Mathil Documentation

Aaron Manning

Contents

1 Introduction

This is the documentation for Mathil, a library for drawing images and creating animations programmatically in Rust. If you are reading this documentation first, please visit the GitHub page for this project at: https://github.com/aaron-jack-manning/mathil, which will be frequently referenced here.

2 Screen Type

Mathil draws rasters directly, to allow drawing images directly from their mathematical representations. The type used in Mathil for representing an image is called a `Screen`. To create a screen use the `Screen::new` function.

```
let blank_screen =
    Screen::new(
        2000, 2000,
        Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
        Colour::from_rgb(20, 20, 20)
    );
```

This functions requires that the resolution be specified, along with the default colour and the bounding box of the screen. The bounding box is just the coordinate system used by Mathil when rendering objects on the screen, and it is constructed with two points, created with `Point::new`, which represent the bottom left and top right bounds. By making the coordinate system independent of the resolution, the resolution can easily be changed and the rendered objects will be scaled accordingly. This does however mean that one must be careful to verify that the aspect ratios of the bounding box and screen do not differ too significantly, or the rendered images will be stretched. A utility function is provided to calculate the aspect ratios of the resolution and the bounding box respectively so that they can be checked so they are not too different.

```
let blank_screen =
    Screen::new(
        2000, 2000,
        Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
        Colour::from_rgb(240, 240, 240)
    )

println!("{:?}", aspect_ratios(&blank_screen));
```

```
> AspectRatios { resolution: 1.0, bounds: 1.0 }
```

In this case, the aspect ratios match so we can be sure that no stretching occurs.

3 Colours

Colours in Mathil are internally represented by RGB values, and can be created from the RGB values using `Colour::from_rgb` or from a hex code string using `Colour::from_hex`.

All colours in the CSS standard are also provided as static variables in the `css_colours` module, for convenience.

Natural colour mixing is an in development feature.

4 Renderables

Rendering in Mathil is handled by implementing the `Renderable` trait. A variety of mathematical objects provided in the `maths_object` module are provided which implement `Renderable`.

The `Renderable` trait also requires a definition of a type which represents the rendering settings. This means, for every possible renderable type, there is a corresponding type which defines how that object is rendered. This abstraction keeps the mathematical object distinct from the way it appears. Rendering settings include things like thicknesses and colours.

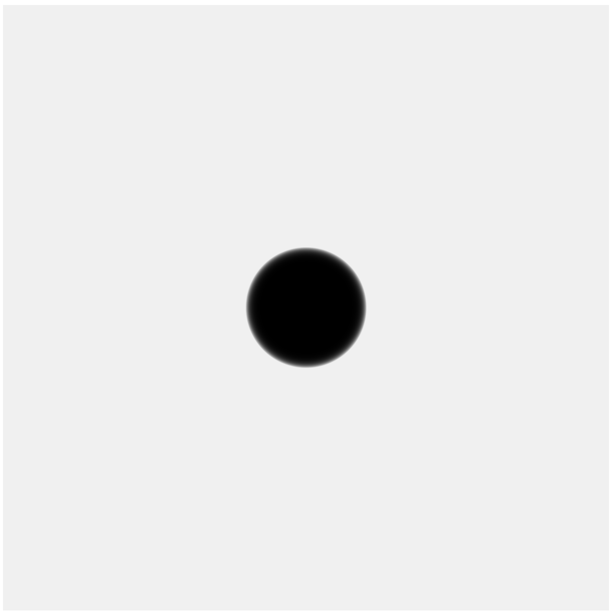
Anything that implements this renderable type can be passed as input, along with an instance of its corresponding render settings type, to the `.render` method which takes ownership of and then returns a `Screen`.

We will cover each of the types that implement `Renderable` and their settings in the following sections.

4.1 Point

One of the key primitives in Mathil is a point. A point is internally just a struct with two fields, both of type `f32`. A point can be created using the `Point::new` function. Points are crucial in the representations of most other renderable types, but they can also be rendered directly.

