Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 FAMath Namespace Reference

Has utility functions, Vector2D, Vector3D, Vector4D, Matrix4x4, and Quaternion classes.

Classes

- class Matrix4x4

  A matrix class used for 4x4 matrices and their manipulations.
- class Quaternion
- class Vector2D

  A vector class used for 2D vectors/points and their manipulations.
- class Vector3D

  A vector class used for 3D vectors/points and their manipulations.
- class Vector4D

  A vector class used for 4D vectors/points and their manipulations.

Functions

- bool **compareFloats** (float x, float y, float epsilon)
- bool **compareDoubles** (double x, double y, double epsilon)
- bool zeroVector (const Vector2D &a)

  Returns true if a is the zero vector.
- Vector2D operator+ (const Vector2D &a, const Vector2D &b)

  2D vector addition.
- Vector2D operator- (const Vector2D &v)

  2D vector negation.
- Vector2D operator- (const Vector2D &a, const Vector2D &b)

  2D vector subtraction.
- Vector2D operator* (const Vector2D &a, const float &k)

  2D vector scalar multiplication. Returns $a * k$, where a is a vector and k is a scalar(float)
- Vector2D operator* (const float &k, const Vector2D &a)

  2D vector scalar multiplication. Returns $k * a$, where a is a vector and k is a scalar(float)
- Vector2D operator/ (const Vector2D &a, const float &k)

2D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.

- float dotProduct (const Vector2D &a, const Vector2D &b)

  Returns the dot product between two 2D vectors.
- float length (const Vector2D &v)

  Returns the length(magnitude) of the 2D vector v.
- Vector2D norm (const Vector2D &v)

  Normalizes the 2D vector v. If the 2D vector is the zero vector v is returned.
- Vector2D PolarToCartesian (const Vector2D &v)

  Converts the 2D vector v from polar coordinates to cartesian coordinates. v should = (r, theta(degrees)) The returned 2D vector = (x, y)
- Vector2D CartesianToPolar (const Vector2D &v)

  Converts the 2D vector v from cartesian coordinates to polar coordinates. v should = (x, y, z) If vx is zero then no conversion happens and v is returned.
  The returned 2D vector = (r, theta(degrees)).
- Vector2D Projection (const Vector2D &a, const Vector2D &b)

  Returns a 2D vector that is the projection of a onto b. If b is the zero vector a is returned.
- bool zeroVector (const Vector3D &a)

  Returns true if a is the zero vector.
- Vector3D operator+ (const Vector3D &a, const Vector3D &b)

  3D vector addition.
- Vector3D operator- (const Vector3D &v)

  3D vector negeation.
- Vector3D operator- (const Vector3D &a, const Vector3D &b)

  3D vector subtraction.
- Vector3D operator* (const Vector3D &a, const float &k)

  *3D vector scalar multiplication. Returns a * k, where a is a vector and k is a scalar(float)*
- Vector3D operator* (const float &k, const Vector3D &a)

  *3D vector scalar multiplication. Returns k * a, where a is a vector and k is a scalar(float)*
- Vector3D operator/ (const Vector3D &a, const float &k)

  3D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.
- float dotProduct (const Vector3D &a, const Vector3D &b)

  Returns the dot product between two 3D vectors.
- Vector3D crossProduct (const Vector3D &a, const Vector3D &b)

  Returns the cross product between two 3D vectors.
- float length (const Vector3D &v)

  Returns the length(magnitude) of the 3D vector v.
- Vector3D norm (const Vector3D &v)

  Normalizes the 3D vector v. If the 3D vector is the zero vector v is returned.
- Vector3D CylindricalToCartesian (const Vector3D &v)

  Converts the 3D vector v from cylindrical coordinates to cartesian coordinates. v should = (r, theta(degrees), z).
  The returned 3D vector = (x, y ,z).
- Vector3D CartesianToCylindrical (const Vector3D &v)

  Converts the 3D vector v from cartesian coordinates to cylindrical coordinates. v should = (x, y, z).
  If vx is zero then no conversion happens and v is returned.
  The returned 3D vector = (r, theta(degrees), z).
- Vector3D SphericalToCartesian (const Vector3D &v)

  Converts the 3D vector v from spherical coordinates to cartesian coordinates. v should = (pho, phi(degrees), theta(degrees)).
  The returned 3D vector = (x, y, z)
- Vector3D CartesianToSpherical (const Vector3D &v)

Converts the 3D vector v from cartesian coordinates to spherical coordinates. If v is the zero vector or if vx is zero then no conversion happens and v is returned.
The returned 3D vector = (r, phi(degrees), theta(degrees)).

- Vector3D Projection (const Vector3D &a, const Vector3D &b)

  Returns a 3D vector that is the projection of a onto b. If b is the zero vector a is returned.
- bool zeroVector (const Vector4D &a)

  Returns true if a is the zero vector.
- Vector4D operator+ (const Vector4D &a, const Vector4D &b)

  4D vector addition.
- Vector4D operator- (const Vector4D &v)

  4D vector negation.
- Vector4D operator- (const Vector4D &a, const Vector4D &b)

  4D vector subtraction.
- Vector4D operator∗ (const Vector4D &a, const float &k)

  4D vector scalar multiplication. Returns a ∗ k, where a is a vector and k is a scalar(float)
- Vector4D operator∗ (const float &k, const Vector4D &a)

  4D vector scalar multiplication. Returns k ∗ a, where a is a vector and k is a scalar(float)
- Vector4D operator/ (const Vector4D &a, const float &k)

  4D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.
- float dotProduct (const Vector4D &a, const Vector4D &b)

  Returns the dot product between two 4D vectors.
- float length (const Vector4D &v)

  Returns the length(magnitude) of the 4D vector v.
- Vector4D norm (const Vector4D &v)

  Normalizes the 4D vector v. If the 4D vector is the zero vector v is returned.
- Vector4D Projection (const Vector4D &a, const Vector4D &b)

  Returns a 4D vector that is the projection of a onto b. If b is the zero vector a is returned.
- Matrix4x4 operator+ (const Matrix4x4 &m1, const Matrix4x4 &m2)

  Adds the two given 4x4 matrices and returns a Matrix4x4 object with the result.
- Matrix4x4 operator- (const Matrix4x4 &m)

  Negates the 4x4 matrix m.
- Matrix4x4 operator- (const Matrix4x4 &m1, const Matrix4x4 &m2)

  Subtracts the two given 4x4 matrices and returns a Matrix4x4 object with the result.
- Matrix4x4 operator∗ (const Matrix4x4 &m, const float &k)

  Multiplies the given 4x4 matrix with the given scalar and returns a Matrix4x4 object with the result.
- Matrix4x4 operator∗ (const float &k, const Matrix4x4 &m)

  Multiplies the the given scalar with the given 4x4 matrix and returns a Matrix4x4 object with the result.
- Matrix4x4 operator∗ (const Matrix4x4 &m1, const Matrix4x4 &m2)

  Multiplies the two given 4x4 matrices and returns a Matrix4x4 object with the result.
- Vector4D operator∗ (const Matrix4x4 &m, const Vector4D &v)

  Multiplies the given 4x4 matrix with the given 4D vector and returns a Vector4D object with the result. The vector is a column vector.
- Vector4D operator∗ (const Vector4D &v, const Matrix4x4 &m)

  Multiplies the given 4D vector with the given 4x4 matrix and returns a Vector4D object with the result. The vector is a row vector.
- void setToIdentity (Matrix4x4 &m)

  Sets the given matrix to the identity matrix.
- bool isIdentity (const Matrix4x4 &m)

  Returns true if the given matrix is the identity matrix, false otherwise.
- Matrix4x4 transpose (const Matrix4x4 &m)

  Returns the tranpose of the given matrix m.

- Matrix4x4 translate (const Matrix4x4 &cm, float x, float y, float z)

  Construct a 4x4 translation matrix with the given floats and post-multiply's it by the given matrix. cm = cm ∗ translate.
- Matrix4x4 scale (const Matrix4x4 &cm, float x, float y, float z)

  Construct a 4x4 scaling matrix with the given floats and post-multiply's it by the given matrix. cm = cm ∗ scale.
- Matrix4x4 rotate (const Matrix4x4 &cm, float angle, float x, float y, float z)

  Construct a 4x4 rotation matrix with the given angle (in degrees) and axis (x, y, z) and post-multiply's it by the given matrix. cm = cm ∗ rotate.

  .
- double det (const Matrix4x4 &m)

  Returns the determinant of the given matrix.
- double cofactor (const Matrix4x4 &m, unsigned int row, unsigned int col)

  Returns the cofactor of the given row and col using the given matrix.
- Matrix4x4 adjoint (const Matrix4x4 &m)

  Returns the adjoint of the given matrix.
- Matrix4x4 inverse (const Matrix4x4 &m)

  Returns the inverse of the given matrix. If the matrix is noninvertible/singular, the identity matrix is returned.
- Quaternion operator+ (const Quaternion &q1, const Quaternion &q2)

  Returns a quaternion that has the result of q1 + q2.
- Quaternion operator- (const Quaternion &q)

  Returns a quaternion that has the result of -q.
- Quaternion operator- (const Quaternion &q1, const Quaternion &q2)

  Returns a quaternion that has the result of q1 - q2.
- Quaternion operator∗ (float k, const Quaternion &q)

  Returns a quaternion that has the result of k ∗ q.
- Quaternion operator∗ (const Quaternion &q, float k)

  Returns a quaternion that has the result of q ∗ k.
- Quaternion operator∗ (const Quaternion &q1, const Quaternion &q2)

  Returns a quaternion that has the result of q1 ∗ q2.
- bool isZeroQuaternion (const Quaternion &q)

  Returns true if quaternion q is a zero quaternion, false otherwise.
- bool isIdentity (const Quaternion &q)

  Returns true if quaternion q is an identity quaternion, false otherwise.
- Quaternion conjugate (const Quaternion &q)

  Returns the conjugate of quaternion q.
- float length (const Quaternion &q)

  Returns the length of quaternion q.
- Quaternion normalize (const Quaternion &q)

  Normalizes quaternion q and returns the normalized quaternion. If q is the zero quaternion then q is returned.
- Quaternion inverse (const Quaternion &q)

  Returns the invese of quaternion q. If q is the zero quaternion then q is returned.
- Quaternion rotationQuaternion (float angle, float x, float y, float z)

  Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.
- Quaternion rotationQuaternion (float angle, const Vector3D &axis)

  Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.
- Quaternion rotationQuaternion (const Vector4D &angAxis)

  Returns a quaternion from the axis-angle rotation representation. The x value in the 4D vector should be the angle(in degrees).

  The y, z and w value in the 4D vector should be the axis.
- Matrix4x4 quaternionRotationMatrixCol (const Quaternion &q)

  Returns a matrix from the given quaterion for column vector-matrix multiplication. Quaternion q should be a unit quaternion.
- Matrix4x4 quaternionRotationMatrixRow (const Quaternion &q)

  Returns a matrix from the given quaterion for row vector-matrix multiplication. Quaternion q should be a unit quaternion.

4.1.1 Detailed Description

Has utility functions, Vector2D, Vector3D, Vector4D, Matrix4x4, and Quaternion classes.

4.1.2 Function Documentation

4.1.2.1 adjoint()

```
Matrix4x4 FAMath::adjoint (
            const Matrix4x4 & m )  [inline]
```

Returns the adjoint of the given matrix.

4.1.2.2 CartesianToCylindrical()

```
Vector3D FAMath::CartesianToCylindrical (
            const Vector3D & v )  [inline]
```

Converts the 3D vector v from cartesian coordinates to cylindrical coordinates. v should = (x, y, z).
If vx is zero then no conversion happens and v is returned.
The returned 3D vector = (r, theta(degrees), z).

4.1.2.3 CartesianToPolar()

```
Vector2D FAMath::CartesianToPolar (
            const Vector2D & v )  [inline]
```

Converts the 2D vector v from cartesian coordinates to polar coordinates. v should = (x, y, z) If vx is zero then no conversion happens and v is returned.
The returned 2D vector = (r, theta(degrees)).

