# FreeSCI pathfinding extension

by

Walter van Niftrik
w.f.b.w.v.niftrik@stud.tue.nl
TU Eindhoven
Student ID: 427453

July 4, 2006

A report on an internship with the FreeSCI project.
January 30, 2006 - May 12, 2006

**Supervisors**

Bettina Speckmann
TU Eindhoven

Christoph Reichenbach
FreeSCI project

**Abstract**

FreeSCI is a free software project that aims to develop a complete reimplementation of the proprietary adventure game interpreter SCI. SCI provides a pathfinding function which allows games to move objects around obstacles. The methods that were used in SCI to implement this functionality have been patented. This is a report on how the semantics of the pathfinding function were determined and a non-infringing substitute was implemented in FreeSCI.

# Contents

**1 Introduction** **1**

**2 Patent** **2**
- 2.1 Patents and infringement . . . . . 2
- 2.2 Sierra's pathfinding patent . . . . . 3

**3 Semantics** **5**
- 3.1 Polygon types . . . . . 6
- 3.2 Keyboard handling . . . . . 8
- 3.3 Display border . . . . . 9
- 3.4 Containment test . . . . . 10
- 3.5 Vertex ordering . . . . . 10

**4 Specification** **10**

**5 Algorithms** **13**
- 5.1 Pathfinding . . . . . 14
- 5.2 Polygon containment . . . . . 14

**6 Implementation** **15**
- 6.1 Data types . . . . . 15
- 6.2 Basic geometry . . . . . 16
- 6.3 Pathfinding . . . . . 17
- 6.4 Rounding real numbers . . . . . 17

**7 Conclusions and recommendations for future work** **18**

# 1 Introduction

SCI[1] is a graphical adventure game interpreter. In SCI games the player assumes the role of the protagonist whom we call *Ego*. SCI games feature both male and female protagonists, but we treat Ego as male in this report.

SCI was developed by Sierra On-Line in the late 1980s and was used commercially for almost a decade. During that time it was extended to keep up with the demands of consumers. Games of the first generation, known as SCI0, are mostly parser-based, where the user has to type in basic commands such as 'open door'. The user controls Ego mainly with the directional keys on the

[1] An acronym for Sierra Creative Interpreter.

keyboard. Mouse support is available in some SCI0 games, but it is very basic. The user can click somewhere to make Ego walk to that position. However, he will simply attempt to traverse a direct trajectory. Should there be any object in his path he will bump into it and stop moving, rather than find his way around it.

Games of the second generation, SCI1, can be almost exclusively controlled with the mouse. Instead of typing in commands with the keyboard, the user can use the mouse to click on icons representing basic actions such as 'walk' and 'take'. The mouse is also the main method for controlling Ego. Contrary to SCI0, games use pathfinding to manoeuvre Ego to his destination while avoiding any obstacles.

The interpreter is an object-oriented stack-based virtual machine that executes SCI scripts. Scripts can communicate with the virtual machine through a software layer called the *kernel*. The kernel provides a variety of high-level functions for tasks such as animation, playing sound and parsing user input. A kernel function called *AvoidPath* implements the pathfinding functionality in SCI1.

FreeSCI is a free software project founded in 1999. The goal of the project is to create a reimplementation of SCI. FreeSCI supports SCI0 games pretty well, but SCI1 support still lacks some important functionality. One of those missing pieces was support for AvoidPath. The goals of this internship were to implement and document this function. There were two main complications. The first was that no documentation of AvoidPath was publicly available. We had to discover the semantics ourselves. This is a common problem when reimplementing proprietary software. The second complication was that Sierra patented their implementation. We had to make sure that our implementation would not infringe their patent.

In Chapter 2 we discuss Sierra's patent. We describe the semantics of AvoidPath and how we determined them in Chapter 3. In Chapter 4 we give the specification for our implementation and in Chapter 5 we describe the algorithms we use. We discuss implementation issues that we encountered in Chapter 6. Finally, we present our conclusions in Chapter 7.

# 2 Patent

In Section 2.1 we provide a brief discussion on patents and patent infringement. In Section 2.2 we summarize Sierra's pathfinding patent [13].

## 2.1 Patents and infringement

A patent gives its holder the right to exclude others from using his or her invention. In exchange for that right the patent holder must fully disclose the invention in the patent document. After the patent expires others may use the invention. A patent document consists of two parts, the specification and the claims.

The specification section describes the invention in detail using everyday language. The claim section defines exactly what is patented. It uses a legal jargon that is difficult to understand for non-patent professionals. Information

on how to interpret patent claims can be found in [5]. For some patents infringement can be avoided by providing extra functionality on top of what is patented. Whether or not this is the case depends on the exact wording of the claims. For Sierra's pathfinding patent this is not the case. This means that a substantially different algorithm is required to avoid infringement.

### 2.2 Sierra's pathfinding patent

The AvoidPath input consists of a start point and an end point and a set of constraining polygons [13]. The output is a path from start point to end point that does not violate any constraints. The patent states that polygons can be assumed not to be self-intersecting. The patent further implies that a point cannot be inside two or more polygons. From that we conclude that any two polygons can be assumed not to intersect. The patent describes three optimization levels. We label these from 0 (no optimization) to 2 (maximum optimization). The optimization levels determine how optimized (in terms of length) the returned path will be.

The patent mentions three types of polygons. The first type is the *barred access* polygon that cannot be entered. The second type is the *total access* polygon that cannot be entered unless the start or end point is inside the polygon. The third type is the *near-point access* polygon. For a point $p$ and a polygon $P$, the *near-point* is a point on the edges of $P$ at minimal distance from $p$. A path may start in a near-point access polygon, but it must exit the polygon through the near-point. Similarly, a path may end in such a polygon, but it must enter the polygon through the near-point. The patent does not accurately describe the semantics of the different polygon types so we had to determine them in another way. We discuss this in Section 3.1.

