IEEE Copyright Notice

Title of Work: **Fine-grained Visualization Pipelines and Lazy Functional Languages**

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

Publication in the **IEEE Transactions on Visualization & Computer Graphics Journal, Vol 12, Issue 5**

Fine-grained Visualization Pipelines and Lazy Functional Languages

David Duke, Malcolm Wallace, Rita Borgo, and Colin Runciman

Abstract—The pipeline model in visualization has evolved from a conceptual model of data processing into a widely used architecture for implementing visualization systems. In the process, a number of capabilities have been introduced, including streaming of data in chunks, distributed pipelines, and demand-driven processing. Visualization systems have invariably built on stateful programming technologies, and these capabilities have had to be implemented explicitly within the lower layers of a complex hierarchy of services. The good news for developers is that applications built on top of this hierarchy can access these capabilities without concern for how they are implemented. The bad news is that by freezing capabilities into low-level services expressive power and flexibility is lost.
In this paper we express visualization systems in a programming language that more naturally supports this kind of processing model. Lazy functional languages support fine-grained demand-driven processing, a natural form of streaming, and pipeline-like function composition for assembling applications. The technology thus appears well suited to visualization applications. Using surface extraction algorithms as illustrative examples, and the lazy functional language Haskell, we argue the benefits of clear and concise expression combined with fine-grained, demand-driven computation. Just as visualization provides insight into data, functional abstraction provides new insight into visualization.

Index Terms—Pipeline model, laziness, functional programming.

1 Introduction

A number of architectures have been proposed and developed for data visualization, including spreadsheets [17], relational databases [23], spray-rendering [24], scene graphs [22], and pipelines [7, 28]. They provide a layer of application-oriented services on which problem-specific visualization tools can be constructed. Of the approaches explored to date, the pipeline model has found the most widespread use. It underlies the implementation of well-known systems such as AVS [28], SCIRun [25], and VTK [26], and also serves as a conceptual model for visualization workflow [6].

Building layers of service abstraction is an approach that has served computing well in the past, giving developers reusable domain-independent blocks for building an application. For the pipeline model, services provide the capability to organize visualization operations within a dataflow-like network. Some pipelined systems extend the basic model with demand-driven evaluation and streaming of dataset chunks, again frozen into the service layer. However, this layered approach fixes design decisions associated with the services, without regard for the operations that are implemented in terms of those services. Pipeline services provide a lazy, dataflow-like model, but client operations are defined as a separate layer of stateful computation.

An alternative set out in this paper is to use a programming technology that naturally supports operations fundamental to pipelined visualization. Implementations of 'lazy' functional languages have advanced significantly over the last decade. They now have well-developed interfaces to low-level services such as graphics and I/O. In this paper we take surfacing as an archetypal visualization task, and reconstruct two fundamental algorithms. We illustrate how pipelining and demand-driven evaluation become naturally integrated within the expression of an algorithm. The result is a simplified presentation, generating fresh insight into how these algorithms are related. The resulting implementations have a pattern of space utilization quite different to their imperative counterparts, occupying an intermediate point between purely in-core and out-of-core approaches.

Section 2 summarizes the main features of the pipeline model, in particular the capabilities that we seek to improve. Section 3 revisits the basic marching cubes algorithm for surface extraction, using the lazy functional language Haskell [14]. Through a series of refinements, we show how pipelining and demand-driven evaluation allow the use of *memoization* to improve performance. An extension to resolve topological ambiguities [18] in Section 4 shows how these fine-grained abstractions can be reused. Section 5, on evaluation, gives particular attention to the space performance of our implementation. The lazy streaming approach set out here features low memory residency, even with larger datasets. Section 6 discusses related research. The work reported here is a first step in a much larger programme of work, and in Section 7 we set out a longer-term vision of how functional programming can contribute novel ways of solving the technical challenges of visualization.

2 Pipelines: Plumbing for Visualization

Pipelines as a conceptual model for the visualization process were first proposed by Haber and McNabb [6]. Their use for implementing visualization is usually traced to the work of Upson et.al. [28] on AVS; they in turn cite the earlier work of Haeberli [7] on the ConMan dataflow system for interactive graphics. All these models represent the visualization process as a directed graph. Nodes are processing elements; arcs represent data dependencies: an arc from component A to B means the output of A is required in order for B to execute. Importantly, this can also be expressed by saying that if B needs to execute, it must first ensure it has up-to-date data from A. In this way, a visualization application can be considered as a demand-driven dataflow system, with update demands on graphics windows at the end of the pipeline "pulling" the data through intermediate transformations. The pipeline originates at source components, typically interfaces to the external environment.

Pipelined systems often provide *memoization*: computed outputs are stored, and only regenerated when there is a change to some parameter of the component used to produce them. For each component there is usually a choice whether to retain its output or regenerate it each time it is required; the trade-off is between memory use and computation time. Figure 1 shows a simple pipeline with four components, each retaining its output: a file reader, an isosurfacer, a filter to compute polygon normals, and a mapper which renders the polygons to

- *Rita Borgo and David Duke are with the School of Computing, University of Leeds, UK, E-mail: {rborgo,djd}@comp.leeds.ac.uk.*
- *Colin Runciman and Malcolm Wallace are with the Department of Computer Science, University of York, UK, E-mail: {Colin.Runciman,Malcolm.Wallace}@cs.york.ac.uk*

Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

a display. Three of the components have parameters. If any parameter is changed, some or all of the pipeline may need to re-execute. In the example, changes to the viewing parameters require only the mapper to re-execute, but if the user adjusts the isosurface threshold, the isosurfacer, normals filter and mapper all need to re-execute, in that order.

