AntIT!

Projektdokumentation

Praktische Programmierung Sommersemester 2019

Ludwig-Maximilians-Universität München

Studiengang Lehramt Gymnasium (modul.)

Fachwissenschaft Informatik, Modul P.15

Verfasser: David Li

Inhaltsverzeichnis

Zielsetzung

Die Digitalisierung macht auch vor dem Bildungsbereich nicht Halt: Die neuen technischen Möglichkeiten werden intensiv ausprobiert, um das Lernen neu zu definieren und für die Lernenden ansprechender zu gestalten. Konzeptionell trifft man auf Schlagworte wie *Serious Gaming* und *Blended Learning*. Serious Gaming verfolgt das Ziel, digitale Spiele über den Unterhaltungswert hinaus zu verwenden, um „ernsthafte“ Themen zu vermitteln. Das Lernen sollte dabei quasi nebenbei passieren und wird durch den spielerischen Ansatz mit positiven Emotionen verknüpft. Blended Learning versucht, die Welt des digitalen Lernens mit der Welt des traditionellerem „Vor-Ort-Lernens“ zu verbinden. Denn auch der soziale Kontakt in direkter Kommunikation ist ein wesentlicher Bestandteil eines guten Lernprozesses. Mit dem Projekt *AntIT!* verfolge ich das Ziel, diese zwei Konzepte auszuführen und im Rahmen einer konkreten Lernplattform zu realisieren. Ich möchte auf die einzelnen Aspekte dieser Zielsetzung nun genauer eingehen und erklären, welche Vorstellungen mir dafür genau vorschweben. Dabei werde ich die Wortbestandteile der zwei Konzepte einzeln durchgehen:

Serious: Grundlegende Konzepte des Programmierens

AntIT! soll dafür ausgelegt sein, Programmieranfängern die grundlegenden Konzepte des Programmierens beizubringen. Die Zielgruppe ist im Bereich der Mittelstufe angesiedelt (ab 8. Jgst). Aus diesem Grund soll auch keine visuelle Programmiersprache wie Scratch oder Snap! eingesetzt werden, sondern es ist erwünscht, dass die Programmierung mit Code im Editor stattfindet. Darüber hinaus soll die Programmierung aber so zugänglich wie möglich sein. Der Verzicht auf objektorientierte Modellierung und statische Typisierung bietet sich an, weil dadurch der Code auf imperative Befehle und dazu ergänzend eine ereignisorientierte Modellierung vereinfacht werden kann. Diese zwei Konzepte erscheinen als gute Kandidaten für den Einstieg: Sie ermöglichen reichhaltige Interaktionen und sind gleichzeitig für Anfänger leicht nachvollziehbar. Außerdem passen sie gut in das thematische Feld, das im nächsten Abschnitt näher vorgestellt wird. Natürlich soll auch nicht aus dem Blick verloren werden, dass im Laufe des Lehrgangs auch

weitere Konzepte hinzukommen, insbesondere bedingte Anweisungen, arithmetische Berechnungen, Zustände und Variablen. Mit diesen Anforderungen rücken Skriptsprachen wie Python oder JavaScript in den Vordergrund oder als Möglichkeit auch eine eigene Miniatursprache. Die Eigenentwicklung einer Sprache wie bei Robot Karol (www.mebis.bayern.de/infoportal/faecher/mint/inf/robot-karol/) soll aber nach Möglichkeit vermieden werden, denn der damit verbundene Aufwand ist hoch. Außerdem erschwert das den Teilnehmenden auch, ihr Vorwissen einzubringen.

Gaming: Spaß an Informatik

Allein die technischen Möglichkeiten einer Programmiersprache sind für viele noch nicht Motivation genug, um sich damit auseinanderzusetzen. Wesentlicher Grundpfeiler von Serious Gaming besteht nun darin, den technischen Aspekt mit einem Thema und einem Anwendungsgebiet zu verbinden, das aus sich heraus einen Unterhaltungswert hat und an dem dann die Inhalte angebunden werden. Dieses Thema ist idealerweise aus der Lebenswelt der Teilnehmenden herausgegriffen und ermöglicht eine emotionale Verbindung. Gleichzeitig soll das Thema auch Anreize für Problemfelder bieten, an denen sich technische Lösungen ausprobieren lassen. Hierzulande bekanntestes Beispiel ist Robot Karol. Das Thema „Roboter“ stößt grundsätzlich auf hohes Interesse und bietet natürlich auch eine Fülle an möglichen Aufgaben. Allerdings ist diese Auswahl durch die Rahmenbedingungen von Robot Karol recht begrenzt. Zum Beispiel existiert zu einer Zeit nur ein einzelner Akteur in der Welt und es finden keine Interaktionen statt. Dadurch entsteht oft der Eindruck, dass Robot Karol „langweilig“ sei. Bei der Themenwahl für AntIT! besteht der Wunsch, mehr „Action“ reinzubringen und damit die Lücke zu aktuellen Computerspielen zu schließen. Zwei Verbesserungen sollen dabei helfen: Einerseits soll eine größere Fülle an Interaktionen erlaubt sein. Eine große Anzahl an Akteuren in verschiedenen Parteien sind gemeinsam in einer Welt unterwegs und befinden sich z.T. auch im Wettstreit. Die Handlungen einzelner Akteure soll damit auch vom Verhalten der anderen Akteure abhängen. Dadurch können Dynamiken entstehen, die auch überraschen können. Außerdem sollen nicht-deterministische Elemente erlaubt. Die Welt darf Zufallselemente enthalten und fordert somit die Akteure auf, dementsprechend ihr Verhalten anzupassen.

Beim Spaß darf natürlich auch die graphische Gestaltung nicht fehlen. Eine ansprechende und zeitgemäße Grafik soll – im Rahmen des Machbaren – vorhanden sein.

Blended: Einsatz in Workshops

Der soziale Aspekt soll beim Lernprozess nicht vernachlässigt werden. Besonders bei intensivem Einsatz von digitalen Medien besteht die Gefahr, dass die einzelnen Lernenden in die Isolation geraten und dadurch das Lernen an Spaß verliert. AntIT! soll gezielt für Gruppen ausgelegt sein, z.B. im Rahmen einer Exkursion an die Universität oder auch im Klassenverband. Innerhalb dieser Gruppe gibt es nun Möglichkeiten zur sozialen Interaktion: In Partnerarbeit können zwei Teilnehmende gemeinsam am Code arbeiten und sich gegenseitig unterstützen. Außerdem können die Teilnehmenden im Rahmen von Turnieren gegeneinander im Wettkampf antreten. Diese Aspekte führen das individuelle E-Learning und traditionellere Gruppendynamiken zusammen. Das wirkt sich auch auf die Rolle des Workshopleiters aus, der sich aus der Rolle des Lehrenden zurückzieht und mehr im Hintergrund als Moderator und Unterstützer den Gruppen Hilfe anbietet.

Learning: Anregung zum Selbstlernen

Als Lernumgebung soll AntIT! auch einen Beitrag leisten zum selbstständigen Lernen. Die Teilnehmenden sollen grundsätzlich in der Lage sein, die Themen des Lehrgangs selbstbestimmt und ihm eigenen Tempo durchzuarbeiten. Auf eine zentrale Vorstellung der Themen soll verzichtet werden, stattdessen soll der Stoff selbstständig erlernt werden und anschließend in Übungsaufgaben ausprobiert werden können. Eine gute Gestaltung dieses Lernpfads ist ein wesentliches Ziel von AntIT!, ebenso eine klare Strukturierung der einzelnen Lernabschnitte. Eine Leistungsdifferenzierung soll stattfinden: Über ein Basisniveau hinaus bieten zusätzliche Aufgaben den Leistungsstärkeren Anreize zur vertieften Mitarbeit. Die Aufgaben sollen direkt am Computer bearbeitet werden können und die Teilnehmenden sollen eine schnelle und direkte Rückmeldung dazu erhalten.

Planung

Auftraggeber für AntIT! ist die Technische Universität München. Im Mai 2016 beginne ich dort eine Stelle als wissenschaftliche Hilfskraft an der Fakultät für Informatik, Schnittstelle Schule-Universität und werde bald mit der Entwicklung dieses Projekts beauftragt. Ein konkretes Ziel gibt es auch: Im Sommer soll ein dreitägiger Workshop im Rahmen des Programms „Mädchen machen Technik“ (www.schueler.tum.de/ferienprogramm) mit dieser Software laufen, den ich auch selber gestalten werde. Damit ist grob ein Zeitraum von drei Monaten vorgegeben, in dem die erste Phase der Entwicklung stattfinden soll.

