Yugoslav Journal of Operations Research
10 (2000), Number 2, 217-234

OPTIMIZATION OF THE RASTER-SCAN CIRCLE-DRAWING ALGORITHM BASED ON PREDICATE TRANSFORMERS

Nedeljko OSTOJI]
Yugoslav Military Academy,
Belgrade, Yugoslavia

Du{an STAR^EVI]
Faculty of Organizational Sciences, University of Belgrade,
Belgrade, Yugoslavia

Abstract: An approach to the optimization of raster-scan circle-drawing algorithms is presented. The approach is based on the program transformation theory, which is a subset of Dijkstra's weakest precondition calculus. In this way, care of the correctness of the algorithm is incorporated in the optimization process. The authors propose that a designer and an implementor define the problem together, translating quality requirements and technology constraints into algorithm requirements. Then the standard solution should be explained and optimized in terms of weakest precondition. Thanks to the approach with an invariant and a bound function, it is possible to separate aspects concerning optimization and correctness. The method is illustrated by the optimization of an integer case circle algorithm.

Keywords: Algorithms, raster graphics, circle drawing.

1. INTRODUCTION

There are several approaches to circle drawing in raster. Although there are solutions based on parallel computational models, [9, 19], as well as those based on neural networks [13], it seems that today commercial raster devices mostly use sequential models to implement basic raster algorithms.

At the same time, there are different criteria as to whether a certain pixel belongs to a given circle or not. The most popular is Bresenham's criterion, thanks to its simplicity. However, it has some major drawbacks: the problem of so-called gaps in disk tracing [11], the problem of circle thickness, difficulties with generalization to

higher dimensions, etc. A novel approach based on the discrete analytical definition of a circle [1] eliminates these drawbacks, but it gives much slower solutions.

Algorithm speed is so important that most commercial graphical applications still include a routine for 2D circle generation in raster based on Bresenham's criterion and restricted to integer radius and centre co-ordinates. This simplification provides the eight-way symmetry of a circle. Thanks to this, the algorithm calculates the points of the circle in one octant of the plane only (e.g. 0-45 degrees). The remaining 7 octants are covered by symmetry arguments and need no further computation. At the same time, the restriction to integer operations makes the algorithm very fast.

In the last couple of years there have been several papers on straight line and circle [8, 21] acceleration algorithms based on short horizontal or vertical line segment generation. Unfortunately, it seems that till now there has been no computer system that directly produces adequate axial segments, which is a basic assumption of such, so-called "run-length slice" algorithms.

The fact that the latter approach is slower than conventional solutions when there is no means for parallel output of several pixels at a time, inspired us to turn back to conventional algorithms. Is it possible to speed them up by transforming known algorithms? Program transforms are often used in rasterization, but their correctness is usually supported on a case-by-case basis via raster semantics or sometimes intuitively. The consequence is the fear that the optimization process could destroy program reliability, which prevents authors from coming to the best solutions. Is it possible to use "code improvement techniques" without fear that they could destroy program reliability? We suggest the use of techniques based on weakest precondition calculus. In this way, we are able to check any idea keeping the algorithm correctness through the optimization process.

The approach to optimization presented herein can be used to improve a wide class of sequential circle generating algorithms. For the sake of simplicity, we illustrate the approach on conventional algorithms with restrictions to integer radius and centre co-ordinates. It is applicable to algorithms with arbitrary radius and centre co-ordinates, including those generated with a novel approach based on the discrete analytical definition of circles. It can be seen through the case study that our approach is applicable to platforms with instruction level parallelism, as well as to pipeline architectures.

In the next section we comment on the drawbacks of existing solutions. In Section 3 we introduce our approach to the optimization, and demonstrate it through the case study given in Section 4. We conclude in Section 5 with some comments and open questions.

2. EXISTING SOLUTIONS

The obvious general approach to circle drawing in raster based on computing x and y values for various angles by using trigonometric functions must be rejected as too inefficient. There exist a few efficient incremental solutions for circles with arbitrary centre and radius. The latest of them [1] is based on the discrete analytical definition of circles, which provides more properties and easier control over them. On the other hand, the solutions with arbitrary centre and radius are still less efficient when compared to those with restrictions to integer radius and centre co-ordinates: they perform several times more operations, they use several times more variables, and their source code is several times longer.

The restrictions to integer radius and centre co-ordinates provide the possibility of exploiting the eight-way symmetry of a circle. The task is usually reduced to plotting a circle defined by $x^2 + y^2 = r^2$ with its centre in origin and with radius r as a nonnegative integer. The algorithm calculates the points of the circle in one octant of the plane only (e.g. 0-45 degrees). The remaining 7 octants can easily be covered by symmetry arguments and need no further computation.

