Kill the Virus!

https://webtech.mylab.th-luebeck.de/ss2020/team-04-d/

Marvin Bergmann, Ramona Kalkmann

Mit diesem Spiel nehmen wir an der Wahl zur Hall-of-Fame teil.

Inhaltsverzeichnis

Kapitel 1

Einleitung

'Am Beispiel einer Spielentwicklung sollen sie eine Auswahl relevanter Webtechnologien wie bspw. **HTML**, **DOM-Tree**, **HTTP** Protokoll, **REST**-Prinzip, sowie die Trennung in **client-** und **serverseitige Logik** spielerisch kennenlernen. Das Spiel selber ist dabei weitestgehend clientseitig zu realisieren und soll nur für die Speicherung von Spielzuständen, wie bspw. Highscores, auf REST-basierte Services zugreifen. Selbst ohne Storagekomponente soll ihr Spiel spielbar sein (einzig und allein das Speichern von Spielzuständen ist natürlich eingeschränkt).' -Auszug aus Referenzdokumentation für das Modul 'Webtechnologie Projekt'

Um auf geeignete Spielideen zu kommen haben wir uns an den Rat von Prof. Dr. Kratzke gehalten und uns auf Spieleseiten im Internet umgesehen um uns inspirieren zu lassen. Das Spielkonzept und die ausgesuchten Anforderungen an das Spiel werden nun im nächsten Kapitel beschrieben.
Alle hier beschriebenen Details können sich im Laufe der Entwicklung ändern. Auch Komplexitätspunkte können im Laufe des Projekts abweichen.

Kapitel 2

Anforderungen und abgeleitetes Spielkonzept

2.1 Gewählte Anforderungen und Komplexitätspunkte

Die Anforderungen, die hier beschrieben werden orientieren sich an der Anforderungsliste und den dazugehörigen Komplexitätspunkten, die Prof. Dr. Kratzke uns zur Auswahl gegeben hat:

- **Target Device:** Das Spiel soll für den Mobile Browser optimiert, aber auch im Webbrowser spielbar sein.
  - Desktop und Browser (Aber Mobile-First Ansatz): 3 Punkte
- **Steuerung:** Gelenkt wird die Spielfigur durch eine Gyrosteuerung und ein Tippen auf dem Screen zum Springen. Im Desktopbrowser wird diese Steuerung von der Tastatur übernommen, wo man sich mit den Pfeiltasten nach rechts / links bewegt und mit der Leertaste springt.
  - Gyrosteuerung: 3 Punkte
- **Darstellung:** Das Level bewegt sich fließend von rechts nach links.
  - Absolut positionierte Elemente (aber ohne Canvas): 2 Punkte
- **Levelsystem:** Levellayouts, die vorher festgelegt und abgespeichert sind und die dann bei Bedarf geladen werden.
  - Komplexere Level-Definitionen (z.B. Labyrinth-Definition): 2 Punkte

2.2 Tabelle: Abdeckung der Komplexitätspunkte

Targets	**Augewählte Komplexitätspunkte**	**Beschreibung und eingebrachte Komplexitätspunkte**
Target Device	3	**3**: Desktop & Browser (Mobile first): Das Spiel ist optimiert für den mobile Browser, ist aber auch im Webbrowser spielbar
Steuerung	3	**3**: Das Spiel ist steuerbar über eine Gyrosteuerung, wenn man es im mobile Browser spielt und über Tasten, wenn man es im Webbrowser spielt.
Darstellung	2	**2**: Das Level bewegt sich fließend von rechts nach links, wenn man sich durch das Level bewegt.
Levelsystem	2	**2**: Komplexere Level-Definition mit Rätseln und labyrinthartigen Aufbauten

2.3 Spielkonzept

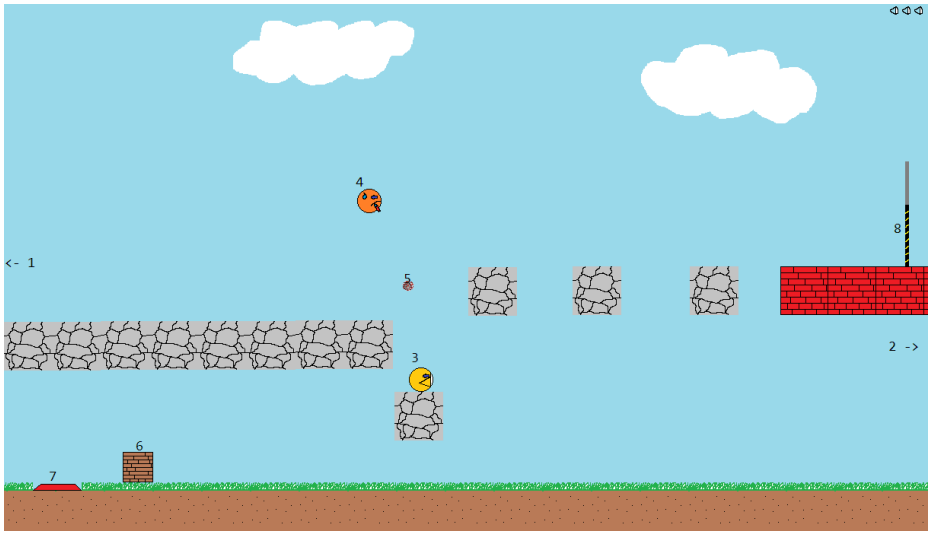

Abbildung 2.1: Levelaufbau

Das Spiel „Kill the Virus“ wird ein 2-dimensionales, browserbasiertes Jump & Run Spiel mit einem Spielfeld variabler Größe (levelabhängig). Die nachfolgende Erklärung des Konzepts orientiert sich an der Abbildung 2.1.
Das Spiel soll ein horizontales Leveldesign besitzen, in dem man sich von Links (**1**) nach Rechts (**2**) vorarbeiten muss um an das Ziel zu gelangen, wird aber wie in vielen klassischen 2-dimensionalen Jump & Run Spielen eine gewisse vertikale Komponente besitzen (z.B. verschiedene Ebenen, Objekte zum draufspringen, Hindernisse welche übersprungen werden müssen, e.t.c.).

Dabei spielt man einen kreisförmigen „Blob“ (**3**) welchen man durch das Level bewegt um so zum Ziel zu gelangen und das nächste Level freizuschalten. Auf dem Weg werden dem Spieler

Rätsel und Hindernisse begegnen welche es zu überwinden gilt, z.B. Elemente wie Schalter, Kisten oder automatische Türen. Im Bild 2.1 muss man bspw. die Kiste, mit der Kennzeichnung (**6**) auf den Schalter mit der Kennzeichnung (**7**) schieben. Dieser Vorgang öffnet dann eine Tür (**8**), durch die der Spieler in das nächste Level entkommen kann, oder die es alternativ erlaubt ein nicht überwindbares Hindernis zu überwinden.

Außerdem gibt es andere, mit dem Coronavirus infizierte „Blobs“ (**4**), welche mit diesen Viren (**5**) auf den Spieler schießen werden, um ihn zu infizieren. Diese KI wird den Spieler aktiv verfolgen und in einem vorgegebenen Takt den Virus schießen , sodass der Spieler beim Lösen der Rätsel diesen ausweichen muss. Pro Abschnitt wird es 1-2 Gegner geben. Hat der Spieler die Rätsel gelöst öffnet sich eine Tür, die gleichzeitig die Gegner aussperrt und durch die man in den nächsten Abschnitt gelangen kann.
Das Ziel ist ein Impfstoff gegen den Virus. Wird der Spieler getroffen, so verliert er einen von insgesamt 3 Lebenspunkten. Sind alle 3 Lebenspunkte (Atemmasken, oben rechts in der Abbildung) verbraucht, stirbt der Spieler an Corona und muss das Level erneut versuchen.
Das Spielkonzept und dessen technische Umsetzung werden modular aufeinander und unabhängig voneinander Aufgebaut, so dass das Spiel leicht erweiterbar und abwandelbar ist (z.B. ist die Spielengine selbst komplett unabhängig vom Leveldesign, so dass beliebig aufgebaute Level spielbar sein werden).
Durch lokal storgae wird der Fortschritt im Spiel gespeichert, sodass der Spieler beim nächsten Mal an der Stelle weiterspielen kann, wo er aufgehört hat.

