

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        | 12       |
| 4.1.2 Function Documentation      | 12       |
| 4.1.2.1 Adjoint() [1/3]           | 12       |
| 4.1.2.2 Adjoint() [2/3]           | 13       |
| 4.1.2.3 Adjoint() [3/3]           | 13       |
| 4.1.2.4 CartesianToCylindrical()  | 13       |
| 4.1.2.5 CartesianToPolar()        | 13       |
| 4.1.2.6 CartesianToSpherical()    | 13       |
| 4.1.2.7 Cofactor() [1/3]          | 14       |
| 4.1.2.8 Cofactor() [2/3]          | 14       |
| 4.1.2.9 Cofactor() [3/3]          | 14       |
| 4.1.2.10 CompareDoubles()         | 14       |
| 4.1.2.11 CompareFloats()          | 14       |
| 4.1.2.12 Conjugate()              | 15       |
| 4.1.2.13 CrossProduct()           | 15       |
| 4.1.2.14 CylindricalToCartesian() | 15       |
| 4.1.2.15 Determinant() [1/3]      | 15       |
| 4.1.2.16 Determinant() [2/3]      | 15       |
| 4.1.2.17 Determinant() [3/3]      | 16       |
| 4.1.2.18 DotProduct() [1/3]       | 16       |
| 4.1.2.19 DotProduct() [2/3]       | 16       |
| 4.1.2.20 DotProduct() [3/3]       | 16       |
| 4.1.2.21 Inverse() [1/4]          | 16       |
| 4.1.2.22 Inverse() [2/4]          | 17       |
| 4.1.2.23 Inverse() [3/4]          | 17       |
| 4.1.2.24 Inverse() [4/4]          | 17       |
| 4.1.2.25 IsIdentity() [1/4]       | 17       |
| 4.1.2.26 IsIdentity() [2/4]       | 17       |
| 4.1.2.27 IsIdentity() [3/4]       | 17       |
| 4.1.2.28 IsIdentity() [4/4]       | 18       |
| 4.1.2.29 IsZeroQuaternion()       | 18       |
| 4.1.2.30 Length() [1/4]           | 18       |

|                              |    |
|------------------------------|----|
| 4.1.2.31 Length() [2/4]      | 18 |
| 4.1.2.32 Length() [3/4]      | 18 |
| 4.1.2.33 Length() [4/4]      | 18 |
| 4.1.2.34 Norm() [1/3]        | 19 |
| 4.1.2.35 Norm() [2/3]        | 19 |
| 4.1.2.36 Norm() [3/3]        | 19 |
| 4.1.2.37 Normalize()         | 19 |
| 4.1.2.38 operator*() [1/24]  | 19 |
| 4.1.2.39 operator*() [2/24]  | 20 |
| 4.1.2.40 operator*() [3/24]  | 20 |
| 4.1.2.41 operator*() [4/24]  | 20 |
| 4.1.2.42 operator*() [5/24]  | 20 |
| 4.1.2.43 operator*() [6/24]  | 20 |
| 4.1.2.44 operator*() [7/24]  | 21 |
| 4.1.2.45 operator*() [8/24]  | 21 |
| 4.1.2.46 operator*() [9/24]  | 21 |
| 4.1.2.47 operator*() [10/24] | 21 |
| 4.1.2.48 operator*() [11/24] | 21 |
| 4.1.2.49 operator*() [12/24] | 22 |
| 4.1.2.50 operator*() [13/24] | 22 |
| 4.1.2.51 operator*() [14/24] | 22 |
| 4.1.2.52 operator*() [15/24] | 22 |
| 4.1.2.53 operator*() [16/24] | 22 |
| 4.1.2.54 operator*() [17/24] | 23 |
| 4.1.2.55 operator*() [18/24] | 23 |
| 4.1.2.56 operator*() [19/24] | 23 |
| 4.1.2.57 operator*() [20/24] | 23 |
| 4.1.2.58 operator*() [21/24] | 23 |
| 4.1.2.59 operator*() [22/24] | 24 |
| 4.1.2.60 operator*() [23/24] | 24 |
| 4.1.2.61 operator*() [24/24] | 24 |
| 4.1.2.62 operator+() [1/7]   | 24 |
| 4.1.2.63 operator+() [2/7]   | 24 |
| 4.1.2.64 operator+() [3/7]   | 25 |
| 4.1.2.65 operator+() [4/7]   | 25 |
| 4.1.2.66 operator+() [5/7]   | 25 |
| 4.1.2.67 operator+() [6/7]   | 25 |
| 4.1.2.68 operator+() [7/7]   | 25 |
| 4.1.2.69 operator-() [1/14]  | 26 |
| 4.1.2.70 operator-() [2/14]  | 26 |
| 4.1.2.71 operator-() [3/14]  | 26 |
| 4.1.2.72 operator-() [4/14]  | 26 |

|  |    |
|--|----|
| 4.1.2.73 operator-() [5/14]              | 26 |
| 4.1.2.74 operator-() [6/14]              | 27 |
| 4.1.2.75 operator-() [7/14]              | 27 |
| 4.1.2.76 operator-() [8/14]              | 27 |
| 4.1.2.77 operator-() [9/14]              | 27 |
| 4.1.2.78 operator-() [10/14]             | 27 |
| 4.1.2.79 operator-() [11/14]             | 28 |
| 4.1.2.80 operator-() [12/14]             | 28 |
| 4.1.2.81 operator-() [13/14]             | 28 |
| 4.1.2.82 operator-() [14/14]             | 28 |
| 4.1.2.83 operator/() [1/3]               | 28 |
| 4.1.2.84 operator/() [2/3]               | 29 |
| 4.1.2.85 operator/() [3/3]               | 29 |
| 4.1.2.86 Orthonormalize() [1/2]          | 29 |
| 4.1.2.87 Orthonormalize() [2/2]          | 29 |
| 4.1.2.88 PolarToCartesian()              | 29 |
| 4.1.2.89 Projection() [1/3]              | 30 |
| 4.1.2.90 Projection() [2/3]              | 30 |
| 4.1.2.91 Projection() [3/3]              | 30 |
| 4.1.2.92 QuaternionToRotationMatrixCol() | 30 |
| 4.1.2.93 QuaternionToRotationMatrixRow() | 30 |
| 4.1.2.94 Rotate() [1/7]                  | 31 |
| 4.1.2.95 Rotate() [2/7]                  | 31 |
| 4.1.2.96 Rotate() [3/7]                  | 31 |
| 4.1.2.97 Rotate() [4/7]                  | 31 |
| 4.1.2.98 Rotate() [5/7]                  | 31 |
| 4.1.2.99 Rotate() [6/7]                  | 32 |
| 4.1.2.100 Rotate() [7/7]                 | 32 |
| 4.1.2.101 RotationQuaternion() [1/3]     | 32 |
| 4.1.2.102 RotationQuaternion() [2/3]     | 32 |
| 4.1.2.103 RotationQuaternion() [3/3]     | 32 |
| 4.1.2.104 Scale() [1/6]                  | 33 |
| 4.1.2.105 Scale() [2/6]                  | 33 |
| 4.1.2.106 Scale() [3/6]                  | 33 |
| 4.1.2.107 Scale() [4/6]                  | 33 |
| 4.1.2.108 Scale() [5/6]                  | 33 |
| 4.1.2.109 Scale() [6/6]                  | 34 |
| 4.1.2.110 SetTolIdentity() [1/3]         | 34 |
| 4.1.2.111 SetTolIdentity() [2/3]         | 34 |
| 4.1.2.112 SetTolIdentity() [3/3]         | 34 |
| 4.1.2.113 SphericalToCartesian()         | 34 |
| 4.1.2.114 Translate() [1/2]              | 35 |

|  |           |
|--|-----------|
| 4.1.2.115 Translate() [2/2] . . . . .                  | 35        |
| 4.1.2.116 Transpose() [1/3] . . . . .                  | 35        |
| 4.1.2.117 Transpose() [2/3] . . . . .                  | 35        |
| 4.1.2.118 Transpose() [3/3] . . . . .                  | 35        |
| 4.1.2.119 ZeroVector() [1/3] . . . . .                 | 36        |
| 4.1.2.120 ZeroVector() [2/3] . . . . .                 | 36        |
| 4.1.2.121 ZeroVector() [3/3] . . . . .                 | 36        |
| <b>5 Class Documentation</b>                           | <b>37</b> |
| 5.1 FAMath::Matrix2x2 Class Reference . . . . .        | 37        |
| 5.1.1 Detailed Description . . . . .                   | 38        |
| 5.1.2 Constructor & Destructor Documentation . . . . . | 38        |
| 5.1.2.1 Matrix2x2() [1/5] . . . . .                    | 38        |
| 5.1.2.2 Matrix2x2() [2/5] . . . . .                    | 38        |
| 5.1.2.3 Matrix2x2() [3/5] . . . . .                    | 38        |
| 5.1.2.4 Matrix2x2() [4/5] . . . . .                    | 39        |
| 5.1.2.5 Matrix2x2() [5/5] . . . . .                    | 39        |
| 5.1.3 Member Function Documentation . . . . .          | 39        |
| 5.1.3.1 Data() [1/2] . . . . .                         | 39        |
| 5.1.3.2 Data() [2/2] . . . . .                         | 39        |
| 5.1.3.3 GetCol() . . . . .                             | 39        |
| 5.1.3.4 GetRow() . . . . .                             | 40        |
| 5.1.3.5 operator()() [1/2] . . . . .                   | 40        |
| 5.1.3.6 operator()() [2/2] . . . . .                   | 40        |
| 5.1.3.7 operator*=( ) [1/2] . . . . .                  | 40        |
| 5.1.3.8 operator*=( ) [2/2] . . . . .                  | 40        |
| 5.1.3.9 operator+=( ) . . . . .                        | 41        |
| 5.1.3.10 operator-=( ) . . . . .                       | 41        |
| 5.1.3.11 operator=( ) [1/2] . . . . .                  | 41        |
| 5.1.3.12 operator=( ) [2/2] . . . . .                  | 41        |
| 5.1.3.13 SetCol() . . . . .                            | 41        |
| 5.1.3.14 SetRow() . . . . .                            | 42        |
| 5.2 FAMath::Matrix3x3 Class Reference . . . . .        | 42        |
| 5.2.1 Detailed Description . . . . .                   | 43        |
| 5.2.2 Constructor & Destructor Documentation . . . . . | 43        |
| 5.2.2.1 Matrix3x3() [1/5] . . . . .                    | 43        |
| 5.2.2.2 Matrix3x3() [2/5] . . . . .                    | 43        |
| 5.2.2.3 Matrix3x3() [3/5] . . . . .                    | 43        |
| 5.2.2.4 Matrix3x3() [4/5] . . . . .                    | 44        |
| 5.2.2.5 Matrix3x3() [5/5] . . . . .                    | 44        |
| 5.2.3 Member Function Documentation . . . . .          | 44        |
| 5.2.3.1 Data() [1/2] . . . . .                         | 44        |

|  |    |
|--|----|
| 5.2.3.2 Data() [2/2]                         | 44 |
| 5.2.3.3 GetCol()                             | 44 |
| 5.2.3.4 GetRow()                             | 45 |
| 5.2.3.5 operator() [1/2]                     | 45 |
| 5.2.3.6 operator() [2/2]                     | 45 |
| 5.2.3.7 operator*() [1/2]                    | 45 |
| 5.2.3.8 operator*() [2/2]                    | 45 |
| 5.2.3.9 operator+=()                         | 46 |
| 5.2.3.10 operator-=()                        | 46 |
| 5.2.3.11 operator=() [1/2]                   | 46 |
| 5.2.3.12 operator=() [2/2]                   | 46 |
| 5.2.3.13 SetCol()                            | 46 |
| 5.2.3.14 SetRow()                            | 47 |
| 5.3 FAMath::Matrix4x4 Class Reference        | 47 |
| 5.3.1 Detailed Description                   | 48 |
| 5.3.2 Constructor & Destructor Documentation | 48 |
| 5.3.2.1 Matrix4x4() [1/5]                    | 48 |
| 5.3.2.2 Matrix4x4() [2/5]                    | 48 |
| 5.3.2.3 Matrix4x4() [3/5]                    | 48 |
| 5.3.2.4 Matrix4x4() [4/5]                    | 49 |
| 5.3.2.5 Matrix4x4() [5/5]                    | 49 |
| 5.3.3 Member Function Documentation          | 49 |
| 5.3.3.1 Data() [1/2]                         | 49 |
| 5.3.3.2 Data() [2/2]                         | 49 |
| 5.3.3.3 GetCol()                             | 49 |
| 5.3.3.4 GetRow()                             | 50 |
| 5.3.3.5 operator() [1/2]                     | 50 |
| 5.3.3.6 operator() [2/2]                     | 50 |
| 5.3.3.7 operator*() [1/2]                    | 50 |
| 5.3.3.8 operator*() [2/2]                    | 50 |
| 5.3.3.9 operator+=()                         | 51 |
| 5.3.3.10 operator-=()                        | 51 |
| 5.3.3.11 operator=() [1/2]                   | 51 |
| 5.3.3.12 operator=() [2/2]                   | 51 |
| 5.3.3.13 SetCol()                            | 51 |
| 5.3.3.14 SetRow()                            | 52 |
| 5.4 FAMath::Quaternion Class Reference       | 52 |
| 5.4.1 Detailed Description                   | 53 |
| 5.4.2 Constructor & Destructor Documentation | 53 |
| 5.4.2.1 Quaternion() [1/3]                   | 53 |
| 5.4.2.2 Quaternion() [2/3]                   | 53 |
| 5.4.2.3 Quaternion() [3/3]                   | 53 |

|  |    |
|--|----|
| 5.4.3 Member Function Documentation          | 53 |
| 5.4.3.1 GetScalar()                          | 54 |
| 5.4.3.2 GetVector()                          | 54 |
| 5.4.3.3 GetX()                               | 54 |
| 5.4.3.4 GetY()                               | 54 |
| 5.4.3.5 GetZ()                               | 54 |
| 5.4.3.6 operator*=( ) [1/2]                  | 54 |
| 5.4.3.7 operator*=( ) [2/2]                  | 55 |
| 5.4.3.8 operator+=( )                        | 55 |
| 5.4.3.9 operator-=( )                        | 55 |
| 5.4.3.10 SetScalar()                         | 55 |
| 5.4.3.11 SetVector()                         | 55 |
| 5.4.3.12 SetX()                              | 55 |
| 5.4.3.13 SetY()                              | 56 |
| 5.4.3.14 SetZ()                              | 56 |
| 5.5 FAMath::Vector2D Class Reference         | 56 |
| 5.5.1 Detailed Description                   | 57 |
| 5.5.2 Constructor & Destructor Documentation | 57 |
| 5.5.2.1 Vector2D() [1/3]                     | 57 |
| 5.5.2.2 Vector2D() [2/3]                     | 57 |
| 5.5.2.3 Vector2D() [3/3]                     | 57 |
| 5.5.3 Member Function Documentation          | 57 |
| 5.5.3.1 GetX()                               | 57 |
| 5.5.3.2 GetY()                               | 58 |
| 5.5.3.3 operator*=( )                        | 58 |
| 5.5.3.4 operator+=( )                        | 58 |
| 5.5.3.5 operator-=( )                        | 58 |
| 5.5.3.6 operator/=( )                        | 58 |
| 5.5.3.7 operator=( ) [1/2]                   | 58 |
| 5.5.3.8 operator=( ) [2/2]                   | 59 |
| 5.5.3.9 SetX()                               | 59 |
| 5.5.3.10 SetY()                              | 59 |
| 5.6 FAMath::Vector3D Class Reference         | 59 |
| 5.6.1 Detailed Description                   | 60 |
| 5.6.2 Constructor & Destructor Documentation | 60 |
| 5.6.2.1 Vector3D() [1/3]                     | 61 |
| 5.6.2.2 Vector3D() [2/3]                     | 61 |
| 5.6.2.3 Vector3D() [3/3]                     | 61 |
| 5.6.3 Member Function Documentation          | 61 |
| 5.6.3.1 GetX()                               | 61 |
| 5.6.3.2 GetY()                               | 61 |
| 5.6.3.3 GetZ()                               | 62 |



|  |     |
|--|-----|
| 5.6.3.4 operator*=( )                        | 62  |
| 5.6.3.5 operator+=( )                        | 62  |
| 5.6.3.6 operator-=( )                        | 62  |
| 5.6.3.7 operator/=( )                        | 62  |
| 5.6.3.8 operator=( ) [1/2]                   | 62  |
| 5.6.3.9 operator=( ) [2/2]                   | 63  |
| 5.6.3.10 SetX()                              | 63  |
| 5.6.3.11 SetY()                              | 63  |
| 5.6.3.12 SetZ()                              | 63  |
| 5.7 FAMath::Vector4D Class Reference         | 63  |
| 5.7.1 Detailed Description                   | 64  |
| 5.7.2 Constructor & Destructor Documentation | 64  |
| 5.7.2.1 Vector4D() [1/3]                     | 65  |
| 5.7.2.2 Vector4D() [2/3]                     | 65  |
| 5.7.2.3 Vector4D() [3/3]                     | 65  |
| 5.7.3 Member Function Documentation          | 65  |
| 5.7.3.1 GetW()                               | 65  |
| 5.7.3.2 GetX()                               | 65  |
| 5.7.3.3 GetY()                               | 66  |
| 5.7.3.4 GetZ()                               | 66  |
| 5.7.3.5 operator*=( )                        | 66  |
| 5.7.3.6 operator+=( )                        | 66  |
| 5.7.3.7 operator-=( )                        | 66  |
| 5.7.3.8 operator/=( )                        | 66  |
| 5.7.3.9 operator=( ) [1/2]                   | 67  |
| 5.7.3.10 operator=( ) [2/2]                  | 67  |
| 5.7.3.11 SetW()                              | 67  |
| 5.7.3.12 SetX()                              | 67  |
| 5.7.3.13 SetY()                              | 67  |
| 5.7.3.14 SetZ()                              | 67  |
| 6 File Documentation                         | 69  |
| 6.1 FAMathEngine.h                           | 69  |
| Index  | 111 |



# Chapter 1

## Namespace Index

### 1.1 Namespace List

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

|                        |   |                   |
|------------------------|---|-------------------|
| <a href="#">FAMath</a> | Has the classes <a href="#">Vector2D</a> , <a href="#">Vector3D</a> , <a href="#">Vector4D</a> , <a href="#">Matrix2x2</a> , <a href="#">Matrix3x3</a> , <a href="#">Matrix4x4</a> , <a href="#">Quaternion</a> , and utility functions . . . . . | <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::Matrix2x2</a>   |    |
| A matrix class used for 2x2 matrices and their manipulations . . . . .      | 37 |
| <a href="#">FAMath::Matrix3x3</a>   |    |
| A matrix class used for 3x3 matrices and their manipulations . . . . .      | 42 |
| <a href="#">FAMath::Matrix4x4</a>   |    |
| A matrix class used for 4x4 matrices and their manipulations . . . . .      | 47 |
| <a href="#">FAMath::Quaternion</a>  |    |
| A quaternion class used for quaternions and their manipulations . . . . .   | 52 |
| <a href="#">FAMath::Vector2D</a>  |    |
| A vector class used for 2D vectors/points and their manipulations . . . . . | 56 |
| <a href="#">FAMath::Vector3D</a>  |    |
| A vector class used for 3D vectors/points and their manipulations . . . . . | 59 |
| <a href="#">FAMath::Vector4D</a>  |    |
| A vector class used for 4D vectors/points and their manipulations . . . . . | 63 |



## 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](#)  
69





## Chapter 4

# Namespace Documentation

### 4.1 FAMath Namespace Reference

Has the classes [Vector2D](#), [Vector3D](#), [Vector4D](#), [Matrix2x2](#), [Matrix3x3](#), [Matrix4x4](#), [Quaternion](#), and utility functions.

