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

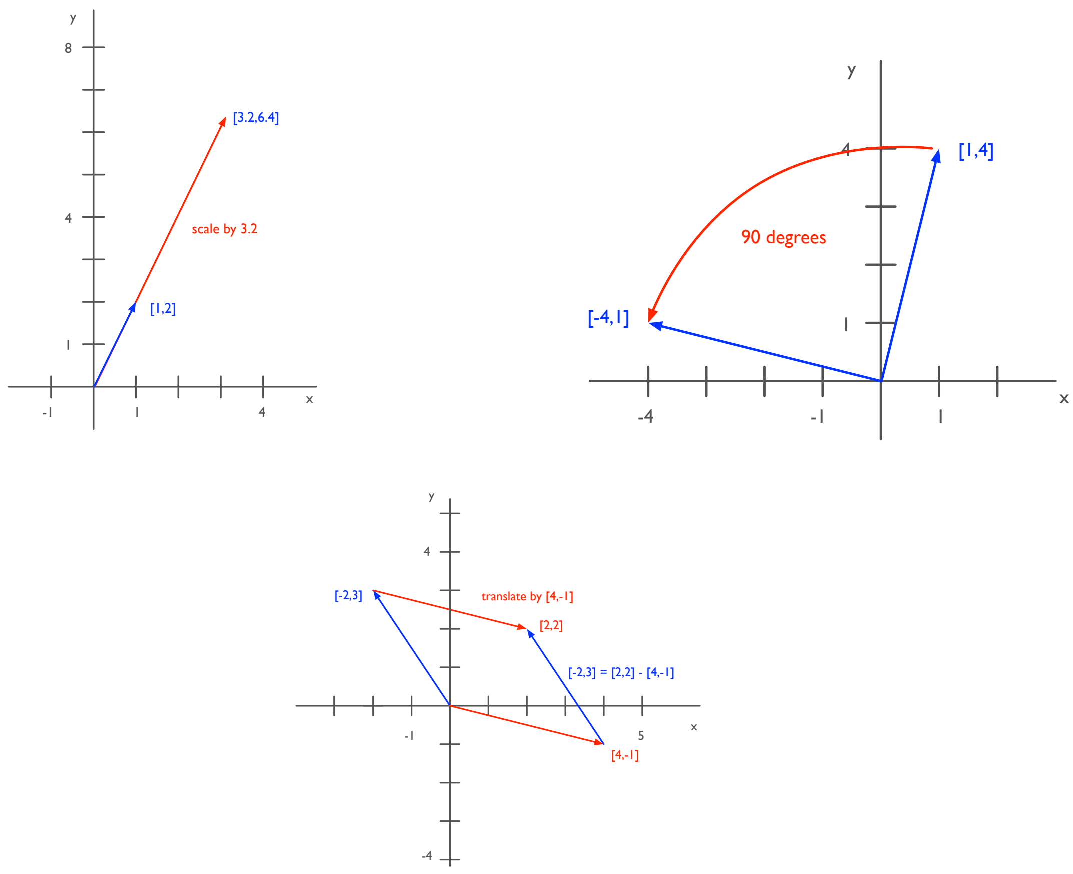

Figure 1: Scaling, rotation, and translation of vectors.

Solution Design in C++

Recall Tutorial 2. In this tutorial, we studied the class `Polygon` which defines a data type for creating 2D polygons using `Vector2D` objects in the plane. The class `Polygon` allows us to construct new `Polygon` objects, read vertex data from a text file, compute the perimeter of a polygon, and scale an existing one.

The latter operation is a linear transformation from one vector space to another that preserves vector addition and scalar multiplication. A polygon shape (e.g., T-Rex) does not change when scaled by a scalar.

Vector transformations from one coordinate space to another are fundamental operations frequently used in geometry and computer graphics. Most notably, we use the linear transformations scaling, rotation, and translation to manipulate vectors in a given vector space (for example, the plane for `Vector2D` objects) to change a vector's magnitude, orientation, and location.

These transformations can be conveniently expressed in matrix form and be denoted as a multiplication of a matrix with a vector. To scale or rotate a vector in the plane, a 2x2 matrix suffices. Unfortunately, translating a vector in the plane using multiplication with a 2x2 matrix cannot be expressed. For instance, a translation by zero would yield a zero vector for all vectors when multiplication is applied.

To solve this problem, we must move from two dimensions to three. Adding an extra dimension allows us to represent scaling, rotation, and translation in a single framework. 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 transformations.

