Functional Approach to Texture Generation

Jerzy Karczmarczuk

University of Caen, France, karczma@info.unicaen.fr

Abstract. We show the applicability of pure functional programming for the construction of modules which create procedural textures for image synthesis. We focus our attention to the construction of *generic* combinators and transformers of textures, which permit to write texture generators of substantial complexity in a very compact and intuitive manner. We present a concrete package implemented in Clean.
Keywords: images, combinators, noise, tesselations, Clean.

1 Introduction

1.1 What Are Textures

Generation of graphical objects by programs is the essence of image synthesis. Since those objects correspond often to data structures manipulated using regular, often generic procedures such as rotations, interpolations, unions, hierarchic embedding, etc., computer graphics is a wonderful training field for functional programming, with its higher-order functions, ubiquitous recursion, and declarative style. The literature is very rich. Already Henderson [1] shows how to *compose functionally* graphic objects, and more recent works show the applicability of Scheme, CAML [2] and Haskell [3] to picture generation. In [4] we described functional methods to model parametric surfaces in 3D. But that papers focus on modelling and *drawing* of graphical objects, i.e., choosing the region where a given figure: a polygon, a zone filled with a colour gradient, etc. will be placed. These programs operate usually upon discrete data structures.

But the decorations of surfaces need often the implementation of another process: the *texturing*, quite different from drawing techniques. There is no destination place the texturing module can *choose*, it gets the coordinates of the *current point* from the rendering engine, e.g., a ray tracer or a scan-line projector, and decides which colour assign to this point, after a possible transformation from the scene/model to the intrinsic texture space (inverse texture mapping), and after analyzing the lighting conditions. Thus, for the texturer the size of its working area is irrelevant. It occupies the whole of the coordinate space, possibly infinite. If we forget the lights, and the coordinate mapping (e.g. the projections), we may say that *textures are functions*: ***Point*** $\rightarrow$ ***Colour***, where ***Point*** is a 2-dimensional point of the texture space. They are "continuous objects", and their assembly from some primitives, transformations and compositions follow different rules than drawing of polygons, etc. Textures may be bitmap images, but the creation of patterns which simulate natural surfaces, geometrically regular or random decorations, is better done algorithmically. The *procedural texturing* has a long history, the reader will find all the information in the book [5], or in other documents available through the Web, e.g., [6, 7]. The importance of *shaders* — user programmable modules which supply textures for ray tracers or other rendering engines increases every year, more and more rendering packages include programmable shaders. The history of functional methods in the realm of procedural texturing is also rather long. Karl Sims [8] used Lisp, and automatically generated compositions and transformations of symbolic expressions to generate exquisite patterns. One of our primary sources of inspiration was the package PAN of Conal Elliott [9], based on Haskell. (See also Pancito of Andrew Cooke [10].)

But despite the fact that the construction of textures is static, dominated by structuring of data, and some quasi-analytic operations, such as the computation of gradients, or filtering, and the concept of modifiable state doesn't need to play any significant role (mainly some dull reassignments of variables, and accumulating loops), the only widely spread procedural texturing language is imperative (very close to a truncated "C"): the RenderMan shading language, see the book [11], the documentation of the package BMRT [12], or the Web site of Pixar [13]. Other approaches to procedural shading, notably the work of Pat Hanrahan [14] also use "imperative" languages, similar in style to RenderMan (although Hanrahan began with a Lisp-style approach, and moreover his language is designed to code *very* fast programs, exploiting directly the hardware pipelines, so his shaders are rather declarative,dataflow style stream processors than imperative procedures. . .). Reading the shaders' codes give a strange impression: *the essence of shaders is declarative*, the coding is "C" like. . . It can and it should be coded in a more proper style, if only for pedagogical purposes.

1.2 Objectives and Structure of This Work

We show here the naturalness and the ease of texture construction using Clean [15]. We constructed a library called ***Clastic***, useful for experimentation in texturing. Higher-order functions and overloading of arithmetic operations to functional objects make it possible to design a decent set of combinators and transformers in an very compact way. We may code simply and universally not only the texture generators, but also the typical bitmap image manipulations available in known image processing packages, and in interactive texture designers, e.g., SynTex [16].

The aim of this work is mainly methodological and pedagogical. Clastic has been used to teach image synthesis, and it is a fragment of a bigger pedagogical project — the construction of a purely functional ray tracer (another nice field for declarative programming) equipped with dynamic shading modules. The texturing library is available separately. It has not been optimized (yet) for speed. Our main leitmotif was to identify useful *high-level abstractions*, and to show how to construct complex textures in a *regular* way, using standard functional composition techniques, and exploiting the geometrical invariance of graphical objects as far as possible.

The structure of the paper is the following. After the introduction we describe the geometric framework of the package, the primary building blocks, and simple texture combinators and transformations. We show how to construct random patterns (various "noise" textures), and also how to generate tesselations possessing non-trivial symmetries. The examples of programmed textures have been chosen to show the specificity of abstract functional coding, not always optimal; some functions within Clastic are coded in a more efficient way.

We will abuse the language, and call *textures* the final rendered bitmap images as well. For definitness and for testing we use simple Cartesian coordinates, a Clean data structure `V2 x y` represents a 2D point with real x and y, and a record `{tr,tg,tb}` belonging to the type `RColour` with real components in $[0-1]$ is a RGB colour[1]. We underline the fact that both types belong to a class of *vectors*, with all standard operations thereupon, e.g., an overloaded operator `*>` multiplies a point or a colour by a scalar, we can subtract them, etc. This is needed for generic interpolation procedures.

The functional layer of Clastic is platform-independent, but the interface used for the generation of examples works under the Windows Clean IO system. It permits the creation of some rendering windows in their "virtual" Cartesian coordinate systems, and launches iterators which fill the Windows *Device-Independent Bitmaps*, unboxed arrays of pixels, whose display is ensured by the Clean IO system. The interface permits also to read BMP files, to convert them into texturing functions, and to save the rendered images. We show here only the definitions of texturing functions, they are the only entities which have to be programmed by the user. The examples are simple, since our main presentation objective is to show *how* to make them from basic blocks, in a didactic environment. In order to follow the examples the reader should be able to read code written in a typical modern functional language. Some Clean particularities should be explained, especially for readers acquainted with Haskell. The sharp (`#`) symbol introduces a local definition (like `let`). The functional composition operator is `o`, and the unary minus sign is denoted by `~`. Bars: "`|`" introduce alternative clauses, as in Haskell, but the default clause "`| otherwise = ...`" can be shortened to "`=`".