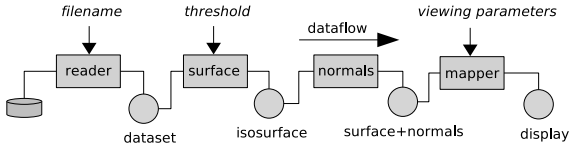

Fig. 1. Isosurfacing Pipeline

There are different approaches to managing pipeline execution. Systems such as IRIS Explorer employ a central executive to monitor the state of components and decide which to re-execute in response to a parameter change. VTK, in contrast, implements a decentralized model[1] where components and data are timestamped, and an update request at one part of the pipeline triggers 'upstream' components to execute only if their data needs to be regenerated. Relative merits of these schemes are discussed in [26].

Streaming [16] is an enrichment to the basic model that allows a pipeline to pass datasets in chunks. For scientific data, such chunks are usually spatially contiguous subsets of the full extent. Some algorithms, for example Marching Cubes [19], can operate on individual chunks in isolation. Others require access to the full dataset, for example surface reconstruction: the dataset may be passed as a sequence of chunks, with downstream and upstream algorithms working on these sequences. In between the extremes of full-dataset versus arbitrary chunk are algorithms such as Gaussian smoothing which require only some overlap between adjacent chunks; this requirement is handled in VTK, for example, by the use of 'ghost' points within chunks.

While pipeline capabilities have advanced, both the services and the algorithms that use those services continue to be implemented using imperative languages, usually C or C++. The underlying computational model is *call-by-value* parameter-passing, yet the way to assemble applications from services is conceptually call-by-need. In contrast, non-strict functional languages such as Haskell [14, 9] use a call-by-need evaluation strategy in which function arguments are only evaluated to the extent they are demanded (if at all). Apart from closely matching the pipeline model, this strategy also provides a 'new kind of glue' [10] for assembling programs from components.

3 Marching Cubes, Functionally

Without giving a full tutorial on Haskell, we need to introduce some key aspects of functional languages, for which we use the classic Marching Cubes algorithm as an exemplar. We first implement it in the standard fashion, iterating through an array of sample values, then refine the implementation into two lazily streaming variations. These illustrate two of the main benefits of laziness – on-demand processing (permitting fine-grained pipelining of input and output data), and automatic sharing of already-computed results.

3.1 Ordinary, array-based algorithm.

First, we explore a straightforward representation of the dataset as a three-dimensional array of sample values.

```
type          XYZ       = (Int,Int,Int)
type Num a => Dataset a = Array XYZ a
```

These `type` definitions declare synonyms for the actual array representation. Concrete type names are capitalised, for instance the `Array` index domain type is `XYZ`. The array is 0-based; its first element is at index `(0,0,0)`. The type variable (lower-case `a`) in the range of the array indicates that the type of the samples themselves is generic (polymorphic). The predicate `Num a` constrains the polymorphism: samples must have arithmetic operations defined over them. Thus, we can reuse the algorithm with bytes, signed words, floats, complex numbers, and so on, without change.

```
isosurface ::
  Num a => a -> Dataset a -> [Triangle]
```

This type declaration of the Marching Cubes `isosurface` function shows that it takes two arguments, a threshold value and the dataset, and computes from them a sequence of triangles approximating the surface. The triangles can be fed directly into e.g. OpenGL for rendering. The full pipeline shown in Figure 1 can be written:[2]

```
pipeline t = mapper view
             . normalize
             . isosurface t
             . reader
```

Here the dot `.` operator means pipelined composition of functions. The last function in the chain is applied to some input (a filename), and its results are fed back to the previous function, whose results are fed back, and so on. The backward direction is just convention – it is equally easy to write forward-composition in the style of unix shell pipes, just less common.

Now to the algorithm itself. We assume the classic table, either hard-coded or generated by the Haskell compiler from some specification. Full details of these tables are not vital to the presentation and are omitted; see [19] for example.

```
mcCaseTable = { 0   |-> []
              , 1   |-> [0,8,3]
              , 3   |-> [1,8,3,9,8,1]
                ...
              , 254 |-> [0,3,8]
              , 255 |-> []
              }
```

Marching Cubes iterates through the dataset from the origin. At every cell it considers whether each of the eight vertices is below or above the threshold, treating this 8-tuple of Booleans as a byte-index into the case table. Having selected from the table which edges have the surface passing through them, we then interpolate the position of the cut point on each edge, and group these points into threes as triangles, adding in the absolute position of the cell on the underlying grid.

```
isosurface threshold sampleArray =
    concat [ mcube threshold lookup (i,j,k)
           | k <- [1 .. ksz-1]
           , j <- [1 .. jsz-1]
           , i <- [1 .. isz-1] ]
  where
    (isz,jsz,ksz) = rangeSize sampleArray
    lookup xyz    = eightFrom sampleArray xyz
```

In Haskell, application of a function to arguments is by juxtaposition – no parentheses are needed – so in the definition of `isosurface`, the arguments are `threshold` and `sampleArray`. The standard array function `rangeSize` extracts the maximum coordinates of the grid.

[1] Since VTK5.0, there is the capability to associate different executives with specific parts of the pipeline.

