An equivalence class algorithm for drawing autostereograms

Harold Thimbleby
Middlesex University
Bounds Green Road
London, N11 2NQ
Email: harold@mdx.ac.uk

July 13, 1995

Abstract

We discuss and show correct an algorithm that constructs autostereograms. The concern is to develop a correct algorithm that may be implemented in a language such as C or Pascal.

Keywords: Autostereogram, program correctness, equivalence class algorithm.

Introduction

Autostereograms are pictures that look pretty, but meaningless until they are viewed in the right way when they reveal a three-dimensional picture. Viewing autostereograms is very satisfying and becomes easier with practice. The same is true of programming, and the purpose of this paper is to discuss how to program autostereograms correctly.

Stereograms, pairs of pictures viewed with binocular type viewers, were invented in the last century. Controversy brewed over how we see in three dimensions, and the discovery of stereoscopic vision clarified different types of depth vision; we can 'see' depth in purely

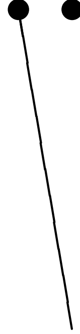

Figure 1: **View the figure holding the page almost flat and at a distance so that the left eye looks directly down one line, and the right eye along the other line. You will see a 'pin' protruding vertically through the sheet.**

flat pictures when there are appropriate cues — perspective, shading, occlusion, etc. — but stereoscopic vision requires different images being presented to each eye. When the eyes see images as if constructed from separate viewpoints, a three-dimensional percept may be obtained. Figure 1 shows a very elegant three-dimensional illusion based on these principles [1].

In the 1960s Bela Julesz discovered that satisfactory stereoscopic images can be constructed from a pair of random dot patterns, so-called *random dot stereograms* or RDSs [2]. His experiments showed that stereopsis can be achieved without any monocular (single eye) cues, such as the geometric contours that are present in Figure 1. A very simple RDS is illustrated in Figure 2. In reconstructing a three dimensional image from a stereogram the human visual system works from the images formed in each eye to inferring a surface that implies them, a process termed *fusion*. Neural net algorithms have been proposed for simulating fusion [3].

As the dot pattern is arbitrary in an random dot stereogram there is no reason why a horizontal correlation should not be introduced. Thus Tyler and Clarke [4] were led to *single image random dot stereograms*, SIRDS, otherwise known as autostereograms. Here, the pair of random dot stereograms are super-imposed but laterally shifted. Thus, an autostereogram can be made from two overlapping rectangular RDS images, ⎕⎕⎕. The lefthand rectangle is the left eye's image and the righthand rectangle is the right eye's image. The middle region of overlap is shared by both eyes, and the pixels to the right of

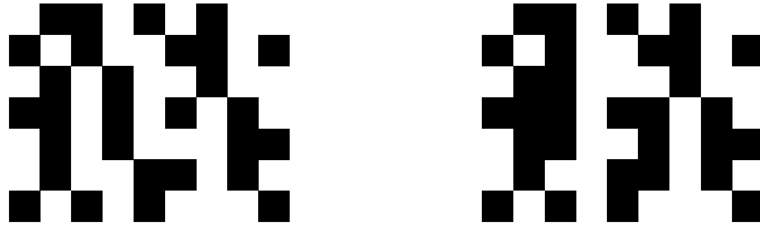

Figure 2: **Simple random dot stereogram. The central 3×3 square is left-shifted in the righthand image. If viewed stereoscopically it appears as a square 'floating' above a background. Notice that noise is introduced (in this case, to the right of the floating square) in a region that in the three dimensional case would have been hidden from view to the left eye.**

the lefthand rectangle must be equal to the pixels to the left of the righthand rectangle.

To see the combined image requires no special viewing equipment. It follows from the description that to view autostereograms your eyes must be directed one to each rectangle, so that the two components of the picture appear to overlap as a single rectangle, thus making a single binocular visual image. This may be done by deconverging the eyes (so the left eye views the left rectangle and the right eye views the corresponding part of the right rectangle) or alternatively by crossing the eyes (so that the right eye views the left rectangle, and the left eye the corresponding part of the right rectangle). Of course, each eye will see a surplus part of the image lying outside the region of complete overlap. This is of no concern when viewing the three-dimensional image, which is necessarily away from the sides of the autostereogram.[1] The main difficulty of viewing autostereograms is that your eyes normally focus at the distance where they are converged; with autostereograms, however, it is necessary to focus on the autostereogram yet converge at another distance.

An autostereogram can be made arbitrarily wide, simply by widening the region of overlap (even providing a substantial 3D scene many metres wide, which could not be viewed in its entirety from any one position). Otherwise autostereograms have no special advantages except to make stereograms more convenient but at the risk of introducing certain depth ambiguities.

[1]Compare Figure 1 where *three* lines are seen: the central three dimensional image as well as each eye's 'spare' line.

If a coloured texture is used instead of random black-and-white patterns, attractive visual effects can be achieved independently of the stereoscopic information, especially when the textures are suitably aligned with the three dimensional image. Such *colour field autostereograms* are like good sculpture: from a distance, they look attractive, and from close up, the image can be 'felt' as the eye beholds the three dimensional folds. Figure 4 shows an example autostereogram.

The computer science in this is to obtain a correct algorithm for drawing autostereograms. References [5, 6] can be recommended for their graphical presentation and for other background information on autostereograms; reference [7] provides detailed discussion of the geometrical calculations.

Problem statement

An autostereogram can be represented as a rectangle composed of pixels. If we assume it is to be viewed with the head upright, then the eyes converge at points along horizontal lines where the autostereogram intersects planes through both eyes. Hence horizontal rows of pixels in the autostereogram are independent of each other, and we need only consider the problem of drawing one row at a time. As described above, an autostereogram can be considered as a pair of overlaid but shifted random dot stereograms. If the pixels in some row are $p_1, p_2 \ldots p_N$, then the constraints of overlaying amount to requiring that $p_a = p_b$ for various pairs of a and b. Figure 3 illustrates how such constraints arise. That is, if we choose p_a to be a white pixel for the left eye's image, then p_b must be a white pixel for the right eye's. A sequence of pairs of a and b are generated by sampling a three dimensional surface along horizontal rows and by performing the necessary geometrical calculations; the distances between each pair (a, b) is a simple matter of geometry.