The patent describes AvoidPath as consisting of three phases, namely a preprocessing phase, a pathfinding phase and a post-processing phase. Even though the pre- and post-processing phases are briefly explained in the patent specification, they are not mentioned in the claims and are therefore not patented. During the pre-processing phase the input is transformed in such a way that the start and end points are not inside any polygon, and that all polygons in the set may not be entered. For example, if the start point is inside a total access polygon that polygon is removed from the polygon set. If the end point is inside a near-point access polygon the end point is moved to the near-point and the original end point will be appended to the previously computed path in the post-processing phase.

We now proceed with describing the pathfinding phase, limiting ourselves to the most important aspects. The basic idea behind the algorithm is to construct a path that follows the direct trajectory from start point to end point, except where the trajectory properly intersects a polygon (this is covered by claim 6 of the patent). If the direct trajectory properly intersects a polygon $P$, then there must be more than one intersection point. Let $p$ be the intersection point closest to the start point, and $q$ be the intersection point closest to the end point. The computed path reaches $p$ on the direct trajectory, then goes from $p$ to $q$ along the edges of the polygon and then resumes the direct trajectory. There are two ways to walk from $p$ to $q$; the algorithm takes the shortest one. Optimization level 0 returns such a path, as illustrated in Figure 1.

At optimization level 1 the algorithm tries to optimize this path. It does

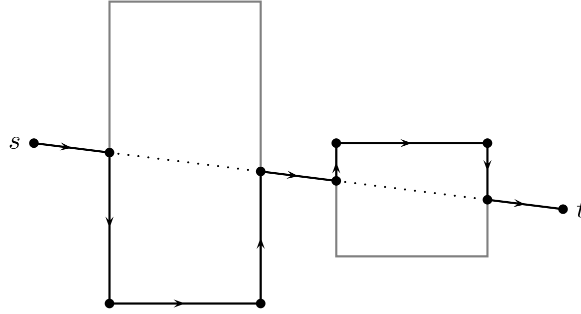


Figure 1: A path from $s$ to $t$ at optimization level 0.

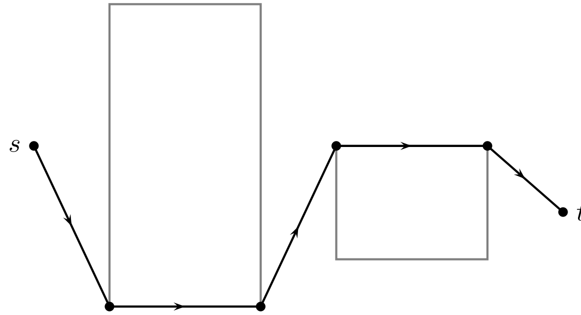


Figure 2: A path from $s$ to $t$ at optimization level 1.

this by attempting to shortcut any two nodes. A shortcut is only possible if it does not properly intersect any polygon in the polygon set. See Figure 2 for an illustration of an optimized version of the path in Figure 1.

The resulting path is not the shortest path from $s$ to $t$. The reason for this is that at optimization level 0 the shortest way about each individual polygon is chosen. However, it is very possible that the shortest path takes the longer way about one or more of the polygons. Optimization level 2 takes care of this by trying all permutations of ways to walk about the polygons. For $n$ polygons this yields $2^n$ combinations. The algorithm chooses the shortest path out of all of these. As can be seen in Figure 3, the shortest path is indeed found for this input.

This raises the question of whether or not the algorithm is able to determine the shortest path for *any* input. This is not the case, as is demonstrated by the counterexample in Figure 4. The shortcut from $v_0$ to $t$ cannot be taken as this would intersect polygon $W$. So the final path will go from $v_0$ to $t$ via $v_1$, even though it is much shorter to go there via $w_0$. Point $w_0$ is never considered for the path as it is part of a polygon that is not intersected by the direct trajectory

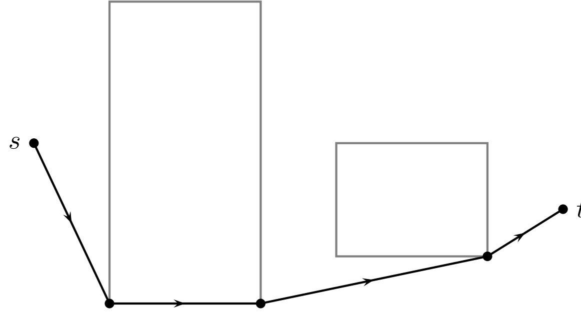


Figure 3: A path from $s$ to $t$ at optimization level 2.

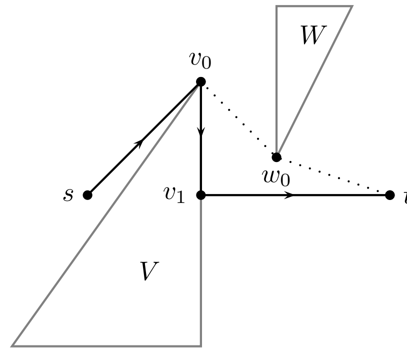


Figure 4: Polygons $V$ and $W$ and a path from $s$ to $t$ at optimization level 1 or 2.

from $s$ to $t$. This illustrates that this algorithm does not guarantee a shortest path, even at the highest optimization level.

The patent further makes an important note about polygon edges that lie along the display border. It is undesirable to have the path use the display border unnecessarily as this might trigger the game into loading an adjoining game scene. In order to prevent the shortest path algorithm from using polygon edges that lie along the display border, the distance for such edges is defined to be infinity.

