Book Section:
Borgo, Rita, Duke, David, Wallace, Malcolm et al. (2006) Multi-cultural visualization : how functional programming can enrich visualization (and vice versa). In: Vision, Modeling, and Visualization 2006 : Proceedings, November 22 - 24, 2006. AKA Verlag - IOS Press, pp. 245-252.

VMV Copyright Notice

Title of Work: **Multi-cultural visualization: How functional programming can enrich visualization (and vice versa)**

Author(s): Rita Borgo, David Duke, Malcolm Wallace, Colin Runciman

Publication in the Proceedings of Vision, Modeling, and Visualization 2006,
edited by T. Aach, C. Bischof, L. Kobbelt, R. Westermann

Multi-cultural visualization: How functional programming can enrich visualization (and vice versa)

Rita Borgo[†], David Duke[†], Malcolm Wallace[‡], Colin Runciman[‡]

†School of Computing
University of Leeds
Leeds, UK

‡Dept. of Computer Science
University of York
York, UK

Email: {rborgo,djd}comp.leeds.ac.uk {\{Malcolm.Wallace,Colin.Runciman}\}cs.york.ac.uk

Abstract

The past two decades have seen visualization flourish as a research field in its own right, with advances on the computational challenges of faster algorithms, new techniques for datasets too large for in-core processing, and advances in understanding the perceptual and cognitive processes recruited by visualization systems, and through this, how to improve the representation of data. However, progress *within* visualization has sometimes proceeded in parallel with that in other branches of computer science, and there is a danger that when novel solutions ossify into 'accepted practice' the field can easily overlook significant advances elsewhere in the community. In this paper we describe recent advances in the design and implementation of pure functional programming languages that, significantly, contain important insights into questions raised by the recent NIH/NSF report on Visualization Challenges. We argue and demonstrate that modern functional languages combine high-level mathematically-based specifications of visualization techniques, concise implementation of algorithms through fine-grained composition, support for writing *correct* programs through strong type checking, and a different kind of modularity inherent in the abstractive power of these languages. And to cap it off, we have initial evidence that in some cases functional implementations are faster than their imperative counterparts.

1 Visualization Goals

Visualization as a discipline is a relatively new concept. In 1987 the U.S. National Science Foundation (NSF) established a Panel of experts [1] to report on the state and potential of visualization as a new science, bringing together graphics and computational technologies. This was not the first prominent call to focus attention on visualization; as far back as 1966, Sutherland [2], following the tradition common to scientists and mathematicians of creating lists of important open problems to focus attention, had compiled a list of 'unsolved problems' in computer graphics. Nor was the 1987 NSF report the last word; several new lists have been compiled since: a 1994 special Issue of IEEE Computer Graphics and Applications [17] was dedicated entirely to research issues in scientific visualization; in 1999 Hibbard [4] created a list of the top ten visualization open problems; and Johnson [5] presented a list of the most important issues crucial to the development of research in scientific visualization, a subset of which can be found in the recent official NIH/NSF report on Visualization Challenges [19].

Identifying a problem is a prerequisite to its solution, and such lists of open problems help to evaluate the state of the art of a field and suggest new research directions. In the context of this paper we take the 2006 NIH/NSF report as representative of the major issues raised by the visualization community. The report is broad, and contains questions we do not even consider here. Rather, we focus our attention on a subset of the open problems, which we will look at from a perspective that spans research in both visualization and functional languages.

An important statement made by the report is that there is currently a need for approaches that go beyond incremental improvements. The capability of evaluating results, sharing of resources, integrating with other disciplines represents a fundamental step towards an answer to visualization challenges. To achieve the result the NSF panel nominated several interesting areas of action:

Domain Integration and Collaboration with Neighbouring Disciplines. There is a need at the domain level to focus on real rather than ideal data, addressing aspects like heterogeneity, change over time, error and uncertainty, scale and data scope. Scientific visualization is a cross-disciplinary process, statistics, data mining and image processing techniques play an important role in the understanding of a phenomena. New applications need to be able to easily integrate with techniques and tools coming from other disciplines.

