

Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4

1 Namespace Index	1
1.1 Namespace List	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Namespace Documentation	7
4.1 FAMath Namespace Reference	7
4.1.1 Detailed Description	13
4.1.2 Function Documentation	13
4.1.2.1 Adjoint() [1/3]	13
4.1.2.2 Adjoint() [2/3]	13
4.1.2.3 Adjoint() [3/3]	13
4.1.2.4 CartesianToCylindrical()	14
4.1.2.5 CartesianToPolar()	14
4.1.2.6 CartesianToSpherical()	14
4.1.2.7 Cofactor() [1/3]	14
4.1.2.8 Cofactor() [2/3]	14
4.1.2.9 Cofactor() [3/3]	15
4.1.2.10 CompareDoubles()	15
4.1.2.11 CompareFloats()	15
4.1.2.12 Conjugate()	15
4.1.2.13 CrossProduct()	15
4.1.2.14 CylindricalToCartesian()	16
4.1.2.15 Determinant() [1/3]	16
4.1.2.16 Determinant() [2/3]	16
4.1.2.17 Determinant() [3/3]	16
4.1.2.18 DotProduct() [1/4]	16
4.1.2.19 DotProduct() [2/4]	17
4.1.2.20 DotProduct() [3/4]	17
4.1.2.21 DotProduct() [4/4]	17
4.1.2.22 Inverse() [1/4]	17
4.1.2.23 Inverse() [2/4]	17
4.1.2.24 Inverse() [3/4]	18
4.1.2.25 Inverse() [4/4]	18
4.1.2.26 IsIdentity() [1/4]	18
4.1.2.27 IsIdentity() [2/4]	18
4.1.2.28 IsIdentity() [3/4]	18
4.1.2.29 IsIdentity() [4/4]	18
4.1.2.30 IsZeroQuaternion()	19

4.1.2.31 Length() [1/4]	19
4.1.2.32 Length() [2/4]	19
4.1.2.33 Length() [3/4]	19
4.1.2.34 Length() [4/4]	19
4.1.2.35 Lerp() [1/4]	20
4.1.2.36 Lerp() [2/4]	20
4.1.2.37 Lerp() [3/4]	20
4.1.2.38 Lerp() [4/4]	20
4.1.2.39 NLERP()	20
4.1.2.40 Norm() [1/3]	21
4.1.2.41 Norm() [2/3]	21
4.1.2.42 Norm() [3/3]	21
4.1.2.43 Normalize()	21
4.1.2.44 operator!=() [1/4]	21
4.1.2.45 operator!=() [2/4]	22
4.1.2.46 operator!=() [3/4]	22
4.1.2.47 operator!=() [4/4]	22
4.1.2.48 operator*() [1/24]	22
4.1.2.49 operator*() [2/24]	22
4.1.2.50 operator*() [3/24]	23
4.1.2.51 operator*() [4/24]	23
4.1.2.52 operator*() [5/24]	23
4.1.2.53 operator*() [6/24]	23
4.1.2.54 operator*() [7/24]	23
4.1.2.55 operator*() [8/24]	24
4.1.2.56 operator*() [9/24]	24
4.1.2.57 operator*() [10/24]	24
4.1.2.58 operator*() [11/24]	24
4.1.2.59 operator*() [12/24]	24
4.1.2.60 operator*() [13/24]	25
4.1.2.61 operator*() [14/24]	25
4.1.2.62 operator*() [15/24]	25
4.1.2.63 operator*() [16/24]	25
4.1.2.64 operator*() [17/24]	25
4.1.2.65 operator*() [18/24]	26
4.1.2.66 operator*() [19/24]	26
4.1.2.67 operator*() [20/24]	26
4.1.2.68 operator*() [21/24]	26
4.1.2.69 operator*() [22/24]	26
4.1.2.70 operator*() [23/24]	27
4.1.2.71 operator*() [24/24]	27
4.1.2.72 operator+() [1/7]	27

4.1.2.73 operator+() [2/7]	27
4.1.2.74 operator+() [3/7]	27
4.1.2.75 operator+() [4/7]	28
4.1.2.76 operator+() [5/7]	28
4.1.2.77 operator+() [6/7]	28
4.1.2.78 operator+() [7/7]	28
4.1.2.79 operator-() [1/14]	28
4.1.2.80 operator-() [2/14]	29
4.1.2.81 operator-() [3/14]	29
4.1.2.82 operator-() [4/14]	29
4.1.2.83 operator-() [5/14]	29
4.1.2.84 operator-() [6/14]	29
4.1.2.85 operator-() [7/14]	30
4.1.2.86 operator-() [8/14]	30
4.1.2.87 operator-() [9/14]	30
4.1.2.88 operator-() [10/14]	30
4.1.2.89 operator-() [11/14]	30
4.1.2.90 operator-() [12/14]	31
4.1.2.91 operator-() [13/14]	31
4.1.2.92 operator-() [14/14]	31
4.1.2.93 operator/() [1/3]	31
4.1.2.94 operator/() [2/3]	31
4.1.2.95 operator/() [3/3]	32
4.1.2.96 operator==() [1/4]	32
4.1.2.97 operator==() [2/4]	32
4.1.2.98 operator==() [3/4]	32
4.1.2.99 operator==() [4/4]	32
4.1.2.100 Orthonormalize() [1/2]	33
4.1.2.101 Orthonormalize() [2/2]	33
4.1.2.102 PolarToCartesian()	33
4.1.2.103 Projection() [1/3]	33
4.1.2.104 Projection() [2/3]	33
4.1.2.105 Projection() [3/3]	34
4.1.2.106 QuaternionToRotationMatrixCol()	34
4.1.2.107 QuaternionToRotationMatrixRow()	34
4.1.2.108 Rotate() [1/7]	34
4.1.2.109 Rotate() [2/7]	34
4.1.2.110 Rotate() [3/7]	35
4.1.2.111 Rotate() [4/7]	35
4.1.2.112 Rotate() [5/7]	35
4.1.2.113 Rotate() [6/7]	35
4.1.2.114 Rotate() [7/7]	36

4.1.2.115 RotationQuaternion() [1/3]	36
4.1.2.116 RotationQuaternion() [2/3]	36
4.1.2.117 RotationQuaternion() [3/3]	36
4.1.2.118 Scale() [1/6]	36
4.1.2.119 Scale() [2/6]	37
4.1.2.120 Scale() [3/6]	37
4.1.2.121 Scale() [4/6]	37
4.1.2.122 Scale() [5/6]	37
4.1.2.123 Scale() [6/6]	37
4.1.2.124 SetToIdentity() [1/3]	38
4.1.2.125 SetToIdentity() [2/3]	38
4.1.2.126 SetToIdentity() [3/3]	38
4.1.2.127 Slerp()	38
4.1.2.128 SphericalToCartesian()	38
4.1.2.129 Translate() [1/2]	39
4.1.2.130 Translate() [2/2]	39
4.1.2.131 Transpose() [1/3]	39
4.1.2.132 Transpose() [2/3]	39
4.1.2.133 Transpose() [3/3]	39
4.1.2.134 ZeroVector() [1/3]	40
4.1.2.135 ZeroVector() [2/3]	40
4.1.2.136 ZeroVector() [3/3]	40
5 Class Documentation	41
5.1 FAMath::Matrix2x2 Class Reference	41
5.1.1 Detailed Description	42
5.1.2 Constructor & Destructor Documentation	42
5.1.2.1 Matrix2x2() [1/5]	42
5.1.2.2 Matrix2x2() [2/5]	42
5.1.2.3 Matrix2x2() [3/5]	42
5.1.2.4 Matrix2x2() [4/5]	43
5.1.2.5 Matrix2x2() [5/5]	43
5.1.3 Member Function Documentation	43
5.1.3.1 Data() [1/2]	43
5.1.3.2 Data() [2/2]	43
5.1.3.3 GetCol()	43
5.1.3.4 GetRow()	44
5.1.3.5 operator()() [1/2]	44
5.1.3.6 operator()() [2/2]	44
5.1.3.7 operator*=() [1/2]	44
5.1.3.8 operator*=() [2/2]	44
5.1.3.9 operator+=()	45

5.1.3.10 operator-=()	45
5.1.3.11 operator=() [1/2]	45
5.1.3.12 operator=() [2/2]	45
5.1.3.13 SetCol()	45
5.1.3.14 SetRow()	46
5.2 FAMath::Matrix3x3 Class Reference	46
5.2.1 Detailed Description	47
5.2.2 Constructor & Destructor Documentation	47
5.2.2.1 Matrix3x3() [1/5]	47
5.2.2.2 Matrix3x3() [2/5]	47
5.2.2.3 Matrix3x3() [3/5]	47
5.2.2.4 Matrix3x3() [4/5]	48
5.2.2.5 Matrix3x3() [5/5]	48
5.2.3 Member Function Documentation	48
5.2.3.1 Data() [1/2]	48
5.2.3.2 Data() [2/2]	48
5.2.3.3 GetCol()	48
5.2.3.4 GetRow()	49
5.2.3.5 operator()() [1/2]	49
5.2.3.6 operator()() [2/2]	49
5.2.3.7 operator*=() [1/2]	49
5.2.3.8 operator*=() [2/2]	49
5.2.3.9 operator+=()	50
5.2.3.10 operator-=()	50
5.2.3.11 operator=() [1/2]	50
5.2.3.12 operator=() [2/2]	50
5.2.3.13 SetCol()	50
5.2.3.14 SetRow()	51
5.3 FAMath::Matrix4x4 Class Reference	51
5.3.1 Detailed Description	52
5.3.2 Constructor & Destructor Documentation	52
5.3.2.1 Matrix4x4() [1/5]	52
5.3.2.2 Matrix4x4() [2/5]	52
5.3.2.3 Matrix4x4() [3/5]	52
5.3.2.4 Matrix4x4() [4/5]	53
5.3.2.5 Matrix4x4() [5/5]	53
5.3.3 Member Function Documentation	53
5.3.3.1 Data() [1/2]	53
5.3.3.2 Data() [2/2]	53
5.3.3.3 GetCol()	53
5.3.3.4 GetRow()	54
5.3.3.5 operator()() [1/2]	54

5.3.3.6 operator>() [2/2]	54
5.3.3.7 operator*=() [1/2]	54
5.3.3.8 operator*=() [2/2]	54
5.3.3.9 operator+=()	55
5.3.3.10 operator-=()	55
5.3.3.11 operator=() [1/2]	55
5.3.3.12 operator=() [2/2]	55
5.3.3.13 SetCol()	55
5.3.3.14 SetRow()	56
5.4 FAMath::Quaternion Class Reference	56
5.4.1 Detailed Description	57
5.4.2 Constructor & Destructor Documentation	57
5.4.2.1 Quaternion() [1/3]	57
5.4.2.2 Quaternion() [2/3]	57
5.4.2.3 Quaternion() [3/3]	57
5.4.3 Member Function Documentation	57
5.4.3.1 GetScalar()	58
5.4.3.2 GetVector()	58
5.4.3.3 GetX()	58
5.4.3.4 GetY()	58
5.4.3.5 GetZ()	58
5.4.3.6 operator*=() [1/2]	58
5.4.3.7 operator*=() [2/2]	59
5.4.3.8 operator+=()	59
5.4.3.9 operator-=()	59
5.4.3.10 SetScalar()	59
5.4.3.11 SetVector()	59
5.4.3.12 SetX()	59
5.4.3.13 SetY()	60
5.4.3.14 SetZ()	60
5.5 FAMath::Vector2D Class Reference	60
5.5.1 Detailed Description	61
5.5.2 Constructor & Destructor Documentation	61
5.5.2.1 Vector2D() [1/3]	61
5.5.2.2 Vector2D() [2/3]	61
5.5.2.3 Vector2D() [3/3]	61
5.5.3 Member Function Documentation	61
5.5.3.1 GetX()	61
5.5.3.2 GetY()	62
5.5.3.3 operator*=()	62
5.5.3.4 operator+=()	62
5.5.3.5 operator-=()	62

5.5.3.6 operator/=()	62
5.5.3.7 operator=() [1/2]	62
5.5.3.8 operator=() [2/2]	63
5.5.3.9 SetX()	63
5.5.3.10 SetY()	63
5.6 FAMath::Vector3D Class Reference	63
5.6.1 Detailed Description	64
5.6.2 Constructor & Destructor Documentation	64
5.6.2.1 Vector3D() [1/3]	65
5.6.2.2 Vector3D() [2/3]	65
5.6.2.3 Vector3D() [3/3]	65
5.6.3 Member Function Documentation	65
5.6.3.1 GetX()	65
5.6.3.2 GetY()	65
5.6.3.3 GetZ()	66
5.6.3.4 operator*=()	66
5.6.3.5 operator+=()	66
5.6.3.6 operator-=()	66
5.6.3.7 operator/=()	66
5.6.3.8 operator=() [1/2]	66
5.6.3.9 operator=() [2/2]	67
5.6.3.10 SetX()	67
5.6.3.11 SetY()	67
5.6.3.12 SetZ()	67
5.7 FAMath::Vector4D Class Reference	67
5.7.1 Detailed Description	68
5.7.2 Constructor & Destructor Documentation	68
5.7.2.1 Vector4D() [1/3]	69
5.7.2.2 Vector4D() [2/3]	69
5.7.2.3 Vector4D() [3/3]	69
5.7.3 Member Function Documentation	69
5.7.3.1 GetW()	69
5.7.3.2 GetX()	69
5.7.3.3 GetY()	70
5.7.3.4 GetZ()	70
5.7.3.5 operator*=()	70
5.7.3.6 operator+=()	70
5.7.3.7 operator-=()	70
5.7.3.8 operator/=()	70
5.7.3.9 operator=() [1/2]	71
5.7.3.10 operator=() [2/2]	71
5.7.3.11 SetW()	71

5.7.3.12 SetX()	71
5.7.3.13 SetY()	71
5.7.3.14 SetZ()	71
6 File Documentation	73
6.1 FAMathEngine.h	73
Index	117

Chapter 1

Namespace Index

1.1 Namespace List

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

FAMath	Has the classes Vector2D , Vector3D , Vector4D , Matrix2x2 , Matrix3x3 , Matrix4x4 , Quaternion , and utility functions	7
------------------------	---	-------------------

Chapter 2

Class Index

2.1 Class List

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

FAMath::Matrix2x2	
A matrix class used for 2x2 matrices and their manipulations	41
FAMath::Matrix3x3	
A matrix class used for 3x3 matrices and their manipulations	46
FAMath::Matrix4x4	
A matrix class used for 4x4 matrices and their manipulations	51
FAMath::Quaternion	
A quaternion class used for quaternions and their manipulations	56
FAMath::Vector2D	
A vector class used for 2D vectors/points and their manipulations	60
FAMath::Vector3D	
A vector class used for 3D vectors/points and their manipulations	63
FAMath::Vector4D	
A vector class used for 4D vectors/points and their manipulations	67

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](#)
73

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.
- **bool operator==** (const **Vector2D** &a, const **Vector2D** &b)
Returns true if a equals to b, false otherwise.
- **bool operator!=** (const **Vector2D** &a, const **Vector2D** &b)
Returns true if a does not equal to b, false otherwise.
- **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.
- **Vector2D Lerp** (const **Vector2D** &start, const **Vector2D** &end, float t)
Linear interpolate between the two vectors start and end.
- **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 .
- **bool operator==** (const **Vector3D** &a, const **Vector3D** &b)
Returns true if a equals to b, false otherwise.
- **bool operator!=** (const **Vector3D** &a, const **Vector3D** &b)
Returns true if a does not equal to b, false otherwise.
- **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.
- [Vector3D Lerp](#) (const [Vector3D](#) &start, const [Vector3D](#) &end, float t)
Linear interpolate between the two vectors start and end.
- bool [ZeroVector](#) (const [Vector4D](#) &a)
Returns true if a is the zero vector.
- [Vector4D operator+](#) (const [Vector4D](#) &a, const [Vector4D](#) &b)
Adds a with b and returns the result.
- [Vector4D operator-](#) (const [Vector4D](#) &v)
Negatives v and returns the result.
- [Vector4D operator-](#) (const [Vector4D](#) &a, const [Vector4D](#) &b)
Subtracts b from a and returns the result.
- [Vector4D operator*](#) (const [Vector4D](#) &a, float k)
*Returns $a * k$.*
- [Vector4D operator*](#) (float k, const [Vector4D](#) &a)
*Returns $k * a$.*
- [Vector4D operator/](#) (const [Vector4D](#) &a, float k)
Returns a / k .
- bool [operator==](#) (const [Vector4D](#) &a, const [Vector4D](#) &b)
Returns true if a equals to b, false otherwise.
- bool [operator!=](#) (const [Vector4D](#) &a, const [Vector4D](#) &b)
Returns true if a does not equal to b, false otherwise.
- 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.
- [Vector4D Lerp](#) (const [Vector4D](#) &start, const [Vector4D](#) &end, float t)
Linear interpolate between the two vectors start and end.
- [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 [SetToIdentity](#) ([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 [operator==](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
- Returns true if q1 equals to q2, false otherwise.*

 - bool [operator!=](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
- Returns true if q1 does not equal to q2, false otherwise.*

 - 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.
- float [DotProduct](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2)
Returns the dot product of the quaternions q1 and q2.
- [Quaternion Lerp](#) (const [Quaternion](#) &q0, const [Quaternion](#) &q1, float t)
Linear Interpolate between quaternions q0 and q1.
- [Quaternion NLERp](#) (const [Quaternion](#) &q0, const [Quaternion](#) &q1, float t)
Normalize Linear Interpolate between quaternions q0 and q1.
- [Quaternion Slerp](#) (const [Quaternion](#) &q0, const [Quaternion](#) &q1, float t)
Spherical Linear Interpolate between quaternions q0 and q1.

