A Library for Declarative Resolution-Independent 2D Graphics

Paul Klint and Atze van der Ploeg

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{paulk,ploeg}@cwi.nl

Abstract. The design of most 2D graphics frameworks has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. As a result, such frameworks restrict expressivity by providing a limited set of shape primitives, a limited set of textures and only affine transformations. For example, non-affine transformations can only be added by invasive modification or complex tricks rather than by simple composition. More general frameworks exist, but they make it harder to describe and analyze shapes. We present a new declarative approach to resolution-independent 2D graphics that generalizes and simplifies the functionality of traditional frameworks, while preserving their efficiency. As a real-world example, we show the implementation of a form of focus+context lenses that gives better image quality and better performance than the state-of-the-art solution at a fraction of the code. Our approach can serve as a versatile foundation for the creation of advanced graphics and higher level frameworks.

Keywords: Declarative Graphics, Design, Resolution-Independence, Optimization, Focus+context lenses

1 Introduction

The design of traditional 2D graphics frameworks, such as Java2D[1] and Processing[2], has been guided by what the computer can draw efficiently, instead of by how graphics can best be expressed and composed. This hinders the ease of programming 2D graphics, since it requires the programmer to express his ideas using the limited vocabulary that has emerged as a result of the focus on procedural optimization of such frameworks.

Suppose we have programmed a visualization in such a traditional framework and we now want to add a focus+context lens, such as the one shown in Figure 1. Since only affine transformations (that take parallel lines to parallel lines) are supported, we cannot add this transformation in a compositional way: it requires trickery or invasive modification.

Instead of worrying about such low-level details, it is desirable to program 2D graphics in a declarative way that is general, simple, expressive, composable and resolution-independent while still being efficient. Previous research on declarative graphics has yielded many elegant approaches to 2D graphics, but none of these exhibit all these traits. This not only restricts direct graphics programming, but it also hinders the creation of higher-level frameworks. For example, during

[1] `http://docs.oracle.com/javase/6/docs/technotes/guides/2d/`
[2] `http://processing.org`

Fig. 1. An example focus+context lens (zoomfactor = 2.5).

our efforts on the Rascal figure library[1], a high-level framework for software visualization, we noticed that our design was influenced by the limitations of the procedural framework used and hence could not grow further in terms of expressiveness and compositionality.

We present a new declarative approach that generalizes and simplifies the functionality of traditional 2D graphics frameworks, while preserving their efficiency. This is achieved by a very effective mapping of our approach to an existing 2D graphics framework (which we will call the *graphics host*). Our approach allows more expressive freedom and can hence serve as a more versatile foundation for advanced 2D graphics and higher-level frameworks. It is available as a library called *Deform*[3] for Scala. Our contributions are:

- The motivation (Section 2) and design (Section 3) of a small, simple and powerful framework for resolution-independent 2D graphics that enables composability and expressiveness.
- A way to implement and optimize this framework (Section 4) by mapping it to a readily-available, highly optimized graphics host. This includes optimizations to speed up this mapping and a way to support clipping so that large scenes can be rendered in real-time.
- An implementation of focus+context lenses that is faster and gives better image quality than the state-of-the-art approach (Section 5). This also acts as a validation of our work.

We discuss open questions in Section 6 and conclude in Section 7.

2 Exploring the Design Space

We now discuss design choices for declarative 2D graphics frameworks and to guide our choices, we use the following design goals:

- *Simplicity*: The programmer should not be overwhelmed by concepts and functions described in inch-thick manuals.
- *Expressivity*: Arbitrary graphics can be expressed in a *natural* way, without the need to encode them in lower-level concepts.
- *Composability*: Graphics can be composed and transformed in general ways.

[3] `https://github.com/cwi-swat/deform`

– *Resolution-independence*: Graphics can be expressed independent of resolution, so that they can be rendered at any level of detail.
– *Analyzability*: The concrete geometry of a shape can be obtained, for example as a list of lines and Bézier curves, so that we can define functions that act on this information to create derived graphics.
– *Optimizability*: Efficient algorithms for 2D graphics can be re-used.

Our analysis now focuses on how to represent *shapes*, *textures* and *transformations*, in the way that has the best fit with our design goals.

2.1 Shapes

Most frameworks offer a fixed set of geometric constructs, such as lines, Bézier curves and circle segments, that can be used to describe the *border* of shapes. For example, a regular polygon with k vertices can be expressed as follows:

$$\begin{array}{l} \mathit{regpolyg}(k) = [\mathit{line}(\mathit{onCircle}(i \times p), \mathit{onCircle}((i+1) \times p)) \mid i \leftarrow [0 \ldots k-1]] \\ \qquad\qquad \textbf{where } \mathit{onCircle}(x) = \langle \sin(x), \cos(x) \rangle, \quad p = (1/k) \times 2 \times \pi \end{array}$$

Here $\langle x, y \rangle$ denotes a point in $\mathbb{R}^2$. A downside of this approach is that shapes that are not compositions of such geometric constructs, such as sine waves, cannot be expressed. Instead, they have to be approximated *when specifying the shape*, which does not give a resolution-independent description of the shape.

