



Programming Design Systems

A free digital book that teaches a practical introduction to the new foundations of graphic design. By Rune Madsen.

[Start Reading](#)

[Subscribe to Newsletter](#)

Introduction

“We are now in transition from an object-oriented to a systems-oriented culture. Here change emanates, not from things, but from the way things are done.”

Jack Burnham (1968), Systems Esthetics

If I asked you to define the role of a graphic designer, what would it be? The answers to this question can vary widely, but my definition would be something like this: A graphic designer is someone who communicates a piece of content in shape and color. This work can take many forms, but up until recently, it mainly consisted of printed products in the form of posters, business cards, book covers, etc.

Today we find ourselves facing new challenges, not because this definition of design has changed much, but because the products we're required to build have changed. We now spend a majority of our time looking at screens instead of paper, and this has created a great need for designers who understand how to design for digital devices. But a digital product is not the same as a printed product. Digital products are displayed on screens of different sizes and with dynamic content. Digital products allow users to interact with their content, and take advantage of motion and animation. Furthermore, digital products often have temporal logic where a linear narrative is replaced by a set of complex states and transitions. All in all, digital products all share a common trait: They are created with programming languages.

For a field rooted in the fine arts, this has been a difficult transition. Many graphic design schools have resorted to teaching a waterfall philosophy where students are positioned to think of themselves as creatives who come up with ideas for others to build. After all, this is easier than adopting a whole new set of processes. However, the fundamental problem with this approach is that static design tools like Illustrator and Sketch fail at prototyping digital systems. Even in web design where the page metaphor is still prevalent, it seems limiting to define the design to just the styles of the page. Is google.com a good website because of the look of the search field? The traditions from the fine arts is a great positive, and this very book builds upon that foundation. However, there is a century-long bond between the field of design and new advances in

technology, and if graphic designers do not become fluent in this new digital reality, they will become irrelevant. We now have the ability to write code that produces beautiful designs, and the designer of tomorrow will have to understand how to deliver on that promise.

This book is the result of a simple question: What happens when we try to redefine the graphic design curriculum using a programming language as the tool for the designer? There are several reasons why this is a powerful concept. First of all, graphic designers have always used systems in their work. We use grid systems to balance our layouts and color circles to pick colors with proper distance to each other. History has shown us that systems can cure the fear of the blank canvas, and it is a powerful concept to encode these ideas into actual software. Second, code enables designers to do things they couldn't do before. Variations of a design can be tested much faster during the prototyping phase, and randomization can be used to reveal designs that the designer would never have created with a pencil. Third, it enables designers to create dynamic systems that can change their designs based on time, place, or use. Throwing a design over the wall for production is a bad legacy of the printed page, and there is no reason for the design process to end with the birth of a product.

This book is structured like an introductory text about graphic design, focusing on the elements of visual design and how they relate to algorithmic design. The book is written for designers wanting to become better programmers and vice versa. As you go through the text, you will notice that it starts with the very basics. The code will be simple and the exercises will be very constrained. If this feels simplistic, keep reading. We will soon enough touch upon more challenging themes, but these basic concepts lay the foundation for some of the more complex ideas. At the end of the book, it is my hope that you have learned two new skills: How to use code to create new and interesting graphic designs, and how to evaluate whether these designs can be considered successful.

I have decided to use the [P5.js JavaScript library](#) for all examples in the book. It has proven to be a great programming environment for beginners while also being powerful enough for advanced users. I am aware that it is impossible to choose a single programming language that will work for everyone, and readers will probably need to port the ideas from this book to other languages and frameworks depending on the nature of specific projects. I have therefore tried to write the text to be as general as possible. It is also important to note that this is not a book about web design. Although the examples can be embedded on any web page, the techniques can be applied to both digital and print work. In fact, I have seen my students create a diverse range of designs over the years, including projects in sculpture, painting, fashion, photography, game design, web design, and printed matter.

The book is written for readers with an introductory knowledge of programming in JavaScript, which means that little time will be devoted to explaining concepts like variables, functions, and loops. For this audience, I highly recommend Daniel Shiffman's [Coding Rainbow YouTube channel](#), where Dan does a much better job explaining these concepts than I ever could. These videos will also introduce the basic concepts of programming in P5.

I am not the first author to write about systems in graphic design, and this book uses ideas from many incredible authors who are too numerous to mention here. I am not the first to write about code and graphic design either, but I've been frustrated by books on the subject that seem to fall in two categories: Those with a focus on code and generative design that do not teach graphic design principles at all, and those celebrating computational design as a spectacle without identifying how projects are made or why they are successful. There is a need for educational material that teaches the fundamentals of graphic design in a modern way, especially if it also gives students mental models for critiquing digital

design projects.

I have chosen to publish this book for free online without a publisher. The main reason for this is freedom. It allows me to write and design the type of book that I would read. This is reflected in the structure of the book, which consists of many shorter chapters that are easier to read on the web and straightforward for teachers to remix for use in classrooms (which is completely legal because of the Creative Commons license). I will be writing these chapters in a nonlinear fashion, working on topics that I find the most interesting at a given time. There is no concrete deadline for the finished book, although I am pursuing it as a full-time endeavor. The source code for the book is available on [GitHub](#), and I encourage all readers to submit issues or pull requests with edits. To stay up to date with the progress of the book, you can subscribe to the newsletter.

I would like to thank a group of people without whom this book would never have existed. When I moved to New York in 2009 to study at The Interactive Telecommunications Program at New York University, I only had a vague notion of wanting to work in the intersection between art and code. As a Flash developer who primarily built banners for advertising, the two years at ITP completely changed my thinking. I would like to thank Dan O'Sullivan for his enormous support for this project, which started as a class at ITP, and is currently growing into a book while I'm a research resident there. This book would not exist without the help and mentorship of Daniel Shiffman, whose work has inspired thousands of students to learn code. I want to thank Stewart Smith, who taught the Visualizing Data class where the early ideas for this book began. My thanks go to all of the ITP faculty and alumni who helped shape these ideas, including Patrick Hebron, Greg Borenstein, Clay Shirky, Danny Rozin, Tom Igoe, David Nolen, Gabe Barcia-Colombo, and George Agudow. A special thanks go to Lauren McCarthy, the original author of P5.js, and to Casey Reas and Ben Fry for their work on Processing. I want to thank Chandler

Abraham for his detailed edits in the chapter on color spaces, as well as Claire Kearney-Volpe for answering my numerous questions about accessibility. Now, let's begin.

What is a design system?

No matter where you go, you are surrounded by systems. We use the word 'system' to describe a group of interacting parts as a common whole, and these can be simple, like a watch, or incredibly complex, like the web of computer networks we call the internet. In this book, the term 'design system' is used to describe a philosophy that encourages designers to define the rules of their designs as a system of instructions that can be used on more than a single product. This is best explained with a simple story.

Let us suppose I am asked to design a label for a local brewery's famous stout beer. I want to make sure that my design is a good fit for the product, so I spend a lot of time researching and tasting the beer before making my final design. The brewery loves my new design, and they ask me to design labels for the rest of their 10 beer types. Now I am faced with a problem. The colors, typography, and illustrations I chose for my label design was a great fit for a strong and dense beer, but it will not work for the other beers. So I end up creating labels with a different visual style for each beer, but without a consistent branding for the brewery.

This problem could have been avoided had I designed the first label with the conviction that it would be a part of something bigger: A design system for the brewery. Instead of focusing on the particular stout, I would have defined a range of visual styles to be used for different beer types. This would include a set of related typefaces and colors with enough variance to be used for the individual beers, but recognizable enough to

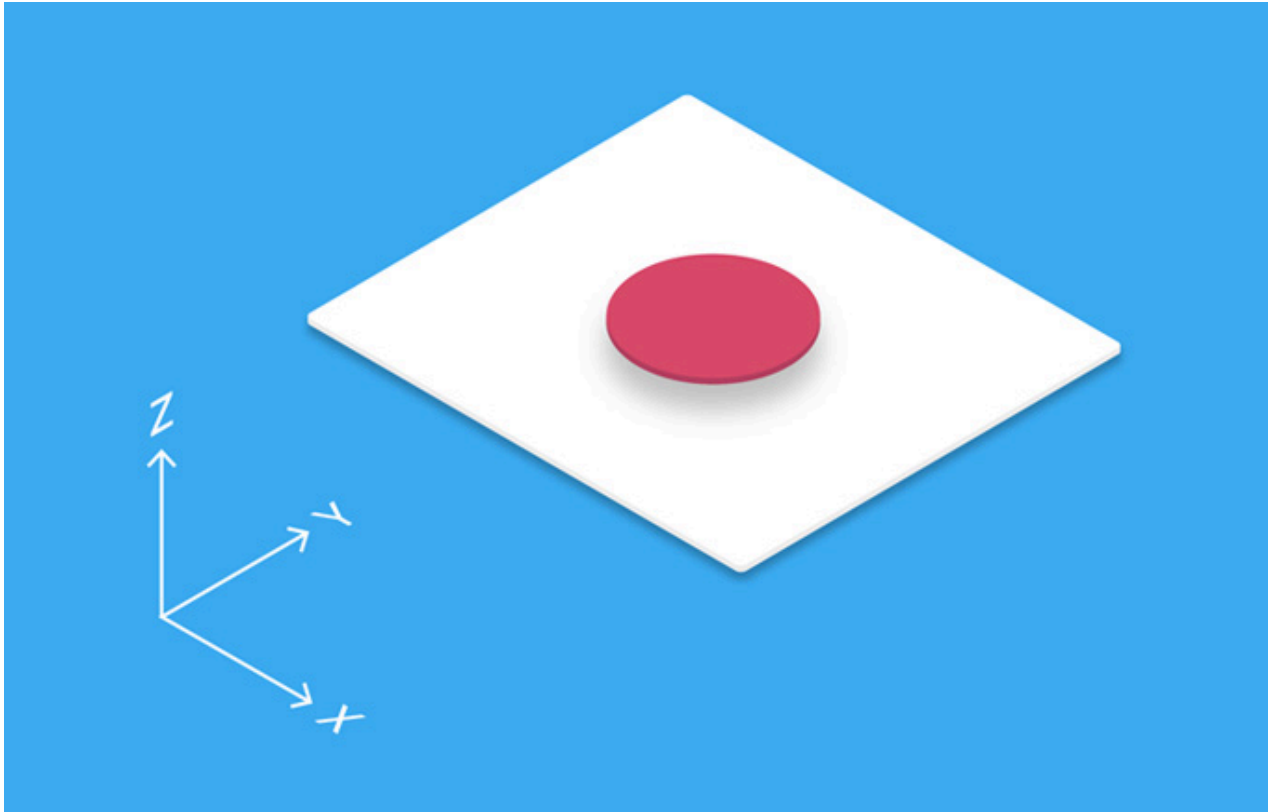
provide a common identity for the brewery. Only after creating this design system would I design the first label.



The Graphics Standards Manual created for the New York Transit Authority in 1970 is one example of a design system. With more than 350 pages, it defines visual guidelines for all signage related to the New York City subway. ©

Design systems offer a different way of thinking, where the designer is forced to consider many scenarios and constraints instead of relying on a one-off design process. This approach is of course nothing new, and some might say that it has always been a part of the designer's job. Thousands of design systems have been created over the years, from the New York

Transit Authority Graphics Standards Manual to Google's Material Design. Most Fortune 500 companies are recognizable for a reason: they have defined strict rules for use of typography and color across their product line.



Google's **Material Design** document was created to help explain the ideas behind the company's visual language. This includes rules for use of color, layering, and animation.

A recent example of a simple design system is the MIT Media Lab logo created by Pentagram. The idea is simple: Fill a 7x7 grid with perpendicular lines to create abstract letters or symbols in black and white. This system is used to create logos with acronyms for the 23 research groups at the lab, and as a basis for building a custom typeface, icons, and patterns.

mit
media
lab

affective
computing

biomechatronics

camera
culture

changing
places

civic
media

design
fiction

fluid
interfaces

human
dynamics

lifelong
kindergarten

macro
connections

mediated
matter

molecular
machines

object-based
media

opera of
the future

personal
robots

playful
systems

responsive
environments

social
computing

social
machines

speech+
mobility

synthetic
neurobiology

tangible
media

viral
communications



The MIT Media Lab design system. ©



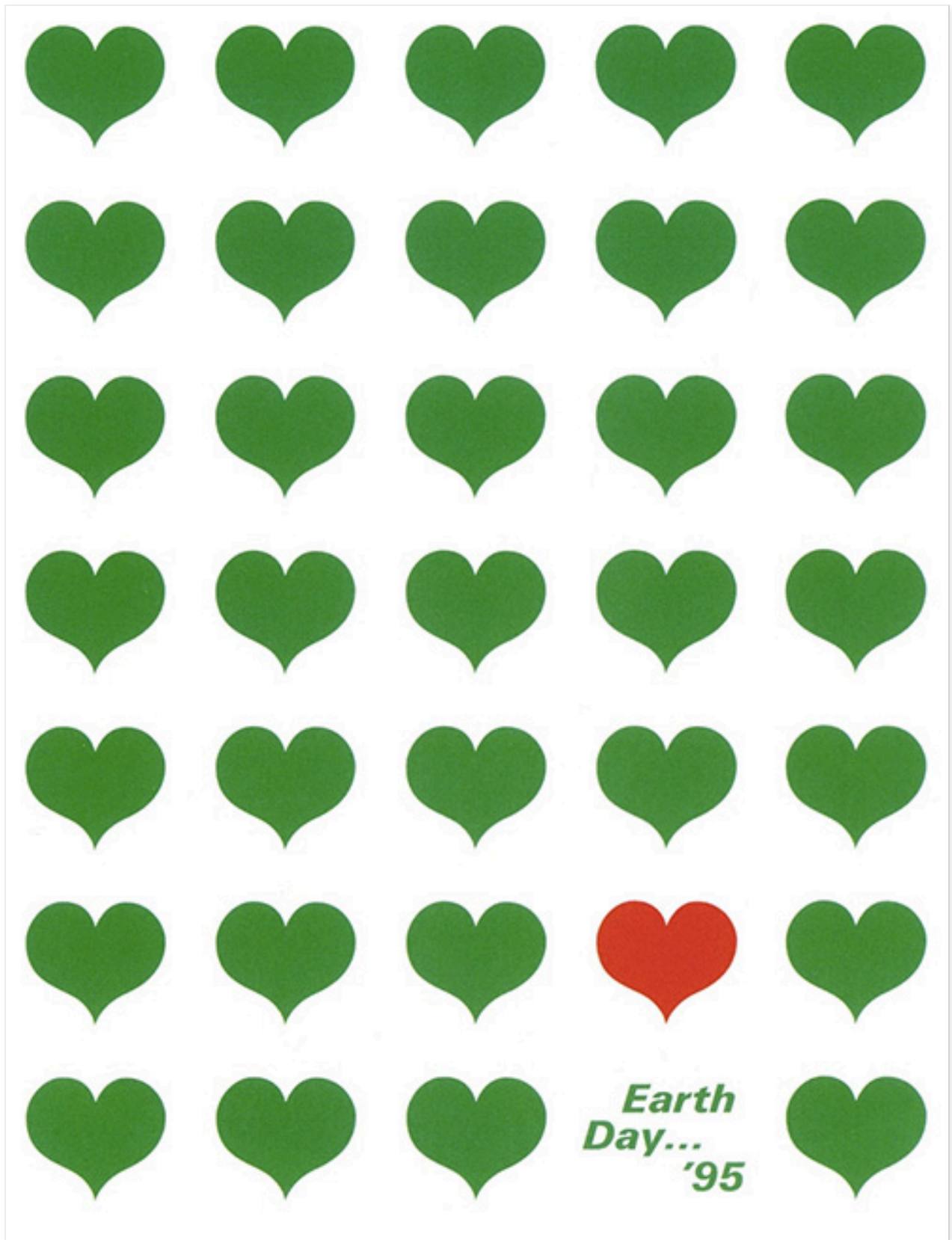
This logo is an excellent example of how a design system can be used to give both a consistent look and a distinct style to an organization. The system is both simple and flexible, and it has room for an almost infinite number of designs.

Design systems are especially interesting today, because digital products *are* systems, and designers who code are no longer confined to the creation of design systems that end up in printed manuals. Code allows designers to not just create designs, but build digital systems that create designs. Granted, more time will be spent on formulating the rules of the system in code, but designers will be free from the limitations imposed by traditional design software. The American computer scientist Donald Knuth writes about this very thing:

“Meta-design is much more difficult than design; it is easier to draw something than to explain how to draw it. One of the problems is that different sets of potential specifications cannot easily be envisioned all at once. Another is that a computer has to be told absolutely everything. However, once we have successfully explained how to draw something in a sufficiently general manner, the same explanation will work for related shapes, in different circumstances; so the time spent in formulating a precise explanation turns out to be worth it..”

Donald Knuth (1986), The Metafont Book

The words “*sufficiently general manner*” are important here. Software can be written to allow a range of possible outputs, and variations of a design system can be generated in (literally) fractions of a second. The ability to procedurally generate designs is one of the most empowering aspects of algorithmic design, whether it leads to generating 45,000 variations of a logo, building an infinite galaxy of planets with generative landscapes, or creating a dynamic article that changes its content based on map selections.



A poster designed by Paul Rand. ©

The same poster recreated in code.

In this book, we will investigate what designers can build when they are encouraged to create design systems and make these systems come to life by writing software.

Shape

Figure and ground

Graphic design is the art of using form and color to successfully convey a message, but this is not as easy as it sounds. The design process is one of trial and error, and it often takes many iterations to create a design that clearly communicates what you're trying to say. Although we tend to think of this process as something involving gut instinct, experienced designers have an ever-expanding checklist of visual relationships that helps guide them through to the final result.

We begin our journey into the field of graphic design by looking at the first item on this checklist: The relationship between a figure and its ground. In the following, we will use a single rectangle to demonstrate how three simple variables – position, size, and rotation – can be manipulated in code to create a variety of different expressions. Although this can seem rather basic, these relationships are crucial ingredients in most successful designs, and they also happen to be a great way to introduce the concepts of visual communication.

Position

Most written languages have a natural reading direction. In a majority of the Western world, characters are arranged in horizontal lines, and we read from left to right, top to bottom. This tells us that the position of a shape can be used to guide the eyes of a user through a design. Graphic novels use the positioning of captions to guide the reader through the

storyline, and most newspapers float quotations to visually separate them from the body text.

First, because of reading direction.

Last, because of reading direction.

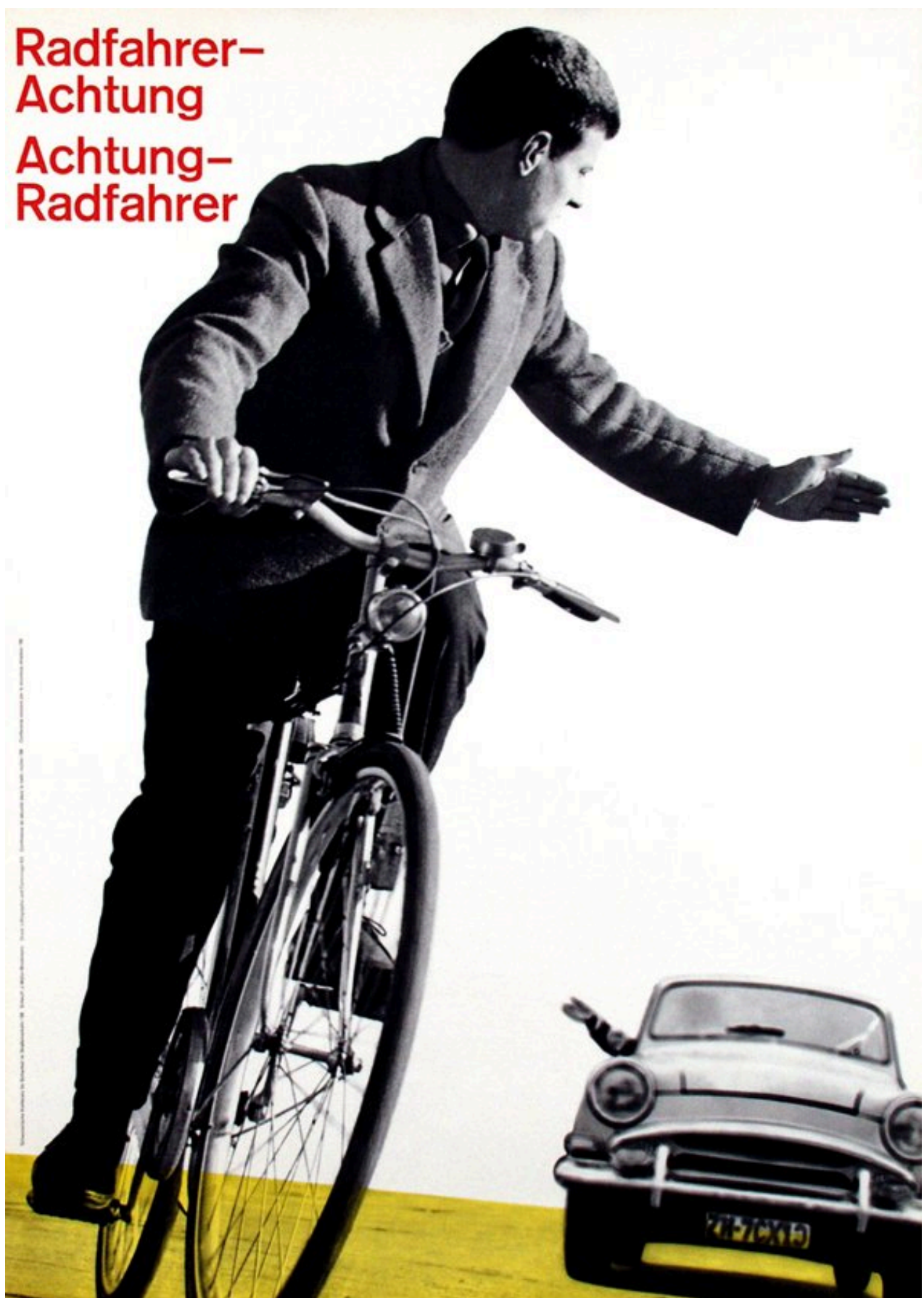
In these posters for Swiss Auto Club, Joseph Müller-Brockmann uses position (and size) to prioritize certain shapes. In the first poster, the vehicle in motion is prioritized over the running child. This heightens tension and makes it apparent that the vehicle is on a collision course with the child. In the second poster, the hand-signaling cyclist is the first shape on the page, which achieves exactly the opposite effect.

Automobil-Club der Schweiz

schützt das Kind !



**Radfahrer-
Achtung
Achtung-
Radfahrer**



We use the `rect()` function to draw a rectangle with P5, and the first two numbers passed to this function determine the position of the shape within the canvas. If these numbers are both zero, the shape will appear in the top left corner, and higher values will draw it further to the right (`x`) and bottom (`y`) of the canvas. This makes it easy to position shapes in code, and many graphic programming languages work this way.

```
rect(0, 0, 100, 100);  
rect(300, 200, 100, 100);
```

Instead of randomly typing numbers until a shape is in the right position, it can be beneficial to use simple math to calculate the position of the shape based on the size of the canvas. This will define the relationship between the canvas and shape explicitly in your code, and make it easier to scale the design. As an example, take the following image.

This design could be achieved by using numbers slightly smaller than the canvas for the `x` and `y` position of the rectangle. However, was I to change the width of the canvas, the rectangle would no longer show up in the right place. I would need to change the `x` position of the rectangle to reflect the new canvas size.

This is fixed in the code below, where the `width` and `height` variables are used to dynamically calculate the position of the rectangle. To calculate the `x` position, we start at the right edge of the canvas (`width`), then subtract the size of the rectangle (`100`), and finally subtract the gap we want between the edge of the canvas and the rectangle (`25`). The same formula is used for the `y` position.

```
rect(width - 100 - 25, height - 100 - 25, 100, 100);
```

Size

By increasing the size of important shapes and decreasing the size of less significant shapes, you can bring clarity to a design. This is a pattern we encounter daily in everything from traffic signs to headlines in an article.

This square is dominant because it takes up most of the canvas.

This square is less dominant because of the large empty space around it.

The Think Small advertising campaign from 1959 is a great example of how size can be used to emphasize supporting text copy. In this series of magazine ads, an image of the Volkswagen Beetle is printed in different sizes within an otherwise empty canvas.



© 1987 VOLKSWAGEN OF AMERICA, INC.

Think small.

Our little car isn't so much of a novelty any more.

A couple of dozen college kids don't try to squeeze inside it.

The guy at the gas station doesn't ask where the gas goes.

Nobody even stares at our shape.

In fact, some people who drive our little

flivver don't even think 32 miles to the gallon is going any great guns.

Or using five pints of oil instead of five quarts.

Or never needing anti-freeze.

Or racking up 40,000 miles on a set of tires.

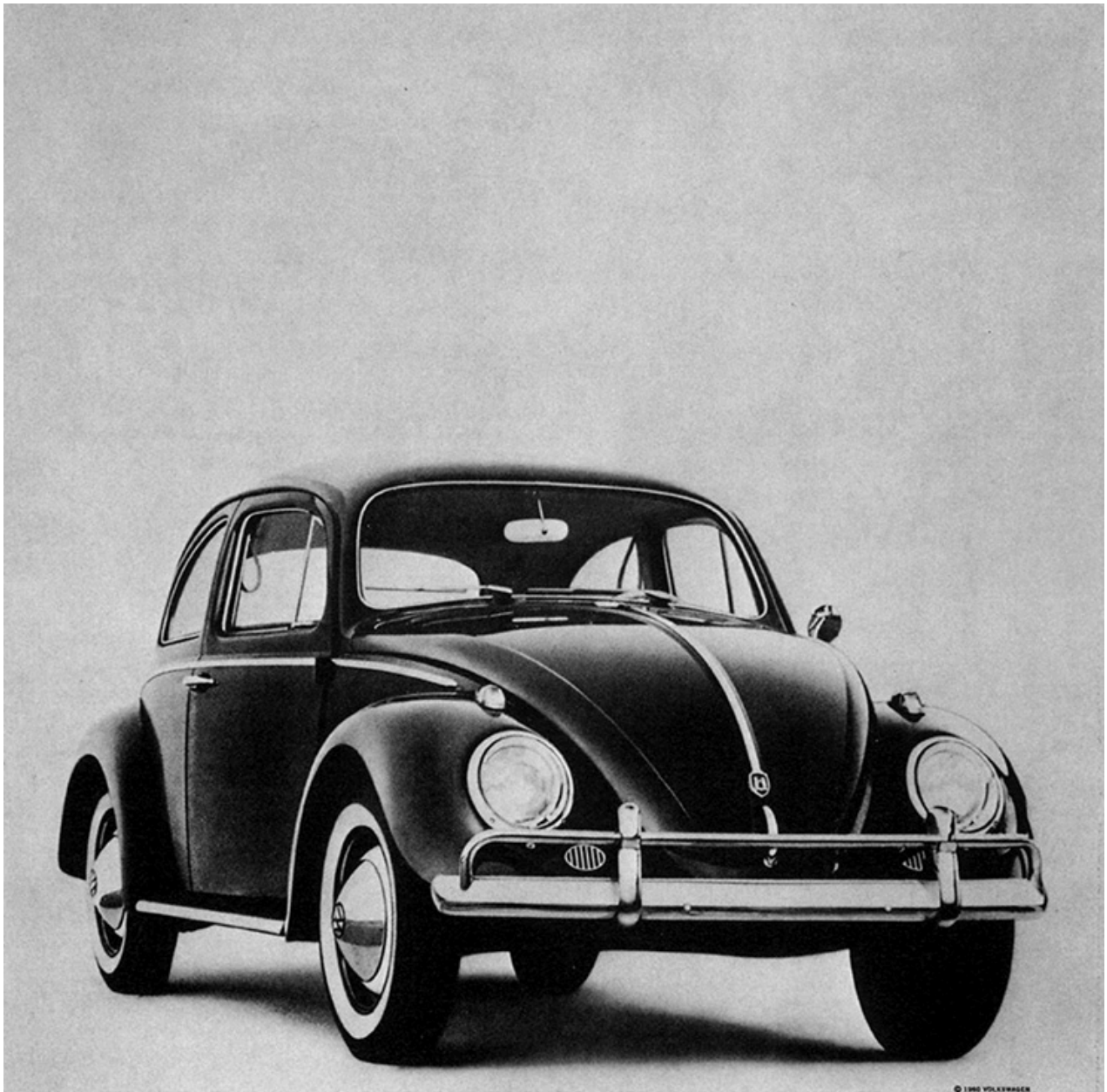
That's because once you get used to

some of our economies, you don't even think about them any more.

Except when you squeeze into a small parking spot. Or renew your small insurance. Or pay a small repair bill. Or trade in your old VW for a new one.



Think it over.



Lemon.

This Volkswagen missed the boat.

The chrome strip on the glove compartment is blemished and must be replaced. Chances are you wouldn't have noticed it; Inspector Kurt Kroner did.

There are 3,389 men at our Wolfsburg factory with only one job: to inspect Volkswagens at each stage of production. (3000 Volkswagens are produced daily; there are more inspectors

than cars.)

Every shock absorber is tested (spot checking won't do), every windshield is scanned. VWs have been rejected for surface scratches barely visible to the eye.

Final inspection is really something! VW inspectors run each car off the line onto the Funktionsprüfstand (car test stand), tote up 189 check points, gun ahead to the automatic

brake stand, and say "no" to one VW out of fifty.

This preoccupation with detail means the VW lasts longer and requires less maintenance, by and large, than other cars. (It also means a used VW depreciates less than any other car.)



We pluck the lemons; you get the plums.

The third and fourth number passed to the `rect()` function define the width and height of the shape. Like the position, it can be beneficial to

calculate the size of the rectangle dynamically based on the size of the canvas. Building on the example from above, the following code shows a truly dynamic design, where the static numbers used for size and spacing have been replaced by calculations. Now, both the position and size of the rectangle scales with the canvas automatically.

```
rect(width - (height/3) - (width/20), height - (height/3)
- (width/20), height/3, height/3);
```

However, there is one big problem with the code above: It has become very hard to read. If you need to change the position of the rectangle, it will take you a while to mentally parse what's going on. To solve this problem, it is often helpful to use variables at the top of your code to store these numbers. This makes the code more readable, which makes it easier for you (and other programmers) to change it later on. The following code produces the same design, but all numbers are now saved into variables with names that clearly communicate their purpose.

```
const size = height / 3;
const margin = width / 20;
const x = width - margin - size;
const y = height - margin - size;
rect(x, y, size, size);
```

Keeping your code organized is very important, especially as you start to make designs with more than a single shape.

Rotation

When you rotate a shape, the whitespace around it changes, and this can be used to make a shape appear active or in motion.

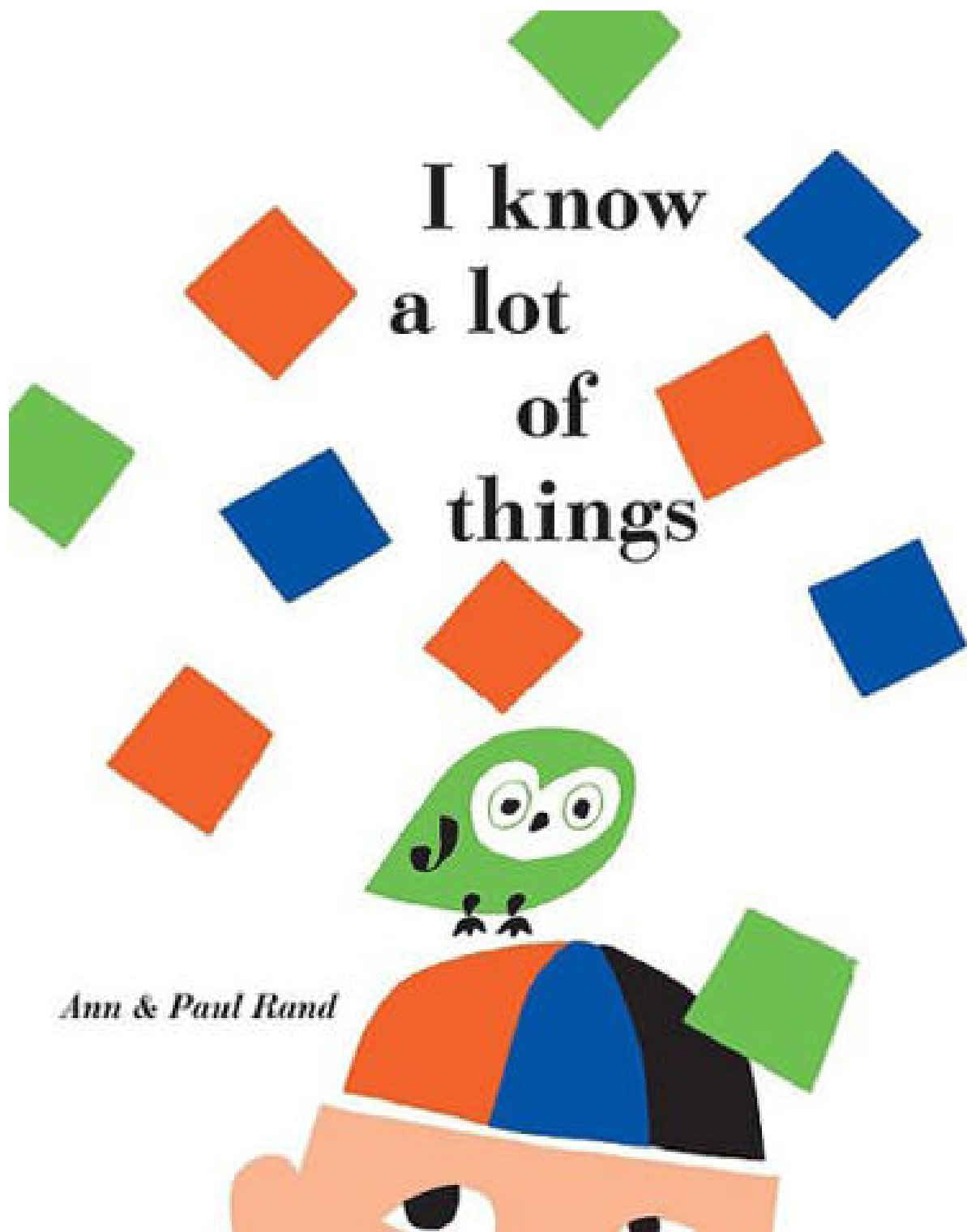
A rectangle with no rotation appears static because of the symmetric whitespace.

A rotation of 45° creates more complex – but still symmetric – whitespace.

A rotation of 27° breaks the symmetry, and the rectangle appears in motion.

Rotated shapes can be found in many design products: Books written for children often use rotated shapes to create fun and playful designs.

Popular magazines use rotated grid systems to make the content look less dry, while publications for a narrow audience might enforce horizontal and vertical lines to imply quality of the content within.



Paul Rand uses rotated rectangles on a playful cover for a children's book. ©



The podium in the Adidas logo is rotated to convey speed and dynamism. ©

The `rotate()` function takes a single number, which is expected to be the desired rotation in radians. A radian is a mathematical unit of angular measure based on the radius of a circle. If you follow the outline of a circle

for the length of its radius, that angle is 1 radian. By the magic of numbers, a full circle is therefore 2π (or about 6.283) radians. For those not interested in the intricacies of PI, the `radians()` function can be used to convert degrees to radians. All of the following lines of code will result in a rotation of (about) 90 degrees.

```
rotate(PI/2);  
rotate(1.57);  
rotate(radians(90));
```

When you use the `rotate()` function, you are not rotating individual shapes, but the entire canvas. Because all shapes are drawn on the canvas, they just happen to rotate too. Combined with `translate()`, which moves the starting point of the canvas, it's possible to achieve exactly the kind of rotation you want.