## 3 Semantics

As the patent is not very precise when it comes to semantics, we performed a series of tests to get a more accurate picture of Sierra's implementation. The AvoidPath stub [10] mentions an additional containment test sub-function and

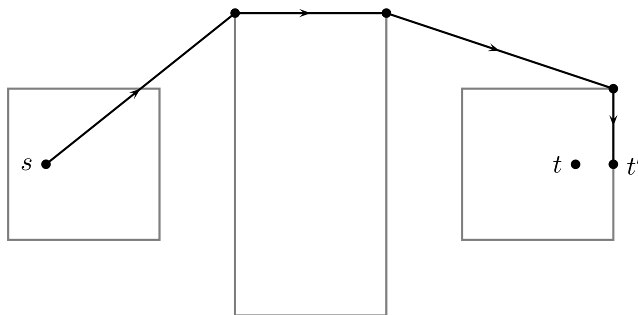


Figure 5: Three barred access polygons and a path computed for start point $s$ and end point $t$.

a fourth polygon type that the patent does not mention. We had to determine the semantics of those as well. It was also unclear how AvoidPath is used for the SCI0-style keyboard support still present in SCI1 games.

We performed most of the tests with the *experimentator* tool, developed by Christoph Reichenbach. This tool takes a polygon set description in text format, and transforms it into an SCI program that can be run both with Sierra SCI and with FreeSCI. The program draws the polygon set on the display and allows the user to select a start and destination point. It then calls AvoidPath and draws the returned path on the display. This has proven to be invaluable for discovering the semantics of AvoidPath. We used slightly modified versions of this tool for experimenting with the containment test sub-function and keyboard support.

We discuss the different polygon types in Section 3.1. Keyboard handling is detailed in Section 3.2 and the special treatment of the display border in Section 3.3. Section 3.4 describes the semantics of the containment test sub-function. Finally, we give the results of SCI vertex ordering tests in Section 3.5.

## 3.1 Polygon types

We performed a test with each of the three polygon types mentioned in the patent. The purpose was to see what the exact semantics are when the start or end point is inside such a polygon.

The purpose of barred access polygons is to prevent Ego from walking through solid objects. Figure 5 illustrates the semantics of this polygon type. The end point $t$ is inside a barred access polygon so $t$ cannot be reached. In that case the path goes to the near point $t'$, and stops there. The start point $s$ is also inside a barred access polygon. This is a little peculiar as that position is not reachable, but we had no justification for disallowing this setup as input. We see that in this case the polygon that contains $s$ is ignored. We observed different semantics in more recent SCI games, where instead of ignoring the polygon the path immediately leaves the polygon via the near point of $s$.

Figure 6 shows the semantics of total access polygons. As we can see, the polygons that contain $s$ and $t$ are ignored. In some game scenes a special event is triggered when Ego enters a particular area. Those areas usually have a total

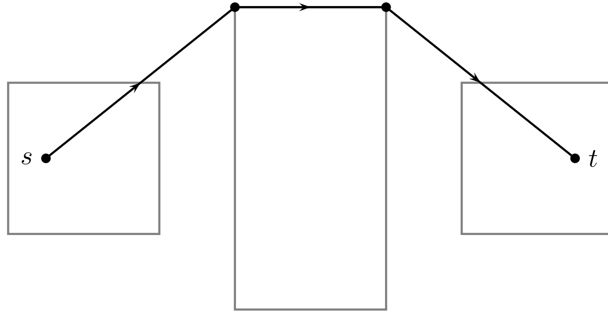


Figure 6: Three total access polygons and a path computed for start point $s$ and end point $t$.

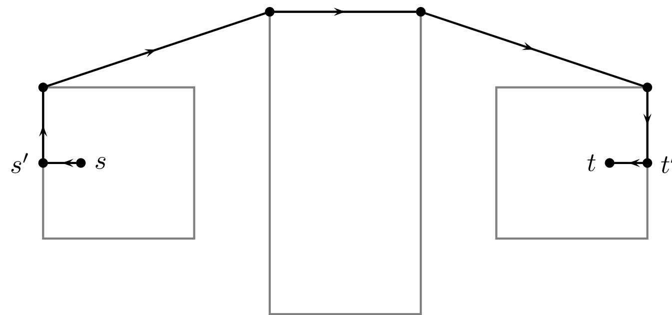


Figure 7: Three near-point access polygons and a path computed for start point $s$ and end point $t$.

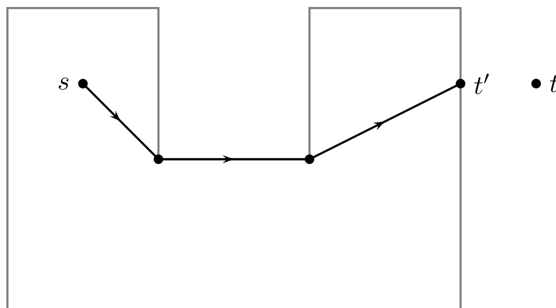


Figure 8: A contained access polygon and a path computed for start point $s$ and end point $t$.

access polygon around it. This prevents the pathfinding routine from directing Ego into such an area when the end point is not in that area.

Figure 7 illustrates the semantics of near-point access polygons. The path goes from $s$ to the near point $s'$ and ends at $t$ coming from the near point $t'$. We did not come across any near-point access polygons in games so the purpose behind these is unknown.

The fourth polygon type is the contained access polygon. Unfortunately we could not test its semantics with the experimentator. The experimentator only works with a particular version of Sierra SCI that predates the addition of this fourth polygon type. In order to solve this problem we added a debug mode to FreeSCI to visualize the polygons on the display, using a different colour for each polygon type. We found a contained access polygon this way and we observed its semantics in Sierra SCI. Figure 8 illustrates these semantics. As the end point $t$ is outside the polygon it cannot be reached. The path will end at the near point $t'$ instead. The idea of this polygon is that it defines the outer boundary and Ego must stay inside of it, i.e. no path may exit this polygon. As any two polygons cannot intersect it follows that there can only be one such polygon. With more than one contained access polygon no path exists that is fully contained in each of those polygons.

