Anthony Errington

Interactive Teaching of Computer Graphics Algorithms

Part II Computer Science Tripos

Churchill College

May 12, 2006

Proforma

Name:	**Anthony Errington**
College:	**Churchill College**
Project Title:	**Interactive Teaching of Computer Graphics Algorithms**
Examination:	**Part II in Computer Science, 2006**
Word Count:	**Approx 11900**
Project Originator:	**Anthony Errington**
Supervisor:	**Dr Neil Dodgson**

Original Aims of the Project

To write an interactive and easy-to-use teaching tool to aid students in their understanding of computer graphics algorithms. The tutor was to include multiple algorithms in a single interactive environment with tools for parameter alteration and visual demonstrations.

Work Completed

The tool has been completed and the results are presented in this dissertation.

Special Difficulties

None encountered.

Declaration

I, Anthony Errington of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Overview

My project concerns the creation of an interactive tool for teaching computer graphics algorithms. I have successfully implemented such a tool covering a number of graphics algorithms. These include those laid out in my original project proposal as well as several extensions.

1.2 Motivation

Computer graphics algorithms are intrinsically visual. Seeing an algorithm in action is a useful and engaging way of understanding its behaviour and methodology, as is the ability to vary parameters and observe the algorithm in an interactive environment. Graphics algorithms are ideal candidates for interactive, visual, animated teaching methods.

Conversely, traditional, static methods of teaching are ill-suited to demonstrating computer graphics algorithms. The need to dynamically display intermediate stages of the algorithm's behaviour is difficult with pen or paper, and fiddly with an electronic presentation. Thus the motivation for an interactive computer graphics teaching tool is strong.

1.3 Related Work

The relevance of e-learning has not been doubted since computers began to emerge as a teaching tool, but it is the advent of the internet which has taken things to higher levels. Engaging, interactive visual content can be distributed easily without leaving a web browser thanks to Java applets and tools like Macromedia Flash.

The particular relevance of e-learning to the field of computer graphics has been well documented and work has been done in the area [4, 6, 7]. However an internet search for existing interactive computer graphics tutorials brings up surprisingly few results. A search for Java applets demonstrating individual algorithms brings up

results of varying degrees of professionalism and success; applets which were either broken, boring or just not very useful were common, but there was a surprising dearth of quality content, particularly for basic 2D algorithms. A common problem I encountered was the applet which simply displays a final result (e.g. a line drawn or a circle filled) with no attempt to animate how the algorithm reaches the result. This is of little help — particularly when one can trivially visualise the desired output without even seeing the algorithm.

Two previous part II projects on computer graphics teaching have been completed but the approaches taken differ somewhat from my own. Richard Powell [5] created a system more focused on technical implementations of 3D algorithms while Akber Datoo [1] covered 2D algorithms but his approach incorporated scripts and help files to explain the algorithms rather than relying solely on animation and interactivity.

1.4 Aims

My aim was to create a tutor incorporating multiple algorithms in a single interactive environment, with consistency in the presentation and ease of use in controlling and switching between the algorithms. The algorithms would be presented in a manner conducive to viewing them in action to gain an understanding of their behaviour. I felt it was important that the focus of the tutor should lie firmly in the realm of interactivity and visualisation rather than concentrating on documentation, instruction or words of advice. Implementing wordy tutorials and explanations would do little to demonstrate computer science skills, and the results would most likely offer nothing that could not already be found in a decent text book [3] or set of lecture notes [2].

Chapter 2

Preparation

2.1 Requirements Analysis

My first action was to take the requirements outlined in the original project proposal and develop them into a more detailed requirements analysis. The central requirement was the creation of a system framework and user interface in which implementations of several computer graphics algorithms would operate. The framework would need to provide the following functionality:

1. Graphical representation of the state of the current algorithm (Visualisation).
    - Individual algorithms should have their own unique graphical content suited to demonstrating their behaviour.
    - Similar algorithms should be able to share common graphical objects or utilities.
2. Display of pseudo code for the current algorithm (Visualisation).
3. Tools for stepping through the algorithm and controlling its flow (Animation).
    - It should be possible to run the algorithm at varying speeds.
    - It should be possible to pause and step through manually.
    - It should be possible to reset the algorithm to its initial state.
    - The possibility of stepping through in either direction was noted as desirable but not essential.
4. Methods for altering the parameters of the current algorithm (Interactivity).
    - Wherever possible parameter alteration should be related to the visual nature of the algorithm. E.g. moving vertices by clicking and dragging rather than typing in coordinates.
    - Parameter alteration should be disabled during algorithm execution as the starting parameters are used by the algorithm.

5. The ability to quickly switch between algorithms.
    - Important state information should be retained between switches to avoid having to re-adjust parameters.

Ease of use, visual clarity and graphical consistency were noted as the sorts of qualities on which the success of the project would hinge.

2.1.1 User Interface

I identified the following key components as requirements for the user interface:

1. A menu for selecting algorithms
2. A canvas for graphical display of the current algorithm. This should be the central and most significant part of the UI.
3. A (scrollable) panel for displaying pseudo code
4. A panel with buttons for controlling the flow of the algorithm

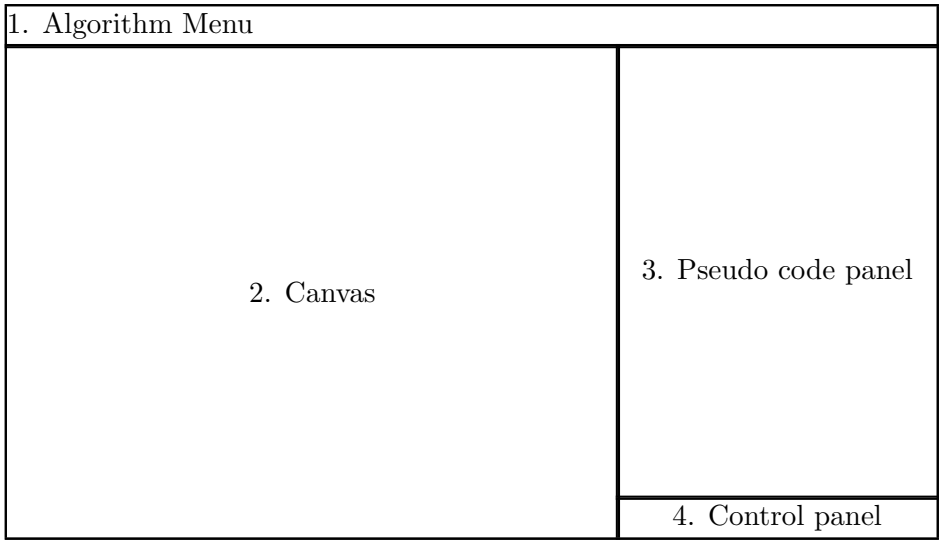


Figure 2.1: The initial concept for the user interface.

I designed a sensible layout for these components as sketched in Figure 2.1

2.2 Design Decisions

2.2.1 Choice of Language

The language I chose to write my system in was Java. This was the obvious choice for two main reasons. Firstly, familiarity; I had experience of using the language from

the part 1B group project and elsewhere in the computer science tripos. Secondly, the portability of Java and its integration with the web in the form of applets make it ideal for e-learning systems.

2.2.2 Choice of Environment & Tools

I chose to use the Eclipse[1] IDE. As an open source IDE encompassing impressive functionality, coupled with the familiarity I had with using it from my Part IB Group Project it was the ideal choice. For convenience I chose to use my own laptop running Windows XP.

2.2.3 Backup & Scheduling

Backup was performed daily to a USB flash drive, as well as regular backups to CD. I also maintained an up to date version of the project on my college file space. The college operates a monthly grandfather based back up system which was thought to be more than adequate for my needs. Initially no changes were made to the schedule outlined in my original proposal.

2.2.4 Development Approach

A rapid prototyping development approach for the system framework was used. The advantages to this were that, given usability and clarity were of paramount importance, getting a usable, visual prototype in place as quickly as possible would help focus development on these key areas, so that design choices could be verified and perfected by actually trying things out. My original schedule allowed enough time for prototyping, even in the possible eventuality of the prototype being entirely discarded and the final system framework being re-coded from the scratch. An additional advantage was that developing a simple line drawing algorithm alongside the prototype framework would allow me to develop a clear idea of how the algorithms and the framework should integrate.

2.2.5 Choice of Algorithms

I had outlined my intention to implement the following algorithms in my project proposal:

1. Line drawing
2. Clipping of a straight line against a rectangle
3. Scan-line polygon filling
4. Polygon clipping against a rectangle

[1] http://www.eclipse.org

These had been decided upon after careful consideration of the possible algorithms to cover and in consultation with my project supervisor so I decided to stick with my original choices, although I decided to attempt implementations of two different line drawing algorithms, the Bresenham method and the Midpoint line method. As the project progressed I also decided to attempt an implementation of a Bezier curve drawing algorithm, one of the proposed extensions in my original proposal.

2.3 Algorithms

It was important to develop a proper understanding of the algorithms I would be implementing. What follows is a collection of brief summaries outlining their general behaviour, subtleties revealed during close inspection and the assumptions I chose to make for my own implementations.

Pixel Coordinates Convention

It became apparent that I needed to adopt a convention for the representation of pixel coordinates. I chose to use the convention that a pixel with integer coordinates (x, y) has its centre at the real coordinate (x, y) as shown in Figure 2.2. Thus the pixel stretches from $(x - 1/2, y - 1/2)$ to $(x + 1/2, y + 1/2)$.

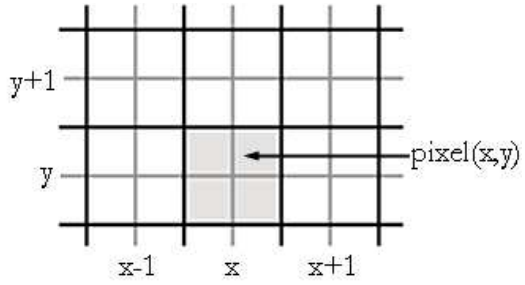
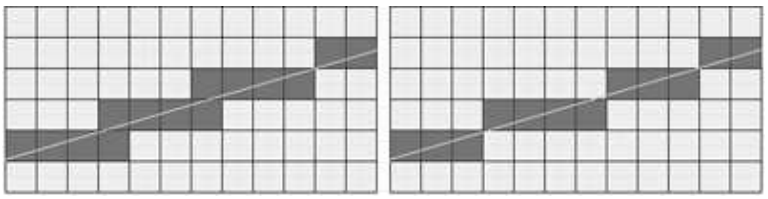


Figure 2.2: Convention used for pixel coordinates

2.3.1 Line Drawing

The problem of straight line drawing is how best to graphically approximate a mathematical line with a graphical line (a set of pixels). For the purpose of demonstrating the basic behaviour of the such an algorithm in as simple a manner as possible I decided to consider exclusively the case where lines are to be drawn at single pixel width and the end points of the line are given by integer coordinates corresponding to a specific pixel. We name these start and end points (x_0, y_0) and (x_1, y_1) respectively. Given a line defined by these points, our algorithm will need to decide which pixels to colour in order to best represent it.

There are two sensible possibilities when considering which pixels should be coloured, as shown in Figure 2.3. Method (a) is to colour every pixel through which



Method (a) Method (b)

Figure 2.3: Two possible approaches when considering which pixels to colour

the actual line passes while method (b) is to colour the 'closest' pixel to the line in each column of the pixel grid. Method (b) is generally considered to provide more pleasing results, and is the approach taken in the Part IB Computer Graphics course [2]; thus it was the obvious choice of method for my implementations.

Bresenham's Line Drawing Algorithm

The following pseudo code demonstrates the behaviour of the Bresenham line drawing algorithm in the case when the line is in the first or eighth octants, with point (x0,y0) to the left of point (x1,y1).

```
d = (y1-y0)/(x1-x0)
x = x0
yi = y0
y=y0
DRAW(x,y)

WHILE x < x1
      x = x + 1
      yi = yi + d
      y = ROUND(yi)
      DRAW(x,y)
END WHILE
```

Here the gradient of the actual line is calculated initially. Then on every increment of the current x coordinate, the gradient is added to the variable y_i. We then round y_i to the nearest integer to get the appropriate y coordinate. The pixel is then drawn.

The algorithm only needs to treat the cases of lines in the first, second, third and eighth octants. A line in any of the remaining octants can be transformed into a line in the mirroring octant by swapping the start and end points . The extension to the second and third octants simply involves simple changes to the above code to exchange the role of x & y.

Midpoint Line Drawing Algorithm

The midpoint line algorithm works by using a decision variable to determine the next pixel to colour. We can define the equation $K(x, y) = ax + by + c$ for some values a, b and c such that $K(x, y) = 0$ for values of x and y on the actual line. $K(x, y)$ is

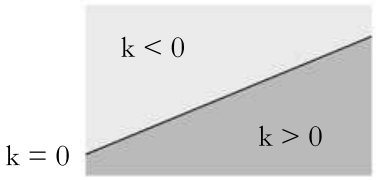
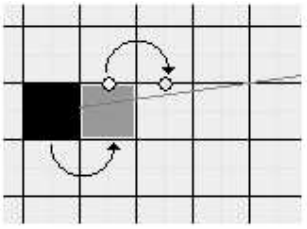


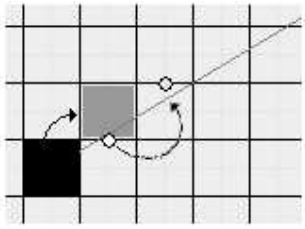
Figure 2.4: The values of $K(x, y)$ divide the plane into three regions

positive for points below the line and negative for points above the line, as shown in Figure 2.4.

In the case for the first octant the algorithm works as follows. At a particular pixel on the line (x_p, y_p), the next pixel must be either immediately to the right (the east pixel, E) or to the right and up one (the northeast pixel, NE). We define the decision variable $d = K(x_p + 1, y_p + 1/2)$, where the point $(x_p + 1, y_p + 1/2)$ is the midpoint M between E and NE. The sign of d tells us whether M falls above or below the actual line, and thus which pixel to pick. If $d > 0$ we choose NE. If $d \leq 0$ we choose E (we could have chosen either for the case $d = 0$).



- $K \leq 0$
- Next pixel: E
- Next point: $(x + 2, y + 1/2)$
- $d' = d + a$



- $K > 0$
- Next pixel: NE
- Next point: $(x + 2, y + 3/2)$
- $d' = d + a + b$

Figure 2.5: The midpoint decision cases

The location of M and the value of d for the next iteration depend on the choice made between NE and E. If E is chosen, M is incremented by one step in the x direction and it can be shown for the new decision value d' satisfies $d' = d + a$. If NE is chosen, M is incremented in both the x and y directions and $d' = d + a + b$. This is summarised in Figure 2.5. The following pseudo code outlines the implementation.

```
a = (y1 - y0)
b = (x0 - x1)
c = x1y0 - x0y1
d = a(x + 1) + b(y + 1/2) + c
DRAW(x,y)                    {The start pixel}
```

```
WHILE x < x1
      x = x + 1
      IF d<0 THEN            {Choose E}
            d = d + a
      ELSE                   {Choose NE}
           d = d + a + b
           y = y + 1
      DRAW(x,y)
END WHILE
```

Lines in the other octants can be handled by suitable reflections about the principal axes.

2.3.2 Bezier Curve Drawing

A Bezier cubic is defined as a function $P(t)$ where t ranges between 0 and 1.

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3 \tag{2.1}$$

Here, points P_0 and P_3 define the end points of the curve while points P_1 and P_2 are control points. The Bezier curve defined by the points lies within the convex hull of the points, as shown in Figure 2.6.

