Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4

Chapter 1

Namespace Index

1.1 Namespace List

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Namespace Documentation

4.1 FAMath Namespace Reference

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

Classes

- class Matrix4x4

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

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

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

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

Functions

- bool **CompareFloats** (float x, float y, float epsilon)
- bool **CompareDoubles** (double x, double y, double epsilon)
- bool ZeroVector (const Vector2D &a)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  Returns a 3D vector that is the projection of a onto b. If b is the zero vector a is returned.
- void Orthonormalize (Vector3D &x, Vector3D &y, Vector3D &z)

  Orthonormalizes the specified vectors. Uses Classical Gram-Schmidt.
- bool ZeroVector (const Vector4D &a)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Matrix4x4 Transpose (const Matrix4x4 &m)

  Returns the tranpose of the given matrix m.
- Matrix4x4 Translate (const Matrix4x4 &cm, float x, float y, float z)

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

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

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

  .
- double Det (const Matrix4x4 &m)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Matrix4x4 QuaternionToRotationMatrixRow (const Quaternion &q)

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

4.1.1 Detailed Description

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

4.1.2 Function Documentation

4.1.2.1 Adjoint()

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

Returns the adjoint of the given matrix.

4.1.2.2 CartesianToCylindrical()

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

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

4.1.2.3 CartesianToPolar()

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

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

4.1.2.4 CartesianToSpherical()

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

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

4.1.2.5 Cofactor()

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

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

4.1.2.6 Conjugate()

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

Returns the conjugate of quaternion q.

4.1.2.7 CrossProduct()

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

Returns the cross product between two 3D vectors.

4.1.2.8 CylindricalToCartesian()

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

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

4.1.2.9 Det()

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

Returns the determinant of the given matrix.

4.1.2.10 DotProduct() [1/3]

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

Returns the dot product between two 2D vectors.

4.1.2.11 DotProduct() [2/3]

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

Returns the dot product between two 3D vectors.

4.1.2.12 DotProduct() [3/3]

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

Returns the dot product between two 4D vectors.

4.1.2.13 Inverse() [1/2]

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

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

4.1.2.14 Inverse() [2/2]

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

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

4.1.2.15 IsIdentity() [1/2]

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

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

4.1.2.16 IsIdentity() [2/2]

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

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

4.1.2.17 IsZeroQuaternion()

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

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

4.1.2.18 Length() [1/4]

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

Returns the length of quaternion q.

4.1.2.19 Length() [2/4]

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

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

4.1.2.20 Length() [3/4]

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

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

4.1.2.21 Length() [4/4]

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

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

4.1.2.22 Norm() [1/3]

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

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

4.1.2.23 Norm() [2/3]

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

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

4.1.2.24 Norm() [3/3]

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

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

4.1.2.25 Normalize()

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

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

4.1.2.26 operator*() [1/14]

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

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

4.1.2.27 operator*() [2/14]

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

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

4.1.2.28 operator*() [3/14]

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

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

4.1.2.29 operator*() [4/14]

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

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

4.1.2.30 operator*() [5/14]

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

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

4.1.2.31 operator*() [6/14]

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

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

4.1.2.32 operator*() [7/14]

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

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

4.1.2.33 operator*() [8/14]

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

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

4.1.2.34 operator*() [9/14]

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

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

4.1.2.35 operator*() [10/14]

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

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

4.1.2.36 operator*() [11/14]

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

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

4.1.2.37 operator*() [12/14]

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

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

4.1.2.38 operator*() [13/14]

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

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

4.1.2.39 operator*() [14/14]

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

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

4.1.2.40 operator+() [1/5]

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

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

4.1.2.41 operator+() [2/5]

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

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

4.1.2.42 operator+() [3/5]

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

2D vector addition.

4.1.2.43 operator+() [4/5]

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

3D vector addition.

4.1.2.44 operator+() [5/5]

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

4D vector addition.

4.1.2.45 operator-() [1/10]

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

Negates the 4x4 matrix m.

4.1.2.46 operator-() [2/10]

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

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

4.1.2.47 operator-() [3/10]

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

Returns a quaternion that has the result of -q.

4.1.2.48 operator-() [4/10]

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

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

4.1.2.49 operator-() [5/10]

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

2D vector subtraction.

4.1.2.50 operator-() [6/10]

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

2D vector negation.

4.1.2.51 operator-() [7/10]

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

3D vector subtraction.

