What is OPENRNDR?

OPENRNDR is an application framework and a library for creative coding written in Kotlin.

OPENRNDR offers APIs for easy and flexible programming of accelerated graphics.

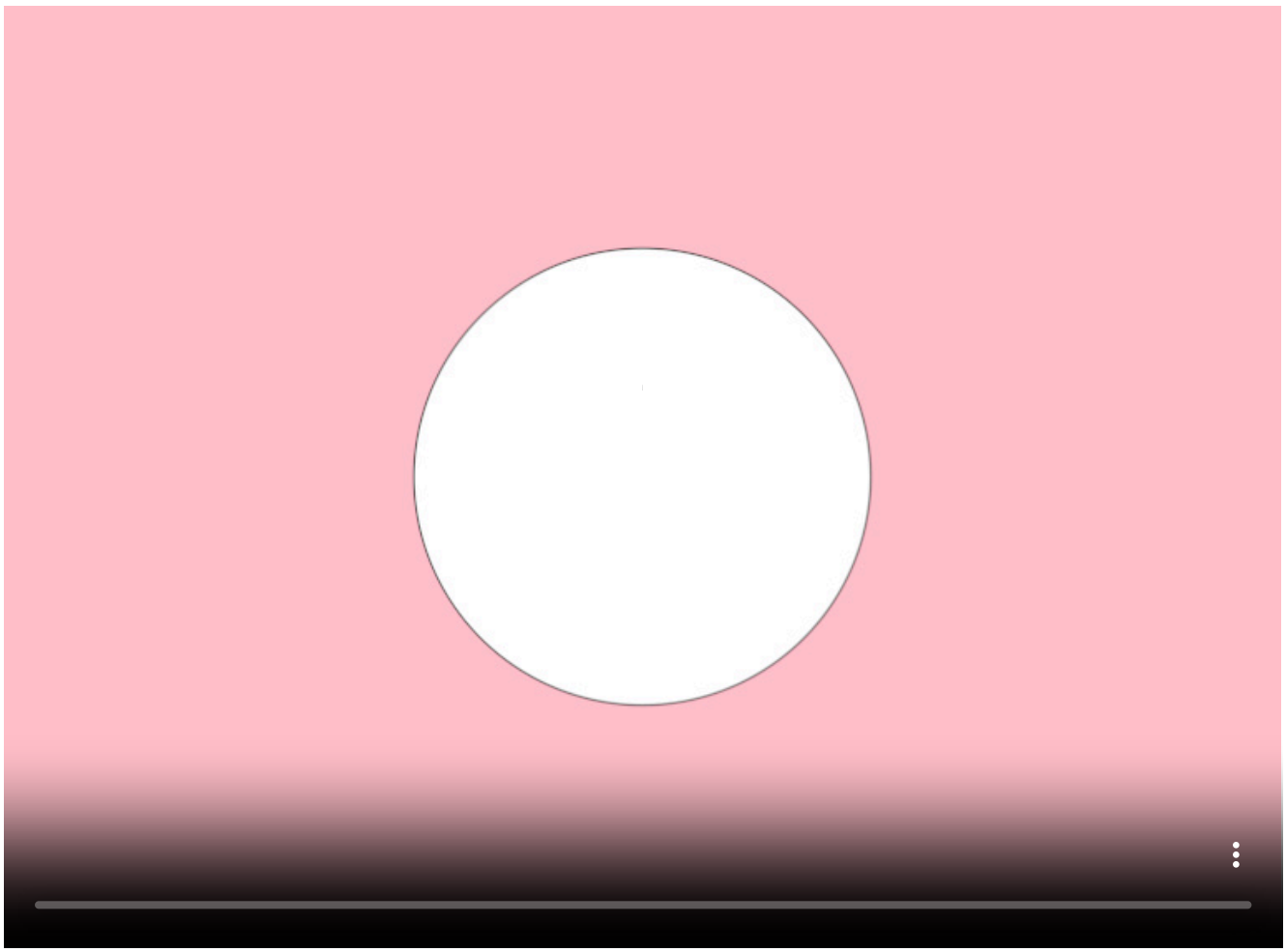
OPENRNDR is intended for prototyping as well as building production quality software.

OPENRNDR is free and open source software. The source code can be found on Github.

OPENRNDR is an initiative of the RNDR studio for interactive and interaction design based in The Netherlands.

A simple OPENRNDR program

Here we show a very simple program written using OPENRNDR.



```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fill = ColorRGBa.WHITE
            drawer.circle(drawer.bounds.center, abs(cos(seconds)) * height * 0.51)
        }
```

```
    }
}
```

Link to the full example

edit on GitHub

TABLE OF CONTENTS

OPENRNDR currently supports desktop platforms including macOS, Windows and Linux. Additionally there's experimental support for running programs in web browsers via KotlinJS.

Operating systems known to work

macOS	10.10 up to 13, both Intel and ARM
Window	10 and 11
Linux	Ubuntu LTS version 18.04, 20.04, 22.04, ArchLinux, Manjaro

Other versions and distributions may work, but it is hard for us to verify.

OpenGL version

OPENRNDR requires a GPU that supports at least **OpenGL 3.3**. This includes relatively old ones like the nVidia 320M (2007) or the Intel HD4000 (2012) but excludes others like the Intel HD3000 (2011).

Programming languages

OPENRNDR is written in Kotlin and intended to run on the JVM. We believe Kotlin offers a well-balanced programming language that is both expressive and easy to read.

The library can likely be used from Java 8+ as that's one of the promises of Kotlin's Java-interop, however the APIs that OPENRNDR provides are making extensive use of Kotlin-specific features that may not translate well to Java.

Development environments

OPENRNDR is environment agnostic, however all our tutorial and reference material assumes Gradle and IntelliJ IDEA (Community Edition) are used.

Long-term support

We have not reached the point at which we can make promises regarding API stability. OPENRNDR is pre-1.0 software, which implies we try not to break things, but at times we have to. At times it is better to make incompatible changes than to continue with inconsistencies in or incompleteness of the API.

Getting Started with OPENRNDR

Download the code editor

To edit and run OPENRNDR programs, we will install an IDE (Integrated Development Environment) called IntelliJ IDEA.

- Download and install IntelliJ IDEA *Community Edition* (note: the paid version is called *Ultimate*).
- Start IntelliJ. On the first run it will offer the option to pick your default settings and continue. Use the default settings.

Download the template program

Instead of creating a project from scratch, the simplest way to start an OPENRNDR-based project is to use the `openrndr-template`:

- If you are running IntelliJ IDEA for the first time, the program menus are hidden. In that case click "Clone Repository" at the top-right area.
  - If program menus are visible, select "File > New > Project from version control...".
- In the dialog that appears:
  - Version Control: choose "Git".
  - URL: enter `https://github.com/openrndr/openrndr-template`
  - Directory: ensure it looks OK.
- When asked where the project should be opened, click on "new window".

If downloading the template fails:

- Make sure Git is installed.
- Did you enter the repository URL correctly?

Run the template program

After downloading the template, IntelliJ IDEA will download the necessary dependencies. See the progress in the status bar at the bottom of the window.

When IntelliJ opens a project for the first time, its `README.md` file is presented automatically. This file contains hints to help you navigate the template project.

When the status bar in IntelliJ shows no more activity, go to the Project view (left panel) and click on `src/main/kotlin/TemplateProgram.kt` to open that file

Note: the Project view can be shown and hidden by clicking the Project button in the top-left area of the window.

Once the source code of TemplateProgram.kt is visible, a small green triangle will appear before the words `fun main()`. Click that triangle to run the program.

Ta-da!

Next steps

Simplifying the Project View

By default, the Project View can look intimidating. The list contains many files and folders, and you haven't even started writing a program! But we can simplify it by hiding files that are not Kotlin programs.

First, let's enable some new scopes for the Project View:

- Make the Gradle panel visible by clicking on the Gradle elephant on the right side of the screen.
- In the panel that shows up, double-click on `openrndr-template > Tasks > openrndr > "add IDE file scopes"`.
- Hide the Gradle panel by clicking on the elephant again.

After running the `add IDE file scopes` task, let's enable the scope that shows code files:

- Click on `Project` ⌄ at the top-left of the Project View.
- Choose ★ `Code` at the bottom of the list.

Only two Kotlin files should be listed, everything else is now hidden. Isn't that a relief?

You can always switch back to the `Project` scope to see all files or switch to one of the others. For instance, the ★ `Media` scope will display only sound, image and video files from your project.

Multiple programs in one project

By default, the `openrndr-template` project comes with two programs. One is called `TemplateProgram.kt` and the other `TemplateLiveProgram.kt`.

The second one provides live-coding! If you run that program by clicking on the green triangle next to `fun main()`, any changes you make to the code will be visible in the running program after you save your changes. No need to stop and start the program.

But you are not limited to two programs, of course. In the Project View on the left, you can copy and paste `.kt` files (using keyboard shortcuts or the mouse). IntelliJ will ask you to name the new file (the name must start with a letter). It will also ask whether you want to add the new file to Git. It's fine to click Cancel.

If you created a third `.kt` file, make some changes to it and then click the green triangle next to `fun main()` to run it.

All your Kotlin programs must be placed under `src/main/kotlin`, otherwise they won't run.

Note: at the top of the window, there is also a green triangle to run programs. By default, clicking that green triangle will re-run the last program you ran, no matter which file you are editing now. This can be surprising when you start editing a different program in your project. For a more intuitive behavior, click on the drop-down on the left of the triangle and select `Current File`. After this change, the top green triangle will run whatever program you are currently editing.

`import` statements

OPENRNDR is very modular. Not every possible feature is available by default: to make classes available, they need to be "imported". For example, if we want to create a `Circle` object, we need to add a line at the top of the program that imports the `Circle` class. After doing this, the compiler will understand what a `Circle` is. Fortunately, we do not need to type such import lines by hand. Let's see how IntelliJ helps with this.

Inside the `extend { ... }` block, go ahead and type `val v = Vector3`. When you do this, a floating dialog should appear. What this dialog is doing is asking you "which Vector3 do you mean?". See, different classes may provide something that sounds like a Vector3. Some of these classes can be part of OPENRNDR, and other classes may be provided by Java or other frameworks.

The one we are looking for is "Vector3 (org.openrndr.math)". If it is highlighted in that floating dialog, press the `ENTER` key. This will add the necessary `import` statement to the beginning of the file, making `Vector3` available for you. If the highlighted item was not the one we were looking for, press the `UP` and `DOWN` arrow keys to choose the right one, then press `ENTER`.

If you accidentally import the wrong one, you can `undo`, or you can also manually delete the import from the top of the file.

When an import is missing, the unrecognized word will be highlighted in red. Let's try this. Delete any `Vector3` related imports from the top of the file. The `Vector3` word will be highlighted as an error. When this happens, we can automatically import the right class by clicking the word (highlighted in red), then press `ALT+ENTER` (or `Command+ENTER` on Mac). A tool tip will show up letting you choose the right import to add.

IntelliJ cleanup tips

In IntelliJ, you can double-tap the `SHIFT` key to open a global search tool that will find whatever you type in your source code, but also in the program settings, menus and actions you can perform.

Let's try it:

- Double-tap `SHIFT` and type `optimize`. In the search results choose `optimize imports`. This will remove no longer used import statements from the beginning of your program.
- Double-tap `SHIFT` and type `reformat`. In the search results choose `reformat code`. This will tidy up the spacing and indentation of your program.

What's next?

At this point you are likely interested in how this program is structured. The guide explains more in the Program basics chapter.

If you are more interested in reading source code, you can find the code for the examples in this guide in the `openrndr-examples` repository.

If you are interested in more advanced examples, we recommend checking out the demo programs in the `orx` repository; most `orx` modules have demos in their `src/demo/kotlin` folder.

Tip for macOS users

When running a program, the console will display `Warning: Running on macOS without -XstartOnFirstThread JVM argument.` To clear this warning (and enable debugging):

- Open the `Run > Edit Configurations...` menu.
- Add `-XstartOnFirstThread` in the `VM Options` text field.
- Click `Ok` to close the dialog.

Tip for Windows multi-GPU users

If your computer has multiple GPUs, you can choose which one OPENRNDR uses like this:

- Run the openrndr-template program by clicking on the green triangle. In IntelliJ, look at the console in the bottom area and note down the full path of the `java.exe` program being used. It probably starts with something like `C:\Users\...`
- Open the Windows Graphics Settings.
- Click Browse and find the exact same `java.exe` you noted down earlier.
- Click Options and choose your preferred GPU, then click Save.

Kotlin language and tools

OPENRNDR is written using the Kotlin programming language, which is similar to Java but more modern and less verbose.

The Gradle build tool is used by the OPENRNDR projects to download dependencies, build and distribute projects.

We recommend to have some fun with the drawing chapter first, then come back to this chapter when you start wondering about the language, the syntax and the framework.

TABLE OF CONTENTS

The Kotlin programming language

Kotlin is a modern, readable and fun language, perfect for creative coding.

A good place to start discovering the language is the Kotlin Tour

Most of the examples on that website can be edited and run to immediately see the result.

From the Official documentation we recommend exploring the *Basics*, *Concepts* and *Standard Library* sections. Data structures like `List`, `Map` and `Set` are explained under the *Standard Library* section, and they are one of the aspects that make working with Kotlin enjoyable. Check them out!

Kotlin in OPENRNDR

When designing a framework like OPENRNDR, many decisions need to be made. What should be favored? Brief syntax? Flexibility? Expressiveness? Execution speed? Similarity with other frameworks?

Those decisions shape what user code will look like. In this section we attempt to explain some of those decisions and possible differences with other languages and frameworks.

Most concepts in the Kotlin programming language will sound familiar to anyone experienced in other languages, but some may be new.

Let's take a look at them.

Functions and Lambdas

If you look at a basic OPENRNDR program and can't stop thinking about that syntax with curly brackets following certain words, this section is for you.

Functions

Please read the section on functions in the Kotlin Tour all the way to the end. There you will learn about Lambda functions.

When you're done, you should probably be able to understand the syntax of a basic OPENRNDR program: when we encounter code like `extend { ... }` we know that we are dealing with two functions. The first one is called `extend()` and the second one, `{ ... }`, is an anonymous lambda function. In Kotlin, it's not necessary to include the parentheses, as in `extend({ ... })`, if the last argument of the `extend` function is a Lambda function.

What `extend { ... }` does is calling the `extend` function to tell OPENRNDR: "here, take this anonymous function and execute it on every animation frame".

Other creative coding frameworks usually expect *one* specific method, often called `draw`, where the user can draw to the screen.

Having an `extend` method that accepts a lambda function gives OPENRNDR the unique capability of combining multiple `extend` blocks in the same program.

The framework provides several such `extend` blocks (called Extensions), for instance, Screenshots, ScreenRecorder and Camera2D, among others. This makes it possible to build programs by composing code blocks as if they were LEGO pieces and add powerful functionality with just one or two lines of code.

Mutability and immutability

According to the coding conventions section of the Kotlin guide, we should prefer using immutable to mutable data structures and declare local variables and properties as `val` rather than `var` if they are not modified after initialization.

To learn more about this topic, please take a look at the hello world, basic types and collections sections of the Kotlin Tour.

This should help you familiarize with the idea of mutable and immutable data in Kotlin.

OPENRNDR data structures

Vectors and Matrices

OPENRNDR makes extensive use of data types like `Vector2`, `Vector3`, `Vector4`, `Matrix2x2`, `Matrix3x3`, `Matrix4x4` and `Matrix5x5` for creating and drawing 2D and 3D shapes

Such data structures are immutable in OPENRNDR. That means that the following would not work:

```
val v = Vector2.ZERO
v.x += 0.1 // error: x is immutable
```

We could switch from `val` to `var`:

```
var v = Vector2.ZERO
```

but we still can't modify the `x` component of that vector, because `x` is still immutable. The solution is to not try changing the components independently, but to assign a new value to the variable:

```
var v = Vector2.ZERO
v += Vector2(0.1, 0.0)
```

The same idea applies to matrices. For example:

```
var m = Matrix4x4.IDENTITY // One of the existing Matrix4x4 "presets"
m = m * 2.0 + 1.0 // Assign a new matrix with all components multiplied by 2.0 and incremented by 1.0
```

In a nutshell, don't try changing individual properties of OPENRNDR objects but assign a new value to the variable instead.

The drawer state is mutable

One area in the framework that does have a mutable state is the drawer.

We can alter properties like `drawer.fill`, `drawer.stroke`, `drawer.strokeWeight` and doing so will affect any following shapes we draw.

One detail: `drawer.fill` holds a `ColorRGBa` variable. This variable has `r`, `g`, `b` and `a` components. But the same way we could not modify the `x` component of a `Vector2`, we cannot modify the `r` component of a `ColorRGBa`. We just need to

assign a new value to the `fill`, instead of trying to modify the components of the color.

Mutating other OPENRNDR objects

A `ShapeContour` is a core OPENRNDR class that holds an open or closed contour made out of straight or curved segments.

Let's imagine we have a circular contour made out of four curved segments. Could we mutate it? Could we animate the position of one of its vertices? Again, the same rule applies here. Contours, segments and vertices are all immutable.

If we want to animate and deform the circle, we need to recreate the contour on every animation frame. We could do this based on the `seconds` variable, based on the previous version of the contour, or based on some other changing data.

OPENRNDR provides many powerful tools to construct and deform contours. We only need to get used to the idea of working with immutable data.

What is Gradle?

Gradle is a build tool used by OPENRNDR components like the core, extensions, template, and guide.

What is a build tool?

A core task a build tool performs is telling the compiler to compile programs. Compiling involves reading your source code files (human-readable text files describing algorithms) and converting each line into instructions the machine can execute. A compiler program usually takes several source code files and outputs one executable program. Compilers often have complex command-line arguments, and build tools take care of such details so we don't need to.

Other essential steps built tools perform include downloading dependencies (third-party libraries used by our source code) and packing those dependencies into our executable program.

Gradle is a complex tool that supports multiple programming languages. It has many plugins providing all kinds of functionality needed by people building and distributing applications.

To configure Gradle, one uses Tasks which are text files describing the steps required to complete something.

Do we always need to use a build tool?

Theoretically, one could write Kotlin or Java programs without using any build tool, but that would involve manually downloading all the dependencies and writing long instructions in the command line to build your programs. Using Gradle makes our life easier.

Do I need to know Gradle to use OPENRNDR?

In most cases, one can happily write OPENRNDR programs without knowing that Gradle exists, although we may need to edit Gradle configuration files in some cases.

Enable or disable extensions

Suppose a program requires access to a MIDI hardware controller. In that case, we need to edit the `build.gradle.kts` file to uncomment a line to enable MIDI; then, we need to click "Sync All Gradle Projects" in the IDE to trigger the downloading of the MIDI libraries.

Add or remove dependencies

Thousands of JAVA libraries are available to our programs. We only need to add one line to the `build.gradle.kts` file to add a dependency.

Three such dependencies (JSON, CSV, and XML) are predefined and we only need to uncomment a line if we need them. See fileIO in the guide for details.

Other dependencies can be easily added in this format: `implementation("org.jbox2d:jbox2d-library:2.2.1.1")`. You can find such dependencies in www.mvnrepository.com. Once found, choose the `Gradle (Kotlin)` tab in that website, copy the `implementation(...)` code and paste it into `build.gradle.kts` inside the `dependencies { ... }` block. Remember to reload Gradle!

What else does Gradle do for OPENRNDR?

- It converts the OPENRNDR source code into multiple distributable libraries.
- It helps publish those libraries to an online database, making them accessible to everyone.

- Similarly, it builds all the ORX extensions (called add-ons or plugins in other frameworks) and helps publish them to the same database.
- It runs your OPENRNDR programs and builds executables you can share with others.
- It builds the OPENRNDR Guide, creates images and video files, embeds them in the documentation, and then publishes the guide online.
- Gradle runs tests to ensure code changes did not break any functionality.
- Gradle updates readme.md files, for example, to list all the ORX extensions in the root readme file.

Thanks to Gradle and the automation it enables, the whole OPENRNDR project can be maintained by a small team of developers.

Are there alternatives to Gradle?

Other build tools often used are Maven, Ant, and cmake.

Gradle is an excellent fit for Kotlin-based projects because Kotlin itself can be used to write Gradle tasks, avoiding the need to work with additional languages.

Program basics

edit on GitHub

TABLE OF CONTENTS

Program basics

Let's have a look at how an OPENRNDR program is structured. Most programs will share a structure like the one below.

```
fun main() = application {
    configure {
    // set Configuration options here
    }

    program {
        // -- what is here is executed once
        extend {
        // -- what is here is executed 'as often as possible'
        }
    }
}
```

application

The `application` block is used to setup the run-time environment of the software we are writing. This block houses two other blocks: `configure` and `program`. Think of it as an `OPENRNDR` application.

Configure

The `configure` block is an optional block that is used to configure the run-time environment. Most commonly it is used to configure the size of the window.

An example configuration that sets the window size, window resizability and title is as follows:

```
fun main() = application {
    configure {
        width = 1280
        height = 720
        windowResizable = true
        title = "OPENRNDR Example"
    }
    program {}
}
```

An example for a full screen window on your second monitor with the mouse pointer hidden:

```
fun main() = application {
    configure {
        fullscreen = Fullscreen.CURRENT_DISPLAY_MODE
        display = displays[1]
        hideCursor = true
    }
    program {}
}
```

Starting your program with a custom configuration looks roughly like this.

```
fun main() = application {
    configure {
    // settings go here
    }
    program {
        // -- one time set-up code goes here
        extend {
        // -- drawing code goes here
        }
    }
}
```

The table below lists a selection of configuration options. See the API for the complete list.

Property	Type	Default value	Description
`width`	`Int`	`640`	initial window width
`height`	`Int`	`480`	initial window height

Property	Type	Default value	Description
`windowResizable`	`Boolean`	`false`	allow resizing of window?
`fullscreen`	`Fullscreen`	`Fullscreen.DISABLED`	When specified, either `Fullscreen.CURRENT_DISPLAY_MODE` to make the window match the current display resolution, or `Fullscreen.SET_DISPLAY_MODE` to change the display resolution to match `width` and `height`.
`position`	`IntVector2?`	`null` (center of the primary display)	initial window position (top-left corner)
`display`	`Display?`	`null` (primary display)	The display on which to create the window. All detected displays are present in the `displays` list within the `application {}` block.
`windowAlwaysOnTop`	`Boolean`	`false`	keep the window floating above other windows?
`unfocusBehaviour`	`UnfocusBehaviour`	`UnfocusBehaviour.NORMAL`	The value `UnfocusBehaviour.THROTTLE` can be specified to throttle the program to 10Hz when unfocused.
`hideCursor`	`Boolean`	`false`	hide the cursor?
`title`	`String`	`"OPENRNDR"`	window title
`hideWindowDecorations`	`Boolean`	`false`	hide window decorations?

Changing the configuration while the program runs

To modify the configuration after the program has started we can set various properties via `application`.

```
fun main() = application {
    program {
        extend {
            if (frameCount % 60 == 0) {
                application.cursorVisible = Random.bool()
                application.windowPosition = Vector2.uniform(0.0, 200.0)
            }
        }
    }
}
```

Program

The program block houses the actual programming logic. Note that `program {}` has a `Program` receiver.

The code inside the `program` block is only executed after a window has been created and a graphical context has been set up. This code is only executed once.

In the `program` block one can install extensions using `extend`. Extensions are by default executed as often as possible. The most important type of extension is the one holding the user code.

A minimal application-program-extend setup would then look like this:

```
fun main() = application {
    program {
        // -- what is here is executed once
        // -- It's a good place to load assets
        extend {
            // -- what is here is executed 'as often as possible'
            drawer.circle(width / 2.0, height / 2.0, 100.0)
        }
    }
}
```

Extend

The `program` block usually contains an `extend` block which gets executed as often as possible.

```
fun main() = application {
    program {
        extend {
            drawer.circle(width / 2.0, height / 2.0, 50.0)
        }
    }
}
```

The `extend` block serves as a “draw loop”, which is what we need for drawing smooth animations. To demonstrate that the result is not a still image, let’s draw a circle located wherever the mouse cursor is:

```
fun main() = application {
    program {
        extend {
            drawer.circle(mouse.position, 50.0)
        }
    }
}
```

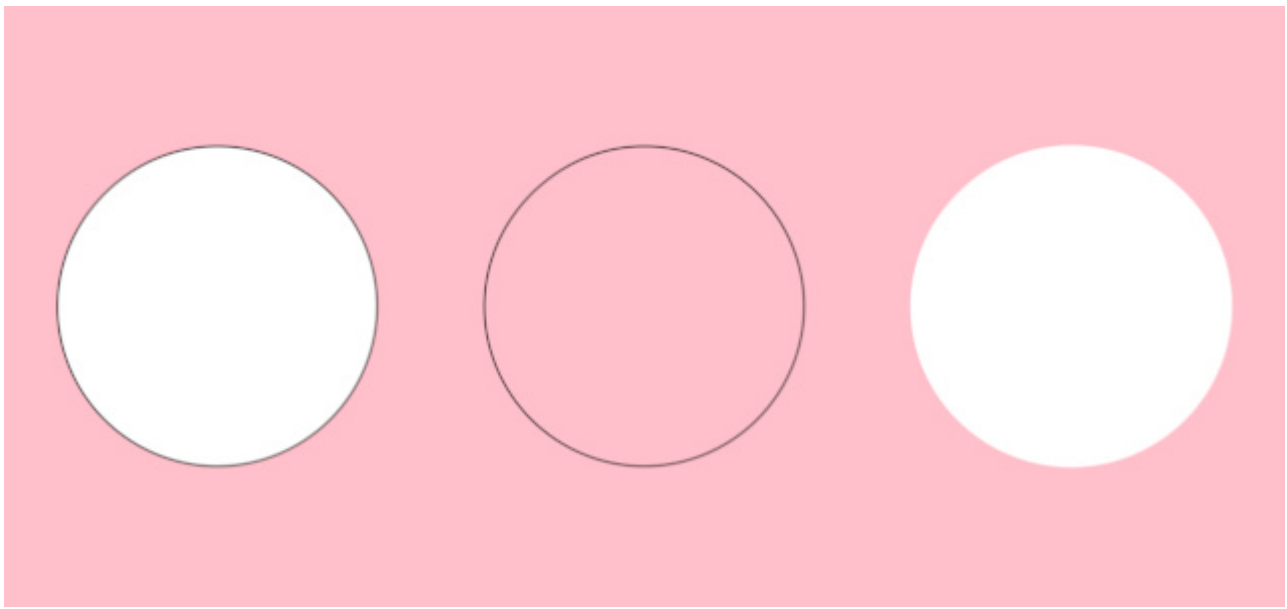
Drawing

edit on GitHub

TABLE OF CONTENTS

Drawing circles

A circle is drawn around coordinates `x`, `y`, i.e. `x` and `y` specify the center of the circle. Circles are filled with the color set in `Drawer.fill` and their stroke is set to `Drawer.stroke`. The width of the stroke follows `Drawer.strokeWeight`.



```
fun main() = application {
    configure {
        height = 300
    }
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)

            // -- draw a circle with white fill and black stroke
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 1.0
            drawer.circle(width / 6.0, height / 2.0, width / 8.0)

            // -- draw a circle without a fill, but with black stroke
            drawer.fill = null
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 1.0
            drawer.circle(width / 6.0 + width / 3.0, height / 2.0, width / 8.0)

            // -- draw a circle with white fill, but without a stroke
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = null
            drawer.strokeWeight = 1.0
            drawer.circle(width / 6.0 + 2 * width / 3.0, height / 2.0, width / 8.0)
        }
```

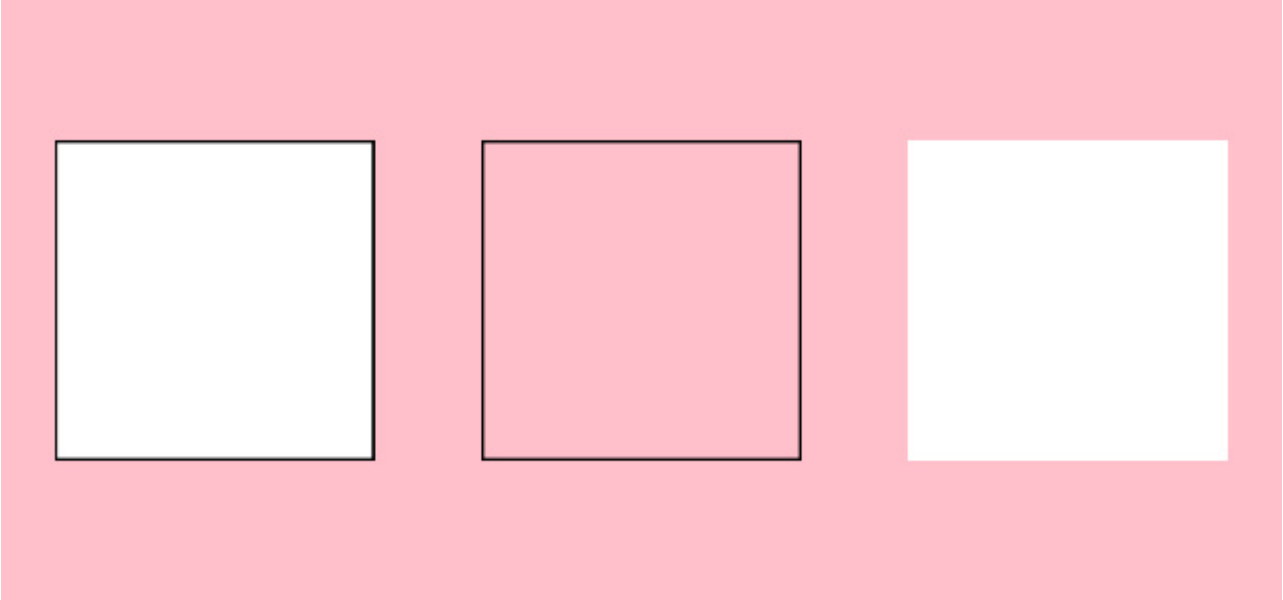
```
    }
}
```

Link to the full example

You may have spotted the two other APIs for drawing circles; `Drawer.circle(center: Vector2, radius: Double)` and `Drawer.circle(circle: Circle)` and wonder what those are for. They are for drawing the exact same circle, but using arguments that may be more convenient in scenarios in which values are provided by `Vector2` or `Circle` types.

```
drawer.circle(mouse.position, 50.0)
```

Drawing rectangles



```
fun main() = application {
    configure {
        height = 300
    }
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)

            // -- draw rectangle with white fill and black stroke
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 1.0
            drawer.rectangle(width / 6.0 - width / 8.0, height / 2.0 - width / 8.0, width / 4.0, width / 4.0)

            // -- draw rectangle without fill, but with black stroke
            drawer.fill = null
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 1.0
            drawer.rectangle(width / 6.0 - width / 8.0 + width / 3.0, height / 2.0 - width / 8.0, width / 4.0, width /
4.0)

            // -- draw a rectangle with white fill, but without stroke
            drawer.fill = ColorRGBa.WHITE
```

```
            drawer.stroke = null
            drawer.strokeWeight = 1.0
            drawer.rectangle(width / 6.0 - width / 8.0 + 2.0 * width / 3.0, height / 2.0 - width / 8.0, width / 4.0,
width / 4.0)
        }
    }
}
```

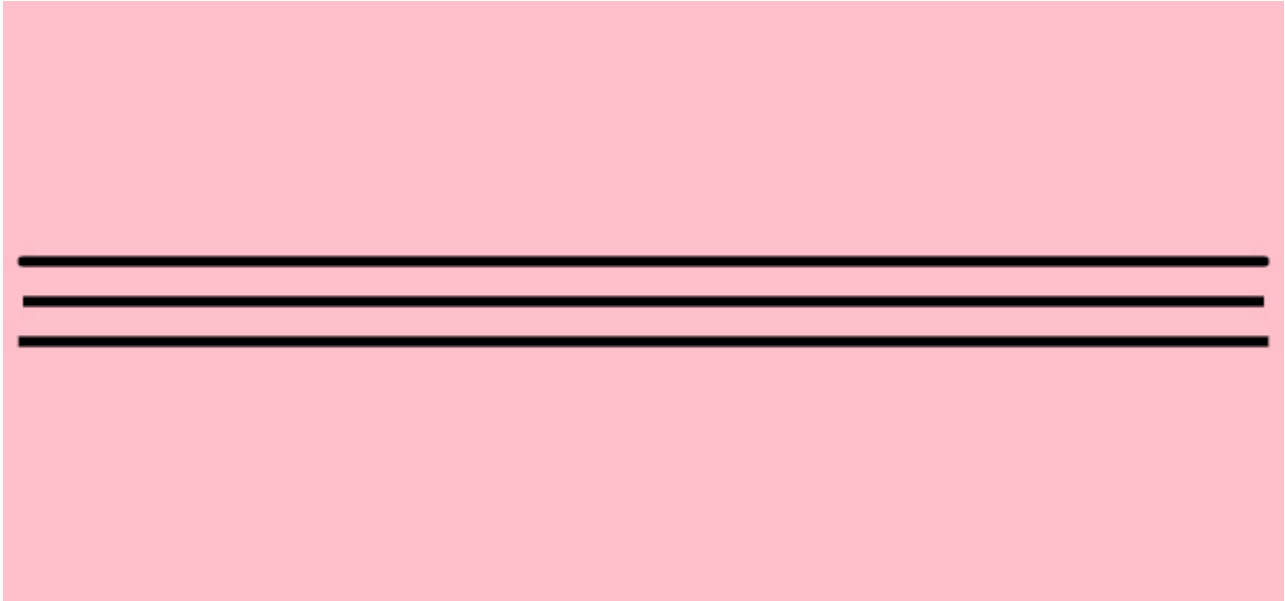
Link to the full example

Drawing lines

Single lines are drawn per segment between two pairs of coordinates using `lineSegment`. Line primitives use `Drawer.stroke` to determine the color drawing color and `Drawer.strokeWeight` to determine the width of the line.

Line endings can be drawn in three styles by setting `Drawer.lineCap`

LineCap.	description
BUTT	butt cap
ROUND	round cap
SQUARE	square cap



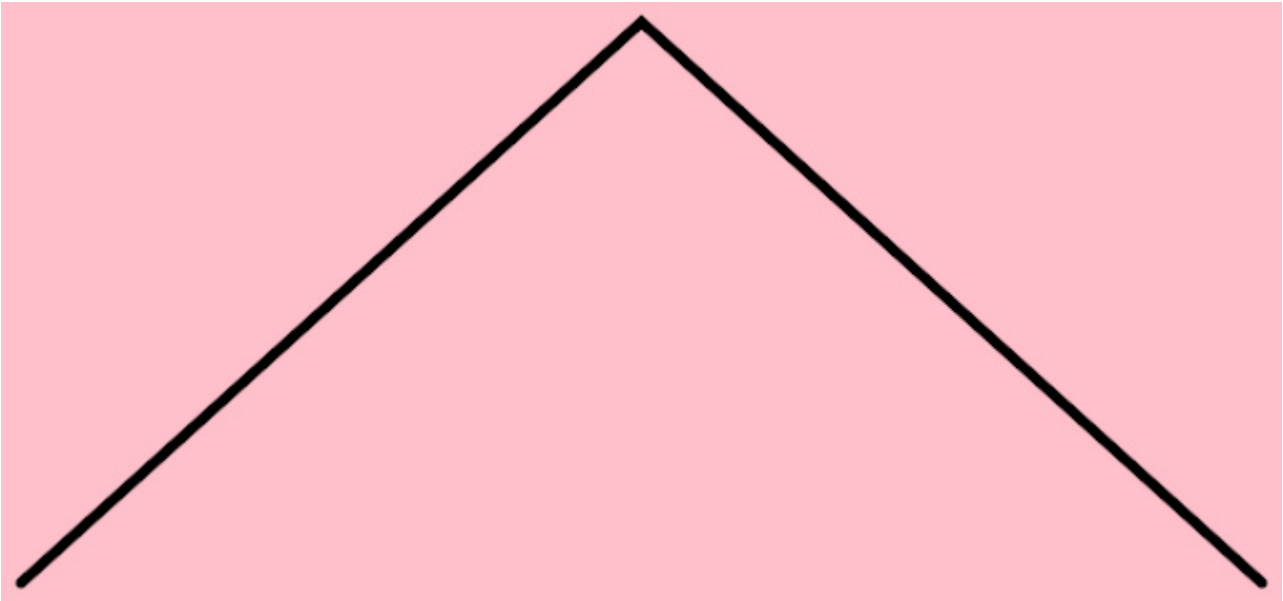
```
fun main() = application {
    configure {
        height = 300
    }
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)
            // -- setup line appearance
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 5.0
            drawer.lineCap = LineCap.ROUND

            drawer.lineSegment(10.0, height / 2.0 - 20.0, width - 10.0, height / 2.0 - 20.0)
```

```
            drawer.lineCap = LineCap.BUTT
            drawer.lineSegment(10.0, height / 2.0, width - 10.0, height / 2.0)

            drawer.lineCap = LineCap.SQUARE
            drawer.lineSegment(10.0, height / 2.0 + 20.0, width - 10.0, height / 2.0 + 20.0)
        }
    }
}
```

Link to the full example

Drawing line strips

A run of connected line segments is called a line strip and is drawn using `lineStrip`. To draw a line strip one supplies a list of points between which line segments should be drawn.



```
fun main() = application {
    configure {
        height = 300
    }
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)
            // -- setup line appearance
            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 5.0
            drawer.lineCap = LineCap.ROUND

            val points = listOf(Vector2(10.0, height - 10.0), Vector2(width / 2.0, 10.0), Vector2(width - 10.0, height
- 10.0))
            drawer.lineStrip(points)
        }
    }
}
```

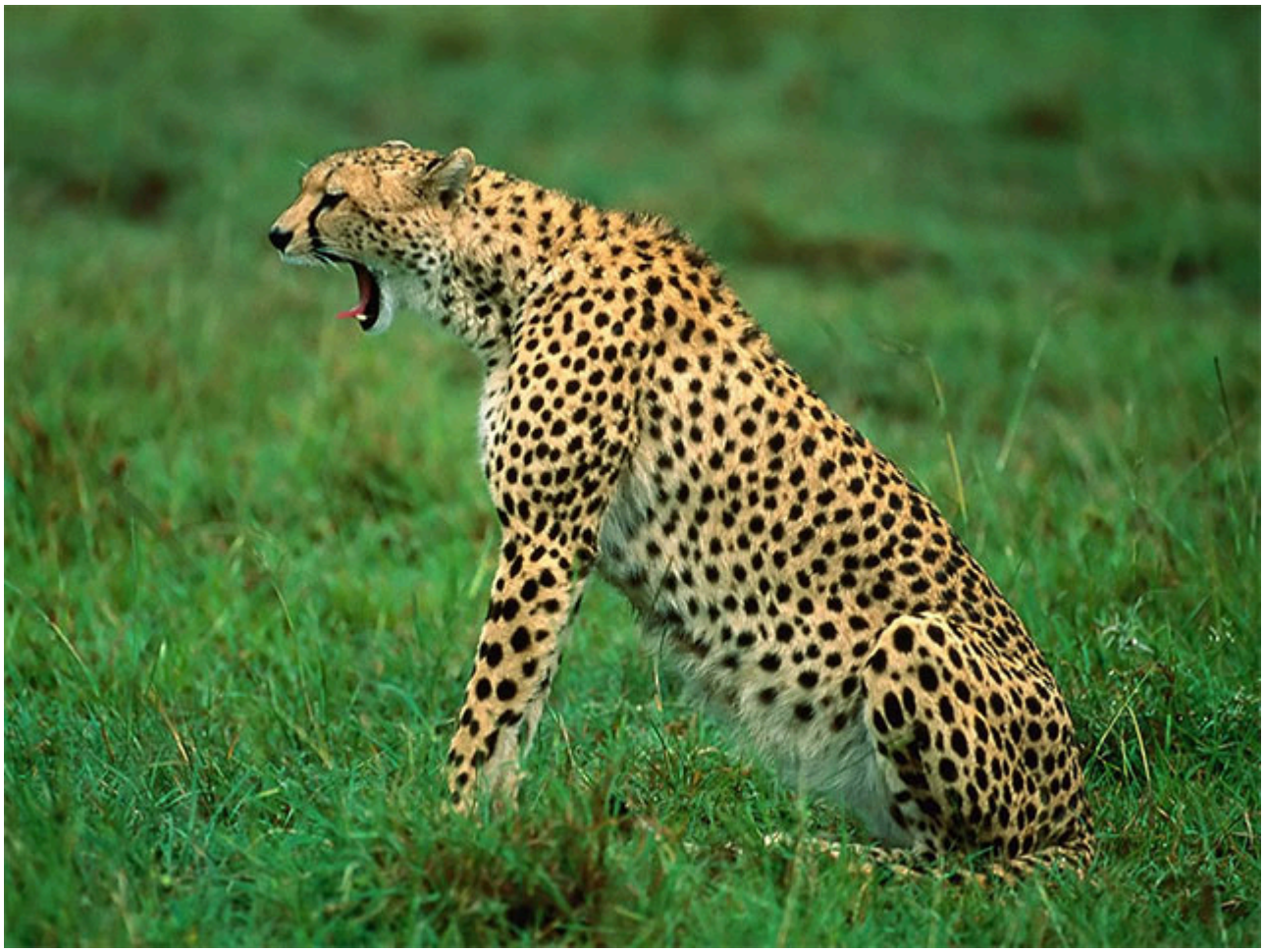
Link to the full example

Images

Images are stored in `ColorBuffer` instances, the image data resides in GPU memory

Loading and drawing images

Images are loaded using the `loadImage` function and drawn using `Drawer.image`.



```
fun main() = application {
    configure {}
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            drawer.image(image)
        }
    }
}
```

Link to the full example

To change the location of the image one can use `Drawer.image` with extra coordinates provided.

```
drawer.image(image, 40.0, 40.0)
```

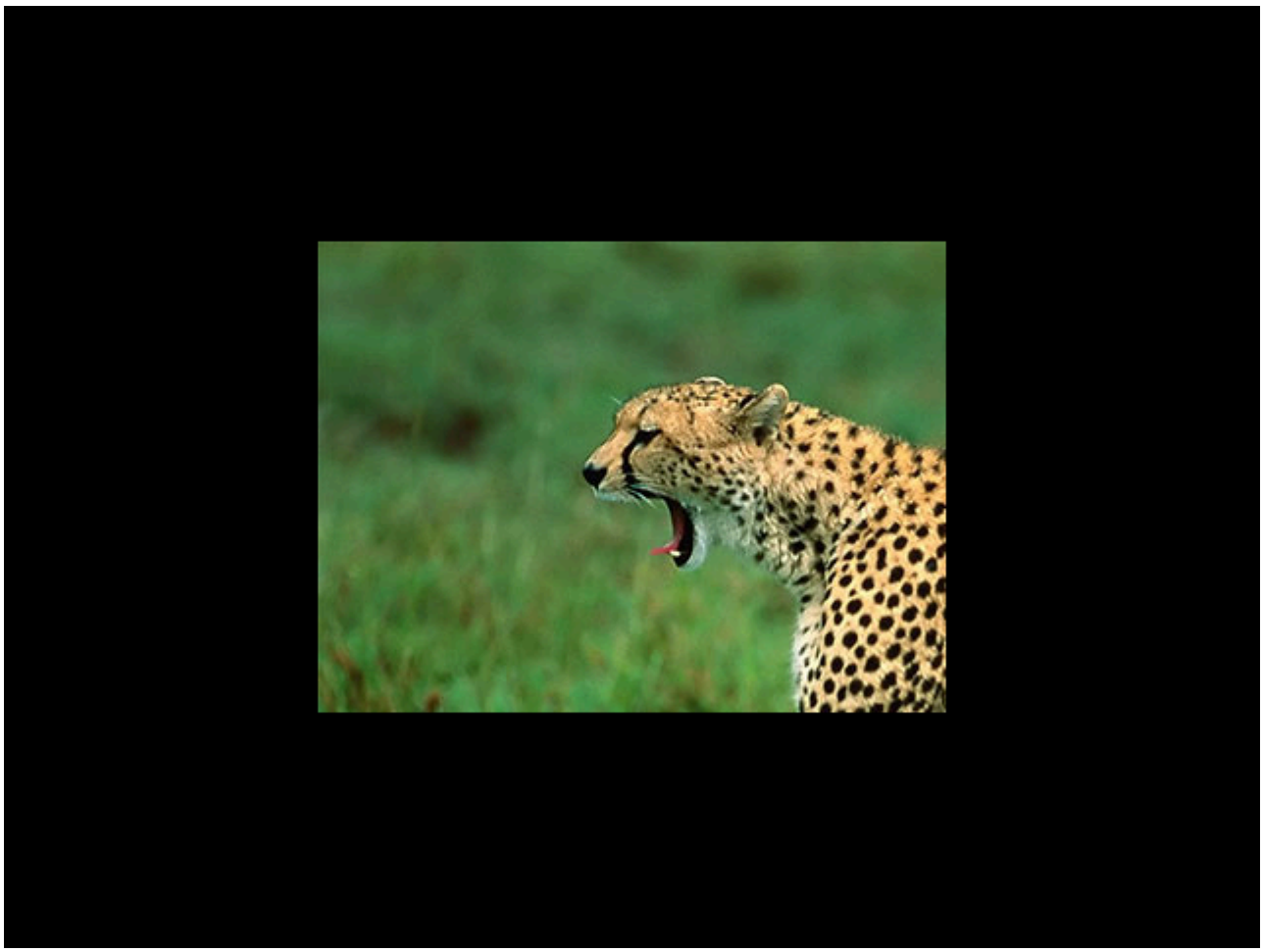
Extra `width` and `height` arguments can be provided to draw a scaled version of the image.

```
drawer.image(image, 40.0, 40.0, 64.0, 48.0)
```

To rotate an image (or any other element you draw) apply transformations.

Drawing parts of images

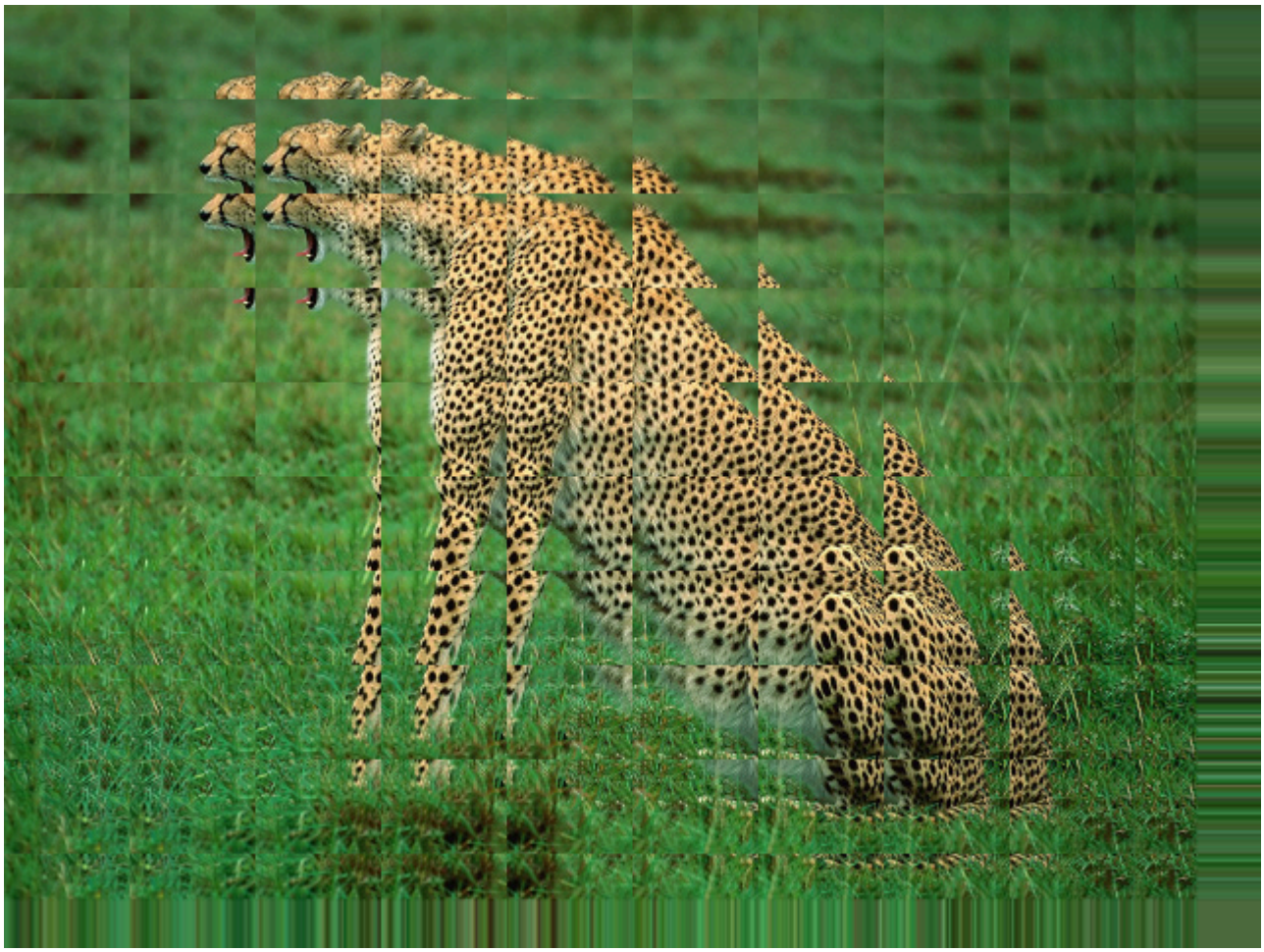
It is possible to draw parts of images by specifying *source* and *target* rectangles. The source rectangle describes the area that should be taken from the image and presented in the target rectangle.



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            val source = Rectangle(0.0, 0.0, 320.0, 240.0)
            val target = Rectangle(160.0, 120.0, 320.0, 240.0)
            drawer.image(image, source, target)
        }
    }
}
```

Link to the full example

Drawing many parts of images



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            val areas = (0..10).flatMap { y ->
                (0..10).map { x ->
                    val source = Rectangle(x * (width / 10.0), y * (height / 10.0), width / 5.0, height / 5.0)
                    val target = Rectangle(x * (width / 10.0), y * (height / 10.0), width / 10.0, height / 10.0)
                    source to target
                }
            }
            drawer.image(image, areas)
        }
    }
}
```

Link to the full example

Changing the appearance of images

A linear color transform can be applied to images by setting `Drawer.drawStyle.colorMatrix` to a `Matrix55` value.

Tinting

Tinting multiplies the image color with a *tint color*.

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            drawer.drawStyle.colorMatrix = tint(ColorRGBa.RED)
            drawer.image(image, 0.0, 0.0)
        }
    }
}
```

Link to the full example

Inverting

Drawing an image with inverted colors can be achieved by using the `invert` color matrix.

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            drawer.drawStyle.colorMatrix = invert
            drawer.image(image, 0.0, 0.0)
        }
    }
}
```

Link to the full example

Grayscale

Drawing an image in grayscale can be achieved by using the `grayscale` color matrix.

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            // -- the factors below determine the RGB mixing factors
            drawer.drawStyle.colorMatrix = grayscale(1.0 / 3.0, 1.0 / 3.0, 1.0 / 3.0)
            drawer.image(image)
        }
    }
}
```

Link to the full example

Concatenating color transforms

Color transforms can be combined using the multiplication operator. This is called transform concatenation. Keep in mind that transform concatenations are read from right to left, and in the following example we first apply the `grayscale` transform and then the `tint` transform.

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        extend {
            // -- here we concatenate the transforms using the multiplication operator.
            drawer.drawStyle.colorMatrix = tint(ColorRGBa.PINK) * grayscale()
            drawer.image(image)
        }
    }
}
```

Link to the full example

Color

In this chapter we discuss and demonstrate OPENRNDR's color functionality.

Basic color

OPENRNDR primarily uses red-green-blue(-alpha) color stored in `ColorRGBa` instances. `ColorRGBa`'s channels store values in the range [0, 1].

Predefined colors

```
ColorRGBa.BLACK
ColorRGBa.WHITE
ColorRGBa.RED
ColorRGBa.GREEN
ColorRGBa.BLUE
ColorRGBa.YELLOW
ColorRGBa.GRAY
ColorRGBa.PINK
```

Custom colors

Custom colors can be made using either the ColorRGBa constructor, or the `rgb` and `rgba` functions. Both use value ranges between 0.0 and 1.0.

```
// -- using the ColorRGBa constructor
val red = ColorRGBa(1.0, 0.0, 0.0)
val green = ColorRGBa(0.0, 1.0, 0.0)
val blue = ColorRGBa(0.0, 0.0, 1.0)
val blueOpaque = ColorRGBa(0.0, 0.0, 1.0, 0.5)

// -- using the rgb and rgba functions
val magenta = rgb(1.0, 0.0, 1.0)
val magentaOpaque = rgb(1.0, 0.0, 1.0, 0.5)
```

Conversion from hex color

RGB color is commonly communicated in hexadecimal codes. `ColorRGBa` provides simple tools to construct color from such hexadecimal codes.

```
// -- construct the OPENRNDR pink from hexadecimal code, using an integer argument
val color1 = ColorRGBa.fromHex(0xffc0cb)
// -- construct the OPENRNDR pink from hexadecimal code, using a string argument; the leading # is optional
val color2 = ColorRGBa.fromHex("#ffc0cb")
```

Color operations

The `ColorRGBa` class offers a number of tools to create variations of colors. For example `ColorRGBa.shade` can be used to create lighter or darker shades of a base color.

```
fun main() = application {
    program {
        extend {
            drawer.stroke = null
            val baseColor = ColorRGBa.PINK
            // -- draw 16 darker shades of pink
            for (i in 0..15) {
                drawer.fill = baseColor.shade(i / 15.0)
                drawer.rectangle(35.0 + (700 / 16.0) * i, 32.0, (700 / 16.0), 64.0)
            }
            // -- draw 16 lighter shades of pink
            for (i in 0..15) {
                drawer.fill = baseColor.shade(1.0 + i / 15.0)
                drawer.rectangle(35.0 + (700 / 16.0) * i, 96.0, (700 / 16.0), 64.0)
            }
        }
    }
}
```

Link to the full example

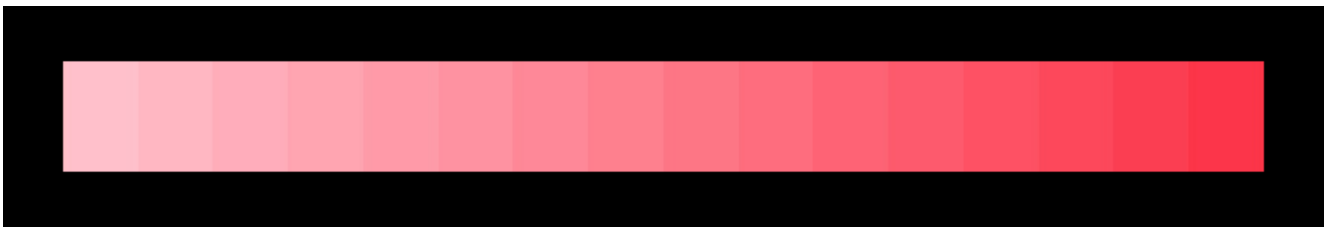
Using `ColorRGBa.opacify` colors can be made more or less opaque.

```
fun main() = application {
    program {
        extend {
            drawer.stroke = null
            val baseColor = ColorRGBa.PINK

            drawer.fill = ColorRGBa.GRAY.shade(0.5)
            drawer.rectangle(35.0, 32.0, 700.0, 64.00)

            // -- draw 16 darker shades of pink
            for (i in 0..15) {
                drawer.fill = baseColor.opacify(i / 15.0)
```

```
                drawer.rectangle(35.0 + (700 / 16.0) * i, 64.0, (700 / 16.0), 64.0)
            }
        }
    }
}
```

Link to the full example

Using `mix(ColorRGBa, ColorRGBa, Double)` colors can be mixed.



```
fun main() = application {
    program {
        extend {
            drawer.stroke = null
            val leftColor = ColorRGBa.PINK
            val rightColor = ColorRGBa.fromHex(0xFC3549)

            // -- draw 16 color mixes
            for (i in 0..15) {
                drawer.fill = mix(leftColor, rightColor, i / 15.0)
                drawer.rectangle(35.0 + (700 / 16.0) * i, 32.0, (700 / 16.0), 64.0)
            }
        }
    }
}
```

Link to the full example

Alternative color models

OPENRNDR offers a wide range of alternative color models. The alternative models use primaries different from red, green and blue.

Class name	Color space description
`ColorRGBa`	sRGB and linear RGB
`ColorHSVa`	Hue, saturation, value
`ColorHSLa`	Hue, saturation, lightness
`ColorXSVa`	Xue, saturation, value, *Kuler*-like colorspace
`ColorXSLa`	Xue, saturation, lightness, *Kuler*-like colorspace
`ColorXYZa`	CIE XYZ colorspace
`ColorYxya`	CIE Yxy colorspace
`ColorLABa`	LAB colorspace

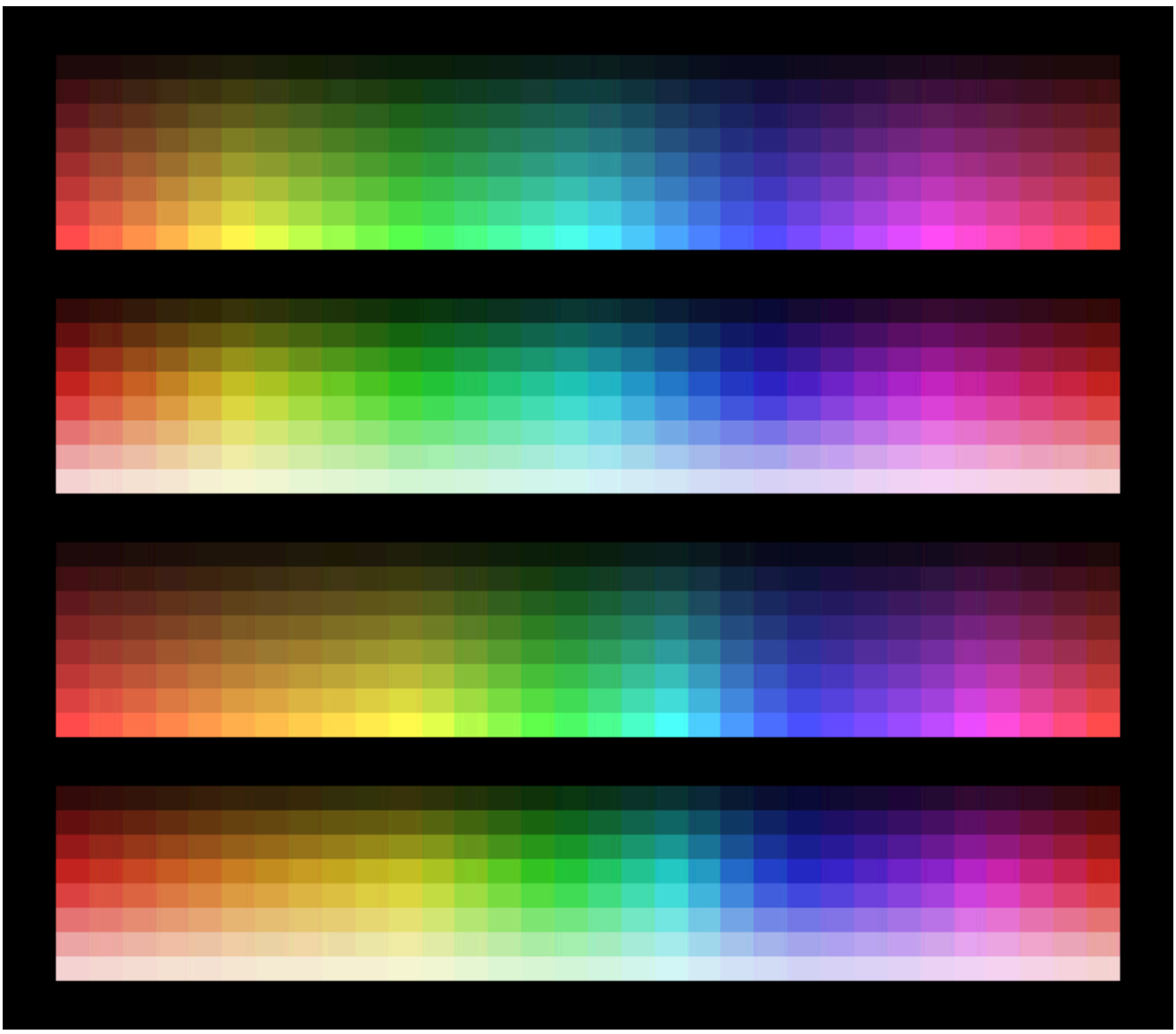
Class name	Color space description
`ColorLCHABa`	LCHab colorspace, a cylindrical variant of LAB
`ColorLSHABa`	LSHab colorspace, a cylindrical variant of LAB, chroma replaced with normalized saturation
`ColorLUVa`	LUV colorspace
`ColorLCHUVa`	LCHuv colorspace, a cylindrical variant of LUV
`ColorLSHUVa`	LSHuv colorspace, a cylindrical variant of LUV, chroma replaced with normalized saturation
`ColorATVa`	Coloroid color space, partial implementation

HSV, HSL, XSV and XSL color

HSV (hue-saturation-value) and HSL ("hue-saturation-lightness") are cylindrical color spaces.

XSV and XSL (for lack of a better name) are transformed versions of HSV and HSL in which the hue component has been stretched and compressed to make the color space better suited for artists. The spaces are better suited for artists because it has red-green and blue-yellow primaries. The XSV and XSL spaces are based on (if not the same as) the Adobe Kuler color spaces.

Below is an example of plots of color swatches for (from top to bottom) HSV, HSL, XSV and XSL. The adjusted hue of the XSV and XSL spaces is clearly visible.



```
fun main() = application {
    configure {
        width = 770
        height = 672
    }
    program {
        extend {
            drawer.stroke = null

            // -- draw hsv swatches
            for (j in 0..7) {
                for (i in 0..31) {
                    drawer.fill = ColorHSVa(360 * (i / 31.0), 0.7, 0.125 + j / 8.0).toRGBa()
                    drawer.rectangle(35.0 + (700 / 32.0) * i, 32.0 + j * 16.0, (700 / 32.0), 16.0)
                }
            }

            // -- draw hsl swatches
            drawer.translate(0.0, 160.0)
            for (j in 0..7) {
                for (i in 0..31) {
                    drawer.fill = ColorHSLa(360 * (i / 31.0), 0.7, 0.125 + j / 9.0).toRGBa()
                    drawer.rectangle(35.0 + (700 / 32.0) * i, 32.0 + j * 16.0, (700 / 32.0), 16.0)
                }
            }

            // -- draw xsv (Kuler) swatches
            drawer.translate(0.0, 160.0)
            for (j in 0..7) {
                for (i in 0..31) {
                    drawer.fill = ColorXSVa(360 * (i / 31.0), 0.7, 0.125 + j / 8.0).toRGBa()
                    drawer.rectangle(35.0 + (700 / 32.0) * i, 32.0 + j * 16.0, (700 / 32.0), 16.0)
                }
            }

            // -- draw xsl (Kuler) swatches
            drawer.translate(0.0, 160.0)
            for (j in 0..7) {
                for (i in 0..31) {
                    drawer.fill = ColorXSLa(360 * (i / 31.0), 0.7, 0.125 + j / 9.0, 1.0).toRGBa()
                    drawer.rectangle(35.0 + (700 / 32.0) * i, 32.0 + j * 16.0, (700 / 32.0), 16.00)
                }
            }
        }
    }
}
```

Link to the full example

Managing draw style

In previous sections we briefly talked about controlling the appearance of drawing primitives using `fill`, `stroke` and `lineCap`. In this section we present the existing draw style properties.

DRAWSTYLE PROPERTIES

Property	Type	Default	Description
`fill`	`ColorRGBa?`	`ColorRGBa.WHITE`	The fill color
`stroke`	`ColorRGBa?`	`ColorRGBa.BLACK`	The stroke color
`strokeWeight`	`Double`	`1.0`	The stroke weight
`smooth`	`Boolean`	`true`	Should contours and segments look smooth or pixelated?
`lineCap`	`LineCap`	`LineCap.BUTT`	The segment cap style
`lineJoin`	`LineJoin`	`LineJoin.MITER`	The segment join style
`miterLimit`	`Double`	`4.0`	The segment join maximum miter length
`fontMap`	`FontMap?`	`null`	The font to use
`kerning`	`KernMode`	`KernMode.METRIC`	The kerning mode used for rendering text
`textSetting`	`TextSettingMode`	`TextSettingMode.SUBPIXEL`	The text setting mode
`colorMatrix`	`Matrix55`	`Matrix55.IDENTITY`	The color matrix (used for images)
`channelWriteMask`	`ChannelMask`	`ChannelMask.ALL`	The channel write mask
`shadeStyle`	`ShadeStyle?`	`null`	The shade style
`blendMode`	`BlendMode`	`BlendMode.OVER`	The blend mode
`quality`	`DrawQuality`	`DrawQuality.QUALITY`	A hint that controls the quality of some primitives
`cullTestPass`	`CullTestPass`	`CullTestPass.ALWAYS`	What fragments be rendered: back or front facing?
`depthTestPass`	`DepthTestPass`	`DepthTestPass.ALWAYS`	When should fragments pass the depth test
`depthWrite`	`Boolean`	`false`	Should the fragment depth be written to the depth buffer?
`stencil`	`StencilStyle`	`StencilStyle()`	The stencil style
`frontStencil`	`StencilStyle`	`StencilStyle()`	The stencil style for front-facing fragments
`backStencil`	`StencilStyle`	`StencilStyle()`	The stencil style for back-facing fragments
`clip`	`Rectangle?`	`null`	A rectangle that describes where drawing will take place

The active draw style

```
val active = drawer.drawStyle.copy()
```

The draw style stack

Styles can be pushed on and popped from a stack maintained by `Drawer`.

```
extend {
    drawer.pushStyle()
    drawer.fill = ColorRGBa.PINK
    drawer.rectangle(100.0, 100.0, 100.0, 100.0)
    drawer.popStyle()
    // colorMatrix, channelWriteMask, blendMode, quality, stencil, frontStencil, backStencil, clip
    drawer.drawStyle
}
```

The `Drawer` provides a helper function called `isolated {}` that pushes style and transforms on a their respective stacks, executes the user code and pops style and transforms back.

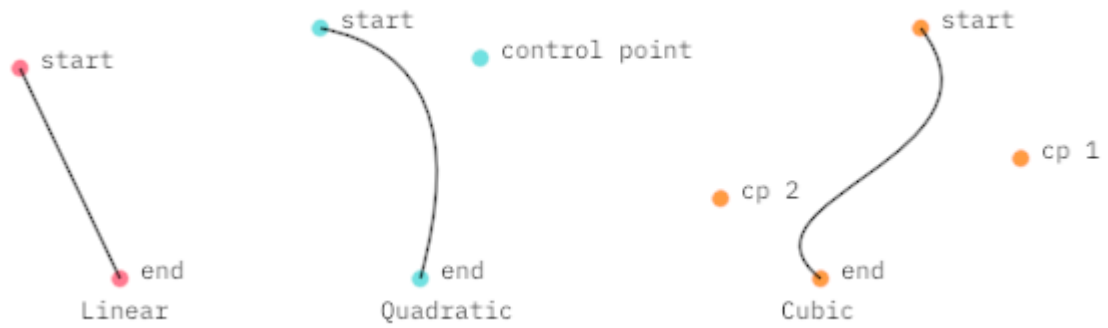
```
extend {
    drawer.isolated {
        fill = ColorRGBa.PINK
        rectangle(100.0, 100.0, 100.0, 100.0)
    }
}
```

Segment, ShapeContour and Shape

OPENRNDR offers a lot of tools for creating and drawing two dimensional shapes.

Segment

The basic element for constructing shapes is the `Segment`: a Bézier curve with a start point, an end point and zero, one or two control points.



Constructing segments

```
// Linear Segment: start point, end point
val seg1 = Segment2D(Vector2(50.0, 55.0), Vector2(100.0, 160.0))

// Quadratic Segment: start point, control point, end point
val seg2 = Segment2D(Vector2(200.0, 35.0), Vector2(280.0, 50.0), Vector2(250.0, 160.0))

// Cubic Segment: start point, control point, control point, end point
val seg3 = Segment2D(Vector2(500.0, 35.0), Vector2(550.0, 100.0), Vector2(400.0, 120.0), Vector2(450.0, 160.0))
```

Drawing segments

```
// Draw one segment
drawer.segment(seg3)

// Draw multiple segments
drawer.segments(listOf(seg1, seg2, seg3))
```

Note that `Segment`, like `Circle`, `Rectangle` and other geometric entities in OPENRNDR, are mathematical representations which can be rendered to the screen, but this is not necessary. A reason to create such geometries without displaying them is to serve as building blocks for constructing more complex designs. We can achieve this by querying curve properties.

Segment properties

The `Segment` class provides multiple methods to query its properties. In the following examples, the `ut` argument is a normalized value that indicates a position in the segment between 0.0 (at the start) and 1.0 (at the end).

```
// Get a point on the curve near the start.
val pos = seg.position(ut = 0.1)

// Get the normal vector near the end.
// This is a vector of length 1.0 perpendicular to the curve.
val normal = seg.normal(ut = 0.9)

// Get the bounding box of the curve as a Rectangle instance.
val rect = seg.bounds

// Get the length of the curve.
val length = seg.length

// Get the point on the curve which is nearest to a given point.
val nearest = seg.nearest(Vector2(50.0, 50.0)).position

// Get 20 equally spaced curve points
val points = seg.equidistantPositions(20)
```

The list of available methods can be found at the API website or in the source code.

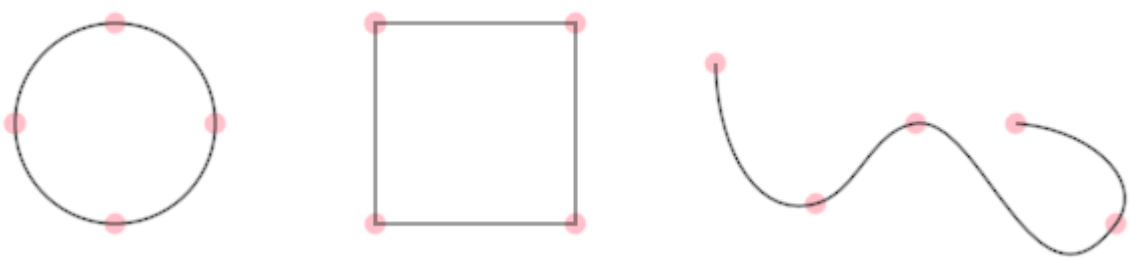
Modifying segments

Several methods return a new Segment based on the original one.

```
// Split a segment at the center returning two segments
val segments = seg.split(0.5)

// Get the center part of a segment
val subSegment = seg.sub(0.25, 0.75)

// Get the segment reversed (the start becomes the end)
val revSegment = seg.reverse

// Get the segment offset by the given distance
val offsetSegment = seg.offset(5.0)
```

ShapeContour

A `ShapeContour` is a collection of `Segment` instances in which each segment ends where the next one starts. A ShapeContour can be closed like the letter O or open like the letter S. It can be used to describe simple shapes like a square, or more complex ones.



Three ShapeContours with 4 segments each. The one on the right is open.

Constructing a ShapeContour using the ContourBuilder

The `ContourBuilder` class offers a simple way of producing complex two dimensional shapes. It employs a vocabulary that is familiar to those who have used SVG.

- `moveTo(position)` move the cursor to the given position
- `lineTo(position)` insert a line contour starting from the cursor, ending at the given position
- `moveOrLineTo(position)` move the cursor if no cursor was previously set or draw a line
- `curveTo(control, position)` insert a quadratic bezier curve starting from the cursor, ending at position
- `curveTo(controlA, controlB, position)` insert a cubic bezier curve starting from the cursor, ending at position
- `continueTo(position)` inside a quadratic bezier curve starting from the cursor and reflecting the tangent of the last control
- `continueTo(controlB, position)` insert a cubic spline
- `arcTo(radiusX, radiusY, largeAngle, sweepFlag, position)`
- `close()` close the contour
- `cursor` a `Vector2` instance representing the current position
- `anchor` a `Vector2` instance representing the current anchor

Let's create a simple `Contour` and draw it. The following program shows how to use the contour builder to create a triangular contour.


```
fun main() = application {
    program {
        extend {
            val c = contour {
                moveTo(Vector2(width / 2.0 - 120.0, height / 2.0 - 120.00))
                // -- here `cursor` points to the end point of the previous command
                lineTo(cursor + Vector2(240.0, 0.0))
                lineTo(cursor + Vector2(0.0, 240.0))
                lineTo(anchor)
                close()
            }
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.contour(c)
        }
```

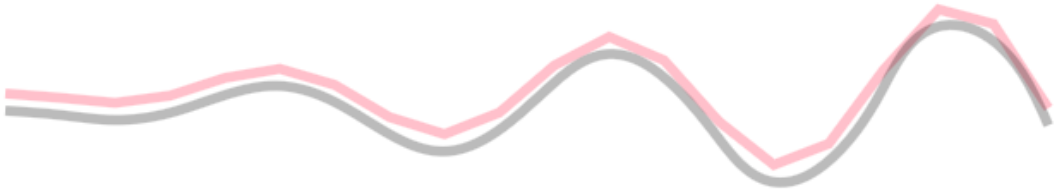
```
    }
}
```

Link to the full example

Constructing a ShapeContour from points

We can use `.fromPoints()` to connect points with straight segments.