Zur Ausarbeitung des Projekts wird mir das Programm AntMe! als Vorlage gestellt. AntMe! ist eine Ameisensimulation, bei der man mithilfe von C#-Code das Verhalten eines Ameisenvolks steuert und damit dann Nahrungsmittel wie Zucker oder Äpfel sammelt und gegen Wanzen und feindliche Ameisen kämpft. Über Duftmarken können die Ameisen miteinander kommunizieren und damit z.B. Ameisenstraßen aufbauen. Der Code wird in einem Editor wie Visual Studio geschrieben und dann kompiliert. Die graphische Oberfläche besteht aus einer GUI und einer 3D-Simulation, in der sich die Ameisen bewegen.

Diese Vorlage erfüllt einige Aspekte meiner Zielsetzung sehr gut. Vor allem, was die thematische Ausarbeitung angeht, gefällt mir das Konzept mit den Ameisen sehr. Die große Ansatz an Akteure erzeugt eine Vielzahl an Interaktionen, die interessant zu beobachten sind. Ein Zufallselement macht jede einzelne Simulation spannend. Außerdem ist die Graphik ansprechend. Kurz: AntMe! schafft es gut, faszinierende Momente zu schaffen, die ähnlich wie bei einem Computerspiel eine große Motivation auslösen.

Leider hat AntMe! auch ein paar Nachteile: Die Notwendigkeit für eine komplette Softwareentwicklungsumgebung macht die Installation sehr aufwendig und stellt eine Hürde für den Workshop im Sommer dar. Außerdem entspricht die Wahl der Programmiersprache nicht meiner Zielsetzung, denn C# ist eine statisch typisierte, objektorientierte Sprache. Außerdem ist die API nicht ereignisorientiert oder imperativ, sondern zustandsorientiert und macht es schwer, größere Abläufe zu

implementieren. Zu diesem Aspekten hinzu fehlt für den Workshop die Möglichkeit, online Ameisenvölker gegeneinander kämpfen zu lassen. Es gibt zum selbstständigen Lernen einzelne Tutorials, doch diese bieten keine große Vielfalt an Möglichkeiten und sind auf die wesentlichen Aufgaben beschränkt.

Damit fällt die Entscheidung, die Vorlage neu zu implementieren. Die größte Änderung wird dabei sein, dass das neue Projekt im Browser läuft: Sowohl der Code-Editor wie auch die Simulation können aus einem Internetbrowser bedient werden. Durch einen gemeinsamen Server können die Teilnehmenden auch online ihre Ameisen gegeneinander antreten lassen. Als geeignete Sprache für die neue Ameisen-API kristallisiert sich schnell JavaScript heraus, weil sie eben eng an den Browser gebunden ist. Mit der Möglichkeit eines Server kann auch eine ganze E-Learning-Umgebung implementiert werden, die durch die Neuentwicklung auch sehr eng mit der Simulation und dem Editor verzahnt werden kann.

Weil der Zeitplan bis zum ersten Workshop nicht ausreicht, alle Ziele zu erfüllen, wird das Projekt in drei Phasen unterteilt. In der ersten Phase soll es darum gehen, die Simulation, den Editor und den Server zu implementieren. Damit sind die Grundlagen für einen Workshop geschaffen, das Learning sollte dann erst mal in Frontalunterricht stattfinden. In der zweiten Phase wird angedacht, die Simulation und die Programmiersprache zu verfeinern und je nach Feedback neue Funktionen einzubauen bzw. anzupassen. Wenn der Funktionsumfang schließlich feststeht, soll in der dritten Phase begonnen werden, das Tutorialsystem mit der E-Learning-Umgebung zu entwickeln.

Als weitgehendes Ein-Mann-Projekt kann ich einen sehr agilen Entwicklungsstil verfolgen: Viele Prototypen, viele Änderungen und Anpassungen, schnelle Iterationen. Innerhalb kurzer Zeit entstehen funktionierende Programme und ich erschließe viele neue Technologien.

Durchführung

1. Phase: Grundlagen

Im Mittelpunkt meiner anfänglichen Arbeit steht die Implementation der 3D-Ansicht im Browser. Mit WebGL liegt eine Technologie vor, die es ermöglicht, selbst komplexe Szenen mit Hardwarebeschleunigung im Browser zu rendern. Anstatt direkt auf die Schnittstelle zuzugreifen, nutze ich die Bibliothek three.js (threejs.org), die eine Abstraktionsschicht über WebGL bereitstellt. Diese Bibliothek ermöglicht es mir, mit vergleichsweise wenig Aufwand die nötigen Elemente darzustellen. Die Unterstützung im Browser ist im Jahr 2016 ziemlich vollständig, auch im mobilen Bereich. Das bestärkt mich weiter, diesen Ansatz weiterzuverfolgen.

Die 3D-Modelle kann ich von AntMe! importieren, diese sind unter einer freien Lizenz verfügbar. Schnell ist es auch geschafft, die Modelle im Browser anzuzeigen und zu betrachten. Nun geht es daran, die Simulation der Ameisen zu definieren und zu implementieren. Das stellt sich als schwieriger heraus, als anfangs angenommen. Durch vielfältige Interaktionsmöglichkeiten erreicht der Code sehr schnell eine hohe Komplexität und gewinnt an Umfang.

Die Entwicklung der API für die Benutzer ist ein Prozess, der sich über zwei Jahre hinweg erstreckt und viele Varianten und Versuche beinhaltet. Eine zu mächtige API höhlt das didaktische Potenzial aus, weil dann keine Herausforderungen mehr bestehen. Eine zu stark vereinfachte API wiederum macht die Arbeit frustrierend und senkt die Motivation. Das geeignete Niveau abzuschätzen, vor allem für Kinder und Jugendliche, ist ohne praktisches Probieren kaum möglich. Während der Entwicklung finden immer wieder Workshops statt, an denen ich ausgiebig Feedback einholen kann und entsprechend die API anpasse.

Eine grundsätzliche Idee hat sich von Anfang an bewährt: Die API der Ameisen ist ereignisorientiert und definiert Ereignisse, wie z.B. „IstUntätig“ oder „SiehtApfel“, an denen das Verhalten für die Ameise festgelegt werden kann. Durch dieses Modell können sich wiederholende Verhaltensmuster ohne die syntaktische Komplexität von Schleifen realisiert werden. Außerdem bieten die Ameisen eine Warteschlange

für Befehle an: Innerhalb eines Ereignisse können der Ameise eine Reihe von Befehlen gegeben werden und diese werden nacheinander abgearbeitet. Damit können Muster gelaufen werden oder gezielt Orte besucht werden. Diese Art der Programmierung hat sich als sehr zugänglich erwiesen. Das konkrete Ergebnis der API wird im nächsten Teil der Dokumentation behandelt.

Es bleibt noch die Aufgabe, einen Server aufzusetzen. Zu diesem Zeitpunkt bin ich nur mit der Programmiersprache PHP vertraut. Diese Sprache erscheint mir für eine Web-Applikation dieser Form nicht geeignet, zumal ich auch gerne auf diesem Gebiet eine neue Technologie kennenlernen möchte. Dadurch fällt die Wahl auf den Einsatz von serverseitigen JavaScript. Als Server wird node.js (nodejs.org) verwendet, als Datenbank kommt mongoDB (mongodb.com) zum Einsatz.

Auch für node.js existieren Bibliotheken, die das Schreiben einer Web-Applikation erleichtern. Meine Wahl fällt auf Express.js (expressjs.com), welches wieder als Abstraktionsschicht über die primitiven Funktionen von node.js fungiert. Mit dem Konzept von Routes und Middlewares mache ich mich schnell vertraut und erlebe es als sehr großen Fortschritt zu php und mod_rewrite. Für das Front-End nutze ich die einfache Templating-Engine ejs (ejs.co). Ich zeige durchaus eine Vorliebe für schlichte Abstraktionsschichten, die mir die Arbeit erleichtern, aber möglichst selber keine zu engen Vorgaben und Modelle setzen. In diesem Sinne gefällt mir diese Struktur gut.

Das Einarbeiten erfolgt mit viel Versuch und Irrtum. Schnell stehen die ersten Seiten, die Datenbank wird angebunden und Benutzer können registriert werden und können sich einloggen. Die Notwendigkeit für Callbacks macht den Code aber sehr verschachtelt und schwer zu schreiben. Erst mit den Umstieg auf Promises und Generatoren kann hier eine wirkliche Erleichterung geschaffen werden.

Insgesamt kann ich den Zeitplan einhalten und im Sommer den ersten Workshop mit AntIT! abhalten.

