mng rämibühl

SoNNic

Neuroevolution in Sonic the Hedgehog

Maturitätsarbeit 2016/17

Autor: Kyrill Hux

Betreuer: Lukas Fässler

Abgabedatum: 09.01.2017

Abstract

SoNNic ist ein Plugin für den BizHawk-Emulator, das sich zum Ziel setzt, durch Neuroevolution mit dem *NEAT*-Algorithmus das erste Level des Spiels *„Sonic the Hedgehog“* zu meistern. Dazu werden dem Arbeitsspeicher des Spieles alle benötigten Informationen entnommen und daraus eine vereinfachte Ansicht generiert, die als Grundlage der *Inputs* von künstlichen neuronalen Netzwerken fungiert. Die *Outputs* der Netzwerke werden auf die Knöpfe des virtuellen Controllers weitergeleitet, um das Interagieren mit dem Spiel zu ermöglichen. Das Programm trainiert durch eine simulierte Evolution die Netzwerke, die sich dadurch autonom verbessern und die Mechaniken des Spiels lernen. Nach einem langen Testlauf konnte SoNNic das erste Level erfolgreich absolvieren und somit die Problemstellung lösen.

Inhaltsverzeichnis

1 Einleitung

Am 13. Juni 2015 veröffentlichte der User „SethBling“ auf der Videoplattform YouTube ein Video, in dem gezeigt wird, wie ein Programm mit dem Namen *„MarI/O“* mithilfe von Neuroevolution komplett selbständig lernt, das Videospiel *„Super Mario World“* zu spielen. Dies geschah auf Basis des Neuroevolutions-Algorithmus, der unter dem Akronym „NEAT“ [1] bekannt ist. Am gleichen Tag wurde ich auf dieses Video aufmerksam, welches mich dermassen faszinierte, dass ich beschloss, mich sofort mit neuronalen Netzen und schliesslich auch mit Neuroevolution zu beschäftigen.
In den folgenden zwei Wochen entwickelte ich eine C#-Bibliothek, die *NEAT* fast vollständig implementiert, und verwendete diese für einige Experimente, wie das Balancieren eines 2D-Würfels auf einer Kante oder das Balancieren eines Stabes auf einer beweglichen Plattform. (Siehe Abbildung 1 und 2)
Als es Zeit war, sich eine Maturitätsarbeit auszusuchen, kamen mir die Projekte des letzten Jahres wieder in den Sinn und ich beschloss, nach dem Vorbild *„MarI/O“* ein Programm zu entwickeln, das über Neuroevolution selbstständig lernt, einen Spieleklassiker zu spielen.
Als ich im Verlauf der Suche auf „*Sonic the Hedgehog*“ stiess, wusste ich, dass mein Ziel das erfolgreiche Abschliessen des ersten Levels sein würde. (Mehr dazu in Kapitel 3.1)

Abbildung 1: Ein *NEAT*-Experiment, bei dem 2D-Würfel das Ziel haben, möglichst genau auf einer Ecke zu balancieren. Als Information wird nur die X-Position und der Winkel der Würfel übergeben.

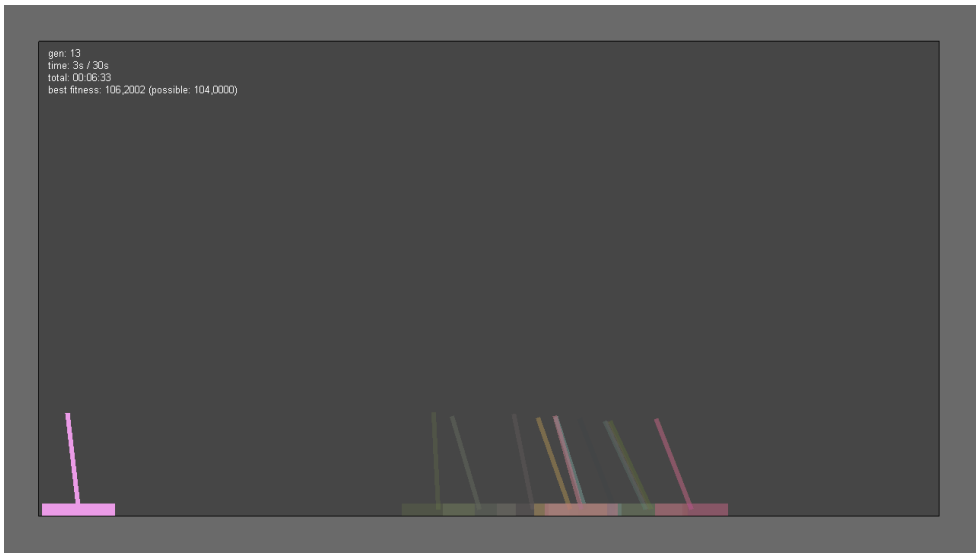

Abbildung 2: Ein weiteres *NEAT*-Experiment. Hier ist das Ziel, den Stab auf der Plattform zu balancieren. Der Computer bekommt als *Input* lediglich den Winkel des Stabs und die Position der Plattform.

2 Theoretische Grundlagen

2.1 Was sind künstliche neuronale Netze?

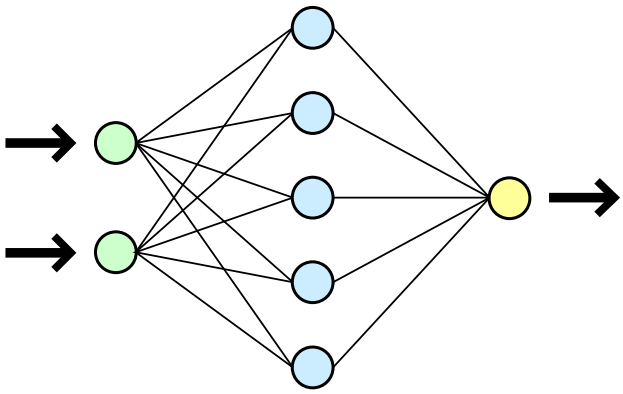

Abbildung 3: Die Struktur eines einfachen neuronalen Netzes. Die grünen Knoten stellen die *input nodes*, die blauen die *hidden nodes* und der gelbe Knoten die in diesem Fall einzige *output node* dar. [2]

In der Informatik sind künstliche neuronale Netze (fortan KNNs) mathematische Modelle, die Netzwerken aus Neuronen, wie man sie aus der Biologie kennt, nachempfunden sind. Auf Abbildung 3 ist die Struktur eines typischen KNNs abgebildet. In diesem Beispiel besteht das Netzwerk aus drei Schichten: einer Schicht Eingangsknoten (*input nodes*, grün), beliebig vielen Schichten „versteckter“ Knoten (*hidden nodes*, blau) und einer Schicht Ausgangsknoten (*output nodes*, gelb). Alle Schichten sind durch Linien (*connections*) verbunden.

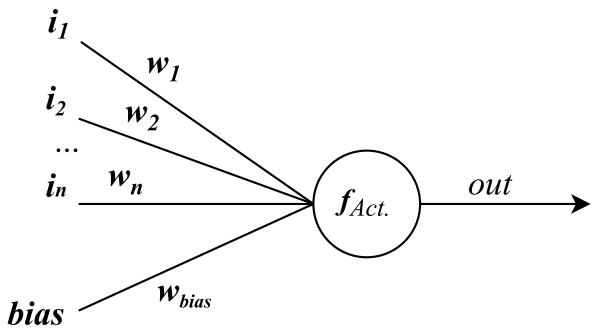

Abbildung 4: Der Aufbau eines künstlichen Neurons schematisch dargestellt.

