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 stuct used for 2D vectors/points.
- struct Vector3D
  A vector stuct used for 3D vectors/points.
- struct Vector4D
  A vector stuct 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 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 3D 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 vectorv2, 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 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 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 vectorv2, 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 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 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 vectorv2, 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 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 \m and returns the result.
- Matrix3x3 operator* (const Matrix3x3 &m1, const Matrix3x3 &m2)

    Multiplies m1 with \m2 and returns the result.
- Vector3D operator* (const Matrix3x3 &m, const Vector3D &v)

    Multiplies m with v and returns the result.
- Vector3D operator* (const Vector3D &v, const Matrix3x3 &m)

    Multiplies v with m and returns the result.
- void SetToIdentity (Matrix3x3 &m)

    Sets m to the identity matrix.
- bool 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 *epsilion* and adaptive *epsilion* 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 *epsilion* and adaptive *epsilion* 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 stuct used for 2D vectors/points.
- struct MathEngine::Vector3D

  A vector stuct used for 3D vectors/points.
- struct MathEngine::Vector4D

  A vector stuct 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 vectorv2, 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 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 vectorv2, 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 vectorv2, 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 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::SetToIdentity (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::SetToIdentity (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.

```
#pragma once

#include <cmath>

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


#define EPSILON 1e-6f
#define PI 3.14159f
#define PI2 6.28319f

namespace MathEngine
{

    struct Vector2D;
    struct Vector3D;
    struct Vector4D;
    class Matrix2x2;
    class Matrix3x3;
    class Matrix4x4;


   //-----------------------------------------------------------------------------------------------------------------
//COMPARISON FUNCTIONS

    inline bool CompareFloats(float x, float y, float epsilon)
    {
        float diff = fabs(x - y);
        //exact epsilon
        if (diff < epsilon)
        {
            return true;
        }

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

    inline bool CompareDoubles(double x, double y, double epsilon)
    {
        double diff = fabs(x - y);
        //exact epsilon
        if (diff < epsilon)
        {
            return true;
        }

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

    inline float Clamp(float value, float a, float b)
    {
        if (value < a)
            return a;

        if (value > b)
            return b;

        return value;
    }


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


   //-----------------------------------------------------------------------------------------------------------------
//2D VECTOR

    struct Vector2D
```

```
    {
        float x = 0.0f;
        float y = 0.0f;
    };

    inline void operator+=(Vector2D& v1, const Vector2D& v2)
    {
        v1.x += v2.x;
        v1.y += v2.y;
    }

    inline void operator-=(Vector2D& v1, const Vector2D& v2)
    {
        v1.x -= v2.x;
        v1.y -= v2.y;
    }

    inline void operator*=(Vector2D& v, float k)
    {
        v.x *= k;
        v.y *= k;
    }

    inline Vector2D operator+(const Vector2D& v1, const Vector2D& v2)
    {
        return Vector2D{ v1.x + v2.x, v1.y + v2.y };
    }

    inline Vector2D operator-(const Vector2D& v)
    {
        return Vector2D{ -v.x, -v.y };
    }

    inline Vector2D operator-(const Vector2D& v1, const Vector2D& v2)
    {
        return Vector2D{ v1.x - v2.x, v1.y - v2.y };
    }

    inline Vector2D operator*(const Vector2D& v, float k)
    {
        return Vector2D{ v.x * k, v.y * k };
    }

    inline Vector2D operator*(float k, const Vector2D& v)
    {
        return Vector2D{ k * v.x, k * v.y };
    }

    inline bool operator==(const Vector2D& v1, const Vector2D& v2)
    {
        return CompareFloats(v1.x, v2.x, EPSILON) && CompareFloats(v1.y, v2.y, EPSILON);
    }

    inline bool operator!=(const Vector2D& v1, const Vector2D& v2)
    {
        return !(v1 == v2);
    }

    inline bool ZeroVector(const Vector2D& v)
    {
        return CompareFloats(v.x, 0.0f, EPSILON) && CompareFloats(v.y, 0.0f, EPSILON);
    }

    inline float DotProduct(const Vector2D& v1, const Vector2D& v2)
    {
        return v1.x * v2.x + v1.y * v2.y;
    }

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

    inline Vector2D Normalize(const Vector2D& v)
    {
        if (ZeroVector(v))
            return v;

        float inverseLength{ 1.0f / Length(v) };

        return v * inverseLength;
    }

    inline Vector2D Lerp(const Vector2D& start, const Vector2D& end, float t)
    {
        if (t < 0.0f)
            return start;
```

```
        else if (t > 1.0f)
            return end;

        return (1.0f - t) * start + t * end;
    }

#if defined(_DEBUG)
    inline void print(const Vector2D& v)
    {
        std::cout « "(" « v.x « ", " « v.y « ")";
    }
#endif


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


  //-----------------------------------------------------------------------------------------------------------------------
//3D VECTOR

    struct Vector3D
    {
        float x = 0.0f;
        float y = 0.0f;
        float z = 0.0f;
    };


    inline void operator+=(Vector3D& v1, const Vector3D& v2)
    {
        v1.x += v2.x;
        v1.y += v2.y;
        v1.z += v2.z;
    }

    inline void operator-=(Vector3D& v1, const Vector3D& v2)
    {
        v1.x -= v2.x;
        v1.y -= v2.y;
        v1.z -= v2.z;
    }

    inline void operator*=(Vector3D& v, float k)
    {
        v.x *= k;
        v.y *= k;
        v.z *= k;
    }

    inline Vector3D operator+(const Vector3D& v1, const Vector3D& v2)
    {
        return Vector3D{ v1.x + v2.x, v1.y + v2.y, v1.z + v2.z };
    }

    inline Vector3D operator-(const Vector3D& v)
    {
        return Vector3D{ -v.x, -v.y, -v.z };
    }

    inline Vector3D operator-(const Vector3D& v1, const Vector3D& v2)
    {
        return Vector3D{ v1.x - v2.x, v1.y - v2.y, v1.z - v2.z};
    }

    inline Vector3D operator*(const Vector3D& v, float k)
    {
        return Vector3D{ v.x * k, v.y * k, v.z * k };
    }

    inline Vector3D operator*(float k, const Vector3D& v)
    {
        return Vector3D{ k * v.x, k * v.y, k * v.z};
    }

    inline bool operator==(const Vector3D& v1, const Vector3D& v2)
    {
        return CompareFloats(v1.x, v2.x, EPSILON) && CompareFloats(v1.y, v2.y, EPSILON) &&
  CompareFloats(v1.z, v2.z, EPSILON);
    }

    inline bool operator!=(const Vector3D& v1, const Vector3D& v2)
    {
        return !(v1 == v2);
    }

    inline bool ZeroVector(const Vector3D& v)
    {
```

```
        return CompareFloats(v.x, 0.0f, EPSILON) && CompareFloats(v.y, 0.0f, EPSILON) && CompareFloats(v.z, 0.0f, EPSILON);
    }

    inline float DotProduct(const Vector3D& v1, const Vector3D& v2)
    {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
    }

    inline Vector3D CrossProduct(const Vector3D& v1, const Vector3D& v2)
    {
        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 };
    }

    inline float Length(const Vector3D& v)
    {
        return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
    }

    inline Vector3D Normalize(const Vector3D& v)
    {
        if (ZeroVector(v))
            return v;

        float inverseLength{ 1.0f / Length(v) };

        return v * inverseLength;
    }

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

    inline Vector3D Lerp(const Vector3D& start, const Vector3D& end, float t)
    {
        if (t < 0.0f)
            return start;
        else if (t > 1.0f)
            return end;

        return (1.0f - t) * start + t * end;
    }

#if defined(_DEBUG)
    inline void print(const Vector3D& v)
    {
        std::cout << "(" << v.x << ", " << v.y << ")";
    }
#endif


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



    //------------------------------------------------------------------------------------------------------------------------------
//4D VECTOR

    struct Vector4D
    {
        float x = 0.0f;
        float y = 0.0f;
        float z = 0.0f;
        float w = 0.0f;
    };


    inline void operator+=(Vector4D& v1, const Vector4D& v2)
    {
        v1.x += v2.x;
        v1.y += v2.y;
        v1.z += v2.z;
        v1.w += v2.w;
    }

    inline void operator-=(Vector4D& v1, const Vector4D& v2)
    {
        v1.x -= v2.x;
        v1.y -= v2.y;
        v1.z -= v2.z;
        v1.w -= v2.w;
    }

