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

```
/* Copyright (C) 2011 Uni Osnabrück
 * This file is part of the LAS VEGAS Reconstruction Toolkit,
 *
 * LAS VEGAS is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * LAS VEGAS is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA
 */


/*
 * Matrix.hpp
 *
 *  @date 26.08.2008
 *  @author Thomas Wiemann (twiemann@uos.de)
 */

#ifndef MATRIX_H_
#define MATRIX_H_
```

```

#include <iostream>
#include <fstream>
#include <iomanip>

#include "Vertex.hpp"

#define _USE_MATH_DEFINES
#include <cmath>

#ifndef M_PI
#define M_PI 3.141592654
#endif

using namespace std;

namespace cpp2014 {

   /**
    * @brief    A 4x4 matrix class definition for use with the provided
    *                   vertex types.
    */
   class Matrix {

      public:

         /**
          * @brief      Default constructor. Initializes a identity matrix.
          */
         Matrix();

         /**
          * @brief      Initializes a matrix wit the given data array. Ensure
          *                      that the array has exactly 16 fields.
          */
         Matrix(float* matrix);
         /**
          * @brief      Copy constructor.
          */
         Matrix(const Matrix& other);

         /**
          * @brief      Constructs a matrix from given axis and angle. Tries to
          *                      avoid a gimbal lock.
          */
         Matrix(Vertex axis, float angle);

         Matrix(const Vertex &position, const Vertex &angles);


         /**
          * Destructor
```

```
         */
        ~Matrix();


        /**
         * @brief      Matrix-Matrix multiplication.
         * @param m The multiplicand.
         * @return This matrix.
         */
        Matrix operator*(const Matrix& m) const;


        /**
         * @brief      Matrix addition operator. Returns a new matrix
         * @param m Matrix to be assigned.
         * @return Thi matrix.
         */
        Matrix& operator=(const Matrix& m);

        /**
         * @brief      Matrix addition operator
         * @param m Addend.
         * @return A new matrix which is the sum of this one and the addend.
         */
        Matrix operator+(const Matrix& m) const;

        /**
         * @brief Elementwisely subtract a matrix from this one.
         * @param m Subtrahend
         * @return The difference of this mtrix and the subtrahend.
         */
        Matrix operator-(const Matrix& m) const;

        /**
         * @brief Negate all entries in this matrix.
         * @return A new <tt>Matrix</tt> with entries negated.
         */
        Matrix operator-() const;

        /**
         * @brief      Matrix-Matrix multiplication (array based). Mainly
         *                      defined for compatibility with other math libs.
         *                      ensure that the used array has at least 16 elements
         *                      to avoid memory access violations.
         */

        /// TODO: DEFINE OPERATOR HERE!

        /**
         * THIS MAKES NO SENSE
         * @brief      Multiplication of Matrix and Vertex types
```

```
         */

        /* Matrix& operator*(const Vertex& v) const; */

        /**
         * @brief      Sets the given index of the Matrix's data field
         *                     to the provided value.
         *
         * @param      i               Field index of the matrix
         * @param      value   new value
         */
        void set(int i, float value);

        /**
         * @brief      Transposes the current matrix
         */
        void transpose();

        /**
         * @brief      Computes an Euler representation (x, y, z) plus three
         *                     rotation values in rad. Rotations are with respect to
         *                     the x, y, z axes.
         */
        void toPostionAngle(float pose[6]);


        /**
         * @brief      Matrix scaling with self assignment.
         */
        Matrix& operator*=(const float f);


        /**
         * @brief      Matrix-Matrix multiplication with self assigment.
         */
        Matrix& operator*=(const Matrix& other);

        /**
         * @brief      Matrix-Matrix multiplication (array based). See \ref{operator*}.
         */
        /// TODO: DEFINE OPERATOR HERE!

        /**
         * @brief      Returns the internal data array. Unsafe. Will probably
         *                     removed in one of the next versions.
         */
        float* getData();

        /**
         * @brief      Indexed element (reading) access.
```

```
             */
            const float& operator[](const int i) const;

            /**
             * @brief      Writeable index access
             */
            float& operator[](const int i);

            /**
             * @brief   Returns the matrix's determinant
             */
            float det();

            /**
             * @brief   Inverts the matrix. Success is true if operation was successful
             */
            Matrix inv(bool& success);

            /**
             * Returns a tring representation of this object.
             * @return A <tt>string</tt> representation.
