

Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4



|                                   |          |
|-----------------------------------|----------|
| <b>1 Namespace Index</b>          | <b>1</b> |
| 1.1 Namespace List                | 1        |
| <b>2 Class Index</b>              | <b>3</b> |
| 2.1 Class List                    | 3        |
| <b>3 File Index</b>               | <b>5</b> |
| 3.1 File List                     | 5        |
| <b>4 Namespace Documentation</b>  | <b>7</b> |
| 4.1 FAMath Namespace Reference    | 7        |
| 4.1.1 Detailed Description        | 10       |
| 4.1.2 Function Documentation      | 11       |
| 4.1.2.1 Adjoint()                 | 11       |
| 4.1.2.2 CartesianToCylindrical()  | 11       |
| 4.1.2.3 CartesianToPolar()        | 11       |
| 4.1.2.4 CartesianToSpherical()    | 11       |
| 4.1.2.5 Cofactor()                | 12       |
| 4.1.2.6 CompareDoubles()          | 12       |
| 4.1.2.7 CompareFloats()           | 12       |
| 4.1.2.8 Conjugate()               | 12       |
| 4.1.2.9 CrossProduct()            | 12       |
| 4.1.2.10 CylindricalToCartesian() | 13       |
| 4.1.2.11 Det()                    | 13       |
| 4.1.2.12 DotProduct() [1/3]       | 13       |
| 4.1.2.13 DotProduct() [2/3]       | 13       |
| 4.1.2.14 DotProduct() [3/3]       | 13       |
| 4.1.2.15 Inverse() [1/2]          | 14       |
| 4.1.2.16 Inverse() [2/2]          | 14       |
| 4.1.2.17 IsIdentity() [1/2]       | 14       |
| 4.1.2.18 IsIdentity() [2/2]       | 14       |
| 4.1.2.19 IsZeroQuaternion()       | 14       |
| 4.1.2.20 Length() [1/4]           | 14       |
| 4.1.2.21 Length() [2/4]           | 15       |
| 4.1.2.22 Length() [3/4]           | 15       |
| 4.1.2.23 Length() [4/4]           | 15       |
| 4.1.2.24 Norm() [1/3]             | 15       |
| 4.1.2.25 Norm() [2/3]             | 15       |
| 4.1.2.26 Norm() [3/3]             | 15       |
| 4.1.2.27 Normalize()              | 16       |
| 4.1.2.28 operator*() [1/14]       | 16       |
| 4.1.2.29 operator*() [2/14]       | 16       |
| 4.1.2.30 operator*() [3/14]       | 16       |

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

|  |           |
|--|-----------|
| 4.1.2.73 SetTolIdentity()                    | 25        |
| 4.1.2.74 SphericalToCartesian()              | 25        |
| 4.1.2.75 Translate()                         | 25        |
| 4.1.2.76 Transpose()                         | 25        |
| 4.1.2.77 ZeroVector() [1/3]                  | 26        |
| 4.1.2.78 ZeroVector() [2/3]                  | 26        |
| 4.1.2.79 ZeroVector() [3/3]                  | 26        |
| <b>5 Class Documentation</b>                 | <b>27</b> |
| 5.1 FAMath::Matrix4x4 Class Reference        | 27        |
| 5.1.1 Detailed Description                   | 28        |
| 5.1.2 Constructor & Destructor Documentation | 28        |
| 5.1.2.1 Matrix4x4() [1/3]                    | 28        |
| 5.1.2.2 Matrix4x4() [2/3]                    | 28        |
| 5.1.2.3 Matrix4x4() [3/3]                    | 28        |
| 5.1.3 Member Function Documentation          | 28        |
| 5.1.3.1 Data() [1/2]                         | 28        |
| 5.1.3.2 Data() [2/2]                         | 29        |
| 5.1.3.3 GetCol()                             | 29        |
| 5.1.3.4 GetRow()                             | 29        |
| 5.1.3.5 operator>() [1/2]                    | 29        |
| 5.1.3.6 operator>() [2/2]                    | 29        |
| 5.1.3.7 operator*=( ) [1/2]                  | 30        |
| 5.1.3.8 operator*=( ) [2/2]                  | 30        |
| 5.1.3.9 operator+=( )                        | 30        |
| 5.1.3.10 operator-=( )                       | 30        |
| 5.1.3.11 SetCol()                            | 30        |
| 5.1.3.12 SetRow()                            | 31        |
| 5.2 FAMath::Quaternion Class Reference       | 31        |
| 5.2.1 Detailed Description                   | 32        |
| 5.2.2 Constructor & Destructor Documentation | 32        |
| 5.2.2.1 Quaternion() [1/3]                   | 32        |
| 5.2.2.2 Quaternion() [2/3]                   | 32        |
| 5.2.2.3 Quaternion() [3/3]                   | 32        |
| 5.2.3 Member Function Documentation          | 32        |
| 5.2.3.1 GetScalar()                          | 33        |
| 5.2.3.2 GetVector()                          | 33        |
| 5.2.3.3 GetX()                               | 33        |
| 5.2.3.4 GetY()                               | 33        |
| 5.2.3.5 GetZ()                               | 33        |
| 5.2.3.6 operator*=( ) [1/2]                  | 33        |
| 5.2.3.7 operator*=( ) [2/2]                  | 34        |

|  |    |
|--|----|
| 5.2.3.8 operator+=()                         | 34 |
| 5.2.3.9 operator-=()                         | 34 |
| 5.2.3.10 SetScalar()                         | 34 |
| 5.2.3.11 SetVector()                         | 34 |
| 5.2.3.12 SetX()                              | 34 |
| 5.2.3.13 SetY()                              | 35 |
| 5.2.3.14 SetZ()                              | 35 |
| 5.3 FMath::Vector2D Class Reference          | 35 |
| 5.3.1 Detailed Description                   | 36 |
| 5.3.2 Constructor & Destructor Documentation | 36 |
| 5.3.2.1 Vector2D() [1/3]                     | 36 |
| 5.3.2.2 Vector2D() [2/3]                     | 36 |
| 5.3.2.3 Vector2D() [3/3]                     | 36 |
| 5.3.3 Member Function Documentation          | 36 |
| 5.3.3.1 GetX()                               | 36 |
| 5.3.3.2 GetY()                               | 37 |
| 5.3.3.3 operator*=( )                        | 37 |
| 5.3.3.4 operator+=()                         | 37 |
| 5.3.3.5 operator-=()                         | 37 |
| 5.3.3.6 operator/=()                         | 37 |
| 5.3.3.7 operator=( ) [1/2]                   | 37 |
| 5.3.3.8 operator=( ) [2/2]                   | 38 |
| 5.3.3.9 SetX()                               | 38 |
| 5.3.3.10 SetY()                              | 38 |
| 5.4 FMath::Vector3D Class Reference          | 38 |
| 5.4.1 Detailed Description                   | 39 |
| 5.4.2 Constructor & Destructor Documentation | 39 |
| 5.4.2.1 Vector3D() [1/3]                     | 40 |
| 5.4.2.2 Vector3D() [2/3]                     | 40 |
| 5.4.2.3 Vector3D() [3/3]                     | 40 |
| 5.4.3 Member Function Documentation          | 40 |
| 5.4.3.1 GetX()                               | 40 |
| 5.4.3.2 GetY()                               | 40 |
| 5.4.3.3 GetZ()                               | 41 |
| 5.4.3.4 operator*=( )                        | 41 |
| 5.4.3.5 operator+=()                         | 41 |
| 5.4.3.6 operator-=()                         | 41 |
| 5.4.3.7 operator/=()                         | 41 |
| 5.4.3.8 operator=( ) [1/2]                   | 41 |
| 5.4.3.9 operator=( ) [2/2]                   | 42 |
| 5.4.3.10 SetX()                              | 42 |
| 5.4.3.11 SetY()                              | 42 |

|  |    |
|--|----|
| 5.4.3.12 SetZ()                              | 42 |
| 5.5 FAMath::Vector4D Class Reference         | 42 |
| 5.5.1 Detailed Description                   | 43 |
| 5.5.2 Constructor & Destructor Documentation | 43 |
| 5.5.2.1 Vector4D() [1/3]                     | 44 |
| 5.5.2.2 Vector4D() [2/3]                     | 44 |
| 5.5.2.3 Vector4D() [3/3]                     | 44 |
| 5.5.3 Member Function Documentation          | 44 |
| 5.5.3.1 GetW()                               | 44 |
| 5.5.3.2 GetX()                               | 44 |
| 5.5.3.3 GetY()                               | 45 |
| 5.5.3.4 GetZ()                               | 45 |
| 5.5.3.5 operator*=( )                        | 45 |
| 5.5.3.6 operator+=( )                        | 45 |
| 5.5.3.7 operator-=( )                        | 45 |
| 5.5.3.8 operator/=( )                        | 45 |
| 5.5.3.9 operator=( ) [1/2]                   | 46 |
| 5.5.3.10 operator=( ) [2/2]                  | 46 |
| 5.5.3.11 SetW()                              | 46 |
| 5.5.3.12 SetX()                              | 46 |
| 5.5.3.13 SetY()                              | 46 |
| 5.5.3.14 SetZ()                              | 46 |
| 6 File Documentation                         | 47 |
| 6.1 FAMathEngine.h                           | 47 |
| Index  | 71 |





# Chapter 1

## Namespace Index

### 1.1 Namespace List

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

|                        |  |                   |
|------------------------|--|-------------------|
| <a href="#">FAMath</a> | Has utility functions, <a href="#">Vector2D</a> , <a href="#">Vector3D</a> , <a href="#">Vector4D</a> , <a href="#">Matrix4x4</a> , and <a href="#">Quaternion</a> classes . . . . | <a href="#">7</a> |
|------------------------|--|-------------------|



## Chapter 2

# Class Index

### 2.1 Class List

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

|   |                    |
|---|--------------------|
| <a href="#">FAMath::Matrix4x4</a>   |                    |
| A matrix class used for 4x4 matrices and their manipulations . . . . .      | <a href="#">27</a> |
| <a href="#">FAMath::Quaternion</a>  |                    |
| A quaternion class used for quaternions and their manipulations . . . . .   | <a href="#">31</a> |
| <a href="#">FAMath::Vector2D</a>  |                    |
| A vector class used for 2D vectors/points and their manipulations . . . . . | <a href="#">35</a> |
| <a href="#">FAMath::Vector3D</a>  |                    |
| A vector class used for 3D vectors/points and their manipulations . . . . . | <a href="#">38</a> |
| <a href="#">FAMath::Vector4D</a>  |                    |
| A vector class used for 4D vectors/points and their manipulations . . . . . | <a href="#">42</a> |



## Chapter 3

# File Index

### 3.1 File List

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

C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/[FAMathEngine.h](#)  
[47](#)



## Chapter 4

# Namespace Documentation

### 4.1 FAMath Namespace Reference

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

#### Classes

- class [Matrix4x4](#)  
*A matrix class used for 4x4 matrices and their manipulations.*
- class [Quaternion](#)  
*A quaternion class used for quaternions and their manipulations.*
- class [Vector2D](#)  
*A vector class used for 2D vectors/points and their manipulations.*
- class [Vector3D](#)  
*A vector class used for 3D vectors/points and their manipulations.*
- class [Vector4D](#)  
*A vector class used for 4D vectors/points and their manipulations.*

#### Functions

- bool [CompareFloats](#) (float x, float y, float epsilon)  
*Returns true if x and y are equal.*
- bool [CompareDoubles](#) (double x, double y, double epsilon)  
*Returns true if x and y are equal.*
- bool [ZeroVector](#) (const [Vector2D](#) &a)  
*Returns true if a is the zero vector.*
- [Vector2D](#) [operator+](#) (const [Vector2D](#) &a, const [Vector2D](#) &b)  
*Adds a with b and returns the result.*
- [Vector2D](#) [operator-](#) (const [Vector2D](#) &v)  
*Negates the vector v and returns the result.*
- [Vector2D](#) [operator-](#) (const [Vector2D](#) &a, const [Vector2D](#) &b)  
*Subtracts b from a and returns the result.*
- [Vector2D](#) [operator\\*](#) (const [Vector2D](#) &a, float k)  
*Returns  $a * k$ .*

