

TUTORIAL 3: VECTOR TRANSFORMATIONS AND MATRICES

Overview

- Matrices
- Vector2D transformations
- Vector3D and 3x3 Matrices
- Object Layout

References

- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)
- Eric Lengyel: Mathematics for 3D Game Programming and Computer Graphics, Course Technology (2012)

NXM MATRIX

- An $n \times m$ matrix **M** is an array of numbers having n rows and m columns. If $n=m$, then we say that the matrix **M** is a square matrix.

$$\mathbf{M}_{3 \times 4} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \end{bmatrix}$$

$$\text{row}(\mathbf{M}_{3 \times 4}, 2) = \begin{bmatrix} M_{21} & M_{22} & M_{23} & M_{24} \end{bmatrix} \quad \text{column}(\mathbf{M}_{3 \times 4}, 3) = \begin{bmatrix} M_{13} \\ M_{23} \\ M_{33} \end{bmatrix}$$

MATRIX SCALAR MULTIPLICATION

- Given a scalar α and an $n \times m$ matrix **M**, the product $\alpha\mathbf{M} = \mathbf{M}\alpha$ is given by

$$\alpha\mathbf{M} = \mathbf{M}\alpha = \begin{bmatrix} \alpha M_{11} & \alpha M_{12} & \cdots & \alpha M_{1m} \\ \alpha M_{21} & \alpha M_{22} & \cdots & \alpha M_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha M_{n1} & \alpha M_{n2} & \cdots & \alpha M_{nm} \end{bmatrix}$$

MATRIX ADDITION

- Matrices add element wise. Given two $n \times m$ matrices **F** and **G**, the sum **F**+**G** is given by

$$\mathbf{F} + \mathbf{G} = \begin{bmatrix} F_{11} + G_{11} & F_{12} + G_{12} & \cdots & F_{1m} + G_{1m} \\ F_{21} + G_{21} & F_{22} + G_{22} & \cdots & F_{2m} + G_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ F_{n1} + G_{n1} & F_{n2} + G_{n2} & \cdots & F_{nm} + G_{nm} \end{bmatrix}$$

MATRIX MULTIPLICATION

- Two matrices **F** and **G** can be multiplied, provided that the number of columns in **F** is equal to the number of rows in **G**. If **F** is $n \times m$ matrix and **G** is an $m \times p$ matrix, then the product **FG** is an $n \times p$ matrix whose (i, j) entry is given by

$$(\mathbf{FG})_{ij} = \sum_{k=1}^m F_{ik} G_{kj}$$

- Actually, an (i, j) entry in **FG** is the dot product of the vectors $\text{row}(\mathbf{F}, i)$ and $\text{column}(\mathbf{G}, j)$.
- An n -dimensional vector can be thought of as $n \times 1$ matrix. Hence, multiplying an $m \times n$ matrix **M** with an n -dimensional vector **V**, written **MV**, yields an m -dimensional vector **V'**.