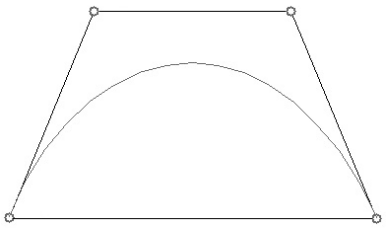


Figure 2.6: A Bezier curve and its convex hull

The naive method to draw a bezier is to draw straight lines segments equally spaced in parameter space t by feeding values into equation 2.1. However, since the distance in real space (x, y) is not linearly related to the distance in parameter space t this results in a poor approximation to the curve.

A more sensible method uses adaptive sub division. We check to see if a straight line drawn between points p_0 and p_3 can be considered an adequate approximation to the Bezier (specified by some given tolerance). If so, we draw the line; otherwise we split the Bezier into two halves. This results in two new Beziers, and we repeat the process for each.

The following pseudo-code illustrates the process:

```
BEGIN DrawCurve
      IF FLAT THEN DrawLine
      ELSE
            Subdivide Curve
            DrawCurve(Left)
            DrawCurve(Right)
      END IF
END Drawcurve
```

A test for flatness is to estimate the point furthest away from the line $\overline{p_0 p_3}$ and check whether the distance is within a given tolerance.

2.3.3 Line Clipping

This is the problem of how to clip a straight line between two points against a clipping boundary. In this case we take the clip area to be a rectangle. Clipping the line means that only the part of the line which falls inside the clipping boundary should be drawn. The clipping rectangle is defined by the lines $x = x_{left}$, $x = x_{right}$, $y = y_{top}$ and $y = y_{bottom}$.

Cohen-Sutherland Algorithm

A brute force approach would be to intersect the line with each of the edges and to check whether any intersection points lie on these edges. The Cohen-Sutherland algorithm provides a more efficient approach by performing initial tests on the line to limit the intersection calculations which need to be performed.

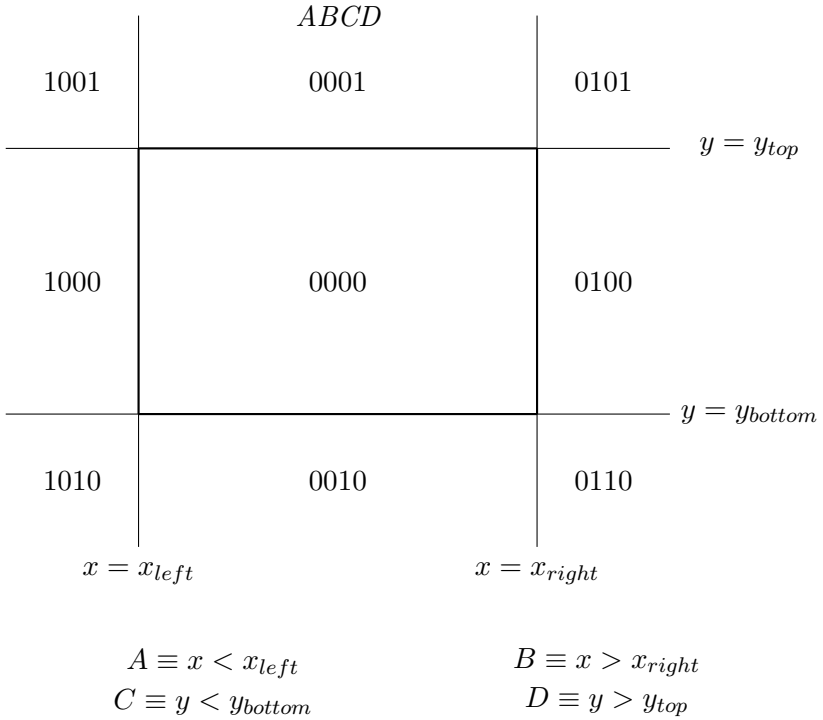


Figure 2.7: The four cases of polygon clipping. The rhombus represents a simple polygon being clipped.

Firstly, we extended the edges of the clip rectangle and divide the plane into nine regions as shown in Figure 2.7. Each region is assigned a four bit code, where each

of the four bits represents where the region lies with respect to one of the four edges, as specified by the inequalities in Figure 2.7.

Each end point of the line segment we are clipping is assigned the code of the region in which it lies. We can then use these endpoint codes to determine whether the line segment lies entirely inside the clip rectangle or entirely in the outside half-plane of an edge. If both codes are equal to zero then the line segment lies entirely inside the rectangle and the line can be trivially accepted. If the logical AND of the codes is not equal to zero, then the line segment must lie entirely in the outside half-plane of one of the edges and therefore can be trivially rejected.

If the line segment cannot be trivially accepted or rejected then we intersect the line segment with one of the edges and throw away the segment in the outside half-plane of the edge. The choice of the edge to clip against is determined by the 1s in the endpoint code. If there are two 1s in the code, then we simply follow a convention to pick between the two edges; e.g. pick the edge corresponding to the first 1 in the code.

Once the line segment has been clipped against an edge, the new endpoint code is calculated and the algorithm proceeds until a line segment is trivially accepted.

2.3.4 Polygon Filling

Given an arbitrary polygon, which pixels do we colour to represent a filled version of the polygon? To solve this problem we use the Polygon scan-conversion algorithm, which works by determining which pixels on each scan line are within the polygon. To do this the algorithm computes the scan line/polygon edge intersections and then fills between intersection points, as shown in Figure 2.8. We say that a pixel is within the polygon if the centre of the pixel lies inside the polygon.

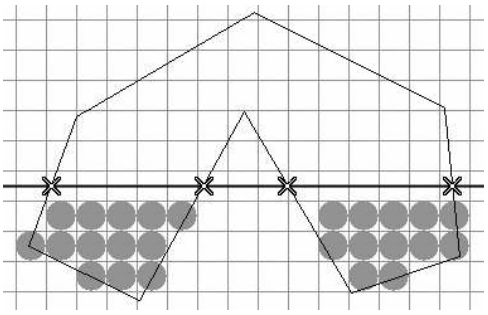


Figure 2.8: Scan line polygon fill process

Polygon Scan-conversion Algorithm

The polygon is broken up into its individual edges which are placed in an edge list (EL), sorted on the lowest y value.

We set the current scan line to be the first scan line that intersects the polygon, and move all edges which intersect that scan line into an active edge list (AEL). We then repeat the following process until the AEL is empty:

1. For each edge in the AEL, we find the intersection point with the current scan line.
2. These points are then sorted into ascending order on the x value.
3. Fill between pairs of intersection points.
4. Move to the next scan line and update the AEL.

The final stage involves removing any edges from the AEL where the y value of the endpoint is less than the y value of the scan line. We then move new edges from the EL to the AEL where the y value of the start point is less than or equal to the y value of the scan line.

There is a subtlety to consider when an endpoint of an edge intersects a scan line exactly. The point needs to be shifted by a small amount to prevent the calculation of two intersection points at the same endpoint.

2.3.5 Polygon Clipping

How do we clip one arbitrary polygon against another? For simplicity we treat the case of clipping against a rectangle but as we shall see, the algorithm used can easily be extended to an arbitrary polygon.

Sutherland-Hodgman Polygon Clipping

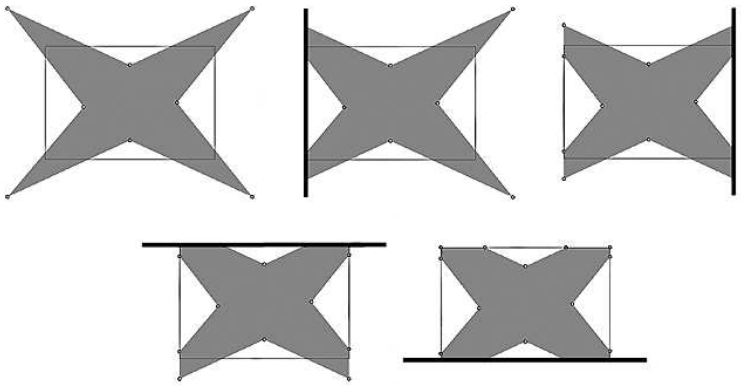
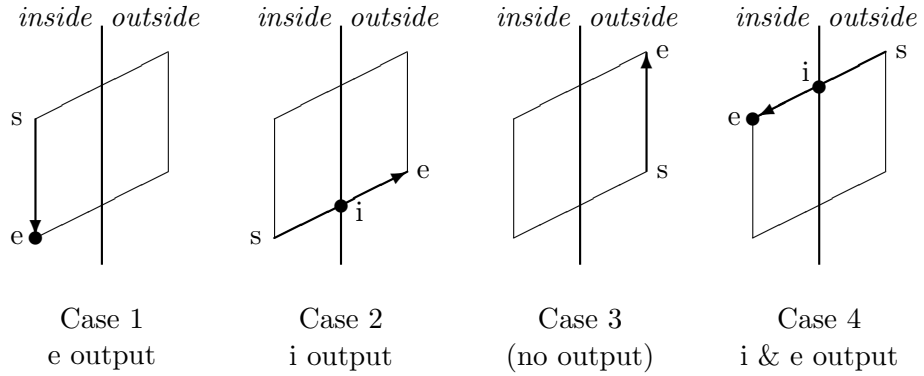


Figure 2.9: Polygon clipping against successive edges of the clipping rectangle

The Sutherland-Hodgman algorithm clips an input polygon against a single infinite clip edge. Thus the complex task of clipping a polygon against another polygon is reduced to a simpler task which can be called recursively. In the case of clipping

against a rectangle we clip against each of the four rectangle edges in turn passing the result each time to the next stage, as shown in Figure 2.9. Thus we can also easily clip against an arbitrary polygon if we so desire.

The algorithm accepts a series of vertices $v_1, v_2, \ldots, v_n$ defining the polygon with edges from v_j to v_{j+1} and v_n to v_1. It then clips against a single, infinite edge and outputs another series of vertices defining the clipped polygon.



Case 1	Case 2	Case 3	Case 4
e output	i output	(no output)	i & e output

Figure 2.10: The four cases of polygon clipping. The rhombus represents a simple polygon being clipped.

The algorithm moves around the vertices of the polygon from v_1 to v_n and then back to v_1, considering an edge at each stage and taking initially v_1 for the start vertex s and v_2 for the end vertex e. At each step the relationship between the edge $\vec{se}$ and the clip edge is examined, and zero, one or two vertices are added to the sequence of vertices representing the output polygon. The are four cases to consider, as shown in Figure 2.10. On occasions when the edge $\vec{se}$ crosses the clipping edge, a new vertex i is created at the intersection point and added to the output. In each case, the current vertex for e is taken for the next start vertex s, and the algorithm proceeds to the next vj which is taken for the next e.

Chapter 3

Implementation

3.1 Overall Design

At its most basic level the system can be split into two components; a set of frameworks incorporating a UI for the display and control of an algorithm and a generic algorithm class which runs in this framework.

It should be noted that my use of the word 'algorithm' refers to the individual graphics algorithms I chose to implement for the system, rather than more general algorithms used in these implementations, unless otherwise stated.

3.1.1 Algorithm

I needed a method for defining an algorithm as a series of simple logical stages of computation with associated lines of pseudo code and functionality to step through these stages from start to finish. Additionally, I needed a system for associating a graphical representation of the algorithm at each step of computation.

I faced a choice between two approaches to handling algorithm computation. One approach is to pre-compute the algorithm in its entirety, creating a log file of relevant data at pre-identified 'stages' of its computation. This log file could then be used to step through the algorithms stages of computation - with the advantage that stepping through backwards is just as simple as stepping through forwards. However certain algorithms, particularly recursive ones like the Bezier curve drawing algorithm, could generate extremely large log files. Also, visualising the algorithms progress would either require adding further data to the log file or creating a further component to interpret the current step of the log file and transform it into some suitable form of graphics.

The alternative was to compute the algorithms step by step in real time, delaying computation of the next step according to user control. This approach involves a sacrifice; it is no longer (straightforwardly) possible to step through in both directions. The advantage is that it is conceptually much simpler. The process of splitting the algorithm into its stages is performed beforehand and embedded in the actual

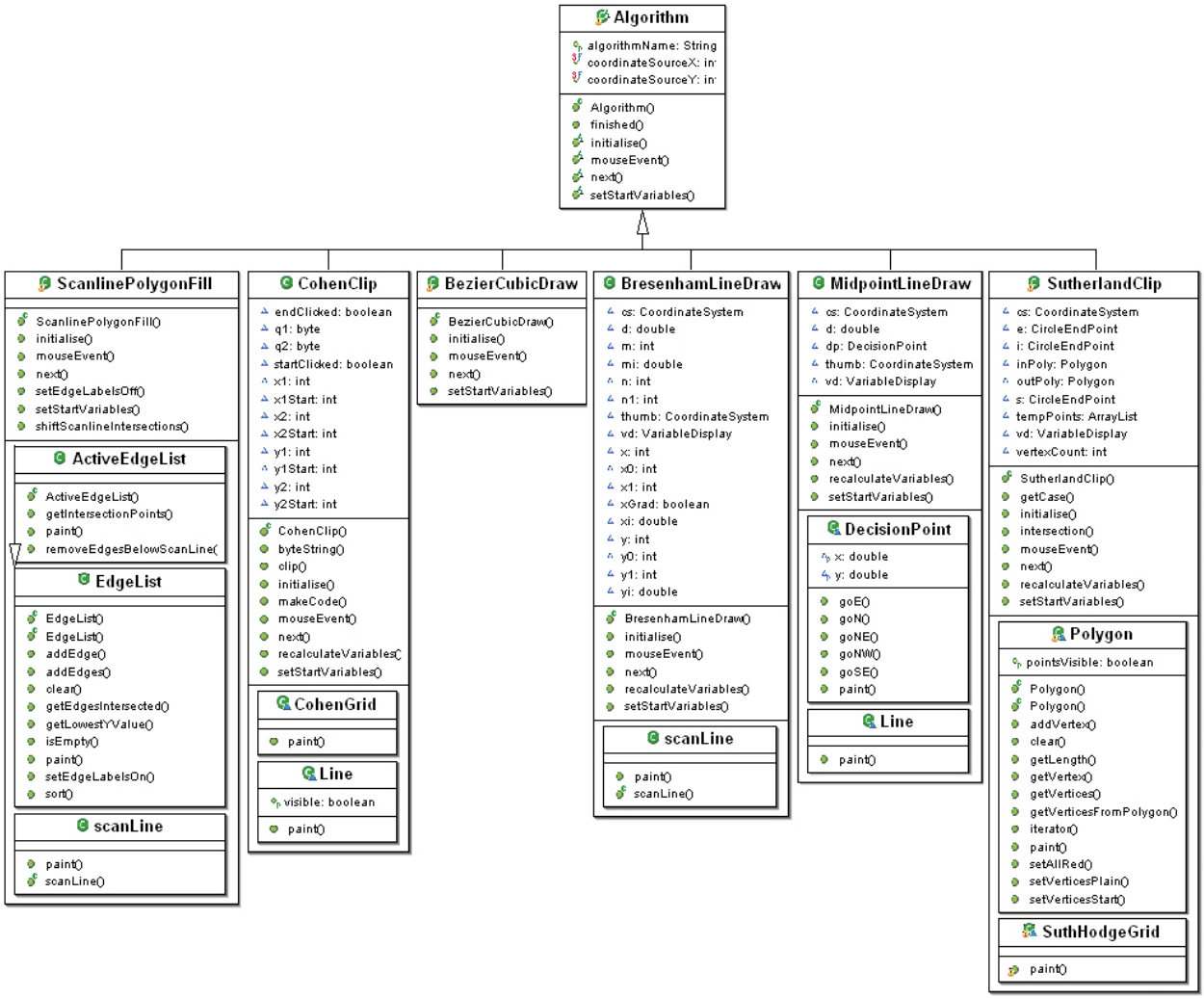


Figure 3.1: UML diagram of the Algorithms Package. The individual classes used to implement each of the algorithms are explored in the following sections.

implementation. Relevant graphics for visualising of the algorithm can be thus be associated simply with the individual stages of computation. Because of this advantage, I chose to adopt this second approach.

With this approach decided upon I created the abstract class Algorithm to represent a generic graphics algorithm. An explanation of the class follows, while later in the chapter I will turn to the implementations of the individual algorithms which extend this parent class, as outlined in Figure 3.1.

The class Algorithm has three important abstract methods called *setStartVariables()*, *initialise()* and *next()* which are now dealt with in turn.

Before stepping through an algorithm the user has the opportunity to graphically alter the parameters which the algorithm receives as input. Thus before execution begins the algorithm needs to exist in some 'start state' conducive to the modification

of parameters. The method *setStartVariables()* is called to handle the correct creation of this initial state.

After modifying the parameters the user starts execution of the algorithm. The method *initialise()* is called to update the algorithm so that is ready to execute based on the (potentially) updated parameters. Additionally an appropriate set of pseudo code statements representing how computation will progress are generated according to the input received. Each pseudo code statement is given an associated execution line number. *initialise()* sets the current line number, which acts like a program counter, to zero as by convention the pseudo code statement representing the first 'stage' of the algorithm's behaviour is given line number zero.

Subsequently the algorithm progresses through its 'stages' of computation. The method *next()*, is called repeatedly to execute each successive computational step. In each individual algorithm implementation this method uses a single (and potentially large) Java switch statement[1] to encapsulate all the logic of the algorithm's computation. The switch statement evaluates the current line number and executes the appropriate case statement, which contains the code relevant to the current line number. This code not only covers the logical computation for the particular stage, but also generates the appropriate graphics objects to represent the algorithm in its current state. Thus computation and visualisation can be dealt with together. Also, the code updates the line number so that execution proceeds to the subsequent stage the next time *next()* is called.

Thus algorithms consisting of any combinations of basic blocks of instructions and complicated loop structures can be split into logical executions steps and converted into a large switch statement. This method is demonstrated in the example in figure 3.2.

3.1.2 Tutor Framework

Algorithms of the form outlined above operate within the system framework and the main controlling class in this framework is called TutorFramework. TutorFramework directs the process of interfacing with the current algorithm. It acts like a central hub, passing messages between classes; most often to the separate Control and Display sub-frameworks which it contains.

TutorFramework contains a Java ArrayList[2] called algorithms which contains a single instance of each of the algorithms. A field called currentAlgorithm holds the instance of the currently selected algorithm. There are a set of methods to handle requests to change the current algorithm, start or restart computation, move to the next step of computation etc.

TutorFramework follows the singleton design pattern, allowing easy access to the single instance from any of the other classes in the project. Thus its large collection

[1] http://java.sun.com/docs/books/tutorial/java/nutsandbolts/switch.html
[2] http://java.sun.com/j2se/1.4.2/docs/api/java/util/ArrayList.html

Line number	Pseudo code	Actual system code
		`switch(lineNumber)`

0	`WHILE x < x1`	`case 0:`
		`IF(n<n1) THEN lineNumber = 1`
		`ELSE lineNumber = 5`

1	`x = x + 1`	`case 1:`
		`x = x + 1`
		`lineNumber = 2`

2	`yi = yi + d`	`case 2:`
		`yi = yi + d`
		`lineNumber = 3`

3	`y = ROUND(yi)`	`case 3:`
		`y = ROUND(yi)`
		`lineNumber = 4`

4	`DRAW(x,y)`	`case 4:`
		`DRAW(x,y)`
		`lineNumber = 5`

5	`END WHILE`	`case 5:` `finished()`

Figure 3.2: Transforming line drawing code into a switch statement.

of controlling methods are easily accessible from any other framework and within the current algorithm.

3.1.3 Control Framework

The class ControlFramework is a sub-class of the Java class JPanel[3] and provides the user with the tools to control the current algorithm. It also controls the display of the pseudo code lines representing the current algorithm since this is the representation of the algorithm most associated with control flow.

User control is handled by a JPanel object called controlPanel which contains buttons for stepping through the algorithm either by a single step or by repeated successive steps. There are also buttons for resetting the current algorithm and adjusting the current run speed. Java ActionListeners[4] associated with each button

[3]http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/JPanel.html
[4]http://java.sun.com/j2se/1.5.0/docs/api/java/awt/event/ActionListener.html

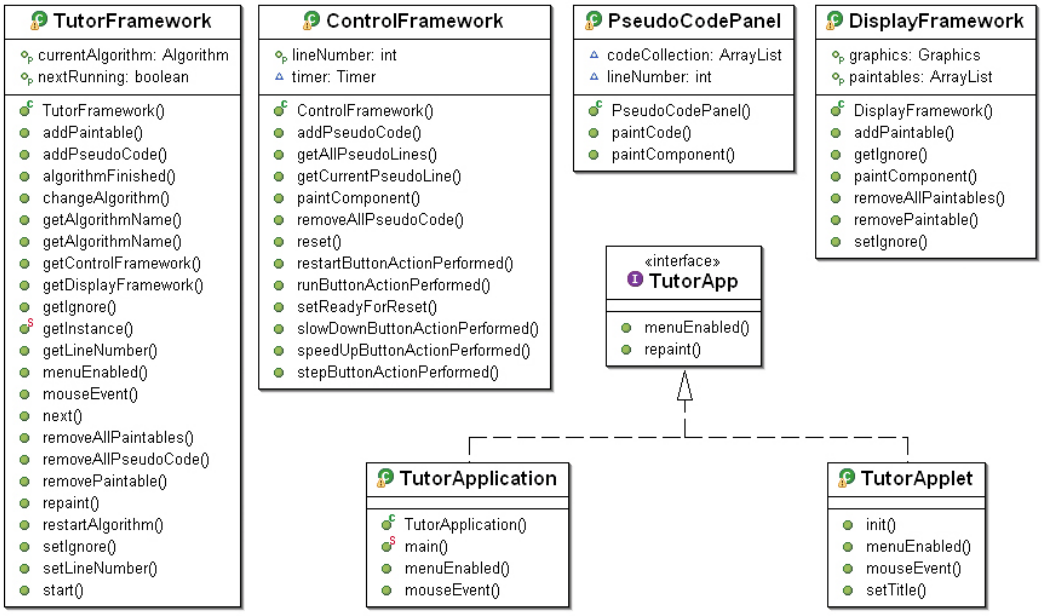


Figure 3.3: UML diagram of the Tutor Package. These classes form the framework outlined in this section

call the relevant methods in TutorFramework when an event occurs.

Running through the algorithm, or automatically stepping through successive steps at a given speed, is made possible by a Timer[5] object, with successive calls to the current algorithms next() method scheduled according to the current speed.

ControlFramework also contains a PseudoCodePanel object. The class PseudoCodePanel is another sub-class of JPanel and is used for the display of pseudo code instructions. It maintains a collection of pseudo code strings associated with their pseudo line numbers. It displays the pseudo code instructions and highlights the current instruction.

3.1.4 Display Framework

In order to discuss this framework I need to introduce the abstract class Paintable, which is expounded in a later section. I use Paintable objects, hereafter referred to as 'Paintables', to graphically represent the current state of an algorithm. The key method of the class is paint(), which draws some set of graphics representing whatever the object is meant to embody.

Paintables are passed from the current algorithm to the DisplayFramework class. DisplayFramework maintains a collection of Paintables and paints them to the canvas by calling their individual paint() methods. Paintables are created and maintained by the current algorithm and thus contain information about the state of the algorithm.

[5]http://java.sun.com/j2se/1.5.0/docs/api/java/util/Timer.html

The role of DisplayFramework is merely to processes the Paintables by executing their paint() method and ensuring they are painted in a sensible order.

The DisplayFramework also provides functionality for visual parameter alteration using Java MouseListeners[6]. On receiving an 'interesting' mouse event such as a click or a drag a method in TutorFramework called mouseEvent() is called which in turn passes on the information about the event to the current algorithm so that it can be processed appropriately.

3.1.5 Tutor App

A TutorApp object is used to put the elements of the framework together so that they can actually run in some environment. TutorApp is an abstract class, its two subclasses being TutorApplication and TutorApplet, general high level classes which allow the system to function as either an application or a Java applet running within a web browser. These classes define the look of the general UI and provide a menu bar with a menu for switching between algorithms. Such requests are passed to the TutorFramework by a call to the method changeAlgorithm(). The UI is nearly identical for both the application and applet versions and takes the form outlined in the preparation section, as shown in Figure 3.4.

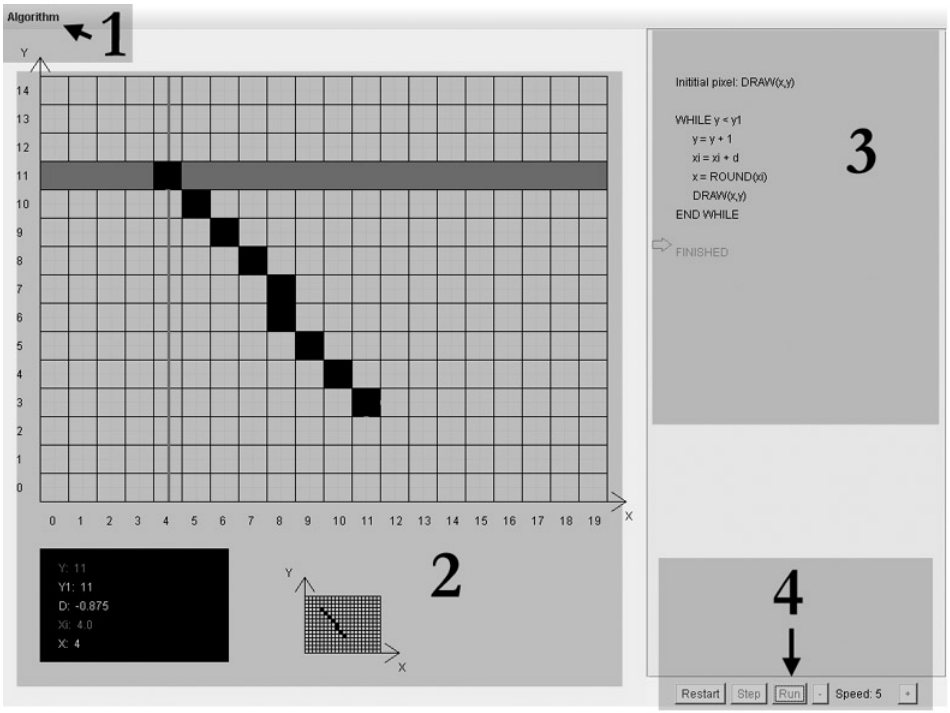
3.2 Algorithm Utilities

3.2.1 Paintable Class

We have already seen how Paintable objects, or Paintables, are passed to the DisplayFramework. I implemented several features to increase the flexibility in graphically representing algorithms with these objects. Firstly I associated colour information with each Paintable object. Within the abstract class Paintable I defined nine static final colours which were sufficient to cover the common colours used repeatedly by the different algorithms. This allowed me to refer to these recurring colours using simple code such as Paintable.RED. I also defined fields holding a current colour and a default colour for each Paintable object. This proved particularly useful for Paintables representing endpoints or edges, where colour might be used to express information about the state of the object – e.g. whether it has been clicked or not.

Finally I defined a protected boolean field called background, used to ensure that certain Paintables are drawn first as background objects. The DisplayFramework initially draws all Paintables with the boolean background set to true, before moving on to foreground Paintables. This provides sufficient power and flexibility without the need for introducing higher levels of complication by associating an 'alpha level' with each Paintable.

[6] http://java.sun.com/j2se/1.5.0/docs/api/java/awt/event/MouseListener.html



1. The menu for switching between algorithms
2. The canvas for graphical display of the current algorithm
3. The pseduo code panel
4. The control panel for controlling the flow of the algorithm

Figure 3.4: The UI layout

3.2.2 Coordinate Systems

An early problem I encountered during development involved Java's default coordinate system, called 'User space'. It starts in the top left hand corner of the drawing area, and works its way downwards to the right. This is the reverse of what I wanted to use for the algorithms I was implementing.

A further consideration was that some of my algorithms would need to display pixel grids magnified to many times their actual size so as to clearly demonstrate the algorithm's behaviour. I realised that it would be extremely useful to be able to abstractly treat this magnified pixel grid as the coordinate system on which I was working.

I created the CoordinateSystem class specifically for this purpose. It allows me to create 'virtual' coordinate systems and to specify calculations and actions using this system. The class takes care of mapping all this back to the reality of Java's 'User space' so that everything is displayed properly. This system led to the dramatic

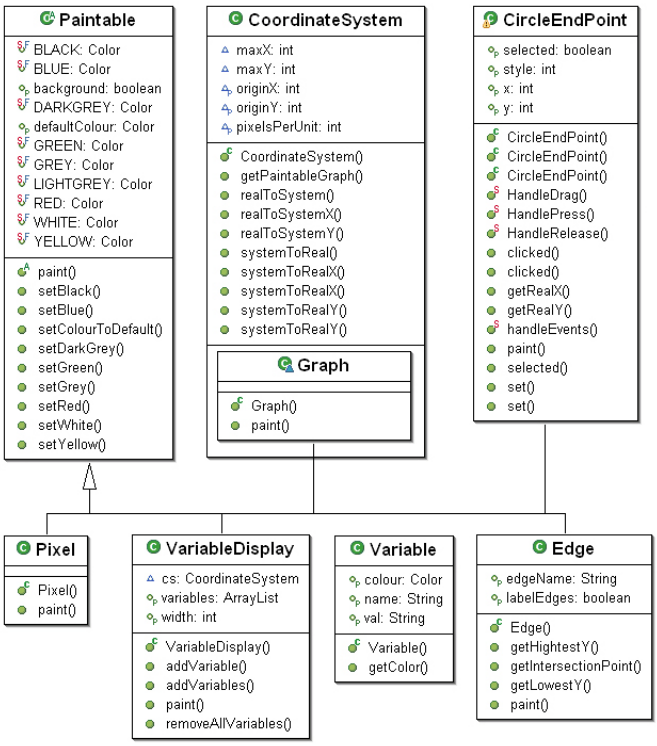


Figure 3.5: UML diagram of the Utilities Package. The Utilities will be outlined in this section

simplification of much of the code for my algorithms. It was also entirely possible (and useful) to use multiple coordinate systems within the same algorithm.

I chose to refer to Java 'User space' as 'real' and the newly created coordinate system as 'system' and created methods within CoordinateSystem such as realtoSystem() which takes a Point specified in 'real' coordinates and returns a point expressed in 'system' coordinates. I also created individual x and y coordinate methods performing the same task on the individual x and y values of a coordinate, as well as methods to convert in the opposite direction, where I followed the convention that the 'real' coordinate returned is that of the top left-hand pixel of the 'system' coordinate. Sample code for some of these methods can be found in Appendix A.

3.2.3 Automated Graph Generation

A common use of the CoordinateSystem class was to create the underlying coordinate system for a grid or graph which would form part of the graphical display of the algorithm. Rather than expensively coding a Paintable class representing the graph within the algorithm it made sense for the CoordinateSystem class to generate such graphs automatically. I created the class Graph, a subclass of Paintable and an inner class of CoordinateSystem, to provide this functionality.

I decided that such a graph would contain marked grid lines in black as standard, while also offering further options for numbered axes, arrowed axes and a faint grey background grid. The constructor within CoordinateSystem takes as arguments three Booleans used to flag which of these features should be included in the Paintable Graph object returned. The graphs and grids used to visualise a number of the final implementations were generated in this manner, saving the time that would have been required to code individual Paintable objects for each of the mildly differing cases.

3.2.4 Circle End Points

Many of the algorithms feature control points—whether they be the endpoints of a line or the vertices of a polygon. Often these control points are used to configure the input for the algorithm. I found there was a need for a class to represent a general endpoint or vertex, which was also flexible enough to allow for points of different sizes, styles and colours for different algorithms. I created the Paintable class CircleEndPoint for this purpose.

Each CircleEndPoint has an associated CoordinateSystem object as well as fields for its x and y coordinates relevant to this coordinate system. The CoordinateSystem is used to convert the system x and y values into actual x and y values for use with the Java User space. There are also fields for the size of the CircleEndPoint and its style—an integer value which determines the style in which the CircleEndPoint is painted. I created five separate styles including entirely filled, white with a coloured outline of varying thickness and a style which also draws a cross through the endpoint.

To simplify their role in parameter setting, each CircleEndPoint also has a Boolean value which flags whether the CircleEndPoint has been clicked on and can currently be considered 'selected'. Also provided is a method clicked() which accepts either integer coordinates or a mouseEvent, coupled with an integer tolerance value. This method converts the coordinate values of the mouse Event into 'system' values using its CoordinateSystem and then checks to see whether the CircleEndPoint can be considered to have been clicked, given a specified tolerance. It returns a Boolean value representing whether the CircleEndPoint was clicked, and updates its boolean field accordingly.

It became apparent that all the algorithms would utilise at least two CircleEndPoints for providing the user with a way of defining the input to the algorithm. Since the requirement for code to handle the mouse movement of the endpoints was common to all the algorithms, I decided to create a set of static methods associated with the CircleEndPoint class which were capable of handling the problem in its entirety. I created the method CircleEndPoint.handleEvents(), which when passed a specific set of parameters handles all user control of the endpoints. The parameters required are an integer ID of the mouseEvent type, the mouseEvent itself, a collection of CircleEndPoint, the associated CoordinateSystem and an integer tolerance value specifying the clicking accuracy required for selection.

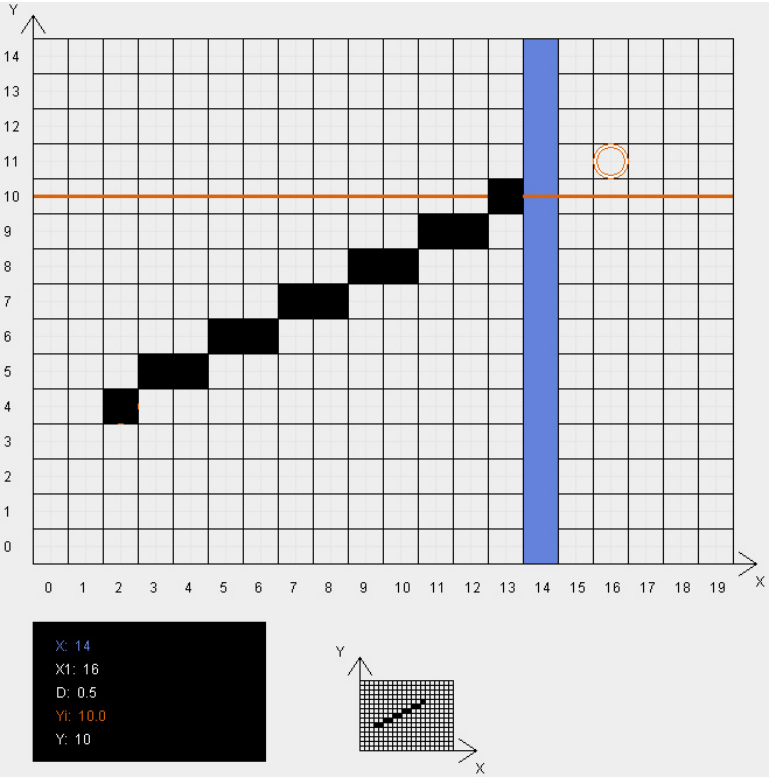


Figure 3.6: The final graphical display for Bresenham line drawing algorithm implementation

3.2.5 Variable Displays

Rather than creating a separate framework to deal with displaying variables in use by the current algorithm, I developed a Paintable class called VariableDisplay. A VariableDisplay object is maintained by the algorithm so that it contains a set of variable names to display with associated up to date values. Variables are defined by a separate class which also allows a colour to be associated with the variable allowing for colour consistency between the variable and the Paintable objects which visualise it.

Figure 3.6 demonstrates the final display for the Bresenham line drawing algorithm and contains examples of most of the Paintable objects and utilities mentioned in the last few sections, including an automatically generated graph, CircleEndPoints for the start & end of the line and a VariableDisplay object.

3.3 Line Drawing

Presentation Decisions

In my preparation I had outlined algorithms for drawing a line in the first octant. I needed to decide whether my implementation should be limited to lines in this octant,

or whether I should extend it so that the user could draw an arbitrary line in any quadrant. The difficulty here was that this would result in slight variations in the algorithm which would require different pseudo code. In the Bresenham algorithm it creates two different sets of pseudo code; in the midpoint algorithm it creates four.

I decided that limiting line drawing to only one or two octants would prove quite restrictive, and it would be much more desirable for the user to be able to draw any line they wanted. Observing how the pseudo code differed for lines in different octants would also be a potentially useful learning exercise for the user. This choice meant that my algorithm would need to evaluate the line to be drawn before commencing computation in order to generate the appropriate set of pseudo code statements.

I wanted the basic graphical display for this algorithm to be a pixel grid with clearly labelled x and y axes. The Paintable Graph generated by the CoordinateSystem class was developed to meet this requirement. A second CoordinateSystem object was used to create a 'thumbnail' grid so that the progress of the algorithm can be viewed at a resolution closer to actual single pixel resolution. I also decided that I needed to graphically display the values of certain variables such as the x and y coordinates, which led to the development of the VariableDisplay utility. Figure 3.6 shows the final version of the Bresenham line drawing algorithm with all these Paintable objects in place.

Parameter Setting

I needed a method for the user to set the parameters (i.e. endpoints) for drawing a given line. The options were either to allow the user to draw a line on a blank grid using a combination of mouse clicks and drags, or to have a default line already in place in the form of a single startpoint and a single endpoint. These points could then be dragged to any grid location.

I chose the second option for a number of reasons. It emphasized the fact that we were only drawing a single line and it allowed the algorithm to be initially displayed in a default state and demonstrated instantly without the need to configure endpoints. It also provided a simple way of preserving the current parameters when a different algorithm was selected—when returning to the line drawing algorithm one would simply encounter the endpoints unmoved from last time. It was this choice which led to the development of the CircleEndPoint utility described earlier.

Recall that I had taken a design decision to implement the program framework by rapid prototyping while concurrently developing a simple line drawing algorithm. This is why it was my work on the line drawing algorithms which led to the development of so many of the utilities that were to be used in my final algorithm implementations. My initial approach proved to be a useful one in this respect.

Implementation Details

The implementation process for each of the algorithms followed the same general pattern. The first stage was to decide how to best represent the algorithm as a sequence of pseudo code instructions. This process was manageable thanks to the time I had taken to familiarise myself with the algorithms during my preparation. With a pseudo code version of the algorithm outlined, I would implement the creation of these pseudo code statements by the algorithm in the *initialise()* method. I would then implement the switch statement which would embody the pseudo code and its control flow. This would form the content of the *next()* method.

With this framework in place the real part of the implementation could begin– writing the code and functions to perform each of the pseudo steps, and then writing the code to handle the creation and management of the appropriate Paintable graphics objects at each step.

Bresenham Algorithm

The behaviour of the Bresenham line algorithm varies between two cases depending on the gradient of the line to be drawn. The first step therefore is to establish the gradient of the line and to set a Boolean value xGrad which is subsequently used to flag which case is being followed. The two cases can be treated as identical except for the fact that the roles of the x and y axes are switched. Therefore I decided to implement a single algorithm with dummy variables n and m standing in for x and y. Then during the process of establishing which case applies the variables n and m can be set to either x and y or y and x. During this process the relevant set of pseudo code instructions are generated. With the values of m and n appropriately set the algorithm can simply proceed:

```
d = (n1-n0)/(m1-m0)
m = m0
ni = n0
n=n0
DRAW(m,n)

WHILE m < m1
      m = m + 1
      ni = ni + d
      n = ROUND(ni)
      DRAW(m,n)
END WHILE
```

Source code for the *initialise()* and *next()* methods can be found in Appendix A.

Midpoint Algorithm

The midpoint algorithm works in a similar manner only this time there are four possible cases to distinguish between. These cases are for lines in the first to fourth quadrants—the other quadrants are dealt with by switching endpoints. Therefore the first task of the algorithm, in the *initialise()* phase, is to establish which case applies

and to generate the appropriate pseudo code for this specific case. A variable called quadrant takes values between 0 and 3 and is used to flag which of the cases applies throughout the algorithm. There was no straightforward way of generalising the algorithm with dummy variables so switch statements on the variable quadrant are performed at several points to establish which code to execute. The code in *initialise()* establishes which value to assign to the field quadrant based on the gradient of the line being drawn. It first ensures that the start point is the left-most point. The code can be found in Appendix A.

With the relevant case determined, the rest of the algorithm is straightforward. Much of the required functionality was embedded in the DecisionPoint Paintable class. An instantiated object of this class contains not only the code required to paint the decision point at its current location, but also built in methods named goEast(), goNorth() etc. These methods update the DecisionPoint's location as appropriate. Use of these methods made the code in the method *next()* little more complicated than the actual pseudo code. The code for the DecisionPoint class is also in Appendix A.

3.4 Bezier Cubic Drawing

Presentation Decisions & Parameter Setting

I opted to present a single Bezier curve defined by its four control points represented by CircleEndPoints. I also created a Paintable object for displaying and editing the tolerance value used for drawing the Bezier. This was an inner-class called ToleranceSlider. The control points and tolerance slider are shown in Figure 3.7.

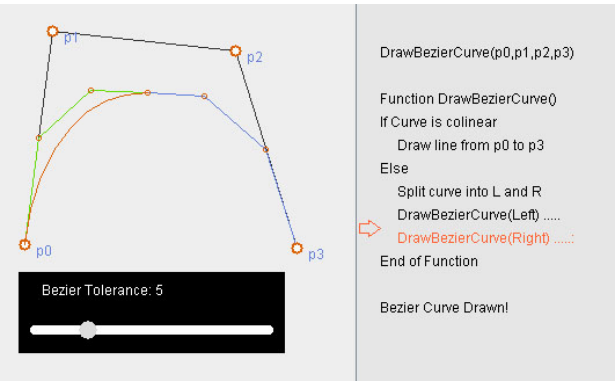


Figure 3.7: Parameter setting & pseudo code for the Bezier curve drawing implementation.

An alternative approach would have been to have the Beizer defined by four sequential mouse clicks, and perhaps have multiple curves drawn on the canvas. However this approach seemed out of place with the approach taken for the other algorithms, and inappropriate given that the algorithm was to be stepped through slowly rather than the curve being drawn instantly.

```
public boolean flat(){
  double mTol = 5.0 + (tolerance*tolerance*tolerance/80);
  double d1 = Math.abs(p0.X() + p2.X() - p1.X() - p1.X());
  double d2 = Math.abs(p0.Y() + p2.Y() - p1.Y() - p1.Y());
  double d3 = Math.abs(p1.X() + p3.X() - p2.X() - p2.X());
  double d4 = Math.abs(p1.Y() + p3.Y() - p2.Y() - p2.Y());
  return((d1+d2+d3+d4)<=mTol);
}
```

Figure 3.8: Code for Bezier flatness test

Implementation Details

The Bezier curve drawing algorithm is recursive, which presented me with a different problem. Recall that a procedure DrawBezierCurve only draws a curve if it is deemed that it can be sufficiently well represented by a straight line. If this is not the case it splits the curve into left and right component curves and calls the procedure recursively on these two curves in turn. How much code was I to encourage the user to step through? I decided that looping through endless calls to DrawBezierCurve would be superfluous. Instead I decided to only present a single call to the procedure DrawBezierCurve, and demonstrate the process of either drawing a straight line or splitting the curve in two and calling the procedure again on the new curves. However, these subsequent calls would not be stepped through recursively but would instead just run to completion in single steps. Thus my final implementation is rather brief but does demonstrate the process of splitting the Bezier, as shown in Figure 3.7.

I needed to implement a suitable algorithm for testing a Bezier curve for flatness. Recall that curve is flat if the error in approximating it by a straight line is within a given tolerance. Since this part of the algorithm is not included in the pseudo code and is effectively hidden from the user it was not necessary to implement the most rigorous or efficient test. The method I chose is based on the observation that the Bezier curve of a uniform straight line has evenly spaced control points along a straight line from p_0 to p_3. Therefore a measure of how good a straight line approximation is for a given curve is to calculate how far the curve deviates from this ideal case. The code in Figure 3.8 implements the test using this method.

Given this flatness test I used experimentation to normalise my range of available tolerances (1-20) so that the results translated to a sensible range of curves when the program was used. Figure 3.9 shows the results of drawing the same curve with different tolerances.

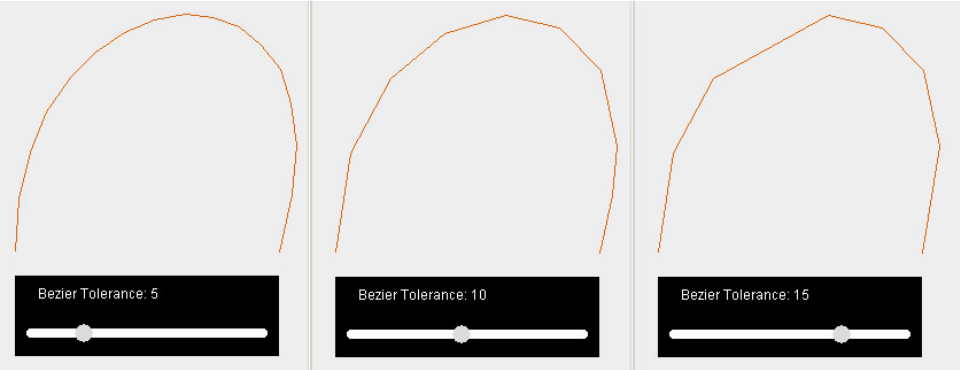


Figure 3.9: A curve drawn at tolerance levels 5, 10 & 15.

3.5 Cohen Line Clipping

Presentation Decisions & Parameter Setting

I adopted the classic graphical display for this algorithm; a nine section grid displaying the four bit area codes for each section and with the clipping rectangle in the middle. I created a separate subclass of Paintable called CohenGrid as an inner class of the Cohen clipping algorithm class. I decided against allowing the user to change the size of the clipping grid as this would add nothing in demonstrating the algorithm's behaviour.

The line to be clipped is represented initially by a pair of movable CircleEndPoints. During the clipping process these endpoints move, leaving 'shadows' of the endpoint in their place so that the behaviour of the algorithm can be tracked. Then, once clipping is complete, the final clipped line is drawn. This required the creation of a Paintable Class Line, an inner class of the algorithm.

I used the VariableDisplay class to display up to date values of the four bit codes generated for the line endpoints as the algorithm progressed, as well as the result of taking the logical AND of these codes. Figure 3.10 shows the final result running.

Implementation Details

This algorithm proved one of the most straightforward to implement. There were two main algorithms to implement. The first, makeCode() generates the appropriate four bit code given a point defined by its coordinates. The second, clip(), performs the next clip once it has been established that the current line needs clipping. It works as follows:

1. Retrieve a point which lies outside the clipping box (we know this applies to at least one of the two points).

2. This point must lie in the outside half-plane of one (or two) of the clip edges. Establish an edge for which this is the case.

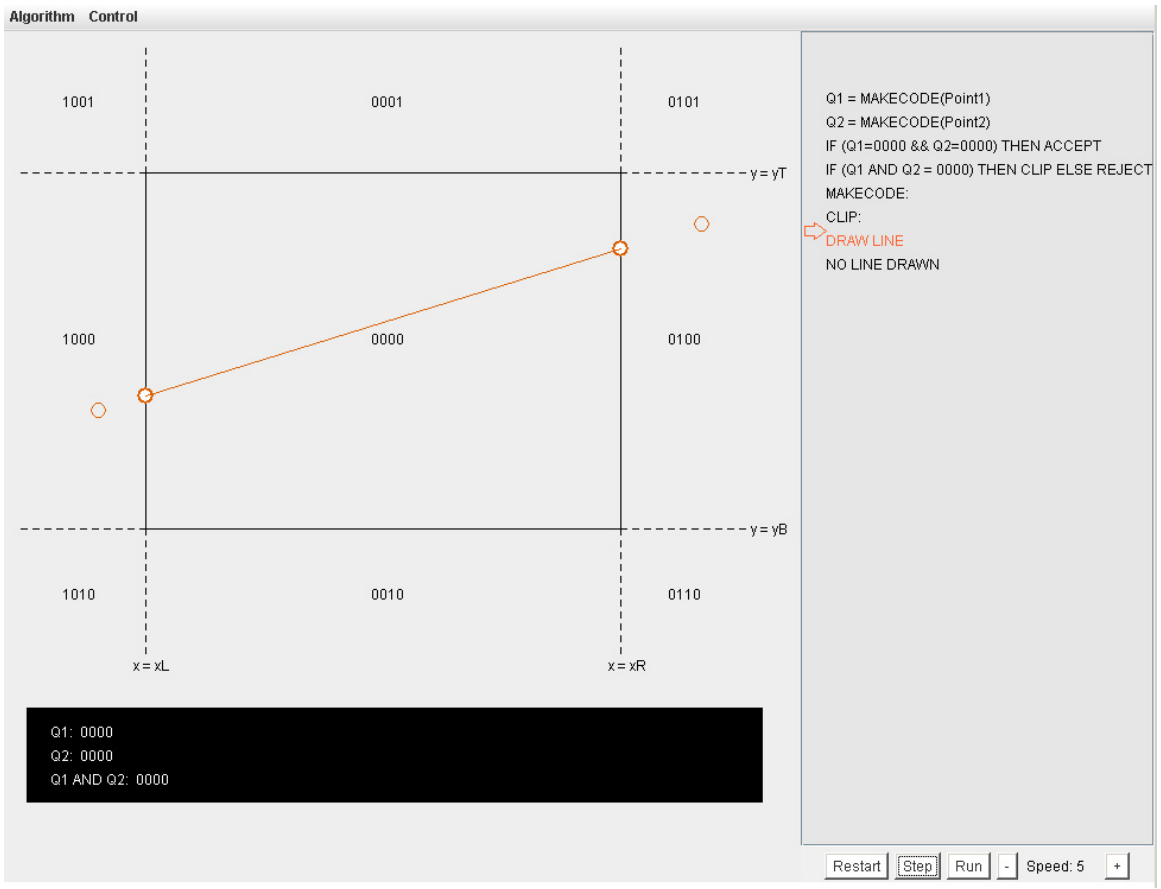


Figure 3.10: The Cohen line clipping algorithm.

3. Clip against this edge. When calculating the coordinates of the new point, one can be trivially retrieved from the clip edge and the other is calculated using coordinate geometry - this requires considering the other point.

The source code for makeCode() and clip() can be found in Appendix A.

3.6 Polygon Scan-line Filling

Presentation Decisions & Parameter Setting

I decided to present a single polygon on a background grid defined by eight vertices — represented as ever using CircleEndPoints. The user can move the position of any of these endpoints to change the polygon's shape and size. I decided that eight vertices was sufficient for creating varied polygons and decided against providing functionality for changing this number. This would have been possible to implement without requiring any changes to the core algorithm but would have required additional control tools. I decided it did not offer enough added functionality to merit the effort.

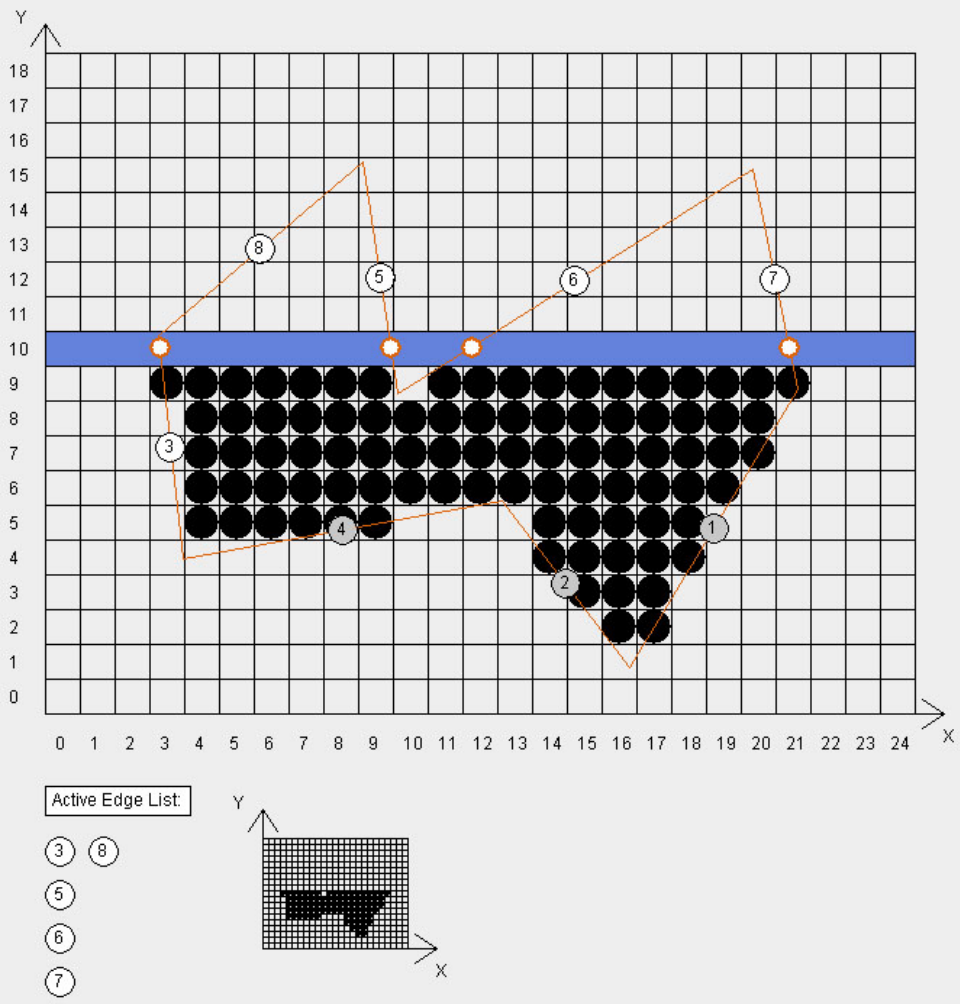


Figure 3.11: The final display for the scan-line polygon fill algorithm.

There were other graphical elements to display once the algorithm was running, the most important being the active edge list (AEL). I decided to label each of the edges of the polygon with a unique number to create a method of easily identifying which were currently active: The labels are coloured differently to represent whether their edge is currently in the AEL or not. I also reused techniques first seen in Bresenham Line Drawing; a blue coloured current scan-line and a thumbnail grid to display the progress at a finer resolution. The result is that once in motion, this algorithm is one of the most striking and perhaps benefits most from the style of presentation I have developed. Figure 3.11 demonstrates the final result.

Implementation Details

Here I will outline some of the details behind the successful graphical exterior. The first thing to note is the use of three separate CoordinateSystems in the implemen-

tation. The first, *cs*, represents the magnified grid of (enlarged, Paintable) pixels and was used to automatically generate the background grid. The second, *es*, is used as the system used to draw the vertices and edges of the polygon and as such its units are single pixels. In a sense the algorithm is all about how to represent the actual polygon as accurately as possible in the alternative CoordinateSystem *cs*. The third CoordinateSystem, thumb, is used for the creation of the reduced-magnification 'thumbnail' graphic displaying the algorithm's progress.

The single most important field in the algorithm is scanLine—an integer value representing the current scan-line. I needed to have the notion of a polygon edge, so I created a (Paintable) inner-class Edge. This class contains references to the two CircleEndPoints defining the edge, as well as a field for the name of the edge (which in reality will mean a unique positive integer) and a Boolean field used to determine whether the name should be painted as a label next to the edge. The class also contains a paint() method used by the display framework to draw the edge and its label.

I used Edge objects in the inner-classes EdgeList and ActiveEdgeList, the second of these being a sub-class of the first. Appendix A contains the full source code for the EdgeList and ActiveEdgeList classes but I include an overview of their functionality here. The first thing to note is that as well as encapsulating the methods which drive the actual algorithm, they are also 'Paintable' objects containing the relevant code to transform their embedded state into a graphical form for visualising the algorithm. This marriage of the algorithmic details with the code for painting graphics proved itself to be a key advantage of the system I adopted as it meant relevant state information for objects could be kept and shared in one place.

The EdgeList class contains a protected ArrayList of Edge objects—the edges contained in the list—as well as a collection of methods: mutator methods such as addEdge() and setEdgeLabelsOn() (which are self-explanatory) as well as accessor methods such as getLowestYValue() which returns the lowest y coordinate of all the edges in the list and is used to obtain the initial scan-line used to start the algorithm. The method sort() sorts the edges by y-value and the method getEdgesInterected() returns an ArrayList of any new edges which intersect the current scan-line.

Also important is the inner-class PointsList. This class is used at the stage where a set of intersection points between the currently active edges and the current scan-line are generated; it maintains this collection of intersection points. It also contains another sort() method which orders the intersection points by x coordinate and a method fillBetweenPoints(), which 'fills' pixels between the sorted intersection points, painting a line of the polygon.

The class ActiveEdgeList extends EdgeList by adding the method getIntersectionPoints() which generates the PointsList object described above. It also adds the method removeEdgesBelowScanline() which, as the name suggests, removes edges which are below the current scan-line and can no longer be considered active. It also contains the code in its paint() method which visualises the Active Edge List.

We can also look at the implementation at a higher level, in terms of what the Framework actually sees: the *initialise()* and *next()* methods which are called to start and successively step through the algorithm.

The *initialise()* method contains a check for the subtle problem outlined in the preparation; that of a vertex falling exactly on a scan-line. The solution is to shift the vertex in question by a small amount. As usual, this method also generates the pseudo code lines for the algorithm. It also creates the initial EdgeList object from all the edges from the input polygon.

The *next()* method contains the now familiar single switch statement. Figure 3.12 presents the switch statement for the algorithm as a large flow graph demonstrating the close mapping between the pseudo code for each step and the actual java code implemented.

3.7 Polygon Clipping

Presentation Decisions & Parameter Setting

The presentation of the polygon is identical to that of the previous algorithm other than that the background grid is replaced by a single clipping rectangle. Parameter setting is handled in an identical manner with eight vertices again sufficient for creating varied polygons of different styles to be clipped.

I also implemented a VariableDisplay to keep track of the vertex count for successive input and output polygons. However, the main tool for successfully visually representing the algorithm turned out to be the ability to change the colours and styles of the CircleEndPoints to help represent which vertices had been 'visited', were yet to be 'visited', had been added to the output polygon and had been rejected. This simple mechanism helped animate the behaviour of the algorithm in a useful way. The final result is displayed in Figure 3.13.

Implementation Details

This was perhaps the most difficult of the algorithms to implement. Once a polygon has been received as input and clipping is in progress there is a period of cycling through the polygon's vertices and applying rules based on the four different cases outlined in the preparation. Thus the identification and processing of these different cases had to be embedded into the switch-case representation of the algorithm, resulting in a large switch statement.

I created a method called getCase() which generates the pseudo line number for the case which applies. The method examines the positions of the current start and end vertices in relation to the current clip edge. Thus by calling setLineNumber(getCase()) I was able to ensure that the line number is changed as appropriate.

I needed some way of defining the clip edges which define the clip rectangle. I chose to define static final integer fields xLeft, yTop etc. representing the x or y

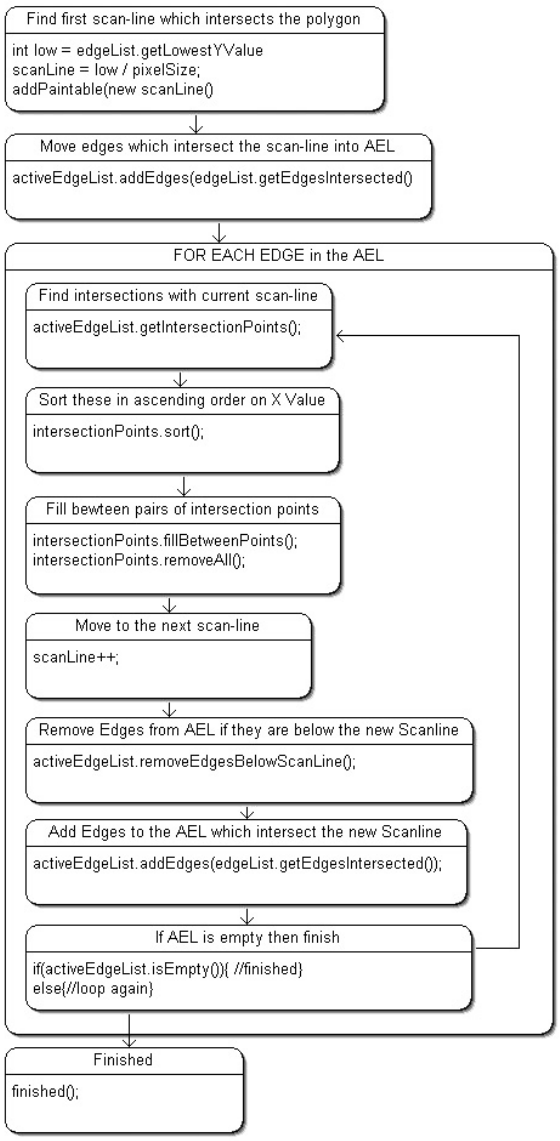


Figure 3.12: The flow of the polygon fill algorithm

Each box represents a pseudo step of the program. At the top of the box is the pseudo code string for the step, while the main body contains how the actual java code for the step. The resultant java code consists mainly of simple method calls to the objects edgeList and activeEdgeList—the real implementation details are embedded in the methods for these classes. For simplicity I have excluded some of the uninteresting code like parenthesis and the break statement at the end of each case.

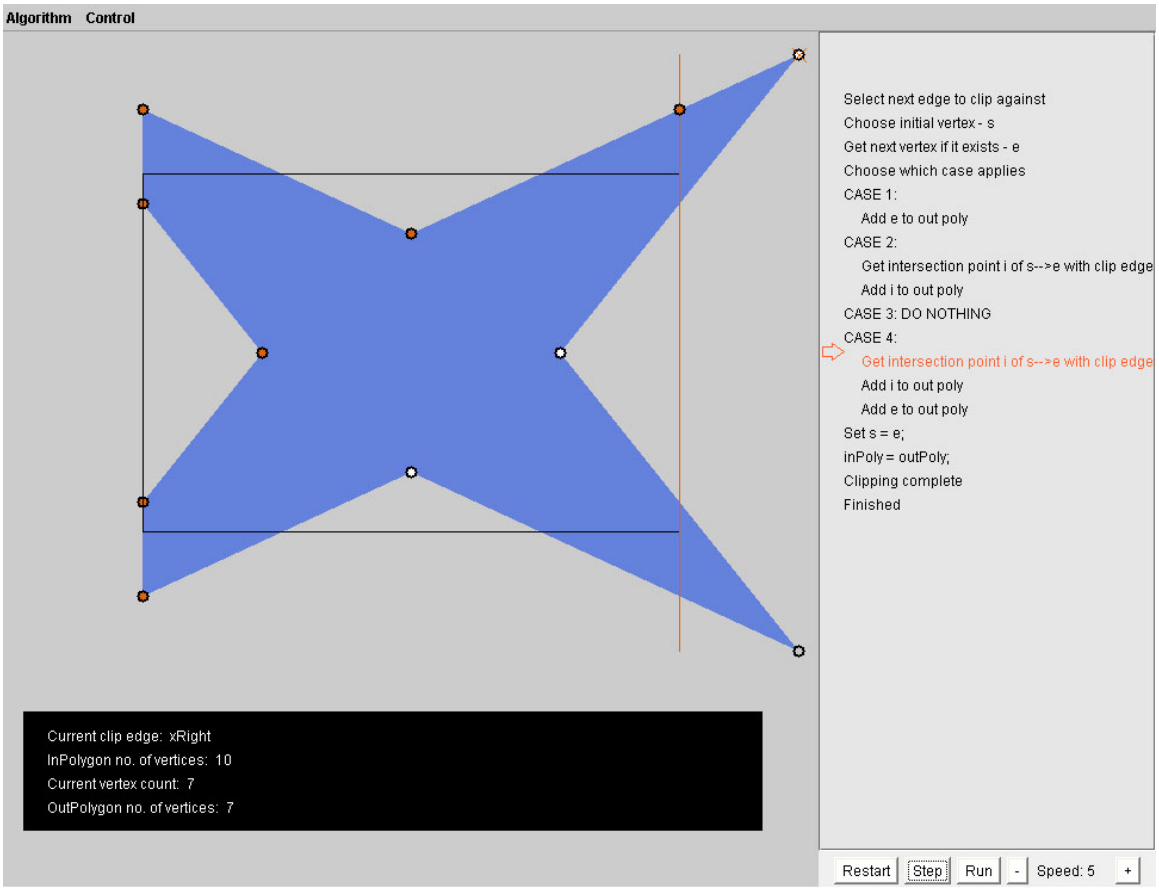


Figure 3.13: The final display for the Sutherland-Hodgman algorithm.

coordinate of the line as appropriate according to the CoordinateSystem used by the algorithm.

The creation of an inner-class called Polygon was required. In the scan-line polygon fill algorithm it was the edges of the polygon that were important to consider. However this algorithm takes one 'polygon' as input and returns a different one as output after each of its successive clipping stages. Thus it was necessary to have some way of representing a polygon as an individual object. The class maintains an array of vertices defining the polygon. The class extended Paintable meaning it also handles the actual drawing of the polygon.

The initial input polygon to the algorithm is an instantiation of the Polygon class using the CircleEndPoints which the user has positioned to define the polygon of their choosing. This Polygon is named inPoly. During the first execution of the algorithm—the clipping of inPoly against the first of our clip edges—a new Polygon object, outPoly is created with the relevant vertices added as inPoly is processed. Then at the end of this, outPoly becomes the next inPoly and the process begins again with the next clip edge until the clipping is complete.

The methods addVertex() and getVertices() in the class Polygon, along with the

method clear() which clears the 'contents' of the polygon, are used during the process of constructing outPoly and ultimately switching inPoly to outPoly. Also included are methods such as setVerticesPlain(), setVerticesStart() and setAllRed() which are important methods used to set the colours of the Polygon's vertices to assist the visualisation process.

Occurrences of cases 2 and 4 during the clipping process require the calculation of a new vertex positioned at the intersection of the clip edge with the line between the two current vertices. The method intersection(), returning a CircleEndPoint, performs this task by performing the necessary coordinate geometry.

An automatically generated grid based on the CoordinateSystem in use was not appropriate for visualising the algorithm, so a Paintable inner-class SuthHodgeGrid was created to represent the clipping box. This class also handled the drawing of the current clipping edge when relevant.

Chapter 4

Evaluation

4.1 Testing

By their visual nature, graphics algorithms are often most effectively tested by simple observation. In the case of my project debugging is made easier still; I have built in functionality to step through the algorithms and visualise the output at every intermediate stage. It is often possible to evaluate exactly where an algorithm goes wrong, or indeed whether it is doing anything right at all in the first place. One might say that the whole purpose of this project was to provide an environment for 'testing out' the algorithms; to verify that they work and hopefully to gain understanding of why they work.

Thus the testing of the actual implementations of the algorithms was straightforward. It was sufficient to test each algorithm by trying out every conceivable variation in setting the parameters. There were only a limited number of possibilities for algorithms such as line drawing or line clipping. Even with the polygon clipping algorithm it was possible to establish the set of possible input polygon types so that every eventuality could be tested.

Of course the real opportunity for eliminating bugs came during development. There were occasions when the results clearly didn't look right, and a solution was worked towards during the implementation itself.

4.2 Evaluating the System as a Whole

For the purpose of evaluation I consider the Java applet version of the project as this is the form most suited to an e-learning tool and feedback I have received relates to this version [1].

As a whole, the system performs well. All the algorithms run smoothly at each of the available speeds. Furthermore the system is extremely stable. Algorithms can be started, stopped, restarted and switched between without any problems. Basic

[1] You can try it out for yourself at http://www.chu.cam.ac.uk/~ae250

functionality like moving endpoints to set parameters is handled in a simple and effective manner.

The greatest strength of the system is perhaps its straightforwardness. It is simple to use, visually appealing and clear in its teaching. The system has been used by Dr Neil Dodgson in his graphics lectures to demonstrate the algorithms featured, which represents a major success; the system at work in the environment it was created for, demonstrating how computer graphics algorithms work in a clear, engaging, interactive manner which goes beyond what is possible with traditional teaching methods.

4.3 Evaluating the Algorithms

Line Drawing

Line drawing is the classic 'first graphics algorithm' to be taught and it proved to be the ideal first algorithm to implement. It is simple in aim yet subtle enough to require a detailed exploration. It also looked good in motion and provided me with an immediate answer to the questions of what my system might need to do and how it could actually be of use.

The implementation of the Bresenham method is particularly effective because it is well suited to step by step animation. The Midpoint method is also successful.

Line Clipping

In retrospect this was a less interesting algorithm to implement. This is because, graphically it seems obvious what should be done: if an endpoint lies outside the clipping box then it should be moved along the line until it is inside the clipping box. The subtlety in the algorithm is the efficient manner in which the task and the necessary calculations are performed, but it was never clear how this could easily be visualised. My implementation works perfectly and demonstrates exactly how the algorithm should work. The problem is that this in itself is neither difficult nor hugely interesting. The algorithm's output for some obvious test cases can be seen in Figure 4.1.

Bezier Curve Drawing

As with line clipping, this algorithm is conceptually fairly simple. My implementation is useful in highlighting the procedure of splitting the Bezier in two, but is otherwise fairly brief.

There was a problem of how to present this algorithm given its recursive nature. I am satisfied with the approach I took, but I think perhaps it limited what I was able to show with the algorithm. A different approach might be to slowly zoom in on the curve following through successive procedure calls for part of the curve. In this way it might also be possible to visualise the flatness test. I suspect this would

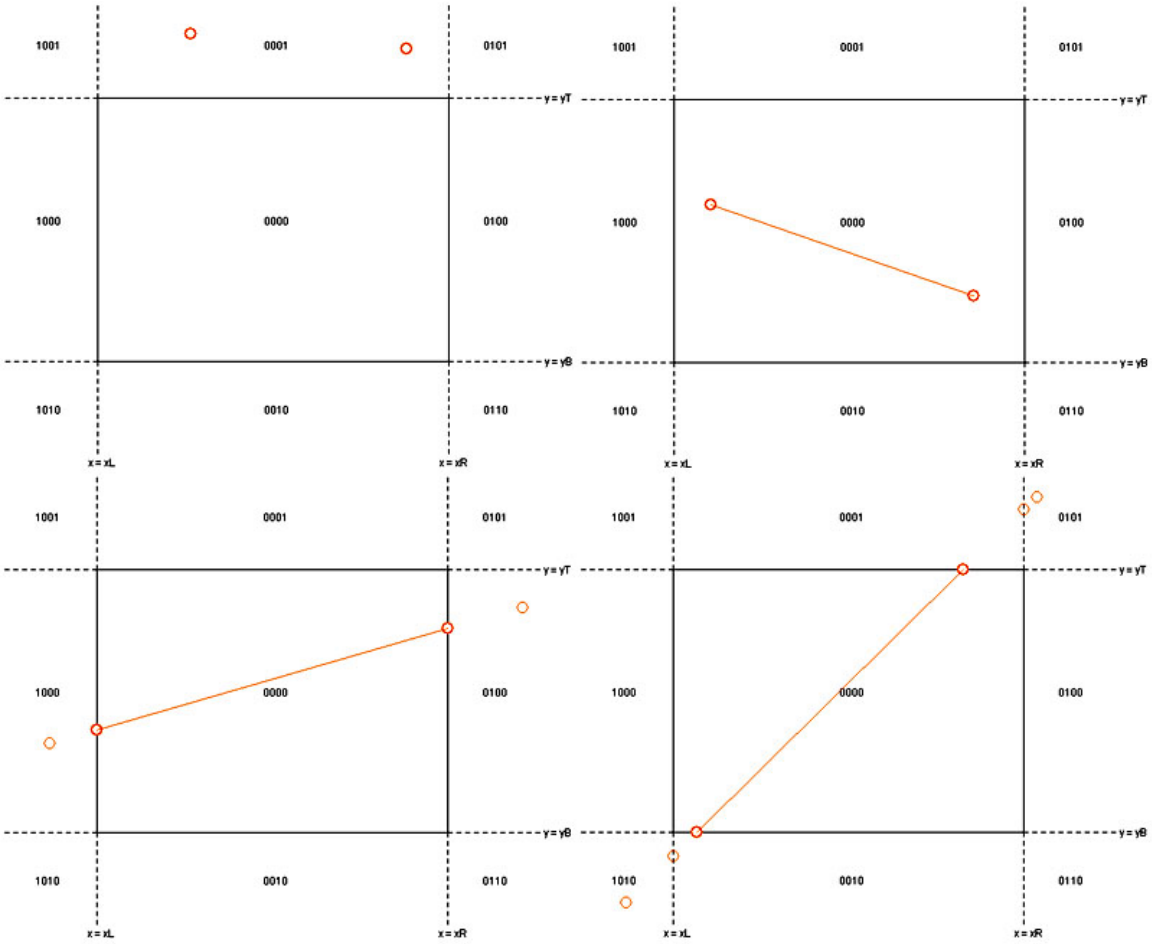


Figure 4.1: Successful line clipping in four different cases

produce a more engaging demonstration of the algorithm, although it would have been significantly more difficult to implement.

Moving back to my actual implementation, a problem was that due to the recursive nature of the algorithm I experienced stack over flow errors when I was experimenting with low tolerances to get a very fine smooth curve. This meant that I had to cap the lowest tolerance to avoid such problems, resulting in a curve less smooth than I would have ideally liked.

Despite these shortcomings, the ability to vary the tolerance and to experiment with the positions of control points leaves an implementation of value to someone who has never encountered Bezier curves before. It would serve as a basic introduction.

Polygon Filling

This implementation was a great success. While the algorithm is not hugely complex, it is hugely beneficial to be able to see it at work to get an understanding of the order in which it works. The visualisation of the Active Edge List was particularly

effective—the labelling of each edge in different colours means it is possible to see the removal and addition of each edge to the list as it happens.

The algorithm performs correctly for any possible input polygon, including convex and concave polygons and also complex polygons which cross themselves multiple times. Some test cases are demonstrated in Figure 4.2.

It was also possible to test the algorithm in the special case where a vertex falls exactly on scan-line by manually setting the default vertex positions within the implementation. The algorithm deals with this by shifting the vertex slightly as required.

Polygon Clipping

Like Polygon filling, this algorithm proved to be ideally suited to the system I had implemented. The colouring of vertices to reflect their current state was an effective way of animating the algorithm's progress.

I was able to verify that the algorithm functions correctly for all types of polygon including complex polygons and cases where the clipping process results in two or more entirely separate polygons as shown in Figure 4.3.

4.4 Feedback

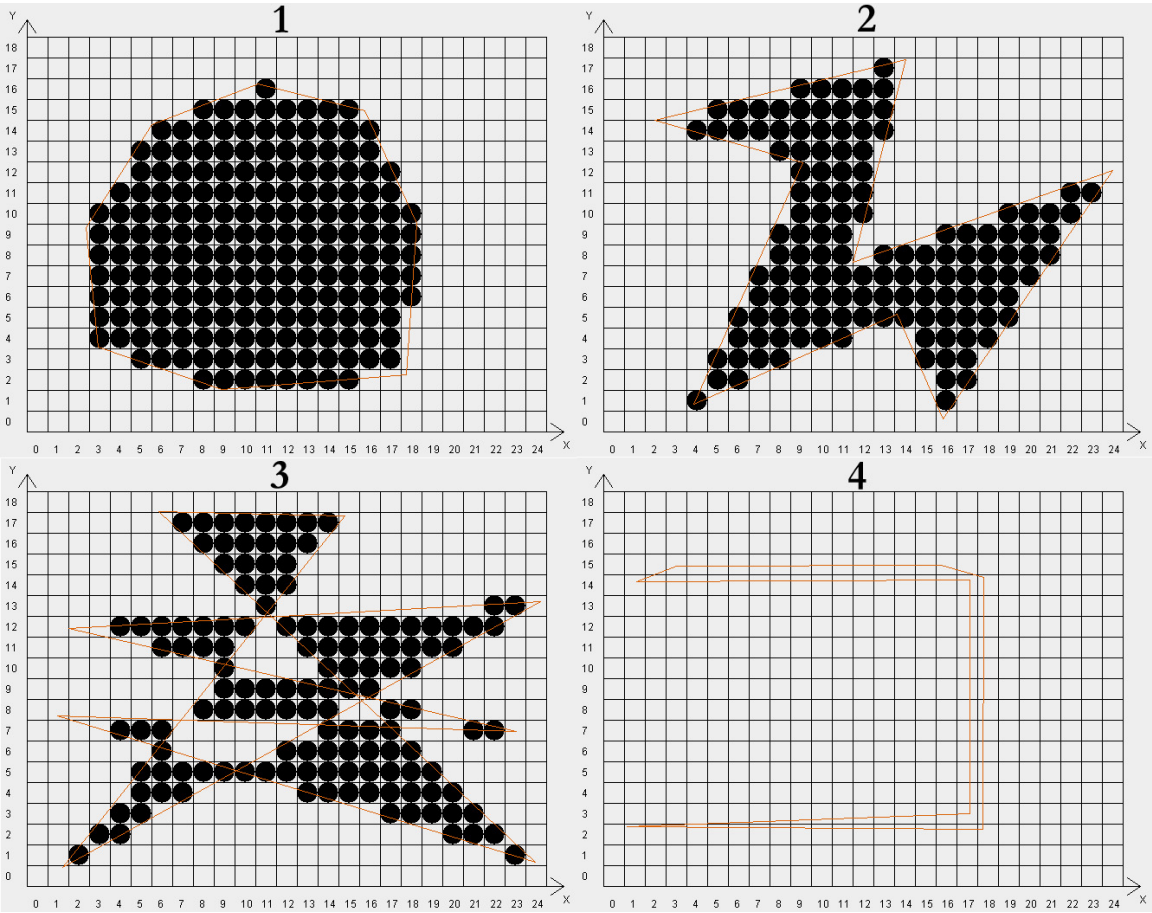
A link to the web applet version has been included on the computer lab teaching webpage for the IB course Computer Graphics & Image Processing. The website includes an invitation for students to provide feedback about the system and whether it is useful to them. At the time of writing two IB students have contacted me. Both indicated that they had found the tool helpful. One said 'generally it is intuitive to use, and demonstrates the algorithms'. Both commented on a problem with the material not fitting entirely on one window at lower screen resolutions. I was able to address this problem in part by moving the controls for running the algorithm to eliminate the need for scrolling.