The standard algorithm (see reference [8] for an early description) for autostereograms works by copying, effectively as follows. First, assign random colours to all pixels. Then for each pair $(a, b), a < b$, copy the colour of p_a to p_b, thus making p_a and p_b the same colour. This copying process proceeds left-to-right across the image.

Unfortunately a problem arises: Suppose we have initially that p_a is black and p_{a+1}

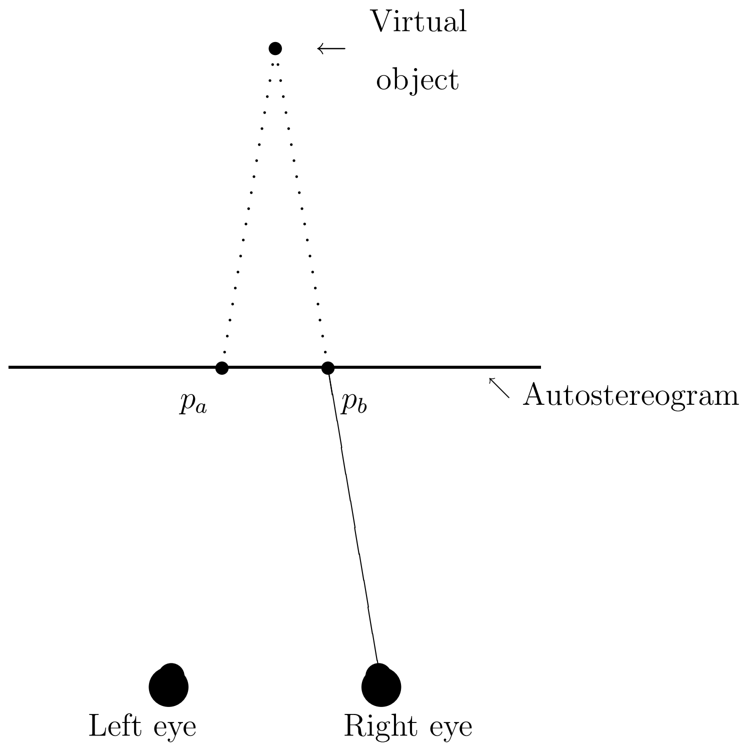

Figure 3: **How constraints arise: viewed from above, and showing lines of sight extended through the (opaque) autostereogram. Each object point is imaged by two suitably separated pixels, p_a and p_b with equal colour. (When viewing cross-eyed the image lies in front of the autostereogram rather than behind, as shown.)**

is white. The first constraint processed may be (a, b), thus p_b is made black. But the next constraint may be $(a+1, b)$, thus p_b is now made white. In general the constraints may be such that conflict cannot be detected so easily. Conflicts can be avoided, however, if we ensure that p_b advances across the image, so that no two constraints ever refer to the same right hand pixel; but this requires the eye-line (Figure 3) through p_b to be perpendicular (or at some other fixed angle) to the autostereogram. Although this solves the conflict problem, it has unfortunate consequences. The geometry is not symmetric, so to view the autostereogram correctly the eyes' convergence must be asymmetric: this causes asymmetric parallax errors. Worse the angular difference (the *disparity*) for a given autostereogram depth range is increased, making the image harder to fuse and see. Thirdly there are many more cases where a line of sight (to the non-perpendicular eye) is obscured by closer points on the object — consider an autostereogram imaging a $\cap$ shaped object: a

sharply angled eye may not see inside it at all. Although standard z-buffer or ray tracing algorithms are easily used to cater for correct points of view, the more points of the image that are not on lines of sight for both eyes, the less data remains in the autostereogram to create an accurate 3D image.

Software — Practice & Experience

Autostereograms make a persuasive case for formal methods because they still look good even when not completely accurate. Indeed the algorithm we derive here was first implemented in C, but at some stage we made a clerical slip in its implementation, yet the output nevertheless still looked correct. The slip was uncovered and corrected as a result of undertaking a proof (described below). The algorithm now published (here and in reference [7]) is correct, but we literally would not have spotted any problems but for the attempt to prove it correct.

One algorithms expert dismissed the slip in our earlier program as a "minor problem," clearly showing no interest in its correctness, a surely dangerous attitude if we were doing something more critical than autostereograms. This view of algorithms, glossing correctness, is unfortunately widespread. The algorithms textbook [9] gives an algorithm for our problem (see below), and is explicit: it uses pseudocode to be expressive at the expense of software engineering issues, such as error handling. As if to prove the point, its first algorithm (p3) is incorrect (certainly, it is not obvious how it is intended to be correctly implemented).

Our first formal justification of our algorithm involved messy inductive arguments on trees; however when trying to prove an algorithm correct you don't want to have a proof as uncertain as the algorithm! In fact, we had unwittingly accepted the invitation of Dijkstra [10] to attempt the proof without the advantages of some nicely defined properties (which we achieve below by introducing an equivalence closure operator, much as suggested by Dijkstra). Nevertheless to discuss various optimisations we will need to introduce trees: proving an algorithm correct is usefully a different activity to expressing it efficiently in a particular language such as C.

The practical software engineering perspective is that a job must be done, and done correctly; you don't simply want possibly wrong results faster, or, worse, wrong results that you do not know are wrong. You want your code to be correct — which means much more than merely copying an 'abstract algorithm' written in pseudocode, however good that algorithm may be in principle. Moreover, the way you prove your implemented code correct has to be simple and not introduce its own doubts and problems.

A number of techniques and notations, Z and VDM being well-known examples, for proving code correct and proceeding to correct implementation have been developed. The elementary mathematics used in this paper is a prerequisite for using any such formal method, and anyone familiar with, say, VDM, will see at once how to adapt and no doubt improve the presentation here. The converse is not true: anyone not familiar with VDM will not understand work done in it, and a simplified VDM proof that was understandable would have no advantages for producing reliable programs. There is obviously a balance between being formal, being understandable, and being correct; the aim of 'formal methods' is not formality for its own sake, but more certain reasoning about programs.