## 3.2 Keyboard handling

Even though the mouse is the main method for moving Ego in SCI1, the games still have keyboard support. This keyboard support exhibits the same behaviour as in SCI0. That is, after the user presses a directional key Ego will move in that direction until he bumps into an object, at which point he will stop. As the user has only indicated a direction, it is impossible to know to what exact location the user wants Ego to go. No attempts to avoid obstacles are made.

The games implement this by calling AvoidPath with the optimization level set to 0. The end point is chosen in such a way that it is outside of the display and exactly in the direction the user has indicated. For example, if Ego is positioned at coordinates (100, 100) and the user presses left; the end point might be (-10000, 100). Optimization level 0 is perfect for implementing this

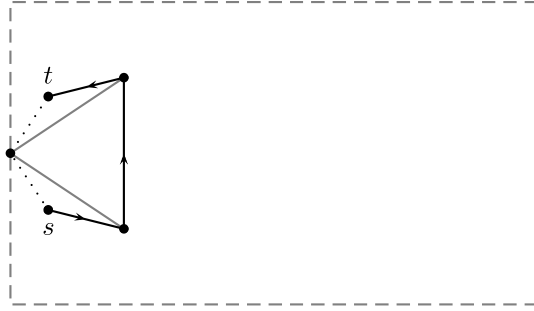


Figure 9: A barred access polygon that touches the display border and a path computed for start point $s$ and end point $t$.

functionality as the returned path will move along the direct trajectory from start point to end point until the first obstacle is encountered (see Figure 1). The games simply take the first two points of the returned path and throw away the rest.

Setting the optimization flag to 0 also causes AvoidPath to make two changes to the polygon set. The first change is that total access polygons are removed. The second is that near-point access polygons are turned into total access polygon. A likely reason for removing total access polygons is to prevent Ego from bumping into such a polygon as if it were an obstacle. Total access polygons are used to bar off special areas and prevent pathfinding from accidentally directing Ego into such an area. However, when the user directs Ego into such an area with the keyboard there is no reason to avoid the area.

As mentioned earlier, we never encountered a near-point access polygon in a game. Therefore we can only speculate as to the reason for turning near-point access polygons into total access polygon. A possible explanation might be that starting in a near-point access polygon could cause Ego to walk off into a different direction than the user had indicated. We have no plausible explanation for why they are turned into total access polygons, rather than being removed altogether.

## 3.3 Display border

As mentioned in Section 2.2, polygon edges along the display border should not be part of the path that is returned by AvoidPath. We determined that this also goes for vertices along the display border, as illustrated in Figure 9. The dotted line shows the path that would have been taken if the polygon were not touching the display border. We also determined that polygon edges along the display border are ignored when it comes to computing the near point. This is illustrated in Figure 10. $t'''$ is the near point in this case and not $t'$ or $t''$, even though both are closer to $t$ than $t'''$ is.

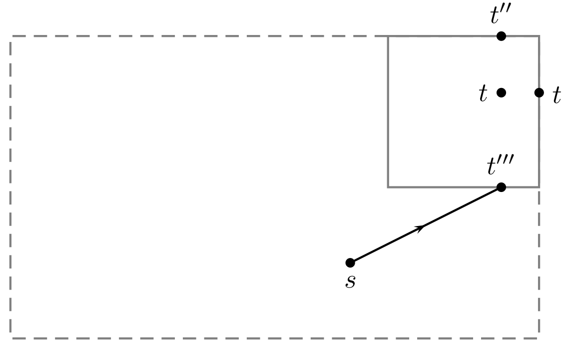


Figure 10: A barred access polygon that touches the display border and a path computed for start point $s$ and end point $t$.

### 3.4 Containment test

The containment test takes as input a point $p$ and a polygon $P$ and determines if $p$ is contained in $P$. This is obviously the case if $p$ lies on the inside of $P$. However, it was unclear if points that lie exactly on an edge of $P$ are considered to be inside or outside of $P$. With a modified version of the experimentator we determined that the former is the case.

### 3.5 Vertex ordering

Polygons are usually defined by a closed path of vertices. It is useful to use a fixed ordering of the vertices (either clockwise or anticlockwise). This has the advantage that the inside of a polygon is always on the same side of the directed edges of that polygon and thereby simplifies the implementation of the pathfinding algorithm. As the patent explicitly discusses the benefits of a fixed vertex ordering, it seemed likely that SCI games use it as well. We ran tests with SCI games in order to confirm this, but we determined that clockwise and anticlockwise ordering are used indiscriminately.

## 4 Specification

In this section we give a formal specification of the AvoidPath kernel function. This function computes a path from a start point to an end point, avoiding a set of polygons. We start out by formalizing the main notions from Section 2.2. A *point* $v$ is a position in the plane. Let $x(v)$ denote the x–coordinate of $v$ and $y(v)$ the y–coordinate of $v$. A *path* is a list of points $(v_0, ..., v_n)$ and a *polygon* is a closed path. We now define the near point:

**Definition 1** *Let $P$ be a polygon and $p$ be a point. Let $E$ be the set of edges of $P$ that do not lie along the display border. The near point of $p$ is a point at minimal distance from $p$ that lies on an edge in $E$.*

There are four different types of polygons, as described in Section 3.1. We now formalize the constraints the polygon types impose on the path that is to

be computed. We start with the barred access polygon. If the start point is contained in the polygon we use the semantics used by later versions of Sierra SCI, i.e. the path must leave the polygon immediately via the near point. This way the traversal through the polygon is minimal.

