School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Intelligent Flood Fill or: The Use of Edge Detection in Image Object Extraction

Paul André
May 2005

Project Supervisor: Kirk Martinez
Second Examiner: Corina Cirstea

A project report submitted for the award of MEng Computer Science

Abstract

Content-Based Image Retrieval systems can often return poor results when attempting to match images with extraneous feature information such as complex backgrounds, shadows, or frames. As a solution, this project aimed to provide a semi-automatic object extraction tool, the 'Intelligent Flood Fill'. This tool would also help users of graphics packages, who can find current object extraction tools inefficient. The solution implemented is based upon innovations and extensions to the floodfill technique, and is both accurate and efficient in itself, and compared to existing methods. The new algorithms presented allow extraction based on scribbles over the image which gather colour data, and/or a bounding box, outside of which the algorithm will try to revert to the last major colour change. An option for sequence extraction has also been implemented for use in VRML model creation. The architecture is flexible, so as to allow the control logic to be rewritten for another language or application as desired, and the user interface intuitive. Failures can occur when the foreground and background overlap in colour space, or when the image is low resolution or noisy. Extensions include making use of dedicated image loading libraries, and the implementation of a live wire boundary system for complex images.

Table of Contents

Acknowledgements

I would like to thank my supervisor, Kirk Martinez, for his advice throughout this project, my second examiner Corina Cirstea for opinions on making this report readable to those who haven't spent a year researching Computer Vision, and fellow student Sina Samangooei for all the discussions on various aspects of both our projects. Also James Stephenson and Peter Kelleher at the V&A Museum for testing various versions of the tool.

CD Contents

The attached CD contains:

- 'readme' file
- installation and user guide
- all the necessary software needed to run this tool on any PC (Java 1.4.2_06 JRE Installer, JAI 1.1.2_01 Installer, and the Intelligent Flood Fill Installer)
- source code
- sample images for testing

1. Project Description

The impetus for this project comes from current problems in two computer vision areas. Firstly in Content Based Image Retrieval (CBIR) systems such as Sculpteur [9], VisualSEEk [22], QBIC [6], and Virage [2] that give the ability to search images based on the content of the image. These all perform some sort of searching and matching based upon features such as colour, texture, shape and/or spatial information such as size and location of regions. Discussions with partners in the Sculpteur project brought up a problem that can occur when attempting to match two pictures. Images, such as those in fig. 1.1, with shaded or complex backgrounds, shadows, or frames, can throw the matching algorithm by providing extraneous feature information.

An example of this problem is apparent in the query "Find images with a similar colour content [to the picture in fig. 1.1(c)]". Using the picture in fig. 1.1(c) as the query picture would introduce unconnected colours - those in the frame - into the query. Incorrect results would then be returned, based upon this initial inconsistency.

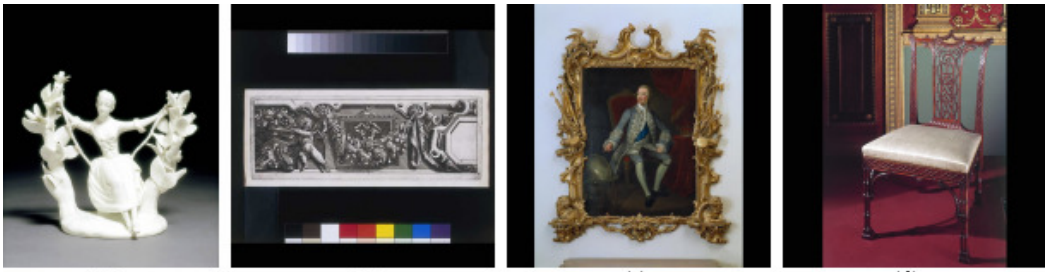


Figure 1.1: Problematic images in CBIR

A related problem can be found in graphics packages; end users can find current tools for object extraction inefficient, or in some cases, completely ineffective. The problems lie in a lack of options or amount of effort involved. Floodfilling or the magic wand tool cater for only single colour objects, with more complex images taking exponentially more time and effort. For advanced object extraction, tools such as the magnetic lasso, while effective, are a hindrance in terms of time and effort.

These problems create the need for a fast and efficient way of extracting objects from images, that will allow museums and partners involved in CBIR systems to match more accurately, and depending on the technologies used, could be incorporated into graphics packages to help users wishing to manipulate objects. Thus, the overall goal is:

- A semi-autonomous tool for extracting objects from images

This main goal will be an amalgam of a user interface and implementations and improvements of existing algorithms. Keeping in mind the dual purpose; that of

CBIR matching and needs of a graphics package user, the algorithms to be looked at are:

- Flood fill – a simple algorithm that can be improved in various ways.
- Edge detection – Sobel and/or Canny edge detection used to gain information about areas of interest in image, and for pre-processing in Snake algorithm.
- Active Contour Model or 'Snake' – with contour points specified by user, this allows a non-uniform area in the image to be identified as an area of interest.

The requirements of the project will be expanded using Use Case diagrams in Section 3, but the main goals and aims are:

- Extraction should be fast and efficient; timings of current best method practices will be used to compare this.
- Tool should allow modification of selection and in general aid the user in selection.
- Algorithm design should be as language independent as possible, for possible re-implementation in various languages.
- Extensibility and flexibility of the architecture should be considered.
- User interface should be intuitive and adhere to any relevant existing design practices.
- Extension of main aim of object extraction is to allow extraction of a sequence of images, for example those used in creating 3D VRML models.

Thus, at completion of the project, the problem posed will have been solved by:

- Creating a standalone tool for professionals, such as museums who have shown interest, to quickly and efficiently extract objects from images.
- The creation of new algorithms to allow successful translation to a graphics package plugin to allow end users to benefit from the development.
- A first look at solving object extraction from sequences.

2. Review of background literature

Feature extraction is a wide and ranging area, with many techniques available, but with no perfect all encompassing method. Many solutions are thus tailored for a specific purpose, using domain specific knowledge to aid their goals. There are a number of techniques that are used again and again – such as edge detection, edge direction and curvature, shape matching by various transforms, active contour models, object description, segmentation algorithms - that form the basis of feature extraction, new solutions generally propose either looking at these algorithms in a new context, or with an extension/improvement/specialisation.

This section looks at the distinction between autonomous and semi-autonomous extraction, an evaluation of similar work to this projects goal, detail on the floodfill and active contour model techniques, and finally a brief review of how colour properties can help in feature extraction.

2.1. On automatic and semi-automatic extraction

Even with assumptions about images afforded by the domain, automatic feature extraction is a problem that has been looked at closely, but with certain conditions or expected failure rates. For example, Chang [4] deems "it will suffice to some extent if 'prominent regions' with 'distinctive features' can be extracted". Although automatic feature extraction is desirable, a quicker, less computationally expensive, and more guaranteed way is to allow a degree of user interaction.

As mentioned, edge detection algorithms such as Sobel [23], and Canny [3], are used in automatic extraction. This project implements Sobel to aid in a Snake algorithm, chosen for both its speed and ease of implementation, and the implementation details are in Section 4.

The QBIC [6] CBIR system uses both types of extraction; its automatic extraction is based on a foreground/background model, which is relevant to museum images as used in this project. This does however narrow the usefulness, and so semi-automatic extraction is also provided within QBIC, using flood filling and snakes.

2.2. Similar work

2.2.1. Magic Wand and Magnetic Lasso

Most popular graphics programs have some sort of tool similar to this project's goal. The flood fill or magic wand tool in Photoshop[1] will fill on colour, and allow the addition and subtraction of regions, but this can be tedious and time consuming. The

[1] http://www.adobe.com/products/photoshop/main.html

magnetic lasso tool – again, in Photoshop – looks to be incredibly useful, with the user outlining an object, with the tool snapping to the edge of an object. The downsides are that it requires a clearly defined edge, and patience and accuracy. This project will use timings with the lasso tool to compare object extraction.

2.2.2. Intelligent Scissors

'Intelligent Scissors' as described by Mortensen and Barrett [13] seem to implement exactly the same functionality as above, a 'live-wire boundary' that snaps to edges based on a graph search. Although testing is not reported in the paper, it can be assumed a similar amount of time and patience is needed. For complex images, many paths can exist and much time can be spent refining the result.

2.2.3. Object Extractor

The 'Object Extractor' tool [20] has a similar idea of allowing a flood fill to incorporate more than one colour, but this implementation has the user clicking on each new colour at a time, initiating a new flood fill. While this will eventually create the desired result, the approach described in this report allows a faster and hopefully more user friendly way of selecting multiple colours.

Colour median filtering [17] is implemented to pre-process the image. This technique, as with most filtering techniques, is designed to rid the image of noise. It works by looking at squared differences in RGB colour space between neighbouring pixels. However, while reducing noise, this also reduces detail, and can reduce edge information, and so is not suitable for this project.

2.2.4. GrabCut

GrabCut [16] proposes a foreground extraction tool based on iterated graph cuts – a process of compiling histograms of grey values for foreground and background, and working out segmentation by energy minimisation based upon inferring opacity variables. This appears to give sufficient results, but fails when edges are not clearly defined, whether by low contrast or camouflage.

2.2.5. Bi-lateral Filtering

Edge preserving smoothing/filtering has been looked at, most interestingly in the technique bi-lateral filtering [25]. This technique has the advantage of reducing the colour map, and preserving edges. The flood fill technique this project proposes should not need a reduced colour map to work with, the loss in detail would also be unacceptable, and although could be dealt with, the extra processing time would hinder the goal of being fast and efficient. If results are not agreeable, this will of course be looked into further.

The basis of techniques and ideas to be implemented in this project will now be looked at.

2.3. Flood fill

A generally recursive algorithm that finds connected regions based on some similarity metric, used mostly in filling areas with colour in graphics programs, though can also be found in implementations of Tetris. Variations that do exist seek to reduce the recursiveness, and make a more linear approach, mostly using linked lists [21]. Initially a compromise will be taken between programming complexity and performance for this project's implementation, and adapted if need be.

'4-way' or '8-way' flood fill algorithms are sometimes mentioned, referring to the number of branches from each pixel.

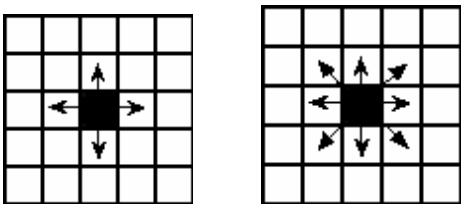


Figure 2.1: 4-way and 8-way flood fill

The 8-way method is hardly ever used as it is designed to leak on the diagonals, and so is useless for this project (could unintentionally fill surrounding objects).

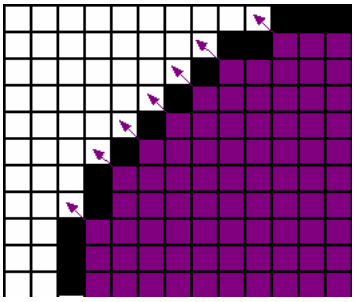


Figure 2.2: An example of 8-way flood fill leaking

The idea is so simple it can hardly be altered, but this project seeks to add innovation in the area of flood-filling colour by introducing extra parameters and combining approaches.

2.4. Active contour models

Introduced in the seminal paper by Kass et al. [11] , active contour models, or snakes, can be used to model the boundary of an object. The snake is an energy minimizing spline, user defined points will adjust their position and attempt to reach a local minima, based upon internal forces such as distance between points and curvature, and external forces such as gradient magnitude. Though this can require repetition or

manual editing, this project proposes to use this algorithm in combination with an improved flood fill to allow user specified object extraction.

This projects implementation is based upon the Greedy algorithm as suggested by Williams and Shah [26]. This is a relatively simple and fast implementation, in which each contour point is evolved in turn, with the initial contour point repeated. As noted by Nixon [14], the local minimum found is only an approximation, not the 'best' local minimum. A possible improvement would be to randomize the way in which the points are visited, instead of sequentially. **Appendix B** details the theory, and implementation details are in Section 4.

2.5. Colour

Colour is an important area for CBIR and image processing in general. Variant and invariant properties of colour are exploited in varying ways to extract information about an image.

The basis of electronic colour representation is the RGB colour space, in which an image "has three components per pixel, one each for the red, green, and blue intensities." [19]. Other colour spaces exist; HSV for example is a more natural way to think about colour, in terms of:

- Hue – the 'exact' colour, i.e. red, green, yellow.
- Saturation – how pure the colour is (at high saturation it will be an exact 'hue', with no saturation all colours will be a shade of grey).
- Intensity/value – how light or dark the colour is.

Fig. 2.3 from the Scientific Visualisation resource at North Carolina State University [15] details this relationship.

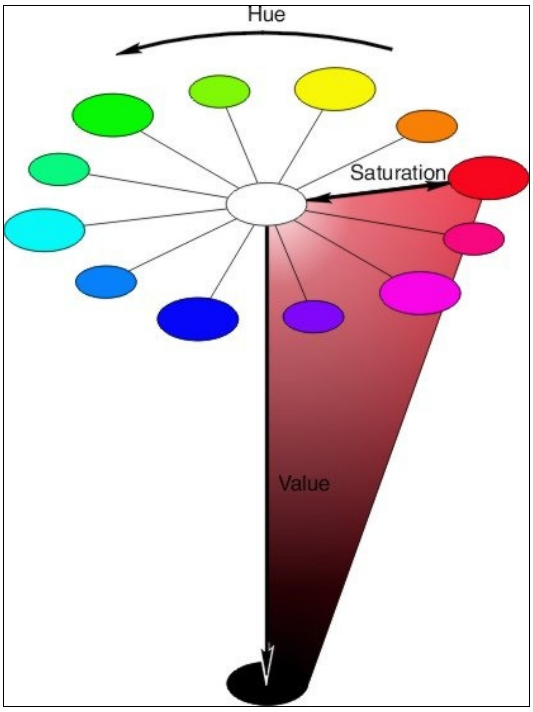


Figure 2.3: Diagram explaining relationship between hue, saturation and intensity.

Luminance is essentially the 'brightness' of an image. It is calculated from the weighted sum of RGB components, and is used in greyscaling the image for use in the Sobel algorithm. It is weighted because the sensitivity of the human eye varies with the colour (or wavelength) it detects (following from the CIE (Commission Internationale de l'Éclairage) photometric curve [1]).

Invariant properties are those which do not change considerably even when the brightness of the image changes, e.g. when a light source is moved. Invariants are used in shadow detection for example, based on the hypothesis that a cast shadow is the same colour as the background, just darker [5]. As described before, the HSV colour space is important here. Hue is an invariant. It describes exactly which colour is detected, dealing only with pure colours and not how light or dark that colour is. Shadows can then be detected by a hue of the same value to a reference pixel, but with less intensity, indicating going from a lit background, to a background in shadow.

As noted by Kender [12], to properly model hue a modulo operation is needed, and anything nearing 360° or at 0° could be considered red – as the above diagram displays. Similarly, normalised RGB is undefined near the black vertex of the colour space. As Salvador [18] concluded, and due to personal testing, initial testing for learning about colour information and edge detection will use the $c_1c_2c_3$ invariant colour model [8].

3. Design

The design of the Intelligent Flood Fill (IFF) tool can be split into 3 areas, each of which will be discussed in turn in this section:

- Architecture design based upon desire for extensibility and flexibility
- Algorithm and library design concentrating on technical aspects
- Application design focusing on the user interface and human-computer interaction issues

Requirements, in the form of use case diagrams, will first be examined.

3.1. Requirements

Designing an algorithm to floodfill an object within an image, and then extracting that object by means of saving the data in an appropriate format is the main aim of this project. At its simplest, the goals for the algorithm are to floodfill an image, given a number of parameters. This is represented in fig. 3.1. The requirements not in the diagram are:

- performance: it should be fast, with reference to current best method practices
- accuracy: again, compared to current practices.

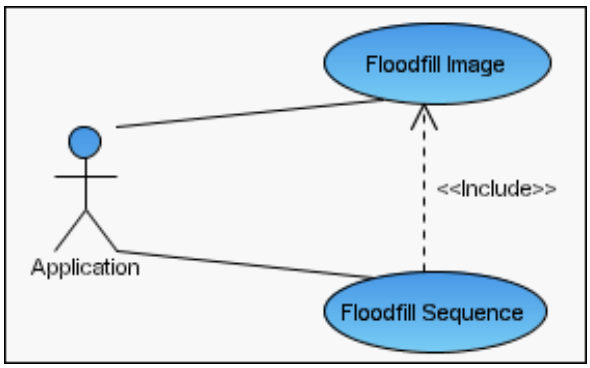


Figure 3.1: Algorithm use case

The algorithm will be incorporated into an appropriate architecture, and a user interface developed to allow users easy access to the features of the tool. The user operations are detailed in fig. 3.2. The non-functional requirements not listed are:

- a logical architecture
- an intuitive user interface
- the ability to use the tool on various platforms.

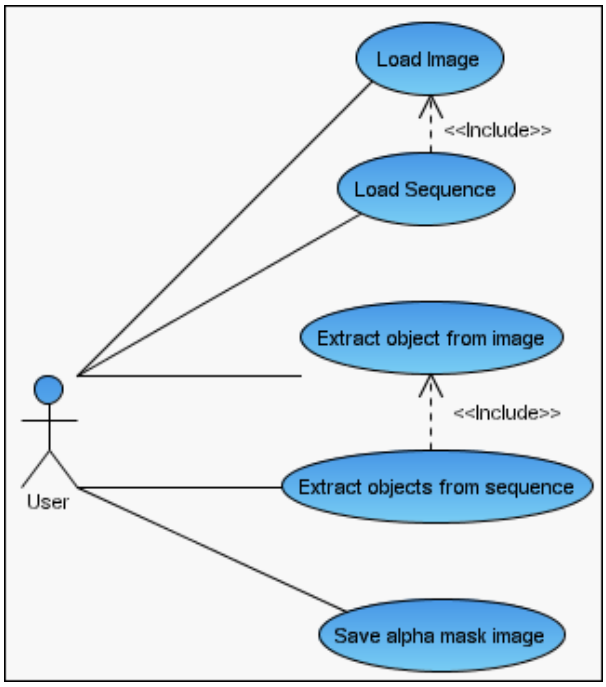


Figure 3.2: User operation use case

3.2. Justification for approach

The floodfill algorithm was chosen as the basis for extension, due to its ability to be used in both CBIR systems, and in a graphics package – since users are familiar with its operation already. Its current failing is in effort and lack of options, as discussed in the project description. Although tools like the Magic Wand, as discussed in Section 2.2.1, provide the ability to add multiple colours, they do so in a tedious and time consuming way. Similarly, more advanced methods such as the snake and lasso are too time consuming for simple images – although for more complex images this extra effort is needed, and so a snake will be implemented here.

