Kapitel 1

Lösungsidee

Die erste Aufgabe, das Einlesen und Darstellen des Museums ist simpel, es müssen bloß die Punkte in den einzelnen Zeilen eingelesen und den Polygonen zugewiesen werden. Dann muss man nur noch die Punkte auf dem Bildschirm ausgeben und miteinander verbinden.

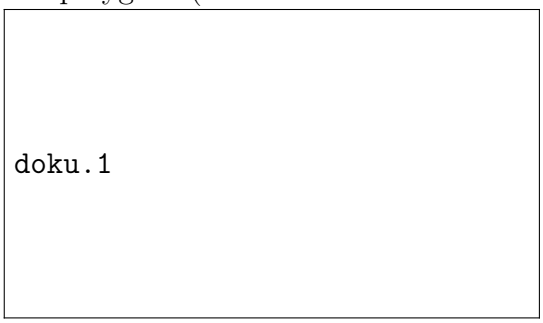
Bei der zweiten Aufgabe ist das größte Problem eine sinnvolle Modellierung für den Weg des Nachtwächters zu entwerfen. Ich habe mich dafür entschieden, diesen Weg, wie das Museum selbst, durch ein Polygon darzustellen, dessen Rand den Weg repräsentiert. Dargestellt und gespeichert kann der Weg dadurch in derselben Weise werden, wie das Museum.
Eine solche Form hat die Vorteile, dass sie sehr einfach zu speichern, auszugeben und zu verarbeiten ist, vor allem wenn die nötigen Methoden bereits für das Museum geschrieben worden sind.
Die Nachteile sind, dass ein Polygon den Weg wahrscheinlich nicht exakt wiedergibt, da Menschen nicht dazu neigen, zuerst eine Richtung beizubehalten und sie dann überraschend zu wechseln, sondern stattdessen in einem Bogen wenden. Eine solcher Weg mit glatten Richtungswechsel wäre realistischer durch Bezierkurven oder ähnliches simuliert. Dies würde allerdings die Berechnung der sichtbaren Bereiche unnötig verkomplizieren, da man die runden Wege ja auch durch das Hinzufügen von Ecken zu einem Polygon approximieren kann.
Ein weiteres Problem ist, dass der Nachtwächter nicht immer exakt denselben Weg laufen wird, und daher möglicherweise in manchen Rundgängen Berei-

che sehen kann, die er in anderen nicht sieht. Dafür ist es am sinnvollsten das Museum nicht nur in sichtbare und unsichtbare Bereiche aufzuteilen, sondern in Bereiche mit unterschiedlichen Sichtbarkeitswahrscheinlichkeiten. Ich habe mich aber zunächst dafür entschieden, ein präzises Ergebnis zu berechnen, bei dem der Nachtwächter wie ein Roboter (vielleicht ist es ja auch einer) nur auf einem exakt definierten Weg läuft. Später (das heißt am 23. April) habe ich allerdings beschlossen, probeweise einen Algorithmus mit Wahrscheinlichkeiten zu implementieren, den ich am Schluss dieses Kapitels beschreibe. Für diesen Algorithmus verwende ich als Dateiformat für den Rundgang kein Polygon, sondern ein Bitmap, in dem die Farben unterschiedliche Aufenthaltswahrscheinlichkeiten für deb Wächter angegeben. Aber das kommt alles später.

Der eigentliche Algorithmus zur Berechnung der Sichtbarkeit läuft grob betrachtet, so ab, dass für jede Seite des Wegpolygons separat berechnet wird, welche Segmente der Seiten der anderen Polygone, der Museumspolygone, von dort aus gesehen werden können. Ein Segment wird dabei jeweils von zwei Punkten auf der jeweiligen Seite begrenzt, die von einem oder zwei Punkten auf der gewählten Seite des Wegpolygons gesehen werden können. Diese Punkte bilden dann jeweils ein Dreieck oder Viereck, dass komplett sichtbar ist.
Sind diese Berechnungen abgeschlossen, ist das gesamte sichtbare Museumspolygon in eine Reihe von (sich überschneidenden) Drei/Vierecken zerlegt, die man auf der Ausgabe des Polygons hervorheben kann. Man könnte sie nun auch mit Standardalgorithmen zu einem großen Polygon zusammenfassen, aber davon habe ich abgesehen, da es eigentlich unnötig ist, wenn man die Ausgabe nicht weiter verarbeiten, sondern nur ausgeben will.
Im folgenden beschreibe ich nun zuerst, wie die Berechnung der von einem Punkt aus sichtbaren Abschnitte abläuft, dann die Berechnung derselben von einer Linie aus und abschließend die Berechnung der Drei-/Vierecke aus den sichtbaren Liniensegmente.

1.1 Sichtbarkeit von einem Punkt aus

Man kann sich einen Strahl vorstellen, der von einem Punkt P im Museum ausgeht und der sich, wie das Licht eines Leuchtturms, langsam um diesen Punkt dreht. Während dieser Rotation trifft er immer eine Seite des Museumspolygons (dazu zähle ich auch die Seiten der „Löcher“):



Wenn diese Seiten nun lichtempfindlich sind, und sich beim Auftreffen des Strahles verfärben, entstehen nun farblich hervorgehobene Bereiche auf den Seiten. Diese hervorgehobenen Bereiche sind nun die zu berechnenden Liniensegmente. (Anmerkung: So kann man auch ohne Computer die sichtbaren Bereiche erkennen. Man streicht einfach das gesamte Museum mit einer lichtempfindlichen Farbe, gibt dem Nachtwächter eine Laterne und lässt ihn im Kreis laufen. Wenn es dann bunt genug geworden ist, braucht man auch keine moderne Kunst mehr aufzustellen. gnukremnA)
Die Erzeugung der Segmente durch einen wandernden Lichtstrahl erklärt auch, warum das Dreieck das entsteht, wenn man die Endpunkte eines Segmentes mit dem Punkt P verbindet, komplett sichtbar ist. Gäbe es Hindernisse innerhalb dieses Dreiecks, wäre der Strahl auf diese getroffen, und gar nicht erst bis zum Segment auf der Polygonseite gekommen.

Für den Algorithmus von Interesse sind aber nur die Endpunkte der Segmente, weshalb es unnötig ist, tatsächlich einen langsam rotierenden Strahl in den Algorithmus zu implementieren. Vielmehr reicht es einen Strahl vom Punkt P auf die Eckpunkte der Museumspolygone zu senden, da, wie man leicht einsieht, es nur dann einen Wechsel der getroffenen Seite geben kann, wenn sie entweder endet (in diesem Fall wird eine weiter entfernte Seite getroffen) oder wenn eine nähere Seite beginnt (dann wird diese getroffen). In jedem Fall findet der Wechsel genau an einem Endpunkt einer Seite statt.

Wenn man nun vom Punkt P aus einen Strahl zu einem anderen Eckpunkt sendet, können sowohl Punkte wie auch Seiten der Museumspolygone getroffen werden. Wird (ausschließlich) eine Seite getroffen, weiß man, dass der Strahl dort endet, da der Nachtwächter ja nicht durch Wände sehen kann. (tja, ohne Röntgenaugen ist man aufgeschmissen ;-). Allerdings ist dieser Fall unwichtig, da der Strahl die Seite wahrscheinlich innerhalb eines sichtbaren Segmentes trifft und nicht an einem Ende.
doku.2 `doku.2`

Komplizierter ist die Situation, wenn ein Eckpunkt getroffen wird. Dieser hält den Strahl nämlich nur dann auf, wenn die anderen Endpunkte beider an dem getroffenen Punkt endenden Wände auf unterschiedlichen Seiten des Strahles liegen (siehe Bild: Fall 1). Liegen diese beiden anderen Endpunkte dagegen auf derselben Seite, kann der Strahl seitlich (beliebig nah) an dem Punkt vorbei. (so etwas nenne ich „Zacke“) Dies bedeutet normalerweise, dass sowohl der eben getroffene Punkt, als auch ein später getroffener sichtbar sind (Fall 2). Es kann allerdings auch passieren, dass der Strahl auf zwei Seiten von solchen Ecken begrenzt wird, und der letzte getroffene Punkt nicht sichtbar ist. (Fall 3):
doku.3 `doku.3`

Falls bei der Berechnung der Treffpunkte, ein Treffpunkt gefunden wird, an dem der Strahl endet, muss man natürlich alle weiter entfernten Punkten ignorieren, da diese dann nicht mehr getroffen werden können.

Der letzte vom Strahl getroffene Punkt auf einer Seite markiert genau dann die Begrenzung eines sichtbaren Liniensegments, wenn der Strahl auf exakt einer Seite von einer Zacke begrenzt wird. Ist er nämlich auf keiner Seite begrenzt, wird nur ein beliebiger Punkt innerhalb eines sichtbares Segments getroffen und kein Randpunkt, wie man an dem Bild auf der vorherigen Seite sehen kann. Wird der Strahl auf beiden Seiten begrenzt, gibt es überhaupt kein sichtbares Segment, und genau genommen ist noch nicht einmal der Punkt selbst sichtbar (siehe Fall 3 und 2). Der andere Randpunkt eines am Rand getroffenen Liniensegmentes liegt hierbei natürlich auf der anderen Seite des Strahles als die begrenzende Zacke. (Intuitiv ist das völlig klar, aber für den Algorithmus ist es wichtig, in welche Richtung das Segment fortgesetzt werden muss).
Liegt der letzte getroffene Punkt nicht auf einer Seite, sondern ist vielmehr

der Endpunkt einer Seite (genauer gesagt zweier Seiten), kann er auch der Randpunkt zweier sichtbaren Segmente sein, dann nämlich wenn der Strahl nicht begrenzt wurde (siehe Fall 1), da jedes sichtbare Segment dort endet. Ist der Strahl auf zwei Seiten begrenzt, ist natürlich wieder nichts sichtbar. Ist er auf nur einer begrenzt, liegt das sichtbare Segment auf der Seite des Strahls auf der nicht die Zacke liegt (vergleiche Fall 2).
Der andere Randpunkt des Segmentes befindet sich immer in Richtung des anderen Endpunktes der Seite.

Die letzte Art der getroffenen Punkte, sind die Eckpunkte der Zacken. Hierbei muss man nur die Punkte der ersten Zacke (also die, die am nächsten zum Startpunkt des Strahles ist) auf jeder Seite betrachten, da darauf folgende Zacken von der ersten verdeckt werden.
Die Eckpunkte markieren immer den Randpunkt eines sichtbaren Segmentes auf einer der beiden, an den Punkt grenzenden Wände. Sichtbar ist dabei allerdings immer nur eine Wand, da die andere von dieser sichtbaren verdeckt wird. Zum Glück ist es aber gerade dadurch, dass die sichtbare Wand die andere verdeckt, relativ simpel herauszufinden, welche die sichtbare ist. Es ist immer die, bei der die andere Seite (bzw. deren Endpunkt) und der Startpunkt auf verschiedenen Seiten der Wand liegen, da durch das Liegen auf verschiedenen Seiten, der Blick auf die andere Wand blockiert wird. Das reicht aus, um die Randpunkte aller sichtbaren Segmente zu finden.
Wendet man das Verfahren beispielhaft auf die oben gezeigten Polygone an, ergibt sich wie gewünscht:
doku.4 `doku.4`

1.2 Sichtbarkeit von einer Linie aus

Nun folgt die Berechnung der, von einer Linie aus sichtbaren Bereiche der Wände. Im Prinzip funktioniert diese ähnlich, wie die Berechnung der von den Punkten aus sichtbaren Liniensegmente.
Der erste Schritt besteht darin, die Bereiche, die man von den Eckpunkten der Linie aus sehen kann, zu berechnen, da die Endpunkte natürlich zu der Linie gehören.
Als nächstes müssen die Liniensegmente der Museumswände gesucht werden, die zwar von der Linie, nicht aber von ihren Endpunkten aus, gesehen werden

können. Der Algorithmus hierfür basiert darauf, das man im Prinzip alles, was man von der Linie aus, auch von einem der Endpunkte sehen kann, es sei denn, der Blick vom Endpunkt aus wird durch die Seite eines anderen Polygons blockiert.
Sieht man an dieser Linie vorbei, um den verdeckten Bereich doch noch zu erblicken, ist es natürlich am besten möglichst nahe an der Seite vorbei zu sehen, um einen möglichst großen Bereich abzudecken. Das heißt also so, dass ein Strahl von der Linie zum äußersten sichtbaren Punkt einen Eckpunkt dieser Polygonenseite berühren muss.
Weiterhin gibt es noch einen Punkt der Museumspolygone, den dieser Strahl berühren muss. Entweder wird der Strahl nochmals von einer Zacke begrenzt, oder er trifft einen Eckpunkt. Würde er nämlich nicht von einer Zacke begrenzt werden und auch mitten auf eine Seite treffen, ist unmöglich, dass er den Endpunkt eines sichtbaren Segments getroffen hat. Man kann den Strahl dann nämlich langsam um den ersten Eckpunkt drehen, wodurch man einen größeren Teil der letztlich getroffenen Seite sieht. Diesen Drehen kann man solange durchführen, bis man mit dem Strahl auf eine Begrenzung (eine zweite Ecke) stößt.