             */
            std::string to_s();

        private:

            /**
             * @brief   Returns a sub matrix without row \ref i and column \ref j.
             */
            void submat(float* submat, int i, int j);

            /**
             * @brief    Calculates the determinant of a 3x3 matrix
             *
             * @param    M  input 3x3 matrix
             * @return   determinant of input matrix
             */
            float det3( const float *M );

            /**
             * @brief      Scales the matrix elemnts by the given factor
             */
            void scale(float f);

            /// Internal data array
            float m[16];

    };


} // namespace cpp2014
#endif /* MATRIX_H_ */
```

Listing 2: Matrix.cpp

```cpp
/*
 * Matrix.hpp
 *
 *  @date 26.08.2008
 *  @author Thomas Wiemann (twiemann@uos.de)
 */
#include "Matrix.hpp"
#include <sstream>
#include <stdexcept>

namespace cpp2014
{

   Matrix::Matrix()
   {
      for(int i = 0; i < 16; i++) m[i] = 0;
      m[0] = m[5] = m[10] = m[15] = 1;
   }


   Matrix::Matrix(float* matrix)
   {
      for(int i = 0; i < 16; i++) m[i] = matrix[i];
   }


   Matrix::Matrix(const Matrix& other)
   {
      for(int i = 0; i < 16; i++) m[i] = other[i];
   }

   Matrix::Matrix(Vertex axis, float angle)
   {
      // Check for gimbal lock
      if(fabs(angle) < 0.0001)
      {

         bool invert_z = axis.z < 0;

         //Angle to yz-plane
         float pitch = atan2(axis.z, axis.x) - M_PI_2;
         if(pitch < 0.0f) pitch += 2.0f * M_PI;

         if(axis.x == 0.0f && axis.z == 0.0) pitch = 0.0f;

         //Transform axis into yz-plane
         axis.x =  axis.x * cos(pitch) + axis.z * sin(pitch);
         axis.z = -axis.x * sin(pitch) + axis.z * cos(pitch);

         //Angle to y-Axis
         float yaw = atan2(axis.y, axis.z);
         if(yaw < 0) yaw += 2 * M_PI;

         Matrix m1, m2, m3;

         if(invert_z) yaw = -yaw;

```

```
        cout << "YAW: " << yaw << " PITCH: " << pitch << endl;

        if(fabs(yaw)   > 0.0001){
           m2 = Matrix(Vertex(1.0, 0.0, 0.0), yaw);
           m3 = m3 * m2;
        }

        if(fabs(pitch) > 0.0001){
           m1 = Matrix(Vertex(0.0, 1.0, 0.0), pitch);
           m3 = m3 * m1;
        }

        for(int i = 0; i < 16; i++) m[i] = m3[i];

     } else {
        float c = cos(angle);
        float s = sin(angle);
        float t = 1.0f - c;
        float tmp1, tmp2;

        // Normalize axis
        Vertex a(axis);
        a.normalize();

        m[ 0] = c + a.x * a.x * t;
        m[ 5] = c + a.y * a.y * t;
        m[10] = c + a.z * a.z * t;

        tmp1 = a.x * a.y * t;
        tmp2 = a.z * s;
        m[ 4] = tmp1 + tmp2;
        m[ 1] = tmp1 - tmp2;

        tmp1 = a.x * a.z * t;
        tmp2 = a.y * s;
        m[ 8] = tmp1 - tmp2;
        m[ 2] = tmp1 + tmp2;

        tmp1 = a.y * a.z * t;
        tmp2 = a.x * s;
        m[ 9] = tmp1 + tmp2;
        m[ 6] = tmp1 - tmp2;

        m[ 3] = m[ 7] = m[11] = 0.0;
        m[12] = m[13] = m[14] = 0.0;
        m[15] = 1.0;
     }
  }


  Matrix::Matrix(const Vertex &position, const Vertex &angles)
  {
     float sx = sin(angles[0]);
     float cx = cos(angles[0]);
     float sy = sin(angles[1]);
     float cy = cos(angles[1]);
     float sz = sin(angles[2]);
```

```
   float cz = cos(angles[2]);

   m[0]  = cy*cz;
   m[1]  = sx*sy*cz + cx*sz;
   m[2]  = -cx*sy*cz + sx*sz;
   m[3]  = 0.0;
   m[4]  = -cy*sz;
   m[5]  = -sx*sy*sz + cx*cz;
   m[6]  = cx*sy*sz + sx*cz;
   m[7]  = 0.0;
   m[8]  = sy;
   m[9]  = -sx*cy;
   m[10] = cx*cy;

   m[11] = 0.0;

   m[12] = position[0];
   m[13] = position[1];
   m[14] = position[2];
   m[15] = 1;
}

Matrix::~Matrix() { }

/**
 * @brief    Transposes the current matrix
 */
void Matrix::transpose()
{
   float m_tmp[16];
   m_tmp[0]  = m[0];
   m_tmp[4]  = m[1];
   m_tmp[8]  = m[2];
   m_tmp[12] = m[3];
   m_tmp[1]  = m[4];
   m_tmp[5]  = m[5];
   m_tmp[9]  = m[6];
   m_tmp[13] = m[7];
   m_tmp[2]  = m[8];
   m_tmp[6]  = m[9];
   m_tmp[10] = m[10];
   m_tmp[14] = m[11];
   m_tmp[3]  = m[12];
   m_tmp[7]  = m[13];
   m_tmp[11] = m[14];
   m_tmp[15] = m[15];
   for(int i = 0; i < 16; i++) m[i] = m_tmp[i];
}

/**
 * @brief    Computes an Euler representation (x, y, z) plus three
 *                   rotation values in rad. Rotations are with respect to
 *                   the x, y, z axes.