2. Phase: Erweiterungen

Mit dem Ergebnis des ersten Workshops bin ich zufrieden. Den Teilnehmenden macht das Programmieren Spaß. Nun gilt es, die letzten Schönheitsfehler zu beseitigen. Der größte ist, dass die API noch nicht sauber funktioniert. Befehle scheinen nicht ausgeführt zu werden und komplexere Ameisen scheitern daran. Durch die hohe Interaktion erweist sich die Analyse als schwierig, zumal ich auch nicht genau weiß, an welcher Komponente dieser Fehler genau liegt. Schließlich gelingt es mir nach einem halben Jahr das Problem ausfinden zu machen und zu korrigieren.

Bei der Verteilung der Nahrungsmittel auf dem Spielfeld gibt es etwas Kritik: Diese ist nicht sehr gut ausgeglichen und führt dazu, dass eigentlich gleichwertige Ameisenvölker zu sehr unterschiedlichen Punktzahlen kommen, weil eines davon mehr Nahrungsmittel erhalten als das andere Volk. Hier entwickle ich einen Code, der die Ausgewogenheit der Verteilung beobachtet und korrigiert.

In dieser Phase kommt auch der Wunsch auf, eine Simulation außerhalb der 3D-Ansicht laufen zu lassen. Aufgrund einiger Kopplungen zwischen Simulation und Ansicht ist das nicht sofort umsetzbar. Die Idee steht im Raum, alles nochmal neu zu schreiben. Nach drei Wochen, wo ich versuche, die gesamte Simulation neu zu schreiben, gebe ich dieses Vorhaben auf. Einige gute Ideen kann ich aber übernehmen und erhalte damit die gewünschte Trennung zwischen Modell und Ansicht. Es folgen weitere Workshops. Der Kern der API beginnt sich zu stabilisieren, obwohl ich hier immer noch weitere Experimente mache.

Bislang hoste ich den Dienst bei einem SaaS-Provider. Dieser kündigt aber an, seinen kostenlosen Dienst einzustellen. Es folgt ein Umzug auf einen kostenpflichtigen Hoster. Parallel dazu hole ich mir eine eigene Domain und schalte die Website zeitgemäß auf HTTPS. Auf Entwicklungsseite ist eine einfache Form des automatischen Deployments implementiert: Nach einem Git-Push wird automatisch der Server neugestartet und mit der neusten Version aktualisiert.

3. Phase: E-Learning

Nach zwei Jahren beginnt nun die dritte Phase der Entwicklung: AntIT! soll zu einer autodidaktischen Plattform wachsen. Als Idee denke ich an, zusätzlich zur Arena-Simulation einzelne Szenarien zu entwickeln, an denen die Teilnehmenden gezielt Fähigkeiten üben können. Begleitet von Tutorials soll ein selbstständiges Erlernen der API möglich sein. Mit der Umsetzung dieses Systems verbringe ich das letzte Jahr der Entwicklung.

Die Simulation an sich ist sehr modular aufgebaut, weshalb es leicht ist, an den Parametern und Optionen Änderungen vorzunehmen. Ich beginne nun, einzelne Aufgaben zu implementieren, von sehr niedriger Schwierigkeit bis zu sehr hoher Schwierigkeit. Der Aufwand für die Implementation hält sich in Grenzen und ist vergleichsweise gering zu der Arbeit, die bei der didaktischen Gestaltung nötig ist! Auch hier ist eine regelmäßige Evaluation in Workshops unabdingbar. Es sind insgesamt neun Stufen verfügbar, in die man durch Lösen von Aufgaben aufsteigen kann. Jede Stufe schaltet neue Aufgaben frei, Tutorials erklären die neuen Möglichkeiten und Programmiertechniken.

Es gibt weiterhin Erweiterungen an der Plattform: Die Ameisen können nun mit Gift gegeneinander kämpfen. Auf dem Spielfeld lässt sich ein Koordinatengitter anzeigen und der Codeeditor erhält einen Linter, mit dem Klammerfehler schneller erkannt werden können. Auf Serverseite werden nun alle Sessions in der Datenbank gespeichert, so dass ein Serverneustart ohne jeglichen Datenverlust funktioniert. Es wird auch ein Bug gefixt, der den Server zum Absturz bringen konnte.

Schließlich wird die Plattform öffentlich gemacht. Das macht eine Auseinandersetzung mit Datenschutz und Impressum notwendig. Beides ist nun auf der Website vorhanden und Besucher können sich registrieren und die Szenarien durchspielen. Damit erreicht die Entwicklung ihr vorläufiges Ende. Ich verlasse Ende März 2019 meine Arbeitsstelle an der TUM. Ein Nachfolger ist bereits gefunden, der auch weiterhin dieses Projekt an der TUM betreut. Im Sommer 2019 sind bereits wieder mehrere Workshops mit AntIT! geplant.

Herausforderungen

Bei der Entwicklung von AntIT! gab es auch viele Dinge, die herausfordernd waren. Meist gab es für die Probleme keine einfache Lösung und es brauchte eine längere Suche, bis eine gute Antwort gefunden wurde. Manchmal unterschätzte ich auch einfach den Aufwand für gewisse Aufgaben. An einigen exemplarischen Beispielen möchte ich zeigen, wie ich mit diesen Situationen umgegangen bin und zu welcher Lösung ich schließlich gelangt bin.

Modularisierung in JavaScript

Die Skriptsprache JavaScript, die bei der Programmierung der Simulation zum Einsatz kommt, ist für kleine Anwendungen im Browser ausgelegt und bietet daher von sich aus keine explizite Möglichkeit, Code in Module und Namensräume zu strukturieren. Beim Zeitpunkt der Entwicklung waren Lösungen wie ES6-Module noch nicht verbreitet und daher aufgrund der Kompatibilität nicht einsetzbar. Bei einer Codebasis von 1500 Zeilen Code ist es sehr deutlich nötig, den Code zu modularisieren. Zusätzlich besteht der Wunsch, die Simulation nach außen gegen Manipulation abzusichern. Auch dazu ist ein Modulsystem hilfreich.

In mehreren Schritten kann trotz der fehlenden Unterstützung ein ausreichend gutes Modulsystem in Javascript implementiert werden. Dabei werden IIFE (Immediately-Invoked Function Expression) eingesetzt. Das Grundgerüst eines solchen Ausdrucks sieht so aus:

```
(function(){
      // Code
}())
```

Durch die Closure werden alle Variablen, die innerhalb der IIFE definiert werden, nach außen hin abgeschattet. Dadurch ist eine erste Modularisierung erreicht. Um Variablen nach außen zu exportieren können diese auf eine globale Variable gelegt werden. Problematisch ist noch, dass eine einzelne IIFE nicht über mehrere Dateien verteilt werden kann und daher ein Namensraum komplett in einer einzigen Datei definiert werden müssen. Diese Beschränkung kann umgangen werden, wenn eine temporäre globale Variable verwendet wird, um den Namensraum zu definieren:

```
// Erste Datei
var _antit = {}

// Datei A
(function(AntIT){
      AntIT.Ants = []
}(_antit))

// Datei B
(function(AntIT){
      console.log(AntIT.Ants.length)
}(_antit))

// weitere Dateien ...

// Letzte Datei
delete window._antit
```

Mithilfe dieser Technik gelingt es, einen Namensraum zu verwalten und diesen nur in bestimmten Dateien zugänglich zu machen.

Callback Hell und Async Heaven

Auf der Serverseite stellt eine andere Eigenschaft von Javascript eine Herausforderung dar: Der Interpreter verfügt über kein Multithreading und läuft immer nur in einem Thread. Das hat den Vorteil, das keine Probleme der Synchronisation auftreten können – gleichzeitig erschwert es aber die Programmierung asynchronen Codes. Dieser muss jeden Schritt der Berechnung in eine eigene Funktion packen, speichern und zu einem späteren Zeitpunkt wieder aufrufen. Als Beispiel hier das Kopieren von einer Datei in eine andere:

```
readFileAsync("in.txt", function(err, content){
      if (!err) {
            writeFileAsync("out.txt", content, function(err){
                  if (!err) console.log("done")
            })
      }
})
```

Die Probleme liegen auf der Hand: Hoher Verschachtelungstiefe, keine Komposition von Fehlerbehandlung, dazu keine Verwendbarkeit von Schleifen und Bedingungen … das ist kein Spaß zum Programmieren. Zum Glück kann auf der Serverseite auf die neuste Version von ECMAScript zugegriffen werden, so dass mit der Zeit die Callback Hell langsam in einen Async Heaven umgewandelt werden konnte.

