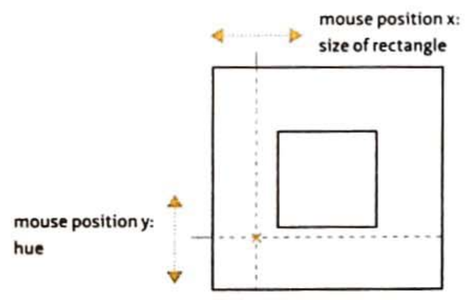
P.1.0

Hello, color

The ability to directly influence 16,777,216 colors gives users an amazing freedom. Simultaneous contrast—without which it would be impossible to perceive colors—is illustrated here by juxtaposing a number of color combinations. Our perception of a color is affected by its neighboring color and the shifting proportions of that color to its background.



→ P_1_0_01.pde

The horizontal position of the mouse controls the size of the color field. Starting in the center, the colored area is depicted with a height and width of 1 to 720 pixels. The vertical mouse position controls the hue. The background passes through the color spectrum from 0 to 360, while the color field passes through the spectrum in the opposite direction, from 360 to 0.

Mouse: *Position x: Size of rectangle* · *Position y: Hue*
Keys: *S: Save PNG* · *P: Save PDF*

→ P_1_0_01.pde

```
void setup() {
  size(720, 720);
  noCursor();
}

void draw() {
  ...
  colorMode(HSB, 360, 100, 100);
  rectMode(CENTER);
  noStroke();
  background(mouseY/2, 100, 100);

  fill(360-mouseY/2, 100, 100);
  rect(360, 360, mouseX+1, mouseX+1);
  ...
}
```

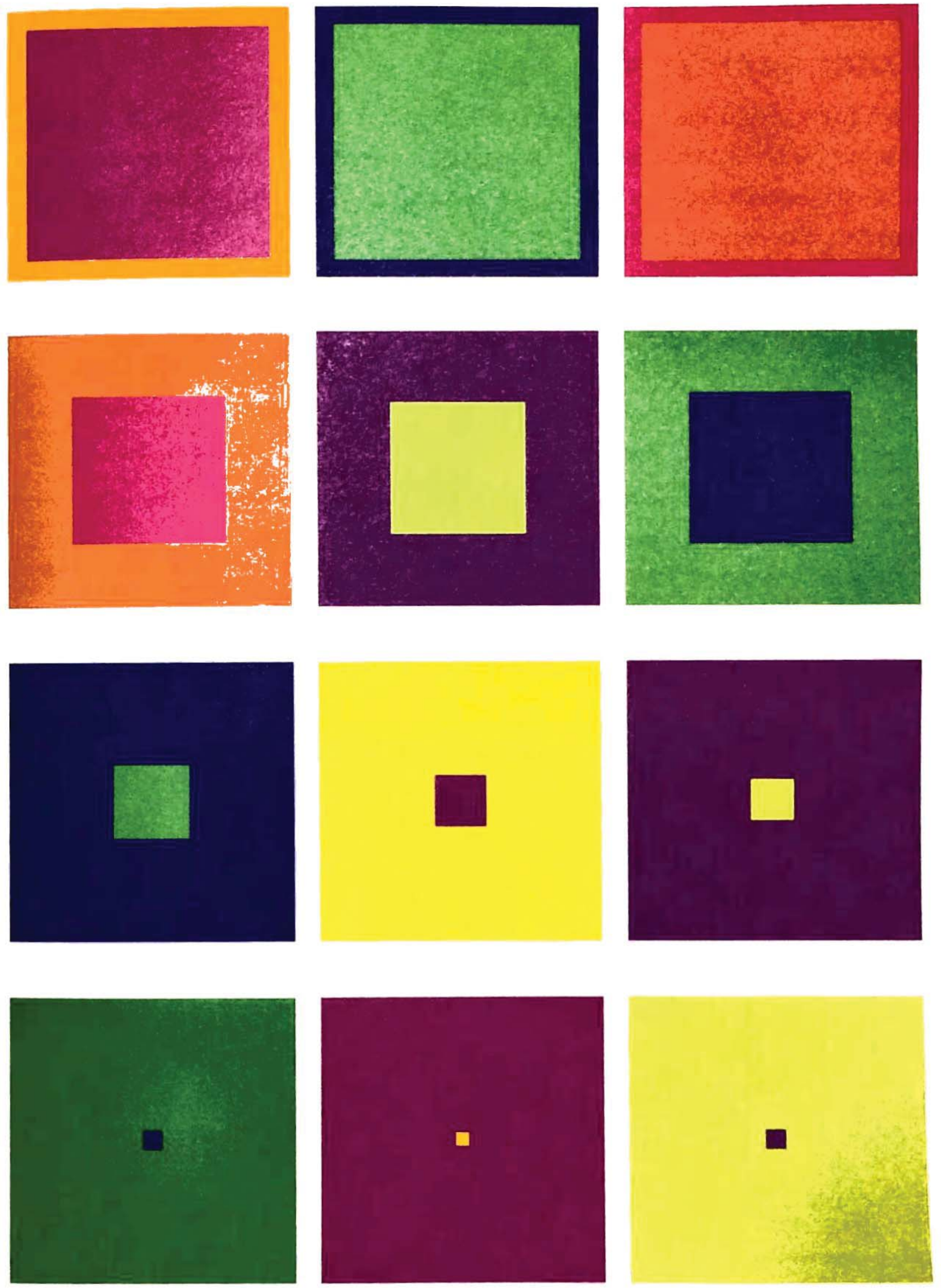
The `setup()` function sets the size of the display window and makes the cursor invisible.

