PATH FINDING IN 2D GAMES

Cosmin Adrian Mandachescu

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2003

0-612-77991-2

Abstract

Path Finding in 2D Games

Cosmin Adrian Mandachescu

A major problem faced by game developers these days is the ability to effectively plan the motion from a *Source* object to a *Target* situated in an environment with several *Obstacles*.

This paper proposes a framework for game design in which all objects have a precalculated bounding box. Polygons with four or more edges (depending on the accuracy desired and the shape of the object) represent the bounding boxes. They are automatically generated at the creation of the object and are filtered to minimize the number of vertices while preserving the overall aspect of the object.

The vertices of all the bounding boxes from a region of interest (situated in the vicinity of the direct path from the Source to the Target) are dynamically triangulated using a triangulation algorithm. The result of such a triangulation is a 2D mesh situated in the empty space available for movement. None of the edges generated by triangulation will cross any hard object (source, target, obstacle.)

A path from the Source to the Target is then derived by navigating on the edges generated by triangulation as well as on the contours of the hard objects. Further smoothing is done by removing redundant points from the discrete path while avoiding collisions.

Acknowledgments

I want to extend my gratitude to all the people who helped me with my thesis.

I will start with Dr. Peter Grogono who introduced me to computer graphics and helped me discover the beauty of it. Due to the computer graphics course that he has taught I have decided about five years ago to pursue a career in this field. Dr. Grogono has supervised my thesis and helped me a great deal with his constructive comments and suggestions and also with the overall structure of my thesis.

I owe many thanks to Robert Carrier from Genicom Consultants and Madeleine Jean from ToonBoom Technologies for their professional support and their partial sponsorship of this thesis.

Thank you Mark Demay for taking the time to read it and for giving me your valuable feedback.

Next I want to thank the members of the examining committee for their informative and well documented remarks.

Finally, I would like to thank my wife Lidia for her never ending support during all the months it took me to finalize my thesis. I never would have finished it if it wasn't for her. To my wife Lidia and my son Robert this thesis is dedicated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Graphics; animation; games

One of the most common forms of art is the visual representation of the perceived reality. Art is not identical to reality but it is able to reflect it in a great detail. Since the beginning of time people were able to effectively express themselves and describe their surroundings through sketches, drawings and paintings. New tools and techniques were always developed to complement the existing ones. Computers are one of the greatest innovations of the last century. Since their creation computers are being used with great results in many areas but the most fascinating of all is the field of computer graphics and, in particular, the employment of computer graphics in the creation of entertainment.

With the help of computers people can, among other things, create, modify and display visual forms of art. Because of their speed they are the ideal tool used to create animation and bring images to life. Animation is not a representation of motion only, but can also be a representation of physical changes that can be light, natural phenomena, different physical states and changes in color.

An important part of the computer animation world is the creation and consumption of computer games. They are interactively animated visual art. Graphics, in a game, are the first thing that will strike a user. It can tell a lot about the quality of the game based on a few, simple to describe, criteria like the amount of detail used in its drawings or the realism of its animation. It is very important, for the success of a game, that independent or user driven characters in a game employ a high degree of realism in their movement.

1.2 Outline of the problem

Path planning is an important research area in the field of Robotics. It is also important for a set of 2D strategy games because: (1) some of the characters of such games need to move in their 2D world from one point to another; (2) the characters may or may not have prior knowledge of the world; (3) the characters may need to plan their movement before they start moving or they may move without a plan; (4) the characters may move on a discrete 2D grid or they may move on the real 2D space. Searching for a suitable path requires the consideration of several factors of which the most common are the game intended to run on, ability to avoid obstacles, speed, and path length. Figure 1 provides an illustration of the problem while Figure 2 illustrates the expected solution.

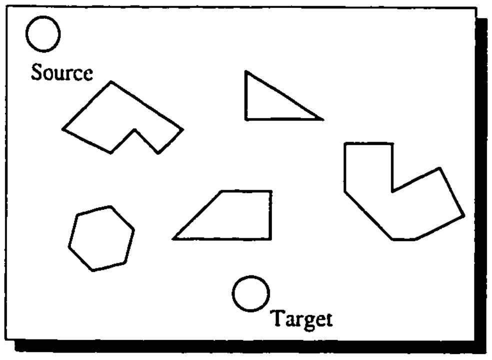

Figure 1: Problem

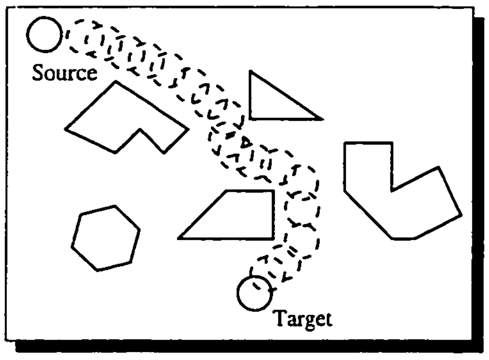

Figure 2: Solution

Being one of the most common problems in computer games, path finding has a multitude of solutions. They are not trivial and, although possibly very different, all otimal solutions have the common goal of reaching a target in the shortest time possible while avoiding stationary or moving obstacles. Artificial Intelligence and Robotics are the two areas to which a lot of path planning research is attributed.

The type of games usually benefiting from a Path Finding algorithm are

- First person shooters.
- Action adventure games.
- Point and click adventures.
- Squad based action games.
- Real time strategy games.

1.3 Possible solutions

If one has to design a game that requires a path finding algorithm there are several options available, depending on the processor speed, amount of memory, complexity of the game, user group for which the game is targeted, and other factors.

If the game is intended to be simple and has limited resources then a simple path finding algorithm may be the most suitable. The best pick might be to design a tile based game (see Section 2.1) and apply one of the well known shortest paths algorithms like Dijkstra or Breadth first search. The tiles of such a game will be considered as the nodes for the algorithm. If these algorithms are more complex than the game demands, then a simple trial and error algorithm can be used. If speed is an issue, then the well known A* algorithm will be chosen. However, more advanced players require a lot more realism from a game than tiles provide. Simple or awkward solutions are rarely accepted and might detract heavily from the overall quality of the game.

A more complex game might dedicate more resources to implementing a suitable path finding algorithm. Continuous 2D/3D movement of characters in a game complicates the solution since there are no intermediate states that can be used as nodes for an algorithm. The main problem faced by a path finding algorithm is the avoidance of obstacles. One way to solve this is to search the field of robotics for a suitable collision detection algorithm and adapt it to the specifications of the game.

1.4 Contributions of Thesis

This paper proposes a step by step solution to the problem of planning a path for an autonomous character in a 2D game. The main goal of this thesis is to define the necessary steps required to create games in which autonomous characters are able to find short enough paths between obstacles. Games do not necessarily need the absolute shortest paths but rather paths that can be found quickly and have an acceptable degree of smoothness. I will define all the necessary data structures required to achieve the goal and show a technique, along with the necessary algorithms that will allow a game designer to implement a complete solution to the path planning problem.

1.5 Overview of Thesis

This thesis is organized into six chapters and one appendix, where the first six chapters are labeled Chapters 1, 2, 3, 4, 5, and 6 and the appendix is labeled Appendix A. Chapter 1 introduces the reader to the problem of path planning and 2D games in general, and motivates the goals of this work.

Chapter 2 presents previous solutions to the path planning and is organized in 2 sections. The first section analyzes the previous path planning work done in 2D games and more specifically in the tile games design. The second section presents the more advanced path planning from robotics research.

Chapter 3 will describe the design of the program used to create game maps that are suitable for the path planning solution presented in this thesis. it also describes the design of a game that uses these generated maps. The implementation of the game editor and the game are further described in Chapter 4.

Chapter 5 shows the results obtained and uses several snapshots of a sample scene created with my program along with the algorithms applied to achieve our main goal which is to find a path between obstacles. Some of the measurements are compiled into a set of tables.

Finally in Chapter 6 I present the conclusion of this thesis and suggest some of the possible future work.

Chapter 2

Background and Related Work

Strategies developed to solve the problem of finding a path can be divided in two categories: *Tile Design* and *Real 2D Design* with each set having several approaches.

2.1 Tile Design

In this approach the background image as well as the characters, are made of tiles - rectangular pixmaps of predetermined size. Characters can move only on the tile grid and some background tiles may be designed as obstacles.

With *Tile Design* an optimal path can be chosen with each step the character takes, which we call the "plan as you go" approach, or the entire path may be calculated in advance, before the first step is taken.

2.1.1 Plan as you go

With this approach the source character does not know in advance about the position and shapes of the obstacles in the scene. It only knows the position of the target.

- Movement in a random direction

The character situated at the source moves toward the goal. If an obstacle is encountered, the character picks a random direction to move on. Eventually, if there are not many obstacles it reaches the goal (Figure 3(a)). However, in certain situations the character might get completely blocked especially if concave obstacles are encountered (Figure 3(b)). Another drawback of this technique is the fact that characters will behave oddly, choosing awkward paths.

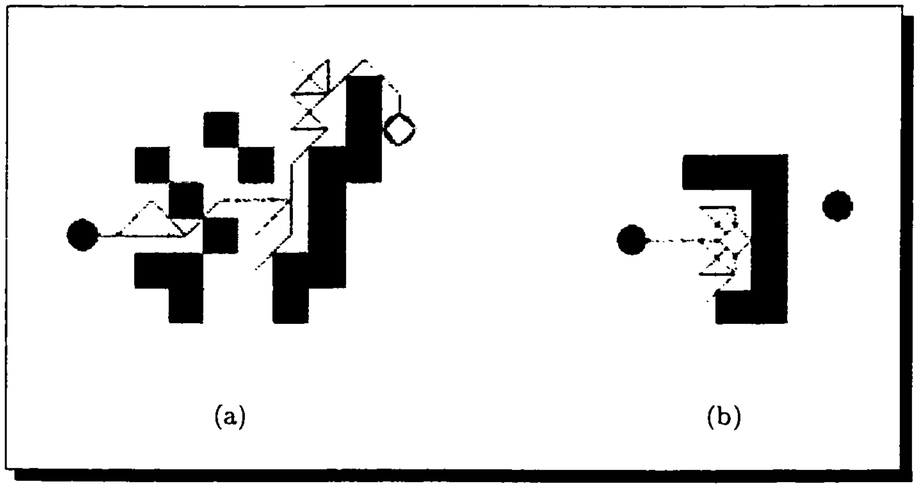

Figure 3: Random Tracing

- Tracing around the obstacle

  With this technique obstacles are avoided by tracing around them (Figure 4(a)), that is moving along their contours. Tracing represents the process of generating a path by recording all the steps that the source object takes in its search for the selected target position. The problem here comes in deciding when to stop tracing. A typical heuristic may be: "Stop tracing when you are heading in the direction you wanted to go when you started tracing". This would work in many situations, but Figure 4(b) shows how one may end up constantly circling around without finding the way out [Sto99].

- Robust tracing

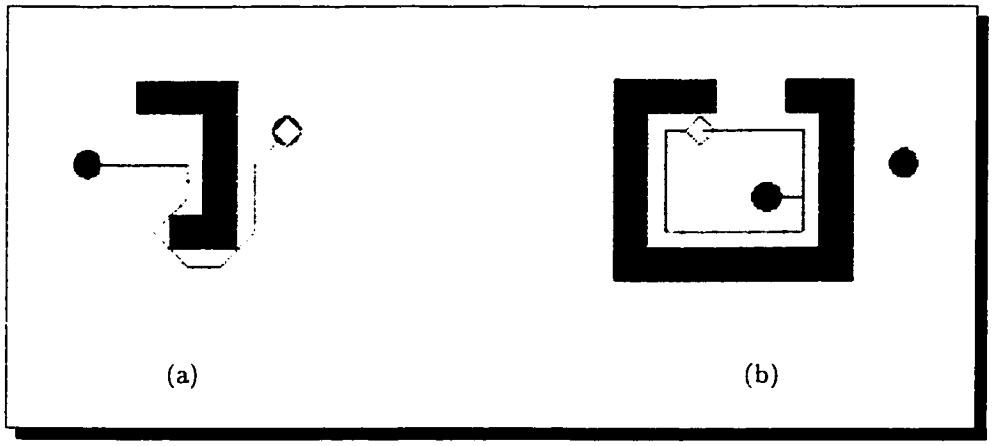

Figure 4: Contour Tracing

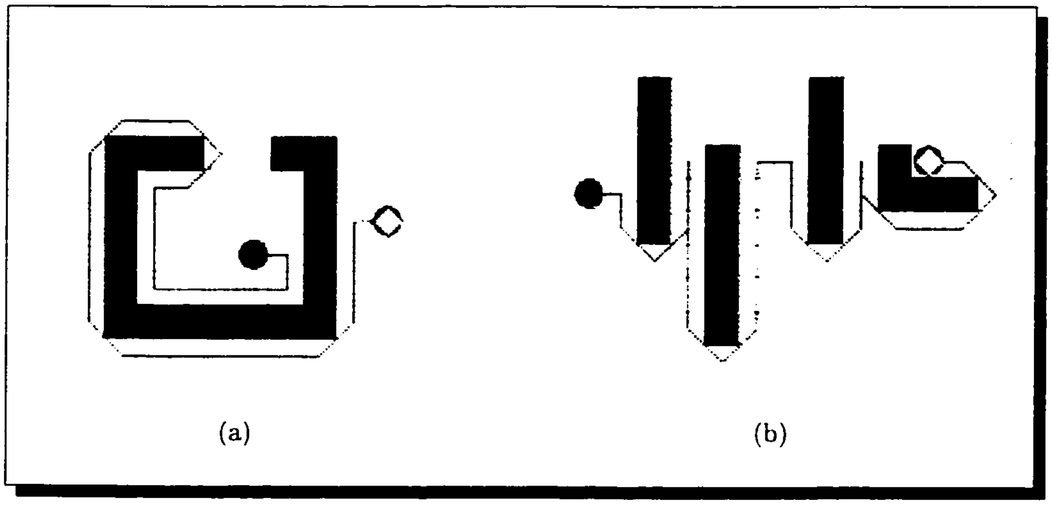

Figure 5: Robust Tracing

A more robust heuristic comes from work on mobile robots: When blocked, calculate the equation of the line from the current position to the goal. Trace until that line is crossed again. Abort if the character ends up at the starting position again. This method is guaranteed to find a way around the obstacle if there is one, as is seen in Figure 5(a). (If the original point of blockage is between the character and the goal when the character crosses the line, the tracing must not stop, or more circling will result.) Figure 5(b) shows the downside of this approach: it will often take more time tracing the obstacle than is needed, making it look pretty simpleminded though not as simple as endless circling. A happy compromise would be to combine both approaches: always use the simpler heuristic for stopping the tracing first, but if circling is detected, switch to the robust heuristic [Sto99].

2.1.2 Advance Planning

If the "Plan as you go" method does not adequately address the requirements of the game then a more advanced technique can be employed. In the *Advance Planning* method, the path is calculated entirely in advance, before the first step is taken. Several *shortest path* algorithms can be used in this case

- **Breadth-first search**

  With breadth-first search the tiles of the map are used as nodes for the network. The algorithm starts at the source node and examines all immediate neighboring nodes, then the nodes two steps away, then three, and so on, until the target node is found. With each step the visited nodes are marked and only new nodes are considered for the path. The following algorithm is used:

```
queue  Q
node source, tmp1, tmp2;
source.top = NULL;
push source on Q;
while Q is not empty
        tmp1 = pop from Q;
        if ( tmp1 == target node )
                while tmp1 != NULL
                        path += tmp1;
                        tmp1 = tmp1.top;
                return path;
        else
            for each neighbor tmp2 of tmp1
                  if tmp2.visited
                        continue;
```

```
                    tmp2.top = tmp1;
                    tmp2.visited = TRUE;
                    push tmp2 on Q;
    return NULL;
```

While breadth-first search guarantees to find the shortest path, its main disadvantage is that it searches all possible paths instead of directing its search toward the goal (Figure 6). Also, different steps in the path calculation have different weights. Diagonal steps are longer than orthogonal ones. An improved solution can be achieved with a bidirectional search, starting from source and goal at the same time (Figure 7).