Dieser Umbau geschieht in zwei Schritten: Zuerst werden Promises eingeführt. Diese besitzen die Fähigkeit zur Komposition und können Fehlerzustände

weiterreichen. Dazu gibt die Funktion ein Promise zurück, das entweder erfüllt oder abgelehnt werden kann. Damit kann die Verschachtelungstiefe im Zaum gehalten werden.

```
readFilePromise("in.txt")
.then(function(content) {
      return writeFilePromise("out.txt", content)
}).then(function() {
      console.log(„done“)
})
```

Auch diese Struktur erzeugt immer noch einiges an Overhead. Das grundsätzliche Problem ist, dass JavaScript keine Pausierung oder Unterbrechung von Funktionsabläufen gestattet. Das wird mit der Einführung von Generatoren behoben. Ein solcher Generator kann bestimmen, dass seine Ausführung an bestimmten stellen unterbrochen wird und zu einem späteren Zeitpunkt wieder fortgesetzt wird. Der Syntax basiert auf den neuen Schlüsselwörtern function* und yield und baut auf Promises auf. Generatoren lösen das Problem noch nicht direkt. Als Zwischenschicht dient dabei die Bibliothek co(github.com/tj/co), die Generatoren genau zu diesem Zweck einsetzen. Damit lässt sich asynchroner Code so schreiben:

```
co.wrap(function*(){
      let result = yield readFilePromise("in.txt")
      yield writeFilePromise("out.txt", result)
      console.log("done")
}
```

Der Unterschied zu synchronen Code ist strukturell fast völlig eliminiert. Es braucht natürlich immer noch einen entsprechenden Kontext – darüber hinaus werden aber Komposition, Fehlerbehandlung und Schleifen vollständig unterstützt:

```
co.wrap(function*(){
      let ants = yield db.readAll()
      for (ant in ants) {
            try {
                  let code = yield db.read(ant)
                  ant.setCode(code)
            } catch (e) {}
      }
      console.log("done")
}
```

Der gesamte Server ist nun in diesem Stil programmiert und erleichtert damit auch die weitere Entwicklung.

Complete Rewrite

Im Laufe der Entwicklung erreichte ich einen Punkt, an dem es schwierig wurde, die Simulation weiterzuentwickeln. Problemfelder waren die großen Abhängigkeiten zwischen allen Modulen und ein kleines Performance-Problem. Meine Vision war es, mithilfe einer stärkeren Ereignisorientierung die einzelnen Teile der Simulation stärker voneinander zu trennen und eine Optimierung im Bereich der 2D-Suche zu implementieren. Ich verfolgte die Idee, die gesamte Simulation nochmal komplett neu zu schreiben. Das hat sich als sehr riskantes Unternehmen herausgestellt.

Die erste Phase der Neuimplementation lief gut, ich konnte den Code in einer verbesserten Struktur schreiben. Mittendrin stellte sich aber heraus, dass das Testen des Codes sehr schwer ist, wenn so viele Teile noch unfertig sind. Viele große Bugs existierten und die Chancen standen schlecht, deren Fehlerquellen in kurzer Zeit zu eliminieren. Eine Online-Recherche zeigte, dass viele Leute diesem Fehler verfallen sind. Ich weiß nun aus eigener Erfahrung, dass ein Complete Rewrite kein einfacher Weg ist.

Trennung von Simulation und View

Nichtsdestotrotz ist die Architektur zur Trennung der Simulation und der Darstellung gut gelungen und ich übernehme diese in die bestehende Simulation. Problematisch war, dass die Simulationsklassen eine direkte Referenz auf 3D-Objekte enthalten. Dadurch kann z.B. keine Simulation mit einer anderen, wie z.B. textuellen, Darstellung laufen. Um die Referenzen zu entfernen, aber trotzdem eine Kommunikation zu erlauben, wird ein Event-Bus eingerichtet. Die Simulation sendet nun nach außen „Events“, die nicht notwendigerweise behandelt werden müssen. 3D-Objekte werden mit eindeutigen Schlüsseln unterschieden. Die 3D-Simulation kann nun als separates Modul zur Simulation hinzu geschaltet werden und alle Ereignisse auch tatsächlich implementieren.

Die Simulation kann nun in ein komplett unabhängiges Modul verpackt werden, das nach außen hin keine Abhängigkeiten hat. Es können nun mehrere Simulationen hintereinander ausgeführt werden und die Ergebnisse in summarischer Form dargestellt werden.

Technische Dokumentation

Es folgen nun Informationen zu den einzelnen Bestandteilen von AntIT! und eine detaillierte Darstellung der technischen Implementation.

Projektstruktur

Im Stammverzeichnis befindet sich eine Konfigurationsdatei, die lokale Einstellungen zum Server beinhaltet. Diese muss zu Beginn ausgefüllt werden. Die Installation und Ausführung erfolgt ganz klassisch für node.js mit

```
npm install
npm start
```

Der Ordner „server“ enthält zusammen mit der Datei base.js den Quellcode für den Server. Die Templates für das Frontend finden sich im Ordner „views“. Der Ordner „public“ wird als statischer Ordner vom Server ausgeliefert und enthält clientseitige Komponenten:

assets	Texturen für die 3D-Simulation
css	Stil-Dateien
deco	Header-Bilder
extern	importierte Unterseiten
images	Bilder für E-Learning-Umgebung
js	Javascript-Bibliotheken: arrow36.js – Nach-oben-Pfeil bootstrap.min.js – Javascript-Bibliothek von Bootstrap 4 codemirror.js – Code-Editor im Browser jquery-3.3.1.slim.min.js – Abhängigkeit von Bootstrap jshint.min.js – Linter im Browser orbit.js – Plugin für three.js zur 3D-Ansichtssteuerung seedrandom.min.js – Zufallsgenerator mit Seed stats.min.js – Plugin für three.js für Statistikfenster three.min.js – Abstraktionsschicht für WebGL und mehr
models	JSON-Modelle für 3D-Simulation
src	Quellcode der Simulation
static	Elemente der Website
txt	einzelne Textdateien

Außerdem enthält das Stammverzeichnis den Ordner „test“, der verschiedene Dateien beinhaltet, die während der Entwicklung wichtig sind.

Das gesamte Projekt ist mit git versioniert und steht zu größten Teilen unter der MIT Lizenz. Die Online-Version ist unter der Adresse ants.arrrg.de aufrufbar. Diesen Server betreibe ich selber. Der Quellcode ist aktuell noch nicht veröffentlicht.

Server

Der Server läuft auf Node.js und beherbergt eine Express-Anwendung. Das Projekt enthält folgende Abhängigkeiten:

bcryptjs	Hashing für Benutzerpasswörter
body-parser	Unterstützung für Post-Inhalte
co	Generator-Bibliothek für asynchronen Code
connect-flash	Verwalter für Benutzerpopupnachrichten
connect-gzip-static	gzip-Optimierung
csurf	Schutz vor CSRF
ejs	Templating-Engine
express	Webserver-Framework
express-session	Benutzer-Session-Manager
mongodb	Datenbanktreiber
monk	Abstraktionsschicht über mongodb

Im globalen Namensraum **App** werden von den einzelnen Modulen folgende Einträge angelegt:

base.js

Die Konfigurationen unter config sind in der Datei config.js anpassbar.

config.databaseUrl: string

> Verbindungsinformationen zur Datenbank, wird nach dem Schema
>
> mongodb://<benutzer>:<passwort>@<server>:<port>/<database>
>
> aufgebaut. Es reicht zu Beginn, wenn die Datenbank leer ist. AntIT! legt bei der Einrichtung automatisch die entsprechenden Tabellen an.

config.serverIp: string

IP-Adresse des Servers, z.B. localhost

config.serverPort: number

Portnummer für den Server, z.B. 3000

config.managerPwd: string

Passwort für den Manager-Bereich zum Anlegen von Kolonien

config.devmode: boolean

Entwicklungsmodus, deaktiviert das Caching von Tutorials und Aufgaben

config.post: string

Postanschrift des Websitebetreibers

config.mail: string

Email-Anschrift des Websitebetreibers

express: Express

Instanz von Express für diesen Server. Auf dieser Instanz werden Routes definiert

db: IMonkManager

Instanz von monk für den Datenbankzugriff

csurf: csurf

Instanz von csurf zum Schutz vor CSRF bei Formularen

start: Promise

Startpunkt des Servers, nach Verbindung der Datenbank und Start von Express können hier eigene Callbacks mit .then() angehängt werden

colonies.js

colo.getCol(path: string): ICollection

Gibt zu einer Kolonie die entsprechende Datenbanktabelle zurück

colo.get(path: string): Beschreibungsobjekt

Gibt zu einer Kolonie das Beschreibungsobjekt zurück. Die Einträge entsprechen dem Schema der Datenbank (siehe weiter unten)

colo.all(): Beschreibungsindex

Gibt alle Beschreibungsobjekte als indexiertes Objekt zurück

colo.refresh(): void

Liest den Index neu ein

users.js

users.login(username: string, password: string, colony: string, req: Express.Request): Promise

