

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 MathEngine Namespace Reference	7
4.1.1 Function Documentation	13
4.1.1.1 Adjoint() [1/3]	13
4.1.1.2 Adjoint() [2/3]	13
4.1.1.3 Adjoint() [3/3]	13
4.1.1.4 Clamp()	13
4.1.1.5 Cofactor() [1/3]	14
4.1.1.6 Cofactor() [2/3]	14
4.1.1.7 Cofactor() [3/3]	14
4.1.1.8 CompareDoubles()	14
4.1.1.9 CompareFloats()	14
4.1.1.10 Conjugate()	15
4.1.1.11 CrossProduct()	15
4.1.1.12 Determinant() [1/3]	15
4.1.1.13 Determinant() [2/3]	15
4.1.1.14 Determinant() [3/3]	15
4.1.1.15 DotProduct() [1/4]	16
4.1.1.16 DotProduct() [2/4]	16
4.1.1.17 DotProduct() [3/4]	16
4.1.1.18 DotProduct() [4/4]	16
4.1.1.19 Identity() [1/4]	16
4.1.1.20 Identity() [2/4]	17
4.1.1.21 Identity() [3/4]	17
4.1.1.22 Identity() [4/4]	17
4.1.1.23 Inverse() [1/4]	17
4.1.1.24 Inverse() [2/4]	17
4.1.1.25 Inverse() [3/4]	17
4.1.1.26 Inverse() [4/4]	18
4.1.1.27 Length() [1/4]	18
4.1.1.28 Length() [2/4]	18
4.1.1.29 Length() [3/4]	18
4.1.1.30 Length() [4/4]	18
4.1.1.31 Lerp() [1/4]	19

4.1.1.32 Lerp() [2/4]	19
4.1.1.33 Lerp() [3/4]	19
4.1.1.34 Lerp() [4/4]	19
4.1.1.35 NLerp()	19
4.1.1.36 Normalize() [1/4]	20
4.1.1.37 Normalize() [2/4]	20
4.1.1.38 Normalize() [3/4]	20
4.1.1.39 Normalize() [4/4]	20
4.1.1.40 operator!=() [1/4]	20
4.1.1.41 operator!=() [2/4]	21
4.1.1.42 operator!=() [3/4]	21
4.1.1.43 operator!=() [4/4]	21
4.1.1.44 operator*() [1/24]	21
4.1.1.45 operator*() [2/24]	21
4.1.1.46 operator*() [3/24]	22
4.1.1.47 operator*() [4/24]	22
4.1.1.48 operator*() [5/24]	22
4.1.1.49 operator*() [6/24]	22
4.1.1.50 operator*() [7/24]	22
4.1.1.51 operator*() [8/24]	23
4.1.1.52 operator*() [9/24]	23
4.1.1.53 operator*() [10/24]	23
4.1.1.54 operator*() [11/24]	23
4.1.1.55 operator*() [12/24]	23
4.1.1.56 operator*() [13/24]	24
4.1.1.57 operator*() [14/24]	24
4.1.1.58 operator*() [15/24]	24
4.1.1.59 operator*() [16/24]	24
4.1.1.60 operator*() [17/24]	24
4.1.1.61 operator*() [18/24]	25
4.1.1.62 operator*() [19/24]	25
4.1.1.63 operator*() [20/24]	25
4.1.1.64 operator*() [21/24]	25
4.1.1.65 operator*() [22/24]	25
4.1.1.66 operator*() [23/24]	26
4.1.1.67 operator*() [24/24]	26
4.1.1.68 operator*=() [1/5]	26
4.1.1.69 operator*=() [2/5]	26
4.1.1.70 operator*=() [3/5]	26
4.1.1.71 operator*=() [4/5]	27
4.1.1.72 operator*=() [5/5]	27
4.1.1.73 operator+() [1/7]	27

4.1.1.74 operator+() [2/7]	27
4.1.1.75 operator+() [3/7]	27
4.1.1.76 operator+() [4/7]	28
4.1.1.77 operator+() [5/7]	28
4.1.1.78 operator+() [6/7]	28
4.1.1.79 operator+() [7/7]	28
4.1.1.80 operator+=() [1/4]	28
4.1.1.81 operator+=() [2/4]	29
4.1.1.82 operator+=() [3/4]	29
4.1.1.83 operator+=() [4/4]	29
4.1.1.84 operator-() [1/14]	29
4.1.1.85 operator-() [2/14]	29
4.1.1.86 operator-() [3/14]	30
4.1.1.87 operator-() [4/14]	30
4.1.1.88 operator-() [5/14]	30
4.1.1.89 operator-() [6/14]	30
4.1.1.90 operator-() [7/14]	30
4.1.1.91 operator-() [8/14]	31
4.1.1.92 operator-() [9/14]	31
4.1.1.93 operator-() [10/14]	31
4.1.1.94 operator-() [11/14]	31
4.1.1.95 operator-() [12/14]	31
4.1.1.96 operator-() [13/14]	32
4.1.1.97 operator-() [14/14]	32
4.1.1.98 operator-=() [1/4]	32
4.1.1.99 operator-=() [2/4]	32
4.1.1.100 operator-=() [3/4]	32
4.1.1.101 operator-=() [4/4]	33
4.1.1.102 operator==() [1/4]	33
4.1.1.103 operator==() [2/4]	33
4.1.1.104 operator==() [3/4]	33
4.1.1.105 operator==() [4/4]	33
4.1.1.106 Orthonormalize()	34
4.1.1.107 QuaternionToRotationMatrixCol3x3()	34
4.1.1.108 QuaternionToRotationMatrixCol4x4()	34
4.1.1.109 QuaternionToRotationMatrixRow3x3()	34
4.1.1.110 QuaternionToRotationMatrixRow4x4()	34
4.1.1.111 Rotate() [1/5]	35
4.1.1.112 Rotate() [2/5]	35
4.1.1.113 Rotate() [3/5]	35
4.1.1.114 Rotate() [4/5]	35
4.1.1.115 Rotate() [5/5]	35

4.1.1.116 Rotate4x4() [1/2]	36
4.1.1.117 Rotate4x4() [2/2]	36
4.1.1.118 RotationQuaternion() [1/3]	36
4.1.1.119 RotationQuaternion() [2/3]	36
4.1.1.120 RotationQuaternion() [3/3]	36
4.1.1.121 Scale() [1/4]	37
4.1.1.122 Scale() [2/4]	37
4.1.1.123 Scale() [3/4]	37
4.1.1.124 Scale() [4/4]	37
4.1.1.125 Scale4x4() [1/2]	37
4.1.1.126 Scale4x4() [2/2]	37
4.1.1.127 SetToIdentity() [1/3]	38
4.1.1.128 SetToIdentity() [2/3]	38
4.1.1.129 SetToIdentity() [3/3]	38
4.1.1.130 Slerp()	38
4.1.1.131 Translate() [1/2]	38
4.1.1.132 Translate() [2/2]	39
4.1.1.133 Transpose() [1/3]	39
4.1.1.134 Transpose() [2/3]	39
4.1.1.135 Transpose() [3/3]	39
4.1.1.136 ZeroQuaternion()	39
4.1.1.137 ZeroVector() [1/3]	40
4.1.1.138 ZeroVector() [2/3]	40
4.1.1.139 ZeroVector() [3/3]	40
5 Class Documentation	41
5.1 MathEngine::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 MathEngine::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 MathEngine::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	MathEngine::Quaternion Struct Reference	56
5.4.1	Detailed Description	56
5.4.2	Member Data Documentation	56
5.4.2.1	scalar	56
5.4.2.2	vector	57
5.5	MathEngine::Vector2D Struct Reference	57
5.5.1	Detailed Description	57
5.5.2	Member Data Documentation	57
5.5.2.1	x	57
5.5.2.2	y	57
5.6	MathEngine::Vector3D Struct Reference	58
5.6.1	Detailed Description	58
5.6.2	Member Data Documentation	58
5.6.2.1	x	58
5.6.2.2	y	58
5.6.2.3	z	58
5.7	MathEngine::Vector4D Struct Reference	59
5.7.1	Detailed Description	59
5.7.2	Member Data Documentation	59
5.7.2.1	w	59
5.7.2.2	x	59
5.7.2.3	y	59
5.7.2.4	z	59
6	File Documentation	61
6.1	MathEngine.h File Reference	61
6.1.1	Macro Definition Documentation	67
6.1.1.1	EPSILON	67
6.1.1.2	PI	67
6.1.1.3	PI2	68
6.1.2	Typedef Documentation	68

6.1.2.1 mat2	68
6.1.2.2 mat3	68
6.1.2.3 mat4	68
6.1.2.4 quaternion	68
6.1.2.5 vec2	68
6.1.2.6 vec3	68
6.1.2.7 vec4	69
6.2 MathEngine.h	69
Index	103

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

MathEngine	7
--------------------------------------	---

Chapter 2

Class Index

2.1 Class List

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

MathEngine::Matrix2x2	
A matrix class used for 2x2 matrices and their manipulations	41
MathEngine::Matrix3x3	
A matrix class used for 3x3 matrices and their manipulations	46
MathEngine::Matrix4x4	
A matrix class used for 4x4 matrices and their manipulations	51
MathEngine::Quaternion	
A quaternion struct used for quaternions and their manipulations	56
MathEngine::Vector2D	
A vector stuct used for 2D vectors/points	57
MathEngine::Vector3D	
A vector stuct used for 3D vectors/points	58
MathEngine::Vector4D	
A vector stuct used for 4D vectors/points	59

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

MathEngine.h	61
--	----

Chapter 4

Namespace Documentation

4.1 MathEngine Namespace Reference

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.
- struct [Quaternion](#)
A quaternion struct used for quaternions and their manipulations.
- struct [Vector2D](#)
A vector struct used for 2D vectors/points.
- struct [Vector3D](#)
A vector struct used for 3D vectors/points.
- struct [Vector4D](#)
A vector struct used for 4D vectors/points.

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.
- float [Clamp](#) (float value, float a, float b)
Returns a clamped value.
- void [operator+=](#) ([Vector2D](#) &v1, const [Vector2D](#) &v2)
Adds the 2D vector v1 to the 2D vector v2 and stores the result in v1.
- void [operator-=](#) ([Vector2D](#) &v1, const [Vector2D](#) &v2)
Subtracts the 2D vector v2 from the 2D vector v1 and stores the result in v1.
- void [operator*=](#) ([Vector2D](#) &v, float k)
Multiplies the 2D vector v by the scalar (float) k and stores the result in v.
- [Vector2D operator+](#) (const [Vector2D](#) &v1, const [Vector2D](#) &v2)

- Adds the two 2D vectors and returns the result.*

 - [Vector2D operator-](#) (const [Vector2D](#) &v)
- Negates the 2D vector v1 and returns the result.*

 - [Vector2D operator-](#) (const [Vector2D](#) &v1, const [Vector2D](#) &v2)
- Subtracts the 2D vector v2 from the 2D vector v1 and returns the result.*

 - [Vector2D operator*](#) (const [Vector2D](#) &v, float k)
- Multiplies the 2D vector v by the scalar (float) k and returns the result.*

 - [Vector2D operator*](#) (float k, const [Vector2D](#) &v)
- Multiplies the scalar (float) k by the 2D vector v and returns the result.*

 - [bool operator==](#) (const [Vector2D](#) &v1, const [Vector2D](#) &v2)
- Returns true if the 2D vector v1 equals to the 2D vector v2, false otherwise.*

 - [bool operator!=](#) (const [Vector2D](#) &v1, const [Vector2D](#) &v2)
- Returns true if the 2D vector v1 does not equal to the 2D vector v2, false otherwise.*

 - [bool ZeroVector](#) (const [Vector2D](#) &v)
- Returns true if the 2D vector v is equal to the zero vector, false otherwise.*

 - [float DotProduct](#) (const [Vector2D](#) &v1, const [Vector2D](#) &v2)
- Returns the dot product between the 2D vectors v1 and v2.*

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

 - [Vector2D Normalize](#) (const [Vector2D](#) &v)
- Normalizes (makes it unit length) the 2D vector v and returns the result.*

 - [Vector2D Lerp](#) (const [Vector2D](#) &start, const [Vector2D](#) &end, float t)
- Linear interpolate between the two vectors start and end.*

 - [void operator+=](#) ([Vector3D](#) &v1, const [Vector3D](#) &v2)
- Adds the 3D vector v1 to the 3D vector v2 and stores the result in v1.*

 - [void operator-=](#) ([Vector3D](#) &v1, const [Vector3D](#) &v2)
- Subtracts the 3D vector v2 from the 3D vector v1 and stores the result in v1.*

 - [void operator*=](#) ([Vector3D](#) &v, float k)
- Multiplies the 3D vector v by the scalar (float) k and stores the result in v.*

 - [Vector3D operator+](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Adds the two 3D vectors and returns the result.*

 - [Vector3D operator-](#) (const [Vector3D](#) &v)
- Negates the 3D vector v and returns the result.*

 - [Vector3D operator-](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Subtracts the 3D vector v2 from the 3D vector v1 and returns the result.*

 - [Vector3D operator*](#) (const [Vector3D](#) &v, float k)
- Multiplies the 3D vector v by the scalar (float) k and returns the result.*

 - [Vector3D operator*](#) (float k, const [Vector3D](#) &v)
- Multiplies the scalar (float) k by the 3D vector v and returns the result.*

 - [bool operator==](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Returns true if the 3D vector v1 equals to the 3D vector v2, false otherwise.*

 - [bool operator!=](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Returns true if the 3D vector v1 does not equal to the 3D vector v2, false otherwise.*

 - [bool ZeroVector](#) (const [Vector3D](#) &v)
- Returns true if the 3D vector v is equal to the zero vector, false otherwise.*

 - [float DotProduct](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Returns the dot product between the 3D vectors v1 and v2.*

 - [Vector3D CrossProduct](#) (const [Vector3D](#) &v1, const [Vector3D](#) &v2)
- Returns the cross product between the 3D vectors v1 and v2.*

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

- [Vector3D Normalize](#) (const [Vector3D](#) &v)
Normalizes (makes it unit length) the 3D vector v and returns the result.
- 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.
- void [operator+=](#) ([Vector4D](#) &v1, const [Vector4D](#) &v2)
Adds the 4D vector v1 to the 4D vector v2 and stores the result in v1.
- void [operator-=](#) ([Vector4D](#) &v1, const [Vector4D](#) &v2)
Subtracts the 4D vector v2 from the 4D vector v1 and stores the result in v1.
- void [operator*=](#) ([Vector4D](#) &v, float k)
Multiplies the 4D vector v by the scalar (float) k and stores the result in v.
- [Vector4D operator+](#) (const [Vector4D](#) &v1, const [Vector4D](#) &v2)
Adds the two 4D vectors and returns the result.
- [Vector4D operator-](#) (const [Vector4D](#) &v)
Negates the 4D vector v and returns the result.
- [Vector4D operator-](#) (const [Vector4D](#) &v1, const [Vector4D](#) &v2)
Subtracts the 4D vector v2 from the 4D vector v1 and returns the result.
- [Vector4D operator*](#) (const [Vector4D](#) &v, float k)
Multiplies the 4D vector v by the scalar (float) k and returns the result.
- [Vector4D operator*](#) (float k, const [Vector4D](#) &v)
Multiplies the scalar (float) k by the 4D vector v and returns the result.
- bool [operator==](#) (const [Vector4D](#) &v1, const [Vector4D](#) &v2)
Returns true if the 4D vector v1 equals to the 4D vector v2, false otherwise.
- bool [operator!=](#) (const [Vector4D](#) &v1, const [Vector4D](#) &v2)
Returns true if the 4D vector v1 does not equal to the 4D vector v2, false otherwise.
- bool [ZeroVector](#) (const [Vector4D](#) &v)
Returns true if the 4D vector v is equal to the zero vector, false otherwise.
- float [DotProduct](#) (const [Vector4D](#) &v1, const [Vector4D](#) &v2)
Returns the dot product between the 4D vectors v1 and v2.
- float [Length](#) (const [Vector4D](#) &v)
Returns the length (magnitude) of the 4D vector v.
- [Vector4D Normalize](#) (const [Vector4D](#) &v)
Normalizes (makes it unit length) the 4D vector v and returns the result.
- [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 [Identity](#) (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](#) (float x, float y)

Returns a 2x2 scaling matrix.
- [Matrix2x2 Scale](#) (const [Vector2D](#) &scaleVector)

Returns a 2x2 scaling matrix.
- [Matrix2x2 Rotate](#) (float angle)

Returns a 2x2 rotation matrix that rotates a point/vector about the origin.
- double [Determinant](#) (const [Matrix2x2](#) &m)

Returns the determinant of 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 $\backslash m$ and returns the result.
- [Matrix3x3 operator*](#) (const [Matrix3x3](#) &m1, const [Matrix3x3](#) &m2)

Multiplies $m1$ with $\backslash m2$ 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 [Identity](#) (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](#) (float x, float y, float z)
- [Matrix3x3 Scale](#) (const [Vector3D](#) &scaleVector)
- [Matrix3x3 Rotate](#) (float angle, float x, float y, float z)
- [Matrix3x3 Rotate](#) (float angle, const [Vector3D](#) &axis)
- double [Determinant](#) (const [Matrix3x3](#) &m)

Returns the determinant of 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 [Identity](#) (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](#) (float x, float y, float z)
Returns a 4x4 translation matrix.
- [Matrix4x4 Translate](#) (const [Vector3D](#) &translateVector)
Returns a 4x4 translation matrix.
- [Matrix4x4 Scale4x4](#) (float x, float y, float z)
Returns a 4x4 scale matrix.
- [Matrix4x4 Scale4x4](#) (const [Vector3D](#) &scaleVector)
Returns a 4x4 scale matrix.
- [Matrix4x4 Rotate4x4](#) (float angle, float x, float y, float z)
Returns a 4x4 rotation matrix about the given axis.
- [Matrix4x4 Rotate4x4](#) (float angle, const [Vector3D](#) &axis)
Returns a 4x4 rotation matrix about the given axis.
- 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.
- void [operator+=](#) ([Quaternion](#) &q1, const [Quaternion](#) &q2)
Adds the quaternion q1 to the quaternion q2 and stores the result in q1.
- void [operator-=](#) ([Quaternion](#) &q1, const [Quaternion](#) &q2)
Subtracts the quaternion q2 from the quaternion q1 and stores the result in q1.
- void [operator*=](#) ([Quaternion](#) &q1, float k)

- Multiplies the quaternion $q1$ by the scalar (float) k and stores the result in $q1$.*

 - void `operator*=` (Quaternion & $q1$, const Quaternion & $q2$)
- Multiplies the quaternion $q1$ by the quaternion $q1$ and stores the result in $q1$.*

 - Quaternion `operator+` (const Quaternion & $q1$, const Quaternion & $q2$)
- Adds the quaternion $q1$ to the quaternion $q2$ and returns the result.*

 - Quaternion `operator-` (const Quaternion & $q1$, const Quaternion & $q2$)
- Subtracts the quaternion $q2$ from the quaternion $q1$ and returns the result.*

 - Quaternion `operator-` (const Quaternion & q)
- Negates the quaternion $q1$ and returns the result.*

 - Quaternion `operator*` (const Quaternion & q , float k)
- Multiplies the quaternion q by the scalar (float) k and returns the result.*

 - Quaternion `operator*` (float k , const Quaternion & q)
- Multiplies the scalar (float) k by the quaternion q and returns the result.*

 - Quaternion `operator*` (const Quaternion & $q1$, const Quaternion & $q2$)
- Multiplies the quaternion $q1$ by the quaternion $q2$ and returns the result.*

 - bool `operator==` (const Quaternion & $q1$, const Quaternion & $q2$)
- Returns true if the quaternion $q1$ equals to the quaternion $q2$, false otherwise.*

 - bool `operator!=` (const Quaternion & $q1$, const Quaternion & $q2$)
- Returns true if the quaternion $q1$ does not equal to the quaternion $q2$, false otherwise.*

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

 - bool `Identity` (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 the quaternion q and returns the normalized quaternion.*

 - Quaternion `Inverse` (const Quaternion & q)
- Returns the invese of the quaternion q .*

 - Quaternion `RotationQuaternion` (float angle, float x , float y , float z)
- Returns a rotation quaternion from the axis-angle representation.*

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

 - Quaternion `RotationQuaternion` (const Vector4D &angAxis)
- Returns a quaternion from the axis-angle representation.*

 - 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 .*

 - Matrix3x3 `QuaternionToRotationMatrixCol3x3` (const Quaternion & q)
- Transforms q into a column-major matrix.*

 - Matrix3x3 `QuaternionToRotationMatrixRow3x3` (const Quaternion & q)
- Transforms q into a row-major matrix.*

 - Matrix4x4 `QuaternionToRotationMatrixCol4x4` (const Quaternion & q)
- Transforms q into a column-major matrix.*

 - Matrix4x4 `QuaternionToRotationMatrixRow4x4` (const Quaternion & q)
- Transforms q into a row-major matrix.*

 - float `DotProduct` (const Quaternion & $q1$, const Quaternion & $q2$)
- Returns the dot product of the quaternions $q1$ and $q2$.*

- [Quaternion Lerp](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2, float t)
Linear Interpolates between quaternions q1 and q2.
- [Quaternion NLerp](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2, float t)
Linear Interpolates between quaternions q1 and q2 and normalizes the result.
- [Quaternion Slerp](#) (const [Quaternion](#) &q1, const [Quaternion](#) &q2, float t)
Spherical Linear Interpolates between quaternions q1 and q2.

