Swinburne University of Technology

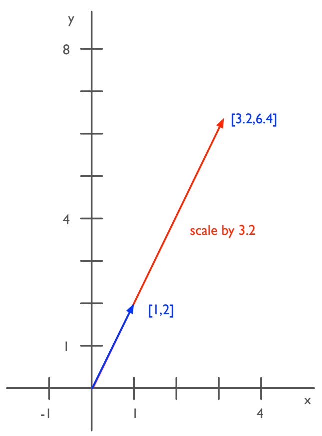
School of Science, Computing and Engineering Technologies

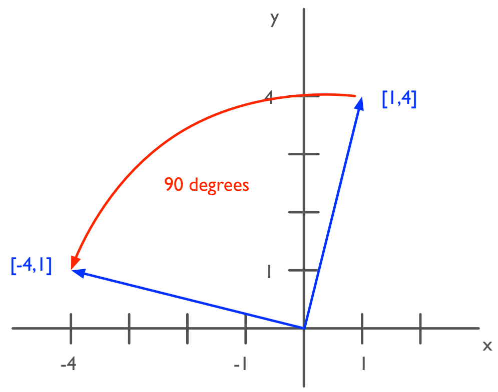
LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	3, Solution Design in C++
Lecturer:	Dr. Markus Lumpe

However difficult life may seem, there is always something you can do and succeed at.

Steven Hawking





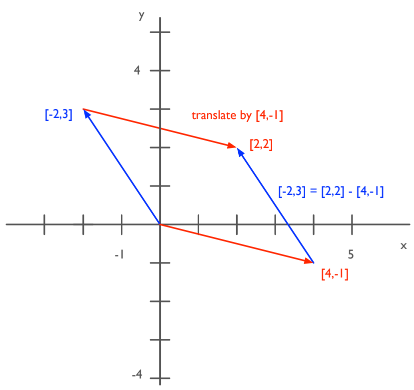


Figure 1: Scaling, rotation, and translation of vectors.

Solution Design in C++

Consider the solution of tutorial 2 in which we developed a small application to represent polygons whose vertices are defined as `Vector2D` objects.

The class Polygon allowed us to create new `Polygon` objects, read vertex data from a text file, compute the perimeter of a polygon, and create a scaled polygon from an existing one.

The latter operation is a linear transformation – a transformation from one vector space to another that preserves vector addition and scalar multiplication. It is for this reason that the shape of the sample polygon (e.g., T-Rex) did not change when we scaled it by a scalar.

Transformations of a set of vectors from one coordinate space to another are basic operations used frequently in geometry and computer graphics. Most notably, we use the linear transformations scaling, rotation, and translation to manipulate vectors in a given vector space (say, in the plane for Vector2D objects).

It turns out that these transformations can be conveniently expressed in matrix form, that is, these transformations can be denoted as a multiplication of a matrix with a vector. For scaling and rotation, a 2x2 matrix suffices to manipulate a vector in the plane (i.e., a `Vector2D` object). Unfortunately, translation of a vector in the plane via multiplication with a 2x2 matrix cannot be expressed. For example, a translation by zero would yield a zero vector for all vectors when multiplication is used.

The goal of this tutorial is to define the necessary infrastructure to scale, rotate, and translate 2D vectors. This requires two new data types:

- a 3D vector that provides the homogeneous coordinates for a 2D vector, and
- a 3x3 matrix that encodes the desired transformation.

We use multiplication of a 3x3 matrix with a 3D vector to perform the desired transformation.

Vector3D

In order to uniformly represent the desired vector transformations of a vector in the plane, we need to convert every `2DVector` object into a `Vector3D` object. Class `Vector3D` is defined as shown below:

```
#include "Vector2D.h"

class Vector3D
{
private:

  Vector2D fBaseVector;
  float fW;

public:

  Vector3D( float aX = 1.0f, float aY = 0.0f, float aW = 1.0f ) noexcept;
  Vector3D( const Vector2D& aVector ) noexcept;

  float x() const noexcept { return fBaseVector.x(); }
  float y() const noexcept { return fBaseVector.y(); }
  float w() const noexcept { return fW; }

  float operator[]( size_t aIndex ) const;

  explicit operator Vector2D() const noexcept;

  Vector3D operator*( const float aScalar ) const noexcept;
  Vector3D operator+( const Vector3D& aOther ) const noexcept;
  float dot( const Vector3D& aOther ) const noexcept;

  friend std::ostream& operator<<(std::ostream& aOStream, const Vector3D& aVector);
};
```

Class `Vector3D` defines an object adapter for `Vector2D` objects. It wraps `Vector2D` objects and extends some of the basic vector operations to homogeneous coordinates. Objects of class `Vector3D` define an additional coordinate, `fW`, which is equal to 1. This coordinate is used to project a 3D vector back to 2D (see additional tutorial notes).

