Projekt „Monstergame“

Projektdokumentation

Philipp Fehr und Lukas Bischof
PkN

Inhaltsverzeichnis

Einleitung

Philipp Fehr und Lukas Bischof haben in dem Projekt „Monstergame“ versucht, den Dijkstra-Algorithmus in einen eigenen zu verpacken, damit man damit eine künstliche Intelligenz für ein einfaches Spiel mit Monstern und einem Spieler zu erstellen, in dem das Ziel ist, dass der Spieler die Monster mit Schüssen besiegen muss, oder eben von den Monstern selbst durch Schüsse besiegt wird.
Während der Entwicklung wurde das Projekt zusätzlich von Herrn Tromsdorff und Herrn Veselcic betreut.

Das Problem

Das zentrale Problem des Projekt ist die Frage, wie die Monster den effizientesten, bzw. kürzesten Weg, zum Spieler finden, damit sie dann freie Schussbahn auf ihn haben. Der ideale Algorithmus für dieses Problem würde jeden Punkt im Raum in Betracht ziehen, welcher nicht durch ein Hindernis verdeckt ist, und dann von der Menge an Punkten jener heraussuchen, welcher eine freie Schussbahn zum Spieler bietet und das beste Verhältnis zwischen grösstmöglichem Winkel, in dem das Monster noch zu treffen ist, und kürzesten Pfad vom Monster zum Punkt bietet.

Verwendete Lösung

Einleitung

Da der ideale Algorithmus ziemlich komplex wäre und auch eine relativ grosse Rechenleistung voraussetzt, ignorierten wir die Selektion der möglichen Punkte nach dem Argument des grösstmöglichen Öffnungswinkels, indem sich der Spieler maximal bewegen kann und immer noch vom Monster getroffen werden kann. Darum entwickelten wir einen Algorithmus, welcher erstmals eine Auswahl an möglichen Punkten sucht und dann per Dijkstra den kürzesten Weg zu einem „abschuss-tauglichen“ Punkt findet.

Funktionsweise Generierung des Graphen

Das quadratische Feld besteht aus vielen kleinen einzelnen Quadraten. Ein Quadrat, oder wir im Code auch „Quad“ genannt, kann dann entweder leer, oder ein Hindernis sein. Der Algorithmus geht dann durch alle Felder, welche ein Hindernis darstellen, durch und schreibt die Punkte an den Ecken jedes Quadrats, insofern diese nicht auch wieder in einem weiteren Hindernis sind, in einen Graphen als alleinstehender Knoten (Der Graph hat bis dahin noch keine Knoten und auch noch keine Kanten). Danach wird jeder Knoten mit allen Knoten verbunden, insofern auf der Strecke zwischen Punkt 1 und Punkt 2 kein Hindernis liegt, sodass ein Graph mit allen möglichst kurzen Wegen zu allen möglichen Hindernis-Ecken entsteht.

Wenn diese Vorbereitung abgeschlossen ist, wird nun für jedes Frame, bzw. jeder Berechnungsaufruf, das Monster und der Spieler in den Graphen

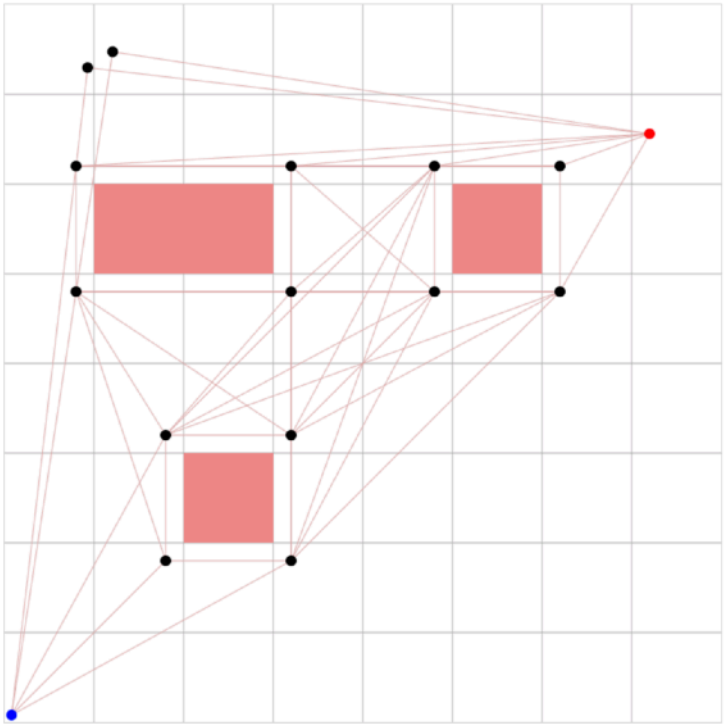


Abb. 1: Exemplarisches Resultat des Generierungsalgorithmus des Graphen. Rot: Monster, Blau: Spieler

eingebunden und mit allen Punkten verbunden, die zwischen sich selbst und dem Monster/ Spieler kein Hindernis stehen haben.

