Programming Languages and Compiler Construction
Department of Computer Science
Christian-Albrechts-University of Kiel

Bachelor Thesis

Implementation of a Library for Declarative, Resolution-Independent 2D Graphics in Haskell

Author: Finn Teegen
Date: 30th September 2013
Advised by: Prof. Dr. Michael Hanus and Dipl.-Inf. Fabian Skrlac

Declaration of Authorship

I hereby certify that this bachelor thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Place, date	Signature

Contents

Figures

Listings

Charts

1. Introduction

Most of today's graphics frameworks are designed according to what computers can draw efficiently instead of to how graphics can be best expressed and composed. As a result, such frameworks are often limited in many respects, e.g., they offer just a limited set of shape primitives, a limited set of textures or allow only affine transformations (that map parallel lines to parallel lines). This hinders the ease of programming 2D graphics considerably, since the programmer is subject to the aforementioned restrictions.

For example, Figure 1.1 shows two graphics that are hard or even not at all expressable in traditional graphics frameworks. Each of these graphics is affected by one of the above-mentioned problems: The spiral is not a Bézier curve and the whirl transformation is no affine transformation.

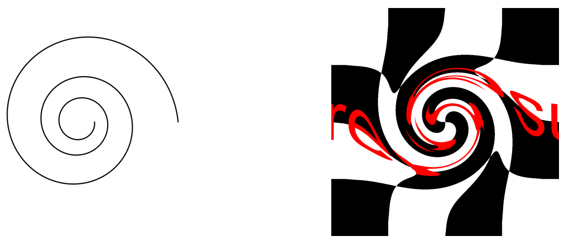

(a) Spiral. **(b)** Whirl transformation.

Figure 1.1.: Example graphics.

Paul Klint and Atze van der Ploeg, two Dutch, have recognized this deficit and elaborated a new declarative approach to resolution-independent 2D graphics which has been published as a paper[1]. They also provide a reference implementation programmed in Scala, a multi-paradigm programming language.

The goal of this thesis is to implement a library in the programming language Haskell that follows this approach and just like the original reference implementation meets the following design objectives:

- *Simplicity*: Our library is not to overwhelm a programmer with tons of concepts and should therefore be usable without substantial previous knowledge.
- *Expressivity*: A programmer should be able to express his ideas in an intuitive manner and should not need to be aware of any low-level concepts.
- *Composability*: Graphics should be composable and transformable in general ways.
- *Resolution-independence*: Graphics can be expressed independent of the actual resolution, meaning that they can be rendered at any level of detail.

The further chapters are organized as follows. At first, Chapter 2 presents the preliminaries of this thesis. This includes the introduction of some advanced features of Haskell that we use in our implementation and that go beyond the existing knowledge of previous lectures. With Cairo and SDL, we further introduce two libraries on which our implementation relies. In Chapter 3 we then illustrate the design choices for our library. Besides, some examples of usage are given. Chapter 4 deals with the main part of this thesis: the implementation itself. This chapter first explains the basic concept and then focuses on some specific implementation parts. Afterwards, we evaluate our implementation in Chapter 5 by comparing it to the original one considering performance and image quality. Here, we also explain possible reasons for occurring differences. At the very end, we discuss the results and give an outlook on some possibilities how to remedy remaining deficiencies.

2. Preliminaries

This chapter gives a brief introduction to fundamentals that are necessary to comprehend the following chapters and our implementation.

2.1. Haskell

Supplementary to the features already known from previous lectures, we introduce some advanced features of Haskell we make use of in our implementation.

2.1.1. Strictness

In General, Haskell uses a strategy called *laziness* to evaluate expressions whereat the evaluation of an expression is delayed until its value is needed (non-strictness). Also, repeated evaluations are avoided (sharing)[4].

Laziness can be useful to increase performance due to the avoidance of needless calculations. But more often than not it reduces performance by adding a constant overhead to everything, meaning the unevaluated expression has to be stored, in case it is evaluated later. Storing expressions is costly and unnecessary if the expression is going to be evaluated anyway.

Compilers like the Glasgow Haskell Compiler (GHC) try to reduce the cost of laziness by attempting to determine which function arguments are always evaluated by a function, and hence let the caller evaluate them.

To help the compiler, Haskell provides several ways to enforce strict evaluation of expressions[5]. For example, the GHC offers a language extension called *BangPatterns*. To give an example, the arguments *b* and *c* of the following function *f* will always be evaluated strictly (signalized with the exclamation mark):

```
f a !b !c = ...
```

Also, there is a variant of the infix application operator ($) that evaluates its argument strictly: ($!).

Last but not least, strictness annotations on constructor fields can be used. For example, both components of the following data type *T* will be fully evaluated to *Int*s when the constructor is called:

```
data T = T !Int !Int
```

In this thesis, there is no source code in which we use any of the described ways to enforce strict evaluation, but the implementation itself makes use of them.

2.1.2. Foreign Function Interface

Via Haskell's Foreign Function Interface (FFI) it is possible call functions from other languages (basically we use C in our thesis) and for C to call Haskell functions. The FFI also allows us to call functions from the runtime system (RTS). We use this functionality to be able to use some methods Cairo (see next section) normally provides but were not provided by the Haskell bindings for Cairo.

2.2. Cairo

Cairo[1] is a 2D graphics library that provides a vector graphics-based, device-independent API. It is designed to use hardware acceleration when available.

Although, Cairo offers a lot of functionality, we just use it to draw lines, Bézier curves and text paths, and to fill or stroke them. Support for Haskell is provided via bindings[2].

2.3. Simple DirectMedia Layer

Simple DirectMedia Layer[3] (SDL) is a cross-platform multimedia library that acts as a wrapper around operating-system-specific functions, providing a simple interface to graphics, sound and input devices.

SDL is divided into several subsystems, namely video, audio, CD-ROM, joystick and timer subsystem. Besides these, there exist a few separate official libraries to provide extended functionality, e.g., SDL_image for adding support for multiple image formats or SDL_net for network support. In our implementation we only use the video subsystem, the timer subsystem and SDL_image. To use SDL with Haskell, bindings[4] are provided.

[1]http://www.cairographics.org/
[2]http://hackage.haskell.org/package/cairo
[3]http://www.libsdl.org/index.php
[4]http://hackage.haskell.org/package/SDL

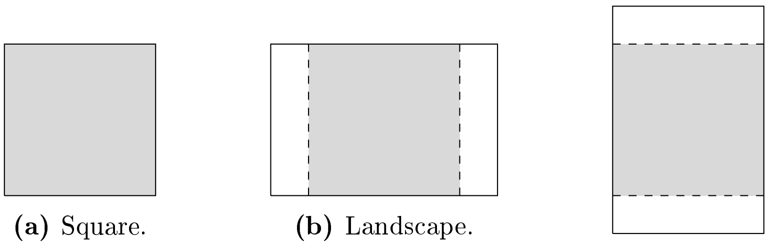
3. Design

The design choices we have made for our framework, correspond largely with those of the original paper. Because they form the basis for our implementation, we illustrate them in detail again along with some examples of usage.