Exploration of Novel Visualization Techniques and Metaphors. As datasets keep growing both in size and complexity, there is a need for exploration of new visualization techniques and approaches. Challenges coming from heterogeneous, multivariate, dynamic data require to easy identify and qualify the design space of visualization techniques. Heterogeneities and complexity apply not only to datasets but also hardware resources. Rapid technology development asks for flexible techniques able to cope with the wide variety of available devices: high resolution and lightweight projectors, flat panel displays, ubiquitous technologies etc.

Systems Evaluation. The field of visualization has unique evaluation challenges. Quantitative evaluation can be easily performed, measuring time and memory required by an application or algorithm, however such metrics do not provide any information about the qualitative impact of a system. Case studies are a means to achieve such an evaluation although they come at a stage where the system design has been already settled and is hard to modify deeply. The ability to develop prototype systems and perform usability studies is a winning quality. Although the outgrowth of these qualities relies in the ontological organization of the visualization process itself and in the formalization of visualization design. Formalization of visualization techniques and approaches (expressiveness of tools and design principles) would help in answering not only 'whether' something helps but also 'why' and 'how'.

Scalability, efficiency in utilizing novel hardware, multifield visualization, visual abstraction are all requirements to allow visualization to move from being seen as a postprocessing step of the scientific computing pipeline, to represent a more complex discipline able to merge different aspects of a common target: modeling, simulation and visualization of data. In the spirit of experimenting with novel approaches, we present in Section 2 some interesting aspects of functional programming that seem to answer some of the desired requirements outlined throughout the NSF report. We then take surface fitting techniques as an example and show in Section 3 results obtained by applying our approach to a well-known surface extraction algorithm: contour following.

2 Functional Programming Goals

Functional programming has at times been seen as an academic tool, an elegant but computationally expensive way of expressing basic problems like the Fibonnaci series or the Towers of Hanoi, or an obscure notation used in esoteric branches of AI such as theorem proving. While this could be partially true for earlier functional languages like Lisp, it no longer holds as a generalization. Modern functional languages such as Haskell [25], Gofer and Clean [24] have expressive polymorphic type systems and are inherently lazy; that is, unlike languages such as Lisp and ML, an expression is evaluated at most once, if its result is required to construct an output. They have been used within scientific domains and in graphics. For example, Chakravarty and Keller [14] develop a Haskell library to solve some of the problems arising from sparse matrix multiplications, while Karczmarczuk [7] expresses some of the fundamentals of Quantum Mechanics in Haskell. Both Haskell and Clean have been employed in the development of videogames platforms including texture generation [15] and mapping [8]. We consider visualization applications separately. Our concern here is how functional programming provides a view on NSF report challenges, different from the more incremental perspective of the technologies 'traditionally' used in visualization.

Functional Languages as a Specification Tool The importance of a flexible type system is too often overlooked; the polymorphic system used in functional languages, and the fact that programs are constructed by composition of small(er) func-

tions, means that type definitions act as a form of machine-checkable documentation. One finds that once a functional program is compile-time correct, there is more likelihood that it is run-time correct than for other classes of language, both because the type system captures more (e.g. constraints on types), and because through fine-grained composition, it is harder to put together two components that don't make sense. Dually, function specification deals only in data-flow, *what* the function does; there is no prescription of *how* the result is computed[1]. Programs expressed in a functional language are also concise, making it easier to comprehend significant computational patterns directly. In contrast, description of imperative algorithms require more use of natural languages to distinguish the 'what' from the 'how'; and the pitfalls of natural languages as a specification technique are well known.

Heterogeneity of data and data-structure A *polymorphic* function is one that is independent of the type of data on which it operates; for example, computing the length of a list is independent of the kind of data held in the list. Polymorphism supports one kind of generic programming, but recent advances in this area make it possible to go further. A *polytypic* function is generic over the organization of its data; thus a polytypic 'size' function can generalize the notion of 'length' from just lists to a whole class of data organization that includes trees and queues. Such a function is written only once, defined over the structure of types, and can then be applied to most types. The ability of generic programming to address heterogeneity issues in visualization draws critically on the 'functional' aspect of FP. Functions are first-class citizens: they can be passed as parameters, and returned as results, from so-called *higher-order* functions. These abstract from common patterns of computation, for example we can map a function over a data structure to produce a new version of the structure but where individual data have been transformed by the function.