The member functions of `Vector3D` are defined as follows:

- The overloaded constructors initialize all member variables with sensible values. In any case, the component `fW` has to be set to 1.

  The constructor `Vector3D( const Vector2D& aVector )` serves as implicit type conversion. It "boxes" a `2DVector` object into a `3DVector` object.

- The component accessors are defined in the usual way. The functions `x()` and `y()` forward to request to the wrapped `Vector2D` object `fBaseVector`.
- The class Vector3D defines an index for the components. This indexer maps to the corresponding vector coordinates. The implementation exploits the standard member variable layout in C++. In case of Vector3D, the member variables can be accessed as if they were defined as an array of `float` values. We just need to convince the compiler that the **this**-pointer (i.e., the pointer to a `Vector3D` object) refers to an array of **float**. We can use C++'s `reinterpret_cast` to achieve this:

  ```
  float Vector3D::operator[]( size_t aIndex ) const
  {
    assert( aIndex < 3 );

    return *(reinterpret_cast<const float*>(this) + aIndex);
  }
  ```

- The operator `Vector2D()` is an explicit type conversion operator. It is called when we type cast a `Vector3D` object to a `Vector2D` object. The operator has to return a `Vector2D` object whose x and y components has been divided by `fW`.
- Scalar multiplication, vector addition, and the dot product extend the 2D operations to 3D ones. That is, we need to account for the `fW` component.
- The output operator has to send the `Vector3D` object to the output stream. To do so, we cast the object `aVector`, passed as parameter, to `Vector2D` using a `static_cast` and send to result to the output stream. They type cast invokes the type conversion operator `Vector2D()` defined in `Vector3D`.

The test code in `main()` can be used to verify the implementation of `Vector3D`. In particular, the sequence

```
Vector2D a( 1.0f, 2.0f );
Vector2D b( 1.0f, 4.0f );
Vector2D c( -2.0f, 3.0f );
Vector2D d( 0.0f, 0.0f );

std::cout << "Test vector implementation: " << std::endl;
std::cout << "Vector a = " << a << std::endl;
std::cout << "Vector b = " << b << std::endl;
std::cout << "Vector c = " << c << std::endl;
std::cout << "Vector d = " << d << std::endl;

Vector3D a3( a );
Vector3D b3( b );
Vector3D c3( c );
Vector3D d3( d );

std::cout << "Vector a3 = " << a << std::endl;
std::cout << "Vector b3 = " << b << std::endl;
std::cout << "Vector c3 = " << c << std::endl;
std::cout << "Vector d3 = " << d << std::endl;

std::cout << "Test homogeneous vectors:" << std::endl;
std::cout << "Vector " << a3 << " * 3.0 = " << a3 * 3.0f << std::endl;
std::cout << "Vector " << a3 << " + " << b3 << " = " << a3 + b3 << std::endl;
std::cout << "Vector " << a3 << " . " << b3 << " = " << a3.dot( b3 ) << std::endl;
std::cout << "Vector " << a3 << "[0] = " << a3[0] << " <=> " << a3 << ".x() = "<<a3.x()<<std::endl;
std::cout << "Vector " << a3 << "[1] = " << a3[1] << " <=> " << a3 << ".y() = "<<a3.y()<<std::endl;
std::cout << "Vector " << a3 << "[2] = " << a3[2] << " <=> " << a3 << ".w() = "<<a3.w()<<std::endl;
```

should generate the following output:

```
Test vector implementation:
Vector a = [1,2]
Vector b = [1,4]
Vector c = [-2,3]
Vector d = [0,0]
Vector a3 = [1,2]
Vector b3 = [1,4]
Vector c3 = [-2,3]
Vector d3 = [0,0]
Test homogeneous vectors:
Vector [1,2] * 3.0 = [1,2]
Vector [1,2] + [1,4] = [1,3]
Vector [1,2] . [1,4] = 10
Vector [1,2][0] = 1 <=> [1,2].x() = 1
Vector [1,2][1] = 2 <=> [1,2].y() = 2
Vector [1,2][2] = 1 <=> [1,2].w() = 1
```

Matrix3x3

Class `Matrix3x3` defines the basic infrastructure to represent vector transformations. It is defined as follows:

```
#include "Vector3D.h"

class Matrix3x3
{
private:

  Vector3D fRows[3];

public:

  Matrix3x3() noexcept;
  Matrix3x3( const Vector3D& aRow1,
             const Vector3D& aRow2,
             const Vector3D& aRow3 ) noexcept;

  Matrix3x3 operator*( const float aScalar ) const noexcept;
  Matrix3x3 operator+( const Matrix3x3& aOther ) const noexcept;

  Vector3D operator*( const Vector3D& aVector ) const noexcept;

  static Matrix3x3 scale( const float aX = 1.0f, const float aY = 1.0f );
  static Matrix3x3 translate( const float aX = 0.0f, const float aY = 0.0f );
  static Matrix3x3 rotate( const float aAngleInDegree = 0.0f );

  const Vector3D row( size_t aRowIndex ) const;
  const Vector3D column( size_t aColumnIndex ) const;
};
```

The components of `Matrix3x3` are 3D row vectors. This allows easy access to rows of a 3x3 matrix. This representation is called row-major order. Class Matrix3x3 defines the basic matrix operations scalar multiplication, matrix addition, and multiplication with a 3D vector. The latter allows us to perform desired vector transformations. In addition, Matrix3x3 defines three static methods that return the corresponding transformation matrix.

