vires: Dokumentation

Marc Huisinga Vincent Lehmann Steffen Wißmann

1. März 2016

Inhaltsverzeichnis

1 Einleitung

Die Dokumentation erläutert die Ergebnisse der Entwicklung des Spiels ***vires*** im Rahmen des Oberstufenprojektes im Fach *Programmierpraktikum*.
Bei dem entwickelten Spiel handelt es sich um ein einfaches Multiplayer-RTS-Spiel, welches mit dem Browser gespielt werden kann.
Genauere Informationen bezüglich des Spiels, den ursprünglichen Zielen der Entwicklung und der anfänglichen Planung lassen sich im Pflichtenheft finden.

2 Benötigte Austattung

Für das Spielen von ***vires*** ist lediglich ein aktueller WebGL- und Websockets-fähiger Webbrowser (Internet Explorer 11, Firefox 4, o.ä) nötig.
Das Spiel verwendet zudem für Kamerabewegungen das Mausrad, eine Drei-Button-Maus ist also auch nötig.
Für das Deployen von ***vires*** ist die ***vires***-Zip nötig, welche unter `BUILD_LINK_HIER_EINFÜGEN` heruntergeladen werden kann.
Für das Compilen von Vires wird eine aktuelle Go-Version, vorzugsweise Version 1.4 oder höher, benötigt.

3 Nutzeranleitung

3.1 Nutzung

NUTZUNG HIER

3.2 Deployment

Um den ***vires***-Server zu starten, muss lediglich die .zip-Datei für das jeweilige Betriebssystem und die jeweilige Prozessorarchitektur unter `BUILD_LINK_HIER_EINFÜGEN` heruntergeladen, entpackt und über die Datei `vires` (bzw. unter Windows `vires.exe`) ausgeführt werden.

3.3 Building

Bei der Kompilierung des Projekts wird zwischen Backend und Frontend unterschieden:

- Für die Kompilierung des Backends wird zuerst eine Go-Installation benötigt.
  Hierfür muss eine aktuelle Go-Version von `https://golang.org/dl/` heruntergeladen und hieraufhin ein Go-Workspace eingerichtet werden.
  Zur Einrichtung eines Go-Workspaces muss ein Ordner für den Workspace erstellt werden und hieraufhin der Pfad des Ordners in der Umgebungsvariable `$GOPATH` gesetzt werden.
  Nach der Einrichtung des Workspaces können der ***vires***-Sourcecode und alle Abhängigkeiten mithilfe des Konsolenkommandos `go get github.com/mhuisi/vires/...` heruntergeladen und für das aktuelle Betriebssystem und die aktuelle Architektur kompiliert werden.
  Die reine Binärdatei kann dann in `$GOPATH/bin` aufgefunden werden und

sollte hieraufhin nach
`$GOPATH/src/github.com/mhuisi/vires/src/vires` bewegt werden.

- FRONTEND-BUILD-PROZESS HIER EINFÜGEN

4 Übertragungsprotokoll

Das Backend von ***vires*** lässt sich mit beliebigen Clients kombinieren, welche in der Lage sind, Websocket-Verbindungen zu eröffnen, insofern sich die Clients an das Übertragungsprotokoll halten.
Der Client, welcher standardmäßig vom Server unterstützt wird, kann also beliebig ausgetauscht werden: Es ist theoretisch einfach möglich, ***vires*** mit einem Desktopclient oder einem Webclient einer anderen Website zu spielen.
Dementsprechend ist das Übertragungsprotokoll auch einer der wichtigsten Unterpunkte dieser Dokumentation.
Da ***vires*** im Vergleich zu vielen anderen Spielen ein relativ langsames Spiel ist, ist es möglich, eine wirklich sichere Client-Server-Architektur aufzubauen, welche es keinem Client erlaubt, sich substantielle Vorteile durch die Veränderung des Client-Programmcodes zu verschaffen.
Der Server verwendet als Übertragungsformat JSON. Im Folgenden wird erläutert, wie das Protokoll aufgebaut ist.

4.1 Client-to-Server

Alle Client-to-Server-Pakete besitzen einen Type, welcher angibt, um welche Art Paket es sich handelt, eine Version, welche angibt, mit welcher Protokollversion der Client arbeitet, und Data, welche beliebige JSON-Daten je nach Art des Pakets enthalten kann. Die Grundstruktur eines Client-to-Server-Pakets sieht wie folgt aus:

```
{
    "Type": "Movement",
    "Version": "0.1",
    "Data": "A payload"
}
```

4.1.1 Movement

Movement ist das Paket, das der Client sendet, wenn er ein Movement starten möchte. Hierfür muss lediglich die Quell-ID einer Cell und die Ziel-ID einer Cell angegeben werden.
Beispielpaket:

```
{
        "Source": 1,
        "Dest": 2
}
```

4.2 Server-to-Client

Alle Server-to-Client-Pakete besitzen ebenfalls einen Type, welcher angibt, um welche Art Paket es sich handelt, eine Version, welche angibt, mit welcher Protokollversion der Client arbeitet, und Data, welche beliebige JSON-Daten je nach Art des Pakets enthalten kann. Die Grundstruktur eines Server-to-Client-Pakets sieht wie folgt aus:

```
{
        "Type": "Collision",
        "Version": "0.1",
        "Data": "A payload"
}
```

4.2.1 Movement

Movement ist das Paket, das zu allen Clients geschickt wird, wenn ein Movement erfolgreich gestartet wurde. Es besteht aus der ID des Movements, der ID des Besitzers, der Menge an Vires, welche sich in dem Movement befinden, dem Startpunkt des Movements, dem Radius des Movements und dem Richtungsvektor des Movements, wobei $|r| = \sqrt{{r_1}^2 + {r_2}^2}$ mit $[|r|] = \frac{Feldeinheiten}{s}$ ist.
Beispielpaket:

```
{
    "ID": 1,
    "Owner": 1,
    "Moving": 10,
    "Body": {
        "Location": {
            "X": 2,
            "Y": 3
        },
        "Radius": 5
    },
    "Direction": {
        "X": 2,
        "Y": 3
    }
}
```