4.1.1 Function Documentation

4.1.1.1 Adjoint() [1/3]

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

Returns the adjoint of m .

4.1.1.2 Adjoint() [2/3]

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

Returns the adjoint of m .

4.1.1.3 Adjoint() [3/3]

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

Returns the adjoint of m .

4.1.1.4 Clamp()

```
float MathEngine::Clamp (
    float value,
    float a,
    float b ) [inline]
```

Returns a clamped value.

Returns a if value < a. Returns b if value > b. Returns value if it is between a and b.

4.1.1.5 Cofactor() [1/3]

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

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

4.1.1.6 Cofactor() [2/3]

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

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

4.1.1.7 Cofactor() [3/3]

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

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

4.1.1.8 CompareDoubles()

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

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

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

4.1.1.9 CompareFloats()

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

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

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

4.1.1.10 Conjugate()

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

Returns the conjugate of quaternion q .

4.1.1.11 CrossProduct()

```
Vector3D MathEngine::CrossProduct (
    const Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Returns the cross product between the 3D vectors $v1$ and $v2$.

4.1.1.12 Determinant() [1/3]

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

Returns the determinant of m .

4.1.1.13 Determinant() [2/3]

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

Returns the determinant of m .

4.1.1.14 Determinant() [3/3]

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

Returns the determinant m .

4.1.1.15 DotProduct() [1/4]

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

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

4.1.1.16 DotProduct() [2/4]

```
float MathEngine::DotProduct (
    const Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Returns the dot product between the 2D vectors *v1* and *v2*.

4.1.1.17 DotProduct() [3/4]

```
float MathEngine::DotProduct (
    const Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Returns the dot product between the 3D vectors *v1* and *v2*.

4.1.1.18 DotProduct() [4/4]

```
float MathEngine::DotProduct (
    const Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Returns the dot product between the 4D vectors *v1* and *v2*.

4.1.1.19 Identity() [1/4]

```
bool MathEngine::Identity (
    const Matrix2x2 & m ) [inline]
```

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

4.1.1.20 Identity() [2/4]

```
bool MathEngine::Identity (
    const Matrix3x3 & m ) [inline]
```

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

4.1.1.21 Identity() [3/4]

```
bool MathEngine::Identity (
    const Matrix4x4 & m ) [inline]
```

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

4.1.1.22 Identity() [4/4]

```
bool MathEngine::Identity (
    const Quaternion & q ) [inline]
```

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

4.1.1.23 Inverse() [1/4]

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

Returns the inverse of m .

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

4.1.1.24 Inverse() [2/4]

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

Returns the inverse of m .

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

4.1.1.25 Inverse() [3/4]

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

Returns the inverse of m .

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

4.1.1.26 Inverse() [4/4]

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

Returns the invese of the quaternion q .

If q is the zero quaternion then q is returned.

4.1.1.27 Length() [1/4]

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

Returns the length of quaternion q .

4.1.1.28 Length() [2/4]

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

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

4.1.1.29 Length() [3/4]

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

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

4.1.1.30 Length() [4/4]

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

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

4.1.1.31 Lerp() [1/4]

```
Quaternion MathEngine::Lerp (  
    const Quaternion & q1,  
    const Quaternion & q2,  
    float t ) [inline]
```

Linear Interpolates between quaternions *q1* and *q2*.

If *t* is not between 0 and 1, it gets clamped.

4.1.1.32 Lerp() [2/4]

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

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

If *t* is not between 0 and 1, it gets clamped.

4.1.1.33 Lerp() [3/4]

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

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

If *t* is not between 0 and 1, it gets clamped.

4.1.1.34 Lerp() [4/4]

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

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

If *t* is not between 0 and 1, it gets clamped.

4.1.1.35 NLerp()

```
Quaternion MathEngine::NLerp (  
    const Quaternion & q1,  
    const Quaternion & q2,  
    float t ) [inline]
```

Linear Interpolates between quaternions *q1* and *q2* and normalizes the result.

If *t* is not between 0 and 1, it gets clamped.

4.1.1.36 Normalize() [1/4]

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

Normalizes the quaternion q and returns the normalized quaternion.

If q is the zero quaternion, q is returned.

4.1.1.37 Normalize() [2/4]

```
Vector2D MathEngine::Normalize (
    const Vector2D & v ) [inline]
```

Normalizes (makes it unit length) the 2D vector v and returns the result.

If v is the zero vector, v is returned.

4.1.1.38 Normalize() [3/4]

```
Vector3D MathEngine::Normalize (
    const Vector3D & v ) [inline]
```

Normalizes (makes it unit length) the 3D vector v and returns the result.

If v is the zero vector, v is returned.

4.1.1.39 Normalize() [4/4]

```
Vector4D MathEngine::Normalize (
    const Vector4D & v ) [inline]
```

Normalizes (makes it unit length) the 4D vector v and returns the result.

If v is the zero vector, v is returned.

4.1.1.40 operator!=(=) [1/4]

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

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

4.1.1.41 operator!=(()) [2/4]

```
bool MathEngine::operator!=(  
    const Vector2D & v1,  
    const Vector2D & v2 ) [inline]
```

Returns true if the 2D vector *v1* does not equal to the 2D vector *v2*, false otherwise.

4.1.1.42 operator!=(()) [3/4]

```
bool MathEngine::operator!=(  
    const Vector3D & v1,  
    const Vector3D & v2 ) [inline]
```

Returns true if the 3D vector *v1* does not equal to the 3D vector *v2*, false otherwise.

4.1.1.43 operator!=(()) [4/4]

```
bool MathEngine::operator!=(  
    const Vector4D & v1,  
    const Vector4D & v2 ) [inline]
```

Returns true if the 4D vector *v1* does not equal to the 4D vector *v2*, false otherwise.

4.1.1.44 operator*() [1/24]

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

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

4.1.1.45 operator*() [2/24]

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

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

4.1.1.46 operator*() [3/24]

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

Multiplies k with m and returns the result.

4.1.1.47 operator*() [4/24]

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

Multiplies m with k and returns the result.

4.1.1.48 operator*() [5/24]

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

Multiplies m with v and returns the result.

The vector v is a column vector.

4.1.1.49 operator*() [6/24]

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

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

Does $m1 * m2$ in that order.

4.1.1.50 operator*() [7/24]

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

Multiplies m with k and returns the result.

4.1.1.51 operator*() [8/24]

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

Multiplies m with v and returns the result.

The vector v is a column vector.

4.1.1.52 operator*() [9/24]

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

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

Does $m1 * m2$ in that order.

4.1.1.53 operator*() [10/24]

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

Multiplies m with k and returns the result.

4.1.1.54 operator*() [11/24]

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

Multiplies m with v and returns the result.

The vector v is a column vector.

4.1.1.55 operator*() [12/24]

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

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

Does $m1 * m2$ in that order.

4.1.1.56 operator*() [13/24]

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

Multiplies the quaternion q by the scalar (float) k and returns the result.

4.1.1.57 operator*() [14/24]

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

Multiplies the quaternion $q1$ by the quaternion $q2$ and returns the result.

4.1.1.58 operator*() [15/24]

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

Multiplies v with m and returns the result.

The vector v is a row vector.

4.1.1.59 operator*() [16/24]

```
Vector2D MathEngine::operator* (
    const Vector2D & v,
    float k ) [inline]
```

Multiplies the 2D vector v by the scalar (float) k and returns the result.

4.1.1.60 operator*() [17/24]

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

Multiplies v with m and returns the result.

The vector v is a row vector.

4.1.1.61 operator*() [18/24]

```
Vector3D MathEngine::operator* (
    const Vector3D & v,
    float k ) [inline]
```

Multiplies the 3D vector v by the scalar (float) k and returns the result.

4.1.1.62 operator*() [19/24]

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

Multiplies v with m and returns the result.

The vector v is a row vector.

4.1.1.63 operator*() [20/24]

```
Vector4D MathEngine::operator* (
    const Vector4D & v,
    float k ) [inline]
```

Multiplies the 4D vector v by the scalar (float) k and returns the result.

4.1.1.64 operator*() [21/24]

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

Multiplies the scalar (float) k by the quaternion q and returns the result.

4.1.1.65 operator*() [22/24]

```
Vector2D MathEngine::operator* (
    float k,
    const Vector2D & v ) [inline]
```

Multiplies the scalar (float) k by the 2D vector v and returns the result.

4.1.1.66 operator*() [23/24]

```
Vector3D MathEngine::operator* (
    float k,
    const Vector3D & v ) [inline]
```

Multiplies the scalar (float) *k* by the 3D vector *v* and returns the result.

4.1.1.67 operator*() [24/24]

```
Vector4D MathEngine::operator* (
    float k,
    const Vector4D & v ) [inline]
```

Multiplies the scalar (float) *k* by the 4D vector *v* and returns the result.

4.1.1.68 operator*=() [1/5]

```
void MathEngine::operator*= (
    Quaternion & q1,
    const Quaternion & q2 ) [inline]
```

Multiplies the quaternion *q1* by the quaternion *q1* and stores the result in *q1*.

4.1.1.69 operator*=() [2/5]

```
void MathEngine::operator*= (
    Quaternion & q1,
    float k ) [inline]
```

Multiplies the quaternion *q1* by the scalar (float) *k* and stores the result in *q1*.

4.1.1.70 operator*=() [3/5]

```
void MathEngine::operator*= (
    Vector2D & v,
    float k ) [inline]
```

Multiplies the 2D vector *v* by the scalar (float) *k* and stores the result in *v*.

4.1.1.71 operator*=() [4/5]

```
void MathEngine::operator*= (
    Vector3D & v,
    float k ) [inline]
```

Multiplies the 3D vector *v* by the scalar (float) *k* and stores the result in *v*.

4.1.1.72 operator*=() [5/5]

```
void MathEngine::operator*= (
    Vector4D & v,
    float k ) [inline]
```

Multiplies the 4D vector *v* by the scalar (float) *k* and stores the result in *v*.

4.1.1.73 operator+() [1/7]

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

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

4.1.1.74 operator+() [2/7]

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

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

4.1.1.75 operator+() [3/7]

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

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

4.1.1.76 operator+() [4/7]

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

Adds the quaternion *q1* to the quaternion *q2* and returns the result.

4.1.1.77 operator+() [5/7]

```
Vector2D MathEngine::operator+ (
    const Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Adds the two 2D vectors and returns the result.

4.1.1.78 operator+() [6/7]

```
Vector3D MathEngine::operator+ (
    const Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Adds the two 3D vectors and returns the result.

4.1.1.79 operator+() [7/7]

```
Vector4D MathEngine::operator+ (
    const Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Adds the two 4D vectors and returns the result.

4.1.1.80 operator+=() [1/4]

```
void MathEngine::operator+= (
    Quaternion & q1,
    const Quaternion & q2 ) [inline]
```

Adds the quaternion *q1* to the quaternion *q2* and stores the result in *q1*.

4.1.1.81 operator+=() [2/4]

```
void MathEngine::operator+= (
    Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Adds the 2D vector *v1* to the 2D vector *v2* and stores the result in *v1*.

4.1.1.82 operator+=() [3/4]

```
void MathEngine::operator+= (
    Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Adds the 3D vector *v1* to the 3D vector *v2* and stores the result in *v1*.

4.1.1.83 operator+=() [4/4]

```
void MathEngine::operator+= (
    Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Adds the 4D vector *v1* to the 4D vector *v2* and stores the result in *v1*.

4.1.1.84 operator-() [1/14]

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

Negates the 2x2 matrix *m*.

4.1.1.85 operator-() [2/14]

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

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

4.1.1.86 operator-() [3/14]

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

Negates the 3x3 matrix *m*.

4.1.1.87 operator-() [4/14]

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

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

4.1.1.88 operator-() [5/14]

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

Negates the 4x4 matrix *m*.

4.1.1.89 operator-() [6/14]

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

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

4.1.1.90 operator-() [7/14]

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

Negates the quaternion *q1* and returns the result.

4.1.1.91 operator-() [8/14]

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

Subtracts the quaternion *q2* from the quaternion *q1* and returns the result.

Returns q1 - q2.

4.1.1.92 operator-() [9/14]

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

Negates the 3D vector *v1* and returns the result.

4.1.1.93 operator-() [10/14]

```
Vector2D MathEngine::operator- (
    const Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Subtracts the 2D vector *v2* from the 2D vector *v1* and returns the result.

Returns v1 - v2.

4.1.1.94 operator-() [11/14]

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

Negates the 3D vector *v* and returns the result.

4.1.1.95 operator-() [12/14]

```
Vector3D MathEngine::operator- (
    const Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Subtracts the 3D vector *v2* from the 3D vector *v1* and returns the result.

Returns v1 - v2.

4.1.1.96 operator-() [13/14]

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

Negates the 4D vector *v* and returns the result.

4.1.1.97 operator-() [14/14]

```
Vector4D MathEngine::operator- (
    const Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Subtracts the 4D vector *v2* from the 4D vector *v1* and returns the result.

Returns *v1* - *v2*.

4.1.1.98 operator-=() [1/4]

```
void MathEngine::operator-= (
    Quaternion & q1,
    const Quaternion & q2 ) [inline]
```

Subtracts the quaternion *q2* from the quaternion *q1* and stores the result in *q1*.

q1 - *q2*

4.1.1.99 operator-=() [2/4]

```
void MathEngine::operator-= (
    Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Subtracts the the 2D vector *v2* from the 2D vector *v1* and stores the result in *v1*.

v1 - *v2*.

4.1.1.100 operator-=() [3/4]

```
void MathEngine::operator-= (
    Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Subtracts the the 3D vector *v2* from the 3D vector *v1* and stores the result in *v1*.

v1 - *v2*.

4.1.1.101 operator-=() [4/4]

```
void MathEngine::operator-= (
    Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Subtracts the the 4D vector *v2* from the 4D vector *v1* and stores the result in *v1*.

v1 - *v2*.

4.1.1.102 operator==() [1/4]

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

Returns true if the quaternion *q1* equals to the quaternion *q2*, false otherwise.

4.1.1.103 operator==() [2/4]

```
bool MathEngine::operator==(
    const Vector2D & v1,
    const Vector2D & v2 ) [inline]
```

Returns true if the 2D vector *v1* equals to the 2D vector *v2*, false otherwise.

4.1.1.104 operator==() [3/4]

```
bool MathEngine::operator==(
    const Vector3D & v1,
    const Vector3D & v2 ) [inline]
```

Returns true if the 3D vector *v1* equals to the 3D vector *v2*, false otherwise.

4.1.1.105 operator==() [4/4]

```
bool MathEngine::operator==(
    const Vector4D & v1,
    const Vector4D & v2 ) [inline]
```

Returns true if the 4D vector *v1* equals to the 4D vector *v2*, false otherwise.

4.1.1.106 Orthonormalize()

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

Orthonormalizes the specified vectors.

4.1.1.107 QuaternionToRotationMatrixCol3x3()

```
Matrix3x3 MathEngine::QuaternionToRotationMatrixCol3x3 (
    const Quaternion & q ) [inline]
```

Transforms q into a column-major matrix.

q should be a rotation quaternion.

4.1.1.108 QuaternionToRotationMatrixCol4x4()

```
Matrix4x4 MathEngine::QuaternionToRotationMatrixCol4x4 (
    const Quaternion & q ) [inline]
```

Transforms q into a column-major matrix.

q should be a rotation quaternion.

4.1.1.109 QuaternionToRotationMatrixRow3x3()

```
Matrix3x3 MathEngine::QuaternionToRotationMatrixRow3x3 (
    const Quaternion & q ) [inline]
```

Transforms q into a row-major matrix.

q should be a unit quaternion.

4.1.1.110 QuaternionToRotationMatrixRow4x4()

```
Matrix4x4 MathEngine::QuaternionToRotationMatrixRow4x4 (
    const Quaternion & q ) [inline]
```

Transforms q into a row-major matrix.

q should be a unit quaternion.

4.1.1.111 Rotate() [1/5]

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

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

q should be a rotation quaternion.

4.1.1.112 Rotate() [2/5]

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

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

q should be a rotation quaternion.

4.1.1.113 Rotate() [3/5]

```
Matrix2x2 MathEngine::Rotate (
    float angle ) [inline]
```

Returns a 2x2 rotation matrix that rotates a point/vector about the origin.

4.1.1.114 Rotate() [4/5]

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

brief Returns a 3x3 rotation matrix about the given axis.

4.1.1.115 Rotate() [5/5]

```
Matrix3x3 MathEngine::Rotate (
    float angle,
    float x,
    float y,
    float z ) [inline]
```

brief Returns a 3x3 rotation matrix about the given axis.

4.1.1.116 Rotate4x4() [1/2]

```
Matrix4x4 MathEngine::Rotate4x4 (
    float angle,
    const Vector3D & axis ) [inline]
```

Returns a 4x4 rotation matrix about the given axis.

4.1.1.117 Rotate4x4() [2/2]

```
Matrix4x4 MathEngine::Rotate4x4 (
    float angle,
    float x,
    float y,
    float z ) [inline]
```

Returns a 4x4 rotation matrix about the given axis.

4.1.1.118 RotationQuaternion() [1/3]

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

Returns a quaternion from the axis-angle representation.

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

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

4.1.1.119 RotationQuaternion() [2/3]

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

Returns a quaternion from the axis-angle representation.

The *angle* should be given in degrees.

4.1.1.120 RotationQuaternion() [3/3]

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

Returns a rotation quaternion from the axis-angle representation.

The *angle* should be given in degrees.

4.1.1.121 Scale() [1/4]

```
Matrix2x2 MathEngine::Scale (
    const Vector2D & scaleVector ) [inline]
```

Returns a 2x2 scaling matrix.

4.1.1.122 Scale() [2/4]

```
Matrix3x3 MathEngine::Scale (
    const Vector3D & scaleVector ) [inline]
```

brief Returns a 3x3 scale matrix.

4.1.1.123 Scale() [3/4]

```
Matrix2x2 MathEngine::Scale (
    float x,
    float y ) [inline]
```

Returns a 2x2 scaling matrix.

4.1.1.124 Scale() [4/4]

```
Matrix3x3 MathEngine::Scale (
    float x,
    float y,
    float z ) [inline]
```

brief Returns a 3x3 scale matrix.

4.1.1.125 Scale4x4() [1/2]

```
Matrix4x4 MathEngine::Scale4x4 (
    const Vector3D & scaleVector ) [inline]
```

Returns a 4x4 scale matrix.

4.1.1.126 Scale4x4() [2/2]

```
Matrix4x4 MathEngine::Scale4x4 (
    float x,
    float y,
    float z ) [inline]
```

Returns a 4x4 scale matrix.

4.1.1.127 SetToIdentity() [1/3]

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

Sets m to the identity matrix.

4.1.1.128 SetToIdentity() [2/3]

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

Sets m to the identity matrix.

4.1.1.129 SetToIdentity() [3/3]

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

Sets m to the identity matrix.