Das heißt also, dass jeder Strahl von der Weglinie zu dem äußeren Rand eines sichtbaren Liniensegmentes mindestens 2 Punkte des Museumspolygons schneidet. Wenn man nun eine Gerade durch je zwei Eckpunkte legt, und den Schnittpunkt mit dem Museumspolygon und dem Weg berechnet, hat man folglich jeden äußeren Randpunkt eines sichtbaren Liniensegments mitsamt einem dazugehörenden Punkt auf der Weglinie gefunden:
doku.5 `doku.5`

Wie man an dem Bild auch erkennen kann, liegt der dem Treffpunkt entgegengesetzte Randpunkt eines getroffenen sichtbaren Segments, immer auf der Seite des Strahls, auf der nicht die letzte berührte Zacke ist.
Falls beide Zacken auf derselben Seite liegen, (oder sogar identisch sind, wie im Bild) trifft der Strahl nicht den Rand eines sichtbaren Liniensegmentes, sondern nur die einen Punkt innerhalb des Segmentes und ist damit nutzlos.

1.3 Sichtbare Bereiche

Da man nun alle Randpunkte der sichtbaren Segmente gefunden hat und jeweils weiß, in welcher Richtung der Rest des Segments liegt, kann man die Segmente selbst finden.
Dazu betrachtet man alle Wände des Museums separat, und sortiert die darauf liegenden Punkte nach ihrem Abstand von einem der Eckpunkte der Wand. Es gibt hierbei zwei verschiedene Sorten von Punkten. Bei den einen liegt der Rest des Segmentes weiter vom gewählten Eckpunkt entfernt, als der Punkt selber, bei den anderen ist es genau umgekehrt. Die erste Art von Punkten nenne ich im folgenden Startpunkte die anderen Endpunkte. Welcher Punkt zu welcher Sorte gehört, wurde im Programm bereits berechnet. (siehe oben)
Hat man einen Start- und Endpunkt, die von dem gleichen Punkt auf der Weglinie sichtbar sind, kann man ein Dreick bilden das diese Punkt als Eckpunkte hat und das komplett vom Punkt auf der Weglinie sichtbar ist, sofern die Strecke zwischen den beiden Randpunkten sichtbar ist. Sind die Start/Endpunkte von unterschiedlichen Punkten aus sichtbar, kann man entsprechend ein solches Viereck machen. Ein solches Drei- oder Viereck nenne ich „Bereich“.
Am einfachsten wäre es nun, wenn auf der Linie abwechselnd, Start und Endpunkte liegen würden. Dann könnte man die Startpunkte mit dem jeweils nächsten Endpunkt verbinden, und hätte alle sichtbaren Segmente der Linie, da kein sichtbares Segment nach einem Endpunkt einen Startpunkt enthält. Dies liegt daran, dass ein Punkt immer bedeutet, dass eine Zacke die Sicht in eine Richtung blockiert. Würde nun auf einen Endpunkt ein Startpunkt folgen, würde diese Zacke innerhalb des Bereichs zu diesen Randpunkten liegen.
Es kann allerdings auch vorkommen, dass auf einen Startpunkt direkt weitere Startpunkte folgen, oder dass auf einen Endpunkt weitere Endpunkte, zum Beispiel, wenn man einem Teil der Weglinie weiter sehen kann, als von einem anderen. Das bedeutet zwar, dass die gesamte Linie, vom ersten Startpunkt bis zum letzten Endpunkt sichtbar ist, man weiß aber nicht, ob auch der Bereich zu den beiden äußersten Punkten sichtbar ist.
Der Unterschied zu dem Fall vorhin liegt darin, dass es hierbei häufig möglich ist, den Bereich von einem der weiter innen liegenden Startpunkte zu einem inneren Endpunkt zu sehen. Leider ist es nicht direkt möglich herauszufin-

den, welche der Punkte einen sichtbaren Bereich erzeugen und häufig gibt es auch mehrere, die dies tun. Eine einfache Lösung hierzu ist es, einfach alle Bereiche für je einen Start- und Endpunkt auf Sichtbarkeit zu überprüfen. Ein Bereich ist genau dann komplett sichtbar, wenn kein Teil der Museumspolygone innerhalb von diesem Bereich liegt, und das ist genau dann der Fall, wenn kein Eckpunkt dieses Polygons dort liegt. (Die einzige Alternative wäre es, wenn der Bereich von einer Seite des Polygones geschnitten würde. In dem Fall wäre aber der Strahl zum Randpunkt des Segments unterbrochen und es wäre überhaupt kein Bereich zu überprüfen). Überprüft man also alle Punkte des Polygons, weiß man definitiv, ob der Bereich sichtbar ist.
Das einzige Problem dabei ist, dass das Viereck möglicherweise konkav ist oder sich selbst schneidet. Das löse ich, indem ich zuerst feststelle, ob einer dieser Fälle vorliegt, indem ich überprüfe, welche Eckpunkte des Vierecks auf welcher Seite der einzelnen Viereckseiten liegen, und das Viereck entsprechend in zwei Dreiecke aufteile. Dann kann man einfach testen, auf welcher Seite bezüglich der Dreieckseiten der Eckpunkt der Museumspolygone liegt. Wenn es immer dieselbe ist, ist er innerhalb der Dreiecke und somit innerhalb des Vierecks, wenn nicht, außerhalb.
Es gibt auch noch schnellere Tests, die feststellen können, ob ein Bereich nicht völlig sichtbar ist, allerdings sagen diese nur, ob der Bereich sichtbar ist oder vielleicht nicht, bzw. ob es nicht sichtbar ist, oder vielleicht doch. Der erste Test ist, zu überprüfen, auf welcher Seite des Strahls von dem Punkt auf der Weglinie zum sichtbaren Randpunkt des Segments der jeweils andere Randpunkt liegt. Liegt er auf der Seite, auf der auch die Zacke liegt, die den Strahl begrenzt, ist diese Zacke innerhalb des Bereichs. (natürlich kann es sein, dass der Strahl garnicht von einer Zacke begrenzt wird).
Der zweite ist, zu überprüfen, ob der andere Randpunkt auf der selben Seite vom Strahl liegt, wie der Punkt von dem man ihn sehen kann. Trifft das zu, kann es sein, dass der Bereich sichtbar ist, oder auch nicht. In jedem Fall ist der Bereich nutzlos, da man im Normalfall, wenn einer der Punkte nicht der Rand der Weglinie ist, mehr sehen kann, wenn sie auf unterschiedlichen Seiten liegen. Nur wenn es sich um Randpunkte der Weglinie handelt, wäre es effizienter, diesen Bereich zu nehmen, als ihn durch einen sich selbst schneidenden und zwei Dreiecke zu ersetzen. Diese Ersatzpolygone werden aber sowieso erzeugt, da sie sichtbar sind, und es auch vorkommen kann, dass nur ein paar von den „Ersatzpolygonen“ sichtbar sind, und der nutzlose Bereich überhaupt nicht. Diese Fälle zu unterscheiden, wäre wahrscheinlich noch ineffizienter, als einfach nur diesen Bereich abzufangen.

Der letzte Test ist, ob es sich bei dem Bereich um ein Dreieck handelt, also sowohl der Start als auch der Endpunkt, vom selben Punkt auf der Linie aus sichtbar sind. In dem Fall muss es komplett sichtbar sein, da es, wenn es nicht so wäre, Start- und Endpunkte auf der Wand geben müsste, die das Segment und somit auch den Bereich für ungültig erklären.

1.4 Der Wahrscheinlichkeitsalgorithmus

Ich habe noch einen zweiten Algorithmus zu dem bisher beschrieben programmiert, der auf komplett andere Weise arbeitet, und wesentlich simpler und langsamer ist. Es wird einfacher von jeder möglichen Position im Museum überprüft, welche anderen Positionen sichtbar sind, und zu jeder Position ist eine Wahrscheinlichkeit gespeichert, die angibt, wie wahrscheinlich eine Aufenthalt des Nachtwächters dort ist.
Wenn man nun bei jeder Position, die Wahrscheinlichkeit, dass der Nachtwächter nicht an der anderen Position ist, von der er sie aus sehen kann, mit der bisherigen Wahrscheinlichkeit multipliziert, dass er die Position nicht sehen kann, erhält man die Wahrscheinlichkeit, dass er die Position nicht sehen kann, unter Berücksichtigung des neuen Punktes. (Also ein Wahrscheinlichkeitsbaum, bei dem der Ast für das nicht Eintreten, des Ereignisses berechnet wird.)
Dieser Algorithmus hat den großen Nachteil, dass er weitaus langsamer ist, als der andere, da ja, für ein sinnvolles Ergebnis, eine große Anzahl von Positionen überprüft werden muss. Dafür kann er wesentlich leichter erweitert werden, zum Beispiel, so dass die Wahrscheinlichkeit für Sichtbarkeit bei weiter entfernten Punkten abnimmt. Dafür muss man nur, die Wahrscheinlichkeit mit der multiplizieren wird (also die für ein Fehlen des Wächters) vor der Multiplikation um einen Bruchteil des Abstands erhöhen. Das habe ich auch implementiert, und zwar so, dass die Wahrscheinlichkeit für „nicht sehen“ wie das Taschenlampenlicht proportional zum Quadrat der Entfernung zunimmt, und ab einer bestimmten Entfernung 100% wird.
Was man auch noch leicht einbauen könnte wäre eine Abnahme der Sicht bei besonders dunklen Stellen, wozu man noch Lampen berücksichtigen müsste, was dadurch möglich ist, dass man die Helligkeit wie die Sichtbarkeit zwischen allen Punkten berechnet. Automatisch gehen auch mehrere Wächter, indem man einfach nicht verbundene Bereiche zeichnet.
Ganz allgemein, kann man diesen Algorithmus gut erweitern, da man ja In-

formationen über alle Punkte hat. Der andere Algorithmus ist dagegen sehr speziell, und kann nur schwer neue Funktionen kriegen.
Weiterhin hat er den Vorteil, dass es nur linear zur Anzahl der Polgonenlinien ist (dafür aber Quadratisch zur Auflösung der Punkte), was bei großen Museen, mit vielen Winkel sinnvoller sein kann, als die ungefähr kubische Laufzeit des anderen. Man kann außerdem schnell ein provisorisches Ergebnis mit geringer Auflösung erhalten.

Wenn es nur um die Sichtbarkeit geht, ist der Algorithmus der über Polgonanalyse geht, zweifelsohne sehr stark überlegen, da er schneller und eleganter ist.
Spielt Rechenzeit aber keine Rolle, wäre der andere, der empirisch vorgeht, wegen seiner Flexibilität besser.

Kapitel 2

Programmdokumentation

2.1 Einlesen der Daten

Ein Polygon wird im Programm als Array von Linien gespeichert. Eine Linie ist dabei ein Record namens `TLine` mit folgenden Feldern:

Name	Typ	Beschreibung
`start`	`TPointf`	Startpunkt der Linie
`dir`	`TPointf`	Richtungsvektor der Linie
`left`	`PLine`	vorherige Linie
`right`	`PLine`	nächste Linie
`sees`	**array of** `TSeenPos`	sichtbare Punkte

Der Typ `TPointf` ist dabei ebenfalls ein Record, mit zwei Feldern `x` und `y` vom Typ `double`. Der Endpunkte der Linie wird nicht gespeichert, da er mit dem Startpunkt der nächsten Linie `right^.start` und dem Punkt `start` + `dir` identisch ist. Die Verweise auf die nächste und vorherige Linie sind eigentlich ebenso sinnlos, da man ja auch über die Arrayindizes auf diese Linien zugreifen kann. Die Zeiger haben aber den Vorteil, dass man sie so setzen kann, dass man bei der letzten Linie im Array automatisch zur ersten kommt, ohne auf die Arraygrenzen achten zu müssen. Was es genau mit `sees` auf sich hat, erkläre ich später (in Sektion 2.3), beim Einlesen ist das nicht weiter wichtig.