3.1. Paths

In general, a path is a function of type $\mathbb{R} \to \mathbb{R}^2$ and since the domain of such a function is unbounded, it is per se not clear, where to begin and end a path. For this reason, we restrict the domain uniformly to $[0, 1]$. Therefore, in order to define a valid path, a function must be well-defined and continuous on at least this interval.

To satisfy our requirement of resolution independence, we choose the coordinate system of the codomain as follows: if the screen is square then the north west corner and south east corner match the points $\langle 1, 1\rangle$ and $\langle -1, -1\rangle$, respectively. The image of a path function is intended to lie within this square and in the following, we will refer to it as the *main viewport*. If the screen is non-square we simply extend the coordinate system along the longer axis. This way, we ensure that graphics maintain their aspect-ratio and the main viewport (shaded gray in Figure 3.1) is always visible.



(a) Square. **(b)** Landscape. **(c)** Portrait.

Figure 3.1.: The coordinate system for different aspect ratios.

Paths can be defined with the *path* constructor, which takes a function as described above. As an example, Listing 3.1 shows the definition of a circle path. A more complex example is the definition of a spiral path as done in Listing 3.2.

```
circlePath :: Path
circlePath = path $ \t → point (sin $ t ∗ 2 ∗ π) (cos $ t ∗ 2 ∗ π)
```

Listing 3.1.: Circle path definition.

```
spiralPath :: Path
spiralPath = path $ \t → let e = exp 1
                             f = (1/50) ∗ (e ∗∗ (s/10))
                             s = 6 ∗ π ∗ (1 + t)
                         in point (f ∗ (cos s)) (f ∗ (sin s))
```

Listing 3.2.: Spiral path definition.

Although the *path* constructor is very convenient for defining arbitrary paths, it takes much effort to express primitive ones such as lines or Bézier curves using it. Hence, we provide special constructors for these, namely *line*, *quad* and *cubic*, each taking two points as arguments denoting start and end of the path. The last two ones additionally require one or two control points, respectively, specifying the course of the curve to be given.

Despite all constructors mentioned so far it is still difficult to define paths that are expressible as a combination of other paths. A good example for this may be a polygon whose edges are lines. In order to simplify the definition of such paths, there is the *join* operation. Given a list of paths it returns a single one representing the concatenation of all list elements. The only condition each path understandably has to meet is that it has to start at its predecessor's end, otherwise a runtime error will be thrown. Listing 3.3 shows how to define a triangle using the *line* constructor along with the *join* operation.

```
trianglePath :: Path
trianglePath = join [(line a b), (line b c), (line c a)]
    where
        a = point 0 (−1)
        b = point 1 1
        c = point (−1) 1
```

Listing 3.3.: Triangle path definition.

At last, we provide the *text* constructor for defining paths on the basis of character sequences (also called strings). Taking a string, a font name and a size it produces a path representing the outline of the given string under consideration of the given parameters. The resulting path is positioned in the center of the screen, closed and cannot be joined with other paths. Note, that the size parameter is designed to be resolution-independent and therefore does not specify the font size in pixels which is the intended behavior in the original implementation but the size of 1em within our coordinate system. By the way, the original implementation only features the ability to define shapes (will be introduced in the next section) on the basis of strings, but not paths, thus eliminating the possibility to draw the outline of a string via invoking the *stroke* operation (again, see next section).

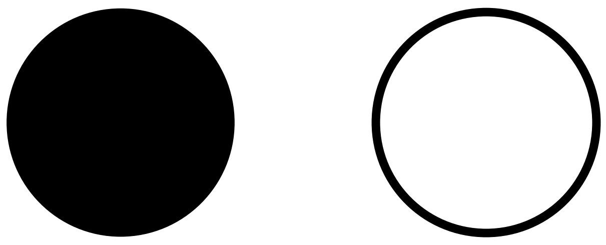
3.2. Shapes

Shapes are based on paths and describe an area that can later be filled using textures (see Section 3.3). There are basically two ways to create a shape: Either you fill a list of paths by using the *shape* constructor or you *stroke* a single path. When using the *shape* constructor, be sure that every given path is *closed*. One or more paths being not closed will result in a runtime error to be thrown, since such a path does not enclose an area. The *stroke* operation in contrast can also be applied to non-closed paths. Stroking a path imitates the result of drawing it with a pen. Therefore, the *stroke* operation requires in addition to the path itself a second parameter specifying the pen width. Listing 3.4 shows both these methods applied to the previously in Listing 3.1 defined *circlePath*. The resulting shapes can be seen in Figure 3.2.

```
filledCircle  :: Shape
filledCircle  = shape [circlePath]

strokedCircle :: Shape
strokedCircle = stroke circlePath $ 1/10
```

Listing 3.4.: Shape creation based on the circle path.



(a) Stroked circle path. **(b)** Filled circle path.

Figure 3.2.: Resulting shapes.

3.3. Textures

We have no possibility to declare the interior of a shape, yet. This is exactly what textures are for. To allow arbitrary textures while also ensuring resolution independence, they are generally described by a function that given a point returns the color at that point[2][3]. Thus, a texture function has the signature $\mathbb{R}^2 \to Color$.

To demonstrate the versatility of this approach, consider the HSV color space in which colors are defined by three components: hue, saturation and value. Hue determines the base color, saturation expresses the amount of gray and value specifies the brightness.For a constant value, hue and saturation are equivalent to polar coordinates of any point within our coordinate system. Having this in mind, we can define a texture function by first converting the Cartesian coordinates of a given point to polar coordinates and then return a color using the *hsv* constructor for colors. Listing 3.5 shows how to finally create a texture by giving our so-defined texture function as an argument to the *texture* constructor.

```
colorTexture :: Texture
colorTexture = texture $ \p → let hue = atan2 (coordinate p) (ordinate p)
                                  saturation = min 1 $ norm p
                              in hsv hue saturation 1
```

Listing 3.5.: Color texture definition.

To give another simpler example, think of a radial gradient. We can program such a texture using the *lerp* function for performing linear interpolation of two colors. As the interpolation parameter we simply pass the distance to the origin. The full texture definition is shown in Listing 3.6.

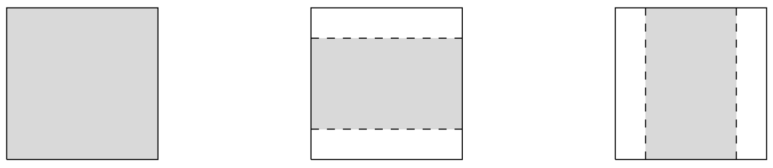
```
radialGradientTexture :: Texture
radialGradientTexture = texture $ \p → lerp red black $ norm p
```

Listing 3.6.: Radial gradient texture definition.

Sometimes you may want to define single-color textures. For this there is the *fillColor* constructor which taking a color returns such a texture.

In order to make our library suitable for daily use, we also provide two functions to create textures on the basis of image files: *file* and *load*. While *load* is an IO function and buffers the image data in the main memory for better performance but to the detriment of usability, *file* as a non-IO function is much more intuitively usable and occupies less memory during runtime at the cost of speed (see Subsection 4.4.2 for implementation details). Both functions support the most commonly used image file formats and take the alpha channel, if present, into consideration.