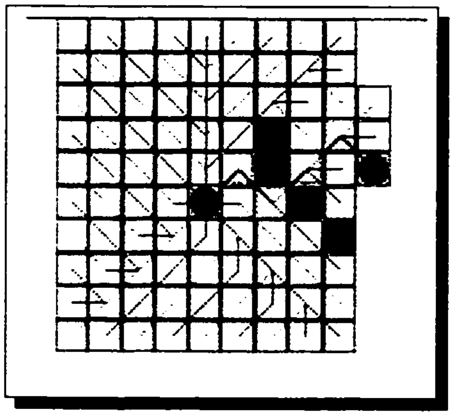
Figure 6: Breadth-first search

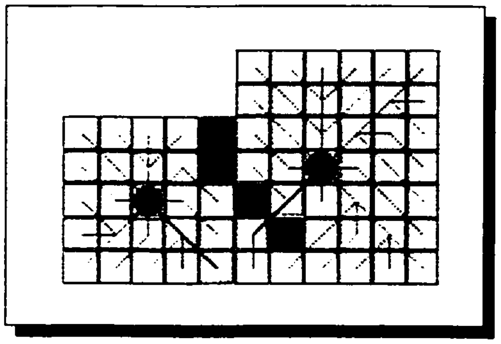
Figure 7: Bidirectional breadth-first search

- **Dijkstra's algorithm**

  Illustrated in Figure 8, Dijkstra's algorithm is more effective than breadth-first search. It searches for the shortest path based on node weight (length) and dynamically updates the weight of each node if better paths are found. Its main disadvantage, just like breadth-first search, is that it doesn't focus on the direction of the goal. The algorithm can be made more efficient by using a bidirectional search.

- **Depth-first search**

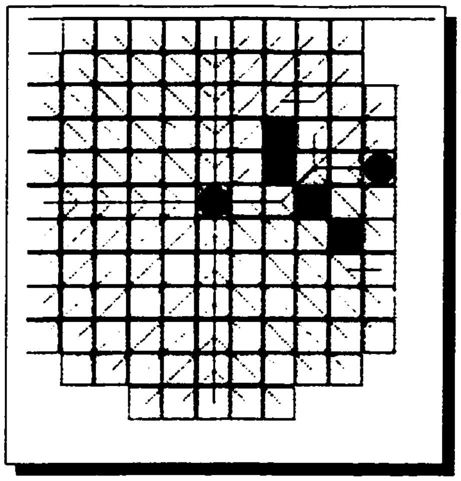

Figure 8: Dijkstra

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records this event by setting v's predecessor field $\pi[v]$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources [THC94a]. The algorithm is illustrated in Figure 9. As an improvement, the iterative-deepening depth first search starts searching paths that have a length at least the length of a straight line from source to the goal (Figure 10).

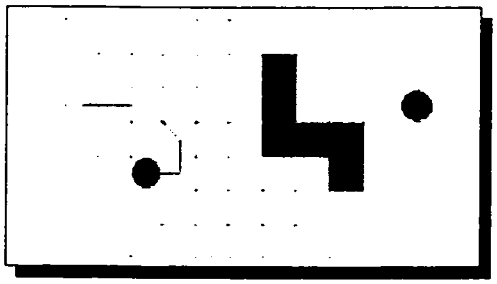

Figure 9: Depth-first search

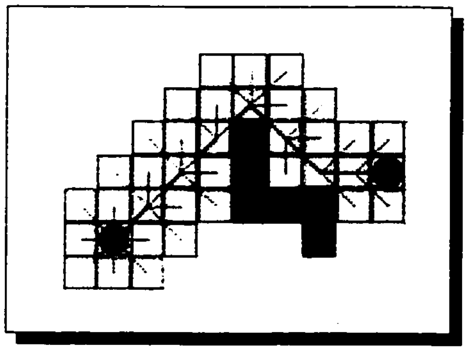

Figure 10: Iterative-deepening depth-first search

- **A* algorithm**

The A^* (pronounced A *star*) algorithm is an old workhose in the academic AI community, used since 1968 for solving different kinds of problems. In essence, the A* algorithm repeatedly examines the most promising unexplored location it has seen. When a location is explored, the algorithm is finished if that location is the goal; otherwise, it makes note of that location's neighbors for further exploration [DeL00].

A* has a couple of interesting properties. It is guaranteed to find the shortest path, as long as the heuristic estimate, $h(n)$, is admissible—that is, it is never greater than the true remaining distance to the goal. It makes the a very efficient use of the heuristic function: no search that uses the same heuristic function h(n) and finds optimal paths will expand fewer nodes than A*, not counting tie-breaking among nodes of equal cost. In Figure 11 we see how A* deals with situations that gave problems to other search algorithms [Sto99].

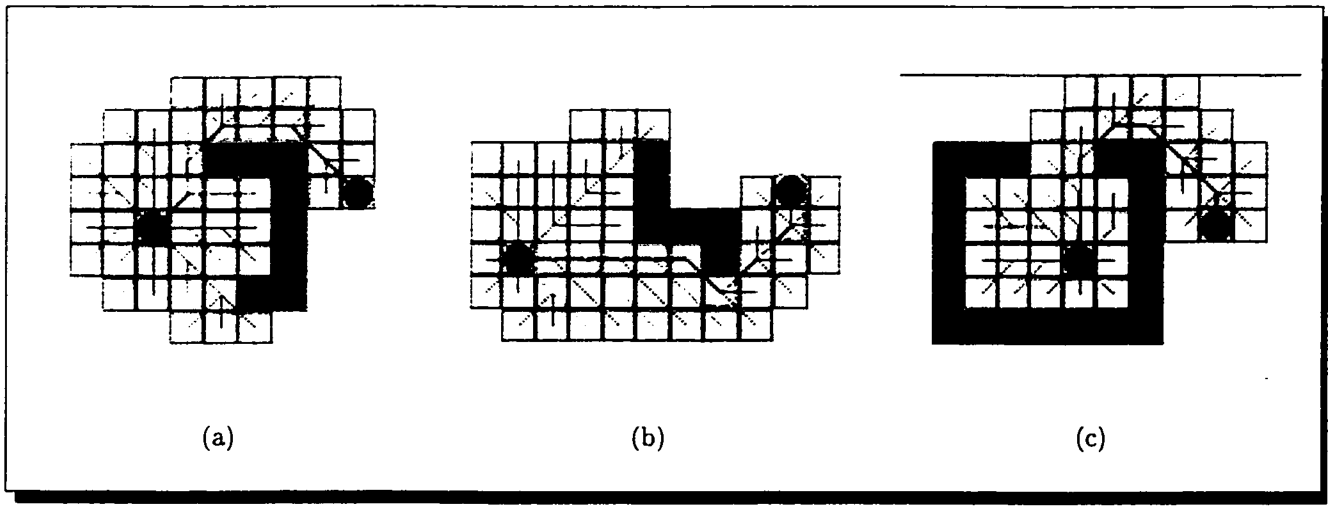

Figure 11: A* Search

The case in which A* is most inefficient is in determining that no path is possible between the start and goal locations; in that case, it examines every possible location accessible from the start before determining that the goal is not among them, as shown in Figure 12 [DeL00]

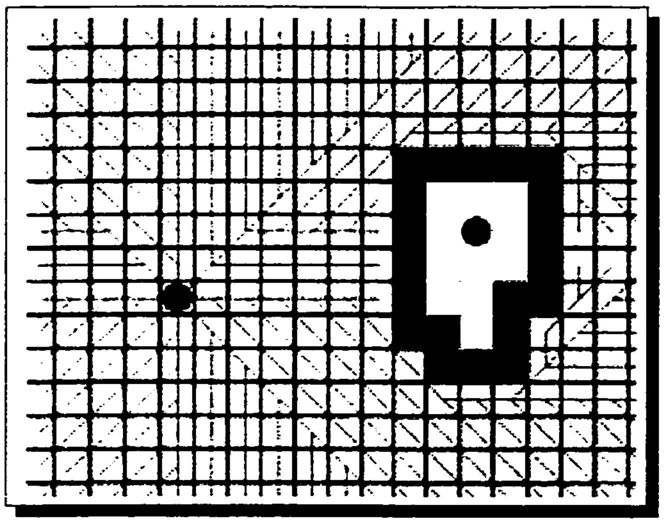

Figure 12: A*. No path to goal

All the pictures from this chapter have been created with the Pathdemo program created by Bryan Stout. It accompanies the book *Game Programming Gems* [DeL00].

2.1.3 Hexagonal Tiles

Similar to the *rectangular tile* approach this tiling design uses a hexagonal grid for its cells (Figure 13). Path finding is the same as with the rectangular tiles except that more calculations are required due to the increased number of possible directions. The main advantage is that it allows a greater degree of movement and, generally, the maps have a better look. Its main disadvantage is that the artwork is more difficult to create. Tiles are designed to be for general purpose, much like a puzzle that has to be put back toghcter in various ways. The difficulty with the hexagonal tiles comes when users have to reassemble the puzzle using hexagonal tiles instead of rectangular ones. This will not affect in any way the performance of the algorithms that calculate the shortest path.

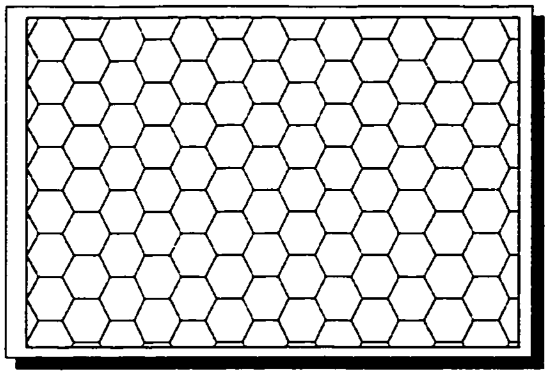

Figure 13: Hexagonal Tiles

2.2 Real 2D Design

With the *Real 2D Design* the objects and the background are made of nonuniform drawings (which can be later converted into polygons) that can take any shape and size. The notion of tiles and path search among adjacent cells can not be used here. Objects can move in any direction and their movement path can take any shape.

2.2.1 Motion Planning in Robotics

Finding a path through static known obstacles can be be solved by dividing the problem into two steps:

- Define a graph to represent the geometric structure of the environment.
- Search the shortest path within this structure.

In the early 80's Lozano-Perez introduced the concept of a robot's configuration space [LP83], [Lat91]. The robot is represented as a point - called a configuration - in a parameter space encoding the robot's degrees of freedom (dofs) - the configuration space. The obstacles in the workspace map as forbidden regions into the configuration space. The complement of these regions is the free space. Path planning for a dimensioned robot is thus "reduced" to the problem of planning a path for a point in a space that has as many dimensions as the robot has dofs.

The geometric structure of the free space can be found in several ways, most common being the *roadmap approach* and the *cell decomposition approach.*

The Roadmap Approach

The idea is to create a precomputed set of connected path edges on which an object may move. The shortest path will lie within this set of connected edges. There are few types of roadmaps among which we mention the *Visibility Graph* and the *Voronoi Diagram.*

Visibility Graph Here obstacles are represented by polygons. Vertices from each polygon (obstacle) are connected to the vertices of all the other polygons as long as the straight line connection does not enter the interior of any other polygon (obstacle). A graph (called roadmap) is formed in this way. Finding the shortest path, within this weighted graph, is reduced to searching only on the connected roadmap using a shortest path algorithm. Figure 14 from [Lat91] is an example of a roadmap.

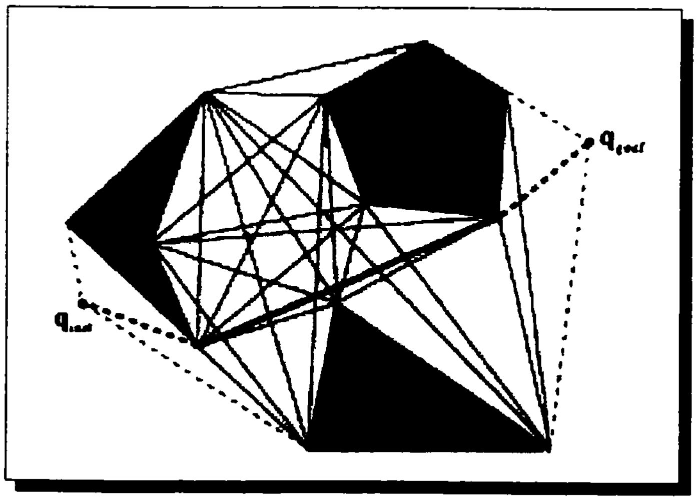

Figure 14: Visibility Graph

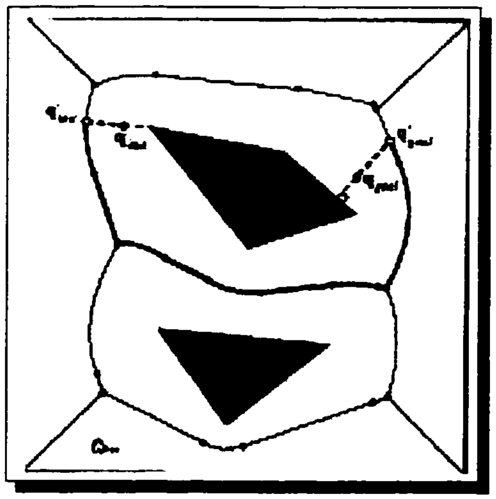

Figure 15: Voronoi Diagram

Voronoi Diagram Figure 15 [Lat91] shows a Voronoi diagram which represents a set of paths composed of points that are equidistant from at least two obstacles.

As opposed to the visibility graph this path keeps the moving object as far away as possible from touching the obstacles.

The Cell Decomposition Approach

Another way to partition the free space is to decompose it into a set of smaller areas free of obstacles. Two methods are available with this approach:

Exact Decomposition. Vertical lines are drawn from each vertex until an intersection with itself, other polygon, or the edge of the space is detected - figure 16. A dual connectivity graph (figure 17) is extracted and will be used to find the shortest path. The shortest path is drawn by connecting the centers of these subspaces.

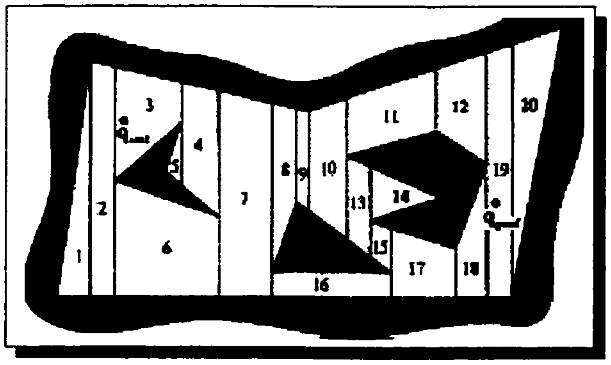

Figure 16: Exact Cell Decomposition

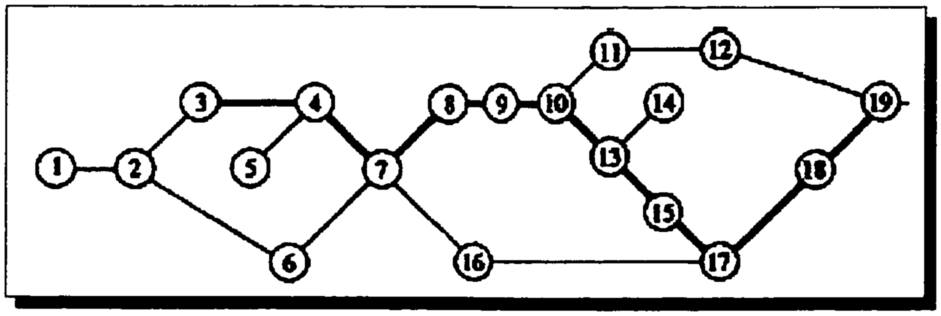

Figure 17: Graph Representation of Figure 16

Approximate Decomposition. This approach uses a recursive method to continue subdividing the cells until one of the following scenarios occurs:

- Each cell lies either completely in free space or completely in the obstacle region
- An arbitrary limit resolution is reached

This method is also called a "quadtree" decomposition because a cell is divided into four smaller cells of the same shape each time it gets decomposed. After the decomposition step, the free path can then be easily found by following the adjacent, decomposed cells through free space. A connected graph of the free adjacent cells is built and the shortest path computation reduces to a search within this graph. An example of this approach is shown in figure 18.

