

COMP3750 Project Report: Mondrian Library

Kerryn Boorman (u4114385)

supervised by Tom Gedeon

November 2008

Abstract

Abstract images in the style of Piet Mondrian as good candidates for exploring ideas about aesthetics and art in a computing environment. This implementation project provides a Mondrian-style image generation and manipulation library, incorporating the work of two previous projects to provide a base for the further exploration of these ideas.

1 Introduction

The abstract artworks of the 20th century Dutch painter Piet Mondrian exhibit the aesthetic qualities of great art combined with a simplicity of form that allows their images to be easily reproduced using computers. In fact, the style of images that Mondrian dubbed neoplastic follow simple structural rules that can be used to produce new Mondrian-like images.

Their simplicity combined with their aesthetic qualities make Mondrian-like images ideal for exploring the concept of aesthetics.

A number of studies have already been conducted with such aims.

McManus, Cheema & Stoker (1993) used a computer to generate sets of images containing one identical to an original Mondrian and two where the lines had been subtly moved. Their study established that the structure of the original Mondrians were more aesthetically pleasing than the modified images.

Two previous projects have been conducted at DCS that have built on the idea of Mondrian-like images.

Jian Shen's Darwindrian project (Shen & Gedeon 2007) produced a Jython application that evolved sets of Mondrian-like images in an attempt to produce more aesthetically pleasing images. It used chromosomes specifying a set of image parameters to produce a generation of images, which a user then ranked in terms of structural and colour fitness. A bacterial evolution algorithm was then applied to produce a new generation of chromosomes based on the rankings. Each subsequent generation was designed to produce more aesthetically pleasing images.

The Mondrian Drawer project was summer research project that I undertook at the beginning of 2008. In it, I developed a drawing application for the creation of Mondrian-like images. It constrained the user to drawing valid Mondrian-like images, but allowed them to adjust the image to suit their taste.

The purpose of this project is to develop an external library that provides the image manipulation and evolution functionalities on these two applications, as well as the intermediate functionality of image generation. The library allow any calling application to create and modify Mondrian-like images. One potential use of the library is as a base from which to examine the benefits of different types of user interfaces. (Such as multi-touch screens, or driving gear as interface)

Section 2 covers the requirements for the library. Section 3 dis-

cusses the library's design. Section 4 covers the details of the implementation. Section 5 evaluates the library and concludes the report.

2 Requirements

The Mondrian Library will be implemented as a Java object library that allows the user to create, manipulate and evolve Mondrian-like images.

Mondrian-like images are defined as follows:

- Images have a solid white¹ background
- All lines are horizontal or vertical, black¹, and either extend to the image boundaries or terminate in the middle of another line (ie no right angles or dangling ends)
- All blocks of colour are fully enclosed by lines, and are red, blue or yellow¹

(images as defined by McManus et al. (1993) as those from Mondrian's classical mature period)

2.1 Image Specification

A new image model must be developed to allow valid Mondrian-like images to be specified in a consistent format across the library. The Library must also provide text formats for specifying its various objects, allowing them to be saved to disk and reloaded. Mondrian-like images must be able to be displayed on the screen and also saved to various standard image formats.

2.2 Image Manipulation

The Library will provide a mutable image object that allows the user to either define a new Mondrian-like image on a blank canvas, or modify an existing image that has been loaded from file.

The following manipulation operations must be supported:

- Creation of a new valid line based on given coordinates

¹The colour definitions of white, black, red, yellow and blue are based on the typical shades used by Mondrian in his paintings, not on the standard RGB definitions of these colours.

- Deletion of an existing line based on a given coordinate
- Relocation of an existing line to a given coordinate
- Colouring of an fill area designated by a given coordinate

The image is to restrict manipulation to operations resulting in a valid Mondrian-like image.

2.3 Image Generation

The Library will provide a chromosome object that stores a set of parameters for the generation of Mondrian-like images.

