



**DUBLIN INSTITUTE
of TECHNOLOGY**

Institiúid Teicneolaíochta Bhaile Átha Cliath

**Development of methods for conversion
of images into 2D worlds and their
efficient exploration using Distributed
Artificial Intelligence**

DT228

BSc in Computer Science

Viacheslav Filonenko

D03102632

Ciaran O'Leary

Abstract

This project seeks to develop software that creates complex environments from images and lets a swarm of agents distribute its forces in an attempt to explore all possible routes and reach a destination. Unlike most maze solving programs, this one doesn't provide any limitation on the format, style and shape of the environment and also makes the process of solving the maze with agents dynamic, perspicuous and fun by utilising such techniques as steering and distributed intelligence. Problems such as converting lines of pixels into line coordinates and identifying junctions in a nonuniform system of tunnels are addressed in this project.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Student Name

2006

Acknowledgements

I am grateful for all the help I have received whilst researching, coding and writing up this project.

- My thanks most especially go to Ciaran O’Leary without whose supervision and help I wouldn’t have been enthusiastic enough to do half of what has been done.
- Also thanks go to Bryan Duggan for helping me understand how vector maths could be used for steering
- To Brian MacNamee for some interesting ideas and help in image processing.
- I would also like to thank Damian Gordon for support and reading through my draft copies.
- Finally I am very thankful to my parents for doing everything possible to allow me work on this project during the Christmas Break.

Table of Contents

ABSTRACT.....	I
ACKNOWLEDGEMENTS	III
1. INTRODUCTION	1
1.1 GOALS.....	1
1.2 MAZES.....	1
1.3 CONTRIBUTIONS AND ACHIEVEMENTS	2
1.4 LESSONS LEARNED	3
2. BACKGROUND ON MAZES	4
2.1 INTRODUCTION.....	4
2.2 TYPES OF MAZES	4
2.3 EXAMPLES	5
3. BACKGROUND ON AUTONOMOUS AGENTS.....	8
3.1 INTRODUCTION.....	8
3.2 AUTONOMOUS AGENTS.....	8
4. DISCUSSION OF SOFTWARE METHODOLOGY	10
4.1 INTRODUCTION.....	10
5. DISCUSSION OF TECHNICAL ARCHITECTURE	11
5.1 INTRODUCTION.....	11
5.2 COMPARISON OF C++ TO JAVA	11
6. ENVIRONMENT STRUCTURE DECISION	14
6.1 INTRODUCTION.....	14
6.2 THE MATRIX	14
6.3 VECTORIZATION	15
6.4 CONCLUSION	15
7. DISCUSSION OF FEATURE IMPLEMENTATION.....	16
7.1 INTRODUCTION.....	16
7.2 CONVERSION ACCURACY BALANCE.....	16
7.3 NAVIGATION ALGORITHM.....	17
7.4 CONCLUSIONS	28
8. DEVELOPMENT PROCESS	29
8.1 THE SOFTWARE DEVELOPMENT PROCESS	29
9. PROGRAM INTERFACE	30
9.1 INTERFACE CONSIDERATIONS	30
10. DEVELOPMENT OF IMAGE CONVERSION	32
10.1 THE IMAGE CONVERSION PROCESS	32
10.2 PSEUDOCODE	32
10.3 LINE REDUCTION	33
10.4 TYPES OF LINES	34
11. CONVERSION ALGORITHMS.....	36
11.1 INTRODUCTION.....	36
11.2 THRESHOLDING.....	36
11.3 FLOODFILL.....	36
11.4 SIMPLE EDGE DETECTION.....	38

11.5 SEAMSTRESS ALGORITHM.....	39
11.6 REFINING THE ALGORITHM.....	41
12. EFFECTS OF CONVERSION APPROACHES.....	45
12.1 BENCHMARKING	45
12.2 ANALYSIS.....	47
13. STEERING	48
13.1 INTRODUCTION.....	48
13.2 SWARM DATA.....	48
13.3 SWARM SENSORS.....	49
14. NODES.....	54
14.1 INTRODUCTION.....	54
14.1 IMPLEMENTATION OF NODES	54
15. CONCLUSIONS AND FUTURE WORK.....	58
15.1 CONCLUSIONS	58
15.2 FUTURE WORK.....	58
APPENDIX.....	59
SOURCE CODE.....	59
REFERENCES.....	96

Table of Figures

Figure 2.1. Orthogonal maze	5
Figure 2.3. Theta maze	6
Figure 2.4. An alternative kind of maze	7
Figure 4.1. Comparison of traditional and RAD models	10
Figure 9.1. Program interface	31
Figure 12.1. The result of conversion done on the sample maze.	47
Figure 13.1. Final sensors layout	49
Figure 13.2. Only side sensors stir.	50
Figure 13.3. One of the front sensors contributes.	51
Figure 13.4. Only one sensor contributes to the change of direction.	51
Figure 13.5. No obstacles nearby.	52
Figure 13.6. Swarm in a tunnel.	52
Figure 13.7. Junctions.	53
Figure 13.8. Open space	53
Figure 14.1. First node identified.	56
Figure 14.2: The swarm works together.	57
Figure 14.3. Maze solved.	57

1. Introduction

1.1 Goals

There are computer generated puzzles all around us. Crosswords, mazes and sudokus are put together in a fraction of a second thanks to widely available databases and algorithms. They are printed in newspapers at night and appear in our hands in the morning to puzzle our half-asleep brains on the way to work, college, school... The purpose of this project is to create software that provides the opposite to the experience of solving a computer-generated maze. The software accepts any kind of image that through a sequence of manipulations is converted into a collection of lines that define the borders in the 2D world. Then a number of agents are spawned that using some steering and tasks-distribution logic move around the environment cutting off deadends in an attempt for all to reach the destination point.

1.2 Mazes

There are various methods of solving mazes available[1] as well as software implementations of them[2]. One generates complex mazes out of squares, the other solves them, then another one lets the user to explore one in 3D. The purpose of this project is to approach representation of mazes and the process of solving them in a slightly more life-like way. Because in real life a maze isn't always boxy and a place we need to find our way out isn't always called a maze, the goal is to accept any kind of environment, where obstacles can come in any form and shape. And since the most common way to represent 2D worlds is an image, there is a need to find a way to convert it into some form that is understandable to a computer. Another goal is to make a swarm of agents do the job of exploring the environment. This will work as a simulation of a swarm of robots in real world and it must be ensured that the swarm is properly abstracted from the rest of the program and agents don't get access to information they wouldn't be able to access in real world and thus cheat. Also the agents are expected to move in a realistic manner and show a form of Swarm Intelligence[3].

In short this is what the program should do:

1. Let the user load an image.
2. Let him choose where the swarm appears and where it needs to go.
3. Convert the image into environment suitable for agents.
4. Put agents in the image and let them move around it and explore it.

While the main goal is to develop the mentioned software and give it as much functionality as possible in the given time, the development can be divided into two separate goals:

- Develop software that accepts any image and turns it into environment that can be populated with any sort of autonomous moving entities[4]. This part can be used separately for quickly creating test environments for any kind of Steering AI (Artificial Intelligence)[5] project that works in 2 dimensions, as long as it does collision detection with walls.
- Develop Distributed AI that effectively explores complex environments. This part should also be able to function in environments generated by other means and even may be useful as a program for robots in real life, provided with the same kind of sensors.

1.3 Contributions and Achievements

Primary contributions are as follows:

- Software was created as specified in the goals, but with a few features missing. Currently the swarm can identify junctions and split in the system of tunnels of limited width (around 22-28 points). Also there is no logic for distributing the swarm in open spaces.
- A Seamstress Algorithm was developed which converts edges defined by pixels into a system of line coordinates.
- A steering model which allows agents to move realistically in nonuniform tunnels, open spaces and other environments.
- A system of nodes was introduced as means of communication between agents. It allows the swarm to effectively distribute itself in a system of tunnels to eliminate all deadends and find the way to the destination point.

Among the other contributions it may be mentioned that a modification of an early version of the software was used as a part of assignment solution based on 'Dalek World' RTS game. It allowed creating levels for this game from images and avoid manually typing in coordinates of obstacles.

1.4 Lessons Learned

From this project the following things were learned:

- How to handle and manipulate images in Java.
- Some image processing techniques such as thresholding and floodfill.
- How to display multiple moving objects in Java.
- How to program Steering Behaviour using vector mathematics.
- The logic behind solving mazes.
- What Swarm Intelligence and Autonomous Moving entities are.
- Classification of mazes.
- Differences between C++ and Java.

2. Background on Mazes

2.1 Introduction

Generally a maze is a complex branching passage, through which a solver needs to find a route[6]. As for labyrinth, most sources state that at least in English it means a passage which from start till end doesn't have a single junction even though it can be quite curved[1,7]. I wouldn't agree with this terminology as labyrinth the way it appeared first time in Homer's Iliada undoubtedly had a great deal of junctions and deadends. If considered a junction less definition of labyrinth, escaping such labyrinth doesn't pose much challenge and that's completely unfair to Tesey. Therefore I will refer to labyrinth as a type of simple 2D maze.

2.2 Types of Mazes

While there are no official classifications of mazes it is possible to group them based on their shapes and various features. Firstly maze definition covers a surprisingly large subset of puzzles. Even though it always presumes finding a route, the system can consist of numbers, arrows, doorways or corridors. Almost all mazes except for corridor mazes use special rules which are expressed by symbols. The only type of maze that doesn't have any symbolic rules is corridor maze. Therefore software will only support this type of maze. Also it is the only type that is constructed and makes sense in real life.

As any system of objects, a maze can be in two and more dimensions. While it is impossible to create any 1 dimensional mazes, 2D mazes are most popular, as they are easy to depict, relatively easy to solve and the only ones that have real life implications. 3D mazes are not hard to depict with the help of computers, while four and more dimensional mazes require special symbolic features to mark passages in other dimensions usually known as portals. In some sort of way we are all familiar with such 3D and 4D systems from games, as any level of first person shooter is a 3D system, while a similar level with portals is already a 4D system. Yet these don't come under the definition of mazes due to the lack of complexity. The goal of this project is to support only 2D mazes, because it is possible to analyze a 2D maze of any complexity while the same is not true for mazes with more dimensions.

The shape of the maze's basic element is another important aspect. Sometimes it is the same as the shape of entire maze. The most common are [8]:

- **Orthogonal:** This is a standard rectangular grid where cells have passages intersecting at right angles.

- **Delta:** A Delta Maze is one composed of interlocking triangles, where each cell may have up to three passages connected to it.
- **Sigma:** A Sigma Maze is one composed of interlocking hexagons, where each cell may have up to six passages connected to it.
- **Theta:** Theta Mazes are composed of concentric circles of passages, where the start or finish is in the center, and the other on the outer edge. Cells usually have four possible passage connections, but may have more due to the greater number of cells in outer passage rings.
- **Upsilon:** Upsilon Mazes are composed of interlocking octagons and squares, where each cell may have up to eight or four possible passages connected to it.

There are more types, but most of them can be characterized with being chaotic, that is not really having any basic shapes.

The most widely used are orthogonal. After them come theta mazes, probably attracting people with the elegance of their circles. As for real-life implementation, theta mazes are even more popular.

2.3 Examples

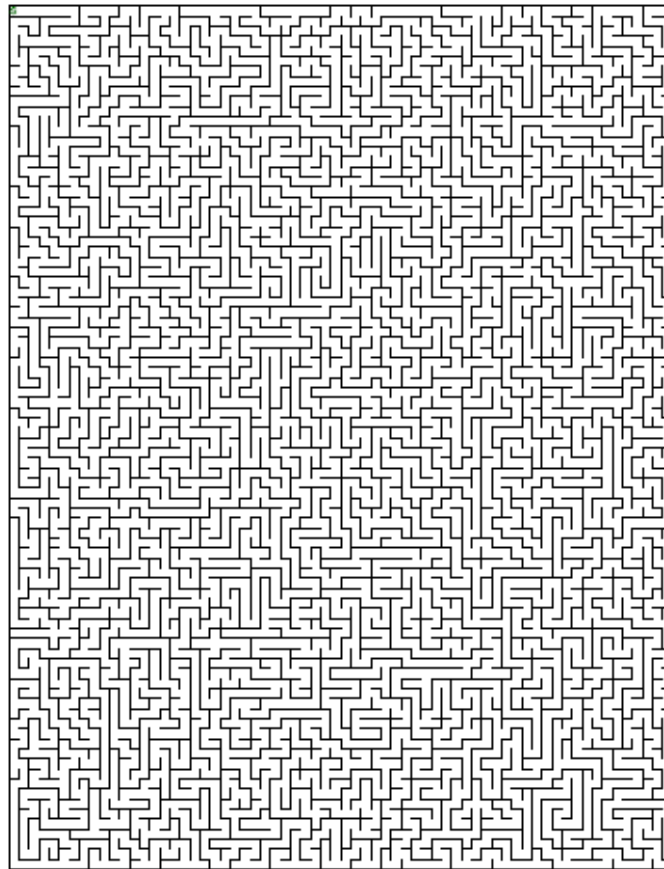


Figure 2.1. Orthogonal maze [9]

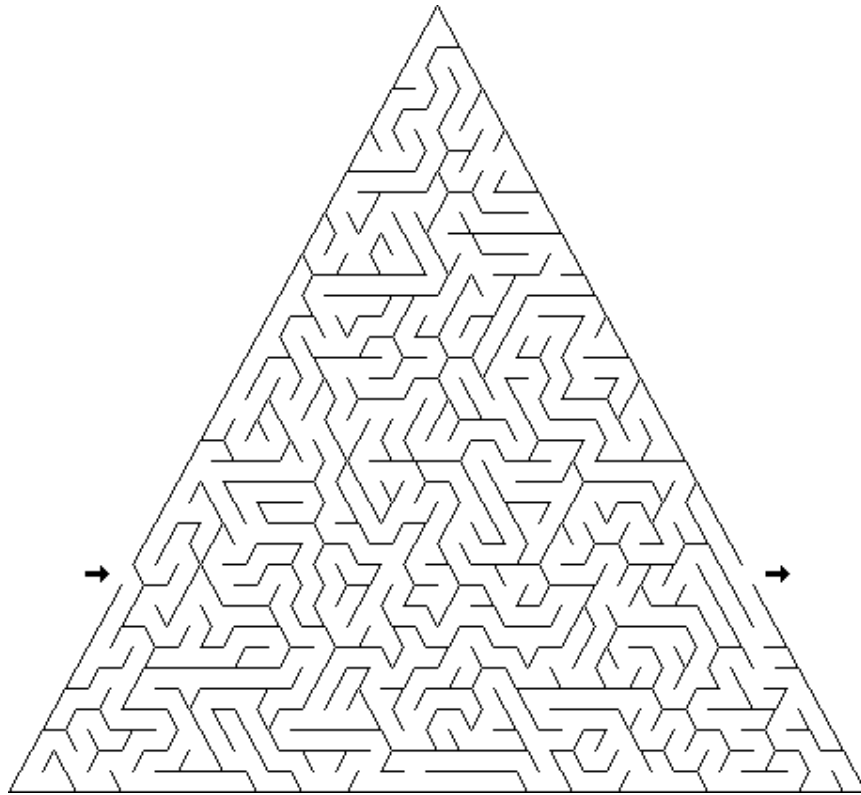


Figure 2.2. Delta maze [8]

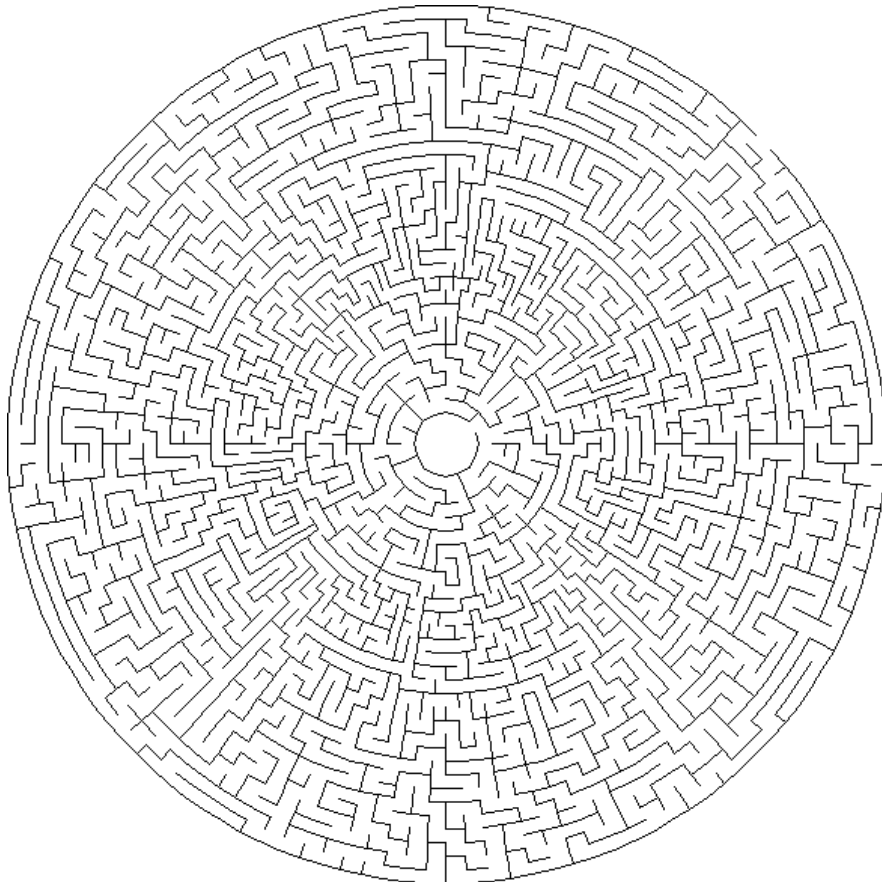


Figure 2.3. Theta maze [8]

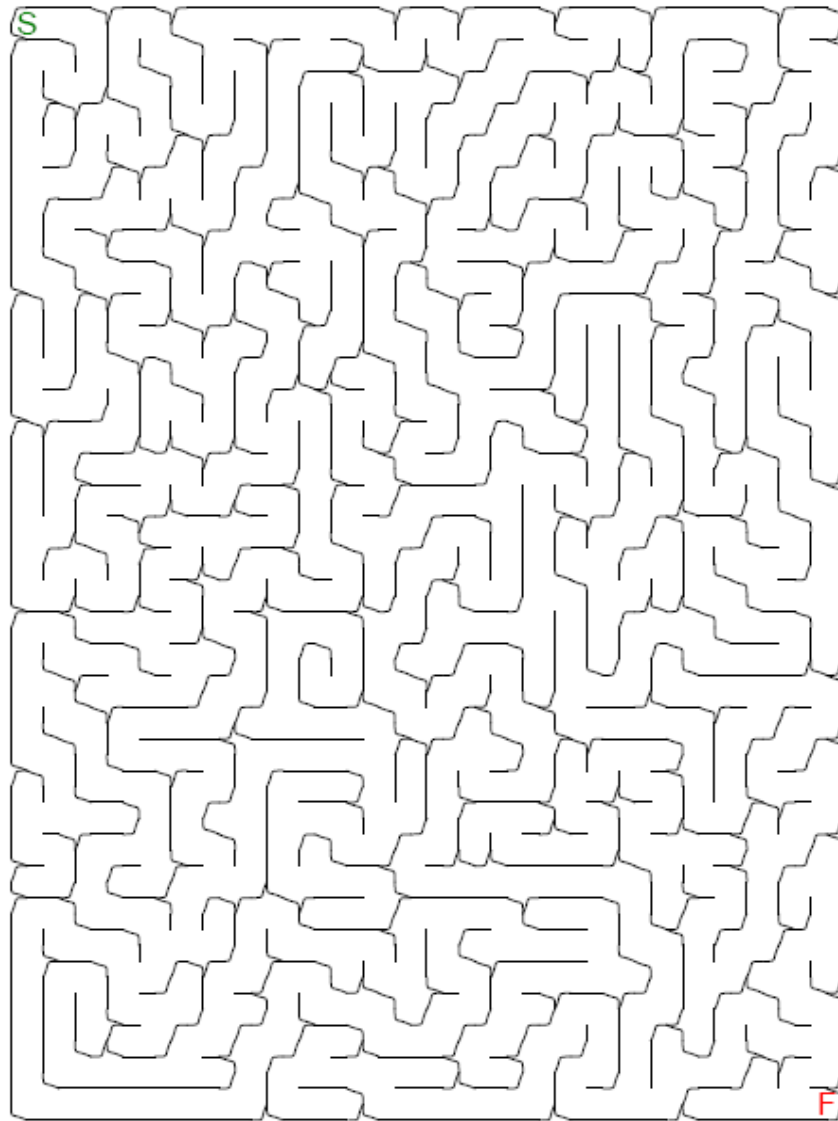


Figure 2.4. An alternative kind of maze [8]

3. Background on Autonomous Agents

3.1 Introduction

Autonomous agents[4] are entities, which once given instructions, are able to function and react on changes in the environment around them without any further human involvement. An autonomous agent may or may not need to be able to handle any possible situation. The instructions an agent acts upon may be a sort of Artificial Intelligence[5], especially if it has a way to learn about the environment. If a number of agents shares the collected knowledge and works together to achieve a common goal, that means they are a form of Distributed Artificial Intelligence, or more accurately its branch called Multi-Agent Systems[10]. Such a system allows agents to collectively reach goals which are otherwise difficult to achieve. Swarm Intelligence is different from Distributed AI in a few ways. Because Swarm Intelligence is mostly used for sophisticated movement of entities in the environment, self-organisation is usually considered to be necessary. Interaction between agents that produces such behaviour is relatively simple, but on the large scale may lead to astonishing results. The same is true for any other application of swarm intelligence. One of the good examples is its use in networking[11]. Because Swarm Intelligence is cheap, robust and simple, its applications are widely researched in the industry.

3.2 Autonomous Agents

An Autonomous Moving Entity usually has one or several goals. These can be hiding from other agents, flocking with them, collecting food, etc. An agent can only be in pursuit of one goal at a time. To allow the agent intelligently switch between goals states are used. When an agent is in one state, it follows a certain set of rules which are supposed to let him reach the current goal in the best way possible. But if for some reason the behaviour is no longer adequate to the environment or the primary goal has changed, the agent will switch to a state that is suitable for the situation. For example an entity that simulates behaviour of a rabbit looks for food most of the time and consumes it, but at some point of time rabbit realises that he ate enough, this triggers the change of states and it will start looking for a place to take a nap. If the rabbit notices a predator while eating, this will also cause a change of states, and the rabbit will try to run away. In the above two examples states were changed because the primary goal has changed. In one case because the goal was fulfilled and in the other case because a more important goal emerged. But a change of goals isn't always necessary. For example if a rabbit starts to suspect that there is a predator nearby, but doesn't see it yet, the goal will remain the same, but a few rules in the new state will be different, like moving around cautiously, staying in the shadow and always being aware of an escape route.

Programming of certain types of behaviour for Autonomous Agents sometimes may be obstructed with the need to reduce CPU usage. This sometimes may lead to sacrificing some functionality for speed. Also it is hardly possible to create a universal agent with only basic algorithms and no manual tweaking. This drastically increases the length of code with various 'If' statements and calls to the environment. This takes away the elegance of initial code but is unfortunately unavoidable. While tweaking like this may seem to make the agents overspecialise, in fact it does the opposite if done properly.

4. Discussion of software methodology

4.1 Introduction

Rapid Application Development Model[12] was used as a software methodology. This model is often used in situations when developed software doesn't have exactly defined requirements or when there is not enough time to split the work stepwise. Compared with traditional methodologies it offers a number of advantages:

- Takes advantage of modern technologies, which often allow leaving out hand-design and paper-based modelling without the loss of quality.
- Helps avoid delays connected with transitions between steps.
- Allows removing some documentation that are irrelevant due to the nature of the project.