#### Classes

- class [Matrix2x2](#)  
*A matrix class used for 2x2 matrices and their manipulations.*
- class [Matrix3x3](#)  
*A matrix class used for 3x3 matrices and their manipulations.*
- 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)  
*Negates 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.*
- [Matrix2x2 operator+](#) (const [Matrix2x2](#) &m1, const [Matrix2x2](#) &m2)  
*Adds m1 with m2 and returns the result.*
- [Matrix2x2 operator-](#) (const [Matrix2x2](#) &m)  
*Negates the 2x2 matrix m.*
- [Matrix2x2 operator-](#) (const [Matrix2x2](#) &m1, const [Matrix2x2](#) &m2)  
*Subtracts m2 from m1 and returns the result.*
- [Matrix2x2 operator\\*](#) (const [Matrix2x2](#) &m, const float &k)  
*Multiplies m with k and returns the result.*
- [Matrix2x2 operator\\*](#) (const float &k, const [Matrix2x2](#) &m)  
*Multiplies k with \m and returns the result.*
- [Matrix2x2 operator\\*](#) (const [Matrix2x2](#) &m1, const [Matrix2x2](#) &m2)  
*Multiplies m1 with \m2 and returns the result.*
- [Vector2D operator\\*](#) (const [Matrix2x2](#) &m, const [Vector2D](#) &v)  
*Multiplies m with v and returns the result.*
- [Vector2D operator\\*](#) (const [Vector2D](#) &v, const [Matrix2x2](#) &m)  
*Multiplies v with m and returns the result.*
- void [SetToIdentity](#) ([Matrix2x2](#) &m)  
*Sets m to the identity matrix.*
- bool [IsIdentity](#) (const [Matrix2x2](#) &m)  
*Returns true if m is the identity matrix, false otherwise.*
- [Matrix2x2 Transpose](#) (const [Matrix2x2](#) &m)

- Returns the tranpose of the given matrix m.*

  - [Matrix2x2 Scale](#) (const [Matrix2x2](#) &cm, float x, float y)  
*Construct a 2x2 scaling matrix with x, y, z and it post-multiplies by cm.*
  - [Matrix2x2 Scale](#) (const [Matrix2x2](#) &cm, const [Vector2D](#) &scaleVector)  
*Construct a 2x2 scaling matrix with the x, y and z values of scaleVector and it post-multiplies by cm.*
  - [Matrix2x2 Rotate](#) (const [Matrix2x2](#) &cm, float angle)  
*Construct a 2x2 rotation matrix with angle (in degrees) post-multiplies it by cm;.*
  - double [Determinant](#) (const [Matrix2x2](#) &m)  
*Returns the determinant m.*
  - double [Cofactor](#) (const [Matrix2x2](#) &m, unsigned int row, unsigned int col)  
*Returns the cofactor of the row and col in m.*
  - [Matrix2x2 Adjoint](#) (const [Matrix2x2](#) &m)  
*Returns the adjoint of m.*
  - [Matrix2x2 Inverse](#) (const [Matrix2x2](#) &m)  
*Returns the inverse of m.*
  - [Matrix3x3 operator+](#) (const [Matrix3x3](#) &m1, const [Matrix3x3](#) &m2)  
*Adds m1 with m2 and returns the result.*
  - [Matrix3x3 operator-](#) (const [Matrix3x3](#) &m)  
*Negates the 3x3 matrix m.*
  - [Matrix3x3 operator-](#) (const [Matrix3x3](#) &m1, const [Matrix3x3](#) &m2)  
*Subtracts m2 from m1 and returns the result.*
  - [Matrix3x3 operator\\*](#) (const [Matrix3x3](#) &m, const float &k)  
*Multiplies m with k and returns the result.*
  - [Matrix3x3 operator\\*](#) (const float &k, const [Matrix3x3](#) &m)  
*Multiplies k with \m and returns the result.*
  - [Matrix3x3 operator\\*](#) (const [Matrix3x3](#) &m1, const [Matrix3x3](#) &m2)  
*Multiplies m1 with \m2 and returns the result.*
  - [Vector3D operator\\*](#) (const [Matrix3x3](#) &m, const [Vector3D](#) &v)  
*Multiplies m with v and returns the result.*
  - [Vector3D operator\\*](#) (const [Vector3D](#) &v, const [Matrix3x3](#) &m)  
*Multiplies v with m and returns the result.*
  - void [SetTolIdentity](#) ([Matrix3x3](#) &m)  
*Sets m to the identity matrix.*
  - bool [IsIdentity](#) (const [Matrix3x3](#) &m)  
*Returns true if m is the identity matrix, false otherwise.*
  - [Matrix3x3 Transpose](#) (const [Matrix3x3](#) &m)  
*Returns the tranpose of the given matrix m.*
  - [Matrix3x3 Scale](#) (const [Matrix3x3](#) &cm, float x, float y, float z)  
*Construct a 3x3 scaling matrix with x, y, z and post-multiplies it by cm.*
  - [Matrix3x3 Scale](#) (const [Matrix3x3](#) &cm, const [Vector3D](#) &scaleVector)  
*Construct a 3x3 scaling matrix with scaleVector and post-multiplies it by cm.*
  - [Matrix3x3 Rotate](#) (const [Matrix3x3](#) &cm, float angle, float x, float y, float z)  
*Construct a 3x3 rotation matrix with angle (in degrees) and axis (x, y, z) and post-multiplies it by cm.*
  - [Matrix3x3 Rotate](#) (const [Matrix3x3](#) &cm, float angle, const [Vector3D](#) &axis)  
*Construct a 3x3 rotation matrix with angle (in degrees) and axis and post-multiplies it by cm.*
  - double [Determinant](#) (const [Matrix3x3](#) &m)  
*Returns the determinant m.*
  - double [Cofactor](#) (const [Matrix3x3](#) &m, unsigned int row, unsigned int col)  
*Returns the cofactor of the row and col in m.*
  - [Matrix3x3 Adjoint](#) (const [Matrix3x3](#) &m)  
*Returns the adjoint of m.*

- [Matrix3x3 Inverse](#) (const [Matrix3x3](#) &m)  
*Returns the inverse of m.*
- [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 post-multiplies it by cm.*
- [Matrix4x4 Translate](#) (const [Matrix4x4](#) &cm, const [Vector3D](#) &translateVector)  
*Constructs a 4x4 translation matrix with the x, y and z values of translateVector and post-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 post-multiplies it by cm.*
- [Matrix4x4 Scale](#) (const [Matrix4x4](#) &cm, const [Vector3D](#) &scaleVector)  
*Construct a 4x4 scaling matrix with the x, y and z values of the scaleVector and post-multiplies it by 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-multiplies it by cm.*
- [Matrix4x4 Rotate](#) (const [Matrix4x4](#) &cm, float angle, const [Vector3D](#) &axis)  
*Construct a 4x4 rotation matrix with angle (in degrees) and axis and post-multiplies it by cm.*
- double [Determinant](#) (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.*
- [Vector3D Rotate](#) (const [Quaternion](#) &q, const [Vector3D](#) &p)  
*Rotates the specified point/vector  $p$  using the quaternion  $q$ .*
- [Vector4D Rotate](#) (const [Quaternion](#) &q, const [Vector4D](#) &p)  
*Rotates the specified point/vector  $p$  using the quaternion  $q$ .*

### 4.1.1 Detailed Description

Has the classes [Vector2D](#), [Vector3D](#), [Vector4D](#), [Matrix2x2](#), [Matrix3x3](#), [Matrix4x4](#), [Quaternion](#), and utility functions.

### 4.1.2 Function Documentation

#### 4.1.2.1 Adjoint() [1/3]

```
Matrix2x2 FAMath::Adjoint (
    const Matrix2x2 & m ) [inline]
```

Returns the adjoint of  $m$ .

**4.1.2.2 Adjoint()** [2/3]

```
Matrix3x3 FAMath::Adjoint (
    const Matrix3x3 & m ) [inline]
```

Returns the adjoint of  $m$ .

**4.1.2.3 Adjoint()** [3/3]

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

Returns the adjoint of  $m$ .

**4.1.2.4 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).

**4.1.2.5 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)).

**4.1.2.6 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)).

#### 4.1.2.7 Cofactor() [1/3]

```
double FAMath::Cofactor (
    const Matrix2x2 & m,
    unsigned int row,
    unsigned int col ) [inline]
```

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

#### 4.1.2.8 Cofactor() [2/3]

```
double FAMath::Cofactor (
    const Matrix3x3 & m,
    unsigned int row,
    unsigned int col ) [inline]
```

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

#### 4.1.2.9 Cofactor() [3/3]

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

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

#### 4.1.2.10 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.

#### 4.1.2.11 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.



#### 4.1.2.12 Conjugate()

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

Returns the conjugate of quaternion  $q$ .

#### 4.1.2.13 CrossProduct()

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

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

#### 4.1.2.14 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).

#### 4.1.2.15 Determinant() [1/3]

```
double FAMath::Determinant (
    const Matrix2x2 & m ) [inline]
```

Returns the determinant  $m$ .

#### 4.1.2.16 Determinant() [2/3]

```
double FAMath::Determinant (
    const Matrix3x3 & m ) [inline]
```

Returns the determinant  $m$ .

#### 4.1.2.17 Determinant() [3/3]

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

Returns the determinant *m*.

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

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

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

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

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

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

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

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

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

#### 4.1.2.21 Inverse() [1/4]

```
Matrix2x2 FAMath::Inverse (
    const Matrix2x2 & m ) [inline]
```

Returns the inverse of *m*.

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

**4.1.2.22 Inverse()** [2/4]

```
Matrix3x3 FAMath::Inverse (  
    const Matrix3x3 & m ) [inline]
```

Returns the inverse of  $m$ .

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

**4.1.2.23 Inverse()** [3/4]

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

Returns the inverse of  $m$ .

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

**4.1.2.24 Inverse()** [4/4]

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

Returns the invese of  $q$ .

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

**4.1.2.25 IsIdentity()** [1/4]

```
bool FAMath::IsIdentity (  
    const Matrix2x2 & m ) [inline]
```

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

**4.1.2.26 IsIdentity()** [2/4]

```
bool FAMath::IsIdentity (  
    const Matrix3x3 & m ) [inline]
```

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

**4.1.2.27 IsIdentity()** [3/4]

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

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

#### 4.1.2.28 IsIdentity() [4/4]

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

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

#### 4.1.2.29 IsZeroQuaternion()

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

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

#### 4.1.2.30 Length() [1/4]

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

Returns the length of quaternion  $q$ .

#### 4.1.2.31 Length() [2/4]

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

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

#### 4.1.2.32 Length() [3/4]

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

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

#### 4.1.2.33 Length() [4/4]

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

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

#### 4.1.2.34 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.

#### 4.1.2.35 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.

#### 4.1.2.36 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.

#### 4.1.2.37 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.

#### 4.1.2.38 operator\*() [1/24]

```
Matrix2x2 FAMath::operator* (
    const float & k,
    const Matrix2x2 & m ) [inline]
```

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

**4.1.2.39 operator\*() [2/24]**

```
Matrix3x3 FAMath::operator* (
    const float & k,
    const Matrix3x3 & m ) [inline]
```

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

**4.1.2.40 operator\*() [3/24]**

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

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

**4.1.2.41 operator\*() [4/24]**

```
Matrix2x2 FAMath::operator* (
    const Matrix2x2 & m,
    const float & k ) [inline]
```

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

**4.1.2.42 operator\*() [5/24]**

```
Vector2D FAMath::operator* (
    const Matrix2x2 & m,
    const Vector2D & v ) [inline]
```

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

The vector  $v$  is a column vector.

**4.1.2.43 operator\*() [6/24]**

```
Matrix2x2 FAMath::operator* (
    const Matrix2x2 & m1,
    const Matrix2x2 & m2 ) [inline]
```

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

Does  $m1 * m2$  in that order.

**4.1.2.44 operator\*() [7/24]**

```
Matrix3x3 FAMath::operator* (
    const Matrix3x3 & m,
    const float & k ) [inline]
```

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

**4.1.2.45 operator\*() [8/24]**

```
Vector3D FAMath::operator* (
    const Matrix3x3 & m,
    const Vector3D & v ) [inline]
```

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

The vector  $v$  is a column vector.

**4.1.2.46 operator\*() [9/24]**

```
Matrix3x3 FAMath::operator* (
    const Matrix3x3 & m1,
    const Matrix3x3 & m2 ) [inline]
```

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

Does  $m1 * m2$  in that order.

**4.1.2.47 operator\*() [10/24]**

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

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

**4.1.2.48 operator\*() [11/24]**

```
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.

**4.1.2.49 operator\*() [12/24]**

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

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

Does  $m1 * m2$  in that order.

**4.1.2.50 operator\*() [13/24]**

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

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

**4.1.2.51 operator\*() [14/24]**

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

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

**4.1.2.52 operator\*() [15/24]**

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

Returns  $a * k$ .

**4.1.2.53 operator\*() [16/24]**

```
Vector2D FAMath::operator* (
    const Vector2D & v,
    const Matrix2x2 & m ) [inline]
```

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

The vector  $v$  is a row vector.



**4.1.2.54 operator\*() [17/24]**

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

Returns  $a * k$ .

**4.1.2.55 operator\*() [18/24]**

```
Vector3D FAMath::operator* (
    const Vector3D & v,
    const Matrix3x3 & m ) [inline]
```

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

The vector  $v$  is a row vector.

**4.1.2.56 operator\*() [19/24]**

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

Returns  $a * k$ .

**4.1.2.57 operator\*() [20/24]**

```
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.

**4.1.2.58 operator\*() [21/24]**

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

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

**4.1.2.59 operator\*() [22/24]**

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

Returns  $k * a$ .

**4.1.2.60 operator\*() [23/24]**

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

Returns  $k * a$ .

**4.1.2.61 operator\*() [24/24]**

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

Returns  $k * a$ .

**4.1.2.62 operator+() [1/7]**

```
Matrix2x2 FAMath::operator+ (
    const Matrix2x2 & m1,
    const Matrix2x2 & m2 ) [inline]
```

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

**4.1.2.63 operator+() [2/7]**

```
Matrix3x3 FAMath::operator+ (
    const Matrix3x3 & m1,
    const Matrix3x3 & m2 ) [inline]
```

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

**4.1.2.64 operator+()** [3/7]

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

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

**4.1.2.65 operator+()** [4/7]

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

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

**4.1.2.66 operator+()** [5/7]

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

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

**4.1.2.67 operator+()** [6/7]

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

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

**4.1.2.68 operator+()** [7/7]

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

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

#### 4.1.2.69 operator-() [1/14]

```
Matrix2x2 FAMath::operator- (
    const Matrix2x2 & m ) [inline]
```

Negates the 2x2 matrix *m*.

#### 4.1.2.70 operator-() [2/14]

```
Matrix2x2 FAMath::operator- (
    const Matrix2x2 & m1,
    const Matrix2x2 & m2 ) [inline]
```

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

#### 4.1.2.71 operator-() [3/14]

```
Matrix3x3 FAMath::operator- (
    const Matrix3x3 & m ) [inline]
```

Negates the 3x3 matrix *m*.

#### 4.1.2.72 operator-() [4/14]

```
Matrix3x3 FAMath::operator- (
    const Matrix3x3 & m1,
    const Matrix3x3 & m2 ) [inline]
```

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

#### 4.1.2.73 operator-() [5/14]

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

Negates the 4x4 matrix *m*.

**4.1.2.74 operator-() [6/14]**

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

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

**4.1.2.75 operator-() [7/14]**

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

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

**4.1.2.76 operator-() [8/14]**

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

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

**4.1.2.77 operator-() [9/14]**

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

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

**4.1.2.78 operator-() [10/14]**

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

Negates the vector  $v$  and returns the result.

#### 4.1.2.79 operator-() [11/14]

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

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

#### 4.1.2.80 operator-() [12/14]

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

Negates the vector  $v$  and returns the result.

#### 4.1.2.81 operator-() [13/14]

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

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

#### 4.1.2.82 operator-() [14/14]

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

Negates  $v$  and returns the result.

#### 4.1.2.83 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.

**4.1.2.84 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.

**4.1.2.85 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.

**4.1.2.86 Orthonormalize()** [1/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

**4.1.2.87 Orthonormalize()** [2/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

**4.1.2.88 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)

**4.1.2.89 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.

**4.1.2.90 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.

**4.1.2.91 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.

**4.1.2.92 QuaternionToRotationMatrixCol()**

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

Transforms  $q$  into a column-major matrix.

$q$  should be a unit quaternion.

**4.1.2.93 QuaternionToRotationMatrixRow()**

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

Transforms  $q$  into a row-major matrix.

$q$  should be a unit quaternion.



**4.1.2.94 Rotate()** [1/7]

```
Matrix2x2 FAMath::Rotate (
    const Matrix2x2 & cm,
    float angle ) [inline]
```

Construct a 2x2 rotation matrix with *angle* (in degrees) post-multiplies it by *cm*;

Returns *cm* \* rotate.

**4.1.2.95 Rotate()** [2/7]

```
Matrix3x3 FAMath::Rotate (
    const Matrix3x3 & cm,
    float angle,
    const Vector3D & axis ) [inline]
```

Construct a 3x3 rotation matrix with *angle* (in degrees) and *axis* and post-multiplies it by *cm*.

Returns *cm* \* rotate.

**4.1.2.96 Rotate()** [3/7]

```
Matrix3x3 FAMath::Rotate (
    const Matrix3x3 & cm,
    float angle,
    float x,
    float y,
    float z ) [inline]
```

Construct a 3x3 rotation matrix with *angle* (in degrees) and axis (*x*, *y*, *z*) and post-multiplies it by *cm*.

Returns *cm* \* rotate.

**4.1.2.97 Rotate()** [4/7]

```
Matrix4x4 FAMath::Rotate (
    const Matrix4x4 & cm,
    float angle,
    const Vector3D & axis ) [inline]
```

Construct a 4x4 rotation matrix with *angle* (in degrees) and *axis* and post-multiplies it by *cm*.

Returns *cm* \* rotate.