Loggt einen Benutzer ein, falls Anmeldedaten in Datenbank vorhanden

users.logout(req: Express.Request): void

Loggt diesen Benutzer aus

users.auth: Express.Middleware

Middleware für Express, um sicherzustellen, dass Benutzer eingeloggt ist

users.register(req: Express.Request, col: Datenbanktabelle): Promise

Registriert einen Benutzer neu

ants.js

ants.prepareAnts(users: [], id: string, addSuffix: boolean): {ants:[], globals:[]}

Nimmt die Rohdaten aus der Datenbank und extrahiert die Ameisen, die vom Benutzer selber stammen und die für ihn sichtbar sind

ants.maximumAnts(level: number): number

Gibt die maximale Menge an Ameisenvölker für ein bestimmtes Level an

Der Server definiert folgende Routes:

/	Zeigt Startseite, falls eingeloggt wird Auflistung der Ameisenvölker angezeigt
/login/:colony GET/POST	Login für ein Kolonie
/logout	Loggt Benutzer aus und leitet weiter auf /
/wettbewerb	Zeigt Arena-Seite an
/addfriend	Fügt einen Freund hinzu
/unfriend	Löscht einen Freund
/home	Startseite für eingeloggte Benutzer mit aktueller Begrüßung
/info	FAQ mit häufigen Fragen
/abstract	Zusammenfassung der API
/register GET/POST	Registrierung für öffentlichen Server
/datenschutz	Zeigt Seite mit Datenschutzhinweis an
/kontakt	Zeigt Seite mit Kontaktinformationen
/simulation	Startet eine Simulation in der Arena
/submit	Schickt das Ergebnis der Simulation an den Server
/stats	Zeigt für Administratoren Simulationsverlauf an
/clearstats	Löscht für Administratoren Simulationsverlauf
/tutorial	Übersicht der verfügbaren Tutorials
/tutorialcheck	Überprüft, ob Quiz im Tutorial richtig gelöst ist
/level	Zeigt Übersicht von Aufgaben
/upgrade	Versucht eine Stufe aufzusteigen
/downgrade	Macht Stufenaufstieg rückgängig
/levelsim	Startet Aufgabe
/submitlevel	Sendet Ergebnis einer Aufgabe an den Server
/competitionDone	Administratoren schalten Stufe 6 frei
/competitionUndone	Administratoren machen Freischaltung rückgängig
/overview	Zeigt für Administratoren den Fortschritt der Benutzer
/edit	Bearbeitet ein Ameisenvolk
/save POST	Speichert Ameisenvolk
/new	Erstellt ein neues Ameisenvolk
/delete	Löscht ein Ameisenvolk
/publish	Gibt ein Ameisenvolk frei
/unpublish	Macht Freigabe rückgängig

/users/:colony	Gibt Root-Administratoren Zugriff auf Benutzer
/users	Auflistung von Benutzern für Administratoren
/users/edit/:id	Bearbeitet einen Benutzer
/user/new	Legt einen neuen Benutzer an
/user/save POST	Speichert Benutzer
/user/delete	Löscht Benutzer
/root GET/POST	Übersichtsseite / Anmeldeseite für Root-Administratoren
/root/logout	Loggt Root-Administratoren aus
/root/edit/:colony	Bearbeitet eine Kolonie
/root/new	Erstellt neue Kolonie
/root/save	Speichert Kolonie
/root/delete	Löscht Kolonie

Bis auf den /root-Route sind alle anderen Routes von der Anwendung aus erreichbar. Intern verwendet die Datenbank für die einzelnen Collections folgendes Schema:

info

Diese Collection definiert verschiedene Kolonien. Jede Kolonie ist in sich abgeschlossen: Benutzer und Ameisenvölker sind nur innerhalb einer Kolonie sichtbar, ebenso die Befugnisse von Administratoren. Über colo.get() kann auf den Eintrag einer Kolonie zugegriffen werden, folgende Informationen gibt es:

colonyName: string	Pfad zur Kolonie
description: string	Beschreibung der Kolonie
active: boolean	Kolonie wird auf Startseite angezeigt oder nicht
created: Date	UTC-Zeitpunkt der Erstellung
competitionDone: boolean	Wettbewerb fertig, damit Stufe 6 frei

sessions

Diese Collection wird verwendet, um Benutzersitzungen zu speichern. Dadurch können angemeldete Benutzer nach einem Neustart des Servers dort weitermachen, wo sie aufgehört haben. Die Dokumente haben folgende Form:

sid: string	Session-ID
session: string	JSON der Session
expires: Date	Ablaufzeitpunkt dieses Dokuments

simStats

Diese Collection speichert für alle Kolonien die Ergebnisse von Simulationen:

antsID: string[]	Liste der beteiligten Ameisenvölker
antsName: string[]	Liste der Namen der Ameisenvölker
userNames: string[]	Liste der Besitzer der Ameisenvölker
userid: ObjectId	ID des ausführenden Benutzers
username: string	Name des ausführenden Benutzers
start: number	Start-Zeitstempel
hash: string	Eindeutige Kennung der Simulation
colony: string	Zugehörige Kolonie
result: number[]	Ergebnispunktzahlen
expires: Date	Ablaufzeitpunkt des Dokuments

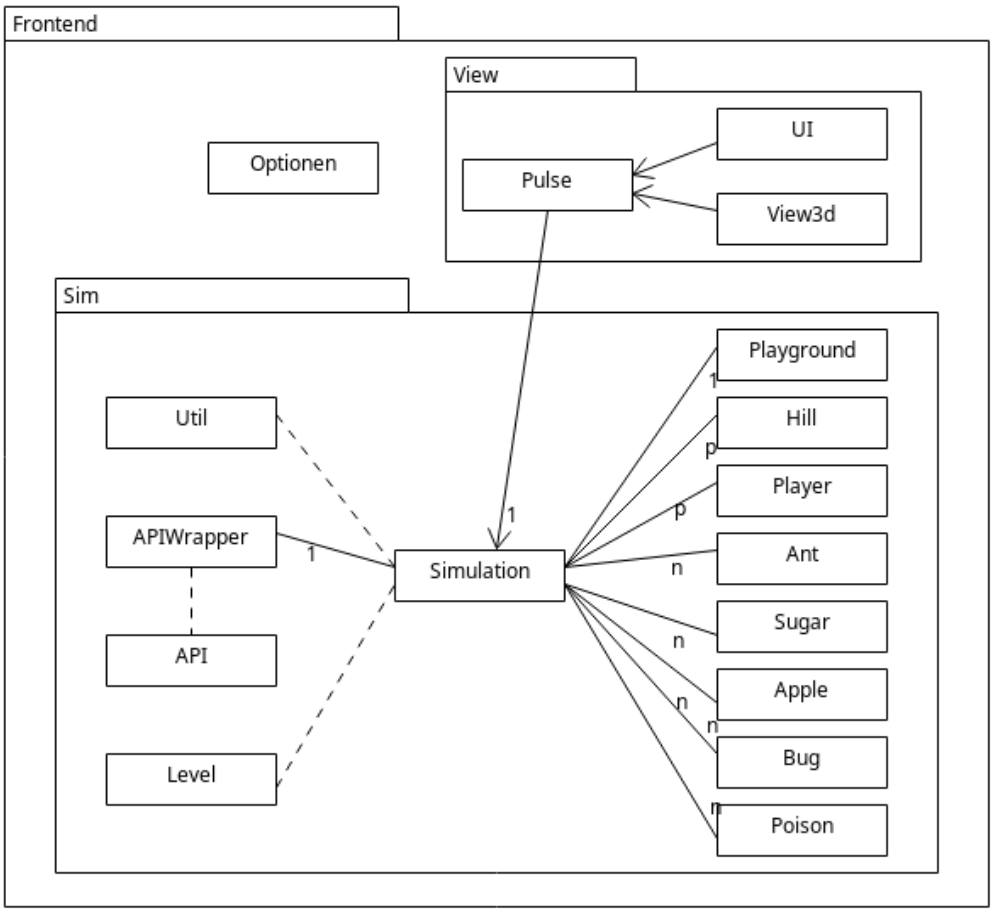
colony_<Pfad>

Jede Kolonie speichert ihre Daten in einer eigenen Collection, dabei bilden die Dokumente die Benutzer ab. Die einzelnen Ameisenvölker der Benutzer sind als Sub-Dokument eingebettet:

username: string	Benutzername zur Anmeldung
displayName: string	Anzeigename auf Weboberfläche
password: string	Passwort, mit bcrypt verschlüsselt
superuser: boolean	Administratorenrechte verfügbar oder nicht
level: number	Benutzerstufe im E-Learning
done: number[]	IDs der gelösten Quizze
solved: number[]	IDs der gelösten Aufgaben
ants	Sub-Dokument mit folgenden Feldern: antid: string – ID eines Ameisenvolks name: string – Name des Ameisenvolks published: boolean – Freigegeben oder nicht code: string – Programmcode der Ameise

