

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	10
4.1.2 Function Documentation	11
4.1.2.1 Adjoint()	11
4.1.2.2 CartesianToCylindrical()	11
4.1.2.3 CartesianToPolar()	11
4.1.2.4 CartesianToSpherical()	11
4.1.2.5 Cofactor()	12
4.1.2.6 CompareDoubles()	12
4.1.2.7 CompareFloats()	12
4.1.2.8 Conjugate()	12
4.1.2.9 CrossProduct()	12
4.1.2.10 CylindricalToCartesian()	13
4.1.2.11 Det()	13
4.1.2.12 DotProduct() [1/3]	13
4.1.2.13 DotProduct() [2/3]	13
4.1.2.14 DotProduct() [3/3]	13
4.1.2.15 Inverse() [1/2]	14
4.1.2.16 Inverse() [2/2]	14
4.1.2.17 IsIdentity() [1/2]	14
4.1.2.18 IsIdentity() [2/2]	14
4.1.2.19 IsZeroQuaternion()	14
4.1.2.20 Length() [1/4]	14
4.1.2.21 Length() [2/4]	15
4.1.2.22 Length() [3/4]	15
4.1.2.23 Length() [4/4]	15
4.1.2.24 Norm() [1/3]	15
4.1.2.25 Norm() [2/3]	15
4.1.2.26 Norm() [3/3]	15
4.1.2.27 Normalize()	16
4.1.2.28 operator*() [1/14]	16
4.1.2.29 operator*() [2/14]	16
4.1.2.30 operator*() [3/14]	16

4.1.2.31 operator*() [4/14]	16
4.1.2.32 operator*() [5/14]	17
4.1.2.33 operator*() [6/14]	17
4.1.2.34 operator*() [7/14]	17
4.1.2.35 operator*() [8/14]	17
4.1.2.36 operator*() [9/14]	17
4.1.2.37 operator*() [10/14]	18
4.1.2.38 operator*() [11/14]	18
4.1.2.39 operator*() [12/14]	18
4.1.2.40 operator*() [13/14]	18
4.1.2.41 operator*() [14/14]	18
4.1.2.42 operator+() [1/5]	19
4.1.2.43 operator+() [2/5]	19
4.1.2.44 operator+() [3/5]	19
4.1.2.45 operator+() [4/5]	19
4.1.2.46 operator+() [5/5]	19
4.1.2.47 operator-() [1/10]	20
4.1.2.48 operator-() [2/10]	20
4.1.2.49 operator-() [3/10]	20
4.1.2.50 operator-() [4/10]	20
4.1.2.51 operator-() [5/10]	20
4.1.2.52 operator-() [6/10]	21
4.1.2.53 operator-() [7/10]	21
4.1.2.54 operator-() [8/10]	21
4.1.2.55 operator-() [9/10]	21
4.1.2.56 operator-() [10/10]	21
4.1.2.57 operator/() [1/3]	22
4.1.2.58 operator/() [2/3]	22
4.1.2.59 operator/() [3/3]	22
4.1.2.60 Orthonormalize()	22
4.1.2.61 PolarToCartesian()	22
4.1.2.62 Projection() [1/3]	23
4.1.2.63 Projection() [2/3]	23
4.1.2.64 Projection() [3/3]	23
4.1.2.65 QuaternionToRotationMatrixCol()	23
4.1.2.66 QuaternionToRotationMatrixRow()	23
4.1.2.67 Rotate()	24
4.1.2.68 RotationQuaternion() [1/3]	24
4.1.2.69 RotationQuaternion() [2/3]	24
4.1.2.70 RotationQuaternion() [3/3]	24
4.1.2.71 Scale()	24
4.1.2.72 SetToIdentity()	25

4.1.2.73 SphericalToCartesian()	25
4.1.2.74 Translate()	25
4.1.2.75 Transpose()	25
4.1.2.76 ZeroVector() [1/3]	25
4.1.2.77 ZeroVector() [2/3]	25
4.1.2.78 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	28
5.1.3.1 Data() [1/2]	28
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]	29
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()	30
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() [1/3]	32
5.2.2.2 Quaternion() [2/3]	32
5.2.2.3 Quaternion() [3/3]	32
5.2.3 Member Function Documentation	32
5.2.3.1 GetScalar()	33
5.2.3.2 GetVector()	33
5.2.3.3 GetX()	33
5.2.3.4 GetY()	33
5.2.3.5 GetZ()	33
5.2.3.6 operator*=() [1/2]	33
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()	34
5.2.3.11 SetVector()	34
5.2.3.12 SetX()	34
5.2.3.13 SetY()	35
5.2.3.14 SetZ()	35
5.3 FAMath::Vector2D Class Reference	35
5.3.1 Detailed Description	36
5.3.2 Constructor & Destructor Documentation	36
5.3.2.1 Vector2D()	36
5.3.3 Member Function Documentation	36
5.3.3.1 GetX()	36
5.3.3.2 GetY()	36
5.3.3.3 operator*=()	36
5.3.3.4 operator+=()	37
5.3.3.5 operator-=()	37
5.3.3.6 operator/=()	37
5.3.3.7 SetX()	37
5.3.3.8 SetY()	37
5.4 FAMath::Vector3D Class Reference	38
5.4.1 Detailed Description	38
5.4.2 Constructor & Destructor Documentation	38
5.4.2.1 Vector3D()	38
5.4.3 Member Function Documentation	39
5.4.3.1 GetX()	39
5.4.3.2 GetY()	39
5.4.3.3 GetZ()	39
5.4.3.4 operator*=()	39
5.4.3.5 operator+=()	39
5.4.3.6 operator-=()	40
5.4.3.7 operator/=()	40
5.4.3.8 SetX()	40
5.4.3.9 SetY()	40
5.4.3.10 SetZ()	40
5.5 FAMath::Vector4D Class Reference	41
5.5.1 Detailed Description	41
5.5.2 Constructor & Destructor Documentation	41
5.5.2.1 Vector4D()	42
5.5.3 Member Function Documentation	42
5.5.3.1 GetW()	42
5.5.3.2 GetX()	42
5.5.3.3 GetY()	42

5.5.3.4 GetZ()	42
5.5.3.5 operator*=()	43
5.5.3.6 operator+=()	43
5.5.3.7 operator-=()	43
5.5.3.8 operator/=()	43
5.5.3.9 SetW()	43
5.5.3.10 SetX()	43
5.5.3.11 SetY()	44
5.5.3.12 SetZ()	44
6 File Documentation	45
6.1 FAMathEngine.h	45
Index	69

Chapter 1

Namespace Index

1.1 Namespace List

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

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

Chapter 2

Class Index

2.1 Class List

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

FAMath::Matrix4x4	
A matrix class used for 4x4 matrices and their manipulations	27
FAMath::Quaternion	
A quaternion class used for quaternions and their manipulations	31
FAMath::Vector2D	
A vector class used for 2D vectors/points and their manipulations	35
FAMath::Vector3D	
A vector class used for 3D vectors/points and their manipulations	38
FAMath::Vector4D	
A vector class used for 4D vectors/points and their manipulations	41

Chapter 3

File Index

3.1 File List

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

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

Chapter 4

Namespace Documentation

4.1 FAMath Namespace Reference

Has utility functions, [Vector2D](#), [Vector3D](#), [Vector4D](#), [Matrix4x4](#), and [Quaternion](#) classes.

Classes

- class [Matrix4x4](#)
A matrix class used for 4x4 matrices and their manipulations.
- class [Quaternion](#)
A quaternion class used for quaternions and their manipulations.
- 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)
Returns true if x and y are equal.
- bool [CompareDoubles](#) (double x, double y, double epsilon)
Returns true if x and y are equal.
- bool [ZeroVector](#) (const [Vector2D](#) &a)
Returns true if a is the zero vector.
- [Vector2D operator+](#) (const [Vector2D](#) &a, const [Vector2D](#) &b)
Adds a with b and returns the result.
- [Vector2D operator-](#) (const [Vector2D](#) &v)
Negates the vector v and returns the result.
- [Vector2D operator-](#) (const [Vector2D](#) &a, const [Vector2D](#) &b)
Subtracts b from a and returns the result.
- [Vector2D operator*](#) (const [Vector2D](#) &a, float k)
*Returns a * k.*

- [Vector2D operator*](#) (float k, const [Vector2D](#) &a)

*Returns $k * a$.*
- [Vector2D operator/](#) (const [Vector2D](#) &a, const float &k)

Returns a / k . If $k = 0$ the returned vector is the zero vector.
- float [DotProduct](#) (const [Vector2D](#) &a, const [Vector2D](#) &b)

Returns the dot product between a and b.
- 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) 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)

Adds a and b and returns the result.
- [Vector3D operator-](#) (const [Vector3D](#) &v)

Negates the vector v and returns the result.
- [Vector3D operator-](#) (const [Vector3D](#) &a, const [Vector3D](#) &b)

Subtracts b from a and returns the result.
- [Vector3D operator*](#) (const [Vector3D](#) &a, float k)

*Returns $a * k$.*
- [Vector3D operator*](#) (float k, const [Vector3D](#) &a)

*Returns $k * a$.*
- [Vector3D operator/](#) (const [Vector3D](#) &a, float k)

Returns a / k .
- float [DotProduct](#) (const [Vector3D](#) &a, const [Vector3D](#) &b)

Returns the dot product between a and b.
- [Vector3D CrossProduct](#) (const [Vector3D](#) &a, const [Vector3D](#) &b)

Returns the cross product between a and b.
- float [Length](#) (const [Vector3D](#) &v)

Returns the length(magnitude) of the 3D vector v.
- [Vector3D Norm](#) (const [Vector3D](#) &v)

Normalizes the 3D vector v.
- [Vector3D CylindricalToCartesian](#) (const [Vector3D](#) &v)

Converts the 3D vector v from cylindrical coordinates to cartesian coordinates.
- [Vector3D CartesianToCylindrical](#) (const [Vector3D](#) &v)

Converts the 3D vector v from cartesian coordinates to cylindrical coordinates.
- [Vector3D SphericalToCartesian](#) (const [Vector3D](#) &v)