The `hobbyCurve` method, found in `orx-shapes`, can be used to create smooth curves.



```
fun main() = application {
    program {
        val points = List(20) {
            Vector2(20.0 + it * 32.0, 100.0 + sin(it * 1.0) * it * 3)
        }
        val wavyContour = ShapeContour.fromPoints(points, closed = false)
        val smoothContour = hobbyCurve(points, closed = false)

        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = null
            drawer.strokeWeight = 5.0
            drawer.stroke = ColorRGBa.PINK
            drawer.contour(wavyContour)

            drawer.translate(0.0, 10.0) // displace 10px down
            drawer.stroke = ColorRGBa.BLACK.opacify(0.5)
            drawer.contour(smoothContour)
        }
    }
}
```

Link to the full example

Constructing a ShapeContour from segments

Notice how each segment starts where the last one ends.

```
fun main() = application {
    program {
        val segments = listOf(Segment2D(Vector2(10.0, 100.0), Vector2(200.0, 80.0)), // Linear Bézier Segment
        Segment2D(Vector2(200.0, 80.0), Vector2(250.0, 280.0), Vector2(400.0, 80.0)), // Quadratic Bézier segment
        Segment2D(Vector2(400.0, 80.0), Vector2(450.0, 180.0), Vector2(500.0, 0.0), Vector2(630.0, 80.0))) // Cubic
Bézier segment
        val horizontalContour = ShapeContour.fromSegments(segments, closed = false)

        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.strokeWeight = 5.0
            drawer.stroke = ColorRGBa.PINK
            drawer.contour(horizontalContour)
        }
    }
}
```

Link to the full example

Constructing a ShapeContour from a primitive

Primitives like `Rectangle`, `Circle` and `LineSegment` can be easily converted into a `ShapeContour`.

```
val c1 = Circle(200.0, 200.0, 50.0).contour
```

Even more ways to construct a ShapeContour

Take a look at `orx-turtle` and `orx-shapes` for other ways to create contours, including regular polygons, rounded rectangles and more.

Drawing a ShapeContour

```
// Draw one contour
drawer.contour(contour1)

// Draw multiple contours
drawer.contours(listOf(contour1, contour2, contour3))
```

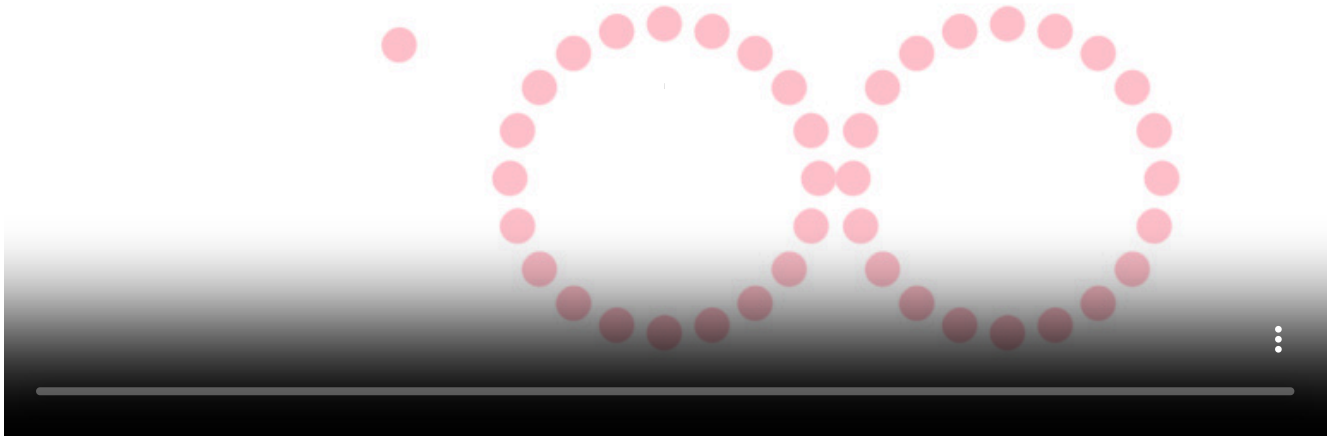
Note that if the contour is closed, the current fill color is used.

ShapeContour properties

The `ShapeContour` provides methods to query its properties similar to the ones found in `Segment`.

```
// Get a point on the contour near the start.
val pos = contour.position(ut = 0.1)

// Get the normal vector near the end.
// This is a vector of length 1.0 perpendicular to the curve.
val normal = contour.normal(ut = 0.9)

// Get the bounding box of the curve as a Rectangle instance.
val rect = contour.bounds

// Get the length of the curve.
val length = contour.length

// Get the point on the curve which is nearest to a given point.
val nearest = contour.nearest(Vector2(50.0, 50.0)).position

// Get 20 equally spaced curve points
val points = contour.equidistantPositions(20)
```

An example of using `.position()` and `.equidistantPositions()`:



```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.stroke = null
            drawer.fill = ColorRGBa.PINK

            val point = Circle(185.0, height / 2.0, 90.0).contour.position((seconds * 0.1) % 1.0)
            drawer.circle(point, 10.0)

            val points0 = Circle(385.0, height / 2.0, 90.0).contour.equidistantPositions(20)
            drawer.circles(points0, 10.0)

            val points1 = Circle(585.0, height / 2.0, 90.0).contour.equidistantPositions((cos(seconds) * 10.0 +
30.0).toInt())
            drawer.circles(points1, 10.0)
        }
```

```
    }
}
```

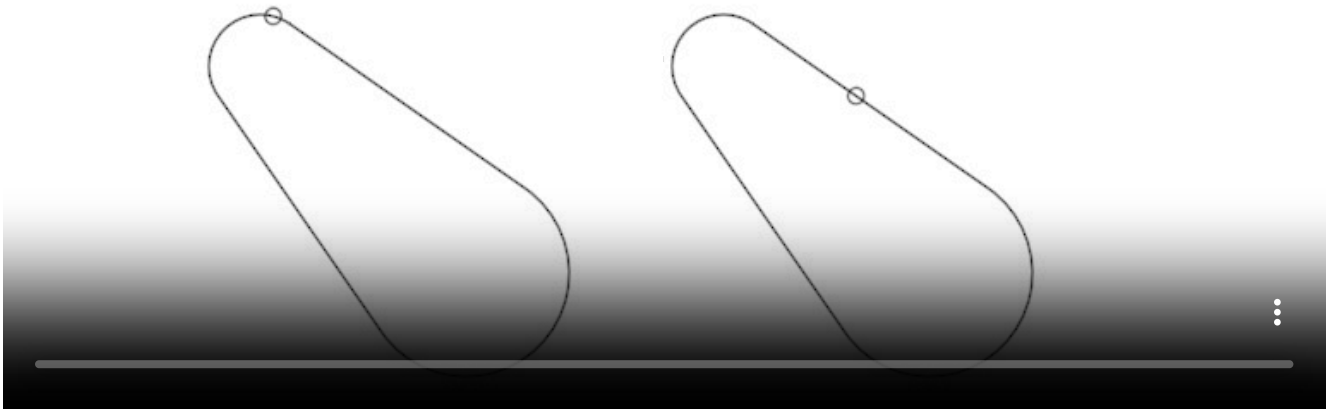
Link to the full example

The list of available methods can be found at the API website or in the source code.

Rectified ShapeContour

The `ut` argument in the `ShapeContour.position()` and `ShapeContour.normal()` methods does not specify a linear position between the start and the end of the contour.

By using rectified contours (defined in `orx-shapes`) we can work with evenly spaced points on contours, or animate elements traveling on a contour at the desired speed even if the contour segments vary greatly in length.



```
fun main() = application {
    program {
        val c = Pulley(Circle(Vector2.ZERO, 30.0), Circle(Vector2.ONE * 120.0, 60.0)).contour
        val cr = c.rectified()

        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = null

            // Go from 0.0 to 1.0 in two seconds
            // slowing down at both ends
            val t = cos(kotlin.math.PI * (seconds % 2.0) / 2.0) * 0.5 + 0.5

            drawer.translate(150.0, 100.0)
            drawer.contour(c)
            // Note how segment length affects the speed
            drawer.circle(c.position(t), 5.0)

            drawer.translate(270.0, 0.0)
            drawer.contour(c)
            // The rectified contour provides a smooth animation
            drawer.circle(cr.position(t), 5.0)
        }
    }
}
```

Link to the full example

Modifying a ShapeContour

SUB()

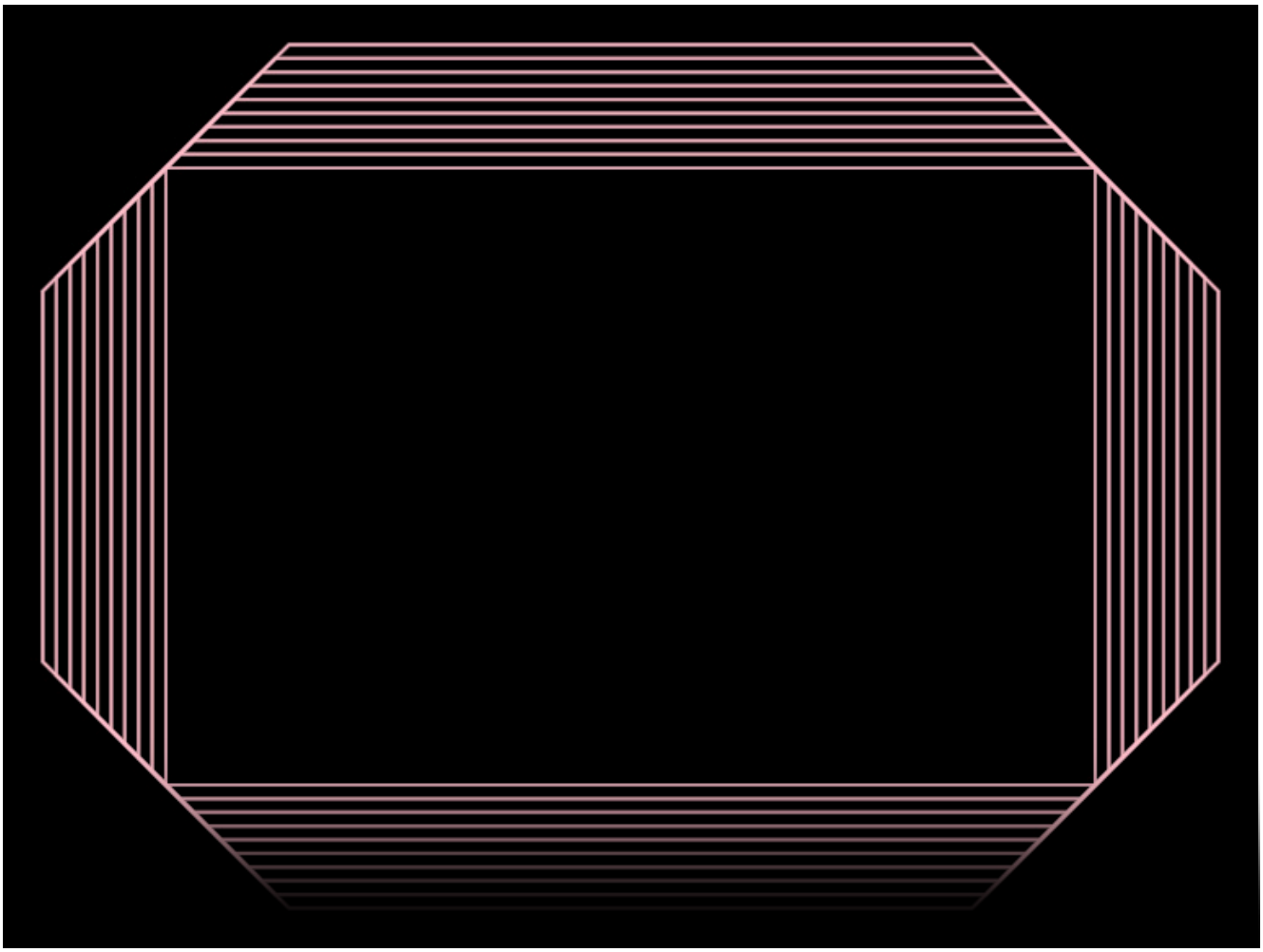
A contour can be cut into a shorter contour using `ShapeContour.sub()`.

```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = null
            drawer.stroke = ColorRGBa.PINK
            drawer.strokeWeight = 4.0

            val sub0 = Circle(185.0, height / 2.0, 100.0).contour.sub(0.0, 0.5 + 0.50 * sin(seconds))
            drawer.contour(sub0)

            val sub1 = Circle(385.0, height / 2.0, 100.0).contour.sub(seconds * 0.1, seconds * 0.1 + 0.1)
            drawer.contour(sub1)

            val sub2 = Circle(585.0, height / 2.0, 100.0).contour.sub(-seconds * 0.05, seconds * 0.05 + 0.1)
            drawer.contour(sub2)
        }
    }
}
```

Link to the full example

OFFSET()

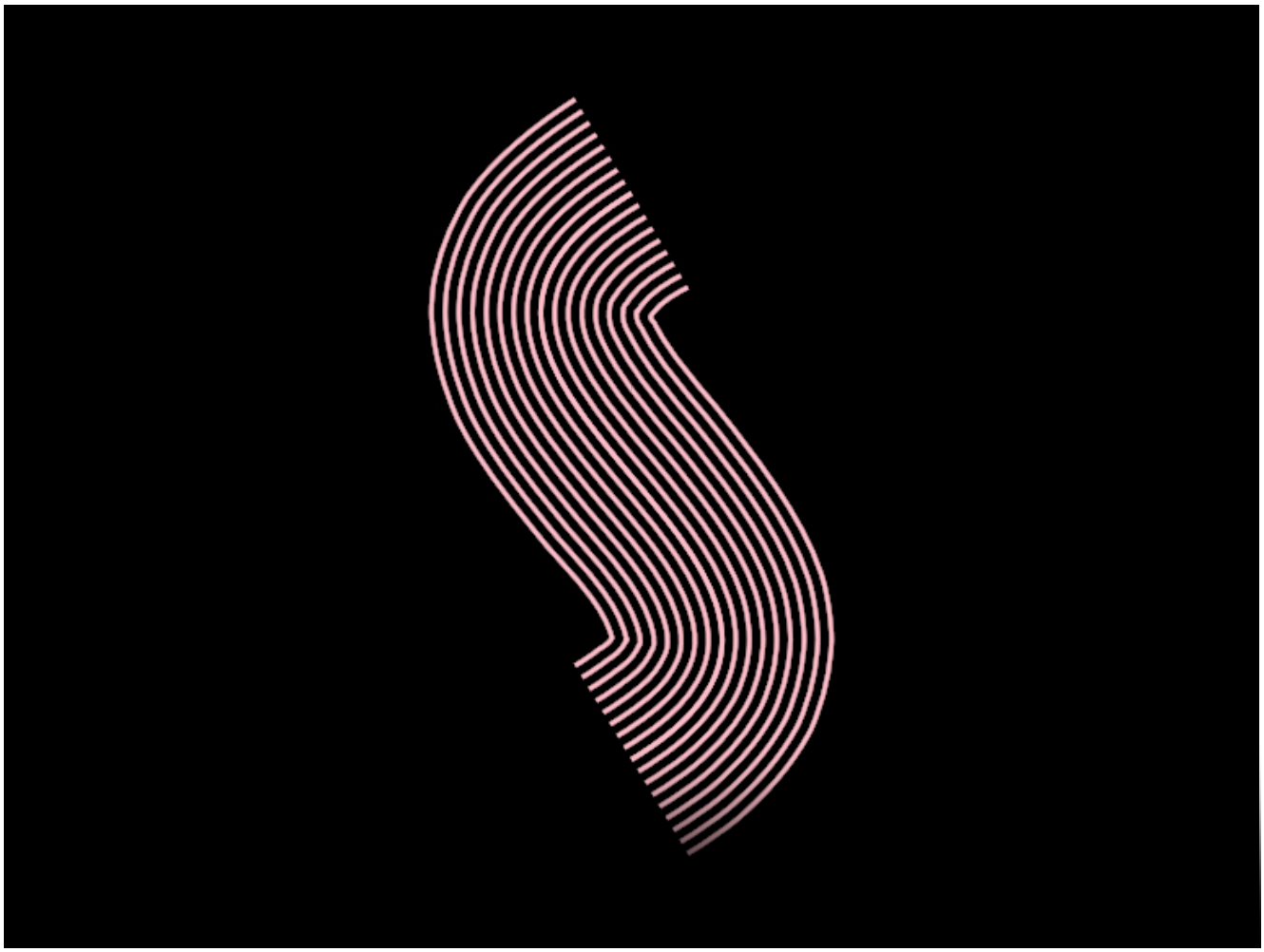
The function `ShapeContour.offset` can be used to create an offset version of a contour.



```
fun main() = application {
    program {
        // -- create a contour from a Rectangle object
        val c = Rectangle(100.0, 100.0, width - 200.0, height - 200.0).contour.reversed

        extend {
            drawer.fill = null
            drawer.stroke = ColorRGBa.PINK
            drawer.contour(c)
            for (i in 1 until 10) {
                val o = c.offset(cos(seconds + 0.5) * i * 10.0, SegmentJoin.BEVEL)
                drawer.contour(o)
            }
        }
    }
}
```

Link to the full example

`ShapeContour.offset` can also be used to offset curved contours. The following demonstration shows a single cubic bezier offset at multiple distances.



```
fun main() = application {
    program {
        val c = contour {
            moveTo(width * (1.0 / 2.0), height * (1.0 / 5.0))
            curveTo(width * (1.0 / 4.0), height * (2.0 / 5.0), width * (3.0 / 4.0), height * (3.0 / 5.0), width * (2.0
/ 4.0), height * (4.0 / 5.0))
        }
        extend {
            drawer.stroke = ColorRGBa.PINK
            drawer.strokeWeight = 2.0
            drawer.lineJoin = LineJoin.ROUND
            drawer.contour(c)
            for (i in -8..8) {
                val o = c.offset(i * 10.0 * cos(seconds + 0.5))
                drawer.contour(o)
            }
        }
    }
}
```

Link to the full example

REVERSED, CLOSE(), TRANSFORM(), ...

For more properties and methods explore the API website or the source code.

Shape

OPENRNDR uses `Shape` to represent planar shapes. We can think of a `Shape` as a group of `ShapeContour` instances, where each `ShapeContour` is a sequence of one or more Bézier `Segment`.

Constructing a Shape using the shape builder

Let's create a `Shape` using the *shape builder*. The shape is created using two contours, one for the *outline* of the shape, and one for the *hole* in the shape


```
fun main() = application {
    program {
        extend {
            val s = shape {
                contour {
                    moveTo(Vector2(width / 2.0 - 120.0, height / 2.0 - 120.00))
                    lineTo(cursor + Vector2(240.0, 0.0))
                    lineTo(cursor + Vector2(0.0, 240.0))
                    lineTo(anchor)
                    close()
                }
                contour {
                    moveTo(Vector2(width / 2.0 - 80.0, height / 2.0 - 100.0))
                    lineTo(cursor + Vector2(190.0, 0.0))
                    lineTo(cursor + Vector2(0.0, 190.00))
                    lineTo(anchor)
                    close()
                }
            }
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.shape(s)
        }
    }
}
```

Link to the full example

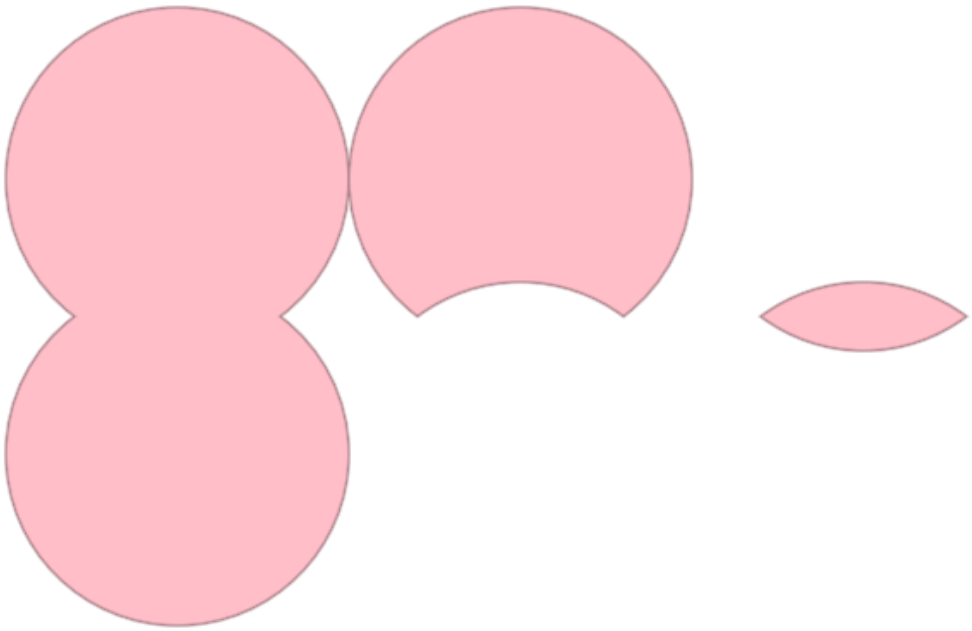
Constructing a Shape from a primitive

Primitives like `Rectangle`, `Circle`, `LineSegment` and `ShapeContour` can be easily converted into a `Shape`.

```
val s = Circle(200.0, 200.0, 50.0).shape
```

Shape Boolean-operations

Boolean-operations can be performed on shapes using the `compound {}` builder. There are three kinds of compounds: *union*, *difference* and *intersection*, all three of them are shown in the example below.

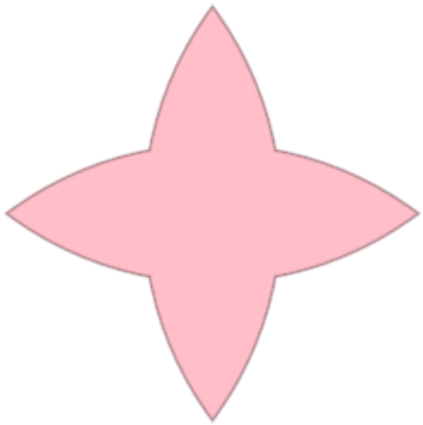


```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = ColorRGBa.PINK.shade(0.7)

            // -- shape union
            val su = compound {
                union {
                    shape(Circle(185.0, height / 2.0 - 80.0, 100.0).shape)
                    shape(Circle(185.0, height / 2.0 + 80.0, 100.0).shape)
                }
            }
            drawer.shapes(su)

            // -- shape difference
            val sd = compound {
                difference {
                    shape(Circle(385.0, height / 2.0 - 80.0, 100.0).shape)
                    shape(Circle(385.0, height / 2.0 + 80.0, 100.0).shape)
                }
            }
            drawer.shapes(sd)

            // -- shape intersection
```

```
            val si = compound {
                intersection {
                    shape(Circle(585.0, height / 2.0 - 80.0, 100.0).shape)
                    shape(Circle(585.0, height / 2.0 + 80.0, 100.0).shape)
                }
            }
            drawer.shapes(si)
        }
    }
}
```

Link to the full example

The *compound builder* is actually a bit more clever than what the previous example demonstrated because it can actually work with an entire tree of compounds. Demonstrated below is the *union* of two *intersections*.

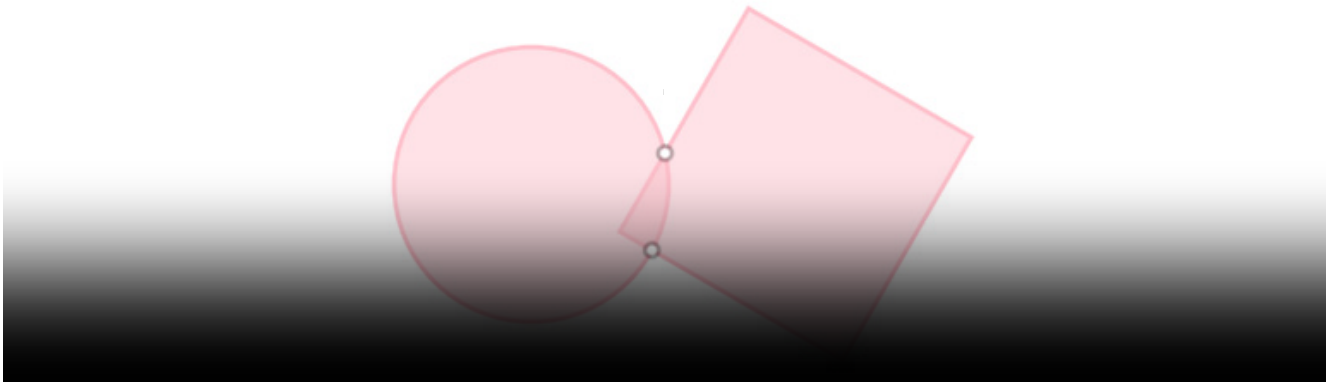


```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = ColorRGBa.PINK.shade(0.7)
            val cross = compound {
                union {
                    intersection {
                        shape(Circle(width / 2.0 - 160.0, height / 2.0, 200.0).shape)
                        shape(Circle(width / 2.0 + 160.0, height / 2.0, 200.0).shape)
                    }
                    intersection {
                        shape(Circle(width / 2.0, height / 2.0 - 160.0, 200.0).shape)
                        shape(Circle(width / 2.0, height / 2.0 + 160.0, 200.0).shape)
                    }
                }
            }
            drawer.shapes(cross)
        }
    }
}
```

Link to the full example

Intersections

Extension methods are provided to find intersections between `Shape`, `ShapeContour` and `Segment` instances.



```
fun main() = application {
    program {
        extend {
            // A rotation transformation to apply to the rectangle
            val rotation = transform {
                translate(width * 0.6, height * 0.5)
                rotate(seconds * 18)
            }

            val circle = Circle(width * 0.4, height * 0.5, 80.0).contour
            val rotatingRect = Rectangle.fromCenter(Vector2.ZERO, 150.0).contour.transform(rotation)

            val intersections = circle.intersections(rotatingRect)

            drawer.clear(ColorRGBa.WHITE)
            drawer.strokeWeight = 2.0
            drawer.stroke = ColorRGBa.PINK
            drawer.fill = ColorRGBa.PINK.opacify(0.5)

            drawer.contour(circle)
            drawer.contour(rotatingRect)

            // Draw intersections as small circles
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = ColorRGBa.BLACK.opacify(0.5)
            drawer.circles(intersections.map {
                it.position
            }, 5.0)
        }
    }
}
```

Link to the full example

Drawing text

OPENRNDR comes with support for rendering bitmap text. There are two modes of operation for writing text, a direct mode that simply writes a string of text at the requested position, and a more advanced mode that can place texts in a designated text area.

As an alternative to bitmap texts, which are stored as an image containing characters rendered at a specific size, it is also possible to obtain glyph contours to draw texts at any scale and to query curve properties.

Simple text rendering

To render simple texts we first make sure a font is loaded and assigned to `drawer.fontMap`, we then use `drawer.text` to draw the text.


```
fun main() = application {
    program {
        val font = loadFont("data/fonts/default.otf", 48.0)
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fontMap = font
            drawer.fill = ColorRGBa.BLACK
            drawer.text("HELLO WORLD", width / 2.0 - 100.0, height / 2.0)
        }
```

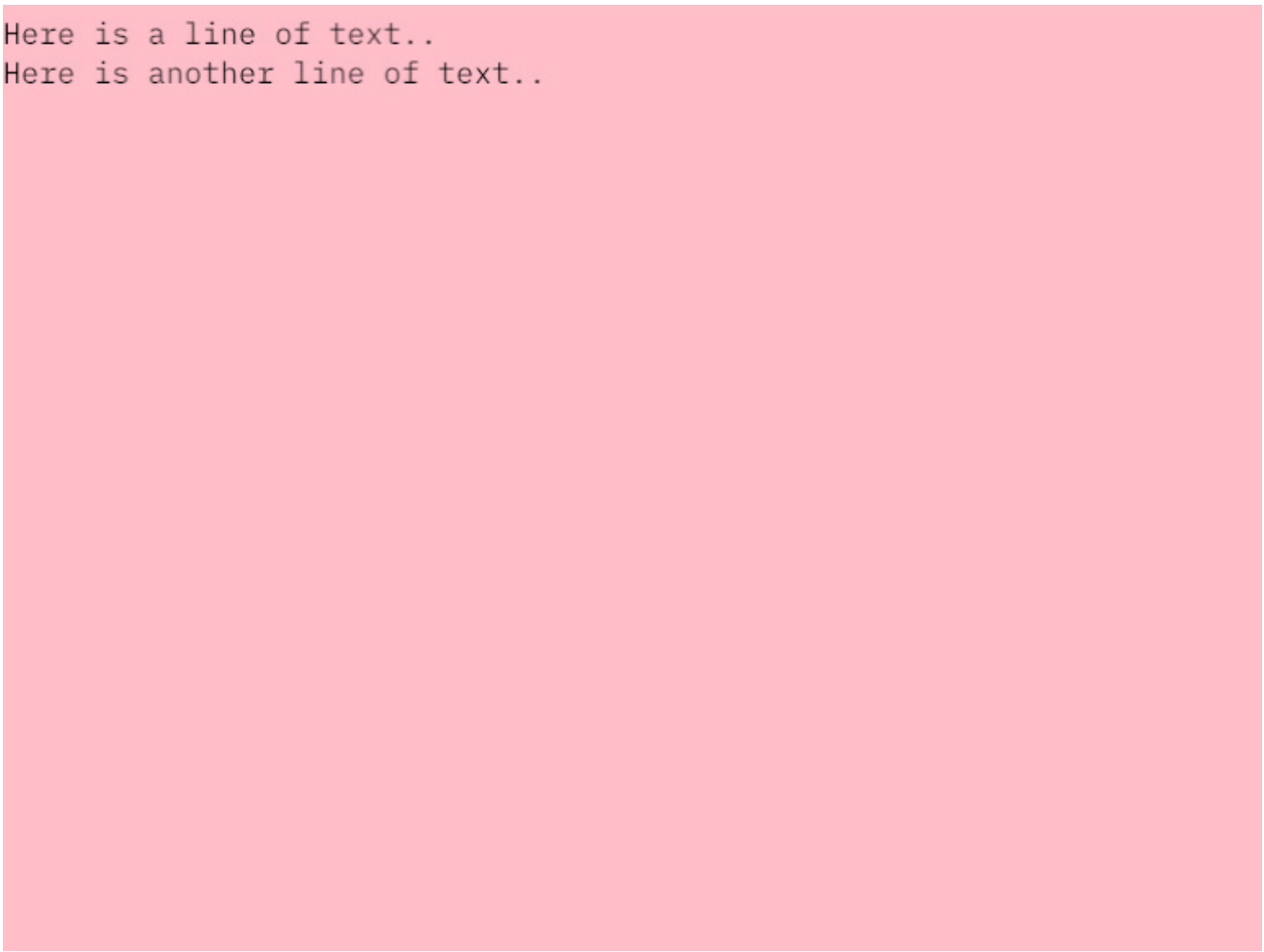
```
    }
}
```

Link to the full example

Advanced text rendering

OPENRNDR comes with a `Writer` class that allows for basic typesetting. The `Writer` tool is based on the concept of text box and a cursor.

Its use is easiest demonstrated through an example:




```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        val font = loadFont("data/fonts/default.otf", 24.0)
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fontMap = font
            drawer.fill = ColorRGBa.BLACK

            writer {
                newLine()
                text("Here is a line of text..")
```

```
                newLine()
                text("Here is another line of text..")
            }
        }
    }
}
```

Link to the full example

Specifying the text area

The `box` field of `Writer` is used to specify where text should be written. Let's set the text area to a 300 by 300 pixel rectangle starting at (40, 40).

We see that the text is now drawn with margins above and left of the text, and that the second line of text is set on two rows.

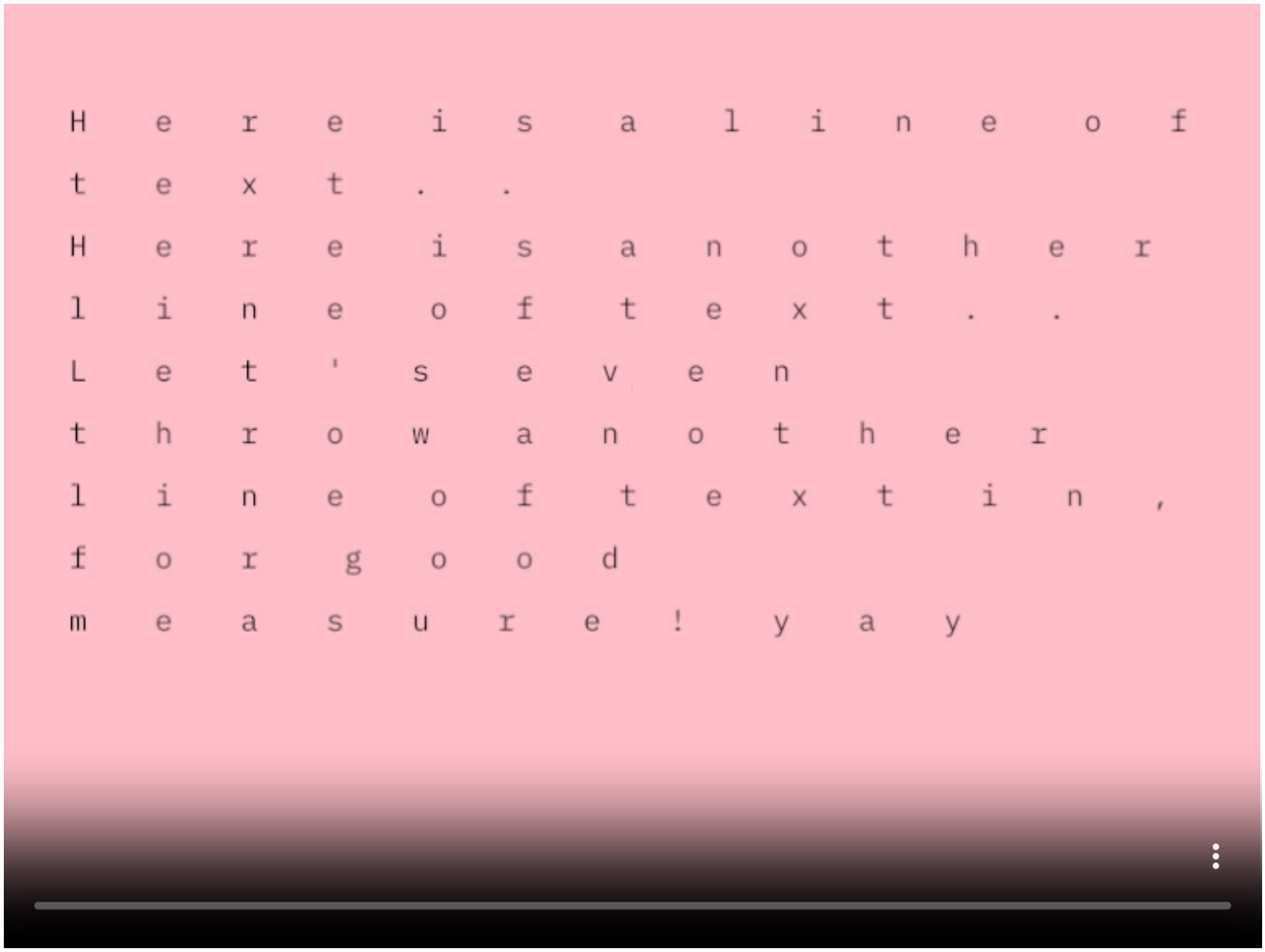

```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        val font = loadFont("data/fonts/default.otf", 24.0)
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fontMap = font
```

```
            drawer.fill = ColorRGBa.BLACK

            writer {
                box = Rectangle(40.0, 40.0, 300.0, 300.0)
                newLine()
                text("Here is a line of text..")
                newLine()
                text("Here is another line of text..")
            }
        }
    }
}
```

Link to the full example

Text properties

Text tracking -the horizontal space between characters- and leading -the vertical space between lines- can be set using `Writer.style.leading` and `Writer.style.tracking`.



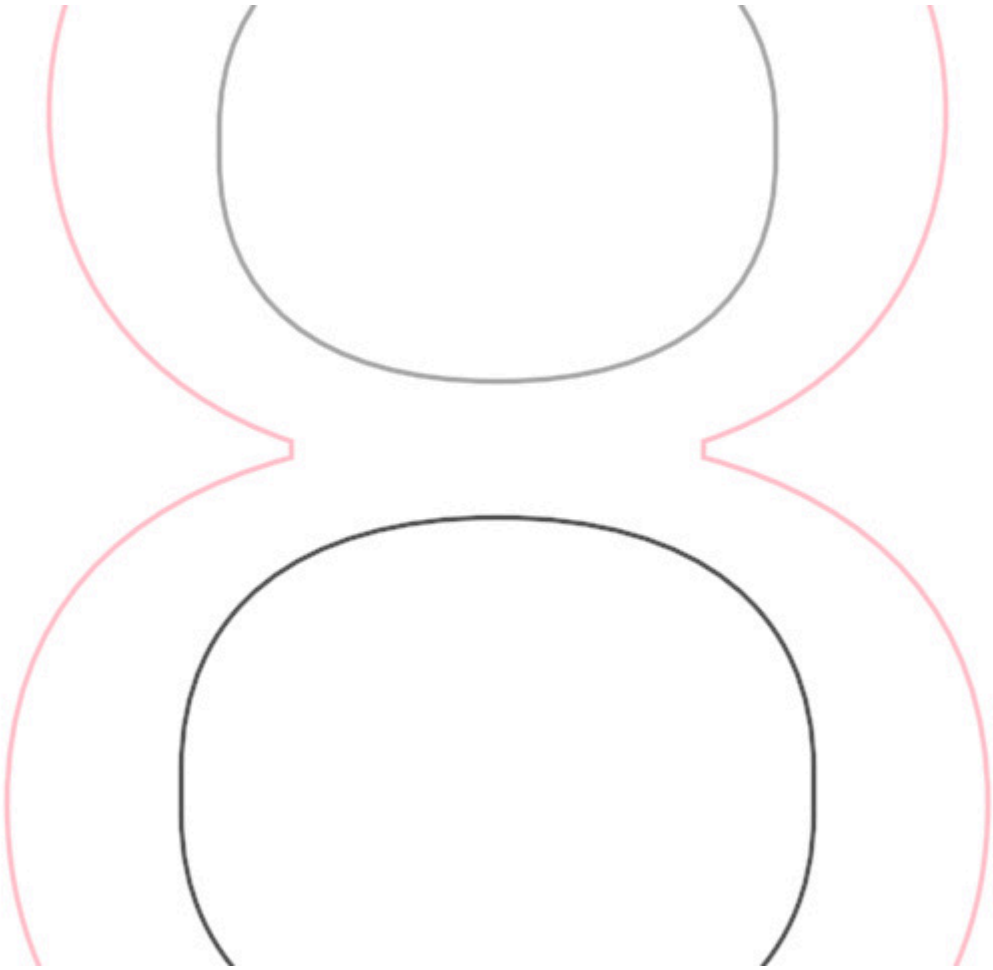

```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        val font = loadFont("data/fonts/default.otf", 24.0)
```

```
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fontMap = font
            drawer.fill = ColorRGBa.BLACK

            writer {
                // -- animate the text leading
                leading = cos(seconds) * 20.0 + 24.0
                // -- animate the text tracking
                tracking = sin(seconds) * 20.0 + 24.0
                box = Rectangle(40.0, 40.0, width - 80.0, height - 80.0)
                newLine()
                text("Here is a line of text..")
                newLine()
                text("Here is another line of text..")
                newLine()
                text("Let's even throw another line of text in, for good measure! yay")
            }
        }
    }
}
```

Link to the full example

Working with text contours

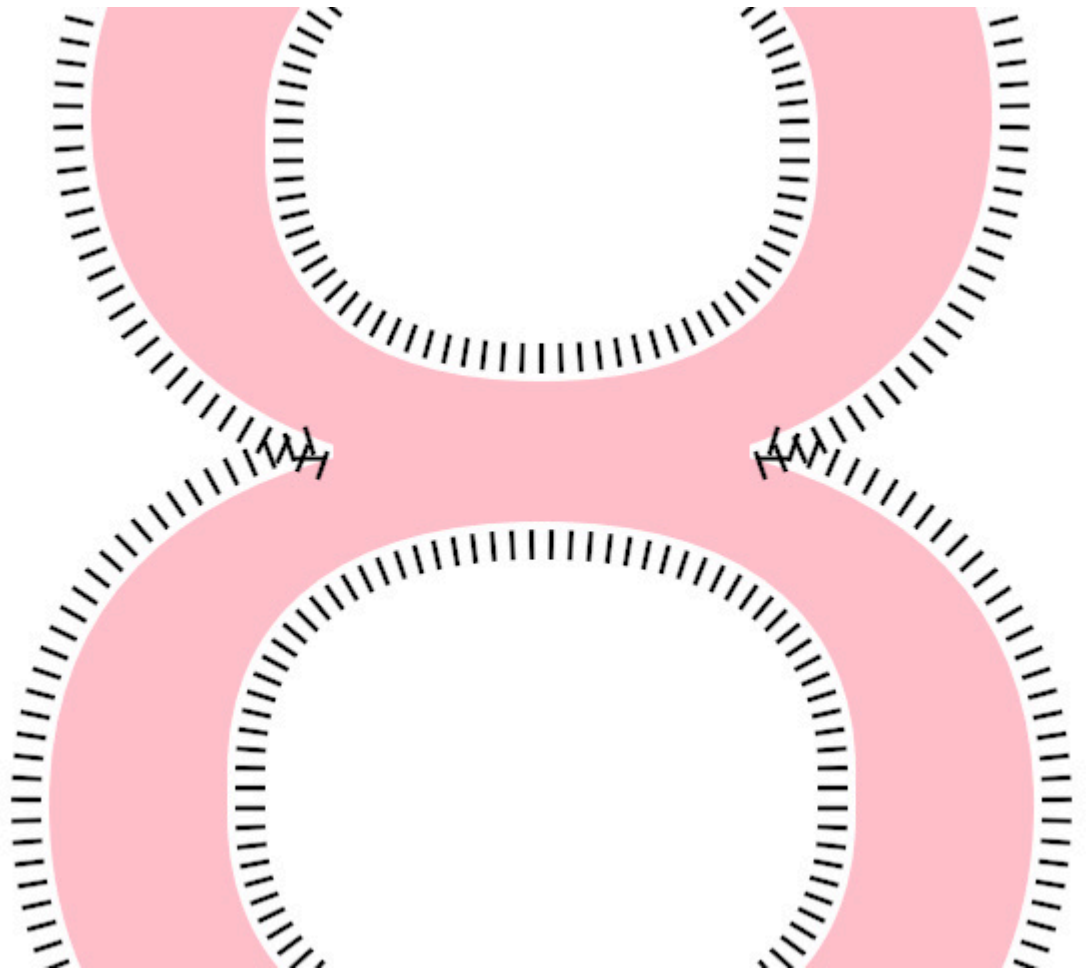
To load the vector data of a font file use the `loadFace()` method, then call the `.glyphForCharacter()` method to obtain a Shape representing a character.



```
fun main() = application {
    program {
        val face = loadFace("data/fonts/default.otf")
        val shape = face.glyphForCharacter(character = '8').shape(scale = 1.0)

        extend {
            drawer.clear(ColorRGBa.WHITE)
            // Center the shape on the screen
            drawer.translate(drawer.bounds.center - shape.bounds.center)

            drawer.fill = null
            drawer.strokeWeight = 2.0

            // Draw each contour found in the character '8' with a different color
            shape.contours.forEachIndexed { i, it ->
                drawer.stroke = listOf(ColorRGBa.PINK, rgb(0.33), rgb(0.66))[i]
                drawer.contour(it)
            }
        }
    }
}
```

Link to the full example

This example visualizes normal vectors around the contour of the character ‘8’, evenly spaced every 10 pixels.



```
fun main() = application {
    program {
        val face = loadFace("data/fonts/default.otf")
        val shape = face.glyphForCharacter('8').shape(1.0)

        // Map each contour in the shape to a list of LineSegment,
        // then combine the resulting lists by calling `.flatten()`.
        val normals = shape.contours.map { c ->
            // Work with rectified contours so `t` values are evenly spaced.
            val rc = c.rectified()
            val stepCount = (c.length / 10).toInt()
            List(stepCount) {
                val t = it / stepCount.toDouble()
                LineSegment(rc.position(t) + rc.normal(t) * 5.0, rc.position(t) + rc.normal(t) * 20.0)
            }
        }.flatten()
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.translate(drawer.bounds.center - shape.bounds.center)

            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.shape(shape)

            drawer.stroke = ColorRGBa.BLACK
            drawer.strokeWeight = 2.0
            drawer.lineSegments(normals)
        }
    }
}
```

Link to the full example

Drawing SVG

Loading a composition and drawing it can be done as follows:

```
fun main() = application {
    program {
        val composition = loadSVG("data/drawing.svg")
        extend {
            drawer.composition(composition)
        }
    }
}
```

Note that OPENRNDR's support for SVG files works best with SVG files that are saved in the Tiny SVG 1.x profile .

When drawing a composition from SVG you will notice that fill and stroke colors from the SVG file are used over the `Drawer` colors.

Compositions

Here a `Composition` is a tree structure with `Composition.root` as its root node.

CompositionNode types

Node Type	Function
`GroupNode`	Holds multiple child nodes in `children`
`ShapeNode`	Holds a a single shape in `shape`
`TextNode`	Holds text (currently not implemented)
`ImageNode`	Holds an image (currently not implemented)
`CompositionNode`	The base class for composition node

CompositionNode properties

Property name	Property type	Description
`transform`	`Matrix44`	local transformation
`fill`	`CompositionColor`	fill color
`stroke`	`CompositionColor`	stroke color
`id`	`String?`	node id
`parent`	`CompositionNode?`	node parent
`effectiveFill` (read-only)	`ColorRGBa?`	the effective fill color, potentially inherited from ancestor
`effectiveStroke`(read-only)	`ColorRGBa?`	the effective stroke color, potentially inherited from ancestor

GroupNode properties

Property name	Property type	Description
children	MutableList<CompositionNode>	child nodes

ShapeNode properties

Property name	Property type	Description
shape	Shape	a single shape

Querying the composition

Finding all shapes in the composition

```
val shapeNodes = composition.findShapes()
```

Modifying the composition

Since a `Composition` contains many immutable parts it is easier to (partially) replace parts of the composition.

```
val m = translate(1.0, 0.0, 0.0);
composition.root.map {
  if (it is ShapeNode) {
    it.copy(shape=it.shape.transform(m))
  } else {
    it
  }
}
```

Creating compositions manually

Compositions are tree structures that can be constructed manually. Below you find an example of constructing a `Composition` in code.

```
val root = GroupNode()
val composition = Composition(root)
val shape = Circle(200.0, 200.0, 100.0).shape
val shapeNode = ShapeNode(shape)
shapeNode.fill = ColorRGBa.PINK
shapeNode.stroke = ColorRGBa.BLACK
// -- add shape node to root
root.children.add(shapeNode)
```

Composition Drawer

OPENRNDR has a much more convenient interface for creating Compositions. The idea behind this interface is that it works in a similar way to `Drawer`.

Below we use `drawComposition {}` to reproduce the same composition as in the previous example.

```
val composition = drawComposition {
    fill = ColorRGBa.PINK
```

```
    stroke = ColorRGBa.BLACK
    circle(Vector2(100.0, 100.0), 50.0)
}
```

Transforms

Transforms work in the same way as in `Drawer`

```
val composition = drawComposition {
    fill = ColorRGBa.PINK
    stroke = ColorRGBa.BLACK
    isolated {
        for (i in 0 until 100) {
            circle(Vector2(0.0, 0.0), 50.0)
            translate(50.0, 50.0)
        }
    }
}
```

Saving compositions to SVG

Compositions can be saved to SVG using the `saveToFile` function.

```
// -- load in a composition
val composition = loadSVG("data/example.svg")
composition.saveToFile(File("output.svg"))
```

Playing videos

OPENRNDR comes with FFMPEG-backed video support.

A simple video player

Relevant APIs

- VideoPlayerFFMPEG.fromFile
- VideoPlayerFFMPEG.play
- VideoPlayerFFMPEG.draw

```
fun main() = application {
    program {
        val videoPlayer = VideoPlayerFFMPEG.fromFile("data/video.mp4")
        videoPlayer.play()
        extend {
            drawer.clear(ColorRGBa.BLACK)
            videoPlayer.draw(drawer)
        }
    }
}
```

Looping a video

```
// Loop: restart when reaching the end
videoPlayer.ended.listen {
    videoPlayer.restart()
}

extend {
    videoPlayer.draw(drawer)
}
```

Video from camera devices

The `VideoPlayerFFMPEG` class can be used to get and display video data from camera devices. To open a camera device you use the `fromDevice()` method. When this method is called without any arguments it attempts to open the default camera device.

`VideoPlayerFFMPEG` has minimal device listing capabilities. The device names of available input devices can be listed using VideoPlayerFFMPEG.listDeviceNames.

Relevant APIs

- VideoPlayerFFMPEG.fromDevice
- VideoPlayerFFMPEG.defaultDevice

Examples

```
fun main() = application {
    program {
        val videoPlayer = VideoPlayerFFMPEG.fromDevice()
        videoPlayer.play()
        extend {
            drawer.clear(ColorRGBa.BLACK)
            videoPlayer.draw(drawer)
        }
    }
}
```

Processing video

Video playback can be combined with render targets and filters.

In the following example the video player output is blurred before presenting it on the screen.

```
fun main() = application {
    program {
        val videoPlayer = VideoPlayerFFMPEG.fromFile("data/video.mp4")
        val blur = BoxBlur()
        val renderTarget = renderTarget(width, height) {
            colorBuffer()
        }
        videoPlayer.play()

        extend {
            drawer.clear(ColorRGBa.BLACK)
            // -- draw the video on the render target
            drawer.withTarget(renderTarget) {
                videoPlayer.draw(drawer)
            }
            // -- apply a blur on the render target's color attachment
            blur.apply(renderTarget.colorBuffer(0), renderTarget.colorBuffer(0))
            // -- draw the blurred color attachment
            drawer.image(renderTarget.colorBuffer(0))
        }
    }
}
```

3D graphics

Draw a rotating 3D box with a minimal `shadeStyle` to simulate lighting. Lower level approach.

```
fun main() = application {
    configure {
        multisample = WindowMultisample.SampleCount(4)
    }
    program {
        val cube = boxMesh(140.0, 70.0, 10.0)

        extend {
            drawer.perspective(60.0, width * 1.0 / height, 0.01, 1000.0)
            drawer.depthWrite = true
            drawer.depthTestPass = DepthTestPass.LESS_OR_EQUAL

            drawer.fill = ColorRGBa.PINK
            drawer.shadeStyle = shadeStyle {
                fragmentTransform = """
                        vec3 lightDir = normalize(vec3(0.3, 1.0, 0.5));
                        float l = dot(va_normal, lightDir) * 0.4 + 0.5;
                        x_fill.rgb *= l;
                    """.trimIndent()
            }
            drawer.translate(0.0, 0.0, -150.0)
            drawer.rotate(Vector3.UNIT_X, seconds * 15 + 30)
            drawer.rotate(Vector3.UNIT_Y, seconds * 5 + 60)
            drawer.vertexBuffer(cube, DrawPrimitive.TRIANGLES)
        }
    }
}
```

Draw a rotating 3D box with a minimal `shadeStyle` to simulate lighting. Uses `Orbital` to simplify creating a 3D camera which can be controlled with the mouse and the keyboard.

```
fun main() = application {
    configure {
        multisample = WindowMultisample.SampleCount(4)
    }
    program {
        val cube = boxMesh(140.0, 70.0, 10.0)
        val cam = Orbital()
        cam.eye = -Vector3.UNIT_Z * 150.0

        extend(cam)
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.shadeStyle = shadeStyle {
```

```
                fragmentTransform = """
                        vec3 lightDir = normalize(vec3(0.3, 1.0, 0.5));
                        float l = dot(va_normal, lightDir) * 0.4 + 0.5;
                        x_fill.rgb *= l;
                    """.trimIndent()
            }
            drawer.vertexBuffer(cube, DrawPrimitive.TRIANGLES)
        }
    }
}
```

Draw ten 2D rectangles in 3D space.

```
fun main() = application {
    configure {
        multisample = WindowMultisample.SampleCount(4)
    }
    program {
        val cam = Orbital()
        cam.eye = Vector3.UNIT_Z * 150.0
        cam.camera.depthTest = false

        extend(cam)
        extend {
            drawer.fill = null
            drawer.stroke = ColorRGBa.PINK
            repeat(10) {
                drawer.rectangle(Rectangle.fromCenter(Vector2.ZERO, 150.0))
                drawer.translate(Vector3.UNIT_Z * 10.0)
            }
        }
    }
}
```

2D drawing operations like `drawer.rectangle`, `drawer.contour`, etc. can have depth related occlusion issues, as they are not designed for 3D usage. To avoid such issues you can create your own vertex buffers and meshes.

See also

- orx-camera
- orx-dnk3
- orx-mesh-generators

Drawing primitives batched

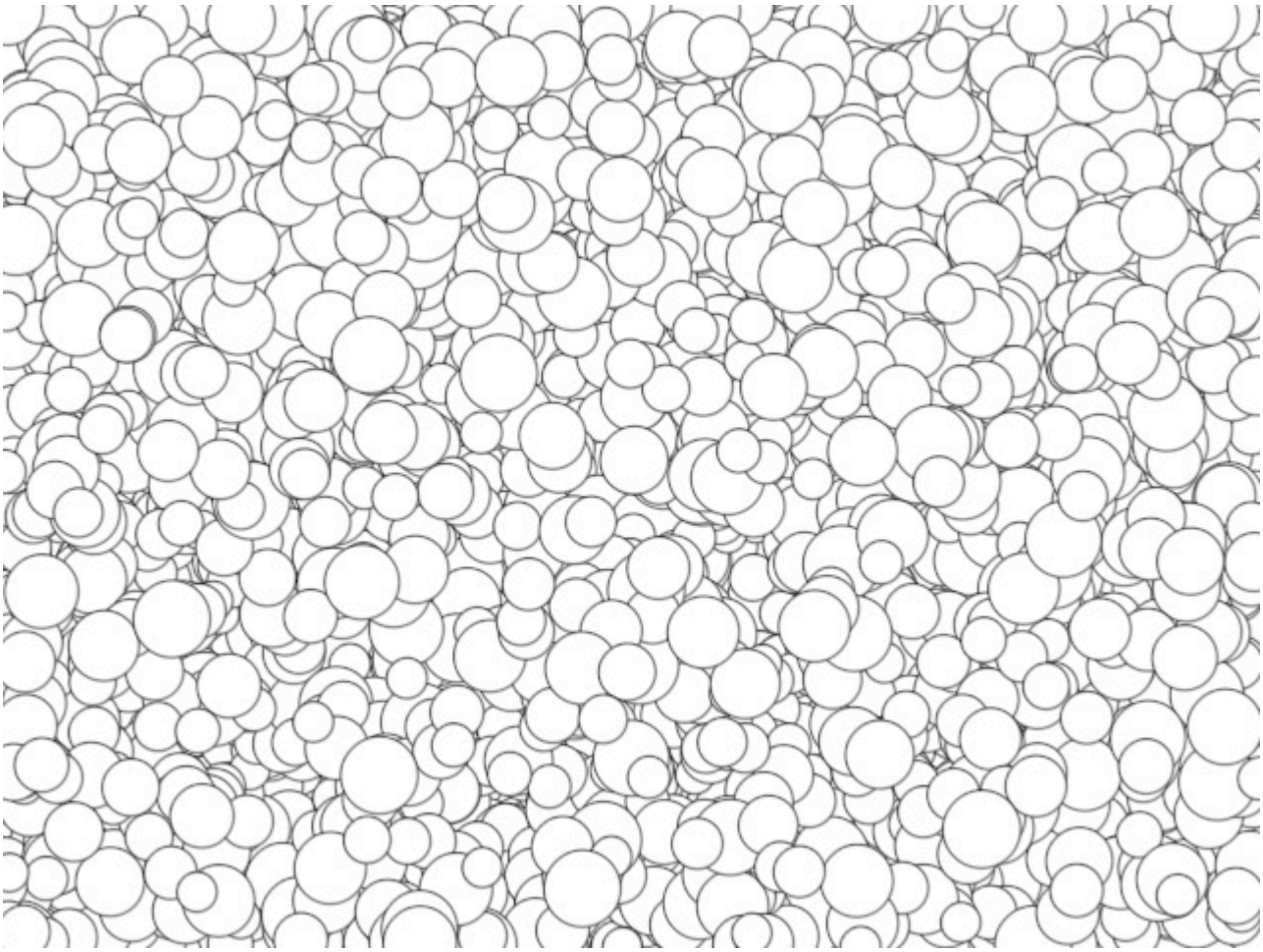
OPENRNDR provides special draw APIs for drawing many circles, rectangles or points at once by doing something called “batching”.

This technique performs much faster because the rendering time is tied to the number of calls we do on the GPU. It takes less time to send one large piece of data from CPU to GPU than sending many small pieces. Using this approach can be beneficial when drawing hundreds or thousands of elements.

Batched circles

This example makes use of `Circle` (a class with properties like `center` and `radius` and some useful methods), not to be confused with `drawer.circle()` (a method that draws pixels on the screen). It is possible to construct a `Circle` by providing a `center` position and a `radius` but also with two or three `Vector2` points that are used for deriving the circumference of a `Circle`.

Calling `drawer.circles()` to draw a list of `Circle` is much faster than calling `drawer.circle()` multiple times.



```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.PINK)
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = ColorRGBa.BLACK
```

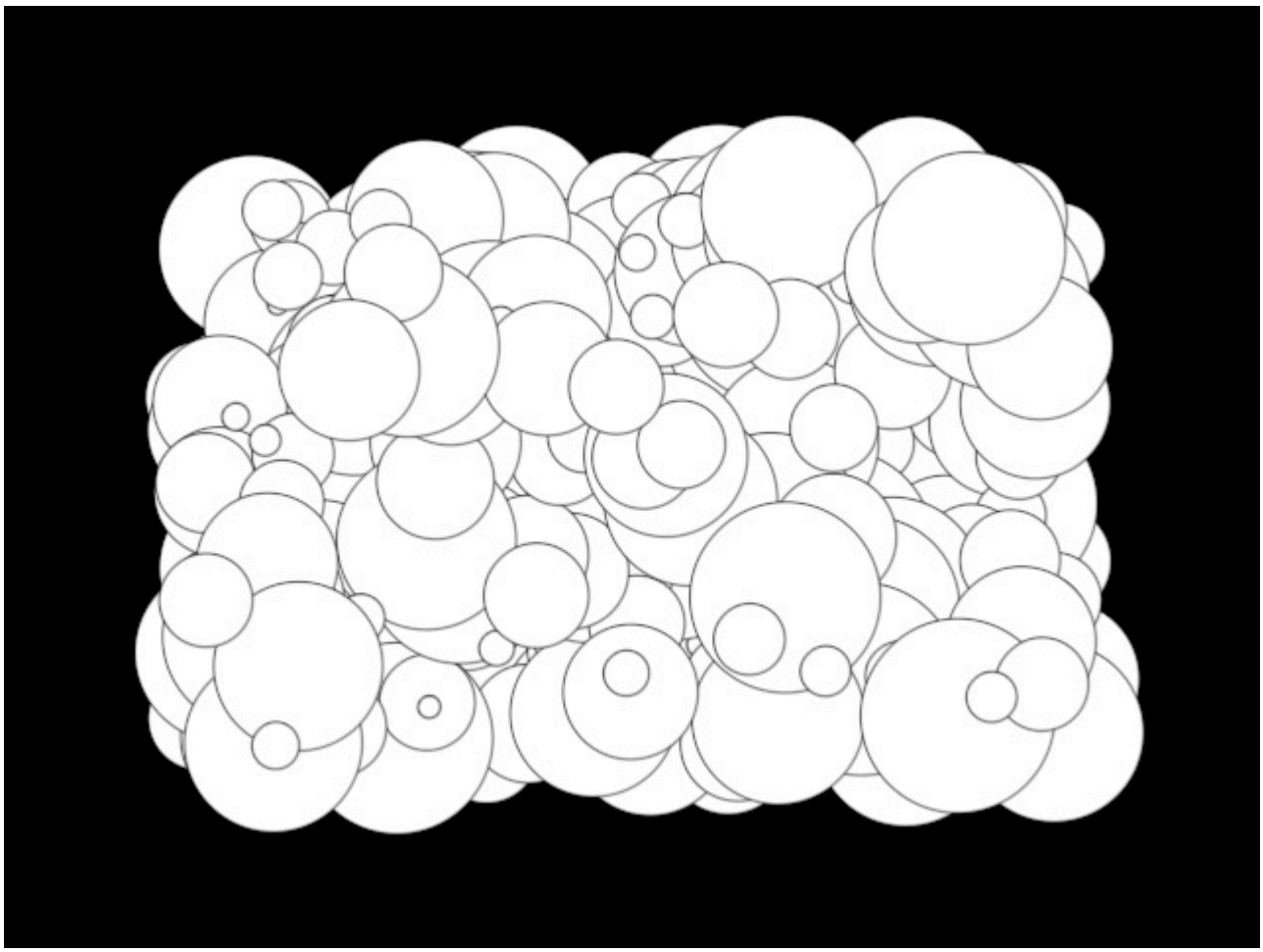
```
            drawer.strokeWeight = 1.0

            val circles = List(50000) {
                Circle(Math.random() * width, Math.random() * height, Math.random() * 10.0 + 10.0)
            }
            drawer.circles(circles)
        }
    }
}
```

Link to the full example

`drawer.circles` has several signatures. One of them accepts a list of `Vector2` for the circle centers and a `Double` to specify the radius for all circles. This example draws 5000 circles on the screen leaving a 100.0 pixel margin around the edges.

```
val area = drawer.bounds.offsetEdges(-100.0)
val positions = List(5000) {
    Random.point(area)
}
drawer.circles(positions, 20.0)
```

To have a unique radius per circle we can provide a list of Double as a second argument:



```
fun main() = application {
    program {
        extend {
```

```
            val area = drawer.bounds.offsetEdges(-100.0)
            val positions = List(400) {
                Random.point(area)
            }
            val radii = List(400) {
                Random.double(5.0, 50.0)
            }
            drawer.circles(positions, radii)
        }
    }
}
```

Link to the full example

What about unique colors and `strokeWeight`s per circle? Creating static or dynamic batches makes it possible, both shown in the next example.

```
fun main() = application {
    program {
        val staticBatch = drawer.circleBatch {
            for (i in 0 until 2000) {
                fill = ColorRGBa.GRAY.shade(Math.random())
                stroke = ColorRGBa.WHITE.shade(Math.random())
                strokeWeight = 1 + Math.random() * 5
                val pos = Random.ring2d(100.0, 200.0) as Vector2
                circle(pos + drawer.bounds.center, 5 + Math.random() * 20)
            }
```

```
        }

        extend {
            drawer.clear(ColorRGBa.GRAY)
            drawer.circles(staticBatch)

            // dynamic batch
            drawer.circles {
                repeat(100) {
                    fill = ColorRGBa.PINK.shade(Math.random())
                    stroke = null
                    val pos = Vector2((it * 160.0) % width, height * 1.0)
                    val radius = Random.double(2.5, 110.0 - it) * 2
                    circle(pos, radius)
                }
            }
        }
    }
}
```

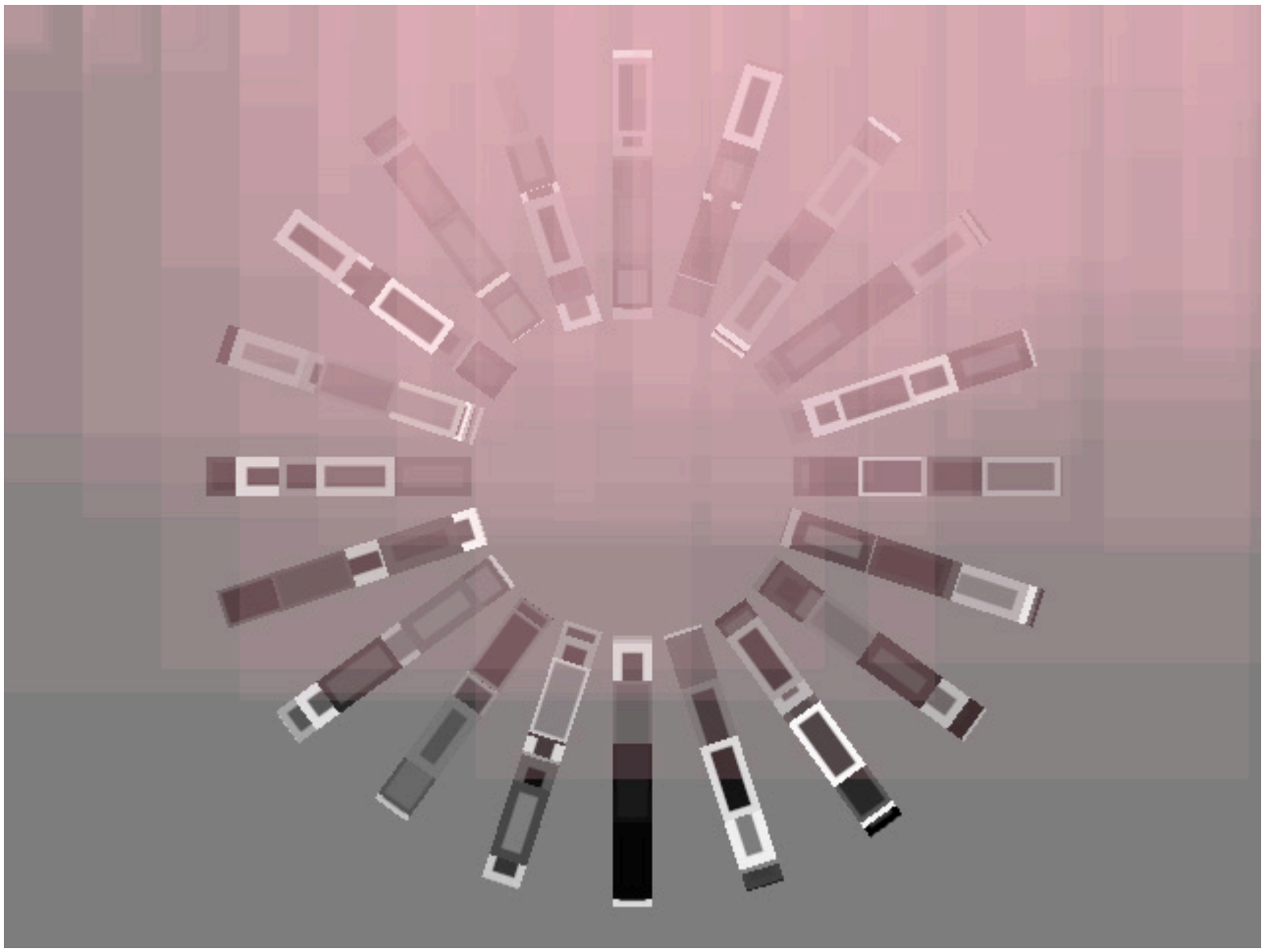
Link to the full example

Batched rectangles

This example calls the `.rectangle()` method of `RectangleBatchBuilder` multiple times to construct a batch. It does so in two different ways: first, to construct a static batch of monochrome rectangles and second, to construct a dynamic batch of translucent pink rectangles in each animation frame.

The `.rectangle()` method takes two arguments: a `Rectangle` object and an optional rotation. `Rectangle`s can be constructed in different ways: with position, width and optional height or by using `Rectangle.fromCenter()`.

Calling `drawer.rectangles()` to draw a rectangle batch is much faster than calling `drawer.rectangle()` multiple times.



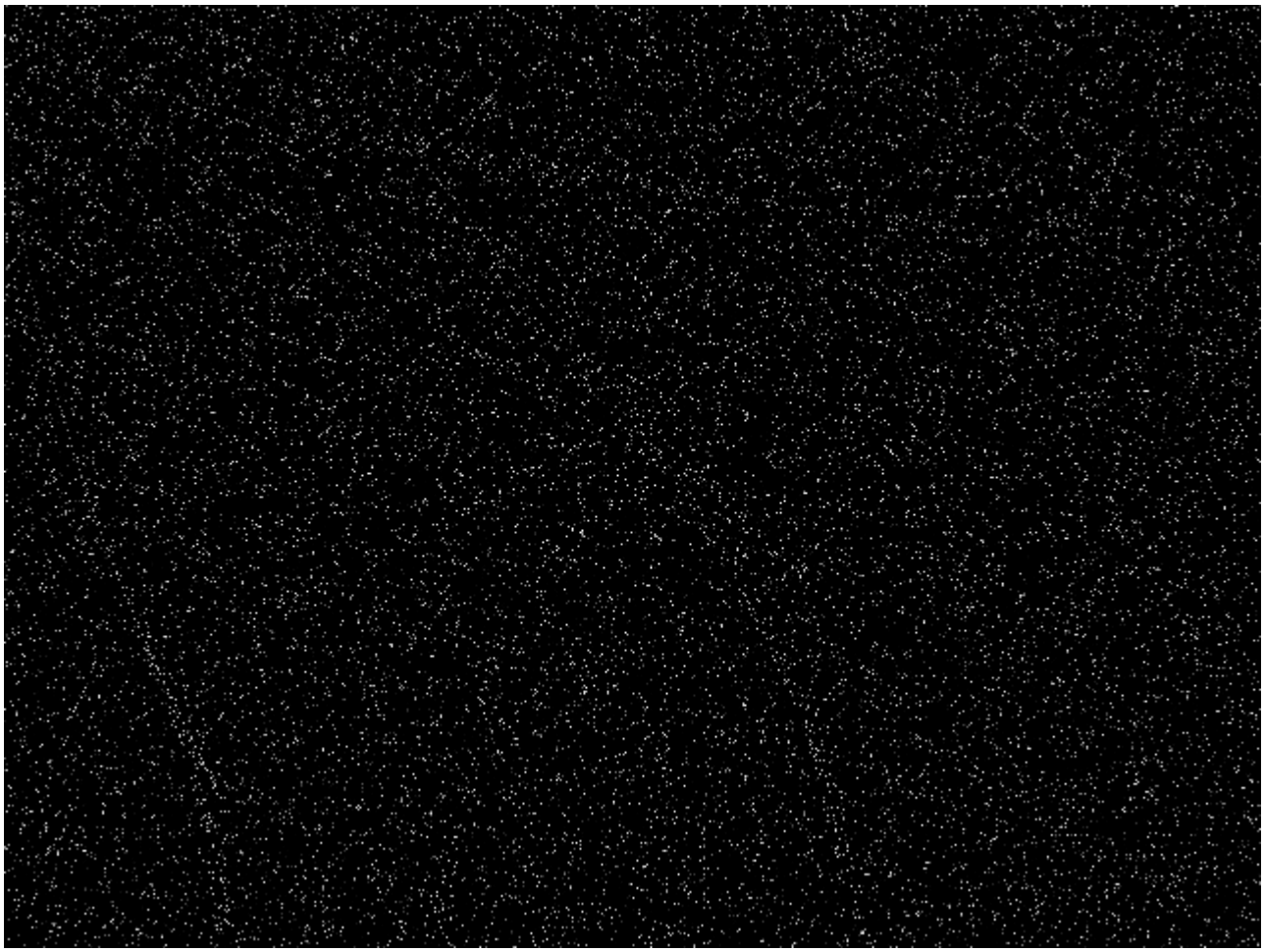
```
fun main() = application {
    program {
        val staticBatch = drawer.rectangleBatch {
            for (i in 0 until 1000) {
                fill = ColorRGBa.GRAY.shade(Math.random())
                stroke = ColorRGBa.WHITE.shade(Math.random())
                strokeWeight = Random.double(1.0, 5.0)
                val angle = Random.int0(20) * 18.0
                val pos = drawer.bounds.center + Polar(angle, Random.double(100.0, 200.0)).cartesian
                val rect = Rectangle.fromCenter(pos, width = 40.0, height = 20.0)
                rectangle(rect, angle) // add rect to the batch
            }
        }

        extend {
            drawer.clear(ColorRGBa.GRAY)
            drawer.rectangles(staticBatch)

            // dynamic batch
            drawer.rectangles {
                repeat(100) {
                    fill = ColorRGBa.PINK.opacify(0.05)
                    stroke = null
                    val pos = Vector2((it * 200.0) % width, 0.0)
                    val size = 5 + Math.random() * Math.random() * height
                    rectangle(Rectangle(pos, size))
                }
```

```
            }
        }
    }
}
```

Link to the full example

Batched points

Drawing batched points is similar to drawing batched circles and rectangles. In this case we use a `PointBatchBuilder` and its `.point()` method. Note that we can specify the color of each point by using `.fill`.



```
fun main() = application {
    program {
        extend {
            drawer.clear(ColorRGBa.BLACK)
            drawer.points {
                repeat(20000) {
                    fill = rgb((it * 0.01 - seconds) % 1)
                    point((it * it * 0.011) % width, (it * 4.011) % height)
                }
            }
        }
    }
}
```

Link to the full example

Transformations

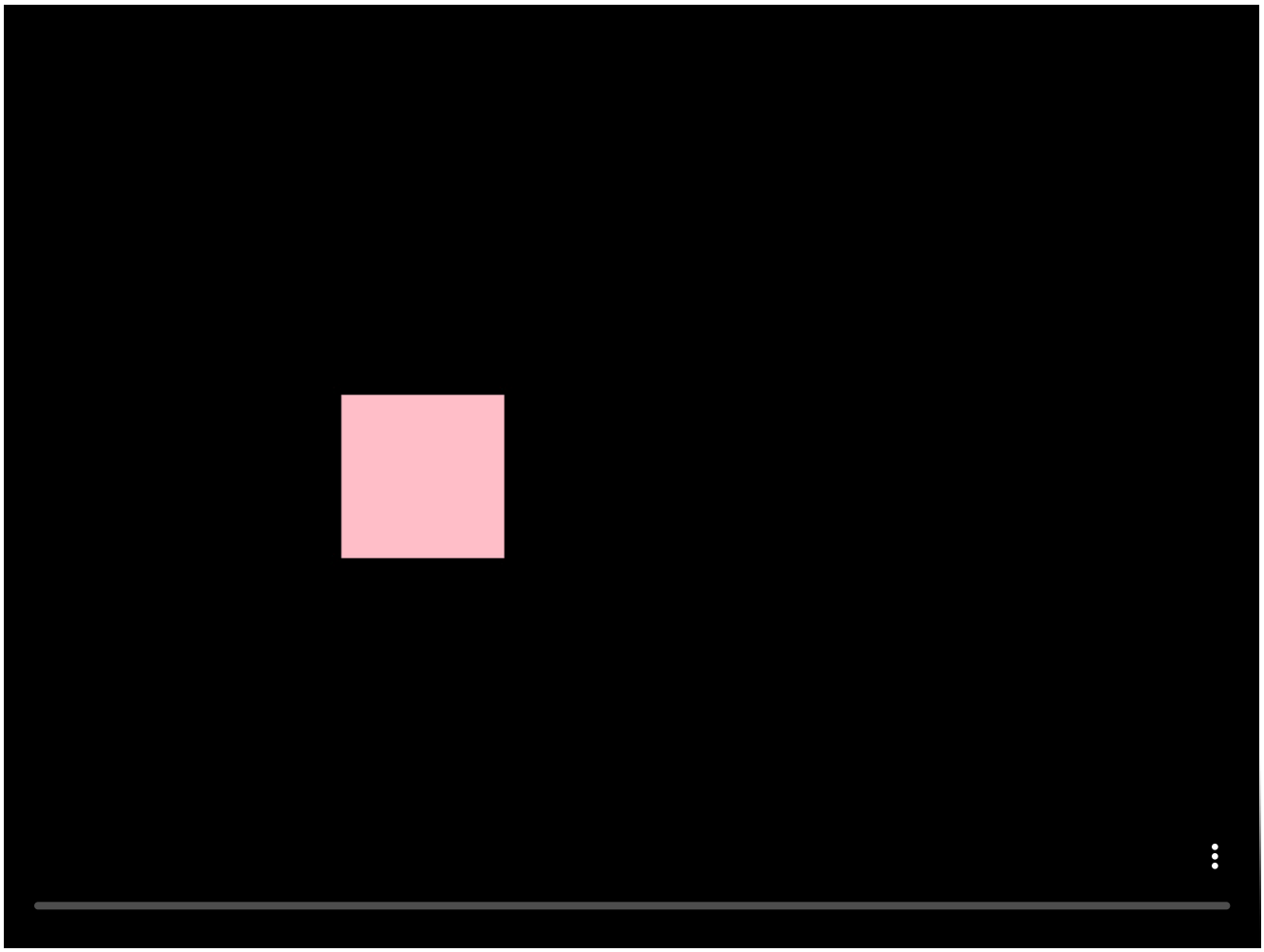
This section covers the topic of placing items on the screen.