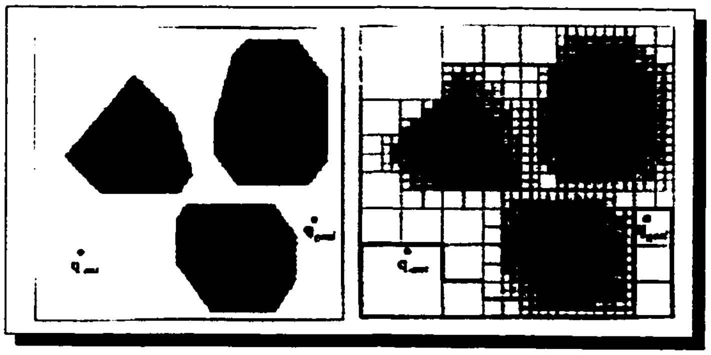

Figure 18: Approximate Decomposition

Figures 14 through 18 have been reproduced from [Lat91].

2.2.2 Steering Behaviors

Craig Reynolds [Rey99] has described a set of principles that can be used used as a basis for the movement of autonomous characters in animation or computer games. Although the term behavior means a set of complex actions and characteristics of a human, animal, or a complex mechanism, here it is used as a synonym for motion.

An autonomous character is an entity in a computer game or animation that requires little or no interaction from the user and is able to improvise its actions. It differs from the *avatar* in a computer game or virtual reality whose actions are directed in real time by a human player. It is also different from the scripted character from a movie or an animation whose actions are well determined.

For the purpose of clarity the motion of a character is divided into three layers:

- Action Selection: strategy, goals, planning
- Steering: path determination
- Locomotion: animation. articulation

We will only focus on steering and, while mentioning different steering types, we will emphasize the behaviors that can be applied to path finding. First, a simple, generic vehicle is defined. Complex shapes can also be *steered* in a similar fashion but we will keep the definitions as general as possible. The vehicle model is defined by its mass, position and velocity vectors, a maximum force and a maximum speed as well as its orientation.

The physics of steering is simple. At each time frame a steering force is applied to the vehicle's point mass. By dividing the steering force by the vehicle's mass we obtain a new acceleration which, in turn, multiplied by the current time frame will give us a new velocity vector. The new velocity vector is added the the old one resulting in a new speed and direction for the vehicle. The steering force is applied in incremental steps giving the vehicle a smooth overall trajectory. Although this velocity alignment model is simple, it is well suited for computer games characters. A more realistic model would have a lot more factors taken into account for movement calculation.

Different types of steering behaviors are:

Seek (or pursuit of a static target) acts to steer the character toward a specified position in global space. The behavior adjusts the character so that its velocity is radially aligned toward the target. The desired velocity is a vector in the direction from the character to the target. The steering vector is the difference between this desired velocity vector and the character's current velocity vector[Rey99]. (Figure 19)

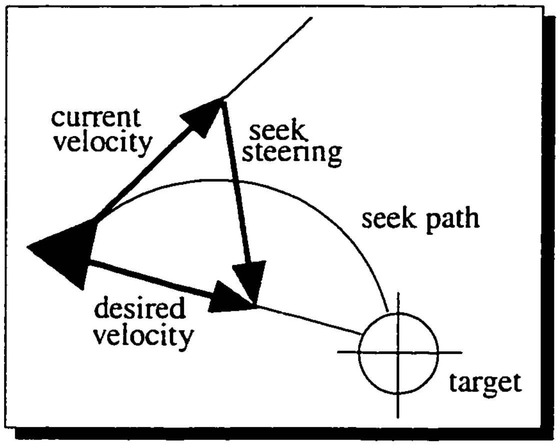

Figure 19: Seek Target

Pursuit or Evasion are similar to **seek** except that the vehicle steers toward or, respectively from, a moving target. Optimal techniques for pursuit or evasion exist in the field of control theory [Isa65]. In natural systems evasion is often "intentionally" nonoptimal in order to be unpredictable, allowing it to foil predictive pursuit strategies, see [CM96].

Offset pursuit refers to steering a path which passes near, but not directly into a moving target. The basic idea is to compute a target point which is offset by a given radius from the predicted future position of the vehicle, and then use **seek** behavior to approach that offset point [Rey99].

Arrival behavior is identical to **seek** while the character is far from its target. But instead of moving through the target at full speed, this behavior causes the character to slow down as it approaches the target, eventually slowing to a stop coincident with the target [Rey99].

Obstacle avoidance behavior relates to avoiding obstacles and not necessarily to *collision detection.* The distance between obstacles is considered large compared to the size of the vehicle. This assumption allows to approximate obstacles by large circles for the sake of simplicity. This behavior is similar to the **offset pursuit** behavior. The goal of the behavior is to keep an imaginary rectangle of free space in front of the character. The rectangle lies along the character forward axis and its height is equal to the diameter of the character's bounding circle. A collision test is done for each obstacle which is close enough and lies in front of the vehicle. The simple test considers only the center point (origin) and the radius of the circle that approximates the obstacle. If collision is detected the character is steered away from the closest colliding obstacle. An unrelated obstacle avoidance technique is described in [EW96] (Figure 20)

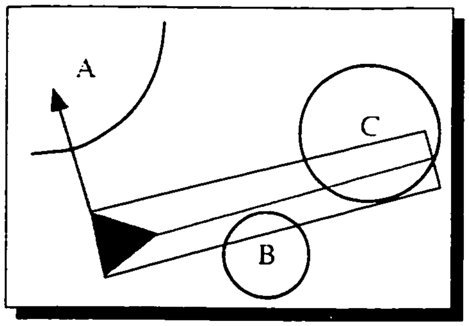

Figure 20: Obstacle Avoidance

Wander is a type of random steering in which the steering direction is retained and, with each frame a small displacement (in a random direction) is applied to it. This avoids an awkward movement generated by applying a random steering force with each frame. The awkwardness of the latter movement comes from the fact that the object changes direction and the magnitude of its displacement with each step. In the former approach the steering direction is kept and only small displacements are applied to it making the path a continuous curve as oppose to a set of random, connected segments. Another way to implement wander would be to use coherent *Perlin noise* [Per85].

Explore is a type of wander behavior in which the character exhaustively covers a region of space.

Path following behavior enables a character to steer along a predetermined path, such as roadway, corridor or tunnel. It does not constrain the character to rigidly follow the path but rather it produces a motion in which the the character moves very close to the path but it is also allowed to **wander** along it. This behavior is similar to a person walking across a hallway. The goal is to steer the character such that with each step it will move closer to the *centerline* of the path. This is accomplished by predicting the future position of the character with each step. This position is then projected on the *centerline* of the path and a steering force is applied to the character's point mass to steer the character toward the path (Figure 21).

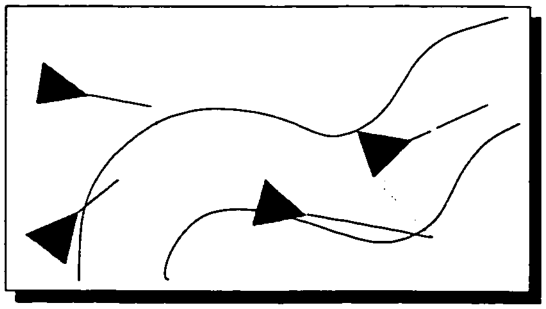
Figure 21: Path Following

Flow field following is a type of behavior in which the character steers to align its motion with the local tangent of a flow field (also known as a *force field* or a *vector field*).

Separation, Cohesion and Alignment relate to a group of characters. In each case the steering behavior determines how a character relates to other characters in its local neighborhood.

Flocking behavior can be obtained by combining the **separation**, **cohesion** and **alignment** to produce the *boids* model of flocks, herds and schools [Rey87].

Leader Following behavior causes one or more characters to follow another moving character designated as the *leader*. The implementation is similar to **arrival** behavior with the arrival point slightly behind the leader. If a follower finds itself in front of the leader it steers laterally before resuming **arrival** behavior. To avoid bumping into each other the followers use the **separation** behavior.

By combining some of the steering behaviors we can create a path finding model. If we have a source and a target objects we can draw a straight line path from the source to the target. Then we combine the **path following** behavior, to keep the source close to the desired path, with **obstacle avoidance**. The **arrival** behavior is used as the source approaches the target. If the target is moving we can also add the **pursue** behavior to our combination. Since the steering behaviors are well defined and relatively independent they can be implemented separately and used concurrently.

Chapter 3

Design

My thesis proposes a solution to the *Path Finding* problem that can be viewed as a combination of a modular design and some efficient techniques used to represent shapes as well as algorithms for navigating the 2D space. I will use "divide and conquer" since characters or obstacles in a game are small enough to make this technique an obvious choice. Also, movements of a character from one frame to another are small compared to the the full size of its navigational space. Shapes will be represented by polygons which are very flexible making this design scalable if more (or less) detail is desired in a character. We have to keep in mind that the more detail we use the more computation is required. Characters, obstacles as well as all the other objects from the game are expected to be hand drawn by an artist using a paint-like program that produces RGB pixmaps. I will not provide nor describe the paint program used to create the art but I will describe the output format that is expected from it. The polygons from my design will be the actual contours of the 2D art that will be extracted after the characters (or the other objects) are drawn and will be saved in data structures used by the game. They will be hidden from the player and only made "visible" to the algorithms using them. They will be the geometrical representation of characters. As for the 2D space that is available for movement (the empty space) I will represent it as a triangulated mesh. This representation, which will be automatically generated by a triangulation algorithm, is suitable for navigational algorithms, especially when a character has to navigate in tight spaces between obstacles.

To accomplish everything described above I will have to separate my design into two parts. First I will define a map editor used for the creation of the background maps. The design of the actual game will be done in the second part.

3.1 Map Editor

The *Map Editor* is an independent program used to create and modify maps. It is a simple tool that must be able to load graphics from a file and selectively paste them onto a canvas. The generated canvas is saved and used as a background scene for the game. Extra functionality will be added to the design of the *Map Editor* which will enable it to perform some otherwise expensive tasks. Much like a good compiler that evaluates some of the program's tasks at compilation time, the *Map Editor* will run a few computation-intensive algorithms whose output will be used by the game.

In Figure 22 we show the design diagram of the editor.

We must specify here that game maps cannot be generated automatically from graphics files unless terrain generators are used. Our game model does not address this possibility nor is it able to use the output of such terrain generators unless they obey certain rules. For the purpose of our design we will consider the maps to be generated interactively by a human player.

The *Map Editor* has a user interface connected to three modules. The **UI** is used by a player to interact with the main modules of the editor: the *Tile Editor*, the *Object Processor* and the *Space Processor*. The description of these modules follows.

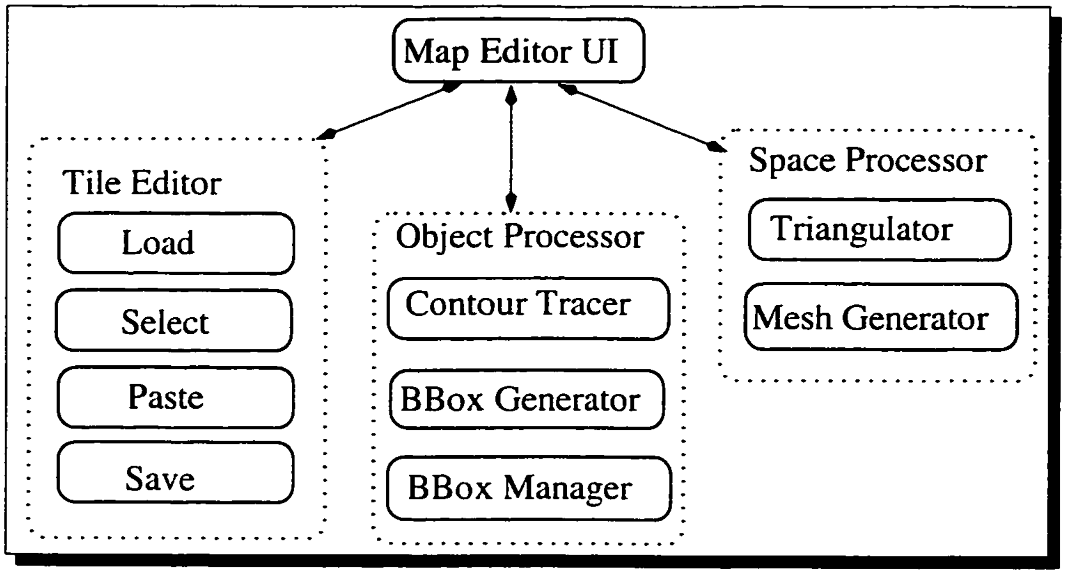

Figure 22: Map Editor

3.1.1 Tile Editor

The artwork for a game is created in a separate process by people with artistic talent and requires specialized software. For a tile based 2D game the final product is a set of pixmaps representing the "building blocks" for the graphical interface of the game. These pixmaps have usually identical sizes but this is not a rule. To simplify their management pixmaps are packaged together into a single file which is often referred to as the *tile set.*

A tile editor must be able to use such a tile set to create a background for a game. A more general tile set would have all pixmaps of equal size with the larger drawings broken into smaller tiles. This constraint will simplify not only the design of the tile editor but also the addition of new tile sets to the game.

We design our tile editor to have the most basic functionality which can be grouped into four parts:

- **Load.** To load a tile set we must know in advance several characteristics of the set:

Width and height of a tile

The number of tiles on a row

The number of rows in the set

The number of pixels between the edge of the tile set and the first tile (on the x and y axis)

The number of pixels between two tiles on the same row

The number of pixels between tiles of two consecutive rows

The file format of the tile set is of little importance since we can always convert it to a "supported" format. Internally the loader will convert the file into an RGBA format.

- **Select.** After loading, the tiles are displayed in a designated window from which users can select them. Only one tile can be selected at a time and its selection is marked by a highlighted contour.

- **Paste.** Selected tiles can be pasted onto a canvas by indicating a set of coordinates with a mouse click. The coordinates are then rounded to a grid such that pasted tiles do not overlap. Pasting is done by simply replacing the pixels from the designated "spot" on the canvas with the pixels of the selected tile. One or more tiles from the tile set is an empty pixmap and may be used to erase a previously "pasted" tile. The canvas onto which the tiles are pasted can have an arbitrary color which may be visible under a tile. Although tiles are rectangular the drawing contained by them can have any shape leaving the unused space completely transparent.

- **Save.** Background maps are created by repeating the select and paste process. When finished, the map will be saved in a file.

3.1.2 Object Processor

A background map, in its final form, contains objects that represent the static obstacles for the game. All the other objects that may change their position or orientation between two consecutive frames are added in a separate process and will be dealt with separately.

The role of the *Object Processor* is to identify the objects (obstacles) from a background map (presented as an RGBA pixmap) and to extract, classify and store all relevant information about them. To accomplish its tasks the *Object Processor* is divided into three components :

> **The Contour Tracer** contains a computation intensive process which has the goal of finding all the objects from a map and trace their outer contours. Objects might have holes in them (inner contours) which the process will identify correctly as holes but, since they are not relevant for path tracing, they will be traced and ignored. The holes have to be traced because they represent a contour and in order to find all the outer contours the process must find all contours and discard the inner ones. To find the contour of an object within a pixmap the object must differentiate itself from its surrounding. This is accomplished in the *Tile Editor* by having a distinctive color or transparency value (the A from RGBA) for the background pixels. The contour of an object is stored in a data structure as a set of consecutive points. A point is defined by a pair of x and y coordinates. Additional objects can be added to the game by tracing their contours in a separate process. All the objects that may be animated throughout the game can only be traced separately since single objects might have different contours for consecutive time frames.
>
> The process is shown in figure 23 (input) and figure 24 (output). The resulting contour for this particular sprite (pixmap) has 1190 points. The total number of points is high but can be substantially reduced in the next step.

