Farouq Adepetu's Math Engine

Generated by Doxygen 1.9.4

Chapter 1

Namespace Index

1.1 Namespace List

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

Namespace Documentation

4.1 FAMath Namespace Reference

Has the classes Vector2D, Vector3D, Vector4D, Matrix2x2, Matrix3x3, Matrix4x4, Quaternion, and utility functions.

Classes

- class Matrix2x2

  A matrix class used for 2x2 matrices and their manipulations.
- class Matrix3x3

  A matrix class used for 3x3 matrices and their manipulations.
- class Matrix4x4

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

  A quaternion class used for quaternions and their manipulations.
- class Vector2D

  A vector class used for 2D vectors/points and their manipulations.
- class Vector3D

  A vector class used for 3D vectors/points and their manipulations.
- class Vector4D

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

Functions

- bool CompareFloats (float x, float y, float epsilon)

  Returns true if x and y are equal.
- bool CompareDoubles (double x, double y, double epsilon)

  Returns true if x and y are equal.
- bool ZeroVector (const Vector2D &a)

  Returns true if a is the zero vector.
- Vector2D operator+ (const Vector2D &a, const Vector2D &b)

  Adds a with b and returns the result.
- Vector2D operator- (const Vector2D &v)

  Negates the vector v and returns the result.

- Vector2D operator- (const Vector2D &a, const Vector2D &b)

  Subtracts b from a and returns the result.
- Vector2D operator* (const Vector2D &a, float k)

  *Returns a * k.*
- Vector2D operator* (float k, const Vector2D &a)

  *Returns k * a.*
- Vector2D operator/ (const Vector2D &a, const float &k)

  Returns a / k. If k = 0 the returned vector is the zero vector.
- bool operator== (const Vector2D &a, const Vector2D &b)

  Returns true if a equals to b, false otherwise.
- bool operator!= (const Vector2D &a, const Vector2D &b)

  Returns true if a does not equal to b, false otherwise.
- float DotProduct (const Vector2D &a, const Vector2D &b)

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

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

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

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

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

  Returns a 2D vector that is the projection of a onto b. If b is the zero vector a is returned.
- Vector2D Lerp (const Vector2D &start, const Vector2D &end, float t)

  Linear interpolate between the two vectors start and end.
- bool ZeroVector (const Vector3D &a)

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

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

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

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

  *Returns a * k.*
- Vector3D operator* (float k, const Vector3D &a)

  *Returns k * a.*
- Vector3D operator/ (const Vector3D &a, float k)

  Returns a / k.
- bool operator== (const Vector3D &a, const Vector3D &b)

  Returns true if a equals to b, false otherwise.
- bool operator!= (const Vector3D &a, const Vector3D &b)

  Returns true if a does not equal to b, false otherwise.
- float DotProduct (const Vector3D &a, const Vector3D &b)

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

  Returns the cross product between a and b.

- float Length (const Vector3D &v)

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

  Normalizes the 3D vector v.
- Vector3D CylindricalToCartesian (const Vector3D &v)

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

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

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

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

  Returns a 3D vector that is the projection of a onto b.
- void Orthonormalize (Vector3D &x, Vector3D &y, Vector3D &z)

  Orthonormalizes the specified vectors.
- Vector3D Lerp (const Vector3D &start, const Vector3D &end, float t)

  Linear interpolate between the two vectors start and end.
- bool ZeroVector (const Vector4D &a)

  Returns true if a is the zero vector.
- Vector4D operator+ (const Vector4D &a, const Vector4D &b)

  Adds a with b and returns the result.
- Vector4D operator- (const Vector4D &v)

  Negatives v and returns the result.
- Vector4D operator- (const Vector4D &a, const Vector4D &b)

  Subtracts b from a and returns the result.
- Vector4D operator* (const Vector4D &a, float k)

  *Returns a * k.*
- Vector4D operator* (float k, const Vector4D &a)

  *Returns k * a.*
- Vector4D operator/ (const Vector4D &a, float k)

  Returns a / k.
- bool operator== (const Vector4D &a, const Vector4D &b)

  Returns true if a equals to b, false otherwise.
- bool operator!= (const Vector4D &a, const Vector4D &b)

  Returns true if a does not equal to b, false otherwise.
- float DotProduct (const Vector4D &a, const Vector4D &b)

  Returns the dot product between a and b.
- float Length (const Vector4D &v)

  Returns the length(magnitude) of the 4D vector v.
- Vector4D Norm (const Vector4D &v)

  Normalizes the 4D vector v.
- Vector4D Projection (const Vector4D &a, const Vector4D &b)

  Returns a 4D vector that is the projection of a onto b.
- void Orthonormalize (Vector4D &x, Vector4D &y, Vector4D &z)

  Orthonormalizes the specified vectors.
- Vector4D Lerp (const Vector4D &start, const Vector4D &end, float t)

  Linear interpolate between the two vectors start and end.
- Matrix2x2 operator+ (const Matrix2x2 &m1, const Matrix2x2 &m2)

  Adds m1 with m2 and returns the result.
- Matrix2x2 operator- (const Matrix2x2 &m)

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

- bool IsIdentity (const Matrix3x3 &m)

    Returns true if m is the identity matrix, false otherwise.
- Matrix3x3 Transpose (const Matrix3x3 &m)

    Returns the tranpose of the given matrix m.
- Matrix3x3 Scale (const Matrix3x3 &cm, float x, float y, float z)

    Construct a 3x3 scaling matrix with x, y, z and post-multiplies it by cm.
- Matrix3x3 Scale (const Matrix3x3 &cm, const Vector3D &scaleVector)

    Construct a 3x3 scaling matrix with scaleVector and post-multiplies it by cm.
- Matrix3x3 Rotate (const Matrix3x3 &cm, float angle, float x, float y, float z)

    Construct a 3x3 rotation matrix with angle (in degrees) and axis (x, y, z) and post-multiplies it by cm.
- Matrix3x3 Rotate (const Matrix3x3 &cm, float angle, const Vector3D &axis)

    Construct a 3x3 rotation matrix with angle (in degrees) and axis and post-multiplies it by cm.
- double Determinant (const Matrix3x3 &m)

    Returns the determinant m.
- double Cofactor (const Matrix3x3 &m, unsigned int row, unsigned int col)

    Returns the cofactor of the row and col in m.
- Matrix3x3 Adjoint (const Matrix3x3 &m)

    Returns the adjoint of m.
- Matrix3x3 Inverse (const Matrix3x3 &m)

    Returns the inverse of m.
- Matrix4x4 operator+ (const Matrix4x4 &m1, const Matrix4x4 &m2)

    Adds m1 with m2 and returns the result.
- Matrix4x4 operator- (const Matrix4x4 &m)

    Negates the 4x4 matrix m.
- Matrix4x4 operator- (const Matrix4x4 &m1, const Matrix4x4 &m2)

    Subtracts m2 from m1 and returns the result.
- Matrix4x4 operator∗ (const Matrix4x4 &m, const float &k)

    Multiplies m with k and returns the result.
- Matrix4x4 operator∗ (const float &k, const Matrix4x4 &m)

    Multiplies k with \m and returns the result.
- Matrix4x4 operator∗ (const Matrix4x4 &m1, const Matrix4x4 &m2)

    Multiplies m1 with \m2 and returns the result.
- Vector4D operator∗ (const Matrix4x4 &m, const Vector4D &v)

    Multiplies m with v and returns the result.
- Vector4D operator∗ (const Vector4D &v, const Matrix4x4 &m)

    Multiplies v with m and returns the result.
- void SetToIdentity (Matrix4x4 &m)

    Sets m to the identity matrix.
- bool IsIdentity (const Matrix4x4 &m)

    Returns true if m is the identity matrix, false otherwise.
- Matrix4x4 Transpose (const Matrix4x4 &m)

    Returns the tranpose of the given matrix m.
- Matrix4x4 Translate (const Matrix4x4 &cm, float x, float y, float z)

    Constructs a 4x4 translation matrix with x, y, z and post-multiplies it by cm.
- Matrix4x4 Translate (const Matrix4x4 &cm, const Vector3D &translateVector)

    Constructs a 4x4 translation matrix with the x, y and z values of translateVector and post-multiplies it by cm.
- Matrix4x4 Scale (const Matrix4x4 &cm, float x, float y, float z)

    Construct a 4x4 scaling matrix with x, y, z and post-multiplies it by cm.
- Matrix4x4 Scale (const Matrix4x4 &cm, const Vector3D &scaleVector)

    Construct a 4x4 scaling matrix with the x, y and z values of the scaleVector and post-multiplies it by cm.
- Matrix4x4 Rotate (const Matrix4x4 &cm, float angle, float x, float y, float z)

Construct a 4x4 rotation matrix with angle (in degrees) and axis (x, y, z) and post-multiplies it by cm.
- Matrix4x4 Rotate (const Matrix4x4 &cm, float angle, const Vector3D &axis)

  Construct a 4x4 rotation matrix with angle (in degrees) and axis and post-multiplies it by cm.
- double Determinant (const Matrix4x4 &m)

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

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

  Returns the adjoint of m.
- Matrix4x4 Inverse (const Matrix4x4 &m)

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

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

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

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

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

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

  *Returns a quaternion that has the result of q1 * q2.*
- bool operator== (const Quaternion &q1, const Quaternion &q2)

  Returns true if q1 equals to q2, false otherwise.
- bool operator!= (const Quaternion &q1, const Quaternion &q2)

  Returns true if q1 does not equal to q2, false otherwise.
- bool IsZeroQuaternion (const Quaternion &q)

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

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

  Returns the conjugate of quaternion q.
- float Length (const Quaternion &q)

  Returns the length of quaternion q.
- Quaternion Normalize (const Quaternion &q)

  Normalizes q and returns the normalized quaternion.
- Quaternion Inverse (const Quaternion &q)

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

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

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

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

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

  Transforms q into a row-major matrix.
- Vector3D Rotate (const Quaternion &q, const Vector3D &p)

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

- Vector4D Rotate (const Quaternion &q, const Vector4D &p)

    Rotates the specified point/vector p using the quaternion q.
- float DotProduct (const Quaternion &q1, const Quaternion &q2)

    Returns the dot product of the quaternions q1 and q2.
- Quaternion Lerp (const Quaternion &q0, const Quaternion &q1, float t)

    Linear Interpolate between quaternions q0 and q1.
- Quaternion NLerp (const Quaternion &q0, const Quaternion &q1, float t)

    Normalize Linear Interpolate between quaternions q0 and q1.
- Quaternion Slerp (const Quaternion &q0, const Quaternion &q1, float t)

    Spherical Linear Interpolate between quaternions q0 and q1.