Die Knoten eines KNNs (mit Ausnahme der *Inputs*) sind Nachbildungen von Neuronen, wie sie beispielsweise auch im menschlichen Nervensystem vorhanden sind. Ihre Funktionsweise ist auf Abbildung 4 dargestellt und vergleichsweise simpel: Alle empfangenen *Input*-Signale $i_1 ... i_n$ werden mit den Gewichtungen $w_1 ... w_n$ multipliziert und die Ergebnisse summiert. Wichtig ist hier, dass dazu noch der *bias*-Faktor ($bias \cdot w_{bias}$) addiert wird, der in den meisten Fällen die Gewichtung der Verbindung w_{bias} ergibt, da der *bias*-Wert gleich 1 ist. Auf die entstandene Summe wird eine Aktivierungsfunktion ($f_{(Act.)}$) angewendet. Diese ist meist eine *Step*-Funktion oder eine *Sigmoid*-Funktion. Das Endergebnis wird dann schliesslich über den *Output* weitergeleitet.
Ein KNN besteht also grob aus hintereinander geschalteten künstlichen Neuronen, die schliesslich alle in den *output-nodes* münden, die das Ergebnis liefern. Damit dieses Ergebnis jedoch sinnvoll für die jeweilige Problemstellung gebraucht werden kann, muss das Netzwerk „trainiert“ werden – es muss „lernen“. Hierfür gibt es viele Methoden, auf die einzugehen den Rahmen dieser Arbeit sprengen würde. Jedoch ist erwähnenswert, dass fast alle Lernmethoden voraussetzen, dass die Topologie, also die Struktur des Netzwerkes, von Anfang an fest vorgegeben ist. Entsprechend viel Arbeit ist schon vor Beginn des Lernprozesses nötig, nur um das Netzwerk zu planen und zu erstellen.

2.2 Was ist NEAT?

„NEAT“ ist eine Abkürzung für einen Algorithmus mit dem langen Namen *„NeuroEvolution of Augmenting Topologies“*, der in einer wissenschaftlichen Publikation des MIT Press Journal mit dem Titel *„Evolving Neural Networks through Augmenting Topologies“* [1] beschrieben wird. *NEAT* ist ein Lernalgorithmus für KNNs, der im Gegensatz zu den meisten konventionellen Lernmethoden nicht voraussetzt, dass der Mensch vor Beginn des Lernens eine Topologie konstruiert. Dieser Algorithmus emuliert eine Evolution, wie wir sie aus der Tierwelt kennen, wo die am besten an ihre Umwelt Angepassten überleben und Nachkommen zeugen, die zufällige Mutationen aufweisen können. Die Vorgehensweise ist dabei die folgende: Im ersten Schritt wird eine vorgegebene Anzahl Genome generiert, aus denen direkt KNNs erstellt werden können. In dieser ersten Generation enthalten diese Genome Informationen für KNNs, welche nur aus der vorher festgelegten Anzahl Inputs und Outputs bestehen, die alle miteinander durch Verbindungen mit zufällig generierten Gewichtungen verbunden sind. Der nächste Schritt ist, zu testen, welches Genom das Beste ist. Dazu werden aus allen Genomen die Netzwerke generiert. Diese müssen jetzt getestet und anschliessend danach bewertet werden, wie nahe sie an das gesetzte Ziel herankommen. Hierfür wird eine Funktion gebraucht, die es ermöglicht, zu messen, wie „gut“ ein Netzwerk ist, also wie nahe es an das gesetzte Ziel herankommt. Da es im Englischen bei der Evolution berühmterweise *„survival of the fittest“* heisst, wurde diese Funktion also entsprechend *Fitness*(-Funktion) benannt. Diese Funktion muss jeweils vom Menschen vor Beginn der Evolution erstellt werden. Wenn also beispielsweise die Aufgabe ist, als Ergebnis die Zahl 5 zu erreichen, könnte die *Fitness*-Funktion $f = 10 - (5 - x)^2$ sein. So wäre $f(5) = 10$, und je weiter x von 5 entfernt ist, desto niedriger der Wert der Funktion. So erhalten also alle generierten Netzwerke mit ihren Genomen einen *Fitness*-Wert. Als letzter Schritt wird die nächste Generation generiert. Dies geschieht zum einen Teil durch zufällige Mutationen von vorhandenen Genomen, zum anderen Teil durch Kreuzung zweier Genome. Welche Genome für beide Fortpflanzungsarten genutzt werden, wird durch gewichteten Zufall bestimmt, in dem die Genome mit besserer *Fitness* eine grössere Wahrscheinlichkeit haben, ausgewählt zu werden. Die neu entstandene Generation wird anschliessend in „Spezies“ aufgeteilt. Mehr Details zu diesem Vorgang können im *NEAT*-Paper nachgelesen werden.

3 Implementierung

3.1 Finden eines geeigneten Spiels

Der erste Schritt nach der Ideenfindung war das Finden eines geeigneten Emulators[1], der über ein Plugin-System verfügen sollte. Nach längerer Suche stiess ich auf den Emulator *„BizHawk“*, der über ein Lua-Skriptsystem und seit kurzer Zeit auch über ein C#-Plugin-System verfügt. Da ich in C# am meisten Erfahrung habe, kam dieses Feature wie gerufen.

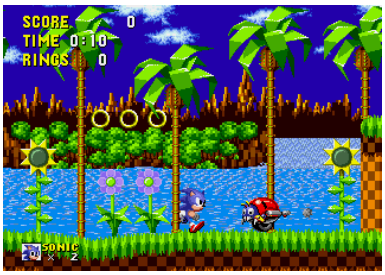

Abbildung 5: Der 1991 auf der Spielekonsole „Sega Genesis“ veröffentlichte Titel *„Sonic the Hedgehog“*.

Abbildung 6: *„Excitebike“*, ein populäres NES-Spiel, das im Jahre 1984 veröffentlicht wurde.

Schliesslich musste noch ein Spiel gefunden werden, das sich für ein solches Neuroevolutions-Experiment eignete. Als erstes kam natürlich die „Super Mario“-Reihe von Nintendo in den Sinn. Doch da „SethBling“ mit *„MarI/O“* diese Aufgabe bereits erfüllt hatte, kam dies für mich nicht mehr in Frage – auch wenn mein Ansatz ein etwas anderer gewesen wäre.
Auf meiner Suche stiess ich auf verschiedene theoretisch geeignete Spiele. Eines davon war der im Jahre 1984 veröffentlichte NES-Titel *„Excitebike“* von Nintendo (Abbildung 6). Auch wenn die lineare Spielweise sich vortrefflich für ein solches Experiment eignen würde, fehlte dem Spiel leider die Art Dokumentation, die für ein Vorhaben wie meines nötig wäre. Der Arbeits- und Zeitaufwand, der mit dem Reverse-Engineering des Arbeitsspeichers des Spieles verbunden wäre, was für ein solches Vorhaben unbedingt nötig ist, ist einfach zu hoch und würde den Rahmen einer Maturitätsarbeit sprengen. Schliesslich fand ich den Klassiker *„Sonic The Hedgehog“* (Abbildung 5), der

[1] Ein Emulator ist ein Programm, das ein anderes System (oder Teile davon) simuliert. Diese Programme werden vor allem zum Spielen älterer Videospiele verwendet, deren Konsolen nicht mehr erhältlich sind und deshalb nur so spielbar sind.

