Contents

1 Introduction

The images above were made in different ways. The first one was made by Van Gogh's hand applying layer over layer of paint. It took him hours. The second was produced in seconds by the combination of four matrices of pixels: one for cyan, one for magenta, one for yellow and one for black. The key difference is that the second image is produced in a non-serial way (that means not step-by-step, but all at the same time).

This book is about the revolutionary computational technique, *fragment shaders*, that is taking digitally generated images to the next level. You can think of it as the equivalent of Gutenberg's press for graphics.

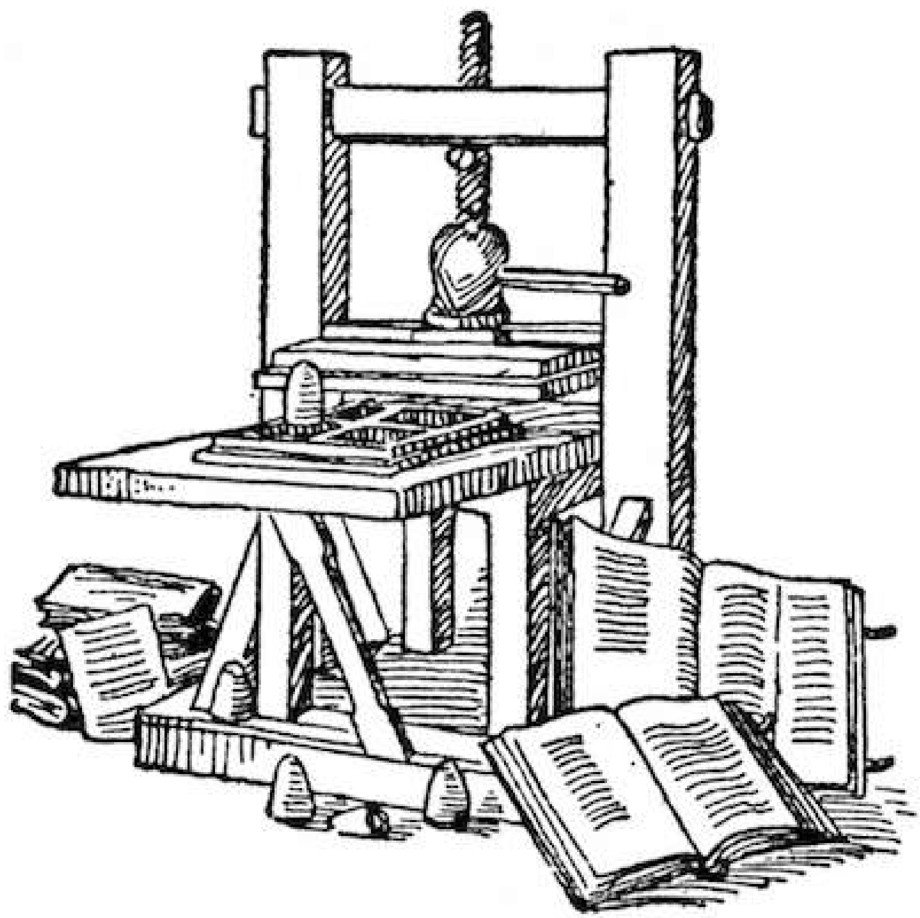


Figure 1: Gutenberg's press

Fragment shaders give you total control over the pixels rendered on the screen at a super fast speed. This is why they're used in all sort of cases, from video filters on cellphones to incredible 3D video games.

Figure 2: Journey by That Game Company

In the following chapters you will discover how incredibly fast and powerful this technique is and how to apply it to your professional and personal work.

1.1 Who is this book for?

This book is written for creative coders, game developers and engineers who have coding experience, a basic knowledge of linear algebra and trigonometry, and who want to take their work to an exciting new level of graphical quality. (If you want to learn how to code, I highly recommend you start with Processing and come back later when you are comfortable with it.)

This book will teach you how to use and integrate shaders into your projects, improving their performance and graphical quality. Because GLSL (OpenGL Shading Language) shaders compile and run on a variety of platforms, you will be able to apply what you learn here to any enviroment that uses OpenGL, OpenGL ES or WebGL. In other words, you will be able to apply and use your knowledge with Processing sketches, openFrameworks applications, Cinder interactive installations, Three.js websites or iOS/Android games.

1.2 What does this book cover?

This book will focus on the use of GLSL pixel shaders. First we'll define what shaders are; then we'll learn how to make procedural shapes, patterns, textures and animations with them. You'll learn the foundations of shading language and apply it to more useful scenarios such as: image processing (image operations, matrix convolutions, blurs, color filters, lookup tables and other effects) and simulations (Conway's game of life, Gray-Scott's reaction-diffusion, water ripples, watercolor effects, Voronoi cells, etc.). Towards the end of the book we'll see a set of advanced techniques based on Ray Marching.

There are interactive examples for you to play with in every chapter. When you change the code, you will see the changes immediately. The concepts can be abstract and confusing, so the interactive examples are essential to helping you learn the material. The faster you put the concepts into motion the easier the learning process will be.

What this book doesn't cover:

- This *is not* an openGL or webGL book. OpenGL/webGL is a bigger subject than GLSL or fragment shaders. To learn more about openGL/webGL I recommend taking a look at: OpenGL Introduction, the 8th edition of the OpenGL Programming Guide (also known as the red book) or WebGL: Up and Running
- This *is not* a math book. Although we will cover a number of algorithms and techniques that rely on an understanding of algebra and trigonometry, we will not explain them in detail. For questions regarding the math I recommend keeping one of the following books nearby: 3rd Edition of Mathematics for 3D Game Programming and computer Graphics or 2nd Edition of Essential Mathematics for Games and Interactive Applications.

1.3 What do you need to start?

Not much! If you have a modern browser that can do WebGL (like Chrome, Firefox or Safari) and a internet connection, click the "Next" Chapter button at the end of this page to get started.

Alternatively, based on what you have or what you need from this book you can:

- Make an off-line version of this book
- Run the examples on a Raspberry Pi without a browser
- Make a PDF of the book to print
- Check the GitHub repository of this book to help resolve issues and share code.

2 Getting started

2.1 What is a fragment shader?

In the previous chapter we described shaders as the equivalent of the Gutenberg press for graphics. Why? And more importantly: what's a shader?

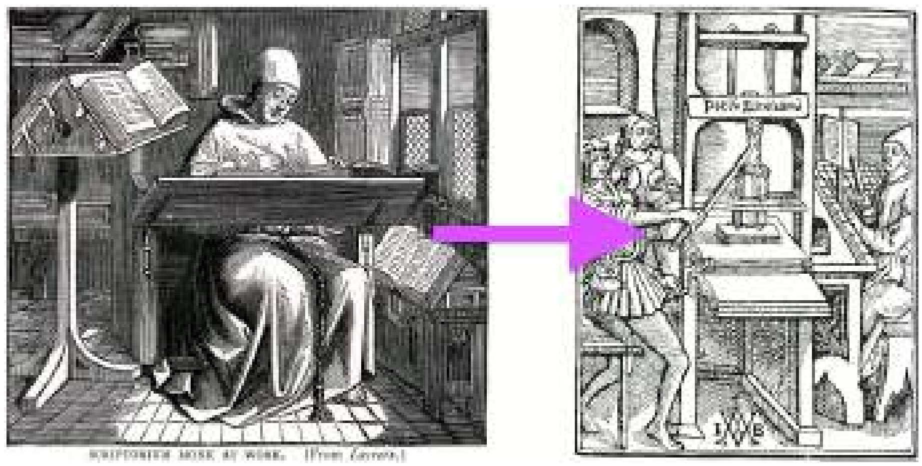


Figure 3: From Letter-by-Letter, Right: William Blades (1891). To Page-by-page, Left: Rolt-Wheeler (1920).

If you already have experience making drawings with computers, you know that in that process you draw a circle, then a rectangle, a line, some triangles until you compose the image you want. That process is very similar to writing a letter or a book by hand - it is a set of instructions that do one task after another.

Shaders are also a set of instructions, but the instructions are executed all at once for every single pixel on the screen. That means the code you write has to behave differently depending on the position of the pixel on the screen. Like a type press, your program will work as a function that receives a position and returns a color, and when it's compiled it will run extraordinarily fast.

2.2 Why are shaders fast?

To answer this, I present the wonders of *parallel processing*.

Imagine the CPU of your computer as a big industrial pipe, and every task as something that passes through it - like a factory line. Some tasks are bigger than others, which means they require more time and energy to deal with. We say they require more processing power. Because of the architecture of computers the jobs are forced to run in a series; each job has to be finished one at a time. Modern computers usually have groups of four processors that work like these

Figure 4: Chinese movable type

pipes, completing tasks one after another to keep things running smoothly. Each pipe is also known as a *thread*.

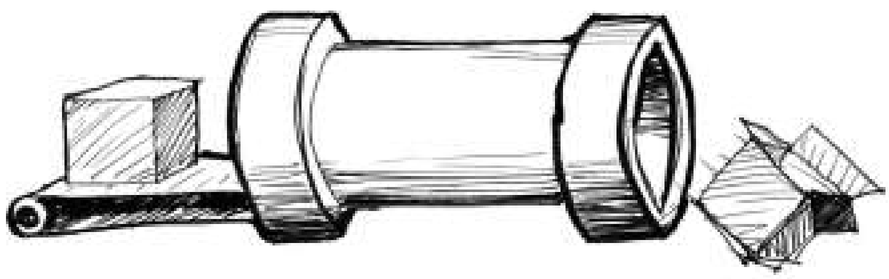
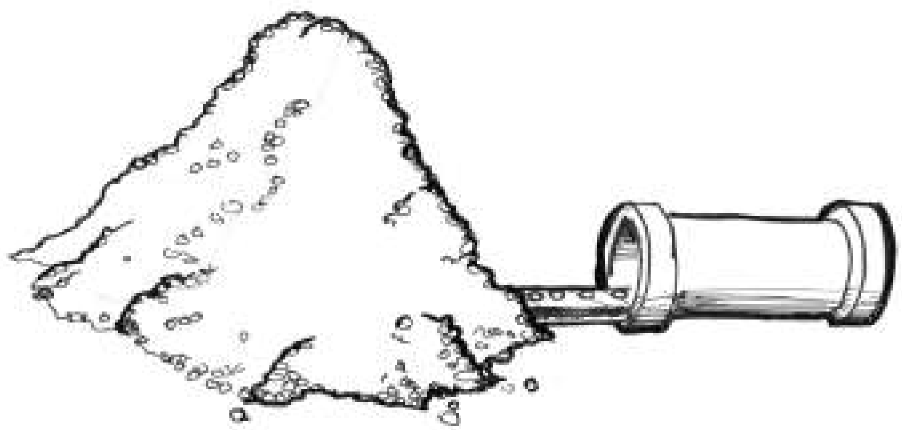


Figure 5: CPU

Video games and other graphic applications require a lot more processing power than other programs. Because of their graphic content they have to do huge numbers of pixel-by-pixel operations. Every single pixel on the screen needs to be computed, and in 3D games geometries and perspectives need to be calculated as well.

Let's go back to our metaphor of the pipes and tasks. Each pixel on the screen represents a simple small task. Individually each pixel task isn't an issue for the CPU, but (and here is the problem) the tiny task has to be done to each pixel on the screen! That means in an old 800x600 screen, 480,000 pixels have to processed per frame which means 14,400,000 calculations per second! Yes! That's a problem big enough to overload a microprocessor. In a modern 2880x1800 retina display running at 60 frames per second that calculation adds up to 311,040,000 calculations per second. How do graphics engineers solve this problem?



This is when parallel processing becomes a good solution. Instead of having a couple of big and powerful microprocessors, or *pipes*, it is smarter to have lots of tiny microprocessors running in parallel at the same time. That's what a Graphic Processor Unit (GPU) is.

Picture the tiny microprocessors as a table of pipes, and the data of each pixel as a ping pong ball. 14,400,000 ping pong balls a second can obstruct almost any pipe. But a table of 800x600 tiny pipes receiving 30 waves of 480,000 pixels a second can be handled smoothly. This works the same at higher resolutions - the more parallel hardware you have, the bigger the stream it can manage.

Another "super power" of the GPU is special math functions accelerated via hardware, so complicated math operations are resolved directly by the microchips instead of by software. That means extra fast trigonometrical and matrix operations - as fast as electricity can go.

2.3 What is GLSL?

GLSL stands for openGL Shading Language, which is the specific standard of shader programs you'll see in the following chapters. There are other types of shaders depending on hardware and Operating Systems. Here we will work with the openGL specs regulated by Khronos Group. Understanding the history of OpenGL can be helpful for understanding most of its weird conventions, for that I recommend taking a look at: openglbook.com/chapter-0-preface-what-is-opengl.html

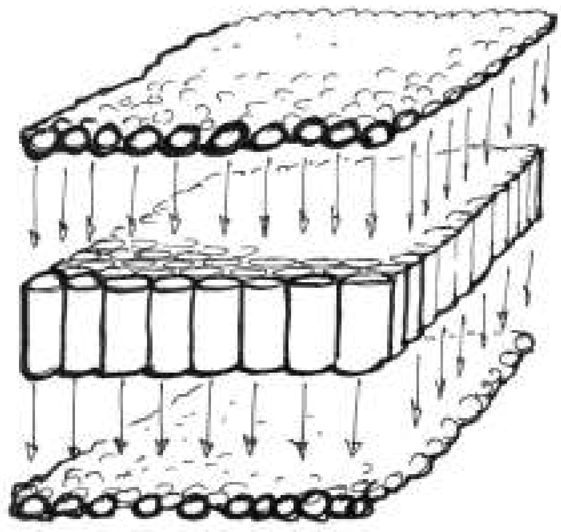


Figure 6: GPU

2.4 Why are Shaders famously painful?

As Uncle Ben said "with great power comes great responsibility," and parallel computation follows this rule; the powerful architectural design of the GPU comes with its own constraints and restrictions.

In order to run in parallel every pipe, or thread, has to be independent from every other thread. We say the threads are *blind* to what the rest of the threads are doing. This restriction implies that all data must flow in the same direction. So it's impossible to check the result of another thread, modify the input data, or pass the outcome of a thread into another thread. Allowing thread-to-thread communications puts the integrity of the data at risk.

Also the GPU keeps the parallel micro-processor (the pipes) constantly busy; as soon as they get free they receive new information to process. It's impossible for a thread to know what it was doing in the previous moment. It could be drawing a button from the UI of the operating system, then rendering a portion of sky in a game, then displaying the text of an email. Each thread is not just **blind** but also **memoryless**. Besides the abstraction required to code a general function that changes the result pixel by pixel depending on its position, the blind and memoryless constraints make shaders not very popular among beginning programmers.

Don't worry! In the following chapters, we will learn step-by-step how to go from simple to advanced shading computations. If you are reading this with a modern browser, you will appreciate playing with the interactive examples. So let's not delay the fun any longer and press *Next* >> to jump into the code!

2.5 Hello World

Usually the "Hello world!" example is the first step to learning a new language. It's a simple one-line program that outputs an enthusiastic welcoming message and declares opportunities ahead.

In GPU-land rendering text is an overcomplicated task for a first step, instead we'll choose a bright welcoming color to shout our enthusiasm!

```
#ifdef GL_ES
precision mediump float;
#endif

uniform float u_time;

void main() {
        gl_FragColor = vec4(1.0,0.0,1.0,1.0);
}
```

If you are reading this book in a browser the previous block of code is interactive. That means you can click and change any part of the code you want to explore. Changes will be updated immediately thanks to the GPU architecture that compiles and replaces shaders *on the fly*. Give it a try by changing the values on line 8.

Although these simple lines of code don't look like a lot, we can infer substantial knowledge from them:

1. Shader Language has a single `main` function that returns a color at the end. This is similar to C.

2. The final pixel color is assigned to the reserved global variable `gl_FragColor`.

3. This C-flavored language has built in *variables* (like `gl_FragColor`), *functions* and *types*. In this case we've just been introduced to `vec4` that stands for a four dimensional vector of floating point precision. Later we will see more types like `vec3` and `vec2` together with the popular: `float`, `int` and `bool`.

4. If we look closely to the `vec4` type we can infer that the four arguments respond to the RED, GREEN, BLUE and ALPHA channels. Also we can see that these values are *normalized*, which means they go from `0.0` to `1.0`. Later, we will learn how normalizing values makes it easier to *map* values between variables.

5. Another important *C feature* we can see in this example is the presence of preprocessor macros. Macros are part of a pre-compilation step. With them it is possible to `#define` global variables and do some basic conditional operation (with `#ifdef` and `#endif`). All the macro commands begin with a hashtag (`#`). Pre-compilation happens right before compiling and copies all the calls to `#defines` and check `#ifdef` (is defined) and `#ifndef` (is not defined) conditionals. In our "hello world!" example above, we only insert the line 2 if `GL_ES` is defined, which mostly happens when the code is compiled on mobile devices and browsers.

6. Float types are vital in shaders, so the level of *precision* is crucial. Lower precision means faster rendering, but at the cost of quality. You can be picky and specify the precision of each variable that uses floating point. In the second line (`precision mediump float;`) we are setting all floats to medium precision. But we can choose to set them to low (`precision lowp float;`) or high (`precision highp float;`).

7. The last, and maybe most important, detail is that GLSL specs don't guarantee that variables will be automatically casted. What does that mean? Manufacturers have different approaches to accelerate graphics card processes but they are forced to guarantee minimum specs. Automatic casting is not one of them. In our "hello world!" example `vec4` has floating point precision and for that it expects to be assigned with `floats`. If you want to make good consistent code and not spend hours debugging white screens, get used to putting the point (`.`) in your floats. This kind of code will not always work:

```
void main() {
    gl_FragColor = vec4(1,0,0,1);          // ERROR
}
```

Now that we've described the most relevant elements of our "hello world!" program, it's time to click on the code block and start challenging all that we've learned. You will note that on errors, the program will fail to compile, showing a white screen. There are some interesting things to try, for example:

- Try replacing the floats with integers, your graphic card may or may not tolerate this behavior.
- Try commenting out line 8 and not assigning any pixel value to the function.
- Try making a separate function that returns a specific color and use it inside `main()`. As a hint, here is the code for a function that returns a red color:

```
vec4 red(){
    return vec4(1.0,0.0,0.0,1.0);
}
```

- There are multiple ways of constructing `vec4` types, try to discover other ways. The following is one of them:

```
vec4 color = vec4(vec3(1.0,0.0,1.0),1.0);
```

Although this example isn't very exciting, it is the most basic example - we are changing all the pixels inside the canvas to the same exact color. In the following chapter we will see how to change the pixel colors by using two types of input: space (the place of the pixel on the screen) and time (the number of seconds since the page was loaded).

2.6 Uniforms

So far we have seen how the GPU manages large numbers of parallel threads, each one responsible for assigning the color to a fraction of the total image. Although each parallel thread is blind to the others, we need to be able to send some inputs from the CPU to all the threads. Because of the architecture of the graphics card those inputs are going to be equal (*uniform*) to all the threads and necessarily set as *read only*. In other words, each thread receives the same data which it can read but cannot change.

These inputs are called `uniform` and come in most of the supported types: `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, `mat4`, `sampler2D` and `samplerCube`. Uniforms are defined with the corresponding type at the top of the shader right after assigning the default floating point precision.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;  // Canvas size (width,height)
uniform vec2 u_mouse;       // mouse position in screen pixels
uniform float u_time;       // Time in seconds since load
```

You can picture the uniforms like little bridges between the CPU and the GPU. The names will vary from implementation to implementation but in this series of examples I'm always passing: `u_time` (time in seconds since the shader started), `u_resolution` (billboard size where the shader is being drawn) and `u_mouse` (mouse position inside the billboard in pixels). I'm following the convention of putting `u_` before the uniform name to be explicit about the nature of this variable but you will find all kinds of names for uniforms. For example ShaderToy.com uses the same uniforms but with the following names:

```
uniform vec3 iResolution;   // viewport resolution (in pixels)
uniform vec4 iMouse;        // mouse pixel coords. xy: current, zw: click
uniform float iTime;        // shader playback time (in seconds)
```

Enough talking, let's see the uniforms in action. In the following code we use `u_time` - the number of seconds since the shader started running - together with a sine function to animate the transition of the amount of red in the billboard.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform float u_time;

void main() {
	gl_FragColor = vec4(abs(sin(u_time)),0.0,0.0,1.0);
}
```

As you can see GLSL has more surprises. The GPU has hardware accelerated angle, trigonometric and exponential functions. Some of those functions are: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `pow()`, `exp()`, `log()`, `sqrt()`, `abs()`, `sign()`, `floor()`, `ceil()`, `fract()`, `mod()`, `min()`, `max()` and `clamp()`.

Now it is time again to play with the above code.

- Slow down the frequency until the color change becomes almost imperceptible.
- Speed it up until you see a single color without flickering.
- Play with the three channels (RGB) in different frequencies to get interesting patterns and behaviors.

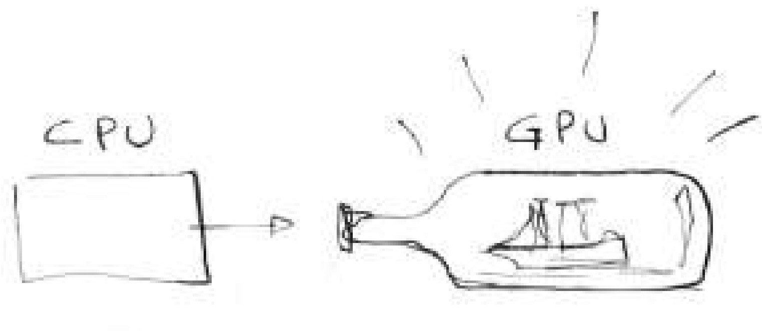
2.7 gl_FragCoord

In the same way GLSL gives us a default output, `vec4 gl_FragColor`, it also gives us a default input, `vec4 gl_FragCoord`, which holds the screen coordinates of the *pixel* or *screen fragment* that the active thread is working on. With `vec4 gl_FragCoord`, we know where a thread is working inside the billboard. In this case we don't call it `uniform` because it will be different from thread to thread, instead `gl_FragCoord` is called a *varying*.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
	vec2 st = gl_FragCoord.xy/u_resolution;
	gl_FragColor = vec4(st.x,st.y,0.0,1.0);
}
```

In the above code we *normalize* the coordinate of the fragment by dividing it by the total resolution of the billboard. By doing this the values will go between `0.0` and `1.0`, which makes it easy to map the X and Y values to the RED and GREEN channel.

In shader-land we don't have too many resources for debugging besides assigning strong colors to variables and trying to make sense of them. You will discover that sometimes coding in GLSL is very similar to putting ships inside bottles. Is equally hard, beautiful and gratifying.



Now it is time to try and challenge our understanding of this code.

- Can you tell where the coordinate `(0.0, 0.0)` is in our canvas?
- What about `(1.0, 0.0)`, `(0.0, 1.0)`, `(0.5, 0.5)` and `(1.0, 1.0)`?
- Can you figure out how to use `u_mouse` knowing that the values are in pixels and NOT normalized values? Can you use it to move colors around?
- Can you imagine an interesting way of changing this color pattern using `u_time` and `u_mouse` coordinates?

After doing these exercises you might wonder where else you can try your new shader-powers. In the following chapter we will see how to make your own shader tools in three.js, Processing, and openFrameworks.

2.8 Running your shader

As part of the construction of this book and my art practice I made an ecosystem of tools to create, display, share and curate shaders. These tools work consistently across Linux, MacOS, Windows and Raspberry Pi and browsers without the need of changing your code.

2.9 Running your shaders on the browser

Display: all live examples in this book are displayed using glslCanvas which makes the process of running standalone shader incredible easy.

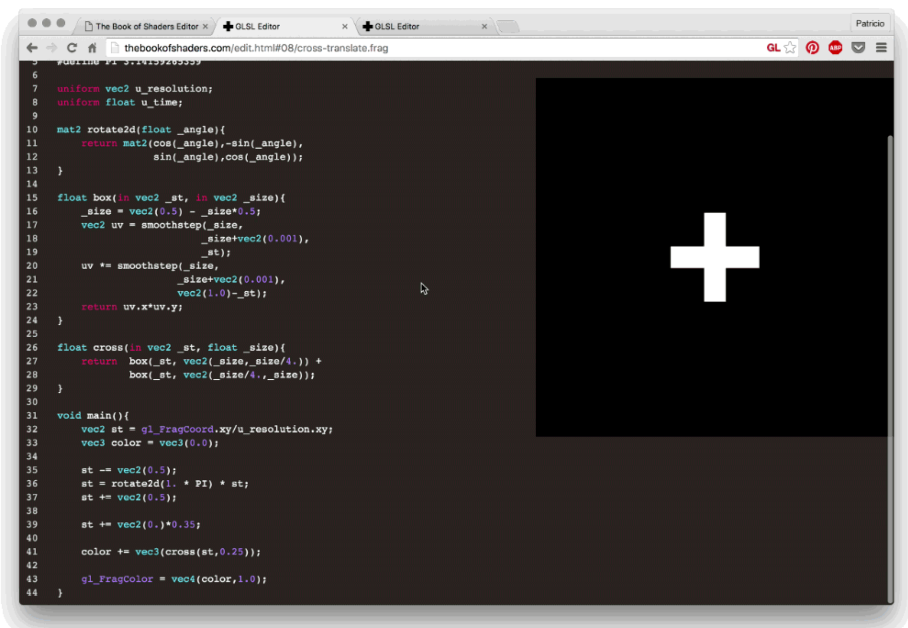
```
<canvas class="glslCanvas" data-fragment-url="yourShader.frag" data-textures="yourInputImage
```

As you can see, it just needs a `canvas` element with `class="glslCanvas"` and the url to your shader in the `data-fragment-url`. Learn more about it here.

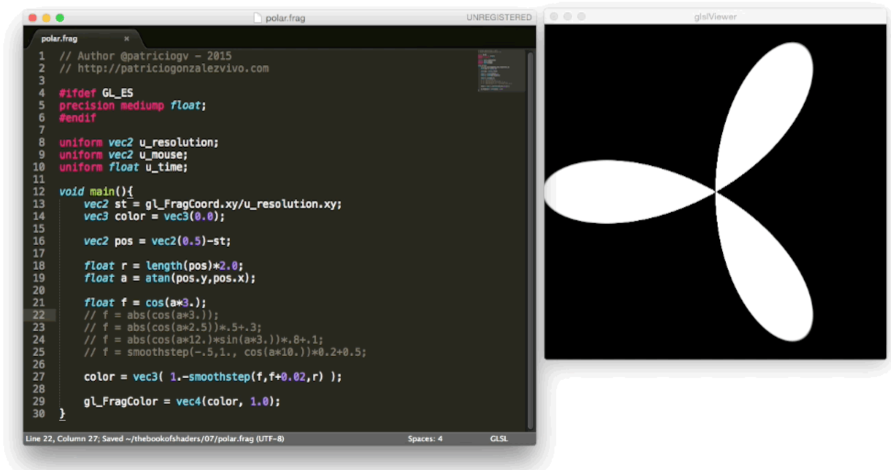
If you are like me, you will probably want to run shaders directly from the console, in that case you should check out glslViewer. This application allows you to incorporate shaders into your `bash` scripts or unix pipelines and use it in a similar way to ImageMagick. Also glslViewer is a great way to compile shaders on your Raspberry Pi, which is the reason openFrame.io uses it to display shader artwork. Learn more about this application here.

```
glslViewer yourShader.frag yourInputImage.png -w 500 -h 500 -s 1 -o yourOutputImage.png
```

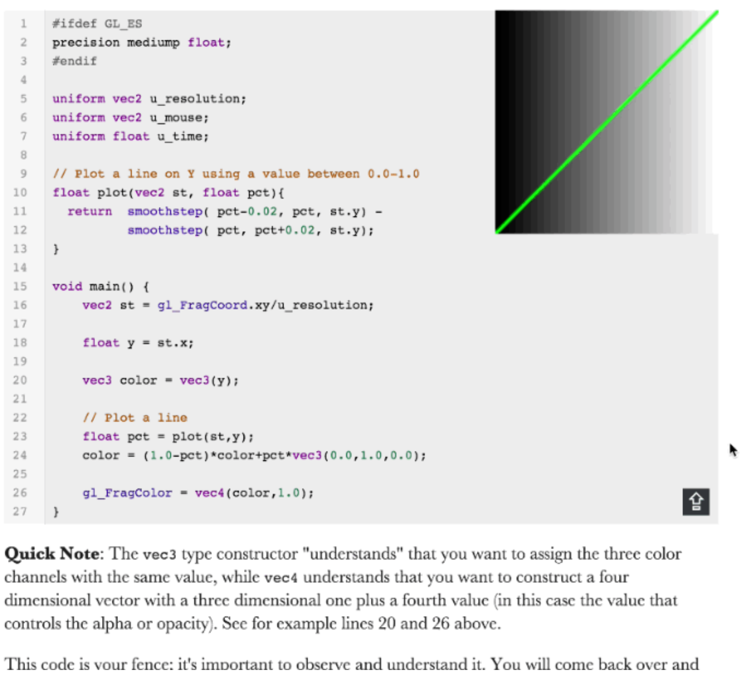
Create: in order to illuminate the experience of coding shaders I made an online editor called glslEditor. This editor is embedded on the book's live examples, it brings a series of handy widgets to make more tangible the abstract experience of working with glsl code. You can also run it as a standalone web application from editor.thebookofshaders.com/. Learn more about it here.



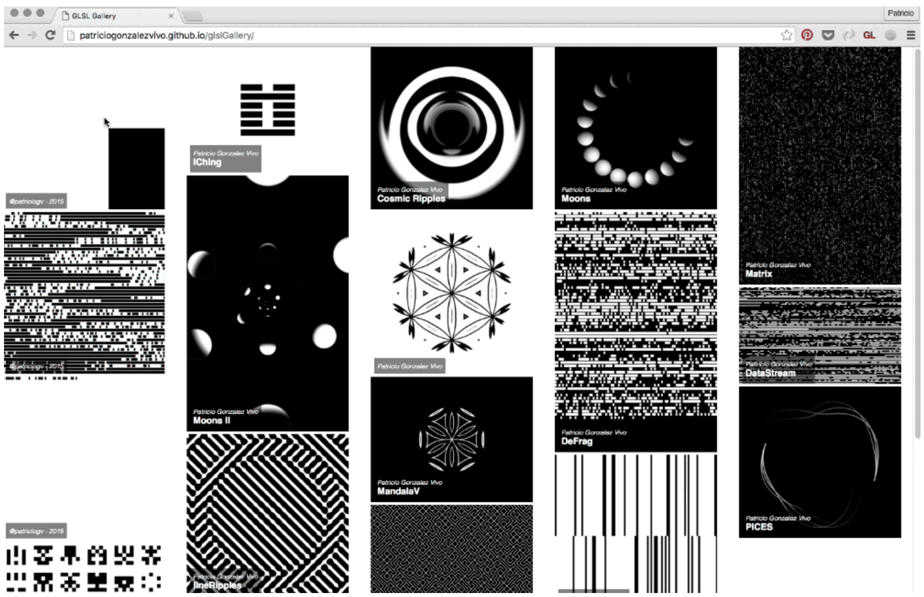
If you prefer to work offline using SublimeText you can install this package for glslViewer. Learn more about it here.



Share: the online editor (editor.thebookofshaders.com/) can share your shaders! Both the embedded and standalone version have an export button where you can get an unique URL's to your shader. Also it has the ability to export directly to an openFrame.io.



Curate: Sharing your code is the beginning of you sharing your shader as artwork! Beside the option to export to openFrame.io I made a tool to curate your shaders into a gallery that can be embedded on any site, it's name is glslGallery. Learn more here.



2.10 Running your shaders on your favorite framework

In case you already have experience programming in a framework like: Processing, Three.js or OpenFrameworks, you're probably excited to try shaders on these platforms you feel comfortable with. The following are examples of how to set shaders in some popular frameworks with the same uniforms that we are going to use throughout this book. (In the GitHub repository for this chapter, you'll find the full source code for these three frameworks.)

2.10.1 In Three.js

The brilliant and very humble Ricardo Cabello (aka MrDoob) has been developing along with other contributors probably one of the most famous frameworks for WebGL, called Three.js. You will find a lot of examples, tutorials and books that teach you how to use this JavaScript library to make cool 3D graphics.

Below is an example of the HTML and JS you need to get started with shaders in three.js. Pay attention to the `id="fragmentShader"` script, here is where you can copy the shaders you find in this book.