A second approach is to describe the border of a shape as a *parametric* curve: a function from $\mathbb{R}$ to $\mathbb{R}^2$. For example, the border of the unit circle can be described by $c(t) = \langle \sin(t \times 2 \times \pi), \cos(t \times 2 \times \pi) \rangle$ on the interval $[0, 1]$. This can be seen as a generalization of using a fixed set of geometric constructs: each geometric construct can be described by a parametric curve and hence a combination of geometric constructs gives rise to a piecewise defined function. For this reason the expression of a regular polygon with k vertices is exactly the same as when using a fixed set of geometric constructs. Although a parametric description does not immediately give an analyzable description of the shape, we can sample the (resolution-independent) function to obtain such a description.

The third and final approach is to describe a shape *implicitly*: as a function that given a point in $\mathbb{R}^2$ tells us whether the point is inside the shape or not. For example, the implicit representation of the unit circle is $c(p) = |p| \leq 1$, where $|p|$ denotes the Euclidian norm. A downside of this approach is that it is often hard to encode a shape in this way. For example, as noted in [2], it requires an arcane insight to understand that the following also represents a regular polygon with k vertices.

$$\begin{array}{l} \mathit{regpolyg}\ (k, \langle x, y \rangle) = (x - j) \times (\sin(q+p) - i) - (\cos(q+p) - j) \times (y - i) \leq 0 \\ \qquad \textbf{where } p = 2 \times \pi / k, \quad q = p \times \lfloor \mathit{atan2}(y, x)/p \rfloor, \quad i = \sin(q), \quad j = \cos(q) \end{array}$$

It is also hard to analyze a shape that is described in this way, since we do not have a representation of the border of the shape.

If we could automatically switch between the parametric and implicit representations we would not have to make a choice between them. However, transforming a parametric representation into an implicit one or vice-versa is non-trivial, especially when the functions are not limited to a certain class. In fact, these are well-known and thoroughly studied problems [3]. In general, exact conversion is

possible for certain classes of functions [4], while other classes of functions require approximate techniques [5]. Since the implicit representation makes it hard to express and analyze shapes, and since it is hard and computationally expensive to automate the conversion between the two representations we have chosen to describe shapes parametrically.

2.2 Textures

Most frameworks offer a fixed set of textures, such as fill colors, images and gradients. Another approach is allow arbitrary textures by specifying the colors of its pixels, but this is not a resolution independent approach. A general, resolution independent way to describe a texture, and the one that we adopt, is by a function that given a point returns the color of the texture at that point [6, 2]. Notice that this way of expressing textures bears resemblance to implicitly defined shapes: implicitly defined shapes are functions of type $\mathbb{R}^2 \rightarrow \mathit{Boolean}$, whereas such textures are functions of type $\mathbb{R}^2 \rightarrow \mathit{Color}$.

2.3 Transformations

Typically, graphics frameworks offer only affine transformations, such as translation, rotation and scaling. Although these transformations cover many use cases, they preclude a whole range of interesting transformations, such as focus+context lenses. A more expressive model is to describe transformations simply as a function from $\mathbb{R}^2$ to $\mathbb{R}^2$.

Parametrically described shapes then require the *forward* transformation, while textures and implicitly defined shapes require the *inverse* transformation. For example, to translate a parametrically defined shape to the right, we define a function that given a parameter first gets the corresponding point on the border of the shape and then applies the forward transformation to that point, which moves the point to the right. To translate a texture to the right, we define a function that given a point first applies the inverse transformation, which moves the point to the left, and then queries the texture at that point. In the same fashion, the inverse transformation is also needed to transform implicitly defined shapes.

If we limit ourselves to affine transformations, obtaining both directions of a transformation is not a problem since such transformations are easily inverted. However, if we allow arbitrary transformations we need to either describe all shapes implicitly and use only the inverse transformation, making it harder to describe shapes, or describe shapes parametrically in which case we need *both* the forward transformation and the inverse transformation, making it harder to describe transformations. We conjecture that shapes are more likely to be application-specific than transformations, which can often be reused. Hence, we have chosen to represent shapes parametrically and require a definition of both directions for transformations.

2.4 Comparison

As a comparison, Table 1 lists the choices made by us and other frameworks. *Traditional* frameworks, like as Java2D, Processing and many others, limit the expressivity of the programmer by only providing support for the most common

		Traditional	Func. image synthesis	Vertigo	Deform
Shapes	Fixed	•			
	Parametric			•	•
	Implicit		•		
Textures	Fixed/pixels	•			
	Function		•		•
Transformations	Affine	•			
	Function			•	•
	Function^{-1}		•		•

Table 1. Design choices for graphics libraries.

use cases. Many declarative graphics frameworks[4] make the same choices [7, 8]. *Functional image synthesis* frameworks, such as Pan [6] and Clastic [2], are based on the notion that an image is simply a function from a point to a color. This allows the elegant definition of many interesting visual mathematical graphics but precludes real-life graphics, since the requirement of implicitly defined shapes makes hard to define complex shapes such as letters. *Vertigo* [9] is an elegant declarative framework for the geometric modeling of 3D shapes, without texturing. In *Deform* we have chosen a combination of design decisions that has not yet been explored: parametric shapes, textures as functions and general transformations. In the rest of this paper we show that this allows us to define a simple, general and resolution-independent framework which is applicable to real-life graphics.

3 Design

It is time to present our approach and illustrate its usage via examples. The basic unit of our framework is a *TexturedShape*, that describes a shape and the texture of its interior. An expression constituting a list of such textured shapes is first *created* using the constructors given in Table 2 and then *displayed* by a render function which interprets the constructors and produces an image. We will now show how to express shapes, textures and transformations in this way. Our examples were programmed in Scala and then hand-transformed into a custom notation which should be easy to understand. The examples use the constructors in Table 2 and some library functions of Deform, both of which will be explained when used.