There are many alternative techniques used in object extraction, some detailed in Section 2, that could have been used to build a new tool, but it was felt the floodfill approach offered the most opportunities for improvement; these new and original algorithms are discussed in Section 3.4.

An interactive approach was used rather than an automatic approach due to the problems in quality and failure rates in automatic extraction as discussed in Section 2.1. A minimal amount of user interaction is still desired, as set out in the project goals, to ensure an efficient use of time.

3.3. Architecture design

The architecture is designed to be extensible and flexible, and allow the changing of component parts without major alterations needed elsewhere. Initially, a Model-View-Controller architecture was considered, with each component to be separate, allowing a greater degree of modification to each with minimal impact to the others. However, in this application, due to the user interaction, there is a tight coupling between the controller and the view – it would be difficult to write a generic controller without taking into account the view, and vice versa. Therefore this project uses a modified MVC architecture, which Sun defines as "Separable Model Architecture" [7], as shown in fig. 3.3.

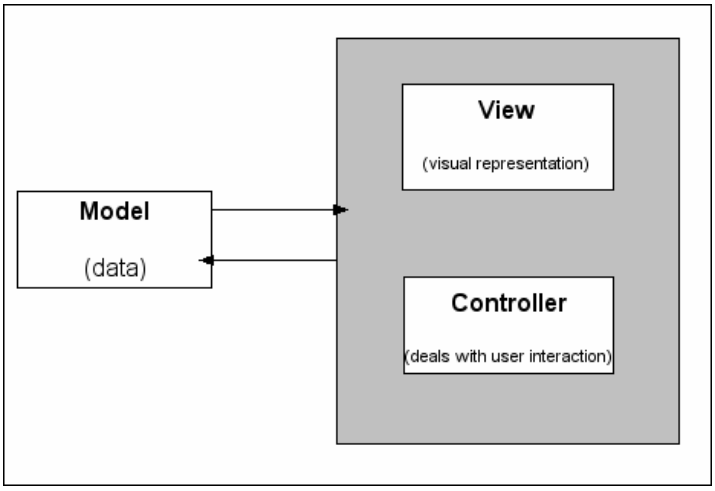


Figure 3.3: Separable Model Architecture

The project still wishes to keep the control logic as separate as possible, and fig. 3.4 displays a brief class diagram. The program will be written with a view to keeping the control logic as self contained and non-Java specific as possible, since this is the part that will be kept and rewritten if transporting to another language or application. In this case, we may see a slight deviation from fig. 3.3, where the control logic for applying Sobel, or a flood fill, will be kept separate. In this case, an array (an image) will be passed to it; the algorithm will make changes, and return an array.
Part of the controller and view are coupled in the 'Application', which will make calls to the algorithms, and handle references to the image that it will display.
The model - the image - is separate and will be wrapped in a display class, and put into a content frame within the view.

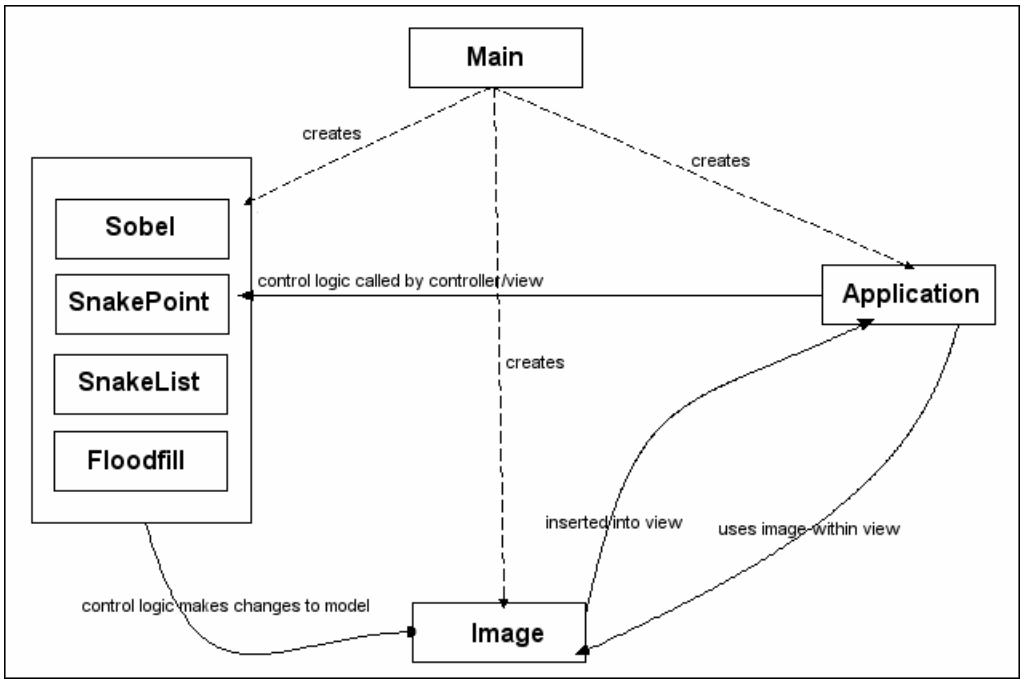


Figure 3.4: Overview of components

Although fig. 3.4 is an oversimplification, a conceptual class diagram can be built from this, with the most important methods and classes listed, see fig. 3.5. Each main class and their role in the overall program will be explained.

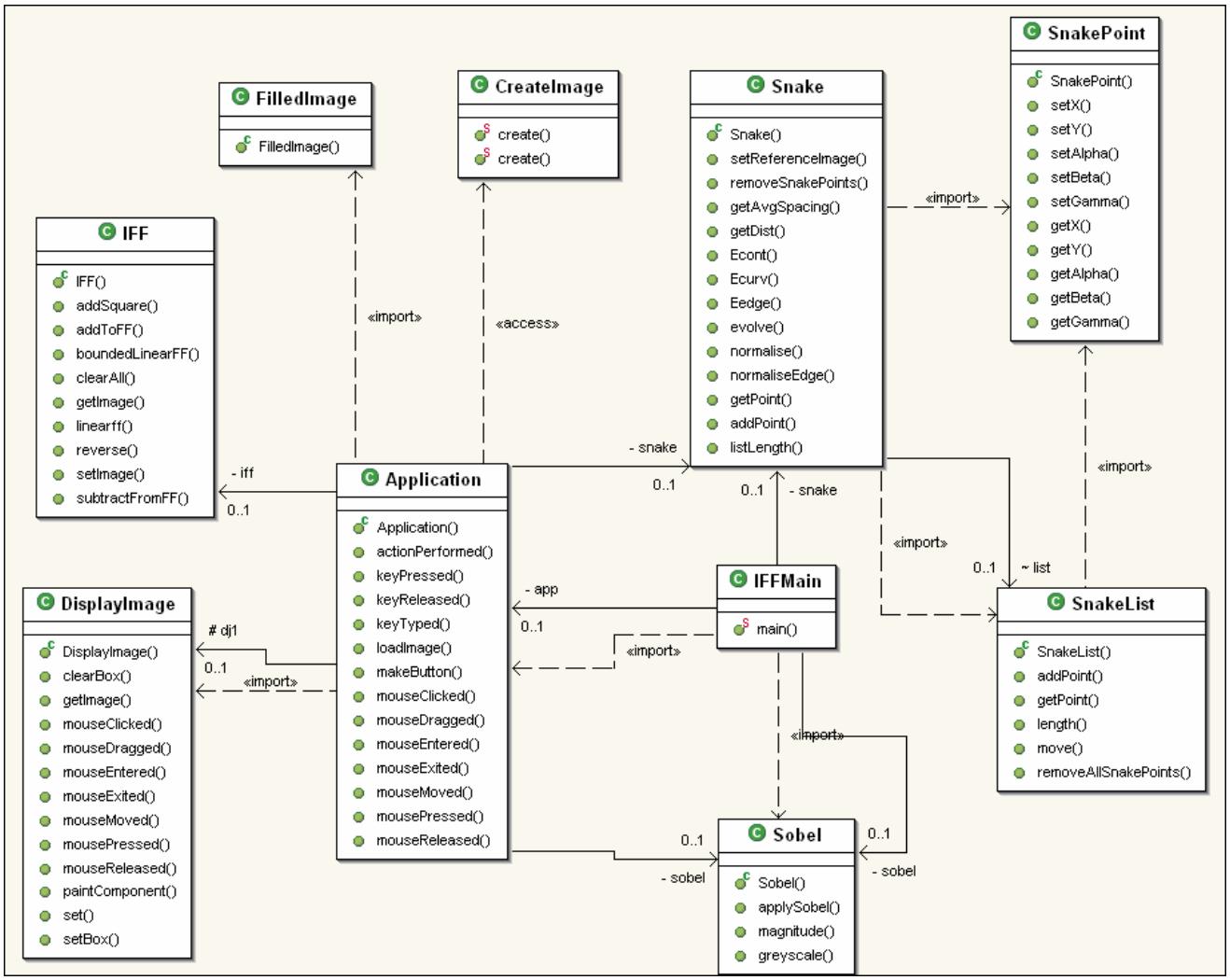


Figure 3.5: Brief class diagram

Application This class acts as the main part of the specialised 'controller & view' component. It will handle part of the interface, and the majority of the control logic of the tool – calling appropriate functions based on user events.

DisplayImage The 'view' component, holding the image and managing user events such as scribbling and box drawing.

FilledImage The model component, holding a path to an image, along with floodfill information in the form of a WritableRaster, or bit mask.

CreateImage An example of a library function, this and other objects will provide access to common functions, such as creating an image from disk, intelligent image resizing, saving images to disk.

Sobel One of the algorithm classes, this will take an image or a path to an image and provide the ability to return an edge detected image.

Snake The three Snake related classes model the Snake algorithm and the contour points. It will provide the ability to evolve the Snake and return co-ordinates of the contour points.

IFF The class that holds all the floodfill algorithms, this allows the specification of an image, bounding box and scribble co-ordinates, and the execution various floodfill algorithms described below. Returns a FilledImage object.

3.4. Algorithm and library design

Together the algorithm and library provide the functionality for achieving the tasks set out in the use case diagrams. Concentrating upon the floodfill algorithms and any other significant library functions, the following section describes their design.

Sobel (existing algorithm)
Initially, all images will have the Sobel operator applied to them in order for use with the Snake. The basis is that within an image an edge represents a change in contrast, or intensity. An operator that approximates first-order differentiation to detect a change in intensity by looking at adjacent pixels (or pixels within a region) can then be used. The image will first be greyscaled using the luminance equation as derived from the CIE photometric curve. The two Sobel kernels, giving the rate of change of luminance along each axis, will then be convolved with the image. The new pixel value will be in a range 0 – 1140 (as proved in **Appendix A**), and this will be normalised to provide a final value. Full theory and research can be found in **Appendix A**. This will be a stand-alone class, being passed an image, and returning an edge detected image.

Snake (Active Contour Model) (existing algorithm)
This algorithm will allow the user to specify a set of contour points around the object, and have the snake fit itself to the contours of the object. It does so by virtue of an energy minimisation process with three terms:

(i) internal energy controlling distance and curvature based upon average distance between points and change in external angle respectively
(ii) image energy based upon edge magnitude at a point given by the Sobel operator
(iii) constraint energy based upon external user interaction.

This design will ignore the constraint energy to simplify actions for the user. The snake equation is further governed by three weighting coefficients, these will initially be set to default values, and the implementation may allow them to be user specified eventually. This project models a SnakePoint object, and a SnakeList object, that keeps track of all the points. A Snake object then implements the algorithm; for each iteration checking a 5x5 area around each contour point, and if the energy is less than the current energy, moving to it. Full theory is provided in **Appendix B**.

Floodfill (existing algorithm)
This is the most important and innovative area of the program, offering some existing, and some new and original algorithms. A simple floodfill provides the basis for four further types of floodfill; each will be examined in turn.

Floodfill (i) Linear Floodfill (adaptation of existing algorithm)
The simplest version; this fills all pixels from a seed point that are within a specified tolerance (i.e. 20%) of the original RGB (colour) value. The 'filling' will be done by setting the alpha value of the pixel accordingly (making it transparent or opaque). Due to probable stack overflowing issues with a purely recursive algorithm, a more linear floodfill will be implemented as described here:

Given an initial seed point and the RGB value of the pixel at that point, check left and set that pixel to be 'filled' until a pixel outside the tolerance is reached, and then do the same moving right of the seed pixel.
Then for the pixels above and below the seed pixel, if within a threshold, recurse by calling the function again.

After applying a floodfill, the image will have an alpha mask associated with it, segmenting the image into one (or more) transparent regions (those that have been floodfilled) and a remaining opaque region(s) (that are not in the floodfill). Thus, when using the alpha channel as a mask in a graphics package, only the (floodfilled) object will be visible.

When floodfilling, the definition of connected pixels and tolerance is given by this (recursive) definition:

Let R denote entire image region, and 'connected' mean within a colour tolerance.
Pixel x_i in an image is connected to pixel x_j if and only if there is a sequence x_i … x_j, such that x_k is neighbouring and connected to x_i, x_{k+n} is neighbouring and connected to x_j, and x_k … x_{k+1} are a sequence of connected pixels, and all the points are in R.

The following floodfills all refer to this basic method.

Floodfill (ii) Scribble Floodfill (new algorithm based on floodfill)
This extends type (i) by allowing a multitude of colours to be specified, rather than just the colour at the seed pixel. The user will 'scribble' over the image drawing a sequence of circles, and colour information from each circle will be added to a histogram – in effect merely a vector. The original floodfill is then performed, but the threshold function, instead of checking if within a tolerance of one colour, will instead check to see if the next pixel is within a tolerance of any pixel in the colour histogram.

Floodfill (iii) Bounded floodfill (new algorithm based on floodfill)
Building upon either type (i) or (ii), the user can specify a bounding box around the object, telling the algorithm that beyond that box there is nothing of interest. If the floodfill does spill outside the box, eight points are attempted to be taken from outside the box, (borrowing the 'out codes' idea from polygon clipping), as shown in fig. 3.6.

Figure 3.6: Showing bounding box in solid red and eight areas which a seed pixel of a new floodfill will be taken from

Eight points are needed as opposed to just one since the background may be a gradient or be textured. The algorithm now has a floodfilled image, but with a floodfill that has gone outside the bounding box. The idea is to now scale back to the last time there was a major colour change. This can be achieved by using each of the eight points as a seed point for a new floodfill, and instead of floodfilling an alpha value, floodfill the original colour back into the image, until reaching a pixel that isn't within a threshold of the seed pixel.

Floodfill (iv) Snake floodfill (new algorithm based on floodfill)
Using the evolved Snake as a boundary for the floodfill, this will also implement the 'backtracking' as explained above, in case the Snake has not always found the edge.

Floodfill (v), (vi) Add and subtract regions (adaptation of existing floodfill)
These are simple additions. To 'add' a region to a floodfill, a simple linear floodfill is performed, only adding pixels within a tolerance of the initial colour. To 'subtract' a region, the same idea is used, but the pixel colour is set back to the original pixel colour.

Floodfill (vii) Reverse (new algorithm)

Although not a floodfill algorithm as such, this merely reverses all the pixels in the current image, from floodfilled to not floodfilled, or vice versa. This will be used in images where it is easier to floodfill the background than it is the object.

Sequence Floodfill (new algorithm based on floodfill)
If implemented, this extension will take bounding box and/or scribble data from one image, and apply that data to all images in a sequence, floodfilling each from the original seed pixel.

3.5. User Interface Design

The UI has to transparently aid the user to their end goal by allowing them to carry out their tasks productively. A list of common issues [24] from design and human computer-interface principles will be explained and related to this project, followed by a final design and the choices and reasoning for it.

3.5.1. HCI Issues

User Control
Rather than putting the computer in control, the users want and need to be able to initiate and control actions; in this context that means keeping the interface flexible, by providing multiple ways to access a function. Since this tool is being developed for a fairly limited number of functions, and the users will already have a mental model of the system from other graphics packages that implement similar functions, the project can use that as a starting basis. Similarly, universal functions such as saving a file should adhere to standards – the ability to save through menus, or using a shortcut key.

Metaphors
These allow the user to transfer skills from real world experience; however they are merely a cognitive bridge, and should not define or limit the implementation of the metaphor. For example, in this project the idea of a 'pen' or 'paintbrush' will be used to scribble onto the image, gaining the colour information.

Feedback
This project will use direct manipulation, and as such the user needs indication through immediate visual response that either the command has been carried out, is being carried out, or why it cannot be carried out. An example is a progress bar.

Consistency
As mentioned earlier, since other graphics packages implement similar functions to this project, consistency should be kept allowing the user to transfer skills between applications, and focus their attention on the task at hand.

Perceived Stability
Status and feedback will be provided to let the user know the application is working, e.g. by having a consistent set of icons, and a clear, finite set of objects, and actions that can be performed on those objects.