4.1.1 Detailed Description

Has the classes Vector2D, Vector3D, Vector4D, Matrix2x2, Matrix3x3, Matrix4x4, Quaternion, and utility functions.

4.1.2 Function Documentation

4.1.2.1 Adjoint() [1/3]

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

Returns the adjoint of *m*.

4.1.2.2 Adjoint() [2/3]

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

Returns the adjoint of *m*.

4.1.2.3 Adjoint() [3/3]

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

Returns the adjoint of *m*.

4.1.2.4 CartesianToCylindrical()

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

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

v should = (x, y, z).
If *vx* is zero then no conversion happens and *v* is returned.
The returned 3D vector = (r, theta(degrees), z).

4.1.2.5 CartesianToPolar()

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

Converts the 2D vector *v* from cartesian coordinates to polar coordinates. *v* should = (x, y) If *vx* is zero then no conversion happens and *v* is returned.
The returned 2D vector = (r, theta(degrees)).

4.1.2.6 CartesianToSpherical()

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

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

If *v* is the zero vector or if *vx* is zero then no conversion happens and *v* is returned.
The returned 3D vector = (r, phi(degrees), theta(degrees)).

4.1.2.7 Cofactor() [1/3]

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

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

4.1.2.8 Cofactor() [2/3]

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

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

4.1.2.9 Cofactor() [3/3]

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

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

4.1.2.10 CompareDoubles()

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

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

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

4.1.2.11 CompareFloats()

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

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

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

4.1.2.12 Conjugate()

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

Returns the conjugate of quaternion *q*.

4.1.2.13 CrossProduct()

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

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

4.1.2.14 CylindricalToCartesian()

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

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

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

4.1.2.15 Determinant() [1/3]

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

Returns the determinant *m*.

4.1.2.16 Determinant() [2/3]

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

Returns the determinant *m*.

4.1.2.17 Determinant() [3/3]

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

Returns the determinant *m*.

4.1.2.18 DotProduct() [1/4]

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

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

4.1.2.19 DotProduct() [2/4]

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

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

4.1.2.20 DotProduct() [3/4]

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

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

4.1.2.21 DotProduct() [4/4]

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

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

4.1.2.22 Inverse() [1/4]

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

Returns the inverse of *m*.

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

4.1.2.23 Inverse() [2/4]

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

Returns the inverse of *m*.

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

4.1.2.24 Inverse() [3/4]

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

Returns the inverse of *m*.

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

4.1.2.25 Inverse() [4/4]

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

Returns the invese of *q*.

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

4.1.2.26 IsIdentity() [1/4]

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

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

4.1.2.27 IsIdentity() [2/4]

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

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

4.1.2.28 IsIdentity() [3/4]

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

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

4.1.2.29 IsIdentity() [4/4]

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

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

4.1.2.30 IsZeroQuaternion()

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

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

4.1.2.31 Length() [1/4]

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

Returns the length of quaternion *q*.

4.1.2.32 Length() [2/4]

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

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

4.1.2.33 Length() [3/4]

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

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

4.1.2.34 Length() [4/4]

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

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

4.1.2.35 Lerp() [1/4]

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

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

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

4.1.2.36 Lerp() [2/4]

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

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

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

4.1.2.37 Lerp() [3/4]

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

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

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

4.1.2.38 Lerp() [4/4]

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

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

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

4.1.2.39 NLerp()

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

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

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

4.1.2.40 Norm() [1/3]

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

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

4.1.2.41 Norm() [2/3]

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

Normalizes the 3D vector *v*.

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

4.1.2.42 Norm() [3/3]

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

Normalizes the 4D vector *v*.

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

4.1.2.43 Normalize()

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

Normalizes *q* and returns the normalized quaternion.

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

4.1.2.44 operator"!=() [1/4]

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

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

4.1.2.45 operator"!=() [2/4]

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

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

4.1.2.46 operator"!=() [3/4]

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

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

4.1.2.47 operator"!=() [4/4]

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

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

4.1.2.48 operator*() [1/24]

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

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

4.1.2.49 operator*() [2/24]

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

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

4.1.2.50 operator*() [3/24]

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

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

4.1.2.51 operator*() [4/24]

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

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

4.1.2.52 operator*() [5/24]

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

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

The vector *v* is a column vector.

4.1.2.53 operator*() [6/24]

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

Multiplies *m1* with \m2 and returns the result.

Does $m1 * m2$ in that order.

4.1.2.54 operator*() [7/24]

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

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

4.1.2.55 operator*() [8/24]

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

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

The vector *v* is a column vector.

4.1.2.56 operator*() [9/24]

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

Multiplies *m1* with \m2 and returns the result.

Does *m1* ∗ *m2* in that order.

4.1.2.57 operator*() [10/24]

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

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

4.1.2.58 operator*() [11/24]

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

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

The vector *v* is a column vector.

4.1.2.59 operator*() [12/24]

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

Multiplies *m1* with \m2 and returns the result.

Does *m1* ∗ *m2* in that order.

4.1.2.60 operator*() [13/24]

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

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

4.1.2.61 operator*() [14/24]

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

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

4.1.2.62 operator*() [15/24]

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

Returns $a * k$.

4.1.2.63 operator*() [16/24]

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

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

The vector *v* is a row vector.

4.1.2.64 operator*() [17/24]

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

Returns $a * k$.

4.1.2.65 operator*() [18/24]

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

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

The vector *v* is a row vector.

4.1.2.66 operator*() [19/24]

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

Returns $a * k$.

4.1.2.67 operator*() [20/24]

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

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

The vector *v* is a row vector.

4.1.2.68 operator*() [21/24]

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

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

4.1.2.69 operator*() [22/24]

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

Returns $k * a$.

4.1.2.70 operator*() [23/24]

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

Returns $k * a$.

4.1.2.71 operator*() [24/24]

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

Returns $k * a$.

4.1.2.72 operator+() [1/7]

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

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

4.1.2.73 operator+() [2/7]

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

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

4.1.2.74 operator+() [3/7]

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

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

4.1.2.75 operator+() [4/7]

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

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

4.1.2.76 operator+() [5/7]

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

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

4.1.2.77 operator+() [6/7]

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

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

4.1.2.78 operator+() [7/7]

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

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

4.1.2.79 operator-() [1/14]

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

Negates the 2x2 matrix *m*.

4.1.2.80 operator-() [2/14]

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

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

4.1.2.81 operator-() [3/14]

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

Negates the 3x3 matrix *m*.

4.1.2.82 operator-() [4/14]

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

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

4.1.2.83 operator-() [5/14]

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

Negates the 4x4 matrix *m*.

4.1.2.84 operator-() [6/14]

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

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

4.1.2.85 operator-() [7/14]

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

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

4.1.2.86 operator-() [8/14]

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

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

4.1.2.87 operator-() [9/14]

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

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

4.1.2.88 operator-() [10/14]

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

Negates the vector *v* and returns the result.

4.1.2.89 operator-() [11/14]

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

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

4.1.2.90 operator-() [12/14]

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

Negates the vector *v* and returns the result.

4.1.2.91 operator-() [13/14]

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

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

4.1.2.92 operator-() [14/14]

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

Negatives *v* and returns the result.

4.1.2.93 operator/() [1/3]

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

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

4.1.2.94 operator/() [2/3]

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

Returns a / k.

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

4.1.2.95 operator/() [3/3]

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

Returns *a* / *k*.

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

4.1.2.96 operator==() [1/4]

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

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

4.1.2.97 operator==() [2/4]

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

Returns true if *a* equals to *b*, false otherwise.

4.1.2.98 operator==() [3/4]

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

Returns true if *a* equals to *b*, false otherwise.

4.1.2.99 operator==() [4/4]

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

Returns true if *a* equals to *b*, false otherwise.

4.1.2.100 Orthonormalize() [1/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

4.1.2.101 Orthonormalize() [2/2]

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

Orthonormalizes the specified vectors.

Uses Classical Gram-Schmidt.

4.1.2.102 PolarToCartesian()

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

Converts the 2D vector *v* from polar coordinates to cartesian coordinates. *v* should = (r, theta(degrees)) The returned 2D vector = (x, y)

4.1.2.103 Projection() [1/3]

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

Returns a 2D vector that is the projection of *a* onto *b*. If *b* is the zero vector *a* is returned.

4.1.2.104 Projection() [2/3]

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

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

If *b* is the zero vector a is returned.

4.1.2.105 Projection() [3/3]

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

Returns a 4D vector that is the projection of *a* onto *b*.

If *b* is the zero vector a is returned.

4.1.2.106 QuaternionToRotationMatrixCol()

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

Transforms *q* into a column-major matrix.

q should be a unit quaternion.

4.1.2.107 QuaternionToRotationMatrixRow()

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

Transforms *q* into a row-major matrix.

q should be a unit quaternion.

4.1.2.108 Rotate() [1/7]

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

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

Returns cm $*$ rotate.

4.1.2.109 Rotate() [2/7]

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

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

Returns cm $*$ rotate.

4.1.2.110 Rotate() [3/7]

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

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

Returns cm $*$ rotate.

4.1.2.111 Rotate() [4/7]

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

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

Returns cm $*$ rotate.

4.1.2.112 Rotate() [5/7]

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

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

Returns cm $*$ rotate.

4.1.2.113 Rotate() [6/7]

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

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

q should be a rotation quaternion.

4.1.2.114 Rotate() [7/7]

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

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

q should be a rotation quaternion.

4.1.2.115 RotationQuaternion() [1/3]

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

Returns a quaternion from the axis-angle rotation representation.

The x value in the 4D vector *v* should be the angle(in degrees).
The y, z and w value in the 4D vector *v* should be the axis.

4.1.2.116 RotationQuaternion() [2/3]

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

Returns a quaternion from the axis-angle rotation representation.

The *angle* should be given in degrees.

4.1.2.117 RotationQuaternion() [3/3]

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

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

The *angle* should be given in degrees.

4.1.2.118 Scale() [1/6]

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

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

Returns cm $*$ scale.

4.1.2.119 Scale() [2/6]

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

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

Returns cm ∗ scale.

4.1.2.120 Scale() [3/6]

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

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

Returns cm ∗ scale.