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

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

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

*Returns the dot product between a and b.*
- float [Length](#) (const [Vector2D](#) &v)
 

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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



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

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

  - [Matrix4x4 Rotate](#) (const [Matrix4x4](#) &cm, float angle, float x, float y, float z)

*Construct a 4x4 rotation matrix with angle (in degrees) and axis (x, y, z) and post-multiply's it by cm.*
- double [Det](#) (const [Matrix4x4](#) &m)

*Returns the determinant m.*
- double [Cofactor](#) (const [Matrix4x4](#) &m, unsigned int row, unsigned int col)

*Returns the cofactor of the row and col in m.*
- [Matrix4x4 Adjoint](#) (const [Matrix4x4](#) &m)

*Returns the adjoint of m.*
- [Matrix4x4 Inverse](#) (const [Matrix4x4](#) &m)

*Returns the inverse of m.*
- [Quaternion operator+](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)

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

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

*Returns a quaternion that has the result of q1 - q2.*
- [Quaternion operator\\*](#) (float k, const [Quaternion](#) &q)

*Returns a quaternion that has the result of k \* q.*
- [Quaternion operator\\*](#) (const [Quaternion](#) &q, float k)

*Returns a quaternion that has the result of q \* k.*
- [Quaternion operator\\*](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)

*Returns a quaternion that has the result of q1 \* q2.*
- bool [IsZeroQuaternion](#) (const [Quaternion](#) &q)

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

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

*Returns the conjugate of quaternion q.*
- float [Length](#) (const [Quaternion](#) &q)

*Returns the length of quaternion q.*
- [Quaternion Normalize](#) (const [Quaternion](#) &q)

*Normalizes q and returns the normalized quaternion.*
- [Quaternion Inverse](#) (const [Quaternion](#) &q)

*Returns the invese of q.*
- [Quaternion RotationQuaternion](#) (float angle, float x, float y, float z)

*Returns a rotation quaternion from the axis-angle rotation representation.*
- [Quaternion RotationQuaternion](#) (float angle, const [Vector3D](#) &axis)

*Returns a quaternion from the axis-angle rotation representation.*
- [Quaternion RotationQuaternion](#) (const [Vector4D](#) &angAxis)

*Returns a quaternion from the axis-angle rotation representation.*
- [Matrix4x4 QuaternionToRotationMatrixCol](#) (const [Quaternion](#) &q)

*Transforms q into a column-major matrix.*
- [Matrix4x4 QuaternionToRotationMatrixRow](#) (const [Quaternion](#) &q)

*Transforms q into a row-major matrix.*

#### 4.1.1 Detailed Description

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

## 4.1.2 Function Documentation

### 4.1.2.1 Adjoint()

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

Returns the adjoint of  $m$ .

### 4.1.2.2 CartesianToCylindrical()

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

Converts the 3D vector  $v$  from cartesian coordinates to cylindrical coordinates.

$v$  should = (x, y, z).

If  $v_x$  is zero then no conversion happens and  $v$  is returned.

The returned 3D vector = (r, theta(degrees), z).

### 4.1.2.3 CartesianToPolar()

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

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

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

### 4.1.2.4 CartesianToSpherical()

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

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

If  $v$  is the zero vector or if  $v_x$  is zero then no conversion happens and  $v$  is returned.

The returned 3D vector = (r, phi(degrees), theta(degrees)).

#### 4.1.2.5 Cofactor()

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

Returns the cofactor of the *row* and *col* in *m*.

#### 4.1.2.6 CompareDoubles()

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

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

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

#### 4.1.2.7 CompareFloats()

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

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

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

#### 4.1.2.8 Conjugate()

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

Returns the conjugate of quaternion *q*.

#### 4.1.2.9 CrossProduct()

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

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

#### 4.1.2.10 CylindricalToCartesian()

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

Converts the 3D vector  $v$  from cylindrical coordinates to cartesian coordinates.

$v$  should = (r, theta(degrees), z).  
The returned 3D vector = (x, y ,z).

#### 4.1.2.11 Det()

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

Returns the determinant  $m$ .

#### 4.1.2.12 DotProduct() [1/3]

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

Returns the dot product between  $a$  and  $b$ .

#### 4.1.2.13 DotProduct() [2/3]

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

Returns the dot product between  $a$  and  $b$ .

#### 4.1.2.14 DotProduct() [3/3]

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

Returns the dot product between  $a$  and  $b$ .

**4.1.2.15 Inverse()** [1/2]

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

Returns the inverse of  $m$ .

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

**4.1.2.16 Inverse()** [2/2]

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

Returns the invese of  $q$ .

If  $q$  is the zero quaternion then  $q$  is returned.

**4.1.2.17 IsIdentity()** [1/2]

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

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

**4.1.2.18 IsIdentity()** [2/2]

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

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

**4.1.2.19 IsZeroQuaternion()**

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

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

**4.1.2.20 Length()** [1/4]

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

Returns the length of quaternion  $q$ .

**4.1.2.21 Length()** [2/4]

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

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

**4.1.2.22 Length()** [3/4]

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

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

**4.1.2.23 Length()** [4/4]

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

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

**4.1.2.24 Norm()** [1/3]

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

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

**4.1.2.25 Norm()** [2/3]

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

Normalizes the 3D vector *v*.

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

**4.1.2.26 Norm()** [3/3]

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

Normalizes the 4D vector *v*.

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

#### 4.1.2.27 Normalize()

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

Normalizes  $q$  and returns the normalized quaternion.

If  $q$  is the zero quaternion then  $q$  is returned.

#### 4.1.2.28 operator\*() [1/14]

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

Multiplies  $k$  with  $m$  and returns the result.

#### 4.1.2.29 operator\*() [2/14]

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

Multiplies  $m$  with  $k$  and returns the result.

#### 4.1.2.30 operator\*() [3/14]

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

Multiplies  $m$  with  $v$  and returns the result.

The vector  $v$  is a column vector.

#### 4.1.2.31 operator\*() [4/14]

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

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

Does  $m1 * m2$  in that order.



**4.1.2.32 operator\*() [5/14]**

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

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

**4.1.2.33 operator\*() [6/14]**

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

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

**4.1.2.34 operator\*() [7/14]**

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

Returns  $a * k$ .

**4.1.2.35 operator\*() [8/14]**

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

Returns  $a * k$ .

**4.1.2.36 operator\*() [9/14]**

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

Returns  $a * k$ .

**4.1.2.37 operator\*() [10/14]**

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

Multiplies  $v$  with  $m$  and returns the result.

The vector  $v$  is a row vector.

**4.1.2.38 operator\*() [11/14]**

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

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

**4.1.2.39 operator\*() [12/14]**

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

Returns  $k * a$ .

**4.1.2.40 operator\*() [13/14]**

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

Returns  $k * a$ .

**4.1.2.41 operator\*() [14/14]**

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

Returns  $k * a$ .

**4.1.2.42 operator+()** [1/5]

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

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

**4.1.2.43 operator+()** [2/5]

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

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

**4.1.2.44 operator+()** [3/5]

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

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

**4.1.2.45 operator+()** [4/5]

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

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

**4.1.2.46 operator+()** [5/5]

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

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

#### 4.1.2.47 operator-() [1/10]

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

Negates the 4x4 matrix *m*.

#### 4.1.2.48 operator-() [2/10]

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

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

#### 4.1.2.49 operator-() [3/10]

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

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

#### 4.1.2.50 operator-() [4/10]

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

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

#### 4.1.2.51 operator-() [5/10]

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

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

**4.1.2.52 operator-() [6/10]**

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

Negates the vector *v* and returns the result.

**4.1.2.53 operator-() [7/10]**

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

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

**4.1.2.54 operator-() [8/10]**

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

Negates the vector *v* and returns the result.

**4.1.2.55 operator-() [9/10]**

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

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

**4.1.2.56 operator-() [10/10]**

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

Negates *v* and returns the result.

#### 4.1.2.57 operator/() [1/3]

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

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

#### 4.1.2.58 operator/() [2/3]

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

Returns  $a / k$ .

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

#### 4.1.2.59 operator/() [3/3]

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

Returns  $a / k$ .

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

#### 4.1.2.60 Orthonormalize() [1/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

#### 4.1.2.61 Orthonormalize() [2/2]

```
void FAMath::Orthonormalize (
    Vector4D & x,
    Vector4D & y,
    Vector4D & z ) [inline]
```

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

**4.1.2.62 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)

**4.1.2.63 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.

**4.1.2.64 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.

**4.1.2.65 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.

**4.1.2.66 QuaternionToRotationMatrixCol()**

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

Transforms  $q$  into a column-major matrix.

$q$  should be a unit quaternion.

**4.1.2.67 QuaternionToRotationMatrixRow()**

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

Transforms  $q$  into a row-major matrix.

$q$  should be a unit quaternion.

**4.1.2.68 Rotate()**

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

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

Returns  $cm * rotate$ .

**4.1.2.69 RotationQuaternion() [1/3]**

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

Returns a quaternion from the axis-angle rotation representation.

The  $x$  value in the 4D vector  $v$  should be the angle(in degrees).

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

**4.1.2.70 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.

**4.1.2.71 RotationQuaternion() [3/3]**

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

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

The  $angle$  should be given in degrees.



#### 4.1.2.72 Scale()

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

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

Returns  $cm * scale$ .

#### 4.1.2.73 SetToIdentity()

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

Sets  $m$  to the identity matrix.

#### 4.1.2.74 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 = ( $x$ ,  $y$ ,  $z$ )

#### 4.1.2.75 Translate()

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

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

Returns  $cm * translate$ .

#### 4.1.2.76 Transpose()

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

Returns the tranpose of the given matrix  $m$ .

**4.1.2.77 ZeroVector()** [1/3]

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

Returns true if *a* is the zero vector.

**4.1.2.78 ZeroVector()** [2/3]

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

Returns true if *a* is the zero vector.

**4.1.2.79 ZeroVector()** [3/3]

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

Returns true if *a* is the zero vector.

## Chapter 5

# Class Documentation

### 5.1 FAMath::Matrix4x4 Class Reference

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

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

#### Public Member Functions

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

### 5.1.1 Detailed Description

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

The datatype for the components is float.

The 4x4 matrix is treated as a row-major matrix.

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 Matrix4x4() [1/3]

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

Creates a new 4x4 identity matrix.

#### 5.1.2.2 Matrix4x4() [2/3]

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

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

If *a* isn't a 4x4 matrix, the behavior is undefined.

#### 5.1.2.3 Matrix4x4() [3/3]

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

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

### 5.1.3 Member Function Documentation

#### 5.1.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

### 5.1.3.2 Data() [2/2]

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

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

### 5.1.3.3 GetCol()

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

Returns the specified *col*.

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

### 5.1.3.4 GetRow()

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

Returns the specified *row*.

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

### 5.1.3.5 operator()() [1/2]

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

Returns a reference to the element at the given (*row*, *col*).

The *row* and *col* values should be between [0,3]. If any of them are out of that range, the first element will be returned.

### 5.1.3.6 operator()() [2/2]

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

Returns a constant reference to the element at the given (*row*, *col*).

The *row* and *col* values should be between [0,3]. If any of them are out of that range, the first element will be returned.

**5.1.3.7 operator\*=( ) [1/2]**

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

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

**5.1.3.8 operator\*=( ) [2/2]**

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

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

**5.1.3.9 operator+=( )**

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

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

**5.1.3.10 operator-=( )**

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

Subtracts *m* from this 4x4 matrix stores the result in this 4x4 matrix.