This paper, then, emphasises the correct and understandably correct implementation of an autostereogram algorithm, intended for a Pascal or C-like language.

Equivalent problem statements

Interestingly the algorithm required for autostereograms also appears in several guises and has been widely studied.

Perhaps the simplest approach is to imagine the pixels $p_1, p_2, p_3 \ldots$ as a set of variables that can take on values depending on their colour. The geometric constraints then represent a system of equations, $p_a - p_b = 0$ and so on. Such sets of equations can be solved by elimination and substitution, or by other numerical methods. (One problem of the standard algorithm is easy to express in this formulation: it corresponds to attempting simple substitution without first making the equations triangular.)

An alternative conception is as follows. Pixels can be placed in sets. Initially each pixel is assigned a different set. As each geometric constraint is processed we find the two

sets containing the pixels and union them, thus indicating that henceforth all pixels in the combined set should be the same colour. Finally, assign specific colours to the sets and draw the row of pixels across the picture, selecting colours according to each pixel's set membership. Expressed in this way, generating autostereograms amounts to a sequence of so-called union/find problems.

Another view of the problem is that it involves finding connected components in a graph. If we take each pixel to be a vertex of a graph, then each geometric constraint is an edge drawn between two vertices. After processing every geometric constraint, various sets of vertices will be joined by edges, but not connected to other sets of vertices. These are the connected components of the graph. Having found them, colours can be allocated and then the pixels displayed according to the colour assigned to the connected component they find themselves in.

Another view is that the geometric constraints form an equivalence relation. It is easy to see that 'the same colour as' is an equivalence relation: it is reflexive because everything is the same colour as itself; it is symmetric because if a is the same colour as b, then b is the same colour as a; and it is transitive because if a is the same colour as b, and b is the same colour as c, then a is the same colour as c. Thus we can say that each constraint (a, b) requires that pixel a is equivalent to — the same colour as — pixel b.

The trivial equivalence relation that puts every pixel equivalent to every other would satisfy the geometric constraints, but would not convey any visual information: every pixel would be the same colour and every one equivalent to every other! Instead, we must find the *smallest* equivalence classes that are consistent with the constraints. After establishing the classes, they are coloured and the pixels drawn accordingly. Because we look for the smallest, least restrictive, equivalence relation, we have the greatest freedom possible to colour the pixels, perhaps to create artistic patterns as well. (This paper briefly discusses these possibilities.)

The equivalence class problem is discussed by Knuth [11, p. 353]; more recent presentations can be found in [9, ch. 22] and [10, ch. 23], with a review of algorithms in [12]. Dijkstra [10] takes a formal approach, and his derivation and algorithm may be compared with ours, though it differs in details. Jones [13] uses equivalence classes as a case study

to illustrate VDM.

Formal problem statement

All of the above formulations of the problem are equivalent. The equivalence relation form is notationally the most helpful for our purposes, and therefore the one we shall use. In fact, the development of our proof occured in the opposite order: we had already written a program, which we wanted to prove correct, and we found that equivalence relations were the most direct way of specifying its behaviour.

We wish to find the smallest equivalence classes given as data pairs (a, b) members of an equivalence relation we shall call R, but which is not given explicitly.

When the equivalence classes are suitably coloured (e.g., randomly black or white), and the elements $a, b, \ldots$, are pixels in a scanline, then the equivalence classes constitute a solution to the autostereogram problem where the relation expresses the three dimensional geometric constraints [7]. We have N pixels on each scan line, numbered $1 \ldots N$.

The algorithm will construct a vector v so that $v.i$ is the colouring of pixel i, and for any pixels i and j supposed to be equivalent to each other according to the 3D image constraints, $v.i$ and $v.j$ will be equal.

Since the values in v are independent of the equivalence data, we shall keep them separate. Therefore the program first constructs a data structure to accommodate all data pairs from each horizontal line of the picture, then obtains v as a separate step.

Let p be the data structure to represent the equivalence relation we want. It will represent a set of pairs (i.e., the pairs in the relation); however, it need not represent all the pairs, but only sufficient to enable the equivalence relation to be extracted. This means that although p represents an equivalence relation, it need not be one itself. If we want to prove the algorithm correct, we must refer to p and what the program does to its contents, yet the mathematical properties of equivalence relations (symmetry and so forth) need not apply to p *itself*, only to what p *represents*. Let us therefore define $p^\star$ to be the smallest equivalence relation that p represents; thus, $p^\star$ is the symmetric, transitive and reflexive closure of p. If p explicitly represents just $(6, 9)$, for instance, then $p^\star$ is

$\{(6,6),(6,9),(9,6),(9,9)\}$ and all the reflexive pairs $(1,1),(2,2)\ldots\ (N,N)$. In summary, $p^\star$ is mathematically convenient, whereas p is programmatically convenient (we intend). Regardless of details of whether the program has this-or-that specific pair in p itself, how or in what order, $p^\star$ will have nice properties we can reason about uniformly.

Notice carefully that $p^\star$ is merely notation. We have not said how to find $p^\star$ from the data structure p; we have merely said that we wish to think about p in terms of $p^\star$.

As the program processes pairs of constraints (a,b) it will modify p to record them. Since we need to talk about p both *before* (a,b) is introduced and *after* it is introduced, we introduce some notation to avoid worrying about details of how this is to be done. Thus, write $p^\star(a,b)$ for the smallest equivalence relation represented by p combined with the pair (a,b). In fact, $p^\star(a,b)$ is an abbreviation for $(p^\star \cup (a,b))^\star$.

With this notation established the specification of the core of the algorithm is now easy to express:

> If the initial value of p is p_{init}, on processing a data pair (a,b) the algorithm shall modify p so that $p^\star = p^\star_{init}(a,b)$.

The question now is how to represent p so that inserting a pair and determining the values v are correct and efficient operations.