1991 für die damals sehr populäre Spielekonsole „Sega Mega Drive“[2] veröffentlicht wurde. Dieses Spiel eignete sich durch seine relativ simple, lineare Spielweise sehr gut für einen solchen Algorithmus: Die Hauptfigur, der blaue Igel „Sonic“, muss von links nach rechts ans Ende des Levels rennen und dabei verschiedenen Gegnern und Hindernissen ausweichen, ganz ähnlich wie Mario bei der fast universell bekannten Konkurrenz von Nintendo. Doch im Gegensatz zu Mario, der sich nur auf aus Blöcken zusammengebauten Levels bewegt, ist Sonics Welt deutlich feiner aufgelöst, mit kleinsten Senkungen und Hebungen.

Abbildung 7: Das erste Level von *Sonic the Hedgehog*. Es kann durchlaufen werden, ohne die Richtung zu wechseln und eignet sich dadurch vortrefflich für das Vorhaben. [4]

Besonders gut für das Vorhaben geeignet war das erste Level, denn dieses erfordert kein Zurücklaufen in die dem Ziel entgegengesetzte Richtung, was nicht nur die Erstellung einer *Fitness*-Funktion erschweren würde, sondern auch den Computer beim Lernprozess behindern würde. Da nun ein Spiel vorhanden war, das ausgezeichnet dokumentiert war und eine passende Spielweise bot, konnte die eigentliche Entwicklung beginnen.

3.2 Entwicklung der Kartenansicht

Der erste Schritt war es, die Spielwelt zu verstehen. Das heisst in diesem Fall, die Spielwelt einlesen und verarbeiten können. Hier kommt der Arbeitsspeicher der emulierten Konsole ins Spiel: Die ganze Topographie des Levels, sowie alle Gegner, Objekte und mehr müssen für schnellen Zugriff während des Spielens in den Arbeitsspeicher geladen werden. Durch Einlesen und Interpretieren des Inhalts können also sehr wertvolle Informationen gewonnen werden. *„Sonic The Hedgehog“* gilt heutzutage als Klassiker und hat eine sehr aktive Community, die stetig an verschiedensten Modifikationen und Zusatzinhalten für das Spiel arbeitet. Um diese Arbeiten zu erleichtern, hat die Community eine ausführliche Dokumentation erarbeitet, die nicht nur den Inhalt und die Funktionsweise des Arbeitsspeichers, sondern auch

[2] „Sega Mega Drive“ und „Sega Genesis“ sind zwei Namen für eine Konsole, die aus markenrechtlichen Gründen für einige Länder unter verschiedenen Namen vermarktet wurde.

vieler anderer wichtiger Spielmechaniken übersichtlich auf Websites zusammengefasst erläutert.[3] Mithilfe dieser Dokumentation und einer sehr hilfsbereiten IRC-Community[4] gelang es mir nach vielen Stunden Arbeit, das komplette Level sowie alle Gegner und sonstig relevanten Objekte aus dem Arbeitsspeicher einzulesen und korrekt zu interpretieren.

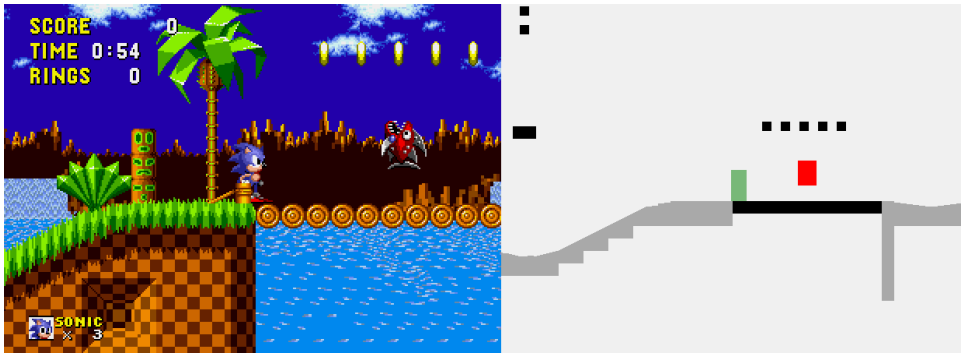

Abbildung 8: Ein Vergleich der Karten- und Spielansicht. Links ist die volle Spielansicht abgebildet, in der noch alle Details und Verzierungen, die für SoNNic irrelevant sind, angezeigt werden. Der rechte Teil zeigt die Kartenansicht, die nur relevante Informationen anzeigt: feste Objekte in schwarz, Gegner in rot, und feste Landschaftsteile in grau. Sonics *Hitbox* wird als (hier grünes) Rechteck in der Mitte der Kartenansicht dargestellt.

Aus diesen Informationen konnte schliesslich eine vereinfachte Version des Spielgeschehens konstruiert werden – eine Art Kartenansicht oder „Map“ (siehe Abbildung 8), wo alle festen Teile des Levels, sowie Objekte mit *Hitboxen*[5] dargestellt werden. Alle unnötigen Details und Verzierungen werden ausgeblendet, da sie für die Navigation im Level irrelevant sind. Die dargestellten Objekte werden in zwei Kategorien unterteilt: Die harmlosen Objekte wie Ringe oder Steine, die Sonic keinen Schaden zufügen, sind schwarz eingefärbt, die Gegner und sonstig schädliche Objekte (wie zum Beispiel Stacheln) sind in roter Farbe eingezeichnet. Übrig bleiben nur noch die Terrain-Stücke, die grau sind (also technisch gesehen auch Teil der schwarzen Objekte sind), sowie Sonic selbst, der als durchsichtiges Rechteck in der Mitte der *Map* erscheint. Die Kartenansicht wird sich später als eines der Kernstücke von

[3] Die umfangreichste Dokumentation, die auch für dieses Projekt verwendet wurde, befindet sich unter http://info.sonicretro.org/Sonic_the_Hedgehog_(16-bit).

[4] Die IRC-Community lässt sich unter *„badnik.net:6667“* im Channel *„#SSRG“* finden.

[5] Hitboxen sind rechteckige Flächen, die spielintern für die Kollisionsabfrage verwendet werden.

SoNNic erweisen, da diese die Daten für die KNNs liefern wird.

3.3 Bestimmung der Inputs // Die Sensoren

Da nun die Daten vorhanden waren, war es Zeit, die KNN-Struktur zu planen. Als erstes mussten die *Inputs* festgelegt werden. Nach einigen Überlegungen entschied ich mich für eine unkonventionelle Methode: Überhaupt keine *Inputs* festzulegen. Bei SoNNic werden die *Inputs* durch Mutationen generiert und als Punkte auf der *Map* relativ zu Sonics Position platziert. Um diese Punkte herum wird eine rechteckige Fläche errichtet, die den Einflussbereich dieser Punkte darstellt. Schneidet diese Fläche nun ein Objekt auf der *Map*, färbt sich diese in eine von zwei Farben: grün, wenn das geschnittene Objekt nicht schädlich ist und rot, wenn das Gegenteil der Fall ist. Die Punkte mit ihren Flächen werden zur Einfachheit „Sensoren“ genannt.

