(Math for Games)

Devin Wiggins

(12/10/16)

**I.Requirement documentation**

I.1 **Description of problem**

Name: MyMathLibrary

Problem: Make a math Library that can be used to make a game

Problem description: Game and simulation programming is heavily reliant on mathematical techniques to manipulate and display virtual environments. This subject aims to teach you the fundamental mathematical skills needed in 2-dimnsional and 3-dimensional games, starting out with the basic mathematical formulae and then moving on to more complex techniques involving Vectors and Matrices. Students work through creating a basic math library to fully understanding the formulae needed to manipulate the Vectors and Matrices used within all games. This library is created as a library that can be redistributed and used within other applications. To ensure the mathematical formulae are correct the maths library will be tested with a Unit Testing application to ensure that the output is correct. Students will also explore basic collision detection techniques used in video games to determine when objects come in to contact with each other and various other needs, as well as gain an understanding of binary operations within programming.

I.2 **Input Description**

For the actual game: the Player(ship) can be controlled by using the arrow keys

For the library: N/A

I.3 **Output** D**escriptio**n

For the math library: a read me is displayed to the console and a file is written to.

For the game: displays game

I.4 **User interface**

Console

Text file

**II.** **Design Documentation**

II.2 **system architecture**

* **Class Name** (Vector2)
  + **In Public**
* **Name**: m\_x(float
* **Parameter(s)**: N/A
* **Description**: variable for x coordinate
* **Name**: m\_y(float)
* **Parameter(s)**:N/A
* **Description**: variable for the y coordinate
* **Name**:Vector()
* **Parameter(s)**:none
* **Description**: default constructor
* **Name**: Vector2()
* **Parameter(s)**: float x1, float y1
* **Description**: constructor
* **Name**: Add(Vector2)
* **Parameter(s)**: Vector2 B
* **Description**: add a Vector2 to another vector2

* **Name**: operator +(Vector2)
* **Parameter(s)**: Vector2 B
* **Description**: overloads + to add 2 vector2’s
* **Name**: Subtract(Vector2)
* **Parameter(s)**: Vector2 B
* **Description**: subtracts vector2 instance from another vector
* **Name**: operator -(Vector2)
* **Parameter(s)**: Vector B
* **Description**: overloads - to use subtraction with vector2’s
* **Name:** ScalerMult(Vector2)
* **Parameters(s)**: float k
* **Description**: scales Vector2
* **Name:** operator \*(Vector2)
* **Parameter(s)**: float k
* **Description**: scales Vector2
* **Name:** Magnitude(float)
* **Parameter(s)**: none
* **Description**: returns the magnitude of the Vector2
* **Name**: Normalize(Vector2)
* **Parameter(s)**: none
* **Description**: returns the Vector2 normalized
* **Name**: DotProduct(float)
* **Parameter(s)**: Vector2 B
* **Description**: returns dot product of 2 vector2s
* **Name**: operator ==(bool)
* **Parameter(s)**: Vector2 &result
* **Description**: overloaded == operator to compare vector3s
* **Name**: operator =(float)
* **Parameter(s)**: float& A
* **Description**: all coordinates can be assigned to a single float
* **Name**: operator <<(friend ofstream&)
* **Parameter(s)**:ofstream& output, Vector2 &n
* **Description**:prints Vector to a file
* **Class Vector3**
  + **In Public**
* **Name**: m\_x(float)
* **Parameter(s)**: N/A
* **Description**: variable for x coordinate
* **Name**: m\_y(float)
* **Parameter(s)**:N/A
* **Description**: variable for the y coordinate
* **Name**: m\_z(float)
* **Parameter(s)**: none
* **Description:** variable for the z coordinate
* **Name**:Vector()
* **Parameter(s)**:none
* **Description**: default constructor
* **Name**: Vector3()
* **Parameter(s)**: float x1, float y1, float z1
* **Description**: constructor
* **Name**: Add(Vector3)
* **Parameter(s)**: Vector3 B
* **Description**: add a Vector2 to another vector3

* **Name**: operator +(Vector3)
* **Parameter(s)**: Vector3 B
* **Description**: overloads + to add 2 vector3’s
* **Name**: Subtract(Vector3)
* **Parameter(s)**: Vector3 B
* **Description**: subtracts vector3 instance from another vector
* **Name**: operator -(Vector3)
* **Parameter(s)**: Vector B
* **Description**: overloads - to use subtraction with vector3’s
* **Name:** ScalerMult(Vector3)
* **Parameters(s)**: float k
* **Description**: scales Vector3
* **Name:** operator \*(Vector3)
* **Parameter(s)**: float k
* **Description**: scales Vector3
* **Name:** Magnitude(float)
* **Parameter(s)**: none
* **Description**: returns the magnitude of the Vector3
* **Name**: Normalize(Vector3)
* **Parameter(s)**: none
* **Description**: returns the Vector3 normalized
* **Name**: DotProduct(float)
* **Parameter(s)**: Vector3 B
* **Description**: returns dot product of 2 vector3s
* **Name**: operator ==(bool)
* **Parameter(s)**: Vector3 &result
* **Description**: overloaded == operator to compare vector3s
* **Name**: operator =(float)
* **Parameter(s)**: float& A
* **Description**: all coordinates can be assigned to a single float
* **Name**: operator <<(friend ofstream&)
* **Parameter(s)**:ofstream& output, Vector3 &n
* **Description**:prints Vector to a file
* **Name**: Normalize(Vector3)
* **Parameter(s)**: none
* **Description**: returns the Vector3 normalized
* **Name**: DotProduct(float)
* **Parameter(s)**: Vector3 B
* **Description**: returns dot product of 2 vector3s
* **Name:** CrossProduct(Vector3)
* **Parameter(s)**: Vector3 B
* **Description**: returns the cross product of a vector3 instance and a Vector3
* **Name**: operator ==(bool)
* **Parameter(s)**: Vector3 &result
* **Description**: overloaded == operator to compare vector3s
* **Name**: operator =(float)
* **Parameter(s)**: float& A
* **Description**: all coordinates can be assigned to a single float
* **Name**: operator <<(friend ofstream&)
* **Parameter(s)**:ofstream& output, Vector3 &n
* **Description**:prints Vector to a file
* **Class Vector4**
  + **In Public**
* **Name**: m\_x(float)
* **Parameter(s)**: N/A
* **Description**: variable for x coordinate
* **Name**: m\_y(float)
* **Parameter(s)**:N/A
* **Description**: variable for the y coordinate
* **Name**: m\_z(float)
* **Parameter(s)**: none
* **Description:** variable for the z coordinate
* **Name**: m\_w(float)
* **Parameter(s)**: none
* **Description:** variable for the w coordinate
* **Name**:Vector()
* **Parameter(s)**:none
* **Description**: default constructor
* **Name**: Vector4()
* **Parameter(s)**: float x1, float y1, float z1, flaot w1
* **Description**: constructor
* **Name**: Add(Vector4)
* **Parameter(s)**: Vector4 B
* **Description**: add a Vector4 to another vector4

* **Name**: operator +(Vector4)
* **Parameter(s)**: Vector4 B
* **Description**: overloads + to add 2 vector4’s
* **Name**: Subtract(Vector4)
* **Parameter(s)**: Vector4 B
* **Description**: subtracts vector4 instance from another vector
* **Name**: operator -(Vector4)
* **Parameter(s)**: Vector4 B
* **Description**: overloads - to use subtraction with vector4’s
* **Name:** ScalerMult(Vector4)
* **Parameters(s)**: float k
* **Description**: scales Vector4
* **Name:** operator \*(Vector4)
* **Parameter(s)**: float k
* **Description**: scales Vector4
* **Name:** Magnitude(float)
* **Parameter(s)**: none
* **Description**: returns the magnitude of the Vector4
* **Name**: Normalize(Vector4)
* **Parameter(s)**: none
* **Description**: returns the Vector4 normalized
* **Name**: DotProduct(float)
* **Parameter(s)**: Vector4 B
* **Description**: returns dot product of 2 vector4s
* **Name**: operator ==(bool)
* **Parameter(s)**: Vector4 &result
* **Description**: overloaded == operator to compare vector4s
* **Name**: operator =(float)
* **Parameter(s)**: float& A
* **Description**: all coordinates can be assigned to a single float
* **Name**: operator <<(friend ofstream&)
* **Parameter(s)**:ofstream& output, Vector4 &n
* **Description**:prints Vector4 to a file
* **Name**: Normalize(Vector4)
* **Parameter(s)**: none
* **Description**: returns the Vector4 normalized
* **Name**: DotProduct(float)
* **Parameter(s)**: Vector4 B
* **Description**: returns dot product of 2 vector4s
* **Name:** CrossProduct(Vector4)
* **Parameter(s)**: Vector4 B
* **Description**: returns the cross product of a vector4 instance and a Vector4
* **Name**: operator ==(bool)
* **Parameter(s)**: Vector4 &result
* **Description**: overloaded == operator to compare vector4s
* **Name**: operator =(float)
* **Parameter(s)**: float& A
* **Description**: all coordinates can be assigned to a single float
* **Name**: operator <<(friend ofstream&)
* **Parameter(s)**:ofstream& output, Vector4 &n
* **Description**:prints Vector to a file
* **Class Name (Matrix2x2)**
  + **In Private:**
  + **Name**:m\_Matrix[4](float)
  + **Parameter(s)**:
  + **Description**: points to current node in list
  + **In Public:**

* · **Name**:Matrix2x2()
* · **Parameter(s)**: none
* · **Description**: default constructor

* · **Name**:Matrix2x2()
* · **Parameter(s)**:float x1, float x2, float y1, float y2, float z1, float z2,
* · **Description**: Constructor for floats
* · **Name**:Matrix2x2()
* · **Parameter(s)**:Vector3 columnA, Vector3 columnB, Vector3 columnC
* · **Description**: constructor for vectors
* · **Name**:operator \* (Matrix2x2)
* · **Parameter(s)**: Matrix2x2& k
* · **Description**: overloaded \* to multiply Matrices
* · **Name**: Mult(Matrix2x2)
* · **Parameter(s)**: Matrix2x2& k
* · **Description**: returns the product of matrix2x2s
* · **Name**:operator \*(Vector2)
* · **Parameter(s)**: Vector2& k
* · **Description**: overloads the \* to return the product of a matrix2 and a vector2

* · **Name**: Mult(Vector2)
* · **Parameter(s)**: Vector2& k
* · **Description**: returns the product of a vector2 and a matrix2
* · **Name**: operator ==(bool)
* · **Parameter(s)**: Matrix2x2& result
* · **Description**:overloaded == operator to compare Vector
* · **Name**: operator <<(friend ofstream& )
* · **Parameter(s)**: ofstream& output, Matrix2x2 &n
* · **Description**: prints to text file
* **Class Name (Matrix3x3)**
  + **In Private:**
  + **Name**:m\_Matrix[9](float)
  + **Parameter(s)**:
  + **Description**: points to current node in list
  + **In Public:**

* · **Name**:Matrix3x3()
* · **Parameter(s)**: none
* · **Description**: default constructor

* · **Name**:Matrix3x3()
* · **Parameter(s)**:float x1, float x2, float x3, float y1, float y2, float y3, float z1, float z2, float z3
* · **Description**: Constructor for floats
* · **Name**:Matrix3x3()
* · **Parameter(s)**:Vector3 columnA, Vector3 columnB, Vector3 columnC
* · **Description**: constructor for vectors
* · **Name**:operator \* (Matrix3x3)
* · **Parameter(s)**: Matrix3x3& k
* · **Description**: overloaded \* to multiply Matrices
* · **Name**: Mult(Matrix3x3)
* · **Parameter(s)**: Matrix3x3& k
* · **Description**: returns the product of matrix3x3s
* · **Name**:operator \*(Vector3)
* · **Parameter(s)**: Vector3& k
* · **Description**: overloads the \* to return the product of a matrix3 and a vector3

* · **Name**: Mult(Vector3)
* · **Parameter(s)**: Vector3& k
* · **Description**: returns the product of a vector3 and a matrix3
* · **Name**: operator ==(bool)
* · **Parameter(s)**: Matrix3x3& result
* · **Description**:overloaded == operator to compare Vector
* · **Name**: RotationX(Matrix3x3)
* · **Parameter(s)**: float d
* · **Description**: rotated in respect to x axis
* · **Name**: RotationY(Matrix3x3)
* · **Parameter(s)**: float d
* · **Description**: rotates matrix in respect to y axis
* · **Name**: RotationZ(Matrix3x3)
* · **Parameter(s)**: float d
* · **Description**: rotates matrix in respect to Z axis
* · **Name**: operator <<(friend ofstream& )
* · **Parameter(s)**: ofstream& output, Matrix3x3 &n
* · **Description**: prints to text file
* **Class Name (Matrix4x4)**
  + **In Private:**
    - * **Name**:m\_Matrix[16](float)
      * **Parameter(s)**:
      * **Description**: points to current node in list
  + **In Public:**

* · **Name**:Matrix4x4()
* · **Parameter(s)**: none
* · **Description**: default constructor

* · **Name**:Matrix4x4()
* · **Parameter(s)**:float x1, float x2, float x3, float x4, float y1, float y2, float y3, float y4, float z1, float z2, float z3, float z4, float w1, float w2, float w3, float w4
* · **Description**: Constructor for floats
* · **Name**:Matrix4x4()
* · **Parameter(s)**:Vector4 columnA, Vector4 columnB, Vector4 columnC, Vector4 ColumnD
* · **Description**: constructor for vectors
* · **Name**:operator \* (Matrix4x4)
* · **Parameter(s)**: Matrix4x4& k
* · **Description**: overloaded \* to multiply Matrices
* · **Name**: Mult(Matrix4x4)
* · **Parameter(s)**: Matrix4x4& k
* · **Description**: returns the product of matrix4x4s
* · **Name**:operator \*(Vector4)
* · **Parameter(s)**: Vector4& k
* · **Description**: overloads the \* to return the product of a matrix4 and a vector4

* · **Name**: Mult(Vector4)
* · **Parameter(s)**: Vector4& k
* · **Description**: returns the product of a vector4 and a matrix4
* · **Name**: operator ==(bool)
* · **Parameter(s)**: Matrix4x4& result
* · **Description**:overloaded == operator to compare Vector
* · **Name**: RotationX(Matrix4x4)
* · **Parameter(s)**: float d
* · **Description**: rotated in respect to x axis
* · **Name**: RotationY(Matrix4x4)
* · **Parameter(s)**: float d
* · **Description**: rotates matrix in respect to y axis
* · **Name**: RotationZ(Matrix4x4)
* · **Parameter(s)**: float d
* · **Description**: rotates matrix in respect to Z axis
* · **Name**: operator <<(friend ofstream& )
* · **Parameter(s)**: ofstream& output, Matrix4x4 &n
* · **Description**: prints to text file

* · **Name**: setDecimalLimit(float)
* · **Parameter(s)**: float value, unsigned int decimals
* · **Description**: sets a decimal limit

**III.** **Implementation Documentation**

III.1 **Source Code**

MyMathLib.h

#pragma once

#include <iostream>

#include <math.h>

#include <cassert>

#include <fstream>

#include <iomanip>

using namespace std;

#define pi 3.14159265359

class Vector2

{

private:

public:

Vector2() {};

Vector2(float x1, float y1) // constructor of a Vector2

: m\_x(x1), m\_y(y1) {}

Vector2 Add(Vector2 B); // Adds a Vector2 instance to a Vector2 and returns a Vector2 of the sum

Vector2 operator +(Vector2 B);

Vector2 Subtract(Vector2 B); // Subtract a Vector2 instance from a Vector4 and returns a Vector2 of the difference

Vector2 operator -(Vector2 B);

Vector2 ScalerMult(float k); // returns a scaled Vector2

Vector2 operator \*(float k);

float Magnitude(); // returns the Magnitude of a Vector2

Vector2 Normalize(); // returns the Vector2, Normalized

float DotProduct(Vector2 B); // returns dot product

bool operator == (Vector2 &result); // overloaded == operator to compare Vectors

float operator =(float& A); // can assign all coordinates of a vector to a single float

friend ofstream& operator << (ofstream & output, Vector2 &n); //prints Vector to file

float m\_x = 0.0f; // x coordinate of a Vector2

float m\_y = 0.0f; // y coordinate of a Vector2

};

class Vector3

{

public:

Vector3() {};

Vector3(float x1, float y1, float z1)

: m\_x(x1), m\_y(y1), m\_z(z1) {}

Vector3 Add(Vector3 B); // Adds a Vector4 instance to a Vector3 and returns a Vecto3 of the sum

Vector3 operator +(Vector3 B);

Vector3 Subtract(Vector3 B); // Subtract a Vector3 instance from a Vector3 and returns a Vector3 of the difference

Vector3 operator -(Vector3 B);

Vector3 ScalerMult(float k); // returns a scaled Vector4

Vector3 operator \*(float k);

float Magnitude(); // returns the Magnitude of a Vector4

Vector3 Normalize(); // returns the Vector4, Normalized

float DotProduct(Vector3 B); // returns a Vector4

Vector3 CrossProduct(Vector3 B); // returns the Cross Product of a vector3 instance and a Vector3

bool operator == (Vector3 &result); // overloaded == operator to compare Vectors

float operator =(float& A); // can assign all coordinates of a vector to a single float

friend ofstream& operator << (ofstream & output, Vector3 &n); //prints Vector to file

float m\_x = 0.0f; // x coordinate of a Vector3

float m\_y = 0.0f; // y coordinate of a Vector3

float m\_z = 0.0f; // z coordinate of a Vector3

};

class Vector4

{

public:

Vector4() {};

Vector4(float x1, float y1, float z1, float w1)

: m\_x(x1), m\_y(y1), m\_z(z1), m\_w(w1) {}

Vector4 Add(Vector4 B); // Adds a Vector4 instance to a Vector4 and returns a Vector of the sum

Vector4 operator +(Vector4 B);

Vector4 Subtract(Vector4 B); // Subtract a Vector4 instance from a Vector4 and returns a Vector4 of the difference

Vector4 operator -(Vector4 B);

Vector4 ScalerMult(float k); // returns a scaled Vector4

Vector4 operator \*(float k);

float Magnitude(); // returns the Magnitude of a Vector4

Vector4 Normalize(); // returns the Vector4, Normalized

float DotProduct(Vector4 B); // returns a Vector4

bool operator == (Vector4 &result); // overloaded == operator to compare Vectors

float operator =(float& A); // can assign all coordinates of a vector to a single float

friend ofstream& operator << (ofstream & output, Vector4 &n); //prints Vector to file

float m\_x = 0.0f; // x coordinate of a Vector4

float m\_y = 0.0f; // y coordinate of a Vector4

float m\_z = 0.0f; // z coordinate of a Vector4

float m\_w = 0.0f; // w coordinate of a Vector4

};

class Matrix2x2

{

float m\_Matrix[4]; // Container of Vector2's

public:

Matrix2x2() {}

Matrix2x2(float x1, float x2, float y1, float y2); // constructor of a matrix2x2 made up of 4 floats

Matrix2x2(Vector2 columnA, Vector2 columnB); // constructor of a Matrix2x2 made up of Vector2s

Matrix2x2 operator \* (Matrix2x2 k); // overloaded multiplication for matrix times a matrix

Matrix2x2 Mult(Matrix2x2 k); // returns the product of matrix 2x2s

Vector2 operator \* (Vector2 k); // overloaded multiplication for Matrix times a Vector

Vector2 Mult(Vector2 k); // returns the product of a Matrix2x2 and a Vector2

bool operator == (Matrix2x2 & result); // overloaded = operator to compare Vectors

friend ofstream& operator << (ofstream & output, Matrix2x2 &n); // prints matrix to file

};

class Matrix3x3

{

float m\_Matrix[9]; // Container of Matrix3x3 values

public:

Matrix3x3() {}

Matrix3x3(float x1, float x2, float x3, float y1, float y2, float y3, float z1, float z2, float z3); // constructor of a Matrix3x3 made up of 9 floats

Matrix3x3(Vector3 columnA, Vector3 columnB, Vector3 columnC); // constructor of a Matrix3x3 made up of Vector3s

Matrix3x3 operator \* (Matrix3x3& k); // overloaded multiplication for matrix times a matrix

Matrix3x3 Mult(Matrix3x3& k); // returns the product of matrix3x3s

Vector3 operator \* (Vector3 k); // overloaded multiplication for Matrix times a Vector

Vector3 Mult(Vector3 k); // returns the product of a Matrix3x3 and a Vector3

bool operator == (Matrix3x3 & result); // overloaded = operator to compare Vectors

Matrix3x3 RotateX(float d); // rotates in respect to X axis

Matrix3x3 RotateY(float d); // rotates in respect to Y axis

Matrix3x3 RotateZ(float d); // rotates in respect to z axis

friend ofstream& operator << (ofstream & output, Matrix3x3 &n); //prints matrix to file

};

class Matrix4x4

{

float m\_Matrix[16];

public:

Matrix4x4(float x1, float x2, float x3, float x4, float y1, float y2, float y3, float y4, float z1, float z2, float z3, float z4, float w1, float w2, float w3, float w4); // constructor of a matrix4x4 made up of 16 floats

Matrix4x4(Vector4 columnA, Vector4 columnB, Vector4 columnC, Vector4 columnD); // constructor of a Matrix4x4 made up of Vector4s

Matrix4x4 operator \* (Matrix4x4& k); // overloaded multiplication for matrix times a matrix

Matrix4x4 Mult(Matrix4x4& k); // returns the product of matrix 4x4s

Vector4 operator \* (Vector4 k); // overloaded multiplication for Matrix times a Vector

Vector4 Mult(Vector4 k); // returns the product of a Matrix4x4 and a Vector4

bool operator == (Matrix4x4 & result); // overloaded = operator to compare Vectors

Matrix4x4 RotateX(float d); // rotates in respect to x axis

Matrix4x4 RotateY(float d); // rotates in respect to y axis

Matrix4x4 RotateZ(float d); // rotates in respect to z axis

friend ofstream& operator << (ofstream & output, Matrix4x4 &n); // prints matrix to file

};

float setDecLimit(float value, unsigned int decimals);

MyMathLib.cpp

#include "MyMathLib.h"

Vector2 Vector2::Add(Vector2 B)

{

// a vector2 instance's coordinate values are added to another vector2's coordinate values

Vector2 newVec = Vector2(m\_x + B.m\_x, m\_y + B.m\_y);

// a new vector is created that is the sum of the two vectors

return newVec;

}

Vector2 Vector2::operator+(Vector2 B)

{

return Add(B);

}

Vector2 Vector2::Subtract(Vector2 B)

{

// a vector2 instance's coordinate values are subtracted from another vector2's coordinate values

Vector2 newVec = Vector2(m\_x - B.m\_x, m\_y - B.m\_y);

// a new vector is created that is the difference of the two vectors

return newVec;

}

Vector2 Vector2::operator-(Vector2 B)

{

return Subtract(B);

}

Vector2 Vector2::ScalerMult(float k)

{

// a scaling value is multiplied by a vector2 instance to upscale or downscale it

Vector2 newVec = Vector2(m\_x \* k, m\_y \* k);

// a new vector is created to represent the new scaled vector

return newVec;

}

Vector2 Vector2::operator\*(float k)

{

return ScalerMult(k);

}

float Vector2::Magnitude()

{

// presents the magnitude of a vector2 instance

float result = sqrt((m\_x \* m\_x) + (m\_y \* m\_y));

return result;

}

Vector2 Vector2::Normalize()

{

// alternate

/\*float x = sqrt(m\_x + m\_y);

float A = x;

float B = x;

A = A \* (m\_x / A);

B = B \* (m\_y / B);

m\_x = B;

m\_y = A;\*/

// changes the vector into a unit vector by dividing

Vector2 newVec = Vector2(m\_x / Magnitude(), m\_y / Magnitude());

return newVec;

}

float Vector2::DotProduct(Vector2 B)

{

// takes coordinate vectors

return (m\_x\*B.m\_x) + (m\_y \* B.m\_y);

}

bool Vector2::operator==(Vector2 & result)

{

// overloads == operator ro accept Vectors equal to Vectors

return m\_x == result.m\_x && m\_y == result.m\_y;

}

float Vector2::operator=(float & A)

{

return m\_x = A; m\_y = A;

}

Vector3 Vector3::Add(Vector3 B)

{

// a vector3 instance's coordinate values are added to another vector3's coordinate values

Vector3 newVec = Vector3(m\_x + B.m\_x, m\_y + B.m\_y, m\_z + B.m\_z);

// a new vector is created that is the sum of the two vectors

return newVec;

}

Vector3 Vector3::operator+(Vector3 B)

{

return Add(B);

}

Vector3 Vector3::Subtract(Vector3 B)

{

// a vector2 instance's coordinate values are subtracted from another vector2's coordinate values

Vector3 newVec = Vector3(m\_x - B.m\_x, m\_y - B.m\_y, m\_z - B.m\_z);

// a new vector is created that is the difference of the two vectors

return newVec;

}

Vector3 Vector3::operator-(Vector3 B)

{

return Subtract(B);

}

Vector3 Vector3::ScalerMult(float k)

{

// a scaling value is multiplied by a vector2 instance to upscale or downscale it

Vector3 newVec = Vector3(m\_x \* k, m\_y \* k, m\_z \* k);

return newVec;

}

Vector3 Vector3::operator\*(float k)

{

return ScalerMult(k);

}

float Vector3::Magnitude()

{

// presents the magnitude of a vector3 instance

float result = sqrt((m\_x \* m\_x) + (m\_y \* m\_y) + (m\_z \* m\_z));

return result;

}

Vector3 Vector3::Normalize()

{

// changes the vector into a unit vector by dividing

Vector3 newVec = Vector3(m\_x / Magnitude(), m\_y / Magnitude(), m\_z / Magnitude());

return newVec;

}

float Vector3::DotProduct(Vector3 B)

{

// takes coordinate vectors multiplies them and returns a single number

return (m\_x\*B.m\_x) + (m\_y \* B.m\_y) + (m\_z \* B.m\_z);

}

Vector3 Vector3::CrossProduct(Vector3 B)

{

//x = (AyBz) - (AzBy); y = (AzBx) - (AxBz); z = (AxBy) - (AyBx)

Vector3 newVec = Vector3((m\_y \* B.m\_z) - (m\_z \* B.m\_y), (m\_z \* B.m\_x) - (m\_x \* B.m\_z), (m\_x \* B.m\_y) - (m\_y \* B.m\_x));

return newVec;

}

bool Vector3::operator==(Vector3 & result)

{

// overloads == operator ro accept Vectors equal to Vectors

return m\_x == result.m\_x && m\_y == result.m\_y && m\_z == result.m\_z;

}

float Vector3::operator=(float & A)

{

return m\_x = A; m\_y = A; m\_z = A;

}

Vector4 Vector4::Add(Vector4 B)

{

// a vector4 instance's coordinate values are added to another vector4's coordinate values

Vector4 newVec = Vector4(m\_x + B.m\_x, m\_y + B.m\_y, m\_z + B.m\_z, m\_w + B.m\_w);

// a new vector is created that is the sum of the two vectors

return newVec;

}

Vector4 Vector4::operator+(Vector4 B)

{

return Add(B);

}

Vector4 Vector4::Subtract(Vector4 B)

{

// a vector2 instance's coordinate values are subtracted from another vector2's coordinate values

Vector4 newVec = Vector4(m\_x - B.m\_x, m\_y - B.m\_y, m\_z - B.m\_z, m\_w - B.m\_w);

// a new vector is created that is the difference of the two vectors

return newVec;

}

Vector4 Vector4::operator-(Vector4 B)

{

return Subtract(B);

}

Vector4 Vector4::ScalerMult(float k)

{

// a scaling value is multiplied by a vector2 instance to upscale or downscale it

Vector4 newVec = Vector4(m\_x \* k, m\_y \* k, m\_z \* k, m\_w \* k);

// a new vector is created to represent the new scaled vector

return newVec;

}

Vector4 Vector4::operator\*(float k)

{

return ScalerMult(k);

}

float Vector4::Magnitude()

{

// presents the magnitude of a vector4 instance

float result = sqrt((m\_x \* m\_x) + (m\_y \* m\_y) + (m\_z \* m\_z) + (m\_w \* m\_w));

return result;

}

Vector4 Vector4::Normalize()

{

// changes the vector into a unit vector by dividing

Vector4 newVec = Vector4(m\_x / Magnitude(), m\_y / Magnitude(), m\_z / Magnitude(), m\_w / Magnitude());

return newVec;

}

float Vector4::DotProduct(Vector4 B)

{

// takes coordinate vectors multiplies them and returns a single number

return (m\_x\*B.m\_x) + (m\_y \* B.m\_y) + (m\_z \* B.m\_z);

}

bool Vector4::operator==(Vector4 & result)

{

// overloads == operator ro accept Vectors equal to Vectors

return m\_x == result.m\_x && m\_y == result.m\_y;

}

float Vector4::operator=(float & A)

{

return m\_w = A; m\_x = A; m\_y = A; m\_z = A;

}

Matrix2x2::Matrix2x2(float x1, float x2, float y1, float y2)

{

/\* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| |

| MatShape[0] = x1 MatShape[1] = x2 |

| MatShape[2] = y1 MatShape[3] = y2 |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_| \*/

m\_Matrix[0] = x1;

m\_Matrix[1] = x2;

m\_Matrix[2] = y1;

m\_Matrix[3] = y2;

// matrix2v2 that is made up of 4 float arguments

}

Matrix2x2::Matrix2x2(Vector2 columnA, Vector2 columnB)

{

m\_Matrix[0] = columnA.m\_x;

m\_Matrix[1] = columnB.m\_x;

m\_Matrix[2] = columnA.m\_y;

m\_Matrix[3] = columnB.m\_y;

// Matrix2x2 that is made up of 2 Vector arguments

}

Matrix2x2 Matrix2x2::operator\*(Matrix2x2 k)

{

// overloads the \* operator be able to multiply 2x2 matrices

return Mult(k);

}

Matrix2x2 Matrix2x2::Mult(Matrix2x2 k)

{

// allows you to multiply 2x2 Matrices

Matrix2x2 newMat = Matrix2x2((m\_Matrix[0] \* k.m\_Matrix[0]) + (m\_Matrix[1] \* k.m\_Matrix[2]),

(m\_Matrix[0] \* k.m\_Matrix[1]) + (m\_Matrix[1] \* k.m\_Matrix[3]),

(m\_Matrix[2] \* k.m\_Matrix[0]) + (m\_Matrix[3] \* k.m\_Matrix[2]),

(m\_Matrix[2] \* k.m\_Matrix[1]) + (m\_Matrix[3] \* k.m\_Matrix[3]));

return newMat;

}

Vector2 Matrix2x2::operator\*(Vector2 k)

{

// overloads the \* operator be able to multiply 2x2 matrices by Vector2s and returns a Vector

return Mult(k);

}

Vector2 Matrix2x2::Mult(Vector2 k)

{

// allows you to multply a Matrix 2x2 by a Vector2

Vector2 resultVec = Vector2(

(k.m\_x \* m\_Matrix[0]) + (k.m\_x \* m\_Matrix[1]),

(k.m\_y \* m\_Matrix[2]) + (k.m\_y \* m\_Matrix[3]));

return resultVec;

}

bool Matrix2x2::operator==(Matrix2x2 & result)

{

// overloads == operator to know to tell if matrices are equal

return m\_Matrix[0] == result.m\_Matrix[0] && m\_Matrix[1] == result.m\_Matrix[1] && m\_Matrix[2] == result.m\_Matrix[2] && m\_Matrix[3] == result.m\_Matrix[3];

}

ofstream & operator<<(ofstream & output, Vector2 & n)

{

output << "<" << setDecLimit(n.m\_x, 3u) << ", " << setDecLimit(n.m\_y, 3u) << ">\n";

return output;

}

ofstream & operator<<(ofstream & output, Vector3 & n)

{

output << "<" << setDecLimit(n.m\_x, 3u) << ", " << setDecLimit(n.m\_y, 3u) << ", " << setDecLimit(n.m\_z, 3u) << ">\n";

return output;

}

ofstream & operator<<(ofstream & output, Vector4 & n)

{

output << "<" << setDecLimit(n.m\_x, 3u) << ", " << setDecLimit(n.m\_y, 3u) << ", " << setDecLimit(n.m\_z, 3u) << ", " << setDecLimit(n.m\_w, 3u) << ">\n";

return output;

}

ofstream & operator<<(ofstream & output, Matrix2x2 & n)

{

// this overloaded operator allows me to be able to print a Matrix to a file

output << "X {" << setDecLimit(n.m\_Matrix[0], 3u) << ", " << setDecLimit(n.m\_Matrix[1], 3u) <<"}\n";

output << "Y {" << setDecLimit(n.m\_Matrix[2], 3u) << ", " << setDecLimit(n.m\_Matrix[3], 3u) <<"}\n";

return output;

}

ofstream & operator<<(ofstream & output, Matrix3x3 & n)

{

// this overloaded operator allows me to be able to print a Matrix to a file

output << "X {" << setDecLimit(n.m\_Matrix[0], 3u) << ", " << setDecLimit(n.m\_Matrix[1], 3u) << ", " << setDecLimit(n.m\_Matrix[2], 3u) << "}\n";

output << "Y {" << setDecLimit(n.m\_Matrix[3], 3u) << ", " << setDecLimit(n.m\_Matrix[4], 3u) << ", " << setDecLimit(n.m\_Matrix[5], 3u) << "}\n";

output << "Z {" << setDecLimit(n.m\_Matrix[6], 3u) << ", " << setDecLimit(n.m\_Matrix[7], 3u) << ", " << setDecLimit(n.m\_Matrix[8], 3u) << "}\n";

return output;

}

ofstream & operator<<(ofstream & output, Matrix4x4 & n)

{

// this overloaded operator allows me to be able to print a Matrix to a file

output << "X {" << setDecLimit(n.m\_Matrix[0], 3u) << ", " << setDecLimit(n.m\_Matrix[1], 3u) << ", " << setDecLimit(n.m\_Matrix[2], 3u) << ", " << setDecLimit(n.m\_Matrix[3], 3u) << "}\n";

output << "Y {" << setDecLimit(n.m\_Matrix[4], 3u) << ", " << setDecLimit(n.m\_Matrix[5], 3u) << ", " << setDecLimit(n.m\_Matrix[6], 3u) << ", " << setDecLimit(n.m\_Matrix[7], 3u) << "}\n";

output << "Z {" << setDecLimit(n.m\_Matrix[8], 3u) << ", " << setDecLimit(n.m\_Matrix[9], 3u) << ", " << setDecLimit(n.m\_Matrix[10], 3u) << ", " << setDecLimit(n.m\_Matrix[11], 3u) << "}\n";

output << "W {" << setDecLimit(n.m\_Matrix[12], 3u) << ", "<< setDecLimit(n.m\_Matrix[13], 3u) << ", " << setDecLimit(n.m\_Matrix[14], 3u) << ", " << setDecLimit(n.m\_Matrix[15], 3u) << "}\n";

return output;

}

Matrix3x3::Matrix3x3(float x1, float x2, float x3, float y1, float y2, float y3, float z1, float z2, float z3)

{

// Matrix3x3 made of 9 float arguments

m\_Matrix[0] = x1; m\_Matrix[1] = x2; m\_Matrix[2] = x3;

m\_Matrix[3] = y1; m\_Matrix[4] = y2; m\_Matrix[5] = y3;

m\_Matrix[6] = z1; m\_Matrix[7] = z2; m\_Matrix[8] = z3;

}

Matrix3x3::Matrix3x3(Vector3 columnA, Vector3 columnB, Vector3 columnC)

{

// Matrix3x3 made up of 3 Vector3 arguments

m\_Matrix[0] = columnA.m\_x; m\_Matrix[1] = columnB.m\_x; m\_Matrix[2] = columnC.m\_x;

m\_Matrix[3] = columnA.m\_y; m\_Matrix[4] = columnB.m\_y; m\_Matrix[5] = columnC.m\_y;

m\_Matrix[6] = columnA.m\_z; m\_Matrix[7] = columnB.m\_z; m\_Matrix[8] = columnC.m\_z;

}

Matrix3x3 Matrix3x3::operator\*(Matrix3x3& k)

{

// overloads the \* operator be able to multiply 3x3 matrices

return Mult(k);

}

Matrix3x3 Matrix3x3::Mult(Matrix3x3& k)

{

// allows you to multply 2 3x3 Matrices

Matrix3x3 newMat = Matrix3x3(

/\*x1\*/ (m\_Matrix[0] \* k.m\_Matrix[0]) + (m\_Matrix[1] \* k.m\_Matrix[3]) + (m\_Matrix[2] \* k.m\_Matrix[6]),

/\*x2\*/ (m\_Matrix[0] \* k.m\_Matrix[1]) + (m\_Matrix[1] \* k.m\_Matrix[4]) + (m\_Matrix[2] \* k.m\_Matrix[7]),

/\*x3\*/ (m\_Matrix[0] \* k.m\_Matrix[2]) + (m\_Matrix[1] \* k.m\_Matrix[5]) + (m\_Matrix[2] \* k.m\_Matrix[8]),

/\*y1\*/ (m\_Matrix[3] \* k.m\_Matrix[0]) + (m\_Matrix[4] \* k.m\_Matrix[3]) + (m\_Matrix[5] \* k.m\_Matrix[6]),

/\*y2\*/ (m\_Matrix[3] \* k.m\_Matrix[1]) + (m\_Matrix[4] \* k.m\_Matrix[4]) + (m\_Matrix[5] \* k.m\_Matrix[7]),

/\*y3\*/ (m\_Matrix[3] \* k.m\_Matrix[2]) + (m\_Matrix[4] \* k.m\_Matrix[5]) + (m\_Matrix[5] \* k.m\_Matrix[8]),

/\*z1\*/ (m\_Matrix[6] \* k.m\_Matrix[0]) + (m\_Matrix[7] \* k.m\_Matrix[3]) + (m\_Matrix[8] \* k.m\_Matrix[6]),

/\*z2\*/ (m\_Matrix[6] \* k.m\_Matrix[1]) + (m\_Matrix[7] \* k.m\_Matrix[4]) + (m\_Matrix[8] \* k.m\_Matrix[7]),

/\*z3\*/ (m\_Matrix[6] \* k.m\_Matrix[2]) + (m\_Matrix[7] \* k.m\_Matrix[5]) + (m\_Matrix[8] \* k.m\_Matrix[8]));

return newMat;

}

Vector3 Matrix3x3::operator\*(Vector3 k)

{

// overloads the \* operator be able to multiply 3x3 matrices by Vector3s

return Mult(k);

}

Vector3 Matrix3x3::Mult(Vector3 k)

{

// this function allows you to multiply a Matrix3x3 by Vector3s

Vector3 newVec = Vector3(

/\*x\*/ (k.m\_x \* m\_Matrix[0]) + (k.m\_x \* m\_Matrix[1]) + (k.m\_x \* m\_Matrix[2]),

/\*y\*/ (k.m\_y \* m\_Matrix[3]) + (k.m\_y \* m\_Matrix[4]) + (k.m\_y \* m\_Matrix[5]),

/\*z\*/ (k.m\_z \* m\_Matrix[6]) + (k.m\_z \* m\_Matrix[7]) + (k.m\_z \* m\_Matrix[8]));

return newVec;

}

bool Matrix3x3::operator==(Matrix3x3 & result)

{

// overloads == operator to know to tell if matrices are equal

return m\_Matrix[0] == result.m\_Matrix[0] && m\_Matrix[1] == result.m\_Matrix[1] && m\_Matrix[2] == result.m\_Matrix[2] && m\_Matrix[3] == result.m\_Matrix[3];

}

Matrix3x3 Matrix3x3::RotateX(float d)

{

// rotates a matrix on the X axis by the "degrees" given as an argument

Matrix3x3 newMat = Matrix3x3(

m\_Matrix[0], m\_Matrix[1], m\_Matrix[2],

m\_Matrix[3], cos(d), -sin(d),

m\_Matrix[6], sin(d), cos(d));

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix3x3 instance and returns it

return \*this;

}

Matrix3x3 Matrix3x3::RotateY(float d)

{

// rotates a matrix on the Y axis by the "degrees" given as an argument

Matrix3x3 newMat = Matrix3x3(

cos(d), m\_Matrix[1], sin(d),

m\_Matrix[3], m\_Matrix[4], m\_Matrix[5],

-sin(d), m\_Matrix[7], cos(d));

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix3x3 instance and returns it

return \*this;

}

Matrix3x3 Matrix3x3::RotateZ(float d)

{

// rotates a matrix on the X axis by the "degrees" given as an argument

Matrix3x3 newMat = Matrix3x3(

cos(d), -sin(d), m\_Matrix[2],

sin(d), cos(d), m\_Matrix[5],

m\_Matrix[6], m\_Matrix[7], m\_Matrix[8]);

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix3x3 instance and returns it

return \*this;

}

Matrix4x4::Matrix4x4(float x1, float x2, float x3, float x4, float y1, float y2, float y3, float y4, float z1, float z2, float z3, float z4, float w1, float w2, float w3, float w4)

{

// Matrix4x4 made up of 16 floats

m\_Matrix[0] = x1; m\_Matrix[1] = x2; m\_Matrix[2] = x3; m\_Matrix[3] = x4;

m\_Matrix[4] = y1; m\_Matrix[5] = y2; m\_Matrix[6] = y3; m\_Matrix[7] = x4;

m\_Matrix[8] = z1; m\_Matrix[9] = z2; m\_Matrix[10] = z3; m\_Matrix[11] = z4;

m\_Matrix[12] = w1; m\_Matrix[13] = w2; m\_Matrix[14] = w3; m\_Matrix[15] = w4;

}

Matrix4x4::Matrix4x4(Vector4 columnA, Vector4 columnB, Vector4 columnC, Vector4 columnD)

{

// Matrix4x4 made up of 4 Vector4 arguments

m\_Matrix[0] = columnA.m\_x; m\_Matrix[1] = columnB.m\_x; m\_Matrix[2] = columnC.m\_x; m\_Matrix[3] = columnD.m\_x;

m\_Matrix[4] = columnA.m\_y; m\_Matrix[5] = columnB.m\_y; m\_Matrix[6] = columnC.m\_y; m\_Matrix[7] = columnD.m\_y;

m\_Matrix[8] = columnA.m\_z; m\_Matrix[9] = columnB.m\_z; m\_Matrix[10] = columnC.m\_z; m\_Matrix[11] = columnD.m\_z;

m\_Matrix[12] = columnA.m\_w; m\_Matrix[13] = columnB.m\_w; m\_Matrix[14] = columnC.m\_w; m\_Matrix[15] = columnD.m\_w;

}

Matrix4x4 Matrix4x4::operator\*(Matrix4x4& k)

{

// overloads the \* operator be able to multiply 3x3 matrices

return Mult(k);

}

Matrix4x4 Matrix4x4::Mult(Matrix4x4& k)

{

// allows you to multiply 4x4 matrices

Matrix4x4 newMat = Matrix4x4(

/\*x1\*/ (m\_Matrix[0] \* k.m\_Matrix[0]) + (m\_Matrix[1] \* k.m\_Matrix[4]) + (m\_Matrix[2] \* k.m\_Matrix[8]) + (m\_Matrix[3] \* k.m\_Matrix[12]),

/\*x2\*/ (m\_Matrix[0] \* k.m\_Matrix[1]) + (m\_Matrix[1] \* k.m\_Matrix[5]) + (m\_Matrix[2] \* k.m\_Matrix[9]) + (m\_Matrix[3] \* k.m\_Matrix[13]),

/\*x3\*/ (m\_Matrix[0] \* k.m\_Matrix[2]) + (m\_Matrix[1] \* k.m\_Matrix[6]) + (m\_Matrix[2] \* k.m\_Matrix[10]) + (m\_Matrix[3] \* k.m\_Matrix[14]),

/\*x4\*/ (m\_Matrix[0] \* k.m\_Matrix[3]) + (m\_Matrix[1] \* k.m\_Matrix[7]) + (m\_Matrix[2] \* k.m\_Matrix[11]) + (m\_Matrix[3] \* k.m\_Matrix[15]),

/\*y1\*/ (m\_Matrix[4] \* k.m\_Matrix[0]) + (m\_Matrix[5] \* k.m\_Matrix[4]) + (m\_Matrix[6] \* k.m\_Matrix[8]) + (m\_Matrix[7] \* k.m\_Matrix[12]),

/\*y2\*/ (m\_Matrix[4] \* k.m\_Matrix[1]) + (m\_Matrix[5] \* k.m\_Matrix[5]) + (m\_Matrix[6] \* k.m\_Matrix[9]) + (m\_Matrix[7] \* k.m\_Matrix[13]),

/\*y3\*/ (m\_Matrix[4] \* k.m\_Matrix[2]) + (m\_Matrix[5] \* k.m\_Matrix[6]) + (m\_Matrix[6] \* k.m\_Matrix[10]) + (m\_Matrix[7] \* k.m\_Matrix[14]),

/\*y4\*/ (m\_Matrix[4] \* k.m\_Matrix[3]) + (m\_Matrix[5] \* k.m\_Matrix[7]) + (m\_Matrix[6] \* k.m\_Matrix[11]) + (m\_Matrix[7] \* k.m\_Matrix[15]),

/\*z1\*/ (m\_Matrix[8] \* k.m\_Matrix[0]) + (m\_Matrix[9] \* k.m\_Matrix[4]) + (m\_Matrix[10] \* k.m\_Matrix[8]) + (m\_Matrix[11] \* k.m\_Matrix[12]),

/\*z2\*/ (m\_Matrix[8] \* k.m\_Matrix[1]) + (m\_Matrix[9] \* k.m\_Matrix[5]) + (m\_Matrix[10] \* k.m\_Matrix[9]) + (m\_Matrix[11] \* k.m\_Matrix[13]),

/\*z3\*/ (m\_Matrix[8] \* k.m\_Matrix[2]) + (m\_Matrix[9] \* k.m\_Matrix[6]) + (m\_Matrix[10] \* k.m\_Matrix[10]) + (m\_Matrix[11] \* k.m\_Matrix[14]),

/\*z4\*/ (m\_Matrix[8] \* k.m\_Matrix[3]) + (m\_Matrix[9] \* k.m\_Matrix[7]) + (m\_Matrix[10] \* k.m\_Matrix[11]) + (m\_Matrix[11] \* k.m\_Matrix[15]),

/\*w1\*/ (m\_Matrix[12] \* k.m\_Matrix[0]) + (m\_Matrix[13] \* k.m\_Matrix[4]) + (m\_Matrix[14] \* k.m\_Matrix[8]) + (m\_Matrix[15] \* k.m\_Matrix[12]),

/\*w2\*/ (m\_Matrix[12] \* k.m\_Matrix[1]) + (m\_Matrix[13] \* k.m\_Matrix[5]) + (m\_Matrix[14] \* k.m\_Matrix[9]) + (m\_Matrix[15] \* k.m\_Matrix[13]),

/\*w3\*/ (m\_Matrix[12] \* k.m\_Matrix[2]) + (m\_Matrix[13] \* k.m\_Matrix[6]) + (m\_Matrix[14] \* k.m\_Matrix[10]) + (m\_Matrix[15] \* k.m\_Matrix[14]),

/\*w4\*/ (m\_Matrix[12] \* k.m\_Matrix[3]) + (m\_Matrix[13] \* k.m\_Matrix[7]) + (m\_Matrix[14] \* k.m\_Matrix[11]) + (m\_Matrix[15] \* k.m\_Matrix[15]));

return newMat;

}

Vector4 Matrix4x4::operator\*(Vector4 k)

{

// overloads the \* operator be able to multiply 4x4 matrices by Vector4s

return Mult(k);

}

Vector4 Matrix4x4::Mult(Vector4 k)

{

// overloads the \* operator be able to multiply 4x4 matrices by Vector4s

Vector4 newVec = Vector4(

/\*x\*/(k.m\_x \* m\_Matrix[0]) + (k.m\_x \* m\_Matrix[1]) + (k.m\_x \* m\_Matrix[2]) + (k.m\_x \* m\_Matrix[3]),

/\*y\*/(k.m\_y \* m\_Matrix[4]) + (k.m\_y \* m\_Matrix[5]) + (k.m\_y \* m\_Matrix[6]) + (k.m\_y \* m\_Matrix[7]),

/\*z\*/(k.m\_y \* m\_Matrix[8]) + (k.m\_z \* m\_Matrix[9]) + (k.m\_z \* m\_Matrix[10]) + (k.m\_z \* m\_Matrix[11]),

/\*w\*/(k.m\_w \* m\_Matrix[12]) + (k.m\_w \* m\_Matrix[13]) + (k.m\_w \* m\_Matrix[14]) + (k.m\_w \* m\_Matrix[15]));

return newVec;

}

bool Matrix4x4::operator==(Matrix4x4 & result)

{

// overloads == operator to know to tell if matrices are equal

return m\_Matrix[0] == result.m\_Matrix[0] && m\_Matrix[1] == result.m\_Matrix[1] && m\_Matrix[2] == result.m\_Matrix[2] && m\_Matrix[3] == result.m\_Matrix[3];

}

Matrix4x4 Matrix4x4::RotateX(float d)

{

// rotates a matrix on the X axis by the "degrees" given as an argument

Matrix4x4 newMat = Matrix4x4(

m\_Matrix[0], m\_Matrix[1], m\_Matrix[2], m\_Matrix[3],

m\_Matrix[4], cos(d), -sin(d), m\_Matrix[7],

m\_Matrix[8], sin(d), cos(d), m\_Matrix[11],

m\_Matrix[12], m\_Matrix[13], m\_Matrix[14], m\_Matrix[15]);

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix4x4 instance and returns it

return \*this;

}

Matrix4x4 Matrix4x4::RotateY(float d)

{

// rotates a matrix on the Y axis by the "degrees" given as an argument

Matrix4x4 newMat = Matrix4x4(

cos(d), m\_Matrix[1], sin(d), m\_Matrix[3],

m\_Matrix[4], m\_Matrix[5], m\_Matrix[6], m\_Matrix[7],

-sin(d), m\_Matrix[9], cos(d), m\_Matrix[11],

m\_Matrix[12], m\_Matrix[13], m\_Matrix[14], m\_Matrix[15]);

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix4x4 instance and returns it

return \*this;

}

Matrix4x4 Matrix4x4::RotateZ(float d)

{

// rotates a matrix on the Z axis by the "degrees" given as an argument

Matrix4x4 newMat = Matrix4x4(

cos(d), -sin(d), m\_Matrix[2], m\_Matrix[3],

sin(d), cos(d), m\_Matrix[6], m\_Matrix[7],

m\_Matrix[8], m\_Matrix[9], m\_Matrix[10], m\_Matrix[11],

m\_Matrix[12], m\_Matrix[13], m\_Matrix[14], m\_Matrix[15]);

\*this = \*this \* newMat; // multiplys the rotated matrix by the actual matrix4x4 instance and returns it

return \*this;

}

float setDecLimit(float value, unsigned int decimals = 0)

{

// function that sets a decimal limit used to set a fixed limit when printing

float power = powf(10.0f, (float)decimals);

float setDecLimitedValue = truncf(value \* power) / power;

if (setDecLimitedValue == -0)

setDecLimitedValue = 0;

return setDecLimitedValue;

}

**IV.** **READ ME**

In order to access this file, copy and paste the following link into browser:

https://github.com/devinwiggins/MathforGames

You can either click clone or download, and download the zip file or you can go to each individual folder or file and download them separately.

After you start up the MyMathLib.exe the console will provide you with further information.