Converts the 3D vector v from spherical coordinates to cartesian coordinates.
- [Vector3D CartesianToSpherical](#) (const [Vector3D](#) &v)

Converts the 3D vector v from cartesian coordinates to spherical coordinates.
- [Vector3D Projection](#) (const [Vector3D](#) &a, const [Vector3D](#) &b)

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

- void **Orthonormalize** (Vector3D &x, Vector3D &y, Vector3D &z)
Orthonormalizes the specified vectors.
- bool **ZeroVector** (const Vector4D &a)
Returns true if a is the zero vector.
- Vector4D **operator+** (const Vector4D &a, const Vector4D &b)
Adds a with b and returns the result.
- Vector4D **operator-** (const Vector4D &v)
Negatives v and returns the result.
- Vector4D **operator-** (const Vector4D &a, const Vector4D &b)
Subtracts b from a and returns the result.
- Vector4D **operator*** (const Vector4D &a, float k)
*Returns $a * k$.*
- Vector4D **operator*** (float k, const Vector4D &a)
*Returns $k * a$.*
- Vector4D **operator/** (const Vector4D &a, float k)
Returns a / k .
- float **DotProduct** (const Vector4D &a, const Vector4D &b)
Returns the dot product between a and b.
- float **Length** (const Vector4D &v)
Returns the length(magnitude) of the 4D vector v.
- Vector4D **Norm** (const Vector4D &v)
Normalizes the 4D vector v.
- Vector4D **Projection** (const Vector4D &a, const Vector4D &b)
Returns a 4D vector that is the projection of a onto b.
- Matrix4x4 **operator+** (const Matrix4x4 &m1, const Matrix4x4 &m2)
Adds m1 with m2 and returns the result.
- Matrix4x4 **operator-** (const Matrix4x4 &m)
Negates the 4x4 matrix m.
- Matrix4x4 **operator-** (const Matrix4x4 &m1, const Matrix4x4 &m2)
Subtracts m2 from m1 and returns the result.
- Matrix4x4 **operator*** (const Matrix4x4 &m, const float &k)
Multiplies m with k and returns the result.
- Matrix4x4 **operator*** (const float &k, const Matrix4x4 &m)
Multiplies k with m and returns the result.
- Matrix4x4 **operator*** (const Matrix4x4 &m1, const Matrix4x4 &m2)
Multiplies m1 with m2 and returns the result.
- Vector4D **operator*** (const Matrix4x4 &m, const Vector4D &v)
Multiplies m with v and returns the result.
- Vector4D **operator*** (const Vector4D &v, const Matrix4x4 &m)
Multiplies v with m and returns the result.
- void **SetToIdentity** (Matrix4x4 &m)
Sets m to the identity matrix.
- bool **IsIdentity** (const Matrix4x4 &m)
Returns true if m 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)
Constructs a 4x4 translation matrix with x, y, z and multiplies it by cm.
- Matrix4x4 **Scale** (const Matrix4x4 &cm, float x, float y, float z)
Construct a 4x4 scaling matrix with x, y, z and it by the cm.
- Matrix4x4 **Rotate** (const Matrix4x4 &cm, float angle, float x, float y, float z)

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

 - double [Det](#) (const [Matrix4x4](#) &m)
Returns the determinant m.
 - double [Cofactor](#) (const [Matrix4x4](#) &m, unsigned int row, unsigned int col)
Returns the cofactor of the row and col in m.
 - [Matrix4x4 Adjoint](#) (const [Matrix4x4](#) &m)
Returns the adjoint of m.
 - [Matrix4x4 Inverse](#) (const [Matrix4x4](#) &m)
Returns the inverse of m.
 - [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 q and returns the normalized quaternion.
 - [Quaternion Inverse](#) (const [Quaternion](#) &q)
Returns the invese of q.
 - [Quaternion RotationQuaternion](#) (float angle, float x, float y, float z)
Returns a rotation quaternion from the axis-angle rotation representation.
 - [Quaternion RotationQuaternion](#) (float angle, const [Vector3D](#) &axis)
Returns a quaternion from the axis-angle rotation representation.
 - [Quaternion RotationQuaternion](#) (const [Vector4D](#) &angAxis)
Returns a quaternion from the axis-angle rotation representation.
 - [Matrix4x4 QuaternionToRotationMatrixCol](#) (const [Quaternion](#) &q)
Transforms q into a column-major matrix.
 - [Matrix4x4 QuaternionToRotationMatrixRow](#) (const [Quaternion](#) &q)
Transforms q into a row-major matrix.

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 m .

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, theta(degrees), z).

4.1.2.3 CartesianToPolar()

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

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

The returned 2D vector = (r, theta(degrees)).

4.1.2.4 CartesianToSpherical()

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

Converts the 3D vector v from cartesian coordinates to spherical coordinates.

If v is the zero vector or if 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 *row* and *col* in *m*.

4.1.2.6 CompareDoubles()

```
bool FAMath::CompareDoubles (
    double x,
    double y,
    double epsilon ) [inline]
```

Returns true if *x* and *y* are equal.

Uses exact *epsilon* and adaptive *epsilon* to compare.

4.1.2.7 CompareFloats()

```
bool FAMath::CompareFloats (
    float x,
    float y,
    float epsilon ) [inline]
```

Returns true if *x* and *y* are equal.

Uses exact *epsilon* and adaptive *epsilon* to compare.

4.1.2.8 Conjugate()

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

Returns the conjugate of quaternion *q*.

4.1.2.9 CrossProduct()

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

Returns the cross product between *a* and *b*.

4.1.2.10 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).

4.1.2.11 Det()

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

Returns the determinant m .

4.1.2.12 DotProduct() [1/3]

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

Returns the dot product between a and b .

4.1.2.13 DotProduct() [2/3]

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

Returns the dot product between a and b .

4.1.2.14 DotProduct() [3/3]

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

Returns the dot product between a and b .

4.1.2.15 Inverse() [1/2]

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

Returns the inverse of m .

If m is noninvertible/singular, the identity matrix is returned.

4.1.2.16 Inverse() [2/2]

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

Returns the invese of q .

If q is the zero quaternion then q is returned.

4.1.2.17 IsIdentity() [1/2]

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

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

4.1.2.18 IsIdentity() [2/2]

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

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

4.1.2.19 IsZeroQuaternion()

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

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

4.1.2.20 Length() [1/4]

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

Returns the length of quaternion q .

4.1.2.21 Length() [2/4]

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

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

4.1.2.22 Length() [3/4]

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

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

4.1.2.23 Length() [4/4]

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

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

4.1.2.24 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.

4.1.2.25 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.

4.1.2.26 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.

4.1.2.27 Normalize()

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

Normalizes q and returns the normalized quaternion.

If q is the zero quaternion then q is returned.

4.1.2.28 operator*() [1/14]

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

Multiplies k with m and returns the result.

4.1.2.29 operator*() [2/14]

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

Multiplies m with k and returns the result.

4.1.2.30 operator*() [3/14]

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

Multiplies m with v and returns the result.

The vector v is a column vector.

4.1.2.31 operator*() [4/14]

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

Multiplies $m1$ with $m2$ and returns the result.

Does $m1 * m2$ in that order.

4.1.2.32 operator*() [5/14]

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

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

4.1.2.33 operator*() [6/14]

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

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

4.1.2.34 operator*() [7/14]

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

Returns $a * k$.

4.1.2.35 operator*() [8/14]

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

Returns $a * k$.

4.1.2.36 operator*() [9/14]

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

Returns $a * k$.

4.1.2.37 operator*() [10/14]

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

Multiplies v with m and returns the result.

The vector v is a row vector.

4.1.2.38 operator*() [11/14]

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

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

4.1.2.39 operator*() [12/14]

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

Returns $k * a$.

4.1.2.40 operator*() [13/14]

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

Returns $k * a$.

4.1.2.41 operator*() [14/14]

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

Returns $k * a$.

4.1.2.42 operator+() [1/5]

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

Adds *m1* with *m2* and returns the result.

4.1.2.43 operator+() [2/5]

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

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

4.1.2.44 operator+() [3/5]

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

Adds *a* with *b* and returns the result.

4.1.2.45 operator+() [4/5]

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

Adds *a* and *b* and returns the result.

4.1.2.46 operator+() [5/5]

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

Adds *a* with *b* and returns the result.

4.1.2.47 operator-() [1/10]

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

Negates the 4x4 matrix *m*.

4.1.2.48 operator-() [2/10]

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

Subtracts *m2* from *m1* and returns the result.

4.1.2.49 operator-() [3/10]

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

Returns a quaternion that has the result of $-q$.

4.1.2.50 operator-() [4/10]

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

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

4.1.2.51 operator-() [5/10]

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

Subtracts *b* from *a* and returns the result.

4.1.2.52 operator-() [6/10]

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

Negates the vector *v* and returns the result.

4.1.2.53 operator-() [7/10]

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

Subtracts *b* from *a* and returns the result.

4.1.2.54 operator-() [8/10]

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

Negates the vector *v* and returns the result.

4.1.2.55 operator-() [9/10]

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

Subtracts *b* from *a* and returns the result.

4.1.2.56 operator-() [10/10]

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

Negates *v* and returns the result.

4.1.2.57 operator/() [1/3]

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

Returns a / k . If $k = 0$ the returned vector is the zero vector.

4.1.2.58 operator/() [2/3]

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

Returns a / k .

If $k = 0$ the returned vector is the zero vector.

4.1.2.59 operator/() [3/3]

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

Returns a / k .

If $k = 0$ the returned vector is the zero vector.

4.1.2.60 Orthonormalize()

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

4.1.2.61 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)

4.1.2.62 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.

4.1.2.63 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.

4.1.2.64 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.

4.1.2.65 QuaternionToRotationMatrixCol()

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

Transforms q into a column-major matrix.

q should be a unit quaternion.

4.1.2.66 QuaternionToRotationMatrixRow()

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

Transforms q into a row-major matrix.

q should be a unit quaternion.

4.1.2.67 Rotate()

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

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

Returns *cm* * rotate.