4.1.2.4 CartesianToSpherical()

```
Vector3D FAMath::CartesianToSpherical (
            const Vector3D & v )  [inline]
```

Converts the 3D vector v from cartesian coordinates to spherical coordinates. If v is the zero vector or if vx is zero then no conversion happens and v is returned.
The returned 3D vector = (r, phi(degrees), theta(degrees)).

4.1.2.5 cofactor()

```
double FAMath::cofactor (
            const Matrix4x4 & m,
            unsigned int row,
            unsigned int col ) [inline]
```

Returns the cofactor of the given row and col using the given matrix.

4.1.2.6 conjugate()

```
Quaternion FAMath::conjugate (
            const Quaternion & q ) [inline]
```

Returns the conjugate of quaternion q.

4.1.2.7 crossProduct()

```
Vector3D FAMath::crossProduct (
            const Vector3D & a,
            const Vector3D & b ) [inline]
```

Returns the cross product between two 3D vectors.

4.1.2.8 CylindricalToCartesian()

```
Vector3D FAMath::CylindricalToCartesian (
            const Vector3D & v ) [inline]
```

Converts the 3D vector v from cylindrical coordinates to cartesian coordinates. v should = (r, theta(degrees), z). The returned 3D vector = (x, y ,z).

4.1.2.9 det()

```
double FAMath::det (
            const Matrix4x4 & m ) [inline]
```

Returns the determinant of the given matrix.

4.1.2.10 dotProduct() [1/3]

```
float FAMath::dotProduct (
            const Vector2D & a,
            const Vector2D & b )  [inline]
```

Returns the dot product between two 2D vectors.

4.1.2.11 dotProduct() [2/3]

```
float FAMath::dotProduct (
            const Vector3D & a,
            const Vector3D & b )  [inline]
```

Returns the dot product between two 3D vectors.

4.1.2.12 dotProduct() [3/3]

```
float FAMath::dotProduct (
            const Vector4D & a,
            const Vector4D & b )  [inline]
```

Returns the dot product between two 4D vectors.

4.1.2.13 inverse() [1/2]

```
Matrix4x4 FAMath::inverse (
            const Matrix4x4 & m )  [inline]
```

Returns the inverse of the given matrix. If the matrix is noninvertible/singular, the identity matrix is returned.

4.1.2.14 inverse() [2/2]

```
Quaternion FAMath::inverse (
            const Quaternion & q )  [inline]
```

Returns the invese of quaternion q. If q is the zero quaternion then q is returned.

4.1.2.15 isIdentity() [1/2]

```
bool FAMath::isIdentity (
            const Matrix4x4 & m )   [inline]
```

Returns true if the given matrix is the identity matrix, false otherwise.

4.1.2.16 isIdentity() [2/2]

```
bool FAMath::isIdentity (
            const Quaternion & q )   [inline]
```

Returns true if quaternion q is an identity quaternion, false otherwise.

4.1.2.17 isZeroQuaternion()

```
bool FAMath::isZeroQuaternion (
            const Quaternion & q )   [inline]
```

Returns true if quaternion q is a zero quaternion, false otherwise.

4.1.2.18 length() [1/4]

```
float FAMath::length (
            const Quaternion & q )   [inline]
```

Returns the length of quaternion q.

4.1.2.19 length() [2/4]

```
float FAMath::length (
            const Vector2D & v )   [inline]
```

Returns the length(magnitude) of the 2D vector v.

4.1.2.20 length() [3/4]

```
float FAMath::length (
            const Vector3D & v )   [inline]
```

Returns the length(magnitude) of the 3D vector v.

4.1.2.21 length() [4/4]

```
float FAMath::length (
            const Vector4D & v )  [inline]
```

Returns the length(magnitude) of the 4D vector v.

4.1.2.22 norm() [1/3]

```
Vector2D FAMath::norm (
            const Vector2D & v )  [inline]
```

Normalizes the 2D vector v. If the 2D vector is the zero vector v is returned.

4.1.2.23 norm() [2/3]

```
Vector3D FAMath::norm (
            const Vector3D & v )  [inline]
```

Normalizes the 3D vector v. If the 3D vector is the zero vector v is returned.

4.1.2.24 norm() [3/3]

```
Vector4D FAMath::norm (
            const Vector4D & v )  [inline]
```

Normalizes the 4D vector v. If the 4D vector is the zero vector v is returned.

4.1.2.25 normalize()

```
Quaternion FAMath::normalize (
            const Quaternion & q )  [inline]
```

Normalizes quaternion q and returns the normalized quaternion. If q is the zero quaternion then q is returned.

4.1.2.26 operator*() [1/14]

```
Matrix4x4 FAMath::operator* (
            const float & k,
            const Matrix4x4 & m )  [inline]
```

Multiplies the the given scalar with the given 4x4 matrix and returns a Matrix4x4 object with the result.

4.1.2.27 operator*() [2/14]

```
Vector2D FAMath::operator* (
            const float & k,
            const Vector2D & a )  [inline]
```

2D vector scalar multiplication. Returns k $*$ a, where a is a vector and k is a scalar(float)

4.1.2.28 operator*() [3/14]

```
Vector3D FAMath::operator* (
            const float & k,
            const Vector3D & a )  [inline]
```

3D vector scalar multiplication. Returns k $*$ a, where a is a vector and k is a scalar(float)

4.1.2.29 operator*() [4/14]

```
Vector4D FAMath::operator* (
            const float & k,
            const Vector4D & a )  [inline]
```

4D vector scalar multiplication. Returns k $*$ a, where a is a vector and k is a scalar(float)

4.1.2.30 operator*() [5/14]

```
Matrix4x4 FAMath::operator* (
            const Matrix4x4 & m,
            const float & k )  [inline]
```

Multiplies the given 4x4 matrix with the given scalar and returns a Matrix4x4 object with the result.

4.1.2.31 operator*() [6/14]

```
Vector4D FAMath::operator* (
            const Matrix4x4 & m,
            const Vector4D & v ) [inline]
```

Multiplies the given 4x4 matrix with the given 4D vector and returns a Vector4D object with the result. The vector is a column vector.

4.1.2.32 operator*() [7/14]

```
Matrix4x4 FAMath::operator* (
            const Matrix4x4 & m1,
            const Matrix4x4 & m2 ) [inline]
```

Multiplies the two given 4x4 matrices and returns a Matrix4x4 object with the result.

4.1.2.33 operator*() [8/14]

```
Quaternion FAMath::operator* (
            const Quaternion & q,
            float k ) [inline]
```

Returns a quaternion that has the result of q * k.

4.1.2.34 operator*() [9/14]

```
Quaternion FAMath::operator* (
            const Quaternion & q1,
            const Quaternion & q2 ) [inline]
```

Returns a quaternion that has the result of q1 * q2.

4.1.2.35 operator*() [10/14]

```
Vector2D FAMath::operator* (
            const Vector2D & a,
            const float & k ) [inline]
```

2D vector scalar multiplication. Returns a * k, where a is a vector and k is a scalar(float)

4.1.2.36 operator*() [11/14]

```
Vector3D FAMath::operator* (
        const Vector3D & a,
        const float & k ) [inline]
```

3D vector scalar multiplication. Returns a ∗ k, where a is a vector and k is a scalar(float)

4.1.2.37 operator*() [12/14]

```
Vector4D FAMath::operator* (
        const Vector4D & a,
        const float & k ) [inline]
```

4D vector scalar multiplication. Returns a ∗ k, where a is a vector and k is a scalar(float)

4.1.2.38 operator*() [13/14]

```
Vector4D FAMath::operator* (
        const Vector4D & v,
        const Matrix4x4 & m ) [inline]
```

Multiplies the given 4D vector with the given 4x4 matrix and returns a Vector4D object with the result. The vector is a row vector.

4.1.2.39 operator*() [14/14]

```
Quaternion FAMath::operator* (
        float k,
        const Quaternion & q ) [inline]
```

Returns a quaternion that has the result of k ∗ q.

4.1.2.40 operator+() [1/5]

```
Matrix4x4 FAMath::operator+ (
        const Matrix4x4 & m1,
        const Matrix4x4 & m2 ) [inline]
```

Adds the two given 4x4 matrices and returns a Matrix4x4 object with the result.

4.1.2.41 operator+() [2/5]

```
Quaternion FAMath::operator+ (
            const Quaternion & q1,
            const Quaternion & q2 )  [inline]
```

Returns a quaternion that has the result of q1 + q2.

4.1.2.42 operator+() [3/5]

```
Vector2D FAMath::operator+ (
            const Vector2D & a,
            const Vector2D & b )  [inline]
```

2D vector addition.

4.1.2.43 operator+() [4/5]

```
Vector3D FAMath::operator+ (
            const Vector3D & a,
            const Vector3D & b )  [inline]
```

3D vector addition.

4.1.2.44 operator+() [5/5]

```
Vector4D FAMath::operator+ (
            const Vector4D & a,
            const Vector4D & b )  [inline]
```

4D vector addition.

4.1.2.45 operator-() [1/10]

```
Matrix4x4 FAMath::operator- (
            const Matrix4x4 & m )  [inline]
```

Negates the 4x4 matrix m.

4.1.2.46 operator-() [2/10]

```
Matrix4x4 FAMath::operator- (
            const Matrix4x4 & m1,
            const Matrix4x4 & m2 )  [inline]
```

Subtracts the two given 4x4 matrices and returns a Matrix4x4 object with the result.

4.1.2.47 operator-() [3/10]

```
Quaternion FAMath::operator- (
            const Quaternion & q )  [inline]
```

Returns a quaternion that has the result of -q.

4.1.2.48 operator-() [4/10]

```
Quaternion FAMath::operator- (
            const Quaternion & q1,
            const Quaternion & q2 )  [inline]
```

Returns a quaternion that has the result of q1 - q2.

4.1.2.49 operator-() [5/10]

```
Vector2D FAMath::operator- (
            const Vector2D & a,
            const Vector2D & b )  [inline]
```

2D vector subtraction.

4.1.2.50 operator-() [6/10]

```
Vector2D FAMath::operator- (
            const Vector2D & v )  [inline]
```

2D vector negation.

4.1.2.51 operator-() [7/10]

```
Vector3D FAMath::operator- (
        const Vector3D & a,
        const Vector3D & b ) [inline]
```

3D vector subtraction.

4.1.2.52 operator-() [8/10]

```
Vector3D FAMath::operator- (
        const Vector3D & v ) [inline]
```

3D vector negeation.

4.1.2.53 operator-() [9/10]

```
Vector4D FAMath::operator- (
        const Vector4D & a,
        const Vector4D & b ) [inline]
```

4D vector subtraction.

4.1.2.54 operator-() [10/10]

```
Vector4D FAMath::operator- (
        const Vector4D & v ) [inline]
```

4D vector negation.

4.1.2.55 operator/() [1/3]

```
Vector2D FAMath::operator/ (
        const Vector2D & a,
        const float & k ) [inline]
```

2D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.

4.1.2.56 operator/() [2/3]

```
Vector3D FAMath::operator/ (
            const Vector3D & a,
            const float & k )  [inline]
```

3D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.

4.1.2.57 operator/() [3/3]

```
Vector4D FAMath::operator/ (
            const Vector4D & a,
            const float & k )  [inline]
```

4D vector scalar division. Returns a / k, where a is a vector and k is a scalar(float) If k = 0 the returned vector is the zero vector.

4.1.2.58 PolarToCartesian()

```
Vector2D FAMath::PolarToCartesian (
            const Vector2D & v )  [inline]
```

Converts the 2D vector v from polar coordinates to cartesian coordinates. v should = (r, theta(degrees)) The returned 2D vector = (x, y)

4.1.2.59 Projection() [1/3]

```
Vector2D FAMath::Projection (
            const Vector2D & a,
            const Vector2D & b )  [inline]
```

Returns a 2D vector that is the projection of a onto b. If b is the zero vector a is returned.

4.1.2.60 Projection() [2/3]

```
Vector3D FAMath::Projection (
            const Vector3D & a,
            const Vector3D & b )  [inline]
```

Returns a 3D vector that is the projection of a onto b. If b is the zero vector a is returned.