Aesthetics
Visual design is an important part of the interface, clarity and simplicity should be promoted. Techniques such as subtractive design – eliminating elements that don't contribute to visual communication; visual hierarchy – determine importance of users' tasks; and affordance – users should be able to easily determine the action and outcome that can be achieved by using an object, will be looked at.

Modality
The application should avoid locking the user into one operation and not allowing them to do anything else before that operation is completed, an acceptable case would be the 'Save As' command. Users should be able to do whatever they want at all times.

Interaction Methods
'WIMP' – Windows Icon Menu Pointer – combined with direct manipulation, provides a common interaction style, and can be used to provide help for the novice user, while providing shortcut keys for expert users, combining the best of WIMP and command language.

3.5.2. Design Choice
The task to be completed is extraction of an image through a number of user interactions followed by a floodfill. The user will already have a mental model of a similar system from numerous graphics packages, and for consistency this should be taken into account.

3.5.2.1. Prototype 1
There are two main areas to look at, how to present (a) the image to be manipulated, and (b) the manipulation tools. Building on users' prior knowledge, a single document interface (SDI) will be used, with the main window showing the image. Consistent menus organised by category will be used to load and save images, and to provide information, and a toolbar will be used for the manipulation functions, drawing from existing standards where possible. The icons in the toolbar will try to adhere to both the metaphors used and similar products.

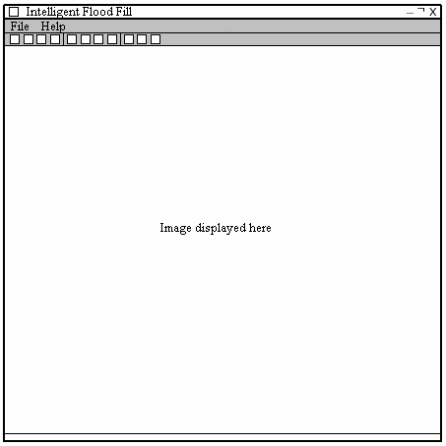


Figure 3.7: User Interface Prototype 1

3.5.2.2. Prototype 2

The SDI approach works well for single images, but with the task of looking at a directory of images, or the extension task of looking at a sequence of images, it falls down. Paint Shop Pro[1] provides a thumbnail approach in the main window, albeit with the ability to open multiple documents and resize each. Even with this ability, it can enforce divided attention, and this project will instead opt to use a 'slide-show' of thumbnailed images. Users may have previous experience of this in a program like Microsoft PowerPoint, but even if not, it should be immediately obvious what to do.

To allow a degree of forgiveness, an 'Undo all' button will be featured, allowing the user at any time to undo any scribble or floodfill information and start again. In prototyping it was noted that it would be an affordance to be able to click on the floodfill button and have the tool automatically perform the floodfill. However, an automatic floodfill would only be able to be performed half the time, due to the constraint of specifying a seed pixel. Because of this, it was decided to keep the floodfill seeding manual for all cases, rather than interfere with the user's model of the system.

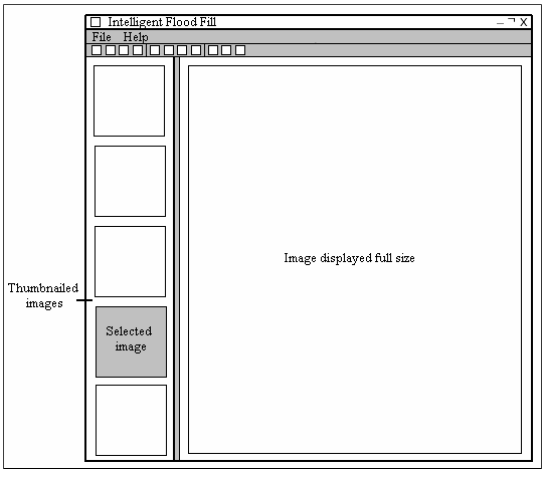


Figure 3.8: User Interface Prototype 2

[1] http://www.jasc.com/products/psp/

4. Implementation

Here the implementation of the tool as discussed in the Design section is looked at. The development language and assisting technologies are looked at, along with any deviations from the design. Important data structures and areas where the use and reuse of components have been most interestingly implemented are next examined. Diagrams are presented to explain how the system and the algorithms interact; followed by a brief look at the user interface, and the software life cycle method employed in the project.

4.1. Technological Decisions

4.1.1. Language and libraries

Java, due to its excellent support for graphical related projects and its platform independence, was chosen as the development language. For intensive tasks such as the algorithm and library functions it had been a possibility to use C++ and use the Java Native Interface to call them, but the Java performance was acceptable, bar one case explained in Section 4.2.

The Java Advanced Imaging (JAI) library was considered for loading, displaying and saving images, but it was found in comparisons that the JAI methods were no faster and allowed less flexibility than creating them from scratch. However, due to the complexities and patented encoding techniques (LZW compression) involved in saving TIFF image files, JAI had to be used for this purpose.

4.1.2. Integrated Development Environment

The Eclipse IDE was used for the majority of development, due to its ease of use and wide community support encompassing many plug-ins, some of which have been used as detailed below.

4.1.3. CASE tool

The Omondo EclipseUML plug-in was used for the generation of use cases and class diagrams, offering integration with Eclipse projects and excellent features.

4.1.4. Source code control

For both backup and concurrency, CVS was used to store the project, again integrated into Eclipse.

4.1.5. Automated testing

As detailed in Section 5, the JUnit testing tool again integrates with Eclipse, and will provide automated testing facilities throughout development.

4.2. Deviations from design

Originally a form of Bresenham's line drawing algorithm was used to draw the circles onto the image when the user 'scribbled', as this allowed exact edge and centre points to be used for gaining the colour information. This proved to be not fast enough however, causing lag when scribbling quickly, and so Java's native method was used, with the centre point plus eight points at varying radii and compass points added to the colour space.

It was found when loading high resolution images (~4000x2000) that Java would soon run out of memory. This was partially solved by optimising the code and increasing the heap size of the virtual machine, but still manifested itself. This is a large problem, and came at too late a time when the possibility of using an optimised image processing library such as VIPS to handle loading images could not be considered. To work round this problem, the images are resized to no bigger than 1024x1024, keeping the original height/width ratio, the work done on that image, and then when saved bi-linear interpolation is used to scale the image to the original size. Inevitably this leads to a loss in detail, but in most cases the final image is acceptable. However, the first recommendation for future work would be to use something like VIPS to solve this.

4.3. Important data structures

A number of data structures are implemented in the project, the following are a couple of the most important.

FilledImage
This is essentially the 'Model' component of the MVC architecture. When an image is floodfilled, rather than store the entire image which would mean degradation in performance and memory usage over time, the FilledImage stores a reference to the path where the image is located, and an array (a WritableRaster) of 1s and 0s – the alpha mask for the image.

Snake construct
The snake implementation involves three classes, implemented as follows:

> **Snake**
> This originally named class holds all the control logic, the algorithm, for the snake and a reference to the list of contour points.
>
> **SnakeList**
> Merely an encapsulation of a Vector, this class provides methods for holding and controlling the contour points of the snake.

SnakePoint
Modelling a contour point, this class provides get and set methods for the co-ordinates of the point, and the alpha, beta, and gamma coefficients controlling the weighting of the three terms in the minimisation function.

4.4. Use and reuse of libraries and components

There is a small overlap between this and the previous section, but this focuses on extensions to Java classes and programming for and with reuse.

DisplayImage
Rather than just using a Swing component or the less flexible JAI display, a DisplayImage object extends JPanel and is used to handle displaying an image, and also providing listeners allowing the ability to scribble or draw a bounding box.

Message passing system
A way of throwing events was needed when threading the various processor intensive tasks, and so a message passing system of broadcasters, listeners and events was created.

A 'Broadcaster' interface defines a broadcast method, and is extended by another task specific interface (e.g. DirectoryLoadBroadcaster extends Broadcaster), which defines the types of broadcast event. Any classes wishing to broadcast an event then implement that interface, (e.g. DirectoryLoader implements DirectoryLoadBroadcaster).

A 'BroadcastEvent' is also supplied, and extended by the task specific event, e.g. DirectoryLoadEvent extends BroadcastEvent.

A listener interface is then created, e.g. DirectoryLoadListener, and implemented by the class which wishes to listen for events relating to the directoy load function.

This small area is an excellent example of the object oriented nature of the architecture, and should allow further arbitrary extensions to be made if needed. A class diagram is provided in fig. 4.1 for illustration.

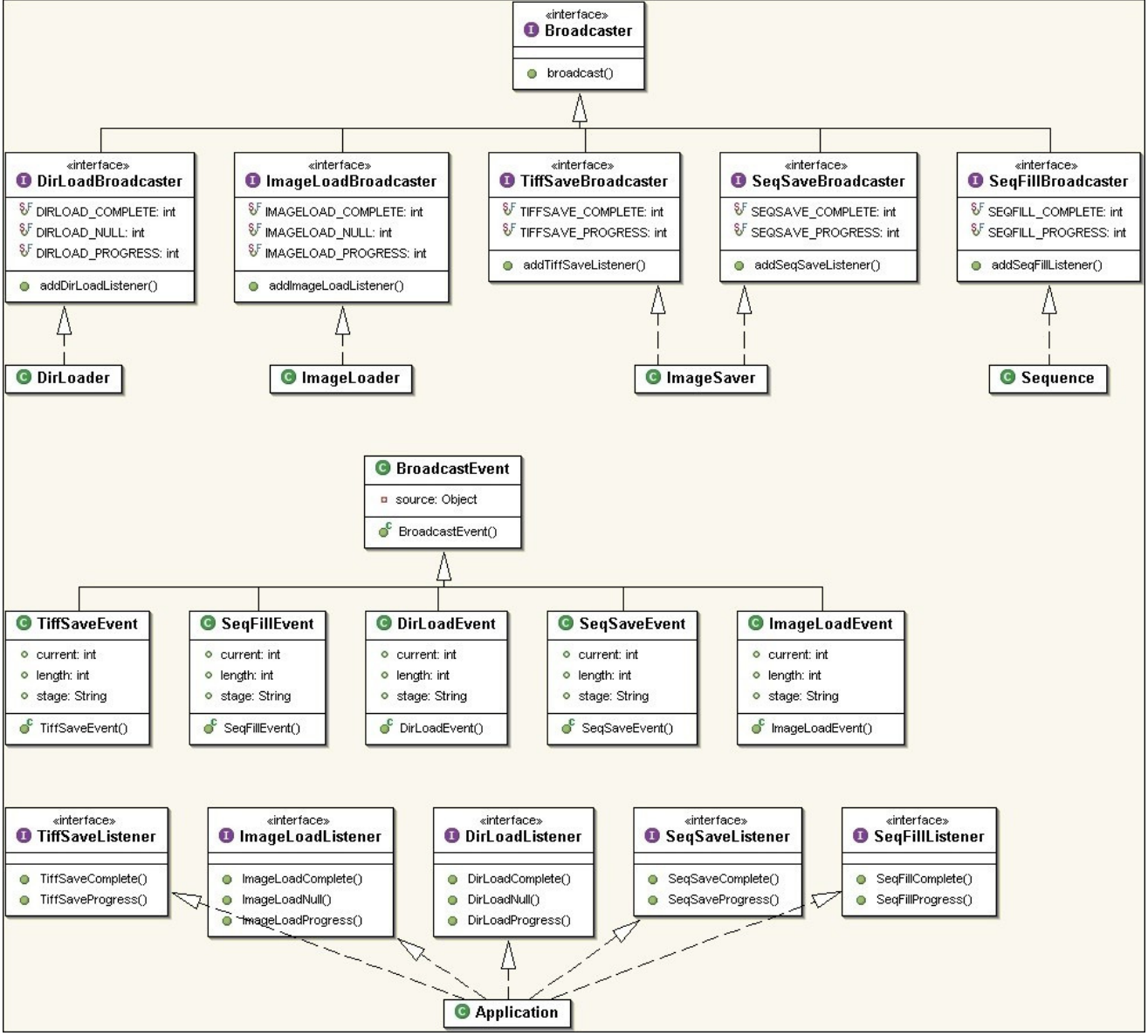


Figure 4.1: Class diagram showing inheritance in message passing system

Library functions

A class of static methods for commonly used functions was created, named 'Tools', allowing a central point for methods called again and again. These include intelligent resizing methods for resizing images to an arbitrary size but keeping the same height/width ratio, the creation of ImageIcons from paths, and resizing images back to the original resolution.

4.5. Algorithm and system implementation

The algorithms were all implemented as detailed in Section 3.4. Sequence diagrams, to serve both as an example of interaction between the components in the framework, and to illustrate the algorithm flow, are presented here for two scenarios. The flow of a scribble floodfill is first shown, and then an example of a sequence floodfill with both scribble and bounding box data.

4.5.1. Scribble floodfill

In fig. 4.2 a full sequence diagram of a scribble based floodfill is presented.

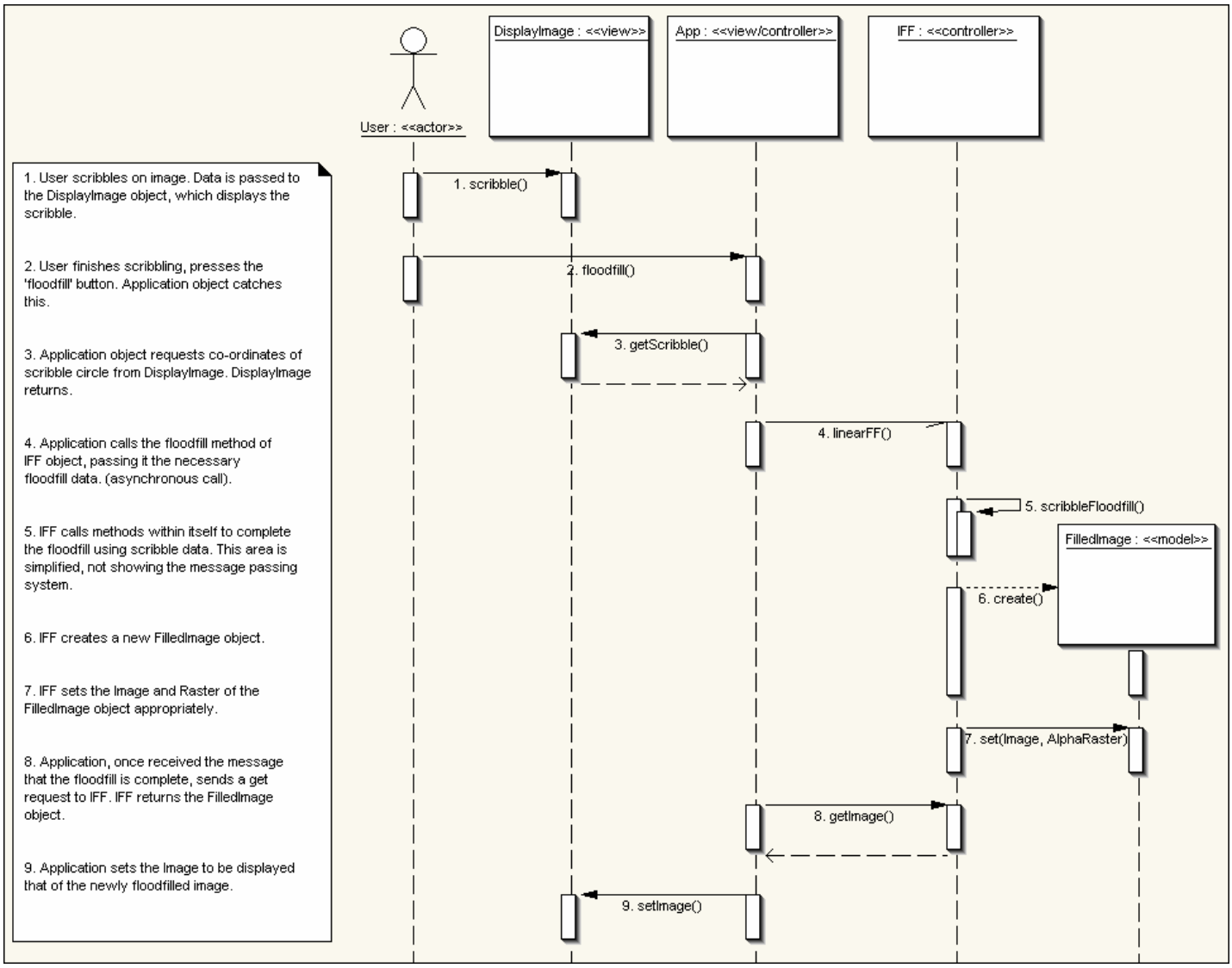


Figure 4.2: Sequence diagram showing scribble floodfill

Four objects are modelled, representing various components of the framework, along with the user, who initially interacts by setting the scribble data. The diagram is self explanatory, though here is the sequence of events in brief:

- User scribbles over object in image and presses floodfill
- Application gets scribble data and passes this to IFF object
- IFF object performs floodfill
- IFF creates a new FilledImage object, storing the floodfilled image
- When the 'complete' message is received by the Application, it gets the FilledImage object, and tells DisplayImage to display the newly floodfilled image.

The only area omitted for brevity is the message passing that would be involved, though this is explained in detail in the previous section.

4.5.2. Sequence floodfill

In fig. 4.3 an example of a sequence floodfill is shown.