Basic transformation use

Translation

Translation moves points in space with an offset.

In the following example we use `Drawer.translate` to move a single rectangle over the screen.

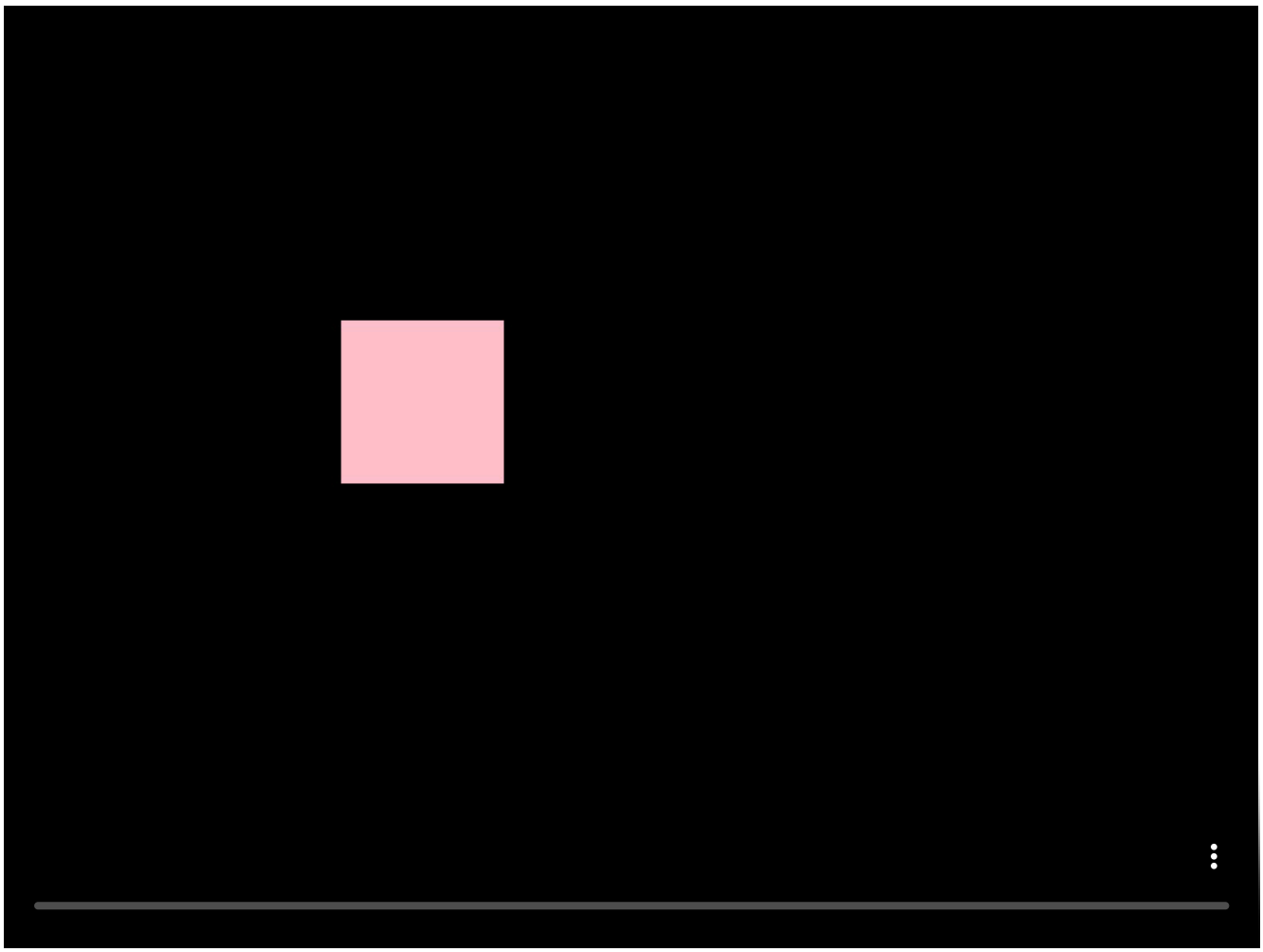


```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            // set up translation
            drawer.translate(seconds * 100.0, height / 2.0)
```

```
            drawer.rectangle(-50.0, -50.0, 100.0, 100.0)
        }
    }
}
```

Link to the full example

translations (and transformations in general) can be stacked on top of each-other. For example we can express a horizontal and a vertical motion as two separate translations



```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            // move the object to the vertical center of the screen
            drawer.translate(0.0, height / 2.0)
            // set up horizontal translation
            drawer.translate(seconds * 100.0, 0.0)
            // set up vertical translation
            drawer.translate(0.0, cos(seconds * Math.PI * 2.0) * 50.00)

            drawer.rectangle(-50.0, -50.0, 100.0, 100.00)
```

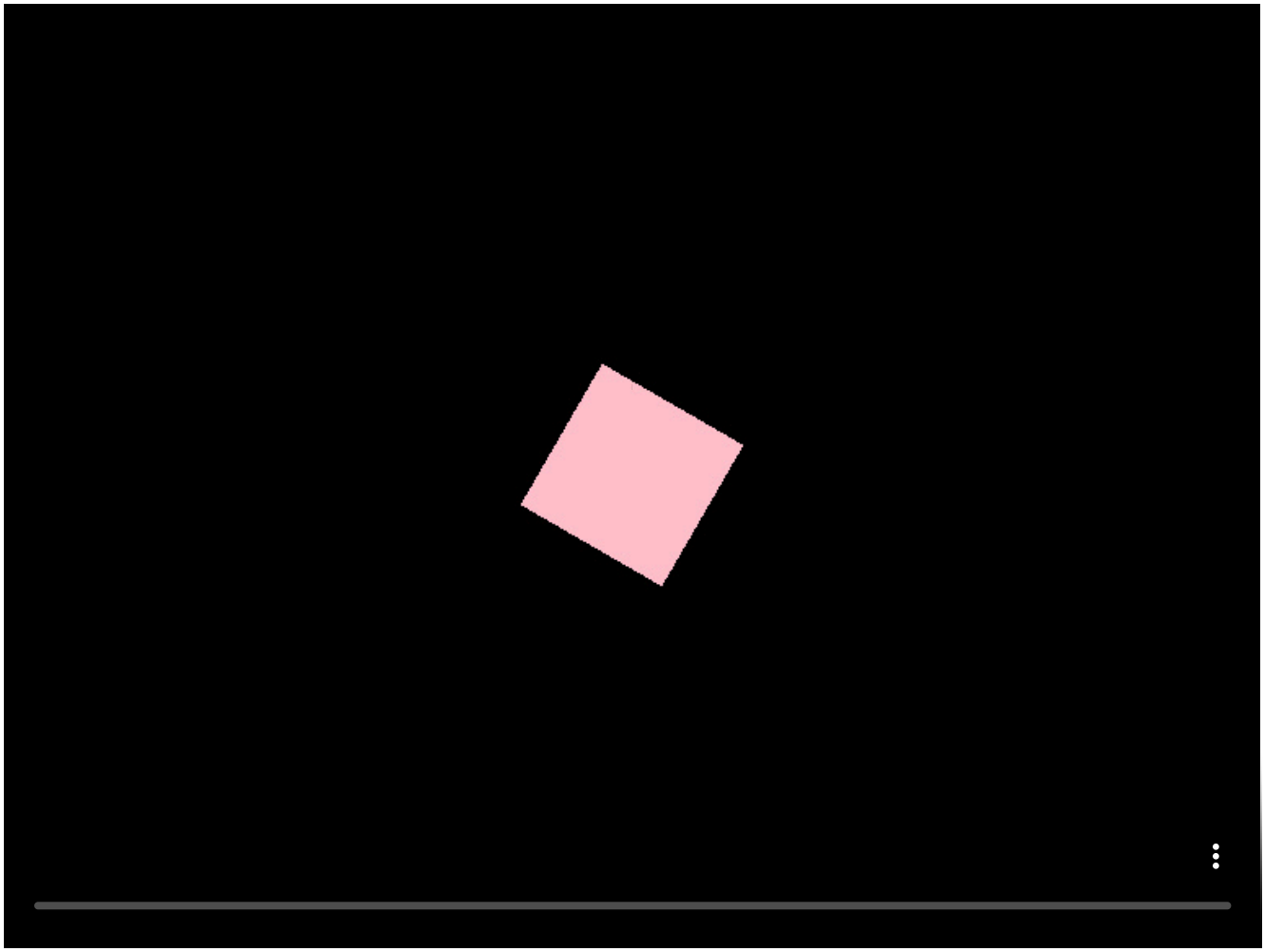
```
        }
    }
}
```

Link to the full example

Rotations

Rotating transformations are performed using `Drawer.rotate()`. The rotation is applied by rotating points around the origin of the coordinate system: (0, 0), which lies in the top-left corner of the window.

In the first rotation example we rotate a rectangle that is placed around the origin but later translated to the center of the screen. Here we notice something that may be counter-intuitive at first: the transformations are easiest read from bottom to top: first `rotate` is applied and only then `translate`.



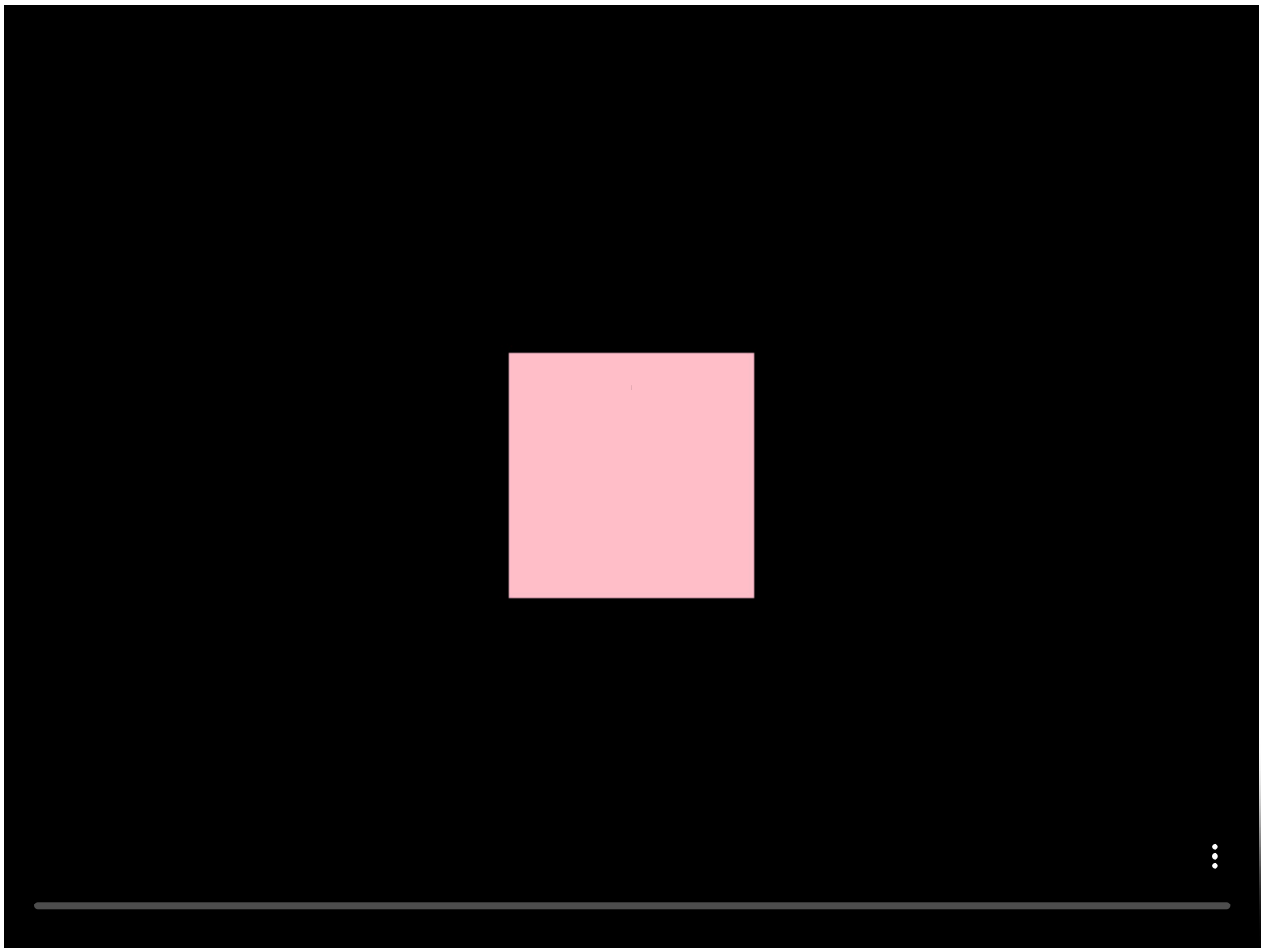
```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null

            // -- translate
            drawer.translate(width / 2.0, height / 2.0)
            // -- rotate
```

```
            drawer.rotate(seconds * 30.0)
            // -- rectangle around the origin
            drawer.rectangle(-50.0, -50.0, 100.0, 100.0)
        }
    }
}
```

Link to the full example

Scaling

Scaling transformations are performed using `Drawer.scale()`. Also scaling is applied around the origin of the coordinate system: (0, 0).

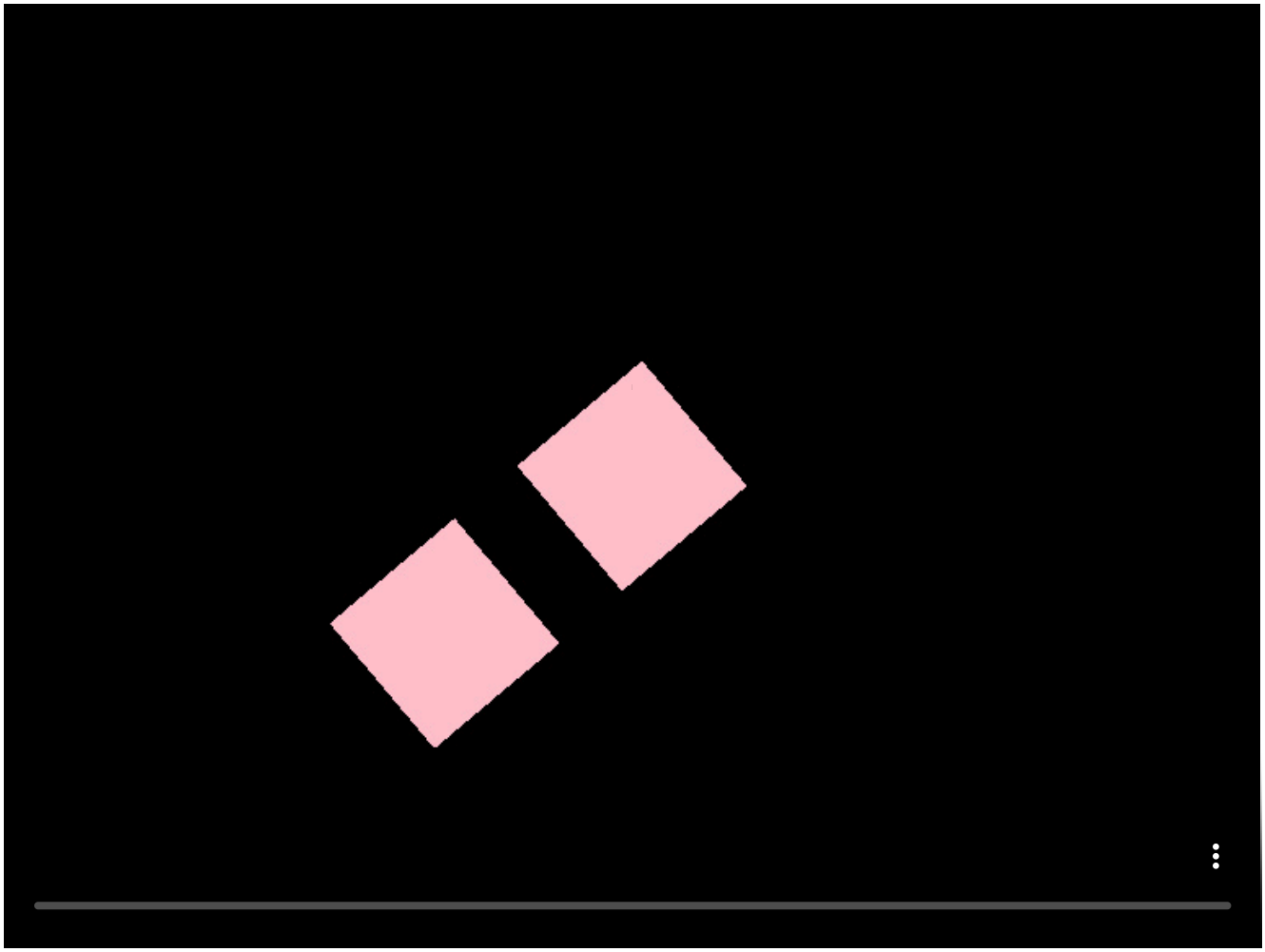


```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null

            // -- translate to the screen center
            drawer.translate(width / 2.0, height / 2.0)
            // -- scale around origin
```

```
            drawer.scale(cos(seconds) + 2.0)
            // -- rectangle around the origin
            drawer.rectangle(-50.0, -50.0, 100.0, 100.00)
        }
    }
}
```

Link to the full example

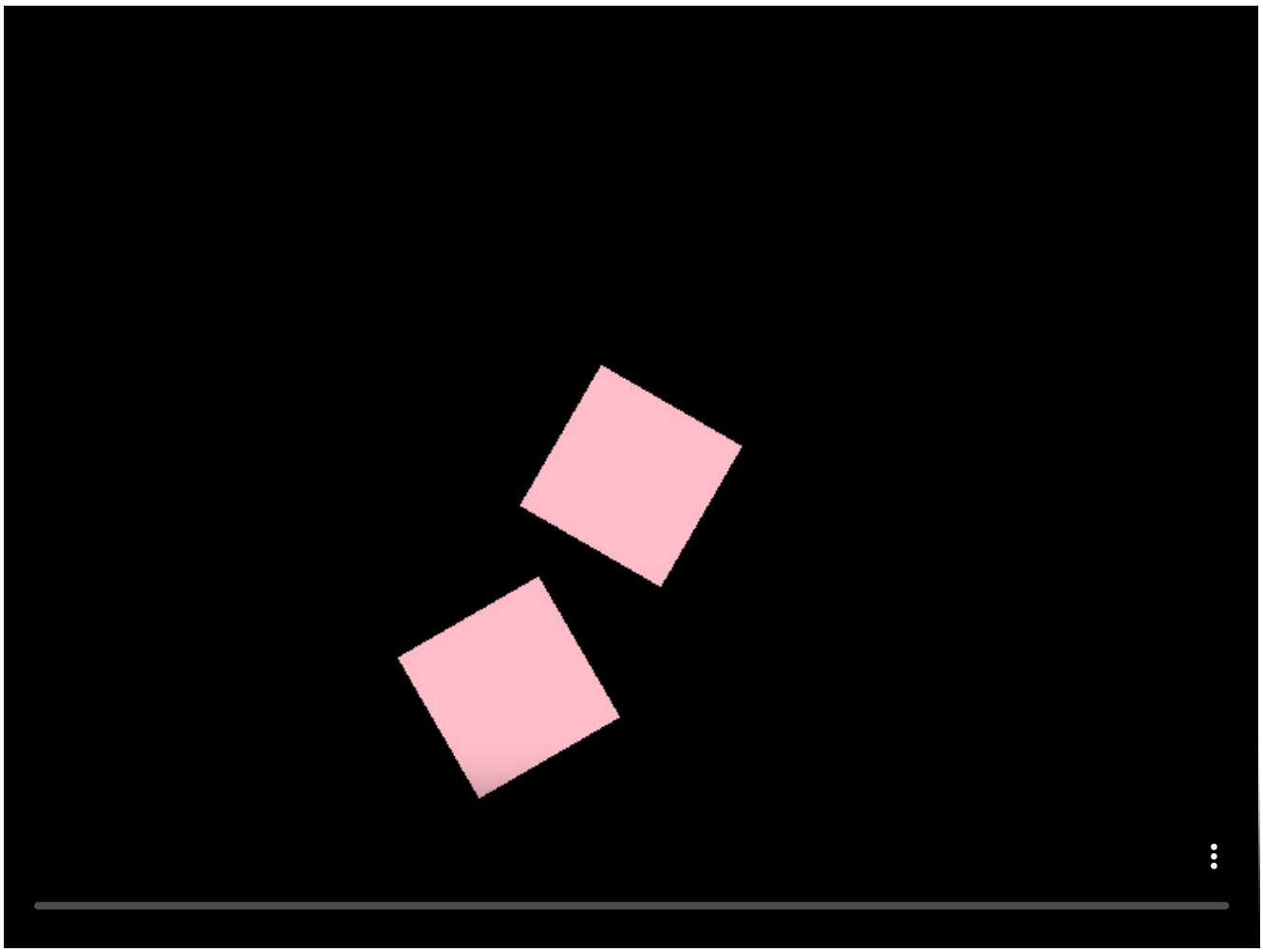
Combining transformations



```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null

            // -- translate to the screen center
            drawer.translate(width / 2.0, height / 2.0)

            drawer.rotate(20.00 + seconds * 60.0)
            // -- rectangle around the origin
            drawer.rectangle(-50.0, -50.0, 100.0, 100.0)
```

```
            // -- draw a second rectangle, sharing the rotation of the first rectangle but with an offset
            drawer.translate(150.0, 0.0)
            drawer.rectangle(-50.0, -50.0, 100.0, 100.0)
        }
    }
}
```

Link to the full example



```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null

            // -- translate to the screen center
            drawer.translate(width / 2.0, height / 2.0)

            drawer.rotate(seconds * 60.0)
            // -- rectangle around the origin
            drawer.rectangle(-50.0, -50.0, 100.0, 100.0)
```

```
            // -- draw a second rectangle, sharing the rotation of the first rectangle but with an offset
            drawer.translate(150.0, 0.0)
            drawer.rotate(seconds * 15.0)
            drawer.rectangle(-50.0, -50.0, 100.0, 100.00)
        }
    }
}
```

Link to the full example

Transform pipeline

OPENRNDR's `Drawer` is build around model-view-projection transform pipeline. That means that three different transformations are applied to determine the screen position.

matrix property	pipeline stage
`model`	model transform
`view`	view transform
`projection`	projection transform

Which matrices are affected by which `Drawer` operations?

operation	matrix property
`fun rotate(…)`	`model`
`fun translate(…)`	`model`
`fun scale(…)`	`model`
`fun lookAt(…)`	`view`
`fun ortho(…)`	`projection`
`fun perspective(…)`	`projection`

Projection matrix

The default projection transformation is set to an orthographic projection using `ortho()`. The origin is in the upper-left corner; positive y points down, positive x points right on the screen.

Perspective projections

```
override fun draw() {
    drawer.perspective(90.0, width*1.0 / height, 0.1, 100.0)
}
```

Transforms

In OPENRNDR transforms are represented by `Matrix44` instances.

OPENRNDR offers tools to construct `Matrix44`

Transform builder

Relevant APIs

```
Matrix44
transform {}
```

In the snippet below a `Matrix44` instance is constructed using the `transform {}` builder. Note that the application order is from bottom to top.

```
drawer.model *= transform {
    rotate(32.0)
    rotate(Vector3(1.0, 1.0, 0.0).normalized, 43.0)
    translate(4.0, 2.0)
    scale(2.0)
}
```

This is equivalent to the following:

```
drawer.rotate(32.0)
drawer.rotate(Vector3(1.0, 1.0, 0.0).normalized, 43.0)
drawer.translate(4.0, 2.0)
drawer.scale(2.0)
```

Applying transforms to vectors

```
    val x = Vector4(1.0, 2.0, 3.0, 1.0)
    val m = transform {
        rotate(Vector3.UNIT_Y, 42.0)
    }
    val transformed = m * x
    val transformedTwice = m * m * x
```

Vectors

The `Vector2`, `Vector3` and `Vector4` classes are used for 2, 3 and 4 dimensional vector representations. Vector instances are immutable; once a Vector has been instantiated its values cannot be changed.

```
val v2 = Vector2(1.0, 10.0)
val v3 = Vector3(1.0, 1.0, 1.0)
val v3 = Vector4(1.0, 1.0, 1.0, 1.0)
```

Standard vectors

```
Vector2.ZERO    // (0, 0)
Vector2.UNIT_X  // (1, 0)
Vector2.UNIT_Y  // (0, 1)

Vector3.ZERO    // (0, 0, 0)
Vector3.UNIT_X  // (1, 0, 0)
Vector3.UNIT_Y  // (0, 1, 0)
Vector3.UNIT_Z  // (0, 0, 1)

Vector4.ZERO    // (0, 0, 0, 0)
Vector4.UNIT_X  // (1, 0, 0, 0)
Vector4.UNIT_Y  // (0, 1, 0, 0)
Vector4.UNIT_Z  // (0, 0, 1, 0)
Vector4.UNIT_W  // (0, 0, 0, 1)
```

Vector arithmetic

The vector classes have operator overloads for the most essential operations.

left operand	operator	right operand	result
`VectorN`	`+`	`VectorN`	addition of two vectors
`VectorN`	`-`	`VectorN`	subtraction of two vectors
`VectorN`	`/`	`Double`	scaled vector
`VectorN`	`*`	`Double`	scaled vector
`VectorN`	`*`	`VectorN`	component-wise multiplication (l.x * r.x, l.y * r.y)
`VectorN`	`/`	`VectorN`	component-wise division (l.x / r.x, l.y / r.y)

Some examples of vector arithmetic in practice

```
val a = Vector2(2.0, 4.0)
val b = Vector2(1.0, 3.0)
val sum = a + b
val diff = a - b
```

```
val scale = a * 2.0
val div = a / 2.0
val cwdiv = a / b
```

Vector properties

property	description
`length`	the length of the vector
`normalized`	a normalized version of the vector

Swizzling and sizing

Vector2 swizzles allow reordering of vector fields, this is a common pattern in GLSL

```
val v3 = Vector2(1.0, 2.0).vector3(z=0.0)
val v2 = Vector3(1.0, 2.0, 3.0).xy
```

Let/copy pattern

Here we present two patterns that make working with immutable Vector classes a bit more convenient.

The copy pattern (which comes from Vectors being Kotlin data classes)

```
val v = Vector2(1.0, 2.0)
val w = v.copy(y=5.0)      // (1.0, 5.0)
```

The let/copy pattern, which combines Kotlin's `let` with `copy`

```
val v = someFunctionReturningAVector().let { it.copy(x=it.x + it.y) }
```

Mixing

Linear interpolation of vectors using `mix()`

```
val m = mix(v0, v1, f)
```

which is short-hand for

```
val m = v0 * (1.0 - f) + v1 * f
```

Randomness

Generating random vectors

```
val v2 = Random.vector2(-1.0, 1.0)
val v3 = Random.vector3(-1.0, 1.0)
val v4 = Random.vector4(-1.0, 1.0)
```

To generate random distributions of vectors see orx-noise.

Quaternions

Quaternions represent rotation through an extension of complex numbers. A full explanation of quaternions and their intrinsics is out of this document's scope, in this section however enough information is provided to use quaternion's effectively as a tool.

In practice quaternions are rarely constructed directly as it is fairly difficult to get an intuition for its argument values.

```
val q = Quaternion(0.4, 0.3, 0.1, 0.1)
```

Instead quaternions are created from Euler-rotation angles and concatenated in quaternion space. Working in quaternion domains warrants consistent rotations and avoids gimbal locks.

```
val q0 = fromAngles(pitch0, yaw0, roll0)
val q1 = fromAngles(pitch1, yaw1, roll1)
val q = q0 * q1
```

Slerp

Spherical linear interpolation, or colloquially "slerping" solves the problem of interpolating or blending between rotations.

```
val q0 = fromAngles(pitch0, yaw0, roll0)
val q1 = fromAngles(pitch1, yaw1, roll1)
val q = slerp(q0, q1, 0.5)
```

Quaternion to matrix

Naturally quaternions can be converted to matrices. Quaternions have a `matrix` property that holds a `Matrix44` representation of the orientation represented by the Quaternion.

```
val q0 = fromAngles(pitch, yaw, roll)
drawer.model *= q0.matrix.matrix44
```

Color buffers

A color buffer is an image stored in GPU memory.

Creating a color buffer

Color buffers are created using the `colorBuffer()` function.

```
val cb = colorBuffer(640, 480)
```

Specifying buffer format

Color buffers can be created in different formats. The buffer format specifies the number and order of channels in the image. Color buffers can have 1 to 4 channels. The `format` argument can be any `ColorFormat` value.

```
val cb = colorBuffer(640, 480, format = ColorFormat.R)
```

Specifying buffer type

The buffer type specifies which data type is used for storing colors in the buffer. The `type` argument can be any `ColorType` value.

```
val cb = colorBuffer(640, 480, type = ColorType.FLOAT16)
```

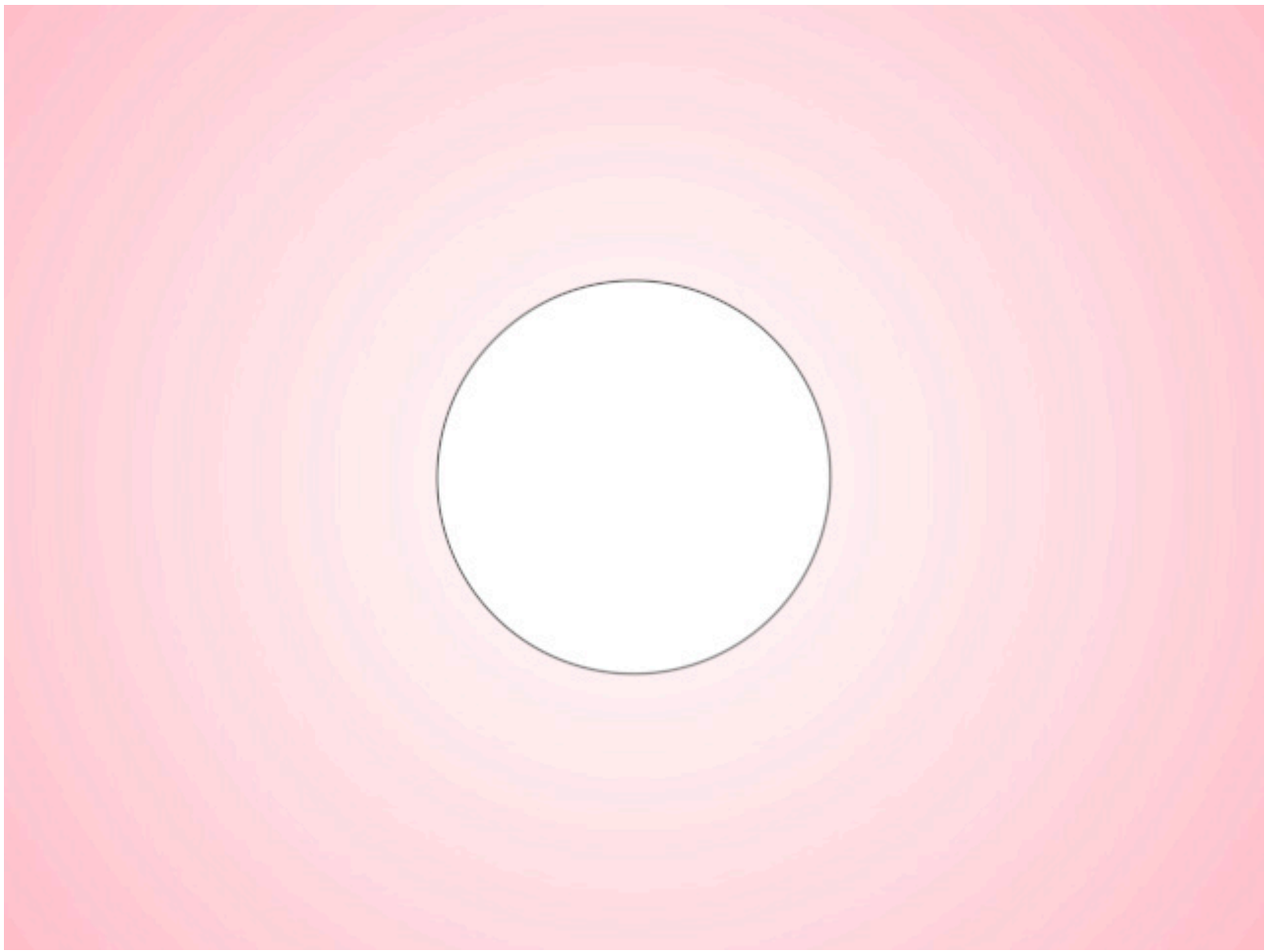
Loading color buffers

Color buffers can be loaded from an image stored on disk. Supported file types are png, jpg, dds and exr (OpenEXR).

```
val cb = loadImage("data/images/pm5544.jpg")
```

Generating color buffers

Use `drawImage` to create a color buffer and draw into it.



```
fun main() = application {
    program {
        val gradientBackground = drawImage(width, height) {
            // Draw anything here, for example, a radial gradient.
            drawer.shadeStyle = RadialGradient(ColorRGBa.WHITE, ColorRGBa.PINK)
            val r = Rectangle.fromCenter(drawer.bounds.center, 800.0, 800.0)
            drawer.rectangle(r)
        }
        extend {
            drawer.image(gradientBackground)
            drawer.circle(drawer.bounds.center, sin(seconds) * 80 + 100)
        }
    }
}
```

Link to the full example

`drawImage` is convenient for creating static images. If a program requires redrawing a color buffer use a render target instead.

Freeing color buffers

If a program creates new buffers while it runs it is important to free those buffers when no longer needed to avoid running out of memory.

```
// -- When done using the buffer call destroy() to free its memory.
```

```
cb.destroy()
```

Saving color buffers

Color buffers can be saved to disk using the `saveToFile` member function. Supported file types are png, jpg, dds and exr (OpenEXR).

```
val cb = colorBuffer(640, 480)
cb.saveToFile(File("output.jpg"))
```

When repeatedly saving color buffers asynchronously (the default) it is possible to run out of memory. This can happen if the software can not save images at the requested frame rate. In such situations we can either set `async = false` in `saveToFile()` or avoid `saveToFile` and use the VideoWriter together with pngSequence or tiffSequence instead.

Note that some image file types take longer to save than others.

Copying between color buffers

Color buffer contents can be copied using the `copyTo` member function. Copying works between color buffers of different formats and types.

```
// -- create color buffers
val cb0 = colorBuffer(640, 480, type = ColorType.FLOAT16)
val cb1 = colorBuffer(640, 480, type = ColorType.FLOAT32)
//-- copy contents of cb0 to cb1
cb0.copyTo(cb1)
```

Writing into color buffers

To upload data into the color buffer one uses the `write` member function.

```
// -- create a color buffer that uses 8 bits per channel (the default)
val cb = colorBuffer(640, 480, type = ColorType.UINT8)

// -- create a buffer (on CPU) that matches the size and layout of the color buffer
val buffer = ByteBuffer.allocateDirect(cb.width * cb.height * cb.format.componentCount * cb.type.componentSize)

// -- fill buffer with random data
for (y in 0 until cb.height) {
    for (x in 0 until cb.width) {
        for (c in 0 until cb.format.componentCount) {
            buffer.put((Math.random() * 255).toInt().toByte())
        }
    }
}

// -- rewind the buffer, this is essential as upload will be from the position we left the buffer at
buffer.rewind()
// -- write into the color buffer
cb.write(buffer)
```

Reading from color buffers

To download data from a color buffer one uses the `read` member function.

```
// -- create a color buffer that uses 8 bits per channel (the default)
val cb = colorBuffer(640, 480, type = ColorType.UINT8)

// -- create a buffer (on CPU) that matches the size and layout of the color buffer
val buffer = ByteBuffer.allocateDirect(cb.width * cb.height * cb.format.componentCount * cb.type.componentSize)

// -- download data into buffer
cb.read(buffer)

// -- read the first UINT8 pixel's color
val r = buffer[0].toUByte().toDouble() / 255.0
val g = buffer[1].toUByte().toDouble() / 255.0
val b = buffer[2].toUByte().toDouble() / 255.0
val a = buffer[3].toUByte().toDouble() / 255.0
val c = rgb(r, g, b, a)
```

Color buffer shadows

To simplify the process of reading and writing from and to color buffers we added a shadow buffer to `ColorBuffer`. A shadow buffer offers a simple interface to access the color buffer's contents.

Note that shadow buffers have more overhead than using `read()` and `write()`.

```
// -- create a color buffer that uses 8 bits per channel (the default)
val cb = colorBuffer(640, 480, type = ColorType.UINT8)
val shadow = cb.shadow

// -- download cb's contents into shadow
shadow.download()

// -- place random data in the shadow buffer
for (y in 0 until cb.height) {
    for (x in 0 until cb.width) {
        shadow[x, y] = ColorRGBa(Math.random(), Math.random(), Math.random())
    }
}

// -- upload shadow to cb
shadow.upload()
```

Render targets and color buffers

A `RenderTarget` specifies a place to draw to. A `RenderTarget` has two kind of buffer attachments: `ColorBuffer` attachments and `DepthBuffer` attachments. At least a single `ColorBuffer` attachment is needed to be able to draw on a `RenderTarget`.

A `ColorBuffer` is a buffer that can hold up to 4 channel color. A `ColorBuffer` can hold 8 bit integer, 16 bit float or 32 bit float channels.

A `DepthBuffer` is a buffer that can hold depth and stencil values.

Creating a render target

The advised method of creating `RenderTarget` instances is to use the `renderTarget {}` builder.

```
val rt = renderTarget(640, 480) {}
```

This creates a render target, but the render target does not have attachments that can hold the actual image data. In the following snippet a render target with a single color buffer attachment is created using the builder.

```
val rt = renderTarget(640, 480) {
    colorBuffer()
}
```

Drawing on a render target

Use `drawer.isolatedWithTarget()` to draw into an off-screen buffer.

Once updated, we can either draw it onto the screen (as this example shows) or process its color buffer further using filters or custom shaders.

```
fun main() = application {
    program {
        // -- build a render target with a single color buffer attachment
        val rt = renderTarget(width, height) {
            colorBuffer()
        }

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.BLACK)
                drawer.fill = ColorRGBa.WHITE
                drawer.stroke = null
                drawer.rectangle(40.0, 40.0, 80.0, 80.0)
            }

            // draw the backing color buffer to the screen
            drawer.image(rt.colorBuffer(0))
        }
```

```
    }
}
```

Drawing contours or shapes on a render target

When drawing `ShapeContour` or `Shape` instances, both a color buffer and a depth buffer are required.

```
fun main() = application {
    program {
        // -- build a render target with color and depth buffer attachments
        val rt = renderTarget(width, height) {
            colorBuffer()
            depthBuffer() // <--
        }

        drawer.isolatedWithTarget(rt) {
            drawer.clear(ColorRGBa.PINK)

            // A closed contour with 12 random points defining 12 straight segments
            drawer.contour(ShapeContour.fromPoints(List(12) {
                drawer.bounds.uniform(50.0)
            }, true))
        }

        extend {
            drawer.image(rt.colorBuffer(0))
        }
    }
}
```

If we forget to include a depth buffer we will be reminded with the following error message: `drawing org.openrndr.shape.contours requires a render target with a stencil attachment`

Render targets and projection transformations

Note that the projection matrix has to fit the render target. This becomes obvious **if the dimensions of the window and the dimensions of the render target differ**. In case of orthographic (2D) projections one can call `ortho(rt)`:

```
fun main() = application {
    program {
        // -- build a render target with a single color buffer attachment
        val rt = renderTarget(400, 400) {
            colorBuffer()
        }

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.BLACK)

                // -- set the orthographic transform that matches with the render target
                ortho(rt)

                drawer.fill = ColorRGBa.WHITE
```

```
                drawer.stroke = null
                drawer.rectangle(40.0, 40.0, 80.0, 80.0)
            }

            // -- draw the backing color buffer to the screen
            drawer.image(rt.colorBuffer(0))
        }
    }
}
```

If we forget to do this the graphics rendered onto our render target may be displaced or have an unexpected scale.

Compositing using render targets and alpha channels

OPENRNDR allows for compositing using `RenderTargets` through the use of transparency encoded in alpha channels. The following code snippet uses two `RenderTarget` instances and clears them using `ColorRGBa.TRANSPARENT`.

```
fun main() = application {
    program {
        val rt0 = renderTarget(width, height) {
            colorBuffer()
        }
        val rt1 = renderTarget(width, height) {
            colorBuffer()
        }

        extend {
            drawer.stroke = null

            // -- bind our first render target, clear it, draw on it, unbind it
            drawer.isolatedWithTarget(rt0) {
                drawer.clear(ColorRGBa.TRANSPARENT)
                drawer.fill = ColorRGBa.WHITE
                drawer.rectangle(40.0, 40.0, 80.0, 80.0)
            }

            // -- bind our second render target, clear it, draw on it, unbind it
            drawer.isolatedWithTarget(rt1) {
                drawer.clear(ColorRGBa.TRANSPARENT)
                drawer.fill = ColorRGBa.PINK
                drawer.rectangle(140.0, 140.0, 80.0, 80.0)
            }
            // -- draw the backing color buffer to the screen
            drawer.image(rt0.colorBuffer(0))
            drawer.image(rt1.colorBuffer(0))
        }
    }
}
```

Creating high precision floating point render targets

The default color buffer format is unsigned 8 bit RGBa. There is support for floating point render targets.

```
val rt = renderTarget(640, 480) {
    colorBuffer(ColorFormat.RGBa, ColorType.FLOAT16)
    colorBuffer(ColorFormat.RGBa, ColorType.FLOAT32)
}
```

Multi-sample anti-aliasing

Render targets can be configured to use multi-sample anti-aliasing. All color and depth buffers that are added in the `renderTarget {}` builder will be created with the same multi sample configuration.

```
// -- here we create a multi-sampled render target
val rt = renderTarget(640, 480, multisample = BufferMultisample.SampleCount(8)) {
    colorBuffer(ColorFormat.RGBa, ColorType.FLOAT16)
    colorBuffer(ColorFormat.RGBa, ColorType.FLOAT32)
}
```

The color buffers that are attached to a multi-sampled render target cannot be drawn directly. In order to use the color buffer it has to be resolved first.

```
fun main() = application {
    program {
        // -- build a render target with color and depth buffer attachments
        val rt = renderTarget(width, height, multisample = BufferMultisample.SampleCount(8)) {
            colorBuffer()
            depthBuffer()
        }

        val resolved = colorBuffer(width, height)

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.BLACK)
                drawer.fill = ColorRGBa.WHITE
                drawer.stroke = null
                drawer.circle(0.0, 0.0, 400.0)
            }

            // -- resolve the render target attachment to `resolved`
            rt.colorBuffer(0).copyTo(resolved)

            // draw the backing color buffer to the screen
            drawer.image(resolved)

            // draw a second circle with no multisampling to compare
            drawer.fill = ColorRGBa.WHITE
            drawer.stroke = null
            drawer.circle(width * 1.0, height * 1.0, 400.0)
        }
    }
}
```

Depth Buffer

A depth buffer is required to be able to draw `Shape` and `ShapeContour` elements on a render target. Without a depth buffer the program will fail to run and an error message will remind you of this requirement.

When drawing 3D graphics a depth buffer is required so elements near the camera are drawn in front of elements farther away from it.

Clearing buffers

```
// clear the color buffer
rt.clearColor(0, ColorRGBa.TRANSPARENT)

// clear the depth buffer
rt.clearDepth()
```

Named attachments

```
val rt = renderTarget(640, 480) {
    colorBuffer("albedo", ColorFormat.RGBa, ColorType.FLOAT16)
    colorBuffer("position", ColorFormat.RGBa, ColorType.FLOAT32)
}
```

Filters and Post-processing

Since OPENRNDR has extensive support for rendering to off-screen buffers it is easy to apply effects and filters to the off-screen buffers.

Basic usage

To demonstrate the ease of using filters we show an example of applying a blur filter to a drawing on a render target.

```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        // -- create an offscreen render target
        val offscreen = renderTarget(width, height) {
            colorBuffer()
            depthBuffer()
        }
        // -- create blur filter
        val blur = BoxBlur()

        // -- create a colorbuffer to hold the blur results
```

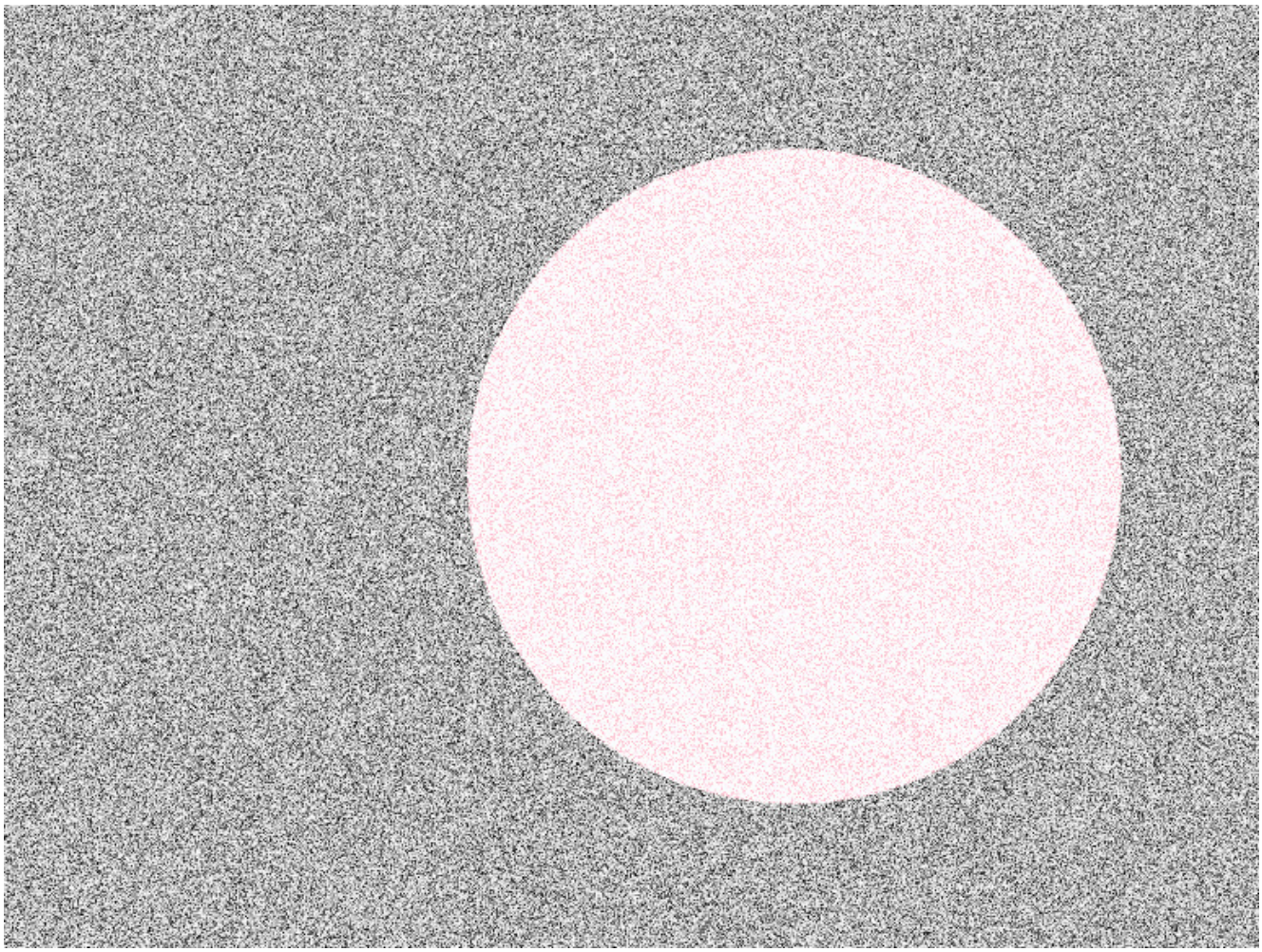
```
        val blurred = colorBuffer(width, height)



        extend {
            // -- draw to offscreen buffer
            drawer.isolatedWithTarget(offscreen) {
                clear(ColorRGBa.BLACK)
                fill = ColorRGBa.PINK
                stroke = null
                circle(cos(seconds) * 100.0 + width / 2, sin(seconds) * 100.0 + height / 2.0, 100.0 + 100.0 *
cos(seconds * 2.0))
            }
            // -- set blur parameters
            blur.window = 30

            // -- blur offscreen's color buffer into blurred
            blur.apply(offscreen.colorBuffer(0), blurred)
            drawer.image(blurred)
        }
    }
}
```

Link to the full example

Writing your own filters

You may be wondering how to create your own filters. If so, good news, it is fairly easy to write your own filter if you are familiar with fragment shaders in GLSL. The easiest way to write your own filter is to use the `Filter` class by extending it. The `Filter` class takes care of setting up render state, geometry and projections so all you have to do is write a shader.

What follows is an example of how to create a Filter from a shader whose code is stored as a String. The filter we will be making is a simple noise filter.



```
fun main() = application {
    val noiseShader = """
        #version 330
        // -- part of the filter interface, every filter has these
        in vec2 v_texCoord0;
        uniform sampler2D tex0;
        out vec4 o_color;

        // -- user parameters
        uniform float gain;
        uniform float time;

        #define HASHSCALE 443.8975
        vec2 hash22(vec2 p) {
            vec3 p3 = fract(vec3(p.xyx) * HASHSCALE);
            p3 += dot(p3, p3.yzx+19.19);
            return fract(vec2((p3.x + p3.y)*p3.z, (p3.x+p3.z)*p3.y));
        }

        void main() {
            float n = hash22(v_texCoord0+vec2(time)).x;
            // here we read from the input image and add noise
            vec4 color = texture(tex0, v_texCoord0) + vec4(vec3(n), 0.0) * gain;
            o_color = color;
        }
        """
```

```
    class Noise : Filter(filterShaderFromCode(noiseShader, "noise-shader")) {
        // -- note the 'by parameters' here; this is what wires the fields up to the uniforms
        var gain: Double by parameters
        var time: Double by parameters

        init {
            gain = 1.0
            time = 0.0
        }
    }
    program {
        // -- create the noise filter
        val noise = Noise()
        val offscreen = renderTarget(width, height) {
            colorBuffer()
            depthBuffer()
        }

        extend {
            // -- draw to offscreen buffer
            drawer.isolatedWithTarget(offscreen) {
                clear(ColorRGBa.BLACK)
                fill = ColorRGBa.PINK
                stroke = null
                circle(cos(seconds) * 100.0 + width / 2, sin(seconds) * 100.0 + height / 2.0, 100.0 + 100.0 *
cos(seconds * 2.0))
            }
            // apply the noise on and to offscreen.colorBuffer(0),
            // this only works for filters that only read from
            // the current fragment.
            noise.time = seconds
            noise.gain = 1.0
            noise.apply(offscreen.colorBuffer(0), offscreen.colorBuffer(0))

            drawer.image(offscreen.colorBuffer(0))
        }
    }
}
```

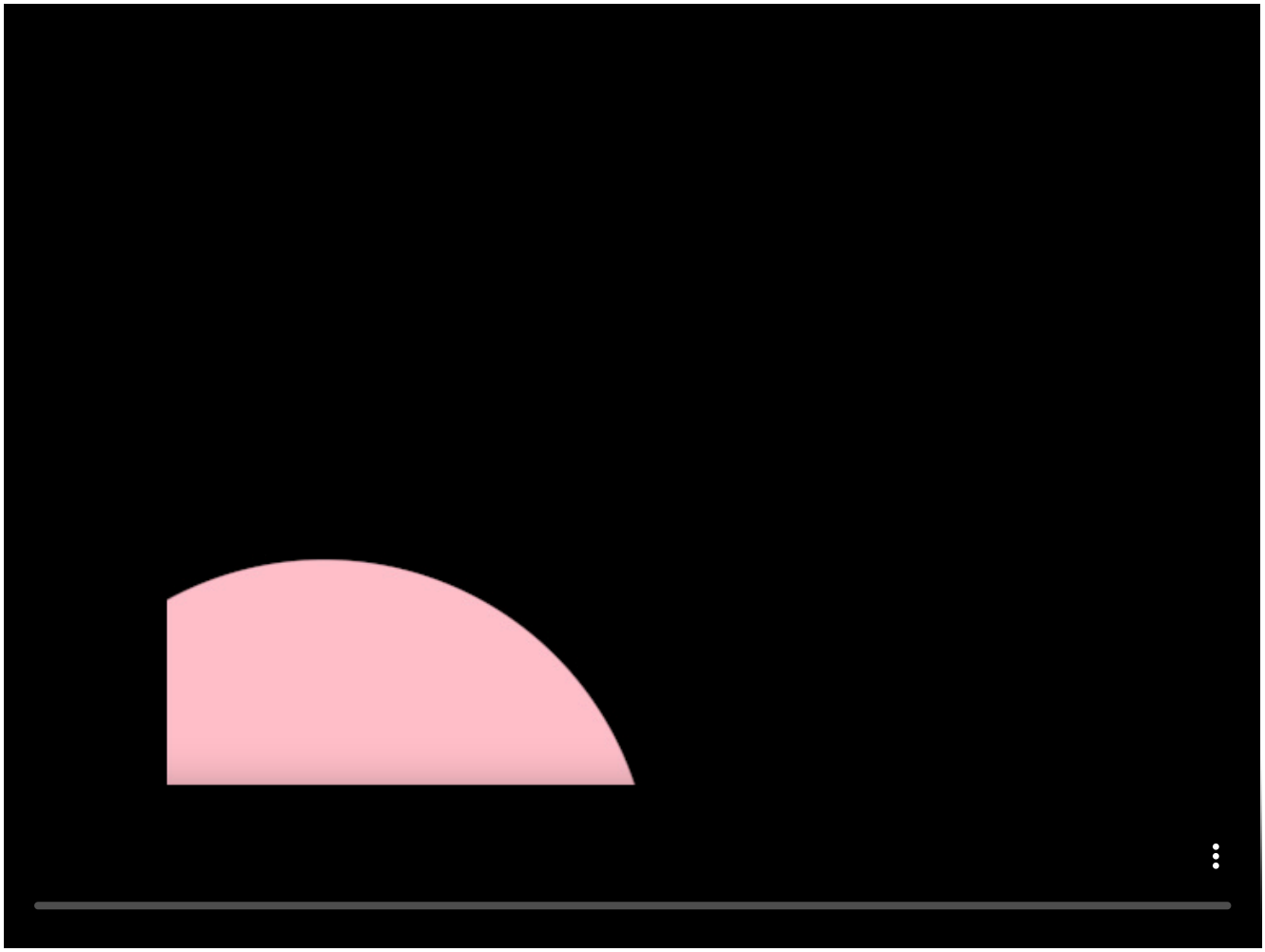
Link to the full example

The orx-fx library

A repository of ready-to-use filters for OPENRNDR can be found in the ORX repository, a partial index of the filters can be found in the orx-fx chapter

Clipping

OPENRNDR's drawer supports a single rectangular clip mask.



```
fun main() = application {
    program {
        extend {
            drawer.stroke = null
            drawer.fill = ColorRGBa.PINK

            // -- set the rectangular clipping mask
            drawer.drawStyle.clip = Rectangle(100.0, 100.0, width - 200.0, height - 200.00)

            drawer.circle(cos(seconds) * width / 2.0 + width / 2.0, sin(seconds) * height / 2.0 + height / 2.0, 200.0)

            // -- restore clipping
            drawer.drawStyle.clip = null
        }
    }
}
```

Link to the full example

Asynchronous image loading

In scenarios in which images are required to be loaded without blocking the draw thread you can use `ColorBufferLoader`. This loader runs on its own thread and uses shared contexts of the underlying graphics API (OpenGL).

`ColorBufferLoader` maintains a prioritized loading queue, such that most recently touched images will be loaded first.

`ColorBufferLoader` maintains an unloading queue, such that image proxies that have not been touched for 5000ms will be unloaded automatically (unless the proxy is requested to be persistent).

The `ColorBufferLoader` loads URL strings, it supports loading from file and http/https.

```
fun main() = application {
    program {
        extend {
            val proxy = colorBufferLoader.loadFromUrl("https://avatars3.githubusercontent.com/u/31103334?s=200&v=4")
            proxy.colorBuffer?.let {
                drawer.image(it)
            }
        }
    }
}
```

Here we see that `fromUrl` can safely be called many times as `ColorBufferLoader` caches these requests. Accessing `proxy.colorBuffer` touches its internal timestamp and queues the image for loading if the proxy state is `NOT_LOADED`.

Handling errors

Sometimes loading fails; for example because the file was not found, or the http connection timed out. `ColorBufferLoader` signals such errors by placing `ColorBufferProxy` in `RETRY` state. The proxy can be taken out of this state by calling its `.retry()` function.

Alternatively some errors cannot be retried; these will be signalled by setting the proxy state to `ERROR`. Such errors should only occur if the loading of the image causes a non-IOException to be thrown.

Cancelling a queued image

Cancelling an image which is queued (and is thus in `QUEUED` state) for loading is done by calling `.cancel()` on the `ColorBufferProxy`. This will set the proxy state back to `NOT_LOADED`.

Shade styles

Shade styles are used to change the drawing behaviour of the `Drawer` affecting the appearance of all drawing primitives.

Shade styles are composed of two types of transforms: vertex transforms and fragment transforms. The two transforms are applied in separate stages of the rendering process. In the vertex transform it is possible to change the geometry of what is drawn, and in the fragment transform it is possible to change the appearance of that geometry. A shade style can affect vertices, fragments or both.

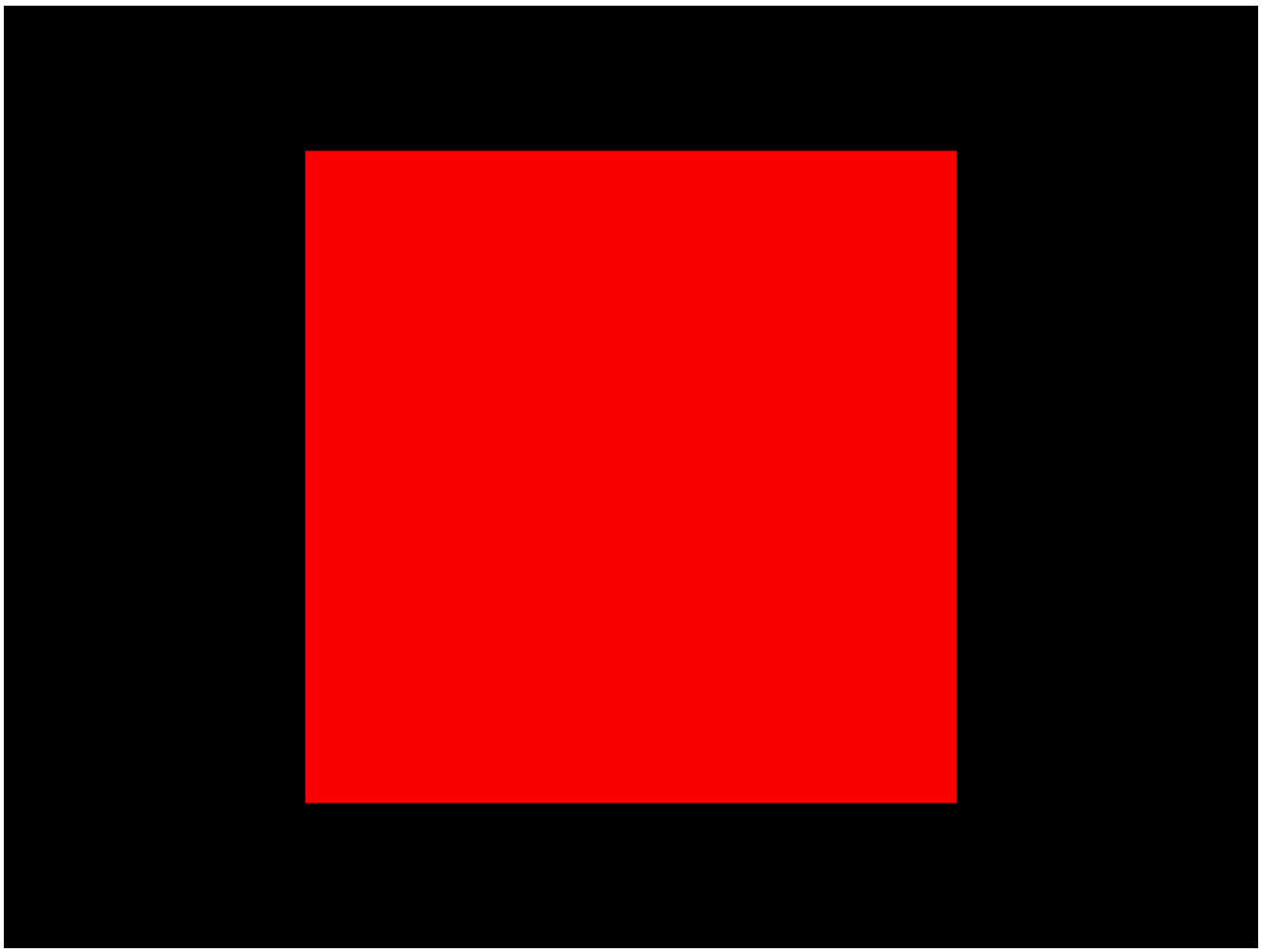
A selection of preset ready-to-use shade styles is provided by `orx-shade-styles`

For those interested in authoring shade styles it is helpful to have some basic understanding of shaders and GLSL.

Basic usage

In essence shade styles are fragments of GLSL code that are inserted into OPENRNDRs templated shaders.

As a quick first step we override the output to red in the following snippet

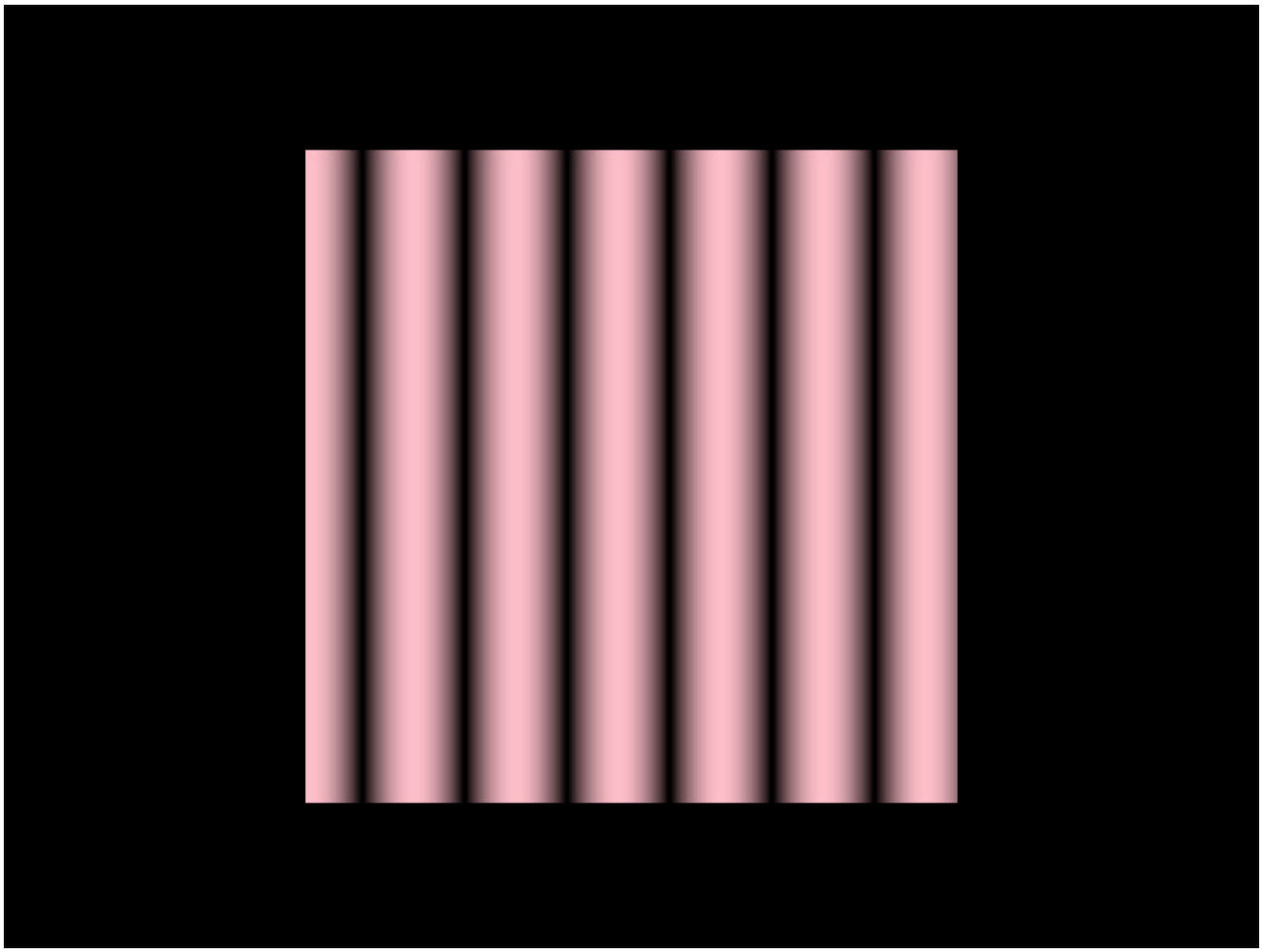


```
fun main() = application {
    program {
        extend {
            drawer.shadeStyle = shadeStyle {
```

```
                fragmentTransform = "x_fill.rgb = vec3(1.0, 0.0, 0.0);"
            }
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.rectangle(width / 2.0 - 200.0, height / 2.0 - 200.0, 400.0, 400.00)
        }
    }
}
```

Link to the full example

The idea of shade styles is to allow more complex changes in the appearance. In the next snippet we create a wavy pattern by using cosines and the screen position.



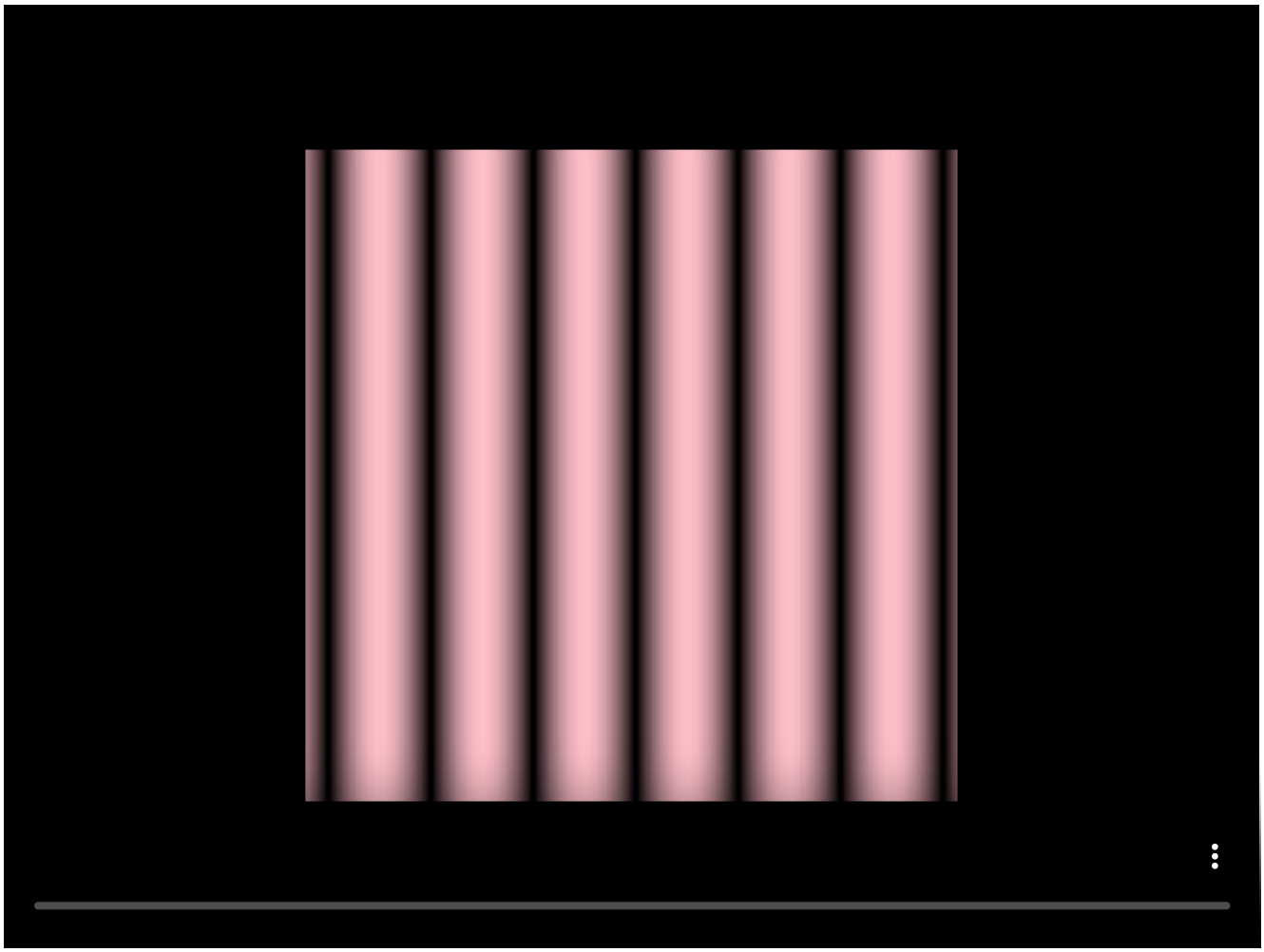
```
fun main() = application {
    program {
        extend {
            drawer.shadeStyle = shadeStyle {
                fragmentTransform = """
 float c = cos(c_screenPosition.x * 0.1) * 0.5 + 0.5;
 x_fill.rgb *= vec3(c, c, c);
                    """.trimMargin()
            }
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.rectangle(width / 2.0 - 200.0, height / 2.0 - 200.0, 400.0, 400.00)
```

```
        }
    }
}
```

Link to the full example

In the next step we introduce animation by adding an external clock signal to the shade style. Shade styles have *parameters* that can be used for this.