4.1.2.61 Projection() [3/3]

```
Vector4D FAMath::Projection (
            const Vector4D & a,
            const Vector4D & b ) [inline]
```

Returns a 4D vector that is the projection of a onto b. If b is the zero vector a is returned.

4.1.2.62 quaternionRotationMatrixCol()

```
Matrix4x4 FAMath::quaternionRotationMatrixCol (
            const Quaternion & q ) [inline]
```

Returns a matrix from the given quaterion for column vector-matrix multiplication. Quaternion q should be a unit quaternion.

4.1.2.63 quaternionRotationMatrixRow()

```
Matrix4x4 FAMath::quaternionRotationMatrixRow (
            const Quaternion & q ) [inline]
```

Returns a matrix from the given quaterion for row vector-matrix multiplication. Quaternion q should be a unit quaternion.

4.1.2.64 rotate()

```
Matrix4x4 FAMath::rotate (
            const Matrix4x4 & cm,
            float angle,
            float x,
            float y,
            float z ) [inline]
```

Construct a 4x4 rotation matrix with the given angle (in degrees) and axis (x, y, z) and post-multiply's it by the given matrix. cm = cm ∗ rotate.

.

4.1.2.65 rotationQuaternion() [1/3]

```
Quaternion FAMath::rotationQuaternion (
            const Vector4D & angAxis ) [inline]
```

Returns a quaternion from the axis-angle rotation representation. The x value in the 4D vector should be the angle(in degrees).

The y, z and w value in the 4D vector should be the axis.

4.1.2.66 rotationQuaternion() [2/3]

```
Quaternion FAMath::rotationQuaternion (
            float angle,
            const Vector3D & axis ) [inline]
```

Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.

4.1.2.67 rotationQuaternion() [3/3]

```
Quaternion FAMath::rotationQuaternion (
            float angle,
            float x,
            float y,
            float z ) [inline]
```

Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.

4.1.2.68 scale()

```
Matrix4x4 FAMath::scale (
            const Matrix4x4 & cm,
            float x,
            float y,
            float z ) [inline]
```

Construct a 4x4 scaling matrix with the given floats and post-multiply's it by the given matrix. cm = cm $*$ scale.

4.1.2.69 setToIdentity()

```
void FAMath::setToIdentity (
            Matrix4x4 & m ) [inline]
```

Sets the given matrix to the identity matrix.

4.1.2.70 SphericalToCartesian()

```
Vector3D FAMath::SphericalToCartesian (
            const Vector3D & v ) [inline]
```

Converts the 3D vector v from spherical coordinates to cartesian coordinates. v should = (pho, phi(degrees), theta(degrees)).
The returned 3D vector = (x, y, z)

4.1.2.71 translate()

```
Matrix4x4 FAMath::translate (
              const Matrix4x4 & cm,
              float x,
              float y,
              float z )  [inline]
```

Construct a 4x4 translation matrix with the given floats and post-multiply's it by the given matrix. cm = cm ∗ translate.

4.1.2.72 transpose()

```
Matrix4x4 FAMath::transpose (
              const Matrix4x4 & m )  [inline]
```

Returns the tranpose of the given matrix m.

4.1.2.73 zeroVector() [1/3]

```
bool FAMath::zeroVector (
              const Vector2D & a )  [inline]
```

Returns true if a is the zero vector.

4.1.2.74 zeroVector() [2/3]

```
bool FAMath::zeroVector (
              const Vector3D & a )  [inline]
```

Returns true if a is the zero vector.

4.1.2.75 zeroVector() [3/3]

```
bool FAMath::zeroVector (
              const Vector4D & a )  [inline]
```

Returns true if a is the zero vector.

Chapter 5

Class Documentation

5.1 FAMath::Matrix4x4 Class Reference

A matrix class used for 4x4 matrices and their manipulations.

```
#include "FAMathEngine.h"
```

Public Member Functions

- Matrix4x4 ()

  Default Constructor.
- Matrix4x4 (float a[ ][4])

  Overloaded Constructor.
- float ∗ data ()

  Returns a pointer to the first element in the matrix.
- const float ∗ data () const

  Returns a constant pointer to the first element in the matrix.
- const float & operator() (unsigned int row, unsigned int col) const

  Returns a constant reference to the element at the given (row, col). The row and col values should be between [0,3]. If any of them are out of that range, the first element will be returned.
- float & operator() (unsigned int row, unsigned int col)

  Returns a reference to the element at the given (row, col). The row and col values should be between [0,3]. If any of them are out of that range, the first element will be returned.
- void setRow (unsigned int row, Vector4D v)

  Sets each element in the given row to the components of vector v. Row should be between [0,3]. If it out of range the first row will be set.
- void setCol (unsigned int col, Vector4D v)

  Sets each element in the given col to the components of vector v. Col should be between [0,3]. If it out of range the first col will be set.
- Matrix4x4 & operator+= (const Matrix4x4 &m)

  Adds this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.
- Matrix4x4 & operator-= (const Matrix4x4 &m)

  Subtracts this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.
- Matrix4x4 & operator∗= (const float &k)

  Multiplies this 4x4 matrix with given scalar k and stores the result in this 4x4 matrix.
- Matrix4x4 & operator∗= (const Matrix4x4 &m)

  Multiplies this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.

5.1.1 Detailed Description

A matrix class used for 4x4 matrices and their manipulations.

The datatype for the components is float.
The 4x4 matrix is treated as a row-major matrix.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Matrix4x4() [1/2]

```
FAMath::Matrix4x4::Matrix4x4 ( )  [inline]
```

Default Constructor.

Creates a new 4x4 identity matrix.

5.1.2.2 Matrix4x4() [2/2]

```
FAMath::Matrix4x4::Matrix4x4 (
            float a[ ][4] )  [inline]
```

Overloaded Constructor.

Creates a new 4x4 matrix with elements initialized to the given 2D array.
If the passed in 2D array isn't a 4x4 matrix, the behavior is undefined.

5.1.3 Member Function Documentation

5.1.3.1 data() [1/2]

```
float * FAMath::Matrix4x4::data ( )  [inline]
```

Returns a pointer to the first element in the matrix.

5.1.3.2 data() [2/2]

```
const float * FAMath::Matrix4x4::data ( ) const  [inline]
```

Returns a constant pointer to the first element in the matrix.

5.1.3.3 operator()() [1/2]

```
float & FAMath::Matrix4x4::operator() (
            unsigned int row,
            unsigned int col )  [inline]
```

Returns a reference to the element at the given (row, col). The row and col values should be between [0,3]. If any of them are out of that range, the first element will be returned.

5.1.3.4 operator()() [2/2]

```
const float & FAMath::Matrix4x4::operator() (
            unsigned int row,
            unsigned int col ) const  [inline]
```

Returns a constant reference to the element at the given (row, col). The row and col values should be between [0,3]. If any of them are out of that range, the first element will be returned.

5.1.3.5 operator*=() [1/2]

```
Matrix4x4 & FAMath::Matrix4x4::operator*= (
            const float & k )  [inline]
```

Multiplies this 4x4 matrix with given scalar k and stores the result in this 4x4 matrix.

5.1.3.6 operator*=() [2/2]

```
Matrix4x4 & FAMath::Matrix4x4::operator*= (
            const Matrix4x4 & m )  [inline]
```

Multiplies this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.

5.1.3.7 operator+=()

```
Matrix4x4 & FAMath::Matrix4x4::operator+= (
            const Matrix4x4 & m )  [inline]
```

Adds this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.

5.1.3.8 operator-=()

```
Matrix4x4 & FAMath::Matrix4x4::operator-= (
            const Matrix4x4 & m )  [inline]
```

Subtracts this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.

5.1.3.9 setCol()

```
void FAMath::Matrix4x4::setCol (
            unsigned int col,
            Vector4D v )  [inline]
```

Sets each element in the given col to the components of vector v. Col should be between [0,3]. If it out of range the first col will be set.

5.1.3.10 setRow()

```
void FAMath::Matrix4x4::setRow (
            unsigned int row,
            Vector4D v )  [inline]
```

Sets each element in the given row to the components of vector v. Row should be between [0,3]. If it out of range the first row will be set.

The documentation for this class was generated from the following file:

- FAMathEngine.h

5.2 FAMath::Quaternion Class Reference

```
#include "FAMathEngine.h"
```

Public Member Functions

- Quaternion ()

    Default Constructor.
- Quaternion (float scalar, float x, float y, float z)

    Overloaded Constructor.
- Quaternion (float scalar, const Vector3D &v)

    Overloaded Constructor.
- Quaternion (const Vector4D &v)

    Overloaded Constructor.
- float & scalar ()

    Returns a reference to the scalar component of the quaternion.
- const float & scalar () const

    Returns a const reference to the scalar component of the quaternion.
- float & x ()

    Returns a reference to the x value of the vector component in the quaternion.
- const float & x () const

    Returns a const reference to the x value of the vector component in the quaternion.
- float & y ()

    Returns a reference to the y value of the vector component in the quaternion.
- const float & y () const

    Returns a const reference to the y value of the vector component in the quaternion.
- float & z ()

    Returns a reference to the z value of the vector component in the quaternion.
- const float & z () const

    Returns a const reference to the z value of the vector component in the quaternion.
- Vector3D vector ()

    Returns the vector component of the quaternion.
- Quaternion & operator+= (const Quaternion &q)

    Adds this quaternion to quaterion q and stores the result in this quaternion.
- Quaternion & operator-= (const Quaternion &q)

    Subtracts this quaternion by quaterion q and stores the result in this quaternion.
- Quaternion & operator*= (float k)

    Multiplies this quaternion by flaot k and stores the result in this quaternion.
- Quaternion & operator*= (const Quaternion &q)

    Multiplies this quaternion by quaterion q and stores the result in this quaternion.

5.2.1 Detailed Description

The datatype for the components is float.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Quaternion() [1/4]

```
FAMath::Quaternion::Quaternion ( ) [inline]
```

Default Constructor.

Constructs an identity quaternion.

5.2.2.2 Quaternion() [2/4]

```
FAMath::Quaternion::Quaternion (
            float scalar,
            float x,
            float y,
            float z ) [inline]
```

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.3 Quaternion() [3/4]

```
FAMath::Quaternion::Quaternion (
            float scalar,
            const Vector3D & v ) [inline]
```

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.4 Quaternion() [4/4]

```
FAMath::Quaternion::Quaternion (
            const Vector4D & v ) [inline]
```

Overloaded Constructor.

Constructs a quaternion with the given values in the 4D vector.
The x value in the 4D vector should be the scalar. The y, z and w value in the 4D vector should be the axis.

5.2.3 Member Function Documentation

5.2.3.1 operator*=() [1/2]

```
Quaternion & FAMath::Quaternion::operator*= (
            const Quaternion & q ) [inline]
```

Multiplies this quaternion by quaterion q and stores the result in this quaternion.

5.2.3.2 operator*=() [2/2]

```
Quaternion & FAMath::Quaternion::operator*= (
             float k )  [inline]
```

Multiplies this quaternion by flaot k and stores the result in this quaternion.

5.2.3.3 operator+=()

```
Quaternion & FAMath::Quaternion::operator+= (
             const Quaternion & q )  [inline]
```

Adds this quaternion to quaterion q and stores the result in this quaternion.

5.2.3.4 operator-=()

```
Quaternion & FAMath::Quaternion::operator-= (
             const Quaternion & q )  [inline]
```

Subtracts this quaternion by quaterion q and stores the result in this quaternion.

5.2.3.5 scalar() [1/2]

```
float & FAMath::Quaternion::scalar ( )  [inline]
```

Returns a reference to the scalar component of the quaternion.

5.2.3.6 scalar() [2/2]

```
const float & FAMath::Quaternion::scalar ( ) const  [inline]
```

Returns a const reference to the scalar component of the quaternion.

5.2.3.7 vector()

```
Vector3D FAMath::Quaternion::vector ( )  [inline]
```

Returns the vector component of the quaternion.

5.2.3.8 x() [1/2]

```
float & FAMath::Quaternion::x ( ) [inline]
```

Returns a reference to the x value of the vector component in the quaternion.