**Definition 2** *Let $P$ be a barred access polygon. If the start point $s$ is contained in $P$ the path must immediately exit the polygon via the near point of $s$. Except for that edge, the path may not intersect the inside of $P$. If the end point $t$ is contained in $P$ the path must end at the near point of $t$, instead of $t$.*

For the total and near-point access polygons the definitions are straightforward.

**Definition 3** *Let $P$ be a total access polygon. The path may not intersect the inside of $P$ unless the start or end point is contained in $P$.*

**Definition 4** *Let $P$ be a near-point access polygon. If the start point $s$ is contained in $P$ the path must immediately exit the polygon via the near point of $s$. Similarly, if the end point $t$ is contained in $P$ the path must reach $t$ via the near point of $t$. Except for those edges, the path may not intersect the inside of $P$.*

It was not possible to test the contained access polygon type with the experimentator, so the exact semantics are unknown. However, this polygon is intuitively similar to the barred access polygon, except that instead of the inside of the polygon it is the outside that cannot be entered. We assume that the semantics of contained access polygon are analogous to those of the barred access polygon.

**Definition 5** *Let $P$ be a contained access polygon. If the start point $s$ is not contained in $P$ the path must immediately enter the polygon via the near point of $s$. Except for that edge, the path may not intersect the outside of $P$. If the end point $t$ is not contained in $P$ the path must end at the near point of $t$, instead of $t$.*

With the addition of the contained access polygon it is no longer the case that the inside of the polygon constitutes the obstacle. We therefore define the *barred area* of a polygon as follows:

**Definition 6** *Let $P$ be a polygon. If $P$ is a contained access polygon then the barred area of $P$ consists of the edges of $P$ and the area outside of $P$. If $P$ is not a contained access polygon then the barred area of $P$ consists of the edges of $P$ and the area inside of $P$.*

We proceed by formalizing the two assumptions from Section 3.1.

**Assumption 1** *All polygons in the polygon set are not self-intersecting.*

**Assumption 2** *Let $S$ be the polygon set. For all points $p$ there is at most one polygon $P$ in $S$ for which $p$ is contained in the barred area of $P$.*

We now specify the sub-functions of AvoidPath. The first sub-function is CONTAINED.

Figure 11: Path from $s$ to $t$ as computed by KeyboardPath.

**Specification 1** Contained($v, P$)
*Input:*

- *A point* $v$*,*
- *A polygon* $P$*.*

*Output:*
***true*** *when* $v$ *is contained in* $P$*,* ***false*** *otherwise.*

The second sub-function is ShortestPath. In order to get the desired display border semantics from Section 3.3, we define the distance from point $s$ to point $t$ to be infinity if $t$ lies along the display border. We make an exception for the case where $t$ is the end point, as the border should be reachable if the user specifically directs Ego to the display border.

**Definition 7** *Let* $s$ *and* $t$ *be points. If* $t$ *does not lie along the display border and* $t$ *is not the end point then the distance from* $s$ *to* $t$ *is the euclidean distance between* $s$ *and* $t$*. Otherwise the distance is* $\infty$*.*

The specification of ShortestPath is as follows:

**Specification 2** ShortestPath($s, t, S$)
*Input:*

- *A start point* $s$*,*
- *An end point* $t$*,*
- *A polygon set* $S$*.*

*Output:*
*A shortest path from* $s$ *to* $t$ *that does not violate the constraints imposed by any polygon in* $S$*.*

The third sub-function KeyboardPath computes a path for keyboard support. This is not a true sub-function of AvoidPath, but we treat it as such to simplify our presentation. The semantics are described in Section 3.2. As computing the entire path as in Figure 1 would be in conflict with claim 6 of the patent, we are forced to change the semantics. Instead of using the patented method, we only compute the first edge of this path and use ShortestPath for the remainder of the path. As the games only use the first edge this works fine in practice. This first edge ends in what we call the *blocking point*.

It is tempting to define the blocking point as the first intersection that is encountered when travelling the direct trajectory from start point to end point. However, this would not work correctly as the polygon edges are valid positions for Ego. If Ego tries to move away from a polygon edge the first intersection will be at the start point. With that intersection as the blocking point Ego would not move. We solve this issue by defining the blocking point as follows:

**Definition 8** *Let $S$ be the polygon set. Let $s$ be the start point and $t$ be the end point. Let $I$ be the set of intersection points of the line $(s, t)$ and the polygons in $S$. The blocking point of $(s, t)$ wrt $S$ is the point $p \in I$ closest to $s$ for which the line $(t, p)$ intersects the inside of the polygon at $p$.*

Figure 11 shows an example of a path computed by this sub-function. The blocking point $s'$ is determined and then ShortestPath is used to compute the portion of the path from $s'$ to $t$. This leads to the following specification:

**Specification 3** KeyboardPath$(s, t, S)$
*Input:*

- *A start point $s$,*
- *An end point $t$,*
- *A polygon set $S$.*

*Let $S'$ be $S$ with all total access polygons removed and all near-point access polygons changed into total access polygons.*

*Output when no blocking point exists:*
*The path $(s, t)$*

*Output when there exists a blocking point $s'$:*
*The path concat*$((s)$, ShortestPath$(s', t, S'))$

# 5 Algorithms

In this section we discuss the algorithms that were considered and motivate our final choice. We describe the pathfinding algorithms in Section 5.1 and the polygon containment algorithms in Section 5.2.

### 5.1 Pathfinding

We learnt by observation that the input size generally does not exceed 100 nodes. At these small input sizes the asymptotic performance of an algorithm is not very significant. We decided that we did not want to sacrifice ease of implementation for an algorithm with better asymptotic behaviour. We also decided to limit ourselves to true shortest path solutions, as we want Ego to take the shortest path to his destination.