4.2.2 Collision

Eine Collision wird zu allen Clients geschickt, wenn eine Collision auf dem Spielfeld stattfindet. Eine Collision besteht immer aus zwei Movement-Typen, deren Aufbau genau der gleiche wie bei dem Movement-Paket unter 4.2.1 ist. Alle Werte in den Movement-Typen wurden nach der Collision berechnet: Stirbt also beispielsweise ein Movement, so wird die Anzahl an Vires in dem Movement null sein.
Beispielpaket:

```
{
    "A": {
        "ID": 1,
        "Owner": 1,
        "Moving": 10,
        "Body": {
            "Location": {
                "X": 2,
                "Y": 3
            },
            "Radius": 5
        },
        "Direction": {
            "X": 2,
            "Y": 3
        }
    },
    "B": {
        "ID": 1,
        "Owner": 1,
        "Moving": 10,
        "Body": {
            "Location": {
                "X": 2,
                "Y": 3
            },
            "Radius": 5
        },
        "Direction": {
            "X": 2,
            "Y": 3
        }
    }
}
```

4.2.3 Conflict

Tritt ein Conflict auf, also trifft ein Movement auf seine Target-Cell, so wird ein Conflict-Paket an alle Clients gesendet. Für das Conflict-Paket gilt das gleiche wie für Collision-Pakete: Alle Werte sind nach dem Conflict berechnet, hat die Cell also nach einem Conflict den Besitzer gewechselt, so wird der neue Besitzer übertragen. Ein Conflict-Paket besteht aus einer Movement-ID, welche angibt, welches Movement mit der Cell kollidiert, einer Cell-ID, welche die Cell identifiziert, die Ziel des Conflicts ist, die Menge an Vires, die nach dem Conflict noch in der Cell vorhanden ist und die ID des Besitzers der Cell. Beispielpaket:

```
{
    "Movement": 5,
    "Cell": {
```

```
        "ID": 1,
        "Stationed": 2,
        "Owner": 10
    }
}
```

4.2.4 EliminatedPlayer

Wird ein Spieler aus dem Spiel ausgeschlossen, sei es weil er keine Cells mehr besitzt oder seine Verbindung getrennt wurde, so wird ein EliminatedPlayer-Paket an alle Clients gesendet. Bei diesem Paket handelt es sich lediglich um die ID des eliminierten Spielers.
Beispielpaket:

```
1
```

4.2.5 Winner

Gewinnt ein Spieler das Spiel, weil er der letzte Überlebende ist, so wird ein Winner-Paket an alle Clients gesendet, welches lediglich die ID des Siegers enthält.
Beispielpaket:

```
1
```

4.2.6 Replication

Vermehrt sich die Anzahl an Vires in den Cells, so wird ein Replication-Paket an alle Clients gesendet. Das Replication-Paket enthält die ID und die neue Anzahl an Stationed Vires aller Cells.
Beispielpaket:

```
[
    {
        "ID": 1,
        "Stationed": 20
    },
    {
        "ID": 1,
        "Stationed": 20
    },
    {
        "ID": 1,
        "Stationed": 20
    }
]
```

4.2.7 Field

Wird das Match gestartet und ein Field generiert, so wird ein Field-Paket an alle Clients übertragen. Das Field enthält die ID, den Ort, den Radius und die Capacity jeder Cell. Es ist garantiert, dass die IDs bei null anfangen und es keine Lücken zwischen den IDs gibt. Außerdem enthält das Paket die Anfangszellen der Spieler des Matches als ID des Besitzers und ID der Cell, die die Anfangszelle des Besitzers darstellt. Letztlich enthält das Paket ebenfalls die Größe des Fields.
Beispielpaket:

```
{
    "Cells": [
        {
            "ID": 1,
            "Body": {
                "Location": {
                    "X": 2,
                    "Y": 3
                },
                "Radius": 5
            },
            "Capacity": 10
        },
        {
            "ID": 1,
            "Body": {
                "Location": {
                    "X": 2,
                    "Y": 3
                },
                "Radius": 5
            },
            "Capacity": 10
        },
        {
            "ID": 1,
            "Body": {
                "Location": {
                    "X": 2,
                    "Y": 3
                },
                "Radius": 5
            },
            "Capacity": 10
        }
    ],
    "StartCells": [
        {
            "Owner": 1,
            "Cell": 2
```

```
        },
        {
            "Owner": 1,
            "Cell": 2
        }
    ],
    "Size": {
        "X": 2,
        "Y": 3
    }
}
```

4.2.8 UserJoined

Tritt ein User dem Room bei, so wird ein UserJoined-Paket an alle Clients gesendet. Bei einem UserJoined-Paket handelt es sich lediglich um die ID des Users.
Beispielpaket:

```
1
```