5.2.3.9 x() [2/2]

```
const float & FAMath::Quaternion::x ( ) const [inline]
```

Returns a const reference to the x value of the vector component in the quaternion.

5.2.3.10 y() [1/2]

```
float & FAMath::Quaternion::y ( ) [inline]
```

Returns a reference to the y value of the vector component in the quaternion.

5.2.3.11 y() [2/2]

```
const float & FAMath::Quaternion::y ( ) const [inline]
```

Returns a const reference to the y value of the vector component in the quaternion.

5.2.3.12 z() [1/2]

```
float & FAMath::Quaternion::z ( ) [inline]
```

Returns a reference to the z value of the vector component in the quaternion.

5.2.3.13 z() [2/2]

```
const float & FAMath::Quaternion::z ( ) const [inline]
```

Returns a const reference to the z value of the vector component in the quaternion.

The documentation for this class was generated from the following file:

- FAMathEngine.h

5.3 FAMath::Vector2D Class Reference

A vector class used for 2D vectors/points and their manipulations.

```
#include "FAMathEngine.h"
```

Public Member Functions

- Vector2D ()

    Default Constructor.
- Vector2D (float x, float y)

    Overloaded Constructor.
- float & x ()

    Returns a reference to the x component.
- float & y ()

    Returns a reference to the y component.
- const float & x () const

    Returns a constant reference to the x component.
- const float & y () const

    Returns a constant reference to the y component.
- Vector2D & operator+= (const Vector2D &b)

    2D vector addition through overloading operator +=.
- Vector2D & operator-= (const Vector2D &b)

    2D vector subtraction through overloading operator -=.
- Vector2D & operator*= (const float &k)

    *2D vector scalar multiplication through overloading operator *=.*
- Vector2D & operator/= (const float &k)

    2D vector scalar division through overloading operator /=.

5.3.1 Detailed Description

A vector class used for 2D vectors/points and their manipulations.

The datatype for the components is float.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Vector2D() [1/2]

```
FAMath::Vector2D::Vector2D ( )  [inline]
```

Default Constructor.

Creates a new 2D vector/point with the components initialized to 0.0.

5.3.2.2 Vector2D() [2/2]

```
FAMath::Vector2D::Vector2D (
            float x,
            float y )  [inline]
```

Overloaded Constructor.

Creates a new 2D vector/point with the components initialized to the arguments.

5.3.3 Member Function Documentation

5.3.3.1 operator*=()

```
Vector2D & FAMath::Vector2D::operator*= (
            const float & k )  [inline]
```

2D vector scalar multiplication through overloading operator *=.

5.3.3.2 operator+=()

```
Vector2D & FAMath::Vector2D::operator+= (
            const Vector2D & b )  [inline]
```

2D vector addition through overloading operator +=.

5.3.3.3 operator-=()

```
Vector2D & FAMath::Vector2D::operator-= (
            const Vector2D & b )  [inline]
```

2D vector subtraction through overloading operator -=.

5.3.3.4 operator/=()

```
Vector2D & FAMath::Vector2D::operator/= (
            const float & k )  [inline]
```

2D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.3.3.5 x() [1/2]

```
float & FAMath::Vector2D::x ( )  [inline]
```

Returns a reference to the x component.

5.3.3.6 x() [2/2]

```
const float & FAMath::Vector2D::x ( ) const  [inline]
```

Returns a constant reference to the x component.

5.3.3.7 y() [1/2]

```
float & FAMath::Vector2D::y ( )  [inline]
```

Returns a reference to the y component.

5.3.3.8 y() [2/2]

```
const float & FAMath::Vector2D::y ( ) const  [inline]
```

Returns a constant reference to the y component.

The documentation for this class was generated from the following file:

- FAMathEngine.h

5.4 FAMath::Vector3D Class Reference

A vector class used for 3D vectors/points and their manipulations.

```
#include "FAMathEngine.h"
```

Public Member Functions

- Vector3D ()

  Default Constructor.
- Vector3D (float x, float y, float z)

  Overloaded Constructor.
- float & x ()

  Returns a reference to the x component.
- float & y ()

  Returns a reference to the y component.
- float & z ()

  Returns a reference to the z component.
- const float & x () const

  Returns a constant reference to the x component.
- const float & y () const

  Returns a constant reference to the y component.
- const float & z () const

  Returns a constant reference to the z component.
- Vector3D & operator+= (const Vector3D &b)

  3D vector addition through overloading operator +=.
- Vector3D & operator-= (const Vector3D &b)

  3D vector subtraction through overloading operator -=.
- Vector3D & operator*= (const float &k)

  *3D vector scalar multiplication through overloading operator *=.*
- Vector3D & operator/= (const float &k)

  3D vector scalar division through overloading operator /=.

5.4.1 Detailed Description

A vector class used for 3D vectors/points and their manipulations.

The datatype for the components is float.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Vector3D() [1/2]

```
FAMath::Vector3D::Vector3D ( )  [inline]
```

Default Constructor.

Creates a new 3D vector/point with the components initialized to 0.0.

5.4.2.2 Vector3D() [2/2]

```
FAMath::Vector3D::Vector3D (
            float x,
            float y,
            float z )  [inline]
```

Overloaded Constructor.

Creates a new 3D vector/point with the components initialized to the arguments.

5.4.3 Member Function Documentation

5.4.3.1 operator*=()

```
Vector3D & FAMath::Vector3D::operator*= (
            const float & k )  [inline]
```

3D vector scalar multiplication through overloading operator *=.

5.4.3.2 operator+=()

```
Vector3D & FAMath::Vector3D::operator+= (
            const Vector3D & b )  [inline]
```

3D vector addition through overloading operator +=.

5.4.3.3 operator-=()

```
Vector3D & FAMath::Vector3D::operator-= (
            const Vector3D & b )  [inline]
```

3D vector subtraction through overloading operator -=.

5.4.3.4 operator/=()

```
Vector3D & FAMath::Vector3D::operator/= (
            const float & k )  [inline]
```

3D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.4.3.5 x() [1/2]

```
float & FAMath::Vector3D::x ( )  [inline]
```

Returns a reference to the x component.

5.4.3.6 x() [2/2]

```
const float & FAMath::Vector3D::x ( ) const  [inline]
```

Returns a constant reference to the x component.

5.4.3.7 y() [1/2]

```
float & FAMath::Vector3D::y ( )  [inline]
```

Returns a reference to the y component.

5.4.3.8 y() [2/2]

```
const float & FAMath::Vector3D::y ( ) const  [inline]
```

Returns a constant reference to the y component.

5.4.3.9 z() [1/2]

```
float & FAMath::Vector3D::z ( )  [inline]
```

Returns a reference to the z component.

5.4.3.10 z() [2/2]

```
const float & FAMath::Vector3D::z ( ) const  [inline]
```

Returns a constant reference to the z component.

The documentation for this class was generated from the following file:

- FAMathEngine.h

5.5 FAMath::Vector4D Class Reference

A vector class used for 4D vectors/points and their manipulations.

```
#include "FAMathEngine.h"
```

Public Member Functions

- Vector4D ()

  Default Constructor.
- Vector4D (float x, float y, float z, float w)

  Overloaded Constructor.
- Vector4D (Vector2D v, float z=0.0f, float w=0.0f)

  Overloaded Constructor.
- Vector4D (Vector3D v, float w=0.0f)

  Overloaded Constructor.
- float & x ()

  Returns a reference to the x component.
- float & y ()

  Returns a reference to the y component.
- float & z ()

  Returns a reference to the z component.
- float & w ()

  Returns a reference to the w component.
- const float & x () const

  Returns a constant reference to the x component.
- const float & y () const

  Returns a constant reference to the y component.
- const float & z () const

  Returns a constant reference to the z component.
- const float & w () const

  Returns a constant reference to the w component.
- Vector4D & operator+= (const Vector4D &b)

  4D vector addition through overloading operator +=.
- Vector4D & operator-= (const Vector4D &b)

  4D vector subtraction through overloading operator -=.
- Vector4D & operator*= (const float &k)

  *4D vector scalar multiplication through overloading operator *=.*
- Vector4D & operator/= (const float &k)

  4D vector scalar division through overloading operator /=.

5.5.1 Detailed Description

A vector class used for 4D vectors/points and their manipulations.

The datatype for the components is float

5.5.2 Constructor & Destructor Documentation

5.5.2.1 Vector4D() [1/4]

```
FAMath::Vector4D::Vector4D ( )  [inline]
```

Default Constructor.

Creates a new 4D vector/point with the components initialized to 0.0.

5.5.2.2 Vector4D() [2/4]

```
FAMath::Vector4D::Vector4D (
            float x,
            float y,
            float z,
            float w )  [inline]
```

Overloaded Constructor.

Creates a new 4D vector/point with the components initialized to the arguments.

5.5.2.3 Vector4D() [3/4]

```
FAMath::Vector4D::Vector4D (
            Vector2D v,
            float z = 0.0f,
            float w = 0.0f )  [inline]
```

Overloaded Constructor.

Creates a new 4D vector/point with the components initialized to the arguments.

5.5.2.4 Vector4D() [4/4]

```
FAMath::Vector4D::Vector4D (
            Vector3D v,
            float w = 0.0f )  [inline]
```

Overloaded Constructor.

Creates a new 4D vector/point with the components initialized to the arguments.

5.5.3 Member Function Documentation

5.5.3.1 operator*=()

```
Vector4D & FAMath::Vector4D::operator*= (
              const float & k )  [inline]
```

4D vector scalar multiplication through overloading operator *=.

5.5.3.2 operator+=()

```
Vector4D & FAMath::Vector4D::operator+= (
              const Vector4D & b )  [inline]
```

4D vector addition through overloading operator +=.

5.5.3.3 operator-=()

```
Vector4D & FAMath::Vector4D::operator-= (
              const Vector4D & b )  [inline]
```

4D vector subtraction through overloading operator -=.

5.5.3.4 operator/=()

```
Vector4D & FAMath::Vector4D::operator/= (
              const float & k )  [inline]
```

4D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.5.3.5 w() [1/2]

```
float & FAMath::Vector4D::w ( )  [inline]
```

Returns a reference to the w component.

5.5.3.6 w() [2/2]

```
const float & FAMath::Vector4D::w ( ) const  [inline]
```

Returns a constant reference to the w component.

5.5.3.7 x() [1/2]

`float & FAMath::Vector4D::x ( ) [inline]`

Returns a reference to the x component.

5.5.3.8 x() [2/2]

`const float & FAMath::Vector4D::x ( ) const [inline]`

Returns a constant reference to the x component.

5.5.3.9 y() [1/2]

`float & FAMath::Vector4D::y ( ) [inline]`

Returns a reference to the y component.

5.5.3.10 y() [2/2]

`const float & FAMath::Vector4D::y ( ) const [inline]`

Returns a constant reference to the y component.

5.5.3.11 z() [1/2]

`float & FAMath::Vector4D::z ( ) [inline]`

Returns a reference to the z component.

5.5.3.12 z() [2/2]

`const float & FAMath::Vector4D::z ( ) const [inline]`

Returns a constant reference to the z component.

The documentation for this class was generated from the following file:

- FAMathEngine.h

Chapter 6

File Documentation

6.1 FAMathEngine.h

```
#pragma once

#include <cmath>

#if defined(_DEBUG)
#include <iostream>
#endif


#define EPSILON 1e-6f
#define PI 3.14159265

namespace FAMath
{
    class Vector2D;
    class Vector3D;
    class Vector4D;

    /*@brief Checks if the two specified floats are equal using exact epsilion and adaptive epsilion.
*/
    inline bool compareFloats(float x, float y, float epsilon)
    {
        float diff = fabs(x - y);
        //exact epsilon
        if (diff < epsilon)
        {
            return true;
        }

        //adapative epsilon
        return diff <= epsilon * (((fabs(x)) >(fabs(y))) ?  (fabs(x)) :  (fabs(y)));
    }

    /*@brief Checks if the two specified doubles are equal using exact epsilion and adaptive epsilion.
*/
    inline bool compareDoubles(double x, double y, double epsilon)
    {
        double diff = fabs(x - y);
        //exact epsilon
        if (diff < epsilon)
        {
            return true;
        }

        //adapative epsilon
        return  diff <= epsilon * (((fabs(x)) >(fabs(y))) ?  (fabs(x)) :  (fabs(y)));
    }


   //------------------------------------------------------------------------------------------------------------------------

    class Vector2D
    {
    public:


        Vector2D();

```

```
        Vector2D(float x, float y);

        float& x();

        float& y();

        const float& x() const;

        const float& y() const;

        Vector2D& operator+=(const Vector2D& b);

        Vector2D& operator-=(const Vector2D& b);

        Vector2D& operator*=(const float& k);

        Vector2D& operator/=(const float& k);

    private:
        float m_x;
        float m_y;
    };


    //------------------------------------------------------------------------------------
    //Vector2D Constructors

    inline Vector2D::Vector2D() :  m_x{ 0.0f }, m_y{ 0.0f }
    {}

    inline Vector2D::Vector2D(float x, float y) :  m_x{ x }, m_y{ y }
    {}

    //------------------------------------------------------------------------------------

    //------------------------------------------------------------------------------------
    //Vector2D Getters and Setters

    inline float& Vector2D::x()
    {
        return m_x;
    }

    inline float& Vector2D::y()
    {
        return m_y;
    }

    inline const float& Vector2D::x()const
{
        return m_x;
    }

    inline const float& Vector2D::y()const
{
        return m_y;
    }

    //------------------------------------------------------------------------------------


    //------------------------------------------------------------------------------------
    //Vector2D Memeber functions

    inline Vector2D& Vector2D::operator+=(const Vector2D& b)
    {
        this->m_x += (double)b.m_x;
        this->m_y += (double)b.m_y;

        return *this;
    }

    inline Vector2D& Vector2D::operator-=(const Vector2D& b)
    {
        this->m_x -= (double)b.m_x;
        this->m_y -= (double)b.m_y;

        return *this;
    }

    inline Vector2D& Vector2D::operator*=(const float& k)
    {
        this->m_x *= (double)k;
        this->m_y *= (double)k;

        return *this;
    }
```

```

    inline Vector2D& Vector2D::operator/=(const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return *this;
        }

        this->m_x /= (double)k;
        this->m_y /= (double)k;

        return *this;
    }

    //--------------------------------------------------------------------------------------

    //--------------------------------------------------------------------------------------
    //Vector2D Non-member functions

    inline bool zeroVector(const Vector2D& a)
    {
        if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
    {
        return Vector2D((double)a.x() + b.x(), (double)a.y() + b.y());
    }

    inline Vector2D operator-(const Vector2D& v)
    {
        return Vector2D(-v.x(), -v.y());
    }

    inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
    {
        return Vector2D((double)a.x() - b.x(), (double)a.y() - b.y());
    }

    inline Vector2D operator*(const Vector2D& a, const float& k)
    {
        return Vector2D((double)a.x() * k, (double)a.y() * k);
    }

    inline Vector2D operator*(const float& k, const Vector2D& a)
    {
        return Vector2D((double)k * a.x(), (double)k * a.y());
    }

    inline Vector2D operator/(const Vector2D& a, const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return Vector2D();
        }

        return Vector2D(a.x() / (double)k, a.y() / (double)k);
    }

    inline float dotProduct(const Vector2D& a, const Vector2D& b)
    {
        return (double)a.x() * b.x() + (double)a.y() * b.y();
    }

    inline float length(const Vector2D& v)
    {
        return sqrt((double)v.x() * v.x() + (double)v.y() * v.y());
    }

    inline Vector2D norm(const Vector2D& v)
    {
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))

        //v is the zero vector
        if (zeroVector(v))
        {
            return v;
        }

        double mag = length(v);

        return Vector2D(v.x() / mag, v.y() / mag);
```

```
    }

    inline Vector2D PolarToCartesian(const Vector2D& v)
    {
        //v = (r, theta)
        //x = rcos((theta)
        //y = rsin(theta)
        float angle = v.y() * PI / 180.0f;

        return Vector2D(v.x() * cos(angle), v.x() * sin(angle));
    }

    inline Vector2D CartesianToPolar(const Vector2D& v)
    {
        //v = (x, y)
        //r = sqrt(vx^2 + vy^2)
        //theta = arctan(y / x)

        if (compareFloats(v.x(), 0.0f, EPSILON))
        {
            return v;
        }

        double theta{ atan2(v.y(), v.x()) * 180.0 / PI };
        return Vector2D(length(v), theta);
    }

    inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting

        Vector2D normB(norm(b));
        return Vector2D(dotProduct(a, normB) * normB);
    }


#if defined(_DEBUG)
    inline void print(const Vector2D& v)
    {
        std::cout « "(" « v.x() « ", " « v.y() « ")";
    }
#endif
    //-------------------------------------------------------------------------------------


  //-----------------------------------------------------------------------------------------------------------------------------




  //-----------------------------------------------------------------------------------------------------------------------------
    class Vector3D
    {
    public:

        Vector3D();

        Vector3D(float x, float y, float z);

        float& x();

        float& y();

        float& z();

        const float& x() const;

        const float& y() const;

        const float& z() const;

        Vector3D& operator+=(const Vector3D& b);

        Vector3D& operator-=(const Vector3D& b);

        Vector3D& operator*=(const float& k);

        Vector3D& operator/=(const float& k);

    private:
        float m_x;
        float m_y;
        float m_z;
    };

    //-------------------------------------------------------------------------------------
```

```
    //Vector3D Constructors

    inline Vector3D::Vector3D() :  m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }
    {}

    inline Vector3D::Vector3D(float x, float y, float z) :  m_x{ x }, m_y{ y }, m_z{ z }
    {}

    //--------------------------------------------------------------------------------------

    //--------------------------------------------------------------------------------------
    //Vector3D Getters and Setters

    inline float& Vector3D::x()
    {
        return m_x;
    }

    inline float& Vector3D::y()
    {
        return m_y;
    }

    inline float& Vector3D::z()
    {
        return m_z;
    }

    inline const float& Vector3D::x()const
{
        return m_x;
    }

    inline const float& Vector3D::y()const
{
        return m_y;
    }

    inline const float& Vector3D::z()const
{
        return m_z;
    }
    //--------------------------------------------------------------------------------------


    //--------------------------------------------------------------------------------------
    //Vector3D Memeber functions

    inline Vector3D& Vector3D::operator+=(const Vector3D& b)
    {
        this->m_x += (double)b.m_x;
        this->m_y += (double)b.m_y;
        this->m_z += (double)b.m_z;

        return *this;
    }

    inline Vector3D& Vector3D::operator-=(const Vector3D& b)
    {
        this->m_x -= (double)b.m_x;
        this->m_y -= (double)b.m_y;
        this->m_z -= (double)b.m_z;

        return *this;
    }

    inline Vector3D& Vector3D::operator*=(const float& k)
    {
        this->m_x *= (double)k;
        this->m_y *= (double)k;
        this->m_z *= (double)k;

        return *this;
    }

    inline Vector3D& Vector3D::operator/=(const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return *this;
        }

        this->m_x /= (double)k;
        this->m_y /= (double)k;
        this->m_z /= (double)k;

        return *this;
```

```
    }

    //-------------------------------------------------------------------------------------

    //-------------------------------------------------------------------------------------
    //Vector3D Non-member functions

    inline bool zeroVector(const Vector3D& a)
    {
        if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON) &&
            compareFloats(a.z(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
    {
        return Vector3D((double)a.x() + b.x(), (double)a.y() + b.y(), (double)a.z() + b.z());
    }

    inline Vector3D operator-(const Vector3D& v)
    {
        return Vector3D(-v.x(), -v.y(), -v.z());
    }

    inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
    {
        return Vector3D(a.x() - b.x(), a.y() - b.y(), a.z() - b.z());
    }

    inline Vector3D operator*(const Vector3D& a, const float& k)
    {
        return Vector3D(a.x() * (double)k, a.y() * (double)k, a.z() * (double)k);
    }

    inline Vector3D operator*(const float& k, const Vector3D& a)
    {
        return Vector3D((double)k * a.x(), (double)k * a.y(), (double)k * a.z());
    }

    inline Vector3D operator/(const Vector3D& a, const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return Vector3D();
        }

        return Vector3D(a.x() / (double)k, a.y() / (double)k, a.z() / (double)k);
    }

    inline float dotProduct(const Vector3D& a, const Vector3D& b)
    {
        //a dot b = axbx + ayby + azbz
        return (double)a.x() * b.x() + (double)a.y() * b.y() + (double)a.z() * b.z();
    }

    inline Vector3D crossProduct(const Vector3D& a, const Vector3D& b)
    {
        //a x b = (aybz - azby, azbx - axbz, axby - aybx)

        return Vector3D((double)a.y() * b.z() - (double)a.z() * b.y(),
            (double)a.z() * b.x() - (double)a.x() * b.z(),
            (double)a.x() * b.y() - (double)a.y() * b.x());
    }

    inline float length(const Vector3D& v)
    {
        //length(v) = sqrt(vx^2 + vy^2 + vz^2)

        return sqrt((double)v.x() * v.x() + (double)v.y() * v.y() + (double)v.z() * v.z());
    }

    inline Vector3D norm(const Vector3D& v)
    {
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
        //v is the zero vector
        if (zeroVector(v))
        {
            return v;
        }

        double mag = length(v);

        return Vector3D(v.x() / mag, v.y() / mag, v.z() / mag);
```

```
    }

    inline Vector3D CylindricalToCartesian(const Vector3D& v)
    {
        //v = (r, theta, z)
        //x = rcos(theta)
        //y = rsin(theta)
        //z = z
        double angle = v.y() * PI / 180.0;

        return Vector3D(v.x() * cos(angle), v.x() * sin(angle), v.z());
    }

    inline Vector3D CartesianToCylindrical(const Vector3D& v)
    {
        //v = (x, y, z)
        //r = sqrt(vx^2 + vy^2 + vz^2)
        //theta = arctan(y / x)
        //z = z
        if (compareFloats(v.x(), 0.0f, EPSILON))
        {
            return v;
        }

        double theta{ atan2(v.y(), v.x()) * 180.0 / PI };
        return Vector3D(length(v), theta, v.z());
    }

    inline Vector3D SphericalToCartesian(const Vector3D& v)
    {
        // v = (pho, phi, theta)
        //x = pho * sin(phi) * cos(theta)
        //y = pho * sin(phi) * sin(theta)
        //z = pho * cos(theta);

        double phi{ v.y() * PI / 180.0 };
        double theta{ v.z() * PI / 180.0 };

        return Vector3D(v.x() * sin(phi) * cos(theta), v.x() * sin(phi) * sin(theta), v.x() *
   cos(theta));
    }

    inline Vector3D CartesianToSpherical(const Vector3D& v)
    {
        //v = (x, y ,z)
        //pho = sqrt(vx^2 + vy^2 + vz^2)
        //phi = arcos(z / pho)
        //theta = arctan(y / x)

        if (compareFloats(v.x(), 0.0f, EPSILON) || zeroVector(v))
        {
            return v;
        }

        double pho{ length(v) };
        double phi{ acos(v.z() / pho) * 180.0 / PI };
        double theta{ atan2(v.y(), v.x()) * 180.0 / PI };

        return Vector3D(pho, phi, theta);
    }

    inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting

        Vector3D normB(norm(b));
        return Vector3D(dotProduct(a, normB) * normB);
    }


#if defined(_DEBUG)
    inline void print(const Vector3D& v)
    {
        std::cout « "(" « v.x() « ", " « v.y() « ", " « v.z() « ")";
    }
#endif
    //--------------------------------------------------------------------------------


   //-----------------------------------------------------------------------------------------------------------




   //-----------------------------------------------------------------------------------------------------------
    class Vector4D
```

```
    {
    public:
        Vector4D();

        Vector4D(float x, float y, float z, float w);

        Vector4D(Vector2D v, float z = 0.0f, float w = 0.0f);

        Vector4D(Vector3D v, float w = 0.0f);

        float& x();

        float& y();

        float& z();

        float& w();

        const float& x() const;

        const float& y() const;

        const float& z() const;

        const float& w() const;

        Vector4D& operator+=(const Vector4D& b);

        Vector4D& operator-=(const Vector4D& b);

        Vector4D& operator*=(const float& k);

        Vector4D& operator/=(const float& k);

    private:
        float m_x;
        float m_y;
        float m_z;
        float m_w;
    };

    //--------------------------------------------------------------------------------------
    //Vector4D Constructors

    inline Vector4D::Vector4D() :  m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }, m_w{ 0.0f }
    {}

    inline Vector4D::Vector4D(float x, float y, float z, float w) :  m_x{ x }, m_y{ y }, m_z{ z }, m_w{
  w }
    {}

    inline Vector4D::Vector4D(Vector2D v, float z, float w) :  m_x{ v.x() }, m_y{ v.y() }, m_z{ z },
  m_w{ w }
    {}

    inline Vector4D::Vector4D(Vector3D v, float w) :  m_x{ v.x() }, m_y{ v.y() }, m_z{ v.z() }, m_w{ w }
    {}

    //--------------------------------------------------------------------------------------

    //--------------------------------------------------------------------------------------
    //Vector4D Getters and Setters

    inline float& Vector4D::x()
    {
        return m_x;
    }

    inline float& Vector4D::y()
    {
        return m_y;
    }

    inline float& Vector4D::z()
    {
        return m_z;
    }

    inline float& Vector4D::w()
    {
        return m_w;
    }

    inline const float& Vector4D::x()const
{
        return m_x;
    }
```

```
    inline const float& Vector4D::y()const
{
        return m_y;
    }

    inline const float& Vector4D::z()const
{
        return m_z;
    }

    inline const float& Vector4D::w()const
{
        return m_w;
    }
    //-------------------------------------------------------------------------------------


    //-------------------------------------------------------------------------------------
    //Vector4D Memeber functions

    inline Vector4D& Vector4D::operator+=(const Vector4D& b)
    {
        this->m_x += (double)b.m_x;
        this->m_y += (double)b.m_y;
        this->m_z += (double)b.m_z;
        this->m_w += (double)b.m_w;

        return *this;
    }

    inline Vector4D& Vector4D::operator-=(const Vector4D& b)
    {
        this->m_x -= (double)b.m_x;
        this->m_y -= (double)b.m_y;
        this->m_z -= (double)b.m_z;
        this->m_w -= (double)b.m_w;

        return *this;
    }

    inline Vector4D& Vector4D::operator*=(const float& k)
    {
        this->m_x *= (double)k;
        this->m_y *= (double)k;
        this->m_z *= (double)k;
        this->m_w *= (double)k;

        return *this;
    }

    inline Vector4D& Vector4D::operator/=(const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return *this;
        }

        this->m_x /= (double)k;
        this->m_y /= (double)k;
        this->m_z /= (double)k;
        this->m_w /= (double)k;

        return *this;
    }

    //-------------------------------------------------------------------------------------

    //-------------------------------------------------------------------------------------
    //Vector4D Non-member functions

    inline bool zeroVector(const Vector4D& a)
    {
        if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON) &&
            compareFloats(a.z(), 0.0f, EPSILON) && compareFloats(a.w(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D((double)a.x() + b.x(), (double)a.y() + b.y(), (double)a.z() + b.z(),
  (double)a.w() + b.w());
    }
```

```
    inline Vector4D operator-(const Vector4D& v)
    {
        return Vector4D(-v.x(), -v.y(), -v.z(), -v.w());
    }

    inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D((double)a.x() - b.x(), (double)a.y() - b.y(), (double)a.z() - b.z(), (double)a.w() - b.w());
    }

    inline Vector4D operator*(const Vector4D& a, const float& k)
    {
        return Vector4D(a.x() * (double)k, a.y() * (double)k, a.z() * (double)k, a.w() * (double)k);
    }

    inline Vector4D operator*(const float& k, const Vector4D& a)
    {
        return Vector4D((double)k * a.x(), (double)k * a.y(), (double)k * a.z(), (double)k * a.w());
    }

    inline Vector4D operator/(const Vector4D& a, const float& k)
    {
        if (compareFloats(k, 0.0f, EPSILON))
        {
            return Vector4D();
        }

        return Vector4D(a.x() / (double)k, a.y() / (double)k, a.z() / (double)k, a.w() / (double)k);
    }

    inline float dotProduct(const Vector4D& a, const Vector4D& b)
    {
        //a dot b = axbx + ayby + azbz + awbw
        return (double)a.x() * b.x() + (double)a.y() * b.y() + (double)a.z() * b.z() + (double)a.w() * b.w();
    }

    inline float length(const Vector4D& v)
    {
        //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
        return sqrt((double)v.x() * v.x() + (double)v.y() * v.y() + (double)v.z() * v.z() + (double)v.w() * v.w());
    }

    inline Vector4D norm(const Vector4D& v)
    {
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
        //v is the zero vector
        if (zeroVector(v))
        {
            return v;
        }

        double mag = length(v);

        return Vector4D(v.x() / mag, v.y() / mag, v.z() / mag, v.w() / mag);
    }

    inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting
        Vector4D normB(norm(b));
        return Vector4D(dotProduct(a, normB) * normB);
    }


#if defined(_DEBUG)
    inline void print(const Vector4D& v)
    {
        std::cout « "(" « v.x() « ", " « v.y() « ", " « v.z() « ", " « v.w() « ")";
    }
#endif
    //--------------------------------------------------------------------------------


  //------------------------------------------------------------------------------------------------------------------------



  //------------------------------------------------------------------------------------------------------------------------
    class Matrix4x4
    {
    public:

```

```
        Matrix4x4();

        Matrix4x4(float a[][4]);

        float* data();

        const float* data() const;

        const float& operator()(unsigned int row, unsigned int col) const;

        float& operator()(unsigned int row, unsigned int col);

        void setRow(unsigned int row, Vector4D v);

        void setCol(unsigned int col, Vector4D v);

        Matrix4x4& operator+=(const Matrix4x4& m);

        Matrix4x4& operator-=(const Matrix4x4& m);

        Matrix4x4& operator*=(const float& k);


        Matrix4x4& operator*=(const Matrix4x4& m);

    private:

        float m_mat[4][4];
    };

    //-----------------------------------------------------------------------------------------
    inline Matrix4x4::Matrix4x4()
    {
        //1st row
        m_mat[0][0] = 1.0f;
        m_mat[0][1] = 0.0f;
        m_mat[0][2] = 0.0f;
        m_mat[0][3] = 0.0f;

        //2nd
        m_mat[1][0] = 0.0f;
        m_mat[1][1] = 1.0f;
        m_mat[1][2] = 0.0f;
        m_mat[1][3] = 0.0f;

        //3rd row
        m_mat[2][0] = 0.0f;
        m_mat[2][1] = 0.0f;
        m_mat[2][2] = 1.0f;
        m_mat[2][3] = 0.0f;

        //4th row
        m_mat[3][0] = 0.0f;
        m_mat[3][1] = 0.0f;
        m_mat[3][2] = 0.0f;
        m_mat[3][3] = 1.0f;
    }



    inline Matrix4x4::Matrix4x4(float a[][4])
    {
        //1st row
        m_mat[0][0] = a[0][0];
        m_mat[0][1] = a[0][1];
        m_mat[0][2] = a[0][2];
        m_mat[0][3] = a[0][3];

        //2nd
        m_mat[1][0] = a[1][0];
        m_mat[1][1] = a[1][1];
        m_mat[1][2] = a[1][2];
        m_mat[1][3] = a[1][3];

        //3rd row
        m_mat[2][0] = a[2][0];
        m_mat[2][1] = a[2][1];
        m_mat[2][2] = a[2][2];
        m_mat[2][3] = a[2][3];

        //4th row
        m_mat[3][0] = a[3][0];
        m_mat[3][1] = a[3][1];
        m_mat[3][2] = a[3][2];
        m_mat[3][3] = a[3][3];
    }