The image is always positioned in that manner that it covers the largest possible area within the main viewport while maintaining its aspect-ratio. Figure 3.3 visualizes this method for different aspect-ratios.



(a) Square image. **(b)** Landscape image. **(c)** Portrait image.

Figure 3.3.: Positioning of images within the main viewport.

At all points that do not lie within the so-positioned image the resulting texture function returns a fully transparent color.

3.4. Textured Shapes

As the name suggests, textured shapes are simply the combination of a shape and a texture. Hereby, the area defined by the shape is filled with the texture. For this reason the constructor for textured shapes is simply named *fill*. Remembering the previous definitions *filledCircle* and *colorTexture* (see Listing 3.4 respectively 3.5), we can produce a drawable object as demonstrated in Listing 3.7. The result can be seen in Figure 3.4.

```
colorCircle :: TexturedShape
colorCircle =  fill filledCircle colorTexture
```

Listing 3.7.: Textured shape definition example.

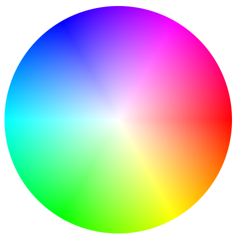


Figure 3.4.: Color circle.

We also provide the *image* constructor to offer a more direct way to create a textured shape on the basis of an image. Internally, the *file* constructor for textures (introduced in Section 3.3) is used.

3.5. Drawings

A drawing is a collection of textured shapes. It can be constructed by calling the *drawing* constructor which takes a list of textured shapes. An example is given by Listing 3.8.

```
myDrawing :: Drawing
myDrawing = drawing [colorCircle]
```

Listing 3.8.: Drawing definition example.

By using the *combine* function, you can join multiple drawings to a single one. This might be helpful when working with two or more drawings to represent different layers.

3.6. Transformations

Typically, traditional graphics frameworks just offer affine transformations, such as translation, rotation, scaling and shearing. Even though many use cases are covered by these, a whole range of interesting transformations is excluded. A more general and also more expressive model is to describe transformations as a function of type $\mathbb{R}^2 \to \mathbb{R}^2$.

In doing so, paths and hence shapes require the *forward* transformation, while textures require the *inverse* transformation. For example, to translate a textured shape to the right, we move every point belonging to its shape to the right by applying the forward transformation. But in order to get the right color at those points, remembering that textures are represented by a function that given a point returns the color at that point, we first have to apply the inverse transformation to be then able to query the color. For this reason, both the forward and backward transformation has to be specified when defining transformations via the *transformation* constructor. An example for the definition of such a transformation is given by Listing 3.9.

```
wave :: Transformation
wave = transformation (\p → let x = ordinate p
                                y = coordinate p
                            in point (x + (sin y)) y)
                      (\p → let x = ordinate p
                                y = coordinate p
                            in point (x − (sin y)) y)
```

Listing 3.9.: Wave transformation.

In the case, you want to define an affine transformation, there is the *affineTransformation* constructor. Since the inverse transformation can be easily computed for affine transformations, it only requires the forward transformation to be given. It is specified by two components: a 2x2 matrix representing the linear mapping and a point representing the translation. For a couple of affine transformations we offer predefined constructors, namely *identity*, *rotate*, *scale*, *shear* and *translate*.

To perform transformations on objects, we provide the *transform* operation. Taking a transformation and the object to be transformed, it returns the transformed object. All objects introduced so far are transformable. One benefit of having both directions of a transformation is that even transformations are transformable. Listing 3.10 shows how we can apply a scaling transformation to our wave transformation defined in Listing 3.9 to produce smaller waves.

```
scaledWave :: Transformation
scaledWave = transform (scale s s) wave
    where
      s = 1/30
```

Listing 3.10.: Scaled wave transformation.

Figure 3.5 can then be obtained by applying the scaled wave transformation to the color circle defined in Listing 3.7.

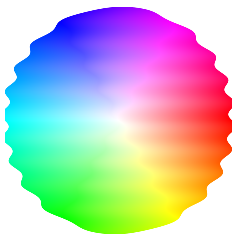


Figure 3.5.: Transformed color circle.

3.7. Window Management

Because, unlike Java (and thus Scala), Haskell does not include a widget toolkit, we were forced to decide on a completely different solution in this aspect. With respect to the goals set in Chapter 1 we wanted to provide a simple and easy-to-use way to create the application window and be able to react to events such as mouse clicks or key presses. Nevertheless, a programmer should still be able to focus on graphics programming.

To present our solution, Listing 3.11 shows the source code of an executable application. For simplicity we omitted the definition of *myDrawing* which can be found in Section 3.5. Its main method calls a method named *run* with a pair specifying the window resolution and a list of functions that should be called whenever the corresponding event occurs (therefore these function are referred to as *callbacks*). In this example, we just pass the *draw* method using the *Draw* constructor which signalizes that this function returns the drawing to be displayed. There are a few more configurable callbacks, listed in Table 3.1.

```
draw :: IO Drawing
draw = return $ myDrawing

main :: IO ()
main = run (640, 480) [(Draw draw)]
```

Listing 3.11.: Example application.

Constructor	Callback signature
Draw	IO Drawing
Init/Quit	IO ()
KeyDown/KeyDown	Key → [Modifier] → IO ()
MouseDown/MouseUp	Point → Button → IO ()
MouseMotion	Point → IO ()
MouseWheel	Point → Wheel → IO ()

Table 3.1.: Configurable callbacks

4. Implementation

Although the design is the same in many parts, our implementation differs significantly from the original one. This is partly because Haskell is a purely functional programming language, while Scala is multi-paradigmatic. Of course, another reason is that we had to use other libraries as a back-end, namely Cairo and SDL, since the originally used Java2D is not available in Haskell.

In this chapter, we first describe the basic concept of our implementation and then take a closer look at the implementation itself. Among other things we explain step-by-step how rendering is done. In particular, we discuss those points at which we had to choose other approaches than the original implementation or intercepted special cases in order to achieve a better performance.

4.1. Basic Concept

Similar to the original implementation, we implement the described approach by mapping it to existing 2D graphics frameworks (from now on referred to as *graphics hosts*). But unlike the original implementation, we use not only one but two libraries, as each alone do not cover the needed functionality. Figure 4.1 shows how our implementation works conceptually where gray indicates functionality from the graphics hosts.

Starting from a textured shape, the combination of a shape and a texture, our first step is to translate the shape into geometry, i.e., lines and Bézier curves (Sampling). This geometry is then used to fill or stroke the shape, depending on its actual type, and thereby obtain a raster telling us which pixels belong to the area defined by the shape (Filling/Stroking). We then simply iterate over these pixels and call the corresponding texture function for each pixel. This way a color raster representing the textured shape is produced (Texturing). Finally, this color raster is sent to the screen in order to be displayed (Blitting).