This chromosome will be based on the Python Chromosome object from the Darwindrian application (Shen & Gedeon 2007). The chromosome will use probabilities and a set of start points to generate one of set of possible Mondrian-like images.

The images generated from the chromosome should be able to be modified using the image manipulation functionality if desired.

2.4 Image Evolution

The Library will provide evolution functionality based that provided in the Darwindrian application developed by Jian Shen (Shen & Gedeon 2007).

Chromosome objects will exist in generations, where they will be ranked in terms of image fitness. Subsequent generations will be evolved by crossing the chromosomes in the previous generation using a genetic algorithm that preferences the fittest ones. This algorithm will be translated from the one used in the Darwindrian Python implementation.

3 Design

As per the requirements, the Mondrian Library is broken into three broad areas of functionality: image manipulation, image generation and image evolution.

The universal image behaviour of load, save and display is provided by the Image superclass, which also stores the internal representation of the image. The design of the image model used in the library is described in the next subsection.

The Image subclass MutableImage provides the image manipulation functionality.

Image generation is provided by the Chromosome object. It stores a set of probability-based parameters that are used to generate GeneratedImage objects.

Image evolution is provided by the EvolutionManager. The EvolutionManager establishes Generations of Chromosomes. The Chromosomes can be viewed and ranked based on the fitness of the images they produce. Once the Chromosomes in a Generation have been appropriately ranked, the Generation can use these values to cross the Chromosomes using the genetic algorithm to produce the next Generation.

The top-level design of each of these classes will be discussed in detail in the subsections following the Image Model design, with their implementation discussed in the corresponding subsections in the Implementation section.

3.1 Image Model

3.1.1 Background

Shen's Darwinian implementation provided an image model based on lists of lines and rectangles – the coloured spaces between the lines. Interactions between objects were computationally expensive as neither lines nor rectangles stored any information about their relationship to the image structure as a whole.

This model was not sufficient to provide the image manipulation facilities required by the MondrianDrawer application, and so I developed a model based on the points of intersection in the image. Neither lines nor coloured rectangles were allowed to overlap, so every line intersection was represented by a point that stored the associated lines and their directions, as well as any rectangles with had a corner originating from this intersection. Rectangles and lines also stored references to their associated intersections.

This Drawer model allowed the model to be manipulated relatively easily, but the complex interactions between the various elements of the image meant that it was difficult to reason about correct behaviour. Slight differences in operations performed on horizontal and vertical lines also led to the duplication of hundreds of lines of code.

3.1.2 Design

The Library introduces a new image model that simplifies the both the logic and its actual implementation, thus reducing redundant data and making it easier to reason about algorithms and operations on the images.

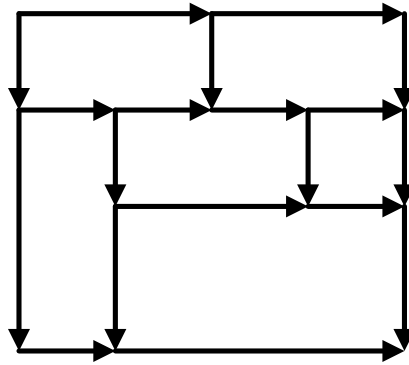


Figure 1: Directed Acyclic Image Graph

The image model is a directed acyclic graph made up of vertical and horizontal line segments. (See figure1). All segments are directed either right (if they are horizontal) or down (if they are vertical). All graphs are rectangular, with non-displaying boundary segments forming the edges of the graph.

After the issues with code duplication experienced in the Drawer model, the Library model normalises the operations on vertical and horizontal lines by specifying a segment's connections relative to its direction. The references `prev` and `next` must be parallel continuations of the current segment, `s1` is an orthogonal line entering the segment at the start, `e2` is an orthogonal line exiting the segment at the end etc.(See figure 2).