4.1.2.68 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 *v* should be the angle(in degrees).

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

4.1.2.69 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.

4.1.2.70 RotationQuaternion() [3/3]

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

Returns a rotation quaternion from the axis-angle rotation representation.

The *angle* should be given in degrees.

4.1.2.71 Scale()

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

Construct a 4x4 scaling matrix with x, y, z and it by the *cm*.

Returns *cm* * scale.

4.1.2.72 SetToIdentity()

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

Sets m to the identity matrix.

4.1.2.73 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)

4.1.2.74 Translate()

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

Constructs a 4x4 translation matrix with x, y, z and multiplies it by cm .

Returns $cm * \text{translate}$.

4.1.2.75 Transpose()

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

Returns the tranpose of the given matrix m .

4.1.2.76 ZeroVector() [1/3]

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

Returns true if a is the zero vector.

4.1.2.77 ZeroVector() [2/3]

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

Returns true if a is the zero vector.

4.1.2.78 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](#) ()
Creates a new 4x4 identity matrix.
- [Matrix4x4](#) (float a[][4])
Creates a new 4x4 matrix with elements initialized to the given 2D array.
- [Matrix4x4](#) (const [Vector4D](#) &r1, const [Vector4D](#) &r2, const [Vector4D](#) &r3, const [Vector4D](#) &r4)
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).
- float & [operator](#)() (unsigned int row, unsigned int col)
Returns a reference to the element at the given (row, col).
- [Vector4D](#) [GetRow](#) (unsigned int row) const
Returns the specified row.
- [Vector4D](#) [GetCol](#) (unsigned int col) const
Returns the specified col.
- void [SetRow](#) (unsigned int row, [Vector4D](#) v)
Sets each element in the given row to the components of vector v.
- void [SetCol](#) (unsigned int col, [Vector4D](#) v)
Sets each element in the given col to the components of vector v.
- [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 m from this 4x4 matrix stores the result in this 4x4 matrix.
- [Matrix4x4](#) & [operator*=](#) (float k)
Multiplies this 4x4 matrix with 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]
```

Creates a new 4x4 identity matrix.

5.1.2.2 Matrix4x4() [2/3]

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

Creates a new 4x4 matrix with elements initialized to the given 2D array.

If *a* 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]
```

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 the 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 the 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 *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 *m* from this 4x4 matrix 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

A quaternion class used for quaternions and their manipulations.

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

Public Member Functions

- [Quaternion](#) (float scalar=1.0f, float x=0.0f, float y=0.0f, float z=0.0f)
Constructs a quaternion with the specified values.
- [Quaternion](#) (float scalar, const [Vector3D](#) &v)
Constructs a quaternion with the specified values.
- [Quaternion](#) (const [Vector4D](#) &v)
Constructs a quaternion with the given values in the 4D vector v.
- 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 /a q and stores the result in this quaternion.
- [Quaternion](#) & [operator-=](#) (const [Quaternion](#) &q)
Subtracts the quaternion q from this and stores the result in this quaternion.
- [Quaternion](#) & [operator*=](#) (float k)
Multiplies this quaternion by k and stores the result in this quaternion.
- [Quaternion](#) & [operator*=](#) (const [Quaternion](#) &q)
Multiplies this quaternion by q and stores the result in this quaternion.

5.2.1 Detailed Description

A quaternion class used for quaternions and their manipulations.

The datatype for the components is float.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Quaternion() [1/3]

```
FAMath::Quaternion::Quaternion (
    float scalar = 1.0f,
    float x = 0.0f,
    float y = 0.0f,
    float z = 0.0f ) [inline]
```

Constructs a quaternion with the specified values.

If no values are specified the identity quaternion is constructed.

5.2.2.2 Quaternion() [2/3]

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

Constructs a quaternion with the specified values.

5.2.2.3 Quaternion() [3/3]

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

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

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 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 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 /a q and stores the result in this quaternion.

5.2.3.9 operator-=()

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

Subtracts the quaternion q from this 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](#) (float x=0.0f, float y=0.0f)
Creates a new 2D vector/point with the components initialized to the arguments.
- float [GetX](#) () const
Returns the x component.
- float [GetY](#) () const
Returns the y component.
- void [SetX](#) (float x)
Sets the x component of the vector to the specified value.
- void [SetY](#) (float y)
Sets the y component to the specified value.
- [Vector2D](#) & [operator+=](#) (const [Vector2D](#) &b)
Adds this vector to vector b and stores the result in this vector.
- [Vector2D](#) & [operator-=](#) (const [Vector2D](#) &b)
Subtracts the vector b from this vector and stores the result in this vector.
- [Vector2D](#) & [operator*=](#) (float k)
Multiplies this vector by k and stores the result in this vector.
- [Vector2D](#) & [operator/=](#) (float k)
Divides this vector by k and stores the result in this vector.

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()

```
FAMath::Vector2D::Vector2D (
    float x = 0.0f,
    float y = 0.0f ) [inline]
```

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 the y component.

5.3.3.3 operator*=()

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

Multiplies this vector by k and stores the result in this vector.

5.3.3.4 operator+=()

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

Adds this vector to vector *b* and stores the result in this vector.

5.3.3.5 operator-=()

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

Subtracts the vector *b* from this vector and stores the result in this vector.

5.3.3.6 operator/=()

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

Divides this vector by *k* and stores the result in this vector.

If *k* is zero, the vector is unchanged.

5.3.3.7 SetX()

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

Sets the x component of the vector 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** (float x=0.0f, float y=0.0f, float z=0.0f)
Creates a new 3D vector/point with the components initialized to the arguments.
- 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)
Adds this vector to vector b and stores the result in this vector.
- **Vector3D** & **operator-=** (const **Vector3D** &b)
Subtracts b from this vector and stores the result in this vector.
- **Vector3D** & **operator*=** (float k)
Multiplies this vector by k and stores the result in this vector.
- **Vector3D** & **operator/=** (float k)
Divides this vector by k and stores the result in this vector.

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()

```
FAMath::Vector3D::Vector3D (
    float x = 0.0f,
    float y = 0.0f,
    float z = 0.0f ) [inline]
```

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]
```

Multiplies this vector by *k* and stores the result in this vector.

5.4.3.5 operator+=()

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

Adds this vector to vector *b* and stores the result in this vector.

5.4.3.6 operator-=()

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

Subtracts *b* from this vector and stores the result in this vector.

5.4.3.7 operator/=()

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

Divides this vector by *k* and stores the result in this vector.

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** (float x=0.0f, float y=0.0f, float z=0.0f, float w=0.0f)
Creates a new 4D vector/point with the components initialized to the arguments.
- 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)
Adds this vector to vector b and stores the result in this vector.
- **Vector4D** & **operator-=** (const **Vector4D** &b)
Subtracts the vector b from this vector and stores the result in this vector.
- **Vector4D** & **operator*=** (float k)
Multiplies this vector by k and stores the result in this vector.
- **Vector4D** & **operator/=** (float k)
Divides this vector by k and stores the result in this vector.

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()

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

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]
```

Multiplies this vector by k and stores the result in this vector.

5.5.3.6 operator+=()

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

Adds this vector to vector b and stores the result in this vector.

5.5.3.7 operator-=()

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

Subtracts the vector b from this vector and stores the result in this vector.

5.5.3.8 operator/=()

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

Divides this vector by k and stores the result in this vector.

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 3.14159f
12 #define PI2 6.28319f
13
14 namespace FAMath
15 {
16
17     inline bool CompareFloats(float x, float y, float epsilon)
18     {
19         float diff = fabs(x - y);
20         //exact epsilon
21         if (diff < epsilon)
22         {
23             return true;
24         }
25
26         //adapative epsilon
27         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
28     }
29
30     inline bool CompareDoubles(double x, double y, double epsilon)
31     {
32         double diff = fabs(x - y);
33         //exact epsilon
34         if (diff < epsilon)
35         {
36             return true;
37         }
38
39         //adapative epsilon
40         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
41     }
42
43     //-----
44
45     class Vector2D
46     {
47     public:
48         Vector2D(float x = 0.0f, float y = 0.0f);
49
50         float GetX() const;
51
52         float GetY() const;
53
54         void SetX(float x);
55
56         void SetY(float y);
57     };
58 }
```

```

86         Vector2D& operator+=(const Vector2D& b);
87
90         Vector2D& operator-=(const Vector2D& b);
91
94         Vector2D& operator*=(float k);
95
100        Vector2D& operator/=(float k);
101
102    private:
103        float mX;
104        float mY;
105    };
106
107
108    //-----
109    //Vector2D Constructors
110
111    inline Vector2D::Vector2D(float x, float y) : mX{ x }, mY{ y }
112    {}
113
114    //-----
115
116    //-----
117    //Vector2D Getters and Setters
118
119    inline float Vector2D::GetX() const
120    {
121        return mX;
122    }
123
124    inline float Vector2D::GetY() const
125    {
126        return mY;
127    }
128
129    inline void Vector2D::SetX(float x)
130    {
131        mX = x;
132    }
133
134    inline void Vector2D::SetY(float y)
135    {
136        mY = y;
137    }
138
139    //-----
140
141
142    //-----
143    //Vector2D Member functions
144
145    inline Vector2D& Vector2D::operator+=(const Vector2D& b)
146    {
147        this->mX += b.mX;
148        this->mY += b.mY;
149
150        return *this;
151    }
152
153    inline Vector2D& Vector2D::operator-=(const Vector2D& b)
154    {
155        this->mX -= b.mX;
156        this->mY -= b.mY;
157
158        return *this;
159    }
160
161    inline Vector2D& Vector2D::operator*=(float k)
162    {
163        this->mX *= k;
164        this->mY *= k;
165
166        return *this;
167    }
168
169    inline Vector2D& Vector2D::operator/=(float k)
170    {
171        if (CompareFloats(k, 0.0f, EPSILON))
172        {
173            return *this;
174        }
175
176        this->mX /= k;
177        this->mY /= k;
178
179        return *this;
180    }

```