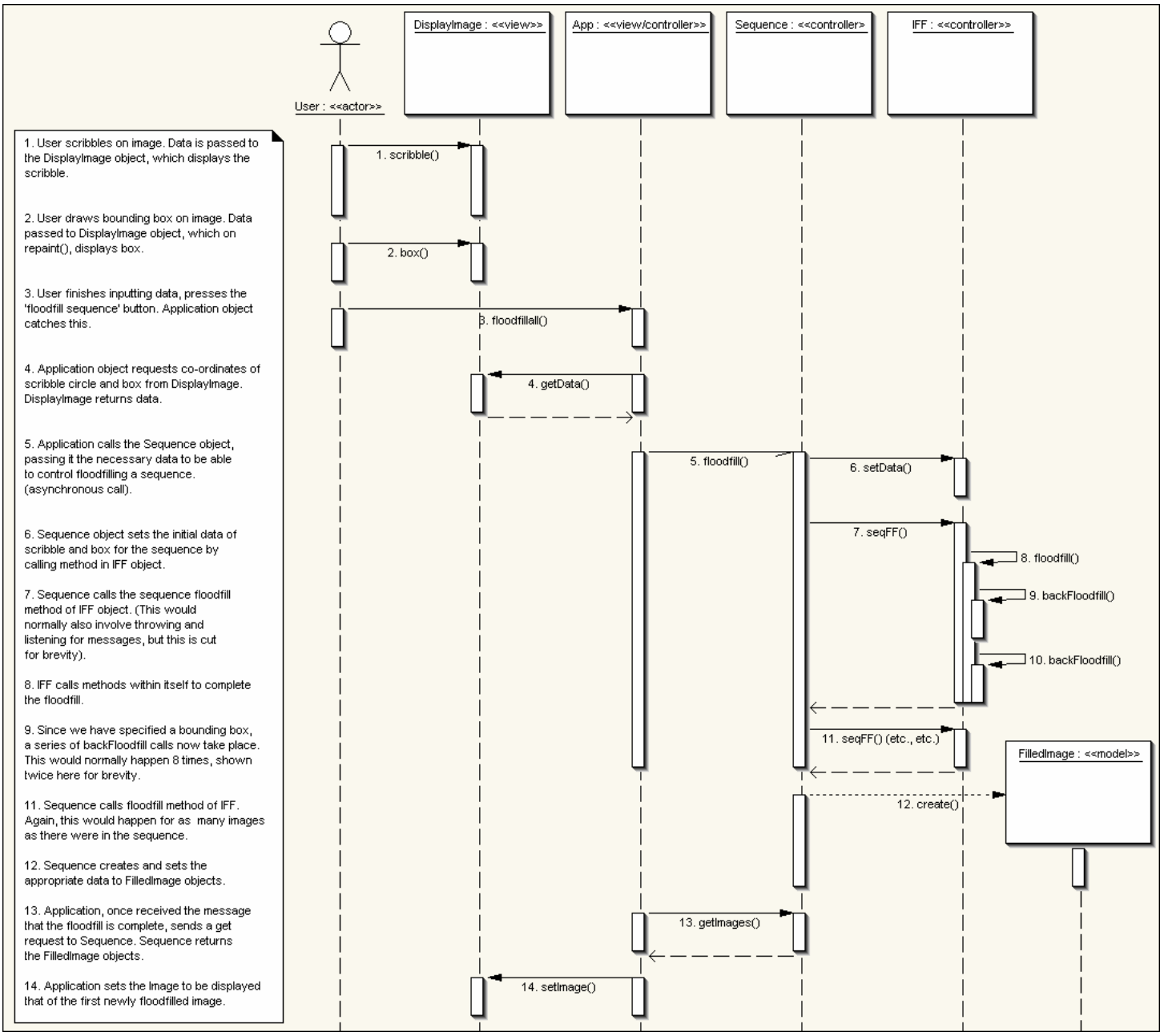


Figure 4.3: Sequence diagram showing sequence floodfill

Here five objects are modelled, the flow of data being roughly the same as in the previous diagram, though this time a Sequence object is introduced to handle the floodfilling of a sequence of images. Again, the message passing is omitted for brevity. The notes in the diagram are self explanatory, but this is the sequence of events:

- The user selects one image from the sequence, and draws scribble and box data on it
- The user selects 'fill all' from the GUI
- The 'Application' object requests the user defined data from the view, and sends this along with the floodfill request to the 'Sequence' object
- Sequence sets appropriate variables in the 'IFF' object, and tells 'IFF' to perform a floodfill on each image, one at a time. A 'FilledImage' object is returned.
- Once Application receives the message all floodfills are done, it requests the vector of FilledImage objects, and tells the view what to display.

4.6. User Interface Implementation

As described in Section 3.5, the user interface (UI) has been implemented after carefully examining HCI and usability issues. Since there has been no deviation from the design reasoning, and further details on how to use the UI are provided in the User Guide in Appendix D, it leaves only three screenshots to be presented, showing the two main views of the program, and an explanation of the toolbar.

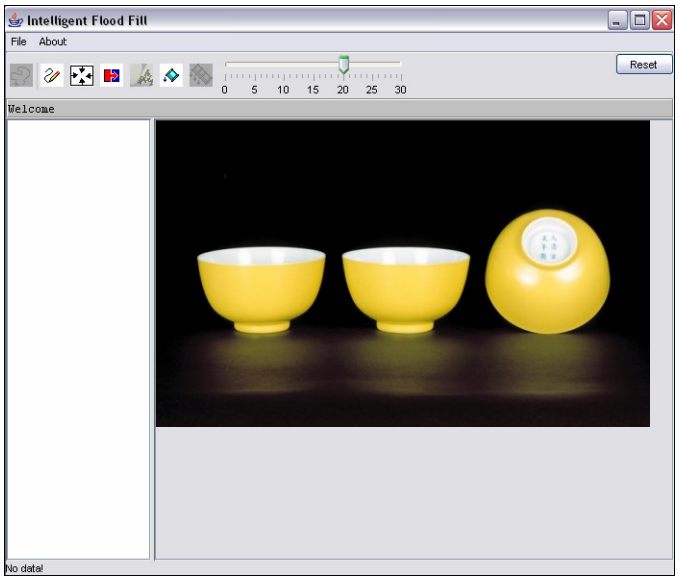


Figure 4.4: User interface, displaying one image

Fig. 4.4 shows the UI with a single image, waiting for user input. Fig. 4.5 presents a sequence being shown on the left hand side, with an image selected and being shown in the main window. Further screenshots are presented in later sections.

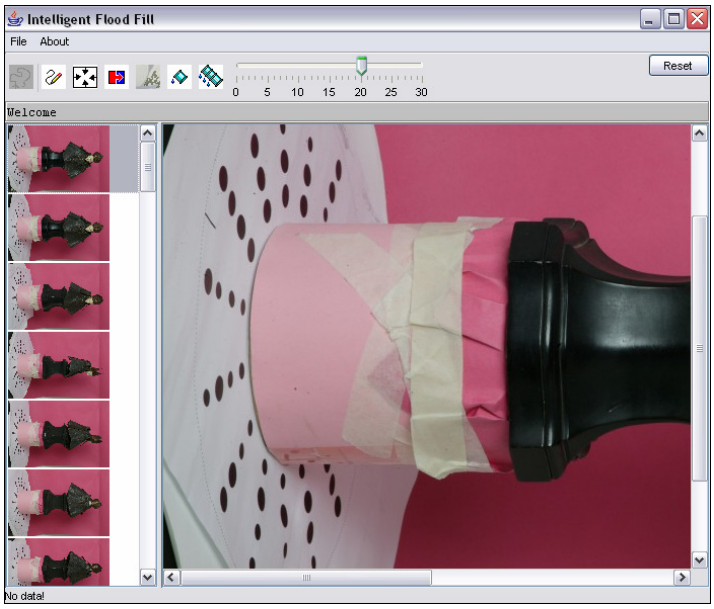


Figure 4.5: User interface in sequence floodfill mode

Fig. 4.6 displays the toolbar in more detail, showing the design recommendation of adhering to existing icon standards in graphics packages, a full explanation of operation can be found in the User Guide, Appendix D.

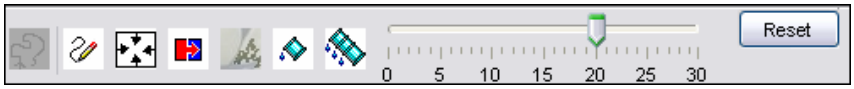


Figure 4.6: Toolbar in application

4.7. Life Cycle Model

This project followed a cross between an incremental and evolutionary software life-cycle model, which was both chosen and incidental. This is evident in the CVS of the tool, implementing one feature at a time, building up functionality. Requirements were initially a little vague and this allowed refinement of the requirements throughout development. A usable product was built up in stages, allowing early functionality, and integration testing throughout on smaller portions of the system which were included in later builds.

5. Testing

Testing will be in 3 phases, a unit test, an integration test, and a system test.

Much of the unit and integration testing has already been done whilst working on the code, but for completeness, and to ensure complete decision coverage, it will be formally documented. Due to the object-oriented nature of the design, the unit testing will cover each class at a time, ensuring each method performs as expected. Where a method has different paths, or multiple decisions, tests will be undertaken to ensure all decisions are covered. The testing tool JUnit[1] will be used within the Eclipse[2] IDE to automate stub and driver generation.

The system test will take place in a number of steps:-

- Capability
- Stability
- Resistance to failure
- Compatibility
- Performance
- Alpha/beta/acceptance testing

These will be explained in detail in 5.3. A test of the entire systems performance and accuracy compared to other tools is presented in Section 6. Some tests have a screenshot associated with them in **Appendix C**; this will be made clear in the text.

5.1. Unit Testing

Each appropriate class in the hierarchy has a test class associated with it – see fig. 5.1 - and this test class defines some 'set-up' steps, and a test method for each method in the original class. Only suitable methods were tested, otherwise the testing would be exhaustive and unproductive.

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

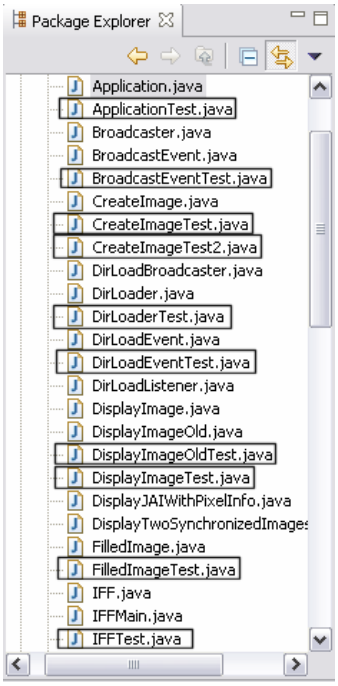
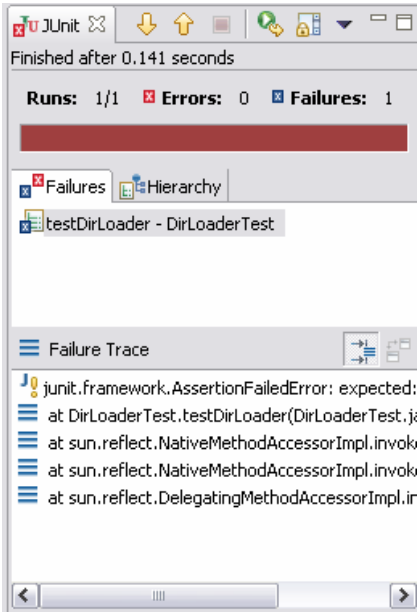


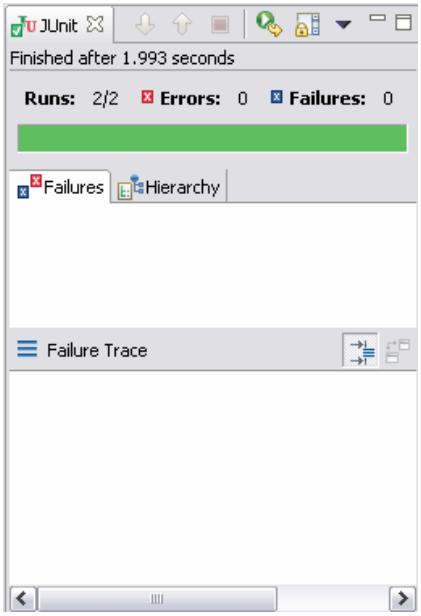
Figure 5.1

In some cases, asserting that the correct object or value is returned is not enough, in cases of image manipulation for instance, we need to know not only that an image was returned, but that the manipulation performed on that image is correct. This can sometimes only be done by checking against a perfect reference image, or by a visual check. A full example is provided below testing the class 'DirLoader', a summary of all other classes will be presented subsequently.

5.1.1. Full unit test example – DirLoader test

The 'DirLoader' class takes a File as input and outputs a Vector of paths to all images in that File (a directory), and a Vector of ImageIcons of those images. Two tests will be run, initially testing the correct number of files are recognised, secondly checking the correct paths and icons are returned. The first test fails – fig. 5.2 - closer inspection reveals that capitalised extensions were not looked for, i.e. ".jpg" files were recognised but not ".JPG". This was easily fixed, and both tests then pass – fig. 5.3.





Figures 5.2 & 5.3: JUnit fail and success

A summarised table of all 68 unit class tests (ref. UT1 – UT68) can be found in **Appendix C**, the tests that did not pass as expected are detailed here.

5.1.2. Test UT4, UT5

This test looked at whether the correct action was performed when clicking within the image. It is dependent on boolean values for its control logic, and it was noted that in certain cases the wrong course of action was taken – a simple re-arrangement of the conditions solved this.

5.1.3. Test UT22

Depending on which button is clicked on the toolbar, values for the icons and variables of the other buttons need to be set. This method is currently a little unstructured, and though works, could be done in a neater way.

5.1.4. Test UT41

The colour information from each circle clicked on is currently saved to aid in the flood filling. However, it was deemed necessary to have more than the colour at the central pixel saved, and so a number of surrounding pixel's colours are also saved. This is hard-coded in however, and so if the radius of the circle ever changed, erroneous colour values would be saved. This was altered so as to be a function of the radius of the circle.

5.1.5. Test UT63

Applying Sobel to a black and white image failed, and so the code was altered to take into account the fact an image may have only one channel – black and white – or three channels – colour.

5.2. Integration Testing

The tool has been tested throughout, and since the number of classes is prohibitive to a full enumeration of the tests, and since both Unit and System testing show an exhaustive list of tests, it was felt there was no need for a full integration test. The focus of the evaluation of the system is as a whole, and any errors here will appear in System testing, and so it is best to concentrate effort upon System testing. A number of errors that appeared during development and testing related to integration testing are described here.

5.2.1. Test IT1

It had only previously been deemed necessary to display a progress bar for intensive operations such as loading an image, or floodfilling a sequence. In combining all relevant classes however, and on a lower specification machine, the time to floodfill a single image occasionally seemed to warrant a progress bar. This task was thus threaded and a progress bar displayed. (See **Appendix C** – Test IT1).

5.2.2. Test IT2

A bug appeared in the code for sequence floodfilling. Upon floodfilling all images, and with the icons updated to display the alpha raster, clicking on an image should display the full-size image and its floodfilled alpha raster. This works for all images apart from the last in the list, regardless of the number of images in the list, which displays the previous selection's alpha raster. After extensive testing the source of this bug can still not be identified, though it will continue to be examined and an update will be appended. In the meantime, a quick fix has been implemented.

5.3. System Testing

System testing will look at a number of scenarios – a typical use of the system - ranging from simple and with correct data, to complex and incorrect or extreme data. Each section will start with a description of what it is trying to achieve, a full example of at least one scenario with outcome, and a table summarising the rest of the scenarios and the results.

5.3.1. Capability

This area tests to see if the basic functions work, examining a single 'transaction' within the system. No erroneous input will be attempted.

Scenario 1 – Load and flood fill an image (fig. 5.4)

No.	Input	Output
1	User selects 'Load Image'	Image displayed
2	User selects 'floodfill' and clicks on image	Floodfill executed, and result displayed

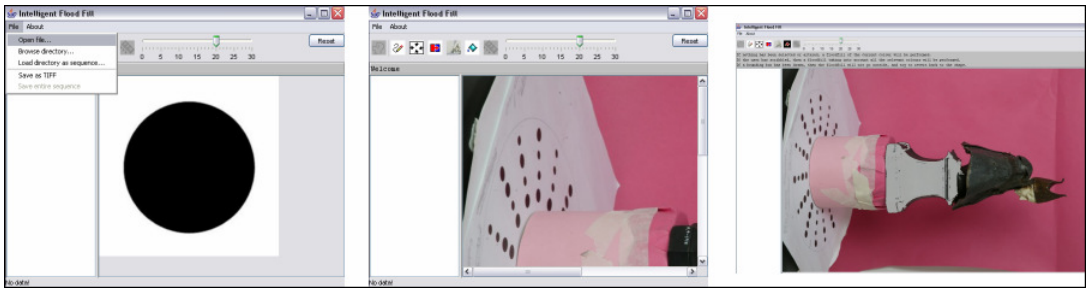


Figure 5.4: Loading and flood filling an image

The following table summarises the other scenarios, and indicates a success or failure, in the event of a failure, a detailed explanation will follow.

Ref.							Pass/Fail
CT1	Load Image	Floodfill					P
CT2		Scribble	Floodfill				P
CT3			Box	Floodfill			P
CT4					Reverse		P
CT5					Add direct		P
CT6					Add region		P
CT7					Subtract region		P (**Appendix C** - Test CT7)
CT8	Load Dir	Select Image	Floodfill				P
CT9			Scribble	Box	Floodfill		P (**Appendix C** - Test CT9)
CT10						Reverse	P
CT11						Add direct	P
CT12						Add region	P
CT13						Subtract region	P
CT14	Load Seq	Select Image	Floodfill				P
CT15			Scribble	Box	Floodfill		P
CT16						Reverse	P
CT17						Add direct	P
CT18						Add region	P
CT19						Subtract region	P
CT20					Floodfill all		P (**Appendix C** - Test CT20)
CT21						Manipulate	P

All simple capability tests passed, though it was noted the display of the scribble and bounding box was erratic when clicking through the toolbar, this will be fixed in time for the stability testing.