```

```
    inline void operator*=(Vector4D& v, float k)
    {
        v.x *= k;
        v.y *= k;
        v.z *= k;
        v.w *= k;
    }

    inline Vector4D operator+(const Vector4D& v1, const Vector4D& v2)
    {
        return Vector4D{ v1.x + v2.x, v1.y + v2.y, v1.z + v2.z, v1.w + v2.w };
    }

    inline Vector4D operator-(const Vector4D& v)
    {
        return Vector4D{ -v.x, -v.y, -v.z, - v.w };
    }

    inline Vector4D operator-(const Vector4D& v1, const Vector4D& v2)
    {
        return Vector4D{ v1.x - v2.x, v1.y - v2.y, v1.z - v2.z, v1.w - v2.w };
    }

    inline Vector4D operator*(const Vector4D& v, float k)
    {
        return Vector4D{ v.x * k, v.y * k, v.z * k, v.w * k };
    }

    inline Vector4D operator*(float k, const Vector4D& v)
    {
        return Vector4D{ k * v.x, k * v.y, k * v.z, k * v.w };
    }

    inline bool operator==(const Vector4D& v1, const Vector4D& v2)
    {
        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);
    }

    inline bool operator!=(const Vector4D& v1, const Vector4D& v2)
    {
        return !(v1 == v2);
    }

    inline bool ZeroVector(const Vector4D& v)
    {
        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);
    }

    inline float DotProduct(const Vector4D& v1, const Vector4D& v2)
    {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z + v1.w * v2.w;
    }

    inline float Length(const Vector4D& v)
    {
        return sqrt(v.x * v.x + v.y * v.y + v.z * v.z + v.w * v.w);
    }

    inline Vector4D Normalize(const Vector4D& v)
    {
        if (ZeroVector(v))
            return v;

        float inverseLength{ 1.0f / Length(v) };

        return v * inverseLength;
    }

    inline Vector4D Lerp(const Vector4D& start, const Vector4D& end, float t)
    {
        if (t < 0.0f)
            return start;
        else if (t > 1.0f)
            return end;

        return (1.0f - t) * start + t * end;
    }

#if defined(_DEBUG)
    inline void print(const Vector4D& v)
    {
        std::cout « "(" « v.x « ", " « v.y « ")";
    }
```

```
#endif


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




    //------------------------------------------------------------------------------------------------------------
    class Matrix2x2
    {
    public:

        Matrix2x2();

        Matrix2x2(float a[][2]);

        Matrix2x2(const Vector2D& r1, const Vector2D& r2);

        Matrix2x2(const Matrix3x3& m);

        Matrix2x2(const Matrix4x4& m);

        float* Data();

        const float* Data() const;

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

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

        Vector2D GetRow(unsigned int row) const;

        Vector2D GetCol(unsigned int col) const;

        void SetRow(unsigned int row, Vector2D v);

        void SetCol(unsigned int col, Vector2D v);

        Matrix2x2& operator=(const Matrix3x3& m);

        Matrix2x2& operator=(const Matrix4x4& m);

        Matrix2x2& operator+=(const Matrix2x2& m);

        Matrix2x2& operator-=(const Matrix2x2& m);

        Matrix2x2& operator*=(float k);

        Matrix2x2& operator*=(const Matrix2x2& m);

    private:

        float mMat[2][2];
    };

    //-------------------------------------------------------------------------------------
    inline Matrix2x2::Matrix2x2()
    {
        //1st row
        mMat[0][0] = 1.0f;
        mMat[0][1] = 0.0f;

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

    inline Matrix2x2::Matrix2x2(float a[][2])
    {
        //1st row
        mMat[0][0] = a[0][0];
        mMat[0][1] = a[0][1];

        //2nd row
        mMat[1][0] = a[1][0];
        mMat[1][1] = a[1][1];
    }

    inline Matrix2x2::Matrix2x2(const Vector2D& r1, const Vector2D& r2)
    {
        SetRow(0, r1);
        SetRow(1, r2);
    }

    inline float* Matrix2x2::Data()
    {
```

```
        return mMat[0];
    }

    inline const float* Matrix2x2::Data()const
{
        return mMat[0];
    }

    inline const float& Matrix2x2::operator()(unsigned int row, unsigned int col)const
{
        if (row > 1 || col > 1)
        {
            return mMat[0][0];
        }
        else
        {
            return mMat[row][col];
        }
    }

    inline float& Matrix2x2::operator()(unsigned int row, unsigned int col)
    {
        if (row > 1 || col > 1)
        {
            return mMat[0][0];
        }
        else
        {
            return mMat[row][col];
        }
    }

    inline Vector2D Matrix2x2::GetRow(unsigned int row)const
{
        if (row < 0 || row > 1)
            return Vector2D{ mMat[0][0], mMat[0][1] };
        else
            return Vector2D{ mMat[row][0], mMat[row][1] };

    }

    inline Vector2D Matrix2x2::GetCol(unsigned int col)const
{
        if (col < 0 || col > 1)
            return Vector2D{ mMat[0][0], mMat[1][0] };
        else
            return Vector2D{ mMat[0][col], mMat[1][col] };
    }

    inline void Matrix2x2::SetRow(unsigned int row, Vector2D v)
    {
        if (row > 1)
        {
            mMat[0][0] = v.x;
            mMat[0][1] = v.y;
        }
        else
        {
            mMat[row][0] = v.x;
            mMat[row][1] = v.y;
        }
    }

    inline void Matrix2x2::SetCol(unsigned int col, Vector2D v)
    {
        if (col > 1)
        {
            mMat[0][0] = v.x;
            mMat[1][0] = v.y;
        }
        else
        {
            mMat[0][col] = v.x;
            mMat[1][col] = v.y;
        }
    }

    inline Matrix2x2& Matrix2x2::operator+=(const Matrix2x2& m)
    {
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                this->mMat[i][j] += m.mMat[i][j];
            }
        }