2.4 Levelkonzept

Die einzelnen Level sind so aufgebaut, dass die von Mal zu mal schwerer werden. Beispiele hierfür sind:

- **Länge und Komplexität:** Je weiter man kommt, desto länger und komplexer werden auch die Level. Man muss mehr Rätsel lösen und über mehrere Ebenen gehen.

- **Gegner:** Die Anzahl der Gegner kann sich, je weiter man kommt, erhöhen und so für eine höhere Schwierigkeit sorgen.

- **Leveldesign:** Das Design der Level kann sich verändern, bspw der Hintergrund und die dazugehörigen Blöcke.

- **Andere Rätsel:** Die Art der Rätsel kann variieren.

- **Level:** Neues Level wird erst freigeschaltet, wenn aktuelles Level gelöst wurde.

Kapitel 3

Architektur und Implementierung

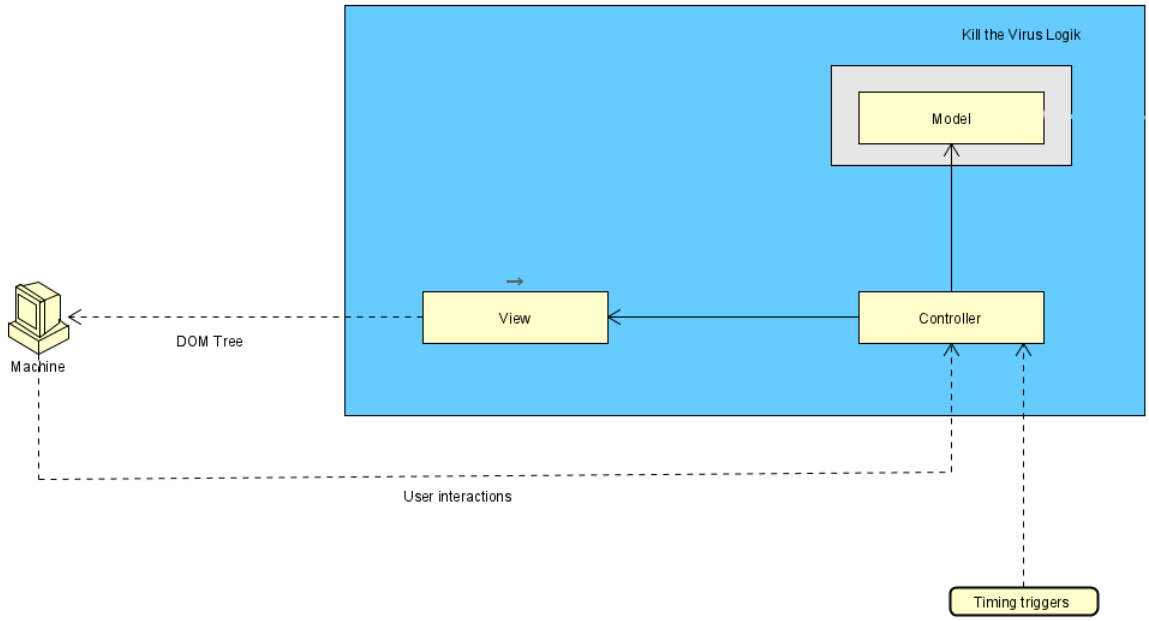

Abbildung 3.1: Architektur

Abbildung 3.1 zeigt die Architektur unseres Spiels im Überblick. Sie ist MVC- basiert, also nach dem Model-View-Controller-Prinzip aufgebaut. Model, View und Controller werden nachfolgend detailliert in den Kapiteln 3.1, 3.2 und 3.3 erläutert.
(Zur Übersicht sind die Methoden- und Variablennamen an den Stellen an denen sie ausführlich beschrieben werden, fettgedruckt dargestellt und normal, falls sie nur erwähnt werden. Zudem werden ein paar Variablen und Methoden nicht beschrieben, falls diese keinen großen Einfluss auf die Architektur haben, da diese noch e.v.l. durch eine alternative Technik ersetzt werden)

Alle Positionsangaben und Größen, welche eine Richtung besitzen (Geschwindigkeit, Beschleunigung und Richtung) werden durch die Klasse **Vector** als 2-dimensionale Vektoren abgebildet. Die Klasse Vektor definiert dazu geeignete Getter- und Settermethoden, sowie arithmetische Operationen.

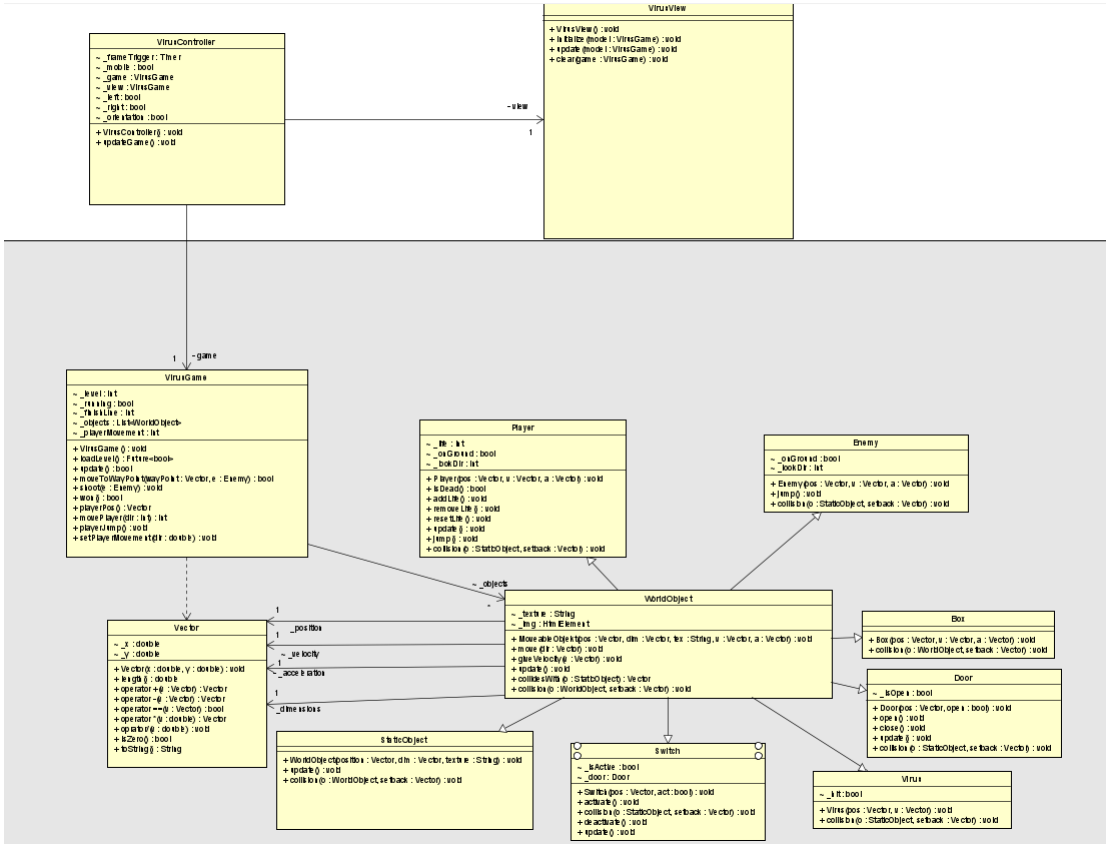

Abbildung 3.2: Klassendiagramm

3.1 Model

Abbildung 3.3 zeigt das Klassendiagramm des Models. (Einfache Getter und Setter werden hier nicht dargestellt und in dieser Doku auch nicht beschrieben, lediglich erwähnt, wenn sie aufgerufen werden. Für weitere Informationen, siehe Quelltext)