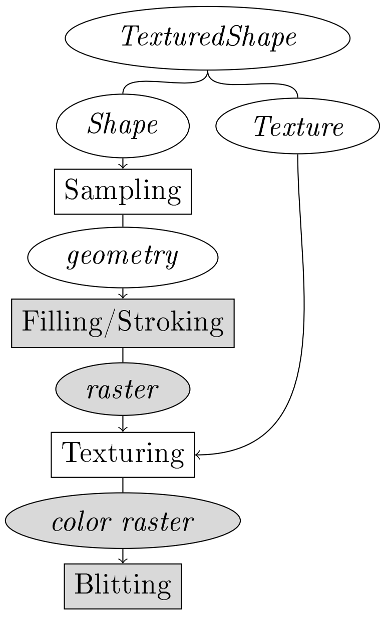


Figure 4.1.: Rendering pipeline.

4.2. Setup

When the *run* method is called, our first step is to set the number of cores to be used to the number of available cores. Usually, this has to be done manually[6] by running a program with *+RTS -N*. In order to provide a less complicated way for this and to be in line with our design objectives as described in Chapter 1, we want this number automatically to be set to the highest value possible. To do so, we use Haskell's FFI to import a GHC-internal function namend *getNumberOfProcessors* allowing us to obtain the number of available cores to which we then set the number of cores to be used by our program.

Next, we initialize the video and timer subsystems of SDL and create the main window. By invoking SDL's *setVideoMode* function we obtain the screen surface. With the *fill surface* and the *color surface* we create two additional (off-screen) surfaces. While the color surface, just like the screen surface, is managed by SDL, the fill surface is a Cairo surface. For each surface, there is a corresponding buffer holding the pixel data: the screen buffer, the fill buffer and the color buffer.

After calling the *Init* callback (see Table 3.1), if provided, we start the event loop within our self-defined *Run* monad (see Listing 4.1).

```
type Run = ReaderT Environment IO
```

Listing 4.1.: Run monad.

This monad is introduced in order to simplify our implementation and combines the *Reader* and *IO* monad. Instead of passing multiple arguments to all our subroutines, the monad provides an environment which can be obtained by invoking the *ask* function. The environment (represented by a record data type *Environment*) holds information such as the screen resolution and allows us to easily access the surfaces and buffers.

After the event loop exited, we call the *Quit* callback, if configured, and then quit our application.

4.3. Event Loop

The event loop is structured quite typically. First, we poll an event using SDL's *pollEvent* function, then we match its actual type using a *case of* expression and finally we execute the corresponding action. The body of the event loop can be seen in Listing 4.2.

In case the polled event is *SDL.Quit*, we simply exit the loop. In any other case, we process the event and afterwards, call again *eventLoop*. If there is no event to process (represented by *SDL.NoEvent*), we call the *Draw* callback to receive the drawing from the user application and pass it to the *renderDrawing* method. After the drawing has been rendered, we refresh the screen and calculate the frames per second (FPS) using SDL's timer subsystem. For any input event, such as a key press or a mouse click, we call the corresponding callback if configured.

```
eventLoop :: Run ()
eventLoop = do
    event ← liftIO $ SDL.pollEvent
    case event of
        SDL.Quit → return ()
        _ → do
            case event of
                SDL.NoEvent → ...
                ... → ...
                _ → return ()
            eventLoop
```

Listing 4.2.: Event loop body.

4.4. Rendering

We now focus on the main part of the implementation, the rendering process. In the last section, we explained how we receive the drawing from the user application and that it is passed to a function namend *renderDrawing*. This function is shown in Listing 4.4 and calls *renderTexturedShape* for each textured shape belonging to the drawing. This is done by invoking *mapM_* with the *renderTexturedShape* function and the list of textured shapes the *Drawing* constructor (see Listing 4.3) holds. As you can see in Listing 4.5, the data type *TexturedShape* is just the pair of a shape and a texture.

```
newtype Drawing = Drawing [TexturedShape]
```

Listing 4.3.: Data type for drawings.

```
renderDrawing :: Drawing → Run ()
renderDrawing (Drawing tss) = mapM_ renderTexturedShape tss
```

Listing 4.4.: *renderDrawing* method.

renderTexturedShape (shown in Listing 4.6) performs three steps on a given textured shape. First, the shape is drawn by invoking *renderShape*. Secondly, we apply the texture to the shape (*renderTexture*). And finally, we display the result on the screen. While the last step is simply done using a single SDL function named *blitSurface*, the other two steps are more complex. For this reason, we explain them in detail, starting with the first one.

4.4.1. Shapes

Internally, we distinguish three types of shapes, each having its own data constructor as shown in Listing 4.7. While the first two data constructors correspond to the func-

```
data TexturedShape = TexturedShape Shape Texture
```

Listing 4.5.: Data type for textured shapes.

```
renderTexturedShape :: TexturedShape → Run ()
renderTexturedShape (TexturedShape shape texture) = do
    renderShape shape
    renderTexture texture
    environment ← ask
    liftIO $ SDL.blitSurface (colorSurface environment) Nothing (screenSurface
        environment) Nothing
    return ()
```

Listing 4.6.: *renderTexturedShape* method.

tions *shape* and *stroke* introduced in Section 3.2, the third one, *TransformedStrokeShape* represents a stroked shape to which a transformation has been applied.

```
data Shape = Shape [Path] |
    Stroke Path Double |
    TransformedStroke Path Double Transformation
```

Listing 4.7.: Data type for shapes.

Our *renderShape* function uses pattern matching on these constructors. In the following three sections we explain, what actually is happens for each type of shape.

Filled Shapes

Listing 4.8 shows the *renderShape* function for the *Shape* constructor. The function *renderWithFillSurface* (shown in Listing 4.9) is a helper function that clears the fill surface and performs some basic setup (e.g., we set the draw color to black) on it before it starts executing the function provided as an argument, in this case *fillPaths*. *fillPaths* draws every path in the given list using a method named *drawPath* and then fills the drawn paths by invoking Cairo's *fill* operation.

Since the *drawPath* method is a central component of our implementation, we now explain it in detail. It is primarily responsible for mapping our design approach efficiently to the graphics hosts.

First, you have to know, how paths are represented within our implementation. Listing 4.10 shows this. The constructors for path are largely self-explanatory, since they correspond with the ones introduced in Section 3.1. Only the *Text* constructor has a fourth field for storing the applied transformation.

The *drawPath* method, just like the *renderShape* function, also uses pattern matching. Listing 4.11 shows the function. It expects a path, the width and height of the screen

```
renderShape (Shape ps) = renderWidthFillSurface fillPaths
    where
        fillPaths  w h = mapM_ (\p → do {drawPath p w h True ; Cairo.fill}) ps
```

Listing 4.8.: *renderShape* for the *Shape* constructor.

```
renderWidthFillSurface :: (Int → Int → Cairo.Render a) → Run a
renderWidthFillSurface rendering = do
    environment ← ask
    liftIO $ Cairo.renderWith (fillSurface  environment) $ do
        Cairo.setSourceRGB 0 0 0
        Cairo.paint
        Cairo.setSourceRGB 1 1 1
        Cairo.setLineCap Cairo.LineCapSquare
        Cairo.setLineJoin Cairo.LineJoinMiter
        rendering (width environment) (height environment)
```

Listing 4.9.: *renderWithFillSurface* helper function.

and a flag specifying if the path to be drawn marks the beginning of a new path or is part of another one.