Geladen wird ein Polygon von der Funktion **loadPolygon**, die als Parameter **loadPolygon**

eine Textdatei `f`, die Anzahl der Punkte im Polygon `len` entgegen nimmt und Zeiger auf zwei Punkte `min`, `max` entgegen nimmt. Zurückgegeben wird zum einen das Polygon, zum anderen die kleinsten und größten Koordinaten des Polygons. Die Koordinaten werden dabei in den beiden Punkten gespeichert. Da die Anzahl der Punkte des Polygons bekannt ist, können einfach so viele Zeilen der Textdatei gelesen werden. Jede Zeile wird dann in zwei Teile, entsprechend den beiden Koordinaten, am ersten Leerzeichen geteilt. Jede Koordinate wird dann in einen Gleitkommawert umgewandelt, als Startpunkt der Linie gespeichert und mit den bisherigen minimalen, maximalen Koordinaten verglichen.
Außer bei der ersten gelesenen Zeile, werden anschließend die restlichen Felder gefüllt. Dabei bekommt `left` einen Zeiger auf die vorherige Linie und das Feld `right` von der letzten Linie bekommt einen Zeiger auf die aktuelle. Ebenfalls wird die Richtung `dir` der letzten Linie durch Subtraktion des letzten gelesenen Punkts vom aktuell gelesenen berechnet. Nachdem alle Punkte gelesen wurden, werden noch die Felder der letzten Linie berechnet, da ja immer nur die Felder der vorherigen initialisiert werden konnten.

loadMuseumCli

Eine ganze Datei wird von der Methode **loadMuseumClick** geladen, die aufgerufen wird, wenn der Benutzer auf den Button `loadMuseum` klickt. Dabei wird jeweils eine Zeile gelesen, dann werden von dieser Zeile die ersten 8 Zeichen gelöscht, die übrig gebliebenen Zeichen werden dann in eine Zahl umgewandelt, und schließlich die oben beschriebene Funktion **loadPolygon** mit der nun bekannten Anzahl von Koordinaten aufgerufen und das Ergebnis in dem letzten Eintrag von `museum` (`museum` ist ein Array von Polygonen) abgespeichert.
Nachdem die Datei vollständig gelesen wurde, wird die Breite und Höhe des Museums über die Variablen `min` und `max` berechnet, und daraus der größte Zoomfaktor, bei dem das Museum vollständig sichtbar ist, indem einfach die Breite und Höhe der zur Ausgabe verwendeten PaintBox durch die Maße des Museums geteilt werden.

loadWayClick

Die andere Methode, die eine Datei einliest ist **loadWayClick** die, wie man am Namen erkennen kann, den Weg des Wächters liest.
Ich habe mich dafür entschieden, ein ähnliches Format, wie bei dem Museumspolygon zu verwenden. In der ersten Zeile steht dabei die Zahl der Punkte, und anschließend stehen in jeder Zeile je zwei Koordinaten. (Also, der einzige Unterschied ist, dass nur ein Polygon drin steht, und vor der Zahl

der Punkte kein „polygon“ steht.
loadWayClick liest die erste Zeile, ruft dann **loadPolygon** auf, um die Punkte zu lesen und speichert das Ergebnis in `guardsWay`.
Anschließend wird die Methode **lookAround** aufgerufen, die die sichtbaren Bereiche berechnet und die Ausgabe aktualisiert. Wie diese Ausgabe konkret geschieht, beschreibe ich nun:

2.2 Ausgabe

Die Ausgabe wird von der Methode **outputPaintBoxPaint** erledigt, deren Aufgabe nur darin besteht, einige Polygone zu zeichnen. **outputPaintBox**
Das Zeichnen von einem Polygon wird dabei über die Delphimethode **Canvas.Polygon** (genau genommen handelt es sich um eine Windowsfunktion, aber Delphi wrappt sie) erledigt, die alle Punkte eines an sie übergebenen Arrays verbindet und den Inhalt des Polygons füllt. Das komplizierteste dabei ist, dass die gespeicherten Koordinaten Gleitkommazahlen sind, aber diese Methode nur Ganze Zahlen verarbeiten kann. Deshalb werden die Koordinaten gerundet an die Methode übergeben, und damit das Bild nicht zu klein wird, werden sie vorher mit einem Zoomfaktor (siehe 2.1) multipliziert. Dann wird die y-Achse invertiert, um dieselbe Ausgabe, wie auf dem Aufgabenblatt zu erhalten. Damit man die Ausgabe auch verschieben kann, wird anschließend noch ein Vektor namens `nullpoint` addiert. Berechnet werden diese Umrechnungen von der Unterfunktion **transform**. **transform**
Die andere Unterprozedur **drawPolygon** ruft diese Funktion auf, um alle Punkte eines übergebenen Polygons umzuwandeln und anschließend zu zeichnen. **drawPolygon**
outputPaintBoxPaint zeichnet nun zuerst das erste in `museum` gespeicherte Polygon (also den Außenrand des Museums), und anschließend mit geänderter Hintergrundfarbe die anderen (die Löcher). Die Farbeinstellungen erfolgen dabei über die Standarddelphimethoden. Danach werden, je nach gewählter Einstellung, mit nochmals geänderter Hintergrundfarbe oder transparent, die sichtbaren Bereiche eingezeichnet. Dabei handelt es sich um Vierecke, die durch jeweils 4 Punkte definiert sind, die in dem Array `seenAreas` gespeichert sind. Da für diese Vierecke nur die Punkte gespeichert sind, kann nicht die Prozedur **drawPolygon** verwendet werden, sondern die Punkte werden **drawPolygon** gleich in **outputPaintBoxPaint** umgerechnet. Dadurch stehen die trans-

formierten Punkte auch noch zur Verfügung, um den jeweils zweiten und dritten Punkt dieser Vierecke (das sind die, die auf der getroffenen Seite des Museumspolygon liegen) mit einer dicken, roten Linie (im Ausdruck auf einem Graustufendrucker nur noch dicken) verbunden. Am Schluss wird noch transparent das Wegpolygon gezeichnet.

2.3 Berechnung der Sichtbarkeit

lookAround

Die sichtbaren Bereiche werden von der Prozedur **lookAround** berechnet. Sie ruft 3 Methoden auf, die den Abschnitten aus dem vorherigen Kapitel entsprechende Operationen durchführen: **lookFromPoint**, **lookFromWayLine** und **convertLookPoints**.
lookAround ruft nun für jede Linie des Weges **lookFromWayLine**, um die Randpunkte alle von einer Linie aus sichtbaren Liniensegmente zu berechnen, und **convertLookPoints**, um aus diesem Punkten die sichtbaren Bereiche zu ermitteln, auf.
lookFromWayLine ruft nun wiederum **lookFromPoint** auf, was ich jetzt zuerst beschreibe.

lookFromPoint

lookFromPoint nimmt als einzigen Parameter `p`, den Punkt entgegen, von dem aus die Sichtbarkeit berechnet werden soll.
Dann werden mit zwei for-Schleifen für jeden Punkt des Museums (also für jeden Punkt jedes Polygons in der Variablen `museum`) folgende Anweisungen durchgeführt: Zuerst wird ein Strahl vom Punkt `p` zum gewählten Punkt erzeugt, indem durch Subtraktion des Punktes `p` von dem gewählten Punkt die Richtung des Strahls berechnet wird. Der durch Startpunkt `p` und Richtung eindeutig definierte Strahl wird zusammen mit dem Museum an die Funktion **findRayHitPoints** übergeben. Diese Prozedur berechnet den ersten Schnittpunkt des Strahls mit den übergebenen Polygonen und wird später beschrieben. Wichtig ist hier allerdings das von ihr zurückgelieferte Ergebnis. Es handelt sich um einen Record mit folgenden Feldern:

Name	**Typ**	**Beschreibung**
`poly`	`integer`	getroffenes Polygon
`line`	`integer`	im Polygon getroffene Linie
`where`	`double`	Position des Punktes auf der Linie
`when`	`double`	Position des Punktes auf dem Strahl
`between`	**array of record**	berührte Zacken

Die Positionen auf der Linie liegen linear im Interval $[0, 1]$, wobei 0 der Startpunkt der Linie ist und 1 der Endpunkt. Die Position auf dem Strahl ist ähnlich, 0 bezeichnet den Startpunkt des Strahles und 1 den Punkt, der durch Addition von Richtung und Startpunkt entsteht, also in diesem Fall, den Punkt zu dem der Strahl geschickt wird. Handelt es sich bei diesem Punkt um den Eckpunkt einer Zacke, kann die Position allerdings auch über 1 lie-

gen. Ein Eintrag im Feld `between` hat folgende Unterfelder:

Name	Typ	Beschreibung
`poly`	`integer`	getroffenes Polygon
`point`	`integer`	im Polygon getroffener Punkt
`side`	`integer`	Seite auf der, die Zacke liegt
`when`	`double`	Position des Punktes auf dem Strahl

Genaugenommen ist `point` nicht direkt der Index des Punktes, sondern der Index der Linie, die den Punkt als Startpunkt hat. `side` ist entweder -1 oder $+1$, wobei -1 links vom Strahl und $+1$ rechts vom Strahl bedeutet.
Diese Seitennummerierung verwende ich auch noch an anderen Stellen im Programm, wobei dann noch 0 hinzu kommt, was aber in diesem Fall bedeuten würde, dass die Zacke auf dem Strahl liegt.

Nach dem Aufruf von **findRayHitPoints** muss nur noch das Ergebnis interpretiert werden. Dazu wird zuerst überprüft, auf welcher Seite der Strahl von Zacken begrenzt wird. Eine Begrenzung auf der linken Seite wird in der `boolean`-Variable `hideLeft`, eine auf der rechten entsprechend in `hideRight` gespeichert.
Anschließend wird die Sichtbarkeit der ersten sichtbaren Zacken auf der linken und rechten Seite des Strahles gespeichert. Dafür wird die Prozedur **searchVisibleLine** mit dem Startpunkt des Strahls und der Linie, deren Startpunkt der Eckpunkt der Zacke ist, aufgerufen. Diese Prozedur überprüft, wie der Strahlenstartpunkt im Bezug zur den Linien, die den Punkt als Start- und Endpunkt haben, liegt, und speichert die Sichtbarkeit im `sees`-Feld einer der beiden Linien. Wie genau sie das tut, beschreibe ich später.

Als nächstes wird überprüft, ob auf beiden Seiten des Strahles Zacken liegen. Ist das der Fall, ist der Strahl nutzlos, abgesehen von der Sichtbarkeitsmarkierung bei den beiden Zacken, und es wird zum nächsten Punkt übergegangen. Ansonsten wird überprüft, auf welcher Seite eine Zacke liegt.
Dann wird überprüft, ob ein Startpunkt getroffen wurde oder nicht. (Ein Endpunkt wird übrigens nie getroffen, da **findRayHitPoints** das als Treffer eines Startpunkts der vorherigen Linie speichern würde).
Wurde ein Eckpunkt getroffen, enthält mindestens eine, der an sie grenzenden Linien, ein sichtbares Segment. Das Problem ist jetzt herauszufinden, um welche es sich handelt. Dazu wird die Funktion **whichSide** aufgerufen, die überprüft auf welcher Seite eines übergebenen Strahls ein übergebener Punkt liegt und die später beschrieben wird. Liegt der Endpunkt der Linie,

also der Startpunkt der nächsten Linie, auf der selben Seite wie die Zacke, oder gibt es keine Zacke, ist der Punkt sichtbar und es kann die Prozedur **addSeePoint** aufgerufen werden, um den Punkt als sichtbar zu speichern. (siehe unten)
Wurde kein Eckpunkt getroffen und der Strahl wird von einer Zacke begrenzt, muss ein Sichtbarkeitseintrag für diese Linie erstellt werden. Dazu muss zuerst einmal festgestellt werden, in welcher Richtung das Liniensegment vom Punkt aus liegt. (Bei einer getroffenen Ecke ist das simpel, von der Ecke weg). Im folgenden nenne ich die Richtung, in der der Startpunkt liegt „linke Seite“ und die Richtung in der der Endpunkt liegt „rechte Seite“. Das erklärt auch die Bezeichnungen „left“/„right“ der Felder im Linienrecord für die vorherige/nächste Linie. Liegt der Startpunkt der Linie auf derselben Seite wie die begrenzende Zacke, muss das sichtbare Segment auf der vom Startpunkt abgewandten Seite, also „rechts“ liegen. Ansonsten umgekehrt.
Die Prozedure **lookFromPoint** ist damit zu Ende beschrieben, abgesehen davon, dass nun der nächste Punkt untersucht wird.

addSeePoint

Da die Prozedur **addSeePoint** sehr kurz ist, beschreibe ich sie schon hier und nicht erst später, wie die anderen aufgerufenen Funktionen. Außerdem versteht man dann die an sie übergebenen Parameter besser. Die sind:

Name	Typ	Beschreibung
`line`	`TLine`	getroffene Linie
`where`	`double`	getroffene Position auf der Linie
`seeRight`	`boolean`	liegt das sichtbare Segment rechts?
`hiddenRaySide`	`integer`	Die Seite auf der eine Zacke liegt
`from`	`TPointf`	Startpunkt des Strahls

Hierbei ist anzumerken, dass `hiddenRaySide` auch 0 sein kann, dann nämlich, wenn keine Zacke den Strahl begrenzt.
Als nächstes zeige ich die Felder des `TSeenPos`-Records, von dessen Typ die Einträge im `sees`-Array sind:

Name	Typ	Beschreibung
`where`	`double`	getroffene Position auf der Linie
`seeRight`	`boolean`	liegt das sichtbare Segment rechts?
`hiddenRaySide`	`integer`	Die Seite auf der eine Zacke liegt
`from`	`TPointf`	Startpunkt des Strahls
`wherePos`	`TPointf`	getroffener Punkt

Wie man sieht, stimmen die Parameter von **addSeePoint** und die Felder von `TSeenPos` fast völlig überein. Die einzigen Unterschiede sind, dass bei

`TSeenPos` die Angabe der getroffenen Linie fehlt - was natürlich daran liegt, dass `TSeenPos`-Variablen Einträge in einem Feld einer Linie sind - und dass ein (redunantes) Feld namens `wherePos` hinzugekommen ist.
addSeePoint sucht nun einfach, von rechts aus, den ersten Eintrag in dem `sees`-Array der Linie, der weiter links als die übergebene Position ist. Dann werden alle Einträge rechts von diesem Eintrag noch ein Feld weiter nach rechts verschoben und die übergegebenen Werte werden eingetragen. Dadurch ist das Array immer aufsteigend sortiert.
Abschließend muss noch ausgerechnet werden, wo der getroffene Punkt liegt, in dem die Richtung mit der Position multipliziert wird und der entstandene Vektor an den Startpunkt der Linie angehängt wird.

Die nächste wichtige Prozedur ist **lookFromWayLine**, die als einzigen Parameter eine Linie entgegen nimmt. Zuerst wird **lookFromPoint** für die beiden Endpunkte aufgerufen. Dann werden alle Paare von Punkten des Museums durchlaufen, indem je zwei Polygone `p1` und `p2` sowie zwei Linien `l1` und `l2` dieser Polygone ausgewählt werden, deren Startpunkte als gewählte Punkte genommen werden. **lookFromWayLi**
Sind zwei Punkte ausgewählt, wird ein Strahl von einem Punkt zum anderen erzeugt, indem einfach ein Punkt als Startpunkt und der anderen minus des Startpunkts als Richtung gewählt wird. Da es egal ist, ob der Strahl von `p1,l1` zu `p2,l2` oder umgekehrt verläuft, wird die Schleife als kleine Optimierung abgebrochen, wenn die Indizes vom ersten Punkt größer sind, als die vom zweiten. Als erstes wird nun die Richtung so gewählt, das der Strahl von `p1,l1` zu `p2,l2` verläuft.
Da nun ein Strahl existiert, kann mit der Funktion **whichSide** überprüft werden, auf welcher Seite vom Strahl die an die Punkte angrenzenden Wände (bzw. deren anderen Eckpunkte) liegen. Liegen beide Wände eines Punktes auf derselben Strahlseite, handelt es sich um eine Zacke und die Seite, auf der sie den Strahl begrenzt wird in `side1` bzw. `side2` gespeichert. Ist es keine Zacke, wird eine 0 gespeichert. Dann kann überprüft werden, ob beide Zacken, sofern vorhanden, auf derselben Seite liegen. Ist das der Fall, trifft der Strahl keinen Begrenzungspunkt eines Segments und ist daher, wie oben beschrieben, nutzlos.
Nun kann der Treffpunkt des Strahles mit der übergebenen Weglinie berechnet werden, wobei die Weglinie ebenfalls als Strahl interpretiert wird. Dann wird die Gleichung $start_1 + u * dir_1 = start_2 + v * dir_2$ nach u und v aufgelöst (das geht da $start_i$ und dir_i 2-dimensionale Vektoren sind), wobei

herauskommt: $u = \frac{dir_{2_x} * (start_{1_y} - start_{2_y}) + dir_{2_y} * (start_{2_x} - start_{1_x})}{dir_{1_x} * dir_{2_y} - dir_{1_y} * dir_{2_x}}$ und entsprechend $v = \frac{dir_{1_x} * (start_{1_y} - start_{2_y}) + dir_{1_y} * (start_{2_x} - start_{1_x})}{dir_{1_x} * dir_{2_y} - dir_{1_y} * dir_{2_x}}$

Das Ergebnis existiert natürlich nur, wenn die Strahlen nicht parallel oder identisch sind, in dem Fall wäre der Divisor 0 und die Punktekombination ist entweder nicht sichtbar, oder schon von den Randpunkten der Weglinie aus gesehen. Weiterhin geben u und v den Abstand vom Strahlstartpunkt in Vielfachen der Länge des Richtungsvektors an.

Im Programm bezeichnet `u` die Position auf dem Strahl und `v` die auf der Weglinie. `v` muss dabei zwischen 0 und 1 liegen, ansonsten wird die Linie nicht getroffen, sondern nur eine identisch liegende Gerade. `u` kann dagegen einen beliebigen Wert haben, liegt er ebenfalls zwischen 0 und 1, ist der Weg zwischen den beiden Punkten, ist er größer als 1, liegt der Weg hinter Punkt `p2,l2`, ansonsten, wenn `u` negativ ist, hinter `p1,l1`. Da es einfacher ist, wenn man davon ausgehen kann, dass der Weg in Richtung des Strahls liegt, wird in diesem letzten Fall, der Strahl umgedreht und `p2,l2` als Startpunkt gewählt.

Anschließend muss geprüft werden, ob dieser Strahl den Weg tatsächlich trifft, oder ob er vorher von einer Wand des Museumspolygons aufgehalten wird. Dafür wird die bereits erwähnte Prozedur **findRayHitPoints** aufgerufen, die den tatsächlichen Treffpunkt eines Strahles mit dem Polygon zurückgibt. Liegt dieser Treffpunkt näher als der Weg, oder zwischen den beiden gewählten Punkten, sind diese nicht beide sichtbar, und können ignoriert werden.

Ansonsten kann man vom Weg wenigstens diese beiden Punkte sehen und über die Prozedur **searchVisibleLine** werden die Sichtbarkeitsmarkierugen der an die beiden Punkt angrenzenden Linien, gespeichert. Vorher wird allerdings noch u in die oben gezeigt Gleichung eingesetzt, um die Koordinaten des Punktes auf der Weglinie zu erhalten, von dem aus die Sichtbarkeit besteht.

Waren die Ecken der gewählten Punkte keine Zacken oder liegt die Weglinie zwischen den beiden Punkten (also $\texttt{u} < 1$), so ist auch nichts wichtiges mehr von dem Wegpunkt aus sichtbar, da der Strahl von dem Wegpunkt zu einem der gewählten Punkte nicht sinnvoll begrenzt wird.

Ansonsten wird nun ein Strahl in die umgekehrte Richtung geschickt, bei dem nicht mehr der Wegpunkt in der Richtung der Strahls liegt, um die vom Sichtstrahl getroffene Stelle auf dem Museumsrand mit **findRayHitPoints** zu finden. Über **addSeePoint** kann nun eine Sichtbarkeitsmarkierung ge-

speichert werden, wofür aber noch herausgefunden werden muss, in welche Richtung das getroffene Segment sichtbar ist. Dafür wird wie in **lookFromPoint** die Seite auf der der Startpunkt der Linie liegt, mit der Seite, auf der die begrenzende Zacke liegt, verglichen. Dies ist, wie unter „Lösungsidee“ beschrieben, die vom Wegpunkt weiter entfernte Zacke, also entweder die erste, oder, wenn der Strahl gedreht werden muss, weil die getroffene Position negativ ist, die zweite Zacke.

lookAround ruft nun, nachdem **lookFromWayLine** erfolgreich die Randpunkte aller von einer Linie aus sichtbaren Segmente berechnet hat, **convertLookPoints** auf, um diese Randpunkte in sichtbare Bereiche umzuwandeln. **convertLookPoi**
Dazu wird für jede Linie im Museumspolygon diese Schleife ausgeführt:
Das (sortierte) `sees`-Array wird von links nach rechts stückweise mit einer while-Schleife durchlaufen. Für jeden Punkt, der dabei am linken Rand eines Segments liegt, (liegt er am rechten Rand, ist irgendein Fehler aufgetreten) wird nun zuerst der nächste Punkt, der am rechten Rand liegt, gesucht. Dann weiß man, dass zwischen diesen Punkten nur Punkte liegen, die an einem linken Rand liegen, und die Indizes der äußersten Punkte am linken Rand, werden in `border1Start` und `border1End` gespeichert. (Also `border1Start` ist der erste Punkt am linken Rand und `border1End` der letzte, so dass zwischen diesen Punkten nur welche liegen, die ebenfalls am linken Rand eines Segments liegen können).
Anschließend wird dasselbe für die Punkte am rechten Rand gemacht, und die Indizes dieses Bereichs in `border2Start` und `border2End` gespeichert. Dann wird der Index für den nächsten Durchlauf auf den des nächsten Punkts hinter `border2End` gesetzt.
Jetzt muss nur noch für alle Paare von Punkten, von denen einer am linken und einer am rechten Rand liegen kann, (also der erste liegt im ersten Interval, der zweite im zweiten) überprüft werden, ob der entsprechende Bereich sichtbar ist, wofür der Reihe nach die unter „Lösungsidee“ beschriebene Tests durchgeführt werden.
Zuerst wird getestet, ob die Punkte `sees[l].from` und `sees[m].from`, also die Punkte, von denen aus die gewählten Segmenträndern sichtbar sind, gleich sind. Wenn ja, ist das resultierende Polygon ein Dreieck und auf jeden Fall sichtbar, so dass die anderen Tests übersprungen werden.
Als nächstes wird ein Strahl von `sees[l].from` zu `sees[l].wherePos` (der gewählte potentielle linke Randpunkt) erzeugt, und überprüft, ob die von einer Zacke begrenzte Seite dieselbe ist, auf der auch `sees[m].from` liegt. Die

begrenzte Seite muss dabei nicht neu berechnet werden, da das alte Ergebnis in `sees[l].hiddenRaySide` beim Speichern der Sichtbarkeitsmarkierung mit gespeichert wurde. Die Seite auf der `sees[l].from` liegt, muss allerdings mit **whichSide** neu berechnet werden. Liegen sie nun auf derselben Seite, ist der Bereich leider unsichtbar, und der nächste wird überprüft. Dann wird getestet, ob `sees[l].from` und `sees[l].wherePos` auf der gleichen Seite liegen, und wenn ja, ist der Bereich, wie oben beschrieben, unsichtbar.
Diese Tests werden anschließend für den Strahl von `sees[m].from` zu `sees[m].wherePos` wiederholt.
Wurde dabei nicht heraus gefunden, dass der Bereich nicht sichtbar ist, muss nun für jeden Punkt des Museums überprüft werden, ob er in dem Bereich liegt. Dafür habe ich eine Extrafunktion **isPointInQuad** geschrieben, die später beschriebene wird, und die das überprüft.
Wurde der Bereich nun für sichtbar erklärt, wird ein neuer Eintrag in das Array `seenAreas` eingefügt, das alle sichtbaren Vierecke in Form von vier Punkten enthält. (die sichtbaren Dreiecke sind auch da, wobei sie allerdings als Vierecke mit einer doppelten Ecke interpretiert werden). Die Punkte, die dieses neue Viereck begrenzen, sind die sichtbaren Randpunkte und die Punkte, von denen aus diese Randpunkte gesehen werden können. Dabei werden die Punkte so eingefügt, dass die sichtbaren Randpunkte die Indizes 1 und 2 bekommen, damit sie bei der Ausgabe extra verbunden werden können.
Danach muss nur das `sees`-Array wieder geleert werden, und die Berechnung kann mit der nächsten Linie fortfahren, bis alle Linien durch sind und die Sichtbarkeit berechnet wurde.