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

Vector3D

To uniformly represent the desired vector transformations of a vector in the plane, we need to convert every `Vector2D` object into a `Vector3D` object. A suitable specification for class `Vector3D` is 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 noexcept;

  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 basic vector operations to homogeneous coordinates. Objects of class `Vector3D` define an additional coordinate, `fW`, which is equal to 1. The three coordinates define a line in 3D space and a 3D vector is projected into 2D when `fW` = 1 (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 )` performs an implicit type conversion. It "boxes" a `Vector2D` object into a `Vector3D` object.
- The component accessors are defined in the usual way—functions `x()` and `y()` forward requests to the wrapped `Vector2D` object `fBaseVector`.
- The class `Vector3D` defines an index operator to map to the corresponding vector coordinates.

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

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

  The implementation exploits the standard member variable layout in C++. In the case of `Vector3D`, the member variables can be accessed as if they were defined as an array of **`float`** values. We convince the compiler that the **this**-pointer (i.e., the pointer to a `Vector3D` object) is a pointer to **float** values. We can use C++'s `reinterpret_cast` to achieve this.

- The member function `Vector2D()` is an explicit type conversion operator. It is called when we typecast a `Vector3D` object to a `Vector2D` object. The operator has to return a `Vector2D` object whose x and y components have been scaled 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 typecast the parameter object `aVector`, passed as a parameter, to `Vector2D` using a `static_cast` and send the result to the output stream. The type cast invokes the type conversion operator `Vector2D()` defined in `Vector3D`.

Use the test code in `main()` 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 perform 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 getS( const float aX = 1.0f, const float aY = 1.0f ) noexcept;
  static Matrix3x3 getT( const float aX = 0.0f, const float aY = 0.0f ) noexcept;
  static Matrix3x3 getR( const float aAngleInDegree = 0.0f ) noexcept;

  const Vector3D& row( size_t aRowIndex ) const noexcept;
  const Vector3D column( size_t aColumnIndex ) const noexcept;

  const Vector3D& operator[]( size_t aRowIndex ) const noexcept;
};
```

The components of class `Matrix3x3` are 3D row vectors, a representation 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, class `Matrix3x3` defines three static methods that return the corresponding transformation matrices.

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 create three vectors, [1,0,0], [0,1,0], and [0,0,1], and assign them to the corresponding entries in the row vector array of the matrix object. An array member initializer is used for this purpose.
  The second constructor accepts three row vectors and copies them into the row vector array of the matrix object. Again, an array member initializer is used for this purpose.
- Scalar multiplication and matrix addition are defined in the usual way. The implementation maps the operations to the corresponding vector operations. These operations produce a new 3x3 matrix initialized with the transformed row vectors of the input matrix (i.e., **this**-object).
- 3x3 matrix multiplication with a 3D vector yields a new 3D vector whose components are the dot product of each row vector and the argument vector.

- The static functions `getS()`, `getT()`, and `getR()` 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 as 3D vectors. The function `row()` returns a reference to 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 that `row()` returns a read-only reference to the corresponding row vector, whereas `column()` returns read-only copies of matrix components. Modifying a component value in a column vector does not affect the underlying matrix.
  We use a constant reference to row vectors as the result type of function `row()`. We avoid an undesired copy of matrix elements. The implementation of the index **`operator[]`** relies on this feature.
- The index **`operator[]`** returns a reference to the corresponding row in the matrix. It uses the function `row()` for this purpose.

To facilitate the use and implementation of some matrix operations, explicitly define the following constant reference within the method body:

```
const Matrix3x3& M = *this;
```

The reference `M` serves as a shorthand for the expression **`*this`** which refers to the current object. For instance, instead of having to write `(*this)[0]` to obtain the first row of a 3x3 matrix, you can use the expression `M[0]`. Your code becomes more readable. There should be no runtime overhead as the reference to the current object **`this`** is stored in the RCX register.

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[0] = " << MA[0] << std::endl;
std::cout << "MA[1] = " << MA[1] << std::endl;
std::cout << "MA[2] = " << MA[2] << std::endl;

Matrix3x3 MB = MA * 2.0f;

std::cout << "MB[0] = " << MB[0] << std::endl;
std::cout << "MB[1] = " << MB[1] << std::endl;
std::cout << "MB[2] = " << MB[2] << std::endl;

Matrix3x3 MC = MB + MA;

std::cout << "MC[0] = " << MC[0] << std::endl;
std::cout << "MC[1] = " << MC[1] << std::endl;
std::cout << "MC[2] = " << MC[2] << std::endl;

Matrix3x3 lS = Matrix3x3::getS( 3.2f, 3.2f );
Matrix3x3 lR = Matrix3x3::getR( 90.0f );
Matrix3x3 lT = Matrix3x3::getT( 4.0f, -1.0f );

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

should generate the following output:

```
Test 3x3 matrix:
MA[0] = [1,1]
MA[1] = [1,1]
MA[2] = [1,1]
MB[0] = [1,1]
MB[1] = [1,1]
MB[2] = [1,1]
MC[0] = [1,1]
MC[1] = [1,1]
MC[2] = [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 generated for 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 matrix `MB`, it becomes 2, and in 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 and most functions require 1 or 3 lines.