A functional path is drawn as follows. First, we sample the path until the sampled points are so close to each other that their distance is smaller than the size of a pixel. Afterwards, we connect all these points with lines.

When we have to draw lines or Bézier curves, we just have to map the primitives to the corresponding Cairo function. Only quadratic Bézier curves have to be converted to cubic ones, since Cairo only supports drawing such.

Join paths are drawn by drawing all its sub paths. This is done by calling recursively the *drawPath* function on each sub path.

At last, text paths are left. The difficulty here is to obtain the outline of the text as lines and Bézier curves in order to be able to transform it if necessary. To achieve this, we first draw the text then read the path's data internally created by Cairo. For this we use some functions we had to import using Haskell's FFI, since the Cairo bindings do not offer them. Afterwards, we build join paths on the basis of this data and transform

```
data Path = FPath (Double → Point) |
    Line !Point !Point |
    Quad !Point !Point !Point |
    Cubic !Point !Point !Point !Point |
    Join [Path] |
    Text String String Double Transformation
```

Listing 4.10.: Data type for paths.

these join paths with the stored transformation. All this is done by the *convertText* function. These so-obtained join paths just have to be drawn what in turn is done by calling *drawPath* for each join path.

Stroked Shape

Stroke shapes are created when applying the *stroke* operation (see Section 3.2) to a path. The *Stroke* constructor holds the path to stroke and the pen width to stroke with.

What actually happens when calling *renderShape* with a stroked shape is shown in Listing 4.12. We first set the line width, we then draw the path using the *drawPath* method and at last, we stroke the drawn path by calling the *stroke* operation provided by Cairo.

Transformed Stroked Shape

As long as we do not apply transformations to stroked shape, everything works fine. However, if we do so, the line width may vary along the underlying path. For example, consider a rectangle. When we apply a scaling transformation to it, that scales by factor 2 in Y direction and leaves the X direction as it is, all horizontal edges become twice as thick as they were before while the vertical edges remain unchanged.

To draw such a shape, we first sample the underlying path to obtain a decomposition into small line segments. To each of these segments we then apply the inverse transformation of the stored transformation to obtain the untransformed counterpart of our line segment, starting at S and ending at E (see Figure 4.2). By calculating the boundary points L and R (their distance is equal to the originally defined line width) and transforming these boundary points using the forward tranformation, we obtain the boundary points of the transformed line segment. The distance of these transformed boundary points to each other is then used as the line width when drawing the corresponding line segment.

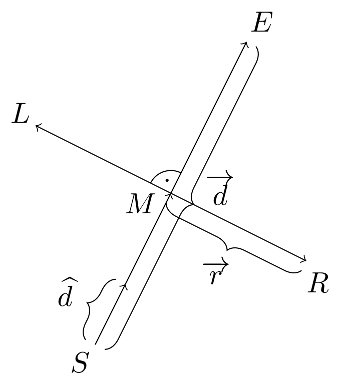


Figure 4.2.: Transformed stroked shape line width calculation.

This method mostly works well, but in some cases we produce incorrect output images (see Section 5.2).

4.4.2. Textures

When *renderTexture* is called, the previously called *renderShape* has already produced the raster telling us which pixels are inside the shape. As explained earlier, we iterate over these pixels. What actually happens when iterating over the pixels depends on the texture's type. Listing 4.13 shows the data type for textures. *FTexture* represents a functional texture while *File* describes an image texture. It is striking that in addition to the file path given by the *String* the *File* constructor has a field for a transformation. This is analogous to the *Text* constructor for text paths or the *TransformedStroke* constructor for shapes.

Multithreading

Instead of using only one thread to iterate over all pixels, we wanted to parallelize this step. Therefore, we split the screen into multiple chunks and iterate simultaneously on all these chunks.

The chunks are created by a self-defined function named *buildChunks*. We always create twice as many chunks as there are cores to achieve better processor utilization. Using *mapM_* together with *forkIO* (provided by the *Control.Concurrent.Thread.Group* package) on our list of chunks, we start multiple threads, each iterating on a certain number of pixels. We then wait for all of them to finish by calling *wait* (provided by the same package).

Functional Textures

When having a function texture, we first test whether the current pixel we iterate over is white. If this is the case, the pixel does not belong to the shape. Therfore, we set the corresponding pixel in the color buffer full transparent. If the current pixel is not white, we know that it is part of the shape's inside. We then convert the screen point to the corresponding point within our coordinate system in order to call the texture function. Thus, we obtain the color of the textured shape at that point. Finally, we combine the color with the value in the fill buffer, meaning that we would make the color half-transparent in case the corresponding pixel in the fill buffer was gray (this happens usually at the border of a shape due to anti-aliasing).

File Textures

If a image texture should be rendered, we first load the image to display into a SDL surface using SDL_image. When iterating over the pixels in the fill buffer, we do nearly the same as if we would do when rendering a functional texture, except that we obtain the corresponding color by reading the pixel data of the SDL surface instead of by calling a texture function.

4.5. Transformations

To handle transformations in an effective manner, we distinguish functional transformations and affine transformations. While the former are represented by two functions of type *Point* → *Point*, the latter is defined by two matrices (see Listing 4.14).

In order to be able to use the *transform* operation with different types (overloading), we introduced a type class named *Transformable* (see Listing 4.15). Every transformable object (see Section 3.6) provides an instance of this type class, defining how that object is transformed.

Listing 4.16 shows the *Transformable* instance definiton for the *Path* data type. Here, we intercept affine transformations as a special case, because map lines and Bézier curves again to lines and Bézier curves when applied.

```
drawPath :: Path → Int → Int → Bool → Cairo.Render ()
drawPath p@(FPath _) w h new = drawPoints $ sample p w h
    where
        drawPoints ((Point x y):ps) = do
            moveEventuallyTo x y new
            mapM_ (\(Point x y) → Cairo.lineTo x y) ps
            if (isClosed p) then Cairo.closePath else return ()
drawPath (Line s e) w h new = do
    moveEventuallyTo sx sy new
    Cairo.lineTo ex ey
        where
            (Point sx sy) = toScreen s w h
            (Point ex ey) = toScreen e w h
drawPath (Quad s e c) w h new = drawPath (Cubic s e c1 c2) w h new
    where
        c1 = s + ((2.0 / 3.0) |* (c − s))
        c2 = e + ((2.0 / 3.0) |* (c − e))
drawPath (Cubic s e c1 c2) w h new = do
    moveEventuallyTo sx sy new
    Cairo.curveTo c1x c1y c2x c2y ex ey
        where
            (Point sx sy) = toScreen s w h
            (Point ex ey) = toScreen e w h
            (Point c1x c1y) = toScreen c1 w h
            (Point c2x c2y) = toScreen c2 w h
drawPath j@(Join ps) w h new = drawPath' ps
    where
        drawPath' (p:ps) = do
            drawPath p w h new
            mapM_ (\p → drawPath p w h False) ps
            if (isClosed j) then Cairo.closePath else return ()
drawPath t@(Text _ _ _ _) w h _ = do
    joinPaths → convertText t w h
    mapM_ (\p → drawPath p w h True) joinPaths
```

