Programmierung einer 3D-Wegsuchfunktion

Die Engine 2
WDL-Skript 3
Die vorgegebenen Skripts 4
Die Wegsuche 5
Wegpunkte 6
SearchWay 7
Wegpunkttypen 8
Der Wegpunkteditor 9

Programmierung einer 3D-Wegsuchfunktion

Die Engine

Ich hatte schon einige Zeit mit dem 3D-Gamestudio programmiert, aber fast jeder Spieltyp benötigt eines: eine Wegsuche

Zunächst aber einige Erklärungen zum Aufbau des 3D-Gamestudios:

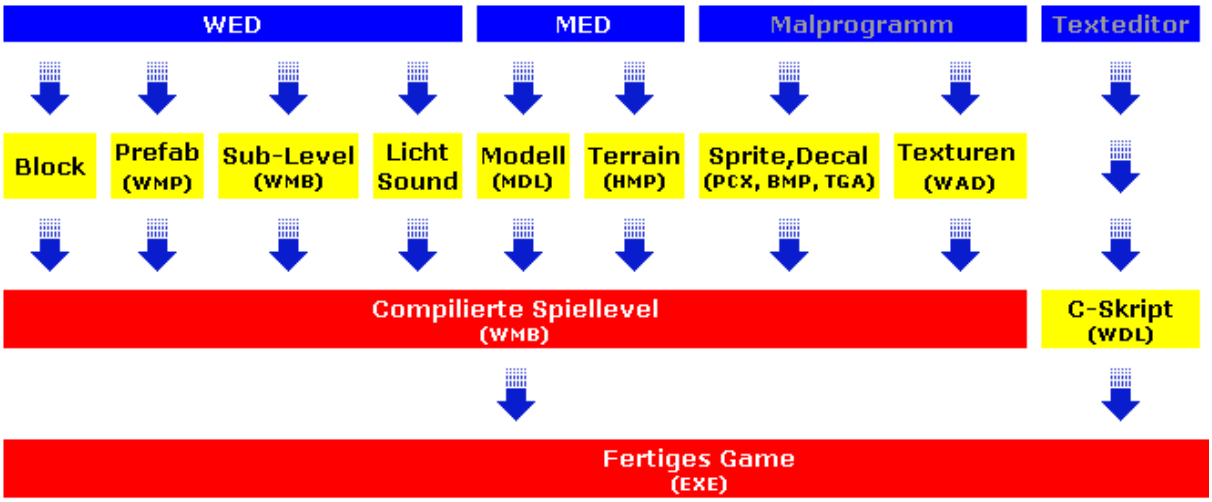


Abbildung 1: Aufbau des 3D-Gamestudios

WED: Dies ist der Leveleditor des 3D-Gamestudios. Hier wird das Level aus einzelnen Blöcken zusammengesetzt.