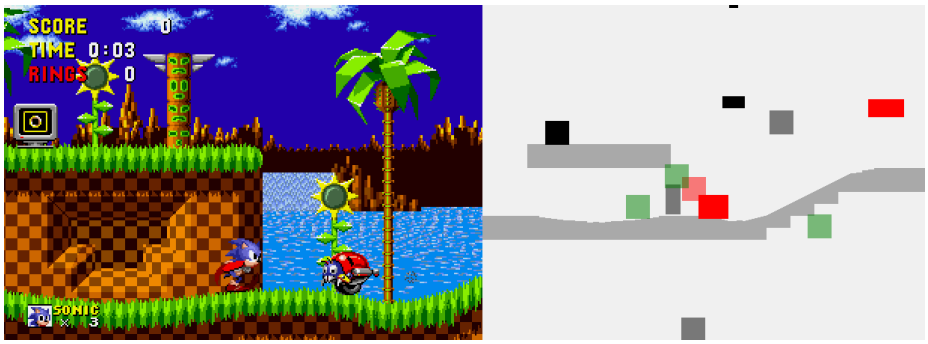

Abbildung 9: Ein Beispiel für Sensoren. Die 6 durchsichtigen, rot, grün und grau eingefärbten Quadrate in der *Map*-Ansicht sind aktuell vorhandene Sensoren. Rot eingefärbte überschneiden sich mit schädlichen, grün eingefärbte mit harmlosen Objekten.

Auf Abbildung 9 ist eine *Map* dargestellt, auf der bereits 6 Sensoren vorhanden sind. Wie weiter oben beschrieben, sind die Sensoren, die etwas Festes (aber nicht Schädliches) berühren, grün eingefärbt, während der Sensor, der sich mit der *Hitbox* des Gegners überschneidet, rot eingefärbt ist. Wie vorher erwähnt, gibt es für jeden Sensor einen zugehörigen *Input* im KNN. Je nach Status des Sensors erhält der *Input* einen anderen Wert: Ist der Sensor grau (keine Berührungen), wird dieser Wert auf 0 gesetzt. Sobald der Sensor auf grün wechselt, wird der Wert auf 1 festgelegt. Bei Rot erhält der *Input* den Wert -1. Somit ist der Kontakt mit Gegnern für das KNN leicht von solchem mit beispielsweise dem Boden unterscheidbar.

3.4 Bestimmung der Outputs

Die Outputs des KNNs waren um einiges schneller erarbeitet: Für jeden sinnvollen Knopf auf dem virtuellen Spielecontroller ist ein *Output* vorhanden: Für das D-Pad, mit dem die Richtungen vorgegeben werden (Abbildung 10 links) , sind 4 *Outputs* vorhanden (oben, unten, links, rechts). Da alle drei Knöpfe (Abbildung 10 rechts) auf dem Controller die gleiche Funktion erfüllen (springen), wären drei Knopf-*Outputs* für das Trainieren des Netzwerkes kontraproduktiv. Deshalb wurde nur ein *Output* für Knopf 1 implementiert. Insgesamt ergibt das also eine Summe von 5 *Outputs*.

Abbildung 10: Darstellung eines Sega Genesis Controllers. Mit dem Steuerkreuz auf der linken Seite wird der Charakter in *Sonic the Hedgehog* bewegt, mit einem beliebigen der drei Knöpfe auf der rechten Seite wird gesprungen. [3]

3.5 Aufstellen der Fitness-Funktion

Da nun alles Nötige vorhanden war, um einem KNN die Kontrolle über einzelne Individuen zu überlassen, musste eine Möglichkeit gefunden werden, *NEAT* mitzuteilen, wie gut ein Individuum sich schlägt, um damit die Evolution zu koordinieren. Dazu wird, wie in Kapitel 2.2 erwähnt, eine *Fitness*-Funktion gebraucht. Im Falle von SoNNic war das Aufstellen einer *Fitness*-Funktion eine relativ triviale Aufgabe. Weil das erste Level kein Rückwärtsgehen erfordert und somit durch stetiges Laufen nach rechts mit geschickten Ausweichmanövern komplett abgeschlossen werden kann, kann die *Fitness* einfach mit der X-Koordinate der Spielfigur Sonic beschrieben werden, die in diesem Falle gleichzeitig den Fortschritt im Level darstellt. Diese Koordinate kann aus dem Arbeitsspeicher des Spiels ausgelesen werden. Die finale *Fitness* ist die *Fitness* im jeweils letzten berechneten Bild

eines Laufs (=Durchgangs eines Individuums).

3.6 Der IdleWatcher

Damit überhaupt mehrere Läufe hintereinander ausgeführt werden können, muss ein Lauf autonom (zum Teil auch vorzeitig) beendet werden können. Diese Aufgabe ist nicht so trivial, wie sie klingen mag, da verschiedene Ereignisse den Lauf nicht mehr fortsetzungswert machen können. Das offensichtlichste Ereignis ist der Tod des Charakters, aber auch bestimmte Verhaltensweisen, wie zum Beispiel das Stehenbleiben über längere Zeitstrecken oder das Laufen in die falsche Richtung (nach links), müssen durch das Beenden des Laufs bestraft werden.
Um die Läufe zu überwachen, wurde der sogenannte „IdleWatcher“ entworfen: Ein Kontrollmechanismus, der bei jedem Lauf im Hintergrund mitläuft und kontrolliert, ob der Lauf planmässig weiterläuft. Der *IdleWatcher* zählt vereinfacht gesagt die *Frames*[6], wo Sonic sich gar nicht oder rückwärts bewegt. Überschreitet diese Zahl einen festgelegten Grenzwert, wird der Lauf beendet. Wenn sich Sonic wieder nach vorne bewegt, wird die Zahl eingefroren und sobald Sonic seine im Level am weitesten liegende frühere Position wieder erreicht hat, wird sie wieder auf 0 zurückgesetzt. An dieser Stelle kommt es auch zum Vorteil, dass Sonic sich während der Todes-Animation spielintern nicht bewegt – somit zählt diese auch als Inaktivität und der Lauf wird beendet.
Mit dem *IdleWatcher* hat SoNNic eine sehr zuverlässige Möglichkeit, ineffektive Läufe frühzeitig zu beenden und eine maximale Zeitnutzung zu garantieren.

3.7 Der Quellcode

Der Quellcode des fertigen Plugins kann auf GitHub unter der folgenden Adresse eingesehen werden:
'http://github.com/realChesta/SoNNic'.

[6] Bewegtbilder bestehen aus mehreren Bildern oder „Frames“ pro Sekunde.

4 Evaluation

4.1 Vorgehen

4.1.1 Ablauf des Evolutionsprozesses

Mit allen nötigen Funktionen, die jetzt eingebaut sind, kann SoNNic komplett autonom den Lernprozess durchführen. Dies läuft wie folgt ab: Als erstes wird die *Map* generiert. Dies muss nur einmal geschehen, da das Level sich im Verlauf der Evolution nicht verändert. Nachdem dies abgeschlossen ist, kann die eigentliche Evolution beginnen: Das erste Individuum der ersten Generation führt seinen Lauf durch. Dazu wird ein Speicherpunkt geladen, der das Spiel direkt an den Anfang des ersten Levels versetzt. Hier übernimmt sofort das KNN des Individuums unter Aufsicht des *IdleWatchers* das Steuer. Sobald der *IdleWatcher* den Lauf für beendet erklärt, wird das Spiel pausiert, der aktuelle *Fitness*-Wert gespeichert und der vorher erwähnte Speicherpunkt wieder geladen. Nun kann das nächste Individuum sein Bestes probieren, das Level zu meistern.
Wenn nun eine ganze Generation auf diese Art und Weise geprüft wurde, kann der *NEAT*-Algorithmus alle gesammelten *Fitness*-Werte vergleichen und daraus die nächste Generation erstellen. Diese durchläuft dann das gleiche Prozedere und kann hoffentlich bessere Resultate liefern. So läuft SoNNic vollständig autonom, bis die gewünschte maximale *Fitness* von einem Individuum erreicht wurde.