The member functions of `Matrix3x3` are defined as follows:

- The overloaded constructors initialize all member variables with sensible values. The default constructor has to yield the *identity matrix*:

  $$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  To construct this matrix, we need to build three vectors, [1,0,0], [0,1,0], and [0,0,1], and assign them to the corresponding entry in the row vector array of the matrix object.
  The second constructor receives three row vectors and copies them into the row vector array of the matrix object.
- Scalar multiplication and matrix addition are defined in the usual way. The implementation maps the operations to the corresponding vector operations. The result is a new 3x3 matrix initialized with three new row vectors resulting from scalar multiplication and vector addition, respectively.
- The multiplication with a 3D vector yields a new 3D vector whose components are the dot product of each row vector with the argument vector.

- The static functions `scale()`, `translate()`, and `rotate()` construct the corresponding transformations matrices:
  - Scale:

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  - Translate:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix}$$

  - Rotate:

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The functions `row()` and `column()` return the corresponding row and column vectors. The function `row()` just returns a copy of the corresponding element from the row array. The function `column()` has to "slice" through the row array vertically. We can use the Vector3D indexer for this purpose. Please note, `row()` and `column()` return read-only copies of matrix components. Changing a component in a row or column vector has no effect on the underlying matrix.

The test code in `main()` can be used to verify the implementation of `Matrix3x3`. In particular, the sequence

```
std::cout << "Test 3x3 matrix:" << std::endl;

Matrix3x3 ma( Vector3D( 1.0f, 1.0f ), Vector3D( 1.0f, 1.0f ), Vector3D( 1.0f, 1.0f ) );

std::cout << "ma: row 1 = " << ma.row( 0 ) << std::endl;
std::cout << "ma: row 2 = " << ma.row( 1 ) << std::endl;
std::cout << "ma: row 3 = " << ma.row( 2 ) << std::endl;

Matrix3x3 mb = ma * 2.0f;

std::cout << "mb: row 1 = " << mb.row( 0 ) << std::endl;
std::cout << "mb: row 2 = " << mb.row( 1 ) << std::endl;
std::cout << "mb: row 3 = " << mb.row( 2 ) << std::endl;

Matrix3x3 mc = mb + ma;

std::cout << "mc: row 1 = " << mc.row( 0 ) << std::endl;
std::cout << "mc: row 2 = " << mc.row( 1 ) << std::endl;
std::cout << "mc: row 3 = " << mc.row( 2 ) << std::endl;

Matrix3x3 lScale = Matrix3x3::scale( 3.2f, 3.2f );
Matrix3x3 lRotate = Matrix3x3::rotate( 90.0f );
Matrix3x3 lTranslate = Matrix3x3::translate( 4.0f, -1.0f );

std::cout << "Scale " << a3 << " by " << 3.2f << " = " << lScale * a3 << std::endl;
std::cout << "Rotate " << b3 << " by " << 90.0f << " degrees = " << lRotate * b3 << std::endl;
std::cout << "Translate " << c3 << " by " << lTranslate.column( 2 ) << " = "
          << lTranslate * c3 << std::endl;
std::cout << "Translate " << d3 << " by " << lTranslate.column( 2 ) << " = "
          << lTranslate * d3 << std::endl;
```

should generate the following output:

```
Test 3x3 matrix:
ma: row 1 = [1,1]
ma: row 2 = [1,1]
ma: row 3 = [1,1]
mb: row 1 = [1,1]
mb: row 2 = [1,1]
mb: row 3 = [1,1]
mc: row 1 = [1,1]
mc: row 2 = [1,1]
mc: row 3 = [1,1]
Scale [1,2] by 3.2 = [3.2,6.4]
Rotate [1,4] by 90 degrees = [-4,1]
Translate [-2,3] by [4,-1] = [2,2]
Translate [0,0] by [4,-1] = [4,-1]
```

The output regarding the matrices `ma`, `mb`, and `mc` might be confusing, but it is correct. The row vectors are printed as `Vector2D` objects and, hence, they are scaled by the w component. The scalar product and matrix addition change this component. In case of matrix `mb` it becomes 2, and in case of matrix `mc` it is 3. This highlights the geometrical interpretation of the w component: any scalar multiple of a 3D vector represents the same point in two-dimensional space. Use the debugger to verify the values of matrices `mb` and `mc`.

This task requires approximately 130-150 lines of code. Most function require 1 or 3 lines of code.