The colors should pass through the hue spectrum in this program. For this reason, `colorMode()` allows users to change the way color value is interpreted. `HSB` specifies the color model, and the three values following it specify the respective range. Hue, for example, can only be specified by values between 0 and 360.

The y-value of the mouse position is divided by 2 to get values from 0 to 360 on the color wheel.

The halved y-value of the mouse position is subtracted from 360, creating values from 360 to 0.

The size of the color field changes relative to the x-value of the mouse position, with a side length between 1 and 720 pixels.

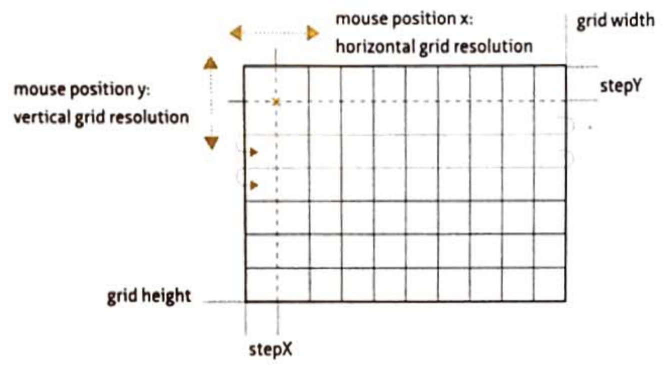


The x-value of the mouse position defines the size of the inside color field, the y-value the hue.
→ P_1_0_01.pde

