

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()	12
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()	13
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]	14
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]	15
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 Orthonormalize()	22
4.1.2.59 PolarToCartesian()	22
4.1.2.60 Projection() [1/3]	22
4.1.2.61 Projection() [2/3]	23
4.1.2.62 Projection() [3/3]	23
4.1.2.63 QuaternionToRotationMatrixCol()	23
4.1.2.64 QuaternionToRotationMatrixRow()	23
4.1.2.65 Rotate()	23
4.1.2.66 RotationQuaternion() [1/3]	24
4.1.2.67 RotationQuaternion() [2/3]	24
4.1.2.68 RotationQuaternion() [3/3]	24
4.1.2.69 Scale()	24
4.1.2.70 SetToIdentity()	24
4.1.2.71 SphericalToCartesian()	25
4.1.2.72 Translate()	25

4.1.2.73 Transpose()	25
4.1.2.74 ZeroVector() [1/3]	25
4.1.2.75 ZeroVector() [2/3]	25
4.1.2.76 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() [1/3]	28
5.1.2.2 Matrix4x4() [2/3]	28
5.1.2.3 Matrix4x4() [3/3]	28
5.1.3 Member Function Documentation	29
5.1.3.1 Data() [1/2]	29
5.1.3.2 Data() [2/2]	29
5.1.3.3 GetCol()	29
5.1.3.4 GetRow()	29
5.1.3.5 operator()() [1/2]	29
5.1.3.6 operator()() [2/2]	30
5.1.3.7 operator*=() [1/2]	30
5.1.3.8 operator*=() [2/2]	30
5.1.3.9 operator+=()	30
5.1.3.10 operator-=()	30
5.1.3.11 SetCol()	31
5.1.3.12 SetRow()	31
5.2 FAMath::Quaternion Class Reference	31
5.2.1 Detailed Description	32
5.2.2 Constructor & Destructor Documentation	32
5.2.2.1 Quaternion() [1/4]	32
5.2.2.2 Quaternion() [2/4]	32
5.2.2.3 Quaternion() [3/4]	33
5.2.2.4 Quaternion() [4/4]	33
5.2.3 Member Function Documentation	33
5.2.3.1 GetScalar()	33
5.2.3.2 GetVector()	33
5.2.3.3 GetX()	33
5.2.3.4 GetY()	34
5.2.3.5 GetZ()	34
5.2.3.6 operator*=() [1/2]	34
5.2.3.7 operator*=() [2/2]	34
5.2.3.8 operator+=()	34
5.2.3.9 operator-=()	34

5.2.3.10 SetScalar()	35
5.2.3.11 SetVector()	35
5.2.3.12 SetX()	35
5.2.3.13 SetY()	35
5.2.3.14 SetZ()	35
5.3 FAMath::Vector2D Class Reference	36
5.3.1 Detailed Description	36
5.3.2 Constructor & Destructor Documentation	36
5.3.2.1 Vector2D() [1/2]	36
5.3.2.2 Vector2D() [2/2]	37
5.3.3 Member Function Documentation	37
5.3.3.1 GetX()	37
5.3.3.2 GetY()	37
5.3.3.3 operator*=()	37
5.3.3.4 operator+=()	37
5.3.3.5 operator-=()	38
5.3.3.6 operator/=()	38
5.3.3.7 SetX()	38
5.3.3.8 SetY()	38
5.4 FAMath::Vector3D Class Reference	38
5.4.1 Detailed Description	39
5.4.2 Constructor & Destructor Documentation	39
5.4.2.1 Vector3D() [1/2]	39
5.4.2.2 Vector3D() [2/2]	40
5.4.3 Member Function Documentation	40
5.4.3.1 GetX()	40
5.4.3.2 GetY()	40
5.4.3.3 GetZ()	40
5.4.3.4 operator*=()	40
5.4.3.5 operator+=()	41
5.4.3.6 operator-=()	41
5.4.3.7 operator/=()	41
5.4.3.8 SetX()	41
5.4.3.9 SetY()	41
5.4.3.10 SetZ()	42
5.5 FAMath::Vector4D Class Reference	42
5.5.1 Detailed Description	43
5.5.2 Constructor & Destructor Documentation	43
5.5.2.1 Vector4D() [1/2]	43
5.5.2.2 Vector4D() [2/2]	43
5.5.3 Member Function Documentation	43
5.5.3.1 GetW()	43

5.5.3.2 GetX()	43
5.5.3.3 GetY()	44
5.5.3.4 GetZ()	44
5.5.3.5 operator*=()	44
5.5.3.6 operator+=()	44
5.5.3.7 operator-=()	44
5.5.3.8 operator/=()	44
5.5.3.9 SetW()	45
5.5.3.10 SetX()	45
5.5.3.11 SetY()	45
5.5.3.12 SetZ()	45
6 File Documentation	47
6.1 FAMathEngine.h	47
Index	71

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	31
FAMath::Vector2D	
A vector class used for 2D vectors/points and their manipulations	36
FAMath::Vector3D	
A vector class used for 3D vectors/points and their manipulations	38
FAMath::Vector4D	
A vector class used for 4D vectors/points and their manipulations	42

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, float k)
*2D vector scalar multiplication. Returns $a * k$, where a is a vector and k is a scalar(float)*
- [Vector2D](#) **operator*** (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, float k)
3D vector scalar multiplication. Returns $a * k$, where a is a vector and k is a scalar(float)
 - [Vector3D operator*](#) (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, 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.
- void **Orthonormalize** (**Vector3D** &x, **Vector3D** &y, **Vector3D** &z)
Orthonormalizes the specified vectors. Uses Classical Gram-Schmidt.
- 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, float k)
4D vector scalar multiplication. Returns $a * k$, where a is a vector and k is a scalar(float)
- **Vector4D operator*** (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, float k)
4D vector scalar division. Returns a / k , where a is a vector and k is a scalar(float) If $k = 0$ the returned vector is the zero vector.
- float **DotProduct** (const **Vector4D** &a, const **Vector4D** &b)
Returns the dot product between two 4D vectors.
- float **Length** (const **Vector4D** &v)
Returns the length(magnitude) of the 4D vector v .
- **Vector4D Norm** (const **Vector4D** &v)
Normalizes the 4D vector v . If the 4D vector is the zero vector v is returned.
- **Vector4D Projection** (const **Vector4D** &a, const **Vector4D** &b)
Returns a 4D vector that is the projection of a onto b . If b is the zero vector a is returned.
- **Matrix4x4 operator+** (const **Matrix4x4** &m1, const **Matrix4x4** &m2)
Adds the two given 4x4 matrices and returns a **Matrix4x4** object with the result.
- **Matrix4x4 operator-** (const **Matrix4x4** &m)
Negates the 4x4 matrix m .
- **Matrix4x4 operator-** (const **Matrix4x4** &m1, const **Matrix4x4** &m2)
Subtracts the two given 4x4 matrices and returns a **Matrix4x4** object with the result.
- **Matrix4x4 operator*** (const **Matrix4x4** &m, const float &k)
Multiplies the given 4x4 matrix with the given scalar and returns a **Matrix4x4** object with the result.
- **Matrix4x4 operator*** (const float &k, const **Matrix4x4** &m)
Multiplies the the given scalar with the given 4x4 matrix and returns a **Matrix4x4** object with the result.
- **Matrix4x4 operator*** (const **Matrix4x4** &m1, const **Matrix4x4** &m2)
Multiplies the two given 4x4 matrices and returns a **Matrix4x4** object with the result.
- **Vector4D operator*** (const **Matrix4x4** &m, const **Vector4D** &v)
Multiplies the given 4x4 matrix with the given 4D vector and returns a **Vector4D** object with the result. The vector is a column vector.
- **Vector4D operator*** (const **Vector4D** &v, const **Matrix4x4** &m)
Multiplies the given 4D vector with the given 4x4 matrix and returns a **Vector4D** object with the result. The vector is a row vector.
- void **SetToIdentity** (**Matrix4x4** &m)
Sets the given matrix to the identity matrix.
- bool **IsIdentity** (const **Matrix4x4** &m)
Returns true if the given matrix is the identity matrix, false otherwise.