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Point::new(0.0, 0.0),
    PointRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.2),
        RenderingType::RoundAntiAliased(10.0)
    )
)
```



We've already covered how the point is created, but let's look more closely at the rendering settings.

Here we have created an instance of the `PointRenderSettings` type, which requires that we specify a colour, a thickness, and a rendering type. The colour is created using any of the three methods discussed in the *colours* section.

The thickness can be specified as a number of pixels using the `Thickness::Absolute(u16)` case or based on the coordinate system specified on the screen using the `Thickness::Relative(f32)` case. Note that the only reason this option is given here is because of the potiential problems that come with relatively determining a point's radius, and as you'll soon see, a line's thickness. Rendering a point guarantees that it won't be stretched even if the aspect ratios don't match. It will instead, take an average of the number of pixels as a proportion of the horizontal and vertical resolution. This is because of the way curves are rendered, so that the thickness of a curve is consistent.

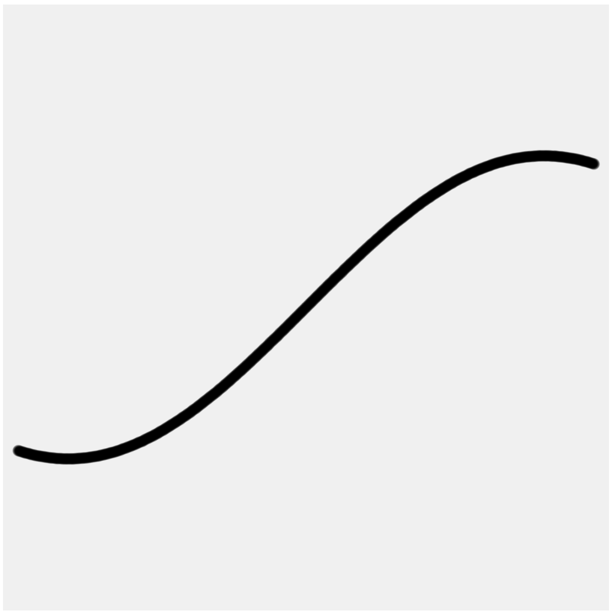
The rendering type specifies how the point will be rendered. `RenderingType::Square` renders the point as a square, `RenderingType::RoundAliased` renders the point as a circle and `RenderingType::RoundAntiAliased(f32)` renders the point as a circle and provides an anti-aliasing effect determined by the provided float. Larger inputs give something closer to `RoundAliased` while values closer to 0 give a softer effect. I would generally recommend starting with 10 for points and 2 for curves (which will be covered in the next section).

