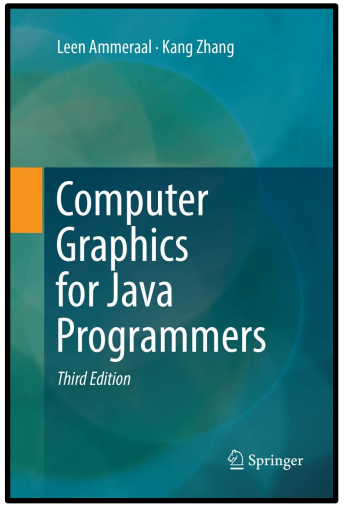
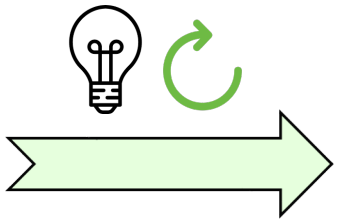
Drawing Heighway's Dragon Part 2
Recursive Function Simplification
From 2^n Recursive Invocations
To n Tail-Recursive Invocations Exploiting Self-Similarity

Scala





$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\,\mathbf{R}$$

$$\mathbf{R} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

slides by  @philip_schwarz FP Illuminated https://fpilluminated.org/

In part 1 we looked at a Scala program that draws Heighway's Dragon.

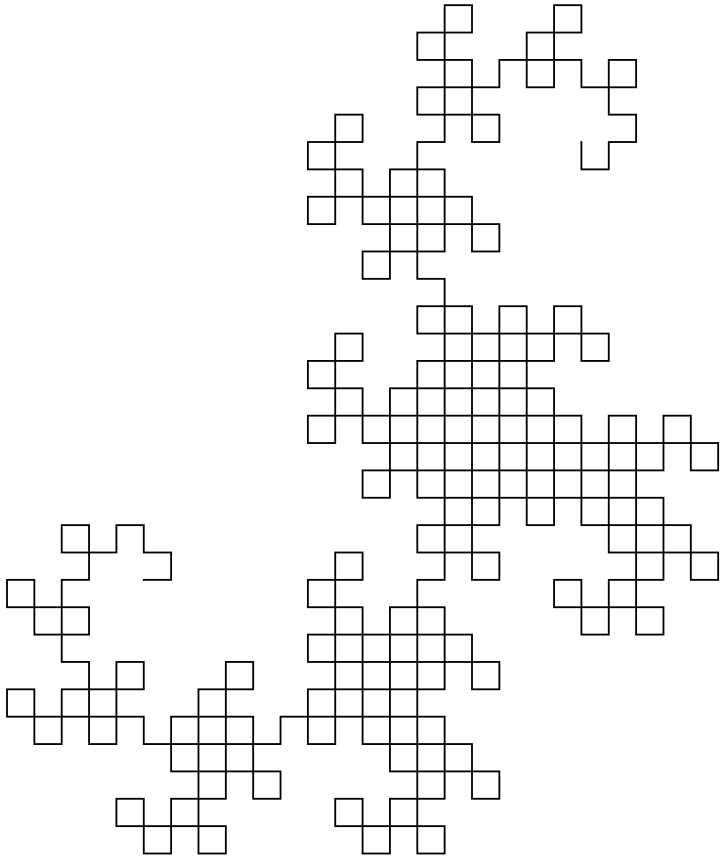
At the heart of the program is the following recursive function used to grow a dragon path, which is a sequence of points:

```
extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)
```

Once the path has been grown to the desired age, drawing the dragon amounts to drawing lines (of the given length) connecting consecutive pairs of the path's points.

On the right we see a drawing of a dragon aged 9.

Thanks to its use of recursion, the function is very simple, yet it is far from obvious why its logic is able to compute a path that results in the aesthetically pleasing dragon pattern.



The first thing we want to do in **part 2** is see if we can exploit the **self-similarity** of **Heighway's Dragon** to **simplify** the **grow function**. While it is already **very simple**, we want to **improve it further**, so that it becomes **simple** to **understand** why it is **able** to **accomplish** its **task**.

The way our program currently draws a **dragon** aged N is by first computing a **dragon path** consisting of a **sequence** of $2^N + 1$ **points** p_1, p_2 , …, p_{2^N+1}, and then drawing **lines** connecting each **point** p_i with next **point** p_{i+1}.

Let us focus on how the **program** computes the **path** of a **dragon** aged N, not in terms of the **exact steps** taken by the **program**, but in terms of the following **high level path-growing process**:

1. start with a **degenerate path** consisting of a **single point** (the **starting point**)
2. **grow** the **path** by computing a **new point**, and adding the latter to the **front** of the **path**
3. repeat the previous step until the **path** consists of $2^N + 1$ **points**.

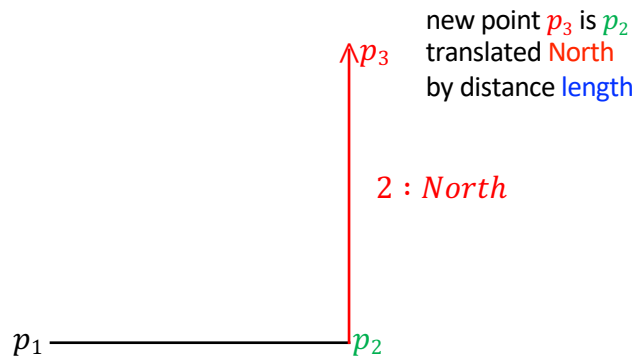
The next ten slides **visualise** the above **process** for a **dragon aged 4**.

To make things easier to understand, rather than just **visualising** the addition of newly computed **points**, the slides also show **lines** connecting the **points**, as if **lines** get drawn at the same time as **points** are computed.

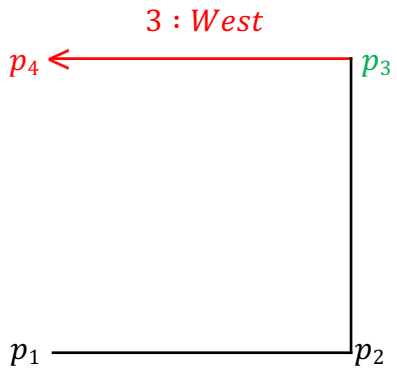
Initially the path consists simply of starting point $\boldsymbol{p_1}$.

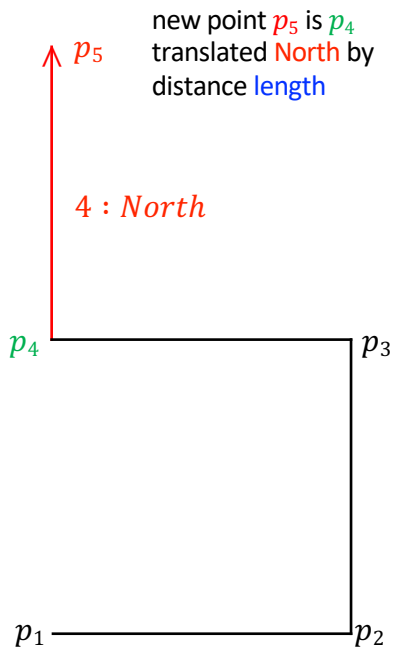
p_1

new point p_2 is p_1 translated East by distance length

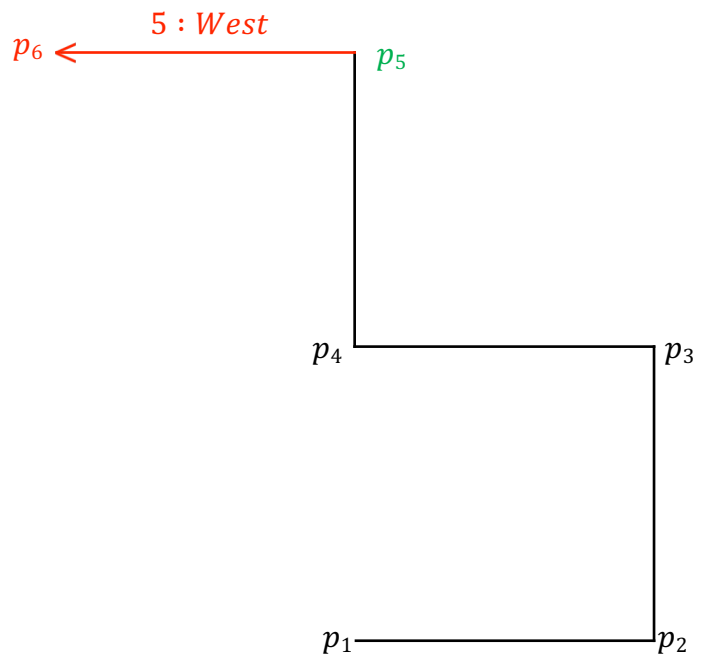

new point p_3 is p_2
translated North
by distance length
p_3
$2 : North$
p_1
p_2

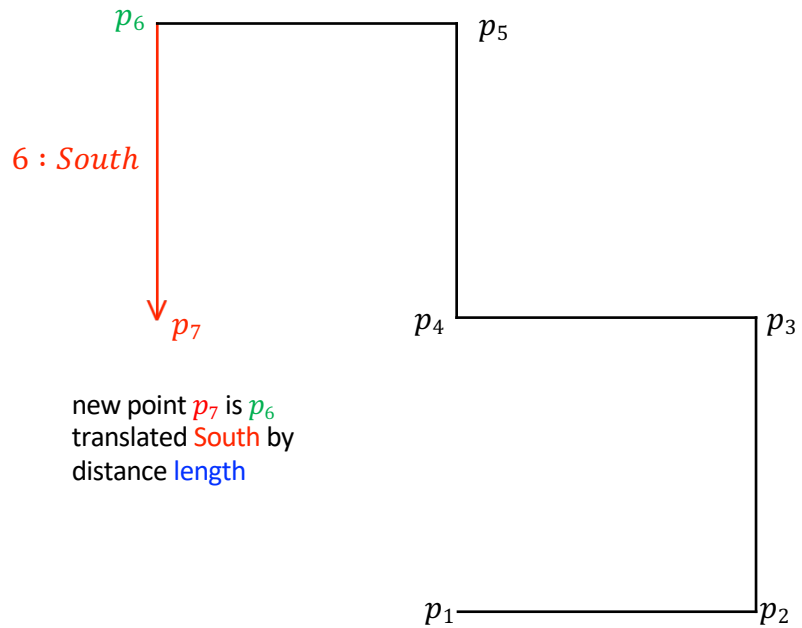
new point p_4 is p_3
translated West by
distance length




new point p_5 is p_4
translated North by
distance length
p_5
$4 : North$
p_4
p_3
p_1
p_2