Da das Monster nun aber auch von einem „freistehenden“ Punkt aus schiessen kann, der nicht direkt an eine Ecke gebunden ist, werden nun Geraden (g_1) an alle mit dem Player verbundenen Nodes angelegt. Ist dies erledigt, wird Wiederrum über alle anderen Punkte iteriert und mehrere Geraden (g_2), welche durch den Punkt gehen und senkrecht den anderen erzeugen Geraden (g_1) steht, auf Scheidungen mit Hindernissen geprüft. Falls nun eine generierte Gerade (g_2) kein Hindernis schneidet, wird nun der Schnittpunkt der Geraden (g_2) mit der „alten“ Geraden (g_1) in den Graphen aufgenommen und direkt mit dem Player, sowie allen direkt erreichbaren Nodes im Graphen, verbunden.

Danach ist der Graph komplett und kann dem Dijkstra-Algorithmus übergeben werden, der dann den Zielpunkt für das Monster zurück gibt.

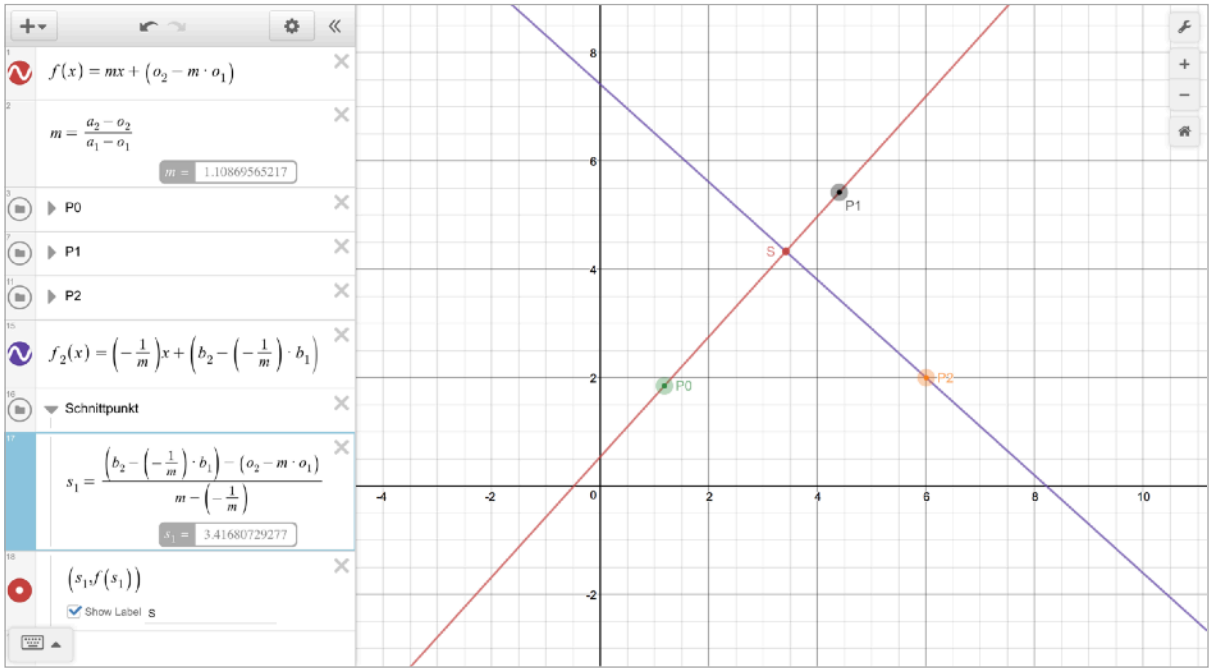


Abb. 2: Die verwendeten mathematischen Funktionen um den Schnittpunkt zwischen zwei Geraden zu erhalten, sowie die Bildung einer linearen Funktion, die rechtwinklig zu einer anderen *f(x)* Funktion ist und durch einen Punkt läuft. *f(x)* ist dabei die Hauptfunktion und $f_2(x)$ die rechtwinklige Funktion

Funktionsweise des Hindernistests

In der Generation des Graphen wird eine Funktion, welche testet, ob sich zwischen zwei Punkten ein Hindernis befindet, sehr häufig gebraucht und ist deshalb eine der wichtigsten Funktionen im ganzen Algorithmus.