```

181
182 //-----
183
184 //-----
185 //Vector2D Non-member functions
186
187 inline bool ZeroVector(const Vector2D& a)
188 {
189     if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON))
190     {
191         return true;
192     }
193     return false;
194 }
195
196 inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
197 {
198     return Vector2D(a.GetX() + b.GetX(), a.GetY() + b.GetY());
199 }
200
201 inline Vector2D operator-(const Vector2D& v)
202 {
203     return Vector2D(-v.GetX(), -v.GetY());
204 }
205
206 inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
207 {
208     return Vector2D(a.GetX() - b.GetX(), a.GetY() - b.GetY());
209 }
210
211 inline Vector2D operator*(const Vector2D& a, float k)
212 {
213     return Vector2D(a.GetX() * k, a.GetY() * k);
214 }
215
216 inline Vector2D operator*(float k, const Vector2D& a)
217 {
218     return Vector2D(k * a.GetX(), k * a.GetY());
219 }
220
221 inline Vector2D operator/(const Vector2D& a, const float& k)
222 {
223     if (CompareFloats(k, 0.0f, EPSILON))
224     {
225         return Vector2D();
226     }
227     return Vector2D(a.GetX() / k, a.GetY() / k);
228 }
229
230 inline float DotProduct(const Vector2D& a, const Vector2D& b)
231 {
232     return a.GetX() * b.GetX() + a.GetY() * b.GetY();
233 }
234
235 inline float Length(const Vector2D& v)
236 {
237     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY());
238 }
239
240 inline Vector2D Norm(const Vector2D& v)
241 {
242     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
243
244     //v is the zero vector
245     if (ZeroVector(v))
246     {
247         return v;
248     }
249
250     float mag{ Length(v) };
251
252     return Vector2D(v.GetX() / mag, v.GetY() / mag);
253 }
254
255 inline Vector2D PolarToCartesian(const Vector2D& v)
256 {
257     //v = (r, theta)
258     //x = rcos(theta)
259     //y = rsin(theta)
260     float angle{ v.GetY() * PI / 180.0f };
261
262     return Vector2D(v.GetX() * cos(angle), v.GetX() * sin(angle));
263 }
264
265 inline Vector2D CartesianToPolar(const Vector2D& v)

```

```

299     {
300         //v = (x, y)
301         //r = sqrt(vx^2 + vy^2)
302         //theta = arctan(y / x)
303
304         if (CompareFloats(v.GetX(), 0.0f, EPSILON))
305         {
306             return v;
307         }
308
309         float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
310         return Vector2D(Length(v), theta);
311     }
312
313 inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
314 {
315     //Projb(a) = (a dot b)b
316     //normalize b before projecting
317
318     Vector2D normB(Norm(b));
319     return Vector2D(DotProduct(a, normB) * normB);
320 }
321
322 #if defined(_DEBUG)
323 inline void print(const Vector2D& v)
324 {
325     std::cout << "(" << v.GetX() << ", " << v.GetY() << ")";
326 }
327 #endif
328 //-----
329
330 //-----
331
332 //-----
333
334 class Vector3D
335 {
336 public:
337     Vector3D(float x = 0.0f, float y = 0.0f, float z = 0.0f);
338
339     float GetX() const;
340
341     float GetY() const;
342
343     float GetZ() const;
344
345     void SetX(float x);
346
347     void SetY(float y);
348
349     void SetZ(float z);
350
351     Vector3D& operator+=(const Vector3D& b);
352
353     Vector3D& operator-=(const Vector3D& b);
354
355     Vector3D& operator*=(float k);
356
357     Vector3D& operator/=(float k);
358
359 private:
360     float mX;
361     float mY;
362     float mZ;
363 };
364
365 //-----
366 //Vector3D Constructors
367
368 inline Vector3D::Vector3D(float x, float y, float z) : mX{ x }, mY{ y }, mZ{ z }
369 {}
370
371 //-----
372 //-----
373 //Vector3D Getters and Setters
374
375 inline float Vector3D::GetX()const
376 {
377     return mX;
378 }
379
380
381

```



```

416     inline float Vector3D::GetY() const
417 {
418     return mY;
419 }
420
421     inline float Vector3D::GetZ() const
422 {
423     return mZ;
424 }
425
426     inline void Vector3D::SetX(float x)
427 {
428     mX = x;
429 }
430
431     inline void Vector3D::SetY(float y)
432 {
433     mY = y;
434 }
435
436     inline void Vector3D::SetZ(float z)
437 {
438     mZ = z;
439 }
440 //-----
441
442 //-----
443 //Vector3D Member functions
444
445     inline Vector3D& Vector3D::operator+=(const Vector3D& b)
446 {
447     this->mX += b.mX;
448     this->mY += b.mY;
449     this->mZ += b.mZ;
450
451     return *this;
452 }
453
454     inline Vector3D& Vector3D::operator-=(const Vector3D& b)
455 {
456     this->mX -= b.mX;
457     this->mY -= b.mY;
458     this->mZ -= b.mZ;
459
460     return *this;
461 }
462
463     inline Vector3D& Vector3D::operator*=(float k)
464 {
465     this->mX *= k;
466     this->mY *= k;
467     this->mZ *= k;
468
469     return *this;
470 }
471
472     inline Vector3D& Vector3D::operator/=(float k)
473 {
474     if (CompareFloats(k, 0.0f, EPSILON))
475     {
476         return *this;
477     }
478
479     this->mX /= k;
480     this->mY /= k;
481     this->mZ /= k;
482
483     return *this;
484 }
485
486 //-----
487
488 //-----
489 //Vector3D Non-member functions
490
491     inline bool ZeroVector(const Vector3D& a)
492 {
493     if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
494         CompareFloats(a.GetZ(), 0.0f, EPSILON))
495     {
496         return true;
497     }
498
499     return false;
500 }
501
502
503
504

```

```

507 inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
508 {
509     return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
510 }
511
514 inline Vector3D operator-(const Vector3D& v)
515 {
516     return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
517 }
518
521 inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
522 {
523     return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
524 }
525
528 inline Vector3D operator*(const Vector3D& a, float k)
529 {
530     return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
531 }
532
535 inline Vector3D operator*(float k, const Vector3D& a)
536 {
537     return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
538 }
539
544 inline Vector3D operator/(const Vector3D& a, float k)
545 {
546     if (CompareFloats(k, 0.0f, EPSILON))
547     {
548         return Vector3D();
549     }
550     return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
551 }
552
553 inline float DotProduct(const Vector3D& a, const Vector3D& b)
554 {
555     //a dot b = axbx + ayby + azbz
556     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
557 }
558
561 inline Vector3D CrossProduct(const Vector3D& a, const Vector3D& b)
562 {
563     //a x b = (aybz - azby, azbx - axbz, axby - aybx)
564     return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
565         a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
566         a.GetX() * b.GetY() - a.GetY() * b.GetX());
567 }
568
572 inline float Length(const Vector3D& v)
573 {
574     //length(v) = sqrt(vx^2 + vy^2 + vz^2)
575     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
576 }
577
581 inline Vector3D Norm(const Vector3D& v)
582 {
583     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
584     //v is the zero vector
585     if (ZeroVector(v))
586     {
587         return v;
588     }
589     float mag{ Length(v) };
590     return Vector3D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag);
591 }
592
605 inline Vector3D CylindricalToCartesian(const Vector3D& v)
606 {
607     //v = (r, theta, z)
608     //x = rcos(theta)
609     //y = rsin(theta)
610     //z = z
611     float angle{ v.GetY() * PI / 180.0f };
612     return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
613 }
614
622 inline Vector3D CartesianToCylindrical(const Vector3D& v)
623 {
624     //v = (x, y, z)
625     //r = sqrt(vx^2 + vy^2 + vz^2)
626     //theta = arctan(y / x)

```

```

627         //z = z
628         if (CompareFloats(v.GetX(), 0.0f, EPSILON))
629         {
630             return v;
631         }
632
633         float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
634         return Vector3D(Length(v), theta, v.GetZ());
635     }
636
637     inline Vector3D SphericalToCartesian(const Vector3D& v)
638     {
639         // v = (pho, phi, theta)
640         //x = pho * sin(phi) * cos(theta)
641         //y = pho * sin(phi) * sin(theta)
642         //z = pho * cos(theta);
643
644         float phi{ v.GetY() * PI / 180.0f };
645         float theta{ v.GetZ() * PI / 180.0f };
646
647         return Vector3D(v.GetX() * sin(phi) * cos(theta), v.GetX() * sin(phi) * sin(theta), v.GetX() *
        cos(theta));
648     }
649
650     inline Vector3D CartesianToSpherical(const Vector3D& v)
651     {
652         //v = (x, y, z)
653         //pho = sqrt(vx^2 + vy^2 + vz^2)
654         //phi = acos(z / pho)
655         //theta = arctan(y / x)
656
657         if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
658         {
659             return v;
660         }
661
662         float pho{ Length(v) };
663         float phi{ acos(v.GetZ() / pho) * 180.0f / PI };
664         float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
665
666         return Vector3D(pho, phi, theta);
667     }
668
669     inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
670     {
671         //Projb(a) = (a dot b)b
672         //normalize b before projecting
673
674         Vector3D normB(Norm(b));
675         return Vector3D(DotProduct(a, normB) * normB);
676     }
677
678     inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
679     {
680         x = Norm(x);
681         y = Norm(CrossProduct(z, x));
682         z = Norm(CrossProduct(x, y));
683     }
684
685 #if defined(_DEBUG)
686     inline void print(const Vector3D& v)
687     {
688         std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ")";
689     }
690 #endif
691
692 //-----
693
694 //-----
695
696 //-----
697
698 class Vector4D
699 {
700 public:
701     Vector4D(float x = 0.0f, float y = 0.0f, float z = 0.0f, float w = 0.0f);
702
703     float GetX() const;
704
705     float GetY() const;
706
707     float GetZ() const;
708
709     float GetW() const;

```