2 Primary textures

Almost any reasonable *real* function on points can be used as the heart of a texture generator. Such simple function: `radGrad p = norm p *> RWhite` constructs a radial, linear gradient of intensity, and `angCol p = hsvtorgb (angle p) 1.0 1.0` — an "azimuthal rainbow" shown on Fig. (1:A). Clastic defines several useful vector and colour functions, such as `norm` computing $\sqrt{x^2+y^2}$, `angle` yielding $\mathrm{atan2}(y,x)$, or `hsvtorgb` converting the set Hue-Saturation-Value into a `RColour` record. Figure (1:B) visualizes the function

```
tacks = cmix RGrey RBlue (sscaled 0.1 tck)
```

[1] The Clean library defines the type `Colour` with components in [0–255]

```
 where
  tck (V2 x y) = floor(sin(x+sin(y+sin x))) - floor(sin(y+sin(x+sin y)))
```

where `cmix` is a linear interpolator of colours (with its third argument usually in $[0-1]$), and `sscaled` is a uniform scaling transformer of a texture function. We note immediately that although such functions can be used to generate interesting tilings, texture *design* needs simpler, regular methods, not requiring an impossible mathematical insight. It is not obvious that the texture `polyg k` (Boolean, needing some colour mapping for its visualization) given by

```
polyg k (V2 x y)
 # kd = Dpi/fromInt k                  //Dpi = 2π
 # kdj= kd * floor (atan2 y x / kd)
 # skdj=sin kdj; ckdj=cos kdj
 = (x-ckdj)*(sin(kdj+kd)-skdj)-(cos(kdj+kd)-ckdj)*(y-skdj)<=0.0
```

represents a ***regular polygon with*** k ***vertices***. The interested reader will find more textures of this kind in [17], and in the gallery of PAN images. But we see also that the symmetry and the periodicity of the texture generator makes definitions very compact, and this will be the main leitmotif of this section: if possible, avoid decisional structures (e.g., conditionals) in composite textures; replace the relation "belongs to W" by the characteristic function of W; exploit directly the primitive function symmetries. In such a way the texture becomes more an ordinary function than a piece of program, and may be more easily designed and composed. Such primitives as the polygon, a regular star and many others are built into Clastic.

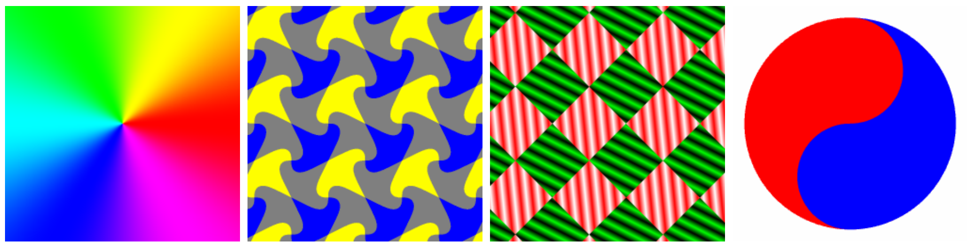


Fig. 1. Simple geometric patterns

The most typical functional texture is a binary filtering relation, say, $H(x,y) > 0$, which separates the space in two zones, the interior and the exterior of an object. The Boolean value of the relation may be transformed into one of two colours, or used as a mask for other textures, as shown on Fig. (1:C). This checkerboard mask is just $\sin(x) - \cos(y) > 0$.

2.1 Basic building blocks, overloading, and combinators

The basic abstraction which replaces the relation $r > 0$ is the function θ of Heaviside, which may be defined as `step x = if (x<0.0) 0.0 1.0`. In fact, Clastic uses *very heavily* the overloading, and our `step` is applicable to other domains as well. The real definition is:

```
step :: !a -> a | <, zero, half, one a
step x | x<zero = zero
       | x>zero = one
       = half
```

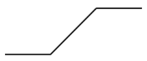
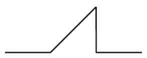
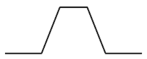
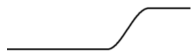
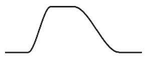
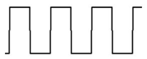
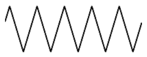
i.e., it is a strict function on a type which may be ordered, and contains the overloaded constants `zero`, `one`, and `half`, whose meaning is evident. This is an important point, several texture generators in Clastic are written in a way enabling the application of the Automatic Differentiation machinery, which permits to compute gradients of functions, precisely and easily, but those functions must be defined on entities which generalize real numbers. The details cannot be discussed here. Moreover, we have overloaded the arithmetic operations on functional objects, which permits to define several functions in a combinatoric style, without explicit parameters, e.g., `f = one + sin`. For simplicity, where this overloading is not explicitly needed, we shall write 0, 1, etc.

Thus, the checkerboard (real valued) mask above may be defined as `chkb (V2 x y) = step (sin x - cos y)`. Masking out the (real) texture `tx` outside the unit disk can be written "normally" as `f p = step (1-norm p) * tx p`, or as `f = step o (one-norm) * tx`. From `step` we may construct by subtraction the function `pulse`, equal to 1 between 0 and 1, and zero outside.

```
pulse  = (id - translated one) step   //where
translated a f x = f (x-a)
```

and use it to construct bands by applying it to `x` and ignoring `y`. Product of two orthogonal, scaled pulses creates a rectangle. Such elementary blocks are frequent in RenderMan shaders which make geometric patterns, and they belong to the standard panoply of builders of signal processing algorithms.