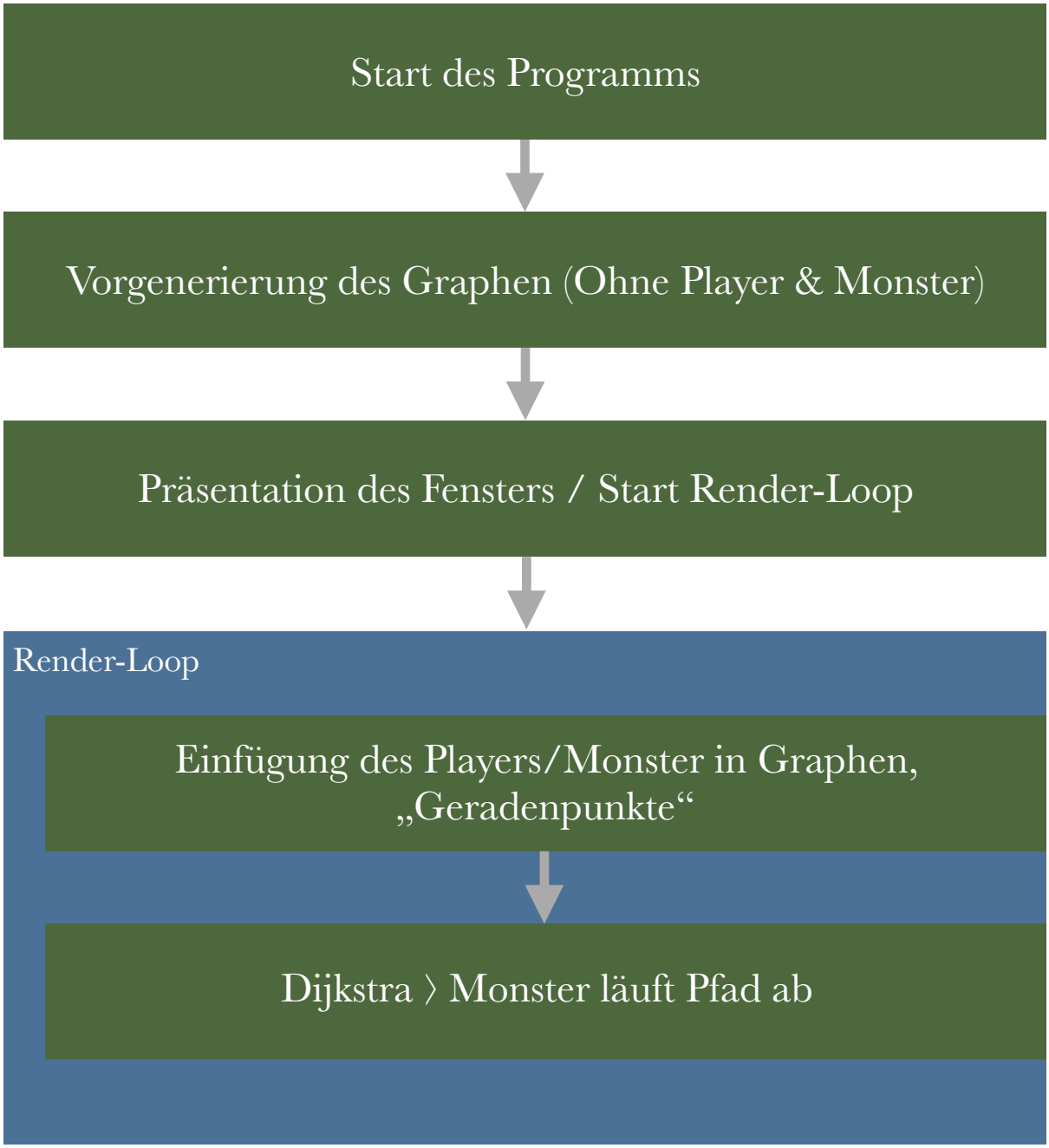
Der Test startet mit einer Überprüfung, ob die Gerade (g_1) zwischen den beiden gegebenen Punkten p_1 und p_2 sich nur in einer Spalte oder Zeile befindet. Wenn dies der Fall ist, dann geht die Funktion einfach alle Felder, welche sich in der Spalte/Zeile zwischen den Punkten p_1 und p_2 befinden durch und testet sie, ob sie ein Hindernis sind oder nicht. Falls ja terminiert die Funktion sofort.