Simulation

Das Herzstück von AntIT! ist die Ameisensimulation. Diese Simulation besteht aus mehreren Teilen, die im folgenden vorgestellt werden. Folgendes Diagramm zeigt die Grundklassenstruktur:



Es gibt drei Pakete. Das Frontend-Paket enthält die gesamte Ameisensimulation, die wiederum in die Simulation an sich und die View zerfällt. Außerhalb der zwei Pakete befinden sich die Optionen, die für alle Teile im Frontend zuständig sind. Die einzelnen Klassen und deren Beziehungen werden nun kurz beschrieben:

Simulation

Diese Klasse verwaltet die Grundstruktur einer Simulation und enthält Referenzen auf die Bestandteile der Simulation. Sie kümmert sich um die Initialisierung der Komponenten und informiert sie, wenn ein Update notwendig ist. Innerhalb des Sim-Namesraums können die Komponenten frei

auf die anderen Komponenten zugreifen. Das ist aufgrund von vielen Interaktionen zwischen den Bestandteilen der Simulation notwendig.

Util

Statische Hilfsklasse, die häufig verwendete Funktionen bereitstellt zu Abstand, Winkel und Drehung.

APIWrapper

Stellt Verknüpfungspunkte bereit, auf denen die API der Ameisen definiert werden kann.

Playground

Jede Simulation besitzt ein Spielfeld. Dieses kümmert sich um die Positionierung der Ameisenhügel, lässt Nahrungsmittel erscheinen und verwaltet die Wanzen. Das Spielfeld kümmert sich auch um das Entfernen von leeren Zuckerhaufen und toten Ameisen.

Hill

Jeder Spieler besitzt einen Ameisenhügel. Dieser verwaltet die Energie eines Spielers und lässt neue Ameisen erscheinen.

Player

Diese Klasse speichert Statistiken zu jedem Spieler und enthält die Referenz auf die KI.

Ant

Diese Klasse steuert eine Ameise, indem Befehle entgegen genommen und in Reihenfolge ausgeführt werden. Es gibt Befehle zur Bewegung, Interaktion mit Nahrungsmittel, Kommunikation und Kampf. Außerdem werden Nahrungsmittel und Gegner wahrgenommen.

Sugar

Stellt einen Zuckerhaufen dar, berechnet die Größe anhand der Restkapazität.

Apple

Stellt einen Apfel dar. Lässt Ameisen warten, bis genügend Träger eingetroffen sind, verwaltet das gemeinsame Tragen und entscheidet, welches Team den Apfel tragen darf.

Bug

Bewegt eine Wanze und lässt sie Ameisen angreifen.

Poison

Verwaltet Giftwolken.

API

Stellt eine API bereit, mit der die KI einer Ameise Befehle erteilen kann.

Level

Stellt die Definitionen von einzelnen Aufgaben bereit.

Pulse

Diese Klasse enthält eine Referenz auf die Simulation und lässt sie in einem bestimmten Takt laufen. Dabei werden Unregelmäßigkeiten korrigiert.

UI

Stellt Status und Statistiken zur Simulation bereit.

View3d

Kümmert sich um die Darstellung der Simulation in 3D. Das Frontend ist so entwickelt, dass die Simulation an sich abgeschlossen ist. Diese kommuniziert nur über einen Bus nach außen und braucht daher keine Referenz auf die Ansicht. Dadurch ist es dem Frontend freigestellt, ob eine Darstellung in 3D stattfindet oder nicht. Der Bus definiert folgende Ereignisse. Dabei wird jedem 3D-Objekt ein eindeutiger Schlüssel zugeordnet. Falls der Schlüssel noch nicht definiert ist, wird ein neues Objekt angelegt.

change-ant-color (key, color)	Ändert die Farbe einer Ameise
change-ant-level-color (key, c1, c2)	Ändert die Farbe der Fühler
move-ant (key, pos, rot)	Bewegt und dreht eine Ameise
move-sugarbox (key, pos)	Bewegt ein Zuckerstückchen
remove-sugarbox (key)	Macht ein Zuckerstückchen unsichtbar
remove-ant (key)	Macht eine Ameise unsichtbar
move-apple (key, pos, height)	Bewegt einen Apfel
remove-apple (key)	Macht einen Apfel unsichtbar
move-bug (key, pos, roty)	Bewegt und dreht eine Wanze
remove-bug (key)	Macht eine Wanze unsichtbar
add-marker (key, pos, color)	Lässt eine Markierung erscheinen
remove-marker (key)	Macht Markierung unsichtbar
update-marker (key)	Aktualisiert Markierung
move-hill (key, pos)	Setzt Hügel an eine Position
change-hill-color (key, color)	Setzt Flaggenfarbe eines Hügels
set-xy (w, h)	Setzt Breite und Höhe des Spielfelds
set-camera (x, y, z)	Setzt Kameraposition
move-sugar (key, pos, scale)	Bewegt und skaliert Zuckerhaufen
remove-sugar (key)	Macht einen Zuckerhaufen unsichtbar
move-spawn-point (key, pos)	Setzt gelben Ring
move-spawn-point2 (key, pos)	Setzt roten Ring
set-ring (pos, color, {inner, outer})	Setzt allgemein einen Ring
spawn-poison (key, pos, color)	Lässt eine Giftwolke erscheinen
remove-poison (key)	Löscht eine Giftwolke
show-grid (pos)	Erzeugt Gitternetz um Punkt
draw-text ({text, pos, size, h, key, color})	Erzeugt Text
remove-text (key)	Löscht Text
add-dead-info (info)	Fügt Todesinfo hinzu
toggle-dead-info ()	Wechselt Sichtbarkeit von Todesinfo

Optionen

Diese statische Klasse stellt Optionen bereit, mit denen sich die Ameisensimulation konfigurieren lässt. Insgesamt sind folgende Optionen verfügbar:

Name	*Wert*	*Beschreibung*
MaximaleSpieler	8	Anzahl der maximalen Teilnehmer pro Simulation
Runden	7500	Anzahl der Ticks für eine Simulation in der Arena
Spielfeld-verhältnis	4.0/3.0	Seitenverhältnis des Spielfelds
SpielfeldGrund-Größe	600000	Quadratpixelzahl des Spielfelds für 0 Spieler, wird für jeden Spieler zusätzlich addiert
HügelAbstand	500	Mindestabstand der Spielerhügel voneinander
HügelRand-Abstand	200	Mindestabstand vom Spielfeldrand
HügelStreifen-Breite	100	Bei der Platzierung werden Hügel bevorzugt am Rand in einem bestimmten Streifenbereich platziert
EckenAbstand	300	Mindestabstand der Hügel von der Spielfeldecke
HügelRadius	40	Radius des Hügels für Simulationsberechnungen
BauErreicht-Radius	10	Radius, damit Ameise den Mittelpunkt des Baus erreicht
SpielerFarben	[…]	Ein Array aus Farbwerten für die Spieler
FühlerFarben	[…]	Je nach Stufe besitzen Ameisen andere Fühlerfarben
ZuckerGröße	250	Anzahl Zucker pro Zuckerhaufen
Zucker-Vergrößerung	0.1	Zoom-Faktor für Zucker-3D-Objekt
ApfelGröße	2.0	Zoom-Faktor für Apfel-3D-Objekt
HügelGröße	1.0	Zoom-Faktor für Hügel-3D-Objekt
AmeisenGröße	2.5	Zoom-Faktor für Ameisen-3D-Objekt
WanzenGröße	1.5	Zoom-Faktor für Wanzen-3D-Objekt
ZuckerBoxGröße	3.0	Zoom-Faktor für Zuckerstücken-3D-Objekt
MarkerGröße	0.6	Zoom-Faktor für Nachrichten-Markierung
MarkerDauer	25	Anzeigedauer von Nachrichten-Markierung
MarkerVer-größerung	1.04	Faktor für Vergrößerung der Markierung
MarkerDurchsich-tigkeit	0.3	Transparenz für Nachrichten-Markierung
MarkerFading	0.92	Faktor für Ausblenden der Markierung
ZuckerStückchen-Höhe	7.3	Höhe eines Zuckerstückchens über dem Boden (auf Rücken von Ameise)
NahrungsMindest-Entfernung	450	Abstand von erscheinenden Nahrungsmittel vom Ameisenhügel