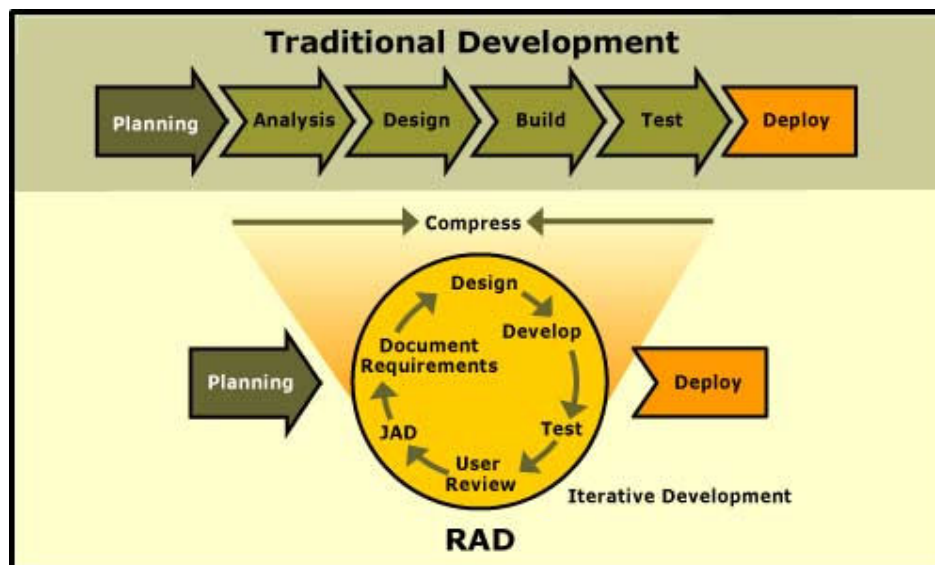


Figure 4.1. Comparison of traditional and RAD models [13]

This appears to be an appropriate approach because this project in its nature is an experiment which is carried out to see how much integration can be achieved between images and autonomous moving entities in the given time. A number of other factors were considered:

- The number of classes is relatively small and their hierarchy is very simple.
- Every next step in the development process can be taken only after the previous one was completed. However the distinction between steps is very abstract and is only required to structure the development process.
- There is no certain definition of requirements for produced software.
- Limited development time.

5. Discussion of technical architecture

5.1 Introduction

The programming language to be used for this project will be Java, in this section I will outline the reasons why I chose this language. At the time this project was started I experience programming in 4 object oriented languages which are C++, C#, Java and VB. VB was excluded from the list of possible choices right away, as this language is only partially Object-Oriented and clearly lacks flexibility. A combination of C# with C++ in .NET environment seemed to be promising, as C++ would be very fast and quiet flexible for the logical part at the backend, especially for the navigation part, and C# would do a fine frontend. It should be simple to connect the two of them in .NET application.

5.2 Comparison of C++ to Java

1. Generally interpreted Java may run up to 20 times slower than C. However if an application is not very calculation-intensive, this difference may even be impossible to notice. This aspect is hardly going to affect performance of the program as it won't use any sophisticated graphics.
2. Arrays in Java and C++ have a very different structure. In Java there's a read-only **length** member that holds the size of array. This allows run-time checking to throw an error when bounds of array are left. Handling this manually requires additional time and may cause problems anyway.
3. No forward declarations are necessary in Java. If you want to use a class or a method before it is defined, you simply use it – the compiler ensures that the appropriate definition exists. While such approach may introduce some disorder, which is easy to cope with by just using search, it is a great deal of help during development process.
4. There are no such things as pointer the way they appear in C++. When you create an object with **new**, you get back a reference. However, Java references don't have to be bound at the point of creation, unlike C++ references that must be initialized when created and cannot be rebound to a different location. The other reason for pointers in C and C++ is to be able to point at any place in memory whatsoever (which makes them unsafe, which is why Java doesn't support them). Pointers are often seen as an efficient way to move through an array of primitive variables. Java arrays allow you to do that in a safer fashion. It is a good practice to avoid using pointers when they are unnecessary, as they are capable of creating all sorts of trouble.

5. There are no destructors in Java. As variables don't have a "scope", to indicate when the object's lifetime is ended – the lifetime of an object is determined in its place by the garbage collector. There is a **finalize()** method that's a member of each class, something like a C++ destructor, but **finalize()** is called by the garbage collector and is supposed to be responsible only for releasing "resources". When you need something done at a specific point, you should not rely upon **finalize()**, but rather create a separate method. In other words everything in C++ will or should be destroyed, but not everything in Java is garbage collected. The garbage collection system in Java is a big plus. Generally it allows creating a resource-friendly system for a programmer who doesn't take care of that at all, and whose approach could be dangerous otherwise.
6. Java does not support default arguments. This could sometimes be a minor disadvantage and introduce a few more lines of code.
7. Java uses a singly-rooted hierarchy, so all objects are ultimately inherited from the root class **Object**. There's one single ultimate hierarchy. In C++, a new inheritance tree can be started anywhere. This could sometimes bring a lot of confusion when working with unusual complex classes, as it will be almost impossible to trace what functionality they have and don't have if this is not stated clearly in documentation.
8. Garbage collection means memory leaks are a great deal harder to cause in Java, yet not impossible, especially if native method calls are made that allocate storage, as these are typically not monitored by the garbage collector. Nevertheless, many memory leaks and resource leaks can be tracked to a poorly written **finalize()** or to not releasing a resource at the end of the block where it is allocated. While high usage of resources is not expected in the program, this is an important point.
9. Java has built-in multithreading support. There's a **Thread** class that is inherited to create a new thread. Mutual exclusion occurs at the level of objects using the **synchronized** keyword as a type qualifier for methods. When a **synchronized** method is entered, it first "locks" the object against any other **synchronized** method using that object and "unlocks" the object only upon exiting the method. Multithreading is expected to be used a lot for agent movement and built-in support is a great advantage.
10. Exception specifications in Java are greatly superior to those in C++. Instead of the C++ approach of calling a function at run-time when the wrong exception is thrown, Java exception specifications are checked and enforced at compile-time. Generally it seems much easier to catch errors in Java as Javadocs provide all the information about exceptions as well as the compiler itself.
11. Java has method overloading, but no operator overloading. The **String** class does use the + and += operators to concatenate strings and **String** expressions

use automatic type conversion, but that's a special built-in case. This is a disadvantage for this project as method overloading can sometimes greatly enhance readability of code. However this is possible not very often.

12. Java has built-in support for comment documentation, so the source code file can also contain its own documentation, which is stripped out and reformatted into HTML through a separate program. This is a great benefit for documentation procedure.
13. Java contains standard libraries for solving specific tasks. C++ relies on non-standard third-party libraries.
14. Generally, Java is more robust with the help of:
 - All array accesses are checked for bounds violations
 - Automatic garbage collection prevents memory leaks
 - Clean, relatively fool-proof exception handling
 - Simple language support for multithreading [14]

6. Environment structure decision

6.1 Introduction

As agents are supposed to navigate and prevail in any kind of environment, it will be critical for general usefulness of the application to offer as convenient way of importing environments as possible. Such method for a 2D world could very well be using images. Of raster and vector images the most widely used is raster.

A raster or bitmap image contains information about the colour of every pixel of the image (unless it's compressed like a Jpeg) [15]. This is not very convenient for the program, because it doesn't provide comprehensive information about objects (obstacles) around an agent. The only information it provides in this form is whether a certain pixel is free or not free and its position against other pixels. At this stage a decision had to be made how the environment will be represented in the programs memory, how it will be perceived by the swarm and which way it will be converted from a bitmap to the desired state. Ideally obstacles should be represented as objects. But having the world stored as matrix could be an alternative.

6.2 The Matrix

This method doesn't require a lot of manipulation with an imported image. We let the user choose the colour which will stand for free space (default is white) so that the rest will stand for obstacles, or we could let him choose a colour for obstacles (default black) and the rest will stand for free space. After this we will simply have a 2D Boolean array, with each value representing a free space or piece of an obstacle. Actually this will be the world the way agents will perceive it. This doesn't seem too promising. Detecting forks and deadends will become a very challenging task. A possible approach could be analyzing a certain number of pixels around an agent and then using some complex algorithm to carry out decisions about possible routes. This method is expected to be very prone to errors. Also the whole model will be unrealistic in its work.

Plus:

- Faster to implement.
- The environment will hardly be altered in the process of import. So there are fewer chances it will get corrupted.
- Importing environment will be fast and will take fewer resources.

Minus:

- Pathfinding and fork/deadend identification can be very tricky.

- Each event of analyzing environment by an agent may take a lot of processing. So in case there is a large swarm all moving in real time and accessing simultaneously one particular array, the program may become very slow.
- Swarms perception of the world will be unrealistic so it is expected it will behave less realistically.

6.3 Vectorization

Another approach is using objects in the model. Or more correctly lines to represent shapes of the objects. Converting a bitmap to a collection of objects will involve a number of steps. First of all it will be important to get rid of useless information, there's plenty of that in a raster image. This will involve edge detection technique, which will return another raster image. This image will contain only edges of whatever shapes there were on the picture. The edges will be one pixel wide. Minor details and colour of objects will be lost. This in fact means that it will no longer be possible to determine if we are inside or outside an object, so it could be a good idea to retrieve such information and later use it when time comes to create objects.

After this it will be necessary to vectorize the given lines. This can be done by simply detecting all the lines or probably carrying out the whole vectorization process.

Plus:

- This is a more real world like model.
- Pathfinding and fork/deadend identification will be less prone to errors.
- Once environment is ready the program will be done with pixels and will only analyze objects, which is faster.

Minus:

- Will take more time to implement.
- The process of vectorization may be tricky and sometimes may produce invalid or incomplete models.
- No guarantee that navigation in an object model world will be more efficient than in the other approach, even though so much manipulations were introduced to the process of importing.

6.4 Conclusion

Considering that vectorization provides more academic value and makes the model more real world oriented, it appears to be more attractive.

7. Discussion of feature implementation

7.1 Introduction

During the design stage it has become clear that this project will have two challenges rather than one. Firstly there is a goal to place a swarm in a labyrinth and give it a behaviour pattern that will allow agents to prevail in such system. The second challenge came out of an idea, that any software developed, whatever purpose it serves, should be usable for a majority of users. As a result came one more major task of converting a widely used bitmap format into program specific system of lines.

Implementation of one of the tasks directly influences requirements of the other. The degree of balance between the two is worth a separate discussion.

7.2 Conversion Accuracy Balance

The process of representing and converting a maze into a system of lines may vary a lot in complexity depending on the given maze. Also the final system may vary in a similar sort of way depending on requirements. Basically it is the problem of conversion between raster and program representation and the problem of navigation that define the above two.

That is if we take a simple orthogonal maze we should have this smooth workflow:

Image → conversion → system of lines → navigation

However if we introduce a more complex image we will have to make sure we deal with the complexity somewhere in the workflow. Otherwise throughout the workflow a lot of very complex and largely useless data will be processed:

Complex Image → conversion → complex object system → navigation?

Processes in bold will be parts that are far from the simple and elegant orthogonal case given in the beginning. Unfortunately even the conversion would come in bold because however close the system is to the image it still has to be a system of coordinates. As for navigation there's got to be a much bigger question mark next to it because while identification of decision points is a challenging task, doing that in a system packed with details and squiggly objects will be near to impossible (at least in this project's time limits).

It is possible to get the final state of the system close to the first diagram through a conversion process. While it is possible, and in fact it may even simplify the process of converting a complex image, at what price will it come?

Complex Image → simplifying conversion → system of lines → navigation

The above diagram gives a suggestion that this way the system will be losing something. And it is the details. While that is the very essence of what the idea was to take a complex maze and represent it in a system of lines, what may happen in the end is that the agents, being unaware of certain details will be living in a different world than the one we populate with them. For example if there was a small section of the wall that would be in the way of an agent, the agent will simply ignore it and just float over/through it because the conversion process has found it irrelevant and removed it from the system. And that wouldn't be a converter's bug. In fact that is his job to remove unnecessary details. The bug will be a design mistake which allowed agents to move in one world while perceiving the other world. A solution to that would be a good balance between simplicity of the line system and its accuracy. A balance that can only be achieved through sufficient testing.

7.3 Navigation Algorithm

When exploring a maze agents will be following an algorithm, which will provide them with enough instructions for every situation they may find themselves in. It also has to eventually take them to the destination. Here will be addressed only the task of taking at least one agent to the destination. The three steps show how the algorithm was improved by addressing more situations and rearranging the order of actions.

Algorithm 1: Initial idea of what decisions will be made by an agent.

```
repeat
    Go through the tunnel
If fork encountered
    Set a new node
    Set node directions to point at all new available tunnels
    Follow direction closest to destination
If node encountered
    If one direction is available
        Mark direction
        Follow direction
    If more than one direction available
Mark direction
    Follow any unmarked direction
```



```

    If all directions are marked
        Follow any direction
    If deadend encountered
        Disable direction that points at it
        If this disables the only direction of a node
            Disable direction that points at it
            Continue to check this doesn't make any more dead nodes
        Return until a node with a valid direction is reached
Until destination reached

```

Algorithm 2: A more accurately done pseudocode with a proper start.

```

Start
Set a new node
Set node directions to point at all new available tunnels
Mark most promising direction
Follow selected direction
Repeat
    Go through the tunnel
    If fork encountered
        Set a new node
        Set node directions to point at all new available tunnels
        Mark most promising direction
        Follow selected direction
    Else If node encountered
        If unmarked directions available
            If more than one unmarked
                Mark most promising unmarked direction
                Follow selected direction
            Else
                Mark the one unmarked direction
                Follow selected direction
        Else
            If more than one available
                Follow most available promising direction
            Else
                Follow available direction
    If deadend encountered
        Disable direction that points at it
        Move back to node

```

```

Repeat while all direction from current node are disabled
    Disable node
    Disable direction that points at it
    Move back to previous node
If one direction is available
    Mark direction
    Follow direction
If more than one direction available
Mark any one unmarked direction
    Follow selected direction
If all directions are marked
    Follow most promising direction

```

Algorithm 3 (currently final): A more efficient version. Algorithm for deadend was a lot simplified because returning from a dead direction could be thought of as just moving along a new route. Also it is safer, as an agent will know what to do if he accidentally encounters a dead node.

```

Start
Set a new node
Set node directions to point at all new available tunnels
Mark most promising direction
Follow selected direction
Repeat
    Go through the tunnel
If fork encountered
    Set a new node
    Set node directions to point at all new available tunnels
    Mark most promising direction
    Follow selected direction
Else If node encountered
    If unmarked directions available
        If more than one unmarked
            Mark most promising unmarked direction
            Follow selected direction
        Else
            Mark the one unmarked direction
            Follow selected direction
    Else if valid directions available
        If more than one available
            Follow most promising available direction

```

```

        Else
            Follow available direction
    Else
        Disable node
        Disable direction that points at it
        Move backward
If deadend encountered
    Disable direction that points at it
    Move backward

```

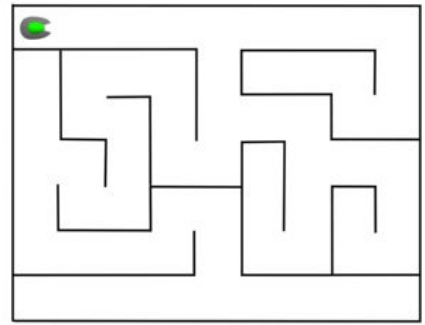
Next the algorithm will be tested by applying it to a swarm of 3 agents placed in a very simple maze, where the goal is to go from top left corner to the bottom right corner. Assume it takes 1 point of time to go through 1 square of maze. The agents are: Alpha - green, Beta – blue and Gamma - yellow. They start out one after another with a difference of 1 point of time. Blue circles will symbolize nodes, while red circles will symbolize dead nodes as well as deadends. In fact deadends are only shown for clarity, they are not going to appear in the linked list. Instead the routes that lead to deadends will be disabled. Finally grey arrows show unmarked directions, green – marked directions and red – disabled directions.

Step 1:

Alpha – spawns.

Creates a new node. Sets one direction – east.

Beta and Gamma – inactive.

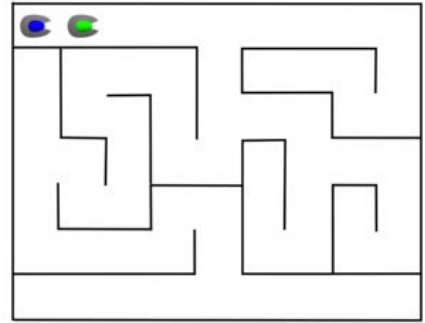


Step 2:

Alpha – follows selected direction.

Beta – spawns and follows the only available direction.

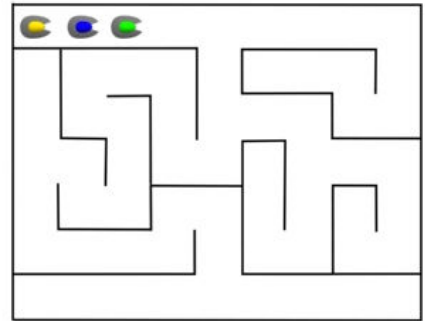
Gamma – inactive.



Step 3:

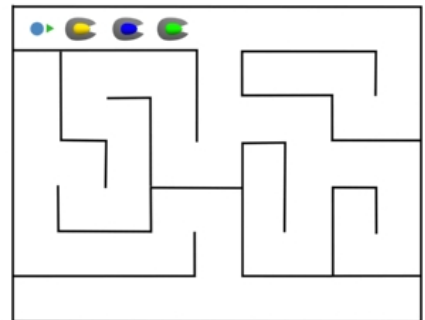
Alpha and Beta – continue to move in selected direction.

Gamma – spawns and follows the only direction.



Step 4:

Alpha, Beta and Gamma follow the same direction.



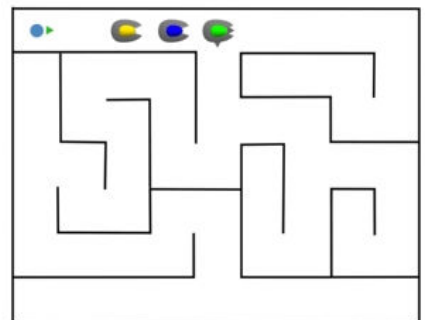
Step 5:

Alpha, Beta and Gamma follow the same direction.

Alpha recognizes a fork. Creates a new node.

Sets two directions – east and south.

Marks south direction as it has a direction closer to destination. Prepares to follow it.

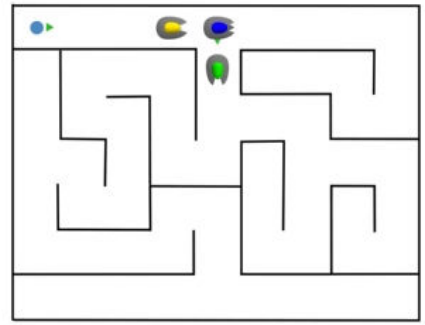


Step 6:

Alpha follows south direction.

Beta reaches a node and follows the only unmarked direction – east.

Gamma continues to go in the same direction.



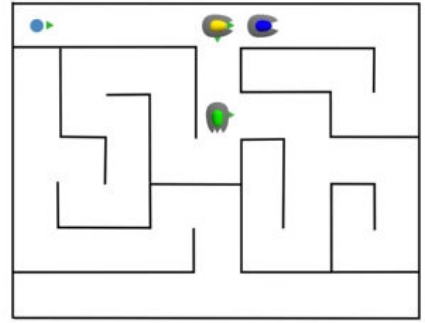
Step 7:

Alpha discovers another fork and creates a node.

Sets 2 directions – east and south. They are equally good. Randomly picks east.

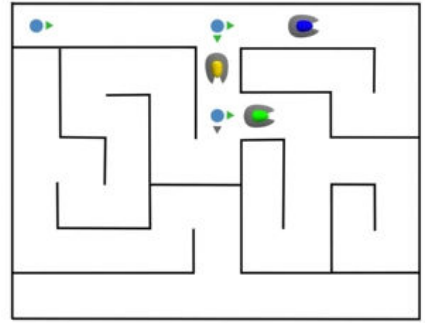
Beta follows same direction.

Gamma reaches a node and follows south.



Step 8:

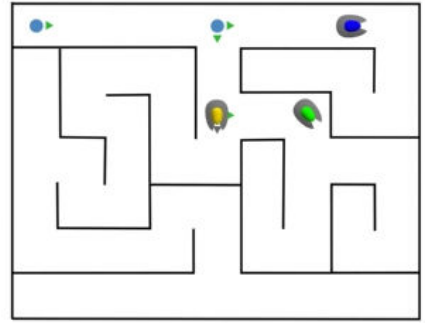
Alpha, Beta and Gamma continues to move forward.



Step 9:

Alpha and Beta continue to follow their directions.

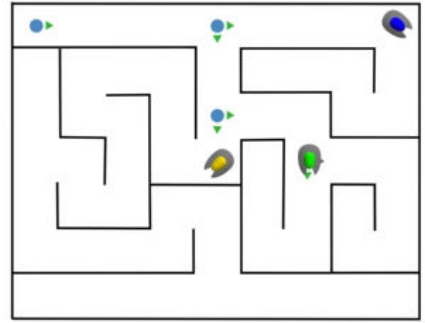
Gamma encounters a node and follows the only unmarked direction – south.



Step 10:

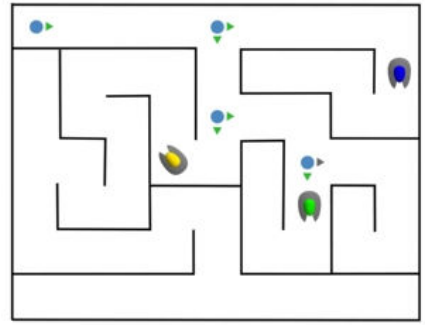
Alpha encounters a fork, creates a new node and two directions: east and south. Follows south direction.

Beta and Gamma follow their directions.



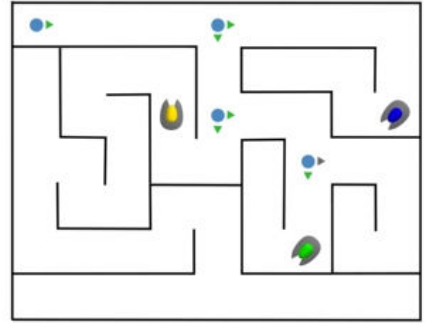
Step 11:

Alpha, Beta and Gamma continues to move forward.



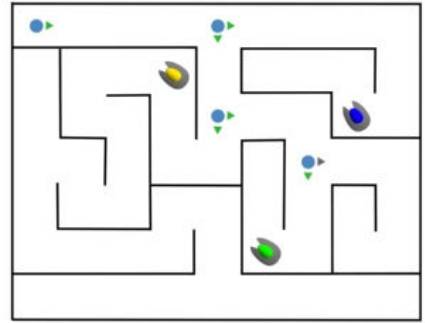
Step 12:

Alpha, Beta and Gamma continues to move forward.



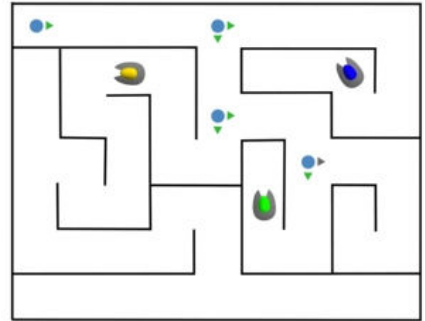
Step 13:

Alpha, Beta and Gamma continues to move forward.



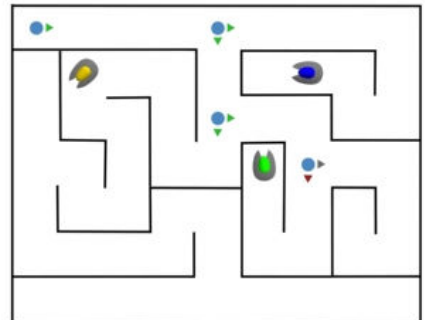
Step 14:

Alpha, Beta and Gamma continues to move forward.



Step 15:

Alpha encounters a deadend and disables direction that points to it. Alpha will now move backward to the previous node.
Beta and Gamma move forward.