```

744
747     void SetX(float x);
748
751     void SetY(float y);
752
755     void SetZ(float z);
756
759     void SetW(float w);
760
763     Vector4D& operator+=(const Vector4D& b);
764
767     Vector4D& operator-=(const Vector4D& b);
768
771     Vector4D& operator*=(float k);
772
777     Vector4D& operator/=(float k);
778
779 private:
780     float mX;
781     float mY;
782     float mZ;
783     float mW;
784 };
785
786 //-----
787 //Vector4D Constructors
788
789 inline Vector4D::Vector4D(float x, float y, float z, float w) : mX{ x }, mY{ y }, mZ{ z }, mW{ w }
790 {}
791
792 //-----
793
794 //-----
795 //Vector4D Getters and Setters
796
797 inline float Vector4D::GetX() const
798 {
799     return mX;
800 }
801
802 inline float Vector4D::GetY() const
803 {
804     return mY;
805 }
806
807 inline float Vector4D::GetZ() const
808 {
809     return mZ;
810 }
811
812 inline float Vector4D::GetW() const
813 {
814     return mW;
815 }
816
817 inline void Vector4D::SetX(float x)
818 {
819     mX = x;
820 }
821
822 inline void Vector4D::SetY(float y)
823 {
824     mY = y;
825 }
826
827 inline void Vector4D::SetZ(float z)
828 {
829     mZ = z;
830 }
831
832 inline void Vector4D::SetW(float w)
833 {
834     mW = w;
835 }
836 //-----
837
838
839 //-----
840 //Vector4D Member functions
841
842 inline Vector4D& Vector4D::operator+=(const Vector4D& b)
843 {
844     this->mX += b.mX;
845     this->mY += b.mY;
846     this->mZ += b.mZ;
847     this->mW += b.mW;
848

```

```

849         return *this;
850     }
851
852     inline Vector4D& Vector4D::operator-=(const Vector4D& b)
853     {
854         this->mX -= b.mX;
855         this->mY -= b.mY;
856         this->mZ -= b.mZ;
857         this->mW -= b.mW;
858
859         return *this;
860     }
861
862     inline Vector4D& Vector4D::operator*=(float k)
863     {
864         this->mX *= k;
865         this->mY *= k;
866         this->mZ *= k;
867         this->mW *= k;
868
869         return *this;
870     }
871
872     inline Vector4D& Vector4D::operator/=(float k)
873     {
874         if (CompareFloats(k, 0.0f, EPSILON))
875         {
876             return *this;
877         }
878
879         this->mX /= k;
880         this->mY /= k;
881         this->mZ /= k;
882         this->mW /= k;
883
884         return *this;
885     }
886
887     //-----
888     //-----
889     //Vector4D Non-member functions
890
891     inline bool ZeroVector(const Vector4D& a)
892     {
893         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
894             CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
895         {
896             return true;
897         }
898
899         return false;
900     }
901
902     inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
903     {
904         return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
905             b.GetW());
906     }
907
908     inline Vector4D operator-(const Vector4D& v)
909     {
910         return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
911     }
912
913     inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
914     {
915         return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
916             b.GetW());
917     }
918
919     inline Vector4D operator*(const Vector4D& a, float k)
920     {
921         return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
922     }
923
924     inline Vector4D operator*(float k, const Vector4D& a)
925     {
926         return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
927     }
928
929     inline Vector4D operator/(const Vector4D& a, float k)
930     {
931         if (CompareFloats(k, 0.0f, EPSILON))
932         {
933             return Vector4D();
934         }
935     }

```

```

950
951     return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
952 }
953
954 inline float DotProduct(const Vector4D& a, const Vector4D& b)
955 {
956     //a dot b = axbx + ayby + azbz + awbw
957     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
958 }
959
960 inline float Length(const Vector4D& v)
961 {
962     //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
963     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
964 v.GetW());
965 }
966
967 inline Vector4D Norm(const Vector4D& v)
968 {
969     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
970     //v is the zero vector
971     if (ZeroVector(v))
972     {
973         return v;
974     }
975
976     float mag{ Length(v) };
977
978     return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
979 }
980
981 inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
982 {
983     //Projb(a) = (a dot b)b
984     //normalize b before projecting
985     Vector4D normB(Norm(b));
986     return Vector4D(DotProduct(a, normB) * normB);
987 }
988
989 #if defined(_DEBUG)
990 inline void print(const Vector4D& v)
991 {
992     std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ", " << v.GetW() << ")";
993 }
994 #endif
995
996 //-----
997
998 //-----
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020     class Matrix4x4
1021     {
1022     public:
1023
1024         Matrix4x4();
1025
1026         Matrix4x4(float a[][4]);
1027
1028         Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
1029
1030         float* Data();
1031
1032         const float* Data() const;
1033
1034         const float& operator()(unsigned int row, unsigned int col) const;
1035
1036         float& operator()(unsigned int row, unsigned int col);
1037
1038         Vector4D GetRow(unsigned int row) const;
1039
1040         Vector4D GetCol(unsigned int col) const;
1041
1042         void SetRow(unsigned int row, Vector4D v);
1043
1044         void SetCol(unsigned int col, Vector4D v);
1045
1046         Matrix4x4& operator+=(const Matrix4x4& m);
1047
1048         Matrix4x4& operator-=(const Matrix4x4& m);
1049
1050         Matrix4x4& operator*=(float k);
1051
1052         Matrix4x4& operator*=(const Matrix4x4& m);

```

```

1097
1098     private:
1099
1100         float mMat[4][4];
1101     };
1102
1103     //-----
1104     inline Matrix4x4::Matrix4x4()
1105     {
1106         //1st row
1107         mMat[0][0] = 1.0f;
1108         mMat[0][1] = 0.0f;
1109         mMat[0][2] = 0.0f;
1110         mMat[0][3] = 0.0f;
1111
1112         //2nd
1113         mMat[1][0] = 0.0f;
1114         mMat[1][1] = 1.0f;
1115         mMat[1][2] = 0.0f;
1116         mMat[1][3] = 0.0f;
1117
1118         //3rd row
1119         mMat[2][0] = 0.0f;
1120         mMat[2][1] = 0.0f;
1121         mMat[2][2] = 1.0f;
1122         mMat[2][3] = 0.0f;
1123
1124         //4th row
1125         mMat[3][0] = 0.0f;
1126         mMat[3][1] = 0.0f;
1127         mMat[3][2] = 0.0f;
1128         mMat[3][3] = 1.0f;
1129     }
1130
1131
1132
1133     inline Matrix4x4::Matrix4x4(float a[][4])
1134     {
1135         //1st row
1136         mMat[0][0] = a[0][0];
1137         mMat[0][1] = a[0][1];
1138         mMat[0][2] = a[0][2];
1139         mMat[0][3] = a[0][3];
1140
1141         //2nd
1142         mMat[1][0] = a[1][0];
1143         mMat[1][1] = a[1][1];
1144         mMat[1][2] = a[1][2];
1145         mMat[1][3] = a[1][3];
1146
1147         //3rd row
1148         mMat[2][0] = a[2][0];
1149         mMat[2][1] = a[2][1];
1150         mMat[2][2] = a[2][2];
1151         mMat[2][3] = a[2][3];
1152
1153         //4th row
1154         mMat[3][0] = a[3][0];
1155         mMat[3][1] = a[3][1];
1156         mMat[3][2] = a[3][2];
1157         mMat[3][3] = a[3][3];
1158     }
1159
1160     inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
1161     {
1162         SetRow(0, r1);
1163         SetRow(1, r2);
1164         SetRow(2, r3);
1165         SetRow(3, r4);
1166     }
1167
1168     inline float* Matrix4x4::Data()
1169     {
1170         return mMat[0];
1171     }
1172
1173     inline const float* Matrix4x4::Data()const
1174     {
1175         return mMat[0];
1176     }
1177
1178     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
1179     {
1180         if (row > 3 || col > 3)
1181         {
1182             return mMat[0][0];

```

```

1183     }
1184     else
1185     {
1186         return mMat[row][col];
1187     }
1188 }
1189
1190 inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
1191 {
1192     if (row > 3 || col > 3)
1193     {
1194         return mMat[0][0];
1195     }
1196     else
1197     {
1198         return mMat[row][col];
1199     }
1200 }
1201
1202 inline Vector4D Matrix4x4::GetRow(unsigned int row) const
1203 {
1204     if (row < 0 || row > 3)
1205         return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
1206     else
1207         return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);
1208 }
1209
1210 inline Vector4D Matrix4x4::GetCol(unsigned int col) const
1211 {
1212     if (col < 0 || col > 3)
1213         return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
1214     else
1215         return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
1216 }
1217
1218 inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
1219 {
1220     if (row > 3)
1221     {
1222         mMat[0][0] = v.GetX();
1223         mMat[0][1] = v.GetY();
1224         mMat[0][2] = v.GetZ();
1225         mMat[0][3] = v.GetW();
1226     }
1227     else
1228     {
1229         mMat[row][0] = v.GetX();
1230         mMat[row][1] = v.GetY();
1231         mMat[row][2] = v.GetZ();
1232         mMat[row][3] = v.GetW();
1233     }
1234 }
1235
1236 inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
1237 {
1238     if (col > 3)
1239     {
1240         mMat[0][0] = v.GetX();
1241         mMat[1][0] = v.GetY();
1242         mMat[2][0] = v.GetZ();
1243         mMat[3][0] = v.GetW();
1244     }
1245     else
1246     {
1247         mMat[0][col] = v.GetX();
1248         mMat[1][col] = v.GetY();
1249         mMat[2][col] = v.GetZ();
1250         mMat[3][col] = v.GetW();
1251     }
1252 }
1253
1254 inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
1255 {
1256     for (int i = 0; i < 4; ++i)
1257     {
1258         for (int j = 0; j < 4; ++j)
1259         {
1260             this->mMat[i][j] += m.mMat[i][j];
1261         }
1262     }
1263     return *this;
1264 }
1265
1266 inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
1267 {
1268     for (int i = 0; i < 4; ++i)
1269     {
1270         for (int j = 0; j < 4; ++j)
1271         {
1272             this->mMat[i][j] -= m.mMat[i][j];
1273         }
1274     }
1275     return *this;
1276 }

```