System Evaluation Although functional programs can be judged quantitatively just like their imperative counterparts, they also admit to a qualitative evaluation. The inherent mathematical structures and foundations of systems based on functional programming make them amenable to proof of correctness:

- each function in a program corresponds to a referentially transparent equation, i.e. it is independent of any dynamic global state
- each side-effect-free equation can therefore be tested or analysed separately from the rest of the code, with full confidence of completeness.

Functional programs can be used as specification of final products that will be implemented in a procedural language thus acting as an executable specification of an algorithm while being a program as well. This property makes them suitable to be used for prototyping, furthermore since they resemble a collection of mathematical equations, functional programs are ruled by the ordinary laws of mathematics and thus easier to derive, transform and verify.

With respect to the need for novel approaches to visualization, the use of functional languages meets the requirement. There has been some effort carried out in the visualization and related fields. In [10] Elliott presents a purely functional Haskell-embedded language for 3D graphics cards, the language integrates procedural surface modeling, shading and texture generation. A less ambitious, but not less interesting, project has been performed in [11] where Fokker provides a functional specification of the JPEG algorithm, showing the gain in terms of clarity and readability of the code obtained through the functional implementation of a complex graphics algorithm. Another work worth mentioning is Page and Moe [12] where the authors report earliest results on the development of a reservoir simulation system in a purely functional language. An important issue raised by the Page and Moe paper is that of the ability of functional languages to scale to distributed or parallel kinds of resources. Functional programs are inherently parallelizable if we take into consideration that the evaluation of an expression cannot have side effects, independent subexpressions can be evaluated in any order or in parallel. In the next section we present some of the results obtained and limits encountered in merging functional languages with classic visualization problems like surface fitting techniques.

[1] This applies also to seemingly 'imperative' features like IO and exception handling, which in functional languages are handled through composition of *commands* within a mathematical structure called a monad.

3 Merging The Two Views

As a starting benchmark and illustrative example, we choose a widely used surface extraction algorithm—contour following—and the lazy functional language Haskell, to show the benefits of clear and concise expression combined with fine-grained, demand-driven computation. As visualization provides insight into data, functional abstraction provides new insight into visualization.

3.1 Contour Following Functionally

Contour following defines a class of algorithms which are capable of preserving both coherence and connectivity of cells. The algorithm's behaviour can be summarized as follow:

1. choose a cell that intersects with the field value;
2. construct the surface representing the intersection of cell and field;
3. for each face of the cell that intersects with this surface, the adjacent cell must also intersect with the surface;
4. *follow* the surface into each adjacent cell repeating steps 2 to 4 for that cell.

The aforementioned process guarantees continuity and topological connectivity of the generated contour, cells are reached through a path of neighbours and inspected only if intersected by the contour. The contour following method sketched above produces only a single connected contour, however a single field value usually corresponds to multiple contours and therefore multiple starting cells are needed. Such cells are called *seeds*, and a set of cells that represent the starting point from which all possible contours in a field can be generated is called a *seed set*. A seed cell is traditionally composed by three fields: an identifier within the seed set, its (i,j,k) indices within the dataset and the range of spanning values. Our implementation of the seed structure in Haskell looks as follows:

```
data Seed a = S PostCode Address (Range a)
              deriving (Eq, Show)

type PostCode = Int
type Address  = (Int, Int, Int)

data Range a = Range a a
             | Empty deriving (Eq, Show)
```

A seed cell is represented as an **algebraic data type**. Algebraic data types in Haskell are introduced by the keyword `data`, followed by the name of the type (in our example `Seed`), an equal sign and then the **constructors** (in our example `S`) of the type being defined. A constructor builds a record from several other types (here, `PostCode`, `Address`, and `Range` are the components of a Seed), and can also be used to pattern-match (or destruct) a record.[2] The name of types and of constructors begin with capital letters. The type variable (lower-case `a`) in the `Seed` type indicates that the type of the samples themselves is generic (polymorphic). Generic programming is an important concept for software development and many modern programming languages provide support for it. Haskell itself provides polymorphic functions and datatypes which together are sufficient to implement polymorphic data structures. As for [16] with respect to generics, modern functional languages provide an expressive power which languages like Java, C# and C^{++} still lack (although language support for generics of these language is continuing to evolve). In the present context genericity is shown in the possibility to reuse the algorithm with bytes, signed words, floats, complex numbers, and so on without change, although it could be pushed much further.