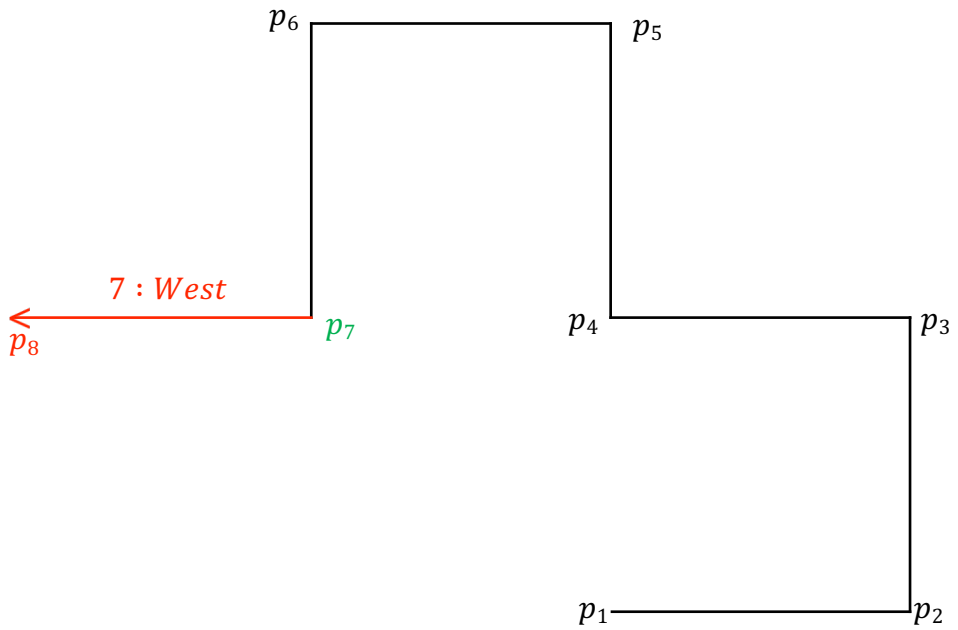
new point p_6 is p_5 translated West by distance length



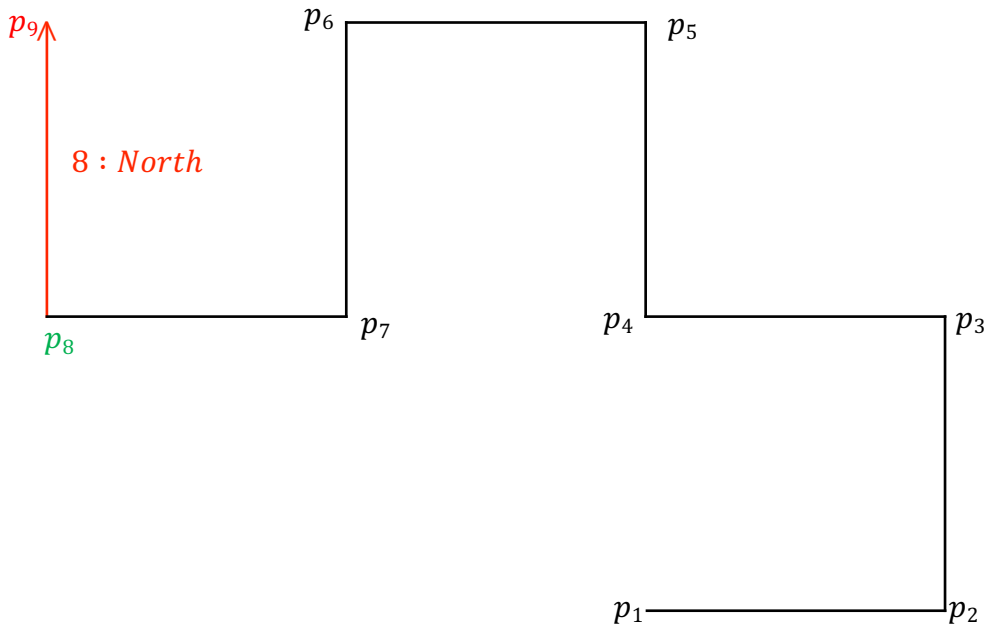


new point p_7 is p_6
translated South by
distance length

new point p_8 is p_7
translated West by a
distance length

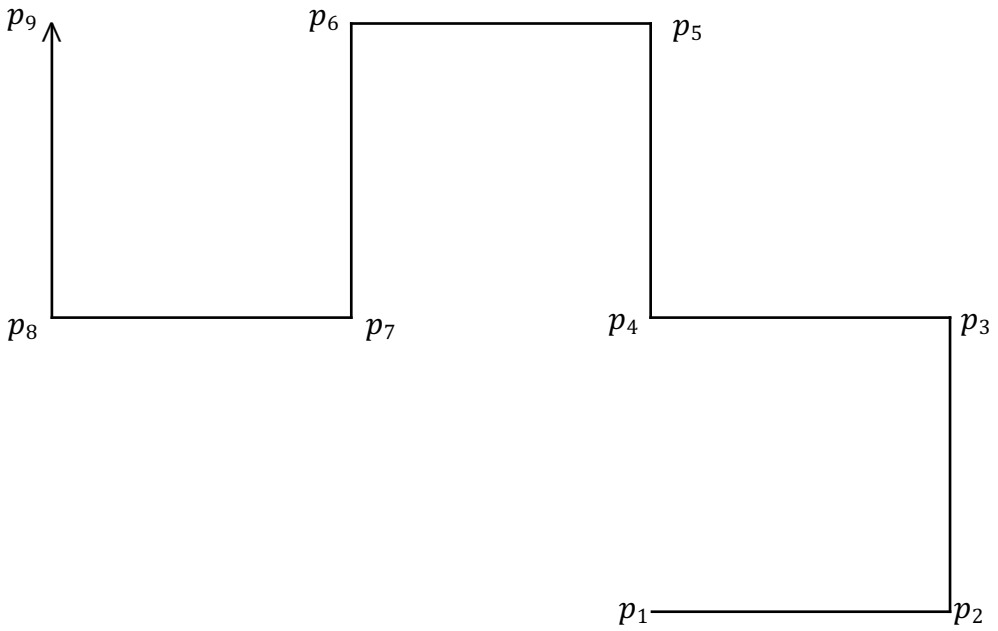


new point p_9 is p_8
translated North by a
distance length



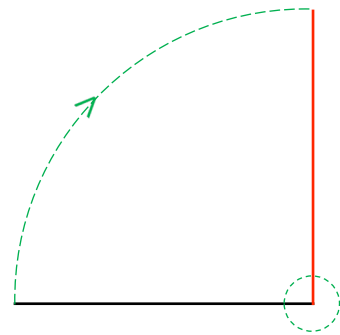
Here is the **path** computed by that **process**. Does the path **exhibit** any **self-similarity**?

Are some **parts** of the **path** the **same** as other **parts**? Let's find out in the next six slides.

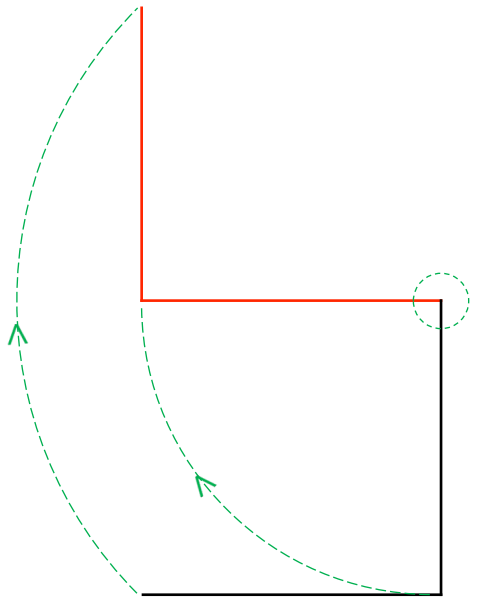


There is no **self-similarity**
in a dragon aged **0**.

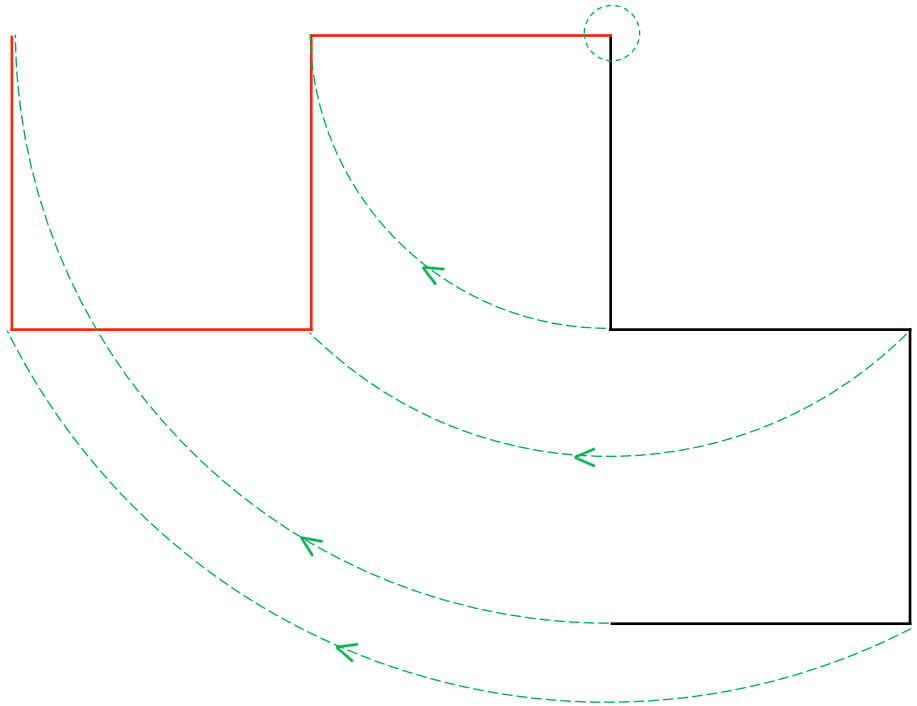
A **dragon aged 1** exhibits **self-similarity** in that the **second part** of its **path** can be computed by taking the **first part** of the **path**, i.e. a **dragon aged 0**, and **rotating** it **ninety degrees** about the **end point** of the **first part**.



