35
REM VARIATIONS IN PROCESSING

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

Building a high-resolution, interactive program that is inspired by `10 PRINT` allows visual design variations that might not be easy or even possible within a Commodore 64 program. Computational visual art has been created on a variety of platforms and in many systems and languages over the last fifty years, but the last decade has seen an explosion in the use of commercial tools for designers with embedded programming languages (most notably, Adobe Flash) along with programming environments designed by visual artists. John Maeda's Design by Numbers system from 2001 offers one example of the latter; a far more influential tool is Ben Fry and Casey Reas's Processing, itself inspired by Maeda's work and started within his research group at the MIT Media Lab.

While a simple BASIC program writes some text to the screen with `10 PRINT "HELLO WORLD"`, a simple Processing program draws a square to the screen with `rect(20, 30, 80, 60);`. This one-line Processing program draws the rectangle with its upper-left corner at coordinate (20, 30) and with a width of 80 pixels and height of 60 pixels. Essentially, Processing is an image-making programming language that builds on knowledge of geometry, photography, typography, animation, and interaction. Under the hood, Processing is based on Java with a specialized toolkit, program framework, and authoring environment, all suited to the development of interactive visual sketches. Because Processing is situated between programming and the visual arts, it serves as a bridge between two professional cultures. Those who approach Processing with a programming background are encouraged to learn more about making sophisticated visual images. From the other side, visual artists learn the fundamentals of procedural literacy.

The algorithm underlying `10 PRINT` is of course not specific to the Commodore 64; it can be executed with a sheet of graph paper, a pen, and a coin to toss. The act of running the algorithm on a number of different platforms reveals what is essential to the algorithm, on the one hand, and to the specific constraints and affordances of the system on the other: lines produced by `PRINT` wrap and scroll automatically, for instance, so characters can accumulate and fill the screen without being addressed by x and y coordinates. More subtle defaults of the Commodore 64 include the color (light blue on blue) and the speed at which each new section of the maze is added. When `10 PRINT` is ported to another platform, certain features of the Commodore 64 must be defined consciously or at least

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

approximated within the new platform; the programmer can renegotiate the precise color, resolution, and speed of the maze. While many elements and aspects of the original program can be modified in BASIC on the Commodore 64, some are more firmly fixed. Primarily, the 40 × 25 character screen that defines the resolution of the grid is fundamental to the computer's video system and defines the number of units that make up the maze.

The first Processing port of 10 PRINT was written to take advantage of the increased resolution of contemporary screens. It does this by making the thickness and ends of the lines into variables that can be changed while the program runs. The lines of the maze can range in width from 0.5 to 10 pixels, and the lines can terminate with a rounded or square end. Like `10 PRINT`, this Processing port maintains the grid of lines at 40 × 25 units, but, unlike `10 PRINT`, it doesn't add each grid unit in sequence from left to right and top to bottom. In the new high-resolution version, the entire maze is refreshed at once. Using the default Processing colors to create white lines and a black background confers a mood similar to that of the original lower-contrast blues of the Commodore 64 (see figure 35.1).

While creating these variations, some additional quick changes were introduced to explore the visual aspects of the `10 PRINT` maze. First, the 50–50 chance to draw a left or right line was altered so it could be reweighted while the program runs to increase the chance of drawing one line instead of the other. The result is shown in figure 35.2. Then, graphic symbols different from the original diagonal lines were used to expose part of the program's structure.

The optical effect of the maze is created as these diagonals align themselves to produce walls and paths. The viewer's eyes dance across the image as they attempt to find their way through the structure. Some symbols also create a strong, but different optical effect, while other symbols generate a boring, flat graphic. Figure 35.3 shows the result of using a blank image (space) and circle in place of the diagonals. This exploration into applying a different visual skin to the fundamental coin-toss structure of the `10 PRINT` program reveals that the appeal of `10 PRINT` derives from the random choice among two or more elements, the precise selection of which optically activates the viewer to create an interesting and culturally relevant image—in this case, the maze.