```

```
        return *this;
    }

    inline Matrix2x2& Matrix2x2::operator-=(const Matrix2x2& m)
    {
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                this->mMat[i][j] -= m.mMat[i][j];
            }
        }

        return *this;
    }

    inline Matrix2x2& Matrix2x2::operator*=(float k)
    {
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                this->mMat[i][j] *= k;
            }
        }

        return *this;
    }

    inline Matrix2x2& Matrix2x2::operator*=(const Matrix2x2& m)
    {
        Matrix2x2 res;

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

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

        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                mMat[i][j] = res.mMat[i][j];
            }
        }

        return *this;
    }

    inline Matrix2x2 operator+(const Matrix2x2& m1, const Matrix2x2& m2)
    {
        Matrix2x2 res;
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                res(i, j) = m1(i, j) + m2(i, j);
            }
        }

        return res;
    }

    inline Matrix2x2 operator-(const Matrix2x2& m)
    {
        Matrix2x2 res;
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                res(i, j) = -m(i, j);
            }
        }

        return res;
    }

    inline Matrix2x2 operator-(const Matrix2x2& m1, const Matrix2x2& m2)
    {
        Matrix2x2 res;
        for (int i = 0; i < 2; ++i)
```

```
        {
            for (int j = 0; j < 2; ++j)
            {
                res(i, j) = m1(i, j) - m2(i, j);
            }
        }

        return res;
    }

    inline Matrix2x2 operator*(const Matrix2x2& m, const float& k)
    {
        Matrix2x2 res;
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                res(i, j) = m(i, j) * k;
            }
        }

        return res;
    }

    inline Matrix2x2 operator*(const float& k, const Matrix2x2& m)
    {
        Matrix2x2 res;
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                res(i, j) = k * m(i, j);
            }
        }

        return res;
    }

    inline Matrix2x2 operator*(const Matrix2x2& m1, const Matrix2x2& m2)
    {
        Matrix2x2 res;

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

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

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

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

        return res;
    }

    inline Vector2D operator*(const Matrix2x2& m, const Vector2D& v)
    {
        Vector2D res;

        res.x = m(0, 0) * v.x + m(0, 1) * v.y;

        res.y = m(1, 0) * v.x + m(1, 1) * v.y;

        return res;
    }

    inline Vector2D operator*(const Vector2D& v, const Matrix2x2& m)
    {
        Vector2D res;

        res.x = v.x * m(0, 0) + v.y * m(1, 0);

        res.y = v.x * m(0, 1) + v.y * m(1, 1);

        return res;
    }

    inline void SetToIdentity(Matrix2x2& m)
```

```
    {
        //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f

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

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

    inline bool Identity(const Matrix2x2& m)
    {
        //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
0.0f

        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                if (i == j)
                {
                    if (!CompareFloats(m(i, j), 1.0f, EPSILON))
                        return false;

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

                }
            }
        }
    }

    inline Matrix2x2 Transpose(const Matrix2x2& m)
    {
        //make the rows into cols

        Matrix2x2 res;

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

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

        return res;
    }

    inline Matrix2x2 Scale(float x, float y)
    {
        //x 0
        //0 y

        Matrix2x2 scale;
        scale(0, 0) = x;
        scale(1, 1) = y;

        return scale;
    }

    inline Matrix2x2 Scale(const Vector2D& scaleVector)
    {
        //x 0
        //0 y

        Matrix2x2 scale;
        scale(0, 0) = scaleVector.x;
        scale(1, 1) = scaleVector.y;

        return scale;
    }

    inline Matrix2x2 Rotate(float angle)
    {
        //c     s
        //-s    c
        //c = cos(angle)
        //s = sin(angle)

        float c = cos(angle * PI / 180.0f);
        float s = sin(angle * PI / 180.0f);
```

```

        Matrix2x2 result;

        //1st row
        result(0, 0) = c;
        result(0, 1) = s;

        //2nd row
        result(1, 0) = -s;
        result(1, 1) = c;

        return result;
    }

    inline double Determinant(const Matrix2x2& m)
    {
        return (double)m(0, 0) * m(1, 1) - (double)m(0, 1) * m(1, 0);
    }

    inline double Cofactor(const Matrix2x2& m, unsigned int row, unsigned int col)
    {
        //cij = ((-1)^i + j) * det of minor(i, j);
        double minor{ 0.0 };

        if (row == 0 && col == 0)
            minor = m(1, 1);
        else if (row == 0 && col == 1)
            minor = m(1, 0);
        else if (row == 1 && col == 0)
            minor = m(0, 1);
        else if (row == 1 && col == 1)
            minor = m(0, 0);

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

    inline Matrix2x2 Adjoint(const Matrix2x2& m)
    {
        //Cofactor of each ijth position put into matrix cA.
        //Adjoint is the tranposed matrix of cA.
        Matrix2x2 cofactorMatrix;
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
            }
        }

        return Transpose(cofactorMatrix);
    }

    inline Matrix2x2 Inverse(const Matrix2x2& m)
    {
        //Inverse of m = adjoint of m / det of m
        double det = Determinant(m);
        if (CompareDoubles(det, 0.0, EPSILON))
            return Matrix2x2();

        return Adjoint(m) * (1.0f / static_cast<float>(det));
    }


#if defined(_DEBUG)
    inline void print(const Matrix2x2& m)
    {
        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                std::cout « m(i, j) « " ";
            }

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


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






  //-------------------------------------------------------------------------------------------------------------
```

```
class Matrix3x3
{
public:

    Matrix3x3();

    Matrix3x3(float a[][3]);

    Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3);

    Matrix3x3(const Matrix2x2& m);

    Matrix3x3(const Matrix4x4& m);

    float* Data();

    const float* Data() const;

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

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

    Vector3D GetRow(unsigned int row) const;

    Vector3D GetCol(unsigned int col) const;

    void SetRow(unsigned int row, Vector3D v);

    void SetCol(unsigned int col, Vector3D v);

    Matrix3x3& operator=(const Matrix2x2& m);

    Matrix3x3& operator=(const Matrix4x4& m);

    Matrix3x3& operator+=(const Matrix3x3& m);

    Matrix3x3& operator-=(const Matrix3x3& m);

    Matrix3x3& operator*=(float k);

    Matrix3x3& operator*=(const Matrix3x3& m);

private:

    float mMat[3][3];
};

//------------------------------------------------------------------------------------
inline Matrix3x3::Matrix3x3()
{
    //1st row
    mMat[0][0] = 1.0f;
    mMat[0][1] = 0.0f;
    mMat[0][2] = 0.0f;

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

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

inline Matrix3x3::Matrix3x3(float a[][3])
{
    //1st row
    mMat[0][0] = a[0][0];
    mMat[0][1] = a[0][1];
    mMat[0][2] = a[0][2];

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

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

inline Matrix3x3::Matrix3x3(const Vector3D& r1, const Vector3D& r2, const Vector3D& r3)
{
    SetRow(0, r1);
```

```
        SetRow(1, r2);
        SetRow(2, r3);
    }

    inline float* Matrix3x3::Data()
    {
        return mMat[0];
    }

    inline const float* Matrix3x3::Data()const
{
        return mMat[0];
    }

    inline const float& Matrix3x3::operator()(unsigned int row, unsigned int col)const
{
        if (row > 2 || col > 2)
        {
            return mMat[0][0];
        }
        else
        {
            return mMat[row][col];
        }
    }

    inline float& Matrix3x3::operator()(unsigned int row, unsigned int col)
    {
        if (row > 2 || col > 2)
        {
            return mMat[0][0];
        }
        else
        {
            return mMat[row][col];
        }
    }

    inline Vector3D Matrix3x3::GetRow(unsigned int row)const
{
        if (row < 0 || row > 2)
            return Vector3D{ mMat[0][0], mMat[0][1], mMat[0][2] };
        else
            return Vector3D{ mMat[row][0], mMat[row][1], mMat[row][2] };

    }

    inline Vector3D Matrix3x3::GetCol(unsigned int col)const
{
        if (col < 0 || col > 2)
            return Vector3D{ mMat[0][0], mMat[1][0], mMat[2][0] };
        else
            return Vector3D{ mMat[0][col], mMat[1][col], mMat[2][col] };
    }

    inline void Matrix3x3::SetRow(unsigned int row, Vector3D v)
    {
        if (row > 2)
        {
            mMat[0][0] = v.x;
            mMat[0][1] = v.y;
            mMat[0][2] = v.z;
        }
        else
        {
            mMat[row][0] = v.x;
            mMat[row][1] = v.y;
            mMat[row][2] = v.z;
        }
    }

    inline void Matrix3x3::SetCol(unsigned int col, Vector3D v)
    {
        if (col > 2)
        {
            mMat[0][0] = v.x;
            mMat[1][0] = v.y;
            mMat[2][0] = v.z;
        }
        else
        {
            mMat[0][col] = v.x;
            mMat[1][col] = v.y;
            mMat[2][col] = v.z;
        }
    }