Notice how parameters like `time` receive a `p_` prefix in the GLSL world. This makes it easy to distinguish the uniforms we send into shaders from other variables declared by the framework.



```
fun main() = application {
    program {
        extend {
            drawer.shadeStyle = shadeStyle {
                fragmentTransform = """
 float c = cos(c_screenPosition.x * 0.1 + p_time) * 0.5 + 0.5;
 x_fill.rgb *= vec3(c, c, c);
                    """.trimMargin()
                parameter("time", seconds)
            }
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.rectangle(width / 2.0 - 200.0, height / 2.0 - 200.0, 400.0, 400.0)
        }
```

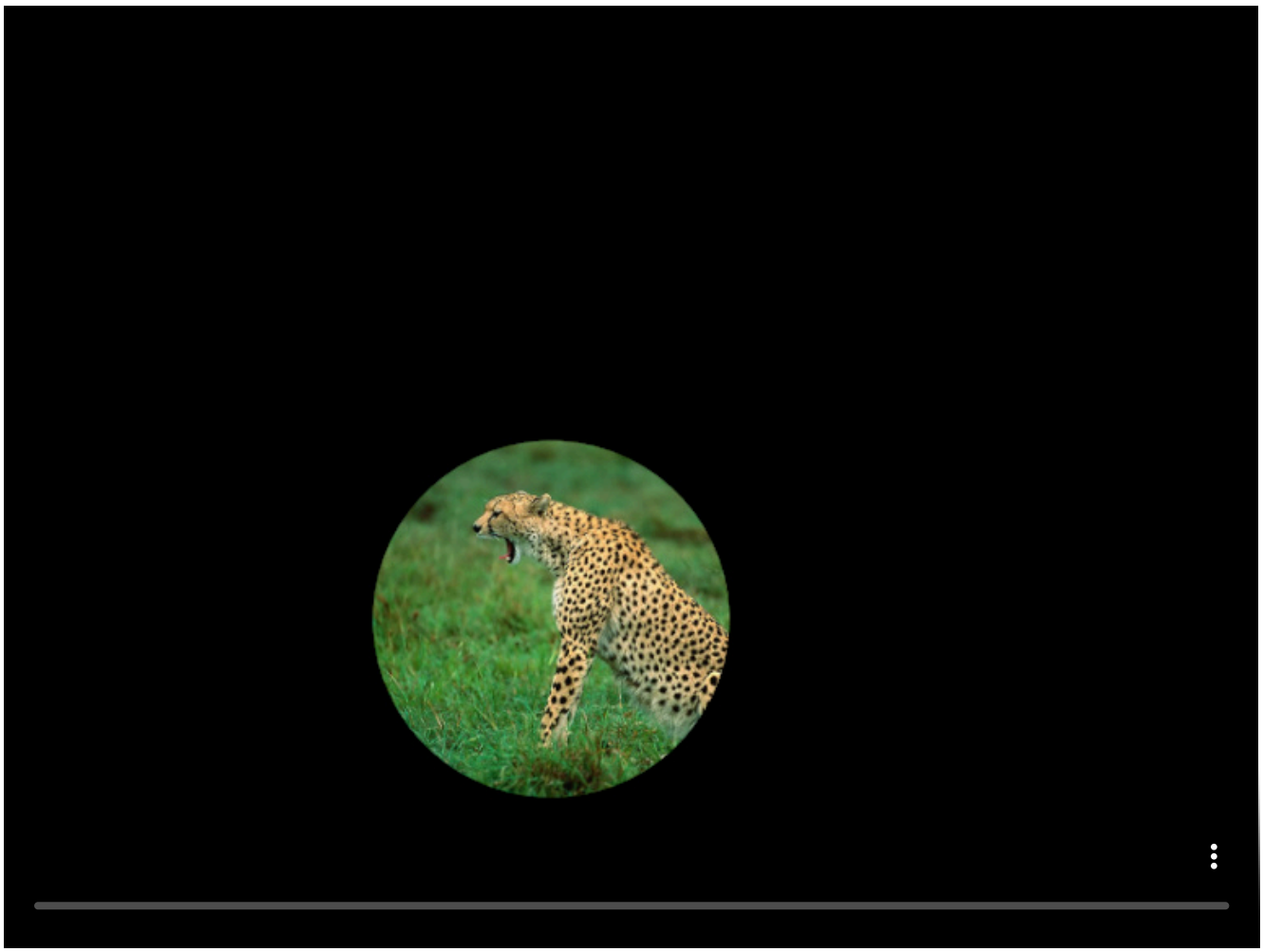
```
    }
}
```

Link to the full example

Usage examples

Here follow some examples of common problems that are solved using shade styles.

Mapping images on shapes



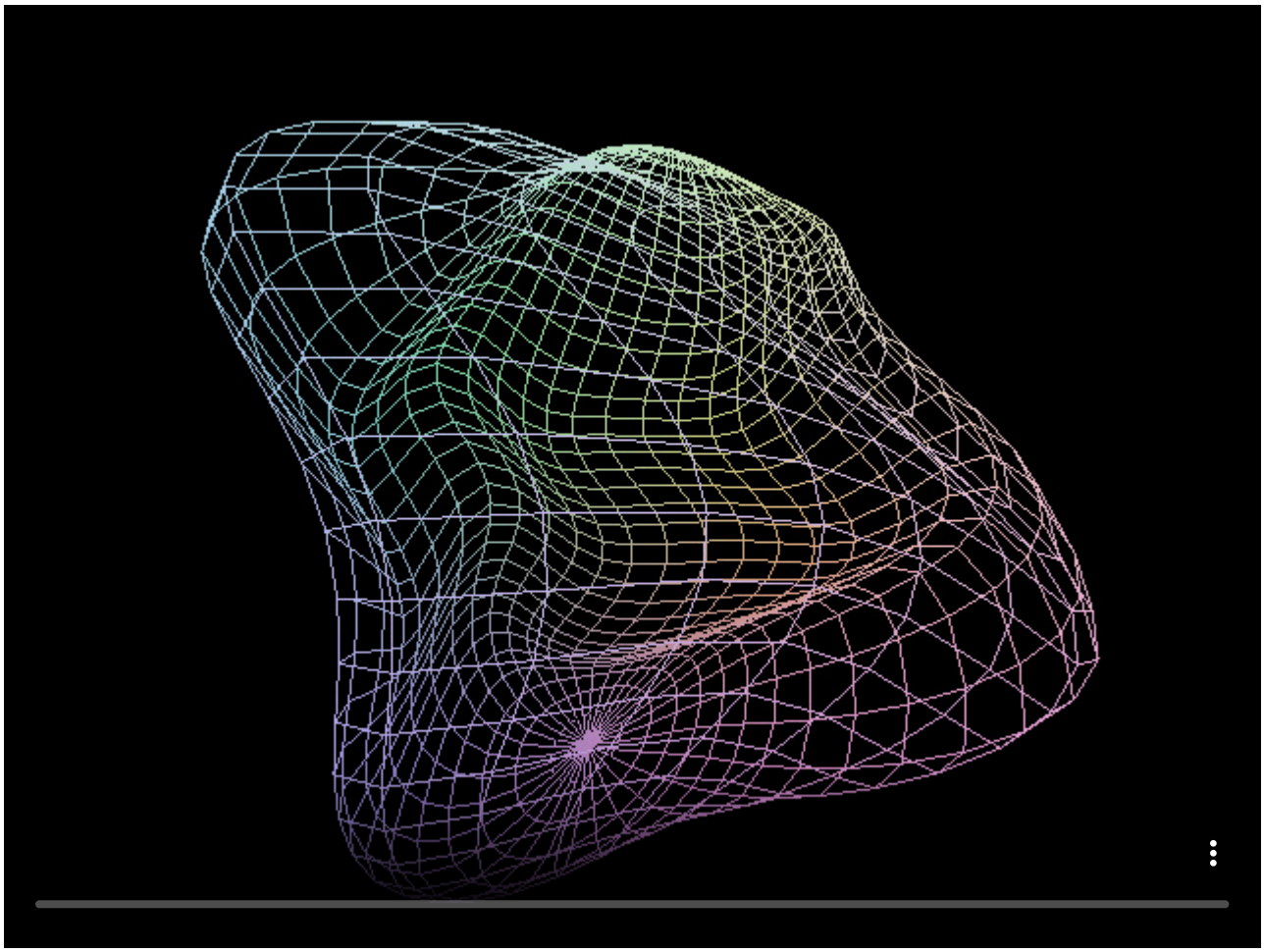
```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        image.filter(MinifyingFilter.LINEAR_MIPMAP_NEAREST, MagnifyingFilter.LINEAR)
        extend {
            drawer.shadeStyle = shadeStyle {
                fragmentTransform = """
                        vec2 texCoord = c_boundsPosition.xy;
                        texCoord.y = 1.0 - texCoord.y;
                        vec2 size = textureSize(p_image, 0);
                        texCoord.x /= size.x/size.y;
                        x_fill = texture(p_image, texCoord);
                    """
                parameter("image", image)
            }
```

```
            val shape = Circle(width / 2.0, height / 2.0, 110.0).shape
            drawer.translate(cos(seconds) * 100.0, sin(seconds) * 100.0)
            drawer.shape(shape)
        }
    }
}
```

Link to the full example

3D mesh distortion

This example shows that one can also modify the vertex shader, in this case to displace vertices using sine functions. The current time in seconds is passed into the shader to produce a wavy effect. The fragment shader uses sine functions to specify colors depending on the world position of each vertex.



```
fun main() = application {
    program {
        val sphere = sphereMesh(32, 32, 0.6)
        val style = shadeStyle {
            vertexTransform = """
                    vec3 p = x_position * 8.0 + p_seconds;
                    // displace the vertices
                    x_position.x += sin(p.y) * 0.1;
                    x_position.y += sin(p.z) * 0.1;
                    x_position.z += sin(p.x) * 0.1;
                """.trimIndent()

            fragmentTransform = """
```

```
                    vec3 c = sin(v_worldPosition) * 0.5 + 0.5;
                    x_fill = vec4(c, 1.0);
                """.trimIndent()
        }

        val camera = OrbitalCamera(Vector3.UNIT_Z, Vector3.ZERO)

        extend(camera)
        extend {
            camera.rotate(0.2, 0.0)
            style.parameter("seconds", seconds)
            drawer.shadeStyle = style
            drawer.vertexBuffer(sphere, DrawPrimitive.LINES)
        }
    }
}
```

Link to the full example

The shade style language

Prefix overview

Listed below is an overview of all the prefixes used in the shade style language.

prefix	Scope	Description
u_	all	system uniforms passed in from Drawer
a_	vertex transform	vertex attribute
va_	fragment transform	varying attribute, interpolation passed from vertex to fragment shader
v_	fragment transform	varying values, interpolation passed from vertex to fragment shader
i_	vertex transform	instance attribute
vi_	fragment transform	varying instance attribute
x_	all	transformable value
p_	all	user provided value
o_	fragment transform	output value (always vec4)
d_	all	shader definitions

Standard uniforms

Listed below is an overview of all the prefixes used in the shade style language.

Uniform name	GLSL type	Description
u_modelNormalMatrix	mat4	matrix used to transform vertex normals from object to world space
u_modelMatrix	mat4	matrix used to transform vertices from object to world space
u_viewNormalMatrix	mat4	matrix used to transform vertex normals from world space to view space
u_viewMatrix	mat4	matrix used to transform vertices from world space to view space

Uniform name	GLSL type	Description
`u_projectionMatrix`	mat4	matrix used to transform vertices from view space to clip space
`u_contentScale`	float	the active content scale
`u_viewDimensions`	vec2	the dimensions of the target viewport
`u_fill`	vec4	the Drawer fill color
`u_stroke`	vec4	the Drawer stroke color
`u_strokeWeight`	float	the Drawer strokeWeight
`u_colorMatrix`	float[25]	the Drawer color matrix

Standard Attributes

Attributes are only directly accessible in the vertex transform. However interpolated forms of the the attributes are passed to the fragment transform.

Attribute name	GLSL type	Description
`a_position`	vec3	the position
`a_normal`	vec3	the normal
`a_color`	vec3	the color

The interpolated versions that are only accessible in the fragment transform.

Attribute name	GLSL type	Description
`va_position`	vec3	the interpolated position
`va_normal`	vec3	the interpolated normal
`va_color`	vec3	the interpolated color

Other interpolated values

These values are calculated in the vertex shader and only accessible in the fragment transform.

Value name	GLSL type	Description
`v_worldNormal`	vec3	interpolated normal in world coordinates
`v_viewNormal`	vec3	interpolated normal in view coordinates
`v_worldPosition`	vec3	interpolated position in world coordinates
`v_viewPosition`	vec3	interpolated position in view coordinates
`v_clipPosition`	vec4	interpolated position in clip coordinates
`v_modelNormalMatrix`	mat4	non-interpolated (flat) model normal matrix

Transformable values

These are values that can be transformed using shade styles.

VERTEX TRANSFORM

Variable name	GLSL type	Description
`x_position`	`vec3`	vertex position, initialized with value `a_position`
`x_normal`	`vec3`	vertex normal, initialized with value `a_normal`
`x_viewMatrix`	`mat4`	view matrix
`x_normalMatrix`	`mat4`	normal matrix, initialized with `normalMatrix`
`x_projectionMatrix`	`mat4`	projection matrix, initialized with `projectionMatrix`
`x_modelMatrix`	`mat4`	model matrix, initialized with `modelMatrix`
`x_modelNormalMatrix`	`mat4`	model normal matrix, initialized with `modelNormalMatrix`
`x_viewNormalMatrix`	`mat4`	view normal matrix, initialized with `viewNormalMatrix`

FRAGMENT TRANSFORM

Variable name	GLSL type	Description
`x_fill`	`vec4`	The fill color written to the fragment
`x_stroke`	`vec4`	The stroke color written to the fragment

Constants

Constant name	Scope	GLSL type	Description
`c_element`	all	int	the element index in batched rendering
`c_instance`	all	int	the instance index in instanced rendering
`c_screenPosition`	fragment transform	vec2	the position on screen in device coordinates
`c_contourPosition`	fragment transform	float	the position on the contour, between 0.0 and contour.length. Only non-zero when drawing line segments and contours
`c_boundsPosition`	fragment transform	vec3	the bounding box position of the current shape or contour stored in `.xy`
`c_boundsSize`	fragment transform	vec3	the bounding box size of the current shape or contour stored in `.xy`

Parameters

Parameters can be used to supply external data to transforms. Parameters are translated to shader uniforms and are exposed by uniforms with the `p_` prefix.

COLORBUFFER PARAMETERS

Can be used to map images.

BUFFERTEXTURE PARAMETERS

Can be used to map custom values.

SUPPORTED PARAMETER TYPES:

JVM type	GLSL type
`float`	`float`
`Vector2`	`vec2`

JVM type	GLSL type
`Vector3`	`vec3`
`Vector4`	`vec4`
`ColorRGBa`	`vec4`
`Matrix44`	`mat4`
`DepthBuffer`	`sampler2D`
`ColorBuffer`	`sampler2D`
`BufferTexture`	`samplerBuffer`

SOURCE CODE

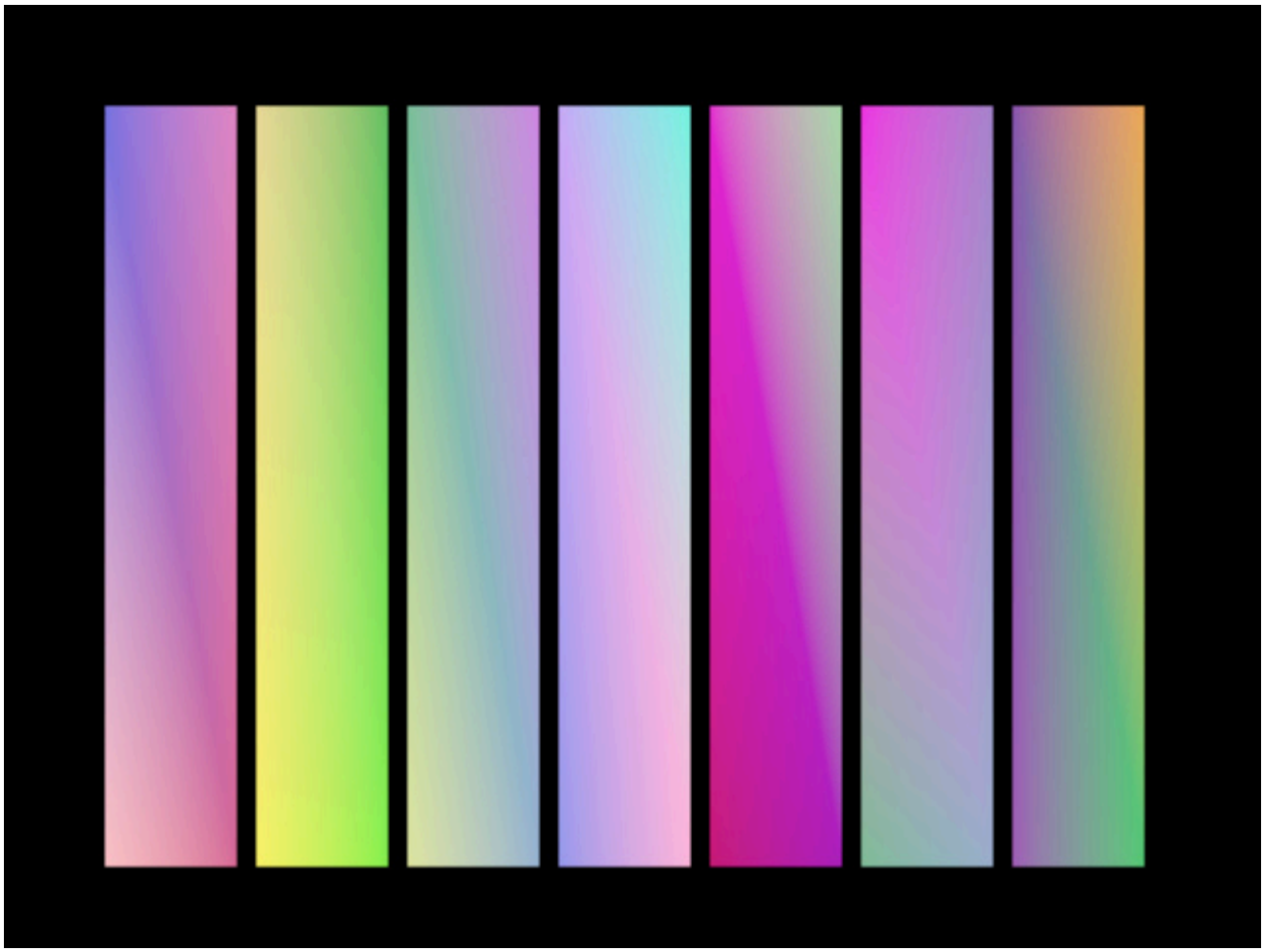
One can explore the source code to find out how attributes and uniforms are used:

- ShadeStyleGLSL.kt (JVM)
- ShadeStyleGLSL.kt (WEBGL)
- ShaderGeneratorsGLCommon.kt

Vertex and fragment preambles

Since the code declared in `vertexTransform` and `fragmentTransform` end up inside the `main()` function of shader programs, you may be wondering how you could declare custom functions, for example, to calculate a noise value, a random value or to do matrix rotations (common functions used in shader programs).

To achieve this, we use two keywords: `vertexPreamble` and `fragmentPreamble`. The code found in these strings gets inserted into the shader programs *before* the `main()` function, allowing us to declare custom functions or even `varying`s (to pass values from the vertex shader into the fragment shader).



```
fun main() = application {
    program {
        val style = shadeStyle {
            // Define the `random` function and declare a `c`
            // variable to pass to the fragment shader.
            vertexPreamble = """
                    float random(vec2 st) {
                        return fract(sin(dot(st.xy,
                            vec2(12.9898, 78.233))) * 43758.5453123);
                    }
                    out vec3 c;
                """.trimIndent()

            // Calculate the value of `c` per vertex.
            // It will get interpolated by the GPU.
            vertexTransform = """
                    c.r = random(x_position.xy);
                    c.g = random(x_position.yx);
                    c.b = random(x_position.xy + 1.0);
                """.trimIndent()

            // Declare a `c` variable to receive from the vertex shader.
            fragmentPreamble = "in vec3 c;"
            // Use the value of `c` to set the color of a pixel.
            fragmentTransform = "x_fill.rgb = c;"
        }
        extend {
```

```
            drawer.shadeStyle = style
            repeat(7) {
                // Notice how we do not set `drawer.fill`.
                drawer.rectangle(50.0 + it * 77, 50.0, 70.0, 390.0)
            }
        }
    }
}
```

Link to the full example

Debugging tip

If you want to study the vertex or the fragment shader in their final form, a simple technique is to provoke an error. Throw in some incorrect syntax: for example, add an `x` character at the beginning of the vertex or the fragment shader.

When you try to run the program, it will fail, and a file called `ShaderError.glsl` will be written into the project's root. Study that file to understand what attributes and uniforms are available and to figure out why your program fails in case the syntax error was not intentional.

Custom rendering

OPENRNDR is designed with the idea that users should be able to draw beyond the primitives offered by Drawer.

Vertex buffers

A vertex buffer is a (on the GPU residing) amount of memory in which vertices that describe geometry are stored. A single vertex consists of a number of customizable attributes such as position, normal and color. In OPENRNDR the attributes of a vertex are described using a VertexFormat.

Vertex buffers allow geometry to be prepared and stored in such a way that graphics hardware can draw it directly.

Declaring a vertex format

A vertex format declaration consists of a list of vertex attributes. Such a declaration is made using the *vertex format builder*: `vertexFormat {}`

To illustrate the declaration procedure let us give an example of a very simple vertex format. This vertex format consists solely of a three dimensional *position* attribute:

```
val vf = vertexFormat {
    position(3)
}
```

Listed below are the attributes that can be added to the vertex format.

name	type	description
`position(dimensions: Int)`	`FLOAT32, VECTOR[2,3,4]_FLOAT32`	position attribute
`normal(dimensions: Int)`	`FLOAT32, VECTOR[2,3,4]_FLOAT32`	normal attribute
`textureCoordinate(dimensions: Int)`	`FLOAT32, VECTOR[2,3,4]_FLOAT32`	texture coordinate attribute
`color(dimensions: Int)`	`FLOAT32, VECTOR[2,3,4]_FLOAT32`	color attribute
`attribute(name: String, type: VertexElementType, arraySize:Int = 1)`	`type`	custom attribute

A more complex vertex format declaration would then look like this:

```
val vf = vertexFormat {
    position(3)
    normal(3)
    color(4)
    attribute("objectID", VertexElementType.FLOAT32)
}
```

Creating a vertex buffer

The vertexBuffer() function is used to create a VertexBuffer instance. For example to create a vertex buffer holding 1000 vertices in our previously defined vertex format `vf` we use the following:

```
val geometry = vertexBuffer(vf, 1000)
```

Placing data in the vertex buffer

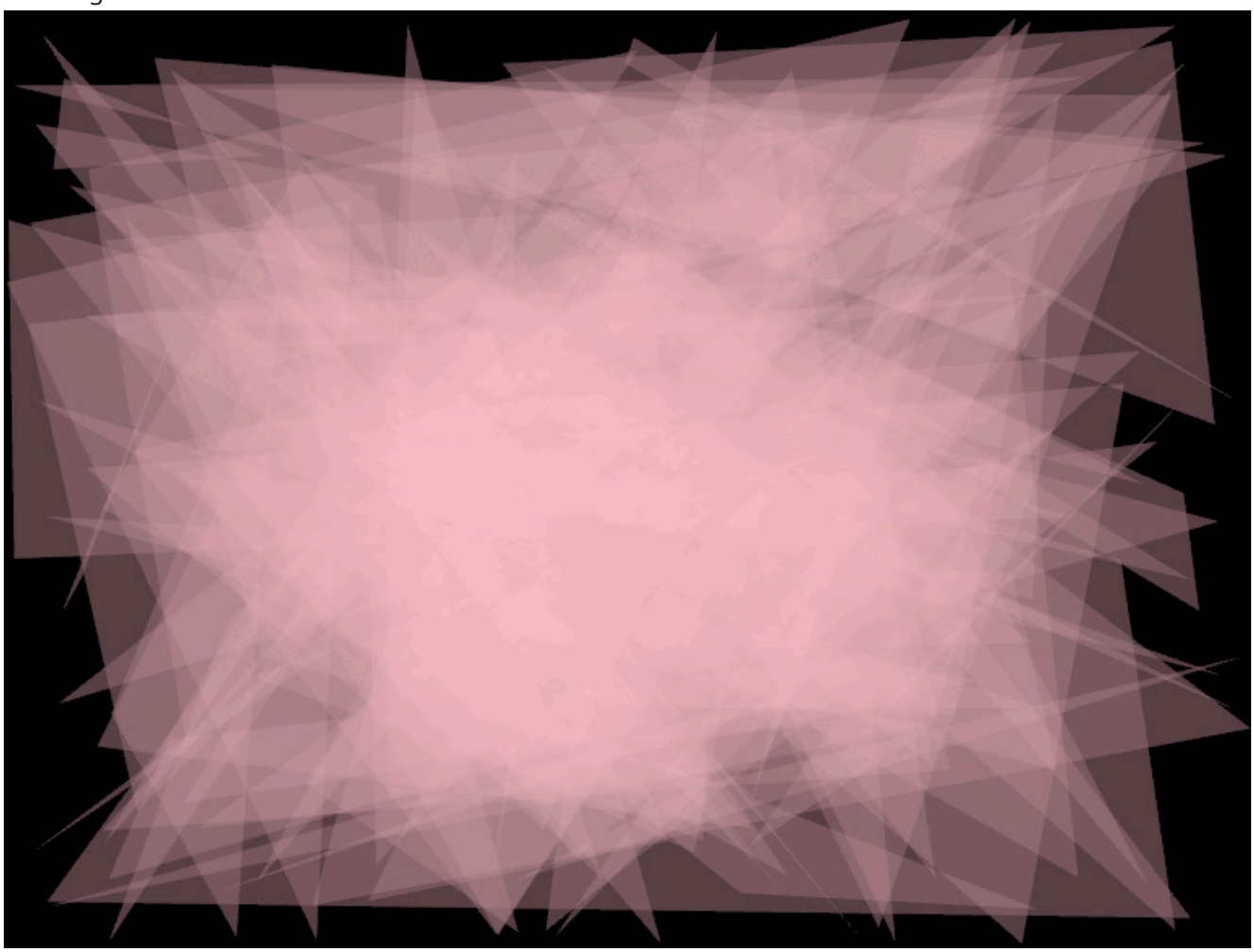
Now that a vertex format has been defined and a vertex buffer has been created we can place data in the vertex buffer. The data placed in the vertex buffer must closely match the vertex format; any form of mismatch will lead to surprising and/or undefined behaviour.

The `VertexBuffer.put {}` function is the easiest and safest way of placing data in the vertex buffer.

In the following example we create a very simple vertex format that holds just a position attribute. After creation we fill the vertex buffer with random data.

```
val geometry = vertexBuffer(vertexFormat {
    position(3)
}, 1000)
geometry.put {
    for (i in 0 until 1000) {
        write(Vector3(Math.random() - 0.5, Math.random() - 0.5, Math.random() - 0.5))
    }
}
```

Drawing vertex buffers



```
fun main() = application {
    program {
        val geometry = vertexBuffer(vertexFormat {
```

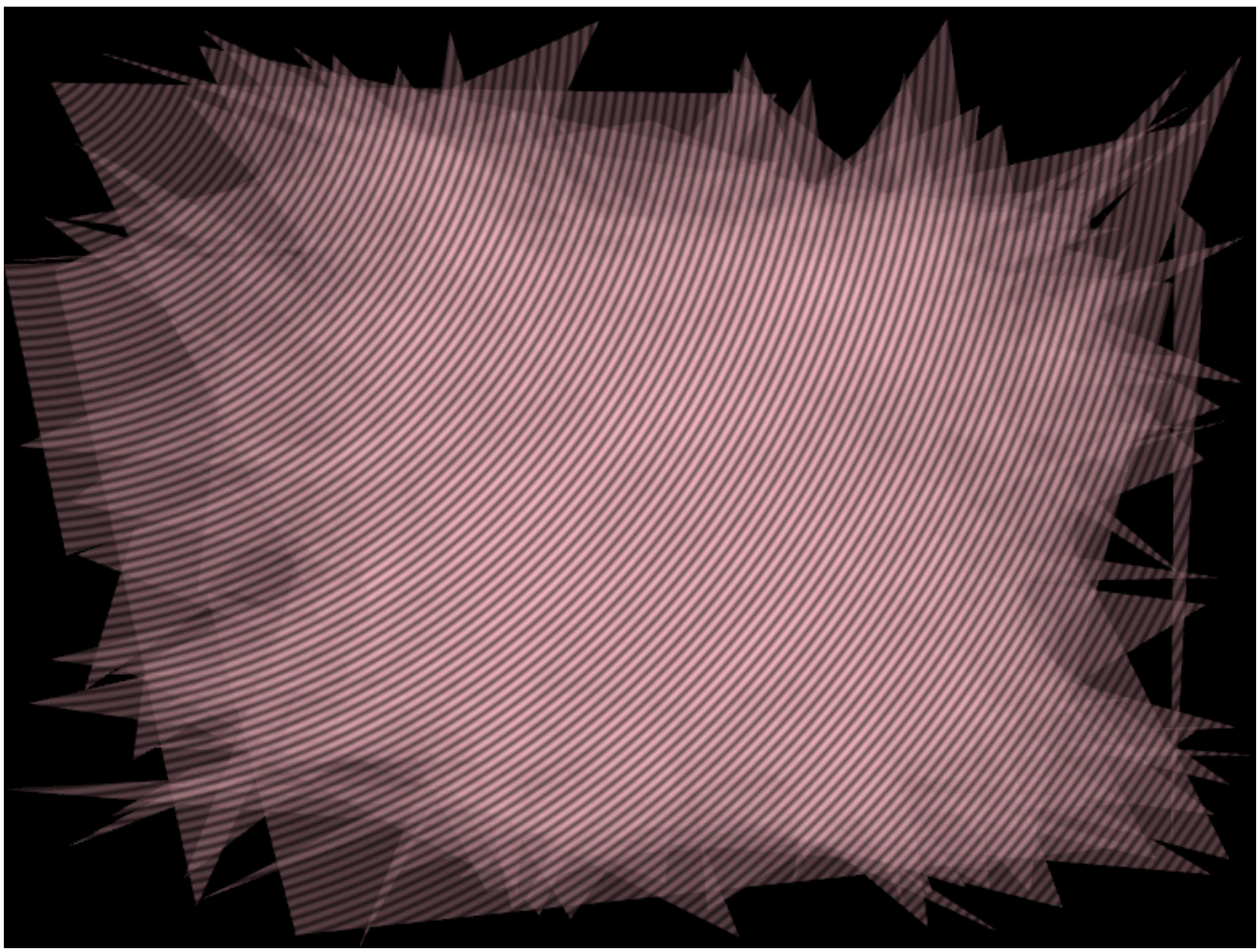
```
            position(3)
        }, 3 * 100)

        geometry.put {
            for (i in 0 until geometry.vertexCount) {
                write(Vector3(Math.random() * width, Math.random() * height, 0.0))
            }
        }

        extend {
            drawer.fill = ColorRGBa.PINK.opacify(0.1)
            drawer.vertexBuffer(geometry, DrawPrimitive.TRIANGLES)
        }
    }
}
```

Link to the full example

Shading geometry

Drawing using Drawer.vertexBuffer will be with respect to the set shade style. Using shade styles the appearance of the geometry in the vertex buffer can be fully customized.

The last snippet can be modified to include a simple shading over the geometry



```
fun main() = application {
    program {
```

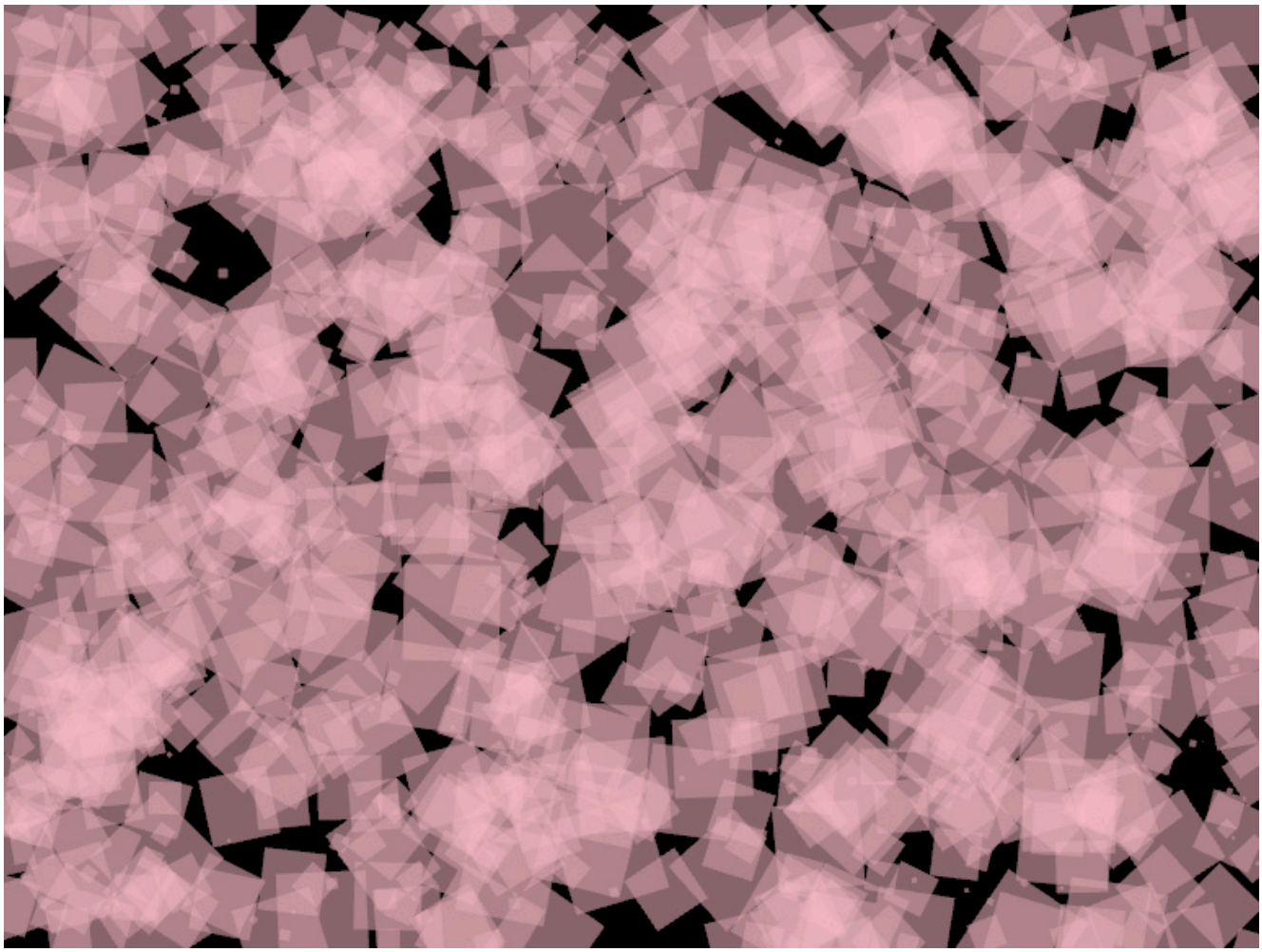
```
        val geometry = vertexBuffer(vertexFormat {
            position(3)
        }, 3 * 100)

        geometry.put {
            for (i in 0 until geometry.vertexCount) {
                write(Vector3(Math.random() * width, Math.random() * height, 0.00))
            }
        }

        extend {
            drawer.shadeStyle = shadeStyle {
                fragmentTransform = """x_fill.rgb *= vec3(cos(length(v_viewPosition))*0.4+0.6);"""
            }
            drawer.fill = ColorRGBa.PINK.opacify(0.1)
            drawer.vertexBuffer(geometry, DrawPrimitive.TRIANGLES)
        }
    }
}
```

Link to the full example

Drawing instances

The same geometry can be drawn more than once in a single draw call.

Per instance attributes can be stored in a vertex buffer. For example a transform per instance can be realized by creating a second vertex buffer which will contain matrix attributes.



```
fun main() = application {
    configure {
        width = 770
        height = 578
    }
    program {
        // -- create the vertex buffer
        val geometry = vertexBuffer(vertexFormat {
            position(3)
        }, 4)

        // -- fill the vertex buffer with vertices for a unit quad
        geometry.put {
            write(Vector3(-1.0, -1.0, 0.0))
            write(Vector3(-1.0, 1.0, 0.0))
            write(Vector3(1.0, -1.0, 0.0))
            write(Vector3(1.0, 1.0, 0.0))
        }

        // -- create the secondary vertex buffer, which will hold transformations
        val transforms = vertexBuffer(vertexFormat {
            attribute("transform", VertexElementType.MATRIX44_FLOAT32)
        }, 1000)

        // -- fill the transform buffer
        transforms.put {
```

```
            repeat(transforms.vertexCount) {
                write(transform {
                    translate(Math.random() * width, Math.random() * height)
                    rotate(Vector3.UNIT_Z, Math.random() * 360.0)
                    scale(Math.random() * 30.0)
                })
            }
        }
        extend {
            drawer.fill = ColorRGBa.PINK.opacify(0.25)
            drawer.shadeStyle = shadeStyle {
                vertexTransform = "x_viewMatrix = x_viewMatrix * i_transform;"
            }
            drawer.vertexBufferInstances(listOf(geometry), listOf(transforms), DrawPrimitive.TRIANGLE_STRIP, 1000)
        }
    }
}
```

Link to the full example

Concurrency and multi-threading

Here we talk about OPENRNDR's primitives for concurrency.

The largest complication in multi-threading in OPENRNDR lies in how the underlying graphics API (OpenGL) can only be used from threads on which an OpenGL context is active. This means that any interactions with `Drawer`, `ColorBuffer`, `VertexBuffer`, `RenderTarget`, `Shader`, `BufferTexture`, `CubeMap`, `ArrayTexture` can only be performed on the primary draw thread or specially created draw threads.

Coroutines

Coroutines, as they are discussed here are a Kotlin specific framework for concurrency. Please read the coroutines overview in the Kotlin reference for an introduction to coroutines

`Program` comes with its own coroutine dispatcher, which guarantees that coroutines will be handled on the primary draw thread. This means that coroutines when executed or resumed by the program dispatcher will block the draw thread.

In the following example we launch a coroutine that slowly counts to 99. Note that the delay inside the coroutine does *not* block the primary draw thread.

```
fun main() = application {
    program {
        var once = true
        extend {
            if (once) {
                once = false
                launch {
                    for (i in 0 until 100) {
                        println("Hello from coroutine world ($i)")
                        delay(100)
                    }
                }
            }
        }
    }
}
```

You may be asking, what is the purpose of the `Program` dispatcher if running coroutines blocks the primary draw thread. The answer is, blocking coroutines are useful when the work performed is light. Light work includes waiting for (off-thread) coroutines to complete and using the results to write to graphics resources.

In the below example we nest coroutines; the outer one is launched on the `Program` dispatcher, the inner one is launched on the `GlobalScope` dispatcher. The `GlobalScope` dispatcher executes the coroutine on a thread (from a thread pool) such that it does not block the primary draw thread. By using `.join()` on the inner coroutine we wait for it to complete, waiting is non-blocking (thanks to coroutine magic!). Once the join operation completes we can write the results to a graphics resource on the primary draw thread.

```
fun main() = application {
    program {
```

```
        val colorBuffer = colorBuffer(512, 512)
        val data = ByteBuffer.allocateDirect(512 * 512 * 4)
        var once = true
        extend {
            if (once) {
                once = false
                launch {
                    // -- launch on GlobalScope
                    // -- this will cause the coroutine to be executed off-thread.
                    GlobalScope.launch {
                        // -- perform some faux-heavy calculations
                        val r = Random(100)
                        for (y in 0 until 512) {
                            for (x in 0 until 512) {
                                for (c in 0 until 4) {
                                    data.put(r.nextBytes(1)[0])
                                }
                            }
                        }
                    }.join() // -- wait for coroutine to complete

                    // -- write data to graphics resources
                    data.rewind()
                    colorBuffer.write(data)
                }
            }
        }
    }
}
```

Secondary draw threads

In some scenarios you may want to have a separate thread on which all graphic resources can be used and drawing is allowed. In those cases you use `drawThread`.

Most graphic resources can be used and shared between threads, with the exception of the `RenderTarget`.

In the next example we create a secondary draw thread and a `ColorBuffer` that is shared between the threads. On the secondary draw thread we create a `RenderTarget` with the color buffer attachment. The image is made visible on the primary draw thread.

```
fun main() = application {
    program {
        val result = colorBuffer(512, 512)
        var once = true
        var done = false
        val secondary = drawThread()

        extend {
            if (once) {
                once = false
                // -- launch on the secondary draw thread (SDT)
                secondary.launch {
```

```
                    // -- create a render target on the SDT.
                    val rt = renderTarget(512, 512) {
                        colorBuffer(result)
                    }

                    // -- make sure we use the draw thread's drawer
                    val drawer = secondary.drawer
                    drawer.withTarget(rt) {
                        drawer.ortho(rt)
                        drawer.clear(ColorRGBa.PINK)
                    }

                    // -- destroy the render target
                    rt.destroy()
                    finish()
                    // -- tell the main thread the work is done
                    done = true
                }
            }
            // -- draw the result when the work is done
            if (done) {
                drawer.image(result)
            }
        }
    }
}
```

Array textures

Array textures are a special type of texture that make it possible to access 2048 layers of texture data from a single texture sampler.

Array textures are encapsulated by the `ArrayTexture` interface

Creation

Array textures are created using the `arrayTexture` function.

```
fun main() = application {
    program {
        // -- create an array texture with 100 layers
        val at = arrayTexture(512, 512, 100)
    }
}
```

Writing to array textures

There are several ways to get texture data into array textures. Let's have a look at them.

One can copy from a ColorBuffer using `.copyTo()`

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)
        val cb = colorBuffer(512, 512)
        // -- copy from the color buffer to the 4th layer of the array texture
        cb.copyTo(at, 4)
    }
}
```

or copy from an array texture layer to another layer

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)
        // -- copy from the 2nd array texture layer to the 4th array texture layer
        at.copyTo(2, at, 4)
    }
}
```

or write to an array texture layer from a direct ByteBuffer

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)
        val buffer = ByteBuffer.allocateDirect(512 * 512 * 4)
```

```
        // fill the buffer with random data
        for (y in 0 until 512) {
            for (x in 0 until 512) {
                for (c in 0 until 4) {
                    buffer.put((Math.random() * 255).toInt().toByte())
                }
            }
        }
        buffer.rewind()
        // -- write the buffer into the 0th layer
        at.write(0, buffer)
    }
}
```

Drawing array textures

Array textures can be drawn using the `Drawer.image` functions.

As example we show how to draw the 0th layer of an array texture

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)

        extend {
            drawer.image(at, 0)
            // -- with position and size arguments
            drawer.image(at, 0, 100.0, 100.0, 256.0, 256.0)
        }
    }
}
```

We can also render batches of array textures by passing lists of layer indexes and source-target rectangle pairs.

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)

        extend {
            drawer.image(at, 0)

            val layers = mutableListOf<Int>()
            val rectangles = mutableListOf<Pair<Rectangle, Rectangle>>()

            for (i in 0 until 100) {
                layers.add((Math.random() * 100).toInt())
                val source = Rectangle(Math.random() * 512.0, Math.random() * 512.0, Math.random() * 512.0,
Math.random() * 512.0)
                val target = Rectangle(Math.random() * 512.0, Math.random() * 512.0, Math.random() * 512.0,
Math.random() * 512.0)
                rectangles.add(source to target)
            }
```

```
            drawer.image(at, layers, rectangles)
        }
    }
}
```

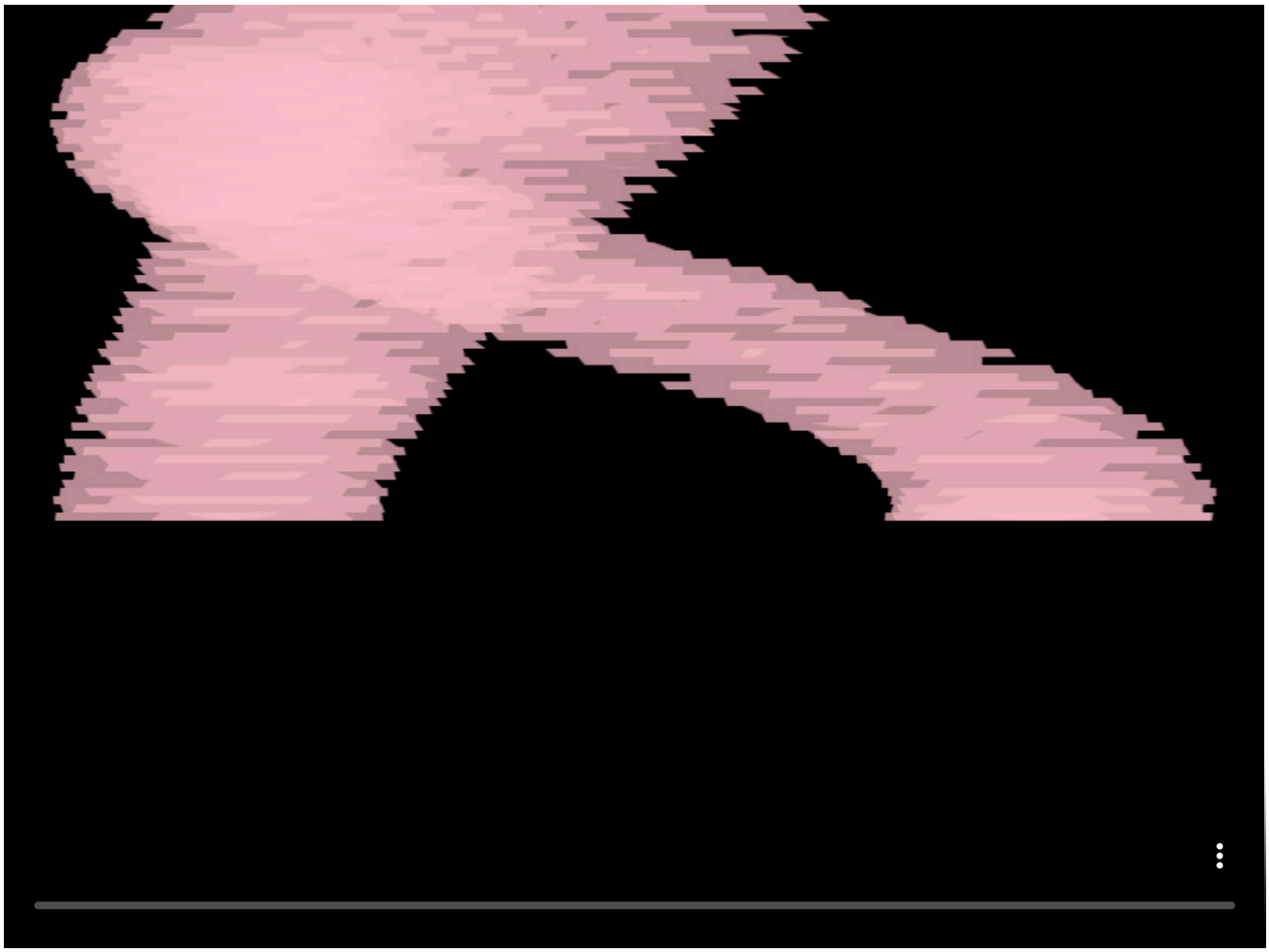
Drawing into array textures

Array textures can be used as attachment for render targets.

Here we show how to use a single layer from an array texture as an attachment for a render target.

```
fun main() = application {
    program {
        val at = arrayTexture(512, 512, 100)
        // -- create a render target
        val rt = renderTarget(512, 512) {
            // -- attach the 0th layer of the array texture
            arrayTexture(at, 0)
            depthBuffer()
        }

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.ortho(rt)
                drawer.clear(ColorRGBa.PINK)
            }
            drawer.image(at, 0)
        }
    }
}
```

Let's conclude this chapter by means of a small slit scanning demonstration. Here we use a single array texture and a list of render targets, all using different layers of the same array texture.



```
fun main() = application {

    program {
        val at = arrayTexture(770, 576, 116)
        val renderTargets = List(at.layers) {
            renderTarget(at.width, at.height) {
                arrayTexture(at, it)
            }
        }
        var index = 0
        extend {
            drawer.clear(ColorRGBa.BLACK)
            drawer.isolatedWithTarget(renderTargets[index % renderTargets.size]) {
                drawer.clear(ColorRGBa.BLACK)
                drawer.fill = ColorRGBa.PINK.opacify(0.5)
                drawer.stroke = null
                for (i in 0 until 20) {
                    drawer.circle(cos(seconds * 5.0 + i) * 256 + width / 2.0, sin(i + seconds * 6.32) * 256 + height /
2.0, 100.0)
                }
            }

            val layers = (0 until at.layers).map {
                (index - it).mod(at.layers)
            }
            val rectangles = (0 until at.layers).map {
```

```
                val span = Rectangle(0.0, it * 5.0, at.width * 1.0, 5.0)
                span to span
            }

            drawer.image(at, layers, rectangles)
            index++
        }
    }
}
```

Link to the full example

Interaction

edit on GitHub

TABLE OF CONTENTS

Events

Events are notifications sent by a sender to one or more subscribers indicating that an action has taken place.

Mouse and keyboard events are among the most commonly used events in OPENRNDR. Before we get to those in the next chapter, let's examine how to create our own custom events.

We will often use events in class instances, so let's create a simple class called `Thing`. This class will include an update method that will emit an event every 60 times it's called. Anyone listening to this event will receive it.

```
class Thing {
    val timeEvent = Event<Int>("time-event")

    private var frame = 0

    fun update() {
        if (++frame % 60 == 0) {
            timeEvent.trigger(frame / 60)
        }
    }
}
```

Sending an event

Notice how events carry a payload, in this case `Int`. This is convenient because it allows us to transmit information together with the event. Mouse and Keyboard events contain details about the mouse position or the key pressed. In this program we are free to choose any type, so lets just broadcasting a message containing the approximate time in seconds.

Passing a name in the event constructor is not necessary, but can be useful for logging and debugging.

Another thing to observe is that `timeEvent` is a public variable. If it was private we couldn't listen to it from outside by calling `thing.timeEvent.listen { ... }`.

At some point in our program execution we need to call `.trigger()` to broadcast the event. We can call it as many times as needed.

Listening to an event

The following small program shows how to listen to an event emitted by a class.

First, let's create one instance of the class called `thing`.

Next, listen to the event `thing` can emit (`timeEvent`).

```
fun main() = application {
    program {
        val thing = Thing()
        extend {
            thing.update()
        }
        thing.timeEvent.listen {
            println("timeEvent triggered! $it")
```

```
        }
    }
}
```

We see a line appear every second:

```
timeEvent triggered! 1
timeEvent triggered! 2
timeEvent triggered! 3
...
```

Events in coroutines and threads

By default our OPENRNDR programs run in a single thread, which happens to be the “rendering thread”. But what would happen if we sent Events from different threads or coroutines? Lets find out.

The `Blob` class is a copy of `Thing` with three changes:

1. To be able to spawn coroutines, we pass a `Program` in the constructor.
2. We add a second event called `doneWaiting`. We use `Unit` as a type when we don’t want to pass any useful data.
3. When the class is constructed, we launch a coroutine to wait for 3 seconds, then trigger the new `doneWaiting` event.

```
class Blob(program: Program) {
    val timeEvent = Event<Int>("time-event")
    val doneWaiting = Event<Unit>("done-waiting")

    private var frame = 0

    fun update() {
        if (++frame % 60 == 0) {
            timeEvent.trigger(frame / 60)
        }
    }

    init {
        program.launch {
            delay(3000)
            doneWaiting.trigger(Unit)
        }
    }
}
```

Now lets use our `Blob` class in a new program that listens to its two events:

```
fun main() = application {
    program {
        val blob = Blob(this)
        extend {
            blob.update()
        }
        blob.timeEvent.listen {
            println("timeEvent triggered! $it")
        }
```

```
        blob.doneWaiting.listen {
            println("done waiting")
        }
    }
}
```

```
timeEvent triggered! 1
timeEvent triggered! 2
timeEvent triggered! 3
done waiting
timeEvent triggered! 4
...
```

Seems to work! right? There's one issue though: the `doneWaiting.listen` function does not run on the rendering thread. This would be the case for events triggered due to external causes (loading a file from the Internet and waiting for its completion or an event coming from a hardware input device).

This will become apparent when we fail to draw on our window:

```
blob.doneWaiting.listen {
    drawer.clear(ColorRGBa.WHITE) // <-- will not work
    println("done waiting")
}
```

The solution is simple though: when constructing the `Event`, we set the `postpone` argument to true:

```
val doneWaiting = Event<Unit>("done-waiting", postpone = true)
```

Now triggering the event no longer sends it immediately, but queues it. The second part of the solution is to actually deliver the queued events by calling `deliver()`.

```
doneWaiting.deliver()
```

It is essential to call `deliver()` from the rendering thread. Since `extend { }` executes in the rendering thread, and `extend` calls `update()`, we can let `update()` call `deliver()` and everything will work nicely.

This is the full program:

```
class Blob(program: Program) {
    val timeEvent = Event<Int>("time-event")
    val doneWaiting = Event<Unit>("done-waiting", postpone = true)

    private var frame = 0

    fun update() {
        if (++frame % 60 == 0) {
            timeEvent.trigger(frame / 60)
        }
        // Deliver any queued events
        doneWaiting.deliver()
    }
```

```
    init {
        program.launch {
            delay(3000)
            // Queue event outside the rendering thread
            doneWaiting.trigger(Unit)
        }
    }
}
```

```
fun main() = application {
    program {
        val blob = Blob(this)
        extend {
            blob.update()
        }
        blob.timeEvent.listen {
            println("timeEvent triggered! $it")
        }
        blob.doneWaiting.listen {
            // White flash when this event is received
            drawer.clear(ColorRGBa.WHITE)
            println("done waiting")
        }
    }
}
```

Mouse and keyboard events

Most user-input interaction in OPENRNDR manifests through events.

Mouse events

A simple demonstration of a listening for mouse button clicks looks as follows:

```
fun main() = application {
    program {
        mouse.buttonDown.listen {
            // -- it refers to a MouseEvent instance here
            println(it.position)
        }
    }
}
```

Every program has a `mouse` object that exposes events and mouse properties. In the example above we attach a listener to the `clicked` event. The `{}` block after `listen` is a short-hand notation for passing a function into `listen`.

There are some limitations to what the listener function can (or should) do. As a rule-of-thumb: don't draw in events. The result of drawing in listener functions is that it does not work. You are encouraged to use listener functions to change the state of your program.

Let's provide a commonly used pattern to deal with this limitation. The idea here is that we introduce variables in our program that are used to communicate between the listener function and the draw function.

```
fun main() = application {
    program {
        // -- a variable to keep of where we clicked
        // -- this is an "optional type" that can be set to null
        var drawPosition: Vector2? = null

        mouse.buttonDown.listen {
            drawPosition = it.position
        }

        extend {
            // -- check if the drawPosition is not null
            drawPosition?.let {
                drawer.circle(it.x, it.y, 100.0)
                // -- reset the drawPosition to null
                drawPosition = null
            }
        }
    }
}
```

Overview of mouse events

event	description	relevant MouseEvent properties
`moved`	generated when mouse has been moved	`position`, `modifiers`
`buttonDown`	generated when a button is pressed	`position`, `button`, `modifiers`
`buttonUp`	generated when a button is released	`position`, `button`, `modifiers`
`scrolled`	generated when mouse wheel is used	`position`, `rotation`
`clicked`	generated when a button has been pressed and released	`position`, `button`, `modifiers`
`dragged`	generated when mouse has been moved while a button is pressed	`position`, `button`, `modifiers`

Keyboard events

OPENRNDR provides two classes of keyboard events. The first are *key* events, which should be used to respond to the user pressing or releasing buttons on the keyboard. The second class are *character* events, which should be used for handling text input as they also deal with composed characters.

To use the *key* events one listens to `keyboard.keyDown` events and compares the `key` value. For example:

```
fun main() = application {
    program {
        keyboard.keyDown.listen {
            // -- it refers to a KeyEvent instance here
            // -- compare the key value to a predefined key constant
            if (it.key == KEY_ARROW_LEFT) {}
        }
    }
}
```

```
fun main() = application {
    program {
        keyboard.keyDown.listen {
            // -- it refers to a KeyEvent instance here
            // -- compare the name value against "a"
            if (it.name == "a") {}
        }
    }
}
```

To use the *character* events one listens to `keyboard.character` events which provide `character` values. For example:

```
fun main() = application {
    var input = ""
    program {
        keyboard.character.listen {
            input += it.character
        }
        keyboard.keyDown.listen {
            // -- it refers to a KeyEvent instance here
```

```
            // -- compare the key value to a predefined key constant
            if (it.key == KEY_BACKSPACE) {
                if (input.isNotEmpty()) {
                    input = input.substring(0, input.length - 1)
                }
            }
        }
    }
}
```

Querying key modifiers

Checking for modifiers can be done by checking if the desired modifier key is active in `modifiers`. In the example below we check if both shift and the left arrow key are pressed.

```
fun main() = application {
    program {
        keyboard.keyDown.listen {
            if (it.key == KEY_ARROW_LEFT && KeyModifier.SHIFT in it.modifiers) {}
        }
    }
}
```

Note that also mouse events come with modifiers and can be queried in a similar way.

Overview of keyboard events

event	description	relevant KeyboardEvent properties
`keyDown`	generated when a key is pressed	`key`, `modifiers`
`keyUp`	generated when a key is released	`key`, `modifiers`
`keyRepeat`	generated when a key is pressed and held	`key`, `modifiers`
`character`	generated when an input character is generated	`character`, `modifiers`

Overview of key constants

We provide constants for most of the functional/navigational keys.

constant	description
`KEY_SPACEBAR`	the spacebar key
`KEY_ESCAPE`	the escape key
`KEY_ENTER`	the enter key
`KEY_TAB`	the tab key
`KEY_BACKSPACE`	the backspace key
`KEY_INSERT`	the insert key
`KEY_DELETE`	the delete key
`KEY_ARROW_RIGHT`	the arrow right key
`KEY_ARROW_LEFT`	the arrow left key

constant	description
KEY_ARROW_UP	the arrow up key
KEY_ARROW_DOWN	the arrow down key
KEY_CAPSLOCK	the capslock key
KEY_PRINT_SCREEN	the print screen or prtscrn key
KEY_F1 ... KEY_F12	the F1 ... F12 function keys
KEY_LEFT_SHIFT	the left shift key
KEY_RIGHT_SHIFT	the right shift key

Event processing order

Mouse and keyboard events are buffered and processed before `draw()` is called. It is possible but not advised to perform drawing on event.

Program windows

An OPENRNDR program can listen to events generated by its window.

Window properties

Window title

The window title can be set and read using the `title` property.

```
fun main() = application {
    configure {
        title = "Lo and behold!"
    }

    program {
        println(window.title)
    }
}
```

Window position

The window position can be read and set using the `position` property.

```
fun main() = application {
    configure {
        position = IntVector2(30, 30)
    }
}
```

Window events

Window move event

This event is generated when the window was moved.

```
fun main() = application {
    program {
        window.moved.listen {
            println("the window was moved")
        }
    }
}
```

Window size event

This event is generated when the window was sized.

```
fun main() = application {
    program {
```

```
        window.sized.listen {
            println("the window was sized")
        }
    }
}
```

Window focus event

This event is generated when the program window gains focus.

```
fun main() = application {
    program {
        window.focused.listen {
            println("the window has gained focus")
        }
    }
}
```

Window unfocus event

This event is generated when the program window loses focus.

```
fun main() = application {
    program {
        window.unfocused.listen {
            println("the window has lost its focus")
        }
    }
}
```

File drops

OPENRNDR programs can listen to file drop events. File drop events are generated by a user dragging and releasing one or more file icons onto the program window.

```
fun main() = application {
    program {
        window.drop.listen {
            println("${it.files.size} files dropped at ${it.position}")
        }
    }
}
```

The following program displays `PNG`, `JPG` or `TIF` images dropped onto its window. If several files are dropped at once, it displays the first of them that is an image.

```
fun main() = application {
    program {
        // Create a dummy image
        var img = drawImage(16, 16) {}

        window.drop.listen { dropped ->
            val firstImage = dropped.files.firstOrNull {
                File(it).extension.lowercase() in listOf("png", "jpg", "tif")
            }

            if (firstImage != null) {
                // Call .destroy() to avoid leaking memory
                img.destroy()

                // Then load a new image into img
                img = loadImage(firstImage)
            }
        }
        extend {
            drawer.imageFit(img, drawer.bounds, fitMethod = FitMethod.Contain)
        }
    }
}
```

By using `drawer.imageFit` the image is centered and made to fit the available window size.

Clipboard

OPENRNDR programs can access the clipboard. Currently only text snippets can be read from and written to the clipboard.

Setting the clipboard content

```
fun main() = application {
    program {
        clipboard.contents = "this is the new clipboard content"
    }
}
```

Getting the clipboard content

Note that `clipboard.contents` is optional and its value can be `null`. The clipboard contents are reported `null` in case the clipboard contents are non-text or the clipboard is empty.

```
fun main() = application {
    program {
        clipboard.contents?.let {
            println("the clipboard contents: $it")
        }
    }
}
```

User Interfaces

This topic discusses the creation of graphical user interfaces in OPENRNDR applications.

The Panel library

The Panel library provides an HTML/CSS like user interface toolkit and is written using OPENRNDR.

Basic Usage

The easiest way to use Panel is to use it as a Program extension. When used as an extension all mouse and keyboard events are automatically handled and drawing of the user interface will take place after your program's `draw()` has been invoked.

To create a very simple user interface that consists of just a single button one would do the following:


```
fun main() = application {
    program {
        var color = ColorRGBa.GRAY.shade(0.250)
        extend(ControlManager()) {
            layout {
                button {
                    label = "click me"
```

```
                    // -- listen to the click event
                    clicked {
                        color = ColorRGBa(Math.random(), Math.random(), Math.random())
                    }
                }
            }
        }
        extend {
            drawer.clear(color)
        }
    }
}
```

Link to the full example

Style sheets

The Panel library borrows a lot of ideas from HTML/CSS based layout systems, one of those ideas is style sheets.

Style sheets can be used as shown in the following example in which a style sheet is used to color a button pink.


```
fun main() = application {
    program {
        extend(ControlManager()) {
            styleSheet(has type "button") {
                background = Color.RGBa(ColorRGBa.PINK)
                color = Color.RGBa(ColorRGBa.BLACK)
```

```
            }

            layout {
                button {
                    label = "click me"
                }
            }
        }
    }
}
```

Link to the full example

Selectors

The following example shows how to build and use complex selectors

```
fun main() = application {
    program {
        styleSheet(has class_ "control-bar") {
            descendant(has type "button") {
                width = 100.percent
            }

            child(has type "slider") {
                width = 100.percent

                and(has state "hover") {
                    background = Color.RGBa(ColorRGBa.RED)
                }
            }
        }
    }
}
```

Elements

The Panel library comes with a built-in set of elements with which user interfaces can be composed.

Element

`Element` is the base class from which all other elements derive. `Element` can be used directly but it is advised to use `Div` instead.

Div

The `Div` represents a rectangular area in which other elements can be placed. The `Div` element is the main ingredient in the creation of layouts. Divs are best created using the `div {}` builder.

```
fun main() = application {
    program {
        controlManager {
            layout {
                div("some-class-here", "another-class-here") {// -- children here
                }
```

```
            }
        }
    }
}
```

Button

An ordinary labelled button. The default width of buttons is set to Auto such that the width is determined by the label contents.


```
fun main() = application {
    program {
        extend(ControlManager()) {
            layout {
                button {
                    label = "Click me "
                    events.clicked.listen {// -- do something with the clicked event
                    }
                }
            }
        }
        extend {
            drawer.clear(ColorRGBa.GRAY.shade(0.250))
        }
    }
}
```

Link to the full example

Slider

A horizontal labelled slider control.

Properties

- `label : String` - the slider label
- `precision : Int` - the number of digits behind the point, set to 0 for an integer slider
- `range: Range` - the slider range, default is `Range(0.0, 10.0)`
- `value : Double` - the slider value

Events

- `valueChanged` - emitted when the slider value has changed


```
fun main() = application {
    program {
        extend(ControlManager()) {
            layout {
                slider {
                    label = "Slide me"
```

```
                    range = Range(0.0, 1.0)
                    value = 0.50
                    precision = 2
                    events.valueChanged.listen {
                        println("the new value is ${it.newValue}")
                    }
                }
            }
        }
    }
}
```

Link to the full example

Since the `value` is clamped to the current `range`, it is better to set the `range` before the `value` to avoid unexpected results.

ColorPickerButton

A button like control that slides out a HSV color picker when clicked

Properties

- `label : String` - the label on the button
- `value : ColorRGBa` - the currently picked color

Events

- `valueChanged` - emitted when a color is picked


```
fun main() = application {
    program {
        extend(ControlManager()) {
            layout {

                colorpickerButton {
                    label = "Pick a color"
                    color = ColorRGBa.PINK
                    events.valueChanged.listen {
                        println("the new color is ${it.color}")
                    }
                }
            }
        }
    }
}
```

Link to the full example

DropdownButton

A button like control that slides out a list of items when clicked.

Properties

- `label : String` - the label on the button
- `value : Item` - the currently picked item

Events

- `valueChanged` - emitted when an option is picked


```
fun main() = application {
    program {
        extend(ControlManager()) {
            layout {
                dropdownButton {
                    label = "Option"

                    item {
                        label = "Item 1"
                        events.picked.listen {
                            println("you picked item 1")
                        }
                    }

                    item {
                        label = "Item 2"
                        events.picked.listen {
                            println("you picked item 2")
                        }
                    }
                }
            }
        }
    }
}
```

Link to the full example

Extensions

edit on GitHub

TABLE OF CONTENTS

Extensions

Extensions add functionality to a Program. Extensions can be used to control how a program draws, setup keyboard and mouse bindings and much more.

Basic extension use

Here we demonstrate how to use an OPENRNDR extension. The extension that we use is the `Screenshots` extension, which, when the space bar is pressed will capture the application window's contents and save it to a timestamped file.

```
fun main() = application {
    program {
        // -- one time setup code goes here
        extend(Screenshots())
        extend {
        // -- drawing code goes here
        }
    }
}
```

Extension configuration

Some extensions have configurable options. They can be set using the configuring `extend` function as follows:

```
fun main() = application {
    program {
        extend(Screenshots()) {
            contentScale = 4.0
        }
    }
}
```

Extension order

The order in which calls to the `extend(...)` method appear in the code matters. `Screenshots` and `ScreenRecorder` should usually be placed before other extensions; otherwise, the content of the produced images or video files may be unexpected.

Built-in and contributed extensions

OPENRNDR provides a few built-in extensions to simplify common tasks. One is `Screenshots`, which is used to create screenshots of your programs. Another is `ScreenRecorder` which is used to write videos to files.

Next to the built-in extensions there is ORX, an extensive repository of provided and contributed OPENRNDR extensions and add-ons. If you work from `openrndr-template` you can easily add and remove extensions from your project by editing the `orxFeatures` property in `build.gradle.kts`.

The `Screenshots` extension

The Screenshots extension saves the current window as an image file under the `screenshots` folder in your project. The default key that triggers saving the image is the space bar key.

To setup the screenshots we only need to add one line to our program:

```
fun main() = application {
    program {
        extend(Screenshots())
        extend {// -- draw here
        }
    }
}
```

The extension provides many configurable options. This example demonstrates how to adjust some of them:

```
fun main() = application {
    program {
        extend(Screenshots()) {
            key = "s"
            folder = "work-in-progress"
            async = false
        }
        extend {// -- draw here
        }
    }
}
```

To discover other configurable options you can use the autocomplete feature (ctrl+space by default) in IntelliJ IDEA or explore its source code.

Writing video files

The `ScreenRecorder` extension

A simple way of writing your program's output to video is offered by the `ScreenRecorder` extension. The extension creates video files named after your Program class name plus the date. For example: `MyProgram-2018-04-11-11.31.03.mp4`. The video files are located in the `video/` directory in your project.

To setup the screen recorder do the following:

```
fun main() = application {
    program {
        extend(ScreenRecorder())
        extend {// -- whatever you draw here ends up in the video
        }
    }
}
```

Note: the `ffmpeg` command-line program is used for video output. If `ffmpeg` is not found in your system OPENRNDR will attempt to use an embedded version of it.

Toggle the `ScreenRecorder` on and off

By default the screen recorder extension adds video frames as long as the program runs. If you keep running the program again and again you will end up with a folder full of video files.

We can make it more flexible by letting the user start and pause the recorder, for instance by pressing the `v` key on the keyboard.

```
fun main() = application {
    program {
        // keep a reference to the recorder so we can start it and stop it.
        val recorder = ScreenRecorder().apply {
            outputToVideo = false
        }
        extend(recorder)
        extend {// -- draw things here
        }
        keyboard.keyDown.listen {
            when {
                it.key == KEY_ESCAPE -> program.application.exit()
                it.name == "v" -> {
                    recorder.outputToVideo = !recorder.outputToVideo
                    println(if (recorder.outputToVideo) "Recording" else "Paused")
                }
            }
        }
    }
}
```

Video formats

By default videos are recorded in h264 format with a `.mp4` file extension, but animated `gif` or `webp`, `png` or `tif` sequences, `prores` and `h265` are also available by enabling the orx-video-profiles extension.

Writing to video using render targets

It is also possible to produce video files without using the `ScreenRecorder` extension. This is needed for more advances uses, for example when the content we want in the video file is not visible but actually exists as a render target.

```
fun main() = application {
    program {
        val videoWriter = VideoWriter()
        videoWriter.size(width, height)
        videoWriter.output("output.mp4")
        videoWriter.start()

        val videoTarget = renderTarget(width, height) {
            colorBuffer()
            depthBuffer()
        }


        var frame = 0
        extend {
            drawer.isolatedWithTarget(videoTarget) {
                clear(ColorRGBa.BLACK)
                rectangle(40.0 + frame, 40.0, 100.0, 100.0)
            }

            videoWriter.frame(videoTarget.colorBuffer(0))
            drawer.image(videoTarget.colorBuffer(0))
            frame++
            if (frame == 100) {
                videoWriter.stop()
                application.exit()
            }
        }
    }
}
```

The `Camera2D` extension

This extension allows the user to easily pan, rotate and scale the view using a mouse.

- For panning, click and drag with the left mouse button.
- For scaling, use the mouse wheel.
- For rotating, click and drag with the right mouse button.

Using the `Camera2D` extension can be useful to find the right framing for a design before taking a screenshot.

You see, sometimes generative design involve randomness, and by using this extension you can choose what is up and what is down, and how much space you want between your creation and the edges of the window after the design has been created.

```
fun main() = application {
    program {
        backgroundColor = ColorRGBa.PINK
        extend(Camera2D())
        extend {
            drawer.rectangle(drawer.bounds.center, 200.0, 50.0)
            drawer.circle(drawer.bounds.position(0.3, 0.3), 100.0)
        }
    }
}
```

Find the simple source code of this extension in the orx repository.

The `NoClear` extension

Creative coding frameworks have two different defaults: either they clear the screen before each animation frame or they don't. OPENRNDR belongs to the first group.

Switching to "draw-without-clearing-the-screen" can be useful to produce complex designs with simple programs.

It is also how pen and paper seems to work: we add ink to the paper and the previous ink does not disappear.

Such behavior can easily be enabled by adding `extend(NoClear())` to our programs.
Here an example that draws circles at the current mouse position:

```
fun main() = application {
    program {
        backgroundColor = ColorRGBa.PINK
        extend(NoClear())
        extend {
            drawer.circle(mouse.position, 20.0)
        }
    }
}
```

Without `NoClear` only one circle would be visible at the current mouse location.

Find additional examples and the source code of orx-no-clear in GitHub.

Writing Program extensions

The extension interface

OPENRNDR provides a simple `Extension` interface with which default `Program` behaviour can be changed.

```
interface Extension {
    var enabled: Boolean
    fun setup(program: Program) {}
    fun beforeDraw(drawer: Drawer, program: Program) {}
    fun afterDraw(drawer: Drawer, program: Program) {}
}
```

In the `setup()` function the extension can setup itself and hook into program events.

The `beforeDraw()` function is called right before the program's `draw()` is executed.

The `afterDraw()` is called after the program's `draw()` is executed.

You can enabled and disable an extension by setting the `enabled` boolean on the extension.

A simple extension

Presented here is the outline of a simple extension that overlays the frames per second on top of the program output.

```
class FPSDisplay : Extension {
    override var enabled: Boolean = true

    var frames = 0
    var startTime: Double = 0.0

    override fun setup(program: Program) {
        startTime = program.seconds
    }

    override fun afterDraw(drawer: Drawer, program: Program) {
        frames++

        drawer.isolated {
            // -- set view and projections
            drawer.view = Matrix44.IDENTITY
            drawer.ortho()

            drawer.text("fps: ${frames / (program.seconds - startTime)}")
        }
    }
}
```

Using the `FPSDisplay` extension from your main program would then look like this:

```
fun main() = application {
    program {
        extend(FPSDisplay())
    }
}
```

Extension application order

In the scenario in which a program has 3 extensions installed like in the snippet below.

```
fun setup() {
    extend(ExtensionA())
    extend(ExtensionB())
    extend(ExtensionC())
}
```

This resulting application order of beforeDraw and afterDraw is then as follows:

```
extensionA.beforeDraw()
extensionB.beforeDraw()
extensionC.beforeDraw()

program.draw()

extensionC.afterDraw()
extensionB.afterDraw()
extensionA.afterDraw()
```

As you can see, the afterDraw() calls are applied in reverse order, this order was decided on to help with push/pop order of transforms and styles.

Animation

edit on GitHub

TABLE OF CONTENTS

Basic animation

From a programmer's perspective, animation works by continuously drawing shapes or images creating the illusion of movement. This is achieved through a draw loop, which is a function that runs repeatedly multiple times per second. Each iteration of the draw loop involves clearing the screen and redrawing all elements in their new positions. By making small changes to the positions, rotations or sizes of shapes in each frame, we can create complex animations and movements.

In this example we draw a circle moving horizontally. The `x` variable increases from 0 up to `width` in a loop, thanks to the `%` modulo operation. `width` is 640 by default.