A **dragon aged 2** also exhibits the same **self-similarity**: the **second part** of its **path** can be computed by taking the **first part** of the **path**, i.e. a **dragon aged 1**, and **rotating** it **ninety degrees** about the **end point** of the **first part**.



A **dragon aged 3** again exhibits the same **self-similarity**: the **second part** of its **path** can be computed by taking the **first part** of the **path**, i.e. a **dragon aged 2**, and **rotating** it **ninety degrees** about the **end point** of the **first part**.



Let us now modify the **high level path-growing process** so that rather than **computing** the **next point** on a **dragon path** by **translating** the previously computed **point** (as seen below in the base case of the **grow function**), it instead **computes** new **points** on the **path** of a **dragon aged N+1** by **rotating** the **points** on the **path** of a **dragon aged N**.

```
extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)
```

The **new process** for a **dragon aged N** consists of **phases 0** through to **N-1**.

The next **n** slides visualise the **new process** for a **dragon aged 4**.

Phase 0

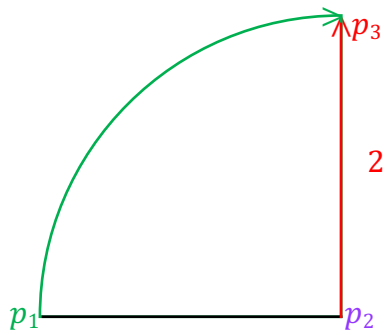
# of points: $2 = 2^0+1$
# of lines: $1 = 2^0$

We start off with two points

Phase 1

- iterate through the existing points on the path in reverse order
- rotating each existing point so as to create a new point
- which then becomes the ‘next’ point on the path

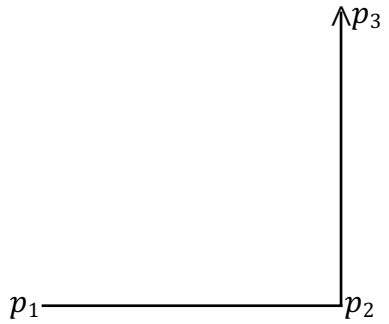
In phase one there is only one existing point to process: p_1



new point p_3 is p_1 rotated 90° about p_2

Phase 1 result

# of points: $3 = 2^1+1$
# of lines: $2 = 2^1$



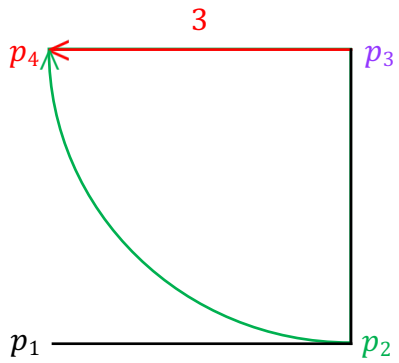
Phase 2

- iterate through the existing points on the path in reverse order
- rotating each existing point so as to create a new point
- which then becomes the ‘next’ point on the path

In phase two there are two existing points to process: p_2 and p_1

1st point

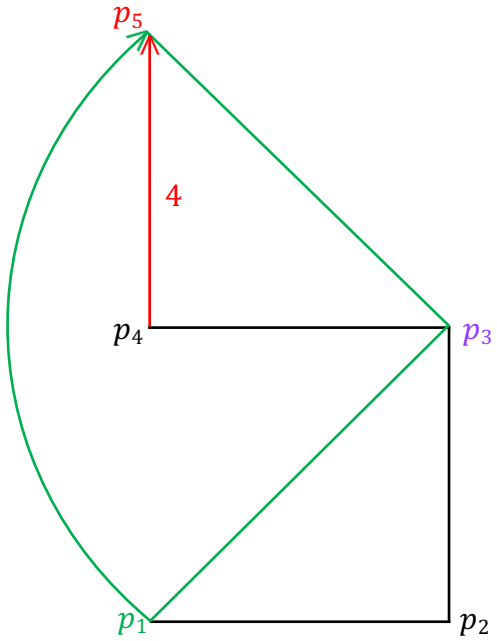
new point p_4 is p_2 rotated 90° about p_3



Phase 2

2nd point

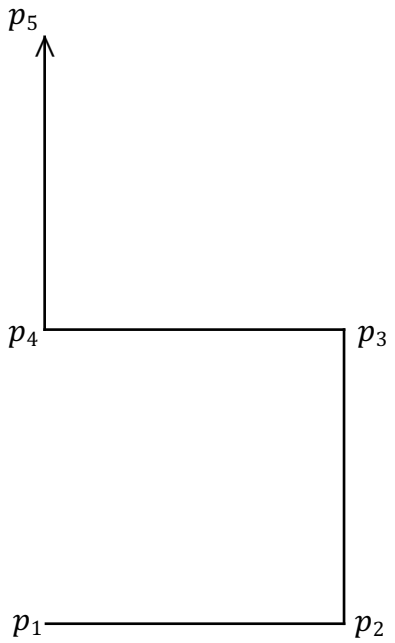
new point p_5 is p_1 rotated 90° about p_3



Phase 2 result

# of points: $5 = 2^2+1$

# of lines: $4 = 2^2$



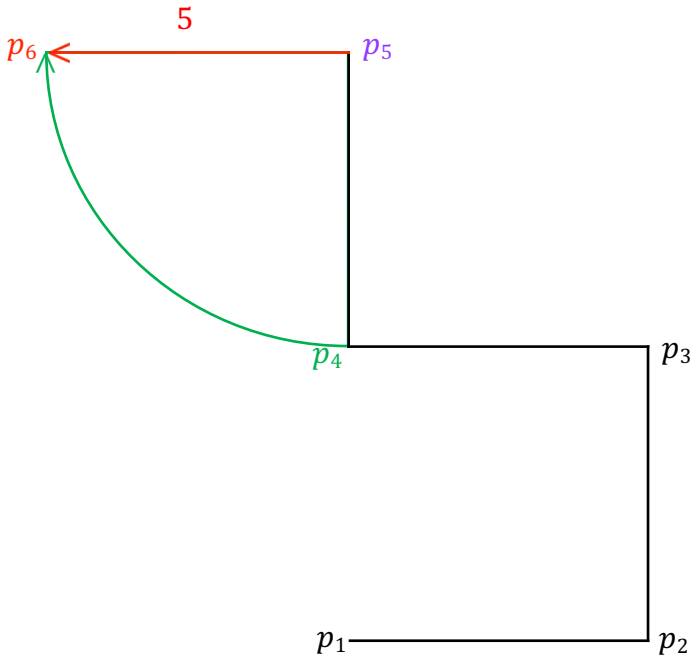
Phase 3

- iterate through the existing points on the path in reverse order
- rotating each existing point so as to create a new point
- which then becomes the ‘next’ point on the path

In phase three there are four existing points to process:
p_4, p_3, p_2, p_1

1st point

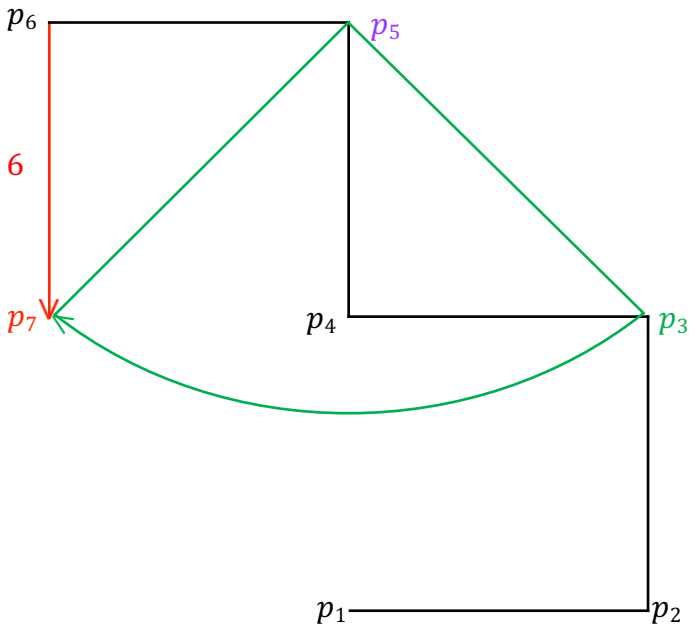
new point p_6 is p_4 rotated 90° about p_5



Phase 3

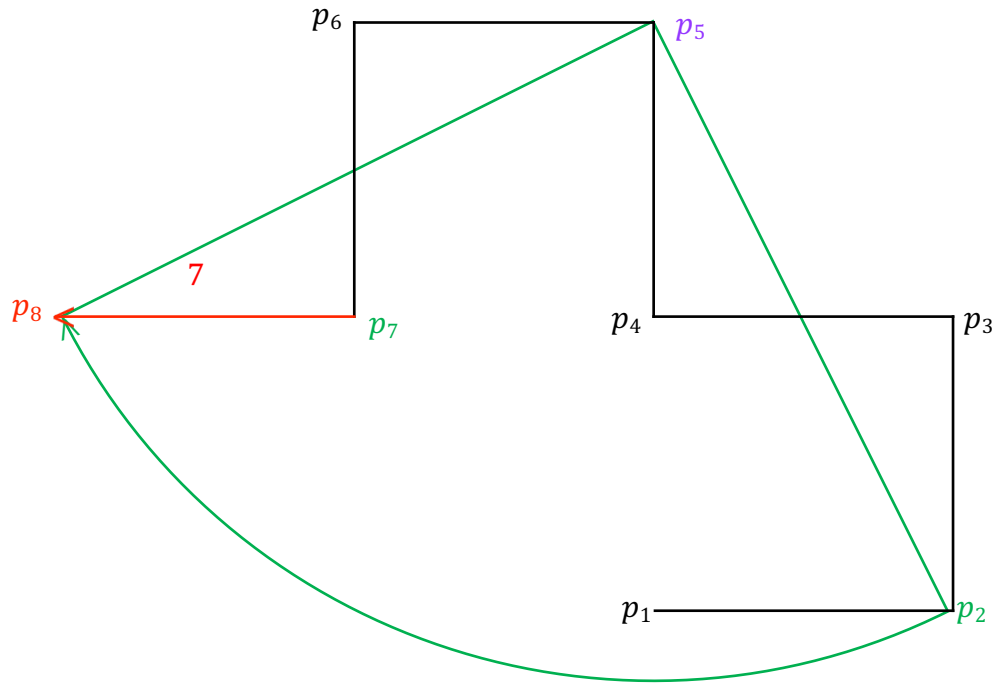
2nd point

new point p_7 is p_3 rotated 90° about p_5



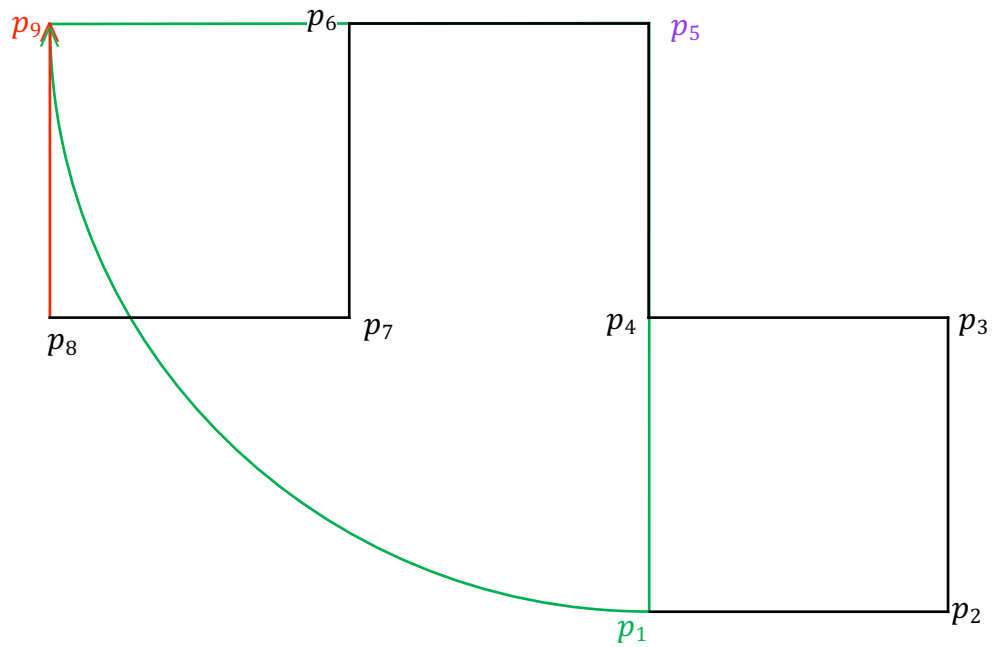
3rd point

new point p_8 is p_2 rotated 90° about p_5



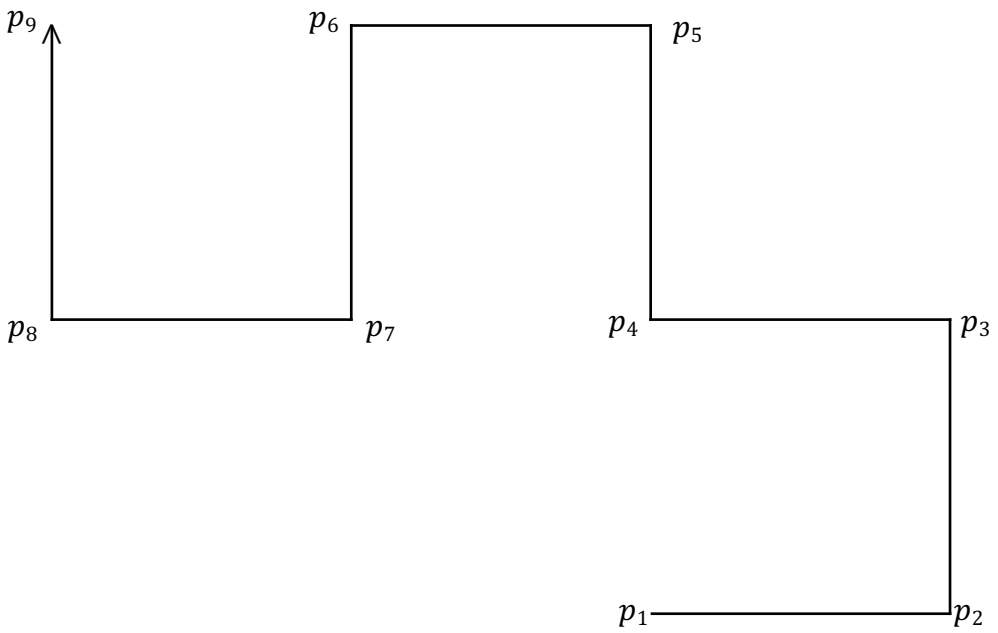
4th point

new point p_9 is p_1 rotated 90° about p_5



Phase 3 result

# of points: $9 = 2^3+1$
# of lines: $8 = 2^3$



Because the **new path-growing process** computes **new points** simply by **rotating existing points**, the **grow function** no longer needs to concern itself with the concepts of **direction** and **line length**.

Remember how the **new process** begins with **two points**?

$p_1 \longrightarrow p_2$

Direction and **line length** are now only needed to compute p_2, when we create the **initial dragon path**.

```
object DragonPath:

  def apply(startPoint: Point): DragonPath = List(startPoint)
```

```
object DragonPath:

  def apply(startPoint: Point, direction: Direction, length: Int): DragonPath =
    val nextPoint = startPoint.translate(direction, distance = length)
    List(nextPoint, startPoint)
```

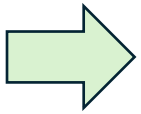
```
extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    <body of function – not shown>
```

```
extension (path: DragonPath)

  def grow(age: Int): DragonPath =

    <new body of function – to be defined>
```

Note that the **apply function** now returns `List(nextPoint, startPoint)`. We have **inverted** the **order of points** in a **path**. Instead of new **points** being added to the **end** of the **path**, they will now new be added at the **front** of the **path**. We are doing this because it is more efficient to **cons** a new **point** onto a **path** than it is to **append** it to the **end** of the **path**. Doing so is legitimate because when the time comes to draw a **line** connecting **points A** and **B**, it makes no difference whether we draw a **line** from **A** to **B** or one from **B** to **A**.

Let's update the Dragon's path to reflect the improvements made on the previous slide.

```
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start)
      .grow(age, length, direction)
```



```
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start, direction, length)
      .grow(age)
```

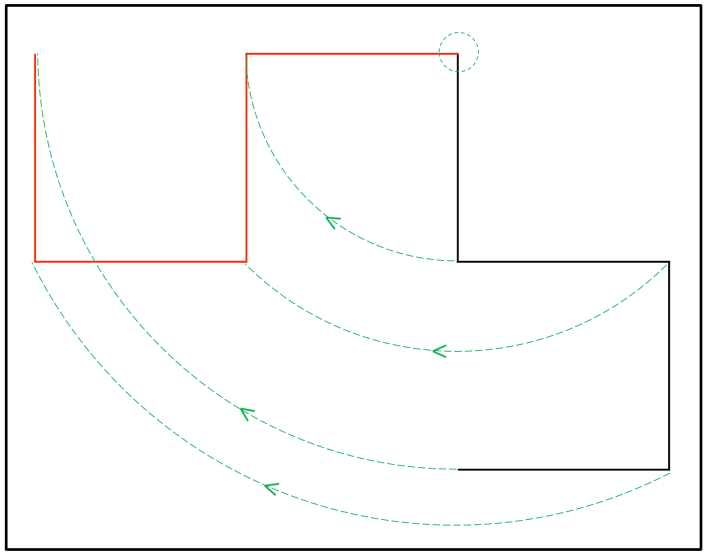
Now it is time to reimplement the **body** of the **grow** function so that it exploits the **dragon**'s **self-similarity**.

Instead of computing the **next new point** on a **dragon path** of **age N** by **translating** the last **point** on the **path**, it has to compute **new points** using the **N-phase approach** described earlier on, in which each **phase** involves the following **steps**:

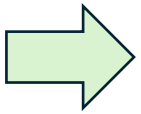
- iterate through the existing points on the path in reverse order
- rotating each existing point so as to create a new point
- which then becomes the 'next' point on the path

While the function remains **recursive**, it now becomes **tail-recursive**: instead of invoking itself **twice**, it invokes itself only **once**, that being the last thing it does.

Rotation of the **points** on a **dragon path** around a **rotation centre** specified by another **point** is **delegated** to a **function** called **rotate**.



```
extension (path: DragonPath)

  def grow(age: Int): DragonPath =

    <new body of function – to be defined>
```



```
extension (path: DragonPath)

  @tailrec
  def grow(age: Int): DragonPath =
    if age == 0 || path.size < 2 then path
    else path.plusRotatedCopy.grow(age - 1)

  private def plusRotatedCopy: DragonPath =
    path.reverse.rotate(rotationCentre = path.head, angle = ninetyDegreesClockwise)
    ++ path
```

```
val ninetyDegreesClockwise = -Math.PI / 2
```

Now that the **grow function** exploits the **dragon's self-similarity**, it is very easy to **understand** how the **function** accomplishes its task: to **grow** a **path**, first **add to it** a **rotated copy of itself**, and then **recursively grow** the **resulting path**.

The **rotation** is **centred** on the **point** that was **most recently** added to the **path**. The **angle of rotation** is defined in **radians**, and is a **negative value** because a **clockwise rotation** is required, rather than the **more customary**, **anti-clockwise** one.

On the previous slide we saw that the **grow function** now delegates **rotation** of the **points** on a **dragon path** to a **function** called **rotate**. Here is its implementation:

```
type Radians = Double

extension (points: List[Point])
  def rotate(rotationCentre: Point, angle: Radians): List[Point] =
    points.map(point => point.rotate(rotationCentre, angle))
```

The **function** doesn't do a lot. To **rotate** a number of **points** by a given **angle** (in **radians**) around a **rotation centre** specified by a **point**, it just **delegates** the **rotation** of each **point** to a second **rotate function** with the following signature:

```
def rotate(rotationCentre: Point, angle: Radians): Point
```

We'll very soon be turning our attention to the implementation of this **second rotate function**.

