

Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4

1 Namespace Index	1
1.1 Namespace List	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Namespace Documentation	7
4.1 FAMath Namespace Reference	7
4.1.1 Detailed Description	11
4.1.2 Function Documentation	11
4.1.2.1 adjoint()	11
4.1.2.2 CartesianToCylindrical()	11
4.1.2.3 CartesianToPolar()	11
4.1.2.4 CartesianToSpherical()	11
4.1.2.5 cofactor()	12
4.1.2.6 conjugate()	12
4.1.2.7 crossProduct()	12
4.1.2.8 CylindricalToCartesian()	12
4.1.2.9 det()	12
4.1.2.10 dotProduct() [1/3]	13
4.1.2.11 dotProduct() [2/3]	13
4.1.2.12 dotProduct() [3/3]	13
4.1.2.13 inverse() [1/2]	13
4.1.2.14 inverse() [2/2]	13
4.1.2.15 isIdentity() [1/2]	14
4.1.2.16 isIdentity() [2/2]	14
4.1.2.17 isZeroQuaternion()	14
4.1.2.18 length() [1/4]	14
4.1.2.19 length() [2/4]	14
4.1.2.20 length() [3/4]	14
4.1.2.21 length() [4/4]	15
4.1.2.22 norm() [1/3]	15
4.1.2.23 norm() [2/3]	15
4.1.2.24 norm() [3/3]	15
4.1.2.25 normalize()	15
4.1.2.26 operator*() [1/14]	16
4.1.2.27 operator*() [2/14]	16
4.1.2.28 operator*() [3/14]	16
4.1.2.29 operator*() [4/14]	16
4.1.2.30 operator*() [5/14]	16

4.1.2.31 operator*() [6/14]	17
4.1.2.32 operator*() [7/14]	17
4.1.2.33 operator*() [8/14]	17
4.1.2.34 operator*() [9/14]	17
4.1.2.35 operator*() [10/14]	17
4.1.2.36 operator*() [11/14]	18
4.1.2.37 operator*() [12/14]	18
4.1.2.38 operator*() [13/14]	18
4.1.2.39 operator*() [14/14]	18
4.1.2.40 operator+() [1/5]	18
4.1.2.41 operator+() [2/5]	19
4.1.2.42 operator+() [3/5]	19
4.1.2.43 operator+() [4/5]	19
4.1.2.44 operator+() [5/5]	19
4.1.2.45 operator-() [1/10]	19
4.1.2.46 operator-() [2/10]	20
4.1.2.47 operator-() [3/10]	20
4.1.2.48 operator-() [4/10]	20
4.1.2.49 operator-() [5/10]	20
4.1.2.50 operator-() [6/10]	20
4.1.2.51 operator-() [7/10]	21
4.1.2.52 operator-() [8/10]	21
4.1.2.53 operator-() [9/10]	21
4.1.2.54 operator-() [10/10]	21
4.1.2.55 operator/() [1/3]	21
4.1.2.56 operator/() [2/3]	22
4.1.2.57 operator/() [3/3]	22
4.1.2.58 PolarToCartesian()	22
4.1.2.59 Projection() [1/3]	22
4.1.2.60 Projection() [2/3]	22
4.1.2.61 Projection() [3/3]	23
4.1.2.62 quaternionRotationMatrixCol()	23
4.1.2.63 quaternionRotationMatrixRow()	23
4.1.2.64 rotate()	23
4.1.2.65 rotationQuaternion() [1/3]	23
4.1.2.66 rotationQuaternion() [2/3]	24
4.1.2.67 rotationQuaternion() [3/3]	24
4.1.2.68 scale()	24
4.1.2.69 setToIdentity()	24
4.1.2.70 SphericalToCartesian()	24
4.1.2.71 translate()	25
4.1.2.72 transpose()	25

4.1.2.73 zeroVector() [1/3]	25
4.1.2.74 zeroVector() [2/3]	25
4.1.2.75 zeroVector() [3/3]	25
5 Class Documentation	27
5.1 FAMath::Matrix4x4 Class Reference	27
5.1.1 Detailed Description	28
5.1.2 Constructor & Destructor Documentation	28
5.1.2.1 Matrix4x4() [1/2]	28
5.1.2.2 Matrix4x4() [2/2]	28
5.1.3 Member Function Documentation	28
5.1.3.1 data() [1/2]	28
5.1.3.2 data() [2/2]	28
5.1.3.3 operator>() [1/2]	29
5.1.3.4 operator>() [2/2]	29
5.1.3.5 operator*=() [1/2]	29
5.1.3.6 operator*=() [2/2]	29
5.1.3.7 operator+=()	29
5.1.3.8 operator-=()	30
5.1.3.9 setCol()	30
5.1.3.10 setRow()	30
5.2 FAMath::Quaternion Class Reference	30
5.2.1 Detailed Description	31
5.2.2 Constructor & Destructor Documentation	31
5.2.2.1 Quaternion() [1/4]	32
5.2.2.2 Quaternion() [2/4]	32
5.2.2.3 Quaternion() [3/4]	32
5.2.2.4 Quaternion() [4/4]	32
5.2.3 Member Function Documentation	32
5.2.3.1 operator*=() [1/2]	32
5.2.3.2 operator*=() [2/2]	33
5.2.3.3 operator+=()	33
5.2.3.4 operator-=()	33
5.2.3.5 scalar() [1/2]	33
5.2.3.6 scalar() [2/2]	33
5.2.3.7 vector()	33
5.2.3.8 x() [1/2]	34
5.2.3.9 x() [2/2]	34
5.2.3.10 y() [1/2]	34
5.2.3.11 y() [2/2]	34
5.2.3.12 z() [1/2]	34
5.2.3.13 z() [2/2]	34

5.3 FAMath::Vector2D Class Reference	35
5.3.1 Detailed Description	35
5.3.2 Constructor & Destructor Documentation	35
5.3.2.1 Vector2D() [1/2]	35
5.3.2.2 Vector2D() [2/2]	36
5.3.3 Member Function Documentation	36
5.3.3.1 operator*=()	36
5.3.3.2 operator+=()	36
5.3.3.3 operator-=()	36
5.3.3.4 operator/=()	36
5.3.3.5 x() [1/2]	37
5.3.3.6 x() [2/2]	37
5.3.3.7 y() [1/2]	37
5.3.3.8 y() [2/2]	37
5.4 FAMath::Vector3D Class Reference	37
5.4.1 Detailed Description	38
5.4.2 Constructor & Destructor Documentation	38
5.4.2.1 Vector3D() [1/2]	38
5.4.2.2 Vector3D() [2/2]	39
5.4.3 Member Function Documentation	39
5.4.3.1 operator*=()	39
5.4.3.2 operator+=()	39
5.4.3.3 operator-=()	39
5.4.3.4 operator/=()	39
5.4.3.5 x() [1/2]	40
5.4.3.6 x() [2/2]	40
5.4.3.7 y() [1/2]	40
5.4.3.8 y() [2/2]	40
5.4.3.9 z() [1/2]	40
5.4.3.10 z() [2/2]	40
5.5 FAMath::Vector4D Class Reference	41
5.5.1 Detailed Description	41
5.5.2 Constructor & Destructor Documentation	42
5.5.2.1 Vector4D() [1/4]	42
5.5.2.2 Vector4D() [2/4]	42
5.5.2.3 Vector4D() [3/4]	42
5.5.2.4 Vector4D() [4/4]	42
5.5.3 Member Function Documentation	42
5.5.3.1 operator*=()	43
5.5.3.2 operator+=()	43
5.5.3.3 operator-=()	43
5.5.3.4 operator/=()	43

5.5.3.5 $w()$ [1/2]	43
5.5.3.6 $w()$ [2/2]	43
5.5.3.7 $x()$ [1/2]	44
5.5.3.8 $x()$ [2/2]	44
5.5.3.9 $y()$ [1/2]	44
5.5.3.10 $y()$ [2/2]	44
5.5.3.11 $z()$ [1/2]	44
5.5.3.12 $z()$ [2/2]	44
6 File Documentation	45
6.1 FAMathEngine.h	45
Index	69

Chapter 1

Namespace Index

1.1 Namespace List

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

FAMath	Has utility functions, Vector2D , Vector3D , Vector4D , Matrix4x4 , and Quaternion classes	7
------------------------	--	-------------------

Chapter 2

Class Index

2.1 Class List

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

FAMath::Matrix4x4	
A matrix class used for 4x4 matrices and their manipulations	27
FAMath::Quaternion	30
FAMath::Vector2D	
A vector class used for 2D vectors/points and their manipulations	35
FAMath::Vector3D	
A vector class used for 3D vectors/points and their manipulations	37
FAMath::Vector4D	
A vector class used for 4D vectors/points and their manipulations	41

Chapter 3

File Index

3.1 File List

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

FAMathEngine.h	??
--------------------------------	-------	----

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 negation.
 - `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 = (rho, 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 v_x is zero then no conversion happens and v is returned.

The returned 3D vector = $(r, \text{phi}(\text{degrees}), \text{theta}(\text{degrees}))$.

- **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 setTolIdentity** (**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 v_x is zero then no conversion happens and v is returned.
The returned 3D vector = $(r, \text{theta}(\text{degrees}), z)$.