4.1.2 Monitoring der Evolution

Auch wenn der Evolutionsprozess komplett autonom abläuft, musste der ganze Prozess natürlich trotzdem überwacht werden. Zu diesem Zweck hat SoNNic ein kleines Fenster, in dem die wichtigsten Informationen zum aktuellen Prozess dargestellt werden (Siehe Abbildung 11).

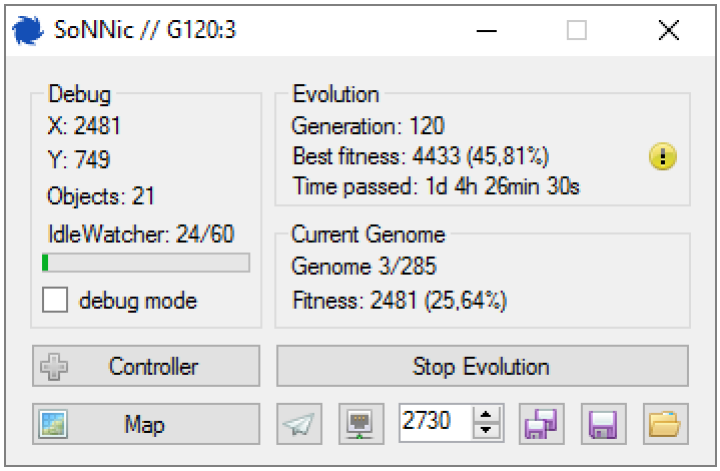

Abbildung 11: Das Kontroll-Fenster. Hier werden alle wichtigen Informationen zur aktuell laufenden Evolution übersichtlich dargestellt.

Dieses Fenster ist in vier Bereiche aufgeteilt: Links oben befindet sich der Debug-Bereich: Hier werden wichtige interne Informationen wie Sonics Position oder der aktuelle Zustand des *IdleWatchers* dargestellt. Der zweite Bereich befindet sich oben rechts und stellt die Eckdaten der aktuellen Evolution dar: Die aktuelle Generations-Nummer, die bisher beste *Fitness*, und die insgesamt seit dem Start der Evolution verstrichene Zeit. Hier befindet sich auch ein kleiner Knopf, der, wenn gedrückt, dafür sorgt, dass bei einer Steigerung der maximalen bisher erreichten *Fitness* das Fenster in der Windows-Taskbar zu blinken beginnt. So kann Aufmerksamkeit auf einen kürzlichen Fortschritt gelenkt werden. Der letzte Informations-Abschnitt befindet sich unter dem Evolutions-Abschnitt und zeigt den aktuellen Status des Individuums, das zurzeit geprüft wird. Eine Kurzübersicht der aktuellen Status kann auch aus dem Titel des Fensters entnommen werden, der wie alle anderen Informationen auch in Echtzeit aktualisiert wird. So kann auch bei minimiertem Fenster der Status über die Windows-Taskbar geprüft werden. (Siehe Abbildung 12)

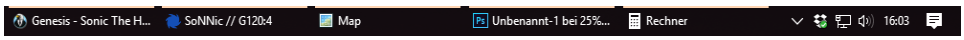

Abbildung 12: Eine Windows-Taskbar mit laufender SoNNic-Instanz. Durch den Fenstertitel kann der Evolutionsfortschritt mit einem Blick überprüft werden.

Übrig bleibt nur noch der letzte Bereich, der fast ausschliesslich aus Knöpfen besteht. Hier kann zum Beispiel eine Darstellung der aktuellen Outputs, also der aktuell gedrückten Knöpfe über den Button „Controller“ (Abbildung 13) oder die *Map*-Ansicht über den Button „Map“ aufgerufen werden.

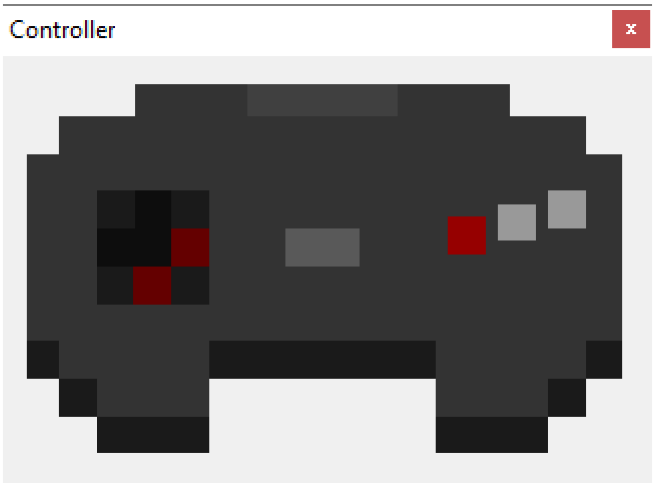

Abbildung 13: Das Controller-Fenster. Hier können die aktuell im Spiel gedrückten Knöpfe beobachtet werden.

Auf der rechten Hälfte dieses Bereiches befindet sich ausserdem auch der „Start/Stop“-Knopf, der die Evolution entsprechend starten und stoppen kann. Unter diesem befinden sich in der rechten unteren Ecke des Fensters drei Knöpfe: der Erste ist für das automatische Speichern zuständig, mit dem Zweiten kann die aktuelle Generation gespeichert werden und mit dem Letzten eine gespeicherte Generation wieder geladen werden. Ein automatisches Speichersystem war insbesondere in früheren Versuchen wichtig, da es manchmal zu Abstürzen kommen konnte und damit der ganze Fortschritt verloren gegangen wäre. Deshalb wurde diese Funktion auch relativ früh im Entwicklungsprozess eingebunden. Mithilfe der anderen beiden Knöpfe kann jeweils die Telegram- und die RemotePanel-Funktionalität ein- und ausgeschaltet werden. Mehr zu diesen beiden Funktionen im folgenden Absatz.

4.1.3 Remote Monitoring

Da die Evolution zum vollständigen Meistern des Levels eine lange Zeitspanne in Anspruch nimmt, wurde das Überwachen derselben über längere Zeit zunehmend schwierig bis unmöglich, da ich nicht immer zur Stelle sein kann. Um dem entgegenzuwirken, habe ich zwei neue Funktionen eingefügt. Die Erste der beiden ist das sogenannte „Remote-Panel“ (Abbildung 14), eine kleine Windows-Anwendung, die sich über TCP zur aktuell laufenden SoNNic-Instanz verbinden und die aktuellen Informationen abrufen kann. Damit ist es also möglich, von einem entfernten Windows-PC den Fortschritt zu prüfen, den SoNNic bisher gemacht hat.

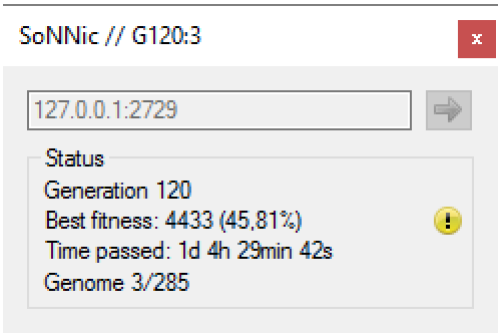

Abbildung 14: Das Remote-Panel. Mithilfe dieser Anwendung können die wichtigsten Eckdaten der laufenden Evolution aus der Ferne überwacht werden.