[2] Our Haskell implementation is actually built directly on the HOpenGL binding, so the mapping phase is implemented slightly differently, via a function that is invoked as the GL display callback. This is the *only* place where the presentation departs from the executable implementation.

The larger expression in square brackets is a list *comprehension*[3], and denotes the sequence of all applications of the function `mcube` to some arguments, where the variables `(i,j,k)` range over (or are *drawn from*) the given enumerations. The enumerators are separated from the main expression by a vertical bar, and the evaluation order causes the final variable `i` to vary most rapidly. This detail is of interest mainly to ensure good cache behaviour, if the array is stored with x-dimension first. The comprehension can be viewed as equivalent to nested loops in imperative languages.

The result of computing `mcube` over any single cell is a sequence of triangles. These per-cube sequences are concatenated into a single global sequence, by the standard function `concat`.

Now we look more closely at the data structure representing an individual cell. For a regular cubic grid, this is just an 8-tuple of values from the full array.

```
type Cell a = (a,a,a,a,a,a,a,a)

eightFrom :: Array XYZ a -> XYZ -> Cell a
eightFrom arr (x,y,z) =
    ( arr!(x,y,z),       arr!(x+1,y,z)
    , arr!(x+1,y+1,z),   arr!(x,y+1,z)
    , arr!(x,y,z+1),     arr!(x+1,y,z+1)
    , arr!(x+1,y+1,z+1), arr!(x,y+1,z+1)
    )
```

Next, we need 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. We have seen two examples thus far: `mcube`'s second argument is the function `lookup`, but also, the composition operator `.` is just a higher-order function as well. Since this operator is the essence of the pipeline model, let's look briefly at its definition:

```
(.) :: (b->c) -> (a->b) -> a -> c
(f . g) x  = f (g x)
```

Dot takes two functions as arguments, with a third argument being the initial data. The result of applying the second function to the data is used as the argument to the first function. The type signature should help to make this clear - each type variable, `a`, `b`, and `c`, stands for any arbitrary (polymorphic) type, where for instance each occurrence of `a` must be the same, but `a` and `b` may be different. Longer chains of these compositions can be built up, as we have already seen in the earlier definition of `pipeline`.

Shortly, we will need another common higher-order function, `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.

Finally, to the definition of `mcube`:

```
mcube :: a -> (XYZ->Cell a) -> XYZ
                            -> [Triangle]
mcube th lookup (x,y,z) =
    group3 (map (interpolate th cell (x,y,z))
                (mcCaseTable ! bools))
  where
    cell  = lookup (x,y,z)
    bools = toByte (map8 (>th) cell)
```

[3]It bears similarities to Zermelo-Frankel (ZF) set comprehensions in mathematics.

```
group3 :: [a] -> [(a,a,a)]
group3 (x:y:z:ps) = (x,y,z):group3 ps
group3 []         = []
```

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 (`mcCaseTable`).

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; `group3` is used again in later steps, and hence is defined globally. The linear interpolation is standard:

```
interpolate :: Num a => a -> Cell a -> XYZ
                          -> 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'.

3.2 Observations.

The implementation outlined so far is naive in several respects: (1) The *entire dataset* is needed in memory before we can begin any processing. (2) The work of comparing a vertex to the threshold value is *repeated* eight times, once for every cell it adjoins. (3) The work of interpolating along an edge is *repeated* if we revisit the same edge again within that cell. (4) The same interpolation calculation is *repeated again* when we visit the three neighbouring cells that share the same edge. The following sections address these issues in turn.

3.3 Streaming the dataset on-demand.

The monolithic array data structure implies that the entire dataset is in memory simultaneously, yet the algorithm only ever needs a small portion of the dataset. At any one moment, a single point and 7 of its neighbours suffices, making up a unit cube of sample values. If we compare this with a typical array or file storage format for regular grids (essentially a linear sequence of samples), then the unit cube is entirely contained within a "window" of the file, corresponding to exactly one plane + line + sample of the volume. The ideal solution is to slide this window over the file, constructing one unit cube on each iteration, and dropping the previous unit cube. Figure 2 illustrates the idea.

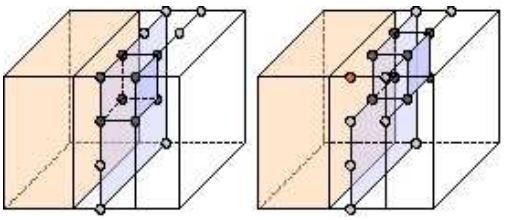

Fig. 2. Sliding a window over a grid

Haskell allows us to read data out of a file in this streamed fashion using lazy file I/O. The content of the file appears to the program as a sequence of bytes, extended on-demand one byte at a time.[4] As for dropping data after it has been consumed, Haskell is a garbage-collected language, so when a datum is no longer referenced by the computation, the memory storing it is recycled automatically.

The datatype representing the dataset is now constructed from a lazy sequence of samples, stored along with the bounds of the grid:

```
data Num a => Dataset a = D XYZ [a]
```

Unlike the `type` definition, which only introduces a synonym, a `data` definition in Haskell can be thought of as a record, tagged with a constructor name (`D`) that can be used in pattern matching.[5]

The sliding window of eight point values (cell) is extracted from the lazy stream of samples as follows. We (conceptually) lay 8 copies of the datastream side-by-side, then repeatedly slice off one value from each of the 8 and glue them together into a cell. Each of the 8 copies of the stream is advanced on its predecessor by an offset representing the distance between the sample points in the original data stream. In Haskell, the `zip` family of functions (here, `zip8`) is used to turn a tuple-of-streams into a stream-of-tuples. (Compare this `mkStream` function with the earlier point-wise `eightFrom`.)