It might be nice to have a language where we can write "$p := p \cup (a,b)$" directly, but we want something suitable for a language like C. (If we implement some conventional functions to do this assignment, we merely beg the question how to represent p — algorithms to perform general set operations are longer and slower than the algorithm we will develop.) Now since p records data for a row of N pixels, and each may be in any one of the at most N equivalence classes, p can be a vector (one dimensional array). Suppose a is an element, $p.a$ can be an element equivalent to it. By further requiring $p.a \geq a$ we avoid cycles: every pixel is recorded as either the same colour as itself ($p.a = a$), or the same colour as a pixel to its right ($p.a > a$), counting conventionally, left to right.

Hence an invariant for p is $\forall i, 1 \leq i \leq N\colon p.i \geq i \wedge (i, p.i) \in R$. Every modification to p shall maintain this, though we must also require that each data pair is actually recorded in p. (The invariant as just given permits p to remain unchanged!) The requirement is

fortunately easy to express: if the pairs (a, b) so far processed constitute a set *Data*, then we require $p^\star = Data^\star$. And ensuring *that* is what we are about ... Why do we need an invariant for p anyway? The algorithm inserts a pair of data into p by relying on the invariant; if, further, it leaves the invariant property unchanged, then the next pair can be inserted the same way, and so on until all pairs are processed. We obtain a great economy: we need only prove that we can process a *single* pair of data correctly.

Many properties follow from the invariant. For example, since i and $p.i$ are equivalent (as a result of the definition above) they are interchangeable in $p^\star(p.i, j)$ and $p^\star(i, j)$, which are equal. Later when we develop the program, we can be clear that $p^\star(i, j)$ is not affected by the assignment $i := p.i$. Of course many properties follow directly from the definition of equivalence relations: by reflexivity, $p^\star(i, i) = p^\star$; and by symmetry $p^\star(i, j) = p^\star(j, i)$.

Initialisation

The invariant of p is not satisfied by everything; it must be initialised. Since R is an equivalence relation it is certainly reflexive, so p is initialised by establishing $\forall i \colon p.i = i$, which also ensures $p.i \geq i$ as required:

$$\textbf{for } i := 1 \textbf{ to } N \textbf{ do } p.i := i \textbf{ od}$$

Another way of putting this is that if there are no constraints on a pixel, it must still be the same colour as itself.

Processing pairs

The processing of a pair (a, b) must establish $p^\star = p^\star_{init}(a, b)$.

We can ignore reflexive pairs $a = b$, since $p^\star$ already represents them. We can take $a < b$, since $p^\star$ is symmetric: representing (a, b) necessarily represents (b, a) as well. So as data pairs (a, b) are processed we could discard cases where $a = b$ and swap cases where $b < a$ if we wanted to rely on $a < b$. In fact, we will rely on this, but as the geometry for our problem assures that $1 \leq a < b \leq N$, it is not necessary to check.

We introduce two variables i and j, initialised by $i, j := a, b$. This immediately establishes $p^\star(i, j) = p^\star_{init}(a, b)$ and $i < j$. The aim now is to vary i, j and p but preserving as invariant $p^\star(i, j) = p^\star_{init}(a, b) \wedge i < j$ until we obtain a state where $p^\star(i, j) = p^\star$, which implies that $p^\star = p^\star_{init}(a, b)$ as required.

Note that the *program* may not need two more variables (i, j) when it could use a and b just as easily. The reason for introducing i and j is that *we* can use them to think more clearly. The program will change i and j but not a or b; thus, at any stage in the operation of the program we can refer to the current values (i and j) and the original values (a and b) and know which is which.[2]

It may be that (a, b) is in $p^\star_{init}$ already (perhaps because we have already processed two pairs, (a, c) and (c, b)), in which case $p^\star(i, j) = p^\star$. This suggests we may manipulate i and j until we can conclude whether $p^\star(i, j) = p^\star_{init}$.

There are two interesting cases:

- Suppose we find $p.i = j$, then $p^\star(i, j) = p^\star(i, p.i)$ and this is $p^\star$. Hence $p^\star = p^\star_{init}(a, b)$.
- Suppose we find $p.i = i$, then an assignment $p.i := j$ introduces the pair (i, j) into p, but since the invariant implies $p^\star(i, j) = p^\star_{init}(a, b)$ the assignment also introduces the relation (a, b) into $p^\star$ as required. Of course the assignment destroys a reflexive pair (i, i) — that is, $(i, p.i)$ — in p but this does not affect the closure $p^\star$ because the closure necessarily includes all the reflexive pairs regardless. The assignment preserves $p.i \geq i$ since $j > i$.

Therefore we could employ a repetitive construct that terminates when $p.i = j$ or when $p.i = i$, followed by the assignment $p.i := j$ which is required if $p.i \neq j$ and has no effect if $p.i = j$. This approach would be weakly correct; that is, *if* it terminates, having preserved the invariants throughout, it achieves $p^\star = p^\star_{init}(a, b)$:

Recall that $a, b, i, j, p \leq N$, so any assignment to i from them preserves $i \leq N$. A strategy to ensure termination, then, is to ensure i increases on each iteration but by

[2] Sometimes the notation a and a' is used for similar purposes, but it is not clear when a becomes a', especially in a loop when a itself would be repeatedly changing. Moreover most programming languages do not permit this notation. Some authors prefer a and A instead, but this approach is more error-prone as both symbols have the same sound.

assignment from these values necessarily less than or equal to N. The loop must then terminate because the integer i cannot both increase indefinitely yet never become larger than N (which is fixed).

If we increase i we must also take steps, as necessary, to restore the invariants. An assignment $i := p.i$ inside the loop — given that $p.i > i$ if the loop is repeating — will increase i, but it might make it larger than j. This suggests one guarded command (A): $i < p.i < j \rightarrow i := p.i$, which increases i and keeps it less than j. If $i < j < p.i$, which is the only other possibility, then the simultaneous assignment (B) $i, j := j, p.i$ increases i and, by $j := p.i$, maintains $j > i$. (Equivalently, it does $i := p.i$, which increases i as before, but swaps i and j, so restoring $i < j$.)