There are several kinds of sequential algorithms for drawing circles with integer radius and centre co-ordinates in raster. Yao and Rokne in [21] classified all algorithms published prior [8] as generated by the "conventional approach". "Conventional" algorithms [4, 6, 15, 17, 20] generate co-ordinates of each pixel, while so called run-length slice circle algorithms generate just the co-ordinates of the ends of horizontal line segments built of the pixels that approximate the circle (see Fig. 1).

Most efficient conventional algorithms are based on Bresenham's approach. They start on a quadrant boundary and generate pixel by pixel making either an axial or a diagonal move. The decision depends on the sign of the "decision variable". For the midpoint algorithm the decision variable is defined as:

Pdc: $d = x^2 + y^2 + 2*x - y - r^2 + 1$.

The short and nice synthesis process given in [18] leads to the solution as in Program 1:

```
void circle_bres (int r)
{   int x = 0;
    int y = r;
    int d = 1 – r;
    while (x < = y) {
        PutPixel (x, y);
        if (d < 0) d + = 3 + 2*x;
        else d + = 5 –2*((y– –) – x);
        x+ +;
    }
}
```

Program 1. A standard conventional solution

In addition to restrictions to integer radius and integer centre co-ordinates, an important characteristic of conventional algorithms is that they draw pixel by pixel. We can say that they are optimized for implementations that lack the means for parallel output of more than one pixel at a time. Other major drawbacks of conventional algorithms are: it is difficult to generalize them to higher dimensions, and it is difficult to determine and control the properties of generated primitives. Besides, if the restrictions to integer radius and centre co-ordinates are abandoned, this leads to immediate loss of symmetry. In any case, it seems that on account of efficiency most of today's commercial solutions include a routine based on this approach. Therefore, there is still interest in improving the performance of this approach that is over 30 years old.

Hsu, Chow and Liu proposed in [8] an approach recognized by Yao and Rokne in [21] as the run-length slice approach. The approach is based on the identification of the ends of horizontal line segments defined by the pixels that approximate the circle (see Fig. 1).

Two algorithms which incrementally find the ends of horizontal line segments are derived in [8]. Test results reveal that as the radius changes from 1 to 128 those algorithms are more than 1.36 times faster than the midpoint algorithm and Bresenham's circle algorithm. However, these algorithms have certain drawbacks, the most important being: calculation of the square in the loop, too many variables, and large values in some variables. The gain in speed over conventional algorithms is obtained due to the reduction in the number of output operations and not due to the reduction in the number of arithmetic operations. We can say that they are optimized for implementations that have the means for the parallel output of more than one pixel at a time. A consequence is that the proposed algorithms become less efficient than conventional algorithms when the value of the x variable becomes large sufficiently during execution of the algorithm.

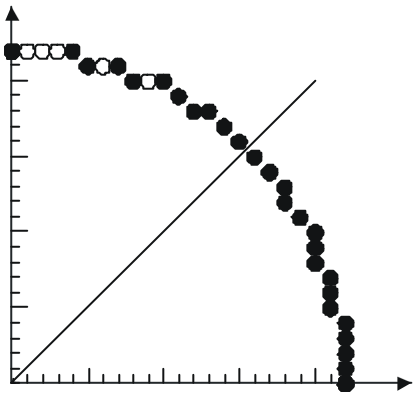

Figure 1: "Horizontal line based" algorithms determine just the co-ordinates of the ends of horizontal line segments (black points), while "conventional" algorithms determine the co-ordinates of each pixel.

Yao and Rokne in [21] improved the run-length slice approach in three ways. First, they eliminated almost all the mentioned drawbacks. Second, they speeded up the algorithms introducing larger steps. And third, they introduced a hybrid approach that uses the run-length slice circle algorithm while there are long runs of pixels, and then switches to the midpoint approach when the length of the remaining runs is at most two.

Analyzing solutions from [21] we noticed that, compared to conventional solutions, run-length slice circle algorithms still use about 50% more variables and a larger number of arithmetic operations per iteration. Besides, there is the calculation of a square and a multiplying operation when switching from the run-length slice circle algorithm to the conventional algorithm. Still, we can say that run-length slice circle algorithms are optimized for implementations that have parallel output of more than one pixel at a time. A consequence is that they become less efficient than conventional algorithms when the generation of short axial lines is not available. It seems that till now there has been no computer system that directly produces adequate axial segments.