```
fun main() = application {
    program {
        extend {
            val pixelsPerSecond = 100.0
            val x = (seconds * pixelsPerSecond) % width
            val y = height * 0.5
            drawer.circle(x, y, 100.0)
        }
    }
}
```

By modifying the speed while the animation progresses we can enhance the animation, making it feel more natural and fluid and mimicking the way objects in the real world accelerate or decelerate as they move.

Let's start with a one second long animation loop:

```
fun main() = application {
    program {
        extend {
            val x = (seconds % 1.0) * width
            val y = height * 0.5
            drawer.circle(x, y, 100.0)
        }
    }
}
```

Note that the expression `(seconds % 1.0)` is always a value between 0.0 and 1.0. What happens if we change that expression to `(seconds % 1.0).pow(3.0)` instead? The result is still a value between 0.0 and 1.0, but by raising the expression to the power of 3.0 we obtain more values closer to 0.0 than to 1.0. We no longer have a linear animation, but an `ease in` effect: start slowly and accelerate.

To simplify calculating acceleration and deceleration curves for animations we can use easing functions to specify the rate of change of a parameter over time.

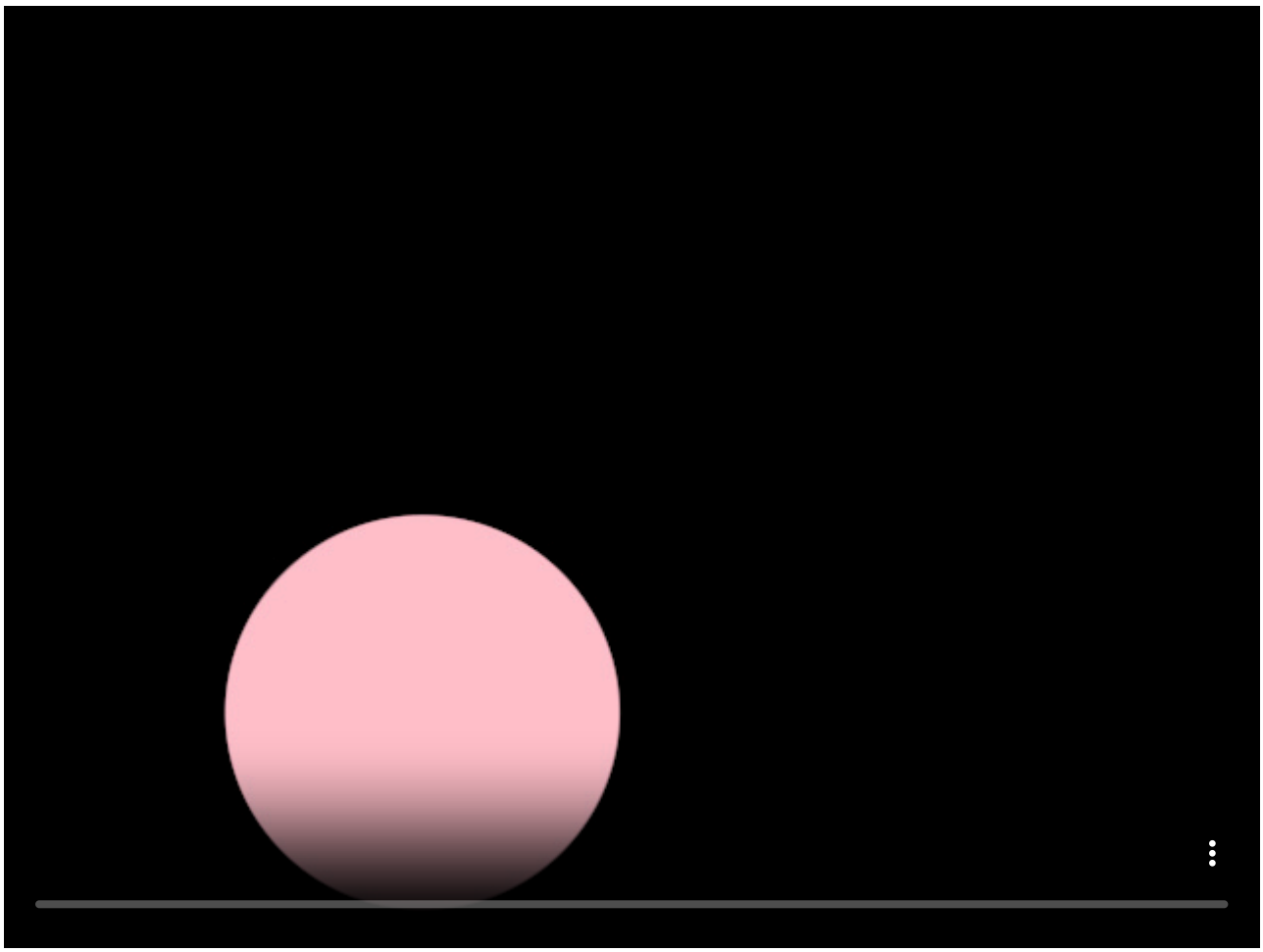
Fortunately OPENRNDR provides orx-easing to help with this task.

Interactive animations

Animatable

The OPENRNDR `Animatable` class provides animation logic.

Displayed below is a very simple animation setup in which we animate a circle from left to right. We do this by animating the `x` property of our animation object.

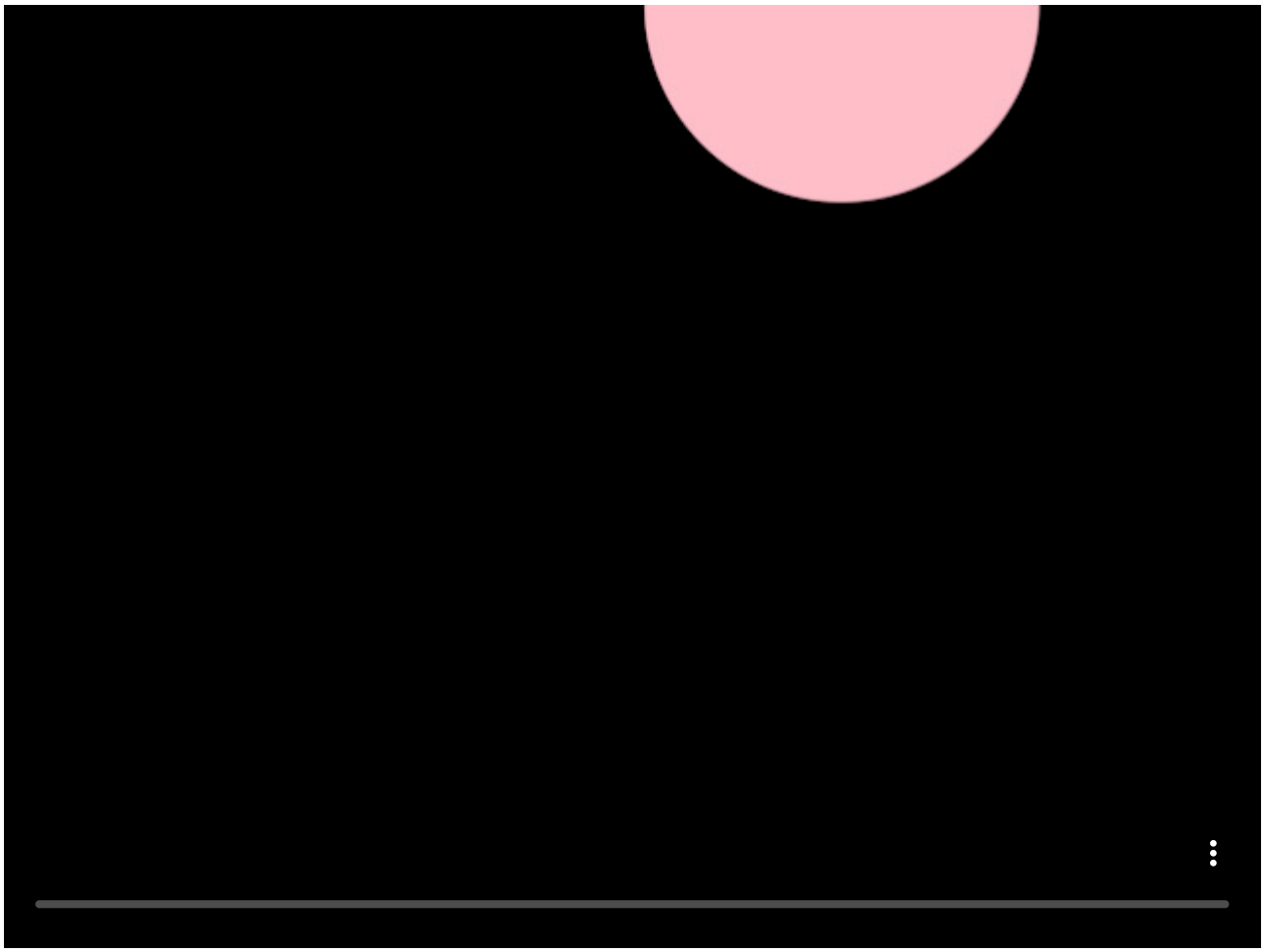


```
fun main() = application {
    program {
        // -- create an animation object
        val animation = object : Animatable() {
            var x = 0.0
            var y = 360.0
        }

        animation.apply {
            ::x.animate(width.toDouble(), 5000)
        }

        extend {
            animation.updateAnimation()
```

```
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.circle(animation.x, animation.y, 100.0)
        }
    }
}
```

Link to the full example

By using `.complete()` we can create sequences of property animations.

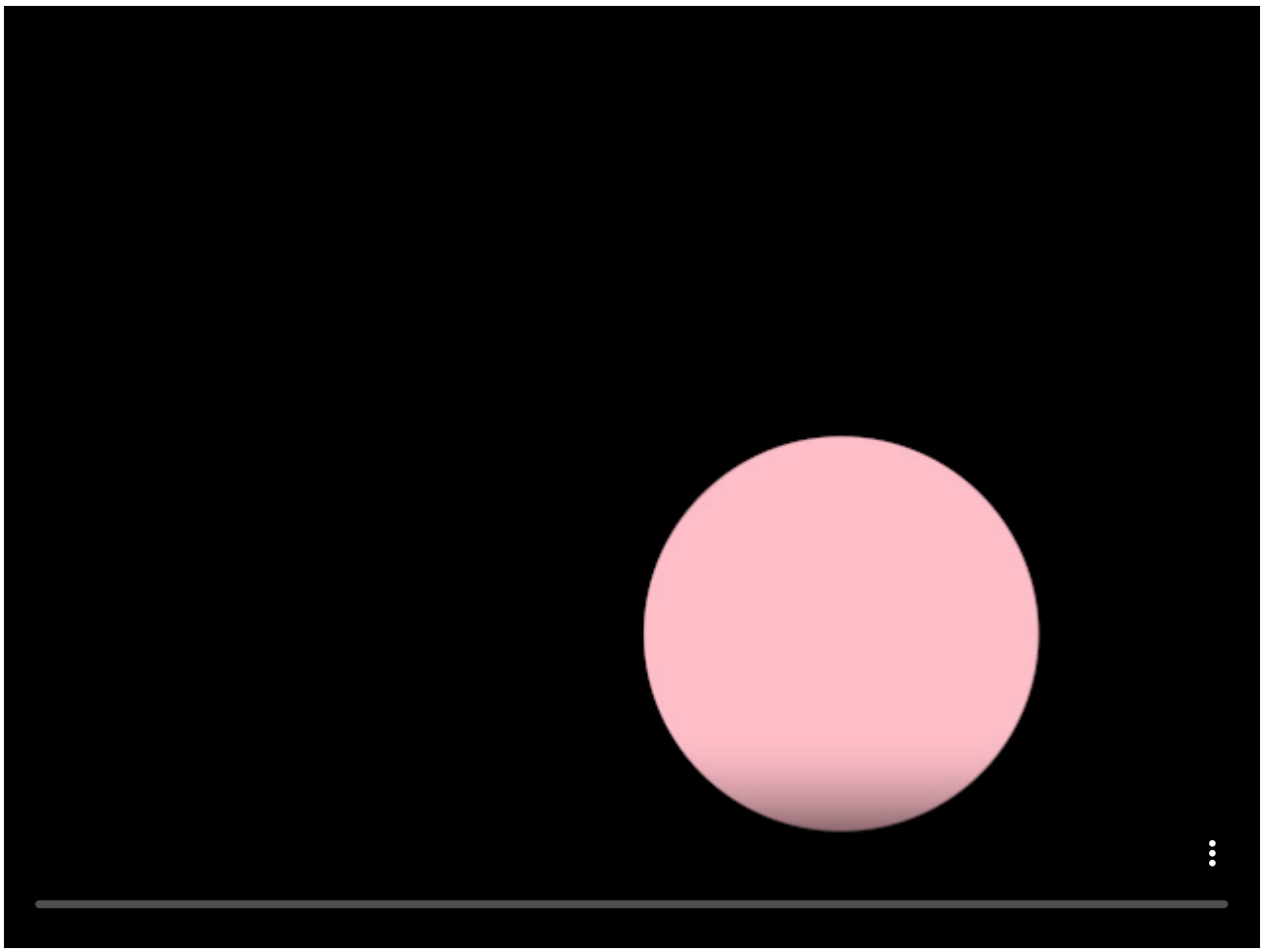


```
fun main() = application {
    program {
        val animation = object : Animatable() {
            var x = 0.0
            var y = 0.0
        }

        animation.apply {
            ::x.animate(width.toDouble(), 5000)
            ::x.complete()
            ::y.animate(height.toDouble(), 5000)
        }

        extend {
            animation.updateAnimation()
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
```

```
            drawer.circle(animation.x, animation.y, 100.0)
        }
    }
}
```

Link to the full example

If we leave out that `::x.complete()` line we will see that animations for `x` and `y` run simultaneously.



```
fun main() = application {
    program {
        val animation = object : Animatable() {
            var x = 0.0
            var y = 0.0
        }

        animation.apply {
            ::x.animate(width.toDouble(), 5000)
            ::y.animate(height.toDouble(), 5000)
        }

        extend {
            animation.updateAnimation()
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.circle(animation.x, animation.y, 100.0)
        }
```

```
    }
}
```

Link to the full example

For those wondering where that `::x.animate()` notation comes from, those are Kotlin's property references.

Easing

A simple trick for making animations less stiff is to specify an easing.

To demonstrate we take one of the previously shown animations and add easings.

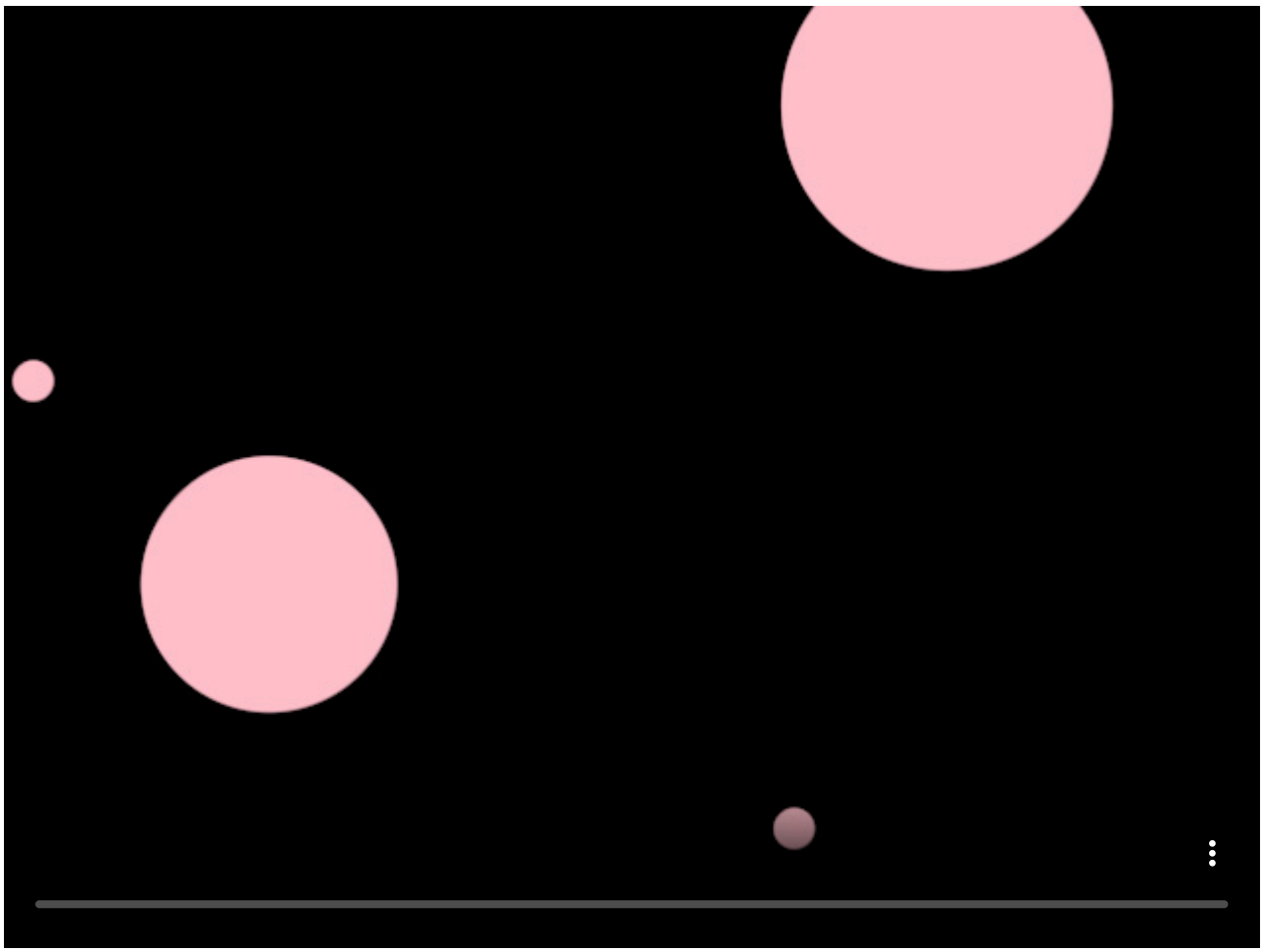
Available Easings

```
fun main() = application {
    program {
        val animation = object : Animatable() {
            var x = 0.0
            var y = 0.0
        }

        animation.apply {
            ::x.animate(width.toDouble(), 5000, Easing.CubicInOut)
            ::y.animate(height.toDouble(), 5000, Easing.CubicInOut)
        }

        extend {
            animation.updateAnimation()
```

```
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.circle(animation.x, animation.y, 100.0)
        }
    }
}
```

Link to the full example

Behavioral animation



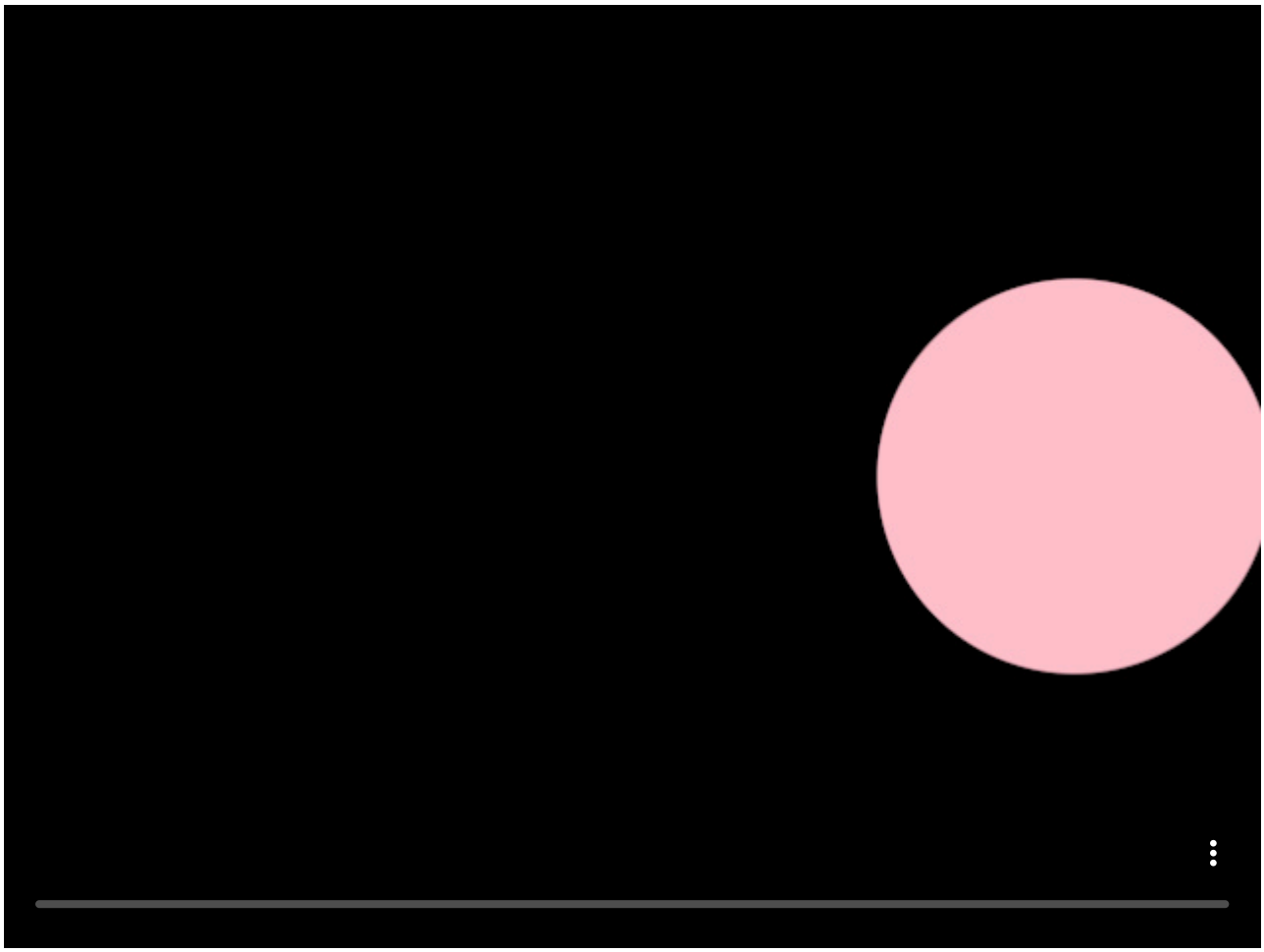
```
fun main() = application {
    program {
        class AnimatedCircle : Animatable() {
            var x = 0.0
            var y = 0.0
            var radius = 100.0
            var latch = 0.0

            fun shrink() {
                // -- first stop any running animations for the radius property
                ::radius.cancel()
                ::radius.animate(10.0, 200, Easing.CubicInOut)
            }

            fun grow() {
                ::radius.cancel()
```

```
                ::radius.animate(Double.uniform(60.0, 90.0), 200, Easing.CubicInOut)
            }

            fun jump() {
                ::x.cancel()
                ::y.cancel()
                ::x.animate(Double.uniform(0.0, width.toDouble()), 400, Easing.CubicInOut)
                ::y.animate(Double.uniform(0.0, height.toDouble()), 400, Easing.CubicInOut)
            }

            fun update() {
                updateAnimation()
                if (!::latch.hasAnimations) {
                    val duration = Double.uniform(100.0, 700.0).toLong()
                    ::latch.animate(1.0, duration).completed.listen {
                        val action = listOf(::shrink, ::grow, ::jump).random()
                        action()
                    }
                }
            }
        }

        val animatedCircles = List(5) {
            AnimatedCircle()
        }
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            for (ac in animatedCircles) {
                ac.update()
                drawer.circle(ac.x, ac.y, ac.radius)
            }
        }
    }
}
```

Link to the full example

Looping animations

While `Animatable` doesn't provide explicit support for looping animations. They can be achieved through the following pattern:



```
fun main() = application {
    val animation = object : Animatable() {
        var x: Double = 0.0
    }

    program {
        extend {
            animation.updateAnimation()
            if (!animation.hasAnimations()) {
                animation.apply {
                    ::x.animate(width.toDouble(), 1000, Easing.CubicInOut)
                    ::x.complete()
                    ::x.animate(0.0, 1000, Easing.CubicInOut)
                    ::x.complete()
                }
            }
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.circle(animation.x, height / 2.0, 100.0)
        }
    }
}
```

Link to the full example

Animatable properties

Thus far we have only worked with `Double` properties in our animations. However, animation is not limited to Doubles.

Any property that is a `LinearType` can be animated through Animatable.

```
fun main() = application {
    program {
        val animation = object : Animatable() {
            var color = ColorRGBa.WHITE
            var position = Vector2.ZERO
        }
        animation.apply {
            ::color.animate(ColorRGBa.PINK, 5000)
            ::position.animate(Vector2(width.toDouble(), height.toDouble()), 5000)
        }
    }
}
```

File Input / Output

Loading files

Load a text file into a `String`

```
val fileContent = File("/path/to/file.txt").readText()
```

Load a text file into a `List<String>`

```
val lines = File("/path/to/file.txt").readLines()
```

Load binary content into a `ByteArray`

```
val bytes = File("/path/to/file.txt").readBytes()
```

For loading large text files one can use `File.bufferedReader()`.

Saving files

Save a `String`

```
File("/path/to/file.txt").writeText("Hello")
```

Save a `ByteArray`

```
File("/path/to/file.txt").writeBytes(myByteArray)
```

For saving large text files one can use `File.bufferedWriter()`.

JSON

The `kotlinx-serialization` library makes it possible to convert JSON files to Kotlin objects and back. The library is enabled by default in recent versions of the openrndr-template.

Loading / converting JSON to an object

```
// An object matching the JSON file.
// Notice how optional properties are nullable.
@Serializable
data class Entry(var time: Double, var easing: String, var rotx: Double?, var roty: Double?, var x: Double?, var y:
Double?, var scale: Double?, var jitter: Double?)

// from disk
// val json = File("/path/to/a/file.json").readText()
```

```
// from a string
val json = """
        [
            {
                "time": 0.0,
                "easing": "cubic-in-out",
                "rotx": 0.0,
                "roty": 30.0,
                "x": 0.0,
                "y": 0.0,
                "scale": 1.0,
                "jitter": 0.0
            },
            {
                "time": 1.0,
                "easing": "cubic-in-out",
                "rotx": 0.0,
                "x": 20.0,
                "y": 0.0,
                "scale": 1.0
            },
            {
                "time": 1.3,
                "easing": "cubic-in-out"
            }
        ]
        """.trimIndent()

fun main() = application {
    program {
        val entries = Json.decodeFromString<List<Entry>>(json)

        entries.forEachIndexed { i, entry ->
            println(i)
            println(entry)
        }
    }
}
```

Saving an object as a JSON file

```
fun main() = application {
    program {
        val points = List(20) {
            Vector2.uniform(drawer.bounds)
        }
        val json = Json.encodeToString(points)
        File("data/points.json").writeText(json)
    }
}
```

CSV

The kotlin-csv library enables loading and saving CSV files. To enable the library:

1. Find the `build.gradle.kts` file in your openrndr-template.
2. Uncomment the line `implementation(libs.csv)`.
3. Reload gradle.

Load a CSV file

```
fun main() = application {
    program {
        // read from `String`
        val csvData = "a,b,c\nd,e,f"
        val rowsA = csvReader().readAll(csvData)

        // read from `java.io.File`
        val file = File("test.csv")
        val rowsB = csvReader().readAll(file)
    }
}
```

Save a CSV file

```
fun main() = application {
    program {
        val rows = listOf(listOf("a", "b", "c"), listOf("d", "e", "f"))
        csvWriter().writeAll(rows, "data/test.csv")
    }
}
```

XML / HTML

The jsoup library enables loading and parsing XML and HTML files. To enable the library:

1. Find the `build.gradle.kts` file in your openrndr-template.
2. Uncomment the line `implementation(libs.jsoup)`.
3. Reload gradle.

Get Wikipedia home page and parse news headlines

```
fun main() = application {
    program {
        val doc = Jsoup.connect("https://en.wikipedia.org/").get()
        println(doc.title())

        val newsHeadlines = doc.select("#mp-itn b a")
        newsHeadlines.forEach {
            println(it.attr("title"))
            println(it.absUrl("href"))
        }
```

```
    }
}
```

edit on GitHub

ORX (OPENRNDR Extras)

The ORX project is a library of tools that can be used in OPENRNDR based programs. ORX contains implementations of datastructures, algorithms and utilities for (mostly) computational graphics. You can find the ORX source code and additional documentation in the ORX repository

Using ORX

Using the OPENRNDR extras is a matter of adding an additional Maven repository and selected dependencies to your Gradle project.

The openrndr-template project makes this simple as the repositories are already set up and one only has to

1. Open the `build.gradle.kts` file
2. Uncomment the desired module names under `orxFeatures`
3. Save
4. Reimport Gradle projects: if using IDEA choose 🔄 *Sync All Gradle Projects* from the Gradle panel.

TABLE OF CONTENTS

orx-noise

A collection of noise generator functions. Source and extra documentation can be found in the orx-noise sourcetree.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-noise` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Uniformly distributed random values

The library provides extension methods for `Double`, `Vector2`, `Vector3`, `Vector4` to create random vectors easily. To create scalars and vectors with uniformly distributed noise you use the `uniform` extension function.

```
val d1 = Double.uniform(0.0, 640.0)
val v2 = Vector2.uniform(0.0, 640.0)
val v3 = Vector3.uniform(0.0, 640.0)
val v4 = Vector4.uniform(0.0, 640.0)
```

To create multiple samples of noise one uses the `uniforms` function.

```
val v2 = Vector2.uniforms(100, Vector2(0.0, 0.0), Vector2(640.0, 640.0))
val v3 = Vector3.uniforms(100, Vector3(0.0, 0.0, 0.0), Vector3(640.0, 640.0, 640.0))
```

The `Random` class can also be used to generate Double numbers and vector, but also booleans and integers.

```
// Boolean
val b = Random.bool(probability = 0.2)

// Int
val i1 = Random.int(0, 640)
val i2 = Random.int0(640)

// Double
val d2 = Random.double(0.0, 640.0)
val d3 = Random.double0(640.0)

// Vectors
val v2 = Random.vector2(0.0, 640.0)
val v3 = Random.vector3(0.0, 640.0)
val v4 = Random.vector4(0.0, 640.0)
```

Perlin, Value and Simplex noise

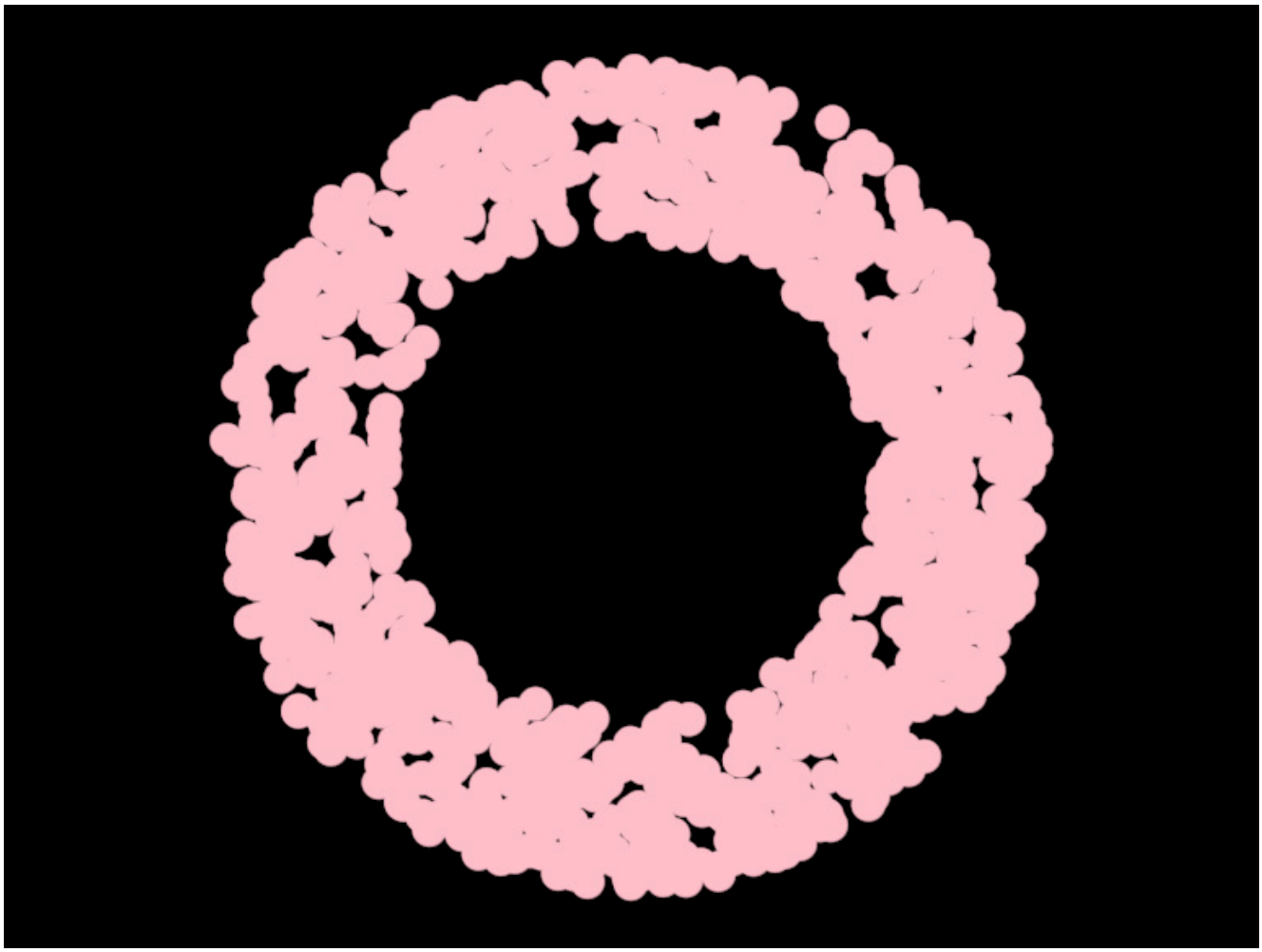
`Random.perlin()` and `Random.value()` accept 2D and 3D arguments. `Random.simplex()` up to 4D. They all return a `Double`. Some examples:

```
// Test vectors to use
val v2 = Vector2(0.1, 0.2)
```

```
val v4 = Vector4(0.1, 0.2, 0.3, 0.4)

// Now generate random values
val d1 = Random.perlin(0.1, 0.2)
val d2 = Random.perlin(v2)
val d3 = Random.value(0.1, 0.2, 0.3)
val d4 = Random.simplex(v4)
```

Uniform ring noise

```
val v2 = Vector2.uniformRing(0.0, 300.0)
val v3 = Vector3.uniformRing(0.0, 300.0)
val v4 = Vector4.uniformRing(0.0, 300.0)
```

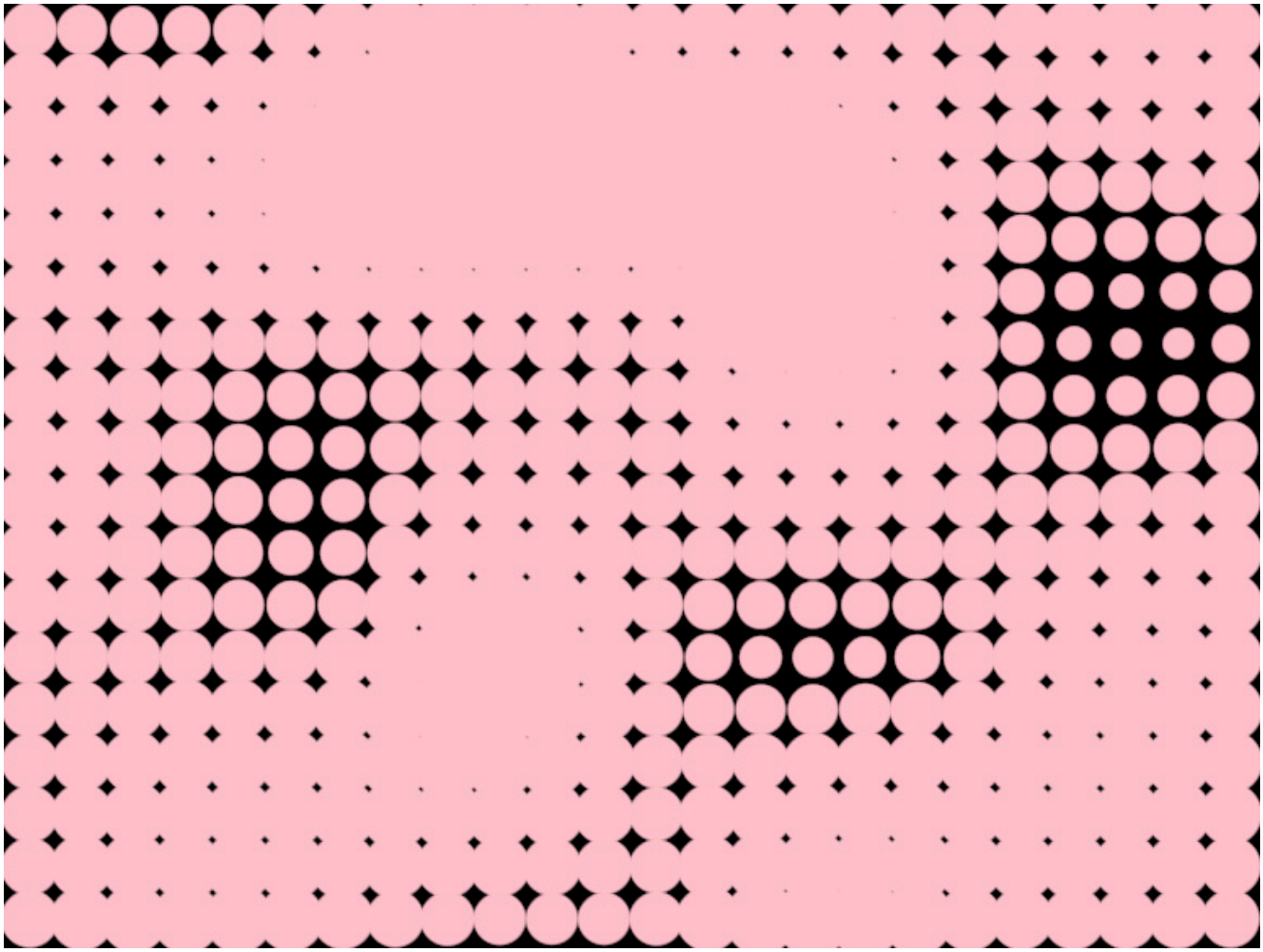


```
fun main() = application {
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            drawer.translate(width / 2.0, height / 2.00)
            for (i in 0 until 1000) {
                drawer.circle(Vector2.uniformRing(150.0, 250.0), 10.0)
            }
        }
```

```
    }
}
```

Link to the full example

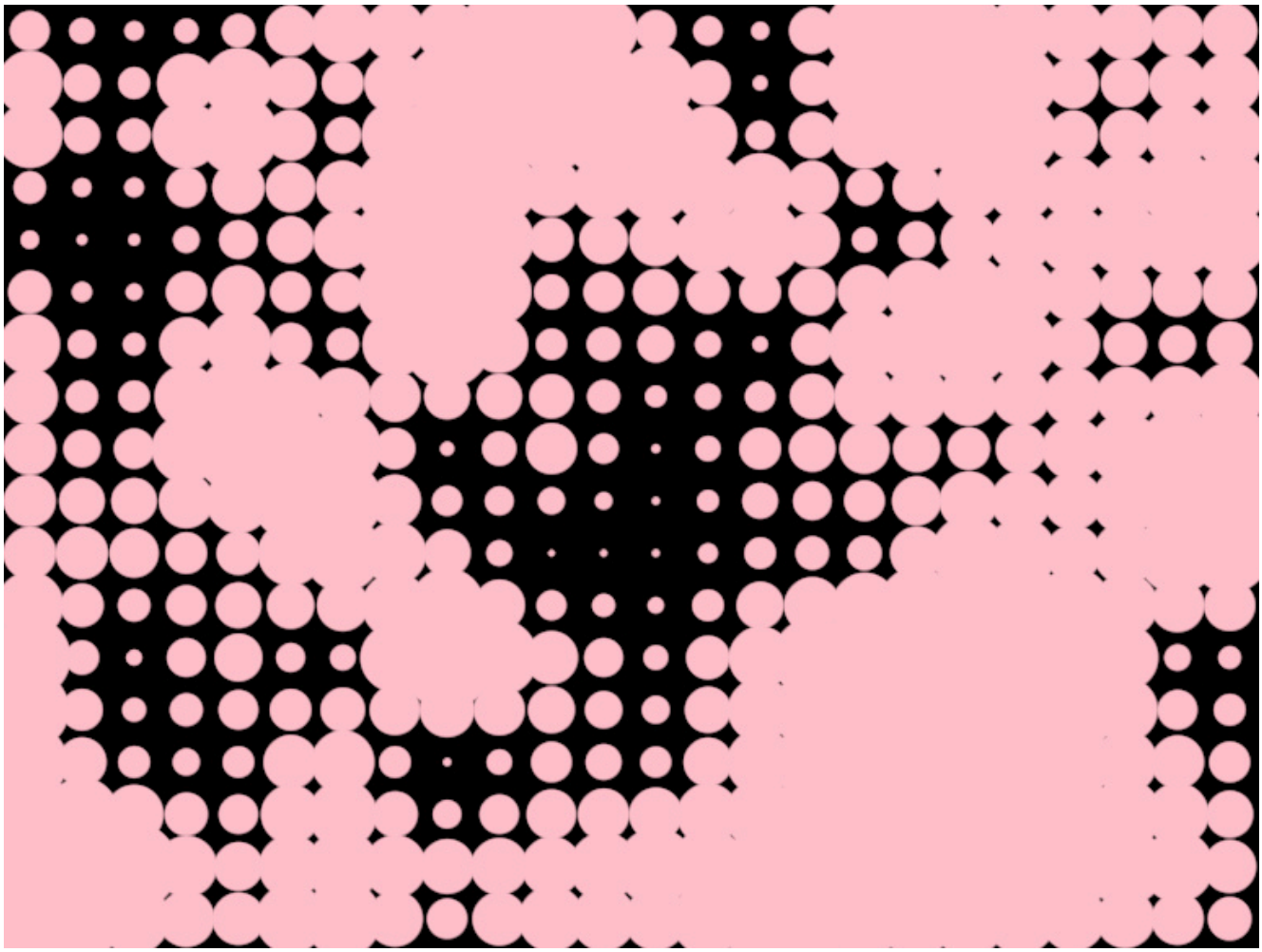
Perlin noise



```
fun main() = application {
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            val scale = 0.005
            for (y in 16 until height step 32) {
                for (x in 16 until width step 32) {
                    val radius = perlinLinear(100, x * scale, y * scale) * 16.0 + 16.0
                    drawer.circle(x * 1.0, y * 1.0, radius)
                }
            }
        }
    }
}
```

Link to the full example

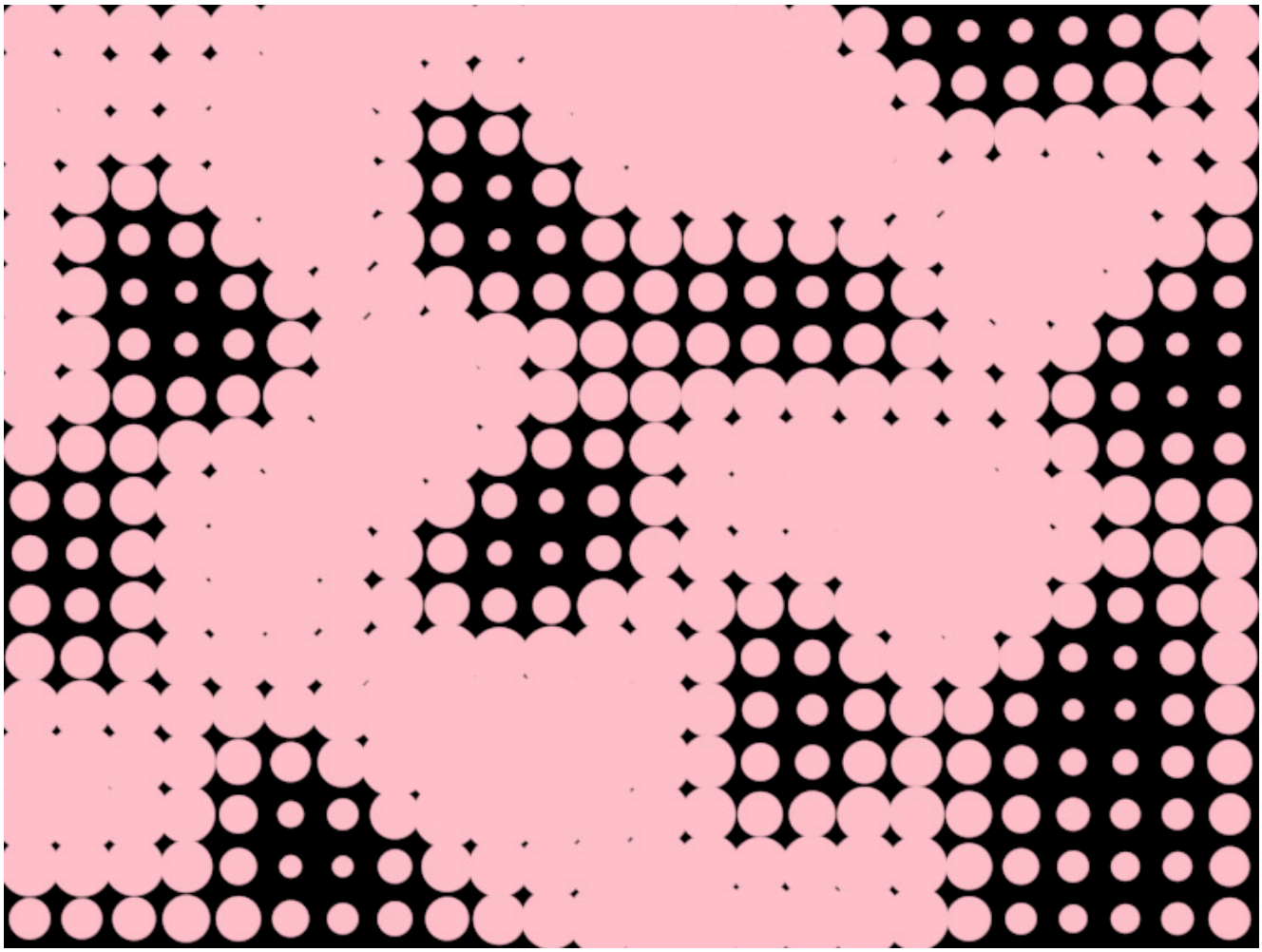
Value noise



```
fun main() = application {
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            val scale = 0.0150
            for (y in 16 until height step 32) {
                for (x in 16 until width step 32) {
                    val radius = valueLinear(100, x * scale, y * scale) * 16.0 + 16.0
                    drawer.circle(x * 1.0, y * 1.0, radius)
                }
            }
        }
    }
}
```

Link to the full example

Simplex noise



```
fun main() = application {
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            val scale = 0.004
            for (y in 16 until height step 32) {
                for (x in 16 until width step 32) {
                    val radius = simplex(100, x * scale, y * scale) * 16.0 + 16.0
                    drawer.circle(x * 1.0, y * 1.0, radius)
                }
            }
        }
    }
}
```

Link to the full example

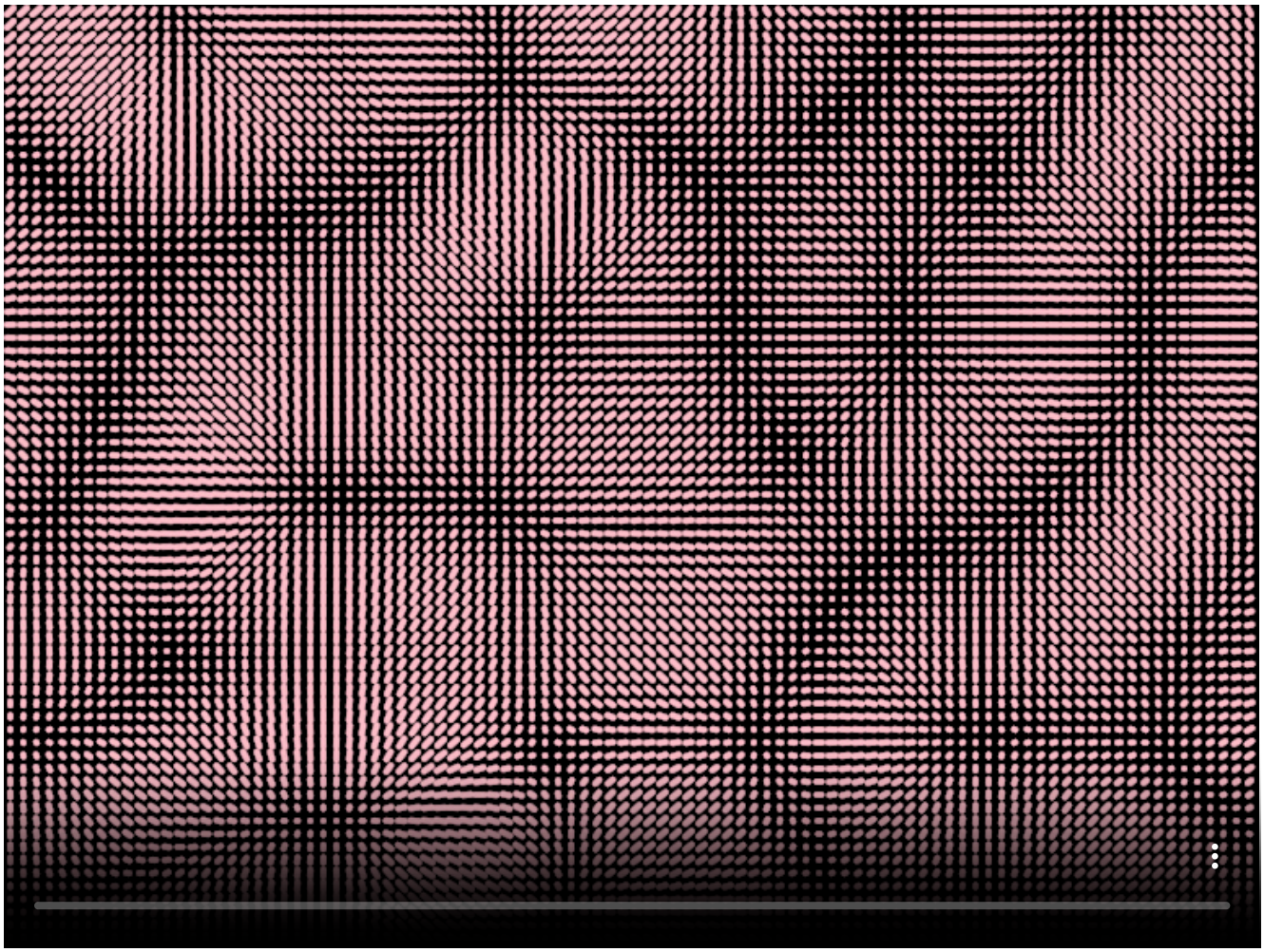
Fractal/FBM noise

```
fun main() = application {
    program {
        extend {
            drawer.fill = ColorRGBa.PINK
            drawer.stroke = null
            val s = 0.0080
            val t = seconds
            for (y in 4 until height step 8) {
                for (x in 4 until width step 8) {
                    val radius = when {
                        t < 3.0 -> abs(fbm(100, x * s, y * s, t, ::perlinLinear)) * 16.0
                        t < 6.0 -> billow(100, x * s, y * s, t, ::perlinLinear) * 2.0
                        else -> rigid(100, x * s, y * s, t, ::perlinLinear) * 16.0
                    }
                    drawer.circle(x * 1.0, y * 1.0, radius)
                }
            }
        }
    }
}
```

Link to the full example

Noise gradients

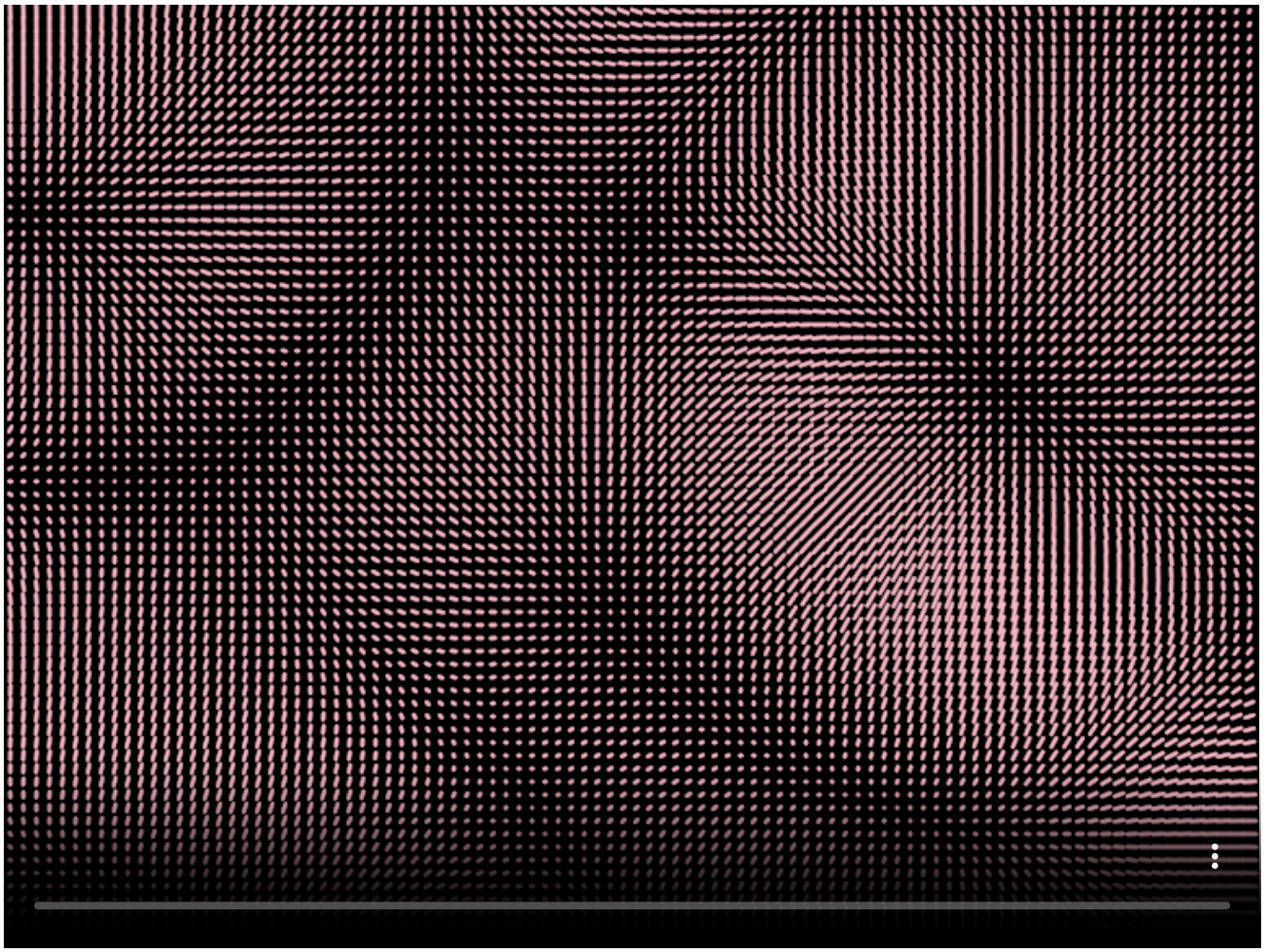
Noise functions have evaluable gradients, a direction to where the value of the function increases the fastest. The `gradient1D`, `gradient2D`, `gradient3D` and `gradient4D` functions can be used to estimate gradients for noise functions.



```
fun main() = application {
    program {
        extend {
            drawer.fill = null
            drawer.stroke = ColorRGBa.PINK
            drawer.lineCap = LineCap.ROUND
            drawer.strokeWeight = 3.0
            val t = seconds
            for (y in 4 until height step 8) {
                for (x in 4 until width step 8) {
                    val g = gradient3D(::perlinQuintic, 100, x * 0.005, y * 0.005, t, 0.0005).xy
                    drawer.lineSegment(Vector2(x * 1.0, y * 1.0) - g * 2.0, Vector2(x * 1.0, y * 1.0) + g * 2.0)
                }
            }
        }
    }
}
```

Link to the full example

Gradients can also be calculated for the fbm, rigid and billow versions of the noise functions. However, we first have to create a function that can be used by the gradient estimator. For this `.fbm()`, `.billow()`, and `.rigid()` can be used (which works through partial application).



```
fun main() = application {
    program {
        val noise = simplex3D.fbm(octaves = 3)
        extend {
            drawer.fill = null
            drawer.stroke = ColorRGBa.PINK
            drawer.lineCap = LineCap.ROUND
            drawer.strokeWeight = 1.5
            val t = seconds
            for (y in 4 until height step 8) {
                for (x in 4 until width step 8) {
                    val g = gradient3D(noise, 100, x * 0.002, y * 0.002, t, 0.002).xy
                    drawer.lineSegment(Vector2(x * 1.0, y * 1.0) - g * 1.0, Vector2(x * 1.0, y * 1.0) + g * 1.0)
                }
            }
        }
    }
}
```

Link to the full example

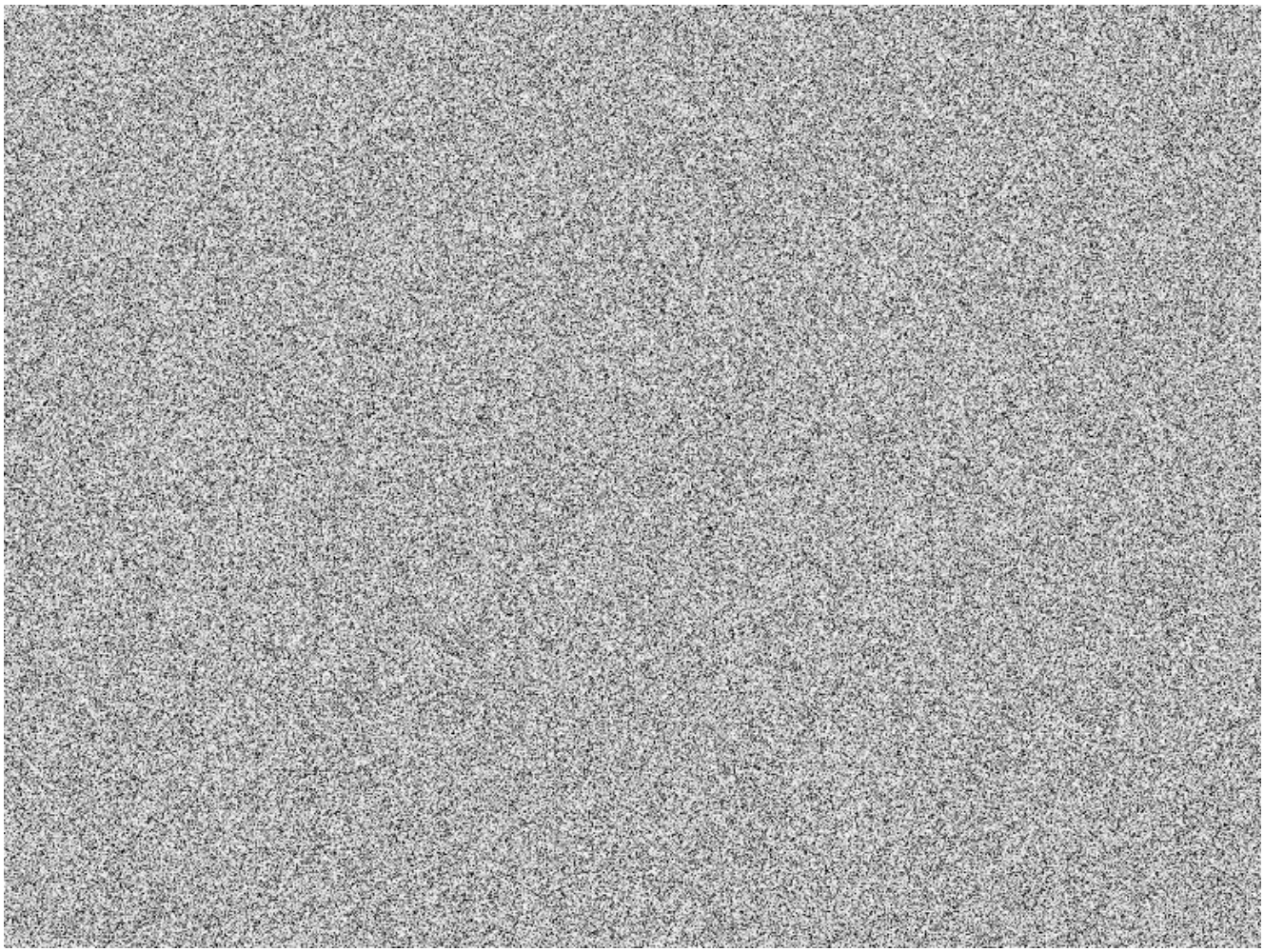
Noise filters

The library contains a number of Filters with which noise image can be generated efficiently on the GPU.

Hash noise

A white-noise-like noise generator.

Parameter	Default value	Description
`seed`	`0.0`	Noise seed
`gain`	`Vector4(1.0, 1.0, 1.0, 0.0)`	Noise gain per channel
`bias`	`Vector4(0.0, 0.0, 0.0, 1.0)`	Value to add to the generated noise
`monochrome`	`true`	Outputs monochrome noise if true
`premultipliedAlpha`	`true`	Outputs premultiplied alpha if true



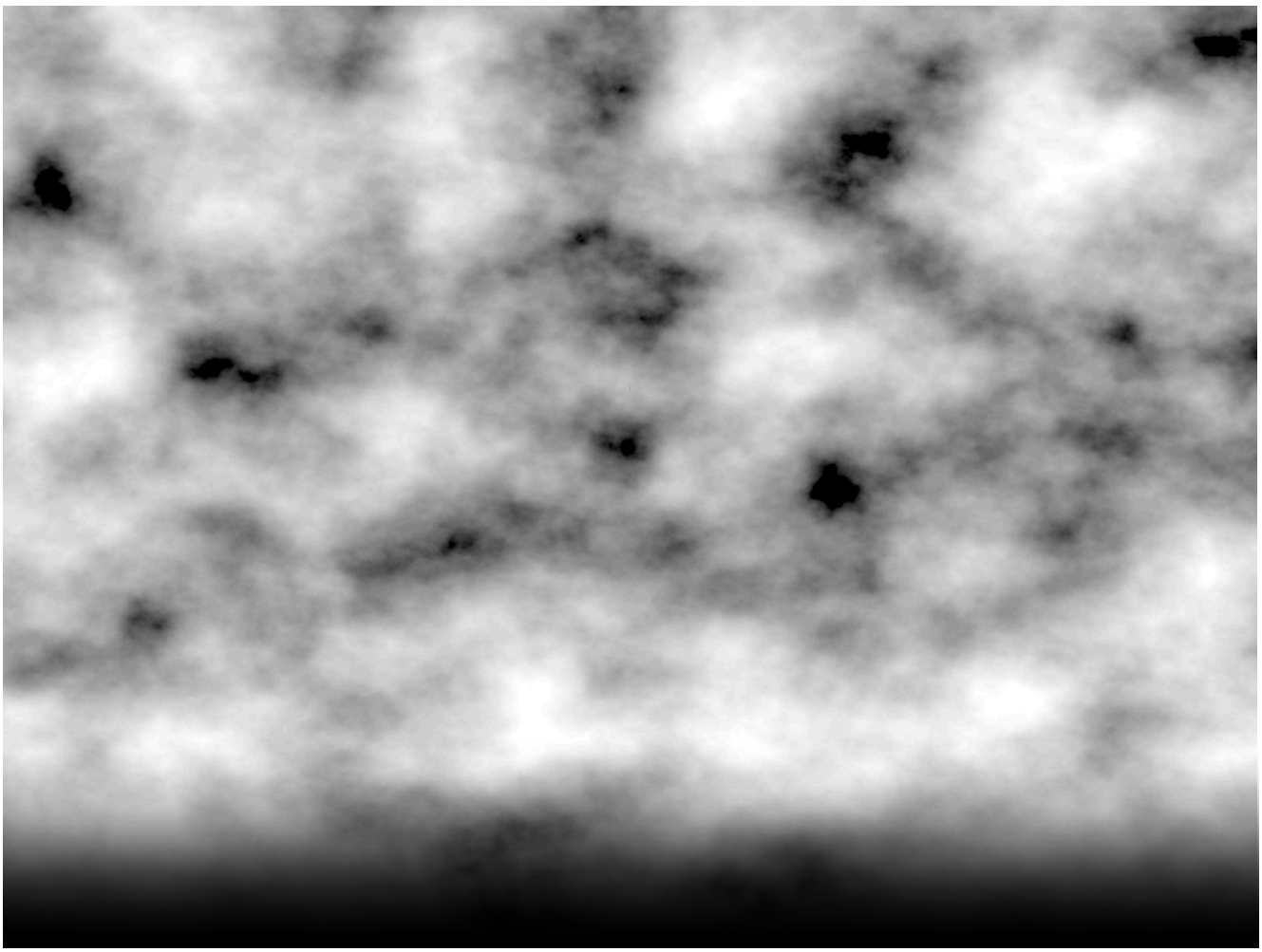
```
fun main() = application {
    program {
        val cb = colorBuffer(width, height)
        val hn = HashNoise()
        extend {
            hn.seed = seconds
            hn.apply(emptyArray(), cb)
            drawer.image(cb)
        }
    }
}
```

Link to the full example

3D Simplex noise filter

The `SimplexNoise3D` filter is based on Ken Perlin's improvement over Perlin noise, but with fewer directional artifacts and, in higher dimensions, a lower computational overhead.

Parameter	Default value	Description
`seed`	`Vector3(0.0, 0.0, 0.0)`	Noise seed / offset
`scale`	`Vector3(1.0, 1.0, 1.0)`	The noise scale at the first octave
`octaves`	`4`	The number of octaves
`gain`	`Vector4(0.5, 0.5, 0.5, 0.5)`	Noise gain per channel per octave
`decay`	`Vector4(0.5, 0.5, 0.5, 0.5)`	Noise decay per channel per octave
`bias`	`Vector4(0.5, 0.5, 0.5, 0.5)`	Value to add to the generated noise
`lacunarity`	`Vector4(2.0, 2.0, 2.0, 2.0)`	Multiplication of noise scale per octave
`premultipliedAlpha`	`true`	Outputs premultiplied alpha if true



```
fun main() = application {
    program {
        val cb = colorBuffer(width, height)
        val sn = SimplexNoise3D()
        extend {
            sn.seed = Vector3(0.0, 0.0, seconds * 0.1)
            sn.scale = Vector3.ONE * 2.0
            sn.octaves = 8
```

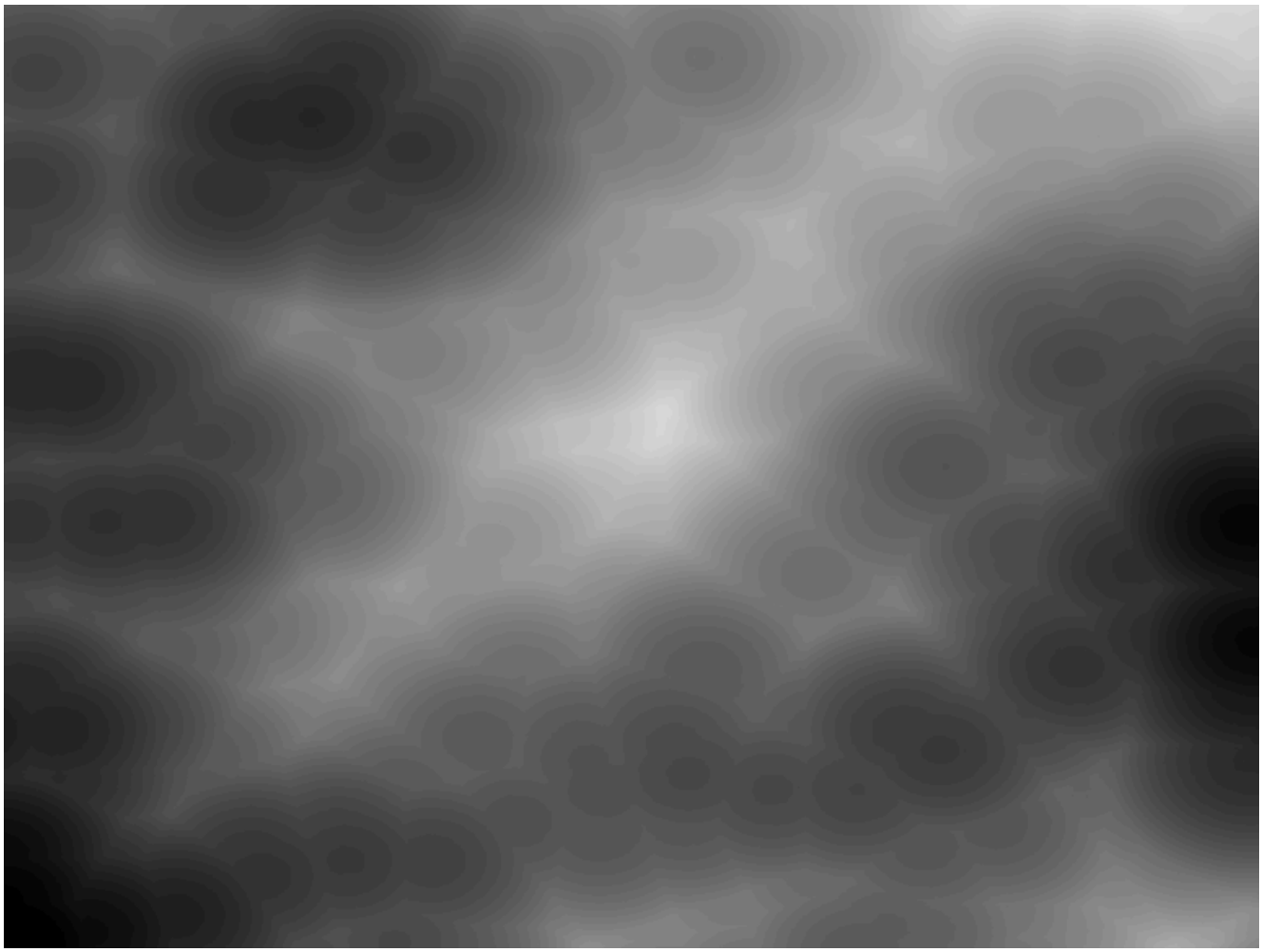
```
            sn.premultipliedAlpha = false
            sn.apply(emptyArray(), cb)
            drawer.image(cb)
        }
    }
}
```

Link to the full example

Cell noise

A cell, Worley or Voronoi noise generator

Parameter	Default value	Description
`seed`	`Vector2(0.0, 0.0)`	Noise seed / offset
`scale`	`Vector2(1.0, 1.0)`	The noise scale at the first octave
`octaves`	`4`	The number of octaves
`gain`	`Vector4(1.0, 1.0, 1.0, 0.0)`	Noise gain per channel per octave
`decay`	`Vector4(0.5, 0.5, 0.5, 0.5)`	Noise decay per channel per octave
`bias`	`Vector4(0.0, 0.0, 0.0, 1.0)`	Value to add to the generated noise
`lacunarity`	`Vector4(2.0, 2.0, 2.0, 2.0)`	Multiplication of noise scale per octave
`premultipliedAlpha`	`true`	Outputs premultiplied alpha if true



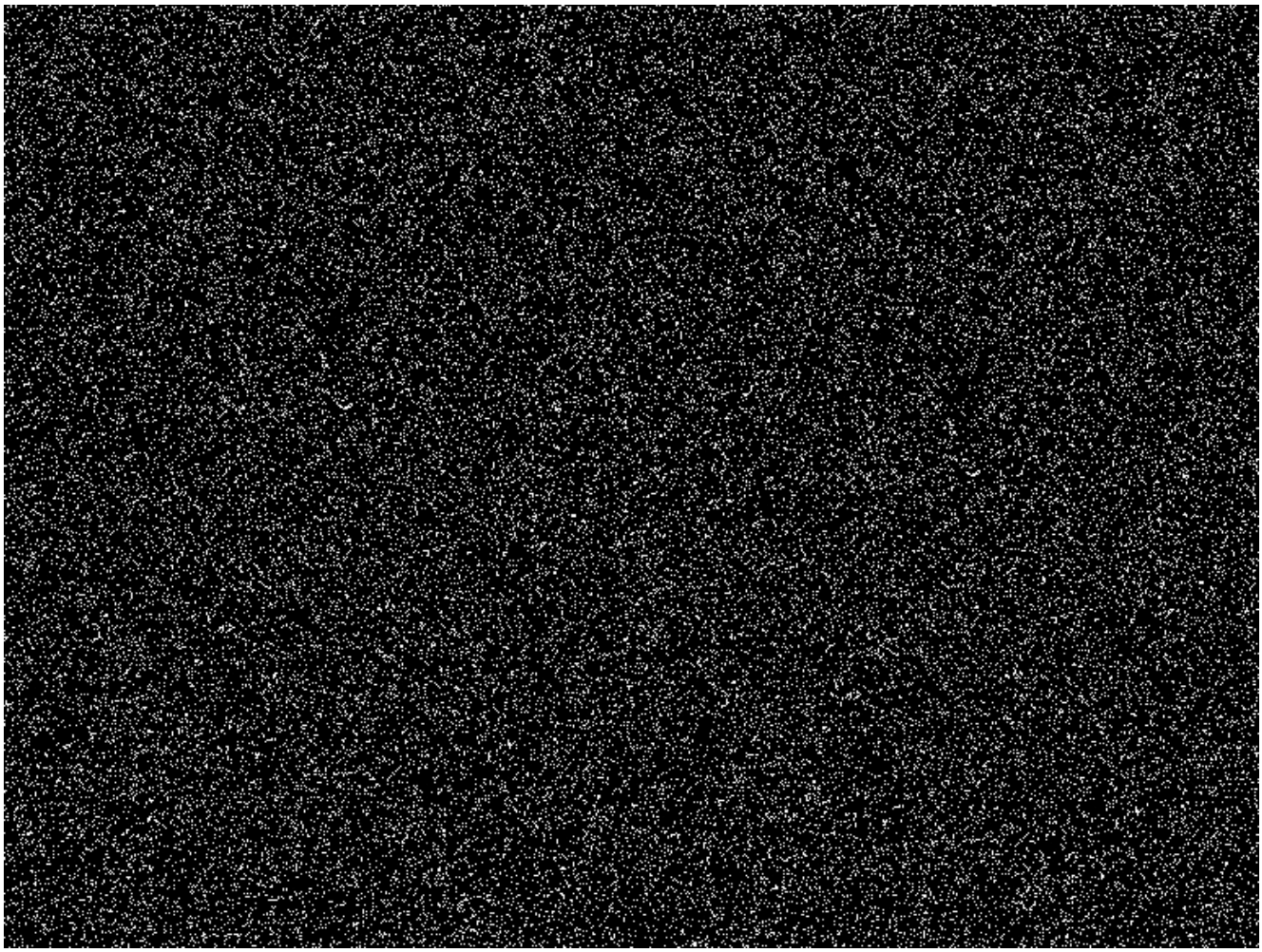
```
fun main() = application {
    program {
        val cb = colorBuffer(width, height)
        val cn = CellNoise()
        extend {
            cn.octaves = 4
            cn.apply(emptyArray(), cb)
            drawer.image(cb)
        }
    }
}
```

Link to the full example

Speckle noise

A speckle noise generator.

Parameter	Default value	Description
`color`	`ColorRGBa.WHITE`	Speckle color
`density`	`1.0`	Speckle density
`seed`	`0.0`	Noise seed
`noise`	`0.0`	Speckle noisiness
`premultipliedAlpha`	`true`	Outputs premultiplied alpha if true



```
fun main() = application {
    program {
        val cb = colorBuffer(width, height)
        val sn = SpeckleNoise()
        extend {
            sn.seed = seconds
            sn.apply(emptyArray(), cb)
            drawer.image(cb)
        }
    }
}
```

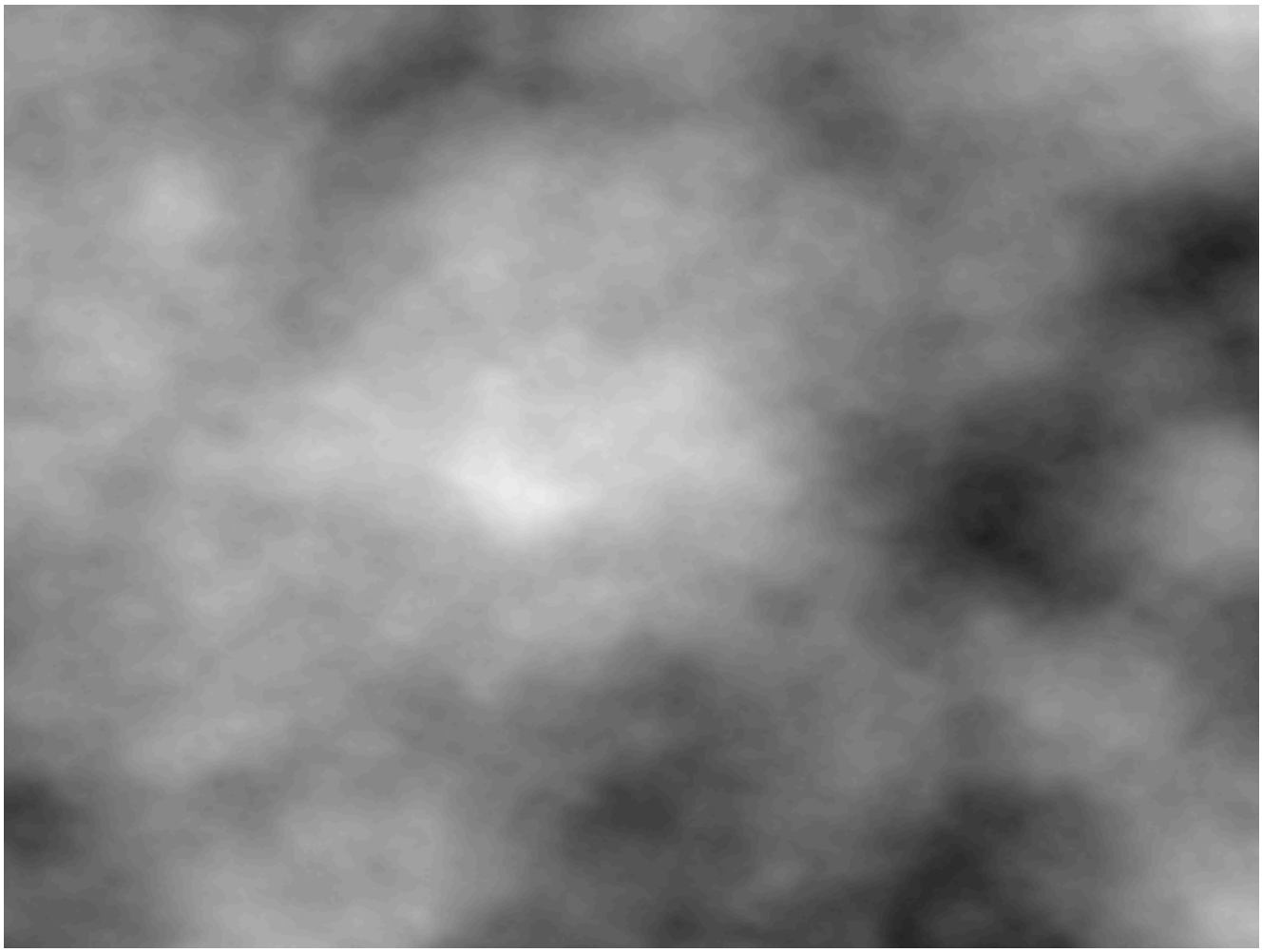
Link to the full example

Value noise

The `ValueNoise` filter generates a simple fractal noise. Value noise is a computationally cheap form of creating 'smooth noise' by interpolating random values on a lattice.

Parameter	Default value	Description
`seed`	`Vector2(0.0, 0.0)`	Noise seed / offset
`scale`	`Vector2(1.0, 1.0)`	The noise scale at the first octave
`octaves`	`4`	The number of octaves
`gain`	`Vector4(1.0, 1.0, 1.0, 0.0)`	Noise gain per channel per octave

Parameter	Default value	Description
`decay`	`Vector4(0.5, 0.5, 0.5, 0.5)`	Noise decay per channel per octave
`bias`	`Vector4(0.0, 0.0, 0.0, 1.0)`	Value to add to the generated noise
`lacunarity`	`Vector4(2.0, 2.0, 2.0, 2.0)`	Multiplication of noise scale per octave
`premultipliedAlpha`	`true`	Outputs premultiplied alpha if true



```
fun main() = application {
    program {
        val cb = colorBuffer(width, height)
        val vn = ValueNoise()
        extend {
            vn.scale = Vector2.ONE * 4.0
            vn.gain = Vector4.ONE * 0.5
            vn.octaves = 8
            vn.apply(emptyArray(), cb)
            drawer.image(cb)
        }
    }
}
```

Link to the full example

orx-kinect

Provides Kinect support (only Kinect version 1 at the moment and only depth camera). Source and extra documentation can be found in the orx sourcetree

Note: the support is split into several modules:

- `orx-kinect-common`
- `orx-kinect-v1`
- `orx-kinect-v1-demo`
- `orx-kinect-v1-natives-linux-x64`
- `orx-kinect-v1-natives-macos`
- `orx-kinect-v1-natives-windows`

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-kinect-v1` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Using the Kinect depth camera

```
fun main() = application {
    configure {
        // default resolution of the Kinect v1 depth camera
        width = 640
        height = 480
    }
    program {
        val kinect = Kinect1()
        val device = kinect.openDevice()
        device.depthCamera.flipH = true // to make a mirror
        device.depthCamera.enabled = true
        extend(kinect)
        extend {
            drawer.image(device.depthCamera.currentFrame)
        }
    }
}
```

Note: depth values are mapped into `0-1` range and stored on a `ColorBuffer` containing only RED color channel.

Mirroring depth camera image

```
kinect.depthCamera.mirror = true
```

Using multiple Kinects

The `kinects.startDevice()` can be supplied with device number (`0` by default):

```
fun main() = application {
    configure {
        width = 640 * 2
        height = 480
    }
    program {
        val kinect = Kinect1()
        val depthCamera1 = kinect.openDevice(0).depthCamera
        val depthCamera2 = kinect.openDevice(1).depthCamera
        depthCamera1.enabled = true
        depthCamera1.flipH = true
        depthCamera2.enabled = true
        depthCamera2.flipH = true
        extend(kinect)
        extend {
            drawer.image(depthCamera1.currentFrame)
            drawer.image(depthCamera2.currentFrame, depthCamera1.resolution.x.toDouble(), 0.0)
        }
    }
}
```

Reacting only to the latest frame from the Kinect camera

Kinect is producing 30 frames per second, while screen refresh rates are usually higher. Usually, if the data from the depth camera is processed, it is desired to react to the latest Kinect frame only once:

```
kinect.depthCamera.getLatestFrame()?.let { frame ->
    myFilter.apply(frame, outputColorBuffer)
}
```

Using color map filters

Raw kinect depth data might be visualized in several ways, the following filters are included:

- `DepthToGrayscaleMapper`
- `DepthToColorsZucconi6Mapper` - Colors of natural light dispersion by Alan Zucconi
- `DepthToColorsTurboMapper` - Turbo, An Improved Rainbow Colormap for Visualization by Google

Find a runnable example here.

Executing native freenect commands

This kinect support is built on top of the freenect library. Even though the access to freenect is abstracted, it is still possible to execute low level freenect commands in the native API.

Midi controllers with orx-midi

The `orx-midi` library provides a simple interface to interact with MIDI controllers.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-midi` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Listing MIDI controllers

To connect to a MIDI controller you will need its device name which can be discovered by calling the `listMidiDevices()` function.