```
mkStream :: XYZ -> [a] -> [Cell a]
mkStream (isz,jsz,ksz) origin =
    zip8 origin
         (drop 1 origin)
         (drop (line+1) origin)
         (drop line origin)
         (drop plane origin)
         (drop (plane+1) origin)
         (drop (planeline+1) origin)
         (drop planeline origin)
  where
    line      = isz
    plane     = isz * jsz
    planeline = plane + line
```

As written, `mkStream` generates 'phantom' cells that wrap around the boundaries of the dataset, at the end of each line and plane. Rather than trying to 'fix' `mkStream`, we insert another small function in the pipeline to eliminate the phantoms. The function `disContinuities` recursively copies items from its input stream to its output stream, but drops items whenever the counter reaches a boundary.

```
disContinuities :: XYZ -> [b] -> [b]
disContinuities (isz,jsz,ksz) = step (0,0,0)
  where
  step (i,j,k) (x:xs)
    | i==(isz-1) = step (0,j+1,k) xs
    | j==(jsz-1) = step (0,0,k+1)
                        (drop (isz-1) xs)
    | k==(ksz-1) = []
    | otherwise  = x : step (i+1,j,k) xs
```

The vertical bars in this definition introduce guarded equations; given a function of the form `f a | p = e`, the call `f v` evaluates to `e` provided `v` matches the pattern `a` *and* the expression `p` evaluates to `True`. Multiple guards are tried from top to bottom until one succeeds.

[4]For efficiency, the underlying system may choose to hold variable-size buffers for the file, but crucially, that buffering can be tuned to match available resources of memory, disc, and processor.

[5]Constructor names are required because in general, a data definition may introduce several alternative constructors for a given type; for example, a type 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.

By glueing the generating and pruning processes together with functional composition, we achieve our aim of transforming a lazy stream of samples from the file into a lazy stream of cells, also extended only on demand. The top-level function for isosurfacing becomes:

```
isosurfaceS thresh (D size samples) =
    concat (zipWith2 (mcubeS thresh)
                     (cellStr samples)
                     allXYZ )
  where
    cellStr = disContinuities size . mkStream
    allXYZ  = [ (i,j,k) | k <- [1 .. ksz-1]
                        , j <- [1 .. jsz-1]
                        , i <- [1 .. isz-1] ]
```

The standard `zipWith2` higher-order function is like `map`, but its function argument is iteratively applied to *two* items, pulled simultaneously from the front of its stream arguments. The `zipWith` family thus ensures that multiple streams are consumed at the same rate.

A feature of this example is the clean separation between generating a list of cells, and dealing with discontinuities. By separating these concerns, the individual functions are simplified. As they are smaller and more generic, they present more opportunity for reuse. The apparent inefficiency of computing 'phantom' cells only to discard them in the next step is eliminated through compiler optimization [27]. Note that the new function `mcubeS` is now slightly different from the previous `mcube`. Instead of passing a lookup function as an argument, we directly pass a `cell` from the stream of incoming cells. Compare the old and new type signatures:

```
mcube  ::
  a -> (XYZ->Cell a) -> XYZ -> [Triangle]
mcubeS ::
  a -> Cell a        -> XYZ -> [Triangle]
```

3.4 Sharing the threshold calculation.

The advantage of call-by-need over call-by-name is that although the evaluation of an item might be delayed until it is needed, it is never repeated, no matter how often the value is used. If we want to share a computation between different parts of the program, we just arrange for the shared value to be constructed in one place, by one expression, rather than constructing it multiple times which leads to multiple evaluations.

In the streaming version of marching cubes presented so far, we can see that the reading of sample values from file is shared and performed only once. However, comparison against the threshold value (in `mcubeS`) is performed eight times for every sample, because on each occasion, the sample is at a different vertex position in the cell. To compute the comparison only once per sample, we just need to do the thresholding against the original byte stream, *before* it is tupled up into cells, rather than after.

```
isosurfaceT th (D size samples) =
  concat (zipWith3 (mcubeT th)
                   (cellStream samples)
                   (idxStream  samples)
                   allXYZ )
 where
  allXYZ     = ...  -- as before
  cellStream = disContinuities size . mkStream
  idxStream  =
    map toByte . cellStream . map (>th)
```

There are now *three* streams of incoming data to be consumed by `mcubeT`: the cells for interpolation, the indexes into the case table, and the co-ordinates. Note how the `idxStream` is itself built using smaller pipelines. The consumer `mcubeT` is now even simpler:

```
mcubeT :: a -> Cell a -> Byte -> XYZ
                              -> [Triangle]
mcubeT th cell index (x,y,z) =
    group3 (map (interpolate th cell (x,y,z))
                (mcCaseTable ! index))
```

3.5 Sharing the edge interpolation calculation.

Taking the notion of sharing-by-construction one step further, we now memoize the interpolation of edges. Recall that, in the result of the `mcCaseTable`, the sequence of edges through which the isosurface passes may have repeats, because the same edge belongs to more than one triangle of the approximated surface.

But in general, an edge that is incident on the isosurface is also common to four separate cells, and we would like to share the interpolation calculation with those cells too. So, just as the threshold calculation was performed at an outer level, on the original datastream, we can do something similar here.

Instead of an 8-tuple of vertices, we build a 12-tuple of possible edges. Before looking up the case table in `mcubeI`, we cannot know which of those edges are actually incident on the surface. But that *does not matter* – we describe how to calculate the interpolation on all 12 edges, safe in the knowledge that each result will only be computed if that edge is actually *needed*!