The types `PostCode` and `Address` are declared as synonyms for a triple and an integer element. `Range` is another algebraic data type, polymorphic on the type variable `a`, as previously explained, and with two **constructors**, `Range` and `Empty`, which define respectively a proper Range with two extremes (`a a`) or an Empty range. The `deriving` clause after the Seed (and Range) datatype declaration states that the compiler shall implicitly produce an instance of the classes `Show` and `Eq` for the newly defined datatype Seed. Both `Show` and `Eq` are built-in classes of Haskell; `deriving Show` tells the compiler that it can automatically *derive* a suitable implementation for the **show** and **read** functions; `deriving Eq` tells the compiler that it can generate a suitable implementation for equality **==**. Haskell's classes have a surface similarity to

[2] In general, a data definition may introduce several alternative constructors for a given type; for example, a type `Dataset` that allowed both regular and rectilinear grids might appear as `Dataset a = Reg XYZ [a] | Rect ([Float],[Float],[Float]) [a]`. Different kinds of dataset are then distinguished by their constructor.

object oriented classes, but in fact the system is independent of any specific data representation; only the common behaviours are factorized. This allows to model properties retroactively, currently available only in Haskell and ML derived languages.

Let's move to the real algorithm specification. In the Haskell implementation we have split the contour following algorithm into two main functions:

- *Traverse Seeds*: which given a seed set and a threshold value, searches the seed set for all the cells that constitute a seed for the given value;
- *Grow Contour*: which given a seed grows the contour, following the contour path through cells adjacent to the seed.

In Haskell:

```
traverse_seeds :: Dataset a → a → [Seed a]
                  → [Triangle]

grow_contour :: Dataset a →a →a →[Triangle]
```

These two declarations represent the type signatures of the functions. The first type signature shows that `traverse_seeds` takes three arguments, a dataset, a value and a list of seeds (i.e. the seed set), and returns a list of Triangles approximating the surface (the triangles can be directly fed up into OpenGL through the Haskell wrapper HOpenGL for rendering). We skip the trivial definition of the `Dataset` datatype since it is semantically similar to the earlier datatype definition for Seed. The second type signature shows that `grow_contour` takes three arguments, a dataset and two values, and returns a list of triangles as well. The implementation of the two functions is as follows:

```
traverse_seeds d thr seeds
 = concat $ map (grow_contour d thr) $
           filter (contains thr) seeds
   where
    contains thr (Seed _ _ r) = thr `inR` r

grow_contour d thr (Seed c _ _)
 = grow_from d thr (enQueue c)
                (MS.insert c MS.empty)
```

In Haskell, application of a function to arguments is by juxtaposition – no parentheses are needed – so in the definition of `traverse_seeds`, the arguments are `d` (the dataset), `thr` (the isovalue threshold) and `seeds` (the seed set). The `contains` function tests if a seed is intersected by the isovalue checking if the threshold value is 'in' (`inR`) the seed range `r`. The underscore keyword `_` is called **wildcard**, it is normally used to replace parameters that are not needed or that can be replaced by anything, in the present case since only the range field is used on the right-hand side of `contains`, the first two components of Seed (`PostCode` and `Address`) can be replaced by the wildcard. It is at this point important to introduce *higher-order* functions. From the very name "functional language" one can surely guess that functions are important. Indeed, passing functions as arguments, and receiving functions as results, comes entirely naturally. A function that receives or returns a function is called higher-order. An example of higher-order function is `map`, which takes a function `f` and applies it to every element of a sequence:

```
map :: (a→b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

This definition uses pattern-matching to distinguish the empty sequence `[]`, from a non-empty sequence whose initial element is `x`, with the remainder of the sequence denoted by `xs`. Colon `:` is used both in pattern-matching, and to construct a new list. The `filter` function is another example of a higher-order function; when applied to a predicate function and a list, it returns the list of those elements that satisfy the predicate.

In the definition of `traverse_seeds`, `filter` removes from the seed set all the seeds that do not contain the given threshold value, `map` then applies `grow_contour` to each of the remaining elements in the filtered seed set. The `$` symbol is a Haskell operator that expresses right-associative binding precedence, in some cases it allows parentheses to be omitted, and to make the code better readable, e.g.