Die logische Darstellung des Spiels wird durch $\mathbb{Z}^2$ realisiert. Die Berechnungen der Objekte erfolgt durch Anlehnung an die newtonschen Axiome der Physik (ohne Berücksichtigung von Masse). Das heißt, jedes Objekt hat zu jedem Zeitpunkt eine bestimmte Position im Raum. Sollen Objekte Beweglich sein, so besitzen sie zudem eine Geschwindigkeit und eine Beschleunigung (welche natürlich auch 0 sein können). Jedes Mal, wenn ein neues Frame berechnet wird, wird nun die Beschleunigung vektoriell zur Geschwindigkeit addiert und dann die Geschwindigkeit zur Position. Es gibt allerdings eine Geschwindigkeitsbegrenzung (sozusagen die Lichtgeschwindigkeit in unserem Universum ;)), da sonst bei der Kollisionsberechnung Objekte mit zu hoher Geschwindigkeit durch zu 'dünne' Objekte durchtunneln könnten, da sich die Position zwischen 2 Frames sprunghaft verändert (eine kontinuierliche Implementation würde zu hohen Rechenaufwand bei vergleichsweise geringem Gewinn für dieses Spiel bedeuten).
Alle Objekte (und deren Hitboxen) werden als Rechtecke abgebildet (selbst wenn diese Grafisch am Ende z.B. rund dargestellt werden), da dies für dieses Spiel ausreichend ist (und bei einer Webanwendung, welche auf einem Smartphone laufen soll, sollte man auch den Rechenaufwand bedenken).

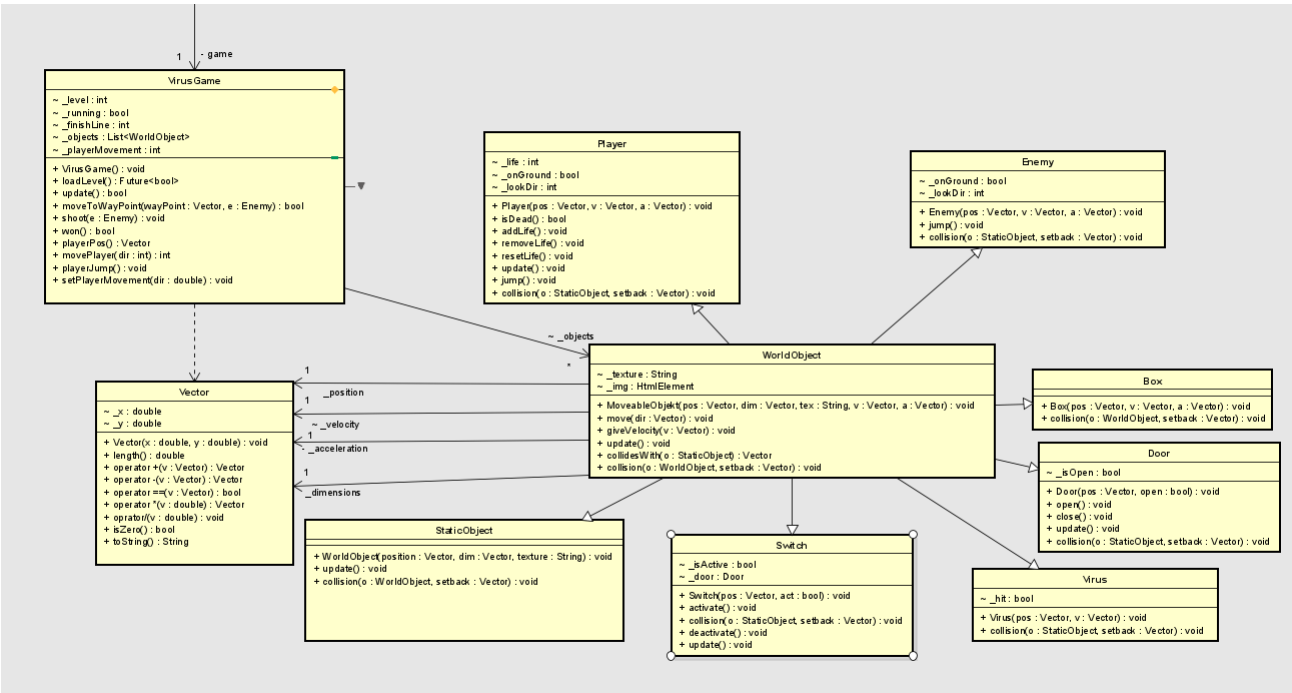

Abbildung 3.3: Model

Die Klasse **WorldObject** ist die Basisklasse aller Objekte, welche sich in der Spielwelt befinden und bildet diese wie folgt ab:

- **_position** speichert die aktuelle Position des Objektes
- **_dimensions** speichert die Höhe und Breite
- **_velocity** speichert die momentane Geschwindigkeit
- **_acceleration** speichert die momentane Beschleunigung, welche auf das Objekt wirkt
- **_texture** speichert welches Bild für die Darstellung dieses Objektes verwendet werden soll (Da für viele Objekte nicht unterschieden wird, was genau sie darstellen. Ein Grasblock ist beispielsweise vom selben Typ WorldObject wie ein Steinboden)
- **move()** nimmt einen Vektor und addiert diesen zur Position
- **giveVelocity()** nimmt einen Vektor und addiert diesen zur aktuellen Geschwindigkeit
- **update()** prüft auf Überschreitung der Maximalgeschwindigkeit und begrenzt gegebenenfalls die Geschwindigkeit. Dann wird diese zur Position addiert und dann die Beschleunigung zur Geschwindigkeit. (Beschleunigungen treten also um einen Frame verzögert in Kraft. Dies ist notwendig für den Algorithmus welcher im Falle einer Kollision die Objekte zurücksetzt; siehe collidesWith()). Zudem können Objekte mit eigenem Verhalten diese Methode überschreiben und so dieses Verhalten für jedes Frame umsetzen.
- **collidesWith()** nimmt eine Referenz auf ein Objekt und überprüft, ob eine Kollision mit diesem stattgefunden hat (ob sich die beiden Objekte überschneiden). Dafür wird für alle vier Eckpunkte des Objektes abgefragt, ob sich dieser innerhalb des Rechteckes des übergebenen Objektes befindet. Falls dem so ist, wird die kürzeste Entfernung zur Kante des Objektes berechnet und als Vektor übergeben. So kann die Spielengine (siehe VirusView) das Objekt an die am nächsten liegende Kante des Objektes zurückgesetzt werden.
  Dies kann aber zu folgendem Problem führen: bewegt sich z.B. ein Objekt langsamer nach