**5.1.3.11 SetCol()**

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

Sets each element in the given *col* to the components of vector *v*.

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

### 5.1.3.12 SetRow()

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

Sets each element in the given *row* to the components of vector *v*.

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

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

- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h

## 5.2 FAMath::Quaternion Class Reference

A quaternion class used for quaternions and their manipulations.

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

### Public Member Functions

- [Quaternion](#) (float scalar=1.0f, float x=0.0f, float y=0.0f, float z=0.0f)  
*Constructs a quaternion with the specified values.*
- [Quaternion](#) (float scalar, const [Vector3D](#) &v)  
*Constructs a quaternion with the specified values.*
- [Quaternion](#) (const [Vector4D](#) &v)  
*Constructs a quaternion with the given values in the 4D vector v.*
- float [GetScalar](#) () const  
*Returns the scalar component of the quaternion.*
- float [GetX](#) () const  
*Returns the x value of the vector component in the quaternion.*
- float [GetY](#) () const  
*Returns the y value of the vector component in the quaternion.*
- float [GetZ](#) () const  
*Returns the z value of the vector component in the quaternion.*
- const [Vector3D](#) & [GetVector](#) () const  
*Returns the vector component of the quaternion.*
- void [SetScalar](#) (float scalar)  
*Sets the scalar component to the specified value.*
- void [SetX](#) (float x)  
*Sets the x component to the specified value.*
- void [SetY](#) (float y)  
*Sets the y component to the specified value.*
- void [SetZ](#) (float z)  
*Sets the z component to the specified value.*
- void [SetVector](#) (const [Vector3D](#) &v)  
*Sets the vector to the specified vector.*
- [Quaternion](#) & [operator+=](#) (const [Quaternion](#) &q)  
*Adds this quaternion to /a q and stores the result in this quaternion.*
- [Quaternion](#) & [operator-=](#) (const [Quaternion](#) &q)  
*Subtracts the quaternion q from this and stores the result in this quaternion.*
- [Quaternion](#) & [operator\\*=](#) (float k)  
*Multiplies this quaternion by k and stores the result in this quaternion.*
- [Quaternion](#) & [operator\\*=](#) (const [Quaternion](#) &q)  
*Multiplies this quaternion by q and stores the result in this quaternion.*

## 5.2.1 Detailed Description

A quaternion class used for quaternions and their manipulations.

The datatype for the components is float.

## 5.2.2 Constructor & Destructor Documentation

### 5.2.2.1 Quaternion() [1/3]

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

Constructs a quaternion with the specified values.

If no values are specified the identity quaternion is constructed.

### 5.2.2.2 Quaternion() [2/3]

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

Constructs a quaternion with the specified values.

### 5.2.2.3 Quaternion() [3/3]

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

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

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

## 5.2.3 Member Function Documentation



### 5.2.3.1 GetScalar()

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

Returns the scalar component of the quaternion.

### 5.2.3.2 GetVector()

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

Returns the vector component of the quaternion.

### 5.2.3.3 GetX()

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

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

### 5.2.3.4 GetY()

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

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

### 5.2.3.5 GetZ()

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

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

### 5.2.3.6 operator\*=( ) [1/2]

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

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

#### 5.2.3.7 operator\*=( ) [2/2]

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

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

#### 5.2.3.8 operator+=( )

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

Adds this quaternion to /a  $q$  and stores the result in this quaternion.

#### 5.2.3.9 operator-=( )

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

Subtracts the quaternion  $q$  from this and stores the result in this quaternion.

#### 5.2.3.10 SetScalar()

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

Sets the scalar component to the specified value.

#### 5.2.3.11 SetVector()

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

Sets the vector to the specified vector.

#### 5.2.3.12 SetX()

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

Sets the x component to the specified value.

### 5.2.3.13 SetY()

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

Sets the y component to the specified value.

### 5.2.3.14 SetZ()

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

Sets the z component to the specified value.

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

- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h

## 5.3 FAMath::Vector2D Class Reference

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

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

### Public Member Functions

- **Vector2D** (float x=0.0f, float y=0.0f)  
*Creates a new 2D vector/point with the components initialized to the arguments.*
- **Vector2D** (const **Vector3D** &v)  
*Creates a new 2D vector/point with the components initialized to the x and y values of the 3D vector.*
- **Vector2D** (const **Vector4D** &v)  
*Creates a new 2D vector/point with the components initialized to the x and y values of the 4D vector.*
- float **GetX** () const  
*Returns the x component.*
- float **GetY** () const  
*Returns the y component.*
- void **SetX** (float x)  
*Sets the x component of the vector to the specified value.*
- void **SetY** (float y)  
*Sets the y component to the specified value.*
- **Vector2D** & **operator=** (const **Vector3D** &v)  
*Sets the x and y components of this 2D vector to the x and y values of the 3D vector.*
- **Vector2D** & **operator=** (const **Vector4D** &v)  
*Sets the x and y components of this 2D vector to the x and y values of the 4D vector.*
- **Vector2D** & **operator+=** (const **Vector2D** &b)  
*Adds this vector to vector b and stores the result in this vector.*
- **Vector2D** & **operator-=** (const **Vector2D** &b)  
*Subtracts the vector b from this vector and stores the result in this vector.*
- **Vector2D** & **operator\*=** (float k)  
*Multiplies this vector by k and stores the result in this vector.*
- **Vector2D** & **operator/=** (float k)  
*Divides this vector by k and stores the result in this vector.*

### 5.3.1 Detailed Description

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

The datatype for the components is float.

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 Vector2D() [1/3]

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

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

#### 5.3.2.2 Vector2D() [2/3]

```
FAMath::Vector2D::Vector2D (
    const Vector3D & v ) [inline]
```

Creates a new 2D vector/point with the components initialized to the x and y values of the 3D vector.

#### 5.3.2.3 Vector2D() [3/3]

```
FAMath::Vector2D::Vector2D (
    const Vector4D & v ) [inline]
```

Creates a new 2D vector/point with the components initialized to the x and y values of the 4D vector.

### 5.3.3 Member Function Documentation

#### 5.3.3.1 GetX()

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

Returns the x component.

### 5.3.3.2 GetY()

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

Returns the y component.

### 5.3.3.3 operator\*=( )

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

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

### 5.3.3.4 operator+=( )

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

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

### 5.3.3.5 operator-=( )

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

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

### 5.3.3.6 operator/=( )

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

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

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

### 5.3.3.7 operator=( ) [1/2]

```
Vector2D & FAMath::Vector2D::operator= (
    const Vector3D & v ) [inline]
```

Sets the x and y components of this 2D vector to the x and y values of the 3D vector.

#### 5.3.3.8 operator=() [2/2]

```
Vector2D & FAMath::Vector2D::operator= (
    const Vector4D & v ) [inline]
```

Sets the x and y components of this 2D vector to the x and y values of the 4D vector.

#### 5.3.3.9 SetX()

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

Sets the x component of the vector to the specified value.

#### 5.3.3.10 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:

- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h↩

## 5.4 FAMath::Vector3D Class Reference

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

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

## Public Member Functions

- **Vector3D** (float x=0.0f, float y=0.0f, float z=0.0f)  
*Creates a new 3D vector/point with the components initialized to the arguments.*
- **Vector3D** (const **Vector2D** &v, float z=0.0f)  
*Creates a new 3D vector/point with the components initialized to the x and y values of the 2D vector and the specified z value;.*
- **Vector3D** (const **Vector4D** &v)  
*Creates a new 3D vector/point with the components initialized to the x, y and z values of the 4D vector.*
- 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 **Vector2D** &v)  
*Sets the x and y components of this 3D vector to the x and y values of the 2D vector and sets the z component to 0.0f.*
- **Vector3D** & **operator=** (const **Vector4D** &v)  
*Sets the x, y and z components of this 3D vector to the x, y and z values of the 4D vector.*
- **Vector3D** & **operator+=** (const **Vector3D** &b)  
*Adds this vector to vector b and stores the result in this vector.*
- **Vector3D** & **operator-=** (const **Vector3D** &b)  
*Subtracts b from this vector and stores the result in this vector.*
- **Vector3D** & **operator\*=** (float k)  
*Multiplies this vector by k and stores the result in this vector.*
- **Vector3D** & **operator/=** (float k)  
*Divides this vector by k and stores the result in this vector.*

### 5.4.1 Detailed Description

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

The datatype for the components is float.

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 Vector3D() [1/3]

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

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

#### 5.4.2.2 Vector3D() [2/3]

```
FAMath::Vector3D::Vector3D (
    const Vector2D & v,
    float z = 0.0f ) [inline]
```

Creates a new 3D vector/point with the components initialized to the x and y values of the 2D vector and the specified z value;.

#### 5.4.2.3 Vector3D() [3/3]

```
FAMath::Vector3D::Vector3D (
    const Vector4D & v ) [inline]
```

Creates a new 3D vector/point with the components initialized to the x, y and z values of the 4D vector.

### 5.4.3 Member Function Documentation

#### 5.4.3.1 GetX()

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

Returns the x component.

#### 5.4.3.2 GetY()

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

Returns y component.



#### 5.4.3.3 GetZ()

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

Returns the z component.

#### 5.4.3.4 operator\*=( )

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

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

#### 5.4.3.5 operator+=( )

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

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

#### 5.4.3.6 operator-=( )

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

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

#### 5.4.3.7 operator/=( )

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

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

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

#### 5.4.3.8 operator=( ) [1/2]

```
Vector3D & FAMath::Vector3D::operator= (
    const Vector2D & v ) [inline]
```

Sets the x and y components of this 3D vector to the x and y values of the 2D vector and sets the z component to 0.0f.

#### 5.4.3.9 operator=() [2/2]

```
Vector3D & FAMath::Vector3D::operator= (
    const Vector4D & v ) [inline]
```

Sets the x, y and z components of this 3D vector to the x, y and z values of the 4D vector.

#### 5.4.3.10 SetX()

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

Sets the x component to the specified value.

#### 5.4.3.11 SetY()

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

Sets the y component to the specified value.

#### 5.4.3.12 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:

- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h↩

## 5.5 FAMath::Vector4D Class Reference

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

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

## Public Member Functions

- [Vector4D](#) (float x=0.0f, float y=0.0f, float z=0.0f, float w=0.0f)  
*Creates a new 4D vector/point with the components initialized to the arguments.*
- [Vector4D](#) (const [Vector2D](#) &v, float z=0.0f, float w=0.0f)  
*Creates a new 4D vector/point with the components initialized to the x and y values of the 2D vector and the specified z and w values.*
- [Vector4D](#) (const [Vector3D](#) &v, float w=0.0f)  
*Creates a new 4D vector/point with the components initialized to x, y and z values of the 3D vector and the specified w value.*
- 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 [Vector2D](#) &v)  
*Sets the x and y components of this 4D vector to the x and y values of the 2D vector and sets the z and w component to 0.0f.*
- [Vector4D](#) & [operator=](#) (const [Vector3D](#) &v)  
*Sets the x, y and z components of this 4D vector to the x, y and z values of the 3D vector and sets the w component to 0.0f.*
- [Vector4D](#) & [operator+=](#) (const [Vector4D](#) &b)  
*Adds this vector to vector b and stores the result in this vector.*
- [Vector4D](#) & [operator-=](#) (const [Vector4D](#) &b)  
*Subtracts the vector b from this vector and stores the result in this vector.*
- [Vector4D](#) & [operator\\*=](#) (float k)  
*Multiplies this vector by k and stores the result in this vector.*
- [Vector4D](#) & [operator/=](#) (float k)  
*Divides this vector by k and stores the result in this vector.*

### 5.5.1 Detailed Description

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

The datatype for the components is float

### 5.5.2 Constructor & Destructor Documentation

### 5.5.2.1 Vector4D() [1/3]

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

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

### 5.5.2.2 Vector4D() [2/3]

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

Creates a new 4D vector/point with the components initialized to the x and y values of the 2D vector and the specified z and w values.