4.2 Function

A `Function` is defined by an interval on the real numbers, and a function from that interval to a `Point`.

To create a new function, we can do so by supplying a rule as a `Box<dyn Fn(f32) -> Point>` and domain as a pair of `f32`s with the `Function::new` function.

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new(
        Box::new(move |t| {
            Point::new(
                0.5 * t, 0.5 * t.sin()
            )
        }),
        (-1.9, 1.9)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



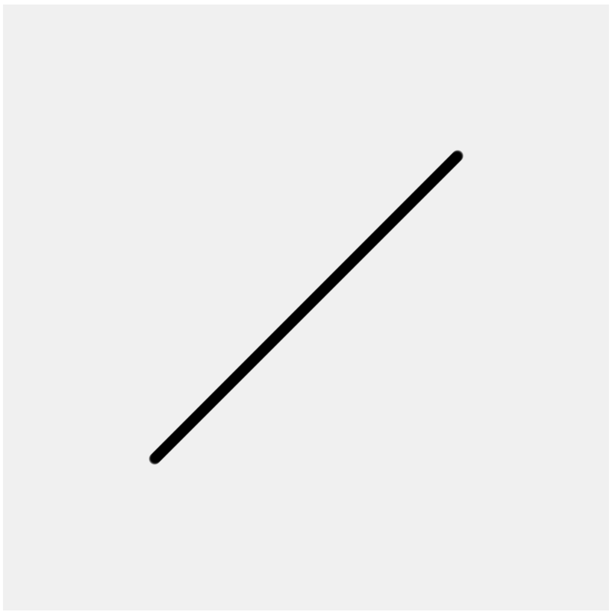
The only new rendering setting we've introduced here is the number of samples, which is 300 in this case. This option exists because a function is rendered by taking a series of samples on the curve. In general, making this number too low will cause the curve to look like a series of disjoint dots, but making the number too high will slow down rendering time. It's best to just tweak this and see the result to get it as low as possible whilst giving a nice clean line.

The `Function` type also has implementations for many helper functions to make generating standard curves easier.

4.2.1 Line Segment

A line segment is constructed by the endpoints. For a line from these inputs, the domain should be `(0.0, 1.0)` since internally this is just dividing the interval. The option of providing a different domain is primarily to make animating a line being drawn easier. The same can be seen with some of the later functions.

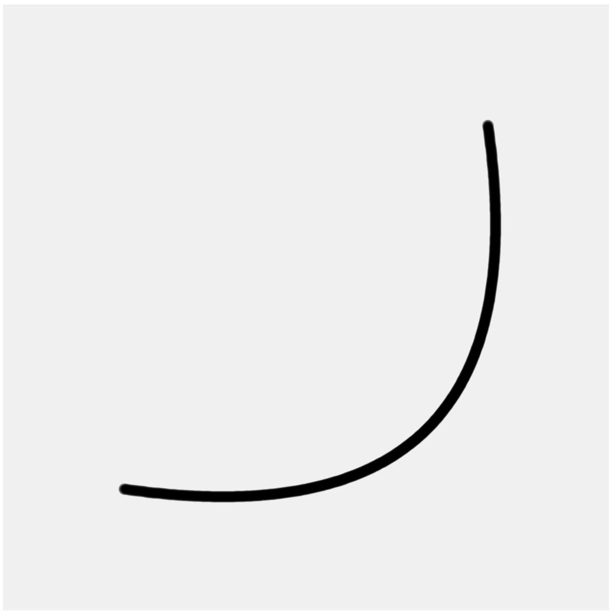
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new_line_segment(
        Point::new(-0.5, -0.5),
        Point::new(0.5, 0.5),