P.1.1.1

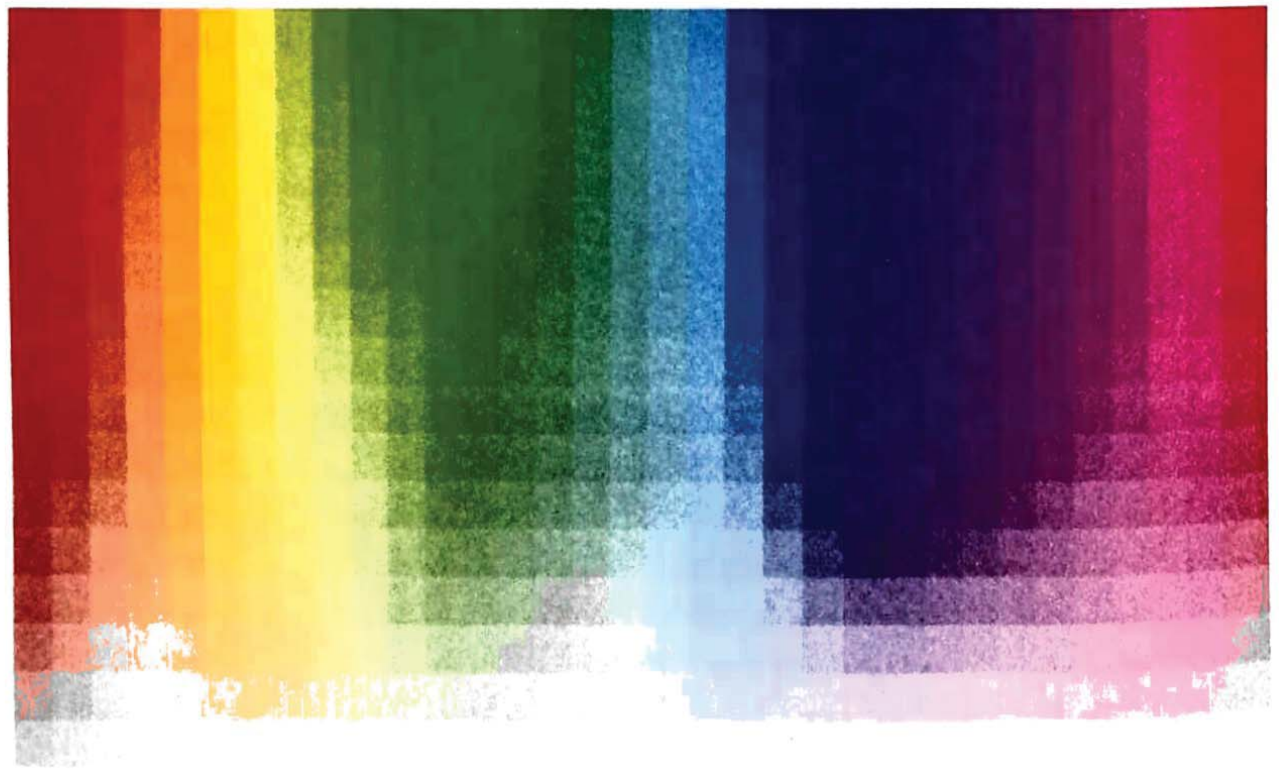
Color spectrum in a grid

This color spectrum is composed of colored rectangles. Each tile is assigned a hue on the horizontal axis and a saturation value on the vertical. The color resolution can be reduced by enlarging the rectangles, so that the primary colors in the spectrum become clearer.



→ P_1_1_1_01.pde

The grid is created by two nested `for` loops. In the outer loop, the y-position is increased, step by step. The inner loop then draws a line by increasing the value for the rectangle's x-position, step by step, until the entire width is processed. The step size is set by the value of the mouse position and is located in the variables `stepX` and `stepY`. It also determines the length and width of the rectangles.



Although the distance of the color values is the same between all the color tiles, the contrasts are perceived as greater in some positions than in others.
→ P_1_1_1_01.pde

Mouse: *Position x/y: Grid resolution*
Keys: *S: Save PNG • P: Save PDF*

→ P_1_1_1_01.pde

```
int stepX;
int stepY;

void setup(){
  size(800, 400);
  background(0);
}


void draw(){
  ...
  colorMode(HSB, width, height, 100);

  stepX = mouseX+2;
  stepY = mouseY+2;

  for (int gridY=0; gridY<height; gridY+=stepY){
    for (int gridX=0; gridX<width; gridX+=stepX){
      fill(gridX, height-gridY, 100);
      rect(gridX, gridY, stepX, stepY);
    }
  }
  ...
}
```

The display size is set using `size()`. The values defined here can be retrieved at any time using the system variables `width` and `height`.

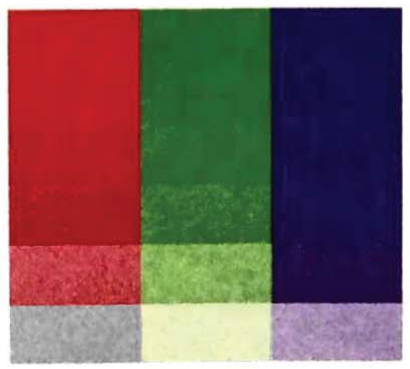
The value range for hue and saturation is set at 800 or 400 using the command `colorMode()`. Hue is no longer defined as a number between 0 and 360 but rather as one between 0 and 800. The same is true of the saturation value.

The addition of 2 prevents `stepX` or `stepY` from being too small, which would lead to longer display times.

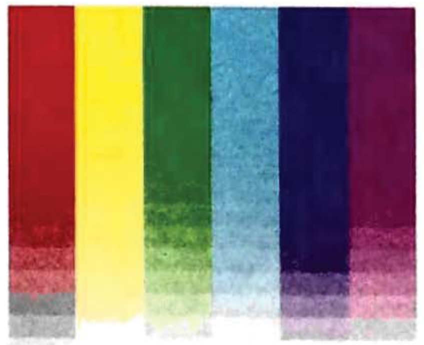
With the help of two nested loops, all the positions in the grid will now be processed. The y-position of the rectangle to be drawn is defined by `gridY` in the outer loop. The value increases only when the inner loop has been processed (i.e., once a complete row of rectangles has been drawn).

The variables `gridX` and `gridY` are used not only to position the tile but also to define the fill color. The hue is determined by `gridX`. Saturation value decreases proportionally to increases in the value `gridY`.