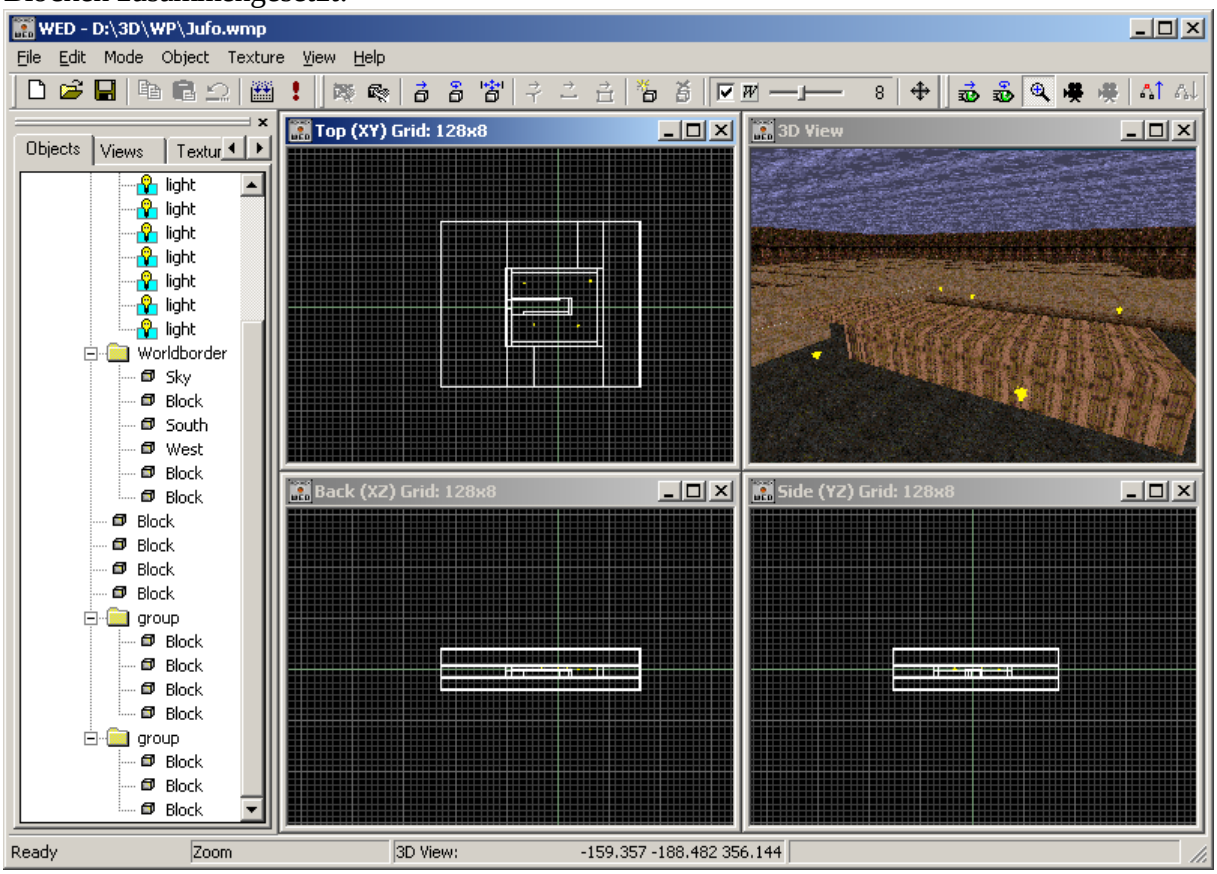
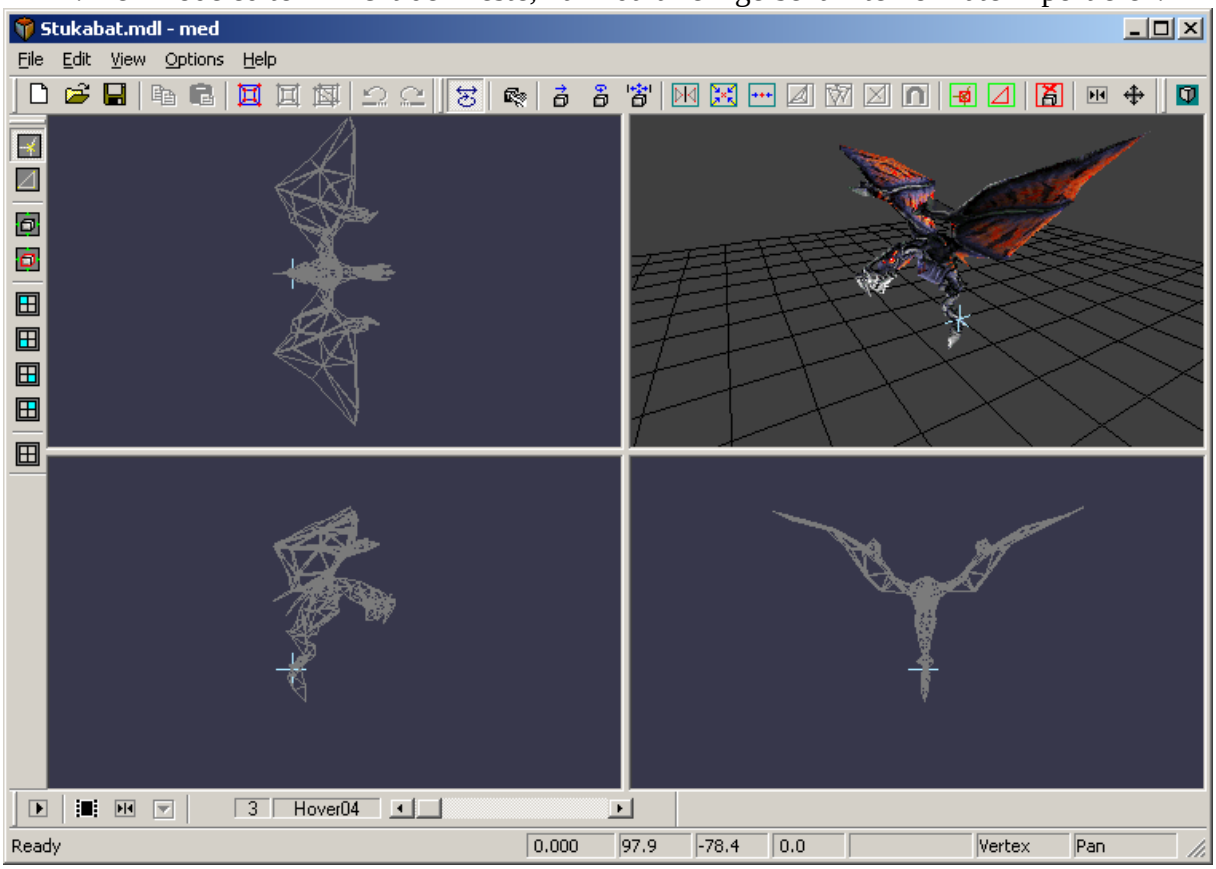