4.2.9 OwnID

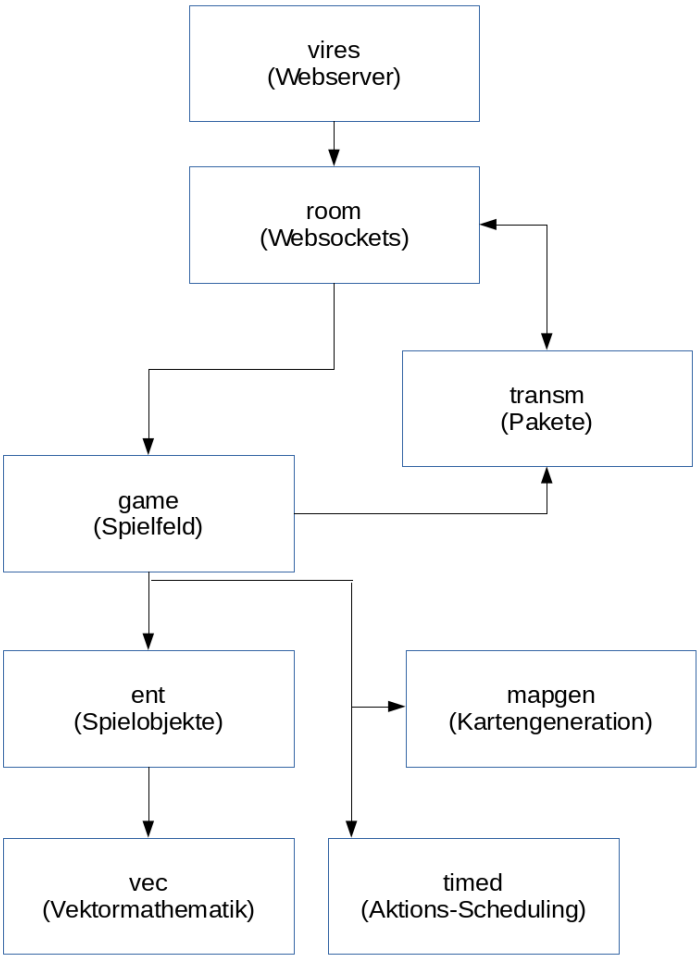
Tritt ein user dem Room bei, so wird ihm mittels eines OwnID-Pakets seine eigene ID im Spiel mitgeteilt. Auch bei dem OwnID-Paket handelt es sich nur um die ID des Users.
Beispielpaket:

```
1
```

5 Programmaufbau

5.1 Backend

Das Backend besteht aus mehrere Paketen und ist wie folgt aufgebaut:



Die Bedeutung der einzelnen Pakete und wichtige Algorithmen werden im Folgenden erklärt.

5.1.1 vires

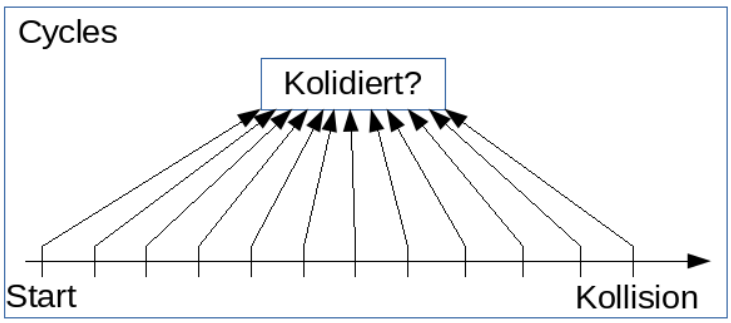
`vires` ist das Main-Package des Programms und kümmert sich um die Verwaltung des Webservers von ***vires***.
Wird eine GET-Anfrage auf einen Link ausgeführt, der eine Room-ID enthält (z.B. `http://localhost/1234`), so wird eine Template für den Raum, welche mit der Room-ID kompiliert wird, an den User gesendet.
Die kompilierte Template öffnet hieraufhin eine Websocket-Verbindung zu `/<roomid>/c`.
Der Handler für `/<roomid>/c` eröffnet dann auch serverseitig die Websocket-Verbindung und übergibt die Verbindung an die jeweilige `room`-Instanz.

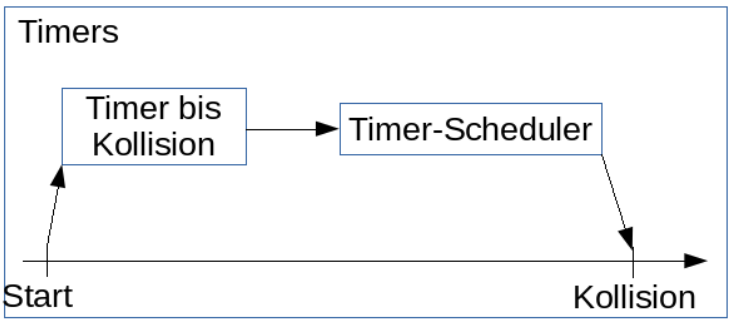
5.1.2 room

`room` verwaltet die Websocket-Verbindungen eines Rooms. Verbindet sich der erste User mit dem Room, so wird der Room erstellt.
Alle Operationen eines Rooms werden von einer Monitor-Goroutine verwaltet, welche den Zugriff auf alle Zustände des Rooms synchronisiert.
Insgesamt verwaltet die Monitor-Goroutine Spielerverbindungen, Matches, Pakete von und an Nutzer und Pakete vom Spiel.
Wird eine Websocket-Verbindung zu dem Room hinzugefügt, so werden für diese Verbindung eine Reader- und eine Writer-Goroutine gestartet.