```
f $ g $ h x  =  f (g (h x))
```

Finally the multiple sequences of Triangles generated by `grow_contour` are joined into a single sequence, by the standard function `concat`. A contour is grown through a breadth-first traversal of the cells intersected by the contour, starting from the seed cell. At each point the queue of cells to be traversed is augmented by those neighbours of the cell under inspection, that intersect the conotour as well. During the traversal there is the need to record the cells that have been already inspected. To track

visited cells, an imperative implementation might implement a bit-array; here, we can simply draw on a standard generic library for sets, and write

```
type MarkSet = MS.Set Int
```

specialising the generic type into one that stores the integers used to refer to cells in the dataset (the `MS` prefix is used to qualify definitions imported from the set library and prevent name clashes). The traversal is performed by the `grow_from` function; `enQueue` is a Haskell library function, polymorphic over the type of element stored in the queue structure (in this case, a PostCode):

```
grow_from :: Dataset a → a → Queue PostCode
             → MarkSet → [Triangles]
grow_from d thr q marked =case viewList q of
  Empty          → []
  (cell : rear) →
   let neighbours = continuations d thr cell
       unvisited = filter (¬·(MS.member
                   marked)) neighbours
       q' = foldl (|>) rear unvisited
       marked' = foldl MS.insert marked
                       unvisited
   in (mcube (at g) (address d cell) thr)++
              (grow_from d thr q' marked')
```

`grow_from` consumes all the seeds contained in the postcode queue. The `continuations` function given a location within a grid and a contour value, determines the neighbouring (`neighbours`) locations that will also intersect that contour. A neighbour is valid if the face defined by the intersection between it and the current cell has a range that includes the threshold.

```
continuations :: (Ord a) => Dataset a
                    → a → Int → [Int]
continuations d thr code =
 map encode $ filter penetrates
 [  (i>0, [v0,v1,v2,v3], (i-1,j,k))
   ,(i<isz-2, [v4,v5,v6,v7], (i+1,j,k))
   ,(j>0,     [v0,v2,v4,v6], (i,j-1,k))
   ,(j<jsz-2, [v1,v3,v5,v7], (i,j+1,k))
   ,(k>0,     [v0,v1,v4,v5], (i,j,k-1))
   ,(k<ksz-2, [v2,v3,v6,v7], (i,j,k+1)) ]
 where
  (i,j,k) = address d code
  penetrates (non_boundary, vs, _) =
      non_boundary ∧ thr ≥ minimum vs
                   ∧ thr ≤ maximum vs
  encode (_, _, addr) = postcode d addr
  line  = isz
  plane = isz*jsz
  v0 = d!(code)
  v1 = d!(code+line)
  ...
  v7 = d!(code+plane+line+1)
```

The predicate `Ord a` constrains the polymorphism: samples must have ordering operations defined over them. Function `penetrates` just tests if the cell is on the boundary, while `encode` returns the cell index inside the dataset interpreting its postcode and address; `isz, jsz, ksz` are the three dimensions of the dataset along the x,y,z axes. The `v0 ...v7` represents the indices to the eight neighbouring cells computed sweeping the dataset with a plane (in this case samples are spread over a regular grid). The `map` function iterates `penetrates` over each of the elements within the list. Note that if one of the `non_boundary` tests fails, the remainder of the expression is not evaluated and the corresponding v_i not computed. The aforementioned behaviour allows for cleaning the code from extra statements like redundant **if ...then ...else** or extra function or exception guards, making the code much more compact.

`unvisited` contains all the neighbouring cells that have yet not been visited while `q'` contains the new queue to which all the unvisited neighbours have been added. `marked'` instead represents the new markset to which the indices of the tested (therefore "traversed" in terms of threshold intersection) neighbouring cells have been added. The `foldl` function is another mportant higher-order function in Haskell. A fold applies a function to a list in a similar way to map, but it accumulates a single result instead of a list. `foldl` is a left-associative type of fold which processes the list from left to right:

```
foldl :: (a → b → a) → a → [b] → a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs
```

In the present case `foldl` applies the merging operator (`|>`) on the two parameters `rear` (back of the Queue) and `unvisited` (list of cells belonging to the neighbourhood of `cell`). An interesting aspect of the present code is the `mcube` function which corresponds to the Haskell implementation of the Marching Cubes approach. The Marching Cubes code, implemented in an optimized version in [13], has been easily re-used within the contour following implementation.

