=0; i<layout->GetNumObjects(); i++ )
661             {
662                 // オブジェクトを描画
663                 glPushMatrix();
664                     glMultMatrixf( layout->GetFrame( i ) );
665                     RenderObj( object );
666                 glPopMatrix();
667 
668                 // オブジェクトの影を描画
669                 RenderShadow( object, layout->GetFrame( i ) );
670             }
671         }
672     }
673     // 全フレームのオブジェクトを描画
674     else if ( on_draw_frames )
675     {
676         float  mat[ 16 ];
677         for ( float t=0.0f; t<=layout->GetNumObjects()-1.0f+0.001f; t+=0.2f )
678         {
679             UpdateModelMat( keyframes.size(), &keyframes.front(), t, mat );
680 
681             // オブジェクトを描画
682             glPushMatrix();
683                 glMultMatrixf( mat );
684                 RenderObj( object );
685             glPopMatrix();
686 
687             // オブジェクトの影を描画
688             RenderShadow( object, mat );
689         }
690     }
691 
692     // アニメーション中のオブジェクトを描画
693     if ( on_animation && !on_draw_frames )
694     {
695         // アニメーション中のオブジェクトを描画
696         glPushMatrix();
697             glMultMatrixf( model_mat );
698             RenderObj( object );
699         glPopMatrix();
700 
701         // オブジェクトの影を描画
702         RenderShadow( object, model_mat );
703     }
704     // 編集モード中の描画
705     else
706     {
707         // 各オブジェクトを描画
708         for ( int i=0; i<layout->GetNumObjects(); i++ )
709         {
710             glPushMatrix();
711                 glMultMatrixf( layout->GetFrame( i ) );
712                 RenderObj( object );
713             glPopMatrix();
714 
715             // オブジェクトの影を描画
```

```
716             RenderShadow( object, layout->GetFrame( i ) );
717         }
718
719         // 操作用の情報を描画
720         layout->Render();
721     }
722
723     // 現在の描画モードを表示
724     if ( on_animation )
725         DrawTextInformation( 0, "Animation Mode" );
726     else
727         DrawTextInformation( 0, "Layout Mode" );
728
729     // 現在の補間モードを表示
730     if ( on_draw_frames || on_draw_trajectory || on_animation )
731     {
732         char  message[ 64 ] = "";
733         sprintf( message, "Position: %s, Orientation: %s", pi_name[ pos_method ], oi_name
                [ ori_method ] );
734         DrawTextInformation( 1, message );
735     }
736     // 現在の操作モードを表示
737     else
738     {
739         DrawTextInformation( 1, layout->GetOperationMode() );
740     }
741
742     // 現在の時刻を表示
743     if ( on_animation && !on_draw_frames )
744     {
745         char  message[ 64 ] = "";
746         sprintf( message, "Time: %2.2f", animation_time );
747         DrawTextInformation( 2, message );
748     }
749
750     // バックバッファに描画した画面をフロントバッファに表示
751     glutSwapBuffers();
752 }
753
754
755 //
756 //  ウィンドウサイズ変更時に呼ばれるコールバック関数
757 //
758 void  ReshapeCallback( int w, int h )
759 {
760     // ウィンドウ内の描画を行う範囲を設定（ここではウィンドウ全体に描画）
761     glViewport(0, 0, w, h);
762
763     // カメラ座標系→スクリーン座標系への変換行列を設定
764     glMatrixMode( GL_PROJECTION );
765     glLoadIdentity();
766     gluPerspective( 45, (double)w/h, 1, 500 );
767
768     // ウィンドウのサイズを記録（テキスト描画処理のため）
769     win_width = w;
770     win_height = h;
771 }
772
773
774 //
775 //  マウスクリック時に呼ばれるコールバック関数
776 //
777 void  MouseClickCallback( int button, int state, int mx, int my )
778 {
```

```
779     // 左ボタンが押されたらドラッグ開始
780     if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_DOWN ) )
781         drag_mouse_l = 1;
782     // 左ボタンが離されたらドラッグ終了
783     else if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_UP ) )
784         drag_mouse_l = 0;
785
786     // 右ボタンが押されたらドラッグ開始
787     if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_DOWN ) )
788         drag_mouse_r = 1;
789     // 右ボタンが離されたらドラッグ終了
790     else if ( ( button == GLUT_RIGHT_BUTTON ) && ( state == GLUT_UP ) )
791         drag_mouse_r = 0;
792
793     // 中ボタンが押されたらドラッグ開始
794     if ( ( button == GLUT_MIDDLE_BUTTON ) && ( state == GLUT_DOWN ) )
795         drag_mouse_m = 1;
796     // 中ボタンが離されたらドラッグ終了
797     else if ( ( button == GLUT_MIDDLE_BUTTON ) && ( state == GLUT_UP ) )
798         drag_mouse_m = 0;
799
800     // シーン配置機能に左クリックを通知
801     if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_DOWN ) )
802         layout->OnMouseDown( mx, my );
803     else if ( ( button == GLUT_LEFT_BUTTON ) && ( state == GLUT_UP ) )
804         layout->OnMouseUp( mx, my );
805
806     // 再描画
807     glutPostRedisplay();
808
809     // 現在のマウス座標を記録
810     last_mouse_x = mx;
811     last_mouse_y = my;
812 }
813
814
815 //
816 // マウス移動時に呼ばれるコールバック関数
817 //
818 void  MouseMotionCallback( int mx, int my )
819 {
820     // シーン配置機能にマウス移動を通知
821     if ( layout )
822         layout->OnMoveMouse( mx, my );
823
824     // 再描画
825     glutPostRedisplay();
826 }
827
828
829 //
830 //  マウスドラッグ時に呼ばれるコールバック関数
831 //
832 void  MouseDragCallback( int mx, int my )
833 {
834     // 右ボタンのドラッグ中は視点を回転する
835     if ( drag_mouse_r )
836     {
837         // 前回のマウス座標と今回のマウス座標の差に応じて視点を回転
838
839         // マウスの横移動に応じてY軸を中心に回転
840         camera_yaw -= ( mx - last_mouse_x ) * 1.0;
841         if ( camera_yaw < 0.0 )
842             camera_yaw += 360.0;
```

```
843         else if ( camera_yaw > 360.0 )
844             camera_yaw -= 360.0;
845
846         // マウスの縦移動に応じてX軸を中心に回転
847         camera_pitch -= ( my - last_mouse_y ) * 1.0;
848         if ( camera_pitch < -90.0 )
849             camera_pitch = -90.0;
850         else if ( camera_pitch > 90.0 )
851             camera_pitch = 90.0;
852     }
853     // 中ボタンのドラッグ中は視点とカメラの距離を変更する
854     if ( drag_mouse_m )
855     {
856         // 前回のマウス座標と今回のマウス座標の差に応じて視点を回転
857
858         // マウスの縦移動に応じて距離を移動
859         camera_distance += ( my - last_mouse_y ) * 0.2;
860         if ( camera_distance < 2.0 )
861             camera_distance = 2.0;
862     }
863
864     // シーン配置機能にマウス移動を通知
865     if ( layout )
866     {
867         layout->Update();
868         layout->OnMoveMouse( mx, my );
869
870         // オブジェクト配置にもとづいて全キーフレーム情報を更新
871         UpdateKeyframes();
872     }
873
874     // 今回のマウス座標を記録
875     last_mouse_x = mx;
876     last_mouse_y = my;
877
878     // 再描画
879     glutPostRedisplay();
880 }
881
882
883 //
884 //  キーボードのキーが押されたときに呼ばれるコールバック関数
885 //
886 void  KeyboardCallback( unsigned char key, int mx, int my )
887 {
888     // s キーでアニメーションの停止・再開
889     if ( key == 's' )
890         on_animation = !on_animation;
891
892     // 数字キーであらかじめ定義されたオブジェクト配置を設定
893     if ( ( key >= '1' ) && ( key <= '9' ) )
894     {
895         SetupScene( key - '0' );
896     }
897
898     // スペースキーでアニメーションを開始
899     if ( key == ' ' )
900     {
901         on_animation = ! on_animation;
902         if ( on_animation )
903             animation_time = 0.0f;
904         on_draw_frames = false;
905     }
906
```

```
907     // pキーで位置補間方法を変更
908     if ( key == 'p' )
909     {
910         pos_method = (PositionInterpolationEnum)( ( pos_method + 1 ) % NUM_PI_METHOD );
911     }
912     // oキーで向き補間方法を変更
913     if ( key == 'o' )
914     {
915         ori_method = (OrientationInterpolationEnum)( ( ori_method + 1 ) % NUM_OI_METHOD
                );
916         if ( ori_method == OI_NONE )
917             ori_method = (OrientationInterpolationEnum)( ori_method + 1 );
918     }
919
920     // fキーで通常・描画全フレーム描画・軌道描画を切り替え
921     if ( key == 'f' )
922     {
923         if ( !on_draw_frames && !on_draw_trajectory )
924             on_draw_trajectory = true;
925         else if ( !on_draw_frames && on_draw_trajectory )
926         {
927             on_draw_frames = true;
928             on_draw_trajectory = false;
929         }
930         else
931             on_draw_frames = false;
932     }
933
934     // aキーでオブジェクトを追加
935     if ( key == 'a' )
936     {
937         layout->AddObject();
938
939         // オブジェクト配置にもとづいて全キーフレーム情報を更新
940         UpdateKeyframes();
941     }
942     // dキーでオブジェクトを削除
943     if ( key == 'd' )
944     {
945         layout->DeleteObject();
946
947         // オブジェクト配置にもとづいて全キーフレーム情報を更新
948         UpdateKeyframes();
949     }
950
951     // tキーで軸の描画モードを変更
952     if ( key == 't' )
953     {
954         bool &  flag = layout->GetRenderOption().enable_xray_mode;
955         flag = ! flag;
956     }
957
958     glutPostRedisplay();
959 }
960
961
962 //
963 //  アイドル時に呼ばれるコールバック関数
964 //
965 void  IdleCallback( void )
966 {
967     // アニメーション処理
968     if ( on_animation )
969     {
```

```
970 #ifdef  WIN32
971         // システム時間を取得し、前回からの経過時間に応じてΔｔを決定
972         static DWORD  last_time = 0;
973         DWORD  curr_time = timeGetTime();
974         float  delta = ( curr_time - last_time ) * 0.001f;
975         if ( delta > 0.1f )
976             delta = 0.1f;
977         last_time = curr_time;
978         animation_time += delta;
979 #else
980         // 固定のΔｔを使用
981         animation_time += 0.02f;
982 #endif
983         // アニメーションの繰り返し（最後まで再生が終わったら時間を０に戻す）
984         if ( animation_time >= layout->GetNumObjects() - 1 )
985             animation_time = 0.0f;
986 
987         // アニメーション中のオブジェクトの位置・向きを更新
988         UpdateModelMat( keyframes.size(), &keyframes.front(), animation_time, model_mat
                );
989 
990         // 再描画の指示を出す（この後で再描画のコールバック関数が呼ばれる）
991         glutPostRedisplay();
992     }
993 }
994 
995 
996 //
997 //  環境初期化関数
998 //
999 void  initEnvironment( void )
1000 {
1001     // 光源を作成する
1002     float  light0_position[] = { 0.0, 10.0, 0.0, 1.0 };
1003     float  light0_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
1004     float  light0_specular[] = { 1.0, 1.0, 1.0, 1.0 };
1005     float  light0_ambient[] = { 0.1, 0.1, 0.1, 1.0 };
1006     glLightfv( GL_LIGHT0, GL_POSITION, light0_position );
1007     glLightfv( GL_LIGHT0, GL_DIFFUSE, light0_diffuse );
1008     glLightfv( GL_LIGHT0, GL_SPECULAR, light0_specular );
1009     glLightfv( GL_LIGHT0, GL_AMBIENT, light0_ambient );
1010     glEnable( GL_LIGHT0 );
1011 
1012     // 光源計算を有効にする
1013     glEnable( GL_LIGHTING );
1014 
1015     // 物体の色情報を有効にする
1016     glEnable( GL_COLOR_MATERIAL );
1017 
1018     // Ｚテストを有効にする
1019     glEnable( GL_DEPTH_TEST );
1020 
1021     // 背面除去を有効にする
1022     glCullFace( GL_BACK );
1023     glEnable( GL_CULL_FACE );
1024 
1025     // 背景色を設定
1026     glClearColor( 0.5, 0.5, 0.8, 0.0 );
1027 
1028 
1029     // オブジェクトの読み込み
1030     object = LoadObj( "car.obj" );
1031     if ( !object || ( object->num_triangles == 0 ) )
1032     {
```

```
1033        // 読み込みに失敗したら終了
1034        printf( "Failed to load the object file." );
1035        exit( -1 );
1036    }
1037    ScaleObj( object, 1.0f, &object_size.x, &object_size.y, &object_size.z );
1038
1039    // オブジェクト配置機能の初期化
1040    layout = new ObjectLayout();
1041
1042    // あらかじめ定義されたオブジェクト配置を設定
1043    SetupScene( 1 );
1044 }
1045
1046
1047 //
1048 //  メイン関数（プログラムはここから開始）
1049 //
1050 int  main( int argc, char ** argv )
1051 {
1052    // GLUTの初期化
1053    glutInit( &argc, argv );
1054    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA | GLUT_STENCIL );
1055    glutInitWindowSize( 640, 640 );
1056    glutInitWindowPosition( 0, 0 );
1057    glutCreateWindow("Keyframe Animation");
1058
1059    // コールバック関数の登録
1060    glutDisplayFunc( DisplayCallback );
1061    glutReshapeFunc( ReshapeCallback );
1062    glutMouseFunc( MouseClickCallback );
1063    glutMotionFunc( MouseDragCallback );
1064    glutPassiveMotionFunc( MouseMotionCallback );
1065    glutKeyboardFunc( KeyboardCallback );
1066    glutIdleFunc( IdleCallback );
1067
1068    // 環境初期化
1069    initEnvironment();
1070
1071    // GLUTのメインループに処理を移す
1072    glutMainLoop();
1073    return 0;
1074 }
```

1.1 幾何形状モデルの読み込みと描画

本プログラムでは、キーフレームアニメーションを行う物体やキーフレームを表す物体を表示するために、Obj 形式の幾何形状モデルを読み込んで描画する。Obj 形式の幾何形状モデルの読み込みや描画には、過去に作成したプログラム（obj.h, obj.cpp）を使用する。ここでは、obj.h のソースコードのみを示し、obj.cpp のソースコードは省略する。obj.h では、幾何形状モデルを表す Obj 構造体、幾何形状モデルの読み込みを行う LoadObj 関数、幾何形状モデルの描画を行う RenderObj 関数、幾何形状モデルの影の描画を行う RenderObjShadow 関数などが定義されている。

メインのサンプルプログラム（keyframe_sample.cpp）から、上記の関数を呼び出して利用する。プログラムの開始時に、initEnvironment 関数の中で、LoadObj 関数を呼び出して、幾何形状モデルの読み込みを行い、グローバル変数として定義された obj 変数に、読み込んだ幾何形状モデルの情報を格納する。画面描画時に呼ばれる DisplayCallback 関数の中で、アニメーション中の物体の位置・向きや、各キーフレームの位置・向きを表すために、RenderObj 関数や RenderObjShadow 関数を呼び出して、読み込んだ幾何形状モデルを描画する。

ソースコード 2: obj.h

```
1  //
2  //  コンピュータアニメーション特論
3  //  幾何形状モデル（Obj形式）の読み込み＆描画のサンプルプログラム
4  //
5
6  #ifndef  _OBJ_H_
7  #define  _OBJ_H_
8
9
10 // ベクトルデータ
11 struct  Vector
12 {
13    float   x, y, z;
14 };
15
16
17 // カラーデータ
18 struct  Color
19 {
20    float  r, g, b;
21 };
22
23
24 //
25 //  幾何形状モデルの表現例（本プログラムでは使用しない）
26 //
27 struct  SampleGeometry
28 {
29    int        num_vertices;  // 頂点数
30    Vector *   vertices;      // 頂点座標配列 [num_vertices]
31    Vector *   normals;       // 法線ベクトル配列 [num_vertices]
32    Color *    colors;        // カラー配列 [num_vertices]
33
34    int        num_triangles; // 三角面数
35    int *      triangles;     // 三角面の頂点番号配列 [num_triangles*3]
36 };
37
38
39 //
40 //  幾何形状モデルの素材情報（Mtl形式）
41 //
42 struct  Mtl
43 {
44    char *     name;          // マテリアル名
45
46    Color      kd;            // 拡散反射光（とりあえず拡散反射光を glColor3f() で使用する）
47
48    char *     texture_name;  // テクスチャ画像のファイル名
49 };
50
51
52 //
53 //  幾何形状モデル（Obj形式）
54 //
55 struct  Obj
56 {
57    int        num_vertices;  // 頂点数
58    Vector *   vertices;      // 頂点座標配列 [num_vertices]
59
60    int        num_normals;
61    Vector *   normals;       // 法線ベクトル配列 [num_normals]
```

```
62
63      int         num_tex_coords;
64      Vector *    tex_coords;     // テクスチャ座標配列 [num_tex_coords]
65
66      int         num_triangles; // 三角面数
67      int *       tri_v_no;       // 三角面の各頂点の頂点座標番号配列 [num_triangles*3]
68      int *       tri_vn_no;      // 三角面の各頂点の法線ベクトル番号配列 [num_triangles*3]
69      int *       tri_vt_no;      // 三角面の各頂点のテクスチャ座標番号配列 [num_triangles
            *3]
70      Mtl **      tri_material;  // 三角面の素材 [num_triangles]
71
72      int         num_materials; // マテリアル数
73      Mtl **      materials;      // マテリアルの配列 [num_materials]
74  };
75
76
77
78  // Objファイルの読み込み
79  Obj *  LoadObj( const char * filename );
80
81  // Mtlファイルの読み込み
82  void  LoadMtl( const char * filename, Obj * obj );
83
84  // 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）
85  void  ScaleObj( Obj * obj, float max_size, float * size_x = NULL, float * size_y = NULL
        , float * size_z = NULL );
86
87  // 幾何形状モデル（Obj形状）の描画
88  void  RenderObj( Obj * obj );
89
90  // 幾何形状モデル（Obj形状）の描画（固定色で描画）
91  void  RenderObjUnicolor( const Obj * obj, float color_r, float color_g, float color_b,
        float color_a );
92
93  // 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）
94  void  RenderObjShadow( const Obj * obj, const float obj_matrix[ 16 ], float light_dir_x
        , float light_dir_y, float light_dir_z, float color_r, float color_g, float color_b
        , float color_a );
95
96  // 幾何形状モデルを頂点配列を使って描画可能なモデルに変換
97  void  ConvertObjForVertexArrays( Obj * obj );
98
99  // 幾何形状モデルを頂点配列を使って描画
100 void  RenderObjWithVertexArrays( Obj * obj );
101
102 // Objファイルの書き出し
103 int  SaveObj( const Obj * obj, const char * filename );
104
105
106
107 #endif // _OBJ_H_
```

1.2 オブジェクト配置操作

本プログラムでは、マウス操作により、各キーフレームの位置や向きを変更できるユーザインタフェースを提供する。本ユーザインターフェースの実現のために、複数のオブジェクトの位置や向きをマウス操作により変更する機能を実現する ObjectLeyout クラス（ObjectLeyout.h, ObjectLeyout.cpp）が、あらかじめ作成されている。ここでは、ObjectLeyout.h のソースコードのみを示し、ObjectLeyout.cpp のソースコードは省略する。

ObjectLeyout クラスには、マウス操作のイベントを処理するメンバ関数（マウス関連のコールバック関数から呼ばれる）、オブジェクトの操作情報の描画を行う Render 関数、オブジェクトの個数や位置・向きを取得する

GetNumObjects 関数・GetPoition 関数・GetOrientation 関数・GetFrame 関数などが定義されている。

メインのサンプルプログラム（keyframe_sample.cpp）から、上記の関数を呼び出して利用する。マウスクリック時に呼ばれる MouseClickCallback 関数から、ObjectLayout オブジェクトの OnMouseDown メンバ関数や OnMouseUp メンバ関数を呼び出す。マウス移動時・ドラッグ時に呼ばれる MouseMotionCallback 関数や MouseDragCallback 関数から、ObjectLayout オブジェクトの OnMoveMouse メンバ関数や Update メンバ関数を呼び出す。また、157~178 行の、オブジェクト配置にもとづいて全キーフレーム情報を更新する UpdateKeyframes 関数で、ObjectLayout オブジェクトから全オブジェクトの位置・向きを取得して、全キーフレームの情報を設定する（1.7 節参照）。

ソースコード 3: ObjectLeyout.h

```
1  //
2  //  コンピュータアニメーション特論
3  //  複数オブジェクトの位置・向きをマウスで操作するためのクラス
4  //
5
6
7  #ifndef _OBJECT_LAYOUT_H_
8  #define _OBJECT_LAYOUT_H_
9
10
11 //#include <vecmath.h>
12 #include <Vector3.h>
13 #include <Point3.h>
14 #include <Point2.h>
15 #include <Matrix3.h>
16 #include <Matrix4.h>
17 #include <Color3.h>
18 #include <Color4.h>
19
20 #include <vector>
21 using namespace  std;
22
23
24 //
25 //  複数オブジェクトの位置・向きをマウスで操作するためのクラス
26 //
27 class  ObjectLayout
28 {
29   protected:
30     /*  内部用構造体・列挙型の定義  */
31
32     // オブジェクト情報を表す構造体
33     struct  Object
34     {
35        Point3f   pos;  // 位置
36        Matrix3f  ori;  // 向き
37        Vector3f  size; // サイズ（描画用）
38
39        float     axis_length; // 軸の長さ
40        Point2f   screen_pos;  // 画面上の座標
41        Matrix4f  frame; // 位置・向き（キャッシュ用）
42     };
43
44     // 操作を表す列挙型
45     enum  OperationEnum
46     {
47        OP_W_TRANSLATION,  // ワールド座標系での移動
48        OP_M_TRANSLATION,
49        OP_W_ROTATION,
50        OP_M_ROTATION      // モデル座標系での回転
51     };
```

```
52
53      // 操作のためのハンドル（各軸・平面）を表す列挙型
54      enum  HandleEnum
55      {
56          H_NONE = -1,
57          H_MODE_CHANGE,
58
59          H_M_X_AXIS,
60          H_M_Y_AXIS,
61          H_M_Z_AXIS,
62          H_M_XY_PLANE,
63          H_M_YZ_PLANE,
64          H_M_ZX_PLANE,
65
66          H_W_X_AXIS,
67          H_W_Y_AXIS,
68          H_W_Z_AXIS,
69          H_W_XY_PLANE,
70          H_W_YZ_PLANE,
71          H_W_ZX_PLANE,
72
73          NUM_HANDLE
74      };
75
76      // 操作に対応するマウス座標軸を表す列挙型
77      enum  MouseControlEnum
78      {
79          NO_MOUSE_CONTROL,
80          MOUSE_PLUS_X,
81          MOUSE_MINUS_X,
82          MOUSE_PLUS_Y,
83          MOUSE_MINUS_Y
84      };
85
86      // 各種描画オプションを表す構造体
87      struct  RenderOption
88      {
89          Color3f    box_color;
90          Color3f    selected_box_color;
91
92          float      box_width;
93          float      axis_width;
94          float      selected_axis_width;
95
96          Color3f    x_axis_color;
97          Color3f    y_axis_color;
98          Color3f    z_axis_color;
99
100         bool       change_axis_length;
101         bool       enable_xray_mode;
102         bool       enable_smooth;
103     };
104
105   protected:
106     /*  動作設定情報  */
107
108     // 描画オプション
109     RenderOption  render_option;
110
111     // デフォルトのオブジェクト情報
112     Object  default_object;
113
114   protected:
115     /*  オブジェクト情報  */
```

```
116
117     // 全オブジェクトの情報
118     vector< Object >  objects;
119
120   protected:
121     /*  操作情報  */
122
123     // 現在選択中のオブジェクト番号
124     int  curr_object;
125
126     // 現在の操作モード
127     OperationEnum  curr_operation;
128
129     // 現在有効なハンドル
130     HandleEnum  active_handle;
131
132     // 操作に対応するマウス座標軸
133     MouseControlEnum  mouse_control;
134
135     // ハンドル操作中かどうかのフラグ
136     bool  on_control;
137
138     // 操作ハンドルの画面上の座標
139     Point2f  handle_pos[ NUM_HANDLE ];
140
141     // 前回のマウス座標
142     int  last_mouse_x;
143     int  last_mouse_y;
144
145
146   public:
147     /*  初期化・終了処理  */
148
149     // コンストラクタ
150     ObjectLayout();
151
152   public:
153     /*  アクセサ  */
154
155     size_t  GetNumObjects() const { return  objects.size(); }
156     int  GetCurrentObject() const { return  curr_object; }
157     Point3f &  GetPosition( int no ) { return  objects[ no ].pos; }
158     Matrix3f &  GetOrientation( int no ) { return  objects[ no ].ori; }
159     Matrix4f &  GetFrame( int no );
160     Matrix4f &  GetTransposedFrame( int no );
161     const Point3f &  GetPosition( int no ) const { return  objects[ no ].pos; }
162     const Matrix3f &  GetOrientation( int no ) const { return  objects[ no ].ori; }
163     RenderOption &  GetRenderOption() { return  render_option; }
164     const char *  GetOperationMode();
165     const float *  GetPositionArray( int no ) { return  & objects[ no ].pos.x; }
166     const float *  GetOrientationArray( int no ) { return  & objects[ no ].ori.m00; }
167     const float *  GetFrameArray( int no ) { GetFrame( no );  return  & objects[ no ].
          frame.m00; }
168     const float *  GetTransposedFrameArray( int no ) { GetTransposedFrame( no );  return
            & objects[ no ].frame.m00; }
169
170   public:
171     /*  オブジェクトの操作  */
172
173     // オブジェクトの追加
174     int  AddObject();
175
176     // オブジェクトの削除
177     int  DeleteObject();
```

```
178     int   DeleteAllObjects();
179
180     // オブジェクトの情報設定
181     void  SetObjectPos( int no, const Point3f & p );
182     void  SetObjectOri( int no, const Matrix3f & o );
183     void  SetObjectFrame( int no, const Matrix4f & f );
184     void  SetObjectSize( int no, const Vector3f & s );
185     void  SetObjectSize( int no, float s );
186
187   public:
188     /*  イベントハンドラ  */
189
190     // オブジェクト位置・視点の変更を通知
191     void  Update();
192
193     // マウス移動時の処理
194     void  OnMoveMouse( int mx, int my );
195
196     // マウスのボタンが押された時の処理
197     void  OnMouseDown( int mx, int my );
198
199     // マウスのボタンが離された時の処理
200     void  OnMouseUp( int mx, int my );
201
202     // オブジェクトの操作情報の描画
203     void  Render();
204
205
206   protected:
207     /*  内部メソッド  */
208
209     // 全オブジェクトをスクリーン座標系に投影
210     void  ProjectObjects();
211
212     // マウス位置に近いオブジェクトを探索
213     int  FindObject( int mx, int my, int threshold );
214
215     // マウス位置に近いハンドルを探索
216     HandleEnum  FindHandle( int mx, int my, OperationEnum op, int threshold );
217 };
218
219
220
221 #endif // _OBJECT_LAYOUT_H_
```

1.3 vecmath 補助関数

本サンプルプログラムでは、行列・ベクトルなどを扱うために vecmath C++ ライブラリを使用する。OpenGL の関数は、行列やベクトルなどの入力を関数の引数として渡す場合、複数の数値型や数値型の配列によって渡す形になっており、vecmath の行列やベクトルを渡すときには、変換が必要になる。毎回このような変換を行わなくとも良いように、OpenGL の関数と同名の関数で、vecmath の行列やベクトルを引数として受け取り、内部で変換を行って OpenGL の関数を呼び出す関数を定義して、利用できるようにする。メインのサンプルプログラム（keyframe_sample.cpp）から、これらの関数を呼び出して利用する。

ソースコード 4: vecmath_gl.h

```
1 //
2 //  コンピュータアニメーション特論
3 //  vecmathオブジェクトを引数としてOpenGL関数を呼び出すための補助関数
4 //
```

```
5
6
7  #ifndef  _VECMATH_GL_H_
8  #define  _VECMATH_GL_H_
9
10
11 #include <Vector3.h>
12 #include <Point3.h>
13 #include <Point2.h>
14 #include <Color3.h>
15 #include <Matrix3.h>
16 #include <Matrix4.h>
17
18
19 inline void  glVertex3f( const Tuple3f & v )
20 {
21    glVertex3f( v.x, v.y, v.z );
22 }
23
24 inline void  glTexCoord2f( const Tuple2f & t )
25 {
26    glTexCoord2f( t.x, t.y );
27 }
28
29 inline void  glNormal3f( const Vector3f & n )
30 {
31    glNormal3f( n.x, n.y, n.z );
32 }
33
34 inline void  glColor3f( const Color3f & c )
35 {
36    glColor3f( c.x, c.y, c.z );
37 }
38
39 inline void  glColor4f( const Color4f & c )
40 {
41    glColor4f( c.x, c.y, c.z, c.w );
42 }
43
44
45 inline void  glTranslatef( const Tuple3f & t )
46 {
47    glTranslatef( t.x, t.y, t.z );
48 }
49
50 inline void  glMultMatrixf( const Matrix4f & m )
51 {
52    Matrix4f  mat;
53    mat.transpose( m );
54    glMultMatrixf( &mat.m00 );
55 }
56
57 inline void  glMultMatrixf( const Matrix3f & m )
58 {
59    Matrix4f  mat;
60    mat.set( m );
61    mat.transpose();
62    glMultMatrixf( &mat.m00 );
63 }
64
65
66 #endif // _VECMATH_GL_H_
```

1.4 位置・向きの補間方法の定義

サンプルプログラム（keyframe_sample.cpp）の 82~99 行で、位置・向き補間方法を表す列挙型（PositionInterpolationEnum, OrientationInterpolationEnum）が定義されている。また、110~111 行で、これらの列挙型を用いて、現在の補間方法を表すグローバル変数が定義されている。

キーボードのキーが押されたときに呼ばれる KeyboardCallback 関数で、キー入力に応じて、これらの変数を変更する。また、物体の位置・向きを更新する UpdateModelMat 関数で、これらの変数にもとづく補間方法を使って、位置・向きを計算する。

1.5 キーフレーム情報の定義

サンプルプログラム（keyframe_sample.cpp）の 119~124 行で、キーフレームの時刻・位置・向きの情報を表す、Keyframe 構造体が定義されている。位置・向きの定義には、vecmath のクラスを使用しており、位置は 3 次元座標を表す Point3f 型、向きは 3×3 行列を表す Matrix3f 型で表す。また、127 行で、この Keyframe 構造体の可変長配列（vector テンプレートクラス）として、全キーフレーム情報を表す keyframes グローバル変数が定義されている。

1.6 キーフレーム情報の設定

サンプルプログラム（keyframe_sample.cpp）の 157~178 行の、オブジェクト配置にもとづいて全キーフレーム情報を更新する UpdateKeyframes 関数で、ObjectLayout オブジェクトから全オブジェクトの位置・向きを取得して、全キーフレームの情報を格納する keyframe グローバル変数（1.5 節参照）に設定する。ObjectLayout に num_keyframes 個のオブジェクトの位置・向きが設定されているときに、num_keyframes 個キーフレームを作成して、各オブジェクトの位置・向きを設定する。キーフレームの時刻については、本プログラムでは編集のためのインターフェースを提供していないため、仮に、i 番目のキーフレームを i 秒として設定する。

1.7 キーフレームアニメーションの位置・向きの計算

サンプルプログラム（keyframe_sample.cpp）の 233~445 行の UpdateModelMat 関数で、全キーフレームと現在時刻の情報にもとづいて、物体の位置・向きの計算を行う。本関数が、キーフレームアニメーションを実現するメインの関数となる。本関数の引数として、キーフレーム数（int num_keyframes）、キーフレーム配列（const Keyframe * keyframes）、時刻（float time）の入力と、位置・向きを表す変換行列を出力するための配列のアドレス（float mat[16]）を受け取る。

本関数では、1.4 節で説明した、位置補間方法（pos_method）と向き補間方法（ori_method）の設定にもとづいて、指定された方法で位置と向きの補間を行う。本来であれば、補間方法ごとに関数を分けるのが望ましいが、レポート課題で作成する関数が分かりやすいように、一つの関数に複数の処理をまとめている。

ソースコード 5 に、本関数の定義と、位置･向きの補間の前後の処理を抜き出している。位置･向きの補間の前段階の処理として、211 行 ~242 行で、現在の時刻に対応する、キーフレーム区間の番号（int seg_no）（0~num_keyframes-1）と、その区間内での正規化時間（t）（0.0~1.0）を計算している。また、246 行 ~247 行で、補間結果の位置と向きを格納するローカル変数（Vector3f p, Matrix3f o）を定義している。位置・向きの補間の後段階の処理として、413 行 ~416 行で、変数 p, o に格納された補間結果の位置・向きにもとづいて、4×4 変換行列を作成して、引数として渡された出力を格納するための配列（mat）に代入している。このとき、vecmath と OpenGL では、行列の要素を格納するときの列・行の順序が異なるため、転置を行う transpose 関数を呼び出して、列・行の反転を行っている。

ソースコード 5: キーフレームアニメーションの位置・向きの計算（前処理・後処理）

```
//
//  物体の位置・向きを更新
```

```
// （キーフレーム数、キーフレーム配列、時刻を入力として、その時刻における位置・向きを表す変換行列を出力）
//
void  UpdateModelMat( int num_keyframes, const Keyframe * keyframes, float time, float mat[ 16 ] )
{
   if ( num_keyframes <= 1 )
      return;

   // 指定時刻に対応する区間の番号と区間内での正規化時間（0.0～1.0）
   int  seg_no = -1;
   float  t = 0.0f;

   // 指定時刻に対応する区間の番号を取得
   for ( int i=0; i<num_keyframes-1; i++ )
   {
      // 指定時刻が i番目の区間に対応するかを判定
      if ( ( time >= keyframes[ i ].time ) && ( time <= keyframes[ i+1 ].time ) )
      {
         seg_no = i;

         // 区間内での正規化時間を計算
         t = ( time - keyframes[ i ].time ) / ( keyframes[ i+1 ].time  - keyframes[ i ].time );

         break;
      }
   }
   if ( seg_no == -1 )
   {
      // 最初のキーフレームより前の時刻が指定されたら、最初の区間の開始時刻を使用
      if ( time < keyframes[ 0 ].time )
      {
         seg_no = 0;
         t = 0.0f;
      }
      // 最後のキーフレームより後の時刻が指定されたら、最後の区間の終了時刻を使用
      else
      {
         seg_no = num_keyframes - 2;
         t = 1.0f;
      }
   }

   // 指定時刻におけるオブジェクトの位置・向き
   Vector3f  p;
   Matrix3f  o;

   // 省略

   // オブジェクトの位置・向きを表す変換行列を配列にコピー
   Matrix4f  f;
   f.set( o, p, 1.0f );
   f.transpose();
   memcpy( mat, &f.m00, sizeof( float ) * 16 );
}
```

1.8 線形補間による位置補間

サンプルプログラム（keyframe_sample.cpp）の UpdateModelMat 関数内部の 250~261 行で、線形補間による位置補間の処理を行っている。前段階の処理で求めた、現在時刻に対応するキーフレーム区間の番号（int seg_no）

（0～num_keyframes-1）と、その区間内での正規化時間（t）（0.0～1.0）にもとづいて、現在時刻に対応する位置を計算する。

線形補間による位置（$\mathbf{p}$）は、区間の開始点での位置（$\mathbf{p}_0$）、区間の終了点での位置（$\mathbf{p}_1$）、正規化時間（t）にもとづいて、次の式で計算できる。

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

vecmath の関数を呼び出すことで、この式の通りの計算を行う。なお、vecmath では、演算子オーバーロードにより、実際の数式に近い形で + や * などの演算子を使った記述が行えるが、演算子オーバーロードを用いると一時的な変数が生成されて若干効率が悪くなるため、演算子オーバーロードは使わずに、scaleAdd メンバ関数の呼び出しにより、同等の計算を実現している。

ソースコード 6: 線形補間による位置補間）

```
//
//  物体の位置・向きを更新
//  （キーフレーム数、キーフレーム配列、時刻を入力として、その時刻における位置・向きを表
    す変換行列を出力）
//
void  UpdateModelMat( int num_keyframes, const Keyframe * keyframes, float time, float
    mat[ 16 ] )
{
   // 省略

   // 位置を線形補間により計算
   if ( pos_method == PI_LINEAR )
   {
      // 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;

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

      // 両端点を線形に補間（下記の書き方でも可）
//    p = t * ( p1 - p0 ) + p0;
   }

   // 省略
}
```

1.9 オイラー角の線形補間による向き補間

サンプルプログラム（keyframe_sample.cpp）の UpdateModelMat 関数内部の 265～394 行で、オイラー角の線形補間による向きの計算を行っている。前段階の処理で求めた、現在時刻に対応するキーフレーム区間の番号（int seg_no）（0～num_keyframes-1）と、その区間内での正規化時間（t）（0.0～1.0）にもとづいて、現在時刻に対応する向きを計算する。

オイラー角の線形補間による向き補間では、向きの表現方法を 3×3 行列による表現とオイラー角による表現の間で変換を行い、オイラー角の線形補間を行う。サンプルプログラム（keyframe_sample.cpp）の 184～198 行で、3×3 行列からオイラー角への変換を行う ConvMatToEuler 関数が作成されているので、本関数を呼び出して変換を行う。オイラー角表現は、軸の順番や各軸の回転角度の範囲によって、複数の表現方法があり、3×3 行列からオイラー角への変換の計算方法も異なる。本関数では、よく用いられる、方位角（ヨー角）→仰角（ピッチ角）→旋回角（ロール角）の順番のオイラー角への変換を行う。

UpdateModelMat 関数内の 372～375 行で、区間の開始点・終了点の向きを取得し、3×3 行列からオイラー角への変換を行っている。その後、 378～385 行で、3 つの角度について、式 1 と同様の線形補間により、計算を行う。このとき、方位角については、0～360 度の間で連続して補間を行うように、2 つの角度が 180 度よりも大きい場

合は、180 度以内になるように変換してから、線形補間を行う。その後、オイラー角から 3×3 行列の変換については、回転行列の積により計算できるため、UpdateModelMat 関数内の 488~931 行で計算を行っている。

ソースコード 7: オイラー角の線形補間による向き補間

```
//
//  物体の位置・向きを更新
//  (キーフレーム数、キーフレーム配列、時刻を入力として、その時刻における位置・向きを表す変換行列を出力)
//
void  UpdateModelMat( int num_keyframes, const Keyframe * keyframes, float time, float mat[ 16 ] )
{
   // 省略

   // 向きをオイラー角の線形補間により計算
   else if ( ori_method == OI_EULER )
   {
      // 区間の両端点の向きを取得
      const Matrix3f &  o0 = keyframes[ seg_no ].ori;
      const Matrix3f &  o1 = keyframes[ seg_no + 1 ].ori;

      // オイラー角に変換
      float  y0, p0, r0;
      float  y1, p1, r1;
      ConvMatToEuler( o0, y0, p0, r0 );
      ConvMatToEuler( o1, y1, p1, r1 );

      // 各回転角度を線形補間
      float  y, p, r;
      if ( y0 < y1 - M_PI )
         y0 += 2.0f * M_PI;
      else if ( y0 > y1 + M_PI )
         y0 -= 2.0f * M_PI;
      y = ( y1 - y0 ) * t + y0;
      p = ( p1 - p0 ) * t + p0;
      r = ( r1 - r0 ) * t + r0;

      // 行列に変換
      Matrix3f  rot;
      o.rotY( y );
      rot.rotX( p );
      o.mul( o, rot );
      rot.rotZ( r );
      o.mul( o, rot );
   }

   // 省略
}
```

2　レポート課題

レポート課題として、残りの 3 つの位置補間方法（Hermite 曲線、Bézier 曲線、B-Spline 曲線）と 1 つの向き補間方法（四元数の球面線形補間）を実現する、計 4 つの処理を作成する。

2.1 Hermite 曲線による位置補間

Hermite 曲線による位置（$\mathbf{p}$）は、区間の開始点での位置（$\mathbf{p}_0$）と速度（$\mathbf{v}_0$）、区間の終了点での位置（$\mathbf{p}_1$）と速度（$\mathbf{v}_1$）、正規化時間（t）にもとづいて、次の式で計算できる。

$$\mathbf{p} = H_0(t)\mathbf{p}_0 + H_1(t)\mathbf{p}_1 + h_0(t)\mathbf{v}_0 + h_1(t)\mathbf{v}_1 \tag{2}$$

$$H_0(t) = 2t^3 - 3t^2 + 1 \tag{3}$$

$$H_1(t) = -2t^3 + 3t^2 \tag{4}$$

$$h_0(t) = t^3 - 2t^2 + t \tag{5}$$

$$h_1(t) = t^3 - t^2 \tag{6}$$

Hermite 曲線では、区間の両端点の位置に加えて、速度の情報が必要になる。

以下の説明文やプログラムの空欄に入るコードや語句を考えて、プログラムを作成せよ。

Hermite 関数の式にもとづいて、区間の両端点の位置・速度 p0, p1, v0, v1 と、区間内のローカル時刻 t から、位置 p を計算する。

まずは、Hermite 関数の係数 a, b, c, d を計算する（ 空欄 A ～ 空欄 D ）。

その後、Vector3 クラスのメンバ関数や演算子を使用して、Hermite 関数を計算する。以下のプログラムでは、メンバ関数を使用して、Hermite 関数の計算を行っている。（ 空欄 E ）。

ソースコード 8: Hermite 曲線による位置補間

```
void  UpdateModelMat( float time, float mat[ 16 ] )
{
   // 省略

   else if ( pos_method == PI_HERMIT )
   {
      // 省略

      // Hermite関数の値を計算

      // ※レポート課題（ここに自分が作成したプログラムを記述する）

      // Hermite関数の係数を計算
      float  a, b, c, d;
      a = 空欄A;
      b = 空欄B;
      c = 空欄C;
      d = 空欄D;

      // Hermite関数の計算
      空欄E
   }

   // 省略
}
```

2.2 Bézier 曲線による位置補間

Bézier 曲線による位置（$\mathbf{p}$）は、4 つのキーフレームで表される区間の各キーフレームの位置（$\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$）、正規化時間（t）にもとづいて、次の式で計算できる。

$$\mathbf{p} = X_0(t)\mathbf{p}_0 + X_1(t)\mathbf{p}_1 + X_2(t)\mathbf{p}_2 + X_3(t)\mathbf{p}_3 \tag{7}$$

$$X_0(t) = (t-1)^3 \tag{8}$$

$$X_1(t) = 3(t-1)^2 t \tag{9}$$

$$X_2(t) = 3(t-1)t^2 \tag{10}$$

$$X_3(t) = t^3 \tag{11}$$

Bézier 曲線は、両端の 2 つのキーフレームの点は通るが、通常、間の 2 つのキーフレームの点は通らないことに注意する。また、前後の区間をなめらかに接続するためには、区間の端点と前後のキーフレームの点が直線に並ぶ必要がある。

以下の説明文やプログラムの空欄に入るコードや語句を考えて、プログラムを作成せよ。

同様に、Bézier 補間の式にもとづいて、区間の 4 点の位置 p0, p1, p2, p3 と、区間内のローカル時刻 s から、位置 p を計算する。

Bézier 関数の係数 a0 ∼ a3 を計算し（ 空欄 A ∼ 空欄 D ）、その後、Bézier 関数の計算を行う。（ 空欄 E ）。

ソースコード 9: Bézier 曲線による位置補間

```
void  UpdateModelMat( float time, float mat[ 16 ] )
{
   // 省略

   else if ( pos_method == PI_BEZIER )
   {
      // 省略

      // Bezier関数の値を計算

      // ※レポート課題（ここに自分が作成したプログラムを記述する）

      // Bezier 関数の係数を計算
      float  a0, a1, a2, a3;
      a0 = 空欄A;
      a1 = 空欄B;
      a2 = 空欄C;
      a3 = 空欄D;

      // Bezier 関数の値を計算
      空欄E
   }

   // 省略
}
```

2.3 B-Spline 曲線による位置補間

B-Spline 曲線による位置（$\mathbf{p}$）は、区間の 1 つ前のキーフレームの位置（$\mathbf{p}_0$）、区間の開始点での位置（$\mathbf{p}_1$）、区間の終了点での位置（$\mathbf{p}_2$）、区間の 1 つ後のキーフレームの位置（$\mathbf{p}_3$）、正規化時間（t）にもとづいて、次の式で計算できる。

$$\mathbf{p} = X_0(t)\mathbf{p}_0 + X_1(t)\mathbf{p}_1 + X_2(t)\mathbf{p}_2 + X_3(t)\mathbf{p}_3 \tag{12}$$

$$X_0(t) = -\frac{1}{6}t^3 + \frac{1}{2}t^2 - \frac{1}{2}t + \frac{1}{6} \tag{13}$$

$$X_1(t) = \frac{1}{2}t^3 - t^2 + \frac{2}{3} \tag{14}$$

$$X_2(t) = -\frac{1}{2}t^3 + \frac{1}{2}t^2 + \frac{1}{2}t + \frac{1}{6} \tag{15}$$

$$X_3(t) = \frac{1}{6}t^3 \tag{16}$$

B-Spline 曲線は、通常、4 つの全てのキーフレームの点は通らないことに注意する。

以下の説明文やプログラムの空欄に入るコードや語句を考えて、プログラムを作成せよ。

同様に、B-Spline 補間の式にもとづいて、区間の 4 点の位置 p0, p1, p2, p3 と、区間内のローカル時刻 t から、位置 p を計算する。

B-Spline 関数の係数 a0 ～ a3 を計算し（ 空欄 A ～ 空欄 D ）、その後、B-Spline 関数の計算を行う。（ 空欄 E ）。

ソースコード 10: B-Spline 曲線による位置補間

```
void  UpdateModelMat( float time, float mat[ 16 ] )
{
   // 省略

   else if ( pos_method == PI_BSPLINE )
   {
      // 省略

      // B-Spline関数の値を計算

      // ※レポート課題（ここに自分が作成したプログラムを記述する）

      // B-Spline 関数の係数を計算
      float  a0, a1, a2, a3;
      a0 = 空欄A;
      a1 = 空欄B;
      a2 = 空欄C;
      a3 = 空欄D;

      // B-Spline 関数の値を計算
      空欄E
   }

   // 省略
}
```

2.4 四元数の球面線形補間による向き補間

四元数の球面線形補間による向きはは、区間の開始点での向き（$\mathbf{o}_0$）、区間の終了点での向き（$\mathbf{o}_1$）、正規化時間（t）から計算する。3×3 行列で表された向き（$\mathbf{o}_0$,$\mathbf{o}_1$）を四元数（4 次元ベクトル）で表された向き（$\mathbf{q}_0$,$\mathbf{q}_1$）に変換してから、球面線形補間を計算する。

$$\mathbf{q} = \frac{sin(1-t)\theta}{sin\theta}\mathbf{q}_0 + \frac{sint\theta}{sin\theta}\mathbf{q}_1 \tag{17}$$

$$\theta = cos^{-1}\mathbf{q}_0\dot{\mathbf{q}}_1 \tag{18}$$

vecmath を使用する場合は、球面線形補間の計算は vecmath のメンバ関数を呼び出すことで行えるため、これらの式を計算するプログラムを作成する必要はない。

以下の説明文やプログラムの空欄に入るコードや語句を考えて、プログラムを作成せよ。

区間の端点の向き o0, 01 と、ローカル時間 t から、補間後の向き o を計算する。
区間の端点の向きは 3 × 3 行列の形式で渡されるため、四元数に変換する（ 空欄 A ）。
球面線形補間を適用する前に、2 つの四元数の間の角度が 90 度以上あれば、共役の四元数を使用する（ 空欄 B ）。
その後、四元数を使って球面線形補間を行う（ 空欄 C ）。
最後に、計算後の四元数を、 3 × 3 行列に変換して、後の処理に渡す（ 空欄 D ）。

ソースコード 11: 四元数と球面線形補間による向き補間

```
void  UpdateModelMat( float time, float mat[ 16 ] )
{
   // 省略

   else if ( ori_method == OI_QUAT )
   {
      // 省略

      // 四元数を使った球面線形補間を計算

      // ※レポート課題（ここに自分が作成したプログラムを記述する）

      Quat4f  q, q0, q1;
      空欄A
      if ( q0.x * q1.x + q0.y * q1.y + q0.z * q1.z + q0.w * q1.w < 0 )
         空欄B
      空欄C
      空欄D
   }

   // 省略
}
```