5.1.3 game

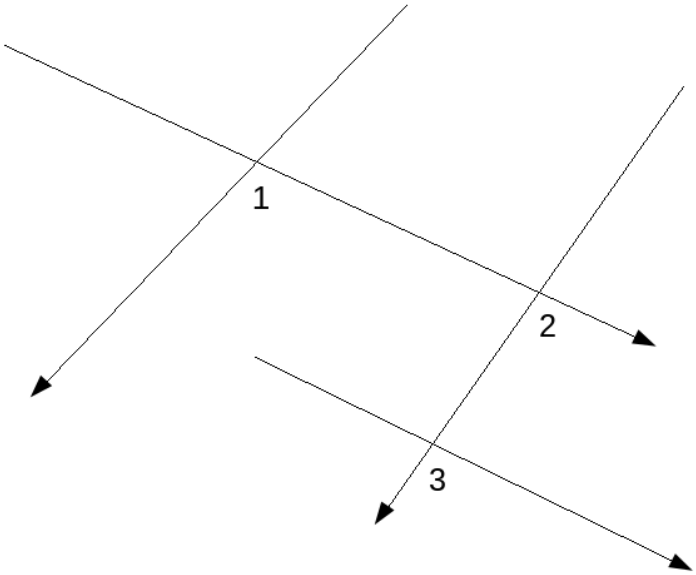
`game` verwaltet eine Spielinstanz. Eine Spielinstanz wird als Spielfeld ausgedrückt, welches sich um alle Spieloperationen kümmert, welche das gesamte Spielfeld erfordern.
Der ***vires***-Server verfolgt einen anderen Ansatz als die meisten Serverapplikationen von Spielen: Anstatt innerhalb eines Game-Loops zu überprüfen, ob bestimmte Bedingungen erfüllt sind, werden alle Aktionen vorberechnet und mittels Timern und einem Scheduler verzögert, bis die Aktion ausgeführt werden soll.





vires ist langsam und der Server soll in der Lage sein, sehr viele Matches gleichzeitig auszuführen. Würde der Server für jedes Match Game-Loops verwenden, so wäre entweder die Bearbeitungszeit sehr schlecht oder eine hohe Überprüfungsfrequenz würde Serverleistung verschwenden.
`room` kommuniziert mit `game`, indem es Methoden von `game` aufruft, wenn Nutzereingaben erfolgen.

`game` kommuniziert mit `room`, indem es Nachrichten über `transm` nach `room` schickt, wenn das Spiel den Spielern etwas mitteilen muss.
Insgesamt verwaltet `game` Movements, Collisions und Conflicts. Jede Aktion auf dem Spielfeld wird über den `timed`-Scheduler ausgeführt, um Zustandszugriffe vom Spiel und von Nutzern zu synchronisieren.
Wird ein Movement gestartet, so wird zuerst geprüft, ob das Movement erlaubt ist. Ist es erlaubt, so wird ein Movement erzeugt, ein Conflict am Zeitpunkt des Conflicts für die Target-Cell geschedulet und mit jedem anderen Movement geprüft, ob es eine Collision gibt.
Gibt es eine Collision, so wird die Collision beiden kollidierenden Movements hinzugefügt und für den Zeitpunkt der Collision geschedulet. Tritt der Conflict auf, so wird das Movement entfernt, die Anzahl an Moving Vires je nach Art des Conflicts der Cell hinzugefügt oder der Cell abgezogen, überprüft, ob die Cell den Besitzer gewechselt hat, überprüft, ob der Besitzer tot ist und geprüft, ob ein Spieler das Match gewonnen hat.
Tritt eine Collision auf, so kämpfen die Movements miteinander oder vereinen sich je nach Art der Collision. Hieraufhin werden alle Collisions beider Movements entfernt und neu berechnet, da die Anzahl an Moving Vires die Größe und die Geschwindigkeit beeinflusst und sich somit auch die Situation für zukünftige Collisions ändert. Stirbt eines der Movements, so wird der Conflict für das jeweilige Movement entfernt.
Es wird also insgesamt beim Erzeugen eines Movements berechnet, welche Collisions nach der aktuellen Situation auftreten können, verändert sich aber die Situation auf dem Spielfeld, so werden erst zum Zeitpunkt der Situationsveränderung die jeweiligen Änderungen an den anliegenden Movements vorgenommen.
Beim Erzeugen der Movements wird zuerst nur davon ausgegangen, dass die auftretenden Collisions alle mit der Startgröße und der Startgeschwindigkeit stattfinden.
Hieraus ergibt sich, dass lediglich die anliegenden Movements verändert werden müssen, und nicht rekursiv alle Movements die nach der aktuellen Collision stattfinden.
Das folgende Szenario ist anzunehmen:



Movements werden als Pfeile dargestellt. Alle vier Movements werden zeitnah hinzugefügt. Beim Hinzufügen aller Movements wird bestimmt, dass mit der Startgeschwindigkeit und dem Startradius des Movements die drei Collisions 1, 2 und 3 in der genannten Reihenfolge stattfinden. Für jede Collision wird für den Zeitpunkt der Collision nach Startgeschwindigkeit und Startradius eine Collision geschedulet.
Zuerst tritt Collision 1 auf. Hieraus ergeben sich drei Fälle für Collision 2:

1. Collision 2 findet nicht mehr statt (Das Movement wurde zerstört)
2. Collision 2 findet schneller statt (Das Movement ist kleiner geworden)
3. Collision 2 finden langsamer statt (Das Movement hat sich mit dem anderen Movement vereint)

In der Folge auf Collision 1 werden alle Collisions der beiden anliegenden Movements aktualisiert.

- Tritt Fall 1 ein, so wird Collision 2 entfernt und Collision 3 findet wie erwartet mit Startgeschwindigkeit und Startradius statt
- Tritt Fall 2 oder 3 ein, so wird Collision 2 stattfinden, und hierbei die Geschwindigkeit und der Radius während Collision 2 angepasst

Es ist also nur nötig, die Collisions, welche mit den kollidierenden Movements verbunden sind, zu aktualisieren, da Fall 1 vom Default abgedeckt wird, während Fall 2 und Fall 3 in dem jeweiligen Zeitpunkt bestimmt werden können. Würde Fall 1 nicht vom Default abgedeckt werden, so müsste man rekursiv alle Collisions aktualisieren, die direkt und indirekt mit den betroffenen Movements verbunden sind, da die Kollisionszeit von Collision 3 aktualisiert werden müsste, und somit auch für alle damit verbundenen Movements.