The changes just discussed can be explored directly on the Commo-

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

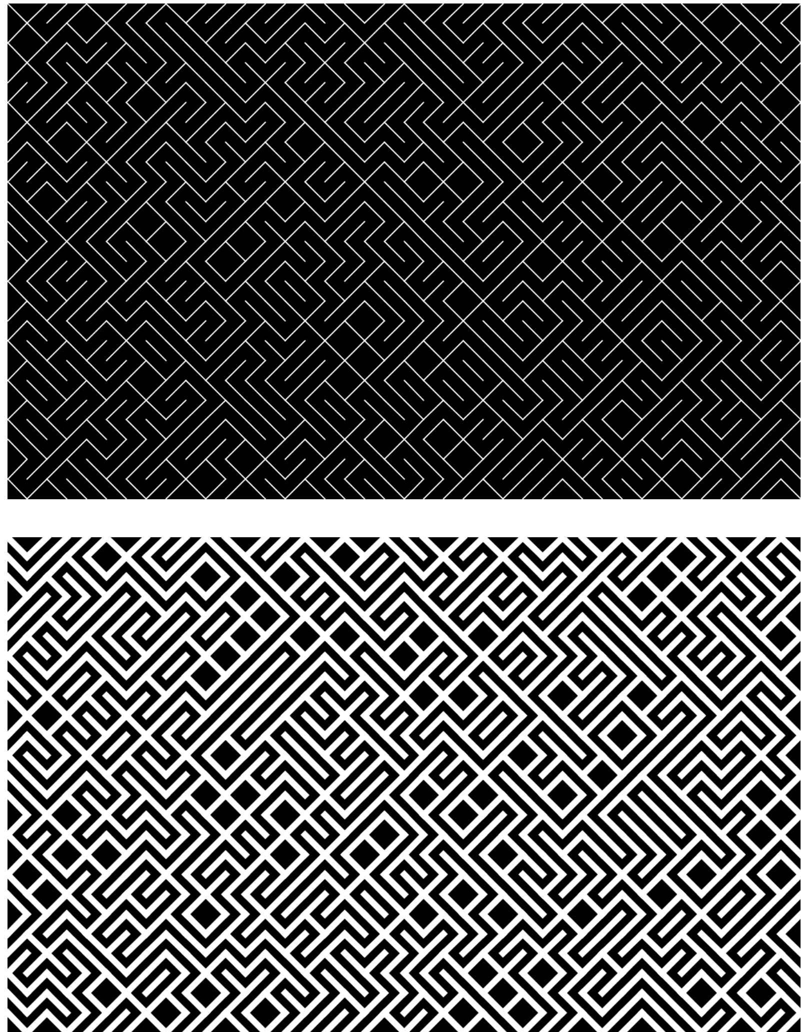


Figure 35.1

Processing ports of `10 PRINT` that explore the effects of changing the line weights and endings.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

