The Lazy Z-Buffer

Steve Hill

September 13, 1994

Abstract

This paper describes a new perspective on a fundamental algorithm of three-dimensional computer graphics, namely z-buffering. An implementation of the z-buffer method in a lazy functional language behaves in a quite different manner to the traditional imperative counterpart. The main result of this paper is to show that the lazy z-buffer is a scan-line method. The effective difference between scan-line methods and z-buffers is one of order of evaluation. A hybrid algorithm with properties common to both z-buffer and scan-line methods is also described.

1 Introduction

In section 2, we describe the implementation of a simple z-buffer algorithm in a functional language [8]. We then (section 3) examine its behaviour under eager and lazy evaluation schemes. In this section we briefly describe some of the background to the work. Readers familair with computer graphics can skip Section 1.1, and those familiar with lazy evaluation can omit Section 1.2.

1.1 Hidden Surface Elimination

Hidden surface elimination is a form of sorting [7]. For example, one approach to hidden surface elimination is to sort all the objects in a scene according to their depth, and then to draw them back to front. Unfortunately this method will not work in all cases. The problem is that the primitives of most graphics systems cannot always be ordered - they are in general incomparable. In order to avoid ambiguous situations, many hidden surface methods adopt a divide and conquer strategy. If two primitives are not comparable, then they are divided into smaller objects, and the resulting objects are tested. A popular example of this approach is described by Warnock [4].

The z-buffer algorithm takes this division to the extreme. In parallel with the normal rasterisation calculations, depth values are calculated and shading calculations performed [10]. In addition to a frame buffer, a *depth* or *z-buffer* is maintained. If a newly rasterised pixel is further away than the value currently in the z-buffer, then it is discarded. If, on the other hand, it is nearer, then both the frame buffer and z-buffer must be updated with new values. One of the benefits of the z-buffer method is that it is possible to take advantage of the coherence of objects during the rasterisation process. Its memory requirement, though large, is determined by the size of the image, and not by its complexity.

There are two main disadvantages to the technique. The z-buffer is large. If depths are represented to a reasonable accuracy (say a single precision floating point number), then the z-buffer will be larger than the frame buffer. The second problem is that redundant shading calculations are often performed. When a given pixel is overwritten by a pixel from a nearer object, the calculations for the colour of the original pixel have been wasted. When a simple shading method is employed, for example [2], the overhead is not great. However, other methods require a significant amount of calculation. For example, the method of [6] requires the interpolation and renormalisation of a vector followed by the appropriate illumination calculations which might typically involve at least one trigonometric function and several dot products. Systems that provide shading languages [9] can lead to extremely expensive calculations.

There is another class of related algorithms called *scan-line* methods which are reviewed in [1]. In these algorithms, scan conversion of all primitives is executed in parallel one pixel at a time, hence there is no large z-buffer to maintain. All the polygons in a scene must therefore be stored. Typically a table is maintained detailing on which scan-line a particular polygon starts and ends. A number of optimisations make use of the spatial coherence of polygons to speed up processing. Only the shading calculations of the topmost polygon need be performed. It is also possible to avoid depth calculations, especially where there are no polygon intersections. These methods become impractical if the number of polygons in a scene is large. Scan-line methods also require all polygons to be defined before hidden surface processing can begin whereas a z-buffer method can be applied incrementally.

1.2 Lazy Functional Programming

Non-strict functional programming languages such as Miranda [8], Haskell [3] and Gofer [5], exhibit a property known as *laziness*. Expressions are always evaluated in a demand-driven fashion. At the top level, the demand for results is driven by the system's printing function. Thus, when the result of an expression, say $2+3$ is requested, the printing function demands a number which forces the evaluation of the addition. The printing function can now convert the number into its textual representation and display it. In a graphics system the printing function is replaced by a function which demands the colour of every pixel on the screen.

As an example of laziness, suppose a function is defined thus:

```
f x y = x
```

an attempt to evaluate the expression[1]

```
f 1 huge_calculation
```

where `huge_calculation` represents something computationally intensive, will obtain the result `1`. Moreover, because the argument `y` of `f` is not required to determine this result, the `huge_calculation` is never evaluated.

This notion of laziness can be extended to data structures. Suppose the first element of a pair of data items (written `(a, b)`) is examined via the function `fst` defined as:

```
fst (a,b) = a
```

Note that `fst` is a function which has a pair as argument and which returns the first element of the pair. It is not a function of two arguments as is `f` above. Evaluation of the term:

```
fst (1, big_calculation)
```

forces only the parts of the expression needed to determine the result, hence the `big_calculation` is not evaluated.