Abbildung 2: Der Level-Editor

MED: Der Modeleditor – nicht der Beste, kann dafür einige bekannte Formate importieren.



WDL-Skript

Für die Programmierung ist aber der Texteditor das wichtigste Werkzeug. Die Programmiersprache des 3D-Gamestudio ist WDL-Skript. Dies ist eine spezielle Programmiersprache für 3D-Spiele. WDL wird zwar auch C-Skript genannt, hat aber dennoch gewaltige Unterschiede. Zum einen stehen viele Funktionen zur Verfügung, die das Skript die Welt sehen lassen, z.B. vorgefertigte 3D-Rechenfunktionen usw. Andererseits fehlen wichtige Elemente der Programmiersprache C. Erst mit dem neuesten Update waren lokale Variablen möglich! Jedes bewegliche Objekt im 3D-Gamestudio besitzt eine WDL-Aktion. Hier ein Beispiel für ein Objekt, das sich einfach nur dreht:

```
Action rotate
{
	while (1) // Endlosschleife
	{
		my.pan += 1; // Drehe ein Grad weiter
		wait(1); // Warte einen Frame
	}
}
```

Der Aufruf von wait(1) ist extrem wichtig. Ohne ihn würde die Engine diese Aktion endlos durchführen und keine andere Aktion aufrufen. Wait(1) lässt die anderen Aktionen dran und lässt die Engine einen Frame darstellen. Dies ist eine einfache Form eines Multithreading-Systems.

Aber diese Aktion hat einen großen Fehler. Die Drehgeschwindigkeit ist abhängig von der Framerate. Bei 16 Frames pro Sekunde dreht sich das Objekt um 16 Grad pro Sekunde. Bei 64 Frames pro Sekunde dreht sich das Objekt allerdings 4 mal so schnell!

```
Action rotate
{
	while (1) // Endlosschleife
	{
		my.pan += time // Drehe etwas weiter
		wait(1); // Warte einen Frame
	}
}
```

Jetzt stimmt die Funktion. Time ist ein vordefinierter Wert, der angibt, wie viele Ticks seit dem letzten Frame vergangen sind. So dreht sich das Objekt um exakt 1 Grad pro 1 Tick. Und ein Tick ist nichts anderes als 1/16 Sekunde. Jetzt dreht sich das Objekt unabhängig von der Framerate um 16 Grad pro Sekunde.

Die vorgegebenen Skripts

Solche Aktionen muss man aber nicht selber schreiben – das 3D-Gamestudio hat für viele Zwecke vorgefertigte Funktionen in Dateien, die man mit „include“-Anweisungen einbinden kann. Ich nutze einige dieser Funktionen, z.B. „msg_show“ zum Anzeigen von Text auf dem Bildschirm, „actor_move“ um das Objekt, das die Aktion ausführt, etwas vorwärtsgehen zu lassen – mit Kollisionserkennung.

```
Action walk_to_camera
{
	While(1) { // Endlosschleife
		vec_set(temp,camera.x); // Kameraposition in Variable temp
		                        // kopieren
		vec_sub(temp,my.x);     // Distanz zwischen Kameraposition
		                        // und Wächter in Temp speichern
		vec_to_angle(my_angle,temp); // Vektor in Winkel umwandeln
		force=20;         // Mit Drehgeschwindigkeit 20 Grad/Tick
		actor_turn();     // Zur Richtung my_angle (zur Kamera) drehen
		force = 4;        // Mit Geschwindigkeit 4 Quants/Tick
		actor_move();     // vorwärtsgehen
		wait(1);    // Einen Frame abwarten
	}
}
```

Diese Aktion bewirkt, das der Wächter mit dieser Aktion immer in Richtung Kamera läuft.