3.1 Shapes

The basis for describing shapes is the *path* constructor, which takes a parametric description of the border of the shape, a function of type $\mathbb{R} \to \mathbb{R}^2$. To allow omission of the domain of this function, it simply must be $[0, 1]$. The *shape* constructor can then be used to create a shape from a list of *closed* paths, paths of which the start and end points are the same. If one of the paths is not closed, then it does not define an area and a run-time error will be thrown. A point is then inside the shape if it is inside any of its closed paths.

[4] Unfortunately, space limitations do not allow a more extensive discussion.

Constructor	*Type*
path	$(\mathbb{R} \to \mathbb{R}^2) \to \mathit{Path}$
shape	$[\mathit{Path}] \to \mathit{Shape}$
analyze	$\mathit{Path} \times (\mathit{ConcreteGeom} \to A) \to A$
	where $A \in \{\mathit{Path}, \mathit{Shape}, \mathit{Texture}, \mathit{TexturedShape}, \mathit{Transformation}\}$
color	$\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathit{Color}$
texture	$(\mathbb{R}^2 \to \mathit{Color}) \to \mathit{Texture}$
fill	$\mathit{Shape} \times \mathit{Texture} \to \mathit{TexturedShape}$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to \mathit{Transformation}$

Table 2. Constructors and functions. $[A]$ indicates a list of As.

As a basic example, consider a circle:

$$\mathit{circ} = \mathit{shape}([\mathit{path}(\lambda t \to \langle \sin(t \times 2 \times \pi), \cos(t \times 2 \times \pi)\rangle)])$$

The coordinate system of our framework is as follows: if the screen is square then the north west corner of the screen is $\langle -1, -1\rangle$ and the south east corner is $\langle 1, 1\rangle$. If the screen is non-square the range of the longest axis is adopted so that graphics maintain their aspect-ratio. An example of a more complex path is the spiral shown in Figure 2(a):

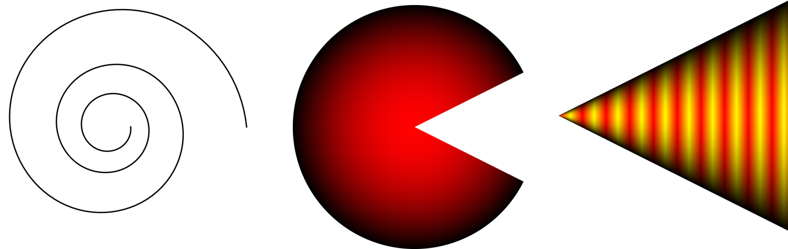
$$\mathit{spiral} = \mathit{path}(\lambda t \to \langle f \times \cos(s), f \times \sin(s)\rangle$$
$$\textbf{where } f = 1/50 \times e^{s/10}, \quad s = 6 \times \pi \times (1 + t)$$

Paths themselves cannot be drawn as they do not define an area. Hence, to produce a drawing of this spiral we use the *stroke* library function to convert this path to a shape given the width of the “pen”:

$$\mathit{stroke}(\mathit{spiral}, 1/200)$$

We do not have to explicitly define a parametric representation for each shape. Instead, we provide library functions that mimic the geometric constructs found in traditional libraries. For example, we can create a triangle as follows:

$$\mathit{triangle} = \mathit{shape}([\mathit{join}([\mathit{line}(a, b), \mathit{line}(b, c), \mathit{line}(c, a)])])$$
$$\textbf{where } a = \langle 0, 0\rangle, \quad b = \langle 1, \tfrac{1}{2}\rangle, \quad c = \langle 1, -\tfrac{1}{2}\rangle$$



(a) A simple spiral (b) Circle with triangle subtracted (c) A filled triangle

Fig. 2. Basic examples

To define functions which act on the geometry of a path, such as the *stroke* function, we offer the *analyze* constructor which takes a path and a function transforming the concrete geometry of the path, namely a list of lines and Bézier

curves, into a path, shape, texture, textured shape or transformation. To ensure resolution-independence, *analyze* is a constructor rather than a function: in this way we delay the sampling of the path until we know the desired resolution, namely when the renderer runs. We also use this constructor to define resolution independent *constructive solid geometry operations* on shapes, set operations such as union and intersection operating on the set of points inside a shape. The implementation of these operations involves analyzing the intersections between the concrete geometry of both shapes. As an example, the shape in Figure 2(b) can be obtained as follows:

$$\mathit{pacman} = \mathit{subtract}(\mathit{circ}, \mathit{triangle})$$

3.2 Textures

To declare the interior of a shape, a texture can be created with the *texture* constructor, which requires a function from a point to a color. A *color* is a value with four numbers, all in the range $[0, 1]$, namely red, green, blue and alpha (transparency). For example, consider the following colors:

$$\mathit{red} = \mathit{color}(1, 0, 0, 1), \mathit{black} = \mathit{color}(0, 0, 0, 1), \mathit{yellow} = \mathit{color}(1, 1, 0, 1)$$

We can now create a radial gradient as follows:

$$\mathit{radgrad} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{red}, x^2 + y^2, \mathit{black}))$$