2 The Algorithm

In this section, we present a functional account of the z-buffer algorithm. The algorithm appears little different from the traditional approach. It is the manner in which it is evaluated that is significant. The implementation is based on the following type:

```
zbuff == array2 (num, colour)

array2 * == [[*]]
```

To explain – a z-buffer is represented as a two dimensional array (list of lists) of depth/colour pairs. The representation of colour is not important for the purposes of this paper, but would probably consist of a triple of stimulus values such as RGB or HSV [1].

The z-buffer method can be described by the following composition:

```
final_image :: [poly] -> [[colour]]

final_image = colours . scan_convert initialzb
```

where `colours` projects the colour information from the z-buffer and is defined as:

[1]Notice that function application is denoted simply by juxtaposition.

```
colours :: array2 (num, colour) -> array2 colour

colours = map2 snd

map2 :: (* -> **) -> [[*]] -> [[**]]

map2 = map . map
```

The value initial_zb is the initial value of the z-buffer, and can be defined as:

```
initial_zb :: zbuff

initial_zb = rep height (rep width (infinity, black))
```

In a realistic implementation, the raster width and height, background colour (here black) and maximum depth (here infinity) would probably be parameters.

The conversion of a list of primitives into an image is performed by the function scan_convert. It processes each primitive in turn producing an updated z-buffer which is then used in processing the next primitive.

```
scan_convert zb [] = zb
scan_convert zb (p:ps)
    = scan_convert (scprim 0 p zb) ps
```

The function scprim is responsible for processing each primitive. It processes the polygon scan-line at a time. There are two cases – the current scan-line is either within or outside the bounds of the shape. The precise details of the representation of primitives is not important here. The following functions are provided to inspect and modify them:

- within_y, within_x – test if the current scan line, or current pixel position lie within the primitive respectively.
- step_y, step_x – update the polygon for the next scan-line or next pixel respectively. In particular, this will involve updating the depth of the polygon.
- depth – returns the depth of the primitive
- shade – returns the colour of the primitive

```
scprim n p (r:rs)
    = scline 0 p row :
      scprim (n+1) (step_y p) rows, if within_y p n
    = r : scprim (n+1) p rows     , otherwise
```

Each scan-line, is handled by the following function:

```
scline n p (p:ps)
    = scpix p pix :
      scline (n+1) (step_x p) pixs, if within_x p n
    = p : scline (n+1) p pixs     , otherwise
```

The function `scpix`is the core of the z-buffer method. The depth of the current pixel in the polygon is calculated and compared with the depth already held in the z-buffer for this position. If the current polygon is nearer, then its depth is placed in the result and a colour for the polygon at this point is calculated.

```
scpix p (d, c)
    = (dp, shade p), if dp < d
    = (d, c)       , otherwise
      where
      dp = depth p
```

As an aside, it is worth noting that an alternative version of the algorithm can be formulated. It provides for more separation of the concerns of control and calculation. A z-buffer is represented by:

```
zbuff == array2 [(num, colour)]
```

Each cell now holds a list of depth/colour pairs. The final image can now be calculated as:

```
final_image = colours . nearest . scan_convert initialzb
```

This version also simplifies the definition of `initialzb` which is now:

```
initialzb = rep height (rep width [])
```

The only other change to the program is in `scpix` which becomes:

```
scpix p l
    = (depth p, shade p) : l
```

This approach is more versatile, since criteria for the selection of pixels may be imposed from outside by substituting different versions of the `nearest` function.

3 Behaviour

The behaviour of the algorithm we have described, when executed under lazy evaluation, is quite surprising. It has the following properties:

- only those shading calculations which are required for the final image are performed.

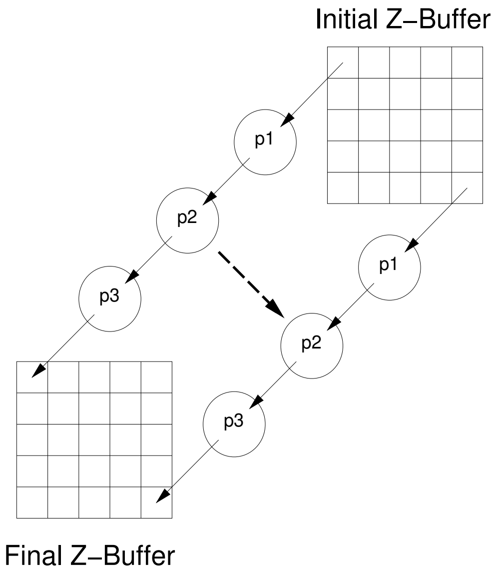

Figure 1: Lazy Z-Buffer

- memory requirement is constant and proportional to the number of primitives. Caveat – this is true only when coherency is not used in shading calculations; see Section 4.