SCALING A 2D VECTOR

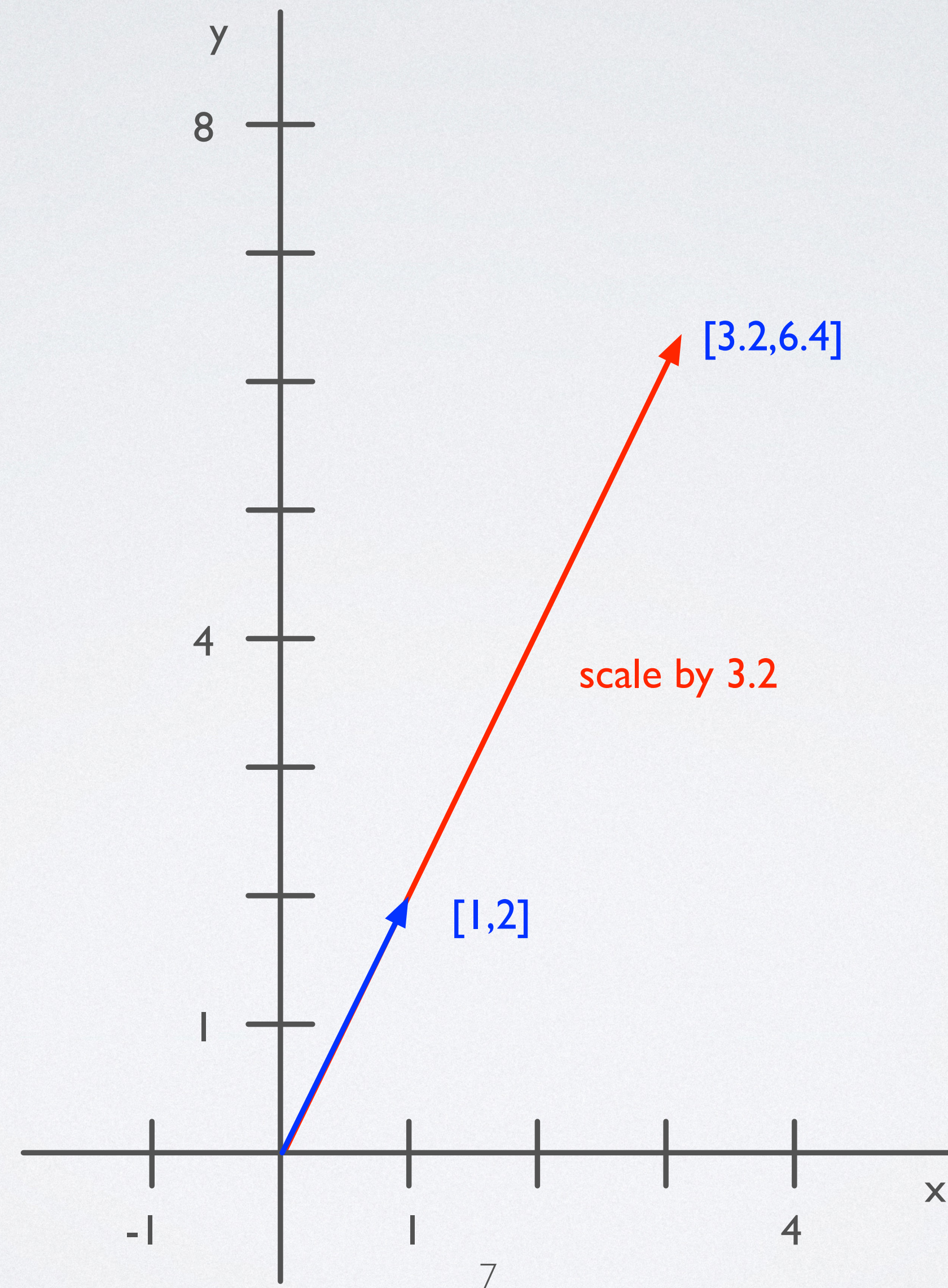
- To scale a vector \mathbf{V} by a factor of α , we simply calculate $\mathbf{V}' = \alpha\mathbf{V}$.
- This operation can also be expressed as the (2-dimensional) matrix product

$$\mathbf{V}' = \begin{bmatrix} V'_x \\ V'_y \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \alpha \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix}$$

- For example, the vector $[1.0 \ 2.0]$ scaled by 3.2 yields a vector $[3.2, 6.4]$:

$$\begin{bmatrix} 3.2 \\ 6.4 \end{bmatrix} = \begin{bmatrix} 3.2 & 0 \\ 0 & 3.2 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix}$$