An optimal $O(n \log n)$ shortest path algorithm was discovered by Hershberger and Suri [8], but it is too complex to implement in such a short time frame. Simpler algorithms exist that are based on visibility graphs. For a polygon set $P$ a visibility graph $G$ contains all vertices of the polygons in $P$. The start and end point are added to $G$. $G$ has an edge between two vertices $v$ and $w$ iff the line $(v, w)$ does not intersect the interior of any polygon in $P$. Once the visibility graph has been constructed the shortest path can be found by using an algorithm for determining the shortest path in a graph, such as Dijkstra's algorithm. The visibility graph on $n$ vertices contains $n^2$ edges in the worst case. This means that any shortest path algorithm based on visibility graphs will have at least $O(n^2)$ running time. The trivial algorithm determines the visibility of two vertices $v$ and $w$ by testing for an intersection between the line $(v, w)$ and all edges in the input set, leading to $O(n^3)$ running time.

Optimal $O(n^2)$ visibility graph algorithms based on arrangements have been found [1, 12]. Ghosh and Mount discovered an output sensitive $O(E + n \log n)$ algorithm [7], where $E$ is the number of edges in the visibility graph. We finally opted to use an $O(n^2 \log n)$ algorithm by Lee. This algorithm is relatively easy to implement and is fast enough for the small input sizes we are dealing with in this application.

The algorithm iterates over the vertices and determines for each vertex $v$ the set of vertices that is visible from $v$. The basic idea is to rotate a sweeping line clockwise around $v$. All edges that are currently intersected by the sweeping line are maintained in a tree $T$ which is updated as the line rotates. The edges in $T$ are ordered by distance from $v$. The sweeping line is implemented by sorting the vertices by angle around $v$ and then processing them in that order. For each vertex $w$ in the sorted list the visibility is computed and $T$ is updated by removing edges incident to $w$ in anticlockwise direction, and adding edges incident to $w$ in clockwise direction. Vertex $w$ is visible iff the following two conditions hold. The first condition is that the line $(v, w)$ does not intersect the inside of the polygon of which $w$ is a vertex, locally at $w$. The second is that the edge in $T$ closest to $v$ does not properly intersect the line $(v, w)$. There are a few degenerate cases which we will not discuss here. A more in-depth discussion of this algorithm can be found in [3].

### 5.2 Polygon containment

For a point $p$ and a possibly non-convex polygon $P$ we need to be able to determine whether $p$ is contained in $P$. One common method for solving this problem is the winding number approach. The winding number of a point $p$ and a polygon $P$ is the number of times $P$ winds around $p$. For simple polygons this number is 1 when $p$ is contained in $P$ and 0 when $p$ is not contained in $P$. We decided not to use this method based on a claim in [9] that it is very inefficient.

However, we recently found claims to the contrary [11]. In hindsight dismissing this method was premature.

The method we used is called the ray crossings approach. This method is based on counting the number of times a ray extending right from $p$ intersects the boundary of $P$. $p$ is contained in $P$ iff the number of ray crossings is odd. In its basic form this algorithm considers points on bottom and left edges to be contained in $P$, and points on top and right edges not to be contained in $P$. This property is desirable for some applications, but as we have seen in Section 3.4, we need a point on any edge of $P$ to be considered to be contained in $P$. In order to overcome this we used an extended version of this algorithm [9] that has the semantics we require.

# 6 Implementation

In this chapter we discuss the more important aspects of the implementation. In Section 6.1 we discuss the data types we used. Section 6.2 has some details on basic geometry functions and how they were used in the implementation. Some details on the implementation of the path-finding functionality can be found in Section 6.3. And Section 6.4 discusses the issue of rounding real numbers to integers.

This report does not contain the actual program text that we produced. The program text is spread across the following four files that can be found on-line:

- http://svn.a-eskwadraat.nl/wsvn/FreeSCI/freesci/branches/glutton/src/engine/kpathing.c?op=file&rev=1571&sc=0
- http://svn.a-eskwadraat.nl/wsvn/FreeSCI/freesci/branches/glutton/src/scicore/aatree.c?op=file&rev=1571&sc=0
- http://svn.a-eskwadraat.nl/wsvn/FreeSCI/freesci/branches/glutton/src/include/aatree.h?op=file&rev=1571&sc=0
- http://svn.a-eskwadraat.nl/wsvn/FreeSCI/freesci/branches/glutton/src/include/list.h?op=file&rev=1571&sc=0

## 6.1 Data types

As the input and output consists of all integers we decided to avoid floating point arithmetic whenever possible. The main reason is that floating point arithmetic can be very slow on hardware without a floating point unit, as the computations must be handled in software in that case. Furthermore, using integers guarantees that the computational results are exact, contrary to floating point arithmetic. A potential pitfall here is overflow. However, in our case this is not a big problem as FreeSCI is written for machines with an integer size of at least 32 bits and the integer size in SCI is only 16 bits.

The two main choices for the polygon data type are an array of vertices, or a circular list of vertices. Arrays are convenient when vertices of a polygon are processed sequentially, but this is not the case here as the visibility graph algorithm processes the vertices by angle. Therefore we chose lists instead of arrays. As C does not provide any built-in list data types we took and modified the list macros from FreeBSD's queue.h header file [6]. We decided to order the

vertices in such a way that the barred area is always left of an edge. That means that barred, total and near-point access polygons use anticlockwise ordering, and contained access polygons use clockwise ordering. As we have seen in Section 3.5, SCI games do not have such a fixed vertex ordering, so we added code to detect the current vertex ordering, and reverse it if necessary.

For the balanced search tree required for the visibility graph algorithm we implemented an Andersson tree [2]. Andersson trees are easier to implement than other BB-trees yet offer similar performance. For storing the visibility graph itself we decided to use an adjacency matrix.