Where *lerp* performs linear interpolation of two colors on each of the four numbers. A *TexturedShape* can then be created using the *fill* constructor. For example, Figure 2(b) shows:

$$\mathit{fill}(\mathit{pacman}, \mathit{radgrad})$$

As another example of defining textures in our framework, consider the interior of the triangle shown in Figure 2(c). For this texture, we first declare a one-dimensional cyclic gradient that cycles between red and yellow:

$$\begin{aligned} \mathit{gradient}(x) = \ & \textbf{if } l \leq \tfrac{1}{2} \textbf{ then } \mathit{lerp}(\mathit{red}, 2 \times l, \mathit{yellow}) \\ & \textbf{else } \mathit{lerp}(\mathit{yellow}, 2 \times (l - \tfrac{1}{2}), \mathit{red}) \\ & \textbf{where } l = x - \lfloor x \rfloor \end{aligned}$$

We can then define the filling of the triangle as follows:

$$\mathit{tritex} = \mathit{texture}(\lambda\langle x, y\rangle \rightarrow \mathit{lerp}(\mathit{gradient}(x \times 10), (2 \times |y|/x)^2, \mathit{black})$$

Where $x \times 10$ repeats the gradient ten times on the horizontal $[0, 1]$ interval and the linear interpolation argument[5] $(2 \times |y|/x)^2$ ensures that the color becomes darker closer to the vertical border of the triangle. A further survey of the power of this way of describing textures is beyond the scope of this paper, for some fascinating examples see [6] and [2].

3.3 Transformations

The *transformation* constructor can be used to describe arbitrary transformations and requires the forward transformation function and its inverse. For example, we can define a scaling transformation as follows:

$$\begin{aligned} \mathit{scale}(s_x, s_y) = \mathit{transformation}(&\lambda\langle x, y\rangle \rightarrow \langle s_x \times x, s_y \times y\rangle, \\ &\lambda\langle x, y\rangle \rightarrow \langle x/s_x, y/s_y\rangle) \end{aligned}$$

[5] When $x = 0$, $|y|/x$ will be ∞ or not-a-number, which will cause *lerp* to return black.

We can use this transformation to scale our previous examples. For example, to make our filled triangle half as big, we can do the following:

$$\mathit{transform}(\mathit{scale}(1/2, 1/2), \mathit{fill}(\mathit{triangle}, \mathit{tritex}))$$

Where the *transform* function is expressed as follows:

$$\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{path}(p)) = \mathit{path}(f \circ p)$$
$$\mathit{transform}(f, \mathit{shape}(l)) = \mathit{shape}([\mathit{transform}(f, p) \mid p \leftarrow l])$$
$$\mathit{transform}(\mathit{transformation}(f, f^{-1}), \mathit{texture}(t)) = \mathit{texture}(t \circ f^{-1})$$
$$\mathit{transform}(f, \mathit{fill}(s, t)) = \mathit{fill}(\mathit{transform}(f, s), \mathit{transform}(f, t))$$

The only constraint on a transformation is that it must be continuous, otherwise it would be possible to transform a closed path (defining an area) into an open path (not defining an area). As an example of a non-affine transformation consider the "wave" transformation shown in Figure 3(a):

$$\mathit{wave} = \mathit{transform}(\lambda\langle x, y\rangle \rightarrow \langle x + \sin(y), y\rangle), \lambda\langle x, y\rangle \rightarrow \langle x - \sin(y), y\rangle)$$

These transformations can be composed using the following *compose* function, which uses the well-known rule $(f \circ g)^{-1} = g^{-1} \circ f^{-1}$.

$$\mathit{compose}(\mathit{transform}(f, f^{-1}), \mathit{transform}(g, g^{-1})) = \mathit{transform}(f \circ g, g^{-1} \circ f^{-1})$$

A benefit of having both directions of a transformation is that we can also transform *transformations*. For example, if we have a rotation transformation and we want to change the center of rotation, we can achieve this by transforming the rotation by a translation. This is done by first applying the inverse translation, then the rotation and then the forward translation. In general, we can transform any transformation by another transformation as follows:

$$\mathit{transform}(t, r) = \mathit{compose}(t, \mathit{compose}(r, \mathit{inverse}(t)))$$
$$\textbf{where } \mathit{inverse}(\mathit{transform}(f, f^{-1})) = \mathit{transform}(f^{-1}, f)$$

As an example, we can transform our wave transformation to produce smaller waves:

$$\mathit{scaledWave} = \mathit{transform}(\mathit{scale}(1/30, 1/30), \mathit{wave})$$

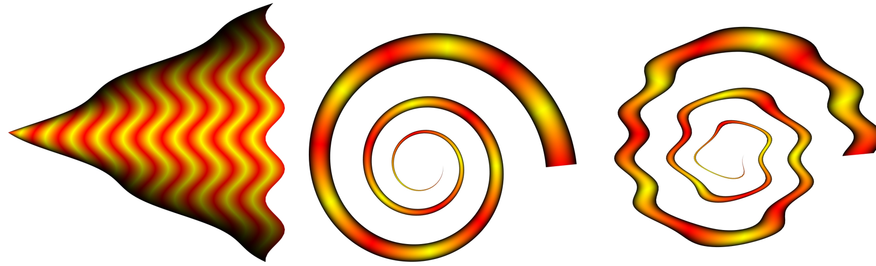
Applying this transformation to our filled triangle produces Figure 3(a).

