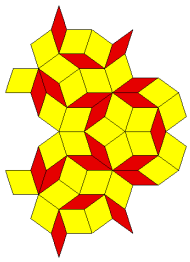
University of Manchester
Department of Computer Science

BSc (Hons) Artificial Intelligence



FINAL YEAR PROJECT

Tile Sets as Turing Machines

Author:
Andrew Webb

Supervisor:
Dr. Ian Pratt-Hartmann

May 4, 2010

Abstract

This report describes a third year project in the computer science department at the University of Manchester. In the course of the project, four programs were created. The first explored the connection between tilings of the plane and tile sets, and shows some tiling problems as general models of computation. It did this by creating tile sets, tilings of which simulate Turing machine behaviour. The purpose of the other programs was to explore the properties of different tile sets, particularly aperiodicity. The second program constructs Wang domino tilings, the third constructs Robinson tilings, and the fourth constructs Penrose tilings.

Tile Sets as Turing Machines

Author: Andrew Webb

Supervisor: Dr. Ian Pratt-Hartmann

May 4, 2010

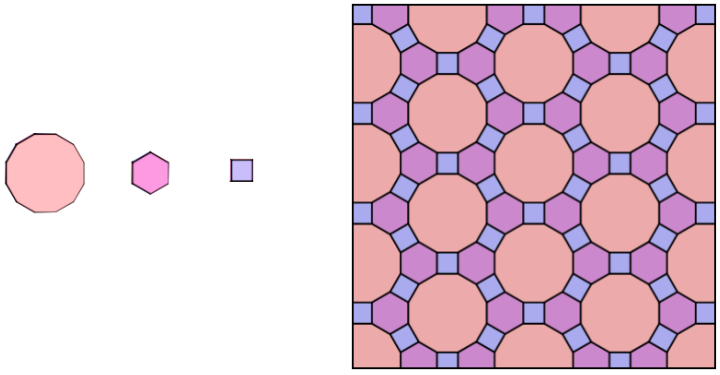
Contents

List of Figures

Chapter 1

Introduction

The main objective of this project is to explore the properties of tilings of the plane, and their relationship to the computability of functions. All references to tiling and tile sets in this report refer to the 2-dimensional model, unless otherwise specified. Therefore, a tile is a 2-dimensional polygon. The only restriction we place on this polygon is that is it not self-intersecting. A tile set is a set of such 2-dimensional polygons, and we consider only finite tile sets. A tile set can tile an area just in case the the tiles can be arranged in such a way that they cover the area without overlapping. Not every tile in the set has to be used and each tile can be used any number of times. A tile set is said to be able to 'tile the plane' if there is an arrangement that can cover an infinite area. Where we refer to higher and lower dimensions the terminology changes in the obvious way; A set of 1-dimensional tiles may 'tile the line' and a set of 3-dimensional tiles may 'tile the volume'. Figure 1.1 shows a tile set and a section of a tiling using only that set.



(a) A tile set

(b) Section of a tiling produced from that set

Figure 1.1: A periodic tiling

The project is broadly split into two sections. The first section is concerned with exploring the relationship between tiling problems and computability, by creating tile sets that can simulate Turing machines. The goals are as follows:

- write a program that, when given the description of a Turing machine, produces a set of tiles; and
- write a program that, when given a set of tiles and an input, tiles the plane in such a way that it simulates the original Turing machine running on that input.

A technical preliminaries chapter follows this one. It gives a formal definition of a Turing machine. Chapter 4 explains precisely how a set of tiles can simulate the behaviour of a Turing machine.

The second part of the project is exploratory; it is to explore the properties of different types of tile sets. Therefore, when the project started, the goals were not well defined. As the project progressed, these goals were defined as follows:

- write a program that can automatically tile the plane with any set of Wang dominoes;
- write a program that can automatically tile the plane with Robinson tiles;
- this program must also allow the user to manually build Robinson tilings;
- write a program that can automatically tile the plane with Penrose tiles; and
- this program must also allow the user to manually build Penrose tilings.

These three sorts of tile sets - Wang dominoes, Robinson tiles, and Penrose tiles - are described in detail in chapter 3, and some definitions are introduced in the technical preliminaries chapter.

The programs produced for this project are of academic interest only, and may be used as demonstration tools.

1.1 Report Overview

Chapter 2 introduces technical terms that are used throughout this report, and includes a formal description of a Turing machine. Chapters 3 and 4 give the essential background of aperiodic tile sets, computability, and the intersection of the two. Chapter 5 covers the design of the tools created for this project. In chapter 6 we will see some of the more interesting issues that arose during the implementation phase of the project. Chapter 7 contains a critical evaluation of the tools created and a comparison to other work. Chapter 8 briefly concludes the report, and we will see how the project may be extended in future work as well as some properties of tilings and computability that were not explored in the project.

Chapter 2

Preliminaries

In the following we define technical terms that will be used throughout the report. Technical terms related to tiling the plane and those related to computability are treated separately.

2.1 Tiling

A tiling is periodic if it contains a finite area such that the tiling can be constructed by translations of this area. A tiling is non-periodic if there is no such area. A tile set is aperiodic just in case it can construct only non-periodic tilings. These definitions will be discussed in more detail, with examples, in chapter 3. The following are tiling problems:

- the origin constrained domino problem: given a set of tiles, determine if it is possible to construct a tiling of the plane from some initial configuration;
- the unconstrained domino problem: given a set of tiles, determine if it is possible to construct a tiling of the plane;
- the periodic domino problem: given a set of tiles, determine if it is possible to construct a periodic tiling of the plane.

The golden ratio is strongly related to Penrose tiles, which will be discussed in detail. Two quantities are in the golden ratio if the ratio of the sum of the quantities to the larger quantity is equal to the ratio of the larger quantity to the smaller one. The symbol φ is used to represent the golden ratio throughout this report.

2.2 Turing machines and computability

The following is a definition of the model of Turing machine used in this report. The definition used is non-standard but is necessarily equivalent to any variation of Turing machine.

The Turing machine is, at all times, in a single state from a set of states Q. It has a bi-infinite tape (that is, a tape that extends infinitely both to the left and to the right) that is divided into discrete and contiguous cells. The machine has a head, which can read, write, and move left or right. The tape contains symbols from a finite alphabet Γ. Formally, the Turing machine is a 7-tuple

$$M = (Q, \Sigma, \Gamma, A, \sigma, s, T),$$

where

- Q is a finite set of states;
- Σ is a finite set symbols that can be used to construct an input string;
- Γ is a finite set, the alphabet. $\Gamma = (\Sigma + \vdash + \sqcup)$ where $\vdash$ is the left end-marker symbol and $\sqcup$ is the blank symbol;
- A is a finite set of actions. $A = \Gamma + \{L, R\}$;
- $\sigma : Q \times \Gamma \to Q \times A$, the transition relation;
- $s \in Q$, the start state; and
- $T \subseteq Q$, the set of accepting states.

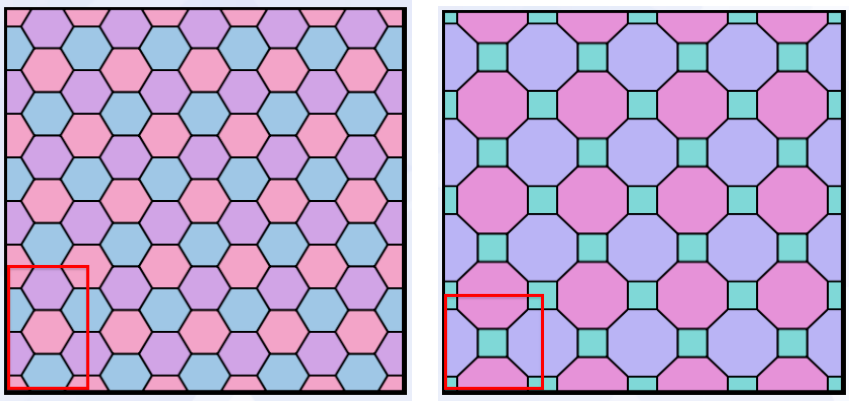
The transition relation σ defines the behaviour of the Turing machine and gives a set of rules of the form $r = (q_0, s, q_1, a)$ where $q_0, q_1 \in Q$, $s \in \Gamma$, and $a \in A$. The rules should be read as 'When in state q_0 and reading symbol s, go to state q_1 and take action a'. If $a \in \Gamma$ this is a write action, and a will be written on to the tape. If $a \in \{L, R\}$ this is a move action and the head will move left or right. Some Turing machines allow a write action and a move action to be taken simultaneously, but the description used here allows only one or the other.

The machine will halt if it reaches an accepting state. The machine will halt in a stuck configuration if it is in a state q_0, reading a symbol s for which there is no transition rule $r = (q_0, s, q_1, a)$. The machine is deterministic if for every combination of current state and symbol there is at most one transition rule and is non-deterministic otherwise. An input string I is computable on a machine M just in case M run on I will eventually reach an accepting state.

Chapter 3

Periodicity and Aperiodic Tile Sets

Most of us understand the idea of a periodic tiling. Figure 3.1 shows two periodic tilings. In each, a section is highlighted in red. They are periodic because the infinite tiling consists of translated copies of this section. Non-periodic tilings, on the other hand, are not so easy to visualise. Such a tiling covers an infinite area with a finite set of tiles in such a way that it doesn't fall in to repetition.



(a) A regular tiling of hexagons

(b) A regular tiling of squares and octagons

Figure 3.1: Two periodic tilings, with the period highlighted in red

3.1 A One-Dimensional Example

To assist in understanding non-periodicity, we will first look at a one-dimensional example, which relates tilings to real numbers. Figure 3.2 shows a set of 10 one-

dimensional tiles, which map onto the digits 0-9. The tiles are coloured so that we can distinguish them, and do not represent edge-matching rules. Each tiling of the line with this tile set corresponds to exactly one infinite string of decimal digits.

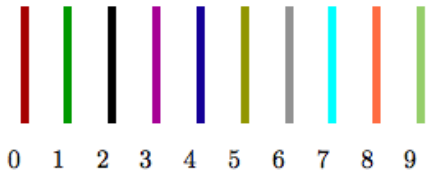
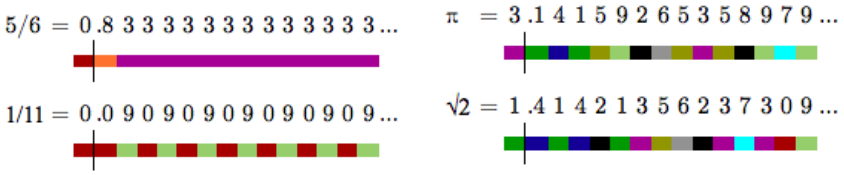


Figure 3.2: One-dimensional tiles mapped onto decimal digits

Recall that any rational number must ultimately reach a periodic sequence of digits, and that an irrational number never falls into periodicity. For example $5/6 = 0.83\dot{3}$ is not periodic, but it contains the infinite periodic sequence $333\ldots$, and so the corresponding tiling contains an infinite periodic sub-tiling. one-dimensional tilings that correspond to irrational numbers will not contain an infinite periodic tiling. Figure 3.3a shows sections of tilings corresponding to the rational numbers 5/6 and 1/11 (in the diagrams, digits after the decimal point appear to the right of the vertical bar). Figure 3.3b shows tilings corresponding to the irrational numbers π and $\sqrt{2}$.