3. AN APPROACH TO OPTIMIZATION BASED ON PREDICATE TRANSFORMERS

We suggest the use of techniques based on weakest precondition calculus so that the algorithm's correctness can be easily maintained through the optimization process. We propose that a designer and an implementor together, as the first step, consider problem definition by translating quality requirements and technology constraints into algorithm requirements. After that a recapitulation of the algorithm design is to be made, where the standard solution should be explained in terms of weakest precondition calculus. Then the implementor and the designer should try to optimize the algorithm using weakest precondition calculus, bearing in mind the requirements from the problem definition step. If possible, different solutions of the problem are to be optimized, and their efficiencies analyzed and compared.

Thanks to this approach, we can separate aspects concerning optimization and correctness. Raster space properties and program code, together with implementation dependent characteristics, give inspiration for the introduction of new predicates. In order to initialize and maintain these predicates, new mechanisms are introduced. Immediately after that, attention is paid to algorithm correctness. If any invariant has been destroyed, compensation mechanisms are introduced to restore the destroyed invariant. As a consequence, it could happened that a modified algorithm is less efficient than the original one. The careful analysis and comparison of available solutions is suggested as a separate phase.

3.1. Task specification

Different approaches exist to program specification. According to [3] they can be classified into three basic families based on: the algebraic, the state machine, and the predicate transform model. In order to use predicate transformer based techniques in optimization efficiently, we suggest using the predicate transform model for the program specification, as well.

There are also other approaches, e.g. [10], but for this purpose we accept that there are two parts of a specification. The most important part is specification of "what the execution of an algorithm is to accomplish". The other part deals with other aspects concerning speed, size and so forth.

The first part of a specification is given in the form: $\{Q\}S\{R\}$, where Q and R are predicates, and S is an algorithm, with the following interpretation: "If the execution of S is begun in a state satisfying Q, then it is guaranteed to terminate after a finite amount of time in a state satisfying R." Q is called the *precondition*, or input assertion of S; and R is called the *postcondition*, *conclusion*, output assertion or result assertion.

The other part of the algorithm specification deals with other aspects concerning speed, size and other implementation dependent characteristics. Certain quality requirements and technology characteristics (constraints and supports) which are to be satisfied and exploited are usually given to the implementor. He/she should translate these characteristics into algorithm requirements and place them into this part of the algorithm specification. This part of a specification should be the basis for implementation dependent optimization.

3.2. Recapitulation of the algorithm design using the wp predicate transformer

After the algorithm specification has been reviewed and supplemented with the part which specifies optimization requirements and possibilities, the algorithm design phase should be recapitulated by the designer and implementor together. We suggest the approach with weakest precondition calculus presented in [5]. The use of predicates and predicate transformers could be very useful when optimizing an algorithm. So, the most important thing in this step is to recognize the predicates which should hold after the execution of certain algorithm portions.

Weakest precondition is a predicate, denoted by $wp\{S, R\}$, that represents the set of all states such that the execution of S begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying R. The general properties of a particular mechanism S are given when we are able to determine $wp\{S, R\}$ for any R. The derivation of $wp\{S, R\}$ can be impractical for certain mechanisms (e.g. for loops). Usually we are not interested in $wp\{S, R\}$, but in some stronger precondition, say Q, for which we can show that $Q \Rightarrow wp\{S, R\}$ without actually forming $wp\{S, R\}$.

The approach to algorithm design based on weakest precondition is nicely presented again in [7], with a lot of explicitly stated strategies, principles, and rules; and with a number of carefully chosen and intently ordered examples. In short, algorithm design starts from R, the predicate which specifies the conclusion. We search for an appropriate mechanism S such that

$$Q \Rightarrow wp\{S, R\}.$$

Often, it is difficult to recognize S at once. Therefore, we try to make a sequence of predicates $Q_1 \ldots Q_n$ for which it is not difficult to recognize mechanisms $S_1 \ldots S_n$ such that:

$$(Q \Rightarrow wp(S_1, Q_1)) \text{ and } (Q_1 \Rightarrow wp(S_2, Q_2)) \text{ and} \ldots \text{and } (Q_{n-1} \Rightarrow wp(S_n, Q_n)) \text{ and } (Q_n \Rightarrow R).$$

In this way, n problems of algorithm synthesis are specified in the form: $\{Q_{i-1}\} S_i \{Q_i\}$. If it is still difficult to recognize S_i, then the refinement procedure is to be repeated (until each S_i is either a primitive statement, or a statement list). The final solution is concatenation: " $S_1; S_2; \ldots; S_n$ ".