5.3.2. Stability

We look at modelling a realistic situation, with a sequence of transactions, but still with no invalid data. Since the scenarios are too long to be modelled in the previous format, a full test scenario is presented, and then the rest summarised.

No.	Input	Output
1	Load Image	Image displayed
2	Scribble, draw bounding box	Scribble and box displayed
3	Floodfill	Floodfill executed and fill displayed
4	Reset	Everything cleared, original image displayed
5	Scribble, floodfill	Floodfill executed and displayed
6	Region clicked to be added	Region added and displayed

The stability tests can be split into two sections, those that take one high level operation (load image, load directory) and then manipulate the image various times, or those that do both a number of times.

5.3.2.1. Stability I

Ref.	Description	Pass/Fail
ST1	Load image, scribble, box, fill	P
ST2	Load image, scribble, box, fill, reset, scribble, box, fill	P
ST3	Load image, scribble, box, fill, reset, scribble, fill	P
ST4	Load image, scribble, box, fill, add, reset, scribble, box, fill	P
ST5	Load image, fill, subtract, scribble, fill	P
ST6	Load image, scribble, fill, reverse, subtract	P
ST7	Load image, scribble, box, fill, reset, scribble, box, fill, subtract, add, reverse, reset, scribble, box, fill	P

These tests were then repeated for the operations loading a directory, and loading a sequence.

5.3.2.2. Stability II

Ref.	Description	Pass/Fail
ST22	Load image, scribble, box, fill, load image, scribble, box, fill	F, see 5.3.2.2.1
ST23	Load image, scribble, box, fill, reset, scribble, box, fill, load image, scribble, box, fill	P
ST24	Load image, scribble, box, fill, load image, scribble, fill	P
ST25	Load image, scribble, box, fill, add, load directory, select image, scribble, fill	P

ST26	Load image, fill, subtract, scribble, fill, load sequence, select image, scribble, box, fill all	P
ST27	Load image, scribble, fill, reverse, subtract, load directory, select image, scribble, fill	P
ST28	Load image, scribble, box, fill, reset, scribble, box, fill, subtract, add, reverse, reset, scribble, box, fill, load directory, select image, scribble, fill, load sequence, select image, scribble, box, fill all, select image, subtract	P

A number of these were repeated to start with the operation 'Load Directory' or 'Load Sequence'.

5.3.2.2.1. Test ST22

After floodfilling an image, loading another image still left the bounding box and scribble from a previous image, meaning this test failed. To solve this, upon loading another image, or a directory, the appropriate objects are cleared.

5.3.3. Resistance to failure

Here the tests focus on users inputting bad data, and trying to move through the program in the wrong order.

Ref.	Description	Pass/Fail
RT1	Load non-existent file through 'Open Image'	F, see 5.3.3.1
RT2	Load non-existent directory through 'Load Directory'	P
RT3	Load non-existent directory through 'Load Sequence'	P (**Appendix C** - Test RT3)
RT4	Scribble outside of image	P
RT5	Draw box outside of image	P
RT6	Add to floodfill outside of image	F, see 5.3.3.2

5.3.3.1. Test RT1

When attempting to load a non-existent file, while not crashing, the program merely did nothing. A more helpful response would be to display an error message explaining the problem. This was easily fixed.

5.3.3.2. Test RT6

Attempting to floodfill or add to a floodfill outside of an image causes errors; since there is already code to check for this when scribbling, it was an easy case of adapting it for this purpose.

5.3.4. Compatibility

Looking at compatibility with end users machines, this focuses on operating systems, hardware, Java versions, and the output being able to be read by graphics programs.

5.3.4.1. Hardware
Any machine capable of running a Java virtual machine should be sufficient, although when dealing with large images, a significant amount of RAM is an advantage. Tested on an AMD Duron 1.67Ghz, 512Mb RAM, an Intel Celeron 3.0Ghz, 256Mb RAM, and an Intel P4 with 1Gb RAM. Performance was sufficient in all.

5.3.4.2. Operating System
Part of the reason for using Java is its cross platform usability, and so the tool worked equally well under Microsoft Windows 2000/XP, and RedHat Linux.

5.3.4.3. Java
Java Runtime Environment versions 1.4.2_05, 1.4.2_06, 1.4.2_08, and 1.5.0_02 were all tested, and worked perfectly.

5.3.4.4. Output
This will be looked at in more detail in Acceptance testing, but for now, the alpha channel can be loaded as a mask or a layer in Paint Shop Pro and Photoshop, which should be suitable.

5.3.5. Performance
This section is split into 3 sections:-
- Capacity – maximum images/tasks
- Accuracy – always produce correct results
- Response time – is it fast enough and consistent

5.3.5.1. Capacity
Simple images and directories could be loaded easily; the hardest task likely to be undertaken is that of loading a sequence of images, all of a large size (~4000x2000). Although this could be done, the footprint of the tool rose considerably, up to around 150Mb. It was clear a reference to a BufferedImage or similar was being held in memory, not allowing the object to be garbage-collected. After looking through the code, this problem was solved and on recompiling the footprint stayed constant. This allows large directories to be loaded and worked on at once and is more than suitable for the final implementation.

5.3.5.2. Accuracy
In developing the algorithms for flood filling, care was taken to ensure the correct behaviour every time. The algorithm always produces what it intends to, and in testing seemed to be more efficient overall than other ways of extracting objects. Detailed images are provided in Section 6.

5.3.5.3. Response time
Java is known not to be the quickest language, and sadly this is the case here. However, the slow areas are not the algorithm itself which executes quickly, but

rather the image loading and manipulation, that could easily be converted to C/C++, and indeed would be if used for a graphics plugin.
The response time in general use of the program is more than adequate, and although it would be an advantage to have faster image loading, it is consistent, and it was never the main goal of the program.

5.3.6. Acceptance Testing

A user guide, available in Appendix D, was written for the program and, after an interest was expressed, the program sent to the V&A Museum and a colleague of my supervisors in France. Feedback was positive, and a number of comments arose, with hopefully more to follow in time.

The first of these was an issue with an old Java VM, which was fixed easily, and also means the program is now compatible with even older Runtime versions.

The second was the requested ability by the V&A to floodfill a number of sequence images at a time, which had already been considered as an extension, and this resulted in the 'Load Directory As Sequence' and the 'Fill All' abilities being developed.

The third was the issue of getting 'lost' when navigating around the user interface. A text area was created that described to the user the various facilities that were available depending on which button was selected, and what actions could be taken next. This was not an ideal solution, but with the limited time available it was the most appropriate. An extension to the work presented here would be some sort of guide and/or wizard through the system.

6. Critical Evaluation

This section will examine the effectiveness of the tool as an object extraction tool, and compare its results and performance against the original specification and competitive tools. Firstly however, a look at the snake implementation and the reasons for it being left out of the final tool are discussed.

6.1. Snake Implementation

During development and testing a major change arose. The snake implementation required a considerable amount of user interaction, extra time in processing, and results were the same or only marginally better than those using just the bounding box floodfill. See fig. 6.1, 6.2 for a comparison of extraction with and without the snake.

Figure 6.1: Showing the graduation from a bounding box and scribble, through the floodfill, and then with addition and subtraction of regions, and the final result.

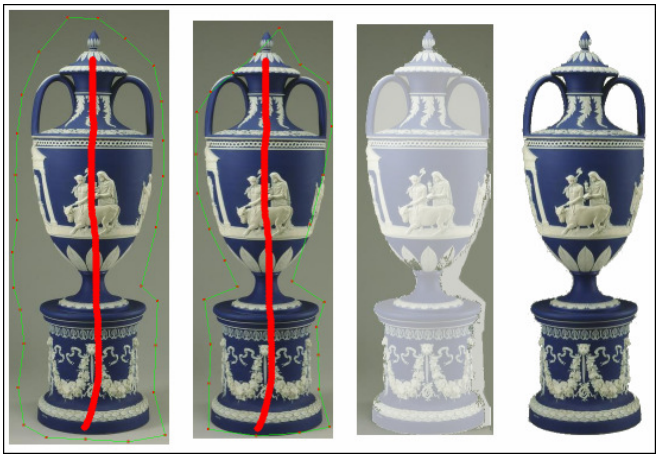


Figure 6.2: Showing the same image but with a snake and scribble information used. As we can see, the floodfill is slightly better, but with region subtraction and addition, the final result is almost exactly the same.

Since one of the aims of the tool was to keep user interaction to a minimum - to keep overall operating time down - the implementation of the snake was felt to offer no advantage, and has been removed. The problems were partly due to the implementation – given more time parameters could be more refined – but also due to inherent problems with the snake, with only approximate local minima being found. To improve this, a different implementation could be used, such as the 'complete' snake which finds the best local minimum, or a dual contour snake [10] that both expands and contracts, and also solves problems of initial contours being far away from target contours. A different approach could also be used, utilising a graph search of the entire image to look for strong features anywhere in the image, as in the Intelligent Scissors tool [13], rather than only looking at features near an edge's gradient energy. Even in these methods however, a lot of user interaction is still involved.

6.2. Algorithm results

The main aim for the project was that of a fast and efficient extraction, in terms of user involvement, operating time, and accuracy of end result. The images in fig. 6.3, 6.4, and 6.5 show increasingly complex images and the results.

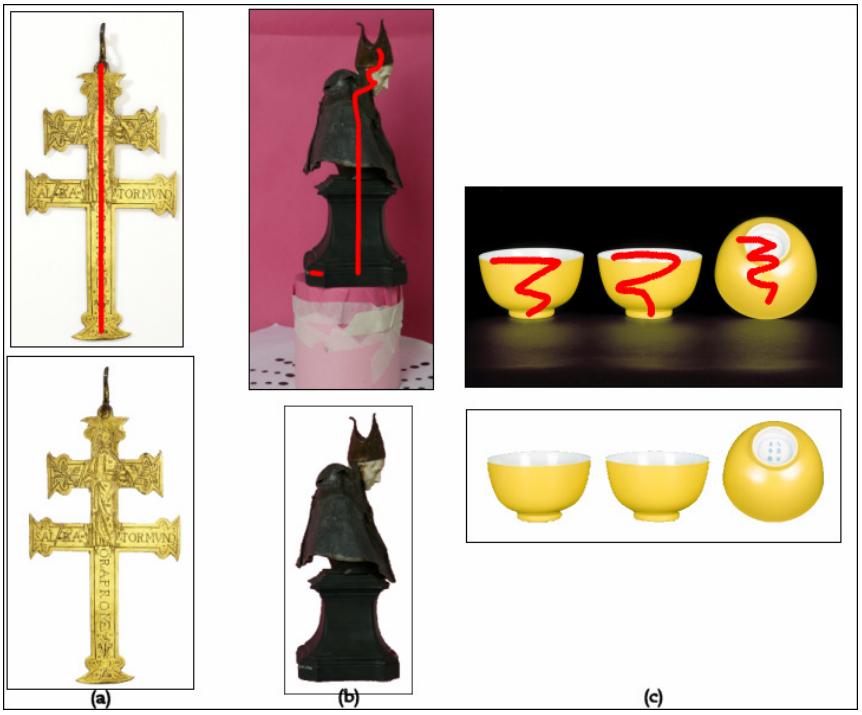


Figure 6.3: Scribble only floodfill and results

In fig. 6.3 the first row indicates the user input – the scribble – and the second row displays the results. For simple images it is clear the user interaction is minimal, and there are no failures. These types of images make up the majority of the single images (rather than sequence images) that the program would be used by the parties that have shown an interest.

Figure 6.4: Bounding box and scribble floodfill. The first row shows the user input, the second row the direct result of the floodfill, and the third row after user interaction.

Fig. 6.4 shows more complex images that have to be used with a bounding box and scribble information. Fig. 6.4(a) shows no interaction was needed after the floodfill, whereas fig. 6.4(b), 6.4(c) both needed some region subtraction and addition. Fig 6.4(c) is one of the most complex images the tool would ever have to handle, as on two occasions the foreground merges with the background. Although a reasonably sufficient image is finally extracted, it requires a considerable amount of user interaction.

Part of an extension to the work, and a feature requested by the V&A Museum, was to look at the problem of the large amount of work currently involved in floodfilling a sequence of images, such as a rotating object. A feature that took one set of user

interaction and applied it to all images was developed. Fig. 6.5 and 6.6 show the interaction and results.

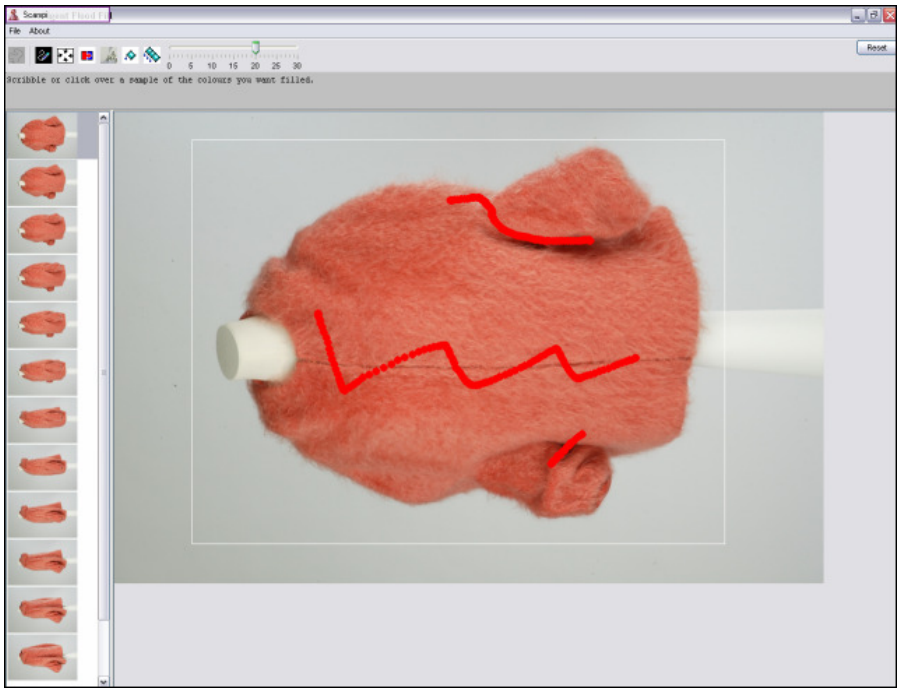


Figure 6.5: A sequence of 41 images is thumbnailed on the left, and the user interaction drawn upon one picture.

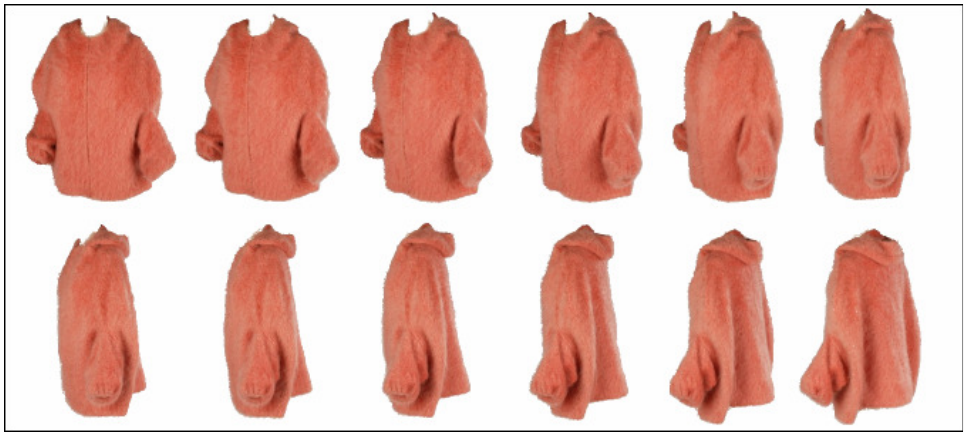


Figure 6.6: With no further interaction, these are the (first 12) results produced from the floodfill. Only a small amount of user interaction would then be needed to make the extraction almost perfect.

6.3. Quantitative testing alongside Magnetic Lasso

We have seen an example of the three main types of floodfill, and the type of user interaction needed and results gained. We will now look at a quantitative test to see how these compare in time and accuracy to a current best practice – the magnetic lasso tool (discussed in Section 2.2.1). The timing will assume the user is competent

with both tools, and will start when the user first interacts with the image, and end when the manipulation is complete and the image is ready to be extracted or saved.

Ref.	Description of task	IFF Time	Magnetic Lasso Time
CE1	Extract image in fig. 9(a)	17 seconds	55 seconds
CE2	Extract image in fig. 9(b)	12 seconds	43 seconds
CE3	Extract image in fig. 9(c)	26 seconds	22 seconds
CE4	Extract image in fig. 10(a)	91 seconds	88 seconds
CE5	Extract image in fig. 10(b)	39 seconds	27 seconds
CE6	Extract image in fig. 10(c)	114 seconds	55 seconds
CE7	Extract ¼ size sequence of images as in fig. 11 (41 images)	8 minutes 28 seconds (further manipulation needed)	8 minutes 02 seconds
CE8	Extract ¼ size sequence of images (image set based on fig. 10(b)) (55 images)	89 seconds (no further manipulation needed)	24 minutes 45 seconds
CE9	Extract full size sequence of images as in fig. 11 (41 images)	16 minutes 07 seconds	12 minutes 18 seconds
CE10	Extract full size sequence of images as in fig. 10(b) (55 images)	7minutes 45 seconds	24 minutes 55 seconds

Analysing the results points to a couple of major advantages and also a disadvantage. The first success is that of simple extraction. With minimal user interaction (2 brush strokes), an object can be perfectly extracted quickly, in the case of tests CE1 and CE2, over 3 times quicker than the Lasso tool.
The second major success is that of simple extraction combined with the sequence extraction. Although the Intelligent Flood Fill (IFF) tool is at an advantage because it has been specifically designed for this purpose, it nonetheless beats using the Lasso, in the case of test CE8, extracting the object 16 times quicker.
The main disadvantage is not with the floodfill itself, but with the image I/O – when manipulating large (~4000x2000) images, the intensive image loading slows the tool down considerably. This does not affect the Lasso tool, and so while its results for ¼ and full size images are roughly the same, the IFF tool times increase. However, for simple extraction it is still over 3 times faster. Two improvements to solve this are recommended in Section 7.