SCALING BY 3.2



ROTATION OF A 2D VECTOR

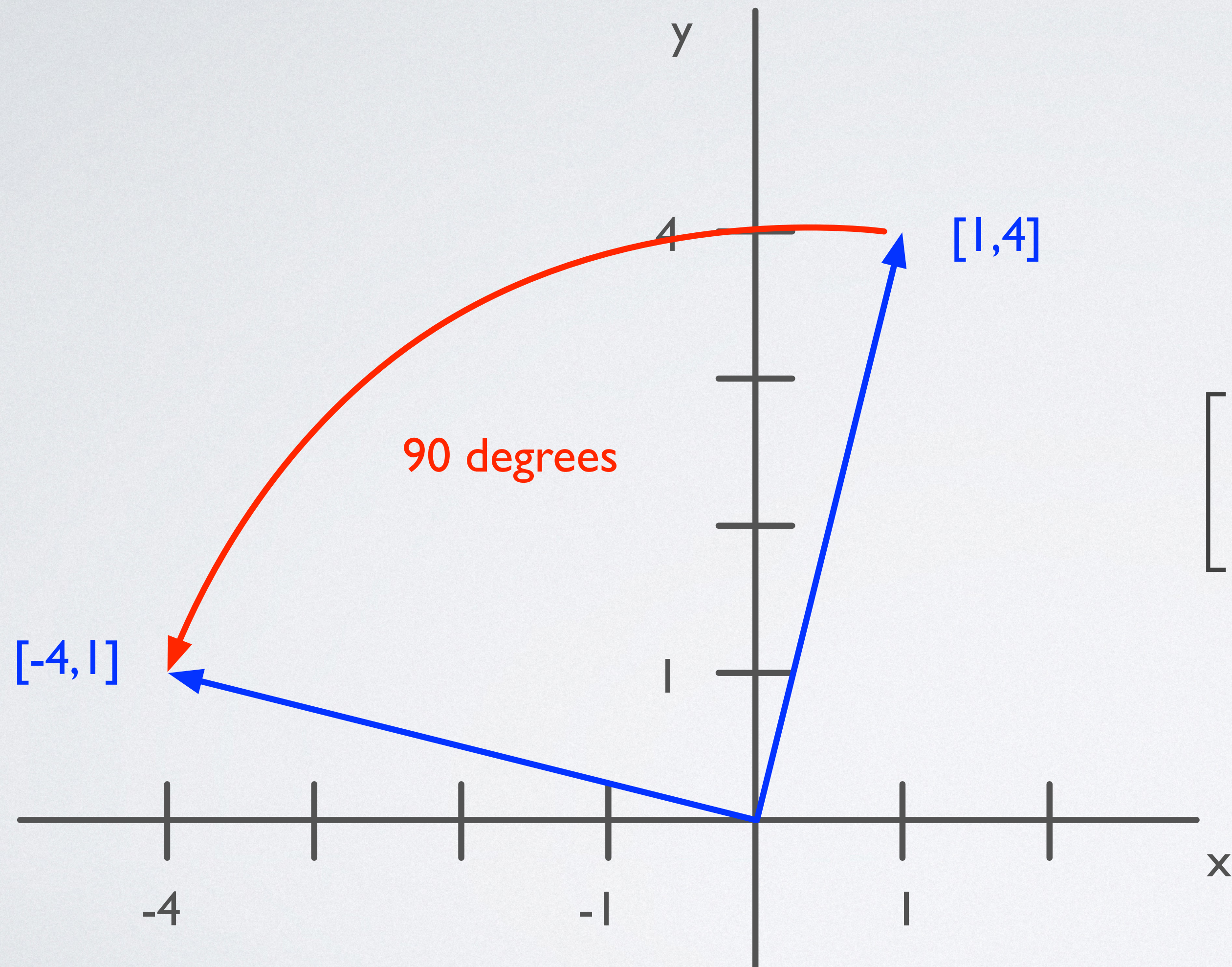
- In 2 dimensions, we can rotate a point in the plane by the following matrix

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

- For example, the vector $[1.0 \ 4.0]$ rotated by 90 degrees yields a vector $[-4.0, 1.0]$:

$$\begin{bmatrix} -4.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 4.0 \end{bmatrix}$$

ROTATION BY 90 DEGREES



$$\begin{bmatrix} -4.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 4.0 \end{bmatrix}$$