Nun müssen nur noch die ganzen Hilfsfunktionen beschrieben werden, die ich bisher ausgelassen habe. Ich gehe dabei in der Reihenfolge vor, in der sie aufeinander angewiesen sind, und in der sie auch im Quellcode stehen.
Danach ist die erste Funktion die Funktion **whichSide**, die in ihren ersten beiden Parametern einen Strahl entgegen nimmt und als dritten einen Punkt. Alle diese Parameter sind `TPointf`s. **whichSide**
Ihre Funktionsweise kann man sich so vorstellen, dass der Strahl zusammen mit dem Punkt, so verschoben wird, dass der Strahl am Ursprungspunkt beginnt. Dann wird das Koordinatensystem linear verzerrt, indem alle Punkte entsprechend ihrem Abstand von der y-Achse parallel zur x-Achse verschoben werden, also indem von jeder x-Koordinate die dazugehörende y-Koordinate multipliziert mit einem konstanten Faktor subtrahiert wird. Dieser Faktor ist dabei so gewählt, dass der Strahl genau auf die alte y-Achse gelegt wird.

Da alle Punkten mit gleicher y-Koordinate um dieselbe Strecke verschoben werden, bleibt dadurch die Ordnung der Punkte innerhalb des Koordinatensystems unverändert, und die Seite des Punktes kann nun einfach durch Vergleich mit der y-Achse ermittelt werden.
Ein Problem gibt es allerdings, wenn der Strahl parallel zur x-Achse ist. Dann kann man ihn nicht so verzerren, dass er auf die y-Achse fällt. Zum Glück kann man aber in diesem Fall einfach die y-Koordinaten vergleichen.
Wenn der Strahl in eine negative x oder y-Richtung geht, um sicherzustellen, dass bei einer Umkehrung beider Vorzeichen der Richtung auch die Seiten getauscht werden.

whichLineSide

Die nächste Funktion ist **whichLineSide**. Sie überprüft, auf welcher Seite einer Linie ein Punkt liegt und wird für eine Funktion gebraucht, die ich später beschreibe.
Der einzige Unterschied zu **whichSide** liegt darin, dass der zweite Parameter hier ebenfalls einen Punkt angibt und keine Richtung. Der Ablauf ist jetzt so, dass einfach die Richtung vom ersten zum zweiten Punkt ausgerechnet wird, und dann als Parameter an **whichSide** übergeben wird.

findRayHitPoin

Danach kommt **findRayHitPoints**, die wie schon beschrieben die Treffpunkte eines übergebenen Strahls mit einem ebenfalls übergebenen Array von Polygonen berechnet.
Dafür wird der Schnittpunkt des übergebenen Strahles mit jeder Linie des Polygons berechnet, mit der gleichen Formel, die ich schon in der Prozedure **lookFromWayLine** beschrieben habe. Wieder gibt es Variablen `u` und `v`, wobei `u` die Position auf dem Strahl und `v` die auf der Linie angibt, und `v` muss wieder im Bereich 0 bis 1 liegen.
Ist der Strahl parallel zur getesteten Linie, kann er sie nicht passieren, sondern endet an einem ihrer Endpunkte. Das heißt, auch wenn der Punkt sichtbar ist, beginnt kein Liniensegment an ihm, und die Linie kann daher ignoriert werden.
Als nächstes werden Rundungsfehler am Rand (z.B.: 1e-20 statt 0) beseitigt, und überprüft, ob der getroffenen Punkt im zulässigen Bereich des Strahls ist, also näher am Startpunkt des Strahles ist, als das bisherige Ergebnis `result.when`, und größer als 0. Anschließend müssen die Zacken erkannt werden. Dazu wird, wenn der getroffene Punkt ein Startpunkt ist (um das genau zu erkennen, mussten die Rundungsfehler beseitigt werden), mit **whichSide** berechnet, auf welchen Seiten des Strahles, die an den Punkt angrenzenden

Seiten sind. Sind beide auf derselben Seite, ist es tatsächlich eine Zacke und die entsprechenden Informationen (Punkt, Seite und Position auf dem Strahl) werden an der passenden Position in das sortierte `between`-Array eingefügt, und zur nächsten Linie gegangen.
Ein Endpunkt einer Linie wird dagegen ignoriert (also die folgende Linie wird dann überprüft), da es sich dabei ja gleichzeitig um den Startpunkt der nächsten handelt und es sinnlos ist, zweimal dieselbe Zacke zu speichern.
Wenn das Programm nun noch nicht beschlossen hat, die Linie zu ignorieren, wurde diese getroffen, und der Treffpunkt wird in `result` gespeichert. Anschließend müssen alle bisherigen Zacken gelöscht werden, die weiter entfernt sind als die getroffene Linie, da sie ja jetzt nicht mehr passiert werden. Da das Array sortiert ist, geht dies recht schnell.

Nun würde die Funktion **addSeePoint** kommen, da ich diese aber schon beschrieben habe, kommt nun **searchVisibleLine**. Diese Prozedur nimmt den **searchVisibleLi**
Startpunkt `raystart` eines Strahls und eine Linie `line` entgegen, wobei der Startpunkt dieser Linie der Punkt ist, der von einem Strahl getroffen wurde. Zuerst wird überprüft, ob der Startpunkt des Strahles bei beiden Linien (der übergebenen Linie und der vorherigen, die den Startpunkt als Endpunkt hat) auf der selben Seite liegt. Das ist gleichbedeutend damit, dass beide Linien auf unterschiedlichen Seiten des Strahles liegen. Wenn ja, besitzen beide Linien am Rand ein sichtbares Segment, und die entsprechenden Punkte, werden mit **addSeePoint** markiert. Wenn nicht, ist nur eine der Linien sichtbar und diese verdeckt die andere, das heißt, dass sie den Startpunkt des Strahls, von der anderen Linie und ihrem Endpunkt trennt. Eine Linie ist daher sichtbar, wenn der Startpunkt des Strahles, und der Endpunkt der anderen Linie, auf verschiedenen Seiten liegen. Hier wird dann wieder **addSeePoint** aufgerufen, allerdings mit anderen Parametern.

Die letzten beiden Funktionen sind **isPointInTri** und **isPointInQuad**. Von **isPointInTri**
ihren Parametern her, sind beide Funktionen sehr ähnlich. **isPointInTri** erwartet die drei Eckpunkte eines Dreiecks und dann den Punkt, der auf seine Lage überprüft werden soll. **isPointInQuad** bekommt dieselben Informationen, allerdings noch einen Punkt mehr.
Die Funktionsweise ist allerdings unterschiedlich. Da ein Dreieck immer konvex und einfach ist, kann einfach überprüft werden, ob der Punkt in Bezug zu allen drei Linien des Dreiecks auf derselben Seite liegt. Wenn ja, ist er in dem Dreieck, ansonsten außerhalb.

Die Funktionsweise von **isPointInQuad** ist dagegen etwas komplizierter, da ein Viereck konkav oder sich selbst schneidend sein kann. Daher werden diese Fälle einzeln überprüft. Wenn zwei Seiten sich schneiden, das Viereck also ungefähr wie eine Sanduhr aussieht, entstehen zwei Dreiecke. Angenommen, das Viereck liegt so, dass eines der Dreiecke oben und das andere unten liegt. Nimmt man die beiden Eckpunkte des oberen Dreiecks und verbindet sie mit einem der unteren, entsteht wieder ein Dreieck, und nimmt man den anderen unteren Punkt, entsteht ein anderes Dreieck. Überlagert man nun diese Dreiecke, so dass man alles abschneidet, das nicht in beiden Dreiecken liegt, entsteht ein neues Dreieck, das exakt dem oberen entspricht. Man muss also nur überprüfen, ob der Punkt in diesen beiden Dreiecken liegt, um festzustellen, ob er im oberen Dreieck liegt. Macht man das äquivalent noch mit dem unteren Dreieck, so erfährt man, ob der Punkt im Viereck liegt.
Der andere störende Fall ist, dass das Viereck konkav ist, also eine der Diagonalen liegt nicht im Viereck. Dann liegt aber die andere Diagonale im Viereck, und das Viereck kann entlang dieser in zwei Dreiecke geteilt werden. Es muss dabei auch nur für eine Diagonale überprüft werden, ob sie im Viereck liegt, da falls sie es

isPointInQuad

2.4 Der empirische Ansatz

Wenn dieser Algorithmus aktiviert wurde, werden die Polgonpunkte direkt nach dem Einlesen, so transformiert, dass sie im Bereich 0 bis 1 liegen, um Veränderung der Genuigkeit besser handhaben zu können.
Die eigentliche Berechnung steht aber komplett in **loadWayProbability-Click**. Zuerst wird ein Bild geladen, dass die Wahrscheinlichkeiten für den Wächterufenthalt enthält. voll schwarz bedeutet 100%, voll weiß 0% und Graustufen liegen dazwischen. Diese umgerechneten Wahrscheinlichkeiten werden in `invWayProbability` gespeichert. **loadWayProbab**
Um Punkte zu erkennen, die nicht im Museum liegen, wird dieses in eine Bitmap gezeichnet, und alle Punkte, die außerhalb oder in einem Loch sind, werden in `invWayProbability` mit einer 2 gekennzeichnet.
Dann wird für jeden Punkt berechnet, welche Bereiche von dort aus sichtbar sind. (Das geht mit dem anderen Algorithmus schneller, als mit einem einfachen Strahlentest für jedes Punktepaar). Nun werden die anderen Punkte durchlaufen, und überprüft, welche in einem sichtbaren Bereich liegt. Ist einer sichtbar, wird die Wahrscheinlichkeit in invSeeProbability mit der invertierten Aufenthaltswahrscheinlickeit multipliziert und entsprechend gesetzt.
Die Ausgabe geschieht wieder in outputPaintBoxPaint. Dort werden einfach, alle Punkte entsprechend eingefärbt, und das Museum drüber gezeichnet.