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 v_x is zero then no conversion happens and v is returned.
The returned 2D vector = $(r, \text{theta}(\text{degrees}))$.

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 v_x is zero then no conversion happens and v is returned.
The returned 3D vector = $(r, \text{phi}(\text{degrees}), \text{theta}(\text{degrees}))$.

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 negation.

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 quaternion q and stores the result in this quaternion.
- [Quaternion](#) & [operator-=](#) (const [Quaternion](#) &[q](#))
Subtracts this quaternion by quaternion q and stores the result in this quaternion.
- [Quaternion](#) & [operator*=](#) (float [k](#))
Multiplies this quaternion by float k and stores the result in this quaternion.
- [Quaternion](#) & [operator*=](#) (const [Quaternion](#) &[q](#))
Multiplies this quaternion by quaternion 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 quaternion 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 float 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 quaternion 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 quaternion 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

```
1 #pragma once
2
3 #include <cmath>
4
5 #if defined(_DEBUG)
6 #include <iostream>
7 #endif
8
9
10 #define EPSILON 1e-6f
11 #define PI 3.14159265
12
13 namespace FAMath
14 {
15     class Vector2D;
16     class Vector3D;
17     class Vector4D;
18
19     /**@brief Checks if the two specified floats are equal using exact epsilon and adaptive epsilon.
20     */
21     inline bool compareFloats(float x, float y, float epsilon)
22     {
23         float diff = fabs(x - y);
24         //exact epsilon
25         if (diff < epsilon)
26         {
27             return true;
28         }
29
30         //adaptive epsilon
31         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
32     }
33
34     /**@brief Checks if the two specified doubles are equal using exact epsilon and adaptive epsilon.
35     */
36     inline bool compareDoubles(double x, double y, double epsilon)
37     {
38         double diff = fabs(x - y);
39         //exact epsilon
40         if (diff < epsilon)
41         {
42             return true;
43         }
44
45         //adaptive epsilon
46         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
47     }
48
49     //-----
50
51     class Vector2D
52     {
53     public:
54
55         Vector2D();
56     }
```

```

74     Vector2D(float x, float y);
75
76     float& x();
77
78     float& y();
79
80     const float& x() const;
81
82     const float& y() const;
83
84     Vector2D& operator+=(const Vector2D& b);
85
86     Vector2D& operator-=(const Vector2D& b);
87
88     Vector2D& operator*=(const float& k);
89
90     Vector2D& operator/=(const float& k);
91
92 private:
93     float m_x;
94     float m_y;
95 };
96
97 //-----
98 //Vector2D Constructors
99
100 inline Vector2D::Vector2D() : m_x{ 0.0f }, m_y{ 0.0f }
101 {}
102
103 inline Vector2D::Vector2D(float x, float y) : m_x{ x }, m_y{ y }
104 {}
105
106 //-----
107 //-----
108 //Vector2D Getters and Setters
109
110 inline float& Vector2D::x()
111 {
112     return m_x;
113 }
114
115 inline float& Vector2D::y()
116 {
117     return m_y;
118 }
119
120 inline const float& Vector2D::x() const
121 {
122     return m_x;
123 }
124
125 inline const float& Vector2D::y() const
126 {
127     return m_y;
128 }
129
130 //-----
131 //-----
132 //Vector2D Member functions
133
134 inline Vector2D& Vector2D::operator+=(const Vector2D& b)
135 {
136     this->m_x += (double)b.m_x;
137     this->m_y += (double)b.m_y;
138
139     return *this;
140 }
141
142 inline Vector2D& Vector2D::operator-=(const Vector2D& b)
143 {
144     this->m_x -= (double)b.m_x;
145     this->m_y -= (double)b.m_y;
146
147     return *this;
148 }
149
150 inline Vector2D& Vector2D::operator*=(const float& k)
151 {
152     this->m_x *= (double)k;
153     this->m_y *= (double)k;
154
155     return *this;
156 }
157

```

```

179
180 inline Vector2D& Vector2D::operator/=(const float& k)
181 {
182     if (compareFloats(k, 0.0f, EPSILON))
183     {
184         return *this;
185     }
186
187     this->m_x /= (double)k;
188     this->m_y /= (double)k;
189
190     return *this;
191 }
192
193 //-----
194
195 //-----
196 //Vector2D Non-member functions
197
198 inline bool zeroVector(const Vector2D& a)
199 {
200     if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON))
201     {
202         return true;
203     }
204
205     return false;
206 }
207
208 inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
209 {
210     return Vector2D((double)a.x() + b.x(), (double)a.y() + b.y());
211 }
212
213 inline Vector2D operator-(const Vector2D& v)
214 {
215     return Vector2D(-v.x(), -v.y());
216 }
217
218 inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
219 {
220     return Vector2D((double)a.x() - b.x(), (double)a.y() - b.y());
221 }
222
223 inline Vector2D operator*(const Vector2D& a, const float& k)
224 {
225     return Vector2D((double)a.x() * k, (double)a.y() * k);
226 }
227
228 inline Vector2D operator*(const float& k, const Vector2D& a)
229 {
230     return Vector2D((double)k * a.x(), (double)k * a.y());
231 }
232
233 inline Vector2D operator/(const Vector2D& a, const float& k)
234 {
235     if (compareFloats(k, 0.0f, EPSILON))
236     {
237         return Vector2D();
238     }
239
240     return Vector2D(a.x() / (double)k, a.y() / (double)k);
241 }
242
243 inline float dotProduct(const Vector2D& a, const Vector2D& b)
244 {
245     return (double)a.x() * b.x() + (double)a.y() * b.y();
246 }
247
248 inline float length(const Vector2D& v)
249 {
250     return sqrt((double)v.x() * v.x() + (double)v.y() * v.y());
251 }
252
253 inline Vector2D norm(const Vector2D& v)
254 {
255     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
256
257     //v is the zero vector
258     if (zeroVector(v))
259     {
260         return v;
261     }
262
263     double mag = length(v);
264
265     return Vector2D(v.x() / mag, v.y() / mag);
266 }

```

```

291     }
292
293     inline Vector2D PolarToCartesian(const Vector2D& v)
294     {
295         //v = (r, theta)
296         //x = rcos(theta)
297         //y = rsin(theta)
298         float angle = v.y() * PI / 180.0f;
299
300         return Vector2D(v.x() * cos(angle), v.x() * sin(angle));
301     }
302
303     inline Vector2D CartesianToPolar(const Vector2D& v)
304     {
305         //v = (x, y)
306         //r = sqrt(vx^2 + vy^2)
307         //theta = arctan(y / x)
308
309         if (compareFloats(v.x(), 0.0f, EPSILON))
310         {
311             return v;
312         }
313
314         double theta{ atan2(v.y(), v.x()) * 180.0 / PI };
315         return Vector2D(length(v), theta);
316     }
317
318     inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
319     {
320         //Projb(a) = (a dot b)b
321         //normalize b before projecting
322
323         Vector2D normB(norm(b));
324         return Vector2D(dotProduct(a, normB) * normB);
325     }
326
327     #if defined(_DEBUG)
328     inline void print(const Vector2D& v)
329     {
330         std::cout << "(" << v.x() << ", " << v.y() << ")";
331     }
332     #endif
333
334     //-----
335
336     //-----
337
338     //-----
339
340     class Vector3D
341     {
342     public:
343
344         Vector3D();
345
346         Vector3D(float x, float y, float z);
347
348         float& x();
349
350         float& y();
351
352         float& z();
353
354         const float& x() const;
355
356         const float& y() const;
357
358         const float& z() const;
359
360         Vector3D& operator+=(const Vector3D& b);
361
362         Vector3D& operator-=(const Vector3D& b);
363
364         Vector3D& operator*=(const float& k);
365
366         Vector3D& operator/=(const float& k);
367
368     private:
369         float m_x;
370         float m_y;
371         float m_z;
372     };
373
374     //-----

```



```

423 //Vector3D Constructors
424
425 inline Vector3D::Vector3D() : m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }
426 {}
427
428 inline Vector3D::Vector3D(float x, float y, float z) : m_x{ x }, m_y{ y }, m_z{ z }
429 {}
430
431 //-----
432
433 //-----
434 //Vector3D Getters and Setters
435
436 inline float& Vector3D::x()
437 {
438     return m_x;
439 }
440
441 inline float& Vector3D::y()
442 {
443     return m_y;
444 }
445
446 inline float& Vector3D::z()
447 {
448     return m_z;
449 }
450
451 inline const float& Vector3D::x() const
452 {
453     return m_x;
454 }
455
456 inline const float& Vector3D::y() const
457 {
458     return m_y;
459 }
460
461 inline const float& Vector3D::z() const
462 {
463     return m_z;
464 }
465 //-----
466
467 //-----
468 //Vector3D Member functions
469
470 inline Vector3D& Vector3D::operator+=(const Vector3D& b)
471 {
472     {
473         this->m_x += (double)b.m_x;
474         this->m_y += (double)b.m_y;
475         this->m_z += (double)b.m_z;
476     }
477     return *this;
478 }
479
480 inline Vector3D& Vector3D::operator-=(const Vector3D& b)
481 {
482     {
483         this->m_x -= (double)b.m_x;
484         this->m_y -= (double)b.m_y;
485         this->m_z -= (double)b.m_z;
486     }
487     return *this;
488 }
489
490 inline Vector3D& Vector3D::operator*=(const float& k)
491 {
492     {
493         this->m_x *= (double)k;
494         this->m_y *= (double)k;
495         this->m_z *= (double)k;
496     }
497     return *this;
498 }
499
500 inline Vector3D& Vector3D::operator/=(const float& k)
501 {
502     {
503         if (compareFloats(k, 0.0f, EPSILON))
504         {
505             return *this;
506         }
507         this->m_x /= (double)k;
508         this->m_y /= (double)k;
509         this->m_z /= (double)k;
510     }
511     return *this;
512 }