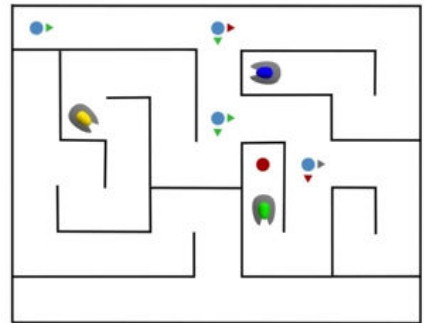


Step 16:

Alpha now moves backward.

Beta discovers a deadend, disables direction pointing to it and will now move backward.

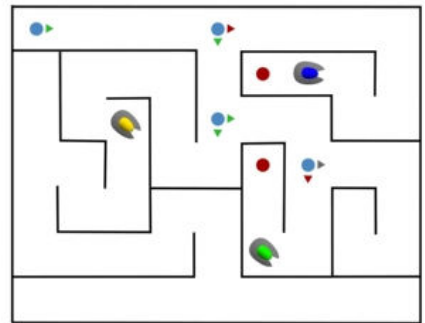
Gamma moves forward.



Step 17:

Alpha and Beta move backward.

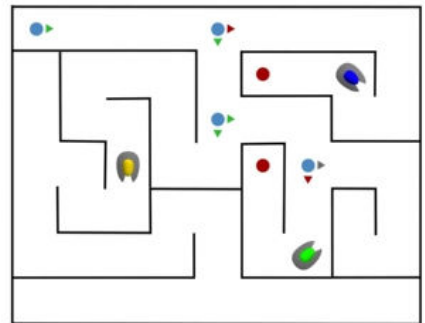
Gamma moves forward.



Step 18:

Alpha and Beta move backward.

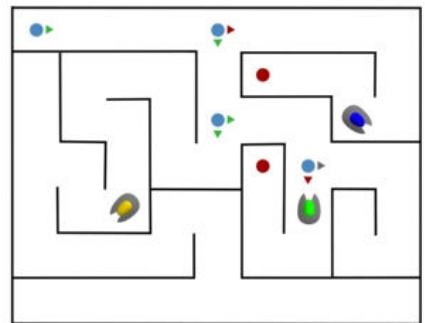
Gamma moves forward.



Step 19:

Alpha and Beta move backward.

Gamma moves forward.

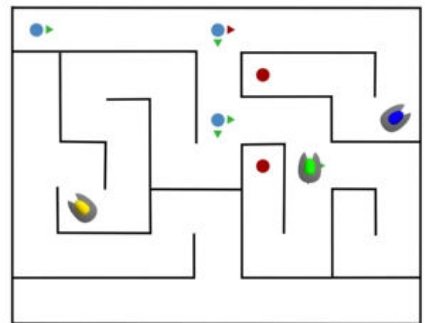


Step 20:

Alpha encounters a node and follows the only valid direction - east.

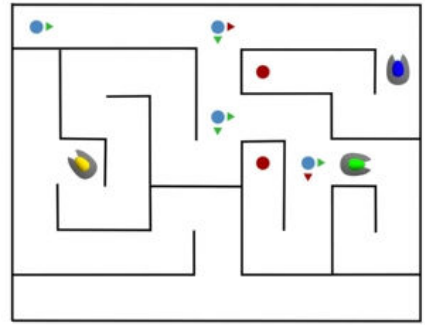
Beta moves backward.

Gamma moves forward.



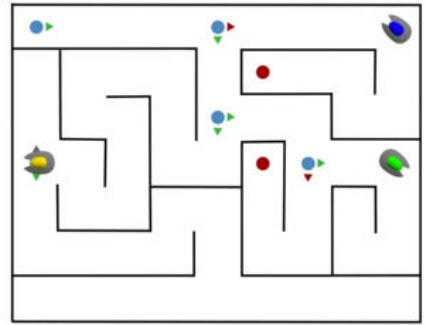
Step 21:

Alpha moves forward.
Beta moves backward.
Gamma moves forward.



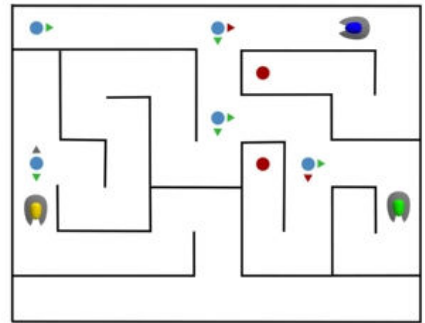
Step 22:

Alpha moves forward.
Beta moves backward.
Gamma encounters a fork, creates a new node and sets two directions: north and south. Marks a more promising south direction and follows it.



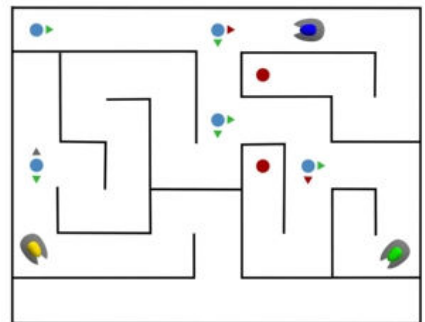
Step 23:

Alpha moves forward.
Beta moves backward.
Gamma moves forward.



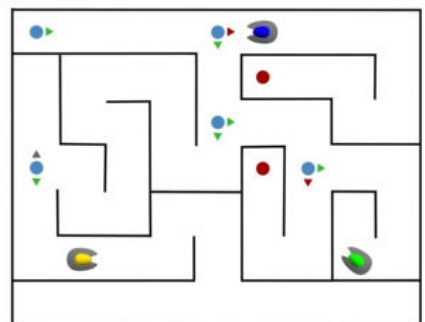
Step 24:

Alpha moves forward.
Beta moves backward.
Gamma moves forward.



Step 25:

Alpha moves forward.
Beta moves backward.
Gamma moves forward.

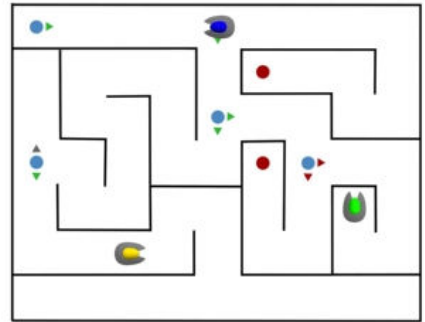


Step 26:

Alpha encounters a deadend and will now move backward. Disables direction that points to the deadend.

Beta comes to a node and follows the only available direction.

Gamma moves forward.

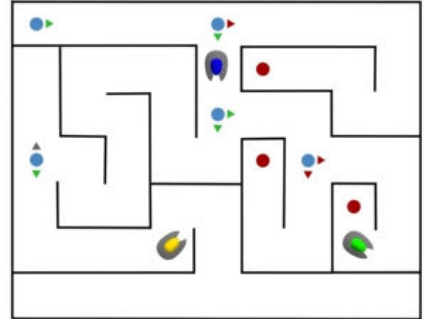


Step 27:

Alpha moves backward.

Beta continues to follow its new direction.

Gamma moves forward.

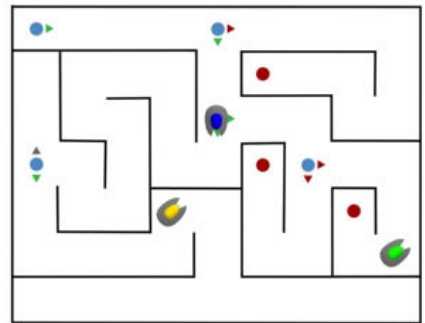


Step 28:

Alpha moves backward.

Beta reaches a node and randomly decides to move south.

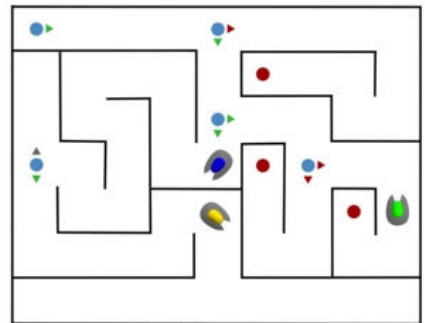
Gamma continues to move forward.



Step 29:

Alpha moves backward.

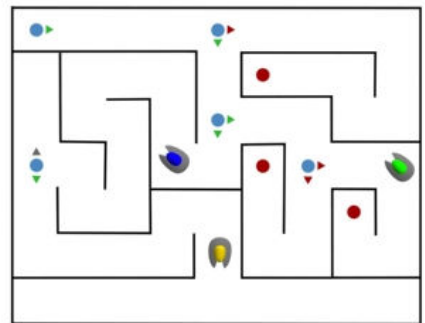
Beta and Gamma move forward.



Step 30:

Alpha moves backward.

Beta and Gamma move forward.



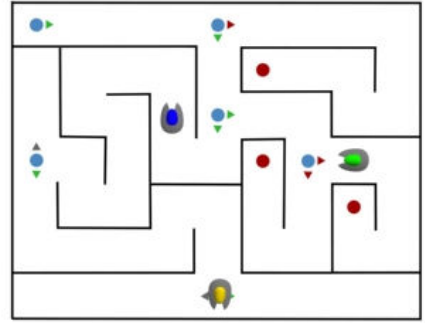
Step 31:

Alpha moves backward.

Beta moves forward.

Gamma encounters a new fork, creates a new node and two directions: east and west.

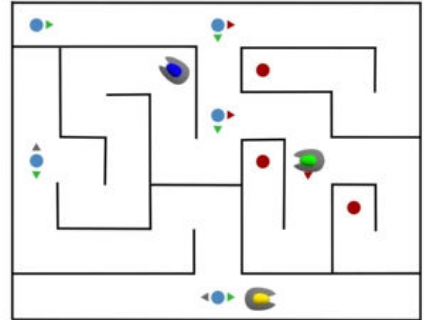
Marks east as most promising and follows it.



Step 32:

Alpha reaches a node without any valid directions, disables direction that points at and will continue to go backward.

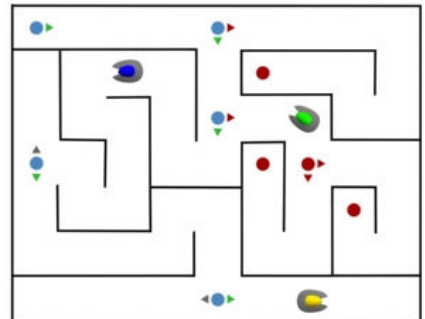
Beta and Gamma move forward.



Step 33:

Alpha moves backward.

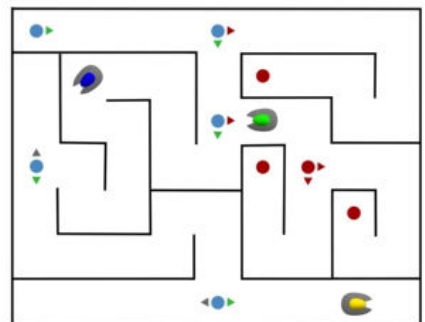
Beta and Gamma move forward.



Step 34:

Alpha moves backward.

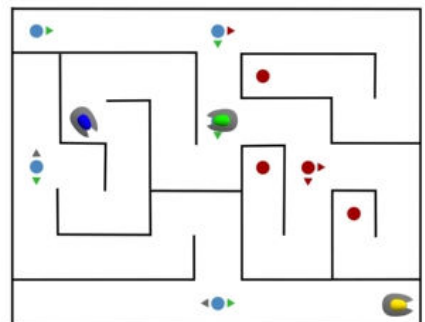
Beta and Gamma move forward.



Step 35:

Alpha encounters a node and follows the only valid direction – south.

Beta moves forward. Gamma reaches destination.



7.4 Conclusions

The algorithm has successfully helped one of the agents to reach the destination. Due to the fact there were three agents which shared information, whether any one of them has made a different choice, the number of steps until one of them reached destination wouldn't significantly vary. The same however isn't true for one or a number of agents that don't have any communication, as they all may get to explore the same part of maze unaware of each other's findings.

In the process of testing a flaw in the algorithm was revealed. On step 28 Beta has made a random decision to move south, however if it decided to move east, it could encounter an unexpected obstacle: agent Alpha. This basically became possible because the eastern route was disabled by Alpha after Beta has started to follow it. There are two ways to deal with this problem: either prevent an agent from following a disabled route right at the point of time it becomes disabled or handle the scenario of two agents meeting each other face to face. The first solution, while slightly improving swarm performance can slightly slow down the software, because every agent will have an additional task of checking if he is following a valid route to be done every point of time. The second solution is more elegant: if two agents meet each other face to face, the one that was following a valid route will turn around and the one that was following a disabled one will follow him. This is because a valid route can just as well be invalid, while an invalid route has to be invalid for sure.

8. Development Process

8.1 The Software Development Process

Software development process was divided into separate stages(steps), where completion of each step provides the grounds for adding new features defined by the next step.

1. Write a number of prototypes to test various image manipulation techniques.
2. Create a framework, which allows the user to easily load an image of his choice into the environment. The image needs to be converted into a form that allows further manipulations.
3. Add functionality to pick a pixel on the image with a mouse click. This event should be able to generate the coordinates of clicked pixel in relation to the image, and not the window.
4. Utilize this functionality to allow the user to pick start and finish positions for the swarm.
5. Introduce thresholding as means to identify objects on the image. Also develop a way to display the results of thresholding.
6. Add means to control the thresholding process.
7. Create a function that will throw away the interior of objects and leave only their edges.
8. Develop a function that will convert the edges into a set of lines, test it and optimize.
9. Develop a way to display the result of the above two functions.
10. Test and clean the code created until this stage. Document the steps. This completes phase one of the development.
11. Introduce environment that tracks and displays a number of autonomous entities moving on top of the image.
12. Find means to detect which is the closest line to the given point and how far it is.
13. Create autonomous moving entities that detect walls with sensors.
14. Introduce logic for steering away from obstacles and navigating tunnels.
15. Introduce an algorithm that will allow an agent to detect junctions.
16. Develop logic for navigating between nodes and updating them.

9. Program Interface

9.1 Interface Considerations

The Application window consists of the following sections: World Space, Toolbar and Information Bar. World space takes up most of the window and shows the two dimensional world as well as the process of image-to-world conversion. The Toolbar provides tools for manipulating the environment. As the process of preparing the environment is divided into several steps, each step has a unique set of tools. These are positioned in the top left corner, while buttons for navigating between steps are in the right corner together with the 'Exit' button. The right side of the window is occupied with 'Information and Tips' textboxes. The Information window displays details about various events that took place in the 2D world, for example, it will display the number of lines that were generated in the process of image conversion. The tips box provides hints about what actions the user may take during the current step and what effects she may expect from using available tools. This was found more helpful than providing the same type of information in a manual as people generally prefer to get their questions answered without having to search anywhere at all. Provided that this information was in a manual text file, if a user didn't understand what and why is or is not happening in the environment he would have to look for the manual file and then look for the specific conditions within the text. This is not only time consuming but can be irritating. While in the case of tips box, the user would only have to look in the bottom right corner of the window. All this however is true if the hints are well written and provide exactly the right information.

When resized, the two bars will preserve their height/width respectively, while the world area will expand to take up the window area. An image is always in the centre of the world area. If it is larger than the world area, it will remain centred but it will be impossible to view its edges that are out side the viewable area. Currently no scrollbars are provided due to the complicated nature of the process of picking. When the functionality of picking a certain pixel with a mouse click was introduced, it turned out to be problematical to directly transform the coordinates of window space the coordinates of an image. As a result an algorithm was developed that takes in account the relative position of panels and produces the precise coordinates of the clicked pixel. While at the current state it works well, introduction of scrollbars would complicate the picking procedure significantly and make very likely to work incorrectly in different environments.

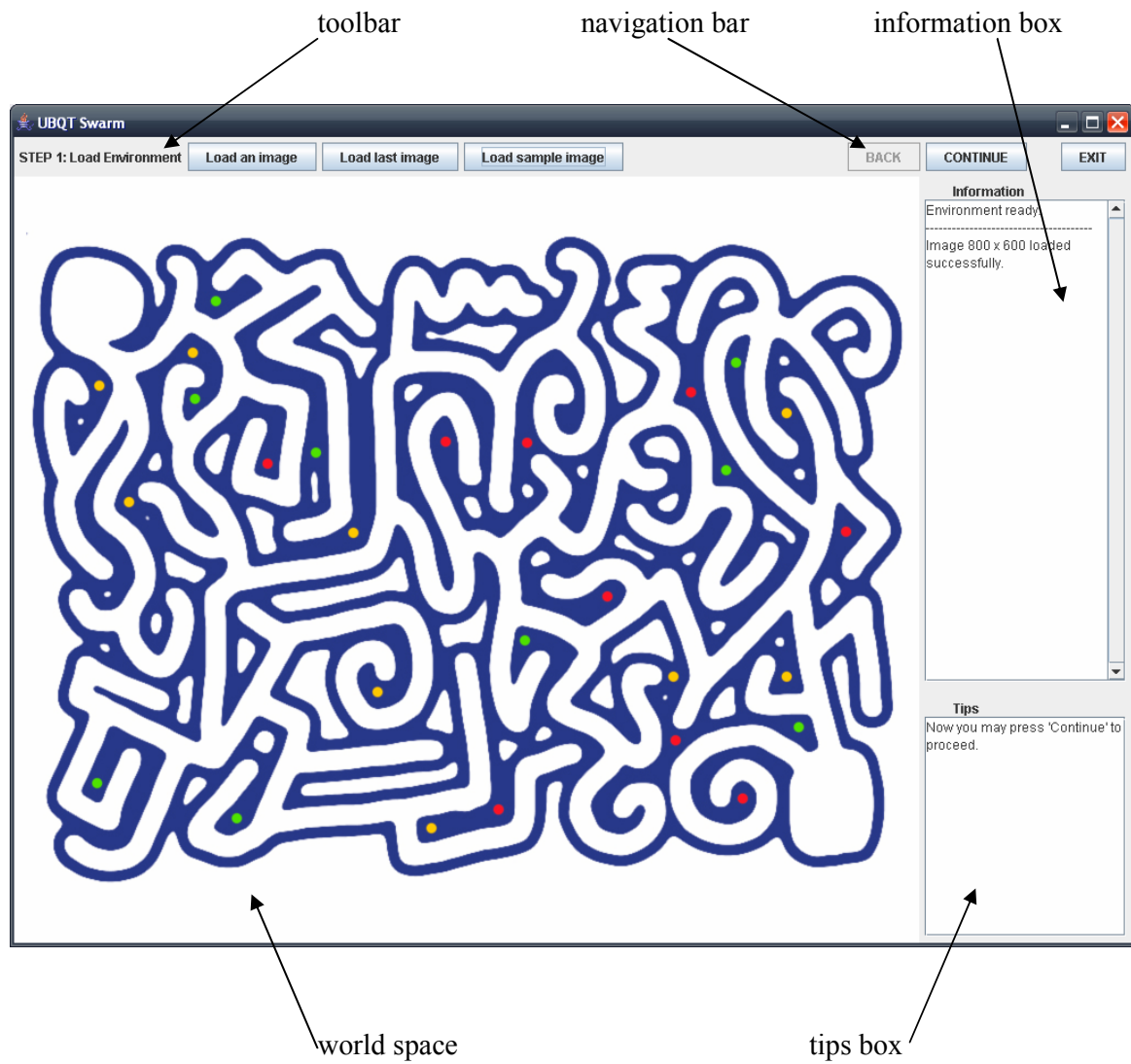


Figure 9.1. Program interface

10. Development of Image Conversion

10.1 The Image Conversion Process

In the development plan the first task was to load an image into the application environment and make it ready for further manipulation. This was done easily as Java provides extensive support for image manipulation. As the next task was to get the image to undergo a number of conversions, before it is possible to extract lines from it, the image was first stored in a 1D array of integers and then converted to a 3D array of integers as suggested in a pixel processing tutorial[1]. At this state the array stores X and Y position of a pixel as well as its alpha, red, green and blue values. This approach provides exhaustive information about every pixel of the image in a form that is very easy to manipulate. However it has its shortcomings: an array created from a relatively large image will take up a lot of space and may not be supported by the java environment. Due to the nature of further image manipulations storing it in this form was found completely unnecessary. The first step in converting the image was thresholding: the result of which is the same image, where a pixel can be either black or white. As there is no need to store information any further in image format, the result of thresholding is stored in a 2D array of Booleans, or in other words in Binary format. As every pixel of image can only be in one of the two states (black or white) after thresholding, this is the most efficient way to store such information without any compression.

The next step was to extract edges of objects that were identified with thresholding. This is done easily by changing the colour of every black pixel to white if it doesn't have a single neighbour pixel which is white. Of course it is necessary to check that the neighbour pixel didn't become white because of the conversion. This can be done by storing a pixel to the left and the whole row above current pixel in the state they were before conversion. This is much more efficient than making a copy of the whole array. After this a simple function was written that can extract only horizontal and vertical lines.

10.2 PseudoCode

This is pseudocode for reading horizontal lines:

```
Isline=false
For number of rows
  For number of columns
    If isline=false
      If currentPixel=black & nextPixel=black
        Isline=true
        X1=columnNumber
```

```

        Y1=rowNumber
    Else
        If currentPixel=white
            Isline=false
            X2=columnNumber
            Y2=rowNumber
            Store line with X1,Y1,X2,Y2 coordinates
    If isline=true
        Isline=false
        X2=columnNumber
        Y2=rowNumber
        Store line with X1,Y1,X2,Y2 coordinates

```

Combined with a similar algorithm for extracting vertical lines it can effectively convert simple hexagonal mazes with wall width of one pixel to an array of lines. While it will work for every other image, the number of lines generated can be very big, which may slow down further functionality of software or prevent it from working at all.

10.3 Line Reduction

Two ways to reduce the number of generated lines were identified: reduce the number of edges being processed and recognise non axis aligned lines.

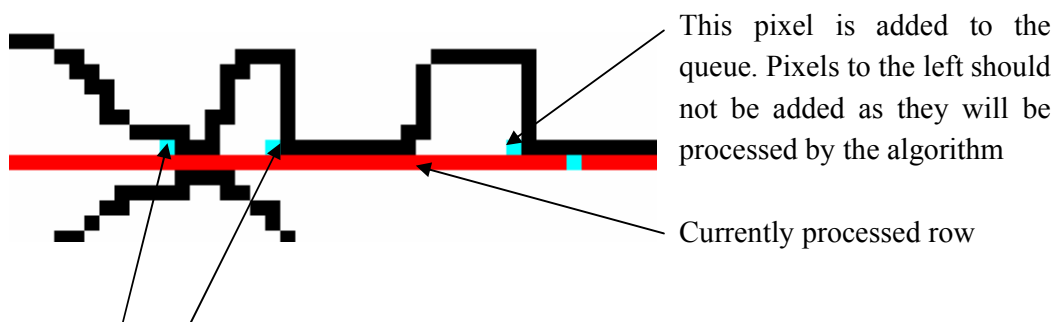
Because thresholding is applied to the whole image, it may generate data unnecessary for the swarm. This is because some edges or even whole objects may not be reachable for the swarm at all. To identify space that is reachable by agents flood fill was used. Because, java doesn't very well support recursive functions, due to stack space being severely constrained, a form of non recursive algorithm was used.

The original pseudocode was as following[2]:

1. Set Q to the empty queue.
2. If the color of *node* is not equal to *target-color*, return.
3. Add *node* to the end of Q .
4. For each element n of Q :
5. Set w and e equal to n .
6. Move w to the west until the color of the node to the west of w no longer matches *target-color*.
7. Move e to the east until the color of the node to the east of e no longer matches *target-color*.
8. Set the color of nodes between w and e to *replacement-color*.
9. For each node n between w and e :

10. If the color of the node to the north of n is *target-color*, add that node to the end of Q .
If the color of the node to the south of n is *target-color*, add that node to the end of Q .
11. Continue looping until Q is exhausted.
12. Return.

Then the process was slightly modified by introducing a watch over the obstacles above and below currently processed pixel. So for example if in the row above a number of black pixels is encountered following with a white pixel, this pixel will be added to the queue, but no following pixel will be added until another row of black pixels is encountered. This at least decreases the size of queue.

