

# Einführung in C++ – Übung 7

## Testatgruppe A (Isaak)

Rasmus Diederichsen

25. November 2014

### Aufgabe 7.1 Implementierung von Quaternionen, Vektoren und Matrizen

Listing 1: Matrix.hpp

```
1  /* Copyright (C) 2011 Uni Osnabrück
2   * This file is part of the LAS VEGAS Reconstruction Toolkit,
3   *
4   * LAS VEGAS is free software; you can redistribute it and/or
5   *   modify
6   * it under the terms of the GNU General Public License as
7   *   published by
8   * the Free Software Foundation; either version 2 of the License,
9   *   or
10  * (at your option) any later version.
11  *
12  * LAS VEGAS is distributed in the hope that it will be useful,
13  * but WITHOUT ANY WARRANTY; without even the implied warranty of
14  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
15  * GNU General Public License for more details.
16  *
17  * You should have received a copy of the GNU General Public
18  *   License
19  * along with this program; if not, write to the Free Software
20  * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
21  *   02111-1307, USA
22  */
23
24 /*
25  * Matrix.hpp
26  *
27  * @date 26.08.2008
28  * @author Thomas Wiemann (twiemann@uos.de)
29  */
30
31 #ifndef MATRIX_H_
32 #define MATRIX_H_
```

```

29
30 #include <iostream>
31 #include <fstream>
32 #include <iomanip>
33
34 #include "Vertex.hpp"
35
36 #define _USE_MATH_DEFINES
37 #include <cmath>
38
39 #ifndef M_PI
40 #define M_PI 3.141592654
41 #endif
42
43 using namespace std;
44
45 namespace cpp2014 {
46
47     /**
48      * @brief A 4x4 matrix class definition for use with the
49      * provided vertex types.
50      */
51     class Matrix {
52     public:
53
54         /**
55          * @brief Default constructor. Initializes a identity
56          * matrix.
57          */
58         Matrix();
59
60         /**
61          * @brief Initializes a matrix wit the given data
62          * array. Ensure that the array has exactly 16
63          * fields.
64          */
65         Matrix(float* matrix);
66
67         /**
68          * @brief Copy constructor.
69          */
70         Matrix(const Matrix& other);
71
72         /**
73          * @brief Constructs a matrix from given axis and
74          * angle. Tries to avoid a gimbal lock.
75          */
76         Matrix(Vertex axis, float angle);
77
78         Matrix(const Vertex &position, const Vertex &angles);
79
80         /**
81          * Destructor

```

```

81      */
82      ~Matrix();
83
84
85      /**
86       * @brief      Matrix-Matrix multiplication.
87       * @param m The multiplicand.
88       * @return This matrix.
89       */
90      Matrix operator*(const Matrix& m) const;
91
92
93      /**
94       * @brief      Matrix addition operator. Returns a new
95       *             matrix
96       * @param m Matrix to be assigned.
97       * @return Thi matrix.
98       */
99      Matrix& operator+=(const Matrix& m);
100
101      /**
102       * @brief      Matrix addition operator
103       * @param m Addend.
104       * @return A new matrix which is the sum of this one and
105       *             the addend.
106       */
107      Matrix operator+(const Matrix& m) const;
108
109      /**
110       * @brief Elementwisely subtract a matrix from this one.
111       * @param m Subtrahend
112       * @return The difference of this mtrix and the subtrahend
113       *             .
114       */
115      Matrix operator-(const Matrix& m) const;
116
117      /**
118       * @brief Negate all entries in this matrix.
119       * @return A new <tt>Matrix</tt> with entries negated.
120       */
121      Matrix operator-() const;
122
123      /**
124       * @brief      Matrix-Matrix multiplication (array based).
125       *             Mainly
126       *             defined for compatibility with
127       *             other math libs.
128       *             ensure that the used array has at
129       *             least 16 elements
130       *             to avoid memory access violations.
131       */
132
133      /// TODO: DEFINE OPERATOR HERE!
134
135      /**
136       * THIS MAKES NO SENSE
137       * @brief      Multiplication of Matrix and Vertex types

```

```

132     */
133
134     /* Matrix& operator*(const Vertex& v) const; */
135
136     /**
137      * @brief      Sets the given index of the Matrix's data
138      *              field
139      *              to the provided value.
140      * @param      i          Field index of the matrix
141      * @param      value      new value
142      */
143     void set(int i, float value);
144
145     /**
146      * @brief      Transposes the current matrix
147      */
148     void transpose();
149
150     /**
151      * @brief      Computes an Euler representation (x, y, z)
152      *              plus three
153      *              rotation values in rad. Rotations
154      *              are with respect to
155      *              the x, y, z axes.
156      */
157     void toPositionAngle(float pose[6]);
158
159     /**
160      * @brief      Matrix scaling with self assignment.
161      */
162     Matrix& operator*=(const float f);
163
164     /**
165      * @brief      Matrix-Matrix multiplication with self
166      *              assignment.
167      */
168     Matrix& operator*=(const Matrix& other);
169
170     /**
171      * @brief      Matrix-Matrix multiplication (array based).
172      *              See \ref{operator*}.
173      */
174     /// TODO: DEFINE OPERATOR HERE!
175
176     /**
177      * @brief      Returns the internal data array. Unsafe.
178      *              Will probably
179      *              removed in one of the next versions
180      *              .
181      */
182     float* getData();
183
184     /**
185      * @brief      Indexed element (reading) access.

```

```

182     */
183     const float& operator[](const int i) const;
184
185     /**
186     * @brief      Writeable index access
187     */
188     float& operator[](const int i);
189
190     /**
191     * @brief      Returns the matrix's determinant
192     */
193     float det();
194
195     /**
196     * @brief      Inverts the matrix. Success is true if
197                   operation was successful
198     */
199     Matrix inv(bool& success);
200
201     /**
202     * Returns a tring representation of this object.
203     * @return A <tt>string</tt> representation.
204     */
205     std::string to_s();
206
207     private:
208
209     /**
210     * @brief      Returns a sub matrix without row \ref i and
211                   column \ref j.
212     */
213     void submat(float* submat, int i, int j);
214
215     /**
216     * @brief      Calculates the determinant of a 3x3 matrix
217     *
218     * @param      M input 3x3 matrix
219     * @return      determinant of input matrix
220     */
221     float det3( const float *M );
222
223     /**
224     * @brief      Scales the matrix elemnts by the given
225                   factor
226     */
227     void scale(float f);
228
229     /// Internal data array
230     float m[16];
231
232 };
233
234 } // namespace cpp2014
235 #endif /* MATRIX_H_ */

```

Listing 2: Matrix.cpp

```

1  /*
2  * Matrix.hpp
3  *
4  * @date 26.08.2008
5  * @author Thomas Wiemann (twiemann@uos.de)
6  */
7  #include "Matrix.hpp"
8  #include <sstream>
9  #include <stdexcept>
10
11 namespace cpp2014
12 {
13
14     Matrix::Matrix()
15     {
16         for(int i = 0; i < 16; i++) m[i] = 0;
17         m[0] = m[5] = m[10] = m[15] = 1;
18     }
19
20
21     Matrix::Matrix(float* matrix)
22     {
23         for(int i = 0; i < 16; i++) m[i] = matrix[i];
24     }
25
26
27     Matrix::Matrix(const Matrix& other)
28     {
29         for(int i = 0; i < 16; i++) m[i] = other[i];
30     }
31
32     Matrix::Matrix(Vertex axis, float angle)
33     {
34         // Check for gimbal lock
35         if(fabs(angle) < 0.0001)
36         {
37
38             bool invert_z = axis.z < 0;
39
40             //Angle to yz-plane
41             float pitch = atan2(axis.z, axis.x) - M_PI_2;
42             if(pitch < 0.0f) pitch += 2.0f * M_PI;
43
44             if(axis.x == 0.0f && axis.z == 0.0) pitch = 0.0f;
45
46             //Transform axis into yz-plane
47             axis.x = axis.x * cos(pitch) + axis.z * sin(pitch);
48             axis.z = -axis.x * sin(pitch) + axis.z * cos(pitch);
49
50             //Angle to y-Axis
51             float yaw = atan2(axis.y, axis.z);
52             if(yaw < 0) yaw += 2 * M_PI;
53
54             Matrix m1, m2, m3;
55
56             if(invert_z) yaw = -yaw;
57

```

```

58     cout << "YAW:␣" << yaw << "␣PITCH:␣" << pitch << endl;
59
60     if(fabs(yaw) > 0.0001){
61         m2 = Matrix(Vertex(1.0, 0.0, 0.0), yaw);
62         m3 = m3 * m2;
63     }
64
65     if(fabs(pitch) > 0.0001){
66         m1 = Matrix(Vertex(0.0, 1.0, 0.0), pitch);
67         m3 = m3 * m1;
68     }
69
70     for(int i = 0; i < 16; i++) m[i] = m3[i];
71
72     } else {
73         float c = cos(angle);
74         float s = sin(angle);
75         float t = 1.0f - c;
76         float tmp1, tmp2;
77
78         // Normalize axis
79         Vertex a(axis);
80         a.normalize();
81
82         m[ 0] = c + a.x * a.x * t;
83         m[ 5] = c + a.y * a.y * t;
84         m[10] = c + a.z * a.z * t;
85
86         tmp1 = a.x * a.y * t;
87         tmp2 = a.z * s;
88         m[ 4] = tmp1 + tmp2;
89         m[ 1] = tmp1 - tmp2;
90
91         tmp1 = a.x * a.z * t;
92         tmp2 = a.y * s;
93         m[ 8] = tmp1 - tmp2;
94         m[ 2] = tmp1 + tmp2;
95
96         tmp1 = a.y * a.z * t;
97         tmp2 = a.x * s;
98         m[ 9] = tmp1 + tmp2;
99         m[ 6] = tmp1 - tmp2;
100
101         m[ 3] = m[ 7] = m[11] = 0.0;
102         m[12] = m[13] = m[14] = 0.0;
103         m[15] = 1.0;
104     }
105 }
106
107
108 Matrix::Matrix(const Vertex &position, const Vertex &angles)
109 {
110     float sx = sin(angles[0]);
111     float cx = cos(angles[0]);
112     float sy = sin(angles[1]);
113     float cy = cos(angles[1]);
114     float sz = sin(angles[2]);

```

```

115         float cz = cos(angles[2]);
116
117         m[0] = cy*cz;
118         m[1] = sx*sy*cz + cx*sz;
119         m[2] = -cx*sy*cz + sx*sz;
120         m[3] = 0.0;
121         m[4] = -cy*sz;
122         m[5] = -sx*sy*sz + cx*cz;
123         m[6] = cx*sy*sz + sx*cz;
124         m[7] = 0.0;
125         m[8] = sy;
126         m[9] = -sx*cy;
127         m[10] = cx*cy;
128
129         m[11] = 0.0;
130
131         m[12] = position[0];
132         m[13] = position[1];
133         m[14] = position[2];
134         m[15] = 1;
135     }
136
137     Matrix::~Matrix() { }
138
139     /**
140      * @brief      Transposes the current matrix
141      */
142     void Matrix::transpose()
143     {
144         float m_tmp[16];
145         m_tmp[0] = m[0];
146         m_tmp[4] = m[1];
147         m_tmp[8] = m[2];
148         m_tmp[12] = m[3];
149         m_tmp[1] = m[4];
150         m_tmp[5] = m[5];
151         m_tmp[9] = m[6];
152         m_tmp[13] = m[7];
153         m_tmp[2] = m[8];
154         m_tmp[6] = m[9];
155         m_tmp[10] = m[10];
156         m_tmp[14] = m[11];
157         m_tmp[3] = m[12];
158         m_tmp[7] = m[13];
159         m_tmp[11] = m[14];
160         m_tmp[15] = m[15];
161         for(int i = 0; i < 16; i++) m[i] = m_tmp[i];
162     }
163
164     /**
165      * @brief      Computes an Euler representation (x, y, z) plus
166                  three
167                  rotation values in rad. Rotations are with
168                  respect to
169                  the x, y, z axes.
170      */
171     void Matrix::toPostionAngle(float pose[6])

```



```

170 {
171     if(pose != 0){
172         float _trX, _trY;
173         if(m[0] > 0.0) {
174             pose[4] = asin(m[8]);
175         } else {
176             pose[4] = (float)M_PI - asin(m[8]);
177         }
178         // rPosTheta[1] = asin( m[8]); // Calculate Y-axis
            angle
179
180         float C = cos( pose[4] );
181         if ( fabs( C ) > 0.005 ) { // Gimball lock?
182             _trX = m[10] / C; // No, so get X-axis
            angle
183             _trY = -m[9] / C;
184             pose[3] = atan2( _trY, _trX );
185             _trX = m[0] / C; // Get Z-axis angle
186             _trY = -m[4] / C;
187             pose[5] = atan2( _trY, _trX );
188         } else { // Gimball lock has
            occurred
189             pose[3] = 0.0; // Set X-axis angle to
            zero
190             _trX = m[5]; //1 // And calculate Z-
            axis angle
191             _trY = m[1]; //2
192             pose[5] = atan2( _trY, _trX );
193         }
194
195         pose[0] = m[12];
196         pose[1] = m[13];
197         pose[2] = m[14];
198     }
199 }
200
201
202 float Matrix::det()
203 {
204     float det, result = 0, i = 1.0;
205     float Msub3[9];
206     int n;
207     for ( n = 0; n < 4; n++, i *= -1.0 ) {
208         submat( Msub3, 0, n );
209         det = det3( Msub3 );
210         result += m[n] * det * i;
211     }
212     return( result );
213 }
214
215 Matrix Matrix::inv(bool& success)
216 {
217     Matrix Mout;
218     float mdet = det();
219     if ( fabs( mdet ) < 0.000000000000005 ) {
220         cout << "Error_\u00a0matrix_\u00a0inverting!\u00a0" << mdet << endl;
221         return Mout;

```

```

222     }
223     float mtemp[9];
224     int i, j, sign;
225     for ( i = 0; i < 4; i++ ) {
226         for ( j = 0; j < 4; j++ ) {
227             sign = 1 - ( (i+j) % 2 ) * 2;
228             submat( mtemp, i, j );
229             Mout[i+j*4] = ( det3( mtemp ) * sign ) / mdet;
230         }
231     }
232     return Mout;
233 }
234
235
236 /**
237  * @brief Returns a sub matrix without row \ref i and column \
238  * ref j.
239  */
240 void Matrix::submat(float* submat, int i, int j)
241 {
242     int di, dj, si, sj;
243     // loop through 3x3 submatrix
244     for( di = 0; di < 3; di ++ ) {
245         for( dj = 0; dj < 3; dj ++ ) {
246             // map 3x3 element (destination) to 4x4 element (source)
247             si = di + ( ( di >= i ) ? 1 : 0 );
248             sj = dj + ( ( dj >= j ) ? 1 : 0 );
249             // copy element
250             submat[di * 3 + dj] = m[si * 4 + sj];
251         }
252     }
253 }
254
255 /**
256  * @brief Calculates the determinant of a 3x3 matrix
257  *
258  * @param M input 3x3 matrix
259  * @return determinant of input matrix
260  */
261 float Matrix::det3( const float *M )
262 {
263     float det;
264     det = (double)( M[0] * ( M[4]*M[8] - M[7]*M[5] )
265                 - M[1] * ( M[3]*M[8] - M[6]*M[5] )
266                 + M[2] * ( M[3]*M[7] - M[6]*M[4] ));
267     return ( det );
268 }
269
270 void Matrix::scale(float f)
271 {
272     if (f != 0.0)
273     {
274         int i;
275         for (i = 0; i < 16; i++) m[i] *= f;
276     }

```

```

277     }
278     float& Matrix::operator[](const int i)
279     {
280         return m[i];
281     }
282
283     const float& Matrix::operator[](const int i) const
284     {
285         return m[i];
286     }
287
288     Matrix Matrix::operator+(const Matrix& m) const
289     {
290         Matrix re;
291         int i;
292         for (i = 0; i < 15; i++)
293         {
294             re[i] = this->m[i] + m[i];
295         }
296         return re;
297     }
298
299     Matrix Matrix::operator-() const
300     {
301         Matrix re;
302         int i;
303         for (i = 0; i < 15; i++)
304         {
305             re[i] = -this->m[i];
306         }
307         return re;
308     }
309
310     Matrix Matrix::operator-(const Matrix& m) const
311     {
312         return *this + -m;
313     }
314
315     Matrix Matrix::operator*(const Matrix& m) const
316     {
317         Matrix re;
318         int i,j,k;
319         for (i = 0; i < 15; i++) re[i] = 0;
320         for (i = 0; i < 4; i++)
321         {
322             for (j = 0; j < 4; j++)
323             {
324                 for (k = 0; k < 4; k++)
325                 {
326                     re[i*4+j] += this->m[i*4+k] * m[k*4+j];
327                 }
328             }
329         }
330         return re;
331     }
332
333     float* Matrix::getData()

```

```

334     {
335         return m;
336     }
337
338     Matrix& Matrix::operator=(const Matrix& m)
339     {
340         if (&m != this)
341         {
342             int i;
343             for (i = 0; i < 15; i++)
344             {
345                 this->m[i] = m[i];
346             }
347             return *this;
348         } else throw runtime_error("Attempt to assign self.");
349     }
350
351     Matrix& Matrix::operator*=(const float f)
352     {
353         this->scale(f);
354         return *this;
355     }
356
357     Matrix& Matrix::operator*=(const Matrix& other)
358     {
359         *this = *this * other;
360         return *this;
361     }
362
363     std::string Matrix::to_s()
364     {
365         stringstream ss;
366         ss << "Matrix:" << endl;
367         ss << fixed;
368         for(int i = 0; i < 16; i++){
369             ss << setprecision(4) << (*this)[i] << " ";
370             if(i % 4 == 3) ss << "\n" << endl;
371         }
372         ss << endl;
373         return ss.str();
374     }
375 } // namespace cpp2014

```

Listing 3: Vertex.hpp

```

1  /**
2   * @file Vertex.hpp
3   *
4   * @date 05.12.2011
5   * @author Thomas Wiemann
6   */
7
8  #ifndef __Vertex_HPP__
9  #define __Vertex_HPP__
10
11  #include <iostream>
12  #include <cmath>
13  #include <iomanip>

```

```

14
15 #include "Global.hpp"
16
17 using namespace std;
18
19 namespace cpp2014
20 {
21
22     /**
23      * @brief   Vector representation with three floats for OpenGL
24      *
25      */
26     class Vertex {
27
28     public:
29
30         /**
31          * @brief   Construcs a default Vertex object
32          */
33         Vertex();
34
35         /**
36          * @brief   Construcs a Vertex object with given values
37          * @param x x-value
38          * @param y y-value
39          * @param z z-value
40          */
41         Vertex(float x, float y, float z);
42
43         /**
44          * @brief   Normalize a Vertex
45          */
46         void normalize();
47
48         /**
49          * @brief   Defines the vector addition
50          * @param   other Vertex to add to this one.
51          * @return  The sum of the two.
52          */
53         Vertex operator+(const Vertex& other) const;
54
55         /**
56          * @brief   Defines the vector subtraction
57          * @param   other Vertex to subtract from this one.
58          * @return  The difference of the two.
59          */
60         Vertex operator-(const Vertex& other) const;
61
62         /**
63          * @brief Defines the negation.
64          * @return A negated copy of this Vertex.
65          */
66         Vertex operator-() const;
67
68         /**
69          * @brief   Construcs the scalar division
70          * @param   f scalar