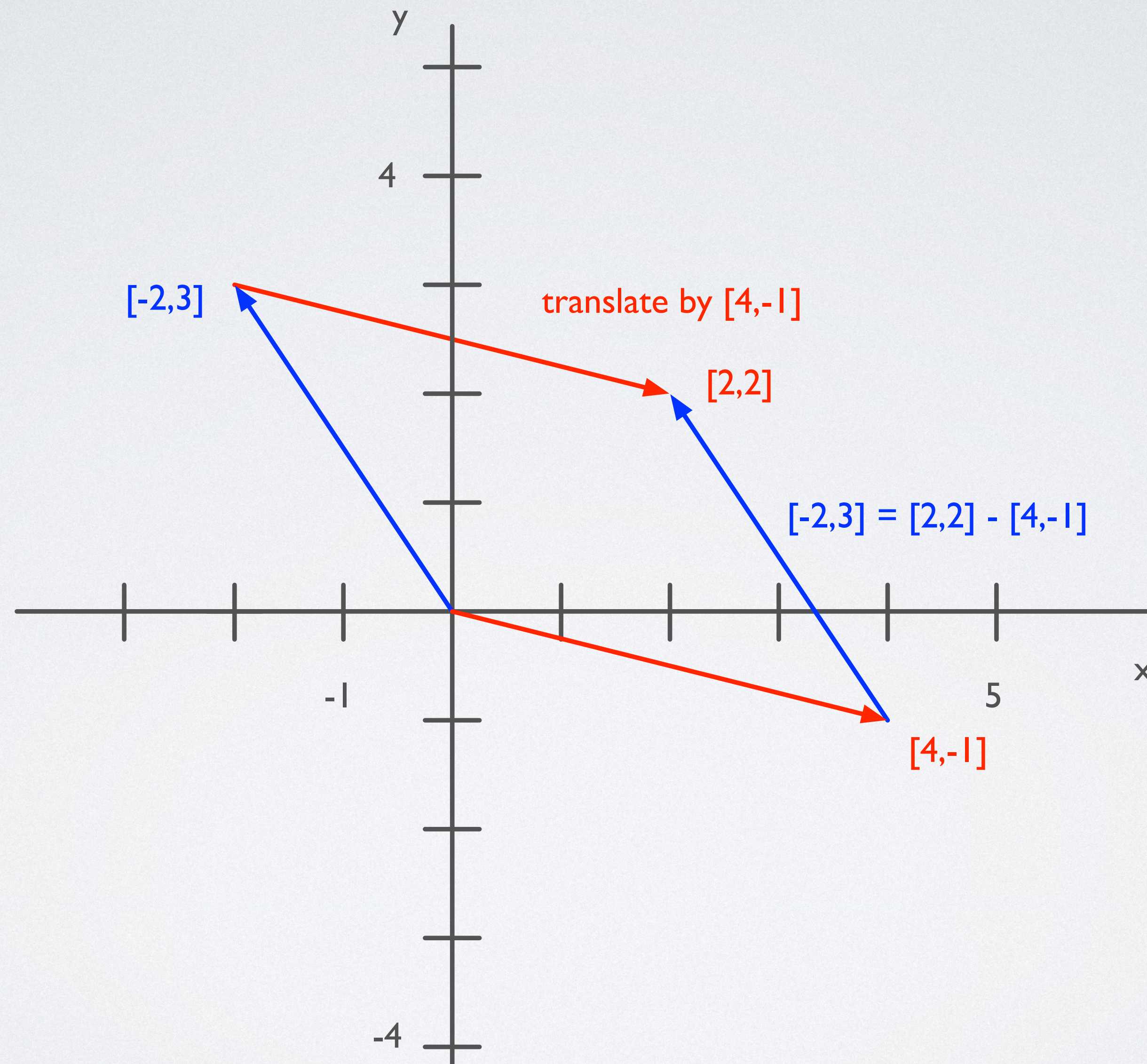
TRANSLATION OF A 2D VECTOR

- Translation of a vector can be achieved by simply adding an offset to it:

$$\mathbf{V}' = \begin{bmatrix} V'_x \\ V'_y \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \end{bmatrix} + \begin{bmatrix} \Delta_x \\ \Delta_y \end{bmatrix}$$

- Translation does not affect the orientation or scale of the axes.
- Unfortunately, this operation cannot be expressed in terms of a multiplication by a 2x2 matrix. We are left with matrix addition, which is computationally undesirable.

TRANSLATION BY $[4, -1]$



HOMOGENEOUS COORDINATES

- There exists a compact and elegant way to represent all transformations, including translations, within a single mathematical entity.
- We extend 2D vectors to 3D homogeneous coordinates (i.e, 3D vectors) and uses 3x3 matrices to transform them:

$$\mathbf{V}_3 = \begin{bmatrix} V_x \\ V_y \\ 1 \end{bmatrix}$$

$$\mathbf{F} = \left[\begin{array}{cc|c} \mathbf{M} & \mathbf{T} \\ \hline \mathbf{0} & 1 \end{array} \right] = \left[\begin{array}{cc|c} M_{11} & M_{12} & T_x \\ M_{21} & M_{22} & T_y \\ \hline 0 & 0 & 1 \end{array} \right]$$