Dr. Neil Dodgson, who used the tool in his lectures, made the following comment:

"It is extremely useful to have an easy-to-use tool for demonstrating six of the key algorithms taught in my course. The live demonstrations were a great asset and student feedback was positive."

4.5 Evaluation of extensibility

The system is not easily extensible for a typical user. It requires programming expertise to create additional content. This is not a failing—my initial specification was not to make a tool for implementing algorithms, rather a collection of useful and complete implementations. Given that the aim of the tool is to teach how the algorithms really work, asking for the developer to be comfortable with how they might be coded does not seem unreasonable.



1. A convex polygon
2. A concave polygon
3. A complex polygon. This is filled correctly according to the algorithm's definition.
4. A slither polygon which results in no pixels being filled

Figure 4.2: Testing the scan-line polygon fill algorithm.

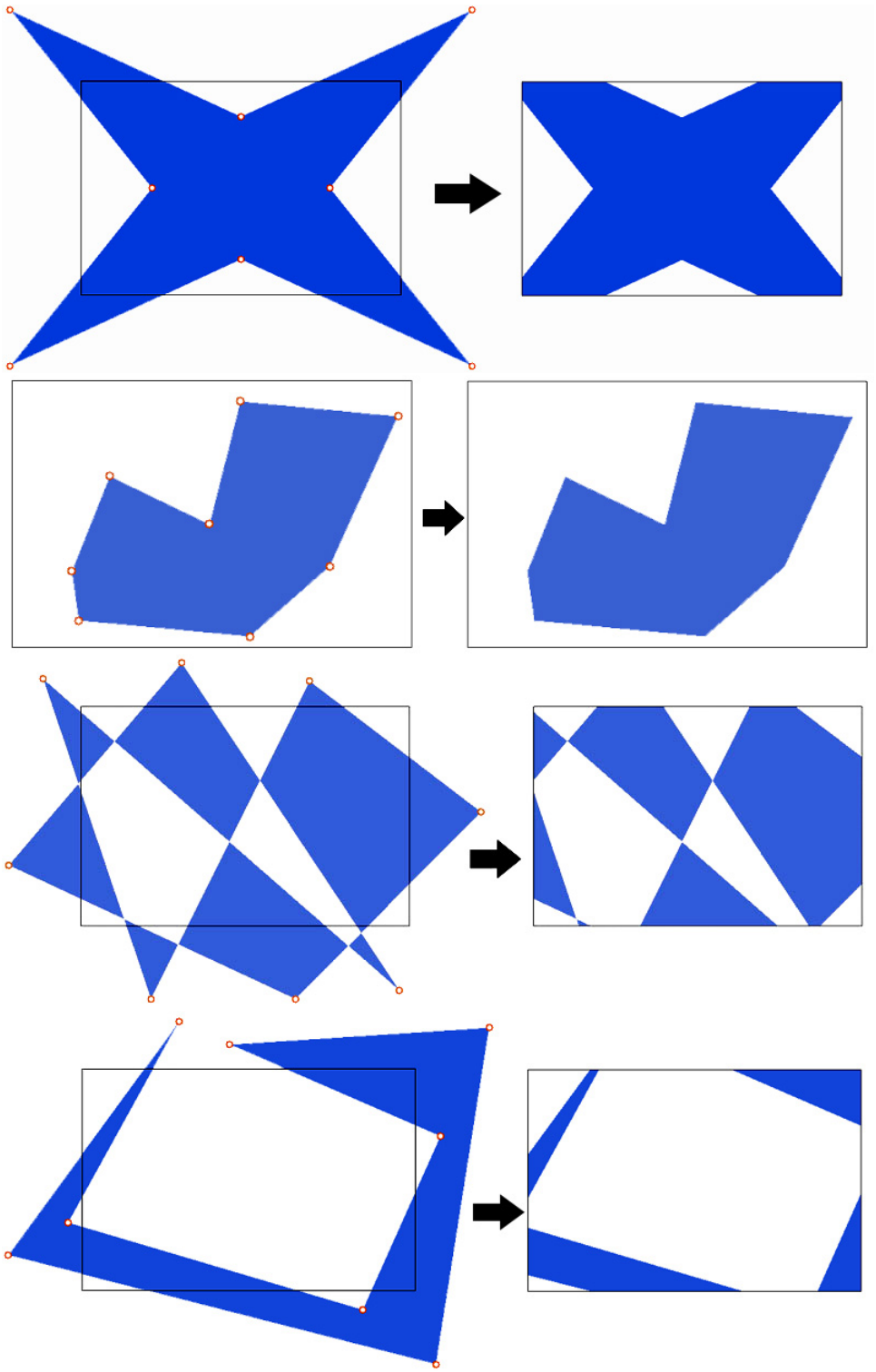


Figure 4.3: Testing the polygon clipping algorithm.

Nevertheless, the prospect of a more automated environment for implementing such algorithms is intriguing. The set of tools I used in creating my implementations are useful and would significantly speed up the process of implementing further algorithms of a similar nature but again only for a competent computer programmer. The question of how these tools could be integrated into an environment specifically geared to the creation of new content is challenging.

4.6 Evaluation of my approach

My approach was to take a collection of algorithms and implement them in a manner suited to explaining their behaviour with clarity. It has been established that I was extremely successful in doing just this. The problem with this approach was that the algorithms themselves drove the creation of the underlying framework. With the benefit of hindsight, if extensibility and flexibility are to be considered of high importance (presumably they would have to be in the context of commercial enterprise) it would perhaps have been more sensible to let the design of the underlying framework drive all else. Ultimately it is unsurprising that my system provides little flexibility beyond what has already (successfully and colourfully) been implemented; it was this very content which drove the design process.

Once flexibility and extensibility are removed from the equation, I would adopt a more optimistic evaluation. The content driven approach brought plenty of success; I believe that the individual implementations of several of the algorithms are superior to any other existing implementations freely available on the web.

Chapter 5

Conclusion

5.1 Completed Work

My original aim was to create a tutor incorporating multiple algorithms in a single interactive environment, with consistency in the presentation and ease of use in controlling and switching between the algorithms. This aim has been met. I implemented six algorithms in a single interactive environment. The system is easy to use and consistent across all the individual algorithms.

I originally stated that the algorithms should be presented in a manner conducive to viewing them in action, with a focus on interactivity and visualisation. I stuck to this guiding principle throughout the implementation. The tutor has been used in IB graphics lectures to demonstrate the algorithms covered and has received a positive response.

In the process of the development I gained a thorough understanding of what is required to put together an interactive tutor. I also developed useful utilities which could possibly be built upon to create a more complete e-learning system in future.

5.2 Future Work

There are some obvious improvements which could be made to my system:

1. The simultaneous demonstration of two different algorithms solving the same problem. The obvious example is to compare the Bresenham and Midpoint line drawing algorithms.

2. The ability to step through algorithms in both directions. This would require a reworking of the basic underlying framework.

3. The Polygon filling algorithm would benefit from a small image displaying the case being applied to the current start and end vertices as the algorithm progresses round the polygon.

Moving beyond my own system and into the wider area of visually and interactively demonstrating algorithms in general, some other possible future work became apparent. It became clear as I worked is that visualisation techniques are the key to successfully demonstrating an algorithm interactively. A robust working implementation of an algorithm is not enough—one needs to think carefully about how the behaviour can be visualised.

The development of a library of visualisation tools and techniques could be created to help with this process. If I were able to start this project all over again while retaining all the experience I have gained from my efforts, I would probably be driven by a different vision. I would try to create a much more powerful and more general underlying framework, incorporating multiple modes of operation and a significant library of visualisation utilities. Algorithms of different shapes and sizes would be able to run in this framework, taking advantage of different parts of the system's overall functionality.

An exciting further step would be to take such a framework and turn into an environment for implementing and visualising algorithms. This would perhaps be a system where algorithms could be entered in the form of either java code or pseudo code and then associated graphics content could quickly be built up using the library of utilities and associating graphics with fragments of code.

This could also be extended to a system where students could enter their own code and build their own algorithms. Ultimately it might be possible to create a simple language for quickly generating graphics content.

5.3 Final Word

I found implementing a system form scratch to be a rewarding experience. I believe that computer learning systems can be of real importance to education. I hope that my system might be of some use to graphics students in the years to come, and look forward to it being built upon either by myself or others.

Bibliography

[1] Akbar Datoo. Interactive teaching of computer graphics. Part II Project, 2000.

[2] Neil A. Dodgson. Computer graphics & image processing, 2005.

[3] James D. Foley et al. *Computer Graphics Principles and Practice*. Addison-Wesley, 1990.

[4] Ingmar Peter and Stefan Gumhold. Teaching computer graphics with java 3d. Technical report, University of Tubingen, 1999.

[5] Richard Powell. Interactive teaching of 3d computer graphics. Part II Project, 1998.

[6] Torsten Ullrich and Dieter W. Fellner. Algoviz a computer graphics algorithm visualization toolkit. Technical report, Institute of Computer Graphics, Braunschweig University of Technology, Mühlenpfordtstr, 2003.

[7] Joaquin A. Vila and Janet D. Hartman. Teaching introductory graphics with web assistance. In *31st ASEE/IEEE Frontiers in Education Conference*, 2001.

Appendix A

Source Code

A.1 CoordinateSystem Conversion Methods

```
public int realToSystemX(int x){
  int xNew = ((x - originX)/pixelsPerUnit);
  if(xNew < 0) xNew = 0;
  if(xNew > maxX)xNew=maxX;
  return xNew;
}

public int realToSystemY(int y){
  int yNew = ((originY - y)/pixelsPerUnit);
  if(yNew < 0) yNew = 0;
  if(yNew > maxY)yNew=maxY;
  return yNew;
}
```

A.2 Bresenham Line Drawing Class

```
public void initialise() {
  // determines the value of xGrad and thus how the aglorithm proceeds
  bresenhamLineDrawing();

  // adds the relevant pseudo code depending on xgrad
  // n1, n, mi and m are set to hold x or y values depending on xgrad
  // i.e. depedning on the nature of the line to draw
  addPseudoCode(0,"Inititial pixel: DRAW(x,y)");
  addPseudoCode(1,"");
  if(xGrad){
    addPseudoCode(2,"WHILE x < x1");
    addPseudoCode(3,"      x = x + 1");
    addPseudoCode(4,"      yi = yi + d");
    addPseudoCode(5,"      y = ROUND(yi)");
    addPseudoCode(6,"      DRAW(x,y)");
```

```
      addPseudoCode(7,"END WHILE");

      n1 = x1;
      n = x;
      mi = yi;
      m = y;
    }
    else{
      addPseudoCode(2,"WHILE y < y1");
      addPseudoCode(3,"      y = y + 1");
      addPseudoCode(4,"      xi = xi + d");
      addPseudoCode(5,"      x = ROUND(xi)");
      addPseudoCode(6,"      DRAW(x,y)");
      addPseudoCode(7,"END WHILE");

      n1 = y1;
      n = y;
      mi = xi;
      m = x;
    }
    addPseudoCode(8,"");
    addPseudoCode(9,"FINISHED");
    setLineNumber(0);
    vd.setVariables(recalculateVariables());
  }

  private void bresenhamLineDrawing(){
    int p1x = startPoint.getX();
    int p1y = startPoint.getY();
    int p2x = endPoint.getX();
    int p2y = endPoint.getY();

    double s1 = p2y-p1y;
    double s2 = p2x-p1x;

    xGrad = true;

    //Rearrange points so p1 is furthest left
    if(p1x>p2x){
      int p3x = p1x;
      int p3y = p1y;
      p1x =p2x;
      p1y =p2y;
      p2x = p3x;
      p2y = p3y;
    }

    d = s1/s2;
```

```
    //this gradient fine if it is betwen -1 and 1
    //otherwise use y gradient

    //Rearrange points so p1 is lowest
    if(d>1||d<-1){
      if(p1y>p2y){
        int p3x = p1x;
        int p3y = p1y;
        p1x =p2x;
        p1y =p2y;
        p2x = p3x;
        p2y = p3y;
      }

      d =s2/s1;
      xGrad=false;
    }

    x0 = p1x;
    y0 = p1y;
    x1 = p2x;
    y1 = p2y;

    x=x0;
    y=y0;
    yi=y0;
    xi=x0;
  }

  public void next() {
    int lineNo = getLineNumber();
    // This is the switch case statement containing the core of the algorithm
    // Each case represents a different 'step' of the aglorithm
    // The current step is determined by the current line number lineNo
    switch(lineNo){

    case 0: {
      //DRAW THE INITIAL PIXEL
      //This graphics stuff to display:
      addPaintable(new Pixel(x,y,cs,false));
      //This new Pseudo Line Number:
      setLineNumber(2);
      break;
    }

    case 2: {
      // Test for y < y1
      if(n<n1){
```

```
        setLineNumber(3);
        break;
      }
      else{
        setLineNumber(7);
        break;
      }
    }

    case 3: {
      // y = y + 1
      n++;
      setLineNumber(4);
      break;
    }

    case 4: {
      // xi = xi + d
      mi+=d;
      setLineNumber(5);
      break;
    }

    case 5: {
      // Set x = ROUND(xi) - +0.5 to turn into true round
      m=(new Float(mi+0.5)).intValue();
      setLineNumber(6);
      break;
    }

    case 6: {
      // Draw(x,y)
      if(xGrad){
        //draw pixel(m,n)
        addPaintable(new Pixel(n,m,cs,false));
        addPaintable(new Pixel(n,m,thumb,false));
      }
      else{
        // draw pixel(n,m)
        addPaintable(new Pixel(m,n,cs,false));
        addPaintable(new Pixel(m,n,thumb,false));
      }
      setLineNumber(2);
      break;
    }

    case 7: {
      // y >= y1 so  go to Finish
```

```
      setLineNumber(9);
      break;
    }

    case 9: {
      finished();
      break;
    }

    }
    // Recalculate variable values in Varaible Display
    vd.setVariables(recalculateVariables());
  }
```