We need also several filtered (smoothed) primitives, e.g. the `ramp` function, going linearly from 0 to 1, often used for clamping the colour components, or `smoothstep` which interpolates between 0 and 1 using the Hermite cubic $h(x) = 3x^2 - 2x^3$, and ensures smooth, differentiable transitions between distinct areas. We complete the collection by various *replicators*, which are simply periodic functions permitting to reduce the coordinates of a point to the periodicity interval of the replicator. Most common is the function `frac` which returns the fractional part of a number, reducing it to $[0 - 1]$. Below we see some plots of primitive blocks, and periodicity generators. Such functions are usually constructed *ad hoc*. A variant of `smoothpulse` was used in our woven patterns, see Fig. (14:A, B), and `symteeth` generated the stripe patterns on Fig. (1:C).

`ramp = max zero o min one`	
`sawtooth = pulse*id`	
`filteredpulse`	
`smoothstep = pulse*hermite +translated one step`	
`smoothpulse`	
`trainpulse`	
`symteeth = symtrain abs`	

2.2 Combination of Shapes

Inequalities $H(\boldsymbol{p}) > 0$ specifiying "solid" objects: polygons, disks, etc., make up their *implicit representation*, well known in the domain of 3D synthesis also, see [18]. The construction of unions or intersections of such objects is easy, with the usage of masking or arithmetic combinations, e.g., if two zones are given by $H_1(\boldsymbol{p}) > 0$ and $H_2(\boldsymbol{p}) > 0$, the function $\max(H_1(\boldsymbol{p}), H_2(\boldsymbol{p}))$ yields their union, and min — the intersection. The negation is the complement. In the space of characteristic functions, the intersection of A and B is their product, and the union is given by $A + B - A \cdot B$. In order to cover one texture by another, we have the masking operator, which has been used to create the Fig. (1:C):

```
fmask h f g = \x -> if (h x <> 0) (f x) (g x)
```

We leave to the reader the construction of Fig. (1:D) by a combination of a unit disk, two smaller and translated disks, and a half-plane (`step x`). In fact, one may create only yin, and obtain yang by a half turn.

Boolean operations (2D Constructive Solid Geometry) and masking, are not the only combinations possible, we end this section by showing another, non-trivial combination: the halftoning, where one texture modulates another. If we take the interior of the disk $x^2 + y^2 < 25$, and put inside a variable grey level: $g = 0.8 - 1/14\sqrt{(x+3.5)^2 + (y-2.8)^2}$, we obtain a simulation of a mat sphere. Very often such colour gradients are used to augment some 2D graphics by cheap 3D effects. But g may be used as a specific mask of a periodic function: $h = 1 - \theta(\sin(\omega x) - \cos(\omega y) + 2.0 - 4.0g)$ with a sufficiently high frequency ω. Higher g means greater statistical chance to obtain $h = 1$. The result is displayed on Fig. (2:A). The Figure (2:B) is another variant of this technique, but here the modulating density is reconstructed from a bitmap image transformed by Clastic into a texture, and the modulated pattern is a "white noise", a high-frequency random function (so, it is rather dithering than halftoning).

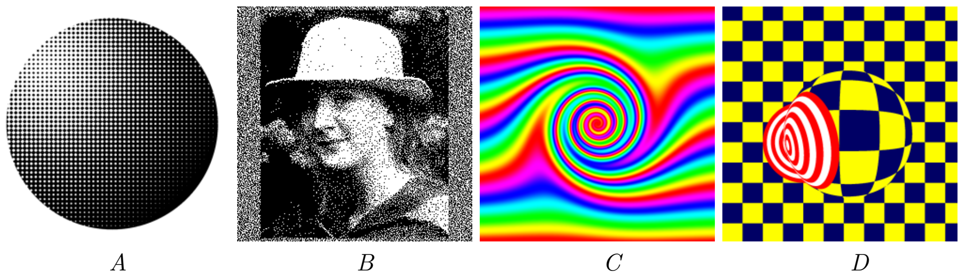


Fig. 2. Texture combinations and deformations

3 Transformations and deformations

Since textures are functions η :: ***Point*** $\rightarrow$ ***Something***, where ***Something*** is usually a geometric scalar (a real, a colour, etc.) they may be "post"-transformed by acting on their co-domain space: $\tilde{\eta}(\boldsymbol{p}) = \mathbb{T}\big(\boldsymbol{p}, \eta(\boldsymbol{p})\big)$. We may change the colours, multiply one texture by another, etc. But the textures may also undergo a global, or a local geometric transformation in the domain space, and this is the essence of all deformations.

Textures, as other implicit graphical objects transform *contravariantly.* Suppose that ζ is a texture (function), and that we have a transformation $\mathbb{R}$ (a rotation, a translation, a local warp, etc.) acting on points, moving them. We want to find the *representation* $\mathbf{T}_R$ of $\mathbb{R}$ acting on ζ. The fundamental property of a representation is that it constitutes a group homomorphic to the original: $\mathbb{R}_1\mathbb{R}_2$ generates $\mathbf{T}_{R_1R_2} = \mathbf{T}_{R_1}\mathbf{T}_{R_2}$, and $\mathbb{R}^{-1} \rightarrow \mathbf{T}_{R^{-1}} = (\mathbf{T}_R)^{-1}$. It is easy to prove that the *definition* of $\zeta' = \mathbf{T}_R\zeta$ as $\zeta'(p) = \zeta\left(\mathbb{R}^{-1}p\right)$, gives a correct representation. For simple affine transformations Clastic define the texture transformers using directly the inverse operations:

```
translated a f = \p -> f (p-a)
scaled c f = \p -> f (p/c)

rotated a = rotated_cs (cos a) (sin a)   //where
rotated_cs co si f = \(V2 x y) -> f (V2 (co*x+si*y) (co*y-si*x))
```

(Note the signs in the last line.) The scaling may be non-uniform; we have overloaded the multiplication and the division for vectors (element-wise), which is useful, even if mathematically a bit bizarre...

Clastic implements many other operations: symmetries, transpositions, replications (coordinate reduction), etc. But in general case if the user wants a specific, complicated, especially *local* (point-dependent) transformation, he must know how to construct its inverse, and this may be very difficult. At any rate this *must* be provided. Then, the package uses a simple generic combinator-deformer `transf` which acts on textures in the following way:

```
transf trf txtr = txtr o trf
```

(or: `transf = flip (o)` for combinatoric maniacs…) with `trf` being the inverse transformation. The properties of deformers are not always obvious, e.g., it is easy to forget the contravariance which implies that `(transf f1 o transf f2) tx` yields the texture `tx o f2 o f1`.