4.1.2.52 operator-() [8/10]

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

3D vector negeation.

4.1.2.53 operator-() [9/10]

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

4D vector subtraction.

4.1.2.54 operator-() [10/10]

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

4D vector negation.

4.1.2.55 operator/() [1/3]

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

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

4.1.2.56 operator/() [2/3]

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

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

4.1.2.57 operator/() [3/3]

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

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

4.1.2.58 Orthonormalize()

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

Orthonormalizes the specified vectors. Uses Classical Gram-Schmidt.

4.1.2.59 PolarToCartesian()

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

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

4.1.2.60 Projection() [1/3]

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

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

4.1.2.61 Projection() [2/3]

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

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

4.1.2.62 Projection() [3/3]

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

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

4.1.2.63 QuaternionToRotationMatrixCol()

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

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

4.1.2.64 QuaternionToRotationMatrixRow()

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

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

4.1.2.65 Rotate()

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

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

.

4.1.2.66 RotationQuaternion() [1/3]

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

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

4.1.2.67 RotationQuaternion() [2/3]

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

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

4.1.2.68 RotationQuaternion() [3/3]

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

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

4.1.2.69 Scale()

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

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

4.1.2.70 SetToIdentity()

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

Sets the given matrix to the identity matrix.

4.1.2.71 SphericalToCartesian()

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

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

4.1.2.72 Translate()

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

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

4.1.2.73 Transpose()

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

Returns the tranpose of the given matrix m.

4.1.2.74 ZeroVector() [1/3]

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

Returns true if a is the zero vector.

4.1.2.75 ZeroVector() [2/3]

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

Returns true if a is the zero vector.

4.1.2.76 ZeroVector() [3/3]

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

Returns true if a is the zero vector.

Chapter 5

Class Documentation

5.1 FAMath::Matrix4x4 Class Reference

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

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

Public Member Functions

- Matrix4x4 ()

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

    Overloaded Constructor.
- Matrix4x4 (const Vector4D &r1, const Vector4D &r2, const Vector4D &r3, const Vector4D &r4)

    Overloaded Constructor. Creates a new 4x4 matrix with each row being set to the specified rows.
- float ∗ Data ()

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

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

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

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

    Returns specified row. Row should be between [0,3]. If it is out of range the first row will be returned.
- Vector4D GetCol (unsigned int col) const

    Returns specified col. Col should be between [0,3]. If it is out of range the first col will be returned.
- void SetRow (unsigned int row, Vector4D v)

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

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

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

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

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

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

5.1.1 Detailed Description

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

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

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Matrix4x4() [1/3]

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

Default Constructor.

Creates a new 4x4 identity matrix.

5.1.2.2 Matrix4x4() [2/3]

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

Overloaded Constructor.

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

5.1.2.3 Matrix4x4() [3/3]

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

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

5.1.3 Member Function Documentation

5.1.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.1.3.2 Data() [2/2]

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

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

5.1.3.3 GetCol()

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

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

5.1.3.4 GetRow()

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

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

5.1.3.5 operator()() [1/2]

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

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

5.1.3.6 operator()() [2/2]

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

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

5.1.3.7 operator*=() [1/2]

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

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

5.1.3.8 operator*=() [2/2]

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

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

5.1.3.9 operator+=()

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

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

5.1.3.10 operator-=()

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

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

5.1.3.11 SetCol()

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

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

5.1.3.12 SetRow()

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

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

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

- FAMathEngine.h

5.2 FAMath::Quaternion Class Reference

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

Public Member Functions

- Quaternion ()

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

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

    Overloaded Constructor.
- Quaternion (const Vector4D &v)

    Overloaded Constructor.
- float GetScalar () const

    Returns the scalar component of the quaternion.
- float GetX () const

    Returns the x value of the vector component in the quaternion.
- float GetY () const

    Returns the y value of the vector component in the quaternion.
- float GetZ () const

    Returns the z value of the vector component in the quaternion.
- const Vector3D & GetVector () const

    Returns the vector component of the quaternion.
- void SetScalar (float scalar)

Sets the scalar component to the specified value.
- void SetX (float x)

    Sets the x component to the specified value.
- void SetY (float y)

    Sets the y component to the specified value.
- void SetZ (float z)

    Sets the z component to the specified value.
- void SetVector (const Vector3D &v)

    Sets the vector to the specified vector.
- Quaternion & operator+= (const Quaternion &q)

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

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

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

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

5.2.1 Detailed Description

The datatype for the components is float.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Quaternion() [1/4]

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

Default Constructor.