```
mcube :: a → (PostCode→Cell a) → PostCode →
         [Triangle]
mcube thresh lookup (x,y,z) =
 group3 (map (interpolate thresh cell
             (x,y,z)) (mcCaseTable ! bools))
```

```
 where
  cell  = lookup (x,y,z)
  bools = toByte (map8 (>thresh) cell)
```

The `cell` of vertex sample values is found using the `lookup` function that has been passed in. We derive an 8-tuple of booleans by comparing each sample with the threshold (`map8` is a higher-order function like `map`, only over a fixed-size tuple rather than an arbitrary sequence), then convert the 8 booleans to a byte (`bools`) to index into the classic case table. The result of indexing the table is the sequence of edges cut by the surface. Using `map`, we perform the interpolation calculation for every one of those edges, and finally group those interpolated points into triples as the vertices of triangles to be rendered. The linear interpolation is standard:

```
interpolate ::Num a => a → Cell a → PostCode
                    → Edge → TriangleVertex
interpolate thresh cell (x,y,z) edge =
  case edge of
    0 →  (x+interp,   y,  z)
    1 →  (x+1, y+interp,  z)
    ...
    11 → (x,   y+1,  z+interp)
 where
  interp = (thresh - a) / (b - a)
  (a,b)  = selectEdgeVertices edge cell
```

Although `interpolate` takes four arguments, it was initially applied to only three in `mcube`. This illustrates another important higher-order technique: a function of n arguments can be *partially applied* to its first k arguments; the result is a specialised function of $n - k$ arguments, with the already-supplied values 'frozen in'. The predicate `Num a` constrains the polymorphism: samples must have arithmetic operations defined over them.

3.2 Observations

The code developed so far features some of the interesting properties of a language like Haskell. While code readability can be seen as a matter of personal flavour, expressiveness and abstract power of the language make the code much more compact and clean with respect to the original imperative coding of the same algorithm. A scientist, researcher or teacher does not always cope easily with high tuned but illegible code: when the desired aspect is the methodology of the computation, a more abstract approach to programming is needed. Haskell syntax is extremely compact, the elegant use of layout allows to get rid of redundant keywords and parentheses (i.e. indenting means continuation of the previous construction). The possibility to partially "clean" the code from administrative details like verbose loops with dozens of exception guards is a real gain. Support for type aliasing restricts the verbosity of complex data structures especially when dealing with generic data. The processes of thinking, algorithmization and coding are intertwined in our specification: although the above code is high-level and uses simplistic type structures, it is already complete and executable (Figure 1 is the result of applying the given code to a seed set generated for the neghip dataset), and can be used to test the correctness of the specification. The functional specification can be used as an executable program specification (formal prototype) even if the final product must be implemented in a procedural language.

4 Considerations

Functional programming can be successfully used to specify and implement complex visualization algorithms. In [13] it is shown how an FP approach can be used to efficiently engineer a full scale visualization problem in terms of performance and memory issues; the context of the current paper widens the approach, extending the view to a broader set of scientific visualization issues. If we consider the desirable characteristics of a specification language, we can outline expressive power and unambiguous semantics as the preferred ones. From this point of view functional languages are computationally complete: a modern language like Haskell exhibits minimal ambiguity and high readability, due to a tight binding with its denotational semantics. Functional languages are actively used in industry as shown in [21], though several issues related to their use remain to be solved. The most common barrier is a lack of some domain-specific libraries, lack of platform support (debugging, profiling and tuning tools), and as a consequence, occasionally poor performance. However when applied to some kinds of problems (for example when involving space allocation) the performance of functional languages rivals C and in the average case they underperform at most on a factor of two. When applied to visualization problems it is shown that implementation of algorithms like