- [Matrix4x4 Transpose](#) (const [Matrix4x4](#) &m)
Returns the tranpose of the given matrix m.
- [Matrix4x4 Translate](#) (const [Matrix4x4](#) &cm, float x, float y, float z)
*Construct a 4x4 translation matrix with the given floats and post-multiply's it by the given matrix. $cm = cm * translate$.*
- [Matrix4x4 Scale](#) (const [Matrix4x4](#) &cm, float x, float y, float z)
*Construct a 4x4 scaling matrix with the given floats and post-multiply's it by the given matrix. $cm = cm * scale$.*
- [Matrix4x4 Rotate](#) (const [Matrix4x4](#) &cm, float angle, float x, float y, float z)
*Construct a 4x4 rotation matrix with the given angle (in degrees) and axis (x, y, z) and post-multiply's it by the given matrix. $cm = cm * rotate$.*
- double [Det](#) (const [Matrix4x4](#) &m)
Returns the determinant of the given matrix.
- double [Cofactor](#) (const [Matrix4x4](#) &m, unsigned int row, unsigned int col)
Returns the cofactor of the given row and col using the given matrix.
- [Matrix4x4 Adjoint](#) (const [Matrix4x4](#) &m)
Returns the adjoint of the given matrix.
- [Matrix4x4 Inverse](#) (const [Matrix4x4](#) &m)
Returns the inverse of the given matrix. If the matrix is noninvertible/singular, the identity matrix is returned.
- [Quaternion operator+](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
Returns a quaternion that has the result of $q1 + q2$.
- [Quaternion operator-](#) (const [Quaternion](#) &q)
Returns a quaternion that has the result of $-q$.
- [Quaternion operator-](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
Returns a quaternion that has the result of $q1 - q2$.
- [Quaternion operator*](#) (float k, const [Quaternion](#) &q)
*Returns a quaternion that has the result of $k * q$.*
- [Quaternion operator*](#) (const [Quaternion](#) &q, float k)
*Returns a quaternion that has the result of $q * k$.*
- [Quaternion operator*](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
*Returns a quaternion that has the result of $q1 * q2$.*
- bool [IsZeroQuaternion](#) (const [Quaternion](#) &q)
Returns true if quaternion q is a zero quaternion, false otherwise.
- bool [IsIdentity](#) (const [Quaternion](#) &q)
Returns true if quaternion q is an identity quaternion, false otherwise.
- [Quaternion Conjugate](#) (const [Quaternion](#) &q)
Returns the conjugate of quaternion q.
- float [Length](#) (const [Quaternion](#) &q)
Returns the length of quaternion q.
- [Quaternion Normalize](#) (const [Quaternion](#) &q)
Normalizes quaternion q and returns the normalized quaternion. If q is the zero quaternion then q is returned.
- [Quaternion Inverse](#) (const [Quaternion](#) &q)
Returns the invese of quaternion q. If q is the zero quaternion then q is returned.
- [Quaternion RotationQuaternion](#) (float angle, float x, float y, float z)
Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.
- [Quaternion RotationQuaternion](#) (float angle, const [Vector3D](#) &axis)
Returns a quaternion from the axis-angle rotation representation. The angle should be given in degrees.
- [Quaternion RotationQuaternion](#) (const [Vector4D](#) &angAxis)
*Returns a quaternion from the axis-angle rotation representation. The x value in the 4D vector should be the angle(in degrees).
The y, z and w value in the 4D vector should be the axis.*
- [Matrix4x4 QuaternionToRotationMatrixCol](#) (const [Quaternion](#) &q)

Returns a matrix from the given quaterion for column vector-matrix multiplication. [Quaternion](#) q should be a unit quaternion.

- [Matrix4x4 QuaternionToRotationMatrixRow](#) (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, phi(degrees), theta(degrees)).

4.1.2.5 Cofactor()

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

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

4.1.2.6 Conjugate()

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

Returns the conjugate of quaternion q .

4.1.2.7 CrossProduct()

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

Returns the cross product between two 3D vectors.

4.1.2.8 CylindricalToCartesian()

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

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

The returned 3D vector = (x, y ,z).

4.1.2.9 Det()

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

Returns the determinant of the given matrix.

4.1.2.10 DotProduct() [1/3]

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

Returns the dot product between two 2D vectors.

4.1.2.11 DotProduct() [2/3]

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

Returns the dot product between two 3D vectors.

4.1.2.12 DotProduct() [3/3]

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

Returns the dot product between two 4D vectors.

4.1.2.13 Inverse() [1/2]

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

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

4.1.2.14 Inverse() [2/2]

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

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

4.1.2.15 IsIdentity() [1/2]

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

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

4.1.2.16 IsIdentity() [2/2]

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

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

4.1.2.17 IsZeroQuaternion()

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

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

4.1.2.18 Length() [1/4]

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

Returns the length of quaternion q.

4.1.2.19 Length() [2/4]

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

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

4.1.2.20 Length() [3/4]

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

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

4.1.2.21 Length() [4/4]

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

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

4.1.2.22 Norm() [1/3]

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

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

4.1.2.23 Norm() [2/3]

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

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

4.1.2.24 Norm() [3/3]

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

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

4.1.2.25 Normalize()

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

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

4.1.2.26 operator*() [1/14]

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

Multiplies the the given scalar with the given 4x4 matrix and returns a [Matrix4x4](#) object with the result.

4.1.2.27 operator*() [2/14]

```
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.

4.1.2.28 operator*() [3/14]

```
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.

4.1.2.29 operator*() [4/14]

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

Multiplies the two given 4x4 matrices and returns a [Matrix4x4](#) object with the result.

4.1.2.30 operator*() [5/14]

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

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

4.1.2.31 operator*() [6/14]

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

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

4.1.2.32 operator*() [7/14]

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

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

4.1.2.33 operator*() [8/14]

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

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

4.1.2.34 operator*() [9/14]

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

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

4.1.2.35 operator*() [10/14]

```
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.

4.1.2.36 operator*() [11/14]

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

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

4.1.2.37 operator*() [12/14]

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

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

4.1.2.38 operator*() [13/14]

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

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

4.1.2.39 operator*() [14/14]

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

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

4.1.2.40 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,
    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,
    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 Orthonormalize()

```
void FAMath::Orthonormalize (
    Vector3D & x,
    Vector3D & y,
    Vector3D & z ) [inline]
```

Orthonormalizes the specified vectors. Uses Classical Gram-Schmidt.

4.1.2.59 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)

4.1.2.60 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.

4.1.2.61 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.

4.1.2.62 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.

4.1.2.63 QuaternionToRotationMatrixCol()

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

Returns a matrix from the given quaterion for column vector-matrix multiplication. [Quaternion](#) q should be a unit quaternion.

4.1.2.64 QuaternionToRotationMatrixRow()

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

Returns a matrix from the given quaterion for row vector-matrix multiplication. [Quaternion](#) q should be a unit quaternion.

4.1.2.65 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. $cm = cm * rotate$.

.

4.1.2.66 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.

4.1.2.67 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.

4.1.2.68 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.

4.1.2.69 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. $cm = cm * scale$.

4.1.2.70 SetToIdentity()

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

Sets the given matrix to the identity matrix.

4.1.2.71 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)

4.1.2.72 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.

4.1.2.73 Transpose()

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

Returns the tranpose of the given matrix *m*.

4.1.2.74 ZeroVector() [1/3]

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

Returns true if *a* is the zero vector.

4.1.2.75 ZeroVector() [2/3]

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

Returns true if *a* is the zero vector.

4.1.2.76 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.
- [Matrix4x4](#) (const [Vector4D](#) &r1, const [Vector4D](#) &r2, const [Vector4D](#) &r3, const [Vector4D](#) &r4)
Overloaded Constructor. Creates a new 4x4 matrix with each row being set to the specified rows.
- 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.
- [Vector4D GetRow](#) (unsigned int row) const
Returns specified row. Row should be between [0,3]. If it is out of range the first row will be returned.
- [Vector4D GetCol](#) (unsigned int col) const
Returns specified col. Col should be between [0,3]. If it is out of range the first col 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 is 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 is 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-=` (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()` [1/3]

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

Default Constructor.

Creates a new 4x4 identity matrix.

5.1.2.2 `Matrix4x4()` [2/3]

```
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.

5.1.2.3 `Matrix4x4()` [3/3]

```
FAMath::Matrix4x4::Matrix4x4 (
    const Vector4D & r1,
    const Vector4D & r2,
    const Vector4D & r3,
    const Vector4D & r4 ) [inline]
```

Overloaded Constructor. Creates a new 4x4 matrix with each row being set to the specified rows.

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 GetCol()

```
Vector4D FAMath::Matrix4x4::GetCol (
    unsigned int col ) const [inline]
```

Returns specified col. Col should be between [0,3]. If it is out of range the first col will be returned.

5.1.3.4 GetRow()

```
Vector4D FAMath::Matrix4x4::GetRow (
    unsigned int row ) const [inline]
```

Returns specified row. Row should be between [0,3]. If it is out of range the first row will be returned.

5.1.3.5 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.

5.1.3.6 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.

5.1.3.7 operator*=() [1/2]

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

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

5.1.3.8 operator*=() [2/2]

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

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

5.1.3.9 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.

5.1.3.10 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.

5.1.3.11 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 is out of range the first col will be set.

5.1.3.12 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 is 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 [GetScalar](#) () const
Returns the scalar component of the quaternion.
- float [GetX](#) () const
Returns the x value of the vector component in the quaternion.
- float [GetY](#) () const
Returns the y value of the vector component in the quaternion.
- float [GetZ](#) () const
Returns the z value of the vector component in the quaternion.
- const [Vector3D](#) & [GetVector](#) () const
Returns the vector component of the quaternion.
- void [SetScalar](#) (float scalar)

- Sets the scalar component to the specified value.*
 - void **SetX** (float x)
- Sets the x component to the specified value.*
 - void **SetY** (float y)
- Sets the y component to the specified value.*
 - void **SetZ** (float z)
- Sets the z component to the specified value.*
 - void **SetVector** (const **Vector3D** &v)
- Sets the vector to the specified vector.*
 - **Quaternion** & **operator+=** (const **Quaternion** &q)
- Adds this quaternion to quaterion q and stores the result in this quaternion.*
 - **Quaternion** & **operator-=** (const **Quaternion** &q)
- Subtracts this quaternion by quaterion q and stores the result in this quaternion.*
 - **Quaternion** & **operator*=** (float k)
- Multiplies this quaternion by flaot k and stores the result in this quaternion.*
 - **Quaternion** & **operator*=** (const **Quaternion** &q)
- Multiplies this quaternion by quaterion q and stores the result in this quaternion.*

5.2.1 Detailed Description

The datatype for the components is float.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Quaternion() [1/4]

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

Default Constructor.

Constructs an identity quaternion.