Constructs an identity quaternion.

5.2.2.2 Quaternion() [2/4]

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

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.3 Quaternion() [3/4]

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

Overloaded Constructor.

Constructs a quaternion with the given values.

5.2.2.4 Quaternion() [4/4]

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

Overloaded Constructor.

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

5.2.3 Member Function Documentation

5.2.3.1 GetScalar()

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

Returns the scalar component of the quaternion.

5.2.3.2 GetVector()

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

Returns the vector component of the quaternion.

5.2.3.3 GetX()

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

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

5.2.3.4 GetY()

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

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

5.2.3.5 GetZ()

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

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

5.2.3.6 operator*=() [1/2]

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

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

5.2.3.9 operator-=()

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

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

5.2.3.10 SetScalar()

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

Sets the scalar component to the specified value.

5.2.3.11 SetVector()

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

Sets the vector to the specified vector.

5.2.3.12 SetX()

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

Sets the x component to the specified value.

5.2.3.13 SetY()

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

Sets the y component to the specified value.

5.2.3.14 SetZ()

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

Sets the z component to the specified value.

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

- FAMathEngine.h

5.3 FAMath::Vector2D Class Reference

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

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

Public Member Functions

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

5.3.1 Detailed Description

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

The datatype for the components is float.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Vector2D() [1/2]

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

Default Constructor.

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

5.3.2.2 Vector2D() [2/2]

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

Overloaded Constructor.

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

5.3.3 Member Function Documentation

5.3.3.1 GetX()

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

Returns the x component.

5.3.3.2 GetY()

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

Returns y component.

5.3.3.3 operator*=()

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

2D vector scalar multiplication through overloading operator *=.

5.3.3.4 operator+=()

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

2D vector addition through overloading operator +=.

5.3.3.5 operator-=()

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

2D vector subtraction through overloading operator -=.

5.3.3.6 operator/=()

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

2D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.3.3.7 SetX()

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

Sets the x component to the specified value.

5.3.3.8 SetY()

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

Sets the y component to the specified value.

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

- FAMathEngine.h

5.4 FAMath::Vector3D Class Reference

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

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

Public Member Functions

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

5.4.1 Detailed Description

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

The datatype for the components is float.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Vector3D() `[1/2]`

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

Default Constructor.

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

5.4.2.2 Vector3D() [2/2]

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

Overloaded Constructor.

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

5.4.3 Member Function Documentation

5.4.3.1 GetX()

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

Returns the x component.

5.4.3.2 GetY()

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

Returns y component.

5.4.3.3 GetZ()

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

Returns the z component.

5.4.3.4 operator*=()

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

3D vector scalar multiplication through overloading operator *=.

5.4.3.5 operator+=()

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

3D vector addition through overloading operator +=.

5.4.3.6 operator-=()

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

3D vector subtraction through overloading operator -=.

5.4.3.7 operator/=()

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

3D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.4.3.8 SetX()

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

Sets the x component to the specified value.

5.4.3.9 SetY()

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

Sets the y component to the specified value.

5.4.3.10 SetZ()

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

Sets the z component to the specified value.

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

- FAMathEngine.h

5.5 FAMath::Vector4D Class Reference

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

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

Public Member Functions

- Vector4D ()

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

    Overloaded Constructor.
- float GetX () const

    Returns the x component.
- float GetY () const

    Returns the y component.
- float GetZ () const

    Returns the z component.
- float GetW () const

    Returns the w component.
- void SetX (float x)

    Sets the x component to the specified value.
- void SetY (float y)

    Sets the y component to the specified value.
- void SetZ (float z)

    Sets the z component to the specified value.
- void SetW (float w)

    Sets the w component to the specified value.
- Vector4D & operator+= (const Vector4D &b)

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

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

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

    4D vector scalar division through overloading operator /=.

5.5.1 Detailed Description

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

The datatype for the components is float

5.5.2 Constructor & Destructor Documentation

5.5.2.1 Vector4D() [1/2]

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

Default Constructor.

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

5.5.2.2 Vector4D() [2/2]

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

Overloaded Constructor.

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

5.5.3 Member Function Documentation

5.5.3.1 GetW()

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

Returns the w component.