 */
void Matrix::toPostionAngle(float pose[6])
```

```
{
   if(pose != 0){
      float _trX, _trY;
      if(m[0] > 0.0) {
         pose[4] = asin(m[8]);
      } else {
         pose[4] = (float)M_PI - asin(m[8]);
      }
      // rPosTheta[1] =  asin( m[8]);      // Calculate Y-axis angle

      float  C    =  cos( pose[4] );
      if ( fabs( C ) > 0.005 )  {          // Gimball lock?
         _trX      =  m[10] / C;          // No, so get X-axis angle
         _trY      =  -m[9] / C;
         pose[3]   = atan2( _trY, _trX );
         _trX      =  m[0] / C;           // Get Z-axis angle
         _trY      = -m[4] / C;
         pose[5]   = atan2( _trY, _trX );
      } else {                             // Gimball lock has occurred
         pose[3] = 0.0;                    // Set X-axis angle to zero
         _trX      =  m[5];  //1           // And calculate Z-axis angle
         _trY      =  m[1];  //2
         pose[5]   = atan2( _trY, _trX );
      }

      pose[0] = m[12];
      pose[1] = m[13];
      pose[2] = m[14];
   }
}


float Matrix::det()
{
   float det, result = 0, i = 1.0;
   float Msub3[9];
   int    n;
   for ( n = 0; n < 4; n++, i *= -1.0 ) {
      submat( Msub3, 0, n );
      det     = det3( Msub3 );
      result += m[n] * det * i;
   }
   return( result );
}

Matrix Matrix::inv(bool& success)
{
   Matrix Mout;
   float  mdet = det();
   if ( fabs( mdet ) < 0.00000000000005 ) {
      cout << "Error matrix inverting! " << mdet << endl;
      return Mout;
```

```
    }
    float  mtemp[9];
    int     i, j, sign;
    for ( i = 0; i < 4; i++ ) {
       for ( j = 0; j < 4; j++ ) {
          sign = 1 - ( (i +j) % 2 ) * 2;
          submat( mtemp, i, j );
          Mout[i+j*4] = ( det3( mtemp ) * sign ) / mdet;
       }
    }
    return Mout;
}


/**
 * @brief   Returns a sub matrix without row \ref i and column \ref j.