Dieser Algorithmus verhält sich gegenüber einigen Alternativen wie folgt:

- Feststellen der nächsten Collision beim Hinzufügen jedes Movements und bei jeder Collision ($T(\frac{n\cdot(n-1)}{2})$, bzw. $\mathcal{O}(n^2)$, beim Hinzufügen von Movements und bei Collisions)
- Rekursive Erfassung der Collisions beim Hinzufügen eines Movements und beim Stattfinden einer Collision ($T(\frac{n\cdot(n-1)}{2})$, bzw. $\mathcal{O}(n^2)$, beim Hinzufügen von Movements und bei Collisions)
- Rekursive Erfassung der kompletten Spielsituation beim Hinzufügen eines Movements ($T(\frac{n\cdot(n-1)}{2})$, bzw. $\mathcal{O}(n^2)$, beim Hinzufügen eines Movements)
- Einfache Erfassung der anliegenden Movements bei der Collision und Annahme, dass alle Collisions unverändert stattfinden beim Hinzufügen eines Movements ($T(n)$, bzw. $\mathcal{O}(n)$ beim Hinzufügen eines Movements und bei Collisions)

Die Zustandsraumkomplexität des Algorithmus ist $\mathcal{O}(n^2)$, da sich für jedes Movement bis n Movements gemerkt werden müssen. Es wäre möglich, die Zustandsraumkomplexität weiterhin auf $\mathcal{O}(n)$ zu reduzieren, indem man sich lediglich die nächste Collision eines Movements anstatt alle Collisions eines Movements merkt, was allerdings in anderen Bereichen für größere Kosten sorgen würde und schwerer umzusetzen ist, als die aktuelle Version des Algorithmus.

5.1.4 transm

`transm` enthält alle Typen, welche für die Datenübertragung und die Verbindung zwischen `room` und `game` benötigt werden. Der Transmitter in `transm` sorgt dafür, dass Typen aus `game` in Pakete umkonvertiert werden, welche dann von `room` an die Spieler weitergeleitet werden können. Immer wenn `game` den Spielern etwas mitteilen möchte, geschieht dies über `transm`. Alle anderen Typen in `transm` sind Typen, die ein Paket oder einen Teil eines Pakets repräsentieren, und sowohl zum Senden als auch zum Empfangen verwendet werden. Bei der Datenübertragung werden diese Typen entweder in das Übertragungsformat JSON serialisiert, oder es wird von JSON in diese Typen serialisiert.
`transm` enthält außerdem eine Main-Funktion, welche dazu verwendet werden kann, um ein JSON-Beispiel für das Übertragungsprotokoll zu generieren.

5.1.5 ent

`ent` beschäftigt sich mit den Entities auf einem Feld und kümmert sich um alle Operationen zwischen Entities, die ohne das gesamte Feld ausgeführt werden können.
Als solches kümmert es sich um Operationen auf den Cells, auf Movements und auf Spielern.
Den wohl wichtigsten und komplexesten Teil von `ent` macht die Kollisionsbestimmung zwischen zwei Movements aus. Da die ***vires***-Kollisionsbestimmung eine *priori*-Kollisionsbestimmung ist, damit die

Timer-Architektur des Spiels verfolgt werden kann, müssen Kollisionen im Vorraus mathematisch festgestellt werden.
Wir betrachten die Bewegung des Movements als eine Vektorgerade $g : \vec{x} = \vec{a} + t \cdot \vec{d}$. $\vec{x}$ ist die neue Position des Movements, $\vec{a}$ die Ausgangsposition des Movements, t die vergangene Zeit und $\vec{d}$ der Richtungsvektor des Movements, wobei $|d| = \sqrt{{d_x}^2 + {d_y}^2}$ die Geschwindigkeit des Movements repräsentiert. Die Cells können als Kreise ausgedrückt werden. Für Kreise gilt, dass die Differenz zwischen einem Punkt auf dem Außenkreis und dem Mittelpunkt genau dem Radius entsprechen muss, es gilt also $c : r = |\vec{u} - \vec{m}|$. Betrachtet man dann den Mittelpunkt des Kreises als Gerade, so ergibt sich $r = |\vec{u} - \vec{a} - t \cdot \vec{d}|$.
In ***vires*** kollidieren zwei Movements, wenn sich der Mittelpunkt des kleineren Movements im größeren Movement befindet, also wenn $|m_1 - m_2| < r_1$. Vereinfacht interessieren uns aber nur die Schnittpunkte des Mittelpunkts des kleineren Movements mit dem Außenkreis des größeren Movements, da das kleinere Movement nach der Collision verschwindet. Es ist also nicht wichtig, dass $|m_1 - m_2| < r_1$, da uns nicht interessiert, über welchen Zeitraum sich das kleinere Movement im größeren Movement befindet. Diese Vereinfachung ermöglicht es uns, das Problem auf folgendes zu reduzieren: Wir wollen die Schnittpunkte der Vektorgerade des kleineren Movements $g : \vec{x} = \vec{a} + t \cdot \vec{d}$ mit dem Außenkreis des größeren Movements $c : r = |\vec{u} - \vec{m}|$ bestimmen. Um das Problem weiter zu vereinfachen, betrachten wir die Bewegung des kleineren Movements relativ zu dem des größeren Movements, anstatt beide Bewegungen parallel zu betrachten. Hieraus ergibt sich die Gleichung $\vec{x_r} = \vec{x_1} - \vec{x_2}$, wobei $\vec{x_1}$ das größere der beiden Movements darstellt. Insgesamt muss also diese relative Vektorgerade den Außenkreis des größeren Movements, welches sich aufgrund der relativen Betrachtung in $m(0,0)$ befindet, schneiden. Dies ergibt $r = |\vec{x_r} - \vec{0}|$, bzw. $r = |\vec{x_r}|$. Löst man nun die Substitution auf, so ergibt sich $r = |\vec{x_1} - \vec{x_2}|$ und hieraufhin $r = |\vec{a_1} + t \cdot \vec{d_1} - (\vec{a_2} + t \cdot \vec{d_2})|$, bzw. $r^2 = (\vec{a_1} + t \cdot \vec{d_1} - (\vec{a_2} + t \cdot \vec{d_2}))^2$. Stellt man diese Gleichung mittels quadratischer Ergänzung für Skalarprodukte und Substitution für Distanzen um, so erhält man:

$$i_{1,2} = \pm \sqrt{\frac{(\vec{m_1} - \vec{m_2}) \cdot (\vec{d_1} - \vec{d_2})}{|\vec{d_1} - \vec{d_2}|}^2 - (m_{1_x} - m_{2_x})^2 - (m_{1_y} - m_{2_y})^2 + (r_1 \vee r_2)^2} - \frac{(\vec{m_1} - \vec{m_2}) \cdot (\vec{d_1} - \vec{d_2})}{|\vec{d_1} - \vec{d_2}|}$$

Mit dieser Funktion lässt sich dann berechnen, nach welchen Strecken die Collision stattfindet.

1. Sind i_1 und i_2 beide positiv, so findet die Collision bei der kürzesten der beiden Strecken statt: $t_c = \frac{i_1 \wedge i_2}{|\vec{d_1} - \vec{d_2}|}$
2. Sind i_1 und i_2 beide negativ, so fand die Collision in der Vergangenheit statt
3. Ist eine der beiden Strecken positiv und die andere negativ, so findet die Collision in diesem Moment statt

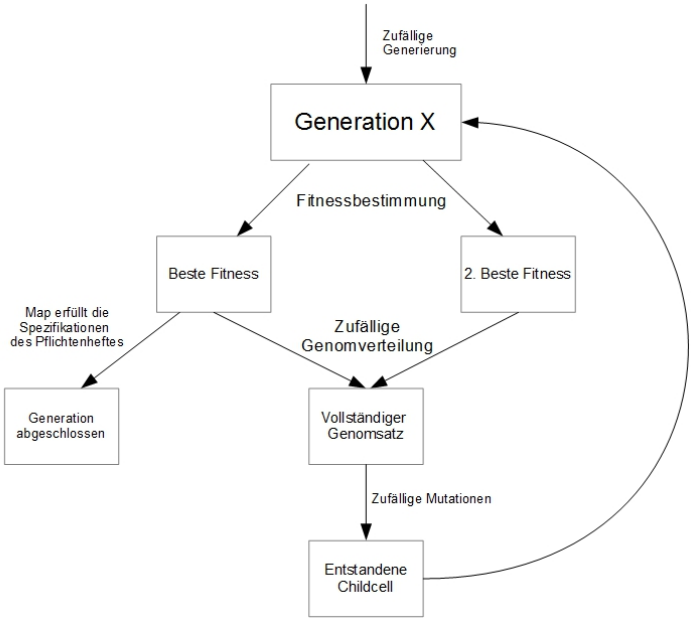
Eine Visualisierung dieser Berechnung kann hier gefunden werden:
http://tube.geogebra.org/material/simple/id/2757139

5.1.6 timed

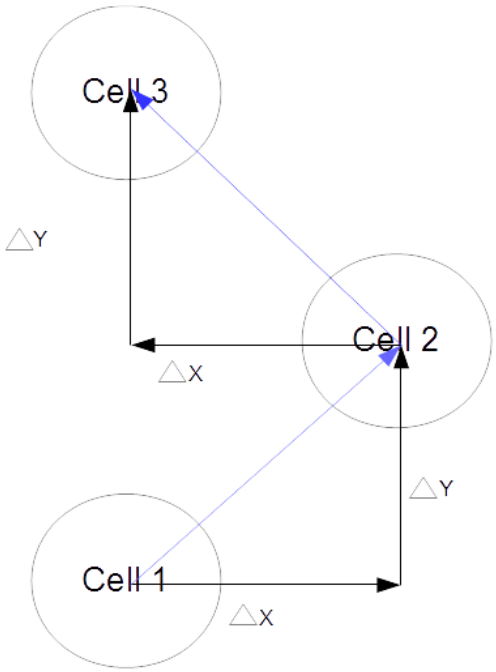
`timed` enthält einen Scheduler für Timer, welcher in ***vires*** verwendet wird, um alle möglichen Aktionen in der Zukunft auszuführen und das Timer-basierte Modell umzusetzen.
Um geschedulte Funktionen auszuführen, würde normalerweise für jede Aktion eine Goroutine benötigt. Jede Goroutine ist etwa 2KiB schwer, was sehr teuer für jede Aktion im Spiel wäre. Dieses Problem löst `timed`, indem es beliebig viele Timer auf einer Goroutine hintereinander ausführt. Um dies zu erreichen, müssen die Timer geordnet, nacheinander ausgeführt und bei Bedarf unterbrochen werden.
Bei dem Scheduler handelt es sich um einen präemptiven Shortest-Time-Remaining-Scheduler, welcher den Garbage Collector, den Arbeitsspeicher und die CPU deutlich weniger belastet, als es mit einer Goroutine für jede Aktion der Fall wäre.
Der Scheduler verwendet eine Monitor-Goroutine, um Zugriffe auf sich selbst zu synchronisieren. Die Monitor-Goroutine wird über einen Actions-Channel angesprochen, welcher verwendet wird, um Zustandsveränderungen an dem Scheduler vorzunehmen.
Am Anfang jeder Iteration führt die Goroutine so lange Actions aus, bis Timer in der Liste sind, die geschedulet werden können. Hieraufhin wird der Timer gestartet und mittels eines select-Statements gleichzeitig auf das Fertigwerden des Timers und Aktionen zum Ausführen gewartet. Führt eine Aktion dazu, dass der aktuell laufende Timer ersetzt werden muss, so wird der aktuelle Timer gestoppt und es werden wieder so lange Actions ausgeführt, bis ein Timer, der geschedulet werden kann, vorhanden ist. Wird ein Timer fertig, so wird seine Aktion ausgeführt, der Timer entfernt und auf den nächsten Timer gewartet.