5.5.3.2 GetX()

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

Returns the x component.

5.5.3.3 GetY()

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

Returns the y component.

5.5.3.4 GetZ()

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

Returns the z component.

5.5.3.5 operator*=()

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

4D vector scalar multiplication through overloading operator *=.

5.5.3.6 operator+=()

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

4D vector addition through overloading operator +=.

5.5.3.7 operator-=()

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

4D vector subtraction through overloading operator -=.

5.5.3.8 operator/=()

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

4D vector scalar division through overloading operator /=.

If k is zero, the vector is unchanged.

5.5.3.9 SetW()

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

Sets the w component to the specified value.

5.5.3.10 SetX()

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

Sets the x component to the specified value.

5.5.3.11 SetY()

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

Sets the y component to the specified value.

5.5.3.12 SetZ()

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

Sets the z component to the specified value.

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

- FAMathEngine.h

Chapter 6

File Documentation

6.1 FAMathEngine.h

```
#pragma once

#include <cmath>

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


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

namespace FAMath
{

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

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

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

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


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

    class Vector2D
    {
    public:

        Vector2D();

        Vector2D(float x, float y);

        float GetX() const;

```

```
       float GetY() const;

       void SetX(float x);

       void SetY(float y);

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

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

       Vector2D& operator*=(float k);

        Vector2D& operator/=(float k);

    private:
        float mX;
        float mY;
    };


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

    inline Vector2D::Vector2D() :  mX{ 0.0f }, mY{ 0.0f }
    {}

    inline Vector2D::Vector2D(float x, float y) :  mX{ x }, mY{ y }
    {}

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

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

    inline float Vector2D::GetX()const
{
        return mX;
    }

    inline float Vector2D::GetY()const
{
        return mY;
    }

    inline void Vector2D::SetX(float x)
    {
        mX = x;
    }

    inline void Vector2D::SetY(float y)
    {
        mY = y;
    }

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


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

    inline Vector2D& Vector2D::operator+=(const Vector2D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;

        return *this;
    }

    inline Vector2D& Vector2D::operator-=(const Vector2D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;

        return *this;
    }

    inline Vector2D& Vector2D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;

        return *this;
    }

    inline Vector2D& Vector2D::operator/=(float k)
    {
        if (CompareFloats(k, 0.0f, EPSILON))
```

```
        {
            return *this;
        }

        this->mX /= k;
        this->mY /= k;

        return *this;
    }

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

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

    inline bool ZeroVector(const Vector2D& a)
    {
        if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
    {
        return Vector2D(a.GetX() + b.GetX(), a.GetY() + b.GetY());
    }

    inline Vector2D operator-(const Vector2D& v)
    {
        return Vector2D(-v.GetX(), -v.GetY());
    }

    inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
    {
        return Vector2D(a.GetX() - b.GetX(), a.GetY() - b.GetY());
    }

    inline Vector2D operator*(const Vector2D& a, float k)
    {
        return Vector2D(a.GetX() * k, a.GetY() * k);
    }

    inline Vector2D operator*(float k, const Vector2D& a)
    {
        return Vector2D(k * a.GetX(), k * a.GetY());
    }

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

        return Vector2D(a.GetX() / k, a.GetY() / k);
    }

    inline float DotProduct(const Vector2D& a, const Vector2D& b)
    {
        return a.GetX() * b.GetX() + a.GetY() * b.GetY();
    }

    inline float Length(const Vector2D& v)
    {
        return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY());
    }

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

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

        float mag{ Length(v) };

        return Vector2D(v.GetX() / mag, v.GetY() / mag);
    }

    inline Vector2D PolarToCartesian(const Vector2D& v)
    {
```

```
        //v = (r, theta)
        //x = rcos((theta)
        //y = rsin(theta)
        float angle = v.GetY() * PI / 180.0f;

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

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

        if (CompareFloats(v.GetX(), 0.0f, EPSILON))
        {
            return v;
        }

        double theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
        return Vector2D(Length(v), theta);
    }

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

        Vector2D normB(Norm(b));
        return Vector2D(DotProduct(a, normB) * normB);
    }


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


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




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

        Vector3D();

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

        float GetX() const;

        float GetY() const;

        float GetZ() const;

        void SetX(float x);

        void SetY(float y);

        void SetZ(float z);

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

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

        Vector3D& operator*=(float k);

        Vector3D& operator/=(float k);

    private:
        float mX;
        float mY;
        float mZ;
    };

    //--------------------------------------------------------------------------------------
    //Vector3D Constructors

    inline Vector3D::Vector3D() :  mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }
    {}
```

```

    inline Vector3D::Vector3D(float x, float y, float z) :  mX{ x }, mY{ y }, mZ{ z }
    {}

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

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

    inline float Vector3D::GetX()const
{
        return mX;
    }

    inline float Vector3D::GetY()const
{
        return mY;
    }

    inline float Vector3D::GetZ()const
{
        return mZ;
    }

    inline void Vector3D::SetX(float x)
    {
        mX = x;
    }

    inline void Vector3D::SetY(float y)
    {
        mY = y;
    }

    inline void Vector3D::SetZ(float z)
    {
        mZ = z;
    }
    //--------------------------------------------------------------------------------------


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

    inline Vector3D& Vector3D::operator+=(const Vector3D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;
        this->mZ += b.mZ;

        return *this;
    }

    inline Vector3D& Vector3D::operator-=(const Vector3D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;
        this->mZ -= b.mZ;

        return *this;
    }

    inline Vector3D& Vector3D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;
        this->mZ *= k;

        return *this;
    }

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

        this->mX /= k;
        this->mY /= k;
        this->mZ /= k;

        return *this;
    }

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