```

```
inline Matrix3x3& Matrix3x3::operator+=(const Matrix3x3& m)
{
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            this->mMat[i][j] += m.mMat[i][j];
        }
    }

    return *this;
}

inline Matrix3x3& Matrix3x3::operator-=(const Matrix3x3& m)
{
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            this->mMat[i][j] -= m.mMat[i][j];
        }
    }

    return *this;
}

inline Matrix3x3& Matrix3x3::operator*=(float k)
{
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            this->mMat[i][j] *= k;
        }
    }

    return *this;
}

inline Matrix3x3& Matrix3x3::operator*=(const Matrix3x3& m)
{
    Matrix3x3 result;

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

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

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

    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            mMat[i][j] = result.mMat[i][j];
        }
    }

    return *this;
}

inline Matrix3x3 operator+(const Matrix3x3& m1, const Matrix3x3& m2)
{
    Matrix3x3 result;
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            result(i, j) = m1(i, j) + m2(i, j);
        }
    }

    return result;
}

inline Matrix3x3 operator-(const Matrix3x3& m)
```

```
    {
        Matrix3x3 result;
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                result(i, j) = -m(i, j);
            }
        }

        return result;
    }

    inline Matrix3x3 operator-(const Matrix3x3& m1, const Matrix3x3& m2)
    {
        Matrix3x3 result;
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                result(i, j) = m1(i, j) - m2(i, j);
            }
        }

        return result;
    }

    inline Matrix3x3 operator*(const Matrix3x3& m, const float& k)
    {
        Matrix3x3 result;
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                result(i, j) = m(i, j) * k;
            }
        }

        return result;
    }

    inline Matrix3x3 operator*(const float& k, const Matrix3x3& m)
    {
        Matrix3x3 result;
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                result(i, j) = k * m(i, j);
            }
        }

        return result;
    }

    inline Matrix3x3 operator*(const Matrix3x3& m1, const Matrix3x3& m2)
    {
        Matrix3x3 result;

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

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

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

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

        return result;
    }

    inline Vector3D operator*(const Matrix3x3& m, const Vector3D& v)
    {
```

```
        Vector3D result;

        result.x = m(0, 0) * v.x + m(0, 1) * v.y + m(0, 2) * v.z;

        result.y = m(1, 0) * v.x + m(1, 1) * v.y + m(1, 2) * v.z;

        result.z = m(2, 0) * v.x + m(2, 1) * v.y + m(2, 2) * v.z;

        return result;
    }

    inline Vector3D operator*(const Vector3D& v, const Matrix3x3& m)
    {
        Vector3D result;

        result.x = v.x * m(0, 0) + v.y * m(1, 0) + v.z * m(2, 0);

        result.y = v.x * m(0, 1) + v.y * m(1, 1) + v.z * m(2, 1);

        result.z = v.x * m(0, 2) + v.y * m(1, 2) + v.z * m(2, 2);

        return result;
    }

    inline void SetToIdentity(Matrix3x3& m)
    {
        //set to identity matrix by setting the diagonals to 1.0f and all other elements to 0.0f

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

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

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

    inline bool Identity(const Matrix3x3& m)
    {
        //Is the identity matrix if the diagonals are equal to 1.0f and all other elements equals to
    0.0f

        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                if (i == j)
                {
                    if (!CompareFloats(m(i, j), 1.0f, EPSILON))
                        return false;

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

                }
            }
        }
    }

    inline Matrix3x3 Transpose(const Matrix3x3& m)
    {
        //make the rows into cols

        Matrix3x3 result;

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

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

        //3rd col = 3rd row
        result(0, 2) = m(2, 0);
```

```
        result(1, 2) = m(2, 1);
        result(2, 2) = m(2, 2);

        return result;
    }

    inline Matrix3x3 Scale(float x, float y, float z)
    {
        //x 0 0
        //0 y 0
        //0 0 z

        Matrix3x3 scale;
        scale(0, 0) = x;
        scale(1, 1) = y;
        scale(2, 2) = z;

        return scale;
    }

    inline Matrix3x3 Scale(const Vector3D& scaleVector)
    {
        //x 0 0
        //0 y 0
        //0 0 z

        Matrix3x3 scale;
        scale(0, 0) = scaleVector.x;
        scale(1, 1) = scaleVector.y;
        scale(2, 2) = scaleVector.z;

        return scale;
    }

    inline Matrix3x3 Rotate(float angle, float x, float y, float z)
    {

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

        Vector3D axis{ x, y, z };
        axis = Normalize(axis);
        x = axis.x;
        y = axis.y;
        z = axis.z;

        float c = cos(angle * PI / 180.0f);
        float s = sin(angle * PI / 180.0f);
        float oneMinusC = 1.0f - c;

        Matrix3x3 result;

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

        //2nd row
        result(1, 0) = (oneMinusC * (x * y)) - (s * z);
        result(1, 1) = c + (oneMinusC * (y * y));
        result(1, 2) = (oneMinusC * (y * z)) + (s * x);

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

        return result;
    }

    inline Matrix3x3 Rotate(float angle, const Vector3D& axis)
    {

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

        Vector3D nAxis(Normalize(axis));
        float x = nAxis.x;
        float y = nAxis.y;
        float z = nAxis.z;