 */
void Matrix::submat(float* submat, int i, int j)
{
    int di, dj, si, sj;
    // loop through 3x3 submatrix
    for( di = 0; di < 3; di ++ ) {
       for( dj = 0; dj < 3; dj ++ ) {
          // map 3x3 element (destination) to 4x4 element (source)
          si = di + ( ( di >= i ) ? 1 : 0 );
          sj = dj + ( ( dj >= j ) ? 1 : 0 );
          // copy element
          submat[di * 3 + dj] = m[si * 4 + sj];
       }
    }
}

/**
 * @brief    Calculates the determinant of a 3x3 matrix
 *
 * @param    M  input 3x3 matrix
 * @return   determinant of input matrix
 */
float Matrix::det3( const float *M )
{
    float det;
    det = (double)(  M[0] * ( M[4]*M[8] - M[7]*M[5] )
          - M[1] * ( M[3]*M[8] - M[6]*M[5] )
          + M[2] * ( M[3]*M[7] - M[6]*M[4] ));
    return ( det );
}


void Matrix::scale(float f)
{
    if (f != 0.0)
    {
       int i;
       for (i = 0; i < 16; i++) m[i] *= f;
    }
```

```
}
float& Matrix::operator[](const int i)
{
   return m[i];
}

const float& Matrix::operator[](const int i) const
{
   return m[i];
}

Matrix Matrix::operator+(const Matrix& m) const
{
   Matrix re;
   int i;
   for (i = 0; i < 15; i++)
   {
      re[i] = this->m[i] + m[i];
   }
   return re;
}

Matrix Matrix::operator-() const
{
   Matrix re;
   int i;
   for (i = 0; i < 15; i++)
   {
      re[i] = -this->m[i];
   }
   return re;
}

Matrix Matrix::operator-(const Matrix& m) const
{
   return *this + -m;
}

Matrix Matrix::operator*(const Matrix& m) const
{
   Matrix re;
   int i,j,k;
   for (i = 0; i < 15; i++) re[i] = 0;
   for (i = 0; i < 4; i++)
   {
      for (j = 0; j < 4; j++)
      {
         for (k = 0; k < 4; k++)
         {
            re[i*4+j] += this->m[i*4+k] * m[k*4+j];
         }
      }
   }
   return re;
}

float* Matrix::getData()
```

```
    {
        return m;
    }

    Matrix& Matrix::operator=(const Matrix& m)
    {
        if (&m != this)
        {
            int i;
            for (i = 0; i < 15; i++)
            {
                this->m[i] = m[i];
            }
            return *this;
        } else throw runtime_error("Attempt␣to␣assign␣self.");
    }

    Matrix& Matrix::operator*=(const float f)
    {
        this->scale(f);
        return *this;
    }

    Matrix& Matrix::operator*=(const Matrix& other)
    {
        *this = *this * other;
        return *this;
    }

    std::string Matrix::to_s()
    {
        stringstream ss;
        ss << "Matrix:" << endl;
        ss << fixed;
        for(int i = 0; i < 16; i++){
            ss << setprecision(4) << (*this)[i] << "␣";
            if(i % 4 == 3) ss << "␣" <<  endl;
        }
        ss << endl;
        return ss.str();
    }
} // namespace cpp2014
```

Listing 3: Vertex.hpp

```
/**
 *  @file Vertex.hpp
 *
 *  @date 05.12.2011
 *  @author Thomas Wiemann
 */

#ifndef __Vertex_HPP__
#define __Vertex_HPP__

#include <iostream>
#include <cmath>
#include <iomanip>
```

```cpp
#include "Global.hpp"

using namespace std;

namespace cpp2014
{

    /**
     * @brief   Vector representation with three floats for OpenGL
     *
     */
    class Vertex {

        public:

            /**
             * @brief   Construcs a default Vertex object
             */
            Vertex();

            /**
             * @brief   Construcs a Vertex object with given values
             * @param x x-value
             * @param y y-value
             * @param z z-value
             */
            Vertex(float x, float y, float z);

            /**
             * @brief   Normalize a Vertex
             */
            void normalize();

            /**
             * @brief   Defines the vector addition
             * @param   other Vertex to add to this one.
             * @return  The sum of the two.
             */
            Vertex operator+(const Vertex& other) const;

            /**
             * @brief   Defines the vector subtraction
             * @param   other Vertex to subtract from this one.
             * @return  The difference of the two.
             */
            Vertex operator-(const Vertex& other) const;

            /**
             * @brief Defines the negation.
             * @return A negated copy of this Vertex.
             */
            Vertex operator-() const;

            /**
             * @brief   Construcs the scalar division
             * @param   f scalar
```

```
 * @return  A scaled vector
 */
Vertex operator/(float f) const;

/**
 * @brief   Defines the scalar product
 * @param   v Vertex
 * @return  Scalar product (as a float)
 */
float operator*(const Vertex& v) const;

/**
 * @brief   Defines the scaling transformation
 * @param   f The scaling factor
 * @return  A scaled vector
 */
Vertex operator*(float f) const;

/**
 * @brief Assignment operator.
 * @param other Vertex whose state this Vertex will assume.
 * @return This Vertex
 */
Vertex& operator=(const Vertex& other);

/**
 * @brief   Defines the access to a Vertex value (readonly)
 * @param i Index of the wanted value
 * @return Const reference of entry at position i
 */
const float& operator[](int i) const;

/**
 * @brief   Defines the access to a Vertex value (read+write)
 * @param i Index wanted value
 * @return Reference to entry at position i
 */
float& operator[](int i);

/**
 * @brief Multiply and assign.