**4.1.2.98 Rotate()** [5/7]

```
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-multiplies it by *cm*.

Returns *cm* \* rotate.

**4.1.2.99 Rotate()** [6/7]

```
Vector3D FAMath::Rotate (
    const Quaternion & q,
    const Vector3D & p ) [inline]
```

Rotates the specified point/vector  $p$  using the quaternion  $q$ .

$q$  should be a rotation quaternion.

**4.1.2.100 Rotate()** [7/7]

```
Vector4D FAMath::Rotate (
    const Quaternion & q,
    const Vector4D & p ) [inline]
```

Rotates the specified point/vector  $p$  using the quaternion  $q$ .

$q$  should be a rotation quaternion.

**4.1.2.101 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.

**4.1.2.102 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.

**4.1.2.103 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.

**4.1.2.104 Scale()** [1/6]

```
Matrix2x2 FAMath::Scale (
    const Matrix2x2 & cm,
    const Vector2D & scaleVector ) [inline]
```

Construct a 2x2 scaling matrix with the x, y and z values of *scaleVector* and it post-multiplies by *cm*.

Returns *cm* \* *scale*.

**4.1.2.105 Scale()** [2/6]

```
Matrix2x2 FAMath::Scale (
    const Matrix2x2 & cm,
    float x,
    float y ) [inline]
```

Construct a 2x2 scaling matrix with x, y, z and it post-multiplies by *cm*.

Returns *cm* \* *scale*.

**4.1.2.106 Scale()** [3/6]

```
Matrix3x3 FAMath::Scale (
    const Matrix3x3 & cm,
    const Vector3D & scaleVector ) [inline]
```

Construct a 3x3 scaling matrix with *scaleVector* and post-multiplies it by *cm*.

Returns *cm* \* *scale*.

**4.1.2.107 Scale()** [4/6]

```
Matrix3x3 FAMath::Scale (
    const Matrix3x3 & cm,
    float x,
    float y,
    float z ) [inline]
```

Construct a 3x3 scaling matrix with x, y, z and post-multiplies it by *cm*.

Returns *cm* \* *scale*.

**4.1.2.108 Scale()** [5/6]

```
Matrix4x4 FAMath::Scale (
    const Matrix4x4 & cm,
    const Vector3D & scaleVector ) [inline]
```

Construct a 4x4 scaling matrix with the x, y and z values of the *scaleVector* and post-multiplies it by *cm*.

Returns *cm* \* *scale*.

#### 4.1.2.109 Scale() [6/6]

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

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

Returns  $cm * scale$ .

#### 4.1.2.110 SetToIdentity() [1/3]

```
void FAMath::SetToIdentity (
    Matrix2x2 & m ) [inline]
```

Sets  $m$  to the identity matrix.

#### 4.1.2.111 SetToIdentity() [2/3]

```
void FAMath::SetToIdentity (
    Matrix3x3 & m ) [inline]
```

Sets  $m$  to the identity matrix.

#### 4.1.2.112 SetToIdentity() [3/3]

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

Sets  $m$  to the identity matrix.

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

**4.1.2.114 Translate()** [1/2]

```
Matrix4x4 FAMath::Translate (
    const Matrix4x4 & cm,
    const Vector3D & translateVector ) [inline]
```

Constructs a 4x4 translation matrix with the x, y and z values of *translateVector* and post-multiplies it by *cm*.

Returns *cm* \* translate.

**4.1.2.115 Translate()** [2/2]

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

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

Returns *cm* \* translate.

**4.1.2.116 Transpose()** [1/3]

```
Matrix2x2 FAMath::Transpose (
    const Matrix2x2 & m ) [inline]
```

Returns the tranpose of the given matrix *m*.

**4.1.2.117 Transpose()** [2/3]

```
Matrix3x3 FAMath::Transpose (
    const Matrix3x3 & m ) [inline]
```

Returns the tranpose of the given matrix *m*.

**4.1.2.118 Transpose()** [3/3]

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

Returns the tranpose of the given matrix *m*.

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

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

Returns true if *a* is the zero vector.

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

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

Returns true if *a* is the zero vector.

**4.1.2.121 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::Matrix2x2 Class Reference

A matrix class used for 2x2 matrices and their manipulations.

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

#### Public Member Functions

- [Matrix2x2](#) ()  
*Creates a new 2x2 identity matrix.*
- [Matrix2x2](#) (float a[ ][2])  
*Creates a new 2x2 matrix with elements initialized to the given 2D array.*
- [Matrix2x2](#) (const [Vector2D](#) &r1, const [Vector2D](#) &r2)  
*Creates a new 2x2 matrix with each row being set to the specified rows.*
- [Matrix2x2](#) (const [Matrix3x3](#) &m)  
*Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 3x3 matrix.*
- [Matrix2x2](#) (const [Matrix4x4](#) &m)  
*Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 4x4 matrix.*
- 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).*
- [Vector2D](#) [GetRow](#) (unsigned int row) const  
*Returns the specified row.*
- [Vector2D](#) [GetCol](#) (unsigned int col) const  
*Returns the specified col.*
- void [SetRow](#) (unsigned int row, [Vector2D](#) v)  
*Sets each element in the given row to the components of vector v.*
- void [SetCol](#) (unsigned int col, [Vector2D](#) v)  
*Sets each element in the given col to the components of vector v.*

- `Matrix2x2 & operator=` (const `Matrix3x3` &m)  
*Sets the values each row to the first two values of the respective rows of the 3x3 matrix.*
- `Matrix2x2 & operator=` (const `Matrix4x4` &m)  
*Sets the values each row to the first two values of the respective rows of the 4x4 matrix.*
- `Matrix2x2 & operator+=` (const `Matrix2x2` &m)  
*Adds this 2x2 matrix with given matrix m and stores the result in this 2x2 matrix.*
- `Matrix2x2 & operator-=` (const `Matrix2x2` &m)  
*Subtracts m from this 2x2 matrix stores the result in this 2x2 matrix.*
- `Matrix2x2 & operator*=` (float k)  
*Multiplies this 2x2 matrix with k and stores the result in this 2x2 matrix.*
- `Matrix2x2 & operator*=` (const `Matrix2x2` &m)  
*Multiplies this 2x2 matrix with given matrix m and stores the result in this 2x2 matrix.*

### 5.1.1 Detailed Description

A matrix class used for 2x2 matrices and their manipulations.

The datatype for the components is float.

The 2x2 matrix is treated as a row-major matrix.

### 5.1.2 Constructor & Destructor Documentation

#### 5.1.2.1 `Matrix2x2()` [1/5]

```
FAMath::Matrix2x2::Matrix2x2 ( ) [inline]
```

Creates a new 2x2 identity matrix.

#### 5.1.2.2 `Matrix2x2()` [2/5]

```
FAMath::Matrix2x2::Matrix2x2 (
    float a[][2] ) [inline]
```

Creates a new 2x2 matrix with elements initialized to the given 2D array.

If *a* isn't a 2x2 matrix, the behavior is undefined.

#### 5.1.2.3 `Matrix2x2()` [3/5]

```
FAMath::Matrix2x2::Matrix2x2 (
    const Vector2D & r1,
    const Vector2D & r2 ) [inline]
```

Creates a new 2x2 matrix with each row being set to the specified rows.



#### 5.1.2.4 Matrix2x2() [4/5]

```
FAMath::Matrix2x2::Matrix2x2 (
    const Matrix3x3 & m ) [inline]
```

Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 3x3 matrix.

#### 5.1.2.5 Matrix2x2() [5/5]

```
FAMath::Matrix2x2::Matrix2x2 (
    const Matrix4x4 & m ) [inline]
```

Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 4x4 matrix.

### 5.1.3 Member Function Documentation

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

```
float * FAMath::Matrix2x2::Data ( ) [inline]
```

Returns a pointer to the first element in the matrix.

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

```
const float * FAMath::Matrix2x2::Data ( ) const [inline]
```

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

#### 5.1.3.3 GetCol()

```
Vector2D FAMath::Matrix2x2::GetCol (
    unsigned int col ) const [inline]
```

Returns the specified *col*.

*Col* should be between [0,1]. If it is out of range the first col will be returned.

#### 5.1.3.4 GetRow()

```
Vector2D FAMath::Matrix2x2::GetRow (
    unsigned int row ) const [inline]
```

Returns the specified *row*.

*Row* should be between [0,1]. If it is out of range the first row will be returned.

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

```
float & FAMath::Matrix2x2::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 [0,1]. 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::Matrix2x2::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 [0,1]. If any of them are out of that range, the first element will be returned.

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

```
Matrix2x2 & FAMath::Matrix2x2::operator*= (
    const Matrix2x2 & m ) [inline]
```

Multiplies this 2x2 matrix with given matrix *m* and stores the result in this 2x2 matrix.

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

```
Matrix2x2 & FAMath::Matrix2x2::operator*= (
    float k ) [inline]
```

Multiplies this 2x2 matrix with *k* and stores the result in this 2x2 matrix.

#### 5.1.3.9 operator+=( )

```
Matrix2x2 & FAMath::Matrix2x2::operator+= (
    const Matrix2x2 & m ) [inline]
```

Adds this 2x2 matrix with given matrix *m* and stores the result in this 2x2 matrix.

#### 5.1.3.10 operator-=( )

```
Matrix2x2 & FAMath::Matrix2x2::operator-= (
    const Matrix2x2 & m ) [inline]
```

Subtracts *m* from this 2x2 matrix stores the result in this 2x2 matrix.

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

```
Matrix2x2 & FAMath::Matrix2x2::operator= (
    const Matrix3x3 & m ) [inline]
```

Sets the values each row to the first two values of the respective rows of the 3x3 matrix.

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

```
Matrix2x2 & FAMath::Matrix2x2::operator= (
    const Matrix4x4 & m ) [inline]
```

Sets the values each row to the first two values of the respective rows of the 4x4 matrix.

#### 5.1.3.13 SetCol()

```
void FAMath::Matrix2x2::SetCol (
    unsigned int col,
    Vector2D v ) [inline]
```

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

*Col* should be between [0,1]. If it is out of range the first col will be set.

### 5.1.3.14 SetRow()

```
void FAMath::Matrix2x2::SetRow (
    unsigned int row,
    Vector2D v ) [inline]
```

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

*Row* should be between [0,1]. 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::Matrix3x3 Class Reference

A matrix class used for 3x3 matrices and their manipulations.

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

### Public Member Functions

- **Matrix3x3** ()  
*Creates a new 3x3 identity matrix.*
- **Matrix3x3** (float a[ ][3])  
*Creates a new 3x3 matrix with elements initialized to the given 2D array.*
- **Matrix3x3** (const **Vector3D** &r1, const **Vector3D** &r2, const **Vector3D** &r3)  
*Creates a new 3x3 matrix with each row being set to the specified rows.*
- **Matrix3x3** (const **Matrix2x2** &m)  
*Creates a new 3x3 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.*
- **Matrix3x3** (const **Matrix4x4** &m)  
*Creates a new 3x3 matrix with each row being set to the first three values of the respective rows of the 4x4 matrix.*
- 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).*
- **Vector3D** **GetRow** (unsigned int row) const  
*Returns the specified row.*
- **Vector3D** **GetCol** (unsigned int col) const  
*Returns the specified col.*
- void **SetRow** (unsigned int row, **Vector3D** v)  
*Sets each element in the given row to the components of vector v.*
- void **SetCol** (unsigned int col, **Vector3D** v)  
*Sets each element in the given col to the components of vector v.*

- **Matrix3x3 & operator=** (const **Matrix2x2** &m)  
*Sets the first two values of the first two rows to the values of the 2x2 matrix.*
- **Matrix3x3 & operator=** (const **Matrix4x4** &m)  
*Sets the values of each row to the first three values of the respective rows of the 4x4 matrix.*
- **Matrix3x3 & operator+=** (const **Matrix3x3** &m)  
*Adds this 3x3 matrix with given matrix m and stores the result in this 3x3 matrix.*
- **Matrix3x3 & operator-=** (const **Matrix3x3** &m)  
*Subtracts m from this 3x3 matrix stores the result in this 3x3 matrix.*
- **Matrix3x3 & operator\*=** (float k)  
*Multiplies this 3x3 matrix with k and stores the result in this 3x3 matrix.*
- **Matrix3x3 & operator\*=** (const **Matrix3x3** &m)  
*Multiplies this 3x3 matrix with given matrix m and stores the result in this 3x3 matrix.*

## 5.2.1 Detailed Description

A matrix class used for 3x3 matrices and their manipulations.

The datatype for the components is float.

The 3x3 matrix is treated as a row-major matrix.

## 5.2.2 Constructor & Destructor Documentation

### 5.2.2.1 Matrix3x3() [1/5]

```
FAMath::Matrix3x3::Matrix3x3 ( ) [inline]
```

Creates a new 3x3 identity matrix.

### 5.2.2.2 Matrix3x3() [2/5]

```
FAMath::Matrix3x3::Matrix3x3 (
    float a[][3] ) [inline]
```

Creates a new 3x3 matrix with elements initialized to the given 2D array.

If *a* isn't a 3x3 matrix, the behavior is undefined.

### 5.2.2.3 Matrix3x3() [3/5]

```
FAMath::Matrix3x3::Matrix3x3 (
    const Vector3D & r1,
    const Vector3D & r2,
    const Vector3D & r3 ) [inline]
```

Creates a new 3x3 matrix with each row being set to the specified rows.

#### 5.2.2.4 Matrix3x3() [4/5]

```
FAMath::Matrix3x3::Matrix3x3 (
    const Matrix2x2 & m ) [inline]
```

Creates a new 3x3 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.

The last value of the first two rows is set to 0. The last row is set to (0, 0, 1);.

#### 5.2.2.5 Matrix3x3() [5/5]

```
FAMath::Matrix3x3::Matrix3x3 (
    const Matrix4x4 & m ) [inline]
```

Creates a new 3x3 matrix with each row being set to the first three values of the respective rows of the 4x4 matrix.

### 5.2.3 Member Function Documentation

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

```
float * FAMath::Matrix3x3::Data ( ) [inline]
```

Returns a pointer to the first element in the matrix.

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

```
const float * FAMath::Matrix3x3::Data ( ) const [inline]
```

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

#### 5.2.3.3 GetCol()

```
Vector3D FAMath::Matrix3x3::GetCol (
    unsigned int col ) const [inline]
```

Returns the specified *col*.

*Col* should be between [0,2]. If it is out of range the first col will be returned.

#### 5.2.3.4 GetRow()

```
Vector3D FAMath::Matrix3x3::GetRow (
    unsigned int row ) const [inline]
```

Returns the specified *row*.

*Row* should be between [0,2]. If it is out of range the first row will be returned.

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

```
float & FAMath::Matrix3x3::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 [0,2]. If any of them are out of that range, the first element will be returned.

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

```
const float & FAMath::Matrix3x3::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 [0,2]. If any of them are out of that range, the first element will be returned.

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

```
Matrix3x3 & FAMath::Matrix3x3::operator*= (
    const Matrix3x3 & m ) [inline]
```

Multiplies this 3x3 matrix with given matrix *m* and stores the result in this 3x3 matrix.

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

```
Matrix3x3 & FAMath::Matrix3x3::operator*= (
    float k ) [inline]
```

Multiplies this 3x3 matrix with *k* and stores the result in this 3x3 matrix.

### 5.2.3.9 operator+=()

```
Matrix3x3 & FAMath::Matrix3x3::operator+= (
    const Matrix3x3 & m ) [inline]
```

Adds this 3x3 matrix with given matrix *m* and stores the result in this 3x3 matrix.

### 5.2.3.10 operator-=()

```
Matrix3x3 & FAMath::Matrix3x3::operator-= (
    const Matrix3x3 & m ) [inline]
```

Subtracts *m* from this 3x3 matrix stores the result in this 3x3 matrix.

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

```
Matrix3x3 & FAMath::Matrix3x3::operator= (
    const Matrix2x2 & m ) [inline]
```

Sets the first two values of the first two rows to the values of the 2x2 matrix.

The last value of the first two rows is set to 0. The last row is set to (0, 0, 1);.

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

```
Matrix3x3 & FAMath::Matrix3x3::operator= (
    const Matrix4x4 & m ) [inline]
```

Sets the values of each row to the first three values of the respective rows of the 4x4 matrix.

### 5.2.3.13 SetCol()

```
void FAMath::Matrix3x3::SetCol (
    unsigned int col,
    Vector3D v ) [inline]
```

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

*Col* should be between [0,2]. If it is out of range the first col will be set.



### 5.2.3.14 SetRow()

```
void FAMath::Matrix3x3::SetRow (
    unsigned int row,
    Vector3D v ) [inline]
```

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

*Row* should be between [0,2]. 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

## 5.3 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.*
- [Matrix4x4](#) (const [Matrix2x2](#) &m)  
*Creates a new 4x4 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.*
- [Matrix4x4](#) (const [Matrix3x3](#) &m)  
*Creates a new 4x4 matrix with the first three values of the first three rows being set to the values of the 3x3 matrix.*
- [Matrix4x4](#) & [operator=](#) (const [Matrix2x2](#) &m)  
*Sets the first two values of the first two rows to the values of the 2x2 matrix.*
- [Matrix4x4](#) & [operator=](#) (const [Matrix3x3](#) &m)  
*Sets the first three values of the first three rows to the values of the 3x3 matrix.*
- 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.*

### 5.3.1 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.

### 5.3.2 Constructor & Destructor Documentation

#### 5.3.2.1 **Matrix4x4()** [1/5]

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

Creates a new 4x4 identity matrix.

#### 5.3.2.2 **Matrix4x4()** [2/5]

```
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.

#### 5.3.2.3 **Matrix4x4()** [3/5]

```
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.

#### 5.3.2.4 Matrix4x4() [4/5]

```
FAMath::Matrix4x4::Matrix4x4 (
    const Matrix2x2 & m ) [inline]
```

Creates a new 4x4 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.

The last two values of the first two rows are set to (0, 0). The values of the 3rd row is set to (0, 0, 1, 0). The values of the 4th row is set to (0, 0, 0, 1).

#### 5.3.2.5 Matrix4x4() [5/5]

```
FAMath::Matrix4x4::Matrix4x4 (
    const Matrix3x3 & m ) [inline]
```

Creates a new 4x4 matrix with the first three values of the first three rows being set to the values of the 3x3 matrix.

The last values of the first three rows are set to 0. The values of the 4th row is set to (0, 0, 0, 1).

### 5.3.3 Member Function Documentation

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

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

Returns a pointer to the first element in the matrix.

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

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

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

#### 5.3.3.3 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.

#### 5.3.3.4 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.

#### 5.3.3.5 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.

#### 5.3.3.6 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.

#### 5.3.3.7 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.

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

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

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

**5.3.3.9 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.

**5.3.3.10 operator-=()**

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

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

**5.3.3.11 operator=() [1/2]**

```
Matrix4x4 & FAMath::Matrix4x4::operator= (
    const Matrix2x2 & m ) [inline]
```

Sets the first two values of the first two rows to the values of the 2x2 matrix.

The last two values of the first two rows are set to (0, 0). The values of the 3rd row is set to (0, 0, 1, 0). The values of the 4th row is set to (0, 0, 0, 1).

**5.3.3.12 operator=() [2/2]**

```
Matrix4x4 & FAMath::Matrix4x4::operator= (
    const Matrix3x3 & m ) [inline]
```

Sets the first three values of the first three rows to the values of the 3x3 matrix.

The last values of the first three rows are set to 0. The values of the 4th row is set to (0, 0, 0, 1).

**5.3.3.13 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.

### 5.3.3.14 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

## 5.4 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.*
- [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.*

### 5.4.1 Detailed Description

A quaternion class used for quaternions and their manipulations.

The datatype for the components is float.

### 5.4.2 Constructor & Destructor Documentation

#### 5.4.2.1 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.

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

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

Constructs a quaternion with the specified values.

#### 5.4.2.3 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.

### 5.4.3 Member Function Documentation

#### 5.4.3.1 GetScalar()

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

Returns the scalar component of the quaternion.

#### 5.4.3.2 GetVector()

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

Returns the vector component of the quaternion.

#### 5.4.3.3 GetX()

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

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

#### 5.4.3.4 GetY()

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

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

#### 5.4.3.5 GetZ()

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

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

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

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

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



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

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

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

#### 5.4.3.8 operator+=( )

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

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

#### 5.4.3.9 operator-=( )

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

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

#### 5.4.3.10 SetScalar()

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

Sets the scalar component to the specified value.

#### 5.4.3.11 SetVector()

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

Sets the vector to the specified vector.

#### 5.4.3.12 SetX()

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

Sets the x component to the specified value.

#### 5.4.3.13 SetY()

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

Sets the y component to the specified value.

#### 5.4.3.14 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

## 5.5 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.*

### 5.5.1 Detailed Description

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

The datatype for the components is float.

### 5.5.2 Constructor & Destructor Documentation

#### 5.5.2.1 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.

#### 5.5.2.2 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.

#### 5.5.2.3 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.

### 5.5.3 Member Function Documentation

#### 5.5.3.1 GetX()

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

Returns the x component.

### 5.5.3.2 GetY()

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

Returns the y component.

### 5.5.3.3 operator\*=( )

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

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

### 5.5.3.4 operator+=( )

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

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

### 5.5.3.5 operator-=( )

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

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

### 5.5.3.6 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.

### 5.5.3.7 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.

#### 5.5.3.8 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.

#### 5.5.3.9 SetX()

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

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

#### 5.5.3.10 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/FAMath↵  
Engine.h

## 5.6 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.*

### 5.6.1 Detailed Description

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

The datatype for the components is float.

### 5.6.2 Constructor & Destructor Documentation

### 5.6.2.1 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.

### 5.6.2.2 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;.

### 5.6.2.3 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.

## 5.6.3 Member Function Documentation

### 5.6.3.1 GetX()

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

Returns the x component.

### 5.6.3.2 GetY()

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

Returns y component.

#### 5.6.3.3 GetZ()

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

Returns the z component.

#### 5.6.3.4 operator\*=( )

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

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

#### 5.6.3.5 operator+=( )

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

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

#### 5.6.3.6 operator-=( )

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

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

#### 5.6.3.7 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.

#### 5.6.3.8 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.



**5.6.3.9 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.

**5.6.3.10 SetX()**

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

Sets the x component to the specified value.

**5.6.3.11 SetY()**

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

Sets the y component to the specified value.

**5.6.3.12 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

**5.7 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.*

### 5.7.1 Detailed Description

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

The datatype for the components is float

### 5.7.2 Constructor & Destructor Documentation

### 5.7.2.1 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.

### 5.7.2.2 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.

### 5.7.2.3 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.

## 5.7.3 Member Function Documentation

### 5.7.3.1 GetW()

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

Returns the w component.

### 5.7.3.2 GetX()

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

Returns the x component.

### 5.7.3.3 GetY()

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

Returns the y component.

### 5.7.3.4 GetZ()

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

Returns the z component.