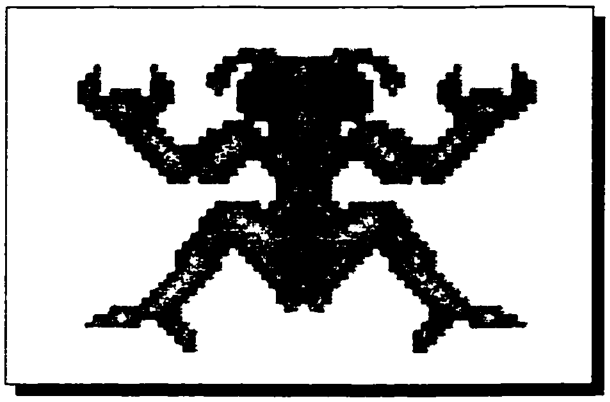

Figure 23: Input sprite

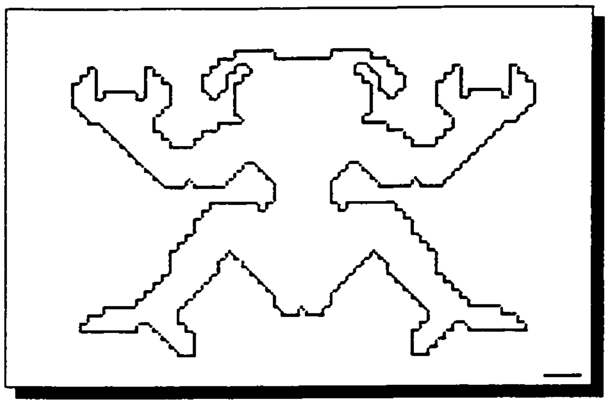

Figure 24: Traced contour

The Bounding Box Generator imports the traced contours and applies a series of algorithms to them in order to reduce their number of points while preserving their shape. A contour traced in the previous step contains one point for each corresponding *edge* pixel such that two consecutive points represent the coordinates of two adjacent pixels. The total number of points can be greatly reduced if we replace consecutive collinear points by the two points representing the ends of the corresponding segment. Further more we can have a tolerance of $\pm 5°$ or more for collinear points, i.e. three consecutive points making an angle of $180° \pm 5°$ are considered collinear. Typically, a bounding box of an object in 2D, consists of four points that are $P_1(x_{min}, y_{min})$, $P_2(x_{min}, y_{max})$, $P_3(x_{max}, y_{max})$, $P_4(x_{max}, y_{min})$. Such an approach approximates every shape with a rectangle and might be suitable for many applications including games. However, a more complex shape is better approximated by using more points.

Figure 25 shows a bounding box which has been filtered down to 170 points without a visible loss of detail. If more filtering is applied the contour loses detail as can easily be seen in Figure 26.

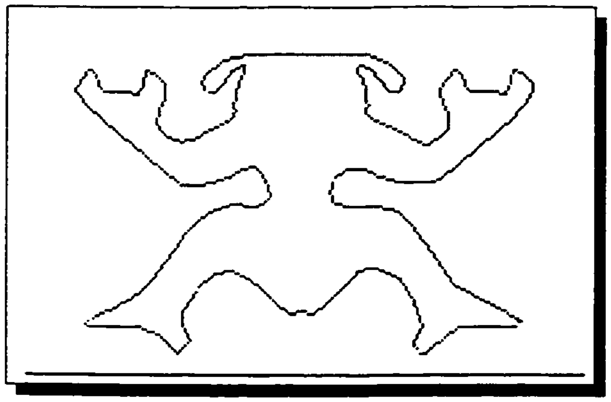
Figure 25: Contour - 170 points

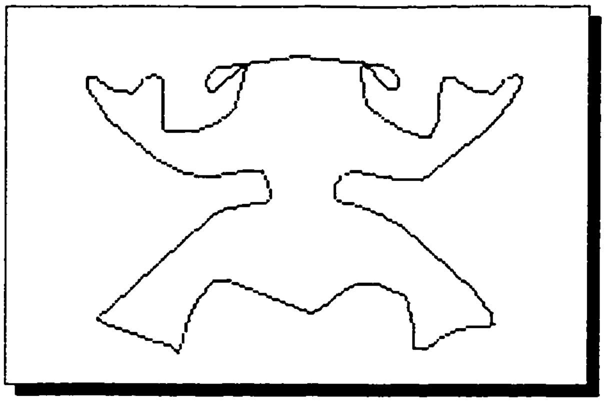
Figure 26: Contour - 120 points

The Bounding Box Manager is responsible for the storage and the management of the bounding boxes. Its main tasks consist of searching for a particular contour given the coordinates of one of its points, as well as finding the closest contour given a specific set of coordinates. To improve searching, each contour is stored together with its generic (rectangular) bounding box. This will ensure a fast selection of potential candidates for the more refined search.

3.1.3 Space Processor

After identifying and classifying all the objects (obstacles) from a map we are left with an unoccupied area (space). This space is the main terrain available for movement for the animated characters. It too can be covered with artistic drawings giving the user a more pleasant view of the game. These drawings don't have any functional value and can be added to the game at a later stage or simply be excluded from the *Contour Tracer*. The role of the *Space Processor* is to create a navigational mesh that will cover the entire available space and will exclude all the obstacles. Such a mesh is obtained by performing a *Delaunay triangulation* on the contour points of all the obstacles traced by the *Contour Tracer*. The mesh will be invisible to the game player and only be used for path computation.

At this stage we need to define several geometric terms as follows:

- **A Delaunay triangulation** of a point set (Figure 27) is a triangulation of the point set with the property that no point in the point set falls in the interior of the circumcircle (circle that passes through all three vertices) of any triangle in the triangulation (Figure 28).

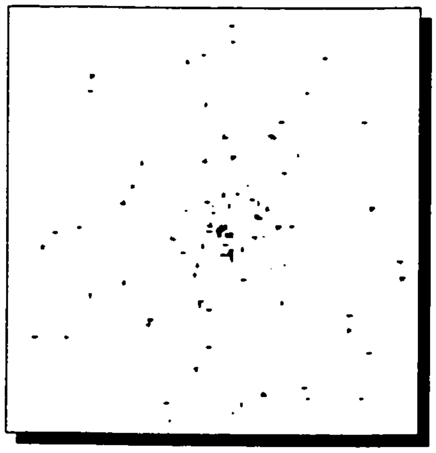
Figure 27: Point set

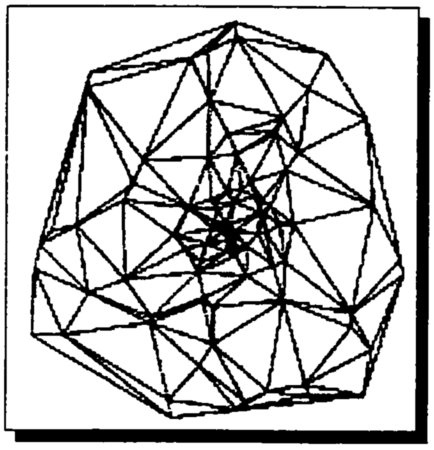
Figure 28: Delaunay triangulation

- **A Voronoi diagram** of a point set is a subdivision of the plane into polygonal regions (some of which may be infinite), where each region is the set of points in the plane that are closer to some input point than to any other input point (Figure 29). (The Voronoi diagram is the geometric dual of the Delaunay triangulation.)

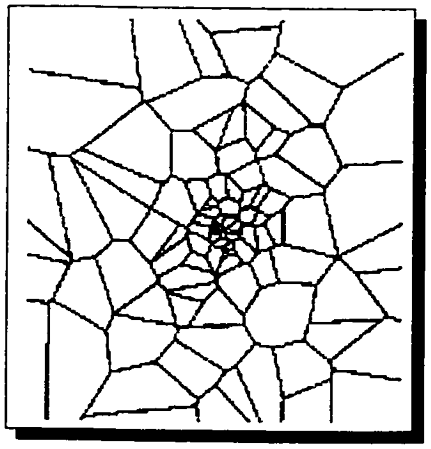
Figure 29: Voronoi diagram

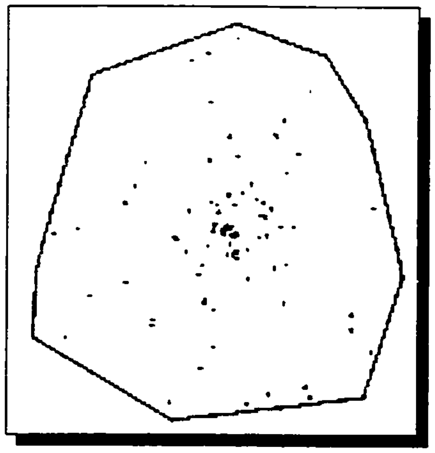
Figure 30: Convex Hull

- **The Convex Hull** of a point set **S** is the smallest convex polygon in the plane that contains all of the points in **S**. A polygon is convex if and only if for any two points in the polygon boundary, the segment connecting the two points is contained entirely inside the polygon. Also no extension of the segments on the boundary of the convex polygon lies inside the polygon (Figure 30).

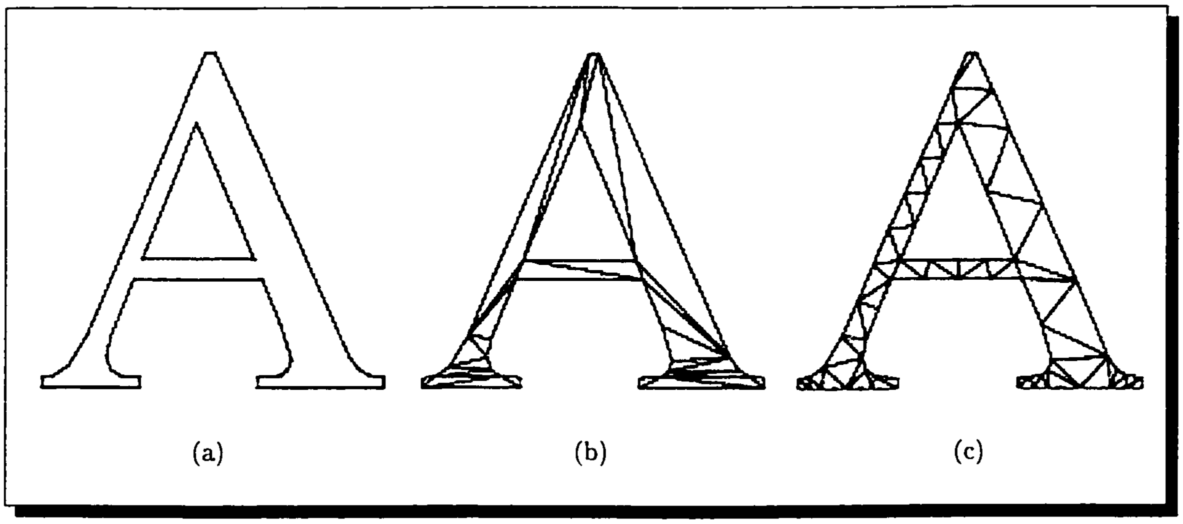

Figure 31: (a) PSLG, (b) Constrained Delaunay, (c) Conforming Delaunay

- **A Planar Straight Line Graph** (PSLG) is a collection of points and segments. Segments are edges whose endpoints are points in the PSLG, and whose presence in any mesh generated from the PSLG is enforced (Figure 31(a).)

- **A constrained Delaunay triangulation** of a PSLG is similar to a Delaunay triangulation, but each PSLG segment is present as a single edge in the triangulation (Figure 31(b).) (A constrained Delaunay triangulation is not truly a Delaunay triangulation.)

- **A conforming Delaunay triangulation** of a PSLG is a true Delaunay triangulation in which each PSLG segment may have been subdivided into several edges by the insertion of additional points, called Steiner points. Steiner points are necessary to allow the segments to exist in the mesh while maintaining the

Delaunay property. Steiner points are also inserted to meet constraints on the minimum angle and maximum triangle area (Figure 31(c)) [She96].

The navigational mesh created by the triangulation process is similar to the 2D grid from the tile-based games but has several advantages. Each triangle from the mesh can be compared to a grid cell, except that a triangle can cover a larger area since its vertices connect the obstacles from the game. With non-uniform triangles the mesh can represent very well the fine detail from the input which will generate many small triangles per surface unit, as well as large spaces that need only few large triangle for accurate representation. This representation will help the collision detection algorithms since large unoccupied spaces will be processed very fast compared to the processing of the same spaces in a 2D grid game design. Another advantage of the triangulated mesh representation is that one triangle is connected to three adjacent "cells" rather than four resulting in faster computation of the next direction.

3.2 Game

We proceed next to the design of the game itself. The idea is to keep it as simple as possible and emphasize only the aspects related to our goal which is *Path Finding*. The main components of the game are illustrated in Figure 32 and are described in the following subsections.

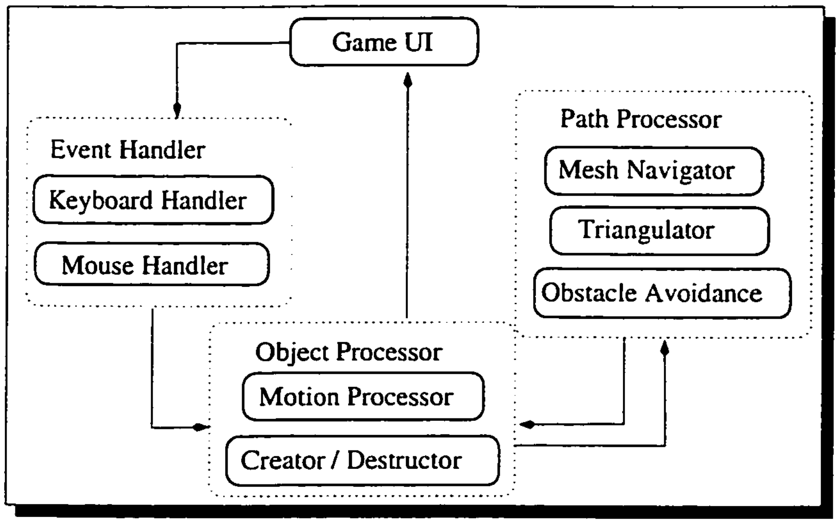

Figure 32: Game description

3.2.1 Event Handler

Our game must be interactive and, therefore, it must have a way to receive input from a user. The *Event Handler* receives events from the user and dispatches them to the appropriate receptors. For the purpose of this game we will consider as external devices only the mouse and keyboard. Other devices like joystick and steering wheel can be easily added at a later stage.

The *Keyboard Handler* is necessary because the keyboard represents a convenient way of sending commands to the game by simply pressing a key. In our case we have animated characters moving within the borders of the game and we want to be able to interactively make the characters appear or disappear. By pressing different keys we can control the creation/destruction of various characters, for example we can use **E** to create an eagle and **W** to create a woodpecker. With the keyboard we can also control some aspects of the character's motion like *Start*, *Stop*, move to the *Left*, *Right*, *Up* or *Down*. We can also increase or decrease the speed of a character by pressing predetermined keys.

The *Mouse Handler* receives various events like *Select* or *Destroy* a character. The mouse is also used to put a character in motion and to set or change its velocity. Simply select a character with a mouse click and, while keeping the mouse button pressed, chose a new direction and release the mouse button. The selected character will move on the direction set from the point where the mouse was pressed until the point where the mouse was released. The new speed will be a ratio of the length of the direction vector and the time it took to generate the new direction. That means that characters can reach fast speeds if the mouse is moved quickly, with the opposite being true as well.