Another example of a non-affine transformation is a "sweep": mapping the [0,1] interval on the x-axis to a given path. For example, by first scaling our filled triangle to make it thinner we can obtain Figure 3(b) as follows:

$$\mathit{fspir} = \mathit{transform}(\mathit{compose}(\mathit{sweep}(\mathit{spiral}), \mathit{scale}(1, 1/40)), \mathit{ftriangle})$$

Other papers [10, 9] have shown how to implement the sweep transformation when only the forward transformation is required, we now show how to handle both directions of this transformation. To define this transformation in a resolution-independent way, we define it as a function which takes the concrete geometry of the path and returns a transformation. Using the *analyze* constructor, we make this function into a transformation.

To prevent changes in speed along the path, we want the norm of the derivative to be constant along the path. To this end, we reparameterize the concrete



(a) Wave transformed triangle. (b) Triangle swept along spiral. (c) The filled triangle swept by a spiral transformed by a wave.

Fig. 3. Non-affine transformation examples

geometry of the path to a new geometrical description, q, with the same shape and a constant norm of the derivative, using an algorithm such as [11]. The forward transformation can then be expressed as follows:

$$\lambda\langle x, y\rangle \rightarrow q(x) + y \times \widehat{q'(x)}$$

Here $\hat{x}$ denotes a normalized vector and q' is the derivative of q.

The inverse transformation works by finding the closest point on the path to the point that is to be transformed. The horizontal coordinate is then the parameter at that point on the path, and the vertical coordinate is the distance of the point to be transformed from the path. More precisely:

$$\lambda v \rightarrow \langle t, sgn(q'(t)) \times |q(t) - v|\rangle \ \textbf{where}\ t = f(v)$$

Here sgn is the sign function and f computes the parameter of the closest point on q to a given point, using an algorithm such as [12]. As a final example of the compositionality this framework gives us, we transform the swept triangle using our wave transformation to obtain Figure 3(c):

$$transform(scaledWave, fspir)$$

4 Implementation and Optimization

Our approach can be efficiently implemented by mapping it to a graphics host. We first describe a basic implementation and then introduce some extensions to allow more optimizations. Finally, we show how we can support clipping and discuss potential further optimization. The implementation of Deform as sketched in this section is surprisingly concise and simple and consists of just 983 lines of Scala code.

4.1 Basic implementation

The main function to implement is the *render* function, which acts as an interpreter for the constructors that may occur in a *TexturedShape*. The pipeline of the *render* function is shown in Figure 4 and is organized as follows; A *TexturedShape* is produced by the user program and its shape is then translated into geometry, i.e., lines and Bézier curves, which are in turn translated to their equivalent representations in the graphics host. The graphics host then fills the shape, producing a raster telling us which pixels are inside the shape. We then simply iterate over these pixels and call the corresponding texture function for each pixel, producing a color raster which is then sent to the display.

The *toBézier* function in this pipeline is also used to interpret *analyze* constructors, namely to generate the concrete geometry which is fed to the function argument of the constructor. We currently use a simple implementation of this function: we sample the function until the samples are so close to each other that the error is smaller that the size of a pixel. Afterwards, the samples are joined by lines.

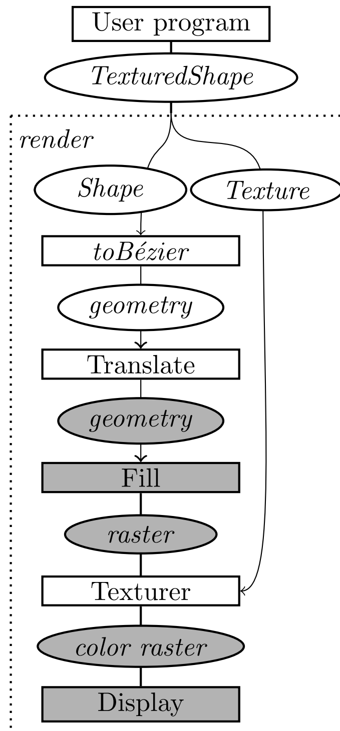


Fig. 4. Rendering pipeline. Gray indicates functionality from the graphics host.

4.2 Special cases

We optimize the basic implementation by intercepting special cases and mapping them to the corresponding functionality of the graphics host. We add a new constructor for each special case, which are shown in Table 3. Several of these new constructors were presented earlier as functions and by transforming them into constructors the *render* function can recognize them and act accordingly. We now discuss the special cases for shapes, textures and transformations.

Shapes The first special case for shapes concerns paths that consist of lines and Bézier curves. It is of course wasteful to use a combination of lines and Bézier curves, only to later approximate it with other lines and Bézier curves. Hence, we extend our *Path* type with extra constructors for these types of paths and a constructor for *join*, so that the *toBézier* function can immediately use these descriptions without sampling.

The second special case for shapes deals with constructive solid geometry operations. The default implementation of these operations is to obtain a concrete geometry of the shapes using *toBézier* and then analyze intersections to produce the new shape. In the case of union or symmetric difference we can skip this analysis. The union of a set of shapes can be implemented by supplying the set of shapes to the fill function of the graphics host and using the *non-zero fill rule*. This tells the renderer to fill any pixel that is inside at least one of the shapes, effectively rendering the union of the shapes. Analogously, we can render the symmetric difference of a list of shapes by using the *even-odd fill rule*, which states that a pixel should be filled if it is inside an odd number of shapes.

Textures If the graphics host has support for a texture, we would like to make use of these optimized capabilities, because then we can completely skip the Texturer step in the pipeline. Hence, we include the constructor *nativeTexture* for these cases, which takes a function that given an affine transformation gives the specific representation for the graphics host of the transformed texture and a regular texture function for use when the transformation of the texture is not affine.

