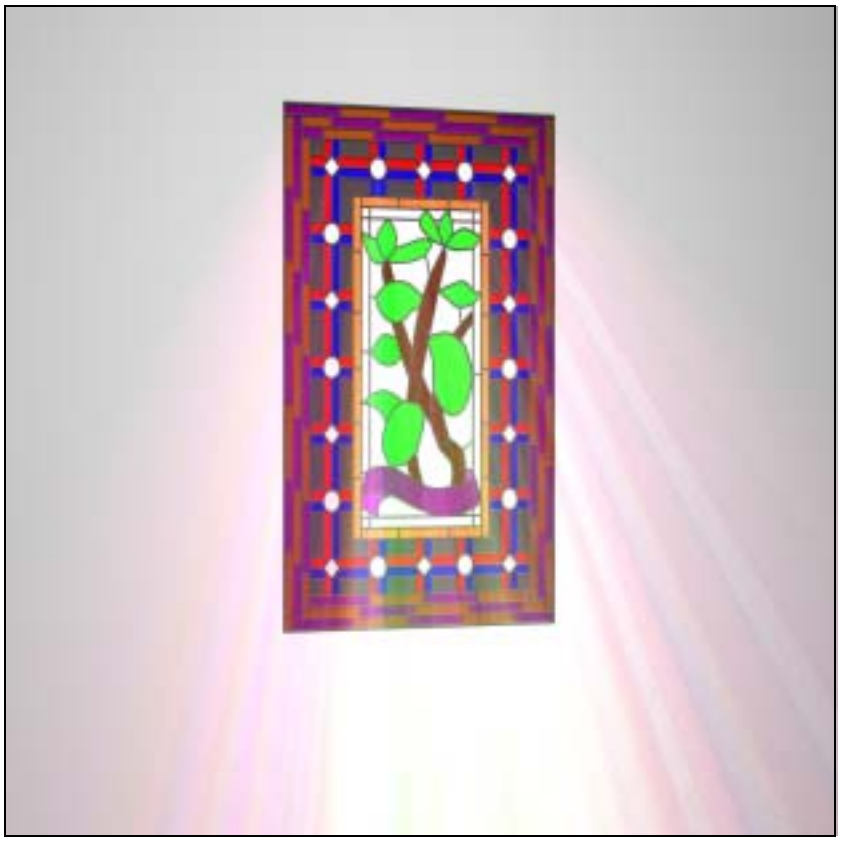
‘Vitrigraph’ Stained Glass Window Designer

Mark S. D. Ashdown
University of Cambridge Computer Laboratory,
J J Thomson Avenue, Cambridge CB3 0FD, UK
www.mark.ashdown.name



Abstract

This dissertation was submitted in 1999 in partial fulfilment of the requirements of the University of Cambridge B.A. Hons Computer Science degree. The aim of the project was to create an application program that can be used to design stained glass windows. The emphasis was on features that assist the physical realisation and financial evaluation of the designs. An extension was implemented to create ray-traced pictures. These allow a design to be evaluated aesthetically, prior to construction. The program has achieved the aims of the project. It provides a simple interface for the non-technical user, to design a detailed window with intuitive manipulation of graphical objects. The novel underlying data structure provides support for algorithms that detect design faults, calculate the production cost based on raw materials and complexity, and produce pleasing and realistic ray-traced images.

Table of Contents

1 Introduction

The production of stained glass windows is a craft that has been practised for centuries. It is now a hobby for individuals as well as being used by companies producing pieces for houses, bars, commercial buildings and restoration projects.

The techniques involved in designing and constructing a stained glass window have been refined over the years. Many books have been written on the subject, craft classes teach the skills and a wide range of specialist tools is available. The application of computing to the field, however, has been limited. Two programs for designing stained glass windows were reviewed while the feasibility of this project was being assessed. The first 'The Glass Eye' makes it easy to produce simple designs, but does not have features for evaluating designs or aiding physical construction. 'American Bevel Designer' is a more complex package, but is suited to producing drawings, rather than making designs from which stained glass windows can be constructed. A bitmap drawing program would be unsuitable. A vector drawing program could be used to create a structured design, but could not store information such as the types of lead and glass, which are essential for financial evaluation. A CAD package could be used, but would not be suited to the specific application, would contain many unnecessary features, and would require the combination of additional programs to allow aesthetic and financial evaluation.

A stained glass window is comprised of individual pieces of glass, cut from sheets of different patterns and colours (figure 1.1). A full-size pattern is drawn and placed on a bench. The glass pieces are cut to the correct shape by placing the glass over the pattern and using a glasscutter to score along the lines. The glass is then snapped along the scores. Traditionally the cutting tip would have been a diamond, but a modern cutter will typically use a tungsten carbide wheel. The stained glass piece is assembled on the bench, with lengths of lead being fixed between the glass pieces to hold them together. The lead is soldered at the joints. Cement is worked into all the cracks to secure the glass pieces, and when it has dried, 'blacking' is used to darken the lead.

The aim of this project was to create a program for designing stained glass windows. Unlike other programs devised for this purpose, the aim of this one was to offer features that assist in the entire process of stained glass window creation from aesthetic design evaluation, to cutting the glass.

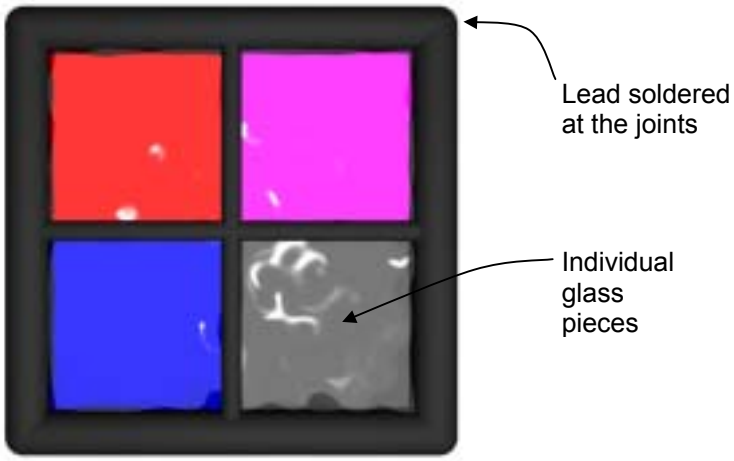


Figure 1.1 A stained glass window

2 Preparation

This section describes the background material and development tools used for the project, and states the criteria formulated to determine successful completion. It then shows the object-oriented design of the whole program, and an overview of the data structures devised to store a stained glass pattern.

2.1 Background Reading

It was necessary to have a thorough understanding of the issues of importance when creating stained glass pieces. This was gained by reading books on the topic [5], studying catalogues of tools and materials from James Hetley Stained Glass Supplies, looking through books of stained glass patterns, and by enlisting the help of an expert: Mr D. H. J. Ashdown owner of Ashdown Sales Ltd, a Cardiff-based company that has been producing stained glass windows for over 50 years, was consulted on issues specific to stained glass. Hetleys was established in 1823 and supplies stained glass tools and materials to amateurs and enthusiasts all over Britain.

Before program design began, Software Engineering techniques were considered [7], especially requirements analysis and object-oriented design.

The storage and manipulation of the stained glass pattern was a major concern in this project. Selection of appropriate methods included choosing how represent the lines of lead and how they fit together in the pattern. Overhauser, Bézier and B-Spline curves were considered as candidates for representing the lines of lead, and their properties were studied [4]. Important details were examined such as the ease with which the lines can be manipulated graphically, the degree of continuity between curve segments, and the way in which the curves can be made to interpolate the defining vertices.

The 'Winged-edge Data Structure' [1] was studied as a basis for representing the whole pattern. This system was originally designed for representing polyhedral 3-dimensional objects for computer vision, where each face was a 2-dimensional polygon with an arbitrary number of straight sides. A similar system was used for this application to represent a 2-dimensional pattern, where each face is a 2-dimensional shape with an arbitrary number of sides that may be straight or curved. The new structure retains the 'winged edge' property whereby each edge is part of up to two faces.

2.2 Development Tools

Java was chosen as the language in which Vitrigraph was written, because it is flexible enough to encode the novel features that were necessary, while also providing the cross-platform compatibility and high-level features important in such an application program. Although Java 1.2 was only available as a beta version when the project was started, it was used because it introduced powerful new graphics features that were useful for displaying the stained glass pattern on the screen. In preparation for the coding, prototypes of important parts of the program were written, such as the routines to display the lines and shapes that form the graphical representation of the stained glass pattern. Standard Java techniques were reviewed [3], and information regarding the new capabilities of Java 1.2 was gained from the online documentation at *http://java.sun.com/*.

The system for backing up the project files was put in place at the start. Scripts for copying the files to the backup server were created so that they could easily be invoked to backup all essential files. The nature of the project meant that the source code, data and documentation constituted a relatively small amount of memory – about 2Mb. The files could therefore be stored after each session of work, forming a series of sets of files holding the stages of development. Each set of files was accompanied by a short explanation of the work that had been done since the last one was saved.

2.3 Requirements Analysis

The program was designed for a user who has only basic computer skills, but is familiar with the design and construction of stained glass windows.

The main stages in the production of a purpose-built stained glass window were identified as:

- Initial input by the customer concerning the general form of the piece
- Design
- Aesthetic evaluation
- Financial evaluation
- Approval by the customer (or a return to the design stage)
- Physical construction

The design stage can be achieved using the software. The aesthetic evaluation, financial evaluation and physical construction can then all be aided by the software using the knowledge it has of the pattern. The fundamental requirements of the program necessary to facilitate these abilities are listed below.

- The user must be able to create a stained glass pattern using simple constructs that are combined graphically.

- The graphical representation of the pattern should relate directly to the intended physical pattern, and incorporate information such as the type of each piece of lead and glass.

- It must be possible to load and save designs from and to files.

- It must be possible to produce printouts of the whole pattern or parts of it, from which the stained glass window can be constructed.

- Financial evaluation routines should compute the cost of producing the window. The routines should give an accurate costing for patterns of varying size and complexity.

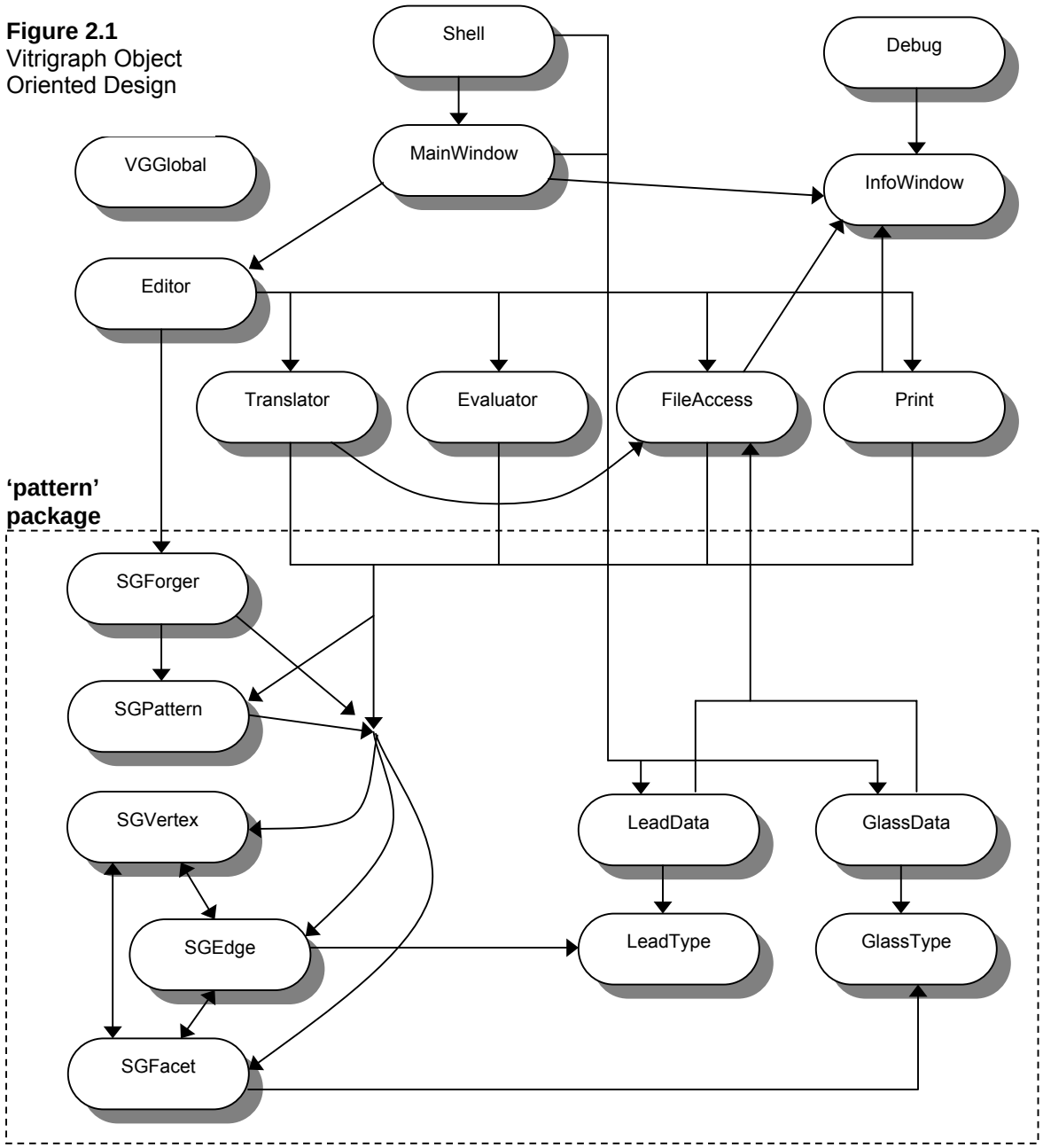
- The program must be able to present the pattern in a form that makes it easy to visualise the finished product. Aesthetic evaluation must be possible.

2.4 Object Oriented Design

The nature of the application and the use of Java, made an object oriented design method appropriate for this project. This approach was used to obtain modularity of program components, and a structure for representing the stained glass pattern that relates directly to the physical object. Figure 2.1 depicts the object-oriented structure of the Vitrigraph program, and the interaction between the objects. Each arrow points from an object, to another object that it accesses.

The objects are implemented directly using Java classes, and grouped into Java packages. The classes that are visible outside their respective packages are shown below. During implementation, it was necessary to include extra classes within the packages as was expected, but these are only used internally and do not affect the interactions shown.

Figure 2.1
Vitrigraph Object Oriented Design



A package called **vitrigraph** contains all of the program code, which is grouped into sub-packages (figure 2.2). The sub-packages and their constituent classes are listed in appendix A.

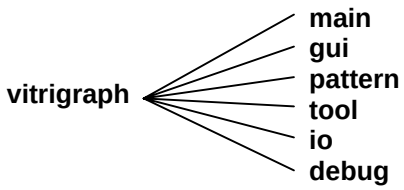


Figure 2.2 The vitrigraph package is split into six sub-packages

The **pattern** package contains the classes that store and manipulate the stained glass pattern. Routines to alter the pattern are only accessible from within this package. Classes such as **Editor** make calls to the **SGForger** class to change a pattern. The **SGPattern** class is used to pass the core data of the pattern between parts of the program, while preventing it from being altered. The **LeadData** and **GlassData** objects hold sets of **LeadType** and **GlassType** objects, which store the attributes of the different types of lead and glass known to the program. They are loaded from a file that can be updated to allow new types to be added. Information about glass and lead was obtained from the catalogues of James Hetley Stained Glass Supplies. The attributes readily available and relevant when designing a stained glass window have been included in the **LeadType** and **GlassType** classes, and are listed below.

Lead Attributes
- Name
- Face width
- Heart width
- Depth
- Price per metre

Glass Attributes
- Name
- Colour
- Thickness
- Price per square metre

The **tool** package contains the classes that manipulate and interpret the stained glass pattern. **Editor** receives events such as mouse clicks from the user interface provided by **MainWindow**. The other classes in the package – **Evaluator** and **Translator –** are passed an **SGPattern** object that allows them to compute additional data from the pattern without altering it. An **SGPattern** object is also passed to the **FileAccess** and **Print** classes, so that the data for a particular pattern can be saved, loaded and printed.

Vitrigraph is a unique program with a specific purpose. It does not have to load or save files in multiple file formats. After consulting the book 'Encyclopaedia of Graphics File Formats' [6], it was decided that standard raster image formats were unsuitable for saving patterns, and standard vector formats could not store the additional information about lead and glass. The AutoCAD DXF format was considered, but it is a complex format incorporating many elaborate features that would not be used in this application. Stained glass patterns are therefore saved to disk in a proprietary file format based on Java object serialisation. This provides a reliable way of storing complex data structures such as those used in Vitrigraph. The files include file format version numbers, and the implementation of the **FileAccess** class allows the file format to be augmented with extra data, while maintaining compatibility with previous versions. This will enable later versions of the program to add to the original file format while still being able to load older patterns, and allow older versions of the program to extract data from files of a newer format. This is achieved using a Java interface within the **io** package that defines the methods that a pattern file object must implement, while allowing other capabilities to be added in future versions.