```

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

inline bool ZeroVector(const Vector3D& a)
{
    if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
        CompareFloats(a.GetZ(), 0.0f, EPSILON))
    {
        return true;
    }

    return false;
}

inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
{
    return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
}

inline Vector3D operator-(const Vector3D& v)
{
    return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
}

inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
{
    return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
}

inline Vector3D operator*(const Vector3D& a, float k)
{
    return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
}

inline Vector3D operator*(float k, const Vector3D& a)
{
    return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
}

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

    return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
}

inline float DotProduct(const Vector3D& a, const Vector3D& b)
{
    //a dot b = axbx + ayby + azbz
    return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
}

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

    return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
        a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
        a.GetX() * b.GetY() - a.GetY() * b.GetX());
}

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

    return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
}

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

    float mag{ Length(v) };

    return Vector3D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag);
}

inline Vector3D CylindricalToCartesian(const Vector3D& v)
{
```

```
        //v = (r, theta, z)
        //x = rcos(theta)
        //y = rsin(theta)
        //z = z
        double angle{ v.GetY() * PI / 180.0f };

        return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
    }

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

        double theta{ atan2(v.GetY(), v.GetX()) * 180.0 / PI };
        return Vector3D(Length(v), theta, v.GetZ());
    }

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

        double phi{ v.GetY() * PI / 180.0 };
        double theta{ v.GetZ() * PI / 180.0 };

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

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

        if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
        {
            return v;
        }

        double pho{ Length(v) };
        double phi{ acos(v.GetZ() / pho) * 180.0 / PI };
        double theta{ atan2(v.GetY(), v.GetX()) * 180.0 / PI };

        return Vector3D(pho, phi, theta);
    }

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

        Vector3D normB(Norm(b));
        return Vector3D(DotProduct(a, normB) * normB);
    }

    inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
    {
        x = Norm(x);
        y = Norm(CrossProduct(z, x));
        z = Norm(CrossProduct(x, y));
    }


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


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



```

```
//-----------------------------------------------------------------------------------------------------------------------
class Vector4D
{
public:
    Vector4D();

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

    float GetX() const;

    float GetY() const;

    float GetZ() const;

    float GetW() const;

    void SetX(float x);

    void SetY(float y);

    void SetZ(float z);

    void SetW(float w);

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

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

    Vector4D& operator*=(float k);

    Vector4D& operator/=(float k);

private:
    float mX;
    float mY;
    float mZ;
    float mW;
};

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

inline Vector4D::Vector4D() :  mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }, mW{ 0.0f }
{}

inline Vector4D::Vector4D(float x, float y, float z, float w) :  mX{ x }, mY{ y }, mZ{ z }, mW{ w }
{}