```

```

71     * @return  A scaled vector
72     */
73     Vertex operator/(float f) const;
74
75     /**
76     * @brief    Defines the scalar product
77     * @param    v Vertex
78     * @return   Scalar product (as a float)
79     */
80     float operator*(const Vertex& v) const;
81
82     /**
83     * @brief    Defines the scaling transformation
84     * @param    f The scaling factor
85     * @return   A scaled vector
86     */
87     Vertex operator*(float f) const;
88
89     /**
90     * @brief Assignment operator.
91     * @param other Vertex whose state this Vertex will assume
92     * @return This Vertex
93     */
94     Vertex& operator=(const Vertex& other);
95
96     /**
97     * @brief    Defines the access to a Vertex value (readonly
98     *           )
99     * @param i Index of the wanted value
100    * @return Const reference of entry at position i
101    */
102    const float& operator[](int i) const;
103
104    /**
105    * @brief    Defines the access to a Vertex value (read+
106    *           write)
107    * @param i Index wanted value
108    * @return Reference to entry at position i
109    */
110    float& operator[](int i);
111
112    /**
113    * @brief Multiply and assign.
114    * @param f Scaling factor
115    * @return this Vertex
116    */
117    Vertex& operator*=(const float f);
118
119    /**
120    * @brief Divide and assign.
121    * @param f Scaling factor
122    * @return this Vertex
123    */
124    Vertex& operator/=(const float f);

```

```

125     * @brief Add and assign.
126     * @param other Vertex to add to this one
127     * @return this Vertex
128     */
129     Vertex& operator+=(const Vertex& other);
130
131     /**
132     * @brief Subtract and assign.
133     * @param other Vertex to subtract from this one
134     * @return this Vertex
135     */
136     Vertex& operator-=(const Vertex& other);
137
138     /**
139     * @brief The three values of a vector
140     */
141     float x, y, z;
142
143     /**
144     * Returns a string representation of this object.
145     * @return A <tt>string</tt> representation.
146     */
147     std::string to_s() const;
148 };
149
150 } // namespace cpp2014
151
152 #endif

```

Listing 4: Vertex.hpp

```

1  /**
2   * @file Vertex.hpp
3   *
4   * @date 05.12.2011
5   * @author Thomas Wiemann
6   */
7
8  #ifndef __Vertex_HPP__
9  #define __Vertex_HPP__
10
11  #include <iostream>
12  #include <cmath>
13  #include <iomanip>
14
15  #include "Global.hpp"
16
17  using namespace std;
18
19  namespace cpp2014
20  {
21
22      /**
23       * @brief Vector representation with three floats for OpenGL
24       *
25       */
26      class Vertex {
27

```

```

28     public:
29
30         /**
31          * @brief   Construcs a default Vertex object
32          */
33         Vertex();
34
35         /**
36          * @brief   Construcs a Vertex object with given values
37          * @param x x-value
38          * @param y y-value
39          * @param z z-value
40          */
41         Vertex(float x, float y, float z);
42
43         /**
44          * @brief   Normalize a Vertex
45          */
46         void normalize();
47
48         /**
49          * @brief   Defines the vector addition
50          * @param   other Vertex to add to this one.
51          * @return  The sum of the two.
52          */
53         Vertex operator+(const Vertex& other) const;
54
55         /**
56          * @brief   Defines the vector subtraction
57          * @param   other Vertex to subtract from this one.
58          * @return  The difference of the two.
59          */
60         Vertex operator-(const Vertex& other) const;
61
62         /**
63          * @brief Defines the negation.
64          * @return A negated copy of this Vertex.
65          */
66         Vertex operator-() const;
67
68         /**
69          * @brief   Construcs the scalar division
70          * @param   f scalar
71          * @return  A scaled vector
72          */
73         Vertex operator/(float f) const;
74
75         /**
76          * @brief   Defines the scalar product
77          * @param   v Vertex
78          * @return  Scalar product (as a float)
79          */
80         float operator*(const Vertex& v) const;
81
82         /**
83          * @brief   Defines the scaling transformation
84          * @param   f The scaling factor