Transformations If a transformation is affine and the path consists of lines and Bézier curves, we transform the geometry directly, instead of by sampling a function. The constructor *affineTransformation* represents such an affine transformation by two matrices (the specification of this type is left open), one for the forward transformation and one for the inverse transformation. We also change the *transform* function into a constructor so that the *toBezier* function can intercept this special case. The *compose* function is also adapted to intercept the special case of composing an affine transformation with another affine transformation, which can be done using matrix multiplication instead of function composition, saving computations when points are transformed.

Constructor	*Type*
line	$\mathbb{R}^2 \times \mathbb{R}^2 \to Path$
quadBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to Path$
cubicBezier	$\mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \times \mathbb{R}^2 \to Path$
join	$[Path] \to Path$
union	$[Shape] \to Shape$
symdiff	$[Shape] \to Shape$
nativeTexture	$(Matrix \to NativeTextureDesc) \times (\mathbb{R}^2 \to Color) \to Texture$
transformation	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \to Transformation$
transform	$Transformation \times A \to A$
	where $A \in \{Path, Shape, TexturedShape, Transformation\}$
affineTransformation	$Matrix \times Matrix \to Transformation$
pathbb	$(\mathbb{R} \to \mathbb{R}^2) \times BBox \to Path$
transformationbb	$(\mathbb{R}^2 \to \mathbb{R}^2) \times (\mathbb{R}^2 \to \mathbb{R}^2) \times (BBox \to BBox) \to Transformation$

Table 3. Additional constructors for special cases.

Performance Note that in traditional frameworks such as Java2D or Processing, the special cases presented above are the *only* things that are expressible. Thus, the interception of these special cases guarantees that drawings that could also be produced using such a library are approximately as fast. We verified this by generating equivalent Java2D and Deform code in which 100,000 shapes (letters) were rendered, each with their own native texture and affine transformation. The Deform code performed 0.8% slower than the direct Java2D calls. This minor difference in speed is due to the fact that the Deform code first builds an intermediate representation of the textured shapes.

4.3 Clipping

For large scenes, involving many shapes, a valuable optimization is *clipping*: determining the bounding boxes of shapes and then ignoring the shapes that are not in view. However, since in our framework shapes and transformations can be arbitrary functions, it is impossible to discover the bounding box of a shape without sampling it.

For this reason we add two new constructors: one to declare a path and its bounding box (the specification of this type is left open) and one to declare a transformation and also a function to forwardly transform a bounding box. In this way the user can optionally give the bounding boxes of transformed shapes. If the

bounding boxes are not supplied, the shapes will simply not profit from clipping. In Deform, all library functions to construct paths and transformations also deal with bounding boxes. For example, lines and Bézier curves get the bounding box induced by their (control) points and *join* produces the smallest bounding box that contains the bounding boxes of its arguments. Affine transformations transform a bounding box by transforming each of its vertices. We currently use axis-aligned bounding boxes, but it is also possible to use non-axis-aligned bounding boxes that fit the shapes more tightly, at the cost of more computations.

4.4 Potential optimization

A potential optimization might be to speed the *toBézier* function by using techniques from the field of curve fitting. We could do the sampling and fitting in parallel, by modifying a curve fitting algorithm such as [13]. We can then stop the sampling earlier if the samples we take lie close enough to the current approximation. We can also use the parameter of each point to improve the speed of our approximation since this is often useful information for curve fitting algorithms [13]. Finally, curve fitting algorithms often estimate a derivative of the shape, so if we numerically compute the derivative, or supply it using an automated differentiation system [14], we can also use this information to more quickly find an approximation of the curve.

5 Case study: Focus+context Lenses

As a real world example of how this framework enables advanced, resolution-independent computer graphics techniques in a compositional way, we show how to implement the form of focus+context lenses that are presented in [15], which have been shown to be useful in human computer interaction [16]. A focus+context lens, such as the one in Figure 1, is a transformation that magnifies a part of the space (the focus area) and shows how this magnified part fits into the rest of the space (the context) through a deformation. We compare our implementation to the previous implementation of this form of focus+context lenses [16]. Our implementation is slightly harder, since we require both directions of the transformation. As we will show, this effort is well spent since it yields a faster implementation that gives better image quality at a fraction of the code.

5.1 Implementation

We first consider the *inverse* transformation as presented in [15, 16]. Figure 5 shows the elements of a lens: r_f is the radius of the focus area, r_l is the radius of the lens and we define m as the magnification factor. The inverse transformation is then defined as follows:

$$l^{-1}(v) = \begin{cases} v/m & |v| < r_f \\ \frac{v}{|v|} \times n^{-1}(|v|) & r_f < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

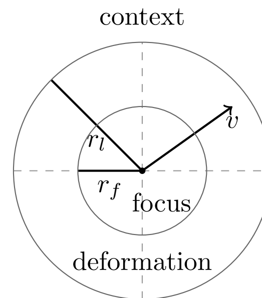


Fig. 5. Lens elements.