Kapitel 3

Quellcode

```
unit a1_i;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls,math, ComCtrls;

const inf=1.0/0.0;
      epsilon=1e-10;          //Kleinste berücksichtigte Differenz
type
  ppointf=^tpointf;
  tpointf=record              //Ein Punkt des Museums
    x,y: double;              //Seine Koordinten
  end;
  TSeenPos=record             //Eine sichtbarer Punkt auf einer Linie
      where: double    ;          //Position auf der Linie (0: Start, 1: Ende)
      seeRight: boolean;      //liegt das Segment in Richtung Ende
      hiddenRaySide: integer; //begrenzte Seite des Strahl zu diesem Punkt
      from,wherePos:tpointf; //Strahlstartposition und getroffener Punkt
    end;
  TSeenArray=array of TSeenPos;//Mehrere getroffene Punkte (wird sortiert)

  PLine=^TLine;
  TLine=record
    start: tpointf;           //Startpunkt der Linie
    dir: tpointf;             //Richtung der Linie
    left,right:PLine;         //vorherige und nächste Linie
```

```
  sees:array of TSeenPos; //auf der Linie getroffene Punkte
end;
TPolygon=array of TLine; //Ein Polygon = eine Liniefolge

TRayHitPoints=record     //Informationen über die Treffpunkte eines Strahls
  poly, line: integer;   //Getroffenes Polygon und dort getroffene Linie
  where, when: double;   //Position des Treffpunktes auf dem Strahl und der Linie
  between: array of record     //Passierte Ecken
             poly, point: integer; //getroffenes Polygon und getroffener Punkt
             side: integer;        //Seite der Eckpunkte (beide sind auf derselben)
             when: double;         //Position auf dem Strahl
           end;
end;
TmuseumForm = class(TForm)
  Panel1: TPanel;
  outputPaintBox: TPaintBox;
  Label1: TLabel;
  museumFileName: TEdit;
  loadMuseum: TButton;
  Label2: TLabel;
  wayFileName: TEdit;
  loadWay: TButton;
  fillSeenAreas: TCheckBox;
  drawBorderedSeenAreas: TCheckBox;
  loadWayProbability: TButton;
  Label3: TLabel;
  wayProbabilityFileName: TEdit;
  polygonalMode: TRadioButton;
  probabilityMode: TRadioButton;
  Panel2: TPanel;
  showWayProbability: TRadioButton;
  showVisibility: TRadioButton;
  ProgressBar1: TProgressBar;
  progress: TLabel;
  procedure loadMuseumClick(Sender: TObject);
  procedure FormCreate(Sender: TObject);
  procedure outputPaintBoxPaint(Sender: TObject);
  procedure loadWayClick(Sender: TObject);
  procedure FormMouseWheel(Sender: TObject; Shift: TShiftState;
    WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);
  procedure outputPaintBoxMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
  procedure outputPaintBoxMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
```

```
    procedure fillSeenAreasClick(Sender: TObject);
    procedure loadWayProbabilityClick(Sender: TObject);
  private
    { Private-Deklarationen }
    clickPos, oldnullpoint:tpoint;
  public
    { Public-Deklarationen }                                          80
    zoom: double;                          //Zoomfaktor für die Ausgabe
    nullpoint:tpoint;                      //Verschiebungsfaktor "
    museum: array of TPolygon;             //Das geladene Museum
    museumSize: tpointf;                   //Größe des Museums
    guardsWay: TPolygon;                   //Der Weg des Wächters
    seenAreas: array of array[0..3] of tpointf; //alle sichtbaren Bereiche

    invSeeProbability: array of array of double;
    invWayProbability: array of array of double;
                                                                      90
    //Berechnet die von einem Punkt aus sichtbaren Segmente
    procedure lookFromPoint(const p: tpointf);
    //Berechnet die von einer Linie aus sichtbaren Segmente
    procedure lookFromWayLine(var line: TLine);
    //Wandelt die getroffenen Segmentpunkte in sichtbare Bereiche um
    procedure convertLookPoints;
    //Berechnet, alle von einem Weg aus sichtbaren Bereiche
    procedure lookAround();
  end;
                                                                      100
var
  museumForm: TmuseumForm;

implementation

{$R *.DFM}

//====================File I/O================
//Lädt ein Polgon mit einer gegeben Anzahl von Punkten und sucht die minimalen
//und maximalen Koordinaten                                           110
function loadPolygon(var f: TextFile; len: integer; min,max: ppointf):TPolygon;
var i,j:integer;
    s:string;
begin
  SetLength(result,len);              //Speicher vorreservieren
  for i:=0 to len-1 do begin
    Readln(f,s);                      //Zeile lesen
    j:=pos(' ',s);                    //Trennzeichen zweischen zwei Koord. suchen
```

```
    with result[i].start do begin
      x:=StrToFloat(copy(s,1,j-1)); //erste Koordinate lesen
      delete(s,1,j);                //sie zusammen mit dem Trennzeichen löschen
      y:=StrToFloat(s);             //zweite Koordinate lesen
      if max<>nil then begin        //äußerste Koordinaten suchen
        if (x>max.X) then max.X:=x;
        if (y>max.Y) then max.Y:=y;
      end;
      if min<>nil then begin
        if (x<min.X) then min.X:=x;
        if (y<min.Y) then min.Y:=y;
      end;
    end;
    if i<>0 then begin
      result[i].left:=@result[i-1];    //Zeiger auf vorherige Linie setzten
      result[i-1].right:=@result[i]; //Dort einen auf diese setzen
      result[i-1].dir.x:=result[i].start.x-result[i-1].start.x; //Richtung
      result[i-1].dir.y:=result[i].start.y-result[i-1].start.y; //    berechnen
    end;
  end;
  //Dasselbe wie am Ende der for-Schleife für die ersten Linie (Nachfolger der letzten)
  result[len-1].right:=@result[0];
  result[0].left:=@result[len-1];
  result[len-1].dir.x:=result[0].start.x-result[len-1].start.x;
  result[len-1].dir.y:=result[0].start.y-result[len-1].start.y;
end;

//Es wurde auf den Button zum Laden des Museums geklickt
procedure TmuseumForm.loadMuseumClick(Sender: TObject);
var museumFile: TextFile;
    s:string;
    maxP,minP: tpointf;
    i,j: integer;
begin
  SetLength(museum,0);
  //Koordinatengrenzen mit maximalen Grenzen besetzten
  maxP.x:=-inf;maxP.y:=-inf;
  minP.x:=inf;minP.y:=inf;
  //Datei laden
  AssignFile(museumFile,museumFileName.Text);
  reset(museumFile);
  repeat
    Readln(museumfile,s); //erste Zeile eines Polygones lesen
    if lowercase(copy(s,1,8))='polygon ' then begin
      delete(s,1,8);                    //"polygon " abschneiden
```

```
      SetLength(museum,length(museum)+1); //Neues Polygon anhängen
      //Polygon in nun reservierten Speicher laden
      museum[high(museum)]:=loadPolygon(museumFile,strtoint(s),@minP,@maxP)
    end else raise Exception.create('Can''t understand "'+s+'"'); //falsches Format
  until eof(museumFile);

  CloseFile(museumFile);                                                    170

  museumSize.x:=maxP.X-minP.X;
  museumSize.y:=maxP.y-minP.y;
  if polygonalMode.checked then begin
    //Wenn der Algortihmus durch Polygonenanalyse arbeitet soll,
    //wird einfach ein passender Zoomfaktor gewählt
    zoom:=min(outputPaintBox.ClientHeight/(maxP.Y-minP.Y),
              outputPaintBox.ClientWidth/(maxP.X-minP.X));
  end else begin
    zoom:=max(museumSize.x,museumSize.y);                                   180
    //Sonst werden vorher die Koordinaten, in den 0 - 1 Bereich gemappt
    for i:=0 to high(museum) do
      for j:=0 to high(museum[i]) do begin
        museum[i,j].start.x:=(museum[i,j].start.x-minP.x)/zoom;
        museum[i,j].start.y:=(museum[i,j].start.y-minP.y)/zoom;
        museum[i,j].dir.x:=(museum[i,j].dir.x)/zoom;
        museum[i,j].dir.y:=(museum[i,j].dir.y)/zoom;
      end;
    zoom:=min(outputPaintBox.ClientHeight,outputPaintBox.ClientWidth);
  end;                                                                      190
  nullpoint.x:=0;
  nullpoint.y:=0;
  Refresh;
end;

//Einen Rundgang laden
procedure TmuseumForm.loadWayClick(Sender: TObject);
var f: TextFile;
    len: integer;
begin                                                                       200
  AssignFile(f,wayFileName.Text);
  reset(f);
  Readln(f,len);                          //Anzahl der Punkte einlesen
  guardsWay:=loadPolygon(f,len,nil,nil);//Rundgang einlesen
  CloseFile(f);

  lookAround;                             //Sichtbarkeit berechnen
```

```
  Refresh;
end;

//====================Ausgabe=================
procedure TmuseumForm.outputPaintBoxPaint(Sender: TObject);
  //Einen Punkt im Koordinatensystem des Museums ins Ausgabesystem umwandeln
  function transform(const p:tpointf): tpoint;
  begin
    result.x:=round(p.x*zoom)+nullpoint.x;
    result.y:=round(-p.y*zoom)+outputPaintBox.ClientHeight+nullpoint.y;
  end;
  //Ein Polygon zeichnen
  procedure drawPolygon(const p:TPolygon);
  var transformedPoly: array of tpoint;
      i:integer;
  begin
    SetLength(transformedPoly,length(p));
    for i:=0 to high(p) do
      transformedPoly[i]:=transform(p[i].start);
    outputPaintBox.Canvas.Polygon(transformedPoly)
  end;
var i,j:integer;
    temp: array[0..3] of tpoint;
    tempColor: integer;
begin
  if length(museum) = 0 then exit; //Kein Museum geladen => schüss
  with outputPaintBox.Canvas do begin
    if polygonalMode.Checked then begin //Im Polygonanalysemodus
      pen.Width:=1;                     //werden die Museumspolygone gefüllt
      pen.color:=clBlack;
      pen.Style:=psSolid;
      brush.Color:=clSilver;
      drawPolygon(museum[0]);           //Hauptpolygon zeichnen
      brush.Color:=clGray;
      for i:=1 to high(museum) do  //Löcher zeichnen
        drawPolygon(museum[i]);

      //Sichtbare Bereiche ausgeben
      if fillSeenAreas.Checked then brush.Color:=$A6D9AD //füllen
      else brush.Style:=bsClear;
      for i:=0 to high(seenAreas) do begin
        if drawBorderedSeenAreas.Checked then begin //umranden
          pen.Width:=1;
          pen.Color:=clGray;
        end else pen.Style:=psClear;
```

```
        //Polygon umrechnen und zeichnen
        for j:=0 to 3 do
          temp[j]:=transform(seenAreas[i,j]);
        Polygon(temp);

        //Treffpunkte auf der Linie
        pen.Style:=psSolid;
        pen.Width:=2;
        pen.Color:=clRed;
        moveto(temp[1].x,temp[1].y);
        LineTo(temp[2].x,temp[2].y);
      end;

      pen.Width:=1;
      pen.Color:=clBlack;
      brush.Style:=bsClear;
      drawPolygon(guardsWay);
    end else begin
      if showWayProbability.Checked then begin
        for i:=0 to high(invWayProbability) do
          for j:=0 to high(invWayProbability[i]) do begin
            tempColor:=min(255,max(0,round(invWayProbability[i,j]*255)));
            Pixels[i+nullpoint.x,j+nullpoint.y+outputPaintBox.ClientHeight-length(invSeeProbability)];
              :=rgb(tempColor,tempColor,tempColor);
          end;
      end else if showVisibility.Checked then begin
        for i:=0 to high(invSeeProbability) do
          for j:=0 to high(invSeeProbability[i]) do begin
            tempColor:=min(255,max(0,round(invSeeProbability[i,j]*255)));
            Pixels[i+nullpoint.x,j+nullpoint.y+outputPaintBox.ClientHeight-length(invSeeProbability)]
              :=rgb(tempColor,tempColor,tempColor);
          end;
      end else
      pen.Width:=1;
      pen.color:=clRed;
      pen.Style:=psSolid;
      brush.style:=bsClear;
      drawPolygon(museum[0]);
      for i:=1 to high(museum) do
        drawPolygon(museum[i]);
    end;
  end;
end;
```

```
//==================LOGIK=================
//Gibt zurück, auf welcher Seite vom Strahl pos,dir der Punkt p liegt.
//-1: links, 0: auf dem Strahl, 1: rechts
function whichSide(const pos,dir: tpointf; const p:tpointf): integer;
var temp:double;
begin
  //Feststellen auf welchwer Seite die Punkte liegen
  //pos1 zum Nullpunkt verschieben;
  //Koordintensystem so verzerren, dass dir auf die y-Achse fällt
  Result:=0;
  if abs(dir.y)<epsilon then begin //verzerren geht nicht, einfacher Vergleich
    if p.y<pos.y then result:=1
    else if p.y>pos.y then result:=-1;
    if dir.x<epsilon then Result:=-result;
  end else begin
    //x_new(x,y) = x - y * dir1.x / dir1.y
    //=> x_new(dir1.x,dir1.y) = 0
    temp:=((p.x - pos.x) - (p.y - pos.y) * dir.x / dir.y);
    if temp>epsilon then result:=1
    else if temp<-epsilon then result:=-1;
    if dir.y<0 then result:=-result;
  end;
end;