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

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

inline float Vector4D::GetX()const
{
    return mX;
}

inline float Vector4D::GetY()const
{
    return mY;
}

inline float Vector4D::GetZ()const
{
    return mZ;
}

inline float Vector4D::GetW()const
{
    return mW;
}

inline void Vector4D::SetX(float x)
{
    mX = x;
}

inline void Vector4D::SetY(float y)
{
    mY = y;
}

inline void Vector4D::SetZ(float z)
{
    mZ = z;
```

```
    }

    inline void Vector4D::SetW(float w)
    {
        mW = w;
    }
    //--------------------------------------------------------------------------------------


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

    inline Vector4D& Vector4D::operator+=(const Vector4D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;
        this->mZ += b.mZ;
        this->mW += b.mW;

        return *this;
    }

    inline Vector4D& Vector4D::operator-=(const Vector4D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;
        this->mZ -= b.mZ;
        this->mW -= b.mW;

        return *this;
    }

    inline Vector4D& Vector4D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;
        this->mZ *= k;
        this->mW *= k;

        return *this;
    }

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

        this->mX /= k;
        this->mY /= k;
        this->mZ /= k;
        this->mW /= k;

        return *this;
    }

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

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

    inline bool ZeroVector(const Vector4D& a)
    {
        if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
            CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
  b.GetW());
    }

    inline Vector4D operator-(const Vector4D& v)
    {
        return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
    }

    inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
```

```
b.GetW());
    }

    inline Vector4D operator*(const Vector4D& a, float k)
    {
        return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
    }

    inline Vector4D operator*(float k, const Vector4D& a)
    {
        return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
    }

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

        return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
    }

    inline float DotProduct(const Vector4D& a, const Vector4D& b)
    {
        //a dot b = axbx + ayby + azbz + awbw
        return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
    }

    inline float Length(const Vector4D& v)
    {
        //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
        return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
 v.GetW());
    }

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

        float mag{ Length(v) };

        return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
    }

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


#if defined(_DEBUG)
    inline void print(const Vector4D& v)
    {
        std::cout « "(" « v.GetX() « ", " « v.GetY() « ", " « v.GetZ() « ", " « v.GetW() « ")";
    }
#endif
    //--------------------------------------------------------------------------------------


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



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

        Matrix4x4();

        Matrix4x4(float a[][4]);

        Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);

        float* Data();

        const float* Data() const;

```

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

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

        Vector4D GetRow(unsigned int row) const;

        Vector4D GetCol(unsigned int col) const;

        void SetRow(unsigned int row, Vector4D v);

        void SetCol(unsigned int col, Vector4D v);

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

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

        Matrix4x4& operator*=(float k);

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

    private:

        float mMat[4][4];
    };

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

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

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

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



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

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

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

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

    inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
  Vector4D& r4)
    {
        SetRow(0, r1);
        SetRow(1, r2);
```

```
        SetRow(2, r3);
        SetRow(3, r4);
    }

    inline float* Matrix4x4::Data()
    {
        return mMat[0];
    }

    inline const float* Matrix4x4::Data()const
{
        return mMat[0];
    }

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

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

    inline Vector4D Matrix4x4::GetRow(unsigned int row)const
{
        if (row < 0 || row > 3)
            return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
        else
            return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);

    }

    inline Vector4D Matrix4x4::GetCol(unsigned int col)const
{
        if (col < 0 || col > 3)
            return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
        else
            return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
    }

    inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
    {
        if (row < 0 || row > 3)
        {
            mMat[0][0] = v.GetX();
            mMat[0][1] = v.GetY();
            mMat[0][2] = v.GetZ();
            mMat[0][3] = v.GetW();
        }
        else
        {
            mMat[row][0] = v.GetX();
            mMat[row][1] = v.GetY();
            mMat[row][2] = v.GetZ();
            mMat[row][3] = v.GetW();
        }
    }

    inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
    {
        if (col < 0 || col > 3)
        {
            mMat[0][0] = v.GetX();
            mMat[1][0] = v.GetY();
            mMat[2][0] = v.GetZ();
            mMat[3][0] = v.GetW();
        }
        else
        {
            mMat[0][col] = v.GetX();
            mMat[1][col] = v.GetY();
            mMat[2][col] = v.GetZ();
```

```
            mMat[3][col] = v.GetW();
        }
    }

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

        return *this;
    }

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

        return *this;
    }

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

        return *this;
    }

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

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

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

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

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

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

        return *this;
    }

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