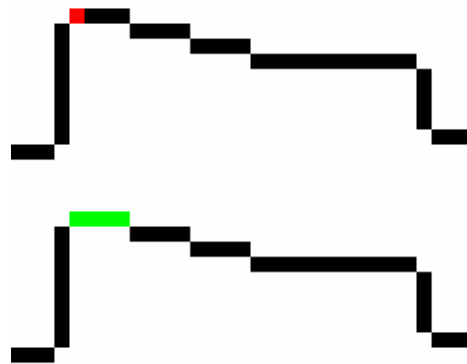


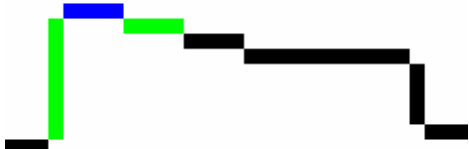



These points need to be queued as well, as their areas are not accessible from any other point in the queue.

10.4 Types of Lines

To identify all kinds of lines in the image and also to introduce a certain degree of approximation seamstress algorithm was written. The challenge was to identify patterns of pixels which all together depict edges. There also was need to allow the algorithm sometimes accept a small variation in pattern as to reduce the number of generated lines. The algorithm would work as following:

1. The image is scanned top-down row by row until a black pixel is encountered. This initialises seamstress sequence.
2. It is measured how far from the point black area stretches vertically and horizontally. The longest one is picked. In the example the horizontal distance is 4 and vertical is 1. Horizontal is picked.



3. Because we picked horizontal state, the algorithm horizontally scans to the left and right from the first line. In the example on the left the result is 1 and on the right is 4, which is the same as the original. If the two results didn't match the initial line, the algorithm would be stopped, and only the first value line would have been recorded. Similarly, if all 3 matched, this would mean it is the top of a curve and only the initial line would be recorded.
 
4. Because the next line is horizontal and to the right, the left most pixel of the first line is recorded as start. Also the right most pixel of the next line is recorded as an end.
 
5. Now the program starts to iterate through every line to the right of the currently processed line. The next horizontal line also turns out to be of size four, so the end pixel is now recorded as its right most pixel and the algorithm continues.
 
6. The next line turns out to be of different size, so the loop breaks and a line with the recorded start and end coordinates is added to the array.
 

There is similar logic for other scenarios, like vertical lines or a pattern that moves left. However for the algorithm to work a few more features are required. First of all, the pixels, for which a line was recorded, need to be erased. Also this wouldn't be a very good algorithm if it only accepted new lines if they had exactly the same length. It may be necessary to distinguish between lines of length 4 and 5, but it may be a good idea to accept line of length 19 if the original one is 20. One of the ways to do so is to divide one value by another and see that the result is within a certain range. Currently results that are greater than 0.5 and less than 1.5 are treated as positive. Cases when one line has length 1 and the other has length 2 are treated separately. Currently the maximum allowed number of positive 1 and 2 comparisons is 5. If this value is too big, round curves may get cut significantly, while if its too small, a lot of redundant lines would be generated.

The final versions of the algorithms are in the next chapter.

11. Conversion Algorithms

11.1 Introduction

To make the Swarm System perceive the environment that is encoded in an image, the image needs to undergo a series of conversions. Some of the steps are partially controlled by the user. Here are the algorithms used in these procedures in the order they are applied to the image.

11.2 Thresholding

While there are four methods in the user interface for thresholding the loaded image, they all really use modifications of the same algorithm. Whether the user participates in generating threshold value, or the program does this all by itself, the information algorithm really needs, is a colour, that we need to highlight in the image, precision – how close a colour of a pixel has to be to the colour we are looking for, to be picked by the algorithm, and finally the image. Normally if a pixel's colour is within the range, it will be coloured white, otherwise it is coloured black, but there is also an opposite scenario.

```
Isline=false
Length=0
For number of rows
  For number of columns
    R= red value of ImageArray at row & column
    G= green value of ImageArray at row & column
    B= blue value of ImageArray at row & column
    If R>(seekColor-precision) & G>(seekColor-precision) &
      B>(seekColor-precision) & R<(seekColor+precision) &
      G<(seekColor+precision) & B<(seekColor+precision)
      Store white point at column and row
    else
      Store black point at column and row
```

11.3 FloodFill

Next it is desirable to find out which areas can't be accessed by agents under any circumstances. To do this area that can be accessed is flood filled. This means that every pixel of the image that is not an obstacle is filled if it can be connected to the

start through other filled pixels. In this case two pixels are connected if one is to the left/right, on top or under the other pixel. Or $(x+1,y)$ $(x-1,y)$ $(x,y+1)$ $(x,y-1)$.

Create queue

Enqueue start position

While queue not empty

 Deque first point in the queue

 If point not filled

 Fill pixel in fillMatrix

 Length=1

 doUp=true

 doDown=true

 while point at row,column - Length is not edge

 if point is not obstacle

 fill pixel in fillMatrix in row, column - Length

 if row above is not outside edges

 if pixel above is black

 doUp=true

 else

 if doUp=true

 enqueue point above

 doUp=false

 if row below is not outside edges

 if pixel below is black

 doDown=true

 else

 if doDown=true

 enqueue point below

 doDown=false

 Length++

 Length=1

 doUp=true

 doDown=true

 while point at row,column + Length is not edge

 if point is not obstacle

 fill pixel in fillMatrix in row, column + Length

 if row above is not outside edges

 if pixel above is black

 doUp=true

 else

 if doUp=true

 enqueue point above

```

doUp=false
if row below is not outside edges
    if pixel below is black
        doDown=true
    else
        if doDown=true
            enqueue point below
            doDown=false
Length++

```

11.4 Simple Edge Detection

Now that in one array there is filled territory, that agents can access, and in another array of the same dimensions there are objects, it is very easy to detect edges of the filled area.

```

For number of rows
  For number of columns
    If fillMatrix[row+1][column] not filled &
      fillMatrix[row-1][column] not filled &
      fillMatrix[row][column+1] not filled &
      fillMatrix[row][column-1] not filled &

      objMatrix[row][column] store white point

```

Of course it is necessary to make sure tested rows and columns in fillMatrix are not outside the bounds. For that a special test function may be used or the edges of image processed separately.

A test function, that accepts row and column as parameters, may look like this:

```

If row<0 or row>maxRow or column<0 or column>maxColumn
  Return false
Else if imgArray[row][column] is black
  return false
Else return true

```

The connection between the returned data and the pixel data may be different, but most importantly it handles requests to the elements of array out of bounds.

11.5 Seamstress Algorithm

When edges are identified, it is time to turn them into an array of lines.
A function scans through the image top-down left to right and each time it encounters a black pixel, seamstress sequence is called:

```
For number of rows
  For number of columns
    If edgeMatrix[row][column] is black
      Call seamstress sequence
```

The sequence:

```
If pixel is surrounded by free space
  Return
X=Column
Y=Row
Iteration=0
Start=Point at Column and Row
isHorizontal=true
Calculate number of pixels right from start position
Calculate number of pixels down from start position
If right is longer
  isHorizontal=true
else
  isHorizontal=false
store end of picked route at X,Y
Finish=X,Y
Erase pixels between Start and Finish
rightOk=false
leftOk=false
If isHorizontal=true
  X,Y are Start
Else
  X,Y are Finish

If 1 pixel down and 1 pixel left from X,Y is black
  If isHorizontal=true
    Calculate number of pixels to the left from X,Y
  Else
    Calculate number of pixels to the down from X,Y
```

```

If fuzzyCompare number of pixels with number from the first line true
    leftOk=true
restore X,Y to end of first line
If 1 pixel down and 1 pixel right from X,Y is black
    If isHorizontal=true
        Calculate number of pixels to the right from X,Y
    Else
        Calculate number of pixels to the down from X,Y

If fuzzyCompare number of pixels with number from the first line true
    rightOk=true
if rightOk=false & leftOk=false
    store line Start-Finish
    return
if rightOk=true & leftOk=true
    store line Start-Finish
    return
X,Y = end of the line that has the same length as original line
Finish=X,Y
If isHor=true & leftOk=true
    X of Start + original length
Erase the used pixels
Dowhile fuzzyCompare original and new length is true
    If leftOk=true
        If isHorizontal=true
            Calculate number of lines to the left
        Else
            Calculate number of lines down
    If rightOk=true
        If isHorizontal=true
            Calculate number of lines to the right
        Else
            Calculate number of lines down
    If fuzzyCompare new and original length is true
        Finish=X,Y
        Erase used pixels
store line Start-Finish

```

The fuzzyCompare function is used to allow a new line of a different length to be accepted in two cases. First case is when original length divided by new length is < 1.5 and > 0.5 . The second case is when one of them is 1 and the other is 2. This however is allowed only a limited number of times, to avoid curves being sliced.

```
Freedom=1.5
maxPixelFreedom=5
if newLength=0
    return false
if (originalLength=1 & newLength=2) or (originalLength=1 &
    newLength=2)
    if iteration< maxPixelFreedom
        iteration++
        return true
if Freedom< originalLength/ newLength<(2- Freedom)
    return true
else
    return false
```

11.6 Refining the Algorithm

This approach however doesn't entirely surround accessible area with lines. It only seems so graphically. In fact there is always a gap of size one between every line. This happens because when one line finishes and its coordinates are saved, the other line would start from the next pixel, which is at least one point away. This cannot be observed from the graphical output, as there is no gap between two pixels that are next to each other. The approach above may be useful when the result is used in the form of pixels, but it can cause a number of problems when used to generate borders of a two dimensional world:

- An agent may escape the bounds of the world when one or several of its sensors hit the gaps and read that there is no wall, so that the agent continues to move right through the wall.
- An agent's movement may appear uneven, because sensors may abruptly read absence of one of the walls, which will cause the agent to move in the direction of the gap. The next read will probably find the wall and cause the agent to move away.
- False Junction Nodes may be created, because sensors may interpret gaps as tunnels.

To avoid these problems every section of a wall has to start at the same point where the other section has ended. Two approaches were attempted to implement this:

1. After the array of lines was generated, iterate through every line and check the distance from its start and end point to the start and end points of every other line in the array. This approach was found to be a bit time consuming in certain cases and showed less improvement than was expected. A decision was made to modify the seamstress algorithm instead.
2. Change the way start and end coordinates are stored by the seamstress algorithm:

```
If pixel is surrounded by free space
    Return
X=Column
Y=Row
Iteration=0
Start=Point at Column and Row
isHorizontal=true
Calculate number of pixels right from start position
Calculate number of pixels down from start position
If right is longer
    isHorizontal=true
else
    isHorizontal=false
store end of picked route at X,Y
Finish=X,Y
Erase pixels between Start and Finish
rightOk=false
leftOk=false
If isHorizontal=true
    X,Y are Start
Else
    X,Y are Finish

If 1 pixel down and 1 pixel left from X,Y is black
    AlternativeLeftStart = X,Y
    If isHorizontal=true
        Calculate number of pixels to the left from X,Y
    Else
        Calculate number of pixels to the down from X,Y
```

```

If fuzzyCompare number of pixels with number from the first line
    is true
        leftOk=true
restore X,Y to end of first line
If 1 pixel down and 1 pixel right from X,Y is black
    AlternativeRightStart = X,Y
    If isHorizontal=true
        Calculate number of pixels to the right from X,Y
    Else
        Calculate number of pixels to the down from X,Y

If fuzzyCompare number of pixels with number from the first line
    is true
        rightOk=true
if rightOk=false & leftOk=false
    If number of pixels to the left>0
        Finish=AlternativeLeftFinish
    Else
        Color point at Finish with Black
    If number of pixels to the right>0
        If isHorizontal=true
            Start=AlternativeRightFinish
        Else
            Color point at Start with Black
    store line Start-Finish
    return
if rightOk=true & leftOk=true
    If number of pixels to the left>0
        Finish=AlternativeLeftFinish
    If number of pixels to the right>0
        If isHorizontal=true
            Start=AlternativeRightFinish
    store line Start-Finish
    return
X,Y = end of the line that has the same length as original line
Finish=X,Y
If isHor=true & leftOk=true
    X of Start + original length
Erase the used pixels
Dowhile fuzzyCompare original and new length is true
    AlternatineFinish=X,Y

```

```

If leftOk=true
    If isHorizontal=true
        Calculate number of lines to the left
    Else
        Calculate number of lines down
If rightOk=true
    If isHorizontal=true
        Calculate number of lines to the right
    Else
        Calculate number of lines down
If fuzzyCompare new and original length is true
    Finish=X,Y
    Erase used pixels
Else
    If number of pixels>0
        Finish=AlternativeFinish
    Else
        Color point at Finish with Black
store line Start-Finish

```

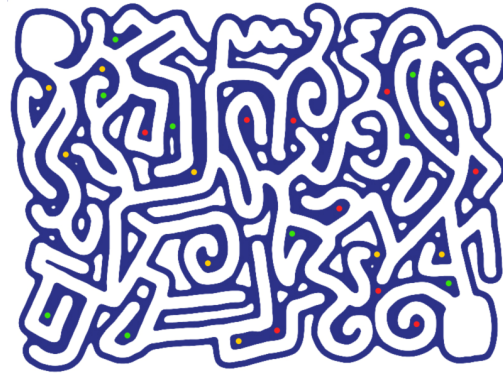
This approach significantly reduces the number of gaps, but doesn't eliminate them completely. This happens because it is not always possible to join several lines, like for example, when 3 lines join in one spot, but the algorithm generates slightly different coordinates for their end points. Therefore it will be necessary to take precautions and avoid the three problems listed above when populating the world with agents.

12. Effects of Conversion Approaches

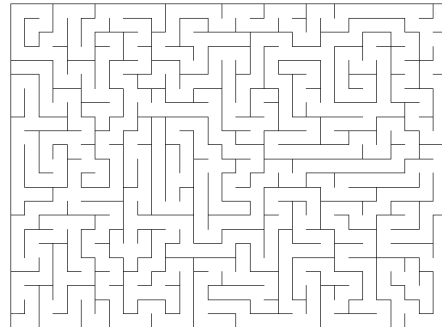
12.1 Benchmarking

Combinations of algorithms from the two previous chapters were applied to a number of different images to identify overall efficiency as well as trends specific to their nature. The following images were used:

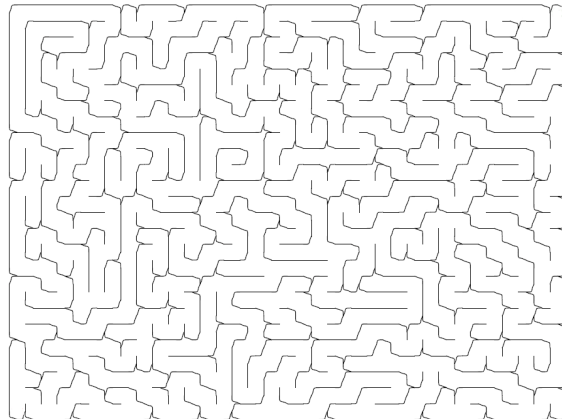
1. Sample maze. There are a large number of edges, almost all of which are curved. Dimensions: 800 x 600.



2. Orthogonal maze from www.onebillionmazes.com
All obstacles are straight line.
Dimensions: 705 x 525.



3. A kind of orthogonal maze with curved corners from www.onebillionmazes.com
All obstacles are straight line.
Dimensions: 905 x 683.



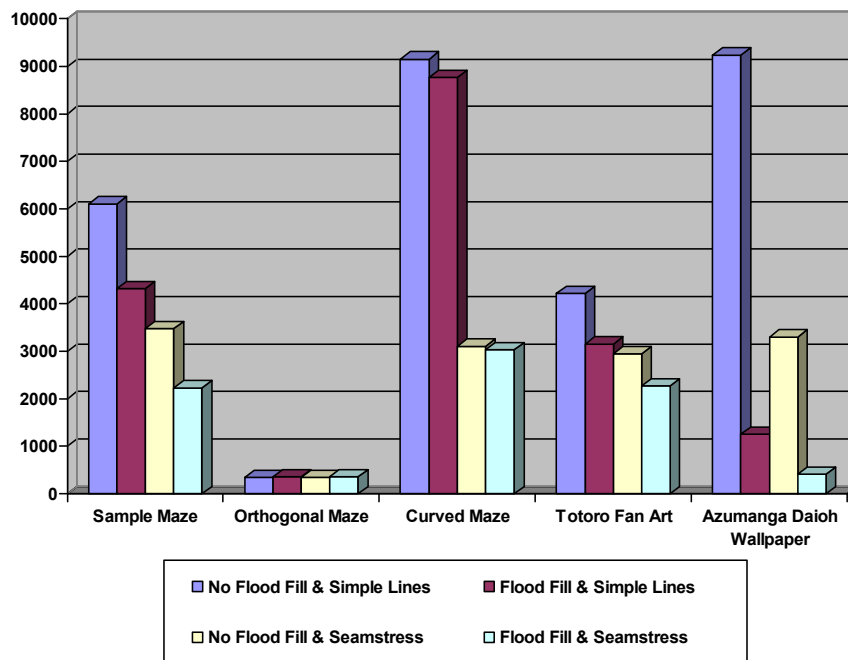
4. A fan art picture of little Totoro.
Because of the cartoon style, it offers easily identifiable obstacles, such as the black outline.
Dimensions: 800 x 600.



5. A wallpaper for Azumanga Daioh cartoon. Here the background can be used as free space and everything else as obstacles. The contours are smooth, sometimes straight, but their overall length is very small compared to the amount of information present in the picture. Dimensions: 1280 x 960.



The following chart shows the number of generated lines for each picture depending on what combination of conversions was applied.



12.2 Analysis

As the bar chart suggests, results for Sample maze benefit both from floodfill and Seamstress algorithms. Floodfill reduces the number of generated lines from almost 6000 to 4000, Seamstress even further to around 3300 and when used together they generate only 2000 lines, thus reducing line array 3 times.

The orthogonal maze has only straight lines, which is why it won't benefit from Seamstress algorithm, because there are no inaccessible areas Flood Fill won't have any effect either.

The curved maze is naturally very similar to the previous maze since both only have obstacles in the form of walls, so Flood Fill doesn't change the resulting array very much. As for Seamstress algorithm, it helps reduce array size by a factor of 3. This is because of the curved corners, which if treated as combinations of horizontal and vertical lines increase the number of lines significantly.

Results for Totoro picture are very similar to the results for the first maze, as the nature of their obstacles are similar: curved lines with thickness greater than 1.

Results for Azumanga cartoon are very different. First of all floodfill leaves only information about contours of the bubble and the two girls, throwing away details that are within these contours. This helps reduce the number of lines 9 times. Also the Seamstress algorithm works more efficiently here than in the previous example as there are a lot more long smooth line, reducing the number of lines 3 times. Altogether the two algorithms decrease the size of generated array almost 30 times.

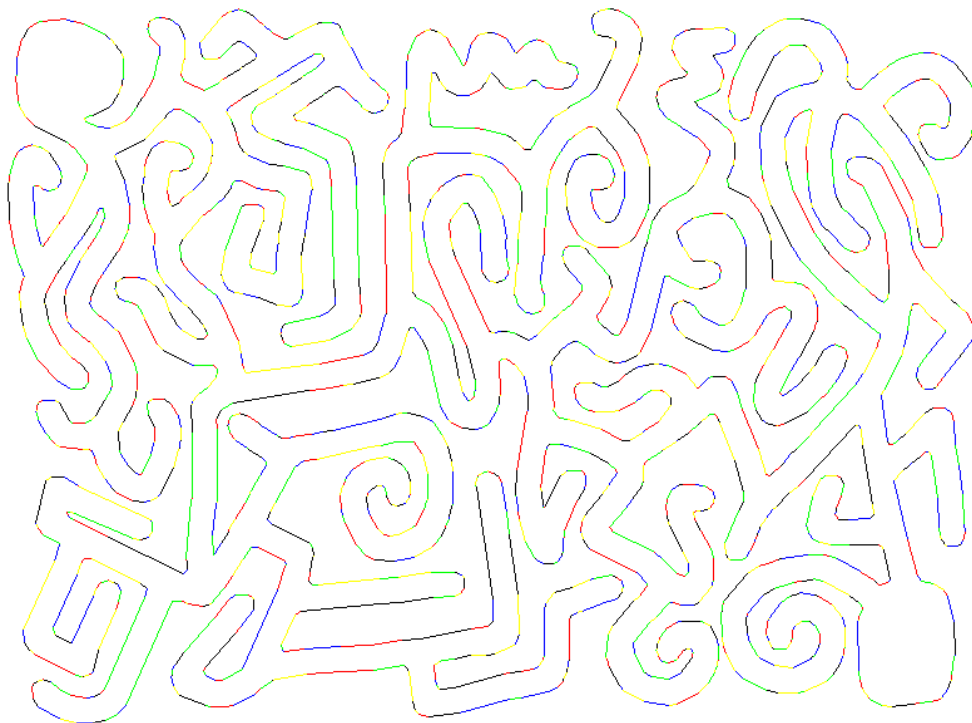


Figure 12.1. The result of conversion done on the sample maze.

13. Steering

13.1 Introduction

The UBQT swarm, named so after the word ‘ubiquity’, consists of a number of exactly the same agents. Each agent technically has access to the following information:

- The start position coordinates. The agent first appears in the world at this point.
- The finish position coordinates. It is the primary goal of every agent to reach this point.
- The coordinates of every wall in the world. An agent should not be able to go through any of the walls.

An agent also has access to positions of other agents, but this information is never used in the current model, and the array of nodes, which will be discussed later.

13.2 Swarm Data

As for information about itself, the following data is available:

- **Position** – coordinates of the current agent’s position in the world.
- **Direction** – a vector which points in the direction the agent is moving.
- **Orientation** – an angle at which the original agent graphics need to be moved to look in current direction. This information is almost never used for the purpose of steering and is only required for correct drawing of the agent. Orientation is recalculated from the direction every turn.
- **Sensors** – coordinates of agent’s sensors in the world as well as their directions in vector format. The coordinates are recalculated from the vectors every new turn.
- **Speed** at which an agent moves. Because no acceleration was introduced into the current model, agents always move with uniform velocity.
- **Sensors length** – this value is used to determine the size of sensors as well as some other distances. A sensor doesn’t always have this length, and if there is a need for a set of sensors to be longer or shorter than the other ones, their length will be set to $\text{senlength} / 2$ or $\text{senlength} * 0.3$. The same is true for some distances between an agent and objects around it. This is supposed to help scale the model easily.

13.3 Swarm Sensors

An agent also tracks some other information, which is mostly derived from information already discussed.