//Überprüft auf welcher Seite von der Linie a–b der Punkt p liegt
function whichLineSide(const a,b: tpointf; const p:tpointf): integer;
var dir: tpointf;
begin
  dir.x:=a.x-b.x; dir.y:=a.y-b.y; //Richtung für einen Strahl
  Result:=whichSide(a,dir,p);    //auf bekanntes Problem zurückleiten
end;

//Berechnet die Treffer eines Strahl mit einem Polygon
procedure findRayHitPoints(polys:array of TPolygon; const start,dir:tpointf;out result:TRayHitPoints);
var poly,line:integer;          //Momentan zu überprüfende Linie
    lineStart,lineDir: ppointf; //Start und Richtung dieser Linie
    u,v: double;                //Position des Treffpunkts auf dem Strahl und der Linie
    p1Side,p2Side:integer;      //Strahlseite auf der an einen Punkt grenzende Linien sind
    i:integer;                  //Loopindex
begin
  result.when:=inf;             //Maximaler Abstand
  SetLength(result.between,0);
  for poly:=0 to high(polys) do
    for line:=0 to high(polys[poly]) do begin
      lineStart:=@polys[poly,line].start; //Zeiger für einfacheren Zugriff
```

```
lineDir:=@polys[poly,line].dir;
//Regel für Schnittpunkte:
{    start+ u*dir = linestart + v*linedir, 0<=v<=1 }

if abs(dir.x*lineDir.y − dir.y*lineDir.x) < epsilon then
  continue; //parallel von Linien und Strahlen ignorieren (siehe Doku)
                                                                    350
//Position des Punktes auf der Linie berechnen
v := (dir.x*(start.y − linestart.y) + dir.y*(linestart.x − start.x))/
(dir.x*lineDir.y − dir.y*lineDir.x);
//Abbrechen, falls außerhalb der Linie
if abs(v)<epsilon then v:=0;
if abs(v−1)<epsilon then v:=1;
if (v<0) or (v>1) then continue;
//Position auf dem Strahl berechnen
u := (lineDir.x*(start.y − linestart.y) + lineDir.y*(linestart.x − start.x))/
            (dir.x*lineDir.y − dir.y*lineDir.x);                    360
//Abbrechen, falls hinter dem Strahl
if abs(u−1)<epsilon then u:=1;
if (u>result.when) or (u<epsilon) then continue;
//Startpunkt getroffen
if v=0 then begin
  //Berechnen auf welcher Seite des Strahls die angrenzenden Linie sind
  p1Side:=whichSide(start,dir,polys[poly,line].left^.start);
  p2Side:=whichSide(start,dir,polys[poly,line].right^.start);
  if (p1Side=p2Side) and (p1Side<>0) then begin //gleiche Seite => Zacke
    //=> der Strahl wird nicht geblockt, nur Strahlen, die neben ihm wären  370
    //Passende Position im sortierten between-Array suchen
    i:=high(result.between);
    while (i>=0) and (result.between[i].when>result.when) do
      dec(i);
    inc(i);
    //Punkt einfügen
    SetLength(result.between,length(result.between)+1);
    move(result.between[i],result.between[i+1],(high(result.between)−i)*sizeof(result.between[0]));
    result.between[i].poly:=poly;
    result.between[i].point:=line;                                  380
    result.between[i].side:=p1Side;
    result.between[i].when:=u;
    continue; //nächste Linie testen
  end;
end else if v=1 then continue; //wird später behandelt

//Kollision mit Polygonenseite
result.poly:=poly;
```

```
      result.line:=line;
      result.when:=u;
      result.where:=v;

      //Alle Zacken löschen, die hinter der Seite liegen
      i:=high(result.between);
      while (i>=0) and (result.between[i].when>u) do
        dec(i);
      setlength(result.between,i+1);
    end;
end;

//Speichert einen getroffenen Punkt
//Im wesentlichen werden die übergebenen Parameter einfach kopiert, nur wird
//sichergestellt, dass das seen-Array sortiert ist
procedure addSeePoint(var line:TLine; const where: double; const seeRight: boolean;
                                    const hiddenRaySide: integer; const from:tpointf);
var i:integer;
begin
  with line do begin
    //Suchen der passenden Position
    i:=high(sees);
    while (i>=0) and (sees[i].where>where) do
      dec(i);
    SetLength(sees,length(sees)+1);
    if (i=-1) and (sees[0].where<=where) then i:=high(sees)
    else inc(i);
    //alle Einträge nach rechts verschieben
    move(sees[i],sees[i+1],(high(sees)-i)*sizeof(sees[i]));
    //Parameter kopieren
    sees[i].where:=where;
    sees[i].seeRight:=seeRight;
    sees[i].hiddenRaySide:=hiddenRaySide;
    sees[i].from:=from;
    sees[i].wherePos.x:=start.x+where*dir.x;
    sees[i].wherePos.y:=start.y+where*dir.y;
  end;
end;

//Diese Prozedur wird aufgerufen, wenn der Strahl den Startpunkt von line getroffen
//hat und festgestellt werden muss, welche Linie sichtbar ist.
//Entweder line, die den Punkt nun mal als Startpunkt hat, line.left^, die den Punkt
//als Endpunkt hat, oder beide.
//Eine Linie ist dann sichtbar, wenn der Startpunkt des Strahles auf der selben
//Linienseite liegt, wie der andere Punkt der anderen Linie
```

```
procedure searchVisibleLine(const raystart: tpointf; var line: TLine);
var rayfromline1,
    rayfromline2: integer; //Seite des Strahlenstart von einer der Linie
begin
  //Fesstellen auf welcher Seite, der an den Startpunkt von line angrenzenden
  //Linien, jeweils der Strahlstartpunkt liegt
  rayfromline2:=whichSide(line.start,line.dir,raystart);                    440
  rayfromline1:=whichSide(line.left^.start,line.left^.dir,raystart);
  if rayfromline1=rayfromline2 then begin //Keine Linie wird von der nderen verdeckt
     addSeePoint(line,0,true,0,raystart);
     addSeePoint(line.left^,1,false,0,raystart);
  end else if whichSide(line.start,line.dir,line.left^.start)<>rayfromline2 then
     //line trennt line.left vom Strahl
     addSeePoint(line,0,true,0,raystart)
  else if whichSide(line.left^.start,line.left^.dir,line.right^.start)<>rayfromline1 then
     //line.left trennt line vom Strahl
     addSeePoint(line.left^,1,false,0,raystart)                             450
  else //Keine Linie sichtbar
     raise Exception.Create('Invisible point');
end;

//Überprüfen, ob der Punkt p in dem Dreieck a–b–c liegt
//Da ein Dreieck immer konvex ist, muss dafür nur überprüft werden,
//ob p in Bezug auf jede Dreieckseite auf derselben Seite liegt
function isPointInTri(const a,b,c,p:tpointf):boolean;
var s1,s2,s3: integer;
begin                                                                       460
  Result:=false;
  s1:=whichLineSide(a,b,p);
  s2:=whichLineSide(b,c,p);
  if s1<>s2 then exit;
  s3:=whichLineSide(c,a,p);
  result:=(s1=s3);
end;

//Überprüfen, ob der Punkt p im Viereck a–b–c–d liegt.
//Dabei werden die Fälle unterschieden, dass sich das Viereck selbst schneiden  470
//und dass es konkav ist.
function isPointInQuad(const a,b,c,d,p:tpointf):boolean;
begin
  if whichLineSide(a,b,c)<>whichLineSide(a,b,d) then begin
    //a–b und c–d schneiden sich
    //Das Viereck lässt sich in zwei Bereiche aufteilen:
      //a–b–c geschnitten mit b–c–d
      //a–c–d geschnitten mit a–b–d
```

```
    Result:=(isPointInTri(a,b,c,p) and isPointInTri(b,c,d,p)) or
            (isPointInTri(a,c,d,p) and isPointInTri(a,b,d,p));
  end else if whichLineSide(a,d,b)<>whichLineSide(a,d,c) then begin
    //a–d und b–c schneiden sich
    //Das Viereck lässt sich in zwei Bereiche aufteilen:
      //a–b–c geschnitten mit a–b–d
      //a–c–d geschnitten mit b–c–d
    Result:=(isPointInTri(a,b,c,p) and isPointInTri(a,b,d,p)) or
            (isPointInTri(a,c,d,p) and isPointInTri(b,c,d,p));
  end else if whichLineSide(a,c,b)=whichLineSide(a,c,d) then begin
    //konkav, da die Digonale a–c außerhalb liegt
    // => Aufteilung entlang der Diagonalen b–d
    Result:=(isPointInTri(a,b,d,p) and isPointInTri(b,c,d,p));
  end else begin
    //entweder konvex, oder konkav mit b–d außerhalb
    // => Aufteilung entlang der Diagonalen a–c ist möglich
    Result:=(isPointInTri(a,b,c,p) and isPointInTri(a,b,d,p));
  end;
end;

//Berechnet alle Segmente die man vom Punkt p aus sehen kann.
procedure TmuseumForm.lookFromPoint(const p: tpointf);
var i,j,k:integer;                //loop-Counter
    dir: tpointf;                 //Richtung des Strahls
    rayHit:TRayHitPoints;         //Informationen über die Strahlentreffer
    hideLeft,hideRight:boolean; //liegt auf einer der beiden Seiten eine Zacke
    hiddenside: integer;          //Seite auf der die einzigste Zacke liegt
    pointSide: integer;           //Seite auf der die Linie an einem getroffenen Punkt liegt
begin
  for i:=0 to high(museum) do
    for j:=0 to high(museum[i]) do begin
      //Für jeden Punkt wird die Richtung dorthin berechnet
      dir.x:=museum[i,j].start.x−p.x;
      dir.y:=museum[i,j].start.y−p.y;
      //Und der Strahl wird losgesendet
      findRayHitPoints(museum,p,dir,rayHit);

      if rayHit.when=inf then raise Exception.create('Hasn''t hit something');

      //Ist links oder rechts eine Zacke
      hideLeft:=false; hideRight:=false;
      for k:=0 to high(rayHit.between) do
        case rayHit.between[k].side of
          −1: hideLeft:=true; //Zacke links
          1: hideRight:=true; //Zacke rechts
```

```
    0: raise Exception.Create('Ray isn''t stopped, after point collision');
  end;

if hideLeft then //Zacke links
  for k:=0 to high(rayhit.between) do
    if rayHit.between[k].side=-1 then begin
      with rayhit.between[k] do
        searchVisibleLine(p,museum[poly,point]);//Eckpunkt ist sichtbar
      break;
    end;

if hideRight then //Zacke rechts
  for k:=0 to high(rayhit.between) do
    if rayHit.between[k].side=1 then begin
      with rayhit.between[k] do
        searchVisibleLine(p,museum[poly,point]);//Eckpunkt ist sichtbar
      break;
    end;

if hideLeft and hideRight then continue; //Links und rechts eine Zacke
                                         // => abbruch, nur Zacken sieht man
//Seite suchen, auf der eine Zacke ist
if hideleft then hiddenside:=-1
else if hideright then hiddenside:=1
else hiddenside:=0;

if rayHit.where=0 then begin //Ecke getroffen
  //Auf welcher Seite liegt die eine angrenzende Linie
  pointSide:=whichSide(p,dir,museum[rayhit.poly,rayhit.Line].right^.start);
  //Wird diese Seite nicht verdeckt
  if (pointSide<>hiddenSide) or (hiddenSide=0) then
    with museum[rayHit.poly,rayHit.line] do //Seite ist sichtbar
      addSeePoint(museum[rayHit.poly,rayHit.line],0,true,hiddenside,p);

  //Wird die andere Seite nicht verdeckt
  if (-pointSide<>hiddenSide) or (hiddenSide=0) then
    with museum[rayHit.poly,rayHit.line].Left^ do //ist sie auch sichtbar
      addSeePoint(museum[rayhit.poly,rayHit.line].Left^,1,false,hiddenside,p);
end else begin //Kante getroffen
  if not (hideleft or hideRight) then continue; //Strahl ist nutzlos
  //feststellen auf welcher Seite der Startpunkt liegt
  //Das Segment liegt in Linienrichtung, wenn der Startpunkt verdeckt wird
  addSeePoint(museum[rayHit.poly,rayHit.line],
              rayHit.where,
              whichSide(p,dir,museum[rayhit.poly,rayhit.Line].start)=hiddenSide,
```

```
                hiddenside,p);
    end;
  end;
end;


//Berechnet alle Segmente, die man von einer Weglinie aus sehen kann
procedure TmuseumForm.lookFromWayLine(var line: TLine);
var p1,l1,p2,l2: integer;          //Polygon und Linien loop-Variablen
    u,v: double;                   //Position des getroffenen Punktes (Strahl,Linie)
    raystart,waystart,waydir:ppointf;//Strahlstart, Weglinienstart, Wegrichtung
    raydir,wayhit: tpointf;        //Strahlrichtung und getroffener Punkt auf der Weglini
    divisor: double;               //zwischengespeicherter Nenner
    hitInfo: TRayHitPoints;        //Informationen über die Treffer eines Strahls
    side1,side2,tempside: integer;
 begin
  //Sichtbarkeit von den Linienecken berechnen
  lookFromPoint(line.start);
  lookFromPoint(line.right^.start);

  //Schönere Namen geben
  waystart:=@line.start;
  waydir:=@line.dir;
  for p1:=0 to high(museum) do
   for p2:=0 to high(museum) do
    if p1<=p2 then
     for l1:=0 to high(museum[p1]) do
      for l2:=0 to high(museum[p2]) do begin
       if (p1=p2) and (l1+1>=l2) then continue;
        //Jetzt sind zwei Punkte museum[p1,l1].start und ...p2,l2... gewählt

       //Strahlvektoren vom ersten zum zweiten Punkt speichern
       raystart:=@museum[p1,l1].start;
       raydir.x:=museum[p2,l2].start.x-raystart.x;
       raydir.y:=museum[p2,l2].start.y-raystart.y;

       //Position der Linien an den Punkten überprüfen
       side1:=whichSide(raystart^,raydir,museum[p1,l1].left^.start);