4.1.1.130 Slerp()

```
Quaternion MathEngine::Slerp (
    const Quaternion & q1,
    const Quaternion & q2,
    float t ) [inline]
```

Spherical Linear Interpolates between quaternions $q1$ and $q2$.

If t is not between 0 and 1, it gets clamped.

4.1.1.131 Translate() [1/2]

```
Matrix4x4 MathEngine::Translate (
    const Vector3D & translateVector ) [inline]
```

Returns a 4x4 translation matrix.

4.1.1.132 Translate() [2/2]

```
Matrix4x4 MathEngine::Translate (
    float x,
    float y,
    float z ) [inline]
```

Returns a 4x4 translation matrix.

4.1.1.133 Transpose() [1/3]

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

Returns the tranpose of the given matrix *m*.

4.1.1.134 Transpose() [2/3]

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

Returns the tranpose of the given matrix *m*.

4.1.1.135 Transpose() [3/3]

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

Returns the tranpose of the given matrix *m*.

4.1.1.136 ZeroQuaternion()

```
bool MathEngine::ZeroQuaternion (
    const Quaternion & q ) [inline]
```

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

4.1.1.137 ZeroVector() [1/3]

```
bool MathEngine::ZeroVector (
    const Vector2D & v ) [inline]
```

Returns true if the 2D vector *v* is equal to the zero vector, false otherwise.

4.1.1.138 ZeroVector() [2/3]

```
bool MathEngine::ZeroVector (
    const Vector3D & v ) [inline]
```

Returns true if the 3D vector *v* is equal to the zero vector, false otherwise.

4.1.1.139 ZeroVector() [3/3]

```
bool MathEngine::ZeroVector (
    const Vector4D & v ) [inline]
```

Returns true if the 4D vector *v* is equal to the zero vector, false otherwise.

Chapter 5

Class Documentation

5.1 MathEngine::Matrix2x2 Class Reference

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

```
#include <MathEngine.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]

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

Creates a new 2x2 identity matrix.

5.1.2.2 `Matrix2x2()` [2/5]

```
MathEngine::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]

```
MathEngine::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]

```
MathEngine::Matrix2x2::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.

5.1.2.5 Matrix2x2() [5/5]

```
MathEngine::Matrix2x2::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.

5.1.3 Member Function Documentation

5.1.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.1.3.2 Data() [2/2]

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

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

5.1.3.3 GetCol()

```
Vector2D MathEngine::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 MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::Matrix2x2::operator= (
    const Matrix3x3 & m )
```

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 & MathEngine::Matrix2x2::operator= (
    const Matrix4x4 & m )
```

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

5.1.3.13 SetCol()

```
void MathEngine::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 MathEngine::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:

- [MathEngine.h](#)

5.2 MathEngine::Matrix3x3 Class Reference

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

```
#include <MathEngine.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]

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

Creates a new 3x3 identity matrix.

5.2.2.2 Matrix3x3() [2/5]

```
MathEngine::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]

```
MathEngine::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]

```
MathEngine::Matrix3x3::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.

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]

```
MathEngine::Matrix3x3::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.

5.2.3 Member Function Documentation

5.2.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.2.3.2 Data() [2/2]

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

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

5.2.3.3 GetCol()

```
Vector3D MathEngine::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 MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::Matrix3x3::operator= (
    const Matrix2x2 & m )
```

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 & MathEngine::Matrix3x3::operator= (
    const Matrix4x4 & m )
```

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 MathEngine::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 MathEngine::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:

- [MathEngine.h](#)

5.3 MathEngine::Matrix4x4 Class Reference

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

```
#include <MathEngine.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]

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

Creates a new 4x4 identity matrix.

5.3.2.2 `Matrix4x4()` [2/5]

```
MathEngine::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]

```
MathEngine::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]

```
MathEngine::Matrix4x4::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.

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]

```
MathEngine::Matrix4x4::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.

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 * MathEngine::Matrix4x4::Data ( ) [inline]
```

Returns a pointer to the first element in the matrix.

5.3.3.2 Data() [2/2]

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

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

5.3.3.3 GetCol()

```
Vector4D MathEngine::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 MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::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 & MathEngine::Matrix4x4::operator= (
    const Matrix2x2 & m )
```

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 & MathEngine::Matrix4x4::operator= (
    const Matrix3x3 & m )
```

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 MathEngine::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 MathEngine::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:

- [MathEngine.h](#)

5.4 MathEngine::Quaternion Struct Reference

A quaternion struct used for quaternions and their manipulations.

```
#include <MathEngine.h>
```

Public Attributes

- float [scalar](#) = 1.0f
- [Vector3D](#) [vector](#)

5.4.1 Detailed Description

A quaternion struct used for quaternions and their manipulations.

The datatype for the components is float. When making an object of this struct, the quaternion is initialized to the identity quaternion.

5.4.2 Member Data Documentation

5.4.2.1 scalar

```
float MathEngine::Quaternion::scalar = 1.0f
```

5.4.2.2 vector

`Vector3D MathEngine::Quaternion::vector`

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

- [MathEngine.h](#)

5.5 MathEngine::Vector2D Struct Reference

A vector stuct used for 2D vectors/points.

```
#include <MathEngine.h>
```

Public Attributes

- float `x` = 0.0f
- float `y` = 0.0f

5.5.1 Detailed Description

A vector stuct used for 2D vectors/points.

The datatype for the components is float. When an object of this struct is made the components are intialized to 0.0f.

5.5.2 Member Data Documentation

5.5.2.1 x

```
float MathEngine::Vector2D::x = 0.0f
```

5.5.2.2 y

```
float MathEngine::Vector2D::y = 0.0f
```

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

- [MathEngine.h](#)

5.6 MathEngine::Vector3D Struct Reference

A vector stuct used for 3D vectors/points.

```
#include <MathEngine.h>
```

Public Attributes

- float [x](#) = 0.0f
- float [y](#) = 0.0f
- float [z](#) = 0.0f

5.6.1 Detailed Description

A vector stuct used for 3D vectors/points.

The datatype for the components is float. When an object of this struct is made the components are intialized to 0.0f.

5.6.2 Member Data Documentation

5.6.2.1 [x](#)

```
float MathEngine::Vector3D::x = 0.0f
```

5.6.2.2 [y](#)

```
float MathEngine::Vector3D::y = 0.0f
```

5.6.2.3 [z](#)

```
float MathEngine::Vector3D::z = 0.0f
```

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

- [MathEngine.h](#)

5.7 MathEngine::Vector4D Struct Reference

A vector stuct used for 4D vectors/points.

```
#include <MathEngine.h>
```

Public Attributes

- float [x](#) = 0.0f
- float [y](#) = 0.0f
- float [z](#) = 0.0f
- float [w](#) = 0.0f

5.7.1 Detailed Description

A vector stuct used for 4D vectors/points.

The datatype for the components is float. When an object of this struct is made the components are intialized to 0.0f.

5.7.2 Member Data Documentation

5.7.2.1 [w](#)

```
float MathEngine::Vector4D::w = 0.0f
```

5.7.2.2 [x](#)

```
float MathEngine::Vector4D::x = 0.0f
```

5.7.2.3 [y](#)

```
float MathEngine::Vector4D::y = 0.0f
```

5.7.2.4 [z](#)

```
float MathEngine::Vector4D::z = 0.0f
```

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

- [MathEngine.h](#)

Chapter 6

File Documentation

6.1 MathEngine.h File Reference

```
#include <cmath>
```

Classes

- struct [MathEngine::Vector2D](#)
A vector struct used for 2D vectors/points.
- struct [MathEngine::Vector3D](#)
A vector struct used for 3D vectors/points.
- struct [MathEngine::Vector4D](#)
A vector struct used for 4D vectors/points.
- class [MathEngine::Matrix2x2](#)
A matrix class used for 2x2 matrices and their manipulations.
- class [MathEngine::Matrix3x3](#)
A matrix class used for 3x3 matrices and their manipulations.
- class [MathEngine::Matrix4x4](#)
A matrix class used for 4x4 matrices and their manipulations.
- struct [MathEngine::Quaternion](#)
A quaternion struct used for quaternions and their manipulations.

Namespaces

- namespace [MathEngine](#)

Macros

- #define [EPSILON](#) 1e-6f
- #define [PI](#) 3.14159f
- #define [PI2](#) 6.28319f

Typedefs

- typedef [MathEngine::Vector2D](#) [vec2](#)
- typedef [MathEngine::Vector3D](#) [vec3](#)
- typedef [MathEngine::Vector4D](#) [vec4](#)
- typedef [MathEngine::Matrix2x2](#) [mat2](#)
- typedef [MathEngine::Matrix3x3](#) [mat3](#)
- typedef [MathEngine::Matrix4x4](#) [mat4](#)
- typedef [MathEngine::Quaternion](#) [quaternion](#)

Functions

- bool [MathEngine::CompareFloats](#) (float x, float y, float epsilon)
Returns true if x and y are equal.
- bool [MathEngine::CompareDoubles](#) (double x, double y, double epsilon)
Returns true if x and y are equal.
- float [MathEngine::Clamp](#) (float value, float a, float b)
Returns a clamped value.
- void [MathEngine::operator+=](#) (Vector2D &v1, const Vector2D &v2)
Adds the 2D vector v1 to the 2D vector v2 and stores the result in v1.
- void [MathEngine::operator-=](#) (Vector2D &v1, const Vector2D &v2)
Subtracts the the 2D vector v2 from the 2D vector v1 and stores the result in v1.
- void [MathEngine::operator*=](#) (Vector2D &v, float k)
Multiplies the 2D vector v by the scalar (float) k and stores the result in v.
- Vector2D [MathEngine::operator+](#) (const Vector2D &v1, const Vector2D &v2)
Adds the two 2D vectors and returns the result.
- Vector2D [MathEngine::operator-](#) (const Vector2D &v)
Negates the 3D vector v1 and returns the result.
- Vector2D [MathEngine::operator-](#) (const Vector2D &v1, const Vector2D &v2)
Subtracts the 2D vector v2 from the 2D vector v1 and returns the result.
- Vector2D [MathEngine::operator*](#) (const Vector2D &v, float k)
Multiplies the 2D vector v by the scalar (float) k and returns the result.
- Vector2D [MathEngine::operator*](#) (float k, const Vector2D &v)
Multiplies the scalar (float) k by the 2D vector v and returns the result.
- bool [MathEngine::operator==](#) (const Vector2D &v1, const Vector2D &v2)
Returns true if the 2D vector v1 equals to the 2D vector v2, false otherwise.
- bool [MathEngine::operator!=](#) (const Vector2D &v1, const Vector2D &v2)
Returns true if the 2D vector v1 does not equal to the 2D vector v2, false otherwise.
- bool [MathEngine::ZeroVector](#) (const Vector2D &v)
Returns true if the 2D vector v is equal to the zero vector, false otherwise.
- float [MathEngine::DotProduct](#) (const Vector2D &v1, const Vector2D &v2)
Returns the dot product between the 2D vectors v1 and v2.
- float [MathEngine::Length](#) (const Vector2D &v)
Returns the length (magnitude) of the 2D vector v.
- Vector2D [MathEngine::Normalize](#) (const Vector2D &v)
Normalizes (makes it unit length) the 2D vector v and returns the result.
- Vector2D [MathEngine::Lerp](#) (const Vector2D &start, const Vector2D &end, float t)
Linear interpolate between the two vectors start and end.
- void [MathEngine::operator+=](#) (Vector3D &v1, const Vector3D &v2)
Adds the 3D vector v1 to the 3D vector v2 and stores the result in v1.
- void [MathEngine::operator-=](#) (Vector3D &v1, const Vector3D &v2)

- Subtracts the the 3D vector v2 from the 3D vector v1 and stores the result in v1.*

 - void `MathEngine::operator*=(Vector3D &v, float k)`

Multiplies the 3D vector v by the scalar (float) k and stores the result in v.

 - Vector3D `MathEngine::operator+(const Vector3D &v1, const Vector3D &v2)`

Adds the two 3D vectors and returns the result.

 - Vector3D `MathEngine::operator-(const Vector3D &v)`

Negates the 3D vector v and returns the result.

 - Vector3D `MathEngine::operator-(const Vector3D &v1, const Vector3D &v2)`

Subtracts the 3D vector v2 from the 3D vector v1 and returns the result.

 - Vector3D `MathEngine::operator*(const Vector3D &v, float k)`

Multiplies the 3D vector v by the scalar (float) k and returns the result.

 - Vector3D `MathEngine::operator*(float k, const Vector3D &v)`

Multiplies the scalar (float) k by the 3D vector v and returns the result.

 - bool `MathEngine::operator==(const Vector3D &v1, const Vector3D &v2)`

Returns true if the 3D vector v1 equals to the 3D vector v2, false otherwise.

 - bool `MathEngine::operator!=(const Vector3D &v1, const Vector3D &v2)`

Returns true if the 3D vector v1 does not equal to the 3D vector v2, false otherwise.

 - bool `MathEngine::ZeroVector(const Vector3D &v)`

Returns true if the 3D vector v is equal to the zero vector, false otherwise.

 - float `MathEngine::DotProduct(const Vector3D &v1, const Vector3D &v2)`

Returns the dot product between the 3D vectors v1 and v2.

 - Vector3D `MathEngine::CrossProduct(const Vector3D &v1, const Vector3D &v2)`

Returns the cross product between the 3D vectors v1 and v2.

 - float `MathEngine::Length(const Vector3D &v)`

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

 - Vector3D `MathEngine::Normalize(const Vector3D &v)`

Normalizes (makes it unit length) the 3D vector v and returns the result.

 - void `MathEngine::Orthonormalize(Vector3D &x, Vector3D &y, Vector3D &z)`

Orthonormalizes the specified vectors.

 - Vector3D `MathEngine::Lerp(const Vector3D &start, const Vector3D &end, float t)`

Linear interpolate between the two vectors start and end.

 - void `MathEngine::operator+=(Vector4D &v1, const Vector4D &v2)`

Adds the 4D vector v1 to the 4D vector v2 and stores the result in v1.

 - void `MathEngine::operator-=(Vector4D &v1, const Vector4D &v2)`

Subtracts the the 4D vector v2 from the 4D vector v1 and stores the result in v1.

 - void `MathEngine::operator*=(Vector4D &v, float k)`

Multiplies the 4D vector v by the scalar (float) k and stores the result in v.

 - Vector4D `MathEngine::operator+(const Vector4D &v1, const Vector4D &v2)`

Adds the two 4D vectors and returns the result.

 - Vector4D `MathEngine::operator-(const Vector4D &v)`

Negates the 4D vector v and returns the result.

 - Vector4D `MathEngine::operator-(const Vector4D &v1, const Vector4D &v2)`

Subtracts the 4D vector v2 from the 4D vector v1 and returns the result.

 - Vector4D `MathEngine::operator*(const Vector4D &v, float k)`

Multiplies the 4D vector v by the scalar (float) k and returns the result.

 - Vector4D `MathEngine::operator*(float k, const Vector4D &v)`

Multiplies the scalar (float) k by the 4D vector v and returns the result.

 - bool `MathEngine::operator==(const Vector4D &v1, const Vector4D &v2)`

Returns true if the 4D vector v1 equals to the 4D vector v2, false otherwise.

 - bool `MathEngine::operator!=(const Vector4D &v1, const Vector4D &v2)`

Returns true if the 4D vector v1 does not equal to the 4D vector v2, false otherwise.

- bool [MathEngine::ZeroVector](#) (const Vector4D &v)
Returns true if the 4D vector v is equal to the zero vector, false otherwise.
- float [MathEngine::DotProduct](#) (const Vector4D &v1, const Vector4D &v2)
Returns the dot product between the 4D vectors v1 and v2.
- float [MathEngine::Length](#) (const Vector4D &v)
Returns the length (magnitude) of the 4D vector v.
- Vector4D [MathEngine::Normalize](#) (const Vector4D &v)
Normalizes (makes it unit length) the 4D vector v and returns the result.
- Vector4D [MathEngine::Lerp](#) (const Vector4D &start, const Vector4D &end, float t)
Linear interpolate between the two vectors start and end.
- Matrix2x2 [MathEngine::operator+](#) (const Matrix2x2 &m1, const Matrix2x2 &m2)
Adds m1 with m2 and returns the result.
- Matrix2x2 [MathEngine::operator-](#) (const Matrix2x2 &m)
Negates the 2x2 matrix m.
- Matrix2x2 [MathEngine::operator-](#) (const Matrix2x2 &m1, const Matrix2x2 &m2)
Subtracts m2 from m1 and returns the result.
- Matrix2x2 [MathEngine::operator*](#) (const Matrix2x2 &m, const float &k)
Multiplies m with k and returns the result.
- Matrix2x2 [MathEngine::operator*](#) (const float &k, const Matrix2x2 &m)
Multiplies k with m and returns the result.
- Matrix2x2 [MathEngine::operator*](#) (const Matrix2x2 &m1, const Matrix2x2 &m2)
Multiplies m1 with m2 and returns the result.
- Vector2D [MathEngine::operator*](#) (const Matrix2x2 &m, const Vector2D &v)
Multiplies m with v and returns the result.
- Vector2D [MathEngine::operator*](#) (const Vector2D &v, const Matrix2x2 &m)
Multiplies v with m and returns the result.
- void [MathEngine::SetToIdentity](#) (Matrix2x2 &m)
Sets m to the identity matrix.
- bool [MathEngine::Identity](#) (const Matrix2x2 &m)
Returns true if m is the identity matrix, false otherwise.
- Matrix2x2 [MathEngine::Transpose](#) (const Matrix2x2 &m)
Returns the tranpose of the given matrix m.
- Matrix2x2 [MathEngine::Scale](#) (float x, float y)
Returns a 2x2 scaling matrix.
- Matrix2x2 [MathEngine::Scale](#) (const Vector2D &scaleVector)
Returns a 2x2 scaling matrix.
- Matrix2x2 [MathEngine::Rotate](#) (float angle)
Returns a 2x2 rotation matrix that rotates a point/vector about the origin.
- double [MathEngine::Determinant](#) (const Matrix2x2 &m)
Returns the determinant of m.
- double [MathEngine::Cofactor](#) (const Matrix2x2 &m, unsigned int row, unsigned int col)
Returns the cofactor of the row and col in m.
- Matrix2x2 [MathEngine::Adjoint](#) (const Matrix2x2 &m)
Returns the adjoint of m.
- Matrix2x2 [MathEngine::Inverse](#) (const Matrix2x2 &m)
Returns the inverse of m.
- Matrix3x3 [MathEngine::operator+](#) (const Matrix3x3 &m1, const Matrix3x3 &m2)
Adds m1 with m2 and returns the result.
- Matrix3x3 [MathEngine::operator-](#) (const Matrix3x3 &m)
Negates the 3x3 matrix m.
- Matrix3x3 [MathEngine::operator-](#) (const Matrix3x3 &m1, const Matrix3x3 &m2)

- Subtracts m2 from m1 and returns the result.*

 - Matrix3x3 [MathEngine::operator*](#) (const Matrix3x3 &m, const float &k)

Multiplies m with k and returns the result.

 - Matrix3x3 [MathEngine::operator*](#) (const float &k, const Matrix3x3 &m)

Multiplies k with \m and returns the result.

 - Matrix3x3 [MathEngine::operator*](#) (const Matrix3x3 &m1, const Matrix3x3 &m2)

Multiplies m1 with \m2 and returns the result.

 - Vector3D [MathEngine::operator*](#) (const Matrix3x3 &m, const Vector3D &v)

Multiplies m with v and returns the result.

 - Vector3D [MathEngine::operator*](#) (const Vector3D &v, const Matrix3x3 &m)

Multiplies v with m and returns the result.

 - void [MathEngine::SetTolIdentity](#) (Matrix3x3 &m)

Sets m to the identity matrix.

 - bool [MathEngine::Identity](#) (const Matrix3x3 &m)

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

 - Matrix3x3 [MathEngine::Transpose](#) (const Matrix3x3 &m)

Returns the tranpose of the given matrix m.

 - Matrix3x3 [MathEngine::Scale](#) (float x, float y, float z)
 - Matrix3x3 [MathEngine::Scale](#) (const Vector3D &scaleVector)
 - Matrix3x3 [MathEngine::Rotate](#) (float angle, float x, float y, float z)
 - Matrix3x3 [MathEngine::Rotate](#) (float angle, const Vector3D &axis)
 - double [MathEngine::Determinant](#) (const Matrix3x3 &m)

Returns the determinant of m.