As an example, let us suppose we want to add rotation to our rectangle example from above. Here, we have added a single `rotate()` function before drawing our rectangle. Notice how the canvas (represented by the darker box) is rotating around its own beginning, not the rectangle.

```
const size = height / 3;  
const margin = width / 20;  
const x = width - margin - size;  
const y = height - margin - size;  
rotate(radians(10));  
rect(x, y, size, size);
```

If we want the rotation to happen around the top left corner of our rectangle, we have to do something that might feel a bit unintuitive. First, we have to move the canvas to the position of our rectangle by using the `translate()` function. After calling the `rotate()` function, we then draw our

rectangle at the top left corner of the translated canvas. As you can see, the rotation now happens around the rectangle.

```
const size = height / 3;
const margin = width / 20;
const x = width - margin - size;
const y = height - margin - size;
translate(x, y);
rotate(radians(10));
rect(0, 0, size, size);
```

As we dive into more layout techniques, we will further investigate how to use `translate()` and `rotate()` to make more sophisticated designs.

Designing a word

A good way to practice these relationships is the ‘design a word’ exercise. Pick an adjective from the dictionary and make a design for it by changing the position, size, and rotation of a single rectangle. This helps you build one of the most fundamental skills in graphic design: The ability to create visual relationships that make sense for your content.

The following example demonstrates the exercise for the word *steep*. How would you make a design for this word using just a rectangle and the three variables? Look for connections between the word and your variables, and you might notice that steepness and rotation are related: You can make the rectangle appear like a steep hill by rotating it. By changing the size, you can make that hill longer, continuing outside the canvas. By changing the position of the rectangle, you can create asymmetric whitespace, making sure that the user notices the rectangle. The following sketches demonstrate these steps in code.

1. First we rotate the rectangle until it looks steep.

2. Then we change the size to make it go beyond the canvas.
3. Then we draw attention to the rectangle by moving it away from the center.

The following examples show a few more designs to be used for inspiration. Keep in mind that not all adjectives will be this straightforward, so you might find yourself longing for more shapes. That is the subject of the next chapter, so do not worry.

Flat [See Code](#)

Big [See Code](#)

Shy [See Code](#)

This chapter demonstrates an approach that will be a continuing theme throughout this book: We will take topics from graphic design theory, understand how these ideas can be implemented in code, and practice these concepts through exercises. Although this was just the first step into the world of graphic design, the relationship between a figure and its ground is important. It is common to find problems in existing designs that can be traced back to this relationship, whether it's a user interface that fails to emphasize an important button, or a graph with ambiguity around its underlying dataset. Remember not to rush to make things pretty, and make these concepts a crucial part of your design process.

EXERCISE

Pick a random adjective from the dictionary, and write a sketch that tries to convey that word by changing the position, size, and rotation of a single rectangle. Do this for a couple of words, and ask friends to guess the word-image combinations.

Basic shapes

As long as we have had the ability to draw, humans have used shapes for visual communication. Although shapes are not words, and therefore have no objective semantic meaning, we have a natural understanding of how to translate the characteristics of shapes into meaning: Cave paintings created more than 30,000 years ago can be appreciated today without a need for translation.

In this chapter, we will look at the three basic shapes: the rectangle, the ellipse, and the triangle. First, we will analyze the characteristics of each shape, and then demonstrate how to use these basic shapes in the design process.

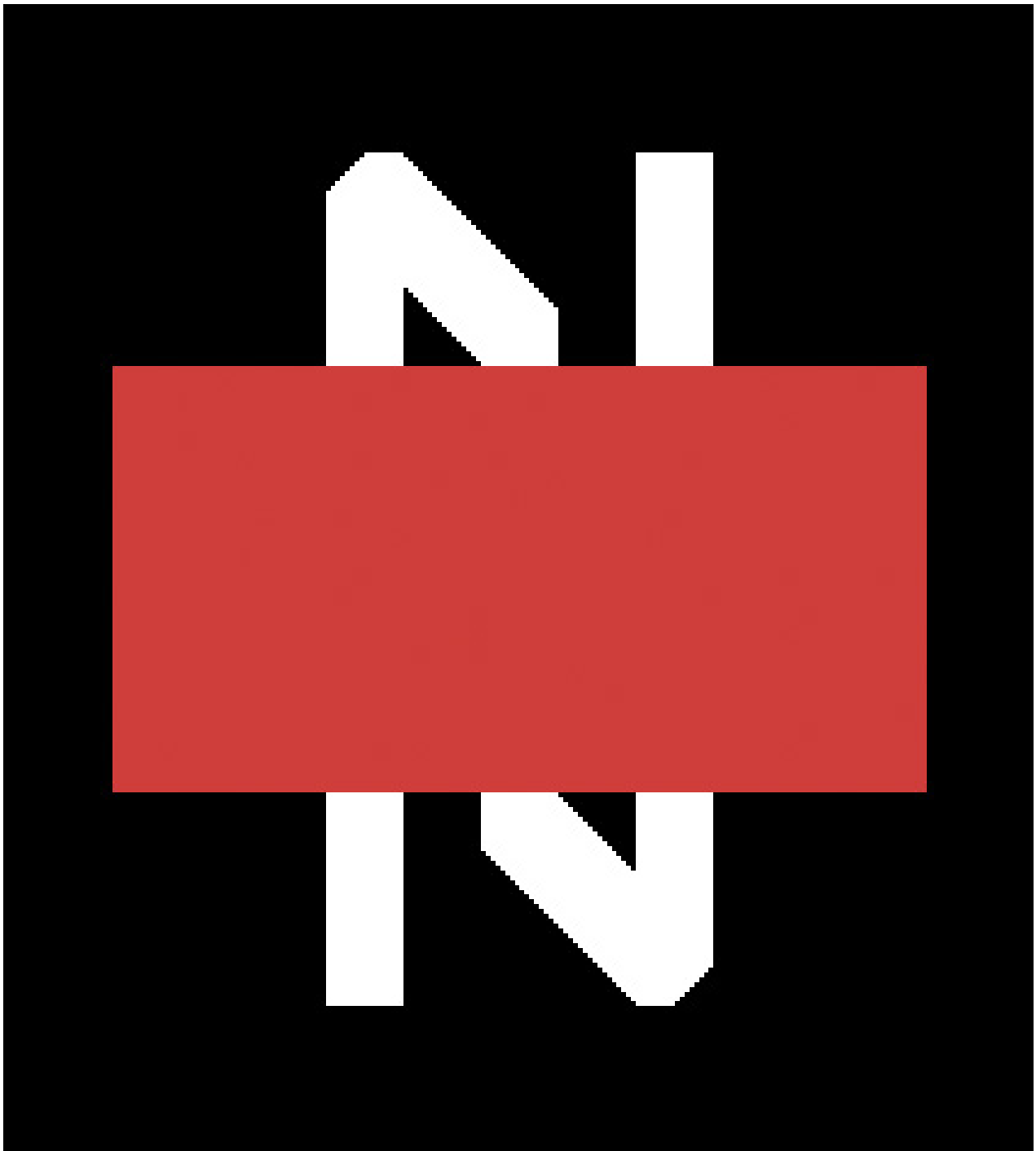
Rectangle

The rectangle is a symmetric, solid shape with parallel lines. As it does not exist much in nature, it has become the symbol for civilization itself. We build cities in rectangular grids, houses with bricks, and our interiors are rectangles too: doors, shelves, and windows.

The rectangle has been used throughout the history of the arts to set up constraints for the artist. We use rectangular canvasses, and grid systems to further divide this canvas into smaller modules. In this digital age, we use rectangular screens and operate with a square as the smallest visual denomination: the pixel.



The squares in the Microsoft logo become a symbol for a window, a flag, and a pixelated screen. ©



The logo for the 8-bit musician Nullsleep has a rectangle on top of a pixelated font as a reference to both anti-authoritarianism and early computer art. ©

In geometry, a rectangle consists of four points connected to form a closed shape with internal angles of 90 degrees. However, the `rect()` function in P5 allows us to draw a rectangle by stating the position of the top left corner (which we will call the origin point ●) as well as the size of the

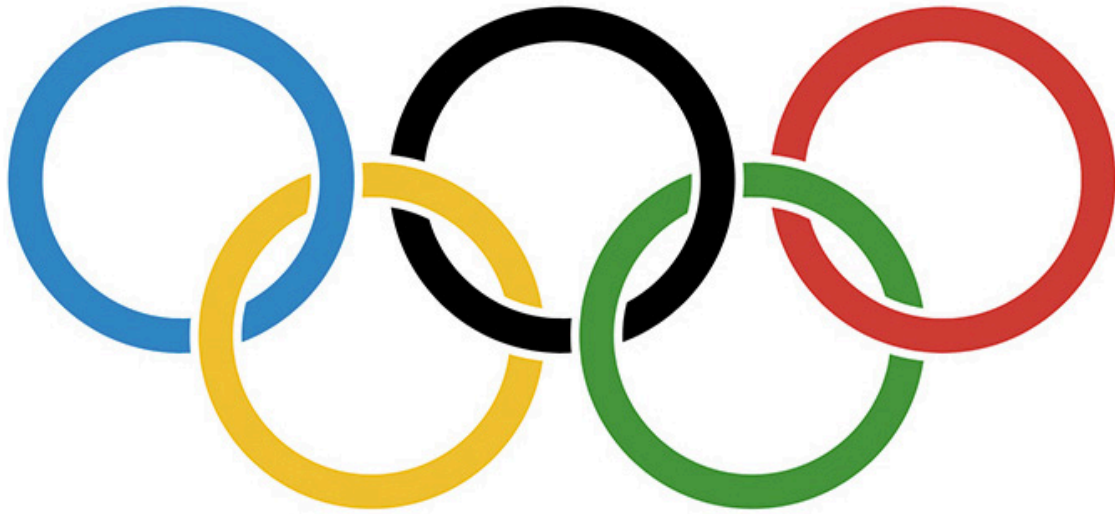
rectangle. As demonstrated below, the `rectMode()` function can be used to change the origin point of the rectangle to the center of the shape. This can be helpful in certain situations, e.g. if you want to draw a rectangle in the center of the canvas without needing to subtract half the size of the rectangle from its position.

```
const size = width * 0.3;  
rect(width/2, height/2, size, size);
```

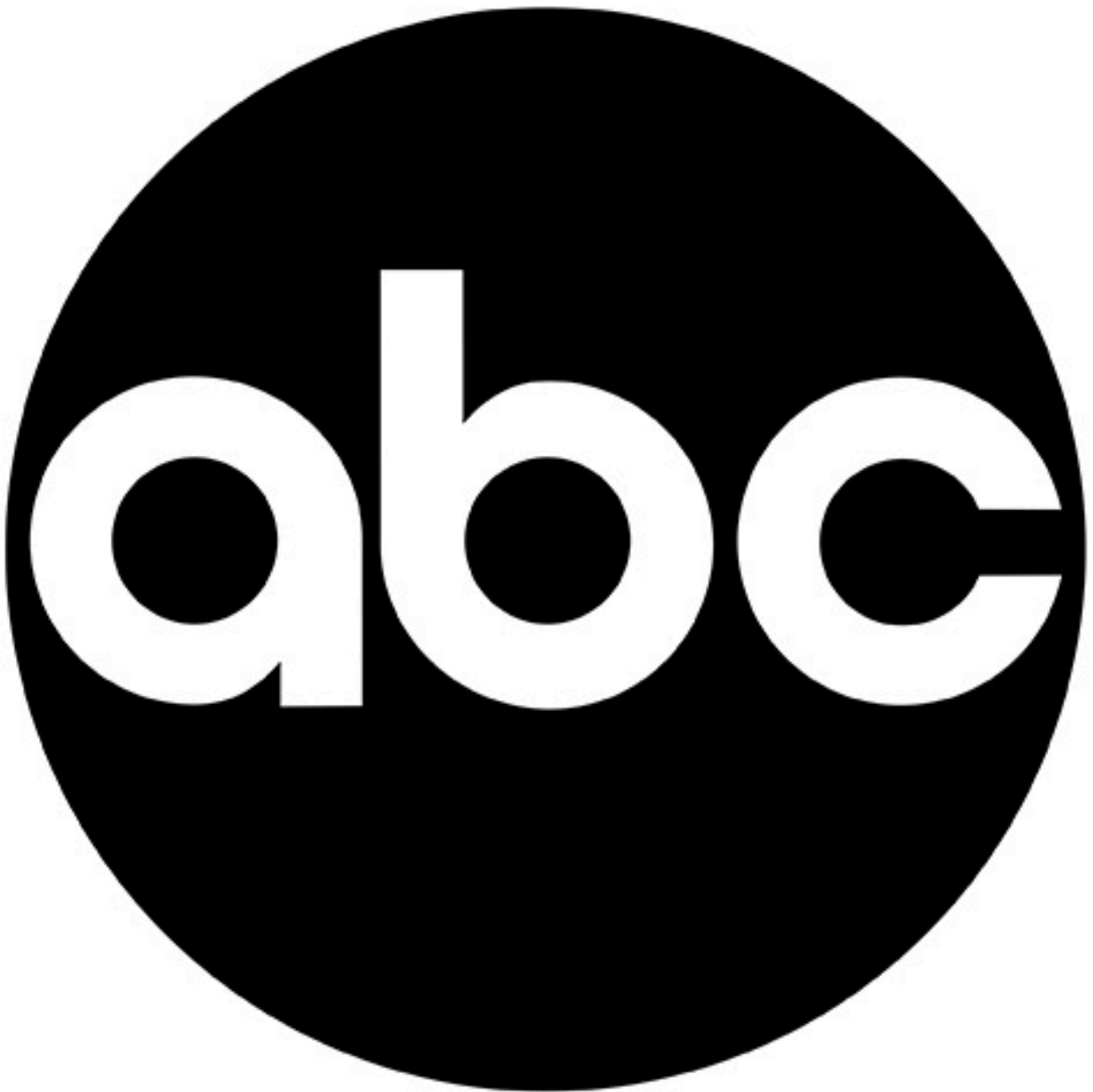
```
rectMode(CENTER);  
const size = width * 0.3;  
rect(width/2, height/2, size, size);
```

Ellipse

The ellipse is a smooth shape found many places in nature, in the shape of planets, raindrops, and the eyes of most animals. With no apparent sense of direction, there is something neutral about the ellipse, and humans tend to gather in ellipses to achieve unity: We dance in circles, and design the seating of most parliaments in elliptical arrangements.



The symbol for the Olympic Games by Pierre de Coubertin consists of five connected ellipses with colors taken from the flags of the participating countries from the 1912 games. ©



The ABC logo by Paul Rand features a simple ellipse with an elliptical typeface. ©

In geometry, an ellipse is a closed shape that you can draw by hammering two nails in the ground, connecting them by a string, and stretching a pen through the string to draw a circular shape. Although the outline of an ellipse looks smooth to the human eye, computers actually draw ellipses as a series of short, straight, connected lines. Unlike `rect()`, the `ellipse()` function will draw an ellipse with an origin point in the center of the shape. As demonstrated below, the `ellipseMode()` function can be

used to change the origin point to the top left corner. Given the nature of the ellipse, this means that the origin point is located outside the outline of the shape.

```
const size = width * 0.3;  
ellipse(width/2, height/2, size, size);
```

```
ellipseMode(CORNER);  
const size = width * 0.3;  
ellipse(width/2, height/2, size, size);
```

Triangle

The triangle is an asymmetric shape, unique among the basic shapes for its directionality. It is commonly known as the symbol for both masculinity (▲) and femininity (▼), and it is widely used in graphic design for its aesthetic qualities. One of our most significant cultural artifacts, the Great Pyramids of Giza, are also famous depictions of the triangle, pointing towards the assumed rotational center of the sky to which the Egyptians ascribed godly qualities.



The Delta logo is a triangle consisting of four smaller triangles. The logo refers to the greek triangle letter by the same name, and the directionality of the triangle is used to imply speed and flight. ©

In geometry, a triangle is a closed shape consisting of three points. The sum of the internal angles of the triangle will always be 180 degrees (or π radians). The triangle has played a central role in many mathematical breakthroughs, including Euclidian geometry, trigonometry, as well as 3D computer graphics. Unlike the `rect()` and `ellipse()` functions that both expect a single position for the shape's origin point, the `triangle()` function needs the coordinates of all three corners of the triangle. This also means that there is no such thing as a `triangleMode()` function. You will need to perform your own calculations to draw a triangle around a specific origin point, which is demonstrated in the examples below.

```
const size = width * 0.15;  
translate(width/2, height/2);
```



```
triangle(0, 0, size, size*2, -size, size*2);
```

```
const size = width * 0.15;  
translate(width/2, height/2);  
triangle(0, -size, size, size, -size, size);
```

An ice cream cone

I give my students the following (somewhat silly) exercise: Design an ice cream cone in black and white with only a single occurrence of each of the basic shape functions in the code. These tight constraints force the students to focus on the characteristics of the shapes, and how they can position, size, and rotate these shapes to achieve an effective design.

The most important aspect of this exercise is obviously to create a design that a majority of users will recognize as an ice cream cone. Whether or not a design accomplishes this is a rather objective task. Of the two designs below, it is clear that the first design manages to solve the assignment while the latter does not. It is also easy to analyze why: Although the shapes are almost identical between the two designs, the latter does not establish the proper visual relationships.

A second (and more subjective) aspect relates to the style of the design. To a certain degree, different styles fit different scenarios. If you are asked to create an icon for a website, an abstract style might be preferred, as simple designs work better in small sizes. On the other hand, if you are making an illustration for a children's book, something more bold and playful might be a better fit. Style can be used to serve a specific function, or it can be used purely for aesthetic qualities (which can also be a function, after all). Finally, the style of a design is where the subjective preferences of the designer is apparent.

In this exercise, I often encourage my students to practice designing in different styles, as it further develops their visual language. An important ingredient in the creation of style is the use of the `fill()`, `stroke()` and `strokeWeight()` functions. As demonstrated below, these function can drastically alter the style of a design.

Equally proportioned shape with the same stroke results in an abstract design.

Thicker strokes and fills makes for a more playful design.

This exercise also encourages students to think systematically when implementing their designs in code. The three basic shape functions can only appear once in the code, so students need to use loops to draw more intricate designs. The following three examples are all by former students of mine, and they are displayed here as successful examples of all the things mentioned above: They all objectively solve the assignment of constructing an ice cream cone out of basic shapes. They all achieve widely different styles by using a clever combination of the basic shape functions and the following relationships: position, size, rotation, fill, line and stroke weight. Finally, they all use repetition (to which we will dedicate an entire part of this book) to draw more than three shapes on the canvas.

Design by Luisa Pereira. [See Code](#)

Design by Shir David. [See Code](#)

Design by Tan Ma. [See Code](#)

When you feel comfortable designing with basic shapes, it is time to introduce more complex shapes in your design process. We will do this in the next chapter by looking at a few foundational concepts from computational geometry before venturing into procedural shape

generation.

EXERCISE

Design an ice cream cone in black and white with only a single occurrence of each of the basic shape functions in the code.

Shape

Custom shapes

Although it is a good exercise to design only with simple shapes, complex shapes offer more possibilities. In a manual design process, complex shapes often take a long time to draw, as every detail of a design will need to be created by hand. Although efforts have been made to automate such tasks, some designs are still tedious to create in current digital design tools like Adobe Illustrator or Sketch. This is particularly true for designs that require the use of repetition or randomization, like a pattern of Sine curves with changing amplitudes. In code, we have the ability to procedurally generate very complex shapes in an instant, and the code required can be quite simple. On the other hand, shapes drawn randomly with a pen can be hard to recreate in code, especially if there is no underlying rule to explain the outline of the shape.

In the following chapters, I will introduce a range of techniques to procedurally draw custom shapes. However, we must first understand the basic concepts of drawing shapes in code, which means looking at the `beginShape()` function, as well as the many vertex functions that can be used to define the outline of a shape.

Programming custom shapes

Most graphics programming languages allow you to draw custom shapes like a Connect the Dots drawing: You define a series of points – which we will refer to as vertices – that are connected via lines to form the outline of a shape.

Each vertex in a shape determines how it is connected to the vertex before it. If it is a simple vertex, it will be connected with a straight line. If it is a curved vertex, it will be connected with a curved line. The shape can optionally become a closed shape by connecting the last vertex to the first vertex. P5.js follows this same concept. Use the `beginShape()` function to start a new custom shape, define the vertices of the shape with the desired vertex functions, and finally connect the lines in the shape by calling the `endShape()` function with an optional argument to close the shape. In the following, we will examine these vertex functions.

Straight lines

The `vertex()` function creates a simple vertex that connects to the vertex before it with a straight line. This is the simplest of the vertex functions, and all shapes created with `beginShape()` must start with a `vertex()` function call to define the starting point of the shape. This is illustrated in the example below. Try dragging the vertices to see the resulting code.

The following examples are all created with simple vertices, but use strokes and fills to achieve very different designs.

[See Code](#)

[See Code](#)

[See Code](#)

Bézier curves

To create a vertex that is connected to the vertex before it with a curved line, we use the `quadraticVertex()` and `bezierVertex()` functions. These are a bit more complex than the `vertex()` function, because they need several `x` and `y` coordinates to control the curve of the line. To understand how this works, let us have a brief look at the concept of Bézier curves.

The Bézier curve algorithm was popularized by Pierre Bézier in the 1960's as a solution to a common problem in computational geometry: Drawing curved lines that can scale to any size. The Bézier curve algorithm solves this problem in a very elegant way by introducing the idea of control points: Invisible gravity points that attract the line to bend it into a curve. A Bézier curve with a single control point is called a quadratic Bézier, while a Bézier curve with two control points is called a cubic Bézier. If you have ever used the Pen tool in Adobe Illustrator, you are already familiar with this concept.

This animation shows how a quadratic Bézier curve is calculated.

This animation shows how a cubic Bézier curve is calculated.

You can draw a quadratic bezier curve with the `quadraticBezier()` function, passing the coordinates for the single control point and the vertex itself. Likewise, you can draw a cubic Bézier curve with the `bezierVertex()` function, passing coordinates for the two control points and the vertex itself. The only difference between the two functions is the addition of an extra control point in the `bezierVertex()` function, which allows you to draw more sophisticated curves. This is illustrated below where both types of curves are used to draw a custom shape. Try dragging the vertices and control points to see the resulting code.

It takes a bit of practice to master the Bézier functions, and knowing how many Béziers you need to draw a specific shape can be hard in the beginning. It does not help that control points are invisible, so it can be helpful to spend some time playing around with the example above before diving into the code. Below are three examples that all use the Bézier functions to create custom shapes.

See Code

See Code

See Code

Contours

While we can draw most shapes with `vertex()`, `quadraticVertex()`, and `bezierVertex()`, these functions won't allow us to create shapes with holes. In P5.js, a hole is called a contour, and you can draw shapes with contours using the `beginContour()` and `endContour()` functions. In essence, the `beginContour()` function instructs P5 that you are starting a new shape that will be subtracted from your main shape. Like `beginShape()`, you use the vertex functions to draw your contour, and use `endContour()` to end the contour.

See Code

```
beginShape();  
  // draw rectangle here  
  beginContour();  
    // draw triangle here  
  endContour();  
endShape();
```

Wet and Sharp

You can practice designing custom shapes by continuing the ‘design a word’ exercise from the previous chapters. My assignment to students sound something like this: Make a design with two shapes in black and white that represents the words ‘wet’ and ‘sharp’. There are several reasons why this is a challenging assignment. First of all, the student has to consider how the outline of a shape can help communicate either of those words. Most designs end up using curved vertices to represent wet and simple vertices to represent sharp, but some designs cleverly achieve the goal by doing the opposite. Also, the fact that these shapes exist in the same canvas encourages the student to consider how the shapes can interact with each other to achieve a more dramatic effect. Pointing a knife-like shape directly at a smooth shape will create a certain tension which would not exist if the shape pointed in the other direction.

By Luna Chen. [See Code](#)

By Sean McIntyre. [See Code](#)

The examples in this chapter have a lot of vertices meticulously defined in code, exactly like you would draw them with the mouse. This is of course not the ultimate promise of algorithmic design. Why make shapes in code when they are faster to draw with a mouse? In the following chapters, we will look at a number of techniques that can be used to draw shapes in a more procedural way.

EXERCISE

Create a design with two shapes in black and white representing the words ‘wet’ and ‘sharp’. The shapes have to be created with the

`beginShape()` and `endShape()` functions.

Shape

Procedural Shapes

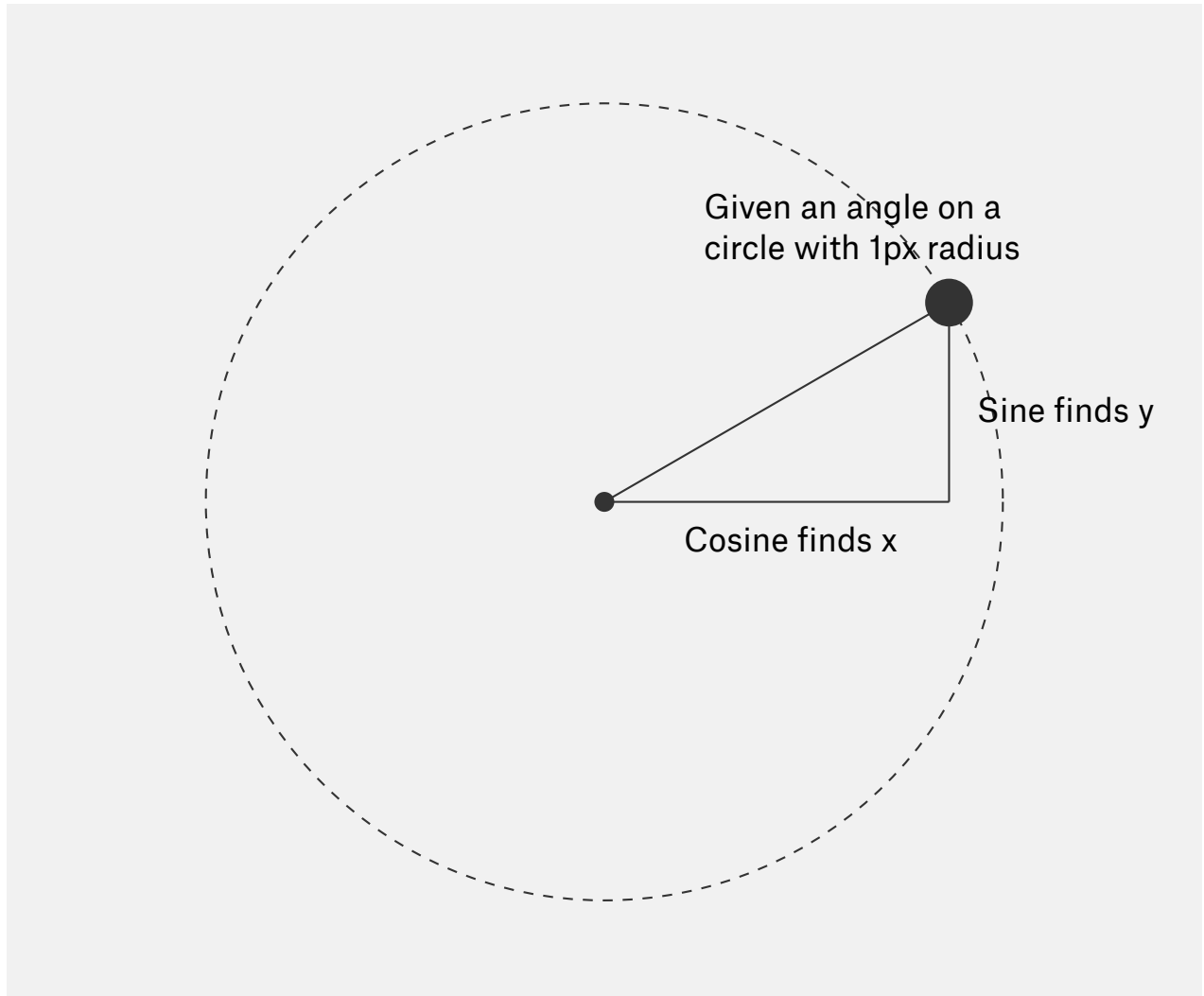
The code for our custom shape examples has so far been rather tedious. We have manually created shapes by typing line after line of vertex function calls, and this strategy will not work for more complex shapes. Also, given the title of the book, this approach might seem a bit anti-climatic. Now that we understand the basics of `beginShape()`, let us have a first look at how to procedurally draw custom shapes using a for loop, and the `sin()` and `cos()` functions.

Sine and Cosine

Over the years, I have seen many students struggle with Sine and Cosine. It is easy to understand why: These words seem rather scary and abstract, especially if you do not consider yourself good at math. However, this is both unfortunate and unnecessary. Unfortunate, because these two functions are a fundamental part of most programmatic designs, and a good understanding of them will allow you to solve many visual problems. Unnecessary, because they are not that hard to learn. Even if you do not understand everything presented in this chapter, you can get started by memorizing two almost identical lines of code.

Sine and Cosine allow us to find any position on the outline of an ellipse. They do this by converting an angle into an `x` position (Cosine) or a `y` position (Sine) for a circle with a 1 pixel radius. These values can then be multiplied by the radius of your actual circle to scale them up. Although it is not strictly necessary to understand how these functions work internally, here is a way to visualize what is going on: Imagine a right sided

triangle connecting the center of the circle to the point on the outline. The Sine function is a quick way to get the ratio between the left side (hypotenuse) and the right side (opposite) of that triangle. The Cosine function is likewise the ratio between the hypotenuse and the bottom side (adjacent) of the triangle.



In P5, these functions are called `sin()` and `cos()`. As described above, they accept a single argument – an angle in radians – and return a value between `-1` and `1` representing the `x` or `y` position on a tiny circle. The two lines below demonstrate how to get these values and multiply them by the radius of your actual circle. Memorize these two lines, as they are very important.

```
const x = cos(RADIANS) * RADIUS;  
const y = sin(RADIANS) * RADIUS;
```

To put this into context, here is an example where we use the same code to draw a small circle 330 degrees along the outline of a bigger circle.

```
translate(width/2, height/2);  
  
noFill();  
const radius = width * 0.3;  
ellipse(0, 0, radius*2, radius*2);  
  
fill(30);  
const x = cos(radians(330)) * radius;  
const y = sin(radians(330)) * radius;  
ellipse(x, y, 20, 20);
```

If you consider all the basic shapes – as well as many complex shapes – they are characterized by having non-overlapping outlines that move around a center point. Some shapes, like the triangle, have just a few vertices, while others – like the ellipse – have many vertices. The `sin()` and `cos()` functions give a way to procedurally draw these types of shapes.

The For Loop

Although we will dedicate an entire part of this book to repetition, let us briefly go over the basic functionality of a `for` loop. A `for` loop allows us to execute code multiple times in a row by incrementing (or decrementing) a variable – often called `i` – until an expression is no longer true and the loop stops. In the following example, we initialize a variable with the number `0`, iterate as long as our variable is lower than `10`, and increment our variable by one between each iteration. The result

is a loop that iterates ten times with our variable incrementing from zero to nine, drawing ten rectangles on the screen.

```
for(let i = 0; i < 10; i++) {  
  rect(0, 0, 100, 100);  
}
```

Unfortunately, all these rectangles have identical positions and sizes because we are passing the same static numbers to the `rect()` function over and over again. This is where `i` comes into play: Because it changes between each iteration of the loop, it can be used to create variance between each rectangle. The example below uses `i` to position the ten rectangles one pixel apart on the x-axis.

```
for(let i = 0; i < 10; i++) {  
  rect(i, 0, 100, 100);  
}
```

Although it might not be immediately clear, this is an important technique when drawing procedural designs. Because `i` increments by one between each iteration, it can be used as a scalar to distribute shapes across the canvas. For example, if we want to position the rectangles next to each other, we can multiply `i` with a number greater than the width of the rectangles.

```
for(let i = 0; i < 10; i++) {  
  rect(i * 105, 0, 100, 100);  
}
```

We can use this same technique to draw custom shapes. Instead of drawing individual shapes in the loop, we use the for loop to procedurally

add vertices between the `beginShape()` and `endShape()` function calls. In the example below, we use this technique to draw ten random vertices in the center of the canvas.

```
translate(width/2, height/2);
beginShape();
for(let i = 0; i < 10; i++) {
  const x = random(-100, 100);
  const y = random(-100, 100);
  vertex(x, y);
}
endShape();
```

The result is certainly a procedural shape, but the use of `random()` does not give us a lot of control over the placement of the vertices: The shape is just a bunch of lines randomly crossing each other. The final step is to put our two techniques together and generate shapes with `sin()` and `cos()` inside of a `for` loop.