```



```

85     * @return A scaled vector
86     */
87     Vertex operator*(float f) const;
88
89     /**
90     * @brief Assignment operator.
91     * @param other Vertex whose state this Vertex will assume
92     * @return This Vertex
93     */
94     Vertex& operator=(const Vertex& other);
95
96     /**
97     * @brief Defines the access to a Vertex value (readonly
98     * )
99     * @param i Index of the wanted value
100    * @return Const reference of entry at position i
101    */
102    const float& operator[](int i) const;
103
104    /**
105    * @brief Defines the access to a Vertex value (read+
106    * write)
107    * @param i Index wanted value
108    * @return Reference to entry at position i
109    */
110    float& operator[](int i);
111
112    /**
113    * @brief Multiply and assign.
114    * @param f Scaling factor
115    * @return this Vertex
116    */
117    Vertex& operator*=(const float f);
118
119    /**
120    * @brief Divide and assign.
121    * @param f Scaling factor
122    * @return this Vertex
123    */
124    Vertex& operator/=(const float f);
125
126    /**
127    * @brief Add and assign.
128    * @param other Vertex to add to this one
129    * @return this Vertex
130    */
131    Vertex& operator+=(const Vertex& other);
132
133    /**
134    * @brief Subtract and assign.
135    * @param other Vertex to subtract from this one
136    * @return this Vertex
137    */
138    Vertex& operator-=(const Vertex& other);

```

```

139         * @brief   The three values of a vector
140         */
141         float x, y, z;
142
143         /**
144         * Returns a tring representation of this object.
145         * @return A <tt>string</tt> representation.
146         */
147         std::string to_s() const;
148     };
149
150 } // namespace cpp2014
151
152 #endif

```

Listing 5: Quaternion.cpp

```

1  #include "Quaternion.hpp"
2  #include <sstream>
3
4  namespace cpp2014 {
5
6      Quaternion::Quaternion(): w(0.), x(0.), y(0.), z(0.) {}
7
8
9      Quaternion::Quaternion(Vertex vec, float angle)
10     {
11         *this = fromAxis(vec, angle);
12     }
13
14     Quaternion::Quaternion(float x, float y, float z, float w):
15         w(w), x(x), y(y), z(z) {}
16
17     Quaternion::Quaternion(float* vec, float w)
18     {
19         *this = fromAxis(Vertex(vec[0], vec[1], vec[2]), w);
20     }
21     Quaternion::~Quaternion() {}
22
23     Quaternion Quaternion::fromAxis(Vertex axis, float angle)
24     {
25         angle /= 2;
26         return Quaternion(axis[0] * sin(angle), axis[1] * sin(angle),
27             axis[2] * sin(angle), cos(angle));
28     }
29
30     Quaternion Quaternion::getConjugate() const
31     {
32         return Quaternion(-x, -y, -z, w);
33     }
34
35     Quaternion Quaternion::operator*(const Quaternion& rq) const
36     {
37         return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
38             w * rq.y + y * rq.w + z * rq.x - x * rq.z,
39             w * rq.z + z * rq.w + x * rq.y - y * rq.x,
40             w * rq.w - x * rq.x - y * rq.y - z * rq.z);
41     }