```
fun main() = application {
    program {
        listMidiDevices().forEach {
            println("name: '${it.name}', vendor: '${it.vendor}', receiver:${it.receive}, transmitter:${it.transmit}")
        }
    }
}
```

Connecting to a MIDI controller

Once you have the controller name you can use `openMidiDevice` to connect to the MIDI controller. For example to use a Behringer BCR2000 controller on a Ubuntu system we can use the following.

```
fun main() = application {
    program {
        val controller = openMidiDevice("BCR2000")
    }
}
```

Tip: request `BCR2000` instead of `BCR2000 [hw:2,0,0]` so your program continues to work after plugging your controller into a different USB port. Caveat: do specify the full name if connecting multiple controllers of the same brand and model.

Listening to the controller

Once connected to a controller we can start listening to the MIDI events it sends out. The orx-midi library supports six types of MIDI events.

```
fun main() = application {
    program {
        val controller = openMidiDevice("BCR2000 [hw:2,0,0]")

        controller.controlChanged.listen {
            println("[control change] channel: ${it.channel}, control: ${it.control}, value: ${it.value}")
        }
```

```
        controller.noteOn.listen {
            println("[note on] channel: ${it.channel}, key: ${it.note}, velocity: ${it.velocity}")
        }
        controller.noteOff.listen {
            println("[note off] channel: ${it.channel}, key: ${it.note},")
        }
        controller.channelPressure.listen {
            println("[channel pressure] channel: ${it.channel}, pressure: ${it.pressure}")
        }
        controller.pitchBend.listen {
            println("[pitch bend] channel: ${it.channel}, pitch: ${it.pitchBend}")
        }
        controller.programChanged.listen {
            println("[program change] channel: ${it.channel}, program: ${it.program}")
        }
    }
}
```

Talking to the controller

MIDI controllers can often react to data received from software. A common use case with MIDI controllers with endless rotary encoders is setting up initial values for the encoders when the program launches. Those values are then reflected in LED lights or in a display in the controller.

```
fun main() = application {
    program {
        val controller = openMidiDevice("BCR2000")

        // send a control change
        controller.controlChange(channel = 1, control = 3, value = 42)

        // send a program change
        controller.programChange(channel = 2, program = 5)

        // send a note on event
        controller.noteOn(channel = 3, key = 60, velocity = 100)

        // send a note off event
        controller.noteOn(channel = 3, key = 60, velocity = 0)

        // send a pitch bend event
        controller.pitchBend(channel = 4, 50)

        // send a channel pressure event
        controller.channelPressure(channel = 4, 100)
    }
}
```

MIDI console

For debugging purposes one can visualize all the MIDI events by using the `MidiConsole`.

```
fun main() = application {
    program {
        val controller = openMidiDevice("Launchpad [hw:4,0,0]")

        extend(MidiConsole()) {
            register(controller)
        }
    }
}
```

Variable binding

One can easily bind MIDI controller inputs like knobs and sliders to program variables. In the following example 7 inputs control the radius, position and color of a circle.

Note that `ColorParameter` binds four consecutive inputs (red, green, blue and alpha).

More about Parameter Annotations.

```
fun main() = application {
    program {
        val controller = openMidiDevice("Launchpad [hw:4,0,0]")

        val settings = object {
            @DoubleParameter("radius", 0.0, 100.0)
            var radius = 0.0

            @DoubleParameter("x", -100.0, 100.0)
            var x = 0.0

            @DoubleParameter("y", -100.0, 100.0)
            var y = 0.0

            @ColorParameter("fill")
            var color = ColorRGBa.WHITE
        }

        bindMidiControl(settings::radius, controller, channel = 0, control = 1)
        bindMidiControl(settings::x, controller, 0, 2)
        bindMidiControl(settings::y, controller, 0, 3)
        bindMidiControl(settings::color, controller, 0, 4)

        extend {
            drawer.fill = settings.color
            drawer.circle(drawer.bounds.center + Vector2(settings.x, settings.y), settings.radius)
        }
    }
}
```

Handling OSC messages with orx-osc

The `orx-osc` osc provides a simple interface to interact with OSC hosts and clients.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-osc` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Listening to OSC messages

To listen to OSC messages we need to start an OSC server and use `listen` function to install listeners

```
fun main() = application {
    program {
        val osc = OSC()
        osc.listen("/live/track/2") { addr, it ->
            // -- get the first value
            val firstValue = it[0] as Float
        }
        extend {
        }
    }
}
```

Note that `.listen()` accepts wildcard characters like `*` and `?`. For instance to listen to addresses containing two words we can use `osc.listen("/*/*")`. Find out more about **Pattern Matching** in the OSC specification.

Sending OSC messages

```
fun main() = application {
    program {
        val osc = OSC()

        extend {
            osc.send("/some/address", listOf(1.0f, 2.0f))
        }
    }
}
```

Specifying IP address and ports

The default IP address for OSC is `localhost` and the in and out ports are both set to `57110` by default. One can specify different values like this:

```
val osc = OSC(InetAddress.getByName("192.168.0.105"), portIn = 10000, portOut = 12000)
```

orx-fx Filters

The orx-fx library contains many filters that can be readily used. See the chapter Filters and post-processing for instructions on using them.

A (more-or-less) complete listing of the effects in orx-fx is maintained in the repository's README.md

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-fx` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Effect Index

In this index we demonstrate selected filters, this is in no way a complete overview of what orx-fx offers.

Blur effects

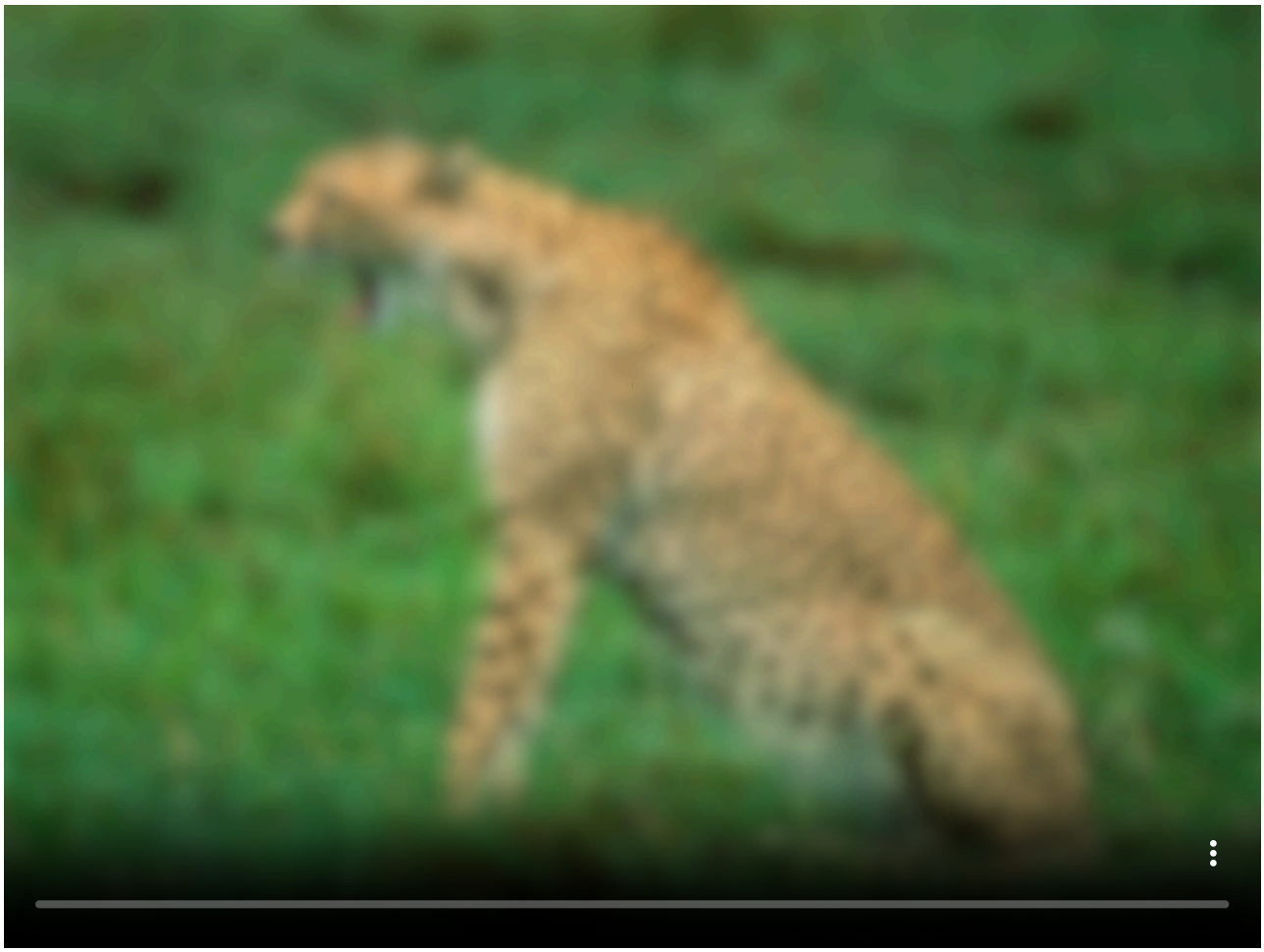
BOXBLUR

```
fun main() = application {
    program {
        // -- load a source image
        val image = loadImage("data/images/cheeta.jpg")
```

```
        // -- create a filter
        val blur = BoxBlur()

        // -- create a colorBuffer where to store the result
        val blurred = colorBuffer(image.width, image.height)

        extend {
            // -- configure the filter
            blur.window = (cos(seconds * 2) * 4.0 + 5.0).toInt()

            // -- filter.apply(source, target)
            blur.apply(image, blurred)

            // -- draw the result
            drawer.image(blurred)
        }
    }
}
```

Link to the full example

APPROXIMATEGAUSSIANBLUR



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val blurred = colorBuffer(image.width, image.height)
        val blur = ApproximateGaussianBlur()
        extend {
```

```
            blur.window = 25
            blur.sigma = cos(seconds * 2) * 10.0 + 10.1
            blur.apply(image, blurred)
            drawer.image(blurred)
        }
    }
}
```

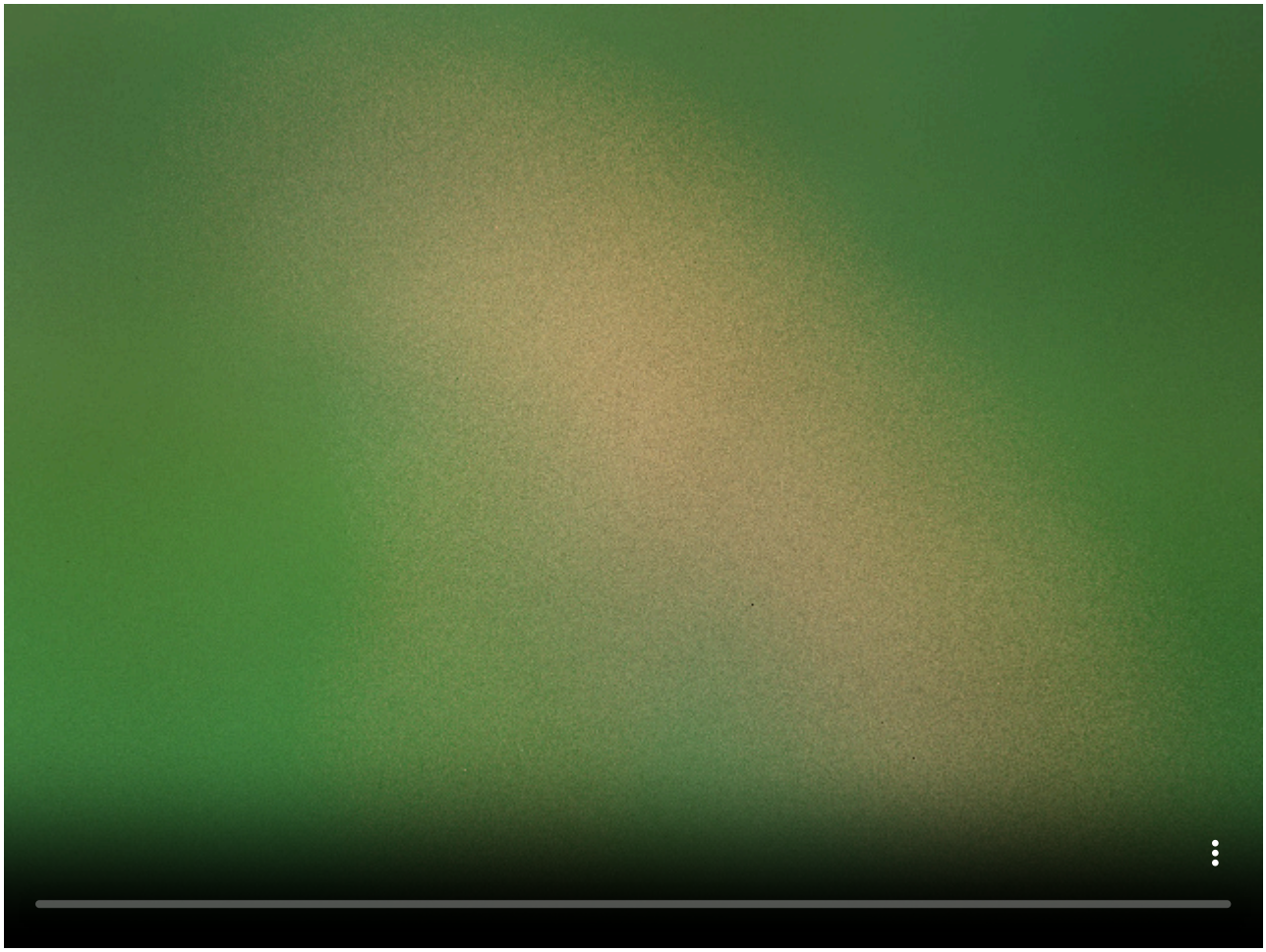
Link to the full example

GAUSSIANBLOOM

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val blurred = colorBuffer(image.width, image.height)
        val bloom = GaussianBloom()
        extend {
            bloom.window = 5
            bloom.sigma = 3.0
            bloom.gain = cos(seconds * 2) * 2.0 + 2.0
            bloom.apply(image, blurred)
            drawer.image(blurred)
        }
    }
}
```

Link to the full example

HASHBLUR



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val blurred = colorBuffer(image.width, image.height)
        val blur = HashBlur()
        extend {
            blur.samples = 50
            blur.radius = cos(seconds * 2) * 25.0 + 25.0
            blur.apply(image, blurred)
            drawer.image(blurred)
        }
    }
}
```

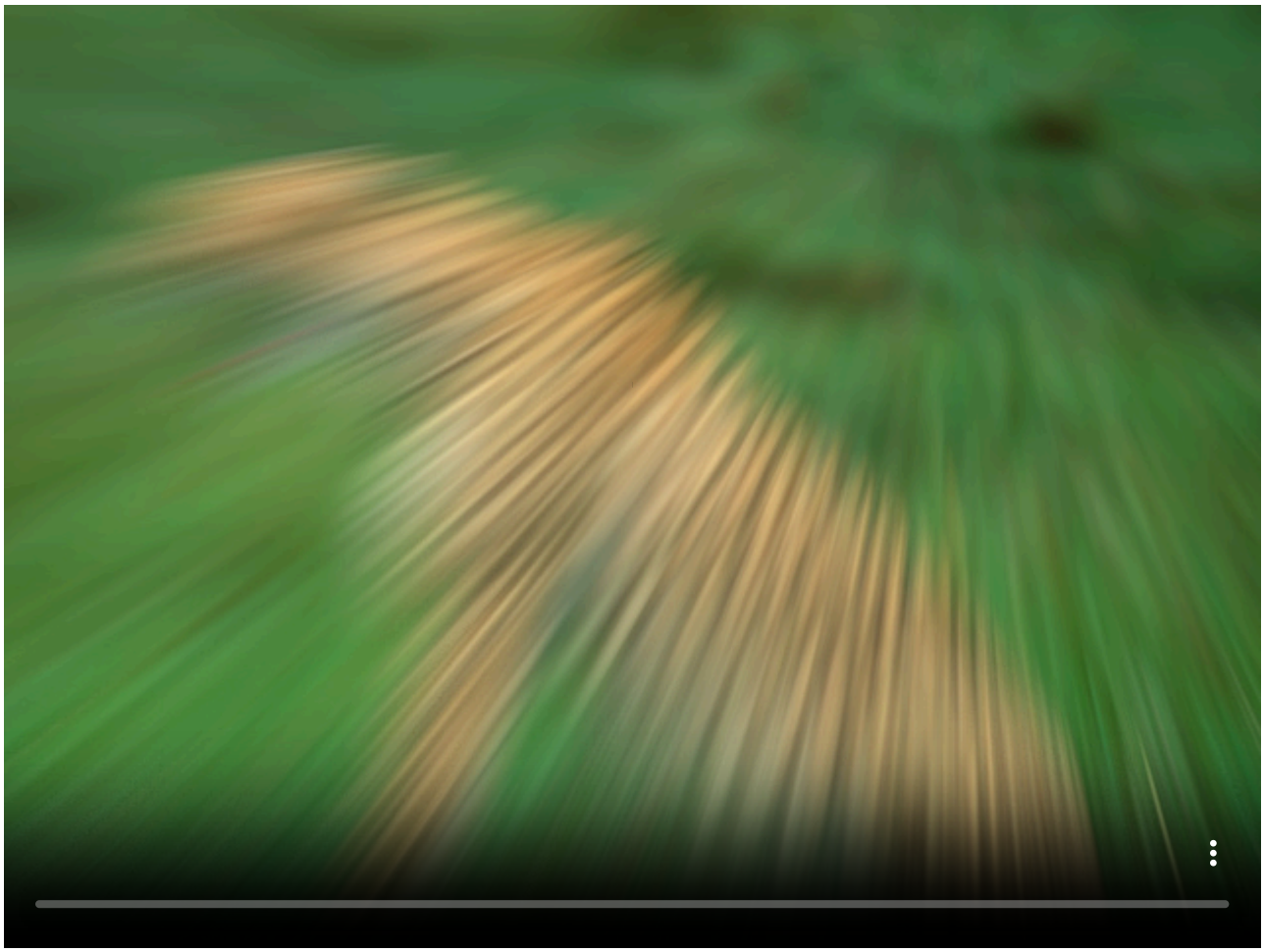
Link to the full example

FRAMEBLUR

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val blurred = colorBuffer(image.width, image.height)
        val blur = FrameBlur()
        val rt = renderTarget(width, height) {
            colorBuffer()
        }

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.BLACK)
                drawer.image(image, cos(seconds * 2) * 40.0, sin(seconds * 2) * 40.0)
            }

            blur.blend = 0.01
            blur.apply(rt.colorBuffer(0), blurred)
            drawer.image(blurred)
        }
    }
}
```

Link to the full example

ZOOMBLUR



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val blurred = colorBuffer(image.width, image.height)
        val blur = ZoomBlur()

        extend {
            blur.center = Vector2(cos(seconds) * 0.5 + 0.5, sin(seconds * 2) * 0.5 + 0.5)
            blur.strength = cos(seconds * 2) * 0.5 + 0.5
            blur.apply(image, blurred)
            drawer.image(blurred)
        }
    }
}
```

Link to the full example

Color filters

CHROMATICABERRATION

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = ChromaticAberration()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.aberrationFactor = cos(seconds * 2) * 10.0
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

COLORCORRECTION

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = ColorCorrection()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.hueShift = cos(seconds * 1) * 180.0
            filter.saturation = cos(seconds * 2)
            filter.brightness = sin(seconds * 3) * 0.1
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

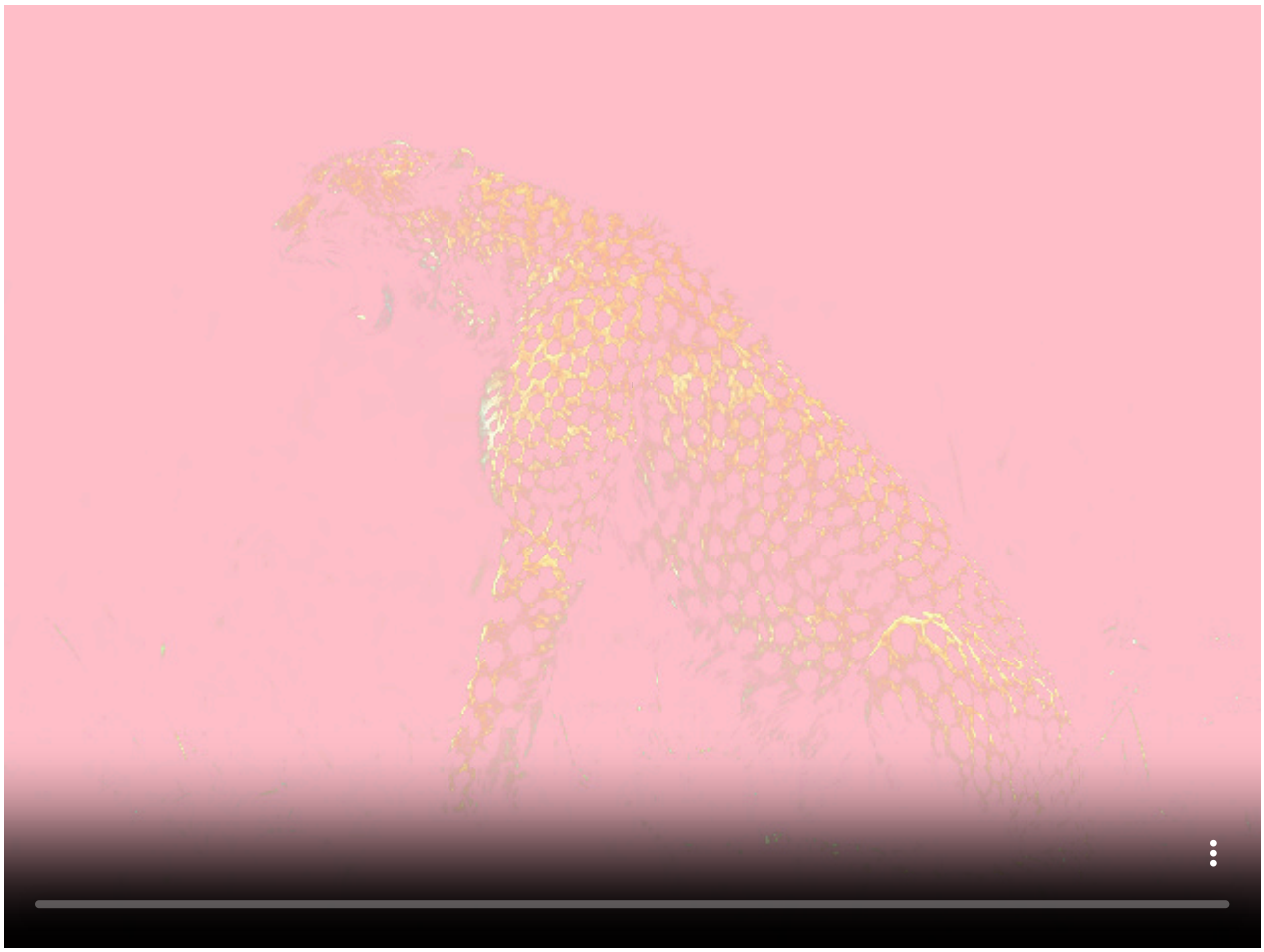
Link to the full example

SEPIA

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Sepia()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.amount = cos(seconds * 2) * 0.5 + 0.5
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

LUMAOPACITY



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = LumaOpacity()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            // -- a pink background to demonstrate the introduced transparencies
            drawer.clear(ColorRGBa.PINK)
            filter.backgroundOpacity = 0.0
            filter.foregroundOpacity = 1.0
            filter.backgroundLuma = cos(seconds) * 0.25 + 0.25
            filter.foregroundLuma = 1.0 - (cos(seconds) * 0.25 + 0.25)

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

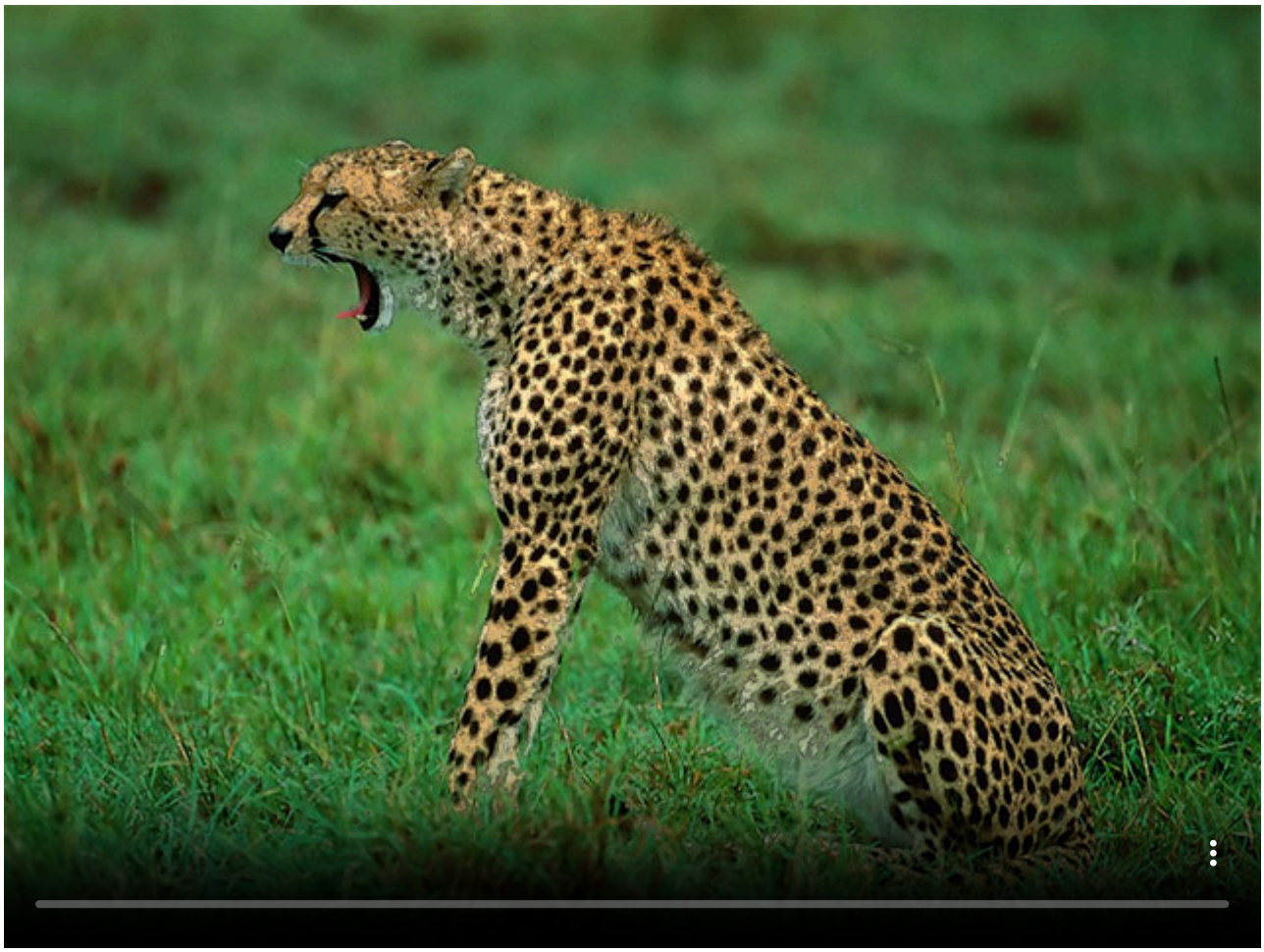
Edge-detection filters

LUMASOBEL

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = LumaSobel()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.backgroundColor = ColorRGBa.PINK.toHSVa().shiftHue(cos(seconds) * 180).toRGBa().shade(0.25)
            filter.edgeColor = ColorRGBa.PINK
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

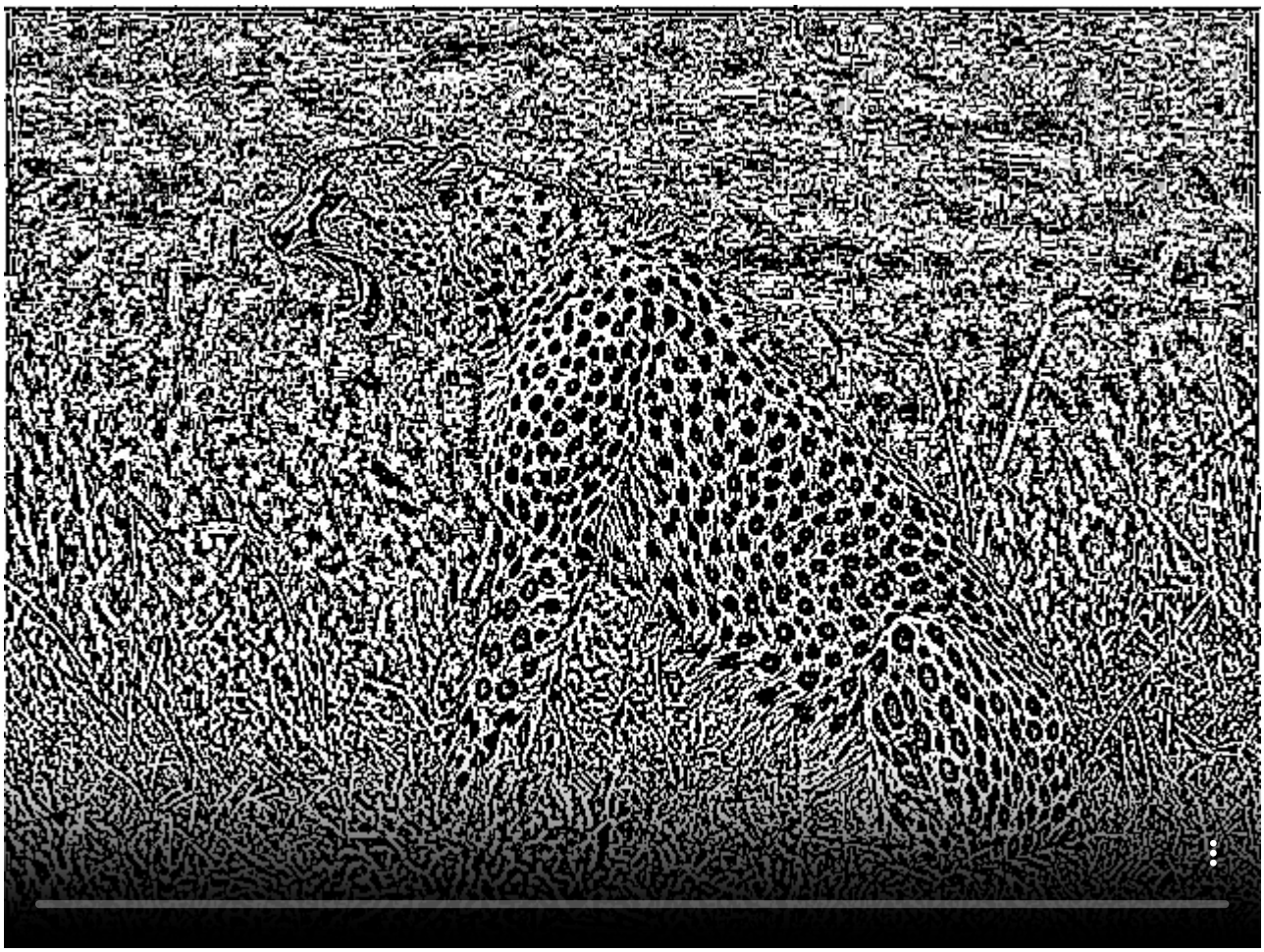
CONTOUR



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Contour()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.backgroundOpacity = 1.0
            filter.contourColor = ColorRGBa.BLACK
            filter.contourWidth = 0.4
            filter.levels = cos(seconds) * 3.0 + 5.1
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

EDGESWORK



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = EdgesWork()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.radius = (cos(seconds) * 5 + 5).toInt()
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

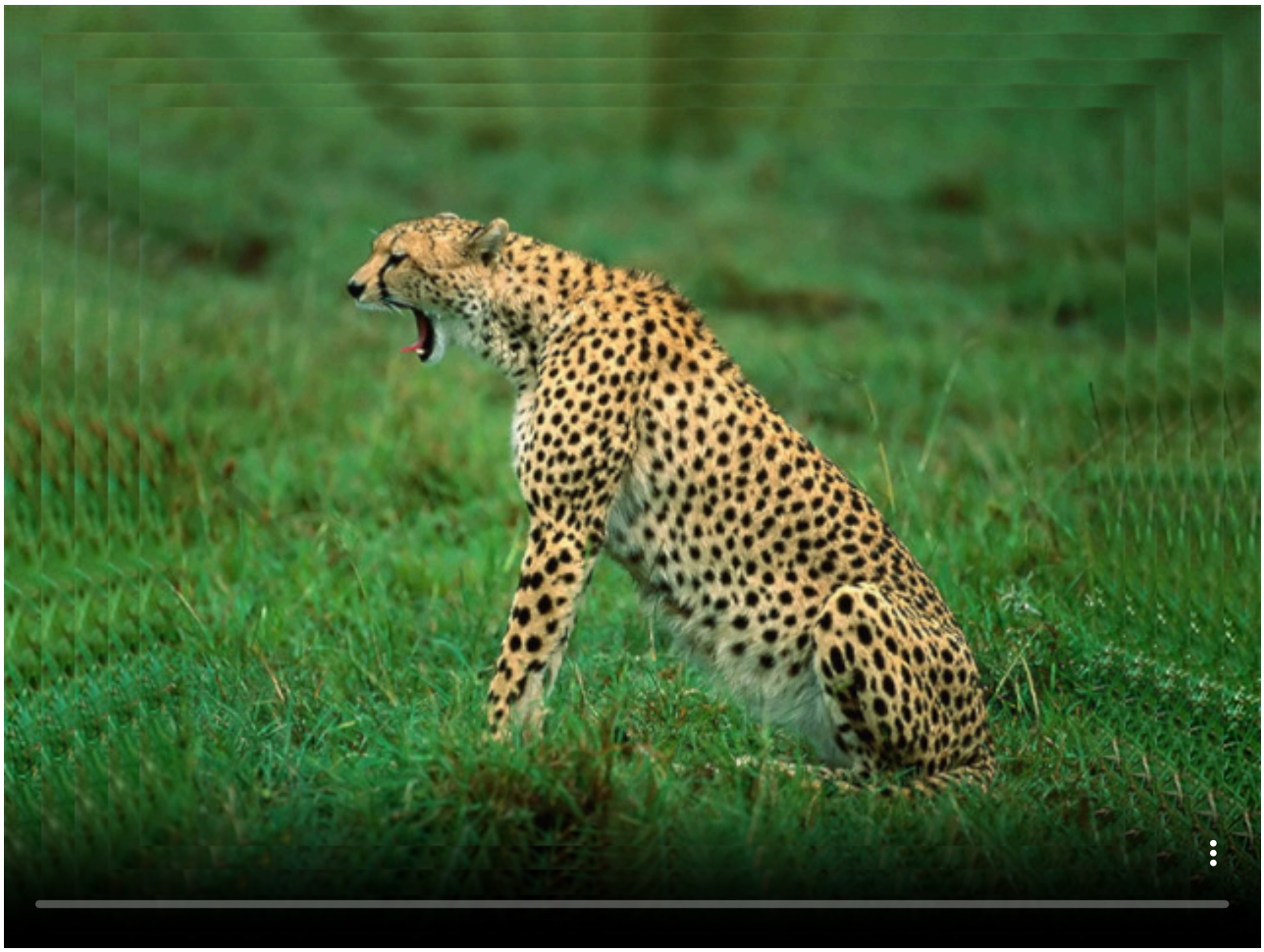
Distortion filters

BLOCKREPEAT

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = BlockRepeat()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.sourceScale = seconds / 5.0
            filter.blockWidth = cos(seconds) * 0.3 + 0.4
            filter.blockHeight = sin(seconds) * 0.3 + 0.4

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

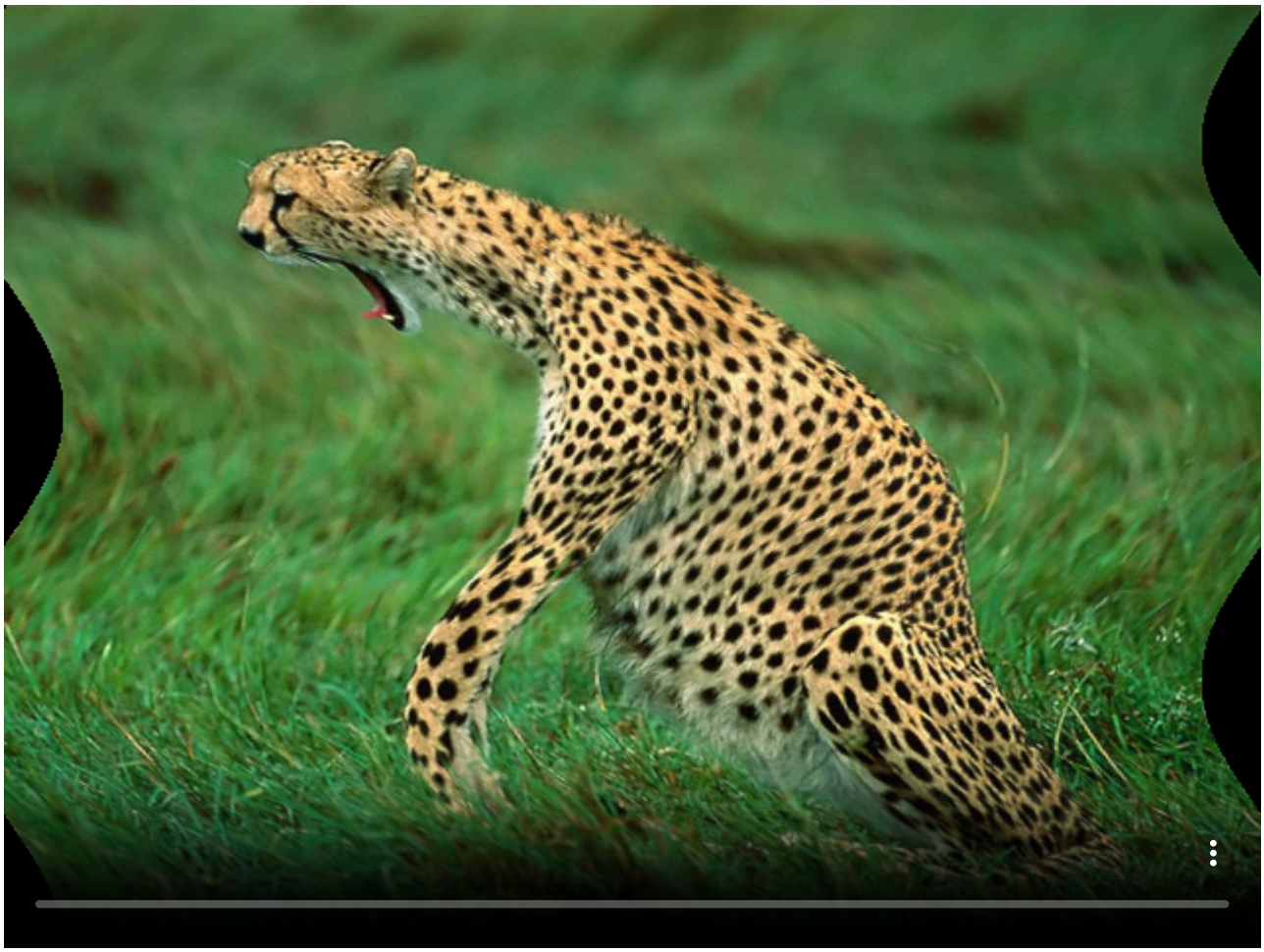
STACKREPEAT



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = StackRepeat()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.repeats = 4
            filter.zoom = (cos(seconds) * 0.1 + 0.11)

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

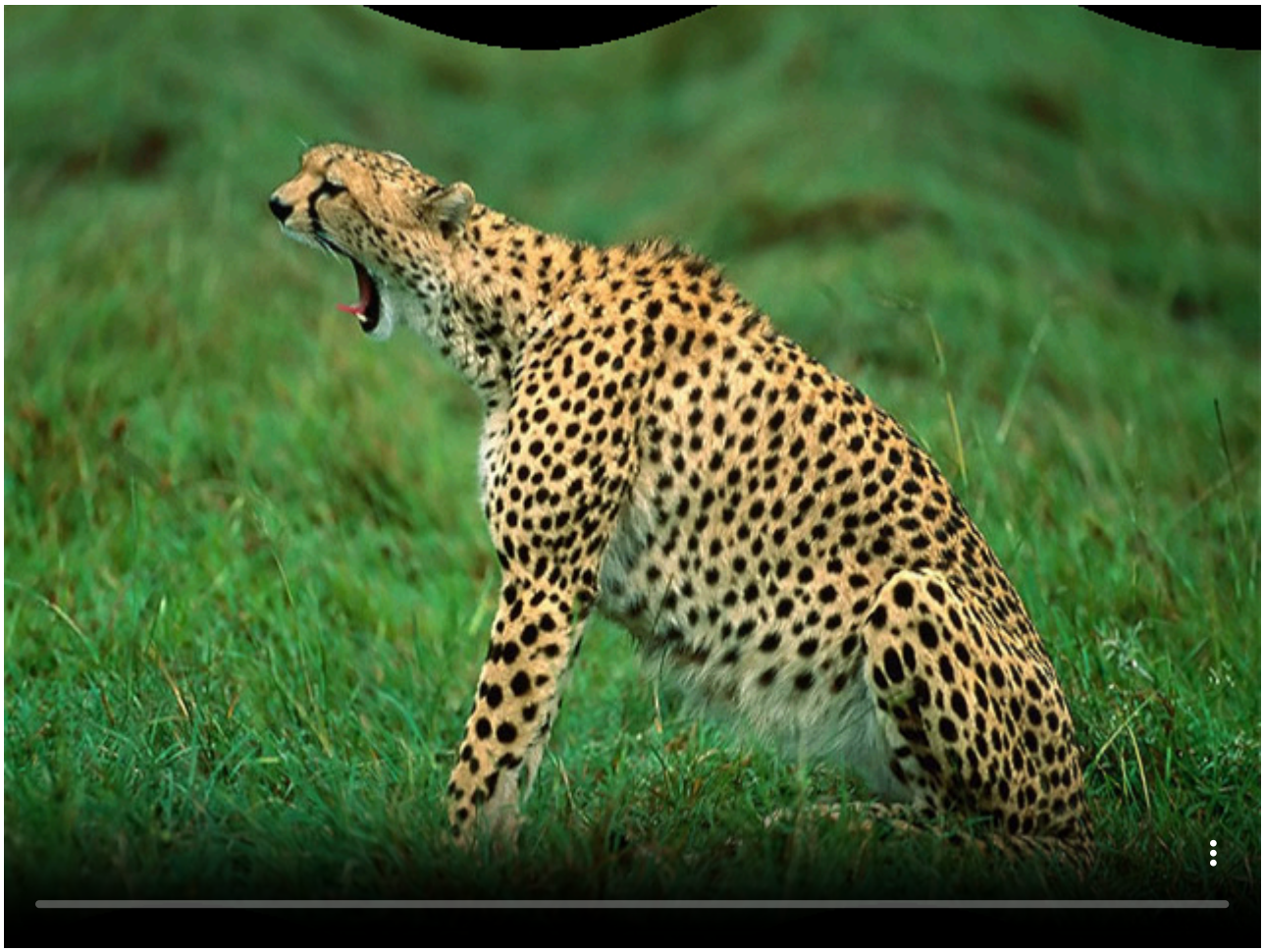
HORIZONTALWAVE



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = HorizontalWave()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.amplitude = cos(seconds) * 0.1
            filter.frequency = sin(seconds) * 4.0
            if (seconds > 2.5) {
                filter.segments = 10
            }

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

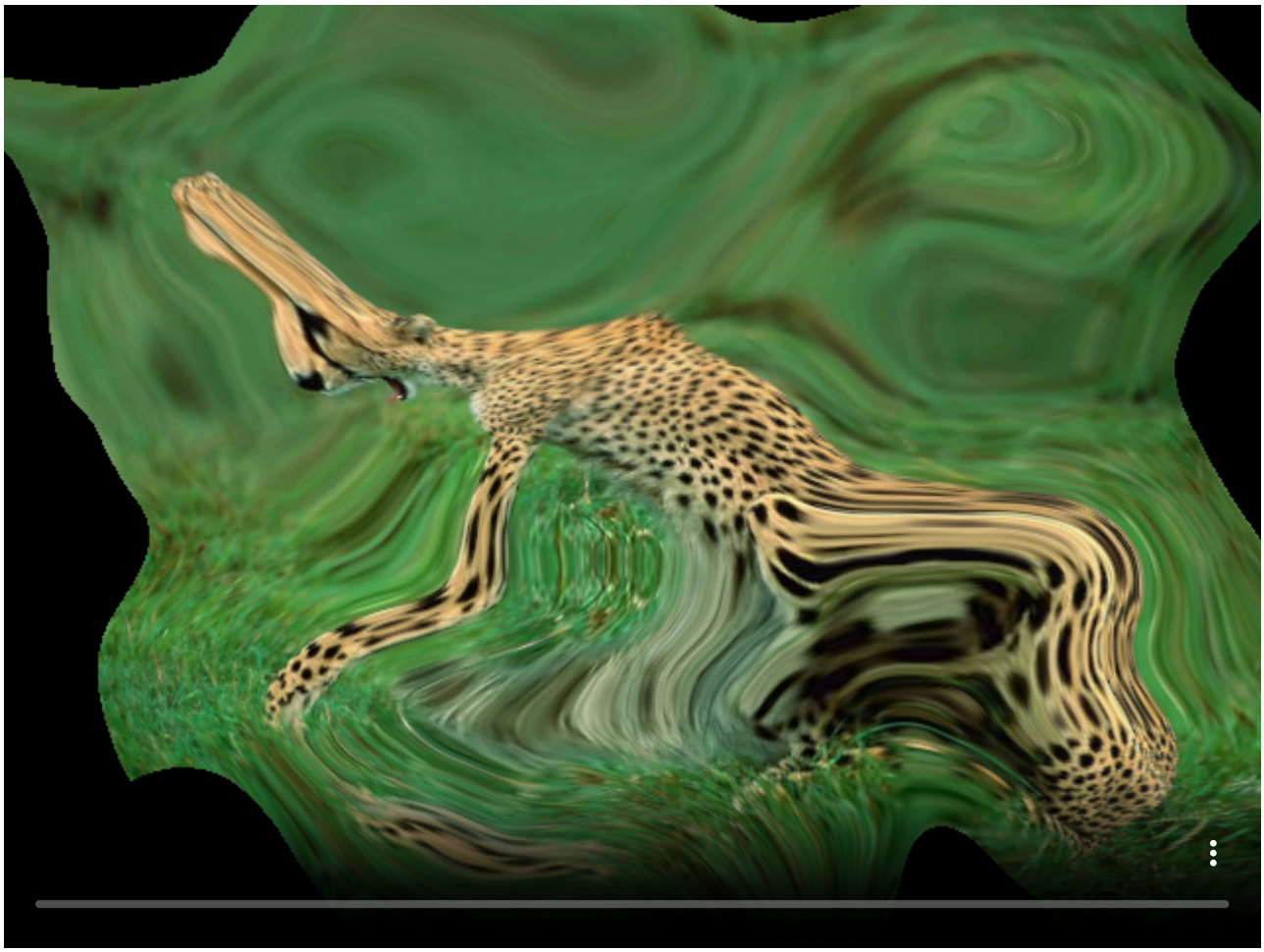
VERTICALWAVE



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = VerticalWave()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.amplitude = cos(seconds) * 0.1
            filter.frequency = sin(seconds) * 4.0
            if (seconds > 2.5) {
                filter.segments = 10
            }

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

PERTURB



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Perturb()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.phase = seconds * 0.1
            filter.decay = 0.168
            filter.gain = cos(seconds) * 0.5 + 0.5

            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

TILES

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Tiles()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.rotation = seconds * 60.0
            filter.xSegments = (10 + cos(seconds * PI / 3) * 5.0).toInt()
            filter.ySegments = 30
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

FISHEYE

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Fisheye()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.strength = cos(seconds) * 0.125
            filter.scale = 1.1
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

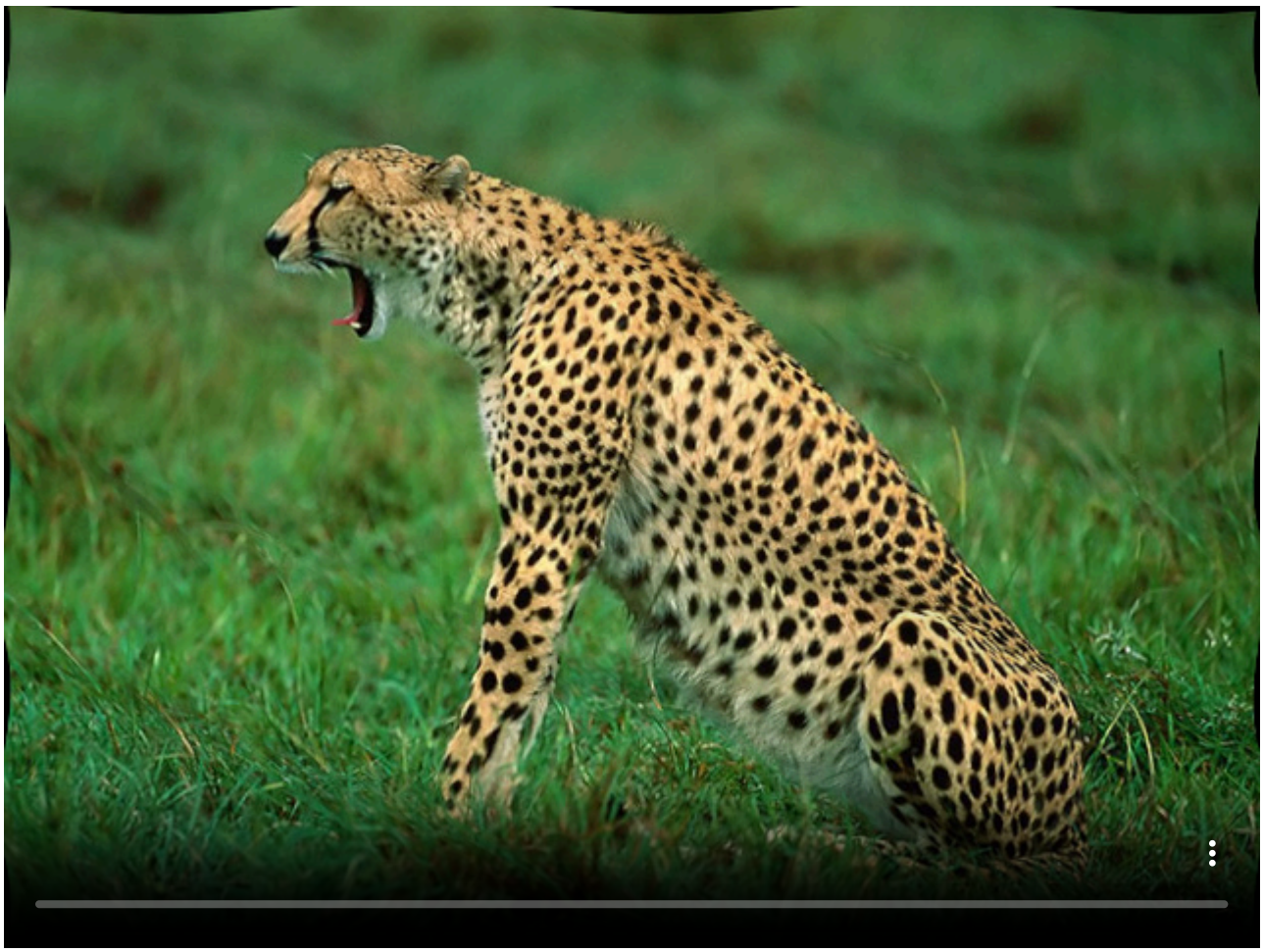
DISPLACEBLEND

```
fun main() = application {
    program {
        val composite = compose {
            colorType = ColorType.FLOAT16
            layer {
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.imageFit(image, 0.0, 0.0, width * 1.0, height * 1.0)
                }
            }
            layer {
                draw {
                    drawer.shadeStyle = linearGradient(ColorRGBa.BLACK, ColorRGBa.WHITE)
                    drawer.stroke = null
                    val size = cos(seconds * PI / 3) * 100.0 + 200.0
                    drawer.rectangle(width / 2.0 - size / 2, height / 2.0 - size / 2, size, size)
                }
                blend(DisplaceBlend()) {
                    gain = cos(seconds * PI / 3) * 0.5 + 0.5
                    rotation = seconds * 60.0
                }
            }
        }

        extend {
            composite.draw(drawer)
        }
```

```
    }
}
```

Link to the full example

STRETCHWAVES

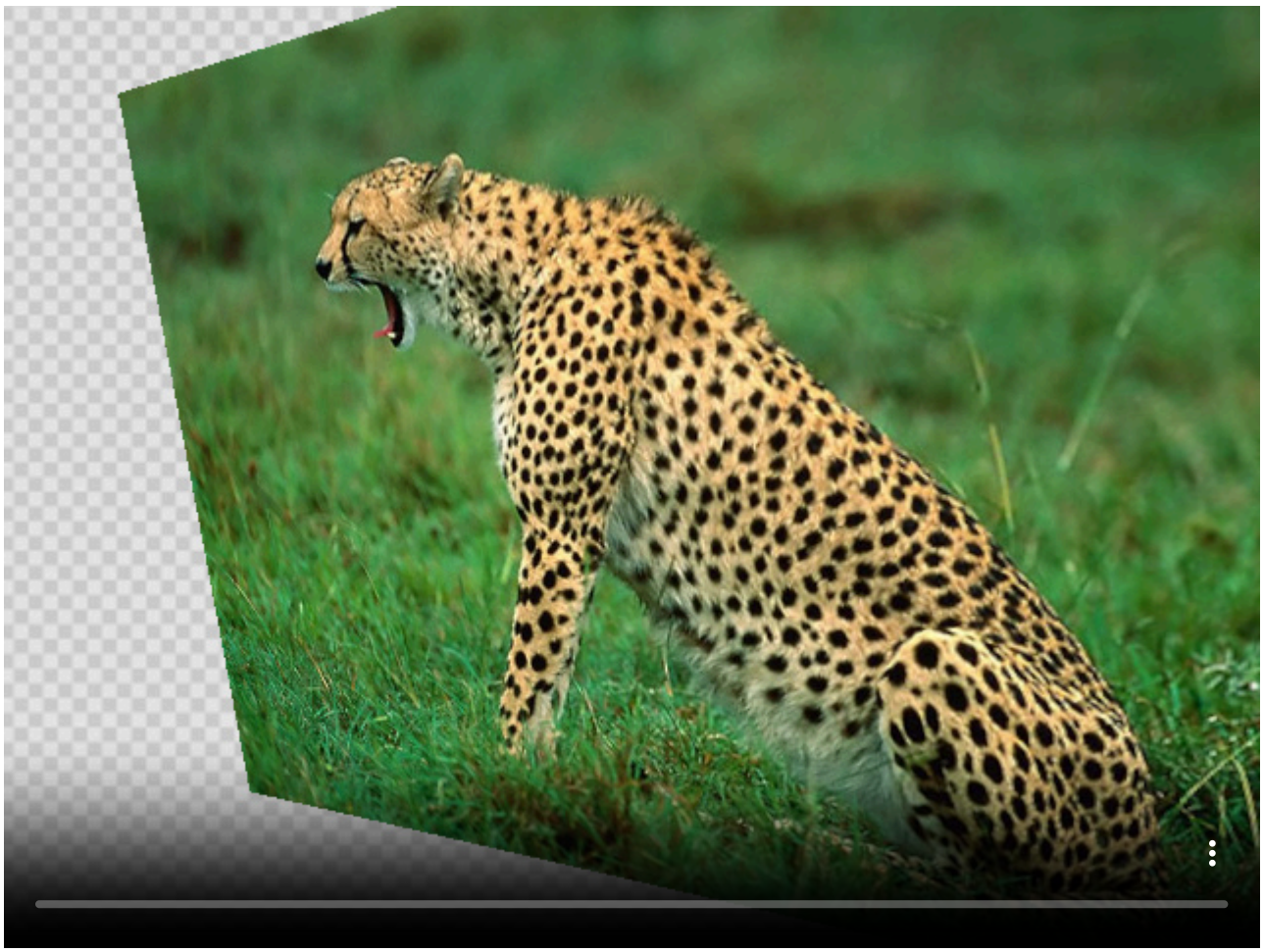


```
fun main() = application {
    program {
        val composite = compose {
            colorType = ColorType.FLOAT16
            layer {
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.imageFit(image, 0.0, 0.0, width * 1.0, height * 1.0)
                }
                post(StretchWaves()) {
                    distortion = 0.25
                    rotation = seconds * 60.0
                    phase = seconds * 0.25
                    frequency = 5.0
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
```

```
    }
}
```

Link to the full example

PERSPECTIVEPLANE



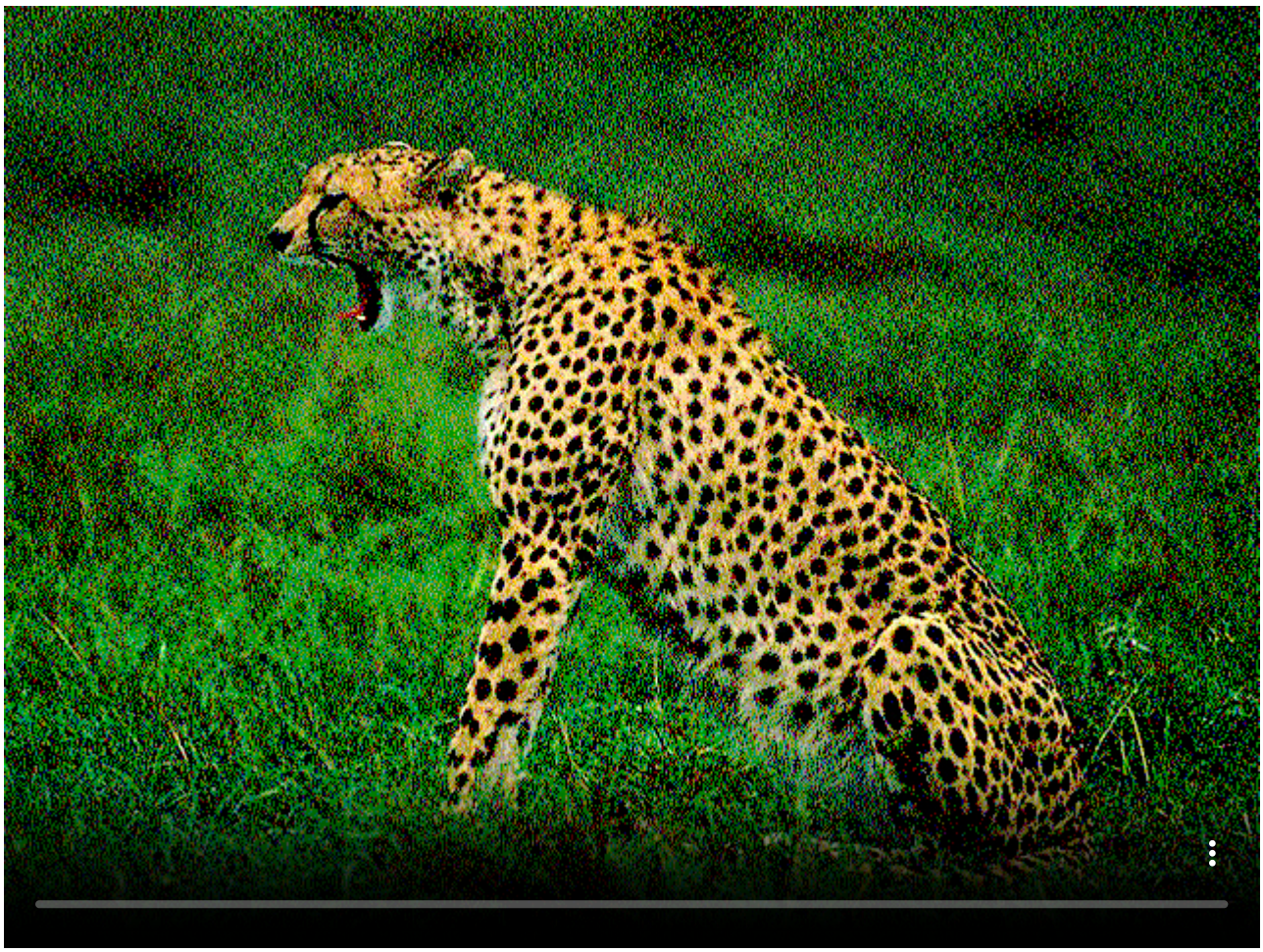
```
fun main() = application {
    program {
        val composite = compose {
            layer {
                post(Checkers())
            }

            layer {
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.imageFit(image, 0.0, 0.0, width * 1.0, height * 1.0)
                }
                post(PerspectivePlane()) {
                    planePitch = cos(seconds) * 22.5
                    planeYaw = sin(seconds) * 22.5
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
```

```
    }
}
```

Link to the full example

Dithering filters

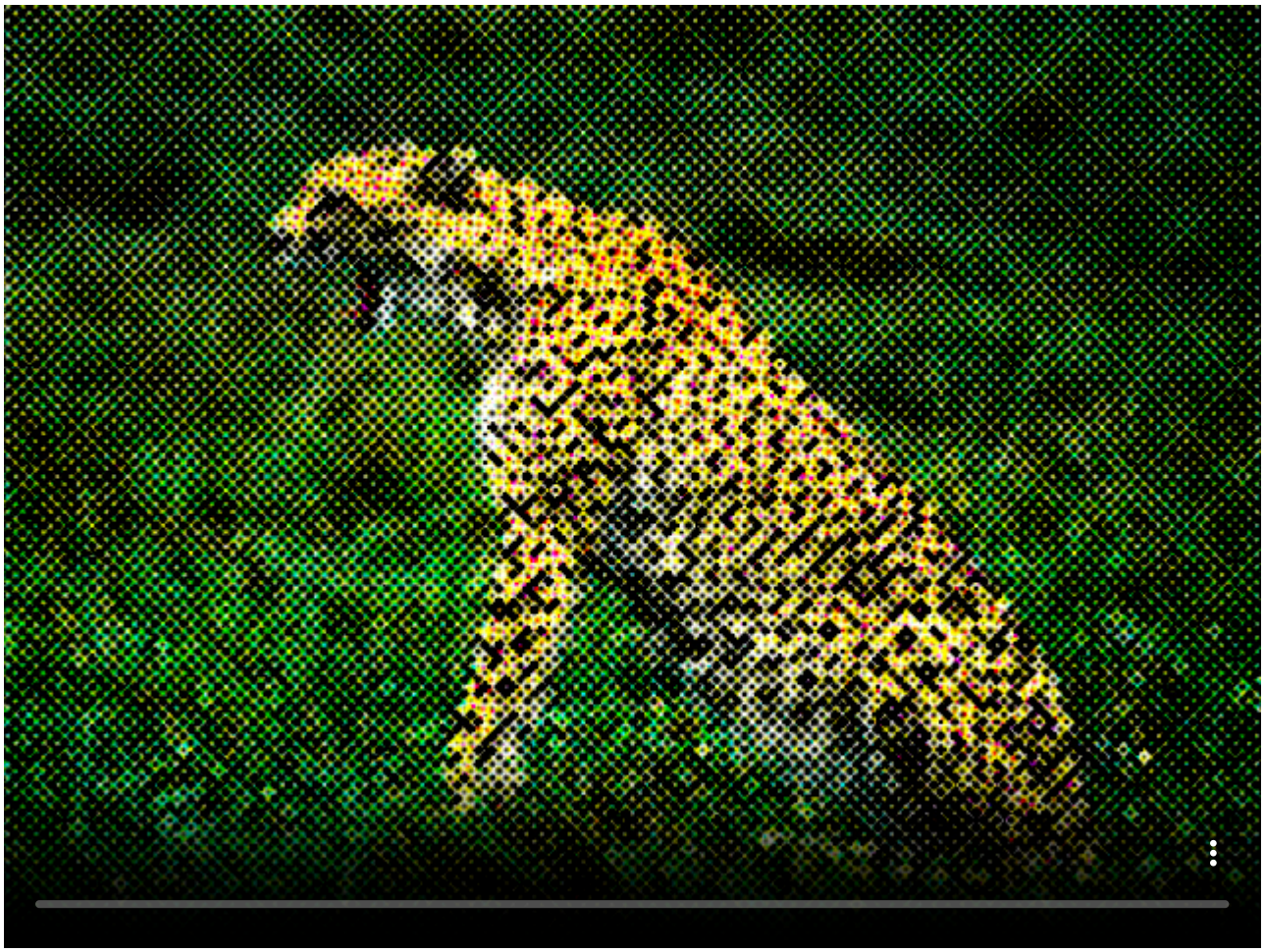
ADITHER



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = ADither()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            filter.pattern = ((seconds / 5.0) * 4).toInt().coerceAtMost(3)
            filter.levels = ((seconds % 1.0) * 3).toInt() + 1
            filter.apply(image, filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

