České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství

Adaptace autonomního agenta na chování uživatele

Adaptation of autonomous agent to user's behavior

BAKALÁŘSKÁ PRÁCE

Vypracoval: Martin Jochec
Vedoucí práce: Ing. Josef Nový, Ph.D.
Rok: 2023

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství Akademický rok 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	**Martin Jochec**
Studijní program:	**Aplikace přírodních věd**
Obor:	**Aplikace softwarového inženýrství**
Název práce česky:	**Adaptace autonomního agenta na chování uživatele**
Název práce anglicky:	**Adaptation of autonomous agent to user's behavior**
Jazyk práce:	**Čeština**

Pokyny pro vypracování:

1. Nastudujte problematiku adaptace agenta na chování uživatele
2. Navrhněte aplikaci pro vizualizaci chování autonomního agenta
3. Navrhněte metody pro adaptaci agenta na chování uživatele ve vhodné podobě pro integraci s navrženou vizualizační aplikací
4. Navrženou aplikaci a metody implementujte a otestujte

Doporučená literatura:

[1] ESTERLE, Lukas, 2022. *Deep learning in multiagent systems.* Deep Learning for Robot Perception and Cognition. Elsevier, 2022, 435-460. ISBN 9780323857871. doi:10.1016/B978-0-32-385787-1.00022-1

[2] RED'KO, Vladimir G. a Danil V. PROKHOROV, 2010. *Learning and Evolution of Autonomous Adaptive Agents.* Advances in Machine Learning I. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, 491-500. Studies in Computational Intelligence. ISBN 978-3-642-05176-0. doi:10.1007/978-3-642-05177-7_25

[3] WANG, Di a Ah-Hwee TAN, 2015. *Creating Autonomous Adaptive Agents in a Real-Time First-Person Shooter Computer Game.* IEEE Transactions on Computational Intelligence and AI in Games. 7(2), 123-138. ISSN 1943-068X. doi:10.1109/TCIAIG.2014.2336702

Jméno a pracoviště vedoucího práce:

Ing. Josef Nový, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze

Jméno a pracoviště konzultanta:

–

.....
vedoucí práce

Datum zadání bakalářské práce: 15. 10. 2022

Termín odevzdání bakalářské práce: 2. 8. 2023

Doba platnosti zadání je dva roky od data zadání.

.....
garant oboru

.....
vedoucí katedry

.....
děkan

V Praze dne 15. 10. 2022

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Děčíně dne ....

Martin Jochec

Poděkování

Děkuji Ing. Josefovi Novému, Ph.D. za vedení mé bakalářské práce a pomoc při její tvorbě a také Mgr. Daně Majerové, Ph.D. za rady týkající se správnosti psaní bakalářské práce a citací.

Martin Jochec

Název práce:
Adaptace autonomního agenta na chování uživatele

Autor: Martin Jochec

Studijní program: Aplikace přírodních věd
Obor: Aplikace softwarového inženýrství
Druh práce: Bakalářská práce

Vedoucí práce: Ing. Josef Nový, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze
Konzultant: –

Abstrakt: Tato bakalářská práce se zabývá adaptací autonomního agenta na chování uživatele. Jejím hlavním cílem je vytvořit jednoduchého autonomního agenta, který se dokáže přizpůsobit uživateli. V mnoha případech jsou autonomní agenti konstruováni tak, že jsou stabilně naprogramováni, což však vytváří problém s neměnností. Tito agenti jsou pak velmi předvídatelní, protože se nemohou přizpůsobit konkrétní situaci. Tato bakalářská práce se zabývá také základní teorií o autonomních agentech a příklady algoritmů, které lze použít ke konstrukci více přizpůsobitelného autonomního agenta. Obsahuje také některá existující řešení pro tuto problematiku i návrh a implementaci vlastního řešení.
Klíčová slova: autonomní agent, neuronová síť, umělá inteligence, rozhodovací strom, adaptace

Title:
Adaptation of autonomous agent to user's behavior

Author: Martin Jochec

Abstract: This bachelor thesis deals with the adaptation of an autonomous agent to user behavior. Its main goal is to build a simple autonomous agent that can adapt to the user. In many cases, autonomous agents are constructed in such a way that they are stably programmed, but this creates an immutability problem. These agents are then very predictable because they cannot adapt to a specific situation. This bachelor thesis also covers the basic theory about autonomous agents and examples of algorithms that can be used to construct a more adaptable autonomous agent. It also includes some existing solutions to the problem as well as the design and implementation of a my own solution.
Key words: autonomous agent, neural network, artificial intelligence, decision tree, adaptation