```
type CellEdge a = (a,a,a,a,a,a,a,a,a,a,a,a)

mkCellEdges :: a -> XYZ -> [a] -> [CellEdge a]
mkCellEdges thresh (isz,jsz,ksz) stream =
    zip12 inter_x
          (drop line inter_x)
          (drop plane inter_x)
          (drop (plane+line) inter_x)
          inter_y
          (drop 1 inter_y)
          (drop plane inter_y)
          (drop (plane+1) inter_y)
          inter_z
          (drop 1 inter_z)
          (drop line inter_z)
          (drop (line+1) inter_z)
  where
    line      = isz
    plane     = isz*jsz
    offset d  = zipWith2 interpolate
                         stream
                         d
    inter_x   = offset (drop 1 stream)
    inter_y   = offset (drop line stream)
    inter_z   = offset (drop plane stream)
    interpolate v0 v1 = (thresh-v0) / (v1-v0)
```

Here, the datastream is offset in each of the x, y, and z dimensions, zipped with the original copy, and the interpolation calculated pairwise in each dimension. The three dimensions are then zipped together, taking four edges from each, to make up each cell.

```
isosurfaceI th (D size samples) =
  concat (zipWith3 mcubeI
                   (edgeStream samples)
                   (idxStream  samples)
                   allXYZ )
 where
  edgeStream = disContinuities size
                   . mkCellEdges th size
  ...  -- idxStream, allXYZ as before
```

Finally, `mcubeI` does no interpolation itself, it merely selects already-interpolated values from the `CellEdge` structure and adds the absolute grid position.

```
mcubeI :: CellEdge a -> Byte -> XYZ
                             -> [Triangle]
mcubeI edges index (x,y,z) =
    group3 (map (selectEdge edges (x,y,z))
                (mcCaseTable ! index))
```

4 Unambiguous Marching Cubes

It is well-known that in the original marching cubes, ambiguous cases can occur, and the original method has been enhanced and generalized in various ways to assure topological correctness of the result. Chernyaev [4] proposed a definitive classification of all the ambiguous cases. Lewiner et.al. [18] completed the resolution of internal ambiguities, and defined a method, 'MC33', guaranteed to yield a manifold surface with no cracks between or within cubes. Although MC33 requires a more extensive set of test/case tables than the original algorithm, changes to the top level structure of the functional implementation are surprisingly small; at the top level, we have simply added, as a fourth stream, the original cells of samples (used to resolve ambiguities), to the streams of ready-interpolated edges, case-table indices and grid positions. Compare the following with `isosurfaceT` and `isosurfaceI`:

```
isosurfaceU thresh (D size samples) =
    concat (zipWith4 (mcubeU thresh)
                     (edgeStream samples)
                     (idxStream  samples)
                     (cellStream samples)
                     allXYZ )
  where
    ...  -- cellStream etc. just as before
```

The `mcubeU` function differs from `mcubeT` in only two ways. (1) It uses a different two-stage case-table to look up the edges incident on the surface, of which the tiling stage occasionally needs the original sample `cell` to test for face-cracks. (2) The edges returned now include a distinguished marker to signal the need for tri-linear interpolation to resolve internal ambiguity. A simple auxiliary function detects the marker, or otherwise just picks the corresponding edge from the edge stream.