```

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

```

```

1361         return res;
1362     }
1363
1364     inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
1365     {
1366         Matrix4x4 res;
1367         for (int i = 0; i < 4; ++i)
1368         {
1369             for (int j = 0; j < 4; ++j)
1370             {
1371                 res(i, j) = m1(i, j) - m2(i, j);
1372             }
1373         }
1374         return res;
1375     }
1376
1377     inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
1378     {
1379         Matrix4x4 res;
1380         for (int i = 0; i < 4; ++i)
1381         {
1382             for (int j = 0; j < 4; ++j)
1383             {
1384                 res(i, j) = m(i, j) * k;
1385             }
1386         }
1387         return res;
1388     }
1389
1390     inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
1391     {
1392         Matrix4x4 res;
1393         for (int i = 0; i < 4; ++i)
1394         {
1395             for (int j = 0; j < 4; ++j)
1396             {
1397                 res(i, j) = k * m(i, j);
1398             }
1399         }
1400         return res;
1401     }
1402
1403     inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
1404     {
1405         Matrix4x4 res;
1406         for (int i = 0; i < 4; ++i)
1407         {
1408             res(i, 0) = (m1(i, 0) * m2(0, 0)) +
1409                 (m1(i, 1) * m2(1, 0)) +
1410                 (m1(i, 2) * m2(2, 0)) +
1411                 (m1(i, 3) * m2(3, 0));
1412
1413             res(i, 1) = (m1(i, 0) * m2(0, 1)) +
1414                 (m1(i, 1) * m2(1, 1)) +
1415                 (m1(i, 2) * m2(2, 1)) +
1416                 (m1(i, 3) * m2(3, 1));
1417
1418             res(i, 2) = (m1(i, 0) * m2(0, 2)) +
1419                 (m1(i, 1) * m2(1, 2)) +
1420                 (m1(i, 2) * m2(2, 2)) +
1421                 (m1(i, 3) * m2(3, 2));
1422
1423             res(i, 3) = (m1(i, 0) * m2(0, 3)) +
1424                 (m1(i, 1) * m2(1, 3)) +
1425                 (m1(i, 2) * m2(2, 3)) +
1426                 (m1(i, 3) * m2(3, 3));
1427         }
1428         return res;
1429     }
1430
1431     inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
1432     {
1433         Vector4D res;
1434
1435         res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());
1436
1437         res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());
1438
1439         res.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());
1440
1441         res.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());
1442     }
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461

```

```

1462     return res;
1463 }
1464
1469 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
1470 {
1471     Vector4D res;
1472
1473     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));
1474
1475     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));
1476
1477     res.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));
1478
1479     res.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));
1480
1481     return res;
1482 }
1483
1486 inline void SetToIdentity(Matrix4x4& m)
1487 {
1488     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1489
1490     //1st row
1491     m(0, 0) = 1.0f;
1492     m(0, 1) = 0.0f;
1493     m(0, 2) = 0.0f;
1494     m(0, 3) = 0.0f;
1495
1496     //2nd row
1497     m(1, 0) = 0.0f;
1498     m(1, 1) = 1.0f;
1499     m(1, 2) = 0.0f;
1500     m(1, 3) = 0.0f;
1501
1502     //3rd row
1503     m(2, 0) = 0.0f;
1504     m(2, 1) = 0.0f;
1505     m(2, 2) = 1.0f;
1506     m(2, 3) = 0.0f;
1507
1508     //4th row
1509     m(3, 0) = 0.0f;
1510     m(3, 1) = 0.0f;
1511     m(3, 2) = 0.0f;
1512     m(3, 3) = 1.0f;
1513 }
1514
1517 inline bool IsIdentity(const Matrix4x4& m)
1518 {
1519     //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f
1520
1521     for (int i = 0; i < 4; ++i)
1522     {
1523         for (int j = 0; j < 4; ++j)
1524         {
1525             if (i == j)
1526             {
1527                 if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1528                     return false;
1529             }
1530             else
1531             {
1532                 if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1533                     return false;
1534             }
1535         }
1536     }
1537 }
1538
1539 }
1540
1543 inline Matrix4x4 Transpose(const Matrix4x4& m)
1544 {
1545     //make the rows into cols
1546
1547     Matrix4x4 res;
1548
1549     //1st col = 1st row
1550     res(0, 0) = m(0, 0);
1551     res(1, 0) = m(0, 1);
1552     res(2, 0) = m(0, 2);
1553     res(3, 0) = m(0, 3);
1554
1555     //2nd col = 2nd row
1556     res(0, 1) = m(1, 0);
1557     res(1, 1) = m(1, 1);

```

```

1558         res(2, 1) = m(1, 2);
1559         res(3, 1) = m(1, 3);
1560
1561         //3rd col = 3rd row
1562         res(0, 2) = m(2, 0);
1563         res(1, 2) = m(2, 1);
1564         res(2, 2) = m(2, 2);
1565         res(3, 2) = m(2, 3);
1566
1567         //4th col = 4th row
1568         res(0, 3) = m(3, 0);
1569         res(1, 3) = m(3, 1);
1570         res(2, 3) = m(3, 2);
1571         res(3, 3) = m(3, 3);
1572
1573         return res;
1574     }
1575
1576     inline Matrix4x4 Translate(const Matrix4x4& cm, float x, float y, float z)
1577     {
1578         //1 0 0 0
1579         //0 1 0 0
1580         //0 0 1 0
1581         //x y z 1
1582
1583         Matrix4x4 t;
1584         t(3, 0) = x;
1585         t(3, 1) = y;
1586         t(3, 2) = z;
1587
1588         return cm * t;
1589     }
1590
1591     inline Matrix4x4 Scale(const Matrix4x4& cm, float x, float y, float z)
1592     {
1593         //x 0 0 0
1594         //0 y 0 0
1595         //0 0 z 0
1596         //0 0 0 1
1597
1598         Matrix4x4 s;
1599         s(0, 0) = x;
1600         s(1, 1) = y;
1601         s(2, 2) = z;
1602
1603         return cm * s;
1604     }
1605
1606     inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
1607     {
1608         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy      0
1609         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx      0
1610         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2      0
1611         //0                    0                    0                    1
1612         //c = cos(angle)
1613         //s = sin(angle)
1614
1615         float c = cos(angle * PI / 180.0f);
1616         float s = sin(angle * PI / 180.0f);
1617
1618         Matrix4x4 r;
1619
1620         //1st row
1621         r(0, 0) = c + (1.0f - c) * (x * x);
1622         r(0, 1) = (1.0f - c) * (x * y) + (s * z);
1623         r(0, 2) = (1.0f - c) * (x * z) - (s * y);
1624
1625         //2nd row
1626         r(1, 0) = (1.0f - c) * (x * y) - (s * z);
1627         r(1, 1) = c + (1.0f - c) * (y * y);
1628         r(1, 2) = (1.0f - c) * (y * z) + (s * x);
1629
1630         //3rd row
1631         r(2, 0) = (1.0f - c) * (x * z) + (s * y);
1632         r(2, 1) = (1.0f - c) * (y * z) - (s * x);
1633         r(2, 2) = c + (1.0f - c) * (z * z);
1634
1635         return cm * r;
1636     }
1637
1638     inline double Det(const Matrix4x4& m)
1639     {
1640         //m00m11(m22m33 - m23m32)
1641         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)
1642         * m(3, 2);
1643     }

```

```

1658         //m00m12(m23m31 - m21m33)
1659         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)
1660         * m(3, 3);
1661         //m00m13(m21m32 - m22m31)
1662         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)
1663         * m(3, 1);
1664         //m01m10(m22m33 - m23m32)
1665         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)
1666         * m(3, 2);
1667         //m01m12(m23m30 - m20m33)
1668         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)
1669         * m(3, 3);
1670         //m01m13(m20m32 - m22m30)
1671         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)
1672         * m(3, 0);
1673         //m02m10(m21m33 - m23m31)
1674         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)
1675         * m(3, 1);
1676         //m02m11(m23m30 - m20m33)
1677         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)
1678         * m(3, 3);
1679         //m02m13(m20m31 - m21m30)
1680         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)
1681         * m(3, 0);
1682         //m03m10(m21m32 - m22m21)
1683         double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
1684         2) * m(3, 1);
1685         //m03m11(m22m30 - m20m32)
1686         double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
1687         0) * m(3, 2);
1688         //m03m12(m20m31 - m21m30)
1689         double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
1690         1) * m(3, 0);
1691         return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
1692     }
1693
1694     inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
1695     {
1696         //cij = (-1)^(i + j) * det of minor(i, j);
1697         double tempMat[3][3]{};
1698         int tr{ 0 };
1699         int tc{ 0 };
1700
1701         //minor(i, j)
1702         for (int i = 0; i < 4; ++i)
1703         {
1704             if (i == row)
1705                 continue;
1706
1707             for (int j = 0; j < 4; ++j)
1708             {
1709                 if (j == col)
1710                     continue;
1711
1712                 tempMat[tr][tc] = m(i, j);
1713                 ++tc;
1714             }
1715             tc = 0;
1716             ++tr;
1717         }
1718
1719         //determinant of minor(i, j)
1720         double det3x3 = (tempMat[0][0] * tempMat[1][1] * tempMat[2][2]) + (tempMat[0][1] *
1721         tempMat[1][2] * tempMat[2][0]) +
1722         (tempMat[0][2] * tempMat[1][0] * tempMat[2][1]) - (tempMat[0][2] * tempMat[1][1] *
1723         tempMat[2][0]) -
1724         (tempMat[0][1] * tempMat[1][0] * tempMat[2][2]) - (tempMat[0][0] * tempMat[1][2] *
1725         tempMat[2][1]);
1726
1727         return pow(-1, row + col) * det3x3;
1728     }
1729
1730     inline Matrix4x4 Adjoint(const Matrix4x4& m)
1731     {
1732         //Cofactor of each ijth position put into matrix cA.
1733     }

```