Two extensions were cited in the project proposal: the use of glass textures and conversion of patterns to ray-tracer format. The first was considered throughout the design of the program so it could be included in the future. It was not used because the benefits were not deemed great enough to divert attention away from other parts of the project. The facility for creating ray-tracer files has been incorporated as a major part of the program, and is the method by which true aesthetic appraisal of patterns is possible.

2.5 Stained Glass Pattern Structure

A stained glass pattern is defined in two-dimensional 'pattern space' where a length of one hundred units corresponds to 35.25mm in real space. This scale is due to technicalities of the Java graphics implementation and is transparent to the user. The design of the components that constitute a stained glass pattern in Vitrigraph, and their interdependencies, are described on the following pages. The pattern is comprised of three basic constructs:

- **Vertices** Two-dimensional points representing joins between segments of lead
- **Edges** Lines between the vertices representing the segments of lead
- **Facets** Closed loops of edges, representing the glass pieces

These constructs form an interconnected structure where facets are defined in terms of edges, which are defined in terms of vertices. An edge has links to the facets of which it is a part, and a vertex has links to the edges it affects. The scheme was inspired by the 'Winged Edge Data Structure' of Baumgart [1].

The data and methods necessary to define a stained glass pattern are held in **SGVertex**, **SGEdge** and **SGFacet** objects. The data held within these objects is private. Changes to the data require calls to the methods of the associated objects, which ensures that a component of a pattern can maintain it's internal state. An **SGPattern** object stores vectors of these three object types, to form the complete definition of a particular pattern. An **SGForger** object adds functionality to this data, allowing the pattern to be manipulated.

An **SGVertex** is the representation of a point in pattern space. The user manipulates the pattern by creating, moving and deleting vertices.

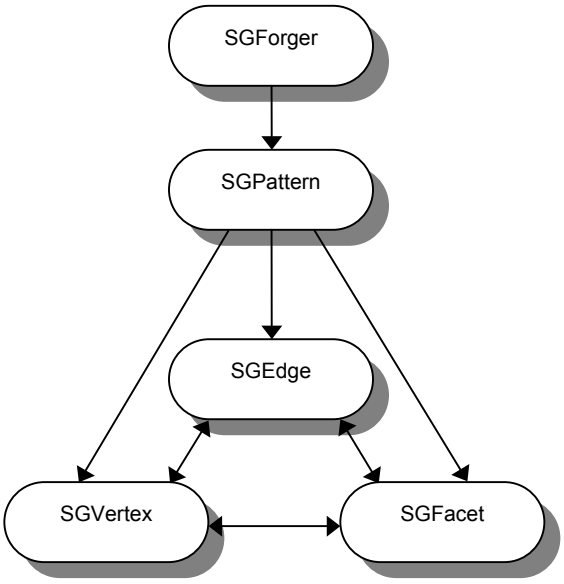


Figure 2.3 Stained glass pattern data structure

An **SGEdge** represents an edge in pattern space defined by the positions of a number of vertices. It was decided that the line shapes that could most accurately and concisely define the outline of a stained glass pattern, were: straight lines, circles, circular arcs and curves that interpolate a set of vertices. The curves require C^1 continuity (continuity of the first derivative) because of the technique used to scour and snap the glass. The type of curves chosen was Bézier curves, because they have the desired attributes, and avoid the additional and unnecessary complexity of the more general B-splines. A new system was devised to join Bézier curves together with C^1 continuity. This is used to make a series of edges form a curve that interpolates a set of vertices. The system is described below.

An **SGEdge** has one of the four types shown in the table below. Each edge has two primary vertices – the end points of the edge – and zero or more secondary vertices which affect the shape of the edge indirectly. The total number of vertices is the number of degrees of freedom of the shape. Figure 2.4 contains examples of the four types of edge.

Edge Type	Primary Vertices	Secondary Vertices	Graphical Representation
Straight	2	0	Straight line
Quadratic	2	1	Quadratic Bézier
Cubic	2	2	Cubic Bézier
Arc	2	1	Circular arc

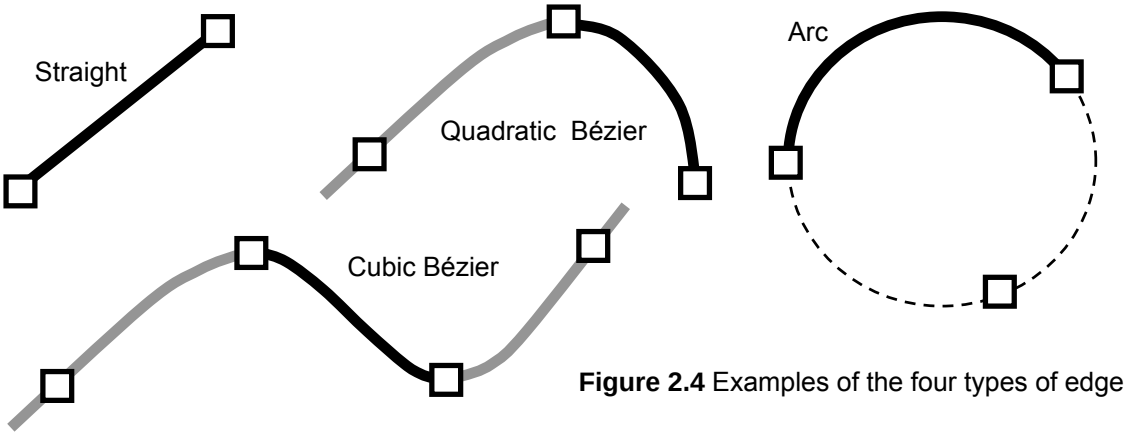


Figure 2.4 Examples of the four types of edge

The edge types were chosen for the following reasons:

- Each edge has two primary vertices: a start vertex and an end vertex. This is important when facets are considered.
- Each line has C^1 continuity throughout. This is a property of Bézier curves, as well as straight lines and arcs, and enables the line to be cut in glass.
- An edge can be sub-divided into two smaller edges using a simple rule. This allows a vertex to split an edge (page 19).
- A 2-dimensional bounding box for each edge can be calculated. The box is used for edge intersection detection (page 18).
- Routines to draw each line type are provided by the Java2D graphics API in Java 1.2
- A line specified with the method above is invariant under translations, rotations and uniform scalings of pattern space. These transformations are simply applied to the primary and secondary vertices of each edge to transform a whole pattern.

Each **SGEdge** object contains a field used as an index by the **LeadData** class to retrieve a **LeadType** object. The information from this object can be used to calculate physical, graphical and financial properties of the edge.

The original design used two separate edge types to represent circles and arcs, each edge interpolating three vertices. As work progressed, however, it became apparent that the single arc type shown above was preferable, and can be used to create circles and arcs that interpolate three vertices (figure 2.5).

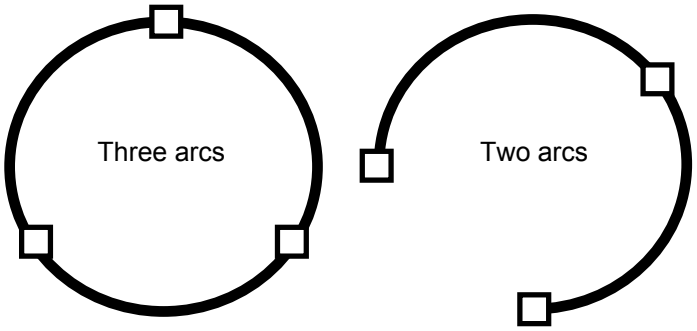


Figure 2.5 Edges are combined to form circles and arcs

It was decided that the user must be able to draw curved lines that interpolate a series of vertices, because this allows general shapes to be intuitively created and accurately positioned. Bézier curves are usually manipulated by positioning the end points which are interpolated and the control points which are not interpolated. A system was devised for deriving the positions of the end points and control points of a Bézier curve from the vertex positions of an edge. This allows the user to specify a series of vertex positions which are then interpolated by a series of Bézier curves.

A curved line interpolating n vertices where $n \geq 3$ is constructed using a quadratic edge at each end and $n - 3$ cubic edges inbetween. Each edge is a Bézier curve whose end points are the primary vertices of the edge, and whose control points are calculated from the positions of the primary and secondary vertices. A curved line is created from a series of edges where neighbouring edges share primary vertices so that

they join at the ends. The secondary vertices of an edge are the other primary vertices of it's neighbouring edges. The system described on page 16 ensures that the control points for neighbouring Bézier curves are equidistant from their common end point and the three points are colinear. Because it is a property of Bézier curves that their direction at their end points is equal to that of the line from the control point to the end point, C^1 continuity between the segments of the curved line is enforced. Continuity is maintained when edges are split in two (page 19) by careful assignment of the primary and secondary vertices of the sub-edges.

A curved line interpolating four points can be constructed as shown in figure 2.6 (a). The edges at the ends of the line are quadratic edges. Figure 2.6 (b) shows a quadratic edge with primary vertices v1 and v2, and secondary vertex v3. Figure 2.6 (c) shows a cubic edge with primary vertices v2 and v3, and secondary vertices v1 and v 4.

An **SGFacet** represents a glass facet in the stained glass pattern. It consists of a series of edges that form a closed loop, where no edge intersects any other edge in the pattern. This means that each edge is part of the perimeter of zero, one or two facets.

Each **SGFacet** object contains a field used as an index by the **GlassData** class to retrieve a **GlassType** object. The information from this object can be used to calculate graphical and financial properties of the facet.

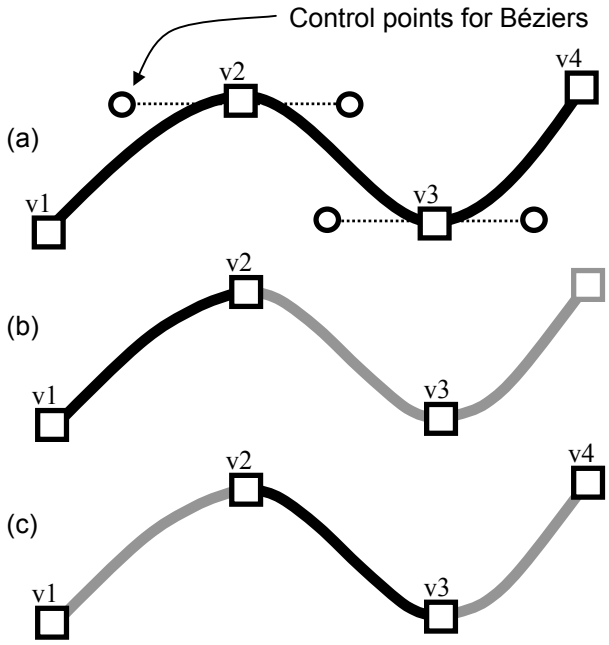


Figure 2.6 A curved line constructed from quadratic and cubic edges

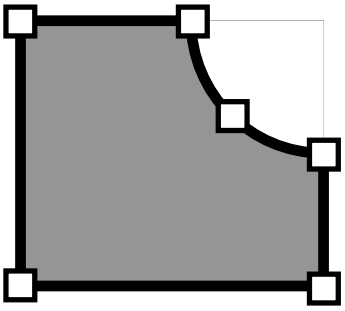


Figure 2.7 A facet

3 Implementation

This section on implementation starts by describing the features accessible via the graphical user interface of the Vitrigraph program. This prompts an explanation of the underlying implementation. The representation and manipulation of the vertices, edges and facets of a stained glass pattern are covered in detail, followed by a description of how information about lead and glass is linked to the pattern. Financial evaluation, saving and loading, and printing of patterns are then described. Finally, the extension to output patterns as scene description files for ray tracing is presented.

3.1 User Interface

The Vitrigraph user interface essentially indicates the high level functions performed by the program. All actions are initiated from the main window, which is represented as a **MainWindow** object from the **gui** package. Further dialogue windows subsequently appear displaying data or asking for information or confirmation.

One of six drawing tools is selected at any time. The current tool selection determines how use of the mouse pointer affects the stained glass pattern displayed in the window.

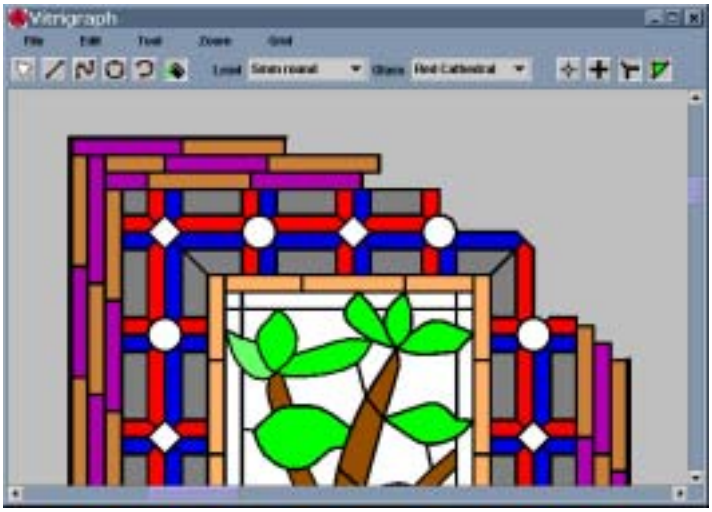


Figure 3.1 The main window of Vitrigraph

The Menu Bar

The options available from the menu bar of Vitrigraph (figure 3.2) are listed and briefly described in the table below.

Figure 3.2 The menu bar

Menu	Options	Explanation
File	New, Open, Save, SaveAs, Exit	Create a new pattern, load and save pattern files (page 26) and exit the program.
Edit	Cut, Copy, Paste, Delete	Editing features enabling parts of a pattern to be deleted or duplicated.
Tool	Translate, Evaluate, Pointer, Line, Curve, Circle, Arc	Translate the pattern to ray-tracer format (page 28), evaluate the pattern financially (page 24). Tools for building a pattern: see next page.
Zoom	zoom factors	Cause the pattern to be displayed at different magnification levels.
Grid	Off, Display Only, Snap to Grid, Choose Grid Size	Toggle use of the grid, and change the grid spacing in millimetres.

The Tool Bar

Figure 3.3 is an image of the toolbar from the main window of Vitrigraph. It is followed by a table that lists and briefly describes the options available.

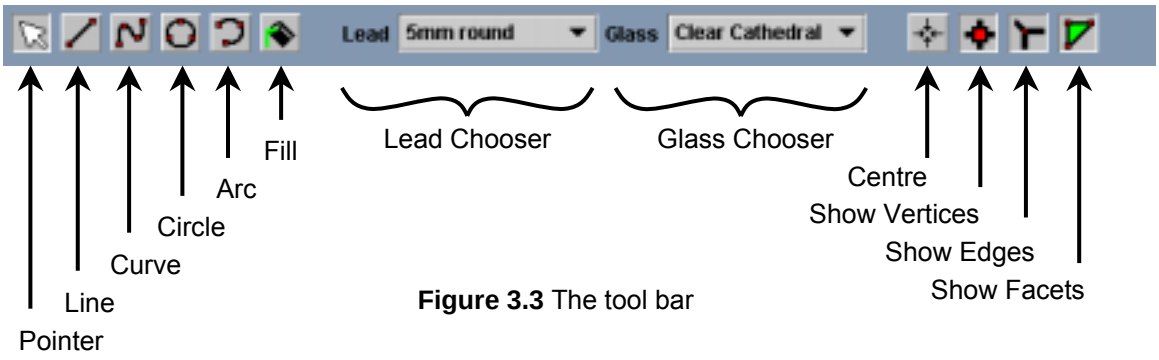


Figure 3.3 The tool bar

Group	Option	Explanation
Drawing Tools	Pointer, Line, Curve, Circle, Arc, Fill	Select vertices. Draw lines of various types. Fill Facets.
Choosers	Lead Chooser, Glass Chooser	Choose a type of lead or glass from a list (see page 23). This is then used by the drawing tools to create edges or fill facets.
Scrolling	Centre	Scroll the display to the centre of the stained glass pattern.
Display Options	ShowVertices, Show Edges, Show Facets	Toggle the display of the vertices, true thickness of the lead, and glass facets.

Control Flow

Figure 3.4 depicts part of the object oriented design on page 7. It depicts the flow of control that ultimately causes the modification of a stained glass pattern. The **Shell** creates a **MainWindow** object that accepts mouse and keyboard events from the user. These are mapped to methods of the **Editor,** such as those to select a particular drawing tool, or register a mouse click at a certain position in the pattern. The **Editor** may then invoke methods of an **SGForger** object of which it may have many. The **SGForger** provides methods such as those to select the vertices in a particular area of the pattern, or fill the facet at a particular position in pattern space. It is essentially a wrapper that adds functionality to the **SGPattern** object, which is a data structure that stores the pattern.

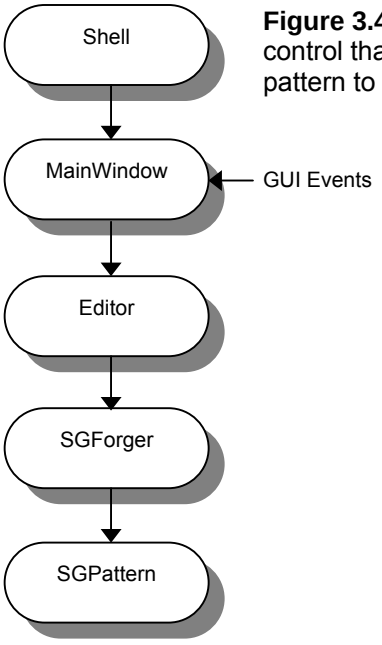


Figure 3.4 The flow of control that allows a pattern to be edited

3.2 Vertex Representation

3.2.1 The SGVertex Class

The **SGVertex** class encapsulates the data and methods associated with a vertex. These are listed in apendix B.

3.2.2 Vertex Manipulation

A vertex is created when the user clicks the mouse in the editing area of the Vitrigraph window. The mouse position is transformed to a point in pattern space which is passed to the **SGVertex** constructor. When a vertex is dragged with the mouse, the relative difference in position is passed to the **translate** method of the vertex.

When the user creates a 'rubber band' by dragging out an area with the mouse (figure 3.5), the **SGForger** object that is handling the current pattern temporarily moves the selected vertices to a separate vector, to allow them to be manipulated as a unit. The **in** method of each vertex is used to determine whether it is inside or outide the rubber band which is transformed into pattern space.

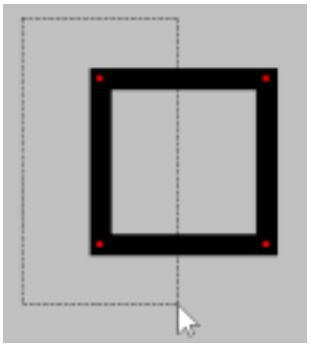


Figure 3.5 'Rubberbanding' to select vertices

The **squareDistanceFrom** method is used to cause vertices to 'snap' together. When the user finishes moving a set of vertices, or creating a new edge, the method is used to calculate the distance from a new or moved vertex, to the other vertices in the pattern. If this is less that a limit defined in the **VGGlobal** class, the vertices are merged, with one vertex replacing the other. In the example on the right, vertex A replaces vertex B. This is achieved by calling B·**replaceWith(**A**),** which calls the **replaceVertex** method of each of B's primary and secondary edges, to remove B from the pattern data structure. A·**killRedundantEdges** is then called, causing any duplicate primary and secondary edges that A now has to be killed. In figure 3.6, one of the edges from A must be killed. B·**kill** is then called to mark B as redundant. It will be removed completely from the pattern data structure by the **SGForger** object. The test to determine if two edges are equal is performed by the **equals** method of the **SGEdge** class, which uses a different algorithm for each edge type.

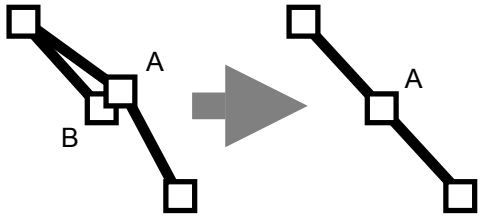


Figure 3.6 Vertices 'snap' together

When the **validate** method of a vertex is called it's primary edges are checked. If they are all redundant the vertex is killed, because it is floating in space, not attached to anything (figure 3.7). When a vertex is killed it calls the **validate** method of all of it's secondary edges.

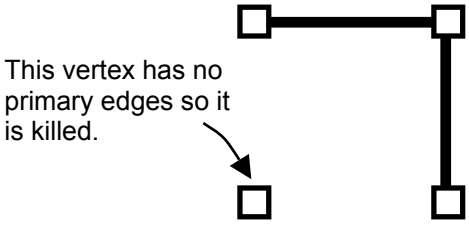


Figure 3.7 Redundant vertices are deleted

When a vertex is moved, it calls the **invalidateImage** method of it's primary and secondary edges to force them to recalculate their graphical representations, which will have been affected.

3.3 Edge Representation

3.3.1 The SGEdge Class

The **SGEdge** class encapsulates the data and methods associated with an edge. These are are listed in appendix B. An edge has one of four types: straight, quadratic, cubic or arc. A straight edge is simply drawn as a straight line between it's two primary vertices, of the appropriate width and colour, which is scaled and translated to account for the zoom factor and scroll position selected by the user. The methods devised for drawing edges of the other types are described below.

3.3.2 Drawing a Curved Edge

The shape of a quadratic edge is a quadratic Bézier curve defined by three points, two of which are the end points (primary vertices) of the edge. The other control point is calculated from the positions of the two end points and that of the single secondary vertex. A cubic edge is a cubic Bézier curve defined by four control points, two of which are the end points of the edge. The other two control points are calculated in the same way as for the quadratic edge, by applying the algorithm to each end of the edge in turn.

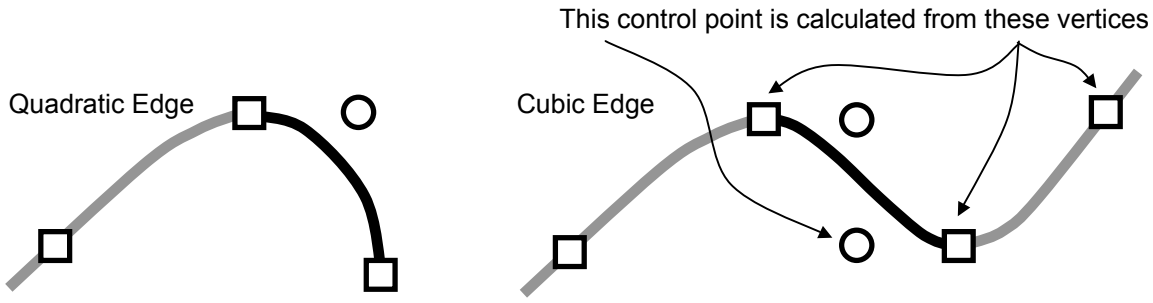


Figure 3.8 Derived Bézier control points for quadratic and cubic edges.

If P and Q are the end points of a curve and R is a secondary vertex, the position of the control point for the Bézier curve is displaced from Q by vector **d** (figure 3.9). The end point of **d** is located using the length *d* and the sine and cosine of angle α, which are calculated (using a temporary variable θ) as follows.

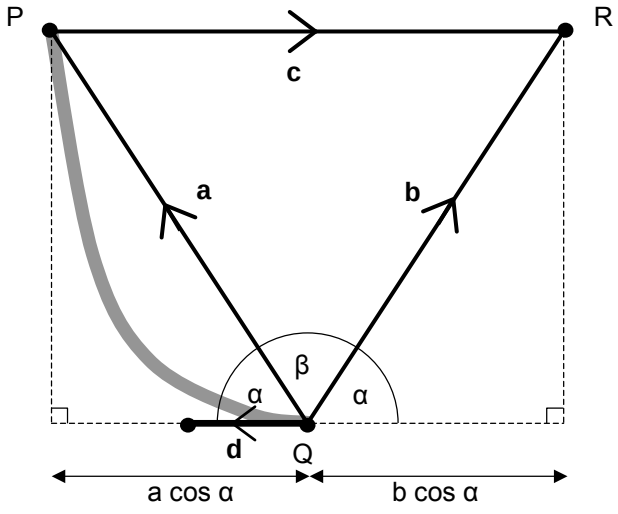


Figure 3.9 Calculations involve the two primary vertices and one secondary vertex

cosine rule:

$$c^2 = a^2 + b^2 - 2ab\cos\beta$$

$$\theta = 2\cos\beta = \frac{a^2 + b^2 - c^2}{ab}$$

trigonometry:

$$\cos\alpha = \cos\left(90° - \frac{\beta}{2}\right) = \sin\left(\frac{\beta}{2}\right)$$

$$\cos^2\alpha = \sin^2\left(\frac{\beta}{2}\right) = \frac{1-\cos\beta}{2}$$

$$\sin\alpha = \sqrt{1-\cos^2\alpha} = \sqrt{\frac{2+\theta}{4}}$$

vector product:

$$\begin{bmatrix} b_x \\ b_y \\ 0 \end{bmatrix} \times \begin{bmatrix} a_x \\ a_y \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ b_x a_y - b_y a_x \end{bmatrix}$$

If $b_x a_y - b_y a_x \geq 0$ then **a** is anti-clockwise about Q from **b,** so **d** is anticlockwise from **a**. Otherwise **d** is clockwise from **a**. The length of **d** is given by $d = min(a \cos \alpha, b \cos \alpha) / 3$

The symmtery of the method above ensures that a curve from Q to R with P as a secondary vertex, will place its control point a distance d from Q, in the opposite direction to **d**. At the end points the direction of a Bézier curve is given by the line between the two control points on that side of the curve. This means that the curves that meet at Q will have C^1 continuity as desired.

3.3.3 Drawing an Arc Edge

The arc edge is defined by two primary vertices and one secondary vertex, referred to here as P, Q and R respectively (see figure 3.10). The centre of the circle is point O, and the radius is r.

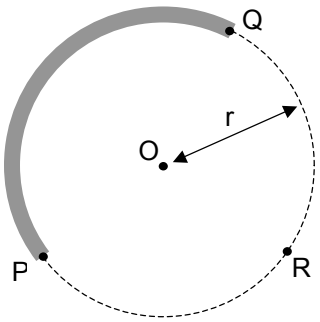


Figure 3.10
An arc edge with primary vertices P and Q, and secondary vertex R

By noting:

$$|P - O| = |Q - O| = |R - O|$$

Rearranging gives:

$$O_y = \frac{-\left((R_x - P_x)\left(P_x^2 + P_y^2 - Q_x^2 - Q_y^2\right) + (P_x - Q_x)\left(P_x^2 + P_y^2 - R_x^2 - R_y^2\right)\right)}{2\left((R_x - P_x)(Q_y - P_y) + (P_x - Q_x)(R_y - P_y)\right)}$$

$$O_x = \frac{2(Q_y - P_y)O_y + P_x^2 + P_y^2 - Q_x^2 - Q_y^2}{2(P_x - Q_x)}$$

Reuse of common sub-terms allows the location of the centre of the circle to be calculated using the above formulas in a total of 27 basic arithmetic operations. The radius is then calculated as:

$$r = |O - P|$$

A value i is calculated:

$$i = (P_x - Q_x)(R_y - Q_y) - (P_y - Q_y)(R_x - Q_x)$$

The sign of i determines the side of the line from P to Q on which point R falls. This establishes whether the arc is drawn clockwise or anti-clockwise from P.

Some special cases need to be handled. When P, Q and R are co-linear, or nearly so, the circle has a very large radius and the denominator in the formula for O_y approaches zero. In this case, if R lies outside the region between P and Q shown in figure 3.11, the arc is approximated by a straight line. If R lies inside that region, the arc is not drawn.

When $P_x = Q_x$ alternative formulae to those above must be used. O_y is calculated as $\frac{1}{2}(P_y + Q_y)$ then O_x is calculated through a rearrangement of the formula $|P - O| = |R - O|$.

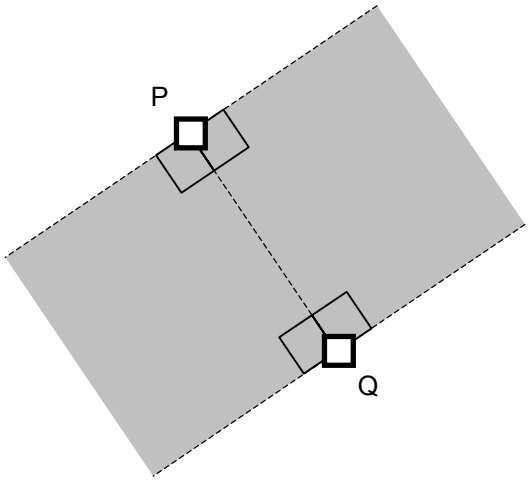


Figure 3.11 An arc with a very large radius is approximated by a straight line between P and Q if R lies outside the shaded region

3.3.4 Edge Intersection Detection

An edge that intersects another edge cannot form part of a facet. The facet detection algorithm excludes edges whose **intersectTag** field is set. The algorithm to detect intersections and set the appropriate **intersectTag** fields is described below.

A pairwise check of all edges was originally used to test for intersection, but during testing with large patterns it became clear that this was too slow. A new algorithm was produced that creates an array containing references to all of the edges. The result of the **getBoundingBox** method of each edge is used to obtain two values, z_1 and z_2 (see figure 3.12). These are the sums of the x and y components of the top-left and bottom-right corners of the bounding box (coordinates are measured from the top-left). The edges are sorted by their z_1 values, then the algorithm proceeds as follows.

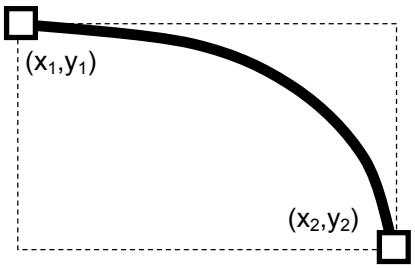


Figure 3.12
Two values are calculated from the bounding box of an edge, $z_1 = x_1 + y_1$ and $z_2 = x_2 + y_2$

for each edge $\boldsymbol{e_1}$ *in the array in order*
  for every subsequent edge $\boldsymbol{e_2}$ *whose* z_1 *value is less than or equal to the* z_2 *value of* $\boldsymbol{e_1}$
    if the intersect tag of either $\boldsymbol{e_1}$ *or* $\boldsymbol{e_2}$ *is not set*
      if $\boldsymbol{e_1}$ *intersects* $\boldsymbol{e_2}$
        set the intersect tags of $\boldsymbol{e_1}$ *and* $\boldsymbol{e_2}$

Conceptually the algorithm advances a diagonal line from the top left as in figure 3.13 (a), which stops when it encounters the top-left corner of a bounding box. It then tests for intersection between the edge whose bounding box it has just found, and any un-tested edges whose bounding boxes intersect the diagonal region defined by the first edge, which is shaded in figure 3.13 (b).

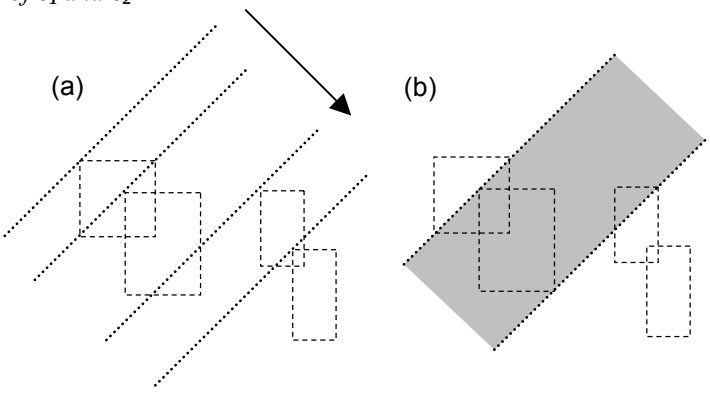


Figure 3.13 Edges are processed in the order in which their bounding boxes are encountered when scanning from the top-left

The original algorithm always used $\frac{1}{2}n^2$ pairwise edge comparisons when testing n edges. Assuming that the bounding boxes are distributed in a regular grid the new algorithm will check each of the n edges against an average number of other edges of the order of $\sqrt{n}$. This gives an average time complexity of $n^{1.5}$. The performance of the new algorithm seen in practice is very much better

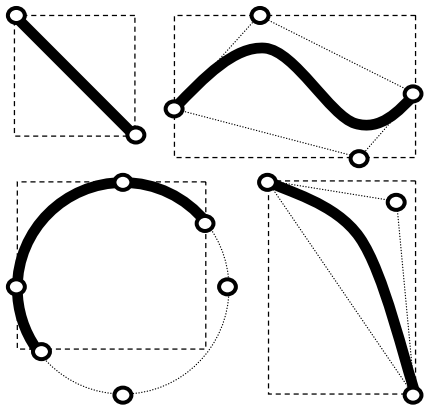


Figure 3.14 Bounding boxes for the different edge types

Bounding boxes must be calculated for edges of each of the four types (figure 3.14). The bounding box of a straight line is simply defined by the end points of the line. The bounding box of a Bézier curve is defined by the minimum and maximum x and y values of the three or four control points: it is a property of Bézier curves that they are contained within the convex hull of those points. When computing the bounding box of an arc, up to six points are considered: the two end points, plus any of the four points at the top, bottom, left and right of the circle that are within the angular extents of the arc.

When a pair of edges must be tested for intersection, the **intersectsEdge** method of one is passed a reference to the other. This method obtains a representation of the graphical

shape of each edge by calling it's **currentEdgeShape** method. Recursion and subdivision are used to determine whether the two shapes intersect.

Subdivision of a straight line shape is simple. Subdivision of a quadratic or cubic Bézier is achieved by applying the standard formulae. A quadratic Bézier with control points P_0, P_1, P_2 is divided into two quadratic Béziers with control points P_0, $\frac{1}{2}(P_0+P_1)$, $\frac{1}{4}(P_0+2P_1+P_2)$ and $\frac{1}{4}(P_0+2P_1+P_2)$, $\frac{1}{2}(P_1+P_2)$, P_2. A cubic Bézier is subdivided in a similar way. An arc is subdivided by sharing it's angular extent between two arcs with the same centre and radius. The algorithm for deciding whether two shapes s_1 and s_2 intersect, is as follows.

```
1  Intersect ( s1 , s2 )
2      if the bounding boxes of s1 and s2 do not intersect, return false
3      if either bounding box is thinner than the lead width of the edge
4          if the distance from the centre of the intersection of the bounding boxes to the end of the edge is
5              less than the lead width, return false
6          else return true
7      subdivide s1 into shapes s1a and s1b
8      if Intersect (s2, s1a) return true
9      if Intersect (s2, s1b) return true
10     return false
```

The parameters are swapped for each recursive call in lines 8 and 9, so the two shapes get alternately subdivided. If the shapes do intersect, the bounding boxes of the shape segments will converge on the intersection point (figure 3.15) until they are small enough to return the value true. If two edges share a common primary vertex, the point at which they meet is excluded from being classed as an intersection by the condition on lines 4 and 5.

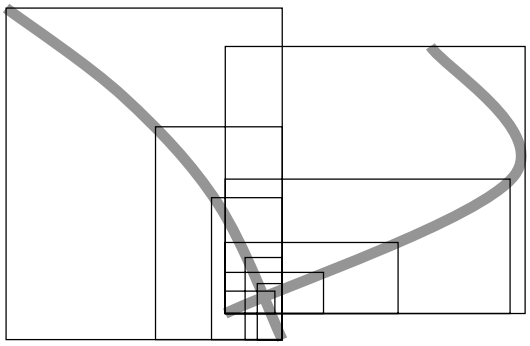


Figure 3.15 The bounding boxes of the shape segments converge on the intersection point

The **intersects** method of an edge determines whether the edge intersects a specified rectangle in pattern space, in much the same way as the edge intersection algorithm above. When the user stops moving a set of vertices or creating a new shape, the affected vertices are incorporated into the rest of the design. If a vertex intersects an edge, it is shifted to the nearest position on the edge by identifying that point parametrically along the edge. The edge is then split into two, and the vertex becomes the common point. Intersection of a vertex and an edge is detected by transforming the graphical representation of the vertex on the screen, to a rectangle in pattern space, which is passed to the **intersects** method of the edges in the pattern.

3.3.5 Splitting an Edge

Due to the system chosen for specifying edges, an edge of any type can be easily subdivided once a vertex has been positioned at a point along it's length. A new edge is made with the same attributes as the original one, and then the primary and secondary vertices of the two edges are reassigned. The assignments that are used are important because, for instance, a curved line comprised of cubic edges must retain it's C^1 continuity when one of the edges is subdivided by a vertex. Source code from the **SGEdge** class for splitting a cubic edge is included in appendix C.

Figure 3.16 on the next page shows how an edge of each type is subdivided by a new vertex N, into two edges that use combinations of the original primary and secondary vertices. The vectors of primary and secondary edges maintained by the vertices are updated accordingly.

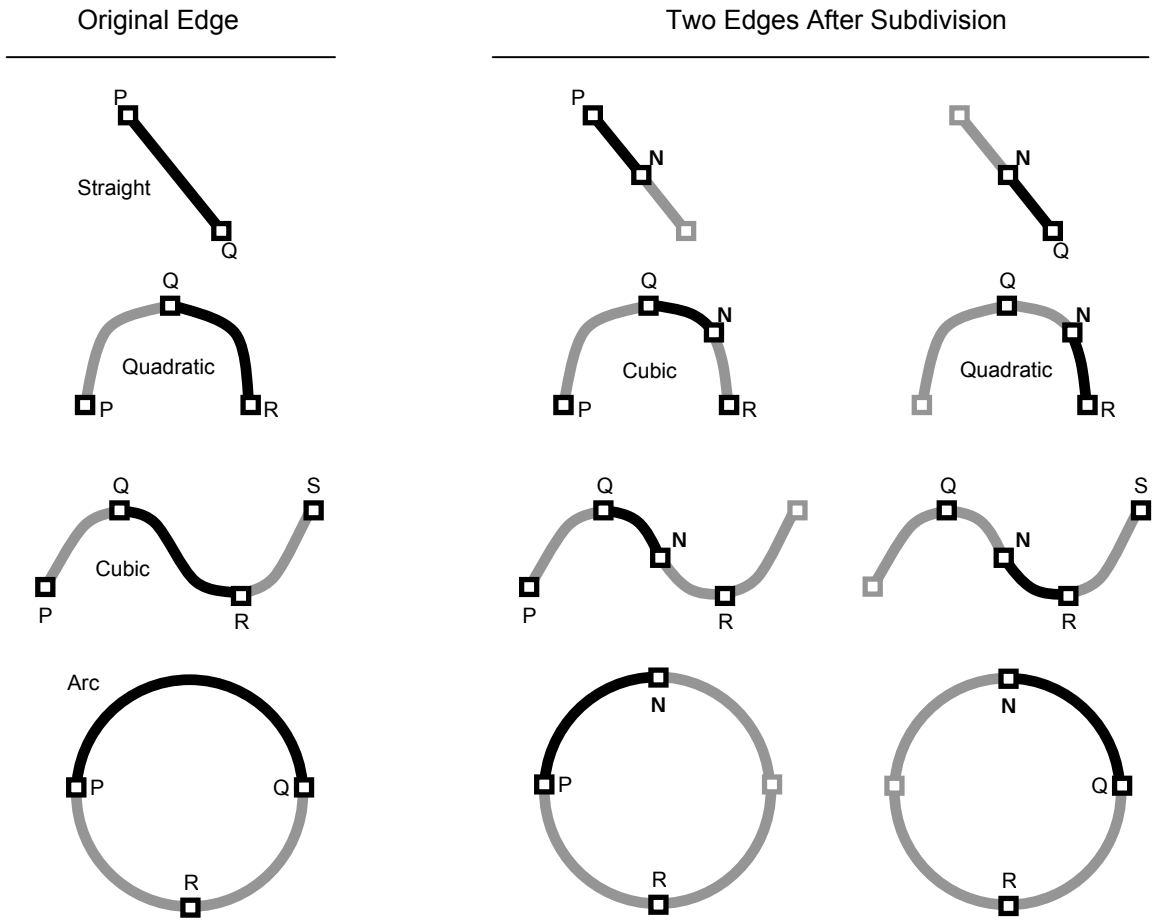


Figure 3.16 Subdivision of an edge by a new vertex N, and the consequent primary and secondary vertex reallocation

3.3.6 Edge Simplification

The **kill** method of a vertex is invoked when the user deletes it, or the when the vertex's **validate** method determines it should be killed. It is then tagged as redundant. The **validate** method of an edge tests whether it should be redundant, and kills it if necessary. An edge, however, may require alteration without being redundant, because one or more of it's secondary vertices is redundant or is equal to one of it's primary vertices.

The algorithm for validating an edge is shown on the right. The primary vertices of the edge are v_0 and v_1. In an arc edge the secondary vertex is v_2. In a quadratic edge the secondary vertex v_2 is associated with the v_0 end of the edge. This is why v_0 and v_1 need to be swapped on line 14 of the algorithm. In a cubic edge the secondary vertices are v_2 and v_3 which are associated with the v_0 and v_1 ends respectively.

```
1   if (redundant(v0) or redundant(v1))
2       then kill
3   if (v0=v1)
4       then kill
5   if (edgeType=quadratic)
6       then if (v2=v0 or v2=v1 or redundant(v2))
7           then edgeType ← straight
8   if (edgeType=cubic)
9       then if (v2=v0 or v2=v1 or redundant(v2))
10          then if (v3=v0 or v3=v1 or redundant(v3))
11              then edgeType ← straight
12              else edgeType ← quadratic
13                   v2 ← v3
14                   swap v0 and v1
15          else if (v3=v0 or v3=v1 or redundant(v3))
16              then edgeType ← quadratic
17  if (edgeType=arc)
18      then if (v2=v0 or v2=v1 or redundant(v2))
19          then edgeType ← straight
```

Vertices are equated using pointer equality since vertices at the same position will have 'snapped' together. The predicate *redundant(v_i)* means that vertex v_i has been tagged as redundant. The method *kill* causes the edge itself to be tagged as redundant. The necessary modifications to the primary and secondary edge lists of the vertices are not shown here. Edge simplifications can be summarised as follows: quadratic and arc edges are simplified to straight lines, while cubic edges are simplified to quadratic edges if one secondary vertex remains, or straight edges if none remains.

3.3.7 Tangent Vectors

The facet algorithms require the direction of the tangent vector at each end of an edge to be calculated. Due to the choice of edge types this is easily achieved. The method for straight edges is simple. For quadratic and cubic edges the difference between the two control points near the end of the edge is used as a tangent vector. As noted previously, a Bézier curve at the end points has the same direction as the line between the two control points associated with that end. For arc edges the vector can be computed using the locations of the end points and the centre of the circle, as shown in figure 3.17.

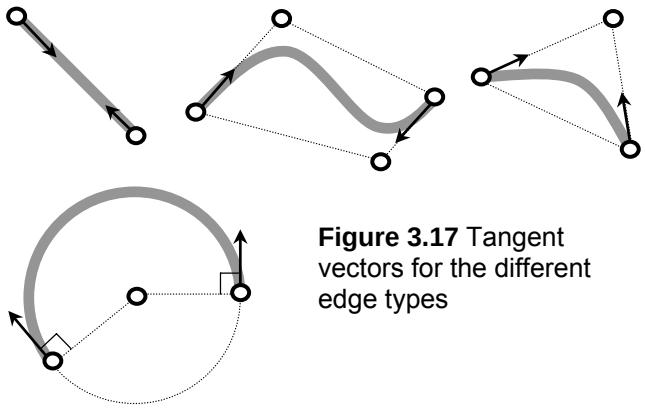


Figure 3.17 Tangent vectors for the different edge types

3.4 Facet Representation

3.4.1 The SGFacet Class

The **SGFacet** class encapsulates the data and methods associated with a facet. These are listed in appendix B.

3.4.2 Facet Detection

The algorithm for detecting facets was inspired, in part, by the Depth-first search and Graham Scan algorithms [2]. Each edge in the pattern has two sides, and can be thought of as two directed edges. Each of these edges can be part of the perimeter of a facet, where the facet is on the left side of the directed edge. Each directed edge has a facet tag which is set to true when it becomes part of a facet. The algorithm processes each of the directed edges in turn, excluding those whose facet tag has already been set. For each one it follows the edge whose tangent angle at its start vertex is the closest clockwise to the tangent angle at the end vertex of the current edge. If a loop of edges is created, a facet has been found. If the search cannot go any further, or finds an edge that is already part of a facet, it fails. Failure also results from finding an edge with it's intersect tag set.

Figure 3.18 shows how a simple facet could be detected. The directed edges numbered 1, 2, 3 and 4 form the perimeter of the facet, whose interior is on the left of each edge. The basic form of the algorithm is shown below.

```
set the facet tags of all edges to false
for each edge e in the pattern
    for each side of e
        if the facet tag of this side of e is false
            create a new facet f
            vertex s ← the start vertex of e
            edge d ← e
            do
                add d to the perimeter of f
                d ← the sharpest edge from the end of d
                if d=null or the facet tag or intersect tag of d is set
                    then finish checking this side of e
            while the end vertex of d is not equal to s
            add d to the perimeter of f
            set the facet tags of all the edges of f
            add f to the pattern
```

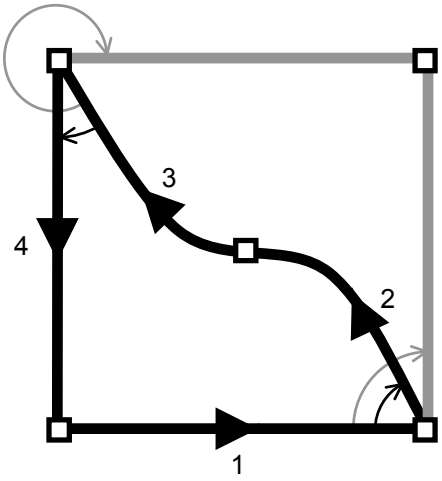


Figure 3.18 Edge 2 is chosen to follow edge 1 because it makes the sharper clockwise angle

3.4.3 Concave Angle Detection

The **markSharpAngles** method of a facet finds 'inward corners' which would be hard to cut in glass. By considering the angles of the tangent vectors of all adjacent pairs of edges on the perimeter, it finds internal angles greater than a threshold value. The vertex at an offending corner is tagged using it's **setSharpFacetAngle** method. A tagged vertex, like the one in the middle of figure 3.19, appears with a ring around it as a visual warning to the user.

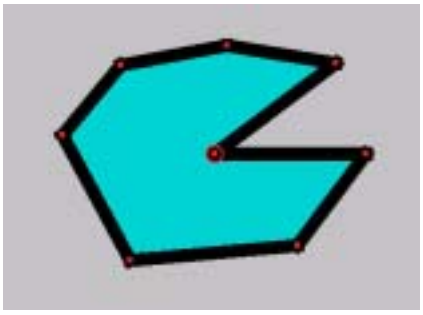


Figure 3.19 A facet containing an inward corner

3.5 Glass and Lead Data

Types of glass and lead are described by **GlassType** and **LeadType** objects which store the information listed on page 8. An **SGEdge** object represents a piece of lead, whose type is identified by it's **leadCode** string. The data for different lead types is held in a file that is loaded when the Vitrigraph program is started. The data is held in a hash table by the **LeadData** class, and is accessed when the attributes of a lead type are needed, such as when an edge is drawn on the screen, printed, financially evaluated, or translated to a part of a ray-tracer file. The **LeadData** class also produces a list of available types of lead ordered by name, to offer to the user on the tool bar.

Glass data is used in an analogous way by the **GlassData** class. Information about 14 types of glass and 12 types of lead was taken from the catalogues of James Hetley Stained Glass Supplies. Extending the system to use more types would simply be a matter of entering more information from the catalogues.

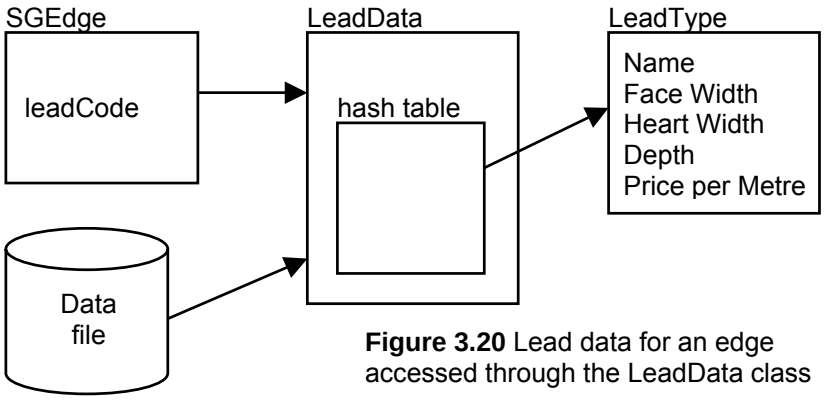


Figure 3.20 Lead data for an edge accessed through the LeadData class

3.6 Pattern Evaluation

Financial evalution of stained glass patterns is used to calculate the cost of construction. It is intended that extras like profit and VAT be added to this cost, as these are essentially arbitrary and may vary between customers. Following an investigation of how stained glass pieces are priced, and what the important points are, the evaluation system used by Vitrigraph was designed loosely, then refined by iterative development. A set of test patterns of varying size, shape, style, complexity, and raw material type were assigned proper costs, and used to verify the evaluation system at different stages.

Constructing a stained glass window is a very labour-intensive task. Most of the cost – typically over 80% – is due to the complexity of the pattern, rather than the raw materials. Consequently the complexity stage of pattern evaluation has received the most scrutiny and refinement.

Calculation of raw material costs involves determining how much of each type of lead and glass is needed. All edges in the pattern have their length in pattern space calculated, which requires adaptive subdivision for quadratic and cubic edges (figure 3.21). This is then converted to a length in metres and an off-cut length is added to account for wastage. The lengths of each lead type required are totalled and combined with the data from the **LeadData** class to give costs for the different lead types. A total cost for all of the lead can then be obtained. The cost of the glass is calculated in much the same way, except that a facet is assessed using it's rectangular bounding box, with an off-cut length added to each dimension to account for wastage (figure 3.22). This is a valid approximation because a facet will generally be cut from a rectangular piece, which itself is cut from a larger rectangular sheet using straight cuts along a ruler placed parallel to one of the sides.

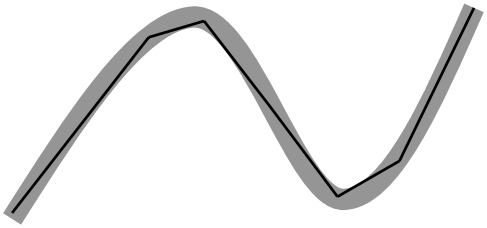


Figure 3.21 The length of a curved edge is approximated using adaptive subdivision

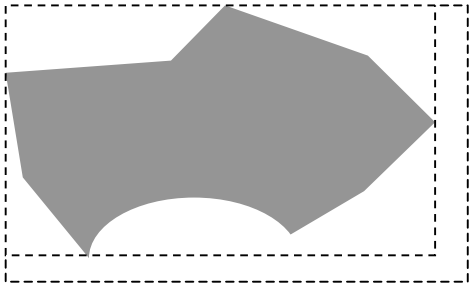


Figure 3.22 The bounding box of a facet is increased to account for wastage

The algorithm to calculate a cost due to the complexity of a pattern was originally designed to traverse the data structure, building up the cost as it went. Adapting this system and assessing the results was a slow process, so a new approach was devised. The new system, following the use of appropriate program constants, now traverses the data structure of a pattern and produces an alternative representation for evaluation purposes. This can be saved as a file of comma-delimited data. All of the test patterns were processed in this way, then loaded into the Microsoft Excel spreadsheet package. An algorithm for calculating the cost of a pattern from this data was written in Microsoft Visual Basic, an Excel worksheet incorporating graphs was made to compare the results from the different patterns, and an Excel macro was created to update the graphs when the algorithm was changed. This spreadsheet system was used to experiment with many evaluation algorithm variants. An effective evaluation algorithm was produced through this iterative development system, which was then rewritten in Java and incorporated into Vitrigraph. The test patterns and some data used to assess the evaluation algorithm are shown in appendix F.

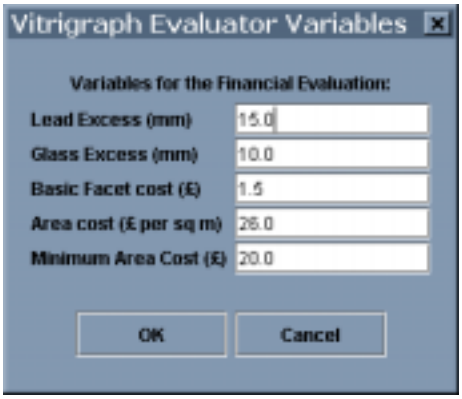


Figure 3.23 Pattern evaluation options

Figure 3.23 shows the options that are available to the user when evaluating a pattern, with the default settings which were obtained through testing in Excel. The graphs and explanations on the following page describe the basic principles behind the final pattern evaluation algorithm.

Figure 3.24 (a) shows how the cost for the area of the pattern is related to the area, using the default settings. It is effectively linear for large areas but is restricted to a minimum value. If **c** is the **'Area Cost'** setting from the options shown in figure 3.23, **m** is the **'Minimum Area Cost'** and **a** is the area of the pattern, the formula for the graph is $\mathbf{ca} + \mathbf{m}e^{-0.5\mathbf{a}}$. Figure 3.24 (b) shows how the cost of a facet is related to the edge count (a weighted combination of the numbers of edges of different types), using the default settings. If **f** is the **'Basic Facet Cost'** (used to scale the graph) and **n** is the variable on the **'Edge Count'** axis, the formula is $\mathbf{f}\times(1+\ln((\mathbf{n}+3)/3))$. The sum of the costs of the facets in a pattern is multiplied by each of the functions shown in figures 3.24 (c) and (d), to effectively give a bulk discount to large patterns, and reduce the price of very simple patterns. The **'Number of Edges'** and **'Number of Facets'** values are the total numbers of edges and facets in the pattern data structure respectively.

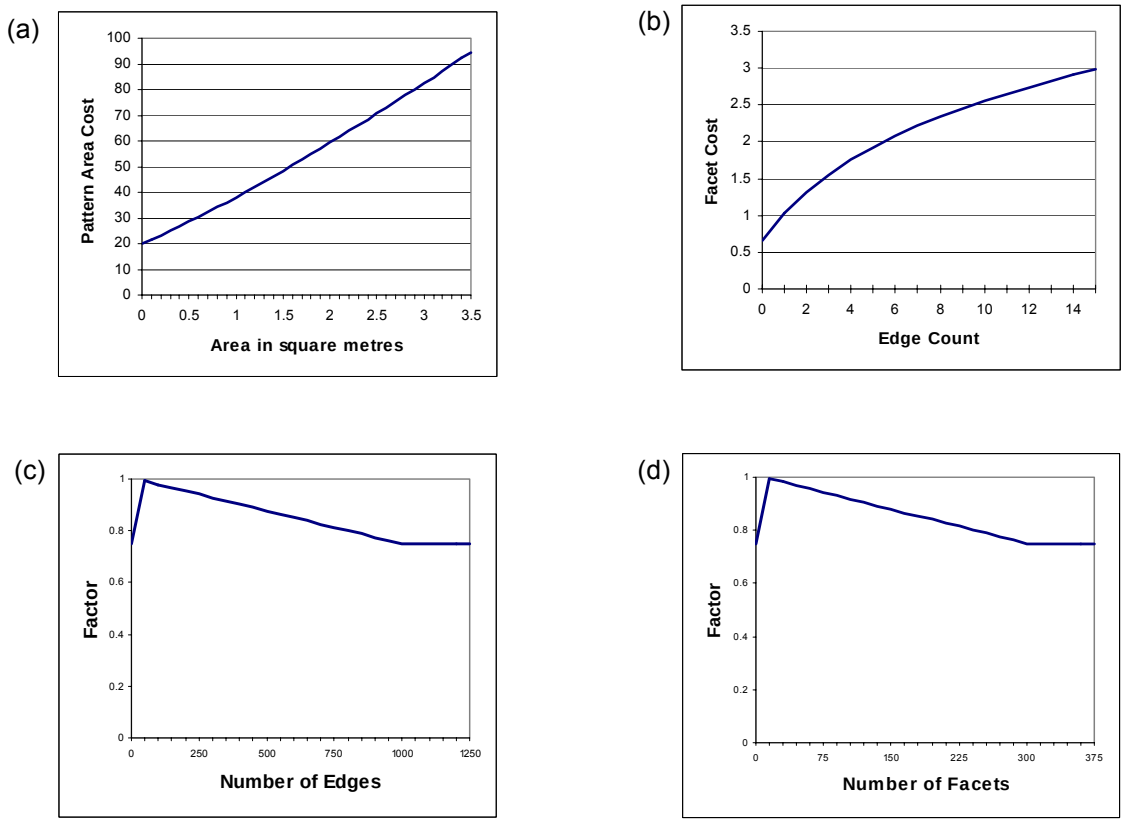


Figure 3.24 The main functions used during financial evaluation of a pattern

3.7 Files

Vitrigraph allows stained glass patterns to be saved to files and loaded from files. Java object serialisation was chosen to implement this as described on page 8. All of the objects that form the pattern data structure implement the Java interface **Serializable**, and have the methods **writeObject** and **readObject**. These cause fundamental data, such as the position of a vertex, to be saved. Data that is not vital, such as the **edgeShape** field of an edge, is not saved, but recreated when a pattern is loaded. To save a pattern, the **Editor** obtains a file name from the user, then passes the corresponding **SGPattern** object to the **FileAccess** class. Loading is achieved in an analogous way. Because the minimum amount of data is saved in a way controlled by the individual objects of the pattern structure, it will be possible to add extra functionality to those objects at a later date while maintaining compatibility with older files.

To save a pattern, the **FileAccess** class saves a single object of a type called **SGFileCurrentVersion**. This object has a constructor that takes an **SGPattern** object. It also has a **getPattern** method that returns an **SGPattern** object compatible with the current version of Vitrigraph. Using this mechanism, different versions of the program can create files which can be used by each other.

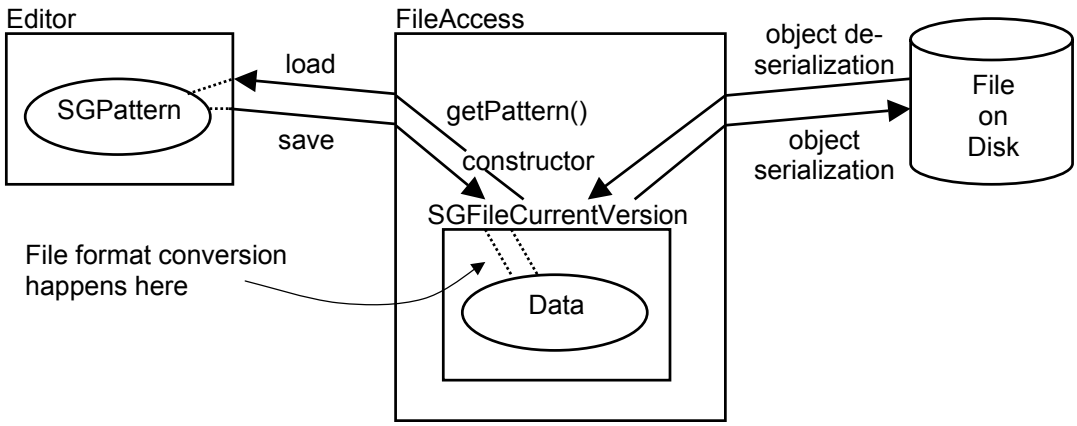


Figure 3.25 Loading and saving a pattern

3.8 Printing

A stained glass pattern created using Vitrigraph must be printed before being constructed from lead and glass. Ideally a life-size version that can be fixed to a bench would be printed, on top of which the stained glass window would be assembled. Stained glass pieces, however, generally have areas of one or two square metres, which would require the type of large format printer not commonly supplied with a standard computer system. Vitrigraph therefore supports a number of standard paper sizes from A4 and A3, up to 36" by 96".

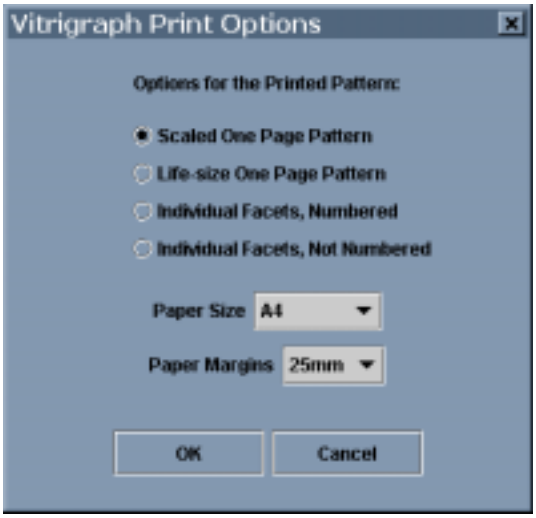


Figure 3.26 Calculations involve the two primary vertices and one secondary vertex

The options provided to the user when printing are shown in figure 3.26. **'Scaled One Page Pattern'** prints a scaled version of the pattern that fits on one sheet of paper, for assessing the general form of the pattern. **'Life-size One Page Pattern'** prints the pattern at exactly the correct scale, so that the stained glass window can be assembled on top. **'Individual Facets, Numbered'** prints a scaled one-page pattern, followed by outlines of all of the facets from which the glass can be cut. The type of glass to be used accompanies each facet. The facets in the scaled pattern and those on the individual sheets, are labelled with matching numbers to allow the pattern to be pieced together. **'Individual Facets, Not Numbered'** simply omits the facet numbering scheme. For example printouts see appendix E.

Figure 3.27 Hardcopy representation of a facet

When life-size facets or whole patterns are printed, the shapes of the edges, the lead face width and the lead heart width are used to produce a representation from which the glass can be cut accurately. An example facet is shown in figure 3.27. The thinner black line portrays the heart of the lead. The glass must be cut to fit inside this line. The thicker grey line portrays the face of the lead. Defects in the edges of a glass piece within this region are tolerable.

3.9 Ray-tracer Translation

A program called POV-Ray (Persistence Of Vision) was chosen to create realistic images from Vitrigraph patterns, because it is free, well known, and readily available for virtually all types of computer. The features available in POV-Ray were reviewed by reading the extensive program documentation, trying examples, and gaining information from the many related web sites such as *http://www.povray.org*. The pattern is translated into POV-Ray format which is then processed by the ray-tracer to create an image that allows the aesthetic appeal of the stained glass piece to be assessed.

The only parts of the lead in a pattern that are seen in any detail are the top and bottom surfaces. A piece of lead is approximated by two cylinders (figure 3.28), with the face width and depth taken from the lead type data. Arc edges are approximated by multiple cylinders with constant angular spacing. Quadratic and cubic edges are converted to many cylinders through adaptive subdivision, using a threshold which is a function of the lead width.

A vertex is translated as two spheres with diameter and centre-separation equal to the maximum face width and depth of it's primary edges. This ensures that the lead pieces form smooth joints (figure 3.29).

Facets are represented in the POV-Ray file as prisms (figure 3.30). The prism vertices are obtained from the shape of the edges on the perimeter using subdivision as for the lead segments. Lower accuracy is used because the width of the lead masks small errors in the shape of the facet.

The POV-Ray objects described above are standard ray-tracing constructs and consequently can be rendered quite quickly. Originally the lead was represented using the POV-Ray 'blob' construct which defines an implicit surface (an iso-surface of a scalar field). This can provide a more realistic profile of the lead and smooth joins at all of the vertices, but greatly increases the processing time for a complex pattern, and does not strongly enhance the overall appearance of the rendered image.

When choosing how to render a stained glass pattern with the ray-tracer, there are a plethora of possibities. A set of options has been chosen to allow functional and aesthetic representations to be generated. The window presented to the user is shown in figure 3.31 and the options are explained in the table on the right.

Figure 3.31 Ray-tracer translation options

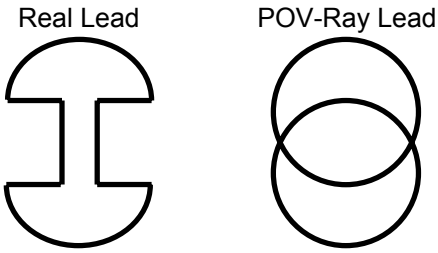


Figure 3.28 Shapes of the lead cross-sections

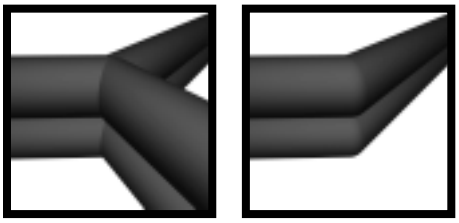


Figure 3.29 Lead sections converging at a vertex

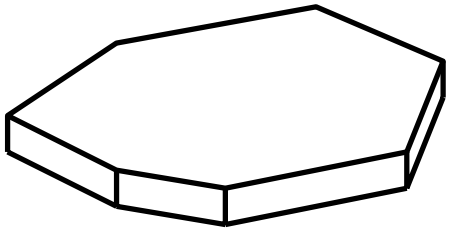


Figure 3.30 POV-Ray glass facet

Style	'Bench' produces a pattern over a flat surface, viewed from directly above to get an idea of the precise layout. 'Sun Beam' uses the POV-Ray 'media' effect and a spotlight to create an aesthetically pleasing picture. See appendix H for examples of both styles.
Background	Selects a black or grey background, or a plane with a colour map that resembles a stone surface.
Lead	'Dark' and 'Grey' finishes give the lead blackened and untreated appearances respectively. A 'Light' finish makes the lead structure more visible.
Glass	The 'Slightly Bumpy' and 'Very Bumpy' options apply a bump map to the surface of the glass facets, causing it to appear bumpy like textured or antique glass.

4 Evaluation

This section lists some of the tests that were used on Vitrigraph to verify the correct operation of the program. The message system in the **vitrigaph.debug.Debug** class was used extensively to report on the operations. Recommendations resulting from the user trials are listed, followed by an explanation of how the finished program fulfills the original requirements. Finally, future enhancements to the program are suggested.

4.1 Program Testing

All components of the program were tested explicitly. All bugs found were fixed. Some of the more elusive ones were due to errors in the Java 1.2 implementation from Sun. The program was moved from the beta version of the development kit to the full version when it became available, but this still contained many errors. One example of such an error is the routine to calculate the bounding rectangle for an arc, which returns a patently incorrect result. This and other problems were overcome by exposing the bug and selecting an alternative implementation strategy.

Pattern Editing
All of the operations through which graphical manipulation of stained glass patterns is achieved, were tested by using them with large and small collections of edges of different types, orientation and lead settings.

- Examples of the four edge types were created, manipulated and used to produce a variety of patterns.
- Intersections of vertices with edges of each type were tested by dragging vertices onto edges in many different arrangements.
- Examples of the 16 combinations of pairwise edge intersections were formed and tested with the edges in various orientations. Possibly problematic configurations were used such as the intersection of two orthogonal axis-aligned straight edges.
- Facets were formed from combinations of edge types, then filled with different glass types and manipulated by moving their constituent edges and vertices.
- Several full patterns were created to test, in particular, the performance of the system when the number of interacting components in the pattern is large.
- All of the above tests were performed with the debugging graphics turned on. These are activiated by setting constants in the **vitrigraph.debug.Debug** class, and cause additional information to be displayed about primary and secondary edge counts of vertices, tangent vectors of edges, bounding boxes of facets, and other properties of the components of a pattern. This extra information allows any anomalies in attributes that are usually hidden to be quickly recognised.
- Debugging messages were also turned on, which provide a textual stream of details about the current state of the pattern and the operations of the various methods invoked when it is altered. This output has not been included as an appendix because it is verbose and voluminous.

The operations described here are illustrated with screen shots in appendix D.

Files
The patterns used to test the program were saved and loaded, with bug fixes being completed inbetween. The system for handling files was verified as working correctly.

Printing
A large format printer was not available during testing, so patterns of different sizes were printed to postscript files which were inspected to ensure that all of the components were present and represented correctly. Different printing sizes were used, from A4 to 36”×96”. A4 printouts were produced on paper of patterns containing facets of known size and lead width. The printed versions of the facets were measured to ensure they had exactly the correct dimensions. Example printouts have been included as appendix E.

Pattern Evaluation
Experimentation with different methods for financially evaluating patterns was conducted as described on page 24. A set of test patterns was used, which were given accurate costs in advance. The patterns and details of their financial evaluations are shown in appendix F. Other patterns were used which were formulated to be unusual in some way, such as one with no facets or edges, to test the behaviour of the evaluation algorithm. Giving costs to stained glass windows is a subjective procedure, and depends on the experience of the person who is going to construct them. All of the patterns in the test set have a relative error of less than 0.1 in their evaluations, apart from pattern B which has a high cost because large round windows of this type are difficult to assemble to exactly the correct size and shape. Capturing these kinds of notions in a general purpose evaluation algorithm is difficult. The Vitrigraph financial evaluation system is intended to be a guide to pricing, rather than an exact measure.

Ray-tracing
All of the test patterns plus others have been translated to ray-tracer format. All of the translation options have been used and subjected to aesthetic evaluation. Inspection of the POV-Ray script files was used to ensure that they not only produce a faithful and eye-catching interpretation of the pattern, but also that the file is structured such that it is amenable to perusal and modification by a suitably experienced human user. Plain text comments and tabulation, for example, are used to make the file easier for a person to read. Appendix G is an example of a simple pattern translated into a POV-Ray file. Appendix H shows the ray-traced version of a more complex pattern. Certain constants in the **vitrigraph.debug.Debug** class were used during testing to activate the inclusion of additional objects in the POV-Ray file, to convey useful information such as the direction of the axes.

Lead in the ray-traced representation of a pattern was originally given a surface finish using specular reflection, plus other definitions from one of the standard ‘metal’ finishes supplied with POV-Ray. Most of the intricacy of the ray-traced components is in the lead, yet the appearance of the lead is much less important than that of the glass. The original finish was therefore replaced with one incorporating only ambient and diffuse reflection, with no appreciable loss in image quality. The change made patterns faster to render because of the decrease in the number of light rays processed by the ray-tracer. The POV-Ray file in appendix G using the new lead finish took 3 minutes 53 seconds to render at 640×640 pixels with anti-aliasing on a Pentium 300MHz machine. Using the old finish it took 5 minutes 34 seconds.

4.2 User Trials

Two people were used to test Vitrigraph from the perspective of a user with basic computer skills. They were given access to the program documentation (appendix I) and were then asked to create a moderately complex pattern, save and load it, print it, use the financial evaluation system on it, translate it to a ray-tracer file and view the rendered result. With a minimal amount of practice the users were able to perform all of these tasks competantly. In general the graphical user interface and graphical method for manipulating parts of a pattern quickly became familiar. A number of issues were discovered and are listed below under the broad categories of those that would be relatively easy to ammend/implement, and those that would require more fundmental changes to the program.

Minor Changes

- An extra tool would be useful that allows a new vertex to be created at any point on a line. A new vertex could be created at the position of a mouse click, then snapped to an edge which it would then split by the same mechanism used for snapping existing vertices to edges, as described on the bottom of page 19.

- The way the shape of an arc relates to it's three vertices could be made more intuitive, by using the current position of the mouse (transformed into pattern space) to draw the arc before the third vertex has been created. This would allow the user to receive feedback on how the proposed position for the new vertex will affect the arc.

- The manipulation of the pattern would be aided by the ability to drag edges (as well as vertices) with the mouse. This could be implemented by detecting an intersection between the mouse pointer and an edge, then moving the vertices associated with that edge.

- A tool would be useful that pulls the end of an edge away from a vertex shared by more than one edge. This could be achieved by replacing the primary vertex of one of the edges by a copy of the common vertex.

Major Changes

- A more flexible editing system would be effected by separating the snapping of vertices to edges, from the splitting of edges. Vertices could be attached to edges without splitting them in two (see bottom of page 33).

4.3 Achievement Criteria

The requirements of the project are listed below. Each is accompanied by an explanation of how it has been satisfied.

- **The user must be able to create a stained glass pattern using simple constructs that are combined graphically.** Manipulation of patterns is simple and straightforward. Optimisations have ensured the interactive user interface remains responsive even for large, complex patterns. All of the editing features work properly as described on page 29.

- **The graphical representation of the pattern should relate directly to the intended physical design, and incorporate information such as the type of each piece of lead and glass.** This is clearly the case. The Java routines used to create the printouts are also used to draw the pattern on the screen, so what you see is what you get.

- **It must be possible to load and save designs from and to files.** This facility has been tested as described on page 30.

- **It must be possible to produce printouts of the whole pattern or parts of it, from which the stained glass window can be constructed.** This facility has been tested as described on page 30. It provides an accurate method for cutting the glass facets. A stained glass window is usually assembled on top of a life-size pattern. Producing this type of printout from Vitrigraph requires a large-format printer for a stained glass window of a reasonable size. Prices for this type of printer in 1999 are around £3000 to £6000 – within the range of a company selling a modest number of stained glass windows.

- **Financial evaluation routines should compute the cost of producing the window. The routines should give an accurate costing for patterns of varying size and complexity.** The evaluation routines give quite accurate cost analyses for the test patterns (page 30). These can be used as a guide to pricing, but the price of a stained glass window should ultimately be set by an expert. This is because there are so many factors that may affect the difficulty of constructing a particular pattern: particular types of glass may be harder to cut than others, certain window shapes will be more susceptible than others to slight errors in the cuts, and the craftsman will be more effective at constructing a pattern in a style with which he is familiar.

- **The program must be able to present the pattern in a form that makes it easy to visualise the finished product. Aesthetic evaluation must be possible.** The ray-tracer translation system allows patterns to be assessed aesthetically by people with no experience in stained glass or computing. This is clear from the example ray-tracer output in appendix H.

4.4 Future Enhancements

A number of possible enhancements to the Vitrigraph program have been identified:

- The additional methods for editing patterns listed in the user trials section (page 31) would increase the ease with which patterns can be created. Other useful additions include rulers to provide continual information about lengths and positions, and the ability to scale a selected part of the pattern. Due to the way facets are comprised of edges, and edges interpolate vertices and are invariant under various transformations (page 10), the scaling could be achieved by simply transforming the locations of the vertices.

- An algorithm to arrange for more than one facet to be printed on each sheet of paper would be useful. This would save paper, especially when using an A3 printer which probably will not be large enough to print the whole pattern on one sheet, but which will waste paper if one sheet is used for each facet.

- Entry, expert costing and analysis of large numbers of patterns would be a lengthly process, but would be necessary if a more accurate evaluation system was needed. More data from the pattern such as the internal angles of each facet, could be used in a more complex algorithm. Experimentation with new algorithms would be simple given the current prototyping system in Visual Basic.

- Two extensions were specified in the proposal for this project: ray-tracer output has been fully implemented, while the use of glass textures has been left as a future enhancement. If bitmap pictures of all of the types of glass could be obtained, from the supplier or through the use of a digital camera, they could be processed by one of the commonly available programs that produce 'tile' images for the backgrounds of web pages. The bitmaps would be stored with the **LeadType** objects, and the Java2D API could fill the facets on the screen with the appropriate textures. POV-Ray supports the use of such textures, so more realistic ray-traced images could also be created. In addition, a variety of bump maps could be created so the bumpy surface of different types of glass could be reproduced by the ray-tracer.

Figure 4.1
'Spectrum Green Blue Opal' glass from the James Hetley web site

- The ease with which patterns are manipulated could be increased by allowing the position of a vertex to be specified either as an absolute location in pattern space, as is currently the case, or as a parametric position relative to another edge. This would require the data structure for an edge to support an unbounded number of sections, each of which could be part of up to two facets. A pattern would then be comprised of vertices, edges, edge sections and facets. The data structures and associated algorithms for most of the program would consequently be more complex.

5 Conclusions

A system has been contrived that can represent a wide variety of stained glass patterns and an efficient implementation has been created. The source code for the program consists of approximately ten thousand lines of code contained in 51 Java class definition files.

If more work was to be done on the program, the most substantial improvement would be gained by adding the ability to specify vertex locations relative to edges as discussed on page 33. This would form a more general and expressive structure for creating patterns. It was left out because it was a fundamental decision and it was not known how difficult the basic system would be to implement. With hindsight it seems this was the right decision: it would have made many parts of the program significantly more complex.

Vitrigraph is a working program that can be used to design and evaluate stained glass windows, and aid with physical construction. It offers a more pragmatic approach to their production than other programs available for a similar purpose. If more time was available, many small additions could be made to the user interface, which would help to give the computer-supported design method the same flexibility as a pencil and paper while employing the many advantages of a digital medium.

Bibliography

[1] Baugmart B. G., 'A polyhedron representation for computer vision' in *vol 44 of AFIPS National Computer Conference Proceedings*, pp589-596, 1975.

[2] Cormen T. H. Leiserson C. E. and Rivest R. L., *Introduction to Algorithms*, MIT Press, 1996.

[3] Flanagan D., *Java in a Nutshell 2nd ed.*, O'Reilly, 1997.

[4] Foley J. van Dam A. Feiner S. Hughes J., *Computer Graphics: Principles and Practice 2nd ed.*, Addison Wesley, 1997.

[5] Frohbieter-Mueller J., *Practical Stained Glass Craft*, David & Charles, 1984.

[6] Murray J. D. and vanRyper W., *Encyclopaedia of Graphics File Formats 2nd ed.*, O'Reilly & Associates, 1996.

[7] Pressman R.S., *Software Engineering*, McGraw-Hill, 1996.

[8] Rade L. and Westergren B., *Beta Mathematics Handbook 2nd ed.*, Chartwell-Bratt, 1992.

Appendix A – Program Structure

Listed below are the six Java packages into which the source code for the program was arranged (see page 8). The main object classes from the diagram on page 7 are listed, together with brief explanations.

main	
Shell	Initialises global variables and loading of glass and lead data, launches rest of the program.
VGGlobal	Stores global constants and variables: version information, filenames, colours, numeric constants. May be used by any part of the program.
gui	
MainWindow	The main program window with which the user interacts. Receives GUI events, which it may pass to other objects. Page 13
InfoWindow	Displays windows to alert the user or prompt for confirmation.
pattern	
SGForger	Stores a stained glass pattern. Allows edges and points to be added, selected, moved and deleted. Maintains an internal state of the pattern.
SGPattern	Stores the core data of the pattern that is passed to the translator, evaluator, file access routines and print routines.
SGVertex	A vertex. Page 15
SGEdge	An edge. Page 16
SGFacet	A facet. Page 22
LeadData	Loads, saves and manipulates data about all available types of lead.
LeadType	Data about a particular type of lead.
GlassData	Loads, saves and manipulates data about all available types of glass.
GlassType	Data about a particular type of glass.
tool	
Editor	Receives user events from the MainWindow object and acts upon them to edit the stained glass pattern.
Evaluator	Implements a number of evaluation algorithms that calculate the cost of the pattern. Provides a user interface to alter settings and outputs the results. Page 24
Translator	Implements translation of the pattern into a file suitable from rendering by a ray-tracer. Provides a user interface to alter settings that affect the translation. Page 28
io	
FileAccess	Handles all loading and saving of data for different file format versions. Provides a user interface to select file names, types and locations if necessary. Page 26
Print	Prints individual facets, or the whole pattern. Provides a user interface to select printer settings and output styles. Page 27
debug	
Debug	Provides a uniform method for creating debugging messages of varying priorities. Provides a method to screen out messages below a certain priority, or to switch them off altogether for the final version of the program. Always displays a message to the user following a critical program error. May be used by any part of the program.

Appendix B – Pattern Representation Classes

This appendix lists the data fields and main externally visible methods of the **SGVertex**, **SGEdge** and **SGFacet** classes. A group of instances of these three classes can jointly represent a stained glass pattern. An **SGPattern** object contains vectors of such instances, to represent a pattern in a single object.

SGVertex Class

Data in the **SGVertex** class:

pos	Two-dimensional position of this vertex in pattern space
primaryEdges	Vector of edges for which this is a primary vertex
secondaryEdges	Vector of edges for which this is a secondary vertex
redundant	Tag to indicate this vertex has been classified redundant
sharpAngleTag	Tag to indicate this vertex forms an internal reflex angle in a facet

Main externally visible methods of the **SGVertex** class:

SGVertex	The constructor. Creates a vertex object, given it's location in pattern space
addPrimaryEdge removePrimaryEdge primaryEdge	Manage the primary edges of this vertex
addSecondaryEdge removeSecondaryEdge secondaryEdge	Manage the secondary edges of this vertex
draw	Draws this vertex
getPos getX getY setPos	Return or set this vertex's position in pattern space
in	Determines whether this vertex is inside a specified rectangle in pattern space
getSharpFacetAngle setSharpFacetAngle	Reads and sets the sharpAngleTag field
kill	Makes this vertex redundant and validates all of it's primary and secondary edges
isRedundant	Returns true if this vertex has been classified redundant (killed)
killRedundantEdges	Kills any duplicate primary or secondary edges
replaceWith	Replaces this vertex throughout the pattern with a different vertex
translate	Translates this vertex in pattern space
squareDistanceFrom	Calculates the square distance of this vertex from a point in pattern space
validate	Causes this vertex to decide whether it is still needed, and kill itself if it is not.
closestEdgeTangent	Returns the primary edge of this vertex that has the tangent angle that is closest clockwise to a specified angle.