```

```
        float c = cos(angle * PI / 180.0f);
        float s = sin(angle * PI / 180.0f);
        float oneMinusC = 1.0f - c;

        Matrix3x3 result;

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

        //2nd row
        result(1, 0) = (oneMinusC * (x * y)) - (s * z);
        result(1, 1) = c + (oneMinusC * (y * y));
        result(1, 2) = (oneMinusC * (y * z)) + (s * x);

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

        return result;
    }

    inline double Determinant(const Matrix3x3& m)
    {
        //m00m11m22 - m00m12m21
        double c1 = (double)m(0, 0) * m(1, 1) * m(2, 2) - (double)m(0, 0) * m(1, 2) * m(2, 1);

        //m01m12m20 - m01m10m22
        double c2 = (double)m(0, 1) * m(1, 2) * m(2, 0) - (double)m(0, 1) * m(1, 0) * m(2, 2);

        //m02m10m21 - m02m11m20
        double c3 = (double)m(0, 2) * m(1, 0) * m(2, 1) - (double)m(0, 2) * m(1, 1) * m(2, 0);

        return c1 + c2 + c3;
    }

    inline double Cofactor(const Matrix3x3& m, unsigned int row, unsigned int col)
    {
        //cij = ((-1)^i + j) * det of minor(i, j);
        Matrix2x2 minor;
        int r{ 0 };
        int c{ 0 };

        //minor(i, j)
        for (int i = 0; i < 3; ++i)
        {
            if (i == row)
                continue;

            for (int j = 0; j < 3; ++j)
            {
                if (j == col)
                    continue;

                minor(r, c) = m(i, j);
                ++c;

            }
            c = 0;
            ++r;
        }

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

    inline Matrix3x3 Adjoint(const Matrix3x3& m)
    {
        //Cofactor of each ijth position put into matrix cA.
        //Adjoint is the tranposed matrix of cA.
        Matrix3x3 cofactorMatrix;
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                cofactorMatrix(i, j) = static_cast<float>(Cofactor(m, i, j));
            }
        }

        return Transpose(cofactorMatrix);
    }

    inline Matrix3x3 Inverse(const Matrix3x3& m)
    {
        //Inverse of m = adjoint of m / det of m
        double det = Determinant(m);
```

```
        if (CompareDoubles(det, 0.0, EPSILON))
            return Matrix3x3();

        return Adjoint(m) * (1.0f / static_cast<float>(det));
    }


#if defined(_DEBUG)
    inline void print(const Matrix3x3& m)
    {
        for (int i = 0; i < 3; ++i)
        {
            for (int j = 0; j < 3; ++j)
            {
                std::cout « m(i, j) « "\t";
            }

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


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




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

        Matrix4x4();

        Matrix4x4(float a[][4]);

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

        Matrix4x4(const Matrix2x2& m);

        Matrix4x4(const Matrix3x3& m);

        Matrix4x4& operator=(const Matrix2x2& m);

        Matrix4x4& operator=(const Matrix3x3& m);

        float* Data();

        const float* Data() const;

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

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

        Vector4D GetRow(unsigned int row) const;

        Vector4D GetCol(unsigned int col) const;

        void SetRow(unsigned int row, Vector4D v);

        void SetCol(unsigned int col, Vector4D v);

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

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

        Matrix4x4& operator*=(float k);

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

    private:

        float mMat[4][4];
    };

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

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

```
        mMat[1][1] = 1.0f;
        mMat[1][2] = 0.0f;
        mMat[1][3] = 0.0f;