If this function computes only the *displacement* ("warping") of the current point, the appropriate deformer is `displace trfun txtr = txtr o (id+trfun)`. Random displacements, especially turbulent ones are particularly useful to generate irregular wood grain, distorted marble veins, etc.

The notorious eddy shown on Fig. (2:C)is the result of applying a deformer

```
eddy ampl damp p
  # (r,phi) = topolar p
  # nphi = phi + ampl*exp(~damp*r*r)
  = V2 (r*cos nphi) (r*sin nphi)
```

and the "lense" effect on Fig. (2:D) is an exercise in optics as shown on Fig. 3.

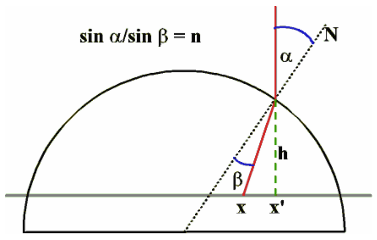


Fig. 3. The lense effect

The package contains several exemplary deformers, including more "physical" vortices useful for simulating whirling cloud patterns, such as on Fig. (4:A), and also some standard 3D mappings/projections discussed in the next section.

But suppose that we want to apply our "eddy" deformer to different zones of the texture space, and generate the celtic pattern on Fig. (4:B), or creating displaced and deformed lenses which would simulate water droplets on another surface. We will see that we must operate with both, direct and inverse operation, so technically the problem may be nasty. Composing global transformation is not difficult, for example it is obvious that a rotation of a texture about an *arbitrary* point may be realized in 3 stages: shifting the rotation centre to zero, rotating, and translating the texture back. Clastic defines:

```
rotatedAbout p0 angle                          //of a texture
 = translated p0 o rotated angle o translated (~p0)
```

For general warping, e.g. a composition of local translations shown on Fig. 4 (C), the situation is more complicated. A function $\mathbb{G}$ of type $\boldsymbol{Point} \rightarrow \boldsymbol{Point}$ which undergoes the geometric transformation $\mathbb{R}$ is a composition $\mathbb{R}\mathbb{G}\mathbb{R}^{-1}$, and the appropriate representation in the domain of textures is $\zeta\left(\mathbb{R}^{-1}\mathbb{G}^{-1}\mathbb{R}\boldsymbol{p}\right)$. Clastic implements three simple, generic deformer modifiers, its translation and rotation:

```
trfshift p0 deformer = ((+) p0) o deformer o ((+) (~p0))
trfturn ang deformer = rot2 ang o deformer o rot2 (~ang)
```

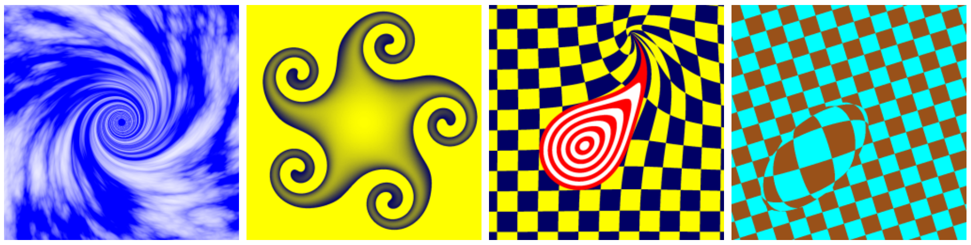


Fig. 4. More complicated mappings

where `rot2` rotates a vector, and of course the scaling. The "pentapus" on Fig. (4:B) is obtained by applying to the unit disk a 5-fold rotated and shifted `eddy` deformer. See also the Fig. (4:D). But suppose we want to translate a region near $(0, 0)$ using the transformation $\boldsymbol{x}' = \boldsymbol{x} + \exp(-\alpha|\boldsymbol{x}|)\boldsymbol{a}$. This is not only non-invertible directly, but if $\boldsymbol{a}$ is too big with respect to the warping range α, the texture may fold disgracefully over itself, with a loss of structural information. The package contains a Newton solver for such equations (for small $\boldsymbol{a}$), and a procedure which iterates (using `foldl`) small warping steps over a list representing a curve. The main purpose of this section was to show that such transformations can be *abstracted*, used generically, and easily composed, which makes their coding sometimes an order of magnitude shorter than the equivalent "classical", imperative approach.

3.1 3D textures

Texturing of surfaces in 3D is also a geometric transformation, and Clastic offers many basic tools for such manipulations. It implements the 3D vector algebra, and includes a simple, non-recursive ray tracer able to render spheres, cubes, and cylinders (and anything the user may wish to implement himself; for us this is just a part of testing interface). The viewport is attached to a virtual camera, the appropriate procedures generate a ray through the rendered pixel, find the intersection with the object, and compute the normal to the surface. The user has only to submit a texture function with ***Points*** belonging to the 3D space. If the user wants to cover the surface with a "wallpaper", an external image, as shown on Fig. (5:A) which presents the True Face of our Moon, Clastic may lift the 2D texture, e.g. a planet surface projection parameterized by (ϑ, φ), into a function of (r, ϑ, φ). The rendering is trivial, and appropriate functions are extremely short, the only interesting problems come from the lighting of such textures, but we cannot discuss this issue here. The remaining pictures on Fig. (5) show a marble sphere, another one covered with a bump-map, showing the interplay between 3D and 2D texturing processes, and a "wooden" cylinder. These constructions need a reasonably complete set of *random noise* generators.