```
mcubeU :: a -> CellEdge a -> Byte -> Cell a
            -> XYZ -> [Triangle]
mcubeU thresh edges index cell (x,y,z) =
    group3 (map (select (x,y,z))
                cases)
  where
    cases =
      mc33tiles cell (mc33CaseTable ! index)
    select idx 12 =
      triInterpolate idx cell thresh
    select idx _  = selectEdge edges idx
```

5 Results and Evaluation

Figures 3 and 4 show the 'fuel' and 'neghip' datasets, surfaced using our functional implementation. Triangles are coloured as a function of the point within the output stream at which they arrive, and the figures thus show how the streaming implementation progresses through the dataset (compare with Figure 2). In the remainder of this section the functional approach is evaluated against three criteria: time and space performance, requirements for data streaming [16], and the issue of clarity and economy of expression.

5.1 Time and Space Profiles

Performance numbers are given for the initial array-based version, and an optimised streaming version of marching cubes written in Haskell, over a range of sizes of dataset (all taken from `volvis.org`), and compared with VTK 5's marching cubes implementation in C++. The relevant characteristics of the datasets are summarised in Table 1, where the streaming window size is calculated in bytes as one

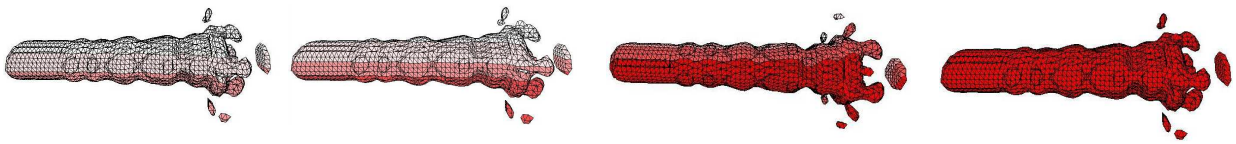

Fig. 3. Functionally surfaced dataset coloured to show age of triangles in stream.

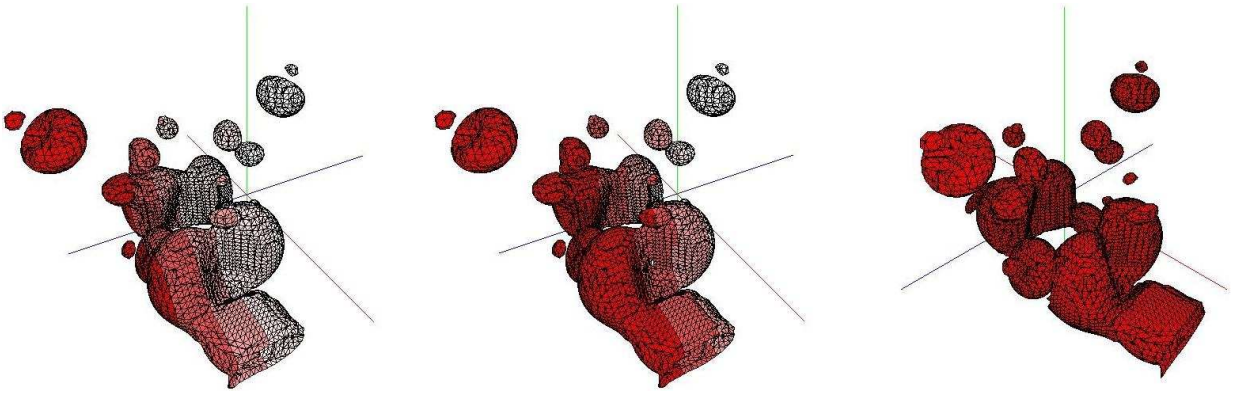

Fig. 4. Streamed dataset, colour mapped to stream position.

Table 1. Dataset Statistics.

dataset	size	input (b)	window (b)	surface (v)
silicium	(98,34,34)	113,288	3,431	103,020
neghip	(64,64,64)	262,144	4,161	131,634
hydrogen	(128,128,128)	2,097,152	16,513	134,952
lobster	(301,324,56)	5,461,344	97,826	1,373,196
engine	(256,256,128)	8,388,608	65,793	1,785,720
statueLeg	(341,341,93)	10,814,133	116,623	553,554
aneurism	(256,256,256)	16,777,216	65,793	1,098,582
skull	(256,256,256)	16,777,216	65,793	18,415,053
stent8	(512,512,174)	45,613,056	262,657	8,082,312
vertebra8	(512,512,512)	134,217,728	262,657	197,497,908

Table 2. Time Performance

	time (s)			(μs/sample)		
dataset	array	stream	VTK	arr	str	VTK
silicium	0.626	0.386	0.19	5.53	3.40	1.67
neghip	1.088	0.852	0.29	4.15	3.25	1.11
hydrogen	8.638	6.694	0.51	4.12	3.19	0.24
lobster	25.37	18.42	5.69	4.65	3.37	1.04
engine	44.51	28.06	5.29	5.31	3.35	0.63
statueLeg	48.78	34.54	2.78	4.51	3.19	0.25
aneurism	72.98	54.44	5.69	4.35	3.24	0.33
skull	79.50	57.19	79.03	4.74	3.41	4.71
stent8	287.5	154.9	33.17	6.30	3.40	0.73
vertebra8	703.0	517.1	755.0	5.24	3.85	5.62

plane+line+1, and the size of the extracted surface is measured in vertices.

Table 2 gives the absolute time taken to compute an isosurface at threshold value 20 for every dataset, on a 2.3GHz Macintosh G5 with 2Gb of memory, compiled with the ghc compiler and `-O2` optimization. We also normalise time against dataset size, giving the average number of microseconds expended per sample.

Table 3 shows the peak live memory usage of each version of the algorithm, as determined by heap profiling. Again, we normalise the absolute numbers against the size of the dataset required in memory at any one time. For the VTK and array-based versions, this is the entire dataset, whilst for the streaming version, it is the (much smaller) window size.

The time performance in Haskell (both array-based and streaming versions) scales linearly with the size of dataset. It can also be seen that the memory performance is exactly proportional to the size of the input (array) or sliding window (streaming). In contrast, the VTK results, both time and memory, are more proportionally weighted to the size of the output surface, than the input.

Although for smaller datasets, our current speeds do not compare well with VTK, the clear and readable specification of the algorithms was our main aim. Elegant Haskell is not necessarily efficient Haskell! But, due to referential transparency and equational reasoning, it is possible to apply formal transformations that preserve the code's semantics whilst improving its runtime performance. Some improvement techniques that achieve speeds within 2-3$\times$ of C code, may currently be applied manually [8, 2], whilst research continues in generalizing similar transformations to be applied automatically during the optimization stage of the compiler [27]. This approach promises (eventually) to allow elegance to co-exist with efficiency.

Even without these possible performance improvements, it is clear that the ability to stream datasets in a natural fashion makes the functional approach much more scalable to larger problem sets. For instance, the streaming Haskell version is actually faster than VTK for the larger surfaces generated by *skull* and *vertebra8*. We speculate that where other toolkits might eventually need to swap to a different out-of-core algorithm, the streaming approach will just continue to work.

5.2 Visualization Criteria

Compare the properties of our fine-grained streaming approach with the requirements for data streaming set out in [16].

Caching is achieved by coupling streamed data access (sec 3.3) with memoization (sec 3.5) of results.

Demand-Driven computation is a natural product of call-by-need evaluation (Section 3.4). Shared sub-expressions are evaluated *at most* once, e.g. the bi-linear interpolant of a given edge is computed only if needed (sec 3.4). Incoming data is transformed only to the extent that it contributes to the growing surface mesh, and grid memory that 'falls off' the back of the window is recycled automatically. By mem-

Table 3. Memory Usage

	memory (MB)			(bytes/residency)		
dataset	array	stream	VTK	array	stream	VTK
silicium	0.120	0.120	1.1	1.06	35.0	9.71
neghip	0.270	0.142	1.4	1.03	34.1	5.34
hydrogen	2.10	0.550	3.0	1.00	33.3	1.43