A soft rainbow effect occurs at full resolution.
→ P_1_1_1_01.pde



The primary colors of a computer monitor—red, green, and blue—in various gradations.

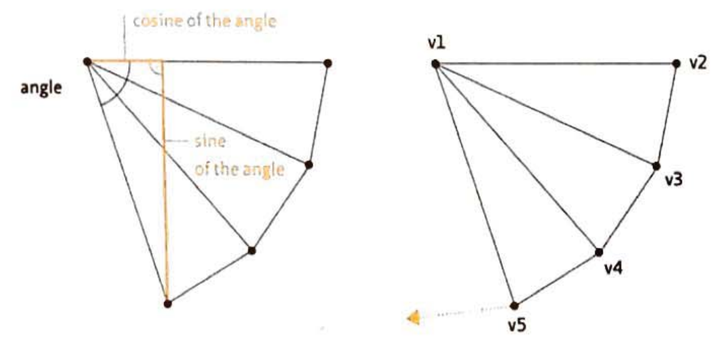


The secondary colors are also visible at this resolution.

P.1.1.2

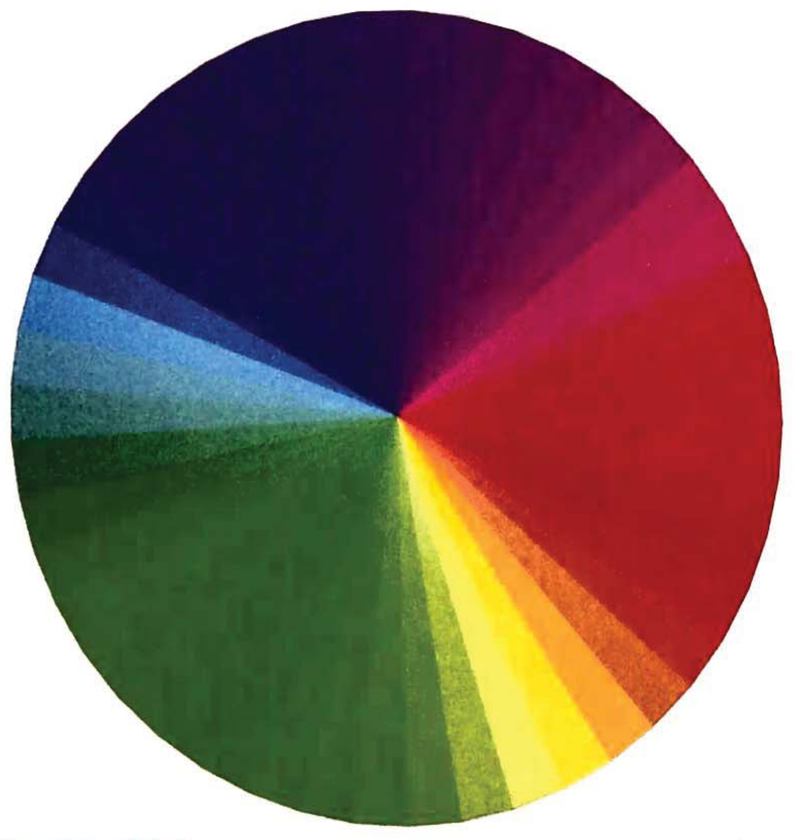
Color spectrum in a circle

There are numerous models for organizing colors. The color spectrum arranged in a circle (color wheel) is a popular model for comparing harmonies, contrasts, and color tones. You can control the number of circle segments, as well as their brightness and saturation values, allowing you to better understand the color arrangement in HSB mode.



→ P_1_1_2_01.pde

The color-wheel segments are arranged in the shape of a fan. The individual vertices are computed from the cosine and sine values of the corresponding angle. A Processing method mode can be used that makes it especially easy to create the wheel segments. The individual points have to be set in the following order: first the middle point, and then the outer ones sequentially.



Segment count 45, key 2
→ P_1_1_2_01.pde

Segment count 12, key 4
→ P_1_1_2_01.pde

Segment count 6, key 5
→ P_1_1_2_01.pde

Mouse: *Position x: Saturation • Position y: Brightness*
Keys: *1-5: Segment count • S: Save PNG • P: Save PDF*

→ P_1_1_2_01.pde

```
void draw(){
  ...
  colorMode(HSB, 360, width, height);
  background(360);

  float angleStep = 360/segmentCount;

  beginShape(TRIANGLE_FAN);
  vertex(width/2, height/2);
  for (float angle=0; angle<=360; angle+=angleStep){
    float vx = width/2 + cos(radians(angle))*radius;
    float vy = height/2 + sin(radians(angle))*radius;
    vertex(vx, vy);
    fill(angle, mouseX, mouseY);
  }
  endShape();
  ...
}
```

The ranges of values for saturation and brightness are adjusted in such a way that mouse coordinates can be taken as their values.

The angle increment depends on how many segments are to be drawn (segmentCount).

The first vertex point is in the middle of the display.

For the other vertices, angle has to be converted from degrees (0–360) to radians (0–2π) because the functions cos() and sin() require the angle be input this way.

The fill color for the next segment is defined: the value of angle as hue, mouseX as saturation, and mouseY as brightness.

The construction of the color segment is ended with endShape().

```
void keyReleased(){
  ...
  switch(key){
  case '1':
    segmentCount = 360;
    break;
  case '2':
    segmentCount = 45;
    break;
  case '3':
    segmentCount = 24;
    break;
  case '4':
    segmentCount = 12;
    break;
  case '5':
    segmentCount = 6;
    break;
  }
}
```

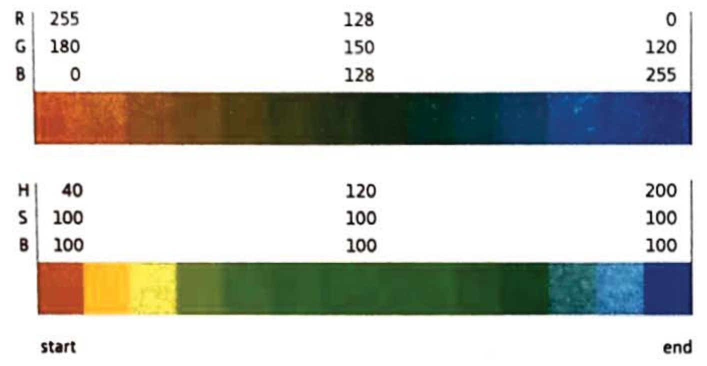
The switch() command checks the last key pressed, which enables easy switching between different cases.

If the key 3 was pressed, for instance, then segmentCount is set at the value 24.

P.1.2.1

Color palettes through interpolation

In each color model, colors have their clearly defined place. The direct path from one color to another always has precisely definable gradations, which will vary depending on the specific model. Using this interpolation you can create color groups in every gradation, as well as locate individual intermediate nuances.



Because a color is not defined by a single number but by several values, it is necessary to interpolate between these values. Depending on the chosen color model, RGB or HSB, the same color is defined by different values, thereby causing the path from one to another to lead past different colors. In the HSB color model, for instance, a detour is made past the color wheel. This difference is due to the characteristics of the color models, both of which can be very useful, depending on the situation. It is therefore important to choose the appropriate color model to solve a specific problem.

Mouse: *Left click: New random color set • Position x: Resolution • Position y: Number of rows*
Keys: *1–2: Interpolation style • S: Save PNG • P: Save PDF • C: Save ASE palette*

```
void draw() {
  ...
  tileCountX = (int) map(mouseX,0,width,2,100);
  tileCountY = (int) map(mouseY,0,height,2,10);
  float tileWidth = width / (float)tileCountX;
  float tileHeight = height / (float)tileCountY;
  color interCol;
  ...
  for (int gridY=0; gridY< tileCountY; gridY++) {
    color col1 = colorsLeft[gridY];
    color col2 = colorsRight[gridY];
    for (int gridX=0; gridX< tileCountX; gridX++) {
      float amount = map(gridX,0,tileCountX-1,0,1);
      ...
      interCol = lerpColor(col1,col2, amount);
      ...
      fill(interCol);
      float posX = tileWidth*gridX;
      float posY = tileHeight*gridY;
      rect(posX, posY, tileWidth, tileHeight);
      ...
    }
  }
}
```

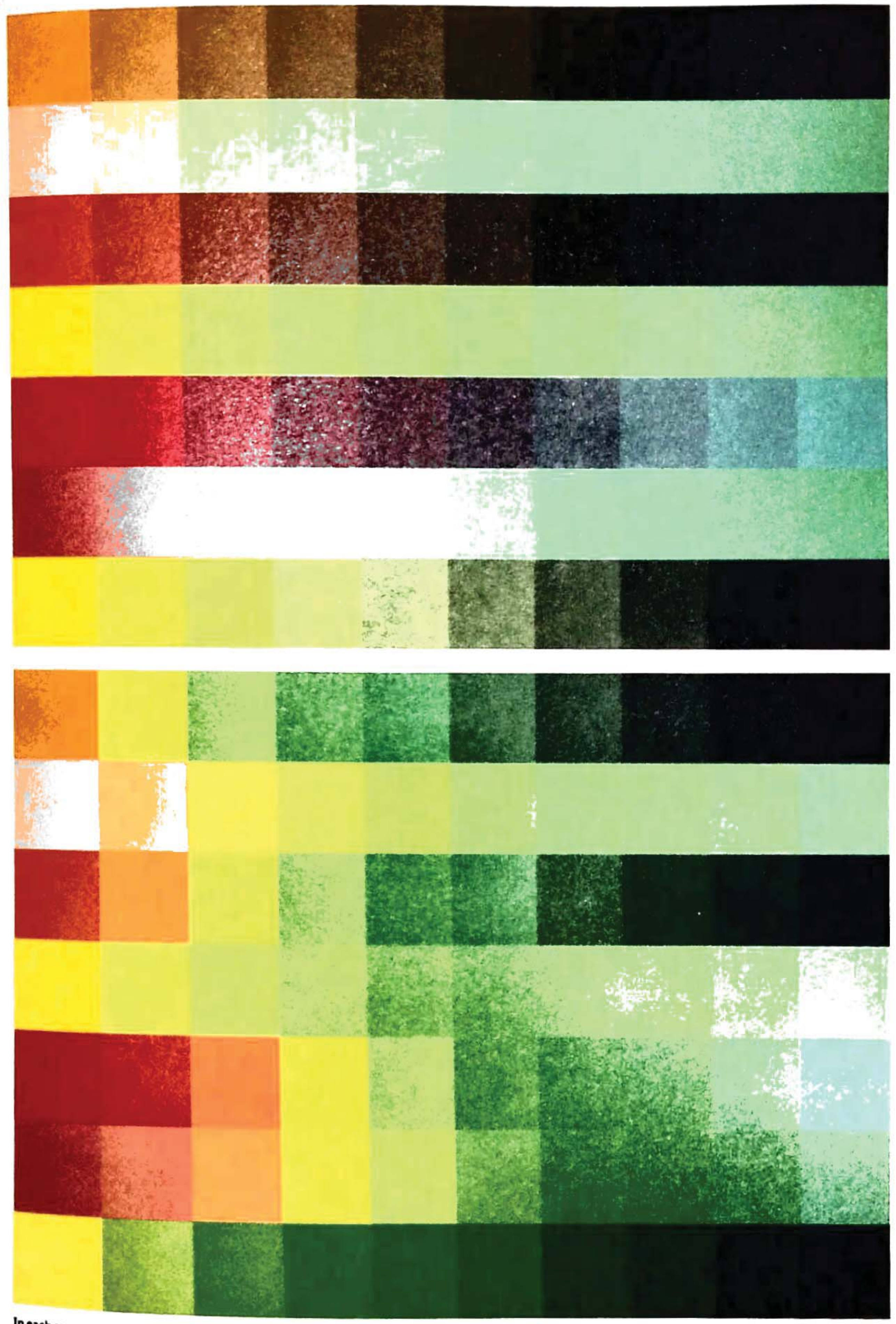
The number of color gradations tileCountX and the number of rows tileCountY are determined by the position of the mouse.

Drawing the grid row by row.

The colors for the left and right columns are set in the arrays colorsLeft and colorsRight.

The intermediary colors are calculated with lerpColor(). This function performs the interpolation between the individual color values. The variable amount, a value between 0 and 1, specifies the position between the start and end color.

In all programs that are described in the color palette chapters, a color palette in ASE format for Adobe applications can be saved using the C key.

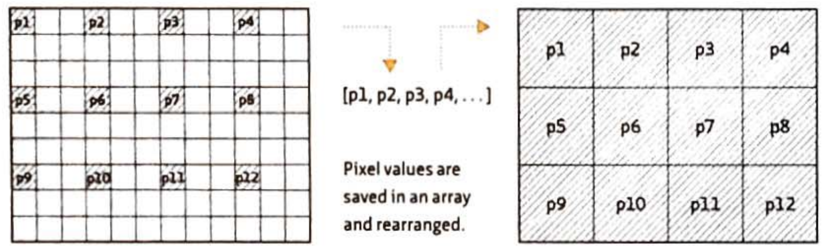

In each row, two colors are interpolated in ten steps. above in the RGB color model and below in the HSB color model
→ P.1.2.1_01.pde

P.1.2.2

Color palettes from images

We constantly are surrounded by color palettes; we need only to record and evaluate them. The colors obtained from the photograph of one individual's wardrobe—a very personal color palette—are selected and sorted using the following program. You can export the resulting color collection and use it as an inspirational color palette.