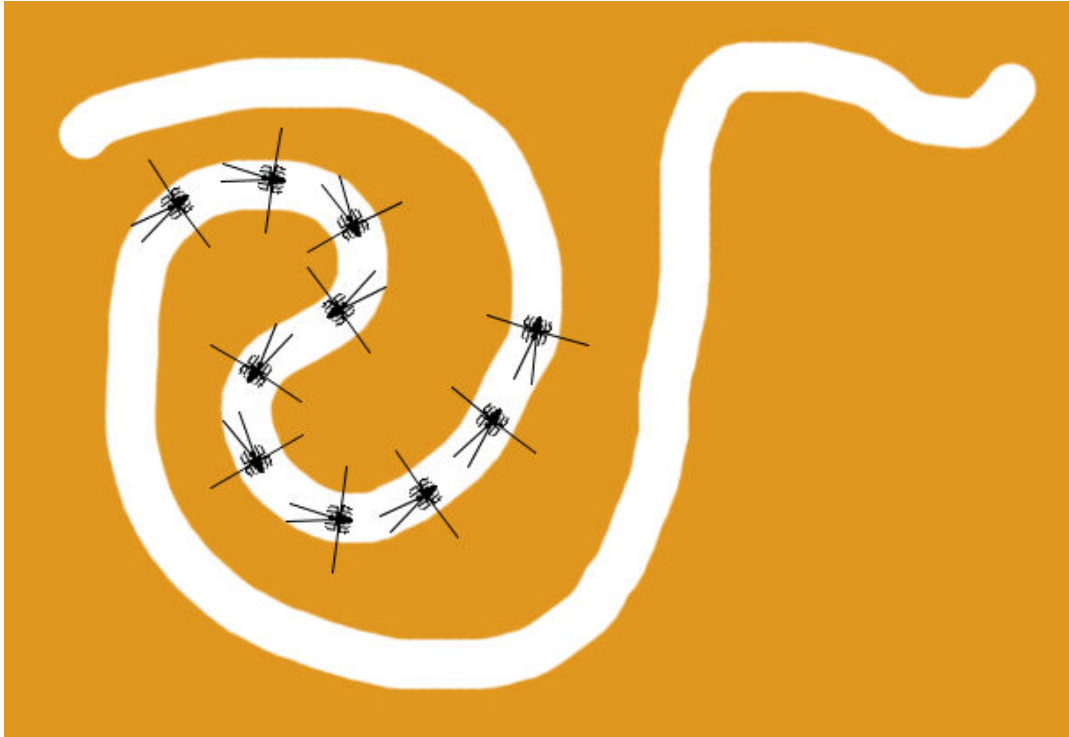


Figure 13.1. Final sensors layout

Without any interference from the world around, an agent would continue to move indefinitely at the same direction at uniform velocity unless he comes across the destination point. Because the very time the agent is spawned, the direction is set to point at the destination, the agent is more than likely to reach it. If various obstacles are added to the model, there arises a need to correctly identify those, which may affect agent's movement. Even though the entire array of lines is directly accessible to the agent, it is impossible to identify surrounding obstacles right away for a few reasons. First of all a line is represented with the coordinates of its start and finish points and to calculate how close a certain segment of this line is to the given position certain calculations are necessary. Secondly an agent isn't just a point, but rather has certain height and width. And finally the agent is constantly moving, so some sort of predicting behaviour is also necessary. To address these conditions a set of sensors was given to every agent. With the help of these sensors an agent is able to determine every point of time if there is a wall nearby in the direction sensor looks and if there is one, how close it is. The first model had four sensors of the same size: one in the front, one in the rear, one to the left and one to the right. Each would start in the centre of the agent. Later this model was altered by removing the rear sensor as it proved useless and making the front sensor twice as longer, because the agent's

movement wasn't predictive enough. This model was then replaced with a different one as a flow as found in it. Because there is a big gap between the front and the side sensors, an agent may be approaching an obstacle without ever noticing because it is a bit to the right or left from the front sensor and doesn't touch it. This may cause the agent to behave strangely once one of the side sensors touches the obstacle and the agent abruptly starts stirring away.

The final version of the model has two side sensors, just like in the previous versions and two front sensors instead of one. These are rotated by 10 and -10 degrees respectively from what was the front sensor. See Figure 13.1.

The sensors modify the agent's movement as follows. If a sensor reads a wall, this generates a force in the opposite direction, which gets stronger, the closer the agent gets to the wall. First there is one very strong force which points in the agent's current direction. It is represented with the **direction** vector, which is given strength by giving it a length of 50. If this value is much lower, the agent's movement will be very uneven, while a much greater value won't allow the agent to stir fast enough. Then readings are taken from the two front sensors. If both intersect with a wall, that means the agent is heading straight for the wall. In this case the two vectors that represent forces generated by the wall's 'anti-gravity' are added together and the resulting vector is given a force of 100 minus the distance. This force then is applied to the initial force at a 90 degree angle. If only one of the front sensors intersects with a wall, the force is only 20 units and it is applied at an angle of 20 degrees from direction. This allows an agent to smoothly stir away from an obstacle in front. After these manipulations readings from the side sensors are addressed. If both sensors hit walls, their force vectors are added together and then applied to the direction. If only one sensor reads a wall, its force applied to the direction vector right away. The strength of side forces is equal to the sensor length minus distance to the wall and is applied at 90 degrees. The side forces tend to make the agent stay at equal distance from walls when in tunnel or stir away from a wall, if outside. The result of all these manipulations is the modified direction vector, which is normalized and then multiplied by the speed.

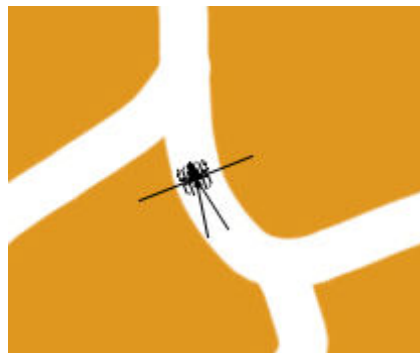


Figure 13.2. Only side sensors stir.

In the figure 13.2 an agent is exploring a tunnel. Currently his direction vector is the same with the direction he faces. Because the front sensors don't touch any walls now, we can expect that whatever steering will be applied to the agent, it won't change the direction significantly. Both side sensors intersect with walls and one of them has penetrated a little deeper than the other one. The two side-force vectors will be added together and a much weaker force pointing away from the closest wall will be generated. This force will be applied to the direction, which will make the agent slightly stir away from the wall the next turn.

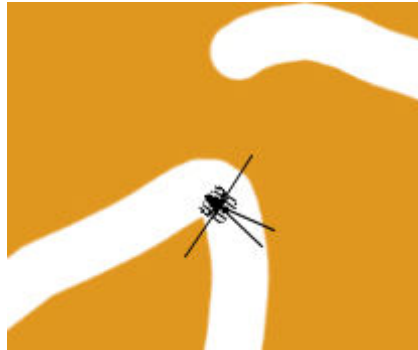


Figure 13.3. One of the front sensors contributes.

On Figure 13.3 one of the front sensors is touching a wall. This will generate a medium-strength force that will try to make the agent stir away from the wall. At the same time, the force generated by the side sensors will try to change the direction in the opposite direction, but compared to the front-sensor force it will be much weaker and will only weaken the stirring motion.

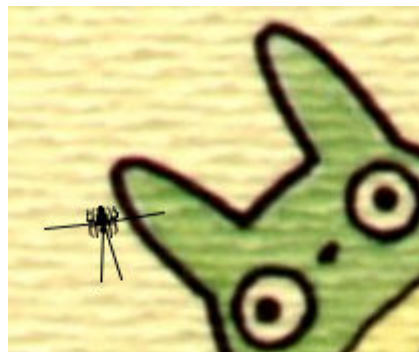


Figure 13.4. Only one sensor contributes to the change of direction.

On Figure 13.4 only one of the side sensors intersects with a wall. To stir away from the object a force will be generated, and because there is no opposing force from the other side, its effect will be relatively strong on the direction vector making the agent move in a curve.



Figure 13.5. No obstacles nearby.

On Figure 13.5 none of the sensors reads any obstacle, so the agent will continue to move in exactly the same direction.

Equipped with this simple logic, the agents are able to navigate complex environments fairly realistically, but without any certain purpose except for avoiding obstacles of course.

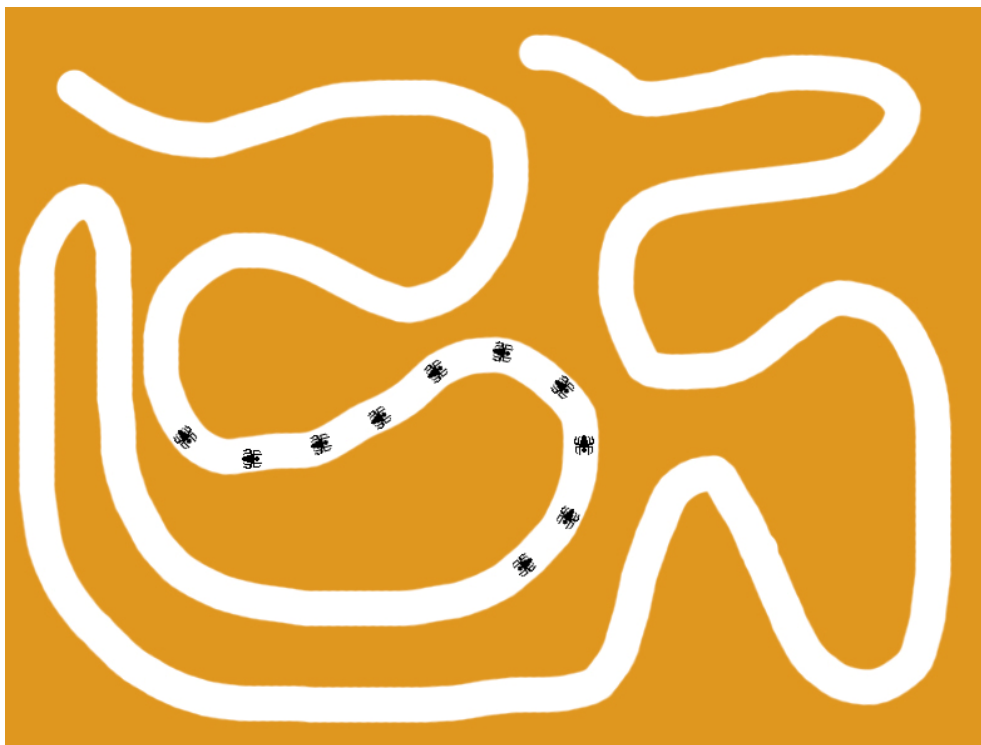


Figure 13.6. Swarm in a tunnel.

Steering will be enough for all the agents to reach destination in a tunnel without a single junction.

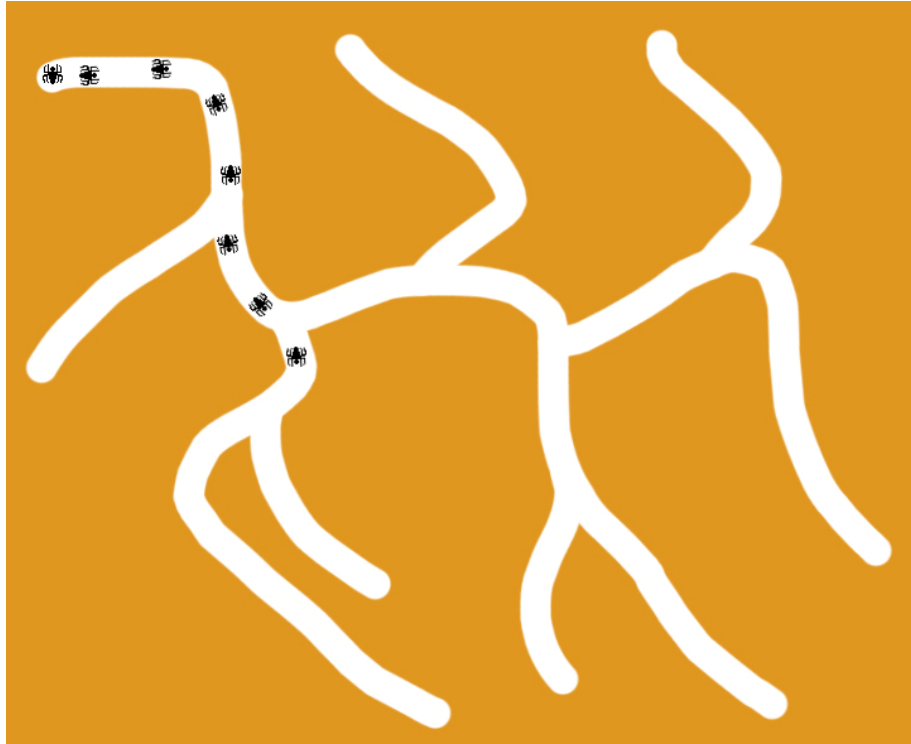


Figure 13.7. Junctions.

Presence of a number of junctions in the tunnel will significantly decrease the swarm's chances to reach the destination.



Figure 13.8. Open space

In an open environment the swarm equipped with just steering will be able to reach the destination point only by pure chance.

14. Nodes

14.1 Introduction

To utilize the swarm's potential as a Distributed AI in the search for destination point it is necessary to make agents take different routes when possible. To be able to do that an agent must be able to identify junctions in the first place. Also it is a good idea to store this information and share it with the other agents. For this purpose nodes were introduced into the environment. Whenever an agent spots a new junction, he will create a node there, which will contain its own position and information about possible routes. But first an agent needs to be able to identify such places. There are a few requirements which need to be met before a node can be created:

- A node can't be placed too close to a wall. Otherwise agents will be likely to stir away from a wall before they get a chance to get into one of the tunnels.
- It can't be placed too close to another node. This is because several points located nearby may meet all the other conditions and in fact will identify the same junction.
- It has to be a junction, which means at least three routes are available to take from the given point.

To avoid checking these conditions every turn, it is possible to use the already available information coming from the side sensors. If one of them identifies that the last time there was a wall, and now it is gone, the three checks can be activated to establish if this is a potential node. Once the conditions are met it is time to identify edges(vectors which point at possible routes). This is a very important process, as the direction vector, stored in an edge, will be the only information that will guide the agent into the right tunnel. Also there is a possibility a gap in the walls, mentioned before, is recognised as a valid edge. Therefore certain measures need to be taken to avoid that. Two different approaches were taken to identify edges and then with their help validate the new node.

14.1 Implementation of Nodes

The first approach used 18 rays generated to point in different directions from the current positions. Those, that didn't hit any walls were collected and tested if they weren't the result of gaps. The test implied rotating each vector by 7 and -7 degrees and then checking against the array of walls again. If neither of the shifted rays hit a wall, it was considered valid. Finally all the valid edges were counted and if the number was greater than 2, the node was added to the array. This method has several weaknesses which altogether make it hardly useful. First of all it often ignored valid routes and secondly the check didn't eliminate the gap problem completely, as invalid

edges would be counted in from time to time. However it delivered a valuable observation, that with a greater number of rays, valid routes would most of the time have room for at least two rays, while false edges would typically stand alone without any valid neighbours. Keeping that in mind and putting all the directions into the groups of neighbours it is possible to do two things: eliminate false directions by discarding groups of one, and identify vectors that go along the centre of a route by adding up all the direction vectors in a group. The new model used 36 rays instead of 18. Every time a ray didn't hit any wall, a new group would be started. Once that happens all the next rays would be included into the group until a ray hits a wall. That closes a group. Once a group is closed, if it includes only one ray, it is discarded, otherwise, the two edge rays are added together and the result is counted in as a valid edge. When all the 36 rays are analyzed, the number of identified edges is counted and if it is greater than 2, the node is added to the array. The pseudocode for the last approach is provided below:

```

Repeat 36 times
    store direction vector for ray in array dirs[]
    store coordinates of the outer end of the ray in ends[]
Create Node theNode
directionsInBranch=0
branches=0
for number of rays repeat
    intersects=false
    for number of lines in World
        if ray intersects line
            intersects=true
            break for loop
    if intersects=false
        increment directionsInBranch
        if isBranch=false
            isBranch=true
            start=current ray
        else
            finish=current ray
    else
        if isBranch=true
            isBranch=false
            if directionsInBranch>1
                add finish to start(as vectors)
                point=coordinates of the start end
                add new branch to the Node(start,point)

```

```
increment branches  
if branches>2 add theNode to the array of Nodes
```

There is some more information edges can store about the environment: whether this route was explored and whether it leads to a deadend. As a result the lifecycle of an edge can be separated into three stages:

1. The edge is created and marked as unexplored(green).
2. Once an agent takes the route the edge points at, the edge is marked as explored(yellow).
3. If an edge points at a deadend or at a route that ends only with deadends it is marked as dead(red). Either way, this edge is no longer of any interest to the swarm.

The node's lifecycle depends upon the state of its edges and is even simpler. Once the node is created it will stay that way (green) until the number of its edges that are alive drops below 2. Once that happens the node is marked as dead (red).

Now when an agent approaches a node, he will be given one of the unexplored directions or if none are left, one of the explored ones, on the condition he didn't just come from there. At the same time it will be checked if the agent has just returned from this node or from a dead node or from no node at all (in case he has just spawned). If so the edge pointing at the route he just took is marked as dead. Also if this brings the number of valid edges to one, this should kill the node. See Figure 14.1



Figure 14.1. First node identified.

A new node is identified, one of the possible routes is taken by the agent and the edge he just came from is marked as dead, because it points to a deadend. All of this should be enough to make the entire swarm reach the destination in a maze of almost any size and complexity on the condition that all the nodes and edges are identified correctly. See Figure 14.3.

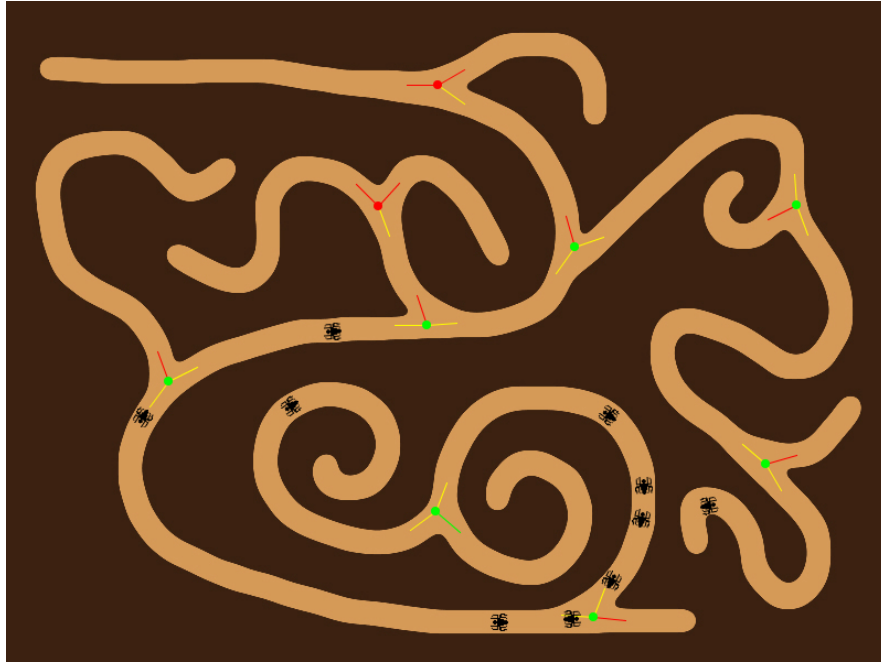


Figure 14.2: The swarm works together.

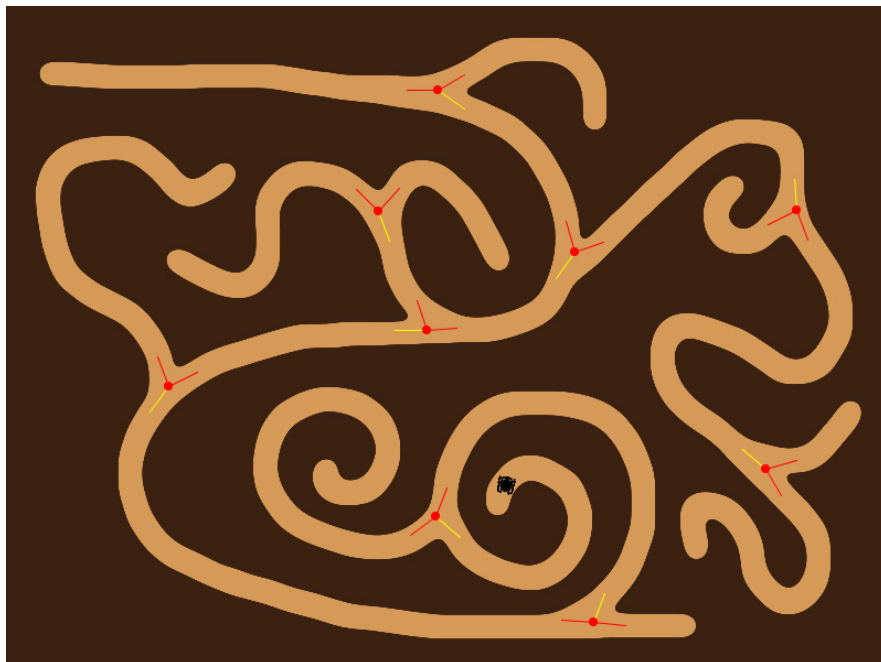


Figure 14.3. Maze solved.

If the entire maze was explored, every node will point to the destination

15. Conclusions and Future Work

15.1 Conclusions

From the work that was carried out along this project it was learned that images can very well be used as a media for storing 2D worlds. This can make the job of defining obstacles a whole lot easier. Because anyone can draw a picture in a few minutes without any special knowledge, this process can even be used as means of entertainment.

The UBQT Swarm isn't very universal right now and may miss a turn here and there, but with some further tuning it undoubtedly can be greatly improved. Currently its only weak point is the process of identifying junctions. The software's functionality could be greatly improved if this process was made to work without failure. Also effective exploration of open spaces is seen as the next logical step.

15.2 Future Work

1. Tune the Seamstress Algorithm to ensure there are no gaps in the generated model.
2. Improve detection of junctions.
3. Introduce logic for placing nodes in open spaces.
4. Add features such as statistics, timer, a log of work carried out by each agent.
5. Look into the use of other image conversion techniques for preparing an image for the use with the Seamstress Algorithm.
6. Make agents avoid collision with each other.
7. Add more features to the communication between agents.

Also several evolution paths are seen for the developed program:

- Further improve functionality.
- Turn it into a tool for creating game levels or test environments for other kinds of Autonomous Agents.
- Turn it into a game, where players firstly draw levels themselves. Simplification and automation of the image conversion process will be necessary.