lobster	5.45	3.10	19.5	1.00	31.7	3.57
engine	8.25	2.10	25.4	0.98	31.9	3.03
statueLeg	11.0	3.72	15.9	1.02	31.9	1.47
aneurism	17.0	2.10	28.1	1.01	31.9	1.67
skull	17.0	2.13	185.3	1.01	32.4	11.04
stent8	46.0	8.35	119.1	1.01	31.8	2.61
vertebra8	137.0	8.35	1,300.9	1.02	31.8	9.69

oization, we extend laziness to wider sharing of computations, e.g. avoiding recomputing the edge interpolant for each neighbouring cell (sec 3.5).

Hardware architecture independence is supported in two ways. Through *polymorphic* types (sec 3.1) functions can be defined independent of the datatypes available on a specific architecture; type *predicates* (e.g. 'Num a') allow developers to set out the minimum requirements that particular types must satisfy. Beyond the scope of this paper, *polytypism* [13], also known as structural polymorphism on types, has the capability to abstract over data organisation and traversal, e.g. a polytypic marching cubes could be applied to other kinds of dataset organization like irregular grids.

Component Based development, as highlighted throughout Section 3, is fundamental to functional programming [10]. In [16], 'components' are coarse-grained modules encapsulating visualization algorithms; in this paper we have shown that component-based assembly can be much finer-grained. For example, the streaming operators would be just as applicable in the implementation of flow algorithms, generating streamlines. Functional building blocks, in the form of combinator libraries, have been developed for a range of problems, e.g. pretty-printing [11], XML transformation [29], and circuit layout [1]. The rich type systems found in functional languages aid in program development; higher-order types exactly describe how components may be safely plugged together. The assembly of functional abstractions from smaller units has a similar feeling to pipeline assembly [26], but applies across all levels of abstraction.

5.3 Clarity

In the quest for faster, more space-efficient algorithms, other qualities required of visualization systems are easily overlooked. Law et.al. briefly discuss other software design properties like robustness and extensibility. But when pictures inform us on critical issues as diverse as surgical procedure, storm forecasting and long-term decision-making linked to climate change, we should add the over-riding criterion that algorithms must be *evidently correct*.

Assurance of correctness is aided in functional programming by three means. (1) Strong static polymorphic type systems, and automatic memory management, eliminate entire classes of errors that are otherwise commonplace in imperative languages. (2) Conciseness of expression means that it is possible for a reader to understand more of the big picture at once. (This paper contains the majority of the marching cubes code – the complete program beyond the classic table is about 200 lines.) (3) As there are no 'side-effects', expressions can be manipulated using the kind of equational reasoning familiar from mathematics.

6 Related Work

While call-by-need is implicit in lazy functional languages, several efforts have explored more *ad hoc* provision of lazy evaluation in imperative implementations of visualization systems. Moran et al. [21] exploit lazy evaluation for working with large time-series datasets in a visualization system based on the *Demand Driven Visualizer* (DDV) calculator paradigm for computation and visualization of large fields. In their system derived fields quantities are evaluated either on demand (*lazy evaluation*), as a whole field (*eager evaluation*), or via a cache of lazily evaluated results (*lazy but thrifty evaluation*).

Lazy evaluation has also been used in several visualization systems. The Unsteady Flow Analysis Toolkit (UFAT) [15] allows users to compute field values on demand. Cox et.al. [5] modified UFAT to support demand-driven loading of data into main memory, achieving good performance in the visualization of large computational fluid dynamics data sets. More generally, in the demand-driven ('*pull-model*') systems noted in Section 2, e.g. VTK [26] and SCIRun [25], laziness underpins a streaming interface; modules can request just the data needed from upstreams modules within given spatial extents, and operations to produce these data are executed only on demand.

Isenburg et.al. [12] propose a streaming mesh format for polygonal meshes and discuss techniques to construct streamed meshes. In this approach mesh elements (faces and vertices) appear interleaved in the stream, and *finalization tags* record when a vertex is last referenced. *Finalized* vertices are guaranteed not to be accessed further, and can thus be removed from main memory, an *ad hoc* form of the general garbage collection techniques provided by the Haskell runtime system. Laziness is achieved by introducing vertices only when needed, but at the cost of re-organizing the entire file data-structure to generate a proper data layout in terms of vertices and faces ordering. The authors propose an application of their streaming approach to isosurface extraction techniques, and show the benefits gained from streamed inputs; but the advantage is limited to cases where the volume data changes often and only few isosurface values are evaluated.

When data are generated by computationally intense simulations and multiple isosurfaces are generated to explore the dataset, the technique of Mascarenhas et.al. [20] is more suitable. Adopting a contour following approach, their method uses sparse traversal to avoid computation at each grid location. Coupled with lazy evaluation of data based on a streamed format, performance is reported as faster than exemplar eager systems (as in [21]). As shown by Chandrasekaran et.al. [3] a lazy approach improves the throughput of an application: from a user's point of view, first results are returned quickly; from the application point of view, often only a portion of the result set is actually needed.