Falls dies nicht der Fall ist, die Gerade g_1 sich also über mehrere Zeilen und/oder Spalten hinwegzieht (diagonal ist), dann wird von der Spalte, in der sich p_1 befindet, bis zur Spalte, in der sich p_2 befindet, durchgegangen und alle Felder, welche die Gerade g_1 in dieser Spalte berührt auf ein Hindernis geprüft. Ebenfalls terminiert die Funktion sofort, falls der Test positiv endet.

Dijkstra Algorithmus

Der Dijkstra Algorithmus sucht in dem generierten Graphen nach dem kürzesten Weg vom Monster zum Spieler. Dafür sucht er immer die Node mit der kürzesten Distanz zum Start – zu Beginn ist dies der Start selber – dann geht er durch alle Connections der Node. Bei jeder Connection schaut er, ob die vorhandene Länge länger als seine Länge plus die Distanz zu der Node ist. Wenn dem so ist, dann setzt er die Länge auf die kürzere. Nachdem alle Connections überprüft wurden beginnt der Prozess wieder von vorne. Wenn der Prozess durch alle Nodes gegangen ist, gibt er den idealen Pfad zurück.

Ablauf des Programms



Verwendete Datenstrukturen

Graph

Der Graph wird mithilfe einer Graphenklasse gespeichert. Diese besteht aus einer Liste mit allen Knoten und einer Root-Node, sowie eine End-Node. Die Klasse bietet zudem einige Funktionen wie `addNode` oder `removeNode`, sowie eine Cloning-Methode.

Node

Die Nodes werden auch mit einer Klasse gespeichert. Die Klasse „Node“ besitzt einen relativen Punkt im Feld, sowie mehrere Kanten, in dem Code auch Verbindung „Connection“ genannt.

Connection – Kante

Eine Connection wird auch durch eine Klasse dargestellt. Sie beinhaltet zwei Weak-References auf Nodes, sowie eine Länge, die aber dynamisch generiert wird.

Zudem bietet sie weitere Methoden wie „`getCounterpart`“, welche den „Gegenspieler“ der mitgegebenen Node in der Verbindung zurückgibt.

Das Feld

Das Feld wird in dem Projekt nicht Feld genannt, da es eigentlich kein Feld in der Logik gibt, sondern nur eine Ansammlung von einzelnen Quadraten - „Quads“, welche von dem QuadController verwaltet wird. Der QuadController bietet dabei wichtige Funktionen wie das Speichern des Feldes, sowie Funktionen, welche quad-bezogen sind, wie bspw. den Hindernistest.

Das Spiel

Der zentrale Knotenpunkt zwischen Logik und Präsentation bildet dabei die Klasse „Game“. Sie hält alle spielrelevanten Daten, wie einen QuadController, sowie den Spieler und das Monster. Zudem ruft die Präsentation die Methode `getPathForMonster` zu einem passenden Zeitpunkt auf.

Reflexion

Nachwort Lukas

Am Anfang des Projektes war es relativ schwer, einen Algorithmus zu entwickeln, welcher nicht zu komplex ist, um ihn in einer vernünftigen Zeit zu berechnen, doch trotzdem das Problem brauchbar löst. Als jedoch alles auf Papier geplant war, war es ziemlich einfach, den Algorithmus auch in Code zu verfassen. Jedoch bereitete das Rendering teilweise ein wenig Kopfschmerzen, da Java einfach nicht für dieses Anwendungsgebiet geeignet ist. Der Rest funktionierte relativ gut, nur das Cloning war bei dem Graphen beanspruchte mehr Zeit als gedacht.

Nachwort Philipp

Ich hatte und habe immer noch Probleme genau zu verstehen wie OpenGL funktioniert, ich denke, dass wird sich aber in nächster Zeit noch bessern. Die Implementation von Dijkstra ging sehr einfach und schnell. Die Bewegung des Monsters hat hingegen bis in die letzte Minute Ärger bereitet. Ich habe sehr viel neues gelernt und würde mich freuen, wenn wir dieses Projekt noch ausbauen könnten.