 - double [MathEngine::Cofactor](#) (const Matrix3x3 &m, unsigned int row, unsigned int col)

Returns the cofactor of the row and col in m.

 - Matrix3x3 [MathEngine::Adjoint](#) (const Matrix3x3 &m)

Returns the adjoint of m.

 - Matrix3x3 [MathEngine::Inverse](#) (const Matrix3x3 &m)

Returns the inverse of m.

 - Matrix4x4 [MathEngine::operator+](#) (const Matrix4x4 &m1, const Matrix4x4 &m2)

Adds m1 with m2 and returns the result.

 - Matrix4x4 [MathEngine::operator-](#) (const Matrix4x4 &m)

Negates the 4x4 matrix m.

 - Matrix4x4 [MathEngine::operator-](#) (const Matrix4x4 &m1, const Matrix4x4 &m2)

Subtracts m2 from m1 and returns the result.

 - Matrix4x4 [MathEngine::operator*](#) (const Matrix4x4 &m, const float &k)

Multiplies m with k and returns the result.

 - Matrix4x4 [MathEngine::operator*](#) (const float &k, const Matrix4x4 &m)

Multiplies k with \m and returns the result.

 - Matrix4x4 [MathEngine::operator*](#) (const Matrix4x4 &m1, const Matrix4x4 &m2)

Multiplies m1 with \m2 and returns the result.

 - Vector4D [MathEngine::operator*](#) (const Matrix4x4 &m, const Vector4D &v)

Multiplies m with v and returns the result.

 - Vector4D [MathEngine::operator*](#) (const Vector4D &v, const Matrix4x4 &m)

Multiplies v with m and returns the result.

 - void [MathEngine::SetTolIdentity](#) (Matrix4x4 &m)

Sets m to the identity matrix.

 - bool [MathEngine::Identity](#) (const Matrix4x4 &m)

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

 - Matrix4x4 [MathEngine::Transpose](#) (const Matrix4x4 &m)

Returns the tranpose of the given matrix m.

- Matrix4x4 [MathEngine::Translate](#) (float x, float y, float z)
Returns a 4x4 translation matrix.
- Matrix4x4 [MathEngine::Translate](#) (const Vector3D &translateVector)
Returns a 4x4 translation matrix.
- Matrix4x4 [MathEngine::Scale4x4](#) (float x, float y, float z)
Returns a 4x4 scale matrix.
- Matrix4x4 [MathEngine::Scale4x4](#) (const Vector3D &scaleVector)
Returns a 4x4 scale matrix.
- Matrix4x4 [MathEngine::Rotate4x4](#) (float angle, float x, float y, float z)
Returns a 4x4 rotation matrix about the given axis.
- Matrix4x4 [MathEngine::Rotate4x4](#) (float angle, const Vector3D &axis)
Returns a 4x4 rotation matrix about the given axis.
- double [MathEngine::Determinant](#) (const Matrix4x4 &m)
Returns the determinant m.
- double [MathEngine::Cofactor](#) (const Matrix4x4 &m, unsigned int row, unsigned int col)
Returns the cofactor of the row and col in m.
- Matrix4x4 [MathEngine::Adjoint](#) (const Matrix4x4 &m)
Returns the adjoint of m.
- Matrix4x4 [MathEngine::Inverse](#) (const Matrix4x4 &m)
Returns the inverse of m.
- void [MathEngine::operator+=](#) (Quaternion &q1, const Quaternion &q2)
Adds the quaternion q1 to the quaternion q2 and stores the result in q1.
- void [MathEngine::operator-=](#) (Quaternion &q1, const Quaternion &q2)
Subtracts the quaternion q2 from the quaternion q1 and stores the result in q1.
- void [MathEngine::operator*=](#) (Quaternion &q1, float k)
Multiplies the quaternion q1 by the scalar (float) k and stores the result in q1.
- void [MathEngine::operator*=](#) (Quaternion &q1, const Quaternion &q2)
Multiplies the quaternion q1 by the quaternion q1 and stores the result in q1.
- Quaternion [MathEngine::operator+](#) (const Quaternion &q1, const Quaternion &q2)
Adds the quaternion q1 to the quaternion q2 and returns the result.
- Quaternion [MathEngine::operator-](#) (const Quaternion &q1, const Quaternion &q2)
Subtracts the quaternion q2 from the quaternion q1 and returns the result.
- Quaternion [MathEngine::operator-](#) (const Quaternion &q)
Negates the quaternion q1 and returns the result.
- Quaternion [MathEngine::operator*](#) (const Quaternion &q, float k)
Multiplies the quaternion q by the scalar (float) k and returns the result.
- Quaternion [MathEngine::operator*](#) (float k, const Quaternion &q)
Multiplies the scalar (float) k by the quaternion q and returns the result.
- Quaternion [MathEngine::operator*](#) (const Quaternion &q1, const Quaternion &q2)
Multiplies the quaternion q1 by the quaternion q2 and returns the result.
- bool [MathEngine::operator==](#) (const Quaternion &q1, const Quaternion &q2)
Returns true if the quaternion q1 equals to the quaternion q2, false otherwise.
- bool [MathEngine::operator!=](#) (const Quaternion &q1, const Quaternion &q2)
Returns true if the quaternion q1 does not equal to the quaternion q2, false otherwise.
- bool [MathEngine::ZeroQuaternion](#) (const Quaternion &q)
Returns true if quaternion q is a zero quaternion, false otherwise.
- bool [MathEngine::Identity](#) (const Quaternion &q)
Returns true if quaternion q is an identity quaternion, false otherwise.
- Quaternion [MathEngine::Conjugate](#) (const Quaternion &q)
Returns the conjugate of quaternion q.
- float [MathEngine::Length](#) (const Quaternion &q)

- Returns the length of quaternion q .*
- Quaternion [MathEngine::Normalize](#) (const Quaternion &q)
Normalizes the quaternion q and returns the normalized quaternion.
- Quaternion [MathEngine::Inverse](#) (const Quaternion &q)
Returns the invese of the quaternion q .
- Quaternion [MathEngine::RotationQuaternion](#) (float angle, float x, float y, float z)
Returns a rotation quaternion from the axis-angle representation.
- Quaternion [MathEngine::RotationQuaternion](#) (float angle, const Vector3D &axis)
Returns a quaternion from the axis-angle representation.
- Quaternion [MathEngine::RotationQuaternion](#) (const Vector4D &angAxis)
Returns a quaternion from the axis-angle representation.
- Vector3D [MathEngine::Rotate](#) (const Quaternion &q, const Vector3D &p)
Rotates the specified point/vector p using the quaternion q .
- Vector4D [MathEngine::Rotate](#) (const Quaternion &q, const Vector4D &p)
Rotates the specified point/vector p using the quaternion q .
- Matrix3x3 [MathEngine::QuaternionToRotationMatrixCol3x3](#) (const Quaternion &q)
Transforms q into a column-major matrix.
- Matrix3x3 [MathEngine::QuaternionToRotationMatrixRow3x3](#) (const Quaternion &q)
Transforms q into a row-major matrix.
- Matrix4x4 [MathEngine::QuaternionToRotationMatrixCol4x4](#) (const Quaternion &q)
Transforms q into a column-major matrix.
- Matrix4x4 [MathEngine::QuaternionToRotationMatrixRow4x4](#) (const Quaternion &q)
Transforms q into a row-major matrix.
- float [MathEngine::DotProduct](#) (const Quaternion &q1, const Quaternion &q2)
Returns the dot product of the quaternions $q1$ and $q2$.
- Quaternion [MathEngine::Lerp](#) (const Quaternion &q1, const Quaternion &q2, float t)
Linear Interpolates between quaternions $q1$ and $q2$.
- Quaternion [MathEngine::NLerp](#) (const Quaternion &q1, const Quaternion &q2, float t)
Linear Interpolates between quaternions $q1$ and $q2$ and normalizes the result.
- Quaternion [MathEngine::Slerp](#) (const Quaternion &q1, const Quaternion &q2, float t)
Spherical Linear Interpolates between quaternions $q1$ and $q2$.

6.1.1 Macro Definition Documentation

6.1.1.1 EPSILON

```
#define EPSILON 1e-6f
```

6.1.1.2 PI

```
#define PI 3.14159f
```

6.1.1.3 PI2

```
#define PI2 6.28319f
```

6.1.2 Typedef Documentation

6.1.2.1 mat2

```
typedef MathEngine::Matrix2x2 mat2
```

6.1.2.2 mat3

```
typedef MathEngine::Matrix3x3 mat3
```

6.1.2.3 mat4

```
typedef MathEngine::Matrix4x4 mat4
```

6.1.2.4 quaternion

```
typedef MathEngine::Quaternion quaternion
```

6.1.2.5 vec2

```
typedef MathEngine::Vector2D vec2
```

6.1.2.6 vec3

```
typedef MathEngine::Vector3D vec3
```

6.1.2.7 vec4

```
typedef MathEngine::Vector4D vec4
```

6.2 MathEngine.h

[Go to the documentation of this file.](#)

```
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 MathEngine
15 {
16
17     struct Vector2D;
18     struct Vector3D;
19     struct Vector4D;
20     class Matrix2x2;
21     class Matrix3x3;
22     class Matrix4x4;
23
24
25 //-----
26 //COMPARISON FUNCTIONS
27
28     inline bool CompareFloats(float x, float y, float epsilon)
29     {
30         float diff = fabs(x - y);
31         //exact epsilon
32         if (diff < epsilon)
33         {
34             return true;
35         }
36
37         //adapative epsilon
38         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
39     }
40
41     inline bool CompareDoubles(double x, double y, double epsilon)
42     {
43         double diff = fabs(x - y);
44         //exact epsilon
45         if (diff < epsilon)
46         {
47             return true;
48         }
49
50         //adapative epsilon
51         return diff <= epsilon * (((fabs(x)) > (fabs(y))) ? (fabs(x)) : (fabs(y)));
52     }
53
54     inline float Clamp(float value, float a, float b)
55     {
56         if (value < a)
57             return a;
58
59         if (value > b)
60             return b;
61
62         return value;
63     }
64
65 //-----
66
67 //-----
68
69 //2D VECTOR
70
71     struct Vector2D
```

```

87     {
88         float x = 0.0f;
89         float y = 0.0f;
90     };
91
92     inline void operator+=(Vector2D& v1, const Vector2D& v2)
93     {
94         v1.x += v2.x;
95         v1.y += v2.y;
96     }
97
98     inline void operator-=(Vector2D& v1, const Vector2D& v2)
99     {
100         v1.x -= v2.x;
101         v1.y -= v2.y;
102     }
103
104     inline void operator*=(Vector2D& v, float k)
105     {
106         v.x *= k;
107         v.y *= k;
108     }
109
110     inline Vector2D operator+(const Vector2D& v1, const Vector2D& v2)
111     {
112         return Vector2D{ v1.x + v2.x, v1.y + v2.y };
113     }
114
115     inline Vector2D operator-(const Vector2D& v)
116     {
117         return Vector2D{ -v.x, -v.y };
118     }
119
120     inline Vector2D operator-(const Vector2D& v1, const Vector2D& v2)
121     {
122         return Vector2D{ v1.x - v2.x, v1.y - v2.y };
123     }
124
125     inline Vector2D operator*(const Vector2D& v, float k)
126     {
127         return Vector2D{ v.x * k, v.y * k };
128     }
129
130     inline Vector2D operator*(float k, const Vector2D& v)
131     {
132         return Vector2D{ k * v.x, k * v.y };
133     }
134
135     inline bool operator==(const Vector2D& v1, const Vector2D& v2)
136     {
137         return CompareFloats(v1.x, v2.x, EPSILON) && CompareFloats(v1.y, v2.y, EPSILON);
138     }
139
140     inline bool operator!=(const Vector2D& v1, const Vector2D& v2)
141     {
142         return !(v1 == v2);
143     }
144
145     inline bool ZeroVector(const Vector2D& v)
146     {
147         return CompareFloats(v.x, 0.0f, EPSILON) && CompareFloats(v.y, 0.0f, EPSILON);
148     }
149
150     inline float DotProduct(const Vector2D& v1, const Vector2D& v2)
151     {
152         return v1.x * v2.x + v1.y * v2.y;
153     }
154
155     inline float Length(const Vector2D& v)
156     {
157         return sqrt(v.x * v.x + v.y * v.y);
158     }
159
160     inline Vector2D Normalize(const Vector2D& v)
161     {
162         if (ZeroVector(v))
163             return v;
164
165         float inverseLength{ 1.0f / Length(v) };
166
167         return v * inverseLength;
168     }
169
170     inline Vector2D Lerp(const Vector2D& start, const Vector2D& end, float t)
171     {
172         if (t < 0.0f)
173             return start;

```



```

212         else if (t > 1.0f)
213             return end;
214
215         return (1.0f - t) * start + t * end;
216     }
217
218 #if defined(_DEBUG)
219     inline void print(const Vector2D& v)
220     {
221         std::cout << "(" << v.x << ", " << v.y << ")";
222     }
223 #endif
224
225 //-----
226
227 //-----
228 //3D VECTOR
229
230 struct Vector3D
231 {
232     float x = 0.0f;
233     float y = 0.0f;
234     float z = 0.0f;
235 };
236
237 inline void operator+=(Vector3D& v1, const Vector3D& v2)
238 {
239     v1.x += v2.x;
240     v1.y += v2.y;
241     v1.z += v2.z;
242 }
243
244 inline void operator-=(Vector3D& v1, const Vector3D& v2)
245 {
246     v1.x -= v2.x;
247     v1.y -= v2.y;
248     v1.z -= v2.z;
249 }
250
251 inline void operator*(Vector3D& v, float k)
252 {
253     v.x *= k;
254     v.y *= k;
255     v.z *= k;
256 }
257
258 inline Vector3D operator+(const Vector3D& v1, const Vector3D& v2)
259 {
260     return Vector3D{ v1.x + v2.x, v1.y + v2.y, v1.z + v2.z };
261 }
262
263 inline Vector3D operator-(const Vector3D& v)
264 {
265     return Vector3D{ -v.x, -v.y, -v.z };
266 }
267
268 inline Vector3D operator-(const Vector3D& v1, const Vector3D& v2)
269 {
270     return Vector3D{ v1.x - v2.x, v1.y - v2.y, v1.z - v2.z };
271 }
272
273 inline Vector3D operator*(const Vector3D& v, float k)
274 {
275     return Vector3D{ v.x * k, v.y * k, v.z * k };
276 }
277
278 inline Vector3D operator*(float k, const Vector3D& v)
279 {
280     return Vector3D{ k * v.x, k * v.y, k * v.z };
281 }
282
283 inline bool operator==(const Vector3D& v1, const Vector3D& v2)
284 {
285     return CompareFloats(v1.x, v2.x, EPSILON) && CompareFloats(v1.y, v2.y, EPSILON) &&
286            CompareFloats(v1.z, v2.z, EPSILON);
287 }
288
289 inline bool operator!=(const Vector3D& v1, const Vector3D& v2)
290 {
291     return !(v1 == v2);
292 }
293
294 inline bool ZeroVector(const Vector3D& v)
295 {
296

```

```

327         return CompareFloats(v.x, 0.0f, EPSILON) && CompareFloats(v.y, 0.0f, EPSILON) &&
CompareFloats(v.z, 0.0f, EPSILON);
328     }
329
330     inline float DotProduct(const Vector3D& v1, const Vector3D& v2)
331     {
332         return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
333     }
334
335     inline Vector3D CrossProduct(const Vector3D& v1, const Vector3D& v2)
336     {
337         return Vector3D{ v1.y * v2.z - v1.z * v2.y, v1.z * v2.x - v1.x * v2.z, v1.x * v2.y - v1.y * v2.x
};
338     }
339
340     inline float Length(const Vector3D& v)
341     {
342         return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
343     }
344
345     inline Vector3D Normalize(const Vector3D& v)
346     {
347         if (ZeroVector(v))
348             return v;
349
350         float inverseLength{ 1.0f / Length(v) };
351
352         return v * inverseLength;
353     }
354
355     inline void Orthonormalize(Vector3D& x, Vector3D& y, Vector3D& z)
356     {
357         x = Normalize(x);
358         y = Normalize(CrossProduct(z, x));
359         z = Normalize(CrossProduct(x, y));
360     }
361
362     inline Vector3D Lerp(const Vector3D& start, const Vector3D& end, float t)
363     {
364         if (t < 0.0f)
365             return start;
366         else if (t > 1.0f)
367             return end;
368
369         return (1.0f - t) * start + t * end;
370     }
371
372     #if defined(_DEBUG)
373     inline void print(const Vector3D& v)
374     {
375         std::cout << "(" << v.x << ", " << v.y << ")";
376     }
377     #endif
378
379     //-----
380
381     //-----
382
383     //4D VECTOR
384
385     struct Vector4D
386     {
387         float x = 0.0f;
388         float y = 0.0f;
389         float z = 0.0f;
390         float w = 0.0f;
391     };
392
393     inline void operator+=(Vector4D& v1, const Vector4D& v2)
394     {
395         v1.x += v2.x;
396         v1.y += v2.y;
397         v1.z += v2.z;
398         v1.w += v2.w;
399     }
400
401     inline void operator-=(Vector4D& v1, const Vector4D& v2)
402     {
403         v1.x -= v2.x;
404         v1.y -= v2.y;
405         v1.z -= v2.z;
406         v1.w -= v2.w;
407     }

```

```

439 inline void operator*=(Vector4D& v, float k)
440 {
441     v.x *= k;
442     v.y *= k;
443     v.z *= k;
444     v.w *= k;
445 }
446
449 inline Vector4D operator+(const Vector4D& v1, const Vector4D& v2)
450 {
451     return Vector4D{ v1.x + v2.x, v1.y + v2.y, v1.z + v2.z, v1.w + v2.w };
452 }
453
456 inline Vector4D operator-(const Vector4D& v)
457 {
458     return Vector4D{ -v.x, -v.y, -v.z, -v.w };
459 }
460
465 inline Vector4D operator-(const Vector4D& v1, const Vector4D& v2)
466 {
467     return Vector4D{ v1.x - v2.x, v1.y - v2.y, v1.z - v2.z, v1.w - v2.w };
468 }
469
472 inline Vector4D operator*(const Vector4D& v, float k)
473 {
474     return Vector4D{ v.x * k, v.y * k, v.z * k, v.w * k };
475 }
476
479 inline Vector4D operator*(float k, const Vector4D& v)
480 {
481     return Vector4D{ k * v.x, k * v.y, k * v.z, k * v.w };
482 }
483
486 inline bool operator==(const Vector4D& v1, const Vector4D& v2)
487 {
488     return CompareFloats(v1.x, v2.x, EPSILON) && CompareFloats(v1.y, v2.y, EPSILON) &&
CompareFloats(v1.z, v2.z, EPSILON)
&& CompareFloats(v1.w, v2.w, EPSILON);
489 }
490
491
494 inline bool operator!=(const Vector4D& v1, const Vector4D& v2)
495 {
496     return !(v1 == v2);
497 }
498
501 inline bool ZeroVector(const Vector4D& v)
502 {
503     return CompareFloats(v.x, 0.0f, EPSILON) && CompareFloats(v.y, 0.0f, EPSILON) &&
CompareFloats(v.z, 0.0f, EPSILON)
&& CompareFloats(v.w, 0.0f, EPSILON);
504 }
505
506
509 inline float DotProduct(const Vector4D& v1, const Vector4D& v2)
510 {
511     return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z + v1.w * v2.w;
512 }
513
516 inline float Length(const Vector4D& v)
517 {
518     return sqrt(v.x * v.x + v.y * v.y + v.z * v.z + v.w * v.w);
519 }
520
525 inline Vector4D Normalize(const Vector4D& v)
526 {
527     if (ZeroVector(v))
528         return v;
529
530     float inverseLength{ 1.0f / Length(v) };
531
532     return v * inverseLength;
533 }
534
539 inline Vector4D Lerp(const Vector4D& start, const Vector4D& end, float t)
540 {
541     if (t < 0.0f)
542         return start;
543     else if (t > 1.0f)
544         return end;
545
546     return (1.0f - t) * start + t * end;
547 }
548
549 #if defined(_DEBUG)
550 inline void print(const Vector4D& v)
551 {
552     std::cout << "(" << v.x << ", " << v.y << ")";
553 }