```

1735         //Adjoint is the tranposed matrix of cA.
1736         Matrix4x4 cA;
1737         for (int i = 0; i < 4; ++i)
1738         {
1739             for (int j = 0; j < 4; ++j)
1740             {
1741                 cA(i, j) = static_cast<float>(Cofactor(m, i, j));
1742             }
1743         }
1744         return Transpose(cA);
1745     }
1746
1747     inline Matrix4x4 Inverse(const Matrix4x4& m)
1748     {
1749         //Inverse of m = adjoint of m / det of m
1750         double determinant = Det(m);
1751         if (CompareDoubles(determinant, 0.0, EPSILON))
1752             return Matrix4x4();
1753         return Adjoint(m) * (1.0f / static_cast<float>(determinant));
1754     }
1755
1756     #if defined(_DEBUG)
1757     inline void print(const Matrix4x4& m)
1758     {
1759         for (int i = 0; i < 4; ++i)
1760         {
1761             for (int j = 0; j < 4; ++j)
1762             {
1763                 std::cout << m(i, j) << " ";
1764             }
1765             std::cout << std::endl;
1766         }
1767     }
1768     #endif
1769
1770     //-----
1771
1772     //-----
1773
1774     class Quaternion
1775     {
1776     public:
1777         Quaternion(float scalar = 1.0f, float x = 0.0f, float y = 0.0f, float z = 0.0f);
1778         Quaternion(float scalar, const Vector3D& v);
1779         Quaternion(const Vector4D& v);
1780         float GetScalar() const;
1781         float GetX() const;
1782         float GetY() const;
1783         float GetZ() const;
1784         const Vector3D& GetVector() const;
1785         void SetScalar(float scalar);
1786         void SetX(float x);
1787         void SetY(float y);
1788         void SetZ(float z);
1789         void SetVector(const Vector3D& v);
1790         Quaternion& operator+=(const Quaternion& q);
1791         Quaternion& operator-=(const Quaternion& q);
1792         Quaternion& operator*=(float k);
1793         Quaternion& operator*=(const Quaternion& q);
1794     private:
1795         float mScalar;
1796         float mX;

```

```

1876         float mY;
1877         float mZ;
1878     };
1879
1880     //-----
1881     inline Quaternion::Quaternion() : mScalar{ 1.0f }, mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }
1882     {
1883     }
1884
1885     inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
1886         mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
1887     {
1888     }
1889
1890     inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
1891         mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
1892     {
1893     }
1894
1895     inline Quaternion::Quaternion(const Vector4D& v) :
1896         mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
1897     {
1898     }
1899
1900     inline float Quaternion::GetScalar() const
1901     {
1902         return mScalar;
1903     }
1904
1905     inline float Quaternion::GetX() const
1906     {
1907         return mX;
1908     }
1909
1910     inline float Quaternion::GetY() const
1911     {
1912         return mY;
1913     }
1914
1915     inline float Quaternion::GetZ() const
1916     {
1917         return mZ;
1918     }
1919
1920     inline const Vector3D& Quaternion::GetVector() const
1921     {
1922         return Vector3D(mX, mY, mZ);
1923     }
1924
1925     inline void Quaternion::SetScalar(float scalar)
1926     {
1927         mScalar = scalar;
1928     }
1929
1930     inline void Quaternion::SetX(float x)
1931     {
1932         mX = x;
1933     }
1934
1935     inline void Quaternion::SetY(float y)
1936     {
1937         mY = y;
1938     }
1939
1940     inline void Quaternion::SetZ(float z)
1941     {
1942         mZ = z;
1943     }
1944
1945     inline void Quaternion::SetVector(const Vector3D& v)
1946     {
1947         mX = v.GetX();
1948         mY = v.GetY();
1949         mZ = v.GetZ();
1950     }
1951
1952     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
1953     {
1954         this->mScalar += q.mScalar;
1955         this->mX += q.mX;
1956         this->mY += q.mY;
1957         this->mZ += q.mZ;
1958
1959         return *this;
1960     }
1961
1962     inline Quaternion& Quaternion::operator-=(const Quaternion& q)

```

```

1963     {
1964         this->mScalar -= q.mScalar;
1965         this->mX -= q.mX;
1966         this->mY -= q.mY;
1967         this->mZ -= q.mZ;
1968
1969         return *this;
1970     }
1971
1972     inline Quaternion& Quaternion::operator*=(float k)
1973     {
1974         this->mScalar *= k;
1975         this->mX *= k;
1976         this->mY *= k;
1977         this->mZ *= k;
1978
1979         return *this;
1980     }
1981
1982     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
1983     {
1984         Vector3D thisVector(this->mX, this->mY, this->mZ);
1985         Vector3D qVector(q.mX, q.mY, q.mZ);
1986
1987         float s{ this->mScalar * q.mScalar };
1988         float dP{ DotProduct(thisVector, qVector) };
1989         float resultScalar{ s - dP };
1990
1991         Vector3D a(this->mScalar * qVector);
1992         Vector3D b(q.mScalar * thisVector);
1993         Vector3D cP(CrossProduct(thisVector, qVector));
1994         Vector3D resultVector(a + b + cP);
1995
1996         this->mScalar = resultScalar;
1997         this->mX = resultVector.GetX();
1998         this->mY = resultVector.GetY();
1999         this->mZ = resultVector.GetZ();
2000
2001         return *this;
2002     }
2003
2004     inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
2005     {
2006         return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
2007             q2.GetY(), q1.GetZ() + q2.GetZ());
2008     }
2009
2010     inline Quaternion operator-(const Quaternion& q)
2011     {
2012         return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2013     }
2014
2015     inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
2016     {
2017         return Quaternion(q1.GetScalar() - q2.GetScalar(),
2018             q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
2019     }
2020
2021     inline Quaternion operator*(float k, const Quaternion& q)
2022     {
2023         return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
2024     }
2025
2026     inline Quaternion operator*(const Quaternion& q, float k)
2027     {
2028         return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
2029     }
2030
2031     inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
2032     {
2033         //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
2034         //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
2035
2036         Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
2037         Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());
2038
2039         float s{ q1.GetScalar() * q2.GetScalar() };
2040         float dP{ DotProduct(q1Vector, q2Vector) };
2041         float resultScalar{ s - dP };
2042
2043         Vector3D a(q1.GetScalar() * q2Vector);
2044         Vector3D b(q2.GetScalar() * q1Vector);
2045         Vector3D cP(CrossProduct(q1Vector, q2Vector));
2046         Vector3D resultVector(a + b + cP);
2047
2048         return Quaternion(resultScalar, resultVector);
2049     }
2050
2051     }
2052
2053
2054
2055
2056
2057
2058
2059
2060

```



```

2061
2062 inline bool IsZeroQuaternion(const Quaternion& q)
2063 {
2064     //zero quaternion = (0, 0, 0, 0)
2065     return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2066         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2067 }
2068
2069 inline bool IsIdentity(const Quaternion& q)
2070 {
2071     //identity quaternion = (1, 0, 0, 0)
2072     return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2073         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2074 }
2075
2076 inline Quaternion Conjugate(const Quaternion& q)
2077 {
2078     //conjugate of a quaternion is the quaternion with its vector part negated
2079     return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2080 }
2081
2082 inline float Length(const Quaternion& q)
2083 {
2084     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
2085     return sqrt(q.GetScalar() * q.GetScalar() + q.GetX() * q.GetX() + q.GetY() * q.GetY() +
2086         q.GetZ() * q.GetZ());
2087 }
2088
2089 inline Quaternion Normalize(const Quaternion& q)
2090 {
2091     //to normalize a quaternion you do q / |q|
2092     if (IsZeroQuaternion(q))
2093         return q;
2094
2095     float d{ Length(q) };
2096
2097     return Quaternion(q.GetScalar() / d, q.GetX() / d, q.GetY() / d, q.GetZ() / d);
2098 }
2099
2100 inline Quaternion Inverse(const Quaternion& q)
2101 {
2102     //inverse = conjugate of q / |q|^2
2103     if (IsZeroQuaternion(q))
2104         return q;
2105
2106     Quaternion conjugateOfQ(Conjugate(q));
2107
2108     float d{ Length(q) };
2109     d *= d;
2110
2111     return Quaternion(conjugateOfQ.GetScalar() / d, conjugateOfQ.GetX() / d,
2112         conjugateOfQ.GetY() / d, conjugateOfQ.GetZ() / d);
2113 }
2114
2115 inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
2116 {
2117     //A roatation quaternion is a quaternion where the
2118     //scalar part = cos(theta / 2)
2119     //vector part = sin(theta / 2) * axis
2120     //the axis needs to be normalized
2121
2122     float ang{ angle / 2.0f };
2123     float c{ cos(ang * PI / 180.0f) };
2124     float s{ sin(ang * PI / 180.0f) };
2125
2126     Vector3D axis(x, y, z);
2127     axis = Norm(axis);
2128
2129     return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2130 }
2131
2132 inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
2133 {
2134     //A roatation quaternion is a quaternion where the
2135     //scalar part = cos(theta / 2)
2136     //vector part = sin(theta / 2) * axis
2137     //the axis needs to be normalized
2138
2139     float ang{ angle / 2.0f };
2140     float c{ cos(ang * PI / 180.0f) };
2141     float s{ sin(ang * PI / 180.0f) };
2142
2143     Vector3D axisN(Norm(axis));
2144
2145     return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
2146 }

```

