vires: Dokumentation

Marc Huisinga Vincent Lehmann Steffen Wißmann

28. Februar 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.
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.

5 Programmaufbau

5.1 Backend

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

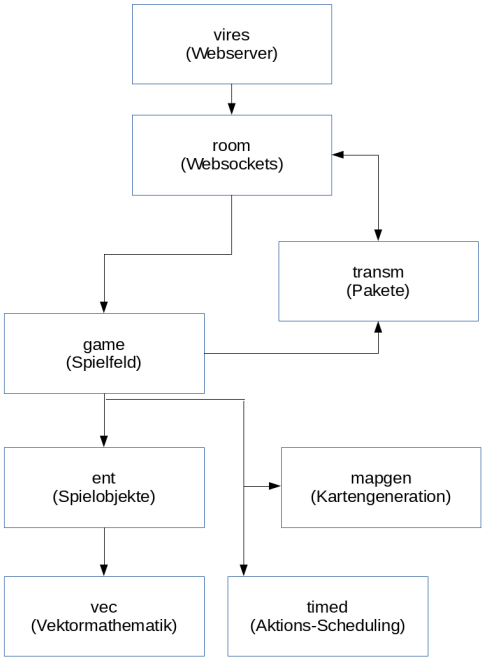


Abbildung 1: Paketaufbau des Backends

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.

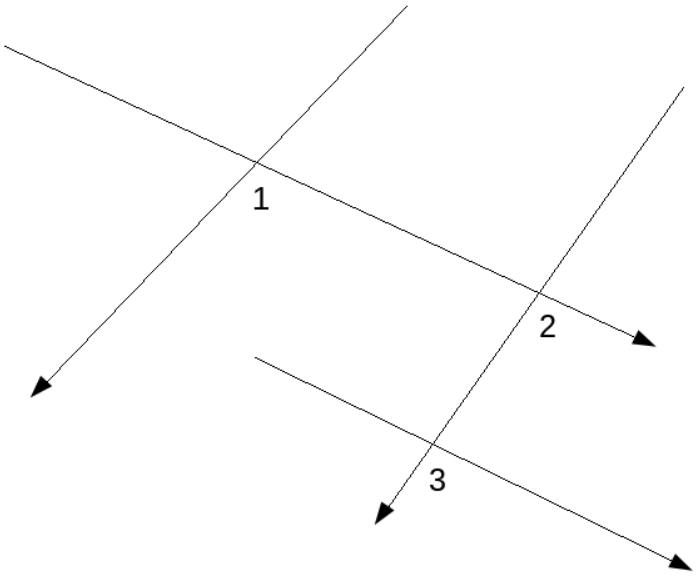
5.1.2 room

`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.
Genauere Informationen über Monitor-Goroutines können unter `MONITOR_GOROUTINE_REF_HIER_EINFÜGEN` gefunden werden.
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 ERKLÄRUNG HIER

5.1.8 vec

`vec` enthält Funktionen für zweidimensionale Vektormathematik.

5.2 Frontend

6 Erkenntnisgewinn

7 Zielreflexion

8 Qualitätsreflexion

9 Projektreflexion