```

```

554 #endif
555
556 //-----
557
558
559
560 //-----
561
562 class Matrix2x2
563 {
564 public:
565     Matrix2x2();
566     Matrix2x2(float a[][2]);
567     Matrix2x2(const Vector2D& r1, const Vector2D& r2);
568     Matrix2x2(const Matrix3x3& m);
569     Matrix2x2(const Matrix4x4& m);
570     float* Data();
571     const float* Data() const;
572     const float& operator()(unsigned int row, unsigned int col) const;
573     float& operator()(unsigned int row, unsigned int col);
574     Vector2D GetRow(unsigned int row) const;
575     Vector2D GetCol(unsigned int col) const;
576     void SetRow(unsigned int row, Vector2D v);
577     void SetCol(unsigned int col, Vector2D v);
578     Matrix2x2& operator=(const Matrix3x3& m);
579     Matrix2x2& operator=(const Matrix4x4& m);
580     Matrix2x2& operator+=(const Matrix2x2& m);
581     Matrix2x2& operator-=(const Matrix2x2& m);
582     Matrix2x2& operator*=(float k);
583     Matrix2x2& operator*=(const Matrix2x2& m);
584 private:
585     float mMat[2][2];
586 };
587 //-----
588 inline Matrix2x2::Matrix2x2()
589 {
590     //1st row
591     mMat[0][0] = 1.0f;
592     mMat[0][1] = 0.0f;
593     //2nd
594     mMat[1][0] = 0.0f;
595     mMat[1][1] = 1.0f;
596 }
597 inline Matrix2x2::Matrix2x2(float a[][2])
598 {
599     //1st row
600     mMat[0][0] = a[0][0];
601     mMat[0][1] = a[0][1];
602     //2nd row
603     mMat[1][0] = a[1][0];
604     mMat[1][1] = a[1][1];
605 }
606 inline Matrix2x2::Matrix2x2(const Vector2D& r1, const Vector2D& r2)
607 {
608     SetRow(0, r1);
609     SetRow(1, r2);
610 }
611 inline float* Matrix2x2::Data()
612 {

```

```

698         return mMat[0];
699     }
700
701     inline const float* Matrix2x2::Data() const
702     {
703         return mMat[0];
704     }
705
706     inline const float& Matrix2x2::operator()(unsigned int row, unsigned int col) const
707     {
708         if (row > 1 || col > 1)
709         {
710             return mMat[0][0];
711         }
712         else
713         {
714             return mMat[row][col];
715         }
716     }
717
718     inline float& Matrix2x2::operator()(unsigned int row, unsigned int col)
719     {
720         if (row > 1 || col > 1)
721         {
722             return mMat[0][0];
723         }
724         else
725         {
726             return mMat[row][col];
727         }
728     }
729
730     inline Vector2D Matrix2x2::GetRow(unsigned int row) const
731     {
732         if (row < 0 || row > 1)
733             return Vector2D{ mMat[0][0], mMat[0][1] };
734         else
735             return Vector2D{ mMat[row][0], mMat[row][1] };
736     }
737
738     inline Vector2D Matrix2x2::GetCol(unsigned int col) const
739     {
740         if (col < 0 || col > 1)
741             return Vector2D{ mMat[0][0], mMat[1][0] };
742         else
743             return Vector2D{ mMat[0][col], mMat[1][col] };
744     }
745
746     inline void Matrix2x2::SetRow(unsigned int row, Vector2D v)
747     {
748         if (row > 1)
749         {
750             {
751                 mMat[0][0] = v.x;
752                 mMat[0][1] = v.y;
753             }
754             else
755             {
756                 mMat[row][0] = v.x;
757                 mMat[row][1] = v.y;
758             }
759         }
760
761     inline void Matrix2x2::SetCol(unsigned int col, Vector2D v)
762     {
763         if (col > 1)
764         {
765             {
766                 mMat[0][0] = v.x;
767                 mMat[1][0] = v.y;
768             }
769             else
770             {
771                 mMat[0][col] = v.x;
772                 mMat[1][col] = v.y;
773             }
774         }
775
776     inline Matrix2x2& Matrix2x2::operator+=(const Matrix2x2& m)
777     {
778         for (int i = 0; i < 2; ++i)
779         {
780             for (int j = 0; j < 2; ++j)
781             {
782                 this->mMat[i][j] += m.mMat[i][j];
783             }
784         }

```

```

785         return *this;
786     }
787
788     inline Matrix2x2& Matrix2x2::operator-=(const Matrix2x2& m)
789     {
790         for (int i = 0; i < 2; ++i)
791         {
792             for (int j = 0; j < 2; ++j)
793             {
794                 this->mMat[i][j] -= m.mMat[i][j];
795             }
796         }
797
798         return *this;
799     }
800
801     inline Matrix2x2& Matrix2x2::operator*=(float k)
802     {
803         for (int i = 0; i < 2; ++i)
804         {
805             for (int j = 0; j < 2; ++j)
806             {
807                 this->mMat[i][j] *= k;
808             }
809         }
810
811         return *this;
812     }
813
814     inline Matrix2x2& Matrix2x2::operator*=(const Matrix2x2& m)
815     {
816         Matrix2x2 res;
817
818         for (int i = 0; i < 2; ++i)
819         {
820             res.mMat[i][0] =
821                 (mMat[i][0] * m.mMat[0][0]) +
822                 (mMat[i][1] * m.mMat[1][0]);
823
824             res.mMat[i][1] =
825                 (mMat[i][0] * m.mMat[0][1]) +
826                 (mMat[i][1] * m.mMat[1][1]);
827         }
828
829         for (int i = 0; i < 2; ++i)
830         {
831             for (int j = 0; j < 2; ++j)
832             {
833                 mMat[i][j] = res.mMat[i][j];
834             }
835         }
836
837         return *this;
838     }
839
840     inline Matrix2x2 operator+(const Matrix2x2& m1, const Matrix2x2& m2)
841     {
842         Matrix2x2 res;
843         for (int i = 0; i < 2; ++i)
844         {
845             for (int j = 0; j < 2; ++j)
846             {
847                 res(i, j) = m1(i, j) + m2(i, j);
848             }
849         }
850
851         return res;
852     }
853
854     inline Matrix2x2 operator-(const Matrix2x2& m)
855     {
856         Matrix2x2 res;
857         for (int i = 0; i < 2; ++i)
858         {
859             for (int j = 0; j < 2; ++j)
860             {
861                 res(i, j) = -m(i, j);
862             }
863         }
864
865         return res;
866     }
867
868     inline Matrix2x2 operator-(const Matrix2x2& m1, const Matrix2x2& m2)
869     {
870         Matrix2x2 res;
871         for (int i = 0; i < 2; ++i)

```

```

878     {
879         for (int j = 0; j < 2; ++j)
880         {
881             res(i, j) = m1(i, j) - m2(i, j);
882         }
883     }
884
885     return res;
886 }
887
888 inline Matrix2x2 operator*(const Matrix2x2& m, const float& k)
889 {
890     Matrix2x2 res;
891     for (int i = 0; i < 2; ++i)
892     {
893         for (int j = 0; j < 2; ++j)
894         {
895             res(i, j) = m(i, j) * k;
896         }
897     }
898
899     return res;
900 }
901
902 inline Matrix2x2 operator*(const float& k, const Matrix2x2& m)
903 {
904     Matrix2x2 res;
905     for (int i = 0; i < 2; ++i)
906     {
907         for (int j = 0; j < 2; ++j)
908         {
909             res(i, j) = k * m(i, j);
910         }
911     }
912
913     return res;
914 }
915
916 inline Matrix2x2 operator*(const Matrix2x2& m1, const Matrix2x2& m2)
917 {
918     Matrix2x2 res;
919     for (int i = 0; i < 4; ++i)
920     {
921         res(i, 0) =
922             (m1(i, 0) * m2(0, 0)) +
923             (m1(i, 1) * m2(1, 0));
924
925         res(i, 1) =
926             (m1(i, 0) * m2(0, 1)) +
927             (m1(i, 1) * m2(1, 1));
928
929         res(i, 2) =
930             (m1(i, 0) * m2(0, 2)) +
931             (m1(i, 1) * m2(1, 2));
932
933         res(i, 3) =
934             (m1(i, 0) * m2(0, 3)) +
935             (m1(i, 1) * m2(1, 3));
936     }
937
938     return res;
939 }
940
941 inline Vector2D operator*(const Matrix2x2& m, const Vector2D& v)
942 {
943     Vector2D res;
944
945     res.x = m(0, 0) * v.x + m(0, 1) * v.y;
946
947     res.y = m(1, 0) * v.x + m(1, 1) * v.y;
948
949     return res;
950 }
951
952 inline Vector2D operator*(const Vector2D& v, const Matrix2x2& m)
953 {
954     Vector2D res;
955
956     res.x = v.x * m(0, 0) + v.y * m(1, 0);
957
958     res.y = v.x * m(0, 1) + v.y * m(1, 1);
959
960     return res;
961 }
962
963 inline void SetToIdentity(Matrix2x2& m)

```

```

983     {
984         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
985
986         //1st row
987         m(0, 0) = 1.0f;
988         m(0, 1) = 0.0f;
989
990         //2nd row
991         m(1, 0) = 0.0f;
992         m(1, 1) = 1.0f;
993     }
994
995     inline bool Identity(const Matrix2x2& m)
996     {
997         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f
1000
1001         for (int i = 0; i < 2; ++i)
1002         {
1003             for (int j = 0; j < 2; ++j)
1004             {
1005                 if (i == j)
1006                 {
1007                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1008                         return false;
1009                 }
1010                 else
1011                 {
1012                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1013                         return false;
1014                 }
1015             }
1016         }
1017     }
1018 }
1019
1020 inline Matrix2x2 Transpose(const Matrix2x2& m)
1021 {
1022     //make the rows into cols
1023
1024     Matrix2x2 res;
1025
1026     //1st col = 1st row
1027     res(0, 0) = m(0, 0);
1028     res(1, 0) = m(0, 1);
1029
1030     //2nd col = 2nd row
1031     res(0, 1) = m(1, 0);
1032     res(1, 1) = m(1, 1);
1033
1034     return res;
1035 }
1036
1037 inline Matrix2x2 Scale(float x, float y)
1038 {
1039     //x 0
1040     //0 y
1041
1042     Matrix2x2 scale;
1043     scale(0, 0) = x;
1044     scale(1, 1) = y;
1045
1046     return scale;
1047 }
1048
1049 inline Matrix2x2 Scale(const Vector2D& scaleVector)
1050 {
1051     //x 0
1052     //0 y
1053
1054     Matrix2x2 scale;
1055     scale(0, 0) = scaleVector.x;
1056     scale(1, 1) = scaleVector.y;
1057
1058     return scale;
1059 }
1060
1061 inline Matrix2x2 Rotate(float angle)
1062 {
1063     //c      s
1064     //-s     c
1065     //c = cos(angle)
1066     //s = sin(angle)
1067
1068     float c = cos(angle * PI / 180.0f);
1069     float s = sin(angle * PI / 180.0f);

```



```

1079
1080     Matrix2x2 result;
1081
1082     //1st row
1083     result(0, 0) = c;
1084     result(0, 1) = s;
1085
1086     //2nd row
1087     result(1, 0) = -s;
1088     result(1, 1) = c;
1089
1090     return result;
1091 }
1092
1093 inline double Determinant(const Matrix2x2& m)
1094 {
1095     return (double)m(0, 0) * m(1, 1) - (double)m(0, 1) * m(1, 0);
1096 }
1097
1098 inline double Cofactor(const Matrix2x2& m, unsigned int row, unsigned int col)
1099 {
1100     //cij = ((-1)^(i + j)) * det of minor(i, j);
1101     double minor{ 0.0 };
1102
1103     if (row == 0 && col == 0)
1104         minor = m(1, 1);
1105     else if (row == 0 && col == 1)
1106         minor = m(1, 0);
1107     else if (row == 1 && col == 0)
1108         minor = m(0, 1);
1109     else if (row == 1 && col == 1)
1110         minor = m(0, 0);
1111
1112     return pow(-1, row + col) * minor;
1113 }
1114
1115 inline Matrix2x2 Adjoint(const Matrix2x2& m)
1116 {
1117     //Cofactor of each ijth position put into matrix cA.
1118     //Adjoint is the tranposed matrix of cA.
1119     Matrix2x2 cofactorMatrix;
1120     for (int i = 0; i < 2; ++i)
1121     {
1122         for (int j = 0; j < 2; ++j)
1123         {
1124             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
1125         }
1126     }
1127
1128     return Transpose(cofactorMatrix);
1129 }
1130
1131 inline Matrix2x2 Inverse(const Matrix2x2& m)
1132 {
1133     //Inverse of m = adjoint of m / det of m
1134     double det = Determinant(m);
1135     if (CompareDoubles(det, 0.0, EPSILON))
1136         return Matrix2x2();
1137
1138     return Adjoint(m) * (1.0f / static_cast<float>(det));
1139 }
1140
1141 #if defined(_DEBUG)
1142 inline void print(const Matrix2x2& m)
1143 {
1144     for (int i = 0; i < 2; ++i)
1145     {
1146         for (int j = 0; j < 2; ++j)
1147         {
1148             std::cout << m(i, j) << " ";
1149         }
1150
1151         std::cout << std::endl;
1152     }
1153 }
1154 #endif
1155
1156 //-----
1157
1158
1159
1160
1161
1162 //-----

```

```

1181     class Matrix3x3
1182     {
1183     public:
1184
1185         Matrix3x3();
1186
1187         Matrix3x3(float a[][3]);
1188
1189         Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3);
1190
1191         Matrix3x3(const Matrix2x2& m);
1192
1193         Matrix3x3(const Matrix4x4& m);
1194
1195         float* Data();
1196
1197         const float* Data() const;
1198
1199         const float& operator()(unsigned int row, unsigned int col) const;
1200
1201         float& operator()(unsigned int row, unsigned int col);
1202
1203         Vector3D GetRow(unsigned int row) const;
1204
1205         Vector3D GetCol(unsigned int col) const;
1206
1207         void SetRow(unsigned int row, Vector3D v);
1208
1209         void SetCol(unsigned int col, Vector3D v);
1210
1211         Matrix3x3& operator=(const Matrix2x2& m);
1212
1213         Matrix3x3& operator=(const Matrix4x4& m);
1214
1215         Matrix3x3& operator+=(const Matrix3x3& m);
1216
1217         Matrix3x3& operator-=(const Matrix3x3& m);
1218
1219         Matrix3x3& operator*=(float k);
1220
1221         Matrix3x3& operator*=(const Matrix3x3& m);
1222
1223     private:
1224
1225         float mMat[3][3];
1226     };
1227
1228     //-----
1229     inline Matrix3x3::Matrix3x3()
1230     {
1231         //1st row
1232         mMat[0][0] = 1.0f;
1233         mMat[0][1] = 0.0f;
1234         mMat[0][2] = 0.0f;
1235
1236         //2nd
1237         mMat[1][0] = 0.0f;
1238         mMat[1][1] = 1.0f;
1239         mMat[1][2] = 0.0f;
1240
1241         //3rd row
1242         mMat[2][0] = 0.0f;
1243         mMat[2][1] = 0.0f;
1244         mMat[2][2] = 1.0f;
1245     }
1246
1247     inline Matrix3x3::Matrix3x3(float a[][3])
1248     {
1249         //1st row
1250         mMat[0][0] = a[0][0];
1251         mMat[0][1] = a[0][1];
1252         mMat[0][2] = a[0][2];
1253
1254         //2nd
1255         mMat[1][0] = a[1][0];
1256         mMat[1][1] = a[1][1];
1257         mMat[1][2] = a[1][2];
1258
1259         //3rd row
1260         mMat[2][0] = a[2][0];
1261         mMat[2][1] = a[2][1];
1262         mMat[2][2] = a[2][2];
1263     }
1264
1265     inline Matrix3x3::Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3)
1266     {
1267         SetRow(0, r1);

```

```

1326         SetRow(1, r2);
1327         SetRow(2, r3);
1328     }
1329
1330     inline float* Matrix3x3::Data()
1331     {
1332         return mMat[0];
1333     }
1334
1335     inline const float* Matrix3x3::Data()const
1336     {
1337         return mMat[0];
1338     }
1339
1340     inline const float& Matrix3x3::operator()(unsigned int row, unsigned int col)const
1341     {
1342         if (row > 2 || col > 2)
1343         {
1344             return mMat[0][0];
1345         }
1346         else
1347         {
1348             return mMat[row][col];
1349         }
1350     }
1351
1352     inline float& Matrix3x3::operator()(unsigned int row, unsigned int col)
1353     {
1354         if (row > 2 || col > 2)
1355         {
1356             return mMat[0][0];
1357         }
1358         else
1359         {
1360             return mMat[row][col];
1361         }
1362     }
1363
1364     inline Vector3D Matrix3x3::GetRow(unsigned int row)const
1365     {
1366         if (row < 0 || row > 2)
1367             return Vector3D{ mMat[0][0], mMat[0][1], mMat[0][2] };
1368         else
1369             return Vector3D{ mMat[row][0], mMat[row][1], mMat[row][2] };
1370     }
1371
1372     inline Vector3D Matrix3x3::GetCol(unsigned int col)const
1373     {
1374         if (col < 0 || col > 2)
1375             return Vector3D{ mMat[0][0], mMat[1][0], mMat[2][0] };
1376         else
1377             return Vector3D{ mMat[0][col], mMat[1][col], mMat[2][col] };
1378     }
1379
1380     inline void Matrix3x3::SetRow(unsigned int row, Vector3D v)
1381     {
1382         if (row > 2)
1383         {
1384             mMat[0][0] = v.x;
1385             mMat[0][1] = v.y;
1386             mMat[0][2] = v.z;
1387         }
1388         else
1389         {
1390             mMat[row][0] = v.x;
1391             mMat[row][1] = v.y;
1392             mMat[row][2] = v.z;
1393         }
1394     }
1395
1396     inline void Matrix3x3::SetCol(unsigned int col, Vector3D v)
1397     {
1398         if (col > 2)
1399         {
1400             mMat[0][0] = v.x;
1401             mMat[1][0] = v.y;
1402             mMat[2][0] = v.z;
1403         }
1404         else
1405         {
1406             mMat[0][col] = v.x;
1407             mMat[1][col] = v.y;
1408             mMat[2][col] = v.z;
1409         }
1410     }
1411 }
1412

```

```

1413 inline Matrix3x3& Matrix3x3::operator+=(const Matrix3x3& m)
1414 {
1415     for (int i = 0; i < 3; ++i)
1416     {
1417         for (int j = 0; j < 3; ++j)
1418         {
1419             this->mMat[i][j] += m.mMat[i][j];
1420         }
1421     }
1422     return *this;
1423 }
1424
1425 inline Matrix3x3& Matrix3x3::operator-=(const Matrix3x3& m)
1426 {
1427     for (int i = 0; i < 3; ++i)
1428     {
1429         for (int j = 0; j < 3; ++j)
1430         {
1431             this->mMat[i][j] -= m.mMat[i][j];
1432         }
1433     }
1434     return *this;
1435 }
1436
1437 inline Matrix3x3& Matrix3x3::operator*=(float k)
1438 {
1439     for (int i = 0; i < 3; ++i)
1440     {
1441         for (int j = 0; j < 3; ++j)
1442         {
1443             this->mMat[i][j] *= k;
1444         }
1445     }
1446     return *this;
1447 }
1448
1449 inline Matrix3x3& Matrix3x3::operator*=(const Matrix3x3& m)
1450 {
1451     Matrix3x3 result;
1452     for (int i = 0; i < 3; ++i)
1453     {
1454         result.mMat[i][0] =
1455             (mMat[i][0] * m.mMat[0][0]) +
1456             (mMat[i][1] * m.mMat[1][0]) +
1457             (mMat[i][2] * m.mMat[2][0]);
1458         result.mMat[i][1] =
1459             (mMat[i][0] * m.mMat[0][1]) +
1460             (mMat[i][1] * m.mMat[1][1]) +
1461             (mMat[i][2] * m.mMat[2][1]);
1462         result.mMat[i][2] =
1463             (mMat[i][0] * m.mMat[0][2]) +
1464             (mMat[i][1] * m.mMat[1][2]) +
1465             (mMat[i][2] * m.mMat[2][2]);
1466     }
1467     for (int i = 0; i < 3; ++i)
1468     {
1469         for (int j = 0; j < 3; ++j)
1470         {
1471             mMat[i][j] = result.mMat[i][j];
1472         }
1473     }
1474     return *this;
1475 }
1476
1477 inline Matrix3x3 operator+(const Matrix3x3& m1, const Matrix3x3& m2)
1478 {
1479     Matrix3x3 result;
1480     for (int i = 0; i < 3; ++i)
1481     {
1482         for (int j = 0; j < 3; ++j)
1483         {
1484             result(i, j) = m1(i, j) + m2(i, j);
1485         }
1486     }
1487     return result;
1488 }
1489
1490 inline Matrix3x3 operator-(const Matrix3x3& m)