While the use of mechanisms such as assignment or alternation is relatively simple, and their weakest precondition can be easily calculated, the use of repetition or recursion mechanisms is much more complicated. The approach suggests converting recursion into iteration for correctness purposes. The general structure of the simplified deterministic iterative mechanism, usually denoted by DO, is:

do BB ® S_{DO} od,

where BB is a Boolean expression called a "guard", and S_{DO} is a mechanism. While BB holds, S_{DO} is executed. Therefore, if DO terminates, it must obtain non BB.

The approach suggests designing an iteration with a clearly defined goal. It must terminate after a finite number of iterations in a state satisfying the given predicate R_{DO}. There are two important concepts concerning the iterative mechanism: the "invariant" and the "bound function". An invariant is a predicate, say P, which must not be destroyed by the DO mechanism. If P is established before DO is executed, it should hold after DO terminates, if it terminates at all. So, if DO terminates, it must obtain P and non BB. A bound function t is a finite integer function of the current state. It has two important properties:

1. P and BB $\Rightarrow$ (t > 0); and
2. each iteration decreases t at least by 1. Attention to these properties guarantees termination of the iterative mechanism.

The approach suggest designing the DO mechanism carefully choosing P such that $R_{DO} \Rightarrow P$. After that, non BB is determined so that P and non BB $\Rightarrow R_{DO}$. Therefore, if DO terminates, it should terminate in a state where R_{DO} holds. When the approach is mastered, it becomes easy to design correct loops. Using this approach,

algorithm correctness would be checked and stated in terms of weakest precondition calculus. Implementation independent optimization should also be reconsidered in this step. We suggest the use of known techniques based on weakest precondition calculus. Different solutions of an algorithm design task are to be considered. If there is only one solution, the designer and implementor should try to find another one. Some solutions, although more expensive in a way, could prove to be more efficient in certain implementations.

3.3. Optimization

A good introduction to the optimization phase is a critical look at a solution obtained through algorithm design. If there are implementation independent possibilities for optimization, they are to be recognized and exploited.

The implementor also has to recognize possibilities based on implementation dependent characteristics. We assume that the implementor is acquainted with the techniques of locating the parts of an algorithm which are of interest for possible optimization. The issue has been discussed in [2].

The problem is that the optimization process could destroy algorithm correctness. Because of that, formal proof of optimized algorithm correctness could be given after optimization. Other approaches propose the use of correct transformations of suitable segments of correct algorithms [14, 16]. What we suggest here is to develop the proof in parallel, hand-in-hand with the optimization. This has already been applied in algorithm design [5, 7, 10]. In this way weakest precondition calculus is not just a tool for proving algorithm correctness. It helps us find optimal mechanisms, and optimization could be considered the final phase of algorithm development.

The optimization techniques we use are not new. We just translate known techniques into a form based on weakest precondition calculus. From the optimization point of view, the most interesting pieces of code are those in iteration bodies. Thanks to the approach with an invariant and a bound function, we are able to separate aspects concerning optimization and correctness. Code and implementation dependent characteristics give us inspiration for the introduction of new predicates. The result is the introduction of new mechanisms in order to initialize and maintain these predicates. Immediately after that, we can pay attention to algorithm correctness. As first, we calculate the impact of new mechanisms on previously defined invariants.

Most calculations to be done are due to the impact of assignment mechanisms on the invariants. Let us consider a simple assignment to simple variable:

" $x := e$ ",

where x is a simple variable, and e is an expression of the same type as x. Assuming expression e can be properly evaluated, the semantics of the assignment mechanism is often given in the simple form:

$$wp("x := e", R) \equiv R_e^x$$

where the right side of the relation denotes the predicate obtained by simultaneously substituting e for all free occurrences of x in R (which is called textual substitution). The problems that could arise from this just given definition, as well as a more precise definition of textual substitution, are discussed in [7]. In practice, almost all calculations are reduced to textual substitution according to the form given above.

In this way, we can check if invariant predicates still hold after the execution of any assignment mechanism. If any invariant has been destroyed, we can determine compensation mechanisms and restore the destroyed invariant. As a consequence of the introduction of compensation mechanisms, it could happened that a modified algorithm is less efficient than the original. A careful analysis and comparison of available solutions is to be made as a separate phase.

3.4. Evaluation and comparison of the solutions

If there are more than one algorithm solving the same problem, they are to be compared. If we have only one solution, it is good practice to try to design another one. In any case, optimized solution efficiency should be analyzed and compared to other known solutions.

Co-operation between the designer and implementor is very important. It is possible that an algorithm design involves constraints which prevent the implementor from obtaining the best solution. Because of that, we propose that the designer and implementor work together during the optimization phase.