Appendix

Source Code

UBQTMMain

```

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import javax.swing.JFileChooser.*;
import javax.swing.border.*;
import javax.imageio.*;
import javax.swing.WindowConstants;

import java.net.URL;

class UBQTMMain extends JFrame implements ActionListener, ChangeListener, MouseListener
{
    private static final long serialVersionUID = -6300968292870654269L;
    String lastImgFileName = "lasttimepath.dat";    // path to file that stores path
to the last image
    String samplefile = "gfx/sample.jpg";           // path to sample image
    String splashfile = "gfx/splash.jpg";           // path to filler image

    Vector world = new Vector();    // array to store all connected clients
    Engine engine;
    boolean[][] objMatrix;          // stores object matrix. used in conversion
    BufferedImage img = null;       // stores the original image
    int iteration;                  // used to control the number of allowed comparisons

    private JPanel upperpan, steppan, generalpan, mainpan, imagepan, statuspan;
    // all the general panels
    private JPanel step1, step2, step3, step4, step5;    // step panels that
replace each other when step changes
    private JLabel stepnum, step2num, step3num, step4num, step5num, statustit, imglab,
spc, tipit;
    private JButton closebut, contbut, backbut;    // buttons from top right panel
    private JTextArea infoarea, tipsarea;
    ImageIcon ic;
    // contains currently active image

    private JButton opennew, openlast, opensample;    // buttons for step 1

    private JButton identstartbut, identfinbut;    // buttons for step 2

    private JButton identcolorbut;    // button
    private JSlider precslider;    // slider and combo box
    private JComboBox objIdMode;    // for step 3

    private JButton viewpic;    // button for step 4

    private JButton runexlp;    // button for step 5

    String[] obsiStrings = { "Basic", "Smart", "by Space", "by Obstacle" };
    // combo box values

    final static String S1 = "Step1";
    final static String S2 = "Step2"; // strings that identify steps
    final static String S3 = "Step3";
    final static String S4 = "Step4";

```

```

final static String S5 = "Step5";

Cursor CrossCursor = new Cursor(Cursor.CROSSHAIR_CURSOR);      // cursors
Cursor DefaultCursor = new Cursor(Cursor.DEFAULT_CURSOR);
Cursor HGlassCursor = new Cursor(Cursor.WAIT_CURSOR);

int Step=1;              // step number
int precision = 55;      // precision value used in identifying objects
int imgCols=0;           // Number of horizontal pixels
int imgRows=0;           // Number of rows of pixels
Color defaultcolor = Color.WHITE;
boolean pickpixel=false; // true if program is in the state of picking a pixel
from current image

Point startpix = new Point(-1,-1);          // store locations of
Point finishpix= new Point(-1,-1);          // start and finish

public static void main(String[] args)
{
    UBQTMMain obj = new UBQTMMain();          //Instantiate an object of this class.
}

public UBQTMMain()              //constructor
{
    super("UBQT Swarm"); // sets title
    setLocation(50, 50);
    setSize(1024, 768);
    setBackground(Color.white);
    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    setContentPane(createContentPane());
    setVisible(true);
}

private Container createContentPane()
{
    setCursor(HGlassCursor);

    URL url = UBQTMMain.class.getResource(splashfile);

    ic = new ImageIcon(url); // load splash image that will occupy "ic" until
environment is loaded

    Container pane = new JPanel(new BorderLayout());

    step1 = new JPanel(); // populate step 1 panel
    step1.add(stepnum = new JLabel("STEP 1: Load Environment"));
    step1.add(opennew = new JButton("Load an image"));
    step1.add(openlast = new JButton("Load last image"));
    step1.add(opensample = new JButton("Load sample image"));
    opennew.addActionListener(this);
    openlast.addActionListener(this);
    opensample.addActionListener(this);

    step2 = new JPanel(); // populate step 2 panel
    step2 = new JPanel(new FlowLayout(FlowLayout.LEFT));
    step2.add(step2num = new JLabel("STEP 2: Identify Start and Finish"));
    step2.add(identstartbut = new JButton("Identify Start"));
    step2.add(identfinbut = new JButton("Identify Finish"));
    identstartbut.addActionListener(this);
    identfinbut.addActionListener(this);
    step2.setVisible(false);

    step3 = new JPanel(); // populate step 3 panel
    step3 = new JPanel(new FlowLayout(FlowLayout.LEFT));
    step3.add(step3num = new JLabel("STEP 3: Identify Obstacles"));
    step3.add(objIdMode = new JComboBox(obsiStrings)); //select which method is
used
    step3.add(identcolorbut = new JButton("Color")); // pick color from image (used
for last 2 methods)
    step3.add(precslider = new JSlider(JSlider.HORIZONTAL,0, 255, precision));
    // sets precision
    identcolorbut.setBackground(Color.WHITE);
    identcolorbut.setEnabled(false);
    precslider.addChangeListener(this);
    objIdMode.addActionListener(this);
    identcolorbut.addActionListener(this);
    step3.setVisible(false);
}

```

```

        step4 = new JPanel(); // populate step 4 panel
        step4 = new JPanel(new FlowLayout(FlowLayout.LEFT));
        step4.add(step4num = new JLabel("STEP 4: Generate Borders"));
        step4.add(viewpic = new JButton("View original image")); // switch between
image and lines
        viewpic.addActionListener(this);

        step5 = new JPanel(); // populate step 4 panel
        step5 = new JPanel(new FlowLayout(FlowLayout.LEFT));
        step5.add(step5num = new JLabel("STEP 5: Explore"));

        steppan = new JPanel(new CardLayout()); // this panel switches between
step panels
        steppan.add(step1,S1);
        steppan.add(step2,S2);
        steppan.add(step3,S3);
        steppan.add(step4,S4);
        steppan.add(step5,S5);

        generalpan = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        generalpan.add(backbut = new JButton("BACK"));
        generalpan.add(contbut = new JButton("CONTINUE")); // general navigation panel
        generalpan.add(spc = new JLabel(""));
        generalpan.add(closebut = new JButton("EXIT"));
        backbut.setEnabled(false);
        contbut.setEnabled(false);
        backbut.addActionListener(this);
        contbut.addActionListener(this);
        closebut.addActionListener(this);

        upperpan = new JPanel(new BorderLayout()); // upper part of the window
        upperpan.add(steppan, BorderLayout.WEST);
        upperpan.add(generalpan, BorderLayout.EAST);

        imagepan = new JPanel(new BorderLayout()); // space for the image
        imagepan.setBackground(Color.WHITE);
        imagepan.add(imglab = new JLabel(ic),BorderLayout.CENTER);
        imglab.addMouseListener(this);
        //imglab.setDoubleBuffered(true);

        statuspan = new JPanel(); // status panel
        defaultcolor = statuspan.getBackground();
        statuspan.setLayout(new BoxLayout(statuspan, BoxLayout.Y_AXIS));
        statuspan.setBorder(new LineBorder(defaultcolor, 5));
        statuspan.add(statustit = new JLabel("Information"));
        statuspan.add(new JScrollPane(infoarea = new JTextArea ("", 20, 15),
// information box

JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,JScrollPane.HORIZONTAL_SCROLLBAR_NEVER));
        infoarea.setEditable(false);
        infoarea.setLineWrap(true);
        infoarea.setWrapStyleWord(true);
        statuspan.add(new JLabel(""));
        statuspan.add(tipit = new JLabel("Tips")); // tips box
        statuspan.add(new JScrollPane(tipsarea = new JTextArea ("", 5, 15),

JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,JScrollPane.HORIZONTAL_SCROLLBAR_NEVER));
        tipsarea.setEditable(false);
        tipsarea.setLineWrap(true);
        tipsarea.setWrapStyleWord(true);

        mainpan = new JPanel(new BorderLayout());
        mainpan.add(imagepan, BorderLayout.CENTER);
        mainpan.add(statuspan, BorderLayout.EAST);
        pane.add(upperpan, BorderLayout.NORTH);
        pane.add(mainpan, BorderLayout.CENTER);
        infoarea.setText(infoarea.getText() + "Environment ready.\n"); // write into
info and tips boxes
        tipsarea.setText("Please, load an image you would like to work with. Use either
of the 3 options.");
        setCursor(DefaultCursor);
        return(pane);
    }

    public void initStep1() // only used for stepping backward
    {

```

```

        tipsarea.setText("Please, load an image you would like to work with. Use either
of the 3 options.");
        backbut.setEnabled(false);
        contbut.setEnabled(true);
        ic = new ImageIcon(img);
        imglab.setIcon(ic);
        imglab.setVisible(true);
        showPanel(S1); // change toolbar to step 1
    }

    public void initStep2() // put environment to the state of step 2
    {
        tipsarea.setText("Here you need to set two points on the image: where the swarm
will spawn and " +
        "where it will try to move. Press the corresponding button and
then mark the point" +
        " with a click on the image. You need to set both start and
destination.");
        backbut.setEnabled(true);
        pickpixel=false;
        if (startpix.getX()!=-1 & finishpix.getX()!=-1) contbut.setEnabled(true);
        else contbut.setEnabled(false);
        drawSF();
        showPanel(S2); // change toolbar to step 2
    }

    public void initStep3() // put environment to the state of step 3
    {
        imagepan.setBackground(Color.WHITE);
        backbut.setEnabled(true);
        contbut.setEnabled(true);
        showPanel(S3); // change toolbar to step 3
        renderObjs(); // identify objects
    }

    public void initStep4() // put environment to the state of step 4
    {
        setCursor(HGClassCursor);
        pickpixel=false;
        viewpic.setText("View original image");
        tipsarea.setText("Now you can see the bounds of the world we generated. " +
        "Differently colored fragments of borders " +
        "stand for different lines. It is desirable that they precisely
define the bounds of object " +
        "but that there are not too many of them. If there seem too
lines, " +
        "you may want to redefine objects by going a step back. ");
        backbut.setEnabled(true);
        contbut.setEnabled(true);
        showPanel(S4); // change toolbar to step 4
        boolean[][] fillMatrix;
        fillMatrix = new boolean[imgRows][imgCols]; // matrix used in fill algorithm

        fill((int)startpix.getX(),(int)startpix.getY(),fillMatrix); // fill
space accesible from

        // start position
        getOnlyOutline(fillMatrix); // leave only borders of filled space

        toLines(); // convert borders to array of lines (world)

        drawLines(); // display the lines on screen

        if(fillMatrix[startpix.y][startpix.x]==false)
        {
            contbut.setEnabled(false); // no area defined
            infoarea.setText(infoarea.getText() + "Spawn point is positioned inside
an object. Impossible to proceed!\n");
        }
        else
        {
            contbut.setEnabled(true);
            if(fillMatrix[finishpix.y][finishpix.x]==true) // it may be
possible to reach destination
            {

```

```

        infoarea.setText(infoarea.getText() + "Destination is within
reachable area.\n");
    }
    else    // will be impossible to reach destination
    {
        infoarea.setText(infoarea.getText() + "Destination is NOT
within reachable area.\n");
    }

    infoarea.setText(infoarea.getText() + "Lines generated: " +
world.size() + ".\n");
}

setCursor(DefaultCursor);
}

public void initStep5()    // put environment to the state of step 3
{
    imagepan.setBackground(Color.WHITE);
    backbut.setEnabled(true);
    contbut.setEnabled(false);
    showPanel(S5);    // change toolbar to step 3
}

public void drawLines()
{
    imagepan.setBackground(Color.BLACK);    // its easier to see colourful
lines on black
    BufferedImage modImage = new
BufferedImage(imgCols,imgRows,BufferedImage.TYPE_INT_ARGB);
    Graphics2D gp = modImage.createGraphics();
    gp.setColor(Color.BLACK);
    gp.fillRect(0,0,imgCols,imgRows);

    Line li;
    Color[] ArrC = {Color.BLUE , Color.GREEN, Color.RED, Color.YELLOW,
Color.WHITE};
    // colours used to draw lines

    int cc=0;

    for(int i=0; i < world.size(); i++)    // iterate through all lines
    {
        boolean isOnly=true;
        li=(Line) world.get(i);

        do
        {
            isOnly=true;
            cc++;    // make sure ends of the current line are not
            if(cc==ArrC.length)cc=0;    // next to the same color
            //check pixels around the start of the line
            if(matchColor(li.getx1()-1,li.gety1()-1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx1(),li.gety1()-1,ArrC[cc], modImage)==true)
isOnly=false;
            if(matchColor(li.getx1()+1,li.gety1()-1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx1()-1,li.gety1(),ArrC[cc], modImage)==true)
isOnly=false;
            if(matchColor(li.getx1()+1,li.gety1()-1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx1()-1,li.gety1()+1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx1(),li.gety1()+1,ArrC[cc], modImage)==true)
isOnly=false;
            if(matchColor(li.getx1()+1,li.gety1()+1,ArrC[cc],
modImage)==true) isOnly=false;
            //check pixels around the end of the line
            if(matchColor(li.getx2()-1,li.gety2()-1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx2(),li.gety2()-1,ArrC[cc], modImage)==true)
isOnly=false;
            if(matchColor(li.getx2()+1,li.gety2()-1,ArrC[cc],
modImage)==true) isOnly=false;
            if(matchColor(li.getx2()-1,li.gety2(),ArrC[cc], modImage)==true)
isOnly=false;

```

```

        if(matchColor(li.getx2()+1,li.gety2()-1,ArrC[cc],
modImage)==true) isOnly=false;
        if(matchColor(li.getx2()-1,li.gety2()+1,ArrC[cc],
modImage)==true) isOnly=false;
        if(matchColor(li.getx2(),li.gety2()+1,ArrC[cc], modImage)==true)
isOnly=false;
        if(matchColor(li.getx2()+1,li.gety2()+1,ArrC[cc],
modImage)==true) isOnly=false;
    }while(isOnly==false);

    gp.setColor(ArrC[cc]); // draw line
    gp.drawLine(li.getx1(),li.gety1(),li.getx2(),li.gety2());
    //System.out.println(li.getx1() + " " + li.gety1() + " " + li.getx2() +
" " + li.gety2());
    }

    ic = new ImageIcon(modImage); // display image
    imglab.setIcon(ic);
}

public boolean matchColor(int x,int y, Color c, BufferedImage modImage) //
compare pixel colour
{
    // with another colour
    if (x>=0 & y>=0 & x<imgCols & y<imgRows)
    {
        int pixel = modImage.getRGB(x,y); // get RGB value of
the pixel
        int cc=c.getRGB(); // get RGB of the colour passed
        if(cc==pixel) return true; // compare them
    }
    return false;
}

public void toLines() // convert edges to lines
{
    world.clear(); // clear array of lines
    for(int row = 0;row < imgRows;row++) //Extract a row of pixel data into a
temporary array
    {
        for(int col = 0;col < imgCols;col++)
        {
            if (objMatrix[row][col]==true) seamstress(row,col); // if
encountered black,
        }

        // init seamstress algorithm
    }
}

public void seamstress(int row, int col) // traces edges and converts them into
lines
{
    if(analyzePix(row-1,col-1)==false & analyzePix(row,col-1)==false &
analyzePix(row+1,col-1)==false
    & analyzePix(row-1,col)==false & analyzePix(row+1,col)==false
    & analyzePix(row-1,col+1)==false & analyzePix(row,col+1)==false
    & analyzePix(row+1,col+1)==false)
    {
        objMatrix[row][col]=false;
        return; // don't record as its just one dot
    }

    int x=col;
    int y=row;

    Point start = new Point(col,row); // record start of the line
    Point finish;

    Point altfin, altfin1;

    int hLen=0; // stores horizontal distance
    int vLen=0; // stores horizontal distance
    int rx=0,ry=0,lx=0,ly=0; // temporarily store progress

    int Length=0; // stores length of the initial section of line
    boolean isHor=true; // true if initial line is horizontal

```

```

iteration=0;    // this is new line, so number of allowed iterations is reset

while(analyzePix(y,x)==true) // calc number of black pixels horizontally from
{
    // the start position
    hLen++;
    x++;
}
x--;
rx=x;
x=col;

while(analyzePix(y,x)==true) // calc number of black pixels vertically from
{
    // the start position
    vLen++;
    y++;
}
y--;
ry=y;
y=row;

if(hLen>vLen) // check which route is longer and pick the longest
{
    Length=hLen;
    isHor=true;
    x=rx; // go back to the horizontal route
}
else
{
    Length=vLen;
    isHor=false;
    y=ry; // go back to the vertical route
}

eraseBack(isHor,x,y,Length); // erase this segment and record finish position
finish = new Point(x,y);

int lLength=0;
boolean rightOk=false;
boolean leftOk=false;

rx=x; // record current position
ry=y;
lx=x;
ly=y;

altfin = new Point(x,y);

if(analyzePix(y+1,col-1)==true) // check segments to the left
{
    x=col-1;
    y++;
    altfin = new Point(x,y);

    if (isHor==true) // horizontally
    {
        while(analyzePix(y,x)==true)
        {
            lLength++;
            x--;
        }
        x++;
    }
    else // vertically
    {
        while(analyzePix(y,x)==true)
        {
            lLength++;
            y++;
        }
        y--;
    }

    if(fuzzyCompare(Length,lLength)==true) // compare intial and new length
    {
        leftOk=true;
        lx=x;
        ly=y;
    }
}

```



```

    }
}

int rLength=0;

x=rx;
y=ry;

altfin1 = new Point(x,y);

if(analyzePix(y+1,x+1)==true) // check segments to the right
{
    x++;
    y++;
    altfin1 = new Point(x,y);

    if (isHor==true) // horizontally
    {
        while(analyzePix(y,x)==true)
        {
            rLength++;
            x++;
        }
        x--;
    }
    else // vertically
    {
        while(analyzePix(y,x)==true)
        {
            rLength++;
            y++;
        }
        y--;
    }

    if(fuzzyCompare(Length,rLength)==true) // compare intial and new length
    {
        rightOk=true;
        rx=x;
        ry=y;
    }
}

//no similar neighbors
if (rightOk==false & leftOk==false)
{
    if (rLength>0)
    {
        finish=altfin1;
    }
    else
    {
        objMatrix[finish.y][finish.x]=true;
    }

    if (lLength>0)
    {
        if(isHor==true)start=altfin;
    }
    else
    {
        objMatrix[start.y][start.x]=true;
    }

    world.add(new
Line((int)start.getX(),(int)start.getY(),(int)finish.getX(),(int)finish.getY()));
    return;
}

//both neighbors similar, so its still a line
if (rightOk==true & leftOk==true)
{
    if (rLength>0)
    {
        finish=altfin1;
    }
}

```

```

        if (lLength>0)
        {
            if(isHor==true) start=altfin;
        }

        world.add(new
Line((int)start.getX(), (int)start.getY(), (int)finish.getX(), (int)finish.getY()));
        return;
    }

    if(rightOk==true)
    {
        finish = new Point(rx,ry);          // erase right segment and record
finish position
        eraseBack(isHor,rx,ry,rLength);
        x=rx;
        y=ry;
    }
    else
    {
        finish = new Point(lx,ly);          // erase left segment and record
finish position
        if(isHor==true) start.x=start.x+Length-1;    // adjust start position
        eraseBack(isHor,lx,ly,lLength);
        x=lx;
        y=ly;
    }

    do          // go through other segments in the same direction
    {
        rLength=0;
        if(leftOk==true)          // check segments to the left
        {
            x--;
            y++;
            altfin = new Point(x,y);

            if (isHor==true)
            {
                while(analyzePix(y,x)==true)
                {
                    rLength++;
                    x--;
                }
                x++;
            }
            else
            {
                while(analyzePix(y,x)==true)
                {
                    rLength++;
                    y++;
                }
                y--;
            }
        }
        else // check segments to the right
        {
            x++;
            y++;
            altfin = new Point(x,y);

            if (isHor==true)
            {
                while(analyzePix(y,x)==true)
                {
                    rLength++;
                    x++;
                }
                x--;
            }
            else
            {
                while(analyzePix(y,x)==true)
                {

```

```

        rLength++;
        y++;
    }
    y--;
}

if(fuzzyCompare(Length,rLength)==true)// compare intial and new length
{
    finish = new Point(x,y);
    eraseBack(isHor,x,y, rLength);
}
else
{
    if (rLength>0)
    {
        finish=altfin;
    }
    else
    {
        objMatrix[finish.y][finish.x]=true;
    }
}

}while(fuzzyCompare(Length,rLength)==true);

world.add(new
Line((int)start.getX(),(int)start.getY(),(int)finish.getX(),(int)finish.getY()));
return;
}

public boolean fuzzyCompare(int initial, int match)    // compare two distances
in a fuzzy way
{
    double freedom=1.5;                                // how great can difference in length
    int maxPerPixelFreedom=5;                          // max number of 1->2 & 2->1 allowed
    if(match==0) return false;

    if((initial==2 & match==1) | (initial==1 & match==2)) // allow 1->2 & 2->1 a
limited number of times
    {
        if(iteration<maxPerPixelFreedom)
        {
            iteration++;
            return true;
        }
    }

    if((double)initial/(double)match < freedom & (double)initial/(double)match >
(2.0 - freedom))
    {
        return true;    // if difference is less than freedom value -> allow
    }
    else return false;
}

public void eraseBack(boolean isHor, int x, int y, int len) // erase used up
pixels
{
    int cur=0;                                          // distance from start
    boolean plus=false;                               // if x/y value increments or decrements
    if (isHor==true)
    {
        if(analyzePix(y,x+1)==true)plus=true;
        while(analyzePix(y,x)==true & cur<len)
        {
            cur++; // if the pixel is not part of a perpendicular, remove it
            if(analyzePix(y+1,x)==false | analyzePix(y-
1,x)==false)objMatrix[y][x]=false;
            if(plus==true)x++;
            else x--;
        }
    }
    else
    {
        if(analyzePix(y+1,x)==true)plus=true;
        while(analyzePix(y,x)==true & cur<len)

```

```

        {
            cur++; // if the pixel is not part of a perpendicular, remove it
            if(analyzePix(y,x+1)==false | analyzePix(y,x-1)==false)
objMatrix[y][x]=false;
            if(plus==true)y++;
            else y--;
        }
    }
}

public void plugGaps()
{
    for(int o=0; o < world.size(); o++)          // iterate through all lines
    {

        Line cur=(Line)world.get(o);

        Point up1= new Point(cur.x1,cur.y1);
        Point up2= new Point(cur.x2,cur.y2);

        for(int i=0; i < world.size(); i++)      // iterate through all lines
        {
            Line che=(Line)world.get(i);
            if(up1.distance(che.x1,che.y1)<2)
            {
                cur.x1=che.x1;
                cur.y1=che.y1;
            }

            else if(up1.distance(che.x2,che.y2)<2)
            {
                cur.x1=che.x2;
                cur.y1=che.y2;
            }

            if(up2.distance(che.x1,che.y1)<2)
            {
                cur.x2=che.x1;
                cur.y2=che.y1;
            }

            else if(up2.distance(che.x2,che.y2)<2)
            {
                cur.x2=che.x2;
                cur.y2=che.y2;
            }
        }

        world.remove(o);
        world.add(cur);
    }
    System.out.println("Finishing gaps");
}

public boolean analyzePix(int row, int col)      // analyze color and accept
positions outside array
{
    if(row<0 | row>=imgRows | col<0 | col>=imgCols) return false;
    else if(objMatrix[row][col]==false) return false;
    else return true;
}

public boolean analyzePixRev(int row, int col)   // analyze color and accept
positions outside array
{
    if(row<0 | row>=imgRows | col<0 | col>=imgCols) return true;
    else if(objMatrix[row][col]==false) return false;
    else return true;
}

public void fill(int x,int y,boolean[][] fillMatrix)    //nonrecursive flood fill
{
    int cu[]={x,y};
    PointQueue Q = new PointQueue();                  // create queue
    if (objMatrix[y][x]==true) return;
    Q.enqueue(new Point(x,y));                         // start position enqueued
}

```

```

while(Q.size()>0)                                // process queue until its empty
{
    Point p=Q.dequeue();
    cu[0]=(int)p.getX();
    cu[1]=(int)p.getY();
    if(objMatrix[cu[1]][cu[0]]==false & fillMatrix[cu[1]][cu[0]]!=true) //
if not filled
    {
        fillMatrix[cu[1]][cu[0]]=true;           // fill pixel
        int le=0;
        boolean doUp=true;    // store information if there is need to
enqueue upper and lower pixel
        boolean doDown=true;
        while((cu[0]-le)>0)    // do until edge is reached to the left
        {
            if(objMatrix[cu[1]][cu[0]-le]==true) // if obstacle -
break loop
                break;
            fillMatrix[cu[1]][cu[0]-le]=true;    // else fill pixel
in fillMatrix
            if(cu[1]>1)                // if upper row exists
            {
                if(objMatrix[cu[1]-1][cu[0]-le]==true) // if
upper pixel is black
                    {
                        doUp=true;
                    }
                    else
                    {
                        if(doUp==true) // if new area
                        {
                            Q.enqueue(new Point(cu[0]-
le,cu[1]-1));    // enqueue
                            doUp=false;
                        }
                    }
                }
            if(cu[1]<imgRows-1)        // if lower row exists
            {
                if(objMatrix[cu[1]+1][cu[0]-le]==true)
                // if lower pixel is black
                    {
                        doDown=true;
                    }
                    else
                    {
                        if(doDown==true)    // if new area
                        {
                            Q.enqueue(new Point(cu[0]-
le,cu[1]+1));    // enqueue
                            doDown=false;
                        }
                    }
                }
            }
            le++;
        }
        le=1;
        doUp=true;
        doDown=true;
        while((cu[0]+le)<(imgCols-1)) // do until edge is reached to
the right
        {
            if(objMatrix[cu[1]][cu[0]+le]==true) // if obstacle -
break loop
                break;
            fillMatrix[cu[1]][cu[0]+le]=true;    // else fill pixel
in fillMatrix
            if(cu[1]>1)                // if upper row exists
            {
                if(objMatrix[cu[1]-1][cu[0]+le]==true)
                // if upper pixel is black
                    {
                        doUp=true;
                    }
                    else
                    {

```

```

                                if(doUp==true)           // if new area
                                {
                                    Q.enqueue(new
Point(cu[0]+1e,cu[1]-1));           // enqueue
                                    doUp=false;
                                }
                                }
                                if(cu[1]<imgRows-1)           // if lower row exists
                                {
                                    if(objMatrix[cu[1]+1][cu[0]+1e]==true)
// if lower pixel is black
                                    {
                                        doDown=true;
                                    }
                                    else
                                    {
                                        if(doDown==true) // if new area
                                        {
                                            Q.enqueue(new
Point(cu[0]+1e,cu[1]+1));           // enqueue
                                            doDown=false;
                                        }
                                    }
                                }
                                le++;
                            }
                        }
                    }

    public void getOnlyOutline(boolean[][] fillMatrix)           // leave pixels that are
only next to filled pixels
    {
        for(int row = 1;row < imgRows-1;row++) // process image without edges
        {
            for(int col = 1;col < imgCols-1;col++)
            {
                if (objMatrix[row][col]==true)
                {
                    if (fillMatrix[row+1][col]!=true & fillMatrix[row-
1][col]!=true
                                & fillMatrix[row][col+1]!=true &
fillMatrix[row][col-1]!=true)
                    {
                        objMatrix[row][col]=false;
                    }
                }
            }
        }
        for(int row = 1;row < imgRows-1;row++)           // process edges
        {
            if(fillMatrix[row][1]==true) objMatrix[row][0]=true;
            else if(objMatrix[row][0]==true & fillMatrix[row][1]==false)
            {
                objMatrix[row][0]=false;
            }
            if(fillMatrix[row][imgCols-2]==true) objMatrix[row][imgCols-1]=true;
            else if(objMatrix[row][imgCols-1]==true & fillMatrix[row][imgCols-
2]==false)
            {
                objMatrix[row][imgCols-1]=false;
            }
        }
        for(int col = 1;col < imgCols-1;col++)           // process edges
        {
            if(fillMatrix[1][col]==true) objMatrix[0][col]=true;
            else if(objMatrix[0][col]==true & fillMatrix[1][col]==false)
            {
                objMatrix[0][col]=false;
            }
            if(fillMatrix[imgRows-2][col]==true) objMatrix[imgRows-1][col]=true;
            else if(objMatrix[imgRows-1][col]==true & fillMatrix[imgRows-
2][col]==false)
            {
                objMatrix[imgRows-1][col]=false;
            }
        }
    }

```

```

    }

}

public int[] getPixColor(Point p) // returns array with RGB value (no alpha
channel!)
{
    int pixel = img.getRGB((int)p.getX(),(int)p.getY());
    int[] color={0,0,0};
    color[0] = (pixel >> 16) & 0xff;
    color[1] = (pixel >> 8) & 0xff;
    color[2] = (pixel >> 0) & 0xff;
    return color;
}

public void renderObjs() // thresholding and drawing objects
{
    int[] picArr = getPicArray();

    objMatrix = new boolean[imgRows][imgCols];
    int a,r,g,b;
    int length=0; // length of a line
    boolean isLine=false; // tells if currently within a line
    BufferedImage modImage = new
BufferedImage(imgCols,imgRows,BufferedImage.TYPE_INT_ARGB); // new image

    Graphics2D gp = modImage.createGraphics();
    gp.setColor(Color.BLACK);

    if (objIdMode.getSelectedIndex()==0) // BASIC mode
    {
        int greyval = 255 - precision;
        for(int row = 0;row < imgRows;row++)
        {
            int[] aRow = new int[imgCols];
            for(int col = 0; col < imgCols;col++)
            {
                int element = row * imgCols + col;
                aRow[col] = picArr[element]; // extract 1 row of pixels
            }

            for(int col = 0;col < imgCols;col++)
            {
                a = (aRow[col] >> 24) & 0xFF;
                r = (aRow[col] >> 16) & 0xFF; // get alpha + RGB value
                g = (aRow[col] >> 8) & 0xFF;
                b = (aRow[col] & 0xFF);

                if ((r < greyval) & (g < greyval) & (b < greyval)) //
test if color is above or below threshold
                {
                    objMatrix[row][col]=true; // store black point
                    if (isLine==true)
                    {
                        length++;
                    }
                    else
                    {
                        isLine=true;
                    }
                }
                else
                {
                    objMatrix[row][col]=false; // store white point
                    if (isLine==true)
                    {
                        isLine=false;
                        gp.drawRect(col-length-1,row,length,0);
                        // draw line (rectangles are used as
                        // they are faster than lines)
                        length=0;
                    }
                }
            }
        }
    }
}

```

```

        if (isLine==true)
        {
            isLine=false;
            gp.drawRect(imgCols-length-1,row,length,0); // draw
line
        }
    }
}
else if (objIdMode.getSelectedIndex()==1 | objIdMode.getSelectedIndex()==2)
// SMART or SPACE mode
{
    int[] sti={0,0,0};
    if (objIdMode.getSelectedIndex()==1)
    {
        sti = getPixColor(startpix);
        int[] fii = getPixColor(finishpix);
        sti[0]=(sti[0]+fii[0])/2;
        sti[1]=(sti[1]+fii[1])/2; // get average color behind start
and end pixels
        sti[2]=(sti[2]+fii[2])/2;
    }
    else
    {
        Color cl=identcolorbut.getBackground();
        int icl = cl.getRGB();
        sti[0] = (icl >> 16) & 0xff;
        sti[1] = (icl >> 8) & 0xff; // get RGB value of selected color
        sti[2] = (icl >> 0) & 0xff;
    }

    for(int row = 0;row < imgRows;row++)
    {
        int[] aRow = new int[imgCols];
        for(int col = 0; col < imgCols;col++)
        {
            int element = row * imgCols + col;
            aRow[col] = picArr[element];
        }

        for(int col = 0;col < imgCols;col++)
        {
            a = (aRow[col] >> 24) & 0xFF;
            r = (aRow[col] >> 16) & 0xFF; // get alpha + RGB value
for current pixel
            g = (aRow[col] >> 8) & 0xFF;
            b = (aRow[col]) & 0xFF;

            // test current pixel color is within range. range is test color +- precision
            if ((r > (sti[0]-precision)) & (g > (sti[1]-precision))
& (b > (sti[2]-precision))
                & (r < (sti[0]+precision)) & (g <
(sti[1]+precision)) & (b < (sti[2]+precision)))
            {
                objMatrix[row][col]=false; // store white point
                if (isLine==true)
                {
                    isLine=false;
                    gp.drawRect(col-length-1,row,length,0);
// draw line (rectangles are used as
length=0;
// they are faster than lines)
                }

            }
            else
            {
                objMatrix[row][col]=true; // store black point
                if (isLine==true)
                {
                    length++;
                }
                else
                {
                    isLine=true;
                }
            }
        }
    }
}

```



```

    }
    if (isLine==true)
    {
        isLine=false;
        gp.drawRect (imgCols-length-1,row,length,0); // draw
line
    }
}
else
{
    precision=255-precision;
    int[] sti={0,0,0};
    Color cl=identcolorbut.getBackground();
    int icl = cl.getRGB();
    sti[0] = (icl >> 16) & 0xff;
    sti[1] = (icl >> 8) & 0xff;
    sti[2] = (icl >> 0) & 0xff;

    for(int row = 0;row < imgRows;row++)
    {
        int[] aRow = new int[imgCols];
        for(int col = 0; col < imgCols;col++)
        {
            int element = row * imgCols + col;
            aRow[col] = picArr[element];
        }

        for(int col = 0;col < imgCols;col++)
        {
            a = (aRow[col] >> 24) & 0xFF;
            r = (aRow[col] >> 16) & 0xFF; // get alpha + RGB value
for current pixel
            g = (aRow[col] >> 8) & 0xFF;
            b = (aRow[col]) & 0xFF;

            // test current pixel color is within range. range is test color +/- precision
            if ((r > (sti[0]-precision)) & (g > (sti[1]-precision))
& (b > (sti[2]-precision))
& (r < (sti[0]+precision)) & (g <
(sti[1]+precision)) & (b < (sti[2]+precision)))
            {
                objMatrix[row][col]=true; // store black
point
                if (isLine==true)
                {
                    length++;
                }
                else
                {
                    isLine=true;
                }
            }
            else
            {
                objMatrix[row][col]=false; // store white point
                if (isLine==true)
                {
                    isLine=false;
                    gp.drawRect (col-length-1,row,length,0);
// draw line (rectangles are used as
length=0; // they are faster than lines)
                }
            }
        }
    }

    if (isLine==true)
    {
        isLine=false;
        gp.drawRect (imgCols-length-1,row,length,0); // draw
line
    }
}
precision=255-precision;

```

```

    }

    ic = new ImageIcon(modImage);          // display the results in image area
    imglab.setIcon(ic);
}

public void writeLastImg(String path) // store path to last opened image in a file
{
    try
    {
        PrintWriter writefile = new PrintWriter(new
FileWriter(lastImgFileName));
        writefile.println(path);
        writefile.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}

public String readLastImg()              // return path to last opened image
{
    try
    {
        BufferedReader input = new BufferedReader(new
FileReader(lastImgFileName));
        String path = input.readLine();
        input.close();
        if (path==null) path = samplefile;
        return path;
    }
    catch (IOException e1)                // if unable to read, open sample image
    {
        return samplefile;
    }
}

public void exit()                       // quit application
{
    int ans = JOptionPane.showConfirmDialog(this,          // display confirm dialog
        "Are you sure you want to Exit?",
        "Confirmation",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE);

    if (ans == JOptionPane.YES_OPTION)
    {
        System.exit(0);
    }
}

public void goForward()                  // called when continue button is clicked. Used
to handle processes that are                // specific to moving through
{
    steps only in one direction
    switch (Step)
    {
        case 1:
            Step=2;
            startpix.setLocation(-1,-1); // make sure start and end coordinates
aren't outside
            finishpix.setLocation(-1,-1); // the bounds of picture because it was
changed in step 1
            initStep2();
            break;
        case 2:
            tipsarea.setText("Here we define objects that will exist in 2D world.
White is for free space and black is" +
                " for objects. There are 4 ways to do it. The currently
selected one is basic and it simply " +
                "turns darker pixels to objects and brighter to space.
You may adjust the threshold with " +
                "the Slider above. To use a different method select it
in the Dropdown box.");
            Step=3;
            pickpixel=false;

```

```

        identstartbut.setBackground(defaultcolor); // reset start/fin buttons
        identfinbut.setBackground(defaultcolor);
        getBestThreshold(); // generate optimal threshold value
        initStep3();
        break;
    case 3:
        Step=4;
        pickpixel=false;
        initStep4();
        break;
    case 4:
        Step=5;
        imagepan.remove(imglab);
        imagepan.add(engine = new Engine(img, startpix, finishpix, world),
        BorderLayout.CENTER);
        engine.setDoubleBuffered(true);
        engine.start();
        initStep5();
        break;
    }
}

public void goBackward() // called when back button is clicked. Used to handle
processes that are
{
    // specific to moving through
    steps only in one direction
    switch (Step)
    {
        case 2:
            Step=1;
            identstartbut.setBackground(defaultcolor); // reset start/fin buttons
            identfinbut.setBackground(defaultcolor);
            initStep1();
            break;
        case 3:
            pickpixel=false;
            Step=2;
            ic = new ImageIcon(img); // display original image
            imglab.setIcon(ic);
            initStep2();
            break;
        case 4:
            tipsarea.setText("Define objects that will exist in 2D world. White is
for free space and black is" +
            " for objects.");
            Step=3;
            initStep3();
            break;
        case 5:
            Step=4;
            engine.stop();
            imagepan.remove(engine);
            imagepan.add(imglab = new JLabel(ic),BorderLayout.CENTER);
            imglab.addMouseListener(this);

            viewpic.setText("View original image");
            tipsarea.setText("Now you can see the bounds of the world we generated.
" +
            "Differently colored fragments of borders " +
            "stand for different lines. It is desirable that they
precisely define the bounds of object " +
            "but that there are not too many of them. If there seem
too lines, " +
            "you may want to redefine objects by going a step back.
");
            backbut.setEnabled(true);
            contbut.setEnabled(true);
            showPanel(S4); // change toolbar to step 4
            //plugGaps();
            drawLines();
            //initStep4();
            break;
    }
}

public void getPackagedImg(String path) // load image from file
{

```

```

        setCursor(HGlassCursor);
        try
        {
            URL url = UBTMain.class.getResource(path);
            img = ImageIO.read(url);
        }
        catch (IOException e)
        {
            infoarea.setText(infoarea.getText() + "Invalid image path!");
            setCursor(DefaultCursor);
            return;
        }
        imgCols = img.getWidth(this);           // Get width and height
        imgRows = img.getHeight(this);          // of the image
        ic = new ImageIcon(img);
        imglab.setIcon(ic);
        contbut.setEnabled(true);
        infoarea.setText(infoarea.getText() + "-----
\n");
        infoarea.setText(infoarea.getText() + "Image " + imgCols + " x " + imgRows + "
loaded successfully.\n");
        tipsarea.setText("Now you may press 'Continue' to proceed.");
        setCursor(DefaultCursor);
    }

    public void getImg(String path)              // load image from file
    {
        setCursor(HGlassCursor);
        try
        {
            img = ImageIO.read(new File(path));           // read file
        }
        catch (IOException e)
        {
            infoarea.setText(infoarea.getText() + "Invalid image path!");
            setCursor(DefaultCursor);
            return;
        }
        imgCols = img.getWidth(this);           // Get width and height
        imgRows = img.getHeight(this);          // of the image
        ic = new ImageIcon(img);
        imglab.setIcon(ic);
        contbut.setEnabled(true);
        infoarea.setText(infoarea.getText() + "-----
\n");
        infoarea.setText(infoarea.getText() + "Image " + imgCols + " x " + imgRows + "
loaded successfully.\n");
        tipsarea.setText("Now you may press 'Continue' to proceed.");
        setCursor(DefaultCursor);
    }

    public void showPanel(String s)              // display one of the toolbars
    {
        CardLayout cl = (CardLayout)(steppan.getLayout());
        cl.show(steppan,s);
    }

    public void startclick()                    // identify start pixel
    {
        if (identstartbut.getBackground()== Color.GREEN)    // exit pick mode
        {
            pickpixel=false;
            identstartbut.setBackground(defaultcolor);
        }
        else // enter pick mode
        {
            if (pickpixel==true)
            {
                identfinbut.setBackground(defaultcolor);
            }
            pickpixel=true;
            defaultcolor = identstartbut.getBackground();
            identstartbut.setBackground(Color.GREEN);
        }
    }
}

```

```

public void finclick()          // identify finish pixel
{
    if (identfinbut.getBackground()== Color.GREEN)      // exit pick mode
    {
        pickpixel=false;
        identfinbut.setBackground(defaultcolor);
    }
    else    // enter pick mode
    {
        if (pickpixel==true)
        {
            identstartbut.setBackground(defaultcolor);
        }
        pickpixel=true;
        defaultcolor = identfinbut.getBackground();
        identfinbut.setBackground(Color.GREEN);
    }
}

public void drawSF()          // draw start/finish circles on the image
{
    int x,y;
    BufferedImage modImage = new BufferedImage(imgCols,imgRows,img.getType());
    Graphics2D gp = modImage.createGraphics();

    gp.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.VALUE_ANTIALIAS
_ON); // antialiasing on
    gp.drawImage(img, 0, 0, null);

    if (startpix.getX()!=-1)          // draw start position if it exists
    {
        x=(int)startpix.getX();
        y=(int)startpix.getY();
        gp.setColor(Color.WHITE);
        gp.fillOval(x-5,y-5,10,10);
        gp.setColor(Color.BLUE);
        gp.fillOval(x-4,y-4,8,8);
    }

    if (finishpix.getX()!=-1)          // draw finish position if it exists
    {
        x=(int)finishpix.getX();
        y=(int)finishpix.getY();
        gp.setColor(Color.WHITE);
        gp.fillOval(x-5,y-5,10,10);
        gp.setColor(Color.RED);
        gp.fillOval(x-4,y-4,8,8);
    }

    ic = new ImageIcon(modImage);      // display updated image
    imglab.setIcon(ic);

    if (startpix.getX()!=-1 & finishpix.getX()!=-1) contbut.setEnabled(true); //
    if both set - allow continue
    }

    public int[] getPicArray()          // get array of pixels
    {
        int[] picArr = new int[imgCols * imgRows];      // create array;
        BufferedImage buffImage = new
        BufferedImage(imgCols,imgRows,BufferedImage.TYPE_INT_ARGB);
        Graphics2D gp = buffImage.createGraphics();
        gp.drawImage(img, 0, 0, null);
        DataBufferInt dataBufferInt =
        (DataBufferInt)buffImage.getRaster().getDataBuffer();
        picArr = dataBufferInt.getData();      // get integer data
        return picArr;
    }

    public void getBestThreshold()          // generate optimal threshold
    value
    {
        long average=0, alow=0, ahigh=0;
        int[] picArr = getPicArray();
        int a,r,g,b;

```

```

for(int row = 0;row < imgRows;row++)
{
    int[] aRow = new int[imgCols];
    for(int col = 0; col < imgCols;col++)
    {
        int element = row * imgCols + col;
        aRow[col] = picArr[element];
    }

    for(int col = 0;col < imgCols;col++)
    {
        a = (aRow[col] >> 24) & 0xFF;
        r = (aRow[col] >> 16) & 0xFF; // get alpha + RGB value for
current pixel
        g = (aRow[col] >> 8) & 0xFF;
        b = (aRow[col]) & 0xFF;
        average=average + (r+g+b)/3; // add greyscale value of the
pixel to average
    }
    average=average/(imgRows*imgCols); // get the actual average

    int greyval;
    for(int row = 0;row < imgRows;row++)
    {
        int[] aRow = new int[imgCols];
        for(int col = 0; col < imgCols;col++)
        {
            int element = row * imgCols + col;
            aRow[col] = picArr[element];
        }

        for(int col = 0;col < imgCols;col++)
        {
            a = (aRow[col] >> 24) & 0xFF;
            r = (aRow[col] >> 16) & 0xFF; // get alpha + RGB value for
current pixel
            g = (aRow[col] >> 8) & 0xFF;
            b = (aRow[col]) & 0xFF;
            greyval=(r+g+b)/3; // get greyscale value
            if(greyval>average)
            {
                ahigh=ahigh + greyval; // add to values above average
            }
            else
            {
                alow=alow + greyval; // add to values below average
            }
        }
    }
    alow=alow/(imgRows*imgCols); // get average for values above and below initial
average
    ahigh=ahigh/(imgRows*imgCols);
    average=(alow+ahigh)/2; // get average of the two values
    precision=(int)average; // set precision to this value
    precslider.setValue(precision); // update slider
}

public void actionPerformed(ActionEvent e) // listen to actions
{
    if (e.getSource() == closebut) // Close
    {
        exit();
    }
    if (e.getSource() == contbut) // Continue
    {
        goForward();
    }
    if (e.getSource() == backbut) // Back
    {
        goBackward();
    }
    if (e.getSource() == opennew) // Open new image
    {
        JFileChooser chooser = new JFileChooser();
        ExampleFileFilter filter = new ExampleFileFilter();
        filter.addExtension("jpg");
    }
}

```

```

        filter.addExtension("gif");
        filter.setDescription("JPG & GIF Images");
        chooser.setFileFilter(filter);
        int returnVal = chooser.showOpenDialog(this);
        if(returnVal == JFileChooser.APPROVE_OPTION)
        {
            getImg(chooser.getSelectedFile().getPath());
            writeLastImg(chooser.getSelectedFile().getPath()); // write
path to this image
        }
    }
    if (e.getSource() == openlast) // Open last image
    {
        getImg(readLastImg());
    }
    if (e.getSource() == opensample) // Open sample image
    {
        getPackagedImg(samplefile);
    }
    if (e.getSource() == identstartbut) // Identify start position
    {
        startclick();
    }
    if (e.getSource() == identfinbut) // identify finish position
    {
        finclick();
    }
    if (e.getSource() == objIdMode) // Object identification mode (dropdownbox)
    {
        if (objIdMode.getSelectedIndex()==3 &
identcolorbut.setBackground()==Color.WHITE)
        {
            identcolorbut.setBackground(Color.BLACK);
        }

        if (objIdMode.getSelectedIndex()==0) // show tips for each mode
        {
            tipsarea.setText("Basic simply turns darker pixels to objects "
+
                                "and brighter to space. You may adjust the
threshold with the Slider above. ");
        }
        else if (objIdMode.getSelectedIndex()==1)
        {
            tipsarea.setText("Smart takes the background of the start pixel
and end pixel " +
                                "and generates threshold based on their average
color. " +
                                "You may adjust the threshold with the Slider
above. ");
        }
        else if (objIdMode.getSelectedIndex()==2)
        {
            tipsarea.setText("Here you can identify the base color of free
space yourself. " +
                                "Click the Color button and then click somewhere
on the image to pick a color. Then click " +
                                "the same button again to exit the pick mode. " +
                                "You may adjust the threshold with the Slider
above. ");
        }
        else if (objIdMode.getSelectedIndex()==3)
        {
            tipsarea.setText("Here you can identify the base color of
obstacle yourself. " +
                                "Click the Color button and then click somewhere
on the image to pick a color. Then click " +
                                "the same button again to exit the pick mode. " +
                                "You may adjust the threshold with the Slider
above. ");
        }
        if (objIdMode.getSelectedIndex()==0 | objIdMode.getSelectedIndex()==1)
        // enable/disable pick colour
        {
            identcolorbut.setEnabled(false);
        }
    }

```

```

        else
        {
            identcolorbut.setEnabled(true);
        }
        precision=(int)precslider.getValue();
        renderObjs();
    }
    if (e.getSource() == identcolorbut)           // Pick colour
    {
        if(pickpixel==false)
        {
            pickpixel=true;
            identcolorbut.setForeground(Color.WHITE);
            ic = new ImageIcon(img);
            imglab.setIcon(ic);
        }
        else
        {
            pickpixel=false;
            identcolorbut.setForeground(Color.DARK_GRAY);
            renderObjs();
        }
    }
    if (e.getSource() == viewpic)           // View original image / View Lines
    {
        if(viewpic.getText() == "View original image")
        {
            viewpic.setText("View Lines");
            ic = new ImageIcon(img);
            imglab.setIcon(ic);
            imagepan.setBackground(Color.WHITE);
        }
        else
        {
            viewpic.setText("View original image");
            drawLines();
        }
    }
}

public void stateChanged(ChangeEvent e)           // listen to state changes
{
    if (e.getSource() == precslider)
    {
        precision=(int)precslider.getValue();
        renderObjs();
    }
}

public void mouseClicked(MouseEvent k)
{
}

public void mousePressed(MouseEvent k)
{
}

public void mouseReleased(MouseEvent k) // get pixel info if in pick mode
{
    if(k.getComponent()==imglab & pickpixel==true)
    {
        int xset = getWidth() - statuspan.getWidth();
        xset = (xset - imgCols)/2;
        int x = k.getX();
        x = x - xset;
        int yset = getHeight() - upperpan.getHeight();
        yset = (yset - imgRows)/2;
        int y = k.getY();
        y = y - yset;
        x = x + 4;           // adjust global coordinates to image coordinates
        y = y + 16;

        if (x>=0 & y>=0 & x<imgCols & y<imgRows) // if within image bounds

```



```

        {
            if (identstartbut.getBackground()== Color.GREEN)    // start
position
            {
                startpix.setLocation(x,y);
                drawSF();
            }
            else if (identfinbut.getBackground()== Color.GREEN) // finish
position
            {
                finishpix.setLocation(x,y);
                drawSF();
            }
            else
            {
                Point finishpix= new Point(x,y);    // pick colour
                int[] col = getPixColor(finishpix);
                Color cl = new Color(col[0],col[1],col[2]);
                identcolorbut.setBackground(cl);
            }
        }
    }

    public void mouseEntered(MouseEvent k)    // show cross cursor if within image and
in pick mode
    {
        if(k.getComponent()==imglab & pickpixel==true)
        {
            setCursor(CrossCursor);
        }
    }

    public void mouseExited(MouseEvent k)    // reset cursor to default
    {
        if(k.getComponent()==imglab & pickpixel==true)
        {
            setCursor(DefaultCursor);
        }
    }
}

```

Engine

```

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.RenderingHints;
import java.awt.geom.AffineTransform;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.Vector;
import java.awt.Point;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.vecmath.Vector2d;

// The Engine class performs the transformations and the painting.

class Engine extends JLabel implements Runnable
{
    BufferedImage img;
    Image sp;

    int agentsnumber=10;    // number of agents