```
<body>
    <div id="container"></div>
    <script src="js/three.min.js"></script>
    <script id="vertexShader" type="x-shader/x-vertex">
        void main() {
            gl_Position = vec4( position, 1.0 );
```

```
    }
</script>
<script id="fragmentShader" type="x-shader/x-fragment">
    uniform vec2 u_resolution;
    uniform float u_time;

    void main() {
        vec2 st = gl_FragCoord.xy/u_resolution.xy;
        gl_FragColor=vec4(st.x,st.y,0.0,1.0);
    }
</script>
<script>
    var container;
    var camera, scene, renderer;
    var uniforms;

    init();
    animate();

    function init() {
        container = document.getElementById( 'container' );

        camera = new THREE.Camera();
        camera.position.z = 1;

        scene = new THREE.Scene();

        var geometry = new THREE.PlaneBufferGeometry( 2, 2 );

        uniforms = {
            u_time: { type: "f", value: 1.0 },
            u_resolution: { type: "v2", value: new THREE.Vector2() },
            u_mouse: { type: "v2", value: new THREE.Vector2() }
        };

        var material = new THREE.ShaderMaterial( {
            uniforms: uniforms,
            vertexShader: document.getElementById( 'vertexShader' ).textContent,
            fragmentShader: document.getElementById( 'fragmentShader' ).textContent
        } );

        var mesh = new THREE.Mesh( geometry, material );
        scene.add( mesh );

        renderer = new THREE.WebGLRenderer();
        renderer.setPixelRatio( window.devicePixelRatio );
```

```
            container.appendChild( renderer.domElement );

            onWindowResize();
            window.addEventListener( 'resize', onWindowResize, false );

            document.onmousemove = function(e){
              uniforms.u_mouse.value.x = e.pageX
              uniforms.u_mouse.value.y = e.pageY
            }
        }

        function onWindowResize( event ) {
            renderer.setSize( window.innerWidth, window.innerHeight );
            uniforms.u_resolution.value.x = renderer.domElement.width;
            uniforms.u_resolution.value.y = renderer.domElement.height;
        }

        function animate() {
            requestAnimationFrame( animate );
            render();
        }

        function render() {
            uniforms.u_time.value += 0.05;
            renderer.render( scene, camera );
        }
    </script>
</body>
```

2.10.2 In Processing

Started by Ben Fry and Casey Reas in 2001, Processing is an extraordinarily simple and powerful environment in which to take your first steps in code (it was for me at least). Andres Colubri has made important updates to the openGL and video in Processing, making it easier than ever to use and play with GLSL shaders in this friendly environment. Processing will search for the shader named `"shader.frag"` in the `data` folder of the sketch. Be sure to copy the examples you find here into that folder and rename the file.

```
PShader shader;

void setup() {
  size(640, 360, P2D);
  noStroke();
```

```
  shader = loadShader("shader.frag");
}

void draw() {
  shader.set("u_resolution", float(width), float(height));
  shader.set("u_mouse", float(mouseX), float(mouseY));
  shader.set("u_time", millis() / 1000.0);
  shader(shader);
  rect(0,0,width,height);
}
```

In order for the shader to work on versions previous to 2.1, you need to add the following line at the beginning of your shader: `#define PROCESSING_COLOR_SHADER`. So that it looks like this:

```
#ifdef GL_ES
precision mediump float;
#endif

#define PROCESSING_COLOR_SHADER

uniform vec2 u_resolution;
uniform vec3 u_mouse;
uniform float u_time;

void main() {
    vec2 st = gl_FragCoord.st/u_resolution;
    gl_FragColor = vec4(st.x,st.y,0.0,1.0);
}
```

For more information about shaders in Processing check out this tutorial.

2.10.3 In openFrameworks

Everybody has a place where they feel comfortable, in my case, that's still the openFrameworks community. This C++ framework wraps around OpenGL and other open source C++ libraries. In many ways it's very similar to Processing, but with the obvious complications of dealing with C++ compilers. In the same way as Processing, openFrameworks will search for your shader files in the data folder, so don't forget to copy the `.frag` files you want to use and change the name when you load them.

```
void ofApp::draw(){
    ofShader shader;
    shader.load("","shader.frag");

    shader.begin();
    shader.setUniform1f("u_time", ofGetElapsedTimef());
```

```
    shader.setUniform2f("u_resolution", ofGetWidth(), ofGetHeight());
    ofRect(0,0,ofGetWidth(), ofGetHeight());
    shader.end();
}
```

If you want to use the full set of uniforms contain on the specs of GlslViewer and GlslCanvas in a more simple way on OpenFrameworks I recomend using the ofxShader addon which will also have support for multiple buffers, material shaders, hotreload and automatic conversion for OpenGL ES in the Raspberry Pi. And your code will be as simple as doing

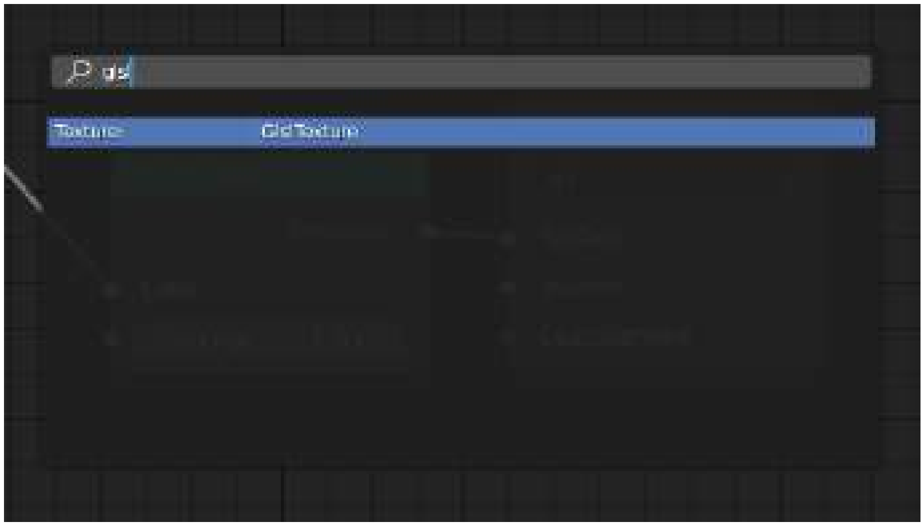
```
//--------------------------------------------------------------
void ofApp::setup(){
    ofDisableArbTex();

    sandbox.allocate(ofGetWidth(), ofGetHeight());
    sandbox.load("grayscott.frag");
}

//--------------------------------------------------------------
void ofApp::draw(){
    sandbox.render();
    sandbox.draw(0, 0);
}
```

For more information about shaders in openFrameworks go to this excellent tutorial made by Joshua Noble.

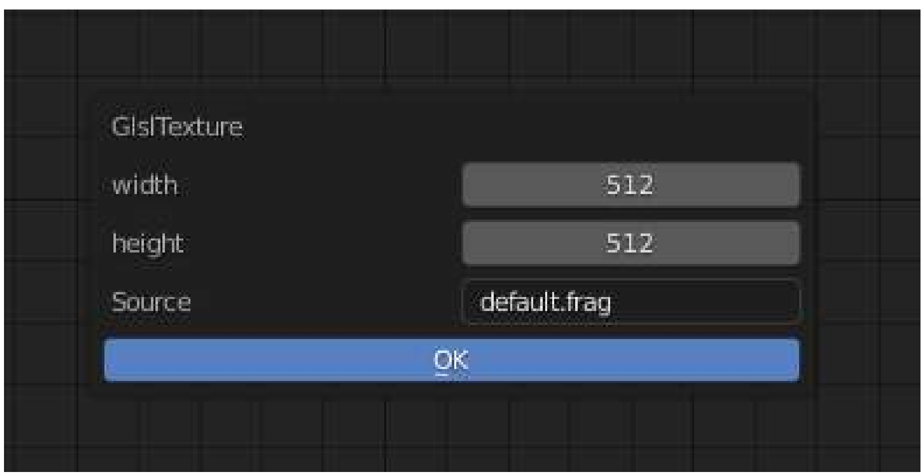
2.10.4 In Blender

GlslTexture is an addon that allows you to programatically generate textures using GLSL Shaders and is fully compatible with the rest of the sandboxes on this chapter. How it works:

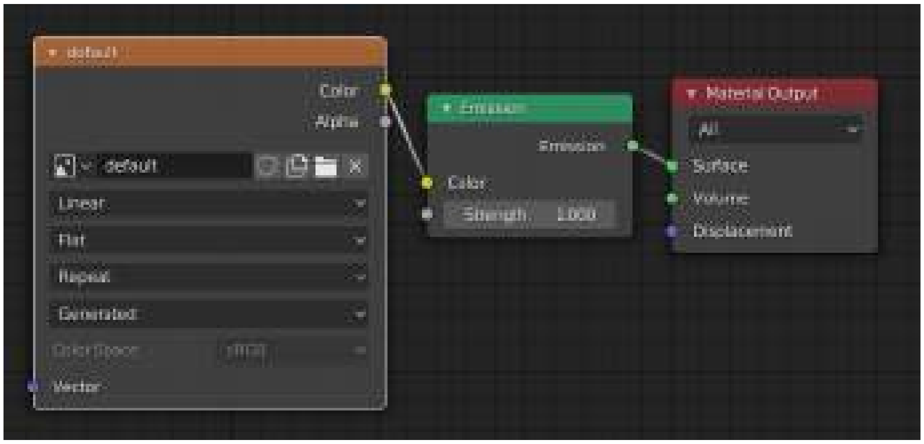
1. Operator Search: `F3` (or `SpaceBar` depending on your setup). Type `GlslTexture`



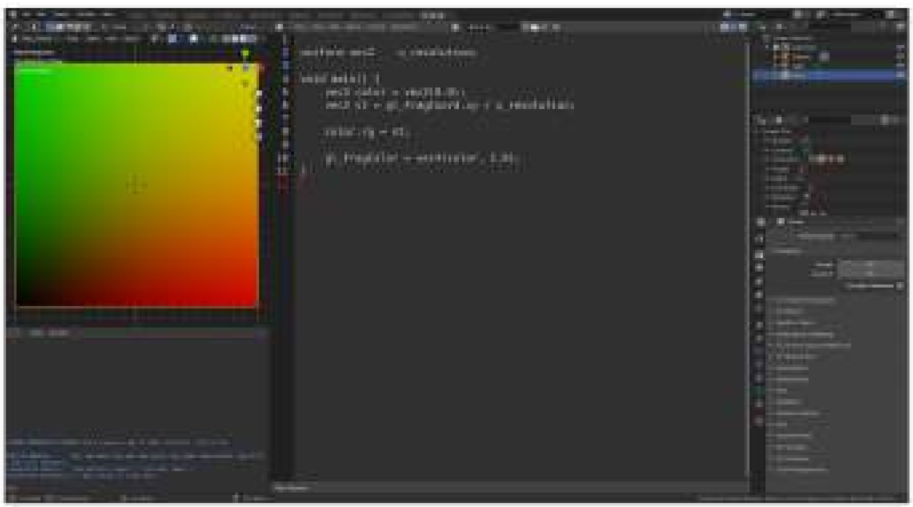
2. Change `width` and `height` size and `Source` file (which can be a path to an external file).



3. Use the Image on your materials. The Image name will be based on the name of the source file.



4. Go to the Text Editor (or an external editor if your source file is external) and edit the shader. It will hot reload.



3 Algorithmic drawing

3.1 Shaping functions

This chapter could be named "Mr. Miyagi's fence lesson." Previously, we mapped the normalized position of x and y to the *red* and *green* channels. Essentially we made a function that takes a two dimensional vector (x and y) and returns a four dimensional vector (r, g, b and a). But before we go further transforming data between dimensions we need to start simpler… much simpler. That means understanding how to make one dimensional functions. The more energy and time you spend learning and mastering this, the stronger your shader karate will be.

Figure 7: The Karate Kid (1984)

The following code structure is going to be our fence. In it, we visualize the normalized value of the x coordinate (`st.x`) in two ways: one with brightness (observe the nice gradient from black to white) and the other by plotting a green line on top (in that case the x value is assigned directly to y). Don't focus too much on the plot function, we will go through it in more detail in a moment.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Plot a line on Y using a value between 0.0-1.0
float plot(vec2 st) {
    return smoothstep(0.02, 0.0, abs(st.y - st.x));
}

void main() {
        vec2 st = gl_FragCoord.xy/u_resolution;

    float y = st.x;

    vec3 color = vec3(y);
```

```
    // Plot a line
    float pct = plot(st);
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

        gl_FragColor = vec4(color,1.0);
}
```

Quick Note: The `vec3` type constructor "understands" that you want to assign the three color channels with the same value, while `vec4` understands that you want to construct a four dimensional vector with a three dimensional one plus a fourth value (in this case the value that controls the alpha or opacity). See for example lines 19 and 25 above.

This code is your fence; it's important to observe and understand it. You will come back over and over to this space between *0.0* and *1.0*. You will master the art of blending and shaping this line.

This one-to-one relationship between x and y (or the brightness) is known as *linear interpolation*. From here we can use some mathematical functions to *shape* the line. For example we can raise x to the power of 5 to make a *curved* line.

```
// Author: Inigo Quiles
// Title: Expo

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float plot(vec2 st, float pct){
  return  smoothstep( pct-0.02, pct, st.y) -
          smoothstep( pct, pct+0.02, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;

    float y = pow(st.x,5.0);

    vec3 color = vec3(y);
```

```
    float pct = plot(st,y);
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

Interesting, right? On line 22 try different exponents: 20.0, 2.0, 1.0, 0.0, 0.2 and 0.02 for example. Understanding this relationship between the value and the exponent will be very helpful. Using these types of mathematical functions here and there will give you expressive control over your code, a sort of data acupuncture that let you control the flow of values.

`pow()` is a native function in GLSL and there are many others. Most of them are accelerated at the level of the hardware, which means if they are used in the right way and with discretion they will make your code faster.

Replace the power function on line 22. Try other ones like: `exp()`, `log()` and `sqrt()`. Some of these functions are more interesting when you play with them using PI. You can see on line 8 that I have defined a macro that will replace any use of `PI` with the value `3.14159265359`.

3.1.1 Step and Smoothstep

GLSL also has some unique native interpolation functions that are hardware accelerated.

The `step()` interpolation receives two parameters. The first one is the limit or threshold, while the second one is the value we want to check or pass. Any value under the limit will return `0.0` while everything above the limit will return `1.0`.

Try changing this threshold value on line 20 of the following code.

```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform float u_time;

float plot(vec2 st, float pct){
  return  smoothstep( pct-0.02, pct, st.y) -
          smoothstep( pct, pct+0.02, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;
```

```
    // Step will return 0.0 unless the value is over 0.5,
    // in that case it will return 1.0
    float y = step(0.5,st.x);

    vec3 color = vec3(y);

    float pct = plot(st,y);
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

The other unique function is known as `smoothstep()`. Given a range of two numbers and a value, this function will interpolate the value between the defined range. The two first parameters are for the beginning and end of the transition, while the third is for the value to interpolate.

```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float plot(vec2 st, float pct){
  return  smoothstep( pct-0.02, pct, st.y) -
          smoothstep( pct, pct+0.02, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution;

    // Smooth interpolation between 0.1 and 0.9
    float y = smoothstep(0.1,0.9,st.x);

    vec3 color = vec3(y);

    float pct = plot(st,y);
    color = (1.0-pct)*color+pct*vec3(0.0,1.0,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

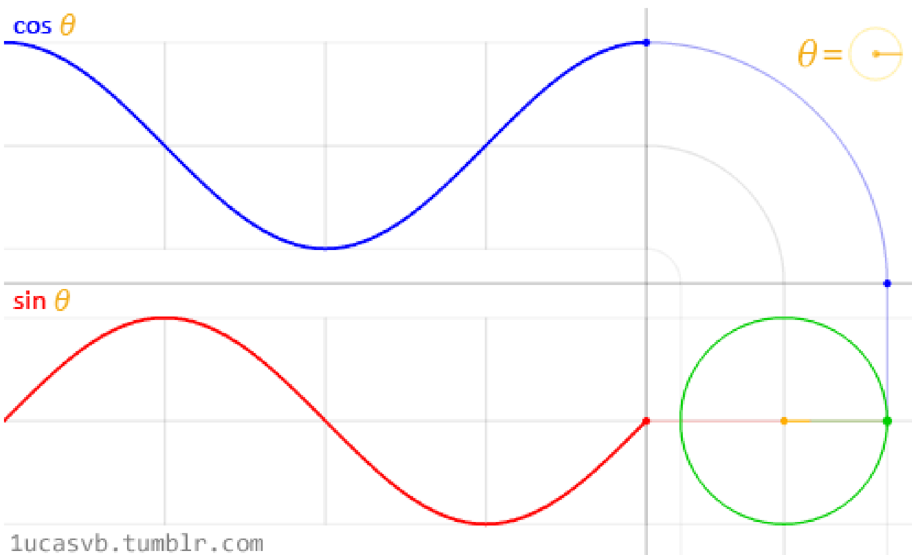
In the previous example, on line 12, notice that we've been using smoothstep to draw the green line on the `plot()` function. For each position along the x axis this function makes a *bump* at a particular value of y. How? By connecting two `smoothstep()` together. Take a look at the following function, replace it for line 20 above and think of it as a vertical cut. The background does look like a line, right?

```
float y = smoothstep(0.2,0.5,st.x) - smoothstep(0.5,0.8,st.x);
```

3.1.2 Sine and Cosine

When you want to use some math to animate, shape or blend values, there is nothing better than being friends with sine and cosine.

These two basic trigonometric functions work together to construct circles that are as handy as MacGyver's Swiss army knife. It's important to know how they behave and in what ways they can be combined. In a nutshell, given an angle (in radians) they will return the correct position of x (cosine) and y (sine) of a point on the edge of a circle with a radius equal to 1. But, the fact that they return normalized values (values between -1 and 1) in such a smooth way makes them an incredible tool.



While it's difficult to describe all the relationships between trigonometric functions and circles, the above animation does a beautiful job of visually summarizing these relationships.

Take a careful look at this sine wave. Note how the y values flow smoothly between +1 and -1. As we saw in the time example in the previous chapter, you can use this rhythmic behavior of `sin()` to animate properties. If you are reading this example in a browser you will see that you can change the code in the formula above to watch how the wave changes. (Note: don't forget the semicolon at the end of the lines.)

Try the following exercises and notice what happens:

- Add time (`u_time`) to x before computing the `sin`. Internalize that **motion** along x.
- Multiply x by `PI` before computing the `sin`. Note how the two phases **shrink** so each cycle repeats every 2 integers.
- Multiply time (`u_time`) by x before computing the `sin`. See how the **frequency** between phases becomes more and more compressed. Note that u_time may have already become very large, making the graph hard to read.
- Add 1.0 to `sin(x)`. See how all the wave is **displaced** up and now all values are between 0.0 and 2.0.
- Multiply `sin(x)` by 2.0. See how the **amplitude** doubles in size.
- Compute the absolute value (`abs()`) of `sin(x)`. It looks like the trace of a **bouncing** ball.
- Extract just the fraction part (`fract()`) of the resultant of `sin(x)`.
- Add the higher integer (`ceil()`) and the smaller integer (`floor()`) of the resultant of `sin(x)` to get a digital wave of 1 and -1 values.

3.1.3 Some extra useful functions

At the end of the last exercise we introduced some new functions. Now it's time to experiment with each one by uncommenting the lines below one at a time. Get to know these functions and study how they behave. I know, you are wondering… why? A quick google search on "generative art" will tell you. Keep in mind that these functions are our fence. We are mastering the movement in one dimension, up and down. Soon, it will be time for two, three and four dimensions!

3.1.4 Advance shaping functions

Golan Levin has great documentation of more complex shaping functions that are extraordinarily helpful. Porting them to GLSL is a really smart move, to start building your own resource of snippets of code.

- Polynomial Shaping Functions: www.flong.com/archive/texts/code/shapers_poly
- Exponential Shaping Functions: www.flong.com/archive/texts/code/shapers_exp
- Circular & Elliptical Shaping Functions: www.flong.com/archive/texts/code/shapers_circ
- Bezier and Other Parametric Shaping Functions: www.flong.com/archive/texts/code/shapers_bez

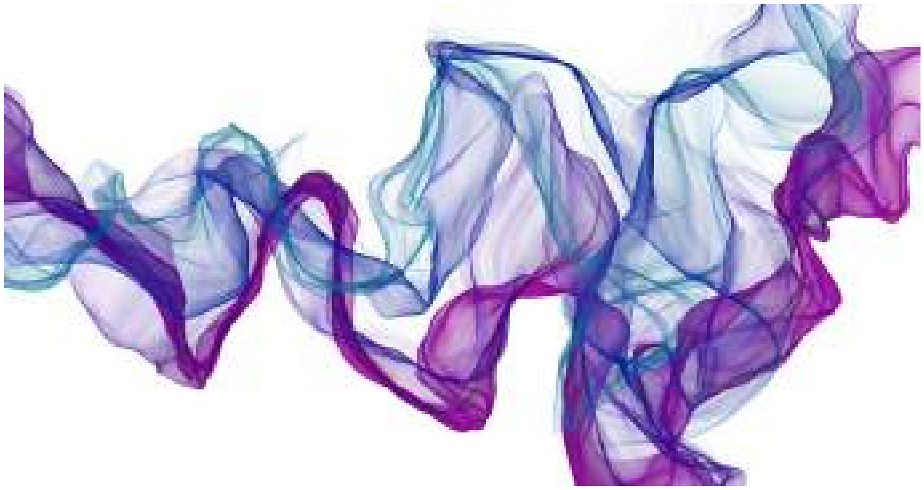


Figure 8: Anthony Mattox (2009)

Like chefs that collect spices and exotic ingredients, digital artists and creative coders have a particular love of working on their own shaping functions.

Iñigo Quiles has a great collection of useful functions. After reading this article take a look at the following translation of these functions to GLSL. Pay attention to the small changes required, like putting the "." (dot) on floating point numbers and using the GLSL name for *C functions*; for example instead of `powf()` use `pow()`:

To keep your motivation up, here is an elegant example (made by Danguafer) of mastering the shaping-functions karate.

In the *Next >>* chapter we will start using our new moves. First with mixing colors and then drawing shapes.

3.1.4.1 Exercise

Take a look at the following table of equations made by Kynd. See how he is combining functions and their properties to control the values between 0.0 and 1.0. Now it's time for you to practice by replicating these functions. Remember the more you practice the better your karate will be.

3.1.4.2 For your toolbox

Here are some tools that will make it easier for you to visualize these types of functions.

- Grapher: if you have a MacOS computer, type `grapher` in your spotlight and you'll be able to use this super handy tool.

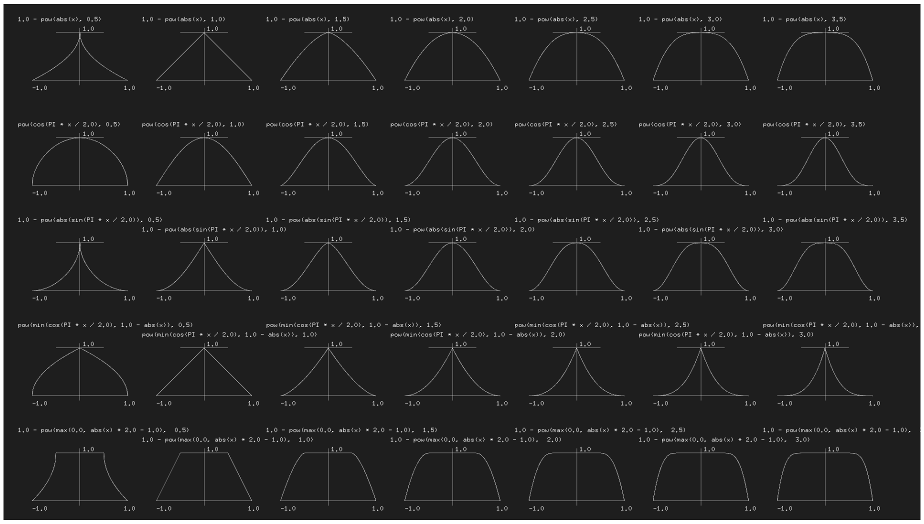


Figure 9: Kynd - www.flickr.com/photos/kynd/9546075099/ (2013)

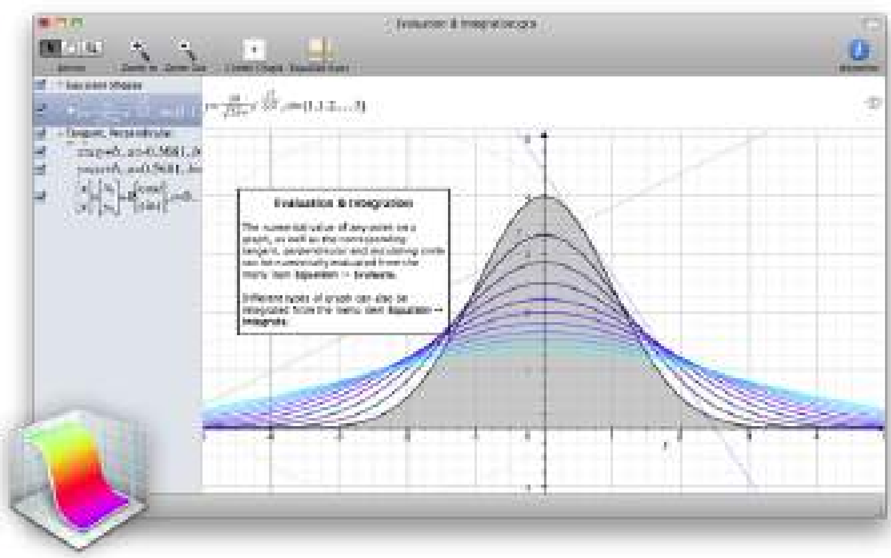


Figure 10: OS X Grapher (2004)

- GraphToy: once again Iñigo Quilez made a tool to visualize GLSL functions in WebGL.

Figure 11: Iñigo Quilez - GraphToy (2010)

- Shadershop: this amazing tool created by Toby Schachman will teach you how to construct complex functions in an incredible visual and intuitive way.

3.2 Colors

We haven't much of a chance to talk about GLSL vector types. Before going further it's important to learn more about these variables and the subject of colors is a great way to find out more about them.

If you are familiar with object oriented programming paradigms you've probably noticed that we have been accessing the data inside the vectors like any regular C-like `struct`.

```
vec3 red = vec3(1.0,0.0,0.0);
red.x = 1.0;
red.y = 0.0;
red.z = 0.0;
```

Defining color using an x, y and z notation can be confusing and misleading, right? That's why there are other ways to access this same information, but with different names. The values of `.x`, `.y` and `.z` can also be called `.r`, `.g` and `.b`, and `.s`, `.t` and `.p`. (`.s`, `.t` and `.p` are usually used for spatial coordinates of a texture, which we'll see in a later chapter.) You can also access the data in a vector by using the index position, `[0]`, `[1]` and `[2]`.

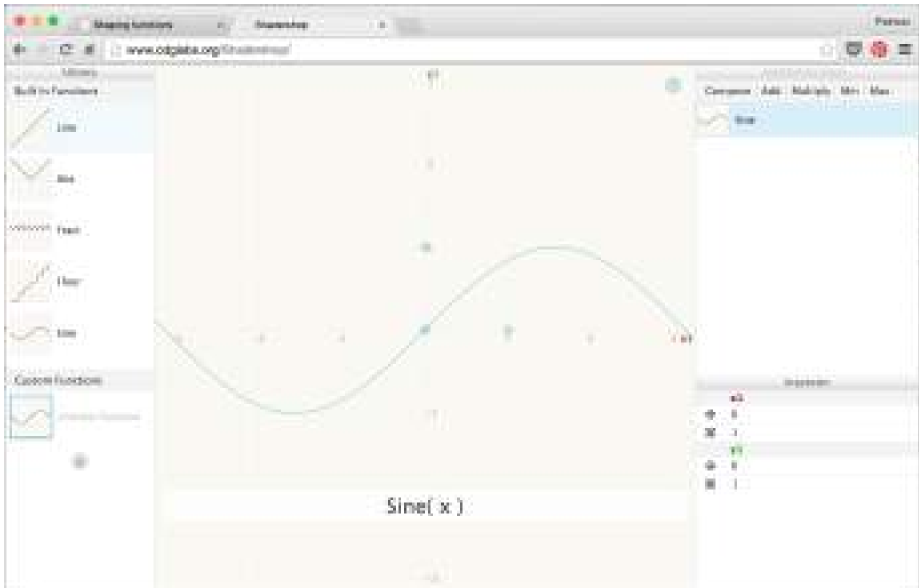


Figure 12: Toby Schachman - Shadershop (2014)

The following lines show all the ways to access the same data:

```
vec4 vector;
vector[0] = vector.r = vector.x = vector.s;
vector[1] = vector.g = vector.y = vector.t;
vector[2] = vector.b = vector.z = vector.p;
vector[3] = vector.a = vector.w = vector.q;
```

These different ways of pointing to the variables inside a vector are just nomenclatures designed to help you write clear code. This flexibility embedded in shading language is a door for you to start thinking interchangably about color and space coordinates.

Another great feature of vector types in GLSL is that the properties can be combined in any order you want, which makes it easy to cast and mix values. This ability is called *swizzle*.

```
vec3 yellow, magenta, green;

// Making Yellow
yellow.rg = vec2(1.0);  // Assigning 1. to red and green channels
yellow[2] = 0.0;        // Assigning 0. to blue channel

// Making Magenta
magenta = yellow.rbg;   // Assign the channels with green and blue swapped
```

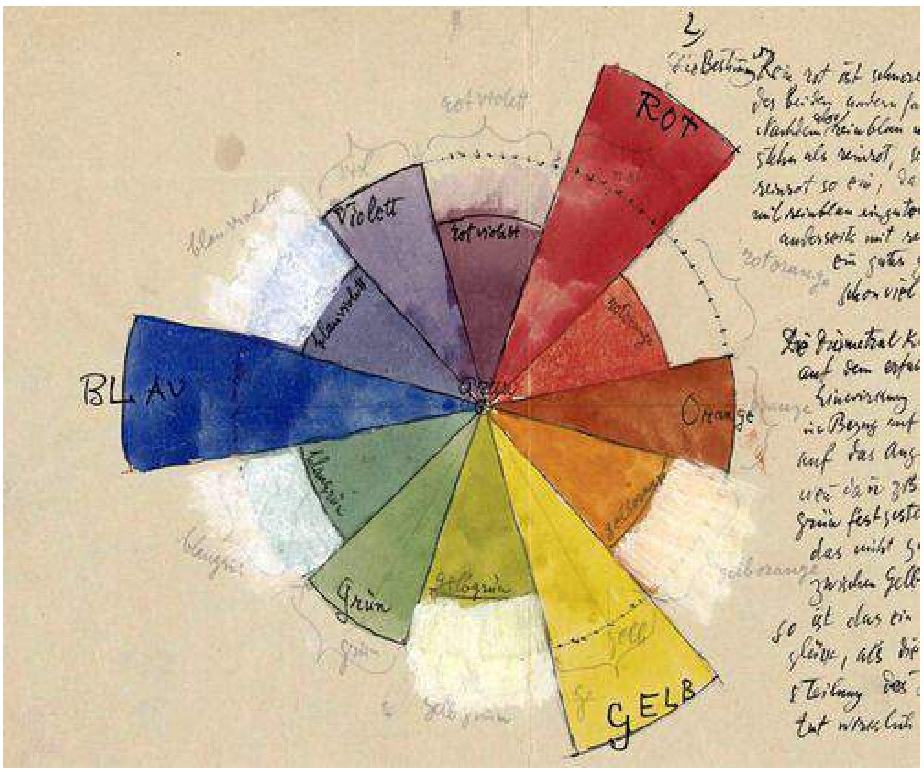


Figure 13: Paul Klee - Color Chart (1931)

```glsl
// Making Green
green.rgb = yellow.bgb; // Assign the blue channel of Yellow (0) to red and blue channels
```

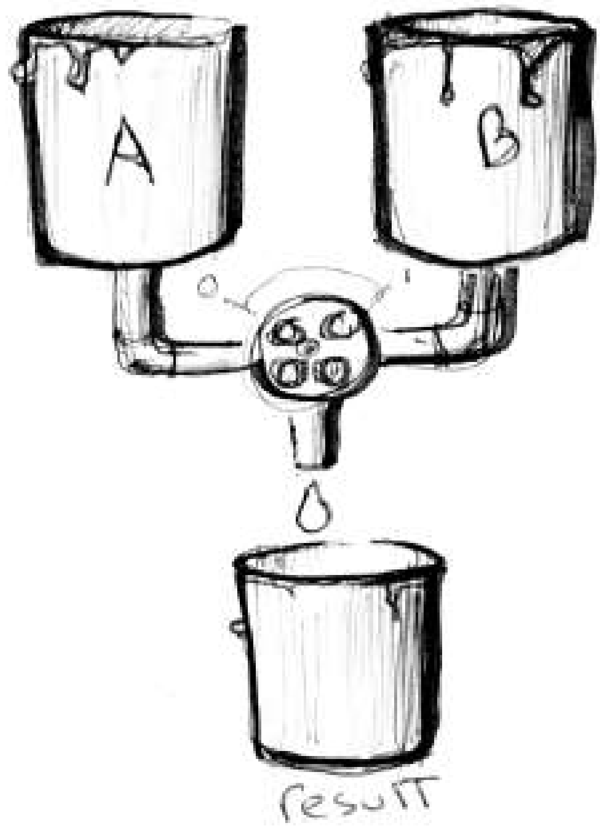
3.2.0.1 For your toolbox

You might not be used to picking colors with numbers - it can be very counter-intuitive. Lucky for you, there are a lot of smart programs that make this job easy. Find one that fits your needs and then train it to deliver colors in `vec3` or `vec4` format. For example, here are the templates I use on Spectrum:

```
vec3({{rn}},{{gn}},{{bn}})
vec4({{rn}},{{gn}},{{bn}},1.0)
```

3.2.1 Mixing color

Now that you know how colors are defined, it's time to integrate this with our previous knowledge. In GLSL there is a very useful function, `mix()`, that lets you mix two values in percentages. Can you guess what the percentage range is? Yes, values between 0.0 and 1.0! Which is perfect for you, after those long hours practicing your karate moves with the fence - it is time to use them!



Check the following code at line 18 and see how we are using the absolute values of a sin wave over time to mix `colorA` and `colorB`.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

vec3 colorA = vec3(0.149,0.141,0.912);
vec3 colorB = vec3(1.000,0.833,0.224);

void main() {
    vec3 color = vec3(0.0);

    float pct = abs(sin(u_time));
```

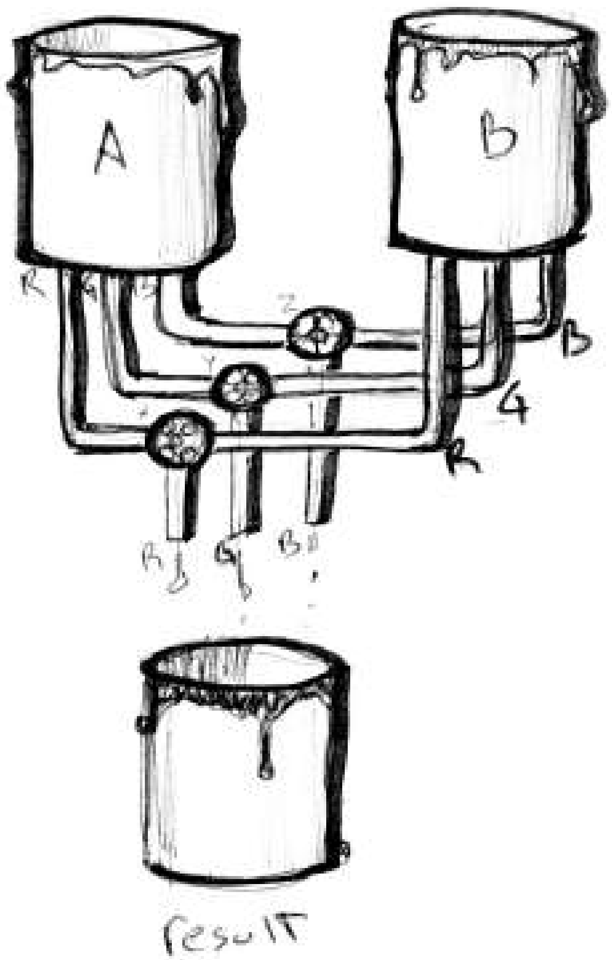
```
    // Mix uses pct (a value from 0-1) to
    // mix the two colors
    color = mix(colorA, colorB, pct);

    gl_FragColor = vec4(color,1.0);
}
```

Show off your skills by:

- Make an expressive transition between colors. Think of a particular emotion. What color seems most representative of it? How does it appear? How does it fade away? Think of another emotion and the matching color for it. Change the beginning and ending color of the above code to match those emotions. Then animate the transition using shaping functions. Robert Penner developed a series of popular shaping functions for computer animation known as easing functions, you can use this example as research and inspiration but the best result will come from making your own transitions.

3.2.2 Playing with gradients

The `mix()` function has more to offer. Instead of a single `float`, we can pass a variable type that matches the two first arguments, in our case a `vec3`. By doing that we gain control over the mixing percentages of each individual color channel, `r`, `g` and `b`.



Take a look at the following example. Like the examples in the previous chapter, we are hooking the transition to the normalized x coordinate and visualizing it with a line. Right now all the channels go along the same line.

Now, uncomment line number 25 and watch what happens. Then try uncommenting lines 26 and 27. Remember that the lines visualize the amount of `colorA` and `colorB` to mix per channel.

```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
```

```
uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

vec3 colorA = vec3(0.149,0.141,0.912);
vec3 colorB = vec3(1.000,0.833,0.224);

float plot (vec2 st, float pct){
  return  smoothstep( pct-0.01, pct, st.y) -
          smoothstep( pct, pct+0.01, st.y);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    vec3 pct = vec3(st.x);

    // pct.r = smoothstep(0.0,1.0, st.x);
    // pct.g = sin(st.x*PI);
    // pct.b = pow(st.x,0.5);

    color = mix(colorA, colorB, pct);

    // Plot transition lines for each channel
    color = mix(color,vec3(1.0,0.0,0.0),plot(st,pct.r));
    color = mix(color,vec3(0.0,1.0,0.0),plot(st,pct.g));
    color = mix(color,vec3(0.0,0.0,1.0),plot(st,pct.b));

    gl_FragColor = vec4(color,1.0);
}
```

You probably recognize the three shaping functions we are using on lines 25 to 27. Play with them! It's time for you to explore and show off your skills from the previous chapter and make interesting gradients. Try the following exercises:

- Compose a gradient that resembles a William Turner sunset
- Animate a transition between a sunrise and sunset using `u_time`.
- Can you make a rainbow using what we have learned so far?
- Use the `step()` function to create a colorful flag.

Figure 14: William Turner - The Fighting Temeraire (1838)

3.2.3 HSB

We can't talk about color without speaking about color space. As you probably know there are different ways to organize color besides by red, green and blue channels.

HSB stands for Hue, Saturation and Brightness (or Value) and is a more intuitive and useful organization of colors. Take a moment to read the `rgb2hsv()` and `hsv2rgb()` functions in the following code.

By mapping the position on the x axis to the Hue and the position on the y axis to the Brightness, we obtain a nice spectrum of visible colors. This spatial distribution of color can be very handy; it's more intuitive to pick a color with HSB than with RGB.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

vec3 rgb2hsb( in vec3 c ){
    vec4 K = vec4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz),
                 vec4(c.gb, K.xy),
                 step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r),
                 vec4(c.r, p.yzx),
                 step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)),
                d / (q.x + e),
                q.x);
}

//  Function from Iñigo Quiles
//  https://www.shadertoy.com/view/MsS3Wc
vec3 hsb2rgb( in vec3 c ){
    vec3 rgb = clamp(abs(mod(c.x*6.0+vec3(0.0,4.0,2.0),
                             6.0)-3.0)-1.0,
                     0.0,
                     1.0 );
    rgb = rgb*rgb*(3.0-2.0*rgb);
    return c.z * mix(vec3(1.0), rgb, c.y);
}
```

```
void main(){
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    // We map x (0.0 - 1.0) to the hue (0.0 - 1.0)
    // And the y (0.0 - 1.0) to the brightness
    color = hsb2rgb(vec3(st.x,1.0,st.y));

    gl_FragColor = vec4(color,1.0);
}
```

3.2.4 HSB in polar coordinates

HSB was originally designed to be represented in polar coordinates (based on the angle and radius) instead of cartesian coordinates (based on x and y). To map our HSB function to polar coordinates we need to obtain the angle and distance from the center of the billboard to the pixel coordinate. For that we will use the `length()` function and `atan(y,x)` (which is the GLSL version of the commonly used `atan2(y,x)`).

When using vector and trigonometric functions, `vec2`, `vec3` and `vec4` are treated as vectors even when they represent colors. We will start treating colors and vectors similarly, in fact you will come to find this conceptual flexibility very empowering.

Note: If you were wondering, there are more geometric functions besides `length` like: `distance()`, `dot()`, `cross`, `normalize()`, `faceforward()`, `reflect()` and `refract()`. Also GLSL has special vector relational functions such as: `lessThan()`, `lessThanEqual()`, `greaterThan()`, `greaterThanEqual()`, `equal()` and `notEqual()`.

Once we obtain the angle and length we need to "normalize" their values to the range between 0.0 to 1.0. On line 27, `atan(y,x)` will return an angle in radians between -PI and PI (-3.14 to 3.14), so we need to divide this number by `TWO_PI` (defined at the top of the code) to get values between -0.5 to 0.5, which by simple addition we change to the desired range of 0.0 to 1.0. The radius will return a maximum of 0.5 (because we are calculating the distance from the center of the viewport) so we need to double this range (by multiplying by two) to get a maximum of 1.0.

As you can see, our game here is all about transforming and mapping ranges to the 0.0 to 1.0 that we like.

```
#ifdef GL_ES
precision mediump float;
#endif
```

```glsl
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform float u_time;

//  Function from Iñigo Quiles
//  https://www.shadertoy.com/view/MsS3Wc
vec3 hsb2rgb( in vec3 c ){
    vec3 rgb = clamp(abs(mod(c.x*6.0+vec3(0.0,4.0,2.0),
                             6.0)-3.0)-1.0,
                     0.0,
                     1.0 );
    rgb = rgb*rgb*(3.0-2.0*rgb);
    return c.z * mix( vec3(1.0), rgb, c.y);
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    // Use polar coordinates instead of cartesian
    vec2 toCenter = vec2(0.5)-st;
    float angle = atan(toCenter.y,toCenter.x);
    float radius = length(toCenter)*2.0;

    // Map the angle (-PI to PI) to the Hue (from 0 to 1)
    // and the Saturation to the radius
    color = hsb2rgb(vec3((angle/TWO_PI)+0.5,radius,1.0));

    gl_FragColor = vec4(color,1.0);
}
```

Try the following exercises:

- Modify the polar example to get a spinning color wheel, just like the waiting mouse icon.
- Use a shaping function together with the conversion function from HSB to RGB to expand a particular hue value and shrink the rest.
- If you look closely at the color wheel used on color pickers (see the image below), they use a different spectrum according to RYB color space. For example, the opposite color of red should be green, but in our example it is cyan. Can you find a way to fix that in order to look exactly like the following image? [Hint: this is a great moment to use shaping functions.]

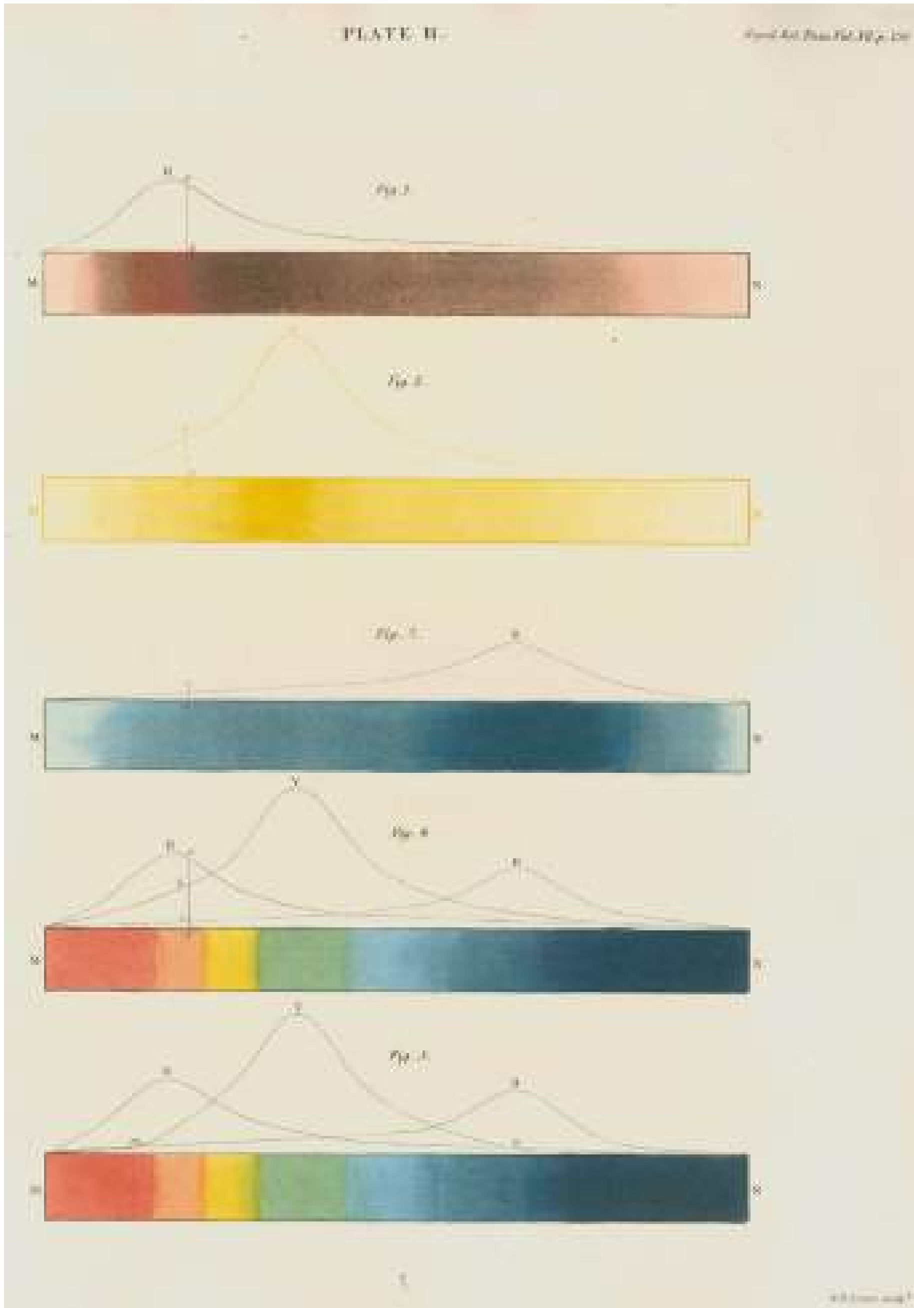


Figure 15: William Home Lizars - Red, blue and yellow spectra, with the solar spectrum (1834)

- Read Josef Albers' book Interaction of Color and use the following shaders examples as practice.

3.2.4.1 Note about functions and arguments

Before jumping to the next chapter let's stop and rewind. Go back and take look at the functions in previous examples. You will notice `in` before the type of the arguments. This is a *qualifier* and in this case it specifies that the variable is read only. In future examples we will see that it is also possible to define arguments as `out` or `inout`. This last one, `inout`, is conceptually similar to passing an argument by reference which will give us the possibility to modify a passed variable.

```
int newFunction(in vec4 aVec4,      // read-only
                out vec3 aVec3,     // write-only
                inout int aInt);    // read-write
```

You may not believe it but now we have all the elements to make cool drawings. In the next chapter we will learn how to combine all our tricks to make geometric forms by *blending* the space. Yep... *blending* the space.

3.3 Shapes

Finally! We have been building skills for this moment! You have learned most of the GLSL foundations, types and functions. You have practiced your shaping equations over and over. Now is the time to put it all together. You are up for this challenge! In this chapter you'll learn how to draw simple shapes in a parallel procedural way.

3.3.1 Rectangle

Imagine we have grid paper like we used in math classes and our homework is to draw a square. The paper size is 10x10 and the square is supposed to be 8x8. What will you do?

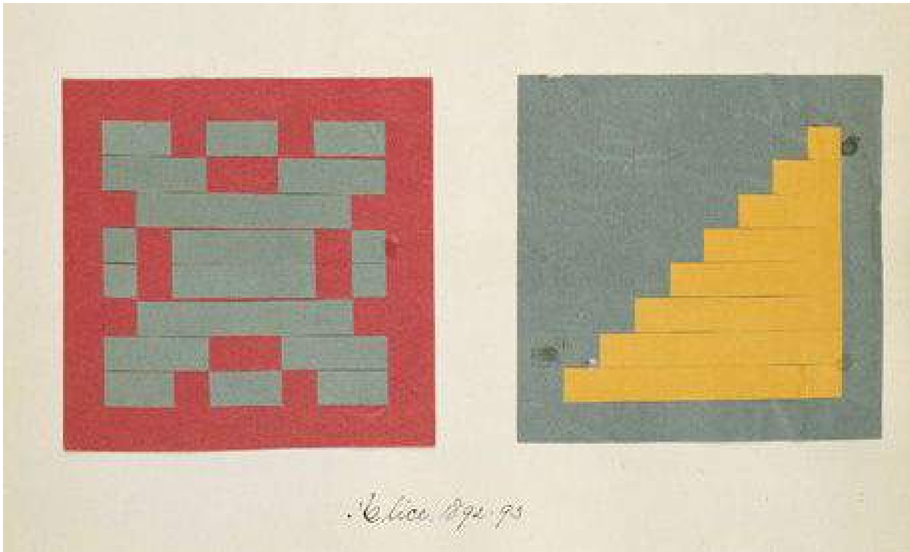


Figure 16: Alice Hubbard, Providence, United States, ca. 1892. Photo: Zindman/Freemont.

You'd paint everything except the first and last rows and the first and last column, right?

How does this relate to shaders? Each little square of our grid paper is a thread (a pixel). Each little square knows its position, like the coordinates of a chess board. In previous chapters we mapped *x* and *y* to the *red* and *green* color channels, and we learned how to use the narrow two dimensional territory between 0.0 and 1.0. How can we use this to draw a centered square in the middle of our billboard?

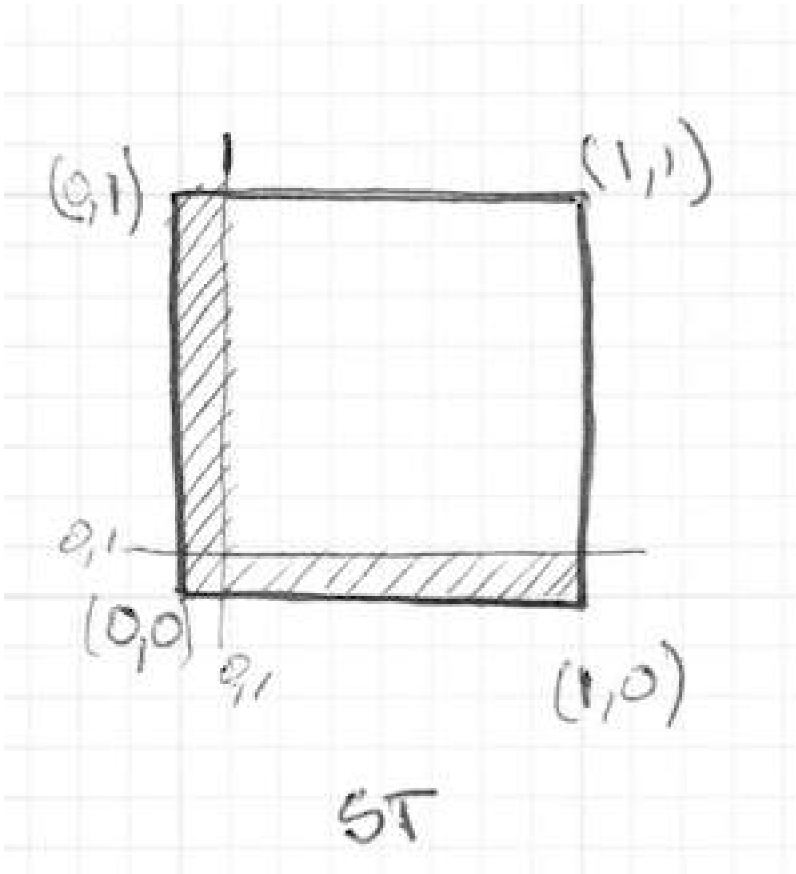
Let's start by sketching pseudocode that uses `if` statements over the spatial field. The principles to do this are remarkably similar to how we think of the grid paper scenario.

```
if ( (X GREATER THAN 1) AND (Y GREATER THAN 1) )
    paint white
else
    paint black
```

Now that we have a better idea of how this will work, let's replace the `if` statement with `step()`, and instead of using 10x10 let's use normalized values between 0.0 and 1.0:

```
uniform vec2 u_resolution;

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // Each result will return 1.0 (white) or 0.0 (black).
    float left = step(0.1,st.x);   // Similar to ( X greater than 0.1 )
    float bottom = step(0.1,st.y); // Similar to ( Y greater than 0.1 )

    // The multiplication of left*bottom will be similar to the logical AND.
    color = vec3( left * bottom );

    gl_FragColor = vec4(color,1.0);
}
```

The `step()` function will turn every pixel below 0.1 to black (`vec3(0.0)`) and the rest to white (`vec3(1.0)`) . The multiplication between `left` and `bottom` works as a logical `AND` operation, where both must be 1.0 to return 1.0 . This draws two black lines, one on the bottom and the other on the left side of the canvas.



In the previous code we repeat the structure for each axis (left and bottom). We can save some lines of code by passing two values directly to `step()` instead of one. That looks like this:

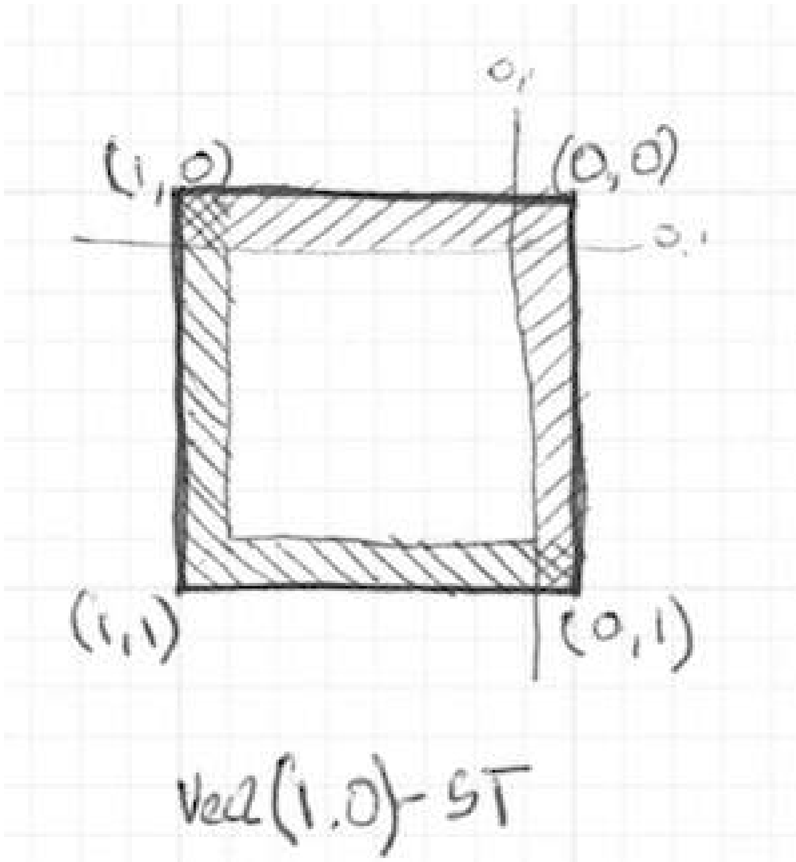
```
vec2 borders = step(vec2(0.1),st);
float pct = borders.x * borders.y;
```

So far, we've only drawn two borders (bottom-left) of our rectangle. Let's do the other two (top-right). Check out the following code:

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
```

```
uniform float u_time;

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // bottom-left
    vec2 bl = step(vec2(0.1),st);
    float pct = bl.x * bl.y;

    // top-right
    // vec2 tr = step(vec2(0.1),1.0-st);
    // pct *= tr.x * tr.y;

    color = vec3(pct);

    gl_FragColor = vec4(color,1.0);
}
```

Uncomment *lines 21-22* and see how we invert the `st` coordinates and repeat the same `step()` function. That way the `vec2(0.0,0.0)` will be in the top right corner. This is the digital equivalent of flipping the page and repeating the previous procedure.



Take note that in *lines 18 and 22* all of the sides are being multiplied together. This is equivalent to writing:

```
vec2 bl = step(vec2(0.1),st);       // bottom-left
vec2 tr = step(vec2(0.1),1.0-st);   // top-right
color = vec3(bl.x * bl.y * tr.x * tr.y);
```

Interesting right? This technique is all about using `step()` and multiplication for logical operations and flipping the coordinates.

Before going forward, try the following exercises:

- Change the size and proportions of the rectangle.
- Experiment with the same code but using `smoothstep()` instead of `step()`. Note that by changing values, you can go from blurred edges to elegant smooth borders.
- Do another implementation that uses `floor()`.
- Choose the implementation you like the most and make a function of it

that you can reuse in the future. Make your function flexible and efficient.

- Make another function that just draws the outline of a rectangle.
- How do you think you can move and place different rectangles in the same billboard? If you figure out how, show off your skills by making a composition of rectangles and colors that resembles a Piet Mondrian painting.

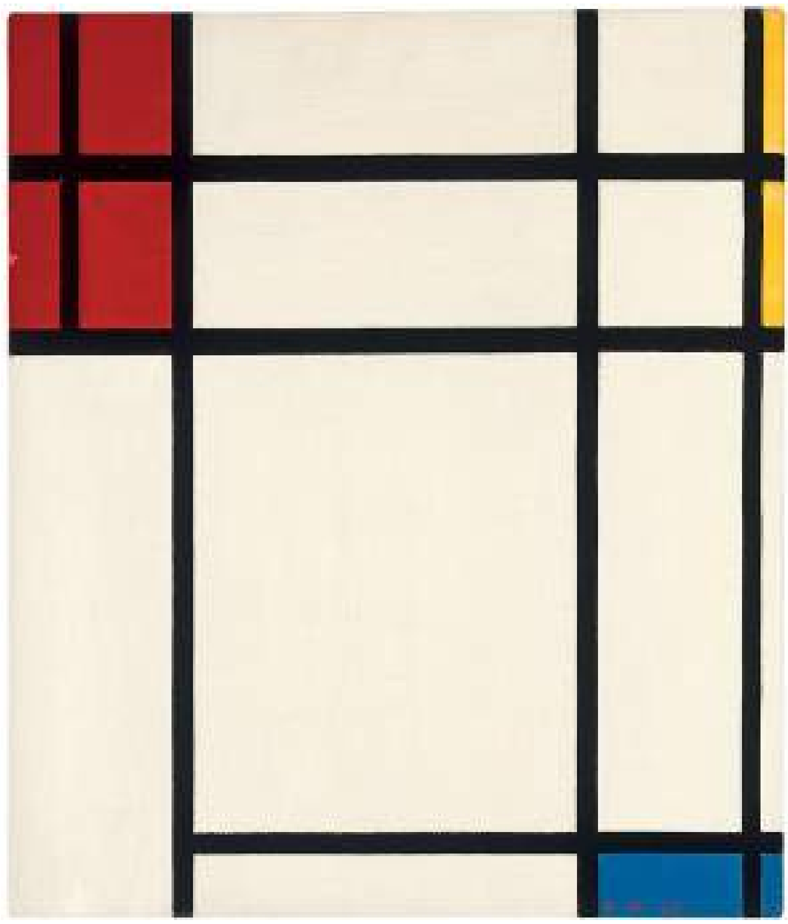
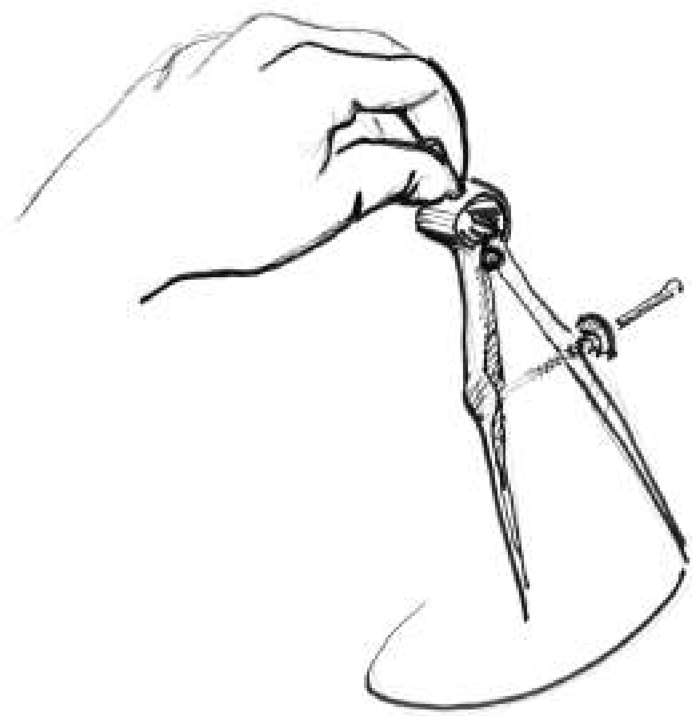


Figure 17: Piet Mondrian - Tableau (1921)

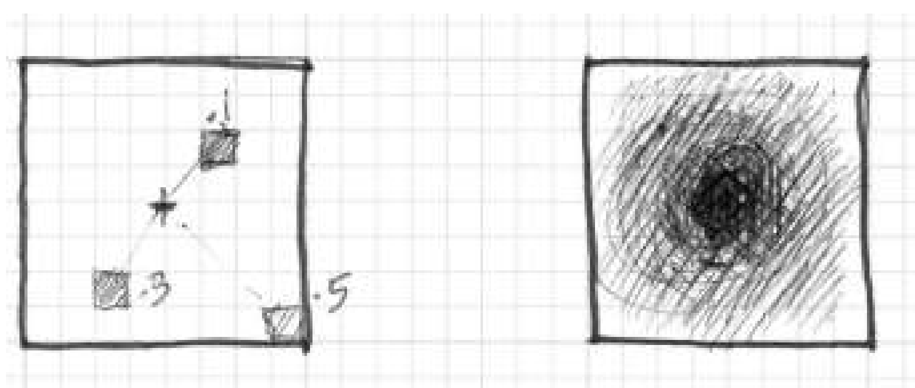
3.3.2 Circles

It's easy to draw squares on grid paper and rectangles on cartesian coordinates, but circles require another approach, especially since we need a "per-pixel" algorithm. One solution is to *re-map* the spatial coordinates so that we can use a `step()` function to draw a circle.

How? Let's start by going back to math class and the grid paper, where we opened a compass to the radius of a circle, pressed one of the compass points at the center of the circle and then traced the edge of the circle with a simple spin.



Translating this to a shader where each square on the grid paper is a pixel implies *asking* each pixel (or thread) if it is inside the area of the circle. We do this by computing the distance from the pixel to the center of the circle.



There are several ways to calculate that distance. The easiest one uses the `distance()` function, which internally computes the `length()` of the difference between two points (in our case the pixel coordinate and the center of the canvas). The `length()` function is nothing but a shortcut of the hypotenuse equation that uses square root (`sqrt()`) internally.

$$c = \sqrt{a^2 + b^2}.$$

You can use `distance()`, `length()` or `sqrt()` to calculate the distance to the center of the billboard. The following code contains these three functions and the non-surprising fact that each one returns exactly same result.

- Comment and uncomment lines to try the different ways to get the same result.

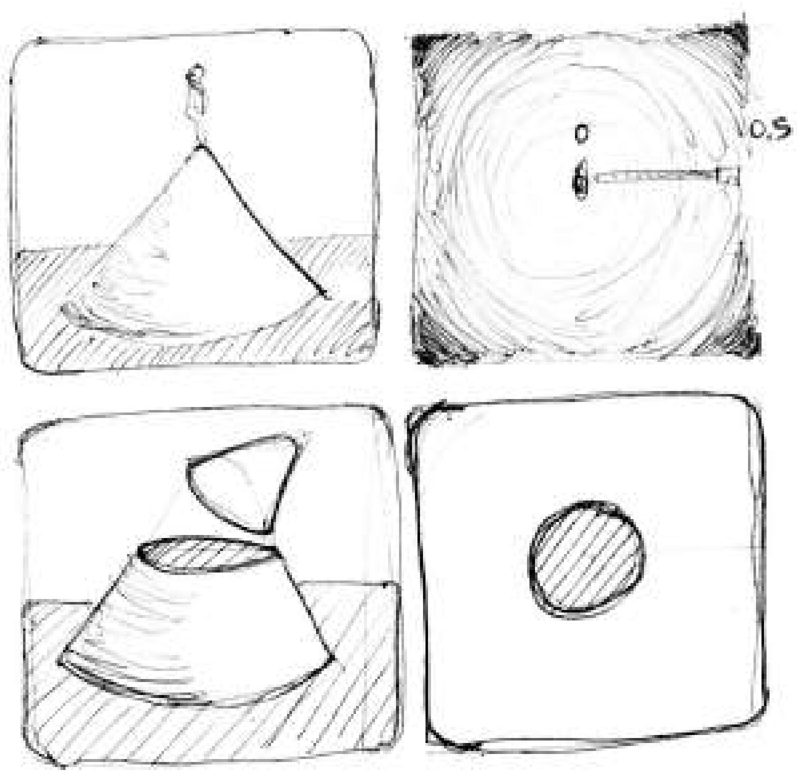
```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main(){
	vec2 st = gl_FragCoord.xy/u_resolution;
    float pct = 0.0;

    // a. The DISTANCE from the pixel to the center
    pct = distance(st,vec2(0.5));

    // b. The LENGTH of the vector
    //    from the pixel to the center
    // vec2 toCenter = vec2(0.5)-st;
    // pct = length(toCenter);

    // c. The SQUARE ROOT of the vector
    //    from the pixel to the center
    // vec2 tC = vec2(0.5)-st;
    // pct = sqrt(tC.x*tC.x+tC.y*tC.y);

    vec3 color = vec3(pct);

	gl_FragColor = vec4( color, 1.0 );
}
```

In the previous example we map the distance to the center of the billboard to the color brightness of the pixel. The closer a pixel is to the center, the lower (darker) value it has. Notice that the values don't get too high because from the center (`vec2(0.5, 0.5)`) the maximum distance barely goes over 0.5. Contemplate this map and think:

- What you can infer from it?
- How we can use this to draw a circle?
- Modify the above code in order to contain the entire circular gradient inside the canvas.

3.3.3 Distance field

We can also think of the above example as an altitude map, where darker implies taller. The gradient shows us something similar to the pattern made by a cone. Imagine yourself on the top of that cone. The horizontal distance to the edge of the cone is 0.5. This will be constant in all directions. By choosing where to "cut" the cone you will get a bigger or smaller circular surface.



Basically we are using a re-interpretation of the space (based on the distance to the center) to make shapes. This technique is known as a "distance field" and is used in different ways from font outlines to 3D graphics.

Try the following exercises:

- Use `step()` to turn everything above 0.5 to white and everything below to 0.0.
- Inverse the colors of the background and foreground.
- Using `smoothstep()`, experiment with different values to get nice smooth borders on your circle.
- Once you are happy with an implementation, make a function of it that you can reuse in the future.
- Add color to the circle.
- Can you animate your circle to grow and shrink, simulating a beating heart? (You can get some inspiration from the animation in the previous chapter.)
- What about moving this circle? Can you move it and place different circles in a single billboard?
- What happens if you combine distances fields together using different functions and operations?

```
pct = distance(st,vec2(0.4)) + distance(st,vec2(0.6));
pct = distance(st,vec2(0.4)) * distance(st,vec2(0.6));
pct = min(distance(st,vec2(0.4)),distance(st,vec2(0.6)));
pct = max(distance(st,vec2(0.4)),distance(st,vec2(0.6)));
pct = pow(distance(st,vec2(0.4)),distance(st,vec2(0.6)));
```

- Make three compositions using this technique. If they are animated, even better!

3.3.3.1 For your tool box

In terms of computational power the `sqrt()` function - and all the functions that depend on it - can be expensive. Here is another way to create a circular distance field by using `dot()` product.

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;
```

```
float circle(in vec2 _st, in float _radius){
    vec2 dist = _st-vec2(0.5);
        return 1.-smoothstep(_radius-(_radius*0.01),
                         _radius+(_radius*0.01),
                         dot(dist,dist)*4.0);
}

void main(){
        vec2 st = gl_FragCoord.xy/u_resolution.xy;

        vec3 color = vec3(circle(st,0.9));

        gl_FragColor = vec4( color, 1.0 );
}
```

3.3.4 Useful properties of a Distance Field

Figure 18: Zen garden

Distance fields can be used to draw almost everything. Obviously the more complex a shape is, the more complicated its equation will be, but once you have the formula to make distance fields of a particular shape it is very easy to combine and/or apply effects to it, like smooth edges and multiple outlines. Because of this, distance fields are popular in font rendering, like Mapbox GL Labels, Matt DesLauriers Material Design Fonts and as is described on Chapter 7 of iPhone 3D Programming, O'Reilly.

Take a look at the following code.

```
#ifdef GL_ES
precision mediump float;
```

```
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main(){
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  st.x *= u_resolution.x/u_resolution.y;
  vec3 color = vec3(0.0);
  float d = 0.0;

  // Remap the space to -1. to 1.
  st = st *2.-1.;

  // Make the distance field
  d = length( abs(st)-.3 );
  // d = length( min(abs(st)-.3,0.) );
  // d = length( max(abs(st)-.3,0.) );

  // Visualize the distance field
  gl_FragColor = vec4(vec3(fract(d*10.0)),1.0);

  // Drawing with the distance field
  // gl_FragColor = vec4(vec3( step(.3,d) ),1.0);
  // gl_FragColor = vec4(vec3( step(.3,d) * step(d,.4)),1.0);
  // gl_FragColor = vec4(vec3( smoothstep(.3,.4,d)* smoothstep(.6,.5,d)) ,1.0);
}
```

We start by moving the coordinate system to the center and shrinking it in half in order to remap the position values between -1 and 1. Also on *line 24* we are visualizing the distance field values using a `fract()` function making it easy to see the pattern they create. The distance field pattern repeats over and over like rings in a Zen garden.

Let's take a look at the distance field formula on *line 19*. There we are calculating the distance to the position on `(.3,.3)` or `vec3(.3)` in all four quadrants (that's what `abs()` is doing there).

If you uncomment *line 20*, you will note that we are combining the distances to these four points using the `min()` to zero. The result produces an interesting new pattern.

Now try uncommenting *line 21*; we are doing the same but using the `max()` function. The result is a rectangle with rounded corners. Note how the rings of the distance field get smoother the further away they get from the center.

Finish uncommenting *lines 27 to 29* one by one to understand the different uses of a distance field pattern.

3.3.5 Polar shapes

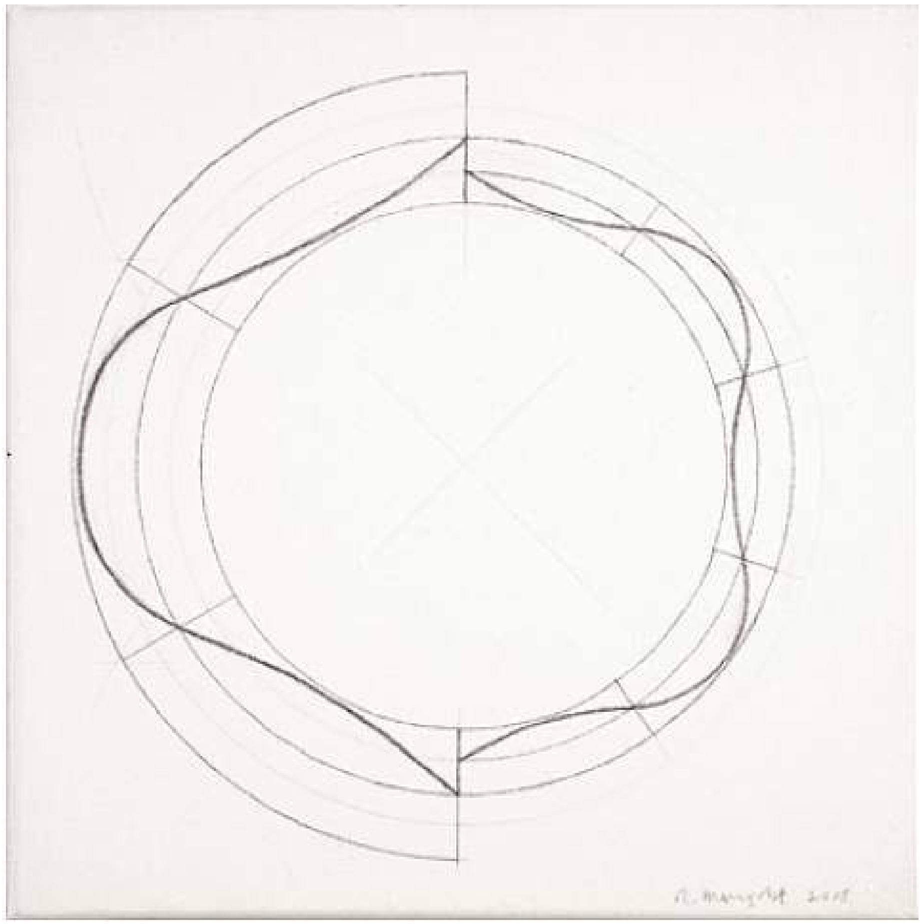


Figure 19: Robert Mangold - Untitled (2008)

In the chapter about color we map the cartesian coordinates to polar coordinates by calculating the *radius* and *angles* of each pixel with the following formula:

```
vec2 pos = vec2(0.5)-st;
float r = length(pos)*2.0;
float a = atan(pos.y,pos.x);
```

We use part of this formula at the beginning of the chapter to draw a circle. We calculated the distance to the center using `length()`. Now that we know about distance fields we can learn another way of drawing shapes using polar coordinates.

This technique is a little restrictive but very simple. It consists of changing the radius of a circle depending on the angle to achieve different shapes. How does the modulation work? Yes, using shaping functions!

Below you will find the same functions in the cartesian graph and in a polar coordinates shader example (between *lines 21 and 25*). Uncomment the functions one-by-one, paying attention the relationship between one coordinate system and the other.

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    vec2 pos = vec2(0.5)-st;

    float r = length(pos)*2.0;
    float a = atan(pos.y,pos.x);

    float f = cos(a*3.);
    // f = abs(cos(a*3.));
    // f = abs(cos(a*2.5))*.5+.3;
    // f = abs(cos(a*12.)*sin(a*3.))*.8+.1;
    // f = smoothstep(-.5,1., cos(a*10.))*0.2+0.5;

    color = vec3( 1.-smoothstep(f,f+0.02,r) );

    gl_FragColor = vec4(color, 1.0);
}
```

Try to:

- Animate these shapes.
- Combine different shaping functions to *cut holes* in the shape to make flowers, snowflakes and gears.
- Use the `plot()` function we were using in the *Shaping Functions Chapter* to draw just the contour.

3.3.6 Combining powers

Now that we've learned how to modulate the radius of a circle according to the angle using the `atan()` to draw different shapes, we can learn how use `atan()` with distance fields and apply all the tricks and effects possible with distance fields.

The trick will use the number of edges of a polygon to construct the distance field using polar coordinates. Check out the following code from Andrew Baldwin.

```
#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359
#define TWO_PI 6.28318530718

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Reference to
// http://thndl.com/square-shaped-shaders.html

void main(){
  vec2 st = gl_FragCoord.xy/u_resolution.xy;
  st.x *= u_resolution.x/u_resolution.y;
  vec3 color = vec3(0.0);
  float d = 0.0;

  // Remap the space to -1. to 1.
  st = st *2.-1.;

  // Number of sides of your shape
  int N = 3;

  // Angle and radius from the current pixel
  float a = atan(st.x,st.y)+PI;
  float r = TWO_PI/float(N);

  // Shaping function that modulate the distance
  d = cos(floor(.5+a/r)*r-a)*length(st);

  color = vec3(1.0-smoothstep(.4,.41,d));
  // color = vec3(d);

  gl_FragColor = vec4(color,1.0);
```

```
}
```

- Using this example, make a function that inputs the position and number of corners of a desired shape and returns a distance field value.
- Mix distance fields together using `min()` and `max()`.
- Choose a geometric logo to replicate using distance fields.

Congratulations! You have made it through the rough part! Take a break and let these concepts settle - drawing simple shapes in Processing is easy but not here. In shader-land drawing shapes is twisted, and it can be exhausting to adapt to this new paradigm of coding.

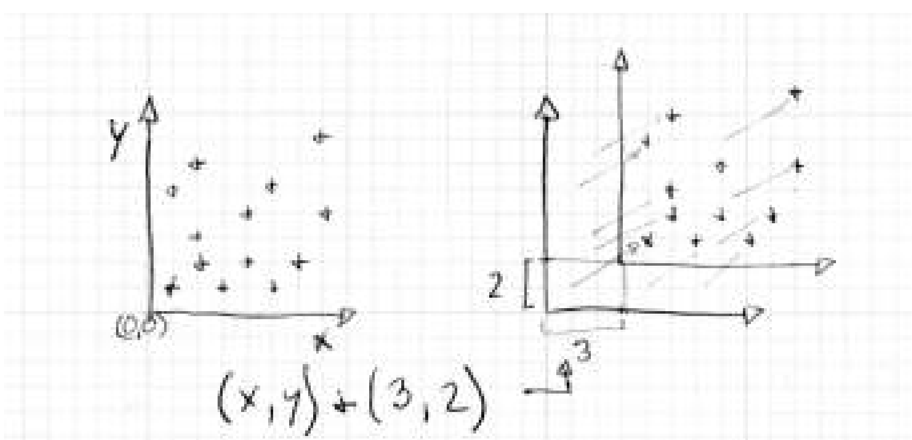
Down at the end of this chapter you will find a link to PixelSpirit Deck this deck of cards will help you learn new SDF functions, compose them into your designs and use on your shaders. The deck has a progressive learning curve, so taking one card a day and working on it will push and challenge your skills for months.

Now that you know how to draw shapes I'm sure new ideas will pop into your mind. In the following chapter you will learn how to move, rotate and scale shapes. This will allow you to make compositions!

3.4 2D Matrices

3.4.1 Translate

In the previous chapter we learned how to make some shapes - the trick to moving those shapes is to move the coordinate system itself. We can achieve that by simply adding a vector to the `st` variable that contains the location of each fragment. This causes the whole space coordinate system to move.



This is easier to see than to explain, so to see for yourself:

- Uncomment line 35 of the code below to see how the space itself moves around.

```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float box(in vec2 _st, in vec2 _size){
    _size = vec2(0.5) - _size*0.5;
    vec2 uv = smoothstep(_size,
                        _size+vec2(0.001),
                        _st);
    uv *= smoothstep(_size,
                    _size+vec2(0.001),
                    vec2(1.0)-_st);
    return uv.x*uv.y;
}

float cross(in vec2 _st, float _size){
    return  box(_st, vec2(_size,_size/4.)) +
            box(_st, vec2(_size/4.,_size));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // To move the cross we move the space
    vec2 translate = vec2(cos(u_time),sin(u_time));
    st += translate*0.35;

    // Show the coordinates of the space on the background
    // color = vec3(st.x,st.y,0.0);

    // Add the shape on the foreground
    color += vec3(cross(st,0.25));

    gl_FragColor = vec4(color,1.0);
}
```

Now try the following exercise:

- Using `u_time` together with the shaping functions move the small cross around in an interesting way. Search for a specific quality of motion you are interested in and try to make the cross move in the same way. Recording something from the "real world" first might be useful - it could be the coming and going of waves, a pendulum movement, a bouncing ball, a car accelerating, a bicycle stopping.

3.4.2 Rotations

To rotate objects we also need to move the entire space system. For that we are going to use a matrix. A matrix is an organized set of numbers in columns and rows. Vectors are multiplied by matrices following a precise set of rules in order to modify the values of the vector in a particular way.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix}$$

GLSL has native support for two, three and four dimensional matrices: `mat2` (2x2), `mat3` (3x3) and `mat4` (4x4). GLSL also supports matrix multiplication (`*`) and a matrix specific function (`matrixCompMult()`).

Based on how matrices behave it's possible to construct matrices to produce specific behaviors. For example we can use a matrix to translate a vector:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

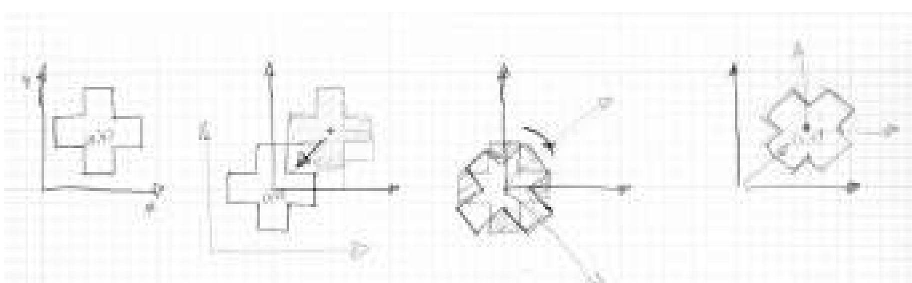
More interestingly, we can use a matrix to rotate the coordinate system:

$$\begin{bmatrix} cos\emptyset & -sin\emptyset & 0 \\ sin\emptyset & cos\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x.cos\emptyset - y.sin\emptyset \\ x.sin\emptyset + y.cos\emptyset \\ 1 \end{bmatrix}$$

Take a look at the following code for a function that constructs a 2D rotation matrix. This function follows the above formula for two dimensional vectors to rotate the coordinates around the `vec2(0.0)` point.

```
mat2 rotate2d(float _angle){
    return mat2(cos(_angle),-sin(_angle),
                sin(_angle),cos(_angle));
}
```

According to the way we've been drawing shapes, this is not exactly what we want. Our cross shape is drawn in the center of the canvas which corresponds to the position `vec2(0.5)`. So, before we rotate the space we need to move shape from the `center` to the `vec2(0.0)` coordinate, rotate the space, then finally move it back to the original place.



That looks like the following code:

```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform float u_time;
```

```
mat2 rotate2d(float _angle){
    return mat2(cos(_angle),-sin(_angle),
                sin(_angle),cos(_angle));
}

float box(in vec2 _st, in vec2 _size){
    _size = vec2(0.5) - _size*0.5;
    vec2 uv = smoothstep(_size,
                        _size+vec2(0.001),
                        _st);
    uv *= smoothstep(_size,
                    _size+vec2(0.001),
                    vec2(1.0)-_st);
    return uv.x*uv.y;
}

float cross(in vec2 _st, float _size){
    return  box(_st, vec2(_size,_size/4.)) +
            box(_st, vec2(_size/4.,_size));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // move space from the center to the vec2(0.0)
    st -= vec2(0.5);
    // rotate the space
    st = rotate2d( sin(u_time)*PI ) * st;
    // move it back to the original place
    st += vec2(0.5);

    // Show the coordinates of the space on the background
    // color = vec3(st.x,st.y,0.0);

    // Add the shape on the foreground
    color += vec3(cross(st,0.4));

    gl_FragColor = vec4(color,1.0);
}
```

Try the following exercises:

- Uncomment line 45 of above code and pay attention to what happens.
- Comment the translations before and after the rotation, on lines 37 and

39, and observe the consequences.

- Use rotations to improve the animation you simulated in the translation exercise.

3.4.3 Scale

We've seen how matrices are used to translate and rotate objects in space. (Or more precisely to transform the coordinate system to rotate and move the objects.) If you've used 3D modeling software or the push and pop matrix functions in Processing, you will know that matrices can also be used to scale the size of an object.

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} S_x . x \\ S_y . y \\ S_z . z \end{bmatrix}$$

Following the previous formula, we can figure out how to make a 2D scaling matrix:

```
mat2 scale(vec2 _scale){
    return mat2(_scale.x,0.0,
                0.0,_scale.y);
}
```

```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265359

uniform vec2 u_resolution;
uniform float u_time;

mat2 scale(vec2 _scale){
    return mat2(_scale.x,0.0,
                0.0,_scale.y);
}

float box(in vec2 _st, in vec2 _size){
    _size = vec2(0.5) - _size*0.5;
    vec2 uv = smoothstep(_size,
                        _size+vec2(0.001),
                        _st);
    uv *= smoothstep(_size,
```

```
                    _size+vec2(0.001),
                    vec2(1.0)-_st);
    return uv.x*uv.y;
}

float cross(in vec2 _st, float _size){
    return  box(_st, vec2(_size,_size/4.)) +
            box(_st, vec2(_size/4.,_size));
}

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    st -= vec2(0.5);
    st = scale( vec2(sin(u_time)+1.0) ) * st;
    st += vec2(0.5);

    // Show the coordinates of the space on the background
    // color = vec3(st.x,st.y,0.0);

    // Add the shape on the foreground
    color += vec3(cross(st,0.2));

    gl_FragColor = vec4(color,1.0);
}
```

Try the following exercises to understand more deeply how this works.

- Uncomment line 42 of above code to see the space coordinate being scaled.
- See what happens when you comment the translations before and after the scaling on lines 37 and 39.
- Try combining a rotation matrix together with a scale matrix. Be aware that the order matters. Multiply by the matrix first and then multiply the vectors.
- Now that you know how to draw different shapes, and move, rotate and scale them, it's time to make a nice composition. Design and construct a fake UI or HUD (heads up display). Use the following ShaderToy example by Ndel for inspiration and reference.

3.4.4 Other uses for matrices: YUV color

YUV is a color space used for analog encoding of photos and videos that takes into account the range of human perception to reduce the bandwidth of chrominance components.

The following code is an interesting opportunity to use matrix operations in GLSL to transform colors from one mode to another.

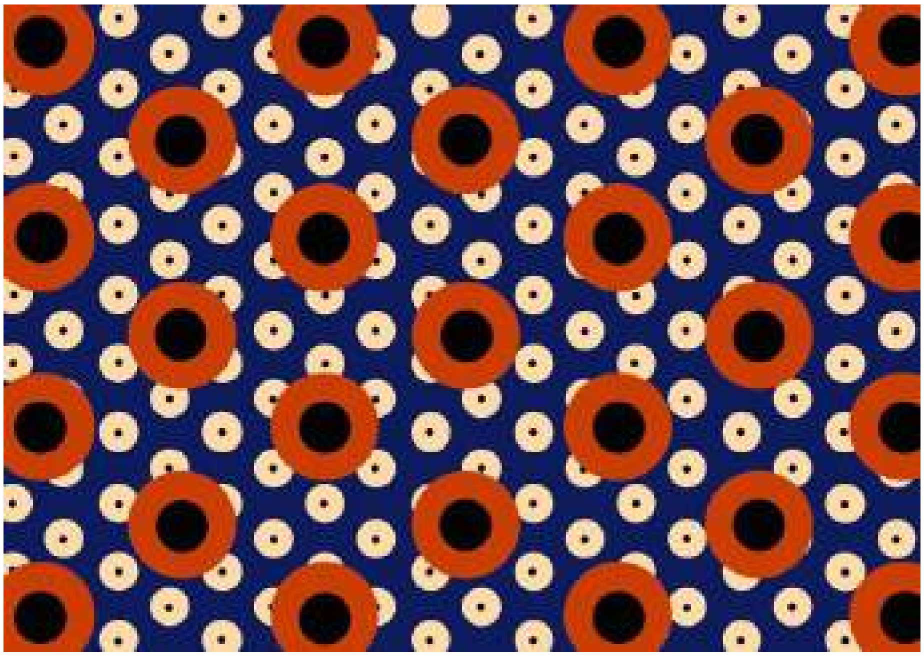
```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

// YUV to RGB matrix
mat3 yuv2rgb = mat3(1.0, 0.0, 1.13983,
                    1.0, -0.39465, -0.58060,
                    1.0, 2.03211, 0.0);

// RGB to YUV matrix
mat3 rgb2yuv = mat3(0.2126, 0.7152, 0.0722,
                    -0.09991, -0.33609, 0.43600,
                    0.615, -0.5586, -0.05639);

void main(){
    vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    // UV values goes from -1 to 1
    // So we need to remap st (0.0 to 1.0)
    st -= 0.5;  // becomes -0.5 to 0.5
    st *= 2.0;  // becomes -1.0 to 1.0

    // we pass st as the y & z values of
    // a three dimensional vector to be
    // properly multiply by a 3x3 matrix
    color = yuv2rgb * vec3(0.5, st.x, st.y);

    gl_FragColor = vec4(color,1.0);
}
```

As you can see we are treating colors as vectors by multiplying them with matrices. In that way we "move" the values around.

In this chapter we've learned how to use matrix transformations to move, rotate and scale vectors. These transformations will be essential for making compositions out of the shapes we learned about in the previous chapter. In the next chapter we'll apply all we've learned to make beautiful procedural patterns. You will find that coding repetition and variation can be an exciting practice.

3.5 Patterns

Since shader programs are executed by pixel-by-pixel no matter how much you repeat a shape the number of calculations stays constant. This means that fragment shaders are particulary suitable for tile patterns.



In this chapter we are going to apply what we've learned so far and repeat it along a canvas. Like in previous chapters, our strategy will be based on multiplying the space coordinates (between 0.0 and 1.0), so that the shapes we draw between the values 0.0 and 1.0 will be repeated to make a grid.

"The grid provides a framework within which human intuition and invention can operate and that it can subvert. Within the chaos of nature patterns provide a constrast and promise of order. From early patterns on pottery to geometric mosaics in Roman baths, people have long used grids to enhance their lives with decoration." 10 PRINT, Mit Press, (2013)

First let's remember the `fract()` function. It returns the fractional part of a number, making `fract()` in essence the modulo of one (`mod(x,1.0)`). In other words, `fract()` returns the number after the floating point. Our normalized coordinate system variable (`st`) already goes from 0.0 to 1.0 so it doesn't make sense to do something like:

```
void main(){
	vec2 st = gl_FragCoord.xy/u_resolution;
	vec3 color = vec3(0.0);
    st = fract(st);
```

```
        color = vec3(st,0.0);
        gl_FragColor = vec4(color,1.0);
}
```

But if we scale the normalized coordinate system up - let's say by three - we will get three sequences of linear interpolations between 0-1: the first one between 0-1, the second one for the floating points between 1-2 and the third one for the floating points between 2-3.

```
// Author @patriciogv - 2015

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

float circle(in vec2 _st, in float _radius){
    vec2 l = _st-vec2(0.5);
    return 1.-smoothstep(_radius-(_radius*0.01),
                         _radius+(_radius*0.01),
                         dot(l,l)*4.0);
}

void main() {
        vec2 st = gl_FragCoord.xy/u_resolution;
    vec3 color = vec3(0.0);

    st *= 3.0;      // Scale up the space by 3
    st = fract(st); // Wrap around 1.0

    // Now we have 9 spaces that go from 0-1

    color = vec3(st,0.0);
    //color = vec3(circle(st,0.5));

        gl_FragColor = vec4(color,1.0);
}
```

Now it's time to draw something in each subspace, by uncommenting line 27. (Because we are multiplying equally in x and y the aspect ratio of the space doesn't change and shapes will be as expected.)

Try some of the following exercises to get a deeper understanding:

- Multiply the space by different numbers. Try with floating point values and also with different values for x and y.

- Make a reusable function of this tiling trick.
- Divide the space into 3 rows and 3 columns. Find a way to know in which column and row the thread is and use that to change the shape that is displaying. Try to compose a tic-tac-toe match.

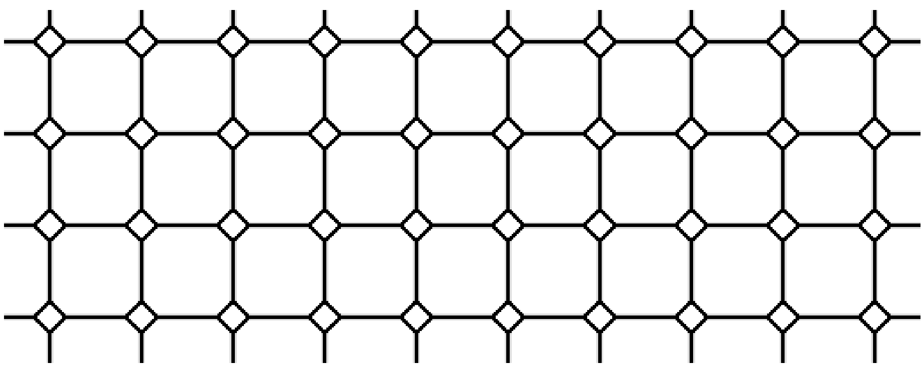
3.5.1 Apply matrices inside patterns

Since each subdivision or cell is a smaller version of the normalized coordinate system we have already been using, we can apply a matrix transformation to it in order to translate, rotate or scale the space inside.

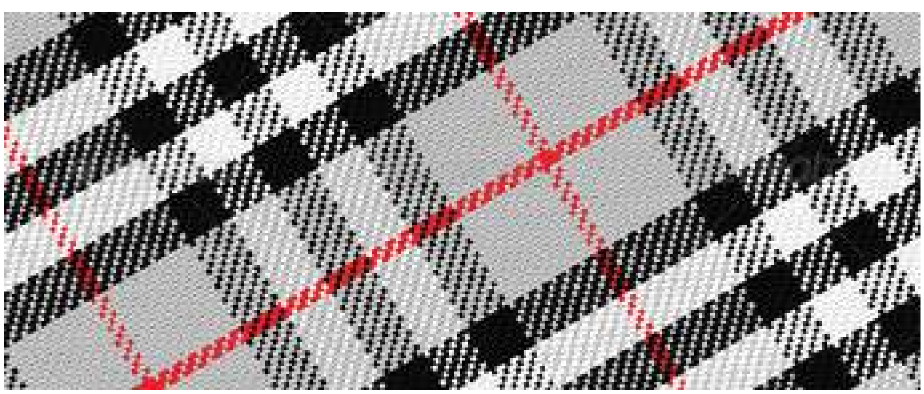
```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

#define PI 3.14159265358979323846

vec2 rotate2D(vec2 _st, float _angle){
    _st -= 0.5;
    _st =  mat2(cos(_angle),-sin(_angle),
                sin(_angle),cos(_angle)) * _st;
    _st += 0.5;
    return _st;
}

vec2 tile(vec2 _st, float _zoom){
    _st *= _zoom;
    return fract(_st);
}

float box(vec2 _st, vec2 _size, float _smoothEdges){
    _size = vec2(0.5)-_size*0.5;
    vec2 aa = vec2(_smoothEdges*0.5);
    vec2 uv = smoothstep(_size,_size+aa,_st);
    uv *= smoothstep(_size,_size+aa,vec2(1.0)-_st);
    return uv.x*uv.y;
}

void main(void){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);
```

```
    // Divide the space in 4
    st = tile(st,4.);

    // Use a matrix to rotate the space 45 degrees
    st = rotate2D(st,PI*0.25);

    // Draw a square
    color = vec3(box(st,vec2(0.7),0.01));
    // color = vec3(st,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

- Think of interesting ways of animating this pattern. Consider animating color, shapes and motion. Make three different animations.
- Recreate more complicated patterns by composing different shapes.

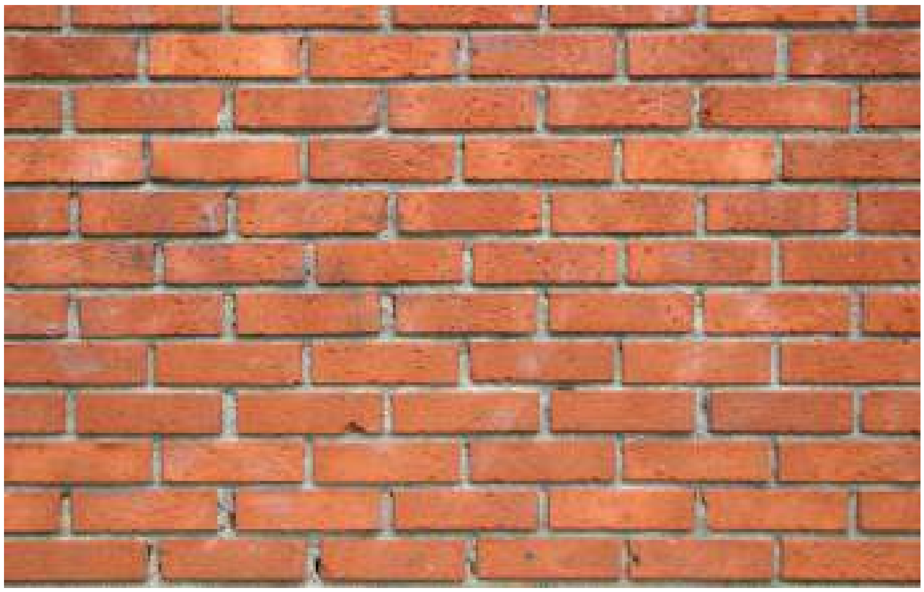


- Combine different layers of patterns to compose your own Scottish Tartan Patterns.



3.5.2 Offset patterns

So let's say we want to imitate a brick wall. Looking at the wall, you can see a half brick offset on x in every other row. How we can do that?



As a first step we need to know if the row of our thread is an even or odd number, because we can use that to determine if we need to offset the x in that row.

______we have to fix these next two paragraphs together______

To determine if our thread is in an odd or even row, we are going to use `mod()` of `2.0` and then see if the result is under `1.0` or not. Take a look at the following formula and uncomment the two last lines.

As you can see we can use a ternary operator to check if the `mod()` of `2.0` is under `1.0` (second line) or similarly we can use a `step()` function which does the same operation, but faster. Why? Although is hard to know how each graphic card optimizes and compiles the code, it is safe to assume that built-in functions are faster than non-built-in ones. Everytime you can use a built-in function, use it!

So now that we have our odd number formula we can apply an offset to the odd rows to give a *brick* effect to our tiles. Line 14 of the following code is where we are using the function to "detect" odd rows and give them a half-unit offset on `x`. Note that for even rows, the result of our function is `0.0`, and multiplying `0.0` by the offset of `0.5` gives an offset of `0.0`. But on odd rows we multiply the result of our function, `1.0`, by the offset of `0.5`, which moves the `x` axis of the coordinate system by `0.5`.

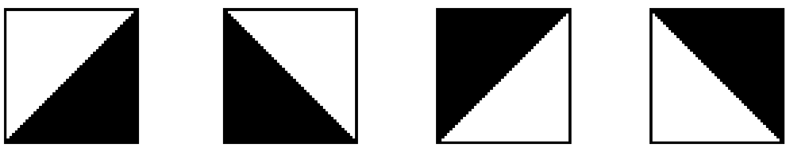
Now try uncommenting line 32 - this stretches the aspect ratio of the coordinate system to mimic the aspect of a "modern brick". By uncommenting line 40 you can see how the coordinate system looks mapped to red and green.

```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform float u_time;

vec2 brickTile(vec2 _st, float _zoom){
    _st *= _zoom;

    // Here is where the offset is happening
    _st.x += step(1., mod(_st.y,2.0)) * 0.5;

    return fract(_st);
}

float box(vec2 _st, vec2 _size){
    _size = vec2(0.5)-_size*0.5;
    vec2 uv = smoothstep(_size,_size+vec2(1e-4),_st);
    uv *= smoothstep(_size,_size+vec2(1e-4),vec2(1.0)-_st);
    return uv.x*uv.y;
}

void main(void){
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // Modern metric brick of 215mm x 102.5mm x 65mm
    // http://www.jaharrison.me.uk/Brickwork/Sizes.html
    // st /= vec2(2.15,0.65)/1.5;

    // Apply the brick tiling
    st = brickTile(st,5.0);

    color = vec3(box(st,vec2(0.9)));

    // Uncomment to see the space coordinates
    // color = vec3(st,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

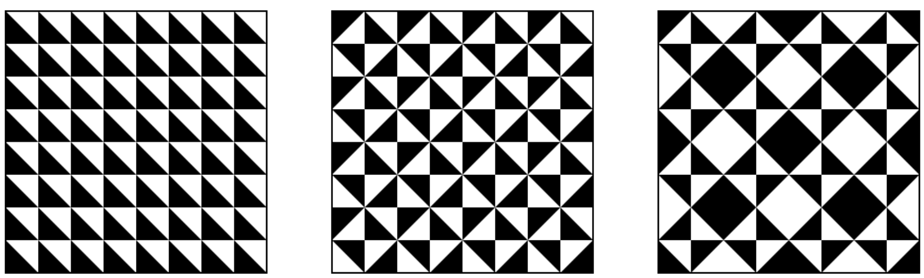
- Try animating this by moving the offset according to time.
- Make another animation where even rows move to the left and odd rows move to the right.
- Can you repeat this effect but with columns?
- Try combining an offset on `x` and `y` axis to get something like this:

3.6 Truchet Tiles

Now that we've learned how to tell if our cell is in an even or odd row or column, it's possible to reuse a single design element depending on its position. Consider the case of the Truchet Tiles where a single design element can be presented in four different ways:



By changing the pattern across tiles, it's possible to construct an infinite set of complex designs.



Pay close attention to the function `rotateTilePattern()`, which subdivides the space into four cells and assigns an angle of rotation to each one.

```
// Author @patriciogv ( patriciogonzalezvivo.com ) - 2015

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265358979323846

uniform vec2 u_resolution;
uniform float u_time;
```

```
vec2 rotate2D (vec2 _st, float _angle) {
    _st -= 0.5;
    _st =  mat2(cos(_angle),-sin(_angle),
                sin(_angle),cos(_angle)) * _st;
    _st += 0.5;
    return _st;
}

vec2 tile (vec2 _st, float _zoom) {
    _st *= _zoom;
    return fract(_st);
}

vec2 rotateTilePattern(vec2 _st){

    //  Scale the coordinate system by 2x2
    _st *= 2.0;

    //  Give each cell an index number
    //  according to its position
    float index = 0.0;
    index += step(1., mod(_st.x,2.0));
    index += step(1., mod(_st.y,2.0))*2.0;

    //      |
    //  2   |   3
    //      |
    //--------------
    //      |
    //  0   |   1
    //      |

    // Make each cell between 0.0 - 1.0
    _st = fract(_st);

    // Rotate each cell according to the index
    if(index == 1.0){
        //  Rotate cell 1 by 90 degrees
        _st = rotate2D(_st,PI*0.5);
    } else if(index == 2.0){
        //  Rotate cell 2 by -90 degrees
        _st = rotate2D(_st,PI*-0.5);
    } else if(index == 3.0){
        //  Rotate cell 3 by 180 degrees
        _st = rotate2D(_st,PI);
```

```
    }

    return _st;
}

void main (void) {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    st = tile(st,3.0);
    st = rotateTilePattern(st);

    // Make more interesting combinations
    // st = tile(st,2.0);
    // st = rotate2D(st,-PI*u_time*0.25);
    // st = rotateTilePattern(st*2.);
    // st = rotate2D(st,PI*u_time*0.25);

    // step(st.x,st.y) just makes a b&w triangles
    // but you can use whatever design you want.
    gl_FragColor = vec4(vec3(step(st.x,st.y)),1.0);
}
```

- Comment, uncomment and duplicate lines 69 to 72 to compose new designs.
- Change the black and white triangle for another element like: half circles, rotated squares or lines.
- Code other patterns where the elements are rotated according to their position.
- Make a pattern that changes other properties according to the position of the elements.
- Think of something else that is not necessarily a pattern where you can apply the principles from this section. (Ex: I Ching hexagrams)

3.7 Making your own rules

Making procedural patterns is a mental exercise in finding minimal reusable elements. This practice is old; we as a species have been using grids and patterns to decorate textiles, floors and borders of objects for a long time: from meanders patterns in ancient Greece, to Chinese lattice design, the pleasure of repetition and variation catches our imagination. Take some time to look at decorative patterns and see how artists and designers have a long history of navigating the fine edge between the predictability of order and the surprise of variation and chaos. From Arabic geometrical patterns, to gorgeous African fabric designs, there is an entire universe of patterns to learn from.

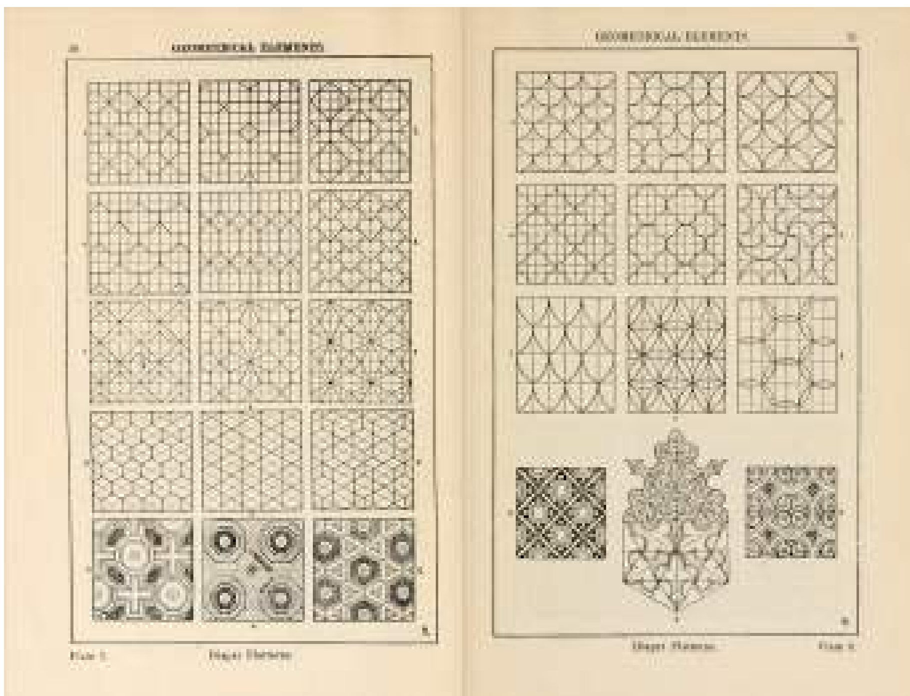


Figure 20: Franz Sales Meyer - A handbook of ornament (1920)

With this chapter we end the section on Algorithmic Drawing. In the following chapters we will learn how to bring some entropy to our shaders and produce generative designs.

4 Generative designs

It is not a surprise that after so much repetition and order the author is forced to bring some chaos.

4.1 Random

Randomness is a maximal expression of entropy. How can we generate randomness inside the seemingly predictable and rigid code environment?

Let's start by analyzing the following function:

Above we are extracting the fractional content of a sine wave. The `sin()` values that fluctuate between `-1.0` and `1.0` have been chopped behind the floating point, returning all positive values between `0.0` and `1.0`. We can use this effect to get some pseudo-random values by "breaking" this sine wave into smaller pieces. How? By multiplying the resultant of `sin(x)` by larger numbers. Go ahead and click on the function above and start adding some zeros.

By the time you get to `100000.0` (and the equation looks like this: `y = fract(sin(x)*100000.0)`) you aren't able to distinguish the sine wave any more. The granularity of the fractional part has corrupted the flow of the sine wave into pseudo-random chaos.

4.2 Controlling chaos

Using random can be hard; it is both too chaotic and sometimes not random enough. Take a look at the following graph. To make it, we are using a `rand()` function which is implemented exactly like we describe above.

Taking a closer look, you can see the `sin()` wave crest at `-1.5707` and `1.5707`. I bet you now understand why - it's where the maximum and minimum of the sine wave happens.

If look closely at the random distribution, you will note that the there is some concentration around the middle compared to the edges.

A while ago Pixelero published an interesting article about random distribution. I've added some of the functions he uses in the previous graph for you to play with and see how the distribution can be changed. Uncomment the functions and see what happens.

If you read Pixelero's article, it is important to keep in mind that our `rand()` function is a deterministic random, also known as pseudo-random. Which means for example `rand(1.)` is always going to return the same value. Pixelero makes reference to the ActionScript function `Math.random()` which is non-deterministic; every call will return a different value.

4.3 2D Random

Now that we have a better understanding of randomness, it's time to apply it in two dimensions, to both the `x` and `y` axis. For that we need a way to transform a two dimensional vector into a one dimensional floating point value. There are different ways to do this, but the `dot()` function is particulary helpful in this case. It returns a single float value between `0.0` and `1.0` depending on the alignment of two vectors.

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}
```

```
void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    float rnd = random( st );

    gl_FragColor = vec4(vec3(rnd),1.0);
}
```

Take a look at lines 13 to 15 and notice how we are comparing the `vec2 st` with another two dimensional vector (`vec2(12.9898,78.233)`).

- Try changing the values on lines 14 and 15. See how the random pattern changes and think about what we can learn from this.
- Hook this random function to the mouse interaction (`u_mouse`) and time (`u_time`) to understand better how it works.

4.4 Using the chaos

Random in two dimensions looks a lot like TV noise, right? It's a hard raw material to use to compose images. Let's learn how to make use of it.

Our first step is to apply a grid to it; using the `floor()` function we will generate an integer table of cells. Take a look at the following code, especially lines 22 and 23.

```
// Author @patriciogv - 2015
// Title: Mosaic

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    st *= 10.0; // Scale the coordinate system by 10
```

```
    vec2 ipos = floor(st);  // get the integer coords
    vec2 fpos = fract(st);  // get the fractional coords

    // Assign a random value based on the integer coord
    vec3 color = vec3(random( ipos ));

    // Uncomment to see the subdivided grid
    // color = vec3(fpos,0.0);

    gl_FragColor = vec4(color,1.0);
}
```

After scaling the space by 10 (on line 21), we separate the integers of the coordinates from the fractional part. We are familiar with this last operation because we have been using it to subdivide a space into smaller cells that go from `0.0` to `1.0`. By obtaining the integer of the coordinate we isolate a common value for a region of pixels, which will look like a single cell. Then we can use that common integer to obtain a random value for that area. Because our random function is deterministic, the random value returned will be constant for all the pixels in that cell.

Uncomment line 29 to see that we preserve the floating part of the coordinate, so we can still use that as a coordinate system to draw things inside each cell.

Combining these two values - the integer part and the fractional part of the coordinate - will allow you to mix variation and order.

Take a look at this GLSL port of the famous `10 PRINT CHR$(205.5+RND(1)); : GOTO 10` maze generator.

```
// Author @patriciogv - 2015
// Title: Truchet - 10 print

#ifdef GL_ES
precision mediump float;
#endif

#define PI 3.14159265358979323846

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

float random (in vec2 _st) {
    return fract(sin(dot(_st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}
```

```
vec2 truchetPattern(in vec2 _st, in float _index){
    _index = fract(((_index-0.5)*2.0));
    if (_index > 0.75) {
        _st = vec2(1.0) - _st;
    } else if (_index > 0.5) {
        _st = vec2(1.0-_st.x,_st.y);
    } else if (_index > 0.25) {
        _st = 1.0-vec2(1.0-_st.x,_st.y);
    }
    return _st;
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st *= 10.0;
    // st = (st-vec2(5.0))*(abs(sin(u_time*0.2))*5.);
    // st.x += u_time*3.0;

    vec2 ipos = floor(st);  // integer
    vec2 fpos = fract(st);  // fraction

    vec2 tile = truchetPattern(fpos, random( ipos ));

    float color = 0.0;

    // Maze
    color = smoothstep(tile.x-0.3,tile.x,tile.y)-
            smoothstep(tile.x,tile.x+0.3,tile.y);

    // Circles
    // color = (step(length(tile),0.6) -
    //          step(length(tile),0.4) ) +
    //         (step(length(tile-vec2(1.)),0.6) -
    //          step(length(tile-vec2(1.)),0.4) );

    // Truchet (2 triangles)
    // color = step(tile.x,tile.y);

    gl_FragColor = vec4(vec3(color),1.0);
}
```

Here I'm using the random values of the cells to draw a line in one direction or the other using the `truchetPattern()` function from the previous chapter (lines 41 to 47).

You can get another interesting pattern by uncommenting the block of lines between 50 to 53, or animate the pattern by uncommenting lines 35 and 36.

4.5 Master Random

Ryoji Ikeda, Japanese electronic composer and visual artist, has mastered the use of random; it is hard not to be touched and mesmerized by his work. His use of randomness in audio and visual mediums is forged in such a way that it is not annoying chaos but a mirror of the complexity of our technological culture.

Take a look at Ikeda's work and try the following exercises:

- Make rows of moving cells (in opposite directions) with random values. Only display the cells with brighter values. Make the velocity of the rows fluctuate over time.
- Similarly make several rows but each one with a different speed and direction. Hook the position of the mouse to the threshold of which cells to show.
- Create other interesting effects.

Using random aesthetically can be problematic, especially if you want to make natural-looking simulations. Random is simply too chaotic and very few things look `random()` in real life. If you look at a rain pattern or a stock chart, which are both quite random, they are nothing like the random pattern we made at the begining of this chapter. The reason? Well, random values have no correlation between them what so ever, but most natural patterns have some memory of the previous state.

In the next chapter we will learn about noise, the smooth and *natural looking* way of creating computational chaos.

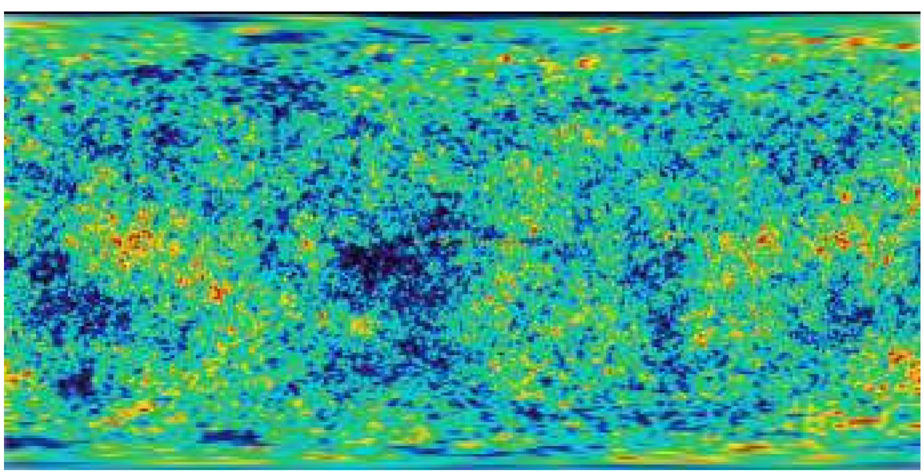
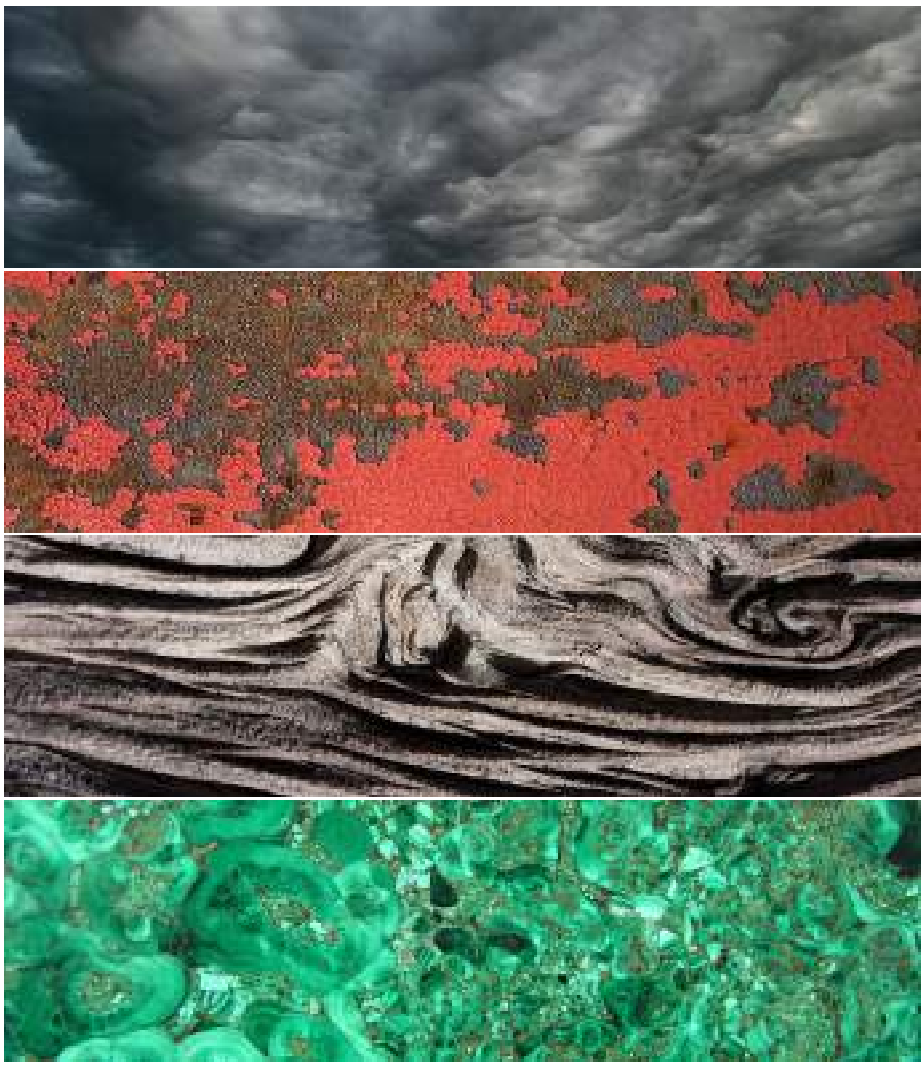


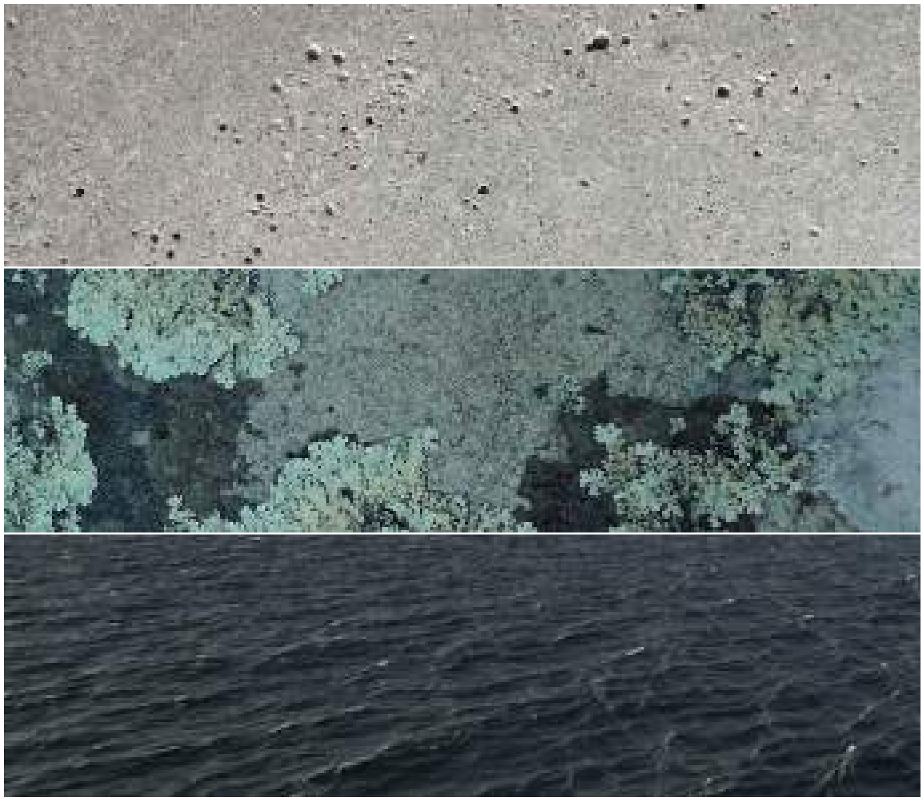
Figure 21: NASA / WMAP science team

4.6 Noise

It's time for a break! We've been playing with random functions that look like TV white noise, our head is still spinning thinking about shaders, and our eyes are tired. Time to go out for a walk!

We feel the air on our skin, the sun in our face. The world is such a vivid and rich place. Colors, textures, sounds. While we walk we can't avoid noticing the surface of the roads, rocks, trees and clouds.





The unpredictability of these textures could be called "random," but they don't look like the random we were playing with before. The "real world" is such a rich and complex place! How can we approximate this variety computationally?

This was the question Ken Perlin was trying to solve in the early 1980s when he was commissioned to generate more realistic textures for the movie "Tron." In response to that, he came up with an elegant *Oscar winning* noise algorithm. (No biggie.)

The following is not the classic Perlin noise algorithm, but it is a good starting point to understand how to generate noise.

In these lines we are doing something similar to what we did in the previous chapter. We are subdividing a continuous floating number (`x`) into its integer (`i`) and fractional (`f`) components. We use `floor()` to obtain `i` and `fract()` to obtain `f`. Then we apply `rand()` to the integer part of `x`, which gives a unique random value for each integer.

After that you see two commented lines. The first one interpolates each random value linearly.

```
y = mix(rand(i), rand(i + 1.0), f);
```

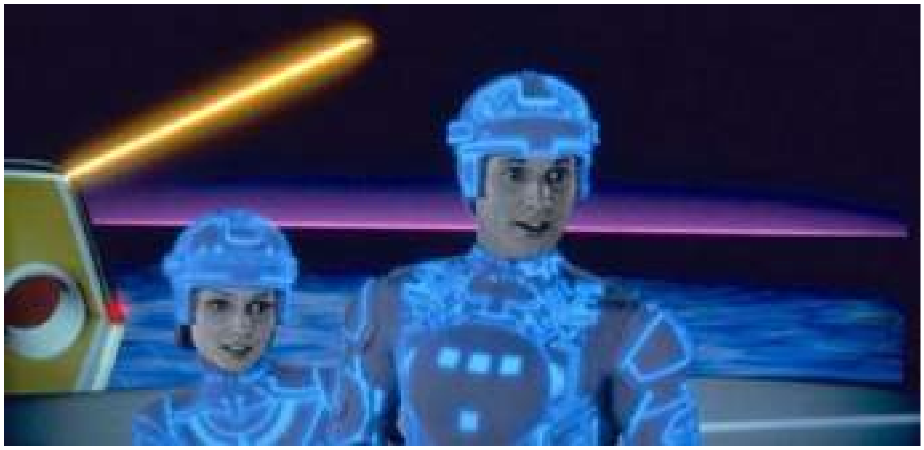


Figure 22: Disney - Tron (1982)

Go ahead and uncomment this line to see how this looks. We use the `fract()` value store in `f` to `mix()` the two random values.

At this point in the book, we've learned that we can do better than a linear interpolation, right? Now try uncommenting the following line, which uses a `smoothstep()` interpolation instead of a linear one.

```
y = mix(rand(i), rand(i + 1.0), smoothstep(0.,1.,f));
```

After uncommenting it, notice how the transition between the peaks gets smooth. In some noise implementations you will find that programmers prefer to code their own cubic curves (like the following formula) instead of using the `smoothstep()`.

```
float u = f * f * (3.0 - 2.0 * f ); // custom cubic curve
y = mix(rand(i), rand(i + 1.0), u); // using it in the interpolation
```

This *smooth randomness* is a game changer for graphical engineers or artists - it provides the ability to generate images and geometries with an organic feeling. Perlin's Noise Algorithm has been implemented over and over in different languages and dimensions to make mesmerizing pieces for all sorts of creative uses.

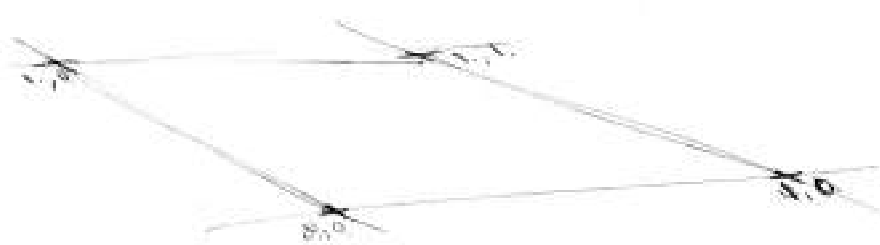
Now it's your turn:

- Make your own `float noise(float x)` function.
- Use your noise function to animate a shape by moving it, rotating it or scaling it.
- Make an animated composition of several shapes 'dancing' together using noise.

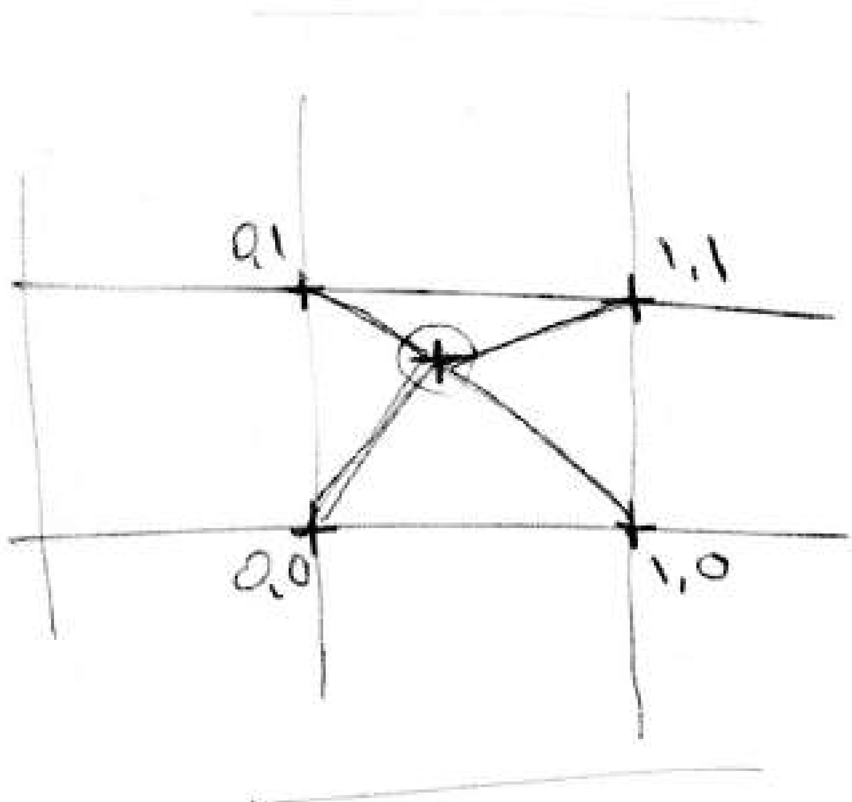
Figure 23: Robert Hodgin - Written Images (2010)

- Construct "organic-looking" shapes using the noise function.
- Once you have your "creature," try to develop it further into a character by assigning it a particular movement.

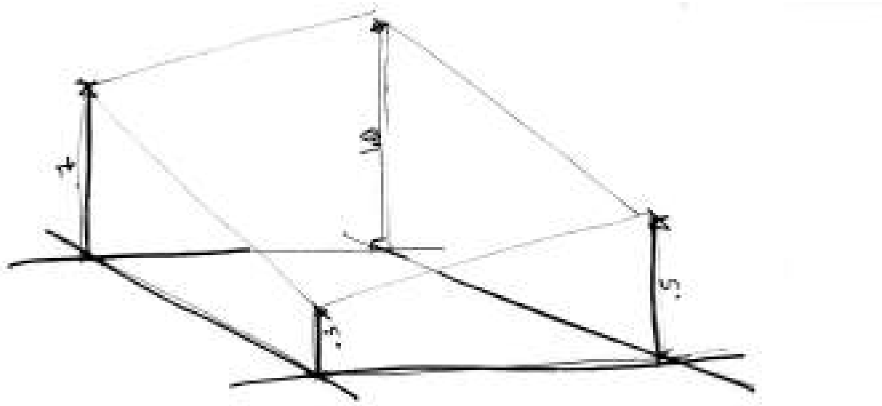
4.7 2D Noise



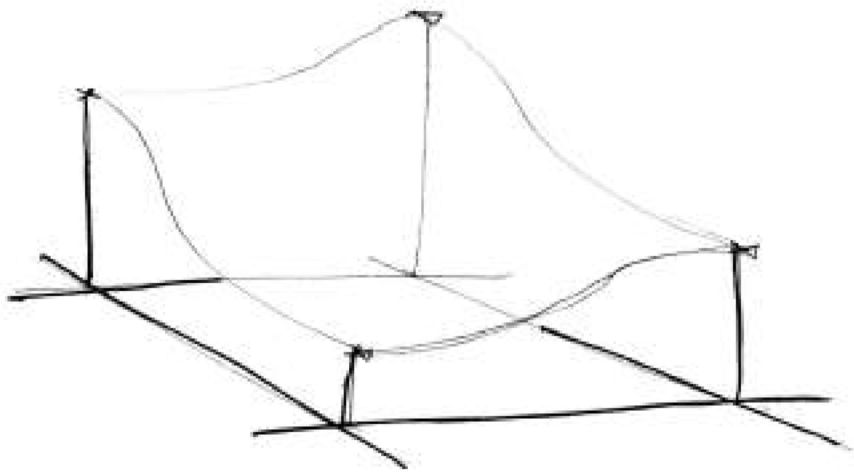
Now that we know how to do noise in 1D, it's time to move on to 2D. In 2D, instead of interpolating between two points of a line (`fract(x)` and `fract(x)+1.0`), we are going to interpolate between the four corners of the square area of a plane (`fract(st)`, `fract(st)+vec2(1.,0.)`, `fract(st)+vec2(0.,1.)` and `fract(st)+vec2(1.,1.)`).



Similarly, if we want to obtain 3D noise we need to interpolate between the eight corners of a cube. This technique is all about interpolating random values, which is why it's called **value noise**.



Like the 1D example, this interpolation is not linear but cubic, which smoothly interpolates any points inside our square grid.



Take a look at the following noise function.

```
#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// 2D Random
float random (in vec2 st) {
    return fract(sin(dot(st.xy,
```

```
                         vec2(12.9898,78.233)))
                 * 43758.5453123);
}

// 2D Noise based on Morgan McGuire @morgan3d
// https://www.shadertoy.com/view/4dS3Wd
float noise (in vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

    // Four corners in 2D of a tile
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));

    // Smooth Interpolation

    // Cubic Hermine Curve.  Same as SmoothStep()
    vec2 u = f*f*(3.0-2.0*f);
    // u = smoothstep(0.,1.,f);

    // Mix 4 coorners percentages
    return mix(a, b, u.x) +
            (c - a)* u.y * (1.0 - u.x) +
            (d - b) * u.x * u.y;
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    // Scale the coordinate system to see
    // some noise in action
    vec2 pos = vec2(st*5.0);

    // Use the noise function
    float n = noise(pos);

    gl_FragColor = vec4(vec3(n), 1.0);
}
```

We start by scaling the space by 5 (line 45) in order to see the interpolation between the squares of the grid. Then inside the noise function we subdivide the space into cells. We store the integer position of the cell along with the fractional positions inside the cell. We use the integer position to calculate the four corners' coordinates and obtain a random value for each one (lines 23-26).

Finally, in line 35 we interpolate between the 4 random values of the corners using the fractional positions we stored before.

Now it's your turn. Try the following exercises:

- Change the multiplier of line 45. Try to animate it.
- At what level of zoom does the noise start looking like random again?
- At what zoom level is the noise is imperceptible?
- Try to hook up this noise function to the mouse coordinates.
- What if we treat the gradient of the noise as a distance field? Make something interesting with it.
- Now that you've achieved some control over order and chaos, it's time to use that knowledge. Make a composition of rectangles, colors and noise that resembles some of the complexity of a Mark Rothko painting.

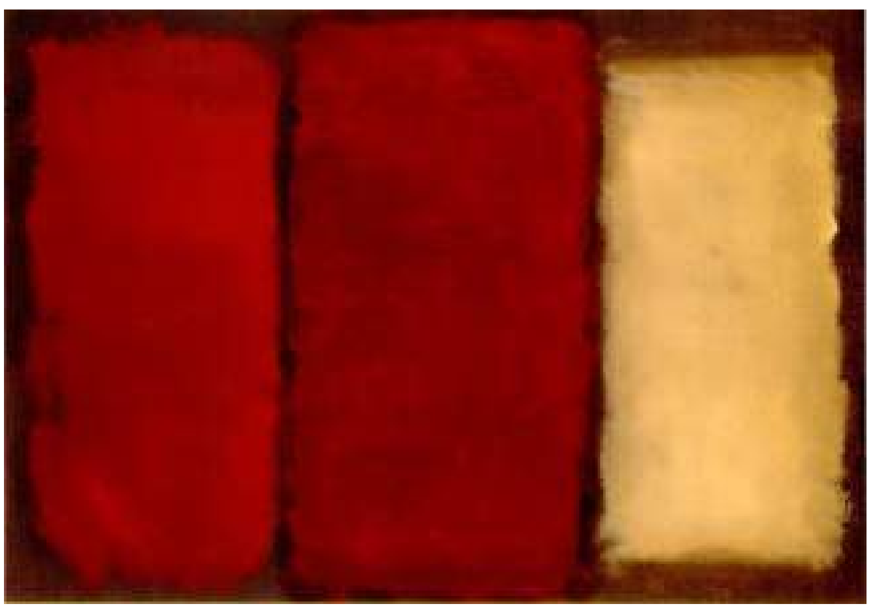
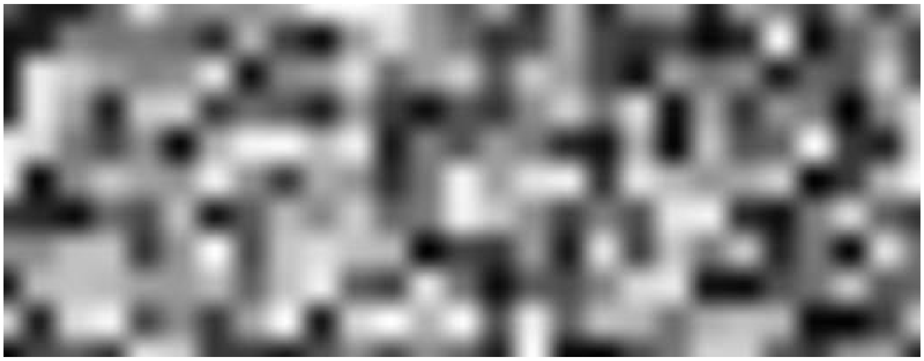


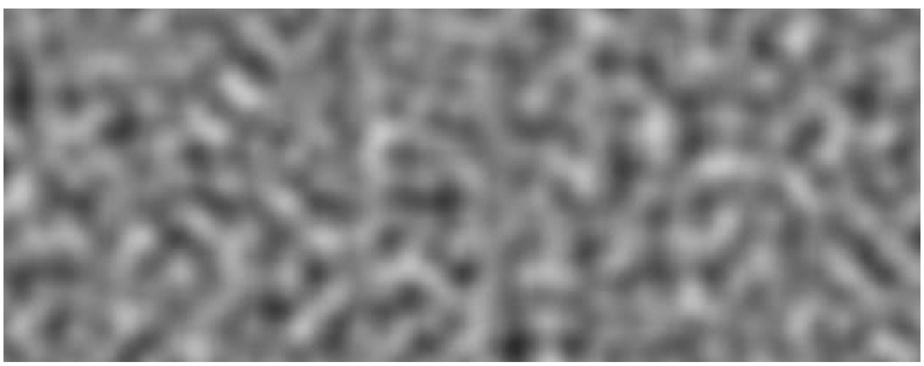
Figure 24: Mark Rothko - Three (1950)

4.8 Using Noise in Generative Designs

Noise algorithms were originally designed to give a natural *je ne sais quoi* to digital textures. The 1D and 2D implementations we've seen so far were interpolations between random *values*, which is why they're called **Value Noise**, but there are more ways to obtain noise...

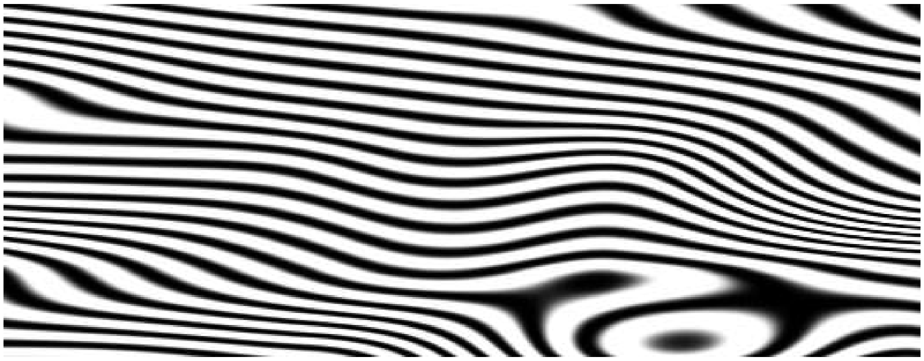


As you discovered in the previous exercises, value noise tends to look "blocky." To diminish this blocky effect, in 1985 Ken Perlin developed another implementation of the algorithm called **Gradient Noise**. Ken figured out how to interpolate random *gradients* instead of values. These gradients were the result of a 2D random function that returns directions (represented by a `vec2`) instead of single values (`float`). Click on the following image to see the code and how it works.



Take a minute to look at these two examples by Inigo Quilez and pay attention to the differences between value noise and gradient noise.

Like a painter who understands how the pigments of their paints work, the more we know about noise implementations the better we will be able to use them. For example, if we use a two dimensional noise implementation to rotate the space where straight lines are rendered, we can produce the following swirly effect that looks like wood. Again you can click on the image to see what the code looks like.



```
pos = rotate2d( noise(pos) ) * pos; // rotate the space
pattern = lines(pos,.5); // draw lines
```

Another way to get interesting patterns from noise is to treat it like a distance field and apply some of the tricks described in the Shapes chapter.

```
color += smoothstep(.15,.2,noise(st*10.)); // Black splatter
color -= smoothstep(.35,.4,noise(st*10.)); // Holes on splatter
```

A third way of using the noise function is to modulate a shape. This also requires some of the techniques we learned in the chapter about shapes.

For you to practice:

- What other generative pattern can you make? What about granite? marble? magma? water? Find three pictures of textures you are interested in and implement them algorithmically using noise.
- Use noise to modulate a shape.
- What about using noise for motion? Go back to the Matrix chapter. Use the translation example that moves the "+" around, and apply some *random* and *noise* movements to it.
- Make a generative Jackson Pollock.

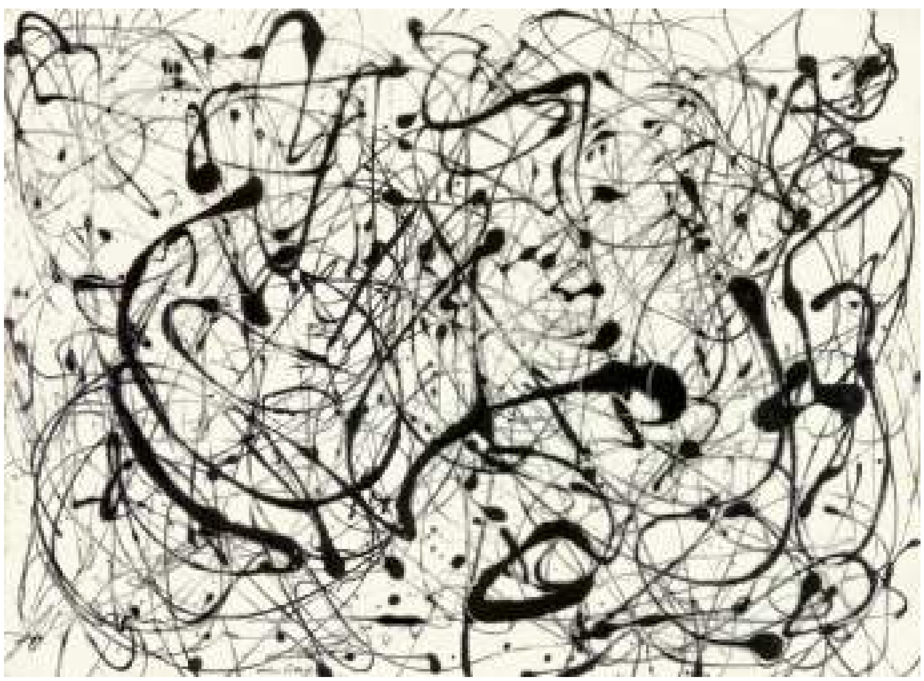


Figure 25: Jackson Pollock - Number 14 gray (1948)

4.9 Improved Noise

An improvement by Perlin to his original non-simplex noise **Simplex Noise**, is the replacement of the cubic Hermite curve ($f(x) = 3x^{2-2x}3$, which is identical to the `smoothstep()` function) with a quintic interpolation curve ($f(x) = 6x^{5-15x}4+10x\hat{}3$). This makes both ends of the curve more "flat" so each border gracefully stitches with the next one. In other words, you get a more continuous transition between the cells. You can see this by uncommenting the second formula in the following graph example (or see the two equations side by side here).

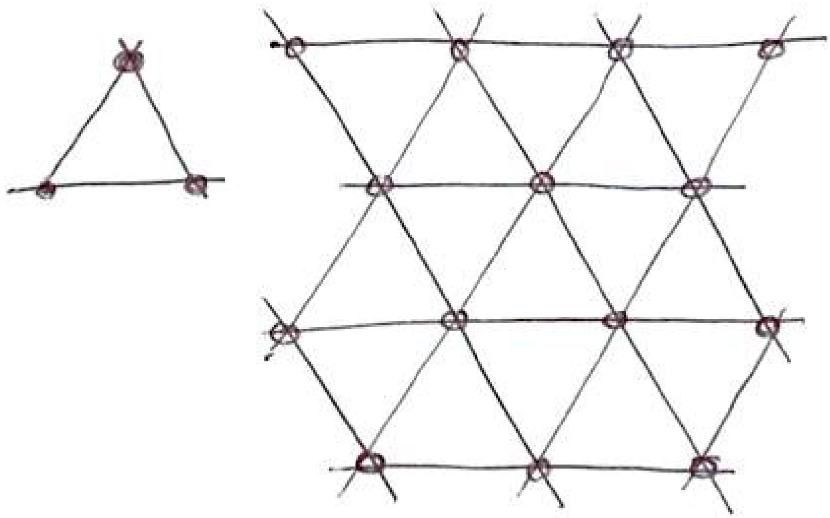
Note how the ends of the curve change. You can read more about this in Ken's own words.

4.10 Simplex Noise

For Ken Perlin the success of his algorithm wasn't enough. He thought it could perform better. At Siggraph 2001 he presented the "simplex noise" in which he achieved the following improvements over the previous algorithm:

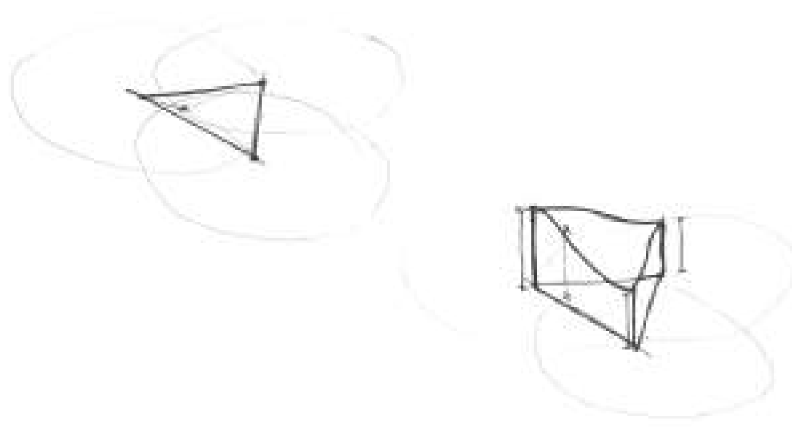
- An algorithm with lower computational complexity and fewer multiplications.
- A noise that scales to higher dimensions with less computational cost.
- A noise without directional artifacts.
- A noise with well-defined and continuous gradients that can be computed quite cheaply.
- An algorithm that is easy to implement in hardware.

I know what you are thinking… "Who is this man?" Yes, his work is fantastic! But seriously, how did he improve the algorithm? Well, we saw how for two dimensions he was interpolating 4 points (corners of a square); so we can correctly guess that for three (see an implementation here) and four dimensions we need to interpolate 8 and 16 points. Right? In other words for N dimensions you need to smoothly interpolate 2 to the N points (2^N). But Ken smartly noticed that although the obvious choice for a space-filling shape is a square, the simplest shape in 2D is the equilateral triangle. So he started by replacing the squared grid (we just learned how to use) for a simplex grid of equilateral triangles.

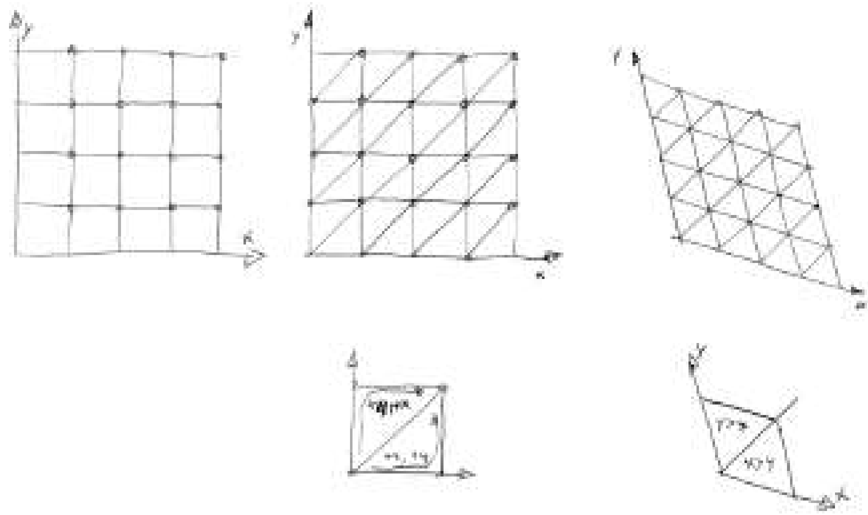


The simplex shape for N dimensions is a shape with N + 1 corners. In other words one fewer corner to compute in 2D, 4 fewer corners in 3D and 11 fewer corners in 4D! That's a huge improvement!

In two dimensions the interpolation happens similarly to regular noise, by interpolating the values of the corners of a section. But in this case, by using a simplex grid, we only need to interpolate the sum of 3 corners.



How is the simplex grid made? In another brilliant and elegant move, the simplex grid can be obtained by subdividing the cells of a regular 4 cornered grid into two isosceles triangles and then skewing it until each triangle is equilateral.



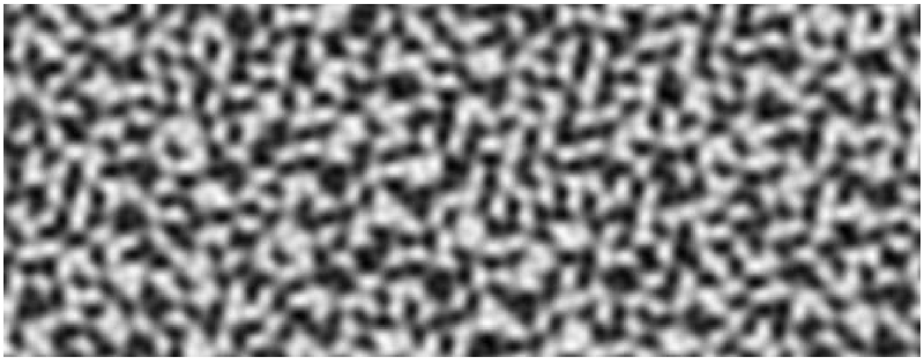
Then, as Stefan Gustavson describes in this paper: *"…by looking at the integer parts of the transformed coordinates (x,y) for the point we want to evaluate, we can quickly determine which cell of two simplices that contains the point. By also comparing the magnitudes of x and y, we can determine whether the point is in the upper or the lower simplex, and traverse the correct three corner points."*

In the following code you can uncomment line 44 to see how the grid is skewed, and then uncomment line 47 to see how a simplex grid can be constructed. Note how on line 22 we are subdividing the skewed square into two equilateral triangles just by detecting if `x > y` ("lower" triangle) or `y > x` ("upper" triangle).

```
// Author @patriciogv - 2015 - patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

vec2 skew (vec2 st) {
    vec2 r = vec2(0.0);
    r.x = 1.1547*st.x;
    r.y = st.y+0.5*r.x;
    return r;
}
```

```
vec3 simplexGrid (vec2 st) {
    vec3 xyz = vec3(0.0);

    vec2 p = fract(skew(st));
    if (p.x > p.y) {
        xyz.xy = 1.0-vec2(p.x,p.y-p.x);
        xyz.z = p.y;
    } else {
        xyz.yz = 1.0-vec2(p.x-p.y,p.y);
        xyz.x = p.x;
    }

    return fract(xyz);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    // Scale the space to see the grid
    st *= 10.;

    // Show the 2D grid
    color.rg = fract(st);

    // Skew the 2D grid
    // color.rg = fract(skew(st));

    // Subdivide the grid into to equilateral triangles
    // color = simplexGrid(st);

    gl_FragColor = vec4(color,1.0);
}
```

All these improvements result in an algorithmic masterpiece known as **Simplex Noise**. The following is a GLSL implementation of this algorithm made by Ian McEwan and Stefan Gustavson (and presented in this paper) which is overcomplicated for educational purposes, but you will be happy to click on it and see that it is less cryptic than you might expect, and the code is short and fast.



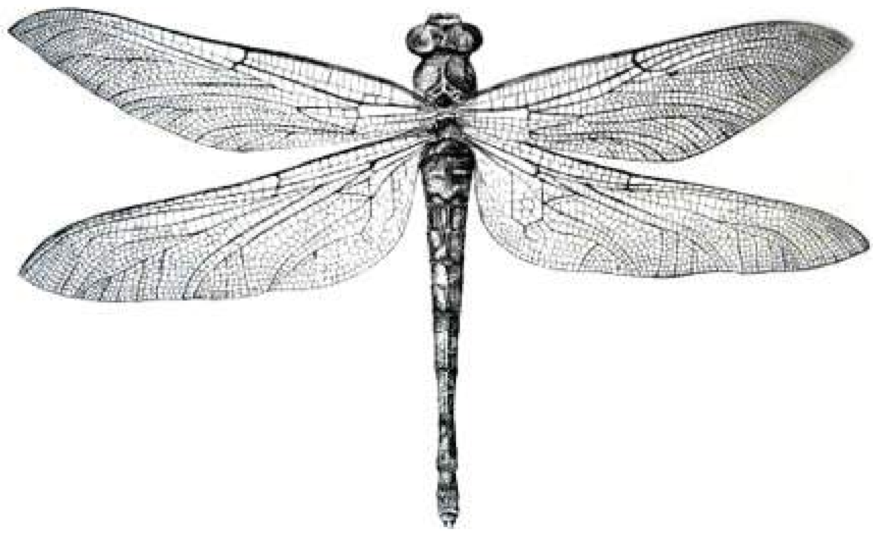
Well… enough technicalities, it's time for you to use this resource in your own expressive way:

- Contemplate how each noise implementation looks. Imagine them as a raw material, like a marble rock for a sculptor. What can you say about about the "feeling" that each one has? Squinch your eyes to trigger your imagination, like when you want to find shapes in a cloud. What do you see? What are you reminded of? What do you imagine each noise implementation could be made into? Following your guts and try to make it happen in code.
- Make a shader that projects the illusion of flow. Like a lava lamp, ink drops, water, etc.
- Use Simplex Noise to add some texture to a work you've already made.

In this chapter we have introduced some control over the chaos. It was not an easy job! Becoming a noise-bender-master takes time and effort.

In the following chapters we will see some well known techniques to perfect your skills and get more out of your noise to design quality generative content with shaders. Until then enjoy some time outside contemplating nature and its intricate patterns. Your ability to observe needs equal (or probably more) dedication than your making skills. Go outside and enjoy the rest of the day!

"Talk to the tree, make friends with it." Bob Ross



4.11 Cellular Noise

In 1996, sixteen years after Perlin's original Noise and five years before his Simplex Noise, Steven Worley wrote a paper called "A Cellular Texture Basis Function". In it, he describes a procedural texturing technique now extensively used by the graphics community.

To understand the principles behind it we need to start thinking in terms of **iterations**. Probably you know what that means: yes, start using `for` loops. There is only one catch with `for` loops in GLSL: the number we are checking against must be a constant (`const`). So, no dynamic loops - the number of iterations must be fixed.

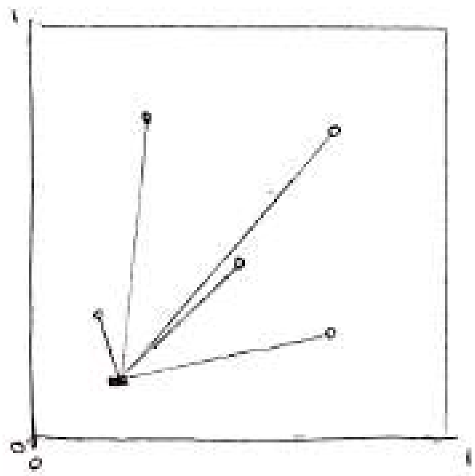
Let's take a look at an example.

4.11.1 Points for a distance field

Cellular Noise is based on distance fields, the distance to the closest one of a set of feature points. Let's say we want to make a distance field of 4 points. What do we need to do? Well, **for each pixel we want to calculate the distance to the closest point**. That means that we need to iterate through all the points, compute their distances to the current pixel and store the value for the one that is closest.

```
float min_dist = 100.; // A variable to store the closest distance to a point

min_dist = min(min_dist, distance(st, point_a));
```

```
    min_dist = min(min_dist, distance(st, point_b));
    min_dist = min(min_dist, distance(st, point_c));
    min_dist = min(min_dist, distance(st, point_d));
```



This is not very elegant, but it does the trick. Now let's re-implement it using an array and a `for` loop.

```
    float m_dist = 100.;  // minimum distance
    for (int i = 0; i < TOTAL_POINTS; i++) {
        float dist = distance(st, points[i]);
        m_dist = min(m_dist, dist);
    }
```

Note how we use a `for` loop to iterate through an array of points and keep track of the minimum distance using a `min()` function. Here's a brief working implementation of this idea:

```
// Author: @patriciogv
// Title: 4 cells DF

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
```

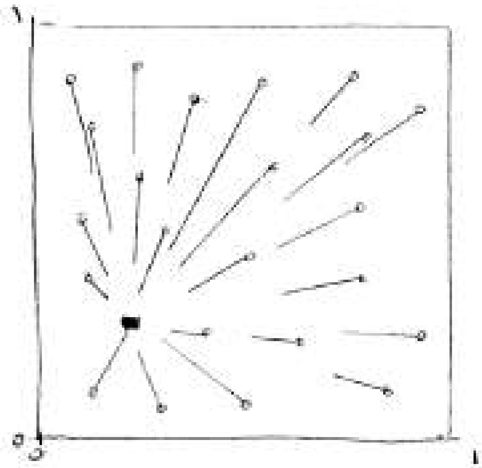
```
    st.x *= u_resolution.x/u_resolution.y;

    vec3 color = vec3(.0);

    // Cell positions
    vec2 point[5];
    point[0] = vec2(0.83,0.75);
    point[1] = vec2(0.60,0.07);
    point[2] = vec2(0.28,0.64);
    point[3] =  vec2(0.31,0.26);
    point[4] = u_mouse/u_resolution;

    float m_dist = 1.;  // minimum distance

    // Iterate through the points positions
    for (int i = 0; i < 5; i++) {
        float dist = distance(st, point[i]);

        // Keep the closer distance
        m_dist = min(m_dist, dist);
    }

    // Draw the min distance (distance field)
    color += m_dist;

    // Show isolines
    // color -= step(.7,abs(sin(50.0*m_dist)))*.3;

    gl_FragColor = vec4(color,1.0);
}
```

In the above code, one of the points is assigned to the mouse position. Play with it so you can get an intuitive idea of how this code behaves. Then try this:

- How can you animate the rest of the points?
- After reading the chapter about shapes, imagine interesting ways to use this distance field!
- What if you want to add more points to this distance field? What if we want to dynamically add/subtract points?

4.11.2 Tiling and iteration

You probably notice that `for` loops and *arrays* are not very good friends with GLSL. Like we said before, loops don't accept dynamic limits on their exit condition. Also, iterating through a lot of instances reduces the performance of your shader significantly. That means we can't use this direct approach for large amounts of points. We need to find another strategy, one that takes advantage of the parallel processing architecture of the GPU.



One way to approach this problem is to divide the space into tiles. Not every pixel needs to check the distance to every single point, right? Given the fact that each pixel runs in its own thread, we can subdivide the space into cells, each one with one unique point to watch. Also, to avoid aberrations at the edges between cells we need to check for the distances to the points on the neighboring cells. That's the main brillant idea of Steven Worley's paper. At the end, each pixel needs to check only nine positions: their own cell's point and the points in the 8 cells around it. We already subdivide the space into cells in the chapters about: patterns, random and noise, so hopefully you are familiar with this technique by now.

```
// Scale
st *= 3.;

// Tile the space
vec2 i_st = floor(st);
vec2 f_st = fract(st);
```

So, what's the plan? We will use the tile coordinates (stored in the integer coordinate, `i_st`) to construct a random position of a point. The `random2f` function we will use receives a `vec2` and gives us a `vec2` with a random position. So, for each tile we will have one feature point in a random position within the tile.

```
vec2 point = random2(i_st);
```

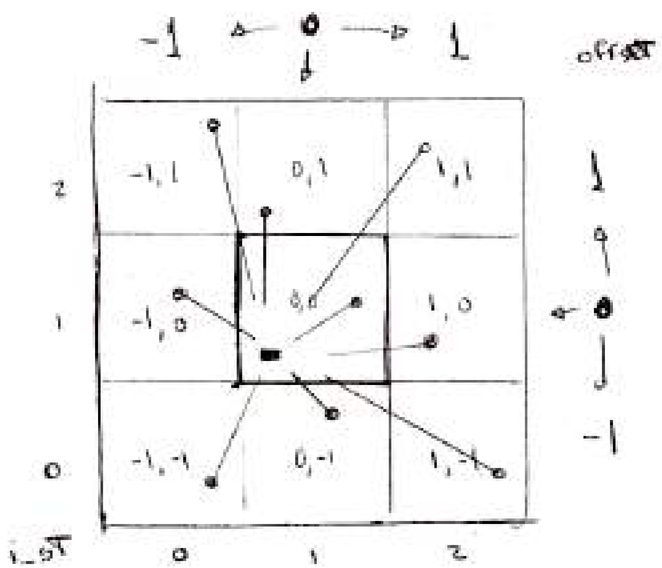
Each pixel inside that tile (stored in the float coordinate, `f_st`) will check their distance to that random point.

```
    vec2 diff = point - f_st;
    float dist = length(diff);
```

The result will look like this:

We still need to check the distances to the points in the surrounding tiles, not just the one in the current tile. For that we need to **iterate** through the neighbor tiles. Not all tiles, just the ones immediately around the current one. That means from `-1` (left) to `1` (right) tile in `x` axis and `-1` (bottom) to `1` (top) in `y` axis. A 3x3 region of 9 tiles can be iterated through using a double `for` loop like this one:

```
for (int y= -1; y <= 1; y++) {
    for (int x= -1; x <= 1; x++) {
        // Neighbor place in the grid
        vec2 neighbor = vec2(float(x),float(y));
        ...
    }
}
```



Now, we can compute the position of the points on each one of the neighbors in our double `for` loop by adding the neighbor tile offset to the current tile coordinate.

```
        ...
        // Random position from current + neighbor place in the grid
        vec2 point = random2(i_st + neighbor);
        ...
```

The rest is all about calculating the distance to that point and storing the closest one in a variable called `m_dist` (for minimum distance).

```
...
vec2 diff = neighbor + point - f_st;

// Distance to the point
float dist = length(diff);

// Keep the closer distance
m_dist = min(m_dist, dist);
...
```

The above code is inspired by this article by Inigo's Quilez where he said:

"… it might be worth noting that there's a nice trick in this code above. Most implementations out there suffer from precision issues, because they generate their random points in"domain" space (like "world" or "object" space), which can be arbitrarily far from the origin. One can solve the issue moving all the code to higher precision data types, or by being a bit clever. My implementation does not generate the points in "domain" space, but in "cell" space: once the integer and fractional parts of the shading point are extracted and therefore the cell in which we are working identified, all we care about is what happens around this cell, meaning we can drop all the integer part of our coordinates away all together, saving many precision bits. In fact, in a regular voronoi implementation the integer parts of the point coordinates simply cancel out when the random per cell feature points are subtracted from the shading point. In the implementation above, we don't even let that cancelation happen, cause we are moving all the computations to "cell" space. This trick also allows one to handle the case where you want to voronoi-shade a whole planet - one could simply replace the input to be double precision, perform the floor() and fract() computations, and go floating point with the rest of the computations without paying the cost of changing the whole implementation to double precision. Of course, same trick applies to Perlin Noise patterns (but i've never seen it implemented nor documented anywhere)."

Recapping: we subdivide the space into tiles; each pixel will calculate the distance to the point in their own tile and the surrounding 8 tiles; store the closest distance. The result is a distance field that looks like the following example:

```
// Author: @patriciogv
// Title: CellularNoise

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
```

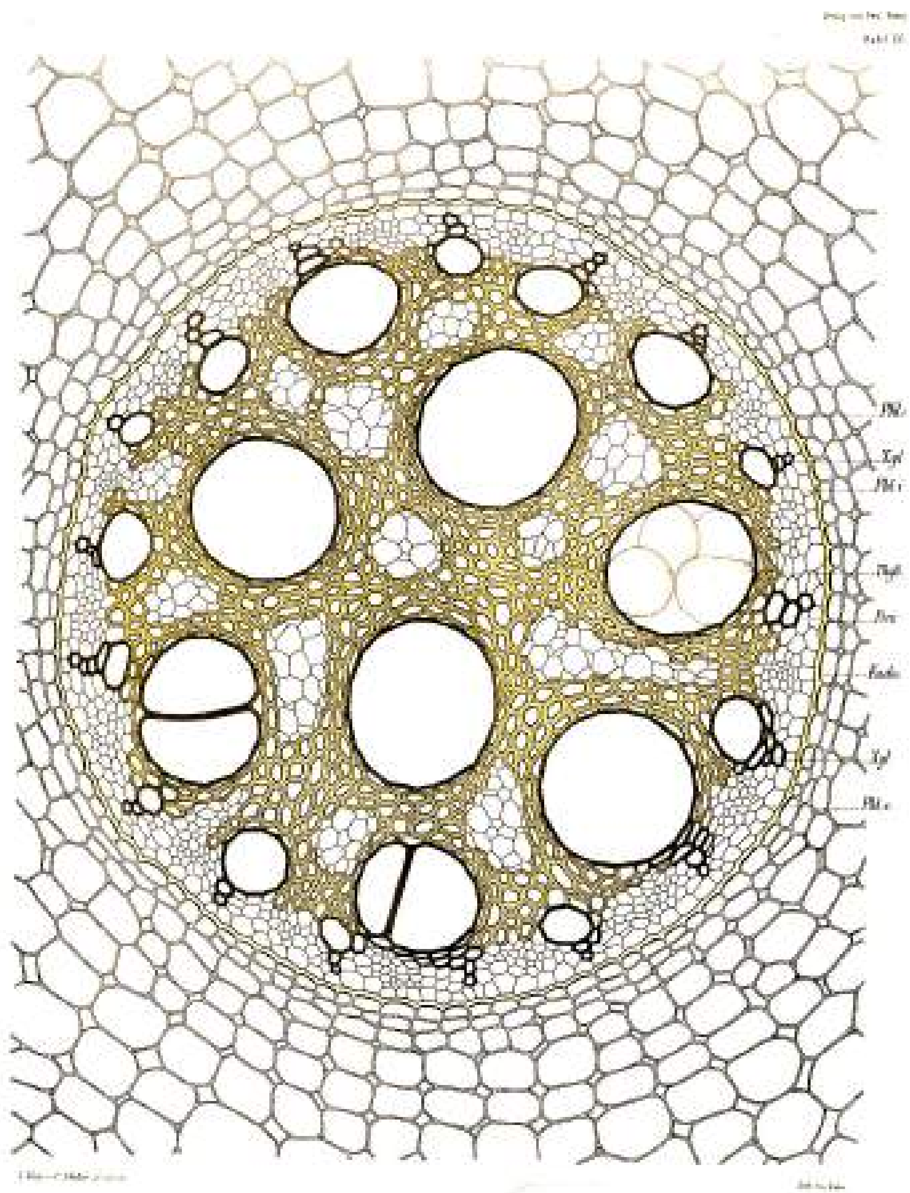
```
uniform float u_time;

vec2 random2( vec2 p ) {
    return fract(sin(vec2(dot(p,vec2(127.1,311.7)),dot(p,vec2(269.5,183.3))))*43758.5453);
}

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st.x *= u_resolution.x/u_resolution.y;
    vec3 color = vec3(.0);

    // Scale
    st *= 3.;

    // Tile the space
    vec2 i_st = floor(st);
    vec2 f_st = fract(st);

    float m_dist = 1.;  // minimum distance

    for (int y= -1; y <= 1; y++) {
        for (int x= -1; x <= 1; x++) {
            // Neighbor place in the grid
            vec2 neighbor = vec2(float(x),float(y));

            // Random position from current + neighbor place in the grid
            vec2 point = random2(i_st + neighbor);

                        // Animate the point
            point = 0.5 + 0.5*sin(u_time + 6.2831*point);

                        // Vector between the pixel and the point
            vec2 diff = neighbor + point - f_st;

            // Distance to the point
            float dist = length(diff);

            // Keep the closer distance
            m_dist = min(m_dist, dist);
        }
    }

    // Draw the min distance (distance field)
    color += m_dist;

    // Draw cell center
```

```
    color += 1.-step(.02, m_dist);

    // Draw grid
    color.r += step(.98, f_st.x) + step(.98, f_st.y);

    // Show isolines
    // color -= step(.7,abs(sin(27.0*m_dist)))*.5;

    gl_FragColor = vec4(color,1.0);
}
```

Explore this further by:

- Scaling the space by different values.
- Can you think of other ways to animate the points?
- What if we want to compute an extra point with the mouse position?
- What other ways of constructing this distance field can you imagine, besides `m_dist = min(m_dist, dist);`?
- What interesting patterns can you make with this distance field?

This algorithm can also be interpreted from the perspective of the points and not the pixels. In that case it can be described as: each point grows until it finds the growing area from another point. This mirrors some of the growth rules in nature. Living forms are shaped by this tension between an inner force to expand and grow, and limitations by outside forces. The classic algorithm that simulates this behavior is named after Georgy Voronoi.



4.11.3 Voronoi Algorithm

Constructing Voronoi diagrams from cellular noise is less hard than what it might seem. We just need to *keep* some extra information about the precise point which is closest to the pixel. For that we are going to use a `vec2` called `m_point`. By storing the vector direction to the center of the closest point, instead of just the distance, we will be "keeping" a "unique" identifier of that point.

```
    ...
    if( dist < m_dist ) {
        m_dist = dist;
        m_point = point;
    }
    ...
```

Note that in the following code that we are no longer using `min` to calculate the closest distance, but a regular `if` statement. Why? Because we actually want to do something more every time a new closer point appears, namely store its position (lines 32 to 37).

```
// Author: @patriciogv
// Title: 4 cells voronoi

#ifdef GL_ES
precision mediump float;
#endif

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main() {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    st.x *= u_resolution.x/u_resolution.y;

    vec3 color = vec3(.0);

    // Cell positions
    vec2 point[5];
    point[0] = vec2(0.83,0.75);
    point[1] = vec2(0.60,0.07);
    point[2] = vec2(0.28,0.64);
    point[3] =  vec2(0.31,0.26);
    point[4] = u_mouse/u_resolution;

    float m_dist = 1.;  // minimum distance
    vec2 m_point;        // minimum position

    // Iterate through the points positions
    for (int i = 0; i < 5; i++) {
        float dist = distance(st, point[i]);
        if ( dist < m_dist ) {
            // Keep the closer distance
            m_dist = dist;
```

```
            // Kepp the position of the closer point
            m_point = point[i];
        }
    }

    // Add distance field to closest point center
    color += m_dist*2.;

    // tint acording the closest point position
    color.rg = m_point;

    // Show isolines
    color -= abs(sin(80.0*m_dist))*0.07;

    // Draw point center
    color += 1.-step(.02, m_dist);

    gl_FragColor = vec4(color,1.0);
}
```

Note how the color of the moving cell (bound to the mouse position) changes color according to its position. That's because the color is assigned using the value (position) of the closest point.

Like we did before, now is the time to scale this up, switching to Steven Worley's paper's approach. Try implementing it yourself. You can use the help of the following example by clicking on it. Note that Steven Worley's original approach uses a variable number of feature points for each tile, more than one in most tiles. In his software implementation in C, this is used to speed up the loop by making early exits. GLSL loops don't allow variable number of iterations, so you probably want to stick to one feature point per tile.

Once you figure out this algorithm, think of interesting and creative uses for it.

4.11.4 Improving Voronoi

In 2011, Stefan Gustavson optimized Steven Worley's algorithm to GPU by only iterating through a 2x2 matrix instead of 3x3. This reduces the amount of work significantly, but it can create artifacts in the form of discontinuities at the edges between the tiles. Take a look to the following examples.

Later in 2012 Inigo Quilez wrote an article on how to make precise Voronoi borders.

Inigo's experiments with Voronoi didn't stop there. In 2014 he wrote this nice article about what he calls voro-noise, a function that allows a gradual blend between regular noise and voronoi. In his words:

"Despite this similarity, the fact is that the way the grid is used in both patterns is

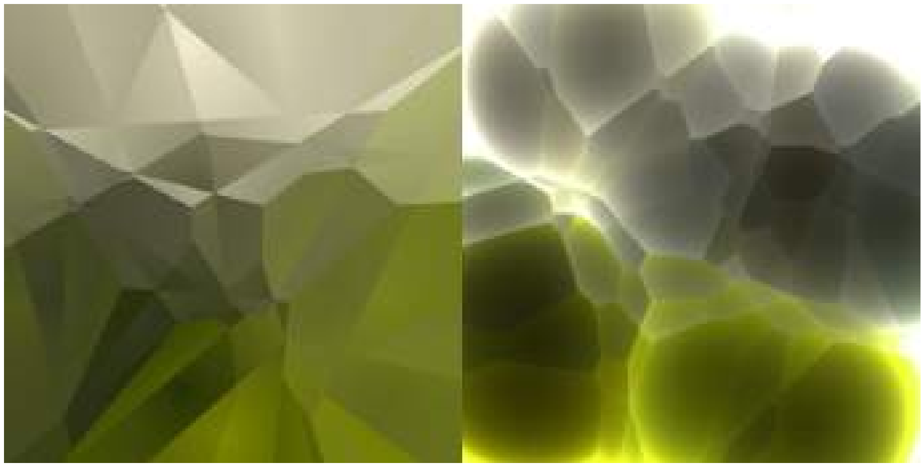


Figure 26: Extended Voronoi - Leo Solaas (2011)

different. Noise interpolates/averages random values (as in value noise) or gradients (as in gradient noise), while Voronoi computes the distance to the closest feature point. Now, smooth-bilinear interpolation and minimum evaluation are two very different operations, or… are they? Can they perhaps be combined in a more general metric? If that was so, then both Noise and Voronoi patterns could be seen as particular cases of a more general grid-based pattern generator?"

Now it's time for you to look closely at things, be inspired by nature and find your own take on this technique!

4.12 Fractal Brownian Motion

Noise tends to mean different things to different people. Musicians will think of it in terms of disturbing sounds, communicators as interference and astrophysicists as cosmic microwave background radiation. These concepts bring us back to the physical reasons behind randomness in the world around us. However, let's start with something more fundamental, and more simple: waves and their properties. A wave is a fluctuation over time of some property. Audio waves are fluctuations in air pressure, electromagnetical waves are fluctuations in electrical and magnetic fields. Two important characteristics of a wave are its amplitude and frequency. The equation for a simple linear (one-dimensional) wave looks like this:

- Try changing the values of the frequency and amplitude to understand how they behave.
- Using shaping functions, try changing the amplitude over time.
- Using shaping functions, try changing the frequency over time.

By doing the last two exercises you have managed to "modulate" a sine wave,

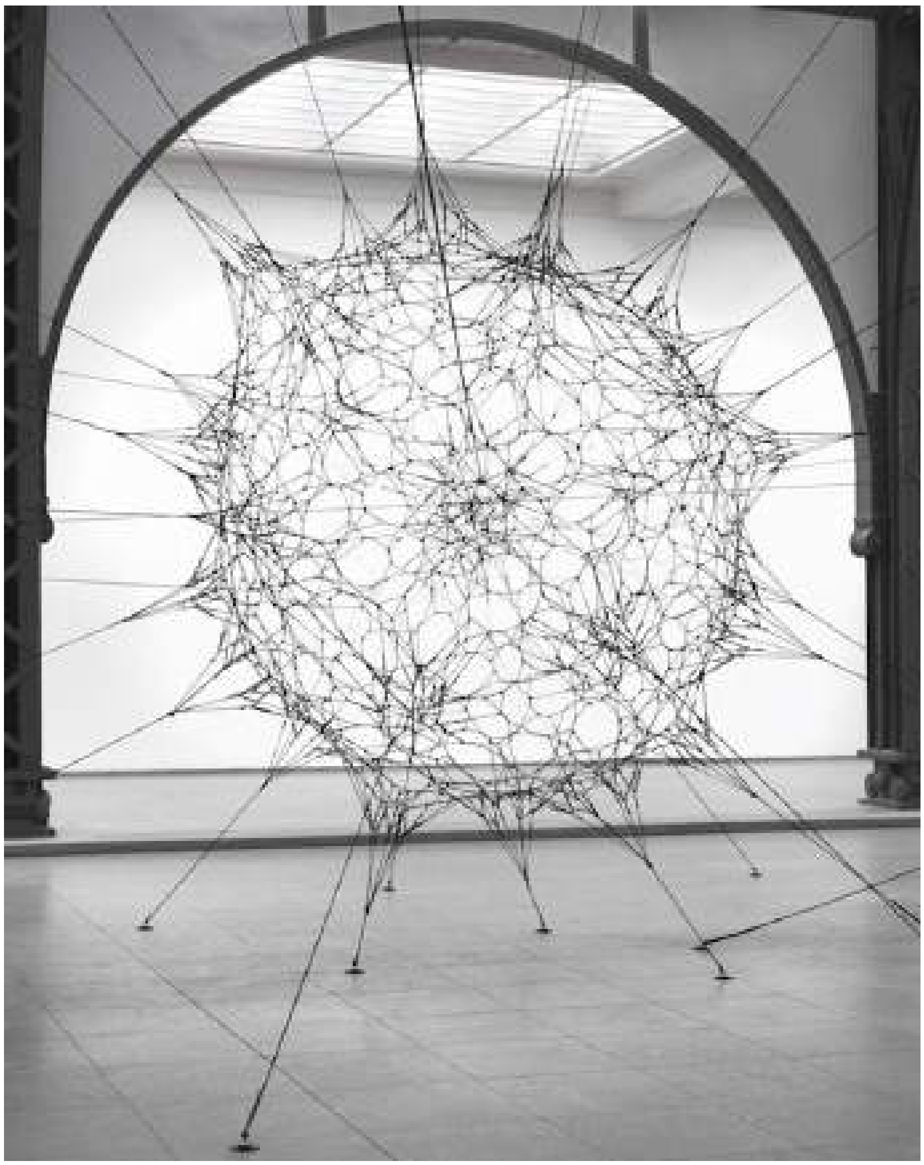


Figure 27: Cloud Cities - Tomás Saraceno (2011)

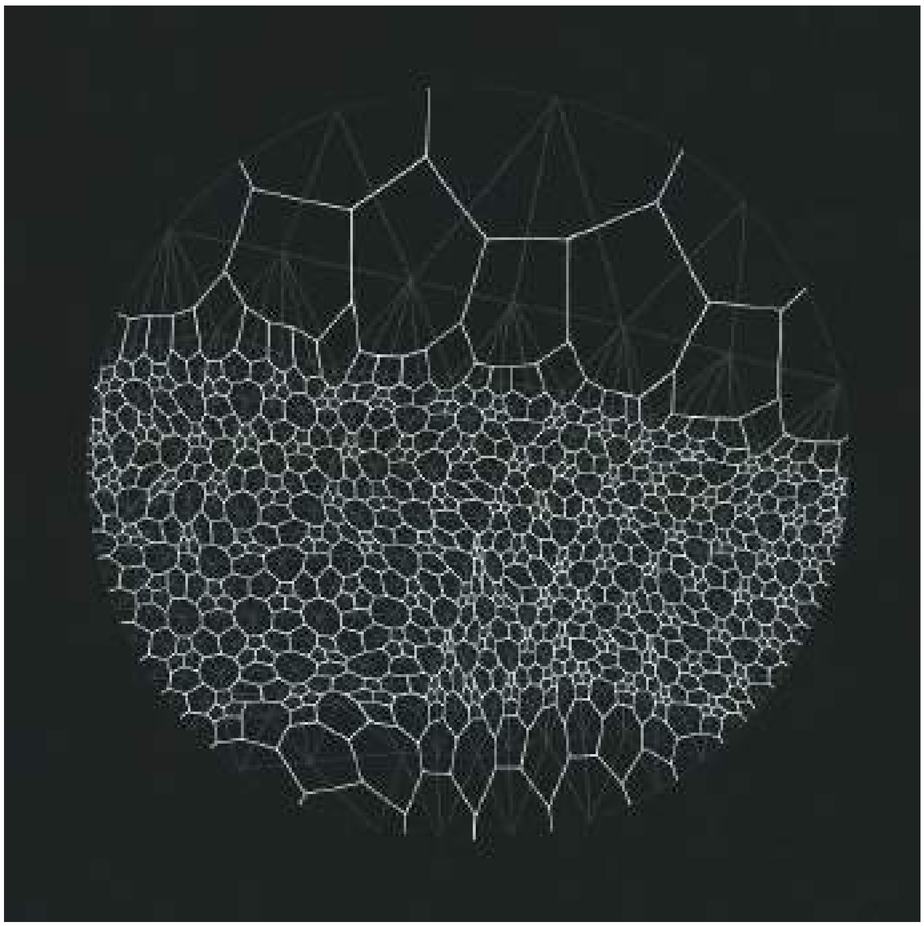


Figure 28: Accretion Disc Series - Clint Fulkerson

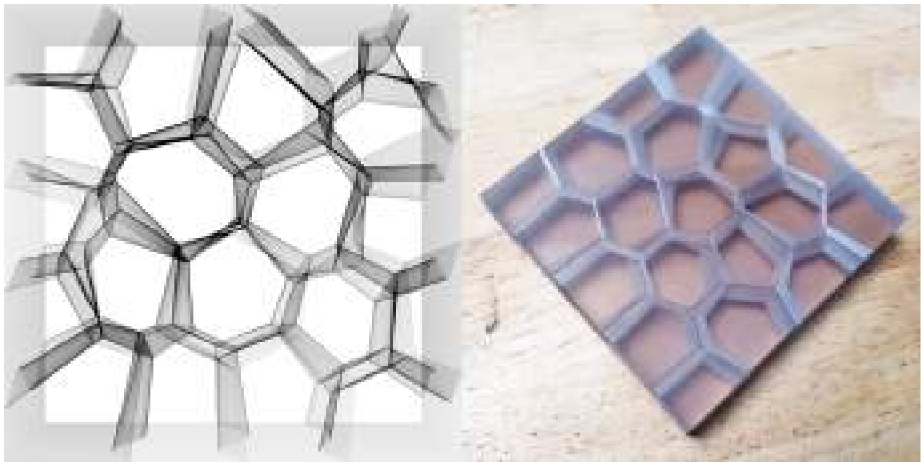


Figure 29: Vonoroi Puzzle - Reza Ali (2015)

Figure 30: Deyrolle glass film - 1831

Figure 31: Due East over Shadequarter Mountain - Matthew Rangel (2005)

and you just created AM (amplitude modulated) and FM (frequency modulated) waves. Congratulations!

Another interesting property of waves is their ability to add up, which is formally called superposition. Comment/uncomment and tweak the following lines. Pay attention to how the overall appearance changes as we add waves of different amplitudes and frequencies together.

- Experiment by changing the frequency and amplitude for the additional waves.
- Is it possible to make two waves cancel each other out? What will that look like?
- Is it possible to add waves in such a way that they will amplify each other?

In music, each note is associated with a specific frequency. The frequencies for these notes follow a pattern which we call a scale, where a doubling or halving of the frequency corresponds to a jump of one octave.

Now, let's use Perlin noise instead of a sine wave! Perlin noise in its basic form has the same general look and feel as a sine wave. Its amplitude and frequency vary somewhat, but the amplitude remains reasonably consistent, and the frequency is restricted to a fairly narrow range around a center frequency. It's not as regular as a sine wave, though, and it's easier to create an appearance of randomness by summing up several scaled versions of noise. It is possible to make a sum of sine waves appear random as well, but it takes many different waves to hide their periodic, regular nature.

By adding different iterations of noise (*octaves*), where we successively increment the frequencies in regular steps (*lacunarity*) and decrease the amplitude (*gain*) of the **noise** we can obtain a finer granularity in the noise and get more fine detail. This technique is called "fractal Brownian Motion" (*fBM*), or simply "fractal noise", and in its simplest form it can be created by the following code:

- Progressively change the number of octaves to iterate from 1 to 2, 4, 8 and 10. See what happens.
- When you have more than 4 octaves, try changing the lacunarity value.
- Also with >4 octaves, change the gain value and see what happens.

Note how with each additional octave, the curve seems to get more detail. Also note the self-similarity while more octaves are added. If you zoom in on the curve, a smaller part looks about the same as the whole thing, and each section looks more or less the same as any other section. This is an important property of mathematical fractals, and we are simulating that property in our loop. We are not creating a *true* fractal, because we stop the summation after a few iterations, but theoretically speaking, we would get a true mathematical fractal if we allowed the loop to continue forever and add an infinite number of noise components. In computer graphics, we always have a limit to the smallest details we can resolve, for example when objects become smaller than a pixel, so there is no need to make infinite sums to create the appearance of a fractal. A lot of terms may be needed sometimes, but never an infinite number.

The following code is an example of how fBm could be implemented in two dimensions to create a fractal-looking pattern:

- Reduce the number of octaves by changing the value on line 37
- Modify the lacunarity of the fBm on line 47
- Explore by changing the gain on line 48

This technique is commonly used to construct procedural landscapes. The self-similarity of the fBm is perfect for mountains, because the erosion processes that create mountains work in a manner that yields this kind of self-similarity across a large range of scales. If you are interested in this use, you should definitely read this great article by Inigo Quiles about advanced noise.

Using more or less the same technique, it's also possible to obtain other effects like what is known as **turbulence**. It's essentially an fBm, but constructed from the absolute value of a signed noise to create sharp valleys in the function.

```
for (int i = 0; i < OCTAVES; i++) {
    value += amplitude * abs(snoise(st));
    st *= 2.;
    amplitude *= .5;
}
```

Figure 32: Blackout - Dan Holdsworth (2010)

Another member of this family of algorithms is the **ridge**, where the sharp valleys are turned upside down to create sharp ridges instead:

```
n = abs(n);     // create creases
n = offset - n; // invert so creases are at top
n = n * n;      // sharpen creases
```

Another variant which can create useful variations is to multiply the noise components together instead of adding them. It's also interesting to scale subsequent noise functions with something that depends on the previous terms in the loop. When we do things like that, we are moving away from the strict definition of a fractal and into the relatively unknown field of "multifractals". Multifractals are not as strictly defined mathematically, but that doesn't make them less useful for graphics. In fact, multifractal simulations are very common in modern commercial software for terrain generation. For further reading, you could read chapter 16 of the book "Texturing and Modeling: a Procedural Approach" (3rd edition), by Kenton Musgrave. Sadly, that book is out of print since a few years back, but you can still find it in libraries and on the second hand market. (There's a PDF version of the 1st edition available for purchase online, but don't buy that - it's a waste of money. It's from 1994, and it doesn't contain any of the terrain modeling stuff from the 3rd edition.)

4.12.1 Domain Warping

Inigo Quiles wrote this other fascinating article about how it's possible to use fBm to warp a space of a fBm. Mind blowing, Right? It's like the dream inside the dream of Inception.

Figure 33: f(p) = fbm(p + fbm(p + fbm(p))) - Inigo Quiles (2002)

A less extreme example of this technique is the following code where the wrap is used to produce this clouds-like texture. Note how the self-similarity property is still present in the result.

Warping the texture coordinates with noise in this manner can be very useful, a lot of fun, and fiendishly difficult to master. It's a powerful tool, but it takes quite a bit of experience to use it well. A useful tool for this is to displace the coordinates with the derivative (gradient) of the noise. A famous article by Ken Perlin and Fabrice Neyret called "flow noise" is based on this idea. Some modern implementations of Perlin noise include a variant that computes both the function and its analytical gradient. If the "true" gradient is not available for a procedural function, you can always compute finite differences to approximate it, although this is less accurate and involves more work.

4.13 Fractals

https://www.shadertoy.com/view/lsX3W4

https://www.shadertoy.com/view/Mss3Wf

https://www.shadertoy.com/view/4df3Rn

https://www.shadertoy.com/view/Mss3R8

https://www.shadertoy.com/view/4dfGRn

https://www.shadertoy.com/view/lss3zs

https://www.shadertoy.com/view/4dXGDX

https://www.shadertoy.com/view/XsXGz2

https://www.shadertoy.com/view/lls3D7

https://www.shadertoy.com/view/XdB3DD

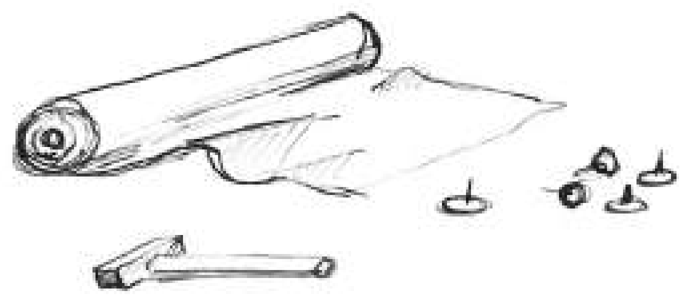
https://www.shadertoy.com/view/XdBSWw

https://www.shadertoy.com/view/llfGD2

https://www.shadertoy.com/view/Mlf3RX

5 Image processing

5.1 Textures



Graphic cards (GPUs) have special memory types for images. Usually on CPUs images are stored as arrays of bytes but GPUs store images as `sampler2D` which is more like a table (or matrix) of floating point vectors. More interestingly, the values of this *table* of vectors are continuous. That means that values between pixels are interpolated in a low level.

In order to use this feature we first need to *upload* the image from the CPU to the GPU, to then pass the `id` of the texture to the right `uniform`. All that happens outside the shader.

Once the texture is loaded and linked to a valid `uniform sampler2D` you can ask for specific color value at specific coordinates (formated on a `vec2` variable) using the `texture2D()` function which will return a color formatted on a `vec4` variable.

```
vec4 texture2D(sampler2D texture, vec2 coordinates)
```

Check the following code where we load Hokusai's Wave (1830) as `uniform sampler2D u_tex0` and we call every pixel of it inside the billboard:

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform sampler2D u_tex0;
uniform vec2 u_tex0Resolution;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
```

```
uniform float u_time;

void main () {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec4 color = vec4(st.x,st.y,0.0,1.0);

    color = texture2D(u_tex0,st);

    gl_FragColor = color;
}
```

If you pay attention you will note that the coordinates for the texture are normalized! What a surprise right? Textures coordinates are consistent with the rest of the things we had seen and their coordinates are between 0.0 and 1.0 which match perfectly with the normalized space coordinates we have been using.

Now that you have seen how we correctly load a texture, it is time to experiment to discover what we can do with it, by trying:

- Scaling the previous texture by half.
- Rotating the previous texture 90 degrees.
- Hooking the mouse position to the coordinates to move it.

Why you should be excited about textures? Well first of all forget about the sad 255 values for channel; once your image is transformed into a `uniform sampler2D` you have all the values between 0.0 and 1.0 (depending on what you set the `precision` to). That's why shaders can make really beautiful post-processing effects.

Second, the `vec2()` means you can get values even between pixels. As we said before the textures are a continuum. This means that if you set up your texture correctly you can ask for values all around the surface of your image and the values will smoothly vary from pixel to pixel with no jumps!

Finally, you can set up your image to repeat in the edges, so if you give values over or lower of the normalized 0.0 and 1.0, the values will wrap around starting over.

All these features make your images more like an infinite spandex fabric. You can stretch and shrink your texture without noticing the grid of bytes they are originally composed of or the ends of it. To experience this take a look at the following code where we distort a texture using the noise function we already made.

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
```

```
#endif

#define PI 3.14159265359

uniform sampler2D u_tex0;
uniform vec2 u_tex0Resolution;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

// Based on Morgan
// https://www.shadertoy.com/view/4dS3Wd
float random (in vec2 st) {
    return fract(sin(dot(st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}

float noise (in vec2 st) {
    vec2 i = floor(st);
    vec2 f = fract(st);

    // Four corners in 2D of a tile
    float a = random(i);
    float b = random(i + vec2(1.0, 0.0));
    float c = random(i + vec2(0.0, 1.0));
    float d = random(i + vec2(1.0, 1.0));

    vec2 u = f * f * (3.0 - 2.0 * f);

    return mix(a, b, u.x) +
            (c - a)* u.y * (1.0 - u.x) +
            (d - b) * u.x * u.y;
}

void main () {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;

    float scale = 2.0;
    float offset = 0.5;

    float angle = noise( st + u_time * 0.1 )*PI;
    float radius = offset;

    st *= scale;
```

```
    st += radius * vec2(cos(angle),sin(angle));

    vec4 color = texture2D(u_tex0,st);

    gl_FragColor = color;
}
```

5.2 Texture resolution

Above examples play well with squared images, where both sides are equal and match our squared billboard. But for non-squared images things can be a little more tricky, and unfortunately centuries of pictorial art and photography found more pleasant to the eye non-squared proportions for images.

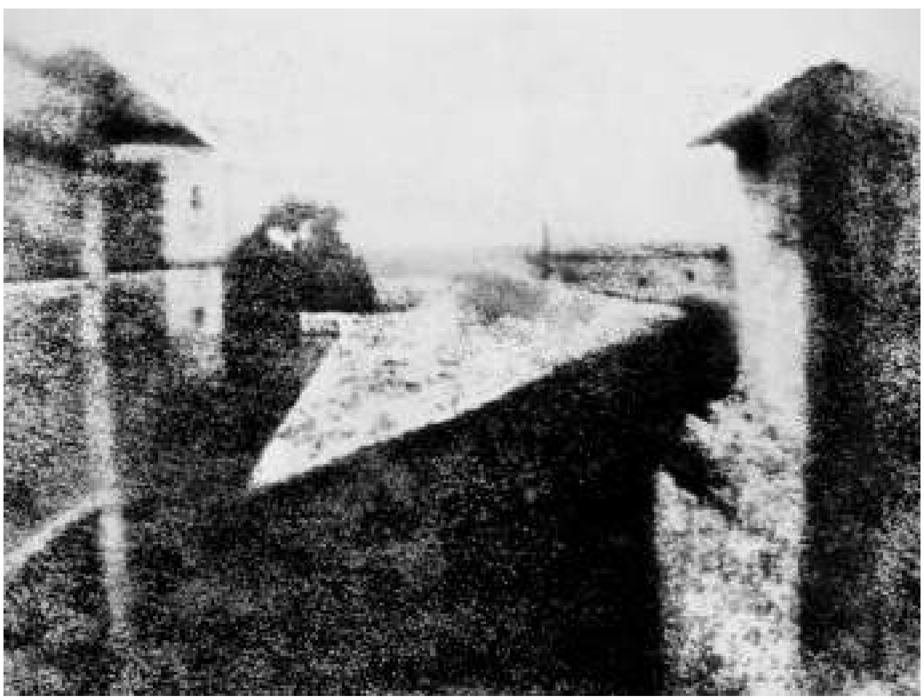


Figure 34: Joseph Nicéphore Niépce (1826)

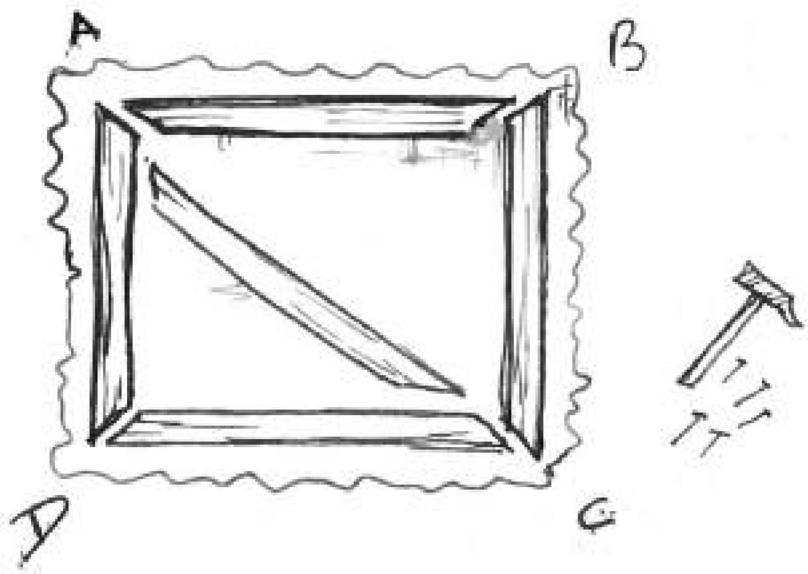
How we can solve this problem? Well we need to know the original proportions of the image to know how to stretch the texture correctly in order to have the original *aspect ratio*. For that the texture width and height are passed to the shader as an `uniform`, which in our example framework are passed as an `uniform vec2` with the same name of the texture followed with proposition `Resolution`. Once we have this information on the shader we can get the aspect ratio by dividing the `width` for the `height` of the texture resolution. Finally by multiplying this ratio to the coordinates on `y` we will shrink this axis to match the original proportions.

Uncomment line 21 of the following code to see this in action.

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform sampler2D u_tex0;
uniform vec2 u_tex0Resolution;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main () {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec4 color = vec4(0.0);

    // Fix the proportions by finding the aspect ratio
    float aspect = u_tex0Resolution.x/u_tex0Resolution.y;
    // st.y *= aspect;  // and then applying to it

    color = texture2D(u_tex0,st);

    gl_FragColor = color;
}
```

- What we need to do to center this image?

5.3 Digital upholstery



You may be thinking that this is unnecessarily complicated… and you are probably right. Also this way of working with images leaves enough room to different hacks and creative tricks. Try to imagine that you are an upholster and by stretching and folding a fabric over a structure you can create better and new patterns and techniques.

This level of craftsmanship links back to some of the first optical experiments ever made. For example on games *sprite animations* are very common, and is inevitably to see on it reminiscence to phenakistoscope, zoetrope and praxinoscope.

This could seem simple but the possibilities of modifying textures coordinates are enormous. For example:

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform sampler2D u_tex0;
uniform vec2 u_tex0Resolution;
```

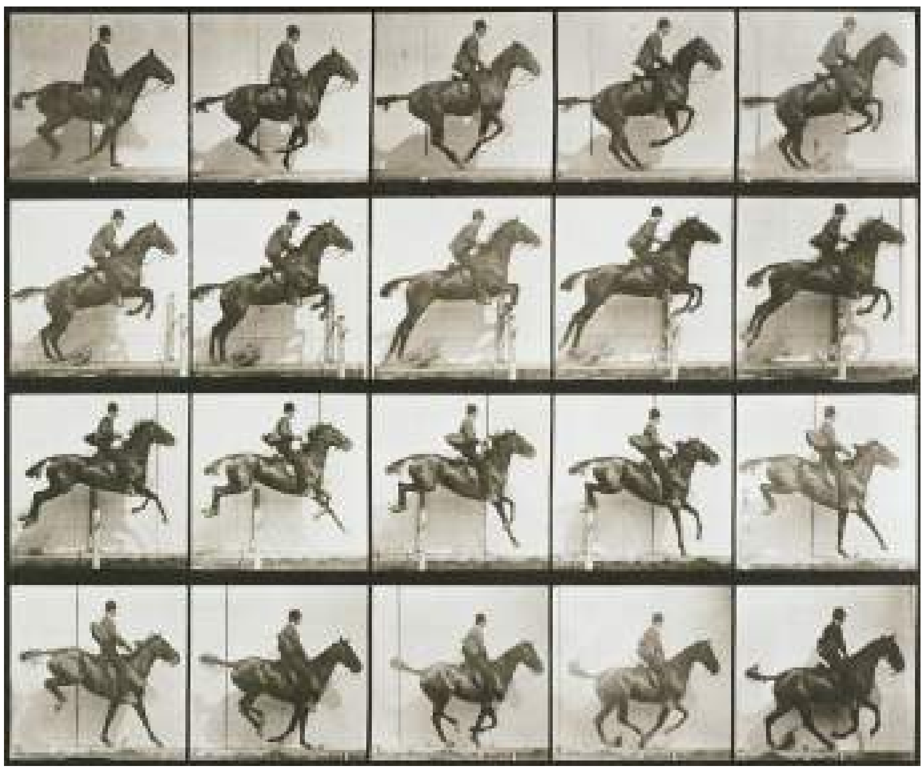


Figure 35: Eadweard's Muybridge study of motion

```
int col = 5;
int row = 4;

uniform vec2 u_resolution;
uniform vec2 u_mouse;
uniform float u_time;

void main () {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec4 color = vec4(0.0);

    // Resolution of one frame
    vec2 fRes = u_tex0Resolution/vec2(float(col),float(row));

    // Normalize value of the frame resolution
    vec2 nRes = u_tex0Resolution/fRes;

    // Scale the coordinates to a single frame
    st = st/nRes;

    // Calculate the offset in cols and rows
    float timeX = u_time*15.;
    float timeY = floor(timeX/float(col));
    vec2 offset = vec2( floor(timeX)/nRes.x,
                        1.0-(floor(timeY)/nRes.y) );
    st = fract(st+offset);

    color = texture2D(u_tex0,st);

    gl_FragColor = color;
}
```

Now is your turn:

- Can you make a kaleidoscope using what we have learned?
- Way before Oculus or google cardboard, stereoscopic photography was a big thing. Could you code a simple shader to re-use these beautiful images?
- What other optical toys can you re-create using textures?

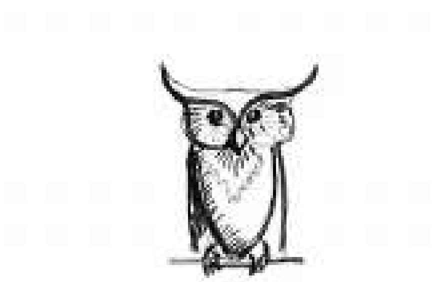
In the next chapters we will learn how to do some image processing using shaders. You will note that finally the complexity of shader makes sense, because it was in a big sense designed to do this type of process. We will start doing some image operations!

5.4 Image operations

5.4.1 Invert

```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
precision mediump float;
#endif

uniform sampler2D u_tex0;

uniform float u_time;
uniform vec2 u_mouse;
uniform vec2 u_resolution;

void main (void) {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = texture2D(u_tex0,st).rgb;

    // color = 1.0-color;

    gl_FragColor = vec4(color,1.0);
}
```

5.4.2 Add, Substract, Multiply and others



```
// Author @patriciogv - 2015
// http://patriciogonzalezvivo.com

#ifdef GL_ES
```

```
precision mediump float;
#endif

uniform sampler2D u_tex0;
uniform sampler2D u_tex1;

uniform float u_time;
uniform vec2 u_mouse;
uniform vec2 u_resolution;

void main (void) {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.);
    vec3 colorA = texture2D(u_tex0,st).rgb;
    vec3 colorB = texture2D(u_tex1,st).rgb;

    color = colorA+colorB;      // Add
    // color = colorA-colorB;      // Diff
    // color = abs(colorA-colorB); // Abs Diff
    // color = colorA*colorB;      // Mult
    // color = colorA/colorB;      // Div
    // color = max(colorA,colorB); // Ligther
    // color = min(colorA,colorB); // Darker

    gl_FragColor = vec4(color,1.0);
}
```

5.4.3 PS Blending modes


```
#ifdef GL_ES
precision mediump float;
#endif

#define BlendLinearDodgef               BlendAddf
#define BlendLinearBurnf                BlendSubstractf
#define BlendAddf(base, blend)          min(base + blend, 1.0)
#define BlendSubstractf(base, blend)    max(base + blend - 1.0, 0.0)
#define BlendLightenf(base, blend)      max(blend, base)
#define BlendDarkenf(base, blend)       min(blend, base)
#define BlendLinearLightf(base, blend)  (blend < 0.5 ? BlendLinearBurnf(base, (2.0 * blend))
#define BlendScreenf(base, blend)       (1.0 - ((1.0 - base) * (1.0 - blend)))
#define BlendOverlayf(base, blend)      (base < 0.5 ? (2.0 * base * blend) : (1.0 - 2.0 * (1
#define BlendSoftLightf(base, blend)    ((blend < 0.5) ? (2.0 * base * blend + base * base *
#define BlendColorDodgef(base, blend)   ((blend == 1.0) ? blend : min(base / (1.0 - blend),
#define BlendColorBurnf(base, blend)    ((blend == 0.0) ? blend : max((1.0 - ((1.0 - base) /
#define BlendVividLightf(base, blend)   ((blend < 0.5) ? BlendColorBurnf(base, (2.0 * blend)
#define BlendPinLightf(base, blend)     ((blend < 0.5) ? BlendDarkenf(base, (2.0 * blend)) :
#define BlendHardMixf(base, blend)      ((BlendVividLightf(base, blend) < 0.5) ? 0.0 : 1.0)
#define BlendReflectf(base, blend)      ((blend == 1.0) ? blend : min(base * base / (1.0 - b

// Component wise blending
#define Blend(base, blend, funcf)       vec3(funcf(base.r, blend.r), funcf(base.g, blend.g),
```

```
#define BlendNormal(base, blend)        (blend)
#define BlendLighten                    BlendLightenf
#define BlendDarken                     BlendDarkenf
#define BlendMultiply(base, blend)      (base * blend)
#define BlendAverage(base, blend)       ((base + blend) / 2.0)
#define BlendAdd(base, blend)           min(base + blend, vec3(1.0))
#define BlendSubstract(base, blend)     max(base + blend - vec3(1.0), vec3(0.0))
#define BlendDifference(base, blend)    abs(base - blend)
#define BlendNegation(base, blend)      (vec3(1.0) - abs(vec3(1.0) - base - blend))
#define BlendExclusion(base, blend)     (base + blend - 2.0 * base * blend)
#define BlendScreen(base, blend)        Blend(base, blend, BlendScreenf)
#define BlendOverlay(base, blend)       Blend(base, blend, BlendOverlayf)
#define BlendSoftLight(base, blend)     Blend(base, blend, BlendSoftLightf)
#define BlendHardLight(base, blend)     BlendOverlay(blend, base)
#define BlendColorDodge(base, blend)    Blend(base, blend, BlendColorDodgef)
#define BlendColorBurn(base, blend)     Blend(base, blend, BlendColorBurnf)
#define BlendLinearDodge                BlendAdd
#define BlendLinearBurn                 BlendSubstract

#define BlendLinearLight(base, blend)   Blend(base, blend, BlendLinearLightf)
#define BlendVividLight(base, blend)    Blend(base, blend, BlendVividLightf)
#define BlendPinLight(base, blend)      Blend(base, blend, BlendPinLightf)
#define BlendHardMix(base, blend)       Blend(base, blend, BlendHardMixf)
#define BlendReflect(base, blend)       Blend(base, blend, BlendReflectf)
#define BlendGlow(base, blend)          BlendReflect(blend, base)
#define BlendPhoenix(base, blend)       (min(base, blend) - max(base, blend) + vec3(1.0))
#define BlendOpacity(base, blend, F, O) (F(base, blend) * O + blend * (1.0 - O))

uniform sampler2D u_tex0;
uniform sampler2D u_tex1;

uniform float u_time;
uniform vec2 u_mouse;
uniform vec2 u_resolution;

void main (void) {
    vec2 st = gl_FragCoord.xy/u_resolution.xy;
    vec3 color = vec3(0.0);

    vec3 colorA = texture2D(u_tex0,st).rgb;
    vec3 colorB = texture2D(u_tex1,st).rgb;

    color = BlendMultiply(colorA,colorB);
    // color = BlendAdd(colorA,colorB);
    // color = BlendLighten(colorA,colorB);
    // color = BlendDarken(colorA,colorB);
```

```
    // color = BlendAverage(colorA,colorB);
    // color = BlendSubstract(colorA,colorB);
    // color = BlendDifference(colorA,colorB);
    // color = BlendNegation(colorA,colorB);
    // color = BlendExclusion(colorA,colorB);
    // color = BlendScreen(colorA,colorB);
    // color = BlendOverlay(colorA,colorB);
    // color = BlendSoftLight(colorA,colorB);
    // color = BlendHardLight(colorA,colorB);
    // color = BlendColorDodge(colorA,colorB);
    // color = BlendColorBurn(colorA,colorB);
    // color = BlendLinearLight(colorA,colorB);
    // color = BlendVividLight(colorA,colorB);
    // color = BlendPinLight(colorA,colorB);
    // color = BlendHardMix(colorA,colorB);
    // color = BlendReflect(colorA,colorB);
    // color = BlendGlow(colorA,colorB);
    // color = BlendPhoenix(colorA,colorB);

    gl_FragColor = vec4(color,1.0);
}
```

5.5 Kernel convolutions

5.6 Filters

6 Appendix

1. How can I navigate this book off-line?
2. How to run the examples on a Raspberry Pi?
3. How to print this book?
4. How can I collaborate with this book?
5. An introduction for those coming from JS by Nicolas Barradeau
6. An introduction for vectors by …
7. An introduction to interpolation by …

7 Examples Gallery

Created by kynd(@kyndinfo) and Patricio Gonzalez Vivo(@patriciogv)

This is a collection of examples extracted from the chapters of this book together with shared shaders kindly donated by other readers using the on-line editor. Feel free to explore and tweak them bit by bit. Once you have something you are proud of, click the "Export" and then copy the "URL to code…". Send it to [@bookofshaders](https://twitter.com/bookofshaders) or [@kyndinfo](https://twitter.com/kyndinfo). We are looking forward to see it!

8 Glossary

8.1 By theme

- TYPES

void bool int float bvec2 bvec3 bvec4 ivec2 ivec3 ivec4 vec2 vec3 vec4 mat2 mat3 mat4 sampler2D samplerCube struct

- QUALIFIERS

attribute const uniform varying precision highp mediump lowp in out inout

- BUILT-IN VARIABLES

gl_Position gl_PointSize gl_PointCoord gl_FrontFacing gl_FragCoord gl_FragColor

- BUILT-IN CONSTANTS

gl_MaxVertexAttribs gl_MaxVaryingVectors gl_MaxVertexTextureImageUnits gl_MaxCombinedTextureImageUnits gl_MaxTextureImageUnits gl_MaxFragmentUniformVectors gl_MaxDrawBuffers

- ANGLE & TRIGONOMETRY FUNCTIONS

radians() degrees() sin() cos() tan() asin() acos() atan()

- EXPONENTIAL FUNCTIONS

pow() exp() log() exp2() log2() sqrt() inversesqrt()

- COMMON FUNCTIONS

abs() sign() floor() ceil() fract() mod() min() max() clamp() mix() step() smoothstep()

- GEOMETRIC FUNCTIONS

length() distance() dot() cross() normalize() facefoward() reflect() refract()

- MATRIX FUNCTIONS

matrixCompMult()

- VECTOR RELATIONAL FUNCTIONS

lessThan() lessThanEqual() greaterThan() greaterThanEqual() equal() notEqual() any() all() not()

- TEXTURE LOOKUP FUNCTIONS

texture2D() textureCube()

8.2 Alphabetical

- A

abs() acos() all() any() asin() atan() attribute

- B

bool bvec2 bvec3 bvec4

- C

ceil() clamp() const cos() cross()

- D

degrees() dFdx() dFdy() distance() dot()

- E

equal() exp() exp2()

- F

faceforward() float floor() fract()

- G

greaterThan() greaterThanEqual() gl_FragColor gl_FragCoord gl_FrontFacing gl_PointCoord gl_PointSize gl_Position gl_MaxCombinedTextureImageUnits gl_MaxDrawBuffers gl_MaxFragmentUniformVectors gl_MaxVaryingVectors gl_MaxVertexAttribs gl_MaxVertexTextureImageUnits gl_MaxTextureImageUnits

- H

highp

- I

in inout int inversesqrt() ivec2 ivec3 ivec4

- L

length() lessThan() lessThanEqual() log() log2() lowp

- M

matrixCompMult() mat2 mat3 mat4 max() mediump min() mix() mod()

- N

normalize() not() notEqual()

- O

out