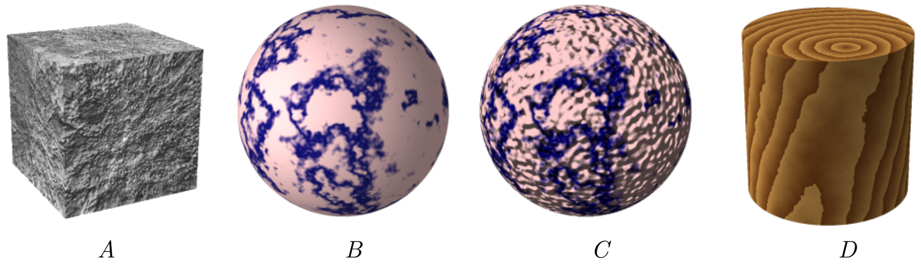


Fig. 5. 3-dimensional texturing

4 Random Textures

Since all "natural" surfaces possess some randomness, a rich collection of *noise* generators belongs to a standard panoply of shaders. In this section we will treat a few generic techniques, random textures are functions as any other generator, and to prevent all confusion: they *must* be state-less, deterministic functions which appear random, but which yield always the same value for the same argument. It is not possible to use a typical (even very good, such as Mersenne Twister), seed-propagating random generator during the texture generation, the results would depend on order of operations. The subject is known, and well covered by the literature, e.g., by the book [5]. See also a very instructive site of Ken Perlin [19] and the tutorial pages of Hugo Elias [20].

4.1 Basic generators

Perlin based his famous noise on the interpolation of random values stored *once* in an array. Those values were supplied before the rendering, using some standard random sequence generator. In a functional approach it is not very natural, the alternative is to use *directly* a pure function, as advocated already by Ward [21]. A typical example of such a function, of type ***Integer*** $\rightarrow$ ***Real*** may be

```
ergodic n
 # n = (n<<13) bitxor n
 = toReal((n*(n*n*15731+789221)+1376312589))/2147483648.0
```

adapted to 32 bit integers and returning a real in $[-1:1]$. We call such functions *ergodic* (rather than "random"), because they are stationary, unstable, returning results without visible correlations even for neighbouring arguments. (Ergodicity is a physical term, computer scientists use to call such mappings *hashing functions*...) Other parametrisations are also used in our package. In order to produce 2-dimensional distributions we may combine the arguments, e.g., defining `ergxy ix iy = ergodic (173 + (13*ix) bitxor (37*iy))`, as suggested by Ward in Graphic Gems.

Real (scalar) random distributions in the space of real (x, y) are based on a gradient version of Perlin noise visualized (specially) on Fig. (6:A) with the viewport size ≈ 12. It is generated by the classical algorithm: the points with integer coordinates $\boldsymbol{q}$ are attributed a random vector $\boldsymbol{g}$ — a pair of numbers in $[-1 : 1]$. Then, for a point $\boldsymbol{p}$ neighbouring 4 nodes $\boldsymbol{q}_i$ we compute $\boldsymbol{\xi}_i = (\boldsymbol{p} - \boldsymbol{q}_i) \cdot \boldsymbol{g}_i$, and finally a bi-Hermite (or better) interpolation combines these four contributions. Clastic contains also generators for vector noise in 2 and 3 dimensions.

4.2 Noise functions

The rest is relatively trivial, and consists in applying the already described techniques of functional compositions and transformations, but since this is an application paper, we discuss shortly what Clastic can do with the basic noise generators.

First, as shown on Fig. (6:A), and on Fig. (5:C) a noise can be interpreted as a height field, and after computing its gradient (for which Clastic has appropriate tools), it may be used as a bump-map.

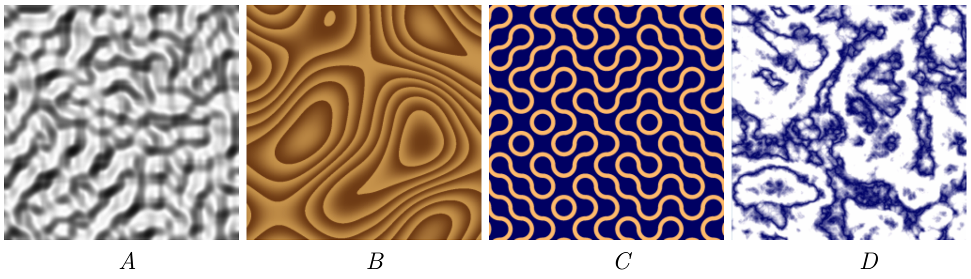


Fig. 6. Some random textures

Even a dull, apparently featureless basic noise instance over a region of size $1-2$ may be useful, if an appropriate colour map is used, see Fig. (6:B). There is nothing original here, the only advantage of Clastic is that such transformations are 1-line programs, not by specific obfuscate tricks, but by the genericity of the whole design. Even the basic discrete noise on integer nodes is useful for random tesselations, such as the Truchet pattern shown on Fig. (6:C) where the discrete noise is used to choose one of two elementary cell patterns.

Clastic contains some accumulating functionals permitting to construct wide-spectrum “fractal noise” and “turbulence” patterns by summing appropriately scaled and weighted basic noise instances: $\sum_{i=0}^{n} (1/2)^{i\kappa} \boldsymbol{g}\left(f_0 2^i \boldsymbol{p}\right)$, where κ is usually of the order 1, and the number of octaves n varies typically between 3 and 9. If composed with a sawtooth replicator, it may generate marble-like patterns like that on Fig. (6:D)

If $\boldsymbol{g}$ is the base noise we obtain Fig. (7:A), and if we sum its absolute value, we obtain the notorious turbulence pattern on Fig. (7:B), containing interesting filamentary structures. Applying to the turbulence some nonlinear transform and thresholding (1 line of code):

```
clouds p=1.0-exp(~4.0*max 0.0 (0.25+fractsum 7 1.1 0.6 p))
```

generates the clouds used to generate the tornado on Fig. (4:A).

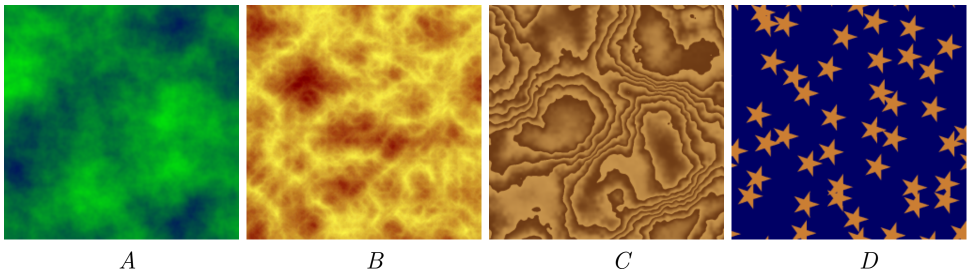


Fig. 7. Other random variants

4.3 Random deformations, and random placement of textural objects

Even more interesting are transformations of some geometric patterns, with noise functions (especially turbulent) used as deformers. One noise can perturb another one, as shown on Fig. (7:C) If a 3-dimensional cylindrical gradient $f(x, y, z) = \sqrt{x^2 + z^2}$ is deformed by random vector fractal sum displacement, rotated and cut by a surface, in a very few lines of code we obtain the generator which yields the Fig. (5:D).