Obsah

Úvod

Tato bakalářská práce se zabývá adaptací autonomního agenta na chování uživatele. Jejím hlavním cílem je vytvořit jednoduchého autonomního agenta, který se dokáže přizpůsobit uživateli. V mnoha případech jsou autonomní agenti konstruováni tak, že jsou stabilně naprogramováni, což však vytváří problém s neměnností. Tito agenti jsou pak velmi předvídatelní, protože se nemohou přizpůsobit konkrétní situaci.

První kapitola se vztahuje k teorii autonomích agentů. Tato kapitola má dvě podkapitoly. První kapitola obsahuje základní informace o autonomním agentech, jak agenty zkonstruovat a také jejich využití. Ve druhé podkapitole jsou popsány rozhodovací stromy, neuronové sítě a genetické algoritmy, což jsou jen některé z mnoha typů algoritmů, které se dají využít pro sestrojení lepšího autonomního agenta.

Kapitola druhá také obsahuje dvě podkapitoly. První podkapitola obsahuje existující řešení na tuto problematiku v herním průmyslu, kde první ze zmíňených řešení je autonomní agent postaven tak, aby za vás dokázal hrát danou hru. Druhé řešení je systém ve hře jménem Shadow of War, kde tento systém běží na pozadí, když je potřeba, a sbírá data z bitev, aby upravil vzhled, vztah nebo atributy nepřítele a znova ho vyslal zaútočit na hlavní postavu. Ve druhé podkapitole je zahrnut návrh umělé inteligence a vizualizace a také použitý software a knihovny potřebný k vytvoření vizualizace a umělé inteligence.

Třetí a poslední kapitola popisuje tvorbu vizualizace a autonomního agenta. První část obsahuje informace a příslušný kód v obrázcích nebo v textu, jak postavit základní bloky vizualizace. Druhá část obsahuje implementaci základní umělé inteligence a také dvou systémů, které budou upravovat autonomního agenta podle akcí uživatele.

Závěr popisuje úspěšnost bakalářské práce, samotnou hru a také případné budoucí rozšíření, pomocí kterého by se dal autonomní agent vylepšit.

Kapitola 1

Teorie autonomních agentů

1.1 Základní informace

1.1.1 Co to je autonomní agent

Autonomní agent[12] je entita, která má schopnost provádět činnosti a rozhodovat se samostatně a nezávisle na vnějších vlivech. Tento pojem je často spojován s oblastí umělé inteligence (AI)[5], kde jsou autonomní agenti navrženi tak, aby mohli vykonávat úkoly a reagovat na prostředí bez potřeby přímého lidského ovládání. Pojem „agent“ ve skutečnosti není tak přesně definován a samotný autonomní agent může mít různou složitost - od primitivních programů založených na pravidlech až po závratně složité systémy. Tato technologie, původně koncipovaná v oblasti výzkumu umělé inteligence, má schopnost zahrnout komplexní techniky umělé inteligence. Oblasti, kde agenti skutečně zazáří, jsou ty, kde se neustále analyzují datové toky, pečlivě sledují rozsáhlé databáze a kde se vyžadují povrchní reakce na události. Tyto aplikace bývají spojeny s technologií uživatelského rozhraní a neustále se vyvíjejícím internetem.

Někdo může agenty považovat za pouhé nástroje, které pomáhají ulehčit lidem od jejich pracných, rutinních úkolů. Horliví obhájci autonomních agentů však tvrdí, že tito agenti mohou být navrženi tak, aby napodobovali a dokonce překonávali kognitivní funkce svých lidských protějšků.