```

2171     }
2172
2173     inline Quaternion RotationQuaternion(const Vector4D& angAxis)
2174     {
2175         //A roatation quaternion is a quaternion where the
2176         //scalar part = cos(theta / 2)
2177         //vector part = sin(theta / 2) * axis
2178         //the axis needs to be normalized
2179
2180         float angle{ angAxis.GetX() / 2.0f };
2181         float c{ cos(angle * PI / 180.0f) };
2182         float s{ sin(angle * PI / 180.0f) };
2183
2184         Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
2185         axis = Norm(axis);
2186
2187         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2188     }
2189
2190     inline Matrix4x4 QuaternionToRotationMatrixCol(const Quaternion& q)
2191     {
2192         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
2193         //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
2194         //2q2q4 - 2q1q3          2q3q4 + 2q1q2          1 - 2q2^2 - 2q3^2      0
2195         //0                      0                      0                      1
2196         //q1 = scalar
2197         //q2 = x
2198         //q3 = y
2199         //q4 = z
2200
2201         float colMat[4][4] = {};
2202
2203         colMat[0][0] = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
2204         colMat[0][1] = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
2205         colMat[0][2] = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
2206         colMat[0][3] = 0.0f;
2207
2208         colMat[1][0] = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
2209         colMat[1][1] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
2210         colMat[1][2] = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
2211         colMat[1][3] = 0.0f;
2212
2213         colMat[2][0] = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
2214         colMat[2][1] = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
2215         colMat[2][2] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
2216         colMat[2][3] = 0.0f;
2217
2218         colMat[3][0] = 0.0f;
2219         colMat[3][1] = 0.0f;
2220         colMat[3][2] = 0.0f;
2221         colMat[3][3] = 1.0f;
2222
2223         return Matrix4x4(colMat);
2224     }
2225
2226     inline Matrix4x4 QuaternionToRotationMatrixRow(const Quaternion& q)
2227     {
2228         //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
2229         //2q2q3 - 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
2230         //2q2q4 + 2q1q3          2q3q4 - 2q1q2          1 - 2q2^2 - 2q3^2      0
2231         //0                      0                      0                      1
2232         //q1 = scalar
2233         //q2 = x
2234         //q3 = y
2235         //q4 = z
2236
2237         float rowMat[4][4] = {};
2238
2239         rowMat[0][0] = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
2240         rowMat[0][1] = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
2241         rowMat[0][2] = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
2242         rowMat[0][3] = 0.0f;
2243
2244         rowMat[1][0] = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
2245         rowMat[1][1] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
2246         rowMat[1][2] = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
2247         rowMat[1][3] = 0.0f;
2248
2249         rowMat[2][0] = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
2250         rowMat[2][1] = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
2251         rowMat[2][2] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
2252         rowMat[2][3] = 0.0f;
2253
2254         rowMat[3][0] = 0.0f;
2255         rowMat[3][1] = 0.0f;
2256         rowMat[3][2] = 0.0f;
2257         rowMat[3][3] = 1.0f;
2258     }

```

```
2271
2272         return Matrix4x4(rowMat);
2273     }
2274
2275     #if defined(_DEBUG)
2276     inline void print(const Quaternion& q)
2277     {
2278         std::cout << "(" << q.GetScalar() << ", " << q.GetX() << ", " << q.GetY() << ", " << q.GetZ();
2279     }
2280     #endif
2281     //-----
2282
2283     //-----
2284 }
```


Index

- Adjoint
 - FAMath, [11](#)
- CartesianToCylindrical
 - FAMath, [11](#)
- CartesianToPolar
 - FAMath, [11](#)
- CartesianToSpherical
 - FAMath, [11](#)
- Cofactor
 - FAMath, [11](#)
- CompareDoubles
 - FAMath, [12](#)
- CompareFloats
 - FAMath, [12](#)
- Conjugate
 - FAMath, [12](#)
- CrossProduct
 - FAMath, [12](#)
- CylindricalToCartesian
 - FAMath, [12](#)
- Data
 - FAMath::Matrix4x4, [28](#)
- Det
 - FAMath, [13](#)
- DotProduct
 - FAMath, [13](#)
- FAMath, [7](#)
 - Adjoint, [11](#)
 - CartesianToCylindrical, [11](#)
 - CartesianToPolar, [11](#)
 - CartesianToSpherical, [11](#)
 - Cofactor, [11](#)
 - CompareDoubles, [12](#)
 - CompareFloats, [12](#)
 - Conjugate, [12](#)
 - CrossProduct, [12](#)
 - CylindricalToCartesian, [12](#)
 - Det, [13](#)
 - DotProduct, [13](#)
 - Inverse, [13](#), [14](#)
 - IsIdentity, [14](#)
 - IsZeroQuaternion, [14](#)
 - Length, [14](#), [15](#)
 - Norm, [15](#)
 - Normalize, [15](#)
 - operator*, [16–18](#)
 - operator+, [18](#), [19](#)
 - operator-, [19–21](#)
 - operator/, [21](#), [22](#)
 - Orthonormalize, [22](#)
 - PolarToCartesian, [22](#)
 - Projection, [22](#), [23](#)
 - QuaternionToRotationMatrixCol, [23](#)
 - QuaternionToRotationMatrixRow, [23](#)
 - Rotate, [23](#)
 - RotationQuaternion, [24](#)
 - Scale, [24](#)
 - SetToIdentity, [24](#)
 - SphericalToCartesian, [25](#)
 - Translate, [25](#)
 - Transpose, [25](#)
 - ZeroVector, [25](#)
- FAMath::Matrix4x4, [27](#)
 - Data, [28](#)
 - GetCol, [29](#)
 - GetRow, [29](#)
 - Matrix4x4, [28](#)
 - operator*=[29](#), [30](#)
 - operator(), [29](#)
 - operator+=[30](#)
 - operator-=[30](#)
 - SetCol, [30](#)
 - SetRow, [30](#)
- FAMath::Quaternion, [31](#)
 - GetScalar, [32](#)
 - GetVector, [33](#)
 - GetX, [33](#)
 - GetY, [33](#)
 - GetZ, [33](#)
 - operator*=[33](#)
 - operator+=[34](#)
 - operator-=[34](#)
 - Quaternion, [32](#)
 - SetScalar, [34](#)
 - SetVector, [34](#)
 - SetX, [34](#)
 - SetY, [34](#)
 - SetZ, [35](#)
- FAMath::Vector2D, [35](#)
 - GetX, [36](#)
 - GetY, [36](#)
 - operator*=[36](#)
 - operator+=[36](#)
 - operator-=[37](#)
 - operator/=[37](#)
 - SetX, [37](#)

- SetY, [37](#)
- Vector2D, [36](#)
- FAMath::Vector3D, [38](#)
 - GetX, [39](#)
 - GetY, [39](#)
 - GetZ, [39](#)
 - operator*=, [39](#)
 - operator+=, [39](#)
 - operator-=, [39](#)
 - operator/=: [40](#)
 - SetX, [40](#)
 - SetY, [40](#)
 - SetZ, [40](#)
 - Vector3D, [38](#)
- FAMath::Vector4D, [41](#)
 - GetW, [42](#)
 - GetX, [42](#)
 - GetY, [42](#)
 - GetZ, [42](#)
 - operator*=, [42](#)
 - operator+=, [43](#)
 - operator-=, [43](#)
 - operator/=: [43](#)
 - SetW, [43](#)
 - SetX, [43](#)
 - SetY, [43](#)
 - SetZ, [44](#)
 - Vector4D, [41](#)
- GetCol
 - FAMath::Matrix4x4, [29](#)
- GetRow
 - FAMath::Matrix4x4, [29](#)
- GetScalar
 - FAMath::Quaternion, [32](#)
- GetVector
 - FAMath::Quaternion, [33](#)
- GetW
 - FAMath::Vector4D, [42](#)
- GetX
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [42](#)
- GetY
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [42](#)
- GetZ
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [42](#)
- Inverse
 - FAMath, [13](#), [14](#)
- IsIdentity
 - FAMath, [14](#)
- IsZeroQuaternion
 - FAMath, [14](#)
- Length
 - FAMath, [14](#), [15](#)
- Matrix4x4
 - FAMath::Matrix4x4, [28](#)
- Norm
 - FAMath, [15](#)
- Normalize
 - FAMath, [15](#)
- operator*
 - FAMath, [16–18](#)
- operator*=
 - FAMath::Matrix4x4, [29](#), [30](#)
 - FAMath::Quaternion, [33](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [42](#)
- operator()
 - FAMath::Matrix4x4, [29](#)
- operator+
 - FAMath, [18](#), [19](#)
- operator+=
 - FAMath::Matrix4x4, [30](#)
 - FAMath::Quaternion, [34](#)
 - FAMath::Vector2D, [36](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [43](#)
- operator-
 - FAMath, [19–21](#)
- operator-=
 - FAMath::Matrix4x4, [30](#)
 - FAMath::Quaternion, [34](#)
 - FAMath::Vector2D, [37](#)
 - FAMath::Vector3D, [39](#)
 - FAMath::Vector4D, [43](#)
- operator/
 - FAMath, [21](#), [22](#)
- operator/=
 - FAMath::Vector2D, [37](#)
 - FAMath::Vector3D, [40](#)
 - FAMath::Vector4D, [43](#)
- Orthonormalize
 - FAMath, [22](#)
- PolarToCartesian
 - FAMath, [22](#)
- Projection
 - FAMath, [22](#), [23](#)
- Quaternion
 - FAMath::Quaternion, [32](#)
- QuaternionToRotationMatrixCol
 - FAMath, [23](#)
- QuaternionToRotationMatrixRow
 - FAMath, [23](#)

Rotate
 FAMath, [23](#)

RotationQuaternion
 FAMath, [24](#)

Scale
 FAMath, [24](#)

SetCol
 FAMath::Matrix4x4, [30](#)

SetRow
 FAMath::Matrix4x4, [30](#)

SetScalar
 FAMath::Quaternion, [34](#)

SetTolIdentity
 FAMath, [24](#)

SetVector
 FAMath::Quaternion, [34](#)

SetW
 FAMath::Vector4D, [43](#)

SetX
 FAMath::Quaternion, [34](#)
 FAMath::Vector2D, [37](#)
 FAMath::Vector3D, [40](#)
 FAMath::Vector4D, [43](#)

SetY
 FAMath::Quaternion, [34](#)
 FAMath::Vector2D, [37](#)
 FAMath::Vector3D, [40](#)
 FAMath::Vector4D, [43](#)

SetZ
 FAMath::Quaternion, [35](#)
 FAMath::Vector3D, [40](#)
 FAMath::Vector4D, [44](#)

SphericalToCartesian
 FAMath, [25](#)

Translate
 FAMath, [25](#)

Transpose
 FAMath, [25](#)

Vector2D
 FAMath::Vector2D, [36](#)

Vector3D
 FAMath::Vector3D, [38](#)

Vector4D
 FAMath::Vector4D, [41](#)

ZeroVector
 FAMath, [25](#)