```

```

510     }
511
512     //-----
513
514     //-----
515     //Vector3D Non-member functions
516
517     inline bool zeroVector(const Vector3D& a)
518     {
519         if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON) &&
520             compareFloats(a.z(), 0.0f, EPSILON))
521         {
522             return true;
523         }
524         return false;
525     }
526
527     inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
528     {
529         return Vector3D((double)a.x() + b.x(), (double)a.y() + b.y(), (double)a.z() + b.z());
530     }
531
532     inline Vector3D operator-(const Vector3D& v)
533     {
534         return Vector3D(-v.x(), -v.y(), -v.z());
535     }
536
537     inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
538     {
539         return Vector3D(a.x() - b.x(), a.y() - b.y(), a.z() - b.z());
540     }
541
542     inline Vector3D operator*(const Vector3D& a, const float& k)
543     {
544         return Vector3D(a.x() * (double)k, a.y() * (double)k, a.z() * (double)k);
545     }
546
547     inline Vector3D operator*(const float& k, const Vector3D& a)
548     {
549         return Vector3D((double)k * a.x(), (double)k * a.y(), (double)k * a.z());
550     }
551
552     inline Vector3D operator/(const Vector3D& a, const float& k)
553     {
554         if (compareFloats(k, 0.0f, EPSILON))
555         {
556             return Vector3D();
557         }
558         return Vector3D(a.x() / (double)k, a.y() / (double)k, a.z() / (double)k);
559     }
560
561     inline float dotProduct(const Vector3D& a, const Vector3D& b)
562     {
563         //a dot b = axbx + ayby + azbz
564         return (double)a.x() * b.x() + (double)a.y() * b.y() + (double)a.z() * b.z();
565     }
566
567     inline Vector3D crossProduct(const Vector3D& a, const Vector3D& b)
568     {
569         //a x b = (aybz - azby, azbx - axbz, axby - aybx)
570
571         return Vector3D((double)a.y() * b.z() - (double)a.z() * b.y(),
572             (double)a.z() * b.x() - (double)a.x() * b.z(),
573             (double)a.x() * b.y() - (double)a.y() * b.x());
574     }
575
576     inline float length(const Vector3D& v)
577     {
578         //length(v) = sqrt(vx^2 + vy^2 + vz^2)
579
580         return sqrt((double)v.x() * v.x() + (double)v.y() * v.y() + (double)v.z() * v.z());
581     }
582
583     inline Vector3D norm(const Vector3D& v)
584     {
585         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
586         //v is the zero vector
587         if (zeroVector(v))
588         {
589             return v;
590         }
591
592         double mag = length(v);
593
594         return Vector3D(v.x() / mag, v.y() / mag, v.z() / mag);
595     }

```

```

624     }
625
630     inline Vector3D CylindricalToCartesian(const Vector3D& v)
631     {
632         //v = (r, theta, z)
633         //x = rcos(theta)
634         //y = rsin(theta)
635         //z = z
636         double angle = v.y() * PI / 180.0;
637
638         return Vector3D(v.x() * cos(angle), v.x() * sin(angle), v.z());
639     }
640
646     inline Vector3D CartesianToCylindrical(const Vector3D& v)
647     {
648         //v = (x, y, z)
649         //r = sqrt(vx^2 + vy^2 + vz^2)
650         //theta = arctan(y / x)
651         //z = z
652         if (compareFloats(v.x(), 0.0f, EPSILON))
653         {
654             return v;
655         }
656
657         double theta{ atan2(v.y(), v.x()) * 180.0 / PI };
658         return Vector3D(length(v), theta, v.z());
659     }
660
665     inline Vector3D SphericalToCartesian(const Vector3D& v)
666     {
667         // v = (pho, phi, theta)
668         //x = pho * sin(phi) * cos(theta)
669         //y = pho * sin(phi) * sin(theta)
670         //z = pho * cos(theta);
671
672         double phi{ v.y() * PI / 180.0 };
673         double theta{ v.z() * PI / 180.0 };
674
675         return Vector3D(v.x() * sin(phi) * cos(theta), v.x() * sin(phi) * sin(theta), v.x() *
cos(theta));
676     }
677
682     inline Vector3D CartesianToSpherical(const Vector3D& v)
683     {
684         //v = (x, y, z)
685         //pho = sqrt(vx^2 + vy^2 + vz^2)
686         //phi = acos(z / pho)
687         //theta = arctan(y / x)
688
689         if (compareFloats(v.x(), 0.0f, EPSILON) || zeroVector(v))
690         {
691             return v;
692         }
693
694         double pho{ length(v) };
695         double phi{ acos(v.z() / pho) * 180.0 / PI };
696         double theta{ atan2(v.y(), v.x()) * 180.0 / PI };
697
698         return Vector3D(pho, phi, theta);
699     }
700
704     inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
705     {
706         //Projb(a) = (a dot b)b
707         //normalize b before projecting
708
709         Vector3D normB(norm(b));
710         return Vector3D(dotProduct(a, normB) * normB);
711     }
712
713
714 #if defined(_DEBUG)
715     inline void print(const Vector3D& v)
716     {
717         std::cout << "(" << v.x() << ", " << v.y() << ", " << v.z() << ")";
718     }
719 #endif
720     //-----
721
722
723
724
725
726
727     //-----
732     class Vector4D

```

```

733     {
734     public:
735         Vector4D();
736
737         Vector4D(float x, float y, float z, float w);
738
739         Vector4D(Vector2D v, float z = 0.0f, float w = 0.0f);
740
741         Vector4D(Vector3D v, float w = 0.0f);
742
743         float& x();
744
745         float& y();
746
747         float& z();
748
749         float& w();
750
751         const float& x() const;
752
753         const float& y() const;
754
755         const float& z() const;
756
757         const float& w() const;
758
759         Vector4D& operator+=(const Vector4D& b);
760
761         Vector4D& operator-=(const Vector4D& b);
762
763         Vector4D& operator*=(const float& k);
764
765         Vector4D& operator/=(const float& k);
766
767     private:
768         float m_x;
769         float m_y;
770         float m_z;
771         float m_w;
772     };
773
774     //-----
775     //Vector4D Constructors
776
777     inline Vector4D::Vector4D() : m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }, m_w{ 0.0f }
778     {}
779
780     inline Vector4D::Vector4D(float x, float y, float z, float w) : m_x{ x }, m_y{ y }, m_z{ z }, m_w{
781     w }
782     {}
783
784     inline Vector4D::Vector4D(Vector2D v, float z, float w) : m_x{ v.x() }, m_y{ v.y() }, m_z{ z },
785     m_w{ w }
786     {}
787
788     inline Vector4D::Vector4D(Vector3D v, float w) : m_x{ v.x() }, m_y{ v.y() }, m_z{ v.z() }, m_w{ w }
789     {}
790
791     //-----
792     //-----
793     //Vector4D Getters and Setters
794
795     inline float& Vector4D::x()
796     {
797         return m_x;
798     }
799
800     inline float& Vector4D::y()
801     {
802         return m_y;
803     }
804
805     inline float& Vector4D::z()
806     {
807         return m_z;
808     }
809
810     inline float& Vector4D::w()
811     {
812         return m_w;
813     }
814
815     inline const float& Vector4D::x() const
816     {
817         return m_x;
818     }

```

```

860
861     inline const float& Vector4D::y() const
862 {
863     return m_y;
864 }
865
866     inline const float& Vector4D::z() const
867 {
868     return m_z;
869 }
870
871     inline const float& Vector4D::w() const
872 {
873     return m_w;
874 }
875 //-----
876
877 //-----
878 //Vector4D Member functions
879
880
881     inline Vector4D& Vector4D::operator+=(const Vector4D& b)
882 {
883     this->m_x += (double)b.m_x;
884     this->m_y += (double)b.m_y;
885     this->m_z += (double)b.m_z;
886     this->m_w += (double)b.m_w;
887
888     return *this;
889 }
890
891     inline Vector4D& Vector4D::operator-=(const Vector4D& b)
892 {
893     this->m_x -= (double)b.m_x;
894     this->m_y -= (double)b.m_y;
895     this->m_z -= (double)b.m_z;
896     this->m_w -= (double)b.m_w;
897
898     return *this;
899 }
900
901     inline Vector4D& Vector4D::operator*=(const float& k)
902 {
903     this->m_x *= (double)k;
904     this->m_y *= (double)k;
905     this->m_z *= (double)k;
906     this->m_w *= (double)k;
907
908     return *this;
909 }
910
911     inline Vector4D& Vector4D::operator/=(const float& k)
912 {
913     if (compareFloats(k, 0.0f, EPSILON))
914     {
915         return *this;
916     }
917
918     this->m_x /= (double)k;
919     this->m_y /= (double)k;
920     this->m_z /= (double)k;
921     this->m_w /= (double)k;
922
923     return *this;
924 }
925
926 //-----
927
928 //-----
929 //Vector4D Non-member functions
930
931
932     inline bool zeroVector(const Vector4D& a)
933 {
934     if (compareFloats(a.x(), 0.0f, EPSILON) && compareFloats(a.y(), 0.0f, EPSILON) &&
935         compareFloats(a.z(), 0.0f, EPSILON) && compareFloats(a.w(), 0.0f, EPSILON))
936     {
937         return true;
938     }
939
940     return false;
941 }
942
943
944     inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
945 {
946     return Vector4D((double)a.x() + b.x(), (double)a.y() + b.y(), (double)a.z() + b.z(),
947 (double)a.w() + b.w());
948 }
949

```

```

950
953     inline Vector4D operator-(const Vector4D& v)
954     {
955         return Vector4D(-v.x(), -v.y(), -v.z(), -v.w());
956     }
957
960     inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
961     {
962         return Vector4D((double)a.x() - b.x(), (double)a.y() - b.y(), (double)a.z() - b.z(),
(double)a.w() - b.w());
963     }
964
968     inline Vector4D operator*(const Vector4D& a, const float& k)
969     {
970         return Vector4D(a.x() * (double)k, a.y() * (double)k, a.z() * (double)k, a.w() * (double)k);
971     }
972
976     inline Vector4D operator*(const float& k, const Vector4D& a)
977     {
978         return Vector4D((double)k * a.x(), (double)k * a.y(), (double)k * a.z(), (double)k * a.w());
979     }
980
985     inline Vector4D operator/(const Vector4D& a, const float& k)
986     {
987         if (compareFloats(k, 0.0f, EPSILON))
988         {
989             return Vector4D();
990         }
991
992         return Vector4D(a.x() / (double)k, a.y() / (double)k, a.z() / (double)k, a.w() / (double)k);
993     }
994
997     inline float dotProduct(const Vector4D& a, const Vector4D& b)
998     {
999         //a dot b = axbx + ayby + azbz + awbw
1000         return (double)a.x() * b.x() + (double)a.y() * b.y() + (double)a.z() * b.z() + (double)a.w() *
b.w();
1001     }
1002
1005     inline float length(const Vector4D& v)
1006     {
1007         //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
1008         return sqrt((double)v.x() * v.x() + (double)v.y() * v.y() + (double)v.z() * v.z() +
(double)v.w() * v.w());
1009     }
1010
1014     inline Vector4D norm(const Vector4D& v)
1015     {
1016         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
1017         //v is the zero vector
1018         if (zeroVector(v))
1019         {
1020             return v;
1021         }
1022
1023         double mag = length(v);
1024
1025         return Vector4D(v.x() / mag, v.y() / mag, v.z() / mag, v.w() / mag);
1026     }
1027
1031     inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
1032     {
1033         //Projb(a) = (a dot b)b
1034         //normalize b before projecting
1035         Vector4D normB(norm(b));
1036         return Vector4D(dotProduct(a, normB) * normB);
1037     }
1038
1039
1040 #if defined(_DEBUG)
1041     inline void print(const Vector4D& v)
1042     {
1043         std::cout << "(" << v.x() << ", " << v.y() << ", " << v.z() << ", " << v.w() << ")";
1044     }
1045 #endif
1046     //-----
1047
1048
1049
1050
1051
1052 //-----
1059     class Matrix4x4
1060     {
1061     public:
1062

```

```

1067         Matrix4x4();
1068
1074         Matrix4x4(float a[][4]);
1075
1078         float* data();
1079
1082         const float* data() const;
1083
1087         const float& operator()(unsigned int row, unsigned int col) const;
1088
1092         float& operator()(unsigned int row, unsigned int col);
1093
1097         void setRow(unsigned int row, Vector4D v);
1098
1102         void setCol(unsigned int col, Vector4D v);
1103
1106         Matrix4x4& operator+=(const Matrix4x4& m);
1107
1110         Matrix4x4& operator-=(const Matrix4x4& m);
1111
1114         Matrix4x4& operator*=(const float& k);
1115
1116
1119         Matrix4x4& operator*=(const Matrix4x4& m);
1120
1121     private:
1122
1123         float m_mat[4][4];
1124     };
1125
1126     //-----
1127     inline Matrix4x4::Matrix4x4()
1128     {
1129         //1st row
1130         m_mat[0][0] = 1.0f;
1131         m_mat[0][1] = 0.0f;
1132         m_mat[0][2] = 0.0f;
1133         m_mat[0][3] = 0.0f;
1134
1135         //2nd
1136         m_mat[1][0] = 0.0f;
1137         m_mat[1][1] = 1.0f;
1138         m_mat[1][2] = 0.0f;
1139         m_mat[1][3] = 0.0f;
1140
1141         //3rd row
1142         m_mat[2][0] = 0.0f;
1143         m_mat[2][1] = 0.0f;
1144         m_mat[2][2] = 1.0f;
1145         m_mat[2][3] = 0.0f;
1146
1147         //4th row
1148         m_mat[3][0] = 0.0f;
1149         m_mat[3][1] = 0.0f;
1150         m_mat[3][2] = 0.0f;
1151         m_mat[3][3] = 1.0f;
1152     }
1153
1154
1155
1156     inline Matrix4x4::Matrix4x4(float a[][4])
1157     {
1158         //1st row
1159         m_mat[0][0] = a[0][0];
1160         m_mat[0][1] = a[0][1];
1161         m_mat[0][2] = a[0][2];
1162         m_mat[0][3] = a[0][3];
1163
1164         //2nd
1165         m_mat[1][0] = a[1][0];
1166         m_mat[1][1] = a[1][1];
1167         m_mat[1][2] = a[1][2];
1168         m_mat[1][3] = a[1][3];
1169
1170         //3rd row
1171         m_mat[2][0] = a[2][0];
1172         m_mat[2][1] = a[2][1];
1173         m_mat[2][2] = a[2][2];
1174         m_mat[2][3] = a[2][3];
1175
1176         //4th row
1177         m_mat[3][0] = a[3][0];
1178         m_mat[3][1] = a[3][1];
1179         m_mat[3][2] = a[3][2];
1180         m_mat[3][3] = a[3][3];
1181     }
1182

```

```

1183     inline float* Matrix4x4::data()
1184     {
1185         return m_mat[0];
1186     }
1187
1188     inline const float* Matrix4x4::data()const
1189     {
1190         return m_mat[0];
1191     }
1192
1193     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
1194     {
1195         if (row > 3 || col > 3)
1196         {
1197             return m_mat[0][0];
1198         }
1199         else
1200         {
1201             return m_mat[row][col];
1202         }
1203     }
1204
1205     inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
1206     {
1207         if (row > 3 || col > 3)
1208         {
1209             return m_mat[0][0];
1210         }
1211         else
1212         {
1213             return m_mat[row][col];
1214         }
1215     }
1216
1217     inline void Matrix4x4::setRow(unsigned int row, Vector4D v)
1218     {
1219         if (row < 0 || row > 3)
1220         {
1221             m_mat[0][0] = v.x();
1222             m_mat[0][1] = v.y();
1223             m_mat[0][2] = v.z();
1224             m_mat[0][3] = v.w();
1225         }
1226         else
1227         {
1228             m_mat[row][0] = v.x();
1229             m_mat[row][1] = v.y();
1230             m_mat[row][2] = v.z();
1231             m_mat[row][3] = v.w();
1232         }
1233     }
1234
1235     inline void Matrix4x4::setCol(unsigned int col, Vector4D v)
1236     {
1237         if (col < 0 || col > 3)
1238         {
1239             m_mat[0][0] = v.x();
1240             m_mat[1][0] = v.y();
1241             m_mat[2][0] = v.z();
1242             m_mat[3][0] = v.w();
1243         }
1244         else
1245         {
1246             m_mat[0][col] = v.x();
1247             m_mat[1][col] = v.y();
1248             m_mat[2][col] = v.z();
1249             m_mat[3][col] = v.w();
1250         }
1251     }
1252
1253     inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
1254     {
1255         for (int i = 0; i < 4; ++i)
1256         {
1257             for (int j = 0; j < 4; ++j)
1258             {
1259                 this->m_mat[i][j] += (double)m.m_mat[i][j];
1260             }
1261         }
1262
1263         return *this;
1264     }
1265
1266     inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
1267     {
1268         for (int i = 0; i < 4; ++i)
1269         {

```



```

1270         for (int j = 0; j < 4; ++j)
1271         {
1272             this->m_mat[i][j] -= (double)m.m_mat[i][j];
1273         }
1274     }
1275
1276     return *this;
1277 }
1278
1279 inline Matrix4x4& Matrix4x4::operator*=(const float& k)
1280 {
1281     for (int i = 0; i < 4; ++i)
1282     {
1283         for (int j = 0; j < 4; ++j)
1284         {
1285             this->m_mat[i][j] *= (double)k;
1286         }
1287     }
1288
1289     return *this;
1290 }
1291
1292 inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
1293 {
1294     Matrix4x4 res;
1295
1296     for (int i = 0; i < 4; ++i)
1297     {
1298         res.m_mat[i][0] = ((double)m_mat[i][0] * m.m_mat[0][0]) +
1299             ((double)m_mat[i][1] * m.m_mat[1][0]) +
1300             ((double)m_mat[i][2] * m.m_mat[2][0]) +
1301             ((double)m_mat[i][3] * m.m_mat[3][0]);
1302
1303         res.m_mat[i][1] = ((double)m_mat[i][0] * m.m_mat[0][1]) +
1304             ((double)m_mat[i][1] * m.m_mat[1][1]) +
1305             ((double)m_mat[i][2] * m.m_mat[2][1]) +
1306             ((double)m_mat[i][3] * m.m_mat[3][1]);
1307
1308         res.m_mat[i][2] = ((double)m_mat[i][0] * m.m_mat[0][2]) +
1309             ((double)m_mat[i][1] * m.m_mat[1][2]) +
1310             ((double)m_mat[i][2] * m.m_mat[2][2]) +
1311             ((double)m_mat[i][3] * m.m_mat[3][2]);
1312
1313         res.m_mat[i][3] = ((double)m_mat[i][0] * m.m_mat[0][3]) +
1314             ((double)m_mat[i][1] * m.m_mat[1][3]) +
1315             ((double)m_mat[i][2] * m.m_mat[2][3]) +
1316             ((double)m_mat[i][3] * m.m_mat[3][3]);
1317     }
1318
1319     for (int i = 0; i < 4; ++i)
1320     {
1321         for (int j = 0; j < 4; ++j)
1322         {
1323             m_mat[i][j] = res.m_mat[i][j];
1324         }
1325     }
1326
1327     return *this;
1328 }
1329
1330 inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
1331 {
1332     Matrix4x4 res;
1333     for (int i = 0; i < 4; ++i)
1334     {
1335         for (int j = 0; j < 4; ++j)
1336         {
1337             res(i, j) = (double)m1(i, j) + m2(i, j);
1338         }
1339     }
1340
1341     return res;
1342 }
1343
1344 inline Matrix4x4 operator-(const Matrix4x4& m)
1345 {
1346     Matrix4x4 res;
1347     for (int i = 0; i < 4; ++i)
1348     {
1349         for (int j = 0; j < 4; ++j)
1350         {
1351             res(i, j) = -m(i, j);
1352         }
1353     }
1354
1355     return res;
1356 }
1357
1358 inline Matrix4x4 operator-=(const Matrix4x4& m)
1359 {
1360     Matrix4x4 res;

```

```

1361
1362 inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
1363 {
1364     Matrix4x4 res;
1365     for (int i = 0; i < 4; ++i)
1366     {
1367         for (int j = 0; j < 4; ++j)
1368         {
1369             res(i, j) = (double)m1(i, j) - m2(i, j);
1370         }
1371     }
1372     return res;
1373 }
1374
1375 inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
1376 {
1377     Matrix4x4 res;
1378     for (int i = 0; i < 4; ++i)
1379     {
1380         for (int j = 0; j < 4; ++j)
1381         {
1382             res(i, j) = (double)m(i, j) * k;
1383         }
1384     }
1385     return res;
1386 }
1387
1388 inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
1389 {
1390     Matrix4x4 res;
1391     for (int i = 0; i < 4; ++i)
1392     {
1393         for (int j = 0; j < 4; ++j)
1394         {
1395             res(i, j) = k * (double)m(i, j);
1396         }
1397     }
1398     return res;
1399 }
1400
1401 inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
1402 {
1403     Matrix4x4 res;
1404     for (int i = 0; i < 4; ++i)
1405     {
1406         res(i, 0) = ((double)m1(i, 0) * m2(0, 0)) +
1407                     ((double)m1(i, 1) * m2(1, 0)) +
1408                     ((double)m1(i, 2) * m2(2, 0)) +
1409                     ((double)m1(i, 3) * m2(3, 0));
1410         res(i, 1) = ((double)m1(i, 0) * m2(0, 1)) +
1411                     ((double)m1(i, 1) * m2(1, 1)) +
1412                     ((double)m1(i, 2) * m2(2, 1)) +
1413                     ((double)m1(i, 3) * m2(3, 1));
1414         res(i, 2) = ((double)m1(i, 0) * m2(0, 2)) +
1415                     ((double)m1(i, 1) * m2(1, 2)) +
1416                     ((double)m1(i, 2) * m2(2, 2)) +
1417                     ((double)m1(i, 3) * m2(3, 2));
1418         res(i, 3) = ((double)m1(i, 0) * m2(0, 3)) +
1419                     ((double)m1(i, 1) * m2(1, 3)) +
1420                     ((double)m1(i, 2) * m2(2, 3)) +
1421                     ((double)m1(i, 3) * m2(3, 3));
1422     }
1423     return res;
1424 }
1425
1426 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
1427 {
1428     Vector4D res;
1429     res.x() = ((double)m(0, 0) * v.x() + (double)m(0, 1) * v.y() + (double)m(0, 2) * v.z() +
1430 (double)m(0, 3) * v.w());
1431     res.y() = ((double)m(1, 0) * v.x() + (double)m(1, 1) * v.y() + (double)m(1, 2) * v.z() +
1432 (double)m(1, 3) * v.w());
1433     res.z() = ((double)m(2, 0) * v.x() + (double)m(2, 1) * v.y() + (double)m(2, 2) * v.z() +
1434 (double)m(2, 3) * v.w());
1435     res.w() = ((double)m(3, 0) * v.x() + (double)m(3, 1) * v.y() + (double)m(3, 2) * v.z() +
1436 (double)m(3, 3) * v.w());
1437     return res;
1438 }
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454

```

```

1455     }
1456
1460     inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
1461     {
1462         Vector4D res;
1463
1464         res.x() = ((double)v.x() * m(0, 0) + (double)v.y() * m(1, 0) + (double)v.z() * m(2, 0) +
1465 (double)v.w() * m(3, 0));
1466         res.y() = ((double)v.x() * m(0, 1) + (double)v.y() * m(1, 1) + (double)v.z() * m(2, 1) +
1467 (double)v.w() * m(3, 1));
1468         res.z() = ((double)v.x() * m(0, 2) + (double)v.y() * m(1, 2) + (double)v.z() * m(2, 2) +
1469 (double)v.w() * m(3, 2));
1470         res.w() = ((double)v.x() * m(0, 3) + (double)v.y() * m(1, 3) + (double)v.z() * m(2, 3) +
1471 (double)v.w() * m(3, 3));
1472
1473         return res;
1474     }
1475
1476     inline void setToIdentity(Matrix4x4& m)
1477     {
1478         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1479
1480         //1st row
1481         m(0, 0) = 1.0f;
1482         m(0, 1) = 0.0f;
1483         m(0, 2) = 0.0f;
1484         m(0, 3) = 0.0f;
1485
1486         //2nd row
1487         m(1, 0) = 0.0f;
1488         m(1, 1) = 1.0f;
1489         m(1, 2) = 0.0f;
1490         m(1, 3) = 0.0f;
1491
1492         //3rd row
1493         m(2, 0) = 0.0f;
1494         m(2, 1) = 0.0f;
1495         m(2, 2) = 1.0f;
1496         m(2, 3) = 0.0f;
1497
1498         //4th row
1499         m(3, 0) = 0.0f;
1500         m(3, 1) = 0.0f;
1501         m(3, 2) = 0.0f;
1502         m(3, 3) = 1.0f;
1503     }
1504
1505     inline bool isIdentity(const Matrix4x4& m)
1506     {
1507         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
1508         0.0f
1509
1510         for (int i = 0; i < 4; ++i)
1511         {
1512             for (int j = 0; j < 4; ++j)
1513             {
1514                 if (i == j)
1515                 {
1516                     if (!compareFloats(m(i, j), 1.0f, EPSILON))
1517                     {
1518                         return false;
1519                     }
1520                 }
1521                 else
1522                 {
1523                     if (!compareFloats(m(i, j), 0.0f, EPSILON))
1524                     {
1525                         return false;
1526                     }
1527                 }
1528             }
1529         }
1530
1531         return true;
1532     }
1533
1534     inline Matrix4x4 transpose(const Matrix4x4& m)
1535     {
1536         //make the rows into cols
1537
1538         Matrix4x4 res;
1539
1540         //1st col = 1st row
1541         res(0, 0) = m(0, 0);
1542         res(1, 0) = m(0, 1);
1543         res(2, 0) = m(0, 2);
1544         res(3, 0) = m(0, 3);
1545
1546         //2nd col = 2nd row

```

```

1546         res(0, 1) = m(1, 0);
1547         res(1, 1) = m(1, 1);
1548         res(2, 1) = m(1, 2);
1549         res(3, 1) = m(1, 3);
1550
1551         //3rd col = 3rd row
1552         res(0, 2) = m(2, 0);
1553         res(1, 2) = m(2, 1);
1554         res(2, 2) = m(2, 2);
1555         res(3, 2) = m(2, 3);
1556
1557         //4th col = 4th row
1558         res(0, 3) = m(3, 0);
1559         res(1, 3) = m(3, 1);
1560         res(2, 3) = m(3, 2);
1561         res(3, 3) = m(3, 3);
1562
1563         return res;
1564     }
1565
1566     inline Matrix4x4 translate(const Matrix4x4& cm, float x, float y, float z)
1567     {
1568         //1 0 0 0
1569         //0 1 0 0
1570         //0 0 1 0
1571         //x y z 1
1572
1573         Matrix4x4 t;
1574         t(3, 0) = x;
1575         t(3, 1) = y;
1576         t(3, 2) = z;
1577
1578         return cm * t;
1579     }
1580
1581     inline Matrix4x4 scale(const Matrix4x4& cm, float x, float y, float z)
1582     {
1583         //x 0 0 0
1584         //0 y 0 0
1585         //0 0 z 0
1586         //0 0 0 1
1587
1588         Matrix4x4 s;
1589         s(0, 0) = x;
1590         s(1, 1) = y;
1591         s(2, 2) = z;
1592
1593         return cm * s;
1594     }
1595
1596     inline Matrix4x4 rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
1597     {
1598         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy      0
1599         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx      0
1600         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2      0
1601         //0                    0                    0                    1
1602         //c = cos(angle)
1603         //s = sin(angle)
1604
1605         double c = cos(angle * PI / 180.0);
1606         double s = sin(angle * PI / 180.0);
1607
1608         Matrix4x4 r;
1609
1610         //1st row
1611         r(0, 0) = c + (1.0 - c) * ((double)x * x);
1612         r(0, 1) = (1.0 - c) * ((double)x * y) + (s * z);
1613         r(0, 2) = (1.0 - c) * ((double)x * z) - (s * y);
1614
1615         //2nd row
1616         r(1, 0) = (1.0 - c) * ((double)x * y) - (s * z);
1617         r(1, 1) = c + (1.0 - c) * ((double)y * y);
1618         r(1, 2) = (1.0 - c) * ((double)y * z) + (s * x);
1619
1620         //3rd row
1621         r(2, 0) = (1.0 - c) * ((double)x * z) + (s * y);
1622         r(2, 1) = (1.0 - c) * ((double)y * z) - (s * x);
1623         r(2, 2) = c + (1.0 - c) * ((double)z * z);
1624
1625         return cm * r;
1626     }
1627
1628     inline double det(const Matrix4x4& m)
1629     {
1630         //m00m11(m22m33 - m23m32)
1631         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);
1644
1645    //m00m12(m23m31 - m21m33)
1646    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);
1647
1648    //m00m13(m21m32 - m22m31)
1649    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);
1650
1651    //m01m10(m22m33 - m23m32)
1652    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);
1653
1654    //m01m12(m23m30 - m20m33)
1655    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);
1656
1657    //m01m13(m20m32 - m22m30)
1658    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);
1659
1660    //m02m10(m21m33 - m23m31)
1661    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);
1662
1663    //m02m11(m23m30 - m20m33)
1664    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);
1665
1666    //m02m13(m20m31 - m21m30)
1667    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);
1668
1669    //m03m10(m21m32 - m22m21)
1670    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);
1671
1672    //m03m11(m22m30 - m20m32)
1673    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);
1674
1675    //m03m12(m20m31 - m21m30)
1676    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);
1677
1678    return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
1679 }
1680
1681 inline double cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
1682 {
1683     //cij = (-1)^(i + j) * det of minor(i, j);
1684     double tempMat[3][3];
1685     int tr{ 0 };
1686     int tc{ 0 };
1687
1688     //minor(i, j)
1689     for (int i = 0; i < 4; ++i)
1690     {
1691         if (i == row)
1692             continue;
1693
1694         for (int j = 0; j < 4; ++j)
1695         {
1696             if (j == col)
1697                 continue;
1698
1699             tempMat[tr][tc] = m(i, j);
1700             ++tc;
1701         }
1702         tc = 0;
1703         ++tr;
1704     }
1705
1706     //determinant of minor(i, j)
1707     double det3x3 = (tempMat[0][0] * tempMat[1][1] * tempMat[2][2]) + (tempMat[0][1] *
    tempMat[1][2] * tempMat[2][0]) +
1708     (tempMat[0][2] * tempMat[1][0] * tempMat[2][1]) - (tempMat[0][2] * tempMat[1][1] *
    tempMat[2][0]) -
1709     (tempMat[0][1] * tempMat[1][0] * tempMat[2][2]) - (tempMat[0][0] * tempMat[1][2] *
    tempMat[2][1]);
1710
1711     return pow(-1, row + col) * det3x3;
1712 }
1713
1714 inline Matrix4x4 adjoint(const Matrix4x4& m)

```

```

1720     {
1721         //Cofactor of each ijth position put into matrix cA.
1722         //Adjoint is the tranposed matrix of cA.
1723         Matrix4x4 cA;
1724         for (int i = 0; i < 4; ++i)
1725         {
1726             for (int j = 0; j < 4; ++j)
1727             {
1728                 cA(i, j) = cofactor(m, i, j);
1729             }
1730         }
1731
1732         return transpose(cA);
1733     }
1734
1735     inline Matrix4x4 inverse(const Matrix4x4& m)
1736     {
1737         //Inverse of m = adjoint of m / det of m
1738         double determinant = det(m);
1739         if (compareDoubles(determinant, 0.0, EPSILON))
1740             return Matrix4x4();
1741
1742         return adjoint(m) * (1.0 / determinant);
1743     }
1744
1745 #if defined(_DEBUG)
1746     inline void print(const Matrix4x4& m)
1747     {
1748         for (int i = 0; i < 4; ++i)
1749         {
1750             for (int j = 0; j < 4; ++j)
1751             {
1752                 std::cout << m(i, j) << " ";
1753             }
1754             std::cout << std::endl;
1755         }
1756     }
1757 #endif
1758
1759 //-----
1760
1761 //-----
1762
1763 class Quaternion
1764 {
1765 public:
1766     Quaternion();
1767
1768     Quaternion(float scalar, float x, float y, float z);
1769
1770     Quaternion(float scalar, const Vector3D& v);
1771
1772     Quaternion(const Vector4D& v);
1773
1774     float& scalar();
1775
1776     const float& scalar() const;
1777
1778     float& x();
1779
1780     const float& x() const;
1781
1782     float& y();
1783
1784     const float& y() const;
1785
1786     float& z();
1787
1788     const float& z() const;
1789
1790     Vector3D vector();
1791
1792     Quaternion& operator+=(const Quaternion& q);
1793
1794     Quaternion& operator-=(const Quaternion& q);
1795
1796     Quaternion& operator*=(float k);
1797
1798     Quaternion& operator*=(const Quaternion& q);
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854

```

```

1855
1856     private:
1857
1858         float m_scalar;
1859         float m_x;
1860         float m_y;
1861         float m_z;
1862     };
1863
1864     //-----
1865     inline Quaternion::Quaternion() : m_scalar{ 1.0f }, m_x{ 0.0f }, m_y{ 0.0f }, m_z{ 0.0f }
1866     {
1867     }
1868
1869     inline Quaternion::Quaternion(float scalar, float x, float y, float z) : m_scalar{ scalar }, m_x{
1870 x }, m_y{ y }, m_z{ z }
1871     {
1872     }
1873     inline Quaternion::Quaternion(float scalar, const Vector3D& v) : m_scalar{ scalar }, m_x{ v.x() },
1874 m_y{ v.y() }, m_z{ v.z() }
1875     {
1876     }
1877     inline Quaternion::Quaternion(const Vector4D& v) : m_scalar{ v.x() }, m_x{ v.y() }, m_y{ v.z() },
1878 m_z{ v.w() }
1879     {
1880     }
1881     inline float& Quaternion::scalar()
1882     {
1883         return m_scalar;
1884     }
1885
1886     inline const float& Quaternion::scalar()const
1887     {
1888         return m_scalar;
1889     }
1890
1891     inline float& Quaternion::x()
1892     {
1893         return m_x;
1894     }
1895
1896     inline const float& Quaternion::x()const
1897     {
1898         return m_x;
1899     }
1900
1901     inline float& Quaternion::y()
1902     {
1903         return m_y;
1904     }
1905
1906     inline const float& Quaternion::y()const
1907     {
1908         return m_y;
1909     }
1910
1911     inline float& Quaternion::z()
1912     {
1913         return m_z;
1914     }
1915
1916     inline const float& Quaternion::z()const
1917     {
1918         return m_z;
1919     }
1920
1921     inline Vector3D Quaternion::vector()
1922     {
1923         return Vector3D(m_x, m_y, m_z);
1924     }
1925
1926     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
1927     {
1928         this->m_scalar += (double)q.m_scalar;
1929         this->m_x += (double)q.m_x;
1930         this->m_y += (double)q.m_y;
1931         this->m_z += (double)q.m_z;
1932
1933         return *this;
1934     }
1935
1936     inline Quaternion& Quaternion::operator-=(const Quaternion& q)
1937     {
1938         this->m_scalar -= (double)q.m_scalar;

```

```

1939         this->m_x -= (double)q.m_x;
1940         this->m_y -= (double)q.m_y;
1941         this->m_z -= (double)q.m_z;
1942
1943         return *this;
1944     }
1945
1946     inline Quaternion& Quaternion::operator*=(float k)
1947     {
1948         this->m_scalar *= (double)k;
1949         this->m_x *= (double)k;
1950         this->m_y *= (double)k;
1951         this->m_z *= (double)k;
1952
1953         return *this;
1954     }
1955
1956     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
1957     {
1958         Vector3D thisVector(this->m_x, this->m_y, this->m_z);
1959         Vector3D qVector(q.m_x, q.m_y, q.m_z);
1960
1961         double s{ (double)this->m_scalar * q.m_scalar };
1962         double dP{ dotProduct(thisVector, qVector) };
1963         double resultScalar{ s - dP };
1964
1965         Vector3D a(this->m_scalar * qVector);
1966         Vector3D b(q.m_scalar * thisVector);
1967         Vector3D cP(crossProduct(thisVector, qVector));
1968         Vector3D resultVector(a + b + cP);
1969
1970         this->m_scalar = resultScalar;
1971         this->m_x = resultVector.x();
1972         this->m_y = resultVector.y();
1973         this->m_z = resultVector.z();
1974
1975         return *this;
1976     }
1977
1978     inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
1979     {
1980         return Quaternion((double)q1.scalar() + q2.scalar(), (double)q1.x() + q2.x(), (double)q1.y() +
1981             q2.y(), (double)q1.z() + q2.z());
1982     }
1983
1984     inline Quaternion operator-(const Quaternion& q)
1985     {
1986         return Quaternion(-q.scalar(), -q.x(), -q.y(), -q.z());
1987     }
1988
1989     inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
1990     {
1991         return Quaternion((double)q1.scalar() - q2.scalar(), (double)q1.x() - q2.x(), (double)q1.y() -
1992             q2.y(), (double)q1.z() - q2.z());
1993     }
1994
1995     inline Quaternion operator*(float k, const Quaternion& q)
1996     {
1997         return Quaternion((double)k * q.scalar(), (double)k * q.x(), (double)k * q.y(), (double)k *
1998             q.z());
1999     }
2000
2001     inline Quaternion operator*(const Quaternion& q, float k)
2002     {
2003         return Quaternion(q.scalar() * (double)k, q.x() * (double)k, q.y() * (double)k, q.z() *
2004             (double)k);
2005     }
2006
2007     inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
2008     {
2009         //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
2010         //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
2011
2012         Vector3D q1Vector(q1.x(), q1.y(), q1.z());
2013         Vector3D q2Vector(q2.x(), q2.y(), q2.z());
2014
2015         double s{ (double)q1.scalar() * q2.scalar() };
2016         double dP{ dotProduct(q1Vector, q2Vector) };
2017         double resultScalar{ s - dP };
2018
2019         Vector3D a(q1.scalar() * q2Vector);
2020         Vector3D b(q2.scalar() * q1Vector);
2021         Vector3D cP(crossProduct(q1Vector, q2Vector));
2022         Vector3D resultVector(a + b + cP);
2023
2024         return Quaternion(resultScalar, resultVector);
2025     }
2026
2027     }
2028
2029
2030
2031
2032
2033

```



```

2034
2037 inline bool isZeroQuaternion(const Quaternion& q)
2038 {
2039     //zero quaternion = (0, 0, 0, 0)
2040     return compareFloats(q.scalar(), 0.0f, EPSILON) && compareFloats(q.x(), 0.0f, EPSILON) &&
2041         compareFloats(q.y(), 0.0f, EPSILON) && compareFloats(q.z(), 0.0f, EPSILON);
2042 }
2043
2046 inline bool isIdentity(const Quaternion& q)
2047 {
2048     //identity quaternion = (1, 0, 0, 0)
2049     return compareFloats(q.scalar(), 1.0f, EPSILON) && compareFloats(q.x(), 0.0f, EPSILON) &&
2050         compareFloats(q.y(), 0.0f, EPSILON) && compareFloats(q.z(), 0.0f, EPSILON);
2051 }
2052
2055 inline Quaternion conjugate(const Quaternion& q)
2056 {
2057     //conjugate of a quaternion is the quaternion with its vector part negated
2058     return Quaternion(q.scalar(), -q.x(), -q.y(), -q.z());
2059 }
2060
2063 inline float length(const Quaternion& q)
2064 {
2065     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
2066     return sqrt((double)q.scalar() * q.scalar() + (double)q.x() * q.x() + (double)q.y() * q.y() +
2067         (double)q.z() * q.z());
2068 }
2069
2072 inline Quaternion normalize(const Quaternion& q)
2073 {
2074     //to normalize a quaternion you do q / |q|
2075
2076     if (isZeroQuaternion(q))
2077         return q;
2078
2079     double d{ length(q) };
2080
2081     return Quaternion(q.scalar() / d, q.x() / d, q.y() / d, q.z() / d);
2082 }
2083
2087 inline Quaternion inverse(const Quaternion& q)
2088 {
2089     //inverse = conjugate of q / |q|^2
2090
2091     if (isZeroQuaternion(q))
2092         return q;
2093
2094     Quaternion conjugateOfQ(conjugate(q));
2095
2096     double d{ length(q) };
2097     d *= d;
2098
2099     return Quaternion(conjugateOfQ.scalar() / d, conjugateOfQ.x() / d, conjugateOfQ.y() / d,
2100         conjugateOfQ.z() / d);
2101 }
2102
2105 inline Quaternion rotationQuaternion(float angle, float x, float y, float z)
2106 {
2107     //A roatation quaternion is a quaternion where the
2108     //scalar part = cos(theta / 2)
2109     //vector part = sin(theta / 2) * axis
2110     //the axis needs to be normalized
2111
2112     double ang{ angle / 2.0 };
2113     double c{ cos(ang * PI / 180.0) };
2114     double s{ sin(ang * PI / 180.0) };
2115
2116     Vector3D axis(x, y, z);
2117     axis = norm(axis);
2118
2119     return Quaternion(c, s * axis.x(), s * axis.y(), s * axis.z());
2120 }
2121
2125 inline Quaternion rotationQuaternion(float angle, const Vector3D& axis)
2126 {
2127     //A roatation quaternion is a quaternion where the
2128     //scalar part = cos(theta / 2)
2129     //vector part = sin(theta / 2) * axis
2130     //the axis needs to be normalized
2131
2132     double ang{ angle / 2.0 };
2133     double c{ cos(ang * PI / 180.0) };
2134     double s{ sin(ang * PI / 180.0) };
2135
2136     Vector3D axisN(norm(axis));
2137
2138     return Quaternion(c, s * axisN.x(), s * axisN.y(), s * axisN.z());

```

```

2139     }
2140
2141     inline Quaternion rotationQuaternion(const Vector4D& angAxis)
2142     {
2143         //A roatation quaternion is a quaternion where the
2144         //scalar part = cos(theta / 2)
2145         //vector part = sin(theta / 2) * axis
2146         //the axis needs to be normalized
2147
2148         double angle{ angAxis.x() / 2.0 };
2149         double c{ cos(angle * PI / 180.0) };
2150         double s{ sin(angle * PI / 180.0) };
2151
2152         Vector3D axis(angAxis.y(), angAxis.z(), angAxis.w());
2153         axis = norm(axis);
2154
2155         return Quaternion(c, s * axis.x(), s * axis.y(), s * axis.z());
2156     }
2157
2158     inline Matrix4x4 quaternionRotationMatrixCol(const Quaternion& q)
2159     {
2160         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
2161         //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
2162         //2q2q4 - 2q1q3          2q3q4 + 2q1q2          1 - 2q2^2 - 2q3^2      0
2163         //0                      0                      0                      1
2164         //q1 = scalar
2165         //q2 = x
2166         //q3 = y
2167         //q4 = z
2168
2169         float colMat[4][4] = {};
2170
2171         colMat[0][0] = 1.0 - 2.0 * q.y() * q.y() - 2.0 * q.z() * q.z();
2172         colMat[0][1] = 2.0 * q.x() * q.y() - 2.0 * q.scalar() * q.z();
2173         colMat[0][2] = 2.0 * q.x() * q.z() + 2.0 * q.scalar() * q.y();
2174         colMat[0][3] = 0.0f;
2175
2176         colMat[1][0] = 2.0 * q.x() * q.y() + 2.0 * q.scalar() * q.z();
2177         colMat[1][1] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.z() * q.z();
2178         colMat[1][2] = 2.0 * q.y() * q.z() - 2.0 * q.scalar() * q.x();
2179         colMat[1][3] = 0.0f;
2180
2181         colMat[2][0] = 2.0 * q.x() * q.z() - 2.0 * q.scalar() * q.y();
2182         colMat[2][1] = 2.0 * q.y() * q.z() + 2.0 * q.scalar() * q.x();
2183         colMat[2][2] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.y() * q.y();
2184         colMat[2][3] = 0.0f;
2185
2186         colMat[3][0] = 0.0f;
2187         colMat[3][1] = 0.0f;
2188         colMat[3][2] = 0.0f;
2189         colMat[3][3] = 1.0f;
2190
2191         return Matrix4x4(colMat);
2192     }
2193
2194     inline Matrix4x4 quaternionRotationMatrixRow(const Quaternion& q)
2195     {
2196         //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
2197         //2q2q3 - 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
2198         //2q2q4 + 2q1q3          2q3q4 - 2q1q2          1 - 2q2^2 - 2q3^2      0
2199         //0                      0                      0                      1
2200         //q1 = scalar
2201         //q2 = x
2202         //q3 = y
2203         //q4 = z
2204
2205         float rowMat[4][4] = {};
2206
2207         rowMat[0][0] = 1.0 - 2.0 * q.y() * q.y() - 2.0 * q.z() * q.z();
2208         rowMat[0][1] = 2.0 * q.x() * q.y() + 2.0 * q.scalar() * q.z();
2209         rowMat[0][2] = 2.0 * q.x() * q.z() - 2.0 * q.scalar() * q.y();
2210         rowMat[0][3] = 0.0f;
2211
2212         rowMat[1][0] = 2.0 * q.x() * q.y() - 2.0 * q.scalar() * q.z();
2213         rowMat[1][1] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.z() * q.z();
2214         rowMat[1][2] = 2.0 * q.y() * q.z() + 2.0 * q.scalar() * q.x();
2215         rowMat[1][3] = 0.0f;
2216
2217         rowMat[2][0] = 2.0 * q.x() * q.z() + 2.0 * q.scalar() * q.y();
2218         rowMat[2][1] = 2.0 * q.y() * q.z() - 2.0 * q.scalar() * q.x();
2219         rowMat[2][2] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.y() * q.y();
2220         rowMat[2][3] = 0.0f;
2221
2222         rowMat[3][0] = 0.0f;
2223         rowMat[3][1] = 0.0f;
2224         rowMat[3][2] = 0.0f;
2225         rowMat[3][3] = 1.0f;
2226
2227         return Matrix4x4(rowMat);
2228     }
2229
2230     inline Matrix4x4 quaternionRotationMatrix(const Quaternion& q)
2231     {
2232         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
2233         //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
2234         //2q2q4 - 2q1q3          2q3q4 + 2q1q2          1 - 2q2^2 - 2q3^2      0
2235         //0                      0                      0                      1
2236         //q1 = scalar
2237         //q2 = x
2238         //q3 = y
2239         //q4 = z
2240
2241         float mat[4][4] = {};
2242
2243         mat[0][0] = 1.0 - 2.0 * q.y() * q.y() - 2.0 * q.z() * q.z();
2244         mat[0][1] = 2.0 * q.x() * q.y() - 2.0 * q.scalar() * q.z();
2245         mat[0][2] = 2.0 * q.x() * q.z() + 2.0 * q.scalar() * q.y();
2246         mat[0][3] = 0.0f;
2247
2248         mat[1][0] = 2.0 * q.x() * q.y() + 2.0 * q.scalar() * q.z();
2249         mat[1][1] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.z() * q.z();
2250         mat[1][2] = 2.0 * q.y() * q.z() - 2.0 * q.scalar() * q.x();
2251         mat[1][3] = 0.0f;
2252
2253         mat[2][0] = 2.0 * q.x() * q.z() - 2.0 * q.scalar() * q.y();
2254         mat[2][1] = 2.0 * q.y() * q.z() + 2.0 * q.scalar() * q.x();
2255         mat[2][2] = 1.0 - 2.0 * q.x() * q.x() - 2.0 * q.y() * q.y();
2256         mat[2][3] = 0.0f;
2257
2258         mat[3][0] = 0.0f;
2259         mat[3][1] = 0.0f;
2260         mat[3][2] = 0.0f;
2261         mat[3][3] = 1.0f;
2262
2263         return Matrix4x4(mat);
2264     }

```

```
2236
2237     return Matrix4x4(rowMat);
2238 }
2239
2240 #if defined(_DEBUG)
2241     inline void print(const Quaternion& q)
2242     {
2243         std::cout << "(" << q.scalar() << ", " << q.x() << ", " << q.y() << ", " << q.z();
2244     }
2245 #endif
2246 //-----
2247
2248 //-----
2249 }
```


Index

- adjoint
 - FAMath, 11
- CartesianToCylindrical
 - FAMath, 11
- CartesianToPolar
 - FAMath, 11
- CartesianToSpherical
 - FAMath, 11
- cofactor
 - FAMath, 11
- conjugate
 - FAMath, 12
- crossProduct
 - FAMath, 12
- CylindricalToCartesian
 - FAMath, 12
- data
 - FAMath::Matrix4x4, 28
- det
 - FAMath, 12
- dotProduct
 - FAMath, 12, 13
- FAMath, 7
 - adjoint, 11
 - CartesianToCylindrical, 11
 - CartesianToPolar, 11
 - CartesianToSpherical, 11
 - cofactor, 11
 - conjugate, 12
 - crossProduct, 12
 - CylindricalToCartesian, 12
 - det, 12
 - dotProduct, 12, 13
 - inverse, 13
 - isIdentity, 13, 14
 - isZeroQuaternion, 14
 - length, 14
 - norm, 15
 - normalize, 15
 - operator*, 15–18
 - operator+, 18, 19
 - operator-, 19–21
 - operator/, 21, 22
 - PolarToCartesian, 22
 - Projection, 22
 - quaternionRotationMatrixCol, 23
 - quaternionRotationMatrixRow, 23
 - rotate, 23
 - rotationQuaternion, 23, 24
 - scale, 24
 - setToIdentity, 24
 - SphericalToCartesian, 24
 - translate, 24
 - transpose, 25
 - zeroVector, 25
- FAMath::Matrix4x4, 27
 - data, 28
 - Matrix4x4, 28
 - operator*=, 29
 - operator(), 28, 29
 - operator+=, 29
 - operator-=, 29
 - setCol, 30
 - setRow, 30
- FAMath::Quaternion, 30
 - operator*=, 32
 - operator+=, 33
 - operator-=, 33
 - Quaternion, 31, 32
 - scalar, 33
 - vector, 33
 - x, 33, 34
 - y, 34
 - z, 34
- FAMath::Vector2D, 35
 - operator*=, 36
 - operator+=, 36
 - operator-=, 36
 - operator/=: 36
 - Vector2D, 35
 - x, 36, 37
 - y, 37
- FAMath::Vector3D, 37
 - operator*=, 39
 - operator+=, 39
 - operator-=, 39
 - operator/=: 39
 - Vector3D, 38
 - x, 39, 40
 - y, 40
 - z, 40
- FAMath::Vector4D, 41
 - operator*=, 42
 - operator+=, 43
 - operator-=, 43
 - operator/=: 43

- Vector4D, [42](#)
 - w, [43](#)
 - x, [43](#), [44](#)
 - y, [44](#)
 - z, [44](#)
- inverse
 - FAMath, [13](#)
- isIdentity
 - FAMath, [13](#), [14](#)
- isZeroQuaternion
 - FAMath, [14](#)
- length
 - FAMath, [14](#)
- Matrix4x4
 - FAMath::Matrix4x4, [28](#)
- norm
 - FAMath, [15](#)
- normalize
 - FAMath, [15](#)
- operator*
 - FAMath, [15–18](#)
- operator*=
 - FAMath::Matrix4x4, [29](#)
 - FAMath::Quaternion, [32](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [42](#)
- operator()
 - FAMath::Matrix4x4, [28](#), [29](#)
- operator+
 - FAMath, [18](#), [19](#)
- operator+=
 - FAMath::Matrix4x4, [29](#)
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [43](#)
- operator-
 - FAMath, [19–21](#)
- operator-=
 - FAMath::Matrix4x4, [29](#)
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [43](#)
- operator/
 - FAMath, [21](#), [22](#)
- operator/=
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [43](#)
- PolarToCartesian
 - FAMath, [22](#)
- Projection
 - FAMath, [22](#)
- Quaternion
 - FAMath::Quaternion, [31](#), [32](#)
- quaternionRotationMatrixCol
 - FAMath, [23](#)
- quaternionRotationMatrixRow
 - FAMath, [23](#)
- rotate
 - FAMath, [23](#)
- rotationQuaternion
 - FAMath, [23](#), [24](#)
- scalar
 - FAMath::Quaternion, [33](#)
- scale
 - FAMath, [24](#)
- setCol
 - FAMath::Matrix4x4, [30](#)
- setRow
 - FAMath::Matrix4x4, [30](#)
- setToIdentity
 - FAMath, [24](#)
- SphericalToCartesian
 - FAMath, [24](#)
- translate
 - FAMath, [24](#)
- transpose
 - FAMath, [25](#)
- vector
 - FAMath::Quaternion, [33](#)
- Vector2D
 - FAMath::Vector2D, [35](#)
- Vector3D
 - FAMath::Vector3D, [38](#)
- Vector4D
 - FAMath::Vector4D, [42](#)
- w
 - FAMath::Vector4D, [43](#)
- x
 - FAMath::Quaternion, [33](#), [34](#)
 - FAMath::Vector2D, [36](#), [37](#)
 - FAMath::Vector3D, [39](#), [40](#)
 - FAMath::Vector4D, [43](#), [44](#)
- y
 - FAMath::Quaternion, [34](#)
 - FAMath::Vector2D, [37](#)
 - FAMath::Vector3D, [40](#)
 - FAMath::Vector4D, [44](#)
- z
 - FAMath::Quaternion, [34](#)
 - FAMath::Vector3D, [40](#)
 - FAMath::Vector4D, [44](#)
- zeroVector
 - FAMath, [25](#)