Putting it together

Starting from our random shape code above, let us replace the random vertices with vertices placed sequentially along the outline of an ellipse. We do this by using the same two lines that we memorized earlier, but instead of passing the same angle to `sin()` and `cos()`, we calculate a different angle on every iteration by multiplying `i` with the angle we want between the vertices. The result is a shape with ten vertices evenly spread around the center of the canvas.

```
translate(width/2, height/2);
beginShape();
for(let i = 0; i < 10; i++) {
  const x = cos(radians(i * 36)) * 100;
```

```
    const y = sin(radians(i * 36)) * 100;
    vertex(x, y);
  }
endShape();
```

By changing the number of iterations and the spacing between the vertices, you can draw all of the basic shapes. The code below adds a few variables on top of the sketch to automatically calculate the spacing based on the number of vertices. Change the `numVertices` variable and another shape will appear.

```
const numVertices = 3; // or 4 or 30
const spacing = 360 / numVertices;
translate(width/2, height/2);
beginShape();
for(let i = 0; i <= numVertices; i++) {
  const x = cos(radians(i * spacing)) * 100;
  const y = sin(radians(i * spacing)) * 100;
  vertex(x, y);
}
endShape();
```

‘Great, we have reinvented the basic shape functions’ you might say. Actually, this technique allows us to draw much more sophisticated shapes. Let’s run through a few examples that all use the same `sin()` and `cos()` formula to draw different types of shapes. We’ll start with the squiggly circle below that has a random radius for each vertex, making it look like it was drawn by hand.

```
translate(width/2, height/2);

beginShape();
```

```

for(let i = 0; i < 100; i++) {
  // Change the radius for every vertex
  const radius = 100 + random(5);
  const x = cos(radians(i * 3.6)) * radius;
  const y = sin(radians(i * 3.6)) * radius;
  vertex(x, y);
}
endShape();

```

The star below is created by alternating between a low and a high radius for each vertex. It's easy to tweak the style of the star by using different numbers or more vertices, or using `rotate()` to change the orientation of the star.

```

translate(width/2, height/2);

// Set the initial radius to 100
let radius = 100;
beginShape();
for(let i = 0; i < 10; i++) {

  // Use the radius in the cos/sin formula
  const x = cos(radians(i * 36)) * radius;
  const y = sin(radians(i * 36)) * radius;
  vertex(x, y);

  // Change the radius for the next vertex
  if(radius == 100) {
    radius = 50;
  } else {
    radius = 100;
  }
}
endShape();

```

Here is a flower created with `quadraticVertex()` where all vertices and control points are positioned using `sin()` and `cos()`. By using a larger radius for the control points (the inverse of the star example above), the curves go outwards. When using Bézier curves, remember to start the shape with a `vertex()` function call. We do this by checking the value of `i` within the loop.

```
translate(width/2, height/2);

// Automatically calculate the spacing
const numVertices = 7;
const spacing = 360 / numVertices;
beginShape();
// Loop one extra time to close shape with a curved line.
for(let i = 0; i < numVertices+1; i++) {
  // Find the position for the vertex
  const angle = i * spacing;
  const x = cos(radians(angle)) * 100;
  const y = sin(radians(angle)) * 100;
  if(i == 0) {
    // If this is the first run of the loop, create simple vertex.
    vertex(x, y);
  }
  else {
    // Otherwise create a quadratic Bézier vertex with a control point
    // halfway in between the points and with a higher radius.
    const cAngle = angle - spacing/2;
    const cX = cos(radians(cAngle)) * 180;
    const cY = sin(radians(cAngle)) * 180;
    quadraticVertex(cX, cY, x, y)
  }
}
endShape();
```

You will often find yourself needing to use just one of the circular functions. The two shapes below are created just like that: The first one uses `sin()` and the second one uses `cos()` (as demonstrated in the code below).

```
strokeWeight(20);
strokeCap(SQUARE);
translate((width/2) - 200, height/2);
beginShape();
for(let i = 0; i < 200; i++) {
  // 2 pixel spacing on the x-axis.
  const x = i * 2;
  // 200 pixel high waveform on the y-axis.
  const y = cos(i * radians(2)) * 100;
  vertex(x, y);
}
endShape();
```

Sine and Cosine can be used to create a range of different shapes during the design process. In this design by Josef Müller-Brockmann, a series of exponentially growing arcs are rotated around the bottom left of the canvas.

beethoven

tonhalle grosser saal
dienstag, den 22. februar 1955,
20.15 uhr
4. ekstrakonzert
der tonhalle-gesellschaft

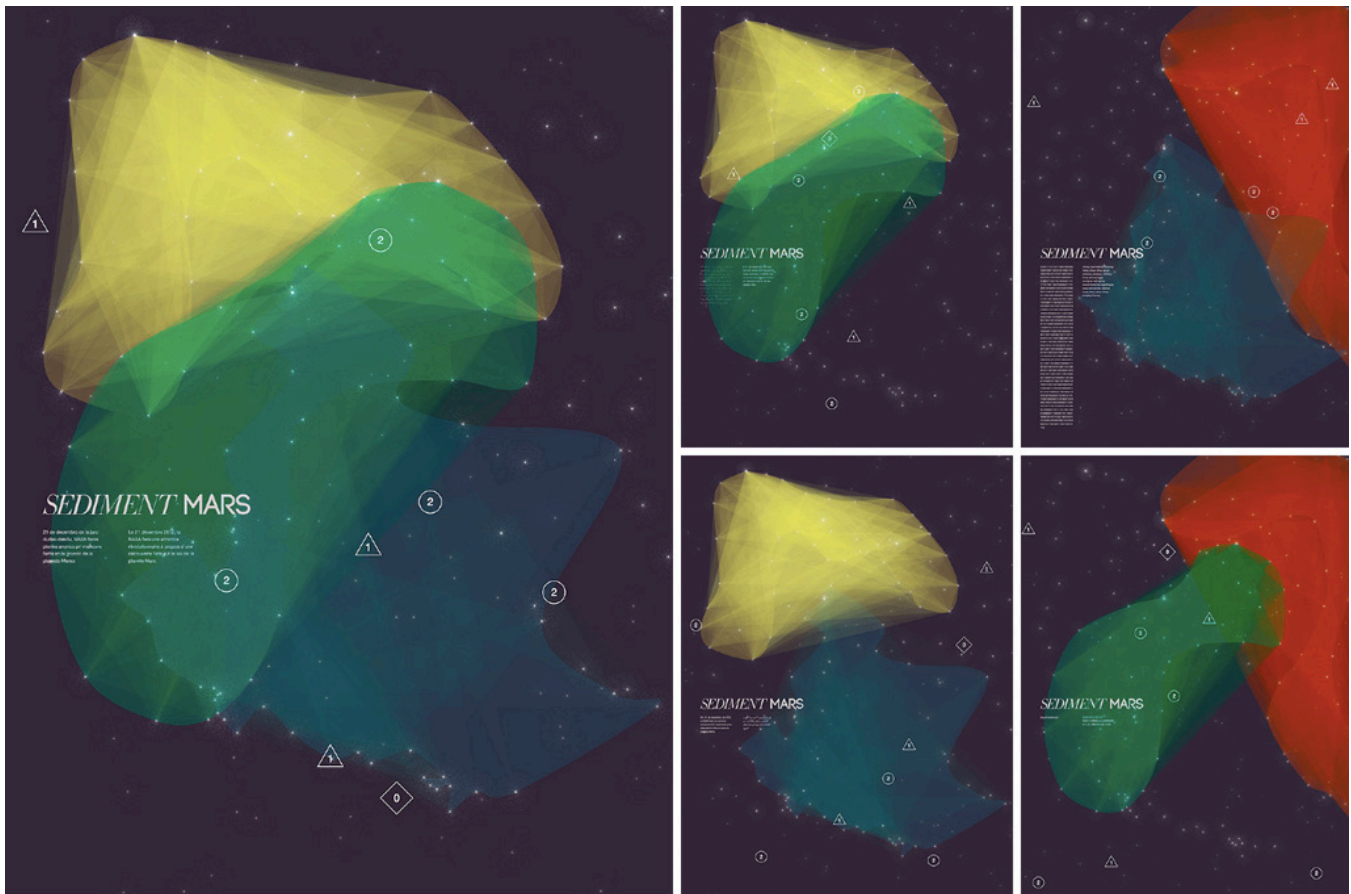
leitung carl schuricht
solist wolfgang schneiderhan

beethoven ouverture zu «coriolan», op. 62
violinkonzert in d-dur, op. 61
siebente sinfonie in a-dur, op. 92

vorverkauf tonhalle-kasse, hug, jecklin,
kuoni
karten zu fr. 3.50 bis 9.50

Beethoven poster by Josef Müller-Brockmann ©

Sediment Mars is a series of generative poster designs by Sarah Hallacher and Alessandra Villaamil. The `sin()` and `cos()` functions are used to generate an elliptical shape, which is then distorted by adding random values to it.



Sediment Mars by Sarah Hallacher and Alessandra Villaamil ©

The project Generative Play is a card game by Adria Navarro that uses procedural drawing to create an infinite amount of generative characters. The character bodies are created using `sin()` and `cos()`.



Generative Play by Adria Navarro ©

This chapter introduced an approach to design that is inherently different than a traditional design process. Rather than individually placing each shape on the canvas, we wrote algorithms to do this for us. Using loops to procedurally draw shapes is a powerful concept, as it allows designers to do more with less code, thus alleviating us from the pains of manually constructing every design object by hand. This is also the hardest thing about procedural design, as designers need to devote more time up front distilling the system into code, and they cannot easily manipulate individual shapes like in a traditional design tool. The American computer scientist Donald Knuth calls this a transition from design to meta-design:

“Meta-design is much more difficult than design; it is easier to draw something than to explain how to draw it. [...] However, once we have successfully explained how to draw

something in a sufficiently general manner, the same explanation will work for related shapes, in different circumstances; so the time spent in formulating a precise explanation turns out to be worth it.”

Donald Knuth (1986), The Metafont Book

This is also the main thesis of this book. When designers learn to not only think systematically about the design process, but also to implement those systems in software, they can build things that were not possible before.

EXERCISE

Try to draw all the basic shapes using the techniques presented in this chapter. Then continue to generate other types of shapes. Can you use `random()` to manipulate the shape outline? Can you use Bézier curves instead of simple vertices?

Color

A short history of color theory

Of all the subjects presented in this book, this part devoted to color theory might be the most perplexing one. Although a basic understanding of the color spectrum is rather easy to develop, color theory is an almost infinitely complex subject with roots in both science and art. It can therefore be a daunting task to learn about color composition in a way that is true to both art history and scientific truth, and I have seen many designers stumbling on the most basic of questions: Is yellow a primary color? Which color combinations are harmonic? What is the true

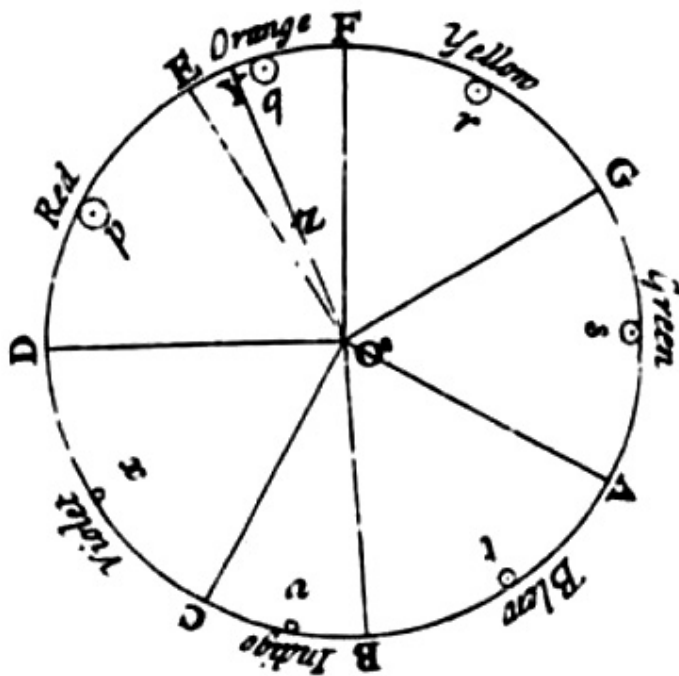
complementary color to blue?

I hope that this chapter on the history of color theory can help answer some of these questions by highlighting both the mistakes and successes of key figures in the field. In this abbreviated and narrow introduction, I am especially interested in the conflict between the two distinct but related fields that both operate under the term 'color theory': Artistic color theory, which is concerned with the visual effects of color combination in the fine arts, and scientific color theory, which describes the nature of color through increasingly complex but precise color models. The following chapters will build on lessons learned in this chapter, and it is my belief that it is essential for designers to develop a solid understanding of this history in order to make good decisions about color.

One of the first known theories about color can be found in *On Colors*, a short text written in ancient Greece. The text was originally attributed to Aristotle, but it is now widely accepted to have been written by members of his Peripatetic school. Based on observations of how color behaves in nature, the text argues that all colors exist in a spectrum between darkness and light, and that four primary colors come from the four elements: fire, air, water, and earth. This can seem rather weird and speculative today, but these observations made sense at the time: A plant is green above ground and white in its roots, thus the color must come from the sun. Likewise, a plant left to dry will lose its vivid colors, thus water provides color too. This theory is typical of how color theorists for centuries used color to establish a general theory of the universe. Despite the erroneous theory, *On Colors* has a series of important observations, like the fact that "darkness is not a colour at all, but is merely an absence of light"¹ – a discovery propelled by watching how clouds darken as they thicken².

Like so many other areas of science, Isaac Newton completely redefined

the conventional theories on the behavior of light when he published the first edition of *Opticks* in 1704. Rather than seeing light as a void of color, Newton discovered that white light is a combination of all colors across the color spectrum. The basics of his experiments was a well-known phenomena: When you shine white light through a prism, the light is split into colors from across the color spectrum. However, Newton discovered that he could recombine these spectral colors to once again turn them into white light.



Newton's color circle used seven colors mapped to a musical octave starting at the tone D.

Newton also discovered that if he blended the first color (red) and last color (violet) of the color spectrum, he could produce magenta, an extra-spectral color that does not exist in the rainbow. This prompted him to wrap the color spectrum into a circle, beginning a tradition of using basic shapes to represent the

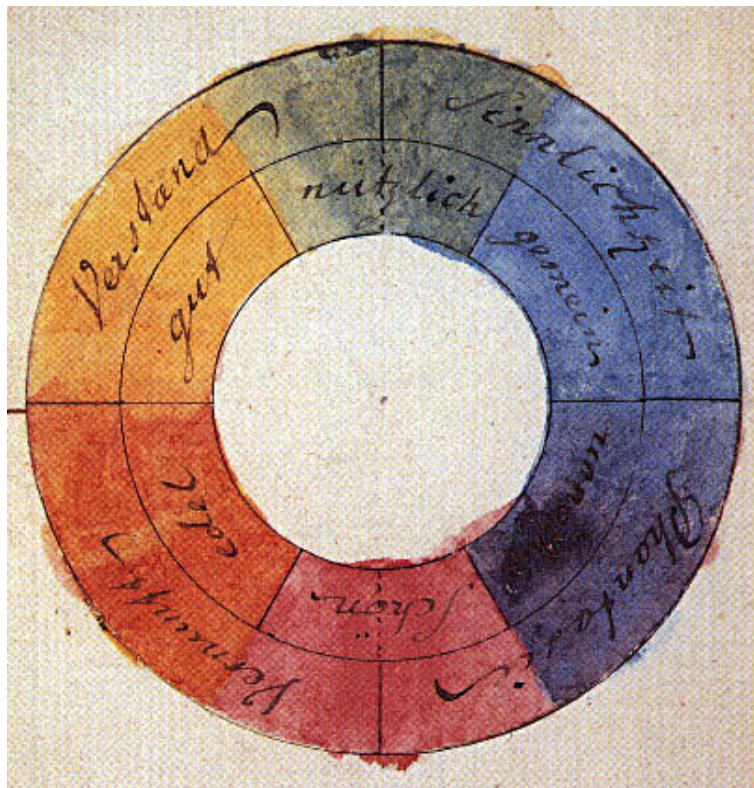
relationship between colors. Newton used a circle because it could be used to predict the result of color mixing for two colors by pointing to the color midway between these colors. The colors on Newton's circle have asymmetric distances to each other because Newton wanted the circle to have seven colors – the exact number of days in a week and musical notes in an octave³.

While Newton was interested in a scientific explanation of color, the German poet Wolfgang von Goethe dedicated his book *Theory of Colors* from 1810 to a more human-centered analysis of the perception of color. Through a series of experiments that measured the eye's response to certain colors, Goethe created what is arguably the most famous color circle of all time. The circle had three primary colors – magenta, yellow, and blue – which he believed could mix all other colors in the spectrum.

This publication was in many ways at odds with Newton's theories, as Goethe believed that the prism, not the light, was responsible for the creation of color, and that darkness was not an absence of light. Although Newton eventually won the argument about the nature of light, Goethe's work is important to us because it focuses on the cognitive effect

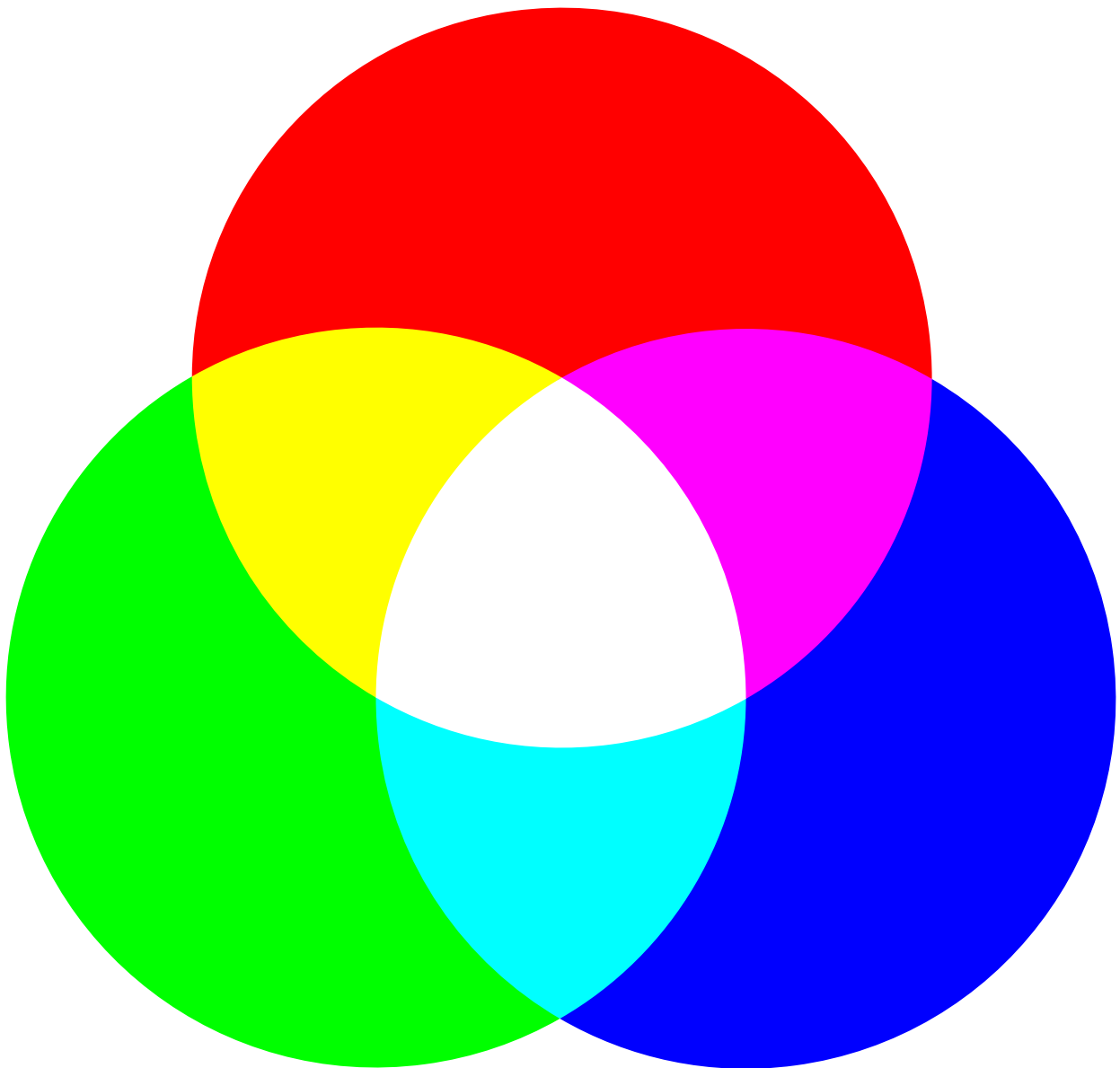
that color has on humans. His research on the effects of after-images and optical illusions is especially interesting, because it points towards the later works of Johannes Itten and Josef Albers⁴.

Even though Newton's and Goethe's color circles may seem to be at odds with each other, they are in some way both correct as they illustrate the

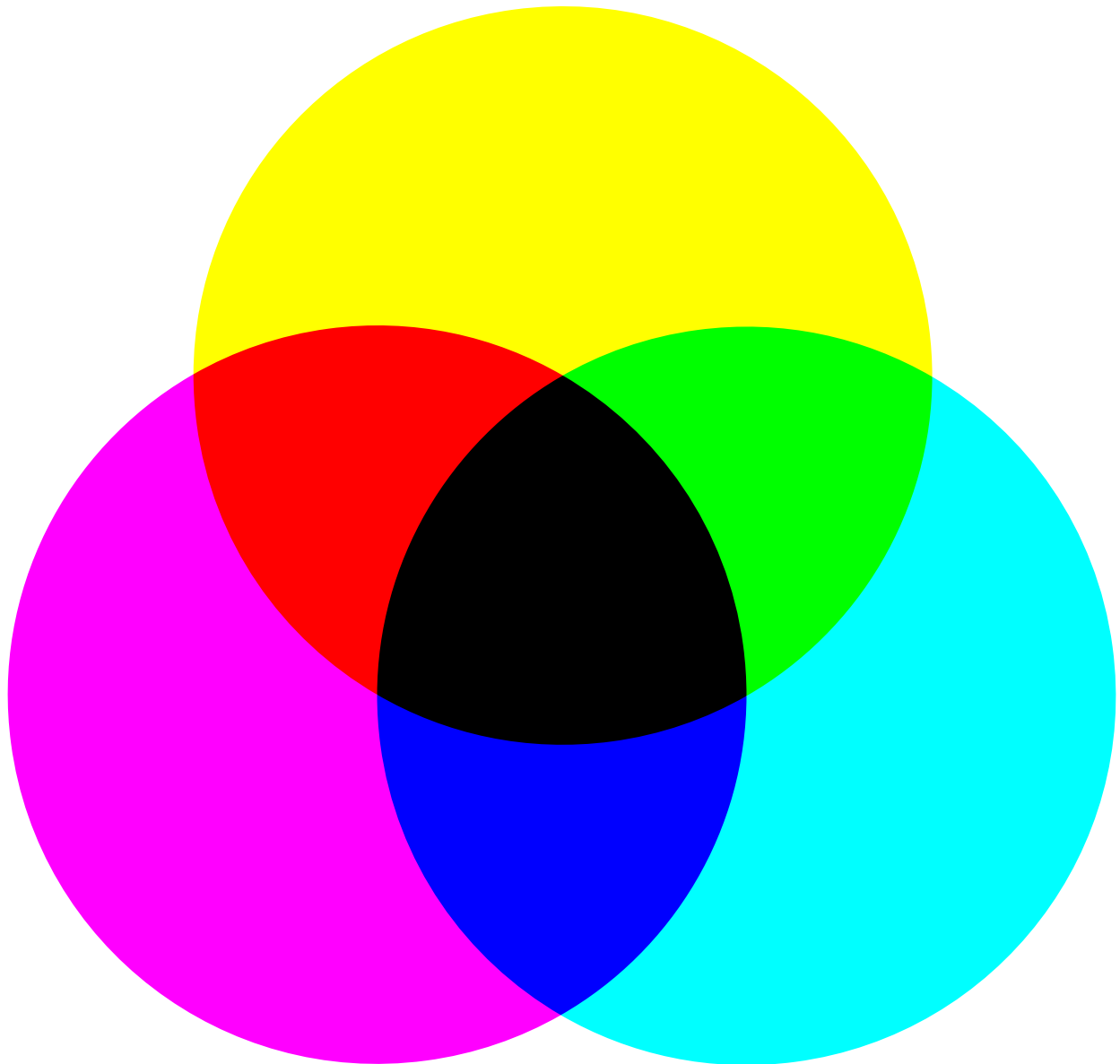


Goethe's color circle with magenta, yellow, and blue primary colors.

behavior of color in different material. Newton describes how his spectral colors can mix most visible colors including white, and this is true because light mixes in an additive way: Combining lights of different colors will eventually result in white light. Goethe describes how his three primary colors can mix most visible colors including black, and this is true because pigments mix in a subtractive way: Combining paints of different colors will eventually result in black paint by subtracting waves of light.



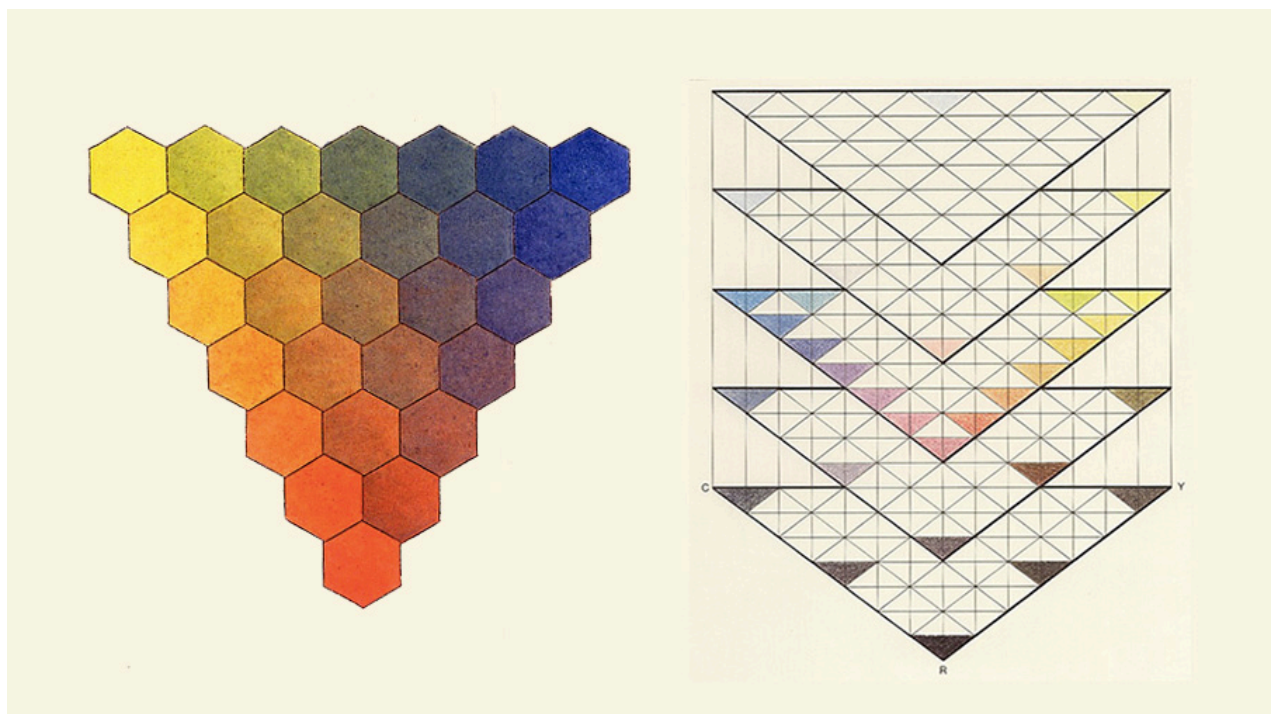
RGB in additive color mixing.



CMY in subtractive color mixing.

In a quest to create a unified notation for color – like we know it from musical notation – artists soon started depicting the color spectrum as 3D solids. A concurrent example of this can be found in Tobias Mayer's color triangle from his book *The Affinity of Color Commentary*, published posthumously in 1775. Mayer sought to accurately define the number of individual colors the human eye can see, and this required him to add another dimension to represent the variations of brightness for each color.

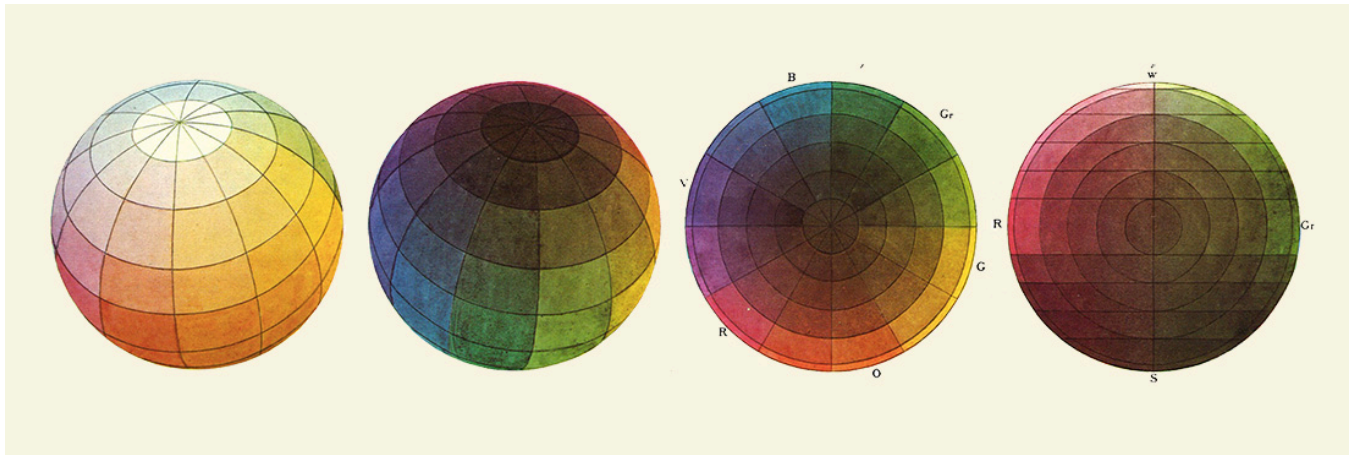
Mayer painted the corners of a triangle with the three traditional primary colors from painting – red, yellow, and blue – and connected the corners by mixing the opposing colors together. Unlike the traditional color circle, he created many variations of this triangle by stacking triangles of different brightnesses on top of each other. This made it possible to define a color by its position within a 3D space, a technique still used to this day. Mayer ultimately failed at creating a color model with perceptually uniform steps, as he did not understand the irregularities of the human eye⁵.



Tobias Mayer's color triangles.

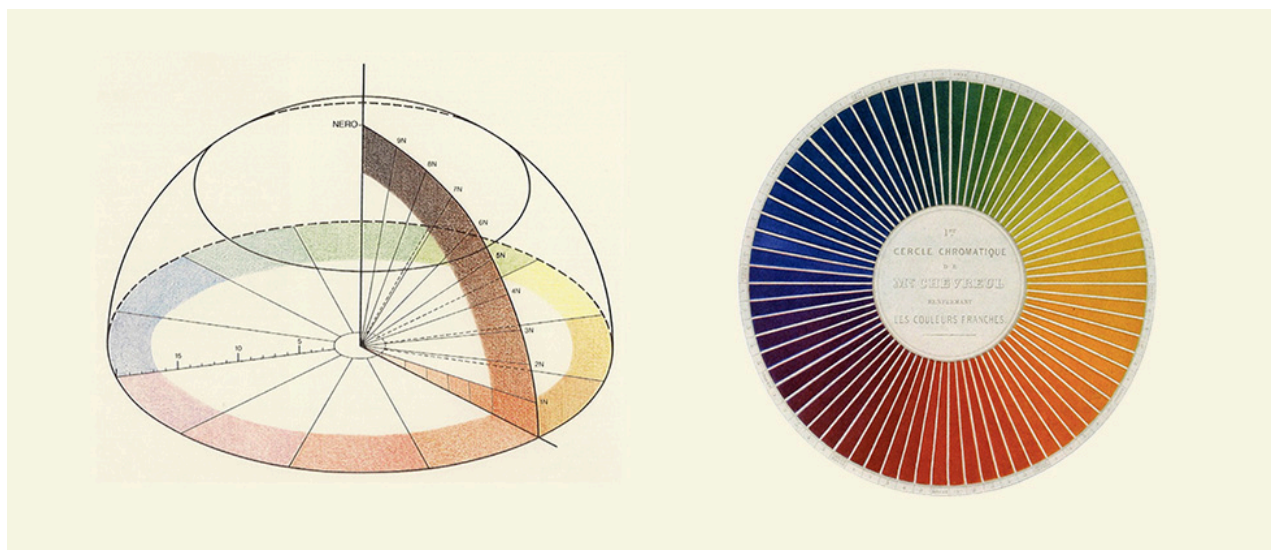
The German painter Philipp Otto Runge took this same approach when creating his spherical representation of the color spectrum, published in his *Color Sphere* manuscript in 1810. Runge's sphere had white and black poles with colored bands running between them. However, like many other representations of color before it, the model did not differentiate between brightness and saturation, which meant that the resulting model had little variation in color intensity. This sphere had the same problem as Mayer's

triangle, as the steps were not perceptually uniform⁶.



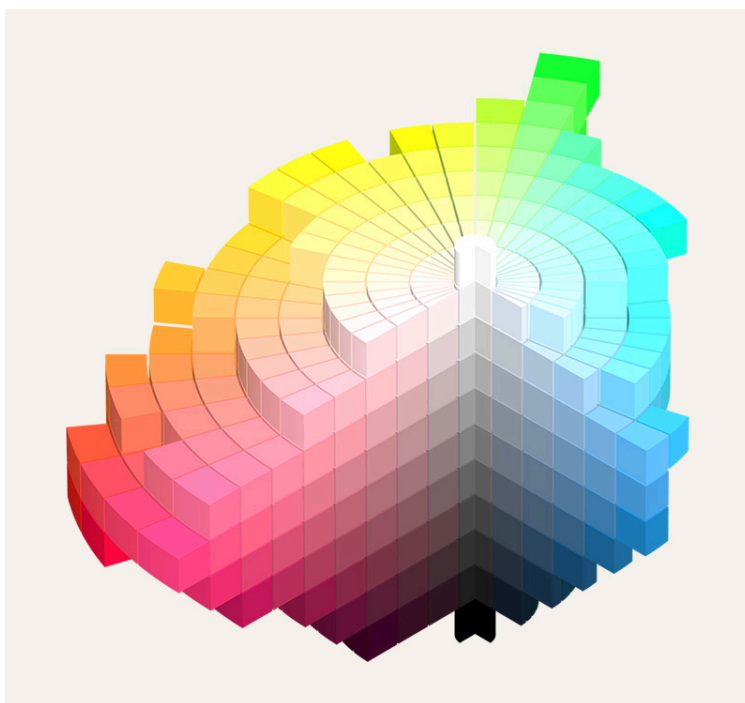
Philipp Otto Runge's color sphere.

Michel Eugène Chevreul attempted to fix this problem in his hemispherical color system from 1839. Rather than mixing colors by focusing on the amount of paint used, he based his selections solely on what perceptually appeared to be the correct mixture. Inspired by the work of Goethe, Chevreul used after-images to test the validity of his mixtures. When a person stares at a green square for a long time and then looks at a white wall, a magenta square will appear. This happens because of fatigue in the green photoreceptors in the eye, and Chevreul used this to establish the complementary colors in his model⁷.



Michel Eugène Chevreul's color sphere.

One of the most historically significant color solids was created by the American painter Albert Henry Munsell in the early 1900's. Like his peers before him, Munsell wanted to create a model with perceptually uniform steps, and although he was a painter, his approach was very scientific: He used human test subjects and a range of mechanical instruments he invented to create a remarkably accurate model. One important detail about Munsell's color system is that he divided the color space into three new dimensions: The hue determined the type of color (red, blue, etc), the value determined the brightness of the color (light or dark), and the chroma determined the saturation of the color (the purity of the color). These dimensions are still used to this day in some representations of the RGB color model.



Visualization of Albert Henry Munsell's color tree from the 1943 renovation.

Munsell first tried to arrange his colors in a sphere, but noted that "the desire to fit a chosen contour, such as the pyramid, cone, or cube, coupled with a lack of proper tests, has led to many distorted statements of colour relations"⁸. Essentially, Munsell realized that his color solid had to have an irregular shape to fit his colors. The explanation for this is

rather simple. Colors with low brightness have much fewer visible colors between zero and full saturation (colors with zero brightness only have one, black). Likewise, some hues have more range than others. You can mix more visible colors between red and white than between yellow and white, because yellow is a lighter color. Another important detail of Munsell's color system is that he preferred the use of a mathematical syntax over color names to indicate a color's position within the color space. This is not unlike how we define colors in programming languages today. Munsell's color system had its flaws and inconsistencies, but it managed to bridge art and science in a way not done before, and it still forms the basis of the curriculum at many fine art institutions.

Many of the European art movements in the early 20th century had a profound interest in the subjective experience of art, and although the Bauhaus school in Germany was a school focused on a modern approach to art, design, and architecture, two important publications on color and perception were written by Bauhaus faculty: *The Art of Color* by Johannes Itten⁹ and *Interaction of Color* by Josef Albers¹⁰.