### 5.5.2.3 Vector4D() [3/3]

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

Creates a new 4D vector/point with the components initialized to x, y and z values of the 3D vector and the specified w value.

## 5.5.3 Member Function Documentation

### 5.5.3.1 GetW()

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

Returns the w component.

### 5.5.3.2 GetX()

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

Returns the x component.

### 5.5.3.3 GetY()

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

Returns the y component.

### 5.5.3.4 GetZ()

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

Returns the z component.

### 5.5.3.5 operator\*=( )

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

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

### 5.5.3.6 operator+=( )

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

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

### 5.5.3.7 operator-=( )

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

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

### 5.5.3.8 operator/=( )

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

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

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

#### 5.5.3.9 operator=() [1/2]

```
Vector4D & FAMath::Vector4D::operator= (
    const Vector2D & v ) [inline]
```

Sets the x and y components of this 4D vector to the x and y values of the 2D vector and sets the z and w component to 0.0f.

#### 5.5.3.10 operator=() [2/2]

```
Vector4D & FAMath::Vector4D::operator= (
    const Vector3D & v ) [inline]
```

Sets the x, y and z components of this 4D vector to the x, y and z values of the 3D vector and sets the w component to 0.0f.

#### 5.5.3.11 SetW()

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

Sets the w component to the specified value.

#### 5.5.3.12 SetX()

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

Sets the x component to the specified value.

#### 5.5.3.13 SetY()

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

Sets the y component to the specified value.

#### 5.5.3.14 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:

- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMath↔ Engine.h

## Chapter 6

# File Documentation

### 6.1 FAMathEngine.h

```
1 #pragma once
2
3 #include <cmath>
4
5 #if defined(_DEBUG)
6 #include <iostream>
7 #endif
8
9
10 #define EPSILON 1e-6f
11 #define PI 3.14159f
12 #define PI2 6.28319f
13
14 namespace FAMath
15 {
16     class Vector2D;
17     class Vector3D;
18     class Vector4D;
19
20     inline bool CompareFloats(float x, float y, float epsilon)
21     {
22         float diff = fabs(x - y);
23         //exact epsilon
24         if (diff < epsilon)
25         {
26             return true;
27         }
28
29         //adapative epsilon
30         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
31     }
32
33     inline bool CompareDoubles(double x, double y, double epsilon)
34     {
35         double diff = fabs(x - y);
36         //exact epsilon
37         if (diff < epsilon)
38         {
39             return true;
40         }
41
42         //adapative epsilon
43         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
44     }
45
46     //-----
47
48     class Vector2D
49     {
50     public:
51
52         Vector2D(float x = 0.0f, float y = 0.0f);
53
54         Vector2D(const Vector3D& v);
55
56         Vector2D(const Vector4D& v);
57
58     };
```

```

82     float GetX() const;
83
86     float GetY() const;
87
90     void SetX(float x);
91
94     void SetY(float y);
95
98     Vector2D& operator=(const Vector3D& v);
99
102    Vector2D& operator=(const Vector4D& v);
103
106    Vector2D& operator+=(const Vector2D& b);
107
110    Vector2D& operator-=(const Vector2D& b);
111
114    Vector2D& operator*=(float k);
115
120    Vector2D& operator/=(float k);
121
122 private:
123     float mX;
124     float mY;
125 };
126
127
128 //-----
129 //Vector2D Constructor
130
131 inline Vector2D::Vector2D(float x, float y) : mX{ x }, mY{ y }
132 {}
133 //-----
134
135 //-----
136 //Vector2D Getters and Setters
137
138 inline float Vector2D::GetX()const
139 {
140     return mX;
141 }
142
143 inline float Vector2D::GetY()const
144 {
145     return mY;
146 }
147
148 inline void Vector2D::SetX(float x)
149 {
150     mX = x;
151 }
152
153 inline void Vector2D::SetY(float y)
154 {
155     mY = y;
156 }
157
158 //-----
159
160
161 //-----
162 //Vector2D Member functions
163
164 inline Vector2D& Vector2D::operator+=(const Vector2D& b)
165 {
166     this->mX += b.mX;
167     this->mY += b.mY;
168
169     return *this;
170 }
171
172 inline Vector2D& Vector2D::operator-=(const Vector2D& b)
173 {
174     this->mX -= b.mX;
175     this->mY -= b.mY;
176
177     return *this;
178 }
179
180 inline Vector2D& Vector2D::operator*=(float k)
181 {
182     this->mX *= k;
183     this->mY *= k;
184
185     return *this;
186 }
187
188 inline Vector2D& Vector2D::operator/=(float k)

```



```

189     {
190         if (CompareFloats(k, 0.0f, EPSILON))
191         {
192             return *this;
193         }
194
195         this->mX /= k;
196         this->mY /= k;
197
198         return *this;
199     }
200
201     //-----
202     //-----
203     //Vector2D Non-member functions
204
205     inline bool ZeroVector(const Vector2D& a)
206     {
207         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON))
208         {
209             return true;
210         }
211
212         return false;
213     }
214
215     inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
216     {
217         return Vector2D(a.GetX() + b.GetX(), a.GetY() + b.GetY());
218     }
219
220     inline Vector2D operator-(const Vector2D& v)
221     {
222         return Vector2D(-v.GetX(), -v.GetY());
223     }
224
225     inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
226     {
227         return Vector2D(a.GetX() - b.GetX(), a.GetY() - b.GetY());
228     }
229
230     inline Vector2D operator*(const Vector2D& a, float k)
231     {
232         return Vector2D(a.GetX() * k, a.GetY() * k);
233     }
234
235     inline Vector2D operator*(float k, const Vector2D& a)
236     {
237         return Vector2D(k * a.GetX(), k * a.GetY());
238     }
239
240     inline Vector2D operator/(const Vector2D& a, const float& k)
241     {
242         if (CompareFloats(k, 0.0f, EPSILON))
243         {
244             return Vector2D();
245         }
246
247         return Vector2D(a.GetX() / k, a.GetY() / k);
248     }
249
250     inline float DotProduct(const Vector2D& a, const Vector2D& b)
251     {
252         return a.GetX() * b.GetX() + a.GetY() * b.GetY();
253     }
254
255     inline float Length(const Vector2D& v)
256     {
257         return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY());
258     }
259
260     inline Vector2D Norm(const Vector2D& v)
261     {
262         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
263
264         //v is the zero vector
265         if (ZeroVector(v))
266         {
267             return v;
268         }
269
270         float mag{ Length(v) };
271
272         return Vector2D(v.GetX() / mag, v.GetY() / mag);
273     }
274
275
276

```

```

302 inline Vector2D PolarToCartesian(const Vector2D& v)
303 {
304     //v = (r, theta)
305     //x = rcos(theta)
306     //y = rsin(theta)
307     float angle{ v.GetY() * PI / 180.0f };
308
309     return Vector2D(v.GetX() * cos(angle), v.GetY() * sin(angle));
310 }
311
312 inline Vector2D CartesianToPolar(const Vector2D& v)
313 {
314     //v = (x, y)
315     //r = sqrt(vx^2 + vy^2)
316     //theta = arctan(y / x)
317
318     if (CompareFloats(v.GetX(), 0.0f, EPSILON))
319     {
320         return v;
321     }
322
323     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
324     return Vector2D(Length(v), theta);
325 }
326
327 inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
328 {
329     //Projb(a) = (a dot b)b
330     //normalize b before projecting
331
332     Vector2D normB(Norm(b));
333     return Vector2D(DotProduct(a, normB) * normB);
334 }
335
336 #if defined(_DEBUG)
337 inline void print(const Vector2D& v)
338 {
339     std::cout << "(" << v.GetX() << ", " << v.GetY() << ")";
340 }
341 #endif
342 //-----
343
344 //-----
345
346 //-----
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363 class Vector3D
364 {
365 public:
366
367     Vector3D(float x = 0.0f, float y = 0.0f, float z = 0.0f);
368
369     Vector3D(const Vector2D& v, float z = 0.0f);
370
371     Vector3D(const Vector4D& v);
372
373     float GetX() const;
374
375     float GetY() const;
376
377     float GetZ() const;
378
379     void SetX(float x);
380
381     void SetY(float y);
382
383     void SetZ(float z);
384
385     Vector3D& operator=(const Vector2D& v);
386
387     Vector3D& operator=(const Vector4D& v);
388
389     Vector3D& operator+=(const Vector3D& b);
390
391     Vector3D& operator-=(const Vector3D& b);
392
393     Vector3D& operator*=(float k);
394
395     Vector3D& operator/=(float k);
396
397 private:
398     float mX;
399     float mY;

```

```

432         float mZ;
433     };
434
435     //-----
436     //Vector3D Constructors
437
438     inline Vector3D::Vector3D(float x, float y, float z) : mX{ x }, mY{ y }, mZ{ z }
439     {}
440
441     //-----
442
443     //-----
444     //Vector3D Getters and Setters
445
446     inline float Vector3D::GetX() const
447     {
448         return mX;
449     }
450
451     inline float Vector3D::GetY() const
452     {
453         return mY;
454     }
455
456     inline float Vector3D::GetZ() const
457     {
458         return mZ;
459     }
460
461     inline void Vector3D::SetX(float x)
462     {
463         mX = x;
464     }
465
466     inline void Vector3D::SetY(float y)
467     {
468         mY = y;
469     }
470
471     inline void Vector3D::SetZ(float z)
472     {
473         mZ = z;
474     }
475     //-----
476
477     //-----
478     //Vector3D Member functions
479
480     inline Vector3D& Vector3D::operator+=(const Vector3D& b)
481     {
482         this->mX += b.mX;
483         this->mY += b.mY;
484         this->mZ += b.mZ;
485
486         return *this;
487     }
488
489     inline Vector3D& Vector3D::operator-=(const Vector3D& b)
490     {
491         this->mX -= b.mX;
492         this->mY -= b.mY;
493         this->mZ -= b.mZ;
494
495         return *this;
496     }
497
498     inline Vector3D& Vector3D::operator*=(float k)
499     {
500         this->mX *= k;
501         this->mY *= k;
502         this->mZ *= k;
503
504         return *this;
505     }
506
507     inline Vector3D& Vector3D::operator/=(float k)
508     {
509         if (CompareFloats(k, 0.0f, EPSILON))
510         {
511             return *this;
512         }
513
514         this->mX /= k;
515         this->mY /= k;
516         this->mZ /= k;
517
518     }

```

```

519         return *this;
520     }
521
522     //-----
523     //-----
524     //Vector3D Non-member functions
525
526     inline bool ZeroVector(const Vector3D& a)
527     {
528         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
529             CompareFloats(a.GetZ(), 0.0f, EPSILON))
530         {
531             return true;
532         }
533         return false;
534     }
535
536     inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
537     {
538         return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
539     }
540
541     inline Vector3D operator-(const Vector3D& v)
542     {
543         return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
544     }
545
546     inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
547     {
548         return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
549     }
550
551     inline Vector3D operator*(const Vector3D& a, float k)
552     {
553         return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
554     }
555
556     inline Vector3D operator*(float k, const Vector3D& a)
557     {
558         return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
559     }
560
561     inline Vector3D operator/(const Vector3D& a, float k)
562     {
563         if (CompareFloats(k, 0.0f, EPSILON))
564         {
565             return Vector3D();
566         }
567         return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
568     }
569
570     inline float DotProduct(const Vector3D& a, const Vector3D& b)
571     {
572         //a dot b = axbx + ayby + azbz
573         return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
574     }
575
576     inline Vector3D CrossProduct(const Vector3D& a, const Vector3D& b)
577     {
578         //a x b = (aybz - azby, azbx - axbz, axby - aybx)
579         return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
580             a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
581             a.GetX() * b.GetY() - a.GetY() * b.GetX());
582     }
583
584     inline float Length(const Vector3D& v)
585     {
586         //length(v) = sqrt(vx^2 + vy^2 + vz^2)
587         return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
588     }
589
590     inline Vector3D Norm(const Vector3D& v)
591     {
592         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
593         //v is the zero vector
594         if (ZeroVector(v))
595         {
596             return v;
597         }
598         float mag{ Length(v) };
599     }
600