### 5.7.3.5 operator\*=( )

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

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

### 5.7.3.6 operator+=( )

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

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

### 5.7.3.7 operator-=( )

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

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

### 5.7.3.8 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.

#### 5.7.3.9 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.

#### 5.7.3.10 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.

#### 5.7.3.11 SetW()

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

Sets the w component to the specified value.

#### 5.7.3.12 SetX()

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

Sets the x component to the specified value.

#### 5.7.3.13 SetY()

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

Sets the y component to the specified value.

#### 5.7.3.14 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     class Matrix2x2;
20     class Matrix3x3;
21     class Matrix4x4;
22
23     inline bool CompareFloats(float x, float y, float epsilon)
24     {
25         float diff = fabs(x - y);
26         //exact epsilon
27         if (diff < epsilon)
28         {
29             return true;
30         }
31
32         //adapative epsilon
33         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
34     }
35
36     inline bool CompareDoubles(double x, double y, double epsilon)
37     {
38         double diff = fabs(x - y);
39         //exact epsilon
40         if (diff < epsilon)
41         {
42             return true;
43         }
44
45         //adapative epsilon
46         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
47     }
48
49     //-----
50
51     class Vector2D
52     {
53     public:
54
55         Vector2D(float x = 0.0f, float y = 0.0f);
56
57         Vector2D(const Vector3D& v);
58     }
```

```

78
81     Vector2D(const Vector4D& v);
82
85     float GetX() const;
86
89     float GetY() const;
90
93     void SetX(float x);
94
97     void SetY(float y);
98
101     Vector2D& operator=(const Vector3D& v);
102
105     Vector2D& operator=(const Vector4D& v);
106
109     Vector2D& operator+=(const Vector2D& b);
110
113     Vector2D& operator-=(const Vector2D& b);
114
117     Vector2D& operator*=(float k);
118
123     Vector2D& operator/=(float k);
124
125 private:
126     float mX;
127     float mY;
128 };
129
130
131 //-----
132 //Vector2D Constructor
133
134 inline Vector2D::Vector2D(float x, float y) : mX{ x }, mY{ y }
135 {}
136 //-----
137
138 //-----
139 //Vector2D Getters and Setters
140
141 inline float Vector2D::GetX()const
142 {
143     return mX;
144 }
145
146 inline float Vector2D::GetY()const
147 {
148     return mY;
149 }
150
151 inline void Vector2D::SetX(float x)
152 {
153     mX = x;
154 }
155
156 inline void Vector2D::SetY(float y)
157 {
158     mY = y;
159 }
160
161 //-----
162
163 //-----
164
165 //Vector2D Member functions
166
167 inline Vector2D& Vector2D::operator+=(const Vector2D& b)
168 {
169     this->mX += b.mX;
170     this->mY += b.mY;
171
172     return *this;
173 }
174
175 inline Vector2D& Vector2D::operator-=(const Vector2D& b)
176 {
177     this->mX -= b.mX;
178     this->mY -= b.mY;
179
180     return *this;
181 }
182
183 inline Vector2D& Vector2D::operator*=(float k)
184 {
185     this->mX *= k;
186     this->mY *= k;
187
188     return *this;

```



```

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

```

```

298     return Vector2D(v.GetX() / mag, v.GetY() / mag);
299 }
300
301 inline Vector2D PolarToCartesian(const Vector2D& v)
302 {
303     //v = (r, theta)
304     //x = rcos(theta)
305     //y = rsin(theta)
306     float angle{ v.GetY() * PI / 180.0f };
307
308     return Vector2D(v.GetX() * cos(angle), v.GetY() * sin(angle));
309 }
310
311 inline Vector2D CartesianToPolar(const Vector2D& v)
312 {
313     //v = (x, y)
314     //r = sqrt(vx^2 + vy^2)
315     //theta = arctan(y / x)
316
317     if (CompareFloats(v.GetX(), 0.0f, EPSILON))
318     {
319         return v;
320     }
321
322     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
323     return Vector2D(Length(v), theta);
324 }
325
326 inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
327 {
328     //Proj_b(a) = (a dot b)b
329     //normalize b before projecting
330
331     Vector2D normB(Norm(b));
332     return Vector2D(DotProduct(a, normB) * normB);
333 }
334
335 #if defined(_DEBUG)
336 inline void print(const Vector2D& v)
337 {
338     std::cout << "(" << v.GetX() << ", " << v.GetY() << ")";
339 }
340 #endif
341 //-----
342
343 //-----
344
345 //-----
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366 class Vector3D
367 {
368 public:
369
370     Vector3D(float x = 0.0f, float y = 0.0f, float z = 0.0f);
371
372     Vector3D(const Vector2D& v, float z = 0.0f);
373
374     Vector3D(const Vector4D& v);
375
376     float GetX() const;
377
378     float GetY() const;
379
380     float GetZ() const;
381
382     void SetX(float x);
383
384     void SetY(float y);
385
386     void SetZ(float z);
387
388     Vector3D& operator=(const Vector2D& v);
389
390     Vector3D& operator=(const Vector4D& v);
391
392     Vector3D& operator+=(const Vector3D& b);
393
394     Vector3D& operator-=(const Vector3D& b);
395
396     Vector3D& operator*=(float k);
397
398     Vector3D& operator/=(float k);
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431

```

```

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

```

```

519         this->mY /= k;
520         this->mZ /= k;
521     }
522     return *this;
523 }
524
525 //-----
526 //-----
527 //-----
528 //Vector3D Non-member functions
529
530 inline bool ZeroVector(const Vector3D& a)
531 {
532     if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
533         CompareFloats(a.GetZ(), 0.0f, EPSILON))
534     {
535         return true;
536     }
537     return false;
538 }
539
540 inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
541 {
542     return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
543 }
544
545 inline Vector3D operator-(const Vector3D& v)
546 {
547     return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
548 }
549
550 inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
551 {
552     return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
553 }
554
555 inline Vector3D operator*(const Vector3D& a, float k)
556 {
557     return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
558 }
559
560 inline Vector3D operator*(float k, const Vector3D& a)
561 {
562     return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
563 }
564
565 inline Vector3D operator/(const Vector3D& a, float k)
566 {
567     if (CompareFloats(k, 0.0f, EPSILON))
568     {
569         return Vector3D();
570     }
571     return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
572 }
573
574 inline float DotProduct(const Vector3D& a, const Vector3D& b)
575 {
576     //a dot b = axbx + ayby + azbz
577     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
578 }
579
580 inline Vector3D CrossProduct(const Vector3D& a, const Vector3D& b)
581 {
582     //a x b = (aybz - azby, azbx - axbz, axby - aybx)
583     return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
584         a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
585         a.GetX() * b.GetY() - a.GetY() * b.GetX());
586 }
587
588 inline float Length(const Vector3D& v)
589 {
590     //length(v) = sqrt(vx^2 + vy^2 + vz^2)
591     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
592 }
593
594 inline Vector3D Norm(const Vector3D& v)
595 {
596     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
597     //v is the zero vector
598     if (ZeroVector(v))
599     {
600         return v;
601     }
602 }

```

```

632
633     float mag{ Length(v) };
634
635     return Vector3D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag);
636 }
637
643 inline Vector3D CylindricalToCartesian(const Vector3D& v)
644 {
645     //v = (r, theta, z)
646     //x = rcos(theta)
647     //y = rsin(theta)
648     //z = z
649     float angle{ v.GetY() * PI / 180.0f };
650
651     return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
652 }
653
660 inline Vector3D CartesianToCylindrical(const Vector3D& v)
661 {
662     //v = (x, y, z)
663     //r = sqrt(vx^2 + vy^2 + vz^2)
664     //theta = arctan(y / x)
665     //z = z
666     if (CompareFloats(v.GetX(), 0.0f, EPSILON))
667     {
668         return v;
669     }
670
671     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
672     return Vector3D(Length(v), theta, v.GetZ());
673 }
674
680 inline Vector3D SphericalToCartesian(const Vector3D& v)
681 {
682     // v = (pho, phi, theta)
683     //x = pho * sin(phi) * cos(theta)
684     //y = pho * sin(phi) * sin(theta)
685     //z = pho * cos(theta);
686
687     float phi{ v.GetY() * PI / 180.0f };
688     float theta{ v.GetZ() * PI / 180.0f };
689
690     return Vector3D(v.GetX() * sin(phi) * cos(theta), v.GetX() * sin(phi) * sin(theta), v.GetX() *
cos(theta));
691 }
692
698 inline Vector3D CartesianToSpherical(const Vector3D& v)
699 {
700     //v = (x, y, z)
701     //pho = sqrt(vx^2 + vy^2 + vz^2)
702     //phi = acos(z / pho)
703     //theta = arctan(y / x)
704
705     if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
706     {
707         return v;
708     }
709
710     float pho{ Length(v) };
711     float phi{ acos(v.GetZ() / pho) * 180.0f / PI };
712     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
713
714     return Vector3D(pho, phi, theta);
715 }
716
721 inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
722 {
723     //ProjB(a) = (a dot b)b
724     //normalize b before projecting
725
726     Vector3D normB(Norm(b));
727     return Vector3D(DotProduct(a, normB) * normB);
728 }
729
734 inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
735 {
736     x = Norm(x);
737     y = Norm(CrossProduct(z, x));
738     z = Norm(CrossProduct(x, y));
739 }
740
741
742 #if defined(_DEBUG)
743 inline void print(const Vector3D& v)
744 {
745     std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ")";
746 }

```

```

747 #endif
748 //-----
749
750
751 //-----
752
753
754 //-----
755
756 class Vector4D
757 {
758 public:
759     Vector4D(float x = 0.0f, float y = 0.0f, float z = 0.0f, float w = 0.0f);
760
761     Vector4D(const Vector2D& v, float z = 0.0f, float w = 0.0f);
762
763     Vector4D(const Vector3D& v, float w = 0.0f);
764
765     float GetX() const;
766
767     float GetY() const;
768
769     float GetZ() const;
770
771     float GetW() const;
772
773     void SetX(float x);
774
775     void SetY(float y);
776
777     void SetZ(float z);
778
779     void SetW(float w);
780
781     Vector4D& operator=(const Vector2D& v);
782
783     Vector4D& operator=(const Vector3D& v);
784
785     Vector4D& operator+=(const Vector4D& b);
786
787     Vector4D& operator-=(const Vector4D& b);
788
789     Vector4D& operator*=(float k);
790
791     Vector4D& operator/=(float k);
792
793 private:
794     float mX;
795     float mY;
796     float mZ;
797     float mW;
798 };
799
800 //-----
801 //Vector4D Constructors
802
803 inline Vector4D::Vector4D(float x, float y, float z, float w) : mX{ x }, mY{ y }, mZ{ z }, mW{ w }
804 {
805 }
806
807 //-----
808
809 //-----
810 //Vector4D Getters and Setters
811
812 inline float Vector4D::GetX() const
813 {
814     return mX;
815 }
816
817 inline float Vector4D::GetY() const
818 {
819     return mY;
820 }
821
822 inline float Vector4D::GetZ() const
823 {
824     return mZ;
825 }
826
827 inline float Vector4D::GetW() const
828 {
829     return mW;
830 }
831
832 inline void Vector4D::SetX(float x)
833 {

```

```

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

```

```

    b.GetW());
964 }
965
966 inline Vector4D operator-(const Vector4D& v)
967 {
970     return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
971 }
972
973 inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
974 {
975     return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
976     b.GetW());
977 }
978
979 inline Vector4D operator*(const Vector4D& a, float k)
980 {
981     return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
982 }
983
984 inline Vector4D operator*(float k, const Vector4D& a)
985 {
986     return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
987 }
988
989 inline Vector4D operator/(const Vector4D& a, float k)
990 {
1000     if (CompareFloats(k, 0.0f, EPSILON))
1001     {
1002         return Vector4D();
1003     }
1004
1005     return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
1006 }
1007
1008 inline float DotProduct(const Vector4D& a, const Vector4D& b)
1009 {
1010     //a dot b = axbx + ayby + azbz + awbw
1011     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
1012 }
1013
1014 inline float Length(const Vector4D& v)
1015 {
1016     //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
1017     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
1018     v.GetW());
1019 }
1020
1021 inline Vector4D Norm(const Vector4D& v)
1022 {
1023     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
1024     //v is the zero vector
1025     if (ZeroVector(v))
1026     {
1027         return v;
1028     }
1029
1030     float mag{ Length(v) };
1031
1032     return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
1033 }
1034
1035 inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
1036 {
1037     //Projb(a) = (a dot b)b
1038     //normalize b before projecting
1039     Vector4D normB(Norm(b));
1040     return Vector4D(DotProduct(a, normB) * normB);
1041 }
1042
1043 inline void Orthonormalize(Vector4D& x, Vector4D& y, Vector4D& z)
1044 {
1045     FAMath::Vector3D tempX(x.GetX(), x.GetY(), x.GetZ());
1046     FAMath::Vector3D tempY(y.GetX(), y.GetY(), y.GetZ());
1047     FAMath::Vector3D tempZ(z.GetX(), z.GetY(), z.GetZ());
1048
1049     tempX = Norm(tempX);
1050     tempY = Norm(CrossProduct(tempZ, tempX));
1051     tempZ = Norm(CrossProduct(tempX, tempY));
1052
1053     x = FAMath::Vector4D(tempX.GetX(), tempX.GetY(), tempX.GetZ(), 0.0f);
1054     y = FAMath::Vector4D(tempY.GetX(), tempY.GetY(), tempY.GetZ(), 0.0f);
1055     z = FAMath::Vector4D(tempZ.GetX(), tempZ.GetY(), tempZ.GetZ(), 0.0f);
1056 }
1057
1058 #if defined(_DEBUG)
1059 inline void print(const Vector4D& v)

```



```

1076     {
1077         std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ", " << v.GetW() << ")";
1078     }
1079 #endif
1080 //-----
1081
1082
1083
1084
1085
1093 //-----
1094 class Matrix2x2
1095 {
1096 public:
1097
1098     Matrix2x2();
1099
1100     Matrix2x2(float a[][2]);
1101
1102     Matrix2x2(const Vector2D& r1, const Vector2D& r2);
1103
1104     Matrix2x2(const Matrix3x3& m);
1105
1106     Matrix2x2(const Matrix4x4& m);
1107
1108     float* Data();
1109
1110     const float* Data() const;
1111
1112     const float& operator()(unsigned int row, unsigned int col) const;
1113
1114     float& operator()(unsigned int row, unsigned int col);
1115
1116     Vector2D GetRow(unsigned int row) const;
1117
1118     Vector2D GetCol(unsigned int col) const;
1119
1120     void SetRow(unsigned int row, Vector2D v);
1121
1122     void SetCol(unsigned int col, Vector2D v);
1123
1124     Matrix2x2& operator=(const Matrix3x3& m);
1125
1126     Matrix2x2& operator=(const Matrix4x4& m);
1127
1128     Matrix2x2& operator+=(const Matrix2x2& m);
1129
1130     Matrix2x2& operator-=(const Matrix2x2& m);
1131
1132     Matrix2x2& operator*=(float k);
1133
1134     Matrix2x2& operator*=(const Matrix2x2& m);
1135
1136 private:
1137
1138     float mMat[2][2];
1139 };
1140
1141 //-----
1142 inline Matrix2x2::Matrix2x2()
1143 {
1144     //1st row
1145     mMat[0][0] = 1.0f;
1146     mMat[0][1] = 0.0f;
1147
1148     //2nd
1149     mMat[1][0] = 0.0f;
1150     mMat[1][1] = 1.0f;
1151 }
1152
1153 inline Matrix2x2::Matrix2x2(float a[][2])
1154 {
1155     //1st row
1156     mMat[0][0] = a[0][0];
1157     mMat[0][1] = a[0][1];
1158
1159     //2nd row
1160     mMat[1][0] = a[1][0];
1161     mMat[1][1] = a[1][1];
1162 }
1163
1164 inline Matrix2x2::Matrix2x2(const Vector2D& r1, const Vector2D& r2)
1165 {
1166     SetRow(0, r1);
1167     SetRow(1, r2);
1168 }
1169
1170
1171
1172

```

```

1221     inline float* Matrix2x2::Data()
1222     {
1223         return mMat[0];
1224     }
1225
1226     inline const float* Matrix2x2::Data()const
1227     {
1228         return mMat[0];
1229     }
1230
1231     inline const float& Matrix2x2::operator()(unsigned int row, unsigned int col)const
1232     {
1233         if (row > 1 || col > 1)
1234         {
1235             return mMat[0][0];
1236         }
1237         else
1238         {
1239             return mMat[row][col];
1240         }
1241     }
1242
1243     inline float& Matrix2x2::operator()(unsigned int row, unsigned int col)
1244     {
1245         if (row > 1 || col > 1)
1246         {
1247             return mMat[0][0];
1248         }
1249         else
1250         {
1251             return mMat[row][col];
1252         }
1253     }
1254
1255     inline Vector2D Matrix2x2::GetRow(unsigned int row)const
1256     {
1257         if (row < 0 || row > 1)
1258             return Vector2D(mMat[0][0], mMat[0][1]);
1259         else
1260             return Vector2D(mMat[row][0], mMat[row][1]);
1261     }
1262
1263     inline Vector2D Matrix2x2::GetCol(unsigned int col)const
1264     {
1265         if (col < 0 || col > 1)
1266             return Vector2D(mMat[0][0], mMat[1][0]);
1267         else
1268             return Vector2D(mMat[0][col], mMat[1][col]);
1269     }
1270
1271     inline void Matrix2x2::SetRow(unsigned int row, Vector2D v)
1272     {
1273         if (row > 1)
1274         {
1275             mMat[0][0] = v.GetX();
1276             mMat[0][1] = v.GetY();
1277         }
1278         else
1279         {
1280             mMat[row][0] = v.GetX();
1281             mMat[row][1] = v.GetY();
1282         }
1283     }
1284
1285     inline void Matrix2x2::SetCol(unsigned int col, Vector2D v)
1286     {
1287         if (col > 1)
1288         {
1289             mMat[0][0] = v.GetX();
1290             mMat[1][0] = v.GetY();
1291         }
1292         else
1293         {
1294             mMat[0][col] = v.GetX();
1295             mMat[1][col] = v.GetY();
1296         }
1297     }
1298
1299     inline Matrix2x2& Matrix2x2::operator+=(const Matrix2x2& m)
1300     {
1301         for (int i = 0; i < 2; ++i)
1302         {
1303             for (int j = 0; j < 2; ++j)
1304             {
1305                 this->mMat[i][j] += m.mMat[i][j];
1306             }
1307         }

```

```

1308     }
1309
1310     return *this;
1311 }
1312
1313 inline Matrix2x2& Matrix2x2::operator-=(const Matrix2x2& m)
1314 {
1315     for (int i = 0; i < 2; ++i)
1316     {
1317         for (int j = 0; j < 2; ++j)
1318         {
1319             this->mMat[i][j] -= m.mMat[i][j];
1320         }
1321     }
1322
1323     return *this;
1324 }
1325
1326 inline Matrix2x2& Matrix2x2::operator*=(float k)
1327 {
1328     for (int i = 0; i < 2; ++i)
1329     {
1330         for (int j = 0; j < 2; ++j)
1331         {
1332             this->mMat[i][j] *= k;
1333         }
1334     }
1335
1336     return *this;
1337 }
1338
1339 inline Matrix2x2& Matrix2x2::operator*=(const Matrix2x2& m)
1340 {
1341     Matrix2x2 res;
1342
1343     for (int i = 0; i < 2; ++i)
1344     {
1345         res.mMat[i][0] =
1346             (mMat[i][0] * m.mMat[0][0]) +
1347             (mMat[i][1] * m.mMat[1][0]);
1348
1349         res.mMat[i][1] =
1350             (mMat[i][0] * m.mMat[0][1]) +
1351             (mMat[i][1] * m.mMat[1][1]);
1352     }
1353
1354     for (int i = 0; i < 2; ++i)
1355     {
1356         for (int j = 0; j < 2; ++j)
1357         {
1358             mMat[i][j] = res.mMat[i][j];
1359         }
1360     }
1361
1362     return *this;
1363 }
1364
1365 inline Matrix2x2 operator+(const Matrix2x2& m1, const Matrix2x2& m2)
1366 {
1367     Matrix2x2 res;
1368     for (int i = 0; i < 2; ++i)
1369     {
1370         for (int j = 0; j < 2; ++j)
1371         {
1372             res(i, j) = m1(i, j) + m2(i, j);
1373         }
1374     }
1375
1376     return res;
1377 }
1378
1379 inline Matrix2x2 operator-(const Matrix2x2& m)
1380 {
1381     Matrix2x2 res;
1382     for (int i = 0; i < 2; ++i)
1383     {
1384         for (int j = 0; j < 2; ++j)
1385         {
1386             res(i, j) = -m(i, j);
1387         }
1388     }
1389
1390     return res;
1391 }
1392
1393 inline Matrix2x2 operator-(const Matrix2x2& m1, const Matrix2x2& m2)
1394 {
1395     Matrix2x2 res;
1396     for (int i = 0; i < 2; ++i)
1397     {
1398         for (int j = 0; j < 2; ++j)
1399         {
1400             res(i, j) = m1(i, j) - m2(i, j);
1401         }
1402     }
1403
1404     return res;
1405 }

```