If the basic noise is scaled so that every pixel has integer coordinates, the result is a random "white noise" pattern, which may be used e.g. to dither other textures. But random spreading of bigger objects, known as "bombing", such as shown on Fig. (7:D), is more difficult. This problem is discussed in [5], and we recognize that in a point-wise, implicit texture generation framework, the solutions are not natural. One possibility is to construct a union of placed objects, with random shift attributed to each of them. This may be very costly.

We present here the "jittering" method, which uses only the standard noise. The technique is similar to those applied to generate the Truchet pattern on Fig. (6:C). An object (here: a star) occupies a unit cell, with intrinsic coordinates between 0 and 1. The (integer) cell coordinates feed the `ergxy` generator, which supplies the shift of the object inside the cell (and it might serve also to change its colour, size, or to disallow its rendering). The fractional part of the point coordinates are used normally for rendering inside the cell.

If the object approaches the cell boundary, it may get truncated, so a more robust algorithm analyses also the cell neighbours. As we see, Clastic handles gracefully this problem.

5 Symmetries and Tesselation

Pavements, wallpaper, rugs... — the necessity to generate textures possessing tiled symmetric patterns is obvious. Texturing does not “redraw” them. In order to replicate a shape, the argument $\boldsymbol{p}$ of the generator must be appropriately *reduced*. The details of the reduction depend so strongly on the result the user wants to obtain, that a method which would be universal is a dream. General tesselations in our functional texturing framework will be discussed elsewhere, here we discuss some simple techniques which produce such tiles as on Fig. (8:B), starting with any motif (Fig. (8:A)) placed near the origin of the coordinate system. *The main objective of this section is to extract some useful, generic abstractions* .

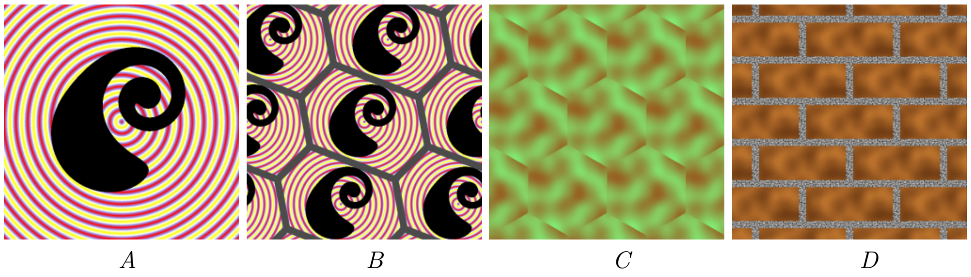


Fig. 8. Simple (and less so) tesselations

The first concept which must be understood is the *symmetry* of the design, the set of transformations which *leave the texture invariant.* We know that there are 17 so called “wallpaper groups”, containing translations together with rotational, reflective and glide-reflective (combination of an axial reflection and a translation along the axis) symmetries, see [22–24], or dozens of other books, e.g. [25], and references available on Internet. We have to distinguish between three distinct entities:

1. The *Unit Cell* (UC), the basic fragment of the “crystallographic”, translational lattice. Its translations fill the entire plane. IF UC is rectangular, the tesselation is trivial, this is what we see as typical background motifs on windowing systems, Web pages, etc.
2. The *Fundamental Region* (FR) (usually a fragment of the Unit Cell), which generates the plane through all its symmetry operations.
3. The *Motif Region* (MR), which *does not need* to cover any of the above.

A non-trivial FR-Motif combination is shown on Fig. (9), a tribute to Escher.

5.1 Translational tiling

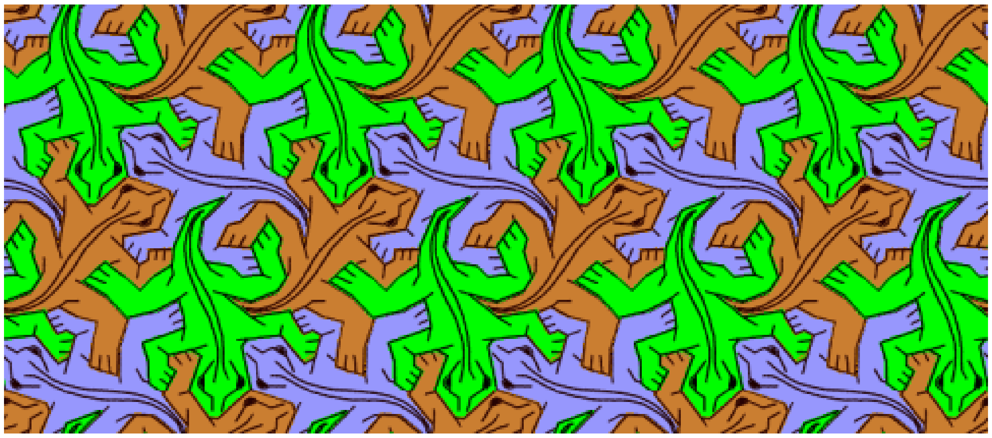


Fig. 9. Escher reptiles

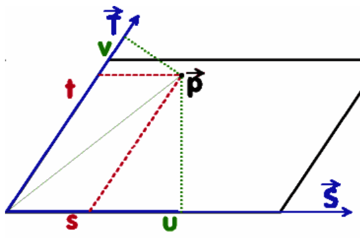


Fig. 10. Unit Cell attributes

Translations are handled universally, independently of other symmetries, and they are always present if the texture occupies the entire plane.

Clastic defines a special data structure, `UnitCell` constructed by the specification of its two non-Cartesian axes, $\boldsymbol{S}$ and $\boldsymbol{T}$. It contains all the conversion procedures permitting to decompose the current point (vector) in local coordinates, to *reduce these coordinates*, and to reconstruct the reduced cartesian vector. For the simplest symmetry, called P1, for which FR and UC are identical (it is a parallelogram) nothing else is needed, unless the Motif is shared between neighbouring cells.

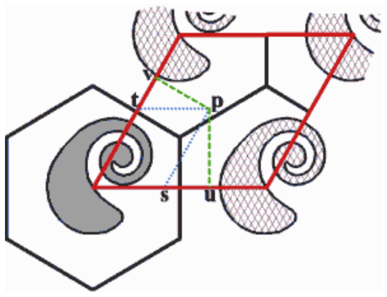


Fig. 11. Hexagonal, P1 tiling

The Figure (11) shows that UC must be cut into 4 pieces in order to reconstruct the hexagonal design. The user must supply the motif definition and its location. So, in which sense the genericity of Clastic may be helpful here?

- The reduction of the Motif area *within* the UC is the only thing to do.
- It is easy to define some universal clichés; Clastic has a universal generator of all hexagonal patterns, the user specifies only two (corner) points inside the UC, the rest is unambiguously constrained.

– The reducing procedure is sensible to the distance of a point to the UC edge, and as a “free bonus” it may add some “mortar” texture to the basic repetitive filling.

All is composable and deformable, and seeing that the classical brick wall has a hexagonal topology, we needed just a few lines to generate the textures on Figs. (8:B, C, and D). One more comment seems useful. The coordinate reducers are functions ***Point*** $\rightarrow$ ***Point***, and they behave as deformers. Their composition requires some attention in order not to distort, or to break the texture inside.

5.2 Rotational symmetry; Group P3

Our second example is the group describing a rotational symmetry of order 3, which generates Echer reptiles.

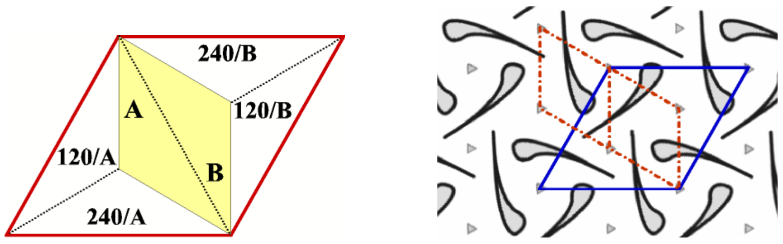


Fig. 12. P3 group: UC, FR, and a pattern with two different FRs

The Unit Cell is a rhombus with 60 and 120 degrees. Only one third of it, the central smaller rhombus, is the FR, the remaining parts are obtained by rotations (to which we attached some colour changing procedures while generating the reptile tiling).

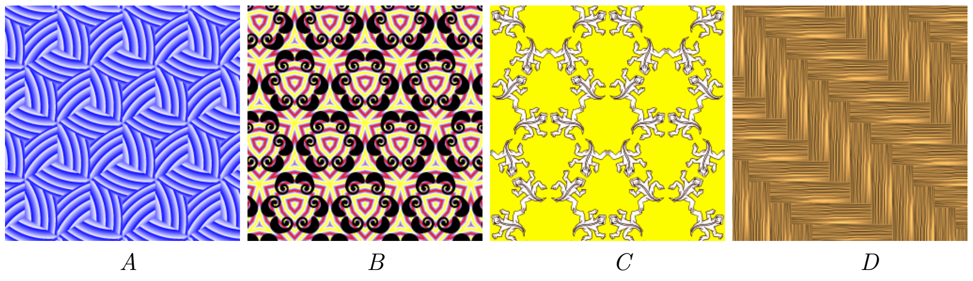


Fig. 13. More tesselations

The relation between the Motif and the Fundamental Region is not universal, and the generation of the Fig. (13:A) ignores this. If the Motif occupies the FR, the reducing procedure for P3 is extremely simple, less than 10 lines: it applies already known `rotatedabout` transformer, using as pivots the corners of FR which are inside UC, and thus filling the Unit Cell. In more complicated cases, such as typical Escher patterns, the human insight is necessary, although there are papers which deal with automatic "Escherization" of arbitrary shapes.

5.3 More "wallpaper" symmetries...

Clastic implements a subset of all wallpaper groups, this work is still in progress. But no new techniques are needed, everything is reused. We had just to add a few primitive reducers for 4- and 6-fold rotations, mirrors, and glide-reflections, and again, in a few lines it was possible to construct the reducers for the P6ML ("kaleidoscopic") group, depicted on Fig. (13:B,C), and a generic (arbitrary proportions, arbitrary filling) classical parquet generator, whose boundary has the symmetry PGG, containing some glide-reflections (about diagonal axes, not immediate to see...)

6 Woven patterns

Tiling is not the only generic method to produce repetitive patterns. Another technique is weaving, knitting, etc., the interlacing of linear objects is also omnipresent (and several programming applications helping to design fabrics are available commercially).

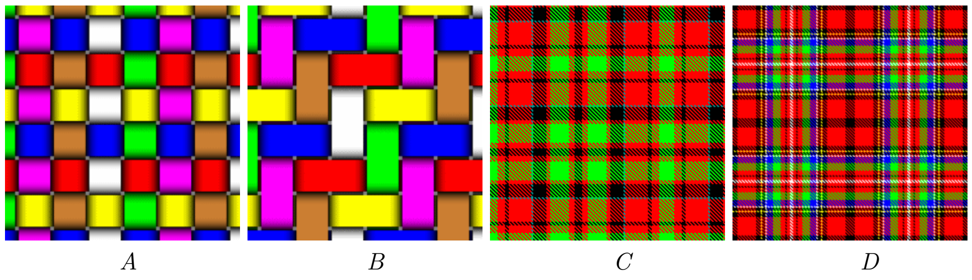


Fig. 14. Woven patterns

We added to Clastic a small subpackage which can generate the patterns shown on Fig. (14). The user specifies his "sett", the sequence of thread colours and their width, and the virtual "loom" driving protocol: a Boolean function on (x, y) (Integers!) which tells the texturing function whether at a given point the horizontal thread masks the vertical or *vice-versa*. The darkening decoration needed just a variant of `smoothpulse` as a mask.