3.2.2 Object Processor

This component controls the animation of all the characters as well as their creation and destruction. All characters available in the game can be duplicated and move independently. Their management consists of animating as well as showing and hiding the characters according to users requests (given by pressing different keys or mouse clicks.) Animation for these characters is achieved in two ways. The characters may change their position, direction of movement and speed from frame to frame. They also change their shape with moving. This is achieved by keeping a set of several different pixmaps for each character. Each pixmap from the set corresponds to a different time frame. The Object Processor is responsible for drawing each object's pixmaps in consecutive order while they are animated or drawing one of their pixmaps (from the set) when they don't move.

3.2.3 Path Processor

The Path Processor contains a triangular mesh generated by the triangulator and its main task consists of finding the shortest path from one point (Source) to another (Target) while avoiding obstacles. The triangulator generates a set of non-intersecting edges that connect the obstacles from the game. For static obstacles the triangulation was done in the Map Editor and saved into a file. This approach saves the processor for other computations during game play. Dynamic obstacles require the triangulator to generate the connecting edges every time objects move in a very active game. However, in this game I don't support dynamic triangulation at this stage. The Mesh Navigator is responsible for effectively choosing the edges on which to navigate during the shortest path calculation. It needs to keep the vertices along with their weight and neighbor list in a very efficient data structure (described in section 4.1.1) to be able to generate the shortest path in real time.

Chapter 4

Implementation

To simplify the implementation I choose a graphics engine that provides a rich interface and has a lot of functionality included. I have evaluated several game engines that seem good candidates for my design but most of them lacked appropriate documentation and examples. After some research I found the Qt library which is a free GUI library that can be used as a game engine as well. Its documentation can be found at [Tro02]. Its main advantage over most free graphics engines are rich documentation, plenty of examples, is developed in C++, it is largely used and has its own FAQ list, it comes together with the source code and last but not least the code developed with Qt runs on any platform (Unix, Linux, Windows, Mac) without any modification. Qt is mainly a GUI library and, therefore, most of its functionality deals with creating many types of buttons, lists, sliders, menus etc. It also has a powerful Canvas module that I will use to implement the game.

As for the implementation, the Editor and the Game will be developed in C++, using some powerful goodies that come with it like the Standard Template Library (STL). The entire code of this project was written by me in C++ with the only exception being the triangle library (used for triangulation) which I took from [She96].

4.1 Implementation of the Map Editor

One of the functionalities of the *Map Editor* is to load sprite maps which are graphics files containing art for the game. After some searching I found a rich sprite library (SpriteLib) created by Ari Feldman. The SpriteLib contains many maps for various types of games. The *Map Editor* will only use the two maps containing tile blocks to built the environment (Figure 33).

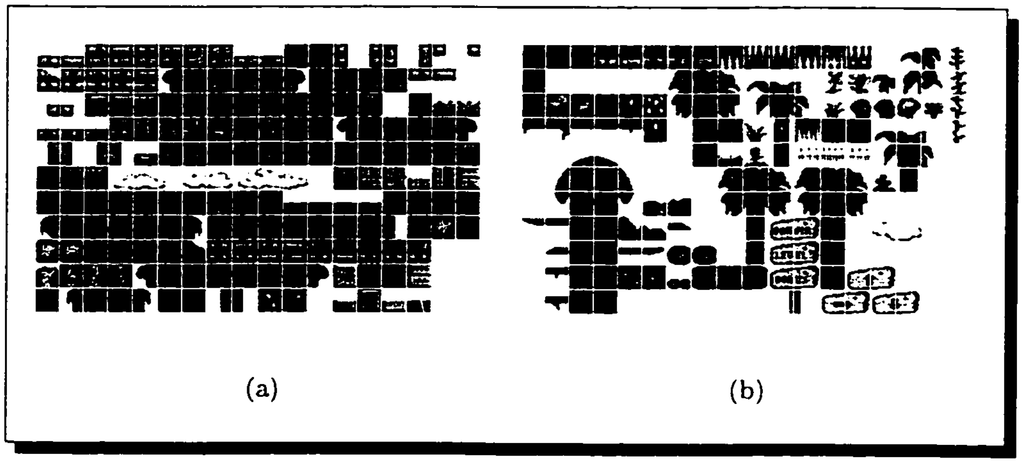

Figure 33: Sprite maps

The tile blocks are small pixmaps assembled together into a file. The two sprite maps used here contain 16 by 16 pixel pixmaps with a 1 pixel separation between them. The square shape of the pixmaps does not limit its content to be square, i.e. the art drawn into a pixmap does not need to cover its entire surface. It is important to know the specification of the sprite maps since they are not all the same. Usually, with free sprites, the game has to adapt to the specifications of the maps. A player using the *Map Editor* will only select these pixmaps and paste them onto a background map with the latter to be used by the game.

The main view of the *Map Editor* is shown in Figure 35. The editor is divided in two areas which we call views. On the left side we have the *Tile View* and on the right side the *Map View*. Both views are implemented using the **Canvas Class** provided by the Qt library [Tro02]. The *Tile View* is used to display sets of sprites which can be loaded from one or multiple files.

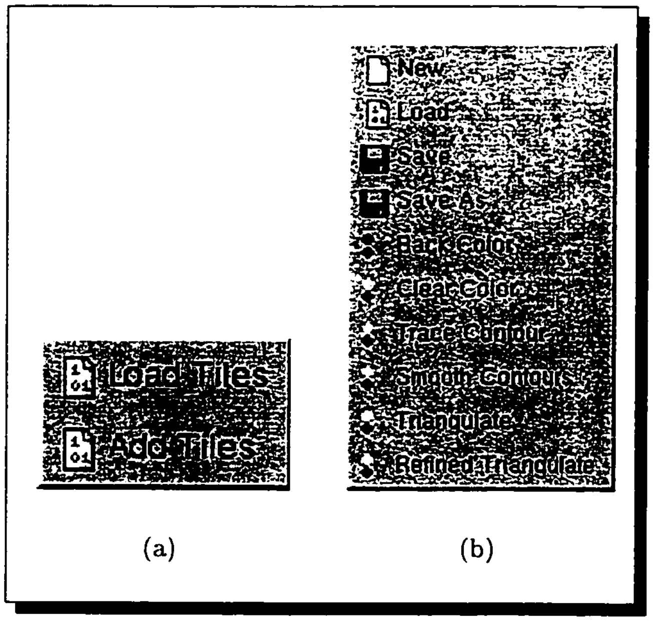

Figure 34: View Menus

Figure 34(a) shows the menu associated with the *Tile View*. The two options available are **Load Tiles** and **Add Tiles**. Both options are used to load sets of tiles into the *Tile View* and while the first option erases the entire view before loading the sprites, the second option adds a new set of tiles at the end of the previously loaded set.

The *Map View* displays the edited background map. The options available for this view are shown in figure 34(b) and they have the following functionality:

New erases the entire map and creates a new empty background of a user specified width and height.

Load opens up a previously saved background map. The currently displayed map is erased.

Save writes back to disk the modified background map.

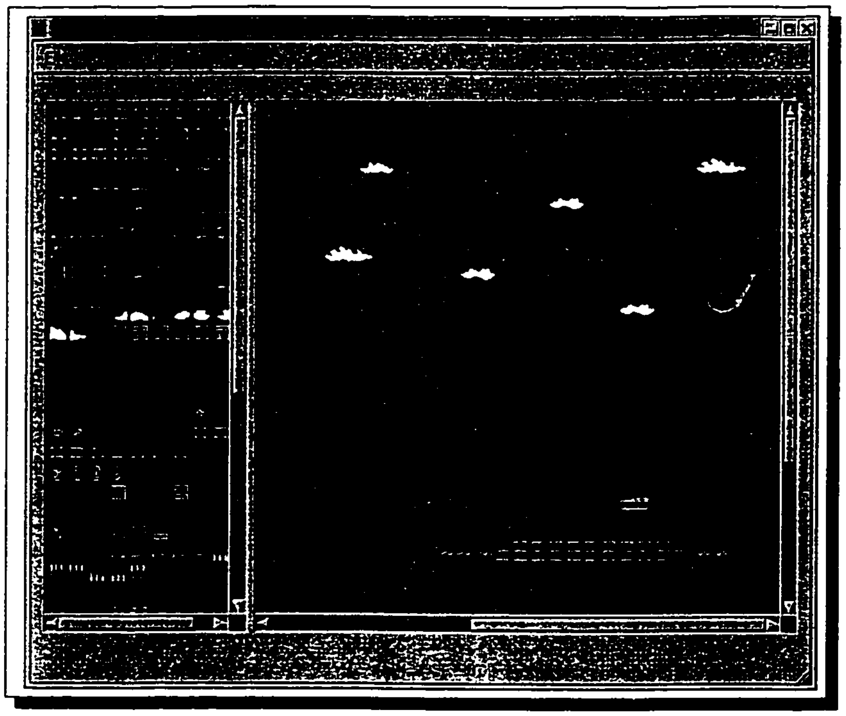

Figure 35: View of the map editor

Save As allows the user to save the current map under a different name.

Back Color opens up a color dialog and allows the user to select a new color for the background. The background color is replaced with the selected one. To differentiate between the color of the background and the color of the tiles we set the alpha value of the background color to zero.

Clear Color erases all pixels from the background map that have the selected color. A color is selected by picking it with the mouse from the background map. Pixels are erased by replacing their color with transparent black.

Trace Contour. After pasting tiles on the background map we trace the contour of all the objects and generate a list of polygons. Tracing the contour is equivalent to the extraction of the boundary of a polygon. Different polygons don't share any common areas, do not intersect in any point, and don't have holes. We are not interested in the information provided by the holes and, therefore, we discard them immediately after tracing. Due to the tracing technique we cannot have intersecting polygons since by tracing two intersecting objects will generate a single contour. Tracing the contour of an object will result in a set of connected points which lie on its perimeter. At this stage for every perimeter pixel of each object we generate a point in our list. A polygon is represented by a linked list of points with the first and last points being the same. By connecting the consecutive points of a polygon list we are able to recreate the original object's perimeter (see 3.1.2).

Smooth Contours applies smoothing and point reducing algorithms to the polygons traced in the previous step. Consecutive collinear points are removed and the remaining points change their coordinates to smooth the jagged edges of the traced objects. Jagged edges are always formed at tracing because, when tracing a contour on a pixmap every turn the algorithm takes is $\pm 90°$. A 45° straight line resembles a staircase when traced. Smoothing also changes the coordinates of all the points from a polygon to floating point numbers from the initial integer value obtained during tracing (see 3.1.3.)

Triangulate Uses the *triangle* algorithm by Jonathan Richard Shewchuk [She96] to obtain the *constrained Delaunay triangulation* of all the points of all the contours. In this triangulation every edge from the traced polygons is present as a single edge in the triangulation. All edges belong to a triangle and none of the triangles intersect. No edge resulted from tracing is shared between two triangles. They only belong to a single triangle. Some edges may not belong to any triangle if they lie on the convex hull of all the points of the entire scene (Figure 30.) The polygons traced in the previous step are passed to the triangulation algorithm as holes and, therefore, none of the additional edges resulting from triangulation will lie inside a polygon. The result of this process is a triangular mesh. The final mesh contains only the original points. No extra points are added during triangulation. (see 3.1.3 and figure 31(b).)

Refined Triangulate Is similar to the above triangulations with the exception that the triangulation is a *conforming Delaunay triangulation* obtained by adding extra points (called Steiner points) to the original set (see 3.1.3 and figure 31(c).) The advantage of using the conforming Delaunay is that the resulting mesh is made of smaller, uniform triangles. Its main disadvantage is the increased size due to the addition of the Steiner points.

The tiles from the *Tile View* are stored as separate pixmaps in a list. The pixmaps are drawn onto the view using a **DISPLAY_SEPARATOR** number of pixels between them. All tiles have a width equal to **TILE_WIDTH** and a height equal to **TILE_HEIGHT**. The width of the view area can be obtained from the view object using the method **width()**. As we mentioned above, a user will select tiles from the *Tile View* and paste them onto the *Map View*. Selection and paste are done with the mouse according to the following:

- **Selection**. Once the user clicks in the *Tile View* area, the coordinates of the mouse are used to find out which pixmap is selected by calculating its position (index) in the pixmap list as follows:

  int $w = \frac{width()-DISPLAY\_SEPARATOR}{DISPLAY\_SEPARATOR\ +\ TILE\_WIDTH}$;
  int $column = \frac{MouseEvent \rightarrow x()}{DISPLAY\_SEPARATOR\ +\ TILE\_WIDTH}$;
  int $row = \frac{MouseEvent \rightarrow y()}{DISPLAY\_SEPARATOR\ +\ TILE\_HEIGHT}$;
  int $index = row * w\ +\ column$;

- **Paste**. Once a tile is selected from the *Tile View* it can be pasted onto the background map starting at position $P(x, y)$ with, x and y computed as follows:

  int $x = MouseEvent \rightarrow x() - (MouseEvent \rightarrow x()\ \%\ TILE\_WIDTH)$;
  int $y = MouseEvent \rightarrow y() - (MouseEvent \rightarrow y()\ \%\ TILE\_HEIGHT)$;

4.1.1 Data Structures and Algorithms

As mentioned above, the programming language used for this project is C++. I have also used the Standard Template Library as much as possible [SGI03].
For the User Interface (**UI**) of this project I have used the Qt library [Tro02] which provides a rich set of widgets and also includes a Canvas module with graphics routines like drawLine which draws a line of a specified color from two points with integer coordinates. I will not describe the functionality of the Qt library nor the specific modules used to create the **UI** since they are not relevant to this thesis. I will however mention that I find Qt to be the most advanced UI development kit presently available.
The basic data structures I used for this project are:

Point creates a 2D point. Method names are intended to be self explanatory.

```
#include <list>
class Point {
public:
   Point(float p_X, float p_Y);
   Point() ;
   ~Point() {};
   Point& operator = (Point p_Value);
   int  operator  == (const Point& p_Value) const;
   int  operator  != (const Point& p_Value) const;
   float X() { return m_X;};
   float Y() { return m_Y;};
   float Distance(Point& p_Point);
private:
   float m_X;
   float m_Y;
};
typedef  list<Point>     PointList;
```

Distance returns the floating point distance from **this** point to p_Point. The operators are required in order to place the points in the STL list.

Segment is a simple 2 point set. I also define a list of segment and, therefore, Segment needs to declare some operators. Segment stores its length to avoid computing it all the time.

```
class Segment {
 public:
     Segment();
     Segment(Point& p_P1, Point& p_P2 );
     ~Segment() {;};
     float    Length() { return m_Length; };
     Segment& operator =  (Segment p_Segment);
     int      operator == (const Segment& p_Segment) const;
     int      operator != (const Segment& p_Segment) const;
     Point    operator [] (int p_Index) const;
     int     Intersect(const Segment& p_Segment, Point *p_Point = NULL);
 private:
     Point m_A;
     Point m_B;
     float m_Length;
};
typedef  list<Segment>   SegmentList;
```

The method Intersect computes the intersecting point between **this** segment and p_Segment. It return 1 if the segments intersect and returns the intersection in p_Point. Otherwise it returns 0.