```

```

1504     {
1505         Matrix3x3 result;
1506         for (int i = 0; i < 3; ++i)
1507         {
1508             for (int j = 0; j < 3; ++j)
1509             {
1510                 result(i, j) = -m(i, j);
1511             }
1512         }
1513
1514         return result;
1515     }
1516
1517 inline Matrix3x3 operator-(const Matrix3x3& m1, const Matrix3x3& m2)
1518 {
1519     Matrix3x3 result;
1520     for (int i = 0; i < 3; ++i)
1521     {
1522         for (int j = 0; j < 3; ++j)
1523         {
1524             result(i, j) = m1(i, j) - m2(i, j);
1525         }
1526     }
1527
1528     return result;
1529 }
1530
1531 inline Matrix3x3 operator*(const Matrix3x3& m, const float& k)
1532 {
1533     Matrix3x3 result;
1534     for (int i = 0; i < 3; ++i)
1535     {
1536         for (int j = 0; j < 3; ++j)
1537         {
1538             result(i, j) = m(i, j) * k;
1539         }
1540     }
1541
1542     return result;
1543 }
1544
1545 inline Matrix3x3 operator*(const float& k, const Matrix3x3& m)
1546 {
1547     Matrix3x3 result;
1548     for (int i = 0; i < 3; ++i)
1549     {
1550         for (int j = 0; j < 3; ++j)
1551         {
1552             result(i, j) = k * m(i, j);
1553         }
1554     }
1555
1556     return result;
1557 }
1558
1559 inline Matrix3x3 operator*(const Matrix3x3& m1, const Matrix3x3& m2)
1560 {
1561     Matrix3x3 result;
1562     for (int i = 0; i < 4; ++i)
1563     {
1564         result(i, 0) =
1565             (m1(i, 0) * m2(0, 0)) +
1566             (m1(i, 1) * m2(1, 0)) +
1567             (m1(i, 2) * m2(2, 0));
1568
1569         result(i, 1) =
1570             (m1(i, 0) * m2(0, 1)) +
1571             (m1(i, 1) * m2(1, 1)) +
1572             (m1(i, 2) * m2(2, 1));
1573
1574         result(i, 2) =
1575             (m1(i, 0) * m2(0, 2)) +
1576             (m1(i, 1) * m2(1, 2)) +
1577             (m1(i, 2) * m2(2, 2));
1578
1579         result(i, 3) =
1580             (m1(i, 0) * m2(0, 3)) +
1581             (m1(i, 1) * m2(1, 3)) +
1582             (m1(i, 2) * m2(2, 3));
1583     }
1584
1585     return result;
1586 }
1587
1588 inline Vector3D operator*(const Matrix3x3& m, const Vector3D& v)
1589 {

```

```

1605         Vector3D result;
1606
1607         result.x = m(0, 0) * v.x + m(0, 1) * v.y + m(0, 2) * v.z;
1608
1609         result.y = m(1, 0) * v.x + m(1, 1) * v.y + m(1, 2) * v.z;
1610
1611         result.z = m(2, 0) * v.x + m(2, 1) * v.y + m(2, 2) * v.z;
1612
1613         return result;
1614     }
1615
1620     inline Vector3D operator*(const Vector3D& v, const Matrix3x3& m)
1621     {
1622         Vector3D result;
1623
1624         result.x = v.x * m(0, 0) + v.y * m(1, 0) + v.z * m(2, 0);
1625
1626         result.y = v.x * m(0, 1) + v.y * m(1, 1) + v.z * m(2, 1);
1627
1628         result.z = v.x * m(0, 2) + v.y * m(1, 2) + v.z * m(2, 2);
1629
1630         return result;
1631     }
1632
1635     inline void SetToIdentity(Matrix3x3& m)
1636     {
1637         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
1638
1639         //1st row
1640         m(0, 0) = 1.0f;
1641         m(0, 1) = 0.0f;
1642         m(0, 2) = 0.0f;
1643
1644         //2nd row
1645         m(1, 0) = 0.0f;
1646         m(1, 1) = 1.0f;
1647         m(1, 2) = 0.0f;
1648
1649         //3rd row
1650         m(2, 0) = 0.0f;
1651         m(2, 1) = 0.0f;
1652         m(2, 2) = 1.0f;
1653     }
1654
1657     inline bool Identity(const Matrix3x3& m)
1658     {
1659         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f
1660
1661         for (int i = 0; i < 3; ++i)
1662         {
1663             for (int j = 0; j < 3; ++j)
1664             {
1665                 if (i == j)
1666                 {
1667                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
1668                         return false;
1669                 }
1670                 else
1671                 {
1672                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
1673                         return false;
1674                 }
1675             }
1676         }
1677     }
1678
1679
1680
1683     inline Matrix3x3 Transpose(const Matrix3x3& m)
1684     {
1685         //make the rows into cols
1686
1687         Matrix3x3 result;
1688
1689         //1st col = 1st row
1690         result(0, 0) = m(0, 0);
1691         result(1, 0) = m(0, 1);
1692         result(2, 0) = m(0, 2);
1693
1694         //2nd col = 2nd row
1695         result(0, 1) = m(1, 0);
1696         result(1, 1) = m(1, 1);
1697         result(2, 1) = m(1, 2);
1698
1699         //3rd col = 3rd row
1700         result(0, 2) = m(2, 0);

```

```

1701         result(1, 2) = m(2, 1);
1702         result(2, 2) = m(2, 2);
1703
1704         return result;
1705     }
1706
1707     inline Matrix3x3 Scale(float x, float y, float z)
1708     {
1709         //x 0 0
1710         //0 y 0
1711         //0 0 z
1712
1713         Matrix3x3 scale;
1714         scale(0, 0) = x;
1715         scale(1, 1) = y;
1716         scale(2, 2) = z;
1717
1718         return scale;
1719     }
1720
1721     inline Matrix3x3 Scale(const Vector3D& scaleVector)
1722     {
1723         //x 0 0
1724         //0 y 0
1725         //0 0 z
1726
1727         Matrix3x3 scale;
1728         scale(0, 0) = scaleVector.x;
1729         scale(1, 1) = scaleVector.y;
1730         scale(2, 2) = scaleVector.z;
1731
1732         return scale;
1733     }
1734
1735     inline Matrix3x3 Rotate(float angle, float x, float y, float z)
1736     {
1737         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy
1738         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx
1739         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2
1740         //c = cos(angle)
1741         //s = sin(angle)
1742
1743         Vector3D axis{ x, y, z };
1744         axis = Normalize(axis);
1745         x = axis.x;
1746         y = axis.y;
1747         z = axis.z;
1748
1749         float c = cos(angle * PI / 180.0f);
1750         float s = sin(angle * PI / 180.0f);
1751         float oneMinusC = 1.0f - c;
1752
1753         Matrix3x3 result;
1754
1755         //1st row
1756         result(0, 0) = c + (oneMinusC * (x * x));
1757         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
1758         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
1759
1760         //2nd row
1761         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
1762         result(1, 1) = c + (oneMinusC * (y * y));
1763         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
1764
1765         //3rd row
1766         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
1767         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
1768         result(2, 2) = c + (oneMinusC * (z * z));
1769
1770         return result;
1771     }
1772
1773     inline Matrix3x3 Rotate(float angle, const Vector3D& axis)
1774     {
1775         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy
1776         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx
1777         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2
1778         //c = cos(angle)
1779         //s = sin(angle)
1780
1781         Vector3D nAxis(Normalize(axis));
1782         float x = nAxis.x;
1783         float y = nAxis.y;
1784         float z = nAxis.z;
1785
1786         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy
1787         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx
1788         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2
1789         //c = cos(angle)
1790         //s = sin(angle)
1791
1792         Vector3D nAxis(Normalize(axis));
1793         float x = nAxis.x;
1794         float y = nAxis.y;
1795         float z = nAxis.z;

```

```

1796     float c = cos(angle * PI / 180.0f);
1797     float s = sin(angle * PI / 180.0f);
1798     float oneMinusC = 1.0f - c;
1799
1800     Matrix3x3 result;
1801
1802     //1st row
1803     result(0, 0) = c + (oneMinusC * (x * x));
1804     result(0, 1) = (oneMinusC * (x * y)) + (s * z);
1805     result(0, 2) = (oneMinusC * (x * z)) - (s * y);
1806
1807     //2nd row
1808     result(1, 0) = (oneMinusC * (x * y)) - (s * z);
1809     result(1, 1) = c + (oneMinusC * (y * y));
1810     result(1, 2) = (oneMinusC * (y * z)) + (s * x);
1811
1812     //3rd row
1813     result(2, 0) = (oneMinusC * (x * z)) + (s * y);
1814     result(2, 1) = (oneMinusC * (y * z)) - (s * x);
1815     result(2, 2) = c + (oneMinusC * (z * z));
1816
1817     return result;
1818 }
1819
1822 inline double Determinant(const Matrix3x3& m)
1823 {
1824     //m00m11m22 - m00m12m21
1825     double c1 = (double)m(0, 0) * m(1, 1) * m(2, 2) - (double)m(0, 0) * m(1, 2) * m(2, 1);
1826
1827     //m01m12m20 - m01m10m22
1828     double c2 = (double)m(0, 1) * m(1, 2) * m(2, 0) - (double)m(0, 1) * m(1, 0) * m(2, 2);
1829
1830     //m02m10m21 - m02m11m20
1831     double c3 = (double)m(0, 2) * m(1, 0) * m(2, 1) - (double)m(0, 2) * m(1, 1) * m(2, 0);
1832
1833     return c1 + c2 + c3;
1834 }
1835
1838 inline double Cofactor(const Matrix3x3& m, unsigned int row, unsigned int col)
1839 {
1840     //cij = ((-1)^(i + j)) * det of minor(i, j);
1841     Matrix2x2 minor;
1842     int r{ 0 };
1843     int c{ 0 };
1844
1845     //minor(i, j)
1846     for (int i = 0; i < 3; ++i)
1847     {
1848         if (i == row)
1849             continue;
1850
1851         for (int j = 0; j < 3; ++j)
1852         {
1853             if (j == col)
1854                 continue;
1855
1856             minor(r, c) = m(i, j);
1857             ++c;
1858         }
1859         c = 0;
1860         ++r;
1861     }
1862
1863     return pow(-1, row + col) * Determinant(minor);
1864 }
1865
1866 inline Matrix3x3 Adjoint(const Matrix3x3& m)
1867 {
1868     //Cofactor of each ijth position put into matrix cA.
1869     //Adjoint is the tranposed matrix of cA.
1870     Matrix3x3 cofactorMatrix;
1871     for (int i = 0; i < 3; ++i)
1872     {
1873         for (int j = 0; j < 3; ++j)
1874         {
1875             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
1876         }
1877     }
1878
1879     return Transpose(cofactorMatrix);
1880 }
1881
1882 inline Matrix3x3 Inverse(const Matrix3x3& m)
1883 {
1884     //Inverse of m = adjoint of m / det of m
1885     double det = Determinant(m);

```



```

1893         if (CompareDoubles(det, 0.0, EPSILON))
1894             return Matrix3x3();
1895
1896         return Adjoint(m) * (1.0f / static_cast<float>(det));
1897     }
1898
1899
1900 #if defined(_DEBUG)
1901     inline void print(const Matrix3x3& m)
1902     {
1903         for (int i = 0; i < 3; ++i)
1904         {
1905             for (int j = 0; j < 3; ++j)
1906             {
1907                 std::cout << m(i, j) << "\t";
1908             }
1909
1910             std::cout << std::endl;
1911         }
1912     }
1913 #endif
1914
1915
1916 //-----
1917
1918
1919 //-----
1920
1921 class Matrix4x4
1922 {
1923 public:
1924     Matrix4x4();
1925
1926     Matrix4x4(float a[][4]);
1927
1928     Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const Vector4D& r4);
1929
1930     Matrix4x4(const Matrix2x2& m);
1931
1932     Matrix4x4(const Matrix3x3& m);
1933
1934     Matrix4x4& operator=(const Matrix2x2& m);
1935
1936     Matrix4x4& operator=(const Matrix3x3& m);
1937
1938     float* Data();
1939
1940     const float* Data() const;
1941
1942     const float& operator()(unsigned int row, unsigned int col) const;
1943
1944     float& operator()(unsigned int row, unsigned int col);
1945
1946     Vector4D GetRow(unsigned int row) const;
1947
1948     Vector4D GetCol(unsigned int col) const;
1949
1950     void SetRow(unsigned int row, Vector4D v);
1951
1952     void SetCol(unsigned int col, Vector4D v);
1953
1954     Matrix4x4& operator+=(const Matrix4x4& m);
1955
1956     Matrix4x4& operator-=(const Matrix4x4& m);
1957
1958     Matrix4x4& operator*=(float k);
1959
1960     Matrix4x4& operator*=(const Matrix4x4& m);
1961
1962 private:
1963     float mMat[4][4];
1964 };
1965
1966 //-----
1967 inline Matrix4x4::Matrix4x4()
1968 {
1969     //1st row
1970     mMat[0][0] = 1.0f;
1971     mMat[0][1] = 0.0f;
1972     mMat[0][2] = 0.0f;
1973     mMat[0][3] = 0.0f;
1974
1975     //2nd
1976     mMat[1][0] = 0.0f;

```

```

2051         mMat[1][1] = 1.0f;
2052         mMat[1][2] = 0.0f;
2053         mMat[1][3] = 0.0f;
2054
2055         //3rd row
2056         mMat[2][0] = 0.0f;
2057         mMat[2][1] = 0.0f;
2058         mMat[2][2] = 1.0f;
2059         mMat[2][3] = 0.0f;
2060
2061         //4th row
2062         mMat[3][0] = 0.0f;
2063         mMat[3][1] = 0.0f;
2064         mMat[3][2] = 0.0f;
2065         mMat[3][3] = 1.0f;
2066     }
2067
2068
2069
2070     inline Matrix4x4::Matrix4x4(float a[][4])
2071     {
2072         //1st row
2073         mMat[0][0] = a[0][0];
2074         mMat[0][1] = a[0][1];
2075         mMat[0][2] = a[0][2];
2076         mMat[0][3] = a[0][3];
2077
2078         //2nd
2079         mMat[1][0] = a[1][0];
2080         mMat[1][1] = a[1][1];
2081         mMat[1][2] = a[1][2];
2082         mMat[1][3] = a[1][3];
2083
2084         //3rd row
2085         mMat[2][0] = a[2][0];
2086         mMat[2][1] = a[2][1];
2087         mMat[2][2] = a[2][2];
2088         mMat[2][3] = a[2][3];
2089
2090         //4th row
2091         mMat[3][0] = a[3][0];
2092         mMat[3][1] = a[3][1];
2093         mMat[3][2] = a[3][2];
2094         mMat[3][3] = a[3][3];
2095     }
2096
2097     inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
Vector4D& r4)
2098     {
2099         SetRow(0, r1);
2100         SetRow(1, r2);
2101         SetRow(2, r3);
2102         SetRow(3, r4);
2103     }
2104
2105     inline float* Matrix4x4::Data()
2106     {
2107         return mMat[0];
2108     }
2109
2110     inline const float* Matrix4x4::Data()const
2111     {
2112         return mMat[0];
2113     }
2114
2115     inline const float& Matrix4x4::operator()(unsigned int row, unsigned int col)const
2116     {
2117         if (row > 3 || col > 3)
2118         {
2119             return mMat[0][0];
2120         }
2121         else
2122         {
2123             return mMat[row][col];
2124         }
2125     }
2126
2127     inline float& Matrix4x4::operator()(unsigned int row, unsigned int col)
2128     {
2129         if (row > 3 || col > 3)
2130         {
2131             return mMat[0][0];
2132         }
2133         else
2134         {
2135             return mMat[row][col];
2136         }

```

```

2137     }
2138
2139     inline Vector4D Matrix4x4::GetRow(unsigned int row) const
2140     {
2141         if (row < 0 || row > 3)
2142             return Vector4D{ mMat[0][0], mMat[0][1], mMat[0][2], mMat[0][3] };
2143         else
2144             return Vector4D{ mMat[row][0], mMat[row][1], mMat[row][2], mMat[row][3] };
2145     }
2146
2147     inline Vector4D Matrix4x4::GetCol(unsigned int col) const
2148     {
2149         if (col < 0 || col > 3)
2150             return Vector4D{ mMat[0][0], mMat[1][0], mMat[2][0], mMat[3][0] };
2151         else
2152             return Vector4D{ mMat[0][col], mMat[1][col], mMat[2][col], mMat[3][col] };
2153     }
2154
2155     inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
2156     {
2157         if (row > 3)
2158         {
2159             mMat[0][0] = v.x;
2160             mMat[0][1] = v.y;
2161             mMat[0][2] = v.z;
2162             mMat[0][3] = v.w;
2163         }
2164         else
2165         {
2166             mMat[row][0] = v.x;
2167             mMat[row][1] = v.y;
2168             mMat[row][2] = v.z;
2169             mMat[row][3] = v.w;
2170         }
2171     }
2172
2173     inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
2174     {
2175         if (col > 3)
2176         {
2177             mMat[0][0] = v.x;
2178             mMat[1][0] = v.y;
2179             mMat[2][0] = v.z;
2180             mMat[3][0] = v.w;
2181         }
2182         else
2183         {
2184             mMat[0][col] = v.x;
2185             mMat[1][col] = v.y;
2186             mMat[2][col] = v.z;
2187             mMat[3][col] = v.w;
2188         }
2189     }
2190
2191     inline Matrix4x4& Matrix4x4::operator+=(const Matrix4x4& m)
2192     {
2193         for (int i = 0; i < 4; ++i)
2194         {
2195             for (int j = 0; j < 4; ++j)
2196             {
2197                 this->mMat[i][j] += m.mMat[i][j];
2198             }
2199         }
2200
2201         return *this;
2202     }
2203
2204     inline Matrix4x4& Matrix4x4::operator-=(const Matrix4x4& m)
2205     {
2206         for (int i = 0; i < 4; ++i)
2207         {
2208             for (int j = 0; j < 4; ++j)
2209             {
2210                 this->mMat[i][j] -= m.mMat[i][j];
2211             }
2212         }
2213
2214         return *this;
2215     }
2216
2217     inline Matrix4x4& Matrix4x4::operator*=(float k)
2218     {
2219         for (int i = 0; i < 4; ++i)
2220         {
2221             for (int j = 0; j < 4; ++j)
2222             {
2223

```

```

2224         this->mMat[i][j] *= k;
2225     }
2226 }
2227
2228     return *this;
2229 }
2230
2231 inline Matrix4x4& Matrix4x4::operator*=(const Matrix4x4& m)
2232 {
2233     Matrix4x4 result;
2234
2235     for (int i = 0; i < 4; ++i)
2236     {
2237         result.mMat[i][0] =
2238             (mMat[i][0] * m.mMat[0][0]) +
2239             (mMat[i][1] * m.mMat[1][0]) +
2240             (mMat[i][2] * m.mMat[2][0]) +
2241             (mMat[i][3] * m.mMat[3][0]);
2242
2243         result.mMat[i][1] =
2244             (mMat[i][0] * m.mMat[0][1]) +
2245             (mMat[i][1] * m.mMat[1][1]) +
2246             (mMat[i][2] * m.mMat[2][1]) +
2247             (mMat[i][3] * m.mMat[3][1]);
2248
2249         result.mMat[i][2] =
2250             (mMat[i][0] * m.mMat[0][2]) +
2251             (mMat[i][1] * m.mMat[1][2]) +
2252             (mMat[i][2] * m.mMat[2][2]) +
2253             (mMat[i][3] * m.mMat[3][2]);
2254
2255         result.mMat[i][3] =
2256             (mMat[i][0] * m.mMat[0][3]) +
2257             (mMat[i][1] * m.mMat[1][3]) +
2258             (mMat[i][2] * m.mMat[2][3]) +
2259             (mMat[i][3] * m.mMat[3][3]);
2260     }
2261
2262     for (int i = 0; i < 4; ++i)
2263     {
2264         for (int j = 0; j < 4; ++j)
2265         {
2266             mMat[i][j] = result.mMat[i][j];
2267         }
2268     }
2269
2270     return *this;
2271 }
2272
2273 inline Matrix4x4 operator+(const Matrix4x4& m1, const Matrix4x4& m2)
2274 {
2275     Matrix4x4 result;
2276     for (int i = 0; i < 4; ++i)
2277     {
2278         for (int j = 0; j < 4; ++j)
2279         {
2280             result(i, j) = m1(i, j) + m2(i, j);
2281         }
2282     }
2283
2284     return result;
2285 }
2286
2287 inline Matrix4x4 operator-(const Matrix4x4& m)
2288 {
2289     Matrix4x4 result;
2290     for (int i = 0; i < 4; ++i)
2291     {
2292         for (int j = 0; j < 4; ++j)
2293         {
2294             result(i, j) = -m(i, j);
2295         }
2296     }
2297
2298     return result;
2299 }
2300
2301 inline Matrix4x4 operator-(const Matrix4x4& m1, const Matrix4x4& m2)
2302 {
2303     Matrix4x4 result;
2304     for (int i = 0; i < 4; ++i)
2305     {
2306         for (int j = 0; j < 4; ++j)
2307         {
2308             result(i, j) = m1(i, j) - m2(i, j);
2309         }
2310     }
2311 }

```

```

2317
2318     return result;
2319 }
2320
2321 inline Matrix4x4 operator*(const Matrix4x4& m, const float& k)
2322 {
2323     Matrix4x4 result;
2324     for (int i = 0; i < 4; ++i)
2325     {
2326         for (int j = 0; j < 4; ++j)
2327         {
2328             result(i, j) = m(i, j) * k;
2329         }
2330     }
2331     return result;
2332 }
2333
2334 inline Matrix4x4 operator*(const float& k, const Matrix4x4& m)
2335 {
2336     Matrix4x4 result;
2337     for (int i = 0; i < 4; ++i)
2338     {
2339         for (int j = 0; j < 4; ++j)
2340         {
2341             result(i, j) = k * m(i, j);
2342         }
2343     }
2344     return result;
2345 }
2346
2347 inline Matrix4x4 operator*(const Matrix4x4& m1, const Matrix4x4& m2)
2348 {
2349     Matrix4x4 result;
2350     for (int i = 0; i < 4; ++i)
2351     {
2352         result(i, 0) =
2353             (m1(i, 0) * m2(0, 0)) +
2354             (m1(i, 1) * m2(1, 0)) +
2355             (m1(i, 2) * m2(2, 0)) +
2356             (m1(i, 3) * m2(3, 0));
2357
2358         result(i, 1) =
2359             (m1(i, 0) * m2(0, 1)) +
2360             (m1(i, 1) * m2(1, 1)) +
2361             (m1(i, 2) * m2(2, 1)) +
2362             (m1(i, 3) * m2(3, 1));
2363
2364         result(i, 2) =
2365             (m1(i, 0) * m2(0, 2)) +
2366             (m1(i, 1) * m2(1, 2)) +
2367             (m1(i, 2) * m2(2, 2)) +
2368             (m1(i, 3) * m2(3, 2));
2369
2370         result(i, 3) =
2371             (m1(i, 0) * m2(0, 3)) +
2372             (m1(i, 1) * m2(1, 3)) +
2373             (m1(i, 2) * m2(2, 3)) +
2374             (m1(i, 3) * m2(3, 3));
2375     }
2376     return result;
2377 }
2378
2379 inline Vector4D operator*(const Matrix4x4& m, const Vector4D& v)
2380 {
2381     Vector4D result;
2382
2383     result.x = m(0, 0) * v.x + m(0, 1) * v.y + m(0, 2) * v.z + m(0, 3) * v.w;
2384
2385     result.y = m(1, 0) * v.x + m(1, 1) * v.y + m(1, 2) * v.z + m(1, 3) * v.w;
2386
2387     result.z = m(2, 0) * v.x + m(2, 1) * v.y + m(2, 2) * v.z + m(2, 3) * v.w;
2388
2389     result.w = m(3, 0) * v.x + m(3, 1) * v.y + m(3, 2) * v.z + m(3, 3) * v.w;
2390
2391     return result;
2392 }
2393
2394 inline Vector4D operator*(const Vector4D& v, const Matrix4x4& m)
2395 {
2396     Vector4D result;
2397
2398     result.x = v.x * m(0, 0) + v.y * m(1, 0) + v.z * m(2, 0) + v.w * m(3, 0);
2399

```

```

2420         result.y = v.x * m(0, 1) + v.y * m(1, 1) + v.z * m(2, 1) + v.w * m(3, 1);
2421
2422         result.z = v.x * m(0, 2) + v.y * m(1, 2) + v.z * m(2, 2) + v.w * m(3, 2);
2423
2424         result.w = v.x * m(0, 3) + v.y * m(1, 3) + v.z * m(2, 3) + v.w * m(3, 3);
2425
2426         return result;
2427     }
2428
2429     inline void SetToIdentity(Matrix4x4& m)
2430     {
2431         //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f
2432
2433         //1st row
2434         m(0, 0) = 1.0f;
2435         m(0, 1) = 0.0f;
2436         m(0, 2) = 0.0f;
2437         m(0, 3) = 0.0f;
2438
2439         //2nd row
2440         m(1, 0) = 0.0f;
2441         m(1, 1) = 1.0f;
2442         m(1, 2) = 0.0f;
2443         m(1, 3) = 0.0f;
2444
2445         //3rd row
2446         m(2, 0) = 0.0f;
2447         m(2, 1) = 0.0f;
2448         m(2, 2) = 1.0f;
2449         m(2, 3) = 0.0f;
2450
2451         //4th row
2452         m(3, 0) = 0.0f;
2453         m(3, 1) = 0.0f;
2454         m(3, 2) = 0.0f;
2455         m(3, 3) = 1.0f;
2456     }
2457
2458     inline bool Identity(const Matrix4x4& m)
2459     {
2460         //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
2461         0.0f
2462
2463         for (int i = 0; i < 4; ++i)
2464         {
2465             for (int j = 0; j < 4; ++j)
2466             {
2467                 if (i == j)
2468                 {
2469                     if (!CompareFloats(m(i, j), 1.0f, EPSILON))
2470                         return false;
2471                 }
2472                 else
2473                 {
2474                     if (!CompareFloats(m(i, j), 0.0f, EPSILON))
2475                         return false;
2476                 }
2477             }
2478         }
2479     }
2480
2481     inline Matrix4x4 Transpose(const Matrix4x4& m)
2482     {
2483         //make the rows into cols
2484
2485         Matrix4x4 result;
2486
2487         //1st col = 1st row
2488         result(0, 0) = m(0, 0);
2489         result(1, 0) = m(0, 1);
2490         result(2, 0) = m(0, 2);
2491         result(3, 0) = m(0, 3);
2492
2493         //2nd col = 2nd row
2494         result(0, 1) = m(1, 0);
2495         result(1, 1) = m(1, 1);
2496         result(2, 1) = m(1, 2);
2497         result(3, 1) = m(1, 3);
2498
2499         //3rd col = 3rd row
2500         result(0, 2) = m(2, 0);
2501         result(1, 2) = m(2, 1);
2502         result(2, 2) = m(2, 2);
2503         result(3, 2) = m(2, 3);
2504
2505         //4th col = 4th row
2506         result(0, 3) = m(3, 0);
2507         result(1, 3) = m(3, 1);
2508         result(2, 3) = m(3, 2);
2509         result(3, 3) = m(3, 3);
2510     }
2511

```

```

2512         //4th col = 4th row
2513         result(0, 3) = m(3, 0);
2514         result(1, 3) = m(3, 1);
2515         result(2, 3) = m(3, 2);
2516         result(3, 3) = m(3, 3);
2517
2518         return result;
2519     }
2520
2521     inline Matrix4x4 Translate(float x, float y, float z)
2522     {
2523         //1 0 0 0
2524         //0 1 0 0
2525         //0 0 1 0
2526         //x y z 1
2527
2528         Matrix4x4 translate;
2529         translate(3, 0) = x;
2530         translate(3, 1) = y;
2531         translate(3, 2) = z;
2532
2533         return translate;
2534     }
2535
2536     inline Matrix4x4 Translate(const Vector3D& translateVector)
2537     {
2538         //1 0 0 0
2539         //0 1 0 0
2540         //0 0 1 0
2541         //x y z 1
2542
2543         Matrix4x4 translate;
2544         translate(3, 0) = translateVector.x;
2545         translate(3, 1) = translateVector.y;
2546         translate(3, 2) = translateVector.z;
2547
2548         return translate;
2549     }
2550
2551     inline Matrix4x4 Scale4x4(float x, float y, float z)
2552     {
2553         //x 0 0 0
2554         //0 y 0 0
2555         //0 0 z 0
2556         //0 0 0 1
2557
2558         Matrix4x4 scale;
2559         scale(0, 0) = x;
2560         scale(1, 1) = y;
2561         scale(2, 2) = z;
2562
2563         return scale;
2564     }
2565
2566     inline Matrix4x4 Scale4x4(const Vector3D& scaleVector)
2567     {
2568         //x 0 0 0
2569         //0 y 0 0
2570         //0 0 z 0
2571         //0 0 0 1
2572
2573         Matrix4x4 scale;
2574         scale(0, 0) = scaleVector.x;
2575         scale(1, 1) = scaleVector.y;
2576         scale(2, 2) = scaleVector.z;
2577
2578         return scale;
2579     }
2580
2581     inline Matrix4x4 Rotate4x4(float angle, float x, float y, float z)
2582     {
2583         //c + (1 - c)x^2    (1 - c)xy + sz    (1 - c)xz - sy    0
2584         //(1 - c)xy - sz    c + (1 - c)y^2    (1 - c)yz + sx    0
2585         //(1 - c)xz + sy    (1 - c)yz - sx    c + (1 - c)z^2    0
2586         //0                0                0                1
2587         //c = cos(angle)
2588         //s = sin(angle)
2589
2590         Vector3D axis{ x, y, z };
2591
2592         axis = Normalize(axis);
2593
2594         x = axis.x;
2595         y = axis.y;
2596         z = axis.z;
2597
2598         float c = cos(angle * PI / 180.0f);

```

```

2609         float s = sin(angle * PI / 180.0f);
2610         float oneMinusC = 1 - c;
2611
2612         Matrix4x4 result;
2613
2614         //1st row
2615         result(0, 0) = c + (oneMinusC * (x * x));
2616         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2617         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2618
2619         //2nd row
2620         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2621         result(1, 1) = c + (oneMinusC * (y * y));
2622         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2623
2624         //3rd row
2625         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2626         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2627         result(2, 2) = c + (oneMinusC * (z * z));
2628
2629         return result;
2630     }
2631
2632     inline Matrix4x4 Rotate4x4(float angle, const Vector3D& axis)
2633     {
2634         //c + (1 - c)x^2      (1 - c)xy + sz      (1 - c)xz - sy      0
2635         // (1 - c)xy - sz      c + (1 - c)y^2      (1 - c)yz + sx      0
2636         // (1 - c)xz + sy      (1 - c)yz - sx      c + (1 - c)z^2      0
2637         //0                      0                      0                      1
2638         //c = cos(angle)
2639         //s = sin(angle)
2640
2641         Vector3D nAxis(Normalize(axis));
2642
2643         float x = nAxis.x;
2644         float y = nAxis.y;
2645         float z = nAxis.z;
2646
2647         float c = cos(angle * PI / 180.0f);
2648         float s = sin(angle * PI / 180.0f);
2649         float oneMinusC = 1 - c;
2650
2651         Matrix4x4 result;
2652
2653         //1st row
2654         result(0, 0) = c + (oneMinusC * (x * x));
2655         result(0, 1) = (oneMinusC * (x * y)) + (s * z);
2656         result(0, 2) = (oneMinusC * (x * z)) - (s * y);
2657
2658         //2nd row
2659         result(1, 0) = (oneMinusC * (x * y)) - (s * z);
2660         result(1, 1) = c + (oneMinusC * (y * y));
2661         result(1, 2) = (oneMinusC * (y * z)) + (s * x);
2662
2663         //3rd row
2664         result(2, 0) = (oneMinusC * (x * z)) + (s * y);
2665         result(2, 1) = (oneMinusC * (y * z)) - (s * x);
2666         result(2, 2) = c + (oneMinusC * (z * z));
2667
2668         return result;
2669     }
2670
2671     inline double Determinant(const Matrix4x4& m)
2672     {
2673         //m00m11(m22m33 - m23m32)
2674         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)
2675         * m(3, 2);
2676
2677         //m00m12(m23m31 - m21m33)
2678         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)
2679         * m(3, 3);
2680
2681         //m00m13(m21m32 - m22m31)
2682         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)
2683         * m(3, 1);
2684
2685         //m01m10(m22m33 - m23m32)
2686         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)
2687         * m(3, 2);
2688
2689         //m01m12(m23m30 - m20m33)
2690         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)
2691         * m(3, 3);
2692
2693         //m01m13(m20m32 - m22m30)
2694         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)
2695         * m(3, 0);

```



```

2694
2695 //m02m10(m21m33 - m23m31)
2696 double c7 = (double)m(0, 2) * m(1, 0) * m(2, 1) * m(3, 3) - (double)m(0, 2) * m(1, 0) * m(2, 3)
    * m(3, 1);
2697
2698 //m02m11(m23m30 - m20m33)
2699 double c8 = (double)m(0, 2) * m(1, 1) * m(2, 3) * m(3, 0) - (double)m(0, 2) * m(1, 1) * m(2, 0)
    * m(3, 3);
2700
2701 //m02m13(m20m31 - m21m30)
2702 double c9 = (double)m(0, 2) * m(1, 3) * m(2, 0) * m(3, 1) - (double)m(0, 2) * m(1, 3) * m(2, 1)
    * m(3, 0);
2703
2704 //m03m10(m21m32 - m22m21)
2705 double c10 = (double)m(0, 3) * m(1, 0) * m(2, 1) * m(3, 2) - (double)m(0, 3) * m(1, 0) * m(2,
    2) * m(3, 1);
2706
2707 //m03m11(m22m30 - m20m32)
2708 double c11 = (double)m(0, 3) * m(1, 1) * m(2, 2) * m(3, 0) - (double)m(0, 3) * m(1, 1) * m(2,
    0) * m(3, 2);
2709
2710 //m03m12(m20m31 - m21m30)
2711 double c12 = (double)m(0, 3) * m(1, 2) * m(2, 0) * m(3, 1) - (double)m(0, 3) * m(1, 2) * m(2,
    1) * m(3, 0);
2712
2713 return (c1 + c2 + c3) - (c4 + c5 + c6) + (c7 + c8 + c9) - (c10 + c11 + c12);
2714 }
2715
2716 inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
2717 {
2718     //cij = (-1)^i + j * det of minor(i, j);
2719     Matrix3x3 minor;
2720     int r{ 0 };
2721     int c{ 0 };
2722
2723     //minor(i, j)
2724     for (int i = 0; i < 4; ++i)
2725     {
2726         if (i == row)
2727             continue;
2728
2729         for (int j = 0; j < 4; ++j)
2730         {
2731             if (j == col)
2732                 continue;
2733
2734             minor(r, c) = m(i, j);
2735             ++c;
2736         }
2737         c = 0;
2738         ++r;
2739     }
2740
2741     return pow(-1, row + col) * Determinant(minor);
2742 }
2743
2744 inline Matrix4x4 Adjoint(const Matrix4x4& m)
2745 {
2746     //Cofactor of each ijth position put into matrix cA.
2747     //Adjoint is the tranposed matrix of cA.
2748     Matrix4x4 cofactorMatrix;
2749     for (int i = 0; i < 4; ++i)
2750     {
2751         for (int j = 0; j < 4; ++j)
2752         {
2753             cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
2754         }
2755     }
2756
2757     return Transpose(cofactorMatrix);
2758 }
2759
2760 inline Matrix4x4 Inverse(const Matrix4x4& m)
2761 {
2762     //Inverse of m = adjoint of m / det of m
2763     double det = Determinant(m);
2764     if (CompareDoubles(det, 0.0, EPSILON))
2765         return Matrix4x4();
2766
2767     return Adjoint(m) * (1.0f / static_cast<float>(det));
2768 }
2769
2770 #if defined(_DEBUG)
2771 inline void print(const Matrix4x4& m)

```

```

2783     {
2784         for (int i = 0; i < 4; ++i)
2785         {
2786             for (int j = 0; j < 4; ++j)
2787             {
2788                 std::cout << m(i, j) << " ";
2789             }
2790             std::cout << std::endl;
2791         }
2792     }
2793 }
2794 #endif
2795
2796 //-----
2797
2798
2799
2800
2801 //-----
2802 //QUATERNION
2803
2804 struct Quaternion
2805 {
2806     float scalar = 1.0f;
2807     Vector3D vector;
2808 };
2809
2810 inline void operator+=(Quaternion& q1, const Quaternion& q2)
2811 {
2812     q1.scalar += q2.scalar;
2813     q1.vector += q2.vector;
2814 }
2815
2816 inline void operator-=(Quaternion& q1, const Quaternion& q2)
2817 {
2818     q1.scalar -= q2.scalar;
2819     q1.vector -= q2.vector;
2820 }
2821
2822 inline void operator*=(Quaternion& q1, float k)
2823 {
2824     q1.scalar *= k;
2825     q1.vector *= k;
2826 }
2827
2828 inline void operator*=(Quaternion& q1, const Quaternion& q2)
2829 {
2830     //q1q2 = [w1, v1][w2, v2] = [w1w2 - v1 dot v2, w1v2 + w2v1 + v1 x v2]
2831     //w is the scalar component and v is the vector component
2832
2833     q1.scalar = q1.scalar * q2.scalar - DotProduct(q1.vector, q2.vector);
2834     q1.vector = q1.scalar * q2.vector + q2.scalar * q1.vector + CrossProduct(q1.vector, q2.vector);
2835 }
2836
2837 inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
2838 {
2839     return Quaternion{ q1.scalar + q2.scalar, q1.vector + q2.vector };
2840 }
2841
2842 inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
2843 {
2844     return Quaternion{ q1.scalar - q2.scalar, q1.vector - q2.vector };
2845 }
2846
2847 inline Quaternion operator-(const Quaternion& q)
2848 {
2849     return Quaternion{ -q.scalar, -q.vector };
2850 }
2851
2852 inline Quaternion operator*(const Quaternion& q, float k)
2853 {
2854     return Quaternion{ q.scalar * k, q.vector * k };
2855 }
2856
2857 inline Quaternion operator*(float k, const Quaternion& q)
2858 {
2859     return Quaternion{ k * q.scalar, k * q.vector };
2860 }
2861
2862 inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
2863 {
2864     //q1q2 = [w1, v1][w2, v2] = [w1w2 - v1 dot v2, w1v2 + w2v1 + v1 x v2]
2865     //w is the scalar component and v is the vector component
2866
2867     float scalarResult{ 0.0f };
2868     Vector3D vectorResult;