Die zweite Möglichkeit zum Überwachen der Evolution aus der Ferne erwies sich als weitaus ausgefeilter und effizienter: Ein Bot in der Messenger-App „Telegram“ (Abbildung 15). Telegram ist eine Smartphone-, aber auch eine PC-Anwendung, die für praktisch alle Plattformen verfügbar ist. Sie ist vergleichbar mit der App „WhatsApp“. Doch Telegram stellt weitaus mehr Funktionen zur Verfügung – sowohl für Benutzer, als auch für Entwickler. Eine dieser Funktionen ist eine API[7], die sogenannte „Bots“ ermöglicht. Bots sind in Telegram automatisierte Programme, die wie normale Benutzer Nachrichten empfangen oder versenden können. Der sogenannte „SoNNicBot“ kann auf Anfrage hin den Status der aktuellen SoNNic-Instanz mitteilen (ganz ähnlich wie das RemotePanel), hat aber gegenüber dem RemotePanel zwei entscheidende Vorteile: Telegram ist universell verfügbar und

[7] Application Programming Interface – eine Schnittstelle, die es Entwicklern ermöglicht, bestimmte Funktionen einfacher in ihre Projekte einzubinden.

muss nicht auf einem Windows-PC ausgeführt werden. Somit kann der Status auch zum Beispiel vom Smartphone aus geprüft werden. Dadurch, dass Telegram eine Messenger-App ist, können aber auch Status-Updates ohne Nachfrage geschickt werden. So wird zum Beispiel immer eine Nachricht gesendet, wenn ein neuer *Fitness*-Rekord erreicht wurde. Auf diese Art und Weise können Informationen zum aktuellen Stand der Evolution ohne hindernde Zusatzprogramme einfach und schnell übermittelt werden.

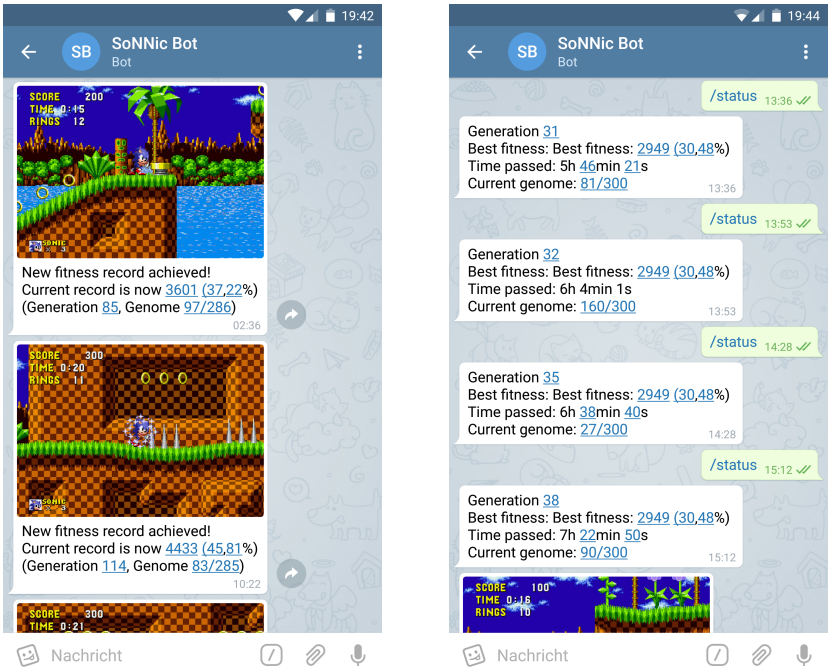

Abbildung 15: Screenshots vom Telegram-Bot. Über diese Funktion kann man sofort über Fortschritte im Evolutionsprozess benachrichtigt werden, oder den aktuellen Status manuell abfragen.

4.2 Resultate

Da nun alle Voraussetzungen für einen vollständigen Langzeit-Lernprozess (also im Bestfall bis zum Meistern des Levels) erfüllt waren, wurde dieser auch gestartet. Der Lernprozess lief auf vierfacher Geschwindigkeit auf meinem Desktop-Computer mit einer „Population“ von 300 Individuen pro Generation.
Erfreulicherweise konnte SoNNic nach fast 289 Stunden (etwas über 12 Tagen) und 1029 Generationen das erste Level von „*Sonic the Hedgehog*“ erfolgreich abschliessen.
Der Verlauf der durchschnittlichen *Fitness*-Werte (Abbildung 16) nimmt eine fast logarithmische Form an: nach anfänglichen sprunghaften Erfolgen verlangsamt sich der Fortschritt über mehrere hundert Generationen, bis er praktisch nicht mehr vorhanden ist. Erst nach etwa 550 Generationen können wieder Erfolge verzeichnet werden und die Kurve steigt treppenartig bis auf eine durchschnittliche *Fitness* von ungefähr 5000. Der Verlauf der Zeit, die jede Generation zum Abschliessen brauchte, gleicht dem der durchschnittlichen *Fitness*-Werte – der Graph verläuft beinahe gleich und weist die gleichen Schwankungen an den gleichen Stellen auf. Hier fallen zusätzlich jedoch einige Ausreisser auf, die wesentlich mehr Zeit gebraucht haben, als die vor- und nachfolgenden Generationen.
Nicht diesem Muster entsprechend verhält sich der Verlauf der besten *Fitness*-Werte. Dieser verläuft in einer Treppenform. Anfangs kann noch vergleichsweise schnell ein Fortschritt erzielt werden, jedoch ziehen sich die „Treppenstufen“ sehr schnell über mehrere hundert Generationen. Besonders fällt hier die Treppenstufe auf, die sich etwa von Generation 400 bis 750 erstreckt, in der praktisch kein Fortschritt gemacht wird.

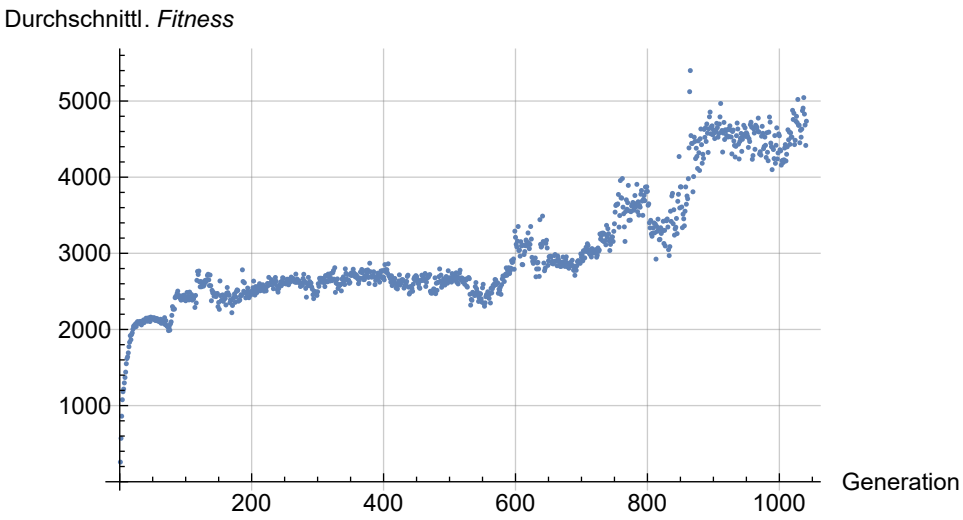

Abbildung 16: Die durchschnittliche *Fitness* aller Generationen im Vergleich. Nach schnellem anfänglichem Erfolg bleibt der Fortschritt über mehrere hundert Generationen praktisch aus. Kurz vor Generation 600 kann ein treppenhafter Erfolg bis zum Erfüllen des Ziels verzeichnet werden.