Where n^{-1} is the function that describes the deformation, by giving the new norm, i.e., distance from the

center of the lens, for the point to be transformed and is a continuous, monotonically increasing function from $[r_f, r_l]$ to $[r_f/m, r_l]$:

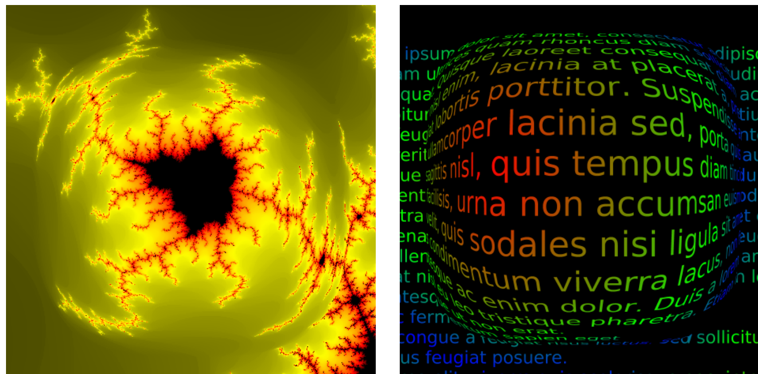
$$n^{-1}(d) = \frac{d}{(1 - p(z)) \times (m-1) + 1} \quad \textbf{where } z = (d - r_f)/(r_l - r_f)$$

Here z describes how far into the deformation area the point is, with zero if the point is on the border of the magnification area and one if it is on the border of the context area. The *profile* function, p, describes the shape of the deformation and can be chosen freely as long as it is a continuous, monotonically increasing function from $[0, 1]$ to $[0, 1]$, such as the identity function. Another variation point is which norm to use to compute $|v|$, which decides the shape of the lens. In general it is possible to use any L^P norm, which are of the form $\sqrt[p]{x^p + y^p}$. The lens is circular with L^2 and with L^∞ the norm resolves to $\max(x, y)$ and the lens is square. The example in Figure 1 uses the Euclidian norm and a Gaussian profile function and Figure 6 shows two more Deform screenshots of other lenses in action.

We now need to derive the *forward* transformation from this inverse transformation. If we have the inverse of the function n^{-1}, then the forward transformation can be expressed as follows:

$$l(v) = \begin{cases} v \times m & |v| < r_i/m \\ \frac{v}{|v|} \times n(|v|) & r_f/m < |v| < r_l \\ v & \text{otherwise} \end{cases}$$

However, for many profile functions, there is no analytic solution for the inverse of n^{-1}. Luckily, n^{-1} is a continuous monotonically increasing function, so we can implement $n(t)$ by numerically searching for the x such that $n^{-1}(x) = t$. We use Newton's method for this, since it is very efficient at finding the roots of monotonic functions. This method requires the derivative of n^{-1}, which can be constructed using the derivative of the profile. In this way only the profile function and its derivative are needed when creating a lens with a different profile.



(a) L^3 norm, linear profile (b) L^4 norm, quadratic profile

Fig. 6. Different types of lenses in action

5.2 Comparison

The previous implementation [16] of this form of focus+context lenses is in the Zoomable Visual Transformation Machine (ZVTM) [17] framework for zoomable user interfaces. The advantage of their approach to implementing these lenses is that it is very loosely coupled with the graphics host, and is thus applicable in many graphical frameworks. In our approach these lenses can be added easily and this yields a better implementation in terms of length of code, speed and image quality.

Code size In the ZVTM implementation, defining the lenses requires about 700 lines of code, and each new lens (with a different norm or profile) requires about 100 lines of code [16]. In our declarative framework, the implementation of these lenses requires 43 lines of code, including the definition of the (reusable) numeric approximation code, while defining a new lens can be done in a single line of code. For example, a rounded square lens with a quadratic profile (with derivative $2 \times x$), as shown in Figure 6(b), is declared as follows:

$$lens(\lambda\langle x, y\rangle \rightarrow \sqrt[4]{x^4 + y^4}, \lambda x \rightarrow x^2, \lambda x \rightarrow 2 \times x)$$

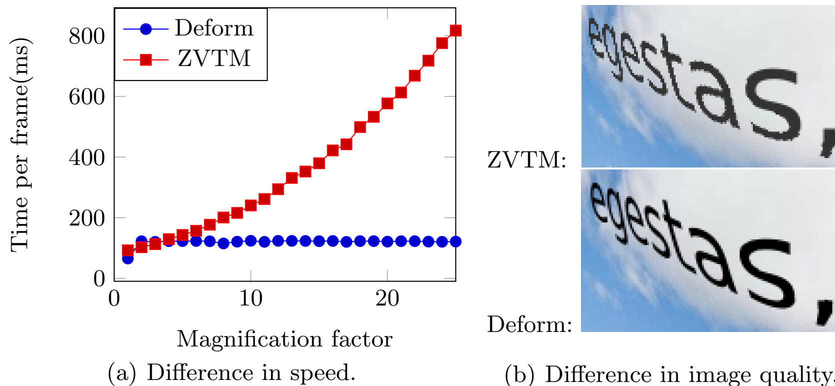
Performance As a performance comparison, we implemented the setup shown in Figure 1 in both Deform and ZVTM and measured the time it took to render a single image at different magnification factors. This was chosen because it is a simple example of a combination of shapes (text) and a texture (bitmap image). The entire picture was 1600x1000 pixels big and the lens had a focus radius of 100 pixels and a lens radius of 200 pixels. Note that both ZVTM and Deform run on the JVM and are built on top of Java2D. Figure 7(a) shows the results of our measurements on an Intel i7 2.8GHz CPU running OpenJDK 1.11.3. All measurements are the average of 100 runs.