```
            {
                res(i, j) = m1(i, j) + m2(i, j);
            }
        }

        return res;
    }

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

        return res;
    }

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

        return res;
    }

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

        return res;
    }

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

        return res;
    }

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

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

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

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

            res(i, 3) = (m1(i, 0) * m2(0, 3)) +
                (m1(i, 1) * m2(1, 3)) +
```

```
                (m1(i, 2) * m2(2, 3)) +
                (m1(i, 3) * m2(3, 3));
        }

        return res;
    }

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

        res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());

        res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());

        res.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());

        res.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());

        return res;
    }

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

        res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));

        res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));

        res.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));

        res.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));

        return res;
    }

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

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

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

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

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

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

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

                }
                else
                {
                    if (!CompareFloats(m(i, j), 0.0f, EPSILON))
                        return false;

                }
```

```
            }
        }
    }

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

        Matrix4x4 res;

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

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

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

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

        return res;
    }

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

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

        return cm * t;
    }

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

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

        return cm * s;
    }

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

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

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

        Matrix4x4 r;

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

```

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

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

        return cm * r;
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```
            }
            tc = 0;
            ++tr;
        }

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

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

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

        return Transpose(cA);
    }

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

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


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

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


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




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

    class Quaternion
    {
    public:

        Quaternion();

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

        Quaternion(float scalar, const Vector3D& v);

        Quaternion(const Vector4D& v);

        float GetScalar() const;

        float GetX() const;

        float GetY() const;

        float GetZ() const;

        const Vector3D& GetVector() const;
```

```
        void SetScalar(float scalar);

        void SetX(float x);

        void SetY(float y);

        void SetZ(float z);

        void SetVector(const Vector3D& v);

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

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

        Quaternion& operator*=(float k);

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

    private:
        float mScalar;
        float mX;
        float mY;
        float mZ;
    };

    //--------------------------------------------------------------------------------------
    inline Quaternion::Quaternion() :  mScalar{ 1.0f }, mX{ 0.0f }, mY{ 0.0f }, mZ{ 0.0f }
    {
    }

    inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
        mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
    {
    }

    inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
        mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
    {
    }

    inline Quaternion::Quaternion(const Vector4D& v) :
        mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
    {
    }

    inline float Quaternion::GetScalar()const
{
        return mScalar;
    }

    inline float Quaternion::GetX()const
{
        return mX;
    }

    inline float Quaternion::GetY()const
{
        return mY;
    }

    inline float Quaternion::GetZ()const
{
        return mZ;
    }

    inline const Vector3D& Quaternion::GetVector()const
{
        return Vector3D(mX, mY, mZ);
    }

    inline void Quaternion::SetScalar(float scalar)
    {
        mScalar = scalar;
    }

    inline void Quaternion::SetX(float x)
    {
        mX = x;
    }

    inline void Quaternion::SetY(float y)
    {
        mY = y;
    }

    inline void  Quaternion::SetZ(float z)
```

```
    {
        mZ = z;
    }

    inline void Quaternion::SetVector(const Vector3D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = v.GetZ();
    }

    inline Quaternion& Quaternion::operator+=(const Quaternion& q)
    {
        this->mScalar += q.mScalar;
        this->mX += q.mX;
        this->mY += q.mY;
        this->mZ += q.mZ;

        return *this;
    }

    inline Quaternion& Quaternion::operator-=(const Quaternion& q)
    {
        this->mScalar -= q.mScalar;
        this->mX -= q.mX;
        this->mY -= q.mY;
        this->mZ -= q.mZ;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(float k)
    {
        this->mScalar *= k;
        this->mX *= k;
        this->mY *= k;
        this->mZ *= k;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(const Quaternion& q)
    {
        Vector3D thisVector(this->mX, this->mY, this->mZ);
        Vector3D qVector(q.mX, q.mY, q.mZ);

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

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

        this->mScalar = resultScalar;
        this->mX = resultVector.GetX();
        this->mY = resultVector.GetY();
        this->mZ = resultVector.GetZ();

        return *this;
    }

    inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
  q2.GetY(), q1.GetZ() + q2.GetZ());
    }

    inline Quaternion operator-(const Quaternion& q)
    {
        return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
    }

    inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion(q1.GetScalar() - q2.GetScalar(),
            q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
    }

    inline Quaternion operator*(float k, const Quaternion& q)
    {
        return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
    }

    inline Quaternion operator*(const Quaternion& q, float k)
    {
```

```
        return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
    }

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

        Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
        Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());

        float s{ q1.GetScalar() * q2.GetScalar() };
        float dP{ DotProduct(q1Vector, q2Vector) };
        float resultScalar{ s - dP };

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

        return Quaternion(resultScalar, resultVector);
    }

    inline bool IsZeroQuaternion(const Quaternion& q)
    {
        //zero quaternion = (0, 0, 0, 0)
        return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
            CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
    }

    inline bool IsIdentity(const Quaternion& q)
    {
        //identity quaternion = (1, 0, 0, 0)
        return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
            CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
    }

    inline Quaternion Conjugate(const Quaternion& q)
    {
        //conjugate of a quaternion is the quaternion with its vector part negated
        return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
    }

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

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

        if (IsZeroQuaternion(q))
            return q;

        double d{ Length(q) };

        return Quaternion(q.GetScalar() / d, q.GetX() / d, q.GetY() / d, q.GetZ() / d);
    }

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

        if (IsZeroQuaternion(q))
            return q;

        Quaternion conjugateOfQ(Conjugate(q));

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

        return Quaternion(conjugateOfQ.GetScalar() / d, conjugateOfQ.GetX() / d,
            conjugateOfQ.GetY() / d, conjugateOfQ.GetZ() / d);
    }

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

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