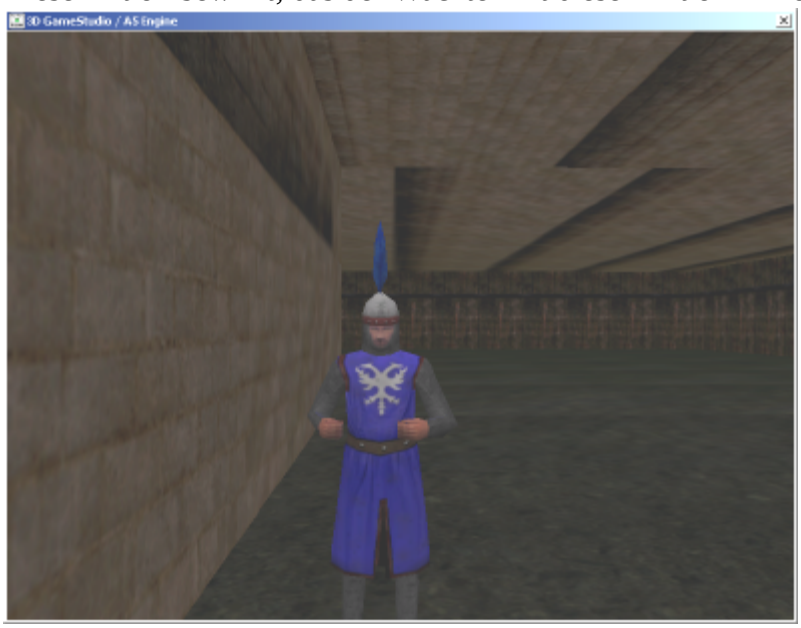


Abbildung 3: Der Wächter läuft zur Kamera

Befindet sich eine Wand zwischen Wächter und Kamera, so läuft er dagegen und bleibt hängen.

Die Wegsuche

Dies ist die Frage, an der meine Wegsuche ansetzt. Wie kann der Wächter zur Kamera finden? Nun, er könnte jetzt nach links abbiegen und solange geradeaus gehen, bis er das Ziel sieht. Hier ist die passende Aktion dafür:

```
Function guard_event()
{
      my.pan+=90; // Nach links abbiegen
      my.skill1=1; // In Skill1 wird gespeichert, ob der Wächter beim
                   // Warten auf freie Sicht ist
}
Action walk_to_camera
{
      my.event=guard_event; // Bei Ereignissen diese Funktion aufrufen
      my.enable_block=on;   // Ereignis "gegen Mauer gehen" aktivieren
      my.skill1=0; // Direkt auf Kamera losgehen
      While(1) {
            If (my.skill1==0){
                  vec_set(temp,camera.x);
                  vec_sub(temp,my.x);
                  vec_to_angle(my_angle,temp);
                  force=20;
                  actor_turn();
            }else{
                  trace_mode=ignore_me+use_box;
                  if (trace(my.x,camera.x)==0){ // Ziel in Sicht
                        my.skill1=0;      // Hindernis umgangen
                  }
            }
            force = 4;
            actor_move();
            wait(1);
      }
 }
```

Dies bringt aber kaum eine Verbesserung:

Solange nur eine Mauer im Weg ist, funktioniert diese Methode.
Ist der Weg aber etwas komplizierter, so bleibt der Wächter hängen oder läuft in die falsche Richtung.