        (0.0, 1.0)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.2.2 Bezier Curve

A bezier curve is a generalising of the line segment, and can be created by a series of points, and a domain which should similarly be `(0.0, 1.0)` under most circumstances.

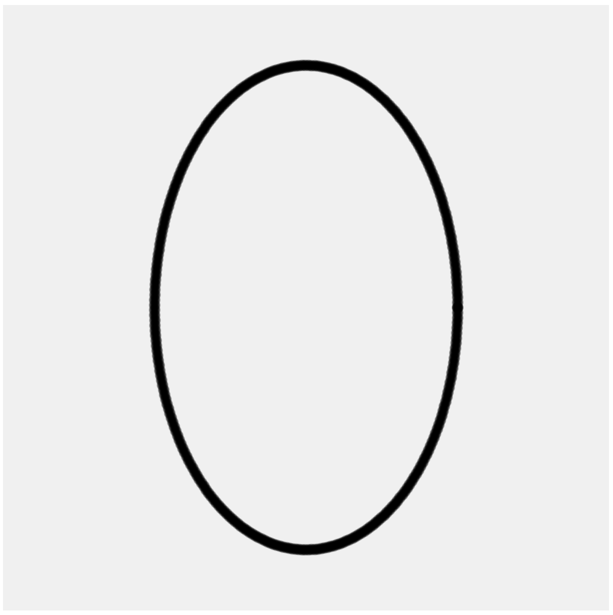
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new_bezier_curve(
        vec![Point::new(-0.6, -0.6), Point::new(0.8, -0.8), Point::new(0.6, 0.6)
            ],
        (0.0, 1.0)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.2.3 Ellipse

An ellipse can be created from the radius across the x and y dimensions, and the center.

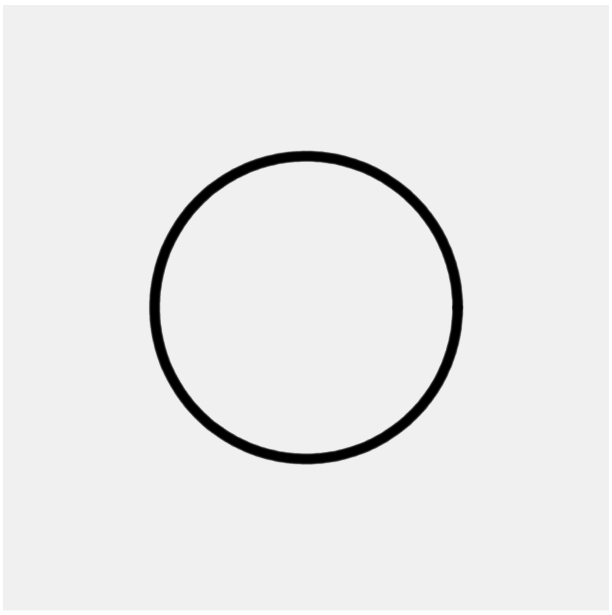
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new_ellipse(
        0.5,
        0.8,
        Point::new(0.0, 0.0),
        (0.0, TAU)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.2.4 Circle

And a circle is just a specific case of an ellipse, where the radii are equal.

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new_circle(
        0.5,
        Point::new(0.0, 0.0),
        (0.0, TAU)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.3 Polygon

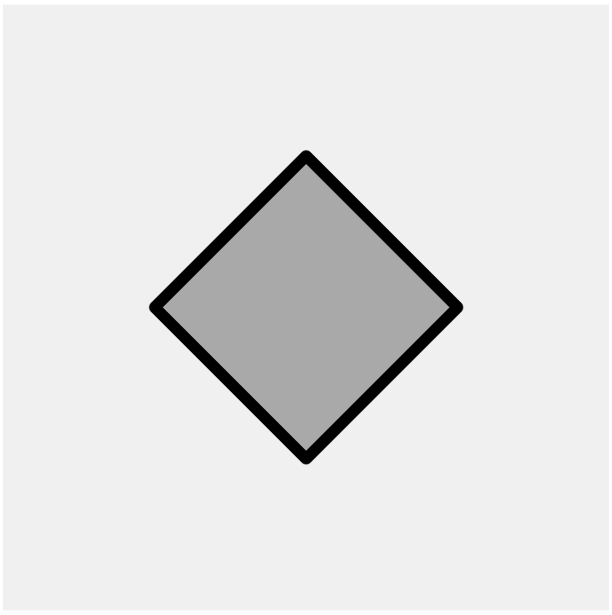
Polygon's are slightly more complex objects, as when rendered they can be rendered by filling their inside, by drawing line segments for each of their sides, or both. The drawing of the external line segments is internally the same as drawing a sequence of line segments as functions. The colour fill for a polygon is the second animation primitive within Mathil. The way these two types are handled is through two independent render settings types, grouped in one overall type.

The internal representation of `PolygonRenderSettings` is as follows:

```
pub struct PolygonRenderSettings {
    sides : Option<PolygonSidesRenderSettings>,
    fill : Option<PolygonFillRenderSettings>,
}
```

The two fields contain options of instances of the render settings for the sides and fill respectively. These are options, as passing None as input allows just the sides or just the fill to be rendered.

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Polygon::new(
        vec![Point::new(0.0, 0.5), Point::new(0.5, 0.0), Point::new(0.0, -0.5),
            Point::new(-0.5, 0.0)]
    ),
    PolygonRenderSettings::new(
        Some(PolygonSidesRenderSettings::new(
            css_colours::BLACK,
            Thickness::Relative(0.02),
            300,
            RenderingType::RoundAntiAliased(2.0)
        )),
        Some(PolygonFillRenderSettings::new(
            css_colours::DARK_GRAY
        ))
    )
)
```

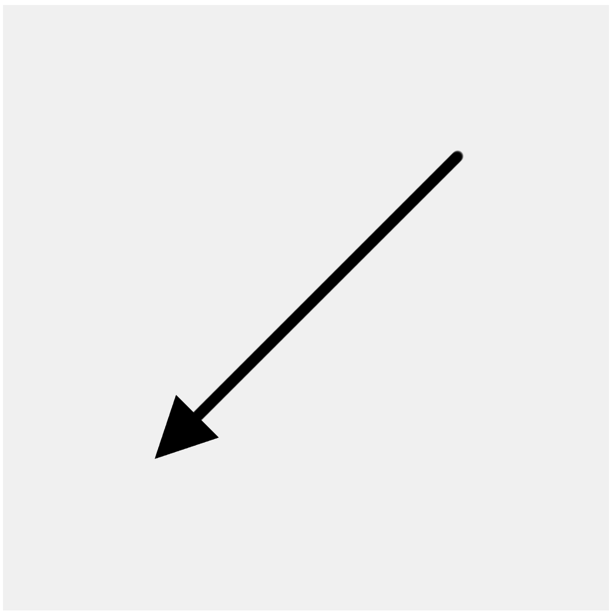


4.4 Vector

Internally, a vector is just a line segment for the body and a polygon for the arrow head.

A vector is created from the head and tail, along with the dimensions of the arrow head. The head should be specified to be where the tip of the arrow is.

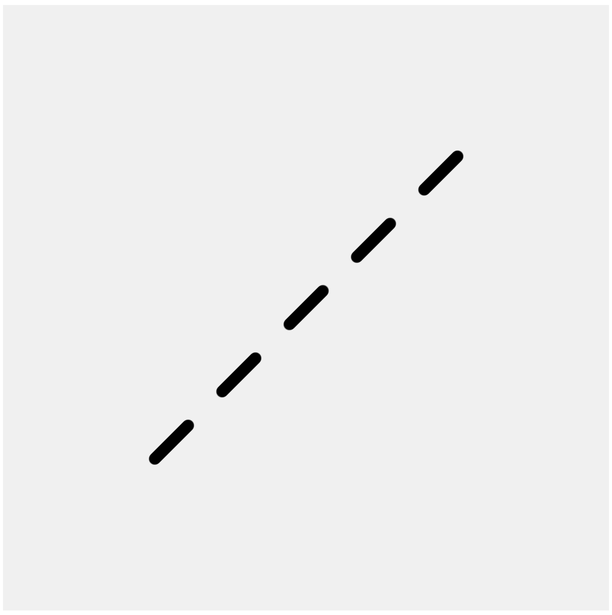
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Vector::new(
        Point::new(-0.5, -0.5),
        Point::new(0.5, 0.5),
        0.2,
        0.2
    ),
    VectorRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.5 Dashed Line

A dashed line is a series of line segments. When creating a dashed line, the start and endpoint are determined, along with the number of dashes. Dashes will always appear on the ends.

