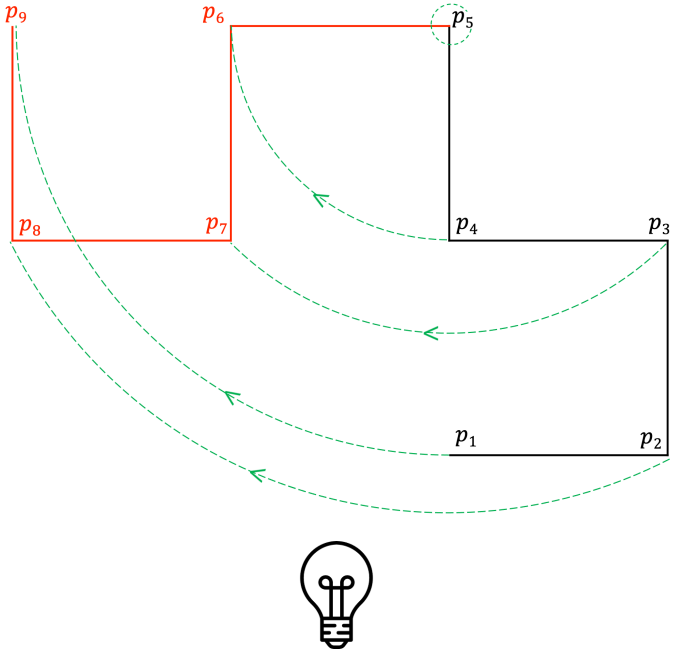
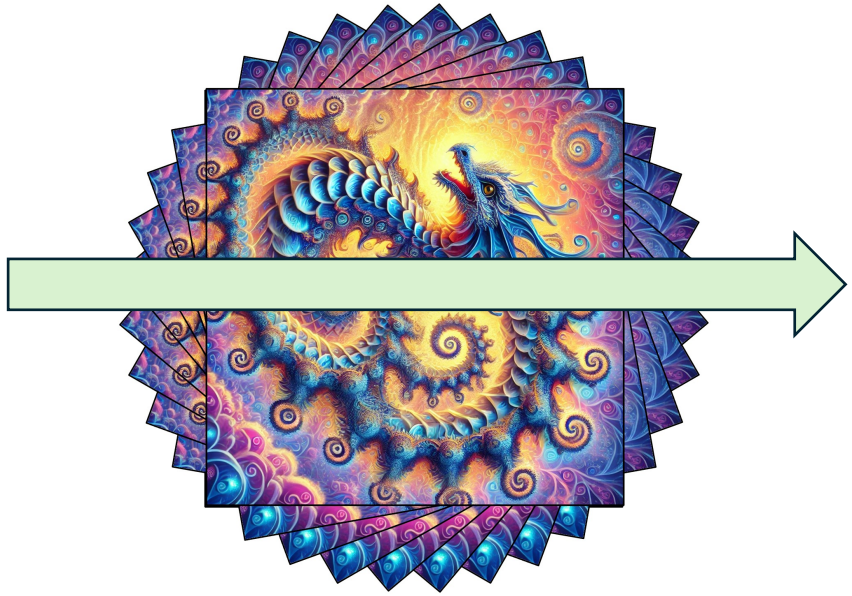
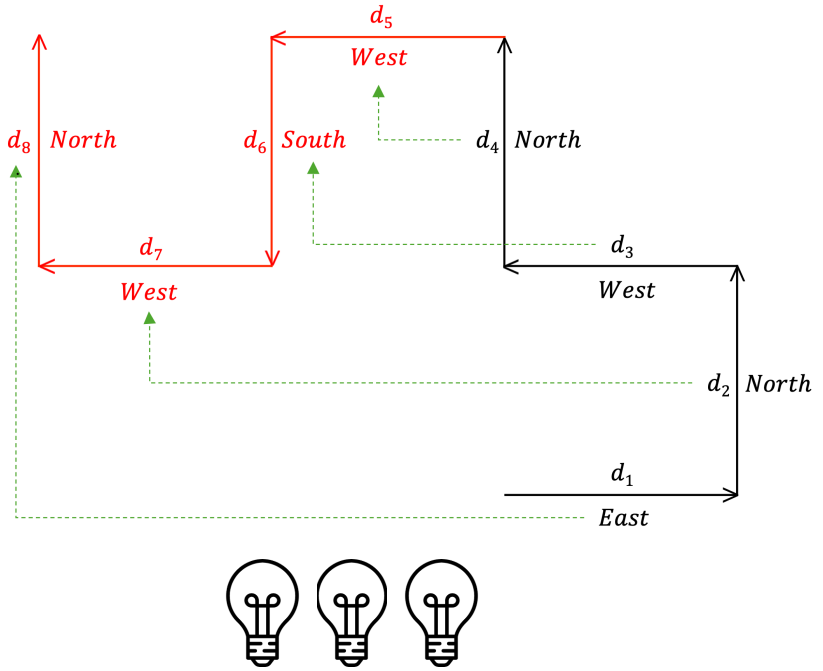
Drawing Heighway's Dragon Part 3

Simplification Through Separation of Concerns

Rotation Without Matrix Multiplication







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

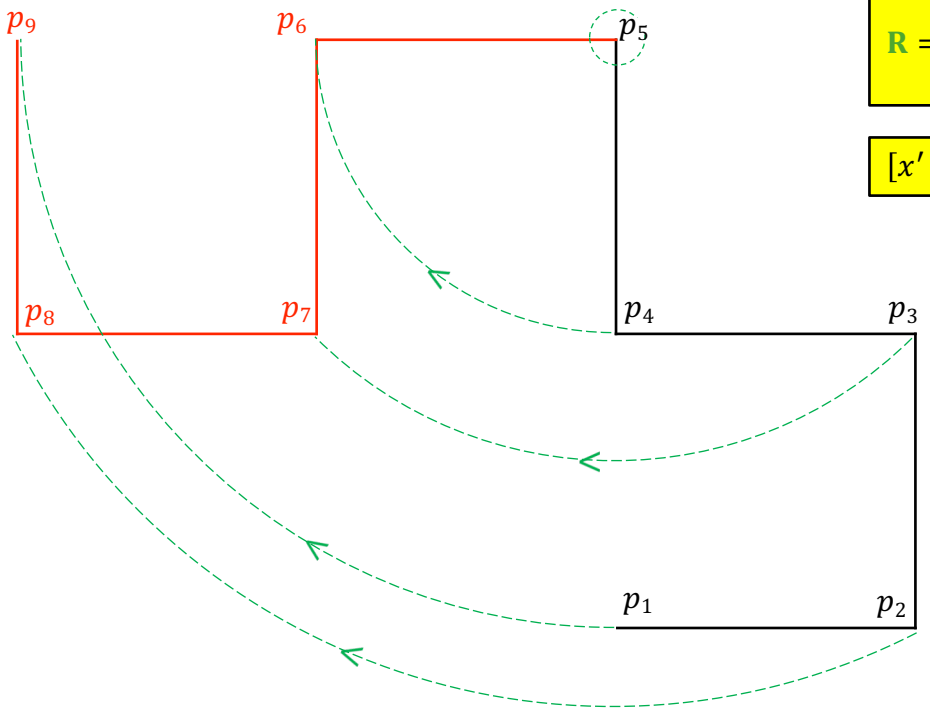
In **part 2**, we **computed** the **path** of a **dragon** by exploiting the **self-similarity** of **dragons**.

A **dragon path** is a **sequence** of **points**.

The **initial dragon path**, for a **dragon** aged **zero**, consists of **two points**, namely the **dragon**'s **starting point**, and the **point** reached from that **starting point** by travelling in the given **direction** by the given **line length**.

Given a **path** for a **dragon** aged **N**, the way we **grew** it into that of a **dragon** aged **N+1**, is by **adding** to the **path** the result of first **reversing** the tail of the **path**, and then **rotating** each of its **points** **90 degrees clockwise** about the **path**'s last **point**.

Here for example, we take the **path** of a **dragon** aged **2**, whose **points** are p_1, p_2, p_3, p_4, p_5, drop its last point, **reverse** the other points into **path** p_4, p_3, p_2, p_1, and then **rotate** each of the latter's **points** about p_5, resulting in new **path** section p_6, p_7, p_8, p_9.




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

```
type DragonPath = List[Point]
```

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

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

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

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

The way that we **rotate** a **point**, is by putting its x and y **coordinates** in a **row vector** with a z value of 1, and computing the **dot product** of the **vector** with **rotation matrix** **R**.

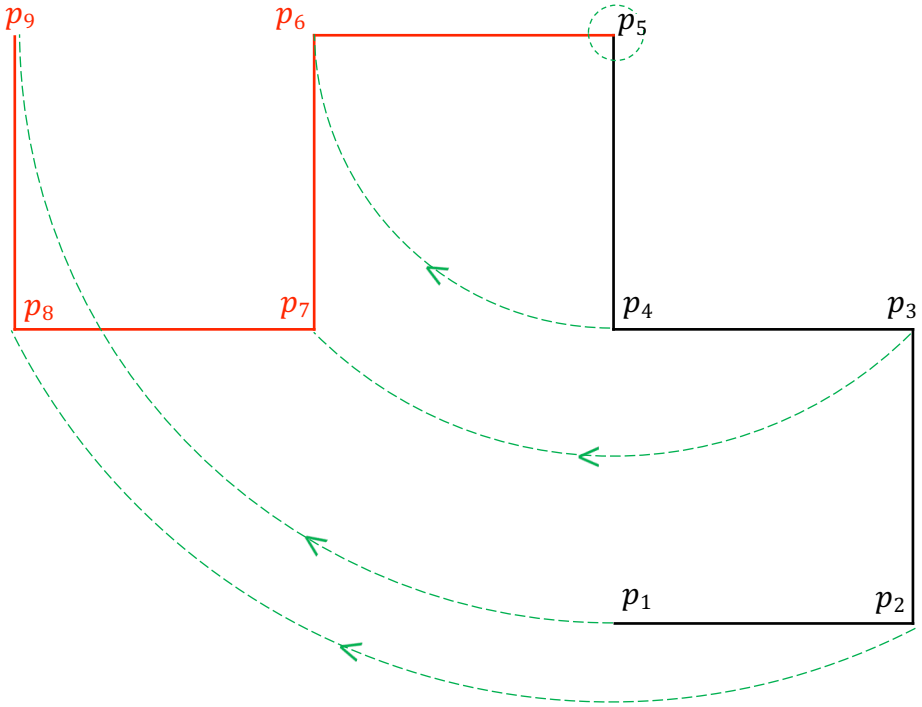
I now realise that we can **greatly simplify** the task of **rotating** part of a **dragon**, if instead of rotating the **dragon**'s **path**, we rotate its **shape**.

As mentioned on the previous slide, the **path** of a **dragon** consists of a sequence of **points**.

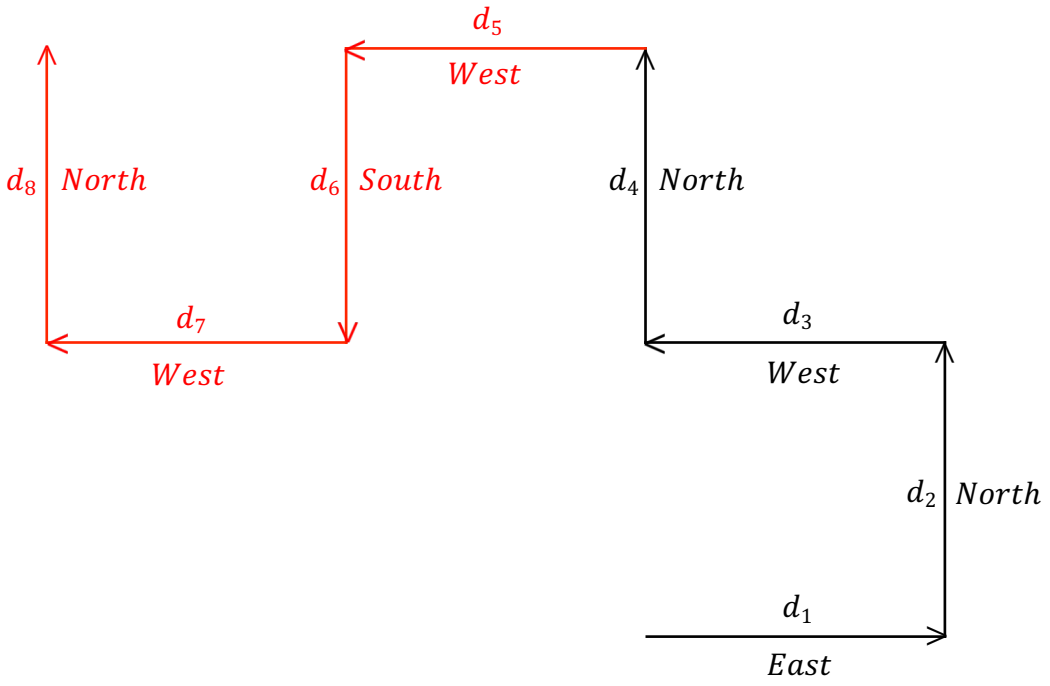
What about the **shape** of a **dragon**? Let's define it to be a sequence of **directions**.

The **diagram** on the left depicts the **rotation** of a **path**. **Points** p_6, p_7, p_8, p_9 are computed by **rotating points** p_1, p_2, p_3, p_4.

The **diagram** on the right depicts the **rotation** of the equivalent **shape**. On the next slide we look at how we can compute directions d_5, d_6, d_7, d_8.



Dragon Path

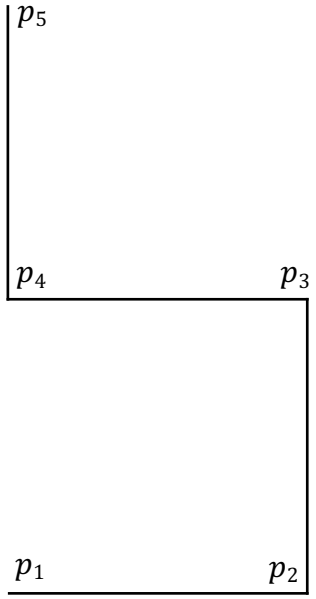


Dragon Shape

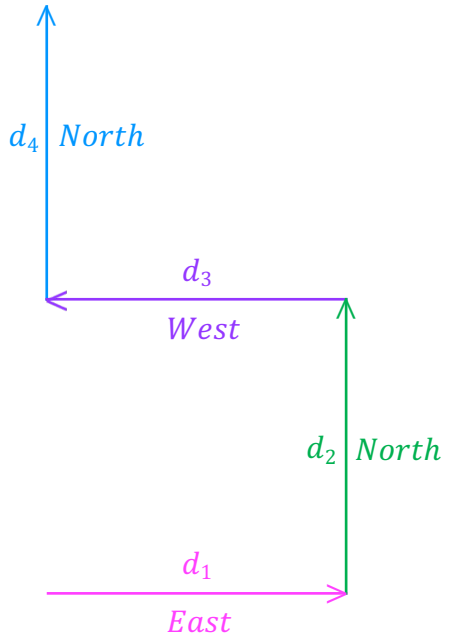
The way we rotate **path** p_1, p_2, p_3, p_4, p_5 about p_5 is by dropping p_5, **reversing** the rest of the points into **path** p_4, p_3, p_2, p_1, and **rotating** each **point** in turn.

The way we will **rotate shape** d_1, d_2, d_3, d_4 is by **reversing** the **shape** into d_4, d_3, d_2, d_1 and then **rotating** each **direction** in turn.

See next slide for how we will **rotate** a **direction**.



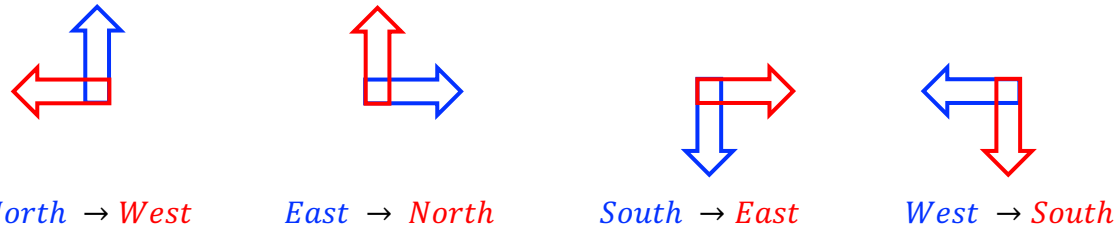
Dragon Path

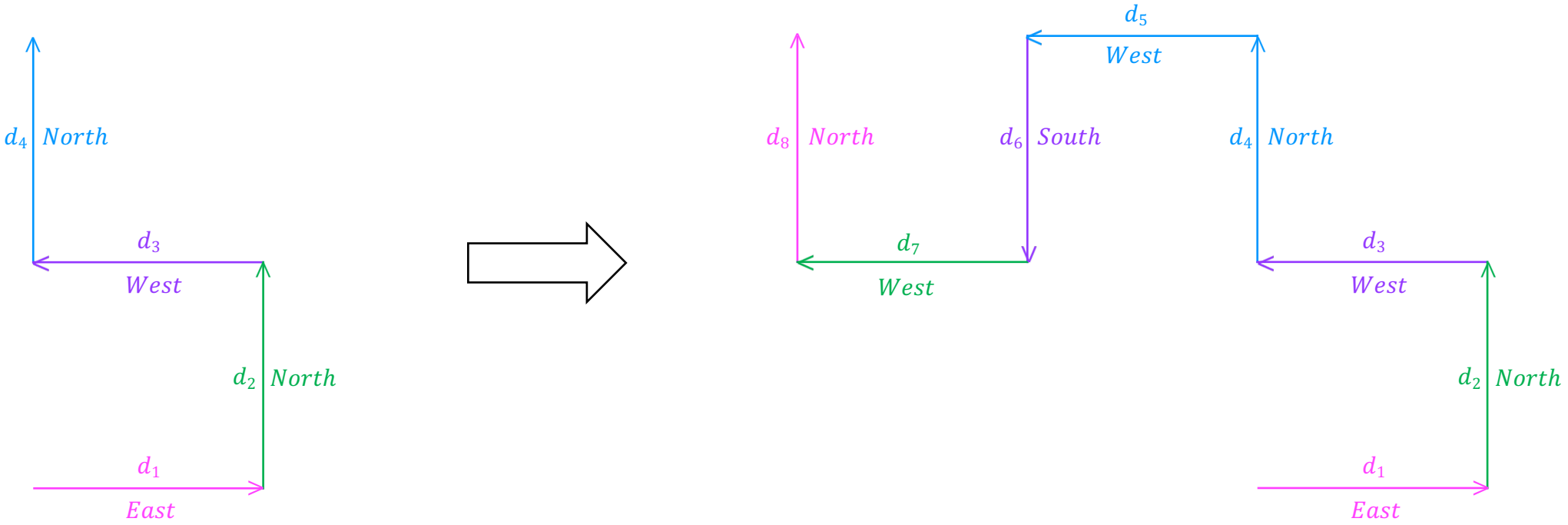


Dragon Shape

As shown on the right, we need to **rotate** each **direction** anticlockwise by 90°.

See below for the result of **copying** the **shape** d_1, d_2, d_3, d_4 of a **dragon** aged **2**, **reversing** it into d_4, d_3, d_2, d_1, **rotating** the latter's **directions**, which results in d_5, d_6, d_7, d_8, and appending the latter to the end of the **shape**.





growing a Dragon Shape

Let’s see how we need to change our **functional core logic** in order to adopt the **new approach** to **rotation**.

The first thing we need to do is change how we compute the **path** of a **dragon**.

Here is how we have been doing it so far:

1. create an **initial dragon path**
2. **grow** the **path**

And here is how we want to do it now:

1. create an **initial dragon shape**
2. **grow** the **shape**
3. turn the **shape** into a **path**

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

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

In the original approach, we first create an initial path, and then grow it, which requires us to be able to rotate a path.

In the new approach, there is no need to grow a path, and and so there is no need to rotate a path either.

Also, rather than creating an intial path, we will be transforming a shape into a path, so the responsibility for creating a path will belong to a shape.

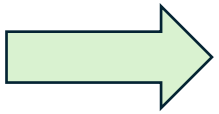
```
type DragonPath = List[Point]

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
```

```
type DragonPath = List[Point]

extension (path: DragonPath)

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

These two responsibilities are moving to new domain entity DragonShape.

Since there is no longer any need to **rotate** a **path**, which amounts to **rotating** its **points**, we can **delete** the **logic** dedicated to **rotating paths** and **points**.

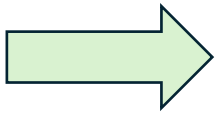
```scala
case class Point(x: Float, y: Float)

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

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)
```

Before we look at the new **DragonShape**, let's add to **Direction** the logic required to **rotate** a **direction**.

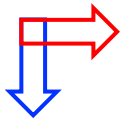
```
enum Direction:
  case North, East, South, West
```



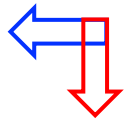
```
enum Direction:
  case North, East, South, West

  def rotated: Direction = this match
    case Direction.North => Direction.West
    case Direction.East  => Direction.North
    case Direction.South => Direction.East
    case Direction.West  => Direction.South
```

North → *West*

East → *North*



South → *East*



West → *South*

Here is new **domain entity DragonShape**.

```
type DragonShape = List[Direction]

object DragonShape:

  def initial(startDirection: Direction): DragonShape =
    List(startDirection)

extension (shape: DragonShape)

  @tailrec
  def grow(age: Int): DragonShape =
    if age == 0 then shape
    else shape.plusRotatedCopy.grow(age - 1)

  private def plusRotatedCopy: DragonShape =
    shape ++ shape.reverse.map(_.rotated)

  def path(startPoint: Point, length: Int): DragonPath =
    shape.foldLeft(List(startPoint)):
      (path, direction) => path.head.translate(direction, length) :: path
```

As mentioned earlier, the **initial dragon shape** consists simply of the **direction** of the **dragon**'s first **line**.

Apart from the **return type**, this is the **same function** that was provided by `DragonPath`.

This **function** is now a bit **simpler** that the analogous one that was provided by `DragonPath`.

To **rotate** a **shape**, we **reverse** it and then **rotate** all of its **directions**. In fact I think we should **extract** such **logic** into a **shape rotation function**, and **inline** this **function**. See next slide for the result.

No surprises here: **transforming** a **sequence** of **directions** and a **starting point** into a **sequence** of **points** is just creating a **second point** by **translating** the **first point**, then creating a **third point** by translating the **second point**, and so on.

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

Here is DragonShape again, after refactoring it by extracting the rotated function from plusRotatedCopy, and then inlining the latter.

```
type DragonShape = List[Direction]

object DragonShape:

  def initial(startDirection: Direction): DragonShape =
    List(startDirection)

extension (shape: DragonShape)

  @tailrec
  def grow(age: Int): DragonShape =
    if age == 0 then shape
    else (shape ++ shape.rotated).grow(age - 1)

  private def rotated: DragonShape =
    shape.reverse.map(_.rotated)

  def path(startPoint: Point, length: Int): DragonPath =
    shape.foldLeft(List(startPoint)):
      (path, direction) => path.head.translate(direction, length) :: path
```

What we have done in this deck is **move** the **process** of **growing** a **dragon**, which requires **rotating** it, from the **stage** in which the **dragon** is **represented** as a **dragon path**, i.e. a sequence of **points**, to a new, **earlier stage**, in which the **dragon** is **represented** as a **dragon shape**, i.e. a sequence of **cardinal directions**.

The **benefit** of doing so is that while **rotating** a **dragon path** involves **rotating points**, which in turn requires **matrix multiplication**, and which is therefore **relatively complex** and **time-consuming**, **rotating** a **dragon shape** involves **rotating cardinal directions**, which is a **trivial** and thus **inexpensive operation**.

While before the **two concerns** of **rotating** a **dragon**'s **shape** and **computing** a **dragon**'s **path** were **intertwined** (we did both at the same time), we have now **separated** the **two concerns** from each other.

The way the **program** that **rotates dragon paths** (part 2) improves on the **original program** (part 1), which was just a **functional version** of the **imperative Pascal program**, is that it makes it **easier to understand** how a **dragon** is **grown**, i.e. by **duplicating** and **rotating** the **dragon**.

As mentioned above, the way the **program** that **rotates dragon shapes** (part 3) improves on the **program** that **rotates dragon paths** (part 2), is that by **simplifying** the rotation **process**, it makes **growing** a **dragon simpler**, **faster**, and **easier to understand**.

In the **first program** (part 1), it was **hard to understand** how a **dragon** is **grown**. While in the **second program** (part 2), it was **easy to understand** the **growth process**, the **process** relied and some **non-trivial rotation-related mathematics**. The **third program** (part 3), which **eliminates** the **need** for such **mathematics**, makes it even **simpler to understand** how a **dragon** is **grown**.

The next slide, shows the **functional core logic** of the **first program** (part 1), and the slide after that shows the **functional core logic** of the **third program** (part 3). On both slides, **logic** that is **the same** in both **programs** is shown with a gray background.

The slide after that compares the **core elements** of the **two programs**.

The **functional core** of the program's **first version** (from part 1)

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

```scala
type DragonPath = List[Point]

object DragonPath:
  def apply(start: Point): DragonPath = List(start)

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)

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

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

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

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

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

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)
```

The **functional core** of the program's **third version** (from this deck – part 3)

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

```scala
type DragonPath = List[Point]

extension (path: DragonPath)

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

```scala
type DragonShape = List[Direction]

object DragonShape:

  def initial(startDirection: Direction): DragonShape =
    List(startDirection)

extension (shape: DragonShape)

  @tailrec
  def grow(age: Int): DragonShape =
    if age == 0 then shape
    else (shape ++ shape.rotated).grow(age - 1)

  private def rotated: DragonShape =
    shape.reverse.map(_.rotated)

  def path(startPoint: Point, length: Int): DragonPath =
    shape.foldLeft(List(startPoint)):
      (path, direction) => path.head.translate(direction, length) :: path
```

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

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

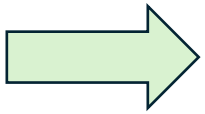
```scala
enum Direction:
  case North, East, South, West

  def rotated: Direction = this match
    case Direction.North => Direction.West
    case Direction.East  => Direction.North
    case Direction.South => Direction.East
    case Direction.West  => Direction.South
```

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

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)
```

How a dragon is grown

In the program's **first version** (from part 1)

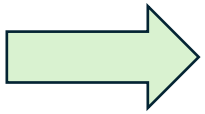
```
extension (path: DragonPath)

  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)
```



In the program's **third version** (from this deck)

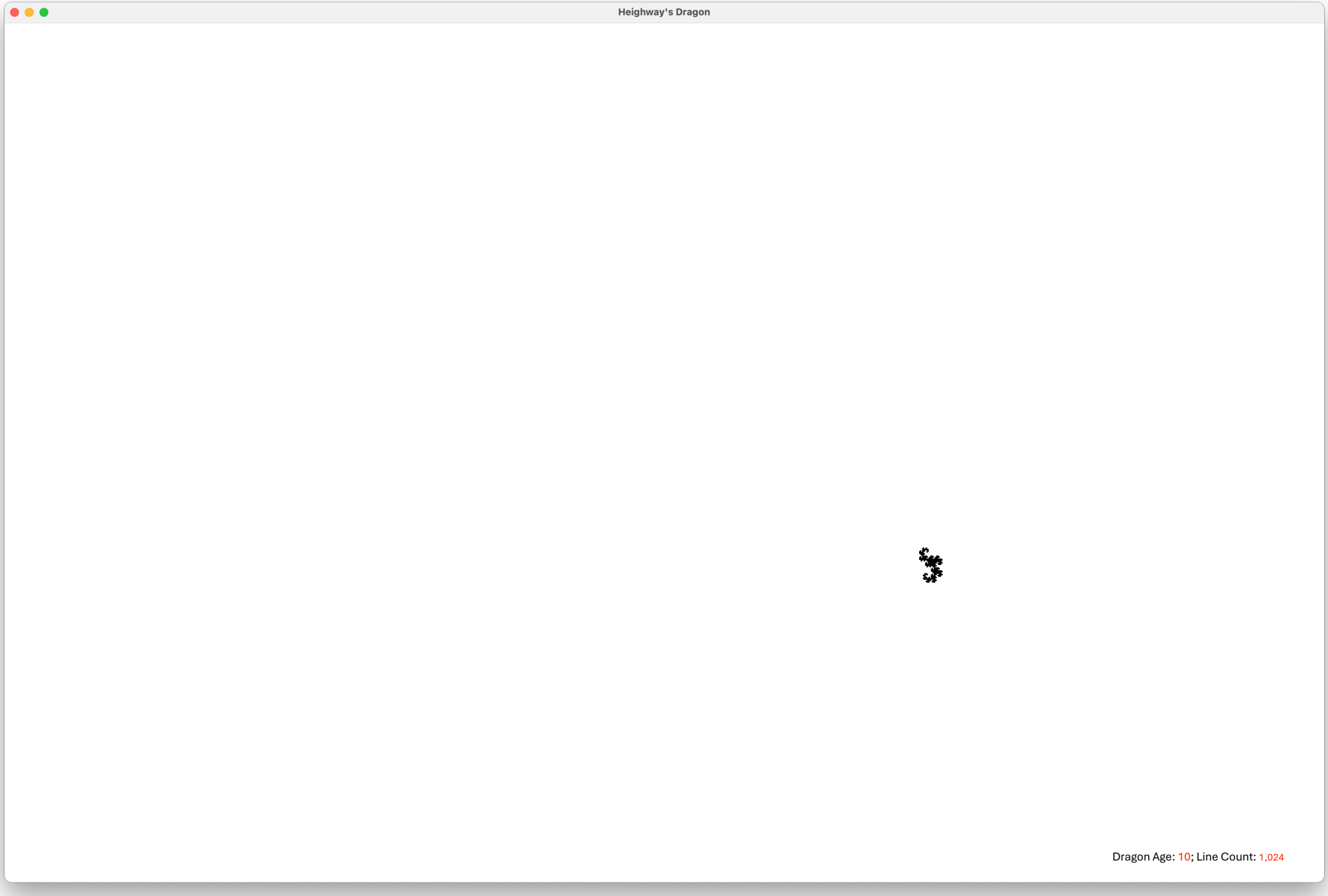
```
extension (shape: DragonShape)

  @tailrec def grow(age: Int): DragonShape =
    if age == 0 then shape
    else (shape ++ shape.rotated).grow(age - 1)

  private def rotated: DragonShape =
    shape.reverse.map(_.rotated)
```

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



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

  def rotated: Direction = this match
    case Direction.North => Direction.West
    case Direction.East  => Direction.North
    case Direction.South => Direction.East
    case Direction.West  => Direction.South
```

To conclude this **deck**, the next **10 slides** demo the **program**'s **output** for a **starting direction** of **East**, a **line length** of **1**, and **ages 10** through to **20**.


Heighway's Dragon
Dragon Age: 10; Line Count: 1,024

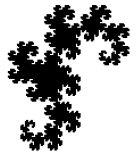
Heighway's Dragon

Dragon Age: 11; Line Count: 2,048

Heighway's Dragon

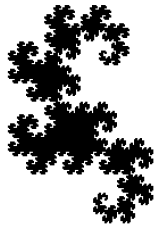
Dragon Age: 12; Line Count: 4,096

Heighway's Dragon



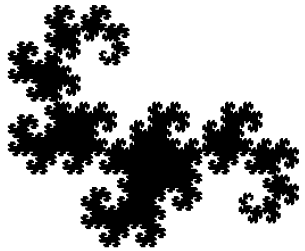
Dragon Age: 13; Line Count: 8,192

Heighway's Dragon



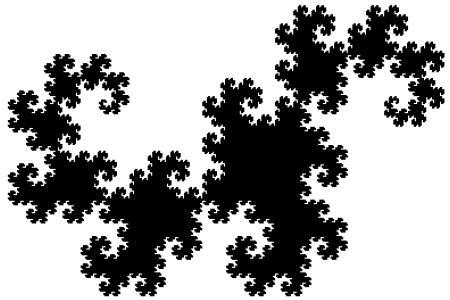
Dragon Age: 14; Line Count: 16,384

Heighway's Dragon



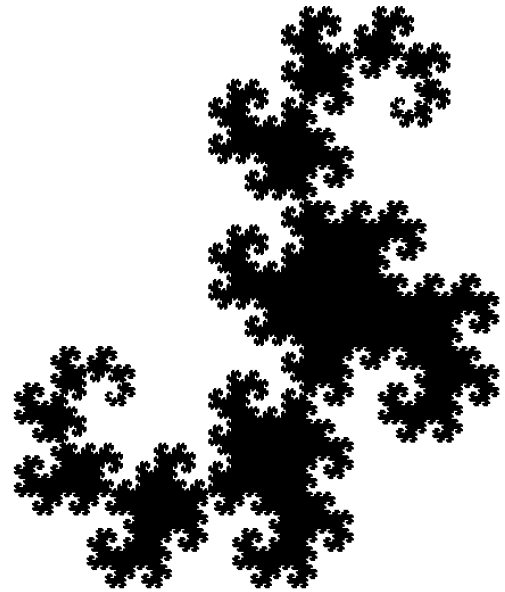
Dragon Age: 15; Line Count: 32,768

Heighway's Dragon



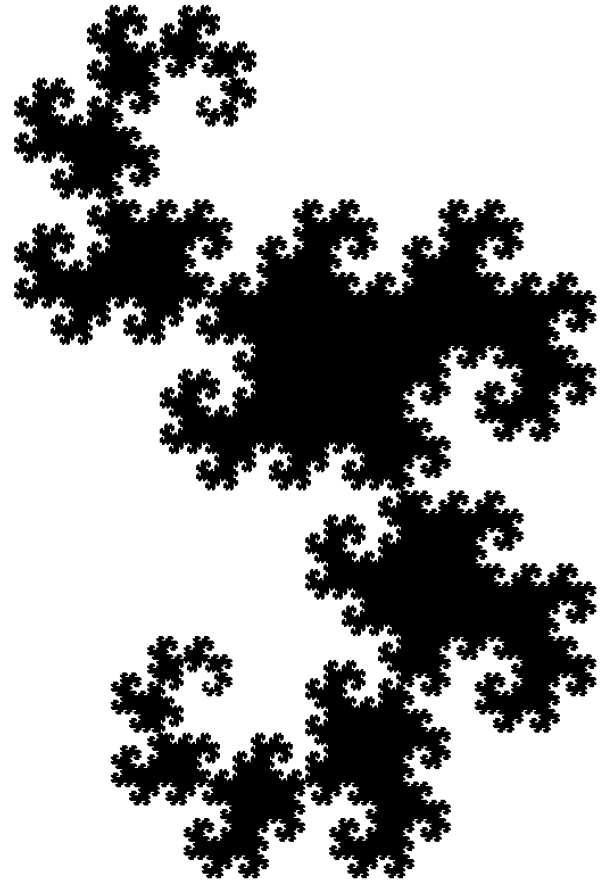
Dragon Age: 16; Line Count: 65,536

Heighway's Dragon



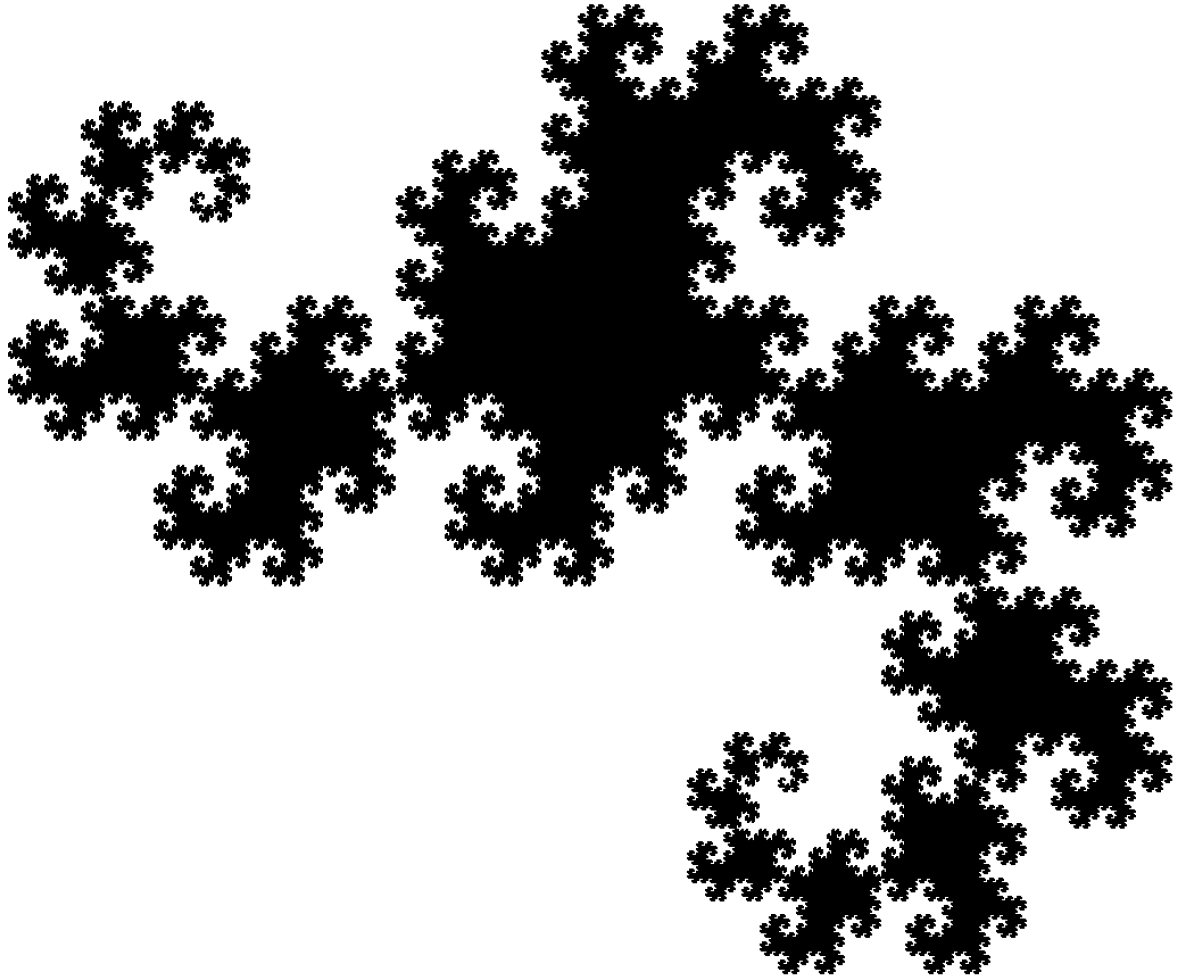
Dragon Age: 17; Line Count: 131,072

Heighway's Dragon

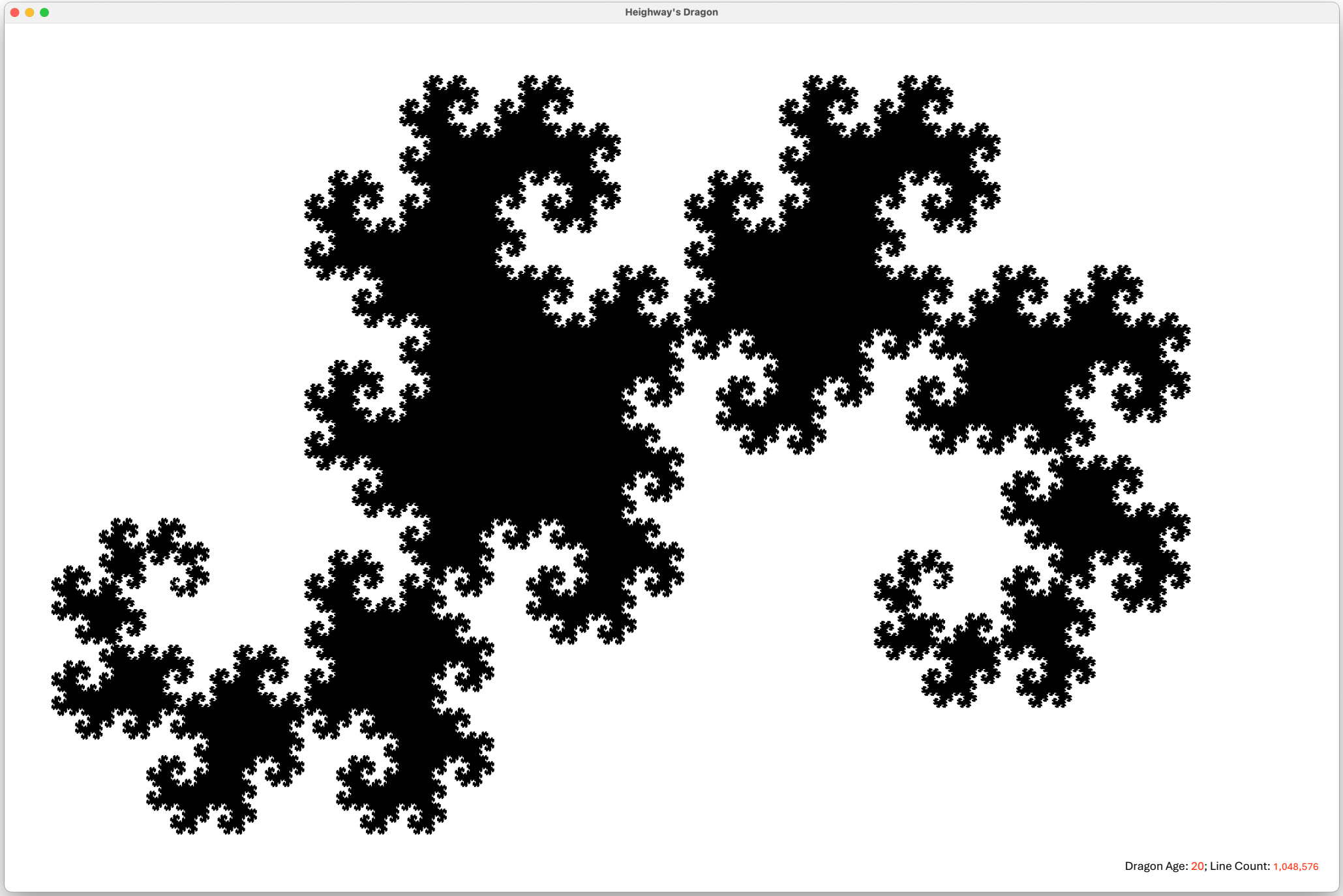


Dragon Age: 18; Line Count: 262,144

Heighway's Dragon



Dragon Age: 19; Line Count: 524,288


Heighway's Dragon
Dragon Age: 20; Line Count: 1,048,576

That's all for **part 3**.

I hope you enjoyed that.

At the end of **part 1** I said that in **part 2** we were going to make the **program** much **more convenient** in that it would allow us to **easily change dragon parameters** and **redraw** the **dragon** each time without having to **rerun** the **program**.

That never happened, so let's do it in **part 4**.

See you there.