5.2.2.2 Quaternion() [2/4]

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

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.3 Quaternion() [3/4]

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

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.4 Quaternion() [4/4]

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

Overloaded Constructor.

Constructs a quaternion with the given values in the 4D vector.

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

5.2.3 Member Function Documentation

5.2.3.1 GetScalar()

```
float FAMath::Quaternion::GetScalar ( ) const [inline]
```

Returns the scalar component of the quaternion.

5.2.3.2 GetVector()

```
const Vector3D & FAMath::Quaternion::GetVector ( ) const [inline]
```

Returns the vector component of the quaternion.

5.2.3.3 GetX()

```
float FAMath::Quaternion::GetX ( ) const [inline]
```

Returns the x value of the vector component in the quaternion.

5.2.3.4 GetY()

```
float FAMath::Quaternion::GetY ( ) const [inline]
```

Returns the y value of the vector component in the quaternion.

5.2.3.5 GetZ()

```
float FAMath::Quaternion::GetZ ( ) const [inline]
```

Returns the z value of the vector component in the quaternion.

5.2.3.6 operator*=() [1/2]

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

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

5.2.3.7 operator*=() [2/2]

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

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

5.2.3.8 operator+=()

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

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

5.2.3.9 operator-=()

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

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

5.2.3.10 SetScalar()

```
void FAMath::Quaternion::SetScalar (
    float scalar ) [inline]
```

Sets the scalar component to the specified value.

5.2.3.11 SetVector()

```
void FAMath::Quaternion::SetVector (
    const Vector3D & v ) [inline]
```

Sets the vector to the specified vector.

5.2.3.12 SetX()

```
void FAMath::Quaternion::SetX (
    float x ) [inline]
```

Sets the x component to the specified value.

5.2.3.13 SetY()

```
void FAMath::Quaternion::SetY (
    float y ) [inline]
```

Sets the y component to the specified value.

5.2.3.14 SetZ()

```
void FAMath::Quaternion::SetZ (
    float z ) [inline]
```

Sets the z component to the specified value.

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 [GetX](#) () const
Returns the x component.
- float [GetY](#) () const
Returns y component.
- void [SetX](#) (float x)
Sets the x component to the specified value.
- void [SetY](#) (float y)
Sets the y component to the specified value.
- [Vector2D](#) & [operator+=](#) (const [Vector2D](#) &b)
2D vector addition through overloading operator +=.
- [Vector2D](#) & [operator-=](#) (const [Vector2D](#) &b)
2D vector subtraction through overloading operator -=.
- [Vector2D](#) & [operator*=](#) (float k)
*2D vector scalar multiplication through overloading operator *=.*
- [Vector2D](#) & [operator/=](#) (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 GetX()

```
float FAMath::Vector2D::GetX ( ) const [inline]
```

Returns the x component.

5.3.3.2 GetY()

```
float FAMath::Vector2D::GetY ( ) const [inline]
```

Returns y component.

5.3.3.3 operator*=()

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

2D vector scalar multiplication through overloading operator *.

5.3.3.4 operator+=()

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

2D vector addition through overloading operator +.

5.3.3.5 operator-=()

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

2D vector subtraction through overloading operator -=.

5.3.3.6 operator/=()

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

2D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.3.3.7 SetX()

```
void FAMath::Vector2D::SetX (
    float x ) [inline]
```

Sets the x component to the specified value.

5.3.3.8 SetY()

```
void FAMath::Vector2D::SetY (
    float y ) [inline]
```

Sets the y component to the specified value.

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 [GetX](#) () const
Returns the x component.
- float [GetY](#) () const
Returns y component.
- float [GetZ](#) () const
Returns the z component.
- void [SetX](#) (float x)
Sets the x component to the specified value.
- void [SetY](#) (float y)
Sets the y component to the specified value.
- void [SetZ](#) (float z)
Sets the z component to the specified value.
- [Vector3D](#) & [operator+=](#) (const [Vector3D](#) &b)
3D vector addition through overloading operator +=.
- [Vector3D](#) & [operator-=](#) (const [Vector3D](#) &b)
3D vector subtraction through overloading operator -=.
- [Vector3D](#) & [operator*=](#) (float k)
*3D vector scalar multiplication through overloading operator *=.*
- [Vector3D](#) & [operator/=](#) (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]

```
FMath::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 GetX()

```
float FAMath::Vector3D::GetX ( ) const [inline]
```

Returns the x component.

5.4.3.2 GetY()

```
float FAMath::Vector3D::GetY ( ) const [inline]
```

Returns y component.

5.4.3.3 GetZ()

```
float FAMath::Vector3D::GetZ ( ) const [inline]
```

Returns the z component.

5.4.3.4 operator*=()

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

3D vector scalar multiplication through overloading operator *.

5.4.3.5 operator+=()

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

3D vector addition through overloading operator +=.

5.4.3.6 operator-=()

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

3D vector subtraction through overloading operator -=.

5.4.3.7 operator/=()

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

3D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.4.3.8 SetX()

```
void FAMath::Vector3D::SetX (
    float x ) [inline]
```

Sets the x component to the specified value.

5.4.3.9 SetY()

```
void FAMath::Vector3D::SetY (
    float y ) [inline]
```

Sets the y component to the specified value.

5.4.3.10 SetZ()

```
void FAMath::Vector3D::SetZ (
    float z ) [inline]
```

Sets the z component to the specified value.

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.
- float [GetX](#) () const
Returns the x component.
- float [GetY](#) () const
Returns the y component.
- float [GetZ](#) () const
Returns the z component.
- float [GetW](#) () const
Returns the w component.
- void [SetX](#) (float x)
Sets the x component to the specified value.
- void [SetY](#) (float y)
Sets the y component to the specified value.
- void [SetZ](#) (float z)
Sets the z component to the specified value.
- void [SetW](#) (float w)
Sets the w component to the specified value.
- [Vector4D](#) & [operator+=](#) (const [Vector4D](#) &b)
4D vector addition through overloading operator +=.
- [Vector4D](#) & [operator-=](#) (const [Vector4D](#) &b)
4D vector subtraction through overloading operator -=.
- [Vector4D](#) & [operator*=](#) (float k)
*4D vector scalar multiplication through overloading operator *=.*
- [Vector4D](#) & [operator/=](#) (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() [1/2]

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

Default Constructor.

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

5.5.2.2 Vector4D() [2/2]

```
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.

5.5.3 Member Function Documentation

5.5.3.1 GetW()

```
float FAMath::Vector4D::GetW ( ) const [inline]
```

Returns the w component.

5.5.3.2 GetX()

```
float FAMath::Vector4D::GetX ( ) const [inline]
```

Returns the x component.

5.5.3.3 GetY()

```
float FAMath::Vector4D::GetY ( ) const [inline]
```

Returns the y component.

5.5.3.4 GetZ()

```
float FAMath::Vector4D::GetZ ( ) const [inline]
```

Returns the z component.

5.5.3.5 operator*=()

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

4D vector scalar multiplication through overloading operator *.

5.5.3.6 operator+=()

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

4D vector addition through overloading operator +.

5.5.3.7 operator-=()

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

4D vector subtraction through overloading operator -.

5.5.3.8 operator/=()

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

4D vector scalar division through overloading operator /.

If k is zero, the vector is unchanged.

5.5.3.9 SetW()

```
void FAMath::Vector4D::SetW (
    float w ) [inline]
```

Sets the w component to the specified value.

5.5.3.10 SetX()

```
void FAMath::Vector4D::SetX (
    float x ) [inline]
```

Sets the x component to the specified value.

5.5.3.11 SetY()

```
void FAMath::Vector4D::SetY (
    float y ) [inline]
```

Sets the y component to the specified value.

5.5.3.12 SetZ()

```
void FAMath::Vector4D::SetZ (
    float z ) [inline]
```

Sets the z component to the specified value.

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
16     /**@brief Checks if the two specified floats are equal using exact epsilon and adaptive epsilon.
17     */
18     inline bool CompareFloats(float x, float y, float epsilon)
19     {
20         float diff = fabs(x - y);
21         //exact epsilon
22         if (diff < epsilon)
23         {
24             return true;
25         }
26
27         //adaptive epsilon
28         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
29     }
30
31     /**@brief Checks if the two specified doubles are equal using exact epsilon and adaptive epsilon.
32     */
33     inline bool CompareDoubles(double x, double y, double epsilon)
34     {
35         double diff = fabs(x - y);
36         //exact epsilon
37         if (diff < epsilon)
38         {
39             return true;
40         }
41
42         //adaptive epsilon
43         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
44     }
45
46     //-----
47
48     class Vector2D
49     {
50     public:
51         Vector2D();
52         Vector2D(float x, float y);
53         float GetX() const;
```

```

78     float GetY() const;
79
82     void SetX(float x);
83
86     void SetY(float y);
87
90     Vector2D& operator+=(const Vector2D& b);
91
94     Vector2D& operator-=(const Vector2D& b);
95
98     Vector2D& operator*=(float k);
99
104    Vector2D& operator/=(float k);
105
106 private:
107     float mX;
108     float mY;
109 };
110
111
112 //-----
113 //Vector2D Constructors
114
115 inline Vector2D::Vector2D() : mX{ 0.0f }, mY{ 0.0f }
116 {}
117
118 inline Vector2D::Vector2D(float x, float y) : mX{ x }, mY{ y }
119 {}
120
121 //-----
122
123 //-----
124 //Vector2D Getters and Setters
125
126 inline float Vector2D::GetX() const
127 {
128     return mX;
129 }
130
131 inline float Vector2D::GetY() const
132 {
133     return mY;
134 }
135
136 inline void Vector2D::SetX(float x)
137 {
138     mX = x;
139 }
140
141 inline void Vector2D::SetY(float y)
142 {
143     mY = y;
144 }
145
146 //-----
147
148
149 //-----
150 //Vector2D Member functions
151
152 inline Vector2D& Vector2D::operator+=(const Vector2D& b)
153 {
154     this->mX += b.mX;
155     this->mY += b.mY;
156
157     return *this;
158 }
159
160 inline Vector2D& Vector2D::operator-=(const Vector2D& b)
161 {
162     this->mX -= b.mX;
163     this->mY -= b.mY;
164
165     return *this;
166 }
167
168 inline Vector2D& Vector2D::operator*=(float k)
169 {
170     this->mX *= k;
171     this->mY *= k;
172
173     return *this;
174 }
175
176 inline Vector2D& Vector2D::operator/=(float k)
177 {
178     if (CompareFloats(k, 0.0f, EPSILON))

```



```

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

```

```

295         //v = (r, theta)
296         //x = rcos(theta)
297         //y = rsin(theta)
298         float angle = v.GetY() * PI / 180.0f;
299
300         return Vector2D(v.GetX() * cos(angle), v.GetY() * 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.GetX(), 0.0f, EPSILON))
310         {
311             return v;
312         }
313
314         double theta{ atan2(v.GetY(), v.GetX()) * 180.0f / 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.GetX() << ", " << v.GetY() << ")";
331     }
332     #endif
333
334     //-----
335
336     //-----
337
338     class Vector3D
339     {
340     public:
341
342         Vector3D();
343
344         Vector3D(float x, float y, float z);
345
346         float GetX() const;
347
348         float GetY() const;
349
350         float GetZ() const;
351
352         void SetX(float x);
353
354         void SetY(float y);
355
356         void SetZ(float z);
357
358         Vector3D& operator+=(const Vector3D& b);
359
360         Vector3D& operator-=(const Vector3D& b);
361
362         Vector3D& operator*=(float k);
363
364         Vector3D& operator/=(float k);
365
366     private:
367         float mX;
368         float mY;
369         float mZ;
370     };
371
372     //-----
373
374     //Vector3D Constructors
375
376     inline Vector3D::Vector3D() : mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }
377     {}

```

```

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

```

```

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

```

```

628         //v = (r, theta, z)
629         //x = rcos(theta)
630         //y = rsin(theta)
631         //z = z
632         double angle{ v.GetY() * PI / 180.0f };
633
634         return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
635     }
636
637     inline Vector3D CartesianToCylindrical(const Vector3D& v)
638     {
639         //v = (x, y, z)
640         //r = sqrt(vx^2 + vy^2 + vz^2)
641         //theta = arctan(y / x)
642         //z = z
643         if (CompareFloats(v.GetX(), 0.0f, EPSILON))
644         {
645             return v;
646         }
647
648         double theta{ atan2(v.GetY(), v.GetX()) * 180.0 / PI };
649         return Vector3D(Length(v), theta, v.GetZ());
650     }
651
652     inline Vector3D SphericalToCartesian(const Vector3D& v)
653     {
654         // v = (pho, phi, theta)
655         //x = pho * sin(phi) * cos(theta)
656         //y = pho * sin(phi) * sin(theta)
657         //z = pho * cos(theta);
658
659         double phi{ v.GetY() * PI / 180.0 };
660         double theta{ v.GetZ() * PI / 180.0 };
661
662         return Vector3D(v.GetX() * sin(phi) * cos(theta), v.GetX() * sin(phi) * sin(theta), v.GetX() *
        cos(theta));
663     }
664
665     inline Vector3D CartesianToSpherical(const Vector3D& v)
666     {
667         //v = (x, y, z)
668         //pho = sqrt(vx^2 + vy^2 + vz^2)
669         //phi = arcos(z / pho)
670         //theta = arctan(y / x)
671
672         if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
673         {
674             return v;
675         }
676
677         double pho{ Length(v) };
678         double phi{ acos(v.GetZ() / pho) * 180.0 / PI };
679         double theta{ atan2(v.GetY(), v.GetX()) * 180.0 / PI };
680
681         return Vector3D(pho, phi, theta);
682     }
683
684     inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
685     {
686         //Projb(a) = (a dot b)b
687         //normalize b before projecting
688
689         Vector3D normB(Norm(b));
690         return Vector3D(DotProduct(a, normB) * normB);
691     }
692
693     inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
694     {
695         x = Norm(x);
696         y = Norm(CrossProduct(z, x));
697         z = Norm(CrossProduct(x, y));
698     }
699
700 #if defined(_DEBUG)
701     inline void print(const Vector3D& v)
702     {
703         std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ")";
704     }
705 #endif
706 //-----
707
708 //-----
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731

```

```

732
733 //-----
734 class Vector4D
735 {
736 public:
737     Vector4D();
738
739     Vector4D(float x, float y, float z, float w);
740
741     float GetX() const;
742
743     float GetY() const;
744
745     float GetZ() const;
746
747     float GetW() const;
748
749     void SetX(float x);
750
751     void SetY(float y);
752
753     void SetZ(float z);
754
755     void SetW(float w);
756
757     Vector4D& operator+=(const Vector4D& b);
758
759     Vector4D& operator-=(const Vector4D& b);
760
761     Vector4D& operator*=(float k);
762
763     Vector4D& operator/=(float k);
764
765 private:
766     float mX;
767     float mY;
768     float mZ;
769     float mW;
770 };
771
772 //-----
773 //Vector4D Constructors
774
775 inline Vector4D::Vector4D() : mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }, mW{ 0.0f }
776 {}
777
778 inline Vector4D::Vector4D(float x, float y, float z, float w) : mX{ x }, mY{ y }, mZ{ z }, mW{ w }
779 {}
780
781 //-----
782 //-----
783 //Vector4D Getters and Setters
784
785 inline float Vector4D::GetX()const
786 {
787     return mX;
788 }
789
790 inline float Vector4D::GetY()const
791 {
792     return mY;
793 }
794
795 inline float Vector4D::GetZ()const
796 {
797     return mZ;
798 }
799
800 inline float Vector4D::GetW()const
801 {
802     return mW;
803 }
804
805 inline void Vector4D::SetX(float x)
806 {
807     mX = x;
808 }
809
810 inline void Vector4D::SetY(float y)
811 {
812     mY = y;
813 }
814
815 inline void Vector4D::SetZ(float z)
816 {
817     mZ = z;

```

```

857     }
858
859     inline void Vector4D::SetW(float w)
860     {
861         mW = w;
862     }
863     //-----
864
865     //-----
866     //Vector4D Member functions
867
868     inline Vector4D& Vector4D::operator+=(const Vector4D& b)
869     {
870         this->mX += b.mX;
871         this->mY += b.mY;
872         this->mZ += b.mZ;
873         this->mW += b.mW;
874
875         return *this;
876     }
877
878     inline Vector4D& Vector4D::operator-=(const Vector4D& b)
879     {
880         this->mX -= b.mX;
881         this->mY -= b.mY;
882         this->mZ -= b.mZ;
883         this->mW -= b.mW;
884
885         return *this;
886     }
887
888     inline Vector4D& Vector4D::operator*=(float k)
889     {
890         this->mX *= k;
891         this->mY *= k;
892         this->mZ *= k;
893         this->mW *= k;
894
895         return *this;
896     }
897
898     inline Vector4D& Vector4D::operator/=(float k)
899     {
900         if (CompareFloats(k, 0.0f, EPSILON))
901         {
902             return *this;
903         }
904
905         this->mX /= k;
906         this->mY /= k;
907         this->mZ /= k;
908         this->mW /= k;
909
910         return *this;
911     }
912 }
913
914 //-----
915 //-----
916 //Vector4D Non-member functions
917
918 inline bool ZeroVector(const Vector4D& a)
919 {
920     if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
921         CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
922     {
923         return true;
924     }
925
926     return false;
927 }
928
929 inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
930 {
931     return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
932 b.GetW());
933 }
934
935 inline Vector4D operator-(const Vector4D& v)
936 {
937     return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
938 }
939
940 inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
941 {
942     return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -

```

```

    b.GetW());
951 }
952
953 inline Vector4D operator*(const Vector4D& a, float k)
954 {
955     return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
956 }
957
958 inline Vector4D operator*(float k, const Vector4D& a)
959 {
960     return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
961 }
962
963 inline Vector4D operator/(const Vector4D& a, float k)
964 {
965     if (CompareFloats(k, 0.0f, EPSILON))
966     {
967         return Vector4D();
968     }
969     return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
970 }
971
972 inline float DotProduct(const Vector4D& a, const Vector4D& b)
973 {
974     //a dot b = axbx + ayby + azbz + awbw
975     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
976 }
977
978 inline float Length(const Vector4D& v)
979 {
980     //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
981     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
982 v.GetW());
983 }
984
985 inline Vector4D Norm(const Vector4D& v)
986 {
987     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
988     //v is the zero vector
989     if (ZeroVector(v))
990     {
991         return v;
992     }
993     float mag{ Length(v) };
994     return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
995 }
996
997 inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
998 {
999     //Projb(a) = (a dot b)b
1000     //normalize b before projecting
1001     Vector4D normB(Norm(b));
1002     return Vector4D(DotProduct(a, normB) * normB);
1003 }
1004
1005 #if defined(_DEBUG)
1006 inline void print(const Vector4D& v)
1007 {
1008     std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ", " << v.GetW() << ")";
1009 }
1010 #endif
1011 //-----
1012
1013 //-----
1014
1015 //-----
1016
1017 class Matrix4x4
1018 {
1019 public:
1020
1021     Matrix4x4();
1022     Matrix4x4(float a[][4]);
1023     Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
1024
1025     float* Data();
1026     const float* Data() const;
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076

```



```

1080     const float& operator()(unsigned int row, unsigned int col) const;
1081
1085     float& operator()(unsigned int row, unsigned int col);
1086
1090     Vector4D GetRow(unsigned int row) const;
1091
1095     Vector4D GetCol(unsigned int col) const;
1096
1100     void SetRow(unsigned int row, Vector4D v);
1101
1105     void SetCol(unsigned int col, Vector4D v);
1106
1109     Matrix4x4& operator+=(const Matrix4x4& m);
1110
1113     Matrix4x4& operator-=(const Matrix4x4& m);
1114
1117     Matrix4x4& operator*=(float k);
1118
1121     Matrix4x4& operator*=(const Matrix4x4& m);
1122
1123 private:
1124     float mMat[4][4];
1125 };
1126
1127 //-----
1128 inline Matrix4x4::Matrix4x4()
1129 {
1130     //1st row
1131     mMat[0][0] = 1.0f;
1132     mMat[0][1] = 0.0f;
1133     mMat[0][2] = 0.0f;
1134     mMat[0][3] = 0.0f;
1135
1136     //2nd
1137     mMat[1][0] = 0.0f;
1138     mMat[1][1] = 1.0f;
1139     mMat[1][2] = 0.0f;
1140     mMat[1][3] = 0.0f;
1141
1142     //3rd row
1143     mMat[2][0] = 0.0f;
1144     mMat[2][1] = 0.0f;
1145     mMat[2][2] = 1.0f;
1146     mMat[2][3] = 0.0f;
1147
1148     //4th row
1149     mMat[3][0] = 0.0f;
1150     mMat[3][1] = 0.0f;
1151     mMat[3][2] = 0.0f;
1152     mMat[3][3] = 1.0f;
1153 }
1154
1155
1156
1157
1158 inline Matrix4x4::Matrix4x4(float a[][4])
1159 {
1160     //1st row
1161     mMat[0][0] = a[0][0];
1162     mMat[0][1] = a[0][1];
1163     mMat[0][2] = a[0][2];
1164     mMat[0][3] = a[0][3];
1165
1166     //2nd
1167     mMat[1][0] = a[1][0];
1168     mMat[1][1] = a[1][1];
1169     mMat[1][2] = a[1][2];
1170     mMat[1][3] = a[1][3];
1171
1172     //3rd row
1173     mMat[2][0] = a[2][0];
1174     mMat[2][1] = a[2][1];
1175     mMat[2][2] = a[2][2];
1176     mMat[2][3] = a[2][3];
1177
1178     //4th row
1179     mMat[3][0] = a[3][0];
1180     mMat[3][1] = a[3][1];
1181     mMat[3][2] = a[3][2];
1182     mMat[3][3] = a[3][3];
1183 }
1184
1185 inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
1186 {
1187     SetRow(0, r1);
1188     SetRow(1, r2);

```

```

1189         SetRow(2, r3);
1190         SetRow(3, r4);
1191     }
1192
1193     inline float* Matrix4x4::Data()
1194     {
1195         return mMat[0];
1196     }
1197
1198     inline const float* Matrix4x4::Data()const
1199     {
1200         return mMat[0];
1201     }
1202
1203     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
1204     {
1205         if (row > 3 || col > 3)
1206         {
1207             return mMat[0][0];
1208         }
1209         else
1210         {
1211             return mMat[row][col];
1212         }
1213     }
1214
1215     inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
1216     {
1217         if (row > 3 || col > 3)
1218         {
1219             return mMat[0][0];
1220         }
1221         else
1222         {
1223             return mMat[row][col];
1224         }
1225     }
1226
1227     inline Vector4D Matrix4x4::GetRow(unsigned int row)const
1228     {
1229         if (row < 0 || row > 3)
1230             return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
1231         else
1232             return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);
1233     }
1234
1235     inline Vector4D Matrix4x4::GetCol(unsigned int col)const
1236     {
1237         if (col < 0 || col > 3)
1238             return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
1239         else
1240             return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
1241     }
1242
1243     inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
1244     {
1245         if (row < 0 || row > 3)
1246         {
1247             mMat[0][0] = v.GetX();
1248             mMat[0][1] = v.GetY();
1249             mMat[0][2] = v.GetZ();
1250             mMat[0][3] = v.GetW();
1251         }
1252         else
1253         {
1254             mMat[row][0] = v.GetX();
1255             mMat[row][1] = v.GetY();
1256             mMat[row][2] = v.GetZ();
1257             mMat[row][3] = v.GetW();
1258         }
1259     }
1260
1261     inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
1262     {
1263         if (col < 0 || col > 3)
1264         {
1265             mMat[0][0] = v.GetX();
1266             mMat[1][0] = v.GetY();
1267             mMat[2][0] = v.GetZ();
1268             mMat[3][0] = v.GetW();
1269         }
1270         else
1271         {
1272             mMat[0][col] = v.GetX();
1273             mMat[1][col] = v.GetY();
1274             mMat[2][col] = v.GetZ();
1275         }
1276     }

```

```

1276         mMat[3][col] = v.GetW();
1277     }
1278 }
1279
1280 inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
1281 {
1282     for (int i = 0; i < 4; ++i)
1283     {
1284         for (int j = 0; j < 4; ++j)
1285         {
1286             this->mMat[i][j] += m.mMat[i][j];
1287         }
1288     }
1289
1290     return *this;
1291 }
1292
1293 inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
1294 {
1295     for (int i = 0; i < 4; ++i)
1296     {
1297         for (int j = 0; j < 4; ++j)
1298         {
1299             this->mMat[i][j] -= m.mMat[i][j];
1300         }
1301     }
1302
1303     return *this;
1304 }
1305
1306 inline Matrix4x4& Matrix4x4::operator*=(float k)
1307 {
1308     for (int i = 0; i < 4; ++i)
1309     {
1310         for (int j = 0; j < 4; ++j)
1311         {
1312             this->mMat[i][j] *= k;
1313         }
1314     }
1315
1316     return *this;
1317 }
1318
1319 inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
1320 {
1321     Matrix4x4 res;
1322
1323     for (int i = 0; i < 4; ++i)
1324     {
1325         res.mMat[i][0] = (mMat[i][0] * m.mMat[0][0]) +
1326             (mMat[i][1] * m.mMat[1][0]) +
1327             (mMat[i][2] * m.mMat[2][0]) +
1328             (mMat[i][3] * m.mMat[3][0]);
1329
1330         res.mMat[i][1] = (mMat[i][0] * m.mMat[0][1]) +
1331             (mMat[i][1] * m.mMat[1][1]) +
1332             (mMat[i][2] * m.mMat[2][1]) +
1333             (mMat[i][3] * m.mMat[3][1]);
1334
1335         res.mMat[i][2] = (mMat[i][0] * m.mMat[0][2]) +
1336             (mMat[i][1] * m.mMat[1][2]) +
1337             (mMat[i][2] * m.mMat[2][2]) +
1338             (mMat[i][3] * m.mMat[3][2]);
1339
1340         res.mMat[i][3] = (mMat[i][0] * m.mMat[0][3]) +
1341             (mMat[i][1] * m.mMat[1][3]) +
1342             (mMat[i][2] * m.mMat[2][3]) +
1343             (mMat[i][3] * m.mMat[3][3]);
1344     }
1345
1346     for (int i = 0; i < 4; ++i)
1347     {
1348         for (int j = 0; j < 4; ++j)
1349         {
1350             mMat[i][j] = res.mMat[i][j];
1351         }
1352     }
1353
1354     return *this;
1355 }
1356
1357 inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
1358 {
1359     Matrix4x4 res;
1360     for (int i = 0; i < 4; ++i)
1361     {
1362         for (int j = 0; j < 4; ++j)

```

```

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

```

```

1462         (m1(i, 2) * m2(2, 3)) +
1463         (m1(i, 3) * m2(3, 3));
1464     }
1465
1466     return res;
1467 }
1468
1472 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
1473 {
1474     Vector4D res;
1475
1476     res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());
1477     res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());
1478     res.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());
1479     res.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());
1480
1481     return res;
1482 }
1483
1484 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
1485 {
1486     Vector4D res;
1487
1488     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));
1489     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));
1490     res.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));
1491     res.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));
1492
1493     return res;
1494 }
1495
1500 inline void SetToIdentity(Matrix4x4& m)
1501 {
1502     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1503
1504     //1st row
1505     m(0, 0) = 1.0f;
1506     m(0, 1) = 0.0f;
1507     m(0, 2) = 0.0f;
1508     m(0, 3) = 0.0f;
1509
1510     //2nd row
1511     m(1, 0) = 0.0f;
1512     m(1, 1) = 1.0f;
1513     m(1, 2) = 0.0f;
1514     m(1, 3) = 0.0f;
1515
1516     //3rd row
1517     m(2, 0) = 0.0f;
1518     m(2, 1) = 0.0f;
1519     m(2, 2) = 1.0f;
1520     m(2, 3) = 0.0f;
1521
1522     //4th row
1523     m(3, 0) = 0.0f;
1524     m(3, 1) = 0.0f;
1525     m(3, 2) = 0.0f;
1526     m(3, 3) = 1.0f;
1527 }
1528
1529 inline bool IsIdentity(const Matrix4x4& m)
1530 {
1531     //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
1532     0.0f
1533
1534     for (int i = 0; i < 4; ++i)
1535     {
1536         for (int j = 0; j < 4; ++j)
1537         {
1538             if (i == j)
1539             {
1540                 if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1541                     return false;
1542             }
1543             else
1544             {
1545                 if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1546                     return false;
1547             }
1548         }
1549     }
1550 }

```

```

1558     }
1559 }
1560 }
1561
1562 inline Matrix4x4 Transpose(const Matrix4x4& m)
1563 {
1564     //make the rows into cols
1565
1566     Matrix4x4 res;
1567
1568     //1st col = 1st row
1569     res(0, 0) = m(0, 0);
1570     res(1, 0) = m(0, 1);
1571     res(2, 0) = m(0, 2);
1572     res(3, 0) = m(0, 3);
1573
1574     //2nd col = 2nd row
1575     res(0, 1) = m(1, 0);
1576     res(1, 1) = m(1, 1);
1577     res(2, 1) = m(1, 2);
1578     res(3, 1) = m(1, 3);
1579
1580     //3rd col = 3rd row
1581     res(0, 2) = m(2, 0);
1582     res(1, 2) = m(2, 1);
1583     res(2, 2) = m(2, 2);
1584     res(3, 2) = m(2, 3);
1585
1586     //4th col = 4th row
1587     res(0, 3) = m(3, 0);
1588     res(1, 3) = m(3, 1);
1589     res(2, 3) = m(3, 2);
1590     res(3, 3) = m(3, 3);
1591
1592     return res;
1593 }
1594
1595 inline Matrix4x4 Translate(const Matrix4x4& cm, float x, float y, float z)
1596 {
1597     //1 0 0 0
1598     //0 1 0 0
1599     //0 0 1 0
1600     //x y z 1
1601
1602     Matrix4x4 t;
1603     t(3, 0) = x;
1604     t(3, 1) = y;
1605     t(3, 2) = z;
1606
1607     return cm * t;
1608 }
1609
1610 inline Matrix4x4 Scale(const Matrix4x4& cm, float x, float y, float z)
1611 {
1612     //x 0 0 0
1613     //0 y 0 0
1614     //0 0 z 0
1615     //0 0 0 1
1616
1617     Matrix4x4 s;
1618     s(0, 0) = x;
1619     s(1, 1) = y;
1620     s(2, 2) = z;
1621
1622     return cm * s;
1623 }
1624
1625 inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
1626 {
1627     //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy    0
1628     //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx    0
1629     //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2    0
1630     //0                0                0                1
1631     //c = cos(angle)
1632     //s = sin(angle)
1633
1634     double c = cos(angle * PI / 180.0);
1635     double s = sin(angle * PI / 180.0);
1636
1637     Matrix4x4 r;
1638
1639     //1st row
1640     r(0, 0) = c + (1.0 - c) * (x * x);
1641     r(0, 1) = (1.0 - c) * (x * y) + (s * z);
1642     r(0, 2) = (1.0 - c) * (x * z) - (s * y);
1643
1644 }

```

```

1656         //2nd row
1657         r(1, 0) = (1.0 - c) * (x * y) - (s * z);
1658         r(1, 1) = c + (1.0 - c) * (y * y);
1659         r(1, 2) = (1.0 - c) * (y * z) + (s * x);
1660
1661         //3rd row
1662         r(2, 0) = (1.0 - c) * (x * z) + (s * y);
1663         r(2, 1) = (1.0 - c) * (y * z) - (s * x);
1664         r(2, 2) = c + (1.0 - c) * (z * z);
1665
1666         return cm * r;
1667     }
1668
1669     inline double Det(const Matrix4x4& m)
1670     {
1671         //m00m11(m22m33 - m23m32)
1672         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)
1673 * m(3, 2);
1674
1675         //m00m12(m23m31 - m21m33)
1676         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)
1677 * m(3, 3);
1678
1679         //m00m13(m21m32 - m22m31)
1680         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)
1681 * m(3, 1);
1682
1683         //m01m10(m22m33 - m23m32)
1684         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)
1685 * m(3, 2);
1686
1687         //m01m12(m23m30 - m20m33)
1688         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)
1689 * m(3, 3);
1690
1691         //m01m13(m20m32 - m22m30)
1692         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)
1693 * m(3, 0);
1694
1695         //m02m10(m21m33 - m23m31)
1696         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)
1697 * m(3, 1);
1698
1699         //m02m11(m23m30 - m20m33)
1700         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)
1701 * m(3, 3);
1702
1703         //m02m13(m20m31 - m21m30)
1704         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)
1705 * m(3, 0);
1706
1707         //m03m10(m21m32 - m22m21)
1708         double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
1709 2) * m(3, 1);
1710
1711         //m03m11(m22m30 - m20m32)
1712         double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
1713 0) * m(3, 2);
1714
1715         //m03m12(m20m31 - m21m30)
1716         double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
1717 1) * m(3, 0);
1718
1719         return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
1720     }
1721
1722     inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
1723     {
1724         //cij = (-1)^(i + j) * det of minor(i, j);
1725         double tempMat[3][3]{};
1726         int tr{ 0 };
1727         int tc{ 0 };
1728
1729         //minor(i, j)
1730         for (int i = 0; i < 4; ++i)
1731         {
1732             if (i == row)
1733                 continue;
1734
1735             for (int j = 0; j < 4; ++j)
1736             {
1737                 if (j == col)
1738                     continue;
1739
1740                 tempMat[tr][tc] = m(i, j);
1741                 ++tc;
1742             }
1743             ++tr;
1744         }

```

```

1735         }
1736         tc = 0;
1737         ++tr;
1738     }
1739
1740     //determinant of minor(i, j)
1741     double det3x3 = (tempMat[0][0] * tempMat[1][1] * tempMat[2][2]) + (tempMat[0][1] *
tempMat[1][2] * tempMat[2][0]) +
1742     (tempMat[0][2] * tempMat[1][0] * tempMat[2][1]) - (tempMat[0][2] * tempMat[1][1] *
tempMat[2][0]) -
1743     (tempMat[0][1] * tempMat[1][0] * tempMat[2][2]) - (tempMat[0][0] * tempMat[1][2] *
tempMat[2][1]);
1744
1745     return pow(-1, row + col) * det3x3;
1746 }
1747
1750 inline Matrix4x4 Adjoint(const Matrix4x4& m)
1751 {
1752     //Cofactor of each ijth position put into matrix cA.
1753     //Adjoint is the tranposed matrix of cA.
1754     Matrix4x4 cA;
1755     for (int i = 0; i < 4; ++i)
1756     {
1757         for (int j = 0; j < 4; ++j)
1758         {
1759             cA(i, j) = Cofactor(m, i, j);
1760         }
1761     }
1762
1763     return Transpose(cA);
1764 }
1765
1769 inline Matrix4x4 Inverse(const Matrix4x4& m)
1770 {
1771     //Inverse of m = adjoint of m / det of m
1772     double determinant = Det(m);
1773     if (CompareDoubles(determinant, 0.0, EPSILON))
1774         return Matrix4x4();
1775
1776     return Adjoint(m) * (1.0 / determinant);
1777 }
1778
1779
1780 #if defined(_DEBUG)
1781 inline void print(const Matrix4x4& m)
1782 {
1783     for (int i = 0; i < 4; ++i)
1784     {
1785         for (int j = 0; j < 4; ++j)
1786         {
1787             std::cout << m(i, j) << " ";
1788         }
1789
1790         std::cout << std::endl;
1791     }
1792 }
1793 #endif
1794
1795
1796 //-----
1797
1798 //-----
1799
1800 class Quaternion
1801 {
1802 public:
1803
1804     Quaternion();
1805
1806     Quaternion(float scalar, float x, float y, float z);
1807
1808     Quaternion(float scalar, const Vector3D& v);
1809
1810     Quaternion(const Vector4D& v);
1811
1812     float GetScalar() const;
1813
1814     float GetX() const;
1815
1816     float GetY() const;
1817
1818     float GetZ() const;
1819
1820     const Vector3D& GetVector() const;

```



```

1853
1856     void SetScalar(float scalar);
1857
1860     void SetX(float x);
1861
1864     void SetY(float y);
1865
1868     void SetZ(float z);
1869
1872     void SetVector(const Vector3D& v);
1873
1876     Quaternion& operator+=(const Quaternion& q);
1877
1880     Quaternion& operator-=(const Quaternion& q);
1881
1884     Quaternion& operator*=(float k);
1885
1888     Quaternion& operator*=(const Quaternion& q);
1889
1890 private:
1891     float mScalar;
1892     float mX;
1893     float mY;
1894     float mZ;
1895 };
1896
1897 //-----
1898 inline Quaternion::Quaternion() : mScalar{ 1.0f }, mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }
1899 {
1900 }
1901
1902 inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
1903     mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
1904 {
1905 }
1906
1907 inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
1908     mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
1909 {
1910 }
1911
1912 inline Quaternion::Quaternion(const Vector4D& v) :
1913     mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
1914 {
1915 }
1916
1917 inline float Quaternion::GetScalar() const
1918 {
1919     return mScalar;
1920 }
1921
1922 inline float Quaternion::GetX() const
1923 {
1924     return mX;
1925 }
1926
1927 inline float Quaternion::GetY() const
1928 {
1929     return mY;
1930 }
1931
1932 inline float Quaternion::GetZ() const
1933 {
1934     return mZ;
1935 }
1936
1937 inline const Vector3D& Quaternion::GetVector() const
1938 {
1939     return Vector3D(mX, mY, mZ);
1940 }
1941
1942 inline void Quaternion::SetScalar(float scalar)
1943 {
1944     mScalar = scalar;
1945 }
1946
1947 inline void Quaternion::SetX(float x)
1948 {
1949     mX = x;
1950 }
1951
1952 inline void Quaternion::SetY(float y)
1953 {
1954     mY = y;
1955 }
1956
1957 inline void Quaternion::SetZ(float z)

```

```

1958     {
1959         mZ = z;
1960     }
1961
1962     inline void Quaternion::SetVector(const Vector3D& v)
1963     {
1964         mX = v.GetX();
1965         mY = v.GetY();
1966         mZ = v.GetZ();
1967     }
1968
1969     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
1970     {
1971         this->mScalar += q.mScalar;
1972         this->mX += q.mX;
1973         this->mY += q.mY;
1974         this->mZ += q.mZ;
1975
1976         return *this;
1977     }
1978
1979     inline Quaternion& Quaternion::operator-=(const Quaternion& q)
1980     {
1981         this->mScalar -= q.mScalar;
1982         this->mX -= q.mX;
1983         this->mY -= q.mY;
1984         this->mZ -= q.mZ;
1985
1986         return *this;
1987     }
1988
1989     inline Quaternion& Quaternion::operator*=(float k)
1990     {
1991         this->mScalar *= k;
1992         this->mX *= k;
1993         this->mY *= k;
1994         this->mZ *= k;
1995
1996         return *this;
1997     }
1998
1999     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
2000     {
2001         Vector3D thisVector(this->mX, this->mY, this->mZ);
2002         Vector3D qVector(q.mX, q.mY, q.mZ);
2003
2004         double s{ (double)this->mScalar * q.mScalar };
2005         double dP{ DotProduct(thisVector, qVector) };
2006         double resultScalar{ s - dP };
2007
2008         Vector3D a(this->mScalar * qVector);
2009         Vector3D b(q.mScalar * thisVector);
2010         Vector3D cP(CrossProduct(thisVector, qVector));
2011         Vector3D resultVector(a + b + cP);
2012
2013         this->mScalar = resultScalar;
2014         this->mX = resultVector.GetX();
2015         this->mY = resultVector.GetY();
2016         this->mZ = resultVector.GetZ();
2017
2018         return *this;
2019     }
2020
2021     inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
2022     {
2023         return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
2024             q2.GetY(), q1.GetZ() + q2.GetZ());
2025     }
2026
2027     inline Quaternion operator-(const Quaternion& q)
2028     {
2029         return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2030     }
2031
2032     inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
2033     {
2034         return Quaternion(q1.GetScalar() - q2.GetScalar(),
2035             q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
2036     }
2037
2038     inline Quaternion operator*(float k, const Quaternion& q)
2039     {
2040         return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
2041     }
2042
2043     inline Quaternion operator*(const Quaternion& q, float k)
2044     {

```

```

2054     return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
2055 }
2056
2059 inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
2060 {
2061     //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
2062     //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
2063
2064     Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
2065     Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());
2066
2067     float s{ q1.GetScalar() * q2.GetScalar() };
2068     float dP{ DotProduct(q1Vector, q2Vector) };
2069     float resultScalar{ s - dP };
2070
2071     Vector3D a(q1.GetScalar() * q2Vector);
2072     Vector3D b(q2.GetScalar() * q1Vector);
2073     Vector3D cP(CrossProduct(q1Vector, q2Vector));
2074     Vector3D resultVector(a + b + cP);
2075
2076     return Quaternion(resultScalar, resultVector);
2077 }
2078
2081 inline bool IsZeroQuaternion(const Quaternion& q)
2082 {
2083     //zero quaternion = (0, 0, 0, 0)
2084     return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2085         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2086 }
2087
2090 inline bool IsIdentity(const Quaternion& q)
2091 {
2092     //identity quaternion = (1, 0, 0, 0)
2093     return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2094         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2095 }
2096
2099 inline Quaternion Conjugate(const Quaternion& q)
2100 {
2101     //conjugate of a quaternion is the quaternion with its vector part negated
2102     return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2103 }
2104
2107 inline float Length(const Quaternion& q)
2108 {
2109     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
2110     return sqrt(q.GetScalar() * q.GetScalar() + q.GetX() * q.GetX() + q.GetY() * q.GetY() +
2111         q.GetZ() * q.GetZ());
2112 }
2113
2116 inline Quaternion Normalize(const Quaternion& q)
2117 {
2118     //to normalize a quaternion you do q / |q|
2119
2120     if (IsZeroQuaternion(q))
2121         return q;
2122
2123     double d{ Length(q) };
2124
2125     return Quaternion(q.GetScalar() / d, q.GetX() / d, q.GetY() / d, q.GetZ() / d);
2126 }
2127
2131 inline Quaternion Inverse(const Quaternion& q)
2132 {
2133     //inverse = conjugate of q / |q|^2
2134
2135     if (IsZeroQuaternion(q))
2136         return q;
2137
2138     Quaternion conjugateOfQ(Conjugate(q));
2139
2140     double d{ Length(q) };
2141     d *= d;
2142
2143     return Quaternion(conjugateOfQ.GetScalar() / d, conjugateOfQ.GetX() / d,
2144         conjugateOfQ.GetY() / d, conjugateOfQ.GetZ() / d);
2145 }
2146
2150 inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
2151 {
2152     //A roatation quaternion is a quaternion where the
2153     //scalar part = cos(theta / 2)
2154     //vector part = sin(theta / 2) * axis
2155     //the axis needs to be normalized
2156
2157     double ang{ angle / 2.0 };
2158     double c{ cos(ang * PI / 180.0) };

```

```

2159     double s{ sin(ang * PI / 180.0) };
2160
2161     Vector3D axis(x, y, z);
2162     axis = Norm(axis);
2163
2164     return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2165 }
2166
2167 inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
2168 {
2169     //A roatation quaternion is a quaternion where the
2170     //scalar part = cos(theta / 2)
2171     //vector part = sin(theta / 2) * axis
2172     //the axis needs to be normalized
2173
2174     double ang{ angle / 2.0 };
2175     double c{ cos(ang * PI / 180.0) };
2176     double s{ sin(ang * PI / 180.0) };
2177
2178     Vector3D axisN(Norm(axis));
2179
2180     return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
2181 }
2182
2183 inline Quaternion RotationQuaternion(const Vector4D& angAxis)
2184 {
2185     //A roatation quaternion is a quaternion where the
2186     //scalar part = cos(theta / 2)
2187     //vector part = sin(theta / 2) * axis
2188     //the axis needs to be normalized
2189
2190     double angle{ angAxis.GetX() / 2.0 };
2191     double c{ cos(angle * PI / 180.0) };
2192     double s{ sin(angle * PI / 180.0) };
2193
2194     Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
2195     axis = Norm(axis);
2196
2197     return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2198 }
2199
2200 inline Matrix4x4 QuaternionToRotationMatrixCol(const Quaternion& q)
2201 {
2202     //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
2203     //2q2q3 + 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
2204     //2q2q4 - 2q1q3      2q3q4 + 2q1q2      1 - 2q2^2 - 2q3^2      0
2205     //0                      0                      0                      1
2206     //q1 = scalar
2207     //q2 = x
2208     //q3 = y
2209     //q4 = z
2210
2211     float colMat[4][4] = {};
2212
2213     colMat[0][0] = 1.0 - 2.0 * q.GetY() * q.GetY() - 2.0 * q.GetZ() * q.GetZ();
2214     colMat[0][1] = 2.0 * q.GetX() * q.GetY() - 2.0 * q.GetScalar() * q.GetZ();
2215     colMat[0][2] = 2.0 * q.GetX() * q.GetZ() + 2.0 * q.GetScalar() * q.GetY();
2216     colMat[0][3] = 0.0f;
2217
2218     colMat[1][0] = 2.0 * q.GetX() * q.GetY() + 2.0 * q.GetScalar() * q.GetZ();
2219     colMat[1][1] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetZ() * q.GetZ();
2220     colMat[1][2] = 2.0 * q.GetY() * q.GetZ() - 2.0 * q.GetScalar() * q.GetX();
2221     colMat[1][3] = 0.0f;
2222
2223     colMat[2][0] = 2.0 * q.GetX() * q.GetZ() - 2.0 * q.GetScalar() * q.GetY();
2224     colMat[2][1] = 2.0 * q.GetY() * q.GetZ() + 2.0 * q.GetScalar() * q.GetX();
2225     colMat[2][2] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetY() * q.GetY();
2226     colMat[2][3] = 0.0f;
2227
2228     colMat[3][0] = 0.0f;
2229     colMat[3][1] = 0.0f;
2230     colMat[3][2] = 0.0f;
2231     colMat[3][3] = 1.0f;
2232
2233     return Matrix4x4(colMat);
2234 }
2235
2236 inline Matrix4x4 QuaternionToRotationMatrixRow(const Quaternion& q)
2237 {
2238     //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
2239     //2q2q3 - 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
2240     //2q2q4 + 2q1q3      2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2      0
2241     //0                      0                      0                      1
2242     //q1 = scalar
2243     //q2 = x
2244     //q3 = y
2245     //q4 = z

```