4.1.2.121 Scale() [4/6]

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

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

Returns cm ∗ scale.

4.1.2.122 Scale() [5/6]

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

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

Returns cm ∗ scale.

4.1.2.123 Scale() [6/6]

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

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

Returns cm ∗ scale.

4.1.2.124 SetToIdentity() [1/3]

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

Sets *m* to the identity matrix.

4.1.2.125 SetToIdentity() [2/3]

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

Sets *m* to the identity matrix.

4.1.2.126 SetToIdentity() [3/3]

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

Sets *m* to the identity matrix.

4.1.2.127 Slerp()

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

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

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

4.1.2.128 SphericalToCartesian()

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

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

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

4.1.2.129 Translate() [1/2]

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

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

Returns cm ∗ translate.

4.1.2.130 Translate() [2/2]

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

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

Returns cm ∗ translate.

4.1.2.131 Transpose() [1/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.132 Transpose() [2/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.133 Transpose() [3/3]

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

Returns the tranpose of the given matrix *m*.

4.1.2.134 ZeroVector() [1/3]

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

Returns true if *a* is the zero vector.

4.1.2.135 ZeroVector() [2/3]

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

Returns true if *a* is the zero vector.

4.1.2.136 ZeroVector() [3/3]

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

Returns true if *a* is the zero vector.

Chapter 5

Class Documentation

5.1 FAMath::Matrix2x2 Class Reference

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

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

Public Member Functions

- Matrix2x2 ()

  Creates a new 2x2 identity matrix.
- Matrix2x2 (float a[ ][2])

  Creates a new 2x2 matrix with elements initialized to the given 2D array.
- Matrix2x2 (const Vector2D &r1, const Vector2D &r2)

  Creates a new 2x2 matrix with each row being set to the specified rows.
- Matrix2x2 (const Matrix3x3 &m)

  Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 3x3 matrix.
- Matrix2x2 (const Matrix4x4 &m)

  Creates a new 2x2 matrix with each row being set to the first two values of the respective rows of the 4x4 matrix.
- float ∗ Data ()

  Returns a pointer to the first element in the matrix.
- const float ∗ Data () const

  Returns a constant pointer to the first element in the matrix.
- const float & operator() (unsigned int row, unsigned int col) const

  Returns a constant reference to the element at the given (row, col).
- float & operator() (unsigned int row, unsigned int col)

  Returns a reference to the element at the given (row, col).
- Vector2D GetRow (unsigned int row) const

  Returns the specified row.
- Vector2D GetCol (unsigned int col) const

  Returns the specified col.
- void SetRow (unsigned int row, Vector2D v)

  Sets each element in the given row to the components of vector v.
- void SetCol (unsigned int col, Vector2D v)

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

- Matrix2x2 & operator= (const Matrix3x3 &m)

  Sets the values each row to the first two values of the respective rows of the 3x3 matrix.
- Matrix2x2 & operator= (const Matrix4x4 &m)

  Sets the values each row to the first two values of the respective rows of the 4x4 matrix.
- Matrix2x2 & operator+= (const Matrix2x2 &m)

  Adds this 2x2 matrix with given matrix m and stores the result in this 2x2 matrix.
- Matrix2x2 & operator-= (const Matrix2x2 &m)

  Subtracts m from this 2x2 matrix stores the result in this 2x2 matrix.
- Matrix2x2 & operator*= (float k)

  Multiplies this 2x2 matrix with k and stores the result in this 2x2 matrix.
- Matrix2x2 & operator*= (const Matrix2x2 &m)

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

5.1.1 Detailed Description

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

The datatype for the components is float.
The 2x2 matrix is treated as a row-major matrix.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Matrix2x2() [1/5]

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

Creates a new 2x2 identity matrix.

5.1.2.2 Matrix2x2() [2/5]

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

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

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

5.1.2.3 Matrix2x2() [3/5]

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

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

5.1.2.4 Matrix2x2() [4/5]

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

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

5.1.2.5 Matrix2x2() [5/5]

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

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

5.1.3 Member Function Documentation

5.1.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.1.3.2 Data() [2/2]

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

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

5.1.3.3 GetCol()

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

Returns the specified *col*.

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

5.1.3.4 GetRow()

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

Returns the specified *row*.

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

5.1.3.5 operator()() [1/2]

```
float & FAMath::Matrix2x2::operator() (
            unsigned int row,
            unsigned int col ) [inline]
```

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

The *row* and *col* values should be between [0,1]. If any of them are out of that range, the first element will be returned.

5.1.3.6 operator()() [2/2]

```
const float & FAMath::Matrix2x2::operator() (
            unsigned int row,
            unsigned int col ) const [inline]
```

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

The *row* and *col* values should be between [0,1]. If any of them are out of that range, the first element will be returned.

5.1.3.7 operator*=() [1/2]

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

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

5.1.3.8 operator*=() [2/2]

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

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

5.1.3.9 operator+=()

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

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

5.1.3.10 operator-=()

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

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

5.1.3.11 operator=() [1/2]

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

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

5.1.3.12 operator=() [2/2]

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

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

5.1.3.13 SetCol()

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

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

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

5.1.3.14 SetRow()

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

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

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

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

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

5.2 FAMath::Matrix3x3 Class Reference

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

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

Public Member Functions

- Matrix3x3 ()

    Creates a new 3x3 identity matrix.
- Matrix3x3 (float a[ ][3])

    Creates a new 3x3 matrix with elements initialized to the given 2D array.
- Matrix3x3 (const Vector3D &r1, const Vector3D &r2, const Vector3D &r3)

    Creates a new 3x3 matrix with each row being set to the specified rows.
- Matrix3x3 (const Matrix2x2 &m)

    Creates a new 3x3 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.
- Matrix3x3 (const Matrix4x4 &m)

    Creates a new 3x3 matrix with each row being set to the first three values of the respective rows of the 4x4 matrix.
- float ∗ Data ()

    Returns a pointer to the first element in the matrix.
- const float ∗ Data () const

    Returns a constant pointer to the first element in the matrix.
- const float & operator() (unsigned int row, unsigned int col) const

    Returns a constant reference to the element at the given (row, col).
- float & operator() (unsigned int row, unsigned int col)

    Returns a reference to the element at the given (row, col).
- Vector3D GetRow (unsigned int row) const

    Returns the specified row.
- Vector3D GetCol (unsigned int col) const

    Returns the specified col.
- void SetRow (unsigned int row, Vector3D v)

    Sets each element in the given row to the components of vector v.
- void SetCol (unsigned int col, Vector3D v)

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

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

5.2.1 Detailed Description

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

The datatype for the components is float.
The 3x3 matrix is treated as a row-major matrix.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Matrix3x3() [1/5]

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

Creates a new 3x3 identity matrix.

5.2.2.2 Matrix3x3() [2/5]

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

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

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

5.2.2.3 Matrix3x3() [3/5]

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

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

5.2.2.4 Matrix3x3() [4/5]

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

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

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

5.2.2.5 Matrix3x3() [5/5]

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

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

5.2.3 Member Function Documentation

5.2.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.2.3.2 Data() [2/2]

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

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

5.2.3.3 GetCol()

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

Returns the specified *col*.

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

5.2.3.4 GetRow()

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

Returns the specified *row*.

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

5.2.3.5 operator()() [1/2]

```
float & FAMath::Matrix3x3::operator() (
             unsigned int row,
             unsigned int col )  [inline]
```

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

The *row* and *col* values should be between [0,2]. If any of them are out of that range, the first element will be returned.

5.2.3.6 operator()() [2/2]

```
const float & FAMath::Matrix3x3::operator() (
             unsigned int row,
             unsigned int col ) const  [inline]
```

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

The *row* and *col* values should be between [0,2]. If any of them are out of that range, the first element will be returned.

5.2.3.7 operator*=() [1/2]

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

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

5.2.3.8 operator*=() [2/2]

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

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

5.2.3.9 operator+=()

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

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

5.2.3.10 operator-=()

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

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

5.2.3.11 operator=() [1/2]

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

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

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

5.2.3.12 operator=() [2/2]

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

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

5.2.3.13 SetCol()

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

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

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

5.2.3.14 SetRow()

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

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

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

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

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

5.3 FAMath::Matrix4x4 Class Reference

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

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

Public Member Functions

- Matrix4x4 ()

  Creates a new 4x4 identity matrix.
- Matrix4x4 (float a[ ][4])

  Creates a new 4x4 matrix with elements initialized to the given 2D array.
- Matrix4x4 (const Vector4D &r1, const Vector4D &r2, const Vector4D &r3, const Vector4D &r4)

  Creates a new 4x4 matrix with each row being set to the specified rows.
- Matrix4x4 (const Matrix2x2 &m)

  Creates a new 4x4 matrix with the first two values of the first two rows being set to the values of the 2x2 matrix.
- Matrix4x4 (const Matrix3x3 &m)

  Creates a new 4x4 matrix with the first three values of the first three rows being set to the values of the 3x3 matrix.
- Matrix4x4 & operator= (const Matrix2x2 &m)

  Sets the first two values of the first two rows to the values of the 2x2 matrix.
- Matrix4x4 & operator= (const Matrix3x3 &m)

  Sets the first three values of the first three rows to the values of the 3x3 matrix.
- float ∗ Data ()

  Returns a pointer to the first element in the matrix.
- const float ∗ Data () const

  Returns a constant pointer to the first element in the matrix.
- const float & operator() (unsigned int row, unsigned int col) const

  Returns a constant reference to the element at the given (row, col).
- float & operator() (unsigned int row, unsigned int col)

  Returns a reference to the element at the given (row, col).
- Vector4D GetRow (unsigned int row) const

  Returns the specified row.
- Vector4D GetCol (unsigned int col) const

  Returns the specified col.

- void SetRow (unsigned int row, Vector4D v)

  Sets each element in the given row to the components of vector v.
- void SetCol (unsigned int col, Vector4D v)

  Sets each element in the given col to the components of vector v.
- Matrix4x4 & operator+= (const Matrix4x4 &m)

  Adds this 4x4 matrix with given matrix m and stores the result in this 4x4 matrix.
- Matrix4x4 & operator-= (const Matrix4x4 &m)

  Subtracts m from this 4x4 matrix stores the result in this 4x4 matrix.
- Matrix4x4 & operator*= (float k)

  Multiplies this 4x4 matrix with k and stores the result in this 4x4 matrix.
- Matrix4x4 & operator*= (const Matrix4x4 &m)

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

5.3.1 Detailed Description

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

The datatype for the components is float.
The 4x4 matrix is treated as a row-major matrix.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Matrix4x4() [1/5]

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

Creates a new 4x4 identity matrix.