Overall, there is a distinct difference in the two tools' modus operandi. We can split images into two groups, "simple images" – those that have a clear foreground/background separation, and "complex images".
The Magnetic Lasso forces the user to do a considerable amount of interaction no matter what type of image. Its advantage is that it easily produces good results, but to get perfect results, requires exponentially more interaction and time.
The IFF tool had a goal to be "fast and efficient". Minimal user interaction is a part of this, and for simple images this is achieved. The results are near perfect as well, unlike when using the Lasso (unless considerable time is spent), in fig. 6.7 we see a comparison of an extraction of two simple images.

(Note: In fig. 6.7, the IFF extractions look 'blocky'. As done in the Magnetic Lasso tool, implementing a Gaussian smooth or similar would solve this. However, the smoothing introduces new colours into the image (it works by averaging the colours of neighbouring pixels), which, if the image were to be used for statistical analysis as in this project, could affect the result. A quick extension to the tool would be to offer the ability to smooth or not, depending on the user's preference and extraction purpose).

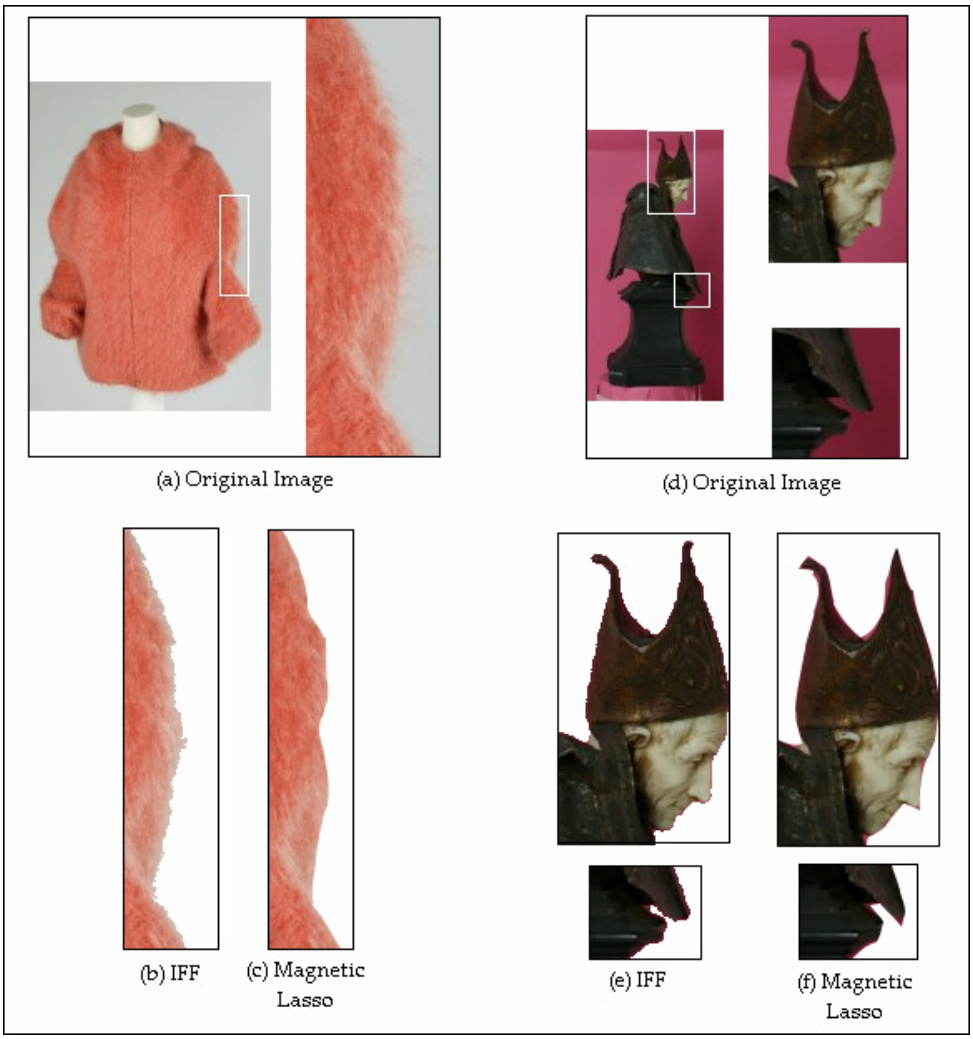


Figure 6.7 – Comparison of extracted objects

We can see that in some cases the Lasso loses detail - such as the top right of the bishop's hat or slicing through parts of the mohair coat – or keeps background detail. The IFF images do not have this failure. For simple images, the goal of being fast, efficient, and accurate has been achieved.

For complex images, although as competent a tool as the Magnetic Lasso, it offers no discernible advantage. The user interaction required is the same, and either takes the same amount of time or a little longer. In most cases, arbitrary accuracy can be achieved in both the Lasso and IFF, but the extra user interaction makes the time spent prohibitive. So, failures in terms of time and user interaction can occur when: (i) object and background overlap in colour space; (ii) the image is not high resolution and has some noise or loss of detail at edges.

6.4. Evaluation against remaining project goals

The main project goal has been concentrated on so far; in the following section the remaining goals are examined.

The second main goal was to allow modification to aid the user in selecting and extracting the object. A number of features were implemented to achieve this; the ability to add and subtract regions as common in graphics packages, and also the more original idea of being able to add an area directly to a floodfill. In the occasions of complex images these provide ample means to modify the selection. Improvements could be made in the form of changing the "brush" size and stroke when adding directly to the floodfill, and also implementing the ability to remove areas of the fill directly.

An extension to the work in the form of sequence extraction was a further goal, and this has been achieved, as described earlier. It is a vast improvement over any other tool, and could be improved further. The ability to take colours from any sequence image, as opposed to just one, would be useful. Using the image processing abilities of VIPS as suggested in Section 7 would improve the timing issues of large sequences.

Non-algorithm goals take up the remainder of the aims for the tool. Extensibility and flexibility has been well achieved, through the use of an appropriate architecture, an object-oriented (OO) approach, and flexibility in trying to keep the algorithm as non-Java specific as possible, so as to allow an easy translation to other languages. As with any OO program, the abstraction could be taken further and further, but I believe this tool represents a good middle ground between over-abstraction and inextensible, un-reusable procedural code.

Discussed in Section 3 is the rationale behind the user interface, and combined with feedback and alterations, the "intuitive user interface" goal is met in general.

However, the opinion that it is easy to "get lost" in the UI has only been partially fixed, and if a complete re-design were done, then although it would largely look the same, a wizard or more prominent help system would be an advantage.

6.5. Reflection

In general, the project was well planned and worked on steadily. The project goals and aims were sensible and achievable, although a little vague initially. Due to the vast research area that is computer vision, this meant that during the first couple of months a huge amount of hours were spent researching all manner of areas, such as shadow detection, which although interesting and informative, weren't strictly relevant. After these first few months however, work was steady and enjoyable, although intensive.

The choice of tools, techniques and methods suited the project, and the assisting technologies such as Eclipse and JUnit made development and testing efficient. The choice of language, Java, is a contentious issue, and as recommended in the project, although it is ideal for the majority of the tool, handling the loading and saving of images in a dedicated image library such as VIPS would remove the limiting factor of the tool in terms of speed. Utilising a logbook allowed a central record of all thoughts for the project, occasionally proving invaluable, and contains many ideas for future work that occurred to me during the course of development.

It turned out one major deviation was taken by removing the Snake implementation from the tool, as it offered no advantage given the project goals. The complexity of the algorithm and hence the implementation meant it was an ambitious undertaking, and possibly given more time it could have been refined to be suitable. However, as recommended in Section 7, the magnetic lasso tool performs better, and incorporating that to deal with the more complex images would be the next step.

Given the chance to do it again, rather than trying to solve everything at once, clear and concise goals from the start would have focused the research more. Using VIPS to handle the image loading and saving would solve the current tools main problems. Minor changes, such as GUI improvements, would also be undertaken.

I feel I have learnt much about developing a program. The importance of well researched requirements, a good initial design - basically good planning - make the implementation a lot easier, although the mistakes that are inevitably made by everyone along the way help to provide another viewpoint, or a rethink of certain areas. Developing a useful program, and in an area that interests you is as important as any software engineering issue. Finally, the need to realise when to stop researching before becoming engulfed in a never ending stream of papers is an essential lesson. Overall, the project has provided much interest and new knowledge.

7. Conclusions and future work

This project has presented the proposal and implementation of a tool for object extraction, based upon innovations and extensions to the floodfill technique.

Intelligent Flood Fill (IFF) provides an accurate and efficient semi-automatic tool, contributing new algorithms for object extraction: allowing the user to 'scribble' over the image to provide colour space data, and set a bounding box to attempt to revert to the last major colour change, used if the foreground/background distributions overlap in colour space. To address the issue of sequence extraction, a feature was developed that took one set of data and performed an extraction on each image, resulting in an average 400% (speed) improvement over current practices.

A GUI was developed to test the algorithm, most useful for the museums currently beta testing the tool. The architecture was developed to be flexible, and future work lies in translating part of the image loading to use an optimised graphics library, improving the limiting factor in the current implementation. Translation to a graphics package plugin would allow end users to benefit from the features presented.

Simple images were able to be extracted perfectly and quicker than current tools, but failures can occur in terms of considerable user interaction when presented with complex foreground/background colour space; extensions to the work include implementation of a live wire boundary, as suggested below, which although involving extra time, ensures accuracy for the more complex cases.

The following areas of future work and extensions are ordered in terms of effort:

- Interface improvements such as ability to zoom, change scribble size, etc.
- Adding further options to the floodfill:
    - Ability to specify different tolerance levels for different parts of the image
    - Allow foreground and background scribbles to better guess the segmentation of the image
- Use image processing library such as VIPS to handle loading and saving of images
- Convert tool into a plugin for PhotoShop or similar graphics package
- Integrate into CBIR system
- Consider automatic extraction for simple images, initially look at foreground/background models.

References

[1] Ashdown, I. 1994. Radiosity: A Programmer's Perspective. New York: John Wiley & Sons.

[2] Bach, J. R. et al., Virage Image Search Engine: An Open Framework for Image Management, *Proc. SPIE 2670, Storage and Retrieval for Still Image and Video Databases IV, 1996.* I. Sethi and R. Jain, eds., pp. 76 – 87.

[3] Canny, J. A Computational Approach to Edge Detection. *IEEE Trans. On PAMI,* 8(6), pp. 679-698. 1986.

[4] Chang, S.F., Smith, J., Wang, H. Automatic Feature Extraction and Indexing for Content-Based Visual Query. Technical Report CU/CTR 414-95-20, Columbia University, 1995.

[5] Cucchiara, R., Grana, C., Piccardi, M., Pratti, A. Detecting objects, shadows and ghosts in video streams by exploiting colour and motion information. *Proc. Of 11th Int. Conf. on Image Analysis and Processing (ICIAP), 2001.* pp. 360 – 365.

[6] Flickner, M., Sawheny, H., Niblack, J., et al. Query by image and video content: The QBIC system. *IEEE Computer,* 1995.

[7] Fowler, A. *A Swing Architecture Overview.* Online article. http://java.sun.com/products/jfc/tsc/articles/architecture/. Sun Microsystems. 2000. [Accessed 07 May 2005]

[8] Gevers, T., Smeulders, A.W.M. Color-based Object Recognition. *Pattern Recognition*. 32:453-464. 1999.

[9] Giorgini, F., Lewis, P., Martinez, K., Addis, M. Content and concept-based retrieval and navigation tools in Sculpteur. Available online at http://eprints.ecs.soton.ac.uk/archive/00008891/01/SCULPTEUR_EVA03_27May03.pdf [Accessed 05 May 2005]

[10] Gunn, S. R., Nixon M.S., A Robust Snake Implementation: a Dual Active Contour, *IEEE Trans. On PAMI*, 19(1), pp. 63 – 68, 1997.

[11] Kass, M., Witkin, A., Terzopoulos, D. Snake: Active Contour Models. *International Journal of Computer Vision*, vol. 1, no. 4, pp321-331, 1988.

[12] Kender, J. Saturation, Hue, and Normalized Colors: Calculation, Digitization Effects, and Use. Tech. Rep., Carnegie-Mellon University, 1976.

[13] Mortensen, E.N., Barrett, W.A. Intelligent Scissors for Image Composition, in *Proceedings of the ACM SIGGRAPH 95: 22nd International Conference on Computer Graphics and Interactive Techniques,* pp. 191-198, Los Angeles, CA, Aug. 1995.

[14] Nixon, M., Aguado, A. 2004. *Feature Extraction and Image Processing.* 2nd ed. Oxford: Newnes.

[15] North Carolina State University, Scientific Visualization Resource, online. Available at http://www2.ncsu.edu/scivis/lessons/colormodels/color_models2.html#hue. [Accessed 23 April 2005]

[16] Rother, C., Kolmogorov, V., Blake, A. Interactive Foreground Extraction using Iterated Graph Cuts. *ACM Transactions on Graphics (SIGGRAPH'04),* 2004.

[17] Russ, J. 1995. The Image Processing Handbook, 2nd ed. Tokyo, Japan: CRC Press.

[18] Salvador, E., Cavallaro, A., Ebrahimi, T. Cast shadow segmentation using invariant colour features. *Computer Vision and Image Understanding*, vol. 95, n.2, August 2004, pp. 238-259.

[19] Sangwine, S.J. Colour in image processing, *Electronics & Communication Engineering Journal*, pp. 211 – 219, 2000.

[20] Saykol, E., Gudukbay, U., Ulusoy, O. A semi-automatic object extraction tool for querying in multimedia databases. In S. Adali and S. Tripathi, editors, *7th Workshop on Multimedia Information Systems MIS'01, Capri, Italy*, pages 11–20, November 2001.

[21] Shaw, J.R. QuickFill: An efficient flood fill algorithm. Online article. http://www.codeproject.com/gdi/QuickFill.asp [Accessed 06 May 2005]

[22] Smith, J.R., Chang, S.F. VisualSEEk: a fully automated content-based image query system, *ACM Multimedia*, 1996.

[23] Sobel, I.E. *Camera Models and Machine Perception*, PhD Thesis, Stanford Univ. 1970.

[24] SunSoft. *Motif 1.2 Style Guide*. California, Sun Microsystems Inc. 1992.

[25] Tomasi, C., Manduchi, R. 1998. Bilateral filtering for gray and color images. In *Proc. IEEE Int. Conf. on Computer Vision*, 836–846.

[26] Williams, D.J., Shah, M. A Fast Algorithm for Active Contours and Curvature Estimation. *CVGIP: Image Understanding*, vol. 55, no. 1, January, pp. 14-26, 1992.

Glossary

Active Contour Model (Snake) A spline that uses three types of energy to move within an image to find object boundaries.

Alpha channel Used in images to store transparency information.

Canny More advanced operator for edge detection than Sobel, involving edge linking and thinning.

CIE Photometric curve States a light source will appear brighter if emitting certain wavelength light.

Content Based Image Retrieval Ability to search for images using the contents of the images themselves, rather than textual metadata.

Edge detection The process of applying an operator to an image to show only edge information, generally based on rate of change of intensity.

Feature extraction The use of algorithms to detect and isolate various desired portions of a digitized image or video stream.

HSV An alternative colour space to RGB, represents colour in terms of Hue, Saturation, and Intensity.

Magic Wand A tool in graphics packages that allows selection of an area based on its colour.

Magnetic Lasso A tool used to create selections, clinging to the edges in images.

RGB (Red Green Blue) colour space The basis of electronic colour representation, each pixel is represented by 3 values: a red, green, and blue intensity.

Sobel Operator for edge detection, based on two templates and convolution.

TIFF image Tagged Image File Format, used in this instance for its alpha channel support.

VIPS Image processing library providing, amongst many methods, the ability to load and handle large images.

VRML File format for representing 3-dimensional interactive vector graphics

Appendix A: Sobel Theory

Sobel theory

The Sobel operator utilises two templates, or kernels, which are convolved with the image. Each template gives the rate of change of luminance along each axis, and are as follows:

$\begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix}$	$\begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$

1(a) – M_x template 1(b) – M_y template

Convolution is a method of multiplying two arrays of numbers to produce a third array. In this instance, one array is the greyscale image; the other is one of the kernels. These are multiplied together to produce one value, and that value is the new pixel value of the central kernel pixel. See Figure 2.