oben als nach rechts, kollidiert aber nahe der oberen linken Ecke mit einem Objekt von rechts, so kann es passieren, dass die obere Kante näher am Punkt liegt, als die rechte und das Objekt somit and diese gesetzt wird, anstatt an die linke Kante. Da dies bei geringen Geschwindigkeiten aber selten auftritt, und das Spiel nicht großartig negativ beeinflusst, wird dies (erst einmal) hingenommen.
Das wird allerdings zu einem Problem bei Beschleunigungen, welche das Objekt konstant an ein Objekt 'pressen', wie z.B. die Gravitation. Bewegt sich nun das Objekt über zwei Blöcke, welche bündig an der Kante, an welche das Objekt gepresst wird aneinander anschließen, entsteht beschriebene Situation. Um dies zu verhindern müssen Kräfte außer Kraft gesetzt werden, falls diese das Objekt wegen Kollision nicht weiter beschleunigen können. Dazu wird jedes Frame, wenn eine Kollision in diese Richtung stattfindet, der Geschwindigkeitsanteil in diese Richtung auf 0 gesetzt. Damit dies aber einen Effekt hat, muss wie in update() beschrieben, die Beschleunigung nach Neuberechnung der Position auf die Geschwindigkeit addiert werden und somit ein Frame verzögert einsetzen.
Wird nun die Position neu berechnet und eine Kollision in die Richtung festgestellt, in die auch eine Beschleunigung wirkt, so wird das Objekt zurückgesetzt und dessen Geschwindigkeit in diese Richtung auf 0 gesetzt. Dies überschriebt dann den Effekt der Beschleunigung, der dann im nächsten Frame nicht mehr zu tragen kommt. Damit dann aber im übernächsten Frame nicht die Beschleunigung wieder einsetzt, falls weiterhin Kontakt mit der Kante besteht, muss dieser Kontakt festgestellt werden, ohne dass dies ein Zurücksetzen des Objektes zur Folge hat.
Dies wird ermöglicht, da eine Kollision bereits erkannt wird, wenn ein Eckpunkt auf der Kante selbst sitzt. Da aber der oben erwähnte Vektor zur nächsten Kante dann 0 ist, wird die Position des Objektes dennoch nicht weiter durch die Kollision verändert.
Das hier beschriebene Behandeln des Kollisionsfalles wird aber in der Methode update() von VirusGame übernommen. Diese Methode gibt im Falle einer Kollision lediglich den Vektor zur nächsten Kante und im anderen Fall den Nullvektor zurück.

- **collision()** Diese Methode bekommt von VirusGame ein WorldObject, sowie den dazu gehörigen Kollisionsvektor (siehe collidesWith()) übergeben und updatet das Objekt im falle einer Kollision. Findet eine Kollision in Richtung einer Beschleunigung statt, wird diese nullifiziert (Für eine detailliertere Beschreibung siehe den Abschnitt collidesWith() weiter oben).
  Hat eine Kollision eines Virus mit dem Spieler stattgefunden, wird die Referenz auf auf das Virusobjekt aus _objects entfernt und dem Spieler wird ein Lebenspunkt abgezogen. Anschließend wird jeder Gegner und der Spieler ein Stück nach unten versetzt und es wird eine zweite Kollisionsabfrage gemacht, um zu testen, ob nun eine Stattfindet. Ist dem so, bedeutet das, das Objekt befindet sich über solidem Boden und es wird _onGound gesetzt damit das Objekt wieder springen kann.
  Damit wird zwar kein Bodenkontakt erkannt, falls das Objekt bereits vorher auf dem Boden stand (da ja nur eine Versetzung stattfindet, wenn das Objekt vor der Verschiebung nicht mit dem momentan abgefragten Objekt kollidiert ist), was aber keinen Einfluss hat, da das bedeutet, das Objekt musste vorher bereits auf den Boden gefallen sein und _onGround wurde damit bereits gesetzt. Allerdings bedeutet dies, dass ein Gegner oder der Spieler zu Levelbeginn in der Luft spawnen muss.
  Am Ende wird noch abgefragt, ob der Spieler in diesem Frame gewonnen hat, oder gestorben ist und die entsprechenden Flags gesetzt.

Die Unterklasse **StaticObject** von WorldObject stellt alle Objekte dar, welche keine Eigenbewegung im Spiel besitzen können. Die Methoden update() und collision() werden hier überschrieben und machen nichts. So können diese Methoden für alle Objekte aufgerufen wer-

den, ohne, dass sie unterschieden werden müssen.

Player ist eine Unterklasse von WorldObject und hat folgende zusätzliche Eigenschaften und Verhalten:

- **_life** speichert wie viele Leben der Spieler gerade hat
- **isDead()** fragt ab, ob die Lebenspunkte kleiner oder gleich 0 sind
- **removeLife()** zieht dem Spieler einen Lebenspunkt ab
- **addLife()** fügt dem Spieler einen Lebenspunkt hinzu
- **resetLife()** setzt die Lebenspunkte auf 3 zurück
- **_onGround** speichert, ob sich der Blob gerade in der Luft befindet, oder nicht. Dies wird benötigt, da ein Blob in der Luft nicht springen können darf.
- **_lookDir** speichert, ob die Blickrichtung des Blobs sich gerade geändert hat. Der Wert ist -1, wenn der Blob seine Blickrichtung nach links geändert hat und 1 falls nach rechts. Dies wird benötigt, damit die Textur entsprechend geupdated wird. Da das Laden dieses Bildes aber Rechenaufwand kostet, wird nur neu geladen, wenn die Blickrichtung geändert wurde und daher stellt die 0 den Zustand dar, dass diese sich nicht geändert hat.
- **_jump()** fügt dem Blob eine Geschwindigkeit nach oben zu, falls dieser sich auf dem Boden befindet (_onGround == true). Dies hat auf Grund der Implementierung der Physikengine automatisch eine Sprunganimation zur Folge.

Wenn ein Spieler unter eine gewisse Tiefe fällt, verliert der Spieler seine Leben. Die entsprechende Textur, sowie Maße, werden im Konstruktor festgelegt.

Enemy ist eine weitere Unterklasse von WorldObject, welche die Gegner des Spielers abbildet. Sie hat die selben Methoden und Variablen wie die Klasse Player, bis auf _life und dazugehörige Getter und Setter. Die entsprechende Textur, sowie Maße, werden im Konstruktor festgelegt. Die Methode update() wird hier überschrieben, um so das KI-Verhalten umzusetzen. Diese Methode benutzt ein in loadLevel() geladenes neuronales Netzwerk, um die KI zu steuern. Dabei werden hier in einem eingabeVektor die Richtung zum Spieler, die aktuelle Bewegungsrichtung, sowie die Information, ob sich ein Objekt in einer spezifische Richtung vor dem Gegner befindet gespeichert und als Eingabe verwertet. Das netzwerk gibt dann in form eines Vektors an, was die nächste Aktion der KI ist. Für weiteres siehe Abschnitt KI.

Virus ist eine Klasse, welche die Viren abbildet, welche die Gegner schießen. Hier wird im Konstruktor lediglich die entsprechende Textur und die Höhe und Breite festgelegt.

Door ist eine Klasse, welche Türen zu den einzelnen Abschnitten darstellt:

- **_isOpen** speichert, ob die Tür offen oder geschlossen ist
- **open()** verschiebt die Tür nach oben, sodass der Spieler unten durch gehen kann
- **close()** verschiebt die Tür nach unten, sodass der Spieler sie nicht passieren kann
- update() wird überschrieben, sodass sie kein Eigenverhalten hat

- collision() Hier wird eine zusätzliche Kollisionsabfrage durchgeführt, damit Objekte, welche breiter, als die Tür sind (Welche dann keine Kollision mit der Tür feststellen würden, da keine Ecke in der Tür liegen würde), zurückgesetzt werden.

Größe und Textur werden hier ebenfalls bereits im Konstruktor festgelegt.