5.3.2.2 Matrix4x4() [2/5]

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

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

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

5.3.2.3 Matrix4x4() [3/5]

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

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

5.3.2.4 Matrix4x4() [4/5]

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

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

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

5.3.2.5 Matrix4x4() [5/5]

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

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

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

5.3.3 Member Function Documentation

5.3.3.1 Data() [1/2]

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

Returns a pointer to the first element in the matrix.

5.3.3.2 Data() [2/2]

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

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

5.3.3.3 GetCol()

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

Returns the specified *col*.

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

5.3.3.4 GetRow()

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

Returns the specified *row*.

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

5.3.3.5 operator()() [1/2]

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

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

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

5.3.3.6 operator()() [2/2]

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

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

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

5.3.3.7 operator*=() [1/2]

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

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

5.3.3.8 operator*=() [2/2]

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

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

5.3.3.9 operator+=()

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

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

5.3.3.10 operator-=()

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

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

5.3.3.11 operator=() [1/2]

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

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

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

5.3.3.12 operator=() [2/2]

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

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

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

5.3.3.13 SetCol()

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

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

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

5.3.3.14 SetRow()

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

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

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

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

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

5.4 FAMath::Quaternion Class Reference

A quaternion class used for quaternions and their manipulations.

`#include "FAMathEngine.h"`

Public Member Functions

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

5.4.1 Detailed Description

A quaternion class used for quaternions and their manipulations.

The datatype for the components is float.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 Quaternion() [1/3]

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

Constructs a quaternion with the specified values.

If no values are specified the identity quaternion is constructed.

5.4.2.2 Quaternion() [2/3]

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

Constructs a quaternion with the specified values.

5.4.2.3 Quaternion() [3/3]

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

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

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

5.4.3 Member Function Documentation

5.4.3.1 GetScalar()

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

Returns the scalar component of the quaternion.

5.4.3.2 GetVector()

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

Returns the vector component of the quaternion.

5.4.3.3 GetX()

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

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

5.4.3.4 GetY()

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

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

5.4.3.5 GetZ()

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

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

5.4.3.6 operator*=() [1/2]

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

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

5.4.3.7 operator*=() [2/2]

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

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

5.4.3.8 operator+=()

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

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

5.4.3.9 operator-=()

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

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

5.4.3.10 SetScalar()

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

Sets the scalar component to the specified value.

5.4.3.11 SetVector()

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

Sets the vector to the specified vector.

5.4.3.12 SetX()

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

Sets the x component to the specified value.

5.4.3.13 SetY()

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

Sets the y component to the specified value.

5.4.3.14 SetZ()

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

Sets the z component to the specified value.

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

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

5.5 FAMath::Vector2D Class Reference

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

`#include "FAMathEngine.h"`

Public Member Functions

- Vector2D (float x=0.0f, float y=0.0f)

  Creates a new 2D vector/point with the components initialized to the arguments.
- Vector2D (const Vector3D &v)

  Creates a new 2D vector/point with the components initialized to the x and y values of the 3D vector.
- Vector2D (const Vector4D &v)

  Creates a new 2D vector/point with the components initialized to the x and y values of the 4D vector.
- float GetX () const

  Returns the x component.
- float GetY () const

  Returns the y component.
- void SetX (float x)

  Sets the x component of the vector to the specified value.
- void SetY (float y)

  Sets the y component to the specified value.
- Vector2D & operator= (const Vector3D &v)

  Sets the x and y components of this 2D vector to the x and y values of the 3D vector.
- Vector2D & operator= (const Vector4D &v)

  Sets the x and y components of this 2D vector to the x and y values of the 4D vector.
- Vector2D & operator+= (const Vector2D &b)

  Adds this vector to vector b and stores the result in this vector.
- Vector2D & operator-= (const Vector2D &b)

  Subtracts the vector b from this vector and stores the result in this vector.
- Vector2D & operator*= (float k)

  Multiplies this vector by k and stores the result in this vector.
- Vector2D & operator/= (float k)

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

5.5.1 Detailed Description

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

The datatype for the components is float.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 Vector2D() [1/3]

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

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

5.5.2.2 Vector2D() [2/3]

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

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

5.5.2.3 Vector2D() [3/3]

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

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

5.5.3 Member Function Documentation

5.5.3.1 GetX()

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

Returns the x component.

5.5.3.2 GetY()

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

Returns the y component.

5.5.3.3 operator*=()

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

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

5.5.3.4 operator+=()

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

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

5.5.3.5 operator-=()

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

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

5.5.3.6 operator/=()

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

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

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

5.5.3.7 operator=() [1/2]

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

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

5.5.3.8 operator=() [2/2]

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

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

5.5.3.9 SetX()

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

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

5.5.3.10 SetY()

```
void FAMath::Vector2D::SetY (
             float y )  [inline]
```

Sets the y component to the specified value.

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

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

5.6 FAMath::Vector3D Class Reference

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

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

Public Member Functions

- Vector3D (float x=0.0f, float y=0.0f, float z=0.0f)

  Creates a new 3D vector/point with the components initialized to the arguments.
- Vector3D (const Vector2D &v, float z=0.0f)

  Creates a new 3D vector/point with the components initialized to the x and y values of the 2D vector and the specified z value;.
- Vector3D (const Vector4D &v)

  Creates a new 3D vector/point with the components initialized to the x, y and z values of the 4D vector.
- float GetX () const

  Returns the x component.
- float GetY () const

  Returns y component.
- float GetZ () const

  Returns the z component.
- void SetX (float x)

  Sets the x component to the specified value.
- void SetY (float y)

  Sets the y component to the specified value.
- void SetZ (float z)

  Sets the z component to the specified value.
- Vector3D & operator= (const Vector2D &v)

  Sets the x and y components of this 3D vector to the x and y values of the 2D vector and sets the z component to 0.0f.
- Vector3D & operator= (const Vector4D &v)

  Sets the x, y and z components of this 3D vector to the x, y and z values of the 4D vector.
- Vector3D & operator+= (const Vector3D &b)

  Adds this vector to vector b and stores the result in this vector.
- Vector3D & operator-= (const Vector3D &b)

  Subtracts b from this vector and stores the result in this vector.
- Vector3D & operator*= (float k)

  Multiplies this vector by k and stores the result in this vector.
- Vector3D & operator/= (float k)

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

5.6.1 Detailed Description

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

The datatype for the components is float.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 Vector3D() [1/3]

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

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

5.6.2.2 Vector3D() [2/3]

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

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

5.6.2.3 Vector3D() [3/3]

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

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

5.6.3 Member Function Documentation

5.6.3.1 GetX()

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

Returns the x component.

5.6.3.2 GetY()

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

Returns y component.

5.6.3.3 GetZ()

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

Returns the z component.

5.6.3.4 operator*=()

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

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

5.6.3.5 operator+=()

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

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

5.6.3.6 operator-=()

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

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

5.6.3.7 operator/=()

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

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

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

5.6.3.8 operator=() [1/2]

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

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

5.6.3.9 operator=() [2/2]

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

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

5.6.3.10 SetX()

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

Sets the x component to the specified value.

5.6.3.11 SetY()

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

Sets the y component to the specified value.

5.6.3.12 SetZ()

```
void FAMath::Vector3D::SetZ (
             float z )  [inline]
```

Sets the z component to the specified value.

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

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

5.7 FAMath::Vector4D Class Reference

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

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

Public Member Functions

- Vector4D (float x=0.0f, float y=0.0f, float z=0.0f, float w=0.0f)

    Creates a new 4D vector/point with the components initialized to the arguments.
- Vector4D (const Vector2D &v, float z=0.0f, float w=0.0f)

    Creates a new 4D vector/point with the components initialized to the x and y values of the 2D vector and the specified z and w values.
- Vector4D (const Vector3D &v, float w=0.0f)

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

    Returns the x component.
- float GetY () const

    Returns the y component.
- float GetZ () const

    Returns the z component.
- float GetW () const

    Returns the w component.
- void SetX (float x)

    Sets the x component to the specified value.
- void SetY (float y)

    Sets the y component to the specified value.
- void SetZ (float z)

    Sets the z component to the specified value.
- void SetW (float w)

    Sets the w component to the specified value.
- Vector4D & operator= (const Vector2D &v)

    Sets the x and y components of this 4D vector to the x and y values of the 2D vector and sets the z and w component to 0.0f.
- Vector4D & operator= (const Vector3D &v)

    Sets the x, y and z components of this 4D vector to the x, y and z values of the 3D vector and sets the w component to 0.0f.
- Vector4D & operator+= (const Vector4D &b)

    Adds this vector to vector b and stores the result in this vector.
- Vector4D & operator-= (const Vector4D &b)

    Subtracts the vector b from this vector and stores the result in this vector.
- Vector4D & operator*= (float k)

    Multiplies this vector by k and stores the result in this vector.
- Vector4D & operator/= (float k)

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

5.7.1 Detailed Description

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

The datatype for the components is float

5.7.2 Constructor & Destructor Documentation

5.7.2.1 Vector4D() [1/3]

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

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

5.7.2.2 Vector4D() [2/3]

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

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

5.7.2.3 Vector4D() [3/3]

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

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

5.7.3 Member Function Documentation

5.7.3.1 GetW()

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

Returns the w component.

5.7.3.2 GetX()

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

Returns the x component.

5.7.3.3 GetY()

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

Returns the y component.

5.7.3.4 GetZ()

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

Returns the z component.

5.7.3.5 operator*=()

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

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

5.7.3.6 operator+=()

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

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

5.7.3.7 operator-=()

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

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

5.7.3.8 operator/=()

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

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

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

5.7.3.9 operator=() [1/2]

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

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

5.7.3.10 operator=() [2/2]

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

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

5.7.3.11 SetW()

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

Sets the w component to the specified value.

5.7.3.12 SetX()

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

Sets the x component to the specified value.

5.7.3.13 SetY()

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

Sets the y component to the specified value.

5.7.3.14 SetZ()

```
void FAMath::Vector4D::SetZ (
            float z )  [inline]
```

Sets the z component to the specified value.