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

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



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

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

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

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

    inline Matrix4x4::Matrix4x4(const Vector4D& r1, const Vector4D& r2, const Vector4D& r3, const
  Vector4D& r4)
    {
        SetRow(0, r1);
        SetRow(1, r2);
        SetRow(2, r3);
        SetRow(3, r4);
    }

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

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

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

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

```
    }

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

    }

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

    inline void Matrix4x4::SetRow(unsigned int row, Vector4D v)
    {
        if (row > 3)
        {
            mMat[0][0] = v.x;
            mMat[0][1] = v.y;
            mMat[0][2] = v.z;
            mMat[0][3] = v.w;
        }
        else
        {
            mMat[row][0] = v.x;
            mMat[row][1] = v.y;
            mMat[row][2] = v.z;
            mMat[row][3] = v.w;
        }
    }

    inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
    {
        if (col > 3)
        {
            mMat[0][0] = v.x;
            mMat[1][0] = v.y;
            mMat[2][0] = v.z;
            mMat[3][0] = v.w;
        }
        else
        {
            mMat[0][col] = v.x;
            mMat[1][col] = v.y;
            mMat[2][col] = v.z;
            mMat[3][col] = v.w;
        }
    }

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

        return *this;
    }

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

        return *this;
    }

    inline Matrix4x4& Matrix4x4::operator*=(float k)
    {
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
```

```
                this->mMat[i][j] *= k;
            }
        }

        return *this;
    }

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

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

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

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

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

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

        return *this;
    }

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

        return result;
    }

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

        return result;
    }

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

```

        return result;
    }

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

        return result;
    }

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

        return result;
    }

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

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

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

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

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

        return result;
    }

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

        result.x = m(0, 0) * v.x + m(0, 1) * v.y + m(0, 2) * v.z + m(0, 3) * v.w;

        result.y = m(1, 0) * v.x + m(1, 1) * v.y + m(1, 2) * v.z + m(1, 3) * v.w;

        result.z = m(2, 0) * v.x + m(2, 1) * v.y + m(2, 2) * v.z + m(2, 3) * v.w;

        result.w = m(3, 0) * v.x + m(3, 1) * v.y + m(3, 2) * v.z + m(3, 3) * v.w;

        return result;
    }

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

        result.x = v.x * m(0, 0) + v.y * m(1, 0) + v.z * m(2, 0) + v.w * m(3, 0);

```

```
        result.y = v.x * m(0, 1) + v.y * m(1, 1) + v.z * m(2, 1) + v.w * m(3, 1);

        result.z = v.x * m(0, 2) + v.y * m(1, 2) + v.z * m(2, 2) + v.w * m(3, 2);

        result.w = v.x * m(0, 3) + v.y * m(1, 3) + v.z * m(2, 3) + v.w * m(3, 3);

        return result;
    }

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

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

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

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

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

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

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

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

                }
            }
        }
    }

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

        Matrix4x4 result;

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

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

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

```

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

        return result;
    }

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

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

        return translate;
    }

    inline Matrix4x4 Translate(const Vector3D& translateVector)
    {
        //1 0 0 0
        //0 1 0 0
        //0 0 1 0
        //x y z 1

        Matrix4x4 translate;
        translate(3, 0) = translateVector.x;
        translate(3, 1) = translateVector.y;
        translate(3, 2) = translateVector.z;

        return translate;
    }

    inline Matrix4x4 Scale4x4(float x, float y, float z)
    {
        //x 0 0 0
        //0 y 0 0
        //0 0 z 0
        //0 0 0 1

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

        return scale;
    }

    inline Matrix4x4 Scale4x4(const Vector3D& scaleVector)
    {
        //x 0 0 0
        //0 y 0 0
        //0 0 z 0
        //0 0 0 1

        Matrix4x4 scale;
        scale(0, 0) = scaleVector.x;
        scale(1, 1) = scaleVector.y;
        scale(2, 2) = scaleVector.z;

        return scale;
    }

    inline Matrix4x4 Rotate4x4(float angle, float x, float y, float z)
    {
        //c + (1 - c)x^2    (1 - c)xy + sz  (1 - c)xz - sy  0
        //(1 - c)xy - sz    c + (1 - c)y^2  (1 - c)yz + sx  0
        //(1 - c)xz + sy    (1 - c)yz - sx  c + (1 - c)z^2  0
        //0                 0               0               1
        //c = cos(angle)
        //s = sin(angle)

        Vector3D axis{ x, y, z };

        axis = Normalize(axis);

        x = axis.x;
        y = axis.y;
        z = axis.z;

        float c = cos(angle * PI / 180.0f);
```

```
        float s = sin(angle * PI / 180.0f);
        float oneMinusC = 1 - c;

        Matrix4x4 result;

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

        //2nd row
        result(1, 0) = (oneMinusC * (x * y)) - (s * z);
        result(1, 1) = c + (oneMinusC * (y * y));
        result(1, 2) = (oneMinusC * (y * z)) + (s * x);

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

        return result;
    }

    inline Matrix4x4 Rotate4x4(float angle, const Vector3D& axis)
    {
        //c + (1 - c)x^2     (1 - c)xy + sz   (1 - c)xz - sy  0
        //(1 - c)xy - sz     c + (1 - c)y^2   (1 - c)yz + sx  0
        //(1 - c)xz + sy     (1 - c)yz - sx   c + (1 - c)z^2  0
        //0                  0                0               1
        //c = cos(angle)
        //s = sin(angle)

        Vector3D nAxis(Normalize(axis));

        float x = nAxis.x;
        float y = nAxis.y;
        float z = nAxis.z;

        float c = cos(angle * PI / 180.0f);
        float s = sin(angle * PI / 180.0f);
        float oneMinusC = 1 - c;

        Matrix4x4 result;

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

        //2nd row
        result(1, 0) = (oneMinusC * (x * y)) - (s * z);
        result(1, 1) = c + (oneMinusC * (y * y));
        result(1, 2) = (oneMinusC * (y * z)) + (s * x);

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

        return result;
    }

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

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

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

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

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

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

```

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

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

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

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

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

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

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

    inline double Cofactor(const Matrix4x4& m, unsigned int row, unsigned int col)
    {
        //cij = (-1)^i + j * det of minor(i, j);
        Matrix3x3 minor;
        int r{ 0 };
        int c{ 0 };

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

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

                minor(r, c) = m(i, j);
                ++c;