```

```

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

```

```

747
748 //-----
749
750
751 //-----
752
753 class Vector4D
754 {
755 public:
756     Vector4D(float x = 0.0f, float y = 0.0f, float z = 0.0f, float w = 0.0f);
757
758     Vector4D(const Vector2D& v, float z = 0.0f, float w = 0.0f);
759
760     Vector4D(const Vector3D& v, float w = 0.0f);
761
762     float GetX() const;
763
764     float GetY() const;
765
766     float GetZ() const;
767
768     float GetW() const;
769
770     void SetX(float x);
771
772     void SetY(float y);
773
774     void SetZ(float z);
775
776     void SetW(float w);
777
778     Vector4D& operator=(const Vector2D& v);
779
780     Vector4D& operator=(const Vector3D& v);
781
782     Vector4D& operator+=(const Vector4D& b);
783
784     Vector4D& operator-=(const Vector4D& b);
785
786     Vector4D& operator*=(float k);
787
788     Vector4D& operator/=(float k);
789
790 private:
791     float mX;
792     float mY;
793     float mZ;
794     float mW;
795 };
796
797 //-----
798 //Vector4D Constructors
799
800 inline Vector4D::Vector4D(float x, float y, float z, float w) : mX{ x }, mY{ y }, mZ{ z }, mW{ w }
801 {}
802
803 //-----
804
805 //-----
806 //Vector4D Getters and Setters
807
808 inline float Vector4D::GetX() const
809 {
810     return mX;
811 }
812
813 inline float Vector4D::GetY() const
814 {
815     return mY;
816 }
817
818 inline float Vector4D::GetZ() const
819 {
820     return mZ;
821 }
822
823 inline float Vector4D::GetW() const
824 {
825     return mW;
826 }
827
828 inline void Vector4D::SetX(float x)
829 {
830     mX = x;
831 }
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872

```

```

873     inline void Vector4D::SetY(float y)
874     {
875         mY = y;
876     }
877
878     inline void Vector4D::SetZ(float z)
879     {
880         mZ = z;
881     }
882
883     inline void Vector4D::SetW(float w)
884     {
885         mW = w;
886     }
887     //-----
888
889     //-----
890     //Vector4D Member functions
891
892     inline Vector4D& Vector4D::operator+=(const Vector4D& b)
893     {
894         {
895             this->mX += b.mX;
896             this->mY += b.mY;
897             this->mZ += b.mZ;
898             this->mW += b.mW;
899
900             return *this;
901         }
902
903     inline Vector4D& Vector4D::operator-=(const Vector4D& b)
904     {
905         {
906             this->mX -= b.mX;
907             this->mY -= b.mY;
908             this->mZ -= b.mZ;
909             this->mW -= b.mW;
910
911             return *this;
912         }
913
914     inline Vector4D& Vector4D::operator*=(float k)
915     {
916         {
917             this->mX *= k;
918             this->mY *= k;
919             this->mZ *= k;
920             this->mW *= k;
921
922             return *this;
923         }
924
925     inline Vector4D& Vector4D::operator/=(float k)
926     {
927         {
928             if (CompareFloats(k, 0.0f, EPSILON))
929             {
930                 return *this;
931             }
932
933             this->mX /= k;
934             this->mY /= k;
935             this->mZ /= k;
936             this->mW /= k;
937
938             return *this;
939         }
940
941     //-----
942     //-----
943     //Vector4D Non-member functions
944
945     inline bool ZeroVector(const Vector4D& a)
946     {
947         {
948             if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
949                 CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
950             {
951                 return true;
952             }
953
954             return false;
955         }
956
957     inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
958     {
959         {
960             return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
961                 b.GetW());
962         }
963     }

```

```

965     inline Vector4D operator-(const Vector4D& v)
966     {
967         return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
968     }
969
970     inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
971     {
972         return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
973             b.GetW());
974     }
975
976     inline Vector4D operator*(const Vector4D& a, float k)
977     {
978         return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
979     }
980
981     inline Vector4D operator*(float k, const Vector4D& a)
982     {
983         return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
984     }
985
986     inline Vector4D operator/(const Vector4D& a, float k)
987     {
988         if (CompareFloats(k, 0.0f, EPSILON))
989         {
990             return Vector4D();
991         }
992
993         return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
994     }
995
996     inline float DotProduct(const Vector4D& a, const Vector4D& b)
997     {
998         //a dot b = axbx + ayby + azbz + awbw
999         return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
1000     }
1001
1002     inline float Length(const Vector4D& v)
1003     {
1004         //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
1005         return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
1006             v.GetW());
1007     }
1008
1009     inline Vector4D Norm(const Vector4D& v)
1010     {
1011         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
1012         //v is the zero vector
1013         if (ZeroVector(v))
1014         {
1015             return v;
1016         }
1017
1018         float mag{ Length(v) };
1019
1020         return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
1021     }
1022
1023     inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
1024     {
1025         //Projb(a) = (a dot b)b
1026         //normalize b before projecting
1027         Vector4D normB(Norm(b));
1028         return Vector4D(DotProduct(a, normB) * normB);
1029     }
1030
1031     inline void Orthonormalize(Vector4D& x, Vector4D& y, Vector4D& z)
1032     {
1033         FAMath::Vector3D tempX(x.GetX(), x.GetY(), x.GetZ());
1034         FAMath::Vector3D tempY(y.GetX(), y.GetY(), y.GetZ());
1035         FAMath::Vector3D tempZ(z.GetX(), z.GetY(), z.GetZ());
1036
1037         tempX = Norm(tempX);
1038         tempY = Norm(CrossProduct(tempZ, tempX));
1039         tempZ = Norm(CrossProduct(tempX, tempY));
1040
1041         x = FAMath::Vector4D(tempX.GetX(), tempX.GetY(), tempX.GetZ(), 0.0f);
1042         y = FAMath::Vector4D(tempY.GetX(), tempY.GetY(), tempY.GetZ(), 0.0f);
1043         z = FAMath::Vector4D(tempZ.GetX(), tempZ.GetY(), tempZ.GetZ(), 0.0f);
1044     }
1045
1046 #if defined(_DEBUG)
1047     inline void print(const Vector4D& v)
1048     {
1049         std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ", " << v.GetW() << ")";
1050     }
1051

```



```

1076 #endif
1077 //-----
1078
1079 //-----
1080
1081
1082 //-----
1083
1090 class Matrix4x4
1091 {
1092 public:
1093
1094     Matrix4x4();
1095
1096     Matrix4x4(float a[][4]);
1097
1098     Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
1099
1100     float* Data();
1101
1102     const float* Data() const;
1103
1104     const float& operator()(unsigned int row, unsigned int col) const;
1105
1106     float& operator()(unsigned int row, unsigned int col);
1107
1108     Vector4D GetRow(unsigned int row) const;
1109
1110     Vector4D GetCol(unsigned int col) const;
1111
1112     void SetRow(unsigned int row, Vector4D v);
1113
1114     void SetCol(unsigned int col, Vector4D v);
1115
1116     Matrix4x4& operator+=(const Matrix4x4& m);
1117
1118     Matrix4x4& operator-=(const Matrix4x4& m);
1119
1120     Matrix4x4& operator*=(float k);
1121
1122     Matrix4x4& operator*=(const Matrix4x4& m);
1123
1124 private:
1125
1126     float mMat[4][4];
1127 };
1128
1129 //-----
1130
1131 inline Matrix4x4::Matrix4x4()
1132 {
1133     //1st row
1134     mMat[0][0] = 1.0f;
1135     mMat[0][1] = 0.0f;
1136     mMat[0][2] = 0.0f;
1137     mMat[0][3] = 0.0f;
1138
1139     //2nd
1140     mMat[1][0] = 0.0f;
1141     mMat[1][1] = 1.0f;
1142     mMat[1][2] = 0.0f;
1143     mMat[1][3] = 0.0f;
1144
1145     //3rd row
1146     mMat[2][0] = 0.0f;
1147     mMat[2][1] = 0.0f;
1148     mMat[2][2] = 1.0f;
1149     mMat[2][3] = 0.0f;
1150
1151     //4th row
1152     mMat[3][0] = 0.0f;
1153     mMat[3][1] = 0.0f;
1154     mMat[3][2] = 0.0f;
1155     mMat[3][3] = 1.0f;
1156 }
1157
1158 inline Matrix4x4::Matrix4x4(float a[][4])
1159 {
1160     //1st row
1161     mMat[0][0] = a[0][0];
1162     mMat[0][1] = a[0][1];
1163     mMat[0][2] = a[0][2];
1164     mMat[0][3] = a[0][3];
1165
1166     //2nd

```

```

1212         mMat[1][0] = a[1][0];
1213         mMat[1][1] = a[1][1];
1214         mMat[1][2] = a[1][2];
1215         mMat[1][3] = a[1][3];
1216
1217         //3rd row
1218         mMat[2][0] = a[2][0];
1219         mMat[2][1] = a[2][1];
1220         mMat[2][2] = a[2][2];
1221         mMat[2][3] = a[2][3];
1222
1223         //4th row
1224         mMat[3][0] = a[3][0];
1225         mMat[3][1] = a[3][1];
1226         mMat[3][2] = a[3][2];
1227         mMat[3][3] = a[3][3];
1228     }
1229
1230     inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
1231     {
1232         SetRow(0, r1);
1233         SetRow(1, r2);
1234         SetRow(2, r3);
1235         SetRow(3, r4);
1236     }
1237
1238     inline float* Matrix4x4::Data()
1239     {
1240         return mMat[0];
1241     }
1242
1243     inline const float* Matrix4x4::Data()const
1244     {
1245         return mMat[0];
1246     }
1247
1248     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
1249     {
1250         if (row > 3 || col > 3)
1251         {
1252             return mMat[0][0];
1253         }
1254         else
1255         {
1256             return mMat[row][col];
1257         }
1258     }
1259
1260     inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
1261     {
1262         if (row > 3 || col > 3)
1263         {
1264             return mMat[0][0];
1265         }
1266         else
1267         {
1268             return mMat[row][col];
1269         }
1270     }
1271
1272     inline Vector4D Matrix4x4::GetRow(unsigned int row)const
1273     {
1274         if (row < 0 || row > 3)
1275             return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
1276         else
1277             return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);
1278     }
1279
1280     inline Vector4D Matrix4x4::GetCol(unsigned int col)const
1281     {
1282         if (col < 0 || col > 3)
1283             return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
1284         else
1285             return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
1286     }
1287
1288     inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
1289     {
1290         if (row > 3)
1291         {
1292             mMat[0][0] = v.GetX();
1293             mMat[0][1] = v.GetY();
1294             mMat[0][2] = v.GetZ();
1295             mMat[0][3] = v.GetW();
1296         }
1297     }

```

```

1298         else
1299         {
1300             mMat[row][0] = v.GetX();
1301             mMat[row][1] = v.GetY();
1302             mMat[row][2] = v.GetZ();
1303             mMat[row][3] = v.GetW();
1304         }
1305     }
1306
1307     inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
1308     {
1309         if (col > 3)
1310         {
1311             mMat[0][0] = v.GetX();
1312             mMat[1][0] = v.GetY();
1313             mMat[2][0] = v.GetZ();
1314             mMat[3][0] = v.GetW();
1315         }
1316         else
1317         {
1318             mMat[0][col] = v.GetX();
1319             mMat[1][col] = v.GetY();
1320             mMat[2][col] = v.GetZ();
1321             mMat[3][col] = v.GetW();
1322         }
1323     }
1324
1325     inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
1326     {
1327         for (int i = 0; i < 4; ++i)
1328         {
1329             for (int j = 0; j < 4; ++j)
1330             {
1331                 this->mMat[i][j] += m.mMat[i][j];
1332             }
1333         }
1334
1335         return *this;
1336     }
1337
1338     inline Matrix4x4& Matrix4x4::operator--(const Matrix4x4& m)
1339     {
1340         for (int i = 0; i < 4; ++i)
1341         {
1342             for (int j = 0; j < 4; ++j)
1343             {
1344                 this->mMat[i][j] -= m.mMat[i][j];
1345             }
1346         }
1347
1348         return *this;
1349     }
1350
1351     inline Matrix4x4& Matrix4x4::operator*=(float k)
1352     {
1353         for (int i = 0; i < 4; ++i)
1354         {
1355             for (int j = 0; j < 4; ++j)
1356             {
1357                 this->mMat[i][j] *= k;
1358             }
1359         }
1360
1361         return *this;
1362     }
1363
1364     inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
1365     {
1366         Matrix4x4 res;
1367
1368         for (int i = 0; i < 4; ++i)
1369         {
1370             res.mMat[i][0] = (mMat[i][0] * m.mMat[0][0]) +
1371                 (mMat[i][1] * m.mMat[1][0]) +
1372                 (mMat[i][2] * m.mMat[2][0]) +
1373                 (mMat[i][3] * m.mMat[3][0]);
1374
1375             res.mMat[i][1] = (mMat[i][0] * m.mMat[0][1]) +
1376                 (mMat[i][1] * m.mMat[1][1]) +
1377                 (mMat[i][2] * m.mMat[2][1]) +
1378                 (mMat[i][3] * m.mMat[3][1]);
1379
1380             res.mMat[i][2] = (mMat[i][0] * m.mMat[0][2]) +
1381                 (mMat[i][1] * m.mMat[1][2]) +
1382                 (mMat[i][2] * m.mMat[2][2]) +
1383                 (mMat[i][3] * m.mMat[3][2]);
1384

```

```

1385         res.mMat[i][3] = (mMat[i][0] * m.mMat[0][3]) +
1386             (mMat[i][1] * m.mMat[1][3]) +
1387             (mMat[i][2] * m.mMat[2][3]) +
1388             (mMat[i][3] * m.mMat[3][3]);
1389     }
1390
1391     for (int i = 0; i < 4; ++i)
1392     {
1393         for (int j = 0; j < 4; ++j)
1394         {
1395             mMat[i][j] = res.mMat[i][j];
1396         }
1397     }
1398
1399     return *this;
1400 }
1401
1402 inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
1403 {
1404     Matrix4x4 res;
1405     for (int i = 0; i < 4; ++i)
1406     {
1407         for (int j = 0; j < 4; ++j)
1408         {
1409             res(i, j) = m1(i, j) + m2(i, j);
1410         }
1411     }
1412
1413     return res;
1414 }
1415
1416 inline Matrix4x4 operator-(const Matrix4x4& m)
1417 {
1418     Matrix4x4 res;
1419     for (int i = 0; i < 4; ++i)
1420     {
1421         for (int j = 0; j < 4; ++j)
1422         {
1423             res(i, j) = -m(i, j);
1424         }
1425     }
1426
1427     return res;
1428 }
1429
1430 inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
1431 {
1432     Matrix4x4 res;
1433     for (int i = 0; i < 4; ++i)
1434     {
1435         for (int j = 0; j < 4; ++j)
1436         {
1437             res(i, j) = m1(i, j) - m2(i, j);
1438         }
1439     }
1440
1441     return res;
1442 }
1443
1444 inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
1445 {
1446     Matrix4x4 res;
1447     for (int i = 0; i < 4; ++i)
1448     {
1449         for (int j = 0; j < 4; ++j)
1450         {
1451             res(i, j) = m(i, j) * k;
1452         }
1453     }
1454
1455     return res;
1456 }
1457
1458 inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
1459 {
1460     Matrix4x4 res;
1461     for (int i = 0; i < 4; ++i)
1462     {
1463         for (int j = 0; j < 4; ++j)
1464         {
1465             res(i, j) = k * m(i, j);
1466         }
1467     }
1468
1469     return res;
1470 }
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481

```

```

1486 inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
1487 {
1488     Matrix4x4 res;
1489
1490     for (int i = 0; i < 4; ++i)
1491     {
1492         res(i, 0) = (m1(i, 0) * m2(0, 0)) +
1493             (m1(i, 1) * m2(1, 0)) +
1494             (m1(i, 2) * m2(2, 0)) +
1495             (m1(i, 3) * m2(3, 0));
1496
1497         res(i, 1) = (m1(i, 0) * m2(0, 1)) +
1498             (m1(i, 1) * m2(1, 1)) +
1499             (m1(i, 2) * m2(2, 1)) +
1500             (m1(i, 3) * m2(3, 1));
1501
1502         res(i, 2) = (m1(i, 0) * m2(0, 2)) +
1503             (m1(i, 1) * m2(1, 2)) +
1504             (m1(i, 2) * m2(2, 2)) +
1505             (m1(i, 3) * m2(3, 2));
1506
1507         res(i, 3) = (m1(i, 0) * m2(0, 3)) +
1508             (m1(i, 1) * m2(1, 3)) +
1509             (m1(i, 2) * m2(2, 3)) +
1510             (m1(i, 3) * m2(3, 3));
1511     }
1512
1513     return res;
1514 }
1515
1520 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
1521 {
1522     Vector4D res;
1523
1524     res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());
1525
1526     res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());
1527
1528     res.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());
1529
1530     res.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());
1531
1532     return res;
1533 }
1534
1539 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
1540 {
1541     Vector4D res;
1542
1543     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));
1544
1545     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));
1546
1547     res.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));
1548
1549     res.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));
1550
1551     return res;
1552 }
1553
1556 inline void SetToIdentity(Matrix4x4& m)
1557 {
1558     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1559
1560     //1st row
1561     m(0, 0) = 1.0f;
1562     m(0, 1) = 0.0f;
1563     m(0, 2) = 0.0f;
1564     m(0, 3) = 0.0f;
1565
1566     //2nd row
1567     m(1, 0) = 0.0f;
1568     m(1, 1) = 1.0f;
1569     m(1, 2) = 0.0f;
1570     m(1, 3) = 0.0f;
1571
1572     //3rd row
1573     m(2, 0) = 0.0f;
1574     m(2, 1) = 0.0f;
1575     m(2, 2) = 1.0f;
1576     m(2, 3) = 0.0f;
1577
1578     //4th row
1579     m(3, 0) = 0.0f;
1580     m(3, 1) = 0.0f;
1581     m(3, 2) = 0.0f;
1582     m(3, 3) = 1.0f;

```

```

1583     }
1584
1587     inline bool IsIdentity(const Matrix4x4& m)
1588     {
1589         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
1590         0.0f
1591
1592         for (int i = 0; i < 4; ++i)
1593         {
1594             for (int j = 0; j < 4; ++j)
1595             {
1596                 if (i == j)
1597                 {
1598                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1599                         return false;
1600                 }
1601                 else
1602                 {
1603                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1604                         return false;
1605                 }
1606             }
1607         }
1608     }
1609 }
1610
1613 inline Matrix4x4 Transpose(const Matrix4x4& m)
1614 {
1615     //make the rows into cols
1616
1617     Matrix4x4 res;
1618
1619     //1st col = 1st row
1620     res(0, 0) = m(0, 0);
1621     res(1, 0) = m(0, 1);
1622     res(2, 0) = m(0, 2);
1623     res(3, 0) = m(0, 3);
1624
1625     //2nd col = 2nd row
1626     res(0, 1) = m(1, 0);
1627     res(1, 1) = m(1, 1);
1628     res(2, 1) = m(1, 2);
1629     res(3, 1) = m(1, 3);
1630
1631     //3rd col = 3rd row
1632     res(0, 2) = m(2, 0);
1633     res(1, 2) = m(2, 1);
1634     res(2, 2) = m(2, 2);
1635     res(3, 2) = m(2, 3);
1636
1637     //4th col = 4th row
1638     res(0, 3) = m(3, 0);
1639     res(1, 3) = m(3, 1);
1640     res(2, 3) = m(3, 2);
1641     res(3, 3) = m(3, 3);
1642
1643     return res;
1644 }
1645
1650 inline Matrix4x4 Translate(const Matrix4x4& cm, float x, float y, float z)
1651 {
1652     //1 0 0 0
1653     //0 1 0 0
1654     //0 0 1 0
1655     //x y z 1
1656
1657     Matrix4x4 t;
1658     t(3, 0) = x;
1659     t(3, 1) = y;
1660     t(3, 2) = z;
1661
1662     return cm * t;
1663 }
1664
1669 inline Matrix4x4 Scale(const Matrix4x4& cm, float x, float y, float z)
1670 {
1671     //x 0 0 0
1672     //0 y 0 0
1673     //0 0 z 0
1674     //0 0 0 1
1675
1676     Matrix4x4 s;
1677     s(0, 0) = x;
1678     s(1, 1) = y;
1679     s(2, 2) = z;
1680

```

```

1681         return cm * s;
1682     }
1683
1684     inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
1685     {
1686
1687         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy      0
1688         // (1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx      0
1689         // (1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2      0
1690         // 0                    0                    0                    1
1691         //c = cos(angle)
1692         //s = sin(angle)
1693
1694         float c = cos(angle * PI / 180.0f);
1695         float s = sin(angle * PI / 180.0f);
1696
1697         Matrix4x4 r;
1698
1699         //1st row
1700         r(0, 0) = c + (1.0f - c) * (x * x);
1701         r(0, 1) = (1.0f - c) * (x * y) + (s * z);
1702         r(0, 2) = (1.0f - c) * (x * z) - (s * y);
1703
1704         //2nd row
1705         r(1, 0) = (1.0f - c) * (x * y) - (s * z);
1706         r(1, 1) = c + (1.0f - c) * (y * y);
1707         r(1, 2) = (1.0f - c) * (y * z) + (s * x);
1708
1709         //3rd row
1710         r(2, 0) = (1.0f - c) * (x * z) + (s * y);
1711         r(2, 1) = (1.0f - c) * (y * z) - (s * x);
1712         r(2, 2) = c + (1.0f - c) * (z * z);
1713
1714         return cm * r;
1715     }
1716
1717     inline double Det(const Matrix4x4& m)
1718     {
1719         //m00m11(m22m33 - m23m32)
1720         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)
1721 * m(3, 2);
1722
1723         //m00m12(m23m31 - m21m33)
1724         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)
1725 * m(3, 3);
1726
1727         //m00m13(m21m32 - m22m31)
1728         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)
1729 * m(3, 1);
1730
1731         //m01m10(m22m33 - m23m32)
1732         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)
1733 * m(3, 2);
1734
1735         //m01m12(m23m30 - m20m33)
1736         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)
1737 * m(3, 3);
1738
1739         //m01m13(m20m32 - m22m30)
1740         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)
1741 * m(3, 0);
1742
1743         //m02m10(m21m33 - m23m31)
1744         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)
1745 * m(3, 1);
1746
1747         //m02m11(m23m30 - m20m33)
1748         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)
1749 * m(3, 3);
1750
1751         //m02m13(m20m31 - m21m30)
1752         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)
1753 * m(3, 0);
1754
1755         //m03m10(m21m32 - m22m21)
1756         double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
1757 2) * m(3, 1);
1758
1759         //m03m11(m22m30 - m20m32)
1760         double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
1761 0) * m(3, 2);
1762
1763         //m03m12(m20m31 - m21m30)
1764         double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
1765 1) * m(3, 0);
1766
1767         return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
1768     }

```

```

1762     }
1763
1764     inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
1765     {
1766         //cij = (-1)^i + j * det of minor(i, j);
1767         double tempMat[3][3]{};
1768         int tr{ 0 };
1769         int tc{ 0 };
1770
1771         //minor(i, j)
1772         for (int i = 0; i < 4; ++i)
1773         {
1774             if (i == row)
1775                 continue;
1776
1777             for (int j = 0; j < 4; ++j)
1778             {
1779                 if (j == col)
1780                     continue;
1781
1782                 tempMat[tr][tc] = m(i, j);
1783                 ++tc;
1784             }
1785             tc = 0;
1786             ++tr;
1787         }
1788
1789         //determinant of minor(i, j)
1790         double det3x3 = (tempMat[0][0] * tempMat[1][1] * tempMat[2][2]) + (tempMat[0][1] *
1791 tempMat[1][2] * tempMat[2][0]) +
1792 (tempMat[0][2] * tempMat[1][0] * tempMat[2][1]) - (tempMat[0][2] * tempMat[1][1] *
1793 tempMat[2][0]) -
1794 (tempMat[0][1] * tempMat[1][0] * tempMat[2][2]) - (tempMat[0][0] * tempMat[1][2] *
1795 tempMat[2][1]);
1796
1797         return pow(-1, row + col) * det3x3;
1798     }
1799
1800     inline Matrix4x4 Adjoint(const Matrix4x4& m)
1801     {
1802         //Cofactor of each ijth position put into matrix cA.
1803         //Adjoint is the tranposed matrix of cA.
1804         Matrix4x4 cA;
1805         for (int i = 0; i < 4; ++i)
1806         {
1807             for (int j = 0; j < 4; ++j)
1808             {
1809                 cA(i, j) = static_cast<float>(Cofactor(m, i, j));
1810             }
1811         }
1812
1813         return Transpose(cA);
1814     }
1815
1816     inline Matrix4x4 Inverse(const Matrix4x4& m)
1817     {
1818         //Inverse of m = adjoint of m / det of m
1819         double determinant = Det(m);
1820         if (CompareDoubles(determinant, 0.0, EPSILON))
1821             return Matrix4x4();
1822
1823         return Adjoint(m) * (1.0f / static_cast<float>(determinant));
1824     }
1825
1826 #if defined(_DEBUG)
1827     inline void print(const Matrix4x4& m)
1828     {
1829         for (int i = 0; i < 4; ++i)
1830         {
1831             for (int j = 0; j < 4; ++j)
1832             {
1833                 std::cout << m(i, j) << " ";
1834             }
1835
1836             std::cout << std::endl;
1837         }
1838     }
1839 #endif
1840
1841 //-----
1842
1843
1844
1845
1846
1847
1848

```



```

1853 //-----
1867 class Quaternion
1868 {
1869 public:
1874     Quaternion(float scalar = 1.0f, float x = 0.0f, float y = 0.0f, float z = 0.0f);
1875
1878     Quaternion(float scalar, const Vector3D& v);
1879
1885     Quaternion(const Vector4D& v);
1886
1889     float GetScalar() const;
1890
1893     float GetX() const;
1894
1897     float GetY() const;
1898
1901     float GetZ() const;
1902
1905     const Vector3D& GetVector() const;
1906
1909     void SetScalar(float scalar);
1910
1913     void SetX(float x);
1914
1917     void SetY(float y);
1918
1921     void SetZ(float z);
1922
1925     void SetVector(const Vector3D& v);
1926
1929     Quaternion& operator+=(const Quaternion& q);
1930
1933     Quaternion& operator-=(const Quaternion& q);
1934
1937     Quaternion& operator*=(float k);
1938
1941     Quaternion& operator*=(const Quaternion& q);
1942
1943 private:
1944     float mScalar;
1945     float mX;
1946     float mY;
1947     float mZ;
1948 };
1949
1950 //-----
1951 inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
1952     mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
1953 {
1954 }
1955
1956 inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
1957     mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
1958 {
1959 }
1960
1961 inline Quaternion::Quaternion(const Vector4D& v) :
1962     mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
1963 {
1964 }
1965
1966 inline float Quaternion::GetScalar()const
1967 {
1968     return mScalar;
1969 }
1970
1971 inline float Quaternion::GetX()const
1972 {
1973     return mX;
1974 }
1975
1976 inline float Quaternion::GetY()const
1977 {
1978     return mY;
1979 }
1980
1981 inline float Quaternion::GetZ()const
1982 {
1983     return mZ;
1984 }
1985
1986 inline const Vector3D& Quaternion::GetVector()const
1987 {
1988     return Vector3D(mX, mY, mZ);
1989 }
1990

```

```

1991     inline void Quaternion::SetScalar(float scalar)
1992     {
1993         mScalar = scalar;
1994     }
1995
1996     inline void Quaternion::SetX(float x)
1997     {
1998         mX = x;
1999     }
2000
2001     inline void Quaternion::SetY(float y)
2002     {
2003         mY = y;
2004     }
2005
2006     inline void Quaternion::SetZ(float z)
2007     {
2008         mZ = z;
2009     }
2010
2011     inline void Quaternion::SetVector(const Vector3D& v)
2012     {
2013         mX = v.GetX();
2014         mY = v.GetY();
2015         mZ = v.GetZ();
2016     }
2017
2018     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
2019     {
2020         this->mScalar += q.mScalar;
2021         this->mX += q.mX;
2022         this->mY += q.mY;
2023         this->mZ += q.mZ;
2024
2025         return *this;
2026     }
2027
2028     inline Quaternion& Quaternion::operator-=(const Quaternion& q)
2029     {
2030         this->mScalar -= q.mScalar;
2031         this->mX -= q.mX;
2032         this->mY -= q.mY;
2033         this->mZ -= q.mZ;
2034
2035         return *this;
2036     }
2037
2038     inline Quaternion& Quaternion::operator*=(float k)
2039     {
2040         this->mScalar *= k;
2041         this->mX *= k;
2042         this->mY *= k;
2043         this->mZ *= k;
2044
2045         return *this;
2046     }
2047
2048     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
2049     {
2050         Vector3D thisVector(this->mX, this->mY, this->mZ);
2051         Vector3D qVector(q.mX, q.mY, q.mZ);
2052
2053         float s{ this->mScalar * q.mScalar };
2054         float dP{ DotProduct(thisVector, qVector) };
2055         float resultScalar{ s - dP };
2056
2057         Vector3D a(this->mScalar * qVector);
2058         Vector3D b(q.mScalar * thisVector);
2059         Vector3D cP(CrossProduct(thisVector, qVector));
2060         Vector3D resultVector(a + b + cP);
2061
2062         this->mScalar = resultScalar;
2063         this->mX = resultVector.GetX();
2064         this->mY = resultVector.GetY();
2065         this->mZ = resultVector.GetZ();
2066
2067         return *this;
2068     }
2069
2070     inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
2071     {
2072         return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
2073             q2.GetY(), q1.GetZ() + q2.GetZ());
2074     }
2075
2076     inline Quaternion operator-(const Quaternion& q)
2077     {
2078

```

```

2081     return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2082 }
2083
2086 inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
2087 {
2088     return Quaternion(q1.GetScalar() - q2.GetScalar(),
2089         q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
2090 }
2091
2094 inline Quaternion operator*(float k, const Quaternion& q)
2095 {
2096     return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
2097 }
2098
2101 inline Quaternion operator*(const Quaternion& q, float k)
2102 {
2103     return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
2104 }
2105
2108 inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
2109 {
2110     //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
2111     //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
2112
2113     Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
2114     Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());
2115
2116     float s{ q1.GetScalar() * q2.GetScalar() };
2117     float dP{ DotProduct(q1Vector, q2Vector) };
2118     float resultScalar{ s - dP };
2119
2120     Vector3D a(q1.GetScalar() * q2Vector);
2121     Vector3D b(q2.GetScalar() * q1Vector);
2122     Vector3D cP(CrossProduct(q1Vector, q2Vector));
2123     Vector3D resultVector(a + b + cP);
2124
2125     return Quaternion(resultScalar, resultVector);
2126 }
2127
2130 inline bool IsZeroQuaternion(const Quaternion& q)
2131 {
2132     //zero quaternion = (0, 0, 0, 0)
2133     return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2134         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2135 }
2136
2139 inline bool IsIdentity(const Quaternion& q)
2140 {
2141     //identity quaternion = (1, 0, 0, 0)
2142     return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
2143         CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
2144 }
2145
2148 inline Quaternion Conjugate(const Quaternion& q)
2149 {
2150     //conjugate of a quaternion is the quaternion with its vector part negated
2151     return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
2152 }
2153
2156 inline float Length(const Quaternion& q)
2157 {
2158     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
2159     return sqrt(q.GetScalar() * q.GetScalar() + q.GetX() * q.GetX() + q.GetY() * q.GetY() +
2160         q.GetZ() * q.GetZ());
2161 }
2162
2166 inline Quaternion Normalize(const Quaternion& q)
2167 {
2168     //to normalize a quaternion you do q / |q|
2169
2170     if (IsZeroQuaternion(q))
2171         return q;
2172
2173     float d{ Length(q) };
2174
2175     return Quaternion(q.GetScalar() / d, q.GetX() / d, q.GetY() / d, q.GetZ() / d);
2176 }
2177
2182 inline Quaternion Inverse(const Quaternion& q)
2183 {
2184     //inverse = conjugate of q / |q|^2
2185
2186     if (IsZeroQuaternion(q))
2187         return q;
2188
2189     Quaternion conjugateOfQ(Conjugate(q));
2190

```

```

2191         float d{ Length(q) };
2192         d *= d;
2193
2194         return Quaternion(conjugateOfQ.GetScalar() / d, conjugateOfQ.GetX() / d,
2195             conjugateOfQ.GetY() / d, conjugateOfQ.GetZ() / d);
2196     }
2197
2198     inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
2199     {
2200         //A roatation quaternion is a quaternion where the
2201         //scalar part = cos(theta / 2)
2202         //vector part = sin(theta / 2) * axis
2203         //the axis needs to be normalized
2204
2205         float ang{ angle / 2.0f };
2206         float c{ cos(ang * PI / 180.0f) };
2207         float s{ sin(ang * PI / 180.0f) };
2208
2209         Vector3D axis(x, y, z);
2210         axis = Norm(axis);
2211
2212         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2213     }
2214
2215     inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
2216     {
2217         //A roatation quaternion is a quaternion where the
2218         //scalar part = cos(theta / 2)
2219         //vector part = sin(theta / 2) * axis
2220         //the axis needs to be normalized
2221
2222         float ang{ angle / 2.0f };
2223         float c{ cos(ang * PI / 180.0f) };
2224         float s{ sin(ang * PI / 180.0f) };
2225
2226         Vector3D axisN(Norm(axis));
2227
2228         return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
2229     }
2230
2231     inline Quaternion RotationQuaternion(const Vector4D& angAxis)
2232     {
2233         //A roatation quaternion is a quaternion where the
2234         //scalar part = cos(theta / 2)
2235         //vector part = sin(theta / 2) * axis
2236         //the axis needs to be normalized
2237
2238         float angle{ angAxis.GetX() / 2.0f };
2239         float c{ cos(angle * PI / 180.0f) };
2240         float s{ sin(angle * PI / 180.0f) };
2241
2242         Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
2243         axis = Norm(axis);
2244
2245         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
2246     }
2247
2248     inline Matrix4x4 QuaternionToRotationMatrixCol(const Quaternion& q)
2249     {
2250         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
2251         //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
2252         //2q2q4 - 2q1q3          2q3q4 + 2q1q2          1 - 2q2^2 - 2q3^2      0
2253         //0                      0                      0                      1
2254         //q1 = scalar
2255         //q2 = x
2256         //q3 = y
2257         //q4 = z
2258
2259         float colMat[4][4] = {};
2260
2261         colMat[0][0] = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
2262         colMat[0][1] = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
2263         colMat[0][2] = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
2264         colMat[0][3] = 0.0f;
2265
2266         colMat[1][0] = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
2267         colMat[1][1] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
2268         colMat[1][2] = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
2269         colMat[1][3] = 0.0f;
2270
2271         colMat[2][0] = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
2272         colMat[2][1] = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
2273         colMat[2][2] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
2274         colMat[2][3] = 0.0f;
2275
2276         colMat[3][0] = 0.0f;
2277         colMat[3][1] = 0.0f;