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 vx 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 vx 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 vx 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() [1/4]

```
float FAMath::DotProduct (
    const Quaternion & q1,
    const Quaternion & q2 ) [inline]
```

Returns the dot product of the quaternions $q1$ and $q2$.

4.1.2.19 DotProduct() [2/4]

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

Returns the dot product between a and b .

4.1.2.20 DotProduct() [3/4]

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

Returns the dot product between a and b .

4.1.2.21 DotProduct() [4/4]

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

Returns the dot product between a and b .

4.1.2.22 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.

4.1.2.23 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.

4.1.2.24 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.

4.1.2.25 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.

4.1.2.26 IsIdentity() [1/4]

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

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

4.1.2.27 IsIdentity() [2/4]

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

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

4.1.2.28 IsIdentity() [3/4]

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

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

4.1.2.29 IsIdentity() [4/4]

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

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

4.1.2.30 IsZeroQuaternion()

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

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

4.1.2.31 Length() [1/4]

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

Returns the length of quaternion q .

4.1.2.32 Length() [2/4]

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

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

4.1.2.33 Length() [3/4]

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

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

4.1.2.34 Length() [4/4]

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

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

4.1.2.35 Lerp() [1/4]

```
Quaternion FAMath::Lerp (
    const Quaternion & q0,
    const Quaternion & q1,
    float t ) [inline]
```

Linear Interpolate between quaternions *q0* and *q1*.

t should be between 0 and 1. If it is not it will get clamped.

4.1.2.36 Lerp() [2/4]

```
Vector2D FAMath::Lerp (
    const Vector2D & start,
    const Vector2D & end,
    float t ) [inline]
```

Linear interpolate between the two vectors *start* and *end*.

t must between 0 and 1, if it is not it will get clamped.

4.1.2.37 Lerp() [3/4]

```
Vector3D FAMath::Lerp (
    const Vector3D & start,
    const Vector3D & end,
    float t ) [inline]
```

Linear interpolate between the two vectors *start* and *end*.

t must between 0 and 1, if it is not it will get clamped.

4.1.2.38 Lerp() [4/4]

```
Vector4D FAMath::Lerp (
    const Vector4D & start,
    const Vector4D & end,
    float t ) [inline]
```

Linear interpolate between the two vectors *start* and *end*.

t must between 0 and 1, if it is not it will get clamped.

4.1.2.39 NLerp()

```
Quaternion FAMath::NLerp (
    const Quaternion & q0,
    const Quaternion & q1,
    float t ) [inline]
```

Normalize Linear Interpolate between quaternions *q0* and *q1*.

t should be between 0 and 1. If it is not it will get clamped.

4.1.2.40 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.

4.1.2.41 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.

4.1.2.42 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.

4.1.2.43 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.

4.1.2.44 operator!=() [1/4]

```
bool FAMath::operator!= (
    const Quaternion & q1,
    const Quaternion & q2 ) [inline]
```

Returns true if $q1$ does not equal to $q2$, false otherwise.

4.1.2.45 operator!=() [2/4]

```
bool FAMath::operator!= (
    const Vector2D & a,
    const Vector2D & b ) [inline]
```

Returns true if *a* does not equal to *b*, false otherwise.

4.1.2.46 operator!=() [3/4]

```
bool FAMath::operator!= (
    const Vector3D & a,
    const Vector3D & b ) [inline]
```

Returns true if *a* does not equal to *b*, false otherwise.

4.1.2.47 operator!=() [4/4]

```
bool FAMath::operator!= (
    const Vector4D & a,
    const Vector4D & b ) [inline]
```

Returns true if *a* does not equal to *b*, false otherwise.

4.1.2.48 operator*() [1/24]

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

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

4.1.2.49 operator*() [2/24]

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

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

4.1.2.50 operator*() [3/24]

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

Multiplies k with m and returns the result.

4.1.2.51 operator*() [4/24]

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

Multiplies m with k and returns the result.

4.1.2.52 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.

4.1.2.53 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.

4.1.2.54 operator*() [7/24]

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

Multiplies m with k and returns the result.

4.1.2.55 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.

4.1.2.56 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.

4.1.2.57 operator*() [10/24]

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

Multiplies m with k and returns the result.

4.1.2.58 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.

4.1.2.59 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.

4.1.2.60 operator*() [13/24]

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

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

4.1.2.61 operator*() [14/24]

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

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

4.1.2.62 operator*() [15/24]

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

Returns $a * k$.

4.1.2.63 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.

4.1.2.64 operator*() [17/24]

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

Returns $a * k$.

4.1.2.65 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.

4.1.2.66 operator*() [19/24]

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

Returns $a * k$.

4.1.2.67 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.

4.1.2.68 operator*() [21/24]

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

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

4.1.2.69 operator*() [22/24]

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

Returns $k * a$.

4.1.2.70 operator*() [23/24]

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

Returns $k * a$.

4.1.2.71 operator*() [24/24]

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

Returns $k * a$.

4.1.2.72 operator+() [1/7]

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

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

4.1.2.73 operator+() [2/7]

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

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

4.1.2.74 operator+() [3/7]

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

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

4.1.2.75 operator+() [4/7]

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

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

4.1.2.76 operator+() [5/7]

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

Adds a with b and returns the result.

4.1.2.77 operator+() [6/7]

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

Adds a and b and returns the result.

4.1.2.78 operator+() [7/7]

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

Adds a with b and returns the result.

4.1.2.79 operator-() [1/14]

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

Negates the 2x2 matrix m .

4.1.2.80 operator-() [2/14]

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

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

4.1.2.81 operator-() [3/14]

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

Negates the 3x3 matrix m .

4.1.2.82 operator-() [4/14]

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

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

4.1.2.83 operator-() [5/14]

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

Negates the 4x4 matrix m .

4.1.2.84 operator-() [6/14]

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

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

4.1.2.85 operator-() [7/14]

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

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

4.1.2.86 operator-() [8/14]

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

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

4.1.2.87 operator-() [9/14]

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

Subtracts b from a and returns the result.

4.1.2.88 operator-() [10/14]

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

Negates the vector v and returns the result.

4.1.2.89 operator-() [11/14]

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

Subtracts b from a and returns the result.

4.1.2.90 operator-() [12/14]

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

Negates the vector v and returns the result.

4.1.2.91 operator-() [13/14]

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

Subtracts b from a and returns the result.

4.1.2.92 operator-() [14/14]

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

Negates v and returns the result.

4.1.2.93 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.

4.1.2.94 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.

4.1.2.95 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.

4.1.2.96 operator==() [1/4]

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

Returns true if $q1$ equals to $q2$, false otherwise.

4.1.2.97 operator==() [2/4]

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

Returns true if a equals to b , false otherwise.

4.1.2.98 operator==() [3/4]

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

Returns true if a equals to b , false otherwise.

4.1.2.99 operator==() [4/4]

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

Returns true if a equals to b , false otherwise.

4.1.2.100 Orthonormalize() [1/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

4.1.2.101 Orthonormalize() [2/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

4.1.2.102 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)

4.1.2.103 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.

4.1.2.104 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.

4.1.2.105 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.

4.1.2.106 QuaternionToRotationMatrixCol()

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

Transforms *q* into a column-major matrix.

q should be a unit quaternion.

4.1.2.107 QuaternionToRotationMatrixRow()

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

Transforms *q* into a row-major matrix.

q should be a unit quaternion.

4.1.2.108 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.

4.1.2.109 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.

4.1.2.110 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.

4.1.2.111 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.

4.1.2.112 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.

4.1.2.113 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.

4.1.2.114 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.

4.1.2.115 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.

4.1.2.116 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.

4.1.2.117 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.

4.1.2.118 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*.

4.1.2.119 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.

4.1.2.120 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.

4.1.2.121 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.

4.1.2.122 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.

4.1.2.123 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.

4.1.2.124 SetTolIdentity() [1/3]

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

Sets m to the identity matrix.

4.1.2.125 SetTolIdentity() [2/3]

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

Sets m to the identity matrix.

4.1.2.126 SetTolIdentity() [3/3]

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

Sets m to the identity matrix.

4.1.2.127 Slerp()

```
Quaternion FAMath::Slerp (
    const Quaternion & q0,
    const Quaternion & q1,
    float t ) [inline]
```

Spherical Linear Interpolate between quaternions $q0$ and $q1$.

t should be between 0 and 1. If it is not it will get clamped.

4.1.2.128 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)

4.1.2.129 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*.

4.1.2.130 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*.

4.1.2.131 Transpose() [1/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.132 Transpose() [2/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.133 Transpose() [3/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.134 ZeroVector() [1/3]

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

Returns true if *a* is the zero vector.

4.1.2.135 ZeroVector() [2/3]

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

Returns true if *a* is the zero vector.

4.1.2.136 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 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     {
211         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON))
212         {
213             return true;
214         }
215
216         return false;
217     }
218
219     inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
220     {
221     {
222         return Vector2D(a.GetX() + b.GetX(), a.GetY() + b.GetY());
223     }
224
225     inline Vector2D operator-(const Vector2D& v)
226     {
227     {
228         return Vector2D(-v.GetX(), -v.GetY());
229     }
230
231     inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
232     {
233     {
234         return Vector2D(a.GetX() - b.GetX(), a.GetY() - b.GetY());
235     }
236
237     inline Vector2D operator*(const Vector2D& a, float k)
238     {
239     {
240         return Vector2D(a.GetX() * k, a.GetY() * k);
241     }
242
243     inline Vector2D operator*(float k, const Vector2D& a)
244     {
245     {
246         return Vector2D(k * a.GetX(), k * a.GetY());
247     }
248
249     inline Vector2D operator/(const Vector2D& a, const float& k)
250     {
251     {
252         if (CompareFloats(k, 0.0f, EPSILON))
253         {
254             return Vector2D();
255         }
256
257         return Vector2D(a.GetX() / k, a.GetY() / k);
258     }
259
260     inline bool operator==(const Vector2D& a, const Vector2D& b)
261     {
262     {
263         return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f);
264     }
265
266     inline bool operator!=(const Vector2D& a, const Vector2D& b)
267     {
268     {
269         return !operator==(a, b);
270     }
271
272     inline float DotProduct(const Vector2D& a, const Vector2D& b)
273     {
274     {
275         return a.GetX() * b.GetX() + a.GetY() * b.GetY();
276     }
277
278     inline float Length(const Vector2D& v)
279     {
280     {
281         return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY());
282     }
283
284     inline Vector2D Norm(const Vector2D& v)
285     {
286     {

```

```

302         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
303
304         //v is the zero vector
305         if (ZeroVector(v))
306         {
307             return v;
308         }
309
310         float mag{ Length(v) };
311
312         return Vector2D(v.GetX() / mag, v.GetY() / mag);
313     }
314
315     inline Vector2D PolarToCartesian(const Vector2D& v)
316     {
317         //v = (r, theta)
318         //x = rcos(theta)
319         //y = rsin(theta)
320         float angle{ v.GetY() * PI / 180.0f };
321
322         return Vector2D(v.GetX() * cos(angle), v.GetX() * sin(angle));
323     }
324
325     inline Vector2D CartesianToPolar(const Vector2D& v)
326     {
327         //v = (x, y)
328         //r = sqrt(vx^2 + vy^2)
329         //theta = arctan(y / x)
330
331         if (CompareFloats(v.GetX(), 0.0f, EPSILON))
332         {
333             return v;
334         }
335
336         float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
337         return Vector2D(Length(v), theta);
338     }
339
340     inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
341     {
342         //Projb(a) = (a dot b)b
343         //normalize b before projecting
344
345         Vector2D normB(Norm(b));
346         return Vector2D(DotProduct(a, normB) * normB);
347     }
348
349     inline Vector2D Lerp(const Vector2D& start, const Vector2D& end, float t)
350     {
351         if (t < 0.0f)
352             t = 0.0f;
353         else if (t > 1.0f)
354             t = 1.0f;
355
356         return (1.0f - t) * start + t * end;
357     }
358
359     #if defined(_DEBUG)
360     inline void print(const Vector2D& v)
361     {
362         std::cout << "(" << v.GetX() << ", " << v.GetY() << ")";
363     }
364     #endif
365
366     //-----
367
368     //-----
369
370     //-----
371
372     class Vector3D
373     {
374     public:
375
376         Vector3D(float x = 0.0f, float y = 0.0f, float z = 0.0f);
377
378         Vector3D(const Vector2D& v, float z = 0.0f);
379
380         Vector3D(const Vector4D& v);
381
382         float GetX() const;
383
384         float GetY() const;
385
386         float GetZ() const;

```

```

420
423     void SetX(float x);
424
427     void SetY(float y);
428
431     void SetZ(float z);
432
435     Vector3D& operator=(const Vector2D& v);
436
439     Vector3D& operator=(const Vector4D& v);
440
443     Vector3D& operator+=(const Vector3D& b);
444
447     Vector3D& operator-=(const Vector3D& b);
448
451     Vector3D& operator*=(float k);
452
457     Vector3D& operator/=(float k);
458
459 private:
460     float mX;
461     float mY;
462     float mZ;
463 };
464
465 //-----
466 //Vector3D Constructors
467
468 inline Vector3D::Vector3D(float x, float y, float z) : mX{ x }, mY{ y }, mZ{ z }
469 {}
470
471 //-----
472
473 //-----
474 //Vector3D Getters and Setters
475
476 inline float Vector3D::GetX() const
477 {
478     return mX;
479 }
480
481 inline float Vector3D::GetY() const
482 {
483     return mY;
484 }
485
486 inline float Vector3D::GetZ() const
487 {
488     return mZ;
489 }
490
491 inline void Vector3D::SetX(float x)
492 {
493     mX = x;
494 }
495
496 inline void Vector3D::SetY(float y)
497 {
498     mY = y;
499 }
500
501 inline void Vector3D::SetZ(float z)
502 {
503     mZ = z;
504 }
505 //-----
506
507
508 //-----
509 //Vector3D Member functions
510
511 inline Vector3D& Vector3D::operator+=(const Vector3D& b)
512 {
513     this->mX += b.mX;
514     this->mY += b.mY;
515     this->mZ += b.mZ;
516
517     return *this;
518 }
519
520 inline Vector3D& Vector3D::operator-=(const Vector3D& b)
521 {
522     this->mX -= b.mX;
523     this->mY -= b.mY;
524     this->mZ -= b.mZ;
525
526     return *this;

```

```

527     }
528
529     inline Vector3D& Vector3D::operator*=(float k)
530     {
531         this->mX *= k;
532         this->mY *= k;
533         this->mZ *= k;
534
535         return *this;
536     }
537
538     inline Vector3D& Vector3D::operator/=(float k)
539     {
540         if (CompareFloats(k, 0.0f, EPSILON))
541         {
542             return *this;
543         }
544
545         this->mX /= k;
546         this->mY /= k;
547         this->mZ /= k;
548
549         return *this;
550     }
551
552     //-----
553
554     //-----
555     //Vector3D Non-member functions
556
557     inline bool ZeroVector(const Vector3D& a)
558     {
559         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
560             CompareFloats(a.GetZ(), 0.0f, EPSILON))
561         {
562             return true;
563         }
564
565         return false;
566     }
567
568     inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
569     {
570         return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
571     }
572
573     inline Vector3D operator-(const Vector3D& v)
574     {
575         return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
576     }
577
578     inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
579     {
580         return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
581     }
582
583     inline Vector3D operator*(const Vector3D& a, float k)
584     {
585         return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
586     }
587
588     inline Vector3D operator*(float k, const Vector3D& a)
589     {
590         return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
591     }
592
593     inline Vector3D operator/(const Vector3D& a, float k)
594     {
595         if (CompareFloats(k, 0.0f, EPSILON))
596         {
597             return Vector3D();
598         }
599
600         return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
601     }
602
603     inline bool operator==(const Vector3D& a, const Vector3D& b)
604     {
605         return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f) &&
606             CompareFloats(a.GetZ(), b.GetZ(), 1e-6f);
607     }
608
609     inline bool operator!=(const Vector3D& a, const Vector3D& b)
610     {
611         return !operator==(a, b);
612     }
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633

```



```

636 inline float DotProduct(const Vector3D& a, const Vector3D& b)
637 {
638     //a dot b = axbx + ayby + azbz
639     return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
640 }
641
642 inline Vector3D CrossProduct(const Vector3D& a, const Vector3D& b)
643 {
644     //a x b = (aybz - azby, azbx - axbz, axby - aybx)
645
646     return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
647         a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
648         a.GetX() * b.GetY() - a.GetY() * b.GetX());
649 }
650
651 inline float Length(const Vector3D& v)
652 {
653     //length(v) = sqrt(vx^2 + vy^2 + vz^2)
654
655     return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
656 }
657
658 inline Vector3D Norm(const Vector3D& v)
659 {
660     //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
661     //v is the zero vector
662     if (ZeroVector(v))
663     {
664         return v;
665     }
666
667     float mag{ Length(v) };
668
669     return Vector3D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag);
670 }
671
672 inline Vector3D CylindricalToCartesian(const Vector3D& v)
673 {
674     //v = (r, theta, z)
675     //x = rcos(theta)
676     //y = rsin(theta)
677     //z = z
678     float angle{ v.GetY() * PI / 180.0f };
679
680     return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
681 }
682
683 inline Vector3D CartesianToCylindrical(const Vector3D& v)
684 {
685     //v = (x, y, z)
686     //r = sqrt(vx^2 + vy^2 + vz^2)
687     //theta = arctan(y / x)
688     //z = z
689     if (CompareFloats(v.GetX(), 0.0f, EPSILON))
690     {
691         return v;
692     }
693
694     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
695     return Vector3D(Length(v), theta, v.GetZ());
696 }
697
698 inline Vector3D SphericalToCartesian(const Vector3D& v)
699 {
700     // v = (pho, phi, theta)
701     //x = pho * sin(phi) * cos(theta)
702     //y = pho * sin(phi) * sin(theta)
703     //z = pho * cos(theta);
704
705     float phi{ v.GetY() * PI / 180.0f };
706     float theta{ v.GetZ() * PI / 180.0f };
707
708     return Vector3D(v.GetX() * sin(phi) * cos(theta), v.GetX() * sin(phi) * sin(theta), v.GetX() *
709         cos(theta));
710 }
711
712 inline Vector3D CartesianToSpherical(const Vector3D& v)
713 {
714     //v = (x, y, z)
715     //pho = sqrt(vx^2 + vy^2 + vz^2)
716     //phi = arcos(z / pho)
717     //theta = arctan(y / x)
718
719     if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
720     {
721         return v;
722     }
723 }

```

```

751     float pho{ Length(v) };
752     float phi{ acos(v.GetZ() / pho) * 180.0f / PI };
753     float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
754     return Vector3D(pho, phi, theta);
755 }
756
757 inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
758 {
759     //Projb(a) = (a dot b)b
760     //normalize b before projecting
761     Vector3D normB(Norm(b));
762     return Vector3D(DotProduct(a, normB) * normB);
763 }
764
765 inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
766 {
767     x = Norm(x);
768     y = Norm(CrossProduct(z, x));
769     z = Norm(CrossProduct(x, y));
770 }
771
772 inline Vector3D Lerp(const Vector3D& start, const Vector3D& end, float t)
773 {
774     if (t < 0.0f)
775         t = 0.0f;
776     else if (t > 1.0f)
777         t = 1.0f;
778     return (1.0f - t) * start + t * end;
779 }
780
781 #if defined(_DEBUG)
782 inline void print(const Vector3D& v)
783 {
784     std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ")";
785 }
786 #endif
787 //-----
788
789 //-----
790
791 //-----
792
793 class Vector4D
794 {
795 public:
796     Vector4D(float x = 0.0f, float y = 0.0f, float z = 0.0f, float w = 0.0f);
797     Vector4D(const Vector2D& v, float z = 0.0f, float w = 0.0f);
798     Vector4D(const Vector3D& v, float w = 0.0f);
799
800     float GetX() const;
801     float GetY() const;
802     float GetZ() const;
803     float GetW() const;
804
805     void SetX(float x);
806     void SetY(float y);
807     void SetZ(float z);
808     void SetW(float w);
809
810     Vector4D& operator=(const Vector2D& v);
811     Vector4D& operator=(const Vector3D& v);
812     Vector4D& operator+=(const Vector4D& b);
813     Vector4D& operator-=(const Vector4D& b);
814     Vector4D& operator*=(float k);
815     Vector4D& operator/=(float k);
816
817
818

```

```

889     private:
890         float mX;
891         float mY;
892         float mZ;
893         float mW;
894     };
895
896     //-----
897     //Vector4D Constructors
898
899     inline Vector4D::Vector4D(float x, float y, float z, float w) : mX{ x }, mY{ y }, mZ{ z }, mW{ w }
900     {}
901
902     //-----
903
904     //-----
905     //Vector4D Getters and Setters
906
907     inline float Vector4D::GetX() const
908     {
909         return mX;
910     }
911
912     inline float Vector4D::GetY() const
913     {
914         return mY;
915     }
916
917     inline float Vector4D::GetZ() const
918     {
919         return mZ;
920     }
921
922     inline float Vector4D::GetW() const
923     {
924         return mW;
925     }
926
927     inline void Vector4D::SetX(float x)
928     {
929         mX = x;
930     }
931
932     inline void Vector4D::SetY(float y)
933     {
934         mY = y;
935     }
936
937     inline void Vector4D::SetZ(float z)
938     {
939         mZ = z;
940     }
941
942     inline void Vector4D::SetW(float w)
943     {
944         mW = w;
945     }
946     //-----
947
948     //-----
949     //Vector4D Member functions
950
951     inline Vector4D& Vector4D::operator+=(const Vector4D& b)
952     {
953         {
954             this->mX += b.mX;
955             this->mY += b.mY;
956             this->mZ += b.mZ;
957             this->mW += b.mW;
958
959             return *this;
960         }
961
962     inline Vector4D& Vector4D::operator-=(const Vector4D& b)
963     {
964         {
965             this->mX -= b.mX;
966             this->mY -= b.mY;
967             this->mZ -= b.mZ;
968             this->mW -= b.mW;
969
970             return *this;
971         }
972
973     inline Vector4D& Vector4D::operator*=(float k)
974     {
975         {
976             this->mX *= k;
977             this->mY *= k;

```

```

976         this->mZ *= k;
977         this->mW *= k;
978
979         return *this;
980     }
981
982     inline Vector4D& Vector4D::operator/=(float k)
983     {
984         if (CompareFloats(k, 0.0f, EPSILON))
985         {
986             return *this;
987         }
988
989         this->mX /= k;
990         this->mY /= k;
991         this->mZ /= k;
992         this->mW /= k;
993
994         return *this;
995     }
996
997     //-----
998     //-----
999     //Vector4D Non-member functions
1000
1001     inline bool ZeroVector(const Vector4D& a)
1002     {
1003         if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
1004             CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
1005         {
1006             return true;
1007         }
1008
1009         return false;
1010     }
1011
1012     inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
1013     {
1014         return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
1015             b.GetW());
1016     }
1017
1018     inline Vector4D operator-(const Vector4D& v)
1019     {
1020         return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
1021     }
1022
1023     inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
1024     {
1025         return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
1026             b.GetW());
1027     }
1028
1029     inline Vector4D operator*(const Vector4D& a, float k)
1030     {
1031         return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
1032     }
1033
1034     inline Vector4D operator*(float k, const Vector4D& a)
1035     {
1036         return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
1037     }
1038
1039     inline Vector4D operator/(const Vector4D& a, float k)
1040     {
1041         if (CompareFloats(k, 0.0f, EPSILON))
1042         {
1043             return Vector4D();
1044         }
1045
1046         return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
1047     }
1048
1049     inline bool operator==(const Vector4D& a, const Vector4D& b)
1050     {
1051         return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f) &&
1052             CompareFloats(a.GetZ(), b.GetZ(), 1e-6f) && CompareFloats(a.GetW(), b.GetW(), 1e-6f);
1053     }
1054
1055     inline bool operator!=(const Vector4D& a, const Vector4D& b)
1056     {
1057         return !operator==(a, b);
1058     }
1059
1060     inline float DotProduct(const Vector4D& a, const Vector4D& b)
1061     {

```

```

1083         //a dot b = axbx + ayby + azbz + awbw
1084         return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
1085     }
1086
1087     inline float Length(const Vector4D& v)
1088     {
1089         //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
1090         return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
1091 v.GetW());
1092     }
1093
1094     inline Vector4D Norm(const Vector4D& v)
1095     {
1096         //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
1097         //v is the zero vector
1098         if (ZeroVector(v))
1099         {
1100             return v;
1101         }
1102
1103         float mag{ Length(v) };
1104
1105         return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
1106     }
1107
1108     inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
1109     {
1110         //Projb(a) = (a dot b)b
1111         //normalize b before projecting
1112         Vector4D normB(Norm(b));
1113         return Vector4D(DotProduct(a, normB) * normB);
1114     }
1115
1116     inline void Orthonormalize(Vector4D& x, Vector4D& y, Vector4D& z)
1117     {
1118         FAMath::Vector3D tempX(x.GetX(), x.GetY(), x.GetZ());
1119         FAMath::Vector3D tempY(y.GetX(), y.GetY(), y.GetZ());
1120         FAMath::Vector3D tempZ(z.GetX(), z.GetY(), z.GetZ());
1121
1122         tempX = Norm(tempX);
1123         tempY = Norm(CrossProduct(tempZ, tempX));
1124         tempZ = Norm(CrossProduct(tempX, tempY));
1125
1126         x = FAMath::Vector4D(tempX.GetX(), tempX.GetY(), tempX.GetZ(), 0.0f);
1127         y = FAMath::Vector4D(tempY.GetX(), tempY.GetY(), tempY.GetZ(), 0.0f);
1128         z = FAMath::Vector4D(tempZ.GetX(), tempZ.GetY(), tempZ.GetZ(), 0.0f);
1129     }
1130
1131     inline Vector4D Lerp(const Vector4D& start, const Vector4D& end, float t)
1132     {
1133         if (t < 0.0f)
1134             t = 0.0f;
1135         else if (t > 1.0f)
1136             t = 1.0f;
1137
1138         return (1.0f - t) * start + t * end;
1139     }
1140
1141     #if defined(_DEBUG)
1142     inline void print(const Vector4D& v)
1143     {
1144         std::cout << "(" << v.GetX() << ", " << v.GetY() << ", " << v.GetZ() << ", " << v.GetW() << ")";
1145     }
1146     #endif
1147     //-----
1148
1149     //-----
1150
1151     class Matrix2x2
1152     {
1153     public:
1154         Matrix2x2();
1155
1156         Matrix2x2(float a[][2]);
1157
1158         Matrix2x2(const Vector2D& r1, const Vector2D& r2);
1159
1160         Matrix2x2(const Matrix3x3& m);
1161
1162         Matrix2x2(const Matrix4x4& m);
1163
1164         float* Data();

```

```

1207
1210     const float* Data() const;
1211
1216     const float& operator()(unsigned int row, unsigned int col) const;
1217
1222     float& operator()(unsigned int row, unsigned int col);
1223
1228     Vector2D GetRow(unsigned int row) const;
1229
1234     Vector2D GetCol(unsigned int col) const;
1235
1240     void SetRow(unsigned int row, Vector2D v);
1241
1246     void SetCol(unsigned int col, Vector2D v);
1247
1250     Matrix2x2& operator=(const Matrix3x3& m);
1251
1254     Matrix2x2& operator=(const Matrix4x4& m);
1255
1258     Matrix2x2& operator+=(const Matrix2x2& m);
1259
1262     Matrix2x2& operator-=(const Matrix2x2& m);
1263
1266     Matrix2x2& operator*=(float k);
1267
1270     Matrix2x2& operator*=(const Matrix2x2& m);
1271
1272 private:
1273
1274     float mMat[2][2];
1275 };
1276
1277 //-----
1278 inline Matrix2x2::Matrix2x2()
1279 {
1280     //1st row
1281     mMat[0][0] = 1.0f;
1282     mMat[0][1] = 0.0f;
1283
1284     //2nd
1285     mMat[1][0] = 0.0f;
1286     mMat[1][1] = 1.0f;
1287 }
1288
1289 inline Matrix2x2::Matrix2x2(float a[][2])
1290 {
1291     //1st row
1292     mMat[0][0] = a[0][0];
1293     mMat[0][1] = a[0][1];
1294
1295     //2nd row
1296     mMat[1][0] = a[1][0];
1297     mMat[1][1] = a[1][1];
1298 }
1299
1300 inline Matrix2x2::Matrix2x2(const Vector2D& r1, const Vector2D& r2)
1301 {
1302     SetRow(0, r1);
1303     SetRow(1, r2);
1304 }
1305
1306 inline float* Matrix2x2::Data()
1307 {
1308     return mMat[0];
1309 }
1310
1311 inline const float* Matrix2x2::Data()const
1312 {
1313     return mMat[0];
1314 }
1315
1316 inline const float& Matrix2x2::operator()(unsigned int row, unsigned int col)const
1317 {
1318     if (row > 1 || col > 1)
1319     {
1320         return mMat[0][0];
1321     }
1322     else
1323     {
1324         return mMat[row][col];
1325     }
1326 }
1327
1328 inline float& Matrix2x2::operator()(unsigned int row, unsigned int col)
1329 {
1330     if (row > 1 || col > 1)
1331     {

```

```

1332         return mMat[0][0];
1333     }
1334     else
1335     {
1336         return mMat[row][col];
1337     }
1338 }
1339
1340 inline Vector2D Matrix2x2::GetRow(unsigned int row) const
1341 {
1342     if (row < 0 || row > 1)
1343         return Vector2D(mMat[0][0], mMat[0][1]);
1344     else
1345         return Vector2D(mMat[row][0], mMat[row][1]);
1346 }
1347
1348 inline Vector2D Matrix2x2::GetCol(unsigned int col) const
1349 {
1350     if (col < 0 || col > 1)
1351         return Vector2D(mMat[0][0], mMat[1][0]);
1352     else
1353         return Vector2D(mMat[0][col], mMat[1][col]);
1354 }
1355
1356 inline void Matrix2x2::SetRow(unsigned int row, Vector2D v)
1357 {
1358     if (row > 1)
1359     {
1360         mMat[0][0] = v.GetX();
1361         mMat[0][1] = v.GetY();
1362     }
1363     else
1364     {
1365         mMat[row][0] = v.GetX();
1366         mMat[row][1] = v.GetY();
1367     }
1368 }
1369
1370 inline void Matrix2x2::SetCol(unsigned int col, Vector2D v)
1371 {
1372     if (col > 1)
1373     {
1374         mMat[0][0] = v.GetX();
1375         mMat[1][0] = v.GetY();
1376     }
1377     else
1378     {
1379         mMat[0][col] = v.GetX();
1380         mMat[1][col] = v.GetY();
1381     }
1382 }
1383
1384 inline Matrix2x2& Matrix2x2::operator+=(const Matrix2x2& m)
1385 {
1386     for (int i = 0; i < 2; ++i)
1387     {
1388         for (int j = 0; j < 2; ++j)
1389         {
1390             this->mMat[i][j] += m.mMat[i][j];
1391         }
1392     }
1393     return *this;
1394 }
1395
1396 inline Matrix2x2& Matrix2x2::operator-=(const Matrix2x2& m)
1397 {
1398     for (int i = 0; i < 2; ++i)
1399     {
1400         for (int j = 0; j < 2; ++j)
1401         {
1402             this->mMat[i][j] -= m.mMat[i][j];
1403         }
1404     }
1405     return *this;
1406 }
1407
1408 inline Matrix2x2& Matrix2x2::operator*=(float k)
1409 {
1410     for (int i = 0; i < 2; ++i)
1411     {
1412         for (int j = 0; j < 2; ++j)
1413         {
1414             this->mMat[i][j] *= k;
1415         }
1416     }
1417 }

```

```

1419     }
1420
1421     return *this;
1422 }
1423
1424 inline Matrix2x2& Matrix2x2::operator*=(const Matrix2x2& m)
1425 {
1426     Matrix2x2 res;
1427
1428     for (int i = 0; i < 2; ++i)
1429     {
1430         res.mMat[i][0] =
1431             (mMat[i][0] * m.mMat[0][0]) +
1432             (mMat[i][1] * m.mMat[1][0]);
1433
1434         res.mMat[i][1] =
1435             (mMat[i][0] * m.mMat[0][1]) +
1436             (mMat[i][1] * m.mMat[1][1]);
1437     }
1438
1439     for (int i = 0; i < 2; ++i)
1440     {
1441         for (int j = 0; j < 2; ++j)
1442         {
1443             mMat[i][j] = res.mMat[i][j];
1444         }
1445     }
1446
1447     return *this;
1448 }
1449
1450 inline Matrix2x2 operator+(const Matrix2x2& m1, const Matrix2x2& m2)
1451 {
1452     Matrix2x2 res;
1453     for (int i = 0; i < 2; ++i)
1454     {
1455         for (int j = 0; j < 2; ++j)
1456         {
1457             res(i, j) = m1(i, j) + m2(i, j);
1458         }
1459     }
1460
1461     return res;
1462 }
1463
1464 inline Matrix2x2 operator-(const Matrix2x2& m)
1465 {
1466     Matrix2x2 res;
1467     for (int i = 0; i < 2; ++i)
1468     {
1469         for (int j = 0; j < 2; ++j)
1470         {
1471             res(i, j) = -m(i, j);
1472         }
1473     }
1474
1475     return res;
1476 }
1477
1478 inline Matrix2x2 operator-(const Matrix2x2& m1, const Matrix2x2& m2)
1479 {
1480     Matrix2x2 res;
1481     for (int i = 0; i < 2; ++i)
1482     {
1483         for (int j = 0; j < 2; ++j)
1484         {
1485             res(i, j) = m1(i, j) - m2(i, j);
1486         }
1487     }
1488
1489     return res;
1490 }
1491
1492 inline Matrix2x2 operator*(const Matrix2x2& m, const float& k)
1493 {
1494     Matrix2x2 res;
1495     for (int i = 0; i < 2; ++i)
1496     {
1497         for (int j = 0; j < 2; ++j)
1498         {
1499             res(i, j) = m(i, j) * k;
1500         }
1501     }
1502
1503     return res;
1504 }
1505
1506 }
1507
1508 }
1509
1510 }
1511
1512 }
1513

```



```

1516     inline Matrix2x2 operator*(const float& k, const Matrix2x2& m)
1517     {
1518         Matrix2x2 res;
1519         for (int i = 0; i < 2; ++i)
1520         {
1521             for (int j = 0; j < 2; ++j)
1522             {
1523                 res(i, j) = k * m(i, j);
1524             }
1525         }
1526         return res;
1527     }
1528 }
1529
1530 inline Matrix2x2 operator*(const Matrix2x2& m1, const Matrix2x2& m2)
1531 {
1532     Matrix2x2 res;
1533     for (int i = 0; i < 4; ++i)
1534     {
1535         res(i, 0) =
1536             (m1(i, 0) * m2(0, 0)) +
1537             (m1(i, 1) * m2(1, 0));
1538
1539         res(i, 1) =
1540             (m1(i, 0) * m2(0, 1)) +
1541             (m1(i, 1) * m2(1, 1));
1542
1543         res(i, 2) =
1544             (m1(i, 0) * m2(0, 2)) +
1545             (m1(i, 1) * m2(1, 2));
1546
1547         res(i, 3) =
1548             (m1(i, 0) * m2(0, 3)) +
1549             (m1(i, 1) * m2(1, 3));
1550     }
1551     return res;
1552 }
1553
1554 inline Vector2D operator*(const Matrix2x2& m, const Vector2D& v)
1555 {
1556     Vector2D res;
1557     res.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY());
1558     res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY());
1559     return res;
1560 }
1561
1562 inline Vector2D operator*(const Vector2D& v, const Matrix2x2& m)
1563 {
1564     Vector2D res;
1565     res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0));
1566     res.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1));
1567     return res;
1568 }
1569
1570 inline void SetToIdentity(Matrix2x2& m)
1571 {
1572     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1573
1574     //1st row
1575     m(0, 0) = 1.0f;
1576     m(0, 1) = 0.0f;
1577
1578     //2nd row
1579     m(1, 0) = 0.0f;
1580     m(1, 1) = 1.0f;
1581 }
1582
1583 inline bool IsIdentity(const Matrix2x2& m)
1584 {
1585     //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
1586     0.0f
1587
1588     for (int i = 0; i < 2; ++i)
1589     {
1590         for (int j = 0; j < 2; ++j)
1591         {
1592             if (i == j)
1593             {
1594                 if (!CompareFloats(m(i, j), 1.0f, EPSILON))

```

```

1618             return false;
1619         }
1620     }
1621     else
1622     {
1623         if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1624             return false;
1625     }
1626 }
1627 }
1628 }
1629 }
1630
1631 inline Matrix2x2 Transpose(const Matrix2x2& m)
1632 {
1633     //make the rows into cols
1634
1635     Matrix2x2 res;
1636
1637     //1st col = 1st row
1638     res(0, 0) = m(0, 0);
1639     res(1, 0) = m(0, 1);
1640
1641     //2nd col = 2nd row
1642     res(0, 1) = m(1, 0);
1643     res(1, 1) = m(1, 1);
1644
1645     return res;
1646 }
1647
1648 inline Matrix2x2 Scale(const Matrix2x2& cm, float x, float y)
1649 {
1650     //x 0
1651     //0 y
1652
1653     Matrix2x2 scale;
1654     scale(0, 0) = x;
1655     scale(1, 1) = y;
1656
1657     return cm * scale;
1658 }
1659
1660 inline Matrix2x2 Scale(const Matrix2x2& cm, const Vector2D& scaleVector)
1661 {
1662     //x 0
1663     //0 y
1664
1665     Matrix2x2 scale;
1666     scale(0, 0) = scaleVector.GetX();
1667     scale(1, 1) = scaleVector.GetY();
1668
1669     return cm * scale;
1670 }
1671
1672 inline Matrix2x2 Rotate(const Matrix2x2& cm, float angle)
1673 {
1674     //c    s
1675     //-s    c
1676     //c = cos(angle)
1677     //s = sin(angle)
1678
1679     float c = cos(angle * PI / 180.0f);
1680     float s = sin(angle * PI / 180.0f);
1681
1682     Matrix2x2 result;
1683
1684     //1st row
1685     result(0, 0) = c;
1686     result(0, 1) = s;
1687
1688     //2nd row
1689     result(1, 0) = -s;
1690     result(1, 1) = c;
1691
1692     return cm * result;
1693 }
1694
1695 inline double Determinant(const Matrix2x2& m)
1696 {
1697     return (double)m(0, 0) * m(1, 1) - (double)m(0, 1) * m(1, 0);
1698 }
1699
1700 inline double Cofactor(const Matrix2x2& m, unsigned int row, unsigned int col)
1701 {
1702     //cij = ((-1)^(i + j)) * det of minor(i, j);
1703     double minor{ 0.0 };
1704 }

```

```

1723         if (row == 0 && col == 0)
1724             minor = m(1, 1);
1725         else if (row == 0 && col == 1)
1726             minor = m(1, 0);
1727         else if (row == 1 && col == 0)
1728             minor = m(0, 1);
1729         else if (row == 1 && col == 1)
1730             minor = m(0, 0);
1731
1732         return pow(-1, row + col) * minor;
1733     }
1734
1735     inline Matrix2x2 Adjoint(const Matrix2x2& m)
1736     {
1737         //Cofactor of each ijth position put into matrix cA.
1738         //Adjoint is the tranposed matrix of cA.
1739         Matrix2x2 cofactorMatrix;
1740         for (int i = 0; i < 2; ++i)
1741         {
1742             for (int j = 0; j < 2; ++j)
1743             {
1744                 cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
1745             }
1746         }
1747
1748         return Transpose(cofactorMatrix);
1749     }
1750
1751     inline Matrix2x2 Inverse(const Matrix2x2& m)
1752     {
1753         //Inverse of m = adjoint of m / det of m
1754         double det = Determinant(m);
1755         if (CompareDoubles(det, 0.0, EPSILON))
1756             return Matrix2x2();
1757
1758         return Adjoint(m) * (1.0f / static_cast<float>(det));
1759     }
1760
1761     #if defined(_DEBUG)
1762     inline void print(const Matrix2x2& m)
1763     {
1764         for (int i = 0; i < 2; ++i)
1765         {
1766             for (int j = 0; j < 2; ++j)
1767             {
1768                 std::cout << m(i, j) << " ";
1769             }
1770
1771             std::cout << std::endl;
1772         }
1773     }
1774     #endif
1775
1776     //-----
1777
1778     //-----
1779
1780     class Matrix3x3
1781     {
1782     public:
1783
1784         Matrix3x3();
1785
1786         Matrix3x3(float a[][3]);
1787
1788         Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3);
1789
1790         Matrix3x3(const Matrix2x2& m);
1791
1792         Matrix3x3(const Matrix4x4& m);
1793
1794         float* Data();
1795
1796         const float* Data() const;
1797
1798         const float& operator()(unsigned int row, unsigned int col) const;
1799
1800         float& operator()(unsigned int row, unsigned int col);
1801     };

```

```

1852     Vector3D GetRow(unsigned int row) const;
1853
1858     Vector3D GetCol(unsigned int col) const;
1859
1864     void SetRow(unsigned int row, Vector3D v);
1865
1870     void SetCol(unsigned int col, Vector3D v);
1871
1877     Matrix3x3& operator=(const Matrix2x2& m);
1878
1881     Matrix3x3& operator=(const Matrix4x4& m);
1882
1885     Matrix3x3& operator+=(const Matrix3x3& m);
1886
1889     Matrix3x3& operator-=(const Matrix3x3& m);
1890
1893     Matrix3x3& operator*=(float k);
1894
1897     Matrix3x3& operator*=(const Matrix3x3& m);
1898
1899 private:
1900
1901     float mMat[3][3];
1902 };
1903
1904 //-----
1905 inline Matrix3x3::Matrix3x3()
1906 {
1907     //1st row
1908     mMat[0][0] = 1.0f;
1909     mMat[0][1] = 0.0f;
1910     mMat[0][2] = 0.0f;
1911
1912     //2nd
1913     mMat[1][0] = 0.0f;
1914     mMat[1][1] = 1.0f;
1915     mMat[1][2] = 0.0f;
1916
1917     //3rd row
1918     mMat[2][0] = 0.0f;
1919     mMat[2][1] = 0.0f;
1920     mMat[2][2] = 1.0f;
1921 }
1922
1923 inline Matrix3x3::Matrix3x3(float a[][3])
1924 {
1925     //1st row
1926     mMat[0][0] = a[0][0];
1927     mMat[0][1] = a[0][1];
1928     mMat[0][2] = a[0][2];
1929
1930     //2nd
1931     mMat[1][0] = a[1][0];
1932     mMat[1][1] = a[1][1];
1933     mMat[1][2] = a[1][2];
1934
1935     //3rd row
1936     mMat[2][0] = a[2][0];
1937     mMat[2][1] = a[2][1];
1938     mMat[2][2] = a[2][2];
1939 }
1940
1941 inline Matrix3x3::Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3)
1942 {
1943     SetRow(0, r1);
1944     SetRow(1, r2);
1945     SetRow(2, r3);
1946 }
1947
1948 inline float* Matrix3x3::Data()
1949 {
1950     return mMat[0];
1951 }
1952
1953 inline const float* Matrix3x3::Data()const
1954 {
1955     return mMat[0];
1956 }
1957
1958 inline const float& Matrix3x3::operator()(unsigned int row, unsigned int col)const
1959 {
1960     if (row > 2 || col > 2)
1961     {
1962         return mMat[0][0];
1963     }
1964     else
1965     {

```

```

1966         return mMat[row][col];
1967     }
1968 }
1969
1970 inline float& Matrix3x3::operator()(unsigned int row, unsigned int col)
1971 {
1972     if (row > 2 || col > 2)
1973     {
1974         return mMat[0][0];
1975     }
1976     else
1977     {
1978         return mMat[row][col];
1979     }
1980 }
1981
1982 inline Vector3D Matrix3x3::GetRow(unsigned int row) const
1983 {
1984     if (row < 0 || row > 2)
1985         return Vector3D(mMat[0][0], mMat[0][1], mMat[0][2]);
1986     else
1987         return Vector3D(mMat[row][0], mMat[row][1], mMat[row][2]);
1988 }
1989
1990 inline Vector3D Matrix3x3::GetCol(unsigned int col) const
1991 {
1992     if (col < 0 || col > 2)
1993         return Vector3D(mMat[0][0], mMat[1][0], mMat[2][0]);
1994     else
1995         return Vector3D(mMat[0][col], mMat[1][col], mMat[2][col]);
1996 }
1997
1998 inline void Matrix3x3::SetRow(unsigned int row, Vector3D v)
1999 {
2000     if (row > 2)
2001     {
2002         mMat[0][0] = v.GetX();
2003         mMat[0][1] = v.GetY();
2004         mMat[0][2] = v.GetZ();
2005     }
2006     else
2007     {
2008         mMat[row][0] = v.GetX();
2009         mMat[row][1] = v.GetY();
2010         mMat[row][2] = v.GetZ();
2011     }
2012 }
2013
2014 inline void Matrix3x3::SetCol(unsigned int col, Vector3D v)
2015 {
2016     if (col > 2)
2017     {
2018         mMat[0][0] = v.GetX();
2019         mMat[1][0] = v.GetY();
2020         mMat[2][0] = v.GetZ();
2021     }
2022     else
2023     {
2024         mMat[0][col] = v.GetX();
2025         mMat[1][col] = v.GetY();
2026         mMat[2][col] = v.GetZ();
2027     }
2028 }
2029
2030 inline Matrix3x3& Matrix3x3::operator+=(const Matrix3x3& m)
2031 {
2032     for (int i = 0; i < 3; ++i)
2033     {
2034         for (int j = 0; j < 3; ++j)
2035         {
2036             this->mMat[i][j] += m.mMat[i][j];
2037         }
2038     }
2039     return *this;
2040 }
2041
2042 inline Matrix3x3& Matrix3x3::operator-=(const Matrix3x3& m)
2043 {
2044     for (int i = 0; i < 3; ++i)
2045     {
2046         for (int j = 0; j < 3; ++j)
2047         {
2048             this->mMat[i][j] -= m.mMat[i][j];
2049         }
2050     }
2051 }
2052

```

```

2053
2054     return *this;
2055 }
2056
2057 inline Matrix3x3& Matrix3x3::operator*=(float k)
2058 {
2059     for (int i = 0; i < 3; ++i)
2060     {
2061         for (int j = 0; j < 3; ++j)
2062         {
2063             this->mMat[i][j] *= k;
2064         }
2065     }
2066     return *this;
2067 }
2068
2069 inline Matrix3x3& Matrix3x3::operator*=(const Matrix3x3& m)
2070 {
2071     Matrix3x3 result;
2072
2073     for (int i = 0; i < 3; ++i)
2074     {
2075         result.mMat[i][0] =
2076             (mMat[i][0] * m.mMat[0][0]) +
2077             (mMat[i][1] * m.mMat[1][0]) +
2078             (mMat[i][2] * m.mMat[2][0]);
2079
2080         result.mMat[i][1] =
2081             (mMat[i][0] * m.mMat[0][1]) +
2082             (mMat[i][1] * m.mMat[1][1]) +
2083             (mMat[i][2] * m.mMat[2][1]);
2084
2085         result.mMat[i][2] =
2086             (mMat[i][0] * m.mMat[0][2]) +
2087             (mMat[i][1] * m.mMat[1][2]) +
2088             (mMat[i][2] * m.mMat[2][2]);
2089     }
2090
2091     for (int i = 0; i < 3; ++i)
2092     {
2093         for (int j = 0; j < 3; ++j)
2094         {
2095             mMat[i][j] = result.mMat[i][j];
2096         }
2097     }
2098
2099     return *this;
2100 }
2101
2102 inline Matrix3x3 operator+(const Matrix3x3& m1, const Matrix3x3& m2)
2103 {
2104     Matrix3x3 result;
2105     for (int i = 0; i < 3; ++i)
2106     {
2107         for (int j = 0; j < 3; ++j)
2108         {
2109             result(i, j) = m1(i, j) + m2(i, j);
2110         }
2111     }
2112
2113     return result;
2114 }
2115
2116 inline Matrix3x3 operator-(const Matrix3x3& m)
2117 {
2118     Matrix3x3 result;
2119     for (int i = 0; i < 3; ++i)
2120     {
2121         for (int j = 0; j < 3; ++j)
2122         {
2123             result(i, j) = -m(i, j);
2124         }
2125     }
2126
2127     return result;
2128 }
2129
2130 inline Matrix3x3 operator-(const Matrix3x3& m1, const Matrix3x3& m2)
2131 {
2132     Matrix3x3 result;
2133     for (int i = 0; i < 3; ++i)
2134     {
2135         for (int j = 0; j < 3; ++j)
2136         {
2137             result(i, j) = m1(i, j) - m2(i, j);
2138         }
2139     }
2140
2141     return result;
2142 }
2143
2144
2145

```