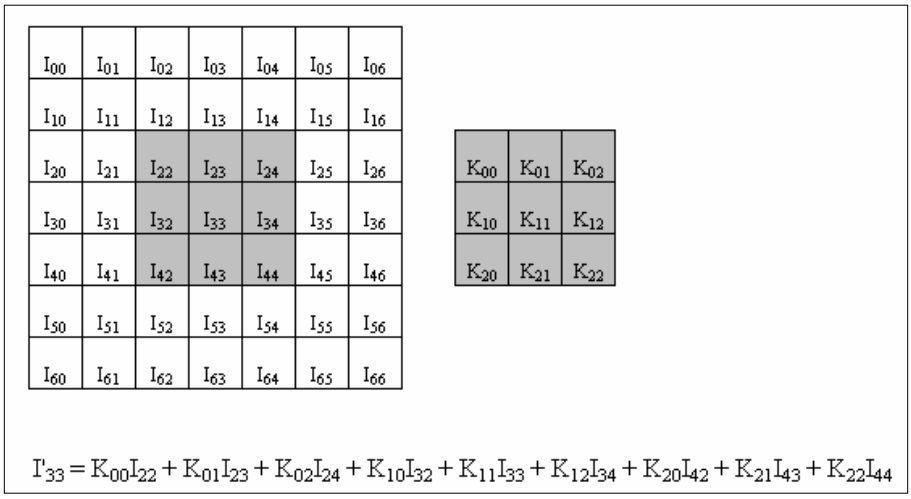


Figure 2

This project's implementation first greyscales the image, using the luminance equation as derived from [1]:

Luminance = 0.265*R + 0.670*G + 0.065*B

Once the convolved values for each pixel have been worked out (S_x and S_y), the magnitude is calculated:

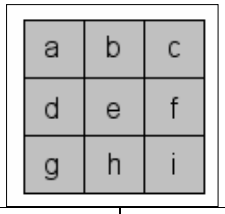
Magnitude = $\sqrt{(S_x)^2 + (S_y)^2}$

A pixel value has to be from 0 – 255. Since the magnitude ranges from 0 – 1140, we must normalise this value before applying it to the image. This maximum value only occurs when we encounter a part of the image that looks like:

Figure 3

Reasoning behind normalisation value

If we let the variables in the following matrix represent the RGB values of the pixels in the image:



$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 4

Then, with reference to the kernels, we are left with two equations:

$$S_x = a - c + 2d - 2f + g - i$$
$$S_y = a + 2b + c - g - 2h - i$$

We are trying to gain the maximum value these equations take, after the magnitude has been calculated from them. We set the shared positive variable '*a*' to 255, and the shared negative variable '*i*' to 0. Variable '*d*' only occurs in S_x and is positive, so we set that to 255, and similarly, variable '*b*' only occurs once and is positive, so set that to 255. Variables '*f*' and '*h*' only occur once and are negative, so set them to 0.
We are left with:

$$S_x = 255 - c + 2 * 255 + g$$
$$S_y = 255 + 2 * 255 + c - g$$

We are left with variables '*c*' and '*g*', of differing signs in each equation. However, they do not cancel each other out, due to the way the magnitude is worked out – by squaring the numbers. It is because of this that we set one of them to '0' and the other to '255', ending up with $S_x = 510$, $S_y = 1020$, and a maximum value of magnitude 1140.

Appendix B: Active Contour Model Theory

Active Contour Model theory

The snake is an energy minimisation process, with three terms: E_{int} – internal energy controlling distance and curvature, E_{image} – image properties i.e. edge magnitude, and E_{con} – constraint energy, usually external. The point is to minimise the following equation:

$$E_{snake} = \int_{s=0}^{1} E_{\text{int}}(\underline{v}(s)) + E_{image}(\underline{v}(s)) + E_{con}(\underline{v}(s))ds$$

where $\underline{v}(s)$ is the set of contour points of the snake. We ignore external constraint energy to simplify actions for the user. The snake energy is thus:

$$E_{snake} = E_{cont} + E_{curv} + E_{edge} = \alpha(s)\left|\frac{d\underline{v}(s)}{ds}\right| + \beta(s)\left|\frac{d^2\underline{v}(s)}{ds^2}\right|^2 + \gamma(s)E_{edge}$$

The first-order differential measures the continuity energy, or the spacing of points, of the curve. It can be calculated by taking the difference between the average Euclidean distance between points and the distance between the current point and the next. Controlling the value of alpha controls the spacing, but in general the minimum energy here is obtained when the current spacing is closest to the average.

$$E_{cont} = \alpha(s)\left|\frac{d\underline{v}(s)}{ds}\right| = \alpha(s)\left|avg.dist. - \left\|\underline{v}_s - \underline{v}_{s+1}\right\|\right|$$

The second-order differential measures the curvature energy, and is implemented as a change in external angle, or curvature between next and previous points, and current point.

$$E_{curv} = \beta(s)\left|\underline{v}_{s+1} - 2\underline{v}_s + \underline{v}_{s-1}\right|^2$$

Image energy is implemented as a magnitude of the gradient, or the Sobel operator at a point, and is controlled by the coefficient gamma. This is inverted so that areas with high edge strength are given a low value, following that the snake is a minimisation process.

Appendix C: Test Data and Results

Unit Testing

Here the table of tests as referenced in Unit Testing, Section 5.1 is presented.

Ref.	Class:Method Tested	Test Data	Expected Outcome	Actual Outcome
UT1	Application : makeButton	Description string and path to image	JButton returned with appropriate icon and tooltiptext	As expected
UT2	Application : loadImage	BufferedImages and height descriptors	Stub class appropriately called	As expected
UT3	Application : loadDirectory		Icons added to listModel	As expected
UT4	Application : mouseClicked	MouseEvent and boolean	Stub class called with correct params	See 5.1.2
UT5	Application : mouseClicked	MouseEvent and boolean	Stub class called with correct params	See 5.1.2
UT6	Application : mouseClicked	MouseEvent and boolean	Stub class called with correct params	As expected
UT7	Application : mousePressed	MouseEvent and boolean	Stub class called with correct params	As expected
UT8 - 21	Application : actionPerformed	ActionEvent	Stub class called with correct params	As expected
UT22 - 30	Application : setButtons	String describing button that fired event	Stub class called with correct params	See 5.1.3
UT31	Application : mouseDragged	MouseEvent	Logic performs correctly	As expected
UT32	Application : valueChanged	ListSelectionEvent describing new index and boolean	Stub class called appropriately	As expected
UT33	Application : valueChanged	ListSelectionEvent describing new index and boolean	Correct objects retrieved and stub class called	As expected
UT34	Application : stateChanged	ChangeEvent	Variable set to correct value from JSlider	As expected
UT35	CreateImage : create	String indicating path	Correct image returned	As expected
UT36	DirLoader : constructor	File indicating directory	Correct type/amount of files identified	As expected
UT37	DirLoader : loadDirectory		Correct paths and icons returned	As expected
UT38	DisplayImage : cnvrtRenderedImg	RenderedImage	Correct BufferedImage returned	As expected
UT39	DisplayImage : paintComponent	Graphics object	Insets * box * image drawn correctly	As expected

UT40	FilledImage : get/set methods	Various	Set or return appropriate object	As expected
UT41	IFF : addScribbleCircle	Co-ordinates of circle	Add appropriate points in circle to array	See 5.1.4
UT42	IFF : exists	RGB colour	Returns true if colour already set * false if not	As expected
UT43	IFF : setSequenceImage	BufferedImages	Either creates an alpha channel or sets appropriate variables	As expected
UT44	IFF : setImage	BufferedImages	Either creates an alpha channel or sets appropriate variables	As expected
UT45	IFF : linearff	Co-ordinates and threshold value	Flood fills correct area	As expected
UT46	IFF : inThreshold	Two RGB values	Boolean indicating if within threshold	As expected
UT47	IFF : reverse		Return reverse of flood fill	As expected
UT48	IFF : boundedLinearFF	Flood fill parameters	Flood fills correct area	As expected
UT49	IFF : inBoundingBox	Co-ordinates	Boolean indicating if within box	As expected
UT50	IFF : addToFF	Co-ordinates and threshold	Flood fills correct area	As expected
UT51	IFF : subtractFromFF	Co-ordinates and threshold	Flood fills correct area	As expected
UT52	ImageLoader : constructor	Path to image	Creates and resizes image appropriately	As expected
UT53	ImageLoader : constructor	Path to image and alpha raster	Creates and resizes image appropriately	As expected
UT54	ImageSaver : saveSequence		Saves correct files to correct location	As expected
UT55	ImageSaver : saveAsTiff		Saves to correct location	As expected
UT56	Sequence : floodfill	Appropriate parameters	Correctly flood fills all images	As expected
UT57	Snake : getAvgSpacing		Return correct spacing value	As expected
UT58	Snake : Ecurv		Return correct curvature value	As expected
UT59	Snake : Eedge		Return correct gradient value	As expected
UT60	Snake : evolve		Iterate snake correctly	As expected
UT61	SnakeList : move	Reference to snakepoint * and co-ordinates	Moves correct point to correct location	As expected

UT62	SnakePoint : get/set methods	Various	Return/set correct object	As expected
UT63	Sobel : applySobel	Image	Iterate over image correctly * calling correct stubs	See 5.1.5
UT64	Sobel : magnitude	RGB values of template of pixels	Return correct magnitude of central pixel according to Sobel	As expected
UT65	Sobel : greyscale	Image	Returns greyscale image	As expected
UT66	Tools : resizeImage	Image	Returns correctly sized image	As expected
UT67	Tools : resizeIcon	ImageIcon	Returns 100x100 image icon	As expected
UT68	Tools : bigImage	Image	Returns correctly sized image	As expected

Test IT1

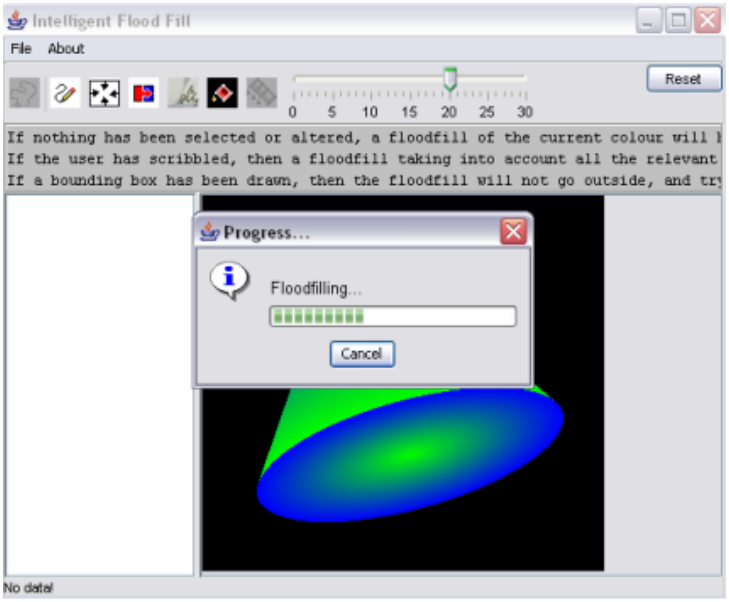


Figure IT1: Showing the floodfill progress bar

Test CT1

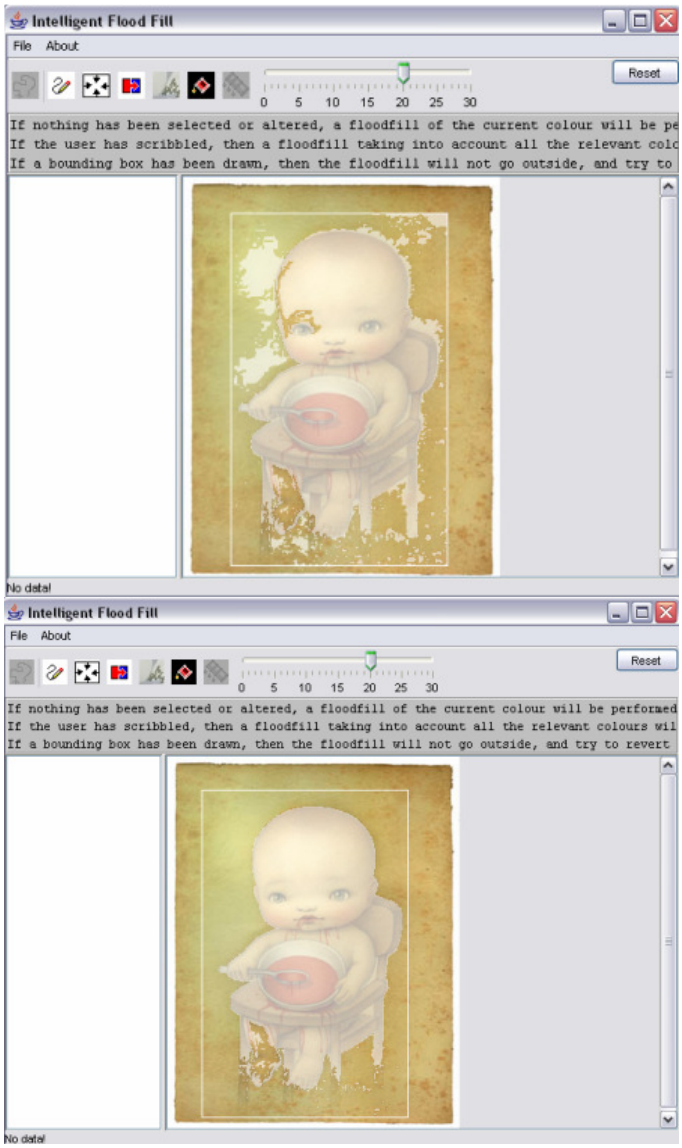


Figure CT1 – Before and after region subtraction

Test CT9

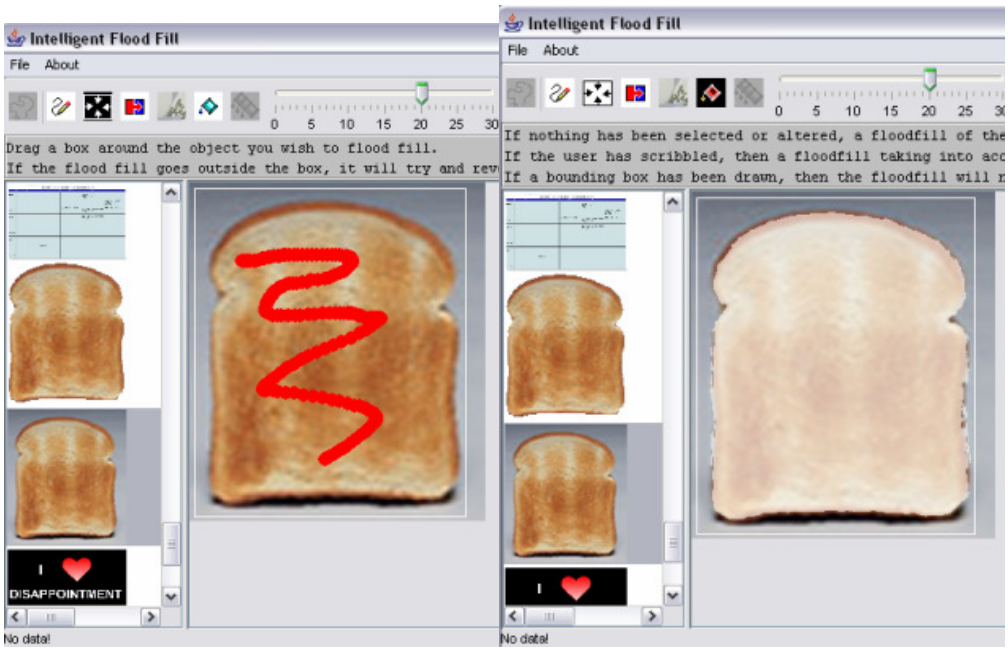


Figure CT9: Before and after floodfill from an image chosen from a directory

Test CT20

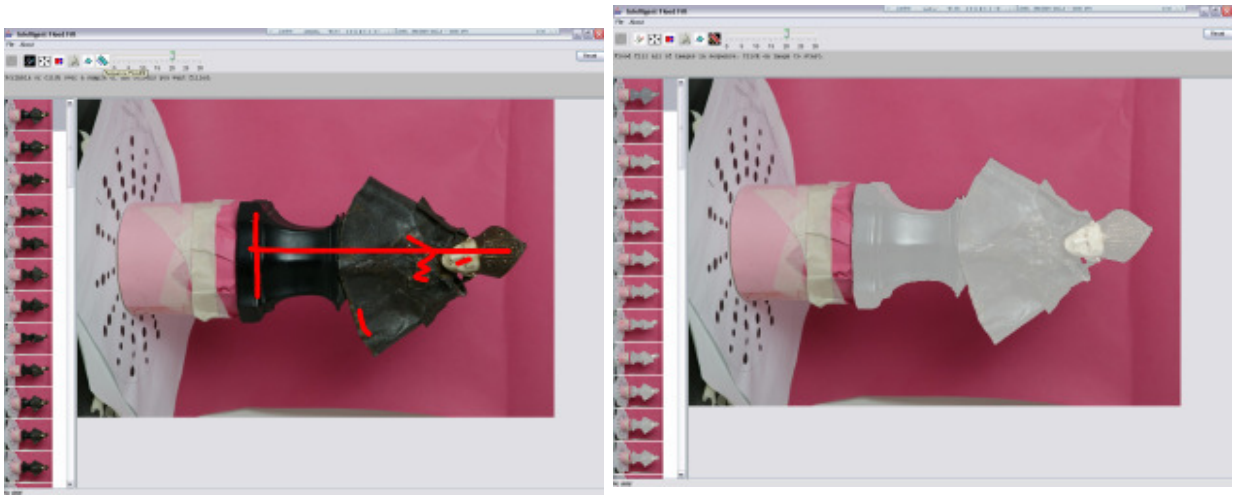


Figure CT20: Before and after sequence floodfill

Test RT3

Figure RT3: Attempting to open a non-existent directory

Appendix D: User Guide

Installation and Tutorial

Intelligent Flood Fill (IFF) is a tool designed for image object extraction, improving upon current flood fill and magic wand techniques.
This introduction is intended as an installation guide to IFF and the Java Advanced Imaging (JAI) library that is required to use it, and a look at the features of IFF and how to use them.

Please contact Paul Andre if there are any errors, omissions, or deliberate mistakes.

A (non-perfectly formatted) PDF of this webpage is available for download: http://www.ecs.soton.ac.uk/~pa302/IFF/IFFGuide.pdf (501KB).

Contents

Installation

Java Advanced Imaging

IFF is written in Java, and this guide assumes you have an appropriate Java Development Kit (JDK) or Runtime Environment (JRE) installed. If not, these are available from http://java.sun.com/. It has only been tested on 1.4.2, previous versions to 1.2 should work, and up to 5.