4. CASE STUDY: OPTIMIZATION OF THE CIRCLE GENERATING ALGORITHM

4.1. Conditions and assumptions

We have discussed the motivation for optimization of the circle generating algorithm in Section 1. For the sake of simplicity, we consider here conventional circle generation in raster restricted to integer radius and centre co-ordinates. Using the approach presented in the previous section, we have arrived at an algorithm which seems to be more efficient than the other available circle-drawing algorithms. The same approach can be used for the optimization of both run-length slice circle algorithms, and those based on the discrete analytical definition of a circle.

The task is usually defined as the problem of activating the sequence of pixels whose centres are the "nearest" to the circle defined by "$f(x, y) = x^2 + y^2 - r^2 = 0$" where r is a nonnegative integer. So, the precondition is

Q: $r \geq 0$,

and the postcondition is

R: All pixels whose centres are the "nearest" to the circle defined by $f(x, y) = x^2 + y^2 - r^2 = 0$ have been activated.

For the sake of presenting our approach, we have chosen a circle with its centre at the origin and integer radius. Pixels lie on grid lines. The algorithm has to select the same pixels as the most widely used Bresenham circle algorithm.

The part of the algorithm which has to determine the next approximation pixel may use only integer addition and/or subtraction, as well as repetition and selection mechanisms. Magnitude comparisons are to be avoided.

4.2. A view of the standard solution

Exploiting the eight-way symmetry, authors usually consider one octant of a circle. Assuming the last activated pixel has co-ordinates (x, y), the invariant relation for the first octant (counting from 0^o counterclockwise), can be stated as

P: All approximation pixels with ordinates less than y have been activated.

The job is finished when all pixels in the first octant are initialized, i.e. when

non BB: $x \leq y$

holds, which implies the guard of the iteration mechanism

BB: $x > y$.

For the first octant, the first pixel to be initialized can be $(x = r, y = 0)$. After that, only north $(x, y+1)$ and north-west $(x-1, y+1)$ pixels are available. For the integer case the choice can be made according to the sign of the sum of the function values in north and north-west pixels: $f(x, y+1) + f(x-1, y+1)$. If the value is negative the north pixel is selected, otherwise the north-west pixel is selected.

The technique of "updating the value already stored" is used to avoid the calculation of squares. The invariant relation

Pd: $d = f(x-1, y+1) + f(x, y+1) = (x-1)^2 + x^2 + 2*((y+1)^2 - r^2)$

is initialized at the point $(x = r, y = 0)$ executing the statement " $d := 3 - 2*r$ ".

The bound function that satisfies the required properties from section 3.2, can be defined as $t : x - y$. There are two progression statements which decrement function t, depending on whether the next point is to be activated by an axial move

Sa: " $y := y + 1$ ",

or a diagonal move

Sd: " $x, y := x-1, y+1$ ".

Using the weakest precondition calculus we can calculate how progression statements impact the relation Pd. Consider the axial move:

$$wp(Sa, Pd) \equiv wp("y := y+1", d = (x-1)^2 + x^2 + 2*((y+1)^2 - r^2))$$

which, after substituting all occurrences of y by $y+1$ yields

$$\equiv d = (x-1)^2 + x^2 + 2*((y+1)^2 - r^2) + 4*(y+1) + 2 .$$

From here we recognize the progress sequence

Sacd: " $y := y+1; d := d+4*y+2$ ".

which keeps Pd invariant. After similar analysis it can be recognized and proved that the sequence

Sdcd: " $x, y := x-1, y+1; d := d-4*(x-y)+2$ "

keeps Pd invariant, simultaneously making a diagonal move.

The solution in a C/C++ notation can take the form:

```
void circle (int r)
{
    int x, y, d;
    x = r; y = 0; d = 3 – 2*r;
    while x > y {
        activate_circle_points (x, y);
        y + +;
        if (d < 0)  d + = 2 + 4*y;
        else {
          x––;
          d + = 2 – 4*(x –– y);
        }
    }
    if (x = =y) activate_circle_points (x, y);
}
```

Program 2. A standard conventional solution

At this point, we should consider the drawbacks of the solution from the implementor's point of view. The algorithm body asks for shifting and four arithmetical

operations per axial move or six arithmetical operations per diagonal move. In addition, magnitude comparison is needed in the guard of the iteration mechanism.

4.3. The optimization

We have to eliminate the shifting and the magnitude comparison. Also, the number of additions/subtractions should be reduced.