```

1401     Matrix2x2 res;
1402     for (int i = 0; i < 2; ++i)
1403     {
1404         for (int j = 0; j < 2; ++j)
1405         {
1406             res(i, j) = m1(i, j) - m2(i, j);
1407         }
1408     }
1409
1410     return res;
1411 }
1412
1413 inline Matrix2x2 operator*(const Matrix2x2& m, const float& k)
1414 {
1415     Matrix2x2 res;
1416     for (int i = 0; i < 2; ++i)
1417     {
1418         for (int j = 0; j < 2; ++j)
1419         {
1420             res(i, j) = m(i, j) * k;
1421         }
1422     }
1423
1424     return res;
1425 }
1426
1427 inline Matrix2x2 operator*(const float& k, const Matrix2x2& m)
1428 {
1429     Matrix2x2 res;
1430     for (int i = 0; i < 2; ++i)
1431     {
1432         for (int j = 0; j < 2; ++j)
1433         {
1434             res(i, j) = k * m(i, j);
1435         }
1436     }
1437
1438     return res;
1439 }
1440
1441 inline Matrix2x2 operator*(const Matrix2x2& m1, const Matrix2x2& m2)
1442 {
1443     Matrix2x2 res;
1444     for (int i = 0; i < 4; ++i)
1445     {
1446         res(i, 0) =
1447             (m1(i, 0) * m2(0, 0)) +
1448             (m1(i, 1) * m2(1, 0));
1449
1450         res(i, 1) =
1451             (m1(i, 0) * m2(0, 1)) +
1452             (m1(i, 1) * m2(1, 1));
1453
1454         res(i, 2) =
1455             (m1(i, 0) * m2(0, 2)) +
1456             (m1(i, 1) * m2(1, 2));
1457
1458         res(i, 3) =
1459             (m1(i, 0) * m2(0, 3)) +
1460             (m1(i, 1) * m2(1, 3));
1461     }
1462
1463     return res;
1464 }
1465
1466 inline Vector2D operator*(const Matrix2x2& m, const Vector2D& v)
1467 {
1468     Vector2D res;
1469
1470     res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY());
1471
1472     res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY());
1473
1474     return res;
1475 }
1476
1477 inline Vector2D operator*(const Vector2D& v, const Matrix2x2& m)
1478 {
1479     Vector2D res;
1480
1481     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0));
1482
1483     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1));
1484
1485     return res;
1486 }
1487
1488 inline Vector2D operator*(const Vector2D& v, const Matrix2x2& m)
1489 {
1490     Vector2D res;
1491
1492     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0));
1493
1494     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1));
1495
1496     return res;
1497 }
1498
1499
1500
1501
1502
1503

```

```

1504
1507 inline void SetToIdentity(Matrix2x2& m)
1508 {
1509     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1510
1511     //1st row
1512     m(0, 0) = 1.0f;
1513     m(0, 1) = 0.0f;
1514
1515     //2nd row
1516     m(1, 0) = 0.0f;
1517     m(1, 1) = 1.0f;
1518 }
1519
1522 inline bool IsIdentity(const Matrix2x2& m)
1523 {
1524     //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f
1525
1526     for (int i = 0; i < 2; ++i)
1527     {
1528         for (int j = 0; j < 2; ++j)
1529         {
1530             if (i == j)
1531             {
1532                 if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1533                     return false;
1534             }
1535             else
1536             {
1537                 if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1538                     return false;
1539             }
1540         }
1541     }
1542 }
1543
1544 }
1545
1548 inline Matrix2x2 Transpose(const Matrix2x2& m)
1549 {
1550     //make the rows into cols
1551
1552     Matrix2x2 res;
1553
1554     //1st col = 1st row
1555     res(0, 0) = m(0, 0);
1556     res(1, 0) = m(0, 1);
1557
1558     //2nd col = 2nd row
1559     res(0, 1) = m(1, 0);
1560     res(1, 1) = m(1, 1);
1561
1562     return res;
1563 }
1564
1569 inline Matrix2x2 Scale(const Matrix2x2& cm, float x, float y)
1570 {
1571     //x 0
1572     //0 y
1573
1574     Matrix2x2 scale;
1575     scale(0, 0) = x;
1576     scale(1, 1) = y;
1577
1578     return cm * scale;
1579 }
1580
1585 inline Matrix2x2 Scale(const Matrix2x2& cm, const Vector2D& scaleVector)
1586 {
1587     //x 0
1588     //0 y
1589
1590     Matrix2x2 scale;
1591     scale(0, 0) = scaleVector.GetX();
1592     scale(1, 1) = scaleVector.GetY();
1593
1594     return cm * scale;
1595 }
1596
1601 inline Matrix2x2 Rotate(const Matrix2x2& cm, float angle)
1602 {
1603     //c    s
1604     //-s   c
1605     //c = cos(angle)
1606     //s = sin(angle)
1607

```

```

1608         float c = cos(angle * PI / 180.0f);
1609         float s = sin(angle * PI / 180.0f);
1610
1611         Matrix2x2 result;
1612
1613         //1st row
1614         result(0, 0) = c;
1615         result(0, 1) = s;
1616
1617         //2nd row
1618         result(1, 0) = -s;
1619         result(1, 1) = c;
1620
1621         return cm * result;
1622     }
1623
1624     inline double Determinant(const Matrix2x2& m)
1625     {
1626         return (double)m(0, 0) * m(1, 1) - (double)m(0, 1) * m(1, 0);
1627     }
1628
1629     inline double Cofactor(const Matrix2x2& m, unsigned int row, unsigned int col)
1630     {
1631         //cij = ((-1)^(i + j)) * det of minor(i, j);
1632         double minor{ 0.0 };
1633
1634         if (row == 0 && col == 0)
1635             minor = m(1, 1);
1636         else if (row == 0 && col == 1)
1637             minor = m(1, 0);
1638         else if (row == 1 && col == 0)
1639             minor = m(0, 1);
1640         else if (row == 1 && col == 1)
1641             minor = m(0, 0);
1642
1643         return pow(-1, row + col) * minor;
1644     }
1645
1646     inline Matrix2x2 Adjoint(const Matrix2x2& m)
1647     {
1648         //Cofactor of each ijth position put into matrix cA.
1649         //Adjoint is the tranposed matrix of cA.
1650         Matrix2x2 cofactorMatrix;
1651         for (int i = 0; i < 2; ++i)
1652         {
1653             for (int j = 0; j < 2; ++j)
1654             {
1655                 cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
1656             }
1657         }
1658
1659         return Transpose(cofactorMatrix);
1660     }
1661
1662     inline Matrix2x2 Inverse(const Matrix2x2& m)
1663     {
1664         //Inverse of m = adjoint of m / det of m
1665         double det = Determinant(m);
1666         if (CompareDoubles(det, 0.0, EPSILON))
1667             return Matrix2x2();
1668
1669         return Adjoint(m) * (1.0f / static_cast<float>(det));
1670     }
1671
1672     #if defined(_DEBUG)
1673     inline void print(const Matrix2x2& m)
1674     {
1675         for (int i = 0; i < 2; ++i)
1676         {
1677             for (int j = 0; j < 2; ++j)
1678             {
1679                 std::cout << m(i, j) << " ";
1680             }
1681             std::cout << std::endl;
1682         }
1683     }
1684     #endif
1685
1686     //-----
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703

```

```

1704
1705
1706 //-----
1714     class Matrix3x3
1715     {
1716     public:
1717
1720         Matrix3x3();
1721
1726         Matrix3x3(float a[][3]);
1727
1730         Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3);
1731
1737         Matrix3x3(const Matrix2x2& m);
1738
1741         Matrix3x3(const Matrix4x4& m);
1742
1745         float* Data();
1746
1749         const float* Data() const;
1750
1755         const float& operator()(unsigned int row, unsigned int col) const;
1756
1761         float& operator()(unsigned int row, unsigned int col);
1762
1767         Vector3D GetRow(unsigned int row) const;
1768
1773         Vector3D GetCol(unsigned int col) const;
1774
1779         void SetRow(unsigned int row, Vector3D v);
1780
1785         void SetCol(unsigned int col, Vector3D v);
1786
1792         Matrix3x3& operator=(const Matrix2x2& m);
1793
1796         Matrix3x3& operator=(const Matrix4x4& m);
1797
1800         Matrix3x3& operator+=(const Matrix3x3& m);
1801
1804         Matrix3x3& operator-=(const Matrix3x3& m);
1805
1808         Matrix3x3& operator*=(float k);
1809
1812         Matrix3x3& operator*=(const Matrix3x3& m);
1813
1814     private:
1815
1816         float mMat[3][3];
1817     };
1818
1819 //-----
1820     inline Matrix3x3::Matrix3x3()
1821     {
1822         //1st row
1823         mMat[0][0] = 1.0f;
1824         mMat[0][1] = 0.0f;
1825         mMat[0][2] = 0.0f;
1826
1827         //2nd
1828         mMat[1][0] = 0.0f;
1829         mMat[1][1] = 1.0f;
1830         mMat[1][2] = 0.0f;
1831
1832         //3rd row
1833         mMat[2][0] = 0.0f;
1834         mMat[2][1] = 0.0f;
1835         mMat[2][2] = 1.0f;
1836     }
1837
1838     inline Matrix3x3::Matrix3x3(float a[][3])
1839     {
1840         //1st row
1841         mMat[0][0] = a[0][0];
1842         mMat[0][1] = a[0][1];
1843         mMat[0][2] = a[0][2];
1844
1845         //2nd
1846         mMat[1][0] = a[1][0];
1847         mMat[1][1] = a[1][1];
1848         mMat[1][2] = a[1][2];
1849
1850         //3rd row
1851         mMat[2][0] = a[2][0];
1852         mMat[2][1] = a[2][1];
1853         mMat[2][2] = a[2][2];
1854     }

```

```

1855
1856 inline Matrix3x3::Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3)
1857 {
1858     SetRow(0, r1);
1859     SetRow(1, r2);
1860     SetRow(2, r3);
1861 }
1862
1863 inline float* Matrix3x3::Data()
1864 {
1865     return mMat[0];
1866 }
1867
1868 inline const float* Matrix3x3::Data()const
1869 {
1870     return mMat[0];
1871 }
1872
1873 inline const float& Matrix3x3::operator()(unsigned int row, unsigned int col)const
1874 {
1875     if (row > 2 || col > 2)
1876     {
1877         return mMat[0][0];
1878     }
1879     else
1880     {
1881         return mMat[row][col];
1882     }
1883 }
1884
1885 inline float& Matrix3x3::operator()(unsigned int row, unsigned int col)
1886 {
1887     if (row > 2 || col > 2)
1888     {
1889         return mMat[0][0];
1890     }
1891     else
1892     {
1893         return mMat[row][col];
1894     }
1895 }
1896
1897 inline Vector3D Matrix3x3::GetRow(unsigned int row)const
1898 {
1899     if (row < 0 || row > 2)
1900         return Vector3D(mMat[0][0], mMat[0][1], mMat[0][2]);
1901     else
1902         return Vector3D(mMat[row][0], mMat[row][1], mMat[row][2]);
1903 }
1904
1905
1906 inline Vector3D Matrix3x3::GetCol(unsigned int col)const
1907 {
1908     if (col < 0 || col > 2)
1909         return Vector3D(mMat[0][0], mMat[1][0], mMat[2][0]);
1910     else
1911         return Vector3D(mMat[0][col], mMat[1][col], mMat[2][col]);
1912 }
1913
1914 inline void Matrix3x3::SetRow(unsigned int row, Vector3D v)
1915 {
1916     if (row > 2)
1917     {
1918         mMat[0][0] = v.GetX();
1919         mMat[0][1] = v.GetY();
1920         mMat[0][2] = v.GetZ();
1921     }
1922     else
1923     {
1924         mMat[row][0] = v.GetX();
1925         mMat[row][1] = v.GetY();
1926         mMat[row][2] = v.GetZ();
1927     }
1928 }
1929
1930 inline void Matrix3x3::SetCol(unsigned int col, Vector3D v)
1931 {
1932     if (col > 2)
1933     {
1934         mMat[0][0] = v.GetX();
1935         mMat[1][0] = v.GetY();
1936         mMat[2][0] = v.GetZ();
1937     }
1938     else
1939     {
1940         mMat[0][col] = v.GetX();
1941         mMat[1][col] = v.GetY();

```



```

1942         mMat[2][col] = v.GetZ();
1943     }
1944 }
1945
1946 inline Matrix3x3& Matrix3x3::operator+=(const Matrix3x3& m)
1947 {
1948     for (int i = 0; i < 3; ++i)
1949     {
1950         for (int j = 0; j < 3; ++j)
1951         {
1952             this->mMat[i][j] += m.mMat[i][j];
1953         }
1954     }
1955
1956     return *this;
1957 }
1958
1959 inline Matrix3x3& Matrix3x3::operator-=(const Matrix3x3& m)
1960 {
1961     for (int i = 0; i < 3; ++i)
1962     {
1963         for (int j = 0; j < 3; ++j)
1964         {
1965             this->mMat[i][j] -= m.mMat[i][j];
1966         }
1967     }
1968
1969     return *this;
1970 }
1971
1972 inline Matrix3x3& Matrix3x3::operator*=(float k)
1973 {
1974     for (int i = 0; i < 3; ++i)
1975     {
1976         for (int j = 0; j < 3; ++j)
1977         {
1978             this->mMat[i][j] *= k;
1979         }
1980     }
1981
1982     return *this;
1983 }
1984
1985 inline Matrix3x3& Matrix3x3::operator*=(const Matrix3x3& m)
1986 {
1987     Matrix3x3 result;
1988
1989     for (int i = 0; i < 3; ++i)
1990     {
1991         result.mMat[i][0] =
1992             (mMat[i][0] * m.mMat[0][0]) +
1993             (mMat[i][1] * m.mMat[1][0]) +
1994             (mMat[i][2] * m.mMat[2][0]);
1995
1996         result.mMat[i][1] =
1997             (mMat[i][0] * m.mMat[0][1]) +
1998             (mMat[i][1] * m.mMat[1][1]) +
1999             (mMat[i][2] * m.mMat[2][1]);
2000
2001         result.mMat[i][2] =
2002             (mMat[i][0] * m.mMat[0][2]) +
2003             (mMat[i][1] * m.mMat[1][2]) +
2004             (mMat[i][2] * m.mMat[2][2]);
2005     }
2006
2007     for (int i = 0; i < 3; ++i)
2008     {
2009         for (int j = 0; j < 3; ++j)
2010         {
2011             mMat[i][j] = result.mMat[i][j];
2012         }
2013     }
2014
2015     return *this;
2016 }
2017
2018 inline Matrix3x3 operator+(const Matrix3x3& m1, const Matrix3x3& m2)
2019 {
2020     Matrix3x3 result;
2021     for (int i = 0; i < 3; ++i)
2022     {
2023         for (int j = 0; j < 3; ++j)
2024         {
2025             result(i, j) = m1(i, j) + m2(i, j);
2026         }
2027     }
2028
2029     return result;
2030 }

```

```

2031     return result;
2032 }
2033
2034 inline Matrix3x3 operator-(const Matrix3x3& m)
2035 {
2036     Matrix3x3 result;
2037     for (int i = 0; i < 3; ++i)
2038     {
2039         for (int j = 0; j < 3; ++j)
2040         {
2041             result(i, j) = -m(i, j);
2042         }
2043     }
2044     return result;
2045 }
2046
2047 inline Matrix3x3 operator-(const Matrix3x3& m1, const Matrix3x3& m2)
2048 {
2049     Matrix3x3 result;
2050     for (int i = 0; i < 3; ++i)
2051     {
2052         for (int j = 0; j < 3; ++j)
2053         {
2054             result(i, j) = m1(i, j) - m2(i, j);
2055         }
2056     }
2057     return result;
2058 }
2059
2060 inline Matrix3x3 operator*(const Matrix3x3& m, const float& k)
2061 {
2062     Matrix3x3 result;
2063     for (int i = 0; i < 3; ++i)
2064     {
2065         for (int j = 0; j < 3; ++j)
2066         {
2067             result(i, j) = m(i, j) * k;
2068         }
2069     }
2070     return result;
2071 }
2072
2073 inline Matrix3x3 operator*(const float& k, const Matrix3x3& m)
2074 {
2075     Matrix3x3 result;
2076     for (int i = 0; i < 3; ++i)
2077     {
2078         for (int j = 0; j < 3; ++j)
2079         {
2080             result(i, j) = k * m(i, j);
2081         }
2082     }
2083     return result;
2084 }
2085
2086 inline Matrix3x3 operator*(const Matrix3x3& m1, const Matrix3x3& m2)
2087 {
2088     Matrix3x3 result;
2089     for (int i = 0; i < 4; ++i)
2090     {
2091         result(i, 0) =
2092             (m1(i, 0) * m2(0, 0)) +
2093             (m1(i, 1) * m2(1, 0)) +
2094             (m1(i, 2) * m2(2, 0));
2095
2096         result(i, 1) =
2097             (m1(i, 0) * m2(0, 1)) +
2098             (m1(i, 1) * m2(1, 1)) +
2099             (m1(i, 2) * m2(2, 1));
2100
2101         result(i, 2) =
2102             (m1(i, 0) * m2(0, 2)) +
2103             (m1(i, 1) * m2(1, 2)) +
2104             (m1(i, 2) * m2(2, 2));
2105
2106         result(i, 3) =
2107             (m1(i, 0) * m2(0, 3)) +
2108             (m1(i, 1) * m2(1, 3)) +
2109             (m1(i, 2) * m2(2, 3));
2110     }
2111     return result;
2112 }

```

```

2130     }
2131
2132     inline Vector3D operator*(const Matrix3x3& m, const Vector3D& v)
2133     {
2134         Vector3D result;
2135
2136         result.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ());
2137
2138         result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ());
2139
2140         result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ());
2141
2142         return result;
2143     }
2144
2145     inline Vector3D operator*(const Vector3D& v, const Matrix3x3& m)
2146     {
2147         Vector3D result;
2148
2149         result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0));
2150
2151         result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1));
2152
2153         result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2));
2154
2155         return result;
2156     }
2157
2158     inline void SetToIdentity(Matrix3x3& m)
2159     {
2160         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
2161
2162         //1st row
2163         m(0, 0) = 1.0f;
2164         m(0, 1) = 0.0f;
2165         m(0, 2) = 0.0f;
2166
2167         //2nd row
2168         m(1, 0) = 0.0f;
2169         m(1, 1) = 1.0f;
2170         m(1, 2) = 0.0f;
2171
2172         //3rd row
2173         m(2, 0) = 0.0f;
2174         m(2, 1) = 0.0f;
2175         m(2, 2) = 1.0f;
2176     }
2177
2178     inline bool IsIdentity(const Matrix3x3& m)
2179     {
2180         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
2181         0.0f
2182
2183         for (int i = 0; i < 3; ++i)
2184         {
2185             for (int j = 0; j < 3; ++j)
2186             {
2187                 if (i == j)
2188                 {
2189                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
2190                         return false;
2191                 }
2192                 else
2193                 {
2194                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
2195                         return false;
2196                 }
2197             }
2198         }
2199     }
2200
2201     inline Matrix3x3 Transpose(const Matrix3x3& m)
2202     {
2203         //make the rows into cols
2204
2205         Matrix3x3 result;
2206
2207         //1st col = 1st row
2208         result(0, 0) = m(0, 0);
2209         result(1, 0) = m(0, 1);
2210         result(2, 0) = m(0, 2);
2211
2212         //2nd col = 2nd row
2213         result(0, 1) = m(1, 0);
2214         result(1, 1) = m(1, 1);

```