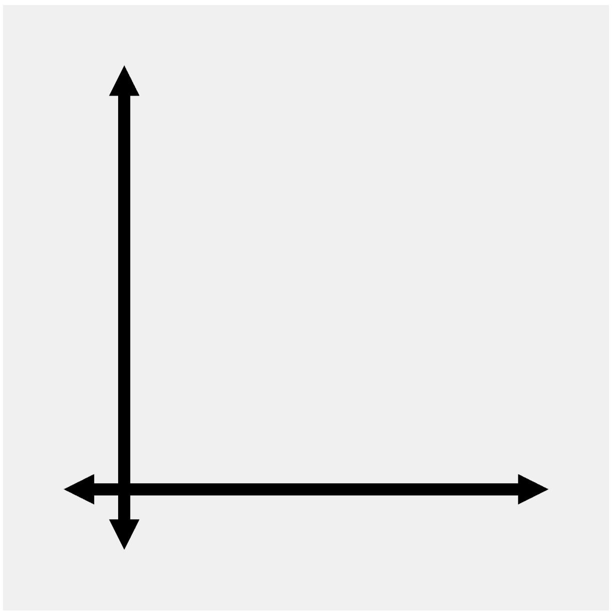
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    DashedLine::new(
        Point::new(-0.5, -0.5),
        Point::new(0.5, 0.5),
        5
    ),
    DashedLineRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
```



4.6 Cartesian Plane

A cartesian plane is just an abstraction on four vectors, since it is a frequently drawn enough object. When creating a cartesian plane, the location of the plane is determined by the bottom right and top left bounds, along with the location of the origin. It will always be drawn so that the axis are parallel to the bounds of the frame.

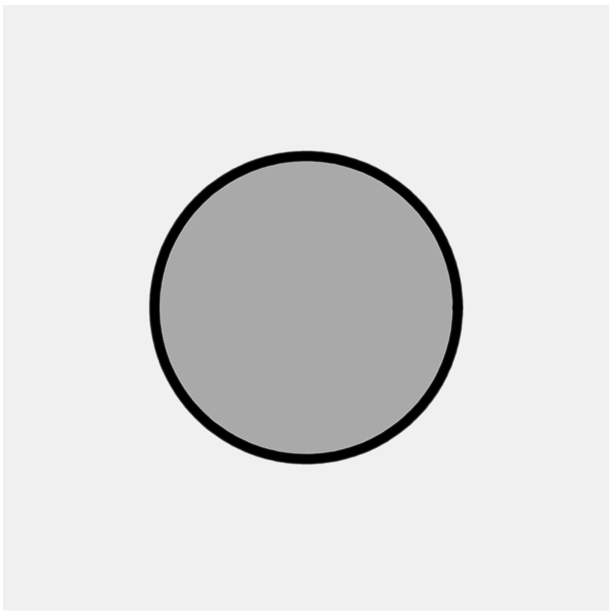
```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    CartesianPlane::new(
        Point::new(-0.8, -0.8),
        Point::new(0.8, 0.8),
        Point::new(-0.6, -0.6),
        0.1,
        0.1
    ),
    CartesianPlaneRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300
    )
)
```



5 Colour Fills

Mathil also provides a function for filling solid colours. For example, consider the example used for creating a circle, now with its interior filled dark gray. To fill this region, we need to specify the desired colour, and a starting point for the fill.

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Function::new_circle(
        0.5,
        Point::new(0.0, 0.0),
        (0.0, TAU)
    ),
    FunctionRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.02),
        300,
        RenderingType::RoundAntiAliased(2.0)
    )
)
.fill(
    Point::new(0.0, 0.0),
    css_colours::DARK_GRAY
)
```



Since only solid colours are filled, if using anti-aliased lines it's recommended that a thinner line without anti aliasing is drawn first, just for the sake of the colour fill, before drawing the final line on top.

6 Output

Mathil supports two types of output file formats, bitmaps and pngs. The functions that write to these files are called `write_to_bitmap` and `write_to_png` respectively, both of which take as input the folder where the created file should be stored and the filename, the path is verified and the file extension added by these functions. While the above code examples have not shown the write to output file step, the example of drawing a point including that line may look like this:

```
Screen::new(
    2000, 2000,
    Point::new(-1.0, -1.0), Point::new(1.0, 1.0),
    Colour::from_rgb(240, 240, 240)
)
.render(
    Point::new(0.0, 0.0),
    PointRenderSettings::new(
        css_colours::BLACK,
        Thickness::Relative(0.2),
        RenderingType::RoundAntiAliased(10.0)
    )
)
.write_to_png(”C:\\images”, ”point”);
```

7 Animation Tools

Mathil also has tools to create a series of images as frames of animation more easily. This is through the `animation` module.