Let us consider again the mechanisms Sacd and Sdcd. The idea is to eliminate the multiplication by 4 introducing a new variable, say *dd,* and a new invariant relation:

Pdd: $d = 2 * dd$

Now we have to answer the question: "how do the progression mechanisms Sacd and Sdcd impact the relation Pdd?" Using the weakest precondition calculus we obtain

$$\begin{aligned} wp(Sacd, Pdd) &\equiv wp(\text{"}y := y+1;\ d := d+4*y+2\text{"},\ d = 2*dd) \\ &\equiv wp(\text{"}y := y+1\text{"},\ d+4*y+2 = 2*dd) \\ &\equiv d+4*(y+1)+2 = 2*dd. \end{aligned}$$

So, the result of executing the mechanism Sacd is the enlargement of the left side in Pdd by $4*(y+1)+2$. To restore the truth of Pdd, the right side has to be enlarged by the same amount. This can be achieved by enlarging dd by $2*(y+1)+1$. Again, we can make use of the previously stored value in the new variable dy defined by the following invariant relation:

Pdy: $dy = 2*y+1$.

The invariant relation Pdy can be initialized executing "dy := 1" when y = 0. The progress sequence for axial move now becomes:

Sac: "y := y+1; dy := dy+2; dd := dd+dy".

After a similar analysis of wp(Sdcd, Pdd) we recognize the need to introduce another variable, say dxy, defined by the invariant relation:

Pdxy: $dxy = 2*(x-y)-1$.

The invariant relation Pdxy can be initialized executing "dxy := 2*x – 1" when y = 0. Therefore, the progress sequence for a diagonal move is:

Sdc: "x := x – 1; y := y+1; dy := dy+2; dxy := dxy – 4; dd := dd – dxy".

Keeping Pdd invariant ensures that both variables d, dd have the same sign in all states encountered in the course of computation. So, there is no need to use both of them.

We can summarise our algorithm into the following form:

```
void OstCircle (int r, int Colour)
{
      int x, y, d, dy, dxy;
      x = r;
      y = 0;
      d = 1 – r;                        // Pd has been initialised
      dy = 1;                           // Pdy has been initialised
      dxy = 2*x – 1;                    // Pdxy has been initialised
      putpixel (x, y, Colour);

      while (dxy > 0) {
      if (d < 0) {                      // select axial or diagonal move
            y++; dy += 2; dxy –= 2;                 //increment like phase
            d += dy; putpixel (x, y, Colour);       //true addition – I/O phase
      } else {
            x––; y++; dy += 2; dxy –= 4;            //increment like phase
            d –= dxy; putpixel (x, y, Colour);      //true addition – I/O phase
    }
  }
}
```

Program 3. An optimised solution

It has been noted in [12] that "an octant could not be left" by its axial move. In this particular case, the first octant could not be left by a vertical move. Hence, we make an unnecessary test of "end of octant condition" after each axial move. Notice that the **while** loop can be implemented as follows:

```
SA:   putpixel (x, y, Colour);
      dxy –= 2; dy += 2; y++;
      if ((d += dy) < 0) goto SA;
SD:   putpixel (x, y, Colour);
      if ((dxy –= 4) <= 0) goto EXIT;
      dy += 2; x– –; y++;
      if ((d –= dxy) < 0) goto SA; else goto SD;
EXIT:
```

Program 4. An implementation of the while loop without "end of octant testing" after an axial move

4.4. Efficiency considerations

Program performances are very machine specific. Although the number of operations is not sufficient for algorithm speed estimation, we used it as a rough estimation of performance. To complement this analysis, we tabulated numerical results from some tests of our solution and the Algorithm from [6], Algorithm YR3 from [21], and Algorithm AJ from [1].

Let us start with considerations regarding implementations with parallelism on the instruction level. Each iteration step of the **optimiz**ed solution produces exactly one approximation point and consists of one "increment like" phase and one "true addition" phase.

The first phase is to simultaneously increment or decrement some of the registers x, y, and d. The addition of 2 or 4 on registers dy and dxy, realized in VLSI technology lasts as an increment operation. All these operations are independent and can be performed simultaneously, consuming the same time as one increment operation.

In the second phase, one of the register-variables dy, or dxy is chosen to be added/subtracted to/from register-variable d. Simultaneously with this, the values of x, y are followed to the part for activating circle points. This is a true addition phase.

It asks for two phases in both the diagonal (outer loop) and the axial (inner loop) parts. The former is an increment like phase (changes to x, y, dx and dxy), and the latter is a true addition phase (changes to d). There is no need to compare values; just a sign test is required. In total, we need about

$$\approx r/\sqrt{2} \approx 0.7*r$$

true addition phases, and the same number of increment like phases.