Switch ist eine Klasse, welche Schalter darstellt:

- **_door** speichert eine Referenz auf die Tür, welche von diesem Schalter geöffnet wird
- **_isActive** speichert, ob der Schalter aktiviert ist
- **activate()** verändert die Textur und ruft open() an _door auf
- **deactivate()** verändert die Textur und ruft close() an **_**door auf
- update() Hier wird jedes Frame der Schalter deaktiviert (Wird bei Kollision mit Spieler oder Box wieder aktiviert), damit der Schalter automatisch deaktiviert wird, sobald keine Kollision mehr stattfindet.
- collision() aktiviert den Schalter, falls Kollision mit Spieler oder Box stattfindet

Größe und Textur werden hier ebenfalls bereits im Konstruktor festgelegt.

Box ist eine Klasse, welche vom Spieler verschiebbare Boxen darstellt.

- collision() Hier wird zusätzlich zur normalen Kollisionsbehandlung für bewegliche Objekte abgefragt, ob die Box mit einem Schalter kollidiert und aktiviert diesen gegebenenfalls

Ablauf des Spielstartes und laufenden Spieles: Der Controller instantiiert die beiden Klassen VirusGame und VirusView jeweils ein mal und vermittelt die Kommunikation zwischen diesen. So werden die graphische Darstellung und die logische Implementation voneinander getrennt und können leichter einzeln verändert werden. Der Controller kommuniziert hierbei mit VirusGame über die folgende Schnittstelle:

- **_level** gibt an, in welchem Level sich der Spieler gerade befindet. Die Levels werden mit Zahlen benannt.
- Mit **_running** kann der Controller in Erfahrung bringen, ob das Spiel gerade läuft, oder nicht. Hierbei wird immer true übermittelt, bis das Spiel pausiert wird (Menü wird angezeigt) oder der Spieler stirbt oder erreicht das Ziel und das Spiel stoppt.
- loadLevel() lädt ein beliebiges Level. Die Informationen über dessen Aufbau wird in einer JSON-Datei gespeichert und von der Methode ausgelesen.
  Diese Datei wird von einem Server bei einer entsprechenden HTML-Request der Methode übertragen.
- update() veranlasst VirusGame das nächste Frame zu berechnen.

Zunächst wird in der **main()** Methode eine Instanz des Controllers VirusController erzeugt. Diese erzeugt ein Objekt von Player und ruft dann die Methode **loadLevel()** in VirusGame auf, welche per HTML-Request die JSON-Datei für das erste Level holt. Diese enthält im wesentlichen nur die Konstruktorparamter aller Objekte, welche das Level bilden. Auf Basis dieser Werte werden dann entsprechend die einzelnen Konstruktoren aufgerufen und Objekte von den

zugehörigen Klassen erzeugt. Die Referenzen werden alle zusammen mit der Player Objekt Referenz in die Liste **_objects** gespeichert. Da jedes Objekt sein eigenes Verhalten und seinen Zustand wie Position selbst speichert, ist damit bereits das gesamte Leveldesign festgehalten.
Dann wird die Methode initialize() der View VirusView mit einer Referenz auf VirusGame aufgerufen (zur Funktion dieser siehe Abschnitt 3.3).
Damit ist die initialisierung des Spiels abgeschlossen. Nun ruft ein Timer in regelmäßigen Abständen von einigen ms die Methode **updateGame()** des Controllers auf, welche in folgender Reihenfolge:

- von VirusGame abfragt, ob das Spiel gerade läuft, oder gestoppt ist
- falls das Spiel läuft, die Methode update() von VirusView aufgerufen
- abgefragt ob das Spiel in dem nun neu berechneten Frame gewonnen wurde
- schließlich auf dem Bildschirm über update() von VirusView das neue Frame rendern lässt

Die Klasse **VirusGame** ist die logische Spielengine, welche alle Frames berechnet und alle Resourcen des Levels verwaltet. Sie hat folgende Komponenten zusätzlich zu den bereits oben erwähnten (sowie einige benötigte Getter und Setter, welche hier nicht weiter aufgelistet sind; siehe Quellcode):

- **_finishLine** speichert die Distanz, welche der Spieler vom Startpunkt aus überwinden muss, um zum Ziel zu gelangen und so zu gewinnen (wird evl durch einen Schalter ersetzt)
- **won** fragt ab, ob der Spieler die zu erreichende Distanz erreicht und somit dieses Level gewonnen hat
- **_playerMovement** speichert, in welche Richtung sich der Spieler gerade bewegen möchte (-1 = links; 0 = keine Bewegung; 1 = rechts).
- **update()** Diese Methode berechnet das nächste Frame.
  Zunächst wird dazu die Position des Spielers in die von _playerMovement bestimmte Richtung bewegt. Dies wird direkt über die move() Methode gemacht, anstatt die Geschwindigkeit zu setzen, da sonst die Geschwindigkeit bei Beendigung der Eingabe wieder verringert werden müsste. Dies ist aber ein Problem, wenn die Geschwindigkeit in diese Richtung auf Grund einer Kollision in Richtung einer Beschleunigung zurückgesetzt wurde, da dann eine nicht gewollte negative Geschwindigkeit addiert wird. Das direkte ansprechen der move() Methode erleichtert die Umsetzung dieses Verhaltens erheblich ohne negative Einbußen.

  Dann geht die Methode alle Objekte der Liste _objects durch und ruft für alle WorldObjects die update() Methode auf, um deren Position und Verhalten neu berechnen zu lassen. Zudem wird für jedes WorldObject wiederum die gesamte Liste durchlaufen und eine Kollision mit jedem Objekt abgefragt. Im falle einer Überschneidung mit einem Objekt, gibt collidesWith() einen Vektor zur nächstgelegenen Kante zurück. Damit kann dann das Objekt an diese Kante zurückgesetzt werden. Dieser Vektor wird mit dem dazu gehörigen Objekt an collision() des momentan berechneten Objekts übergeben.

- **movePlayer()** wird vom Controller aufgerufen, wenn eine Eingabe zum Bewegen stattgefunden hat und setzt _playerMovement auf den entsprechenden Wert. Dies passiert additiv, so dass ein gleichzeitiges laufen nach links und rechts (auf einem Desktopbrowser) zu einem Stillstand führt.

- **setPlayerMovement()** Hier wird _playerMovement auf den Winkel des smartphones um die Z-Achse gesetzt.
- **playerJump()** wird vom Controller aufgerufen, wenn eine Eingabe zum Springen eingeht und ruft die jump() Methode von _player auf
- **moveToWaypoint()** nimmt einen Vektor zum nächsten Wegpunkt und lässt den übergebenen Blob ein stück näher zu diesem Bewegen. Dabei werden auch Entscheidungen getroffen, ob der Blob über ein Hindernis, oder auf eine andere Ebene springen muss. Diese Wegpunkte sind in der Leveldatei festgelegt, b.z.w. ist ein Richtungsvektor in richtung Spieler (je nachdem ob sich Spieler und Gegner auf der selben Höhe befinden, oder nicht; siehe Abschnitt AI).
- **shoot** erzeugt ein Virusobjekt an der Position des übergebenen Gegners mit einer Geschwindigkeit in Richtung Spieler (dazu wird die Differenz der Positionen genommen, der Vektor zu einem Einheitsvektor gemacht und dann um einen Geschwindigkeitswert multipliziert). Die Referenz darauf wird dann _objects hinzugefügt.