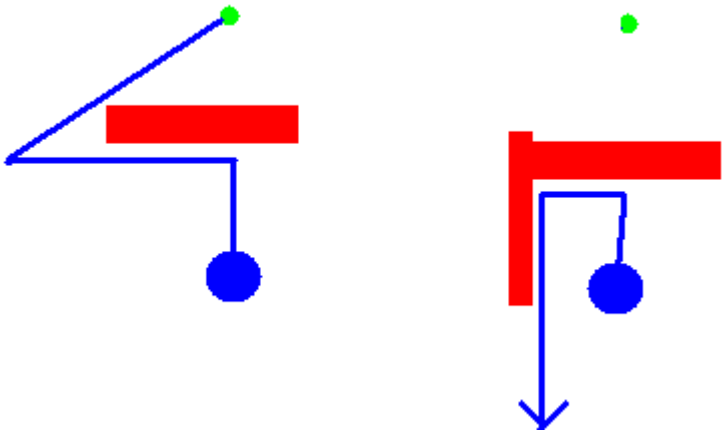


Abbildung 4: Der Wächter läuft in die Unendlichkeit

Also brauchen wir eine ganz andere Methode. Was tun wir, wenn wir den Weg nicht kennen? Wir haben entweder einen Stadtplan oder fragen jemanden, der sich auskennt. Es gibt auch die Möglichkeit, durch die Gegend zu laufen und sich selbst einen Stadtplan zu zeichnen, bzw. sich Zeichen zu machen, wo man schon einmal war. Nun, ich habe mich für die Möglichkeit entschieden, jemanden zu fragen, der gerade in der Nähe herumsteht. Diese (unsichtbaren) Personen werden vom Leveldesigner gesetzt.

Wegpunkte

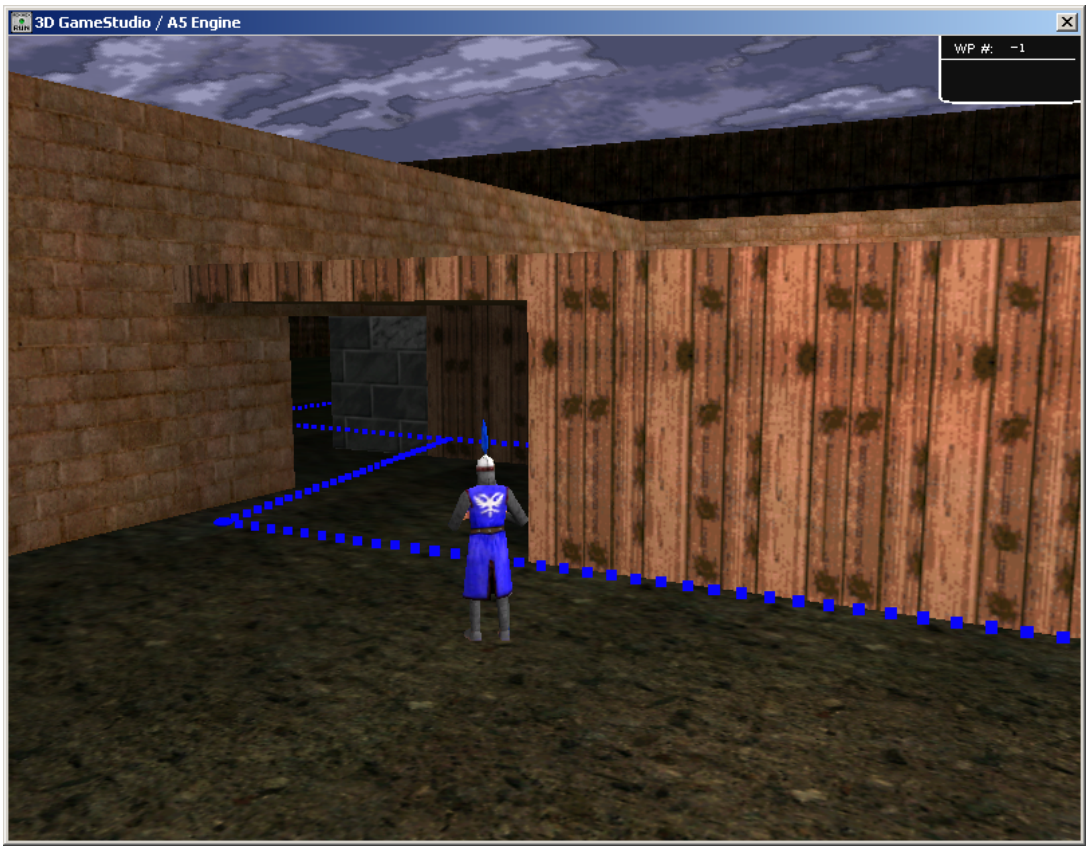


Abbildung 5: Die Wegpunkte

Vom Leveldesigner werden einige unsichtbare Punkte auf der Karte gesetzt – Wegpunkte. Anschließend werden diese mit (eigentlich auch unsichtbaren) Linien verbunden – Pfade. In jedem Wegpunkt ist die Information enthalten, wohin man von diesem Wegpunkt aus gelangen kann. Jeder Wegpunkt hat folgende Informationen gespeichert:

- eigene Nummer
- Typ (Boden oder Luft)
- Position (X, Y und Z)
- Nummern der erreichbaren Wegpunkte

Nun haben wir ein Netz aus Wegpunkten und Pfaden. Wie aber kann der Wächter nun wissen, in welche Richtung er gehen muss?
Zunächst wird der sichtbare Wegpunkt, der dem Wächter am nächsten ist, als Startwegpunkt festgelegt. Der sichtbare Wegpunkt, der dem Ziel (der Kamera) am nächsten ist, wird als Zielwegpunkt festgelegt. Was nun? Wir haben Startwegpunkt und Zielwegpunkt. Was wir brauchen, ist die Richtung, in die der Wächter als nächstes laufen soll. Zunächst läuft der Wächter also in Richtung Startwegpunkt. Ist er dort angekommen, muss er wissen, welcher Wegpunkt der nächste auf dem Weg zum Ziel ist. Dabei muss er zunächst in Gedanken auf dem „Stadtplan“ den Weg ermitteln.

SearchWay

Die Abbildung rechts ist ein Beispiel für die Wegsuche. Wir starten bei Wegpunkt 1 und müssen den nächsten Punkt auf dem Weg zu Punkt 9 finden. Dies ist entweder 2 oder 7. Beide führen zum Ziel, aber der Wächter soll auf dem kürzesten Weg zum Ziel finden!

Zunächst einige Vorbereitungen. Auf der Suche nach dem Weg müssen wir speichern, ob wir einen Wegpunkt schon einmal überprüft haben und nach welcher zurückgelegten Distanz wir das getan haben. Zunächst setzen wir bei jedem Wegpunkt diesen Wert auf 999999, um zu zeigen, dass wir diesen Punkt noch prüfen müssen.

Für die Suche nach dem Weg habe ich die Funktion „SearchWay“ programmiert. Diese Funktion ist rekursiv, d.h. sie ruft sich selbst auf. Diese Funktion erwartet 2 Parameter: die Nummer des Wegpunktes, der untersucht werden soll und die Distanz, die bisher zurück gelegt wurde.

Was tut diese Funktion?

Zunächst prüft sie, ob der Wegpunkt, den sie untersuchen soll, bereits untersucht wurde. Nur wenn der Weg, der jetzt untersucht wird, kürzer ist als alle bisher untersuchten Wege, wird er erneut untersucht.

Dann wird geprüft, ob das Ziel direkt erreichbar ist. Wenn ja, wird die Entfernung bis zum Ziel zurückgegeben.

Ist das Ziel nicht erreichbar, so müssen alle erreichbaren Wegpunkte geprüft werden. Im Beispiel wird zunächst die 2 geprüft. Dazu ruft sich die Funktion SearchWay einfach selbst auf.

Wenn das Ziel erreichbar ist, so wird dies als „Bester Weg“ zwischengemerkt.

Danach wiederholt sich die Überprüfung mit Wegpunkt Nr. 7

Wenn das Ziel erreicht wurde, so wird die dafür benötigte Entfernung zurückgegeben.

Dadurch, dass sich diese Funktion immer selbst aufruft, haben wir schließlich als Rückgabewert die Entfernung zum Ziel und WP_Next ist auf den Wegpunkt gesetzt, zu dem der Wächter direkt laufen muss. Ist er dort angekommen, wird der Weg erneut gesucht.