            }
            c = 0;
            ++r;
        }


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

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

        return Transpose(cofactorMatrix);
    }

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

        return Adjoint(m) * (1.0f / static_cast<float>(det));
    }


#if defined(_DEBUG)
    inline void print(const Matrix4x4& m)
```

```
    {
        for (int i = 0; i < 4; ++i)
        {
            for (int j = 0; j < 4; ++j)
            {
                std::cout « m(i, j) « " ";
            }

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

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




//------------------------------------------------------------------------------------------------------------------------
//QUATERNION

    struct Quaternion
    {
        float scalar = 1.0f;
        Vector3D vector;
    };

    inline void operator+=(Quaternion& q1, const Quaternion& q2)
    {
        q1.scalar += q2.scalar;
        q1.vector += q2.vector;
    }

    inline void operator-=(Quaternion& q1, const Quaternion& q2)
    {
        q1.scalar -= q2.scalar;
        q1.vector -= q2.vector;
    }

    inline void operator*=(Quaternion& q1, float k)
    {
        q1.scalar *= k;
        q1.vector *= k;
    }

    inline void operator*=(Quaternion& q1, const Quaternion& q2)
    {
        //q1q2 = [w1, v1][w2, v2] = [w1w2 - v1 dot v2, w1v2 + w2v1 + v1 x v2]
        //w is the scalar component and v is the vector component

        q1.scalar = q1.scalar * q2.scalar - DotProduct(q1.vector, q2.vector);
        q1.vector = q1.scalar * q2.vector + q2.scalar * q1.vector + CrossProduct(q1.vector, q2.vector);
    }

    inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion{ q1.scalar + q2.scalar, q1.vector + q2.vector };
    }

    inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion{ q1.scalar - q2.scalar, q1.vector - q2.vector };
    }

    inline Quaternion operator-(const Quaternion& q)
    {
        return Quaternion{ -q.scalar, -q.vector};
    }

    inline Quaternion operator*(const Quaternion& q, float k)
    {
        return Quaternion{ q.scalar * k, q.vector * k };
    }

    inline Quaternion operator*(float k, const Quaternion& q)
    {
        return Quaternion{ k * q.scalar, k * q.vector };
    }

    inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
    {
        //q1q2 = [w1, v1][w2, v2] = [w1w2 - v1 dot v2, w1v2 + w2v1 + v1 x v2]
        //w is the scalar component and v is the vector component

        float scalarResult{ 0.0f };
        Vector3D vectorResult;
```

```
        scalarResult = q1.scalar * q2.scalar - DotProduct(q1.vector, q2.vector);
        vectorResult = q1.scalar * q2.vector + q2.scalar * q1.vector + CrossProduct(q1.vector, q2.vector);

        return Quaternion{ scalarResult, vectorResult };
    }

    inline bool operator==(const Quaternion& q1, const Quaternion& q2)
    {
        return CompareFloats(q1.scalar, q2.scalar, EPSILON) && (q1.vector == q2.vector);
    }

    inline bool operator!=(const Quaternion& q1, const Quaternion& q2)
    {
        return !(q1 == q2);
    }

    inline bool ZeroQuaternion(const Quaternion& q)
    {
        //zero quaternion = (0, 0, 0, 0)
        return CompareFloats(q.scalar, 0.0f, EPSILON) && ZeroVector(q.vector);
    }

    inline bool Identity(const Quaternion& q)
    {
        //identity quaternion = (1, 0, 0, 0)
        return CompareFloats(q.scalar, 1.0f, EPSILON) && CompareFloats(q.vector.x, 0.0f, EPSILON) &&
            CompareFloats(q.vector.y, 0.0f, EPSILON) && CompareFloats(q.vector.z, 0.0f, EPSILON);
    }

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

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

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

        if (ZeroQuaternion(q))
            return q;

        float inverseMagnitdue{ 1.0f / Length(q) };

        return Quaternion{ q.scalar * inverseMagnitdue, q.vector * inverseMagnitdue };
    }

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

        if (ZeroQuaternion(q))
            return q;

        float inverseMagnitdue{ 1.0f / Length(q) };

        return Quaternion{ Conjugate(q) * inverseMagnitdue };
    }

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

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

        Vector3D axis{ x, y, z };
        axis = Normalize(axis);

        return Quaternion{ c, s * axis };
    }

    inline Quaternion RotationQuaternion(float angle, const Vector3D& axis)
    {
```

```
        //A roatation quaternion is a quaternion where the
        //scalar part = cos(theta / 2)
        //vector part = sin(theta / 2) * axis
        //the axis needs to be normalized

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

        Vector3D axisN(Normalize(axis));

        return Quaternion{ c, s * axisN };
    }

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

        float angle{ angAxis.x / 2.0f };
        float c{ cos(angle * PI / 180.0f) };
        float s{ sin(angle * PI / 180.0f) };

        Vector3D axis{ angAxis.y, angAxis.z, angAxis.w };
        axis = Normalize(axis);

        return Quaternion{ c, s * axis };
    }

    inline Vector3D Rotate(const Quaternion& q, const Vector3D& p)
    {
        //To rotate a point/vector using quaternions you do qpq*, where p = (0, x, y, z) is the point/vector in quaternion from, q is a rotation quaternion
        //and q* is its conjugate.

        Quaternion point{ 0.0f, p };

        Quaternion result(q * point * Conjugate(q));

        //The rotated vector/point is in the vector component of the quaternion.
        return result.vector;
    }

    inline Vector4D Rotate(const Quaternion& q, const Vector4D& p)
    {
        //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
        //and q* is its conjugate.

        Quaternion point{ 0.0f, p.x, p.y, p.z };

        Quaternion result(q * point * Conjugate(q));

        //The rotated vector/point is in the vector component of the quaternion.
        return Vector4D{ result.vector.x, result.vector.y, result.vector.z, p.w };
    }

    inline Matrix3x3 QuaternionToRotationMatrixCol3x3(const Quaternion& q)
    {
        //1 - 2q3^2 - 2q4^2     2q2q3 - 2q1q4       2q2q4 + 2q1q3
        //2q2q3 + 2q1q4         1 - 2q2^2 - 2q4^2   2q3q4 - 2q1q2
        //2q2q4 - 2q1q3         2q3q4 + 2q1q2       1 - 2q2^2 - 2q3^2
        //q1 = scalar
        //q2 = x
        //q3 = y
        //q4 = z