3.2 View

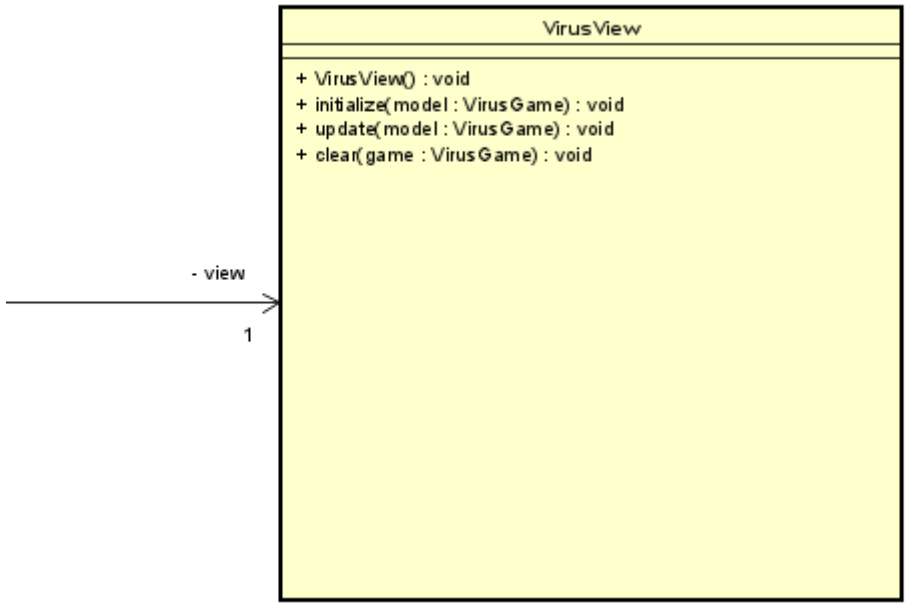

Abbildung 3.4: VirusView

Abbildung 3.4 zeigt das Klassendiagramm der VirusView.

Die **VirusView** ist für die grafische Darstellung des Spieles verantwortlich. Alle anzuzeigenden Elemente werdem einem div HTML Element 'game-view' hinzugefügt, welcher als Ausgabebildschirm für das eigentliche Level dient. Alle Elemente werden um einen gewissen Zoomfaktor multipliziert relativ zu diesem Element positioniert, so dass die Größenverhältnisse bei jeder Größe der 'game-view' erhalten bleiben (vorausgesetzt dessen Seitnverhältnisse vernändern sich nicht allzusehr). So ist das Spiel für jede Bildschirmgröße skalierbar und kann auf einem Mobilebrowser wie auf einem Desktopbrowser angezeigt werden. Zudem bietet dies eine gute Möglichkeit das Websitelayout beliebig anzupassen, ohne den Quellcode ändern zu müssen.

- **initialize()** bekommt eine Referenez auf das bereits initialisierte VirusGame Objekt und geht dessen Liste von Levelobjekten _objects durch. Für jedes Element wird ein entsprechendes HTML ImageElement erstellt, die richtigen Bilder und positionen ausgelesen und in dem HTML Element gesetzt. Anschließend wird dieses dem Dom-Tree als Kindknoten eines HTML Elements 'game-view' zugefügt (siehe Abschnitt HTML / CSS-Dokumentation) und eine Referenz darauf wird an das Objekt übergeben. Zudem werden Höhe und Breite der UI-Elemente der Html-Datei festgelegt, damit diese mit dem Bildschirm skaliert werden.
- **update()** bekommt eine Referenz auf das VirusGame Objekt und geht dessen Liste von Levelobjekten _objects durch. Für jedes Element wird die Referenz auf sein zugehöriges HTML ImageElement geholt. Dann wird die Position des Objektes ausgelesen und die entsprechenden Attribute des HTML Elements werden geupdated. Falls das Objekt ein Blob ist, wird zudem angefragt, ob dieser seine Blickrichtung geändert hat und in diesem Fall die richtige Textur geladen. Zudem wird die Lebensanzeige entsprechend geupdatet und diese wird zusammen mit dem Menü-Button für mobile-Browser im Z-Buffer nach ganz oben gelegt.
- **clear()** löscht alle Elemente aus dem Level aus dem DOM-tree, z.B. zum neu Laden eines Levels

3.3 Controller

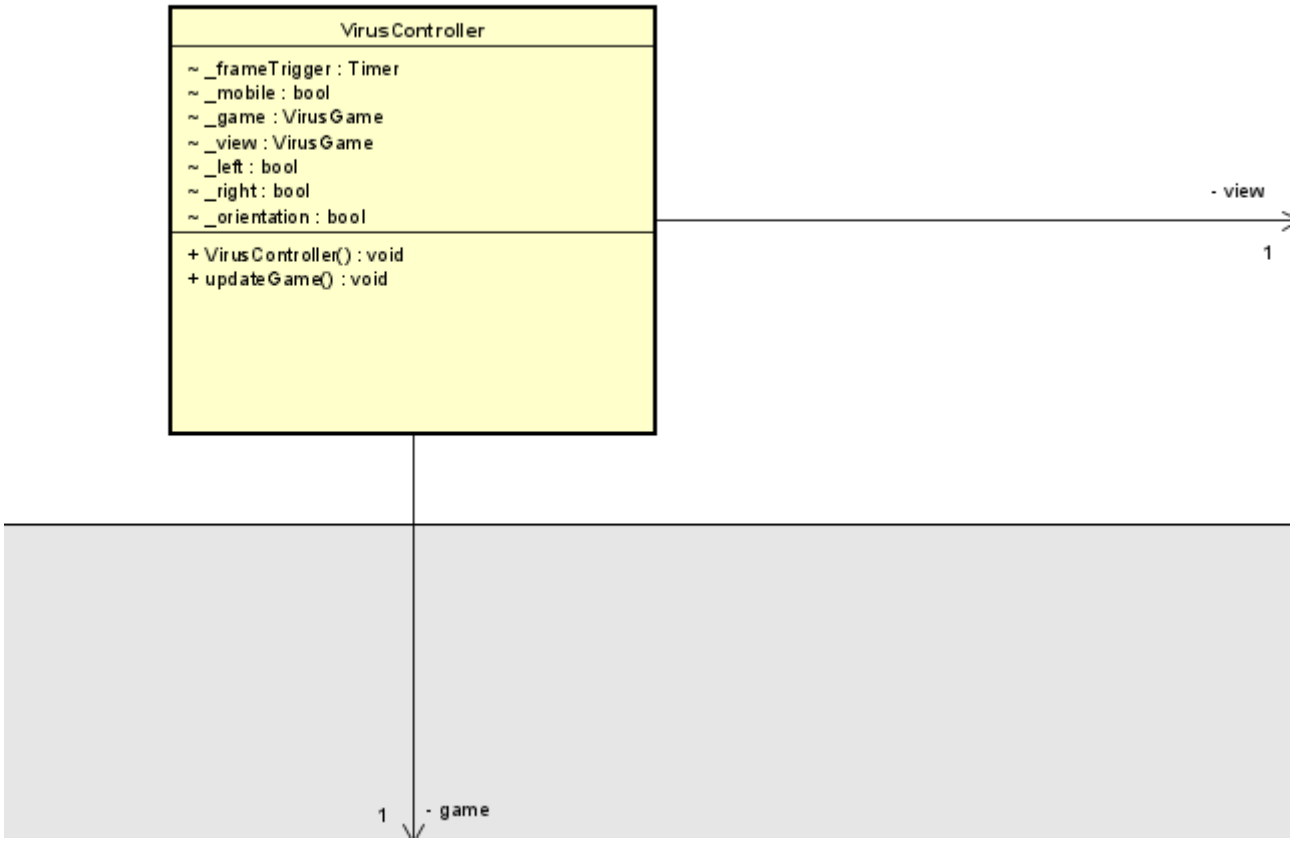

Abbildung 3.5: VirusController

Abbildung 3.5 zeigt das Klassendiagramm des Controllers **VirusController**. Der Controller ist für die Kommunikation zwischen In- und Output sowie der Spielengine zuständig. Außerdem verarbeitet er zeitgesteuerte Events, wie die Frames. Der Spieler hat die Möglichkeit seinen Blob über die Tastatur zu steuern, wenn er das Spiel im PC-Browser startet, oder über eine Gyrosteuerung und Touchdisplay, wenn der Spiel im Browser des Handys startet.