As a follower of the Mazdaznan religion, Itten's view of the arts was highly influenced by his spiritual beliefs. Following a strict vegetarian diet, he was famous for performing rhythmic breathing exercises with his students in order to have them realize their full creative potential¹¹. In his mind – like Goethe's – it was the subjective experience of color that mattered, and his book focuses on how color can be combined to invoke feelings in the viewer. The central idea in Itten's work is the existence of seven color contrasts that artists must master in order to know the effect of their color choices. Some of these contrasts are simple, like the contrast of light and dark that exists when colors of different brightnesses appear next to each other, or the contrast of hue that can be seen when colors of different hues are used together¹². These observations can still be used by aspiring designers to guide decisions around color, as they give us a way to classify

color and think systematically about their use. Itten even operated with a RYB color sphere remarkably similar to that of Runge to help explain these ideas. Other of Itten's contrasts can seem rather arbitrary, like his law of simultaneous contrast that states how certain colors create visual effects when used together. Itten often uses his own subjective experience to establish a generalized theory on color and perception, as demonstrated in the quote below.

“For the solution of many problems, however, there are objective considerations that outweigh subjective preferences. Thus a meat market may be decorated in light green and blue-green tones, so that the various meats will appear fresher and redder. [...] If a commercial artist were to design a package for coffee bearing yellow and white stripes, or one with blue polka-dots for spaghetti, he would be wrong because these form and color features are in conflict with the theme.”

Johannes Itten¹³

Here, Itten's personal preferences towards the color palette bleed into an unnecessarily strict generalization about color and subject. Who is to say that yellow stripes or blue polka-dots cannot be used effectively when designing food product labeling?

Josef Albers, a student of Itten's at the Bauhaus, took a more demonstrative approach in his *Interaction of Color* from 1963. Using opaque pieces of colored paper, Albers sets out to show the highly dynamic nature of color, particularly how humans tend to perceive a color based on the colors around it. Rather than trying to establish some unified theory about why color behaves this way, Albers describes how students can repeat these experiments to experience it on their own. This has made

The Interaction of Color one of the most important and timeproof books on color composition. Pictured below is one of his most famous examples with two small squares on colored backgrounds. The viewer naturally assumes that the squares are filled with colors from the opposite backgrounds, when they in reality are the exact same color.

These two small squares have the same color. Click the button to verify.

As illustrated above, our art history is full of arguments over the nature of primary colors, which is in part caused by the confusion over subtractive and additive color models. It is notoriously hard to mix yellow from darker paints, which is why Goethe and other artists thought of yellow as a 'pure' color with qualities different from the rest of the color spectrum. We know today that the concept of primary colors is actually a rather arbitrary one, and there is no such thing as 'pure' primary colors for pigments. One can choose any three colors to mix a subset of the spectrum, and although some primaries can mix a wider range of colors, it is impossible to mix the entire visible color spectrum in a subtractive color model.

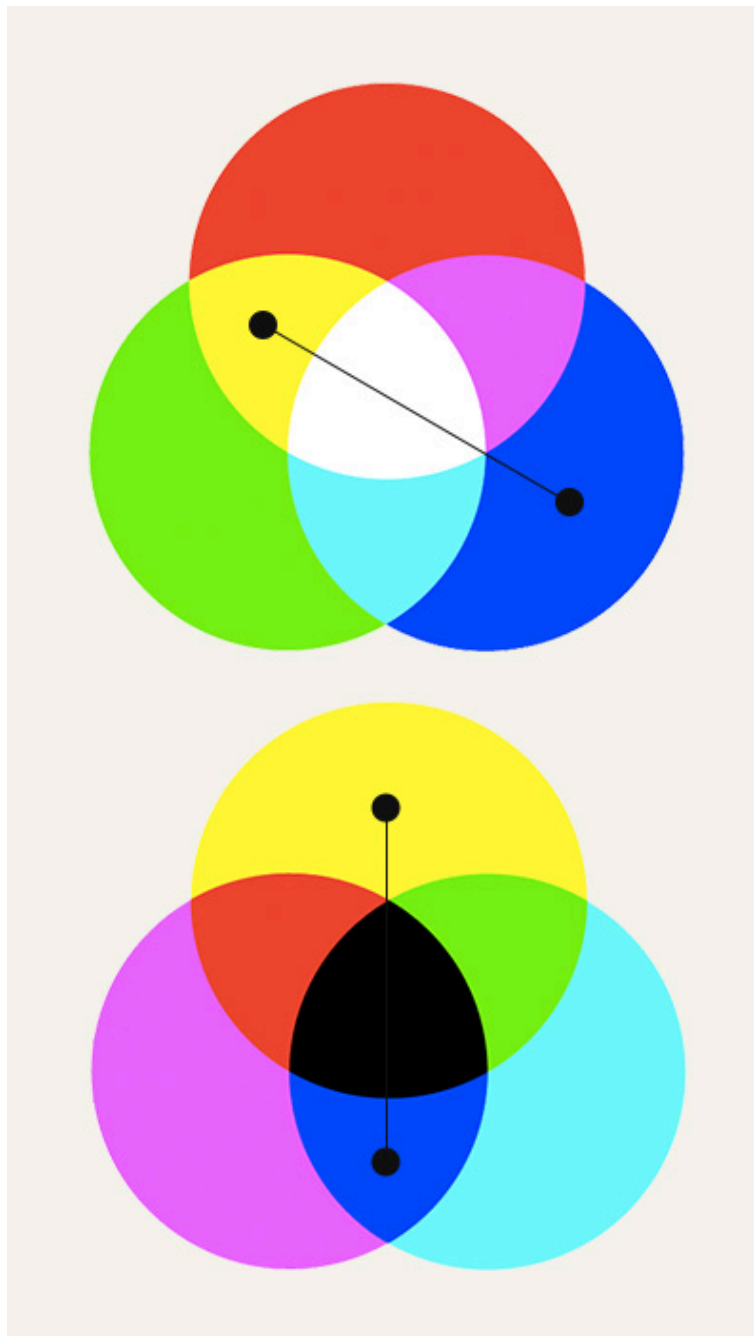
"The conclusion [...] is that primary colors are only useful fictions. They are either imaginary variables adopted by mathematical models of color vision, or they are imperfect but economical compromises adopted for specific color mixing purposes with lights, paints, dyes or inks."

Bruce MacEvoy¹⁴

These discoveries are deeply integrated into the devices that we all use on a daily basis. The industry standard for desktop printers and other pigment-based printing mechanisms with subtractive color mixing is to have three colors based on the CMY color model: cyan, magenta, and yellow. It is now well understood that this particular set of colors can mix an acceptable range of colors in ink. Printers also have a black ink because

these primary colors cannot mix to a true black, and it has the added advantage of saving costly colored ink. However, professional printers can have many more ink cartridges for better color accuracy. Epson, a leader in digital printing technologies, uses ten ink colors in their UltraChrome® HDR technology.

The industry standard for computer screens and other light-based display technologies with additive color mixing is to have three primaries per pixel based on the RGB color model: red, green, and blue. These three colors mix to an acceptable range of the visible color spectrum, where the exact amount is decided by the quality of the monitor, but also the computer's graphics card. Any digital design tool today will allow designers to define colors based on a combination of these three primaries.



A special bonus of the RGB and CMY color models is that even though they have different primary colors, they share complementary colors.

Additive RGB and subtractive CMY share complementary colors.

Just like there is common agreement on the scientific nature of color today, it is also known that the human experience of color is a highly complex and subjective phenomenon. It is generally accepted that it is impossible to create a simple, predictive theory about color harmony – the type of approach that Goethe and Itten believed in. A number of factors determine your response to a specific combination of colors, including gender, age, mood, personal background, and current trends in society¹⁵. In some sense, this should be a relief to aspiring designers. For one, it relieves them from participating in irrelevant discussions about which color circle has the ‘correct’ complementary colors. Also, without a simple algorithm to find harmonic colors, the student has no choice but to use their own eyes.

When reading this account of artists and scientists who dedicated their professional lives to the creation of models that help other artists make educated decisions about color composition, it should be clear that the way designers today interact with color – the color picker – leaves much to be desired. The color picker is as omnipresent as it is broken: With no significant changes over the last decade, it fails to provide a meaningful visual representation of the color spectrum, even though such models have existed for more than 300 years. Instead, it uses a rectangular area to show a single hue at a time, and designers are left with no way to visualize the relationship between the selected colors, or even understand the difference between a perceptually uniform color model and its counterpart. The consequence is that this entire history of color theory is neglected in modern design tools, which means that it is lost on students

too.

Luckily, we are not bound to digital design tools in this book. In the following chapters, we will examine color models, color spaces, and many techniques that can be used to generate color schemes in code. In order to not make the same mistakes as the people before us, these chapters will not seek to propose a unified theory about which colors are best for certain scenarios. Instead, we will get to know the color palette, and learn how to see the effects of different color combinations. This will hopefully lead to students developing a sound theoretical foundation upon which to base their practice.

-
1. Loeb Classical Library (1936) *Aristotle's Minor Works*, p. 7. London
 2. Gottschalk, H. B. (1964) *The De Coloribus and Its Author*, p. 59-85. Hermes 92. Bd..H. 1: JSTOR. Web. 11 Jan. 2017
 3. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 25. University of Chicago Press
 4. Sloane, Patricia (1967) *Colour: Basic Principles New Directions*, p. 28-30. Studio Vista
 5. Lowengard, Sarah (2006) *The Creation of Color in Eighteenth-Century Europe New York*, para. 129-139, Columbia University Press)
 6. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 48. University of Chicago Press
 7. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 175-176. University of Chicago Press
 8. Munsell, A.H (1912) *A Pigment Color System and Notation*, pp. 239. The American Journal of Psychology. Vol. 23. University of Illinois Press
 9. Itten, Johannes (1973) *The Art of Color: the subjective experience and objective rationale of color*. Van Nostrand Reinhold
 10. Albers, Josef (1963) *Interaction of Color*. Yale University
 11. Droste, Magdalena (2002) *Bauhaus*, p. 25. Taschen
 12. Itten, Johannes (1970) *The Elements of Color*, p. 33-44. Van Nostrand Reinhold
 13. Itten, Johannes (1970) *The Elements of Color*, p. 26. Van Nostrand Reinhold
 14. MacEvoy, Bruce. *Color Vision Handprint : Colormaking Attributes*. N.p., 1 Aug. 2015. Web. 11 Jan. 2017.
 15. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
 16. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press

17. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
18. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
19. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*, p. 9. Arthur Niggli
20. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 2. The College Mathematics Journal, Vol. 23
21. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 8-9. The College Mathematics Journal, Vol. 23
22. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 10-11. The College Mathematics Journal, Vol. 23
23. Malik, Peter (2017) *P. Beatty III (P47): The Codex, Its Scribe, and Its Text*, p. 31 - 38. New Testament tools, studies and documents, Vol. 52
24. Fry, Stephen (2008) *The Machine That Made Us*. BBC UK
25. Apollonio, Umbro (2009) *Futurist Manifestos*, p. 104. Tate Publishing
26. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*. Arthur Niggli
27. Gerstner, Karl (1964) *Designing Programmes*. Arthur Niggli
28. <https://www.nytc.com/who-we-are/culture/our-history/#2000-1971-timeline>
29. Peter, Ian (2004) *So, who really did invent the Internet?*. The Internet History Project
30. https://web.archive.org/web/20050106024725/http://www.subtraction.com:80/archives/2004/1231_grid_computi.php
31. Arnheim, Rudolf (1974) *Art and Visual Perception*, p. 8. University of California Press
32. Heider, G.M. (1977) *More about Hull and Koffka*, American Psychologist, 32(5), 383
33. Kroeger, Michael (2008) *Conversations With Students*, p. 27. Princeton Architectural Press
34. Design Systems International was co-founded by the author

Color

Color models and color spaces

While the previous chapter traced some of the important developments in the history of color theory, this chapter takes a deeper look at the current landscape of digital color theory. When working with color in programming languages, one will encounter quite a few terms used – often interchangeably – to describe a color’s position within the color spectrum. In this chapter, we will look at three of these terms – color models, color spaces, and color profiles – and examine why it is important

to develop a decent understanding of these concepts when working with color in code.

Color models

To understand the nature of something, it can be helpful to create a visual representation of the subject. In fact, humans tend to do this quite often, from scribbling notes in lectures, to drawing charts and maps to explain specific datasets. We do this because many of us are visual learners, and seeing something is different than hearing it. Throughout history, artists and scientists have depicted the color spectrum in all sorts of different models, with the goal of turning the abstract concept of the color spectrum into something comprehensible.

A color model is a visualization that depicts the color spectrum as a multidimensional model. Most modern color models have 3 dimensions (like RGB), and can therefore be depicted as 3D shapes, while other models have more dimensions (like CMYK). In the following, we will look at the RGB, HSV, and HSL color models, which are all prevalent in current digital design tools and programming languages. These color models all use the same RGB primary colors, which makes them good examples of how color models can visualize the same color spectrum in widely different dimensions.

RGB is a color model with three dimensions – red, green, and blue – that are mixed to produce a specific color. When defining colors in these dimensions, one has to know the sequence of colors in the color spectrum, e.g. that a mix of 100% red and green produces yellow. The RGB color model is often depicted as a cube by mapping the red, green, and blue dimensions onto the x, y, and z axis in 3D space. This is illustrated in the interactive example below, where all possible color mixes are represented within the bounds of the cube.

The RGB color model is not an especially intuitive model for creating colors in code. While you might be able to guess the combination of values to use for some colors such as yellow (equal amounts of red and green) or the red color used on Coca-Cola bottles (lots of red with a little bit of blue), less pure colors are much harder to guess in this color model. What values would you use for a dark purple? How about finding the complimentary color for cyan? If you cannot find the answer, it is because humans do not think about colors as mixes of red, green, and blue lights.

HSV is a cylindrical color model that remaps the RGB primary colors into dimensions that are easier for humans to understand. Like the Munsell Color System, these dimensions are hue, saturation, and value.

- *Hue* specifies the angle of the color on the RGB color circle. A 0° hue results in red, 120° results in green, and 240° results in blue.
- *Saturation* controls the amount of color used. A color with 100% saturation will be the purest color possible, while 0% saturation yields grayscale.
- *Value* controls the brightness of the color. A color with 0% brightness is pure black while a color with 100% brightness has no black mixed into the color. Because this dimension is often referred to as brightness, the HSV color model is sometimes called HSB, including in P5.js.

It is important to note that the three dimensions of the HSV color model are interdependent. If the value dimension of a color is set to 0%, the amount of hue and saturation does not matter as the color will be black. Likewise, if the saturation of a color is set to 0%, the hue does not matter as there is no color used. Because the hue dimension is circular, the HSV color model is best depicted as a cylinder. This is illustrated in the interactive example below, where all possible color mixes are represented within the bounds of the cylinder.

HSL is another cylindrical color model that shares two dimensions with HSV, while replacing the value dimension with a lightness dimension.

- *Hue* specifies the angle of the color on the RGB color circle, exactly like HSV.
- *Saturation* controls the purity of the color, exactly like HSV.
- *Lightness* controls the luminosity of the color. This dimension is different from the HSV value dimension in that the purest color is positioned midway between black and white ends of the scale. A color with 0% lightness is black, 50% is the purest color possible, and 100% is white.

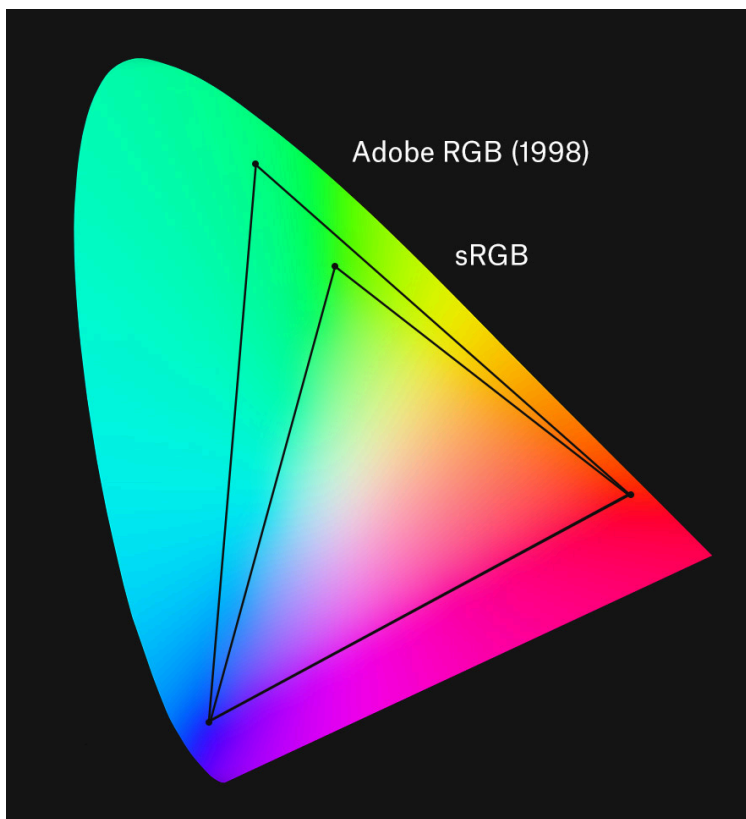
Even though the saturation dimension theoretically is similar between the two color models (controlling how much pure color is used), the resulting saturation scales differ between the models caused by the brightness to lightness remapping. Like HSV, the HSL color model is best depicted as a cylinder, which is illustrated in the interactive example below.

There are plenty of other ways to visualize the color spectrum in a multi-dimensional space. The CMYK color model has four dimensions, which means that one has to use either animation or multiple 3D shapes to visualize the states of the model. Another color model called CIELAB is modeled on the opponent-process theory of human perception with two of three dimensions representing scales from red to green and yellow to blue – two opponent color pairs that humans cannot perceive simultaneously.

Color spaces

Color models provide for a good way to visualize the color spectrum, but they are inadequate when it comes to defining and displaying colors on computer screens. To explain this, let us assume that you own a laptop computer as well as a larger, external screen for your home office. Now, let us also assume that you are running a P5.js sketch showing a yellow

ellipse on both screens. In a world without color spaces, these two screens would turn on their red and green subpixels and be done with it. However, what if your larger screen has more expensive lights that look wildly different from the ones on your laptop screen? This would result in two very different kinds of yellow. This is the problem that color spaces set out to solve.



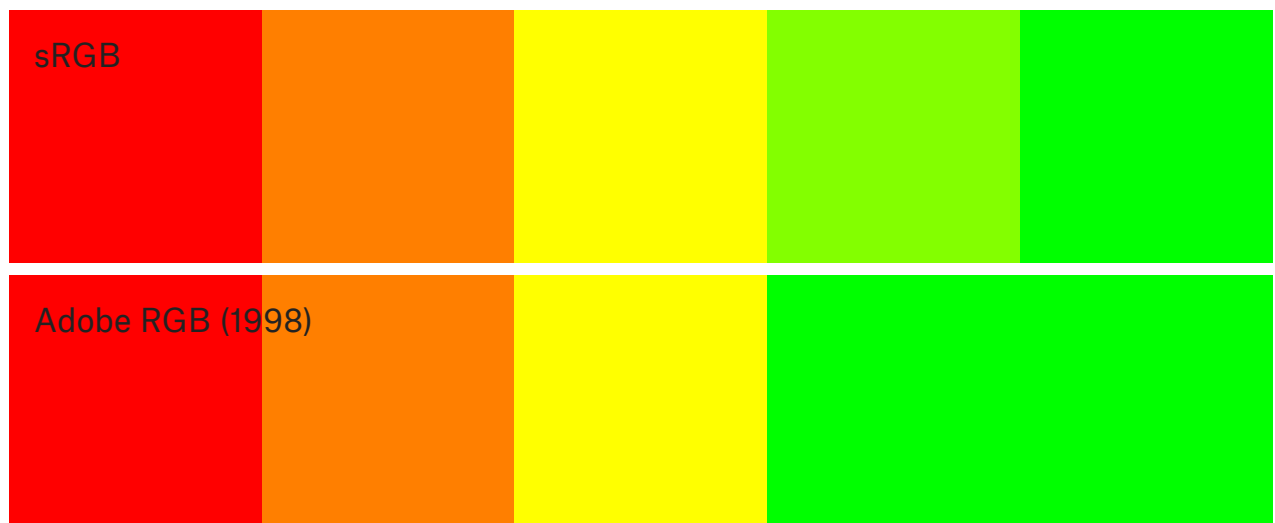
The CIE chromaticity diagram showing the color gamuts of the Adobe RGB (1998) and sRGB color spaces.

This chromaticity diagram was created by the International Commission on Illumination (CIE). It was based on a number of vision experiments on human subjects in the 1930's, and it accurately defines the relationship between the wavelength of a color and the perceived effect on the human eye. This diagram – which is also a color space called CIEXYZ – is

very important as all modern color spaces define their absolute range of colors (called a color gamut) in relation to this color space. The two triangles inside the curved shape indicate the color gamuts of two popular color spaces: sRGB and Adobe RGB (1998). The corners of each triangle define the primary colors of each color gamut, and you might notice that while the two color spaces share the same red and blue primaries, the

green primary color is different between the two. To put it another way, a primary color only has absolute meaning when it refers to a specific color space. In our example from above, color spaces allow your two computer monitors to show an identical yellow color by following a standard process: First, it converts the yellow color from the color space of the P5 sketch to the CIE XYZ color space (also called a reference color space). Then, because every monitor knows the exact color of their primary lights in relation to the CIE XYZ color space, it can determine the amount of primary lights to mix.

The sRGB color space has the smallest color gamut of the two color spaces, which means that it covers the smallest range of colors. It was created for use by computer monitors, and the smaller gamut reflects the exact colors of the primary lights in most HDTVs and computer monitors. This also means that the sRGB color space is easy to adapt for hardware manufacturers, which is why it has become the most widely used color space for digital files. Whenever you come across a color or an image on a website, it is most likely an sRGB color. Even though sRGB is a great color space for the range of colors that can be shown on a screen, the color gamut is not wide enough to support colors printed in ink – especially in the green-blue parts of the spectrum. The Adobe RGB (1998) color space has a much wider RGB color gamut that was carefully chosen to cover most of the colors that CMYK printers can produce. This also means that a specific set of colors can look very different depending on the color space it adheres to, as demonstrated in the example below that shows the same RGB values in the two color spaces. Notice how the last two green colors look identical in the Adobe RGB (1998) color space, because most screens cannot display the green primary color of the wider color gamut.



A simulation of how most monitors render the two color spaces. The last two colors in the Adobe RGB (1998) color space will look identical.

It is important to note that although color models are abstract mathematical concepts, it is impossible to visualize a color model without an accompanying color space. The RGB, HSV, and HSL color model examples from above are all visualized within the sRGB color space, because that is the default color space of the internet.

Color profiles

A digital image can adhere to a specific color space by embedding a color profile in its metadata. This tells any program that wants to read the image that the pixel values are stated according to a particular color space, and images without a color profile are often assumed to be sRGB. Color profiles are important in order to correctly reproduce identical colors across multiple devices, and you will often see professional printing services require image files to be set to a specific color space (most likely Adobe RGB (1998) or ProPhoto RGB, a color space with a very wide color gamut). This assures that the colors in your image are not interpreted to be the wrong color space. If you have ever pasted an image into an existing Photoshop project only to have the colors look wrong, you have been a victim of this. As an example, if you paste an image with an Adobe

RGB (1998) profile into a Photoshop file with a sRGB profile, Photoshop will interpret the pixel values to be within the smaller color gamut, changing the colors of your pasted image. Because of this, most digital design tools have built-in commands to convert between color spaces, and Photoshop actually does a good job of alerting the user before reinterpreting color profiles. Conversion between color spaces is especially beneficial for designers wanting to design print products in code, as their digital assets will need to be converted from sRGB to a print-specific color profile before printing.



The left-hand side of this Paul Klee painting was correctly converted from Adobe RGB (1998) to sRGB, while the right-hand side wrongly reinterpreted the colors into sRGB without conversion. ©

If a digital image uses a color profile with a wide color gamut, it is almost

guaranteed to lose colors on most screens because most screens can only show colors within the sRGB gamut. However, many newer screens support wider color gamuts. The Apple iMac retina screen uses a RGB color space called DCI-P3 with a color gamut that spans about the same range as Adobe RGB (1998), but it includes more red-yellow colors and excludes some green-blue colors. To highlight the complexity of color management, some browsers running on retina computers may oversaturate the colors of sRGB images without color profiles, while other browsers will correctly convert the sRGB pixel values into DCI-P3. Although this book will not dive further into the complicated aspects of color management, there are plenty of [good resources](#) out there for the interested reader.

Color in P5.js

As a browser-based JavaScript library, all color values in P5.js adhere to sRGB, the standard color space for the internet. You can define these colors in all three of the aforementioned color models: RGB, HSV (called HSB), and HSL. Passing color values to the `fill()` and `stroke()` functions is the main way to color shapes in P5.js. This sets the current fill and stroke colors for all subsequent shapes, and this setting is remembered until you `fill()` or `stroke()` again, or disable the stroke or fill entirely with the `noStroke()` or `noFill()` functions.

The default color model in P5.js is RGB, which means that the `fill()` and `stroke()` functions expect three numbers between 0 and 255, indicating the amount of red, green, and blue to use for the color. The reason behind this specific range is that a maximum of 256 values can be stored in a single byte (8 bits), allowing each RGB color to only take up 24 bits. Even though 256 different amounts of red, green, and blue might not sound like much, this can produce 16,777,216 distinct colors which is actually much more than what the human eye can perceive.

```
noStroke();  
fill(210, 70, 50);  
ellipse(150, height/2, 200, 200);  
  
fill(245, 225, 50);  
ellipse(300, height/2, 200, 200);  
  
fill(50, 120, 170);  
ellipse(450, height/2, 200, 200);
```

P5.js also allows you to use an alternative hexadecimal syntax known from web design for specifying colors in the RGB color model. Instead of using three numbers, the hex syntax uses a hashtag followed by a six-character string to represent the primary color values. Each primary color has two characters in this string, using the numbers 0-9 to represent zero to nine and the letters A-F to represent ten to fifteen. With 16 variations per character, each primary color can therefore specify a value between 0 and 255 in just two characters.

```
noStroke();  
fill("#d24632");  
ellipse(150, height/2, 200, 200);  
  
fill("#f5e132");  
ellipse(300, height/2, 200, 200);  
  
fill("#3278aa");  
ellipse(450, height/2, 200, 200);
```

The `colorMode()` function in P5.js can be used to switch to another color model, which means that the `fill()` and `stroke()` functions will expect color ranges according to the new color model's dimensions. The default

numerical ranges for HSV (called HSB in P5.js) and HSL are 0-360 for hue (indicating an angle), and 0-100 for saturation and brightness/lightness (indicating a percentage). The following code example uses both the HSV and HSL color models to draw the three ellipses.

```
noStroke();
colorMode(HSB);
fill(7, 76, 82);
ellipse(150, height/2, 200, 200);

fill(54, 80, 96);
ellipse(300, height/2, 200, 200);

colorMode(HSL);
fill(205, 55, 42);
ellipse(450, height/2, 200, 200);
```

It is also possible to change the default numerical ranges for each color model. This can be done by passing in three additional numbers when calling the `colorMode()` function, as demonstrated below where all three dimensions of the HSV color model are set to 0-1 ranges.

```
noStroke();
colorMode(HSB, 1, 1, 1);
fill(0.0195, 0.76, 0.82);
ellipse(150, height/2, 200, 200);

fill(0.15, 0.80, 0.96);
ellipse(300, height/2, 200, 200);

fill(0.569, 0.71, 0.67);
ellipse(450, height/2, 200, 200);
```

In the following chapters, we will examine a range of different techniques for combining colors programmatically in P5.js. Many of these examples will use the HSL color model, as it is an intuitive way to navigate the color spectrum.

Color

Perceptually uniform color spaces

“In visual perception a color is almost never seen as it really is - as it physically is. This fact makes color the most relative medium in art.”

Josef Albers (1963), *Interaction of Color*

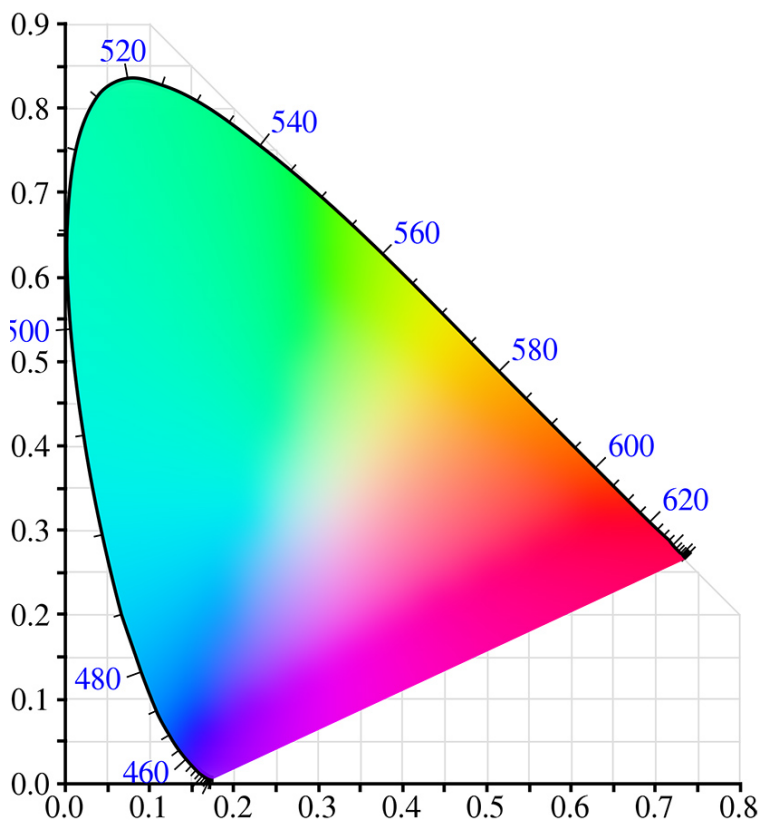
If you rounded up a group of graphic designers and asked them to define the concept of perceptually uniform color spaces, there is a good chance that none of them would know what to say. On the surface, perceptual uniformity is somewhat easy to explain: These color spaces are human-friendly alternatives to color spaces such as sRGB, and they are incredibly helpful for designers working in code. Despite of this, they can feel daunting to use in programmatic designs. Perceptually uniform color spaces have roots in scientific color theory, and this community does little to make them accessible to a larger audience. In this chapter, we will look at the concept of perceptually uniform color spaces, and answer some common questions related to them: What are they? Why do we need them? How can we use them in code?

What is wrong with sRGB?

Let us pretend that you want to design a poster with ten squares changing in color from green to blue in equal steps. “Easy”, you might say, and whip up some code that creates an equal change in hue between each

rectangle. Convinced that the result will be a nice looking gradient, you are surprised to see the following output after running the code.

You might notice something odd about this colored strip of rectangles. Although the colors change from green to blue, they appear to change a lot more towards the end of the strip. The green colors look almost identical, while the blue colors are more diverse. It also has a lot of variation in the lightness of the colors, with the cyan colors in the middle looking brighter than the blue colors. This happens because the default sRGB color space (and any color model built on it like HSV and HSL) is irregular, which means that even though the rectangles have evenly spaced hue values, the corresponding effect is not linear to the human eye.



The CIE chromaticity diagram showing wavelength of major colors in the color spectrum.

To explain why, we need to look at the chromaticity diagram we briefly discussed in the last chapter. This diagram is the result of extensive scientific experiments in the 1930's, and it plots the visible color spectrum onto a scale based on the human vision. The first thing you might notice is that the diagram has a lot of green in it. The blue numbers displayed on the edge

of the color spectrum show the wavelength of the corresponding color, and you will notice that the colors from around 520nm to 560nm all look green. But if you take another 40nm range, e.g. 460nm to 500nm, it includes a much broader set of colors between blue, cyan, and green. This explains why a majority of the rectangles in our design above are green, and why we see a sudden shift towards blue at the end of the scale: moving linearly through the hues will look disproportionate to the eye. If we want to operate with color as it relates to the human vision, we need a color space built on these human measurements, and that is what perceptually uniform color spaces are.

The following rectangles also have an even distribution in hues, but this time the colors were created with a perceptually uniform color space. Notice how the colors remain constant in their lightness, and that the hues are evenly distributed to make a linear color gradient.

Why do we need perceptually uniform color spaces? Because working with color in code is different than working with color in traditional design tools. Traditional tools encourage designers to think in manual workflows with the color picker as the primary way of choosing color combinations. In this scenario, designers use their eyes to decide whether a color is right or wrong, and the RGB values play no role in this decision. Code is different, because programming languages encourage designers to think about colors as numbers or positions within the chosen color model. This skill is hard to learn if the numbers do not correspond with the output. Perceptually uniform color spaces allow us to align numbers in our code with the visual effect perceived in our viewers.

In some cases, perceptually uniformity is essential. A simple example like wanting to choose a random color to be readable against a dark background can be hard in irregular color spaces, because colors with the same lightness or brightness vary greatly in how bright they appear (blue

and yellow both have 100% brightness in HSV, but blue is much darker than yellow). One would need to do all sorts of calculation based on the chosen hue to make the random colors equally bright.

If designers are not aware of this, it can even lead to misleading designs. A good example is the use of continuous color scales in data visualization. For certain map types, designers use a gradient to color geographic areas to reflect the value of a data point, and the user can compare colors between regions to get a sense of the data. If the designer created the color scale in a regular color space, the perceived colors will be different from the data points reflected in the color values. To have the design show the actual data, a perceptually uniform color space is required.

A better solution

The International Commission on Illumination (CIE) created the aforementioned chromaticity diagram in the 1930's to solve this problem. This diagram is actually a 2D view of a color space called CIE XYZ, which in the 1970's was replaced with the slightly improved CIE LUV and CIE LAB color spaces. It is hard to describe how these color spaces work without going into the underlying math, but they generally allow you to specify color, not in light mixes, but in dimensions that relate more to the human vision, and they do sophisticated color transformations to ensure that these dimensions reflect how the human vision works. For example, the CIE LUV color space has two dimensions – u and v – that represent color scales from red to green and yellow to blue. To create a color in the CIE LUV color space, one has to define the lightness of the color (l), whether it is reddish or greenish (u), and whether it is yellowish or bluish (v). Similarly, humans compute signals from our retina cones via the opponent process model, which makes it impossible to see reddish-green or yellowish-blue colors.

Even though these color spaces are based on human perception, they are

not intuitive when working in code. Like a RGB color space, it can be hard to guess which LUV numbers are required to create e.g. a dark purple or bright cyan. Thankfully, perceptually uniform color spaces can also remap their dimensions in different color models, so designers can work with more intuitive dimensions, while keeping the perceptual uniformity.

HSLuv

The [HSLuv project](#) is one of the more recent attempts at making these color spaces more intuitive. It allows you to use the CIELUV color space in the same dimensions as the HSL color model. Referred to as a human-friendly HSL, the original code was written in the Haxe programming language, but the project is now implemented in most of the popular programming languages, including JavaScript.

Before diving into the code-specifics, it is important to understand how HSLuv differs from HSL. HSLuv allows you to define a color based on three dimensions – hue, saturation, and lightness – but contrary to a HSL color model based on sRGB, colors that share the same value for a dimension are guaranteed to look similar. Two colors with an identical lightness value will look equally bright, and two colors with the same saturation will have the same perceived color purity. Like HSL, the saturation and lightness dimension is represented as a percentage between `0` and `100`, but in HSLuv those percentages reflect the perceived color mixing. A gray color with a lightness of `50` is guaranteed to be mid-gray.