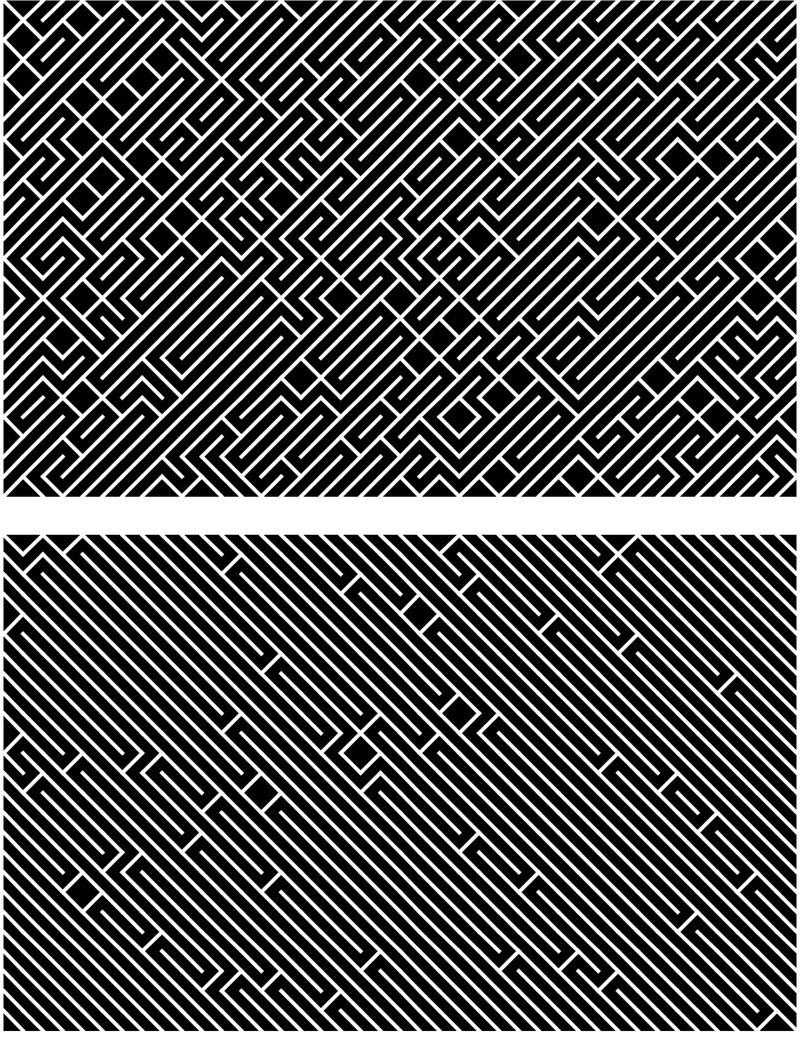


Figure 35.2
Processing ports of `10 PRINT` that explore different weightings for the random values. The top image has a 25 percent chance of drawing a left-leaning diagonal line and the bottom image has a 95 percent chance.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

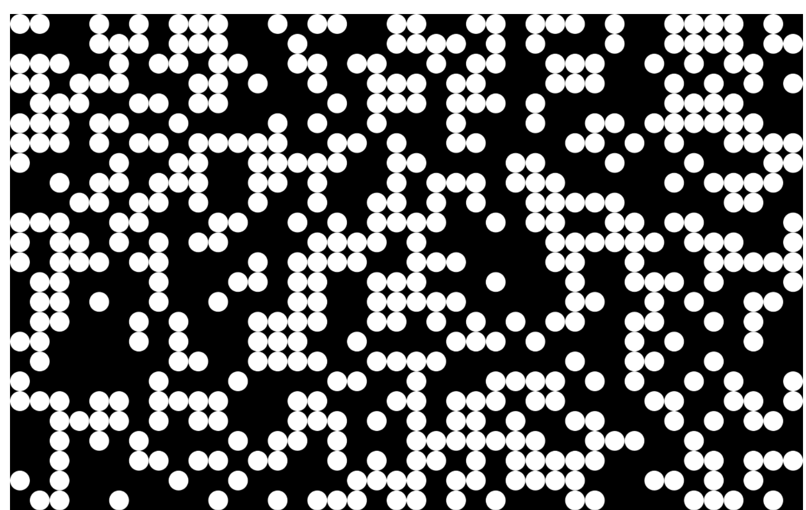


Figure 35.3

Processing port of `10 PRINT` that replaces the lines with circles and blank spaces.

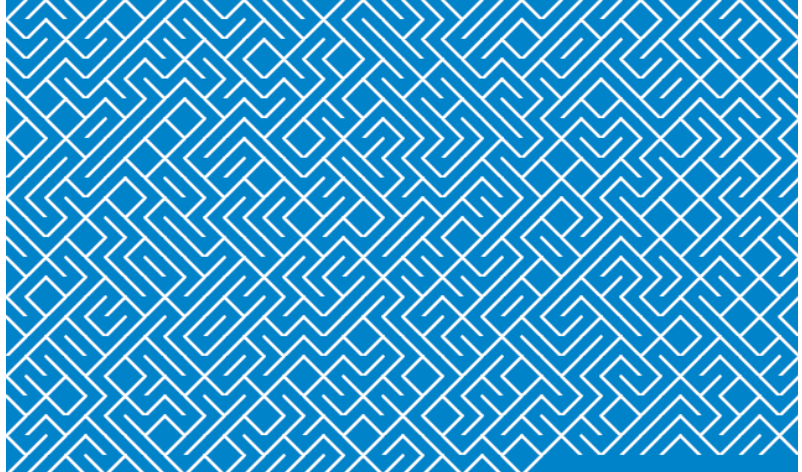


Figure 35.4

Processing port of `10 PRINT` focused on closely imitating the behavior of the Commodore 64.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

dore 64, some more easily and some less so, as has been shown to some extent in the remark Variations in BASIC. It is convenient to explore these changes in Processing, however. For one thing, Processing exposes many dimensions of variation, down to the pixel level, which would be difficult to change on the original platform. For another, a programmer who is highly fluent in Processing can work through ideas and problems easily using that system.

Some variations that are most easily accomplished on the Commodore 64 are the ones involving reweighting the distribution of lines and replacing the lines with spaces and circles. As discussed earlier, `10 PRINT`'s distribution of ╱ and ╲ can be altered by simply changing the ".5" in "205.5," for instance:

```
10 PRINT CHR$(205.25+RND(1)); : GOTO 10
```

The `10 PRINT` variation to show spaces and circles instead of diagonal lines also changes the selection of value 205 or 206 to choose between the numerical code of the space character, 32, and that for a circle character, 113:

```
10 PRINT CHR$(32+(INT(RND(1)+.5)*81)); : GOTO 10
```

Writing and running this first Processing port, which focuses entirely on the image of the maze and on allowing runtime changes in parameters, points out an important visual aspect of the original `10 PRINT`: watching the maze building up or accumulating one unit at a time on the Commodore 64 is a major component of the experience. This behavior doesn't naturally take place with Processing because the entire screen updates at once, not character by character. Processing makes more extensive use of the computer's double-buffered graphics system. This allows graphics to be drawn to an off-screen image buffer, stored in RAM, and, once completed, pushed across to the computer screen. For programs that feature animation or interaction, and thanks to today's much faster hardware, the result is a new image written and drawn to the screen about sixty times per second.

The lines of Processing programs do not begin with numbers, as they do in Commodore 64 BASIC. Each line is executed in order from top to bottom and according to some higher-level rules. This can be seen in the following Processing port of `10 PRINT`. This program does not allow the

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

user to interactively vary parameters, but it does reproduce the character-at-a-time construction of the original:

```
int w = 16;
int h = 16;
int index = 0;

void setup() {
  size(640, 384);
  background(#0000ff);
  strokeWeight(3);
  stroke(224);
  smooth();
}

void draw() {
  int x1 = w*index;
  int x2 = x1 + w;
  int y1 = h*23;
  int y2 = h*24;

  if (random(2) < 1) {
    line(x2, y1, x1, y2);
  } else {
    line(x1, y1, x2, y2);
  }

  index++;
  if (index == width/w) {
    PImage p = get(0, h, width, h*23);
    background(#0000ff);
    set(0, 0, p);
    index = 0;
  }
}
```


Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

The primary structure of the program is defined by the `setup()` and `draw()` blocks. Variables may be defined outside of these blocks, but everything else is sequenced through them. When the program starts, the variables outside of the blocks are declared and assigned. Next, the lines of code inside of `setup()` are read from top to bottom. Here, the size of the display window is set to be 640 pixels wide and 384 pixels high, and the colors for the background and lines are defined. Next, the code inside `draw()` runs from top to bottom. Whatever code is inside the `draw()` block runs, from top to bottom, once for each frame until the program is terminated. The code within the `if` statement, inside `draw()`, samples a random value and then draws one of the two possible lines. The code in the `if` block at the bottom of `draw()` moves the maze up when the maze line that is currently drawing is filled. This code behaves and looks more similar to the canonical Commodore 64 `10 PRINT` (see figure 35.4), but the process is defined differently.

There is a way within Processing to make a `10 PRINT` port that is in some ways a better approximation of the program on the Commodore 64. This method uses the text console of the Processing Development Environment (PDE) instead of the pixels in the display window; this console is typically used for error messages and writing debug statements through the `print()` and `println()` functions, which are similar to `PRINT` in BASIC. The console, however, can only print text; programs themselves are not typed in this area, and graphics cannot be drawn there. The other significant difference is that the graphic characters of PETSCII are not available in the native console font for the PDE. As a result the "/" and "\" (slash and backslash) characters need to be used in place of diagonal graphics and . This results in space between the lines which prevents the illusion of a continuous maze. The output is similar to that of the first `10 PRINT` port in Perl, shown in figure 25.3. A Processing program that produces this approximation of `10 PRINT` can also be realized in one line:

```
void draw() { print((random(1)<0.5) ?'/' :'\\'); }
```

When the program is run in Processing, an empty display window opens and the text is printed to the console as seen in figure 35.5. This exploration raises a crucial difference between writing this program in Commodore 64 BASIC and writing it in Processing. The one-line Processing program is

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

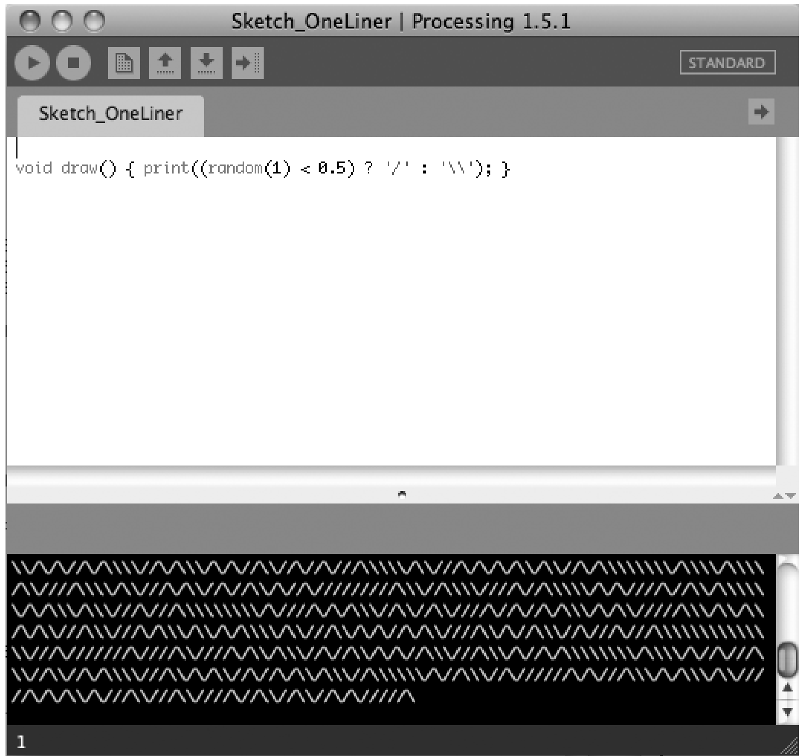


Figure 35.5
This one-line Processing `10 PRINT` port is algorithmically more similar to the Commodore 64 program, but the visual output is extremely different. The code is written in the text editor and the output is drawn to the console rather than opening a new display window.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

quite similar as code but produces a divergent result, one that looks a great deal like that of the first Perl one-liner and the Apple II one-liner discussed in the previous remark.

Evaluating the similarities and differences between the Commodore 64 `10 PRINT` program and the Processing port shows that the shape of the small component lines, and specifically the shape of their ends, is a subtle but crucial factor. In the Commodore 64 `10 PRINT` image, each single-character diagonal line comes to a point on both ends. This is a result of the characters being 8 × 8 pixel tiles with thick lines that run all the way to the corners. This maze is created by tiling these 64-pixel squares.

In the Processing program, each added line is free to extend beyond any particular box within the window. A Processing window is a continuous surface of pixels, each of which can be addressed precisely with an x- and y-coordinate. The `10 PRINT` program comprises 320 × 200 pixels, but the controllable resolution is 40 × 25, for a total of 1000 elements. A Processing version of the program can utilize all of the pixels in a window—and in a screen-filling window on a large, contemporary display, this can mean millions of pixels. A 1080p high-definition display, for example, is composed of 2,073,600 pixels.

With this enhanced resolution in mind, a third version of `10 PRINT` in Processing follows—one that takes more liberties with the original program. The number of rows and columns in the grid is variable, the direction of the line defines its color (black or white), each line is defined as a quadrilateral to give the shape more flexibility, and a third color is used for the background. Each time the code is run, the size of the grid unit is defined at random as a power of 2 (2, 4, 8, 16, or 32), and the thickness of the lines is set randomly to 2, 4, or 8. Figure 35.6 shows some of the varied results. With the ability to further define the graphics, the code becomes longer than a one-liner, but still fairly compact.

The decision to display one direction of lines as white and the other as black triggers the viewer's evolved perception to create depth within this two-dimensional image. The ordinary process of visual perception indicates that there is a light source that is reflecting off one directional edge and creating a shadow on the other. The angle at which the lines terminate in this program enhances the effect by occlusion and termination at the edge. This creates an isometric perspective that further enhances the perceived dimensionality. These effects work well in some randomly determined con-

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

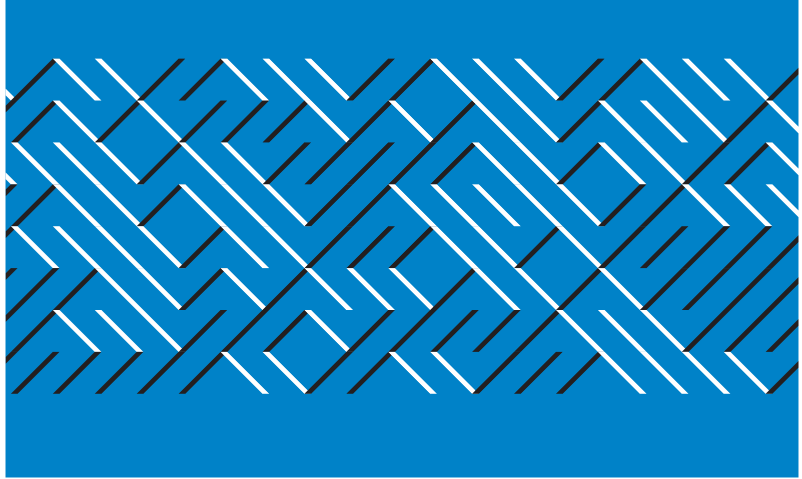
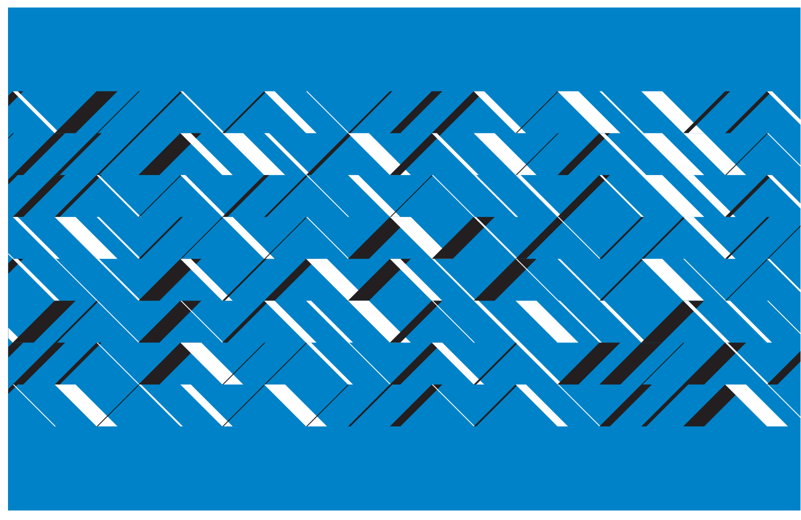