```

2230         result(2, 1) = m(1, 2);
2231
2232         //3rd col = 3rd row
2233         result(0, 2) = m(2, 0);
2234         result(1, 2) = m(2, 1);
2235         result(2, 2) = m(2, 2);
2236
2237         return result;
2238     }
2239
2240     inline Matrix3x3 Scale(const Matrix3x3& cm, float x, float y, float z)
2241     {
2242         //x 0 0
2243         //0 y 0
2244         //0 0 z
2245
2246         Matrix3x3 scale;
2247         scale(0, 0) = x;
2248         scale(1, 1) = y;
2249         scale(2, 2) = z;
2250
2251         return cm * scale;
2252     }
2253
2254     inline Matrix3x3 Scale(const Matrix3x3& cm, const Vector3D& scaleVector)
2255     {
2256         //x 0 0
2257         //0 y 0
2258         //0 0 z
2259
2260         Matrix3x3 scale;
2261         scale(0, 0) = scaleVector.GetX();
2262         scale(1, 1) = scaleVector.GetY();
2263         scale(2, 2) = scaleVector.GetZ();
2264
2265         return cm * scale;
2266     }
2267
2268     inline Matrix3x3 Rotate(const Matrix3x3& cm, float angle, float x, float y, float z)
2269     {
2270         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy
2271         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx
2272         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2
2273         //c = cos(angle)
2274         //s = sin(angle)
2275
2276         Vector3D axis(x, y, z);
2277         axis = Norm(axis);
2278         x = axis.GetX();
2279         y = axis.GetY();
2280         z = axis.GetZ();
2281
2282         float c = cos(angle * PI / 180.0f);
2283         float s = sin(angle * PI / 180.0f);
2284         float oneMinusC = 1.0f - c;
2285
2286         Matrix3x3 result;
2287
2288         //1st row
2289         result(0, 0) = c + (oneMinusC * (x * x));
2290         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2291         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2292
2293         //2nd row
2294         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2295         result(1, 1) = c + (oneMinusC * (y * y));
2296         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2297
2298         //3rd row
2299         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2300         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2301         result(2, 2) = c + (oneMinusC * (z * z));
2302
2303         return cm * result;
2304     }
2305
2306     inline Matrix3x3 Rotate(const Matrix3x3& cm, float angle, const Vector3D& axis)
2307     {
2308         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy
2309         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx
2310         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2
2311         //c = cos(angle)
2312         //s = sin(angle)
2313
2314         Vector3D nAxis(Norm(axis));

```

```

2333     float x = nAxis.GetX();
2334     float y = nAxis.GetY();
2335     float z = nAxis.GetZ();
2336
2337     float c = cos(angle * PI / 180.0f);
2338     float s = sin(angle * PI / 180.0f);
2339     float oneMinusC = 1.0f - c;
2340
2341     Matrix3x3 result;
2342
2343     //1st row
2344     result(0, 0) = c + (oneMinusC * (x * x));
2345     result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2346     result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2347
2348     //2nd row
2349     result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2350     result(1, 1) = c + (oneMinusC * (y * y));
2351     result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2352
2353     //3rd row
2354     result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2355     result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2356     result(2, 2) = c + (oneMinusC * (z * z));
2357
2358     return cm * result;
2359 }
2360
2361 inline double Determinant(const Matrix3x3& m)
2362 {
2363     //m00m11m22 - m00m12m21
2364     double c1 = (double)m(0, 0) * m(1, 1) * m(2, 2) - (double)m(0, 0) * m(1, 2) * m(2, 1);
2365
2366     //m01m12m20 - m01m10m22
2367     double c2 = (double)m(0, 1) * m(1, 2) * m(2, 0) - (double)m(0, 1) * m(1, 0) * m(2, 2);
2368
2369     //m02m10m21 - m02m11m20
2370     double c3 = (double)m(0, 2) * m(1, 0) * m(2, 1) - (double)m(0, 2) * m(1, 1) * m(2, 0);
2371
2372     return c1 + c2 + c3;
2373 }
2374
2375 inline double Cofactor(const Matrix3x3& m, unsigned int row, unsigned int col)
2376 {
2377     //cij = ((-1)^(i + j)) * det of minor(i, j);
2378     Matrix2x2 minor;
2379     int r{ 0 };
2380     int c{ 0 };
2381
2382     //minor(i, j)
2383     for (int i = 0; i < 3; ++i)
2384     {
2385         if (i == row)
2386             continue;
2387
2388         for (int j = 0; j < 3; ++j)
2389         {
2390             if (j == col)
2391                 continue;
2392
2393             minor(r, c) = m(i, j);
2394             ++c;
2395         }
2396         c = 0;
2397         ++r;
2398     }
2399
2400     return pow(-1, row + col) * Determinant(minor);
2401 }
2402
2403 inline Matrix3x3 Adjoint(const Matrix3x3& m)
2404 {
2405     //Cofactor of each ijth position put into matrix cA.
2406     //Adjoint is the tranposed matrix of cA.
2407     Matrix3x3 cofactorMatrix;
2408     for (int i = 0; i < 3; ++i)
2409     {
2410         for (int j = 0; j < 3; ++j)
2411         {
2412             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
2413         }
2414     }
2415
2416     return Transpose(cofactorMatrix);
2417 }
2418
2419
2420
2421
2422
2423
2424
2425

```

```

2430     inline Matrix3x3 Inverse(const Matrix3x3& m)
2431     {
2432         //Inverse of m = adjoint of m / det of m
2433         double det = Determinant(m);
2434         if (CompareDoubles(det, 0.0, EPSILON))
2435             return Matrix3x3();
2436
2437         return Adjoint(m) * (1.0f / static_cast<float>(det));
2438     }
2439
2440
2441 #if defined(_DEBUG)
2442     inline void print(const Matrix3x3& m)
2443     {
2444         for (int i = 0; i < 3; ++i)
2445         {
2446             for (int j = 0; j < 3; ++j)
2447             {
2448                 std::cout << m(i, j) << " ";
2449             }
2450
2451             std::cout << std::endl;
2452         }
2453     }
2454 #endif
2455
2456
2457 //-----
2458
2459
2460 //-----
2461
2462     class Matrix4x4
2463     {
2464     public:
2465
2466         Matrix4x4();
2467
2468         Matrix4x4(float a[][4]);
2469
2470         Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
2471
2472         Matrix4x4(const Matrix2x2& m);
2473
2474         Matrix4x4(const Matrix3x3& m);
2475
2476         Matrix4x4& operator=(const Matrix2x2& m);
2477
2478         Matrix4x4& operator=(const Matrix3x3& m);
2479
2480         float* Data();
2481
2482         const float* Data() const;
2483
2484         const float& operator()(unsigned int row, unsigned int col) const;
2485
2486         float& operator()(unsigned int row, unsigned int col);
2487
2488         Vector4D GetRow(unsigned int row) const;
2489
2490         Vector4D GetCol(unsigned int col) const;
2491
2492         void SetRow(unsigned int row, Vector4D v);
2493
2494         void SetCol(unsigned int col, Vector4D v);
2495
2496         Matrix4x4& operator+=(const Matrix4x4& m);
2497
2498         Matrix4x4& operator-=(const Matrix4x4& m);
2499
2500         Matrix4x4& operator*=(float k);
2501
2502         Matrix4x4& operator*=(const Matrix4x4& m);
2503
2504     private:
2505
2506         float mMat[4][4];
2507     };
2508
2509 //-----
2510
2511     inline Matrix4x4::Matrix4x4()
2512     {
2513         //1st row
2514         mMat[0][0] = 1.0f;
2515         mMat[0][1] = 0.0f;
2516         mMat[0][2] = 0.0f;

```

```

2588         mMat[0][3] = 0.0f;
2589
2590         //2nd
2591         mMat[1][0] = 0.0f;
2592         mMat[1][1] = 1.0f;
2593         mMat[1][2] = 0.0f;
2594         mMat[1][3] = 0.0f;
2595
2596         //3rd row
2597         mMat[2][0] = 0.0f;
2598         mMat[2][1] = 0.0f;
2599         mMat[2][2] = 1.0f;
2600         mMat[2][3] = 0.0f;
2601
2602         //4th row
2603         mMat[3][0] = 0.0f;
2604         mMat[3][1] = 0.0f;
2605         mMat[3][2] = 0.0f;
2606         mMat[3][3] = 1.0f;
2607     }
2608
2609
2610
2611     inline Matrix4x4::Matrix4x4(float a[][4])
2612     {
2613         //1st row
2614         mMat[0][0] = a[0][0];
2615         mMat[0][1] = a[0][1];
2616         mMat[0][2] = a[0][2];
2617         mMat[0][3] = a[0][3];
2618
2619         //2nd
2620         mMat[1][0] = a[1][0];
2621         mMat[1][1] = a[1][1];
2622         mMat[1][2] = a[1][2];
2623         mMat[1][3] = a[1][3];
2624
2625         //3rd row
2626         mMat[2][0] = a[2][0];
2627         mMat[2][1] = a[2][1];
2628         mMat[2][2] = a[2][2];
2629         mMat[2][3] = a[2][3];
2630
2631         //4th row
2632         mMat[3][0] = a[3][0];
2633         mMat[3][1] = a[3][1];
2634         mMat[3][2] = a[3][2];
2635         mMat[3][3] = a[3][3];
2636     }
2637
2638     inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
2639     {
2640         SetRow(0, r1);
2641         SetRow(1, r2);
2642         SetRow(2, r3);
2643         SetRow(3, r4);
2644     }
2645
2646     inline float* Matrix4x4::Data()
2647     {
2648         return mMat[0];
2649     }
2650
2651     inline const float* Matrix4x4::Data()const
2652     {
2653         return mMat[0];
2654     }
2655
2656     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
2657     {
2658         if (row > 3 || col > 3)
2659         {
2660             return mMat[0][0];
2661         }
2662         else
2663         {
2664             return mMat[row][col];
2665         }
2666     }
2667
2668     inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
2669     {
2670         if (row > 3 || col > 3)
2671         {
2672             return mMat[0][0];
2673         }

```

```

2674         else
2675         {
2676             return mMat[row][col];
2677         }
2678     }
2679
2680     inline Vector4D Matrix4x4::GetRow(unsigned int row) const
2681     {
2682         if (row < 0 || row > 3)
2683             return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
2684         else
2685             return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);
2686     }
2687
2688     inline Vector4D Matrix4x4::GetCol(unsigned int col) const
2689     {
2690         if (col < 0 || col > 3)
2691             return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
2692         else
2693             return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
2694     }
2695
2696     inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
2697     {
2698         if (row > 3)
2699         {
2700             mMat[0][0] = v.GetX();
2701             mMat[0][1] = v.GetY();
2702             mMat[0][2] = v.GetZ();
2703             mMat[0][3] = v.GetW();
2704         }
2705         else
2706         {
2707             mMat[row][0] = v.GetX();
2708             mMat[row][1] = v.GetY();
2709             mMat[row][2] = v.GetZ();
2710             mMat[row][3] = v.GetW();
2711         }
2712     }
2713
2714     inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
2715     {
2716         if (col > 3)
2717         {
2718             mMat[0][0] = v.GetX();
2719             mMat[1][0] = v.GetY();
2720             mMat[2][0] = v.GetZ();
2721             mMat[3][0] = v.GetW();
2722         }
2723         else
2724         {
2725             mMat[0][col] = v.GetX();
2726             mMat[1][col] = v.GetY();
2727             mMat[2][col] = v.GetZ();
2728             mMat[3][col] = v.GetW();
2729         }
2730     }
2731
2732     inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
2733     {
2734         for (int i = 0; i < 4; ++i)
2735         {
2736             for (int j = 0; j < 4; ++j)
2737             {
2738                 this->mMat[i][j] += m.mMat[i][j];
2739             }
2740         }
2741         return *this;
2742     }
2743
2744     inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
2745     {
2746         for (int i = 0; i < 4; ++i)
2747         {
2748             for (int j = 0; j < 4; ++j)
2749             {
2750                 this->mMat[i][j] -= m.mMat[i][j];
2751             }
2752         }
2753         return *this;
2754     }
2755
2756     inline Matrix4x4& Matrix4x4::operator*=(float k)
2757     {
2758
2759
2760

```



```

2761         for (int i = 0; i < 4; ++i)
2762         {
2763             for (int j = 0; j < 4; ++j)
2764             {
2765                 this->mMat[i][j] *= k;
2766             }
2767         }
2768         return *this;
2769     }
2770
2771     inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
2772     {
2773         Matrix4x4 result;
2774
2775         for (int i = 0; i < 4; ++i)
2776         {
2777             result.mMat[i][0] =
2778                 (mMat[i][0] * m.mMat[0][0]) +
2779                 (mMat[i][1] * m.mMat[1][0]) +
2780                 (mMat[i][2] * m.mMat[2][0]) +
2781                 (mMat[i][3] * m.mMat[3][0]);
2782
2783             result.mMat[i][1] =
2784                 (mMat[i][0] * m.mMat[0][1]) +
2785                 (mMat[i][1] * m.mMat[1][1]) +
2786                 (mMat[i][2] * m.mMat[2][1]) +
2787                 (mMat[i][3] * m.mMat[3][1]);
2788
2789             result.mMat[i][2] =
2790                 (mMat[i][0] * m.mMat[0][2]) +
2791                 (mMat[i][1] * m.mMat[1][2]) +
2792                 (mMat[i][2] * m.mMat[2][2]) +
2793                 (mMat[i][3] * m.mMat[3][2]);
2794
2795             result.mMat[i][3] =
2796                 (mMat[i][0] * m.mMat[0][3]) +
2797                 (mMat[i][1] * m.mMat[1][3]) +
2798                 (mMat[i][2] * m.mMat[2][3]) +
2799                 (mMat[i][3] * m.mMat[3][3]);
2800         }
2801
2802         for (int i = 0; i < 4; ++i)
2803         {
2804             for (int j = 0; j < 4; ++j)
2805             {
2806                 mMat[i][j] = result.mMat[i][j];
2807             }
2808         }
2809
2810         return *this;
2811     }
2812
2813     inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
2814     {
2815         Matrix4x4 result;
2816         for (int i = 0; i < 4; ++i)
2817         {
2818             for (int j = 0; j < 4; ++j)
2819             {
2820                 result(i, j) = m1(i, j) + m2(i, j);
2821             }
2822         }
2823
2824         return result;
2825     }
2826
2827     inline Matrix4x4 operator-(const Matrix4x4& m)
2828     {
2829         Matrix4x4 result;
2830         for (int i = 0; i < 4; ++i)
2831         {
2832             for (int j = 0; j < 4; ++j)
2833             {
2834                 result(i, j) = -m(i, j);
2835             }
2836         }
2837
2838         return result;
2839     }
2840
2841     inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
2842     {
2843         Matrix4x4 result;
2844         for (int i = 0; i < 4; ++i)
2845         {
2846             for (int j = 0; j < 4; ++j)

```

```

2854         {
2855             result(i, j) = m1(i, j) - m2(i, j);
2856         }
2857     }
2858
2859     return result;
2860 }
2861
2862 inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
2863 {
2864     Matrix4x4 result;
2865     for (int i = 0; i < 4; ++i)
2866     {
2867         for (int j = 0; j < 4; ++j)
2868         {
2869             result(i, j) = m(i, j) * k;
2870         }
2871     }
2872
2873     return result;
2874 }
2875
2876 inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
2877 {
2878     Matrix4x4 result;
2879     for (int i = 0; i < 4; ++i)
2880     {
2881         for (int j = 0; j < 4; ++j)
2882         {
2883             result(i, j) = k * m(i, j);
2884         }
2885     }
2886
2887     return result;
2888 }
2889
2890 inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
2891 {
2892     Matrix4x4 result;
2893
2894     for (int i = 0; i < 4; ++i)
2895     {
2896         result(i, 0) =
2897             (m1(i, 0) * m2(0, 0)) +
2898             (m1(i, 1) * m2(1, 0)) +
2899             (m1(i, 2) * m2(2, 0)) +
2900             (m1(i, 3) * m2(3, 0));
2901
2902         result(i, 1) =
2903             (m1(i, 0) * m2(0, 1)) +
2904             (m1(i, 1) * m2(1, 1)) +
2905             (m1(i, 2) * m2(2, 1)) +
2906             (m1(i, 3) * m2(3, 1));
2907
2908         result(i, 2) =
2909             (m1(i, 0) * m2(0, 2)) +
2910             (m1(i, 1) * m2(1, 2)) +
2911             (m1(i, 2) * m2(2, 2)) +
2912             (m1(i, 3) * m2(3, 2));
2913
2914         result(i, 3) =
2915             (m1(i, 0) * m2(0, 3)) +
2916             (m1(i, 1) * m2(1, 3)) +
2917             (m1(i, 2) * m2(2, 3)) +
2918             (m1(i, 3) * m2(3, 3));
2919     }
2920
2921     return result;
2922 }
2923
2924 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
2925 {
2926     Vector4D result;
2927
2928     result.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());
2929
2930     result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());
2931
2932     result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());
2933
2934     result.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());
2935
2936     return result;
2937 }
2938
2939 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
2940 {

```

```

2957     Vector4D result;
2958
2959     result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));
2960
2961     result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));
2962
2963     result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));
2964
2965     result.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));
2966
2967     return result;
2968 }
2969
2970 inline void SetToIdentity(Matrix4x4& m)
2971 {
2972     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
2973
2974     //1st row
2975     m(0, 0) = 1.0f;
2976     m(0, 1) = 0.0f;
2977     m(0, 2) = 0.0f;
2978     m(0, 3) = 0.0f;
2979
2980     //2nd row
2981     m(1, 0) = 0.0f;
2982     m(1, 1) = 1.0f;
2983     m(1, 2) = 0.0f;
2984     m(1, 3) = 0.0f;
2985
2986     //3rd row
2987     m(2, 0) = 0.0f;
2988     m(2, 1) = 0.0f;
2989     m(2, 2) = 1.0f;
2990     m(2, 3) = 0.0f;
2991
2992     //4th row
2993     m(3, 0) = 0.0f;
2994     m(3, 1) = 0.0f;
2995     m(3, 2) = 0.0f;
2996     m(3, 3) = 1.0f;
2997 }
2998
2999 inline bool IsIdentity(const Matrix4x4& m)
3000 {
3001     //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
3002     0.0f
3003
3004     for (int i = 0; i < 4; ++i)
3005     {
3006         for (int j = 0; j < 4; ++j)
3007         {
3008             if (i == j)
3009             {
3010                 if (!CompareFloats(m(i, j), 1.0f, EPSILON))
3011                     return false;
3012             }
3013             else
3014             {
3015                 if (!CompareFloats(m(i, j), 0.0f, EPSILON))
3016                     return false;
3017             }
3018         }
3019     }
3020 }
3021
3022 inline Matrix4x4 Transpose(const Matrix4x4& m)
3023 {
3024     //make the rows into cols
3025
3026     Matrix4x4 result;
3027
3028     //1st col = 1st row
3029     result(0, 0) = m(0, 0);
3030     result(1, 0) = m(0, 1);
3031     result(2, 0) = m(0, 2);
3032     result(3, 0) = m(0, 3);
3033
3034     //2nd col = 2nd row
3035     result(0, 1) = m(1, 0);
3036     result(1, 1) = m(1, 1);
3037     result(2, 1) = m(1, 2);
3038     result(3, 1) = m(1, 3);
3039
3040     //3rd col = 3rd row
3041     result(0, 2) = m(2, 0);

```