The best available solution published in [6] also consists of an "increment like" phase and a "true addition" phase:

```
void MidpointCircle (int radius, int Colour)
{// Foley at all.
  int x, y, d, deltaE, deltaSE;
  x = 0;
  y = radius;
  d = 1 – radius;
  deltaE = 3;
  deltaSE = –2*radius + 5;
  putpixel (x, y, Colour);

  while (y > x) {
```

```
    if (d < 0) {            // Select E
            d + = deltaE;                                   //true addition phase
            deltaE + = 2; deltaSE + = 2; x+ +;              //increment like phase
    } else {
            d + = deltaSE;                                  //true addition phase
            deltaE + = 2; deltaSE + = 4; x+ +; y––;         //increment like phase
    }
    putpixel (x,y,Colour);
  }
}
```

Program 5. Midpoint circle algorithm from [6]

The output procedure putpixel is sequentially concatenated to the selection mechanism, while our solution executes it in parallel with the true addition operation. Although this solution uses the same number of variables, it performs the "end of octant" test in each point, i.e. more than twice times. In addition, it asks for a value comparison in the "end of octant" test. It can be noticed that the value of variable d is first tested then calculated, while our solution tests it just after calculation, having the value of the sign test in accumulator without further operations. Hence, we expect our solution to be faster than alternative algorithms in implementations with parallelism on the instruction level, as well as on pipeline architectures.

Although our algorithm is optimized for platforms with the possibility of simultaneously executing several instructions, we can compare it to other solutions under consumption that instructions are executed sequentially.

The total number of pixels in the region $x \leq y$ is approximately equal to

$$w = (\sqrt{2}/2) * r .$$

The number of diagonal moves in the same region, which is equal to the number of axial line segments, is approximately equal to

$$h = (1 - \sqrt{2}/2) * r .$$

The number of axial moves in this region is equal to the difference w - h.

Denoting by Na, Ni, Ns, and Nc the number of true additions, increments, shifts, and comparisons respectively, we can calculate the number of operations in different solutions. In the loop of our solution, we have

$$Na = h + w \approx 0.71 * r,$$

$$Ni = h + 3*w \approx 2.41 * r.$$

We can notice that there are no comparisons or shifts. The total number of arithmetic operations in our algorithm is approximately

$N = Na + Ni \approx 3.12*r.$

In the same way we calculate the number of operations in solutions from the literature. The totals are tabulated in Table 1. We can see that alternative algorithms perform 18-23% more operations than our solution. Algorithm AJ from [1] solves a more general case without any restriction, but it performs more than 15 times more operations than our solution. This is why conventional algorithms are still in use, in spite of all their drawbacks. When radius and centre co-ordinates have non-integer values there is no symmetry, and each pixel co-ordinate is calculated separately. Even if we compare the number of operations for only one octant, Algorithm AJ requires 90% more operations compared to our solution.

Table 1: Comparison of the number of operations

	Number of operations / r					
Algo-rithm	Na	Ni	Ns	Nc	N	N(Ai) : N(Ours)
Ours	.71	2.41	–	–	3.12	––
Foley	.71	2.41	–	.71	3.83	1.23
YR3	.36	.92	.21	.51	3.68	1.18
AJ	22.11	7.93	–	17.42	47.42	15.20

Table 2 gives a comparison of the best computation times of our solution and the algorithm from [6] for some specific radii. The testing is done on two hardware platforms.

Table 2: Comparison of computation time

	Pentium-S			AMD K-5		
radius	Pg4	Pg5	Pg5 : Pg4	Pg4	Pg5	Pg5 : Pg4
60	1.2	3.2	2.67	3.2	5.6	1.75
125	2.2	5.6	2.55	6.0	11.0	1.83
250	6.6	10.0	1.52	11.0	22.0	2.00
500	11.0	19.8	1.80	22.0	38.0	1.73
1000	23.0	39.6	1.72	60.0	110.0	1.83
2000	46.2	78.0	1.69	110.0	170.0	1.55

The times listed in Table 2 do not reflect real running times since no pixels are written. A function call and output operations are too expensive in so short a code. We must switch them of if we want to compare computation times. The figures in Table 2 shows that our algorithm is 52-167% faster, which is considerably better than expected from the number of operations comparison. This is because there is not much

dependence among the instructions of our algorithm. This offers better possibilities for parallel execution on the instruction level, as we commented earlier in this section.

5. CONCLUSION

We presented a disciplined approach to raster algorithm optimization based on implementation dependent characteristics. The method is based on the weakest precondition calculus. In this way, the care of algorithm correctness is taken through the optimization process. This gives more freedom in implementing the ideas for optimization.