These are not the properties of a z-buffer algorithm, rather they are typical of a scan-line method. Why is this? The answer lays in the way in which the z-buffer is calculated. A partial evaluation of an image is given by:

```
final_image [p1, .. , pn]
  = colours (scan_convert initialzb [p1, .. pn])
  = colours (scprim 0 pn ... (scprim 0 p1 initialzb) ...)
```

Demand for the colour of a pixel propagates down this expression to the `initialzb`. An initial pixel value is passed to the first primitive which calculates its colour and depth at the current position. Note that the colour is not actually evaluated since it has not yet been demanded. The depth is needed since it is compared with the previous depth value. The updated depth/colour pair is passed to the next primitive, and so-on until finally the nearest pair is passed to the `colours` function which strips the depth information and passes the shade to the printer. Only at this point is the shade calculated. In figure 1 the evaluation is depicted as a pipeline of processes. The entire pipeline proceeds from one pixel to the next.

Under eager evaluation things are very different. The properties of the algorithm become:

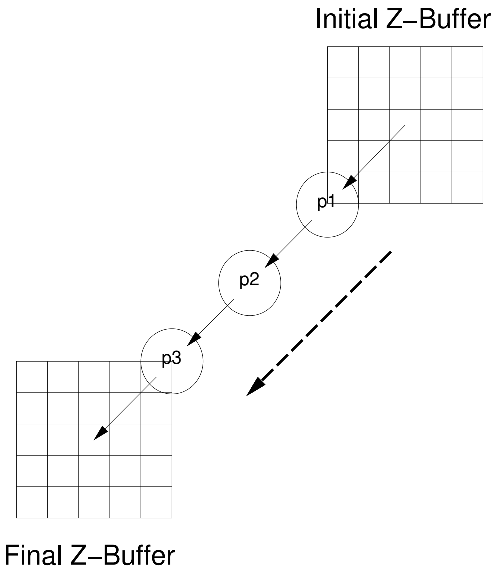

Figure 2: Eager Z-Buffer

- memory requirement is constant and proportional to the size of the image
- the number of shading calculations is usually greater than the number of visible, shaded pixels in the final image.

These features are typical of a traditional z-buffer algorithm. The evaluation proceeds by evaluating the `initialzb` which is passed to the first primitive. This is rendered to create an updated version of the z-buffer which is in turn passed to the next primitive and so-on. Finally the `colours` function projects the colour information from the z-buffer. As in the lazy scheme, the algorithm can be described as a pipeline, see figure 2. However, in this case the entire z-buffer is processed at each stage before being sent to the next.

We conclude that the essential difference between a scan-line and traditional z-buffer method is merely that of order of evaluation. Laziness leads to a scan-line method, and eagerness to a traditional z-buffer.

Many optimisations to scan-line methods employing a number of coherences are described in the literature [1]. The lazily evaluated z-buffer described here cannot compete with these methods. They take advantage of a global knowledge unavailable to each process in the lazy z-buffer. For example it is not possible, without major upheaval, to avoid depth calculations.

4 Variations

It is worth examining in more detail the behaviour of the lazy version of the z-buffer algorithm. When scan-converting a polygon it is usually the case that the shade of a particular pixel is related to the shade of its neighbour. This coherence is exploited to accelerate the shading calculations. If this technique is used in the lazy z-buffer, the effect is to increase the storage requirements. Each primitive scan-conversion process may retain a chain of unevaluated shading calculations which are only forced when a pixel is found to be visible. Alternatively, they may be garbage-collected if none of the polygon is visible. The size of the suspended calculation is likely to be proportional to the perimeter of the polygon, so could represent a fairly large storage overhead, particularly if there are many primitives.