Figure 35.6

Processing port of 10 PRINT that adds a new line shape, colors, and variation of grid units.

 10 PRINT CHR$(205.5+RND(1)); : GOTO 10

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022



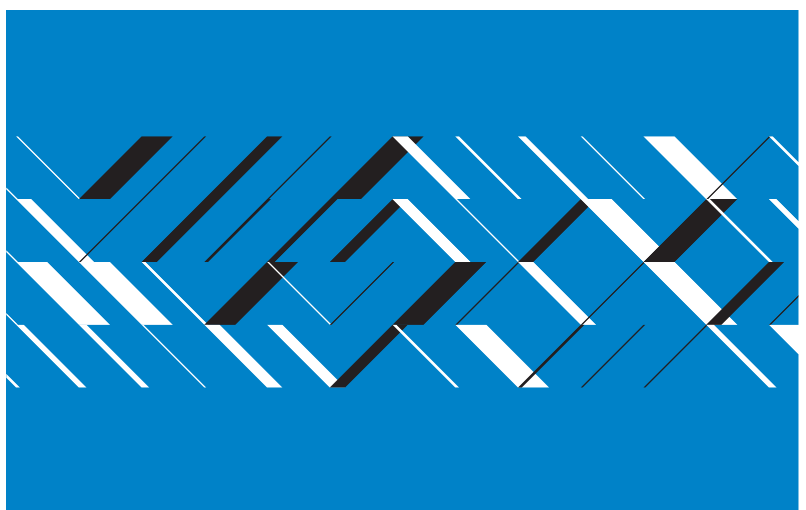


Figure 35.7

Processing program based on `10 PRINT`, but significantly different, in which each line has a random thickness.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022

figurations and are subverted by others (figure 35.6). This is accomplished with the following program:

```
size(1020, 680);
noStroke();
background(0, 0, 255);
int rows = int(pow(2, int(random(1, 6))));
int u = height / (rows + 4);
int thickness = int(pow(2, int(random(1, 4))));
int uth1 = u / thickness;
int uth2 = u + uth1;
int startX = int(-u * 0.75);
int startY = height/2 - rows/2 * u;
int endX = width+u;
int endY = height/2 + rows/2 * u;
for (int x = startX; x < endX; x += u) {
  for (int y = startY; y < endY; y += u) {
    if (random(1) > 0.5) {
      fill(255);
      quad(x, y, x+u, y+u, x+uth2, y+u, x+uth1, y);
    }
    else {
      fill(0);
      quad(x, y+u, x+u, y, x+uth2, y, x+uth1, y+u);
    }
  }
}
```

It is worth noting that, despite all of the random options in the newly defined program, the line weight remains constant throughout. To check the visual effect of selecting a random line weight, one can simply move lines 6–8 of the program (declaring and defining `thickness`, `uth1`, and `uth2`) right underneath the second line beginning with `for`, so they are within that `for` loop. The results are shown in figure 35.7. At this stage, the program distinguishes itself significantly from its parent and emerges as a qualitatively unique algorithm.

Downloaded from http://direct.mit.edu/books/chapter-pdf/2025475/c000900_9780262305501.pdf by guest on 05 December 2022