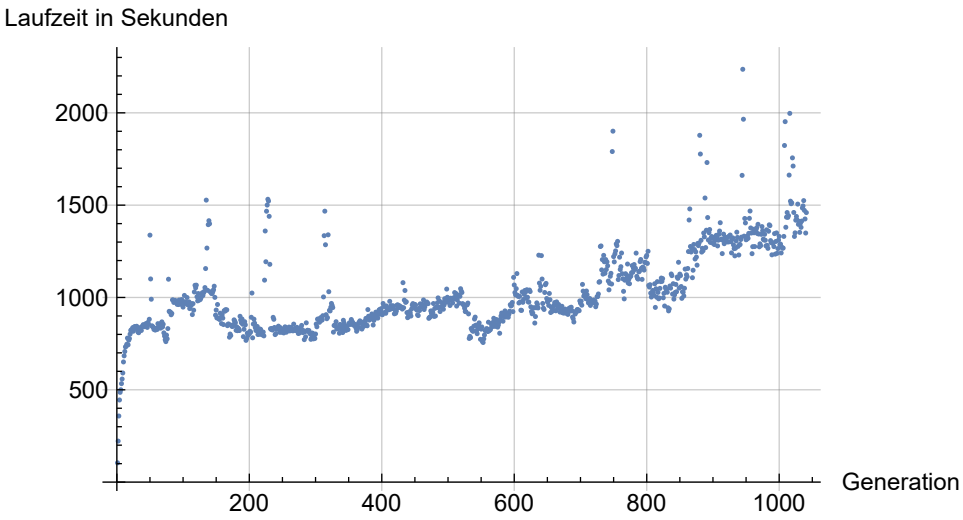

Abbildung 17: Die Laufzeit aller Generationen im Vergleich. Die Laufzeit scheint der durchschnittlichen *Fitness* (Abbildung 16) zu folgen. Alle markanten Stellen können auch hier wiedergefunden werden. Auffallend einige Ausreisser, die jedoch Ausnahmen bleiben.

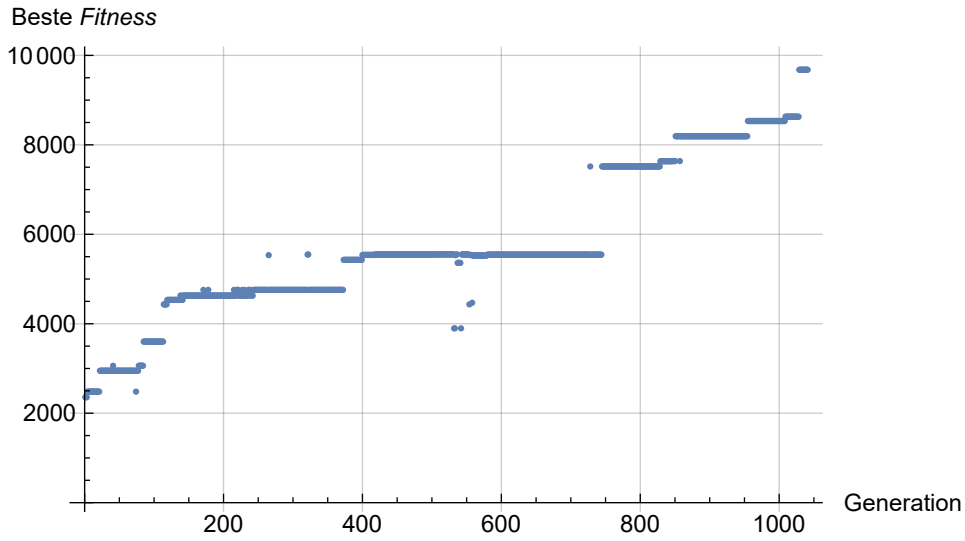

Abbildung 18: Die besten *Fitness*-Werte jeder Generation. SoNNic scheint mit einigen Stellen im Level zu kämpfen zu haben, da oft über mehrere hundert Generationen hinweg kein Fortschritt gemacht wird.

4.3 Interpretation der Resultate

Beobachtet man die Laufzeit der Generationen im Vergleich zu den durchschnittlichen *Fitness*-Werten, kann man eine offensichtliche Korrelation feststellen. Da die *Fitness* dem Fortschritt im Level entspricht, kann dieser Zusammenhang damit erklärt werden, dass es länger dauert, weiter im Level voranzukommen. Entsprechend steigt die Laufzeit einer Generation proportional zu seiner durchschnittlichen *Fitness*, bzw. zur *Fitness*-Summe.
Da die *Fitness*-Werte einfach die X-Koordinate des jeweiligen Spielcharakters darstellen, kann von den *Fitness*-Werten auf die Position im Level zurückgeschlossen werden. Um dies zu bewerkstelligen, reicht es die X- und Y-Achsen in Abbildung 18 vertauschen. Vom resultierenden Diagramm lässt sich dann ablesen, welche Stellen im Level die grössten Schwierigkeiten verursachten.

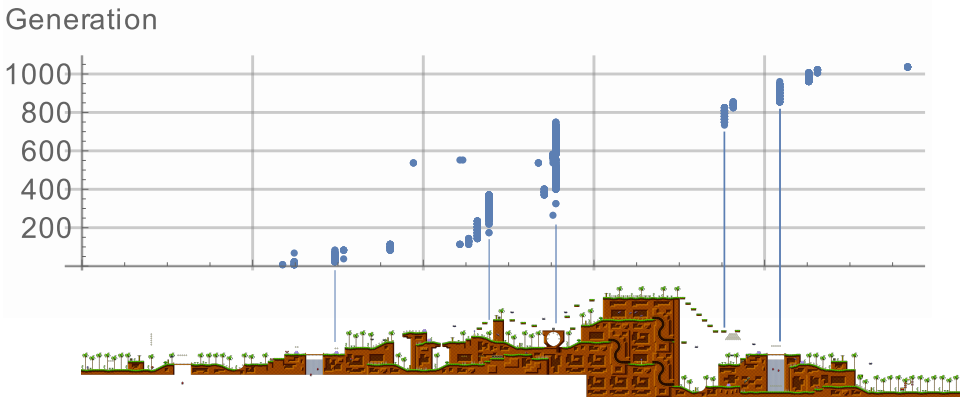

Abbildung 19: Mit den Generationen auf der Y-Achse kann auf die Problemstellen im Level geschlossen werden. Die zwei grössten scheinen der Looping und die Stelle kurz davor zu sein, da sich dort mehrere Stachel-Blöcke in kurzen Abständen zueinander befinden.

Beim Betrachten von Abbildung 19 fallen sofort zwei Stellen auf, die scheinbar die grössten Probleme verursacht haben. Die Erste der beiden (Abbildung 20) befindet sich im Level an der Strecke vor und bis an die X-Koordinate 4575. Schaut man sich diese an, lassen sich die Probleme leicht erklären: Hier ist eine hohe Konzentration an Stachel-Blöcken, die bei Berührung der oberen Seite den sofortigen Tod bewirken. Da drei sehr nah aneinander liegende Stachel-Blöcke und eine Stufe direkt aufeinanderfolgen, musste SoNNic also einen genauen Rhythmus finden, der zum Erfolg verhelfen würde.