The primary function used to create an animation is called animate. The signature of this function is as follows:

```
fn animate(fn(f32, u32) -> Screen, f32, u8, &'static str)
```

The idea of this function is that you can provide as input a function pointer to a function which generates each frame of video based on the timestamp, and therefore the timestamp and length of the animation can be supplied as other inputs to make the video framerate independent. Notice that the function which generates each frame also takes in a ‘u32‘, when trying create a framerate independent animation, this should simply be ignored. However, this variable will be passed in as the current zero indexed frame number in the animation, which is sometimes useful for handling special cases in animations where frame rate independence is not cruical, or for cases where rendering at a higher frame rate giving faster motion is desired. The animate function also takes as input the length of the animation in seconds, the framerate and the output folder to write the frames of animation to.

The animation module also includes a function called smooth which is used to smooth out animations by mapping the interval from 0 to 1 onto itself with an increasing function such that the rate of increase is lowest at the endpoints and greatest in the middle. There are two functions to use in this smoothing from the Smoother enum, parameterised on how harsh the smoothing effect is, where values closer to zero have a less significant effect. An example of this function being used can be seen in the *curve intersection* example, which applies it to the domain of the curves being drawn.

The final argument to the animate function is just a path to a folder where the frames of animation will be written.

Here is the code behind the curve intersection example from the GitHub page, incorporating many of the ideas explained above.

```
fn main() {
    animate(curve_intersection_generator, 4.0, 60, "")
}

pub fn curve_intersection_generator(time_stamp : f32, frame : u32) -> Screen {

    let line_thickness =
        Thickness::Relative(0.044);
    let cartesian_plane_thickness =
        Thickness::Relative(0.022);

    let mut screen =
        Screen::new(
            3840, 2160,
            Point::new(-7.11, -4.0), Point::new(7.22, 4.0),
            Colour::from_hex("#2c3e50")
        )
        .render(
            CartesianPlane::new(
                Point::new(-6.11, -3.0),
                Point::new(6.11, 3.0),
                Point::new(0.0, 0.0),
                0.3,
                0.3
            ),
            CartesianPlaneRenderSettings::new(
                Colour::from_hex("#ecf0f1"),
                cartesian_plane_thickness,
                500
            )
        )
        .render(
            Function::new_bezier_curve(
                vec![
                    Point::new(-5.0, 2.5),
                    Point::new(-4.0, -9.0),
                    Point::new(2.0, 9.0),
                    Point::new(6.0, -2.0)
                ],
                if time_stamp < 1.0 {
                    (0.0, smooth(time_stamp, Smoother::Arctan(2.0)))
                }
                else {
                    (0.0, 1.0)
                }
            ),
            FunctionRenderSettings::new(
                Colour::from_hex("#2ecc71"),
                line_thickness,
                if time_stamp < 1.0 {
                    (1000.0 * smooth(time_stamp, Smoother::Arctan(2.0))) as u16
                }
                else {
                    1000
                },
                RenderingType::RoundAntiAliased(2.0)
            )
        );

    if time_stamp > 2.0 {
        screen =
            screen.render(
                Function::new_line_segment(
                    Point::new(-6.11, 1.0),
                    Point::new(6.11, 1.0),
                    if time_stamp < 3.0 {
                        (0.0, smooth(time_stamp - 2.0, Smoother::Arctan(2.0)))
                    }
                    else {
                        (0.0, 1.0)
                    }
                ),
                FunctionRenderSettings::new(
                    Colour::from_hex("#e67e22"),
                    line_thickness,
                    1000,
                    RenderingType::RoundAntiAliased(2.0)
                )
            );
    }

    screen
}
```

This will write a series of files entitled `frame_00001.png`, `frame_00002.png`, `frame_00003.png`... to the specified output folder.

Mathil doesn't currently have a tool for converting these frames into a video file, so for now *ffmpeg* is the recommended option. The command to combine a series of frames named according to how Mathil outputs them into an `.mp4` is as follows:

```
ffmpeg -framerate 60 -i frame_%05d.png output.mp4
```

Here the framerate is set at 60fps, but this can obviously be changed to match whichever settings were given in the source code.