NahrungsMaximalEntfernung	1000	Maximaler Abstand erscheinender Nahrungsmittel vom Ameisenhügel
NahrungsWartezeit	450	Zeitlicher Abstand, in dem neue Nahrungsmittel erscheinen
NahrungAbstand	150	Abstand von Nahrungsmittel voneinander
AmeiseWartezeit	30	Anzahl Ticks zwischen der Geburt von Ameisen
Ameisen-Maximum	100	Maximale Anzahl an Ameisen
AmeiseGeschwindigkeit	5	Gehgeschwindigkeit einer Ameise pro Tick
AmeiseDrehgeschwindigkeit	8	Drehgeschwindigkeit in Grad pro Tick
AmeiseSichtweite	70	Sichtweite einer Ameise
AmeiseTragkraft	5	Anzahl Zucker, die eine Ameise tragen kann
PunkteProZucker	5	Punkte pro Zucker
ZuckerVerlangsamung	0.75	Verlangsamungsfaktor beim Tragen von Zucker
TicksProSekunde	40	Simulationsgeschwindigkeit
MaximalÜbersprungeneFrames	10	Falls die 3D-Ansicht mit dem Rendern zu langsam ist, werden bis zu dieser Anzahl Frames übersprungen
AmeisenFürApfel	4	Benötigte Mindestanzahl Ameisen zum Apfeltragen
MaximumAmeisenFürApfel	20	Höchstzahl an Ameisen beim Apfeltragen
ApfelMinGeschwindigkeit	0.2	Geschwindigkeit des Apfel bei Minimalzahl Träger
ApfelMaxGeschwindigkeit	2.0	Geschwindigkeit des Apfel bei Maximalzahl Träger
ApfelRadius	15	Bereich, in denen Ameisen den Apfel tragen können
PunkteProApfel	1000	Punktzahl für einen Apfel beim Bau
AnfangsEnergie	4000	Startmenge von Energie für neue Ameisen
EnergieFür-Ameise	200	Benötigte Anzahl an Energie für eine Ameise
EnergieProApfel	2000	Energie für einen Apfel
EnergiePro-Zucker	10	Energie für einen Zucker
AmeisenReichweite	3000	Maximale Anzahl Schritte, bevor Ameise an Erschöpfung stirbt, wird im Bau zurückgesetzt

WanzenPro-Spieler	1	Legt die Anzahl an Wanzen pro Spieler fest
WanzenWartezeit	300	Erscheinungsdauer der Wanzen
WanzenKampf-weite	12	Angriffsradius einer Wanze
AmeiseEnergie	60	Lebensenergie einer Ameise
WanzenAngriff	10	Schaden einer Wanze an einer Ameise
WanzeDrehge-schwindigkeit	5	Drehgeschwindigkeit einer Wanze
WanzeGeschwin-digkeit	3	Gehgeschwindigkeit einer Wanze
WanzeSichtweite	60	Sichtweite einer Wanze
WanzenHügel-Abstand	250	Mindestabstand, den eine Wanze von einem Hügel bewahrt
WanzeVoraus-Winkel	45	Winkel für das Ereignis „SiehtWanzeVoraus“
ZufallRichtungs-Verschiebung	11	Ameise wird bei jeder Bewegung um einen kleinen Winkel versetzt, verhindert komplette Überlagerungen
EntwicklerModus	false	Aktiviert Zugriff auf Simulationsparameter
Levelmodus	false	Wird für die Simulation von Aufgaben eingesetzt
Toleranz	3	Toleranzradius für das genaue Erreichen eines Ziels
ZuckerRadius	10	Wirkungsbereich eines Zuckerhaufens
JobLimit	200	Maximale Anzahl Befehle in Warteschlange
Harmonie	false	Aktiviert den „Nur-Sammel-Modus“

Schnittstelle zum Server

Damit der Server eine Simulation starten kann, lädt dieser den Quellcode der KI in die Website. Dabei ist der Quellcode mit base64 codiert, damit er nicht direkt ausgelesen werden kann. Außerdem befindet sich die KI in einer Sandbox, in der nur die Befehle der Ameisen API und eine whitelist zur Verfügung stehen (z.B. Math und console.log). Im Browser wird dann folgender Code ausgeführt:

```
var levels = [5,3,3,9]
var colors = [4,3,6,7]
AntIT.SetzeSeed("hallo123")
injectCode() // evaluiert die KI in der Sandbox
hash = "1560577725326-5b7e79f30924f42e85b81abf-52596"
returnUrl = "/wettbewerb"
AntIT.StarteSimulation(hash, returnUrl, null, levels, colors)
```

Der Server gibt eine Liste von Ameisenstufen und Ameisenfarben weiter. Außerdem wird ein Seed gesetzt und der Simulation wird ein Hash zugeordnet. Die Simulation startet, sobald alle Resourcen geladen sind und die 3D-Ansicht initialisiert ist (bzw. bei Simulation ohne 3D gleich beim Laden der Seite)

Ameisen-API

Um die KI einer Ameise zu entwickeln, kann über eine API auf verschiedene Befehle, Funktionen und Ereignisse zugegriffen werden. Die Entwicklung einer KI findet in einem Webeditor statt. Ein Beispiel für eine Ameise sieht so aus:

```
1  var Ameise = AntIT.NeueAmeise("Bmeise")
2
3  Ameise.wenn("IstUntätig", function(){
4      Gehe(300)
5      Drehe(20)
6  })
7
8  Ameise.wenn("SiehtZucker", function(zuc){
9      GeheZuZielDirekt(zuc)
10     NimmZucker()
11     GeheZuBau()
12     LadeZuckerAb()
13 })
14
15 Ameise.wenn("SiehtWanzeVoraus", function(wanz){
16     Drehe(90)
17     Gehe(200)
18 })
```

In der Zeile 1 wird ein neues Ameisenvolk erzeugt. Als Parameter wird der Name des Ameisenvolks übergeben. Für dieses Ameisenvolk können nun eine Reihe von Ereignissen definiert werden. Diese werden zu den entsprechenden Zeitpunkten aufgerufen und enthalten eventuell Parameter. Innerhalb von Ereignissen kann die KI dann für die einzelnen Ameisen Befehle definieren. Dabei enthält jede Ameise eine Warteschlange an Befehlen. Diese Warteschlange wird vor jedem Ereignis

zurückgesetzt und kann innerhalb eines Ereignisses mit Befehlen aufgefüllt werden. Die Befehle sind in der API definiert und werden nun im Folgenden dargestellt. Dabei werden im ersten Teil die Ereignisse beschrieben, dann die Befehle und zum Schluss eine Reihe von weiteren Funktionen und Techniken.

Ereignisse

„IstGeboren“

Eine Ameise ist im Bau neu geboren worden.

„IstUntätig“

Eine Ameise hat alle Befehle abgearbeitet, die Warteschlange ist nun leer.

„SiehtZucker“ (Sichtungsobjekt)

Die Ameise sieht in ihrer Sichtweite einen Zuckerhaufen. Als Argument wird das Objekt übergeben, welches dann zur Zielvisierung verwendet werden kann. Das Ereignis wird für einen Zuckerhaufen einmalig aufgerufen, solange es in der Sichtweite bleibt.

„SiehtApfel“ (Sichtungsobjekt)

Die Ameise sieht in ihrer Sichtweite einen Apfel, analog zu „SiehtZucker“

„RandErreicht“

Die Ameise ist am Rand des Spielfelds angestoßen.

„SiehtWanze“ (Sichtungsobjekt)

Die Ameise sieht in ihrer Sichtweite eine Wanze.

„SiehtWanzeVoraus“ (Sichtungsobjekt)

Die Ameise sieht in ihrer Sichtweite eine Wanze, die sich im 45-Grad-Winkel voraus befindet.

„SiehtGegner“ (Sichtungsobjekt)

Die Ameise sieht in ihrer Sichtweite eine gegnerische Ameise.

„Tick“

Wird jeden Tick der Simulation aufgerufen. Vorsichtig verwenden.

Befehle

Gehe(schritte)

Die Ameise geht die Anzahl Schritte geradeaus.

Drehe(winkel)

Die Ameise dreht sich um die Winkelzahl. Positive Zahlen bedeuten eine Linksdrehung, negative Zahlen eine Rechtsdrehung.

DreheZuRichtung(himmelsrichtung)

Das Spielfeld wird in Himmelsrichtungen unterteilt, dabei beginnt die Zählung bei Osten mit 0 und geht gegen den Uhrzeigersinn herum, also Süden = 90, Westen = 180, Norden = 270. Die Fahnen der Ameisenhügel zeigen immer nach Osten. Die Ameise dreht sich zur angegebenen Himmelsrichtung hin.