```

```

2897     scalarResult = q1.scalar * q2.scalar - DotProduct(q1.vector, q2.vector);
2898     vectorResult = q1.scalar * q2.vector + q2.scalar * q1.vector + CrossProduct(q1.vector,
2899     q2.vector);
2900
2901     return Quaternion{ scalarResult, vectorResult };
2902 }
2903
2904 inline bool operator==(const Quaternion& q1, const Quaternion& q2)
2905 {
2906     return CompareFloats(q1.scalar, q2.scalar, EPSILON) && (q1.vector == q2.vector);
2907 }
2908
2909 inline bool operator!=(const Quaternion& q1, const Quaternion& q2)
2910 {
2911     return !(q1 == q2);
2912 }
2913
2914 inline bool ZeroQuaternion(const Quaternion& q)
2915 {
2916     //zero quaternion = (0, 0, 0, 0)
2917     return CompareFloats(q.scalar, 0.0f, EPSILON) && ZeroVector(q.vector);
2918 }
2919
2920 inline bool Identity(const Quaternion& q)
2921 {
2922     //identity quaternion = (1, 0, 0, 0)
2923     return CompareFloats(q.scalar, 1.0f, EPSILON) && CompareFloats(q.vector.x, 0.0f, EPSILON) &&
2924     CompareFloats(q.vector.y, 0.0f, EPSILON) && CompareFloats(q.vector.z, 0.0f, EPSILON);
2925 }
2926
2927 inline Quaternion Conjugate(const Quaternion& q)
2928 {
2929     //conjugate of a quaternion is the quaternion with its vector part negated
2930     return Quaternion{ q.scalar, -q.vector };
2931 }
2932
2933 inline float Length(const Quaternion& q)
2934 {
2935     //length of a quaternion = sqrt(scalar^2 + x^2 + y^2 + z^2)
2936     return sqrt(q.scalar * q.scalar + q.vector.x * q.vector.x + q.vector.y * q.vector.y +
2937     q.vector.z * q.vector.z);
2938 }
2939
2940 inline Quaternion Normalize(const Quaternion& q)
2941 {
2942     //to normalize a quaternion you do q / |q|
2943
2944     if (ZeroQuaternion(q))
2945         return q;
2946
2947     float inverseMagnitdue{ 1.0f / Length(q) };
2948
2949     return Quaternion{ q.scalar * inverseMagnitdue, q.vector * inverseMagnitdue };
2950 }
2951
2952 inline Quaternion Inverse(const Quaternion& q)
2953 {
2954     //inverse = conjugate of q / |q|
2955
2956     if (ZeroQuaternion(q))
2957         return q;
2958
2959     float inverseMagnitdue{ 1.0f / Length(q) };
2960
2961     return Quaternion{ Conjugate(q) * inverseMagnitdue };
2962 }
2963
2964 inline Quaternion RotationQuaternion(float angle, float x, float y, float z)
2965 {
2966     //A roatation quaternion is a quaternion where the
2967     //scalar part = cos(theta / 2)
2968     //vector part = sin(theta / 2) * axis
2969     //the axis needs to be normalized
2970
2971     float ang{ angle / 2.0f };
2972     float c{ cos(ang * PI / 180.0f) };
2973     float s{ sin(ang * PI / 180.0f) };
2974
2975     Vector3D axis{ x, y, z };
2976     axis = Normalize(axis);
2977
2978     return Quaternion{ c, s * axis };
2979 }
2980
2981 inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
2982 {

```

```

3010         //A roatation quaternion is a quaternion where the
3011         //scalar part = cos(theta / 2)
3012         //vector part = sin(theta / 2) * axis
3013         //the axis needs to be normalized
3014
3015         float ang{ angle / 2.0f };
3016         float c{ cos(ang * PI / 180.0f) };
3017         float s{ sin(ang * PI / 180.0f) };
3018
3019         Vector3D axisN(Normalize(axis));
3020
3021         return Quaternion{ c, s * axisN };
3022     }
3023
3029     inline Quaternion RotationQuaternion(const Vector4D& angAxis)
3030     {
3031         //A roatation quaternion is a quaternion where the
3032         //scalar part = cos(theta / 2)
3033         //vector part = sin(theta / 2) * axis
3034         //the axis needs to be normalized
3035
3036         float angle{ angAxis.x / 2.0f };
3037         float c{ cos(angle * PI / 180.0f) };
3038         float s{ sin(angle * PI / 180.0f) };
3039
3040         Vector3D axis{ angAxis.y, angAxis.z, angAxis.w };
3041         axis = Normalize(axis);
3042
3043         return Quaternion{ c, s * axis };
3044     }
3045
3050     inline Vector3D Rotate(const Quaternion& q, const Vector3D& p)
3051     {
3052         //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
point/vector in quaternion form, q is a rotation quaternion
3053         //and q* is its conjugate.
3054
3055         Quaternion point{ 0.0f, p };
3056
3057         Quaternion result(q * point * Conjugate(q));
3058
3059         //The rotated vector/point is in the vector component of the quaternion.
3060         return result.vector;
3061     }
3062
3067     inline Vector4D Rotate(const Quaternion& q, const Vector4D& p)
3068     {
3069         //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the
point/vector, q is a rotation quaternion
3070         //and q* is its conjugate.
3071
3072         Quaternion point{ 0.0f, p.x, p.y, p.z };
3073
3074         Quaternion result(q * point * Conjugate(q));
3075
3076         //The rotated vector/point is in the vector component of the quaternion.
3077         return Vector4D{ result.vector.x, result.vector.y, result.vector.z, p.w };
3078     }
3079
3084     inline Matrix3x3 QuaternionToRotationMatrixCol3x3(const Quaternion& q)
3085     {
3086         //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3
3087         //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2
3088         //2q2q4 - 2q1q3          2q3q4 + 2q1q2          1 - 2q2^2 - 2q3^2
3089         //q1 = scalar
3090         //q2 = x
3091         //q3 = y
3092         //q4 = z
3093
3094         Matrix3x3 colMat;
3095
3096         colMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
3097         colMat(0, 1) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
3098         colMat(0, 2) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
3099
3100         colMat(1, 0) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
3101         colMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
3102         colMat(1, 2) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
3103
3104         colMat(2, 0) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
3105         colMat(2, 1) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
3106         colMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;
3107
3108         return colMat;
3109     }
3110
3115     inline Matrix3x3 QuaternionToRotationMatrixRow3x3(const Quaternion& q)

```

```

3116     {
3117         //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3
3118         //2q2q3 - 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2
3119         //2q2q4 + 2q1q3      2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2
3120         //q1 = scalar
3121         //q2 = x
3122         //q3 = y
3123         //q4 = z
3124
3125         Matrix3x3 rowMat;
3126
3127         rowMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
3128         rowMat(0, 1) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
3129         rowMat(0, 2) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
3130
3131         rowMat(1, 0) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
3132         rowMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
3133         rowMat(1, 2) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
3134
3135         rowMat(2, 0) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
3136         rowMat(2, 1) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
3137         rowMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;
3138
3139         return rowMat;
3140     }
3141
3142 inline Matrix4x4 QuaternionToRotationMatrixCol4x4(const Quaternion& q)
3143 {
3144     //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
3145     //2q2q3 + 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 - 2q1q2      0
3146     //2q2q4 - 2q1q3      2q3q4 + 2q1q2      1 - 2q2^2 - 2q3^2      0
3147     //0                      0                      0                      1
3148     //q1 = scalar
3149     //q2 = x
3150     //q3 = y
3151     //q4 = z
3152
3153     Matrix4x4 colMat;
3154
3155     colMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
3156     colMat(0, 1) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
3157     colMat(0, 2) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
3158
3159     colMat(1, 0) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
3160     colMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
3161     colMat(1, 2) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
3162
3163     colMat(2, 0) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
3164     colMat(2, 1) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
3165     colMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;
3166
3167     return colMat;
3168 }
3169
3170 inline Matrix4x4 QuaternionToRotationMatrixRow4x4(const Quaternion& q)
3171 {
3172     //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
3173     //2q2q3 - 2q1q4      1 - 2q2^2 - 2q4^2      2q3q4 + 2q1q2      0
3174     //2q2q4 + 2q1q3      2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2      0
3175     //0                      0                      0                      1
3176     //q1 = scalar
3177     //q2 = x
3178     //q3 = y
3179     //q4 = z
3180
3181     Matrix4x4 rowMat;
3182
3183     rowMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
3184     rowMat(0, 1) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
3185     rowMat(0, 2) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
3186
3187     rowMat(1, 0) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
3188     rowMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
3189     rowMat(1, 2) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
3190
3191     rowMat(2, 0) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
3192     rowMat(2, 1) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
3193     rowMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;
3194
3195     return rowMat;
3196 }
3197
3198 inline float DotProduct(const Quaternion& q1, const Quaternion& q2)
3199 {
3200     //q1 dot q2 = [w1, v1] dot [w2, v2] = w1w2 + v1 dot v2
3201     //w is the scalar component and v is the vector component.
3202
3203 }
3204
3205
3206
3207
3208
3209
3210
3211
3212

```

```

3213     return q1.scalar * q2.scalar + DotProduct(q1.vector, q2.vector);
3214 }
3215
3220 inline Quaternion Lerp(const Quaternion& q1, const Quaternion& q2, float t)
3221 {
3222     if (t < 0.0f)
3223         return q1;
3224     else if (t > 1.0f)
3225         return q2;
3226
3227     //Compute the cosine of the angle between the quaternions
3228     float cosOmega = DotProduct(q1, q2);
3229
3230     Quaternion newQ2;
3231     //If the dot product is negative, negate q2 to so we take the shorter arc
3232     if (cosOmega < 0.0f)
3233     {
3234         newQ2 = -q2;
3235         cosOmega = -cosOmega;
3236     }
3237     else
3238     {
3239         newQ2 = q2;
3240     }
3241
3242     return (1.0f - t) * q1 + t * newQ2;
3243 }
3244
3249 inline Quaternion NLERP(const Quaternion& q1, const Quaternion& q2, float t)
3250 {
3251     if (t < 0.0f)
3252         return q1;
3253     else if (t > 1.0f)
3254         return q2;
3255
3256     //Compute the cosine of the angle between the quaternions
3257     float cosOmega = DotProduct(q1, q2);
3258
3259     Quaternion newQ2;
3260     //If the dot product is negative, negate q2 to so we take the shorter arc
3261     if (cosOmega < 0.0f)
3262     {
3263         newQ2 = -q2;
3264         cosOmega = -cosOmega;
3265     }
3266     else
3267     {
3268         newQ2 = q2;
3269     }
3270
3271     return Normalize((1.0f - t) * q1 + t * newQ2);
3272 }
3273
3278 inline Quaternion Slerp(const Quaternion& q1, const Quaternion& q2, float t)
3279 {
3280     //Formula used is
3281     //k1 = sin((1 - t)omega) * omega) / sin(omega);
3282     //k2 = (sin(tomega) * omega) / sin(omega)
3283     //newQ = k1q1 * k2q2
3284     //Omega is the angle between the q0 and q1.
3285
3286     if (t < 0.0f)
3287         return q1;
3288     else if (t > 1.0f)
3289         return q2;
3290
3291     //Compute the cosine of the angle between the quaternions
3292     float cosOmega = DotProduct(q1, q2);
3293
3294     Quaternion newQ2;
3295     //If the dot product is negative, negate q2 to so we take the shorter arc
3296     if (cosOmega < 0.0f)
3297     {
3298         newQ2 = -q2;
3299         cosOmega = -cosOmega;
3300     }
3301     else
3302     {
3303         newQ2 = q2;
3304     }
3305
3306     float k1{ 0.0f };
3307     float k2{ 0.0f };
3308
3309     //Linear interpolate if the quaternions are very close to protect dividing by zero.
3310     if (cosOmega > 0.9999f)
3311     {

```

```

3312         k1 = 1.0f - t;
3313         k2 = t;
3314     }
3315     else
3316     {
3317         //sin of the angle between the quaternions is
3318         //sin(omega) = 1 - cos^2(omega) from the trig identity
3319         //sin^2(omega) + cos^2(omega) = 1.
3320         float sinOmega{ sqrt(1.0f - cosOmega * cosOmega) };
3321
3322         //retrieve the angle
3323         float omega{ atan2(sinOmega, cosOmega) };
3324
3325         //Compute inverse to avoid dividng multiple times
3326         float oneOverSinOmega{ 1.0f / sinOmega };
3327
3328         k1 = sin((1.0f - t) * omega) * oneOverSinOmega;
3329         k2 = sin(t * omega) * oneOverSinOmega;
3330     }
3331
3332     return k1 * q1 + k2 * newQ2;
3333 }
3334
3335 #if defined(_DEBUG)
3336 inline void print(const Quaternion& q)
3337 {
3338     std::cout << "(" << q.scalar << ", " << q.vector.x << ", " << q.vector.y << ", " << q.vector.z << ")";
3339 }
3340 #endif
3341
3342 //-----
3343 }
3344
3345 typedef MathEngine::Vector2D vec2;
3346 typedef MathEngine::Vector3D vec3;
3347 typedef MathEngine::Vector4D vec4;
3348 typedef MathEngine::Matrix2x2 mat2;
3349 typedef MathEngine::Matrix3x3 mat3;
3350 typedef MathEngine::Matrix4x4 mat4;
3351 typedef MathEngine::Quaternion quaternion;

```


Index

- Adjoint
 - MathEngine, [13](#)
- Clamp
 - MathEngine, [13](#)
- Cofactor
 - MathEngine, [13](#), [14](#)
- CompareDoubles
 - MathEngine, [14](#)
- CompareFloats
 - MathEngine, [14](#)
- Conjugate
 - MathEngine, [14](#)
- CrossProduct
 - MathEngine, [15](#)
- Data
 - MathEngine::Matrix2x2, [43](#)
 - MathEngine::Matrix3x3, [48](#)
 - MathEngine::Matrix4x4, [53](#)
- Determinant
 - MathEngine, [15](#)
- DotProduct
 - MathEngine, [15](#), [16](#)
- EPSILON
 - MathEngine.h, [67](#)
- GetCol
 - MathEngine::Matrix2x2, [43](#)
 - MathEngine::Matrix3x3, [48](#)
 - MathEngine::Matrix4x4, [53](#)
- GetRow
 - MathEngine::Matrix2x2, [43](#)
 - MathEngine::Matrix3x3, [48](#)
 - MathEngine::Matrix4x4, [53](#)
- Identity
 - MathEngine, [16](#), [17](#)
- Inverse
 - MathEngine, [17](#)
- Length
 - MathEngine, [18](#)
- Lerp
 - MathEngine, [18](#), [19](#)
- mat2
 - MathEngine.h, [68](#)
- mat3
 - MathEngine.h, [68](#)
- mat4
 - MathEngine.h, [68](#)
- MathEngine, [7](#)
 - Adjoint, [13](#)
 - Clamp, [13](#)
 - Cofactor, [13](#), [14](#)
 - CompareDoubles, [14](#)
 - CompareFloats, [14](#)
 - Conjugate, [14](#)
 - CrossProduct, [15](#)
 - Determinant, [15](#)
 - DotProduct, [15](#), [16](#)
 - Identity, [16](#), [17](#)
 - Inverse, [17](#)
 - Length, [18](#)
 - Lerp, [18](#), [19](#)
 - NLerp, [19](#)
 - Normalize, [19](#), [20](#)
 - operator!=, [20](#), [21](#)
 - operator*, [21–26](#)
 - operator*=[26](#), [27](#)
 - operator+, [27](#), [28](#)
 - operator+=, [28](#), [29](#)
 - operator-, [29–32](#)
 - operator-=, [32](#)
 - operator==, [33](#)
 - Orthonormalize, [33](#)
 - QuaternionToRotationMatrixCol3x3, [34](#)
 - QuaternionToRotationMatrixCol4x4, [34](#)
 - QuaternionToRotationMatrixRow3x3, [34](#)
 - QuaternionToRotationMatrixRow4x4, [34](#)
 - Rotate, [34](#), [35](#)
 - Rotate4x4, [35](#), [36](#)
 - RotationQuaternion, [36](#)
 - Scale, [36](#), [37](#)
 - Scale4x4, [37](#)
 - SetToIdentity, [37](#), [38](#)
 - Slerp, [38](#)
 - Translate, [38](#)
 - Transpose, [39](#)
 - ZeroQuaternion, [39](#)
 - ZeroVector, [39](#), [40](#)
- MathEngine.h, [61](#)
 - EPSILON, [67](#)
 - mat2, [68](#)
 - mat3, [68](#)
 - mat4, [68](#)
 - PI, [67](#)
 - PI2, [67](#)

- quaternion, 68
- vec2, 68
- vec3, 68
- vec4, 68
- MathEngine::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
- MathEngine::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
- MathEngine::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
- MathEngine::Quaternion, 56
 - scalar, 56
 - vector, 56
- MathEngine::Vector2D, 57
 - x, 57
 - y, 57
- MathEngine::Vector3D, 58
 - x, 58
 - y, 58
 - z, 58
- MathEngine::Vector4D, 59
 - w, 59
 - x, 59
 - y, 59
 - z, 59
- Matrix2x2
 - MathEngine::Matrix2x2, 42, 43
- Matrix3x3
 - MathEngine::Matrix3x3, 47, 48
- Matrix4x4
 - MathEngine::Matrix4x4, 52, 53
- NLerp
 - MathEngine, 19
- Normalize
 - MathEngine, 19, 20
- operator!=
 - MathEngine, 20, 21
- operator*
 - MathEngine, 21–26
- operator*=
 - MathEngine, 26, 27
 - MathEngine::Matrix2x2, 44
 - MathEngine::Matrix3x3, 49
 - MathEngine::Matrix4x4, 54
- operator()
 - MathEngine::Matrix2x2, 44
 - MathEngine::Matrix3x3, 49
 - MathEngine::Matrix4x4, 54
- operator+
 - MathEngine, 27, 28
- operator+=
 - MathEngine, 28, 29
 - MathEngine::Matrix2x2, 44
 - MathEngine::Matrix3x3, 49
 - MathEngine::Matrix4x4, 54
- operator-
 - MathEngine, 29–32
- operator-=
 - MathEngine, 32
 - MathEngine::Matrix2x2, 45
 - MathEngine::Matrix3x3, 50
 - MathEngine::Matrix4x4, 55
- operator=
 - MathEngine::Matrix2x2, 45
 - MathEngine::Matrix3x3, 50
 - MathEngine::Matrix4x4, 55
- operator==
 - MathEngine, 33
- Orthonormalize
 - MathEngine, 33
- PI
 - MathEngine.h, 67
- PI2
 - MathEngine.h, 67
- quaternion
 - MathEngine.h, 68
- QuaternionToRotationMatrixCol3x3
 - MathEngine, 34
- QuaternionToRotationMatrixCol4x4
 - MathEngine, 34
- QuaternionToRotationMatrixRow3x3
 - MathEngine, 34
- QuaternionToRotationMatrixRow4x4

- MathEngine, [34](#)
- Rotate
 - MathEngine, [34](#), [35](#)
- Rotate4x4
 - MathEngine, [35](#), [36](#)
- RotationQuaternion
 - MathEngine, [36](#)
- scalar
 - MathEngine::Quaternion, [56](#)
- Scale
 - MathEngine, [36](#), [37](#)
- Scale4x4
 - MathEngine, [37](#)
- SetCol
 - MathEngine::Matrix2x2, [45](#)
 - MathEngine::Matrix3x3, [50](#)
 - MathEngine::Matrix4x4, [55](#)
- SetRow
 - MathEngine::Matrix2x2, [45](#)
 - MathEngine::Matrix3x3, [50](#)
 - MathEngine::Matrix4x4, [55](#)
- SetToIdentity
 - MathEngine, [37](#), [38](#)
- Slerp
 - MathEngine, [38](#)
- Translate
 - MathEngine, [38](#)
- Transpose
 - MathEngine, [39](#)
- vec2
 - MathEngine.h, [68](#)
- vec3
 - MathEngine.h, [68](#)
- vec4
 - MathEngine.h, [68](#)
- vector
 - MathEngine::Quaternion, [56](#)
- w
 - MathEngine::Vector4D, [59](#)
- x
 - MathEngine::Vector2D, [57](#)
 - MathEngine::Vector3D, [58](#)
 - MathEngine::Vector4D, [59](#)
- y
 - MathEngine::Vector2D, [57](#)
 - MathEngine::Vector3D, [58](#)
 - MathEngine::Vector4D, [59](#)
- z
 - MathEngine::Vector3D, [58](#)
 - MathEngine::Vector4D, [59](#)
- ZeroQuaternion
 - MathEngine, [39](#)
- ZeroVector
 - MathEngine, [39](#), [40](#)