       tempside:=whichSide(raystart^,raydir,museum[p1,l1].right^.start);
       if (side1<>tempside) then side1:=0; //Keine Zacke an Punkt 1

       side2:=whichSide(raystart^,raydir,museum[p2,l2].left^.start);
       tempside:=whichSide(raystart^,raydir,museum[p2,l2].right^.start);
       if (side2<>tempside) then side2:=0; //Keine Zacke an Punkt 2
```

```
if (side1=side2) and (side1<>0) then
  continue; //Strahl ist nutzlos, da zwei Zacken auf derselben Seite

//Schnittpunkt mit dem Weg berechnen (siehe findRayHitPoints)
divisor:=raydir.x*wayDir.y - raydir.y*wayDir.x;
if abs(divisor)<epsilon then continue;
v := (raydir.x*(raystart.y - waystart.y) + raydir.y*(waystart.x - raystart.x))/divisor;
if (v<=0) or (v>=1) then continue;
u := (wayDir.x*(raystart.y - waystart.y) + wayDir.y*(waystart.x - raystart.x))/divisor;

if u<0 then begin //Strahl verläuft falschherum
  //Startpunkt wechseln
  raystart:=@museum[p2,l2].start;
  //Richtung invertieren
  raydir.x:=-raydir.x;
  raydir.y:=-raydir.y;
  //Neue Position in Anbetracht der beiden Änderungen berechnen
  u:=-u+1;
  //Zackeninformationen über die beiden Punkte tauschen
  tempside:=-side1;
  side1:=-side2;
  side2:=tempside;
end;

//Einen Strahl in Richtung des Weges aus senden
findRayHitPoints(museum,raystart^,raydir,hitInfo);
if (hitInfo.when<u) or (hitInfo.when<1) then
  continue; //Strahl zu früh unterbrochen

//Vorhin auf der Weglinie getroffenen Punkt ausrechnen
wayhit.x:=raystart^.x+u*raydir.x;
wayhit.y:=raystart^.y+u*raydir.y;
//Die beiden gewählten Punkte sind von diesem Punkt aus sichtbar
searchVisibleLine(wayhit,museum[p1,l1]);
searchVisibleLine(wayhit,museum[p2,l2]);

if (side1=0)or(side2=0) then continue; //Alle Punkte sind berücksichtigt
if u<1 then continue; //Weglinie kommt zu früh

//Strahl in die andere Richtung senden
raydir.x:=-raydir.x;
raydir.y:=-raydir.y;
//und Kollision mit dem Polygon berechnen
findRayHitPoints(museum,raystart^,raydir,hitInfo);
```

```
        //Sichtbaren Punkt speichern
        //Das Segment liegt auf der anderen Seite, als der zweite Punkt, also
        //wenn der Startpunkt der Linie auf derselben Seite wie dieser Punkt liegt,
        //zeigt das Segment in Richtung des Endes der Linie
        addSeePoint(museum[hitinfo.poly,hitinfo.line], hitinfo.where,
                    whichSide(raystart^,raydir,museum[hitinfo.poly,hitinfo.Line].start) >0=(side2>0),
                    side1,wayhit);
      end;
end;

//Berechnet die sichtbaren Bereiche aus den sichtbaren Segmenten
procedure TmuseumForm.convertLookPoints;
var i,j,k,l,m,n,o:integer;    //loop-Variablen (Schachtelung ist sehr übersichtlich)

    border1Start,border1End,border2Start,border2End: integer;

    raydir:tpointf;
    nextLoop: boolean;
begin
  border2Start:=-1; border1End:=-1; //wegen warnung über fehlende initialisierung
  //Es wird der größtmögliche sichtbare Bereich gewählt.
  //Das heißt bei mehreren aufeinanderfolgenden nach rechts blickenden Punkten
  //wird der erste genommen, bei nach links blickenden dagegen der letzte
  for i:=0 to high(museum) do
    for j:=0 to high(museum[i]) do
      //Linie auswählen
      with museum[i,j] do begin
        if length(sees)=0 then continue; //Keine sichtbaren Punkte => Weiter
        if length(sees)=1 then begin //Macht keinen Sinn => Abbruch
          setlength(seenAreas,0);
          exit;
        end;
        k:=0;
        while k<=high(sees) do begin
          border2End:=high(sees);
          border1Start:=k; //erster nach rechts blickender Punkt (Startpunkt)
          for l:=k+1 to high(sees) do
            if not sees[l].seeRight then begin
              border1End:=l-1; //letzter Startpunkt
              border2Start:=l; //erster nach links blickender (Endpunkt)
              break;
            end;
          if border1End<border1Start then begin //Macht keinen Sinn => Abbruch
            setlength(seenAreas,0);
            exit;
```

```
end;
for l:=border1End+1 to high(sees) do
  if sees[l].seeRight then begin
    border2End:=l-1; //letzer Endpunkt
    break;
  end;
k:=border2End+1;                                                    710

for l:=border1Start to border1End do //Je einen Startpunkt
  for m:=border2Start to border2End do begin //und Endpunkt auswählen
    if (abs(sees[l].from.x-sees[m].from.x)>epsilon) or
       (abs(sees[l].from.y-sees[m].from.y)>epsilon) then begin
      //Wenn kein Dreieck, dann, Sichtbarkeitstests durchführen
      //Strahl von dem Punkt, von dem aus der Startpunkt sichtbar ist
      //zum Startpunkt erstellen
      raydir.x:=sees[l].wherePos.x-sees[l].from.x;
      raydir.y:=sees[l].wherePos.y-sees[l].from.y;                  720
      //Liegt, der andere Blickpunkt auf derselben Seite, wie die
      //blockierende Zacke, so ist der Bereich nicht komplett sichtbar
      if sees[l].hiddenRaySide=
         whichSide(sees[l].from,raydir,sees[m].from) then continue;
      //Liegt, der andere Blickpunkt auf derselben Seite, wie der von
      //dort gesehene Endpunkt, so ist der Bereich nicht komplett sichtbar
      if whichSide(sees[l].from,raydir,sees[m].wherePos)=
         whichSide(sees[l].from,raydir,sees[m].from) then continue;
      //Die gleichen Tests für den Endpunkt ausführen
      raydir.x:=sees[m].wherePos.x-sees[m].from.x;                  730
      raydir.y:=sees[m].wherePos.y-sees[m].from.y;
      if sees[m].hiddenRaySide=
         whichSide(sees[m].from,raydir,sees[l].from) then continue;
      if whichSide(sees[m].from,raydir,sees[l].wherePos)=
         whichSide(sees[m].from,raydir,sees[l].from) then continue;

      //Haben die bisherigen Tests nichts erbracht, so muss für jede
      //Ecke des Polygon überprüft werden, ob sie innerhalb des neuen
      //Bereichs wäre
      nextLoop:=false;                                              740
      for n:=0 to high(museum) do
        for o:=0 to high(museum[n]) do
          if isPointInQuad(sees[l].from,sees[l].wherePos,sees[m].wherePos,sees[m].from,
                museum[n,o].start) then begin
            nextLoop:=true; //Leider ist die Ecke drin => nächste Kombination
            break;
          end;
      if nextLoop then continue;
```

```
            end;
                                                                    750
            //Bereich hat die Tests bestanden und ist somit sichtbar.
            SetLength(seenAreas,length(seenAreas)+1);
            seenAreas[high(seenAreas)][0]:=sees[l].from;
            seenAreas[high(seenAreas)][1]:=sees[l].wherePos;
            seenAreas[high(seenAreas)][2]:=sees[m].wherePos;
            seenAreas[high(seenAreas)][3]:=sees[m].from;
          end;
        end;
        //Bisherige Segmente löschen
        setlength(sees,0);                                          760
      end;
end;

//
procedure TmuseumForm.lookAround();
var i:integer;
begin
  setlength(seenAreas,0);
  for i:=0 to high(guardsway) do
  begin                                                             770
    lookFromWayLine(guardsway[i]);
    convertLookPoints;
  end;
end;



//==============Sonstiges (GUI)====================
procedure TmuseumForm.FormMouseWheel(Sender: TObject; Shift: TShiftState;
  WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);     780
begin
  if WheelDelta<0 then zoom:=zoom/(1.01*(WheelDelta/60))
  else zoom:=-zoom*1.01*WheelDelta/60;
  Refresh;
end;

procedure TmuseumForm.outputPaintBoxMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  clickPos.x:=x;                                                    790
  clickPos.y:=y;
  oldnullpoint:=nullpoint;
end;
```

```
procedure TmuseumForm.outputPaintBoxMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if ssleft in shift then begin
    nullpoint.x:=oldnullpoint.x+x-clickPos.x;
    nullpoint.y:=oldnullpoint.y+y-clickPos.y;                                    800
    Refresh;
  end;
  if zoom<>0 then
  Caption:='BWINF 24.2.1 '+floatToStr((x-nullpoint.x)/zoom)+':'+
                          floattostr(-(y-outputPaintBox.ClientHeight-nullpoint.y)/zoom);
end;

procedure TmuseumForm.fillSeenAreasClick(Sender: TObject);
begin
  Refresh;                                                                       810
end;

procedure TmuseumForm.FormCreate(Sender: TObject);
begin
  SetLength(museum,0);
  DecimalSeparator:='.';
end;

procedure TmuseumForm.loadWayProbabilityClick(Sender: TObject);
var guardWay: tbitmap;                                                           820
  procedure drawPolygon(const p:TPolygon);
  var transformedPoly: array of tpoint;
      i:integer;
  begin
    SetLength(transformedPoly,length(p));
    for i:=0 to high(p) do begin
      transformedPoly[i].x:=round(p[i].start.x*zoom);
      transformedPoly[i].y:=guardWay.height-round(p[i].start.y*zoom);
    end;
    guardWay.Canvas.Polygon(transformedPoly)                                     830
  end;
var
    x,y,x2,y2,i,j:integer;
    maxp,posp:integer;
    pos,pos2:tpointf;
    hitinfo: TRayHitPoints;
    visible: boolean;
    seeWidth: double;
```

```
begin
  guardWay:=TBitmap.create;
  guardWay.LoadFromFile(wayProbabilityFileName.text);
  zoom:=guardWay.width;
  if zoom<>guardWay.height then raise Exception.Create('nicht quadratisch');
  SetLength(invWayProbability,guardWay.width,guardWay.height);
  SetLength(invSeeProbability,guardWay.width,guardWay.height);
  for x:=0 to guardWay.width-1 do
    for y:=0 to guardWay.height-1 do
      invWayProbability[x,y]:=getrvalue(guardWay.Canvas.Pixels[x,y]) / 255;


  guardWay.Canvas.brush.color:=clRed;
  guardWay.Canvas.pen.style:=psClear;
  guardWay.Canvas.pen.width:=1;
  guardWay.Canvas.pen.Style:=psSolid;
  guardWay.Canvas.FillRect(rect(0,0,guardWay.width,guardWay.height));
  guardWay.Canvas.brush.color:=clWhite;
  drawPolygon(museum[0]);
  guardWay.Canvas.brush.color:=clRed;

  for i:=1 to high(museum) do
    drawPolygon(museum[i]);

  for x:=0 to high(invWayProbability) do
    for y:=0 to high(invWayProbability[x]) do
      if guardWay.Canvas.Pixels[x,y] = clred then
        invWayProbability[x,y]:=2;

  guardWay.free;

  for x:=0 to high(invSeeProbability) do
    for y:=0 to high(invSeeProbability[x]) do
      invSeeProbability[x,y]:=1;

  seeWidth:=sqr(4/Max(museumsize.x,museumsize.Y));

  maxp:=length(invSeeProbability)*length(invSeeProbability[0]);
  ProgressBar1.Max:=maxp;
  for x:=0 to high(invSeeProbability) do
    for y:=0 to high(invSeeProbability[x])do begin
      posp:=x*length(invSeeProbability[0])+y;
      ProgressBar1.Position:=posp;
      progress.Caption:=inttostr(posp)+'/'+inttostr(maxp);
```

```
      Application.ProcessMessages;
      if invWayProbability[x,y] < 1 then begin
        pos.x:=x/high(invSeeProbability);
        pos.y:=1-y/high(invSeeProbability[x]);
        setlength(seenAreas,0);
        try
          lookFromPoint(pos);                                                    890
          convertLookPoints;
        except
          continue;
        end;
        for x2:=0 to high(invSeeProbability) do
          for y2:=0 to high(invSeeProbability[x]) do begin
            if (x=x2)and(y=y2) then begin
              invSeeProbability[x2,y2]:=invSeeProbability[x,y]*invWayProbability[x,y];
              continue;
            end;                                                                 900
            if invWayProbability[x2,y2] = 2 then continue;
            pos2.x:=x2/high(invSeeProbability);
            pos2.y:=1-y2/high(invSeeProbability);
            if sQr(pos.x-pos2.x)+sQr(pos.y-pos2.y)>seeWidth then continue;

            visible:=false;
            for i:=0 to high(seenAreas) do
              if isPointInTri(seenAreas[i,0],seenAreas[i,1],
                              seenAreas[i,2],pos2) then begin
                visible:=true;                                                   910
                break;
              end;

            if visible then
              invSeeProbability[x2,y2]:=invSeeProbability[x,y]*(
                 max(0,min(1,invWayProbability[x,y]+(sQr(pos.x-pos2.x)+sQr(pos.y-pos2.y))/seeWidth))
                                    );
          end;
      end;
    end;                                                                         920
  Refresh;
end;

end.
```

Kapitel 4

Ablaufprotokoll

Die einzelnen Vierecke: