Matrix

```
matrix::matrix(int r, int c, double diagVal){
	m = NULL;
	Resize(r, c, true);
	for (int i = 0; i < rows; i++)
		m[i][i] = diagVal;
}

matrix::matrix(int r, int c, double *vals)  {
	m = NULL;
	Resize(r, c, true);

	for (int i = 0; i < rows; i++)
	{
		for (int j = 0; j < cols; j++)
		{
			m[i][j] = vals[i * cols + j];
		}
	}
}

matrix::matrix(const matrix &m1){
	m = NULL;
	Resize(m1.rows, m1.cols, false);
	Copy(m1);
}

matrix::~matrix(){
	delete [] m[0];
	delete [] m;
}

matrix &matrix::operator=(const matrix &m1){
	Resize(m1.rows, m1.cols, false);
	Copy(m1);
	return *this;
}

void matrix::Resize(int r, int c, bool clearNewMem){
	if (m != NULL)
	{
		delete [] m[0];
		delete [] m;
	}

	rows = r; cols = c;
	m = new double*[rows];
	assert(m != NULL);

	m[0] = new double[rows*cols];
	assert(m[0] != NULL);

	for (int i = 1; i < rows; i++)
		m[i] = m[i-1] + cols;

	if (clearNewMem)
		memset(m[0], 0, rows*cols*sizeof(double));
}
```

Matrix

```
void matrix::Copy(const matrix &m1){
	assert(rows == m1.rows && cols == m1.cols);

	for (int i = 0; i < rows; i++)
	{
		for (int j = 0; j < cols; j++)
		{
			m[i][j] = m1.m[i][j];
		}
	}
}

void matrix::Print(int nPlaces) const{
	std::streamsize oldP = cout.precision();
	cout.precision(nPlaces);
	for (int i = 0; i < rows; i++)
	{
		for (int j = 0; j < cols; j++)
			cout << m[i][j] << "\t";
		cout << endl;
	}
	cout.precision(oldP);
}

void matrix::SwapRows(int i, int j){
	double tempI;
	double tempJ;
	for (int u = 0; u < cols; u++){
		tempI = m[i][u];
		tempJ = m[j][u];
		m[i][u] = tempJ;
		m[j][u] = tempI;
	}

}

void matrix::AddRowMultiple(double d, int i, int j, int startCol){
	for (startCol; startCol < cols; startCol++){
		m[j][startCol] = m[i][startCol]*d + m[j][startCol];

	}

}
void matrix::MultiplyRow(double d, int i){
	for (int u = 0; u < cols; u++){
		m[i][u] = m[i][u]*d;

	}
}

double matrix::Determinant(void){
	assert(rows == cols);

	return 1;
}

bool matrix::GaussJordanUnstable(void){
	for (int j = 0; j < min(rows, cols); j++)
```

```
	{
		if (IsZero(m[j][j]))
			return false;

		MultiplyRow(1.0 / m[j][j], j);
		for (int i = 0; i < rows; i++)
		{
			if (j != i)
				AddRowMultiple(-m[i][j], j, i, j);
		}
	}
	return true;
}

bool matrix::GaussJordan(void){
	double maxValue;
	double maxAbsValue;
	int iwillswap;

	for(int i=0; i < rows; i++){
		maxValue = m[i][i];
		maxAbsValue = abs(maxValue);
		iwillswap = i;
		for(int j=i+1; j < rows; j++){
			if(abs(m[j][i]) > maxAbsValue){
				maxAbsValue = abs(m[j][i]);
				maxValue = m[j][i];
				iwillswap = j;
			}
		}
		if(maxValue == 0){
			return false;
		}else if(maxAbsValue > 0){
			SwapRows(i,iwillswap);
			MultiplyRow((1/maxValue),i );
			for(int j=0; j < rows; j++){
				if( i != j ){
					AddRowMultiple(-m[j][i],i,j,0);
				}
			}
		}
	}
	return true;
}

matrix matrix::SubMatrix(int rStart, int cStart, int nRows, int nCols){
	matrix result(nRows, nCols);
	for (int i = 0; i < nRows; i++)
		for (int j = 0; j < nCols; j++)
			result.m[i][j] = m[i+rStart][j+cStart];

	return result;
}

bool matrix::Solve(matrix &result, const matrix &RHS){
	assert(rows == RHS.rows);

	matrix temp = Augment(*this, RHS);
```

```
	if (temp.GaussJordan())
	{
		result = temp.SubMatrix(0, cols, RHS.rows, RHS.cols);
		return true;
	}
	else
	{
		result = temp;
		return false;
	}
}

matrix matrix::SolveUnstable(const matrix &RHS) {
	assert(rows == RHS.rows);
	matrix temp = Augment(*this, RHS);
	temp.GaussJordanUnstable();
	return temp.SubMatrix(0, cols, RHS.rows, RHS.cols);
}

bool matrix::Inverse(matrix &inv){
	assert(rows == cols);

	matrix identity(rows, cols, 1.0);
	inv.Resize(rows, cols);
	return Solve(inv, identity);
}

matrix matrix::InverseUnstable(){
	assert(rows == cols);
	return SolveUnstable(matrix(rows, cols, 1.0));
}

matrix matrix::Transpose(){
	matrix result (cols, rows);

	for (int i = 0; i < rows; i++){
		for (int j = 0; j < cols; j++){
			result[i][j] = m[j][i];
		}
	}

	return result;
}

matrix matrix::Augment(const matrix &m1, const matrix &m2){
	assert(m1.rows == m2.rows);

	int i, j;
	int rows = m1.rows;
	int cols = m1.cols + m2.cols;
	matrix result(rows, cols);

	for (i = 0; i < rows; i++)
	{
		for (j = 0; j < m1.cols; j++)
			result.m[i][j] = m1.m[i][j];
		for (j = 0; j < m2.cols; j++)
			result.m[i][j+m1.cols] = m2.m[i][j];
```

```
	}
	return result;
}

void matrix::MakeRotationX(double alpha){
	(*this) = matrix(4, 4, 1.0);

	m[0][0] = 1;
	m[0][1] = 0;
	m[0][2] = 0;
	m[0][3] = 0;

	m[1][0] = 0;
	m[1][1] = cos(alpha*(PI/180));
	m[1][2] = -sin(alpha*(PI/180));
	m[1][3] = 0;

	m[2][0] = 0;
	m[2][1] = sin(alpha*(PI/180));
	m[2][2] = cos(alpha*(PI/180));
	m[2][3] = 0;

	m[3][0] = 0;
	m[3][1] = 0;
	m[3][2] = 0;
	m[3][3] = 1;
}

void matrix::MakeRotationY(double alpha){
	(*this) = matrix(4, 4, 1.0);

	m[0][0] = cos(alpha*(PI/180));
	m[0][1] = 0;
	m[0][2] = sin(alpha*(PI/180));
	m[0][3] = 0;

	m[1][0] = 0;
	m[1][1] = 1;
	m[1][2] = 0;
	m[1][3] = 0;

	m[2][0] = -sin(alpha*(PI/180));
	m[2][1] = 0;
	m[2][2] = cos(alpha*(PI/180));
	m[2][3] = 0;

	m[3][0] = 0;
	m[3][1] = 0;
	m[3][2] = 0;
	m[3][3] = 1;
}

void matrix::MakeRotationZ(double alpha){
	(*this) = matrix(4, 4, 1.0);

	m[0][0] = cos(alpha*(PI/180));
	m[0][1] = -sin(alpha*(PI/180));
	m[0][2] = 0;
```

```
	m[0][3] = 0;

	m[1][0] = sin(alpha*(PI/180));
	m[1][1] = cos(alpha*(PI/180));
	m[1][2] = 0;
	m[1][3] = 0;

	m[2][0] = 0;
	m[2][1] = 0;
	m[2][2] = 1;
	m[2][3] = 0;

	m[3][0] = 0;
	m[3][1] = 0;
	m[3][2] = 0;
	m[3][3] = 1;
}

void matrix::MakeRotation(double alpha, const vector &v){
	(*this) = matrix(4, 4, 1.0);

	vector temp =  v;
	temp.Normalize();
	alpha = alpha * (PI/180);

	m[0][0] = temp[0]* temp[0] * ( 1 - cos( alpha ) )  + cos( alpha );
	m[0][1] = (temp[0] * temp[1]) * ( 1 - cos( alpha ) ) - (temp[2] * sin( alpha ) );
	m[0][2] = (temp[0] * temp[2]) * ( 1 - cos( alpha ) ) + (temp[1] * sin( alpha ) );
	m[0][3] = 0;

	m[1][0] = (temp[0] * temp[1]) * ( 1 - cos( alpha ) ) + (temp[2] * sin( alpha ) );
	m[1][1] = temp[1] * temp[1] * ( 1 - cos( alpha ) ) + cos( alpha );
	m[1][2] = (temp[1] * temp[2]) * ( 1 - cos( alpha ) ) - (temp[0] * sin( alpha ) );
	m[1][3] = 0;

	m[2][0] = (temp[0] * temp[2]) * ( 1 - cos( alpha ) ) - (temp[1] * sin( alpha ) );
	m[2][1] = (temp[1] * temp[2]) * ( 1 - cos( alpha ) ) + (temp[0] * sin( alpha ) );
	m[2][2] = temp[2] * temp[2] * ( 1 - cos( alpha ) ) + cos( alpha );
	m[2][3] = 0;

	m[3][0] = 0;
	m[3][1] = 0;
	m[3][2] = 0;
	m[3][3] = 1;
}

void matrix::MakeScale(double x, double y, double z){
	(*this) = matrix(4, 4, 1.0);

	for (int i = 0; i < 3; i++){
		for (int j = 0; j < 3; j++){
			if(i==0){
				m[i][j] = m[i][j] * x;
			}else if(i==1){
				m[i][j] = m[i][j] * y;
			}else if(i==2){
				m[i][j] = m[i][j] * z;
			}
```

```
			}
		}
}

void matrix::MakeTranslation(double x, double y, double z){
		(*this) = matrix(4, 4, 1.0);

		m[0][3] = x;
		m[1][3] = y;
		m[2][3] = z;
		m[3][3] = 1;
}

matrix matrix::Identity(int size) { return matrix(size, size, 1.0); }
matrix matrix::Zero(int size) { return matrix(size, size, 0.0); }
matrix matrix::RotationX(double alpha){matrix m(4, 4); m.MakeRotationX(alpha); return m;}
matrix matrix::RotationY(double alpha){matrix m(4, 4); m.MakeRotationY(alpha); return m;
}
matrix matrix::RotationZ(double alpha){matrix m(4, 4); m.MakeRotationZ(alpha); return m;}
matrix matrix::Rotation(double alpha, const vector &v) { matrix m(4, 4);
m.MakeRotation(alpha, v); return m; }
matrix matrix::Scale(double x, double y, double z) { matrix m(4, 4); m.MakeScale(x, y,
		z); return m; }
matrix matrix::Translation(double x, double y, double z) { matrix m(4, 4);
m.MakeTranslation(x, y, z); return m; }

bool IsZero(const matrix &m){
		for (int i = 0; i < m.Rows(); i++){
			for (int j = 0; j < m.Cols(); j++){
				if( !IsEqual(m[i][j],0) ){
					return false;
				}
			}
		}
		return true;
}

bool IsIdentity(const matrix &m){
		for (int i = 0; i < m.Rows(); i++){
			for (int j = 0; j < m.Cols(); j++){
				if(i==j){
					if(m[i][j] != 1){
						return false;
					}
				}else{
					if(m[i][j] != 0){
						return false;
					}
				}
			}
		}

		return true;
}

bool IsEqual(const matrix &m1, const matrix &m2){
		if ((m1.Rows() != m2.Rows()) || (m1.Cols() != m2.Cols()))
			return false;
```

Matrix

```
	for (int i = 0; i < m1.Rows(); i++){
		for (int j = 0; j < m1.Cols(); j++){
			if( !IsEqual( m1[i][j] , m2[i][j] ) ){
				return false;
			}
		}
	}
	return true;
}

bool IsNotEqual(const matrix &m1, const matrix &m2){
	return !IsEqual(m1, m2);
}

matrix matrix::operator*=(const matrix &m1){
	matrix tmp(rows, m1.cols);
	multiply(tmp, *this, m1);
	*this = tmp;
	return *this;
}

void multiply(matrix &result, const matrix &m1, const matrix &m2){
	assert(&result != &m1 && &result != &m2);
	assert(m1.Cols() == m2.Rows());
	assert(result.Rows() == m1.Rows());
	assert(result.Cols() == m2.Cols());

	double temp = 0.0;

	for (int i = 0; i < m1.Rows() ; i++){
		for (int j = 0; j < m1.Cols() ; j++){

			for (int u = 0; u < m1.Rows(); u++){
				temp = temp + (m1[i][u]*m2[u][j]);
			}

			result[i][j] = temp;
			temp = 0.0;
		}
	}
}

matrix operator+(const matrix &m1, const matrix &m2){
	assert(m1.Rows() == m2.Rows() && m1.Cols() == m2.Cols());

	matrix result(m1.Rows(), m1.Cols());

	for (int i = 0; i < m1.Rows(); i++){
		for (int j = 0; j < m1.Cols(); j++){
			result[i][j] = m1[i][j]+m2[i][j];
		}
	}
	return result;
}

matrix operator-(const matrix &m1, const matrix &m2){
	assert(m1.Rows() == m2.Rows() && m1.Cols() == m2.Cols());
```

Matrix

```
	matrix result(m1.Rows(), m1.Cols());

	for (int i = 0; i < m1.Rows(); i++){
		for (int j = 0; j < m1.Cols(); j++){
			result[i][j] = m1[i][j]-m2[i][j];
		}
	}
	return result;
}

matrix operator*(const matrix &m1, const matrix &m2){
	assert(m1.Cols() == m2.Rows());

	matrix result(m1.Rows(), m2.Cols());

	double temp = 0.0;

	for (int i = 0; i < m1.Rows() ; i++){
		for (int j = 0; j < m1.Cols() ; j++){

			for (int u = 0; u < m1.Rows(); u++){
				temp = temp + (m1[i][u]*m2[u][j]);
			}

			result[i][j] = temp;
			temp = 0.0;
		}
	}
	return result;
}

point operator*(const matrix &m, const point &p){
	assert((m.Rows() == 3 && m.Cols() == 3) || (m.Rows() == 4 && m.Cols() == 4));

	double xTemp = 0.0;
	double yTemp = 0.0;
	double zTemp = 0.0;
	double wTemp = 0.0;

	for (int i = 0; i < m.Rows() ; i++){
		for (int j = 0; j < m.Cols() ; j++){
			if(i == 0){
				xTemp = xTemp + m[i][j]*p[j];
			}else if(i == 1){
				yTemp = yTemp + m[i][j]*p[j];
			}else if(i == 2){
				zTemp = zTemp + m[i][j]*p[j];
			}else if(i == 3){
				wTemp = wTemp + m[i][j]*p[j];
			}
		}
	}

	return point(xTemp,yTemp,zTemp,wTemp);
}

vector operator*(const matrix &m, const vector &v){
```

```
	assert((m.Rows() == 3 && m.Cols() == 3) || (m.Rows() == 4 && m.Cols() == 4));

	double xTemp = 0.0;
	double yTemp = 0.0;
	double zTemp = 0.0;
	double wTemp = 0.0;

	for (int i = 0; i < m.Rows() ; i++){
		for (int j = 0; j < m.Cols() ; j++){
			if(i == 0){
				xTemp = xTemp + m[i][j]*v[j];
			}else if(i == 1){
				yTemp = yTemp + m[i][j]*v[j];
			}else if(i == 2){
				zTemp = zTemp + m[i][j]*v[j];
			}
		}
	}

	return vector(xTemp,yTemp,zTemp,wTemp);
}
```