```

2146     }
2147
2148     return result;
2149 }
2150
2151 inline Matrix3x3 operator*(const Matrix3x3& m, const float& k)
2152 {
2153     Matrix3x3 result;
2154     for (int i = 0; i < 3; ++i)
2155     {
2156         for (int j = 0; j < 3; ++j)
2157         {
2158             result(i, j) = m(i, j) * k;
2159         }
2160     }
2161
2162     return result;
2163 }
2164
2165 inline Matrix3x3 operator*(const float& k, const Matrix3x3& m)
2166 {
2167     Matrix3x3 result;
2168     for (int i = 0; i < 3; ++i)
2169     {
2170         for (int j = 0; j < 3; ++j)
2171         {
2172             result(i, j) = k * m(i, j);
2173         }
2174     }
2175
2176     return result;
2177 }
2178
2179 inline Matrix3x3 operator*(const Matrix3x3& m1, const Matrix3x3& m2)
2180 {
2181     Matrix3x3 result;
2182
2183     for (int i = 0; i < 4; ++i)
2184     {
2185         result(i, 0) =
2186             (m1(i, 0) * m2(0, 0)) +
2187             (m1(i, 1) * m2(1, 0)) +
2188             (m1(i, 2) * m2(2, 0));
2189
2190         result(i, 1) =
2191             (m1(i, 0) * m2(0, 1)) +
2192             (m1(i, 1) * m2(1, 1)) +
2193             (m1(i, 2) * m2(2, 1));
2194
2195         result(i, 2) =
2196             (m1(i, 0) * m2(0, 2)) +
2197             (m1(i, 1) * m2(1, 2)) +
2198             (m1(i, 2) * m2(2, 2));
2199
2200         result(i, 3) =
2201             (m1(i, 0) * m2(0, 3)) +
2202             (m1(i, 1) * m2(1, 3)) +
2203             (m1(i, 2) * m2(2, 3));
2204     }
2205
2206     return result;
2207 }
2208
2209 inline Vector3D operator*(const Matrix3x3& m, const Vector3D& v)
2210 {
2211     Vector3D result;
2212
2213     result.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ());
2214     result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ());
2215     result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ());
2216
2217     return result;
2218 }
2219
2220 inline Vector3D operator*(const Vector3D& v, const Matrix3x3& m)
2221 {
2222     Vector3D result;
2223
2224     result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0));
2225     result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1));
2226     result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2));
2227
2228     return result;
2229 }

```

```

2249     }
2250
2253     inline void SetToIdentity(Matrix3x3& m)
2254     {
2255         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
2256
2257         //1st row
2258         m(0, 0) = 1.0f;
2259         m(0, 1) = 0.0f;
2260         m(0, 2) = 0.0f;
2261
2262         //2nd row
2263         m(1, 0) = 0.0f;
2264         m(1, 1) = 1.0f;
2265         m(1, 2) = 0.0f;
2266
2267         //3rd row
2268         m(2, 0) = 0.0f;
2269         m(2, 1) = 0.0f;
2270         m(2, 2) = 1.0f;
2271     }
2272
2275     inline bool IsIdentity(const Matrix3x3& m)
2276     {
2277         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f
2278
2279         for (int i = 0; i < 3; ++i)
2280         {
2281             for (int j = 0; j < 3; ++j)
2282             {
2283                 if (i == j)
2284                 {
2285                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
2286                         return false;
2287                 }
2288                 else
2289                 {
2290                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
2291                         return false;
2292                 }
2293             }
2294         }
2295     }
2296 }
2297
2298
2301     inline Matrix3x3 Transpose(const Matrix3x3& m)
2302     {
2303         //make the rows into cols
2304
2305         Matrix3x3 result;
2306
2307         //1st col = 1st row
2308         result(0, 0) = m(0, 0);
2309         result(1, 0) = m(0, 1);
2310         result(2, 0) = m(0, 2);
2311
2312         //2nd col = 2nd row
2313         result(0, 1) = m(1, 0);
2314         result(1, 1) = m(1, 1);
2315         result(2, 1) = m(1, 2);
2316
2317         //3rd col = 3rd row
2318         result(0, 2) = m(2, 0);
2319         result(1, 2) = m(2, 1);
2320         result(2, 2) = m(2, 2);
2321
2322         return result;
2323     }
2324
2329     inline Matrix3x3 Scale(const Matrix3x3& cm, float x, float y, float z)
2330     {
2331         //x 0 0
2332         //0 y 0
2333         //0 0 z
2334
2335         Matrix3x3 scale;
2336         scale(0, 0) = x;
2337         scale(1, 1) = y;
2338         scale(2, 2) = z;
2339
2340         return cm * scale;
2341     }
2342
2347     inline Matrix3x3 Scale(const Matrix3x3& cm, const Vector3D& scaleVector)
2348     {

```



```

2349         //x 0 0
2350         //0 y 0
2351         //0 0 z
2352
2353         Matrix3x3 scale;
2354         scale(0, 0) = scaleVector.GetX();
2355         scale(1, 1) = scaleVector.GetY();
2356         scale(2, 2) = scaleVector.GetZ();
2357
2358         return cm * scale;
2359     }
2360
2361     inline Matrix3x3 Rotate(const Matrix3x3& cm, float angle, float x, float y, float z)
2362     {
2363         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy
2364         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx
2365         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2
2366         //c = cos(angle)
2367         //s = sin(angle)
2368
2369         Vector3D axis(x, y, z);
2370         axis = Norm(axis);
2371         x = axis.GetX();
2372         y = axis.GetY();
2373         z = axis.GetZ();
2374
2375         float c = cos(angle * PI / 180.0f);
2376         float s = sin(angle * PI / 180.0f);
2377         float oneMinusC = 1.0f - c;
2378
2379         Matrix3x3 result;
2380
2381         //1st row
2382         result(0, 0) = c + (oneMinusC * (x * x));
2383         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2384         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2385
2386         //2nd row
2387         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2388         result(1, 1) = c + (oneMinusC * (y * y));
2389         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2390
2391         //3rd row
2392         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2393         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2394         result(2, 2) = c + (oneMinusC * (z * z));
2395
2396         return cm * result;
2397     }
2398
2399     inline Matrix3x3 Rotate(const Matrix3x3& cm, float angle, const Vector3D& axis)
2400     {
2401         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy
2402         //(1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx
2403         //(1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2
2404         //c = cos(angle)
2405         //s = sin(angle)
2406
2407         Vector3D nAxis(Norm(axis));
2408         float x = nAxis.GetX();
2409         float y = nAxis.GetY();
2410         float z = nAxis.GetZ();
2411
2412         float c = cos(angle * PI / 180.0f);
2413         float s = sin(angle * PI / 180.0f);
2414         float oneMinusC = 1.0f - c;
2415
2416         Matrix3x3 result;
2417
2418         //1st row
2419         result(0, 0) = c + (oneMinusC * (x * x));
2420         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2421         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2422
2423         //2nd row
2424         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2425         result(1, 1) = c + (oneMinusC * (y * y));
2426         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2427
2428         //3rd row
2429         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2430         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2431         result(2, 2) = c + (oneMinusC * (z * z));
2432
2433         return cm * result;
2434     }

```

```

2444     }
2445
2446 inline double Determinant(const Matrix3x3& m)
2447 {
2448     //m00m11m22 - m00m12m21
2449     double c1 = (double)m(0, 0) * m(1, 1) * m(2, 2) - (double)m(0, 0) * m(1, 2) * m(2, 1);
2450
2451     //m01m12m20 - m01m10m22
2452     double c2 = (double)m(0, 1) * m(1, 2) * m(2, 0) - (double)m(0, 1) * m(1, 0) * m(2, 2);
2453
2454     //m02m10m21 - m02m11m20
2455     double c3 = (double)m(0, 2) * m(1, 0) * m(2, 1) - (double)m(0, 2) * m(1, 1) * m(2, 0);
2456
2457     return c1 + c2 + c3;
2458 }
2459
2460 inline double Cofactor(const Matrix3x3& m, unsigned int row, unsigned int col)
2461 {
2462     //cij = ((-1)^(i + j)) * det of minor(i, j);
2463     Matrix2x2 minor;
2464     int r{ 0 };
2465     int c{ 0 };
2466
2467     //minor(i, j)
2468     for (int i = 0; i < 3; ++i)
2469     {
2470         if (i == row)
2471             continue;
2472
2473         for (int j = 0; j < 3; ++j)
2474         {
2475             if (j == col)
2476                 continue;
2477
2478             minor(r, c) = m(i, j);
2479             ++c;
2480         }
2481         c = 0;
2482         ++r;
2483     }
2484
2485     return pow(-1, row + col) * Determinant(minor);
2486 }
2487
2488 inline Matrix3x3 Adjoint(const Matrix3x3& m)
2489 {
2490     //Cofactor of each ijth position put into matrix cA.
2491     //Adjoint is the tranposed matrix of cA.
2492     Matrix3x3 cofactorMatrix;
2493     for (int i = 0; i < 3; ++i)
2494     {
2495         for (int j = 0; j < 3; ++j)
2496         {
2497             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
2498         }
2499     }
2500
2501     return Transpose(cofactorMatrix);
2502 }
2503
2504 inline Matrix3x3 Inverse(const Matrix3x3& m)
2505 {
2506     //Inverse of m = adjoint of m / det of m
2507     double det = Determinant(m);
2508     if (CompareDoubles(det, 0.0, EPSILON))
2509         return Matrix3x3();
2510
2511     return Adjoint(m) * (1.0f / static_cast<float>(det));
2512 }
2513
2514 #if defined(_DEBUG)
2515 inline void print(const Matrix3x3& m)
2516 {
2517     for (int i = 0; i < 3; ++i)
2518     {
2519         for (int j = 0; j < 3; ++j)
2520         {
2521             std::cout << m(i, j) << "\t";
2522         }
2523         std::cout << std::endl;
2524     }
2525 }
2526 #endif
2527
2528 #endif
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540

```

```

2541
2542 //-----
2543
2544
2545 //-----
2553     class Matrix4x4
2554     {
2555     public:
2556
2559         Matrix4x4();
2560
2565         Matrix4x4(float a[][4]);
2566
2569         Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
2570
2577         Matrix4x4(const Matrix2x2& m);
2578
2584         Matrix4x4(const Matrix3x3& m);
2585
2592         Matrix4x4& operator=(const Matrix2x2& m);
2593
2599         Matrix4x4& operator=(const Matrix3x3& m);
2600
2603         float* Data();
2604
2607         const float* Data() const;
2608
2613         const float& operator()(unsigned int row, unsigned int col) const;
2614
2619         float& operator()(unsigned int row, unsigned int col);
2620
2625         Vector4D GetRow(unsigned int row) const;
2626
2631         Vector4D GetCol(unsigned int col) const;
2632
2637         void SetRow(unsigned int row, Vector4D v);
2638
2643         void SetCol(unsigned int col, Vector4D v);
2644
2647         Matrix4x4& operator+=(const Matrix4x4& m);
2648
2651         Matrix4x4& operator-=(const Matrix4x4& m);
2652
2655         Matrix4x4& operator*=(float k);
2656
2659         Matrix4x4& operator*=(const Matrix4x4& m);
2660
2661     private:
2662
2663         float mMat[4][4];
2664     };
2665
2666 //-----
2667     inline Matrix4x4::Matrix4x4()
2668     {
2669         //1st row
2670         mMat[0][0] = 1.0f;
2671         mMat[0][1] = 0.0f;
2672         mMat[0][2] = 0.0f;
2673         mMat[0][3] = 0.0f;
2674
2675         //2nd
2676         mMat[1][0] = 0.0f;
2677         mMat[1][1] = 1.0f;
2678         mMat[1][2] = 0.0f;
2679         mMat[1][3] = 0.0f;
2680
2681         //3rd row
2682         mMat[2][0] = 0.0f;
2683         mMat[2][1] = 0.0f;
2684         mMat[2][2] = 1.0f;
2685         mMat[2][3] = 0.0f;
2686
2687         //4th row
2688         mMat[3][0] = 0.0f;
2689         mMat[3][1] = 0.0f;
2690         mMat[3][2] = 0.0f;
2691         mMat[3][3] = 1.0f;
2692     }
2693
2694
2695
2696     inline Matrix4x4::Matrix4x4(float a[][4])
2697     {
2698         //1st row

```

```

2699     mMat[0][0] = a[0][0];
2700     mMat[0][1] = a[0][1];
2701     mMat[0][2] = a[0][2];
2702     mMat[0][3] = a[0][3];
2703
2704     //2nd
2705     mMat[1][0] = a[1][0];
2706     mMat[1][1] = a[1][1];
2707     mMat[1][2] = a[1][2];
2708     mMat[1][3] = a[1][3];
2709
2710     //3rd row
2711     mMat[2][0] = a[2][0];
2712     mMat[2][1] = a[2][1];
2713     mMat[2][2] = a[2][2];
2714     mMat[2][3] = a[2][3];
2715
2716     //4th row
2717     mMat[3][0] = a[3][0];
2718     mMat[3][1] = a[3][1];
2719     mMat[3][2] = a[3][2];
2720     mMat[3][3] = a[3][3];
2721 }
2722
2723 inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
2724 {
2725     SetRow(0, r1);
2726     SetRow(1, r2);
2727     SetRow(2, r3);
2728     SetRow(3, r4);
2729 }
2730
2731 inline float* Matrix4x4::Data()
2732 {
2733     return mMat[0];
2734 }
2735
2736 inline const float* Matrix4x4::Data()const
2737 {
2738     return mMat[0];
2739 }
2740
2741 inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
2742 {
2743     if (row > 3 || col > 3)
2744     {
2745         return mMat[0][0];
2746     }
2747     else
2748     {
2749         return mMat[row][col];
2750     }
2751 }
2752
2753 inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
2754 {
2755     if (row > 3 || col > 3)
2756     {
2757         return mMat[0][0];
2758     }
2759     else
2760     {
2761         return mMat[row][col];
2762     }
2763 }
2764
2765 inline Vector4D Matrix4x4::GetRow(unsigned int row)const
2766 {
2767     if (row < 0 || row > 3)
2768         return Vector4D(mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3]);
2769     else
2770         return Vector4D(mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3]);
2771 }
2772
2773 inline Vector4D Matrix4x4::GetCol(unsigned int col)const
2774 {
2775     if (col < 0 || col > 3)
2776         return Vector4D(mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0]);
2777     else
2778         return Vector4D(mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col]);
2779 }
2780
2781 inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
2782 {
2783     if (row > 3)

```

```

2785     {
2786         mMat[0][0] = v.GetX();
2787         mMat[0][1] = v.GetY();
2788         mMat[0][2] = v.GetZ();
2789         mMat[0][3] = v.GetW();
2790     }
2791     else
2792     {
2793         mMat[row][0] = v.GetX();
2794         mMat[row][1] = v.GetY();
2795         mMat[row][2] = v.GetZ();
2796         mMat[row][3] = v.GetW();
2797     }
2798 }
2799
2800 inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
2801 {
2802     if (col > 3)
2803     {
2804         mMat[0][0] = v.GetX();
2805         mMat[1][0] = v.GetY();
2806         mMat[2][0] = v.GetZ();
2807         mMat[3][0] = v.GetW();
2808     }
2809     else
2810     {
2811         mMat[0][col] = v.GetX();
2812         mMat[1][col] = v.GetY();
2813         mMat[2][col] = v.GetZ();
2814         mMat[3][col] = v.GetW();
2815     }
2816 }
2817
2818 inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
2819 {
2820     for (int i = 0; i < 4; ++i)
2821     {
2822         for (int j = 0; j < 4; ++j)
2823         {
2824             this->mMat[i][j] += m.mMat[i][j];
2825         }
2826     }
2827
2828     return *this;
2829 }
2830
2831 inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
2832 {
2833     for (int i = 0; i < 4; ++i)
2834     {
2835         for (int j = 0; j < 4; ++j)
2836         {
2837             this->mMat[i][j] -= m.mMat[i][j];
2838         }
2839     }
2840
2841     return *this;
2842 }
2843
2844 inline Matrix4x4& Matrix4x4::operator*=(float k)
2845 {
2846     for (int i = 0; i < 4; ++i)
2847     {
2848         for (int j = 0; j < 4; ++j)
2849         {
2850             this->mMat[i][j] *= k;
2851         }
2852     }
2853
2854     return *this;
2855 }
2856
2857 inline Matrix4x4& Matrix4x4::operator*(const Matrix4x4& m)
2858 {
2859     Matrix4x4 result;
2860
2861     for (int i = 0; i < 4; ++i)
2862     {
2863         result.mMat[i][0] =
2864             (mMat[i][0] * m.mMat[0][0]) +
2865             (mMat[i][1] * m.mMat[1][0]) +
2866             (mMat[i][2] * m.mMat[2][0]) +
2867             (mMat[i][3] * m.mMat[3][0]);
2868
2869         result.mMat[i][1] =
2870             (mMat[i][0] * m.mMat[0][1]) +
2871             (mMat[i][1] * m.mMat[1][1]) +

```