SCALE, ROTATION, AND TRANSLATION MATRIX

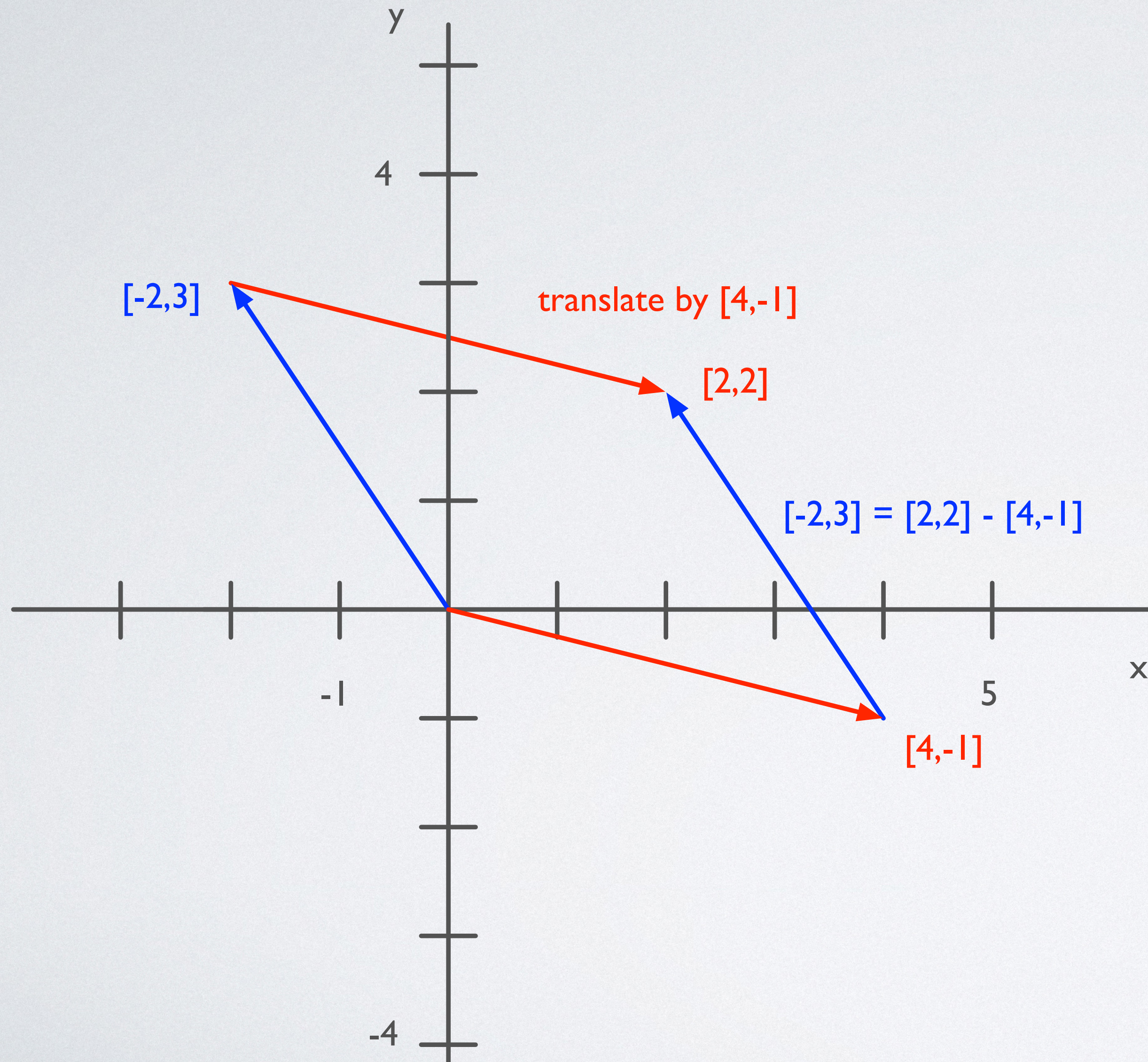
$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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

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

Note: In the translation matrix \mathbf{T} , the 2x2 matrix \mathbf{M} is the identity matrix. That is, under translation, scale and rotation remain unchanged.

TRANSLATION BY $[4, -1]$



Translate $[-2, 3]$:

$$\begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

Translate origin:

$$\begin{bmatrix} 4 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

TO AND FROM HOMOGENEOUS COORDINATES

In order to obtain 3-dimensional homogeneous coordinates, we add a w-coordinate with the value 1:

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} \Rightarrow \begin{bmatrix} V_x \\ V_y \\ V_w \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ 1 \end{bmatrix}$$

To return to 2D, we divide all coordinates by V_w and drop the w-coordinate:

$$\begin{bmatrix} V'_x/V_w \\ V'_y/V_w \\ V_w/V_w \end{bmatrix} = \begin{bmatrix} V_x \\ V_y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} V_x \\ V_y \end{bmatrix}$$


```
#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

// implicit type conversion from Vector2D to Vector3D

// index-based coordinate access

// explicit conversion operator to Vector2D

// 3D scalar multiplication

// 3D vector addition

// 3D dot product


```
#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;
```

```
// multiplication with scalar
```

```
    Matrix3x3 operator+( const Matrix3x3& aOther ) const noexcept;
```

```
// matrix addition
```

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

```
// multiplication with a vector
```

```
    static Matrix3x3 scale( const float aX = 1.0f, const float aY = 1.0f );
```

```
// build scaling matrix
```

```
    static Matrix3x3 translate( const float aX = 0.0f, const float aY = 0.0f );
```