We expect that this approach can be of help in the implementation dependent optimization phase of algorithm development. To illustrate our approach we considered the optimization of a conventional circle generating algorithm restricted to integer radius and centre co-ordinates. Using the method presented herein we have come to a solution which seems to be more efficient than the other available circle-drawing algorithms. The approach can be used for circles with arbitrary radius and centre co-ordinates, regardless of whether the circle definition is based on the conventional definition or the discrete analytical definition.

The proposed approach can prove helpful in optimizing circle generating algorithms for different target video adapters. We expect that this approach can be used to improve sequential circle generating algorithms when a computer system can make the simultaneous output of as few as two pixels. Developers of circle-based algorithms (e.g. circle brush or general arc) may also benefit from this approach. We expect this approach to influence parallel circle-drawing algorithms, as well.

There is the danger that the output from the design phase could involve such constraints that optimization could fail to achieve the best solution. The careful analysis and comparison of available solutions is suggested as a separate phase.

The question arises as to what extent this kind of the optimization could be achieved automatically by a compiler. Our approach to the matter is that too many implementors expect compiler optimizers to compensate for their own intellectual inadequacies. We are working on the specification of computer aided facilities that should help us reach better solutions on manipulating algorithms and influencing the way we think, but not completely solving problems instead of us.

REFERENCES

[1] Andres, E., and Jacob, M., "The discrete analytical hyperspheres", IEEE Transactions on Vizualization and Computer Graphics, 3 (1) (1997) 75-86.
[2] Bentley, J.L., Writing Efficient Programs, Prentice-Hall, Englewood Cliffs, N.J., 1982.
[3] Berg, H.K., Boebert, W.E., Franta, W.R., and Moher, T.G., Formal Methods of Program Verification and Specification, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.
[4] Bresenham, J.E., Algorithm for Circular Arc Generation, Springer-Verlag, 1985, 213.

[5] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.
[6] Foley, J.D., and Van Dam, A., Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, Mass., 1982, reprint with corrections, 1984.
[7] Gries, D., The Science of Programming, Berlin-Heidelberg-New York, Springer, 1981.
[8] Hsu, S.Y., Chow, L.R., and Liu, H.C., "A new approach for the generation of circles", Computer Graphics Forum, 12 (2) (1993) 105-109.
[9] Huang, J., and Banissi, E., "An improved parallel circle-drawing algorithm", IEEE Computer Graphics and Applications, 10 (5) (1997) 60-67.
[10] Jones, C.B., Software Development: A Rigorous Approach, Prentice-Hall International, Inc., London, 1980.
[11] Kulpa, Z., "On the properties of discrete circles, rings and disks", Computer Vision, Graphics and Image Processing, 10 (1979) 348-365.
[12] McIlroy, M.D., "Best approximate circles on integer grids." ACM Transactions on Graphics, 2 (4) (1983) 237-263.
[13] Meriaux, M., and Vidal, B., "Line and circle drawing on a Boolean neural network", Proceedings of Computer Aided Design and Computer Graphics (CAD & CG 89), Beijing, China, 1989, 93-98.
[14] Piteway, M.L.V., "The algebra of algorithms", in: Fundamental Algorithms of Computer Graphics, Advances in Systems and Applications, NATO ASI Series, Vol F17, Springer-Verlag, 1985, 837-853.
[15] Posch, K.C., and Fellner, W.D., "The circle-brush algorithm", ACM Transactions on Graphics, 8 (1) (1989) 1-24.
[16] Sproull, R.F., "Using program transformations to derive line-drawing algorithms", ACM Transactions on Graphics, 2 (4) (1983) 259-273.
[17] Suenaga, Y., Kamae, T., and Kobayashi, T., "A high speed algorithm for the generation of stright lines and circular arcs", IEEE Transactions on Computers, 28 (10) (1979) 728-736.
[18] Wirth, N., "Drawing lines, circles, and ellipses in a raster", in: Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra, Springer-Verlag, 1990, 428-434.
[19] Wright, W.E., "Parallelization of Bresenham's line and circle algorithms", IEEE Computer Graphics and Applications, 10 (5) (1990) 60-67.
[20] Wu, X., and Rokne, J.G., "Double-step incremental generation of lines and circles", Computer Vision, Graphics and Image Processing, 37 (1987) 331-344.
[21] Yao, C., and Rokne, J.G., "Hybrid scan-conversion of circles." IEEE Transactions on Visualization and Computer Graphics, 1 (4) (1995) 311-318.