- **_frameTrigger** ein Timer, welcher das Berechnen und Darstellen jedes neuen Frames zur richtigen Zeit veranlasst
- Mit **updateGame()** ist der Listener für _frameTrigger (siehe Abschnitt Ablauf des Spielstartes und laufenden Spieles in Model)

Beim Erstellen des VirusController Objektes wird der Timer erstellt und auf ein Zeitintervall gesetzt. Dann werden die folgenden ActionListener erstellt:

- **window.onDeviceOrientation** behandelt den Input der Gyrosteuerung und ruft dann mit entsprechendem Parameter die Methode setPlayerMovement() von VirusGame auf.
- **querySelector('html').onTouchStart** wird aufgerufen, sobald auf den Touchscreen getippt wurde und ruft dann playerJump() von VirusGame auf.
- **querySelector('body').onKeyDown** wird aufgerufen, sobald eine Taste gedrückt wird und fragt dann ab, welche Taste gedrückt wurde. Daraufhin wird entsprechend die Methode movePlayer() mit den richtigen Parametern b.z.w. playerJump() von VirusGame aufgerufen.
- **querySelector('body').onKeyUp** wird aufgerufen, sobald eine Taste losgelassen wird und fragt dann ab, welche Taste dies war. Daraufhin wird entsprechend die Methode movePlayer() von VirusGame mit den richtigen Parametern aufgerufen.

Zudem werden verschiedene Listener für die Buttons erstellt, in denen die UI entsprechend verändert wird und gegebenenfalls ein neues Level geladen wird.

querySelector('body').onKeyDown und querySelector('body').onKeyUp müssen beide abgefragt werden, da die Laufrichtung für den Spieler bei loslassen wieder zurückgesetzt werden muss und diese additiv übergeben wird (siehe Abschnitt VirusGame). Dazu gibt es zwei bools in VirusController, welche speichern, ob der Listener bereits die Richtung gesetzt hat, damit dieser Wert bei Gedrückthalten der Taste nicht dauerhaft aufaddiert werden soll (andernfalls könnte dies beim Loslassen nicht mehr rückgängig gemacht werden).

3.4 Sequenzdiagramm laufendes Spiel

Der detaillierte Ablauf des Spieles wurde oben bereits beschrieben. Hier ist noch einmal das Sequenzdiagramm und eine grobe vereinfachte Zusammenfassung des Ablaufes:

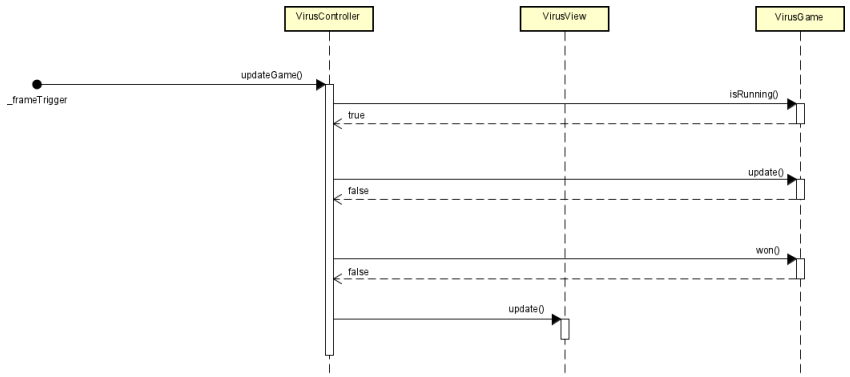

Abbildung 3.6: Sequenzdiagramm laufendes Spiel

3.5 KI Verhalten und Implementation

Auf Grund von Erfolgen in der Experimentierphase haben wir uns dazu entschieden, die KI-Gegner vom vorherigen Konzept zugunsten einer „echten“ KI abzuwandeln. Dabei wird die KI nun nicht mehr wie vorher geplant gescriptet, sondern von einem rekurrenten neuronalen Netzwerk gesteuert. Der grobe Aufbau, Entwicklungsprozess und Lernvorgang soll hier kurz beschrieben werden. Das Netzwerk ist wie bei einem normalen Feedforward-Netz in verschiedene Ebenen unterteilt (wobei das erste die Eingabeebene und das letzte die Ausgabeebene ist), welche miteinander verbunden sind. Allerdings gibt es festgelegte Ausgänge, welche ihre Ausgabe bei der nächsten Eingabe an dafür vorgesehene Eingänge in das Netzwerk zurückgibt. Dadurch wird eine Art Gedächtnis ermöglicht, welches zeitlich dynamisches Verhalten ermöglicht. Die bisher günstigste Konfiguration hat sich als ein Netz mit insgesamt 5 Ebenen (insclusive Ein- und Ausgabeebene) ergeben mit 12->15->10->8->2 Neronen, sowie 4 Ein- und Ausgangsneuronen für das Gedächtnis. Als Aktivierungsfunktion (benötigt, um nichtlineares Verhalten zu ermöglichen) benutzen wir $atan(x) * 2/\pi$, so ist die Ein- und Ausgabe auf das Intervall [-1, 1] beschränkt. Weiterhin wurden die Gewichte auf das Intervall [-4, 4] beschränkt, um eine zu hohe Instaibilität zu verhindern. Diese Parameter können sich aber im Verlauf des Experimentierens ändern und sind daher nur die bisher optimalsten.

Als eingabe erhlten die Gegner 8 Bit, welche angeben, ob sich oben, oben-rechts, recht, unten-rechts, unten, unten-links, links, oder oben-links vom Gegner Objekte befinden, der normierte Richtungsvektor zum Spieler, sowie die eigene Bewegungsrichtung. Die beiden Ausgänge bestimmen Geschwindigkeit in xund y-Richtung. Intern werden die Ebenen mathematisch als Matrizen repräsentiert, so kann die Ausgabe, von der Eingabe als Vektor dargestellt durch einfache Matrizenmultiplikation berechnet werden. Dafür benutzen wir die selbst erstellten Klassen Matrix und Network in dem Modul „network“. Es hat sich al günstiger für die KI erwiesen, diese fliegend zu machen, weswegen wir das Gegnerkonzept dahingehend abgeändert haben. Jedes Enemy-Objekt verfügt über sein eigenes Netzwerk, welches als Gehirn fungiert. Als Trainingsalgorithmus verwenden wir das Konzept der genetischen Algorithmen. Das heißt, es wird eine Population von Gegnern an einer zufälligen Position in dafür speziell vorgesehenen Trainingsleveln mit zufälligen Gewichten für die Netzwerke generiert. Dann wird nach einer gewissen Zeitgrenze ausgewertet, welcher Gegner den Spieler berührt hat, bzw diesem am nächten ist.