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

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

Chapter 6

File Documentation

6.1 FAMathEngine.h

```
#pragma once

#include <cmath>

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


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

namespace FAMath
{
    class Vector2D;
    class Vector3D;
    class Vector4D;
    class Matrix2x2;
    class Matrix3x3;
    class Matrix4x4;

    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)));
    }


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

    class Vector2D
    {
    public:

        Vector2D(float x = 0.0f, float y = 0.0f);

        Vector2D(const Vector3D& v);
```

```cpp

        Vector2D(const Vector4D& v);

        float GetX() const;

        float GetY() const;

        void SetX(float x);

        void SetY(float y);

        Vector2D& operator=(const Vector3D& v);

        Vector2D& operator=(const Vector4D& v);

        Vector2D& operator+=(const Vector2D& b);

        Vector2D& operator-=(const Vector2D& b);

        Vector2D& operator*=(float k);

        Vector2D& operator/=(float k);

    private:
        float mX;
        float mY;
    };


    //-------------------------------------------------------------------------------------
    //Vector2D Constructor

    inline Vector2D::Vector2D(float x, float y) :  mX{ x }, mY{ y }
    {}
    //-------------------------------------------------------------------------------------

    //-------------------------------------------------------------------------------------
    //Vector2D Getters and Setters

    inline float Vector2D::GetX()const
{
        return mX;
    }

    inline float Vector2D::GetY()const
{
        return mY;
    }

    inline void Vector2D::SetX(float x)
    {
        mX = x;
    }

    inline void Vector2D::SetY(float y)
    {
        mY = y;
    }

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


    //-------------------------------------------------------------------------------------
    //Vector2D Memeber functions

    inline Vector2D& Vector2D::operator+=(const Vector2D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;

        return *this;
    }

    inline Vector2D& Vector2D::operator-=(const Vector2D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;

        return *this;
    }

    inline Vector2D& Vector2D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;

        return *this;
```

```
}

inline Vector2D& Vector2D::operator/=(float k)
{
    if (CompareFloats(k, 0.0f, EPSILON))
    {
        return *this;
    }

    this->mX /= k;
    this->mY /= k;

    return *this;
}

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

//-----------------------------------------------------------------------------------------
//Vector2D Non-member functions

inline bool ZeroVector(const Vector2D& a)
{
    if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON))
    {
        return true;
    }

    return false;
}

inline Vector2D operator+(const Vector2D& a, const Vector2D& b)
{
    return Vector2D(a.GetX() + b.GetX(), a.GetY() + b.GetY());
}

inline Vector2D operator-(const Vector2D& v)
{
    return Vector2D(-v.GetX(), -v.GetY());
}

inline Vector2D operator-(const Vector2D& a, const Vector2D& b)
{
    return Vector2D(a.GetX() - b.GetX(), a.GetY() - b.GetY());
}

inline Vector2D operator*(const Vector2D& a, float k)
{
    return Vector2D(a.GetX() * k, a.GetY() * k);
}

inline Vector2D operator*(float k, const Vector2D& a)
{
    return Vector2D(k * a.GetX(), k * a.GetY());
}

inline Vector2D operator/(const Vector2D& a, const float& k)
{
    if (CompareFloats(k, 0.0f, EPSILON))
    {
        return Vector2D();
    }

    return Vector2D(a.GetX() / k, a.GetY() / k);
}

inline bool operator==(const Vector2D& a, const Vector2D& b)
{
    return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f);
}

inline bool operator!=(const Vector2D& a, const Vector2D& b)
{
    return !operator==(a, b);
}

inline float DotProduct(const Vector2D& a, const Vector2D& b)
{
    return a.GetX() * b.GetX() + a.GetY() * b.GetY();
}

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

inline Vector2D Norm(const Vector2D& v)
{
```

```
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))

        //v is the zero vector
        if (ZeroVector(v))
        {
            return v;
        }

        float mag{ Length(v) };

        return Vector2D(v.GetX() / mag, v.GetY() / mag);
    }

    inline Vector2D PolarToCartesian(const Vector2D& v)
    {
        //v = (r, theta)
        //x = rcos((theta)
        //y = rsin(theta)
        float angle{ v.GetY() * PI / 180.0f };

        return Vector2D(v.GetX() * cos(angle), v.GetX() * sin(angle));
    }

    inline Vector2D CartesianToPolar(const Vector2D& v)
    {
        //v = (x, y)
        //r = sqrt(vx^2 + vy^2)
        //theta = arctan(y / x)

        if (CompareFloats(v.GetX(), 0.0f, EPSILON))
        {
            return v;
        }

        float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
        return Vector2D(Length(v), theta);
    }

    inline Vector2D Projection(const Vector2D& a, const Vector2D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting

        Vector2D normB(Norm(b));
        return Vector2D(DotProduct(a, normB) * normB);
    }

    inline Vector2D Lerp(const Vector2D& start, const Vector2D& end, float t)
    {
        if (t < 0.0f)
            t = 0.0f;
        else if (t > 1.0f)
            t = 1.0f;

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

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



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




  //------------------------------------------------------------------------------------------------------------------
    class Vector3D
    {
    public:

        Vector3D(float x = 0.0f, float y = 0.0f, float z = 0.0f);

        Vector3D(const Vector2D& v, float z = 0.0f);

        Vector3D(const Vector4D& v);

        float GetX() const;

        float GetY() const;

        float GetZ() const;
```

```
        void SetX(float x);

        void SetY(float y);

        void SetZ(float z);

        Vector3D& operator=(const Vector2D& v);

        Vector3D& operator=(const Vector4D& v);

        Vector3D& operator+=(const Vector3D& b);

        Vector3D& operator-=(const Vector3D& b);

        Vector3D& operator*=(float k);

        Vector3D& operator/=(float k);

    private:
        float mX;
        float mY;
        float mZ;
    };

    //-------------------------------------------------------------------------------------
    //Vector3D Constructors

    inline Vector3D::Vector3D(float x, float y, float z) :  mX{ x }, mY{ y }, mZ{ z }
    {}

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

    //-------------------------------------------------------------------------------------
    //Vector3D Getters and Setters

    inline float Vector3D::GetX()const
{
        return mX;
    }

    inline float Vector3D::GetY()const
{
        return mY;
    }

    inline float Vector3D::GetZ()const
{
        return mZ;
    }

    inline void Vector3D::SetX(float x)
    {
        mX = x;
    }

    inline void Vector3D::SetY(float y)
    {
        mY = y;
    }

    inline void Vector3D::SetZ(float z)
    {
        mZ = z;
    }
    //-------------------------------------------------------------------------------------


    //-------------------------------------------------------------------------------------
    //Vector3D Memeber functions

    inline Vector3D& Vector3D::operator+=(const Vector3D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;
        this->mZ += b.mZ;

        return *this;
    }

    inline Vector3D& Vector3D::operator-=(const Vector3D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;
        this->mZ -= b.mZ;

        return *this;
```

```
    }

    inline Vector3D& Vector3D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;
        this->mZ *= k;

        return *this;
    }

    inline Vector3D& Vector3D::operator/=(float k)
    {
        if (CompareFloats(k, 0.0f, EPSILON))
        {
            return *this;
        }

        this->mX /= k;
        this->mY /= k;
        this->mZ /= k;

        return *this;
    }

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

    //-----------------------------------------------------------------------------------
    //Vector3D Non-member functions

    inline bool ZeroVector(const Vector3D& a)
    {
        if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
            CompareFloats(a.GetZ(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector3D operator+(const Vector3D& a, const Vector3D& b)
    {
        return Vector3D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ());
    }

    inline Vector3D operator-(const Vector3D& v)
    {
        return Vector3D(-v.GetX(), -v.GetY(), -v.GetZ());
    }

    inline Vector3D operator-(const Vector3D& a, const Vector3D& b)
    {
        return Vector3D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ());
    }

    inline Vector3D operator*(const Vector3D& a, float k)
    {
        return Vector3D(a.GetX() * k, a.GetY() * k, a.GetZ() * k);
    }

    inline Vector3D operator*(float k, const Vector3D& a)
    {
        return Vector3D(k * a.GetX(), k * a.GetY(), k * a.GetZ());
    }

    inline Vector3D operator/(const Vector3D& a, float k)
    {
        if (CompareFloats(k, 0.0f, EPSILON))
        {
            return Vector3D();
        }

        return Vector3D(a.GetX() / k, a.GetY() / k, a.GetZ() / k);
    }

    inline bool operator==(const Vector3D& a, const Vector3D& b)
    {
        return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f) &&
            CompareFloats(a.GetZ(), b.GetZ(), 1e-6f);
    }

    inline bool operator!=(const Vector3D& a, const Vector3D& b)
    {
        return !operator==(a, b);
    }

```

```cpp
    inline float DotProduct(const Vector3D& a, const Vector3D& b)
    {
        //a dot b = axbx + ayby + azbz
        return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ();
    }

    inline Vector3D CrossProduct(const Vector3D& a, const Vector3D& b)
    {
        //a x b = (aybz - azby, azbx - axbz, axby - aybx)

        return Vector3D(a.GetY() * b.GetZ() - a.GetZ() * b.GetY(),
            a.GetZ() * b.GetX() - a.GetX() * b.GetZ(),
            a.GetX() * b.GetY() - a.GetY() * b.GetX());
    }

    inline float Length(const Vector3D& v)
    {
        //length(v) = sqrt(vx^2 + vy^2 + vz^2)

        return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ());
    }

    inline Vector3D Norm(const Vector3D& v)
    {
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
        //v is the zero vector
        if (ZeroVector(v))
        {
            return v;
        }

        float mag{ Length(v) };

        return Vector3D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag);
    }

    inline Vector3D CylindricalToCartesian(const Vector3D& v)
    {
        //v = (r, theta, z)
        //x = rcos(theta)
        //y = rsin(theta)
        //z = z
        float angle{ v.GetY() * PI / 180.0f };

        return Vector3D(v.GetX() * cos(angle), v.GetX() * sin(angle), v.GetZ());
    }

    inline Vector3D CartesianToCylindrical(const Vector3D& v)
    {
        //v = (x, y, z)
        //r = sqrt(vx^2 + vy^2 + vz^2)
        //theta = arctan(y / x)
        //z = z
        if (CompareFloats(v.GetX(), 0.0f, EPSILON))
        {
            return v;
        }

        float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };
        return Vector3D(Length(v), theta, v.GetZ());
    }

    inline Vector3D SphericalToCartesian(const Vector3D& v)
    {
        // v = (pho, phi, theta)
        //x = pho * sin(phi) * cos(theta)
        //y = pho * sin(phi) * sin(theta)
        //z = pho * cos(theta);

        float phi{ v.GetY() * PI / 180.0f };
        float theta{ v.GetZ() * PI / 180.0f };

        return Vector3D(v.GetX() * sin(phi) * cos(theta), v.GetX() * sin(phi) * sin(theta), v.GetX() * cos(theta));
    }

    inline Vector3D CartesianToSpherical(const Vector3D& v)
    {
        //v = (x, y ,z)
        //pho = sqrt(vx^2 + vy^2 + vz^2)
        //phi = arcos(z / pho)
        //theta = arctan(y / x)

        if (CompareFloats(v.GetX(), 0.0f, EPSILON) || ZeroVector(v))
        {
            return v;
        }
```

```
        float pho{ Length(v) };
        float phi{ acos(v.GetZ() / pho) * 180.0f / PI };
        float theta{ atan2(v.GetY(), v.GetX()) * 180.0f / PI };

        return Vector3D(pho, phi, theta);
    }

    inline Vector3D Projection(const Vector3D& a, const Vector3D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting

        Vector3D normB(Norm(b));
        return Vector3D(DotProduct(a, normB) * normB);
    }

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

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

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


#if defined(_DEBUG)
    inline void print(const Vector3D& v)
    {
        std::cout « "(" « v.GetX() « ", " « v.GetY() « ", " « v.GetZ() « ")";
    }
#endif
    //--------------------------------------------------------------------------------------


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




   //------------------------------------------------------------------------------------------------------------
    class Vector4D
    {
    public:
        Vector4D(float x = 0.0f, float y = 0.0f, float z = 0.0f, float w = 0.0f);

        Vector4D(const Vector2D& v, float z = 0.0f, float w = 0.0f);

        Vector4D(const Vector3D& v, float w = 0.0f);

        float GetX() const;

        float GetY() const;

        float GetZ() const;

        float GetW() const;

        void SetX(float x);

        void SetY(float y);

        void SetZ(float z);

        void SetW(float w);

        Vector4D& operator=(const Vector2D& v);

        Vector4D& operator=(const Vector3D& v);

        Vector4D& operator+=(const Vector4D& b);

        Vector4D& operator-=(const Vector4D& b);

        Vector4D& operator*=(float k);

        Vector4D& operator/=(float k);