SGEdge Class

Data in the **SGEdge** class:

v	Vector storing references to the 2,3 or 4 vertices that define the edge
edgeType	Takes one of a number of integer constants that denote the type of this edge
leadCode	String used to identify the type of lead of which this edge is made
facet	A two-element array holding references to the facets on either side of this edge
redundant	Tag to indicate that this edge has been killed
edgeShape	Caches the graphical representation of the edge
imageValid	Boolean value that indicates whether **edgeShape** is now valid
intersectTag	Tag to indicate if the edge intersects another edge in the pattern
facetTag	Tag for each side of the edge, used in the facet finding algorithm. Page 22

Main externally visible methods of the **SGEdge** class:

SGEdge	The constructor. Creates an edge object given 2,3 or 4 vertices, and a lead type
getVertex	Returns one of the primary vertices specified by an index
replaceVertex	Occurances of a specified vertex in **v** are replaced with a different vertex
associateWithFacet deassociateWithFacet getFacetAssociation	Manage the **facet** vector
setFacetTag clearFacetTags getFacetTag	Manage the **facetTag** settings
currentEdgeShape	Returns an object that defines the graphical representation of this edge
draw	Draws this edge
Hardcopy1 Hardcopy2	Draws this edge when creating a print out
equals	Tests for equality between this edge and another
getBoundingBox	Returns the rectangular bounding box for this edge in pattern space
setIntersectTag getIntersectTag	Set and get the **intersectTag** field
getLeadCode	Returns the **leadCode** string
getLength	Calculates the pattern space length of this edge for financial evaluation. Page 24
getSharpestEdge	Returns the sharpest edge following this one. Used for facet detection. Page 22
getTangentAngle	Returns the tangent angle at one end of the edge
intersects	Calculates whether this edge intersects a specified rectangle
intersectsEdge	Determines whether this edge intersects another edge
invalidateImage	Called by a vertex to invalidate the **edgeShape** that is cached to speed up drawing
validate	Validates the edge, possibly performing edge simplifications or killing it. Page 20
kill	Marks the edge as redundant and validates it's vertices
isRedundant	Returns true if this edge has been killed
subdivide	Subdivides the edge into two, given a vertex to form the common point

SGFacetClass

Data in the **SGFacet** class:

edges	A vector storing the edges that form the facet perimeter
glassCode	Used to identify the type of glass of which this facet is made
facetShape	Caches the graphical representation of the facet
imageValid	Boolean value that indicates whether **facetShape** is now valid
redundant	Tag to indicate that this facet has been killed

Main externally visible methods of the **SGFacet** class:

SGFacet	The constructor. Creates a facet object given a glass code
addEdge removeEdge replaceEdge containsEdge getEdges	Manage the vector of edges
draw	Draws this facet
drawHardcopy	Draws this facet when creating a printout
getBoundingRectangle	Returns the rectangular bounding box for this facet in pattern space
getGlassCode	Returns the **glassCode** string
kill	Marks this facet as redundant
isRedundant	Returns true if this facet has been killed
markSharpAngles	Sets the **sharpAngleTag** field of the vertices that form reflex internal angles. Page 22
invalidateImage	Called by an edge to invalidate the **facetShape** that is cached to speed up drawing
setGlassCode getGlassCode	Set and get the **glassCode** string
validate	Validates the facet, possibly killing it

Appendix C – Example Code

The source code in this appendix is that for a method of the **SGEdge** class that splits the edge with a vertex. If the edge is a cubic edge this method is called, and passed the vertex as the parameter **p**. Extra comments have been inserted into the code to explain it more thoroughly than the Java comments.

```
/**
  *  Subdivides this cubic edge, creating a new cubic edge.
  */
private SGEdge subdivideCubicEdge(SGVertex p)
{
```

The relevant points in pattern space are referenced by the following **SGVertex** and **SGPoint** variables.

```
   final SGVertex
      f = v[0],           // The vertices that define this edge
      g = v[1],
      h = v[2],
      i = v[3];
   final SGPoint          // Control points of the bezier
      b = controlPointPos(g.getPos(),f.getPos(),h.getPos()),
      c = controlPointPos(f.getPos(),g.getPos(),i.getPos());
```

The x and y coordinates of the relevant points are put into **final** values to speed up computation.

```
   final float
      ax = f.getX(),       // The coords of the three points that define the
      ay = f.getY(),       // cubic bezier
      bx = (float)b.getX(),
      by = (float)b.getY(),
      cx = (float)c.getX(),
      cy = (float)c.getY(),
      dx = g.getX(),
      dy = g.getY(),
      px = p.getX(),       // The coords of the point that's going to divide the bezier
      py = p.getY();
```

The values of **t0** and **t1** will start at 0 and 1 (the two ends of the cubic Bézier curve) and move towards each other until they converge on a point close to the vertex that will split the edge.

```
   float
      t0 = 0,              // Two values of t that home in on the required value
      t1 = 1;

   // Iteratively home in on the value of t that defines the
   // point on the bezier closest to p

   for(;;)
   {
```

The standard formula for cubic Bézier is used to calculate two points (**x0**,**y0**) and (**x1**,**y1**) on the curve.

```
      // Calculate square distance of p from first point on Bezier
      final float
         s1    = 1-t0,
         tt0   = t0 * t0,
         x0    = s1*(s1*(s1*ax + 3*t0*bx) + 3*tt0*cx) + tt0*t0*dx,
         y0    = s1*(s1*(s1*ay + 3*t0*by) + 3*tt0*cy) + tt0*t0*dy,
         dist0 = (x0-px)*(x0-px) + (y0-py)*(y0-py);

      // Calculate square distance of p from second point on Bezier
      final float
         s2    = 1-t1,
         tt1   = t1 * t1,
         x1    = s2*(s2*(s2*ax + 3*t1*bx) + 3*tt1*cx) + tt1*t1*dx,
         y1    = s2*(s2*(s2*ay + 3*t1*by) + 3*tt1*cy) + tt1*t1*dy,
         dist1 = (x1-px)*(x1-px) + (y1-py)*(y1-py);
```

The point on the curve which is farthest from the vertex, moves closer.

```
      if (dist0<dist1) t1 = (t0+t1)/2;
      else t0 = (t0+t1)/2;
```

When a threshold is reached, the point is deemed close enough. The vertex is ‘snapped’ to the position on the curve.

```
      if (t1-t0 < 0.0005f)
      {  p.setPos(x0,y0);
         break;
      }
   }
```

The vertices of this edge are changed, and the primary and secondary edge vectors of the vertices are updated.

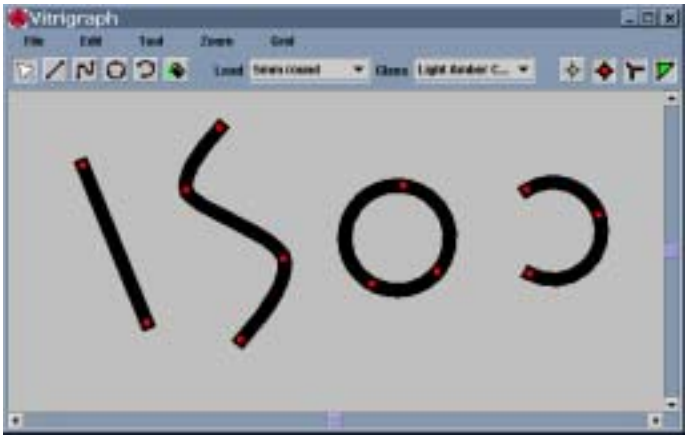
```
   this.replaceVertex(f,p);
   this.replaceVertex(h,f);
   f.removePrimaryEdge(this);
   f.addSecondaryEdge(this);
   h.removeSecondaryEdge(this);
   p.addPrimaryEdge(this);

   f.inPrimaryEdgesReplaceSecondaryVertex(g,p);
   i.inPrimaryEdgesReplaceSecondaryVertex(f,p);
```

A new edge using some of the original vertices of this edge, plus the vertex **p**, is created and returned from the method to be included in the pattern data structure. A redundancy check by vertex **p** ensures that any edges that are now duplicates of other edges are deleted.

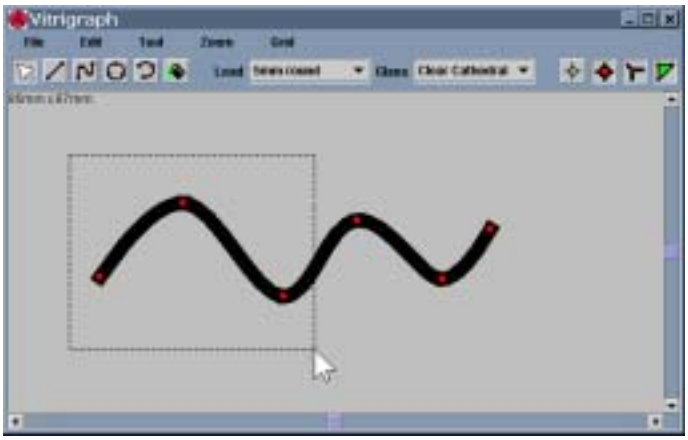
```
   SGEdge newEdge = new SGEdge(f,p,h,g,leadCode);
   p.killRedundantEdges();
   return newEdge;
}
```

Appendix D – Screen Shots

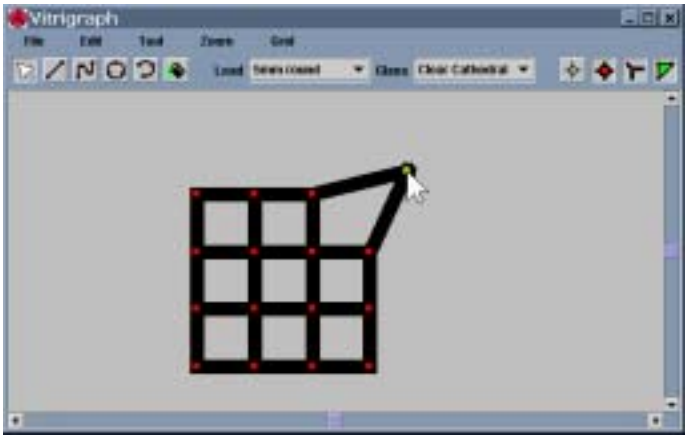
This appendix contains screen shots of the graphical user interface of Vitrigraph. Pictures demonstrating the use of the tools for creating and altering stained glass patterns are included.



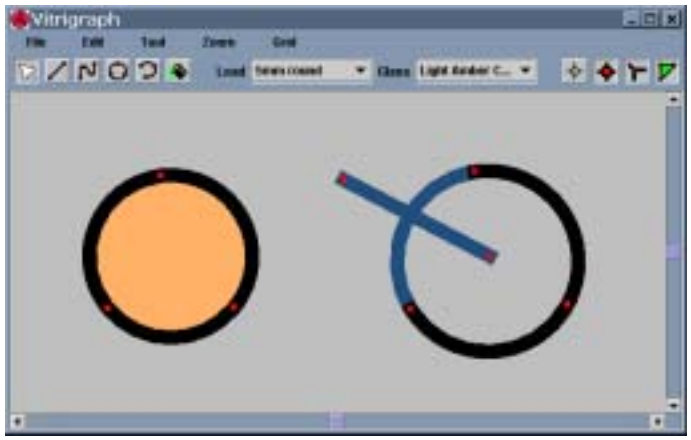
Results of the line, curve, arc and circle tools



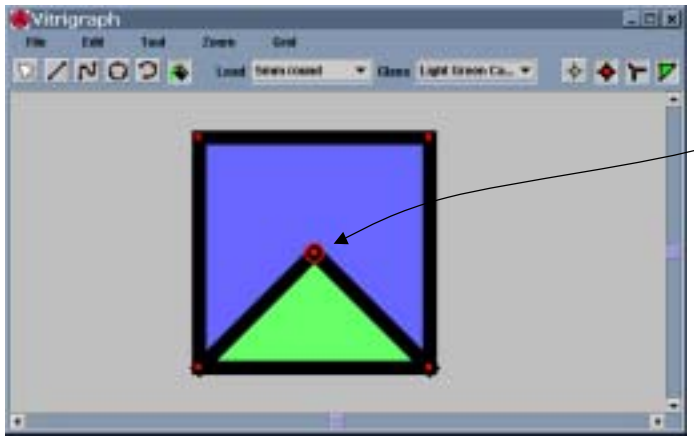
Rubberbanding to select vertices



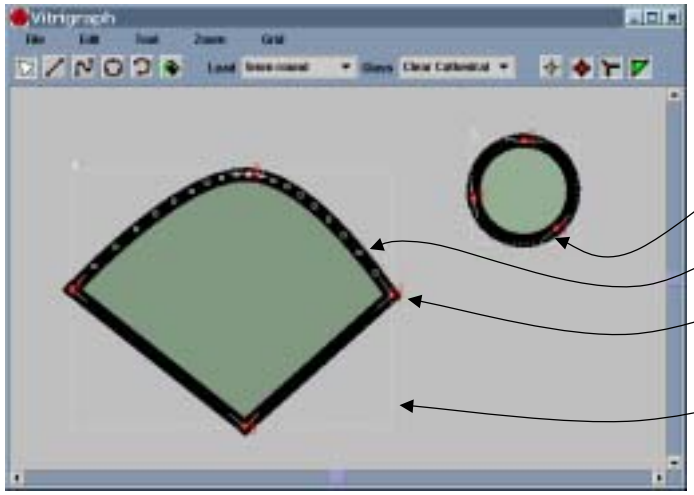
Dragging a vertex to a new position



Left: The facet is filled with amber glass.
Right: The edge intersection means that no facet is created.



A reflex internal angle in the upper facet causes a vertex to he highlighted.



Debugging graphics

Tangent angles

Curve fitting guidelines

Counts of primary and secondary edges

Facet bounding boxes with edge counts

Appendix E - Printouts

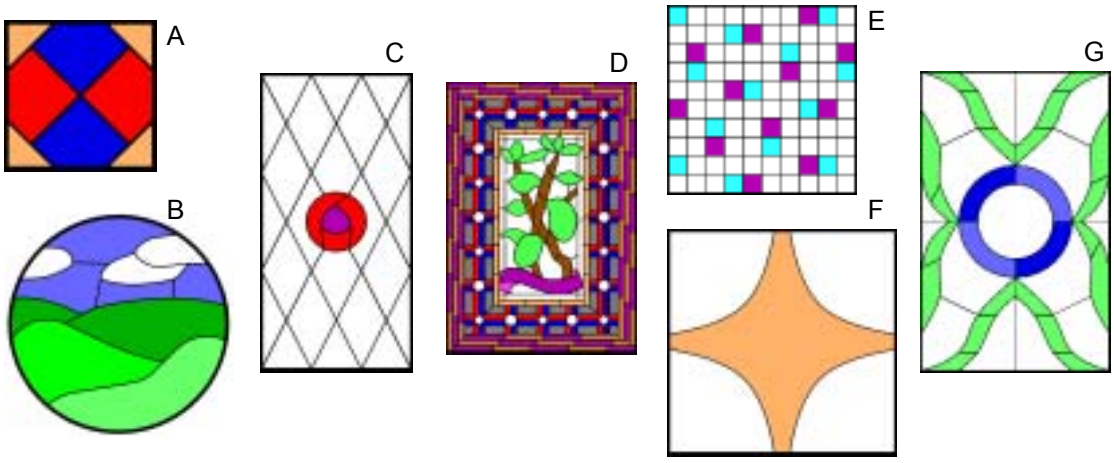
The next two pages are examples of the printouts produced by Vitrigraph. They have both been generated from pattern A in appendix F. The first page contains a scaled, numbered version of the whole pattern. The second contains one of the facets. The black and grey lines show the heart and face widths of the lead. The facet is the correct size: 150mm from left to right and 200mm from top to bottom.

PRINTOUT 1

PRINTOUT 2

Appendix F – Financial Evaluation Testing

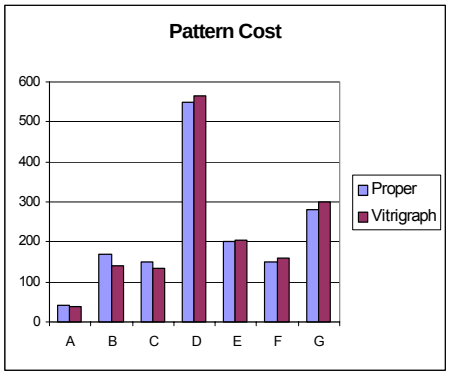
The stained glass patterns used to test the financial evaluation algorithms are shown below.

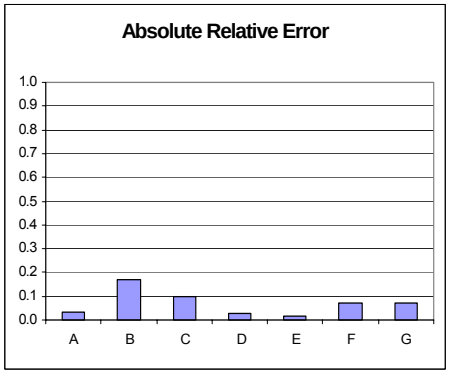


Pattern	Width (m)	Height (m)	Facets	Edges	Vertices
A	0.3	0.3	8	24	17
B	0.9	0.9	11	55	45
C	0.6	1.2	28	70	43
D	1.0	1.4	340	1114	775
E	1.0	1.0	100	220	121
F	1.5	1.5	5	28	24
G	1.0	1.6	41	101	61

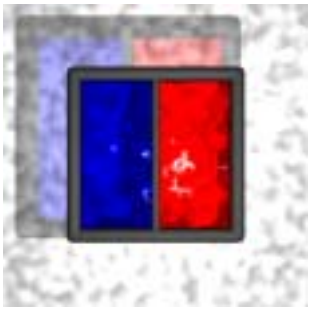
The table below contains the expert and Vitrigraph-generated costs for the test patterns, taken from the spreadsheet used to experiment with financial evaluation algorithms. The graphs show that the Vitrigraph evaluations are quite close to the proper costs, and would give a useful guideline when pricing a pattern.

Pattern	Expert	Vitrigraph	Error
A	40.00	38.70	0.03
B	170.00	141.43	0.17
C	150.00	135.03	0.10
D	550.00	565.81	0.03
E	200.00	203.03	0.02
F	150.00	160.95	0.07
G	280.00	300.61	0.07





Appendix G – Ray-tracer Output



The file shown in this appendix was created by Vitrigraph then used by POV-Ray to generate the ray-traced image on the left. This simple example demonstrates how the parts of the stained glass window are represented using POV-Ray components.

```
// Created by Vitrigraph

//The stone slab background
plane {
  <0,-1,0>, -100
  texture {
    pigment {
      bozo
      turbulence 0.25
      color_map {
        [0 color rgb <0.9, 0.9, 0.9> ]
        [1 color rgb <0.6, 0.6, 0.6> ]
        }
    scale 2
    }
    finish{ ambient 0.4 }
    }
  }
```

Stone slab background

```
//The camera looks at the centre of the window
camera {
  location <417.000000,-15.599987,417.000000>
  look_at <417.000000,0.000000,417.000000>
  up <0.000000,0.000000,-1.000000>
  right <1.000000,0.000000,0.000000>
}
```

Camera

```
//A light to the top right of the viewer
light_source {
  <419.399951,-15.599987,419.399951>
  fade_distance 15.599987
}

//A light behind the centre of the window
light_source {
  <417.000000, 60, 417.000000>
  color rgb 1
  fade_distance 0.529166
}
```

Light sources

```
//The appearance of the lead
#declare Lead_Texture  =
texture {
  pigment { color rgb 0.35 }
  finish {
    metallic
    ambient 0.2
    diffuse 0.85
    roughness 0.05
    brilliance 2.0
    }
  }
```

Surface finish of the lead

```
//The following cylinders represent the lead in the pattern
cylinder {
  <411.000000,-0.250000,411.000000>,
  <411.000000,-0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <411.000000,0.250000,411.000000>,
  <411.000000,0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,-0.250000,423.000000>,
  <423.000000,-0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,0.250000,423.000000>,
  <423.000000,0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <423.000000,-0.250000,423.000000>,
  <423.000000,-0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <423.000000,0.250000,423.000000>,
  <423.000000,0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,-0.250000,411.000000>,
  <411.000000,-0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,0.250000,411.000000>,
  <411.000000,0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <411.000000,-0.250000,423.000000>,
  <416.999951,-0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
```

Edges

```
cylinder {
  <411.000000,0.250000,423.000000>,
  <416.999951,0.250000,423.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <423.000000,-0.250000,411.000000>,
  <416.999951,-0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <423.000000,0.250000,411.000000>,
  <416.999951,0.250000,411.000000>, 0.500000
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,-0.250000,411.000000>,
  <416.999951,-0.250000,423.000000>, 0.299999
  open
  texture { Lead_Texture }
}
cylinder {
  <416.999951,0.250000,411.000000>,
  <416.999951,0.250000,423.000000>, 0.299999
  open
  texture { Lead_Texture }
}
```

Edges

```
//The following spheres represent the vertices in the pattern
sphere {
  <411.000000,-0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <411.000000,0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <411.000000,-0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <411.000000,0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <423.000000,-0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <423.000000,0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <423.000000,-0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <423.000000,0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <416.999951,-0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <416.999951,0.250000,423.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <416.999951,-0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
sphere {
  <416.999951,0.250000,411.000000>
  0.500000
  texture { Lead_Texture }
}
```

Vertices

```
//The definition of the glass finish
#declare Glass_Finish=
finish {
  specular 1
  roughness 0.001
  ambient 0
  diffuse 0
  reflection 0.1
  refraction 1
  ior 1.5
}
```

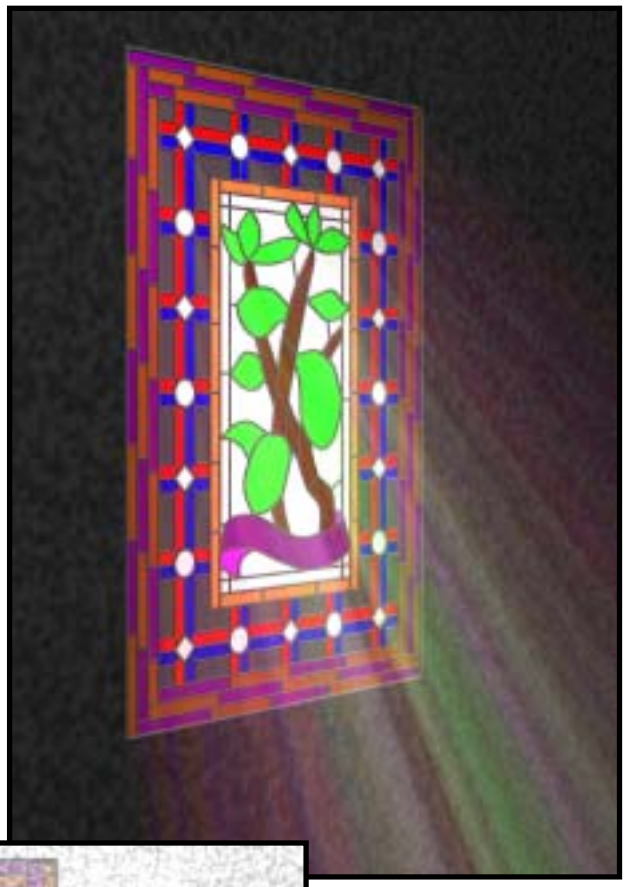
Surface and internal appearance of the glass

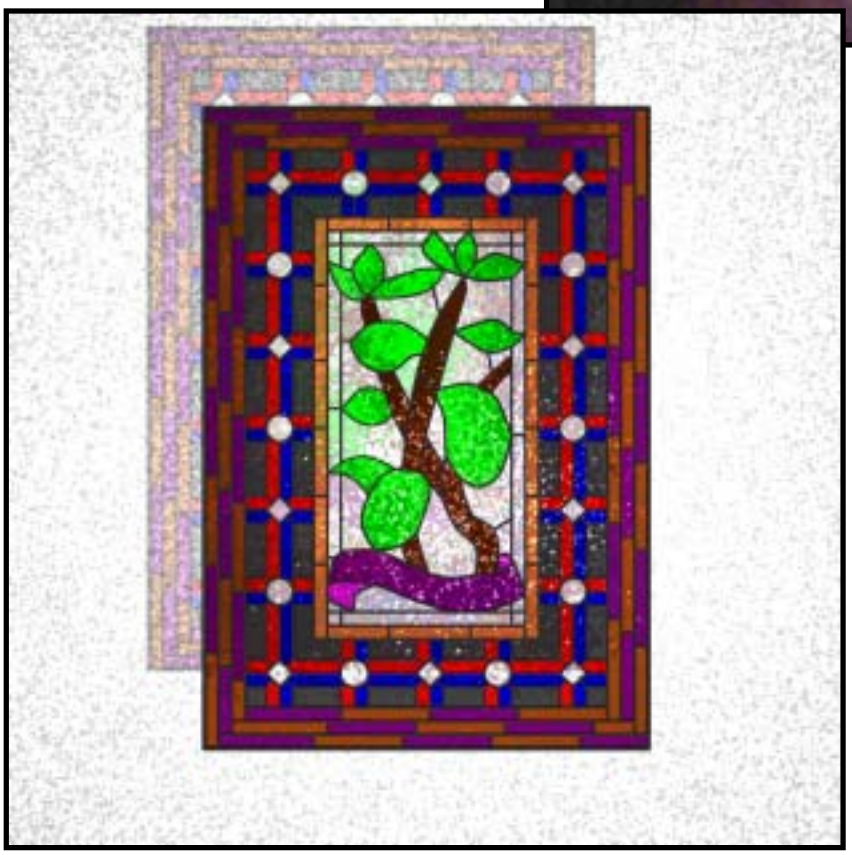
```
//The following prisms represent the glass facets
prism {
  linear_sweep
  linear_spline
  -0.200000, 0.200000
  5
  <416.999951,423.000000>,
  <416.999951,411.000000>,
  <423.000000,411.000000>,
  <423.000000,423.000000>,
  <416.999951,423.000000>
  texture {
    normal { bumps 0.3 }
    pigment { rgbf<1.000000,0.000000,0.000000, 0.94> }
    finish { Glass_Finish }
  }
}
prism {
  linear_sweep
  linear_spline
  -0.200000, 0.200000
  5
  <416.999951,411.000000>,
  <416.999951,423.000000>,
  <411.000000,423.000000>,
  <411.000000,411.000000>,
  <416.999951,411.000000>
  texture {
    normal { bumps 0.3 }
    pigment { rgbf<0.000000,0.000000,0.900000, 0.94> }
    finish { Glass_Finish }
  }
}
```

Facets

Appendix H – Rendered Pattern

These demonstrations of the ‘sunbeam’ (right) and ‘bench’ (bottom) options for ray-tracer files were rendered from test pattern D in appendix F. The images would be shown to a customer to convey the aesthetic qualities of the pattern, before commencing construction.





Appendix I – Program Documentation

Vitrigraph

Stained Glass Window Designer

Contents

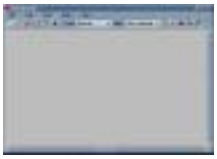
Getting Started

Vitrigraph is a program for designing stained glass windows. It requires Java 1.2. If you do not already have a copy of Java 1.2 installed, it is available for Windows, Unix and Linux from http://java.sun.com/

The program is executed by starting the **vitrigraph.main.Shell** class using the Java implementation suitable for your hardware and operating system.

Creating Designs

The main program screen looks like this:



The menus have these functions:

File Allows designs to be loaded and saved, and new designs created. Also initiates printing.

Edit Allows parts of designs to be cut, copied, pasted and deleted.

Tool Selects one of the six drawing tools on the tool bar.
Translates the design to a POV-Ray file.
Evaluates the design to obtain a price based on raw materials and complexity.

Zoom Sets the zoom level used to display the design.
At 100% zoom the design is shown approximately life-size.

Grid Turns the grid off, displays the grid, and sets snap-to-grid on. When snap-to-grid is on, vertices that are moved snap to the nearest grid position when they are released.
The 'Choose Grid Size' option allows the separation of the grid dots to be entered in millimetres.

The stained glass design is created by drawing lengths of lead using the tools described below. The lengths of lead are drawn between points called vertices. The design is modified by moving, deleting and creating vertices. Areas between the lengths of lead can be filled with glass.

The tool bar:

The six buttons on the left of the tool bar select the tools that can be used to draw the stained glass window design. They are:

Pointer Vertices can be selected individually, or by dragging a box around a group of vertices.
Vertices can be dragged around.

Line A straight line is drawn by clicking on the start and end points of the line.

Curve A curved line is drawn by clicking on two or more points that are on the line.

Circle Three points are created, through which the circle passes.

Arc Three points are created. The first two are the start and end points of the arc.
The third one is another point on the arc.

Fill Fills an area surrounded in lead, with glass.

The lead selector, allows the current type of lead to be chosen, which is subsequently used to form lines drawn with the line, curve, circle and arc tools. The glass selector allows the current type of glass to be chosen, which is subsequently used to fill glass pieces with the fill tool.

The four buttons on the right of the toolbar are:

Centre Moves the scrollbars on the window so that the middle of the design is in the middle of the window.

Show Vertices Hides or shows the vertices between which the lead lengths are drawn.

Show Lead Width Selects whether the lead is drawn with its true width, as it will look in the finished stained glass window, or drawn as thin black lines that show the shape of the lead segments, but not the width.

Show Glass Selects whether the glass pieces are displayed or not.

Lead Intersections
When a lead piece appears blue instead of black, this indicates that it crosses over another lead piece. The piece should be moved to avoid this before glass pieces are added.

Inward Angles
When a vertex appears with a circle drawn around it, this indicates that the vertex lies on the outside of a glass piece, and the glass piece contains a sharp inward angle. The glass piece would therefore be very hard to cut. A design that contains such a vertex can still be created and printed - the ring around the vertex is just a warning.

Translating to POV-Ray

The Translate option in the Tool menu allows a stained glass design to be represented as a POV-Ray file. This allows a realistic ray-traced picture of the stained glass window to be generated, before the window is actually built for real. A picture of a ray-traced design is shown below:

When a design has been created for someone, and approval is required before moving on to actually making the window, the designer can select the Translate option. This displays a number of options for the POV-Ray file:

Style	**Bench**	The stained glass piece is displayed, lying flat, with the viewer looking down onto it.
	Sunbeam	The stained glass piece is viewed from an angle, with light shining through from behind.
Background	**None**	No background is used. The space behind the piece is black.
	Grey Plane	The space behind the design is coloured grey.
	Stone Slab	A dappled-grey stone slab is used as a background.
Lead	**Dark**	The lead is dark, like traditional blackened lead.
	Grey	The lead is grey, like unblackened lead.
	Light	The lead is bright, like aluminium
Glass	**Flat**	The glass is totally flat.
	Slightly Bumpy	The surface of the glass is wavy.
	Very Bumpy	The surface of the glass is bumpy, like textured glass.

When the options have been chosen, a file name is given to the translated file, and it is saved. For use with POV-Ray, files are often given a '.pov' extension. The file can be loaded into POV-Ray and rendered at the desired detail level. Note: the 'sunbeam' option produces a nice effect but will be much slower to render that the 'bench' option.

POV-Ray is a free program, and can be obtained from http://www.povray.org/

Once the POV-Ray file has been produced by Vitrigraph, it can be edited to add many varied effects. Information about how to use the POV-Ray scripting language is included with the program, and their are many web sites for enthusiasts.

Evaluating the Cost of a Design

Once a design has been drawn, it can be evaluated financially. The Evaluate option is chosen from the Tool menu.

Vitrigraph calculates:

- The length of each type of lead needed, plus a small amount of wastage for each piece. The total cost of the lead is calculated.
- The area of each type of glass needed, including a small addition to the width and height of each piece due to wastage.
- The number of glass pieces, and their complexity. This and the basic facet cost are used to get a cost for the complexity of the fabrication of the design.
- The area of the design. This is used in a formula to calculate the cost of producing the design, independent of complexity.

From the above amounts, the total cost of fabricating the design can be calculated. Extra amounts such as profit and VAT for a stained glass window that is being made for sale to a customer are intended to added onto the cost calculated by the Vitrigraph evaluation.

Before evaluation begins, variables that affect the calculations can be altered. An explanation of the purpose of each variable, plus it's default value, is given below.

Lead Excess	The length added to each lead piece to account for wastage. Default = 15mm
Glass Excess	The length added to the width and height of each glass piece. Default = 10mm
Basic Facet Cost	An amount that determines how much is added to the cost of the design by having many facets, and facets that are complex and therefore hard to cut. Default = £1.30
Area Cost	The basic cost per square metre for any design. Default = £30
Minimum Area Cost	The minimum cost for the stained glass window, even if everything else is free. Default = £16

Printing

The Print option in the File menu prints the design to paper. Before a design is printed, the print options window is first displayed. The 'Paper Size' option allows one of a number of standard paper sizes to be chosen. The 'Paper Margins' option allows the blank area around the outside of the paper to be reduced if extra space is needed for large glass pieces.

Four printing modes are available:

Scaled One Page Pattern	The whole pattern is printed. It is scaled to fit on one sheet of paper.
Life-size One Page Pattern	The whole pattern is printed life-size. The paper must be big enough to accommodate the whole design. Black and grey parts of the lead show the heart and face of the lead respectively.
Individual Facets, Numbered	The first page contains a scaled, one-page pattern. Subsequent pages each contain one glass piece. Black and grey parts of the lead show the heart and face of the lead respectively. Each glass piece is accompanied by a number that corresponds to it's position in the one page pattern, and the type of glass with which the piece has been filled.
Individual Facets, Not Numbered	As above, except the glass pieces are not numbered.

When the print options have been selected, a window appears where the printer should be selected. Clicking on 'Ok' then prints the design.

The printouts of individual facets look like this:

The glass type of the facet above is Grey Cathedral, and the number of the facet is 20, corresponding to the same number which will be on the one page scaled printout. The black line outlining the glass piece, is the heart of the lead. The glass piece should be cut so that it is inside the black outline. The grey line is the total width of the lead. Defects such as chips in the glass will not be noticed if they fall within this area. A life-size one-page pattern is intended to be fixed to a bench so that the glass can be cut, and stained glass window assembled, on top of the print out.