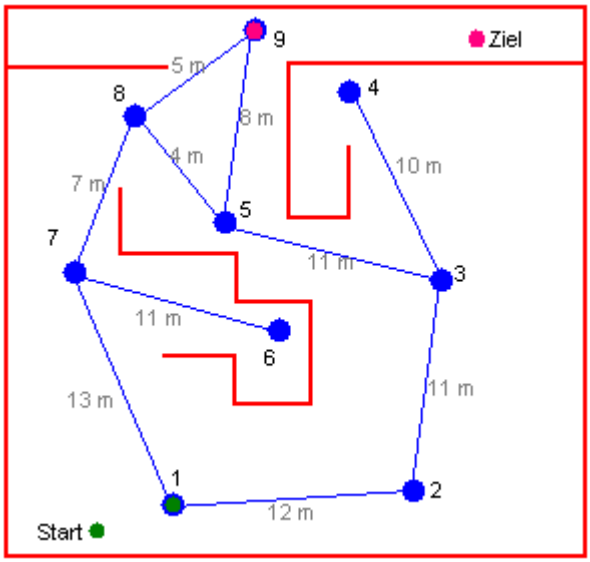


Abbildung 6: Der „Stadtplan“

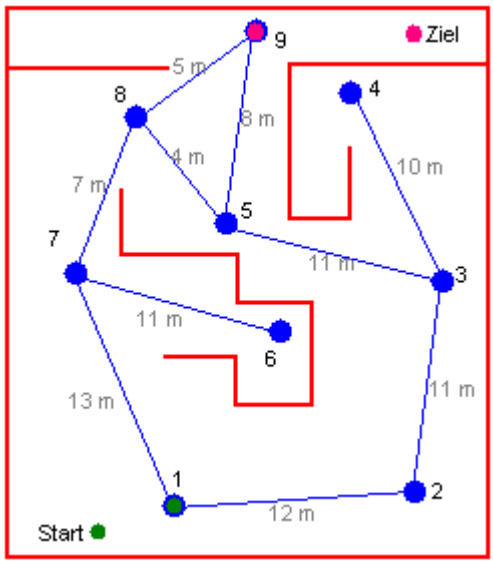
```
Var WP_Next;
function searchWay(wp,dist)
{
  if (wp_vis[wp]<dist){return(-1);}
  wp_vis[wp]=dist;
  var localTMP;
  var localResult=0;
  var localBest=999999;
  var localBestWP=0;
  for (localTMP=1;localTMP<=
      wp.wp_seen; localTMP+=1)
  {
    if (ReadSeen(localTMP)==wp_dest)
    {
      WP_Next=WP_Dest;
      return(dist+
        vec_dist(wp.x,wp_dest.x));
    }
  }
  for (localTMP=1;localTMP<=
      wp.wp_seen; localTMP+=1)
  {
    entSeen=ReadSeen(localTMP);
    localResult=searchWay(entSeen,
      dist+vec_dist(wp.x,entSeen.x));
    if (localResult>=0){
      if (localResult<localBest){
        localBest=localResult;
        localBestWP=entSeen;
      }
    }
  }
  if (localBest==999999){return(-1);}
  WP_Next=localBestWP;
  return(localBest);
}
```

HINWEIS: Dieser Code ist nicht vollständig, einige spezielle 3D-Gamestudio-Funktionen (wie z.B. `ptr_for_handle`) wurden weggelassen.

Programmierung einer 3D-Wegsuchfunktion

Hier ist der Ablauf einer Wegsuche von 1 nach 9:
Bei einem Funktionsaufruf ist der erste Parameter die Nummer des Wegpunktes, der 2. Parameter ist die bisher zurückgelegte Entfernung.

Funktionsaufruf mit (1, 0m)
Noch nicht besucht, Ziel nicht sichtbar
Funktionsaufruf mit (2, 12m)
Noch nicht besucht, Ziel nicht sichtbar
(1, 24m) Schon besucht, zurück zu 2
(3, 23m) Noch nicht besucht, Ziel nicht sichtbar
(2, 34m) Schon besucht, zurück zu 3
(4, 33m) Noch nicht besucht, Ziel nicht sichtbar
(3, 43m) Schon besucht, zurück zu 4
Ziel nicht erreichbar, zurück zu 3
(5, 34m) Noch nicht besucht, Ziel sichtbar
Gebe 42 m zurück an 3.
Beste Möglichkeit: 5 mit 42 m. Zurück an 2
Beste Möglichkeit: 3 mit 42 m. Zurück an 1
Funktionsaufruf mit (7, 13m)
Noch nicht besucht, Ziel nicht sichtbar
(1, 26m) Schon besucht, zurück an 7
(6, 24m) Noch nicht besucht, Ziel nicht sichtbar
(7, 35m) Schon besucht, zurück an 6
Ziel nicht erreichbar, zurück zu 7
(8, 20m) Noch nicht besucht, Ziel sichtbar
Gebe 25m zurück an 7
Beste Möglichkeit: 8 mit 25m. Zurück an 1
Beste Möglichkeit: 7 mit 25m. Zurück an Wächter