```

```
    private:
        float mX;
        float mY;
        float mZ;
        float mW;
    };

    //--------------------------------------------------------------------------------------
    //Vector4D Constructors

    inline Vector4D::Vector4D(float x, float y, float z, float w) :  mX{ x }, mY{ y }, mZ{ z }, mW{ w }
    {}

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

    //--------------------------------------------------------------------------------------
    //Vector4D Getters and Setters

    inline float Vector4D::GetX()const
{
        return mX;
    }

    inline float Vector4D::GetY()const
{
        return mY;
    }

    inline float Vector4D::GetZ()const
{
        return mZ;
    }

    inline float Vector4D::GetW()const
{
        return mW;
    }

    inline void Vector4D::SetX(float x)
    {
        mX = x;
    }

    inline void Vector4D::SetY(float y)
    {
        mY = y;
    }

    inline void Vector4D::SetZ(float z)
    {
        mZ = z;
    }

    inline void Vector4D::SetW(float w)
    {
        mW = w;
    }
    //--------------------------------------------------------------------------------------


    //--------------------------------------------------------------------------------------
    //Vector4D Memeber functions

    inline Vector4D& Vector4D::operator+=(const Vector4D& b)
    {
        this->mX += b.mX;
        this->mY += b.mY;
        this->mZ += b.mZ;
        this->mW += b.mW;

        return *this;
    }

    inline Vector4D& Vector4D::operator-=(const Vector4D& b)
    {
        this->mX -= b.mX;
        this->mY -= b.mY;
        this->mZ -= b.mZ;
        this->mW -= b.mW;

        return *this;
    }

    inline Vector4D& Vector4D::operator*=(float k)
    {
        this->mX *= k;
        this->mY *= k;
```

```
        this->mZ *= k;
        this->mW *= k;

        return *this;
    }

    inline Vector4D& Vector4D::operator/=(float k)
    {
        if (CompareFloats(k, 0.0f, EPSILON))
        {
            return *this;
        }

        this->mX /= k;
        this->mY /= k;
        this->mZ /= k;
        this->mW /= k;

        return *this;
    }

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

    //-----------------------------------------------------------------------------------
    //Vector4D Non-member functions

    inline bool ZeroVector(const Vector4D& a)
    {
        if (CompareFloats(a.GetX(), 0.0f, EPSILON) && CompareFloats(a.GetY(), 0.0f, EPSILON) &&
            CompareFloats(a.GetZ(), 0.0f, EPSILON) && CompareFloats(a.GetW(), 0.0f, EPSILON))
        {
            return true;
        }

        return false;
    }

    inline Vector4D operator+(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D(a.GetX() + b.GetX(), a.GetY() + b.GetY(), a.GetZ() + b.GetZ(), a.GetW() +
 b.GetW());
    }

    inline Vector4D operator-(const Vector4D& v)
    {
        return Vector4D(-v.GetX(), -v.GetY(), -v.GetZ(), -v.GetW());
    }

    inline Vector4D operator-(const Vector4D& a, const Vector4D& b)
    {
        return Vector4D(a.GetX() - b.GetX(), a.GetY() - b.GetY(), a.GetZ() - b.GetZ(), a.GetW() -
 b.GetW());
    }

    inline Vector4D operator*(const Vector4D& a, float k)
    {
        return Vector4D(a.GetX() * k, a.GetY() * k, a.GetZ() * k, a.GetW() * k);
    }

    inline Vector4D operator*(float k, const Vector4D& a)
    {
        return Vector4D(k * a.GetX(), k * a.GetY(), k * a.GetZ(), k * a.GetW());
    }

    inline Vector4D operator/(const Vector4D& a, float k)
    {
        if (CompareFloats(k, 0.0f, EPSILON))
        {
            return Vector4D();
        }

        return Vector4D(a.GetX() / k, a.GetY() / k, a.GetZ() / k, a.GetW() / k);
    }

    inline bool operator==(const Vector4D& a, const Vector4D& b)
    {
        return CompareFloats(a.GetX(), b.GetX(), 1e-6f) && CompareFloats(a.GetY(), b.GetY(), 1e-6f) &&
            CompareFloats(a.GetZ(), b.GetZ(), 1e-6f) && CompareFloats(a.GetW(), b.GetW(), 1e-6f);
    }

    inline bool operator!=(const Vector4D& a, const Vector4D& b)
    {
        return !operator==(a, b);
    }

    inline float DotProduct(const Vector4D& a, const Vector4D& b)
    {
```

```
        //a dot b = axbx + ayby + azbz + awbw
        return a.GetX() * b.GetX() + a.GetY() * b.GetY() + a.GetZ() * b.GetZ() + a.GetW() * b.GetW();
    }

    inline float Length(const Vector4D& v)
    {
        //length(v) = sqrt(vx^2 + vy^2 + vz^2 + vw^2)
        return sqrt(v.GetX() * v.GetX() + v.GetY() * v.GetY() + v.GetZ() * v.GetZ() + v.GetW() *
    v.GetW());
    }

    inline Vector4D Norm(const Vector4D& v)
    {
        //norm(v) = v / length(v) == (vx / length(v), vy / length(v))
        //v is the zero vector
        if (ZeroVector(v))
        {
            return v;
        }

        float mag{ Length(v) };

        return Vector4D(v.GetX() / mag, v.GetY() / mag, v.GetZ() / mag, v.GetW() / mag);
    }

    inline Vector4D Projection(const Vector4D& a, const Vector4D& b)
    {
        //Projb(a) = (a dot b)b
        //normalize b before projecting
        Vector4D normB(Norm(b));
        return Vector4D(DotProduct(a, normB) * normB);
    }

    inline void Orthonormalize(Vector4D& x, Vector4D& y, Vector4D& z)
    {
        FAMath::Vector3D tempX(x.GetX(), x.GetY(), x.GetZ());
        FAMath::Vector3D tempY(y.GetX(), y.GetY(), y.GetZ());
        FAMath::Vector3D tempZ(z.GetX(), z.GetY(), z.GetZ());

        tempX = Norm(tempX);
        tempY = Norm(CrossProduct(tempZ, tempX));
        tempZ = Norm(CrossProduct(tempX, tempY));

        x = FAMath::Vector4D(tempX.GetX(), tempX.GetY(), tempX.GetZ(), 0.0f);
        y = FAMath::Vector4D(tempY.GetX(), tempY.GetY(), tempY.GetZ(), 0.0f);
        z = FAMath::Vector4D(tempZ.GetX(), tempZ.GetY(), tempZ.GetZ(), 0.0f);
    }

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

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


#if defined(_DEBUG)
    inline void print(const Vector4D& v)
    {
        std::cout « "(" « v.GetX() « ", " « v.GetY() « ", " « v.GetZ() « ", " « v.GetW() « ")";
    }
#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.GetX();
            mMat[0][1] = v.GetY();
        }
        else
        {
            mMat[row][0] = v.GetX();
            mMat[row][1] = v.GetY();
        }
    }

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

    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.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY());

        res.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY());

        return res;
    }

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

        res.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0));

        res.SetY(v.GetX() * m(0, 1) + v.GetY() * 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 IsIdentity(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(const Matrix2x2& cm, float x, float y)
{
    //x 0
    //0 y

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

    return cm * scale;
}

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

    Matrix2x2 scale;
    scale(0, 0) = scaleVector.GetX();
    scale(1, 1) = scaleVector.GetY();

    return cm * scale;
}

inline Matrix2x2 Rotate(const Matrix2x2& cm, 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 cm * 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.GetX();
            mMat[0][1] = v.GetY();
            mMat[0][2] = v.GetZ();
        }
        else
        {
            mMat[row][0] = v.GetX();
            mMat[row][1] = v.GetY();
            mMat[row][2] = v.GetZ();
        }
    }

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

    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.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ());

        result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ());

        result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ());

        return result;
    }

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

        result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0));

        result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1));

        result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * 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 IsIdentity(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(const Matrix3x3& cm, 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 cm * scale;
    }

    inline Matrix3x3 Scale(const Matrix3x3& cm, const Vector3D& scaleVector)
    {
```

```
        //x 0 0
        //0 y 0
        //0 0 z

        Matrix3x3 scale;
        scale(0, 0) = scaleVector.GetX();
        scale(1, 1) = scaleVector.GetY();
        scale(2, 2) = scaleVector.GetZ();

        return cm * scale;
    }

    inline Matrix3x3 Rotate(const Matrix3x3& cm, 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 = Norm(axis);
        x = axis.GetX();
        y = axis.GetY();
        z = axis.GetZ();

        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 cm * result;
    }

    inline Matrix3x3 Rotate(const Matrix3x3& cm, 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(Norm(axis));
        float x = nAxis.GetX();
        float y = nAxis.GetY();
        float z = nAxis.GetZ();

        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 cm * 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.GetX();
        mMat[0][1] = v.GetY();
        mMat[0][2] = v.GetZ();
        mMat[0][3] = v.GetW();
    }
    else
    {
        mMat[row][0] = v.GetX();
        mMat[row][1] = v.GetY();
        mMat[row][2] = v.GetZ();
        mMat[row][3] = v.GetW();
    }
}

inline void Matrix4x4::SetCol(unsigned int col, Vector4D v)
{
    if (col > 3)
    {
        mMat[0][0] = v.GetX();
        mMat[1][0] = v.GetY();
        mMat[2][0] = v.GetZ();
        mMat[3][0] = v.GetW();
    }
    else
    {
        mMat[0][col] = v.GetX();
        mMat[1][col] = v.GetY();
        mMat[2][col] = v.GetZ();
        mMat[3][col] = v.GetW();
    }
}

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.SetX(m(0, 0) * v.GetX() + m(0, 1) * v.GetY() + m(0, 2) * v.GetZ() + m(0, 3) * v.GetW());

        result.SetY(m(1, 0) * v.GetX() + m(1, 1) * v.GetY() + m(1, 2) * v.GetZ() + m(1, 3) * v.GetW());

        result.SetZ(m(2, 0) * v.GetX() + m(2, 1) * v.GetY() + m(2, 2) * v.GetZ() + m(2, 3) * v.GetW());

        result.SetW(m(3, 0) * v.GetX() + m(3, 1) * v.GetY() + m(3, 2) * v.GetZ() + m(3, 3) * v.GetW());

        return result;
    }

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

        result.SetX(v.GetX() * m(0, 0) + v.GetY() * m(1, 0) + v.GetZ() * m(2, 0) + v.GetW() * m(3, 0));

        result.SetY(v.GetX() * m(0, 1) + v.GetY() * m(1, 1) + v.GetZ() * m(2, 1) + v.GetW() * m(3, 1));

        result.SetZ(v.GetX() * m(0, 2) + v.GetY() * m(1, 2) + v.GetZ() * m(2, 2) + v.GetW() * m(3, 2));

        result.SetW(v.GetX() * m(0, 3) + v.GetY() * m(1, 3) + v.GetZ() * m(2, 3) + v.GetW() * 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 IsIdentity(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(const Matrix4x4& cm, 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 cm * translate;
```

```
    }

    inline Matrix4x4 Translate(const Matrix4x4& cm, 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.GetX();
        translate(3, 1) = translateVector.GetY();
        translate(3, 2) = translateVector.GetZ();

        return cm * translate;
    }

    inline Matrix4x4 Scale(const Matrix4x4& cm, 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 cm * scale;
    }

    inline Matrix4x4 Scale(const Matrix4x4& cm, 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.GetX();
        scale(1, 1) = scaleVector.GetY();
        scale(2, 2) = scaleVector.GetZ();

        return cm * scale;
    }

    inline Matrix4x4 Rotate(const Matrix4x4& cm, 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 = Norm(axis);

        x = axis.GetX();
        y = axis.GetY();
        z = axis.GetZ();

        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 cm * result;
    }
```

```
    inline Matrix4x4 Rotate(const Matrix4x4& cm, 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(Norm(axis));

        float x = nAxis.GetX();
        float y = nAxis.GetY();
        float z = nAxis.GetZ();

        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 cm * 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


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



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

    class Quaternion
```

```
    {
    public:
        Quaternion(float scalar = 1.0f, float x = 0.0f, float y = 0.0f, float z = 0.0f);

        Quaternion(float scalar, const Vector3D& v);

        Quaternion(const Vector4D& v);

        float GetScalar() const;

        float GetX() const;

        float GetY() const;

        float GetZ() const;

        Vector3D GetVector() const;

        void SetScalar(float scalar);

        void SetX(float x);

        void SetY(float y);

        void SetZ(float z);

        void SetVector(const Vector3D& v);

        Quaternion& operator+=(const Quaternion& q);

        Quaternion& operator-=(const Quaternion& q);

        Quaternion& operator*=(float k);

        Quaternion& operator*=(const Quaternion& q);

    private:
        float mScalar;
        float mX;
        float mY;
        float mZ;
    };

    //-------------------------------------------------------------------------------------
    inline Quaternion::Quaternion(float scalar, float x, float y, float z) :
        mScalar{ scalar }, mX{ x }, mY{ y }, mZ{ z }
    {}

    inline Quaternion::Quaternion(float scalar, const Vector3D& v) :
        mScalar{ scalar }, mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
    {}

    inline Quaternion::Quaternion(const Vector4D& v) :
        mScalar{ v.GetX() }, mX{ v.GetY() }, mY{ v.GetZ() }, mZ{ v.GetW() }
    {}

    inline float Quaternion::GetScalar()const
{
        return mScalar;
    }

    inline float Quaternion::GetX()const
{
        return mX;
    }

    inline float Quaternion::GetY()const
{
        return mY;
    }

    inline float Quaternion::GetZ()const
{
        return mZ;
    }

    inline Vector3D Quaternion::GetVector()const
{
        return Vector3D(mX, mY, mZ);
    }

    inline void Quaternion::SetScalar(float scalar)
    {
        mScalar = scalar;
    }

    inline void Quaternion::SetX(float x)
```

```
    {
        mX = x;
    }

    inline void Quaternion::SetY(float y)
    {
        mY = y;
    }

    inline void  Quaternion::SetZ(float z)
    {
        mZ = z;
    }

    inline void Quaternion::SetVector(const Vector3D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = v.GetZ();
    }

    inline Quaternion& Quaternion::operator+=(const Quaternion& q)
    {
        this->mScalar += q.mScalar;
        this->mX += q.mX;
        this->mY += q.mY;
        this->mZ += q.mZ;

        return *this;
    }

    inline Quaternion& Quaternion::operator-=(const Quaternion& q)
    {
        this->mScalar -= q.mScalar;
        this->mX -= q.mX;
        this->mY -= q.mY;
        this->mZ -= q.mZ;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(float k)
    {
        this->mScalar *= k;
        this->mX *= k;
        this->mY *= k;
        this->mZ *= k;

        return *this;
    }

    inline Quaternion& Quaternion::operator*=(const Quaternion& q)
    {
        Vector3D thisVector(this->mX, this->mY, this->mZ);
        Vector3D qVector(q.mX, q.mY, q.mZ);

        float scalar{ this->mScalar * q.mScalar };
        float dotProduct{ DotProduct(thisVector, qVector) };
        float resultScalar{ scalar - dotProduct };

        Vector3D a(this->mScalar * qVector);
        Vector3D b(q.mScalar * thisVector);
        Vector3D crossProduct(CrossProduct(thisVector, qVector));
        Vector3D resultVector(a + b + crossProduct);

        this->mScalar = resultScalar;
        this->mX = resultVector.GetX();
        this->mY = resultVector.GetY();
        this->mZ = resultVector.GetZ();

        return *this;
    }

    inline Quaternion operator+(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion(q1.GetScalar() + q2.GetScalar(), q1.GetX() + q2.GetX(), q1.GetY() +
  q2.GetY(), q1.GetZ() + q2.GetZ());
    }

    inline Quaternion operator-(const Quaternion& q)
    {
        return Quaternion(-q.GetScalar(), -q.GetX(), -q.GetY(), -q.GetZ());
    }

    inline Quaternion operator-(const Quaternion& q1, const Quaternion& q2)
    {
        return Quaternion(q1.GetScalar() - q2.GetScalar(),
```

```
            q1.GetX() - q2.GetX(), q1.GetY() - q2.GetY(), q1.GetZ() - q2.GetZ());
    }

    inline Quaternion operator*(float k, const Quaternion& q)
    {
        return Quaternion(k * q.GetScalar(), k * q.GetX(), k * q.GetY(), k * q.GetZ());
    }

    inline Quaternion operator*(const Quaternion& q, float k)
    {
        return Quaternion(q.GetScalar() * k, q.GetX() * k, q.GetY() * k, q.GetZ() * k);
    }

    inline Quaternion operator*(const Quaternion& q1, const Quaternion& q2)
    {
        //scalar part = q1scalar * q2scalar - q1Vector dot q2Vector
        //vector part = q1Scalar * q2Vector + q2Scalar * q1Vector + q1Vector cross q2Vector

        Vector3D q1Vector(q1.GetX(), q1.GetY(), q1.GetZ());
        Vector3D q2Vector(q2.GetX(), q2.GetY(), q2.GetZ());

        float scalar{ q1.GetScalar() * q2.GetScalar() };
        float dotProduct{ DotProduct(q1Vector, q2Vector) };
        float resultScalar{ scalar - dotProduct };

        Vector3D a(q1.GetScalar() * q2Vector);
        Vector3D b(q2.GetScalar() * q1Vector);
        Vector3D crossProduct(CrossProduct(q1Vector, q2Vector));
        Vector3D resultVector(a + b + crossProduct);

        return Quaternion(resultScalar, resultVector);
    }

    inline bool operator==(const Quaternion& q1, const Quaternion& q2)
    {
        return CompareFloats(q1.GetScalar(), q2.GetScalar(), 1e-6f) && CompareFloats(q1.GetX(), q2.GetX(), 1e-6f) &&
            CompareFloats(q1.GetY(), q2.GetY(), 1e-6f) && CompareFloats(q1.GetZ(), q2.GetZ(), 1e-6f);
    }

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

    inline bool IsZeroQuaternion(const Quaternion& q)
    {
        //zero quaternion = (0, 0, 0, 0)
        return CompareFloats(q.GetScalar(), 0.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
            CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
    }

    inline bool IsIdentity(const Quaternion& q)
    {
        //identity quaternion = (1, 0, 0, 0)
        return CompareFloats(q.GetScalar(), 1.0f, EPSILON) && CompareFloats(q.GetX(), 0.0f, EPSILON) &&
            CompareFloats(q.GetY(), 0.0f, EPSILON) && CompareFloats(q.GetZ(), 0.0f, EPSILON);
    }

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

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

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

        if (IsZeroQuaternion(q))
            return q;

        float magnitdue{ Length(q) };

        return Quaternion(q.GetScalar() / magnitdue, q.GetX() / magnitdue, q.GetY() / magnitdue, q.GetZ() / magnitdue);
    }

    inline Quaternion Inverse(const Quaternion& q)
    {
```

```
        //inverse = conjugate of q / |q|

        if (IsZeroQuaternion(q))
            return q;

        Quaternion conjugateOfQ(Conjugate(q));

        float magnitdue{ Length(q) };

        return Quaternion(conjugateOfQ.GetScalar() / magnitdue, conjugateOfQ.GetX() / magnitdue,
            conjugateOfQ.GetY() / magnitdue, conjugateOfQ.GetZ() / magnitdue);
    }

    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 = Norm(axis);

        return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
    }

    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(Norm(axis));

        return Quaternion(c, s * axisN.GetX(), s * axisN.GetY(), s * axisN.GetZ());
    }

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

        Vector3D axis(angAxis.GetY(), angAxis.GetZ(), angAxis.GetW());
        axis = Norm(axis);

        return Quaternion(c, s * axis.GetX(), s * axis.GetY(), s * axis.GetZ());
    }

    inline Matrix4x4 QuaternionToRotationMatrixCol(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.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
        colMat(0, 1) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
        colMat(0, 2) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();

        colMat(1, 0) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
        colMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
        colMat(1, 2) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();

        colMat(2, 0) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();
        colMat(2, 1) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();
        colMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();
```

```
        return colMat;
    }

    inline Matrix4x4 QuaternionToRotationMatrixRow(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.GetY() * q.GetY() - 2.0f * q.GetZ() * q.GetZ();
        rowMat(0, 1) = 2.0f * q.GetX() * q.GetY() + 2.0f * q.GetScalar() * q.GetZ();
        rowMat(0, 2) = 2.0f * q.GetX() * q.GetZ() - 2.0f * q.GetScalar() * q.GetY();

        rowMat(1, 0) = 2.0f * q.GetX() * q.GetY() - 2.0f * q.GetScalar() * q.GetZ();
        rowMat(1, 1) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetZ() * q.GetZ();
        rowMat(1, 2) = 2.0f * q.GetY() * q.GetZ() + 2.0f * q.GetScalar() * q.GetX();

        rowMat(2, 0) = 2.0f * q.GetX() * q.GetZ() + 2.0f * q.GetScalar() * q.GetY();
        rowMat(2, 1) = 2.0f * q.GetY() * q.GetZ() - 2.0f * q.GetScalar() * q.GetX();
        rowMat(2, 2) = 1.0f - 2.0f * q.GetX() * q.GetX() - 2.0f * q.GetY() * q.GetY();

        return rowMat;
    }

    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, q is a rotation quaternion
        //and q* is its conjugate.

        Quaternion point(0.0f, p);

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

        return result.GetVector();
    }

    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);

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

        return Vector4D(result.GetVector(), p.GetW());
    }

    inline float DotProduct(const Quaternion& q1, const Quaternion& q2)
    {
        return q1.GetScalar() * q2.GetScalar() + q1.GetX() * q2.GetX() + q1.GetY() * q2.GetY() +
q1.GetZ() * q2.GetZ();
    }

    inline Quaternion Lerp(const Quaternion& q0, const Quaternion& q1, float t)
    {
        if (t < 0.0f)
            t = 0.0f;
        else if (t > 1.0f)
            t = 1.0f;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q0, q1);

        Quaternion newQ1;
        //If the dot product is negative, negate q1 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ1 = -q1;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ1 = q1;
        }

        return (1.0f - t) * q0 + t * q1;
```

```
    }

    inline Quaternion NLerp(const Quaternion& q0, const Quaternion& q1, float t)
    {
        if (t < 0.0f)
            t = 0.0f;
        else if (t > 1.0f)
            t = 1.0f;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q0, q1);

        Quaternion newQ1;
        //If the dot product is negative, negate q1 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ1 = -q1;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ1 = q1;
        }

        return Normalize((1.0f - t) * q0 + t * q1);
    }

    inline Quaternion Slerp(const Quaternion& q0, const Quaternion& q1, float t)
    {
        //Formula used is
        //k0 = sin((1 - t)omega) * omega) / sin(omega);
        //k1 = (sin(tomega) * omega) / sin(omega)
        //newQ = k0q0 * k1q1
        //Omega is the angle between the q0 and q1.

        if (t < 0.0f)
            t = 0.0f;
        else if (t > 1.0f)
            t = 1.0f;

        //Compute the cosine of the angle between the quaternions
        float cosOmega = DotProduct(q0, q1);

        Quaternion newQ1;
        //If the dot product is negative, negate q1 to so we take the shorter arc
        if (cosOmega < 0.0f)
        {
            newQ1 = -q1;
            cosOmega = -cosOmega;
        }
        else
        {
            newQ1 = q1;
        }

        float k0{ 0.0f };
        float k1{ 0.0f };

        //Linear interpolate if the quaternions are very close to protect dividing by zero.
        if (cosOmega > 0.9999f)
        {
            k0 = 1.0f - t;
            k1 = 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 };

            k0 = sin((1.0f - t) * omega) * oneOverSinOmega;
            k1 = sin(t * omega) * oneOverSinOmega;
        }

        return k0 * q0 + k1 * newQ1;
    }

#if defined(_DEBUG)
    inline void print(const Quaternion& q)
    {
```

```
        std::cout « "(" « q.GetScalar() « ", " « q.GetX() « ", " « q.GetY() « ", " « q.GetZ();
    }
#endif
    //-----------------------------------------------------------------------------------
    
    inline Vector2D::Vector2D(const Vector3D& v) :  mX{ v.GetX() }, mY{ v.GetY() }
    {}
    
    inline Vector2D::Vector2D(const Vector4D& v) :  mX{ v.GetX() }, mY{ v.GetY() }
    {}
    
    inline Vector2D& Vector2D::operator=(const Vector3D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
    
        return *this;
    }
    
    inline Vector2D& Vector2D::operator=(const Vector4D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
    
        return *this;
    }
    
    inline Vector3D::Vector3D(const Vector2D& v, float z) :  mX{ v.GetX() }, mY{ v.GetY() }, mZ{ z }
    {}
    
    inline Vector3D::Vector3D(const Vector4D& v) :  mX{ v.GetX() }, mY{ v.GetY() }, mZ{ v.GetZ() }
    {}
    
    inline Vector3D& Vector3D::operator=(const Vector2D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = 0.0f;
    
        return *this;
    }
    
    inline Vector3D& Vector3D::operator=(const Vector4D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = v.GetZ();
    
        return *this;
    }
    
    inline Vector4D::Vector4D(const Vector2D& v, float z, float w) :  mX{ v.GetX() }, mY{ v.GetY() },
   mZ{ z }, mW{ w }
    {}
    
    inline Vector4D::Vector4D(const Vector3D& v, float w) :  mX{ v.GetX() }, mY{ v.GetY() }, mZ{
   v.GetZ() }, mW{ w }
    {}
    
    inline Vector4D& Vector4D::operator=(const Vector2D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = 0.0f;
        mW = 0.0f;
    
        return *this;
    }
    
    inline Vector4D& Vector4D::operator=(const Vector3D& v)
    {
        mX = v.GetX();
        mY = v.GetY();
        mZ = v.GetZ();
        mW = 0.0f;
    
        return *this;
    }
    
    inline Matrix2x2::Matrix2x2(const Matrix3x3& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
    
        //2nd row
        mMat[1][0] = m(1, 0);
```

```
        mMat[1][1] = m(1, 1);
    }

    inline Matrix2x2::Matrix2x2(const Matrix4x4& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
    }

    inline Matrix2x2& Matrix2x2::operator=(const Matrix3x3& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);

        return *this;
    }

    inline Matrix2x2& Matrix2x2::operator=(const Matrix4x4& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);

        return *this;
    }

    inline Matrix3x3::Matrix3x3(const Matrix2x2& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        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(const Matrix4x4& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = m(0, 2);

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        mMat[1][2] = m(1, 2);

        //3rd row
        mMat[2][0] = m(2, 0);
        mMat[2][1] = m(2, 1);
        mMat[2][2] = m(2, 2);
    }

    inline Matrix3x3& Matrix3x3::operator=(const Matrix2x2& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        mMat[1][2] = 0.0f;
```

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

        return *this;
    }

    inline Matrix3x3& Matrix3x3::operator=(const Matrix4x4& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = m(0, 2);

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        mMat[1][2] = m(1, 2);

        //3rd row
        mMat[2][0] = m(2, 0);
        mMat[2][1] = m(2, 1);
        mMat[2][2] = m(2, 2);

        return *this;
    }

    inline Matrix4x4::Matrix4x4(const Matrix2x2& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = 0.0f;
        mMat[0][3] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        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(const Matrix3x3& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = m(0, 2);
        mMat[0][3] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        mMat[1][2] = m(1, 2);
        mMat[1][3] = 0.0f;

        //3rd row
        mMat[2][0] = m(2, 0);
        mMat[2][1] = m(2, 1);
        mMat[2][2] = m(2, 2);
        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::operator=(const Matrix2x2& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
```

```
        mMat[0][1] = m(0, 1);
        mMat[0][2] = 0.0f;
        mMat[0][3] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        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;

        return *this;
    }

    inline Matrix4x4& Matrix4x4::operator=(const Matrix3x3& m)
    {
        //1st row
        mMat[0][0] = m(0, 0);
        mMat[0][1] = m(0, 1);
        mMat[0][2] = m(0, 2);
        mMat[0][3] = 0.0f;

        //2nd row
        mMat[1][0] = m(1, 0);
        mMat[1][1] = m(1, 1);
        mMat[1][2] = m(1, 2);
        mMat[1][3] = 0.0f;

        //3rd row
        mMat[2][0] = m(2, 0);
        mMat[2][1] = m(2, 1);
        mMat[2][2] = m(2, 2);
        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;

        return *this;
    }

  //---------------------------------------------------------------------------------------------------------------
}
```

Index