Polygon is a list of consecutive points with first and last points being the same. It keeps the boundary information (contour) extracted from the distinctive objects from the edited map. In figure 42 is displayed the set of extracted polygons from the objects shown in figure 41.

```
class Polygon {
 public:
     Polygon();
     Polygon(PointList &p_List);
     ~Polygon();
     void AddPoint(Point p_Point);
     Polygon& operator = (Polygon p_Value);
     PointList& GetPointList() { return m_PointList;};
     int isHole();
     void Smooth();
     void RemoveCollinear();
     void RemoveDuplicates();
     // Given the segment AB return the intersection points and
     // the segments intersected
     int Intersect( Segment& p_AB, PointList* p_Points = NULL,
                    SegmentList* p_Segments = NULL );
     // computes an interior point to the polygon
     void InteriorPoint(Point& p_Point);
 private:
     // returns 1 if the three points are collinear
     int   Collinear( Point& p_P1, Point& p_P2, Point& p_P3);
     PointList m_PointList;
     //keeps its bounding box used by Intersect
     float  m_MinX;
     float  m_MaxX;
     float  m_MinY;
     float  m_MaxY;
};
typedef list<Polygon>   PolygonList;
```

Mesh is the structure that I chose to hold the set of all edges of all the polygons together with all the edges resulted from triangulation. I used a **Hash Map** to take advantage of the fast search available.

```
class HashFunc {
        public:
             size_t operator() ( Point p_Point ) const
             {
                  return ( size_t) ( p_Point.X()  + p_Point.Y() );
             }
    };
    typedef  hash_map<Point, PointList, HashFunc>  MapType;
    typedef  MapType::value_type ValuePair;
    class Mesh{
      public:
            Mesh();
           ~Mesh();
            void AddSegment(Segment& p_Segment);
            void RemoveSegment(Segment& p_Segment);
            SegmentList& GetSegments() { return m_Edges;};
            MapType& GetVertices() { return m_Vertices;};
      private:
            MapType      m_Vertices;
            SegmentList  m_Edges;
    };
```

Contour is the main class of my project and does most of the work from tracing the contour to finding the shortest path.

```
    class Contour{
    public:
           Contour();
          ~Contour();
          void TraceContour(unsigned char *p_Img,
                                 int p_Width, int p_Height);
          void SmoothPolygons();
          void Dijkstra( Point& p_Source, Point& p_Target,
                                 PointList& p_Points );
          void SmoothPath( PointList& p_Points );
          PolygonList& GetPolygons() { return m_Polygons; };
          Mesh&           GetTriangles() { return m_Triangles; };
```

```
private :
     int isEdgePixel(unsigned char *p_Img, int p_Width,
                     int p_Height, int p_Pixel );
     // turn left or right for tracing
     int changeDirection(int p_PrevRow, int p_PrevCol,
                         int p_Row, int p_Col,
                         int *p_NextRow, int *p_NextCol,
                         int p_Direction);
     PolygonList  m_Polygons;
     Mesh          m_Triangles;
};
```

To create a mesh a user will have to load one or more tile sets (see figure 33) into the left side of the editor (see figure 35). Then, using any combination of these tiles, the user must create a map by selecting tiles from the *Tile View* and pasting them onto the *Map View.* Once the desired map has been created the next step is to extract the contour. With a right click of the mouse in the *Map View* the menu containing all the options shows up (see figure 34.) By selecting the option *Trace Contour* the boundary information is extracted and displayed on the screen. At this stage the contours contain the maximum number of points possible - one point for each boundary pixel. The polygons are stored in the Contour Class.
A contour is extracted by starting at the rightmost pixel and keep going until the first edge pixel is encountered. An edge pixel is a marked (black or otherwise) pixel that has at least one of its 4 neighbors unmarked (white.) Once a black pixel is encountered turn left (relative to the direction of the last move) else if a white pixel is encountered turn right. Add all edge pixels in the PolygonList defined above. Stop and close the polygon when the tracing arrives at the starting point. Continue in the same way for all the remaining objects. Figure 36 shows the 4-Neighbors of a pixel as well as the principle of tracing around a contour.

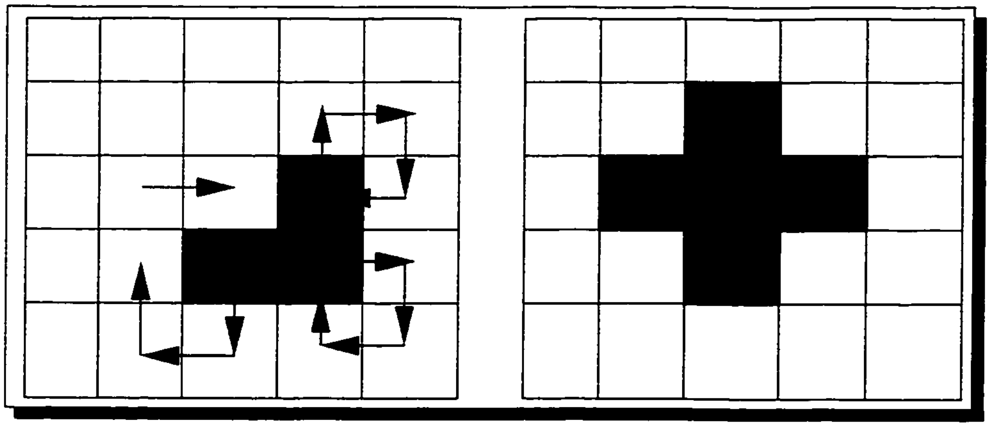

Figure 36: Tracing (left) 4 Neighbors (right)

The program traces the contour in the method **Contour::TraceContour**. After all the polygons have been traced and stored in the PolygonList of the **Contour** object the contours may be smoothed with **Contour::SmoothPolygons** to reduce the number of points. After tracing and smoothing, the polygons are triangulated using the *triangle* program [She96]. I have also defined a Triangle class which uses *triangle* to triangulate. I don't list the Triangle class here since it has mostly methods that deal with the specifications of *triangle*. At this stage we have created a mesh stored in the **Contour** object and displayed on the screen (see figure 43)

4.2 Implementation of the Game

The game, like the **Map Editor**, is implemented in C++ and uses the Qt library [Tro02]. The Canvas module of the Qt library provides routines to draw pixmaps. A timer is used to repaint the main view at regular intervals which I set to 90 ms. That means that every 90 ms different pixmaps can be drawn in different places which makes the game animated. The game is simple and its main function consists of drawing animated characters that move freely within the space of the game. The background on which the characters move is created by the **Map Editor** and contains obstacles. The game can handle any number of animated characters at the same time. Characters are created by pressing a specified key and duplicates are also allowed. In this game I have 8 types of animated characters and one animated explosion. Characters are animated in two ways. One is to change their positions with every frame and the other is to draw themselves differently every frame. Figure 37 shows some of the characters used by this game and their different pixmaps. All the characters used have been gathered from the internet. I have spent a great deal of time to find suitable characters for my game and to manually edit/scale/place each frame of each character to make it suitable for this game.

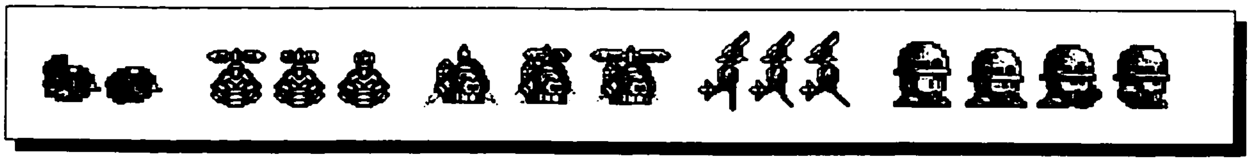

Figure 37: Different characters

The main view of the game, with some characters roaming freely, can be seen in Figure 38.

4.2.1 Data Structures and Algorithms

The classes used to hold the animated characters inherit from QCanvasSprite and the whole game is drawn on a QCanvas [Tro02]. An animated character can have any number of pixmaps. These pixmaps are drawn one after the other in consecutive frames. This gives a character the animated effect. The game employs the same data structures as the Map Editor described in section 4.1.1. I have added here an extra structure to hold a **Point** and its **Weight**. This structure is the main data type stored in the priority queue used by the shortest path algorithm. The class is defined as follows:

```
class PQElement {
 public:
     PQElement(Point& p_Point, float p_Weight = MAXFLOAT);
```

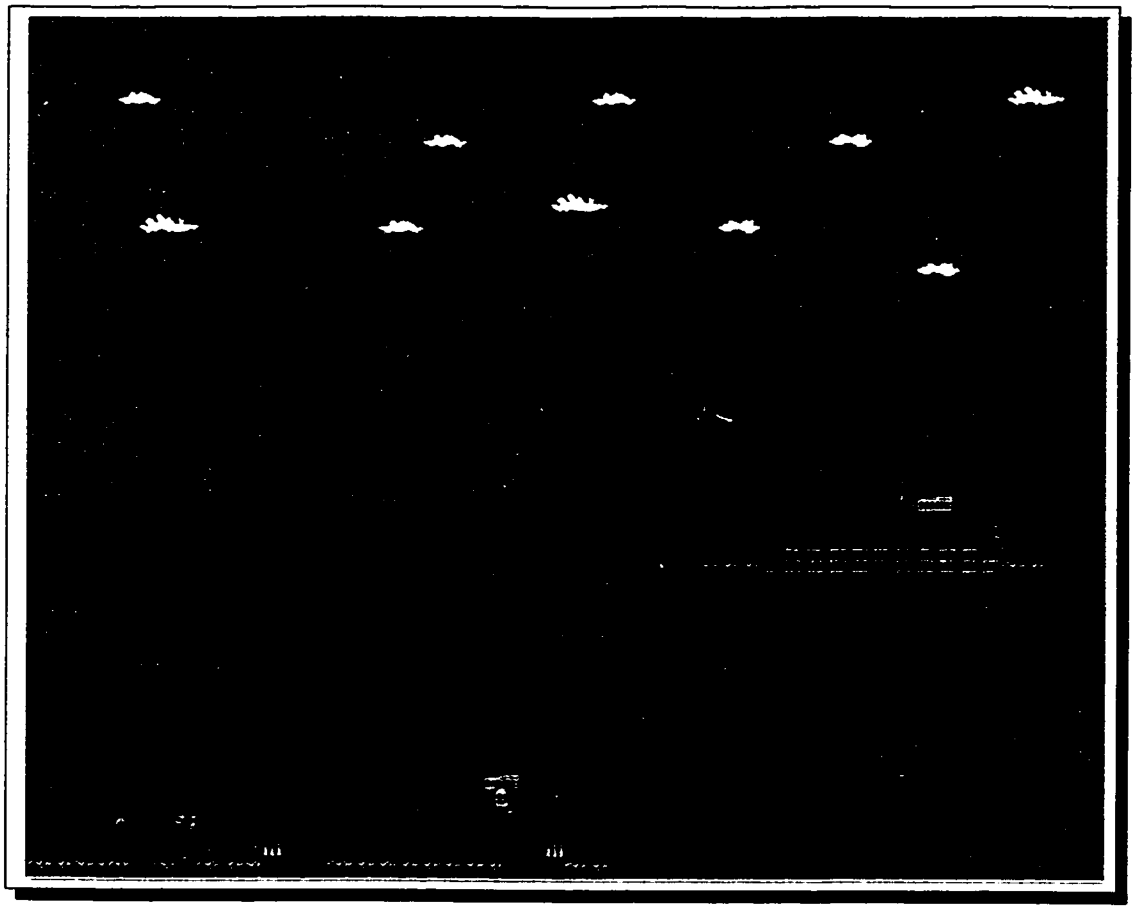

Figure 38: Main view of the game

```
    PQElement();
    ~PQElement() {};
    PQElement operator = ( const PQElement& p_Value );
    int operator <  ( const PQElement& p_Value ) const;
    int operator <= ( const PQElement& p_Value ) const;
    int operator >  ( const PQElement& p_Value ) const;
    int operator >= ( const PQElement& p_Value ) const;
    int operator == ( const PQElement& p_Value ) const;
    float  GetWeight() { return m_Weight; };
    Point& GetPoint()  { return m_Point; };
 private:
   Point  m_Point;
   float  m_Weight;
};
```

```
typedef list<PQElement> PQueue;
```

As with the previous classes the declaration of the operators is needed in order to store the objects in an STL list.

The algorithms used here are Dijkstra's *Shortest Path* which is defined in the **Contour** class as well as **Contour::SmoothPath** and **Polygon::Intersect**. I apply these algorithms to find the shortest path between 2 given Points A and B. The 2 points define a Segment which I will refer to as AB. To find the shortest path between A and B I compute the intersection of AB with all the Polygons from my PolygonList (stored in Contour). Let P be the intersecting Point that is the closest one to A and Q be the intersecting Point closest to B. In the next step I use Dijkstra's shortest path algorithm to find the shortest path (on the triangulated mesh) between P and Q. This algorithm is applied on the **Mesh** structure I use to store all the contours and their triangulation. After Dijkstra I have a path (represented as a PointList) consisting of a straight line from A to P, the shortest path from P to Q and another straight line from Q to B. In the last step I smooth this path by straightening out consecutive segments that do not intersect other polygons (another Polygon intersection check here). At the end of this stage a short enough path is produced. A short enough is not necessarily the shortest path but is enough for a realistic short path in a game. The algorithms employed here are my implementation and use the specific data structures described previously. Their description follows.

Polygon::Intersect. Check intersection of **this Polygon** with a given **Segment** If there is an intersection return 1 and also store every intersection **Point** in the given **PointList** and every intersecting Segment in the **SegmentList**. If there is no intersection return 0. To speed up the process I check first if any of the two Points of the given Segment lies inside the bounding box of the Polygon. The algorithm passes through all the Points of this Polygon (two consecutive Points form a Segment) and checks their Segment's intersection of with the given Segment. Let A,B be the given Segment's Points and C and D be two consecutive Points on the contour of the current Polygon. The directed line segments AB and CD are given by the equations:

$$AB = A + r \times (B - A); r \in [0, 1] \tag{1}$$

$$CD = C + s \times (D - C); s \in [0, 1] \tag{2}$$

If AB and CD intersect, then equation 3 is true for some r and s

$$A + r \times (B - A) = C + s \times (D - C) \tag{3}$$

By solving equation 3 for x and y we can find the values of r and s. If $r \in [0, 1]$ and $s \in [0, 1]$ then the two segments intersect. and their intersection point P is given by the equation 4.

$$P = A + r \times (B - A); \tag{4}$$

[ORo98] [Kir92]

Contour::Dijkstra I have implemented *Dijkstra's Shortest Path* algorithm following the description from [THC94b]. Again **STL** has been of great help here. Polygon::Intersect (described above) has returned the closest and farthest intersected Polygon Segments. It did not search for the intersections with the entire triangular mesh, specifically it didn't search within the segments resulted from triangulation. This approach effectively narrows the search and saves on computations. For implementation I used the priority queue **PQueue** which is defined above to be a **list<PQElement>**. I have also defined a hash_map [SGI03] to hold the computed predecessor of each vertex. The full implementation of Dijkstra's algorithm is available in the section A.

Contour::SmoothPath. The shortest path generated by **Contour::Dijkstra** contains a path composed of segments from the contour of the polygons as well as segments generated by the triangulation and represents only the shortest known path. This path is not smooth and is visibly longer than necessary but it is calculated quickly and contains a small number of points. **Contour::SmoothPath** will smooth this path by removing unnecessary points. The idea is that 2 consecutive Segments AB, BC might be reduced to the segment AC if by this elimination we don't create Polygon collisions. Figure 39 illustrates the output of the Dijkstra's algorithm run on the triangulated mesh. The path has 41 Points. After Smooth we are left with a better path and with only 7 Points.

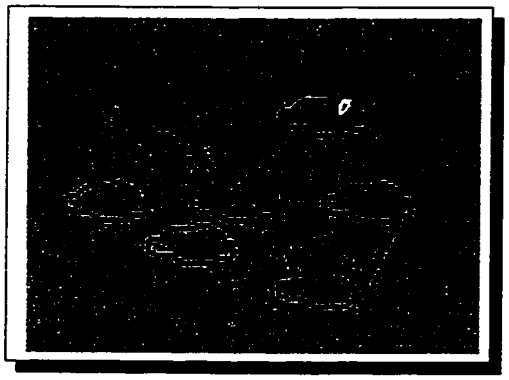
Figure 39: Shortest Path after Contour::Dijkstra

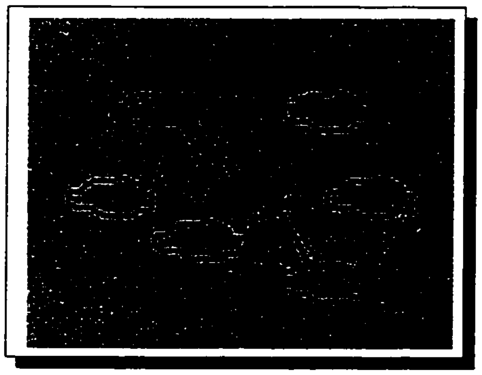
Figure 40: Shortest Path after Contour::Smooth

Chapter 5

Results

The results of this project are shown in Fig. 41 through Fig. 48 and Table 1 through Table 3. The computer used for this project has an AMD Athlon XP 1800 CPU and 256 MB memory. Figure 41 and Figure 47 are two scenes created with the MapEditor described in section 4.1. The former figure contains a scene with objects that have more or less a straight boundary, while the latter figure contains a set of objects with non-straight boundaries.

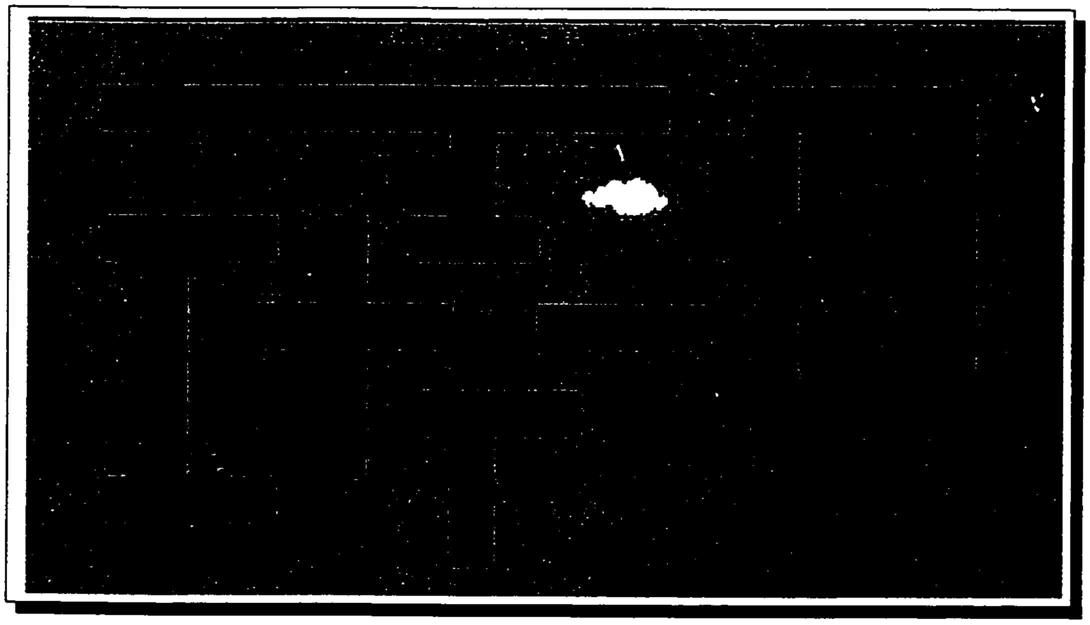

Figure 41: Sample Scene

After extracting the object boundaries (tracing the contours) from the two scenes and applying a set of smoothing algorithms (see section 4.1) we are left with a set of Polygons shown in Fig. 42 and Fig. 48. We show in table 1 the results of contour tracing and smoothing applied to the two maps. We can see that both maps generate about the same number of output points and it takes about the same time to extract and smooth their boundaries. The final number of points differ by a large amount because of their shapes.

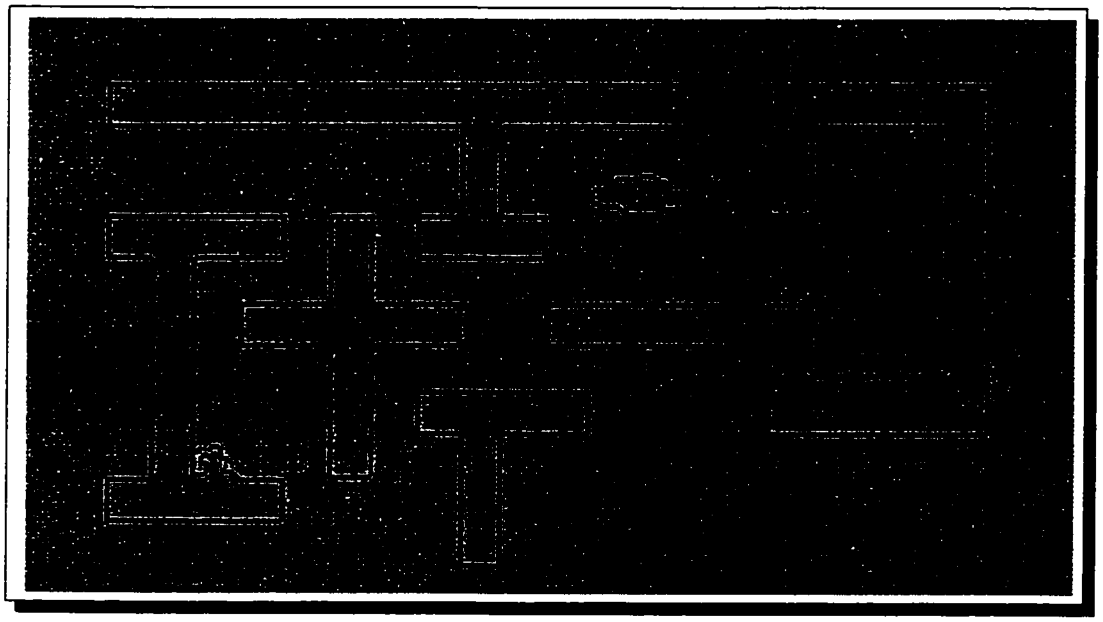
Figure 42: Polygonal Contour

Next we triangulate the polygons of Figure 41 using the two available triangulation methods. With the constrained Delaunay triangulation we have the same number of output points as we pass at the input and the result of triangulation is shown in Figure 43. Figure 44 shows the result of applying the conforming Delaunay algorithm to the same input set of polygons. The number of edges resulted from the latter triangulation is significantly larger because of the addition of *Steiner* points (see 3.1.3 and Fig. 31(c).) Table 2 shows the time it takes to triangulate the same input set of points using the two algorithms. The lower half of the Table 2 shows the result of applying triangulation to the same input image that has not been smoothed (see

Input	Trace time	Polygons	Total points	Smooth	Final points	Output
Figure 41	21 ms	7	2560 points	2 ms	327 points	Figure 42
Figure 47	22 ms	15	2519 points	1 ms	1117 points	Figure 48

Table 1: Tracing the contours of various scenes

Triangulation	Input points	Resulting edges	Time	Output
Constrained Delaunay	327	639	8 ms	Figure 43
Conforming Delaunay	327	4626	81 ms	Figure 44
Constrained Delaunay	2560	4641	81 ms	Not Provided
Conforming Delaunay	2560	10025	251 ms	Not Provided

Table 2: Results of various triangulations of the contour extracted from Figure 41

Table 1.)

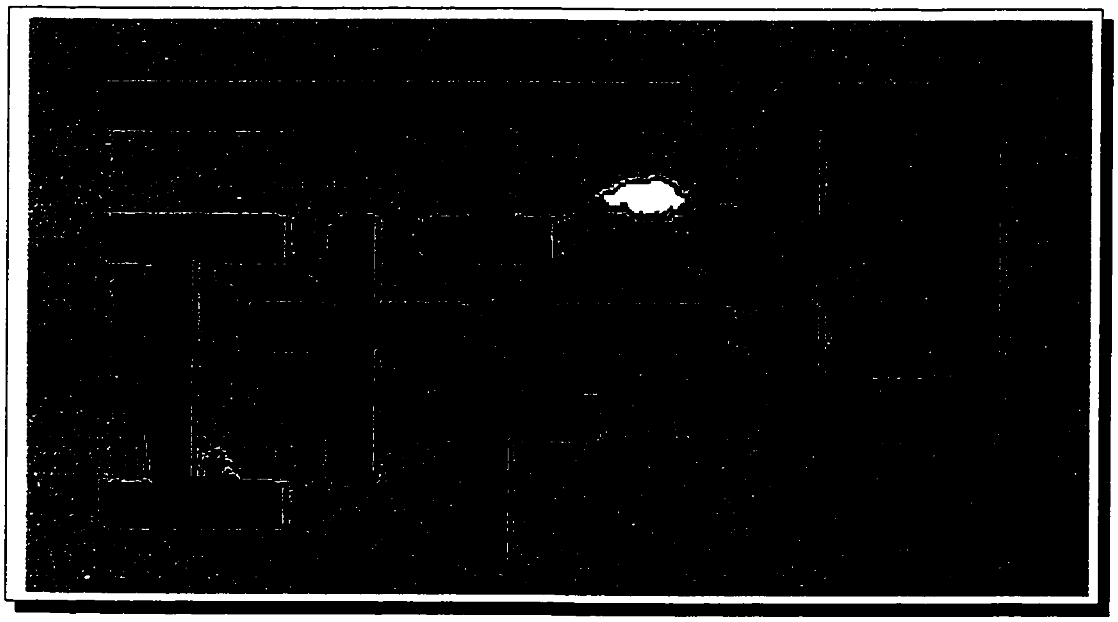

Figure 43: Constrained Delaunay Triangulation

Finally, Figure 45 and Figure 46 show the shortest path computed using Dijkstra's algorithm and smoothed with Contour::Smooth. The chosen trajectory is shown as a straight dotted line while the final, shortest found path is shown as a bold dash-dotted line. The statistics of the final algorithms are shown in Table 3.

Figure 44: Conforming Delaunay Triangulation

Number of edges	Time for Dijkstra	Shortest Path	Time to smooth	Final path
639	2 ms	18 points	0 ms	8 points
4626	169 ms	54 points	0 ms	13 points
4641	68 ms	56 points	1 ms	7 points
10025	198 ms	271 points	9 ms	12 points

Table 3: Results of applying Dijkstra algorithm to find a path for a chosen trajectory

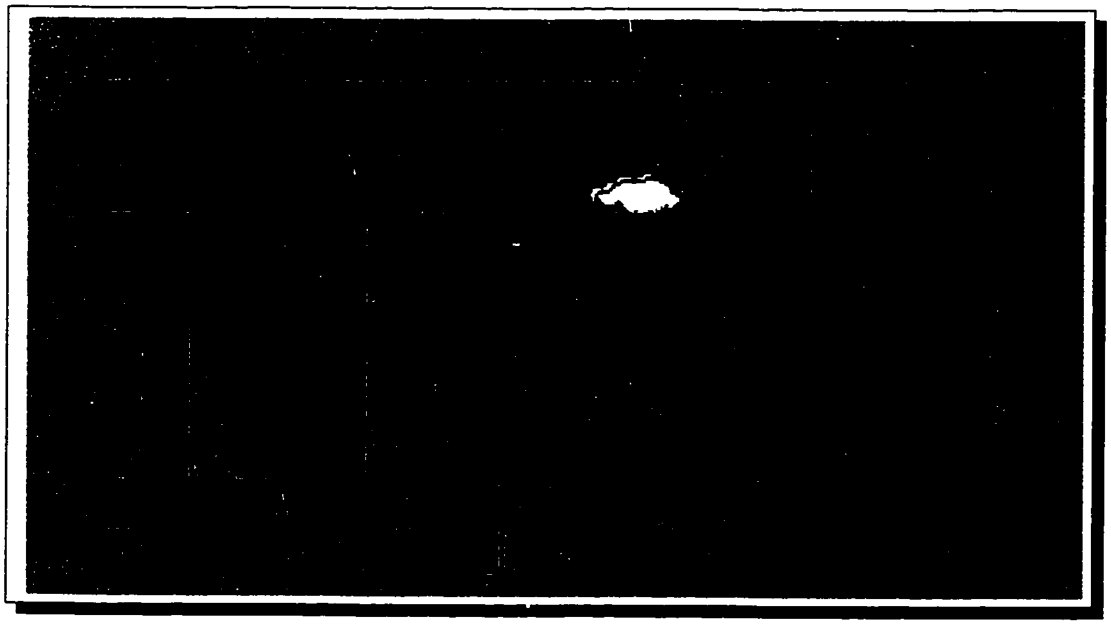

Figure 45: Shortest Path - Object View

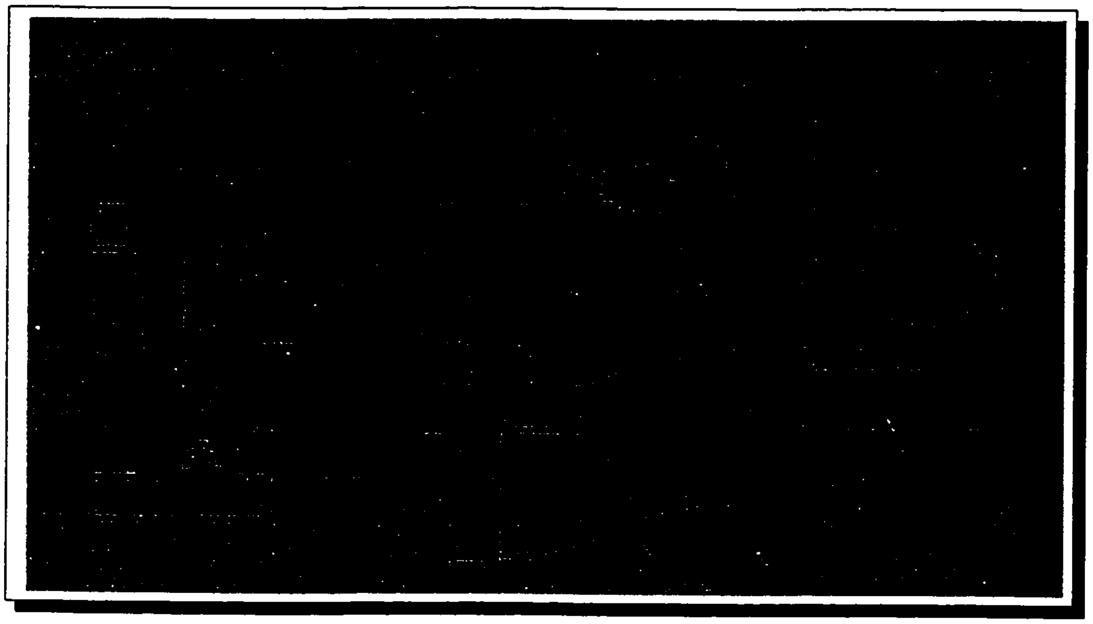
Figure 46: Shortest Path - Polygon View

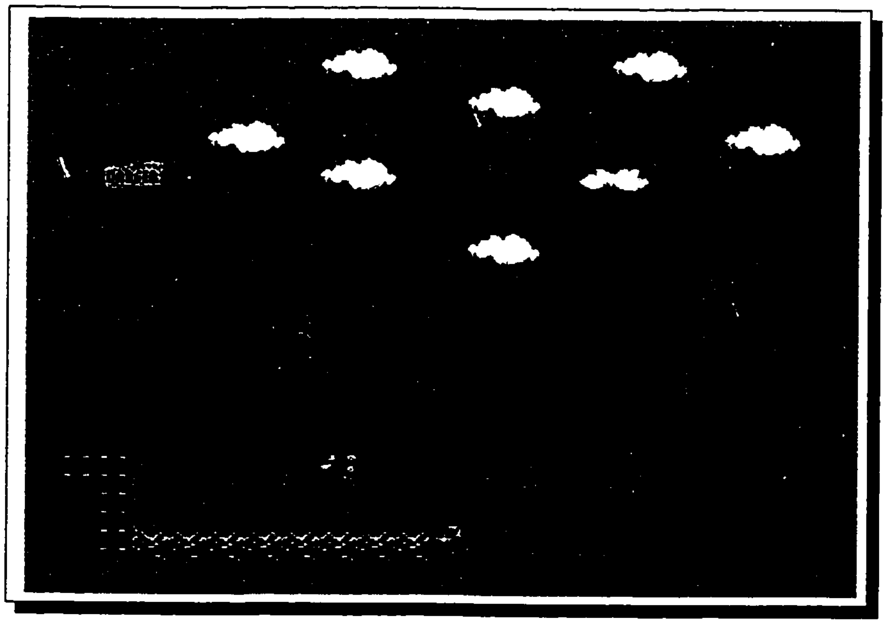
Figure 47: Rich Scene

Figure 48: Polygonal Contour

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Path Planning in a 2D game is a complex task which require custom made algorithms depending on the specifications of each game. While some games employ simple *plan as you go* techniques others require the computation of the entire path before any movement starts. In this thesis we have developed a step by step solution for Path Planning for a 2D game where the input is an image representing the background of the game and the output is a short, obstacle avoiding path between two selected points on the image. We have successfully combined the solutions of various problems from computational geometry (triangulation) and pattern recognition (contour tracing) and applied them to our solution of the Path Finding algorithm. We have defined a compact and sufficient set of necessary data structures used to accomplish our goal. We have made an effective use of computer graphics and imaging techniques to demonstrate our solution. We make the claim that our approach has a clear advantage over other path planning approaches because of the following:

- It is a well defined incremental approach to find a short enough path for a 2D game populated with obstacles.