```

```

2295         colMat[3][2] = 0.0f;
2296         colMat[3][3] = 1.0f;
2297
2298         return Matrix4x4(colMat);
2299     }
2300
2301     inline Matrix4x4 QuaternionToRotationMatrixRow(const Quaternion& q)
2302     {
2303         //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
2304         //2q2q3 - 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
2305         //2q2q4 + 2q1q3      2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2      0
2306         //0                  0                  0                  1
2307         //q1 = scalar
2308         //q2 = x
2309         //q3 = y
2310         //q4 = z
2311
2312         float rowMat[4][4] = {};
2313
2314         rowMat[0][0] = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
2315         rowMat[0][1] = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
2316         rowMat[0][2] = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
2317         rowMat[0][3] = 0.0f;
2318
2319         rowMat[1][0] = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
2320         rowMat[1][1] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
2321         rowMat[1][2] = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
2322         rowMat[1][3] = 0.0f;
2323
2324         rowMat[2][0] = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
2325         rowMat[2][1] = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
2326         rowMat[2][2] = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
2327         rowMat[2][3] = 0.0f;
2328
2329         rowMat[3][0] = 0.0f;
2330         rowMat[3][1] = 0.0f;
2331         rowMat[3][2] = 0.0f;
2332         rowMat[3][3] = 1.0f;
2333
2334         return Matrix4x4(rowMat);
2335     }
2336
2337     #if defined(_DEBUG)
2338     inline void print(const Quaternion& q)
2339     {
2340         std::cout << "(" << q.GetScalar() << ", " << q.GetX() << ", " << q.GetY() << ", " << q.GetZ();
2341     }
2342     #endif
2343     //-----
2344
2345     inline Vector2D::Vector2D(const Vector3D& v) : mX{ v.GetX() }, mY{ v.GetY() }
2346     {}
2347
2348     inline Vector2D::Vector2D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }
2349     {}
2350
2351     inline Vector3D::Vector3D(const Vector2D& v, float z) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ z }
2352     {}
2353
2354     inline Vector3D::Vector3D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
2355     {}
2356
2357     inline Vector4D::Vector4D(const Vector2D& v, float z, float w) : mX{ v.GetX() }, mY{ v.GetY() },
2358     mZ{ z }, mW{ w }
2359     {}
2360
2361     inline Vector4D::Vector4D(const Vector3D& v, float w) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{
2362     v.GetZ() }, mW{ w }
2363     {}
2364
2365     inline Vector2D& Vector2D::operator=(const Vector3D& v)
2366     {
2367         mX = v.GetX();
2368         mY = v.GetY();
2369     }
2370
2371     inline Vector2D& Vector2D::operator=(const Vector4D& v)
2372     {
2373         mX = v.GetX();
2374         mY = v.GetY();
2375     }
2376
2377     inline Vector3D& Vector3D::operator=(const Vector2D& v)
2378     {
2379         mX = v.GetX();
2380         mY = v.GetY();
2381         mZ = 0.0f;
2382     }

```

```
2384     }
2385
2386     inline Vector3D& Vector3D::operator=(const Vector4D& v)
2387     {
2388         mX = v.GetX();
2389         mY = v.GetY();
2390         mZ = v.GetZ();
2391     }
2392
2393     inline Vector4D& Vector4D::operator=(const Vector2D& v)
2394     {
2395         mX = v.GetX();
2396         mY = v.GetY();
2397         mZ = 0.0f;
2398         mW = 0.0f;
2399     }
2400
2401     inline Vector4D& Vector4D::operator=(const Vector3D& v)
2402     {
2403         mX = v.GetX();
2404         mY = v.GetY();
2405         mZ = v.GetZ();
2406         mW = 0.0f;
2407     }
2408
2409 //-----
2409 }
```

# Index

- Adjoint
  - FAMath, [11](#)
- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h, [47](#)
- CartesianToCylindrical
  - FAMath, [11](#)
- CartesianToPolar
  - FAMath, [11](#)
- CartesianToSpherical
  - FAMath, [11](#)
- Cofactor
  - FAMath, [11](#)
- CompareDoubles
  - FAMath, [12](#)
- CompareFloats
  - FAMath, [12](#)
- Conjugate
  - FAMath, [12](#)
- CrossProduct
  - FAMath, [12](#)
- CylindricalToCartesian
  - FAMath, [12](#)
- Data
  - FAMath::Matrix4x4, [28](#)
- Det
  - FAMath, [13](#)
- DotProduct
  - FAMath, [13](#)
- FAMath, [7](#)
  - Adjoint, [11](#)
  - CartesianToCylindrical, [11](#)
  - CartesianToPolar, [11](#)
  - CartesianToSpherical, [11](#)
  - Cofactor, [11](#)
  - CompareDoubles, [12](#)
  - CompareFloats, [12](#)
  - Conjugate, [12](#)
  - CrossProduct, [12](#)
  - CylindricalToCartesian, [12](#)
  - Det, [13](#)
  - DotProduct, [13](#)
  - Inverse, [13](#), [14](#)
  - IsIdentity, [14](#)
  - IsZeroQuaternion, [14](#)
  - Length, [14](#), [15](#)
  - Norm, [15](#)
  - Normalize, [15](#)
  - operator\*, [16–18](#)
  - operator+, [18](#), [19](#)
  - operator-, [19–21](#)
  - operator/, [21](#), [22](#)
  - Orthonormalize, [22](#)
  - PolarToCartesian, [22](#)
  - Projection, [23](#)
  - QuaternionToRotationMatrixCol, [23](#)
  - QuaternionToRotationMatrixRow, [23](#)
  - Rotate, [24](#)
  - RotationQuaternion, [24](#)
  - Scale, [24](#)
  - SetToIdentity, [25](#)
  - SphericalToCartesian, [25](#)
  - Translate, [25](#)
  - Transpose, [25](#)
  - ZeroVector, [25](#), [26](#)
- FAMath::Matrix4x4, [27](#)
  - Data, [28](#)
  - GetCol, [29](#)
  - GetRow, [29](#)
  - Matrix4x4, [28](#)
  - operator\*=, [29](#), [30](#)
  - operator(), [29](#)
  - operator+=, [30](#)
  - operator-=, [30](#)
  - SetCol, [30](#)
  - SetRow, [30](#)
- FAMath::Quaternion, [31](#)
  - GetScalar, [32](#)
  - GetVector, [33](#)
  - GetX, [33](#)
  - GetY, [33](#)
  - GetZ, [33](#)
  - operator\*=, [33](#)
  - operator+=, [34](#)
  - operator-=, [34](#)
  - Quaternion, [32](#)
  - SetScalar, [34](#)
  - SetVector, [34](#)
  - SetX, [34](#)
  - SetY, [34](#)
  - SetZ, [35](#)
- FAMath::Vector2D, [35](#)
  - GetX, [36](#)
  - GetY, [36](#)
  - operator\*=, [37](#)
  - operator+=, [37](#)

- operator-=, 37
- operator/=: 37
- operator=: 37
- SetX, 38
- SetY, 38
- Vector2D, 36
- FAMath::Vector3D, 38
  - GetX, 40
  - GetY, 40
  - GetZ, 40
  - operator\*=, 41
  - operator+=, 41
  - operator-=, 41
  - operator/=: 41
  - operator=: 41
  - SetX, 42
  - SetY, 42
  - SetZ, 42
  - Vector3D, 39, 40
- FAMath::Vector4D, 42
  - GetW, 44
  - GetX, 44
  - GetY, 44
  - GetZ, 45
  - operator\*=, 45
  - operator+=, 45
  - operator-=, 45
  - operator/=: 45
  - operator=: 45, 46
  - SetW, 46
  - SetX, 46
  - SetY, 46
  - SetZ, 46
  - Vector4D, 43, 44
- GetCol
  - FAMath::Matrix4x4, 29
- GetRow
  - FAMath::Matrix4x4, 29
- GetScalar
  - FAMath::Quaternion, 32
- GetVector
  - FAMath::Quaternion, 33
- GetW
  - FAMath::Vector4D, 44
- GetX
  - FAMath::Quaternion, 33
  - FAMath::Vector2D, 36
  - FAMath::Vector3D, 40
  - FAMath::Vector4D, 44
- GetY
  - FAMath::Quaternion, 33
  - FAMath::Vector2D, 36
  - FAMath::Vector3D, 40
  - FAMath::Vector4D, 44
- GetZ
  - FAMath::Quaternion, 33
  - FAMath::Vector3D, 40
  - FAMath::Vector4D, 45
- Inverse
  - FAMath, 13, 14
- IsIdentity
  - FAMath, 14
- IsZeroQuaternion
  - FAMath, 14
- Length
  - FAMath, 14, 15
- Matrix4x4
  - FAMath::Matrix4x4, 28
- Norm
  - FAMath, 15
- Normalize
  - FAMath, 15
- operator\*
  - FAMath, 16–18
- operator\*=
  - FAMath::Matrix4x4, 29, 30
  - FAMath::Quaternion, 33
  - FAMath::Vector2D, 37
  - FAMath::Vector3D, 41
  - FAMath::Vector4D, 45
- operator()
  - FAMath::Matrix4x4, 29
- operator+
  - FAMath, 18, 19
- operator+=
  - FAMath::Matrix4x4, 30
  - FAMath::Quaternion, 34
  - FAMath::Vector2D, 37
  - FAMath::Vector3D, 41
  - FAMath::Vector4D, 45
- operator-
  - FAMath, 19–21
- operator-=
  - FAMath::Matrix4x4, 30
  - FAMath::Quaternion, 34
  - FAMath::Vector2D, 37
  - FAMath::Vector3D, 41
  - FAMath::Vector4D, 45
- operator/
  - FAMath, 21, 22
- operator/=
  - FAMath::Vector2D, 37
  - FAMath::Vector3D, 41
  - FAMath::Vector4D, 45
- operator=
  - FAMath::Vector2D, 37
  - FAMath::Vector3D, 41
  - FAMath::Vector4D, 45, 46
- Orthonormalize
  - FAMath, 22
- PolarToCartesian
  - FAMath, 22



Projection  
    FAMath, [23](#)

Quaternion  
    FAMath::Quaternion, [32](#)

QuaternionToRotationMatrixCol  
    FAMath, [23](#)

QuaternionToRotationMatrixRow  
    FAMath, [23](#)

Rotate  
    FAMath, [24](#)

RotationQuaternion  
    FAMath, [24](#)

Scale  
    FAMath, [24](#)

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

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

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

SetTolIdentity  
    FAMath, [25](#)

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

SetW  
    FAMath::Vector4D, [46](#)

SetX  
    FAMath::Quaternion, [34](#)  
    FAMath::Vector2D, [38](#)  
    FAMath::Vector3D, [42](#)  
    FAMath::Vector4D, [46](#)

SetY  
    FAMath::Quaternion, [34](#)  
    FAMath::Vector2D, [38](#)  
    FAMath::Vector3D, [42](#)  
    FAMath::Vector4D, [46](#)

SetZ  
    FAMath::Quaternion, [35](#)  
    FAMath::Vector3D, [42](#)  
    FAMath::Vector4D, [46](#)

SphericalToCartesian  
    FAMath, [25](#)

Translate  
    FAMath, [25](#)

Transpose  
    FAMath, [25](#)

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

Vector3D  
    FAMath::Vector3D, [39](#), [40](#)

Vector4D  
    FAMath::Vector4D, [43](#), [44](#)

ZeroVector  
    FAMath, [25](#), [26](#)