```

2872         (mMat[i][2] * m.mMat[2][1]) +
2873         (mMat[i][3] * m.mMat[3][1]));
2874
2875     result.mMat[i][2] =
2876         (mMat[i][0] * m.mMat[0][2]) +
2877         (mMat[i][1] * m.mMat[1][2]) +
2878         (mMat[i][2] * m.mMat[2][2]) +
2879         (mMat[i][3] * m.mMat[3][2]);
2880
2881     result.mMat[i][3] =
2882         (mMat[i][0] * m.mMat[0][3]) +
2883         (mMat[i][1] * m.mMat[1][3]) +
2884         (mMat[i][2] * m.mMat[2][3]) +
2885         (mMat[i][3] * m.mMat[3][3]);
2886 }
2887
2888 for (int i = 0; i < 4; ++i)
2889 {
2890     for (int j = 0; j < 4; ++j)
2891     {
2892         mMat[i][j] = result.mMat[i][j];
2893     }
2894 }
2895
2896 return *this;
2897 }
2898
2899 inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
2900 {
2901     Matrix4x4 result;
2902     for (int i = 0; i < 4; ++i)
2903     {
2904         for (int j = 0; j < 4; ++j)
2905         {
2906             result(i, j) = m1(i, j) + m2(i, j);
2907         }
2908     }
2909
2910     return result;
2911 }
2912
2913 inline Matrix4x4 operator-(const Matrix4x4& m)
2914 {
2915     Matrix4x4 result;
2916     for (int i = 0; i < 4; ++i)
2917     {
2918         for (int j = 0; j < 4; ++j)
2919         {
2920             result(i, j) = -m(i, j);
2921         }
2922     }
2923
2924     return result;
2925 }
2926
2927 inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
2928 {
2929     Matrix4x4 result;
2930     for (int i = 0; i < 4; ++i)
2931     {
2932         for (int j = 0; j < 4; ++j)
2933         {
2934             result(i, j) = m1(i, j) - m2(i, j);
2935         }
2936     }
2937
2938     return result;
2939 }
2940
2941 inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
2942 {
2943     Matrix4x4 result;
2944     for (int i = 0; i < 4; ++i)
2945     {
2946         for (int j = 0; j < 4; ++j)
2947         {
2948             result(i, j) = m(i, j) * k;
2949         }
2950     }
2951
2952     return result;
2953 }
2954
2955 inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
2956 {
2957     Matrix4x4 result;
2958     for (int i = 0; i < 4; ++i)

```

```

2969     {
2970         for (int j = 0; j < 4; ++j)
2971         {
2972             result(i, j) = k * m(i, j);
2973         }
2974     }
2975
2976     return result;
2977 }
2978
2979 inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
2980 {
2981     Matrix4x4 result;
2982
2983     for (int i = 0; i < 4; ++i)
2984     {
2985         result(i, 0) =
2986             (m1(i, 0) * m2(0, 0)) +
2987             (m1(i, 1) * m2(1, 0)) +
2988             (m1(i, 2) * m2(2, 0)) +
2989             (m1(i, 3) * m2(3, 0));
2990
2991         result(i, 1) =
2992             (m1(i, 0) * m2(0, 1)) +
2993             (m1(i, 1) * m2(1, 1)) +
2994             (m1(i, 2) * m2(2, 1)) +
2995             (m1(i, 3) * m2(3, 1));
2996
2997         result(i, 2) =
2998             (m1(i, 0) * m2(0, 2)) +
2999             (m1(i, 1) * m2(1, 2)) +
3000             (m1(i, 2) * m2(2, 2)) +
3001             (m1(i, 3) * m2(3, 2));
3002
3003         result(i, 3) =
3004             (m1(i, 0) * m2(0, 3)) +
3005             (m1(i, 1) * m2(1, 3)) +
3006             (m1(i, 2) * m2(2, 3)) +
3007             (m1(i, 3) * m2(3, 3));
3008     }
3009
3010     return result;
3011 }
3012
3013 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
3014 {
3015     Vector4D result;
3016
3017     result.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());
3018     result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());
3019     result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());
3020     result.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());
3021
3022     return result;
3023 }
3024
3025 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
3026 {
3027     Vector4D result;
3028
3029     result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));
3030     result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));
3031     result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));
3032     result.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * m(3, 3));
3033
3034     return result;
3035 }
3036
3037 inline void SetToIdentity(Matrix4x4& m)
3038 {
3039     //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
3040
3041     //1st row
3042     m(0, 0) = 1.0f;
3043     m(0, 1) = 0.0f;
3044     m(0, 2) = 0.0f;
3045     m(0, 3) = 0.0f;
3046
3047     //2nd row
3048     m(1, 0) = 0.0f;
3049     m(1, 1) = 1.0f;

```

```

3070         m(1, 2) = 0.0f;
3071         m(1, 3) = 0.0f;
3072
3073         //3rd row
3074         m(2, 0) = 0.0f;
3075         m(2, 1) = 0.0f;
3076         m(2, 2) = 1.0f;
3077         m(2, 3) = 0.0f;
3078
3079         //4th row
3080         m(3, 0) = 0.0f;
3081         m(3, 1) = 0.0f;
3082         m(3, 2) = 0.0f;
3083         m(3, 3) = 1.0f;
3084     }
3085
3086     inline bool IsIdentity(const Matrix4x4& m)
3087     {
3088         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
3089         0.0f
3090
3091         for (int i = 0; i < 4; ++i)
3092         {
3093             for (int j = 0; j < 4; ++j)
3094             {
3095                 if (i == j)
3096                 {
3097                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
3098                         return false;
3099                 }
3100                 else
3101                 {
3102                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
3103                         return false;
3104                 }
3105             }
3106         }
3107     }
3108
3109     inline Matrix4x4 Transpose(const Matrix4x4& m)
3110     {
3111         //make the rows into cols
3112
3113         Matrix4x4 result;
3114
3115         //1st col = 1st row
3116         result(0, 0) = m(0, 0);
3117         result(1, 0) = m(0, 1);
3118         result(2, 0) = m(0, 2);
3119         result(3, 0) = m(0, 3);
3120
3121         //2nd col = 2nd row
3122         result(0, 1) = m(1, 0);
3123         result(1, 1) = m(1, 1);
3124         result(2, 1) = m(1, 2);
3125         result(3, 1) = m(1, 3);
3126
3127         //3rd col = 3rd row
3128         result(0, 2) = m(2, 0);
3129         result(1, 2) = m(2, 1);
3130         result(2, 2) = m(2, 2);
3131         result(3, 2) = m(2, 3);
3132
3133         //4th col = 4th row
3134         result(0, 3) = m(3, 0);
3135         result(1, 3) = m(3, 1);
3136         result(2, 3) = m(3, 2);
3137         result(3, 3) = m(3, 3);
3138
3139         return result;
3140     }
3141
3142     inline Matrix4x4 Translate(const Matrix4x4& cm, float x, float y, float z)
3143     {
3144         //1 0 0 0
3145         //0 1 0 0
3146         //0 0 1 0
3147         //x y z 1
3148
3149         Matrix4x4 translate;
3150         translate(3, 0) = x;
3151         translate(3, 1) = y;
3152         translate(3, 2) = z;
3153
3154         return cm * translate;
3155     }

```



```

3164     }
3165
3170     inline Matrix4x4 Translate(const Matrix4x4& cm, const Vector3D& translateVector)
3171     {
3172         //1 0 0 0
3173         //0 1 0 0
3174         //0 0 1 0
3175         //x y z 1
3176
3177         Matrix4x4 translate;
3178         translate(3, 0) = translateVector.GetX();
3179         translate(3, 1) = translateVector.GetY();
3180         translate(3, 2) = translateVector.GetZ();
3181
3182         return cm * translate;
3183     }
3184
3189     inline Matrix4x4 Scale(const Matrix4x4& cm, float x, float y, float z)
3190     {
3191         //x 0 0 0
3192         //0 y 0 0
3193         //0 0 z 0
3194         //0 0 0 1
3195
3196         Matrix4x4 scale;
3197         scale(0, 0) = x;
3198         scale(1, 1) = y;
3199         scale(2, 2) = z;
3200
3201         return cm * scale;
3202     }
3203
3208     inline Matrix4x4 Scale(const Matrix4x4& cm, const Vector3D& scaleVector)
3209     {
3210         //x 0 0 0
3211         //0 y 0 0
3212         //0 0 z 0
3213         //0 0 0 1
3214
3215         Matrix4x4 scale;
3216         scale(0, 0) = scaleVector.GetX();
3217         scale(1, 1) = scaleVector.GetY();
3218         scale(2, 2) = scaleVector.GetZ();
3219
3220         return cm * scale;
3221     }
3222
3227     inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, float x, float y, float z)
3228     {
3229         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy    0
3230         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx    0
3231         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2    0
3232         //0                  0                  0                  1
3233         //c = cos(angle)
3234         //s = sin(angle)
3235
3236         Vector3D axis(x, y, z);
3237
3238         axis = Norm(axis);
3239
3240         x = axis.GetX();
3241         y = axis.GetY();
3242         z = axis.GetZ();
3243
3244         float c = cos(angle * PI / 180.0f);
3245         float s = sin(angle * PI / 180.0f);
3246         float oneMinusC = 1 - c;
3247
3248         Matrix4x4 result;
3249
3250         //1st row
3251         result(0, 0) = c + (oneMinusC * (x * x));
3252         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
3253         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
3254
3255         //2nd row
3256         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
3257         result(1, 1) = c + (oneMinusC * (y * y));
3258         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
3259
3260         //3rd row
3261         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
3262         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
3263         result(2, 2) = c + (oneMinusC * (z * z));
3264
3265         return cm * result;
3266     }

```

```

3267
3272 inline Matrix4x4 Rotate(const Matrix4x4& cm, float angle, const Vector3D& axis)
3273 {
3274     //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy    0
3275     //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx    0
3276     //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2    0
3277     //0                0                0                1
3278     //c = cos(angle)
3279     //s = sin(angle)
3280
3281     Vector3D nAxis(Norm(axis));
3282
3283     float x = nAxis.GetX();
3284     float y = nAxis.GetY();
3285     float z = nAxis.GetZ();
3286
3287     float c = cos(angle * PI / 180.0f);
3288     float s = sin(angle * PI / 180.0f);
3289     float oneMinusC = 1 - c;
3290
3291     Matrix4x4 result;
3292
3293     //1st row
3294     result(0, 0) = c + (oneMinusC * (x * x));
3295     result(0, 1) = (oneMinusC * (x * y)) + (s * z);
3296     result(0, 2) = (oneMinusC * (x * z)) - (s * y);
3297
3298     //2nd row
3299     result(1, 0) = (oneMinusC * (x * y)) - (s * z);
3300     result(1, 1) = c + (oneMinusC * (y * y));
3301     result(1, 2) = (oneMinusC * (y * z)) + (s * x);
3302
3303     //3rd row
3304     result(2, 0) = (oneMinusC * (x * z)) + (s * y);
3305     result(2, 1) = (oneMinusC * (y * z)) - (s * x);
3306     result(2, 2) = c + (oneMinusC * (z * z));
3307
3308     return cm * result;
3309 }
3310
3311 inline double Determinant(const Matrix4x4& m)
3312 {
3313     //m00m11(m22m33 - m23m32)
3314     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)
3315 * m(3, 2);
3316
3317     //m00m12(m23m31 - m21m33)
3318     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)
3319 * m(3, 3);
3320
3321     //m00m13(m21m32 - m22m31)
3322     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)
3323 * m(3, 1);
3324
3325     //m01m10(m22m33 - m23m32)
3326     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)
3327 * m(3, 2);
3328
3329     //m01m12(m23m30 - m20m33)
3330     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)
3331 * m(3, 3);
3332
3333     //m01m13(m20m32 - m22m30)
3334     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)
3335 * m(3, 0);
3336
3337     //m02m10(m21m33 - m23m31)
3338     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)
3339 * m(3, 1);
3340
3341     //m02m11(m23m30 - m20m33)
3342     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)
3343 * m(3, 3);
3344
3345     //m02m13(m20m31 - m21m30)
3346     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)
3347 * m(3, 0);
3348
3349     //m03m10(m21m32 - m22m21)
3350     double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
3351 2) * m(3, 1);
3352
3353     //m03m11(m22m30 - m20m32)
3354     double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
3355 0) * m(3, 2);
3356
3357     //m03m12(m20m31 - m21m30)

```

```

3349     double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
3350 1) * m(3, 0);
3351     return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
3352 }
3353
3354 inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
3355 {
3356     //cij = (-1)^(i + j) * det of minor(i, j);
3357     Matrix3x3 minor;
3358     int r{ 0 };
3359     int c{ 0 };
3360
3361     //minor(i, j)
3362     for (int i = 0; i < 4; ++i)
3363     {
3364         if (i == row)
3365             continue;
3366
3367         for (int j = 0; j < 4; ++j)
3368         {
3369             if (j == col)
3370                 continue;
3371
3372             minor(r, c) = m(i, j);
3373             ++c;
3374         }
3375         c = 0;
3376         ++r;
3377     }
3378
3379     return pow(-1, row + col) * Determinant(minor);
3380 }
3381
3382 inline Matrix4x4 Adjoint(const Matrix4x4& m)
3383 {
3384     //Cofactor of each ijth position put into matrix cA.
3385     //Adjoint is the tranposed matrix of cA.
3386     Matrix4x4 cofactorMatrix;
3387     for (int i = 0; i < 4; ++i)
3388     {
3389         for (int j = 0; j < 4; ++j)
3390         {
3391             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
3392         }
3393     }
3394
3395     return Transpose(cofactorMatrix);
3396 }
3397
3398 inline Matrix4x4 Inverse(const Matrix4x4& m)
3399 {
3400     //Inverse of m = adjoint of m / det of m
3401     double det = Determinant(m);
3402     if (CompareDoubles(det, 0.0, EPSILON))
3403         return Matrix4x4();
3404
3405     return Adjoint(m) * (1.0f / static_cast<float>(det));
3406 }
3407
3408 #if defined(_DEBUG)
3409 inline void print(const Matrix4x4& m)
3410 {
3411     for (int i = 0; i < 4; ++i)
3412     {
3413         for (int j = 0; j < 4; ++j)
3414         {
3415             std::cout << m(i, j) << " ";
3416         }
3417
3418         std::cout << std::endl;
3419     }
3420 }
3421 #endif
3422
3423 //-----
3424
3425 //-----
3426
3427 class Quaternion

```

```

3446     {
3447     public:
3452         Quaternion(float scalar = 1.0f, float x = 0.0f, float y = 0.0f, float z = 0.0f);
3453
3456         Quaternion(float scalar, const Vector3D& v);
3457
3463         Quaternion(const Vector4D& v);
3464
3467         float GetScalar() const;
3468
3471         float GetX() const;
3472
3475         float GetY() const;
3476
3479         float GetZ() const;
3480
3483         Vector3D GetVector() const;
3484
3487         void SetScalar(float scalar);
3488
3491         void SetX(float x);
3492
3495         void SetY(float y);
3496
3499         void SetZ(float z);
3500
3503         void SetVector(const Vector3D& v);
3504
3507         Quaternion& operator+=(const Quaternion& q);
3508
3511         Quaternion& operator-=(const Quaternion& q);
3512
3515         Quaternion& operator*=(float k);
3516
3519         Quaternion& operator*=(const Quaternion& q);
3520
3521     private:
3522         float mScalar;
3523         float mX;
3524         float mY;
3525         float mZ;
3526     };
3527
3528     //-----
3529     inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
3530         mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
3531     {}
3532
3533     inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
3534         mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
3535     {}
3536
3537     inline Quaternion::Quaternion(const Vector4D& v) :
3538         mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
3539     {}
3540
3541     inline float Quaternion::GetScalar()const
3542     {
3543         return mScalar;
3544     }
3545
3546     inline float Quaternion::GetX()const
3547     {
3548         return mX;
3549     }
3550
3551     inline float Quaternion::GetY()const
3552     {
3553         return mY;
3554     }
3555
3556     inline float Quaternion::GetZ()const
3557     {
3558         return mZ;
3559     }
3560
3561     inline Vector3D Quaternion::GetVector()const
3562     {
3563         return Vector3D(mX, mY, mZ);
3564     }
3565
3566     inline void Quaternion::SetScalar(float scalar)
3567     {
3568         mScalar = scalar;
3569     }
3570
3571     inline void Quaternion::SetX(float x)

```

```

3572     {
3573         mX = x;
3574     }
3575
3576     inline void Quaternion::SetY(float y)
3577     {
3578         mY = y;
3579     }
3580
3581     inline void Quaternion::SetZ(float z)
3582     {
3583         mZ = z;
3584     }
3585
3586     inline void Quaternion::SetVector(const Vector3D& v)
3587     {
3588         mX = v.GetX();
3589         mY = v.GetY();
3590         mZ = v.GetZ();
3591     }
3592
3593     inline Quaternion& Quaternion::operator+=(const Quaternion& q)
3594     {
3595         this->mScalar += q.mScalar;
3596         this->mX += q.mX;
3597         this->mY += q.mY;
3598         this->mZ += q.mZ;
3599
3600         return *this;
3601     }
3602
3603     inline Quaternion& Quaternion::operator-=(const Quaternion& q)
3604     {
3605         this->mScalar -= q.mScalar;
3606         this->mX -= q.mX;
3607         this->mY -= q.mY;
3608         this->mZ -= q.mZ;
3609
3610         return *this;
3611     }
3612
3613     inline Quaternion& Quaternion::operator*=(float k)
3614     {
3615         this->mScalar *= k;
3616         this->mX *= k;
3617         this->mY *= k;
3618         this->mZ *= k;
3619
3620         return *this;
3621     }
3622
3623     inline Quaternion& Quaternion::operator*=(const Quaternion& q)
3624     {
3625         Vector3D thisVector(this->mX, this->mY, this->mZ);
3626         Vector3D qVector(q.mX, q.mY, q.mZ);
3627
3628         float scalar{ this->mScalar * q.mScalar };
3629         float dotProduct{ DotProduct(thisVector, qVector) };
3630         float resultScalar{ scalar - dotProduct };
3631
3632         Vector3D a(this->mScalar * qVector);
3633         Vector3D b(q.mScalar * thisVector);
3634         Vector3D crossProduct(CrossProduct(thisVector, qVector));
3635         Vector3D resultVector(a + b + crossProduct);
3636
3637         this->mScalar = resultScalar;
3638         this->mX = resultVector.GetX();
3639         this->mY = resultVector.GetY();
3640         this->mZ = resultVector.GetZ();
3641
3642         return *this;
3643     }
3644
3645     inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
3646     {
3647         return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
3648             q2.GetY(), q1.GetZ() + q2.GetZ());
3649     }
3650
3651     inline Quaternion operator-(const Quaternion& q)
3652     {
3653         return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
3654     }
3655
3656     inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
3657     {
3658         return Quaternion(q1.GetScalar() - q2.GetScalar(),

```

```

3664         q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
3665     }
3666
3669     inline Quaternion operator*(float k, const Quaternion& q)
3670     {
3671         return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
3672     }
3673
3676     inline Quaternion operator*(const Quaternion& q, float k)
3677     {
3678         return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
3679     }
3680
3683     inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
3684     {
3685         //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
3686         //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector
3687
3688         Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
3689         Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());
3690
3691         float scalar{ q1.GetScalar() * q2.GetScalar() };
3692         float dotProduct{ DotProduct(q1Vector, q2Vector) };
3693         float resultScalar{ scalar - dotProduct };
3694
3695         Vector3D a(q1.GetScalar() * q2Vector);
3696         Vector3D b(q2.GetScalar() * q1Vector);
3697         Vector3D crossProduct(CrossProduct(q1Vector, q2Vector));
3698         Vector3D resultVector(a + b + crossProduct);
3699
3700         return Quaternion(resultScalar, resultVector);
3701     }
3702
3705     inline bool operator==(const Quaternion& q1, const Quaternion& q2)
3706     {
3707         return CompareFloats(q1.GetScalar(), q2.GetScalar(), 1e-6f) && CompareFloats(q1.GetX(),
3708         q2.GetX(), 1e-6f) &&
3709             CompareFloats(q1.GetY(), q2.GetY(), 1e-6f) && CompareFloats(q1.GetZ(), q2.GetZ(), 1e-6f);
3710     }
3711
3713     inline bool operator!=(const Quaternion& q1, const Quaternion& q2)
3714     {
3715         return !operator==(q1, q2);
3716     }
3717
3720     inline bool IsZeroQuaternion(const Quaternion& q)
3721     {
3722         //zero quaternion = (0, 0, 0, 0)
3723         return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
3724             CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
3725     }
3726
3729     inline bool IsIdentity(const Quaternion& q)
3730     {
3731         //identity quaternion = (1, 0, 0, 0)
3732         return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
3733             CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
3734     }
3735
3738     inline Quaternion Conjugate(const Quaternion& q)
3739     {
3740         //conjugate of a quaternion is the quaternion with its vector part negated
3741         return Quaternion(q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
3742     }
3743
3746     inline float Length(const Quaternion& q)
3747     {
3748         //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
3749         return sqrt(q.GetScalar() * q.GetScalar() + q.GetX() * q.GetX() + q.GetY() * q.GetY() +
3750         q.GetZ() * q.GetZ());
3751     }
3752
3756     inline Quaternion Normalize(const Quaternion& q)
3757     {
3758         //to normalize a quaternion you do q / |q|
3759
3760         if (IsZeroQuaternion(q))
3761             return q;
3762
3763         float magnitdue{ Length(q) };
3764
3765         return Quaternion(q.GetScalar() / magnitdue, q.GetX() / magnitdue, q.GetY() / magnitdue,
3766         q.GetZ() / magnitdue);
3767     }
3772     inline Quaternion Inverse(const Quaternion& q)
3773     {

```

```

3774         //inverse = conjugate of q / |q|
3775
3776         if (IsZeroQuaternion(q))
3777             return q;
3778
3779         Quaternion conjugateOfQ(Conjugate(q));
3780
3781         float magnitdue{ Length(q) };
3782
3783         return Quaternion(conjugateOfQ.GetScalar() / magnitdue, conjugateOfQ.GetX() / magnitdue,
3784             conjugateOfQ.GetY() / magnitdue, conjugateOfQ.GetZ() / magnitdue);
3785     }
3786
3791     inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
3792     {
3793         //A roatation quaternion is a quaternion where the
3794         //scalar part = cos(theta / 2)
3795         //vector part = sin(theta / 2) * axis
3796         //the axis needs to be normalized
3797
3798         float ang{ angle / 2.0f };
3799         float c{ cos(ang * PI / 180.0f) };
3800         float s{ sin(ang * PI / 180.0f) };
3801
3802         Vector3D axis(x, y, z);
3803         axis = Norm(axis);
3804
3805         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
3806     }
3807
3812     inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
3813     {
3814         //A roatation quaternion is a quaternion where the
3815         //scalar part = cos(theta / 2)
3816         //vector part = sin(theta / 2) * axis
3817         //the axis needs to be normalized
3818
3819         float ang{ angle / 2.0f };
3820         float c{ cos(ang * PI / 180.0f) };
3821         float s{ sin(ang * PI / 180.0f) };
3822
3823         Vector3D axisN(Norm(axis));
3824
3825         return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
3826     }
3827
3833     inline Quaternion RotationQuaternion(const Vector4D& angAxis)
3834     {
3835         //A roatation quaternion is a quaternion where the
3836         //scalar part = cos(theta / 2)
3837         //vector part = sin(theta / 2) * axis
3838         //the axis needs to be normalized
3839
3840         float angle{ angAxis.GetX() / 2.0f };
3841         float c{ cos(angle * PI / 180.0f) };
3842         float s{ sin(angle * PI / 180.0f) };
3843
3844         Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
3845         axis = Norm(axis);
3846
3847         return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
3848     }
3849
3854     inline Matrix4x4 QuaternionToRotationMatrixCol(const Quaternion& q)
3855     {
3856         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
3857         //2q2q3 + 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
3858         //2q2q4 - 2q1q3      2q3q4 + 2q1q2      1 - 2q2^2 - 2q3^2      0
3859         //0                  0                  0                  1
3860         //q1 = scalar
3861         //q2 = x
3862         //q3 = y
3863         //q4 = z
3864
3865         Matrix4x4 colMat;
3866
3867         colMat(0, 0) = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
3868         colMat(0, 1) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
3869         colMat(0, 2) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
3870
3871         colMat(1, 0) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
3872         colMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
3873         colMat(1, 2) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
3874
3875         colMat(2, 0) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
3876         colMat(2, 1) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
3877         colMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();

```

```

3878
3879         return colMat;
3880     }
3881
3882     inline Matrix4x4 QuaternionToRotationMatrixRow(const Quaternion& q)
3883     {
3884         //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
3885         //2q2q3 - 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
3886         //2q2q4 + 2q1q3          2q3q4 - 2q1q2          1 - 2q2^2 - 2q3^2      0
3887         //0                      0                      0                      1
3888         //q1 = scalar
3889         //q2 = x
3890         //q3 = y
3891         //q4 = z
3892
3893         Matrix4x4 rowMat;
3894
3895         rowMat(0, 0) = 1.0f - 2.0f * q.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
3896         rowMat(0, 1) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
3897         rowMat(0, 2) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
3898
3899         rowMat(1, 0) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
3900         rowMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
3901         rowMat(1, 2) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
3902
3903         rowMat(2, 0) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
3904         rowMat(2, 1) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
3905         rowMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
3906
3907         return rowMat;
3908     }
3909
3910     inline Vector3D Rotate(const Quaternion& q, const Vector3D& p)
3911     {
3912         //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
3913         //point/vector, q is a rotation quaternion
3914         //and q* is its conjugate.
3915
3916         Quaternion point(0.0f, p);
3917
3918         Quaternion result(q * point * Conjugate(q));
3919
3920         return result.GetVector();
3921     }
3922
3923     inline Vector4D Rotate(const Quaternion& q, const Vector4D& p)
3924     {
3925         //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
3926         //point/vector, q is a rotation quaternion
3927         //and q* is its conjugate.
3928
3929         Quaternion point(0.0f, p);
3930
3931         Quaternion result(q * point * Conjugate(q));
3932
3933         return Vector4D(result.GetVector(), p.GetW());
3934     }
3935
3936     inline float DotProduct(const Quaternion& q1, const Quaternion& q2)
3937     {
3938         return q1.GetScalar() * q2.GetScalar() + q1.GetX() * q2.GetX() + q1.GetY() * q2.GetY() +
3939         q1.GetZ() * q2.GetZ();
3940     }
3941
3942     inline Quaternion Lerp(const Quaternion& q0, const Quaternion& q1, float t)
3943     {
3944         if (t < 0.0f)
3945             t = 0.0f;
3946         else if (t > 1.0f)
3947             t = 1.0f;
3948
3949         //Compute the cosine of the angle between the quaternions
3950         float cosOmega = DotProduct(q0, q1);
3951
3952         Quaternion newQ1;
3953         //If the dot product is negative, negate q1 to so we take the shorter arc
3954         if (cosOmega < 0.0f)
3955         {
3956             newQ1 = -q1;
3957             cosOmega = -cosOmega;
3958         }
3959         else
3960         {
3961             newQ1 = q1;
3962         }
3963
3964         return (1.0f - t) * q0 + t * q1;
3965     }

```



```

3981     }
3982
3983     inline Quaternion NLERp(const Quaternion& q0, const Quaternion& q1, float t)
3984     {
3985         if (t < 0.0f)
3986             t = 0.0f;
3987         else if (t > 1.0f)
3988             t = 1.0f;
3989
3990         //Compute the cosine of the angle between the quaternions
3991         float cosOmega = DotProduct(q0, q1);
3992
3993         Quaternion newQ1;
3994         //If the dot product is negative, negate q1 to so we take the shorter arc
3995         if (cosOmega < 0.0f)
3996         {
3997             newQ1 = -q1;
3998             cosOmega = -cosOmega;
3999         }
4000         else
4001         {
4002             newQ1 = q1;
4003         }
4004
4005         return Normalize((1.0f - t) * q0 + t * q1);
4006     }
4007
4008     inline Quaternion Slerp(const Quaternion& q0, const Quaternion& q1, float t)
4009     {
4010         //Formula used is
4011         //k0 = sin((1 - t)omega) * omega) / sin(omega);
4012         //k1 = (sin(tomega) * omega) / sin(omega)
4013         //newQ = k0q0 * k1q1
4014         //Omega is the angle between the q0 and q1.
4015
4016         if (t < 0.0f)
4017             t = 0.0f;
4018         else if (t > 1.0f)
4019             t = 1.0f;
4020
4021         //Compute the cosine of the angle between the quaternions
4022         float cosOmega = DotProduct(q0, q1);
4023
4024         Quaternion newQ1;
4025         //If the dot product is negative, negate q1 to so we take the shorter arc
4026         if (cosOmega < 0.0f)
4027         {
4028             newQ1 = -q1;
4029             cosOmega = -cosOmega;
4030         }
4031         else
4032         {
4033             newQ1 = q1;
4034         }
4035
4036         float k0( 0.0f );
4037         float k1( 0.0f );
4038
4039         //Linear interpolate if the quaternions are very close to protect dividing by zero.
4040         if (cosOmega > 0.9999f)
4041         {
4042             k0 = 1.0f - t;
4043             k1 = t;
4044         }
4045         else
4046         {
4047             //sin of the angle between the quaternions is
4048             //sin(omega) = 1 - cos^2(omega) from the trig identity
4049             //sin^2(omega) + cos^2(omega) = 1.
4050             float sinOmega{ sqrt(1.0f - cosOmega * cosOmega) };
4051
4052             //retrieve the angle
4053             float omega{ atan2(sinOmega, cosOmega) };
4054
4055             //Compute inverse to avoid dividng multiple times
4056             float oneOverSinOmega{ 1.0f / sinOmega };
4057
4058             k0 = sin((1.0f - t) * omega) * oneOverSinOmega;
4059             k1 = sin(t * omega) * oneOverSinOmega;
4060         }
4061
4062         return k0 * q0 + k1 * newQ1;
4063     }
4064
4065     #if defined(_DEBUG)
4066     inline void print(const Quaternion& q)
4067     {

```

```

4076         std::cout << "(" << q.GetScalar() << ", " << q.GetX() << ", " << q.GetY() << ", " << q.GetZ();
4077     }
4078 #endif
4079     //-----
4080
4081     inline Vector2D::Vector2D(const Vector3D& v) : mX{ v.GetX() }, mY{ v.GetY() }
4082     {}
4083
4084     inline Vector2D::Vector2D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }
4085     {}
4086
4087     inline Vector2D& Vector2D::operator=(const Vector3D& v)
4088     {
4089         mX = v.GetX();
4090         mY = v.GetY();
4091
4092         return *this;
4093     }
4094
4095     inline Vector2D& Vector2D::operator=(const Vector4D& v)
4096     {
4097         mX = v.GetX();
4098         mY = v.GetY();
4099
4100         return *this;
4101     }
4102
4103     inline Vector3D::Vector3D(const Vector2D& v, float z) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ z }
4104     {}
4105
4106     inline Vector3D::Vector3D(const Vector4D& v) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
4107     {}
4108
4109     inline Vector3D& Vector3D::operator=(const Vector2D& v)
4110     {
4111         mX = v.GetX();
4112         mY = v.GetY();
4113         mZ = 0.0f;
4114
4115         return *this;
4116     }
4117
4118     inline Vector3D& Vector3D::operator=(const Vector4D& v)
4119     {
4120         mX = v.GetX();
4121         mY = v.GetY();
4122         mZ = v.GetZ();
4123
4124         return *this;
4125     }
4126
4127     inline Vector4D::Vector4D(const Vector2D& v, float z, float w) : mX{ v.GetX() }, mY{ v.GetY() },
4128     mZ{ z }, mW{ w }
4129     {}
4130
4131     inline Vector4D::Vector4D(const Vector3D& v, float w) : mX{ v.GetX() }, mY{ v.GetY() }, mZ{
4132     v.GetZ() }, mW{ w }
4133     {}
4134
4135     inline Vector4D& Vector4D::operator=(const Vector2D& v)
4136     {
4137         mX = v.GetX();
4138         mY = v.GetY();
4139         mZ = 0.0f;
4140         mW = 0.0f;
4141
4142         return *this;
4143     }
4144
4145     inline Vector4D& Vector4D::operator=(const Vector3D& v)
4146     {
4147         mX = v.GetX();
4148         mY = v.GetY();
4149         mZ = v.GetZ();
4150         mW = 0.0f;
4151
4152         return *this;
4153     }
4154
4155     inline Matrix2x2::Matrix2x2(const Matrix3x3& m)
4156     {
4157         //1st row
4158         mMat[0][0] = m(0, 0);
4159         mMat[0][1] = m(0, 1);
4160
4161         //2nd row
4162         mMat[1][0] = m(1, 0);

```

```

4161         mMat[1][1] = m(1, 1);
4162     }
4163
4164     inline Matrix2x2::Matrix2x2(const Matrix4x4& m)
4165     {
4166         //1st row
4167         mMat[0][0] = m(0, 0);
4168         mMat[0][1] = m(0, 1);
4169
4170         //2nd row
4171         mMat[1][0] = m(1, 0);
4172         mMat[1][1] = m(1, 1);
4173     }
4174
4175     inline Matrix2x2& Matrix2x2::operator=(const Matrix3x3& m)
4176     {
4177         //1st row
4178         mMat[0][0] = m(0, 0);
4179         mMat[0][1] = m(0, 1);
4180
4181         //2nd row
4182         mMat[1][0] = m(1, 0);
4183         mMat[1][1] = m(1, 1);
4184
4185         return *this;
4186     }
4187
4188     inline Matrix2x2& Matrix2x2::operator=(const Matrix4x4& m)
4189     {
4190         //1st row
4191         mMat[0][0] = m(0, 0);
4192         mMat[0][1] = m(0, 1);
4193
4194         //2nd row
4195         mMat[1][0] = m(1, 0);
4196         mMat[1][1] = m(1, 1);
4197
4198         return *this;
4199     }
4200
4201     inline Matrix3x3::Matrix3x3(const Matrix2x2& m)
4202     {
4203         //1st row
4204         mMat[0][0] = m(0, 0);
4205         mMat[0][1] = m(0, 1);
4206         mMat[0][2] = 0.0f;
4207
4208         //2nd row
4209         mMat[1][0] = m(1, 0);
4210         mMat[1][1] = m(1, 1);
4211         mMat[1][2] = 0.0f;
4212
4213         //3rd row
4214         mMat[2][0] = 0.0f;
4215         mMat[2][1] = 0.0f;
4216         mMat[2][2] = 1.0f;
4217     }
4218
4219     inline Matrix3x3::Matrix3x3(const Matrix4x4& m)
4220     {
4221         //1st row
4222         mMat[0][0] = m(0, 0);
4223         mMat[0][1] = m(0, 1);
4224         mMat[0][2] = m(0, 2);
4225
4226         //2nd row
4227         mMat[1][0] = m(1, 0);
4228         mMat[1][1] = m(1, 1);
4229         mMat[1][2] = m(1, 2);
4230
4231         //3rd row
4232         mMat[2][0] = m(2, 0);
4233         mMat[2][1] = m(2, 1);
4234         mMat[2][2] = m(2, 2);
4235     }
4236
4237     inline Matrix3x3& Matrix3x3::operator=(const Matrix2x2& m)
4238     {
4239         //1st row
4240         mMat[0][0] = m(0, 0);
4241         mMat[0][1] = m(0, 1);
4242         mMat[0][2] = 0.0f;
4243
4244         //2nd row
4245         mMat[1][0] = m(1, 0);
4246         mMat[1][1] = m(1, 1);
4247         mMat[1][2] = 0.0f;

```