Abbildung 20: Der erste der beiden problematischsten Stellen: Hier sind drei Stachelblöcke und eine nachfolgende Treppe in sehr kleinen Abständen zueinander positioniert, was sehr genaues Timing erfordert.

Die zweite Strecke ist die am meisten Schwierigkeiten verursachende Stelle des ganzen Levels. Der Grund hierfür ist ein Looping. Hier musste SoNNic alle bekannten Prinzipien, wie zum Beispiel das Springen über Hindernisse, über Bord werfen und einen komplett neuen Ansatz verfolgen. Denn dieser Abschnitt ist nur zu meistern, wenn mit einer bestimmten Anfangsgeschwindigkeit durch den Looping gerannt wird. Dennoch konnte SoNNic auch diese Herausforderung überwinden.

Abbildung 21: Der Looping-Abschnitt. Dieser Abschnitt erwies sich als besonders problematisch, da zum Überwinden alles Gelernte über Bord geworfen werden muss.

Alle restlichen Stellen sind einfache Stufen oder einzelne Hindernisse wie Steine, die im Weg stehen und übersprungen werden müssen. Diese Hindernisse konnten innert weniger Generationen überwunden werden.

5 Schlussfolgerungen

In Anbetracht der Ergebnisse lässt sich sagen, dass das Ziel des Experiments erfolgreich erreicht wurde – SoNNic konnte das erste Level erfolgreich abschliessen. Allerdings dauerte das trotz vierfacher Beschleunigung ganze 12 Tage, was dreierlei Ursachen haben könnte:

- Bestimmte Parameter, die hier für den *NEAT*-Algorithmus verwendet wurden, waren nicht optimal eingestellt und behinderten damit den Lernprozess.
- Der *NEAT*-Algorithmus ist nicht optimal für diese Aufgabe geeignet.
- „*Sonic the Hedgehog*“ ist durch die beschriebenen problematischen Stellen für einen solchen Lernprozess nicht gut geeignet.

Im Nachhinein kann ich mir mehrere Verbesserungen vorstellen, die den Lernprozess um einiges beschleunigen würden: Zum einen könnten die *NEAT*-Parameter für SoNNic optimiert werden. Dies würde SoNNic eine deutlich effizientere Evolution ermöglichen und den Fortschritt verschnellern. Allerdings fehlte mir im Verlauf des Experiments hierzu die Zeit, da dies zum grössten Teil durch blosses Versuchen und Beobachten bewerkstelligt wird.
Zum anderen konnte ich beobachten, dass durch die zufällige Platzierung der Sensoren einige Überschneidungen entstanden sind, die wahrscheinlich nicht zur Effizienz des Lernens beitragen. Ich schreibe 'wahrscheinlich', da es auch sein könnte, dass durch überlappende Sensoren eine Art „and-Gate“ entsteht, das einen ganz eigenen, positiven Einfluss auf das Lernen haben könnte.
Als dritte Verbesserung könnten zusätzlich zu den Sensoren permanente *Inputs* in die KNNs eingebaut werden. Daten, die diese *Inputs* enthalten, könnten zum Beispiel die X-Position des Spielcharakters oder spielinterne Timer-Werte sein. Letzteres sind Werte, die bestimmte Level-Elemente wie dauerhaft springende Gegner oder in späteren Leveln schwingende Plattformen koordinieren. Diese Werte lesen zu können würde SoNNic sicherlich dabei helfen, sich in solchen Abschnitten besser zurechtzufinden.

6 Ausblick

Blickt man über das ursprünglich gesetzte Ziel heraus, stellt sich natürlich die Frage, ob SoNNic die gelernten Fähigkeiten auch in späteren Leveln einsetzen kann, um diese erfolgreich abzuschliessen. Um diese Frage zu beantworten, habe ich SoNNic mit der Generation 1030, die das erste Level erfolgreich abgeschlossen hat, den Lauf durch Level 2 und 3 versuchen lassen.
Die Ergebnisse sind hier interessant zu sehen: SoNNic konnte knapp einen Viertel des zweiten Levels allein mit dem Training durch das erste Level meistern. Jedoch scheiterte der Versuch bei einem Level-Element, das SoNNic komplett unbekannt war: eine schwingende Plattform (Abbildung 22). Diese Plattform bewegt sich hin- und her und der Spieler muss versuchen, auf diese zu heraufzuspringen, um so auf eine höher gelegene Ebene zu gelangen. Durch das fehlende Training für solche Plattformen konnte SoNNic diese Passage nicht überwinden. Hätte SoNNic mit Level 2 trainiert, würde diese Stelle bestimmt eine ähnliche Herausforderung darstellen, wie der Looping in Level 1. Da diese Plattformen intern auch durch Timer gesteuert werden, könnten die in den Schlussfolgerungen erwähnten Zusatz-*Inputs* bestimmt eine grosse Hilfe beim Meistern dieses Abschnittes darstellen.

Abbildung 22: Die schwingende Plattform in Level 2. Diese Stelle erfordert ein genaues Timing und ist somit mit dem Training aus dem ersten Level nicht zu meistern.

Beim dritten Level sah es ähnlich aus: nach etwa einem Fünftel des Levels stiess SoNNic auf ein unüberwindbares Hindernis: eine hohe Mauer, die nur mithilfe einer weiter vorne im Level gelegenen Sprungfeder überwunden wer-

den kann (Abbildung 23). Auch hier gilt: Durch Training in diesem Level könnte dieses Hindernis überwunden werden. Allerdings gibt es keine ähnlichen Stellen im ersten Level und diese Situation war völlig unbekannt.

Abbildung 23: Die Mauer mit der Sprungfeder in Level 3. Der Spieler muss die Sprungfeder auf der erhöhten Plattform benutzen, um über die Mauer zu springen. Diese Stelle erwies sich als ebenfalls nicht mit dem Training aus dem ersten Level überwindbar.

Nach diesen zwei Versuchen lässt sich feststellen, dass SoNNic bekannte Situationen zwar durchlaufen kann, aber bei grösseren Unstimmigkeiten mit dem Gelernten Schwierigkeiten aufweist. Ich bin überzeugt, dass diese Probleme mithilfe der in den Schlussfolgerungen beschriebenen Verbesserungen zumindest zum grössten Teil behoben werden könnten und SoNNic dann mit dem gleichen KNN mehr als nur ein Level durchlaufen könnte. Hätte SoNNic nur mit dem zweiten oder dritten Level trainiert, würde ein Lauf durch das erste Level vermutlich ähnlich aussehen, wie der jetzige Lauf durch Level 2 und 3.

7 Literatur

[1] Kenneth. O. Stanley, Risto Miikulainen:
Evolving Neural Networks through Augmenting Topologies,
MIT Press Journal: Evolutionary Computation,
Volume 10, Issue 2, Sommer 2002.

[2] Dake, Mysid:
Vectorized by Mysid in CorelDraw on an image by Dake.,
CC BY 1.0,
https://commons.wikimedia.org/w/index.php?curid=1412126

[3] BLUEamnesiac:
Sega Genesis 3-Button Controller,
http://blueamnesiac.deviantart.com/art/Sega-Genesis-3-Button-Controller-353867157

[4] Sonic Galaxy:
Green Hill Zone Act 1
http://www.sonicgalaxy.net/img/maps/gen/sonic/ghz-1.png

Bestätigung der Eigenständigkeit

Der Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbstständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

Kilchberg, den 5. Januar 2017