Even though it is not a built-in color mode, HSLuv works great with P5.js. To use the library, you first need to download the [latest HSLuv release](#), and then include the library file in your HTML file. This makes the HSLuv color conversion functions accessible in your sketch.

```
<script src="p5.min.js" type="text/javascript"></script>
<script src="hsluv.min.js" type="text/javascript"></script>
```

Every implementation of HSLuv includes four functions that can be used to convert between HSLuv and RGB. We can use one of these functions – `hsluvToRgb()` – to convert the HSLuv color values into RGB values that the `fill()` and `stroke()` functions can understand. The `hsluvToRgb()` function expects an array with three values – the desired hue, saturation, and lightness of the color – and returns another array with RGB values in the range of 0 to 1. Because P5.js expects RGB values between 0 and 255, we need to multiply the array values to scale them up. This boils down to two lines of code, which is illustrated in the example below.

```
// First convert the HSLuv values to a RGB array
const rgb = hsluv.hsluvToRgb([0, 50, 50]);

// Then use the RGB values in a scale of 0-255
fill(rgb[0] * 255, rgb[1] * 255, rgb[2] * 255);
```

This means that every time you have to create a perceptually uniform color for the `fill()` or `stroke()` function, you need an extra line of code to handle the HSLuv to RGB conversion. This can be prevented by creating small helper functions that wrap these two lines of code.

```
function fillHsluv(h, s, l) {
  const rgb = hsluv.hsluvToRgb([h, s, l]);
  fill(rgb[0] * 255, rgb[1] * 255, rgb[2] * 255);
}

function strokeHsluv(h, s, l) {
  const rgb = hsluv.hsluvToRgb([h, s, l]);
  stroke(rgb[0] * 255, rgb[1] * 255, rgb[2] * 255);
}
```

```
}
```

You can now use these two functions instead of the built-in fill and stroke functions. It is essentially a way to make your own colorMode in P5.js without using the colorMode function. Below is a quick example to demonstrate how to use them.

```
noStroke();  
fillHsluv(0, 100, 50);  
ellipse(150, height/2, 200, 200);  
  
noFill();  
strokeWeight(5);  
strokeHsluv(120, 100, 50);  
ellipse(300, height/2, 200, 200);  
  
noStroke();  
fillHsluv(240, 100, 50);  
ellipse(450, height/2, 200, 200);
```

Now that we have the ability to use a perceptually uniform color space in P5.js, we can replicate the rectangle gradient experiment from the beginning of the chapter. The following example uses the same color values to draw a strip of rectangles using both HSL and HSLuv. Notice how the colors in the latter example look equally bright.

```
const w = width / 10;  
  
colorMode(HSL);  
for(let i = 0; i < numRects; i++) {  
  fill(i * 18, 100, 50);  
  rect(i * w, 0, w, height/2);  
}
```

```

}

colorMode(RGB);
translate(0, height/2);
for(let i = 0; i < numRects; i++) {
  fillHsluv(i * 18, 100, 50);
  rect(i * w, 0, w, height/2);
}

```

We can also use these new functions to choose random colors that are readable against a specific background color. The example below shows a line of text in random colors using both HSL and HSLuv. Notice how the first example sometimes use very bright yellows even though the lightness is constant. The latter example that uses the perceptually uniform color space does not have this problem.

```

const fontSize = 30;
textSize(fontSize);

translate(50, 50 + fontSize);
colorMode(HSL);
for(let i = 0; i < 10; i++) {
  fill(random(360), 100, 50);
  text("Can you read this line of text?", 0, i * fontSize);
}

colorMode(RGB);
translate(0, 340);
for(let i = 0; i < 10; i++) {
  fillHsluv(random(360), 100, 50);
  text("Can you read this line of text?", 0, i * fontSize);
}

```

Even though P5.js does not understand the concept of perceptually uniform color spaces, this chapter has demonstrated how to use the HSLuv JavaScript library to convert from a perceptually uniform color space into the sRGB color space that P5.js uses. In the coming chapters, we will use this technique to procedurally generate color schemes and use them to create dynamic designs in P5.js

Color

Color schemes

Effective use of color is vital to the success of any design project. Imagine a yellow Coca-Cola logo or the Mona Lisa painted in a saturated pink, and you might realize that the choice of colors has a drastic impact on any design. This does not mean that colors always have a clear semiotic purpose. While it might be obvious why stop signs are painted red (*alert, danger, there will be blood if you ignore this*), most color schemes are harder to interpret so directly. This leaves us with an art form that, like graphic design in general, is both objective and subjective. It is objectively a bad idea to use yellow text on a white background, because the lack of contrast makes the text hard to read. Likewise, one should not use red and green as primary colors in a data visualization when approximately 8% of men world-wide suffer from red-green color blindness¹⁶. It is important to know these rules when working with color, as they enable designers to create graphics that are accessible to a majority of users. However, to master the art of color combination, designers also have to know how color is used in different cultures and contexts, observe current trends in the arts, and develop a personal style based on this knowledge. Because of this, many authors have struggled to create good ways of teaching designers how to think about color combination.

One popular method is to create categories of color schemes named after the relationship between the hues of the colors in each scheme. These categories are given names such as *complementary* (two colors with opposite hues), *triadic* (three colors spaced evenly across the color spectrum), and *tetradic* (four colors spaced evenly across the color spectrum). Authors also include a more hybrid set of categories for color schemes that do not fit in any of the former categories. One explanation for the popularity of this approach might be that teachers can visualize the categories by placing shapes on the color circle, and students can change the dimensions of these shapes to create variations of the color schemes. This technique is described in many books about graphic design, and you will often encounter these terms in design critiques.

A triadic color scheme with three evenly spaced hues .

A tetradic color scheme with four evenly spaced hues.

Even though this systematic approach might seem like a perfect fit for this book, I believe the method to be highly problematic. One problem is that color schemes within a single category do not have any coherent visual effect. Triadic and tetradic color schemes can look remarkably similar, even identical, if the spacing between hues is tweaked slightly. Also, the visual effect of a pure triadic color scheme in the sRGB color space is very different from the same color scheme in CIELUV. Worst of all, this approach tends to ignore the saturation and lightness dimensions, which are left for designers to figure out for themselves. The three color schemes below all have the same hue values, but produce wildly different visual effects by changing their saturation and lightness values. A color theory that ignores two-thirds of the color dimensions will not help designers make better decisions.

These three triadic colors schemes have the same hue values, and look identical when plotted on the color circle. The term 'triadic' does not help to describe their distinct visual

effects.

Instead, this chapter will present a color theory built around the three dimensions of the HSL color model. By focusing on the hue, saturation, and lightness of colors – and how these dimensions interact – designers can learn how changes in code are reflected visually, and compose interesting color combinations from this knowledge. In the following, we will go through these dimensions in reverse order, using the [HSLuv library](#) to ensure that changes in our code reflect actual perceptual changes in color.

Lightness

The lightness of a color determines how much black is mixed into the color (See [Color Models and Color Spaces](#)). The contrast of light and dark is a significant one, and even though we can create contrasts between colors by manipulating any of the HSL dimensions, the term ‘contrast’ most times refers to changes in lightness. The examples below demonstrate the effect of both a low-contrast and a high-contrast color scheme. The first example appears soft and light, which is a result of the high lightness values with small differences in lightness between each color. The second example appears bolder and has a significant positive/negative effect caused by the large differences in lightness for neighbouring colors.

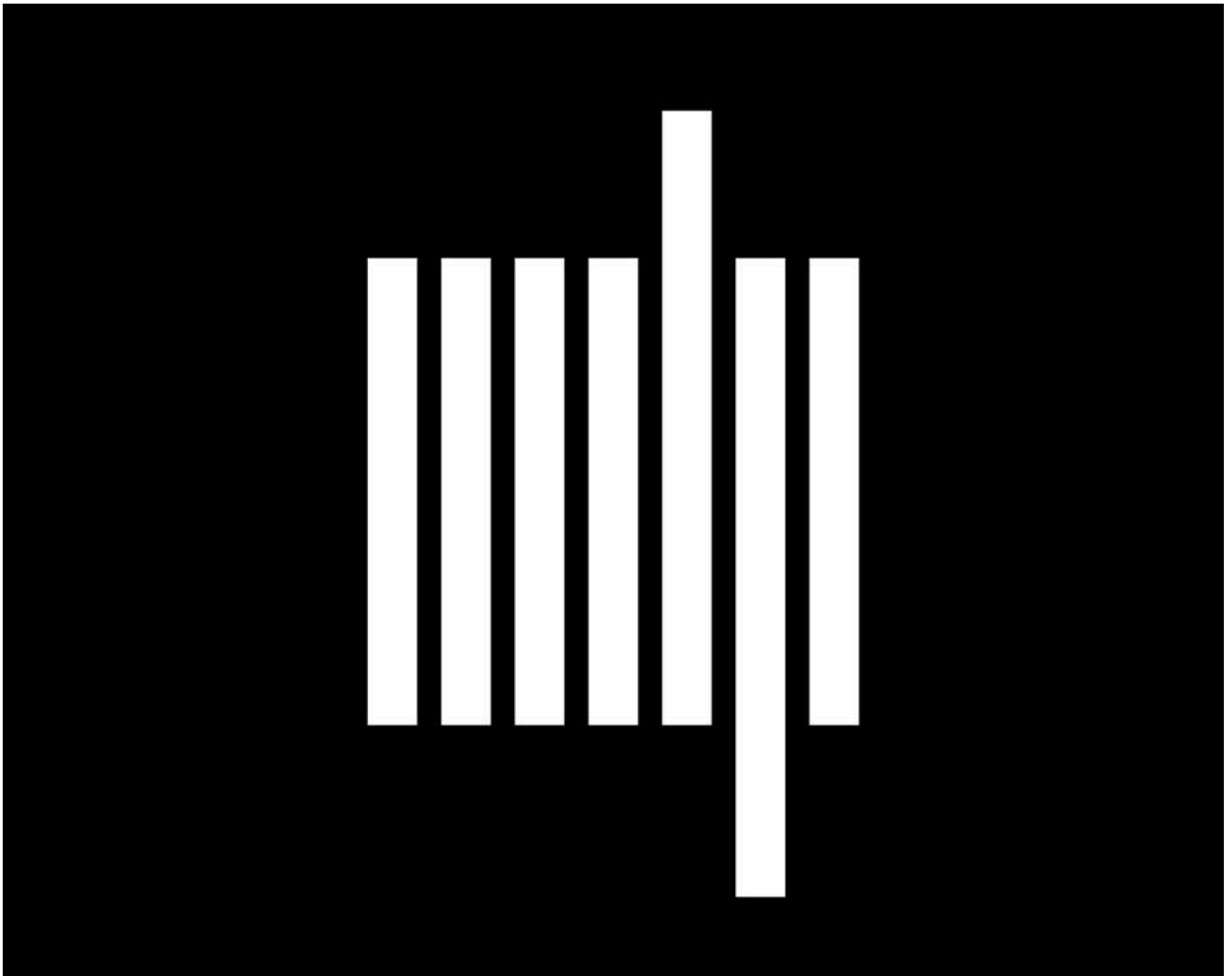
A color scheme with low contrast

```
fillHsluv(0, 0, 90);  
rect(0, 0, width, height);  
fillHsluv(0, 0, 85);  
rect(145, 95, 375, 200);  
fillHsluv(0, 0, 95);  
rect(85, 155, 375, 200);
```

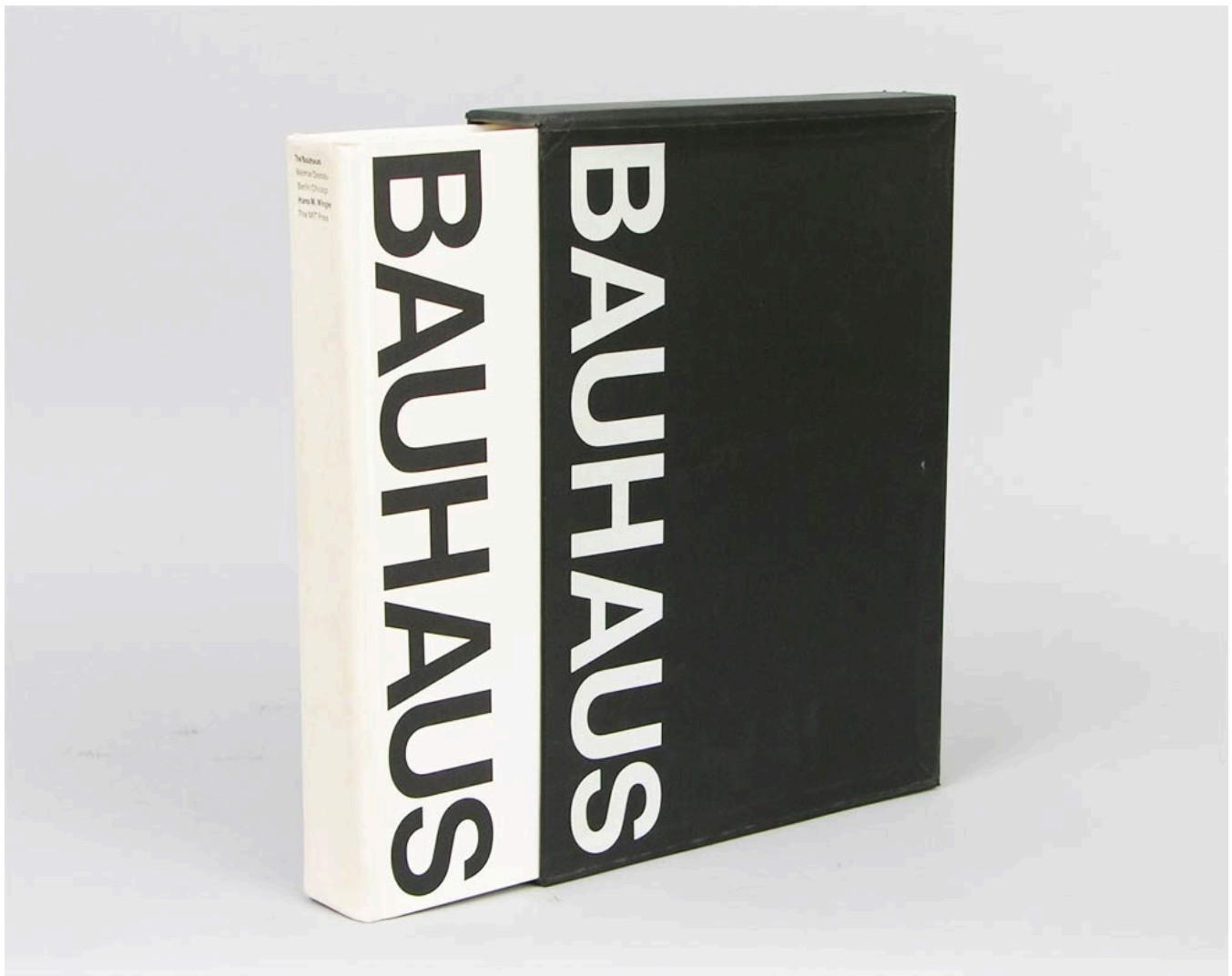

A color scheme with high contrast

```
fillHsluv(0, 0, 0);  
rect(0, 0, width, height);  
fillHsluv(0, 0, 50);  
rect(145, 95, 375, 200);  
fillHsluv(0, 0, 100);  
rect(85, 155, 375, 200);
```

Muriel Cooper was a highly influential graphic designer who did important work in early digital design and user interface design. As co-founder of the MIT Media Lab and Design Director of MIT Press, she oversaw production of more than 500 books, and she is widely known for her black and white designs. Below are two of her more famous designs that rely solely on the lightness dimension.



The logo for MIT Press © .



Cover design for a book on the Bauhaus published by MIT Press © .

Choosing a proper contrast is especially important when working with text, as readability is determined by the contrast between text and background. The World Wide Web Consortium recommends in their [Web Content Accessibility Guidelines](#) that body text should have a contrast ratio of at least 4.5:1 (or lower than 0.222 as a fraction), and they provide the following formula to calculate this contrast ratio for two lightness values.

```
const contrastRatio = (l1 + 0.05) / (l2 + 0.05);
```

This formula requires the two lightness values to be provided in relative

luminance, which refers to the Y dimension in the CIEXYZ color space. To calculate the W3C contrast ratio for a HSLuv color, we therefore first need to convert the lightness value into the CIEXYZ color space, and then use the formula above to calculate the contrast ratio. This is demonstrated in the code example below, where the `contrastRatio()` function can calculate this contrast ratio based on two HSLuv lightness values.

```
function lightnessToLuminance(l) {
  if (l <= 8) {
    return 1.0 * l / 903.2962962;
  } else {
    return 1.0 * Math.pow((l + 16) / 116, 3);
  }
}

function contrastRatio(l1, l2) {
  l1 = lightnessToLuminance(l1);
  l2 = lightnessToLuminance(l2);
  return (l1 + 0.05) / (l2 + 0.05);
}

function setup() {
  console.log(contrastRatio(40, 70)); // BAD!
0.35521707859730733
  console.log(contrastRatio(40, 90)); // GOOD!
0.19988073069469958
}
```

Lightness plays an important role in any color scheme, both when it comes to accessibility and aesthetics. Like the previous exercises in this book, it is recommended that aspiring designers practice designing only in black and white to learn how to establish proper contrast in their designs.

'What does it look like in black and white' is a good question to ask if a design seems cluttered, as it can reveal a lack of contrast between shapes in the design.

Saturation

The saturation of a color controls the purity of the color from grayscale to full color (See [Color Models and Color Spaces](#)). You can use this dimension to create color combinations that range from the very muted to the extremely bright. The two examples below use the same lightness and hue values, and differ only in their saturation values.

A desaturated color scheme

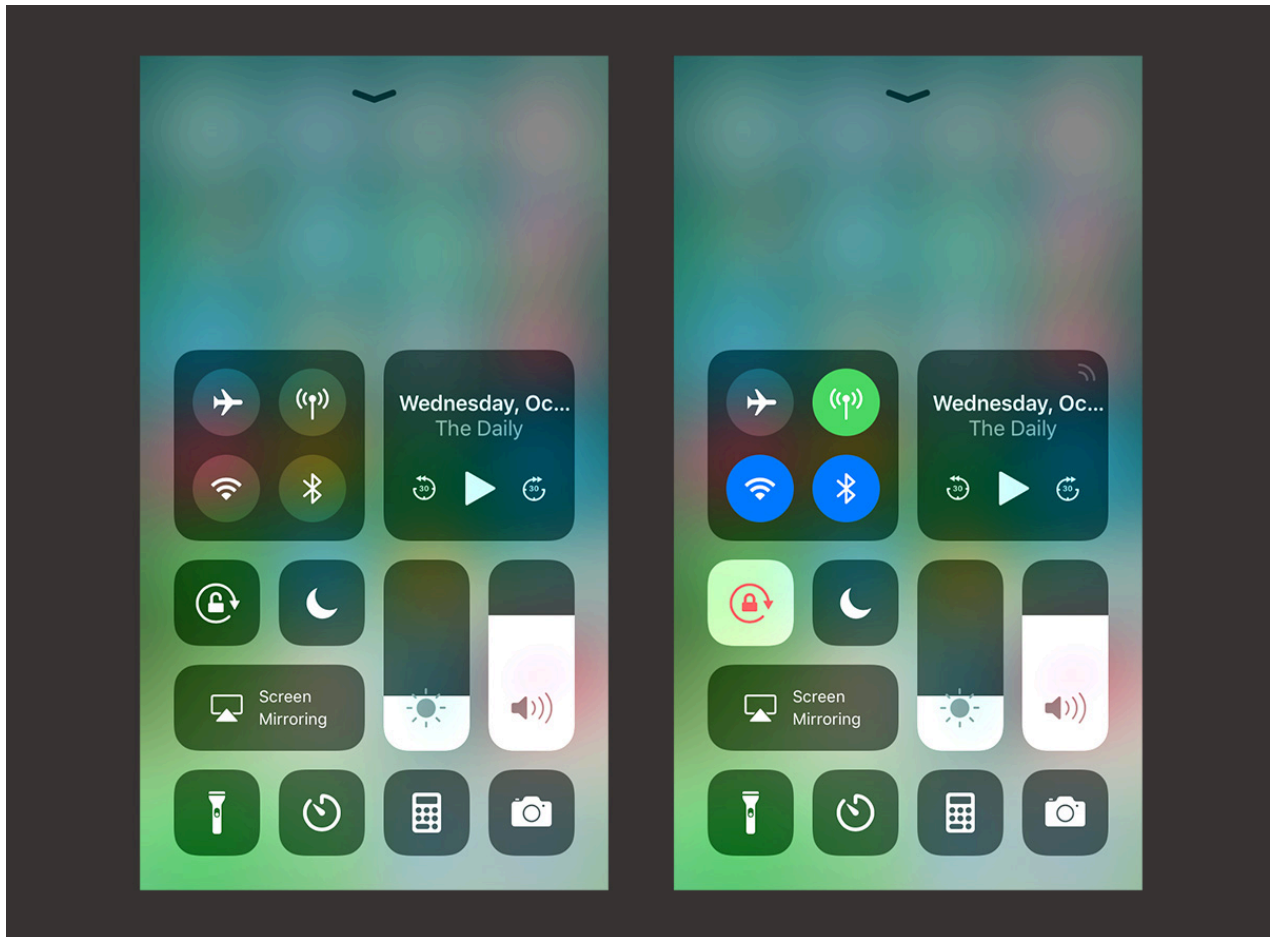
```
fillHsluv(40, 30, 65);  
rect(0, 0, width, height);  
fillHsluv(10, 40, 40);  
rect(145, 95, 375, 200);  
fillHsluv(75, 50, 85);  
rect(85, 155, 375, 200);
```

A saturated color scheme

```
fillHsluv(40, 100, 65);  
rect(0, 0, width, height);  
fillHsluv(10, 100, 40);  
rect(145, 95, 375, 200);  
fillHsluv(75, 100, 85);  
rect(85, 155, 375, 200);
```

In user interface design, saturation is often used to distinguish passive and active interface components. Apple's iOS operating system utilizes a desaturated color scheme for general interface elements, but fully

saturated colors are used for key actions such as active toggle buttons and app notifications. This makes it possible for users to quickly interpret the state of the interface and notice when a new app event happened, like an LED lighting up on an old switchboard.



The control center in iOS 11 uses a monochrome color scheme but fully saturated colors for active buttons. © .

A good analogy to consider when working with saturations of color is the way many countries have distinct ways of painting houses in their cities. The saturations of these paints can vary greatly, and although these colors do not have any inherent meaning, they do say something about the time, place, and people. Imagine a small village in Japan with its muted, desaturated colors, and compare this to a place like Mexico, where houses are painted in very pure, saturated colors. These colors reflect the culture

around them, and you should consider your content in the same way: Does it demand lively colors or a subdued, modernist scheme? The saturation of your colors is the key to this.



A Japanese town in desaturated colors. Image by [663h](#) .



Houses in Mexico painted in saturated colors.

Hue

The hue dimension determines which actual color to show as represented by color names such as red, green, and blue (See [Color Models and Color Spaces](#)). As mentioned in a previous chapter, there is no coherent theory on which hue combinations produce harmonic results. Although many have tried, it is impossible to make a generalized theory about such a thing.¹⁷ However, it can be beneficial to draw some conclusions from how hue combinations exist in the real world.

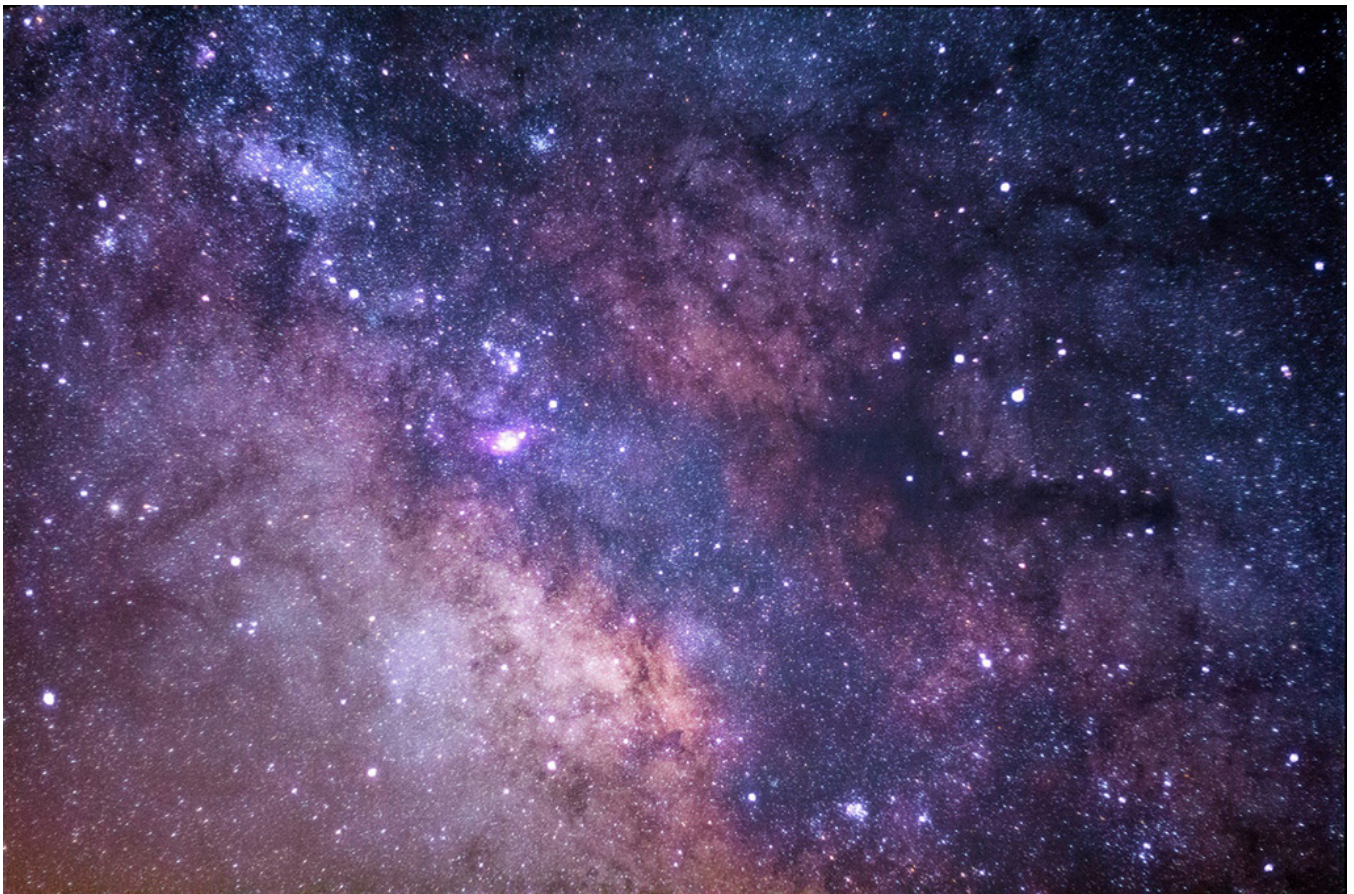
In nature, we often see small hue differences in the red-green parts of the color spectrum. In the spring, trees and plants will take on an almost monochrome bright green color scheme, but as the season turns to fall, these colors will spread slightly into a multitude of analogous colors of

green, yellow, and red. You will often find such hue combinations in design products looking to evoke feelings of tranquility or peace, such as yoga studios, organic food products, and wedding invitations. A different type of analogous color scheme can be found in photos from outer space, where brighter colors in the blue/green parts of the spectrum create an almost alien effect. These colors are often used in technology or software products that want to appear streamlined and deliberately manufactured. It is not a coincidence that Apple and Microsoft stores look like the inside of a spaceship: They are designed to make customers feel like they entered a state-of-the-art science lab, because it makes them accept the higher price point. These are two types of color schemes with colors close to each other on the spectrum, but with very different visual effects.

Analogous colors in nature.



Analogous colors in space.



```
fillHsluv(75, 95, 70);  
rect(0, 0, width, height);  
fillHsluv(35, 90, 40);  
rect(145, 95, 375, 200);  
fillHsluv(55, 100, 80);  
rect(85, 155, 375, 200);
```

```
fillHsluv(270, 65, 15);  
rect(0, 0, width, height);  
fillHsluv(295, 70, 55);  
rect(145, 95, 375, 200);  
fillHsluv(278, 100, 73);  
rect(85, 155, 375, 200);
```

A more profound effect happens when the distance between hues is

increased. However, this statement can be deceiving. Although colors close to each other have a nice, analogous effect, colors on opposite sides of the color circle do not necessarily have the most pronounced hue contrast. The visual effect of two hues cannot be calculated by a simple formula, as much is determined by the actual hues chosen, how they are used in a design, as well as their saturation and lightness values. Nevertheless, it is important to consider the spacing of hues like any other relationship in the design process: Does my content demand flat, monochrome colors or a varied burst of the color palette?

.

Low hue contrast.

Medium hue contrast.

High hue contrast.

We cannot discuss the hue dimension without mentioning the various types of color blindness that makes it hard for many to distinguish certain hues. The most prevalent color blindness is red-green, which makes it hard to distinguish red and green hues from each other. If the green retina cone is severely damaged, some people can only see the color blue. A more infrequent type of color blindness is blue-yellow, where blue, yellow, and green is indistinguishable. Also, in the rarest of cases, some people have complete monochrome vision. Up to 8% of men and 0.5% of women worldwide suffer from color blindness ¹⁸. This means that a color scheme, especially one needed for instructional use, should not rely solely on hues to create color contrasts. Lightness should always be considered, as the ability to distinguish colors by contrast is shared by almost anyone who are not vision-impaired.



Normal color vision.



Mild red-green blindness.



Severe red-green blindness.

Color scheme examples

These three dimensions of color can inspire a lifetime of experiments with color combination. While some color schemes consist of colors that only vary in one dimension (such as some monochrome designs), most color schemes combine changes in hue, saturation, and lightness to achieve a palette of colors. The Brooklyn-based chocolate producer Mast Brothers is

famous for their colorful packaging designs where colored patterns are used to denote the flavor profile of the chocolate. These patterns provide a great case study of how the three dimensions of color can be manipulated to create different expressions.

The image shows the packaging for MAST SMOKE CHOCOLATE. The background is a textured, light brown or terracotta color. Overlaid on this are several horizontal, wavy black lines that resemble smoke or liquid splatters. A white rectangular label is centered on the upper half of the package.

MAST

SMOKE

CHOCOLATE

70 G

Smoke © .

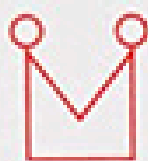
MAST

MINT

CHOCOLATE

70 G

Mint © .



MAST BROTHERS

BROOKLYN BLEND
CHOCOLATE

2.5 OZ / 70 G

The first pattern is for a dark chocolate made with smoked beans, and the designer has chosen a high-contrast, monochrome color scheme with wavelike shapes to imply smoke floating in the air. Notice how these colors do not seem ‘natural’ per se, but are chosen to convey the taste of the product. The second pattern is for a dark chocolate with mint leaves and features a gradient of colors changing in hue and lightness from dark green to bright yellow with a constant saturation in the 50’s. The gradient provides for an interesting way of visualizing the two distinct flavors that despite their different characteristics blend well together. The final pattern features a four-color scheme on top of a lighter background where saturated colors with large hue contrasts are used to color paint-like shapes. A playful and creative design for the borough of Brooklyn.

Procedural color schemes

So far we have manually hard-coded color values to create color schemes. To really take advantage of the fact that we are using code to generate these designs, we should investigate how to procedurally generate these colors. That is, use a loop to create a lot of colors in just a few lines of code. This means looking at the `color()` function and how to dynamically create color objects with a loop.

The `color()` function in P5.js can be used to create a reusable color object, that can be used in the `fill()` and `stroke()` functions again and again. This means that, rather than having the same color values scattered throughout the code, we can assign a single color object to a variable on top of our code, and refer to this variable whenever the color needs to be used. Consider the following code where the same red color is used multiple times.

```
// First use of red
```

```
fill(225, 35, 35);
rect(50, 50, 200, 180);

fill(40, 185, 155);
rect(200, 100, 200, 180);

// Second use of red
fill(225, 35, 35);
rect(350, 150, 200, 180);
```

This example can be rewritten using the `color()` function, so the color values appear only once in the code.

```
// Define the color object once
const red = color(225, 35, 35);
// Use it here
fill(red);
rect(50, 50, 200, 180);

fill(40, 185, 155);
rect(200, 100, 200, 180);

// Use it here
fill(red);
rect(350, 150, 200, 180);
```

To use the `color()` function with HSLuv values, we need to create a small function that performs the HSLuv to RGB conversion before creating the color object. Besides the use of the `color()` function, this function is identical to the `fillHsluv()` and `strokeHsluv()` functions from the last chapter. Remember that you must include the HSLuv JavaScript file for this to work.

```
function colorHsluv(h, s, l) {
```

```
const rgb = hsluv.hsluvToRgb([h, s, l]);
return color(rgb[0] * 255, rgb[1] * 255, rgb[2] * 255);
}

const red = colorHsluv(225, 35, 35);
fill(red);
```

Now, what if we want to use multiple colors in a design? It would make sense to just add more variables to the code above. Although that is perfectly fine for just a few colors, it does not make sense for a large number of colors. In this scenario, it is more sensible to use an array that allows for colors to be added and removed without introducing new variables. The code below stores three colors in an array and uses them to draw a color scheme.

```
const colors = [
  colorHsluv(40, 100, 65),
  colorHsluv(10, 100, 40),
  colorHsluv(75, 100, 85)
];

fill(colors[0]);
rect(0, 0, width, height);
fill(colors[1]);
rect(145, 95, 375, 200);
fill(colors[2]);
rect(85, 155, 375, 200);
```

Finally, we can use a loop to dynamically create colors objects. We do this by using an empty array, and pushing a new color object to the array on every loop iteration. This examples uses the `random()` function to ensure that the colors are different between each run of the loop.

```
// Start with empty array
const colors = [];
for(let i = 0; i < 3; i++) {
  // Push new color with random hue, saturation, and lightness into
  // array every time
  colors.push(
    colorHsluv(
      random(360),
      random(100),
      random(100)
    )
  )
}

fill(colors[0]);
rect(0, 0, width, height);
fill(colors[1]);
rect(145, 95, 375, 200);
fill(colors[2]);
rect(85, 155, 375, 200);
```

With all these concepts in place, we are ready to procedurally generate color schemes. One strategy – which is probably also the simplest – is to stick with `random()`, and create different types of color schemes by changing the values passed to the `random()` function.

Random lightness.

```
const colors = [];
for(let i = 0; i < 3; i++) {
  colors.push(
    colorHsluv(240, 20, random(100))
  )
}
```



```
fill(colors[0]);
rect(0, 0, width, height);
fill(colors[1]);
rect(145, 95, 375, 200);
fill(colors[2]);
rect(85, 155, 375, 200);
```

Random hue.

```
const colors = [];
for(let i = 0; i < 3; i++) {
  colors.push(
    colorHsluv(random(360), 90, 50)
  )
}

fill(colors[0]);
rect(0, 0, width, height);
fill(colors[1]);
rect(145, 95, 375, 200);
fill(colors[2]);
rect(85, 155, 375, 200);
```

However, there is a good chance that the `random()` function will choose numbers close to each other, resulting in very similar colors. A more powerful strategy is to use the loop's incrementing `i` variable to calculate the values passed to the `color()` function. This technique is identical to what was demonstrated in the [Procedural Shapes](#) chapter, but this time we use it for color values and not `x` and `y` positions. The example below uses `i` to create a monochrome color scheme with lightness values of `20`, `50`, and `80`, while keeping the hue and saturation constant.

```
const colors = [];  
for(let i = 0; i < 3; i++) {  
  colors.push(  
    colorHsluv(240, 100, 20 + (i * 30))  
  )  
}  
  
fill(colors[0]);  
rect(0, 0, width, height);  
fill(colors[1]);  
rect(145, 95, 375, 200);  
fill(colors[2]);
```

Below, the same technique is used in all dimensions. Notice how the hue values are incrementing while the saturation and lightness values are decrementing. The final result is a color scheme where the background is light and saturated but the front colors are darker and less saturated.

```
const colors = [];  
for(let i = 0; i < 3; i++) {  
  colors.push(  
    colorHsluv(  
      100 + (i * 80),  
      100 - (i * 20),  
      90 - (i * 30)  
    )  
  )  
}  
  
fill(colors[0]);  
rect(0, 0, width, height);  
fill(colors[1]);  
rect(145, 95, 375, 200);
```

```
fill(colors[2]);  
rect(85, 155, 375, 200);
```

This code can become even more exciting by adding a few variables to store the initial color values and how much they should change between each iteration of the loop. Rather than hardcoding these variables, we can use the `random()` function to pick different values every time the code runs. Below is the same code run three times to produce three different color schemes from the same algorithm. By changing the values passed to the `random()` function, this code can produce a multitude of different outputs.

```
// Which color values should we start with?  
const startHue = random(0, 360);  
const startSat = random(40, 100);  
const startLig = random(0, 60);  
// How much should each color change?  
const changeHue = random(10, 120);  
const changeSat = random(15, 40);  
const changeLig = random(15, 40)  
const colors = [];  
for(let i = 0; i < 3; i++) {  
  colors.push(  
    colorHsluv(  
// Use these values in the same algorithm as before  
      startHue + (i * changeHue),  
      startSat + (i * changeSat),  
      startLig + (i * changeLig)  
    )  
  )  
}  
  
fill(colors[0]);
```

```
rect(0, 0, width, height);  
fill(colors[1]);  
rect(145, 95, 375, 200);  
fill(colors[2]);  
rect(85, 155, 375, 200);
```

We cannot end this chapter without discussing another useful technique for procedural color generation – the `lerp()` function – which can be used to calculate transitions from one color to another. The `lerp()` function has nothing to do with color, as it can be used to calculate any number between two numbers. The function expects – besides the two range numbers – an interpolation amount that is used to calculate the resulting number. An interpolation amount of `0` will return the first number, `0.5` will return the number midway between the two numbers, and `1` will return the second number.

```
lerp(0, 100, 0.2) // => 20  
lerp(0, 50, 0.5) // => 25  
lerp(0, 360, 0.8) // => 288
```

As a digital color consists of three numerical values, we can use this function three times to calculate any color between two colors. It is important to note that P5.js has a `colorLerp()` function that performs this calculation in just one line of code. However, it only works with built-in color modes and not the HSLuv library. The example below finds the color midway between a dark saturated green and a lighter desaturated blue.

```
// Find hue between 120 and 240  
const h = lerp(120, 240, 0.5);  
// Find saturation between 95 and 40  
const s = lerp(96, 40, 0.5);  
// Find lightness between 31 and 74  
const l = lerp(31, 74, 0.5);
```

```
const midwayColor = colorHsluv(h, s, l);
```

This technique is even more powerful when used in combination with a loop, where the interpolation amount can be calculated by dividing `i` by its largest possible value. Because this calculation is performed over and over with an incrementing `i` value, it will produce interpolation amounts between `0` and `1` with the number of steps being equal to how many times the loop runs. This method is very useful when drawing gradients that change from one color to another, like the example below where a color swatch from Google's [Material Design](#) document is recreated.

50	#FFF3E0
100	#FFE0B2
200	#FFCC80
300	#FFB74D
400	#FFA726
500	#FF9800
600	#FB8C00
700	#F57C00
800	#EF6C00
900	#E65100

```
const boxh = height / 10;
for(let i = 0; i < 10; i++) {
  const h = lerp(64, 22, i / 9);
  const s = lerp(86, 90, i / 9);
  const l = lerp(96, 56, i / 9);
  fillHsluv(h, s, l);
  rect(0, i * boxh, width, boxh);
}
```

A color swatch from Material Design  is recreated with the `lerp()` function in a loop.

This chapter introduced techniques to help designers explore the color spectrum through the hue, saturation, and lightness dimensions of the HSL color model. Using these techniques, designers can move away from the 2D color solid known from the color picker, and approach color combination by focusing on the relationship between colors in a 3D space. Whether these techniques are used to quickly test different color combinations, or built directly into digital design products, they are another important tool for a designer wanting to treat design as a systematic art.

EXERCISE

Design a simple book cover for one of your favorite books. The design should use basic or custom shapes, but no typography. Once you have a design that conveys something in the storyline, consider which type of color scheme is needed to support your design. Keep in mind that a science fiction thriller might need very different colors than a romance novel. Then, color the shapes in your design using the techniques presented in this chapter. Rather than hard-coding the colors, try to make a design where the color scheme is different every

time the sketch runs. The challenge is to make a dynamic color scheme with a consistent visual style.

1. Loeb Classical Library (1936) *Aristotle's Minor Works*, p. 7. London
2. Gottschalk, H. B. (1964) *The De Coloribus and Its Author*, p. 59-85. *Hermes* 92. Bd..H. 1: JSTOR. Web. 11 Jan. 2017
3. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 25. University of Chicago Press
4. Sloane, Patricia (1967) *Colour: Basic Principles New Directions*, p. 28-30. Studio Vista
5. Lowengard, Sarah (2006) *The Creation of Color in Eighteenth-Century Europe* New York, para. 129-139, Columbia University Press)
6. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 48. University of Chicago Press
7. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 175-176. University of Chicago Press
8. Munsell, A.H (1912) *A Pigment Color System and Notation*, pp. 239. *The American Journal of Psychology*. Vol. 23. University of Illinois Press
9. Itten, Johannes (1973) *The Art of Color: the subjective experience and objective rationale of color*. Van Nostrand Reinhold
10. Albers, Josef (1963) *Interaction of Color*. Yale University
11. Droste, Magdalena (2002) *Bauhaus*, p. 25. Taschen
12. Itten, Johannes (1970) *The Elements of Color*, p. 33-44. Van Nostrand Reinhold
13. Itten, Johannes (1970) *The Elements of Color*, p. 26. Van Nostrand Reinhold
14. MacEvoy, Bruce. *Color Vision Handprint : Colormaking Attributes*. N.p., 1 Aug. 2015. Web. 11 Jan. 2017.
15. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. *Color Research and Application*. Volume 35, Issue 4
16. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
17. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. *Color Research and Application*. Volume 35, Issue 4
18. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
19. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*, p. 9. Arthur Niggli
20. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 2. *The College Mathematics Journal*, Vol. 23
21. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 8-9. *The College Mathematics Journal*, Vol. 23
22. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 10-11. *The College Mathematics Journal*, Vol. 23
23. Malik, Peter (2017) *P. Beatty III (P47): The Codex, Its Scribe, and Its Text*, p. 31 - 38. New

Testament tools, studies and documents, Vol. 52

24. Fry, Stephen (2008) *The Machine That Made Us*. BBC UK
25. Apollonio, Umbro (2009) *Futurist Manifestos*, p. 104. Tate Publishing
26. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*. Arthur Niggli
27. Gerstner, Karl (1964) *Designing Programmes*. Arthur Niggli
28. <https://www.nytco.com/who-we-are/culture/our-history/#2000-1971-timeline>
29. Peter, Ian (2004) *So, who really did invent the Internet?*. The Internet History Project
30. https://web.archive.org/web/20050106024725/http://www.subtraction.com:80/archives/2004/1231_grid_computi.php
31. Arnheim, Rudolf (1974) *Art and Visual Perception*, p. 8. University of California Press
32. Heider, G.M. (1977) *More about Hull and Koffka*, *American Psychologist*, 32(5), 383
33. Kroegeer, Michael (2008) *Conversations With Students*, p. 27. Princeton Architectural Press
34. Design Systems International was co-founded by the author

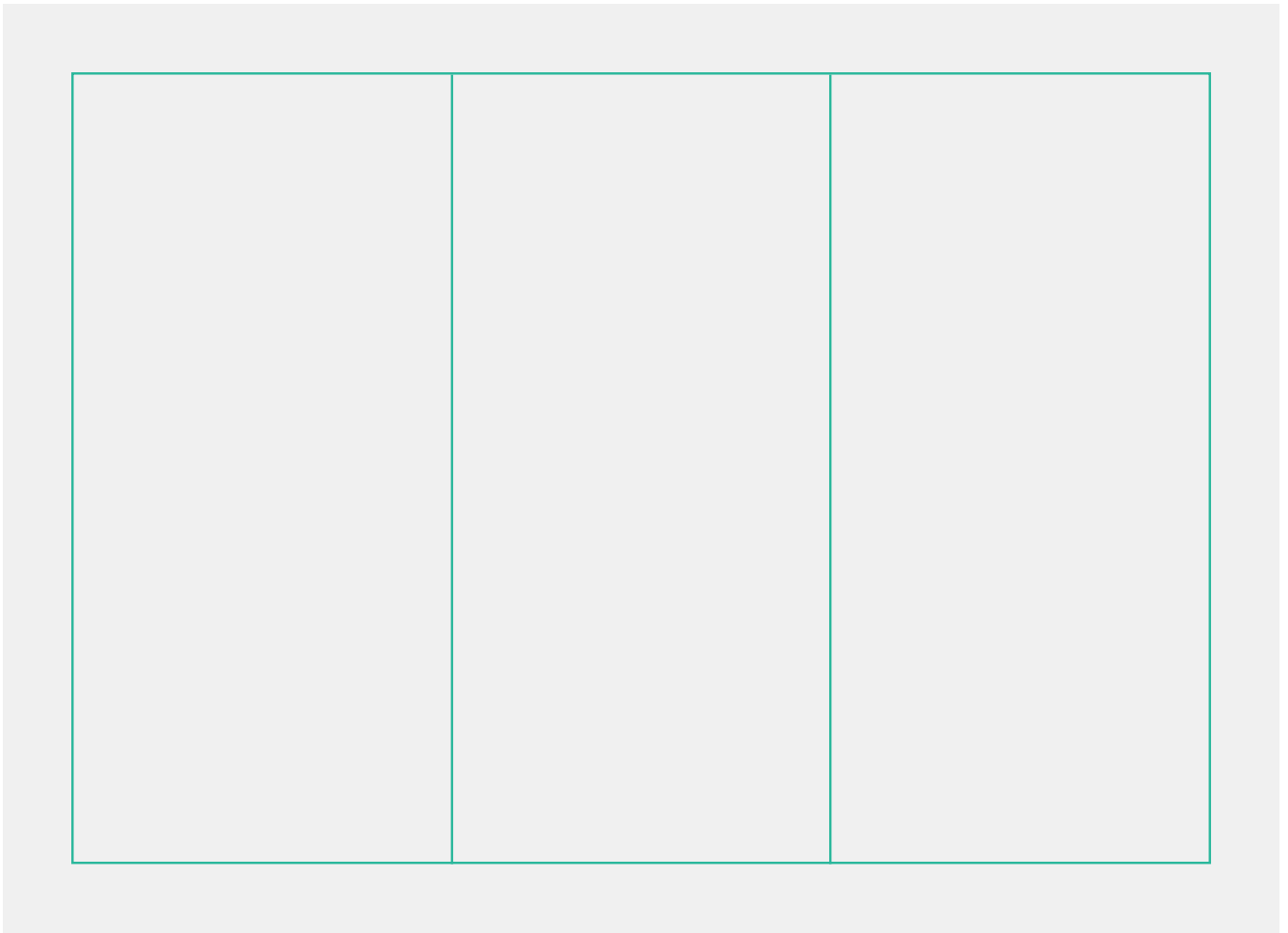
Layout

A short history of geometric composition

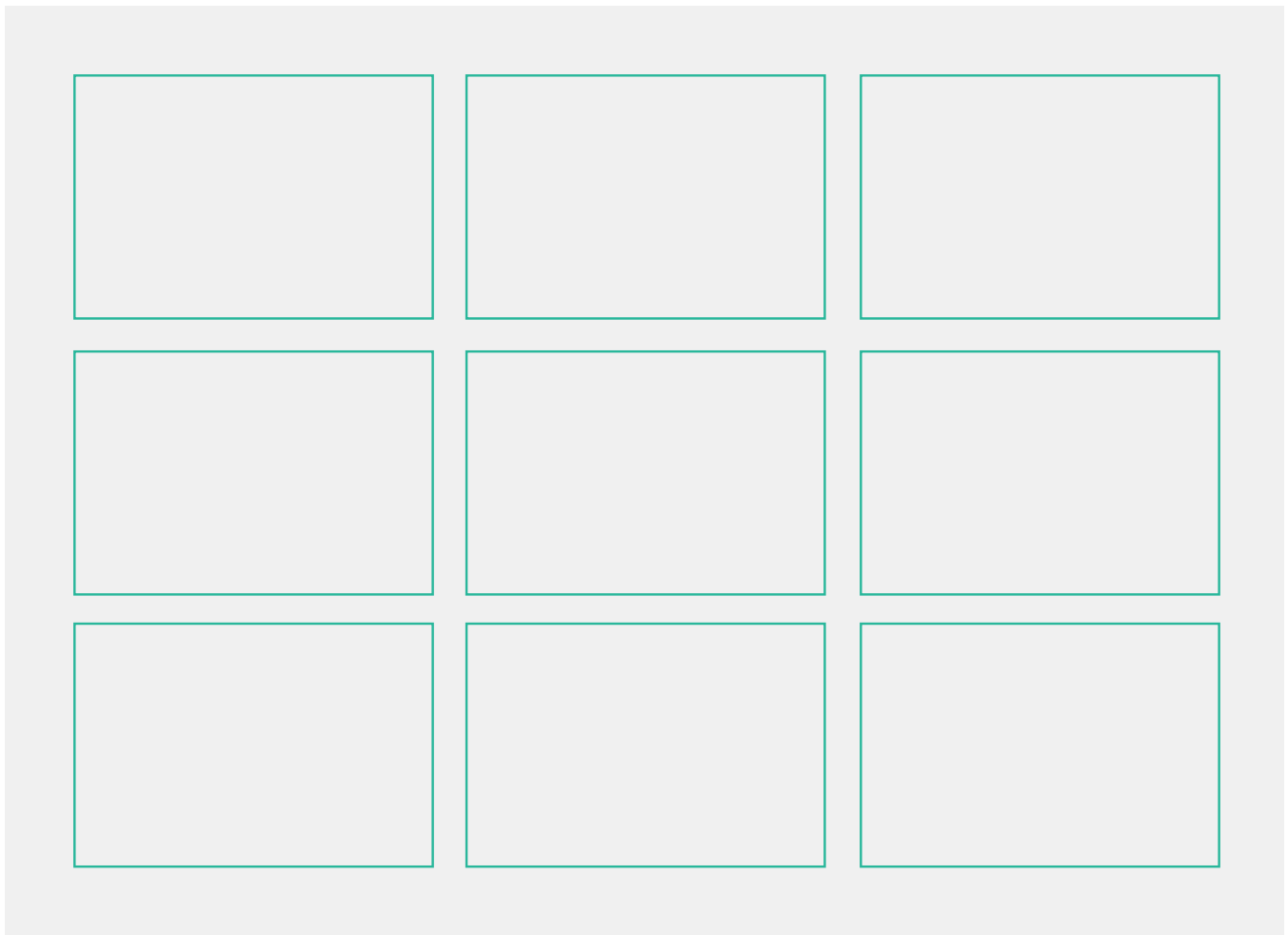
“Anyone willing to take the necessary trouble will find that, with the aid of the grid system, he is better fitted to find a solution to his design problems which is functional, logical and also more aesthetically pleasing”

Joseph Müller-Brockmann¹⁹

This part of the book focuses on principles of geometric composition and how they can help designers create organized and beautiful layouts. The term geometric composition covers a range of techniques used to guide the positioning of elements in a design. This can be as simple as a ratio – such as the rule of thirds ($\frac{1}{3}$) where the page is split into three equally sized rectangles – or as sophisticated as a full grid system with multiple columns and rows that control the alignment of text and other shapes.



A canvas divided into thirds.



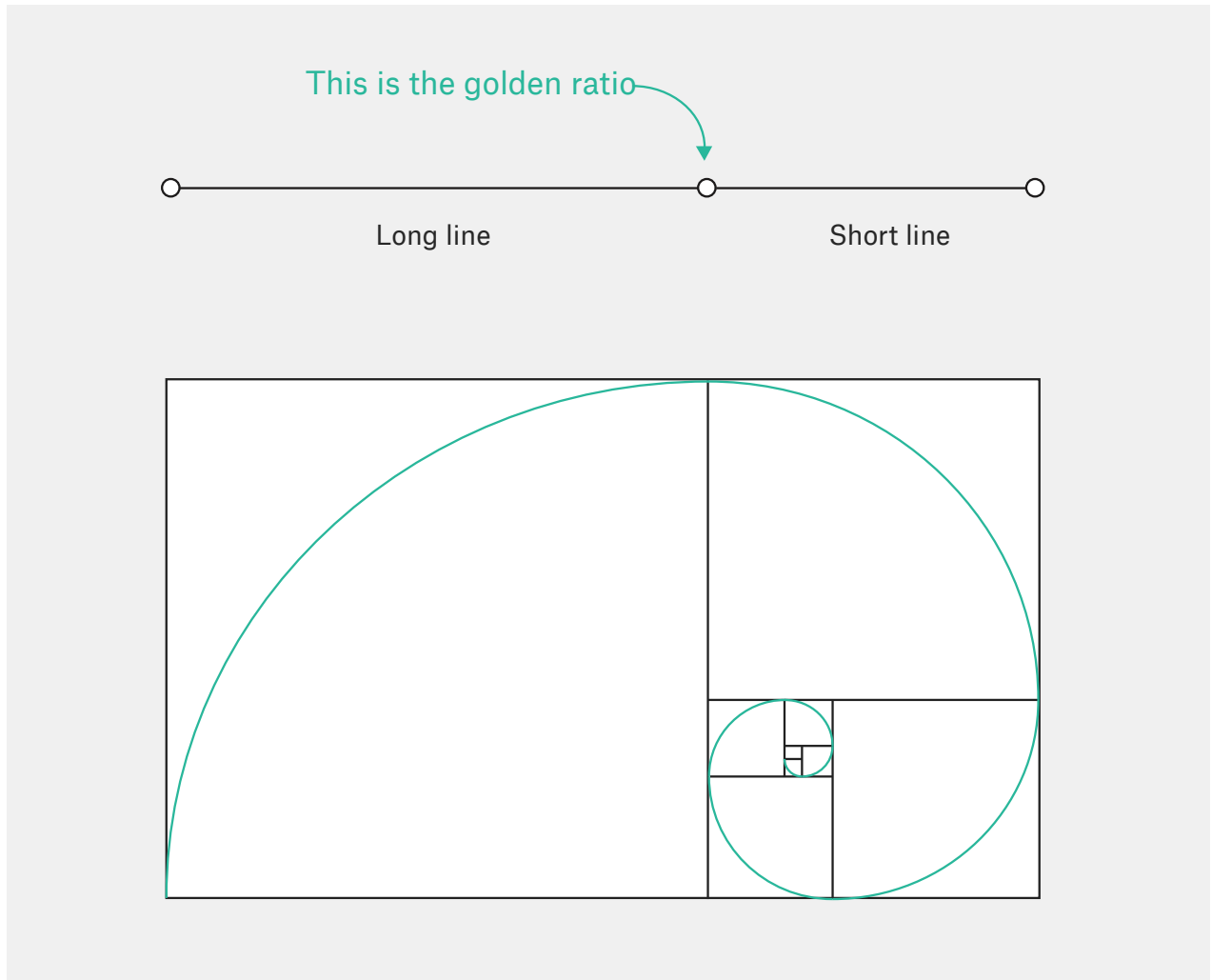
A canvas divided into a grid.

Before diving into the specifics of how to use these concepts when designing with code, we will first spend the rest of this chapter looking at examples of geometric composition from art history. This history will pay special attention to how new technology – first the printing press and then the computer – played a significant role in developing these ideas.

A brief critical note on the subject – especially the idea of the golden ratio – is necessary here. The golden ratio is best understood by imagining two lines, one long and one short. When the ratio between the long and the short line is the same as the ratio between the two lines combined and the long line, this is the golden ratio. That number happens to be

1.61803398875 , and it can be used in all sorts of ways. For example, if you

create a rectangle where the width is 1.61803398875 times larger than the height, it is called a golden rectangle. If you divide this rectangle into smaller rectangles using the same golden ratio and draw a curve through the corners of these smaller rectangles, you end up with a golden spiral. The golden spiral is the most popular visualization of the golden ratio, but do not get fooled by this complexity: The golden ratio is still just a number.



The golden ratio visualized as both a simple line segment and as a golden spiral.

Many of us remember hearing about the golden ratio in elementary school or early art education, and the takeaway was likely something like this: The golden ratio, or the divine proportion, is a number that is universally agreed upon to be aesthetically pleasing when used to position elements

of a design within the canvas. Because it is especially beautiful to the human eye, it has been used throughout history by artists to create masterpieces. The concept of the golden ratio has been used to describe everything from the architecture of the Parthenon in ancient Greece to the paintings of Leonardo Da Vinci, and a search for “The golden ratio in art” will reveal hundreds of papers claiming to have found the golden ratio in pretty much any possible artifact.

The only problem is that most of this likely is pure fiction. Although there are examples of artists who used the golden ratio extensively, many of these conclusions are cases of a post-rationalization where authors set out to find the golden ratio and so do it. This is no different than conspiracy theorists finding hidden symbols on the American dollar bills or fake shadows on photos from the moon landing. In a paper called *Misconceptions about the golden ratio* from 1992, George Markowsky argues:

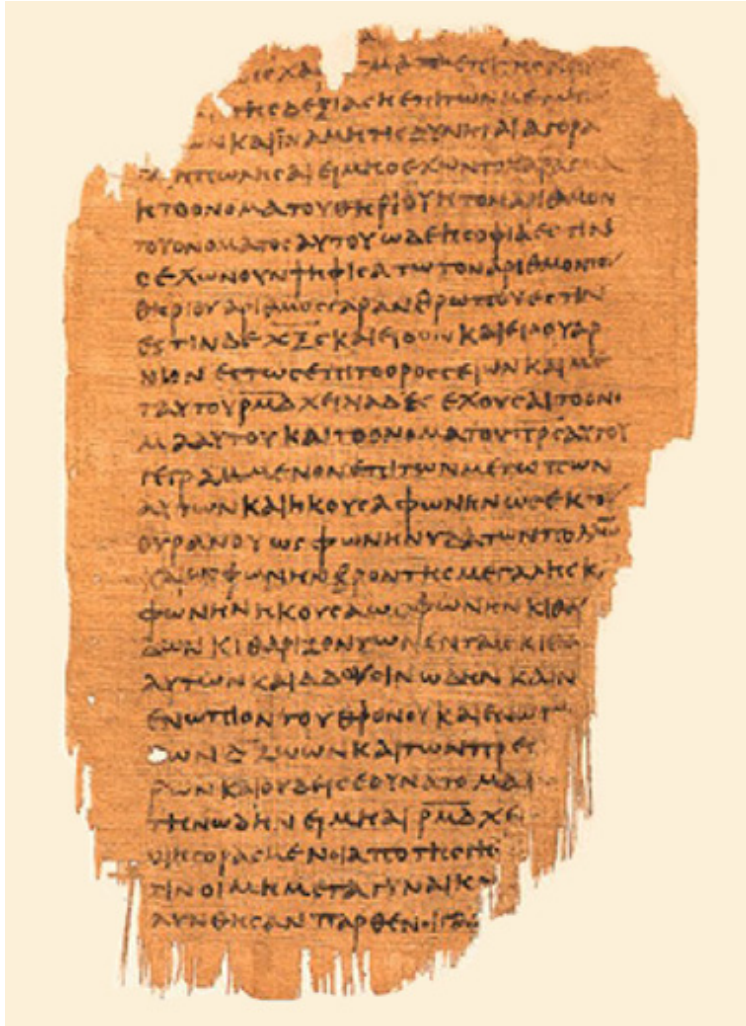
“Generally, the mathematical properties [of the golden ratio] are correctly stated, but much of what is presented about it in art, architecture, literature and esthetics is false or seriously misleading. Unfortunately, these statements about the golden ratio have achieved the status of common knowledge and are widely repeated.”

George Markowsky²⁰

As an example, the Parthenon was built in ancient Greece, and no source text indicates that the Athenians were aware of the golden ratio.

Furthermore, one has to ignore large parts of the building’s foundation in order to fit a golden rectangle on the structure of the building²¹. Leonardo Da Vinci did illustrate a book named *De Divina Proportione* about ratios in art, but there is no indication that he used these ideas much in his own

work. Although numerous academics have found the golden ratio in Da Vinci's work, these discoveries appear to be both subjective and flawed²². It is important to separate fact from fiction when discussing systematic design principles, and this is especially true for the subject of geometric composition. Rather than beginning a fruitless search for such divine proportions to serve a gross simplification of the world, we will instead focus on a more concrete history of how technology long has inspired designers to use geometric constraints to ease the burden of their work. As these techniques developed, they became design methodologies used even when designers were not limited by the constraints of the machine, and they are today widely adopted by graphic designers to make consistent and organized layouts.



Papyrus 47, the oldest surviving version of the Book of Revelation.

We will begin this story with an example that predates these ideas: The early handwritten book also called a codex. The codex was a significant improvement over the rolled papyrus, having stacked pages in a form that resembles the modern book. The popularity of the codex coincided – or was perhaps caused by – the rise of Christianity, and the earliest Christian manuscripts from the second century are all codices. These texts

are missing most of the presentational techniques we know today: The texts have no headings, no letter case, and the lack of punctuation turn them into one long paragraph. These texts were merely a medium for communicating the spoken word, and little thought has been put into the book as a medium. Most important, they were written by hand without any guides to help the scribe. As a result, the number of lines varies greatly from page to page, the lines often slope, and the right margin changes based on how the scribe decided to break the words²³.

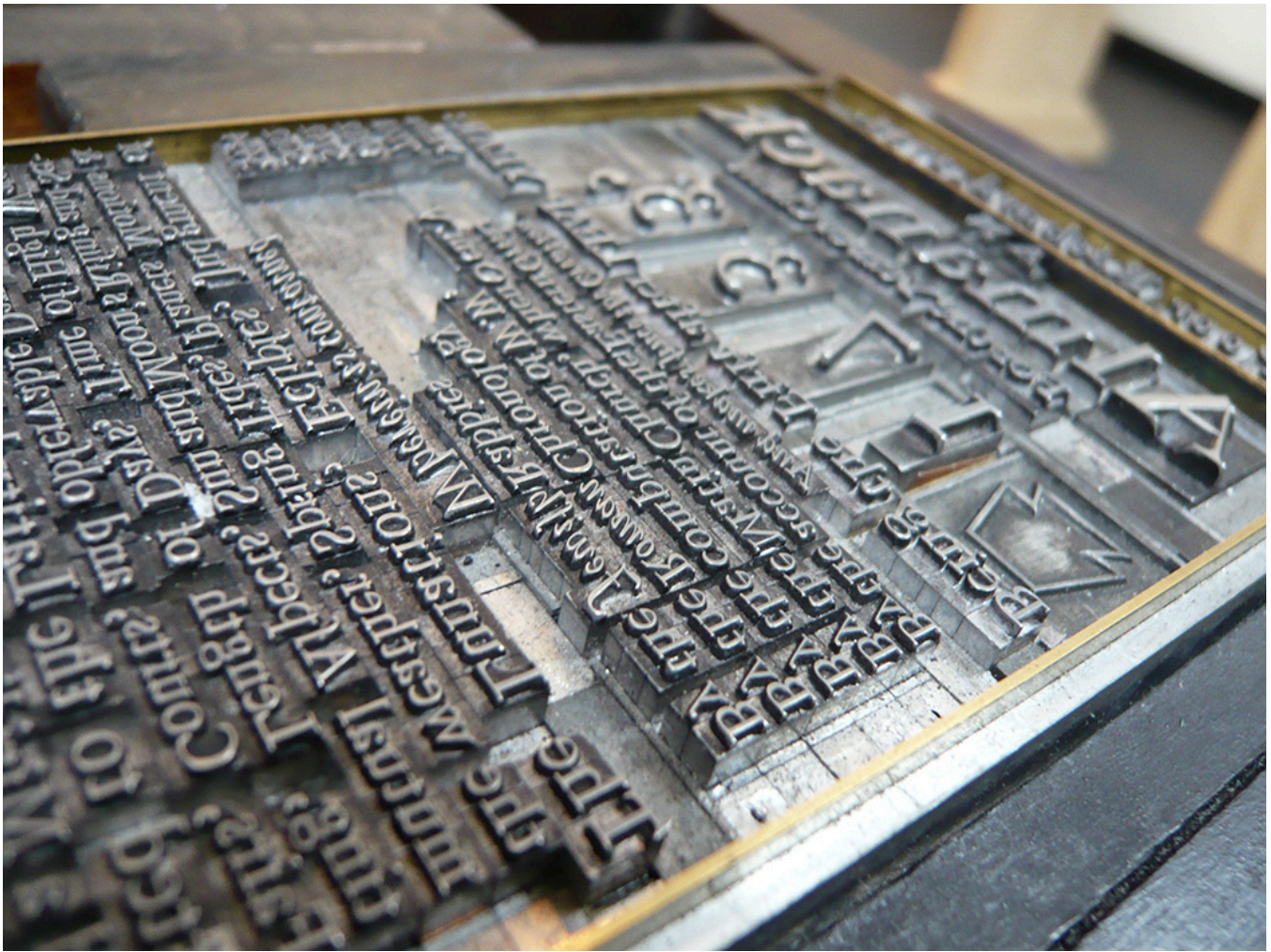
Compare this to the much later illuminated manuscripts – books written

by hand in monasteries – that flourished in the Middle Ages. These texts are much more deliberate in their layout. They use Gothic script, a type of lettering that is very slow to write, and manuscript pages are often painted with colorful miniature illustrations. Most important, these manuscripts were not created from a blank page. Scribes would plan the design of the book, and central to this planning was the creation of geometric helper lines to guide the scribe while writing. First, a rectangle would be drawn on the page to indicate the page margins. Then, this rectangle was split into a grid of horizontal lines to guide the text while some parts were set aside for illustrations. Finally, the scribe would carefully write and illustrate the book according to the grid. This is in essence no different to how designers today divide a canvas into smaller pieces to organize their layouts.

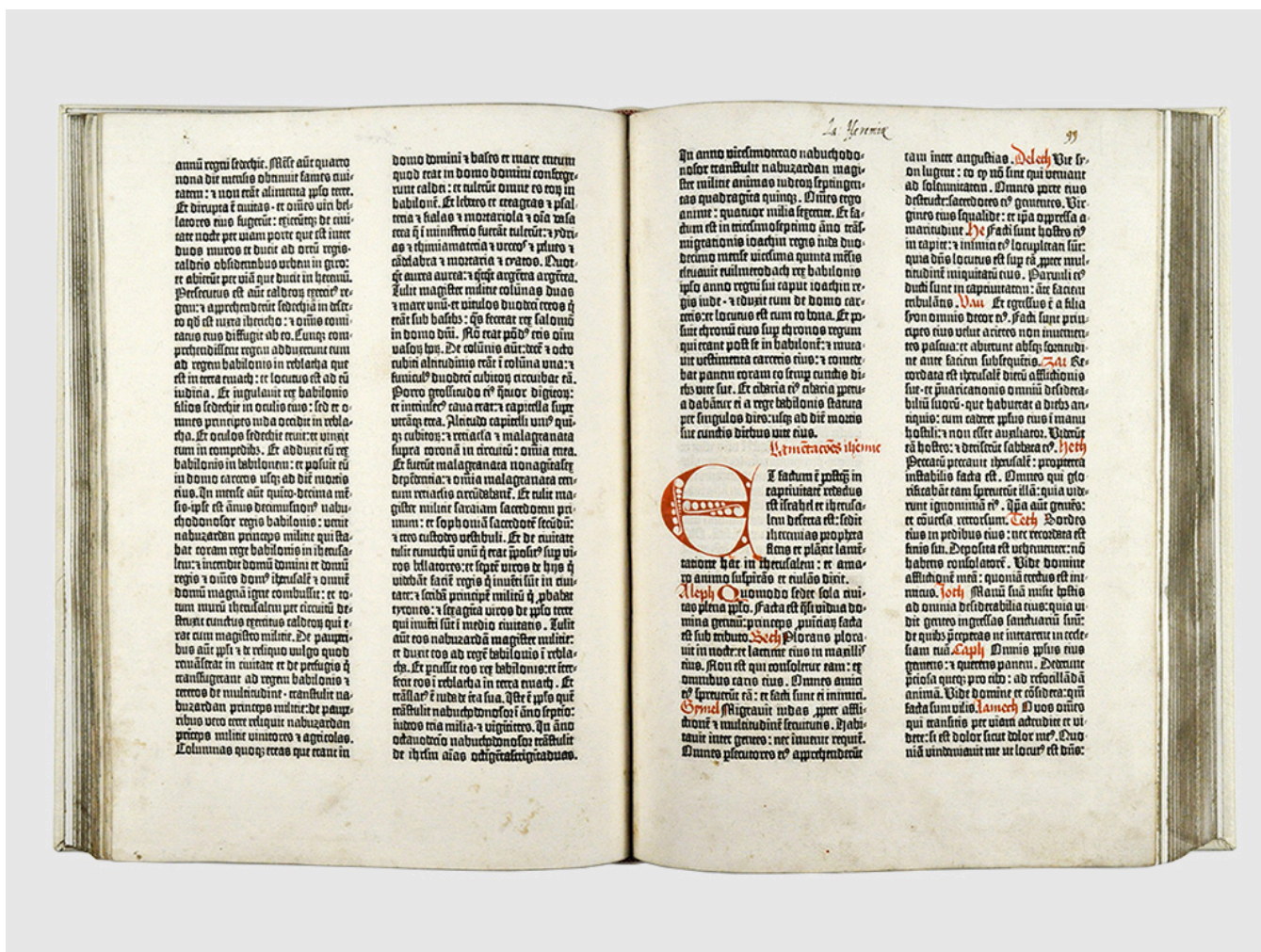


A picture of the Malmesbury Bible where the underlying grid clearly stands out next to the Gothic script and illustrations.

These methods changed from manual techniques to mechanical systems when first Bi Cheng in China (1045 AD) and later Johannes Gutenberg in Germany (1450 AD) invented movable type. This new printing technology used small clay, wood, or metal letter blocks that could be slotted onto a frame, inked and used to mass produce identical prints. One can imagine how the critics at the time must have sounded a lot like what we hear today about the computer: That these new machines constrain the artist and make the design products uniform. In the case of Gutenberg's early printed Bibles, one can argue that the opposite is true. The printing press did force Gutenberg to think about all aspects of book design in a rectangular grid, but the creativity displayed within these constraints is remarkable. To achieve a neat two-column layout, each letter was produced in multiple widths so the line of text could be justified in both ends. The typeface itself was the result of months of careful work producing letters that were both a reference to the Gothic script of yesterday and the mechanical processes of tomorrow. The printing press did constrain the design, but printers found a world of creativity within these geometric grids. Gutenberg's Bibles are, in the words of Stephen Fry, *"much more beautiful than [they] need to be"*²⁴.