Listing 4.11.: *drawPath* method.

```
renderShape (Stroke p l) = renderWidthFillSurface strokePath
    where
        strokePath w h = do
            Cairo.setLineWidth $ l * (fromIntegral $ min w h) / 2.0
            drawPath p w h True
            Cairo.stroke
```

Listing 4.12.: *renderShape* for the *Stroke* constructor.

```
data Texture = FTexture (Point → Color) |
    File String Transformation
```

Listing 4.13.: Data type for textures.

```
data Transformation = FTransformation (Point → Point) (Point → Point) |
    AffineTransformation Matrix Matrix
```

Listing 4.14.: Transformation data type.

```
class Transformable a where
    transform :: Transformation → a → a
```

Listing 4.15.: *Transformable* type class.

```
instance Transformable Path where
    transform f (FPath p) = FPath $ (transform f) . p
    transform f@(AffineTransformation _ _) (Line s e) =
        Line (transform f s) (transform f e)
    transform f@(AffineTransformation _ _) (Quad s e c) =
        Quad (transform f s) (transform f e) (transform f c)
    transform f@(AffineTransformation _ _) (Cubic s e c1 c2) =
        Cubic (transform f s) (transform f e) (transform f c1) (transform f c2)
    transform f (Join ps) = Join [(transform f p) | p → ps]
    transform f (Text str font size f') = Text str font size (compose f f')
    transform f p = transform f $ toFPath p
```

Listing 4.16.: *Transformable* instance for paths.

5. Evaluation

In this chapter we evaluate our implementation, mainly by comparing it to the original one considering performance and image quality.

5.1. Performance

Before comparing our implementation to the original one, we want to find out to what extent multithreading speeds up the performance when rendering textures (see Subsection 4.4.2). To do so, we rendered a complex texture, namely the Mandelbrot set, which is also provided as a demo application, at a screen resolution of 1024x1024 using a various number of cores[1].

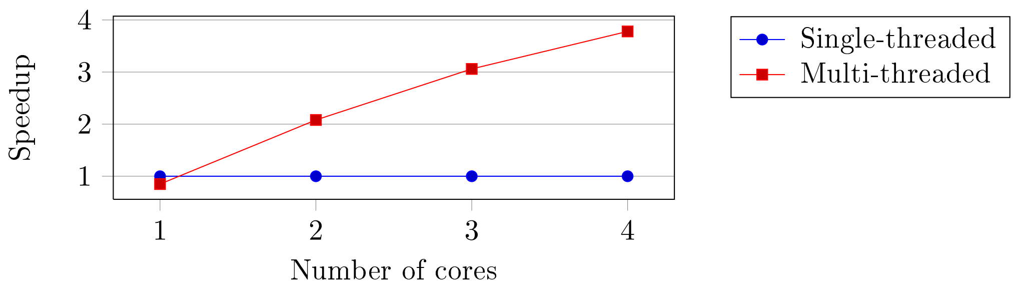


Chart 5.1: Speedup due to multithreading

Chart 5.1 shows that the performance increases almost linearly with the number of cores when multithreading is enabled. But if only one core is available multithreading leads to an up to 15 percent worse performance compared to the single-threaded variant. This is because multithreading procudes amn overhead that reduces the performance on a single-core processor.

In order to compare our implementation's performance to to that of the original one, we ran several benchmarks[2].

The first one is to test the path stroking performance. For this, it renders 50 lines and Bézier curves, 30 join paths, 30 functional paths and 20 text paths, all randomly created.

As we can see in Chart 5.2, our implementation a slower in the first two cases, but about 12 times faster in stroking functional paths. The last value is missing for the

[1]Processor: Intel Core i7-3615QM 4x2.3GHz, Main memory: 8GB, Operating system: Ubuntu 12.04 LTS 64-Bit, GHC version: 7.4.1

[2]Processor: Intel Core 2 Duo T6400 2x2.0GHz, Main memory: 2GB, Operating system: Ubuntu 12.04 LTS 64-Bit, GHC version: 7.4.1

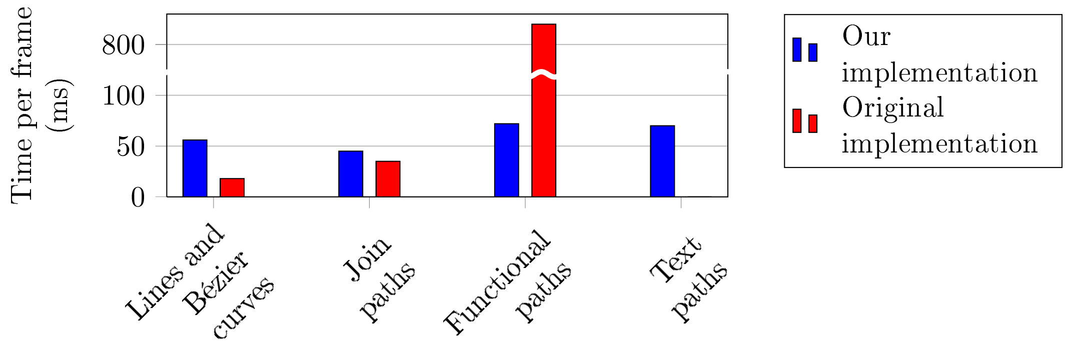


Chart 5.2: Path stroking performance

original implementation, since it is unable to stroke text path.

Next, we tested the performance in path filling. Similar to the first benchmark, we render 30 join paths, 30 functional paths and 20 text paths and fill them.

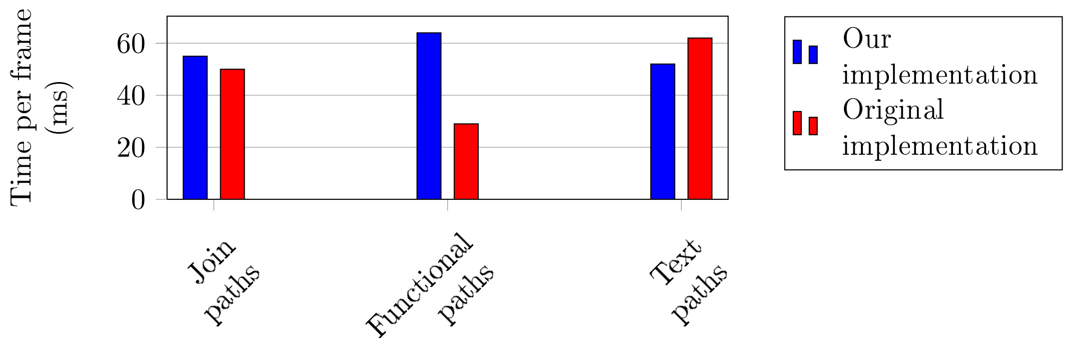


Chart 5.3: Path filling performance

Chart 5.3 shows that again, our implementation is slower than the original one (except in case of text rendering).