```

```

int releasecount=20;          // time between agent spawns
int agWidth;                  // width of an agent
int agHeight;                  // height of an agent

Vector agents;

int count=releasecount;

private Thread thread;
private BufferedImage bimg;

world) public Engine(BufferedImage img, Point startpix, Point finishpix, Vector
{
    setBackground(Color.WHITE);
    this.img=img;

    agents = new Vector();

    while(agentsnumber>0)
    {
        agents.add(new Agent(startpix, finishpix, world));
        agentsnumber--;
    }

    try
    {
        URL url = UBQTMMain.class.getResource("gfx/spider.gif");
        sp = ImageIO.read(url);          // read file
    }

    catch (IOException e)
    {
        System.out.println("No agent image");
    }

    agWidth=sp.getWidth(this);
    agHeight=sp.getHeight(this);
}

public void drawEngine(int w, int h, Graphics2D g2)
{
    int x = (getWidth() - img.getWidth())/2;
    int y = (getHeight() - img.getHeight())/2;
    AffineTransform at = new AffineTransform();
    at.translate(x,y);

    g2.drawImage(img, at, this);

    g2.setTransform(at);

    Agent ag0=(Agent)agents.get(0);
    for (int i = 0; i < ag0.nodes.size(); i++)
    {
        Node nd=(Node)ag0.nodes.get(i);

        for(int k=0; k<nd.branches.size();k++)
        {
            Branch br=(Branch)nd.branches.get(k);
            if(br.dead==true)
            {
                g2.setColor(Color.RED);
            }
            else if(br.explored==true)
            {
                g2.setColor(Color.YELLOW);
            }
            else
            {
                g2.setColor(Color.GREEN);
            }
        }

        g2.drawLine((int)nd.position.x, (int)nd.position.y, br.edge.x, br.edge.y);
    }
}

```

```

        if(nd.dead==true)
        {
            g2.setColor(Color.RED);
        }
        else
        {
            g2.setColor(Color.GREEN);
        }
        g2.fillOval(((int)nd.position.x)-4,((int)nd.position.y)-4,8,8);
    }

    for (int i = 0; i < agents.size(); i++)
    {
        Agent ag=(Agent)agents.get(i);

        g2.translate(-agWidth/2,-agHeight/2);
        g2.rotate(ag.orientation, (int)ag.position.x+agWidth/2,
(int)ag.position.y+agHeight/2);
        g2.drawImage(sp, (int)ag.position.x, (int)ag.position.y, this);
        g2.rotate(-ag.orientation, (int)ag.position.x+agWidth/2,
(int)ag.position.y+agHeight/2);
        g2.translate(agWidth/2,agHeight/2);
    }
}

public Graphics2D createGraphics2D(int w, int h)
{
    Graphics2D g2 = null;
    if (bimg == null || bimg.getWidth() != w || bimg.getHeight() != h) {
        bimg = (BufferedImage) createImage(w, h);
        // reset(w, h);
    }
    g2 = bimg.createGraphics();
    g2.setBackground(getBackground());

    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setRenderingHint(RenderingHints.KEY_RENDERING,
        RenderingHints.VALUE_RENDER_QUALITY);
    g2.clearRect(0, 0, w, h);
    return g2;
}

public void paint(Graphics g)
{
    Dimension d = getSize();
    Graphics2D g2 = createGraphics2D(d.width, d.height);
    drawEngine(d.width, d.height, g2);
    g2.dispose();
    g.drawImage(bimg, 0, 0, this);
}

public void start()
{
    thread = new Thread(this);
    thread.setPriority(Thread.MIN_PRIORITY);
    thread.start();
    for (int i = 0; i < agents.size(); i++)
    {
        Agent ag=(Agent)agents.get(i);
        ag.run();
    }
}

public synchronized void stop()
{
    Agent ag0=(Agent)agents.get(0);
    ag0.nodes.clear();
    for (int i = 0; i < agents.size(); i++)
    {
        Agent ag=(Agent)agents.get(i);
        ag.stop();
    }
}

```

```

        thread = null;
    }

    public void run()
    {
        Thread me = Thread.currentThread();
        while (thread == me) {
            count--;
            if(count==0)
            {
                count=releasecount;
                if(agentsnumber<agents.size())
                {
                    Agent ag=(Agent)agents.get(agentsnumber);
                    ag.start();
                    agentsnumber++;
                }
            }

            repaint();
            try {
                thread.sleep(20);
            } catch (InterruptedException e) { break; }
        }
        thread = null;
    }
}

```

Agent

```

import java.awt.Point;
import java.util.Vector;
import javax.vecmath.Vector2d;
import javax.vecmath.Matrix3d;
import javax.vecmath.Vector3d;
import java.awt.geom.Line2D;

class Agent implements Runnable
{
    public static Vector nodes=new Vector();    // the collection of nodes is
shared between all agents

    private Thread thread;
    public Vector2d position;                  // current position in the world
    public Vector2d direction;                 // direction in which the agent is moving
    public double orientation; // used to show graphically which direction agent
faces
    Point start;                               // the start position
    Point finish;                             // the finish position
    Vector2d comefrom;                         // stores coordinates of last visited node
    Vector2d senfrontr, senfrontl, senright, senleft; // sensors
    double speed=0.7;                         // agents speed
    int senlength=30;                         // length of sensors
    Vector world;                             // all the walls in the world
    boolean rightwall, leftwall, readingChanged; // used to initiate
creation of a new node
    int visited; // stores numbewr of steps passed after last node visit

    public Agent(Point startpix,Point finishpix, Vector worldgen) // constructor
    {
        world=new Vector();
        Line li;
        Line2D.Double li2d;
        for(int i=0; i<worldgen.size();i++) // convert lines from the world
into Line2D format
        {
            // this is required to be able to perform
            li=(Line)worldgen.get(i); // line intersect tests
            li2d=new
            Line2D.Double(li.getx1(),li.gety1(),li.getx2(),li.gety2());

```

```

        world.add(li2d);
    }
    visited=0;
    comefrom=new Vector2d(0,0);    // doesn't come from any node yet
    start=startpix;
    finish=finishpix;
    position=new Vector2d((int)startpix.getX(),(int)startpix.getY());
    senfrontr = (Vector2d) position.clone();
    senfrontl = (Vector2d) position.clone();
    senright = (Vector2d) position.clone();
    senleft = (Vector2d) position.clone();
    direction = new Vector2d(finish.x-position.x,finish.y-position.y); //
face finish position
    direction.normalize();
    calcSensors();    // calculate sensors coordinates

}

    public void update()
    {
        if(finish.distance(position.x,position.y)<15) thread=null; // if
destination finished - stop

        if (visited!=0)visited--;    // count down from the last node visit
// ----- communicate with nodes -----

        if(nodes.size()>0 & visited==0)    // check if a node is nearby
        {
            double dist=1000;
            double tmp;
            Point p;
            Node nd;
            int pos=0;

            for(int i=0;i<nodes.size();i++) // get distance to the closest
node
            {
                p = new Point((int)position.x,(int)position.y);
                nd=(Node)nodes.get(i);
                tmp=p.distance(nd.position.x,nd.position.y);
                if (tmp<dist)
                {
                    dist=tmp;
                    pos=i;
                }
            }

            if(dist<senlength/4) // if node is very close
            {
                nd=(Node)nodes.get(pos);
                Vector2d backw = (Vector2d) direction.clone();
                backw.x=-backw.x;
                backw.y=-backw.y;
                if(comefrom.x==0 & comefrom.y==0)
                    // if doesn't come from any node
                else if(comefrom==nd.position)nd.killEntrance(backw);
                // if comes from the same node
                else
                {
                    for(int i=0;i<nodes.size();i++)
                    {
                        Node ndt=(Node)nodes.get(i);
                        if(comefrom==ndt.position & ndt.dead==true)
// agent comes from a dead node
                        {
                            nd.killEntrance(backw);
                            break;
                        }
                    }
                }
                Vector2d headto=nd.getdirection(backw); // give agent
new direction

                direction = (Vector2d) headto.clone();
                comefrom = nd.position;
            }
        }
    }
}

```

```

        position=(Vector2d)nd.position.clone();
        visited=60;
    }
    else if (dist<senlength) // if not too close, stir in its
direction
    {
        nd=(Node)nodes.get(pos);
        direction = new Vector2d(nd.position.x-
position.x,nd.position.y-position.y);
    }

    // ----- end communicate with nodes -----

    calcSensors();          // calculate sensors positions

    double distfr=1000;
    double distfl=1000;
    double distr=1000; // if this variable has value 1000 after the check
    double distl=1000; // that means the relevant sensor doesn't touch any
wall at all

    double tmpdist=0;
    Line2D li;

    for(int i=0; i<world.size();i++) // check which sensors touch walls
    {
        // and if they touch, how far is the closest wall
        li=(Line2D.Double)world.get(i);

        if(li.intersectsLine(position.x,position.y,senfrontr.x,senfrontr.y)==true)
        {
            tmpdist=li.ptLineDist(position.x,position.y);
            if (tmpdist<distfr) distfr=tmpdist;
        }

        if(li.intersectsLine(position.x,position.y,senfrontl.x,senfrontl.y)==true)
        {
            tmpdist=li.ptLineDist(position.x,position.y);
            if (tmpdist<distfl) distfl=tmpdist;
        }

        if(li.intersectsLine(position.x,position.y,senright.x,senright.y)==true)
        {
            tmpdist=li.ptLineDist(position.x,position.y);
            if (tmpdist<distr) distr=tmpdist;
        }

        if(li.intersectsLine(position.x,position.y,senleft.x,senleft.y)==true)
        {
            tmpdist=li.ptLineDist(position.x,position.y);
            if (tmpdist<distl) distl=tmpdist;
        }
    }

    boolean tmpr=true;
    boolean tmp1=true;

    if (distr==1000)          // check specifically left and right sensors
    {
        tmpr=false;
    }
    else
    {
        tmpr=true;
    }

    if (distl==1000)
    {
        tmp1=false;
    }
    else
    {
        tmp1=true;
    }

```

```

        if(tmpr!=rightwall | tmpl!=leftwall) // if their read has changed from
the last cycle
    {
        // check if there is need for a new node
        double dist=1000;
        double tmp;
        Point p;
        Node nd;

        for(int i=0;i<nodes.size();i++)
        {
            p = new Point((int)position.x, (int)position.y);
            nd=(Node)nodes.get(i);
            tmp=p.distance(nd.position.x, nd.position.y);
            if (tmp<dist) dist=tmp;
        }

        if (dist>senlength) createnode(); // if there are no nodes
naerby initiate node creation
    }

    rightwall=tmpr;
    leftwall=tmpl;

    direction.x=direction.x*(senlength*50); // give current direction
vector enough power
    direction.y=direction.y*(senlength*50);

    if(distfr!=1000 | distfl!=1000)
    {

        if(distfr!=1000 & distfl!=1000) // calculate impact from both
front sensors
        {
            double angle;
            if(distfl<distfr) // generate angle at which the force
will affect direction
            {
                tmpdist=distfl;
                angle=90;
            }
            else
            {
                tmpdist=distfr;
                angle=-90;
            }
            Vector2d tem=rotate(direction,Math.toRadians(angle));
            tem.normalize();
            Vector2d rforce=(Vector2d)tem.clone();
            rforce.x=(rforce.x*(senlength-tmpdist)*100);
            rforce.y=(rforce.y*(senlength-tmpdist)*100);
            direction.add(rforce);
        }
        else if(distfr!=1000) // calculate impact from right front
sensor
        {
            Vector2d tem=rotate(direction,Math.toRadians(20));
            tem.normalize();
            Vector2d rforce=(Vector2d)tem.clone();
            rforce.x=-(rforce.x*(senlength-distfr)*20);
            rforce.y=-(rforce.y*(senlength-distfr)*20);
            direction.add(rforce);
        }
        else if(distfl!=1000) // calculate impact from left front sensor
        {
            Vector2d tem=rotate(direction,Math.toRadians(-20));
            tem.normalize();
            Vector2d lforce=(Vector2d)tem.clone();
            lforce.x=-(lforce.x*(senlength-distfl)*20);
            lforce.y=-(lforce.y*(senlength-distfl)*20);
            direction.add(lforce);
        }
    }

    if(distr!=1000 | distl!=1000) // calculate impact of side forces from
both walls
    {

```

```

        Vector2d tem=rotate(direction,Math.toRadians(90));
        tem.normalize();
        if(distr!=1000 & distl!=1000)
        {
            Vector2d rforce=(Vector2d)tem.clone();
            rforce.x=-(rforce.x*(senlength-distr));
            rforce.y=-(rforce.y*(senlength-distr));

            Vector2d lforce=(Vector2d)tem.clone();
            lforce.x=(lforce.x*(senlength-distl));
            lforce.y=(lforce.y*(senlength-distl));

            tem.add(rforce,lforce);
            direction.add(tem);
        }
        else if(distr!=1000)    // calculate impact of side forces from
right wall
        {
            Vector2d rforce=(Vector2d)tem.clone();
            rforce.x=-(rforce.x*(senlength-distr));
            rforce.y=-(rforce.y*(senlength-distr));
            direction.add(rforce);
        }
        else if(distl!=1000)    // calculate impact of side forces from
left wall
        {
            Vector2d lforce=(Vector2d)tem.clone();
            lforce.x=(lforce.x*(senlength-distl));
            lforce.y=(lforce.y*(senlength-distl));
            direction.add(lforce);
        }
    }

    direction.normalize(); // normalizes direction vector after all the
forces were applied to it

    if(direction.x<0)orientation=-direction.angle(new Vector2d(0,-1)); //
adjust rotation
    else orientation=direction.angle(new Vector2d(0,-1));

    Vector2d temppos=new Vector2d();

    if(distfr<(senlength/2) & distfl<(senlength/2))    // if wall in front
is too close - reduce speed
    {

        temppos.x=position.x+direction.x*(speed/3); // get new position
        temppos.y=position.y+direction.y*(speed/3);

    }

    temppos.x=position.x+direction.x*speed;    // get new position
    temppos.y=position.y+direction.y*speed;

    position=(Vector2d)temppos.clone();    // this is required to make sure
agent is drawn correctly
}

public void start()
{
    thread = new Thread(this);
    thread.setPriority(Thread.MIN_PRIORITY);
    thread.start();
}

public synchronized void stop()
{
    thread = null;
}

public void createnode()    // create a new node if all conditions are met
{
    double podist=1000;
    double tmp;

```



```

        for(int i=0; i<world.size();i++)          // check that the point is not too
close to a wall
    {
        Line2D.Double li=(Line2D.Double)world.get(i);
        Point po = new Point((int)position.x, (int)position.y);
        tmp=po.distance(li.x1,li.y1);
        if(tmp<podist)podist=tmp;

        tmp=po.distance(li.x2,li.y2);
        if(tmp<podist)podist=tmp;
    }

    if (podist<10)return;          // if wall too close, don't proceed

    Vector2d up = new Vector2d(0,1);
    Point[] ends= new Point[36]; // stores coordinates of each rays end
    Vector2d[] dirs= new Vector2d[36]; // stores vector of each ray
    int mult=0;

    for(int a=0; a<36; a++) // generate coordinates for all 36 rays in
advance
    {
        mult=mult+10;

        dirs[a]=rotate(up,Math.toRadians(mult));
// first generate vector based on the angle
        ends[a]=new Point(0,0);
        ends[a].x=(int) (position.x+dirs[a].x*(senlength)); // then get
coordinates
        ends[a].y=(int) (position.y+dirs[a].y*(senlength)); // with the help of
vector

        Line2D li;
        boolean inter;
        Vector2d po=(Vector2d) position.clone();
        Node thenode = new Node(po);          // create new node

        int dirsinbranch=0;
        int branches=0;
        boolean isbranch=false;
        Vector2d start = new Vector2d();
        Vector2d finish = new Vector2d();
        int done=0;
        int j=0;
        boolean started=false;

        while(done<ends.length)
        {
            if (j>=ends.length) j=j-ends.length;

            inter=false;

            for(int i=0; i<world.size();i++)          // iterate through every wall
            {

                li=(Line2D.Double)world.get(i);

                if(li.intersectsLine(position.x,position.y,ends[j].x,ends[j].y)==true)
                {
                    inter=true;          // ray intersects with a wall
                    break;
                }
            }

            if(inter==false)          // direction doesn't intersect walls
            {
                if(started==true)
                {
                    done++;
                    dirsinbranch++;
                    if(isbranch==false)          // started new branch
                    {
                        isbranch=true;

```

```

        start=dirs[j];
    }
    else // already inside branch
    {
        finish=dirs[j];
    }
}
else
{
    started=true;
}
}
else // direction intersects walls
{
    if(started==true)
    {
        done++;
        if(isbranch==true) // just finished branch
        {
            isbranch=false;
            if(dirsinbranch>1) // if branch has at
least 2 directions next to each other
            {
                Point nd = new Point();
                start.add(finish); // get average
direction
                nd=new Point(0,0);
                nd.x=(int) (position.x+start.x*(senlength/2));
                // generate branch coordinates
                nd.y=(int) (position.y+start.y*(senlength/2));
                thenode.add(start,nd);
                branches++;
            }
            dirsinbranch=0;
        }
    }
    j++;
}

if(branches>2) nodes.add(thenode); // there are at least 3
branches - add to the collection of nodes
}

public void calcSensors() // calculate coordinates for all sensors
{
    Vector2d tem=rotate(direction,Math.toRadians(10));
    senfrontr.x=position.x+tem.x*senlength; // get front sensor
    senfrontr.y=position.y+tem.y*senlength;

    tem=rotate(direction,Math.toRadians(-10));
    senfrontl.x=position.x+tem.x*senlength; // get back sensor
    senfrontl.y=position.y+tem.y*senlength;

    tem=rotate(direction,Math.toRadians(90));
    senright.x=position.x+tem.x*senlength; // get right sensor
    senright.y=position.y+tem.y*senlength;

    senleft.x=position.x-tem.x*senlength; // get left sensor
    senleft.y=position.y-tem.y*senlength;
}

public Vector2d rotate(Vector2d vec, double angle) // rotates a vector by a
given angle
{
    Matrix3d mat=new Matrix3d();
    Vector3d vec3=new Vector3d(vec.x,vec.y,0);
    mat.rotZ(angle);
    mat.transform(vec3);
    Vector2d ret=new Vector2d(vec3.x,vec3.y);
    return ret;
}

public void run() {
    Thread me = Thread.currentThread();

```

```

        while (thread == me) {
            update();
            try {
                thread.sleep(5);
            } catch (InterruptedException e) { break; }
        }
        thread = null;
    }

    public void reset()
    {
    }
}

```

Node

```

import java.awt.Point;
import java.util.Vector;
import javax.vecmath.Vector2d;

class Node                                // holds Node details
{
    Vector2d position;                    // its position in the world
    Vector branches;                     // points to all directions the node has
    boolean dead;                        // true if less than two branches are valid
    int lastval=0;                       // used to generate different directions

    public Node(Vector2d position)        // constructor
    {
        this.position = position;
        branches=new Vector();
        dead=false;
    }

    public void add(Vector2d direction, Point edge) // adds a new branch
    {
        branches.add(new Branch(direction,edge));
    }

    public Vector2d getdirection(Vector2d entr) // gives an agent a valid
direction to follow
    {
        double ang=360;
        double tmp;
        int pos=0;

        for(int i=0; i<branches.size(); i++) // make direction from
which agent came explored
        {
            Branch b = (Branch)branches.get(i);
            tmp=b.direction.angle(entr);
            if (tmp<ang)
            {
                ang=tmp;
                pos=i;
            }
        }
        Branch b = (Branch)branches.get(pos);
        b.explore();

        for(int i=0; i<branches.size(); i++) // first try to find an unexplored
branch
        {
            b = (Branch)branches.get(i);
            if(b.explored==false & i!=pos)
            {
                b.explore();
                lastval=i;
                return b.direction;
            }
        }
    }
}

```

```

    }

    for(int i=0; i<branches.size(); i++) // if all are explored, give one
that is explored
    {
        Branch bt = (Branch)branches.get(i);
        if(bt.dead==false & i!=pos & i!=lastval)
        {
            lastval=i;
            return bt.direction;
        }
    }

    for(int i=0; i<branches.size(); i++) // if all are explored, give one
that is explored
    {
        Branch bt = (Branch)branches.get(i);
        if(bt.dead==false & i!=pos)
        {
            lastval=i;
            return bt.direction;
        }
    }

    for(int i=0; i<branches.size(); i++) // if all are explored, give one
that is explored
    {
        Branch bt = (Branch)branches.get(i);
        if(bt.dead==false)
        {
            lastval=i;
            return bt.direction;
        }
    }
    lastval=0;
    b = (Branch)branches.get(0); // if none works give some direction
    return b.direction;
}

public void killEntrance(Vector2d entr) // kill branch from which agent came
{
    double ang=360;
    double tmp;
    int pos=0;
    for(int i=0; i<branches.size(); i++)
    {
        Branch b = (Branch)branches.get(i);
        tmp=b.direction.angle(entr);
        if (tmp<ang)
        {
            ang=tmp;
            pos=i;
        }
    }
    Branch b = (Branch)branches.get(pos);
    b.kill();

    int live=0;
    for(int i=0; i<branches.size(); i++)
    {
        b = (Branch)branches.get(i);
        if(b.dead==false)live++;
    }
    if (live<2) dead=true; // if only one valid direction - kill node
    if (live<1)
    {
        b = (Branch)branches.get(pos);
        b.dead=false;
    }
}
}

```

Branch

```

import java.awt.Point;
import javax.vecmath.Vector2d;

class Branch // holds details of a direction
{
    Vector2d direction;
    Point edge;

    boolean explored; // true if at least one agent already took this route
    boolean dead; // true if an agent discovered it leads to a deadend

    public Branch(Vector2d direction, Point edge) // constructor
    {
        this.direction = direction;
        this.edge = edge;
        explored=false;
        dead=false;
    }

    public void explore()
    {
        explored=true;
    }

    public void kill()
    {
        dead=true;
    }
}

```

Line

```

class Line // holds line details
{
    int x1;
    int y1;
    int x2;
    int y2;

    public Line(int x1,int y1,int x2,int y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    int getx1()
    {
        return x1;
    }

    int gety1()
    {
        return y1;
    }

    int getx2()
    {
        return x2;
    }

    int gety2()
    {
        return y2;
    }

    void setLine(int x1,int y1,int x2,int y2)

```

```

        {
            this.x1 = x1;
            this.y1 = y1;
            this.x2 = x2;
            this.y2 = y2;
        }
    }
}

```

PointQueue

```

import java.util.*;
import java.awt.*;

public class PointQueue extends Vector
{
    public void enqueue(Point element)
    {
        addElement (element);
    }

    public Point dequeue ()
    {
        Point obj = (Point)elementAt(0);
        removeElementAt (0);
        return obj;
    }
}

```

References

1. *Maths Connects*.
Online at <http://www.newton.cam.ac.uk/wmy2kposters/june/index.html>
Accessed on 12th November 2006.
2. *Maze Solver*.
Online at <http://www.c-sharpcorner.com/Code/2002/Nov/MazeSolver.asp>
Accessed on 26th November 2006.
3. Russel C. Erehart, James Kennedy. *Swarm Intelligence*, Morgan Kaufmann, 2001.
4. Mat Buckland. *Programming Game AI by Example*, Wordware Publishing 2005.
5. John McCarthy. *What is Artificial Intelligence?*
Online at <http://www-formal.stanford.edu/jmc/whatisai/whatisai.html>
Accessed on 17th November 2006.
6. P. Rybski, A. Larson, H. Veerarghavan. *Performance Evaluation of a Multi-Robot Search & Retrieval System: Experiences with MiniDart*, 2004.
7. *Maze*. Online at <http://en.wikipedia.org/wiki/Maze>
Accessed on 26th November 2006.
8. *Think Labyrinth: Maze Classification*.
Online at www.astrolog.org/labyrinth/algrithm.htm
Accessed on 15th November 2006.
9. *One Billion Mazes*. Online at <http://www.onebillionmazes.com>
Accessed on 6th December 2006.
10. Katia P. Sycara. *Multiagent Systems*. AI Magazine 19(2).
11. Marco Mamei, Ronaldo Menezes, Robert Tolksdorf, Franco Zambonelli.
Case Studies for Self-Organization in Computer Science. Online at:
<http://www.agentgroup.unimo.it/Zambonelli/PDF/JSA.pdf>
12. *RAD White Paper: What is Rapid Application Development?*, CASEMaker Inc.
13. *Creative Data: Rapid Application Development – development Methodology (RAD)*. Online at <http://www.credata.com/research/rad.html>
Accessed on 20th March 2007.

14. B. Eckel. *Comparing C++ and Java*. Online at
<http://www.javacoffeebreak.com/articles/thinkinginjava/comparingc++andjava.html>
Accessed on 5th November 2006.
15. *What are vector and raster images and which is better?* Online at
http://www.anu.edu.au/ITA/corecomputer/notes/vector_raster_cc.html
Accessed on 8th December 2006.