Nun kennt der Wächter den ersten Punkt des besten Weges: 7
Diesem Punkt dreht er sich zu und läuft vorwärts, bis er angekommen ist. Dann ruft er die Funktion erneut auf – diesmal mit den Startwerten (7, 0m) So erhält er den 2. Punkt. Ein Neu-Aufruf wäre hier zwar nicht nötig, da der Weg vorher schon komplett ermittelt wurde, aber da der Weg immer neu berechnet wird, kann der Wächter auch auf sich verschiebende Wände reagieren. Öffnet sich direkt neben ihm eine Abkürzung, so benutzt er diese.

Wegpunkttypen

Es gibt verschiedene Wegpunkt-Typen. Ein Wegpunkt kann nicht mit einem Wegpunkt eines anderen Typs verbunden werden. Bisher verwende ich die Typen Luft und Boden. Es gibt natürlich viele andere Verwendungsmöglichkeiten (auf dem Wasser, unter Wasser)

Der Wegpunkteditor

Die Wegpunkte können in Benutzung verändert werden. Eine Tür kann z.B. einen Pfad trennen, wenn sie sich schließt. Die Engine muss mit dem Startparameter –d wpedit gestartet werden, damit die Wegpunkte editiert werden können.
Mit der Konsolenanweisung wp_show() werden die Wegpunkte und Pfade angezeigt. Dann kann der Benutzer mit der Tastatur Wegpunkte setzen und entfernen. In der rechten, oberen Bildschirmecke wird die Nummer des Wegpunkts, bei dem die Kamera sich befindet, angezeigt.
Mit der Tastatur kann man Wegpunkte erzeugen und löschen. Wegpunkte können mit Pfaden verbunden werden und die Pfade können auch wieder getrennt werden.

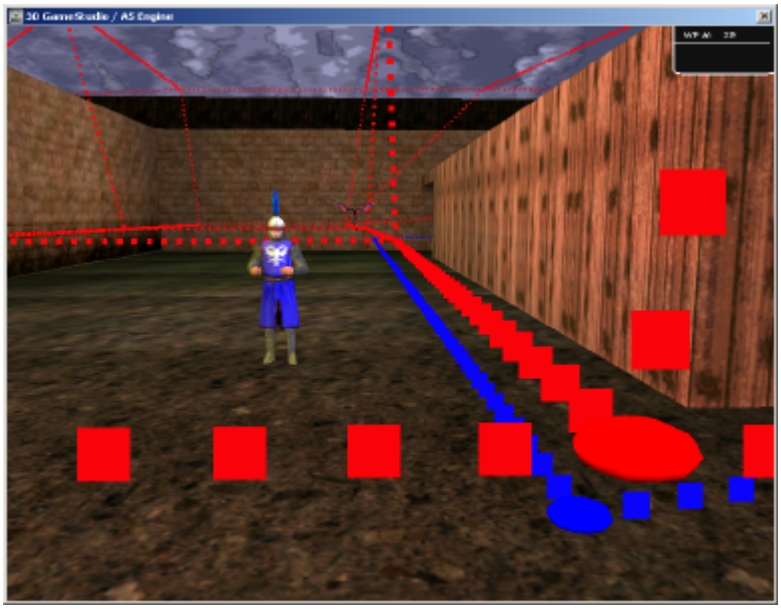


Abbildung 8: Der Wegpunkteditor

Rot: Flugwegpunkte + Pfade
Blau: Bodenwegpunkte + Pfade

Abbildungsverzeichnis:

Abbildung 1: Aufbau des 3D-Gamestudios....2
Abbildung 2: Der Level-Editor....2
Abbildung 3: Der Wächter läuft zur Kamera....4
Abbildung 4: Der Wächter läuft in die Unendlichkeit....5
Abbildung 5: Die Wegpunkte....6
Abbildung 9: Der „Stadtplan“....7
Abbildung 10: Der „Stadtplan“ mit Entfernungen....8
Abbildung 11: Der Wegpunkteditor....9

Mein Projekt im Internet: http://www.htwc.de/daniel/jufo/2003/pathfinding.php oder unter www.danielgrunwald.de

Daniel Grunwald

Daniel Grunwald
Bierberger Str. 9
31174 Schellerten

daniel@danielgrunwald.de
www.danielgrunwald.de