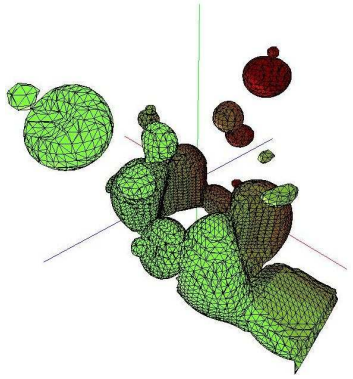

Figure 1: Isosurface Extraction from a Set of Seeds generated for the Neghip dataset. Different colors indicate different seeds.

Marching Cubes and Marching Tetrahedra with a functional approach [13] can outperform the VTK counterparts for certain large datasets. The purpose of this work is not claim that functional approaches should replace optimized imperative code. We show instead how functional programming can be successfully applied to face some of the problems peculiar to the visualization field, and how it represents a challenging field worthy of further investigation if we wish to gain useful insight on topics where the integration of results and techniques, human collaboration, and a need for general reusable patterns play a master role.

References

[1] B. McCormick and T. DeFanti and M. Brown. Visualization in scientific computing. *Journal of Computer Graphics,* 21, 1987

[2] I. E. Sutherland. Ten Unsolved Problems in Computer Graphics. *Journal of Datamation,* 12(5):22–27, 1966

[3] J. Blinn. Keynote Address: SIGGRAPH 98. *Journal of Computer Graphics,* 33(1):43–47, 1999

[4] B. Hibbard. Top Ten Visualization Problems. *Journal of Computer Graphics,* 33(2):21-22, 1999

[5] C. Johnson. Top Scientific Visualization Research Problems. *Journal of IEEE Computer Graphics and Applications* 24(4):13–17, 2004

[6] A. U. Frank, and W. Kuhn. *Lecture Notes in Computer Science,* 951:184, 1995

[7] Jerzy Karczmarczuk. Scientific computation and functional programming. *Comput. Sci. Eng.,* 1(3):64–72, 1999

[8] M. Wiering and P. Achten and M. J. Plasmeijer, Using Clean for Platform Games. In *Lecture Notes in Computer Science: Selected Papers from the 11th Int. Workshop on Implementation of Functional Languages* 1868:1–17, 2000

[9] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In . *SIGGRAPH,* 1987, 163–169

[10] C. Elliott. Programming graphics processors functionally. In *ACM SIGPLAN Workshop on Haskell,* 2004, 45–56

[11] J. Fokker Functional Specification of JPEG Decompression, and an Implementation for Free. In *EG Workshop on Programming Paradigms in Graphics* 1995, 102-117

[12] R. L. Page and B. D. Moe Experience with a large scientific application in a functional language. In *Int. Conf on Functional programming languages and computer architecture* 1993, 3–11

[13] D. Duke and M. Wallace and R. Borgo and C. Runciman. Fine-grained Visualization Pipelines and Lazy Functional Languages. In *IEEE Visualization Conf.,* 2006, (under final review)

[14] M. Chakravarty and G. Keller. An Approach to Fast Arrays in Haskell. In *Advanced Functional Programming* 2002, 27–58

[15] J. Karczmarczuk. Functional Approach to Texture Generation. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages* 2002, 225–242

[16] R. Garcia and J. Jarvi and A. Lumsdaine and J.G. Siek and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented Programing, Systems, Languages, and Appl.* 2003, 115–134

[17] L.J. Rosenblum. Research Issues in Scientific Visualization. In *IEEE Computer Graphics and Applications* vol.14, n.2, March/Apr 1994

[18] J. Blinn, What I Like and Don't Like About the State of Visualization Today. In *VIS '03: Proc. of the 14th IEEE Visualization 2003 (VIS'03)*, 2003, 68

[19] C. R. Johnson and R. Moorehead and T. Munzner and H. Pfister and P. Rheingans and T. S. Yoo. *NIH-NSF Visualization Research Challenges Report* IEEE Press, 2006

[20] P. S. Heckbert. *Ten Unsolved Problems in Rendering-Workshop on Rendering Algorithms and Systems.* 1987

[21] A. Moran *Report on the First Commerical Users of Functional Programming Workshop*, 2004 P. Wadler, University of Edinburgh.

[22] C. L. Bajaj and V. Pascucci and D. R. Schikore *Seed Sets and Search Structures for Optimal Isocontour Extraction.* Technical Report 99-35, Austin, TX, 1999

[23] N. Nethercote and A. Mycroft The cache behaviour of large lazy functional programs on stock hardware. In *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, 2002, 44–55

[24] *Clean* http://www.cs.ru.nl/ clean

[25] *Haskell: A Purely Functional Language* http://www.haskell.org