```
        double s{ sin(ang * PI / 180.0) };

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

        return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
    }

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

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

        Vector3D axisN(Norm(axis));

        return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
    }

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

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

        Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
        axis = Norm(axis);

        return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
    }

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

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

        colMat[0][0] = 1.0 - 2.0 * q.GetY() * q.GetY() - 2.0 * q.GetZ() * q.GetZ();
        colMat[0][1] = 2.0 * q.GetX() * q.GetY() - 2.0 * q.GetScalar() * q.GetZ();
        colMat[0][2] = 2.0 * q.GetX() * q.GetZ() + 2.0 * q.GetScalar() * q.GetY();
        colMat[0][3] = 0.0f;

        colMat[1][0] = 2.0 * q.GetX() * q.GetY() + 2.0 * q.GetScalar() * q.GetZ();
        colMat[1][1] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetZ() * q.GetZ();
        colMat[1][2] = 2.0 * q.GetY() * q.GetZ() - 2.0 * q.GetScalar() * q.GetX();
        colMat[1][3] = 0.0f;

        colMat[2][0] = 2.0 * q.GetX() * q.GetZ() - 2.0 * q.GetScalar() * q.GetY();
        colMat[2][1] = 2.0 * q.GetY() * q.GetZ() + 2.0 * q.GetScalar() * q.GetX();
        colMat[2][2] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetY() * q.GetY();
        colMat[2][3] = 0.0f;

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

        return Matrix4x4(colMat);
    }

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

```

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

        rowMat[0][0] = 1.0 - 2.0 * q.GetY() * q.GetY() - 2.0 * q.GetZ() * q.GetZ();
        rowMat[0][1] = 2.0 * q.GetX() * q.GetY() + 2.0 * q.GetScalar() * q.GetZ();
        rowMat[0][2] = 2.0 * q.GetX() * q.GetZ() - 2.0 * q.GetScalar() * q.GetY();
        rowMat[0][3] = 0.0f;

        rowMat[1][0] = 2.0 * q.GetX() * q.GetY() - 2.0 * q.GetScalar() * q.GetZ();
        rowMat[1][1] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetZ() * q.GetZ();
        rowMat[1][2] = 2.0 * q.GetY() * q.GetZ() + 2.0 * q.GetScalar() * q.GetX();
        rowMat[1][3] = 0.0f;

        rowMat[2][0] = 2.0 * q.GetX() * q.GetZ() + 2.0 * q.GetScalar() * q.GetY();
        rowMat[2][1] = 2.0 * q.GetY() * q.GetZ() - 2.0 * q.GetScalar() * q.GetX();
        rowMat[2][2] = 1.0 - 2.0 * q.GetX() * q.GetX() - 2.0 * q.GetY() * q.GetY();
        rowMat[2][3] = 0.0f;

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

        return Matrix4x4(rowMat);
    }

#if defined(_DEBUG)
    inline void print(const Quaternion& q)
    {
        std::cout « "(" « q.GetScalar() « ", " « q.GetX() « ", " « q.GetY() « ", " « q.GetZ();
    }
#endif
    //--------------------------------------------------------------------------------------


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

Index