```

4248
4249     //3rd row
4250     mMat[2][0] = 0.0f;
4251     mMat[2][1] = 0.0f;
4252     mMat[2][2] = 1.0f;
4253
4254     return *this;
4255 }
4256
4257 inline Matrix3x3& Matrix3x3::operator=(const Matrix4x4& m)
4258 {
4259     //1st row
4260     mMat[0][0] = m(0, 0);
4261     mMat[0][1] = m(0, 1);
4262     mMat[0][2] = m(0, 2);
4263
4264     //2nd row
4265     mMat[1][0] = m(1, 0);
4266     mMat[1][1] = m(1, 1);
4267     mMat[1][2] = m(1, 2);
4268
4269     //3rd row
4270     mMat[2][0] = m(2, 0);
4271     mMat[2][1] = m(2, 1);
4272     mMat[2][2] = m(2, 2);
4273
4274     return *this;
4275 }
4276
4277 inline Matrix4x4::Matrix4x4(const Matrix2x2& m)
4278 {
4279     //1st row
4280     mMat[0][0] = m(0, 0);
4281     mMat[0][1] = m(0, 1);
4282     mMat[0][2] = 0.0f;
4283     mMat[0][3] = 0.0f;
4284
4285     //2nd row
4286     mMat[1][0] = m(1, 0);
4287     mMat[1][1] = m(1, 1);
4288     mMat[1][2] = 0.0f;
4289     mMat[1][3] = 0.0f;
4290
4291     //3rd row
4292     mMat[2][0] = 0.0f;
4293     mMat[2][1] = 0.0f;
4294     mMat[2][2] = 1.0f;
4295     mMat[2][3] = 0.0f;
4296
4297     //4th row
4298     mMat[3][0] = 0.0f;
4299     mMat[3][1] = 0.0f;
4300     mMat[3][2] = 0.0f;
4301     mMat[3][3] = 1.0f;
4302 }
4303
4304 inline Matrix4x4::Matrix4x4(const Matrix3x3& m)
4305 {
4306     //1st row
4307     mMat[0][0] = m(0, 0);
4308     mMat[0][1] = m(0, 1);
4309     mMat[0][2] = m(0, 2);
4310     mMat[0][3] = 0.0f;
4311
4312     //2nd row
4313     mMat[1][0] = m(1, 0);
4314     mMat[1][1] = m(1, 1);
4315     mMat[1][2] = m(1, 2);
4316     mMat[1][3] = 0.0f;
4317
4318     //3rd row
4319     mMat[2][0] = m(2, 0);
4320     mMat[2][1] = m(2, 1);
4321     mMat[2][2] = m(2, 2);
4322     mMat[2][3] = 0.0f;
4323
4324     //4th row
4325     mMat[3][0] = 0.0f;
4326     mMat[3][1] = 0.0f;
4327     mMat[3][2] = 0.0f;
4328     mMat[3][3] = 1.0f;
4329 }
4330
4331 inline Matrix4x4& Matrix4x4::operator=(const Matrix2x2& m)
4332 {
4333     //1st row
4334     mMat[0][0] = m(0, 0);

```

```
4335         mMat[0][1] = m(0, 1);
4336         mMat[0][2] = 0.0f;
4337         mMat[0][3] = 0.0f;
4338
4339         //2nd row
4340         mMat[1][0] = m(1, 0);
4341         mMat[1][1] = m(1, 1);
4342         mMat[1][2] = 0.0f;
4343         mMat[1][3] = 0.0f;
4344
4345         //3rd row
4346         mMat[2][0] = 0.0f;
4347         mMat[2][1] = 0.0f;
4348         mMat[2][2] = 1.0f;
4349         mMat[2][3] = 0.0f;
4350
4351         //4th row
4352         mMat[3][0] = 0.0f;
4353         mMat[3][1] = 0.0f;
4354         mMat[3][2] = 0.0f;
4355         mMat[3][3] = 1.0f;
4356
4357         return *this;
4358     }
4359
4360     inline Matrix4x4& Matrix4x4::operator=(const Matrix3x3& m)
4361     {
4362         //1st row
4363         mMat[0][0] = m(0, 0);
4364         mMat[0][1] = m(0, 1);
4365         mMat[0][2] = m(0, 2);
4366         mMat[0][3] = 0.0f;
4367
4368         //2nd row
4369         mMat[1][0] = m(1, 0);
4370         mMat[1][1] = m(1, 1);
4371         mMat[1][2] = m(1, 2);
4372         mMat[1][3] = 0.0f;
4373
4374         //3rd row
4375         mMat[2][0] = m(2, 0);
4376         mMat[2][1] = m(2, 1);
4377         mMat[2][2] = m(2, 2);
4378         mMat[2][3] = 0.0f;
4379
4380         //4th row
4381         mMat[3][0] = 0.0f;
4382         mMat[3][1] = 0.0f;
4383         mMat[3][2] = 0.0f;
4384         mMat[3][3] = 1.0f;
4385
4386         return *this;
4387     }
4388
4389     //-----
4390 }
```


Index

- Adjoint
 - FAMath, [13](#)
- C:/Users/Work/Desktop/First Game Engine/First-Game-Engine/FA Math Engine/Header Files/FAMathEngine.h, [73](#)
- CartesianToCylindrical
 - FAMath, [13](#)
- CartesianToPolar
 - FAMath, [14](#)
- CartesianToSpherical
 - FAMath, [14](#)
- Cofactor
 - FAMath, [14](#)
- CompareDoubles
 - FAMath, [15](#)
- CompareFloats
 - FAMath, [15](#)
- Conjugate
 - FAMath, [15](#)
- CrossProduct
 - FAMath, [15](#)
- CylindricalToCartesian
 - FAMath, [15](#)
- Data
 - FAMath::Matrix2x2, [43](#)
 - FAMath::Matrix3x3, [48](#)
 - FAMath::Matrix4x4, [53](#)
- Determinant
 - FAMath, [16](#)
- DotProduct
 - FAMath, [16, 17](#)
- FAMath, [7](#)
 - Adjoint, [13](#)
 - CartesianToCylindrical, [13](#)
 - CartesianToPolar, [14](#)
 - CartesianToSpherical, [14](#)
 - Cofactor, [14](#)
 - CompareDoubles, [15](#)
 - CompareFloats, [15](#)
 - Conjugate, [15](#)
 - CrossProduct, [15](#)
 - CylindricalToCartesian, [15](#)
 - Determinant, [16](#)
 - DotProduct, [16, 17](#)
 - Inverse, [17, 18](#)
 - IsIdentity, [18](#)
 - IsZeroQuaternion, [18](#)
 - Length, [19](#)
 - Lerp, [19, 20](#)
 - NLerp, [20](#)
 - Norm, [20, 21](#)
 - Normalize, [21](#)
 - operator!=, [21, 22](#)
 - operator*, [22–27](#)
 - operator+, [27, 28](#)
 - operator-, [28–31](#)
 - operator/, [31](#)
 - operator==, [32](#)
 - Orthonormalize, [32, 33](#)
 - PolarToCartesian, [33](#)
 - Projection, [33](#)
 - QuaternionToRotationMatrixCol, [34](#)
 - QuaternionToRotationMatrixRow, [34](#)
 - Rotate, [34, 35](#)
 - RotationQuaternion, [36](#)
 - Scale, [36, 37](#)
 - SetToIdentity, [37, 38](#)
 - Slerp, [38](#)
 - SphericalToCartesian, [38](#)
 - Translate, [38, 39](#)
 - Transpose, [39](#)
 - ZeroVector, [39, 40](#)
- FAMath::Matrix2x2, [41](#)
 - Data, [43](#)
 - GetCol, [43](#)
 - GetRow, [43](#)
 - Matrix2x2, [42, 43](#)
 - operator*=, [44](#)
 - operator(), [44](#)
 - operator+=, [44](#)
 - operator-=, [45](#)
 - operator=, [45](#)
 - SetCol, [45](#)
 - SetRow, [45](#)
- FAMath::Matrix3x3, [46](#)
 - Data, [48](#)
 - GetCol, [48](#)
 - GetRow, [48](#)
 - Matrix3x3, [47, 48](#)
 - operator*=, [49](#)
 - operator(), [49](#)
 - operator+=, [49](#)
 - operator-=, [50](#)
 - operator=, [50](#)
 - SetCol, [50](#)
 - SetRow, [50](#)

- FAMath::Matrix4x4, 51
 - Data, 53
 - GetCol, 53
 - GetRow, 53
 - Matrix4x4, 52, 53
 - operator*=, 54
 - operator(), 54
 - operator+=, 54
 - operator-=, 55
 - operator=, 55
 - SetCol, 55
 - SetRow, 55
- FAMath::Quaternion, 56
 - GetScalar, 57
 - GetVector, 58
 - GetX, 58
 - GetY, 58
 - GetZ, 58
 - operator*=, 58
 - operator+=, 59
 - operator-=, 59
 - Quaternion, 57
 - SetScalar, 59
 - SetVector, 59
 - SetX, 59
 - SetY, 59
 - SetZ, 60
- FAMath::Vector2D, 60
 - GetX, 61
 - GetY, 61
 - operator*=, 62
 - operator+=, 62
 - operator-=, 62
 - operator/=: 62
 - operator=, 62
 - SetX, 63
 - SetY, 63
 - Vector2D, 61
- FAMath::Vector3D, 63
 - GetX, 65
 - GetY, 65
 - GetZ, 65
 - operator*=, 66
 - operator+=, 66
 - operator-=, 66
 - operator/=: 66
 - operator=, 66
 - SetX, 67
 - SetY, 67
 - SetZ, 67
 - Vector3D, 64, 65
- FAMath::Vector4D, 67
 - GetW, 69
 - GetX, 69
 - GetY, 69
 - GetZ, 70
 - operator*=, 70
 - operator+=, 70
 - operator-=, 70
 - operator/=: 70
 - operator=, 70, 71
 - SetW, 71
 - SetX, 71
 - SetY, 71
 - SetZ, 71
 - Vector4D, 68, 69
- GetCol
 - FAMath::Matrix2x2, 43
 - FAMath::Matrix3x3, 48
 - FAMath::Matrix4x4, 53
- GetRow
 - FAMath::Matrix2x2, 43
 - FAMath::Matrix3x3, 48
 - FAMath::Matrix4x4, 53
- GetScalar
 - FAMath::Quaternion, 57
- GetVector
 - FAMath::Quaternion, 58
- GetW
 - FAMath::Vector4D, 69
- GetX
 - FAMath::Quaternion, 58
 - FAMath::Vector2D, 61
 - FAMath::Vector3D, 65
 - FAMath::Vector4D, 69
- GetY
 - FAMath::Quaternion, 58
 - FAMath::Vector2D, 61
 - FAMath::Vector3D, 65
 - FAMath::Vector4D, 69
- GetZ
 - FAMath::Quaternion, 58
 - FAMath::Vector3D, 65
 - FAMath::Vector4D, 70
- Inverse
 - FAMath, 17, 18
- IsIdentity
 - FAMath, 18
- IsZeroQuaternion
 - FAMath, 18
- Length
 - FAMath, 19
- Lerp
 - FAMath, 19, 20
- Matrix2x2
 - FAMath::Matrix2x2, 42, 43
- Matrix3x3
 - FAMath::Matrix3x3, 47, 48
- Matrix4x4
 - FAMath::Matrix4x4, 52, 53
- NLerp
 - FAMath, 20

- Norm
 - FAMath, [20](#), [21](#)
- Normalize
 - FAMath, [21](#)
- operator!=
 - FAMath, [21](#), [22](#)
- operator*
 - FAMath, [22–27](#)
- operator*=
 - FAMath::Matrix2x2, [44](#)
 - FAMath::Matrix3x3, [49](#)
 - FAMath::Matrix4x4, [54](#)
 - FAMath::Quaternion, [58](#)
 - FAMath::Vector2D, [62](#)
 - FAMath::Vector3D, [66](#)
 - FAMath::Vector4D, [70](#)
- operator()
 - FAMath::Matrix2x2, [44](#)
 - FAMath::Matrix3x3, [49](#)
 - FAMath::Matrix4x4, [54](#)
- operator+
 - FAMath, [27](#), [28](#)
- operator+=
 - FAMath::Matrix2x2, [44](#)
 - FAMath::Matrix3x3, [49](#)
 - FAMath::Matrix4x4, [54](#)
 - FAMath::Quaternion, [59](#)
 - FAMath::Vector2D, [62](#)
 - FAMath::Vector3D, [66](#)
 - FAMath::Vector4D, [70](#)
- operator-
 - FAMath, [28–31](#)
- operator-=
 - FAMath::Matrix2x2, [45](#)
 - FAMath::Matrix3x3, [50](#)
 - FAMath::Matrix4x4, [55](#)
 - FAMath::Quaternion, [59](#)
 - FAMath::Vector2D, [62](#)
 - FAMath::Vector3D, [66](#)
 - FAMath::Vector4D, [70](#)
- operator/
 - FAMath, [31](#)
- operator/=
 - FAMath::Vector2D, [62](#)
 - FAMath::Vector3D, [66](#)
 - FAMath::Vector4D, [70](#)
- operator=
 - FAMath::Matrix2x2, [45](#)
 - FAMath::Matrix3x3, [50](#)
 - FAMath::Matrix4x4, [55](#)
 - FAMath::Vector2D, [62](#)
 - FAMath::Vector3D, [66](#)
 - FAMath::Vector4D, [70](#), [71](#)
- operator==
 - FAMath, [32](#)
- Orthonormalize
 - FAMath, [32](#), [33](#)
- PolarToCartesian
 - FAMath, [33](#)
- Projection
 - FAMath, [33](#)
- Quaternion
 - FAMath::Quaternion, [57](#)
- QuaternionToRotationMatrixCol
 - FAMath, [34](#)
- QuaternionToRotationMatrixRow
 - FAMath, [34](#)
- Rotate
 - FAMath, [34](#), [35](#)
- RotationQuaternion
 - FAMath, [36](#)
- Scale
 - FAMath, [36](#), [37](#)
- SetCol
 - FAMath::Matrix2x2, [45](#)
 - FAMath::Matrix3x3, [50](#)
 - FAMath::Matrix4x4, [55](#)
- SetRow
 - FAMath::Matrix2x2, [45](#)
 - FAMath::Matrix3x3, [50](#)
 - FAMath::Matrix4x4, [55](#)
- SetScalar
 - FAMath::Quaternion, [59](#)
- SetToIdentity
 - FAMath, [37](#), [38](#)
- SetVector
 - FAMath::Quaternion, [59](#)
- SetW
 - FAMath::Vector4D, [71](#)
- SetX
 - FAMath::Quaternion, [59](#)
 - FAMath::Vector2D, [63](#)
 - FAMath::Vector3D, [67](#)
 - FAMath::Vector4D, [71](#)
- SetY
 - FAMath::Quaternion, [59](#)
 - FAMath::Vector2D, [63](#)
 - FAMath::Vector3D, [67](#)
 - FAMath::Vector4D, [71](#)
- SetZ
 - FAMath::Quaternion, [60](#)
 - FAMath::Vector3D, [67](#)
 - FAMath::Vector4D, [71](#)
- Slerp
 - FAMath, [38](#)
- SphericalToCartesian
 - FAMath, [38](#)
- Translate
 - FAMath, [38](#), [39](#)
- Transpose
 - FAMath, [39](#)
- Vector2D

FAMath::Vector2D, [61](#)
Vector3D
FAMath::Vector3D, [64](#), [65](#)
Vector4D
FAMath::Vector4D, [68](#), [69](#)

ZeroVector
FAMath, [39](#), [40](#)