A.3 Midpoint Line Drawing Class

```
public void initialise() {
    //Rearrange points so startPoint is leftmost
    if(startPoint.getX()>endPoint.getX()){
      switchPoints();
    }

    double s1 = endPoint.getY()-startPoint.getY(); //could be negative
    double s2 = endPoint.getX()-startPoint.getX(); //must be positive

    double grad = s1/s2;

    // Here we are using the gradient to establish which case applies
    if(grad>1){
      quadrant = 1;

      if(startPoint.getY()>endPoint.getY()){
        switchPoints();
      }
    }
    else if(grad>0)quadrant = 0;
    else if(grad<-1){
        quadrant = 2;
        if(startPoint.getY()>endPoint.getY()){
          switchPoints();
        }
      }
    else quadrant = 3;

    x0 = startPoint.getX();
    y0 = startPoint.getY();
    x1 = endPoint.getX();
```

```
y1 = endPoint.getY();

// The values for use in the algorithm are calculated
a = y1-y0;
b = -(x1-x0);
c = x1*y0-x0*y1;
x = x0;
y = y0;

//This pseudo code instruction is common to all the cases
addPseudoCode(0,"Inititial pixel: DRAW(x,y)");

//We set the value of d as appropriate, and add the correct pseudo code
//based on the case we are dealing with (defined by quadrant)
switch(quadrant){
  case 0:{
    dp.setX(x+1.0);
    dp.setY(y+0.5);
    d = a * (x+1) + b *(y+0.5)+c;
    addPseudoCode(1,"WHILE x < x1");
    addPseudoCode(2,"      x = x + 1");
    addPseudoCode(3,"      IF d<0 THEN");
    addPseudoCode(4,"            d = d + a");
    addPseudoCode(5,"");
    addPseudoCode(6,"      ELSE");
    addPseudoCode(7,"           d = d + a + b");
    addPseudoCode(8,"           y = y + 1");
    addPseudoCode(9,"      DRAW(x,y)");
    addPseudoCode(10,"END WHILE");
    break;
  }
  case 1:{
    dp.setX(x+0.5);
    dp.setY(y+1.0);
    d = a * (x+0.5) + b *(y+1)+c;
    addPseudoCode(1,"WHILE y < y1");
    addPseudoCode(2,"      y = y + 1");
    addPseudoCode(3,"      IF d<0 THEN");
    addPseudoCode(4,"            d = d + b + a");
    addPseudoCode(5,"            x = x + 1");
    addPseudoCode(6,"      ELSE");
    addPseudoCode(7,"            d = d + b");
    addPseudoCode(8,"");
    addPseudoCode(9,"      DRAW(x,y)");
    addPseudoCode(10,"END WHILE");
    break;
  }
  case 2:{
```

```
        dp.setX(x-0.5);
        dp.setY(y+1.0);
        d = a * (x-0.5) + b *(y+1)+c;
        addPseudoCode(1,"WHILE y < y1");
        addPseudoCode(2,"      y = y + 1");
        addPseudoCode(3,"      IF d<0 THEN");
        addPseudoCode(4,"            d = d + b");
        addPseudoCode(5,"");
        addPseudoCode(6,"      ELSE");
        addPseudoCode(7,"            d = d + b - a");
        addPseudoCode(8,"            x = x - 1");
        addPseudoCode(9,"      DRAW(x,y)");
        addPseudoCode(10,"END WHILE");
        break;
      }
      case 3:{
        dp.setX(x+1.0);
        dp.setY(y-0.5);
        d = a * (x+1) + b *(y-0.5)+c;
        addPseudoCode(1,"WHILE x < x1");
        addPseudoCode(2,"      x = x + 1");
        addPseudoCode(3,"      IF d<0 THEN");
        addPseudoCode(4,"            d = d + a - b");
        addPseudoCode(5,"            y = y - 1");
        addPseudoCode(6,"      ELSE");
        addPseudoCode(7,"            d = d + a");
        addPseudoCode(8," ");
        addPseudoCode(9,"      DRAW(x,y)");
        addPseudoCode(10,"END WHILE");
        break;
      }
    }
    // We set the 'program counter' to the initial line
    setLineNumber(0);
    // We add the paintable object representing the decision point
    addPaintable(dp);
  }
  class DecisionPoint extends Paintable{
    // Coordinates for the dp
    double x;
    double y;

    //The paint method. Draws the dp as a white point with
    // thick black outline
    public void paint(java.awt.Graphics g){
      g.setColor(Paintable.BLACK);
      g.fillOval(cs.systemToRealX(x)+12,cs.systemToRealY(y)+12,8,8);
      g.setColor(Paintable.WHITE);
```

```
    g.fillOval(cs.systemToRealX(x)+13,cs.systemToRealY(y)+13,6,6);
  }
  //Methods for moving the dp
  public void setX(double x){
    this.x = x;
  }

  public void setY(double y){
    this.y = y;
  }

  public void goE(){
    x=x+1.0;
  }

  public void goNE(){
    x=x+1.0;
    y=y+1.0;
  }
  public void goNW(){
    x=x-1.0;
    y=y+1.0;
  }

  public void goN(){
    y=y+1.0;
  }

  public void goSE(){
    y=y-1.0;
    x=x+1.0;
  }
}
```

A.4 Cohen Line Clipping Class

```
public byte makeCode(int x, int y){
  byte q = 0;
  if(x<xLeft)q+=8;
  if(x>xRight)q+=4;
  if(y<yBottom)q+=2;
  else if(y>yTop)q+=1;
  return q;
}
```

```
  public void clip(){
    //CLIP AGAINST ONE EDGE
    if(q1!=0){
      if((q1&8)!=0){
        y1 = y1 + (y2-y1)*(xLeft-x1)/(x2-x1);
        x1 = xLeft;
      }
      if((q1&4)!=0){
        y1 = y1 + (y2-y1)*(xRight-x1)/(x2-x1);
        x1 = xRight;
      }
      if((q1&15)==2){
        x1 = x1 + (x2-x1)*(yBottom-y1)/(y2-y1);
        y1 = yBottom;
      }
      if((q1&15)==1){
        x1 = x1 + (x2-x1)*(yTop-y1)/(y2-y1);
        y1 = yTop;
      }
      //Adds a clone to track the movement of the points
      addPaintable(new CircleEndPoint(point1,12,cs));
      //Update the Point onscreen
      point1.set(x1,y1);
    }
    else{//q2=! 0
      if((q2&8)!=0){
        y2 = y2 + (y1-y2)*(xLeft-x2)/(x1-x2);
        x2 = xLeft;
      }
      if((q2&4)!=0){
        y2 = y2 + (y1-y2)*(xRight-x2)/(x1-x2);
        x2 = xRight;
      }
      if((q2&15)==2){
        x2 = x2 + (x1-x2)*(yBottom-y2)/(y1-y2);
        y2 = yBottom;
      }
      if((q2&15)==1){
        x2 = x2 + (x1-x2)*(yTop-y2)/(y1-y2);
        y2 = yTop;
      }
      addPaintable(new CircleEndPoint(point2,12,cs));
//      Update the Point onscreen
      point2.set(x2,y2);
    }
  }
```

A.5 Scan-line Polygon Fill Class

```
public class EdgeList extends Paintable {
  protected ArrayList edges;

  public EdgeList(){
    edges = new ArrayList();
  }

  public EdgeList(Edge e0,Edge e1,Edge e2,Edge
        e3,Edge e4,Edge e5, Edge e6, Edge e7){
    edges = new ArrayList();
    edges.add(e0);
    edges.add(e1);
    edges.add(e2);
    edges.add(e3);
    edges.add(e4);
    edges.add(e5);
    edges.add(e6);
    edges.add(e7);
  }

  public void setEdgeLabelsOn(){
    Iterator it = edges.iterator();

    while(it.hasNext()){
      Edge e = ((Edge)it.next());
      e.setLabelEdges(true);
      e.setDarkGrey();
    }
  }

  public void addEdge(Edge e){
    edges.add(e);
  }

  public void addEdges(ArrayList newEdges){
    Iterator it = newEdges.iterator();

    while(it.hasNext()){
      edges.add((Edge)it.next());
    }
  }

  public boolean isEmpty(){
    return edges.isEmpty();
  }
```

```
public void sort(){
  // Sort the edges in order of y-value (least first)
  for (int i = 1; i < edges.size(); i++) {
    int j = i;
    Edge e = ((Edge)edges.get(i));
    int low = e.getLowestY();

    while ((j > 0) && (((Edge)edges.get(j-1)).getLowestY() > low)) {
      edges.set(j,((Edge)edges.get(j-1)));
      j--;
    }
    edges.set(j,e);
  }

  //Rename the edges:
  Iterator it = edges.iterator();
  int i = 1;

  while(it.hasNext()){
    Edge e = (Edge)it.next();
    e.setEdgeName(""+i++);
  }
}

public ArrayList getEdgesIntersected(){
  ArrayList intersectedEdges = new ArrayList();
  Iterator it = edges.iterator();

  while(it.hasNext()){
    Edge e = (Edge)it.next();
    if(e.getLowestY()/24==scanLine){
      intersectedEdges.add(e);
      //ACTIVE - set white
      e.setWhite();
    }
  }

  it = intersectedEdges.iterator();

  while(it.hasNext()){
    Edge e = (Edge)it.next();
    edges.remove(e);
  }

  return intersectedEdges;
}

public int getLowestYValue(){
```

```
    if(edges.isEmpty()){
      return 0;
    }
    else return ((Edge)edges.get(0)).getLowestY();
  }

  public void clear(){
    edges.clear();
  }

  public void paint(java.awt.Graphics g){
    //empty but required as the class extends Paintable
  }
}

public class ActiveEdgeList extends EdgeList {

  public ActiveEdgeList(){
    super();
  }

  public void getIntersectionPoints(){
    Iterator it = edges.iterator();

    while(it.hasNext()){
      Edge e = (Edge)it.next();
      // If highest y point < scanline, don't calc ints. point
      if(e.getHightestY()<(scanLine*24)+12
        || e.getLowestY()>(scanLine*24)+12  ){
        //Ignore this Edge - it is too high or low
        // and does not affect the scanline
      }
      else{
        Point p = e.getIntersectionPoint(scanLine);
        CircleEndPoint c = new CircleEndPoint(p.x,p.y,15,1,es);
        intersectionPoints.addPoint(c);
        addPaintable(c);
      }
    }
  }

  public void removeEdgesBelowScanLine(){

    ArrayList remove = new ArrayList();
    Iterator it = edges.iterator();

    while(it.hasNext()){
      Edge e = (Edge)it.next();
```

```
        if(e.getHightestY()/pixelSize<scanLine){
          remove.add(e);
        }
      }
      it = remove.iterator();

      while(it.hasNext()){
        Edge e = (Edge)it.next();
        edges.remove(e);
        //Appear Grey
        e.setDarkGrey();
      }
    }

    public void paint(java.awt.Graphics g){
      g.setColor(Paintable.WHITE);
      g.fillRect(50,120+pixelSize*yPixels, 100,20);
      g.setColor(Paintable.BLACK);
      g.drawRect(50,120+pixelSize*yPixels, 100,20);
      g.drawString("Active Edge List: ",55,134+pixelSize*yPixels);

      Iterator it = edges.iterator();
      int i = 15;
      while(it.hasNext()){
        Edge e = (Edge)it.next();
        g.setColor(Paintable.WHITE);
        if(i>105){
          g.fillOval(80,20+pixelSize*yPixels+i, 20,20);
          g.setColor(Paintable.BLACK);
          g.drawString(e.getEdgeName(),86,34+pixelSize*yPixels+i);
          g.drawOval(80,20+pixelSize*yPixels+i, 20,20);
        }
        else{
          g.fillOval(50,140+pixelSize*yPixels+i, 20,20);
          g.setColor(Paintable.BLACK);
          g.drawString(e.getEdgeName(),56,154+pixelSize*yPixels+i);
          g.drawOval(50,140+pixelSize*yPixels+i, 20,20);
        }
        i+=30;
      }
    }
  }
```

Appendix B

Project Proposal

Part II CST Project Proposal

Interactive Computer Graphics Teaching Tool

A. Errington, Churchill College

Originator: A. Errington

21 October 2005

Project Supervisor: Dr. Neil Dodgson

Director of Studies: Dr. John Fawcett

Project Overseers: Prof. Andrew Pitts & Prof. Ross Anderson

Introduction

This project involves the implementation of an interactive teaching tool to aid students in their understanding of basic computer graphics algorithms by providing visual demonstrations of the algorithms in action, and the results of altering parameters.

The focus is very much on an interactive method of teaching. Rather than relying on tedious explanations or pen and paper demonstrations ill suited to teaching graphics algorithms, the tool would provide an environment for learning by actions, experimentation and observation. It should be very straightforward to use and clear what is going on without the need for excessive explanation.

The algorithms to cover would be taken from Part 1B Computer Graphics & Image Processing, and the system would be aimed at a typical student taking this course. The basic criterion would be to implement the following algorithms:

1. Line drawing
2. Clipping of a straight line against a rectangle
3. Scanline polygon filling
4. Polygon clipping against a rectangle

There would be the possibility of further implementing one or more of the following algorithms as extensions:

- Bezier curves
- 3D Scanline algorithms
- Ray tracing
- BSP trees

Work that has to be done

A basic system framework and user interface would be developed, in which implementations of the different algorithms would operate. Such a system would be easily extendible due to its modular nature.

The style of interactive learning will vary for different types of algorithm. However, in each case the visual presentation of the algorithm will follow a similar style laid down by the framework:

1. An area of the screen for a graphical representation of the algorithm in action, and for direct mouse driven control of parameters
2. An area for the display of pseudo-code relating to the current behaviour of the algorithm
3. Tools for stepping through the algorithm at different speeds, and setting break points.
4. Tools for the display and editing of the algorithms parameters

The project has the following main sections:

1. A study of the chosen computer graphics algorithms and how they work, and research into common misconceptions and difficulties in understanding these algorithms

2. Design and implementation of the system framework and User Interface
3. Design and implementation of the algorithms and algorithm specific teaching content
4. Testing and evaluating the success of the implementation and its effectiveness as a teaching tool
5. Writing the dissertation

Resources

No special resources will be required. I will use my own personal machine running Windows XP, in conjunction with some use of PWF machines.

Backup Plans

I will back up daily to USB flash drive and CD-R. Additional permanent backups will be made before any significant implementation of new code or changes to existing code.

Starting Point

The program is to be written in Java, which I have working knowledge of from courses in Parts 1A & 1B and from my part 1B group project. All my previous knowledge of computer graphics algorithms comes from Part 1B Computer Graphics & Image Processing. I have some rough implementations of line drawing algorithms in Java which I coded as part of my revision last term, although I do not envisage reusing this code as it was not written to be an interactive step by step animated demonstration.

Timetable

I have timetabled the project in 10 blocks of roughly 2 weeks each as follows (holidays have been counted as equivalent to a single 2 week block of working time):

1. Project proposal
2. 24th October - 4th November 2005 Research of the algorithms and relevant teaching methods. Rapid prototyping of system framework. Design of user interface and interfaces for algorithms
3. 7th - 18th November 2005 Coding of system framework & user interface
4. 21st November - 2nd December 2005 Continuing development of framework, in parallel to implementation of line drawing algorithm
5. 5th December 2005 - 13th January 2006 (Christmas Holiday) Implementation of line drawing and line clipping algorithms
6. 16th - 27th January 2006 Implementation of polygon filling & clipping algorithms
7. 30th January - 10th February 2006 Preparation for progress report. Remaining coding on implementation of polygon algorithms
8. 13th - 24th February 2006 System testing & evaluating, extensions

9. 27th February - 17th Match 2006 Finish testing and draft dissertation
10. 20th March -28th April 2006 (Easter Holiday) Dissertation final draft
11. 29th April - 19th May 2006 3 weeks unallocated at the end of the project to allow for emergency or disaster.

Extending the system by implementing further algorithms will be possible if excellent progress is made with the implementation of the four core algorithms. On the other hand, I have left enough leeway to be able to cope with delays should implementation prove difficult and take longer than anticipated.

Milestones

- 21st October 2005 - Proposal submitted
- 4th November 2005 - Full design specification and choice of tools complete
- 2nd December 2005 - System framework complete with partially working line drawing algorithm in place
- 3rd Feb 2006- Progress report - all algorithms should be implemented and usable
- 10th March - All coding & testing completed
- 7th April - Dissertation first draft complete
- 28th April - Dissertation final draft complete