Finally, we compared the texturing performance. For this, we rendered a functional texture (the previously defined *colorTexture*, see Listing 3.5), a single-color texture and two image textures, a small one (16x16 pixels) and a larger one (512x512 pixels).

The results can be seen in Chart 5.4. For unknown reasons, our implementation renders functional textures significantly slower than the original one, even though we tried to optimize this process by parallelizing it and adding strictness annotations to the source code. Our implementation's performance in this case is more than three times worse. In the other three cases our implementation is about half as fast as the original one.

All in all, our implementation's performance is obviously not as good as the original one's.

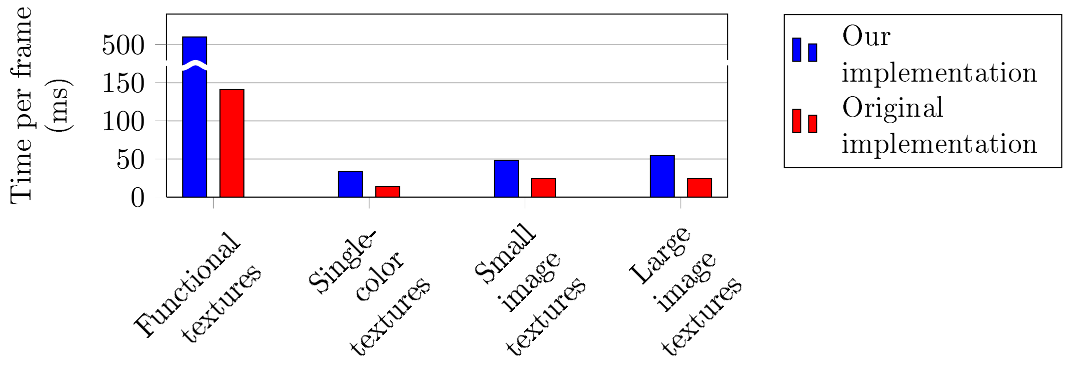
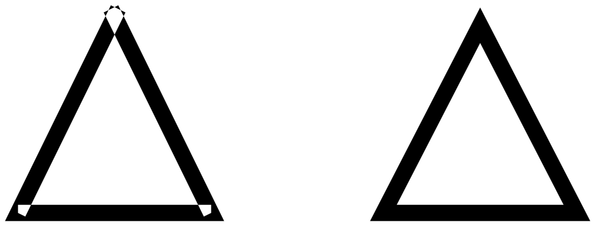


Chart 5.4: Texture rendering performance

5.2. Image Quality

Besides performance, we also compared the quality of the produced images. Even though both implementations are based upon almost the same design choices, there are some differences in image quality due to the different realization.

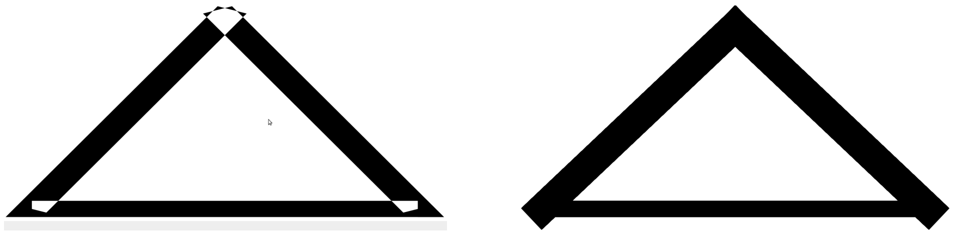
At first, we stroked a triangle path as defined in Listing 3.3 and compare the output images of both implementations to each other. As you can see in Figure 5.1a, the original implementation produces unwanted white areas at the triangle's corners where the lines representing its edges overlap. This presumably arises from applying the wrong filling rule (*even-odd rule* instead of *non-zero winding rule*). Another problem is that the upper corner is not properly connected what in turn results in a non-closed triangle. It seems as if the original implementation does not test whether the path to stroke is closed or not. As Figure 5.1b shows, our implementation is not affected by any of these two issues.



(a) Original implementation. **(b)** Our implementation.

Figure 5.1.: Comparison of stroked triangles.

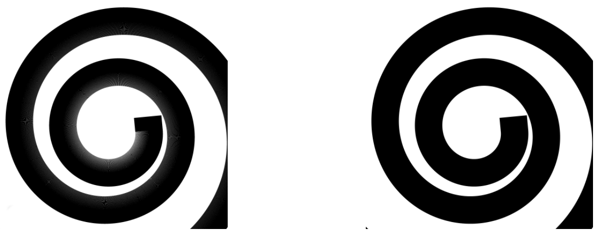
Secondly, we compared stroked triangles to which a scaling transformation had been applied. Figure 5.2a shows that the original implementation suffers again from the same problems as in the first test case, but the dynamically changing line width was considered correctly. Our implementation produces the image shown in Figure 5.2b. Since we decompose the underlying path into small line segments and draw each of these line segments with an individually calculated line width (see Subsection 4.4.1), we no longer know where edges appear and are therefore unable to connect them properly. This results in the inaccurate output image shown here.



(a) Original implementation. **(b)** Our implementation.

Figure 5.2.: Comparison of transformed stroked triangles.

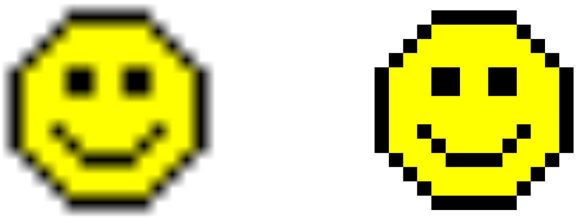
Next, we compared stroked spirals. The original implementation produces the output image shown in Figure 5.3a in which a type of interference pattern appears at the inside of the curve. Most likely, this is caused again by using the wrong filling rule. The output image generated by our implementation is shown in Figure 5.3b and looks as expected.



(a) Original implementation. **(b)** Our implementation.

Figure 5.3.: Comparison of stroked spirals.

At last, we displayed an image in both implementations. Since the size of the image was chosen smaller than the screen's resolution, the image had to be resized. The difference that can be seen in Figure 5.4 results from the fact that our implementation uses nearest-neighbor interpolation while the original one uses bilinear interpolation.



(a) Original implementation. **(b)** Our implementation.

Figure 5.4.: Comparison of image textures.

Overall, the differences in image quality are considered to be minimal, not least because both implementations produce identical output images in all other test cases than those shown here. Although we have shown examples in which the original implementation, unlike ours, produces faulty output images, ultimately both implementations are not entirely accurate and thus to be regarded as equal considering their image quality.

6. Conclusion

In the final chapter, we review the results and give an outlook on future works.

6.1. Results

The main goal of this thesis was to implement a library following the approach originally presented by Paul Klint and Atze van der Ploeg. In view of what we have achieved, it can be said that we have attained the objectives set in Chapter 1. We provide a library allowing programmers to realize resolution-independent 2D graphics in a simple yet expressive and composable way. In some respects, our implementation is even more flexible or kept simpler than the original one, for example text is realized as a path and window management has shrunk to a single invocation of the *run* method.