 * @param f Scaling factor
 * @return this Vertex
 */
Vertex& operator*=(const float f);

/**
 * @brief Divide and assign.
 * @param f Scaling factor
 * @return this Vertex
 */
Vertex& operator/=(const float f);

/**
```

```
         * @brief Add and assign.
         * @param other Vertex to add to this one
         * @return this Vertex
         */
        Vertex& operator+=(const Vertex& other);

        /**
         * @brief Subtract and assign.
         * @param other Vertex to subtract from this one
         * @return this Vertex
         */
        Vertex& operator-=(const Vertex& other);

        /**
         * @brief   The three values of a vector
         */
        float x, y, z;

        /**
         * Returns a tring representation of this object.
         * @return A <tt>string</tt> representation.
         */
        std::string to_s() const;
    };

} // namespace cpp2014

#endif
```

Listing 4: Vertex.hpp

```
/**
 *  @file Vertex.hpp
 *
 *  @date 05.12.2011
 *  @author Thomas Wiemann
 */

#ifndef __Vertex_HPP__
#define __Vertex_HPP__

#include <iostream>
#include <cmath>
#include <iomanip>

#include "Global.hpp"

using namespace std;

namespace cpp2014
{

    /**
     * @brief   Vector representation with three floats for OpenGL
     *
     */
    class Vertex {

```

```
	public:

		/**
		 * @brief   Construcs a default Vertex object
		 */
		Vertex();

		/**
		 * @brief   Construcs a Vertex object with given values
		 * @param x x-value
		 * @param y y-value
		 * @param z z-value
		 */
		Vertex(float x, float y, float z);

		/**
		 * @brief   Normalize a Vertex
		 */
		void normalize();

		/**
		 * @brief   Defines the vector addition
		 * @param   other Vertex to add to this one.
		 * @return  The sum of the two.
		 */
		Vertex operator+(const Vertex& other) const;

		/**
		 * @brief   Defines the vector subtraction
		 * @param   other Vertex to subtract from this one.
		 * @return  The difference of the two.
		 */
		Vertex operator-(const Vertex& other) const;

		/**
		 * @brief Defines the negation.
		 * @return A negated copy of this Vertex.
		 */
		Vertex operator-() const;

		/**
		 * @brief   Construcs the scalar division
		 * @param   f scalar
		 * @return  A scaled vector
		 */
		Vertex operator/(float f) const;

		/**
		 * @brief   Defines the scalar product
		 * @param   v Vertex
		 * @return  Scalar product (as a float)
		 */
		float operator*(const Vertex& v) const;

		/**
		 * @brief   Defines the scaling transformation
		 * @param   f The scaling factor
```

```
         * @return  A scaled vector
         */
        Vertex operator*(float f) const;

        /**
         * @brief Assignment operator.
         * @param other Vertex whose state this Vertex will assume.
         * @return This Vertex
         */
        Vertex& operator=(const Vertex& other);

        /**
         * @brief   Defines the access to a Vertex value (readonly)
         * @param i Index of the wanted value
         * @return Const reference of entry at position i
         */
        const float& operator[](int i) const;

        /**
         * @brief   Defines the access to a Vertex value (read+write)
         * @param i Index wanted value
         * @return Reference to entry at position i
         */
        float& operator[](int i);

        /**
         * @brief Multiply and assign.
         * @param f Scaling factor
         * @return this Vertex
         */
        Vertex& operator*=(const float f);

        /**
         * @brief Divide and assign.
         * @param f Scaling factor
         * @return this Vertex
         */
        Vertex& operator/=(const float f);

        /**
         * @brief Add and assign.
         * @param other Vertex to add to this one
         * @return this Vertex
         */
        Vertex& operator+=(const Vertex& other);

        /**
         * @brief Subtract and assign.
         * @param other Vertex to subtract from this one
         * @return this Vertex
         */
        Vertex& operator-=(const Vertex& other);

        /**
```

```
         * @brief   The three values of a vector
         */
        float x, y, z;

        /**
         * Returns a tring representation of this object.
         * @return A <tt>string</tt> representation.