A loop with these assignments terminates, since both possibilities increase i (by integers at least one) keeping it bounded by N, in fact keeping it less than j. The other invariant is easily seen to be maintained: (A) the assignment $i := p.i$ is innocuous because $p^\star(p.i, j) = p^\star(i, j)$; (B) the assignment $i, j := j, p.i$ is innocuous because $p^\star(j, p.i) = p^\star(i, j)$.

From the above considerations we can directly write down this part of the algorithm:

$i, j := a, b;$

do

$\quad i < p.i < j \rightarrow i := p.i \qquad \longleftarrow$(A)

$\quad \Box$

$\quad i < j < p.i \rightarrow i, j := j, p.i \qquad \longleftarrow$(B)

od;

$p.i := j$

Aside. The algorithm published in [14] contains an additional concurrent assignment, $p.i := j$ in case (B). This is correct but inefficient.

Serialisation and optimisation

The loop developed above repeatedly refers to $p.i$. We therefore introduce a variable pi with invariant $pi = p.i$. This makes accessing the value of $p.i$ more efficient; in fact it is such a simple optimisation many compilers would do this transformation automatically for

us. But to do it explicitly will afford us some insights into other optimisations that may be beyond the capabilities of a compiler.

The assignment (A) $i := p.i$ can be rewritten $i := pi$. The parallel assignment (B) $i, j := j, p.i$ can be rewritten $i, j := j, pi$, which has an equivalent serial form: $i := j; j := pi$. Both cases change i, so they must be followed by $pi := p.i$ to restore the invariant of pi.

The loop iterates while one guard or the other is true. Given that $i < j$ the disjunction of the guards is $i < pi \wedge pi \neq j$. This can be the condition of a conventional **while** loop that terminates under the same conditions as the original.

When $pi = j$ the loop terminates directly; otherwise the guards only distinguish $pi < j$ or $pi > j$. Since these cases are exclusive, the guarded commands can be replaced with a conventional **if** with a single test:

...

$pi := p.i$;

while $i < pi$ **and** $pi \neq j$ **do**

  if $pi < j$ **then** $i := pi$

    else $i := j; j := pi$

  fi;

  $pi := p.i$

od

...

Those minor transformations are essential if the algorithm is to be implemented in C or Pascal. We now consider more interesting optimisations.

Rearranging p but keeping $p^\star$ unchanged may make subsequent processing more efficient. To explore this idea further we must for the first time distinguish different representations of the same $p^\star$.

We can think of p as a forest of trees, each $(i, p.i)$ being an edge in a tree representing a pair in R. Each tree has a root r, where $p.r = r$, which is the greatest element of the equivalence class represented by that tree. Note that a subtree contains elements less than its root and in the same class, but it need not contain all the members of the class

less than the root; nor need an upward (i.e., increasing) path from an element include all members of the class greater or equal to it. These 'omissions' make the processing of p more efficient; they would also make reasoning about the algorithm much harder, which is why we introduced the closure $p^\star$.

Represent the largest member of a class containing i in $p^\star$ by $p \uparrow i$. (This is the root value of the tree containing i.) Since $i \le p.i \le N$, the following is an operational definition:

$$p \uparrow i = \begin{cases} i, & p.i = i \\ p \uparrow (p.i), & p.i > i. \end{cases}$$

Represent by $p \Downarrow i$ the elements in the subtree with root i. Specifically,

$$p \Downarrow i = \{i\} \cup \bigcup_{p.e=i} p \Downarrow e.$$

Now each iteration of the loop follows the linked lists from a and b to find the closest common element, possibly but not necessarily the root. On termination of the loop, j is some element in the same class and no smaller than i. Subsequent operations involving any element in $p \Downarrow a$ will necessarily run up the same path at least to the same element. We could avoid this computation by recording j in each element of $p \Downarrow i$. More easily, we can record it in just those elements of $p \Downarrow i$ that have been processed; this is an optimisation called *collapsing* [11, 15]. Collapsing increases the time of the loop by a constant factor (roughly doubling it), as is clear from the following implementation appended to the previous code:

$\{\ p^\star = p^\star_{init}(a, b)\ \}$

$i := a;$

do $p.i \ne j \rightarrow$

  $p.i := j; i := p.i$

od

Collapsing may save considerable work in subsequent merges; in fact, if any element in $p \Downarrow i$ occurs in a subsequent merge, the speed-up will recoup all the time that anyway would have been taken in reaching j! If this occurs more than once, then there are actual gains. Unfortunately it turns out for autostereogram data that this optimisation typically

saves 10% of iterations — but to break even it would have been necessary to save nearer 50% of them. A more efficient optimisation is still possible, however, for this pattern of data.

We know that $i < j$, also we know that we want to record $\max(p\uparrow i, p\uparrow j)$ to replace $p\uparrow i$. Hence we can partially collapse by performing the assignment $p.i := j$ within the loop. Only in case A would the assignment increase $p.i$, since its guard is $p.i < j$:

while $i < pi$ **and** $pi \neq j$ **do**
  if $pi < j$ **then** $p.i := j; i := pi$
    else $i := j; j := pi$
  fi;
  $pi := p.i$
od

The effect of this optimisation is to shorten the length of the path from a to $p\uparrow a$, generally at least halving it, because at each step $p.i$ is made to point further along it, reducing the number of links from $p.i$ to $p\uparrow i$. This modified loop may be followed by $p.a := j$, a worthwhile optimisation if j is increased by the loop and if any element in $p\Downarrow a$ is subsequently (likely to be) processed.

Other methods in the literature [16] include: *splitting* where each element i is made to point to $p.(p.i)$; and *halving* where every other element is made to point to $p.(p.i)$. Both methods approximately halve the length of the list, but splitting requires about twice the number of assignments.

Such optimisations are justified on the assumption that future merges will tend to examine the same parts of p repeatedly. To speed the algorithm up, however, the savings gained in the future must out-weigh the costs of the additional assignments required to modify p. It turns out for typical autostereogram data (where most paths are of length 1 or 2) that this is not the case; so the putative optimisations *increase* the running time in practice.