Now that we enjoy the benefit of having a plusRotatedCopy function whose intention-revealing name makes the exact details of its body less important for the purpose of quickly understanding how dragon paths are grown, it makes sense to make the function more efficient: instead of the function appending to the path a reversed and rotated copy of the path, it can use a left fold to avoid the append operation and do both the reversing and the rotation at the same time

```
private def plusRotatedCopy =
  path.reverse.rotate(rotationCentre = path.head, angle = ninetyDegreesClockwise)
    ++ path
```

```
private def plusRotatedCopy =
  path.foldLeft(path): (growingPath, point) =>
    point.rotate(rotationCentre = path.head, angle = ninetyDegreesClockwise)
      :: growingPath
```

Here is a **recap** of how we changed the **grow function**.

```
extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)
```

```
extension (path: DragonPath)

  @tailrec
  def grow(age: Int): DragonPath =
    if age == 0 || path.size < 2 then path
    else path.plusRotatedCopy.grow(age - 1)

  private def plusRotatedCopy =
    path.foldLeft(path): (growingPath, point) =>
      point.rotate(rotationCentre = path.head, angle = ninetyDegreesClockwise)
        :: growingPath
```

```
val ninetyDegreesClockwise = -Math.PI / 2
```

A couple of slides ago we trivially implemented the **rotate** function that operates on a **dragon path** by getting it to **delegate** the **rotation** of a **single point** on that **path** to a **second rotate function**, with the following **signature**:

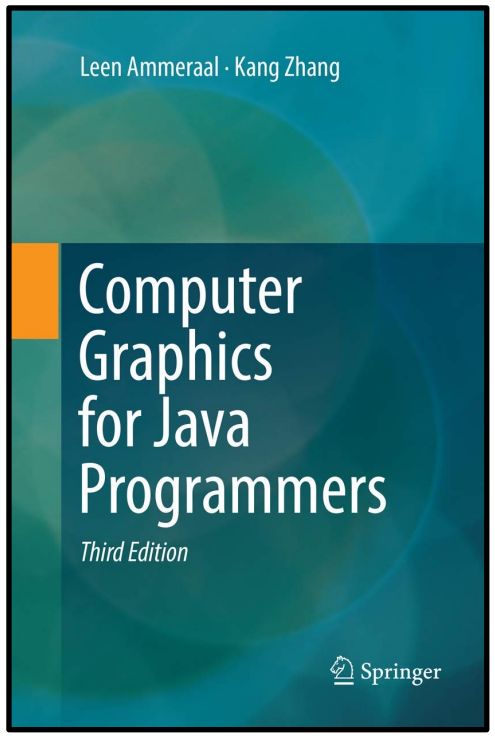
```
def rotate(rotationCentre: Point, angle: Radians): Point
```

How can we **implement** this **function**?

Let's consult **Computer Graphics for Java Programmers** and learn (or reacquaint ourselves with) just enough **graphics-related mathematics** to be able to **implement** the **function**.

Leen Ammeraal

https://www.linkedin.com/in/leen-ammeraal-b97b968/



Kang Zhang

https://profiles.utdallas.edu/kang.zhang

Rotation of a **point** about the **origin** of a **two-dimensional cartesian coordinate system** is a **linear transformation**.

A **linear transformation** maps a **vector** $\mathbf{v}$ to another **vector** $\mathbf{v'}$.

So the first **concept** that we need to be familiar with is that of a **vector**.

2.1 Vectors

…

A **vector** is a **directed line segment**, characterized by its **length** and its **direction** only. Figure 2.1 shows two **representations** of the same **vector** $\mathbf{a} = \mathbf{PQ} = \mathbf{b} = \mathbf{RS}$. Thus a **vector** is not altered by a **translation**.

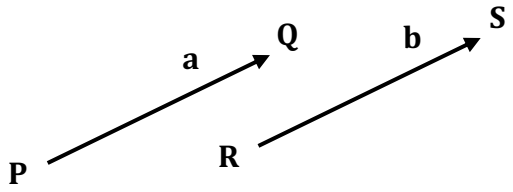


Figure 2.1 Two equal vectors

The sum $\boldsymbol{c}$ of the vectors $\mathbf{a}$ and $\mathbf{b}$, written

$$\mathbf{c} = \mathbf{a} + \mathbf{b}$$

can be obtained as the **diagonal** of a **parallelogram**, with $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ starting at the same **point**, as shown in Figure 2.2.

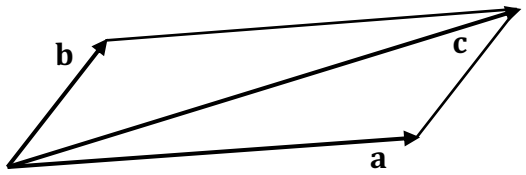
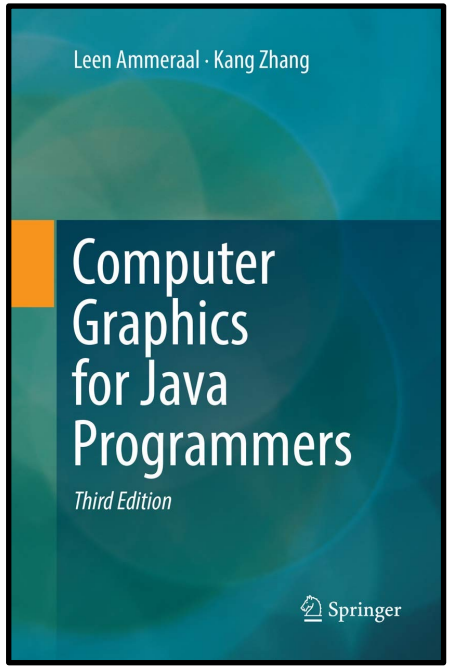


Figure 2.2 Vector Addition

The **length** of a **vector** $\mathbf{a}$ is denoted by $|\mathbf{a}|$. A **vector** with **zero length** is the **zero vector**, written as $\mathbf{0}$. The **notation** $-\mathbf{a}$ is used for the **vector** that has **length** $|\boldsymbol{a}|$ and whose **direction** is **opposite** to that of $\mathbf{a}$. For any **vector** $\mathbf{a}$ and real number c, the **vector** $c\mathbf{a}$ has **length** $|c||\mathbf{a}|$. If $\mathbf{a} = \mathbf{0}$ or $c = 0$, then $c\mathbf{a} = \mathbf{0}$; …



Leen Ammeraal

Kang Zhang

Springer

The **next slide** is **only relevant** in that it introduces **a couple of bits of terminology** that will be **referenced later**.

I recommend just speeding through it, concentrating only on the two sections highlighted in yellow.

Figure 2.3 shows three **unit vectors** $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ in a **three-dimensional space**. They are **mutually perpendicular** and have **length** 1. Their **directions** are the **positive directions** of the **coordinate axes**. …

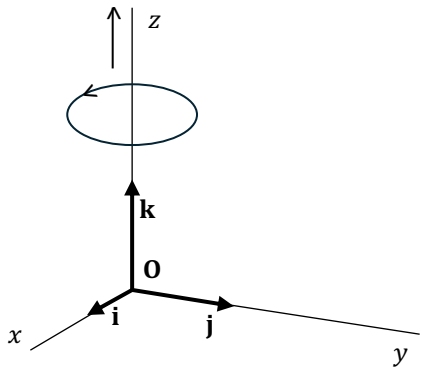


Figure 2.3: Right-handed coordinate system

We say that $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ form a **triple** of **orthogonal unit vectors**. The **coordinate system** is **right-handed**, which means that if a **rotation** of $\mathbf{i}$ in the **direction** of $\mathbf{j}$ through 90◦ corresponds to turning a **right-handed screw**, then $\mathbf{k}$ has the **direction** in which the **screw** advances.

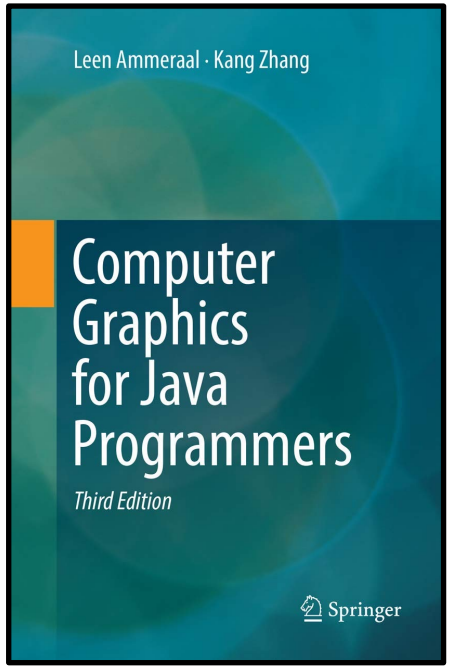
We often choose the **origin** $\mathbf{O}$ of the **coordinate system** as the **initial point** of all **vectors**. Any **vector** $\mathbf{v}$ can be written as a **linear combination** of the **unit vectors** $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$:

$$\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

The **real numbers** x, y and z are the **coordinates** of the **endpoint** $\mathbf{P}$ of **vector** $\mathbf{v} = \mathbf{OP}$. We often write this **vector** $\mathbf{v}$ as

$$\mathbf{v} = [\ x\ \ y\ \ z\ ] \qquad \text{or} \qquad \mathbf{v} = (x,\ y,\ z)$$

The numbers x, y and z are sometimes called the **elements** or **components** of **vector** v.



Leen Ammeraal

Kang Zhang

Springer

The reason why we are interested in expressing **rotation** as a **linear transformation**, is that the latter can be written as a **matrix multiplication**.

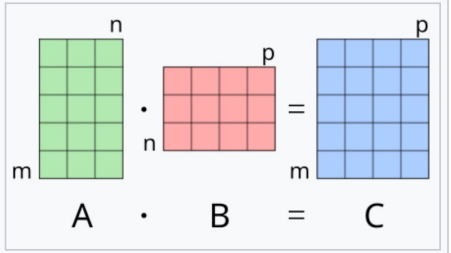
So the next **concept** that we need to be familiar with is that of **matrix multiplication**.

WIKIPEDIA
The Free Encyclopedia

Matrix multiplication

From Wikipedia, the free encyclopedia

In mathematics, specifically in linear algebra, **matrix multiplication** is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the **matrix product**, has the number of rows of the first and the number of columns of the second matrix. The product of matrices **A** and **B** is denoted as **AB**.



For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The result matrix has the number of rows of the first and the number of columns of the second matrix.

Matrix times matrix [edit]

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix[5][6][7][8]

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

That is, the entry c_{ij} of the product is obtained by multiplying term-by-term the entries of the ith row of $\mathbf{A}$ and the jth column of $\mathbf{B}$, and summing these n products. In other words, c_{ij} is the dot product of the ith row of $\mathbf{A}$ and the jth column of $\mathbf{B}$.

Therefore, $\mathbf{AB}$ can also be written as

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

Thus the product $\mathbf{AB}$ is defined if and only if the number of columns in $\mathbf{A}$ equals the number of rows in $\mathbf{B}$,[1] in this case n.

The next **concept** that we need to be familiar with is that of a **linear transformation**.

3.2 Linear Transformation

A **transformation** T is a **mapping**

$$\mathbf{v} \to \mathrm{T}\mathbf{v} = \mathbf{v}'$$

such that each **vector** $\mathbf{v}$ (in the **vector space** we are dealing with) is assigned its **unique image** $\mathbf{v}'$. Let us begin with the xy-plane and associate with each **vector** $\mathbf{v}$ the point P, such that

$$\mathbf{v} = \mathbf{OP}$$

$\mathbf{O}$ is the **origin** of the **coordinate system**

Then the **transformation** T is also a **mapping**

$$\mathrm{P} \to \mathrm{P}'$$

for each point P in the xy-plane, where $\mathbf{OP}' = \mathbf{v}'$.

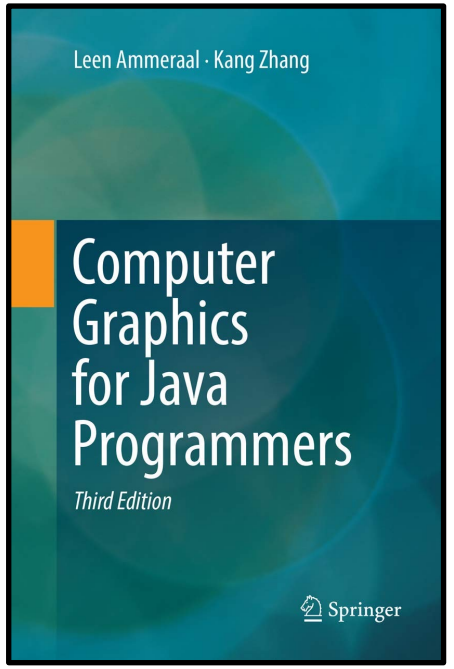
A **transformation** is said to be **linear** if the following is true for any two **vectors** $\mathbf{v}$ and $\mathbf{w}$ and for any **real number** λ:

$$\mathrm{T}(\mathbf{v} + \boldsymbol{w}) = \mathrm{T}(\mathbf{v}) + T(\boldsymbol{w})$$
$$\mathrm{T}(\lambda\mathbf{v}) = \lambda T(\mathbf{v})$$

By using $\lambda = 0$ in the last **equation**, we find that, for any **linear transformation**, we have

$$\mathrm{T}(\mathbf{0}) = \mathbf{0}$$

$\mathbf{0}$ is the **zero vector**, the **vector** of **length zero**



Leen Ammeraal

Kang Zhang

Springer

We can write any **linear transformation** as a **matrix multiplication**. For example, consider the following **linear transformation**:

$$\begin{cases} x' = 2x \\ y' = x + y \end{cases}$$

We can write this as the **matrix product**

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.1}$$

Or as the following:

$$[x' \quad y'] = [x \quad y] \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} \tag{3.2}$$

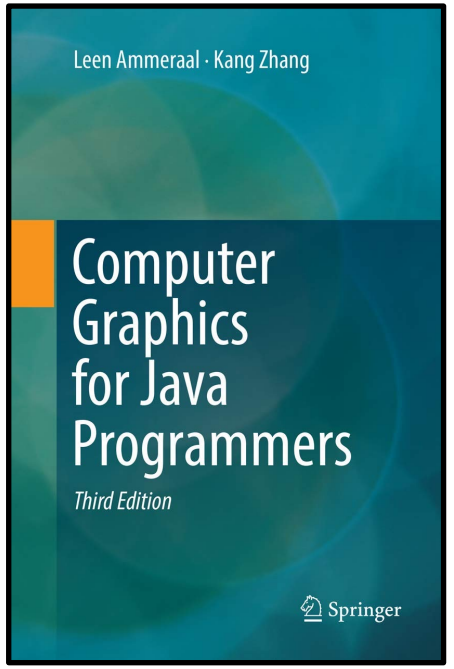
The above **notation** (3.1) is normally used in **standard mathematics textbooks**; in **computer graphics** and **other applications** in which **transformations** are **combined**, the **notation** of (3.2) is also **popular** because it avoids a source of **mistakes**, as we will see in a moment. We will therefore adopt this **notation**, using **row vectors**.

It is interesting to note that, in (3.2), the **rows** of the 2×2 **transformation matrix** are the **images** of the **unit vectors** $(1,0)$ and $(0,1)$, respectively, while these **images** are the **columns** in (3.1). You can easily verify this by **substituting** $[1 \ 0]$and $[0 \ 1]$ for $[x \ y]$ in (3.2), as the bold **matrix elements** below illustrate:

$$[\mathbf{2} \quad \mathbf{1}] = [1 \quad 0] \begin{bmatrix} \mathbf{2} & \mathbf{1} \\ 0 & 1 \end{bmatrix}$$

$$[\mathbf{0} \quad \mathbf{1}] = [0 \quad 1] \begin{bmatrix} 2 & 1 \\ \mathbf{0} & \mathbf{1} \end{bmatrix}$$

This principle also applies to other **linear transformations**. It provides us with a **convenient way** of finding the **transformation matrices**.



Leen Ammeraal

Kang Zhang

Springer

Next, let's see how in **two-dimensional space**, **rotation** of a **point** P **about the origin** can be expressed as the **multiplication** of the **row vector** for $\mathbf{OP}$ by a 2×2 **matrix**.

Rotation

To rotate all points in the xy-plane about **O** through the angle φ, we can now easily write the transformation matrix, using the rule we have just been discussing. We simply find the images of the unit vectors $(1,0)$ and $(0,1)$. As we know from elementary trigonometry, rotating the points $P(1,0)$ and $Q(0,1)$ about **O** through the angle φ gives $P(\cos\varphi, \sin\varphi)$ and $Q(-\sin\varphi, \cos\varphi)$. It follows that $(\cos\varphi, \sin\varphi)$ and $(-\sin\varphi, \cos\varphi)$ are the desired images of the unit vectors $(1,0)$ and $(0,1)$, as Figure 3.1 illustrates.

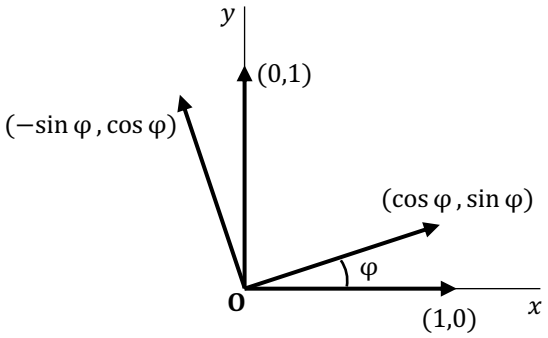
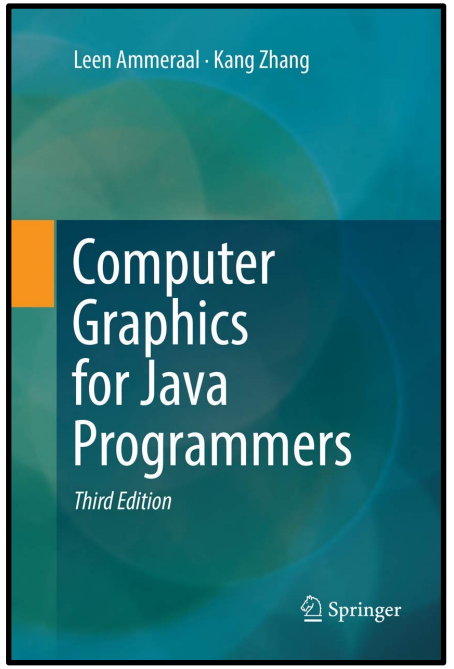


Figure 3.1: Rotation of unit vectors

Then all we need to do is to write these two images as the rows of our rotation matrix:

$$[x' \quad y'] = [x \quad y]\begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \tag{3.3}$$



Leen Ammeraal

Kang Zhang

Springer

We have seen that because **rotation** of a **point** P **about** the **origin** is a **linear transformation**, in **two-dimensional space** we can **express** such a **rotation** as the **multiplication** of the **row vector** for **OP** by a 2×2 **matrix**.

The **problem** is that what we want to do is **rotate** a **point** not **about** the **origin**, but **about** an **arbitrary point**, but it turns out that such a **rotation** is not a **linear transformation**, so we can't use the same **approach** to **express** the **rotation** as a **matrix multiplication**.

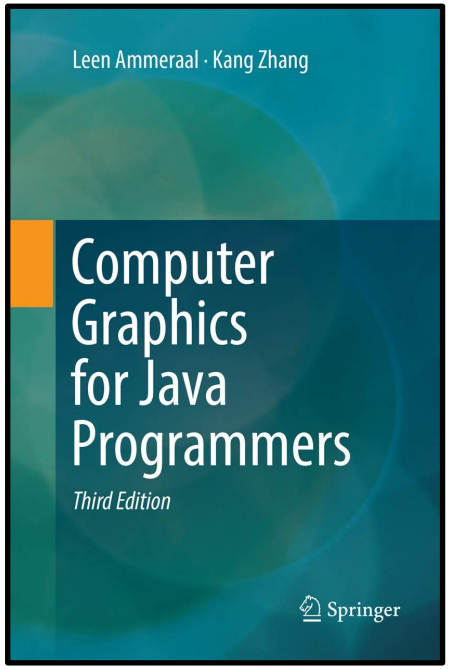
But there is a **solution** to the **problem**.

3.6 Rotation about an arbitrary point

So far we have only performed **rotations** about the **origin** O. A **rotation** about any **point** other than O is not a **linear transformation**, since it does not **map** the **origin** onto **itself**. It can nevertheless be described by a **matrix multiplication**, provided we use **homogeneous coordinates**. A **rotation** about the **point** C (x_C, y_C) through the **angle** φ can be performed in three steps:

1. A **translation** from C to O, …
   …
2. A **rotation** about O **through** the **angle** φ …
   …
3. A **translation** from O to C …
   …

So the next two **concepts** that we need to be familiar with are **translations** and **homogenous coordinates**.



Leen Ammeraal

Kang Zhang

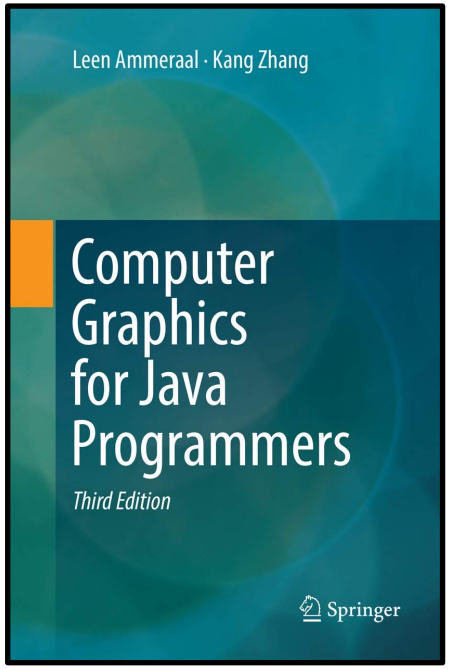
Springer

3.3 Translations

Shifting all **points** in the xy-plane a **constant distance** in a **fixed direction** is referred to as a **translation**. This is another **transformation**, which we can write as:

$$\begin{cases} x' = x + a \\ y' = y + b \end{cases}$$

We refer to the number pair (a, b) as the **shift vector,** or **translation vector**. Although this **transformation** is a very simple one, it is not **linear**, as we can easily see by the fact that the **image** of the **origin** $(0,0)$ is (a, b) , while this can only be the **origin** itself with **linear transformations**. Consequently, we cannot obtain the **image** $(x,\ y)$ by multiplying $(x,\ y)$ by a 2×2 **transformation matrix** T, which prevents us from combining such a **matrix** with other **transformation matrices** to obtain **composite transformations**. Fortunately, there is a solution to this problem as described in the following section.



Leen Ammeraal

Kang Zhang

Springer

3.4 Homogenous Coordinates

To express all the **transformations** introduced so far as **matrix multiplications** in order to **combine** various **transformation effects**, we add one more **dimension**. As illustrated in Figure 3.4, the **extra dimension** W makes any point $P = (x, y)$ of **normal coordinates** have a whole **family** of **homogeneous coordinate representations** (wx, wy, w) for any value of w except $\mathbf{0}$. For example, $(3, 6, 1)$, $(0.3, 0.6, 0.1)$, $(6, 12, 2)$, $(12, 24, 4)$ and so on, represent the same point in **two-dimensional space**. Similarly, 4-tuples of **coordinates** represent points in **three-dimensional space**. When a **point** is mapped onto the $W = 1$ **plane**, in the form $(x, y, 1)$, it is said to be **homogenized**. In the above example, point $(3,6,1)$ is **homogenized**, and the numbers 3, 6 and 1 are **homogeneous coordinates**.

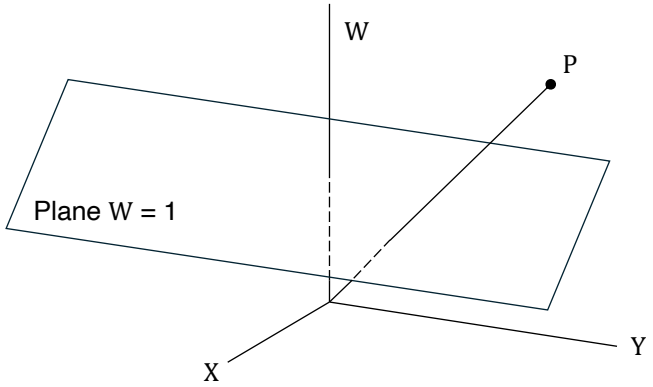
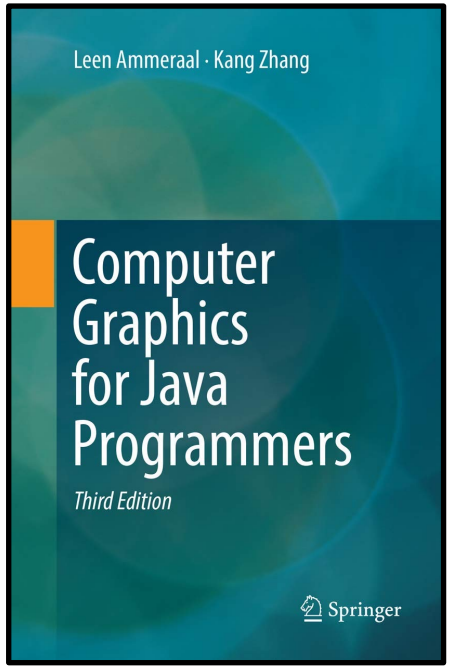


Figure 3.4: A homogeneous coordinate system with the plane $W = 1$

In general, to convert a **point** from **normal coordinates** to **homogeneous coordinates**, add a new **dimension** to the right with value 1. To convert a **point** from **homogeneous coordinates** to **normal coordinates**, divide all the **dimension values** by the rightmost **dimension value**, and then discard the **rightmost dimension**.



Leen Ammeraal

Kang Zhang

Springer

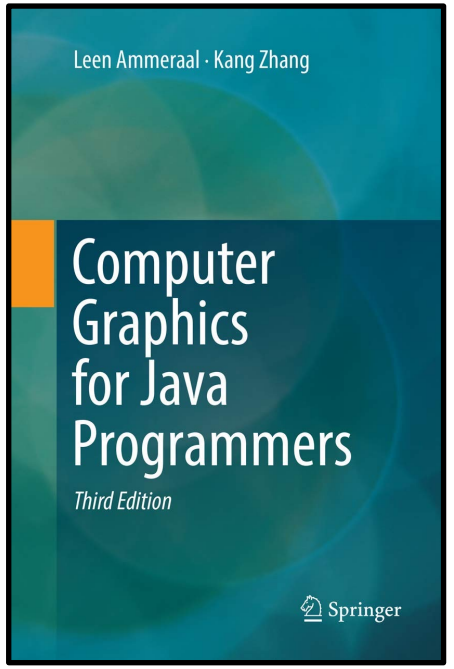
Next, let's see how **translation** can be described by a 3×3 **matrix** and how the 2×2 **rotation matrix** that we saw earlier on can be described as a 3×3 **matrix**.

Having introduced **homogeneous coordinates**, we are able to describe a **translation** by a **matrix multiplication** using a 3×3 instead of a 2×2 **matrix**. Using a **shift vector** (a, b), we can write the **translation** of Section 3.3 as the following **matrix product**:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{bmatrix}$$

Since we cannot **multiply** a 3×3 by a 2×2 **matrix**, we will also add a **row** and a **column** to **linear transformation matrices** if we want to **combine** these with **translations** (and possibly with other **non-linear transformations**). These **additional rows** and **columns** simply consist of **zeros** followed by a **one** at the end. For example, we can use the following **equation** instead of 3.3 (in Section 3.2) for a **rotation** about O **through** the **angle** φ:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\begin{bmatrix}\cos\varphi & \sin\varphi & 0\\ -\sin\varphi & \cos\varphi & 0\\ 0 & 0 & 1\end{bmatrix}$$



Leen Ammeraal

Kang Zhang

Springer

We are finally ready to see the 3×3 **matrix** that we can use to **rotate** a **point** about **arbitrary point** C (x_C, y_C) **through angle** φ.

A **translation** from C to **O**

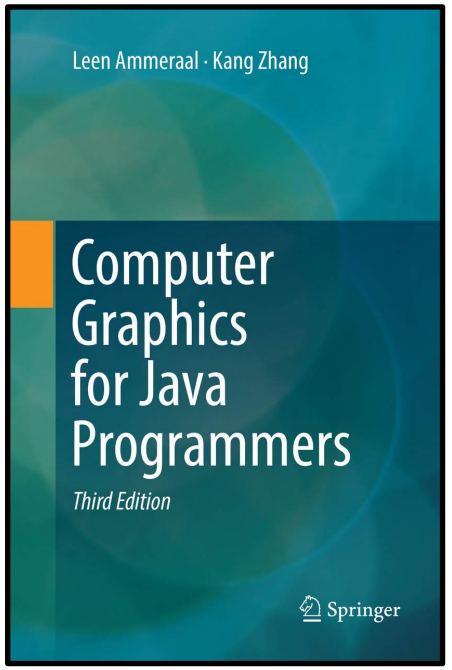
A **rotation** about **O** **through** the **angle** φ

A **translation** from **O** to C

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

$$= \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ -x_C\cos\varphi + y_C\sin\varphi + x_C & -x_C\sin\varphi - y_C\cos\varphi + y_C & 1 \end{bmatrix}$$

A **rotation** about **C** **through** the **angle** φ



Leen Ammeraal

Kang Zhang

Springer

To **rotate point** $\mathrm{P} = (x, y)$ **about point** $\mathrm{C} = (x_\mathrm{C}, y_\mathrm{C})$ **through** an **angle** φ, resulting in **rotated point** $\mathrm{P}' = (x', y')$, we are going to **multiply row vector** $[x \quad y \quad 1]$ by **rotation matrix** $\mathbf{R}$.

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\,\mathbf{R}$$

Let's use a **library** to do **matrix multiplication**.

The following should do the job.

GitHub https://github.com/dieproht/matr

README Apache-2.0 license

matr ['mætə]

build passing chat on gitter Scala Steward helping maven-central v0.0.3

A Scala 3 matrix library.

Matr is an attempt to bring together safety, flexibility and simplicity for matrix calculations.

- Safety through typed shapes and immutable data
- Flexibility through loose coupling between interface, data and operations
- Simplicity through an easily accessible API and "batteries included" default implementations, both with zero dependencies

Here is how we can use the **library** to **implement** the **rotate function**.