→ P_1_2_2_01.pde



An image is scanned in a specific grid spacing.

Palette with color fields.

The pixels of a loaded image are scanned using the mouse position in a specific grid spacing, one by one and row for row, in order to define the respective color value. These values are stored in an array and can be sorted by hue, saturation, brightness, or gray value.

Mouse: *Position x: Resolution*
Keys: *1–3: Change example image • 4–8: Change sort mode • S: Save PNG*
P: Save PDF • C: Save ASE palette

→ P_1_2_2_01.pde

```
PImage img;
color[] colors;

String sortMode = null;

void setup(){
  ...
  img = loadImage("pic1.jpg");
}

void draw(){
  ...
  int tileCount = width / max(mouseX, 5);
  float rectSize = width / float(tileCount);
  →
```

The variable img is declared as an image variable.

The currently selected sorting mode is always stored in the variable sortMode. The default is to not sort, and the value is therefore set at null (undefined).

The specified image is loaded and stored in img.

The number of rows and columns in the grid tileCount depends on the x-value of the mouse. The function max() selects the larger of the two given values.

The grid resolution just calculated is now used to define the size of the tiles, rectSize.

```
  int i = 0;
  colors = new color[tileCount*tileCount];
  for (int gridY=0; gridY<tileCount; gridY++) {
    for (int gridX=0; gridX<tileCount; gridX++) {
      int px = (int) (gridX*rectSize);
      int py = (int) (gridY*rectSize);
      colors[i] = img.get(px, py);
      i++;
    }
  }
  if (sortMode != null) colors = GenerativeDesign.sortColors(
                                   this, colors, sortMode);

  i = 0;
  for (int gridY=0; gridY<tileCount; gridY++) {
    for (int gridX=0; gridX<tileCount; gridX++) {
      fill(colors[i]);
      rect(gridX*rectSize, gridY*rectSize, rectSize, rectSize);
      i++;
    }
  }
  ...
}
```

The colors array is initialized. When the tileCount is 10, for example, the array is set with length 100.

Now the image is scanned, row by row, with the previously calculated grid spacing, rectSize. The color value of the pixel at the position px and py is set with img.get() and written in the color array.

When a sort mode has been selected (i.e., sortMode is not equal to null), the colors are sorted using the function sortColors(). The parameters passed to this function are a reference to the Processing program this, the array colors, and a value for the sorting mode sortMode.

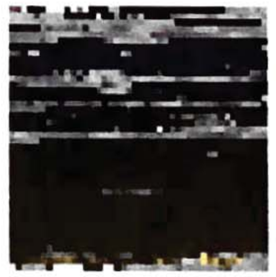
In order to draw the palette, the grid is processed again. The fill colors for the tiles are taken, value by value, from the array colors.

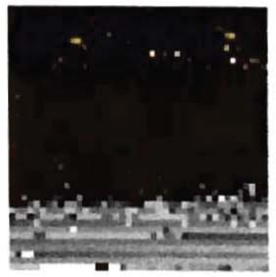
```
void keyReleased(){
  if (key == 'c' || key == 'C') GenerativeDesign.saveASE(this,
                                   colors, timestamp()+".ase");
  ...
  if (key == '4') sortMode = null;
  if (key == '5') sortMode = GenerativeDesign.HUE;
  if (key == '6') sortMode = GenerativeDesign.SATURATION;
  if (key == '7') sortMode = GenerativeDesign.BRIGHTNESS;
  if (key == '8') sortMode = GenerativeDesign.GRAYSCALE;
}
```

The function saveASE() allows an array of colors to be saved as an Adobe Swatch Exchange (ASE) file. The palette can then be loaded as a color swatch library, for instance, in Adobe Illustrator.

The keys 4 to 8 control what sorting function is applied to colors. For this the sortMode is set at null (no sorting) or at one of the constants, HUE, SATURATION, BRIGHTNESS or GRAYSCALE, from the Generative Design library.

Original image: *Subway Tunnel.*
→ Photograph: Stefan Eigner


Pixels arranged according to hue.
→ P_1_2_2_01.pde


Pixels arranged according to saturation.