```

```

42
43     Vertex Quaternion::operator*(const Vertex& vec) const
44     {
45         Vertex vn(vec);
46         vn.normalize();
47
48         Quaternion vecQuat, resQuat;
49         vecQuat.x = vn.x;
50         vecQuat.y = vn.y;
51         vecQuat.z = vn.z;
52         vecQuat.w = 0.0f;
53
54         resQuat = vecQuat * getConjugate();
55         resQuat = *this * resQuat;
56
57         return Vertex(resQuat.x, resQuat.y, resQuat.z);
58     }
59
60     std::string Quaternion::to_s() const
61     {
62         stringstream ss;
63         ss << "Quaternion:" << "\n" << x << "\n" << y << "\n" << z <<
64             "\n" << w << std::endl;
65         return ss.str();
66     }
67 }

```

Listing 6: Quaternion.cpp

```

1  #include "Quaternion.hpp"
2  #include <sstream>
3
4  namespace cpp2014 {
5
6      Quaternion::Quaternion(): w(0.), x(0.), y(0.), z(0.) {}
7
8
9      Quaternion::Quaternion(Vertex vec, float angle)
10     {
11         *this = fromAxis(vec, angle);
12     }
13
14     Quaternion::Quaternion(float x, float y, float z, float w):
15         w(w), x(x), y(y), z(z) {}
16
17     Quaternion::Quaternion(float* vec, float w)
18     {
19         *this = fromAxis(Vertex(vec[0], vec[1], vec[2]), w);
20     }
21     Quaternion::~Quaternion() {}
22
23     Quaternion Quaternion::fromAxis(Vertex axis, float angle)
24     {
25         angle /= 2;
26         return Quaternion(axis[0] * sin(angle), axis[1] * sin(angle),
27             axis[2] * sin(angle), cos(angle));
28     }

```

```

29
30 Quaternion Quaternion::getConjugate() const
31 {
32     return Quaternion(-x, -y, -z, w);
33 }
34
35 Quaternion Quaternion::operator*(const Quaternion& rq) const
36 {
37     return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
38                       w * rq.y + y * rq.w + z * rq.x - x * rq.z,
39                       w * rq.z + z * rq.w + x * rq.y - y * rq.x,
40                       w * rq.w - x * rq.x - y * rq.y - z * rq.z);
41 }
42
43 Vertex Quaternion::operator*(const Vertex& vec) const
44 {
45     Vertex vn(vec);
46     vn.normalize();
47
48     Quaternion vecQuat, resQuat;
49     vecQuat.x = vn.x;
50     vecQuat.y = vn.y;
51     vecQuat.z = vn.z;
52     vecQuat.w = 0.0f;
53
54     resQuat = vecQuat * getConjugate();
55     resQuat = *this * resQuat;
56
57     return Vertex(resQuat.x, resQuat.y, resQuat.z);
58 }
59
60 std::string Quaternion::to_s() const
61 {
62     stringstream ss;
63     ss << "Quaternion:" << "\n" << x << "\n" << y << "\n" << z <<
64         "\n" << w << std::endl;
65     return ss.str();
66 }
67 }

```

## Aufgabe 7.2 Zweidimensionaler Zugriff auf Matrizen

Wäre das Array zweidimensional, würde eine einfache Anwendung des `[]`-Operators eine Zeile, also einen Pointer des Arrays zurückliefern. Diese wäre dann wiederum mit `[]` indizierbar. Für den Compiler ergäbe sich bei `m[x][y]` der Aufruf `m.operator[](x)[y]`. Allerdings wäre hierbei dann kein Bounds-Check möglich, weil `[]` kein C++-Operator ist. Abhilfe könnte man schaffen, indem man eine dedizierte Klasse erstellt, deren einzige öffentliche Funktionalität `operator[](int i)` ist, die die Eingabe überprüfen kann, und unter Aufruf von `[]` aus der ersten Klasse den zweidimensionalen Zugriff realisiert.

```

1 class Matrix {
2     class Proxy {
3         float* data;

```

```

4         Proxy(float* f): data(f) {}
5         float& operator[](int i)
6         {
7             if(i >= 0 && i < 5)
8                 return data[i];
9         }
10    }
11
12    float data[5][5];
13    Proxy operator[](int i)
14    {
15        if(i >= 0 && i < 5)
16            return Proxy(data[i]);
17    }
18 }
19

```