5.1.7 mapgen

`mapgen` enthält einen Algorithmus zur Generation von Maps, welche die vorher festgelegten Spezifikationen des Pflichtenheftes erfüllen. Maps sollten zuerst mithilfe eines Force-Directed Graphs generiert werden, wovon jedoch schnell, zum einen bedingt durch die Komplexität des Algorithmus, zum anderen durch die mangelnde Erfüllbarkeit der festgelegten Spezifikationen, abgewichen wurde. Eine weitere Möglichkeit bestand darin, einen zufälligen Algorithmus zu verwenden, welcher die Maps zufällig generiert. Die Problematik darin liegt allerdings in der Laufzeit begründet. Ein zufälliger Algorithmus kann in sehr kurzer Zeit zu einem brauchbaren Ergebnis führen, jedoch leidet die Effizienz extrem und die Generation kann zu viel Zeit in Anspruch nehmen.
Schlussendlich wurde sich für die Anwendung eines genetischen Algorithmus entschieden. Dieser bietet die Zufälligkeit einer zufälligen Generierung, allerdings auch ein systematisches Verfahren, welches sich einer perfekten Lösung immer weiter annähert, diese aber mit großer Wahrscheinlichkeit nie erreichen wird.



Ein genetischer Algorithmus arbeitet nach dem Prinzip der Evolution. Dabei werden in der 1. Generation (im Falle von ***vires*** besteht jede Generation aus 8 verschiedenen Maps) alle Maps zufällig erzeugt und für jede Map wird eine den Spezifikationen des Pflichtenheftes entsprechende Fitness zugewiesen. Hat in einer Generation noch nicht mindestens eine Map die Spezifikationen erfüllt, so werden die beiden Maps mit der besten Fitness gekreuzt. Dies geschieht wie in der Genetik zufällig, was bedeutet, dass die Childmap eine zufällige Zahl von Genomen von Vater und Mutter erhält. Um weiterhin eine gewisse Zufälligkeit im Algorithmus zu gewährleisten, und um weiterhin zu garantieren, dass sich der Algorithmus nicht in einer Endlosschleife verfängt, werden bei jeder Kreuzung zu einer gewissen Wahrscheinlichkeit Mutationen eingebaut, welche ein Genom zufällig verändern. Im Fall von ***vires*** wird mit einer gewissen Wahrscheinlichkeit eine Cell aus der Map entfernt und es wird eine neue, zufällige Cell hinzugefügt. Anschließend wird für die entstandende Childcell eine eigene Fitness berechnet. Dieser Prozess wird solange wiederholt, bis eine Map die Spezifikationen erfüllt und somit als Map zum Spielen von ***vires*** verwendet werden kann. Auch bei einem genetischen Algorithmus kann die benötigte Zeit zur Generierung einer Map stark vom Zufall abhängen, jedoch bei weitem nicht so ausgeprägt wie bei einem vollständig zufälligen Algorithmus.



Die Errechnung der Fitness einer Map erfolgt im Fall von ***vires*** sowohl über die Distanzen zwischen den Cells zueinander, als auch über die Distanzen zwischen dem Mittelpunkt der Map und den Cells. Dies hat den Vorteil, dass die Bildung eines leeren Kreises in der Mitte der Map verhindert wird. Unterschreiten die Distanzen zwischen den Cells einen gewissen Schwellwert(beispielsweise Cells kollidieren bei der Generation miteinander), so wird die Fitness der Map auf 0 gesetzt und sie ist somit ungeeignet für die weitere Verwertung. Andernfalls errechnet sich die Fitness der Map aus der minimalen Distanz einer Cell zu ihrem nächsten Nachbarn, und der maximalen Distanz einer Cell zu ihrem nächsten Nachbarn. Die Errechnung der Distanzen erschließt sich aus dem Satz des Pythagoras und erfolgt folgendermaßen:

$$D_{Cells} = \pm\sqrt{\Delta X^2 + \Delta Y^2}$$

Wobei gilt:

$$\Delta X = Cell1_x - Cell2_x$$

und

$$\Delta Y = Cell1_y - Cell2_y$$

Für die Distanz vom Map Mittelpunkt zur nächstgelegenen Cell gilt:

$$D_{Middle} = \pm\sqrt{(Mid_x - Cell_x)^2 + (Mid_y - Cell_y)^2}$$

Die Fitness einer Map wird nun wie folgt berechnet:

$$Fitness = \frac{D_{CellsMin}}{D_{CellsMax}} \cdot 1000 - \frac{D_{CellToMid}}{D_{MidToBorder}} \cdot 100$$

Nach dieser Berechnung kann eine Map eine maximale Fitness von 1000 erreichen. Eine solche Map wäre nach der gegebenen Kalkulation quasi perfekt. Im Falle von ***vires*** wird eine Fitness von mindestens 500 gefordert, damit die Map als spielbar eingestuft wird.

5.1.8 vec

`vec` enthält Funktionen für zweidimensionale Vektormathematik.