(a) 5/6 and 1/11

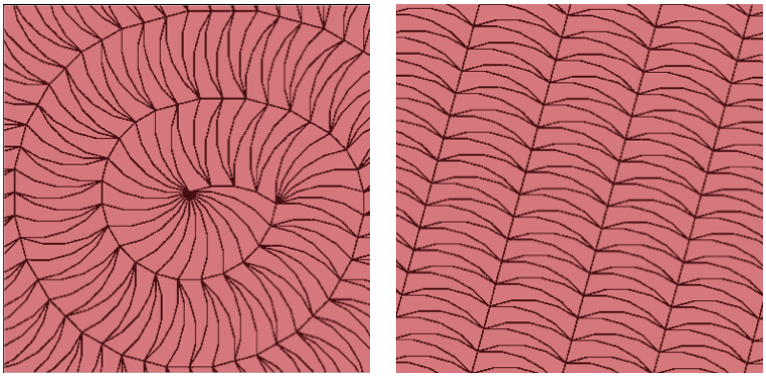
(b) π and $\sqrt{2}$

Figure 3.3: Real numbers and their corresponding one-dimensional tilings

The one-to-one correspondence between one-dimensional tilings and real numbers gives an interesting property: since there are countably many rational numbers and uncountably many irrational numbers, there are uncountably many tilings of the line that do not contain periodic sub-tilings. This is true for any tile set of size greater than one; unary encoding cannot represent real numbers.

3.2 Periodicity in Two Dimensions

Moving on to two dimensions, figure 3.4a shows a section of a spiral tiling, which is constructed using a single tile. Another way to think about non-periodicity is that a shifted copy of a tiling will never match the original exactly. Any tiling which forms a spiral pattern cannot be constructed by translations of a finite section[1], because it has a central point of rotational symmetry, and therefore a shifted copy of the whole tiling can never match the original. However, the same tile can be used to construct a periodic tiling, as shown in 3.4b.



(a) Non-periodic spiral (b) Periodic

Figure 3.4: Non-periodic and periodic tilings, using the same tile set[1]

A tile set is aperiodic just in case it can *only* form non-periodic tilings. For a time it was not known if aperiodic tile sets could exist. Wang dominoes are square tiles with coloured edges, first proposed by Hao Wang. They may not be rotated or reflected and must be placed such that adjacent domino edges have the same colour. Wang conjectured that any Wang domino set that can tile the plane non-periodically can also tile the plane periodically. The conjecture was disproved when an aperiodic set of 20,426 dominoes was found. Figure 3.5 shows a smaller aperiodic set of thirteen dominoes.

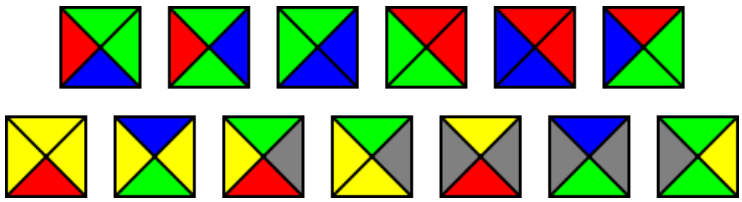


Figure 3.5: Aperiodic Wang domino set

As we shall see in chapter 4, it is possible to produce a set of Wang dominoes that, given an initial configuration, will simulate the behaviour of a Turing

machine. Since there are Turing machines that never halt and never fall into periods of computation, this forms the proof that there are aperiodic Wang domino sets.

The only general algorithm for tiling with any set of Wang dominoes involves extensive unbounded backtracking. Tiling a finite area has exponential time complexity, and there is no algorithm to determine if any given set of tiles can tile the infinite plane; it is an undecidable problem. In the worst case, every time we try to place a tile it will always be the last one we try. If we have a set of 13 tiles, as in the figure above, there are 13^{mn} ways of tiling a $m \times n$ area. However, there are much better algorithms for tiling the plane with specific tile sets, such as the Robinson tiles and the Penrose tiles.

3.3 The Robinson Tiles

The Robinson tiles are a set of 56 tiles. They consist of those seen in figure 3.6, as well as all possible rotations and reflections. The set can tile the plane non-periodically but not periodically. The edge-matching rules are usually represented by 'bumps' and 'humps'. Figure 3.7 shows the same tile set represented using arrows. A symmetrical bump or hump becomes a single arrow. Asymmetrical bumps or humps become double arrows, with the second arrow offset from the centre. Arrows point inwards for bumps and outwards for humps. The arrow representation ignores the shapes of the corners and will be used throughout this report.

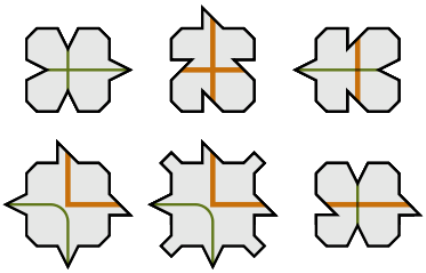


Figure 3.6: The Robinson tiles

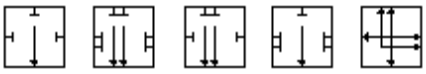
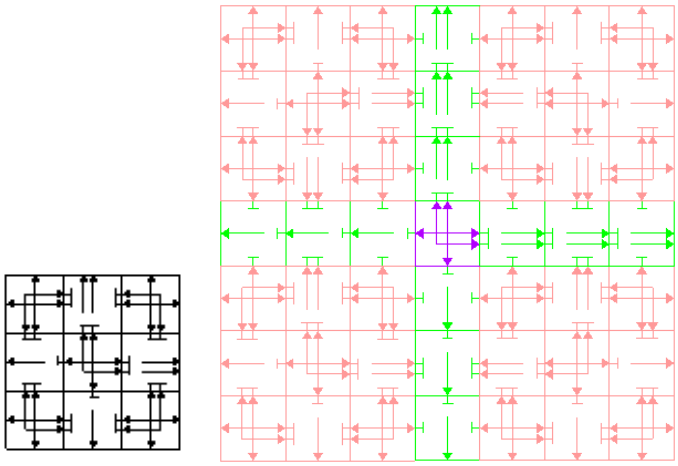


Figure 3.7: Alternative representation of the Robinson Tiles

The first four tiles in figure 3.7 have one outward-pointing arrow, and will be referred to as arm tiles. The last tile in the figure has four outward-pointing

arrows and will be referred to as a cross tile. The shape of the corners forces cross tile to appear in every tiling and appear infinitely often.[6]

Figure 3.8 shows what is known as a 3-square and a 7-square. In general these are named $(2^n - 1)$-squares, for n $\in \mathbb{N}$. Each of these structures is aperiodic, and each $(2^n - 1)$-square appears four times in each $(2^{n+1} - 1)$-square, once for each possible orientation.[6] The 7-square in the figure consists of four rotated 3-squares - shown in orange. The centre tile of the 7-square - shown in green - can then have one of four rotations, which then determines which tiles fill in the gaps - shown in purple.



(a) 3-square (b) 7-square

Figure 3.8: $(2^n - 1)$-squares

Since cross tiles must appear in every tiling, and each $(2^n - 1)$-square uniquely determines the location of a $(2^{n+1} - 1)$-square in the tiling, then for all n $\in \mathbb{N}$ there is a $(2^n - 1)$-square in the tiling. Since these structures are not periodic, all Robinson tilings are non-periodic.

3.4 The Penrose Tiles

The Penrose tiles are an aperiodic set of tiles first investigated by Roger Penrose. There are three variants. We will consider only the thick and thin rhomb variant shown in figure 3.9. The thick rhomb has internal angles of $2\pi/5$ *rad* and $3\pi/5$ *rad*. The thin rhomb has internal angles of $\pi/5$ *rad* and $4\pi/5$ *rad*. Edge matching rules are added to prevent periodic tiling. These are usually represented with notches or colours, as shown in the figure. When displaying full tilings, these edge restrictions are not usually displayed. The lengths of the sides of the two tiles are equal.

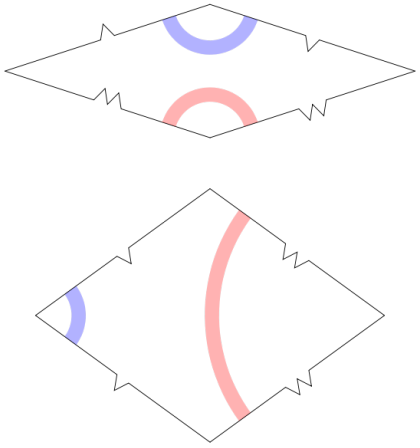


Figure 3.9: The thick and thin rhomb Penrose tiles

In general, it is undecidable whether a tile set is aperiodic; there is no algorithm to determine if any tile set is aperiodic. However, we can prove that some Penrose tilings are non-periodic. It is possible to produce Penrose tilings that have fivefold rotational symmetry, as shown in figure 3.10. Only tilings with two- four- and sixfold rotational symmetry can also have translational symmetry[2]. Tilings with fivefold rotational symmetry must have exactly one centre of rotation and so are non-periodic.

Further, we can prove that all Penrose tilings are non-periodic. As the size of a finite Penrose tiling increases, the ratio of thick to thin rhombs asymptotically approaches φ; in an infinite Penrose tiling, the thick and thin rhombs are in the golden ratio. Since φ is irrational, the tiling cannot be periodic. If it were, the tiling would consist of repetitions of a finite area. This finite area would have some number of thick rhombs, m, and some number of thin rhombs, n. The ratio of thick to thin rhombs would then be the rational number m/n[14].

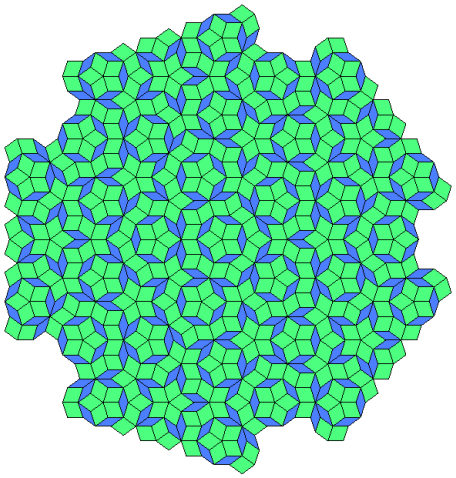


Figure 3.10: A penrose tiling with fivefold rotational symmetry

Penrose tilings have the following properties:

- There are uncountably infinitely many Penrose tilings;
- Any finite region that appears in a tiling appears infinitely often in every Penrose tiling. This property is clearly true of all periodic tilings, but is non-trivial for a non-periodic tiling;
- Because of the previous point, if we took a finite area of a Penrose tiling it would be impossible to determine where in the tiling we were looking and which tiling we were looking at; and
- They are quasicrystals[2].

Penrose tilings have no translational symmetry but are highly ordered and structured by a process known as inflation/deflation.

3.5 Inflation/Deflation

Both Robinson tilings and Penrose tilings can be generated by processes called inflation and deflation. How this is done for each set of tiles will be described further in chapter 5. For now, we will consider inflation and deflation in general and see a one-dimensional example. Inflation and deflation are transition functions from one tiling to another. That is, if we apply either process to an infinite tiling of the plane we will get another, different tiling. It is of interest in this project because both processes, when applied to a *finite* tiling, result in a larger tiling. This means the processes can be used to tile an arbitrarily large - but finite - area.

Suppose we have a set of two one-dimensional tiles. One is red and represented by the label R. The other is blue and represented by the label B. We then define a transition function that, when applied to an existing tiling, replaces every instance of R with the string RBB and every instance of B with the string BRR[11]. This is our inflation function. Starting with an R tile, figure 3.11 shows the result of applying the inflation rule one, two, and three times.

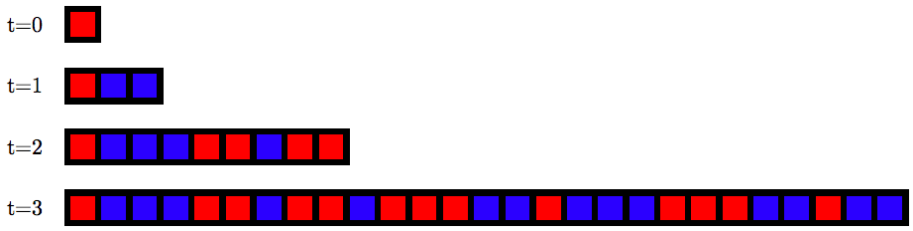


Figure 3.11: A single tile expanded to a 27-length tiling in 3 inflation steps

Each time we inflate, the length of the tiling increases by a factor of 3. We can apply the inflation rule any number of times, and the result will always be a larger, non-periodic tiling. As we shall see in chapter 5, a similar method can be used to generate arbitrarily large Robinson and Penrose tilings.

Chapter 4

Computability of Tile Sets

This chapter outlines how a set of tiles might simulate the behaviour of a Turing machine, and therefore constitute a general model of computation. A tiling that simulates a Turing machine is an instance of the origin constrained domino problem. The Turing machine has an initial state and an initial configuration of symbols on the tape. This means we already know which tiles belong on the first row, and must try to continue the tiling from that row.

The idea is to take a Turing machine, as described in section 2.2, and transform it into a set of tiles, where each tile represents one cell of the tape. The initial row of the tiling represents the initial configuration of the Turing machine. The tiles produced must force subsequent rows to represent subsequent configurations of the Turing machine; if a row represents the machine at time t, then the row above it must represent the same machine at time $t + 1$. Once we have this behaviour then, by induction, we can see that the tiling would represent the - possibly infinite - computation of the machine.

4.1 The Set of Tiles

All of the tiles we have seen so far use colours, arrows, or bumps to represent their edge matching rules. These representation methods are for clarity only, and are all equivalent. The Turing machine simulation tiles shown in this section use a combination of arrows and labels to represent edge matching rules. Arrows must match such that they flow across edges. An edge with no arrow head must be adjacent to another edge with no arrow head. Adjacent edges must also agree on their labels. The tiles shown can not be reflected or rotated. It is the labels at the top of the tiles that represent the current configuration of the machine.

A different set of tiles is needed to simulate each Turing machine. The sort of tiles we need are shown in figures 4.1 to 4.5. The first tiles needed are the initial configuration tiles, shown in figure 4.1. These are placed on the first row of the tiling to represent the initial configuration of the machine, including the machine's input. The arrows on the left and right edges of these tiles (with no

accompanying label) are to ensure that the initial configuration tiles can only appear on the initial row; all other tiles that have an arrow on the left or right edge have an accompanying label, and so can not appear on the same row as an initial configuration tile. The s part of the label represents the current symbol at that part of the tape. The q_0 part of the label represents the initial state. The initial row consists of an infinite number of these tiles extending to the right, and possibly to the left as well. Figure 4.2 shows such an initial configuration, starting in state q_0 with input 101 followed by an infinite number of blank cells extending to the right.

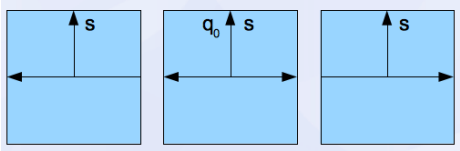


Figure 4.1: The initial configuration tiles

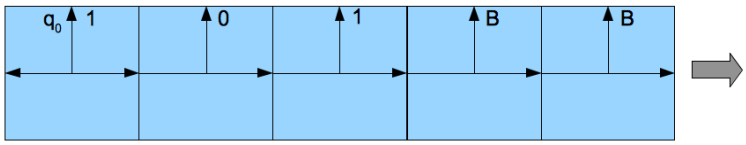


Figure 4.2: An initial configuration

The other tiles are constructed from the rules of the Turing machine. The simplest tile - and most common one in any tiling - is the alphabet tile, shown in figure 4.3. For each row, the alphabet tile is used for every cell of the tape except for the one currently being read by the head of the machine. The function of the alphabet tile is to take the symbol from the tile below and retransmit it to the tile above. This clearly represents a cell of the tape not changing from one time-step to the next. There must be one alphabet tile for each symbol of the Turing machine.

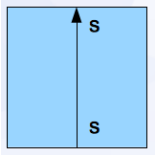


Figure 4.3: The alphabet tile

The remaining tiles represent the actions that take place at the read/write head of the machine. The possible actions are writing on to the tape, shown in

figure 4.4, and moving the head left or right, shown in 4.5. The write tile has a state label q and a symbol label s at the bottom, and a state label q_{next} and a symbol label s' at the top. The behaviour of this type of tile should be read as 'when in state q and reading symbol s, go to state q_{next} and write symbol s'.' Each write tile directly corresponds to a Turing machine rule.

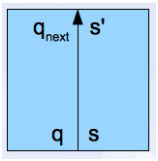


Figure 4.4: The 'write' action tile

The move actions involve moving the head to the left or the right. Each action requires two tiles. Figure 4.5 shows only the tiles required for moving the head to the right. The horizontal reflections of these tiles are also required. The first tile in the figure is an action tile. It takes a state q and a symbol s_0 from the tile below, then transmits the state q_{next} to the tile to the right and retransmits the symbol, unaltered, to the tile above. The second tile is a merge tile. It takes the state q_{next} from the left, then transmits it to the tile above, along with whatever symbol it takes from the tile below. The effect of these two tiles is to move the head one cell to the right and change the state, but leave the symbols of the two cells unaltered. Each pair of tiles corresponds to a Turing machine rule.

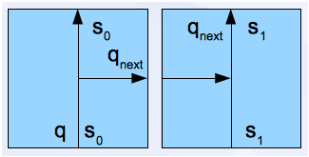


Figure 4.5: The 'left' and 'right' action tile

Every Turing machine description can be translated into a set tiles consisting of initial configuration, alphabet, write, action, and merge tiles. We introduce a blank tile, with no arrows or labels on any side, and use this tile to tile the half of the plane below the initial configuration row. Since the tiling represents the configuration of the machine over time, if the machine reaches a halting state or a stuck configuration the tiling will become stuck. Since the tiling can only be completed if the Turing machine will never halt, and the halting problem is undecidable, this proves the undecidability of the origin constrained domino problem[6]. It is also undecidable if the Turing machine, and therefore the tiling, become periodic.

Chapter 5

Design Overview

This chapter discusses the high-level design aspects of the project, including design methodologies, justifications of any software and platforms used, and the reasoning behind the main tiling algorithms. Implementation details are left until chapter 6.

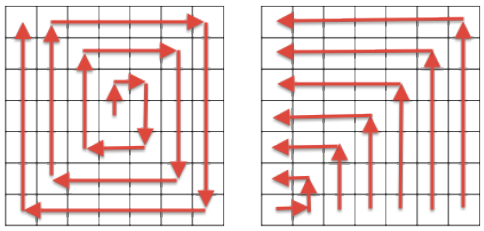
5.1 The Wang Domino Tiler

The purpose of the Wang domino tiler is to tile the plane with a given set of Wang tiles. It follows a simple trial and error algorithm with unbounded backtracking. The algorithm works as follows:

```
Choose a starting position;
loop
  Place a tile;
  if tile is in a legal position then
    Go to the next position;
  else
    if all tiles have been tried then
      Backtrack to the previous position;
    else
      Try a different tile;
    end if
  end if
end loop
```

The algorithm simply tries all tiles in all positions until it finds a configuration that is legal. When it finds a legal configuration it extends the area. Once a tile has been successfully placed, and the tiling is legal so far, the program moves on to the next position. There are several ways to determine where the next position is. The program must traverse the area in such a way that, given infinite time, it would traverse at least one quadrant of the infinite plane, visiting every tile location exactly once. Visiting one quadrant is sufficient because if it

is possible to tile the top-right quarter of the infinite plane, then this tiling can be translated an arbitrary amount to the bottom-left, tiling the whole plane. Figure 5.1 shows two methods of traversing the infinite plane. The first method places tiles in a spiral pattern extending outwards from some origin. The second method is the same but tiles only one quadrant. The Wang domino tiler uses the second method; as we shall see in chapter 6, this method was easier to implement.



(a) A spiral method of visiting every location in turn

(b) Another method, visiting locations in only one quadrant

Figure 5.1: Methods of visiting every tile location in the infinite plane

If the tiler reaches a stuck configuration, that is a configuration where it is impossible to place another tile, it traverses the plane using the same method *backwards* until it finds a position where an alternative tile could have been placed. The Wang domino tiler does not allow the user to manually build tilings. This is because for a set of t tiles and an area $m \times n$, there are t^{mn} possible ways of tiling the area. If, for example, only one of these configurations is legal, manually tiling would be impractical.

5.2 The Robinson Tiler

The purpose of the Robinson tiler is to tile the plane with the Robinson tiles automatically. It must also allow the user to manually interact with a tiling to extend it.

5.2.1 Automatic tiling

The automatic tiling algorithm takes advantage of the following facts:

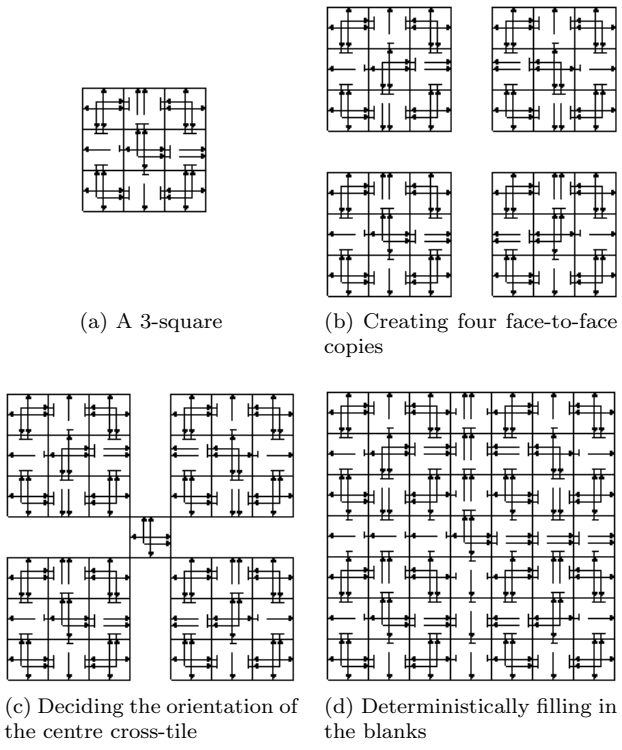
- each $(2^n - 1)$-square, including the single cross-tile, has one of four distinct orientations;
- the orientation of a $(2^n - 1)$-square uniquely determines the location of the $(2^{n+1} - 1)$-square it is located within;

- the $(2^{n+1}-1)$-square consists of four (2^n-1)-squares, one in each orientation, as well as a centre cross-tile. The remaining tiles can be deterministically filled in.

From these we can conceive of an inflation algorithm to tile arbitrarily large areas. The second loop of the algorithm is illustrated in figures 5.2b to 5.2d. The algorithm works in the following way:

Place a single cross-tile (1-square) in one of four orientations;
loop
Create three copies of the current (2^n-1)-square, in the three other orientations, and position these three copies such that the centre cross-tiles of them are face-to-face (figure 5.2b);
The orientation of the new $(2^{n+1}-1)$-square is now determined by the orientation of the centre cross-tile. Choose an orientation for this tile (figure 5.2c);
The remaining blank spaces are filled in by trial and error (figure 5.2d);
end loop

The last step of the loop, filling in the blank positions, is deterministic and no backtracking is required.



(a) A 3-square

(b) Creating four face-to-face copies

(c) Deciding the orientation of the centre cross-tile

(d) Deterministically filling in the blanks

Figure 5.2: A single inflation step from a 3-square to a 7-square

The inflation algorithm constructs relatively large tilings in few steps when compared to a trial-and-error backtracking algorithm. For large tilings, each iteration covers approximately four times the area of the previous iteration; the algorithm has exponential space- and time-complexity. After n iterations, the result is a non-periodic tiling that covers an area of $2^{2n} - 2^{n+1} + 1$.

5.2.2 Manual tiling

The Robinson tiler program must allow the user to interact with tilings and manually extend them. First, the user selects one of the five Robinson tiles and an orientation. Then, he clicks where he wishes the tile to be placed. If the tile placement is legal, then the tile will be added. If it is not a legal placement then the user will be notified and the tile will not be added. A tile placement is legal if and only if all four edges are either

A) adjacent to an agreeing edge; OR

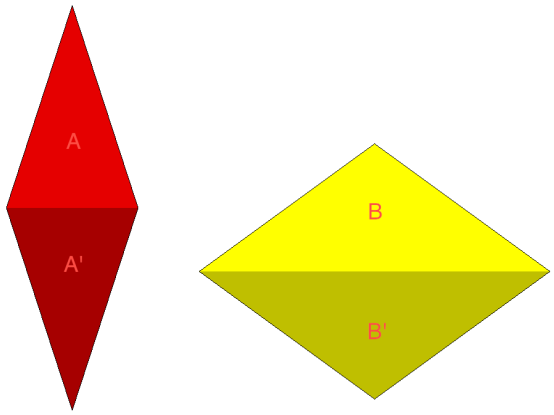
B) adjacent to an empty position

5.3 The Penrose Tiler

The purpose of the Penrose tiler is to automatically tile the plane with Penrose tiles, as well as allowing the user to manually construct tilings.

5.3.1 Automatic tiling

The algorithm for automatic tiling is based on a deflation method as described in section 3.5. This particular deflation function is based on an observation of the configurations that the tiles can appear in. For the purposes of deflation we will split the tiles in half. Figure 5.3 shows the tiles and the way in which they are split. The thick rhomb will be referred to as tiles B and B'. The thin rhomb will be referred to as tiles A and A'.



(a) The A and A' tiles

(b) The B and B' tiles

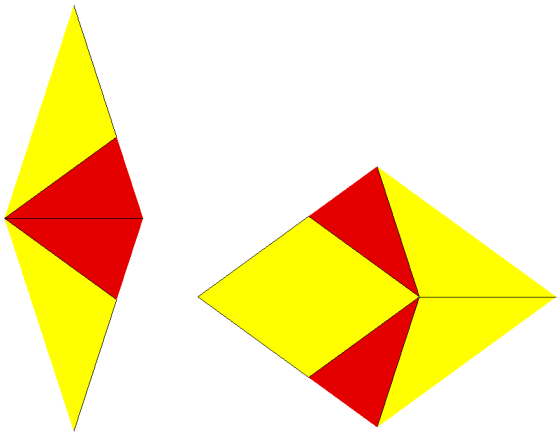
Figure 5.3: The thick and thin rhomb tiles, split in half

When we split the tiles in this way, we can see that in any tiling the half tiles always fall in one of two patterns. These two patterns are shown in figure 5.4. Figure 5.6 shows a tiling, with thick lines showing how the tiles fall into these two patterns. The first thing to notice is that, if the groups are considered to be tiles, they also form a Penrose tiling. From this observation, De Bruijn[9] formulated a method of constructing penrose tilings, which works as follows:

```
Start with a single tile;
loop
  for all tiles in tiling do
    if tile.type=A then
      Replace tile with configuration in figure 5.5a;
    else if tile.type=B then
      Replace tile with configuration in figure 5.5b;
    end if
  end for
end loop
```

Also note that the deflation function does not respect tile boundaries; a single tile, when deflated, becomes a selection of half tiles. This means that when using this method to construct a tiling, some tiles will be left incomplete around the edges, as can be seen around the edges of figure 5.6. After each iteration the program must complete these edges. The following must appear inside the main loop of the algorithm.

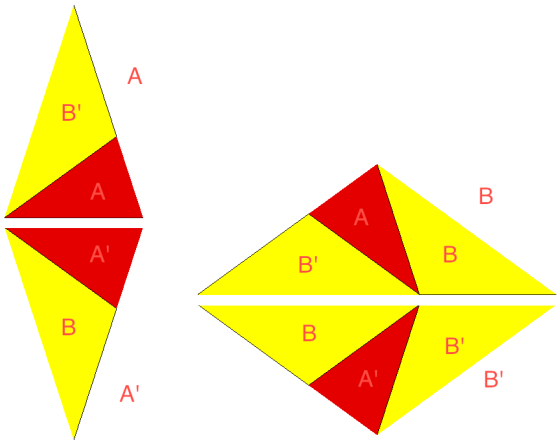
```
for all tiles in tiling do
  if tile is incomplete then
    complete the tile
  end if
end for
```



(a) Pattern 1, forming a larger A tile

(b) Pattern 2, forming a larger B tile

Figure 5.4: Half tiles must fall into one of these two patterns



(a) Deflation of an A tile

(b) Deflation of a B tile

Figure 5.5: The thick and thin rhombs, deflated

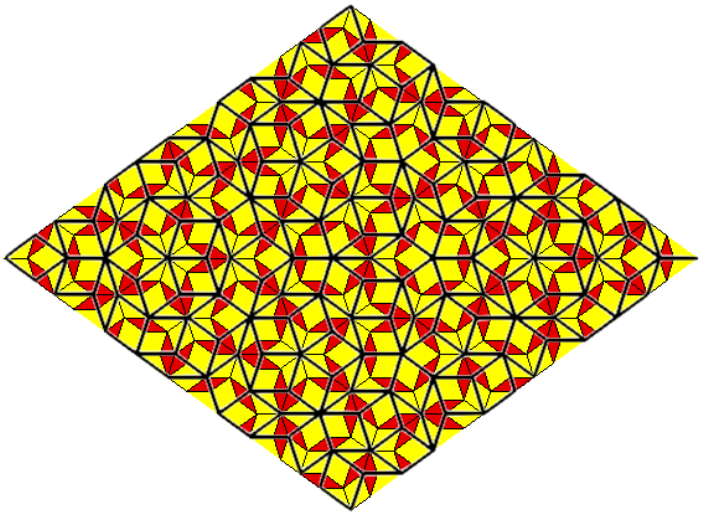


Figure 5.6: A tiling, overlaid with the outline of the tiling it was deflated from

At each iteration every A half-tile becomes two tiles, and every B half-tile becomes three tiles. For large tilings the number of tiles increases by a factor of approximately φ. Because the number of tiles increases exponentially in the number of iterations, the algorithm clearly has exponential time- and space-complexity. This method of tiling is not suitable for finding all possible tilings of a given finite area, because at each iteration the area covered increases. The process is deterministic and large tilings can be constructed relatively quickly when compared to a trial-and-error backtracking tiler.

5.3.2 Manual tiling

The manual tiler allows the user to interact with an existing tiling. First the user selects which of the two tiles to add, then he clicks where he wishes to add it to the tiling. If the position is legal then the tile will be added. The user should also be able to zoom and pan.

Unlike the Robinson tiles, the Penrose tiles do not appear in neat grids; in order to determine where the user wants the tile to be placed, it must be snapped to the nearest possible edge.

5.4 Tile Sets as Turing Machines

The Turing machine simulator has two parts. The first takes the description of a Turing machine and produces a tile set. The second takes that tile set with an input and produces a tiling. The structure of the program is shown in figure 5.7.

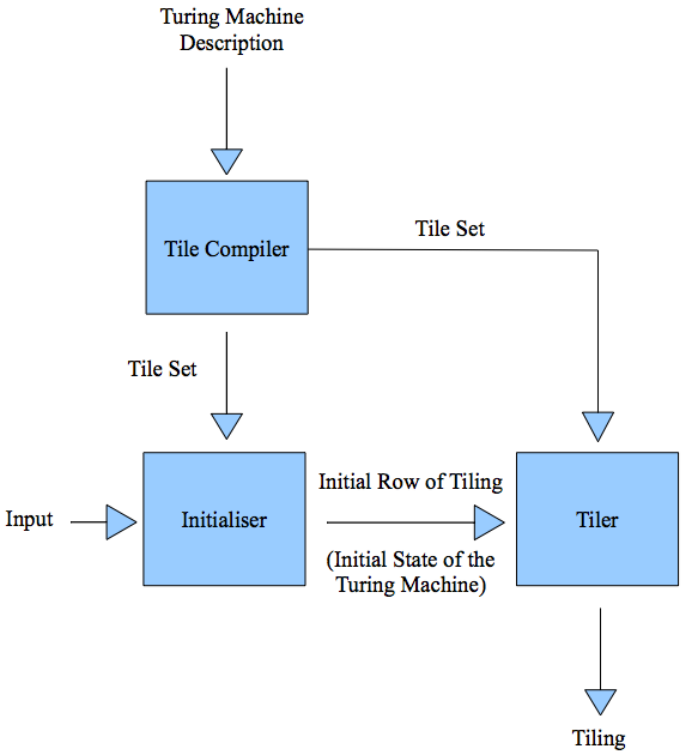


Figure 5.7: Structure of the Turing machine simulator

5.4.1 The tile compiler

The tile compiler program takes the description of a Turing machine and produces a set of tiles. The description takes consists of a set of symbols, a start state, a set of halting states, a set of actions, and a list of rules. Examples are shown in tables 5.1 and 5.2. The example Turing machine takes an input of zeros and ones and flips them. Blank symbols are unchanged.

Initial State	5
Halting States	{8}
Symbols	S={0, 1 ,BLANK}
Actions	{≪, ≫, S}

Table 5.1: The first part of the Turing machine description

State	Symbol	State'	Action
5	0	7	1
7	1	5	$\gg$
5	1	10	0
10	0	5	$\gg$
5	BLANK	8	BLANK

Table 5.2: The rest of the Turing machine description, a list of rules

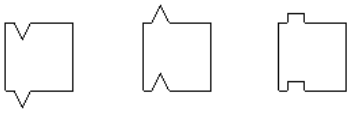
The tiles are constructed as described in section 4.1. The set of symbols from table 5.1 is used to construct a set of alphabet tiles and initial configuration tiles. There must be one alphabet tile for each symbol, with the symbol on the bottom and top edge, in order to transmit each symbol unchanged. There must be one initial configuration tile for each symbol, with the symbol on the top edge, representing the input of the Turing machine. There must also be one initial configuration tile for each symbol that also carries the starting state, both on the top edge, to represent the head of the machine at initialization along with the input at that cell. The initial configuration tiles have special symbols on the left and right edges. This is to ensure that only initial configuration tiles can appear on the first row and that those tiles cannot appear on any other row. It also ensures that only one cell in the initial row can carry a state with it.

Then, the rules in table 5.2 are used to construct tiles. Any rule whose action is to write onto the tape becomes an action tile, where the current state and current symbol are represented on the bottom edge of the tile and the next state and next symbol are represented on the top edge of the tile. A rule that moves the head to the left or the right becomes two tiles. The first tile has the current state and current symbol represented on the bottom edge, the unchanged symbol represented on the top edge, and the next state represented on the left or right edge. The second tile, which represents the position that the head of the machine is moving to, has the next state represented on its left or right edge, that cell's current symbol represented on its bottom edge, and the next state and the unchanged symbol represented on its top edge. This second tile is called a merge tile. Since the cell that the head is moving to could contain any of the symbols, for every rule that moves the head we must construct one merge tile for every possible symbol. The construction of merge tiles can lead to duplicates, which must be removed. The result of a move action tile and its corresponding merge tile is to move the head of the Turing machine from one cell to another, change the current state, and leave the symbols of both cells unchanged.

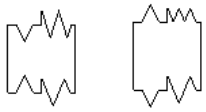
To illustrate, the tiles produced from the example Turing machine described by tables 5.1 and 5.2 are shown in figure 5.8. '1' and '0' symbols are represented by pointed humps or bumps, and the blank symbol is represented by a square bump. States are represented by a binary representation of the state number.

(a) Initial configuration tiles for start state

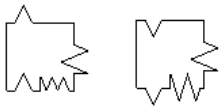
(b) The other initial configuration tiles



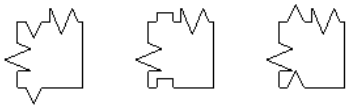
(c) Alphabet tiles, for retransmitting symbols



(d) Write tiles, for changing symbols



(e) Move tiles, for sending the state left or right



(f) Merge tiles, for receiving the state from a move tile

Figure 5.8: The tiles generated from the example Turing machine

5.4.2 The tiler

The purpose of the tiler part of the program is to take the tiles produced by the tile compiler, along with an input, and use them to construct an initial row of the tiling. The initial row represents the state of the initial configuration of the Turing machine. The program then tries to tile the plane above this initial

row. Conceptually, the rows are infinite in length.

The tiling algorithm works in much the same way as the naïve Wang domino tiler. Since the tiles are square there is a discrete grid of tile positions, each of which must be filled with a tile. The tiler must visit each of these positions in turn, trying to tile the plane in a way such that adjacent tiles have matching edges. It keeps a decision stack and if it reaches a stuck configuration, and it has tried all possible tiles in the current position, it will begin to backtrack. Like the Wang domino tiler, it doesn't matter how the algorithm traverses the plane; as long as every position is visited exactly once, the algorithm will produce a tiling, if possible. However, as each row represents the configuration of the machine at a given time step, and each configuration of the machine is determined by the row before, an obvious solution is for the algorithm to traverse the plane by completing each row in turn, as shown in figure 5.9.

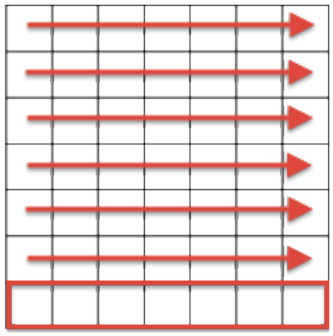


Figure 5.9: The Turing machine simulator tiles the plane one row at a time

The row highlighted in red represents the initial row of the tiling, which is already constructed before the traversal begins. The rows are infinite in length. Clearly, the program can only tile finite lengths. It must decide how much of each row to tile before it moves on to the next row. Conveniently, most of the tiling will consist of the alphabet sort of tiles seen in figure 5.8c. The only place the other sorts of tiles will appear - and trial-and-error tiling must be done - is where the head of the Turing machine is. A solution to the traversal problem is to decide on a finite length of row to tile, and if the head of the machine approaches the edge of the tiling area then the tiling can be extended horizontally.

If the original Turing machine is deterministic then the tiling algorithm will also be deterministic. For a deterministic tiling there is only one situation in which the algorithm will have to backtrack, and when it does it only needs to backtrack by one position. The algorithm may have to backtrack when the head of the Turing machine moves left. Where this happens a move tile of the sort shown in figure 5.8e is required, and a merge tile as shown in figure 5.8f must be placed to the left of it. However, the algorithm may have already placed an alphabet tile in that position. Because the backtracking is limited, rather the

extensive backtracking employed by the Wang domino tiler, the time complexity of this algorithm is much lower.

While the tile compiler part of this program needs information about the input Turing machine, the tiler part is mostly a generic tiler, working similarly to the Wang domino tiler; it is given the required tile set and it needs almost no information about the machine it is simulating. However, the tiler does need to know the set of accepting halting states of the machine. Without this information, the tiling algorithm would be unable to distinguish between a stuck configuration and an accepting configuration; the machine halts after an accepting configuration, so the tiling can not be completed. In this situation, the algorithm would backtrack until it had unraveled the whole tiling.

5.5 Platform, Software, and Design Methodology

The programs written for this project were written in C++, with the exception of the Penrose tiler which was written in C. I chose this programming language for the following reasons:

- memory management and dynamic memory allocation are relatively simple;
- I already had experience with the language;
- it is a cross-platform programming language;
- it interfaces with implementations of OpenGL, the cross-platform computer graphics API specification, easily; and
- it allows object-oriented programming.

I chose to use OpenGL because I was already familiar with the specification. OpenGL allows efficient rendering of 2D and 3D graphics. The tiling problems are inherently visual, and it is important for the programs to be able to quickly render thousands of tiles.

I chose to use an object-oriented language because, while designing the programs, the most obvious way to think about the tiles was as objects with their own attributes, properties and actions.

Java is a viable alternative to C++. Using Java would have made memory management easier, as it is done automatically. However, C++ generally performs the same tasks more quickly and performance was an important consideration. I also had more experience with C++, and had never used Java in conjunction with an implementation of OpenGL.

The Wang domino tiler and the Turing machine simulator make extensive use of an unbounded backtracking algorithm. Performance may have been improved by using Prolog for these parts. However, I had no experience with interfacing Prolog with C++ and chose not to.

The programs were designed using an iterative design methodology. Development began with the design of the essential features, followed by implementation, testing, and debugging. This process was then repeated for any additional features. The design phase was also revisited in order to improve the efficiency of the algorithms.

Where appropriate, such as with the Turing machine simulator, a bottom-up design methodology was also used; the two programs were written and tested separately before being interfaced together. Implementation of the GUI of each tiler program did not start until the implementation and testing of the program was finished.

Chapter 6

Implementation and Testing

This chapter describes the most important and most interesting implementation issues. It describes exactly how some of the algorithms were implemented and how the designs had to change because of practical considerations.

6.1 The Wang Domino Tiler

Wang dominoes are distinguished by the colours of their edges. The program stores tiles as objects, with each edge being represented by three integers, for red, green, and blue. The Wang dominoes are square, so they always appear in neat grids. This means that before we start tiling, while we do not know which tiles appear where, we do know the positions of the tiles. For this reason, the Wang Domino tiler keeps a two-dimensional data structure of tile objects. The indices of the tile in the data structure correspond to the tiles position in the plane. For example, the tile held at [0][5] in the data structure appears in the same column as the origin tile, and 5 units to the right of it. The tiler has a 'blank tile' object, which is a tile with special edge labels. The data structure is initialised as being filled with these blank tiles. The program stores the tiles in a two-dimensional vector (or, more accurately, a vector of vectors of tiles). The vectors are initialised at a certain length and are resized when they start to become full. I chose to use vectors because they can be extended from the back end in constant time. Extending vectors from the front end involves pushing objects in the vector along and is done in linear time. However, the program only tries to tile the up-right quadrant of the plane, and the tiling data structure will never have to be extended to the left or bottom.

The program traverses its data structure replacing the blank tiles with those from its tile set. The tile set is read in at runtime from a text file. The traversal method was outlined in the design chapter. As each tile is placed, the program checks if it 'agrees' with the surrounding tiles. A tile's agreement with its neighbours is decided by the following algorithm:

for all adjacent tiles **do**

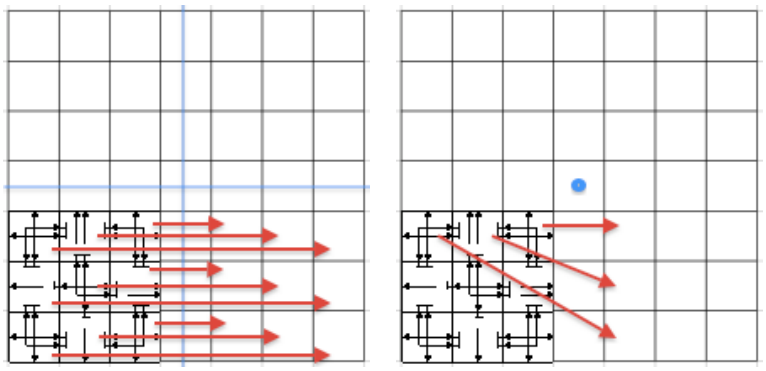
```
    if adjacent tile is not blank AND edges disagree then
      return FALSE;
    end if
  end for
  return TRUE;
```

In other words, the tile being considered must have matching edges with all its adjacent tiles that are not blank. When a valid placement is made, the program moves on to the next position. When an invalid placement is made, the program tries putting another tile in its place. If all choices have been exhausted for that position, the tiling backtracks to the first previous position where there is another choice that could have been made. If ever the program backtracks to the origin and all choices are exhausted in that position, then a tiling cannot be constructed. In order to have this backtracking behaviour, the program keeps a stack of decisions. A 'decision' consists of the position at which the tile was placed and the index of the tile that was placed there. When a new decision is made it is pushed onto the stack. When the program backtracks by one position, a decision is popped off the stack, and the index of the tile is incremented (the next tile is tried).

6.2 The Robinson Tiler

The most challenging part of the Robinson tiler to implement was the inflation method shown in figure 5.3. The first step of the inflation is to make three copies of the current tiling. There are two methods of deciding where copied tiles should go, one involving reflection and the other involving rotation. Whichever is used, the individual copied tiles must then be rotated or reflected into the correct orientation. Using the rotation method, one copy is rotated by 90°, one is rotated by 180°, and one is rotated by 270°. Using the reflection method, one copy is reflected in the x-axis, one is reflected in the y-axis, and one is reflected in both axes.

Figure 6.1a shows the method of placing new tiles based on reflections of the current tiling. The lines of symmetry are shown in blue and reflections in the x-axis are indicated with arrows.. Figure 6.1b shows the method of placing new tiles based on rotations of the current tiling. The centre of rotation is shown in blue, and some of the rotations are shown. I decided to use the rotation method. Whichever method is used, tiles are simply copied over to the new positions. The copied tiles will be in the wrong orientation and must also be rotated or reflected. There are two ways of doing this; either the edge labels can be manipulated to change the orientation of the model of the tile, or the program can keep track of how many times the tile has been rotated, and have OpenGL rotate the rendering of the tile. I opted for the latter method, and the tiles are rotated into the correct orientation because OpenGL does not have a reflect function.



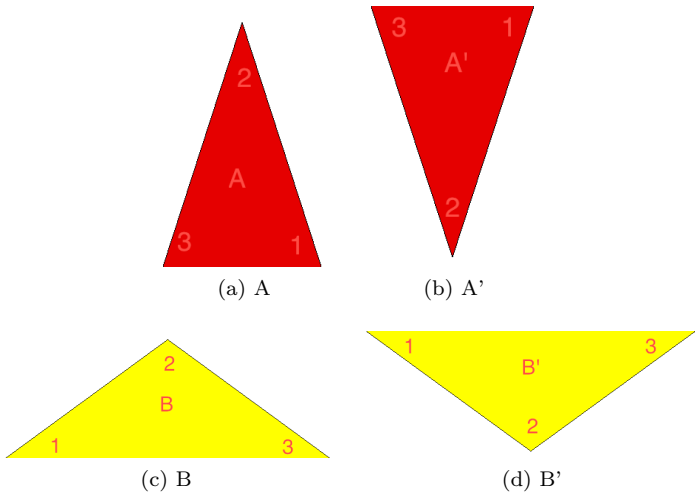
(a) Reflection method (b) Rotation method

Figure 6.1: Two methods of creating copies of Robinson $(2^n - 1)$-squares

6.3 The Penrose Tiler

6.3.1 Automatic tiling

The automatic Penrose tiler algorithm involves replacing every half-tile with a configuration of tiles. The program must deal with half-tiles as objects. Each half tile has a number of properties, including whether it is a thin or thick rhombus, which half of the rhombus it is, and the co-ordinates of the three vertices. The vertices are ordered consistently, as shown in figure 6.2.



(a) A (b) A'

(c) B (d) B'

Figure 6.2: The four half-tiles, and the ordering of their vertices

The half-tiles deflate as shown in figure 5.5. For example, a B tile (which we label *B_original* in the algorithm below) becomes an A tile, a B tile, and a B' tile. We can see from the figure how to determine the vertices of the new tiles, and the following outlines the algorithm for doing so:

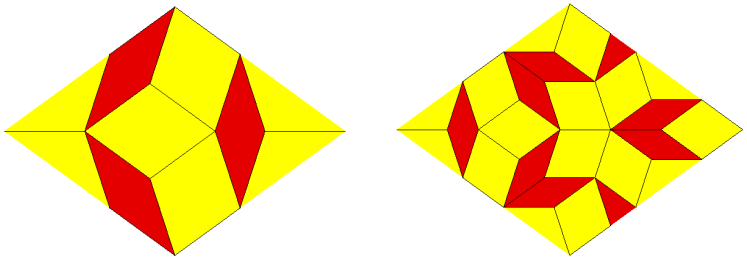
for all B half-tiles in tiling **do**
  initialize A, B, B';
  A.vertex1 := B_original.vertex1;
  A.vertex2 := point $\varphi/(\varphi+1)$ along B_original.vertex1 $\rightarrow$ B_original.vertex3 length;
  A.vertex3 := point $\varphi/(\varphi+1)$ along B_original.vertex1 $\rightarrow$ B_original.vertex2 length;
  B.vertex1 := B_original.vertex3;
  B.vertex2 := A.vertex2;
  B.vertex3 := B_original.vertex1;
  B'.vertex1 := A.vertex2;
  B'.vertex2 := A.vertex3;
  B'.vertex3 := B_original.vertex1;
  Remove B_original from data structure;
  Add A, B, B' to data structure;
end for

There is another, similar algorithm for deflating A type tiles. Each tile in turn is removed from the the tile data structure, and the tiles produced from the deflation are added to it. The data structure needed to be capable of dynamic memory allocation. The other tilers used square tiles, and so it made sense to use a two-dimensional array where the indices of the array indicate the position of the tile. The Penrose tiles do not have this property, so it stores the half-tiles in a dynamically-sized one-dimensional array.

This deflation method introduces incomplete tiles; around the edges, some half-tiles are placed without their corresponding half-tile, as shown in figure 6.3. These incomplete tiles introduce a problem for manual interaction; the user will not be able to extend the tiling if the edge-tiles are not whole. The following outlines an algorithm for completing these unfinished tiles:

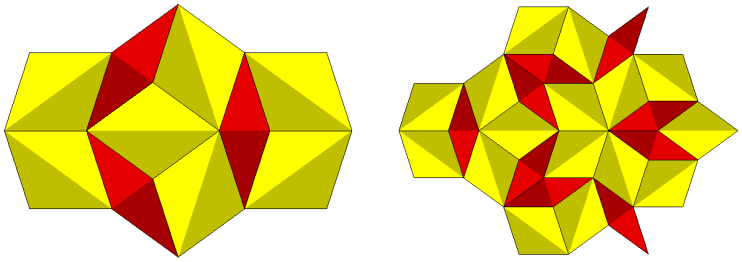
for all tiles (a) in tiling **do**
  for all tiles (b) in tiling **do**
    Check if tile a and tile b share vertices 1 and 3;
  end for
  if no tile shares vertices 1 and 3 with tile a **then**
    Introduce a new tile that completes tile a;
  end if
end for

This algorithm is applied to every generation of deflation. The new tile that is introduced shares vertices 1 and 3 with the tile that it completes. Figure 6.4 shows the same generations of deflation as figure 6.3, with this algorithm applied. In the figure, tile halves have been shaded so the original structure can be seen.



(a) Generation 2 (b) Generation 3

Figure 6.3: Generations of Penrose deflation, with incomplete tiles around the edges



(a) Generation 2 (b) Generation 3

Figure 6.4: Generations of Penrose deflation, with complete tiles around the edges

6.3.2 Manual tiler

The manual part of the tiler works by allowing the user to drag tiles - which actually consist of two half-tile objects - and drop them into place. In order for the user to be able to interact with the tiling in this way, there must be a way for the program to determine exactly where the user wants the tile to be placed. Since the tiles don't appear in a neat square grid this is a less trivial problem than it is for the Robinson tiler. When the user drags a tile to a location and clicks, the program compares the positions of the four edges of that tile to the positions of the four edges of every tile currently in the tiling. Whichever edge in the tiling is closest to an edge of the manually placed tile, while being within a certain threshold distance, the tile will be snapped to that position. Then, if the placement is legal - that is, if the edges match - then the tile will be added. The closest edge is the edge with the lowest sum of squares of the x and y distances between vertices. Figure 6.5 illustrates this process. The yellow tile represents a manually placed tile. The red circles represent the distance

thresholds. If there are no vertices within the threshold then the tile will not snap. The red arrows show which position the tile will snap to. The algorithm also checks if the user is trying to place a tile in a location where a tile already exists, by checking if there are any tiles that share all four vertices. Duplicate tiles will not be added.

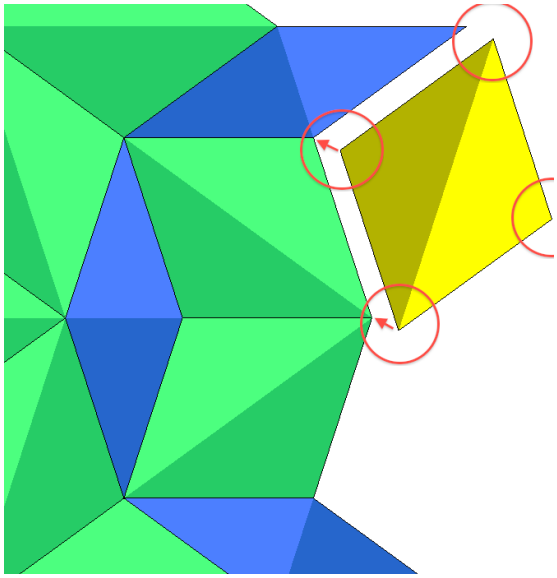


Figure 6.5: A manually placed Penrose tile snapping into position

6.4 The Turing Machine Simulator

6.4.1 Tile compiler

This part of the program takes a description of a Turing machine and converts it to a set of tiles. The first consideration is how the description is stored. The program reads in the description from a text file that contains:

- the start state;
- the set of accepting states; and
- the set of rules.

The tiles that are produced are as shown in figure 5.8. The program stores these tiles as objects with four labels vectors, each representing an edge. Each edge label can contain a symbol, a state, or a special symbol - such as the initial configuration tiles have on their sides. Each is stored as an integer. While the number of symbols and special symbols is defined in the start of the description file, the number of states can be arbitrarily increased. Since some of the tiles

can be thought of as transmitting states (such as the 'move' tiles), and some can be thought of as receiving states (such as the 'merge' tiles), there must be a distinction made between the two. In the program, this is done by making the receiving edge label the complement of the transmitting edge label. For example, if a move tile transmits a state number 5 to the right, the corresponding merge tile will have the receiving label -5. Two edges agree not if their edge labels are the same, but if their sum is 0. Figure 6.6 shows a sample tiling, with the state parts of the edges labelled.

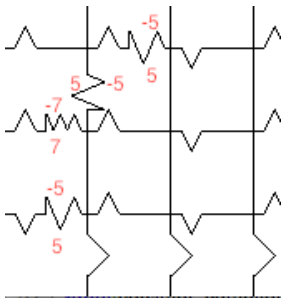


Figure 6.6: A partially labelled tiling

The states are stored in the text file in decimal form, but displayed in a binary representation. An outward bump represents a 1, and an inward bump represents a 0. Figure 6.7 shows several state numbers represented.

(a) 3 (b) 4 (c) 16

Figure 6.7: Binary representations of three states

6.4.2 Tiler

The tiler takes this tile set and tries to produce a tiling by trial and error. It tries all tiles in all positions. The tiler only moves on to the next position if all tiles so far are legally placed. Decisions are pushed onto a decision stack. If the program has to backtrack, then previous decisions are popped off the decision stack. If all possible combinations are exhausted then it is not possible to construct a tiling.

Since the tiles are roughly square, tilings take the form of a grid. The most appropriate data structure for storing tilings is a two-dimensional vector - actually a vector of vectors of tiles, as figure 6.8 represents. In the figure, the vertical columns represent vectors of tiles. The lower, horizontal row represent the vector of vectors. The blue arrows represent the extendability of the

structure. Note that extending the structure vertically involves extending each of the inner vectors, and extending the structure horizontally involves adding extra vectors onto the right. The indices of the vectors represent the physical location of the tiles, so the tile objects do not need to store location information, unlike the Penrose tiles. Vectors were chosen instead of arrays because they automatically resize if required when objects are pushed on to them. The vectors are set to some initial size. The vertical vectors must resize after some number of Turing machine computation steps. The horizontal vector must only resize when the head of the Turing machine approaches the edge of the tiling. The Turing machine cannot extend horizontally to the left - and so cannot represent a Turing machine with bi-infinite tape - because of the higher time complexity of extending a vector from the beginning, which would involve shifting all the other items in the vector along.

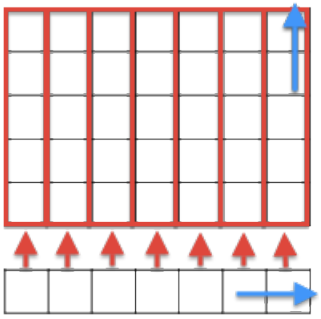


Figure 6.8: A representaion of a two-dimensional vector

Chapter 7

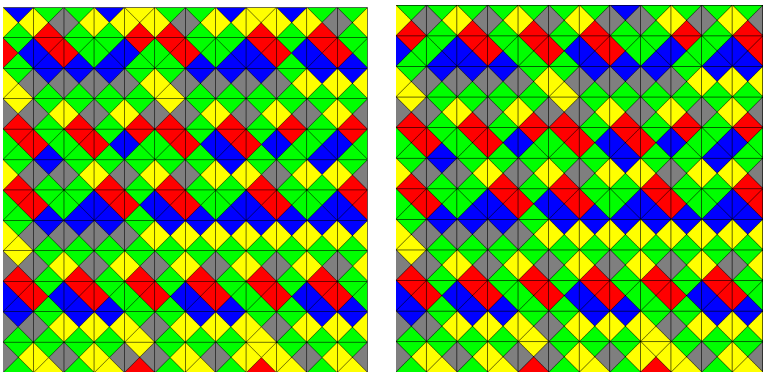
Evaluation of the Tools

In this chapter I will critically evaluate the programs produced for the project. I will do this by comparing the original purpose with the actual behaviour of each program. Known bugs in the programs will be discussed, along with possible solutions. This chapter also discusses features that would be added to the programs if there were more time available.

7.1 The Wang Domino Tiler

The Wang domino problem is conceptually very simple, and we can see that the tiling program models it correctly; it attempts all possible combinations of tile positions, trying to tile as large an area as possible while obeying the edge matching rules. An infinite run of the program on some tile set would backtrack to the origin with all options exhausted just when the tile set can not tile the plane.

However, due to the extremely large search space of possible configurations, even tiling a small area can take a long time. Figures 7.1a and 7.1b show the program tiling the plane using the tile set introduced in the chapter 5. The set can tile the plane, but only non-periodically. The first figure shows a 12x12 tiling produced after just a few seconds, and the second figure shows the configuration of the tiling several minutes later. Note that the program has been unable to extend the tiling beyond the 12x12 area. This is because for a 12x12 area there are 2.557×10^{160} possible configurations, while for a 13x13 area there are 1.805×10^{188} possible configurations, and it is possible that only a few of them are valid.



(a) Tiling after several seconds (b) Tiling after several minutes

Figure 7.1: Two valid configurations of a 12x12 area, using the same tile set

It would be useful if the Wang tiler program gave the user the option to store the current configuration of the tiling and the current decision stack in a text file. Then, the program could load that configuration later and continue the tiling. Having the option to save the current configuration would also be useful for the Turing machine simulator program. It would also be useful if the Wang tiler program could check for periodicity.

7.2 The Robinson Tiler

The Robinson tiler, while it meets the requirements, would benefit from a more intuitive user interface. When manually interacting, the user selects which tile they would like to place by dragging and dropping it from a bar at the top of the screen. A better interface might make use of a mouse scroll wheel, to scroll through the available tiles. Because the inflation process is non-deterministic - there are four possible directions a Robinson tiling can be inflated in - the user must decide the direction of inflation. It would be useful if the tiling program gave the user the option to undo previous inflation steps and change the orientation.

7.3 The Penrose Tiler

The automatic Penrose tiler works in two stages. First, it deflates every tile, increasing the size of the tiling. Second, it finds incomplete half-tiles, which are around the edge, and completes them. Figures 6.3 and 6.4 show two finite tilings before and after this second step is applied. This step is necessary to allow the user to manually add tiles, but it greatly increases the time complexity of the program. Table 7.1 shows a comparison of how long the program took to complete each iteration with and without performing the edge completion step.

The figures in the table are averaged from three runs of the program. Without edge completion, each iteration takes approximately φ^2 times as long as the previous iteration. This is because the number of tiles increases by the same factor at each iteration; the time complexity of the deflation algorithm is $O(n)$ in the number of tiles. With edge completion, each iteration takes approximately φ^4 times as long to complete as the previous iteration. Again, since the number of tiles increases by a factor of φ^2 every time the algorithm is applied, the time complexity of the deflation algorithm with edge completion is $O(n^2)$ in the number of tiles.

Iteration	Deflation only, time (ms)	Deflation and edge completion, time (ms)
1	0	0
2	0	0
3	0	0
4	0	2
5	0	20
6	1	150
7	2	1095
8	4	7871
9	12	57844
10	32	412698
11	93	-
12	160	-
13	374	-
14	969	-

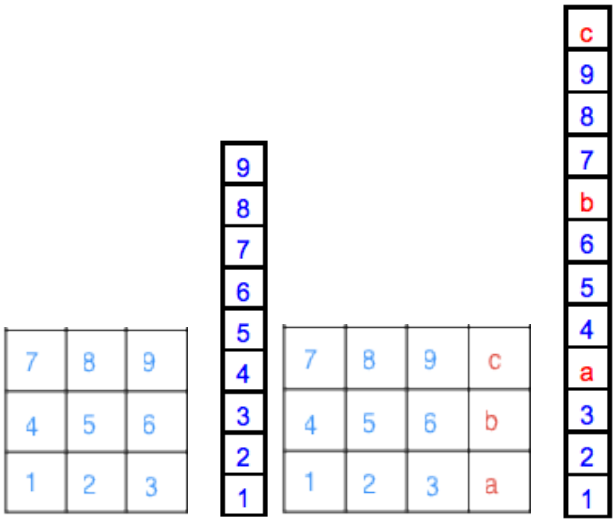
Table 7.1: Comparison of time taken to complete each iteration with edge completion turned off and on

Note that by the 10th iteration the deflation algorithm alone takes 32 milliseconds, whereas the deflation algorithm followed by the edge completion algorithm takes almost 7 minutes. The reason the edge completion step takes so long is that it compares every half-tile to every other half-tile, and checks if they form a whole tile together. The time complexity of the algorithm could be greatly reduced by changing the data structure used to store the tiling. The program currently stores the tiles in a one-dimensional array. If this were changed to a data structure where an element's position in the structure were indicative of its position in the tiling, then the edge completion algorithm could be made to run much more quickly. This is because two half-tiles can only form a whole tile if they are adjacent to one another, and each Penrose tile can only ever be adjacent to at most four other tiles. If the algorithm could infer which tiles are spatially close to each other, this would essentially reduce the time complexity of edge completion to $O(n)$ in the number of tiles; for n tiles, the algorithm would perform $4n$ comparisons, not n^2. One data structure that could be used is a kd-tree, where each split in the tree partitions the 2D space, and only the

leaf nodes would hold tile objects. The edge-matching algorithm would then traverse only the parts of the kd-tree that are spatially close to the starting tile.

7.4 The Tile Compiler

If the tile representing the head of the Turing machine moves towards the right of the tiling area, then the tiling must be extended horizontally. To do so is simple; The vectors must be resized, and the new area must be filled in with blank alphabet tiles, as described in chapter 5. However, the program keeps a decision stack, to keep track of which configurations have been tried so far. The stack has one decision entry for every tile currently in the tiling. When the area is extended horizontally, and alphabet tiles are used to fill in the new area, the decision stack must be modified to reflect the change. Elements in the stack can only be accessed from the end, and new entries cannot be placed in the middle, so restructuring it involves popping every entry off the stack, and then reconstructing it while inserting extra decision entries. Figures 7.2a and 7.2b represent a decision stack and a corresponding 3x3 tiling area. Each entry in the decision stack stores the tile currently placed at the position in the tiling with the same number. Figures 7.2c and 7.2d represent the same tiling, now 3x4, after being horizontally resized. It also shows the corresponding decision stack, and how it must be reconstructed.



(a) 3x3 tiling (b) (c) 3x4 tiling, resized (d)

Figure 7.2: A tiling and its decision stack, before and after horizontal resizing

Because the decision stack has to be rebuilt every time, the time complexity of the horizontal resize algorithm is linear in the number of tiles in the tiling.

A linked list would be a more appropriate data structure; push and pop actions can be added to make it function like a stack when required, and when the tiling is resized new decision entries can be added in the middle by changing the *next* and *previous* pointers of the appropriate nodes, as shown in figure 7.3. This would still involve visiting every element in the data structure every time there is a horizontal resize, but would be less computationally expensive than having to deconstruct and reconstruct the stack.

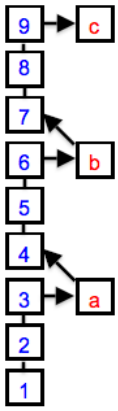


Figure 7.3: A linked list of decisions

Chapter 8

Summary and Future Work

Over the course of this report we have seen: a program that attempts to tile the plane with a given set of Wang dominoes, programs that can construct Robinson and Penrose tilings, and a program that, when given the description of a Turing machine, can produce tiles a set of tiles, tilings of which simulate the behaviour of that Turing machine.

The remainder of this chapter outlines possible extensions of the programs. All of the methods used so far to generate tilings do so by trial and error or by using a deflation or inflation function. Each method starts with a tile set, and incrementally covers a larger and larger finite area with them. Using such methods, no matter how large a finite area is covered, we can't say whether the tiling can be extended to fill the infinite plane, or whether it can do so non-periodically. This section outlines a method by which we can, with a small set of parameters, describe an infinite tiling in its entirety[4]. Given more time, producing a program that uses this method would have been the next logical step of the project.

The method involves taking an n-dimensional orthogonal projection, or shadow, of an integer lattice of more than n dimensions. First we will look at a 1D example. We start with a 2D plane, and consider all points on the plane with integer x and y coordinates. Joining these points gives us a two dimensional integer lattice, which consists of a grid of squares. We then choose any line that lies in this plane, and call it E. We then consider the area around line E, such that the area would be just wide enough to contain one (unrotated) square from the integer lattice. We name this area C. Selecting all edges that lie entirely within area C gives us a 'staircase' pattern through the plane. This is shown in figure 8.1. The edges highlighted in blue are those that fall within the yellow area C.

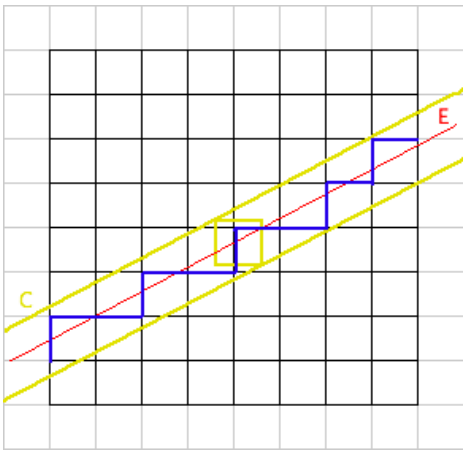


Figure 8.1: A line intersecting a 2D lattice

If we orthogonally project the edges in this staircase onto the original line E, the result is a tiling of the 1D line. Note that the structure of the tiling depends on the angle of the line E. With this method, instead of starting with a set of tiles and attempting to form a tiling with them, we are guaranteed to form a tiling and can ensure we get a non-periodic tiling. If the angle of the line E is rational then the ratio of horizontal lines to vertical lines crossed by the infinite line will also be rational, and the tiling will be periodic. If the angle of E is irrational, then the tiling will be non-periodic.

This idea can be extended to produce 2D tilings in the following way. We consider an integer lattice with more than two dimensions. For example, a 3D lattice consists of the edges joining points with integer x, y and z values in an infinite volume. This lattice would form unit cubes. We then take any plane E in the volume, and consider the volume C around the plane such that one (unrotated) cube from the lattice could fit inside the volume. If we select all edges of the lattice that fall wholly within C, and orthogonally project them onto E, the result is a two dimensional tiling. The tiling can be periodic or non-periodic depending on the orientation of the original plane.

The method can also be used to produce Penrose tilings. It has been shown that all Penrose tilings are 2D projections of a 5D integer lattice[8]. The lattice consists of edges joining integer points in five dimensional space. The lattice forms a grid of penteracts, which are the 5D analog of the cube. The method works as above; we take a plane E, and consider the 5D space C around the plane such that one of the penteracts could fit within it. The orientation of the plane E can be chosen such that the edges that fall within the space C, when projected onto E, form a Penrose tiling. The method can also be used to produce 3D tilings, or tilings in higher dimensions.

It would have been interesting to see if it is possible to simulate Turing machines using Penrose tiles, or a modification of Penrose tiles. It is not obvious if this is possible. In this project, the tiles that have been used so far to simulate

the behaviour of Turing machines have been roughly square in shape, with the particular shape of the edges corresponding to state and symbol information. Because of the square shape of the tiles, they fall into rows and columns, where each column represents a particular cell on the tape of the machine and each row represents the configuration of the machine at a single time step. The Penrose tiles are not square, and so do not have these properties. The tiles have particular edge matching rules to ensure they can only form non-periodic tiles. It may be possible to produce a modified set of Penrose tiles corresponding to a given Turing machine, such that they can produce a non-periodic tiling just when the machine never halts or falls into periods of computation, produces a periodic tiling just when the machine does fall into periods of computation, or fails to produce a tiling just when the machine halts. Had I had more time with this project, I would have determined whether it was possible to produce such a tile set.

Bibliography

[1] AMS feature column - Penrose tiles talk across miles, August 2005. http://www.ams.org/featurecolumn/archive/penrose.html.

[2] Crystallographic restriction, November 2009. http://www.mathpages.com/HOME/kmath547/kmath547.htm.

[3] Non-periodic tilings with n-fold symmetry, November 2009. http://www.mathpages.com/HOME/kmath539/kmath539.htm.

[4] About Quasitiler 3.0, May 2010. http://www.geom.uiuc.edu/apps/quasitiler/.

[5] David Austin. Up and down the tiles. *Notices of the AMS*, 52(6):600–601, 2005.

[6] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag Berlin Heidelberg, 1997.

[7] Nigel Cutland. *Computability*. Cambridge University Press, 1980.

[8] N. G. de Bruijn. Algebraic theory of Penrose's non-periodic tilings of the plane, I, II. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen Series A*, 84(1):39–52,53–66, 1981.

[9] N. G. de Bruijn. Updown generation of Penrose tilings. *Indagationes Mathematicae, New Series 1(2)*, pages 201–219, 1990.

[10] Martin Gardner. *Penrose Tiles to Trapdoor Ciphers, revised edition*. The Mathematical Association of America, 1997.

[11] Andrew Glassner. Penrose Tiling. *Computer Graphics and Applications, IEEE*, 18(4):78–86, 1998.

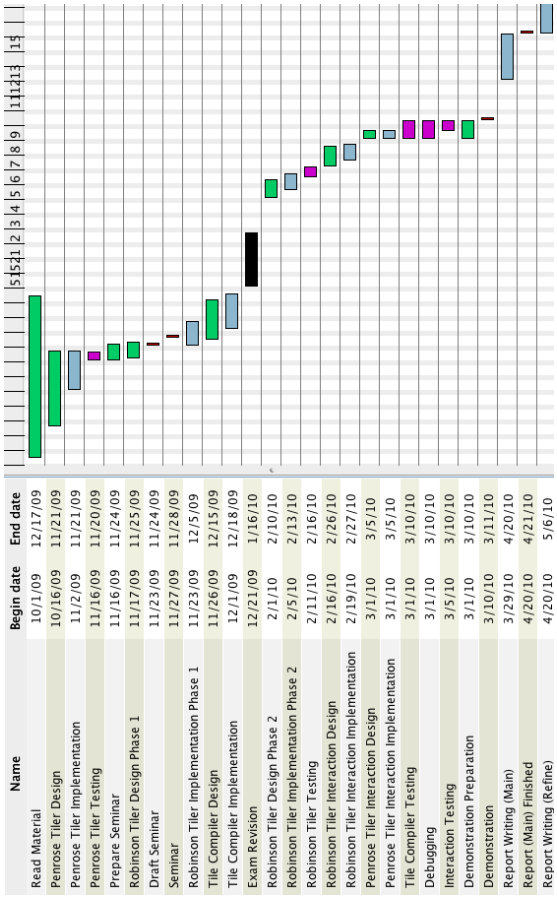
[12] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., 1997.

[13] Dexter C. Kozen. *Theory of Computation*. Springer-Verlag London Limited, 2006.

[14] Roger Penrose. Pentaplexity - a class of non-periodic tilings of the plane. *The Mathematical Intelligencer*, 2(1):32–37, 1979.

Appendix A

Project Plan



Name	Begin date	End date
Read Material	10/1/09	12/17/09
Penrose Tiler Design	10/16/09	11/21/09
Penrose Tiler Implementation	11/2/09	11/21/09
Penrose Tiler Testing	11/16/09	11/20/09
Prepare Seminar	11/16/09	11/24/09
Robinson Tiler Design Phase 1	11/17/09	11/25/09
Draft Seminar	11/23/09	11/24/09
Seminar	11/27/09	11/28/09
Robinson Tiler Implementation Phase 1	11/23/09	12/5/09
Tile Compiler Design	11/26/09	12/15/09
Tile Compiler Implementation	12/1/09	12/18/09
Exam Revision	12/21/09	1/16/10
Robinson Tiler Design Phase 2	2/1/10	2/10/10
Robinson Tiler Implementation Phase 2	2/5/10	2/13/10
Robinson Tiler Testing	2/11/10	2/16/10
Robinson Tiler Interaction Design	2/16/10	2/26/10
Robinson Tiler Interaction Implementation	2/19/10	2/27/10
Penrose Tiler Interaction Design	3/1/10	3/5/10
Penrose Tiler Interaction Implementation	3/1/10	3/5/10
Tile Compiler Testing	3/1/10	3/10/10
Debugging	3/1/10	3/10/10
Interaction Testing	3/5/10	3/10/10
Demonstration Preparation	3/1/10	3/10/10
Demonstration	3/10/10	3/11/10
Report Writing (Main)	3/29/10	4/20/10
Report (Main) Finished	4/20/10	4/21/10
Report Writing (Refine)	4/20/10	5/6/10

Figure A.1: Project Gantt chart

Appendix B

Demonstrations

B.1 The Penrose Tiler

Figure B.1 shows a Penrose tiling produced by 8 deflation steps of an A-type tile, with the edge completion algorithm turned on.

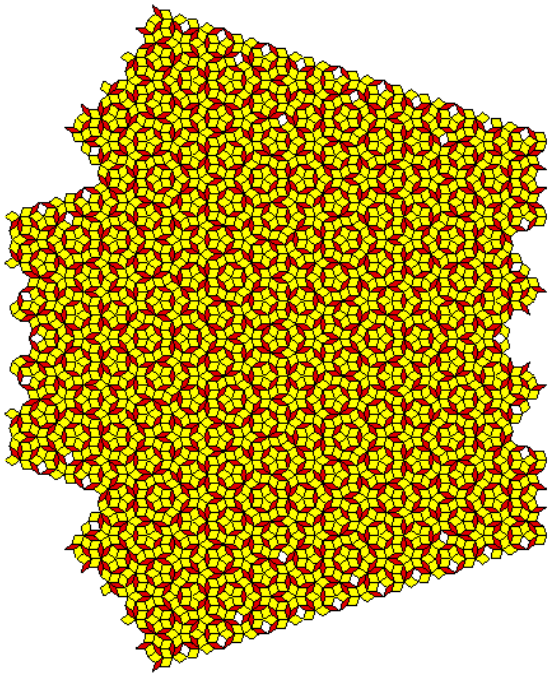


Figure B.1: A Penrose tiling

B.2 The Turing Machine Simulator

The following is a description of a simple Turing machine that takes an input of zeroes and ones and determines if the input string is a palindrome.

Initial State	0
Halting States	{16, 17}
Symbols	S={0, 1 ,BLANK, $\vdash$}
Actions	{$\ll$, $\gg$, S}

Table B.1: The first part of the Turing machine description

State	Symbol	State'	Action
0	1	BLANK	$\vdash$
0	BLANK	1	$\vdash$
1	$\vdash$	2	$\gg$
2	1	3	BLANK
2	0	4	BLANK
2	BLANK	8	$\ll$
3	BLANK	15	$\gg$
15	1	15	$\gg$
15	0	15	$\gg$
15	BLANK	5	$\ll$
4	BLANK	6	$\gg$
6	1	6	$\gg$
6	0	6	$\gg$
6	BLANK	7	$\ll$
5	1	11	BLANK
5	0	12	BLANK
5	BLANK	8	$\ll$
11	BLANK	13	$\ll$
12	BLANK	14	$\ll$
7	1	12	BLANK
7	0	11	BLANK
7	BLANK	8	$\ll$
8	BLANK	8	$\ll$
8	$\vdash$	9	$\gg$
9	BLANK	17	1
13	1	13	$\ll$
13	0	13	$\ll$
13	BLANK	2	$\gg$
14	1	14	$\ll$
14	0	14	$\ll$
14	BLANK	14	$\ll$
14	$\vdash$	10	$\gg$
10	BLANK	16	0

Table B.2: The rest of the Turing machine description, a list of rules

The Turing machine simulator, when given this description, produced the following set of tiles:

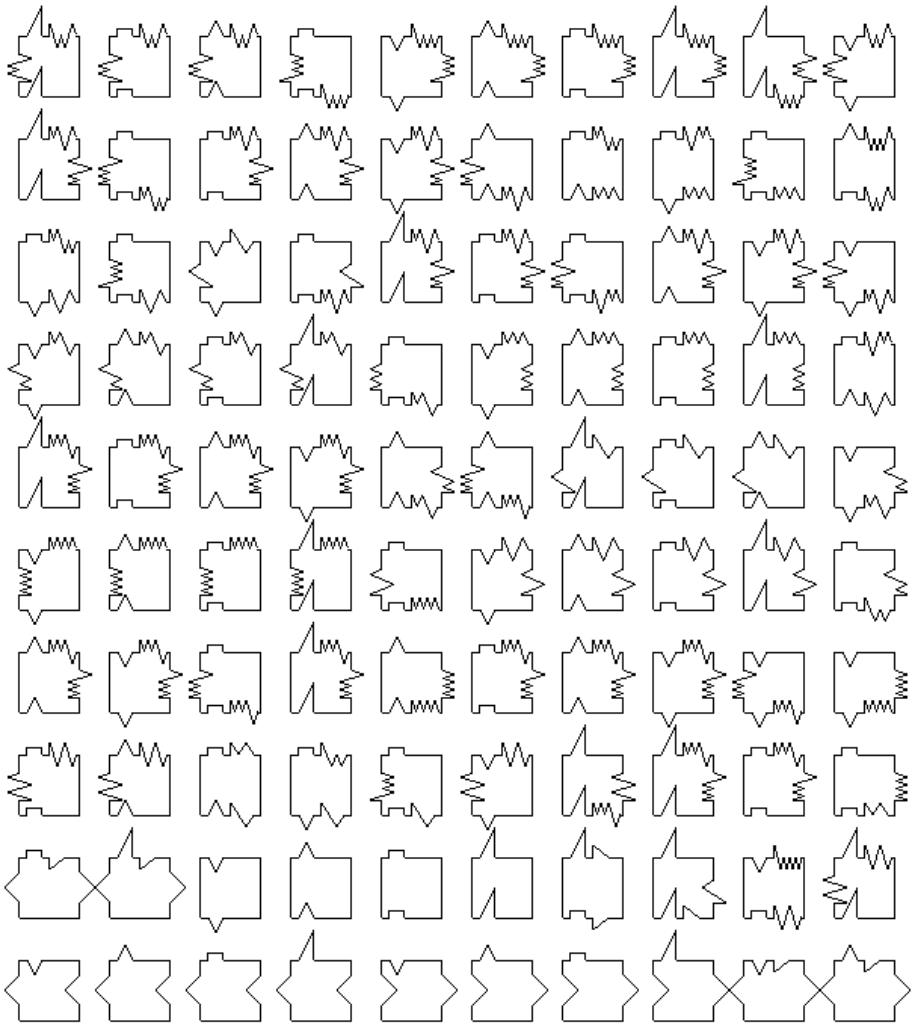


Figure B.2: 100 palindrome checker tiles

The tiler program is then given the above tile set and the input 10101, which is a palindrome. The tiler reaches a stuck configuration, a section of which is shown in figure B.3. In the figure, the last line of the tiling - and thus the final configuration of the Turing machine - is highlighted in green. In the last step, the head of the machine is in state 16, which is an accepting state of the machine, and the symbol at the head of the machine is 1, indicating that the input was a palindrome.

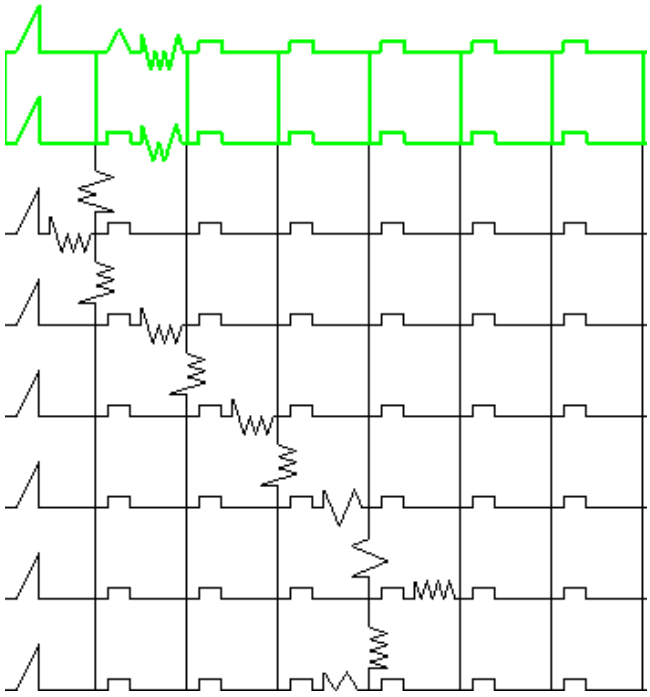


Figure B.3: A section of the tiling produced on input 10101

The tiler program is then given the same tile set and the input 10010, which is not a palindrome. The tiler reaches a stuck configuration, a section of which is shown in figure B.4. In the last step, the head of the machine is in state 17, which is an accepting state of the machine, and the symbol at the head of the machine is 0, indicating that the input was not a palindrome.

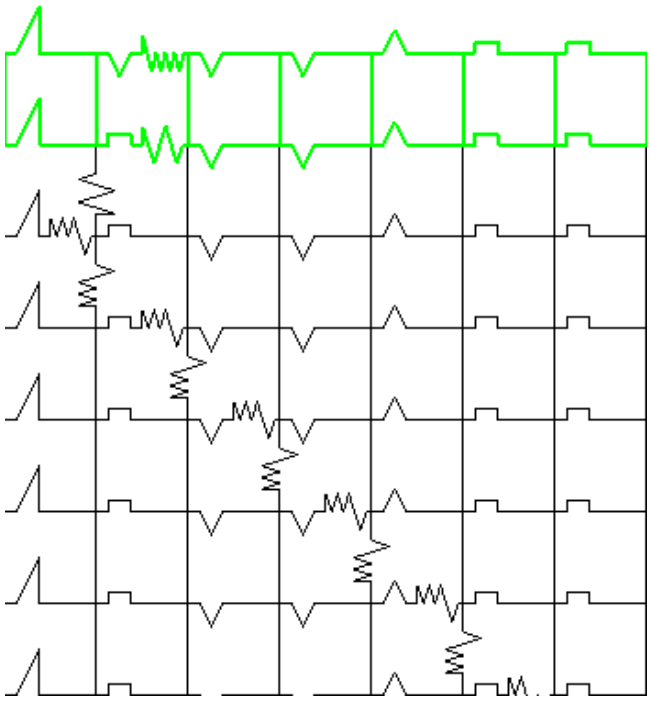


Figure B.4: A section of the tiling produced on input 10010