For example a common operation when deleting segments from the image is that of merging a segment with the one following it. This occurs when the removed segment was the equivalent of the bar at a T intersection. (See subsection 4.3 on the implementation of `MutableImage` for more information)

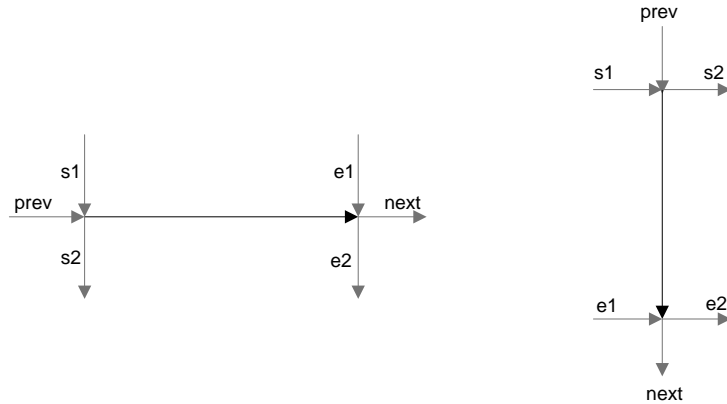


Figure 2: Line connections

As you can see from the pseudo code in figure 3, it is easy to assert the correct state to permit merging, and to merge the segments, without any reference to the orientation of segment s .

3.1.3 Traversing the model

The directed nature of the graph allowed the development an algorithm that always traverses the graph along the same path.

If provided with the top-left horizontal segment of the image, the traversal algorithm will traverse all lines in the image, right-to-left and top-to-bottom. This path is illustrated in figure 4, with the solid horizontal lines being traversed first, and the sets of vertical being traversed left-to-right after the last horizontal of the line above where they start.

The algorithm is based on an iterator model. The iterator stores a line h – the first horizontal at the beginning of iteration – and two sorted lists, one of vertical lines, one of horizontals.

It starts at the stored h , and follows it through until it ends, adding to the verticals list any vertical lines connected below the horizontals as they are processed.

It then sorts the verticals left-to-right, adding to the horizontals list any horizontals connected to the verticals.

```

1 mergeNext (segment s) {
2   // must have next to merge with
3   assert s.next != null
4
5   // no incoming connections at the mergepoint
6   assert s.e1 == null && s.e2 == null
7
8   // move the end coordinate of s to that of next
9   s.endX = s.next.endX
10  s.endY = s.next.endY
11
12  // update the references in s to point to those
13  // at the end of s.next (updates the specified
14  // lines to point back to s as well)
15  s.setE1(next.e1)
16  s.setE2(next.e2)
17  s.setNext(next.next)
18 }

```

Figure 3:

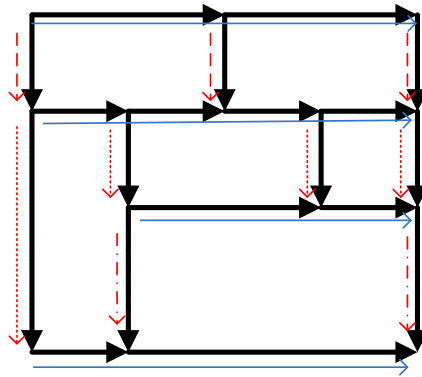


Figure 4: Traversal of the Graph

It then sorts the horizontals list top-to-bottom and left-to-right, so that the first one is the leftmost one in the highest location, and sets

this to the new h, removing it from the list of horizontals.

This h is then processed as before, following its *.next* through to the end, making sure to remove any *.next* from the horizontals list when it is set.

The verticals are then processed as before, and the cycle continues until h is not set and both lists are empty.

3.2 Describing Filled Spaces

The above traversal algorithm allows the image model to be purely line-based, despite the need to specify coloured fills for the rectangular spaces. Because the top-left line of any space is always encountered first, it is simple just to mark this line with a colour if the area is to be filled.