5.2 Frontend

6 Erkenntnisgewinn

6.1 Backend

6.1.1 Go

Go hat sich als Backend-Sprache als praktisch erwiesen, da das Ökosystem und Go's Unterstützung für Goroutinen und Channels viele Dinge vereinfacht haben.
Die Menge an Sourcecode, welcher nötig war, um das Backend zu entwickeln, war deutlich geringer, als es beispielsweise in Java der Fall gewesen wäre, da Go deutlich weniger Verbosität aufweißt, als es in Java der Fall wäre.
Gorilla hat sich ebenfalls als sehr nützliches Framework gezeigt. Der Grund hierfür ist hauptsächlich, dass Gorilla nicht versucht, mittels Inversion Of Control dem eigenen Code die Kontrolle zu entreißen, was bedeutet, dass für die Nutzung des Frameworks lediglich Go-Code und keine externen Tools benötigt werden. Der Lernaufwand ist also deutlich geringer - und der entstandene Code liest sich, als würde man eine Library verwenden. Der Webserver, das Routing des Webservers und die Verwendung von Websockets haben sich deshalb als sehr einfach erwiesen.
Die Verwendung von Goroutinen und Channels hat sich in einigen Bereichen unseres Projekts als sehr nützlich erwiesen. Für die Verbindungen der User, den Scheduler und die Verbindung zwischen verschiedener Komponenten des Programms stellen Channels eine sehr gute Lösung dar.
Ist das Problem allerdings zustandsorientiert in seiner Natur, so ist die Verwendung von Channels nicht mehr viel praktischer als die Verwendung von Mutexes. Für ein solches Problem gibt es drei mögliche Lösungen: Entweder man verwendet ein Lock, man verwendet eine Monitor-Goroutine oder man verwendet einen Channel als Lock. Die erste Lösung ist wahrscheinlich die performanteste, da keine extra Goroutine zur Synchronisation benötigt wird, während sich bei der Verwendung einer Monitor-Goroutine alle synchronisierten Operationen an einer Stelle im Code befinden, was die Synchronisation relativ leicht überblickbar macht.
Die Timer-Architektur hat sich dagegen als nicht wirklich erfolgreich gezeigt. Da wir noch nicht in der Lage waren, den Server wirklich unter einer großen Load zu testen, können wir keine wirkliche Aussage über die wirkliche Performanz dieses Ansatzes treffen.
Klar ist allerdings, dass diese Architektur deutlich komplizierter zu entwickeln ist und einen stark in seiner Entwicklung einschränkt. Da alle Kollisionen im

Voraus bestimmt werden müssen, muss ein Priori-Kollisionsalgorithmus verwendet werden, welcher deutlich komplizierter als die meisten Posteri-Kollisionsalgorithmen ist. Für den Priori-Kollisionsalgorithmus mussten wir unseren eigenen Algorithmus entwickeln, während wir mit einer Posteri-Kollisionsdetektion lediglich einen Quad-Tree hätten verwenden müssen. Auch das Scheduling von Timern ist nicht einfach, und besonders die mathematische Bestimmung der Kollisionszeit ist ziemlich kompliziert. Hinzu kommt noch, dass ein Priori-Kollisionsalgorithmus die Möglichkeit verhindert, geometrische Objekte zu verwenden, die nicht leicht mithilfe von Vektoren ausgedrückt werden können. Sich bewegende Kreise sind noch vergleichsweise einfach zu implementieren, während sich rotierende Dreiecke ein, im Bezug auf die Performanz, fast unlösbares Problem darstellen würden.
Trotzdem hat ein Priori-Kollisionsalgorithmus den Vorteil, dass es nicht passieren kann, dass sich Movements innerhalb eines Frames kreuzen können, ohne dass eine Kollision festgestellt wird, wenn sich die Movements zu schnell bewegen.
Insgesamt würden wir diese Entscheidung aufgrund ihren limitierenden Implikationen nicht erneut treffen.
Während dem Schreiben des Programms wurden verschiedene Go-Datentypen gebenchmarkt. Im Spiel gibt es überall Listen, die ungeordnet sind, bei denen aber auch trotzdem oft Lookups, Inserts und Removes erfolgen. Gebenchmarkt wurden Slices und Maps, sowie "generische" Versionen dieser Typen mithilfe von `interface{}` (was im Grunde `Object` in Java entspricht).
In der Theorie sollte die Iteration über Maps langsamer von statten gehen als bei Slices, da es bei der Iteration zu Cache-Misses kommt.
Das Lookup und Inserten sollte dagegen bei Slices langsamer sein.
Bei Removes sollten sich Slices und Maps ähnlich verhalten, da die Ordnung der Slices nicht eingehalten werden muss.
Zudem sollten die Versionen mit `interface{}`-Typen deutlich langsamer sein als ihre konkreten Versionen, da es hierbei bei jedem Zugriff zu Cache-Misses kommt und Casts zu den konkreten Typen auch nicht unbedingt billig sind.
Im Folgenden sind USlices unsortierte Slices mit `interface{}`, Slices sind unsortierte Slices mit einem konkreten Typen, Set sind Maps mit `interface{}` als Key und CoherentSet sind Maps mit konkreten Typen als Key.
Die Benchmarks haben das Folgende ergeben:

Benchmark	iterations	time/iteration
BenchmarkUsliceAdd	10000000	149 ns/op
BenchmarkSetAdd	3000000	465 ns/op
BenchmarkSetLookup	10000000	363 ns/op
BenchmarkUsliceLookup	10000	831147 ns/op
BenchmarkCoherentSetLookup	20000000	88.1 ns/op
BenchmarkSliceLookup	200000	170624 ns/op
BenchmarkUsliceFilter	100000000	144 ns/op
BenchmarkSetFilter	5000000	322 ns/op
BenchmarkSliceFilter	300000000	7.09 ns/op
BenchmarkCoherentSetFilter	10000000	101 ns/op
BenchmarkUsliceRemove	10000	511529 ns/op
BenchmarkSetRemove	10000000	457 ns/op
BenchmarkSliceRemove	200000	45687 ns/op
BenchmarkCoherentSetRemove	20000000	167 ns/op