For autostereograms, a major part of the overhead occurs in the calculation of the data. This calculation may be optimised in various ways, for example using table lookup but the

details, though quite routine, are beyond the scope of this paper.

Colouring the equivalence classes

> And thus the native hue of resolution
> Is sicklied o'er with the pale cast of thought
> (William Shakespeare. *Hamlet*, Act III, Scene I.)

Having constructed p, we now need to choose colours for every pixel on each line of the autostereogram consistently with the the equivalence classes calculated for that line. It is useful to distinguish the colour palette allocated (for artistic effects) from the colour combinations required (to make the autostereogram work). Our approach, then, will be to construct a vector v such that $v.i$ is the colour index of pixel i. Every pixel with the same colour index will be coloured the same, but at this point we do not need to decide whether this colour is, say, red or pink. If the colour index of pixel i is $v.i$, then we require $v.i = v.j$ whenever $(i, j) \in p^\star$. Unfortunately p does not yield this directly; we will have to derive v from p.

Interestingly we have already specified a suitable valuation, namely $v.i = p \uparrow i$, which is easily seen to have the desired properties, and which gives v index values from $1 \ldots N$. Call this intended state of affairs $\mathbf{Q}$, that $\forall i, 1 \le i \le N : v.i = p \uparrow i$. Obtaining $\mathbf{Q}$ directly would be magic, but we can get there in easy stages.

Consider $\mathbf{Q}(t): \forall i, t \le i \le N : v.i = p \uparrow i$: that only part of the vector v, the elements $v.t$ to $v.N$, is correct. This predicate $\mathbf{Q}(t)$ has the relevant properties that $\mathbf{Q}(N+1)$ is trivially true (by magic:-) and $\mathbf{Q}(1)$ is $\mathbf{Q}$. The crucial observation is that given $\mathbf{Q}(t+1)$, we can establish $\mathbf{Q}(t)$ by simple operations. What is required, therefore, is a loop taking t in steps of 1 down from N to 1, and maintaining $\mathbf{Q}(t)$ throughout. Such a loop terminates with $t = 1$ and $\mathbf{Q}(1)$, and hence $\mathbf{Q}$ established as required:

```
{ Q(N+1) }
for t := N down to 1 do
    { Q(t+1) assumed }
        ⋮
    { Q(t) established }
od
{ Q(1) established }
```

At the start of the code inside this loop $\mathbf{Q}(t+1)$ is true. Now consider $p.t$; there are just two cases $p.t > t$ and $p.t = t$. In the first case $v.(p.t) = p \uparrow t$ because $\mathbf{Q}(t+1)$ implies $v.(p.t) = p \uparrow (p.t)$ and $p \uparrow (p.t) = p \uparrow t$. In the second case $p.t = p \uparrow t$ by definition of $\uparrow$. Hence we arrive at the following program:

```
for t := N down to 1 do
    if
        p.t = t → v.t := p.t          ←——(C)
        □
        p.t > t → v.t := v.(p.t)
    fi
od
```

The assignment (C) colours the equivalence classes recorded in v with a numerical label unique for each class. The label is arbitrary; we could use it to index into a palette to chose a colour. Or if the assignment was $v.t := \text{Random}(\{\text{Black}, \text{White}\})$ instead we would obtain simple black and white random dot autostereograms.

As for serialisation, since $p.t \geq t$, if the first guard $p.t = t$ is false then $p.t > t$ is implied and the other guard must be true; we can use a conventional **if** form for the guarded conditional. Appendix 1 shows both modifications.

Further applications of equivalence classes

Since t decreases and the loop only refers to p and v with subscripts larger or equal to t, we would be safe using p as an alias for v. This halves the memory requirements. Moreover, when $p.t = t$, $p.t$ and $v.(p.t)$ would be the same thing, so the body of the loop can become very simply $p.t := p.(p.t)$ without any conditions. Interestingly this is an application of the equivalence class algorithm itself: in a language such as FORTRAN we could declare to the compiler that p and v are `equivalent`, permitting it to allocate minimal memory. (This optimisation destroys the invariants of p; fortunately we only ever need v after all data has been processed in p, so no conflict arises in practice.)

Generating colour field autostereograms

Figure 4 gives an example picture drawn using the autostereogram algorithm. The pixel colours have been chosen from a texture instead of a random distribution with no spatial pattern, thus creating a colour field autostereogram. A coloured autostereogram generated by the same algorithm has been published elsewhere [17].

A simple way to construct a colour field autostereogram is to use a colour palette, c, so pixel i is coloured $c.(v.i)$. The colour values in c are taken from the corresponding line in the given texture. Now, pixels with the same v value generally lie in many differently coloured parts of the texture, so the painter's algorithm [18] can be used do determine an appropriate single palette colour for all of them. Using the painter's algorithm, assign colours to the palette from the texture in the least important parts first (e.g., the edges) finishing with the most important part (e.g., the centre) last. The colours taken from near the centre of the texture then override previous allocations to the palette from the edges. Finally, draw the pixels using the palette to map their index to the appropriate texture colours.

Given v as above, where $v.i$ is a unique index for all pixels equivalent to pixel i, we now need to create the palette c with $c.(v.i)$ the specific colour for pixel i. The colours are taken from the texture and 'painted' onto the palette from the outside inwards along the row, painting from both sides inwards, using t and u working inwards, with a loop that

terminates when they intercept, when $t = \lfloor N/2 \rfloor + 1$. Some entries in the palette may be repeatedly painted, but they remain set to colours taken closer to the centre of the texture.

$$
\begin{array}{l}
\textbf{for } t := N \textbf{ down to } \lfloor N/2 \rfloor + 1 \textbf{ do} \\
\quad c.(v.t) := \text{Texture}(t, y); \\
\quad u := N + 1 - t; \\
\quad c.(v.u) := \text{Texture}(u, y) \\
\textbf{od}
\end{array}
$$

Here we assume that Texture(u, y) gives us the colour of point (u, y), point u on row y, of the intended texture field. Finally, simply display the colours taken from the palette:

$$
\begin{array}{l}
\textbf{for } t := 1 \textbf{ to } N \textbf{ do} \\
\quad \text{Display}(t, y, c.(v.t)) \\
\textbf{od}
\end{array}
$$

This algorithm minimises distortion of the texture in the centre of the picture, because the painter's algorithm renders that area last. By reversing the direction of the painter's algorithm loop, distortion can be minimised at the edges instead, which may be desirable for æsthetic reasons. Indeed, by choosing an arbitrary point, and rendering the pattern towards that point (which may be chosen differently on each scan line of the picture), the distortion can be finely controlled.