See this **deck**'s code **repository** for the necessary **library import**s.

```
extension (p: Point)

  def rotate(rotationCentre: Point, angle: Radians): Point =
    val (c, φ) = (rotationCentre, angle)
    val (cosφ, sinφ) = (math.cos(φ).toFloat, math.sin(φ).toFloat)
    val rotationMatrix: Matrix[3,3,Float] = MatrixFactory[3, 3, Float].fromTuple(
      (              cosφ,                          sinφ,              0f),
      (             -sinφ,                          cosφ,              0f),
      (-c.x * cosφ + c.y * sinφ + c.x,   -c.x * sinφ - c.y * cosφ + c.y,   1f)
    )
    val rowVector: Matrix[1, 3, Float] = MatrixFactory[1, 3, Float].rowMajor(p.x, p.y, 1f)
    val rotatedRowVector: Matrix[1, 3, Float] = rowVector dot rotationMatrix
    val (x, y) = (rotatedRowVector(0, 0), rotatedRowVector(0, 1))
    Point(x, y)
```

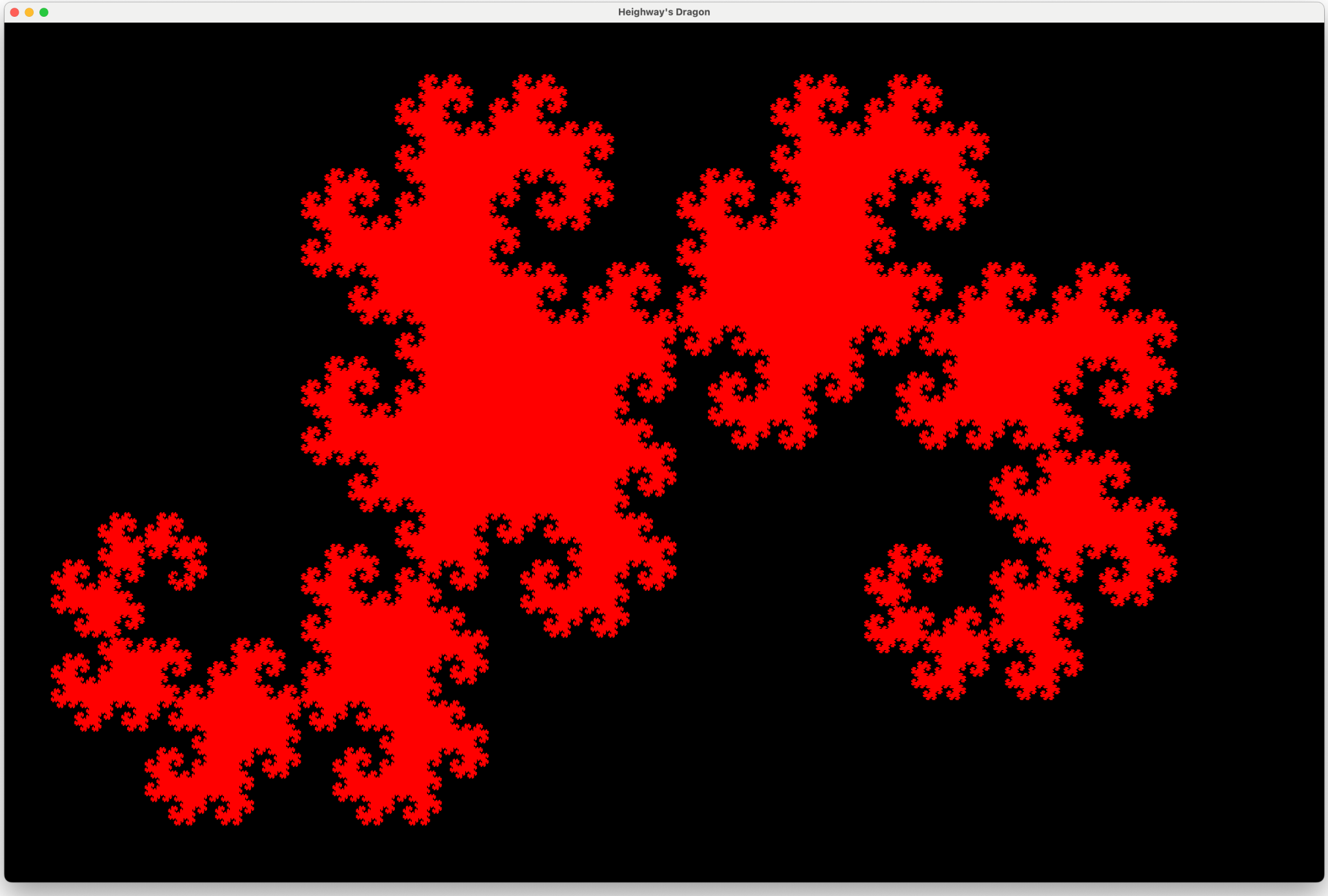
$$\mathbf{R} = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ -x_C\cos\varphi + y_C\sin\varphi + x_C & -x_C\sin\varphi - y_C\cos\varphi + y_C & 1 \end{bmatrix}$$

$$[x' \quad y' \quad 1] = [x \quad y \quad 1]\,\mathbf{R}$$

infix binary library function **dot** performs a **matrix multiplication** by calculating the **dot product**.

Now the we have implemented the function for rotating a point, let's use it to draw a **dragon** aged 20 with a **line length** of 1 **pixel**.

It works!



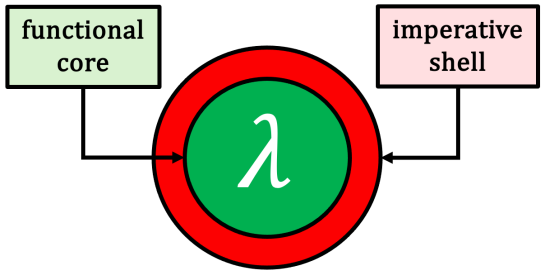
The next slide recaps the code for the **imperative shell**, which is unchanged apart from frame size, line colour and background colour.

The slide after that shows the code for the **functional core**, which is where we have been making our improvements and simplifications.

Of the two versions of the `plusRotatedCopy` function, we are showing the one that is particularly **easy** to **understand**, over the one that is **more performant**.

```
import javax.swing.SwingUtilities

@main def main(): Unit =
  // Create the frame/panel on the event dispatching thread.
  SwingUtilities.invokeLater(
    new Runnable():
      def run(): Unit = displayDragonFrame()
  )
```

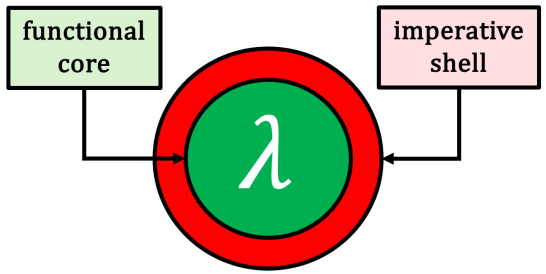
```
import java.awt.Color
import javax.swing.{JFrame, WindowConstants}

def displayDragonFrame(): Unit =
val panel = DragonPanel(lineColour = Color.red, backgroundColour = Color.black)
  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Heighway's Dragon")
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
  frame.setSize(1800,1200)
  frame.add(panel)
  frame.setVisible(true)
```




```
import java.awt.{Color, Graphics}
import javax.swing.*

class DragonPanel(lineColour: Color, backgroundColour: Color) extends JPanel:

  override def paintComponent(g: Graphics): Unit =

    val panelHeight = getSize().height - 1

    def startPoint: Point =
      val panelWidth = getSize().width - 1
      val panelCentre = Point(panelWidth / 2, panelHeight / 2)
      panelCentre
        .translate(South, panelHeight / 7)
        .translate(West, panelWidth / 5)

    def draw(line: Line): Unit =
      val (ax, ay) = line.start.deviceCoords(panelHeight)
      val (bx, by) = line.end.deviceCoords(panelHeight)
      g.drawLine(ax, ay, bx, by)

    def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
      Dragon(start, age, length, direction)
        .path
        .lines
        .foreach(draw)

    super.paintComponent(g)
    setBackground(backgroundColour)
    g.setColor(lineColour)

    drawDragon(startPoint, age = 17, length = 1, direction = East)
```

```scala
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start, direction, length)
      .grow(age)
```

```scala
type DragonPath = List[Point]

val ninetyDegreesClockwise: Radians = -Math.PI / 2

object DragonPath:

  def apply(startPoint : Point, direction: Direction, length: Int): DragonPath =
    val nextPoint = startPoint.translate(direction, amount = length)
    List(nextPoint, startPoint)

extension (path: DragonPath)

  def lines: List[Line] =
    if path.length < 2 then Nil
    else path.zip(path.tail)

  @tailrec
  def grow(age: Int): DragonPath =
    if age == 0 || path.size < 2 then path
    else path.plusRotatedCopy.grow(age - 1)

  private def plusRotatedCopy =
    path.reverse.rotate(rotationCentre=path.head, angle=ninetyDegreesClockwise)
      ++ path
```




```scala
type Line = (Point, Point)

extension (line: Line)
  def start: Point = line(0)
  def end: Point = line(1)
```

```scala
case class Point(x: Float, y: Float)

type Radians = Double

extension (p: Point)

  def deviceCoords(panelHeight: Int): (Int, Int) =
    (Math.round(p.x), panelHeight - Math.round(p.y))

  def translate(direction: Direction, amount: Float): Point =
    direction match
      case North => Point(p.x, p.y + amount)
      case South => Point(p.x, p.y - amount)
      case East  => Point(p.x + amount, p.y)
      case West  => Point(p.x - amount, p.y)

  def rotate(rotationCentre: Point, angle: Radians): Point =
    val (c, φ) = (rotationCentre, angle)
    val (cosφ, sinφ) = (math.cos(φ).toFloat, math.sin(φ).toFloat)
    val rotationMatrix: Matrix[3,3,Float] = MatrixFactory[3, 3, Float].fromTuple(
      (              cosφ,                              sinφ,               0f),
      (             -sinφ,                              cosφ,               0f),
      (-c.x * cosφ + c.y * sinφ + c.x,   -c.x * sinφ - c.y * cosφ + c.y,   1f)
    )
    val rowVector: Matrix[1,3,Float] = MatrixFactory[1,3,Float].rowMajor(p.x,p.y,1f)
    val rotatedRowVector: Matrix[1, 3, Float] = rowVector dot rotationMatrix
    val (x, y) = (rotatedRowVector(0, 0), rotatedRowVector(0, 1))
    Point(x, y)

extension (points: List[Point])

  def rotate(rotationCentre: Point, angle: Radians) : List[Point] =
    points.map(point => point.rotate(rotationCentre, angle))
```

```scala
enum Direction:
  case North, East, South, West
```

I hope you enjoyed that.

See you in part three.