```

3049         result(1, 2) = m(2, 1);
3050         result(2, 2) = m(2, 2);
3051         result(3, 2) = m(2, 3);
3052
3053         //4th col = 4th row
3054         result(0, 3) = m(3, 0);
3055         result(1, 3) = m(3, 1);
3056         result(2, 3) = m(3, 2);
3057         result(3, 3) = m(3, 3);
3058
3059         return result;
3060     }
3061
3062     inline Matrix4x4 Translate(const Matrix4x4& cm, float x, float y, float z)
3063     {
3064         //1 0 0 0
3065         //0 1 0 0
3066         //0 0 1 0
3067         //x y z 1
3068
3069         Matrix4x4 translate;
3070         translate(3, 0) = x;
3071         translate(3, 1) = y;
3072         translate(3, 2) = z;
3073
3074         return cm * translate;
3075     }
3076
3077     inline Matrix4x4 Translate(const Matrix4x4& cm, const Vector3D& translateVector)
3078     {
3079         //1 0 0 0
3080         //0 1 0 0
3081         //0 0 1 0
3082         //x y z 1
3083
3084         Matrix4x4 translate;
3085         translate(3, 0) = translateVector.GetX();
3086         translate(3, 1) = translateVector.GetY();
3087         translate(3, 2) = translateVector.GetZ();
3088
3089         return cm * translate;
3090     }
3091
3092     inline Matrix4x4 Scale(const Matrix4x4& cm, float x, float y, float z)
3093     {
3094         //x 0 0 0
3095         //0 y 0 0
3096         //0 0 z 0
3097         //0 0 0 1
3098
3099         Matrix4x4 scale;
3100         scale(0, 0) = x;
3101         scale(1, 1) = y;
3102         scale(2, 2) = z;
3103
3104         return cm * scale;
3105     }
3106
3107     inline Matrix4x4 Scale(const Matrix4x4& cm, const Vector3D& scaleVector)
3108     {
3109         //x 0 0 0
3110         //0 y 0 0
3111         //0 0 z 0
3112         //0 0 0 1
3113
3114         Matrix4x4 scale;
3115         scale(0, 0) = scaleVector.GetX();
3116         scale(1, 1) = scaleVector.GetY();
3117         scale(2, 2) = scaleVector.GetZ();
3118
3119         return cm * scale;
3120     }
3121
3122     inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
3123     {
3124         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy    0
3125         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx    0
3126         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2    0
3127         //0                    0                    0                    1
3128
3129         //c = cos(angle)
3130         //s = sin(angle)
3131
3132         Vector3D axis(x, y, z);
3133
3134         axis = Norm(axis);
3135
3136         x = axis.GetX();

```

```

3156     y = axis.GetY();
3157     z = axis.GetZ();
3158
3159     float c = cos(angle * PI / 180.0f);
3160     float s = sin(angle * PI / 180.0f);
3161     float oneMinusC = 1 - c;
3162
3163     Matrix4x4 result;
3164
3165     //1st row
3166     result(0, 0) = c + (oneMinusC * (x * x));
3167     result(0, 1) = (oneMinusC * (x * y)) + (s * z);
3168     result(0, 2) = (oneMinusC * (x * z)) - (s * y);
3169
3170     //2nd row
3171     result(1, 0) = (oneMinusC * (x * y)) - (s * z);
3172     result(1, 1) = c + (oneMinusC * (y * y));
3173     result(1, 2) = (oneMinusC * (y * z)) + (s * x);
3174
3175     //3rd row
3176     result(2, 0) = (oneMinusC * (x * z)) + (s * y);
3177     result(2, 1) = (oneMinusC * (y * z)) - (s * x);
3178     result(2, 2) = c + (oneMinusC * (z * z));
3179
3180     return cm * result;
3181 }
3182
3183 inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, const Vector3D& axis)
3184 {
3185     //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy      0
3186     // (1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx      0
3187     // (1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2      0
3188     //0                    0                    0                    1
3189     //c = cos(angle)
3190     //s = sin(angle)
3191
3192     Vector3D nAxis(Norm(axis));
3193
3194     float x = nAxis.GetX();
3195     float y = nAxis.GetY();
3196     float z = nAxis.GetZ();
3197
3198     float c = cos(angle * PI / 180.0f);
3199     float s = sin(angle * PI / 180.0f);
3200     float oneMinusC = 1 - c;
3201
3202     Matrix4x4 result;
3203
3204     //1st row
3205     result(0, 0) = c + (oneMinusC * (x * x));
3206     result(0, 1) = (oneMinusC * (x * y)) + (s * z);
3207     result(0, 2) = (oneMinusC * (x * z)) - (s * y);
3208
3209     //2nd row
3210     result(1, 0) = (oneMinusC * (x * y)) - (s * z);
3211     result(1, 1) = c + (oneMinusC * (y * y));
3212     result(1, 2) = (oneMinusC * (y * z)) + (s * x);
3213
3214     //3rd row
3215     result(2, 0) = (oneMinusC * (x * z)) + (s * y);
3216     result(2, 1) = (oneMinusC * (y * z)) - (s * x);
3217     result(2, 2) = c + (oneMinusC * (z * z));
3218
3219     return cm * result;
3220 }
3221
3222 inline double Determinant(const Matrix4x4& m)
3223 {
3224     //m00m11(m22m33 - m23m32)
3225     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)
3226 * m(3, 2);
3227
3228     //m00m12(m23m31 - m21m33)
3229     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)
3230 * m(3, 3);
3231
3232     //m00m13(m21m32 - m22m31)
3233     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)
3234 * m(3, 1);
3235
3236     //m01m10(m22m33 - m23m32)
3237     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)
3238 * m(3, 2);
3239
3240     //m01m12(m23m30 - m20m33)
3241     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)
3242 * m(3, 3);

```

```

3244
3245 //m01m13(m20m32 - m22m30)
3246 double c6 = (double)m(0, 1) * m(1, 3) * m(2, 0) * m(3, 2) - (double)m(0, 1) * m(1, 3) * m(2, 2)
    * m(3, 0);
3247
3248 //m02m10(m21m33 - m23m31)
3249 double c7 = (double)m(0, 2) * m(1, 0) * m(2, 1) * m(3, 3) - (double)m(0, 2) * m(1, 0) * m(2, 3)
    * m(3, 1);
3250
3251 //m02m11(m23m30 - m20m33)
3252 double c8 = (double)m(0, 2) * m(1, 1) * m(2, 3) * m(3, 0) - (double)m(0, 2) * m(1, 1) * m(2, 0)
    * m(3, 3);
3253
3254 //m02m13(m20m31 - m21m30)
3255 double c9 = (double)m(0, 2) * m(1, 3) * m(2, 0) * m(3, 1) - (double)m(0, 2) * m(1, 3) * m(2, 1)
    * m(3, 0);
3256
3257 //m03m10(m21m32 - m22m21)
3258 double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
    2) * m(3, 1);
3259
3260 //m03m11(m22m30 - m20m32)
3261 double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
    0) * m(3, 2);
3262
3263 //m03m12(m20m31 - m21m30)
3264 double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
    1) * m(3, 0);
3265
3266 return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
3267 }
3268
3271 inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
3272 {
3273     //cij = (-1)^i + j * det of minor(i, j);
3274     Matrix3x3 minor;
3275     int r{ 0 };
3276     int c{ 0 };
3277
3278     //minor(i, j)
3279     for (int i = 0; i < 4; ++i)
3280     {
3281         if (i == row)
3282             continue;
3283
3284         for (int j = 0; j < 4; ++j)
3285         {
3286             if (j == col)
3287                 continue;
3288
3289             minor(r, c) = m(i, j);
3290             ++c;
3291
3292         }
3293         c = 0;
3294         ++r;
3295     }
3296
3297     return pow(-1, row + col) * Determinant(minor);
3298 }
3299
3300 inline Matrix4x4 Adjoint(const Matrix4x4& m)
3301 {
3302     //Cofactor of each ijth position put into matrix cA.
3303     //Adjoint is the tranposed matrix of cA.
3304     Matrix4x4 cofactorMatrix;
3305     for (int i = 0; i < 4; ++i)
3306     {
3307         for (int j = 0; j < 4; ++j)
3308         {
3309             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
3310         }
3311     }
3312
3313     return Transpose(cofactorMatrix);
3314 }
3315
3316 inline Matrix4x4 Inverse(const Matrix4x4& m)
3317 {
3318     //Inverse of m = adjoint of m / det of m
3319     double det = Determinant(m);
3320     if (CompareDoubles(det, 0.0, EPSILON))
3321         return Matrix4x4();
3322
3323     return Adjoint(m) * (1.0f / static_cast<float>(det));
3324 }
3325
3326
3327
3328
3329
3330
3331

```

```

3332
3333
3334 #if defined(_DEBUG)
3335     inline void print(const Matrix4x4& m)
3336     {
3337         for (int i = 0; i < 4; ++i)
3338         {
3339             for (int j = 0; j < 4; ++j)
3340             {
3341                 std::cout << m(i, j) << " ";
3342             }
3343             std::cout << std::endl;
3344         }
3345     }
3346 }
3347 #endif
3348
3349 //-----
3350
3351
3352
3353 //-----
3354
3355 class Quaternion
3356 {
3357 public:
3358     Quaternion(float scalar = 1.0f, float x = 0.0f, float y = 0.0f, float z = 0.0f);
3359
3360     Quaternion(float scalar, const Vector3D& v);
3361
3362     Quaternion(const Vector4D& v);
3363
3364     float GetScalar() const;
3365
3366     float GetX() const;
3367
3368     float GetY() const;
3369
3370     float GetZ() const;
3371
3372     Vector3D GetVector() const;
3373
3374     void SetScalar(float scalar);
3375
3376     void SetX(float x);
3377
3378     void SetY(float y);
3379
3380     void SetZ(float z);
3381
3382     void SetVector(const Vector3D& v);
3383
3384     Quaternion& operator+=(const Quaternion& q);
3385
3386     Quaternion& operator-=(const Quaternion& q);
3387
3388     Quaternion& operator*=(float k);
3389
3390     Quaternion& operator*=(const Quaternion& q);
3391
3392 private:
3393     float mScalar;
3394     float mX;
3395     float mY;
3396     float mZ;
3397 };
3398
3399 //-----
3400
3401 inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
3402     mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
3403 {}
3404
3405 inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
3406     mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
3407 {}
3408
3409 inline Quaternion::Quaternion(const Vector4D& v) :
3410     mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
3411 {}
3412
3413 inline float Quaternion::GetScalar()const
3414 {
3415     return mScalar;
3416 }
3417
3418
3419
3420

```

```

3461     inline float Quaternion::GetX() const
3462 {
3463     return mX;
3464 }
3465
3466     inline float Quaternion::GetY() const
3467 {
3468     return mY;
3469 }
3470
3471     inline float Quaternion::GetZ() const
3472 {
3473     return mZ;
3474 }
3475
3476     inline Vector3D Quaternion::GetVector() const
3477 {
3478     return Vector3D(mX, mY, mZ);
3479 }
3480
3481     inline void Quaternion::SetScalar(float scalar)
3482 {
3483     mScalar = scalar;
3484 }
3485
3486     inline void Quaternion::SetX(float x)
3487 {
3488     mX = x;
3489 }
3490
3491     inline void Quaternion::SetY(float y)
3492 {
3493     mY = y;
3494 }
3495
3496     inline void Quaternion::SetZ(float z)
3497 {
3498     mZ = z;
3499 }
3500
3501     inline void Quaternion::SetVector(const Vector3D& v)
3502 {
3503     mX = v.GetX();
3504     mY = v.GetY();
3505     mZ = v.GetZ();
3506 }
3507
3508     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
3509 {
3510     this->mScalar += q.mScalar;
3511     this->mX += q.mX;
3512     this->mY += q.mY;
3513     this->mZ += q.mZ;
3514
3515     return *this;
3516 }
3517
3518     inline Quaternion& Quaternion::operator-=(const Quaternion& q)
3519 {
3520     this->mScalar -= q.mScalar;
3521     this->mX -= q.mX;
3522     this->mY -= q.mY;
3523     this->mZ -= q.mZ;
3524
3525     return *this;
3526 }
3527
3528     inline Quaternion& Quaternion::operator*=(float k)
3529 {
3530     this->mScalar *= k;
3531     this->mX *= k;
3532     this->mY *= k;
3533     this->mZ *= k;
3534
3535     return *this;
3536 }
3537
3538     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
3539 {
3540     Vector3D thisVector(this->mX, this->mY, this->mZ);
3541     Vector3D qVector(q.mX, q.mY, q.mZ);
3542
3543     float scalar{ this->mScalar * q.mScalar };
3544     float dotProduct{ DotProduct(thisVector, qVector) };
3545     float resultScalar{ scalar - dotProduct };
3546
3547     Vector3D a(this->mScalar * qVector);

```



```

3548     Vector3D b(q.mScalar * thisVector);
3549     Vector3D crossProduct(CrossProduct(thisVector, qVector));
3550     Vector3D resultVector(a + b + crossProduct);
3551
3552     this->mScalar = resultScalar;
3553     this->mX = resultVector.GetX();
3554     this->mY = resultVector.GetY();
3555     this->mZ = resultVector.GetZ();
3556
3557     return *this;
3558 }
3559
3560 inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
3561 {
3562     return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
3563 q2.GetY(), q1.GetZ() + q2.GetZ());
3564 }
3565
3566 inline Quaternion operator-(const Quaternion& q)
3567 {
3568     return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
3569 }
3570
3571 inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
3572 {
3573     return Quaternion(q1.GetScalar() - q2.GetScalar(),
3574 q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
3575 }
3576
3577 inline Quaternion operator*(float k, const Quaternion& q)
3578 {
3579     return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
3580 }
3581
3582 inline Quaternion operator*(const Quaternion& q, float k)
3583 {
3584     return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
3585 }
3586
3587 inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
3588 {
3589     //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
3590     //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
3591
3592     Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
3593     Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());
3594
3595     float scalar{ q1.GetScalar() * q2.GetScalar() };
3596     float dotProduct{ DotProduct(q1Vector, q2Vector) };
3597     float resultScalar{ scalar - dotProduct };
3598
3599     Vector3D a(q1.GetScalar() * q2Vector);
3600     Vector3D b(q2.GetScalar() * q1Vector);
3601     Vector3D crossProduct(CrossProduct(q1Vector, q2Vector));
3602     Vector3D resultVector(a + b + crossProduct);
3603
3604     return Quaternion(resultScalar, resultVector);
3605 }
3606
3607 inline bool IsZeroQuaternion(const Quaternion& q)
3608 {
3609     //zero quaternion = (0, 0, 0, 0)
3610     return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
3611 CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
3612 }
3613
3614 inline bool IsIdentity(const Quaternion& q)
3615 {
3616     //identity quaternion = (1, 0, 0, 0)
3617     return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
3618 CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
3619 }
3620
3621 inline Quaternion Conjugate(const Quaternion& q)
3622 {
3623     //conjugate of a quaternion is the quaternion with its vector part negated
3624     return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
3625 }
3626
3627 inline float Length(const Quaternion& q)
3628 {
3629     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
3630     return sqrt(q.GetScalar() * q.GetScalar() + q.GetX() * q.GetX() + q.GetY() * q.GetY() +
3631 q.GetZ() * q.GetZ());
3632 }
3633
3634 inline Quaternion Normalize(const Quaternion& q)

```

```

3657     {
3658         //to normalize a quaternion you do q / |q|
3659
3660         if (IsZeroQuaternion(q))
3661             return q;
3662
3663         float magnitdue{ Length(q) };
3664
3665         return Quaternion(q.GetScalar() / magnitdue, q.GetX() / magnitdue, q.GetY() / magnitdue,
3666             q.GetZ() / magnitdue);
3667     }
3668
3669     inline Quaternion Inverse(const Quaternion& q)
3670     {
3671         //inverse = conjugate of q / |q|
3672
3673         if (IsZeroQuaternion(q))
3674             return q;
3675
3676         Quaternion conjugateOfQ(Conjugate(q));
3677
3678         float magnitdue{ Length(q) };
3679
3680         return Quaternion(conjugateOfQ.GetScalar() / magnitdue, conjugateOfQ.GetX() / magnitdue,
3681             conjugateOfQ.GetY() / magnitdue, conjugateOfQ.GetZ() / magnitdue);
3682     }
3683
3684     inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
3685     {
3686         //A roatation quaternion is a quaternion where the
3687         //scalar part = cos(theta / 2)
3688         //vector part = sin(theta / 2) * axis
3689         //the axis needs to be normalized
3690
3691         float ang{ angle / 2.0f };
3692         float c{ cos(ang * PI / 180.0f) };
3693         float s{ sin(ang * PI / 180.0f) };
3694
3695         Vector3D axis(x, y, z);
3696         axis = Norm(axis);
3697
3698         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
3699     }
3700
3701     inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
3702     {
3703         //A roatation quaternion is a quaternion where the
3704         //scalar part = cos(theta / 2)
3705         //vector part = sin(theta / 2) * axis
3706         //the axis needs to be normalized
3707
3708         float ang{ angle / 2.0f };
3709         float c{ cos(ang * PI / 180.0f) };
3710         float s{ sin(ang * PI / 180.0f) };
3711
3712         Vector3D axisN(Norm(axis));
3713
3714         return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
3715     }
3716
3717     inline Quaternion RotationQuaternion(const Vector4D& angAxis)
3718     {
3719         //A roatation quaternion is a quaternion where the
3720         //scalar part = cos(theta / 2)
3721         //vector part = sin(theta / 2) * axis
3722         //the axis needs to be normalized
3723
3724         float angle{ angAxis.GetX() / 2.0f };
3725         float c{ cos(angle * PI / 180.0f) };
3726         float s{ sin(angle * PI / 180.0f) };
3727
3728         Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
3729         axis = Norm(axis);
3730
3731         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
3732     }
3733
3734     inline Matrix4x4 QuaternionToRotationMatrixCol(const Quaternion& q)
3735     {
3736         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
3737         //2q2q3 + 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
3738         //2q2q4 - 2q1q3      2q3q4 + 2q1q2      1 - 2q2^2 - 2q3^2      0
3739         //0                      0                      0                      1
3740         //q1 = scalar
3741         //q2 = x
3742         //q3 = y
3743         //q4 = z

```

```

3764
3765     Matrix4x4 colMat;
3766
3767     colMat(0, 0) = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
3768     colMat(0, 1) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
3769     colMat(0, 2) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
3770
3771     colMat(1, 0) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
3772     colMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
3773     colMat(1, 2) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
3774
3775     colMat(2, 0) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
3776     colMat(2, 1) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
3777     colMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
3778
3779     return colMat;
3780 }
3781
3782 inline Matrix4x4 QuaternionToRotationMatrixRow(const Quaternion& q)
3783 {
3784     //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
3785     //2q2q3 - 2q1q4        1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
3786     //2q2q4 + 2q1q3        2q3q4 - 2q1q2        1 - 2q2^2 - 2q3^2      0
3787     //0                    0                    0                    1
3788     //q1 = scalar
3789     //q2 = x
3790     //q3 = y
3791     //q4 = z
3792
3793     Matrix4x4 rowMat;
3794
3795     rowMat(0, 0) = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
3796     rowMat(0, 1) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
3797     rowMat(0, 2) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
3798
3799     rowMat(1, 0) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
3800     rowMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
3801     rowMat(1, 2) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
3802
3803     rowMat(2, 0) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
3804     rowMat(2, 1) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
3805     rowMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
3806
3807     return rowMat;
3808 }
3809
3810 inline Vector3D Rotate(const Quaternion& q, const Vector3D& p)
3811 {
3812     //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
3813     point/vector, q is a rotation quaternion
3814     //and q* is its conjugate.
3815
3816     Quaternion point(0.0f, p);
3817
3818     Quaternion result(q * point * Conjugate(q));
3819
3820     return result.GetVector();
3821 }
3822
3823 inline Vector4D Rotate(const Quaternion& q, const Vector4D& p)
3824 {
3825     //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
3826     point/vector, q is a rotation quaternion
3827     //and q* is its conjugate.
3828
3829     Quaternion point(0.0f, p);
3830
3831     Quaternion result(q * point * Conjugate(q));
3832
3833     return Vector4D(result.GetVector(), p.GetW());
3834 }
3835
3836 #if defined(_DEBUG)
3837 inline void print(const Quaternion& q)
3838 {
3839     std::cout << "(" << q.GetScalar() << ", " << q.GetX() << ", " << q.GetY() << ", " << q.GetZ();
3840 }
3841 #endif
3842
3843 //-----
3844
3845 inline Vector2D::Vector2D(const Vector3D& v) : mX{ v.GetX() }, mY{ v.GetY() }
3846 {}
3847
3848 inline Vector2D::Vector2D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }
3849 {}
3850
3851 inline Vector2D& Vector2D::operator=(const Vector3D& v)

```