## 6.2 Basic geometry

We implemented basic geometry functions such as those in [9]. Some examples of those functions are:

- LEFT, which takes three points $p_0$, $p_1$ and $p_2$ and returns **true** iff $p_2$ lies left of the directed line $(p_0, p_1)$.
- COLLINEAR, which takes three points $p_0$, $p_1$ and $p_2$ and returns **true** iff $p_0$, $p_1$ and $p_2$ are collinear.

These functions are the foundation for implementing some of the more complex functions. We give several examples in the remainder of this section.

Given two edges $(p_0, p_1)$ and $(p_1, p_2)$ we use LEFT to detect the convexity of the vertex at $p_1$. Namely, the vertex at $p_1$ is convex iff $\text{LEFT}(p_0, p_1, p_2)$. This relies on a fixed vertex ordering as described in Section 6.1.

As we described in Chapter 4, we need to compute the blocking point for keyboard support. The blocking point is the first intersection point $p$ that is encountered when travelling the direct trajectory from start point $s$ to end point $t$ that has the property that the line $(t, p)$ intersects the inside of the polygon that contains $p$, locally at $p$. We refer to this property as $\text{INSIDE}(p, t)$. We make a case distinction on whether or not $p$ lies exactly on a vertex. If $p$ does not lie on a vertex it lies on exactly one edge $(p_0, p_1)$. Because of the vertex ordering we know that the inside of the polygon is left of that edge. This implies that $\text{INSIDE}(p, t)$ iff $\text{LEFT}(p_0, p_1, t)$. If $p$ lies on a vertex there are two neighbouring vertices at $p_0$ and $p_1$. We now need to make another case distinction on the convexity of the vertex at $p$ (using LEFT as demonstrated earlier in this section). If the vertex at $p$ is convex that implies that $\text{INSIDE}(p, t)$ iff $\text{LEFT}(p_0, p, t)$ and $\text{LEFT}(p, p_1, t)$. If the vertex at $p$ is not convex that implies that $\text{INSIDE}(p, t)$ iff $\text{LEFT}(p_0, p, t)$ or $\text{LEFT}(p, p_1, t)$.

One of the steps in the visibility graph algorithm is the ordering by angle of vertices around a specific point $p$. The angle is 0 for points on the ray extending right from $p$, and increases clockwise. Let $\alpha(q, p)$ denote the angle (in degrees) of point $q$ relative to point $p$. We used the QSORT POSIX function to do the ordering. This function calls a callback function for comparing two elements (in our case points). The arctangent function could be used here to compute the two angles, but this function is generally very slow. We chose a different solution that is outlined below.

We partition the plane around $p$ into two parts (we ignore the trivial partition $\{p\}$ for which no angle can be determined). Partition $A = \{(x, y) \in \mathbb{Z}^2 | y < y(p) \vee (y = y(p) \wedge x > x(p))\}$, being the set of all points $q$ for which $0 \leq$

$\alpha(q,p) < 180$. Partition $B = \{(x,y) \in \mathbb{Z}^2 | y > y(p) \vee (y = y(p) \wedge x < x(p))\}$, being the set of all points $q$ for which $180 \leq \alpha(q,p) < 360$.

If the two points $p_0$ and $p_1$ that are to be compared are not in the same partition the one in partition $A$ is trivially at a smaller angle. Otherwise, $p_0$ and $p_1$ are in the same partition. Two things have been accomplished by partitioning the plane in this way. The first is that COLLINEAR$(p, p_0, p_1)$ holds iff $\alpha(p_0, p) = \alpha(p_1, p)$ (distance from $p$ is the comparison criteria in this case). The second is that LEFT$(p, p_0, p_1)$ holds iff $\alpha(p_1, p) < \alpha(p_0, p)$. We implemented the sorting by angle using LEFT and COLLINEAR in this manner, thus avoiding the use of expensive trigonometry functions.

## 6.3 Pathfinding

The first step of SHORTESTPATH is to check for both the start and end point whether it is contained in the barred area of any polygon in the polygon set $P$. If necessary, we then deal with this situation according to the specification in Chapter 4. If a start or end point $p$ is contained in a total access polygon we remove that polygon from $P$. For other polygon types we compute the near point of $p$ and use that for pathfinding instead. We add the original point $p$ to the path during the post-processing phase. The only exception is the case where the end point is contained in the barred area of a barred or contained access polygon. In that case the path ends at the near point of $p$, rather than $p$.

We find the near point of $p$ by first computing, for each edge $e$ of $P$, the point on $e$ that is closest to $p$. This is straightforward [4]. If $P$ has $n$ vertices this gives us $n$ candidates. Out of all the candidates at minimal distance from $p$ we randomly pick one as the near point.

After the pre-processing step the start and end points are not strictly contained in the barred area of any of the polygons. However, they may lie on an edge. Before we execute the visibility graph algorithm, we merge them into the polygon set, generally as a polygon of a single vertex. As the algorithm does not support intersecting polygons, we must be careful when merging a point $p$ that lies on an edge of one of the polygons in the polygon set. In order to solve this problem we first check if a vertex is already present at $p$, if that is the case then we do nothing. If there is no matching vertex we check if there is an edge $e = (p_0, p_1)$ that contains $p$, if there is then we split $e$ into two edges $(p_0, p)$ and $(p, p_1)$. This ensures that no intersecting polygons are created. After executing the visibility graph algorithm we use Dijkstra's algorithm to find a path in the visibility graph.