```

2259
2260     float rowMat[4][4] = {};
2261
2262     rowMat[0][0] = 1.0 - 2.0 * q.GetY() * q.GetY() - 2.0 * q.GetZ() * q.GetZ();
2263     rowMat[0][1] = 2.0 * q.GetX() * q.GetY() + 2.0 * q.GetScalar() * q.GetZ();
2264     rowMat[0][2] = 2.0 * q.GetX() * q.GetZ() - 2.0 * q.GetScalar() * q.GetY();
2265     rowMat[0][3] = 0.0f;
2266
2267     rowMat[1][0] = 2.0 * q.GetX() * q.GetY() - 2.0 * q.GetScalar() * q.GetZ();
2268     rowMat[1][1] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetZ() * q.GetZ();
2269     rowMat[1][2] = 2.0 * q.GetY() * q.GetZ() + 2.0 * q.GetScalar() * q.GetX();
2270     rowMat[1][3] = 0.0f;
2271
2272     rowMat[2][0] = 2.0 * q.GetX() * q.GetZ() + 2.0 * q.GetScalar() * q.GetY();
2273     rowMat[2][1] = 2.0 * q.GetY() * q.GetZ() - 2.0 * q.GetScalar() * q.GetX();
2274     rowMat[2][2] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetY() * q.GetY();
2275     rowMat[2][3] = 0.0f;
2276
2277     rowMat[3][0] = 0.0f;
2278     rowMat[3][1] = 0.0f;
2279     rowMat[3][2] = 0.0f;
2280     rowMat[3][3] = 1.0f;
2281
2282     return Matrix4x4(rowMat);
2283 }
2284
2285 #if defined(_DEBUG)
2286     inline void print(const Quaternion& q)
2287     {
2288         std::cout << "(" << q.GetScalar() << ", " << q.GetX() << ", " << q.GetY() << ", " << q.GetZ();
2289     }
2290 #endif
2291     //-----
2292
2293     //-----
2294 }

```


Index

- Adjoint
 - FAMath, [11](#)
- CartesianToCylindrical
 - FAMath, [11](#)
- CartesianToPolar
 - FAMath, [11](#)
- CartesianToSpherical
 - FAMath, [11](#)
- Cofactor
 - FAMath, [12](#)
- Conjugate
 - FAMath, [12](#)
- CrossProduct
 - FAMath, [12](#)
- CylindricalToCartesian
 - FAMath, [12](#)
- Data
 - FAMath::Matrix4x4, [29](#)
- Det
 - FAMath, [12](#)
- DotProduct
 - FAMath, [13](#)
- FAMath, [7](#)
 - Adjoint, [11](#)
 - CartesianToCylindrical, [11](#)
 - CartesianToPolar, [11](#)
 - CartesianToSpherical, [11](#)
 - Cofactor, [12](#)
 - Conjugate, [12](#)
 - CrossProduct, [12](#)
 - CylindricalToCartesian, [12](#)
 - Det, [12](#)
 - DotProduct, [13](#)
 - Inverse, [13](#)
 - IsIdentity, [14](#)
 - IsZeroQuaternion, [14](#)
 - Length, [14](#), [15](#)
 - Norm, [15](#)
 - Normalize, [15](#)
 - operator*, [15–18](#)
 - operator+, [18](#), [19](#)
 - operator-, [19–21](#)
 - operator/, [21](#), [22](#)
 - Orthonormalize, [22](#)
 - PolarToCartesian, [22](#)
 - Projection, [22](#), [23](#)
 - QuaternionToRotationMatrixCol, [23](#)
 - QuaternionToRotationMatrixRow, [23](#)
 - Rotate, [23](#)
 - RotationQuaternion, [23](#), [24](#)
 - Scale, [24](#)
 - SetToIdentity, [24](#)
 - SphericalToCartesian, [24](#)
 - Translate, [25](#)
 - Transpose, [25](#)
 - ZeroVector, [25](#)
- FAMath::Matrix4x4, [27](#)
 - Data, [29](#)
 - GetCol, [29](#)
 - GetRow, [29](#)
 - Matrix4x4, [28](#)
 - operator*=[, 30](#)
 - operator()[, 29](#)
 - operator+=[, 30](#)
 - operator-=[, 30](#)
 - SetCol, [30](#)
 - SetRow, [31](#)
- FAMath::Quaternion, [31](#)
 - GetScalar, [33](#)
 - GetVector, [33](#)
 - GetX, [33](#)
 - GetY, [33](#)
 - GetZ, [34](#)
 - operator*=[, 34](#)
 - operator+=[, 34](#)
 - operator-=[, 34](#)
 - Quaternion, [32](#), [33](#)
 - SetScalar, [34](#)
 - SetVector, [35](#)
 - SetX, [35](#)
 - SetY, [35](#)
 - SetZ, [35](#)
- FAMath::Vector2D, [36](#)
 - GetX, [37](#)
 - GetY, [37](#)
 - operator*=[, 37](#)
 - operator+=[, 37](#)
 - operator-=[, 37](#)
 - operator/=[, 38](#)
 - SetX, [38](#)
 - SetY, [38](#)
 - Vector2D, [36](#)
- FAMath::Vector3D, [38](#)
 - GetX, [40](#)
 - GetY, [40](#)
 - GetZ, [40](#)

- operator*=, 40
- operator+=, 40
- operator-=, 41
- operator/=: 41
- SetX, 41
- SetY, 41
- SetZ, 41
- Vector3D, 39
- FAMath::Vector4D, 42
 - GetW, 43
 - GetX, 43
 - GetY, 43
 - GetZ, 44
 - operator*=, 44
 - operator+=, 44
 - operator-=, 44
 - operator/=: 44
 - SetW, 44
 - SetX, 45
 - SetY, 45
 - SetZ, 45
 - Vector4D, 43
- GetCol
 - FAMath::Matrix4x4, 29
- GetRow
 - FAMath::Matrix4x4, 29
- GetScalar
 - FAMath::Quaternion, 33
- GetVector
 - FAMath::Quaternion, 33
- GetW
 - FAMath::Vector4D, 43
- GetX
 - FAMath::Quaternion, 33
 - FAMath::Vector2D, 37
 - FAMath::Vector3D, 40
 - FAMath::Vector4D, 43
- GetY
 - FAMath::Quaternion, 33
 - FAMath::Vector2D, 37
 - FAMath::Vector3D, 40
 - FAMath::Vector4D, 43
- GetZ
 - FAMath::Quaternion, 34
 - FAMath::Vector3D, 40
 - FAMath::Vector4D, 44
- Inverse
 - FAMath, 13
- IsIdentity
 - FAMath, 14
- IsZeroQuaternion
 - FAMath, 14
- Length
 - FAMath, 14, 15
- Matrix4x4
 - FAMath::Matrix4x4, 28
- Norm
 - FAMath, 15
- Normalize
 - FAMath, 15
- operator*
 - FAMath, 15–18
- operator*=
 - FAMath::Matrix4x4, 30
 - FAMath::Quaternion, 34
 - FAMath::Vector2D, 37
 - FAMath::Vector3D, 40
 - FAMath::Vector4D, 44
- operator()
 - FAMath::Matrix4x4, 29
- operator+
 - FAMath, 18, 19
- operator+=
 - FAMath::Matrix4x4, 30
 - FAMath::Quaternion, 34
 - FAMath::Vector2D, 37
 - FAMath::Vector3D, 40
 - FAMath::Vector4D, 44
- operator-
 - FAMath, 19–21
- operator-=
 - FAMath::Matrix4x4, 30
 - FAMath::Quaternion, 34
 - FAMath::Vector2D, 37
 - FAMath::Vector3D, 41
 - FAMath::Vector4D, 44
- operator/
 - FAMath, 21, 22
- operator/=
 - FAMath::Vector2D, 38
 - FAMath::Vector3D, 41
 - FAMath::Vector4D, 44
- Orthonormalize
 - FAMath, 22
- PolarToCartesian
 - FAMath, 22
- Projection
 - FAMath, 22, 23
- Quaternion
 - FAMath::Quaternion, 32, 33
- QuaternionToRotationMatrixCol
 - FAMath, 23
- QuaternionToRotationMatrixRow
 - FAMath, 23
- Rotate
 - FAMath, 23
- RotationQuaternion
 - FAMath, 23, 24
- Scale

- FAMath, [24](#)
- SetCol
 - FAMath::Matrix4x4, [30](#)
- SetRow
 - FAMath::Matrix4x4, [31](#)
- SetScalar
 - FAMath::Quaternion, [34](#)
- SetToIdentity
 - FAMath, [24](#)
- SetVector
 - FAMath::Quaternion, [35](#)
- SetW
 - FAMath::Vector4D, [44](#)
- SetX
 - FAMath::Quaternion, [35](#)
 - FAMath::Vector2D, [38](#)
 - FAMath::Vector3D, [41](#)
 - FAMath::Vector4D, [45](#)
- SetY
 - FAMath::Quaternion, [35](#)
 - FAMath::Vector2D, [38](#)
 - FAMath::Vector3D, [41](#)
 - FAMath::Vector4D, [45](#)
- SetZ
 - FAMath::Quaternion, [35](#)
 - FAMath::Vector3D, [41](#)
 - FAMath::Vector4D, [45](#)
- SphericalToCartesian
 - FAMath, [24](#)
- Translate
 - FAMath, [25](#)
- Transpose
 - FAMath, [25](#)
- Vector2D
 - FAMath::Vector2D, [36](#)
- Vector3D
 - FAMath::Vector3D, [39](#)
- Vector4D
 - FAMath::Vector4D, [43](#)
- ZeroVector
 - FAMath, [25](#)