```

3861     {
3862         mX = v.GetX();
3863         mY = v.GetY();
3864
3865         return *this;
3866     }
3867
3868     inline Vector2D& Vector2D::operator=(const Vector4D& v)
3869     {
3870         mX = v.GetX();
3871         mY = v.GetY();
3872
3873         return *this;
3874     }
3875
3876     inline Vector3D::Vector3D(const Vector2D& v, float z) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ z }
3877     {}
3878
3879     inline Vector3D::Vector3D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
3880     {}
3881
3882     inline Vector3D& Vector3D::operator=(const Vector2D& v)
3883     {
3884         mX = v.GetX();
3885         mY = v.GetY();
3886         mZ = 0.0f;
3887
3888         return *this;
3889     }
3890
3891     inline Vector3D& Vector3D::operator=(const Vector4D& v)
3892     {
3893         mX = v.GetX();
3894         mY = v.GetY();
3895         mZ = v.GetZ();
3896
3897         return *this;
3898     }
3899
3900     inline Vector4D::Vector4D(const Vector2D& v, float z, float w) : mX{ v.GetX() }, mY{ v.GetY() },
3901     mZ{ z }, mW{ w }
3902     {}
3903
3904     inline Vector4D::Vector4D(const Vector3D& v, float w) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{
3905     v.GetZ() }, mW{ w }
3906     {}
3907
3908     inline Vector4D& Vector4D::operator=(const Vector2D& v)
3909     {
3910         mX = v.GetX();
3911         mY = v.GetY();
3912         mZ = 0.0f;
3913         mW = 0.0f;
3914
3915         return *this;
3916     }
3917
3918     inline Vector4D& Vector4D::operator=(const Vector3D& v)
3919     {
3920         mX = v.GetX();
3921         mY = v.GetY();
3922         mZ = v.GetZ();
3923         mW = 0.0f;
3924
3925         return *this;
3926     }
3927
3928     inline Matrix2x2::Matrix2x2(const Matrix3x3& m)
3929     {
3930         //1st row
3931         mMat[0][0] = m(0, 0);
3932         mMat[0][1] = m(0, 1);
3933
3934         //2nd row
3935         mMat[1][0] = m(1, 0);
3936         mMat[1][1] = m(1, 1);
3937     }
3938
3939     inline Matrix2x2::Matrix2x2(const Matrix4x4& m)
3940     {
3941         //1st row
3942         mMat[0][0] = m(0, 0);
3943         mMat[0][1] = m(0, 1);
3944
3945         //2nd row
3946         mMat[1][0] = m(1, 0);
3947         mMat[1][1] = m(1, 1);

```

```

3946     }
3947
3948     inline Matrix2x2& Matrix2x2::operator=(const Matrix3x3& m)
3949     {
3950         //1st row
3951         mMat[0][0] = m(0, 0);
3952         mMat[0][1] = m(0, 1);
3953
3954         //2nd row
3955         mMat[1][0] = m(1, 0);
3956         mMat[1][1] = m(1, 1);
3957
3958         return *this;
3959     }
3960
3961     inline Matrix2x2& Matrix2x2::operator=(const Matrix4x4& m)
3962     {
3963         //1st row
3964         mMat[0][0] = m(0, 0);
3965         mMat[0][1] = m(0, 1);
3966
3967         //2nd row
3968         mMat[1][0] = m(1, 0);
3969         mMat[1][1] = m(1, 1);
3970
3971         return *this;
3972     }
3973
3974     inline Matrix3x3::Matrix3x3(const Matrix2x2& m)
3975     {
3976         //1st row
3977         mMat[0][0] = m(0, 0);
3978         mMat[0][1] = m(0, 1);
3979         mMat[0][2] = 0.0f;
3980
3981         //2nd row
3982         mMat[1][0] = m(1, 0);
3983         mMat[1][1] = m(1, 1);
3984         mMat[1][2] = 0.0f;
3985
3986         //3rd row
3987         mMat[2][0] = 0.0f;
3988         mMat[2][1] = 0.0f;
3989         mMat[2][2] = 1.0f;
3990     }
3991
3992     inline Matrix3x3::Matrix3x3(const Matrix4x4& m)
3993     {
3994         //1st row
3995         mMat[0][0] = m(0, 0);
3996         mMat[0][1] = m(0, 1);
3997         mMat[0][2] = m(0, 2);
3998
3999         //2nd row
4000         mMat[1][0] = m(1, 0);
4001         mMat[1][1] = m(1, 1);
4002         mMat[1][2] = m(1, 2);
4003
4004         //3rd row
4005         mMat[2][0] = m(2, 0);
4006         mMat[2][1] = m(2, 1);
4007         mMat[2][2] = m(2, 2);
4008     }
4009
4010     inline Matrix3x3& Matrix3x3::operator=(const Matrix2x2& m)
4011     {
4012         //1st row
4013         mMat[0][0] = m(0, 0);
4014         mMat[0][1] = m(0, 1);
4015         mMat[0][2] = 0.0f;
4016
4017         //2nd row
4018         mMat[1][0] = m(1, 0);
4019         mMat[1][1] = m(1, 1);
4020         mMat[1][2] = 0.0f;
4021
4022         //3rd row
4023         mMat[2][0] = 0.0f;
4024         mMat[2][1] = 0.0f;
4025         mMat[2][2] = 1.0f;
4026
4027         return *this;
4028     }
4029
4030     inline Matrix3x3& Matrix3x3::operator=(const Matrix4x4& m)
4031     {
4032         //1st row

```

```

4033         mMat[0][0] = m(0, 0);
4034         mMat[0][1] = m(0, 1);
4035         mMat[0][2] = m(0, 2);
4036
4037         //2nd row
4038         mMat[1][0] = m(1, 0);
4039         mMat[1][1] = m(1, 1);
4040         mMat[1][2] = m(1, 2);
4041
4042         //3rd row
4043         mMat[2][0] = m(2, 0);
4044         mMat[2][1] = m(2, 1);
4045         mMat[2][2] = m(2, 2);
4046
4047         return *this;
4048     }
4049
4050     inline Matrix4x4::Matrix4x4(const Matrix2x2& m)
4051     {
4052         //1st row
4053         mMat[0][0] = m(0, 0);
4054         mMat[0][1] = m(0, 1);
4055         mMat[0][2] = 0.0f;
4056         mMat[0][3] = 0.0f;
4057
4058         //2nd row
4059         mMat[1][0] = m(1, 0);
4060         mMat[1][1] = m(1, 1);
4061         mMat[1][2] = 0.0f;
4062         mMat[1][3] = 0.0f;
4063
4064         //3rd row
4065         mMat[2][0] = 0.0f;
4066         mMat[2][1] = 0.0f;
4067         mMat[2][2] = 1.0f;
4068         mMat[2][3] = 0.0f;
4069
4070         //4th row
4071         mMat[3][0] = 0.0f;
4072         mMat[3][1] = 0.0f;
4073         mMat[3][2] = 0.0f;
4074         mMat[3][3] = 1.0f;
4075     }
4076
4077     inline Matrix4x4::Matrix4x4(const Matrix3x3& m)
4078     {
4079         //1st row
4080         mMat[0][0] = m(0, 0);
4081         mMat[0][1] = m(0, 1);
4082         mMat[0][2] = m(0, 2);
4083         mMat[0][3] = 0.0f;
4084
4085         //2nd row
4086         mMat[1][0] = m(1, 0);
4087         mMat[1][1] = m(1, 1);
4088         mMat[1][2] = m(1, 2);
4089         mMat[1][3] = 0.0f;
4090
4091         //3rd row
4092         mMat[2][0] = m(2, 0);
4093         mMat[2][1] = m(2, 1);
4094         mMat[2][2] = m(2, 2);
4095         mMat[2][3] = 0.0f;
4096
4097         //4th row
4098         mMat[3][0] = 0.0f;
4099         mMat[3][1] = 0.0f;
4100         mMat[3][2] = 0.0f;
4101         mMat[3][3] = 1.0f;
4102     }
4103
4104     inline Matrix4x4& Matrix4x4::operator=(const Matrix2x2& m)
4105     {
4106         //1st row
4107         mMat[0][0] = m(0, 0);
4108         mMat[0][1] = m(0, 1);
4109         mMat[0][2] = 0.0f;
4110         mMat[0][3] = 0.0f;
4111
4112         //2nd row
4113         mMat[1][0] = m(1, 0);
4114         mMat[1][1] = m(1, 1);
4115         mMat[1][2] = 0.0f;
4116         mMat[1][3] = 0.0f;
4117
4118         //3rd row
4119         mMat[2][0] = 0.0f;

```

```
4120         mMat[2][1] = 0.0f;
4121         mMat[2][2] = 1.0f;
4122         mMat[2][3] = 0.0f;
4123
4124         //4th row
4125         mMat[3][0] = 0.0f;
4126         mMat[3][1] = 0.0f;
4127         mMat[3][2] = 0.0f;
4128         mMat[3][3] = 1.0f;
4129
4130         return *this;
4131     }
4132
4133     inline Matrix4x4& Matrix4x4::operator=(const Matrix3x3& m)
4134     {
4135         //1st row
4136         mMat[0][0] = m(0, 0);
4137         mMat[0][1] = m(0, 1);
4138         mMat[0][2] = m(0, 2);
4139         mMat[0][3] = 0.0f;
4140
4141         //2nd row
4142         mMat[1][0] = m(1, 0);
4143         mMat[1][1] = m(1, 1);
4144         mMat[1][2] = m(1, 2);
4145         mMat[1][3] = 0.0f;
4146
4147         //3rd row
4148         mMat[2][0] = m(2, 0);
4149         mMat[2][1] = m(2, 1);
4150         mMat[2][2] = m(2, 2);
4151         mMat[2][3] = 0.0f;
4152
4153         //4th row
4154         mMat[3][0] = 0.0f;
4155         mMat[3][1] = 0.0f;
4156         mMat[3][2] = 0.0f;
4157         mMat[3][3] = 1.0f;
4158
4159         return *this;
4160     }
4161
4162     //-----
4162 }
```





# Index

- Adjoint
  - FAMath, [12](#), [13](#)
- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h, [69](#)
- CartesianToCylindrical
  - FAMath, [13](#)
- CartesianToPolar
  - FAMath, [13](#)
- CartesianToSpherical
  - FAMath, [13](#)
- Cofactor
  - FAMath, [13](#), [14](#)
- CompareDoubles
  - FAMath, [14](#)
- CompareFloats
  - FAMath, [14](#)
- Conjugate
  - FAMath, [14](#)
- CrossProduct
  - FAMath, [15](#)
- CylindricalToCartesian
  - FAMath, [15](#)
- Data
  - FAMath::Matrix2x2, [39](#)
  - FAMath::Matrix3x3, [44](#)
  - FAMath::Matrix4x4, [49](#)
- Determinant
  - FAMath, [15](#)
- DotProduct
  - FAMath, [16](#)
- FAMath, [7](#)
  - Adjoint, [12](#), [13](#)
  - CartesianToCylindrical, [13](#)
  - CartesianToPolar, [13](#)
  - CartesianToSpherical, [13](#)
  - Cofactor, [13](#), [14](#)
  - CompareDoubles, [14](#)
  - CompareFloats, [14](#)
  - Conjugate, [14](#)
  - CrossProduct, [15](#)
  - CylindricalToCartesian, [15](#)
  - Determinant, [15](#)
  - DotProduct, [16](#)
  - Inverse, [16](#), [17](#)
  - IsIdentity, [17](#)
  - IsZeroQuaternion, [18](#)
  - Length, [18](#)
  - Norm, [18](#), [19](#)
  - Normalize, [19](#)
  - operator\*, [19–24](#)
  - operator+, [24](#), [25](#)
  - operator-, [25–28](#)
  - operator/, [28](#), [29](#)
  - Orthonormalize, [29](#)
  - PolarToCartesian, [29](#)
  - Projection, [29](#), [30](#)
  - QuaternionToRotationMatrixCol, [30](#)
  - QuaternionToRotationMatrixRow, [30](#)
  - Rotate, [30–32](#)
  - RotationQuaternion, [32](#)
  - Scale, [32](#), [33](#)
  - SetToIdentity, [34](#)
  - SphericalToCartesian, [34](#)
  - Translate, [34](#), [35](#)
  - Transpose, [35](#)
  - ZeroVector, [35](#), [36](#)
- FAMath::Matrix2x2, [37](#)
  - Data, [39](#)
  - GetCol, [39](#)
  - GetRow, [39](#)
  - Matrix2x2, [38](#), [39](#)
  - operator\*=, [40](#)
  - operator(), [40](#)
  - operator+=, [40](#)
  - operator-=, [41](#)
  - operator=, [41](#)
  - SetCol, [41](#)
  - SetRow, [41](#)
- FAMath::Matrix3x3, [42](#)
  - Data, [44](#)
  - GetCol, [44](#)
  - GetRow, [44](#)
  - Matrix3x3, [43](#), [44](#)
  - operator\*=, [45](#)
  - operator(), [45](#)
  - operator+=, [45](#)
  - operator-=, [46](#)
  - operator=, [46](#)
  - SetCol, [46](#)
  - SetRow, [46](#)
- FAMath::Matrix4x4, [47](#)
  - Data, [49](#)
  - GetCol, [49](#)
  - GetRow, [49](#)
  - Matrix4x4, [48](#), [49](#)

- operator\*=, 50
- operator(), 50
- operator+=, 50
- operator-=, 51
- operator=, 51
- SetCol, 51
- SetRow, 51
- FAMath::Quaternion, 52
  - GetScalar, 53
  - GetVector, 54
  - GetX, 54
  - GetY, 54
  - GetZ, 54
  - operator\*=, 54
  - operator+=, 55
  - operator-=, 55
  - Quaternion, 53
  - SetScalar, 55
  - SetVector, 55
  - SetX, 55
  - SetY, 55
  - SetZ, 56
- FAMath::Vector2D, 56
  - GetX, 57
  - GetY, 57
  - operator\*=, 58
  - operator+=, 58
  - operator-=, 58
  - operator/=: 58
  - operator=, 58
  - SetX, 59
  - SetY, 59
  - Vector2D, 57
- FAMath::Vector3D, 59
  - GetX, 61
  - GetY, 61
  - GetZ, 61
  - operator\*=, 62
  - operator+=, 62
  - operator-=, 62
  - operator/=: 62
  - operator=, 62
  - SetX, 63
  - SetY, 63
  - SetZ, 63
  - Vector3D, 60, 61
- FAMath::Vector4D, 63
  - GetW, 65
  - GetX, 65
  - GetY, 65
  - GetZ, 66
  - operator\*=, 66
  - operator+=, 66
  - operator-=, 66
  - operator/=: 66
  - operator=, 66, 67
  - SetW, 67
  - SetX, 67
  - SetY, 67
  - SetZ, 67
  - Vector4D, 64, 65
- GetCol
  - FAMath::Matrix2x2, 39
  - FAMath::Matrix3x3, 44
  - FAMath::Matrix4x4, 49
- GetRow
  - FAMath::Matrix2x2, 39
  - FAMath::Matrix3x3, 44
  - FAMath::Matrix4x4, 49
- GetScalar
  - FAMath::Quaternion, 53
- GetVector
  - FAMath::Quaternion, 54
- GetW
  - FAMath::Vector4D, 65
- GetX
  - FAMath::Quaternion, 54
  - FAMath::Vector2D, 57
  - FAMath::Vector3D, 61
  - FAMath::Vector4D, 65
- GetY
  - FAMath::Quaternion, 54
  - FAMath::Vector2D, 57
  - FAMath::Vector3D, 61
  - FAMath::Vector4D, 65
- GetZ
  - FAMath::Quaternion, 54
  - FAMath::Vector3D, 61
  - FAMath::Vector4D, 66
- Inverse
  - FAMath, 16, 17
- IsIdentity
  - FAMath, 17
- IsZeroQuaternion
  - FAMath, 18
- Length
  - FAMath, 18
- Matrix2x2
  - FAMath::Matrix2x2, 38, 39
- Matrix3x3
  - FAMath::Matrix3x3, 43, 44
- Matrix4x4
  - FAMath::Matrix4x4, 48, 49
- Norm
  - FAMath, 18, 19
- Normalize
  - FAMath, 19
- operator\*
  - FAMath, 19–24
- operator\*=
  - FAMath::Matrix2x2, 40
  - FAMath::Matrix3x3, 45

- FAMath::Matrix4x4, [50](#)
- FAMath::Quaternion, [54](#)
- FAMath::Vector2D, [58](#)
- FAMath::Vector3D, [62](#)
- FAMath::Vector4D, [66](#)
- operator()
  - FAMath::Matrix2x2, [40](#)
  - FAMath::Matrix3x3, [45](#)
  - FAMath::Matrix4x4, [50](#)
- operator+
  - FAMath, [24, 25](#)
- operator+=
  - FAMath::Matrix2x2, [40](#)
  - FAMath::Matrix3x3, [45](#)
  - FAMath::Matrix4x4, [50](#)
  - FAMath::Quaternion, [55](#)
  - FAMath::Vector2D, [58](#)
  - FAMath::Vector3D, [62](#)
  - FAMath::Vector4D, [66](#)
- operator-
  - FAMath, [25–28](#)
- operator-=
  - FAMath::Matrix2x2, [41](#)
  - FAMath::Matrix3x3, [46](#)
  - FAMath::Matrix4x4, [51](#)
  - FAMath::Quaternion, [55](#)
  - FAMath::Vector2D, [58](#)
  - FAMath::Vector3D, [62](#)
  - FAMath::Vector4D, [66](#)
- operator/
  - FAMath, [28, 29](#)
- operator/=
  - FAMath::Vector2D, [58](#)
  - FAMath::Vector3D, [62](#)
  - FAMath::Vector4D, [66](#)
- operator=
  - FAMath::Matrix2x2, [41](#)
  - FAMath::Matrix3x3, [46](#)
  - FAMath::Matrix4x4, [51](#)
  - FAMath::Vector2D, [58](#)
  - FAMath::Vector3D, [62](#)
  - FAMath::Vector4D, [66, 67](#)
- Orthonormalize
  - FAMath, [29](#)
- PolarToCartesian
  - FAMath, [29](#)
- Projection
  - FAMath, [29, 30](#)
- Quaternion
  - FAMath::Quaternion, [53](#)
- QuaternionToRotationMatrixCol
  - FAMath, [30](#)
- QuaternionToRotationMatrixRow
  - FAMath, [30](#)
- Rotate
  - FAMath, [30–32](#)
- RotationQuaternion
  - FAMath, [32](#)
- Scale
  - FAMath, [32, 33](#)
- SetCol
  - FAMath::Matrix2x2, [41](#)
  - FAMath::Matrix3x3, [46](#)
  - FAMath::Matrix4x4, [51](#)
- SetRow
  - FAMath::Matrix2x2, [41](#)
  - FAMath::Matrix3x3, [46](#)
  - FAMath::Matrix4x4, [51](#)
- SetScalar
  - FAMath::Quaternion, [55](#)
- SetToIdentity
  - FAMath, [34](#)
- SetVector
  - FAMath::Quaternion, [55](#)
- SetW
  - FAMath::Vector4D, [67](#)
- SetX
  - FAMath::Quaternion, [55](#)
  - FAMath::Vector2D, [59](#)
  - FAMath::Vector3D, [63](#)
  - FAMath::Vector4D, [67](#)
- SetY
  - FAMath::Quaternion, [55](#)
  - FAMath::Vector2D, [59](#)
  - FAMath::Vector3D, [63](#)
  - FAMath::Vector4D, [67](#)
- SetZ
  - FAMath::Quaternion, [56](#)
  - FAMath::Vector3D, [63](#)
  - FAMath::Vector4D, [67](#)
- SphericalToCartesian
  - FAMath, [34](#)
- Translate
  - FAMath, [34, 35](#)
- Transpose
  - FAMath, [35](#)
- Vector2D
  - FAMath::Vector2D, [57](#)
- Vector3D
  - FAMath::Vector3D, [60, 61](#)
- Vector4D
  - FAMath::Vector4D, [64, 65](#)
- ZeroVector
  - FAMath, [35, 36](#)