NimmZucker()

Wenn die Ameise in Reichweite eines Zuckerhaufens steht, lädt sie sich Zucker auf. Ansonsten macht der Befehl nichts.

LadeZuckerAb()

Die Ameise entlädt ihre Zuckerladung. Im Bau werden Energie und Punkte gutgeschrieben, außerhalb des Baus geht die Ladung verloren.

TrageApfel()

Wenn die Ameise in Reichweite eines Apfels steht, dann stellt sie sich zum Apfel und wartet, bis genügend Träger da sind. Dann tragen sie den Apfel gemeinsam zum Bau.

GeheZuBau()

Die Ameise macht sich direkt auf dem Weg zurück zum Bau.

GeheZuZiel(sichtungsobjekt)

Die Ameise visiert das Sichtungsobjekt an und geht darauf zu.

GeheZuBauDirekt()

Variante von GeheZuBau(), wobei die Ameise auf Sichtungen verzichtet. Die Ameise reagiert dann nicht auf Nahrungsmittel und Nachrichten.

GeheZuZielDirekt(sichtungsobjekt)

Variante von GeheZuZiel() ohne Sichtungen.

SetzeGift()

Die Ameise setzt eine Giftwolke um sich herum vom Radius 80. Die Ameise besitzt nur eine Giftladung und muss nach dem Setzen zum Bau zurück und die Ladung neu aufladen. Giftwolken werden erst giftig, wenn sich drei Wolken überlagern – dann sterben gegnerische Ameisen und Wanzen sofort.

Warte(ticks)

Lässt die Ameise eine Weile auf der Stelle stehen.

FührеAlteBefehleAus()

Im Normalfall wird der Rest der Warteschlange bei einem Ereignis entfernt. Mit diesem Befehl können die entfernten Ereignisse wieder in die Warteschlange aufgenommen werden. Davor und danach können weitere Befehle stehen, diese werden dann resp. vor und nach den alten Befehlen ausgeführt.

SendeNachricht(betreff, inhalt)

Die Ameise sendet eine Nachricht an alle ihre Kollegen im Team. Eine Nachricht wird durch den Betreff gekennzeichnet. Eine Ameise, die eine Nachricht erhält, bekommt das Ereignis „#<betreff>“. Auf dieses Ereignis kann die Ameise reagieren. Der Inhalt ist beliebig, es können Texte, Zahlen oder Sichtungsobjekte übergeben werden – und auch komplexe Daten.

SetzeLimit(anzahl)

Spezieller Befehl, der direkt vor dem Befehl SendeNachricht() aufgerufen werden wird. Dieser legt fest, an wie viele befreundete Ameisen in der Umgebung die Nachricht gesendet wird.

DreheZuObjekt(sichtungsobjekt)

Dreht die Ameise so, dass sie das Objekt anvisiert.

DreheWegVonObjekt(sichtungsobjekt)

Dreht die Ameise so, dass sie vom Objekt wegschaut.

Zusätzliche Funktionen

Ameise.SetzeTeams(anzahl)

Diese Funktion wird außerhalb der Ereignisse auf dem Ameisenobjekt aufgerufen. Damit lässt sich das Ameisenvolk in eine Anzahl von Team zuteilen. Die Ameisen werden der Reihe nach von 0, 1, 2, … aus zyklisch nummeriert.

InTeam(nummer | num1, num2, …): boolean

Kann innerhalb von Ereignissen verwendet werden. Prüft, ob die Ameise ein Teil des Teams / einer der Teams ist.

Ameise.SetzeTeamFolge(num1, num2, num3, …)

Alternative Möglichkeit, Teams zu setzen. Der Zuteilungszyklus kann explizit angegeben werden. Die erste Ameise kommt in Team num1, die zweite Ameise in Team num2, usw. Wird das Ende der Liste erreicht, beginnt sie wieder von vorne.

Zufall(von, bis): number

Erzeugt eine Zufallszahl inklusive von und inklusive bis.

SchrittZahl: number

Die Anzahl der Schritt, die die Ameise bereits gelaufen ist. Eine Ameise hat eine maximale Reichweite von 3000 Schritten und kann diese Reichweite nur im Bau zurücksetzen.

TickZahl: number

Anzahl der Ticks bisher in der Simulation.

Distanz(obj1, obj2): number

Berechnet die Distanz zwischen zwei Objekten, z.B. Sichtungsobjekte oder einer der beiden eingebauten Objekte Bau und Position.

Richtung(obj1, obj2): number

Berechnet die Himmelsrichtung, von der aus obj1 das obj2 sieht.

Bau: Objekt

Stellt den Heimatbau für Berechnungen zur Verfügung.

Position: Objekt

Stellt die aktuelle Position für Berechnungen zur Verfügung.

HatZucker: boolean, HatApfel: boolean

Prüfung, ob die Ameise aktuell Zucker geladen hat oder einen Apfel trägt.

Blickrichtung: number

Die Himmelsrichtung, in die die Ameise gerade schaut.

Gedächtnis: hash-map

Für jede Ameise existiert eine Hashmap, auf der sie selbstdefinierte Daten speichern kann. Diese Hashmap ist nur für eine einzelne Ameise zugänglich und persistent. Neue Werte können z.B. mit Gedächtnis.<key> = <value> angelegt werden und mit Gedächtnis.<key> wieder ausgelesen werden.

Aktiv(sichtungsobjekt): boolean

Prüft, ob sich ein Sichtungsobjekt noch auf dem Spielfeld befindet.

function* und yield

Normalerweise werden Ereignisse synchron ausgeführt. Um asynchron weitere Aktionen durchzuführen, kann über den Generator function* und yield die Ausführung angehalten werden und zu einem späteren Zeitpunkt wieder fortgesetzt werden:

```
Ameise.wenn("IstUntätig", function*(){
      Gehe(400)
      yield Drehe(90)
      console.log("Hello Howdy!") // wird asynchron ausgeführt
})
```

Zugang zum Quellcode

Das Gesamtpaket von AntIT! kann vorläufig unter der Adresse

https://arrrg.de/dist/antit_neu.zip

heruntergeladen werden.

Persönliches Fazit

Das Projekt AntIT! ist für mich in mehrerer Hinsicht eine einmalige Erfahrung. Zum einen ist es das erste Mal, dass ich ein Softwareprojekt in diesem Umfang realisiert habe. Alle meine vorherigen Projekte waren deutlich kleiner und hatten vor allem auch eine viel kürzere Laufzeit. An AntIT! arbeite ich nun seit drei Jahren. Das eröffnet mir einen Blick für langfristig wichtige Entscheidungen, wie z.B. die Wahl der Technologie und den Umgang mit bestehenden Code. Ich habe gelernt, dass man eine gewachsene Codebase nicht von heute auf morgen neu schreiben kann. Besser ist es, in kleinen Schritten das Programm zu verbessern. AntIT! ist auch deshalb etwas besonderes, weil die Entwicklung sehr eng an das Feedback von Teilnehmern gebunden ist. Im intensiven Austausch mit Schülerinnen und Schüler konnte ich sehr nah miterleben, wie Menschen mit meiner Software umgehen und dabei auch verstehen, welche Ideen funktionieren und welche nicht. Über viele Iterationen hinaus konnte ich die Abläufe immer weiter optimieren. Dieser Austausch ist sehr erfrischend und hat mir auch sehr viel Spaß gemacht. AntIT! ist zudem meine erste große, selbstständige Webapplikation. Auf diesem Gebiet konnte ich sehr viele Dinge lernen, auch was das Hosting, die Domainregistrierung und das Deployment angeht. Ich habe mich mit rechtlichen Aspekten und dem Datenschutz auseinandergesetzt.

Natürlich gibt es auch Bereiche, in denen ich Entwicklungspotenzial sehe. Einerseits überlege ich mir, meine nächsten Projekte auf der Basis von TypeScript aufzubauen. Durch die fehlende Typisierung wird es bei einem Projekt dieser Größe zunehmend schwieriger, die Struktur zu wahren. Außerdem frage ich mich, ob der Einsatz von MongoDB in diesem Zusammenhang notwendig gewesen war. Wahrscheinlich hätte hier auch eine normale SQL-Datenbank gereicht. Zuallerletzt finde ich es schwer, als Einzelperson an einem Projekt zu arbeiten. Es ist oft passiert, dass es mir schwer gefallen ist, mich für neue Bereiche und Aufgaben zu motivieren. Das ist schade, denn mir macht die Softwareentwicklung viel Spaß. Diesen Spaß mit anderen Menschen zu teilen – das wünsche ich mir für meine weiteren Tätigkeiten. Nichtsdestotrotz bleibt AntIT! für mich eine außerordentliche Erfahrung und wird mich sicherlich noch einige Zeit begleiten.