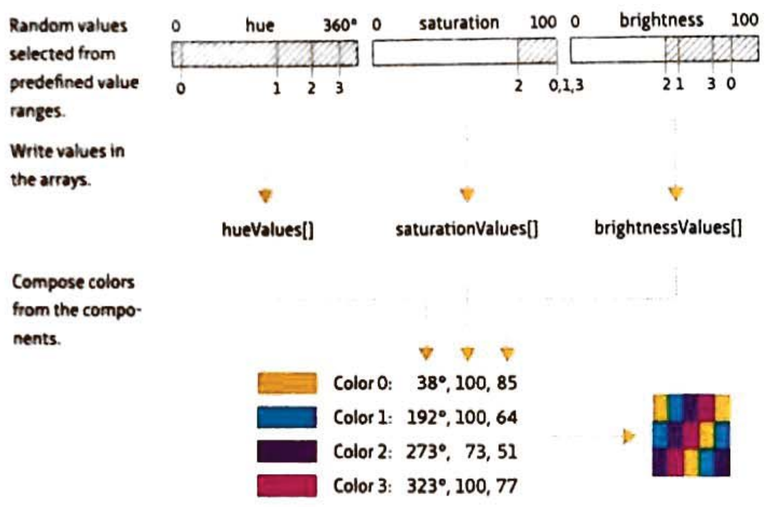
Pixels arranged according to brightness.

P.1.2.3

Color palettes from rules

All colors are made up of three components: hue, saturation, and brightness. The value for these color components are defined using a set of rules. By using controlled random functions, you can quickly create different palettes in specific color nuances.

→ P_1_2_3_01.pde



Values for hue, saturation and brightness are randomly selected from predefined ranges of values. This combination of rule sets—the definition of value ranges—and random functions means that new palettes are continually created and that they always produce specific color nuances.

Because the perception of color depends on context, the produced colors are drawn in an interactive grid. Even the color nuances emerge more distinctly.

Mouse *Position x/y: Grid resolution*
Keys: *0–9: Change color palette • S: Save PNG • P: Save PDF • C: Save ASE palette*

→ P_1_2_3_01.pde

```
int[] hueValues = new int[tileCountX];
int[] saturationValues = new int[tileCountX];
int[] brightnessValues = new int[tileCountX];
```

An individual array is used to save hue, saturation and brightness. Depending on what key is pressed (0–9), the arrays are filled according to different rules.

```
void draw() {
  ...
  int index = counter % currentTileCountX;

  fill(hueValues[index],
       saturationValues[index],
       brightnessValues[index]);
  rect(posX, posY, tileWidth, tileHeight);
  counter++;
  ...
}
```

When the grid is drawn, the colors are selected from the arrays, one by one. The continually incrementing variable counter starts to cycle through the same values because of the modulo operator, %. When currentTileCountX is 3, for instance, index will consecutively hold the values 0, 1, 2, 0, 1, 2 ... This means only the first colors in the arrays are used in the grid.

```
if (key == '1') {
  for (int i=0; i<tileCountX; i++) {
    hueValues[i] = (int) random(0,360);
    saturationValues[i] = (int) random(0,100);
    brightnessValues[i] = (int) random(0,100);
  }
}
```

When key 1 is pressed, the three arrays are filled with random values from the complete ranges of values. This means any color can appear in the palette.

```
if (key == '2') {
  for (int i=0; i<tileCountX; i++) {
    hueValues[i] = (int) random(0,360);
    saturationValues[i] = (int) random(0,100);
    brightnessValues[i] = 100;
  }
}
```

Here brightness is always set at the value 100. The result is a palette dominated by bright colors.

```
if (key == '3') {
  for (int i=0; i<tileCountX; i++) {
    hueValues[i] = (int) random(0,360);
    saturationValues[i] = 100;
    brightnessValues[i] = (int) random(0,100);
  }
}
```

When the saturation value is set at 100, no pastel tones are created.

```
if (key == '7') {
  for (int i=0; i<tileCountX; i++) {
    hueValues[i] = (int) random(0,180);
    saturationValues[i] = 100;
    brightnessValues[i] = (int) random(50,90);
  }
}
```

Here a restriction occurs in all color components. For instance, the hues are only selected from the first half of the color wheel, creating warmer colors.

```
if (key == '9') {
  for (int i=0; i<tileCountX; i++) {
    if (i%2 == 0) {
      hueValues[i] = (int) random(0,360);
      saturationValues[i] = 100;
      brightnessValues[i] = (int) random(0,100);
    }
    else {
      hueValues[i] = 195;
      saturationValues[i] = (int) random(0,100);
      brightnessValues[i] = 100;
    }
  }
}
```

It is also possible to mix two color palettes. The expression i%2 produces alternately the numbers 0 and 1. When the result is 0, a darker, more saturated color is saved in the array.

Otherwise the second rule is applied, and hue and brightness are set to fixed values. These values produce bright blue tones.