        Matrix3x3 colMat;

        colMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
        colMat(0, 1) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
        colMat(0, 2) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;

        colMat(1, 0) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
        colMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
        colMat(1, 2) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;

        colMat(2, 0) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
        colMat(2, 1) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
        colMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;

        return colMat;
    }

    inline Matrix3x3 QuaternionToRotationMatrixRow3x3(const Quaternion& q)
```

```
{
    //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3
    //2q2q3 - 2q1q4          1 - 2q2^2 - 2q4^2  2q3q4 + 2q1q2
    //2q2q4 + 2q1q3          2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2
    //q1 = scalar
    //q2 = x
    //q3 = y
    //q4 = z

    Matrix3x3 rowMat;

    rowMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
    rowMat(0, 1) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
    rowMat(0, 2) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;

    rowMat(1, 0) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
    rowMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
    rowMat(1, 2) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;

    rowMat(2, 0) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
    rowMat(2, 1) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
    rowMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;

    return rowMat;
}

inline Matrix4x4 QuaternionToRotationMatrixCol4x4(const Quaternion& q)
{
    //1 - 2q3^2 - 2q4^2      2q2q3 - 2q1q4      2q2q4 + 2q1q3      0
    //2q2q3 + 2q1q4          1 - 2q2^2 - 2q4^2  2q3q4 - 2q1q2      0
    //2q2q4 - 2q1q3          2q3q4 + 2q1q2      1 - 2q2^2 - 2q3^2  0
    //0                      0                  0                  1
    //q1 = scalar
    //q2 = x
    //q3 = y
    //q4 = z

    Matrix4x4 colMat;

    colMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
    colMat(0, 1) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
    colMat(0, 2) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;

    colMat(1, 0) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
    colMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
    colMat(1, 2) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;

    colMat(2, 0) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;
    colMat(2, 1) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;
    colMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;

    return colMat;
}

inline Matrix4x4 QuaternionToRotationMatrixRow4x4(const Quaternion& q)
{
    //1 - 2q3^2 - 2q4^2      2q2q3 + 2q1q4      2q2q4 - 2q1q3      0
    //2q2q3 - 2q1q4          1 - 2q2^2 - 2q4^2  2q3q4 + 2q1q2      0
    //2q2q4 + 2q1q3          2q3q4 - 2q1q2      1 - 2q2^2 - 2q3^2  0
    //0                      0                  0                  1
    //q1 = scalar
    //q2 = x
    //q3 = y
    //q4 = z

    Matrix4x4 rowMat;

    rowMat(0, 0) = 1.0f - 2.0f * q.vector.y * q.vector.y - 2.0f * q.vector.z * q.vector.z;
    rowMat(0, 1) = 2.0f * q.vector.x * q.vector.y + 2.0f * q.scalar * q.vector.z;
    rowMat(0, 2) = 2.0f * q.vector.x * q.vector.z - 2.0f * q.scalar * q.vector.y;

    rowMat(1, 0) = 2.0f * q.vector.x * q.vector.y - 2.0f * q.scalar * q.vector.z;
    rowMat(1, 1) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.z * q.vector.z;
    rowMat(1, 2) = 2.0f * q.vector.y * q.vector.z + 2.0f * q.scalar * q.vector.x;

    rowMat(2, 0) = 2.0f * q.vector.x * q.vector.z + 2.0f * q.scalar * q.vector.y;
    rowMat(2, 1) = 2.0f * q.vector.y * q.vector.z - 2.0f * q.scalar * q.vector.x;
    rowMat(2, 2) = 1.0f - 2.0f * q.vector.x * q.vector.x - 2.0f * q.vector.y * q.vector.y;

    return rowMat;
}

inline float DotProduct(const Quaternion& q1, const Quaternion& q2)
{
    //q1 dot q2 = [w1, v1] dot [w2, v2] = w1w2 + v1 dot v2
    //w is the scalar component and v is the vector component.

```

```
        return q1.scalar * q2.scalar + DotProduct(q1.vector, q2.vector);
    }

    inline Quaternion Lerp(const Quaternion& q1, const Quaternion& q2, float t)
    {
        if (t < 0.0f)
            return q1;
        else if (t > 1.0f)
            return q2;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q1, q2);

        Quaternion newQ2;
        //If the dot product is negative, negate q2 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ2 = -q2;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ2 = q2;
        }

        return (1.0f - t) * q1 + t * newQ2;
    }

    inline Quaternion NLerp(const Quaternion& q1, const Quaternion& q2, float t)
    {
        if (t < 0.0f)
            return q1;
        else if (t > 1.0f)
            return q2;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q1, q2);

        Quaternion newQ2;
        //If the dot product is negative, negate q2 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ2 = -q2;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ2 = q2;
        }

        return Normalize((1.0f - t) * q1 + t * newQ2);
    }

    inline Quaternion Slerp(const Quaternion& q1, const Quaternion& q2, float t)
    {
        //Formula used is
        //k1 = sin((1 - t)omega) * omega) / sin(omega);
        //k2 = (sin(tomega) * omega) / sin(omega)
        //newQ = k1q1 * k2q2
        //Omega is the angle between the q0 and q1.

        if (t < 0.0f)
            return q1;
        else if (t > 1.0f)
            return q2;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q1, q2);

        Quaternion newQ2;
        //If the dot product is negative, negate q2 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ2 = -q2;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ2 = q2;
        }

        float k1{ 0.0f };
        float k2{ 0.0f };

        //Linear interpolate if the quaternions are very close to protect dividing by zero.
        if (cosOmega > 0.9999f)
        {
```

```
            k1 = 1.0f - t;
            k2 = t;
        }
        else
        {
            //sin of the angle between the quaternions is
            //sin(omega) = 1 - cos^2(omega) from the trig identity
            //sin^2(omega) + cos^2(omega) = 1.
            float sinOmega{ sqrt(1.0f - cosOmega * cosOmega) };

            //retrieve the angle
            float omega{ atan2(sinOmega, cosOmega) };

            //Compute inverse to avoid dividng multiple times
            float oneOverSinOmega{ 1.0f / sinOmega };

            k1 = sin((1.0f - t) * omega) * oneOverSinOmega;
            k2 = sin(t * omega) * oneOverSinOmega;
        }

        return k1 * q1 + k2 * newQ2;
    }

#if defined(_DEBUG)
    inline void print(const Quaternion& q)
    {
        std::cout « "(" « q.scalar « ", " « q.vector.x « ", " « q.vector.y « ", " « q.vector.z « ")";
    }
#endif


 //-----------------------------------------------------------------------------------------------------------------------
}

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;
```

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