For the ability to load and save TIFF image files, the Java Advanced Imaging (JAI) library is used. This will need to be installed first. We will work on the basis that a JRE is installed. The download page for JAI can be found by following the "Download" button for the "JRE Install" here: http://java.sun.com/products/java-media/jai/downloads/download-1_1_2_01.html. Following the link through should bring up this page:

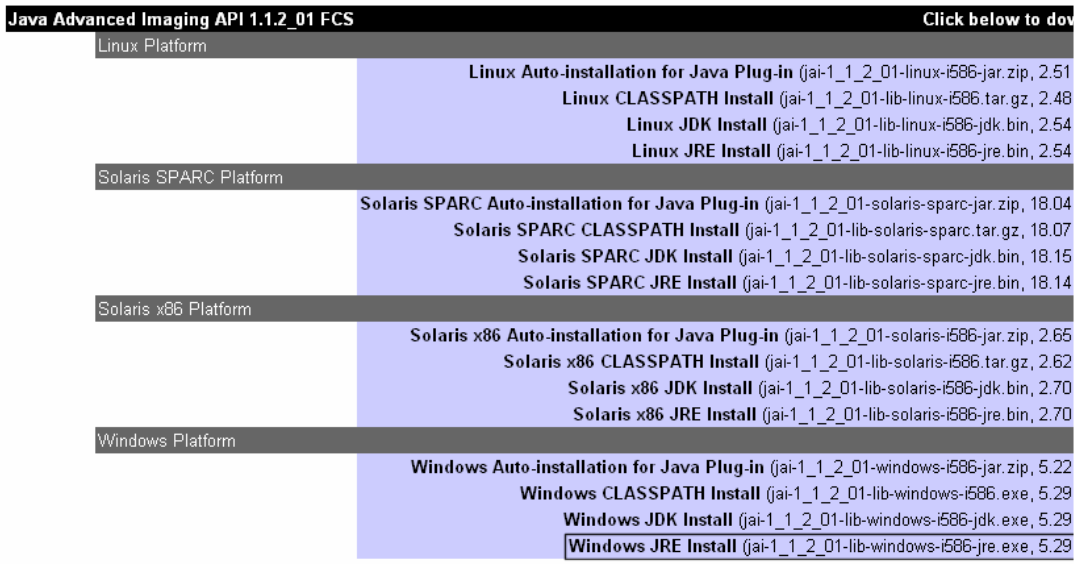


Figure 1 - JAI download page

Download the Windows JRE Install (5.29Mb), as circled in Figure 1. Running this installer should search for your installed JRE (or if using the JDK installer, an installed JDK), and automatically try and install to that directory:

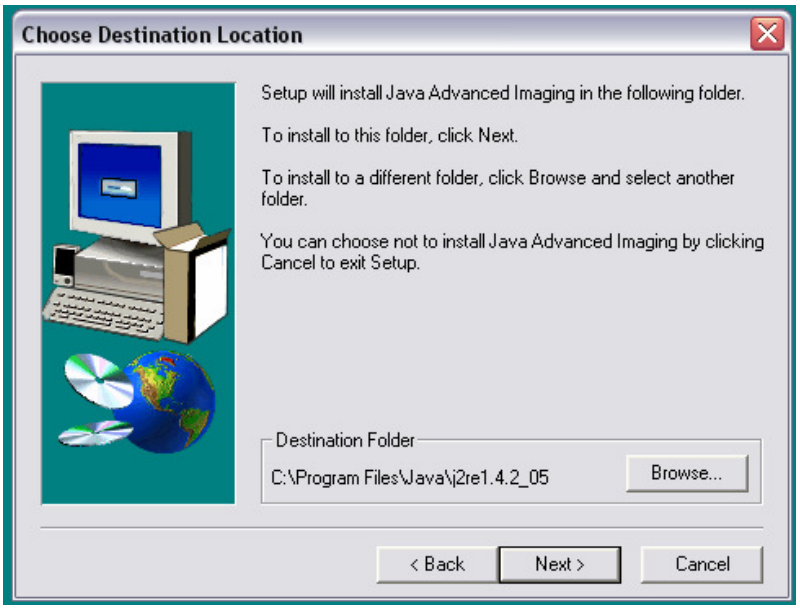


Figure 2 - JAI Install

This should install JAI into the correct directory, without the need for changing CLASSPATHs. The installation of JAI is now complete.

Intelligent Flood Fill

The tool can be downloaded from http://www.ecs.soton.ac.uk/~pa302/IFF/IFF06.exe (112KB).

Installing is a simple process, open IFF06.exe and either extract to the default folder (Figure 3), or specify your own.

Figure 3 - Extract IFF

3 files should now be resident in the specified directory, an archive, a batch file 'run.bat' and an images directory. Installation is now complete.

Running IFF

Starting IFF can be done in one of two ways.

Double-clicking on the file "run.bat" in C:\IFF06\ should launch the tool.

Or, from command line, navigate to the install directory, e.g.
cd c:\IFF06\
and then type
java -Xmx256m -jar IFF06.jar

If neither of these work, you may have to prefix "java" with the actual location of the file "java.exe". This may look like so:
C:\IFF06\> "C:\Program Files\Java\j2re1.4.2_05\bin\java.exe" -Xmx256m -jar IFF06.jar

User interface explanation

On first running IFF you should see a screen as in Figure 4. (Yes, including the circle, it's there as a test image :)).

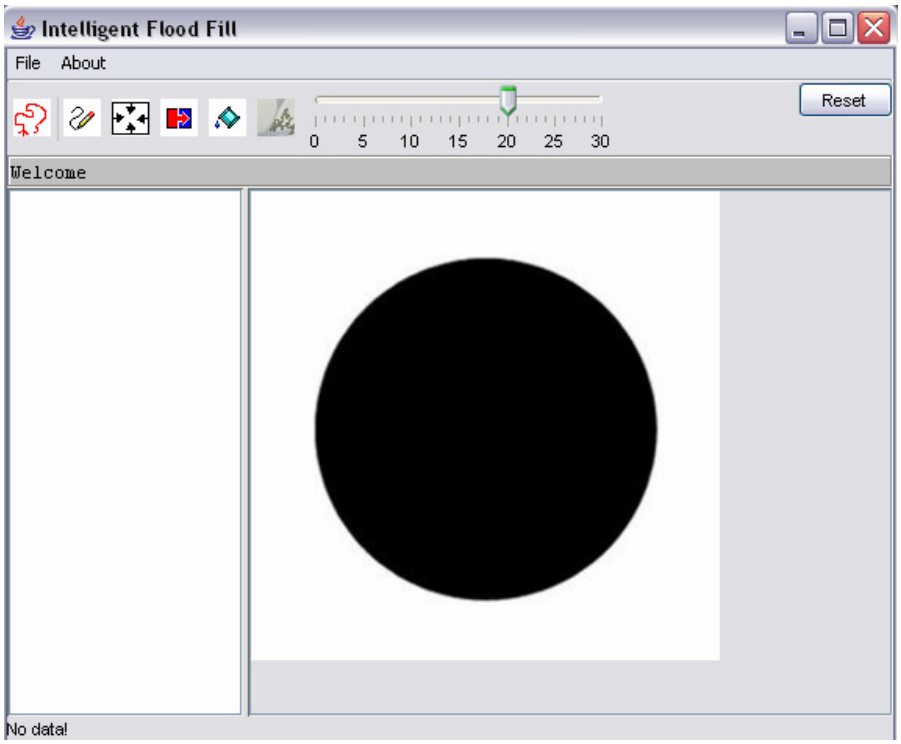


Figure 4 - User interface

The next section will deal with what each flood fill tool does; here we will look at how to move around the UI.

'File -> Open file...' will open an image file (currently support for jpeg, gif, png, bmp, tiff with no alpha channel), and display in the main window.
Note: because of edge detection operators applied when loading an image, a file of considerable size may take a few seconds to load. Also, if loading a file larger than 1024x1024, the image is resized when worked upon, and resized back to original size when saved.

'File -> Browse directory...' will display in the left hand window a list of all the images in the selected directory.
Note: this feature does not currently support all image types, only jpeg, gif, bmp. Also, attempting to load a directory with a lot of large images will take a considerable amount of time.

'File -> Save as TIFF' will save the image and any flood fill information associated with it. The flood fill information will be represented in the alpha channel of the image. This can then be loaded as a mask or a separate layer in an image manipulation program.
Note: User must specify file extension, either ".tif" or ".tiff".

Now we will look at the various buttons, and their functions. In the next section,

we will see how to use them in a typical scenario.

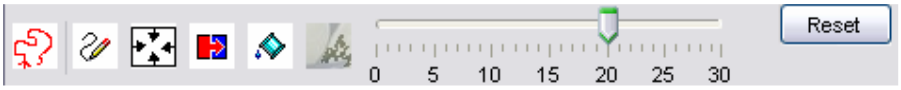


Figure 5 - Toolbar

This button is currently disabled. Although the feature is working, it is not suitable for testing at this time. (For those interested, it performs a 'snake' or active contour model over user specified contour points.)

This allows the user to 'scribble' over the image, as if scribbling over the image with a red pen. It is used to gather colour information for one of the types of flood fill.

This allows the user to draw a bounding box around the object in the image. Again, used in a flood fill situation to tell the flood fill not to go outside of a certain area.

This reverses the flood fill on the current image. I.e., all areas flood filled will be set to not flood filled, all areas not flood filled will be flood filled.

Performs one of five types of flood fill, depending on the context (described in next section).

Allows the user to add a 5x5 square of pixels directly to the flood fill, in case of a small area being difficult to add by flood filling.

This sets the threshold of the flood fill. Default is set to '20', i.e. if a pixel's colour is within 20 (out of 255) units of its neighbour, then we will choose to flood fill it. Setting it to '0' would mean only the exact pixel colour selected would be filled.

This resets any scribble, bounding box, and flood fill data on the image, back to its original state.

Flood fill guide

There are 5 types of flood fill that can be performed, here we describe them each in turn.

1. Simple flood fill

This is the most simple case. Just by clicking the 'fill' icon, and clicking anywhere on the image, a flood fill from that point is executed. Once a colour that is too far away from the original point is reached, the flood fill terminates. See Figure 6; the original click point is marked with an 'X'.

Figure 6 - Simple flood fill, before and after.

As you can see, only a small part has been flood filled. Although it may be possible to flood fill the entire image like this, point by point, it is time consuming.

2. Scribble flood fill

By using the 'scribble' tool, the user can scribble or click over the image, and in doing so acquires all the colour information under the scribble. A flood fill is then performed (again by clicking the 'fill' button, and this time clicking anywhere on the image), but instead of just the single colour information being used, all colour information is utilised. This way, the vast majority of images can be completely flood filled in a quick and efficient manner. See Figure 7 for an example of the original image, the scribble, and the flood filled image.

Figure 7 - Examples of working scribble flood fill

However, this does not work for all images. Any images that have colour in them similar to the background colour can cause the flood fill to 'spill out' into the background. An example of this is shown in Figure 8 - we see the entire image and background has been flood filled. In this case, the next flood fill can be used.

Figure 8 - Scribble flood fill gone too far

3. Bounded flood fill

Utilising a bounding box specified around the object by using the 'box' button, together with scribble information, allows for a third type of flood fill.

It operates by testing to see if it has gone outside the bounding box. If so, then it reverts back to the last time it encountered a major colour change (for example, at the edge of an image.)

This is guaranteed never to go outside the bounding box.

As we can see from Figure 9, for images where a scribble does not work, this is significantly better. However, there will generally still be some extraneous information. To solve this, flood fills 4 and 5 - adding and subtracting regions - come in.

Figure 9 - Bounding box, before and after

4 and 5. Adding and subtracting regions

As shown in Figure 9, it may sometimes be useful to add or subtract regions to/from a flood fill. The threshold slider is most useful in these cases too, reducing the threshold is often a good idea. Trial and error are best here.

If partial background information has been filled, we perform a 'reverse' flood fill, and subtract that region. This can be performed b selecting the 'fill' button, holding CTRL (control key), and clicking the area.

Similarly, if part of the object has not been filled, or if the subtraction of the region has eaten into the object, then by selecting the 'fill' button, holding SHIFT (shift key), and clicking the area, we can add to the flood fill.

After a few subtractions and using the ability to add a small 5x5 square directly to the flood fill as described before, we can end up with Figure 10.

Figure 10 - Adding and subtracting regions

Sequence flood fill

This feature can be used for floodfilling a sequence of images in one go, those used in creating VRML models for example. So instead of individually floodfilling one image at a time, out of possible hundreds, all can be floodfilled with just a couple of clicks. A quick scenario will be run through.

1. Choose 'Load Directory As Sequence' from the 'File' menu (Figure 11). Point to a directory containing only the images in the sequence you wish to manipulate. If loading a large directory, this could take sometime. This should result in something like Figure 12.

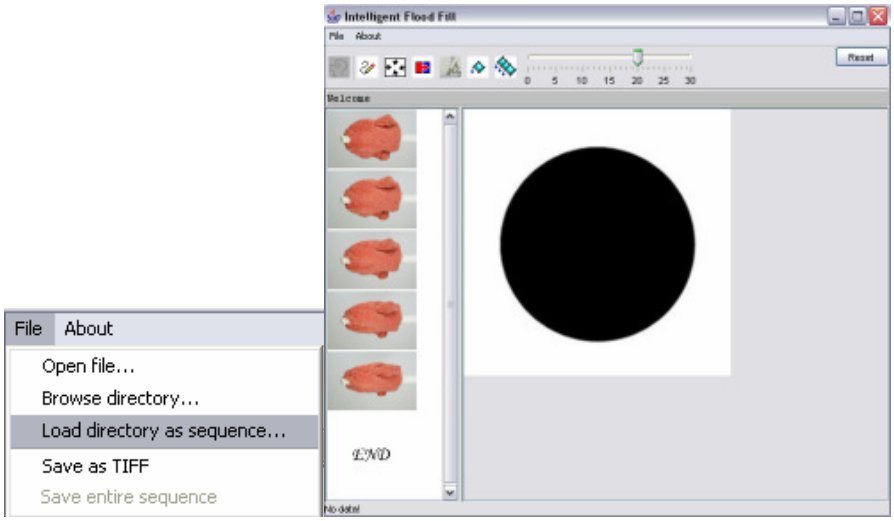


Figure 11 - Load Directory As Sequence, Figure 12 - Result of loading directory

2. Information for the flood fill, such as a scribble and bounding box, need only be specified on one of the images. However, it is important to choose the right image, as the information will be used for all flood fills. So, it is important to choose the image where the object is at its 'largest' (otherwise the bounding box may cut off areas in other images) - or at least make allowances for the object being different sizes - and also choose an image that has a sample of all the colours that are present in the object viewed from any angle.

For this sequence, the first image has a good angle where there are a couple of darker and lighter areas that represent a good colour sample, and is also represenative of the object at its largest size. Scribble and bounding box information will be painted on as described previously.

Figure 13 - Image with scribble and bouncing box information

3. To floodfill the sequence, use the 'Floodfill all' button, as opposed to the normal floodbill button. Again, with a large amount of high resolution images, this could take sometime. After floodfilling, the icons on the left will change to show the floodfill, and the main image will load again to show the new floodfill.

Figure 14 - Floodfill all button

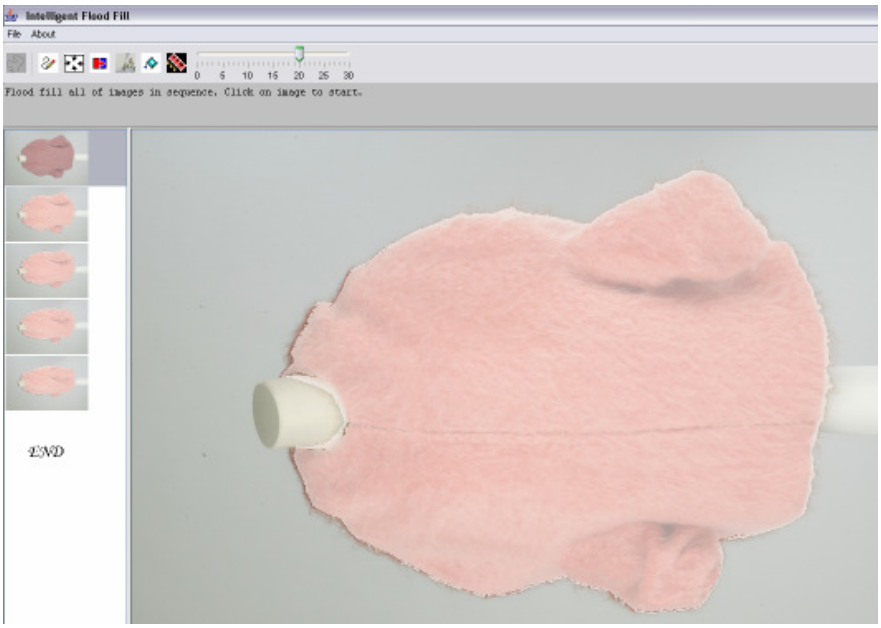


Figure 15 - Floodfilled sequence

4. Changes can now be made to individual images if needed, using the same techniques as described before. In order to make a change to an individual image rather than floodfilling all again, then use the normal floodfill button rather than the 'floodfill all'.

5. If all images are satisfactory then select 'Save As Sequence' from the 'File' menu to save the entire sequence to a folder. You will be asked to supply one filename, e.g. "sequence.tif" and all files will then be given the names "sequence0.tif", "sequence1.tif" and so forth.

Troubleshooting

This section will be updated as needed.

1. Slider bar
There are known issues about the slider bar appearing squashed at small resolutions, or when not maximised. This is being fixed.