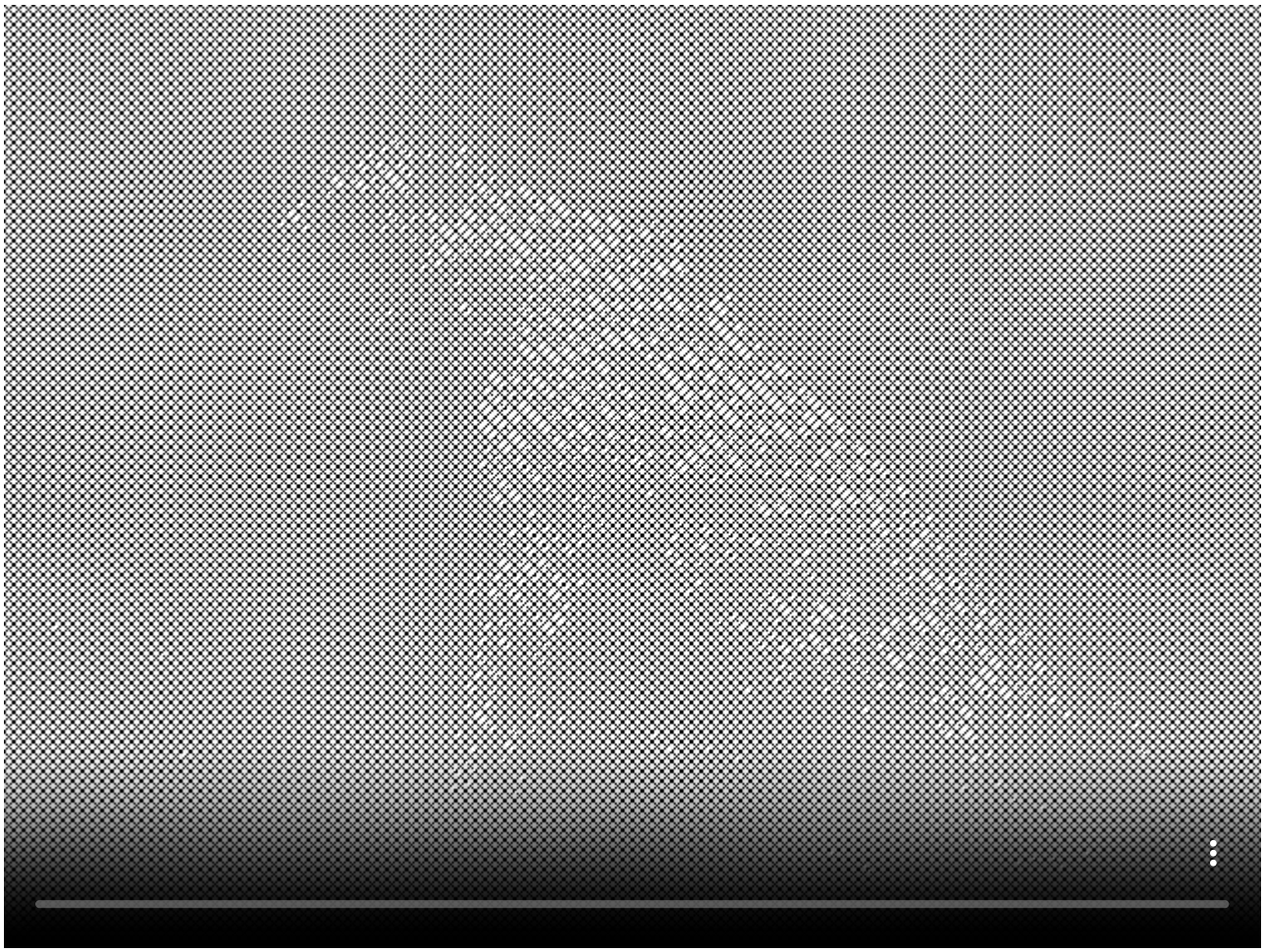
CMYKHALFTONE



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = CMYKHalftone()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            // -- need a white background because the filter introduces transparent areas
            drawer.clear(ColorRGBa.WHITE)
            filter.dotSize = 1.2
            filter.scale = cos(seconds) * 2.0 + 6.0
            filter.apply(image, filtered)

            drawer.image(filtered)
        }
    }
}
```

Link to the full example

CROSSHATCH

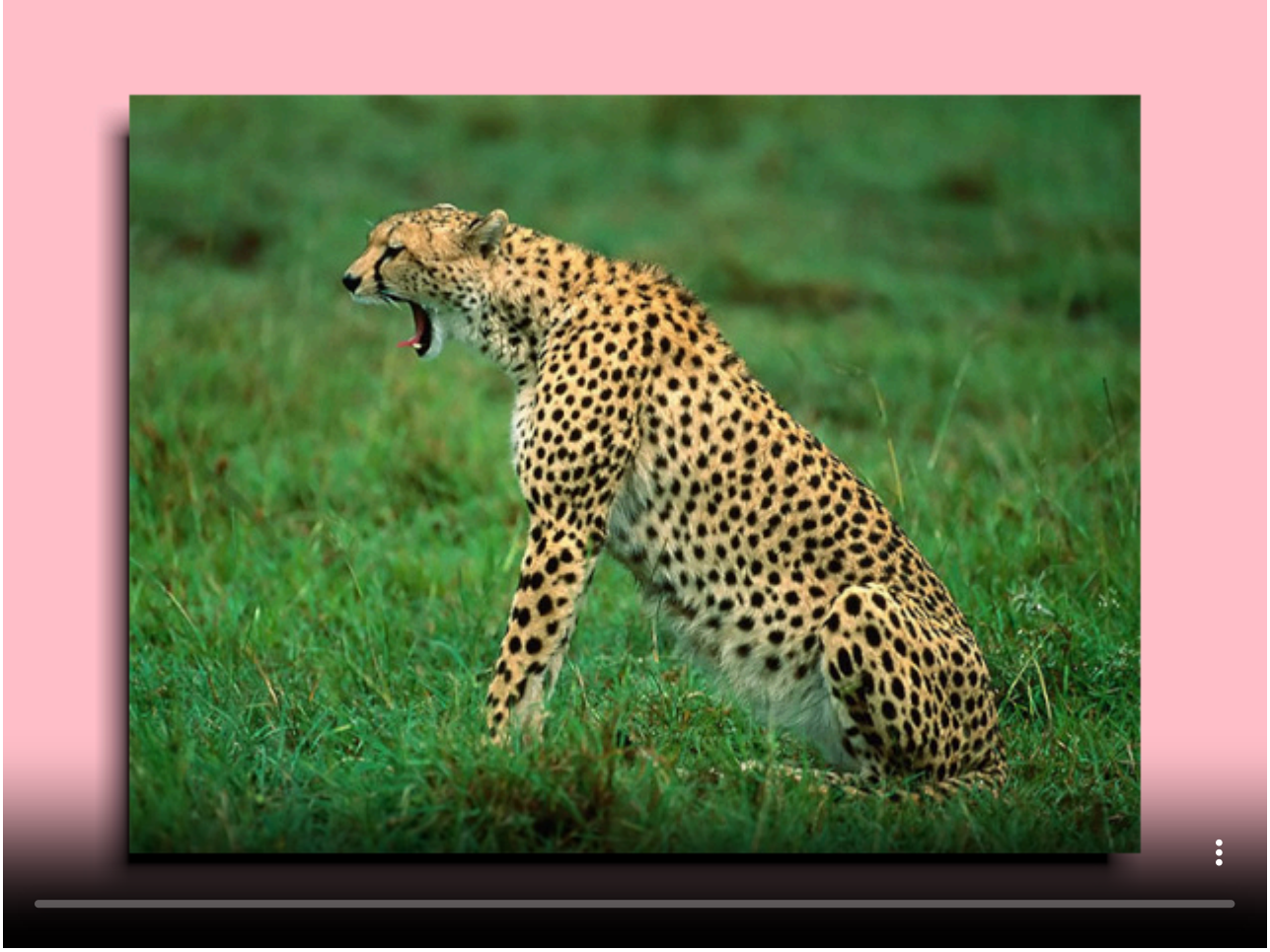


```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = Crosshatch()
        val filtered = colorBuffer(image.width, image.height)

        extend {
            // -- need a white background because the filter introduces transparent areas
            drawer.clear(ColorRGBa.WHITE)
            filter.t1 = cos(seconds * PI) * 0.25 + 0.25
            filter.t2 = filter.t1 + cos(seconds * 3) * 0.25 + 0.25
            filter.t3 = filter.t2 + cos(seconds * 2) * 0.25 + 0.25
            filter.t4 = filter.t3 + cos(seconds * 1) * 0.25 + 0.25

            filter.apply(image, filtered)

            drawer.image(filtered)
        }
    }
}
```

Link to the full example

Shadow filters

DROPSHADOW



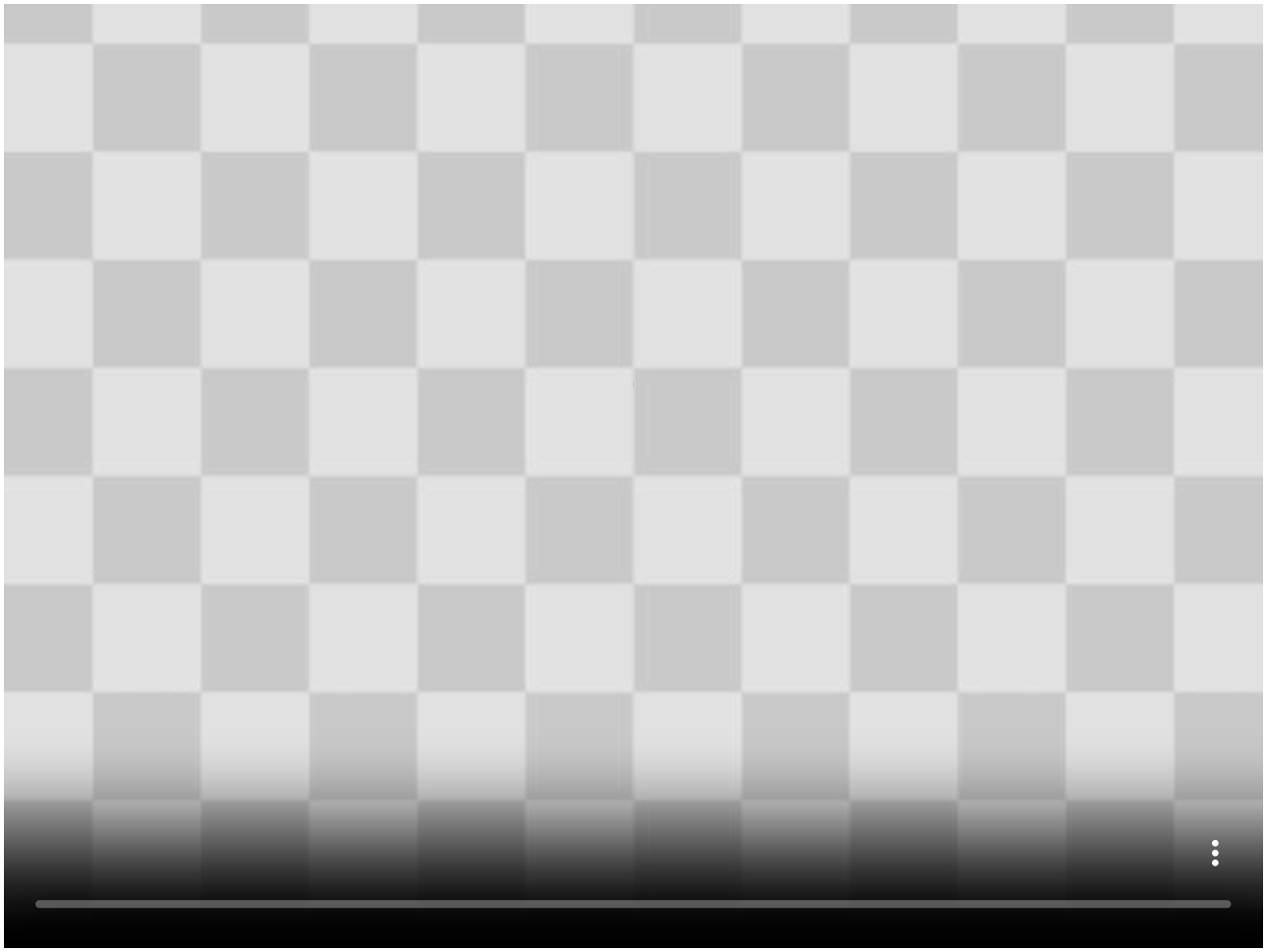
```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val filter = DropShadow()
        val filtered = colorBuffer(image.width, image.height)

        val rt = renderTarget(width, height) {
            colorBuffer()
        }

        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.TRANSPARENT)
                drawer.image(image, (image.width - image.width * 0.8) / 2, (image.height - image.height * 0.8) / 2,
image.width * 0.8, image.height * 0.8)
            }
            // -- need a pink background because the filter introduces transparent areas
            drawer.clear(ColorRGBa.PINK)
            filter.window = (cos(seconds) * 16 + 16).toInt()
            filter.xShift = cos(seconds * 2) * 16.0
            filter.yShift = sin(seconds * 2) * 16.0
            filter.apply(rt.colorBuffer(0), filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example

Pattern filters

CHECKERS

`Checkers` is a simple checker generator filter.



```
fun main() = application {
    program {
        val composite = compose {
            layer {
                post(Checkers()) {
                    size = cos(seconds) * 0.6 + 0.4
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example

edit on GitHub

Compositing using orx-compositor

orx-compositor offers a simple DSL for the creation of layered graphics. The compositor manages blending and post-processing of layers for you.

`orx-compositor` works well together with `orx-fx`, `orx-gui`, and `orx-olive`, although they are not a required combination it is worth checking out what the combination has to offer.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-compositor` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Workflow

Let us now work through the workflow of `orx-compositor`. One usually starts with an OPENRNDR skeleton program:

```
fun main() = application {
    program {
        extend {
        }
    }
}
```

Which by itself, of course, does nothing. Let's extend this skeleton a bit and add the basics for layered graphics. We add a `composite` using `compose {}` and we make sure that our OPENRNDR program draws it on refresh.

Note, if this fails you can fix it by adding `import org.openrndr.extra.compositor.draw`.

```
fun main() = application {
    program {
        val composite = compose {
        }

        extend {
            composite.draw(drawer)
        }
    }
}
```

Now let's draw something. We do this by adding a `draw {}` inside the `compose {}`. Here we see we use `drawer` like we would use it normally (it is captured in the closure). We also added a `println` to demonstrate that the code inside `compose {}` is executed once, however, code inside `draw {}` is executed every time the composite is drawn.

```
fun main() = application {
    program {
        val composite = compose {
            println("this is only executed once")

            draw {
```

```
                drawer.fill = ColorRGBa.PINK
                drawer.stroke = null
                drawer.circle(width / 2.0, height / 2.0, 175.0)
            }
        }

        extend {
            composite.draw(drawer)
        }
    }
}
```

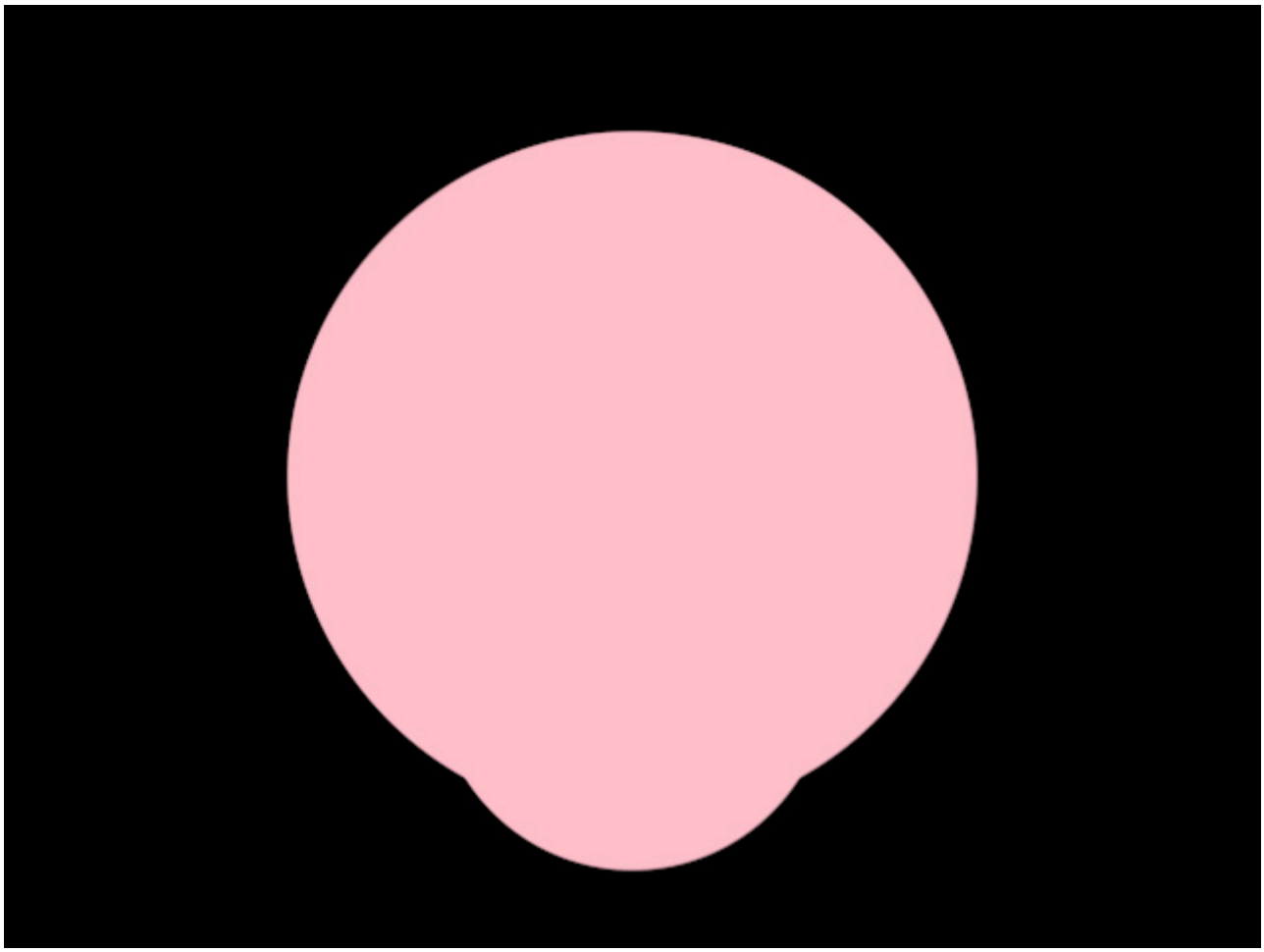
Let's get to what `orx-compositor` promises: layered graphics. We do this by adding a `layer {}` inside our composite, and inside this layer we add another `draw`.

Every layer has an isolated draw state to prevent users from leaking draw state.

```
fun main() = application {
    program {
        val composite = compose {
            draw {
                drawer.fill = ColorRGBa.PINK
                drawer.stroke = null
                drawer.circle(width / 2.0, height / 2.0, 175.00)
            }

            layer {
                draw {
                    drawer.fill = ColorRGBa.PINK
                    drawer.stroke = null
                    drawer.circle(width / 2.0, height / 2.0 + 100.0, 100.0)
                }
            }
        }

        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example

This produces:



You may be thinking: “yeah great, we added all that extra structure to the code, but it doesn’t do a single thing that could not be achieved by drawing two circles consecutively”. And you’re right. However, there are now two things we can add with ease: *blends* and *posts*. Here a *blend* describes how a layer’s contents should be combined with the layer it covers, and a *post* a filter that is applied after the contents have been drawn.

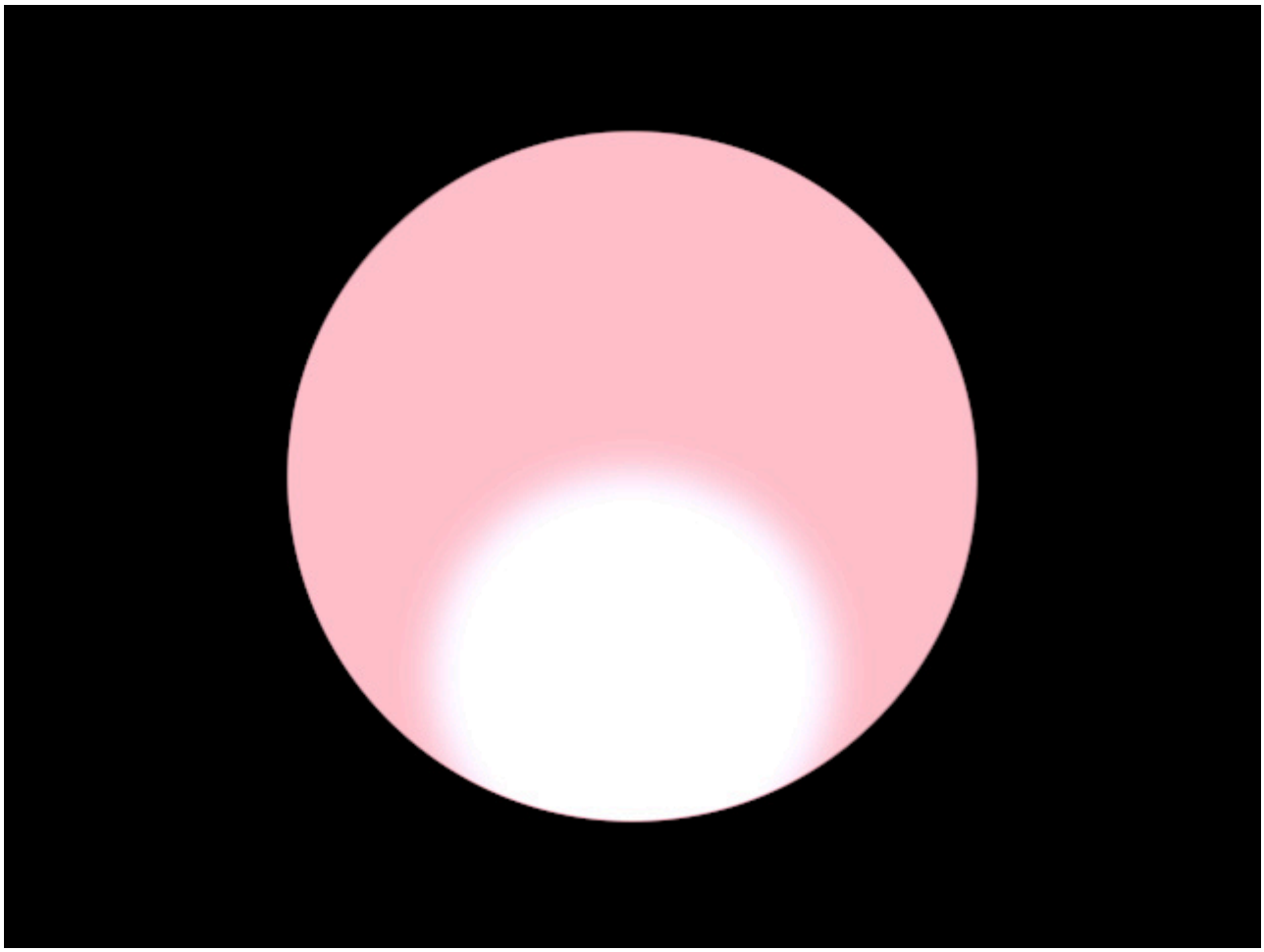
Let’s add a blend and a post to our layer and see what it does:

```
fun main() = application {
    program {
        val composite = compose {
            draw {
                drawer.fill = ColorRGBa.PINK
                drawer.stroke = null
                drawer.circle(width / 2.0, height / 2.0, 175.0)
            }

            layer {
                blend(Add()) {
                    clip = true
                }
                draw {
                    drawer.fill = ColorRGBa.PINK
                    drawer.stroke = null
                    drawer.circle(width / 2.0, height / 2.0 + 100.0, 100.0)
                }
                post(ApproximateGaussianBlur()) {
                    window = 25
```

```
                    sigma = 10.00
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example

The output:



We now see a couple of differences. The smaller circle is blurred while the larger circle is not; The area where the two circles overlap is brighter; The smaller circle is clipped against the larger circle.

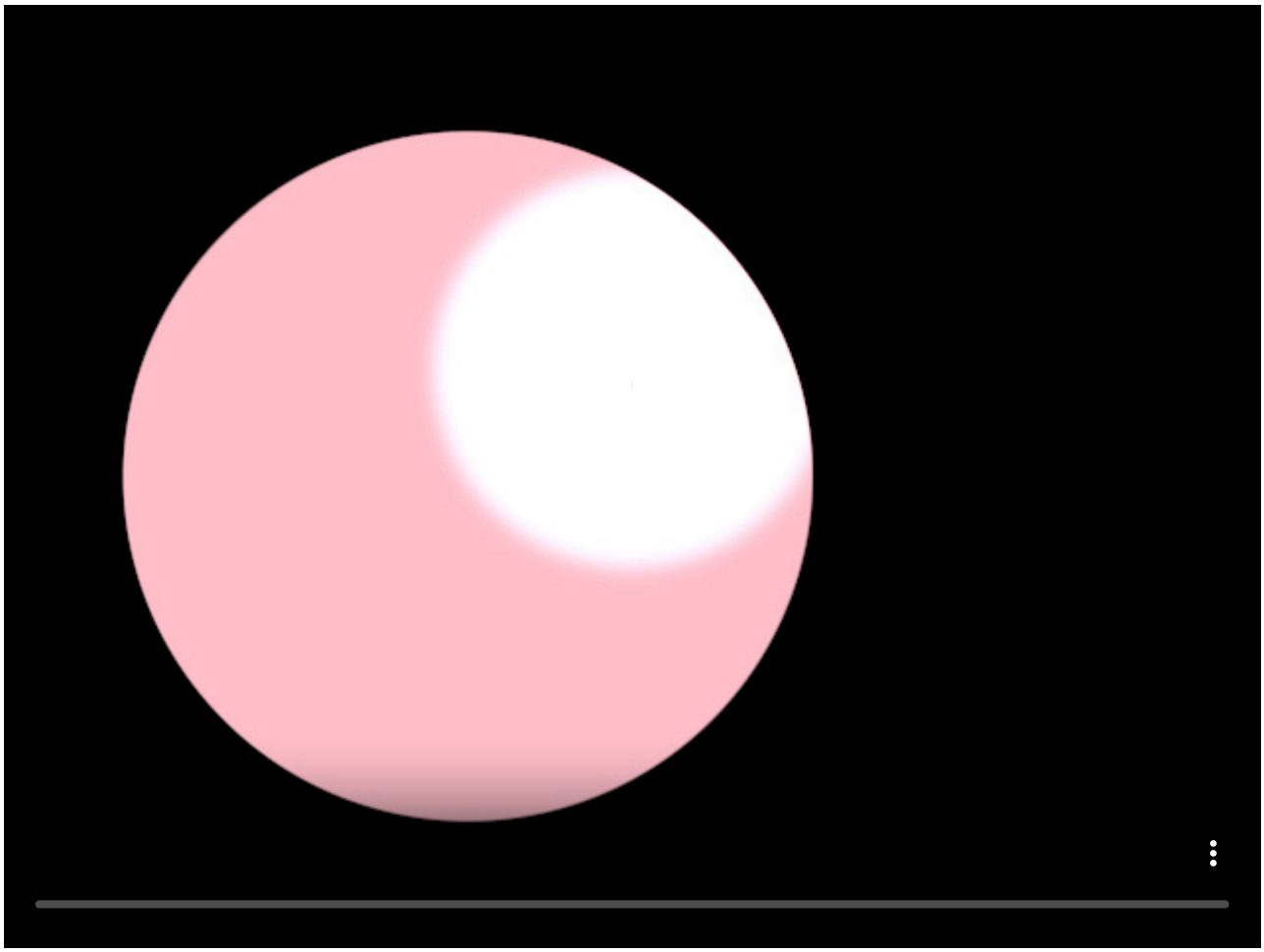
These are a results that are not as easily replicated without `orx-compositor`.

Note that the parameters for the *post* filters (and *blend*) can be animated, just as the layers contents can be animated:

```
fun main() = application {
    program {
        val composite = compose {
            draw {
                drawer.fill = ColorRGBa.PINK
                drawer.stroke = null
```

```
                drawer.circle(width / 2.0 + sin(seconds * 2) * 100.0, height / 2.0, 175.0)
            }

            layer {
                blend(Add()) {
                    clip = true
                }
                draw {
                    drawer.fill = ColorRGBa.PINK
                    drawer.stroke = null
                    drawer.circle(width / 2.0, height / 2.0 + cos(seconds * 2) * 100.0, 100.0)
                }
                post(ApproximateGaussianBlur()) {
                    // -- this is actually a function called for every draw
                    window = 25
                    sigma = cos(seconds) * 10.0 + 10.01
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
    }
}
```

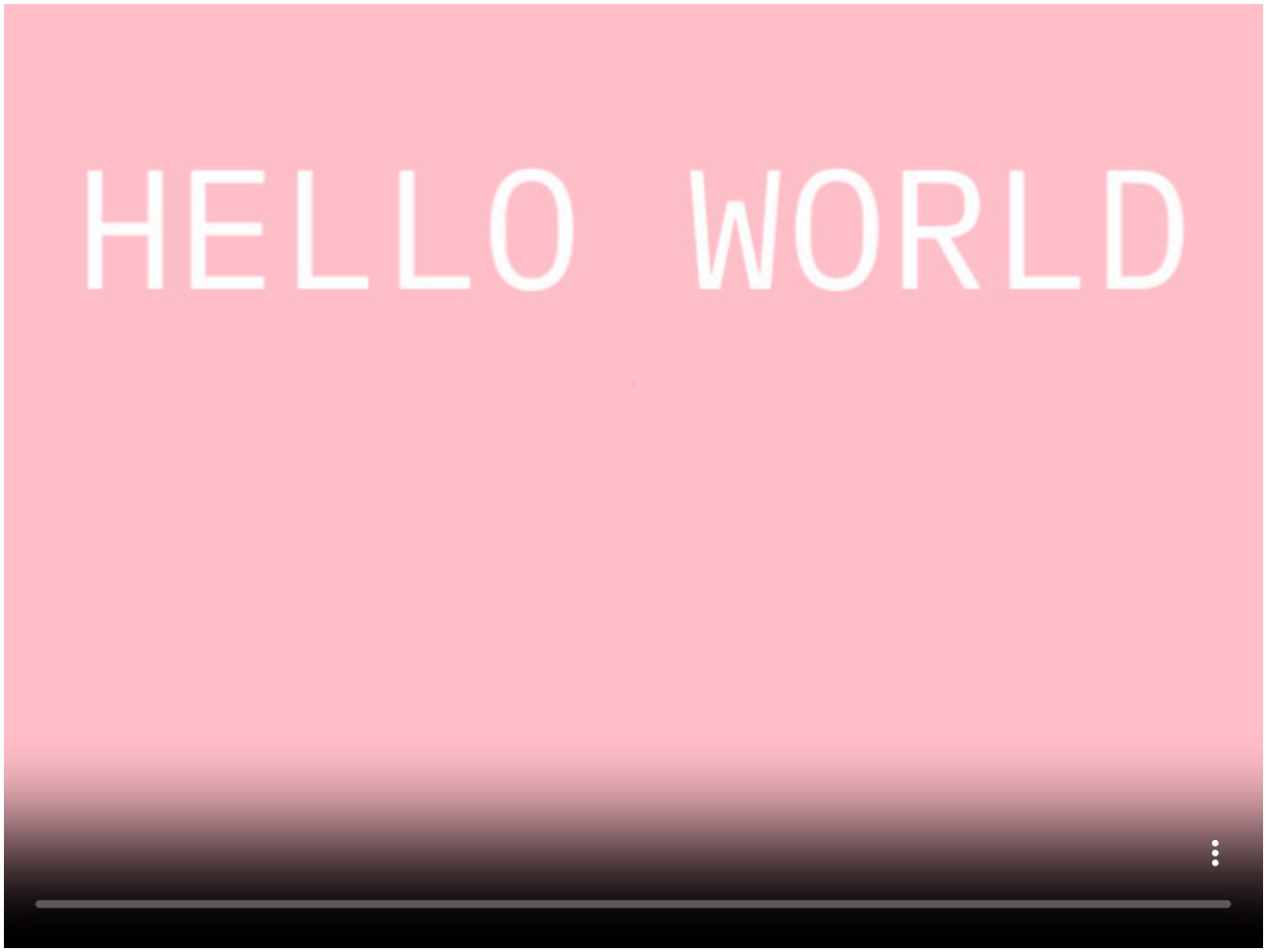
Link to the full example



Common use-cases

Masking

In this case we have a text and an image that we only want to draw where there is text. This can be achieved by using nested layers and a `Normal` blend with `clip` enabled.




```
fun main() = application {
    program {
        val composite = compose {
            draw {
                drawer.clear(ColorRGBa.PINK)
            }
            layer {
                // -- we nest layers to prevent the text layer to be blended with the background
                // -- before it is blended with the image layer
                layer {
                    // -- notice how we load the font inside the layer
                    // -- this only happens once
                    val font = loadFont("data/fonts/default.otf", 112.0)
                    draw {
                        drawer.fill = ColorRGBa.WHITE
                        drawer.fontMap = font
                        val message = "HELLO WORLD"
                        writer {
                            val w = textWidth(message)
                            cursor = Cursor((width - w) / 2.0, height / 2.0 + cos(seconds) * 200.0)
                            text(message)
```

```
                    }
                }
            }
            layer {
                // -- again, loading resources inside the layer is perfectly fine
                // -- it is also a good way to keep code free of clutter
                val image = loadImage("data/images/cheeta.jpg")

                // -- we use a normal blend here
                blend(Normal()) {
                    // -- and we set `clip` to true
                    clip = true
                }
                draw {
                    // -- we modify the image opacity as a demonstration
                    drawer.drawStyle.colorMatrix = tint(ColorRGBa.WHITE.opacify(cos(seconds * 4)))
                    drawer.image(image)
                }
            }
        }
    }
    extend {
        composite.draw(drawer)
    }
}
```

Link to the full example

Drop shadows

In case you want to place text over an image and want to guarantee the text is readable. You can use a drop shadow *post* effect to draw the text with a bit of a shadow that sets the text apart from the image.


```
fun main() = application {
    program {
        val composite = compose {
            draw {
                drawer.clear(ColorRGBa.PINK)
            }
            layer {
                // -- load the image inside the layer
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.image(image)
                }
            }
            // -- add a second layer with text and a drop shadow
            layer {
                // -- notice how we load the font inside the layer
                // -- this only happens once
                val font = loadFont("data/fonts/default.otf", 112.0)

                draw {
                    drawer.fill = ColorRGBa.WHITE
                    drawer.fontMap = font
                    val message = "HELLO WORLD"
                    writer {
                        box = Rectangle(0.0, 0.0, width * 1.0, height * 1.0)
                        val w = textWidth(message)
                        cursor = Cursor((width - w) / 2.0, height / 2.0 + cos(seconds) * 200.0)
```

```
                        text(message)
                    }
                }
                post(DropShadow()) {
                    window = 10
                    gain = 1.0
                    yShift = -sin(seconds) * 8.0
                }
            }
        }

        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example

Multiple effects per layer

Post effects are not limited to one per layer. One can create a chain of post-processing filters by just calling `post() {}` multiple times per layer. In the following example we create a text layer that uses 3 post effects: two distortion effects followed by a blur filter.

```
fun main() = application {
    program {
        val composite = compose {
```

```
            layer {
                // -- load the image inside the layer
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.image(image)
                }
            }

            // -- add a second layer with text and a drop shadow
            layer {
                // -- notice how we load the font inside the layer
                // -- this only happens once
                val font = loadFont("data/fonts/default.otf", 112.0)
                draw {
                    drawer.fill = ColorRGBa.BLACK
                    drawer.fontMap = font
                    val message = "HELLO WORLD"
                    writer {
                        box = Rectangle(0.0, 0.0, width * 1.0, height * 1.0)
                        val w = textWidth(message)
                        cursor = Cursor((width - w) / 2.0, height / 2.0)
                        text(message)
                    }
                }
                // -- this effect is processed first
                post(HorizontalWave()) {
                    amplitude = cos(seconds * 3) * 0.1
                    frequency = sin(seconds * 2) * 4
                    segments = (1 + Math.random() * 20).toInt()
                    phase = seconds
                }
                // -- this is the second effect
                post(VerticalWave()) {
                    amplitude = sin(seconds * 3) * 0.1
                    frequency = cos(seconds * 2) * 4
                    segments = (1 + Math.random() * 20).toInt()
                    phase = seconds
                }
                // -- and this effect is processed last
                post(ApproximateGaussianBlur()) {
                    sigma = cos(seconds * 2) * 5.0 + 5.01
                    window = 25
                }
            }
        }
        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example

Opacity

`orx-fx` is made with opacity as a first-class citizen. By default a layer is set to be fully transparent, most blending and post operations are using and preserving opacity.

Blending

Blending describes how the contents of two layers are combined in a composite. The blend functionality `orx-compositor` can be used with any filter that performs a blend operation. The `orx-fx` filter collection provides a selection of ready-made blend filters.

The following (`orx-fx`) blend filters work well with opacity and have a configurable `clip` option with which the destination layer can be clipped against the source input's opacity:

- `ColorBurn`
- `ColorDodge`
- `Darken`
- `HardLight`
- `Lighten`
- `Multiply`
- `Normal`
- `Overlay`
- `Screen`
- `Add`
- `Subtract`

Reusing a layer

It is possible to use the color buffer of a previously declared layer by using `aside`.

```
fun main() = application {
    program {

        val composite = compose {
            // -- keep a reference to the layer for later use
            val first = aside {
                draw {// -- draw something
                }// post(...) { ... }

            }

            layer {
                draw {
                    drawer.image(first) // <-- reuse a previous layer
                }
                post(ApproximateGaussianBlur())
                blend(Add())
            }
        }
        extend {
            composite.draw(drawer)
        }
```

```
    }
}
```

demo

Multisampling

Edges on rotated or curved contours can look pixelated in some cases. We can control the smoothness / anti-aliasing of each layer by specifying its multisampling level like this:

```
layer(multisample = BufferMultisample.SampleCount(8)) {
```

where `8` is the desired level. Values between 0 (the default) and 16 are typically used.

Quick UIs

orx-gui provides a simple mechanism to create near zero-effort user interfaces. `orx-gui` is a tool written on top of OPENRNDR's UI library with the intention of taking away most mental and work overhead involved in creating simple user interfaces intended for prototyping and hacking purposes. The core principle of `orx-gui` is to generate user interfaces only from annotated classes and properties.

`orx-gui` relies on annotated classes and properties using the annotations in `orx-parameters`

`orx-gui` is incredibly powerful in combination with the live coding environment `orx-olive`, the guide covers that in the live coding section. That said, it is not a required combination.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-gui` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Basic workflow

Using orx-gui starts with the OPENRNDR program skeleton and extension through `Gui`. It is somewhat uncommon, but this time we want to keep a reference to the extension.

```
fun main() = application {

    program {
        val gui = GUI()
        extend(gui)
    }
}
```

Link to the full example

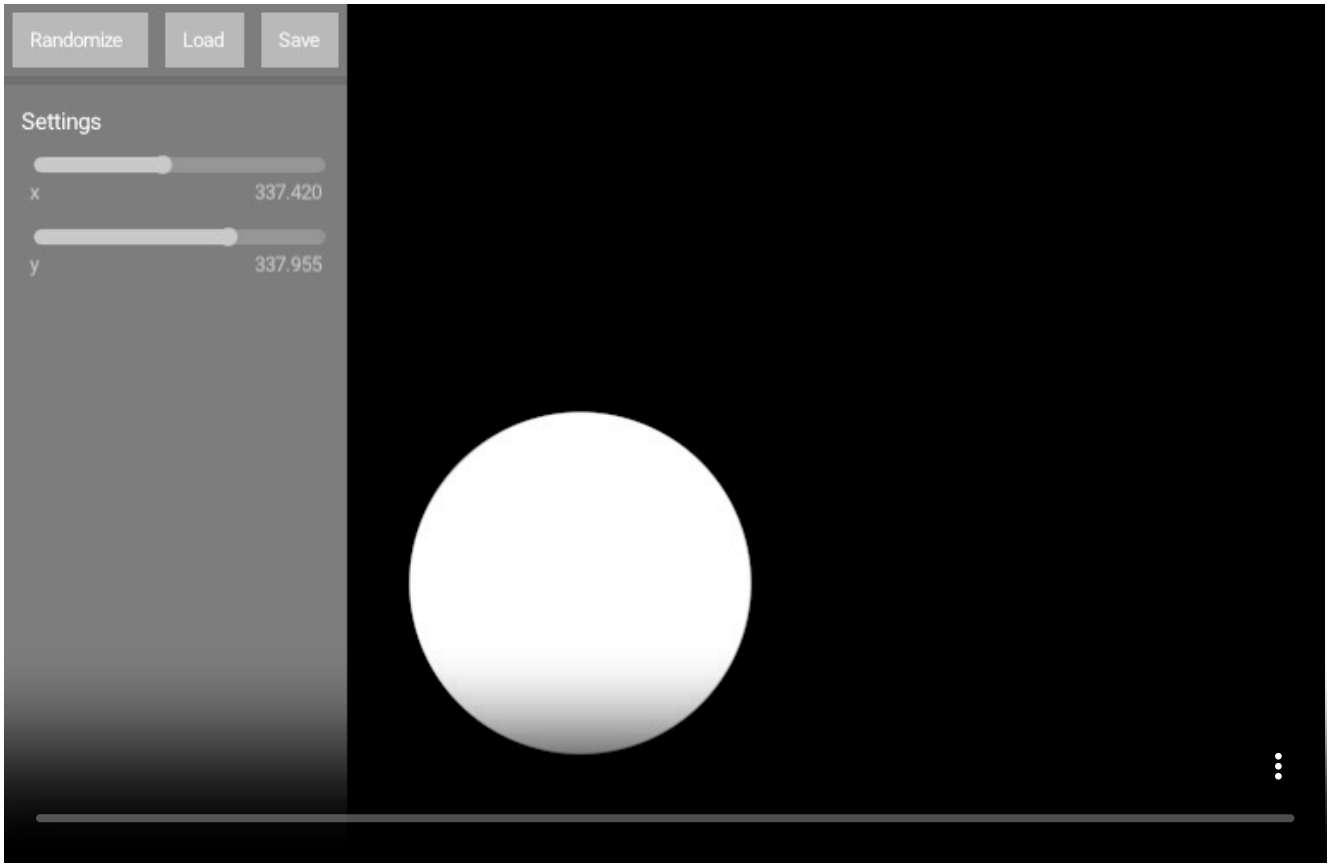
This shows a side panel with 3 buttons: The randomize button can be pressed to randomize the parameters in the sidebar. The load and save button can be used to save the parameter settings to a .json file. Note that the sidebar can be hidden by pressing the F11 button. Compartments can be collapsed by clicking on the compartment header.

Now we want the sidebar put to good use. We can populate the sidebar by adding `orx-parameters` annotated objects. In the following example we create such an annotated `object` and add it.

```
fun main() = application {
    program {
        val gui = GUI()

        val settings = object {
            @DoubleParameter("x", 0.0, 770.0)
            var x: Double = 385.0

            @DoubleParameter("y", 0.0, 500.0)
            var y: Double = 250.0

            // Use `var` for your annotated variables.
            // `val` will produce no UI element!
            @DoubleParameter("z", -10.0, 10.0)
            val z: Double = 0.0
        }

        // -- this is why we wanted to keep a reference to gui
        gui.add(settings, "Settings")

        // -- pitfall: the `extend` has to take place after gui is populated
        extend(gui)
        extend {
```

```
            // -- use our settings
            drawer.circle(settings.x, settings.y, 100.0)
        }
    }
}
```

Link to the full example



We now see that the sidebar is populated with a *settings* compartment that contains the *x* and *y* parameters. Now whenever we move one of the sliders we will also move our circle that we draw using `settings.x` and `settings.y`

Filter workflow

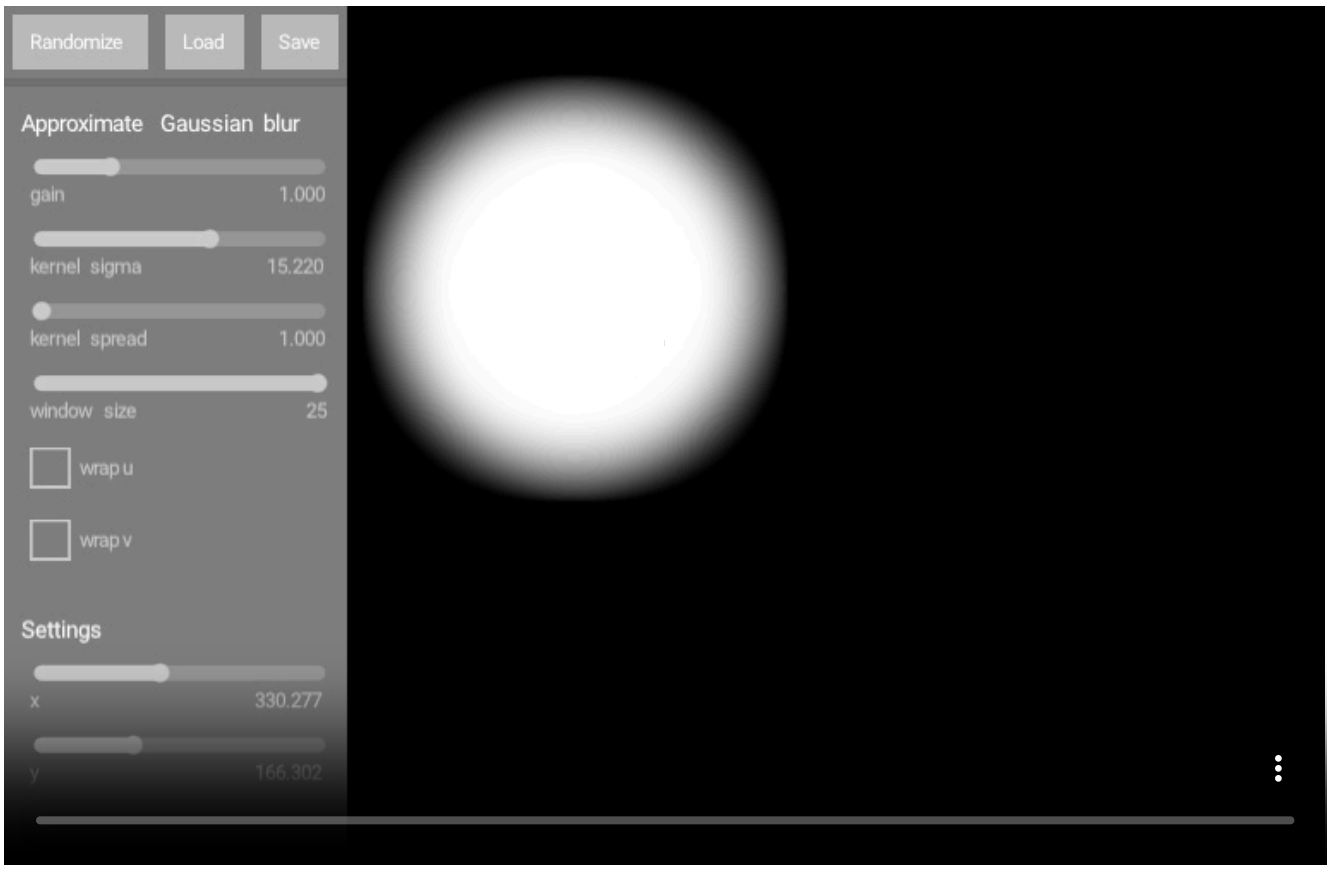
Not only can user objects added to the sidebar, also most of the Filters in `orx-fx` have `orx-parameters` annotations. That means that we can generate quick user interfaces for any of the filters that `orx-fx` provides.

The guide covers filters in the Filter and Post-processing chapter and an index of provided filters can be found in `orx-filter` index

```
fun main() = application {
    program {
        val gui = GUI()
        val settings = object {
            @DoubleParameter("x", 0.0, 770.0)
            var x: Double = 385.0

            @DoubleParameter("y", 0.0, 500.0)
            var y: Double = 250.0
        }
        val rt = renderTarget(width, height) {
```

```
            colorBuffer()
        }

        val filtered = colorBuffer(width, height)
        val blur = ApproximateGaussianBlur()

        gui.add(blur)
        gui.add(settings, "Settings")
        // -- pitfall: the `extend` has to take place after gui is populated
        extend(gui)
        extend {
            drawer.isolatedWithTarget(rt) {
                drawer.clear(ColorRGBa.BLACK)
                // -- use our settings
                drawer.circle(settings.x, settings.y, 100.0)
            }
            blur.apply(rt.colorBuffer(0), filtered)
            drawer.image(filtered)
        }
    }
}
```

Link to the full example



We now see that the sidebar is populated with a *settings* and an *Approximate Gaussian blur* compartment, which contains parameters for the blur filter that we are using.

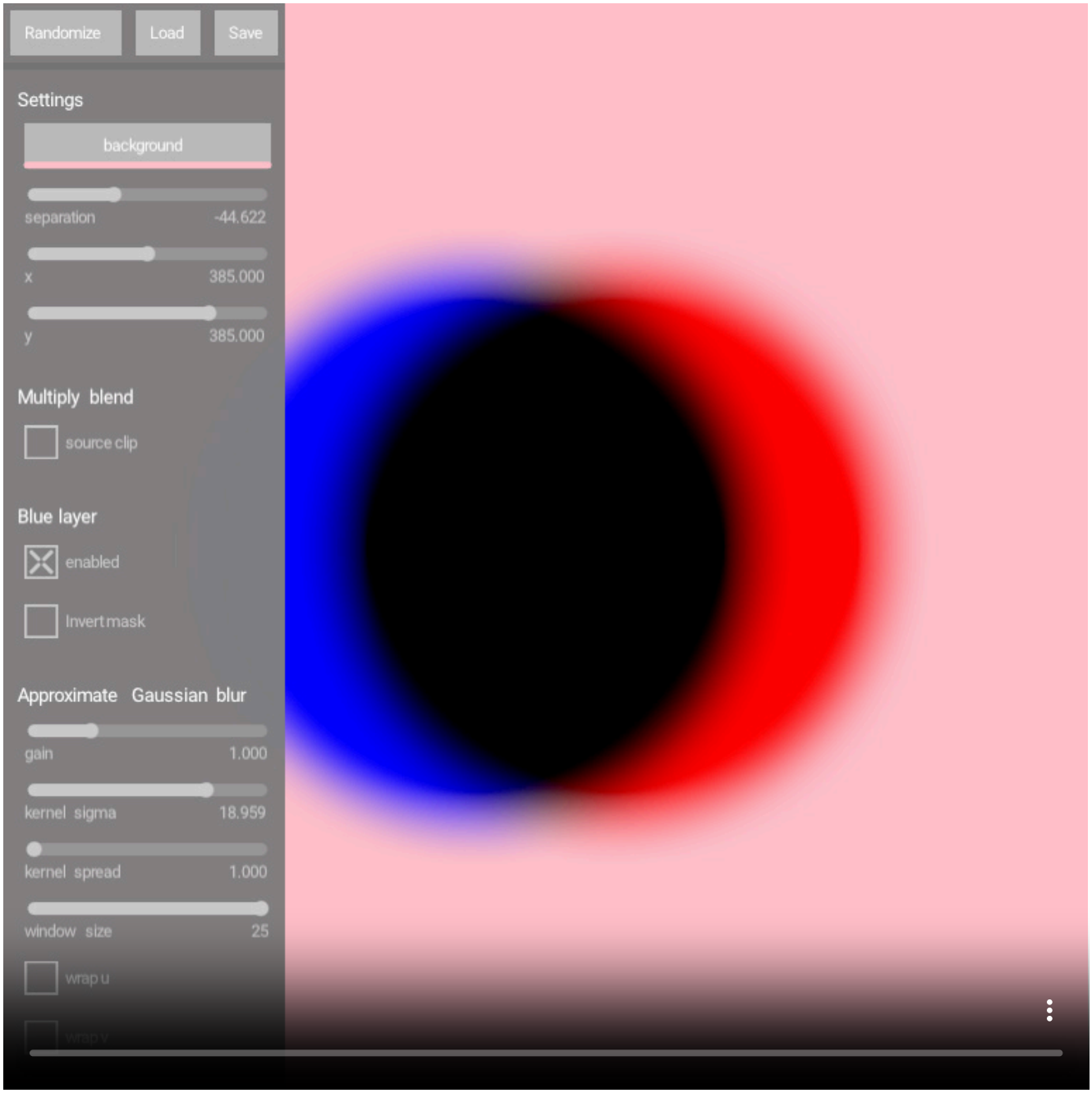
Compositor workflow

`orx-gui` and `orx-compositor` work nicely together. We still need one or more objects for our settings and we can insert any of the blend and post filters in the sidebar as we please. The guide covers `orx-compositor` in the Compositor chapter

```
fun main() = application {
    program {
        val gui = GUI()
        val settings = object {
            @DoubleParameter("x", 0.0, 770.0)
            var x: Double = 385.0

            @DoubleParameter("y", 0.0, 500.0)
            var y: Double = 385.0

            @DoubleParameter("separation", -150.0, 150.0)
            var separation: Double = 0.0

            @ColorParameter("background")
            var background = ColorRGBa.PINK
        }
        gui.add(settings, "Settings")

        // -- create a composite
        val composite = compose {
            layer {
                draw {
                    drawer.clear(settings.background)
                }
            }
            layer {
                layer {
                    draw {
                        drawer.fill = ColorRGBa.RED
                        drawer.circle(settings.x - settings.separation, settings.y, 200.0)
                    }
                }
                layer {
                    draw {
                        drawer.fill = ColorRGBa.BLUE
                        drawer.circle(settings.x + settings.separation, settings.y, 200.0)
                    }
                    // -- add blend to layer and sidebar
                    blend(gui.add(Multiply(), "Multiply blend"))// -- add a layer to the sidebar to toggle it on / off

                }.addTo(gui, "Blue layer")
                // -- add post to layer and sidebar
                post(gui.add(ApproximateGaussianBlur())) {
                    sigma = sin(seconds) * 10.0 + 10.01
                    window = 25
                }
            }
        }
        extend(gui)
```

```
        extend {
            composite.draw(drawer)
        }
    }
}
```

Link to the full example



We now see that the sidebar is populated with a *settings*, *Multiply blend*, *Blue layer*, and an *Approximate gaussian blur* compartment. This creates composites that are easy to tweak.

Parameter annotations

We have seen some of the annotations in the workflow descriptions and we have linked to the `orx-parameters` code before. But we haven't really spent time explaining the annotations until now. Annotations are a Kotlin language feature that is used to attach metadata to code, if you are unfamiliar with them you are encouraged to read the annotations chapter in the Kotlin Language Guide. However, to effectively use the `orx-parameters` annotations basically

all you have to know is where and when to apply the annotations. The annotations are used in front of mutable class/object properties.

Currently `orx-parameters` has a small set of parameter annotations:

DoubleParameter

`DoubleParameter` is used in combination with `Double` types. It takes a label, minimum-value, maximum value, and optional precision and order arguments. `orx-gui` will generate a slider control for annotated properties.

```
val settings = object {
    @DoubleParameter("x", 0.0, 100.0, precision = 3, order = 0)
    var x = 0.0
}
```

IntParameter

`IntParameter` is used in combination with `Int` types. It takes a label, minimum-value, maximum value, and an optional order argument. `orx-gui` will generate a slider control for annotated properties.

```
val settings = object {
    @IntParameter("x", 0, 100, order = 0)
    var x = 0
}
```

ColorParameter

`ColorParameter` is used in combination with `Color` types. It takes an optional order argument. `orx-gui` will generate a color picker control for annotated properties.

```
val settings = object {
    @ColorParameter("color", order = 0)
    var color = ColorRGBa.PINK
}
```

TextParameter

`TextParameter` is used in combination with `String` types. It takes an optional order argument. `orx-gui` will generate a text field control for annotated properties.

```
val settings = object {
    @TextParameter("text", order = 0)
    var text = "default text value"
}
```

BooleanParameter

`BooleanParameter` is used in combination with `Boolean` types. It takes an optional order argument. `orx-gui` will generate a checkbox or toggle control for annotated properties.

```
val settings = object {
    @BooleanParameter("option", order = 0)
    var b = false
}
```

XYParameter

`XYParameter` is used in combination with `Vector2` types. It takes an optional order argument. `orx-gui` will generate a two dimensional pad control for annotated properties.

```
val settings = object {
    @XYParameter("xy", order = 0)
    var xy = Vector2.ZERO
}
```

Vector2Parameter

`Vector2Parameter` is used in combination with `Vector2` types. It takes an optional order argument. `orx-gui` will generate a vertical slider for annotated properties.

```
val settings = object {
    @Vector2Parameter("vector2", order = 0)
    var v2 = Vector2.ZERO
}
```

Vector3Parameter

`Vector3Parameter` is used in combination with `Vector3` types. It takes an optional order argument. `orx-gui` will generate a vertical slider for annotated properties.

```
val settings = object {
    @Vector3Parameter("vector3", order = 0)
    var v3 = Vector3.ZERO
}
```

Vector4Parameter

`Vector4Parameter` is used in combination with `Vector4` types. It takes an optional order argument. `orx-gui` will generate a vertical slider for annotated properties.

```
val settings = object {
    @Vector4Parameter("vector4", order = 0)
    var v4 = Vector4.ZERO
}
```

DoubleListParameter

`DoubleListParameter` is used in combination with a list of `Double`. It takes an optional order argument. `orx-gui` will generate a set of vertical sliders.

```
@DoubleListParameter("Mixer", order = 0)
var mixer = MutableList(5) { 0.5 }
```

OptionParameter

`OptionParameter` is used in combination with an `enum`. It takes an optional order argument. `orx-gui` will generate a dropdown including all options in the enum.

```
enum class Parity { Odd, Even }

@OptionParameter("Parity", order = 0)
var parity = Parity.Odd
```

ActionParameter

`ActionParameter` is a bit of an odd-one-out, it is not used to annotate properties but to annotate 0-argument functions. `orx-gui` will generate a button control that will call the function when clicked.

```
val settings = object {
    @ActionParameter("save", order = 0)
    fun doSave() {
        println("file saved!")
    }
}
```

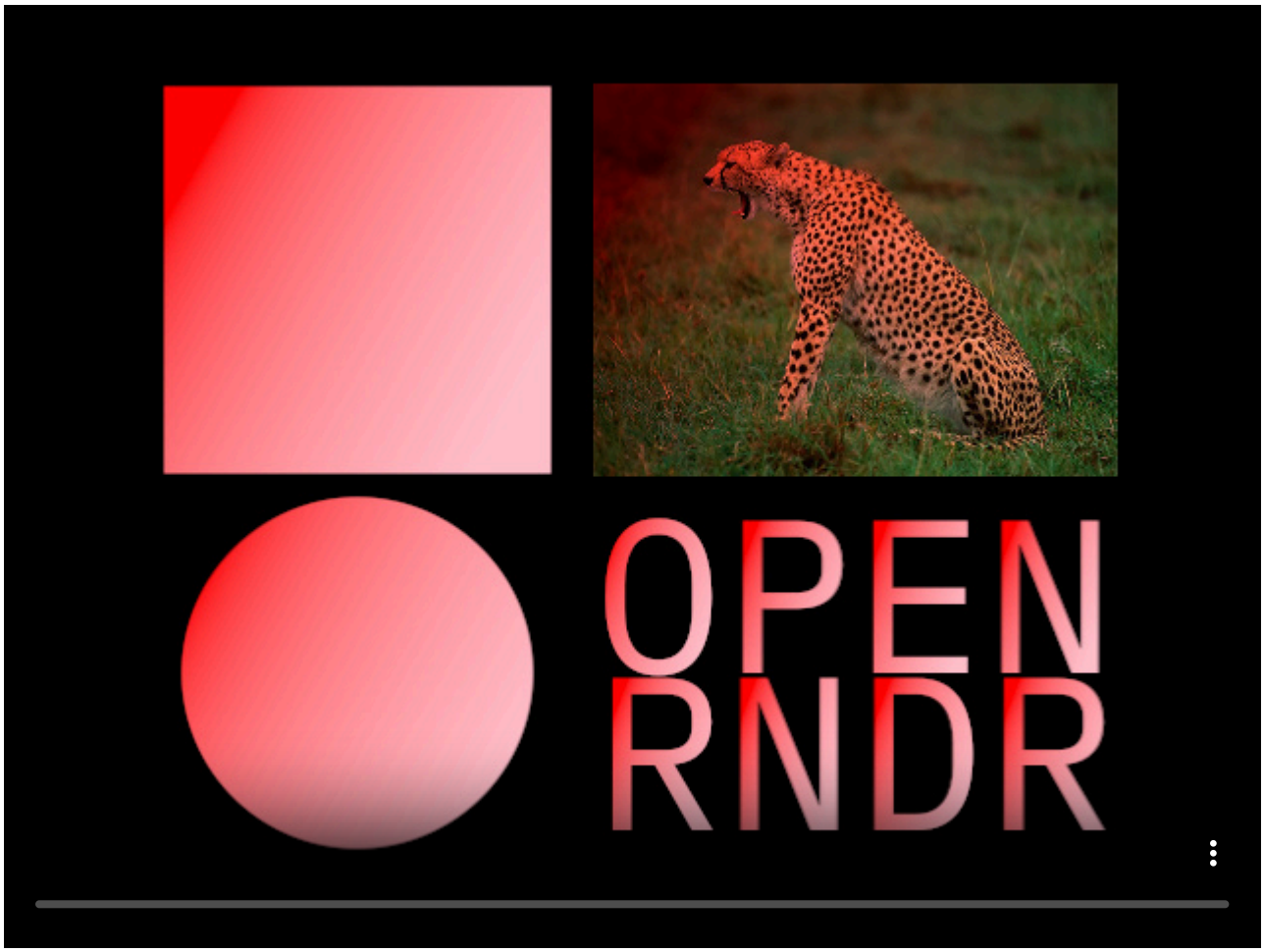
Shade style presets

The `orx-shade-styles` library provides a number of preset shade styles

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-shade-styles` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

linearGradient

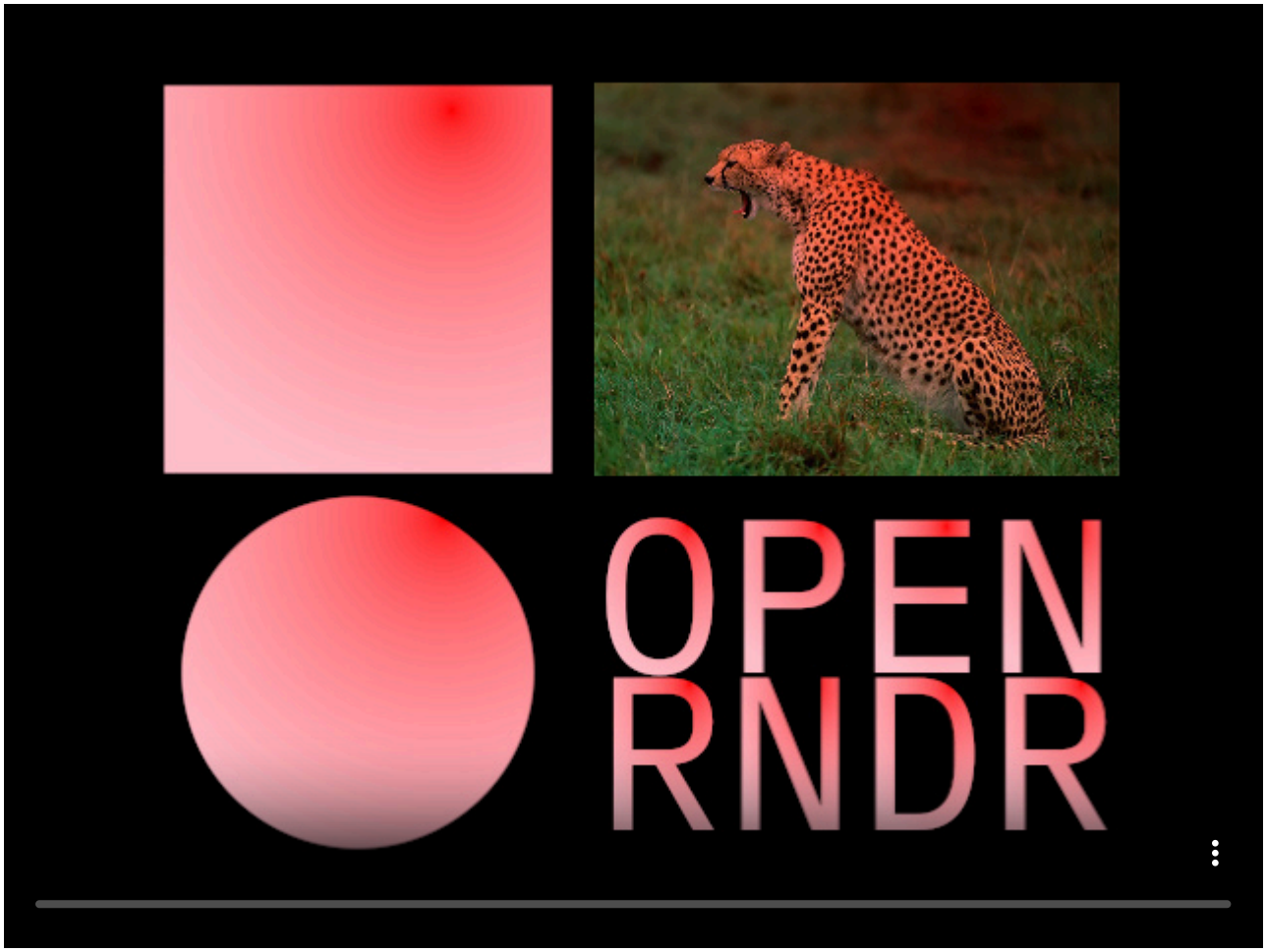



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val font = loadFont("data/fonts/default.otf", 144.0)
        extend {
            drawer.shadeStyle = linearGradient(ColorRGBa.PINK, ColorRGBa.RED, rotation = seconds * 60.0)
            drawer.rectangle(80.0, 40.0, 200.0, 200.0)
            drawer.circle(180.0, 340.0, 90.0)
            drawer.image(image, 300.0, 40.0, 640 * (200 / 480.0), 200.0)
            drawer.fontMap = font
            drawer.text("OPEN", 300.0, 340.0)
            drawer.text("RNDR", 300.0, 420.0)
        }
```

```
    }
}
```

Link to the full example

radialGradient




```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val font = loadFont("data/fonts/default.otf", 144.0)
        extend {
            drawer.shadeStyle = radialGradient(ColorRGBa.RED, ColorRGBa.PINK, length = 0.5, offset =
Vector2(cos(seconds), sin(seconds * 0.5)))
            drawer.rectangle(80.0, 40.0, 200.0, 200.0)
            drawer.circle(180.0, 340.0, 90.0)
            drawer.image(image, 300.0, 40.0, 640 * (200 / 480.0), 200.0)
            drawer.fontMap = font
            drawer.text("OPEN", 300.0, 340.0)
            drawer.text("RNDR", 300.0, 420.0)
        }
    }
}
```

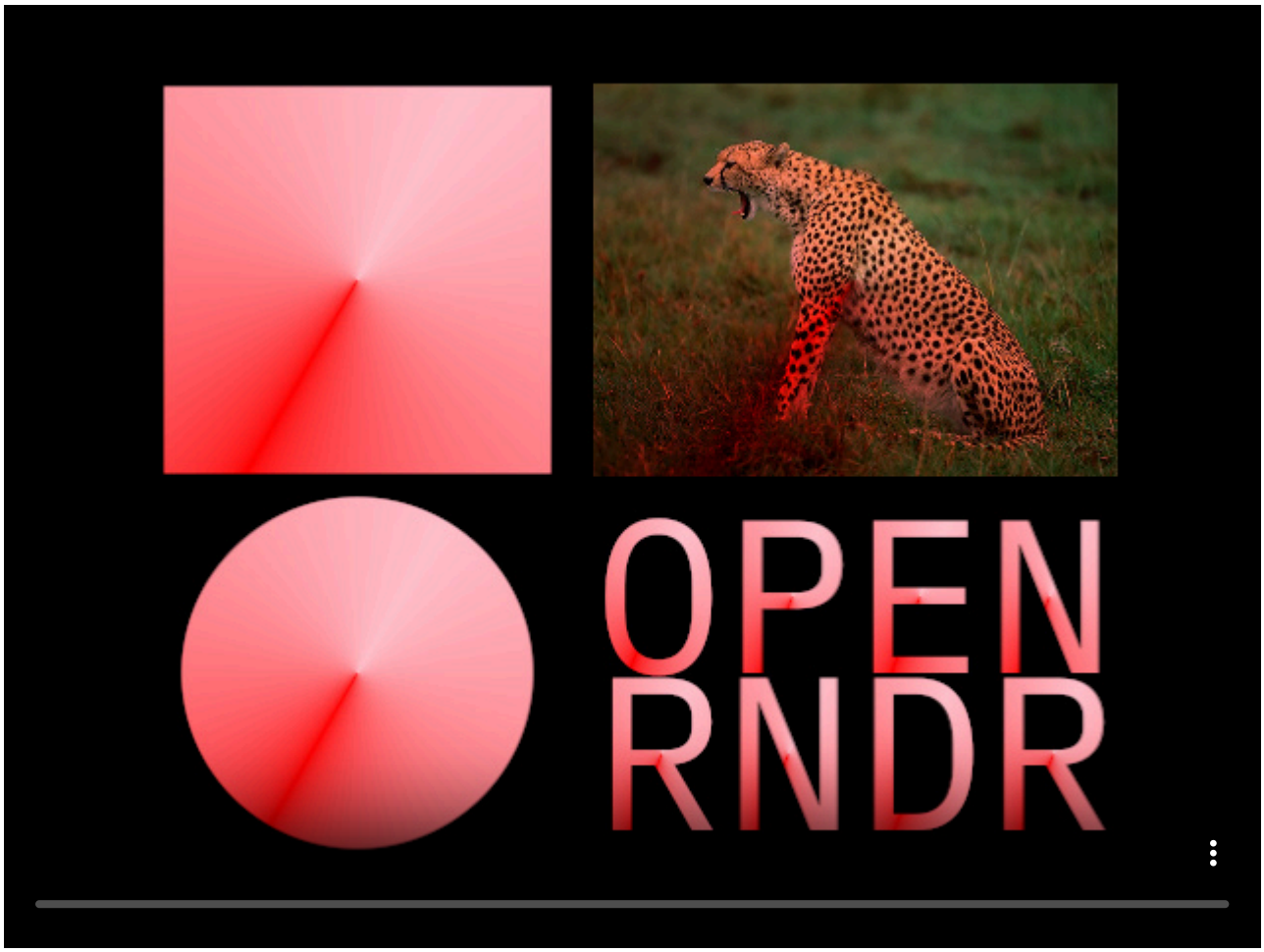
Link to the full example

angularGradient


```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val font = loadFont("data/fonts/default.otf", 144.0)
        extend {
            drawer.shadeStyle = angularGradient(ColorRGBa.RED, ColorRGBa.PINK, rotation = seconds * 60.0)
            drawer.rectangle(80.0, 40.0, 200.0, 200.0)
            drawer.circle(180.0, 340.0, 90.0)
            drawer.image(image, 300.0, 40.0, 640 * (200 / 480.0), 200.0)
            drawer.fontMap = font
            drawer.text("OPEN", 300.0, 340.0)
            drawer.text("RNDR", 300.0, 420.0)
        }
    }
}
```

Link to the full example

halfAngularGradient




```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        val font = loadFont("data/fonts/default.otf", 144.0)
        extend {
            drawer.shadeStyle = halfAngularGradient(ColorRGBa.RED, ColorRGBa.PINK, rotation = seconds * 60.0)
            drawer.rectangle(80.0, 40.0, 200.0, 200.0)
            drawer.circle(180.0, 340.0, 90.0)
            drawer.image(image, 300.0, 40.0, 640 * (200 / 480.0), 200.0)
            drawer.fontMap = font
            drawer.text("OPEN", 300.0, 340.0)
            drawer.text("RNDR", 300.0, 420.0)
        }
    }
}
```