- It can handle any type of nonuniform obstacles like the drawings from a game.

- It adapts well to the various shapes of the obstacles. The contours generated contain minimal number of points necessary to reconstruct the shape. Shapes with straight boundaries contain fewer points than nonuniform shapes. Fewer points means fewer edges resulted from triangulation and faster computation of the shortest path.

- While it searches for a path on a 2D triangular mesh, the final shortest path does not have to be a part of that mesh. In other words we have combined the advantages of using known shortest path algorithms like Dijkstra's with a smoothing technique that is allowed to take shortcuts.

- By using a combination of C++ with STL we have minimized the amount of code that a programmer has to write.

- The triangulation technique offers the possibility of removal some redundant edges (see 6.2). The result of that will be a general mesh with not only triangles but nonuniform shapes as well that will be smaller and, therefore faster to process.

- The triangulation used is more effective than using a Visibility Graph since it generates fewer edges and a shortest path algorithm runs faster on fewer edges. As can be seen in the figure 49 the triangulation of this simple scene generates 25 edges (including the edges of the traced contour). The Visibility Graph of the same scene generates 58 edges (figure 50. We can safely claim that the edges resulted from triangulation are a significantly smaller subset of the edges resulted from Visibility Graph computation.

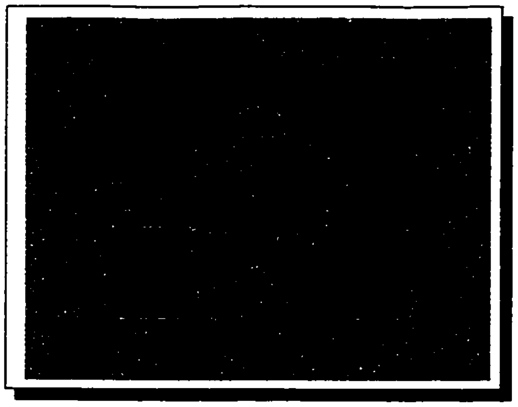

Figure 49: Triangulation

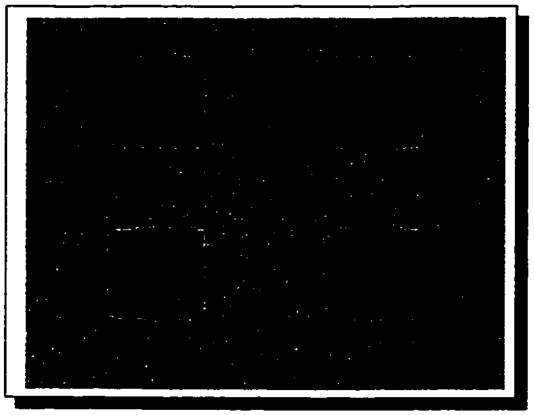

Figure 50: Visibility Graph

6.2 Future Work

The ultimate goal of work initiated in this thesis is to create a framework for the design of 2D games that require an advanced way of path planning in which the path must be determined before the first step is taken. While the solution described has successfully accomplished the initial goal it also raises new problems that should be addressed in future work.

Smaller Navigational Mesh. If we start out with a scene that contains many irregular shaped objects (see fig. 47 and table 1), after extracting the boundaries and smoothing the contours we are left with a large number of contour points. This happens because segments, defined by two consecutive boundary points, often change their direction and therefore cannot be eliminated by the algorithm from the method **Contour::SmoothPolygons**. Triangulation of such polygons will generate a high number of edges that are very close to each other because polygon segments have a short length and represent one triangle edge each. This problem can be solved by removing the edges resulted from triangulation that connect the same objects but are too close from each other.

Faster Shortest Path Calculation. While Dijkstra's shortest path calculation is very efficient and guarantees to find the shortest path it does not consider the direction of the goal. Another future work task would be to replace Dijkstra's algorithm with bidirectional Dijkstra (start searching from source and target at the same time) or with the A^* algorithm which considers the direction of the goal and constantly tries to move in that direction before exploring others.

Fitting Through Obstacles The Shortest Path computed in this thesis does not take into consideration the shape and size of the source character and might generate a shortest path within narrow empty spaces through which the source character does not fit making it an invalid path. Future research in this direction will always consider the shape and size of the source and will generate paths that lie outside of the Minkowski Sum convolution spaces. Refer to figure 51 to see the input and output of Minkowski Sum computation.

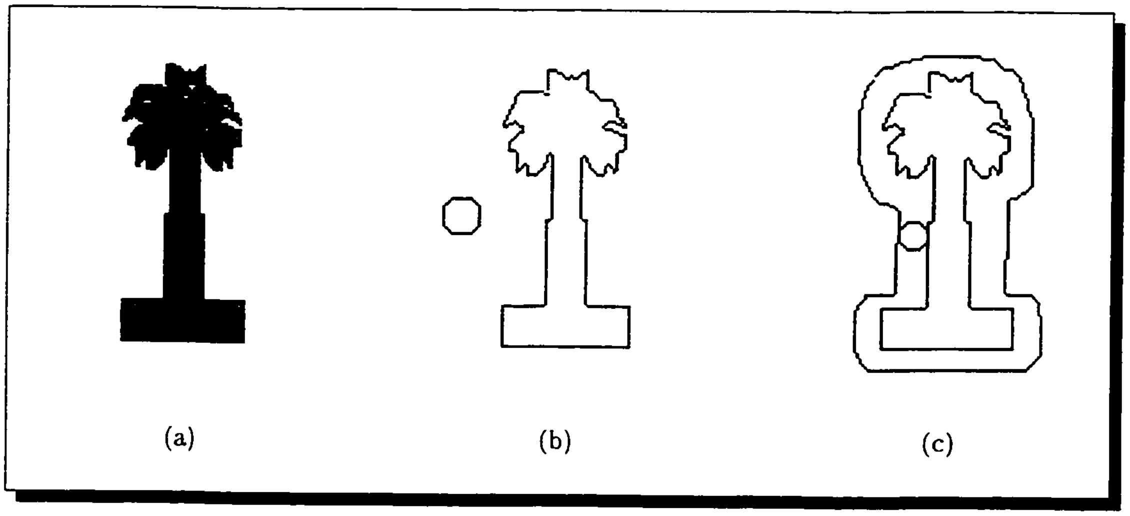

Figure 51: Input and output of Minkowski Sum computation

Animated Obstacles Another challenge for future work will be to be able to generate a path through moving obstacles. This will require a faster, custom made triangulation algorithm that will be specialized in generating meshes without triangulating every point of the contours but only points situated at certain distances from one another (in the same polygon .)

Appendix A

Source Code

Here is Dijkstra's shortest path algorithm implementation for this project:

```
void Contour::Dijkstra( Point& p_Source, Point& p_Target,
                                PointList& p_Points )
{
PointList   pl;
SegmentList sl;
PolygonList::iterator pit;
PointList::iterator   ptit;
SegmentList::iterator sit;
Point     NearPoint = p_Target; // first intersection with a Polygon
Polygon   *NearPoly = NULL;
Segment   *NearSeg = NULL;
float     NearDistance = p_Source.Distance(p_Target);
Point     FarPoint = p_Source;
Polygon   *FarPoly = NULL;
Segment   *FarSeg = NULL;
float     FarDistance = 0.0;
float     dist;

p_Points.clear();
if( m_Triangles.GetSegments().empty() )
{
   cout << " !!!! Triangulate first !!!! " << endl;
   return;
```

```
}
// find first the intersections with the polygons
Segment  s(p_Source, p_Target);

for( pit = m_Polygons.begin(), pit != m_Polygons.end(); ++pit )
{
    if( pit->Intersect( s, &pl, &sl ) )
    {
         sit = sl.begin();
         ptit = pl.begin();
         for( ; ptit != pl.end() && sit != sl.end(); ++ptit, ++sit )
         {
             dist = p_Source.Distance(*ptit) ;
             if( dist < NearDistance )
             {
                 NearDistance = dist;
                 NearPoint    = (*ptit);
                 NearPoly     = &(*pit);
                 NearSeg      = &(*sit);
             }
             if( dist > FarDistance )
             {
                 FarDistance = dist;
                 FarPoint    = (*ptit);
                 FarPoly     = &(*pit);
                 FarSeg      = &(*sit);
             }
         }
    }
}
if( !(NearPoly && FarPoly) )
{
    // There are no intersections with polygons
    p_Points.push_back( p_Source );
    p_Points.push_back( p_Target );
    return;
```

```
}
// Now Find the shortest path from NearSeg to FarSeg
Queue pq;
Queue::iterator pqit;
hash_map<Point, PQElement, HashFunc> PredMap;
hash_map<Point, PQElement, HashFunc>::iterator prit;
typedef hash_map<Point, PQElement, HashFunc>::value_type PredValue;
MapType::iterator mit;
MapType& map = m_Triangles.GetVertices();
Point p, v;
Point p1 = (*NearSeg)[0];
Point p2 = (*NearSeg)[1];
Point p3 = (*FarSeg)[0];
Point p4 = (*FarSeg)[1];
PQElement ele;
float du, dv;

for( mit = map.begin(); mit != map.end(); ++mit )
{
    p = (*mit).first ;
    if( p == p1 )
    {
        dv = NearPoint.Distance(p1);
        pq.push_back( PQElement( p, dv) );
        PredMap.insert(PredValue(p, PQElement(NearPoint, dv)) );
    }
    else if( p == p2 )
    {
        dv = NearPoint.Distance(p2);
        pq.push_back( PQElement( p, dv) );
        PredMap.insert(PredValue(p, PQElement(NearPoint, dv)) );
    }
    else
    {
        pq.push_back( PQElement( p ));
    }
```

```
}
pq.push_back( PQElement( FarPoint ) );
pqit = min_element( pq.begin(), pq.end() );
ele = *pqit;
while( !pq.empty() )
{
    pqit = min_element(pq.begin(), pq.end());
    ele = *pqit;
    pq.erase(pqit);
    du = ele.GetWeight();
    p  = ele.GetPoint();
    mit = map.find( p );
    if( mit != map.end() )
    {
        for( ptit = (*mit).second.begin();
              ptit != (*mit).second.end(); ++ptit )
        {
            v = *ptit;
            dv = du + p.Distance(v);
            prit = PredMap.find(v);
            if( prit != PredMap.end() )
            {
               // the node is already in PQ and has known weight
               if( dv < (*prit).second.GetWeight() )
               {
                   (*prit).second = PQElement(p, dv);
                    pqit = find( pq.begin(), pq.end(), PQElement(v) );
                    if( pqit != pq.end() )
                    {
                        *pqit = PQElement(v, dv);
                    }
               }
            }
            else
            {
                // insert new node into PQ
        PredMap.insert(PredValue(v, PQElement(p, dv)) );
        pqit = find( pq.begin(), pq.end(), PQElement(v) );
        if( pqit != pq.end() )
        {
            *pqit = PQElement(v, dv);
        }
    }
}
// add the target as well
if( p == p3 || p == p4 )
{
    dv = du + p.Distance(FarPoint);
    prit = PredMap.find(FarPoint);
    if( prit != PredMap.end() )
    {
       // the node is already in PQ and has known weight
       if( dv < (*prit).second.GetWeight() )
       {
          (*prit).second = PQElement(p, dv);
          pqit = find( pq.begin(), pq.end(), PQElement(FarPoint));
          if( pqit != pq.end() )
          {
            *pqit = PQElement(FarPoint, dv);
          }
       }
    }
    else
    {
        // insert new node into PQ
        PredMap.insert(PredValue(FarPoint, PQElement(p, dv)));
        pqit = find( pq.begin(), pq.end(), PQElement(FarPoint));
        if( pqit != pq.end() )
        {
            *pqit = PQElement(FarPoint, dv);
        }
    }
```

```
            }
        }
        else if( p == FarPoint )
        {
            // we got to the FarPoint, time to break.
            pq.clear();
            break;
        }
    }
    // Now create the path
    p_Points.push_front( p_Target );
    p_Points.push_front( FarPoint );
    prit = PredMap.find(FarPoint);
    p = (*prit).second.GetPoint();
    while( p != NearPoint )
    {
       p_Points.push_front( p );
       prit = PredMap.find(p);
       p = (*prit).second.GetPoint();
    }
    p_Points.push_front( NearPoint );
    p_Points.push_front( p_Source );
    return ;
    }
```

Bibliography

[CM96] Dave Cliff and Geoffrey Miller. Co-evolution of pursuit and evasion II: Simulation methods and results. In Pollack and Wilson, editors, *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.* ISBN 0-262-63178-4, MIT Press., 1996. http://www.cogs.susx.ac.uk/users/davec/pe.html.

[DeL00] Mark DeLoura. The basics of A^* for path planning. In *Game Programming Gems*, pages 254–263. Charles River Media, 2000.

[EW96] Parris Egbert and Scott Winkler. Collision-free object movement using vector fields. *IEEE Computer Graphics and Applications*, pages 18–24, 1996. http://www.computer.org/cga/cg1996/g4toc.htm.

[Isa65] Rufus Isaacs. Seek and pursuit. In *Differential Games: A Mathematical Theory with Application to Warfare and Pursuit, Control and Optimization.* John Wiley and Sons, New York, 1965.

[Kir92] David Kirk. Faster line segment intersection. In *Graphics Gems III*, pages 199–202. Academic Press, 1992.

[Lat91] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[LP83] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, pages 108–120, February 1983. Vol C-32, No. 2.

[ORo98] Joseph ORourke. Segment-Segment intersection. In *Computational Geometry in C (2nd Edition)*, pages 220–225. Cambridge University Press, 1998.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH '85 Proceedings, Computer Graphics*, pages 287–296, 1985. http://www.mrl.nyu.edu/perlin/doc/oscar.html.

[Rey87] Craig Reynolds. Flocks, Herds, and Schools: A distributed behavioral model. *SIGGRAPH '87 Conference Proceedings, Computer Graphics*, pages 25–34, 1987. http://www.red3d.com/cwr/boids/.

[Rey99] Craig Reynolds. Steering behaviors for autonomous characters. *Conference Proceedings of the Games Developer Conference*, pages 763–782, 1999. http://www.red3d.com/cwr/steer/.

[SGI03] Alexander Stepanov SGI. Standard Template Library. *http://www.sgi.com/tech/stl/*, 2003.

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[Sto99] Bryan Stout. Intelligent path-finding. *http://www.gamasutra.com/features/19990212/sm_01.htm*, 1999.

[THC94a] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. Depth-first search. In *Introduction to Algorithms*, pages 477–479. McGraw-Hill Book Company, 1994.

[THC94b] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. Shortest paths and relaxation. In *Introduction to Algorithms*, pages 518–532. McGraw-Hill Book Company, 1994.

[Tro02] Trolltech. Qt library. *http://doc.trolltech.com/3.0*, 2002.