Generally some fine tuning of the texture and 3D image may be necessary to achieve satisfactory æsthetic results. Figure 4, for example, was based on a tiled texture made by scaling a pattern to width equal to the separation of equal pixels at the maximum imaged distance.

Further work

The successful combination of a good three dimensional image with a pleasing texture is a creative artistic effort. As pointed out earlier, though, an autostereogram with certain errors can still produce a satisfactory three dimensional image.

It follows that the quality of a texture may be improved by judiciously introducing errors. For example, geometry may require that many pixels in a row are the same colour, but ignoring one geometric constraint may enable upto half those pixels to assume a different, better texture colour. Thus, the texture might be improved considerably. Moreover, half the pixels, being a different colour, cannot then be fused with the other half, so the three dimensional image (less the omitted point) becomes easier to see. How should the 'errors' be chosen?

Conclusions

This paper has discussed an application of the equivalence class problem to autostereograms. Our concern was to prove the algorithm correct, which we did by introducing the notation $p^\star$ to describe the operation of the algorithm at the level of equivalence classes, without concern for details of representation. To proceed from the correct algorithm to its expression in a conventional programming language required more detailed examination of the representation. (This may be an argument for programming languages that do not require unnecessary serialisation — a source of possible programming errors.) However, we found that the more concrete analysis gave us a function that had direct use in a later stage of the algorithm itself. Finally, the algorithm is pleasingly self-referential: its memory requirements can be almost halved by a simple application of equivalence classes themselves.

The general points this paper made are well known in principle, but deserve repeating:

- Incorrect programs may appear to be correct. Programs for autostereograms are an example of this.
- Obtaining a correct algorithm is easier when one is able to reason with convenient properties, here equivalence relations.
- Transforming an algorithm into a correct program to run in a particular language is separate from proving the algorithm correct. Efficiency considerations often lead to

other insights, here they led directly to the approach used for colouring the equivalence classes.

- Some published algorithms are abstract, and it may still be difficult to obtain a correct program from them.

Acknowledgements

The impetus for this work arose through collaboration with Ian Witten and Stuart Inglis on a visit to Waikato University Department of Computer Science in 1993. The author is extremely grateful for the University supporting his visit. The anonymous referees made several helpful comments.

References

[1] James, W. (1890) *The Principles of Psychology*, **2**, p94, Henry Holt & Co.

[2] Julesz, B. (1971) *Foundations of Cyclopean Perception*, University of Chicago Press.

[3] Marr, D. & Poggio, T. (1979) "A Computational Theory of Human Stereo Vision," *Proceedings of the Royal Society of London*, **B204**, pp. 304–328.

[4] Tyler, C.W. & Clarke, M.B. (1990) "The Autostereogram," *SPIE Stereoscopic Displays and Applications*, **1258**, pp. 182–196.

[5] Horibuchi, S. ed. (1994) *Stereogram*, Boxtree Ltd.

[6] Richardson, D. (1994) *Create stereograms on your PC*, Waite Group Press.

[7] Thimbleby, H. W., Inglis, S. & Witten, I. H. (1994) "Displaying 3D Images: Algorithms for Single Image Random Dot Stereograms," *IEEE Computer*, **27**(10), pp38–48.

[8] Stork, D. G. & Rocca, C. (1989) "Software for Generating Auto-random-dot Stereograms," *Behaviour Research Methods, Instruments & Computers*, **5**, pp. 525–534.

[9] Corman, T. H., Leiserson, C. E. & Rivest, R. L. (1992) *Introduction to Algorithms*, MIT Press.

[10] Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice-Hall.

[11] Knuth, D. E. (1973) *The Art of Computer Programming*, 2nd. ed., Addison-Wesley.

[12] Galil, Z. & Italiano, G. F. (1991) "Data Structures and Algorithms for Disjoint Set Union Problems," *ACM Computing Surveys*, **23**(3), pp319–344.

[13] Jones, C. B. (1990) *Systematic Software Development using VDM*, 2nd. ed., Prentice Hall International.

[14] Thimbleby, H. W. & Neesham, C. (1993) "How to Play Tricks with Dots," *New Scientist*, **140**(1894), pp. 26–29.

[15] Tarjan, R. E. (1975) "Efficiency of a Good but not Linear Set Union Algorithm," *Journal of the ACM*, **22**, pp. 215–225.

[16] Tarjan, R. E. & van Leeuwen, J. (1984) "Worst-case Analysis of Set Union Algorithms," *Journal of the ACM*, **31**(2), pp. 245–281.

[17] Thimbleby, H. W. (1993, 13–19 November) "Extra: Science," *TV Times*, **151**(46), pp. 82–83.

[18] Newman, W. M. & Sproull, R. F. (1979) *Principles of Interactive Computer Graphics*, 2nd. ed., McGraw-Hill.

Appendix 1. Putting the code together

```
for i := 1 to N do                                    ←—(initialise p)
    p.i := i
od;
while given (i, j) do                                 ←—(insert pairs into p)
    pi := p.i;
    while pi < i and pi ≠ j do
        if pi < j then i := pi
            else i := j; j := pi
        fi;
        pi := p.i
    od;
    p.i := j
od;
for t := N down to 1 do                               ←—(construct v)
    if p.t = t then v.t := Random({Black, White})
        else v.t := v.(p.t)
    fi
od
```

Note. This code generates a single scan line for a random black/white dot autostereogram. The function *given* scans the 3D image and yields true for each line while it assigns values to i and $j, 1 \leq i < j \leq N$. Its full definition is given in reference [7]. If $z_{x,y}$ is a depth coordinate of a visible point on the object normalised to $(0, 1)$, then *given* assigns $i := x - s, j := x + s$ where $s = E\frac{1-\mu z_{x,y}}{2-\mu z_{x,y}}$, E is the separation of the viewer's eyes (measured in pixels) and μ is a geometric constant appropriate for the viewing conditions, typically 0·3.