Link to the full example

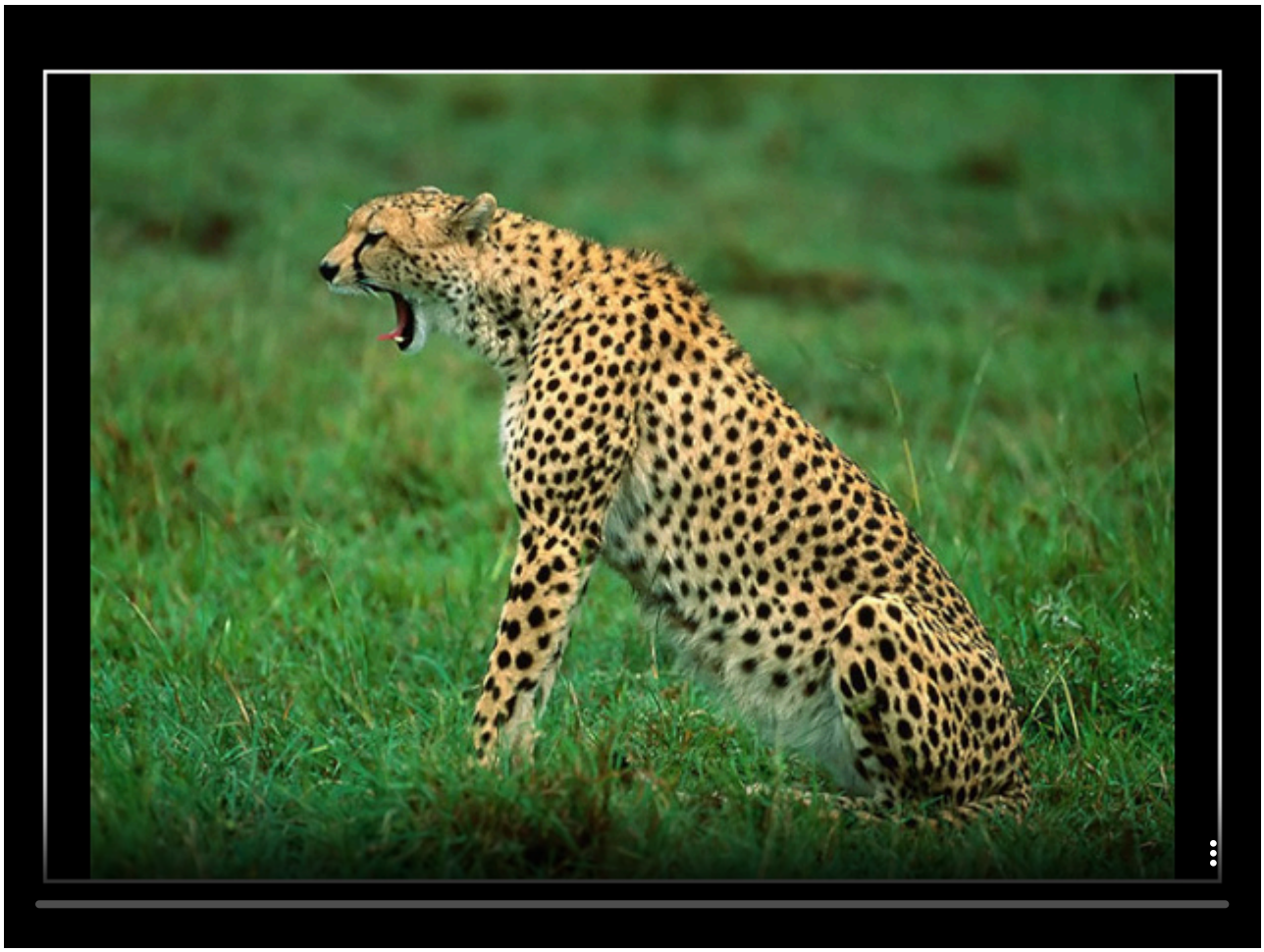
Image fit

`orx-image-fit` provides functionality to making the drawing and placement of images somewhat easier. `orx-image` Fits images in frames with two options, contain and cover, similar to CSS object-fit.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-image-fit` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Contain mode

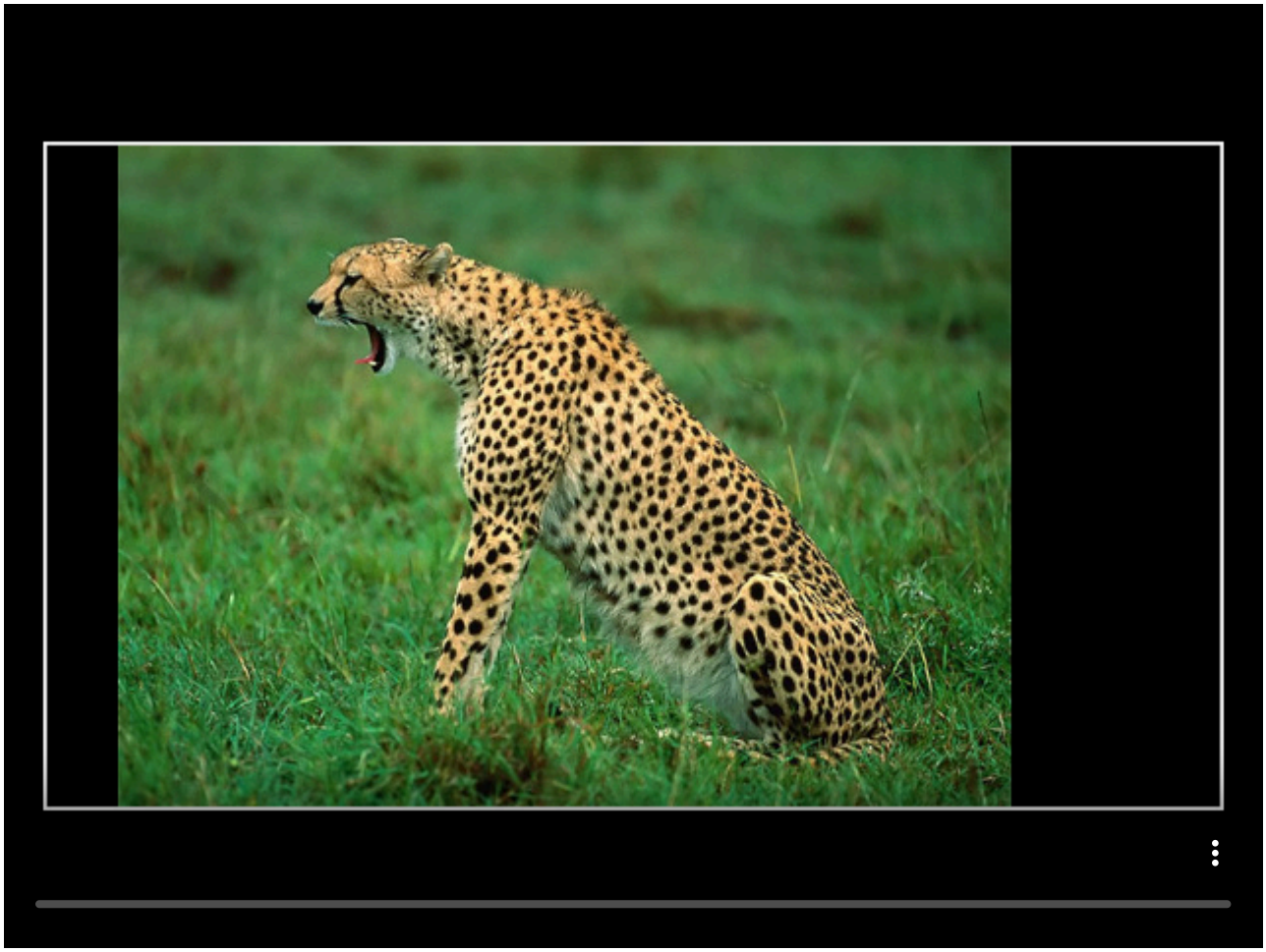


```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        extend {
            val margin = cos(seconds) * 50.0 + 50.0
            drawer.imageFit(image, 20.0, 20.0 + margin / 2, width - 40.0, height - 40.0 - margin, fitMethod =
FitMethod.Contain)
            // -- illustrate the placement rectangle
            drawer.fill = null
            drawer.stroke = ColorRGBa.WHITE
            drawer.rectangle(20.0, 20.0 + margin / 2.0, width - 40.0, height - 40.0 - margin)
        }
```

```
    }
}
```

Link to the full example

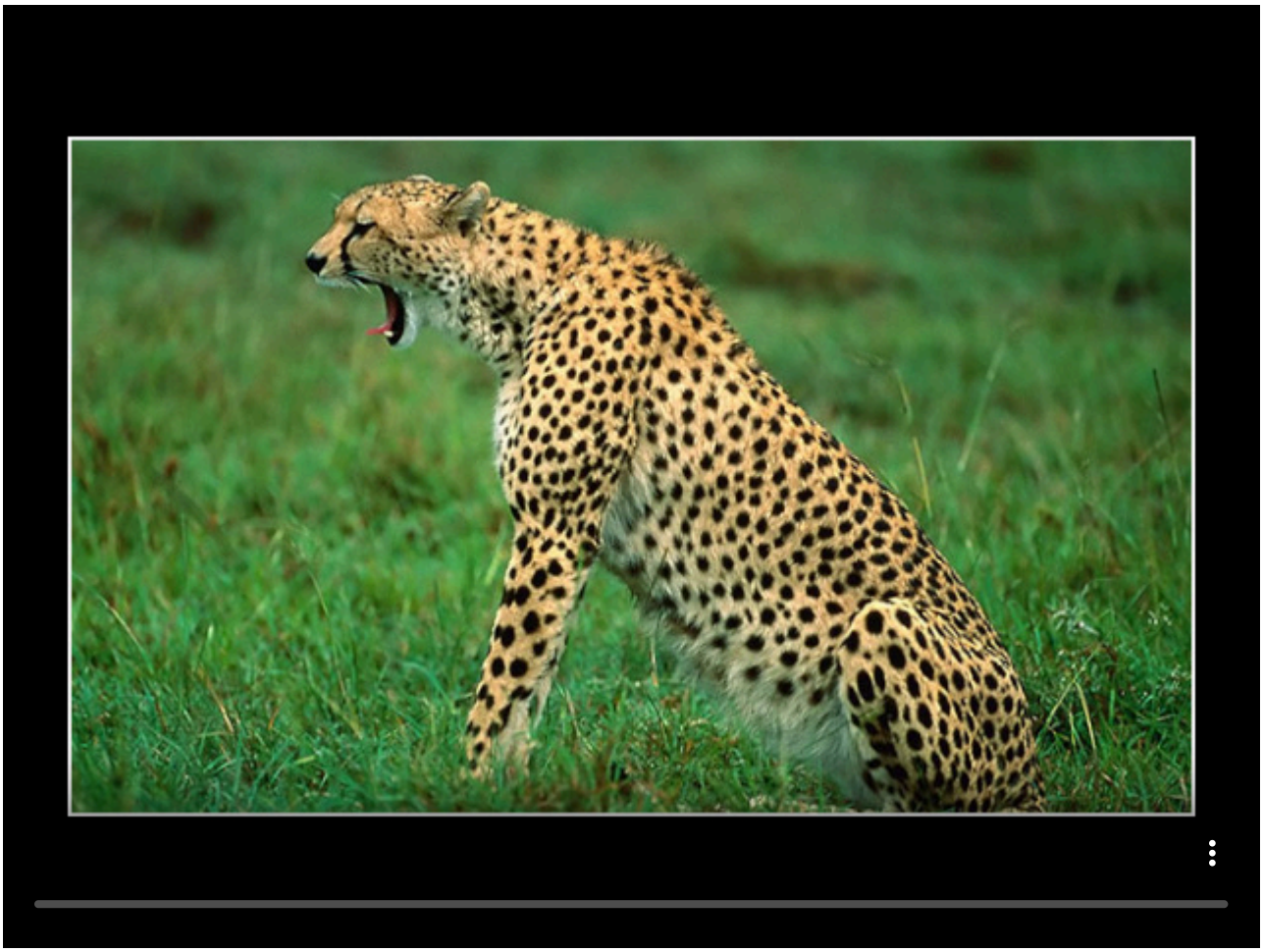
Additionally the placement of the image in the rectangle can be adjusted



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        extend {
            val margin = 100.0
            drawer.imageFit(image, 20.0, 20.0 + margin / 2, width - 40.0, height - 40.0 - margin, horizontalPosition =
cos(seconds) * 1.0, fitMethod = FitMethod.Contain)
            // -- illustrate the placement rectangle
            drawer.fill = null
            drawer.stroke = ColorRGBa.WHITE
            drawer.rectangle(20.0, 20.0 + margin / 2.0, width - 40.0, height - 40.0 - margin)
        }
    }
}
```

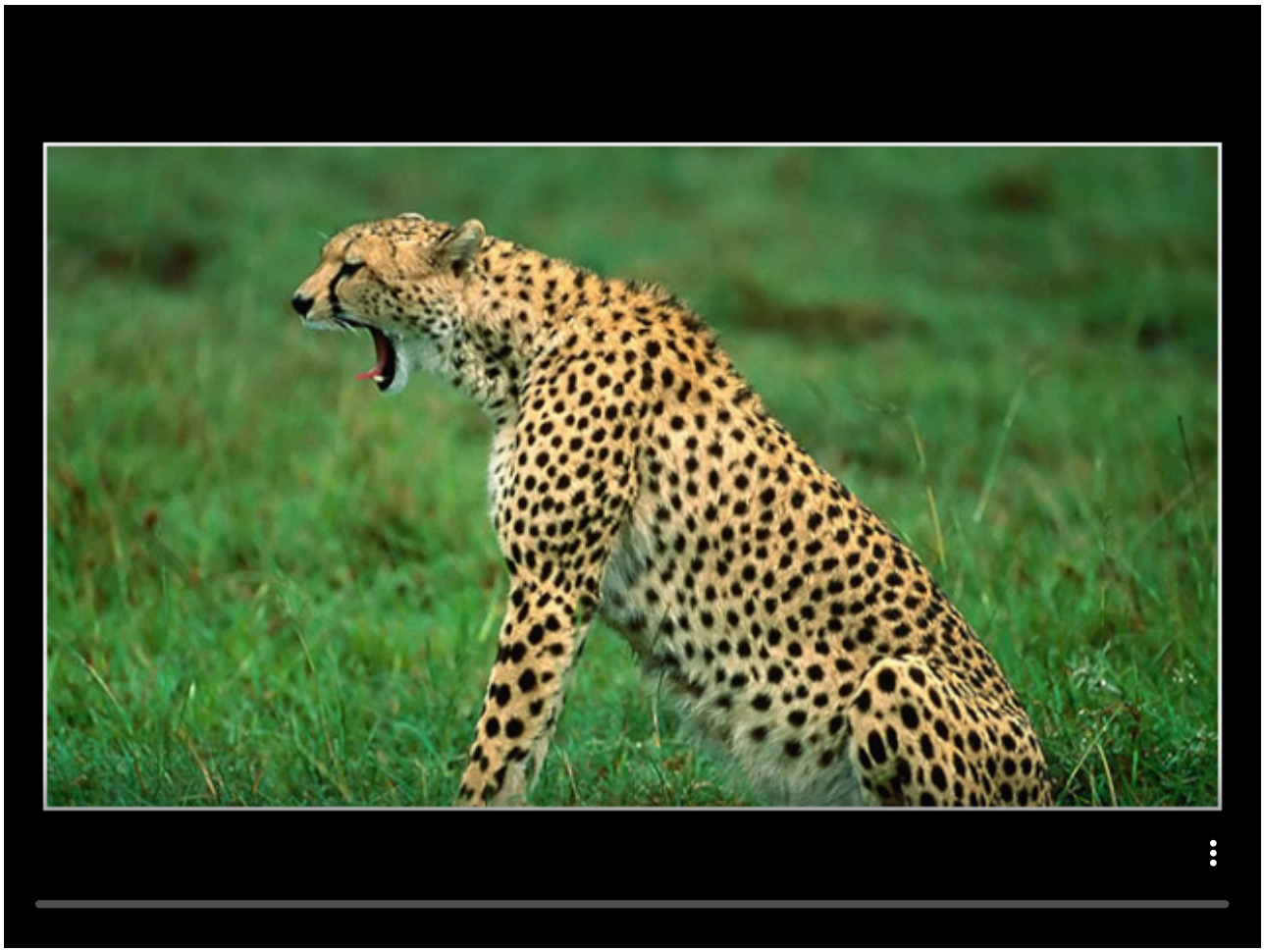
Link to the full example

Cover mode



```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        extend {
            // -- calculate dynamic margins
            val xm = cos(seconds) * 50.0 + 50.0
            val ym = sin(seconds) * 50.0 + 50.0

            drawer.imageFit(image, 20.0 + xm / 2.0, 20.0 + ym / 2, width - 40.0 - xm, height - 40.0 - ym)

            // -- illustrate the placement rectangle
            drawer.fill = null
            drawer.stroke = ColorRGBa.WHITE
            drawer.rectangle(20.0 + xm / 2.0, 20.0 + ym / 2.0, width - 40.0 - xm, height - 40.0 - ym)
        }
    }
}
```

Link to the full example



Additionally the placement of the image in the rectangle can be adjusted

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")
        extend {
            val margin = 100.0
            drawer.imageFit(image, 20.0, 20.0 + margin / 2, width - 40.0, height - 40.0 - margin, verticalPosition =
cos(seconds) * 1.0)

            // -- illustrate the placement rectangle
            drawer.fill = null
            drawer.stroke = ColorRGBa.WHITE
            drawer.rectangle(20.0, 20.0 + margin / 2.0, width - 40.0, height - 40.0 - margin)
        }
    }
}
```

Link to the full example

Poisson fills

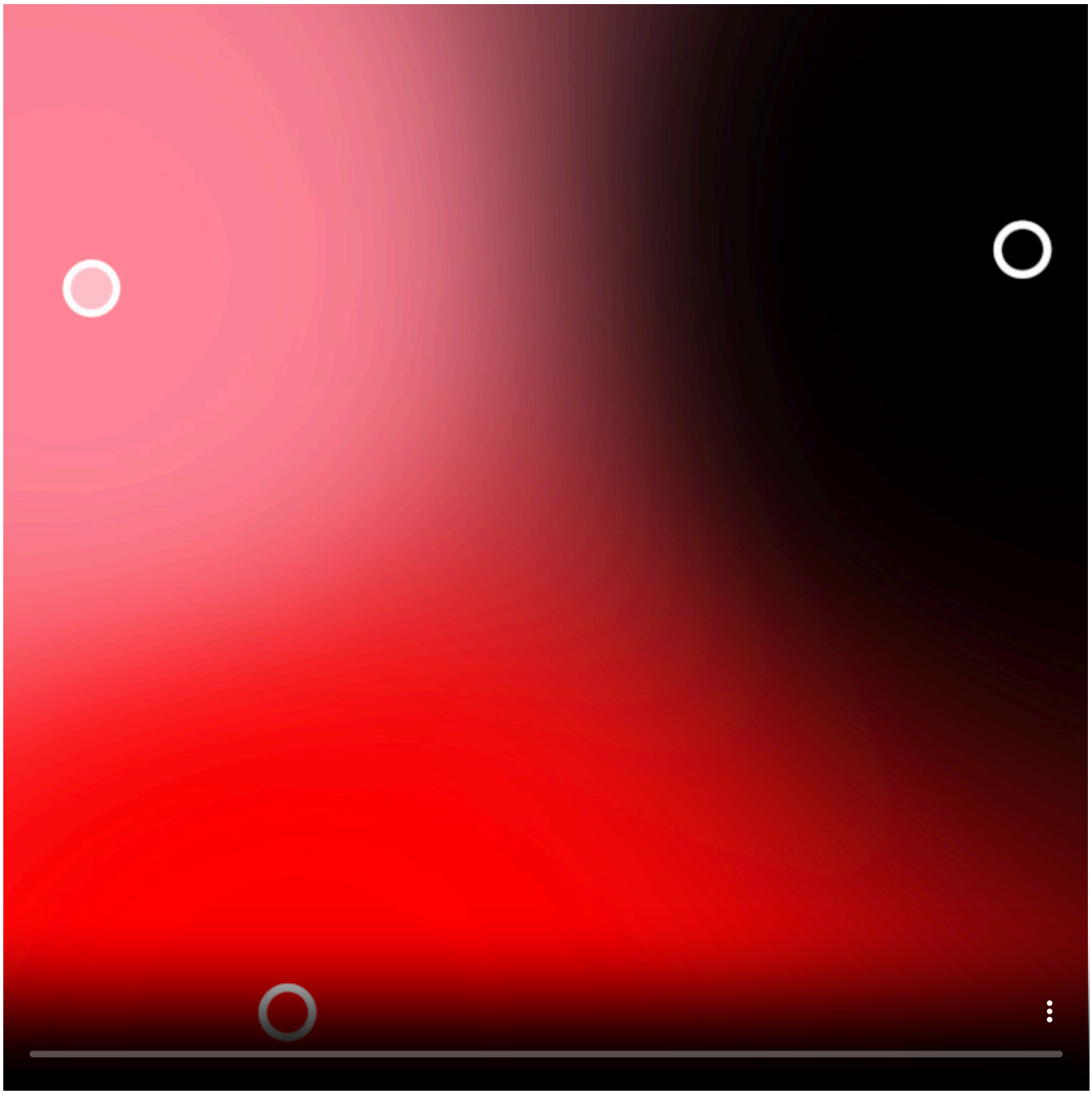
orx-poisson-fills offers tool for GPU-based Poisson operations.

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-poisson-fills` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

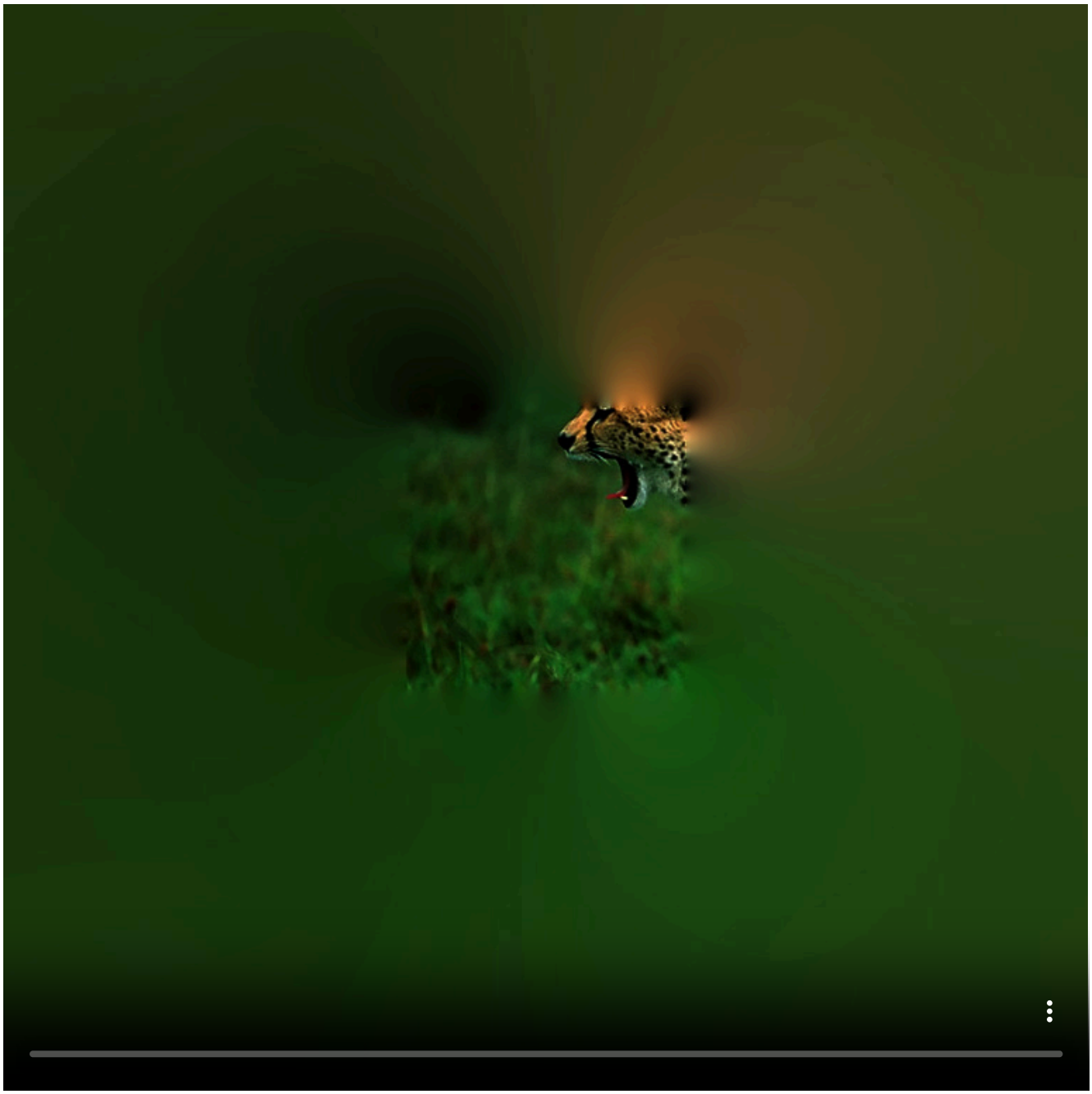
Filling

The `PoissonFill` filter can be used to fill in transparent parts of an image. In the following example we use `orx-compose` to simplify the code a bit. The same results can be achieved using render targets.



```
fun main() = application {
    program {
        val c = compose {
            layer {
                draw {
                    drawer.stroke = null
                    drawer.fill = ColorRGBa.RED
                    drawer.circle((cos(seconds) * 0.5 + 0.5) * width, (sin(seconds * 0.5) * 0.5 + 0.5) * height, 20.0)
                    drawer.fill = ColorRGBa.PINK
                    drawer.circle((sin(seconds * 2.0) * 0.5 + 0.5) * width, (cos(seconds) * 0.5 + 0.5) * height, 20.0)

                    drawer.fill = ColorRGBa.BLACK
                    drawer.circle((sin(seconds * 1.0) * 0.5 + 0.5) * width, (cos(seconds * 2.0) * 0.5 + 0.5) * height,
20.0)
                }
                post(PoissonFill())
            }
            layer {
                // -- an extra layer just to demonstrate where the original data points are drawn
                draw {
                    drawer.stroke = ColorRGBa.WHITE
                    drawer.strokeWeight = 5.0
                    drawer.fill = ColorRGBa.RED
                    drawer.circle((cos(seconds) * 0.5 + 0.5) * width, (sin(seconds * 0.5) * 0.5 + 0.5) * height, 20.0)
                    drawer.fill = ColorRGBa.PINK
                    drawer.circle((sin(seconds * 2.0) * 0.5 + 0.5) * width, (cos(seconds) * 0.5 + 0.5) * height, 20.0)

                    drawer.fill = ColorRGBa.BLACK
                    drawer.circle((sin(seconds * 1.0) * 0.5 + 0.5) * width, (cos(seconds * 2.0) * 0.5 + 0.5) * height,
20.0)
                }
            }
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

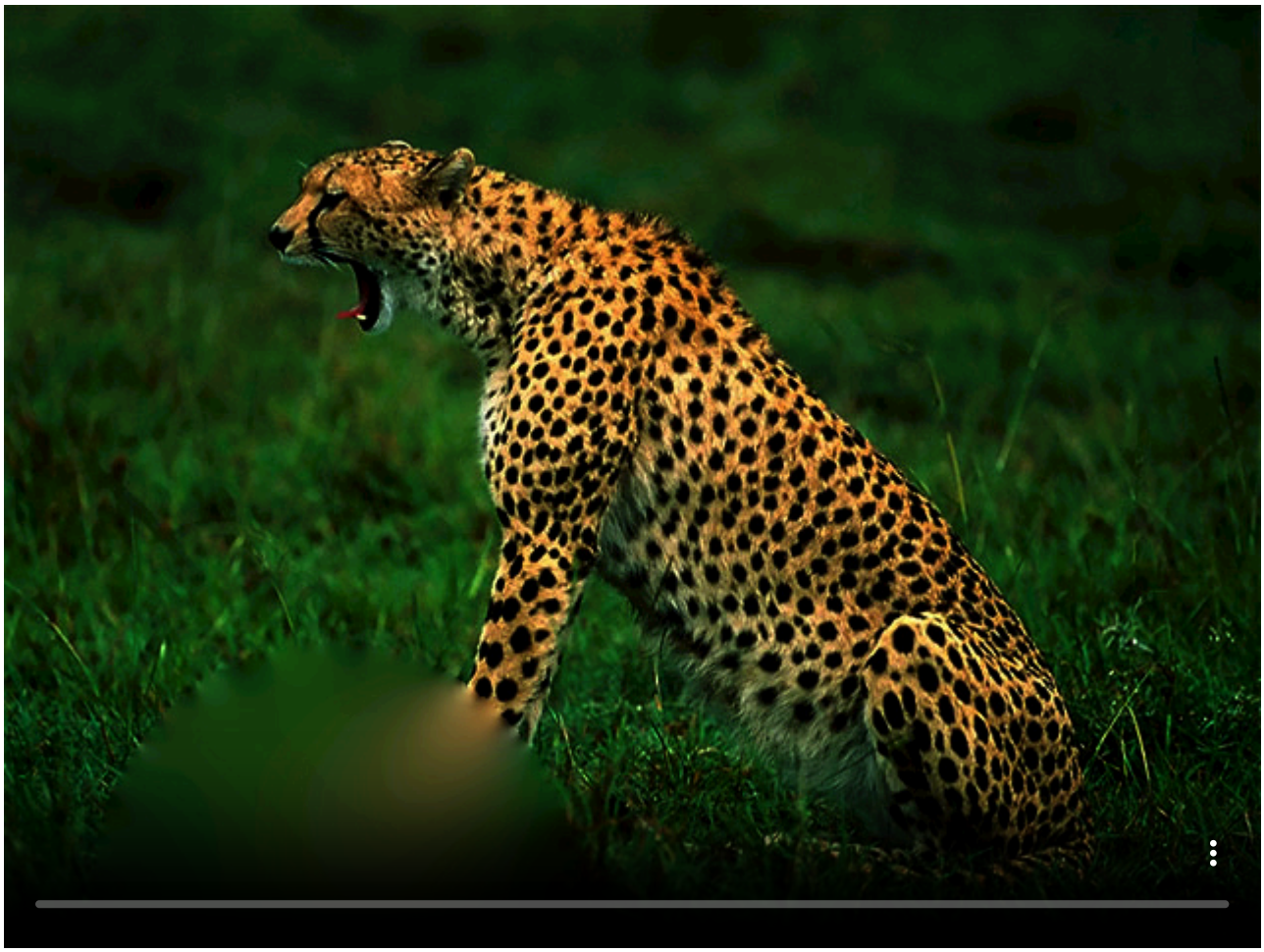
Link to the full example



```
fun main() = application {
    program {
        val c = compose {
            layer {
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.image(image, Rectangle((cos(seconds) * 0.5 + 0.5) * 100.0, (sin(seconds) * 0.5 + 0.5) *
100.0, 200.0, 200.0), Rectangle(width / 2 - 100.0, height / 2.0 - 100.0, 200.0, 200.0))
                }
                post(PoissonFill())
            }
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

Link to the full example

Blending



```
fun main() = application {
    program {
        val c = compose {
            layer {
                val image = loadImage("data/images/cheeta.jpg")
                draw {
                    drawer.image(image)
                }
            }
            layer {
                draw {
                    drawer.stroke = null
                    drawer.fill = ColorRGBa.BLACK
                    drawer.circle((cos(seconds) * 0.5 + 0.5) * width, (sin(seconds * 0.5) * 0.5 + 0.5) * height,
120.0)
                }
                blend(PoissonBlend())
            }
        }
        extend {
            c.draw(drawer)
        }
```

```
    }
}
```

Link to the full example

```
fun main() = application {
    program {
        val image = loadImage("data/images/cheeta.jpg")

        val c = compose {
            layer {
                draw {
                    drawer.image(image)
                }
            }
            layer {
                draw {
                    drawer.stroke = ColorRGBa.GRAY
                    drawer.fill = null
                    drawer.strokeWeight = 40.0
                    drawer.circle((cos(seconds) * 0.5 + 0.5) * width, (sin(seconds * 0.5) * 0.5 + 0.5) * height,
120.0)
                }
                post(LumaOpacity()) {
                    this.backgroundLuma = 0.25
                }
                blend(PoissonBlend())
```

```
            }
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

Link to the full example

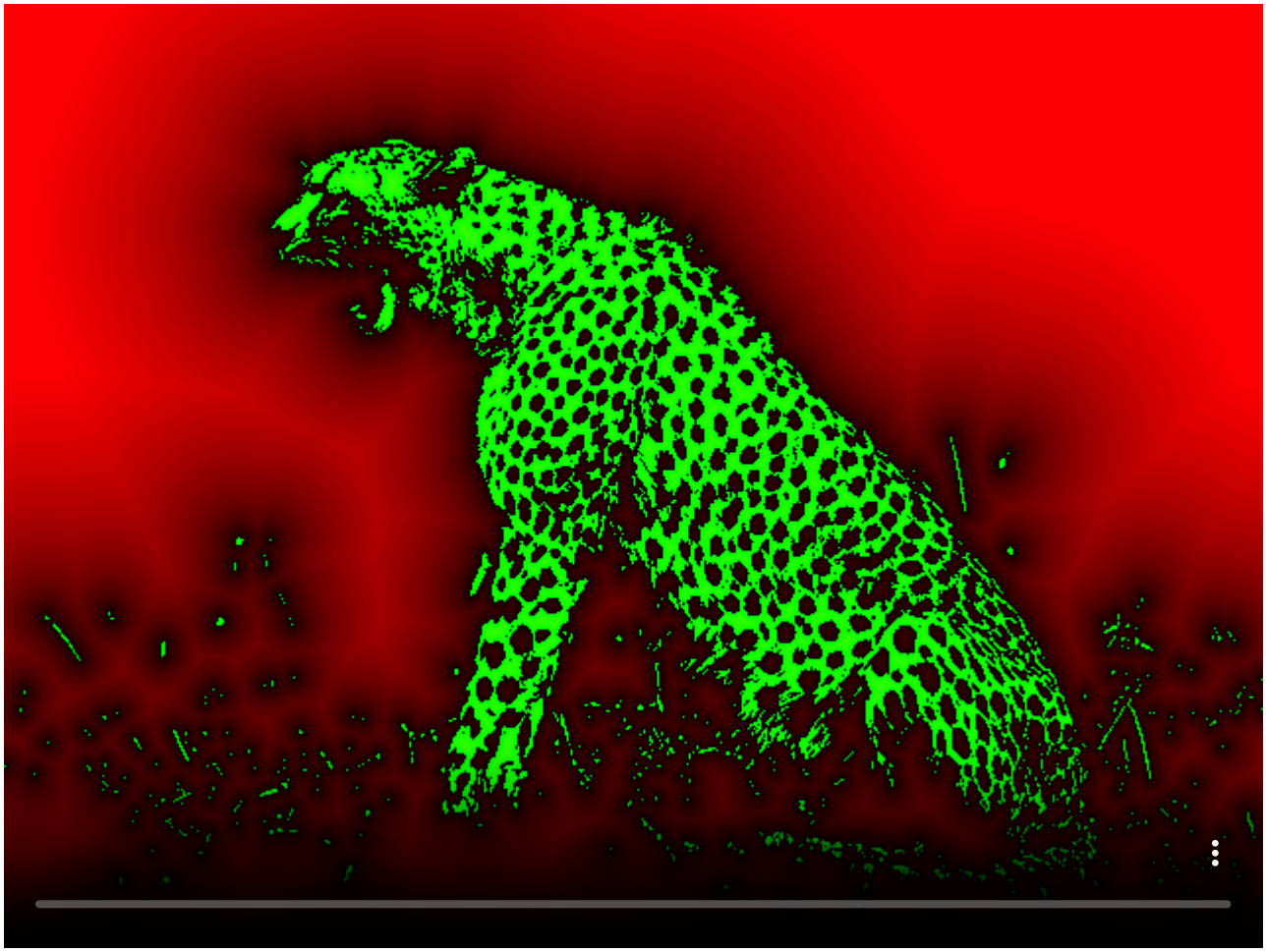
edit on GitHub

Distance fields using orx-jumpflood

Prerequisites

Assuming you are working on an `openrndr-template` based project, all you have to do is enable `orx-jumpflood` in the `orxFeatures` set in `build.gradle.kts` and reimport the gradle project.

Distance field visualization



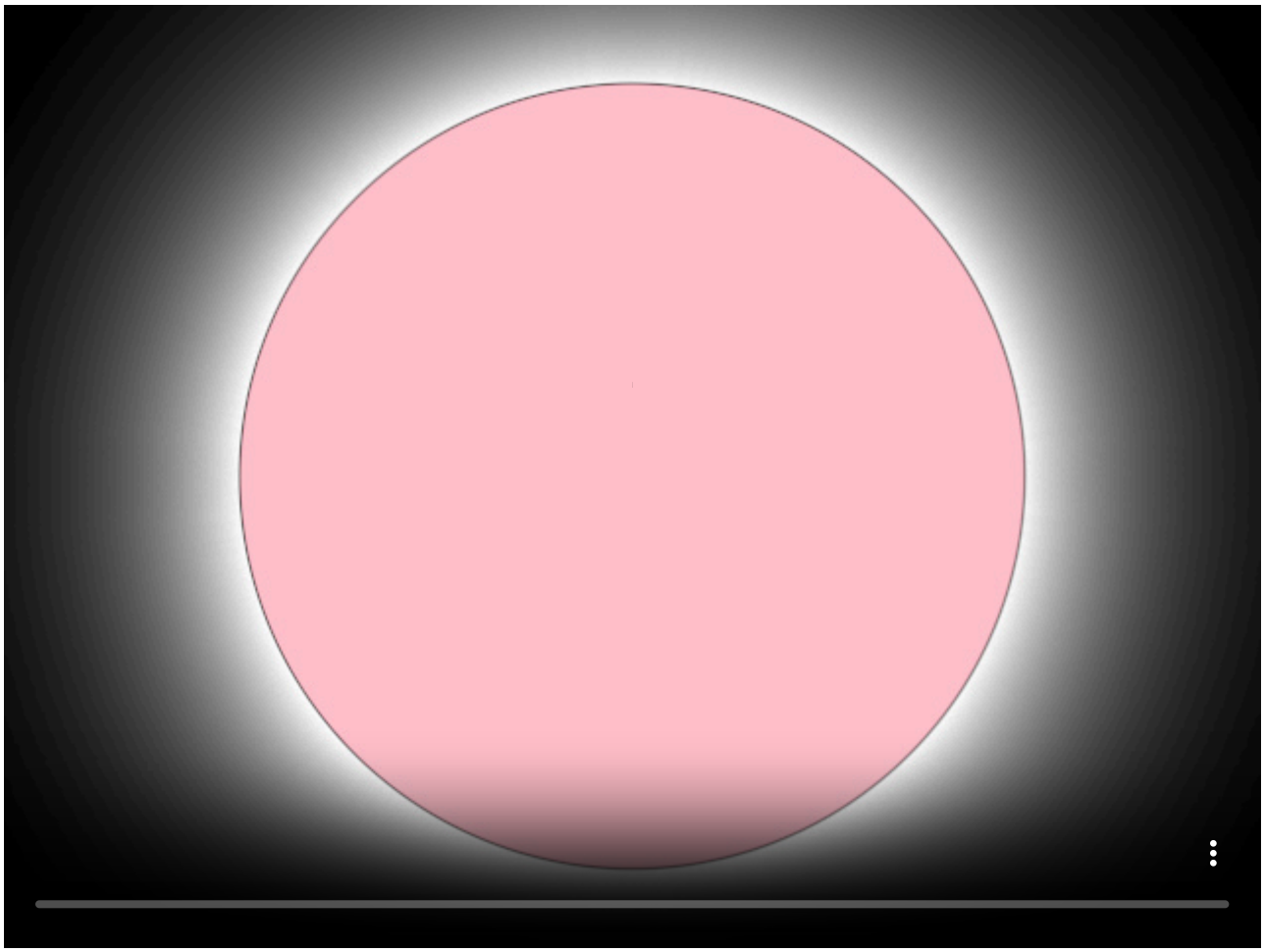
```
fun main() = application {
    program {

        val image = loadImage("data/images/cheeta.jpg")

        val c = compose {
            layer {
                colorType = ColorType.FLOAT32

                draw {
                    drawer.image(image)
                }
                post(DistanceField()) {
                    threshold = cos(seconds) * 0.5 + 0.5
```

```
                    distanceScale = 0.008
                }
            }
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

Link to the full example

Outer glow

orx-jumpflood comes with a filter that creates Photoshop-style outer glow effect.



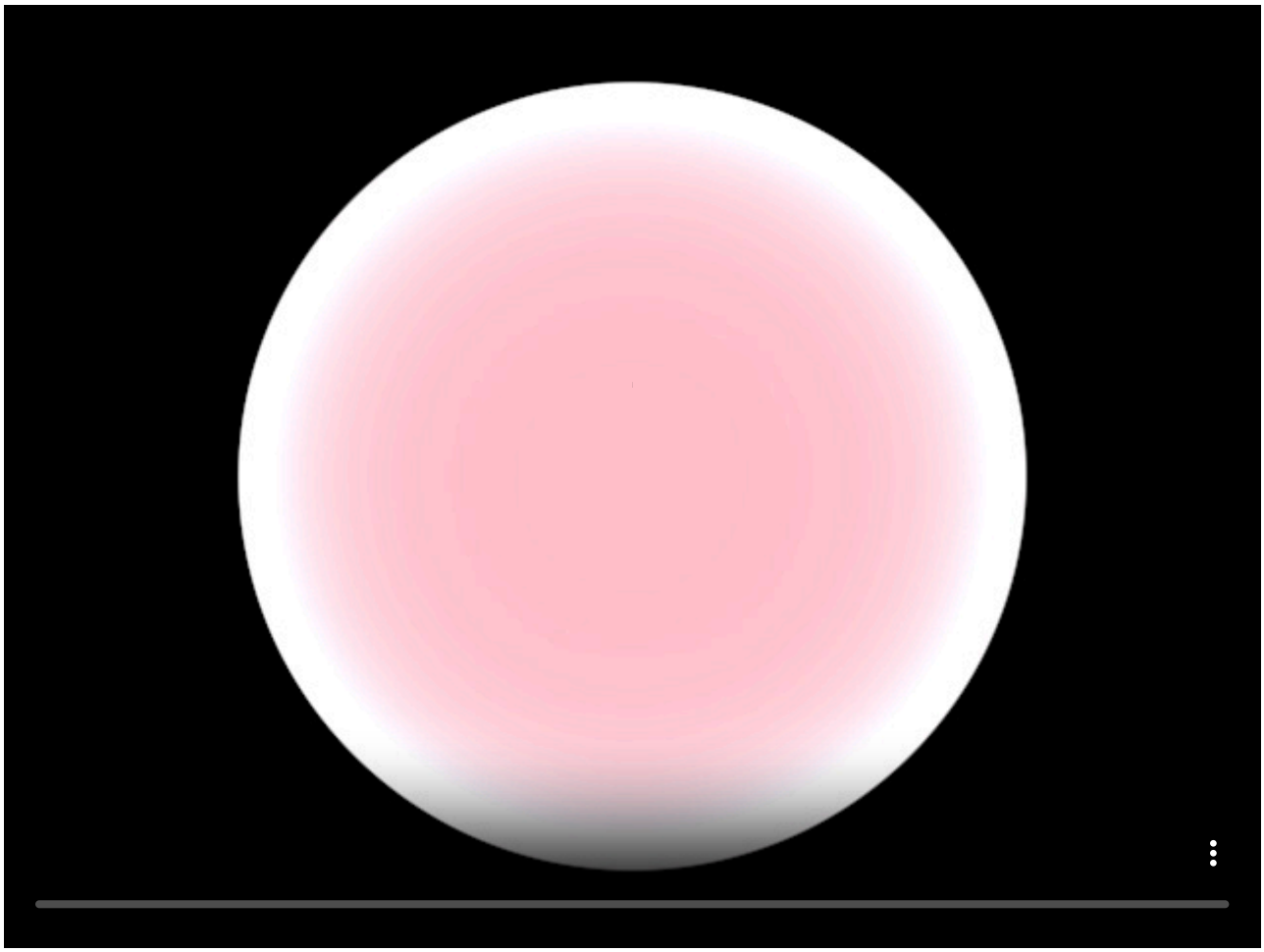
```
fun main() = application {
    program {
        val c = compose {
            layer {
                draw {
                    drawer.fill = ColorRGBa.PINK
                    drawer.circle(width / 2.0, height / 2.0, 200.0)
                }
                post(OuterGlow()) {
                    this.width = (cos(seconds) * 0.5 + 0.5) * 100.0
                }
            }
```

```
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

Link to the full example

Inner glow

Similar to the outer glow effect, but the glow is placed in the inside of the shape.

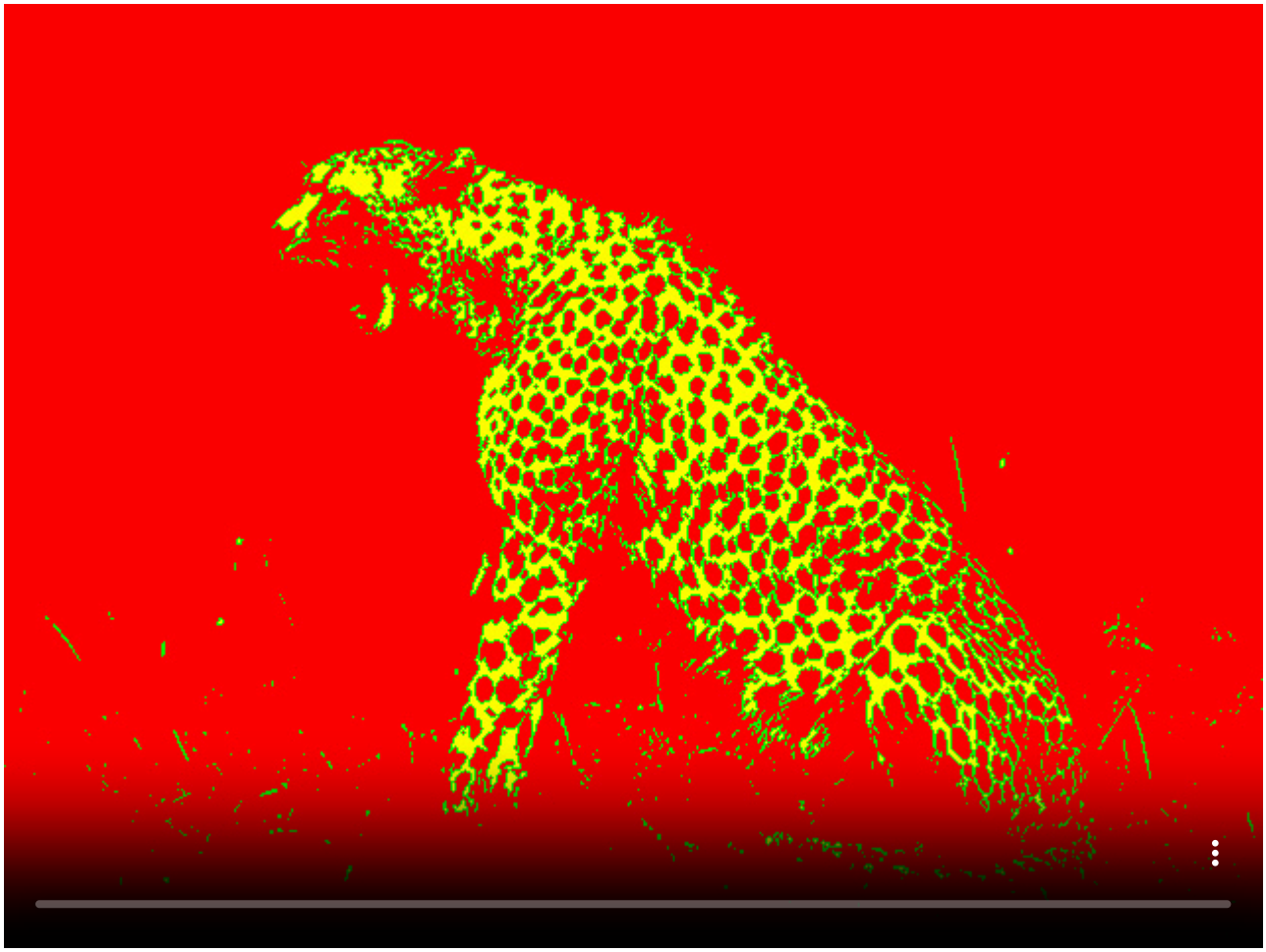


```
fun main() = application {
    program {
        val c = compose {
            layer {
                draw {
                    drawer.fill = ColorRGBa.PINK
                    drawer.circle(width / 2.0, height / 2.0, 200.0)
                }
                post(InnerGlow()) {
                    this.width = (cos(seconds) * 0.5 + 0.5) * 100.0
                }
            }
        }
        extend {
            c.draw(drawer)
```

```
        }
    }
}
```

Link to the full example

Sampling distance



```
fun main() = application {
    program {

        val image = loadImage("data/images/cheeta.jpg")

        val c = compose {
            // -- make sure accumulation is done in float32
            colorType = ColorType.FLOAT32
            layer {
                colorType = ColorType.FLOAT32
                draw {
                    drawer.image(image)
                }
                post(DistanceField()) {
                    threshold = cos(seconds) * 0.5 + 0.5
                    distanceScale = 1.0
                }
            }
            layer {
```

```
                draw {
                    // -- use the accumulation buffer to get the distance field
                    accumulation?.let {
                        val s = it.shadow
                        s.download()
                        drawer.fill = ColorRGBa.PINK
                        drawer.stroke = null
                        for (y in 0 until height step 10) {
                            for (x in 0 until width step 10) {
                                val distance = s[x, y].r
                                drawer.circle(x * 1.0, y * 1.0, distance * 0.05)
                            }
                        }
                        // -- clear the accumulated contents
                        it.fill(ColorRGBa.TRANSPARENT)
                    }
                }
            }
        }
        extend {
            c.draw(drawer)
        }
    }
}
```

Link to the full example

Logging

OPENRNDR uses kotlin-logging, which is a Kotlin flavoured wrapper around slf4j, to log its internal workings. Log messages are displayed in the IDE's console and saved to the `application.log` file.

Configure logging

In `build.gradle.kts` we can modify the `applicationLogging` variable.

- `Logging.NONE` to disable logging.
- `Logging.SIMPLE` to display monochrome log messages in the console.
- `Logging.FULL` to display coloured log messages in the console to distinguish their types.

In `src/main/resources/log4j2.yaml` we can change the logging level. To make it verbose we can replace `level: info` with `level: all`.

Exception handling

It is possible to change how exception errors are presented by IntelliJ IDEA: Open the `Run > Edit Configurations` menu and make sure the `VM Options` text field contains `-Dorg.openrndr.exceptions=JVM`. This can sometimes help figure out why a program is crashing.

Crashing shaders

If a ShadeStyle crashes, a `ShaderError.glsl` file is created at the root of the project. The content of the file is the actual shader program OPENRNDR tried to use. Studying this program can help figure out why shaders fail. A common reason is using incorrect names for methods, uniforms or variables.

Debugging video exporting

When a video file is produced, a `ffmpegOutput.txt` is created at the root of the project. Studying this file can help diagnose problems with video exporting.

Enabling OpenGL debug messages

If your graphics hardware and drivers support OpenGL debug contexts you can open the `Run > Edit Configurations...` menu in IntelliJ and make sure the `VM Options` text field contains `-Dorg.openrndr.gl3.debug=true` to enable the debug messages.

Using RenderDoc

RenderDoc is a graphics debugger currently available for Vulkan, D3D11, D3D12, OpenGL, and OpenGL ES development on Windows, Linux, Android, Stadia, and Nintendo Switch™.

This post explains how to use RenderDoc with OPENRNDR.

Advanced topics

TABLE OF CONTENTS

Application Flow

This section covers default and alternate application flow.

Default application flow

The default application flow aims at single window application. For clarity we list the skeleton for an OPENRNDR program below.

```
fun main() {
    // -- define an application
    application {
        // -- at this point there is no window or graphical context
        // -- attempting to work with graphics resources will lead to errors.
        // -- configure the application window
        configure {
            width = 770
            height = 578
        }
        // -- define the program
        program {
            // -- at this point there is a graphical context

            // -- extend the program with drawing logic
            extend {}
        }
    }
}
```

Start-up or configuration dialogs

In some scenarios it is desirable to present a simple dialog before the main program commences, for example in the case you want the user to to configure resolution and fullscreen settings. While OPENRNDR natively doesn't offer the tools to create user interfaces it does offer the functionality to create a window to host a configuration dialog.

```
fun main() {
    val settings = object {
        var width: Int = 640
    }

    // -- configuration
    application {
        program {// -- somehow get values in the settings object
        }
    }

    // -- application blocks until the window is closed
    application {
        // -- configure using the settings object
```

```
        configure {
            width = settings.width
        }
        program {
        }
    }
}
```

edit on GitHub

Presentation Control

OPENRNDR programs can use any of the two presentation modes.

The default mode is automatic presentation, the `draw` method is called as often as possible. The other mode is manual presentation, in which it is the developer's responsibility to request `draw` to be called.

Setting the presentation mode

The presentation mode can be set and changed at run-time.

```
fun main() = application {
    program {
        window.presentationMode = PresentationMode.AUTOMATIC
    }
}
```

Using the manual presentation mode

The presentation mode is set to manual, a request to draw can be made using `window.requestDraw`.

In the following example `draw()` is only called after a mouse click.

```
fun main() = application {
    program {
        window.presentationMode = PresentationMode.MANUAL
        mouse.buttonDown.listen {
            window.requestDraw()
        }

        extend {
            drawer.clear(ColorRGBa.PINK.shade(Math.random()))
        }
    }
}
```

Note that in manual presentation mode `draw()` is still called when the window is resized.

Headless Applications

OPENRNDR can be ran in headless mode on machines that have EGL support. Using the EGL backed headless mode Programs can be ran without active graphical environment. This makes it for example possible to use OPENRNDR to create command line utilities that can be run in a SSH session or as a background service.

The default backend on the JVM is GLFW, in order to use headless mode you will need to run OPENRNDR with EGL by adding `-Dorg.openrndr.application=EGL` to the VM arguments in the launch configuration.

Limitations

PLATFORMS

Only supported on platforms that support EGL for context creation; which is Linux.

BACKBUFFER

Headless applications cannot draw on the backbuffer, because there is no backbuffer. In order to draw you need to create a RenderTarget and draw on it. Render target contents can easily be saved to file, or rendered to video.

MOUSE AND KEYBOARD EVENTS

Headless applications cannot handle mouse or keyboard input.

COLORBUFFERLOADER

Headless applications (currently) cannot create secondary/shared contexts and as such ColorBufferLoader does not work.

Low-level Drawing

This text is intended for developers looking to provide drawing functionality that is fully independent of the `Drawer` class. Developers that desire to do so will essentially use the same building blocks that OPENRNDR uses internally.

A good reference for low-level drawing is the `Filter` class and it is encouraged to look at its source code as it provides a good idea of how all the low-level components interact and its implementation is fully independent from `Drawer`.

All low-level drawing is performed using OPENRNDR's `Driver` class. This class provides a handful of functions for drawing. Below you will find a list of the essential ones.

- `Driver.setState()`
- `Driver.drawVertexBuffer()`
- `Driver.drawIndexedVertexBuffer()`
- `Driver.drawInstances()`
- `Driver.drawIndexedInstances()`

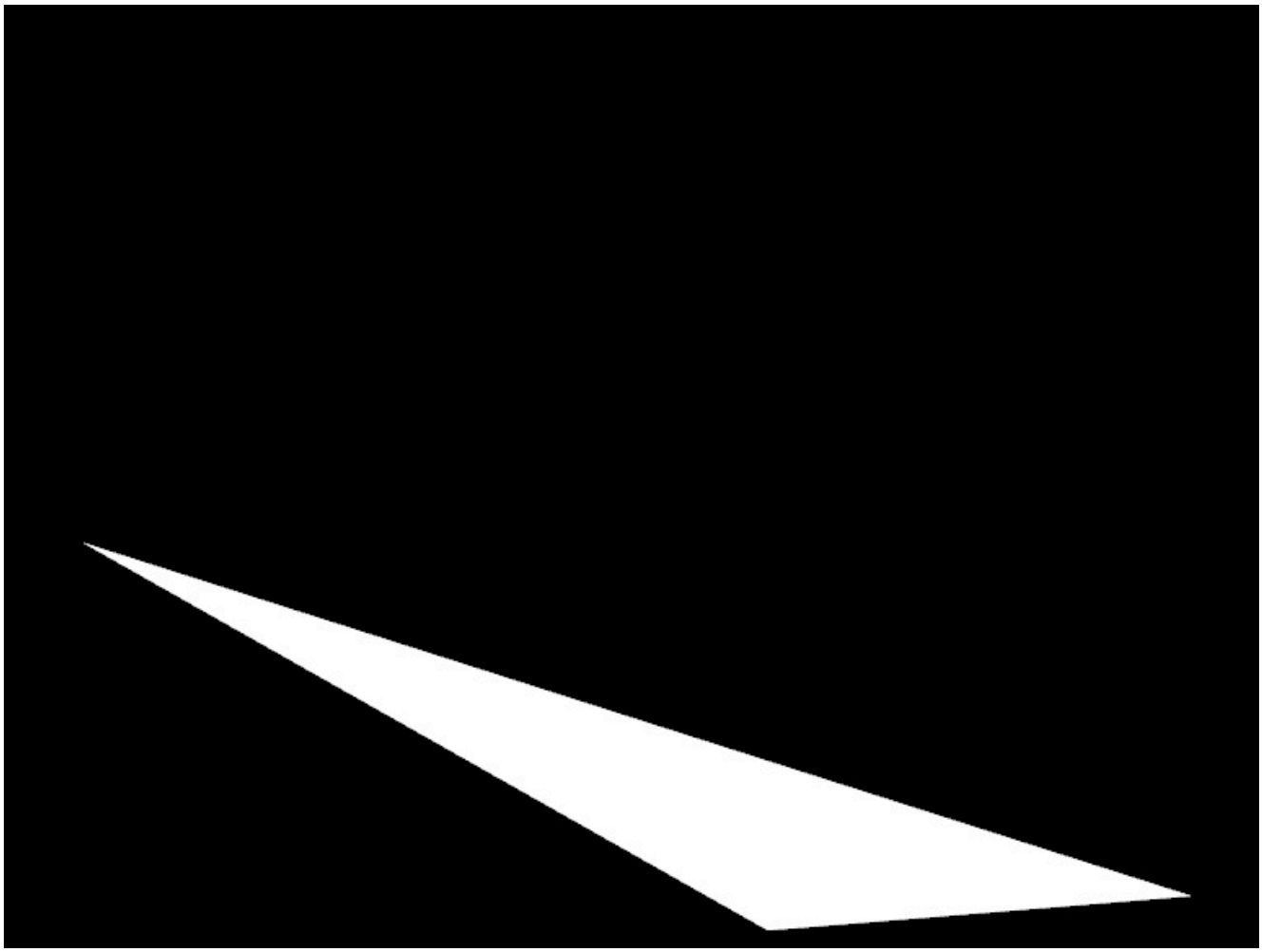
Classes that belong to the low-level drawing API:

- `Shader`
- `IndexBuffer`
- `VertexBuffer`
- `VertexFormat`
- `ColorBuffer`
- `DepthBuffer`
- `ArrayTexture`
- `CubeMap`
- `DrawStyle`
- `RenderTarget`

We discourage writing code that uses OpenGL directly; even though currently only an OpenGL 3.3 implementation exists for `Driver`, in the future we may add implementations that are not based on OpenGL.

Example

In the following example we show the minimum steps required for drawing a single triangle.



```
fun main() = application {
    program {
        val geometry = vertexBuffer(vertexFormat {
            position(3) // -- this attribute is named "position"
        }, 3)

        geometry.put {
            for (i in 0 until geometry.vertexCount) {
                write(Vector3(2.0 * Math.random() - 1.0, 2.0 * Math.random() - 1.0, 0.0))
            }
        }

        // -- code for the vertex shader
        val vs = """
        #version 330
        in vec3 a_position;  // -- driver adds a_ prefix (a for attribute)
        void main() {
            gl_Position = vec4(a_position, 1.0);
        }
        """

        // -- code for the fragment shader
        val fs = """
        #version 330
        out vec4 o_output;
        void main() {
```

```
            o_output = vec4(1.0);
        }
        """

        val shader = Shader.createFromCode(vsCode = vs, fsCode = fs, name = "custom-shader")

        extend {
            shader.begin()
            Driver.instance.drawVertexBuffer(shader, listOf(geometry), DrawPrimitive.TRIANGLES, 0, 3)
            shader.end()
        }
    }
}
```

Link to the full example

ColorBuffers sampled as integer numbers

Default `ColorBuffer`s created by OPENRNDR store values for each color channel as byte, however when used in shaders they are provided by `sampler2D` as floating point values in the range `0..1`. In rare cases it is desired to access these integer values directly, either as signed or unsigned integers, without conversion to floating point numbers. Special types of `isampler2D` and `usampler2D` come to the rescue, but color buffers have to be configured correctly for such a use.

Note: this mechanism is used internally in `orx-kinect` to process raw kinect data directly on GPU, as the depth readings are provided as integer numbers in the range of `0-2047` or `0-4096` depending on the kinect version.

Example use

```
fun main() = application {
    configure {
        width = 640
        height = 480
    }
    program {
        val magicNumber: Byte = 42
        // 1x1-pixel buffer holding one unsigned short / uint16 value
        val intColorBuffer = colorBuffer(1, 1, format = ColorFormat.R, type = ColorType.UINT16_INT)

        // the following line is crucial, non-nearest filtering will result in
        // sampling texels always equal to 0 in shaders
        intColorBuffer.filter(MinifyingFilter.NEAREST, MagnifyingFilter.NEAREST)

        // standard color buffer to match the output window size, with ColorType.UINT8
        // by default, which is still sampled as a floating point number in shaders
        val imageColorBuffer = colorBuffer(width, height)

        // fragment shader to transform integer data into something visible
        // if input is matching the magic number, we are outputting whiteness
        // usampler2D is used for unsigned integer values
        val intInputRenderer = Filter(Shader.createFromCode(fsCode = """
                    #version 330

                    uniform usampler2D tex0;
                    out     vec4 color;

                    const uint EXPECTED_VALUE = uint($magicNumber);

                    void main() {
                        uvec4 texel = texelFetch(tex0, ivec2(0), 0);
                        color = vec4(
                            vec3((texel.r == EXPECTED_VALUE) ? 1.0 : 0.0),
                            1.0
                        );
                    }
```

```
                """.trimIndent(), vsCode = Filter.filterVertexCode, name = "shader with usampler2D as input"))

        // in the same example we are also testing texture write / read
        val inData = ByteBuffer.allocateDirect(2) // 2 because we are storing shorts
        val outData = ByteBuffer.allocateDirect(2)
        inData.rewind()
        inData.put(magicNumber)
        inData.put(0)
        inData.rewind()
        intColorBuffer.write(inData)
        outData.rewind()
        intColorBuffer.read(outData)
        outData.rewind()
        println("outData should contain $magicNumber, is: ${outData.short}")
        extend {
            intInputRenderer.apply(intColorBuffer, imageColorBuffer)
            drawer.image(imageColorBuffer, 0.0, 0.0, width.toDouble(), height.toDouble())
        }
    }
}
```

Compute shaders

Since version 0.3.36 OPENRNDR comes with compute shader functionality for select platforms. Compute shader support only works on systems that support OpenGL 4.3 or higher. MacOS support is provided via Angle.

Example use

This example is composed of two code blocks. The first block is a compute shader program written in GLSL which produces an `outputImg` by mixing three input colors:

- A pixel sampled from `inputImg`.
- A `fillColor` sent as a uniform from Kotlin.
- A color generated by calculating the cosine of the coordinates of the pixel currently being processed.

A typical location for such a compute shader could be `data/compute-shaders/fill.cs`.

```
#version 430
layout(local_size_x = 1, local_size_y = 1) in;

uniform vec4 fillColor;
uniform float seconds;
layout(rgba8) uniform readonly image2D inputImg;
layout(rgba8) uniform writeonly image2D outputImg;

void main() {
    ivec2 coords = ivec2(gl_GlobalInvocationID.xy);
    float v = cos(coords.x * 0.01 + coords.y * 0.01 + seconds) * 0.5 + 0.5;
    vec4 wave = vec4(v, 0.0, 0.0, 1.0);
    vec4 inputImagePixel = imageLoad(inputImg, coords);

    imageStore(outputImg, coords, wave + inputImagePixel + fillColor);
}
```

The second code block is an OPENRNDR program making use of the compute shader:

1. It creates a compute shader program from a file.
2. Initializes the input buffer using a image loaded from disk.
3. Inside the `extend` block (called multiple times per second) some uniforms are updated: a `fillColor`, the time in `seconds`, the `inputImg` and the `outputImg`. In this example only the time changes on every frame.
4. The compute shader is executed, writing the result to the `outputBuffer`.
5. Finally, the result is displayed by calling `drawer.image()`.

```
fun main() = application {
    program {
        val cs = ComputeShader.fromCode(File("data/compute-shaders/fill.cs").readText(), "cs1")

        val tempBuffer = loadImage("data/images/cheeta.jpg")
        val inputBuffer = colorBuffer(width, height, type = ColorType.UINT8)
        tempBuffer.copyTo(inputBuffer)
        val outputBuffer = colorBuffer(width, height, type = ColorType.UINT8)

        extend {
            cs.uniform("fillColor", ColorRGBa.PINK.shade(0.1))
            cs.uniform("seconds", seconds)
            cs.image("inputImg", 0, inputBuffer.imageBinding(0, ImageAccess.READ))
            cs.image("outputImg", 1, outputBuffer.imageBinding(0, ImageAccess.WRITE))
            cs.execute(outputBuffer.width, outputBuffer.height, 1)
            drawer.image(outputBuffer)
        }
```

```
    }
}
```

Link to the full example

If we prefer to work with floating point buffers, we could set the `type` of both `ColorBuffer` instances to `ColorType.FLOAT32` instead of `ColorType.UINT8`, and use `layout(rgba32f)` instead of `layout(rgba8)` in the GLSL code.

ComputeStyle

To simplify working with compute shaders, OPENRNDR provides the `computeStyle` builder, which should feel familiar if you've used `shadeStyle`. The `computeStyle` builder takes care of declaring the GLSL version, the layout and the uniforms, which helps avoid typos and other errors in the code.

The same program above written using `shadeStyle` looks like this:

```
fun main() = application {
    program {
        val cs = computeStyle {
            computeTransform = """
                    ivec2 coords = ivec2(gl_GlobalInvocationID.xy);
                    float v = cos(coords.x * 0.01 + coords.y * 0.01 + p_seconds) * 0.5 + 0.5;
                    vec4 wave = vec4(v, 0.0, 0.0, 1.0);
                    vec4 inputImagePixel = imageLoad(p_inputImg, coords);

                    imageStore(p_outputImg, coords, wave + inputImagePixel + p_fillColor);
                """.trimIndent()
        }

        val tempBuffer = loadImage("data/images/cheeta.jpg")
        val inputBuffer = colorBuffer(width, height, type = ColorType.UINT8)
        tempBuffer.copyTo(inputBuffer)
        val outputBuffer = colorBuffer(width, height, type = ColorType.UINT8)

        extend {
            cs.parameter("fillColor", ColorRGBa.PINK.shade(0.1))
            cs.parameter("seconds", seconds)
            cs.image("inputImg", inputBuffer.imageBinding(0, BufferAccess.READ))
            cs.image("outputImg", outputBuffer.imageBinding(0, ImageAccess.WRITE))
            cs.execute(outputBuffer.width, outputBuffer.height, 1)
            drawer.image(outputBuffer)
        }
    }
}
```

Link to the full example

Notice how we used `cs.parameter()` instead of `cs.uniform()`, the `cs.image()` method lost one argument, and the GLSL code got simplified by writing only the content of its `main()` function.

The names of the parameters we passed to the compute shader received a `p_` prefix, for instance `seconds` became `p_seconds` in the GLSL code. This makes parameters easy to distinguish from other GLSL variables.

A growing list of uses cases

OPENRNDR can be used to create a wide variety of programs involving 2D and 3D graphics, interactive experiences, live coding sessions and more.

Here we list some use cases for inspiration.

TABLE OF CONTENTS

Live coding with orx-olive

By using Kotlin's ability to run script files we can build a live coding environment. The `orx-olive` library simplifies the work to be done to set up a live coding environment. Code and additional documentation for the library can be found in the Github repository.

Prerequisites

Assuming you are working on an `openrndr-template` based project, `orx-olive` is enabled by default.

Basic example

```
fun main() = application {
    configure {
        width = 768
        height = 576
    }
    oliveProgram {
        extend {
            // drawer.fill = ColorRGBa.PINK
            drawer.rectangle(0.0, 0.0, 100.0, 200.0)
        }
    }
}
```

Try editing the source code to change the fill color of the rectangle (or any other property) and save your changes. The new color should appear instantly without having to re-run the program.

Interaction with extensions

The Olive extension works well together with other extensions. In the following example we see the use of `Camera2D` in combination with `Olive`.

```
fun main() = application {
    oliveProgram {
        extend(Camera2D())
        extend {
            drawer.rectangle(0.0, 0.0, 100.0, 200.0)
        }
    }
}
```

Adding persistent state

Sometimes you want to keep parts of your application persistent, that means its state will survive a script reload.

This is how we can make the `Camera2D` from the previous example persistent:

```
fun main() = application {
    oliveProgram {
        val camera by Once {
            persistent {
                Camera2D()
            }
        }
        extend(camera)
        extend {
            drawer.rectangle(0.0, 0.0, 100.0, 200.0)
        }
    }
}
```

The same approach can be used to maintain a persistent connection to a webcam:

```
fun main() = application {
    oliveProgram {
        val webcam by Once {
            persistent {
                VideoPlayerFFMPEG.fromDevice()
            }
        }
        webcam.play()
        extend {
            webcam.colorBuffer?.let {
                drawer.image(it, 0.0, 0.0, 128.0, 96.0)
            }
        }
    }
}
```

GUI workflow

`orx-gui` is built with the `orx-olive` environment in mind. Its use is similar to the workflows described prior, however, in live mode the ui comes with some extra features to make live-coding more fun. The best part is that `orx-gui` can retain parameter settings between script changes by default, so nothing jumps around.

Notice the use of `Reloadable()` in the `settings` object.

```
@file:Suppress("UNUSED_LAMBDA_EXPRESSION")
import org.openrndr.Program
import org.openrndr.color.ColorRGBa
import org.openrndr.draw.*
import org.openrndr.extra.compositor.compose
import org.openrndr.extra.compositor.draw
import org.openrndr.extra.compositor.layer
import org.openrndr.extra.compositor.post
import org.openrndr.extra.gui.GUI
import org.openrndr.extra.parameters.*

fun main() = application {
```

```
    configure {
        width = 800
        height = 800
    }
    oliveProgram {
        val gui = GUI()
        val settings = @Description("User settings") object : Reloadable() {
            @DoubleParameter("x", 0.0, 1000.0)
            var x = 0.0
        }
        val composite = compose {
            draw {
                drawer.clear(ColorRGBa.PINK)
                drawer.circle(settings.x, height / 2.0, 100.0)
            }
        }
        extend(gui) {
            add(settings)
        }
        extend {
            composite.draw(drawer)
        }
    }
}
```

Generating pen plotter art

A pen plotter is a computer output device that can draw with a pen (or a brush, marker, etc) on a paper. Some such devices move the pen in the X and Y axes, while others can move the paper instead.

Pen plotters were introduced in the 1950s and 60s and have become popular among artists since the mid 2000s.

Unlike other output devices like ink-jet and laser printers which accept pixels, pen plotters must be fed with vector data: lines and curves.

OPENRNDR provides a rich toolset to generate and manipulate vector data.

SVG vs g-code

There are two main file formats used to send designs to pen plotters:

- SVG (Scalable Vector Graphics), used with newer devices like the AxiDraw.
- g-code, often supported by older plotters, CNC devices and laser cutters.

OPENRNDR can easily load, manipulate, generate and save SVG files. There is a non-yet-merged contribution to add g-code support.

The following sections focus on SVG.

Hello world

This is one of the simplest programs we can write to produce an SVG file containing just a circle.

```
fun main() = application {
    program {
        val design = drawComposition {
            circle(drawer.bounds.center, 200.0)
        }
        design.saveToFile(File("data/design.svg"))
    }
}
```

The API in the composition drawer is almost identical to the one of the standard drawer: we can use methods like `segment`, `contour`, `shape`, `circle`, `rectangle`, etc.

Note: if you are using OPENRNDR / ORX version 0.4.5 you need to enable `orx-composition` and `orx-svg` in the `build.gradle.kts` file.

Interaction

Lets take our simple program a step further and make it interactive. We will generate a new design every time the mouse is clicked and save the design when we press the `s` key on the keyboard.

```
fun main() = application {
    program {
        // Create an empty composition
        val design = drawComposition {}
```

```
        // A function to draw concentric circles into the composition.
        fun generateDesign() {
            design.clear()
            val separation = 4.0
            val steps = (drawer.bounds.center.distanceTo(mouse.position) / separation).toInt()
            design.draw {
                repeat(steps) {
                    circle(drawer.bounds.center, (it + 1) * separation)
                }
            }
        }

        // Draw the composition onto the window
        extend {
            drawer.clear(ColorRGBa.WHITE)
            drawer.fill = null
            drawer.composition(design)
        }

        // Generate a new design every time we click the mouse
        mouse.buttonDown.listen {
            generateDesign()
        }

        // Show a save dialog when pressing the `s` key, then save the design
        // with the chosen file name into the selected folder.
        keyboard.keyDown.listen {
            if (it.name == "s") {
                saveFileDialog(supportedExtensions = listOf("SVG" to listOf("svg"))) { file ->
                    design.saveToFile(file)
                }
            }
        }
    }
}
```

With the last program we could create minimal designs by clicking the mouse button on different locations on the program window and then save the design when pressing the `s` key.

If we open the resulting design in Inkscape we will notice that the document size is 640 pixels wide and 480 pixels height, matching the default OPENRNDR window size. In Inkscape we could choose a different document size (A4 for instance) then re-center and scale the design to match our preferences.

Lets find out how to specify the SVG document dimensions, and how to make what we see in the window match what gets exported as SVG. (Coming soon...)

Find more tips on using OPENRNDR with pen plotters in the forum.

Best practices

Coming soon.