The calculations could be performed eagerly, but this would risk performing many more calculations than are actually required. The risk of not performing them is that memory will be needlessly exhausted. If the evaluator were able to recognise low memory situations and convert to an eager evaluation method then it might be possible to get the best of both worlds. For this to work effectively, the programmer would have to annotate expressions that are candidates for eager evaluation in such situations. Ordinary strictness annotations can introduce non-termination, but in this case the program would have exhausted its memory space and would have terminated with an error anyway. The annotations would serve merely as an escape route for the desperate evaluator.

The use of coherence in shading calculation is intended to accelerate calculations. However, in situations where only a few pixels of a polygon are visible in the final image, using coherence may in fact be performing more calculation than would a direct shading calculation. In this case, the length of the chain of suspensions will be a measure of the complexity of the calculations required. If the relative cost of the direct and incremental calculations was known, then an evaluator could switch between the two equivalent formulations according to the problem size. However, a programmer would need a detailed knowledge of the performance characteristics of the underlying architecture in order to make the most of this approach.

It is possible, by mixing modes of evaluation, to devise hybrid versions of the z-buffer algorithm. For example, evaluating the spines of the z-buffer data structure eagerly gives a two-dimensional structure with the elements unevaluated. In this scheme, shading calculations will be suspended until they are required (if at all), but scan-conversion can proceed as soon as a primitive is available. When scan-converted, the primitive will construct a suspension of the shading calculation required at each pixel of the object. Figure 3 shows a representation of how the suspensions might look after two primitives have been processed. This hybrid method has the following properties:

- only those shading calculations which are required for the final image are

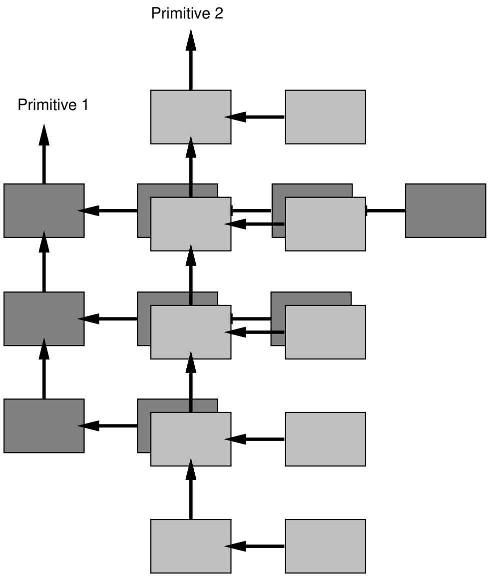

Figure 3: Suspended Shading Calculations

performed.

- memory requirement is large, being proportional to the sum of the size of the image and the perimeter of the primitives.

In this scheme, the processes of scan-conversion and shading calculation have been de-coupled. In the traditional method the two proceed in parallel. There is scope in this scheme to change evaluation mode here also. When memory becomes low (and given the memory usage of this method this is quite likely), eagerly evaluate all shading calculations to obtain a fully evaluated z-buffer. The method can then revert to lazy evaluation. The effect is that the method defers shading calculations for as long as possible within its memory constraints, but if necessary they are performed prematurely. This entails the risk that some calculations may be performed unnecessarily.

5 Conclusions

Most accounts for hidden surface removal present z-buffer and scan-line methods as quite different. When viewed from the functional perspective, it is plain that they are but two extreme aspects of a single algorithm. Since the task they perform is identical, perhaps this should not be such a surprise. Evaluation schemes that mix laziness and eagerness can be used to produce a hybrid version of the z-buffer algorithm which may be of use where shading calculations are expensive, but a scan-line method is not applicable.

References

[1] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 1990.

[2] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6):623–628, June 1971.

[3] P. Hudak, S. Peyton Jones, and P.L Wadler (editors). Report on the functional programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[4] Warnock J. A hidden-surface algorithm for computer generated half-tone pictures. Technical Report TR 4-15, University of Utah, Computer Science Dept., 1969. NTIS AD-743 671.

[5] Mark P. Jones. *Introduction to Gofer 2.20*, 1991. Available via ftp from *nebula.cs.yale.edu*.

[6] Phong Bui-Tuong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6), June 1975.

[7] I.E. Sutherland, R.F. Sproul, and Schumacker R.A. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, March 1974.

[8] D. A. Turner. An overview of Miranda. *SIGPLAN Notices*, December 1986.

[9] S. Upstill. *The RenderMan Companion*. Addison Wesley, 1990. RenderMan is a registered trademark of Pixar.

[10] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques*. Addison Wesley, 1992.