Sierpinski's Triangle

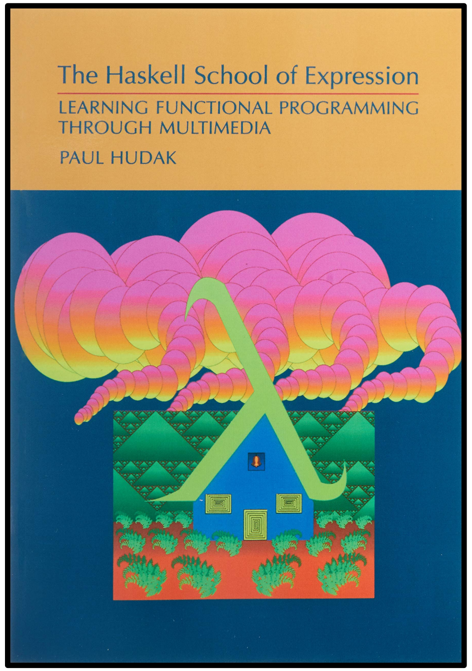
Polyglot FP for Fun and Profit
Haskell and Scala

Take the very first baby steps on the path to doing **graphics** in **Haskell** and **Scala**

Learn about a simple yet educational **recursive algorithm** producing images that are pleasing to the eye

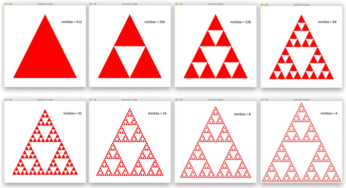
Learn how **functional programs** deal with the **side effects** required to draw images

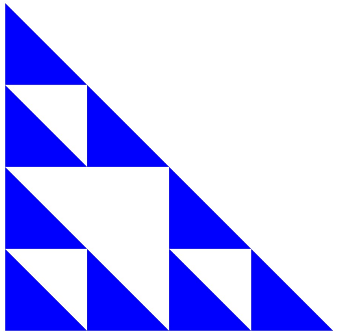
See how libraries like **Gloss** and **Doodle** make drawing **Sierpinski's triangle** a doddle



inspired by, and based on, the work of

Paul E. Hudak





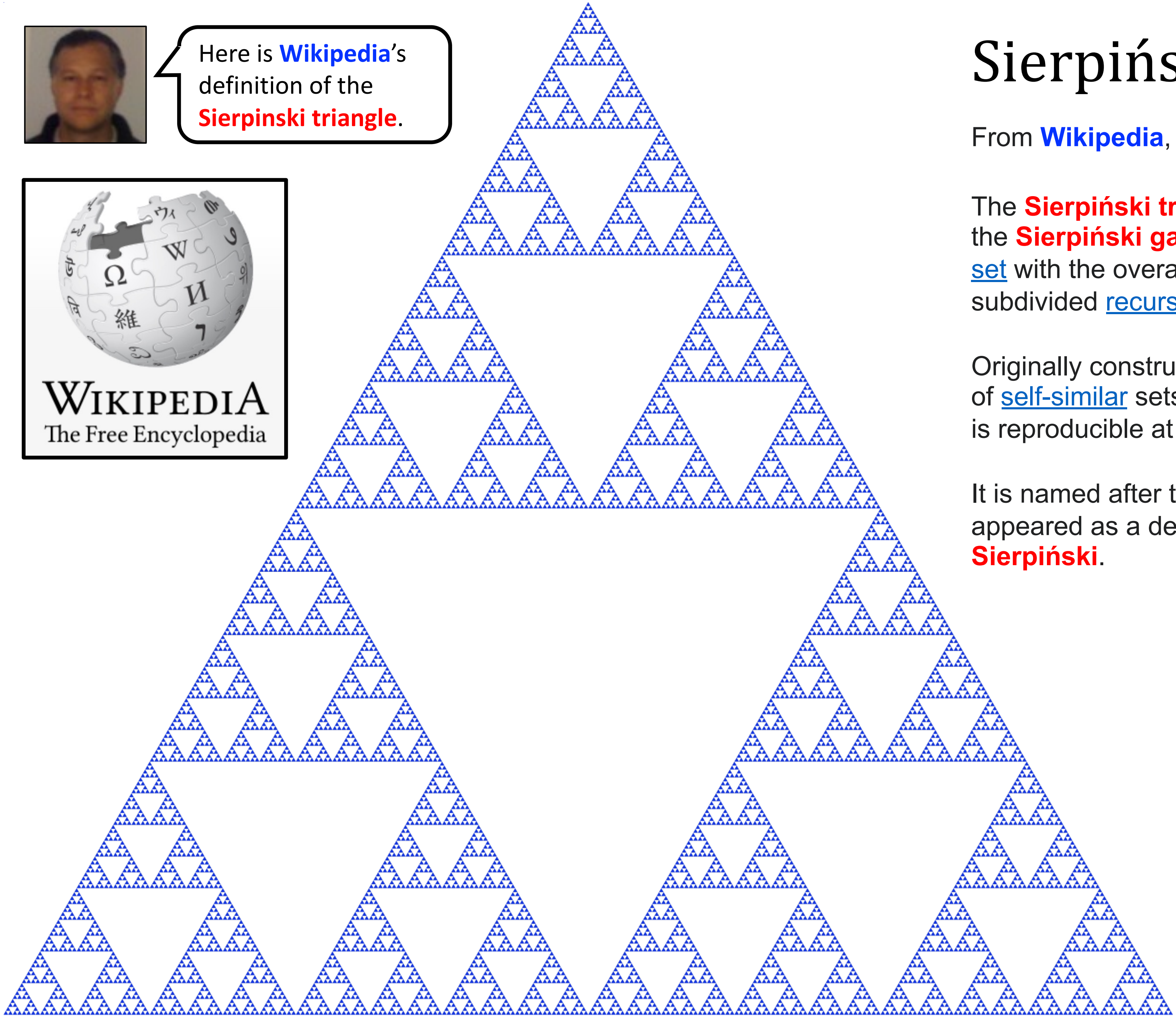
Cats Effect

Haskell Gloss

Scala Doodle

slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

Here is **Wikipedia**'s definition of the **Sierpinski triangle**.



Sierpiński triangle

From **Wikipedia**, the free encyclopedia

The **Sierpiński triangle** (sometimes spelled ***Sierpinski***), also called the **Sierpiński gasket** or **Sierpiński sieve**, is a fractal attractive fixed set with the overall shape of an equilateral triangle, subdivided recursively into smaller equilateral triangles.

Originally constructed as a curve, this is one of the basic examples of self-similar sets—that is, it is a mathematically generated pattern that is reproducible at any magnification or reduction.

It is named after the Polish mathematician Wacław Sierpiński, but appeared as a decorative pattern many centuries before the work of **Sierpiński**.

@philip_schwarz

The first thing we are going to do is look at a **Haskell** program that draws **Sierpinski's triangle**.

The program is presented by **Paul Hudak** in chapter 3 of his book titled **The Haskell School of Expression**. The chapter in question is called **Simple Graphics**.

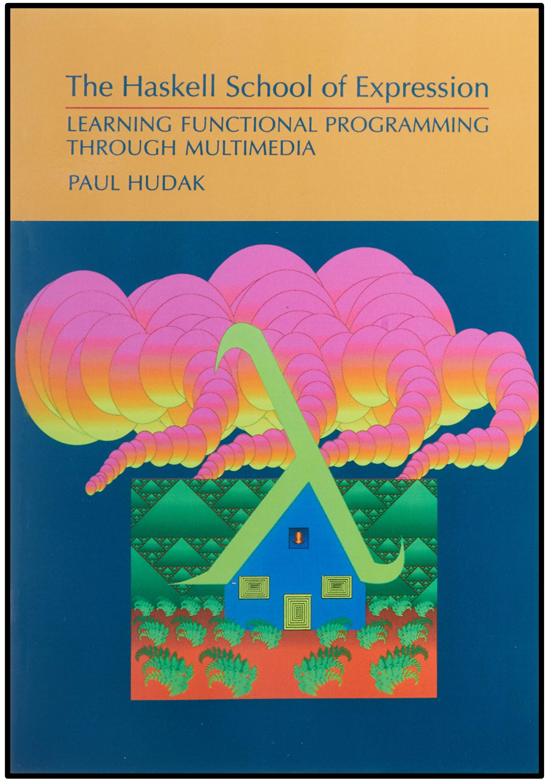
The program is presented in section **3.4 Some Examples**. In preceding sections **3.1** to **3.3**, the author gives the reader an introduction to how to do graphics in **Haskell**:

> **3.1 Basic Input/Output**
> While this section is topical, interesting and useful, I think it is best, for our purposes, to tackle it in a second stage, once we have familiarised ourselves with the logic for drawing the **Sierpinski triangle**. Let's skip the section for now and come back to it later.
>
> **3.2 Graphics Windows**
> If you are a programmer and you have already done any non-zero amount of graphics, you'll be able to understand the graphics-specific aspects of the program without reading this section, so let's skip it.
>
> **3.3 Drawing Graphics Other Than Text**
> All we need from this section are a couple of lines of code.

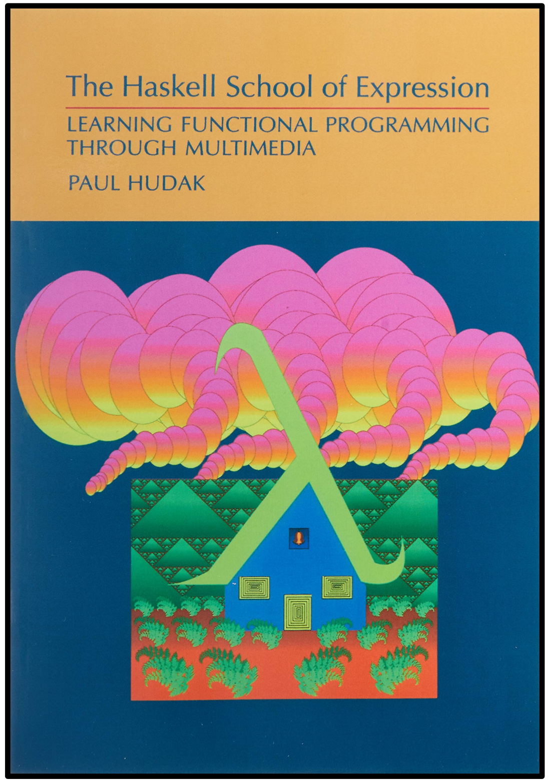
So here is the plan:

1. dive right into the **Haskell** code for **Sierpinski's triangle**
2. have a first go at writing an equivalent **Scala** program
3. look at the book's section on **Basic Input/Output** in **Haskell**
4. have a go at using **Cats Effect** to adapt the **Scala** program so that it manages **side effects** the same way the **Haskell** program does.

That's not all though: we'll be doing more after that!



Paul E. Hudak



There are some simple **fractal images** that are pleasing to the eye yet very easy to describe and draw.

One such image is called **Sierpinski's Triangle**, which can be described via successive drawings of a triangle.

The **first drawing** is a single triangle.

The **second drawing** subdivides the first triangle into three triangles, each one-half the original in both length and height.

The **third drawing** subdivides each of the triangles in the **second drawing** in a similar way.

Now imagine this **ad infinitum**, and there you have **Sierpinski's Triangle**.

Of course, we cannot actually show this infinitely-dense triangle in a graphics window, because we are limited by pixel size (and our eyes would not be sharp enough to see the details).

So to draw **Sierpinski's Triangle** we will stop subdividing the triangles when we reach some **predetermined image size**, and then just draw each tiny triangle completely at that point.

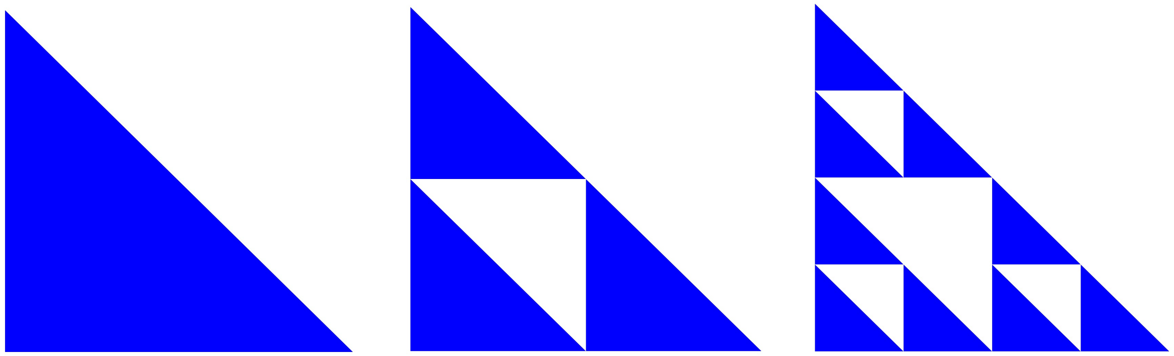
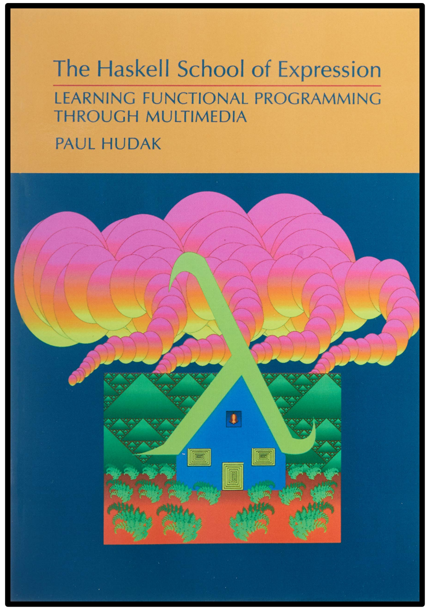


Figure 3.1: First three constructions of **Sierpinski's Triangle**

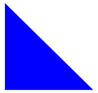
First I will define a function **fillTri** that draws a blue-filled triangle, given x and y coordinates and a size (all in pixel coordinates):

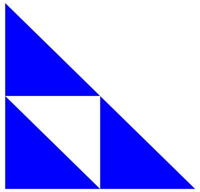
```
import SOEGraphics

fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size = …
```



The rest of the algorithm is relly very easy (and elegant), and is presented in one fell swoop:

```
minSize :: Int
minSize = 8

sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in do sierpinskiTri w x y size2
        sierpinskiTri w x (y - size2) size2
        sierpinskiTri w (x + size2) y size2
```

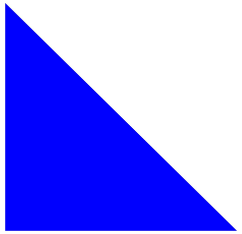




Note the recursive calls to **sierpinskiTri**; when the size drops to 8 or less, **fillTri** is called instead.

Here is the implementation of `fillTri`:

```
fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size =
  drawInWindow w
    (withColor Blue
      (polygon [(x,y),(x + size,y),(x,y - size)]))
```



And here are the `SOEGraphics` functions and data that `fillTri` depends on:

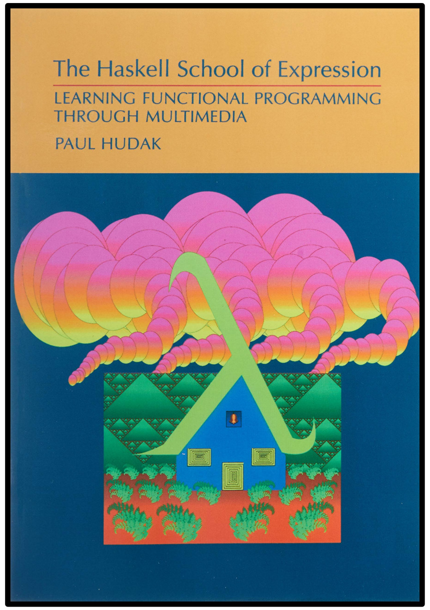
```
polygon :: [Point] -> Graphic
polygon pts :: =
```
Draws a closed polygon with vertices `pts`.
The last point is connected back to the first, and thus the polygon can be filled with a color.

```
type Point = (Int,Int)
```

```
withColor :: Color -> Graphic -> Graphic
withColor c g =
```
Changes the color of drawings in a `Graphic` – the default colour is white.

```
data Color = Black | Blue | Green | Cyan | Red | Magenta | Yellow | White
```

```
drawInWindow :: Window -> Graphic -> IO ()
drawInWindow w g =
```
Draws a given `Graphic` value on a given `Window`.



Using **sierpinskiTri** is easy enough; The only trickery is to use a number that is a power of two for the initial size, to make the subdivisions look most uniform by avoiding rounding errors.

```
main :: IO ()
main =
  runGraphics(
    do w <- openWindow "Sierpinski's Triangle" (400,400)
       sierpinskiTri w 50 300 256
  )
```

And here are the **SOEGraphics** functions and data that **main** depends on:

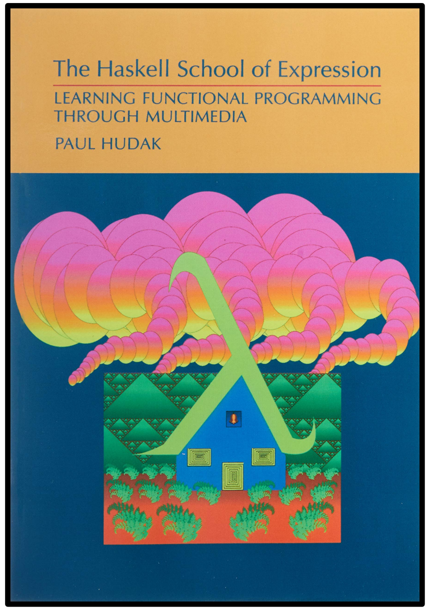
```
runGraphics :: IO() -> IO()
```

`runGraphics action =` Runs a graphics "action". This is needed because of special operating system tasks that need to be set up to perform graphics IO.

```
openWindow :: Title -> Size -> IO Window
```

`openWindow title size =` Creates a new, unique window
- `title:` the string displayed in the title bar of the graphics window
- `size:` the size of the window, i.e. it width and its height

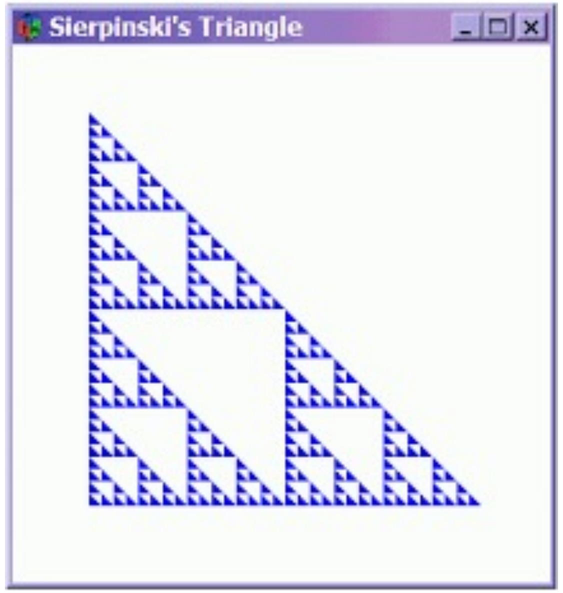
```
type Title = String
type Size = (Int, Int)
```



Here is the entirety of the **Haskell** program that we have just seen.

```haskell
import SOEGraphics

  minSize :: Int
  minSize = 8

  fillTri :: Window -> Int -> Int -> Int -> IO ()
  fillTri w x y size =
    drawInWindow w
      (withColor Blue
        (polygon [(x,y),(x + size,y),(x,y - size)]))

  sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
  sierpinskiTri w x y size =
    if size <= minSize
    then fillTri w x y size
    else let size2 = size `div` 2
    in do sierpinskiTri w x y size2
          sierpinskiTri w x (y - size2) size2
          sierpinskiTri w (x + size2) y size2

  main :: IO ()
  main =
    runGraphics(
      do w <- openWindow "Sierpinski's Triangle" (400,400)
         sierpinskiTri w 50 300 256
    )
```

Sadly, the book contains only a single figure with a sample result of running the program.



In upcoming slides we'll run the program ourselves, so we can do it more justice, by generating several **Sierpinski** triangles.

@philip_schwarz

Any program that displays some kind of result to the user does so by causing **side effects**. In our case, instead of printing something to the screen or to a file, the program draws triangles on the screen.

In a program that does not use **functional programming**, any and all functions are allowed to cause **side effects**.

As an example, let's take **Haskell** functions `sierpinskiTri` and `fillTri` and rewrite them in **Scala** *without* using **functional programming**.

```haskell
import SOEGraphics

sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in do sierpinskiTri w x y size2
        sierpinskiTri w x (y - size2) size2
        sierpinskiTri w (x + size2) y size2

fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size =
  drawInWindow w
    (withColor Blue
      (polygon [(x,y),(x + size,y),(x,y - size)]))
```

```scala
import java.awt.{Color, Graphics}

def sierpinskiTri(g:Graphics, x: Int, y: Int, size: Int): Unit =
  if size <= minSize
  then fillTri(g, x, y, size)
  else
    val halfSize = size / 2
    sierpinskiTri(g, x, y, halfSize)
    sierpinskiTri(g, x, y - halfSize, halfSize)
    sierpinskiTri(g, x + halfSize, y, halfSize)

def fillTri(g:Graphics, x: Int, y: Int, size: Int): Unit =
  val xs = Array(x, x + size, x)
  val ys = Array(y, y, y - size)
  g.setColor(Color.blue)
  g.fillPolygon(xs, ys, 3)
```

The **Scala** version of the `fillTri` function is **side-effecting**. When it executes `g.fillPolygon(xs, ys, 3)`, a triangle gets drawn on the screen as a **side effect**. We can tell that the function is performing **side effects** because its return type is **Unit**: since it is not returning anything of value, it must be performing one or more **side effects** that are of value. Even if a function does return something of value, it can still perform side effects. In a **Scala** program that does not use **functional programming**, any and all functions may perform **side effects**.

We just discussed the Scala version of the fillTri function and said that it is side-effecting. What about the Haskell version?

```
fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size =
  drawInWindow w
    (withColor Blue
      (polygon [(x,y),(x + size,y),(x,y - size)]))
```

```
polygon :: [Point] -> Graphic
withColor :: Color -> Graphic -> Graphic
```

```
drawInWindow :: Window -> Graphic -> IO()
```

The polygon and withColor functions that it invokes do not perform any side effects: they both return a value of type Graphic.

The fillTri function takes the Graphic value returned by withColor and passes it to the drawInWindow function, returning whatever it returns. While the drawInWindow function does not perform any side effects, it returns a value that describes the performing of a side effect.

The return value of drawInWindow has type IO () and is called an IO action, because at some suitable later point in the execution of the program, it will be executed, and at that point it will produce a side effect.

Rather than being side-effecting, i.e. producing a side effect, the drawInWindow function is said to be effectful: it returns a value representing an effect.

If the type of the returned value were that of a list, the effect would be that of multiplicity (the value would represent zero or more answers). If, instead, the type of the returned value were Maybe (Option in Scala), then the effect would be that of optionality (the value would be either present or absent).

The actual type of the returned value is IO (). The returned value is an action, a value describing the effect of performing a side effect.

The side effect is not performed at the time when the action is created, but rather at a later time, when the action is executed.

Now that we have discussed both the **Scala** version and the **Haskell** version of the **fillTri** function, let's discuss the **sierpinskiTri** function, starting with the **Scala** version.

Just like in the **fillTri** function, its return type is **Unit**: it does not return anything of value, so it must be performing some **side effect** that is of value. While it does not directly perform **side effects**, it does perform them indirectly. In the base case, it calls the **fillTri** function, which performs the **side effect** of drawing a triangle on the screen. In the recursive case, the function calls itself sequentially three times, so it indirectly performs the **side effects** performed by those three recursive invocations.

Let's contrast that with the behaviour of the **Haskell** **sierpinskiTri** function.

Just like the **fillTri** function, its return type is **IO ()**: it returns something of value, namely an **action** describing the performing of a **side effect**. In the **base case**, **sierpinskiTri** just returns the simple **action** returned by **fillTri**. In the **recursive case**, **sierpinskiTri** takes the three **actions** returned by the three **recursive invocations** of itself, and returns a **composite action** whose **side effect**, to be produced later, when the **composite action** is executed, consists of three **side effects** produced by **sequentially** executing the three **actions** returned by the **recursive calls**. The **composite action** is created by wrapping in a '**do**' the three **actions** returned by the **recursive calls**.

```haskell
sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in do sierpinskiTri w x y size2
        sierpinskiTri w x (y - size2) size2
        sierpinskiTri w (x + size2) y size2

fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size =
  drawInWindow w
    (withColor Blue
      (polygon [(x,y),(x + size,y),(x,y - size)]))
```

```scala
def sierpinskiTri(g:Graphics, x: Int, y: Int, size: Int): Unit =
  if size <= minSize
  then fillTri(g, x, y, size)
  else
    val halfSize = size / 2
    sierpinskiTri(g, x, y, halfSize)
    sierpinskiTri(g, x, y - halfSize, halfSize)
    sierpinskiTri(g, x + halfSize, y, halfSize)

def fillTri(g:Graphics, x: Int, y: Int, size: Int): Unit =
  val xs = Array(x, x + size, x)
  val ys = Array(y, y, y - size)
  g.setColor(Color.blue)
  g.fillPolygon(xs, ys, 3)
```

As we said in the previous slide, in the Haskell program, the side effects are not produced during the execution of `sierpinskiTri`. The side effects are produced when the action that is the value of `main` is executed.

```
main :: IO ()
main =
  runGraphics(
    do w <- openWindow "Sierpinski's Triangle" (400,400)
       sierpinskiTri w 50 300 256
  )
```

`runGraphics :: IO() -> IO()` - Runs a graphics “action”. This is needed because of special operating system tasks that need to be set up to perform graphics IO.

`openWindow :: Title -> Size -> IO Window` - Creates a new, unique window

The `openWindow` function returns a value of type `IO Window`, i.e. an action describing the performance of a side effect which produces a `Window` value.

The `main` action is the result of carrying out the following steps:

- taking the action returned by `openWindow` and the action returned by `sierpinskiTri` and combining them into a composite action.
- passing the composite action to `runGraphics`, which returns an enriched composite action describing both the side effects described by the composite action plus some additional side effects needed to perform graphics IO.

i.e. the `main` action is the action returned by the `runGraphics` function.

It is only when the `main` action is executed that any side effects are produced. i.e. the triangles get drawn only when the action that is the value of `main` gets executed.

In the past few slides, we took **Haskell** functions **sierpinskiTri** and **fillTri** and rewrote them in **Scala** *without* using **functional programming**. This slide reproduces the two functions and adds code so that we now have a complete **Scala** program that draws the same triangles as the **Haskell** program.

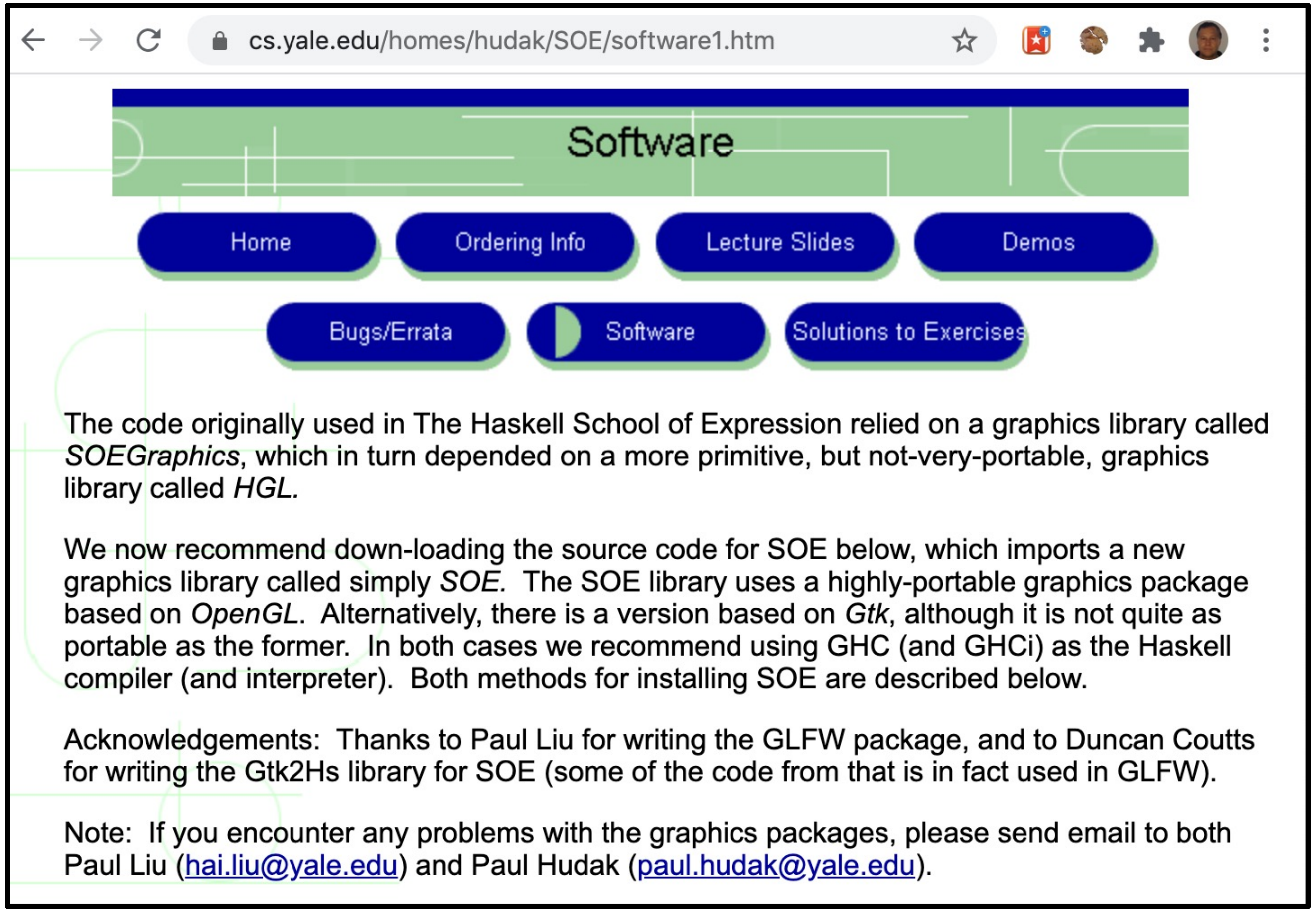
We saw that the way the **Haskell** program manages **side effects** is by using functions that create **actions**, i.e. **pure values** that merely describe **side effects**, and combining such **actions** into more complex **composite actions**, and eventually producing **side effects** by executing a topmost **composite action** that is the result of the whole program.

The **Scala** program is not written using **functional programming**: it manages **side effects** simply by allowing any and all functions to produce **side effects** on the spot, as part of their execution.

```scala
import java.awt.{Color, Graphics}
import javax.swing.{JPanel, JFrame}

@main def sierpinski: Unit =

  val title = "Sierpinski’s Triangle"
  val windowPosition = (0,0)
  val width = 600
  val height = 600
  val backgroundColour = Color.white
  val triangleColour = Color.blue
  val triangleSize = 512
  val triangleXPos = 50
  val triangleYPos = 550

  val minSize = 8

  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Sierpinski")
  frame.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE)
  frame.setBackground(backgroundColour);
  frame.setSize(width, height);

  val sierpinskiTriangle =
    SierpinskiJPanel(
      triangleXPos,
      triangleYPos,
      triangleSize,
      minSize,
      triangleColour
    )
  frame.add(sierpinskiTriangle);
  frame.setVisible(true)
```

```scala
class SierpinskiJPanel(x:Int, y:Int, size:Int, minSize:Int, colour:Color) extends JPanel:

  override def paintComponent(g: Graphics): Unit =
    sierpinskiTriangle(g)

  def sierpinskiTriangle(g: Graphics): Unit =
    g.setColor(colour)
    sierpinskiTriangle(g, x, y, size)

  def sierpinskiTriangle(g: Graphics, x: Int, y: Int, size: Int): Unit =
    if size <= minSize
    then fillTriangle(g, x, y, size)
    else
      val halfSize = size / 2
      sierpinskiTriangle(g, x, y, halfSize)
      sierpinskiTriangle(g, x, y - halfSize, halfSize)
      sierpinskiTriangle(g, x + halfSize, y, halfSize)

  def fillTriangle(g: Graphics, x: Int, y: Int, size: Int): Unit =
    val xs = Array(x, x + size, x)
    val ys = Array(y, y, y - size)
    g.fillPolygon(xs, ys, 3)
```

Earlier on I said that we planned to run the **Haskell** program ourselves, so that we could do it more justice, by generating several triangles. Alas, it turns out that running the program would require a bit of Yak shaving.


cs.yale.edu/homes/hudak/SOE/software1.htm

Software

Home | Ordering Info | Lecture Slides | Demos

Bugs/Errata | Software | Solutions to Exercises

The code originally used in The Haskell School of Expression relied on a graphics library called *SOEGraphics*, which in turn depended on a more primitive, but not-very-portable, graphics library called *HGL.*

We now recommend down-loading the source code for SOE below, which imports a new graphics library called simply *SOE.* The SOE library uses a highly-portable graphics package based on *OpenGL*. Alternatively, there is a version based on *Gtk*, although it is not quite as portable as the former. In both cases we recommend using GHC (and GHCi) as the Haskell compiler (and interpreter). Both methods for installing SOE are described below.

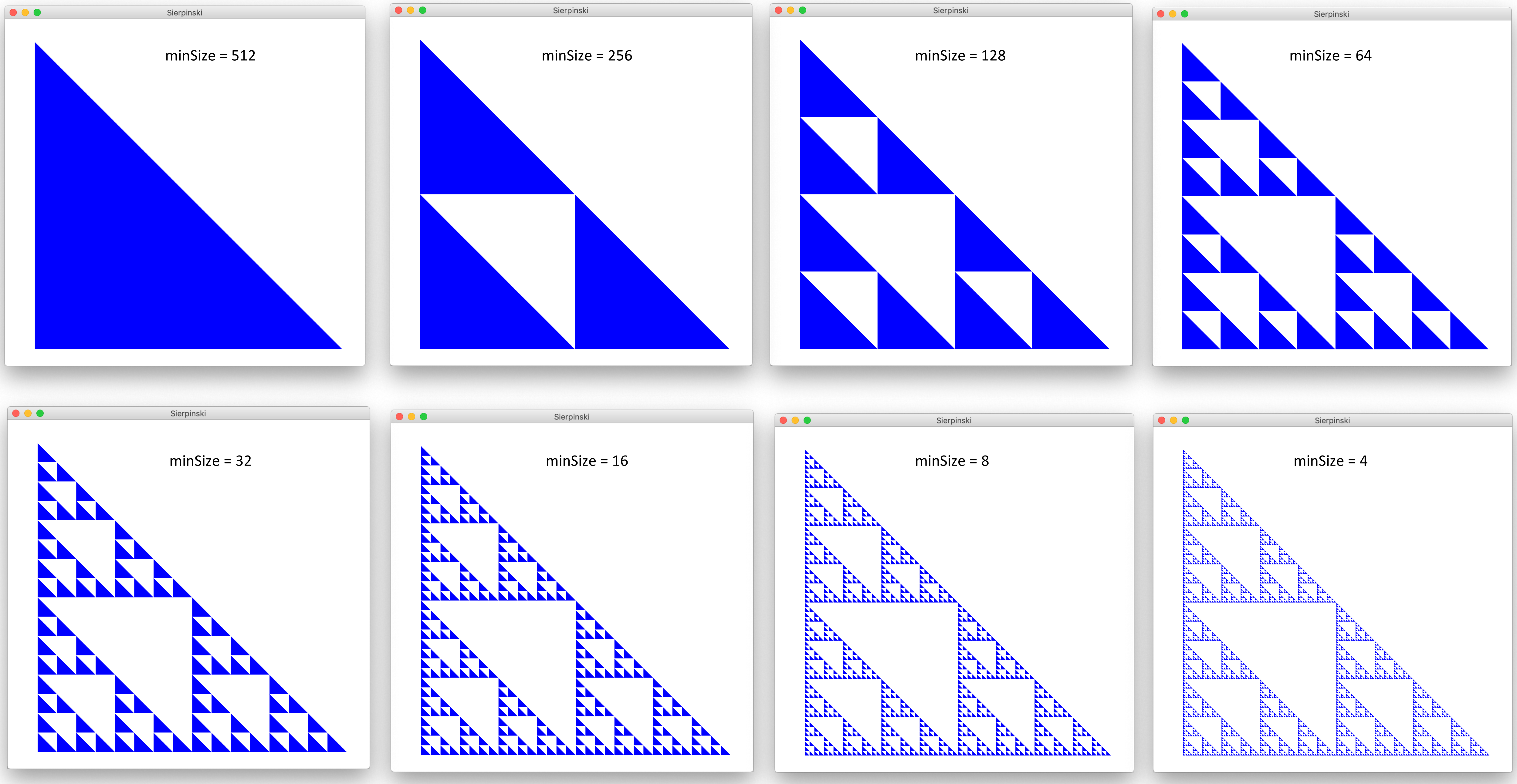
Acknowledgements: Thanks to Paul Liu for writing the GLFW package, and to Duncan Coutts for writing the Gtk2Hs library for SOE (some of the code from that is in fact used in GLFW).

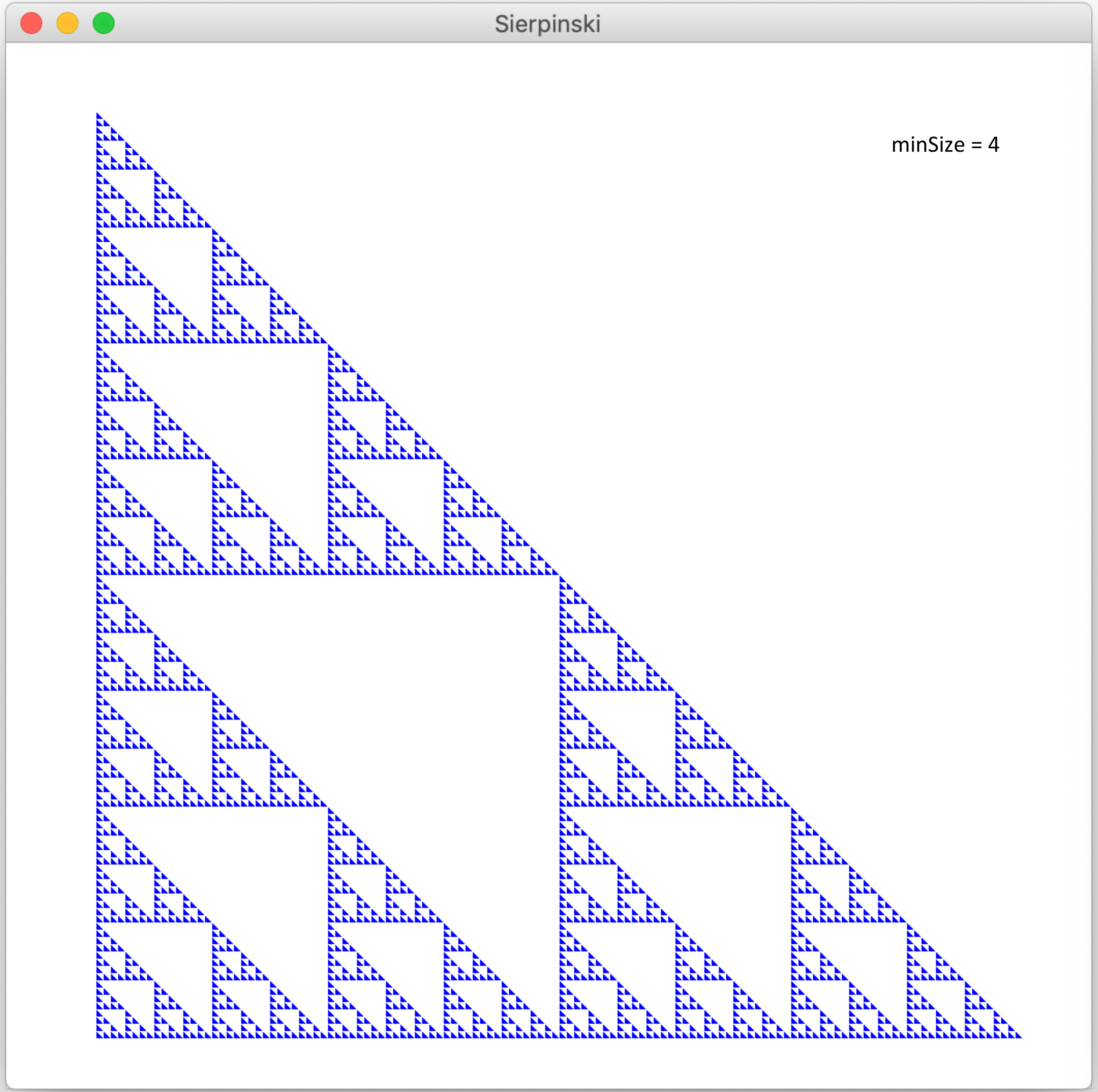
Note: If you encounter any problems with the graphics packages, please send email to both Paul Liu (hai.liu@yale.edu) and Paul Hudak (paul.hudak@yale.edu).

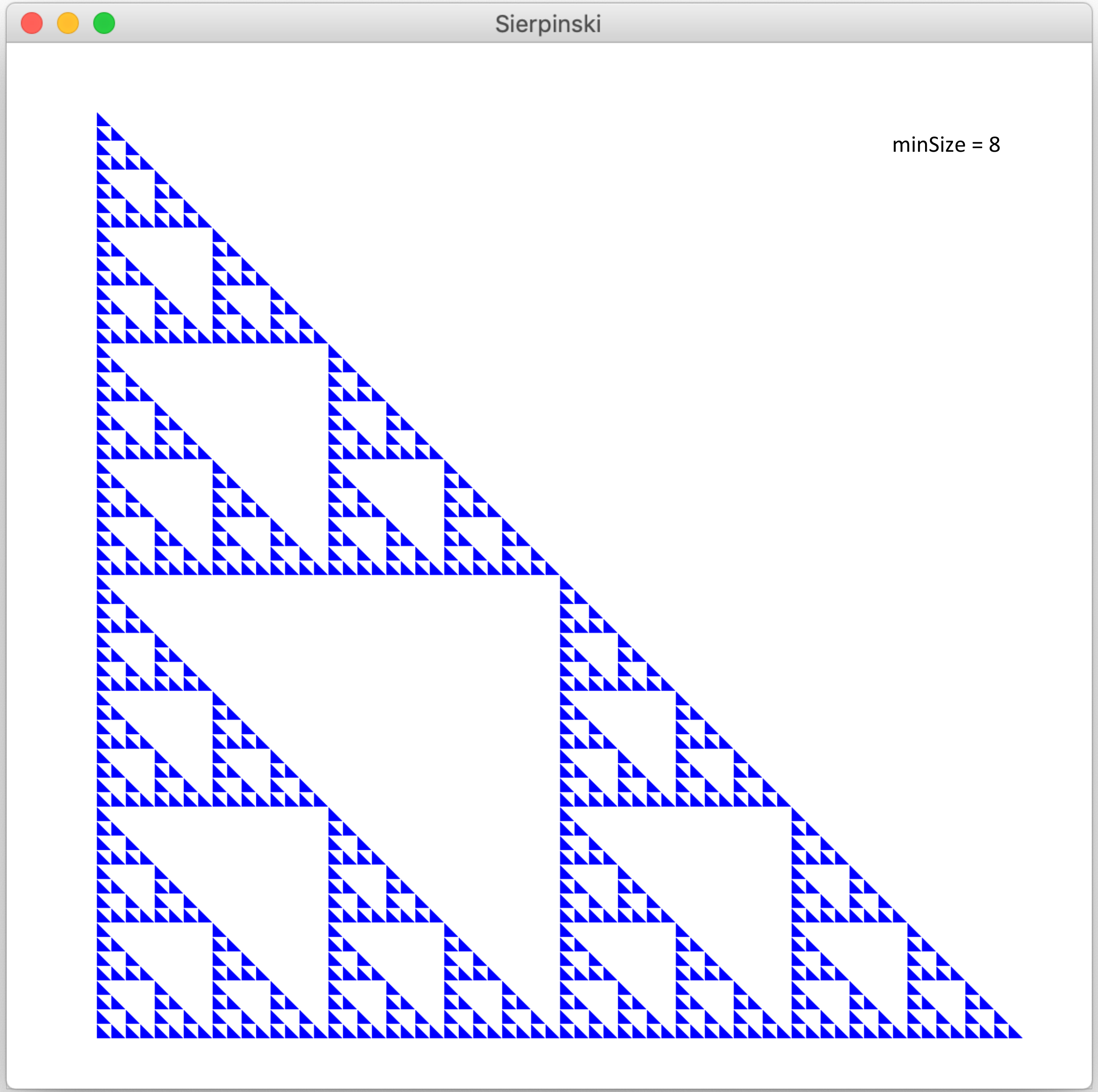
I'd rather spend that time rewriting the program using a more modern graphics library.

Whilst we *will* do that in upcoming slides, for now let's just run the **Scala** program, which produces the same results as the **Haskell** one – see the next two slides.

@philip_schwarz


Sierpinski
minSize = 512
Sierpinski
minSize = 256
Sierpinski
minSize = 128
Sierpinski
minSize = 64
Sierpinski
minSize = 32
Sierpinski
minSize = 16
Sierpinski
minSize = 8
Sierpinski
minSize = 4


Sierpinski
minSize = 4


Sierpinski
minSize = 8

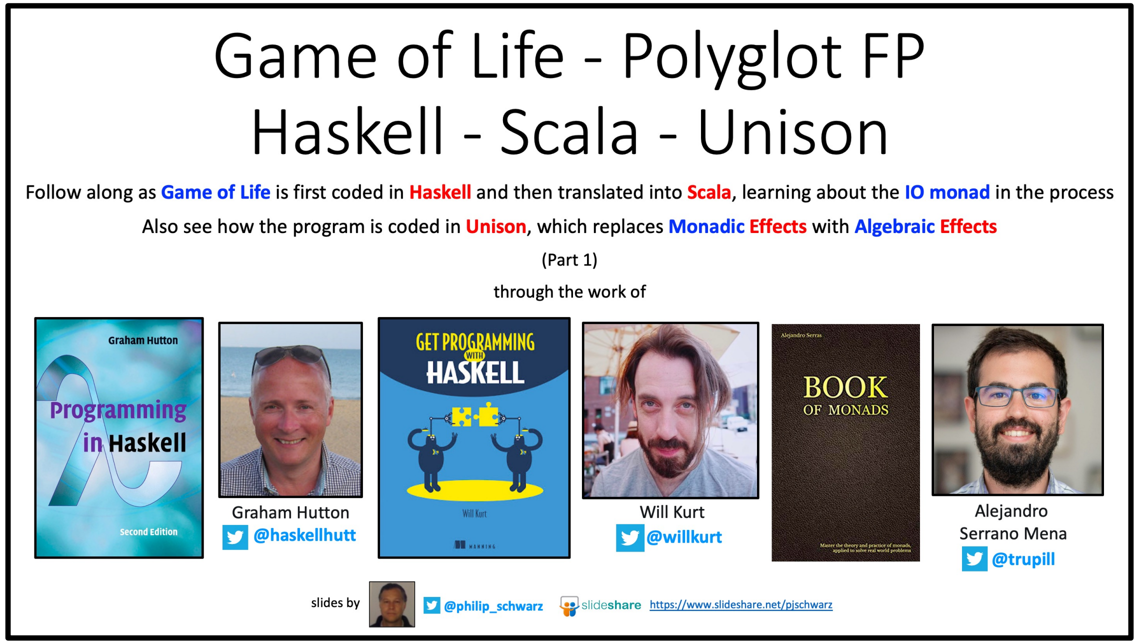
@philip_schwarz

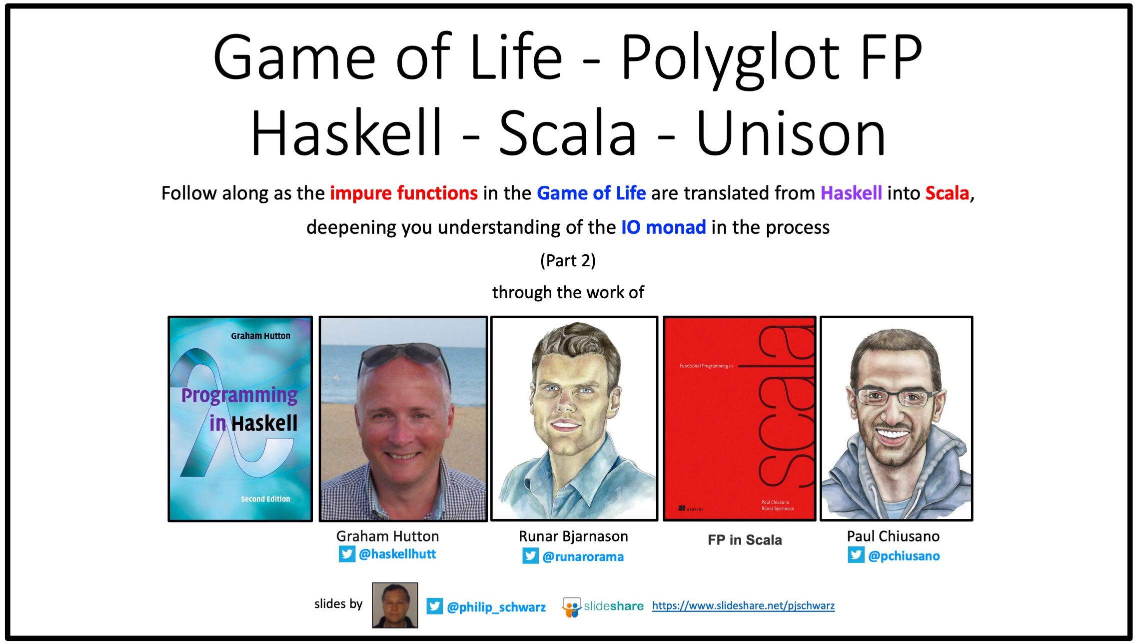
The previous slides provided only a basic, and at times handwavy, introduction to **Haskell**'s **IO type**.

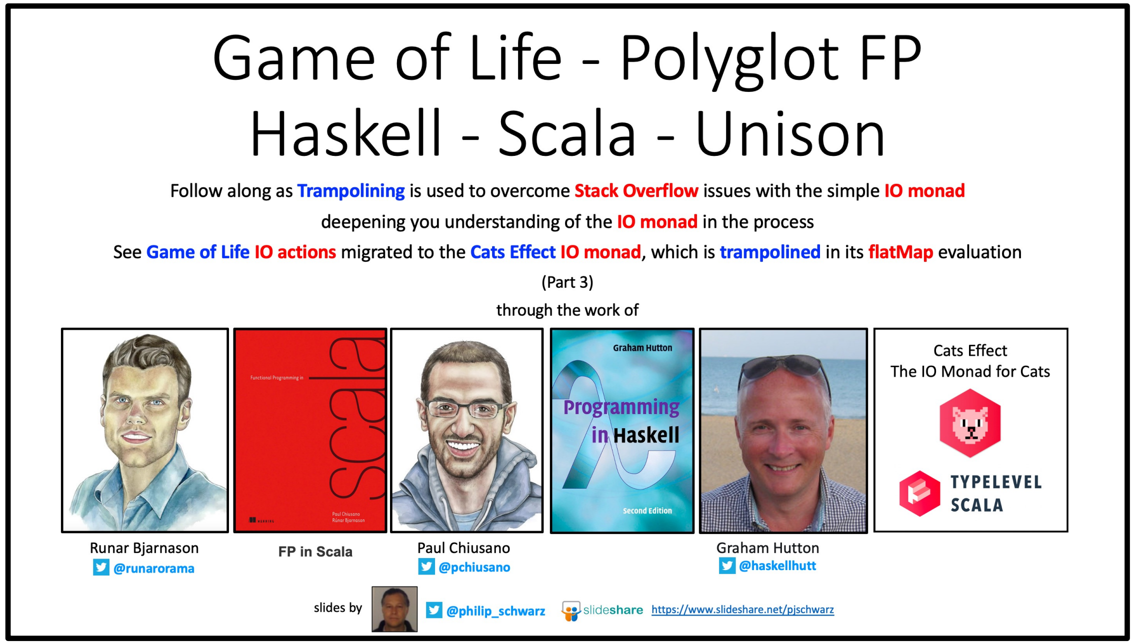
Now that we have seen a little bit of eye candy, let's get a better understanding of the **IO type**, by looking at the following section of **Paul Hudak**'s book: **3.1 Basic Input/Output**.

If you want a more comprehensive introduction to the subject, one that also covers the equivalent **Scala** concepts, then consider looking at the slide decks shown on this slide.

If on the other hand, you are well versed in the subject, feel free to skip the next 4 slides.





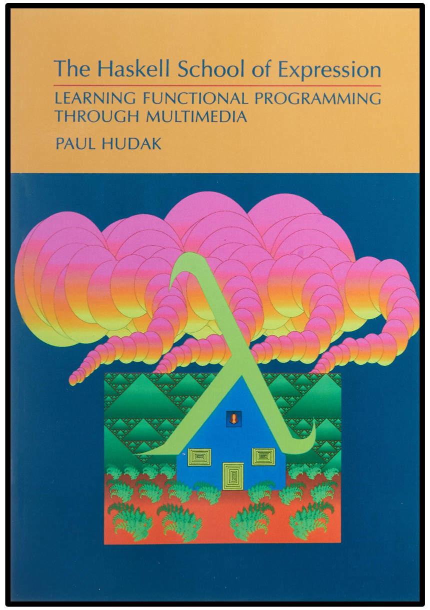


Graphics in Haskell is consistent with the notion of computation via calculation, although it is special enough to warrant the use of special terminology and notation.

…

Graphics is a special case of *input/output* (IO) processing in Haskell, and thus I will begin with a discussion of this more general idea.

3.1 Basic Input/Output.



The Haskell Report defines the result of a program as the value of the name `main` in the module `Main`. On the other hand, the Hugs implementation of Haskell allows you to type whatever expression you wish to the Hugs prompt, and it will evaluate it for you. But in either case, the Haskell system "executes a program" by evaluating an expression, which (for a well-behaved program) eventually yields a value. The system must then display the value on your computer screen in some way that makes sense to you. Most systems will try to display the result in the same way that you would type it in as part of your program. So an integer is printed as an integer, a string as a astring, a list as a list, and so on. I will refer to the area of the computer screen where this result is printed as the ***standard output area***, which may vary from one implementation to another.

But what if a program is intended to write to a file or print a file on a printer or, the main topic of this chapter, draw a picture in a graphics window? These are examples of ***output***, and there are related questions about ***input***: For example, how does a program receive input from a keyboard or mouse?

In general, how does Haskell's "expression-oriented" notion of "computation by calculation" accommodate these various kinds of input and output?

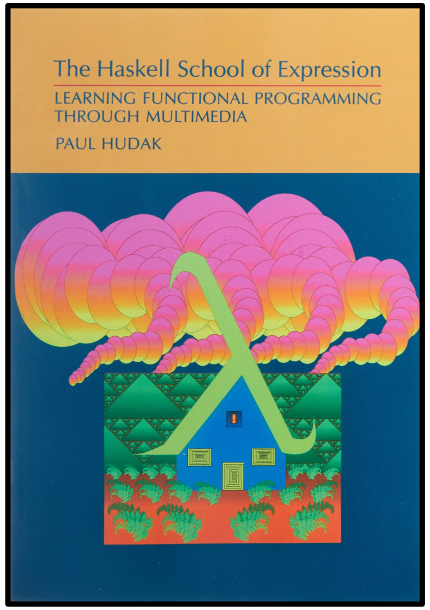
The answer is fairly simple: In Haskell, there is a special kind of value called an ***action***. When a Haskell system evaluates an expression that yields an action, it knows not to try and display the result in the standard output area, but rather, to "take the appropriate action". There are primitive actions - such as writing a single character to a file or receiving a single character from the keyboard – as well as compound actions – such as printing an entire string to a file. Haskell expressions that evaluate to actions are commonly called commands, because they command the Haskell system to perform some kind of action. Haskell functions that yield actions when they are applied are also commonly called commands.

Commands are still just expressions, of course, and some commands return a value for subsequent use by the program: keyboard input, for instance. A command that returns a value of type T has type `IO T`; if no useful value is returned the command has type `IO ()`. The simplest example of a command is return x, which for a value `x::T` immediately returns x and has type `IO T`.

…

To make these ideas clearer, let's consider a few examples. **A very useful command is the `putStr` command, which prints a string argument to the standard output area, and has type `String -> IO ()`. The `()` simply indicates that there is no useful result returned from this action; its sole purpose is to print its argument to the standard output area**. So the program:

```
module Main where
main = putStr "Hello World\n"
```



is the canonical `"Hello World"` program, which is often the first program that people write in a new language.

Suppose now that we want to perform *two* **actions**, such as first writing to a file named `"testFile.txt"`, then printing to the **standard output area**. **Haskell** has a special keyword, **do**, to denote the beginning of a sequence of commands such as this, and so we can write:

```
do writeFile "testFile.txt" "Hello File System"
   putStr "Hello World\n"
```

A **do** expression allows us to sequence an arbitrary number of commands, each of type `IO ()`, using layout to distinguish them. When used in this way, the result of a **do** expression also has type `IO ()`.

Where the file-writing function writeFile has type:

```
writeFile :: FilePath -> String -> IO ()
type FilePath = String
```

So far we have only used **actions** having type `IO ()` (i.e. **output actions**). But what about **input**? As above, we will consider **input** from both the user and the file system.

To get a line of input from the user (which will be typed in the standard input area of the computer screen, usually the same as the standard output area) we can use the function:

```
getLine :: IO String
```

Suppose, for example, that we wish to read a line of input using this function, and then write that line (a string) to a file. To do this we write the compound command:

```
  do s <- getLine
     writeFile "testFile.txt" s
```

Note the syntax for binding s to the result of executing the **getLine command**; because the type of getLine is **IO String**, the type of s is **String**. Its value is then used in the next line as an argument to the writeFile **command**.

Similarly, we can read the entire contents of a file using the **command readFile :: FilePath -> IO String**. For example:

```
  do s <- readFile "testFile.txt"
     putStr s
```

There are many other **commands** available for file, system, and user IO, some in the **Standard Prelude**, and some in various libraries … I will not discuss any of these here; rather, in the next section I will concentrate on ***graphics IO***.

Before that, however, I want to emphasize that, despite the special **do** syntax, **Haskell's IO commands** are no different in status from any other Haskell function or value. **For example, it is possible to create a *list* of actions**, such as:

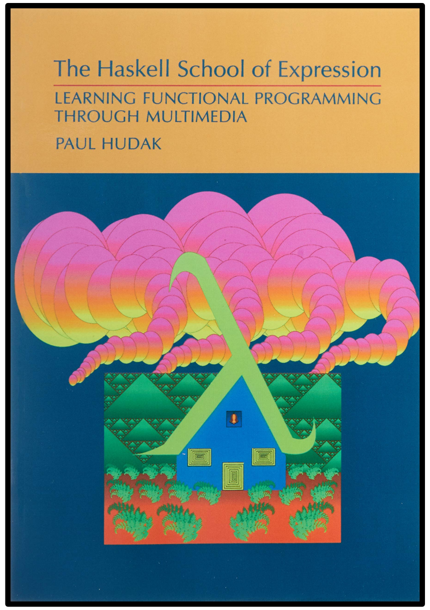
```
 actionList = [ putStr "Hello World\n" ,
                writeFile "testFile.txt" "Hello File System",
                putStr "File successfully written." ]
```

However, a list of actions is just a list of values; they actually don't *do* anything until they are sequenced appropriately using a do expression, and then returned as the value main of the overall program. Still, it is often convenient to place actions into a list as above, and the Haskell Report and Libraries have some useful functions for turning them into commands. In particular, the function sequence_ in the Standard Prelude, when used with IO, has type:

```
 sequence_ :: [IO a] -> IO ()
```

and can thus be applied to the actionList above to yield the single command

```
 main = sequence_ actionList
```



From the function **putChar :: Char -> IO ()**, which prints a single character to the standard output area, we can define the function **putStr** used earlier, which prints an entire string. To do this, let's first define a function that converts a list of characters (i.e. a string) into **a list of IO actions**:

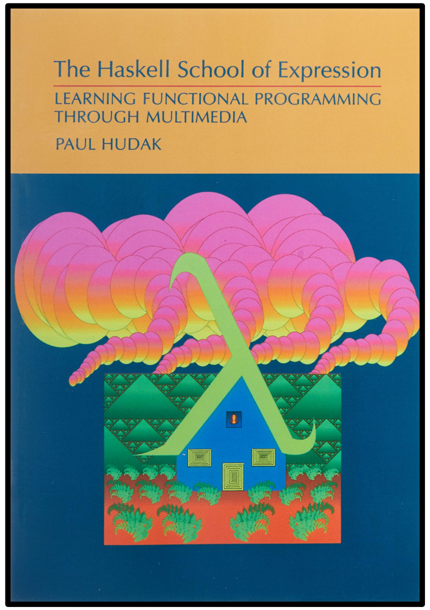
```
putCharList :: String -> [IO ()]
putCharList [] = []
putCharList (c:cs) = putChar c:putCharList cs
```

With this, **putStr** is easily defined:

```
putStr :: String -> IO ()
putStr s = sequence_ (putCharList s)
```

Note that the expression **putCharList** is **a list of actions**, and **sequence_** is used to turn them into a single (compound) command, just as we did earlier. …

IO processing in Haskell is consistent with everything you have learned about programming with expressions and reasoning through calculation, although that may not be completely obvious yet. Indeed, it turns out that a do expression is just syntax for a more primitive way of combining actions using functions. This secret will be revealed in full in Chapter 18.



Now that we have a better understanding of Haskell's IO type, let's turn to the equivalent concept in Scala and see if we can use it to make the Scala program behave more like the Haskell one.

As we saw earlier, Haskell's IO type is a monad. While there is no predefined IO monad in the Scala standard library, we can use the IO monad provided by the Scala library Cats Effect.

We can modify the three highlighted functions on the right so that rather than being side-effecting, i.e. returning `Unit`, they are effectful, i.e. they return `IO[Unit]`. But we cannot do the same for the `paintComponent` function, because it overrides a function defined by `JPanel`, which is provided by Swing (a GUI widget toolkit), and so we cannot change `paintComponent`'s signature.

Because our Scala program uses AWT (Abstract Windowing Toolkit) and Swing, it cannot avoid relying on side effects, but at least we can change the core of the program from being side-effecting to being effectful. We can get `paintComponent` to use the program's pure core to create an IO action, which it then executes.

See the next slide for the required changes.

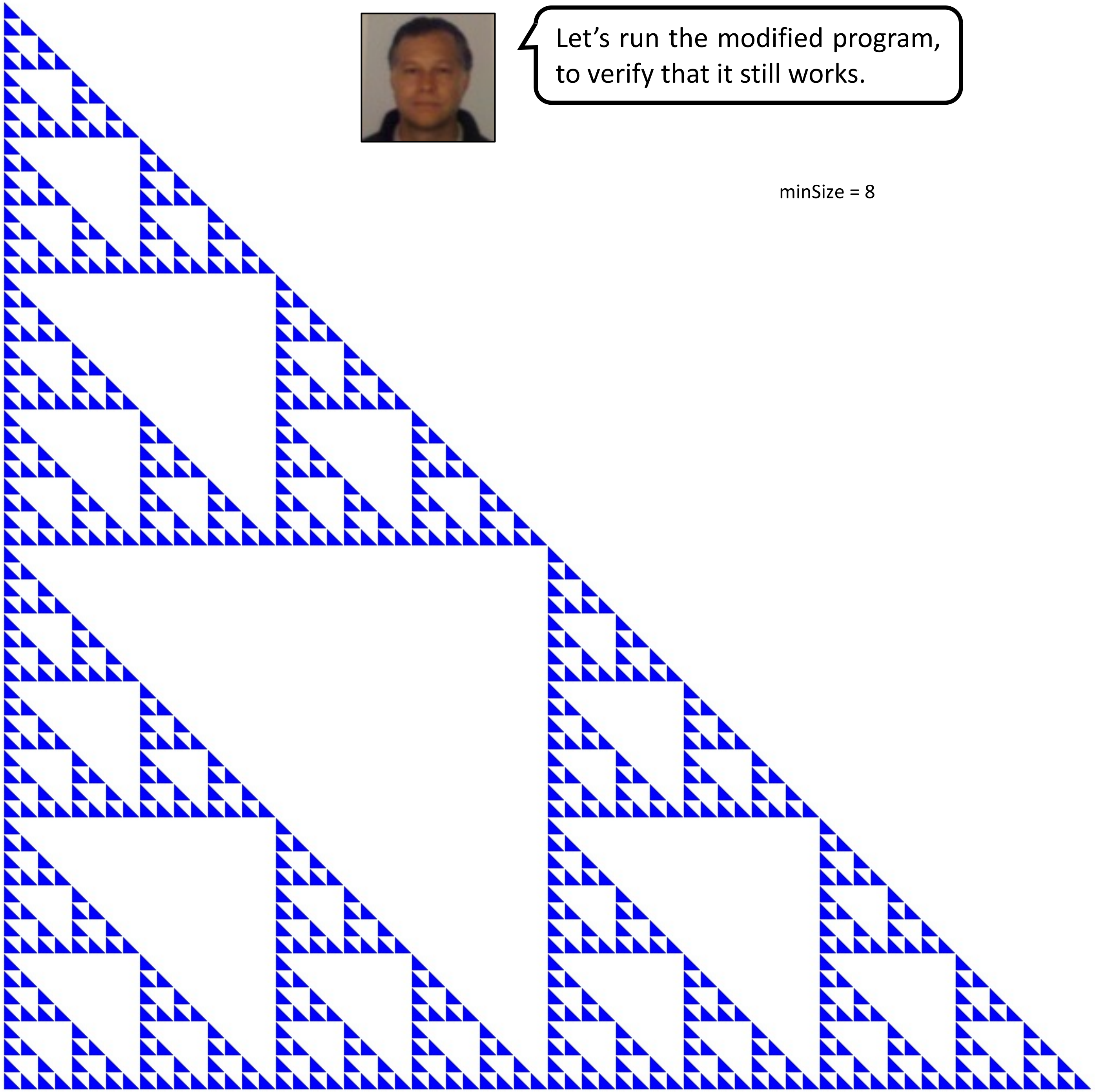
```scala
class SierpinskiJPanel(x:Int, y:Int, size:Int, minSize:Int, colour:Color)
extends JPanel:

  override def paintComponent(g: Graphics): Unit =
    sierpinskiTriangle(g)

  def sierpinskiTriangle(g: Graphics): Unit =
    g.setColor(colour)
    sierpinskiTriangle(g, x, y, size)

  def sierpinskiTriangle(g: Graphics, x: Int, y: Int, size: Int): Unit =
    if size <= minSize
    then fillTriangle(g, x, y, size)
    else
      val halfSize = size / 2
      sierpinskiTriangle(g, x, y, halfSize)
      sierpinskiTriangle(g, x, y - halfSize, halfSize)
      sierpinskiTriangle(g, x + halfSize, y, halfSize)

  def fillTriangle(g: Graphics, x: Int, y: Int, size: Int): Unit =
    val xs = Array(x, x + size, x)
    val ys = Array(y, y, y - size)
    g.fillPolygon(xs, ys, 3)
```

All we have to do is add the code highlighted in green. The **main** function remains unchanged. The **paintComponent** function first creates an **action** that describes the **side effects** needed to draw triangles on the screen, and then performs those **side effects** by running the **action**.

```
import cats.effect.unsafe.implicits._
import cats.effect.IO

class SierpinskiJPanel(x:Int, y:Int, size:Int, minSize:Int, colour:Color) extends JPanel:

  override def paintComponent(g: Graphics): Unit =
    sierpinskiTriangle(g).unsafeRunSync()

  def sierpinskiTriangle(g: Graphics): IO[Unit] =
    for
      _ <- IO{ g.setColor(colour) }
      _ <- sierpinskiTriangle(g, x, y, size)
    yield ()

  def sierpinskiTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
    if size <= minSize
    then fillTriangle(g, x, y, size)
    else
      val halfSize = size / 2
      for
        _ <- sierpinskiTriangle(g, x, y, halfSize)
        _ <- sierpinskiTriangle(g, x, y - halfSize, halfSize)
        _ <- sierpinskiTriangle(g, x + halfSize, y, halfSize)
      yield ()

  def fillTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
    val xs = Array(x, x + size, x)
    val ys = Array(y, y, y - size)
    IO{ g.fillPolygon(xs, ys, 3) }
```

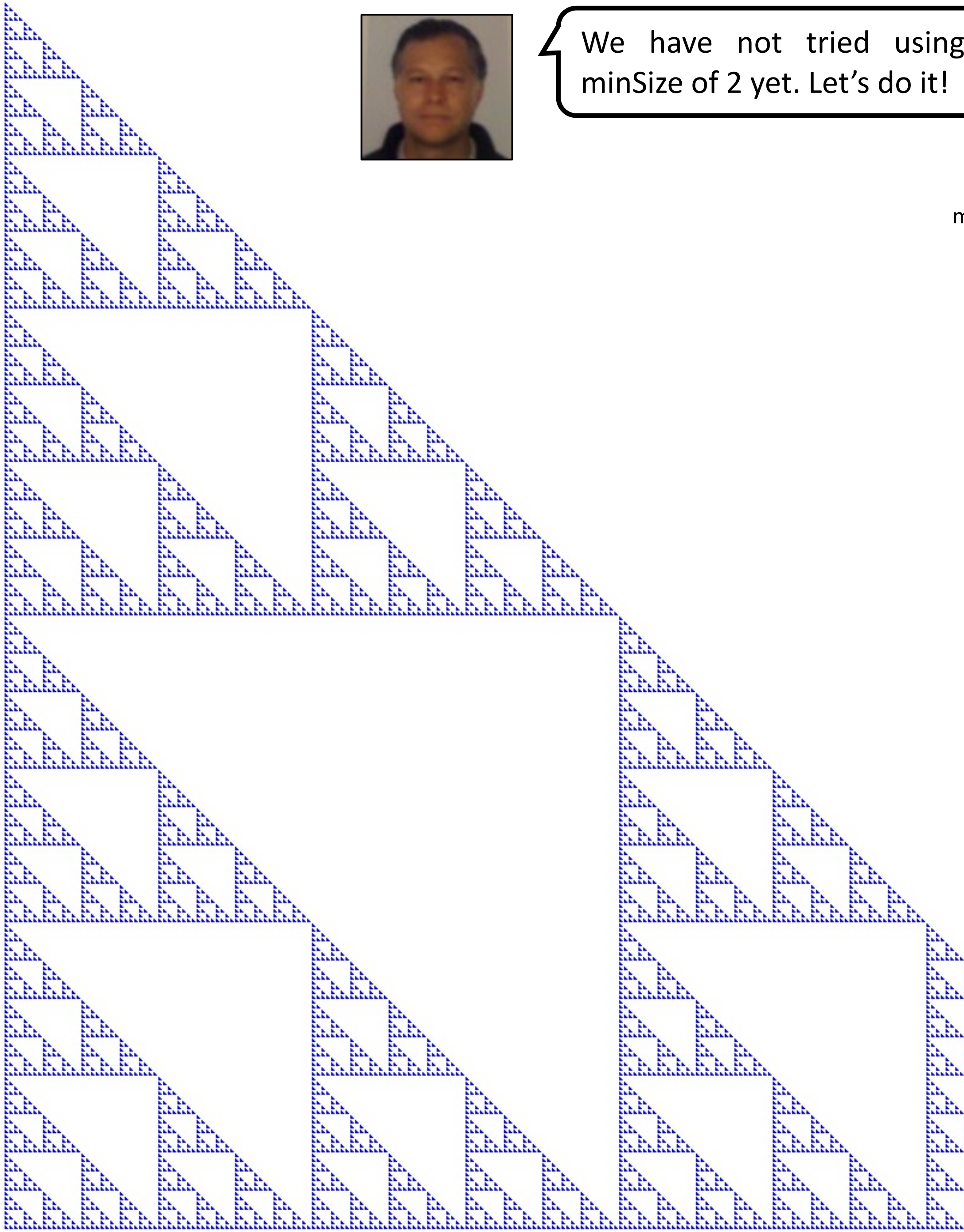
import **Cats Effect**

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

@main def sierpinski: Unit =

  val title = "Sierpinski's Triangle"
  val windowPosition = (0,0)
  val width = 600
  val height = 600
  val backgroundColour = Color.white
  val triangleColour = Color.blue
  val triangleSize = 512
  val triangleXPos = 50
  val triangleYPos = 550

  val minSize = 8

  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Sierpinski")
  frame.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE)
  frame.setBackground(backgroundColour);
  frame.setSize(width, height);

  val sierpinskiTriangle =
    SierpinskiJPanel(
      triangleXPos,
      triangleYPos,
      triangleSize,
      minSize,
      triangleColour
    )
  frame.add(sierpinskiTriangle);
  frame.setVisible(true)
```



Let’s run the modified program, to verify that it still works.

minSize = 8



We have not tried using minSize of 2 yet. Let’s do it!

@philip_schwarz

Remember a few slides ago, when saw **Paul Hudak** saying the following?

> **a** do **expression is just syntax for a more primitive way of combining actions using functions**. This secret will be revealed in full in Chapter 18.

Let’s take a look at the relevant section in the next three slides (actually, feel free to ignore the third one).

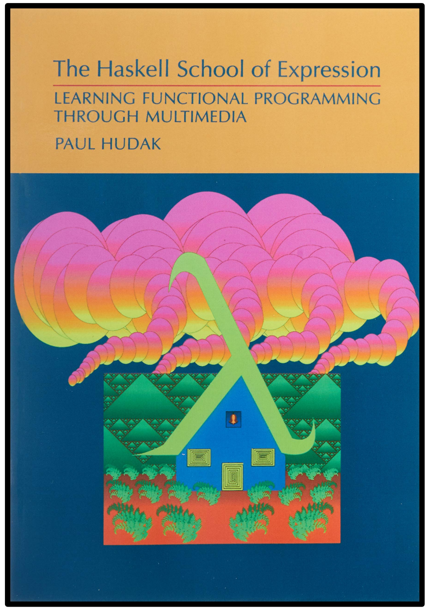
18.2 The Monad Class

There are several classes in **Haskell** that are related to the notion of a **monad**, which can be viewed as a generalization of the principles that underlie **IO**. Because of this, although the names of classes and methods may seem unusual, these **"monadic" operations** are rather intuitive and useful for general programming[2].

There are three classes associated with **Monads**: `Functor` …, `Monad` … and `MonadPlus` …

The **Monad** class defines four basic operators: `(>>=)` (often pronounced **"bind"**), `(>>)` (often pronounced **"sequence"**), ***return***, and ***fail***:

```
class Monad m where
  (>>=)   :: m a -> (a -> m b) -> m b
  (>>)    :: m a -> m b -> m b
  return  :: a -> m a
  fail    :: String -> m a

  m >> k  = m >>= \_ -> k
  fail s  = error s
```



The default methods for `(>>)` and `fail` define behaviours that are almost always just what is needed. Therefore, most instances of **Monad** need only define `(>>=)` and `return`.

Before giving examples of particular instances of **Monad**, I will first reveal another secret in Haskell, namely that the do syntax is actually shorthand for use of the monadic operators! The rules for this are a bit more involved than those for other syntax we've seen, but are still straightforward. The first rule is this:

```
do e => e
```

So something like `do putStr "Hello World"` is equivalent to just `putStr "Hello World"`.

2 Moggi (Moggi, 1989) was one of the first to point out the value of **monads** in describing the semantics of programming languages, and Wadler first popularized their use in functional programming (Wadler, 1992; Peyton Jones and Wadler, 1993).

The next rule is:

```
do e1; e2; …; en
  => e1 >> do e2; …; en
```

For example, combining this rule with the previous one means that:

```
do writeFile "testFile.txt" "Hello File System"
   putStr "Hello World"
```

is equivalent to:

```
writeFile "testFile.txt" "Hello File System" >>
putStr "Hello World"
```

Note now that the sequencing of two commands is just the application of the function `(>>)` to two values of type `IO ()`. There is no magic here – it is all just functional programming.

DETAILS

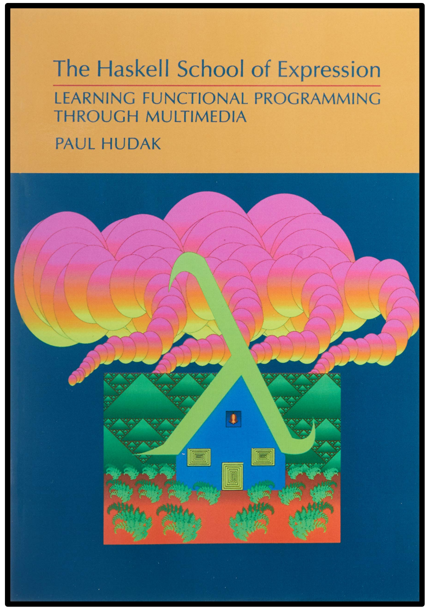
What is the type of `(>>)` ? From the type class declaration we know that its most general type is:

```
(>>) :: Monad m => m a -> m b -> m b
```

However, in the case above, its two arguments both have type `IO ()`, so the type of `(>>)` must be:

```
(>>) :: IO () -> IO () -> IO ()
```

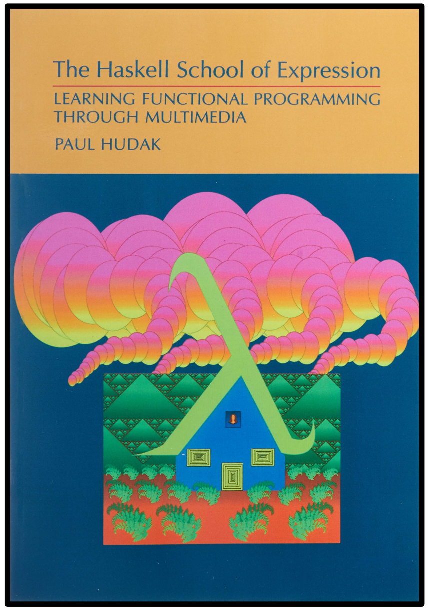
That is, `m = IO`, `a = ()` and `b = ()`. Thus, the type of the result is `IO ()`, as expected.



The rule for pattern matching is the most complex, because we must deal with the situation where the pattern match fails:

```
do pat <- e1; e2; …; en
  => let ok pat = do e2; …; en
          ok _    = fail "…"
     in e1 >>= ok
```

DETAILS
The string argument to **fail** is a compiler-generated error-message, preferably giving some indication of the location of the patter-math failure.



The right way to think of **(>>=)** above is simply this: It "executes" **e1**, and then applies **ok** to the result. What happens after that is defined by **ok**. If the match succeeds, the rest of the commands are executed, otherwise the operation **fail** in the **monad** class is called, which in most cases (because of the default method) results in an error.

A special case of the above rule is the case where the pattern **pat** is just a name, in which case the match cannot fail, so the rule simplifies to:

```
do x <- e1; e2; …; en
  => e1 >>= \x -> do e2; …; en
```

The final rule deals with the **let** notation within a **do** expression:

```
do let <- declist; e2; …; en
  => let <- declist in do e2; …; en
```

DETAILS
Although we have not used this feature, note that a **let** inside of a **do** can take multiple definitions, as implied by the name *declist*.

As mentioned earlier, because you already understand **Haskell IO**, you should have a fair amount of intuition about what the **monadic** operators do. Unfortunately, we can't look very closely at the instance of **Monad** for the type **IO**, because it ultimately relies on the state of the underlying operating system, which we don't have direct access to other than through primitive operations that communicate with it. Even then, these operations vary from system to system. Nevertheless, a proper implementation of **IO** in **Haskell** is obliged to obey the following **monad** laws…

In the next slide we take the **Haskell** **`sierpinskiTri`** function and show how, instead of sequencing **IO actions** using ‘**do**’, it can do so using **`sequence_`** or **`>>`**.

sequence multiple **IO actions** using '**do**'

```haskell
sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in do sierpinskiTri w x y size2
        sierpinskiTri w x (y - size2) size2
        sierpinskiTri w (x + size2) y size2
```

sequence a list of **IO actions** using **sequence_**

```haskell
sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in sequence_
       [sierpinskiTri w x y size2,
        sierpinskiTri w x (y - size2) size2,
        sierpinskiTri w (x + size2) y size2]
```

sequence multiple **IO actions** using **>>**

```haskell
sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in sierpinskiTri w x y size2 >>
     sierpinskiTri w x (y - size2) size2 >>
     sierpinskiTri w (x + size2) y size2
```

In the nex slide, we use **Cats** and **Cats Effect** to do something similar with the two **Scala** **`sierpinskiTriangle`** functions.

@philip_schwarz

```scala
def sierpinskiTriangle(g: Graphics): IO[Unit] =
  for
    _ <- IO{ g.setColor(colour) }
    _ <- sierpinskiTriangle(g, x, y, size)
  yield ()

def sierpinskiTriangle(g:Graphics, x:Int, y:Int, size:Int):IO[Unit] =
  if size <= minSize
  then fillTriangle(g, x, y, size)
  else
    val halfSize = size / 2
    for
      _ <- sierpinskiTriangle(g, x, y, halfSize)
      _ <- sierpinskiTriangle(g, x, y - halfSize, halfSize)
      _ <- sierpinskiTriangle(g, x + halfSize, y, halfSize)
    yield ()
```

sequencing **actions** using a **for comprehension**

```scala
def sierpinskiTriangle(g: Graphics): IO[Unit] =
  IO{ g.setColor(colour) } flatMap { _ =>
    sierpinskiTriangle(g, x, y, size)
  }

def sierpinskiTriangle(g:Graphics, x:Int, y:Int, size:Int):IO[Unit] =
  if size <= minSize
  then fillTriangle(g, x, y, size)
  else
    val halfSize = size / 2
    sierpinskiTriangle(g, x, y, halfSize) flatMap { _ =>
      sierpinskiTriangle(g, x, y - halfSize, halfSize) flatMap { _ =>
        sierpinskiTriangle(g, x + halfSize, y, halfSize)
      }
    }
```

putting **actions** in a list and sequencing them using **sequence_**

sequencing **actions** using **flatMap**

sequencing **actions** using **>>**

```scala
def sierpinskiTriangle(g: Graphics): IO[Unit] =
  List(
    IO{ g.setColor(colour) },
    sierpinskiTriangle(g, x, y, size)
  ).sequence_

def sierpinskiTriangle(g:Graphics, x:Int, y:Int, size:Int):IO[Unit] =
  if size <= minSize
  then fillTriangle(g, x, y, size)
  else
    val halfSize = size / 2
    List(
      sierpinskiTriangle(g, x, y, halfSize),
      sierpinskiTriangle(g, x, y - halfSize, halfSize),
      sierpinskiTriangle(g, x + halfSize, y, halfSize)
    ).sequence_
```

```scala
import cats.implicits._
```

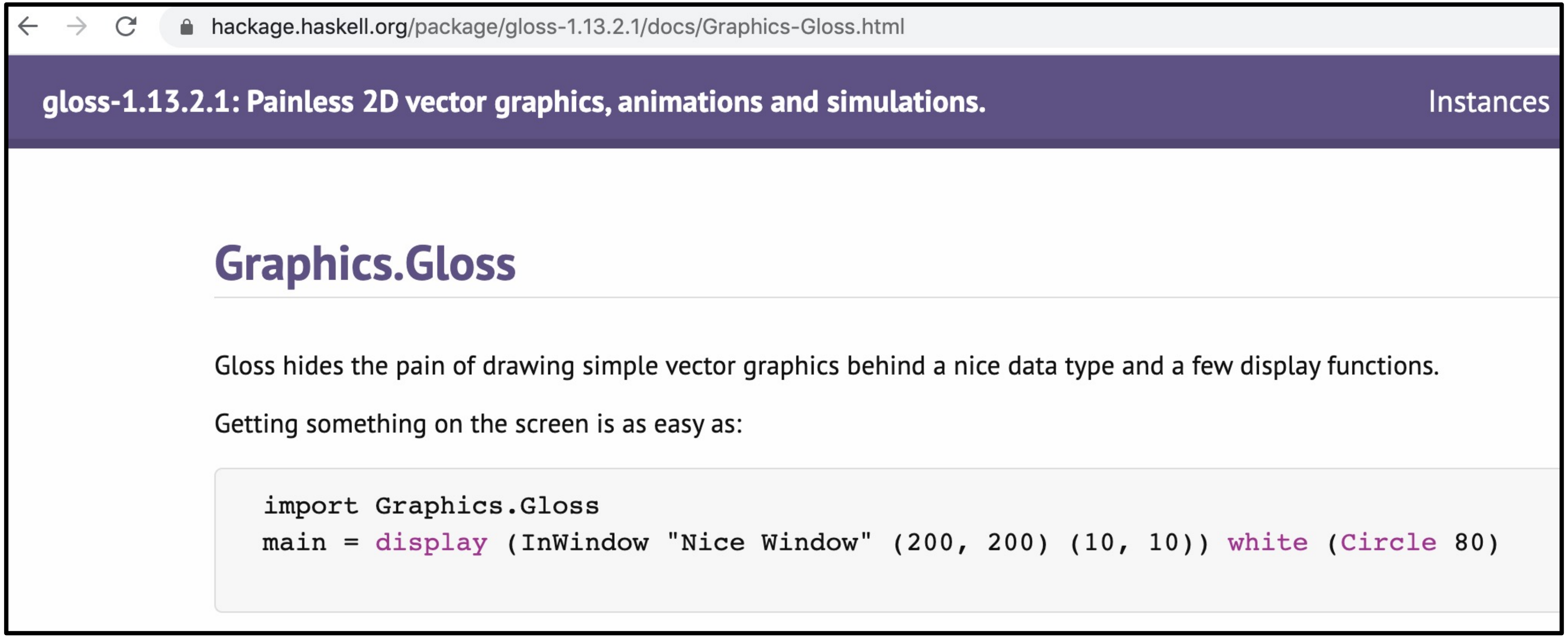
```scala
def sierpinskiTriangle(g: Graphics): IO[Unit] =
  IO{ g.setColor(colour) } >> sierpinskiTriangle(g, x, y, size)

def sierpinskiTriangle(g:Graphics, x:Int, y:Int, size:Int):IO[Unit] =
  if size <= minSize
  then fillTriangle(g, x, y, size)
  else
    val halfSize = size / 2
    sierpinskiTriangle(g, x, y, halfSize) >>
    sierpinskiTriangle(g, x, y - halfSize, halfSize) >>
    sierpinskiTriangle(g, x + halfSize, y, halfSize)
```

I said earlier that instead of going through the Yak shaving required to run the **Haskell** program, I'd rather rewrite the program using a more modern graphics library.

There is a library called **`Gloss`**, whose documentation says the following:

- Gloss **hides the pain** of drawing simple vector graphics behind a nice data type and a few display functions.
- Get something cool on the screen in **under 10 minutes**.

That's exactly what we need. In the next two slides we identify the few things that we'll need in order to draw the **Sierpinski triangle**, and come up with a new version of the program that uses **`Gloss`** instead of **`SOEGraphics`**.



Graphics.Gloss

Safe Haskell	None
Language	Haskell2010

Gloss hides the pain of drawing simple vector graphics behind a nice data type and a few display functions.

Getting something on the screen is as easy as:

```
import Graphics.Gloss
main = display (InWindow "Nice Window" (200, 200) (10, 10)) white (Circle 80)
```

First we create a Picture, e.g. a Circle with a radius of 80 pixels.

A picture is drawn on a Display, so we create a Display that consists of a window that has the desired title and is of the desired size (width an height) and is located at the desired position (x and y coordinates).

To draw a picture, we call the display function with a Display, the desired background colour for the Display, and the Picture to be drawn on the Display.

Just like the drawInWindow function in the SOEGraphics library, the display function in the Gloss library does not produce any side effects: it returns an IO action.

Graphics.Gloss.Interface.Pure.Display

Safe Haskell	None
Language	Haskell2010

Display mode is for drawing a static picture.

Documentation

module Graphics.Gloss.Data.Display

module Graphics.Gloss.Data.Picture

module Graphics.Gloss.Data.Color

display

:: Display	Display mode.
-> Color	Background color.
-> Picture	The picture to draw.
-> IO ()	

Open a new window and display the given picture.

Graphics.Gloss.Data.Display

Safe Haskell	Safe-Inferred
Language	Haskell2010

Documentation

data **Display** # Source

Describes how Gloss should display its output.

Constructors

InWindow String (Int, Int) (Int, Int)	Display in a window with the given name, size and position.
FullScreen	Display full screen.

On the next slide we see the code for the new **Haskell** version of the program.

gloss-1.3.4.1: Painless 2D vector graphics, animations and simulations.

Graphics.Gloss.Data.Picture

Data types for representing pictures.

Documentation

`type Point = (Float, Float)`

A point on the x-y plane. Points can also be treated as `Vectors`, so Graphics.

`type Vector = Point`

A vector can be treated as a point, and vis-versa.

`type Path = [Point]`

A path through the x-y plane.

`data Picture`

A 2D picture

Constructors

Blank	A blank picture, with nothing in it.
Polygon Path	A polygon filled with a solid color.
Line Path	A line along an arbitrary path.
Circle Float	A circle with the given radius.
ThickCircle Float Float	A circle with the given thickness and radius. If the thick
Text String	Some text to draw with a vector font.
Bitmap Int Int ByteString	A bitmap image with a width, height and a ByteString h
Color Color Picture	A picture drawn with this color.
Translate Float Float Picture	A picture translated by the given x and y coordinates.
Rotate Float Picture	A picture rotated by the given angle (in degrees).
Scale Float Float Picture	A picture scaled by the given x and y factors.
Pictures [Picture]	A picture consisting of several others.

```haskell
fillTriangle :: Int -> Int -> Int -> Picture
fillTriangle x y size =
  let xPos = fromIntegral x
      yPos = fromIntegral y
      side = fromIntegral size
      bottomLeftPoint  = (xPos       , yPos)
      bottomRightPoint = (xPos + side, yPos)
      topPoint         = (xPos       , yPos + side)
      triangle = polygon [bottomLeftPoint,
                          bottomRightPoint,
                          topPoint]
      colouredTriangle = color triangleColour triangle
   in colouredTriangle

sierpinskiTriangle :: Int -> Int -> Int -> Picture
sierpinskiTriangle x y size =
  if size <= minSize
  then fillTriangle x y size
  else
    let halfSize = size `div` 2
     in pictures [sierpinskiTriangle x y halfSize,
                  sierpinskiTriangle x (y + halfSize) halfSize,
                  sierpinskiTriangle (x + halfSize) y halfSize]
```

```haskell
import Graphics.Gloss
windowTitle = "Sierpinski"
windowPosition = (0,0)
width = 600
height = 600
windowDimensions = (width, height)
backgroundColour = white
horizontalShift = -(fromIntegral width)/2
verticalShift = -(fromIntegral height)/2

windowDisplay :: Display
windowDisplay =
  InWindow windowTitle
           windowDimensions
           windowPosition

triangleColour = red
triangleSize = 512
triangleXPos = 50
triangleYPos = 50

minSize :: Int
minSize = 8
```

```haskell
main :: IO ()
main = let triangle = sierpinskiTriangle triangleXPos triangleYPos triangleSize
           shiftedTriangle = (translate horizontalShift verticalShift triangle)
        in display windowDisplay backgroundColour shiftedTriangle
```

While there isn't a `Picture` that is a triangle, there is one that is a `Polygon`. We'll create triangles using this.

We can put a number of pictures in a list and then use `Pictures` to **compose** them into a single picture. We'll compose into a single picture the three triangles produced by recursive calls to `sierpinskiTriangle`.

We can change the default `color` of a `Picture`.

We can `translate` a picture - we'll use this to get the final picture into the same position as in the existing Haskell program.

Also (not shown), there are uncapitalised aliases for picture constructors: `color`, `pictures`, `polygon`, etc.

```haskell
import SOEGraphics

minSize :: Int
minSize = 8

fillTri :: Window -> Int -> Int -> Int -> IO ()
fillTri w x y size =
  drawInWindow w
    (withColor Blue
      (polygon [(x,y),(x + size,y),(x,y - size)]))

sierpinskiTri :: Window -> Int -> Int -> Int -> IO ()
sierpinskiTri w x y size =
  if size <= minSize
  then fillTri w x y size
  else let size2 = size `div` 2
  in do sierpinskiTri w x y size2
        sierpinskiTri w x (y - size2) size2
        sierpinskiTri w (x + size2) y size2

main :: IO ()
main =
  runGraphics(
    do w <- openWindow "Sierpinski's Triangle" (400,400)
       sierpinskiTri w 50 300 256
  )
```

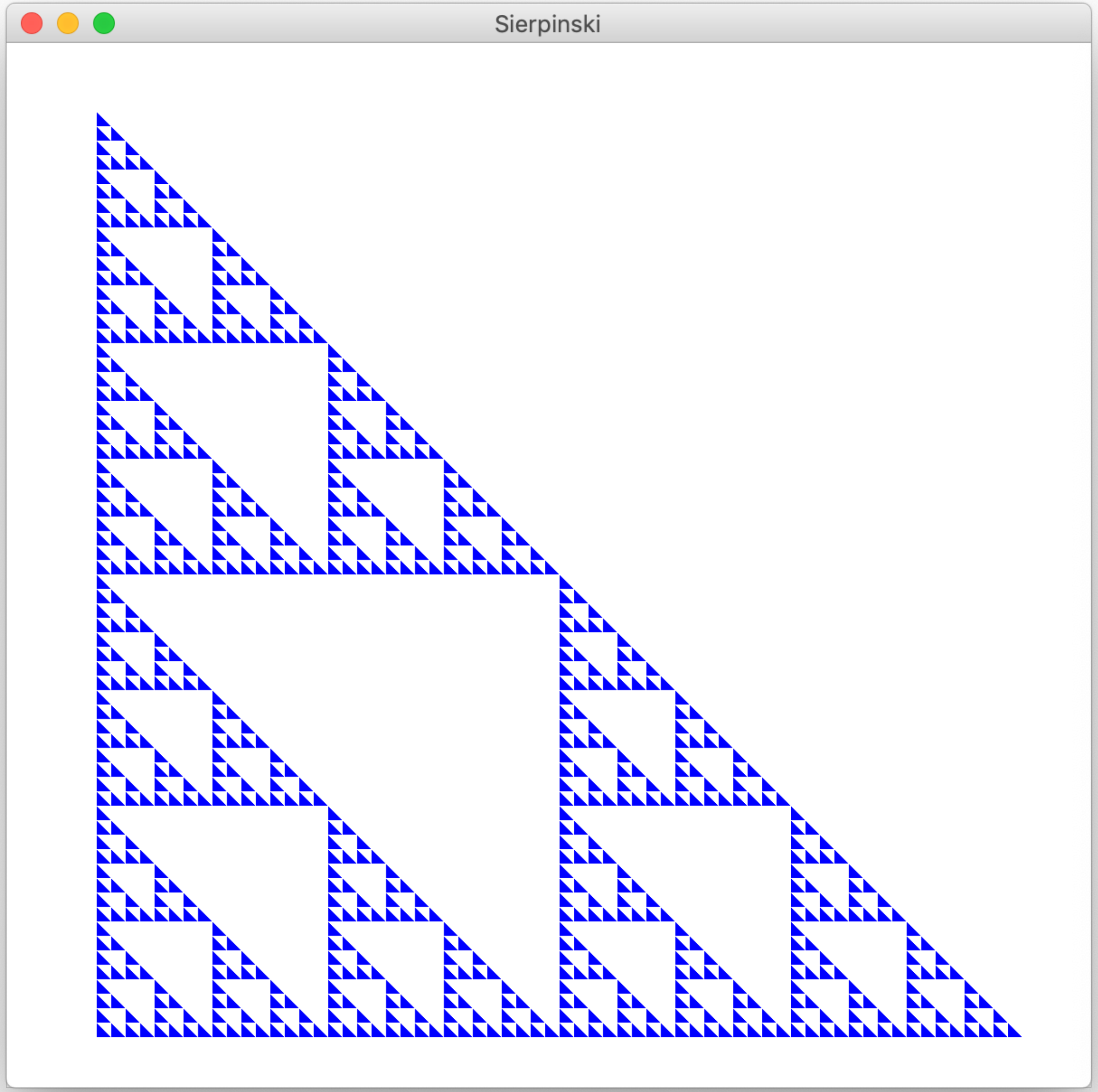
Here again are the the current version of the **Haskell** program (on the left), which uses **SOEGraphics**, and the new version (below), which uses **Gloss**.

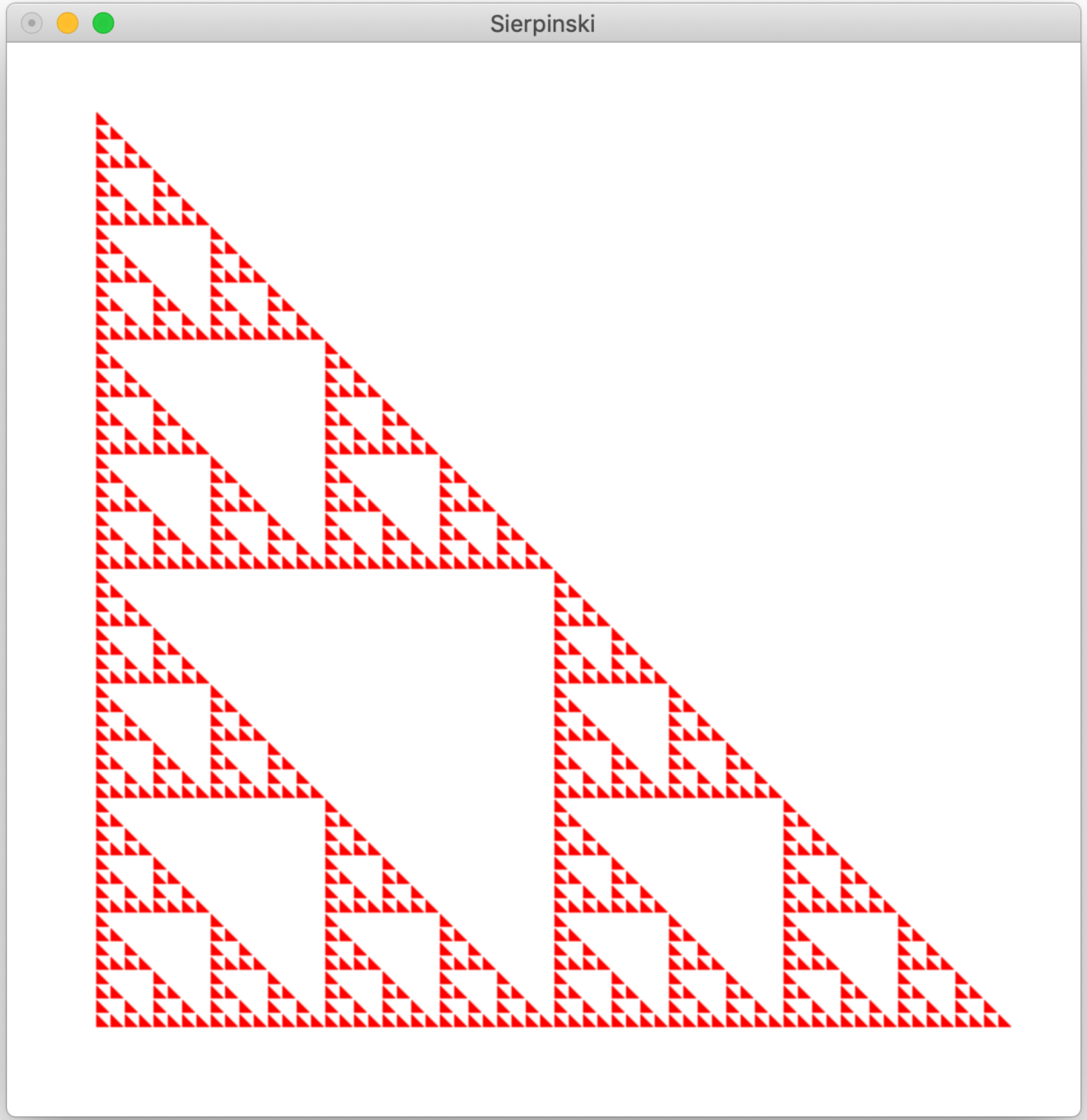
```haskell
import Graphics.Gloss
windowTitle = "Sierpinski"
windowPosition = (0,0)
width = 600
height = 600
windowDimensions = (width, height)
backgroundColour = white
horizontalShift = -(fromIntegral width)/2
verticalShift = -(fromIntegral height)/2

windowDisplay :: Display
windowDisplay =
  InWindow windowTitle
           windowDimensions
           windowPosition

triangleColour = red
triangleSize = 512
triangleXPos = 50
triangleYPos = 50

minSize :: Int
minSize = 8
```

```haskell
fillTriangle :: Int -> Int -> Int -> Picture
fillTriangle x y size =
  let xPos = fromIntegral x
      yPos = fromIntegral y
      side = fromIntegral size
      bottomLeftPoint  = (xPos       , yPos)
      bottomRightPoint = (xPos + side, yPos)
      topPoint         = (xPos       , yPos + side)
      triangle = polygon [bottomLeftPoint,
                          bottomRightPoint,
                          topPoint]
      colouredTriangle = color triangleColour triangle
   in colouredTriangle

sierpinskiTriangle :: Int -> Int -> Int -> Picture
sierpinskiTriangle x y size =
  if size <= minSize
  then fillTriangle x y size
  else
    let halfSize = size `div` 2
     in pictures [sierpinskiTriangle x y halfSize,
                  sierpinskiTriangle x (y + halfSize) halfSize,
                  sierpinskiTriangle (x + halfSize) y halfSize]
```

```haskell
main :: IO ()
main = let triangle = sierpinskiTriangle triangleXPos triangleYPos triangleSize
           shiftedTriangle = (translate horizontalShift verticalShift triangle)
        in display windowDisplay backgroundColour shiftedTriangle
```

Let’s run the **Scala** program again (on the left), and the new **Haskell** program (on the right).





To conclude this slide deck, let's write a new, much simpler version of the **Scala** program, using **Doodle**, a library which, like **Haskell's Gloss**, adopts the concepts of **picture composition** and of the **separation** between, on the one hand, creating a picture, a **description** of something to be drawn, and on the other hand, doing the actual drawing, by processing/interpreting the picture.

Doodle

https://www.creativescala.org/doodle/

https://github.com/creativescala/doodle

Principles

Doodle: Compositional Vector Graphics

A few principles guide the design of **Doodle**, and differentiate it from other graphics libraries. The section explains these principles.

Pictures are Created by Composition

In Doodle a picture is constructed by combining together smaller pictures. For example, **we can create a row by putting pictures beside each other**. This idea of creating complex things from simpler things is known as ***composition***.

There are several implications of this, which means that **Doodle** operates differently to many other graphics libraries. This first is that **Doodle does not draw anything on the screen until you explicitly ask it to, say by calling the `draw` method**. **A picture represents a description of something we want to draw**. **A backend turns this description into something we can see (which might be on the screen or in a file)**. **This separation of description and action is known as the *interpreter pattern***. **The description is a "program" and a backend is an "interpreter" that runs that program.** In the graphics world the approach that **Doodle** takes is sometimes known as retained mode, while the approach of drawing immediately to the screen is known as immediate mode.

Another implication is that **Doodle can allow relative layout of objects**. **In Doodle we can say that one picture is next to another and Doodle will work out where on the screen they should be**. This requires a retained mode API as you need to keep around information about a picture to work out how much space it takes up.

A final implication is that **pictures have no mutable state**. **This is needed for composition** so you can, for example, put a picture next to itself and have things render correctly.

All of these ideas are core to functional programming, so you may have seen them in other contexts if you have experienced with functional programming. If not, don't worry. You'll quickly understand them once you start using **Doodle**, as **Doodle** makes the ideas very concrete.

Doodle

Image

The `Image` library is the easiest way to create images using Doodle. The tradeoff the `Image` library makes is that it only support a (large but limited) subset of operations that are supported across all the backends.

`Image` is based on *composition* and the *interpreter pattern*.

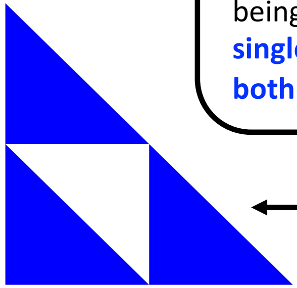
Composition basically means that **we build big Images out of small Images**. For example, if we have an `Image` describing a red square and an Image describing a blue square

```
val redSquare = Image.square(100).fillColor(Color.red)
val blueSquare = Image.square(100).fillColor(Color.blue)
```

we can create an Image describing a red square next to a blue square by combining them together.

```
val combination = redSquare.beside(blueSquare)
```

Are you thinking what I am thinking? Yes, this should simplify things: **instead of creating three different triangles**, as we have being doing up to now, **we create a single triangle and then place it both above itself and next to itself**.

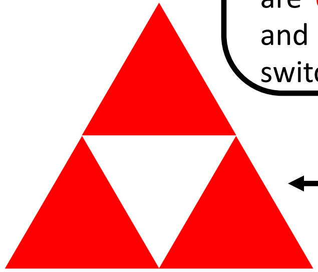


The **interpreter pattern** means that **we separate describing the Image from rendering it**. **Writing**

```
Image.square(100)
```

doesn't draw anything. **To draw an image we need to call the `draw()` method**. This separation is important for **composition**; if we were to immediately draw we would lose **composition**.

By the way, did you notice that while the triangles that we have been drawing up to now have been **isosceles** (two equal sides and two equal angles), the triangles on the **Wikipedia** page are **equilateral** (three equal sides and three equal angles)? Let's switch to **equilateral** triangles.



Basic Shapes

…

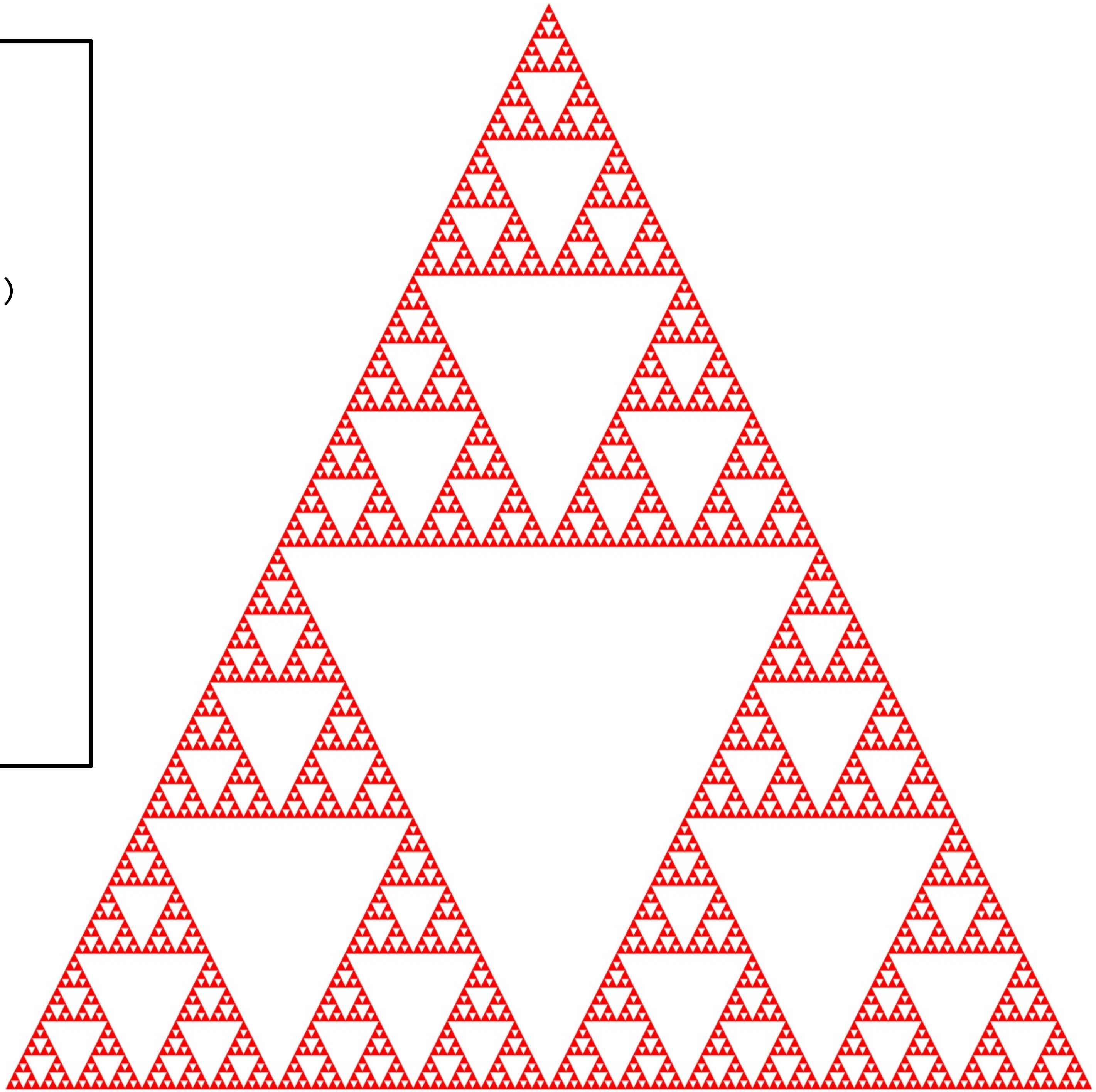
`Image.triangle(width, height)` creates an **isoceles triangle** with the given width and height.

@philip_schwarz

On the next slide: a simple **Scala** program that draws the **Sierpinski Triangle** using **Doodle**. Also: an example of the program's output.

Subsequent slides show many more examples.

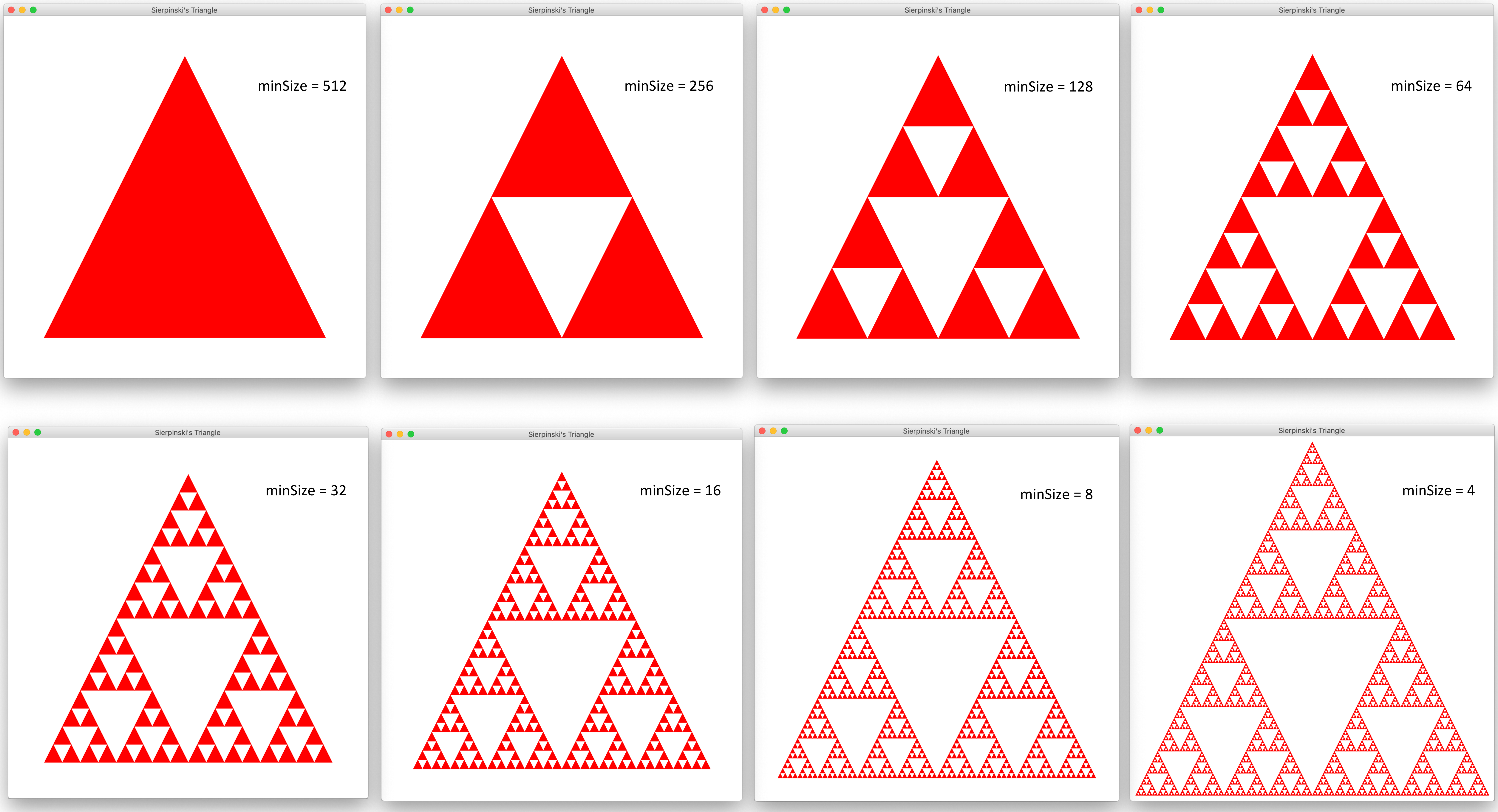
```scala
val minSize = 64

def sierpinskiTriangle(size: Int): Image =
  if size <= minSize
  then fillTriangle(size)
  else
    val triangle = sierpinskiTriangle(size / 2)
    triangle above (triangle beside triangle)

def fillTriangle(size: Int): Image =
  Image.triangle(size,size)
       .strokeColor(Color.red)

@main def sierpinski: Unit =
  sierpinskiTriangle(512)
  .draw(Frame.size(660, 660)
             .title("Sierpinski’s Triangle")
             .background(Color.white)
             .fillColor(Color.red))
```

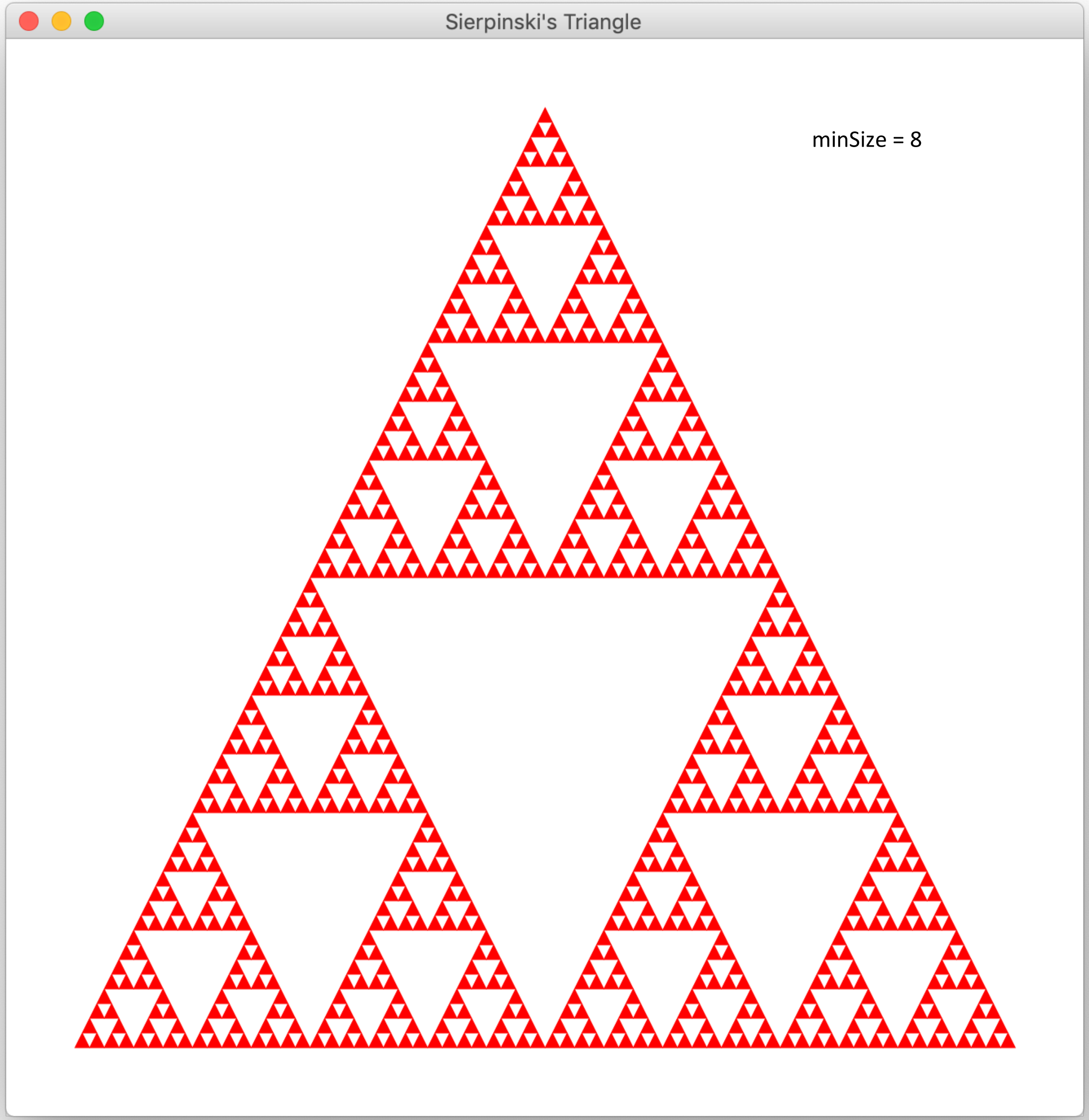


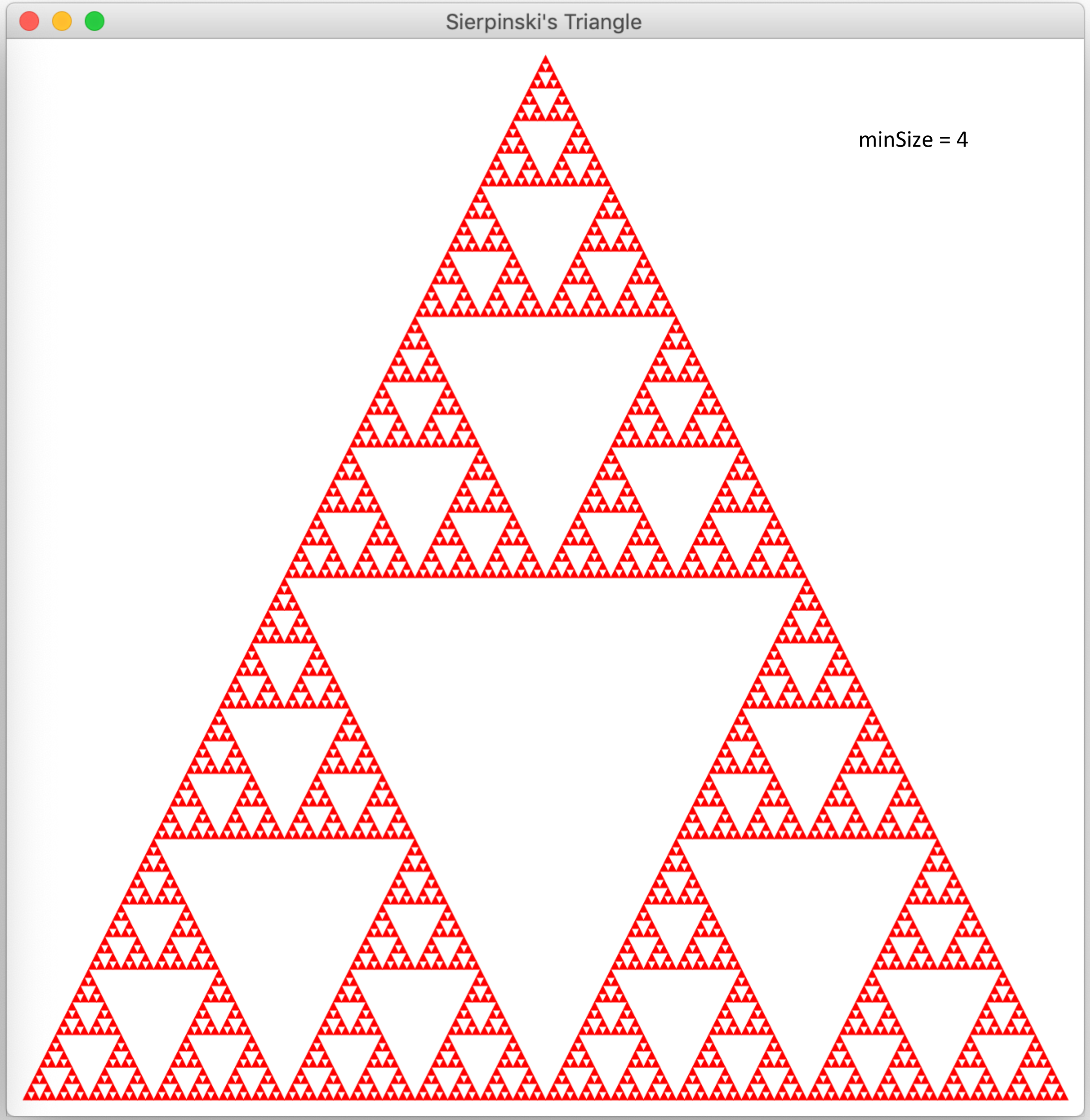
```scala
val frameTitle = "Sierpinski's Triangle"
val frameWidth = 660
val frameHeight = 660
val frameBackgroundColour = Color.white
val frame = Frame.size(frameWidth, frameHeight)
                 .title(title)
                 .background(frameBackgroundColour)
                 .fillColor(Color.red)

val triangleSize = 512
val triangleColour = Color.red

val minSize = 64

def sierpinskiTriangle(size: Int): Image =
  if size <= minSize
  then fillTriangle(size)
  else
    val triangle = sierpinskiTriangle(size / 2)
    triangle above (triangle beside triangle)

def fillTriangle(size: Int): Image =
  Image.triangle(size,size)
       .strokeColor(triangleColour)

@main def sierpinski: Unit =
  sierpinskiTriangle(triangleSize)
  .draw(frame)
```

Same code as on the previous slide, except that we have extracted several explaining variables and we are now showing the required imports.

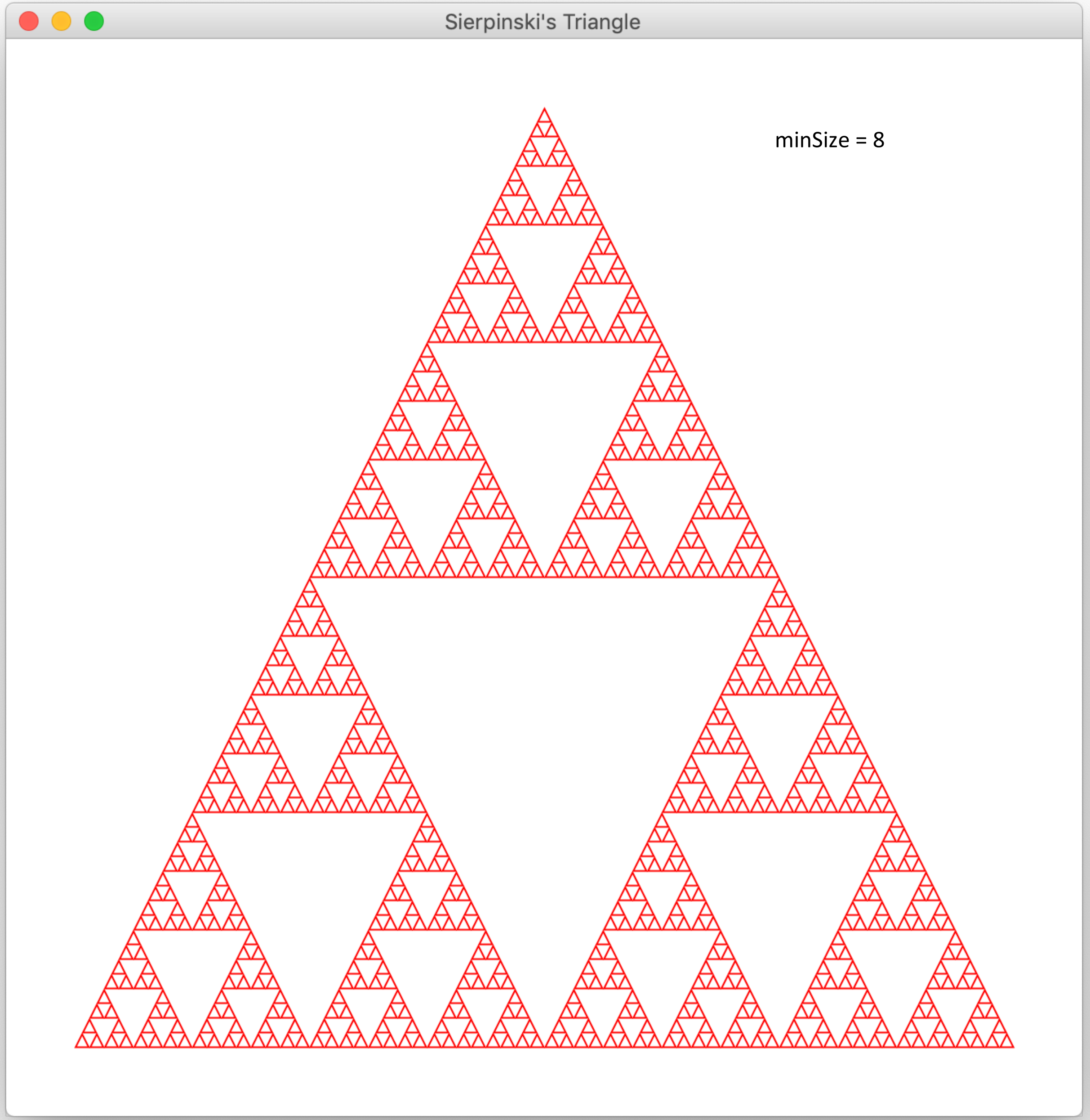
```scala
import doodle.core.Transform.translate
import doodle.core._
import doodle.image._
import doodle.image.syntax._
import doodle.image.syntax.core._
import doodle.java2d._
import doodle.java2d.effect.Frame
```

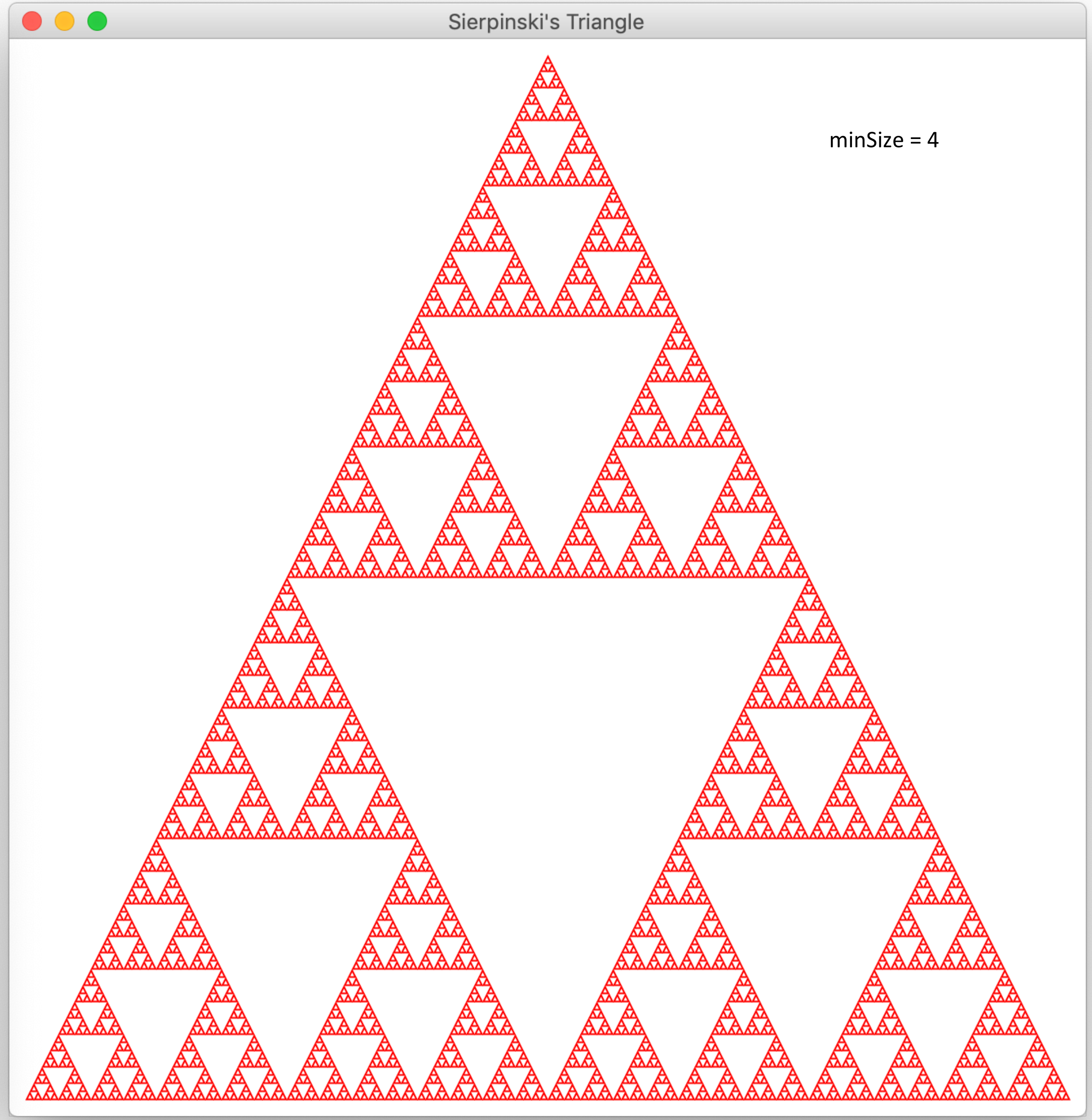

Sierpinski's Triangle
minSize = 512
Sierpinski's Triangle
minSize = 256
Sierpinski's Triangle
minSize = 128
Sierpinski's Triangle
minSize = 64
Sierpinski's Triangle
minSize = 32
Sierpinski's Triangle
minSize = 16
Sierpinski's Triangle
minSize = 8
Sierpinski's Triangle
minSize = 4


Sierpinski's Triangle
minSize = 8


Sierpinski's Triangle
minSize = 4

Just for fun, same as the previous slide, but **without colouring in the triangles**.





```scala
@main def sierpinski: Unit =
  val title = "Sierpinski's Triangle"
  val windowPosition = (0,0)
  val width = 600
  val height = 600
  val backgroundColour = Color.white
  val triangleColour = Color.blue
  val triangleSize = 512
  val triangleXPos = 50
  val triangleYPos = 550
  val minSize = 8

  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Sierpinski")
  frame.setDefaultCloseOperation(
    JFrame.EXIT_ON_CLOSE)
  frame.setBackground(backgroundColour);
  frame.setSize(width, height);

  val sierpinskiTriangle =
    SierpinskiJPanel(
      triangleXPos,
      triangleYPos,
      triangleSize,
      minSize,
      triangleColour
    )
  frame.add(sierpinskiTriangle);
  frame.setVisible(true)
```

For comparison, here are the two **Scala** programs together (the new one is on the right).

@philip_schwarz

```scala
class SierpinskiJPanel(
  x:Int, y:Int, size:Int, minSize:Int, colour:Color)
extends JPanel:

  override def paintComponent(g: Graphics): Unit =
    sierpinskiTriangle(g).unsafeRunSync()

  def sierpinskiTriangle(g: Graphics): IO[Unit] =
    IO{ g.setColor(colour) } >>
    sierpinskiTriangle(g, x, y, size)

  def sierpinskiTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
    if size <= minSize
    then fillTriangle(g, x, y, size)
    else
      val halfSize = size / 2
      sierpinskiTriangle(g, x, y, halfSize) >>
      sierpinskiTriangle(g, x, y - halfSize, halfSize) >>
      sierpinskiTriangle(g, x + halfSize, y, halfSize)

  def fillTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
    val xs = Array(x, x + size, x)
    val ys = Array(y, y, y - size)
    IO{ g.fillPolygon(xs, ys, 3) }
```

```scala
val frameTitle = "Sierpinski's Triangle"
val frameWidth = 660
val frameHeight = 660
val frameBackgroundColour = Color.white
val frame = Frame.size(frameWidth, frameHeight)
                 .title(title)
                 .background(frameBackgroundColour)
val triangleSize = 512
val triangleColour = Color.red
val minSize = 128

def sierpinskiTriangle(size: Int): Image =
  if size <= minSize
  then fillTriangle(size)
  else
    val triangle = sierpinskiTriangle(size / 2)
    triangle above (triangle beside triangle)

def fillTriangle(size: Int): Image =
  Image.triangle(size,size)
       .strokeColor(triangleColour)

@main def sierpinski: Unit =
  sierpinskiTriangle(triangleSize).draw(frame)
```

Deferring side effects with the IO monad

```scala
def sierpinskiTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
  if size <= minSize
  then fillTriangle(g, x, y, size)
  else
    val halfSize = size / 2
    sierpinskiTriangle(g, x, y, halfSize) >>
    sierpinskiTriangle(g, x, y - halfSize, halfSize) >>
    sierpinskiTriangle(g, x + halfSize, y, halfSize)
```

Create three different **IO actions**, each describing the drawing of a triangle, and then **compose** the three into a single **combined IO action**.

```scala
def fillTriangle(g: Graphics, x: Int, y: Int, size: Int): IO[Unit] =
  val xs = Array(x, x + size, x)
  val ys = Array(y, y, y - size)
  IO{ g.fillPolygon(xs, ys, 3) }
```

Rather than immediately drawing a triangle, **suspend** the actual drawing by wrapping the call to `fillPolygon` in an **IO action**.

```scala
sierpinskiTriangle(g).unsafeRunSync()
```

First create an **IO action**, a **pure value**, and then execute the **action**, which has the **side effect** of drawing the triangle.

Deferring side effects with Doodle's 'retained mode'

```scala
def sierpinskiTriangle(size: Int): Image =
  if size <= minSize
  then fillTriangle(size)
  else
    val triangle = sierpinskiTriangle(size / 2)
    triangle above (triangle beside triangle)
```

Create a single `Image`, i.e. a description of a triangle to be drawn, and then use **composition** to create a more complex `Image` by **combining** the `Image` with itself, twice!

```scala
def fillTriangle(size: Int): Image =
    Image.triangle(size,size)
      .strokeColor(triangleColour)
```

Rather than immediately drawing a triangle, return a **description** of the triangle to be drawn.

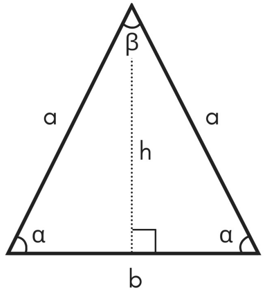
```scala
sierpinskiTriangle(triangleSize).draw(frame)
```

First create an `Image`, a pure value, and then call **`draw`** on the `Image`, which has the **side effect** of drawing the triangle.

Uh-oh, I forgot that when we switched to **Doodle**, we said that we were also going to switch from **isosceles** triangles to **equilateral** ones!

We are currently creating a triangle as follows:

```
Image.triangle(width = size, height = size)
```



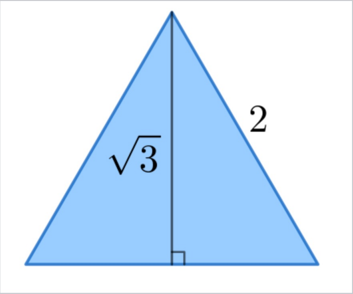
But that creates a triangle with b=`size`; h=`size`; $a = \sqrt{\left(\frac{b}{2}\right)^2 + h^2}$.

i.e. in the resulting triangle, the length b of the base differs from the length a of the other two sides of the triangle.

In order to create a triangle whose three sides are all of the same length b (i.e. $a = b$), we need to specify the correct height h as a function of b, i.e. $h = \frac{\sqrt{3}}{2}b$ ($\sqrt{3}$ is also known as Theodorus' constant).

Let's define a function **heightFromWidth**, which given the width of an **equilateral** triangle, computes its height:

```
object EquilateralTriangle:
  // From https://en.wikipedia.org/wiki/Square_root_of_3:
  // The square root of 3 ...is also known as Theodorus' constant,
  // after Theodorus of Cyrene, who proved its irrationality.
  private val constantOfTheodorus: Double = Math.sqrt(3)

  // From https://en.wikipedia.org/wiki/Equilateral_triangle:
  // The altitude (height) h from any side a is √3÷2×a
  private val widthToHeightMultiplier = constantOfTheodorus / 2

  def heightFromWidth(width: Double): Double =
    widthToHeightMultiplier * width
```

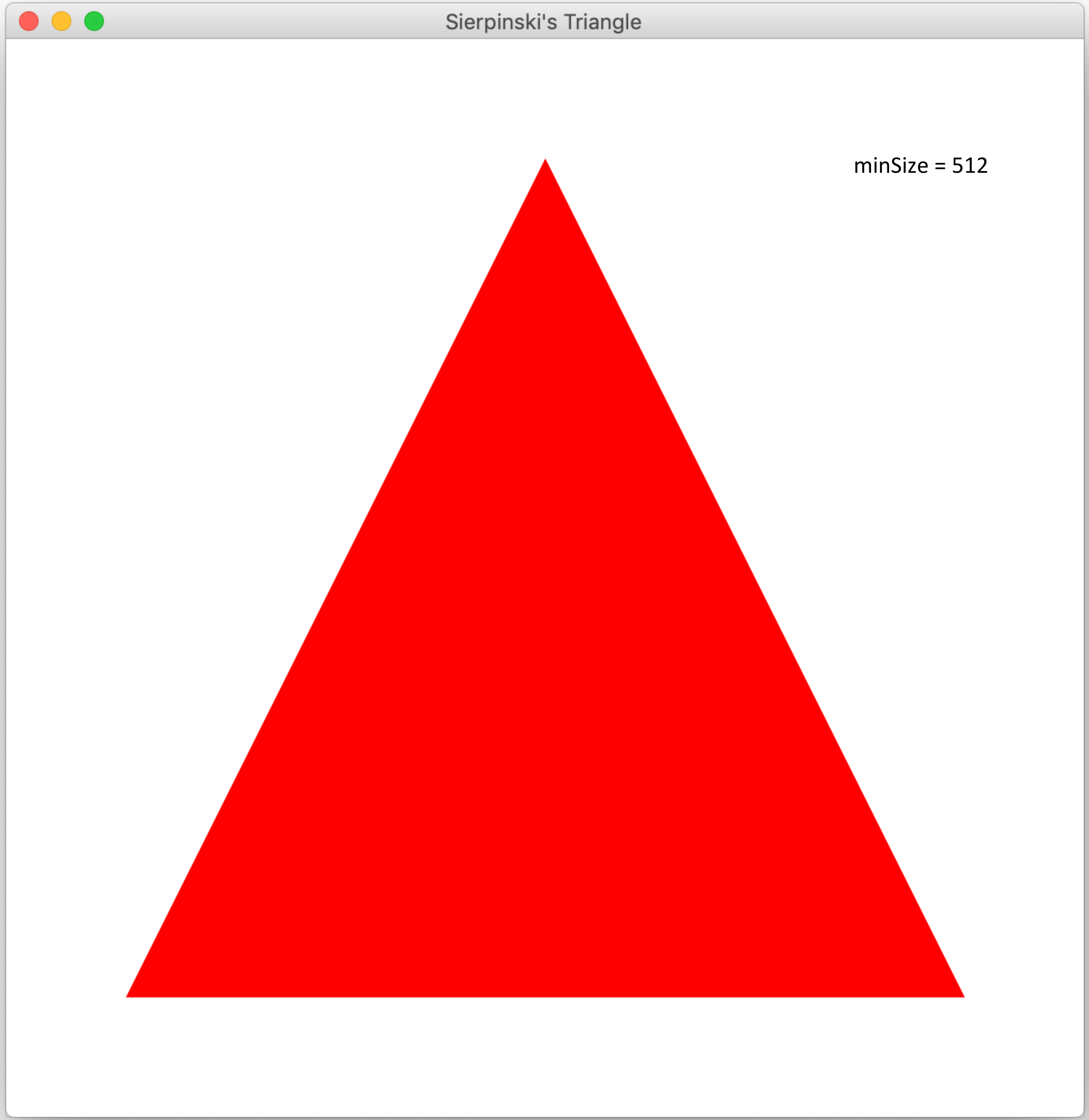
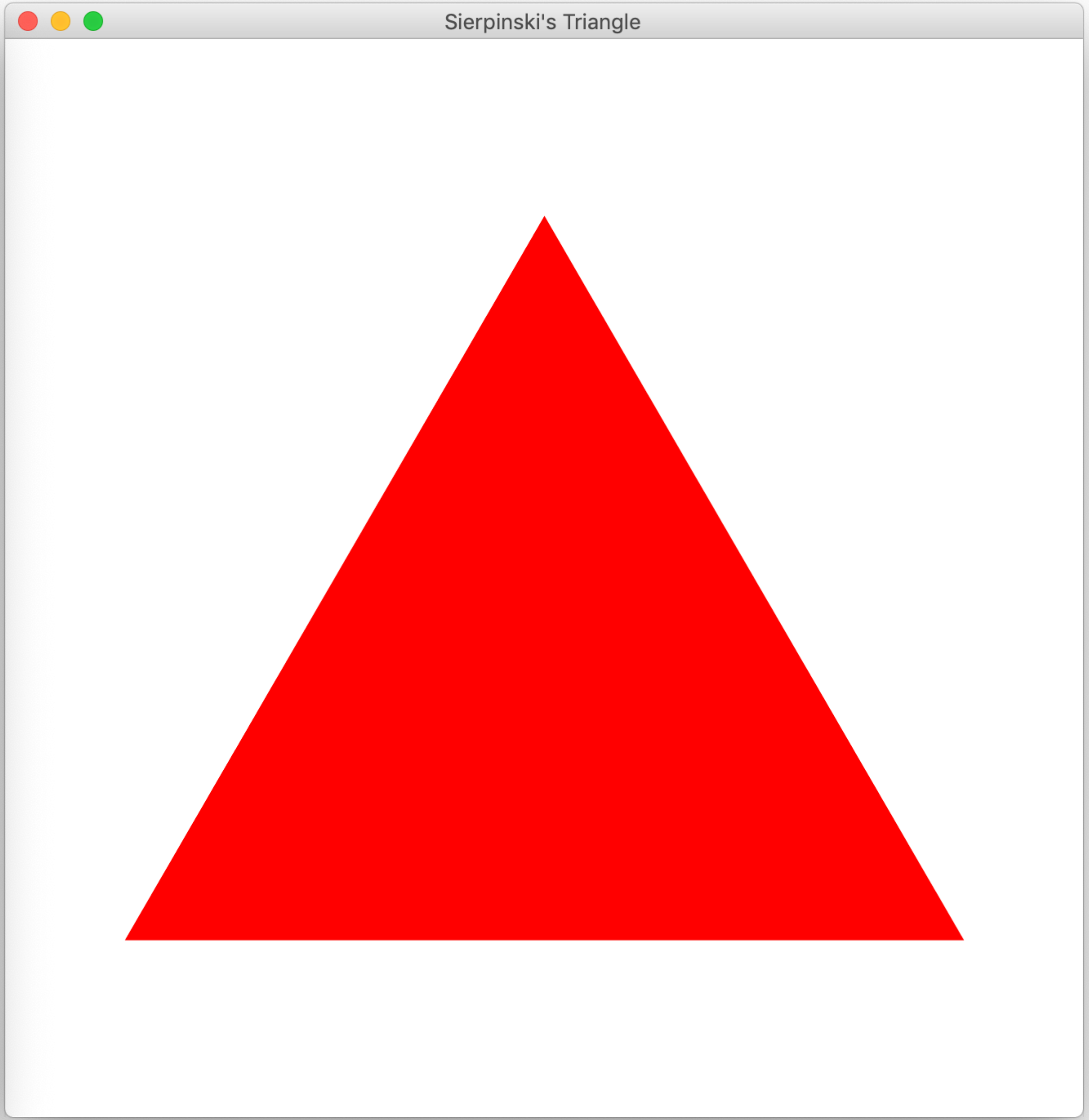


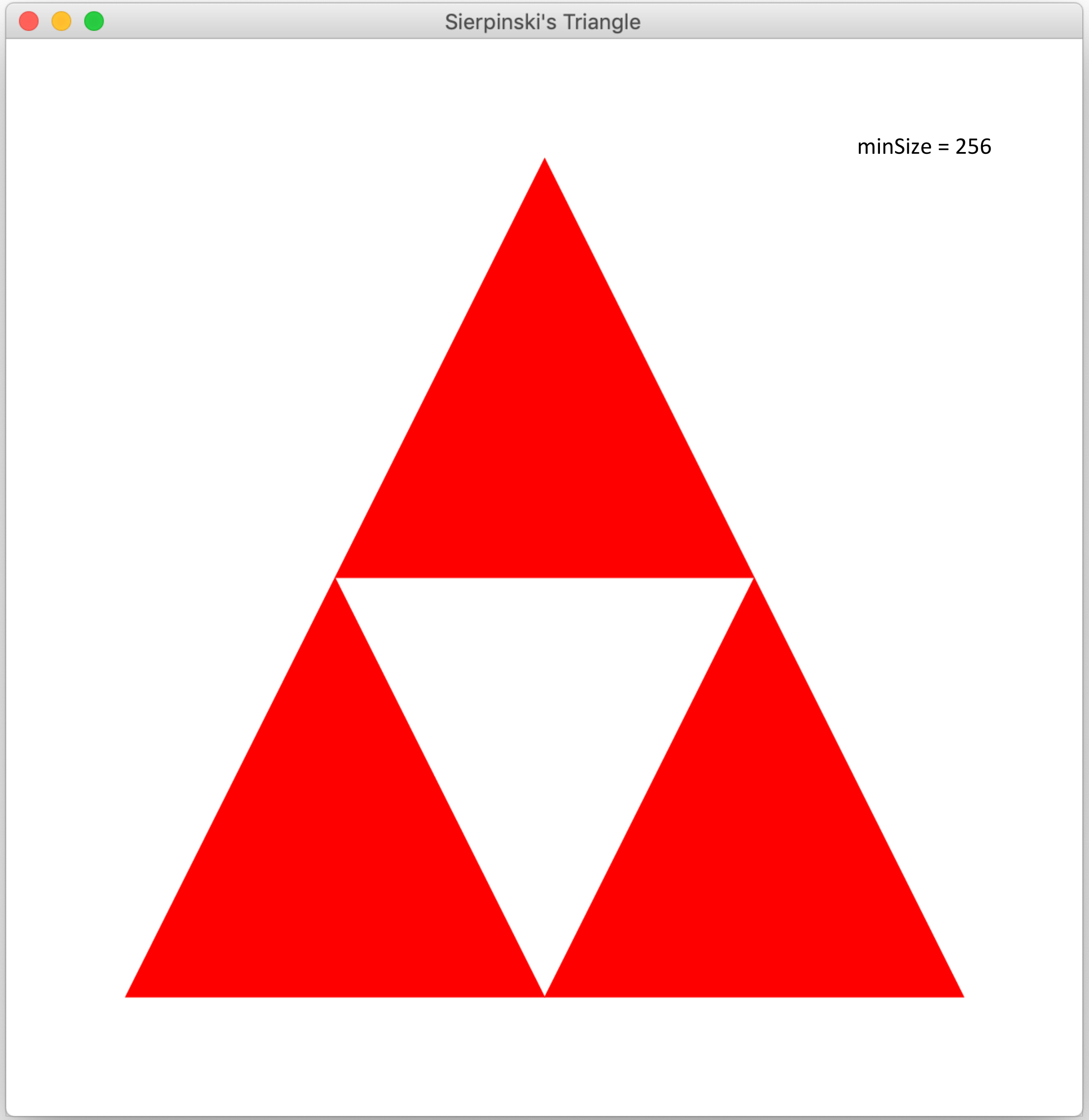
The height of an equilateral triangle with sides of length 2 equals the square root of 3.

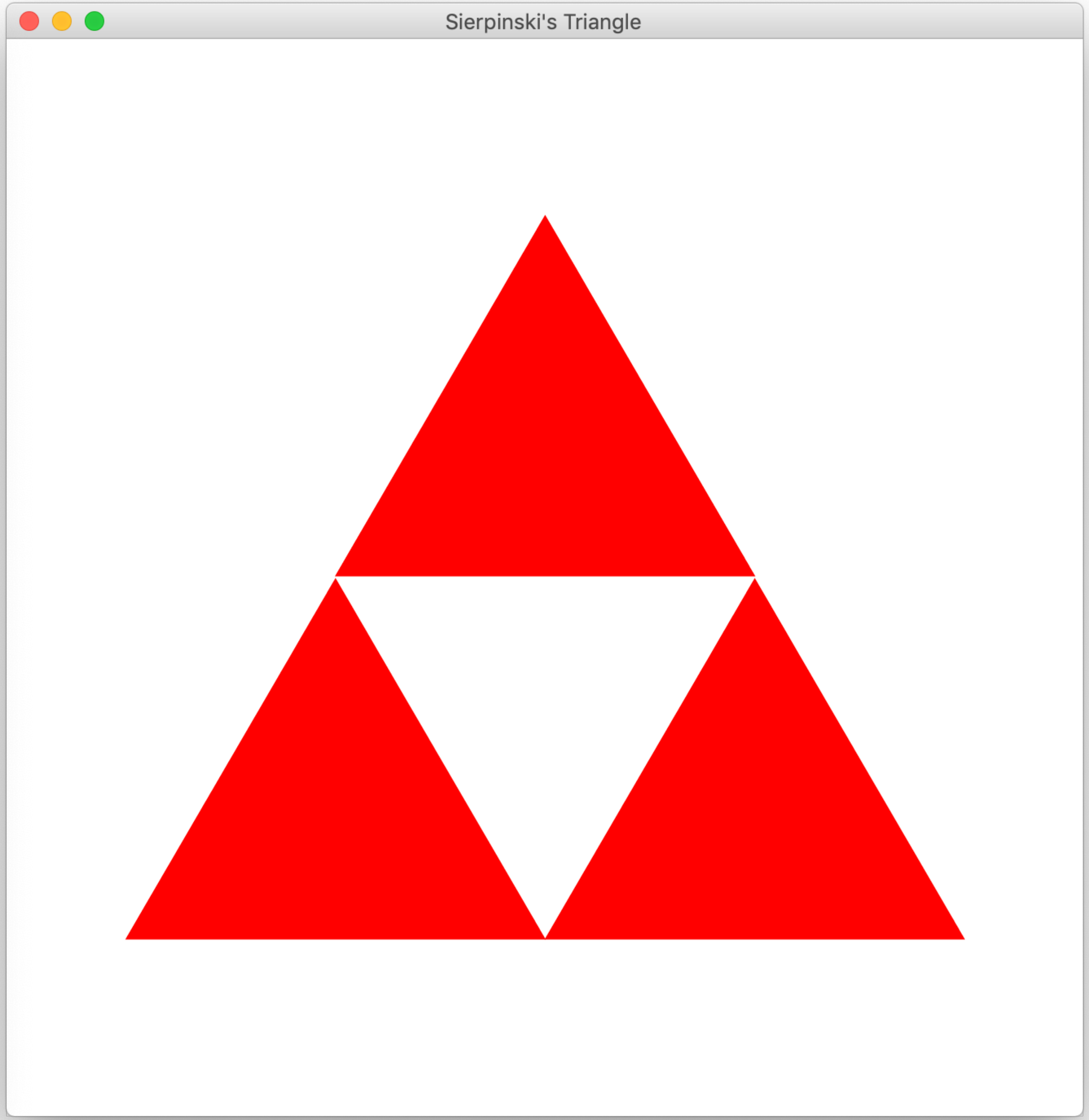
Now we can create an **equilateral** triangle as follows:

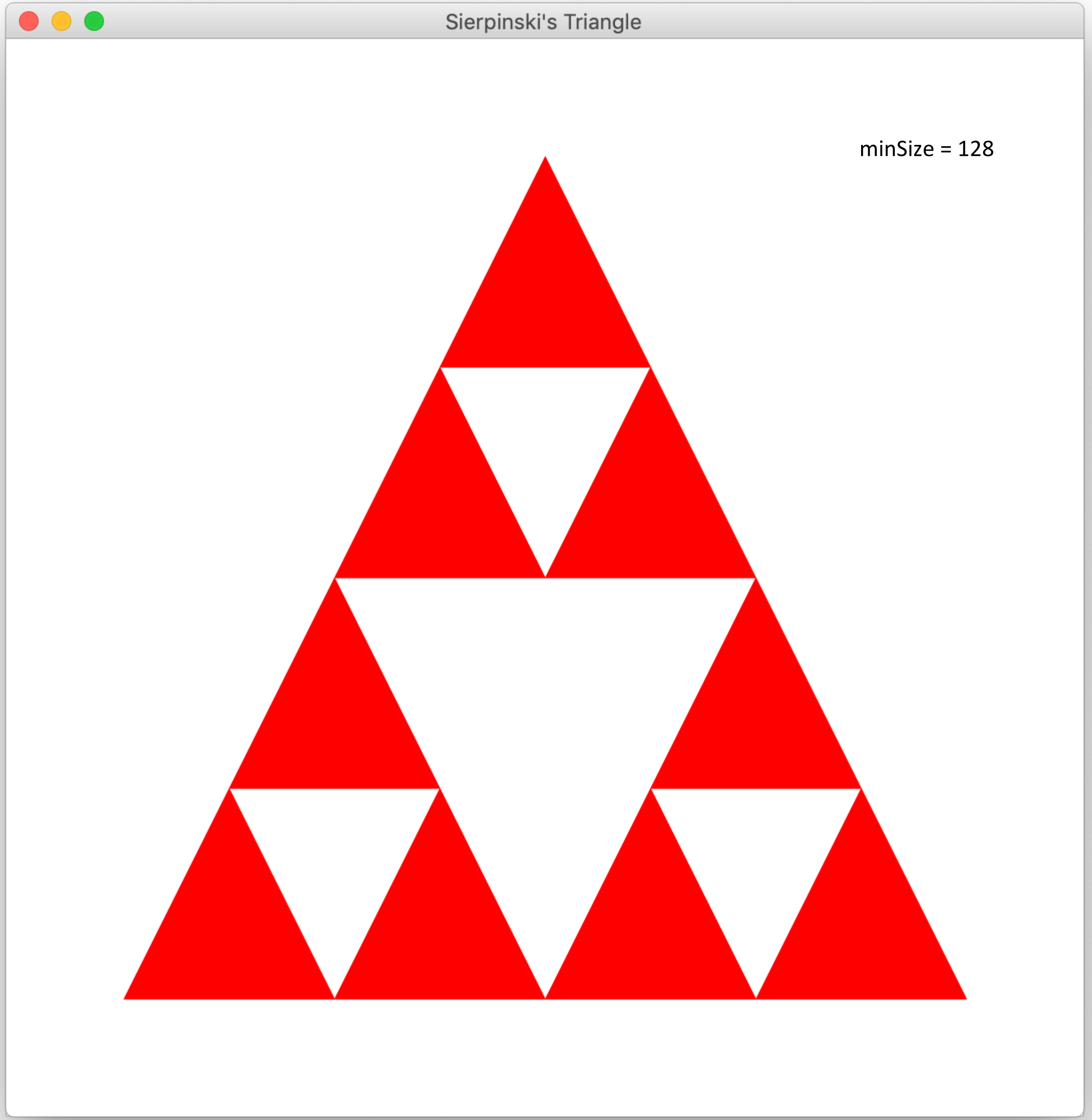
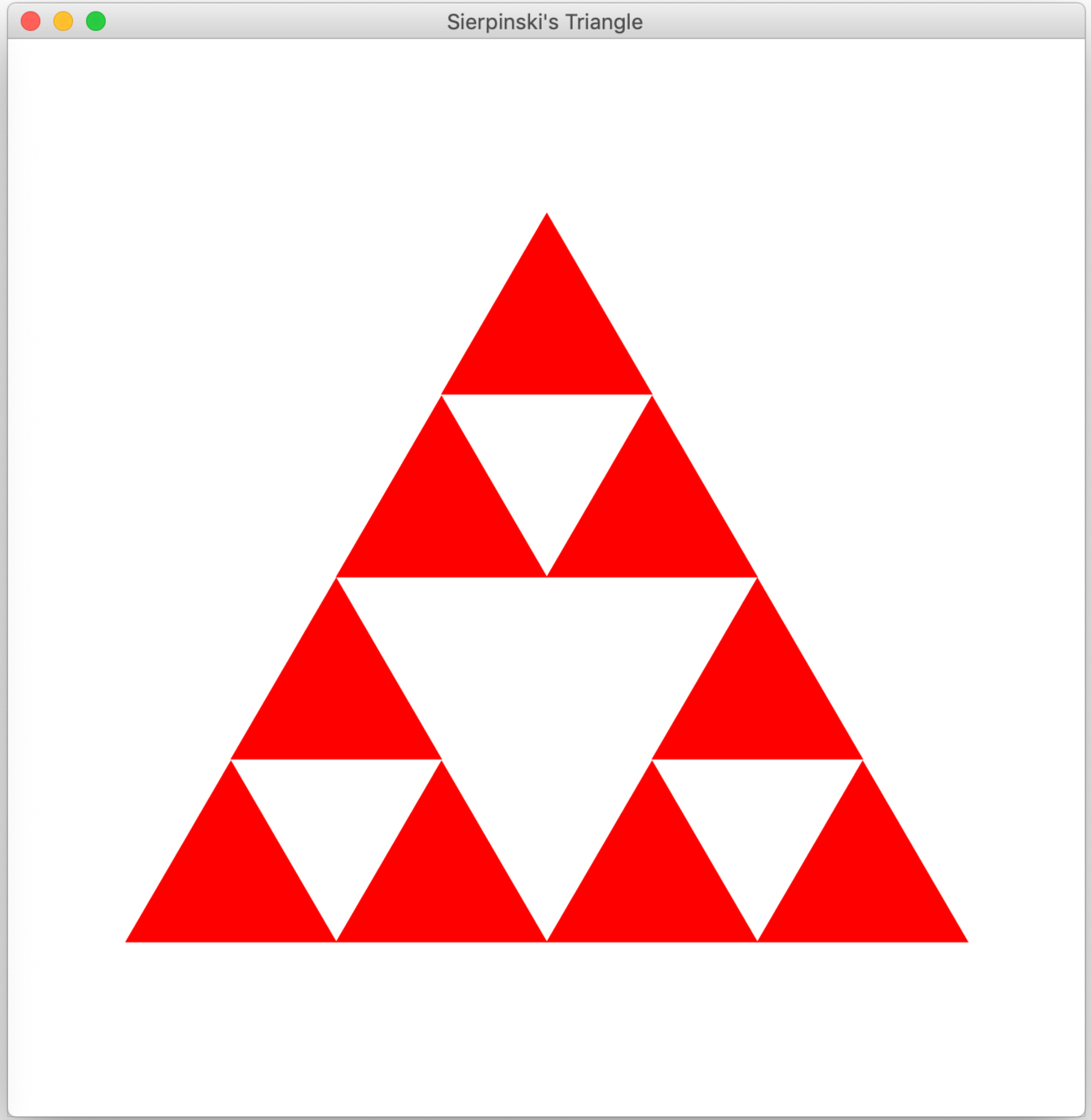
```
Image.triangle(width = size, height = EquilateralTriangle.heightFromWidth(size))
```

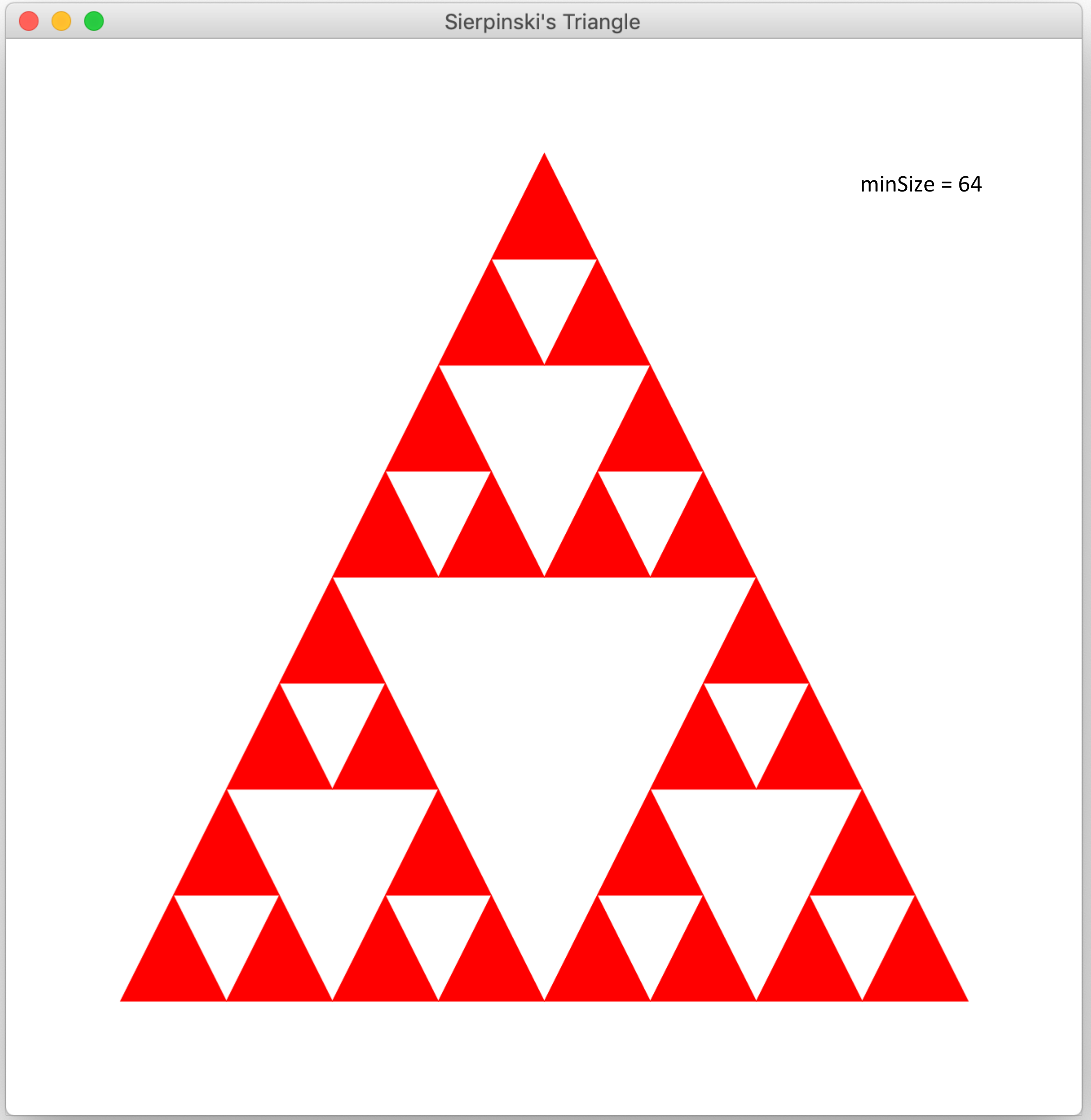
In the final slides of this deck, we just compare the **Sierpinski** triangles drawn using **isosceles** triangles, with those drawn using **equilateral** triangles.

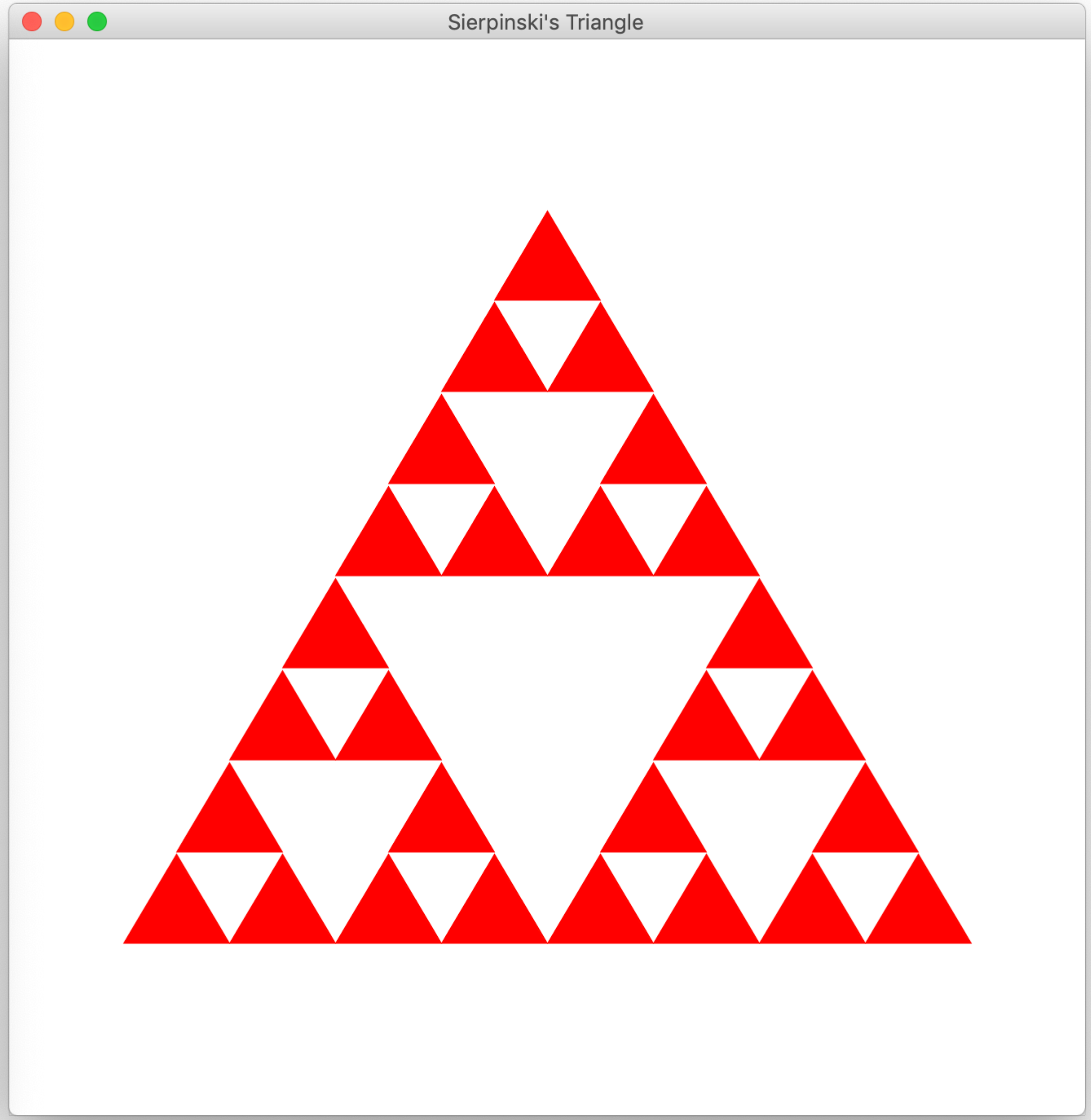

Sierpinski's Triangle
minSize = 512

Sierpinski's Triangle

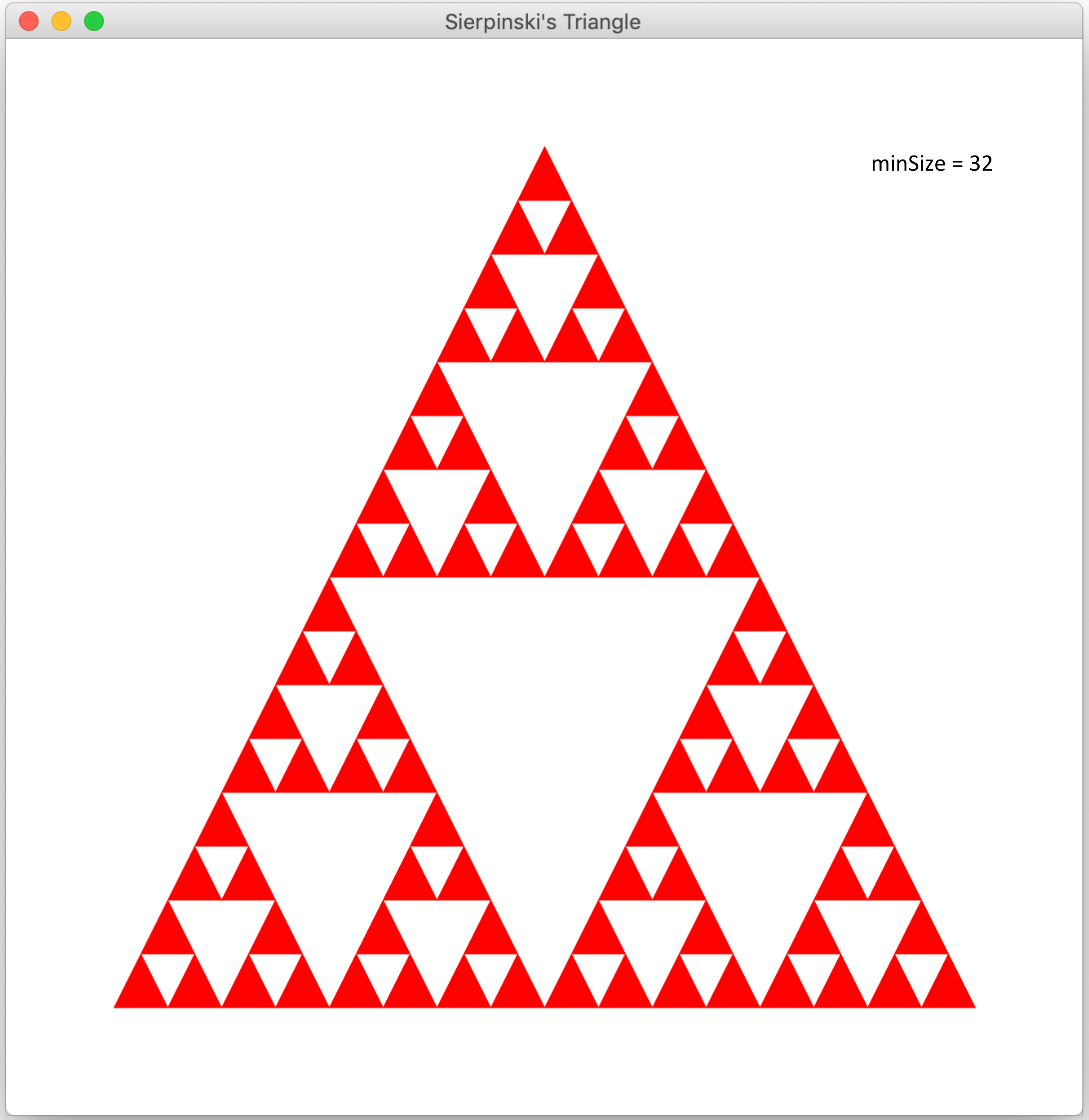

Sierpinski's Triangle
minSize = 256

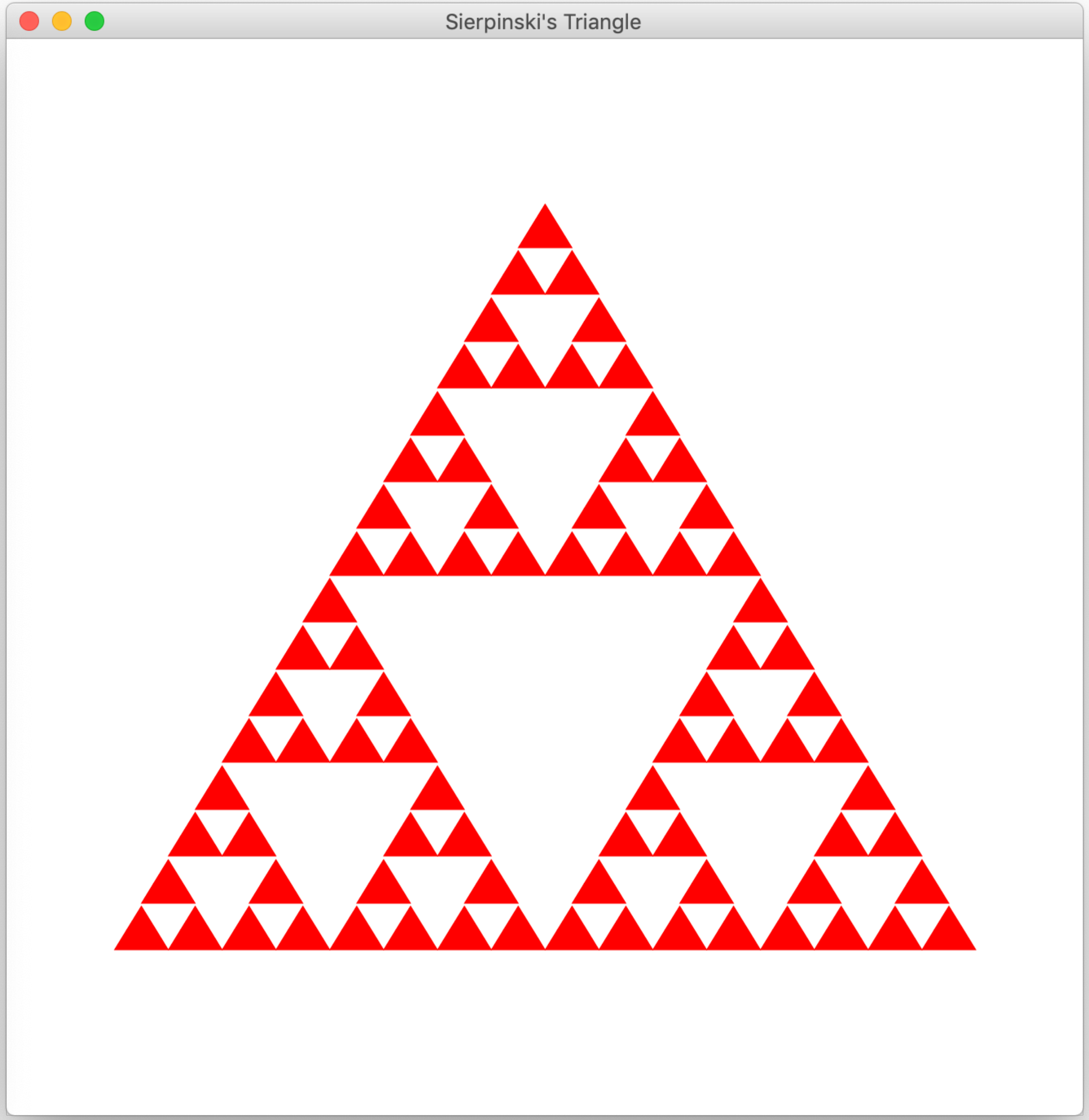

Sierpinski's Triangle

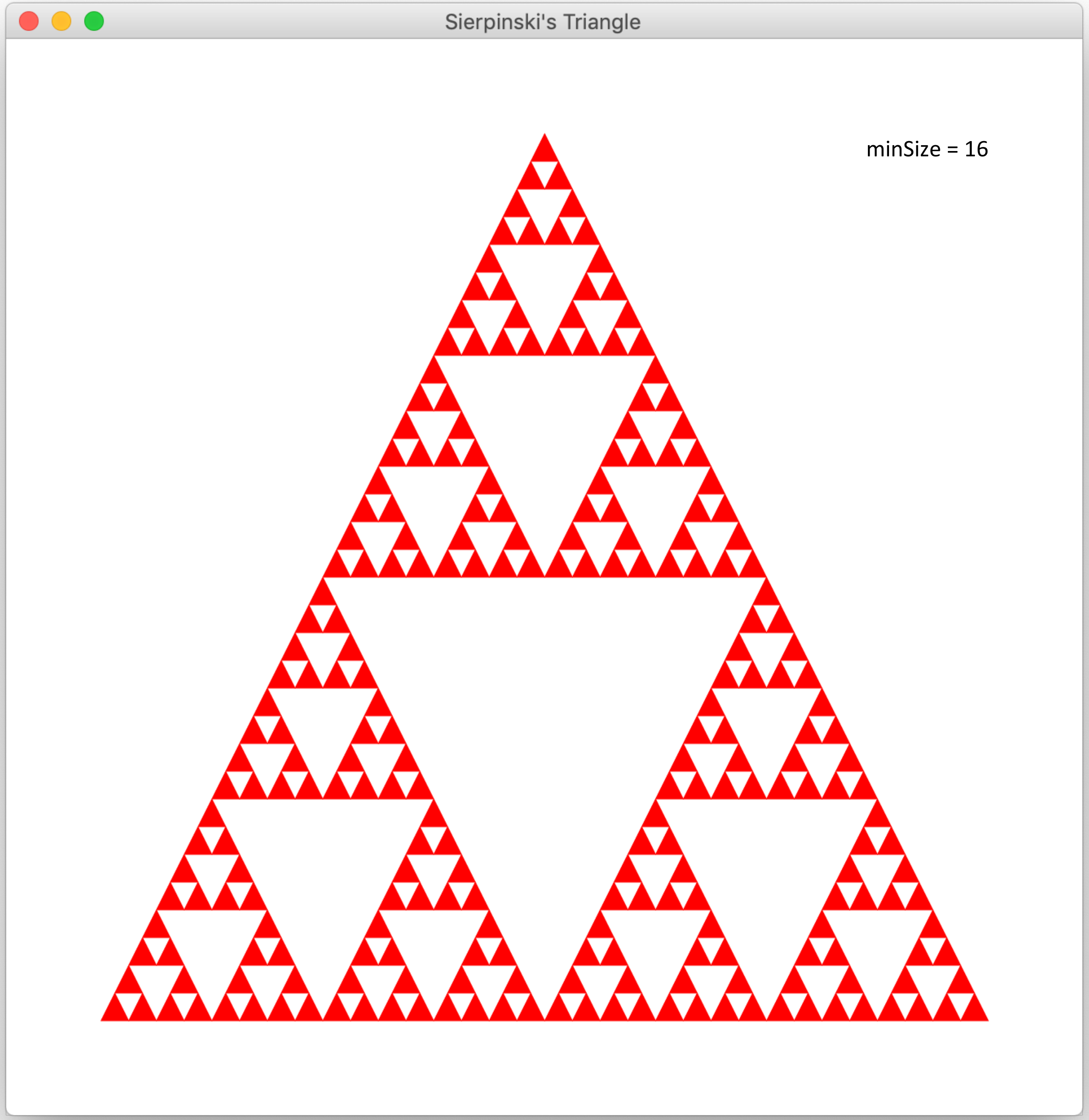

Sierpinski's Triangle
minSize = 128

Sierpinski's Triangle

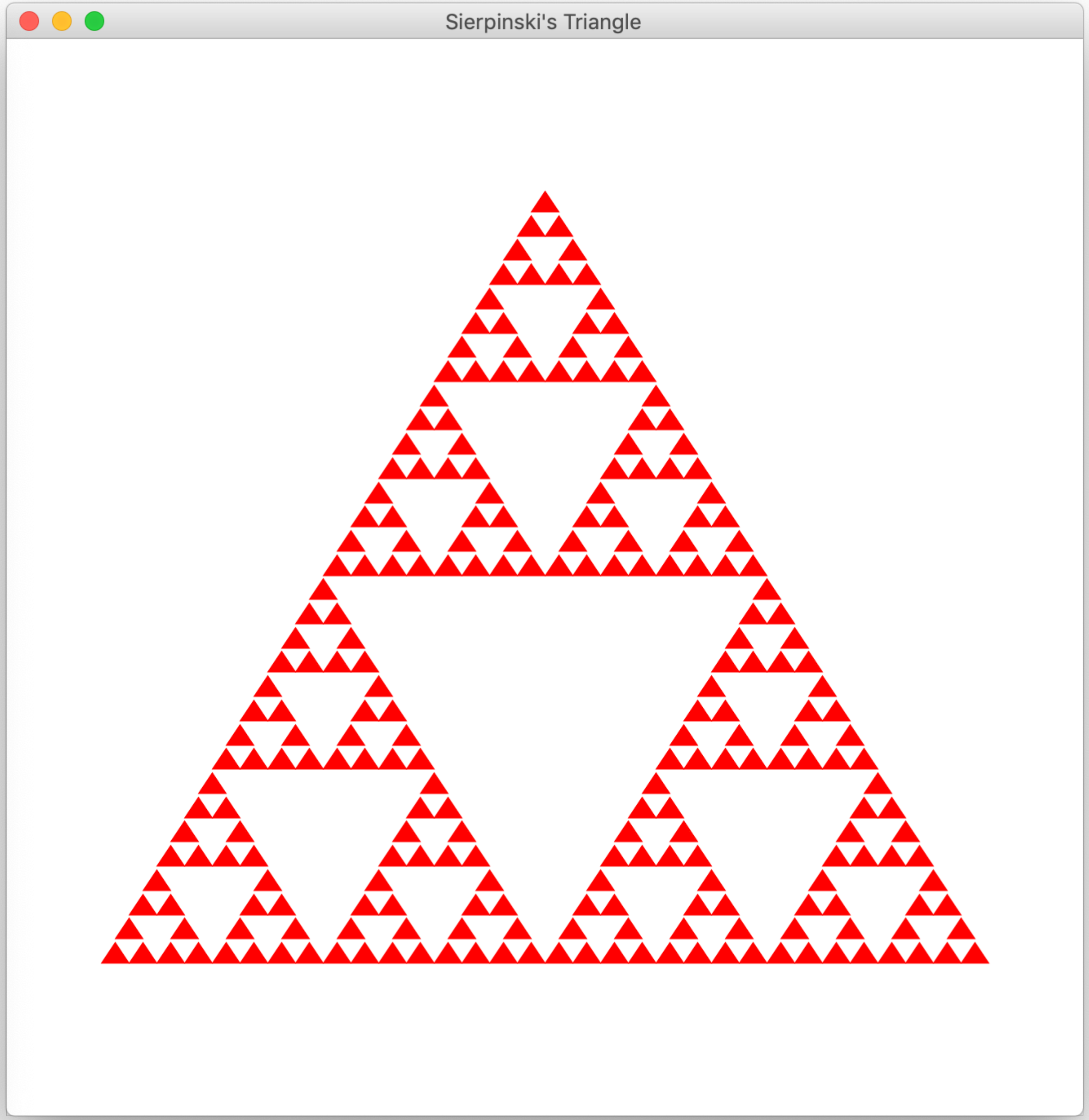

Sierpinski's Triangle
minSize = 64

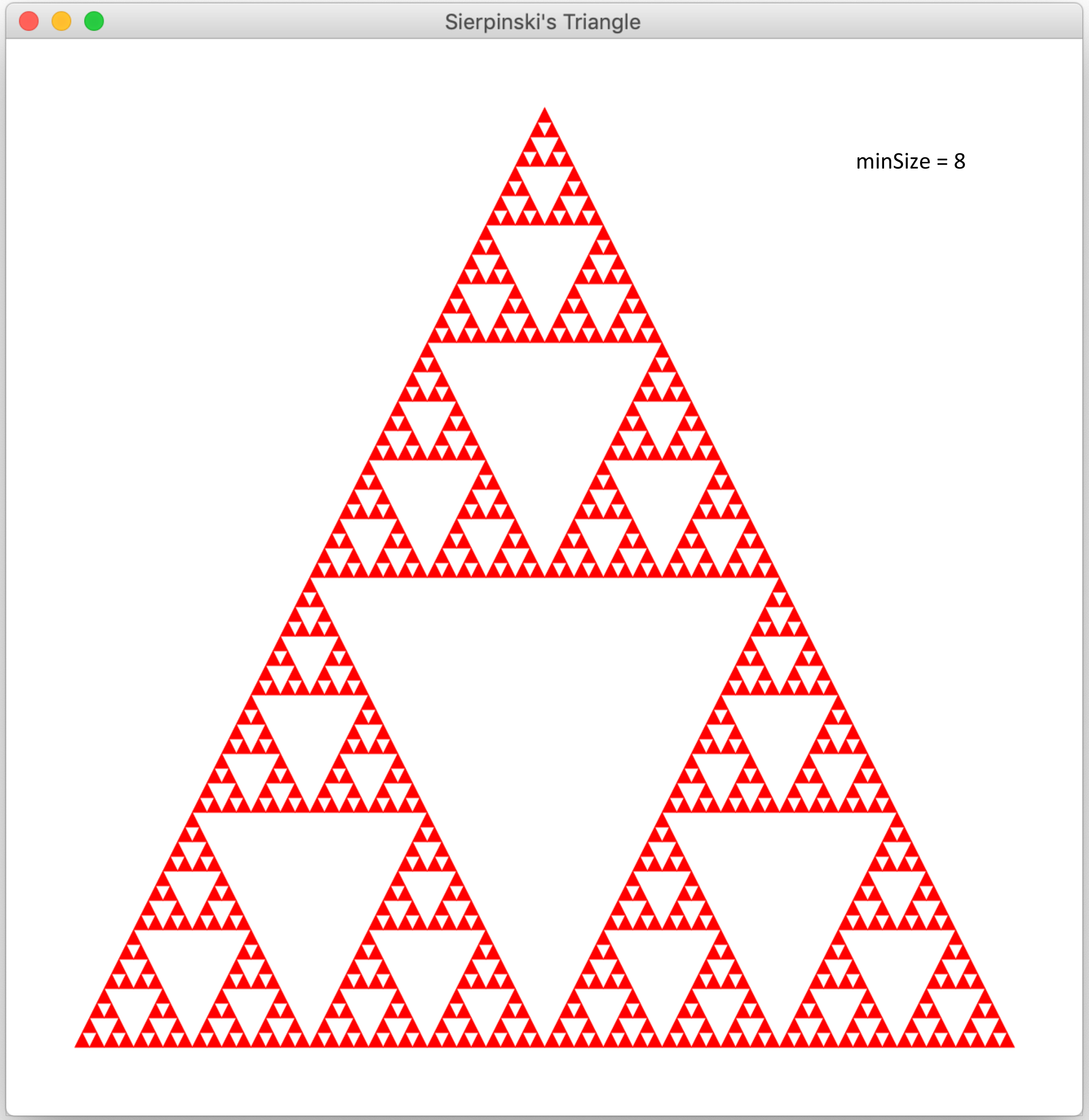

Sierpinski's Triangle

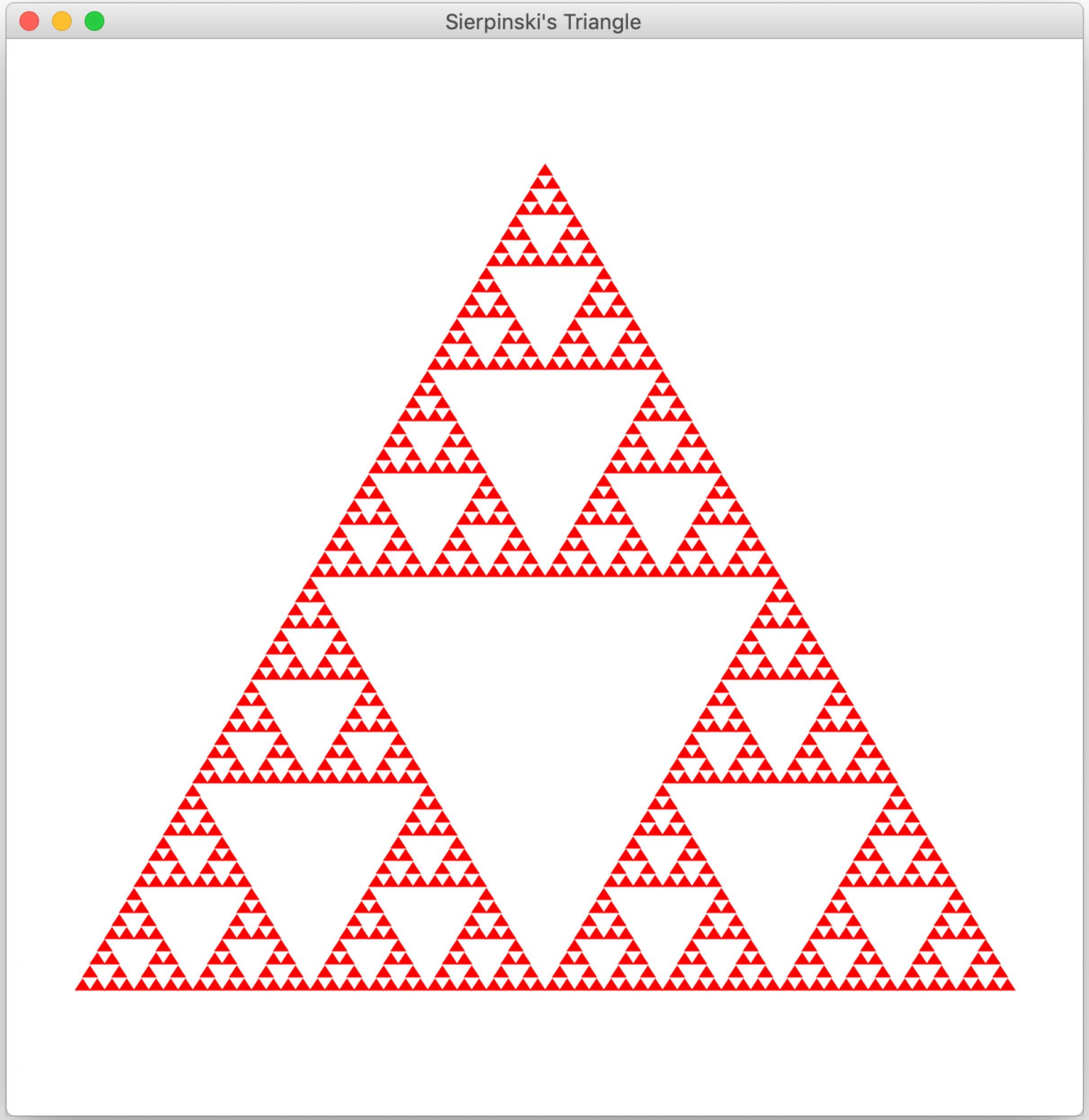

Sierpinski's Triangle
minSize = 32


Sierpinski's Triangle


Sierpinski's Triangle
minSize = 16


Sierpinski's Triangle


Sierpinski's Triangle
minSize = 8


Sierpinski's Triangle


That’s all! I hope you found this slide deck useful.