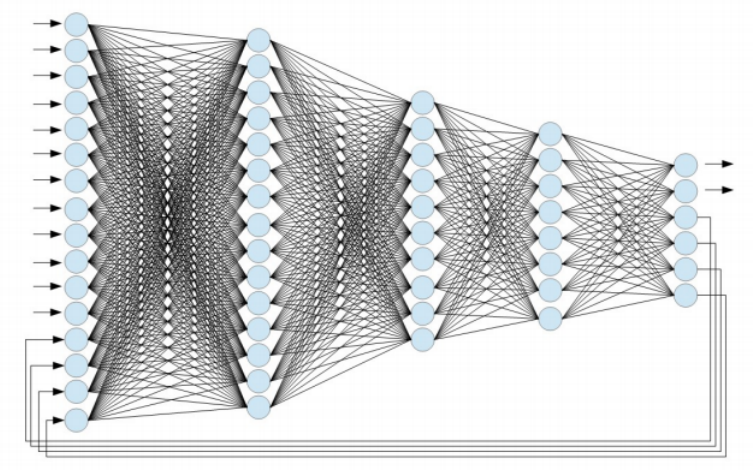

Abbildung 3.7: Neuronales Netzwerk

Das Netwerk des besten Gegners wird dann an alle anderen der Population „vererbt“ (kopiert) und alle werden mutiert (die Gewichte werden um einen kleinen Zufallswert variiert). Danach beginnt der Prozess mit diesen neuen Netzen von vorne (bei gleichzeitiger Trainingslevelrotation). Dadurch tritt nach dem Prinzip der Evolution ein Konvergenzprozess in Kraft, welcher die KI lernen lässt. Dieser Trainingsprozess wird von einem speziell dafür entwickelten Controller (Trainer) übernommen. Wurde ein befriedigendes Ergebnis erreicht, wird dieses Netz in einer JSON-Datei gespeichert und kann dann vom Spiel ausgelesen werden. Der Trainingsprozess ist unabhängig und getrennt vom eigentlichen Spiel implementiert und vom Spieler nicht sichtbar/ausführbar. Ist ein Netzwerk trainiert, so bleibt dieses im eigentlichen Spiel unverändert. Noch weist der Trainingsprozess zwar brauchbare Ergebnisse, bietet aber noch Verbesserungspotential. Das soll im weiteren Verlaufe hauptsächlich durch mehr und bessere Trainingslevel und dem Finden der richtigen Netwerkkonfiguration erreicht werden.

3.6 HTML / CSS Dokumentation

Die Html-Eelemente stellen alle Bilder Dinge da, die wir im Spiel implementiert haben.

Alle Elemente des Spiels werden der GameView im DOM-tree hinzugefügt und in der VirusView dann relativ zu dem Devision-Element positioniert. Dadurch lässt sich das Division-Element später beliebig im Browser positionieren, ohne dass die Quellcodeimplementation geändert werden muss. Das bietet eine Flexibilität bei der Gestaltung.

Zusätzlich gibt es noch ein Devision-Element, welches die Menü-Elemente enthält, sowie ein Html-Image-Element für den Help-screen. Diese können bei Bedarf vom Controller angezeigt oder versteckt werden.

Außerdem gibt es einfache CSS-Animationen, welche beispielsweise dafür sorgen, dass der Spieler blinkt, wenn er getroffen wird.

3.7 KI-Trainer, LevelEditor & Lokaler Server

KI Trainer

Wird die main.dart Datei mit dem gesetzten boolean für den Trainingsmodus gestartet, so lädt das Spiel die Dateien lvl-1.json, lvl-2.json, lvl-3.json als Trainingslevel und trainiert damit automatisch die KI. Das neue Netzwerk wird als network.json Datei gespeichert und beim starten des Spieles im normalen Modus geladen und für die Steuerung der KI genutzt (Achtung! Das alte wird überschrieben und muss gegebenenfalls manuell gesichert werden).

Level Editor

Wenn auch nicht zum eigentlichen Spiel gehörend, gibt es im Repository einen Leveleditor zum Erstellen neuer Levels. Dieser kommt in einer Kompilierten .jar Datei, sowie dem Java Quellcode. Es soll hier nur kurz die Bedienung erläutert werden.

- Mit den Pfeiltasten lässt sich das Spielfeld bewegen. - Zum Auswählen eines Objektes mit der Maus scrollen. - Zum Platzieren eines Objektes linke Maustaste, zum Löschen rechte Maustaste drücken - Zum Löschen der Finish-Line f drücken (es muss eine neue platziert werden, damit das Level spielbar ist!) - Wird ein Schalter platziert, so folgt danach automatisch der Modus zum Platzieren der dazugehörigen Tür - Mit L lässt sich ein Level öffnen - S speichert das aktuelle Level. Wurde vorher eines mit L geladen, so wird diese Datei überschrieben, andernfalls lässt sich der Speicherort auswählen

Lokaler Server

Zusätzlich wird im Repo ein lokaler Server zur Verfügung gestellt, welcher es ermöglicht, das Spiel ohne Internetverbindung (oder falls der Git-Server das Spiel nicht mehr hosted) zu spielen. Hierzu einfach die Datei server.dart aus dem entsprechenden Ordner ausführen, oder die bash Datei nutzen (unter Windows) und dann die main.dart des Spieles ausführen. Es muss vorher in der main Datei der boolean für ein lokales Spiel gesetzt werden!

Kapitel 4

Tabellen Anforderungen und Arbeitsaufteilung

4.1 Tabelle: Anforderungen

Anforderung	Erfüllt?	Beschreibung
AF-1: Spiel als Single-Page-App	Ja	Ein-Spieler-Game,HTML-single-Player, statische Website, Spielressourcen relativ zueinander adressiert.
AF-2: Einfaches, abgeleitetes Spielprinzip, Balance zwischen technischer Komplexität und Spielkonzept	Jein	Spiel ist schnell und intuitiv erfassbar, aber technische Komplexität mit einer KI vielleicht etwas zu hoch. Gemessen an den ausgewählten Komplexitätspunkten allerdings für unser Empfinden passend.
AF-3: DOM-Tree-basiert	Ja	MVC-Prinzip, DOM-Tree als View.
AF-4: Target Device: Your Choice	Ja	Mobile- und Webbrowser fähig. Mobile first. Android und IOS berücksichtigt.
AF-5: Mobile First Prinzip	Ja	Mobiles Spielen mit Gyro-Steuerung, Desktop-Spielen mit Tastatur-Steuerung.
AF-6: Intuitive Spielfreude	Ja	Spiel ist intuitiv spielbar. Spielfreude wird geweckt durch das Lösen von Rätseln und Gegner, die es schwerer machen (Meinungsfrage).
AF-7: Das Spiel muss ein Levelkonzept vorsehen	Jein	Level werden ohne Veränderung des Quellcodes nachgeladen, steigende Schwierigkeit mit zunehmendem Level durch Komplexität und Steigende Gegnerzahl.
AF-8: Ggf. erforderliche Speicherkonzepte sind Client-seitig zu realisieren	Ja	Gesammelte Daten bleiben auf Gerät, keine zentralen Server (mehr).
AF-9: Basic Libraries	Ja	Nur Libraries, die dem HTML5 Standart entsprechen.
AF-10: Keine Spielereien	Ja	Keine Sound- und Videoinhalte. CSS-Animationen.
AF-11: Dokumentation	Ja	Nach Anfänglicher Verwirrung zwecks Umfang nun stetig weitergeführte Dokumentation. Sowohl Quellcode, als auch Spielprinzip sind dokumentiert. Anforderungstabelle enthalten. Tabelle für Arbeitsaufteilung.

4.2 Tabelle: Arbeitsaufteilung

	Marvin Bergmann	**Ramona Kalkmann**
UML-Klassendiagramme		**x**
Implementierung Model	**x**	U*
Implementierung View	**x**	U*
Implementierung Controller	**x**	**x**
KI	**x**	
Design	**x**	**x**
Levelimplementierung	U*	**x**
Dokumentation	U	**x**

*Grund für unregelmäßigere, bzw weniger Commits und Pushes ist, dass meistens gemeinsam per Bildschirmübertragung gearbeitet wurde.