```

```
    inline float* Matrix4x4::data()
    {
        return m_mat[0];
    }

    inline const float* Matrix4x4::data()const
{
        return m_mat[0];
    }

    inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
{
        if (row > 3 || col > 3)
        {
            return m_mat[0][0];
        }
        else
        {
            return m_mat[row][col];
        }
    }

    inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
    {
        if (row > 3 || col > 3)
        {
            return m_mat[0][0];
        }
        else
        {
            return m_mat[row][col];
        }
    }

    inline void Matrix4x4::setRow(unsigned int row, Vector4D v)
    {
        if (row < 0 || row > 3)
        {
            m_mat[0][0] = v.x();
            m_mat[0][1] = v.y();
            m_mat[0][2] = v.z();
            m_mat[0][3] = v.w();
        }
        else
        {
            m_mat[row][0] = v.x();
            m_mat[row][1] = v.y();
            m_mat[row][2] = v.z();
            m_mat[row][3] = v.w();
        }
    }

    inline void Matrix4x4::setCol(unsigned int col, Vector4D v)
    {
        if (col < 0 || col > 3)
        {
            m_mat[0][0] = v.x();
            m_mat[1][0] = v.y();
            m_mat[2][0] = v.z();
            m_mat[3][0] = v.w();
        }
        else
        {
            m_mat[0][col] = v.x();
            m_mat[1][col] = v.y();
            m_mat[2][col] = v.z();
            m_mat[3][col] = v.w();
        }
    }

    inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
    {
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                this->m_mat[i][j] += (double)m.m_mat[i][j];
            }
        }

        return *this;
    }

    inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
    {
        for (int i = 0; i < 4; ++i)
        {
```

```
            for (int j = 0; j < 4; ++j)
            {
                this->m_mat[i][j] -= (double)m.m_mat[i][j];
            }
        }

        return *this;
    }

    inline Matrix4x4& Matrix4x4::operator*=(const float& k)
    {
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                this->m_mat[i][j] *= (double)k;
            }
        }

        return *this;
    }

    inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
    {
        Matrix4x4 res;

        for (int i = 0; i < 4; ++i)
        {
            res.m_mat[i][0] = ((double)m_mat[i][0] * m.m_mat[0][0]) +
                ((double)m_mat[i][1] * m.m_mat[1][0]) +
                ((double)m_mat[i][2] * m.m_mat[2][0]) +
                ((double)m_mat[i][3] * m.m_mat[3][0]);

            res.m_mat[i][1] = ((double)m_mat[i][0] * m.m_mat[0][1]) +
                ((double)m_mat[i][1] * m.m_mat[1][1]) +
                ((double)m_mat[i][2] * m.m_mat[2][1]) +
                ((double)m_mat[i][3] * m.m_mat[3][1]);

            res.m_mat[i][2] = ((double)m_mat[i][0] * m.m_mat[0][2]) +
                ((double)m_mat[i][1] * m.m_mat[1][2]) +
                ((double)m_mat[i][2] * m.m_mat[2][2]) +
                ((double)m_mat[i][3] * m.m_mat[3][2]);

            res.m_mat[i][3] = ((double)m_mat[i][0] * m.m_mat[0][3]) +
                ((double)m_mat[i][1] * m.m_mat[1][3]) +
                ((double)m_mat[i][2] * m.m_mat[2][3]) +
                ((double)m_mat[i][3] * m.m_mat[3][3]);
        }

        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                m_mat[i][j] = res.m_mat[i][j];
            }
        }

        return *this;
    }

    inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
    {
        Matrix4x4 res;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                res(i, j) = (double)m1(i, j) + m2(i, j);
            }
        }

        return res;
    }

    inline Matrix4x4 operator-(const Matrix4x4& m)
    {
        Matrix4x4 res;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                res(i, j) = -m(i, j);
            }
        }

        return res;
    }
```

```
    inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
    {
        Matrix4x4 res;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                res(i, j) = (double)m1(i, j) - m2(i, j);
            }
        }

        return res;
    }

    inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
    {
        Matrix4x4 res;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                res(i, j) = (double)m(i, j) * k;
            }
        }

        return res;
    }

    inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
    {
        Matrix4x4 res;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                res(i, j) = k * (double)m(i, j);
            }
        }

        return res;
    }

    inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
    {
        Matrix4x4 res;

        for (int i = 0; i < 4; ++i)
        {
            res(i, 0) = ((double)m1(i, 0) * m2(0, 0)) +
                ((double)m1(i, 1) * m2(1, 0)) +
                ((double)m1(i, 2) * m2(2, 0)) +
                ((double)m1(i, 3) * m2(3, 0));

            res(i, 1) = ((double)m1(i, 0) * m2(0, 1)) +
                ((double)m1(i, 1) * m2(1, 1)) +
                ((double)m1(i, 2) * m2(2, 1)) +
                ((double)m1(i, 3) * m2(3, 1));

            res(i, 2) = ((double)m1(i, 0) * m2(0, 2)) +
                ((double)m1(i, 1) * m2(1, 2)) +
                ((double)m1(i, 2) * m2(2, 2)) +
                ((double)m1(i, 3) * m2(3, 2));

            res(i, 3) = ((double)m1(i, 0) * m2(0, 3)) +
                ((double)m1(i, 1) * m2(1, 3)) +
                ((double)m1(i, 2) * m2(2, 3)) +
                ((double)m1(i, 3) * m2(3, 3));
        }

        return res;
    }

    inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
    {
        Vector4D res;

        res.x() = ((double)m(0, 0) * v.x() + (double)m(0, 1) * v.y() + (double)m(0, 2) * v.z() + (double)m(0, 3) * v.w());
        res.y() = ((double)m(1, 0) * v.x() + (double)m(1, 1) * v.y() + (double)m(1, 2) * v.z() + (double)m(1, 3) * v.w());
        res.z() = ((double)m(2, 0) * v.x() + (double)m(2, 1) * v.y() + (double)m(2, 2) * v.z() + (double)m(2, 3) * v.w());
        res.w() = ((double)m(3, 0) * v.x() + (double)m(3, 1) * v.y() + (double)m(3, 2) * v.z() + (double)m(3, 3) * v.w());

        return res;
```

```
    }

    inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
    {
        Vector4D res;

        res.x() = ((double)v.x() * m(0, 0) + (double)v.y() * m(1, 0) + (double)v.z() * m(2, 0) +
  (double)v.w() * m(3, 0));
        res.y() = ((double)v.x() * m(0, 1) + (double)v.y() * m(1, 1) + (double)v.z() * m(2, 1) +
  (double)v.w() * m(3, 1));
        res.z() = ((double)v.x() * m(0, 2) + (double)v.y() * m(1, 2) + (double)v.z() * m(2, 2) +
  (double)v.w() * m(3, 2));
        res.w() = ((double)v.x() * m(0, 3) + (double)v.y() * m(1, 3) + (double)v.z() * m(2, 3) +
  (double)v.w() * m(3, 3));

        return res;
    }

    inline void setToIdentity(Matrix4x4& m)
    {
        //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f

        //1st row
        m(0, 0) = 1.0f;
        m(0, 1) = 0.0f;
        m(0, 2) = 0.0f;
        m(0, 3) = 0.0f;

        //2nd row
        m(1, 0) = 0.0f;
        m(1, 1) = 1.0f;
        m(1, 2) = 0.0f;
        m(1, 3) = 0.0f;

        //3rd row
        m(2, 0) = 0.0f;
        m(2, 1) = 0.0f;
        m(2, 2) = 1.0f;
        m(2, 3) = 0.0f;

        //4th row
        m(3, 0) = 0.0f;
        m(3, 1) = 0.0f;
        m(3, 2) = 0.0f;
        m(3, 3) = 1.0f;
    }

    inline bool isIdentity(const Matrix4x4& m)
    {
        //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
  0.0f

        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                if (i == j)
                {
                    if (!compareFloats(m(i, j), 1.0f, EPSILON))
                    {
                        return false;
                    }
                }
                else
                {
                    if (!compareFloats(m(i, j), 0.0f, EPSILON))
                    {
                        return false;
                    }
                }
            }
        }
    }

    inline Matrix4x4 transpose(const Matrix4x4& m)
    {
        //make the rows into cols

        Matrix4x4 res;

        //1st col = 1st row
        res(0, 0) = m(0, 0);
        res(1, 0) = m(0, 1);
        res(2, 0) = m(0, 2);
        res(3, 0) = m(0, 3);

        //2nd col = 2nd row
```

```
        res(0, 1) = m(1, 0);
        res(1, 1) = m(1, 1);
        res(2, 1) = m(1, 2);
        res(3, 1) = m(1, 3);

        //3rd col = 3rd row
        res(0, 2) = m(2, 0);
        res(1, 2) = m(2, 1);
        res(2, 2) = m(2, 2);
        res(3, 2) = m(2, 3);

        //4th col = 4th row
        res(0, 3) = m(3, 0);
        res(1, 3) = m(3, 1);
        res(2, 3) = m(3, 2);
        res(3, 3) = m(3, 3);

        return res;
    }

    inline Matrix4x4 translate(const Matrix4x4& cm, float x, float y, float z)
    {
        //1 0 0 0
        //0 1 0 0
        //0 0 1 0
        //x y z 1

        Matrix4x4 t;
        t(3, 0) = x;
        t(3, 1) = y;
        t(3, 2) = z;

        return cm * t;
    }

    inline Matrix4x4 scale(const Matrix4x4& cm, float x, float y, float z)
    {
        //x 0 0 0
        //0 y 0 0
        //0 0 z 0
        //0 0 0 1

        Matrix4x4 s;
        s(0, 0) = x;
        s(1, 1) = y;
        s(2, 2) = z;

        return cm * s;
    }

    inline Matrix4x4 rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
    {

        //c + (1 - c)x^2     (1 - c)xy + sz   (1 - c)xz - sy   0
        //(1 - c)xy - sz     c + (1 - c)y^2   (1 - c)yz + sx   0
        //(1 - c)xz + sy     (1 - c)yz - sx   c + (1 - c)z^2   0
        //0                  0                0                1
        //c = cos(angle)
        //s = sin(angle)

        double c = cos(angle * PI / 180.0);
        double s = sin(angle * PI / 180.0);

        Matrix4x4 r;

        //1st row
        r(0, 0) = c + (1.0 - c) * ((double)x * x);
        r(0, 1) = (1.0 - c) * ((double)x * y) + (s * z);
        r(0, 2) = (1.0 - c) * ((double)x * z) - (s * y);

        //2nd row
        r(1, 0) = (1.0 - c) * ((double)x * y) - (s * z);
        r(1, 1) = c + (1.0 - c) * ((double)y * y);
        r(1, 2) = (1.0 - c) * ((double)y * z) + (s * x);

        //3rd row
        r(2, 0) = (1.0 - c) * ((double)x * z) + (s * y);
        r(2, 1) = (1.0 - c) * ((double)y * z) - (s * x);
        r(2, 2) = c + (1.0 - c) * ((double)z * z);

        return cm * r;
    }

    inline double det(const Matrix4x4& m)
    {
        //m00m11(m22m33 - m23m32)
        double c1 = (double)m(0, 0) * m(1, 1) * m(2, 2) * m(3, 3) - (double)m(0, 0) * m(1, 1) * m(2, 3)
```

```
* m(3, 2);

        //m00m12(m23m31 - m21m33)
        double c2 = (double)m(0, 0) * m(1, 2) * m(2, 3) * m(3, 1) - (double)m(0, 0) * m(1, 2) * m(2, 1)
* m(3, 3);

        //m00m13(m21m32 - m22m31)
        double c3 = (double)m(0, 0) * m(1, 3) * m(2, 1) * m(3, 2) - (double)m(0, 0) * m(1, 3) * m(2, 2)
* m(3, 1);

        //m01m10(m22m33 - m23m32)
        double c4 = (double)m(0, 1) * m(1, 0) * m(2, 2) * m(3, 3) - (double)m(0, 1) * m(1, 0) * m(2, 3)
* m(3, 2);

        //m01m12(m23m30 - m20m33)
        double c5 = (double)m(0, 1) * m(1, 2) * m(2, 3) * m(3, 0) - (double)m(0, 1) * m(1, 2) * m(2, 0)
* m(3, 3);

        //m01m13(m20m32 - m22m30)
        double c6 = (double)m(0, 1) * m(1, 3) * m(2, 0) * m(3, 2) - (double)m(0, 1) * m(1, 3) * m(2, 2)
* m(3, 0);

        //m02m10(m21m33 - m23m31)
        double c7 = (double)m(0, 2) * m(1, 0) * m(2, 1) * m(3, 3) - (double)m(0, 2) * m(1, 0) * m(2, 3)
* m(3, 1);

        //m02m11(m23m30 - m20m33)
        double c8 = (double)m(0, 2) * m(1, 1) * m(2, 3) * m(3, 0) - (double)m(0, 2) * m(1, 1) * m(2, 0)
* m(3, 3);

        //m02m13(m20m31 - m21m30)
        double c9 = (double)m(0, 2) * m(1, 3) * m(2, 0) * m(3, 1) - (double)m(0, 2) * m(1, 3) * m(2, 1)
* m(3, 0);

        //m03m10(m21m32 - m22m21)
        double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
2) * m(3, 1);

        //m03m11(m22m30 - m20m32)
        double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
0) * m(3, 2);

        //m03m12(m20m31 - m21m30)
        double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
1) * m(3, 0);

        return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
    }

    inline double cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
    {
        //cij = (-1)^i + j * det of minor(i, j);
        double tempMat[3][3];
        int tr{ 0 };
        int tc{ 0 };

        //minor(i, j)
        for (int i = 0; i < 4; ++i)
        {
            if (i == row)
                continue;

            for (int j = 0; j < 4; ++j)
            {
                if (j == col)
                    continue;

                tempMat[tr][tc] = m(i, j);
                ++tc;

            }
            tc = 0;
            ++tr;
        }

        //determinant of minor(i, j)
        double det3x3 = (tempMat[0][0] * tempMat[1][1] * tempMat[2][2]) + (tempMat[0][1] *
tempMat[1][2] * tempMat[2][0]) +
            (tempMat[0][2] * tempMat[1][0] * tempMat[2][1]) - (tempMat[0][2] * tempMat[1][1] *
tempMat[2][0]) -
            (tempMat[0][1] * tempMat[1][0] * tempMat[2][2]) - (tempMat[0][0] * tempMat[1][2] *
tempMat[2][1]);

        return pow(-1, row + col) * det3x3;
    }

    inline Matrix4x4 adjoint(const Matrix4x4& m)
```

```
    {
        //Cofactor of each ijth position put into matrix cA.
        //Adjoint is the tranposed matrix of cA.
        Matrix4x4 cA;
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                cA(i, j) = cofactor(m, i, j);
            }
        }

        return transpose(cA);
    }

    inline Matrix4x4 inverse(const Matrix4x4& m)
    {
        //Inverse of m = adjoint of m / det of m
        double determinant = det(m);
        if (compareDoubles(determinant, 0.0, EPSILON))
            return Matrix4x4();

        return adjoint(m) * (1.0 / determinant);
    }


#if defined(_DEBUG)
    inline void print(const Matrix4x4& m)
    {
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                std::cout « m(i, j) « " ";
            }

            std::cout « std::endl;
        }
    }
#endif


//------------------------------------------------------------------------------------------------------------------------------




//------------------------------------------------------------------------------------------------------------------------------

    class Quaternion
    {
    public:

        Quaternion();

        Quaternion(float scalar, float x, float y, float z);

        Quaternion(float scalar, const Vector3D& v);

        Quaternion(const Vector4D& v);

        float& scalar();

        const float& scalar() const;

        float& x();

        const float& x() const;

        float& y();

        const float& y() const;

        float& z();

        const float& z() const;

        Vector3D vector();

        Quaternion& operator+=(const Quaternion& q);

        Quaternion& operator-=(const Quaternion& q);

        Quaternion& operator*=(float k);

        Quaternion& operator*=(const Quaternion& q);