When painting, the space is coloured before its top-left border line, with the other borders being painted as they are encountered. Painting the space can be done by following the connections in the coloured line to establish the boundary coordinates of the space. Running down the line's *s2* and along its *next* finds the bottom and right boundaries. (An *e2* on the vertical *s2.next* indicates the bottom of the space and when an *e2* on the *next.next* indicates the right-hand side of the space). The coordinates from these ends can be combined to indicate the bottom-right corner of the space, thus allowing it to be painted.

Using a line to indicate a fill also simplifies the implementation of the fill methods, allowing them simply to search for the appropriate fill line - the closest horizontal line with an *s2* that start above and to the left of the given coordinate.

3.3 Other benefits

The new design also provides a simple method for image validation that has been invaluable during debugging: traverse the model, and for each segment check that its connections conform to the rules of Mondrian-like images and that its coordinates are correct within the graph.

3.4 Basic Image Services

The basic Image class is the parent of the MutableImage and the GeneratedImage classes. It provides the basic images services on an immutable image.

It is provided to the user as instantiable class to cater to the possibility of an application wanting to do basic Image display without any additional functionality. As such it allows images to be loaded from and saved to text files, painted onto Java graphics objects for display, and saved to various image formats.

3.5 Image Manipulation

The image manipulation services are provided by the MutableImage object. The MutableImage is the only object in the library that can be changed after it has been initialised.

Three different considerations influenced the design of the MutableImage - user convenience, image integrity and execution efficiency.

User convenience means that the user should be able to issue commands easily, and not have to deal with the internal representation of the image. Therefore the operations are provided as a set of methods that perform specific well defined actions, and, as per the requirements, take as parameters coordinates within the image space. The image will either perform the operation as requested, or throw an exception detailing why the request could not be fulfilled. The coordinates are approximated to indicate anything within a three percent margin of those provided to allow for differences between the display and the model etc.

Additional image integrity and efficiency considerations, as detailed in subsection 4.3 of the Implementation section, led to the development of the current interface, which allows the following operations:

```

MutableImage(int width, int height)
a blank image is created with the specified width and height
addLine(int sx, int sy, int ex, int ey)
attempt to draw a line between the given coordinates
deleteLine(int x, int y)
attempt to delete a line at the given coordinates
startMove(int x, int y)
intiate a move of the lines at the given coordinates (if successful, only update-
Move operations permitted until endMove)
updateMove(int x, int y)
attempt to move the lines specified in start move to the given coordinates
endMove()
conclude the current move and return to normal operation
getFillColor(int x, int y)
get colour for rectangle containing given coordinates
setFillColor(MondrianColor c, int x, int y)
set the colour of the rectangle containing the given coordinates to c
cycleFillColor(int x, int y)
change fill colour of rectangle containing the given coordinates to the 'next' fill
colour

```

Of course, as a subclass of Image, all of the basic image services of save, load etc are also available.

3.6 Image Generation

The image generation services are provided by the Chromosome object, which creates GeneratedImage objects using its stored parameters.

The Chromosome is designed to mimic the operation of the Python Chromosome object used by Darwindrian.

It stores lists of parameters representing the probabilities that control image generation. The types of parameters stored are the same as those stored by Darwindrian – a list of initial points, a list of line direction probabilities at each loop iteration, a list of probabilities deciding the likelihood of lines from a given point type (defined as the shape formed by the lines around the point eg. right-angle or cross etc), and list of probabilities for the various coloured fills.

Image generation is an iterative process, based on that used by Darwindrian. The Library also make image generation recreatable – each image is generated using a specific random seed, which together

with its Chromosome and dimensions, uniquely identifies it. The dimensions are necessary as the Library extends the capabilities of the Darwindrian Chromosome to allow the generation of images of any given size. This is achieved by expressing the initial points as x-y percentages, which can be multiplied to give equivalent coordinates for any image space.

Shen's generation process (as described in (Shen & Gedeon 2007)) and the differences between it and the new process adopted by the Library are discussed below.

Darwindrian generates images by looping over all the initial points specified in the Chromosome a fixed number of times, and emitting lines from them.

In each iteration, the points were assessed based on their type – The decision whether to emit a line was made by the Chromosome, based on probabilities stored for each point type.

After the final iteration of the main loop, the specified image is unlikely to be a valid Mondrian-like one – it may contain right angles and lines that terminate in the middle of the air. Darwindrian solved this issue with a validation phase, which forced any invalid points to emit new lines until they were valid.

The generation process used in the Library is also iterative, but reverses the process used in Darwindrian. Instead of building an incorrect model, and then forcing it to become valid, the Library generates an image by building the maximal model – the one with all possible lines that could be emitted from the generation points – and then iteratively removing lines from the points based on removal and direction probabilities in the Chromosome.

The final phase of both generations is to allocate fill colours based on the colour distribution stored in the Chromosome.

The Library stores images generated from Chromosomes in `GeneratedImage` objects. `GeneratedImage` is a subclass of `Image` which in addition to the image model, holds a reference to the generating Chromosome, and a copy of the random seed used for this particular image. `GeneratedImages` and `Chromosomes` are immutable, as any modification would invalidate the cross references. However, `GeneratedImages` can be converted to `MutableImage` objects if the user wishes to manipulate them. Chromosomes can not be modified, but new Chromosomes can be specified from existing ones, and from modifications of existing ones.

Because a Chromosome can generate a very large set of different

images (possibly one image for every value of the long random seed) the Library provides an additional class for managing Chromosome-Image relationships. The ChromosomeImageMap is a map of all possible random seeds to their associated GeneratedImage for a given Chromosome.

This functionality will be particularly useful when combined with the Evolution interface. The original Darwindrian displayed a single image generated from a Chromosome, and asked the user to rate the Chromosome's fitness. This was a less than ideal approach, as the fitness of the image does not reflect the fitness of the range of images that can be generated from a Chromosome. By providing access to more of that range, the fitness of the Chromosome can be more accurately assessed.

3.7 Image Evolution

The Library provides evolution functionality through the EvolutionManager class, and its related classes Generation and Chromosome.

EvolutionManager is a manager construct that keeps track of an evolutionary series. It stores the history of the previous generations, as provides a single interface for evolving and regressing generations.

Each Generation object contains a list of Chromosomes and a list of images generated from the Chromosomes.

The evolution of a new generation from an existing generation is done using the evolutionary algorithm designed by Jian Shen for the Darwindrian (Shen & Gedeon 2007). Chromosomes are ranked in terms of fitness and their recorded fitness, and crossed to produce new Chromosomes for the next generation. Chromosomes are crossed using Jian's bacterial crossing approach, where the new Chromosome is predominately the same as its main parent, and has only absorbed a small amount of information from the second parent.

3.8 Image Save Formats

Images from the Library can be saved as both standard image files, and as text descriptions. Other objects can also be saved as text.

Images will be saved with an initial descriptive section describing the image model, which in the case of a GeneratedImage will be followed by its random seed and and a description of its Chromosome.

Generations, and Generation histories can also be saved. A Generation is represented by a list of its Chromosomes, and a history is simply a list of Generations.

```
Image: X x Y

Line: (x1, y1) (x2, y2)
Line: (x1, y1) (x2, y2) Fill RED
...

Seed: value

Chromosome:
Complexity: value
Loop:      value

DirDist:  1          up:      down:      left:      right:
...
DirDist:  loop      up:      down:      left:      right:

StructDist:      cross:      nodal:      line:      free:
ColourDist:      red:      blue:      yellow:

OriginPoints:
1          %x %y
...
comp      %x %y

Colour Fit: value
Struct Fit: value
```

The text format used by the library is human readable, as seen above, but it uses the line-based structure of the image model to describe the object. In Darwindrian and Drawer the text descriptions contained both lines and fills separately, whereas the Library indicates fills on the top-left line of a section. This may seem less clear than the previous formats for human reading, but as any real examination of the image would require plotting all the lines, there is really no difference in the information provided. In any case, given a copy of the Library, it should not be necessary to attempt to read the files, they can be simply reloaded.

3.9 Testing and Showcase Application

The last aspect of the project is the new GUI interface, the Mondrian-Imager.

It was developed to showcase the library and to facilitate testing of the library's functionality. Its simple design is built around the provision of three tabs, each of which provides access to a different area of library functionality.

The Draw tab presents a blank canvas and a status bar, and allows the user to test the image manipulation functionality. Adding a line is done by dragging the mouse, with the left mouse button down, either horizontally or vertically on the canvas. Deleting a line is done by right-clicking a segment (alt-click when running under OSX). Fill colour is cycled by left-clicking in a space. Lines or intersections can be moved by dragging them with the right mouse button (dragging with alt under OSX).

The Generate tab provides a form on the left for specifying the parameters of a Chromosome, including a seed, and a canvas on the right which displays an image generated from that Chromosome with that seed. Modifying the parameters creates a new Chromosome with the requested parameters and generates a new image. Modifying the seed causes the canvas to display the different images that can be generated from the currently specified Chromosome.

The Evolve tab presents a user interface based on that of the Darwindrian application, with a list of current images on the left, and a large canvas for displaying them individually on the right. Two scroll bars indicate the current fitness ratings of the Chromosome for the displayed image, and allow them to be adjusted. Buttons allow the user to evolve the next generation of images from the current generation, or revert to the previous generation. It does not currently provide access to the ChromosomeImageMaps of Chromosomes, only single images as were provided by Darwindrian.

4 Implementation

The implementation section covers the details of the logic and code used to create the Library.

4.1 Image Model Implementation

The design section covered most of the interesting aspects of the implementation of the image model when it discussed the connections in the basic Line object and the design of the traversal algorithm.

The model is implemented as a LineSet object. It stores a single reference to a horizontal line representing the top-left corner of the image. Traversal is implemented by the provision of standard Java iterator for the Set, which uses the traversal algorithm discussed in the design section to determine the next line in the set.

The iterator also uses a special SortedLine class to specify the order in which the verticals and horizontals are to be sorted for extraction.

4.2 Base Image Implementation

The Image classes display and save their image models by painting them onto Java graphic contexts.

The images use the RGB colour definitions of the typical shades used by Mondrian, as defined by Jian Shen for the Darwindrian application. For the purposes of this discussion, I will refer to the colours as red, blue, yellow, white and black, although they do not conform to the standard RGB definitions of these colours.

Images are painted onto a white rectangle the size of the image dimensions. Each line is painted when it is encountered by the model iterator. Fills are painted when the horizontal line forming the top left corner of their space is encountered. Their boundaries are established by following the surrounding lines down and to the right. Because of the order in which the iterator finds lines, the top left horizontal is always found first, and the fill will always be under all of its enclosing lines.

Painting is more efficient in the library than it is in the Drawer or in Darwindrian as they all store the same number of lines, but the Drawer and Darwindrian must also store a list of fills, and must iterate over both collections, whereas the library achieves only requires a single traversal of the lines.

4.3 Manipulation Implementation

The redesign of the image model, as well as the small number of continuing errors in the line moving code in the MondrianDrawer made it necessary to reimplement the manipulation methods for inclusion in the library.

This redesign, with the benefits in logical provided by the new image model, has resulted in faster, less convoluted methods which are also reliable and robust.

4.3.1 Adding Lines

For a line to be valid in a Modrian-like image, it must go from the middle of one existing line to the middle of another.

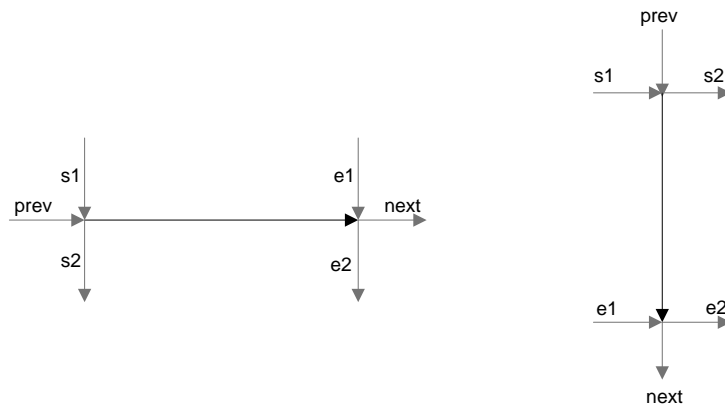


Figure 5: Line connections for adding line

As you can see in figure 5, the image model represents this line and its bounding lines as five distinct segments. The line is bounded by $s1$ and $s2$ at one end, and $e1$ and $e2$ at the other end.

To add a new line to the model, the user specifies a pair of start and end coordinates. If these coordinates specify a horizontal or vertical line within the image space a new line will be added.

The model cannot add a line between arbitrary points, only between existing lines, so it needs to locate the start line and end line

that the new line must be extended to to make it valid.

These lines are found by iterating through the image model and checking the location of existing orthogonal lines. It keeps track of the closest lines to the beginning and end of the new line, as well as any lines crossed by the new line. If any lines start or end exactly on the new line only the one that ends at the new line is recorded.

Once the method has checked all the lines, and has the start and end, it can begin to add the new line.

The found end line is added to the list of crossed lines, and these are sorted to be in order of their proximity to the start line.

If the start line does not end at the start coordinates of the new line, the start line is split into two so that the new line can be connected in.

The `addLine` method then loops through all the crossed lines in order, splitting them if needed, and creating the segments joining them that form the new line. If any of the needed segments already exist, they are simply skipped over.

Thus `addLine` efficiently adds a valid line to the image model.

4.3.2 Deleting Lines

There are two approaches that could be taken for deleting a line from the model.

The more complicated approach would be allow to the user to indicate any line segment in the model, and if the model is not valid after that segment is removed, continue removing lines until it is.

The other approach is to only allow segments to be removed if the model is still valid without them.

The library, like the `Drawer` before it, only implements the first approach. This may seem limiting, but as a user, it is simple to see which lines need to be removed before you can remove a particular segment. The nature of the connections in the model makes this difficult to determine programmatically, because whether or not a certain line needs be removed is dependent on all the lines around it.

Thus `deleteLine` merely establishes if a segment has bounding lines on all sides (that is, `s1`, `s2`, `e1` and `e2` are not `null`). It takes a coordinate and checks the closest segment to that point. If the segment is bounded, it removes it by setting the back references to it in its connecting lines to `null`. Each pair of bounding lines is then

checked to see if they should be merged – this occurs when the removed segment was the only reason for the intersection.

4.3.3 Relocating Lines

Moving lines is the most complicated operation that can be performed on a `MutableImage`.

The basic move case is simple: a single segment, with no previous or next segments, moving parallel to its bounding lines. If this movement does not reach the ends of the bounding lines, it is a simple matter of updating the line's coordinates, and those of the bounds, to indicate the new position of the intersection.

This basic case is not useful, however, as it could just as easily be done by deleting the existing line and replacing it in the desired location. In fact deletion and addition has more flexibility, as the basic move would not allow a line to cross an intersection that occurs in one of its bounding lines, despite the fact that the space in which the line is may not be affected by this intersection.

However, to provide a useful move facility, the user must be able to drag segments to locations that invalidate their current connections. To enable this to be done efficiently - allowing the user to see what their changes would look like without completely revalidating the model for every

Moving is implemented as a three-part process. The user initiates a move at a particular coordinate, and if there is a line or an intersection there, the image establishes all segments that will be affected by the move. It then moves into a `MoveState`, which restricts the actions on the image to move updates until the end of the move is signalled.

This is necessary because the image model is different during a move – the moving lines are allowed to slide on top of the image within certain bounds, with their connection information stored in separate objects. Thus the image must be reconciled to the standard format before normal operations resume.

The move case implemented in the Library provides enough functionality to be really useful. It allows the user to move whole lines or intersections (moving two orthogonal lines at once) within the space that they exist. It does not however, allow lines to move on top of, or past, existing lines.

This is equivalent to the functionality provided for moving in the `Drawer`. The reimplementations with the new image model has how-

ever eliminated the persistent bug in the Drawer's move that would occasionally cause its image model to become invalid.

4.3.4 Changing Fill Colours

Fill colours in the Library image model are stored on the top-left line of the 'fill space', or the space enclosed by a rectangle of lines. The benefits this has for painting have already been discussed, but it also makes it simple to change the colour of a fill.

Every coordinate in the image space is associated with a particular fill – that of the closest 'fill line' above and to the left of it. Thus a fill command iterates through the model to establish the closest fill line, and then performs the requested action.

Three fill actions are available to the user – `getFillColour` which returns the colour (white, red, yellow or blue) of the selected fill, `setFillColour` which sets the colour of the fill to a specified valid fill colour, and `cycleFillColour`, which changes the specified fill to the next colour in a fixed list of fill colours.

4.4 Generation Implementation

As described the Generation subsection 3.6 of Design, the Chromosome side of image generation is very similar to that used by Darwindrian.

The actual image generation is done as part of the initialisation of the `GeneratedImage` object. The image is initialised, and then all possible lines from the points in the Chromosome are added. Special `GeneratePoint` objects are defined to hold references to the points for generation. The image then loops through all the generation points removing the lines surrounding them based on the probabilities calculated by the Chromosome.

Add and removing the lines use the methods defined for the implementation of image manipulation.

4.5 Evolution Implementation

The Evolution facilities are implemented as Java versions of the Python code written for Darwindrian.

5 Evaluation and Conclusion

This report covers the design and implementation of a new library for generating, evolving and modifying Mondrian-like images.

The Library meets the basic requirements set down for the project, although due to illness and the associated time constraints, there was not time to explore the additional avenues listed as potential extensions to the project – ie those of exploring in detail the range of ways images can be generated, and implementing a WiiMote interface, or anything more than basic functionality, for the new Mondrian Imager application.

The Library provides a more efficient and logical image model than the applications it is based on, and thus exhibits improved efficiency, comprehensibility and robustness.

It extends the abilities of the Darwindrian model to allow Chromosome image generation to be seeded and thus made repeatable, and provides the useful abilities to move backwards and forwards through the Generations in the EvolutionManager.

It improves the implementation of the Drawer functionality greatly – predominately through the new image model, and the robust implementation of the line moving facility.

5.1 Future Work

During this project, I noticed a number of areas that could do with further development.

In particular, the reasoning and logic underpinning the design of the Chromosome and its crossing mechanisms should be examined, as it seems possible to define a more deterministic Chromosome that would result in more predictable images, and thus its fitness would be better able to be assessed. The use of a bacterial evolutionary approach could also be contrasted by that of a normal genetic approach compare the differences in the images produced when the child chromosome is more of a combination of its parents.

There is also a potential to establish a algorithm that builds a Chromosome from a drawn image by calculating the distribution of the lines around the points, and of the colours etc.

Although a basic comparison showed that the style of images produced by the new generation technique is similar to the old, the differences could be further explored.

The listed extensions - the exploration of the range of ways, and varying amounts of user control, that can be used to create images, and the development of a WiiMote interface for the Image application - could also be explored.

References

- McManus, I. C., Cheema, B. & Stoker, J. (1993), 'The Aesthetics of Composition - A Study of Mondrian', *Empirical Studies of the Arts* **11**, 83–94.
- Shen, J. Y. & Gedeon, T. (2007), 'Cyber-Genetic Neo-Plasticism – An AI program Creating Mondrian-like Paintings by using Interactive Bacterial Evolution Algorithm'.