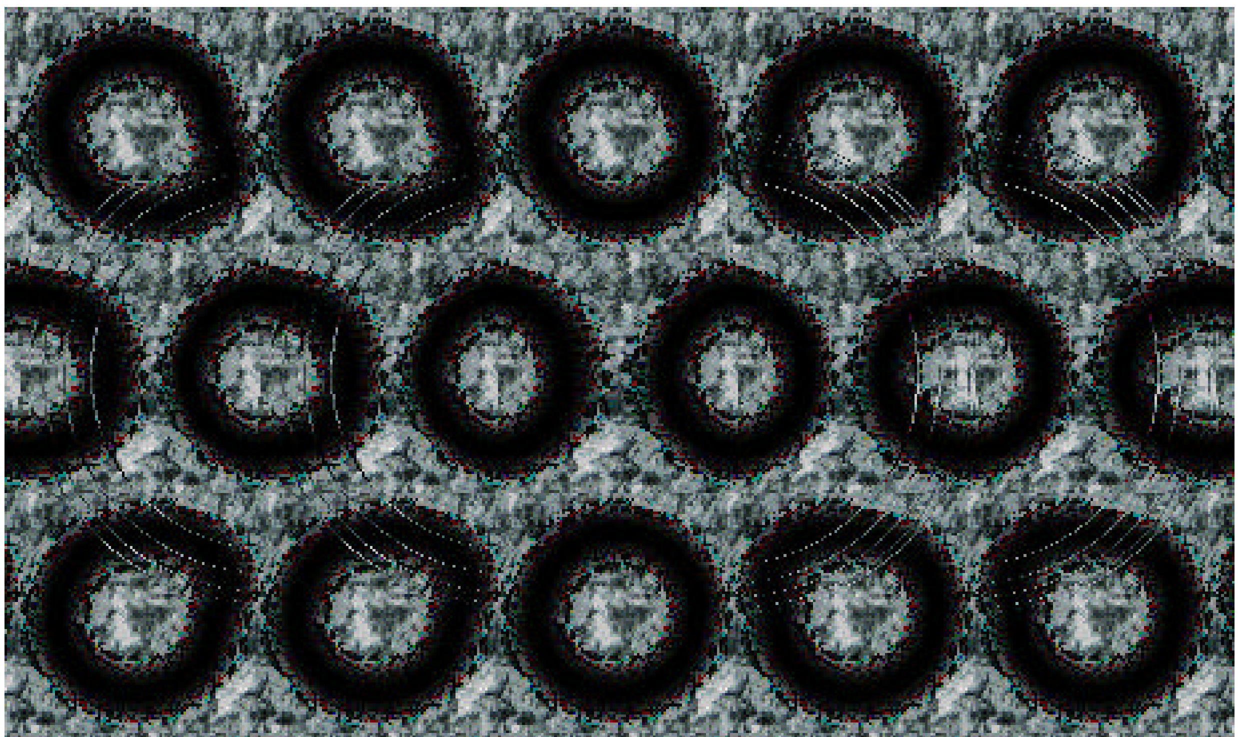

Figure 4: **A simple autostereogram of a ring. See Appendix 2 for viewing instructions.**

Appendix 2. Viewing autostereograms

The geometry of Figure 4 has been chosen so that when correct eye convergence occurs, not only is the three-dimensional image visible, but your eyes are converging at a distance twice the distance to the autostereogram. Therefore, if you have difficulty viewing it, you can try placing a sheet of glass (or other semi-reflective sheet) over the autostereogram. Then converge your eyes on your reflection, which is also at twice the distance from your eyes as the autostereogram. This is an easy way to achieve correct convergence, though you initially focus too far into the distance. Short sighted people may find it easier to view autostereograms if they remove their glasses, as this facilitates focusing (provided they are more-or-less equally short sighted in each eye).

Another method is to photocopy the autostereogram onto a transparent sheet. Then try looking through the sheet at a blank wall.

Although designed to be geometrically correct for divergent viewing, the autostereogram shown in Figure 4 may also be viewed by cross-eyed viewing, though the 3D image will then be inverted and have minor depth distortions. To facilitate cross-eyed viewing, hold a pen (or other pointer) in front of the autostereogram and move it nearer or closer to the autostereogram so that the tip of the pen is pointing (via each eye) at two adjacent similar parts of the picture — you may find it easier to mark suitable points first, though some autostereograms have separate 'guide spots' for the same purpose. Now look at the pen; you will see a single pen and the two parts of the autostereogram will fall on corresponding parts of your retinas. All you have to do now is get the autostereogram in focus whilst remaining cross-eyed. Long sighted people may find it easier to view autostereograms using this method if they remove their glasses.

Bright light helps view autstereograms as this causes your pupils to contract, and eases focusing. In any method, it is essential to hold the autostereogram upright with respect to the head; that is, any plane through both eyes must intercept the autostereogram along lines of pattern repetition in the autostereogram. Note that some people are incapable of seeing autostereograms.

Programmers have more options. A rapidly animated series of autostereograms of the same geometric model, but using different textures, has no stable monocular image, and

hence is easier to view binocularly. Secondly, when pairs of constraints are processed, the left pixel can be labelled green and the right red. Then, when the row of pixels is drawn, only luminance is taken from the texture, and hue from the labelling (using black for a pixel labelled both red and green, and white for neither). The autostereogram may now *also* be viewed as a conventional red/green anaglyph with the aid of appropriate glasses. Once the 3D image is seen with glasses, they can be removed, and the 3D image from the autostereogram will remain!