Movable type letter blocks cast in lead. Photo by [purdman1](#) .



One of the last 49 surviving copies of the Gutenberg Bible.

The early 1900's saw a proliferation of new technology from the Industrial Revolution becoming available to artists and designers, and many of the artistic and political movements of this period incorporated these mechanical processes. Central to most of the European art movements (Italy's Futurism, France's Dadaism, Germany's Bauhaus), as well as the Marxist movements in Russia and China, was the use of the printing press as a way to rebel against the notion of fine art. One example is the Italian Futurists who, inspired by urbanism and the machine, experimented with the established grid of the printing press. If Gutenberg's life work was to fit designs into a rectangular grid, the Futurists saw it as their mission to break this grid.

“My revolution is aimed at the so-called harmony of the page, which is contrary to the flux and reflux, the leaps and bursts of style that run through the page. On the same page, therefore, we will use three or four colors of ink, or even twenty different typefaces if necessary”

Umbro Apollonio²⁵



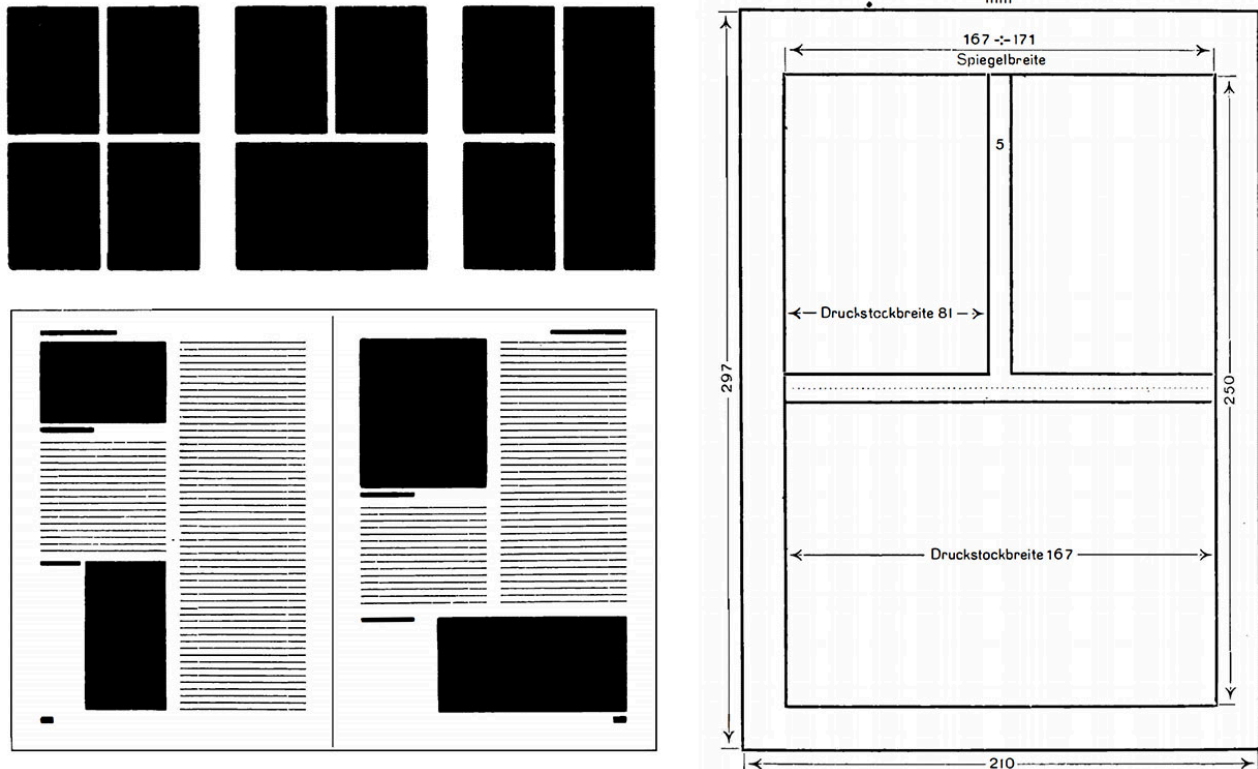
Ardengo Soffici from 1915.

This design by Ardengo Soffici shows a deliberate breaking of the grid with rectangular letter blocks arranged in a chaotic way. These experiments might not seem terribly groundbreaking to the modern eye, but they had a drastic impact on the graphic style of the following decades.

As graphic design evolved from a printmaker's task in the 1920's to a

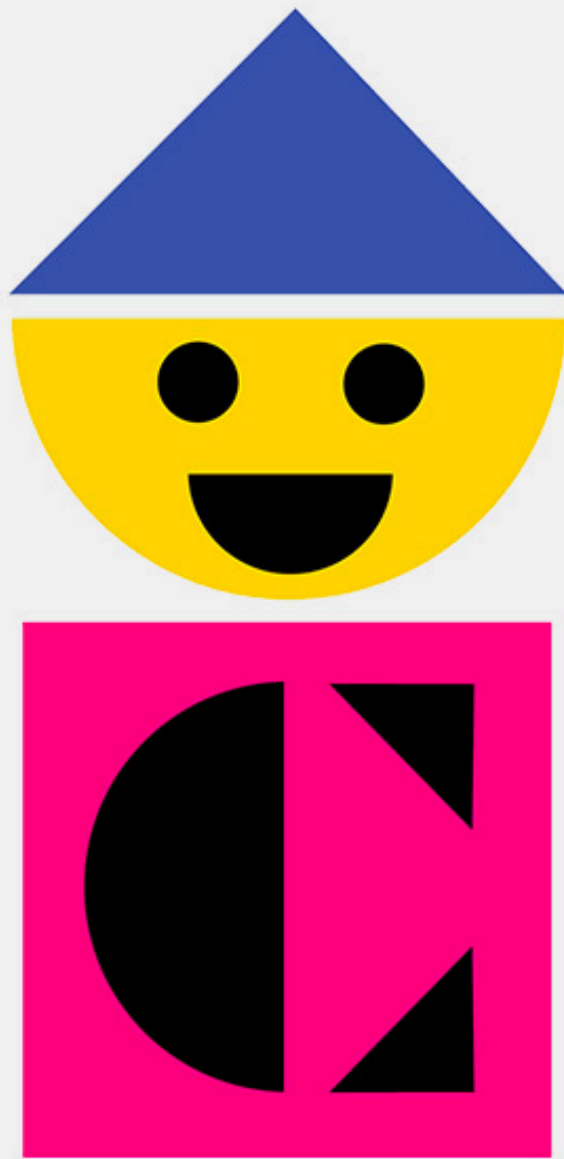
flourishing profession in the 1950's, these geometric composition techniques matured with it. In 1928, Jan Tschichold released *Die Neue Typographie*, a strict modernist manifesto that argued for the use of sans-

serif typefaces and uniform, left-aligned grid layouts. Although Tschichold did not use the word *grid* nor formalize its use, most of the techniques of the modern grid system were present: The division of the canvas into smaller, uniform rectangles to guide the text on the page.

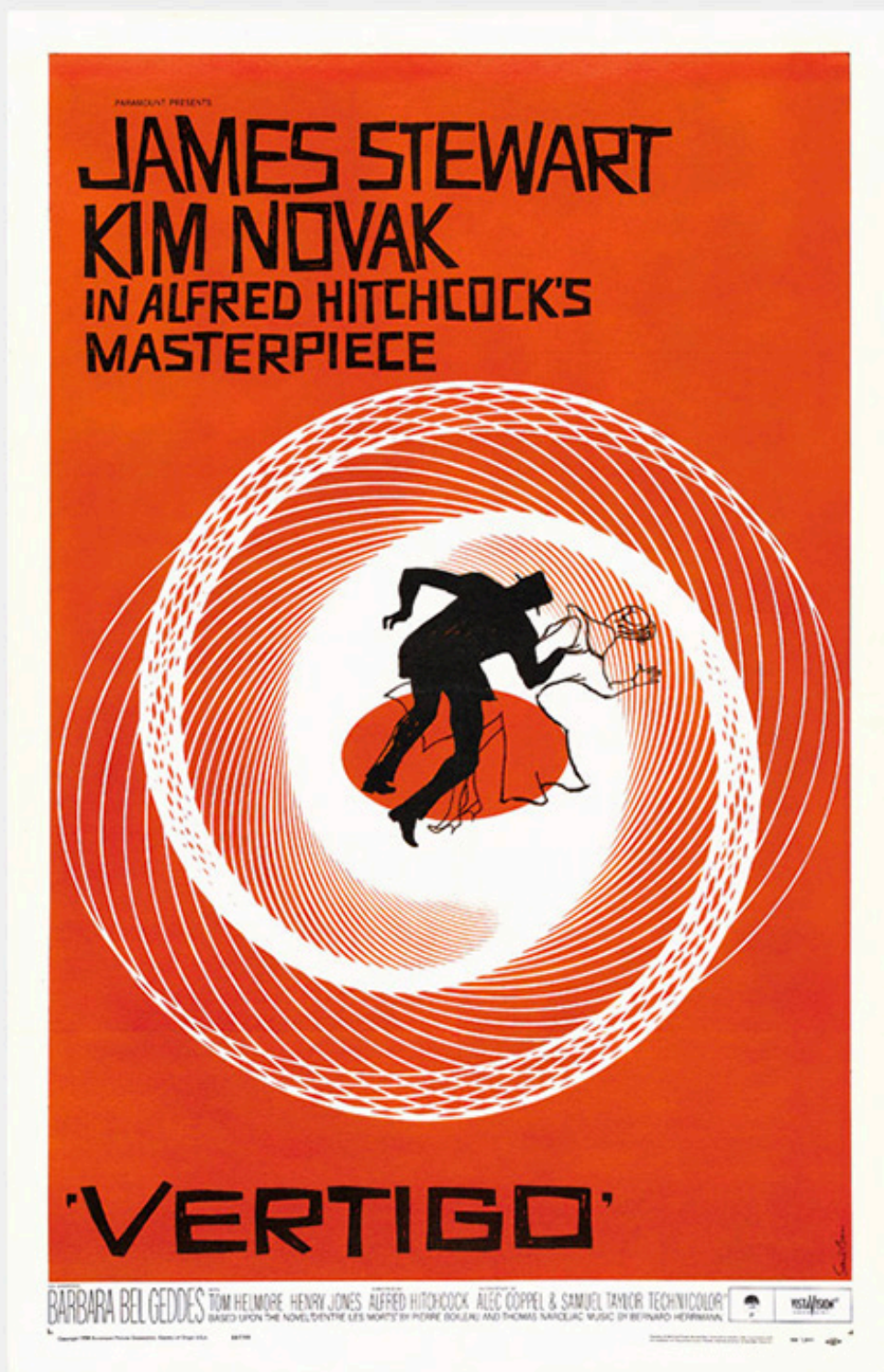


Examples of page divisions from Jan Tschichold's *Die Neue Typographie*. ©

These ideas quickly spread throughout Europe, and – propelled by European emigration caused by the Second World War – to the United States. Here, the grid system was adopted in products of the Golden Age of Advertising in the 1950's and 1960's by designers such as Paul Rand and Saul Bass who championed the use of expressive shapes and typography within the confines of the grid. When we look at these design products today, they stand out as incredible examples of this duality: An artistic playfulness constrained by geometric principles.



Colorforms logo by Paul Rand, 1959. ©



Vertigo poster by Saul Bass, 1959. ©

Tschichold's modernist ideas are clearly visible in one of the most famous advertising designs of the postwar era – The *Think Small* Volkswagen

campaign. The Volkswagen Beetle was originally designed by Ferdinand Porsche for the people of Adolf Hitler's Third Reich, and selling this car to the American consumer was no small feat. With clever text copy and a very modernist look, the *Think Small* campaigns helped make the Beetle a massive success in the US market.



© 1988 VOLKSWAGEN OF AMERICA, INC.

Think small.

Our little car isn't so much of a novelty any more.

A couple of dozen college kids don't try to squeeze inside it.

The guy at the gas station doesn't ask where the gas goes.

Nobody even stares at our shape.

In fact, some people who drive our little

flirver don't even think 32 miles to the gallon is going any great guns.

Or using five pints of oil instead of five quarts.

Or never needing anti-freeze.

Or racking up 40,000 miles on a set of tires.

That's because once you get used to

some of our economies, you don't even think about them any more.

Except when you squeeze into a small parking spot. Or renew your small insurance. Or pay a small repair bill. Or trade in your old VW for a new one.

Think it over.



The Think Small advertising campaign. ©



Lemon.

This Volkswagen missed the boot.

The chrome strip on the glove compartment is blemished and must be replaced. Chances are you wouldn't have noticed it; Inspector Kurt Kröner did.

There are 3,389 men at our Wolfsburg factory with only one job: to inspect Volkswagens at each stage of production. 13,000 Volkswagens are produced daily; there are more inspectors

than cars.)

Every shock absorber is tested (spot checking won't do), every windshield is scanned. VWs have been rejected for surface scratches barely visible to the eye.

Final inspection is really something! VW inspectors run each car off the line onto the Funktionsprüfstand (car test stand), tote up 189 check points, gun ahead to the automatic

broke stand, and say "no" to one VW out of fifty.

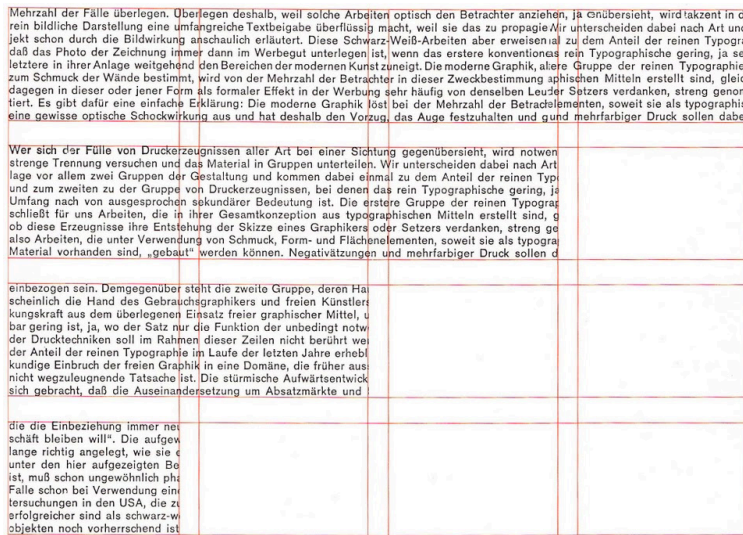
This preoccupation with detail means the VW lasts longer and requires less maintenance, by and large, than other cars. (It also means a used VW depreciates less than any other car.)



We pluck the lemons; you get the plums.

A new design aesthetic called the Swiss Style emerged in the 1960's,

where designers began arguing for an even more systematic and mathematical approach to design. Two books written by Swiss designers are considered central to this movement: *Grid Systems in Graphic Design* by Joseph Müller-Brockmann²⁶ and *Designing Programmes* by Karl Gerstner²⁷. Brockmann argued for an absolute minimalist design style and the use of grid systems to organize the elements of a design into a strict visual hierarchy.



A diagram from Joseph Müller-Brockmann's *Grid Systems in Graphic Design* illustrating how to align type to a grid. ©

It is important to understand how Brockmann's concept of a grid system separates itself from his predecessors. Brockmann believed that a grid system should be absolutely uniform, created by dividing the canvas into equally-sized columns and rows. Designers could then use these basic

building blocks – called modules – to flow text or images vertically along columns, horizontally along rows, or both. In Brockmann's view, there was no place for the designer to manually tweak the size of some elements by eye. The beauty came from the strictness of the proportional grid.

It is fair to say that the Swiss Style represented an extremist view of how geometric composition should be used in visual design. For fans of this purely functional art, the work of Brockmann holds great clarity because of its strictness and lack of ornaments. For critics, the same designs are

seen as cold and brutalist, lacking the creativity and playfulness seen in work by American colleagues.

Internationale Juni-Festwochen 1962 Stadttheater Zürich

Direktor
Dr. Herbert Graf

Freitag, 1. Juni
20.00 Uhr
Eröffnungsvorstellung

Fidelio
Oper von
L. van Beethoven

Leitung
Otto Klemperer
Hainer Hill

In den Hauptpartien
Jean Cook
Sena Jurinac
Heinz Borst
James McCracken
Deszö Ernster
Gustav Neidlinger
Leonhard Pöckl

Sonntag, 3. Juni
20.00 Uhr
Welturaufführung
Donnerstag, 7. Juni
20.00 Uhr

Blackwood und Co.
von Armin Schibler

Leitung
Nello Santi
Lotti Mansouri
Max Bignens
Juan Tena

Mittwoch, 6. Juni
19.30 Uhr
Freitag, 15. Juni
19.30 Uhr

Der Prophet
Oper von
G. Meyerbeer

Leitung
S. Krachmalnick
Lotti Mansouri
Hainer Hill
Michel de Lutry

In den Hauptpartien
Virginia Gordon
Sandra Warfield
Heinz Borst
James McCracken
Heinz Borst
Fritz Peter
Andrew Foldi
Siegfried Tappolet
Ralph Telasko

Freitag, 8. Juni
20.00 Uhr

Le Mystère de la
Nativité
von Frank Martin

Leitung
Ernest Ansermet
Georg Reinhardt
Heinrich Wendel

Mitwirkende
Mary Davenport
Regina Sarfaty
Vera Schlosser
Werner Ernst
Reinhold Glüther
Wolfram Mertz
Victor de Narké
Leonhard Pöckl
Fritz Peter
Glade Peterson
Abe Polakoff
Siegfried Tappolet
Ralph Telasko
Robert Thomas
Gottfried Zeithammer

Samstag, 9. Juni
20.00 Uhr

Il Trovatore
Oper von
Giuseppe Verdi

Leitung
Nello Santi
Herbert Graf
Max Röthlisberger

In den Hauptpartien
Virginia Gordon
Sandra Warfield
Heinz Borst
James McCracken
Abe Polakoff

Dienstag, 12. Juni
20.00 Uhr

Die Zauberflöte
Oper von
W.A. Mozart

Leitung
Hans Erismann
Rudolf Hartmann
Max Röthlisberger

Gastspiel
Maria Stader
Ernst Häfliger
Peter Lagger

Mittwoch, 13. Juni
19.30 Uhr

Die Fledermaus
Operette von
Johann Strauss

Leitung
S. Krachmalnick
Herbert Graf
Max Röthlisberger
René Hubert

In den Hauptpartien
Adèle Leigh
Eva-Maria Rogner
Regina Sarfaty
Wolfram Mertz
Leonhard Pöckl
Alfred Rasser
Rudolf Schock
Ralph Telasko
Robert Thomas

Samstag, 16. Juni
20.00 Uhr

Orpheus
und Eurydike
Oper von
Chr. W. von Gluck

Leitung
Robert F. Denzler
Hans Zimmermann
Max Röthlisberger
Jaroslav Berger

In der Hauptpartie
Regina Sarfaty

Sonntag, 17. Juni
20.00 Uhr
Mittwoch, 20. Juni
20.00 Uhr
Neu-Inszenierung

Der Freischütz
Oper von Carl Maria
von Weber

Leitung
Rudolf Kempe
Herbert Graf
Rudolf Heinrich

Gastspiel
Ingrid Bjoner
Hanny Steffek
Gottlob Frick
Fritz Uhl

Donnerstag, 21. Juni
20.00 Uhr

Die Nachtlall/
Die Geschichte
vom Soldaten
von Igor Strawinsky

Leitung
Victor Reinshagen
Hans Zimmermann
Hans Erni

In den Hauptpartien
Die Nachtlall:
Reri Grist
Glade Peterson
Die Geschichte
vom Soldaten:
Virginia Zango
Hans-Joachim Frick
Franz Matter
Bill Ross

Samstag, 23. Juni
19.00 Uhr
Dienstag, 26. Juni
19.00 Uhr

Der Rosenkavalier
Oper von
Richard Strauss

Leitung
Peter Maag
Herbert Graf
Max Röthlisberger

In den Hauptpartien
Lisa Della Casa
Anneliese
Rothenberger
Regina Sarfaty
Rudolf Knoll
James Pease

Sonntag, 24. Juni
20.00 Uhr

Il Barbiere
di Siviglia
Oper von
Gioacchino Rossini

Leitung
Nello Santi
Lotti Mansouri
Max Röthlisberger

In den Hauptpartien
Reri Grist
Heinz Borst
Fernando Corena
Robert Kerns
Fritz Peter

Mittwoch, 27. Juni
20.00 Uhr

Don Giovanni
Oper von
W.A. Mozart

Leitung
Peter Maag
Josef Giesen
Max Röthlisberger

In den Hauptpartien
Maria van Dongen
Reri Grist
Vera Schlosser
Heinz Borst
Fernando Corena
Werner Ernst
George London
Glade Peterson

Ballet
du XXIème Siècle
du Théâtre Royal
de la Monnaie
Bruxelles

Leitung
Maurice Béjart
André Vandermoot

Choreographie
Maurice Béjart
Janine Charrat

Freitag, 29. Juni
20.00 Uhr
Sonntag, 1. Juli
14.30 Uhr
1. Programm

Hommage
à Igor Strawinsky

Pulcinella
Musik von
Igor Strawinsky

Jeu de Cartes
Musik von
Igor Strawinsky

Le Sacre
du Printemps
Musik von
Igor Strawinsky

Samstag, 30. Juni
19.00 Uhr
Sonntag, 1. Juli
20.00 Uhr
2. Programm

Divertimento
Musik von
Fernand Schirren

Fantaisie
Concertante
Musik von
S. Prokofiev

Sonate à trois
Musik von
Béla Bartók

Bolero
Musik von
Maurice Ravel

**Opernhaus
Zürich**

**Eröffnung
der Spielzeit
1968 / 69**

Palestrina

Musikalische Legende von Hans Pfitzner

Erstaufführung
Samstag, 7. September, 19.00 Uhr

Musikalische Leitung:
Inszenierung:
Bühnenbild/Kostüme:
Chöre:

Alberto Erede
Herbert Graf
Max Röthlisberger
Hans Erismann

Der Wildschütz

Komische Oper von Albert Lorz

Neuinszenierung
Samstag, 14. September, 20.00 Uhr

Musikalische Leitung:
Inszenierung:
Bühnenbild/Kostüme:
Chöre:

Matthias Aeschbacher
Martin Markun
Monika von Zallinger
Hans Erismann

The concept of a grid system has so far served two purposes. First, it is the byproduct of the technical manufacturing processes of design products. The printing press works by assembling small rectangular letter blocks on a plate, so the designer has to think about the design process in terms of a grid. In this case, the grid is a concrete limitation that designers have to work around. Secondly, grid systems are also a design methodology used by designers to achieve organized layouts even in places that do not have such technical limitations. When grid systems became popular in web design in the mid-2000's, it was caused by a combination of these factors. On the one hand, HTML (the document format for websites) operate with rectangular elements that when rendered in a browser pushes subsequent elements down on the page. The column grid is therefore a natural container, as content can flow vertically without breaking the design. On the other hand, computers are not bound by the same limitations as mechanical systems, so the continued use of grid systems on the web today is as much a best practice that is independent of the constraints of the medium. A brief look at how nytimes.com, one of the most popular websites on the internet, evolved in its first decade might help illustrate this point.

The New York Times launched their daily news website in January 1996²⁸, just a few years after the internet finished its transition from a private network of academic institutions to a public, commercial medium²⁹. Although many of the web technologies we use today existed in some form, web designers were limited to a few basic techniques when creating websites. The first version of nytimes.com consisted of a single 575 x 300 pixel image split into clickable regions. As images provided the only way of creating consistent designs across browsers, this technique – called image mapping – was used heavily in the early days of the web. The design took much inspiration from the printed paper, featuring a centered logo, a date strip with the newspaper slogan (*"All the news that's fit to print"*, ironically), and a three-column layout. Given that this image was

created manually in a graphics program and not programmatically, many of the systematic qualities that we know from today's grid systems on the web were missing: The columns were not based on a uniform unit, and the content did not extend vertically below the screen area. As many new technologies, early web pages copied the form of existing technologies, and this web design is clearly reminiscent of the newspaper grid.



The New York Times website on November 12, 1996. ©

At the end of 1998, nytimes.com had replaced the static image with individual HTML elements, allowing their content to be updated individually throughout the day. This layout featured many of the techniques we consider vital to a modern user experience on the web: A sidebar with stacked menu navigation, listings of articles within a vertical column that extends beyond the screen, and tags to categorize this content. Still, this wider three-column layout did not conform to a uniform measurement for the grid columns, and pages varied in their presentation techniques.



The New York Times website on February 1, 1999.

©

This layout used HTML attributes and not a shared CSS file to achieve the design. This means that each page was responsible for its visual design, and much of the variation in grid measurements could be explained by the fact that designers at the New York Times weren't forced to think systematically about the design. As the use of CSS accelerated in the following years, web

designers began to argue for bringing the strict modernist grid to the web.

The first Google result for "CSS grid system" is from an article titled [Flexible Layouts with CSS Positioning](#) by Dug Falby on A List Apart in 2002. In the article, Falby argues for a universal, flexible layout system that can adapt to the size of the browser. In 2004, the Design Director for New York Times Online, Khoi Vinh, published a blog post titled [Grid Computing and Design](#) where he argued for the use of uniform grid units in web design:

"The new layout uses eight columns and four 'super columns,' and it shoehorns everything into that structure [...] Each column is 95 pixels wide and separated by a 10 pixel

gutter, which means I can create graphics of logical widths in increments of roughly 95 pixels each”

Khoi Vinh³⁰

This is the first online reference that explicitly describes a technique for web layouts that mirrors the techniques presented in Brockmann’s *Grid Systems in Graphic Design*. Vinh further developed these ideas in a presentation with Mark Boulton from South by Southwest in 2008 titled *Grids are Good*. Also in 2008, popular CSS libraries such as 960 Grid System and Blueprint CSS implemented this stricter grid philosophy, and these ideas have fundamentally not changed in modern CSS libraries used to this day.

[HOME PAGE](#) [MY TIMES](#) [TODAY'S PAPER](#) [VIDEO](#) [MOST POPULAR](#) [TIMES TOPICS](#) [Log In](#) [Register Now](#) [Free 14-Day Trial](#)

The New York Times

Wednesday, June 14, 2006 Last Update: 4:01 PM ET

[NYT Since 1981](#)

[Get Home Delivery](#) | [Personalize Your Weather](#)

[JOB MARKET](#)
[REAL ESTATE](#)
[AUTOS](#)

[WORLD](#)
U.S.
Washington
Education
[N.Y./REGION](#)
[BUSINESS](#)
[TECHNOLOGY](#)
[SPORTS](#)
[SCIENCE](#)
[HEALTH](#)
[OPINION](#)
[ARTS](#)
Books
Movies
Music
Television
Theater
[STYLE](#)
Dining & Wine
Fashion & Style
Home & Garden
Weddings & Celebrations
[TRAVEL](#)

[Blogs](#)
[Cartoons](#)
[Classifieds](#)
[Corrections](#)
[Crossword/ Games](#)
[Learning Network](#)
[NYC Guide](#)
[Obituaries](#)
[Podcasts](#)
[The Public Editor](#)
[Sunday Magazine](#)
[Weather](#)
[Week in Review](#)

[After Iraq Visit, Bush Urges Patience](#)

By CHRISTINE HAUSER 1:54 PM ET

The message suggested that although certain milestones have been achieved in Iraq, the U.S. mission now included broader measurements of success.

- [Video Report](#)
- [News Analysis: Bush Seizes on a Step Forward](#)
- [New Security Plan Brings Few Changes to Baghdad](#)

[Man Suspected of Stabbing 4 in N.Y. Is Arrested](#)

By AL BAKER and JOHN HOLUSHA 11:23 AM ET

The man was arrested shortly after two women were stabbed outside the W hotel at 47th and Broadway early this morning.

[Financial Crunch](#)



Musadeq Sadeq/Associated Press

[Afghans Call for Release of 47 From Guantánamo](#)

An Afghan government delegation said that half of the Afghans still being held were not guilty of serious crimes.

[Shares Plummet After Delay in Airbus Jet Delivery](#)

By NICOLA CLARK International Herald Tribune 2:33 PM ET

The parent company's stock slid by as much as a third after news that the A380 airplane is behind schedule.

[Led by Higher Rents, Prices Continue to Climb](#)

By JEREMY W. PETERS 2:52 PM ET

The report reaffirmed the belief among many economists and investors that the Fed will raise interest rates again.

- [Consumer Price Index Report](#) (pdf)
- [Core Producer Prices Rise a Little More Than Forecast](#)

[TRAVEL »](#)

[Buenos Aires Beef](#)

Off to the fair, champion bulls in tow, at the annual Argentine Rural Society Fair.

- [Slide Show](#)

[World Cup '06 LIVE](#)

[Germany-Poland](#)

Rob Mackey is covering all the action, minute by minute, of the match in Dortmund.

- [Tunisia 2, Saudi Arabia 2](#)
- [Spain 4, Ukraine 0](#)
- [World Cup Blog](#)
- [Scores and Standings](#)
- [Complete Coverage](#)

[OPINION »](#)

- [Friedman: G.M. — Again](#)
- [The Opinionator](#)
- [Cohen: Internet Politics](#)
- [Editorial: Bush's Trip](#)

[MARKETS](#)

4:30 PM ET BigCharts

DOW	10,816.92	+110.78	+1.03%
NAS	2,086.00	+13.53	+0.65%
\$SP	1,230.04	+6.35	+0.52%

[My Portfolio »](#)

Stock Quotes:

The New York Times website embraced many of these ideas with a 970px grid system in 2006. Although not strictly pixel perfect, the central design principle is an 11-column, 82-pixel column grid with a visual style that is visibly stricter than its predecessor. This format was used until a major redesign in 2013.

Grid systems are an integral part of digital design today. When building digital products, designers use the grid to establish empty containers that can be dynamically filled with content for the individual user. The way these grid systems are applied is essentially no different from how printmakers used them centuries ago, but one could argue that there is an even greater need for them today with the explosion of digital content: Digital design products must adapt to different screen sizes and show dynamic content, so there is no true original design. This forces designers to think systematically about visual layout. Hopefully, this short history has helped clarify how designers through centuries have used geometric composition techniques to create beautiful and balanced designs – in print and on the web. In the following chapters, we will investigate how to apply these ideas to our own designs in P5.js.

-
1. Loeb Classical Library (1936) *Aristotle's Minor Works*, p. 7. London
 2. Gottschalk, H. B. (1964) *The De Coloribus and Its Author*, p. 59-85. Hermes 92. Bd..H. 1: JSTOR. Web. 11 Jan. 2017
 3. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 25. University of Chicago Press
 4. Sloane, Patricia (1967) *Colour: Basic Principles New Directions*, p. 28-30. Studio Vista
 5. Lowengard, Sarah (2006) *The Creation of Color in Eighteenth-Century Europe New York*, para. 129-139, Columbia University Press)
 6. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 48. University of Chicago Press
 7. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 175-176. University of Chicago Press
 8. Munsell, A.H (1912) *A Pigment Color System and Notation*, pp. 239. The American Journal of Psychology. Vol. 23. University of Illinois Press

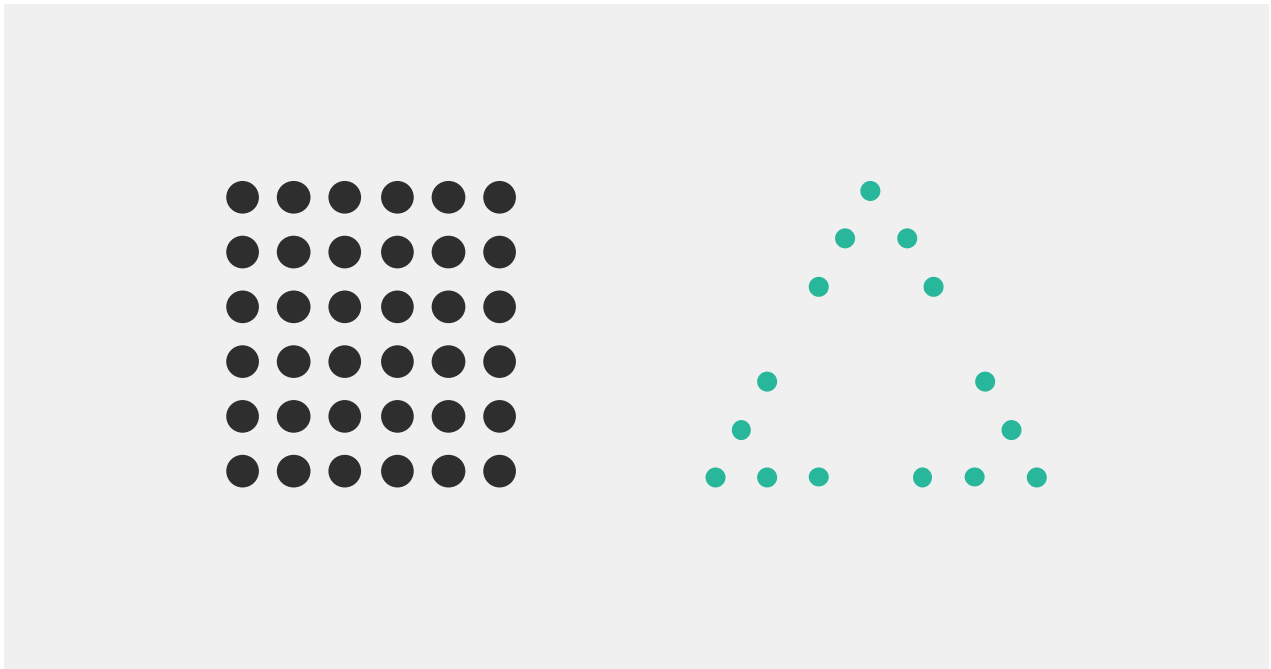
9. Itten, Johannes (1973) *The Art of Color: the subjective experience and objective rationale of color*. Van Nostrand Reinhold
10. Albers, Josef (1963) *Interaction of Color*. Yale University
11. Droste, Magdalena (2002) *Bauhaus*, p. 25. Taschen
12. Itten, Johannes (1970) *The Elements of Color*, p. 33-44. Van Nostrand Reinhold
13. Itten, Johannes (1970) *The Elements of Color*, p. 26. Van Nostrand Reinhold
14. MacEvoy, Bruce. *Color Vision Handprint : Colormaking Attributes*. N.p., 1 Aug. 2015. Web. 11 Jan. 2017.
15. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
16. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
17. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
18. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
19. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*, p. 9. Arthur Niggli
20. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 2. The College Mathematics Journal, Vol. 23
21. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 8-9. The College Mathematics Journal, Vol. 23
22. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 10-11. The College Mathematics Journal, Vol. 23
23. Malik, Peter (2017) *P. Beatty III (P47): The Codex, Its Scribe, and Its Text*, p. 31 - 38. New Testament tools, studies and documents, Vol. 52
24. Fry, Stephen (2008) *The Machine That Made Us*. BBC UK
25. Apollonio, Umbro (2009) *Futurist Manifestos*, p. 104. Tate Publishing
26. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*. Arthur Niggli
27. Gerstner, Karl (1964) *Designing Programmes*. Arthur Niggli
28. <https://www.nytc.com/who-we-are/culture/our-history/#2000-1971-timeline>
29. Peter, Ian (2004) *So, who really did invent the Internet?*. The Internet History Project
30. https://web.archive.org/web/20050106024725/http://www.subtraction.com:80/archives/2004/1231_grid_computi.php
31. Arnheim, Rudolf (1974) *Art and Visual Perception*, p. 8. University of California Press
32. Heider, G.M. (1977) *More about Hull and Koffka*, American Psychologist, 32(5), 383
33. Kroeger, Michael (2008) *Conversations With Students*, p. 27. Princeton Architectural Press
34. Design Systems International was co-founded by the author

Geometric composition

“By making visual categories explicit, by extracting underlying principles, and by showing structural relations at work, [the aim is] not to replace spontaneous intuition but to sharpen it, to shore it up, and to make its elements communicable.”