7 Conclusion

The purely functional streaming implementation of marching cubes developed in this paper demonstrates significant space savings compared with an approach based on monolithic datasets. This is not in itself surprising, given prior work on streaming, but shows that elegance and clarity need not be sacrificed to meet certain performance criteria. Streaming within visualization occupies an important niche between fully in-core and fully out-of-core methods. A feature of the functional approach is that data is pulled off-disk as needed, without the programmer resorting to explicit control over buffering. Data is retained in memory only as long as required: in our case a sample is held while $plane + line + 1$ cells are processed, and discarded automatically.

Motivated by the demands of large-scale data, visualization researchers have explored techniques for lazy and demand-driven evaluation. But deployment of these techniques has been limited by the need to access these services from within an imperative programming system. We have shown how a programming technology based fundamentally on lazy evaluation allows the use of streaming, call-by-need, and memoization at a fine level of granularity. Functional forms for traversal and computation can be reused across different algorithms; in surface extraction for example, the sliding window used here is applicable to sweep-based seed-set generation. We are currently exploring use of generic programming techniques to extend the work to other types of grid (e.g. tetrahedral meshes and unstructured datasets), and to other surface extraction methods. The result should be a small set of combinators from which specific traversal and surfacing tools can be constructed.

Source Material: All programs used in the paper are available from: `http://hackage.haskell.org/trac/PolyFunViz/`

ACKNOWLEDGEMENTS

The work reported in this paper was funded by the UK Engineering and Physical Sciences Research Council.

REFERENCES

[1] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[2] M. Chakravarty and G. Keller. An approach to fast arrays in haskell. In *Advanced Functional Programming 2002*, number 2638 in Lecture notes in Computer Science, pages 27–58. Springer-Verlag, 2003.

[3] S. Chandrasekaran and M. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[4] E. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical Report Technical Report CERN CN 95-17, CERN, 1995.

[5] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization '97*, pages 235–ff. IEEE Computer Society Press, 1997.

[6] R. Haber and D. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990.

[7] P. Haeberli. ConMan: a visual programming language for interactive graphics. In *Proceedings of SIGGRAPH'88*, pages 103–111. ACM Press, 1988.

[8] P. H. Hartel et. al. Benchmarking implementations of functional languages with pseudoknot, a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–656, 1996.

[9] Haskell: A purely functional language. http://www.haskell.org, Last visited 27-03-2006.

[10] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989. See also http://www.cs.chalmers.se/ rjmh/Papers/whyfp.html.

[11] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer., editors, *Advanced Functional Programming*, volume 925. Springer Verlag, 1995.

[12] M. Isenburg and P. Lindstrom. Streaming meshes. In *Proceedings of Visualization'05*, page 30, 2005.

[13] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129, pages 68–114. Springer-Verlag, 1996.

[14] S. P. Jones, editor. *Haskell'98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[15] D. Lane. UFAT: a particle tracer for time-dependent flow fields. In *Proceedings of Visualization '94*, pages 257–264. IEEE Computer Society Press, 1994.

[16] C. Law, W. Schroeder, K. Martin, and J. Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *Proceedings of Visualization '99*, pages 225–232. IEEE Computer Society Press, 1999.

[17] M. Levoy. Spreadsheets for images. *Computer Graphics*, 28(Proceedings of SIGGRAPH'94):139–146, 1994.

[18] T. Lewiner, H. Lopes, A. Vieira, and G. Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[19] W. Lorensen and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of SIGGRAPH'87*, pages 163–169. ACM Press, 1987.

[20] A. Mascarenhas, M. Isenburg, V. Pascucci, and J. Snoeyink. Encoding volumetric grids for streaming isosurface extraction. In *3DPVT*, pages 665–672, 2004.

[21] P. Moran and C. Henze. Large field visualization with demand-driven calculation. In *Proceedings of Visualization'99*, pages 27–33. IEEE Computer Society Press, 1999.

[22] D. Nadeau. Volume scene graphs. In *Proceedings of the Symposium on Volume Visualization*, pages 49–56. ACM Press, 2000.

[23] C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI'00: Proceedings of Advanced Visual Interfaces*, pages 128–135. ACM Press, 2000.

[24] A. Pang and K. Smith. Spray rendering: visualization using smart particles. In G. Nielson and R. Bergeron, editors, *Proceedings of Visualization'93*, pages 283–290. IEEE Computer Society Press, 1993.

[25] S. Parker, D. Weinstein, and C. Johnson. The SCIRun computational steering software system. In *Modern software tools for scientific computing*, pages 5–44. Birkhauser Boston Inc., 1997.

[26] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, second edition, 1998.

[27] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proc. of International Conference on Functional Programming*, pages 124–132. ACM Press, 2002.

[28] C. Upson, T. Faulhaber Jr, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, 1989.

[29] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 148–159. ACM Press, 1999.