But there is no denying the fact that our library's performance is in need of improvement, especially when using functional textures. An approach to improve the overall performance is presented and discussed in the next section.

Another fact to mention is that constructive solid geometry operations, such as union, difference or intersection, are missing in our implementation. They offer an elegant way to define and compose shapes, but have not been implemented due to their complexity.

6.2. Outlook

Last but not least, we want to discuss some ideas for future works. Since performance is the main drawback of our implementation, it would make sense to concentrate efforts on this aspect. An approach to this problem is speeding up path rendering, i.e., filling and stroking of paths. One possible way to increase the path rendering performance lies in the use of GPUs, since they are very powerful and predestined for such tasks. For example, there is a proprietary extension to OpenGL called *NV_path_rendering* providing full GPU-acceleration for path rendering[7]. Unfortunately, this extension is only available on CUDA-capable GPUs by Nvidia and therefore not usable with all graphics boards. Regardless, it would be interesting to pursue this idea.

Optimizing the texturing step would also be worthwhile, especially because this is the major bottleneck of our implementation. For example, more strictness information (see Subsection 2.1.1) could be added to the source code. Also, profiling[1] might help to find the causes for the insufficient texture rendering performance. Once they were identified, they could be eliminated.

[1]http://www.haskell.org/haskellwiki/How_to_profile_a_Haskell_program

Apart from the performance, missing features could be added, e.g., one could implement constructive solid geometry operations (see Section 6.1).

Finally, it would be desirable to improve the image quality when drawing transformed stroked shapes (see Figure 5.2b in Section 5.2). To achieve this, the method for drawing transformed stroked shapes would have to be changed. If the border of a transformed stroke shape could be obtained as a closed path, we could use our already existing method to draw filled shapes in order to produce a correct output.

A. Installation and Usage

A.1. Installation

The distributed source directory contains a *.cabal* file, so you simply have to run *cabal*, in order to compile and install our library:

```
$ cabal configure --ghc
$ cabal build
$ cabal install
```

Alternatively, you can use the provided *Setup.hs* via *runhaskell*:

```
$ runhaskell Setup.hs configure --ghc
$ runhaskell Setup.hs build
$ runhaskell Setup.hs install
```

As the *--ghc* switch already indicates, the GHC is required, since we make use of GHC-specific language features.

A.2. Usage

After the installation has been successfully completed, you can use our library in your own applications by simply importing it with the following single line of code:

```
import Deform
```

Moreover, you can compile and run the included sample applications, which are to be found in the *demos* folder. Since many of them are kept very simple and therefore easy to understand, they may serve as a good starting point to write your own applications using our library.

All applications that use our library should be compiled at least with the *-threaded* switch to enable multithreading and thereby benifit from the optimizations we have made in the texturing step of the rendering pipeline (see Subsection 4.4.2). We recommend to use the *-O2* and *-optc-O3* options as well in order to achieve the best performance possible:

```
ghc --make -threaded -O2 -optc-O3 <source file>.hs
```

B. API Reference

Subsequently, we give a brief overview over the constructors and operations provided by our library. Each function is listed along with its type signature. For a full list of all available functions and data types including detailed explanations take a look at the documentation which can be easily generated by invoking *cabal* with the *haddock* argument:

```
$ cabal haddock
```

Again, this can also be done by executing the *Setup.hs* file:

```
$ runhaskell Setup.hs haddock
```

Colors

Function		Signature
hsv	::	Double → Double → Double → Color
hsva	::	Double → Double → Double → Double → Color
rgb	::	Double → Double → Double → Color
rgba	::	Double → Double → Double → Double → Color

Drawings

Function		Signature
combine	::	[Drawing] → Drawing
drawing	::	[TexturedShape] → Drawing

Linear Interpolation

Function		Signature
lerp	::	$A \to A \to$ Double $\to A$
		where $A \in$ {Color, Double, Point}
lerpNoAlpha	::	Color → Color → Double → Color

Paths

Function		Signature
cubic	::	Point → Point → Point → Point → Path
isClosed	::	Path → Bool
isConnected	::	Path → Path → Bool
join	::	[Path] → Path
line	::	Point → Point → Path
path	::	(Double → Point) → Path
quad	::	Point → Point → Point → Path
text	::	String → String → Double → Path

Points

Function		Signature
(.*)	::	Point → Point → Double
(|*)	::	Double → Point → Point
(*|)	::	Point → Double → Point
(/|)	::	Point → Double → Point
coordinate	::	Point → Double
distance	::	Point → Double
distanceSquared	::	Point → Double
norm	::	Point → Double
normSquared	::	Point → Double
ordinate	::	Point → Double
point	::	Double → Double → Point

Shapes

Function		Signature
shape	::	[Path] → Shape
stroke	::	Path → Double → Shape

Textured Shapes

Function		Signature
fill	::	Shape → Texture → TexturedShape
image	::	String → TexturedShape

Textures

Function		Signature
file	::	String → Texture
fillColor	::	Color → Texture
load	::	String → IO Texture
texture	::	(Point → Color) → Texture

Transformations

Function		Signature
affineTransformation	::	(Double, Double, Double, Double) → Point → Transformation
compose	::	Transformation → Transformation → Transformation
identity	::	Transformation
inverse	::	Transformation → Transformation
lens	::	Point → Norm → Profile → Double → Double → Double → Transformation
rotate	::	Double → Transformation
scale	::	Double → Double → Transformation
shear	::	Double → Double → Transformation
transform	::	Transformation → A → A **where** $A \in$ {Drawing, Path, Point, Shape, Texture TexturedShape, Transformation}
transformation	::	(Point → Point) → (Point → Point) → Transformation
translate	::	Double → Double → Transformation

Variables

Function		Signature
get	::	Variable a → IO a
modify	::	Variable a → $(a \to a)$ → IO ()
set	::	Variable a → a → IO ()
variable	::	a → Variable a

Window Management

Function		Signature
export	::	String → IO Bool
run	::	(Int, Int) → [Callback] → IO ()

Bibliography

[1] Klint, P. and van der Ploeg, A.: *A Library for Declarative Resolution- Independent 2D Graphics*. Lecture Notes in Computer Science. Springer, 2013.

[2] Karczmarczuk, J.: *Functional approach to texture generation*. In Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages, PADL'02, pages 225-242, London, UK, UK, 2002. Springer-Verlag.

[3] Elliott, C.: *Functional image synthesis*. In Proceedings of Bridges, 2001.

[4] *Performance/Strictness*, http://www.haskell.org/haskellwiki/Performance/Strictness, 23.09.2013

[5] *Haskell/Strictness*, http://en.wikibooks.org/wiki/Haskell/Strictness, 23.09.2013

[6] *Concurrency*, http://www.haskell.org/haskellwiki/Concurrency, 17.09.2013

[7] *NV Path Rendering*, https://developer.nvidia.com/nv-path-rendering, 27.09.2013