```

```
    private:

        float m_scalar;
        float m_x;
        float m_y;
        float m_z;
    };

    //-----------------------------------------------------------------------------------
    inline Quaternion::Quaternion() :  m_scalar{ 1.0f }, m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }
    {
    }

    inline Quaternion::Quaternion(float scalar, float x, float y, float z) :  m_scalar{ scalar }, m_x{
 x }, m_y{ y }, m_z{ z }
    {
    }

    inline Quaternion::Quaternion(float scalar, const Vector3D& v) :  m_scalar{ scalar }, m_x{ v.x() },
 m_y{ v.y() }, m_z{ v.z() }
    {
    }

    inline Quaternion::Quaternion(const Vector4D& v) :  m_scalar{ v.x() }, m_x{ v.y() }, m_y{ v.z() },
 m_z{ v.w() }
    {
    }

    inline float& Quaternion::scalar()
    {
        return m_scalar;
    }

    inline const float& Quaternion::scalar()const
{
        return m_scalar;
    }

    inline float& Quaternion::x()
    {
        return m_x;
    }

    inline const float& Quaternion::x()const
{
        return m_x;
    }

    inline float& Quaternion::y()
    {
        return m_y;
    }

    inline const float& Quaternion::y()const
{
        return m_y;
    }

    inline float& Quaternion::z()
    {
        return m_z;
    }

    inline const float& Quaternion::z()const
{
        return m_z;
    }

    inline Vector3D Quaternion::vector()
    {
        return Vector3D(m_x, m_y, m_z);
    }

    inline Quaternion& Quaternion::operator+=(const Quaternion& q)
    {
        this->m_scalar += (double)q.m_scalar;
        this->m_x += (double)q.m_x;
        this->m_y += (double)q.m_y;
        this->m_z += (double)q.m_z;

        return *this;
    }

    inline Quaternion& Quaternion::operator-=(const Quaternion& q)
    {
        this->m_scalar -= (double)q.m_scalar;
```

```
        this->m_x -= (double)q.m_x;
        this->m_y -= (double)q.m_y;
        this->m_z -= (double)q.m_z;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(float k)
    {
        this->m_scalar *= (double)k;
        this->m_x *= (double)k;
        this->m_y *= (double)k;
        this->m_z *= (double)k;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(const Quaternion& q)
    {
        Vector3D thisVector(this->m_x, this->m_y, this->m_z);
        Vector3D qVector(q.m_x, q.m_y, q.m_z);

        double s{ (double)this->m_scalar * q.m_scalar };
        double dP{ dotProduct(thisVector, qVector) };
        double resultScalar{ s - dP };

        Vector3D a(this->m_scalar * qVector);
        Vector3D b(q.m_scalar * thisVector);
        Vector3D cP(crossProduct(thisVector, qVector));
        Vector3D resultVector(a + b + cP);

        this->m_scalar = resultScalar;
        this->m_x = resultVector.x();
        this->m_y = resultVector.y();
        this->m_z = resultVector.z();

        return *this;
    }

    inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion((double)q1.scalar() + q2.scalar(), (double)q1.x() + q2.x(), (double)q1.y() + q2.y(), (double)q1.z() + q2.z());
    }

    inline Quaternion operator-(const Quaternion& q)
    {
        return Quaternion(-q.scalar(), -q.x(), -q.y(), -q.z());
    }

    inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion((double)q1.scalar() - q2.scalar(), (double)q1.x() - q2.x(), (double)q1.y() - q2.y(), (double)q1.z() - q2.z());
    }

    inline Quaternion operator*(float k, const Quaternion& q)
    {
        return Quaternion((double)k * q.scalar(), (double)k * q.x(), (double)k * q.y(), (double)k * q.z());
    }

    inline Quaternion operator*(const Quaternion& q, float k)
    {
        return Quaternion(q.scalar() * (double)k, q.x() * (double)k, q.y() * (double)k, q.z() * (double)k);
    }

    inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
    {
        //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
        //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector

        Vector3D q1Vector(q1.x(), q1.y(), q1.z());
        Vector3D q2Vector(q2.x(), q2.y(), q2.z());

        double s{ (double)q1.scalar() * q2.scalar() };
        double dP{ dotProduct(q1Vector, q2Vector) };
        double resultScalar{ s - dP };

        Vector3D a(q1.scalar() * q2Vector);
        Vector3D b(q2.scalar() * q1Vector);
        Vector3D cP(crossProduct(q1Vector, q2Vector));
        Vector3D resultVector(a + b + cP);

        return Quaternion(resultScalar, resultVector);
    }
```

```
    inline bool isZeroQuaternion(const Quaternion& q)
    {
        //zero quaternion = (0, 0, 0, 0)
        return compareFloats(q.scalar(), 0.0f, EPSILON) && compareFloats(q.x(), 0.0f, EPSILON) &&
            compareFloats(q.y(), 0.0f, EPSILON) && compareFloats(q.z(), 0.0f, EPSILON);
    }

    inline bool isIdentity(const Quaternion& q)
    {
        //identity quaternion = (1, 0, 0, 0)
        return compareFloats(q.scalar(), 1.0f, EPSILON) && compareFloats(q.x(), 0.0f, EPSILON) &&
            compareFloats(q.y(), 0.0f, EPSILON) && compareFloats(q.z(), 0.0f, EPSILON);
    }

    inline Quaternion conjugate(const Quaternion& q)
    {
        //conjugate of a quaternion is the quaternion with its vector part negated
        return Quaternion(q.scalar(), -q.x(), -q.y(), -q.z());
    }

    inline float length(const Quaternion& q)
    {
        //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
        return sqrt((double)q.scalar() * q.scalar() + (double)q.x() * q.x() + (double)q.y() * q.y() +
(double)q.z() * q.z());
    }

    inline Quaternion normalize(const Quaternion& q)
    {
        //to normalize a quaternion you do q / |q|

        if (isZeroQuaternion(q))
            return q;

        double d{ length(q) };

        return Quaternion(q.scalar() / d, q.x() / d, q.y() / d, q.z() / d);
    }

    inline Quaternion inverse(const Quaternion& q)
    {
        //inverse = conjugate of q / |q|^2

        if (isZeroQuaternion(q))
            return q;

        Quaternion conjugateOfQ(conjugate(q));

        double d{ length(q) };
        d *= d;

        return Quaternion(conjugateOfQ.scalar() / d, conjugateOfQ.x() / d, conjugateOfQ.y() / d,
conjugateOfQ.z() / d);
    }

    inline Quaternion rotationQuaternion(float angle, float x, float y, float z)
    {
        //A roatation quaternion is a quaternion where the
        //scalar part = cos(theta / 2)
        //vector part = sin(theta / 2) * axis
        //the axis needs to be normalized

        double ang{ angle / 2.0 };
        double c{ cos(ang * PI / 180.0) };
        double s{ sin(ang * PI / 180.0) };

        Vector3D axis(x, y, z);
        axis = norm(axis);

        return Quaternion(c, s * axis.x(), s * axis.y(), s * axis.z());
    }

    inline Quaternion rotationQuaternion(float angle, const Vector3D& axis)
    {
        //A roatation quaternion is a quaternion where the
        //scalar part = cos(theta / 2)
        //vector part = sin(theta / 2) * axis
        //the axis needs to be normalized

        double ang{ angle / 2.0 };
        double c{ cos(ang * PI / 180.0) };
        double s{ sin(ang * PI / 180.0) };

        Vector3D axisN(norm(axis));

        return Quaternion(c, s * axisN.x(), s * axisN.y(), s * axisN.z());
```

```
}

inline Quaternion rotationQuaternion(const Vector4D& angAxis)
{
    //A roatation quaternion is a quaternion where the
    //scalar part = cos(theta / 2)
    //vector part = sin(theta / 2) * axis
    //the axis needs to be normalized

    double angle{ angAxis.x() / 2.0 };
    double c{ cos(angle * PI / 180.0) };
    double s{ sin(angle * PI / 180.0) };

    Vector3D axis(angAxis.y(), angAxis.z(), angAxis.w());
    axis = norm(axis);

    return Quaternion(c, s * axis.x(), s * axis.y(), s * axis.z());
}

inline Matrix4x4 quaternionRotationMatrixCol(const Quaternion& q)
{
    //1 - 2q3^2 - 2q4^2     2q2q3 - 2q1q4       2q2q4 + 2q1q3       0
    //2q2q3 + 2q1q4         1 - 2q2^2 - 2q4^2   2q3q4 - 2q1q2       0
    //2q2q4 - 2q1q3         2q3q4 + 2q1q2       1 - 2q2^2 - 2q3^2   0
    //0                     0                   0                   1
    //q1 = scalar
    //q2 = x
    //q3 = y
    //q4 = z

    float colMat[4][4] = {};

    colMat[0][0] = 1.0 - 2.0 * q.y() * q.y() - 2.0 * q.z() * q.z();
    colMat[0][1] = 2.0 * q.x() * q.y() - 2.0 * q.scalar() * q.z();
    colMat[0][2] = 2.0 * q.x() * q.z() + 2.0 * q.scalar() * q.y();
    colMat[0][3] = 0.0f;

    colMat[1][0] = 2.0 * q.x() * q.y() + 2.0 * q.scalar() * q.z();
    colMat[1][1] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.z() * q.z();
    colMat[1][2] = 2.0 * q.y() * q.z() - 2.0 * q.scalar() * q.x();
    colMat[1][3] = 0.0f;

    colMat[2][0] = 2.0 * q.x() * q.z() - 2.0 * q.scalar() * q.y();
    colMat[2][1] = 2.0 * q.y() * q.z() + 2.0 * q.scalar() * q.x();
    colMat[2][2] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.y() * q.y();
    colMat[2][3] = 0.0f;

    colMat[3][0] = 0.0f;
    colMat[3][1] = 0.0f;
    colMat[3][2] = 0.0f;
    colMat[3][3] = 1.0f;

    return Matrix4x4(colMat);
}

inline Matrix4x4 quaternionRotationMatrixRow(const Quaternion& q)
{
    //1 - 2q3^2 - 2q4^2     2q2q3 + 2q1q4       2q2q4 - 2q1q3       0
    //2q2q3 - 2q1q4         1 - 2q2^2 - 2q4^2   2q3q4 + 2q1q2       0
    //2q2q4 + 2q1q3         2q3q4 - 2q1q2       1 - 2q2^2 - 2q3^2   0
    //0                     0                   0                   1
    //q1 = scalar
    //q2 = x
    //q3 = y
    //q4 = z

    float rowMat[4][4] = {};

    rowMat[0][0] = 1.0 - 2.0 * q.y() * q.y() - 2.0 * q.z() * q.z();
    rowMat[0][1] = 2.0 * q.x() * q.y() + 2.0 * q.scalar() * q.z();
    rowMat[0][2] = 2.0 * q.x() * q.z() - 2.0 * q.scalar() * q.y();
    rowMat[0][3] = 0.0f;

    rowMat[1][0] = 2.0 * q.x() * q.y() - 2.0 * q.scalar() * q.z();
    rowMat[1][1] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.z() * q.z();
    rowMat[1][2] = 2.0 * q.y() * q.z() + 2.0 * q.scalar() * q.x();
    rowMat[1][3] = 0.0f;

    rowMat[2][0] = 2.0 * q.x() * q.z() + 2.0 * q.scalar() * q.y();
    rowMat[2][1] = 2.0 * q.y() * q.z() - 2.0 * q.scalar() * q.x();
    rowMat[2][2] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.y() * q.y();
    rowMat[2][3] = 0.0f;

    rowMat[3][0] = 0.0f;
    rowMat[3][1] = 0.0f;
    rowMat[3][2] = 0.0f;
    rowMat[3][3] = 1.0f;
```

```

        return Matrix4x4(rowMat);
    }

#if defined(_DEBUG)
    inline void print(const Quaternion& q)
    {
        std::cout « "(" « q.scalar() « ", " « q.x() « ", " « q.y() « ", " « q.z();
    }
#endif
    //--------------------------------------------------------------------------------------


 //-----------------------------------------------------------------------------------------------------------------------
}
```

Index