Rudolf Arnheim ³¹

In the late 1800's, psychologists in Germany performed a range of studies that would later form the foundation of Gestalt psychology. This new branch of psychology stated that humans, because we live in a complex world, seek to derive fast, simplified conclusions about what we see. The first thing most people see when presented with the drawing below is not 51 circles, but rather the groups these circles form based on their distance to each other and other visual similarity: A rectangle and a triangle. Observing how humans naturally try to turn a complex world into simple, actionable insights, the German psychologist Kurt Koffka would famously state that *“the whole is something else than the sum of its parts”* ³².






Humans naturally group complex input into simpler visual categories, such as these circles being perceived as a single rectangle and triangle

Because this branch of psychology is devoted to the mechanisms of perception, and since it emerged in Germany at the same time of the Bauhaus school, the Gestalt principles have long been used by artists and designers to anticipate the effects of their work. This is not a chapter on psychology, but Gestalt theory teaches us an important lesson about graphic design: Users of your design will naturally draw conclusions based on the entirety of your design, and if you do not formalize the content into a coherent layout, you are not in control of how the design is perceived. In other words, your entire design is a shape in itself, and that shape has to be designed too.

Those who are just beginning their design career might think that the ability to create clean and organized layouts is something that automatically comes with experience. Although practice does make perfect, it is remarkably hard to consistently arrange shapes on a page without a basic system to guide the decisions. Luckily, there is a technique that even the most gifted designers use to organize their layouts, achieve a

balance between the shapes used, and spark new ideas whenever creativity falls short. This chapter focuses on an important layout technique in graphic design often referred to as geometric composition, which entails dividing the canvas into smaller parts and using these divisions to arrange the visual elements. This technique can be used to create endless organized and expressive designs, and it happens to be a great technique for those of us using code in the design process.

Canvas Division

To demonstrate what geometric composition looks like in practice, let us imagine we are asked to design a poster for an upcoming photographic exhibition, and that the client wants this poster to hold exactly three important photos from the exhibition. In our first attempt at designing such a poster, we will position and scale the images to each take up one third of the canvas. Although this might not be the most thrilling layout, it guarantees that each image has an equal amount of space and that the horizontal lines created between the bordering images are evenly distributed from top to bottom. This is demonstrated in the code below where we are using three rectangles with different colored fills (  ) instead of images in order to not worry about image cropping. Notice how the height of each image is set to be exactly one-third of the canvas height, and how the `y` positions of the images are based on this value too.

```
const imgHeight = height / 3;

noStroke();
fill(75, 185, 165);
rect(0, 0, width, imgHeight);

fill(120, 155, 155);
rect(0, imgHeight, width, imgHeight);
```



```
fill(30, 50, 50);  
rect(0, 2 * imgHeight, width, imgHeight);
```

Many designers use this Rule of Thirds in situations where centering a shape is considered too dull or static. Some will even argue that placing important shapes around the thirds of the canvas will increase the dynamism and aesthetics of the design. Putting the validity of that discussion aside, we have already achieved something that would be hard to do without a layout system: Positioning and scaling three images evenly across a canvas. However, the result appears somewhat dense because there is no whitespace between the images. To make up for this, we can introduce margins (a term used to describe empty space around content) between our images to make the design more airy. The attentive reader will notice that the math required for calculating the height of each image is now a bit more complex. First, we have to find the size of the margin, which we calculate based on the canvas height in order to make our layout responsive in case we ever resize the canvas.

```
const margin = height / 20;
```

We then calculate the combined height of all three images, which is the canvas height without our margins. Note that since we are only adding the margin between the images, there are only two margins for three images.

```
const allHeight = height - 2 * margin;
```

Finally, we divide the `allHeight` variable by the number of images, which gives us the height of a single image. This value will also be used with the `margin` variable to calculate the position of each image.

```
const imgHeight = allHeight / 3;
```

The full code example below uses these variables to place each image at the correct position. By changing the value of the `margin` variable, we can easily increase or decrease the amount of whitespace, or even use the `random()` function to randomize the margin every time the code runs.

```
const margin = height / 20;  
const allHeight = height - 2 * margin;  
const imgHeight = allHeight / 3;  
  
background(240);  
noStroke();  
  
fill(75, 185, 165);  
rect(0, 0, width, imgHeight);  
  
fill(120, 155, 155);  
rect(0, imgHeight + margin, width, imgHeight);  
  
fill(30, 50, 50);  
rect(0, 2 * (imgHeight + margin), width, imgHeight);
```

We can continue our quest to add more whitespace by introducing margins around the edges of the canvas as well. Like a framed painting, this will remove the denseness of the layout even more, and call attention to each image as a separate piece of content. It will also allow us to add captions underneath each photo if necessary. We use the same code from above to calculate the height of each image, adding two extra margins for the top and bottom. We also use the same type of calculation for a new `imgWidth` variable to find the width of each image.

```
const margin = height / 20;
const imgWidth = width - 2 * margin;
const allHeight = height - 4 * margin;
const imgHeight = allHeight / 3;

background(240);
noStroke();
fill(30);

fill(75, 185, 165);
rect(margin, margin, imgWidth, imgHeight);

fill(120, 155, 155);
rect(margin, margin + imgHeight + margin, imgWidth, imgHeight);

fill(30, 50, 50);
rect(margin, margin + 2 * (imgHeight + margin), imgWidth,
imgHeight);
```

You might notice how the calculations are becoming longer the further down the page we go. This is because each image has to find its `y` position based on the number of images that came before it. There are always multiple ways of achieving the same outcome when writing code, and since some designers will find long calculations hard to read, let us instead rewrite the same example using the `translate()` function.

As described in earlier chapters, the `translate()` function can be used to move the canvas itself. If you draw a rectangle with `rect(0, 0, 50, 50)`, a square will normally appear in the top left corner of the canvas. However, if you write `translate(100, 100)` in the code before drawing the rectangle, the square will now appear `100` pixels down and `100` pixels to the right because of the translation. You might think of this like what happens when you click and drag on Google Maps: All shapes stay in the

same positions within the canvas, but the canvas itself is being moved around. One powerful (and challenging) aspect of translations is that they are cumulative, which means that they are added on top of previous translations. This is best demonstrated with a quick example. Notice how the second translation is added on top of the previous one, making the rectangles show up below each other despite having the same `x` and `y` values.

```
background(240);  
noStroke();  
fill(30);  
  
translate(175, 100);  
rect(0, 0, 150, 130);  
  
translate(0, 175);  
rect(0, 0, 150, 130);
```

We can therefore use the `translate()` function before drawing an image to move the canvas down to the correct position. We no longer need the long calculations for the last images, since the translations are added on top of each other as we go along. Note that we are adding a few more lines of code, but that is the compromise we have to make.

```
const margin = height / 20;  
const imgWidth = width - 2 * margin;  
const allHeight = height - 4 * margin;  
const imgHeight = allHeight / 3;  
  
background(240);  
noStroke();  
  
// Move down to the position of the first image and draw it
```

```
translate(margin, margin);
fill(75, 185, 165);
rect(0, 0, imgWidth, imgHeight);
// Move down to the second image position and draw it
translate(0, margin + imgHeight);
fill(120, 155, 155);
rect(0, 0, imgWidth, imgHeight);
// Move down to the last image position and draw it
translate(0, margin + imgHeight);
fill(30, 50, 50);
rect(0, 0, imgWidth, imgHeight);
```

The designs from the poster exercise might look simple, but they offer a first taste of an indisputable fact: Geometric composition is a great strategy when designing in code. It might take a little longer to write the code compared to making the same design in a traditional design tool, but the result is a pixel-perfect, balanced design that makes it easy to test variations by tweaking a single variable.

Procedural Layouts

So far, we have manually calculated the position for each image in our code based on the `margin`, `imgWidth`, and `imgHeight` variables. This might not be a problem when working with three images, but it quickly gets repetitive with a lot of content. As described in the [Procedural Shapes](#) chapter, a for-loop can be used to run the same piece of code multiple times after each other. Since all the images in the poster follow the same layout rule, we can use a for-loop to draw the images procedurally one after the other. The layout rule can be loosely translated to something like this: For each image, move down the value of `margin` (for the initial whitespace at the top of the canvas) and then move down the value of `imageHeight + margin` as many times as there are prior images. This description can be translated into code to look something

like this:

```
const imgY = margin + imageNum * (imageHeight + margin);
```

Note how this single line of code can be used to find the `y` position for each image by changing the value of `imageNum` from `0` (first image) to `2` (last image). We can put this calculation to work inside of a for-loop where the `i` variable also increments from `0` to `2` in steps of one, just like we need.

```
for(let i = 0; i < 3; i++) {  
  const imgY = margin + i * (imgHeight + margin);  
}
```

The example below uses this technique to draw the same layout with a for-loop. As explained in the [Color Schemes](#) chapter, we also use an array of color objects to draw different fills for the "images" with the same code.

```
const margin = height / 20;  
const imgWidth = width - 2 * margin;  
const allHeight = height - 4 * margin;  
const imgHeight = allHeight / 3;  
const colors = [  
  color(75, 185, 165),  
  color(120, 155, 155),  
  color(30, 50, 50)  
];  
  
background(240);  
noStroke();  
  
for(let i = 0; i < 3; i++) {  
  fill(colors[i]);
```

```
    const imgY = margin + i * (imgHeight + margin);  
    rect(margin, imgY, imgWidth, imgHeight);  
}
```

We can also rewrite this code to use the `translate()` function inside of the for-loop. However, we need to be smart about where exactly we put it. Since the first image needs to call `translate(margin, margin)` to position itself by the outer margin, but each of the following images need to call `translate(0, imgHeight + margin)` to move the canvas in the correct amount, we will do the first translation before the for-loop and the subsequent translations *after* drawing each image inside of the for-loop. This produces the same design as above, but the cumulative translations are easier to read for some. Keep in mind that both of these techniques are perfectly valid and you should use whichever one makes the most sense to you.

```
const margin = height / 20;  
const imgWidth = width - 2 * margin;  
const allHeight = height - 4 * margin;  
const imgHeight = allHeight / 3;  
const colors = [  
  color(75, 185, 165),  
  color(120, 155, 155),  
  color(30, 50, 50)  
];  
  
background(240);  
noStroke();  
  
// Translate to the position of the first image  
translate(margin, margin);  
for(let i = 0; i < 3; i++) {  
  fill(colors[i]);
```

```
rect(0, 0, imgWidth, imgHeight);  
  
// Translate to the position of the next image  
translate(0, imgHeight + margin);  
}
```

These are the key principles concerning geometric composition: To use a division of the canvas – with or without margins – to guide the size and position of the content. Now let us explore how to use these ideas in more detail, and uncover how to use the same techniques to make actual layouts in code.

The Grid System

Although we have only divided our canvas into thirds, the same method can be used with fewer or more divisions. If the poster example required us to use four photos, we would not need to make a lot of changes to the code to make that happen. However, as we increase the number of divisions, the space for each image becomes narrower, and at some point this will be unsuitable for our content. To make up for this, we can introduce another division – this time by dividing the canvas width – to add more flexibility to the layout system. This is the beginning of what in graphic design is referred to as a grid system.

Before: A single division with margins.

After: Two divisions with margins.

A few important comments are needed here. First, we might as well begin to use the correct terms now that we are using a proper grid system. Graphic designers often refer to the spaces within a grid system as *modules*, so this is the term that will be used from now on. Also, with all this talk about modules, one might easily be a bit confused: What are

these modules and where are they in the code? The concept of modules is in spirit comparable to the guides that designers use in design tools such as Photoshop or Illustrator. They are horizontal and vertical lines that can help the designer position the content, but they are not a part of the actual design. In the same way, the modules of our grid systems are numbers that we use to position and size the content, but besides storing these numbers in a few variables, the modules are not necessarily visible in the code nor the design.

With six modules, there is no longer a one-to-one mapping between the number of photos in our poster and the number of modules in the grid system. This makes it possible to explore how to use the grid in a more creative way. In order to do this, we first need to use the `imgWidth` and `imgHeight` calculations from earlier in this chapter to find the width and height of the modules. The code is exactly the same, except for renaming the variables to `moduleHeight` and `moduleWidth` and introducing a division in the latter.

```
const margin = height / 20;  
const allWidth = width - 3 * margin;  
const allHeight = height - 4 * margin;  
const moduleWidth = allWidth / 2;  
const moduleHeight = allHeight / 3;
```

So how do we draw three pieces of content inside six modules? The first option is to pick three modules to hold the images and leave the three remaining modules blank. This is the first time where we are presented with actual decisions to be made around composition, since we can create a great number of designs using this approach. The examples below demonstrate how to use this six-module grid system to create three designs with different levels of whitespace.

Layout with room for text to flow left-right-left next to each image. [See Code](#)

Layout with room for text in the middle of the canvas. [See Code](#)

Layout with room for text in the left-hand side of the canvas. [See Code](#)

The next option is to draw a single piece of content across multiple modules. This introduces the ability to highlight certain pieces of content by changing the relative scale between the photos. This is demonstrated in the example below where a single photo covers the uppermost four modules while the remaining photos use the last two modules at the bottom of the canvas. Note how the `translate()` function is used to minimize the code needed to calculate the positions of the photos.

```
const margin = height / 20;
const allWidth = width - 3 * margin;
const allHeight = height - 4 * margin;
const moduleWidth = allWidth / 2;
const moduleHeight = allHeight / 3;

background(240);
noStroke();

translate(margin, margin);
fill(75, 185, 165);
rect(0, 0, 2 * moduleWidth + margin, 2 * moduleHeight +
margin);

translate(0, 2 * (moduleHeight + margin));
fill(120, 155, 155);
rect(0, 0, moduleWidth, moduleHeight);

fill(30, 50, 50);
rect(moduleWidth + margin, 0, moduleWidth, moduleHeight);
```

This was a brief introduction to the concept of a grid system, and how the modules of a grid system can be used to position content within the canvas. As we continue our journey into the world of geometric composition, keep in mind that the core idea remains unchanged. That is, to use the modules as building blocks for a final composition.

Composition Strategies

“The idea of the grid is that it gives you a system of order and still gives you plenty of variety. [...] But the grid never changes. It is always the interior that changes, and that is what makes the thing come alive.”

Paul Rand³³

Our imaginary poster with three images was a worthwhile project for demonstrating the basics of geometric composition, but it is a somewhat simplified scenario compared to the type of content that a designer normally encounters. In order to explore this concept further, let us spend the rest of this chapter investigating other ways of using these same ideas in code, and by looking at designers who take different approaches to geometric composition. These designers utilize the grid in different ways. There are formalist designers who strictly follow the lines of the grid, more idea-driven designers who play within the grid, and designers who rarely use geometric helpers. Our first steps into the world of geometric composition is therefore constructed as – rather than a set of do’s and don’ts – a journey from a strict to a more lenient way of using geometric composition. Even though the examples vary in their number of divisions and use of the `translate()` function, they are all essentially using the same calculations as before. This time, we will also use content placeholders to simplify the code examples: A heading (■), a paragraph (■), and a picture (■).

A strict approach to geometric composition means that most of the content in your design should align with the lines created by the grid. In order to achieve this, paragraph text can be sized to the module and justified to create sharp edges on both the left and right side of the text block, and images can be scaled and cropped to take up the entire space of their module(s). The result is often a clean and balanced layout that, despite this rather formalist approach to composition, leaves much room for individual expression based on the modules and the content used. This is demonstrated below with a module structure that is very similar to our previous examples.

```
const margin = height / 30;
const allWidth = width - 3 * margin;
const allHeight = height - 5 * margin;
const moduleWidth = allWidth / 2;
const moduleHeight = allHeight / 4;

background(240);
noStroke();

fill(30, 50, 50);
rect(margin, margin, moduleWidth, moduleHeight / 4);

fill(120, 155, 155);
rect(margin + moduleWidth + margin, margin + moduleHeight +
margin, moduleWidth, moduleHeight);

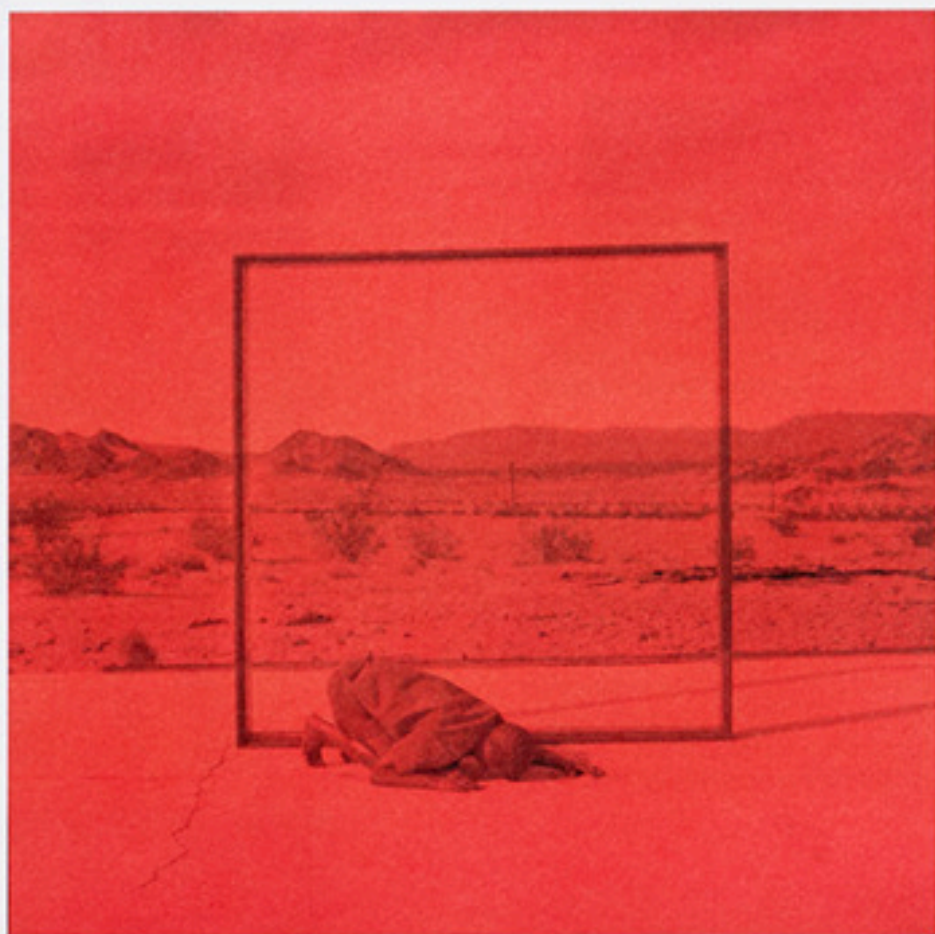
fill(75, 185, 165);
rect(margin, margin + 2 * (moduleHeight + margin), 2 *
moduleWidth + margin, 2 * moduleHeight + margin);
```

Toggle Grid

A real-world example of this composition technique can be found in the visual identity for the Whitney Museum designed by Dutch design studio Experimental Jetset. The identity system is based on a strict grid system where the vertical and horizontal lines of the content is broken up by a stretchable and skewed “W” that also functions as a logotype for the museum. The strictness of the grid system combined with the ever-changing dynamic logo makes for a good visual language for a white-wall museum that seeks to constantly renew itself through original exhibition work.

WHITNEY

**MEMBER
CALENDAR**
DEC 2016–
JAN 2017



This member calendar for Whitney Museum has a strict geometric layout similar to the code example above. ©

CURRENT EXHIBITIONS



**VIDA AMERICANA: MEXICAN MURALISTS
REMAKE AMERICAN ART, 1925–1945**
FEB 17–MAY 17, 2020



CAULEEN SMITH: MUTUALITIES
FEB 17–MAY 17, 2020



**AGNES PELTON: DESERT
TRANSCENDENTALIST**
MAR 13–JUNE 28, 2020



**SALMAN TOOR: HOW WILL I
KNOW**
MAR 20–JULY 5, 2020



**MAKING KNOWING: CRAFT IN
ART, 1950–2019**
NOV 22, 2019–JAN 2021

The website of Whitney Museum has a mix of different horizontal divisions, but the strict interpretation of the grid is visible throughout the website. ©

For designers who have a hard time creating compelling layouts from a blank canvas, this stricter approach to composition can serve as a concrete starting point for further exploration. The examples below show a number of variations of the same idea.

This layout uses only two of the four vertical divisions which makes for an ordered and balanced design with lots of whitespace. [Toggle Grid](#) [See Code](#)

A very large image with text content acting as small annotations. [Toggle Grid](#) [See Code](#)

The larger heading takes precedence over the smaller paragraph and image in a layout with heavy whitespace. [Toggle Grid](#) [See Code](#)

If this is an example of a strict interpretation of geometric composition, then what does it mean to design with a more lenient interpretation of the grid? The key is to find ways of removing the box-like aesthetic of the grid, and there are many ways of doing this. Let us first explore how to break the uniformity of the whitespace, and we will do this by allowing multiple pieces of content to use the same module. The overlapping content makes for a less formal composition while still adhering to the rules of the grid. The example below uses three horizontal divisions in order to allow content overlaps in the middle of the canvas. Notice how the heading is placed at the bottom of the bottom-most modules.

```
const margin = height / 15;
const allWidth = width - 4 * margin;
const allHeight = height - 5 * margin;
const moduleWidth = allWidth / 3;
const moduleHeight = allHeight / 4;
const headingHeight = moduleHeight / 2;
```

```
background(240);  
noStroke();  
  
translate(margin, margin);  
  
fill(75, 185, 165);  
rect(moduleWidth + margin, 0, 2 * moduleWidth + margin, 4 *  
moduleHeight + 3 * margin);  
  
fill(120, 155, 155);  
rect(0, moduleHeight + margin, 2 * moduleWidth + margin, 2 *  
moduleHeight + margin);  
  
fill(30, 50, 50);  
rect(0, 4 * moduleHeight + 3 * margin - headingHeight, 3 *  
moduleWidth + 2 * margin, headingHeight);
```

Toggle Grid

A real-world example of this approach can be found in the visual identity for CCC, an art cinema and cultural center in Santiago de Chile. The visual language created by the American design studio Design Systems International³⁴ in collaboration with Simón Sepúlveda is based on a simple layout system with three basic building blocks: A logo with three iconic C's, a grid system based on the rule of thirds, and a playful color palette. These elements can be combined to produce an endless number of assets for the institution. For an institution where most of the marketing material is created by a rotating team of volunteers, this simple but flexible layout system helps streamline their public communications, and the colored blocks produced by the geometric composition makes for an extremely recognizable identity. Notice how the text is placed around the margins of the canvas for a more playful expression.



02.12.2019

> miércoles 02, sala A
19:00hrs comentarios
Leonardo DiCaprio

DÍA DEL
A R T E

This poster for CCC has an overlapping geometric layout similar to the code example above. ©



> lunes 15, sala B
18:00hrs comentarios
Leonardo DiCaprio

CICLO
15.09.2019

CINE ITALIANO



28.11.2019

> lunes 18, sala C
18:00hrs comentarios Leonardo DiCaprio

S I L E N C E

The three designs below all use the same approach to create three different designs with overlapping content.

Multi-layered layout with a large heading. [Toggle Grid](#) [See Code](#)

A denser layout with text overlay typical for a photo book. [Toggle Grid](#) [See Code](#)

A dynamic composition with overlapping content and no margins. [Toggle Grid](#)
[See Code](#)

Another way to deviate from the strictness of the grid is to change the way our content is placed within the modules. Except for a few outliers, the strategy so far has been to scale the content to the full width and height of the module, and align the content at the top of the modules. We can open up more possibilities by scaling the content in different ways, and aligning the content to just one or two sides. This is demonstrated in the example below, where some images overflow the modules by the value of a margin, while other images shrink to align to a corner within their modules. This makes for a less organized layout where few of the shapes align, but without the chaos of a free-for-all layout.

```
const margin = height / 15;
const allWidth = width - 3 * margin;
const allHeight = height - 4 * margin;
const moduleWidth = allWidth / 2;
const moduleHeight = allHeight / 3;

background(240);
noStroke();
fill(75, 185, 165);

translate(margin, margin);

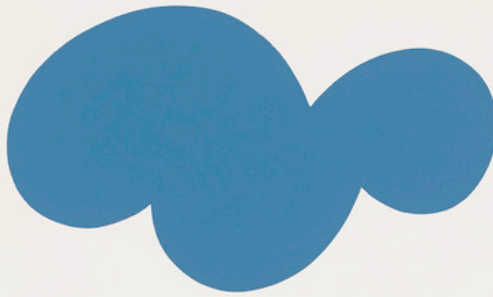
rect(0, margin / 2, moduleWidth, moduleHeight - margin);
```

```
rect(moduleWidth + margin, 0, moduleWidth, moduleHeight);  
rect(margin / 2, moduleHeight + margin, moduleWidth + margin /  
2, moduleHeight);  
rect(moduleWidth + 2 * margin, moduleHeight + margin,  
moduleWidth - margin, moduleHeight + margin);  
rect(0, 2 * (moduleHeight + margin), moduleWidth - margin,  
moduleHeight - margin);  
rect(moduleWidth, 2 * moduleHeight + 3 * margin, moduleWidth +  
margin, moduleHeight - margin);
```

Toggle Grid

The following two examples demonstrate different ways of playing within the grid. The first poster by the American graphic design Paul Rand has a layout similar to the code example above. It is arguable whether Rand used a grid system at all to create this design, but the consistent margins and alignment of some shapes might be an indication that the approach was similar to ours. The second example is by the American graphic designer Jacqueline Casey, who is best known for her work as Director of MIT's Office of Publications. Here, she plays with horizontal misalignment by dividing the canvas into many thin modules and offsetting the type from the center using the modules as a baseline for the typography. Combined with a black and white color scheme, the result is a rigid yet playful design that invites the reader to examine the text more closely.

background



May 5-9 '81



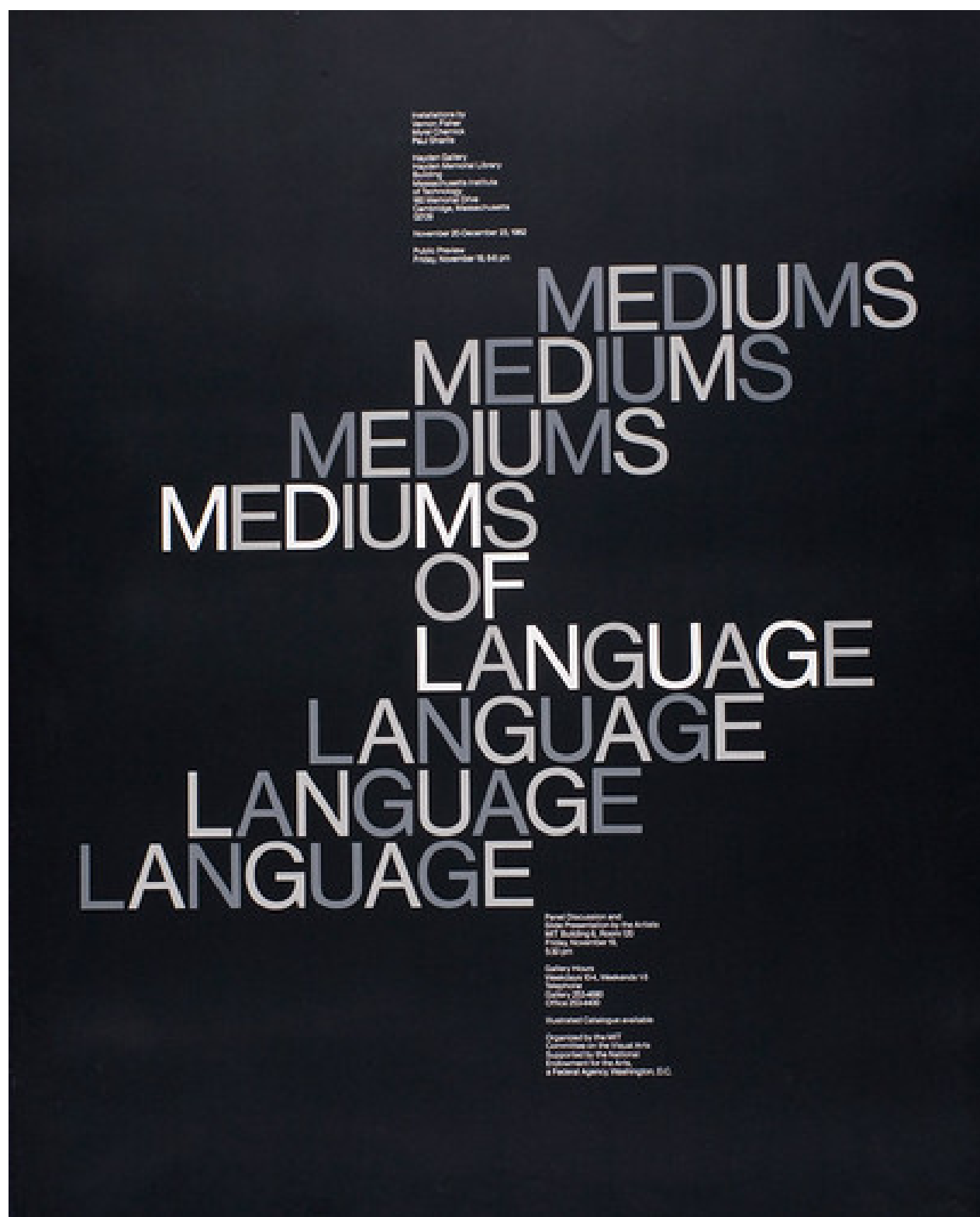
Golden Circle



Kauai Surf Hotel
Hawaii



IBM Golden Circle poster by Paul Rand. ©



Mediums of Language by Jacqueline Casey. ©

The three examples below use the same approach to create designs that look less rigid while still adhering to the main lines created by the grid.

This layout loosely uses the grid to position overlapping elements. This helps break the uniformity of the evenly sized modules. [Toggle Grid](#) [See Code](#)

This scrapbook-like layout has many smaller images that flow vertically along the modules. [Toggle Grid](#) [See Code](#)

A simple two-module grid is used to position content with a heading in the page margin. [Toggle Grid](#) [See Code](#)

This chapter introduced a set of geometric composition techniques that can be used to construct graphic layouts by dividing the canvas into smaller modules that are in turn used to arrange the content. These ideas can help designers create a plan for the entire composition before focusing on individual pieces of content. In some way, geometrical composition is a way to think about the general before the specific, and this can be a blessing or a curse depending on how these ideas are used. As always, rules should not be followed blindly. Graphic design is above all a human endeavour, and geometric composition techniques should be used to explore layouts and generate new ideas suitable for the specific content, not as a prison for templating content into a visual monoculture. It should be mentioned that the techniques presented in this chapter were meant to introduce the concept of geometric composition to designers working in code, but P5.js cannot and should not be used for all purposes. Many designers will need to use grid systems when designing websites, but this will need to be done with CSS. It is my hope that even though the specifics of each programming language will vary, the overall approach to designing with grid systems in code will remain relevant.

The next chapter of this book is also devoted to the concept of grid systems. This will give us an opportunity to focus on key details related to

the use of grid systems in code, including how to write reusable code that can be shared across projects, using more sophisticated grid systems, and even using multiple grid systems on top of each other in the same design.

EXERCISE

Pick one of your favorite musical artists and design a digital banner for this artist using the artist name, a short description, and an image. The banner is needed to promote the artist on the home page of a musical streaming service. Ask yourself whether the style of music calls for a rigid layout or a more loose interpretation of the grid, and try to come up with a layout that makes sense for your content. Make sure to use the code examples in this chapter if you are stuck.

-
1. Loeb Classical Library (1936) *Aristotle's Minor Works*, p. 7. London
 2. Gottschalk, H. B. (1964) *The De Coloribus and Its Author*, p. 59-85. *Hermes* 92. Bd..H. 1: JSTOR. Web. 11 Jan. 2017
 3. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 25. University of Chicago Press
 4. Sloane, Patricia (1967) *Colour: Basic Principles New Directions*, p. 28-30. Studio Vista
 5. Lowengard, Sarah (2006) *The Creation of Color in Eighteenth-Century Europe New York*, para. 129-139, Columbia University Press)
 6. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 48. University of Chicago Press
 7. Ball, Philip (2003) *Bright Earth: Art and the Invention of Color*, p. 175-176. University of Chicago Press
 8. Munsell, A.H (1912) *A Pigment Color System and Notation*, pp. 239. The American Journal of Psychology. Vol. 23. University of Illinois Press
 9. Itten, Johannes (1973) *The Art of Color: the subjective experience and objective rationale of color*. Van Nostrand Reinhold
 10. Albers, Josef (1963) *Interaction of Color*. Yale University
 11. Droste, Magdalena (2002) *Bauhaus*, p. 25. Taschen
 12. Itten, Johannes (1970) *The Elements of Color*, p. 33-44. Van Nostrand Reinhold
 13. Itten, Johannes (1970) *The Elements of Color*, p. 26. Van Nostrand Reinhold
 14. MacEvoy, Bruce. *Color Vision Handprint : Colormaking Attributes*. N.p., 1 Aug. 2015. Web. 11

Jan. 2017.

15. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
16. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
17. O'Connor, Zena (2010) *Color Harmony Revisited*, p. 267-273. Color Research and Application. Volume 35, Issue 4
18. Gegenfurtner, Karl. R.; Sharpe, Lindsay. T. (2001) *Color Vision: From Genes to Perception*, p. 3-11. Cambridge University Press
19. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*, p. 9. Arthur Niggli
20. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 2. The College Mathematics Journal, Vol. 23
21. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 8-9. The College Mathematics Journal, Vol. 23
22. Markowsky, George (1992) *Misconceptions about the Golden Ratio*, p. 10-11. The College Mathematics Journal, Vol. 23
23. Malik, Peter (2017) *P. Beatty III (P47): The Codex, Its Scribe, and Its Text*, p. 31 - 38. New Testament tools, studies and documents, Vol. 52
24. Fry, Stephen (2008) *The Machine That Made Us*. BBC UK
25. Apollonio, Umbro (2009) *Futurist Manifestos*, p. 104. Tate Publishing
26. Müller-Brockmann, Josef (1981) *Grid Systems in Graphic Design*. Arthur Niggli
27. Gerstner, Karl (1964) *Designing Programmes*. Arthur Niggli
28. <https://www.nytco.com/who-we-are/culture/our-history/#2000-1971-timeline>
29. Peter, Ian (2004) *So, who really did invent the Internet?*. The Internet History Project
30. https://web.archive.org/web/20050106024725/http://www.subtraction.com:80/archives/2004/1231_grid_computi.php
31. Arnheim, Rudolf (1974) *Art and Visual Perception*, p. 8. University of California Press
32. Heider, G.M. (1977) *More about Hull and Koffka*, American Psychologist, 32(5), 383
33. Kroeger, Michael (2008) *Conversations With Students*, p. 27. Princeton Architectural Press
34. Design Systems International was co-founded by the author

... more to come

You have reached the end of a part of this book that is still being written.
Please subscribe to the newsletter to receive updates about new chapters!