         */
        std::string to_s() const;
  };

} // namespace cpp2014

#endif
```

Listing 5: Quaternion.cpp

```
#include "Quaternion.hpp"
#include <sstream>

namespace cpp2014 {

   Quaternion::Quaternion(): w(0.), x(0.), y(0.), z(0.) {}


   Quaternion::Quaternion(Vertex vec, float angle)
   {
      *this = fromAxis(vec, angle);
   }

   Quaternion::Quaternion(float x, float y, float z, float w):
      w(w), x(x), y(y), z(z) {}

   Quaternion::Quaternion(float* vec, float w)
   {
      *this = fromAxis(Vertex(vec[0], vec[1], vec[2]), w);
   }
   Quaternion::~Quaternion() {}

   Quaternion Quaternion::fromAxis(Vertex axis, float angle)
   {
      angle /= 2;
      return Quaternion(axis[0] * sin(angle), axis[1] * sin(angle),
            axis[2] * sin(angle), cos(angle));
   }

   Quaternion Quaternion::getConjugate() const
   {
      return Quaternion(-x, -y, -z, w);
   }

   Quaternion Quaternion::operator*(const Quaternion& rq) const
   {
      return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
            w * rq.y + y * rq.w + z * rq.x - x * rq.z,
            w * rq.z + z * rq.w + x * rq.y - y * rq.x,
            w * rq.w - x * rq.x - y * rq.y - z * rq.z);
   }
```

```cpp

    Vertex Quaternion::operator*(const Vertex& vec) const
    {
       Vertex vn(vec);
       vn.normalize();

       Quaternion vecQuat, resQuat;
       vecQuat.x = vn.x;
       vecQuat.y = vn.y;
       vecQuat.z = vn.z;
       vecQuat.w = 0.0f;

       resQuat = vecQuat * getConjugate();
       resQuat = *this * resQuat;

       return Vertex(resQuat.x, resQuat.y, resQuat.z);
    }

    std::string Quaternion::to_s() const
    {
       stringstream ss;
       ss << "Quaternion:" << "\n" << x << "\n" << y << "\n" << z <<
            "\n" << w << std::endl;
       return ss.str();
    }

}
```

Listing 6: Quaternion.cpp

```cpp
#include "Quaternion.hpp"
#include <sstream>

namespace cpp2014 {

    Quaternion::Quaternion(): w(0.), x(0.), y(0.), z(0.) {}


    Quaternion::Quaternion(Vertex vec, float angle)
    {
       *this = fromAxis(vec, angle);
    }

    Quaternion::Quaternion(float x, float y, float z, float w):
       w(w), x(x), y(y), z(z) {}

    Quaternion::Quaternion(float* vec, float w)
    {
       *this = fromAxis(Vertex(vec[0], vec[1], vec[2]), w);
    }
    Quaternion::~Quaternion() {}

    Quaternion Quaternion::fromAxis(Vertex axis, float angle)
    {
       angle /= 2;
       return Quaternion(axis[0] * sin(angle), axis[1] * sin(angle),
             axis[2] * sin(angle), cos(angle));
    }
```

```
    Quaternion Quaternion::getConjugate() const
    {
       return Quaternion(-x, -y, -z, w);
    }

    Quaternion Quaternion::operator*(const Quaternion& rq) const
    {
       return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
             w * rq.y + y * rq.w + z * rq.x - x * rq.z,
             w * rq.z + z * rq.w + x * rq.y - y * rq.x,
             w * rq.w - x * rq.x - y * rq.y - z * rq.z);
    }

    Vertex Quaternion::operator*(const Vertex& vec) const
    {
       Vertex vn(vec);
       vn.normalize();

       Quaternion vecQuat, resQuat;
       vecQuat.x = vn.x;
       vecQuat.y = vn.y;
       vecQuat.z = vn.z;
       vecQuat.w = 0.0f;

       resQuat = vecQuat * getConjugate();
       resQuat = *this * resQuat;

       return Vertex(resQuat.x, resQuat.y, resQuat.z);
    }

    std::string Quaternion::to_s() const
    {
       stringstream ss;
       ss << "Quaternion:" << "\n" << x << "\n" << y << "\n" << z <<
            "\n" << w << std::endl;
       return ss.str();
    }

}
```

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 `[]` indiziebar. 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.

```
    class Matrix {
       class Proxy {
          float* data;
```

```
        Proxy(float* f): data(f) {}
        float& operator[](int i)
        {
            if(i >= 0 && i < 5)
                return data[i];
        }
    }

    float data[5][5];
    Proxy operator[](int i)
    {
        if(i >= 0 && i < 5)
        return Proxy(data[i]);

    }
}
```