```
// build translation matrix
```

```
    static Matrix3x3 rotate( const float aAngleInDegree = 0.0f );
```

```
// build rotation matrix
```

```
    const Vector3D row( size_t aRowIndex ) const;
```

```
// return read-only row vector
```

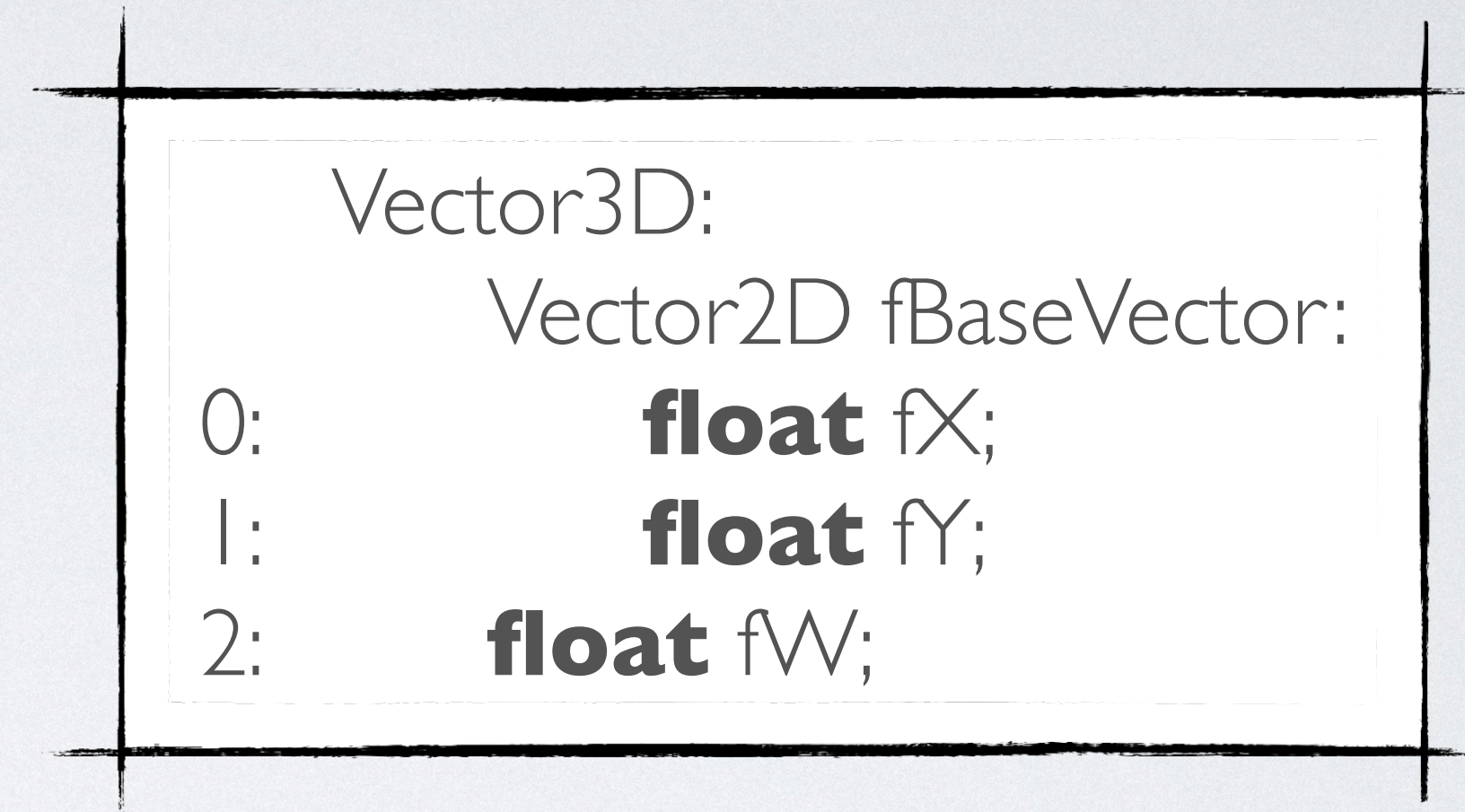
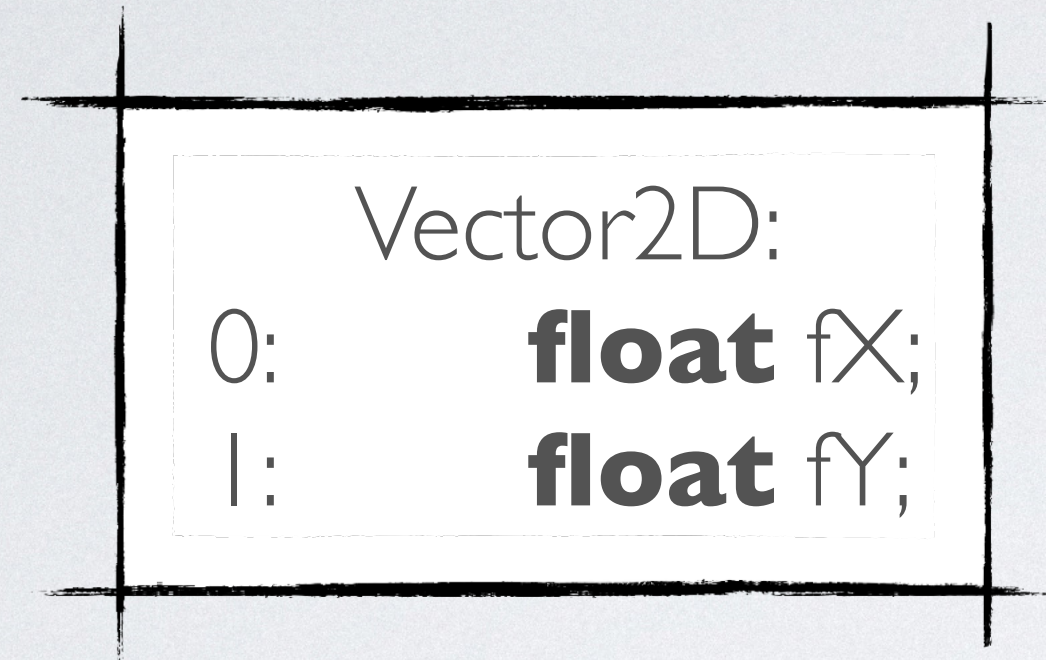
```
    const Vector3D column( size_t aColumnIndex ) const;
```

```
// return read-only column vector
```

```
};
```

CLASS MATRIX3X3

OBJECT LAYOUT VECTOR2D & VECTOR3D



- Typically, the memory layout of classes in C++ is left to the implementation. However, the layout usually follows some standard principles:
 - Non-static member variables with the same access level are placed adjacent to each other with the layout preserving the order of the declaration of member variables. Technically, the C++ object layout yields a record structure in which the fields are the member variables.
 - If there are no virtual members, the object record just contains the member variables. In case of `Vector2D` and `Vector3D`, the member variables `fX`, `fY`, and `fW` can be viewed as elements of a packed array of **float** values.