This work is not complete, manufacturing baskets, rustic chairs, ring armours, etc. with hexagonal or octogonal symmetry may be technically more involved, requiring some symmetry considerations, but Clastic is prepared to deal with such problems.

7 Further Work and Conclusions

Our texturing library belongs to a broader pedagogical project: teaching of image synthesis with the aid of functional methods. Clastic is less suitable for those who just need a few textures, more for those who want to learn how to do them, and how to build texturing applications, with better interfaces, and well tuned generators of regular patterns and noise. It contains a rich collection of utilities, including some bitmap processing routines, noise generators, geometric tools, and equation solvers. The advantage of the presented functional methods with respect to, say, the Renderman shading language, is twofold.

Universal functional languages are rich enough to write *complete* rendering applications. Their data abstraction facilities make it easy to implement recursive ray tracers, and also to exploit them as the *scene description* languages, with the definition of objects, cameras, light sources, etc. Their genericity: overloading based on a system of type classes, and the possibility to use partial applications and other higher-order functional objects, facilitates the implementation of reusable building blocks and transformers for all kind of graphical objects. The texturing is just a concrete application of this methodology.

Textural functional objects are convenient for teaching. Some informal specifications as "shifted threshold", or "filtering out the Blue component of the texture A by a chessboard pattern" have ***almost direct***, very short, (often one line) concrete implementations. Image synthesis involves geometry, numerics, some knowledge of optics, etc., and requires many test runs. Even if the impatience of students makes them often looking for speed in order to economise computer time instead of their own, the possibility to transform an abstract specification in an algorithm, and to debug it fast before converting it into an optimized version, is very important.

Only part of this work is reported here. The tesselation sub-package will be presented elsewhere. We have omitted also the discussion of the Automatic Differentiation subpackage: a generalization of the functional differentiation framework described in [26] to general vector structures, permitting an easy and precise computation of gradients (in $\boldsymbol{x}$ and colour spaces), and useful for the construction of bump maps, contours of implicitly defined areas, etc. This issue is discussed in the Clastic tutorial available from the author. The interface to a ray tracer could not be discussed either. Complete shaders need not only a point in the texture space, but are provided by the rendering engine with the actual surface point in the scene space, the normal to the surface, the parameters of light sources, the pixel resolution (e.g. for the anti-aliasing) etc. While functional methods permit elegant and natural coding of these features, they remain outside the aims of this paper.

8 Acknowledgments

We are indebted to Conal Elliott for his work and interesting discussions. We thank also one of the Reviewers for pointing out missing references.

References

1. Peter Henderson, *Functional Programming, Application and Implementation*, Prentice-Hall, (**1980**). Also: *Functional Geometry*, Symposium on Lisp and Functional Programming, (**1982**).
2. Emmanuel Chailloux, Guy Cousineau, *Programming Images in ML*, ACM SIGPLAN Workshop on ML and its Applications (**1992**).
3. Simon Peyton Jones, S. Finne, *Pictures: a Simple Structured Graphic Model*, Preoceedings, Glasgow Functional Programming Workshop, (**1996**).
4. Jerzy Karczmarczuk, *Geometric Modelling in Functional Style*, Proc., III Latino-Americal Conf. on Functional Programming, Recife, Brazil, (**1999**).
5. David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, *Texturing and Modeling. A Procedural Approach*, AP Professional, (**1998**).
6. B. Gibson-Winge, *Texture Synthesis*, `www.threedgraphics.com/texsynth` .
7. John C. Hart, *Procedural Texturing*, Web course, available from the site `graphics.eecs.wsu.edu/cpts548/procedural/sld0001.htm` .
8. Karl Sims, *Artificial Evolution for Computer Graphics*, Comp. Graphics **25**(4), pp. 319–328, (**1991**). See also the site `genarts.com/karl/papers/siggraph91.html` .
9. Conal Elliott, *Functional Images*, `research.microsoft.com/~conal/Pan` with references, plenty of additional documentation and examples.
10. Andrew Cooke, *Pancito*, site `www.acooke.org/jara/pancito` .
11. Steve Upstill, *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley, (**1990**).
12. Larry Gritz, *Blue Moon Rendering Tools*, Exluna Inc., `www.exluna.com/bmrt/` .
13. Web site `www.pixar.com` .
14. Pat Hanrahan, Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, *A Real-Time Procedural Shading System for Programmable Graphics Hardware*, SIGGRAPH, (**2001**). See also `graphics.stanford.edu/projects/shading` .
15. Rinus Plasmaijer, Marko van Eekelen, *Concurrent Clean Language Report, Version 1.3*, HILT B.V. and University of Nijmegen, (**1998**). See also `www.cs.kun.nl/~clean` .
16. Sean Gibb, Peter Graumann, *SynTex*, Synthetic Realms, Calgary, Canada. Web site `www.SyntheticRealms.com` .
17. Pedagoguery Software Inc., *GrafEq*, `www.peda.com/grafeq` .
18. Jules Bloomenthal (ed.), *Introduction to Implicit Surfaces*, Kaufmann, (**1997**).
19. Ken Perlin, `www.noisemachine.com` , see also `mrl.nyu.edu/perlin` .
20. Hugo Elias, tutorial, `freespace.virgin.net/hugo.elias` .
21. G. Ward, A recursive Implementation of the Perlin Noise Function, in *Graphic Gems II*, ed. James Arvo, AP PROFESSIONAL, pp. 396–401, (**1991**).
22. Doris Schattschneider, *The Plane Symmetry Groups: Their recognition and notation*, American Math. Monthly. **85**, pp. 439–450, (**1978**).
23. Xah Lee, *The Discontinuous Groups of Rotation and Translation in the Plane*, Web pages `www.best.com/~xah/`. Contains a good overview of literature.
24. David E. Joyce, *Wallpaper Groups (Plane Symmetry Groups)*, tutorial. Web site `aleph0.clarku.edu/~djoyce/home.html` .
25. A. Shubnikov, V. Koptsik, *Symmetry in Science and Art*, Plenum, (**1974**).
26. J. Karczmarczuk, *Functional Differentiation of Computer Programs*, Journal of Higher Order and Symbolic Computing **14**, (**2001**).