We can see that in ZVTM the magnification factor has a huge impact on performance, whereas in Deform it has no effect at all. This is because ZVTM does not feature non-affine transformations in general and uses a trick to achieve focus+context lenses; It renders the lens area *twice*: once without magnification and once with magnification. Afterwards, both renderings are sampled to produce the lens area. The second, magnified rendering uses a buffer of width and height $2 \times m \times r_l$. Hence the amount of pixels in this buffer is $(2 \times m \times r_l)^2$, which explains the quadratic growth of the ZVTM rendering time.

Image quality As a final comparison, we consider the image quality of both approaches as shown in Figure 7(b). This notable difference in image quality is caused by the fact that Deform performs the discretization of shapes and textures later. ZVTM performs the discretization *before* applying the lens, while Deform performs the discretization *after* applying the lens. Hence Deform does not suffer from aliasing artifacts.

6 Discussion

While our framework is very expressive, it currently does not support post-processing image filters such as blurs. These filters are computationally very expensive and require low-level optimizations for real-time performance. Halide [18] is an example of a language that is specifically designed for such filters; the



(a) Difference in speed.   (b) Difference in image quality.

Fig. 7. Performance and image quality comparison.

programmer gives a concise declarative description of the filter along with a schedule that states how the filter must be implemented. This yields very good results, outperforming hand tuned assembly code in some cases. It would be interesting to explore how the Halide way of describing filters can be fitted into our framework.

Another open question is how we can exploit the massive power that is available via GPUs: which paths, transformations and textures can be executed on the GPU and how? How can these parts work together with functionality that cannot be executed on the GPU? Answering these questions will lead to a truly high-performance implementation of Deform.

7 Conclusion

We have presented a novel declarative framework for resolution-independent 2D graphics that is simple, expressive and composable while still being applicable to real-life graphics. We have shown how to implement this framework such that it easily maps to readily available, highly-optimized procedural graphics libraries and have also shown how this framework can support clipping, so that it is possible to render very large scenes. We have shown a simple benchmark that shows that our framework is as fast as directly using the graphics host, thanks to the interception of special cases. As a real-world example, we have implemented focus+context lenses. The result is faster and smaller than the state-of-the-art implementation and has better image quality. Our framework liberates the programmer from the limitations of traditional frameworks and we expect that it forms an excellent foundation for creating resolution-independent graphics and higher-level visualization tools in a wide range of domains.

Acknowledgements We thank Robert van Liere and Tijs van der Storm for their helpful comments.

References

1. Klint, P., Lisser, B., van der Ploeg, A.: Towards a one-stop-shop for analysis, transformation and visualization of software. In: Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011), Springer (2011) 1–18

2. Karczmarczuk, J.: Functional approach to texture generation. In: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages. PADL '02, London, UK, UK, Springer-Verlag (2002) 225–242
3. Hoffmann, C.M.: Implicit curves and surfaces in CAGD. IEEE Computer Graphics and Applications **13** (1993) 79–88
4. Sederberg, T.W., Anderson, D.C., Goldman, R.N.: Implicit representation of parametric curves and surfaces. Computer Vision, Graphics, and Image Processing **28**(1) (1984) 72–84
5. Dokken, T., Thomassen, J.: Overview of approximate implicitization. Topics in Algebraic Geometry and Geometric modelling, AMS series on Contemporary Mathematics CONM 334 **28**(1) (2003) 169–184
6. Elliott, C.: Functional image synthesis. In: Proceedings of Bridges. (2001)
7. Finne, S., Jones, S.P.: Pictures: A simple structured graphics model. In: In Glasgow Functional Programming Workshop, Ullapool. (1995)
8. Matlage, K., Gill, A.: ChalkBoard: Mapping functions to polygons. In: Proceedings of the Symposium on Implementation and Application of Functional Languages. (Sep 2009)
9. Elliott, C.: Programming graphics processors functionally. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. Haskell '04, New York, NY, USA, ACM (2004) 45–56
10. Karczmarczuk, J.: Geometric modelling in functional style. In: Proceedings of the III Latino-American Workshop on Functional Programming, CLAPF'99. (1999) 8–9
11. Casciola, G., Morigi, S.: Reparametrization of NURBS curves. International Journal of Shape Modeling **2** (1996) 103–116
12. Ma, Y.L., Hewitt, W.T.: Point inversion and projection for NURBS curves and surfaces: control polygon approach. Comput. Aided Geom. Des. **20**(2) (May 2003) 79–99
13. Schneider, P.J.: An algorithm for automatically fitting digitized curves. In Glassner, A.S., ed.: Graphics gems. Academic Press Professional, Inc., San Diego, CA, USA (1990) 612–626
14. Elliott, C.: Beautiful differentiation. In: International Conference on Functional Programming (ICFP). (2009)
15. Carpendale, M.S.T., Montagnese, C.: A framework for unifying presentation space. In: Proceedings of the 14th annual ACM Symposium on User interface software and technology. UIST '01, New York, NY, USA, ACM (2001) 61–70
16. Pietriga, E., Bau, O., Appert, C.: Representation-independent in-place magnification with Sigma lenses. IEEE Transactions on Visualization and Computer Graphics **16** (2010) 455–467
17. Pietriga, E.: A Toolkit for Addressing HCI Issues in Visual Language Environments. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2005) 145–152
18. Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., Durand, F.: Decoupling algorithms from schedules for easy optimization of image processing pipelines. ACM Transactions on Graphics **31**(4) (2012) 32