## 6.4 Rounding real numbers

Even though we use integers almost everywhere in the implementation we cannot completely avoid the use of real numbers. The two main instances where we use real numbers are when computing intersections and near points. We round the real numbers to integers right away. However, we must take care not to round to a point that is in the barred area of a polygon, because that would make that point unreachable. First we strengthen Assumption 2 as follows:

**Assumption 3** *Let $P$ be the polygon set. For all points $p \in \mathbb{Q}^2$ that lie on an*

*edge of one of the polygons in $P$, there is at least one point $p' \in \mathbb{Z}^2$ at distance $< \sqrt{2}$ from $p$ that is not strictly contained in the barred area of any polygon in $P$.*

This means that besides intersecting polygons, we also exclude polygons that are very close together. It would be very hard to manoeuvre Ego between two such polygons with the directional keys on the keyboard. We therefore believe that the input polygon set will not contain polygons that are very close together. When rounding a point $p \in \mathbb{Q}^2$ we first try the closest point $p' \in \mathbb{Z}^2$. If $p'$ is contained in the barred area of a polygon we try all other points in $\mathbb{Z}^2$ at a distance less than $\sqrt{2}$ from $p$ until we find one for which this is not the case.

# 7 Conclusions and recommendations for future work

The main goal of this internship was to implement the AvoidPath function without infringing Sierra's pathfinding patent. We accomplished this by choosing an algorithm that is very different from the one described in the patent. The algorithm we chose has one advantage over the algorithm used by Sierra, namely that it always computes the shortest path. Our implementation performed well in the basic tests we performed. However, at this time we are aware of two limitations.

The first limitation is related to the rounding of intersecting points as in Section 6.4. Our implementation tries nearby points in $\mathbb{Z}^2$. If an intersected polygon is very thin, a rounded point may end up on the wrong side of the polygon. This would make it possible to walk through very thin polygons with the keyboard. We used the experimentator to test very thin polygons in Sierra SCI and we observed that it suffers from the same limitation. A possible solution for this problem would be to ignore rounded points that lie left of the intersected edge.

The second limitation is related to Assumption 2. In practice there are game scenes that contain intersecting polygons, and thus violate this assumption. As we currently know of only one scene for which this is the case[2], this issue was ignored for the present project. This scene works correctly in Sierra SCI, even though tests with the experimentator showed that Sierra SCI does not support intersecting polygons in general. If there are more game scenes that violate the assumption, it might be necessary to add support for intersecting polygons. This would create an interesting problem as the visibility graph algorithm we used is fundamentally incompatible with intersecting polygons.

The algorithm stores the edges that the sweeping line intersects in a tree, ordered by distance. Let $e_0$ and $e_1$ be edges. If $e_0$ and $e_1$ do not intersect then their ordering does not depend on the position of the sweeping line, but if they intersect at a point $p$ this ordering flips when the sweeping line hits $p$. It may be possible to store intersection points and initiate a tree reordering when the sweeping line hits an intersection point. However, it might be better to detect intersecting polygons and fall back to the trivial $O(n^3)$ algorithm, which does not make use of such an ordered data structure. Other problems arise when

[2]Conquests of the Longbow, room 210.

a vertex $v$ of one polygon lies on an edge $e$ of another. In that case it would be possible to create a path that goes through $e$ via $v$, as $e$ does not properly intersect any edges incident to $v$ and thus would not obstruct visibility.

It may be possible to add support for intersecting polygons by first merging the intersecting polygons into one polygon. In some cases this does not work, however. Let $P$ and $Q$ be two polygons whose intersection is a single point $p$. If we now merge $P$ and $Q$ we get a polygon that is self-intersecting at $p$, which would still be illegal input for the visibility graph algorithm as it violates Assumption 1.

All of this suggests that there is no quick fix that adds support for intersecting polygons. If it turns out that support for intersecting polygons is necessary, further research will need to be performed.

# References


[1] T. Asano, T. Asano, L. J. Guibas, J. Hershberger and H. Imai. *Visibility of disjoint polygons*, Algorithmica, Vol. 1, pp 49-63, 1986.

[2] A. Andersson. *Balanced Search Trees Made Simple*, Third Workshop on Algorithms and Data Structures, pp 60-71, Springer-Verlag, 1993.

[3] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf. *Computational geometry: Algorithms and applications*, Section 15.2, Springer-Verlag, Berlin, 1997.

[4] P. Bourke. *Minimum Distance between a Point and a Line*, http://astronomy.swin.edu.au/~pbourke/geometry/pointline/, 1988.

[5] H. M. Eisenberg. *Patent law you can use. Reading a Patent - Part II The Description and the Claims*, http://www.yale.edu/ocr/invent_guidelines/docs/reading_patent2_claims.pdf, 1999.

[6] The FreeBSD Project, FreeBSD source code file queue.h, revision 1.66, http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/sys/queue.h?rev=1.66&content-type=text/plain, 2006.

[7] S. K. Ghosh and D. M. Mount. *An output-sensitive algorithm for computing visibility graphs*, SIAM Journal on Computing, Vol. 20, No. 5, pp. 888-910, October 1991.

[8] J. Hershberger and S Suri. *An Optimal Algorithm for Euclidean Shortest Paths in the Plane*, SIAM Journal on Computing, Vol. 28 , No. 6, pp. 2215-2256, December 1999.

[9] J. O'Rourke. *Computational geometry in C*, second edition, Cambridge University Press, Cambridge, 1998.

[10] L. Skovlund, C. Reichenbach. FreeSCI source code file kpathing.c, revision 1505, http://svn.a-eskwadraat.nl/wsvn/FreeSCI/freesci/branches/glutton/src/engine/kpathing.c?op=file&rev=1505&sc=0, 2002.

[11] B. Sunday. *Fast Winding Number Inclusion of a Point in a Polygon*, http://www.geometryalgorithms.com/Archive/algorithm_0103/algorithm_0103.htm, 2001.

[12] E. Welzl. *Constructing the visibility graph for n line segments in* $O(n^2)$ *time*, Information Processing Letters, pp 167-171, 1985.

[13] K. A. Williams, D. C. Iden and L. L. Scott. *System and methods for intelligent movement on computer displays*, United States Patent No. 5287446, 1990.