Z hlediska definice se za agenty považují entity, které:

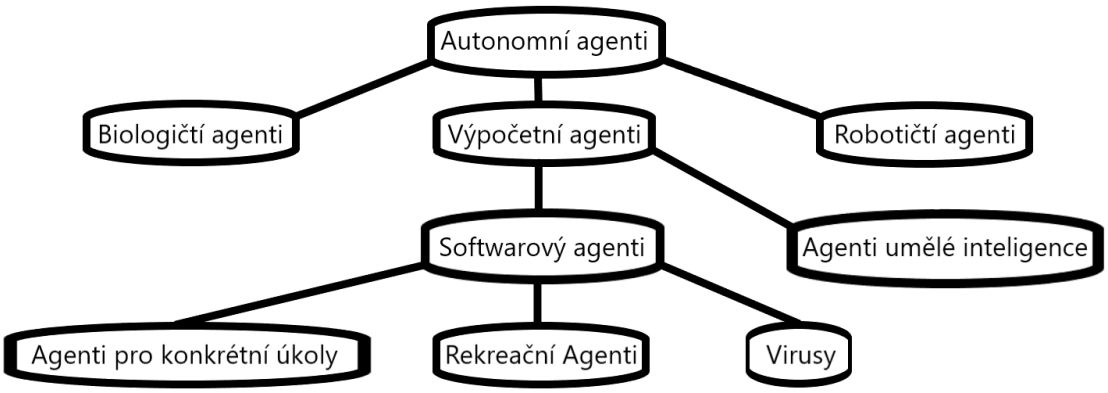
1. Vnímají a reagují na události pomocí předem definovaných reakcí
2. Zkoumají datové toky v určitém prostředí
3. Vykazují chování, které představuje záměry vlastníka nebo uživatele agenta
4. Proaktivně jednají z vlastní iniciativy
5. Fungují nepřetržitě

Agenti mají rovněž schopnost realizovat následující funkce:

1. Učení - Agent se adaptuje na základě statistické analýzy dat a pozorování parametrů procesu (ačkoli teoreticky by mohly být začleněny i složitější formy učení)

2. Komunikace - Může probíhat buď mezi jinými agenty, nebo mezi lidmi a zahrnuje např. výměnu dat nebo parametrů

Stojí za zmínku, že ačkoli agenti mají určitý stupeň autonomie, jejich činnost a logika je omezena buď zájmy jejich vlastníků nebo jejich zaměřením. Rozsah autonomie se může značně lišit, od agentů zaměřených na konkrétní úkoly s úzkým rozsahem použití až po agenty podobné virům, které se rychle šíří a fungují nezávisle na svých vlastnících. Rozdělení autonomních agentů lze vidět na obrázku 1.1



Obrázek 1.1: Typy autonomních agentů

Agenti se od obecných softwarových programů liší v řadě ohledů, např. v nepřetržitém provozu, reaktivním chováním a částečné autonomii. Stojí za zmínku, že zpětnovazební řídicí systémy mají některé z těchto vlastností společné, ale obvykle se omezují na sledování spojitých veličin a řízení žádaných hodnot nebo jednoduchých vynucovacích funkcí (např. termostatů). Často se mluví o autonomních robotech a agentech jako o jednom a tom samém, ale je důležité si uvědomit, že zatímco roboti se skládají z hardwaru specificky určeného k vykonávání určených funkcí, agenti se skládají výhradně ze softwarových funkcí pro analýzu dat a komunikaci. Chování robotů se optimalizuje vyladěním senzorů, efektorů a řídicích programů a musí se optimalizovat také s ohledem na energetické a fyzikální parametry, které u agentů obvykle nejsou důležité.

1.1.2 Konstrukce autonomních agentů

Agenti jsou navrženi tak, aby byli efektivní pro jednotlivé úkoly, s cílem maximalizovat výkon a spolehlivost a zároveň minimalizovat pravděpodobnost chyby, náklady a další relevantní vlastnosti. Inteligentní chování autonomního agenta se týká míry

do jaké optimálně reaguje na sledované události, přičemž kritéria kvality zahrnují správnost, rychlost a spolehlivost reakcí. Cílem agenta je optimalizovat užitek pro svého vlastníka. Chování agenta je posuzováno jako inteligentní bez ohledu na to, zda jsou základní výpočetní procesy implementovány jako pevný řídicí algoritmus, nebo zda zahrnují plánování, učení a uvažování.

Agenty lze vyvíjet jako reaktivní systémy na základě důkladné analýzy a pochopení úkolů a kontextu, v němž agent působí. Inteligentní chování není funkcí výpočtů agenta v reálném čase, ale optimálního výběru omezeného počtu pravidel nebo chování, které celkově generují úspěšné chování. Principy umělé inteligence, včetně plánování a učení, lze využít k tomu, aby agent úspěšně reagoval v mnoha různých stavech, které není třeba ve fázi návrhu předvídat.

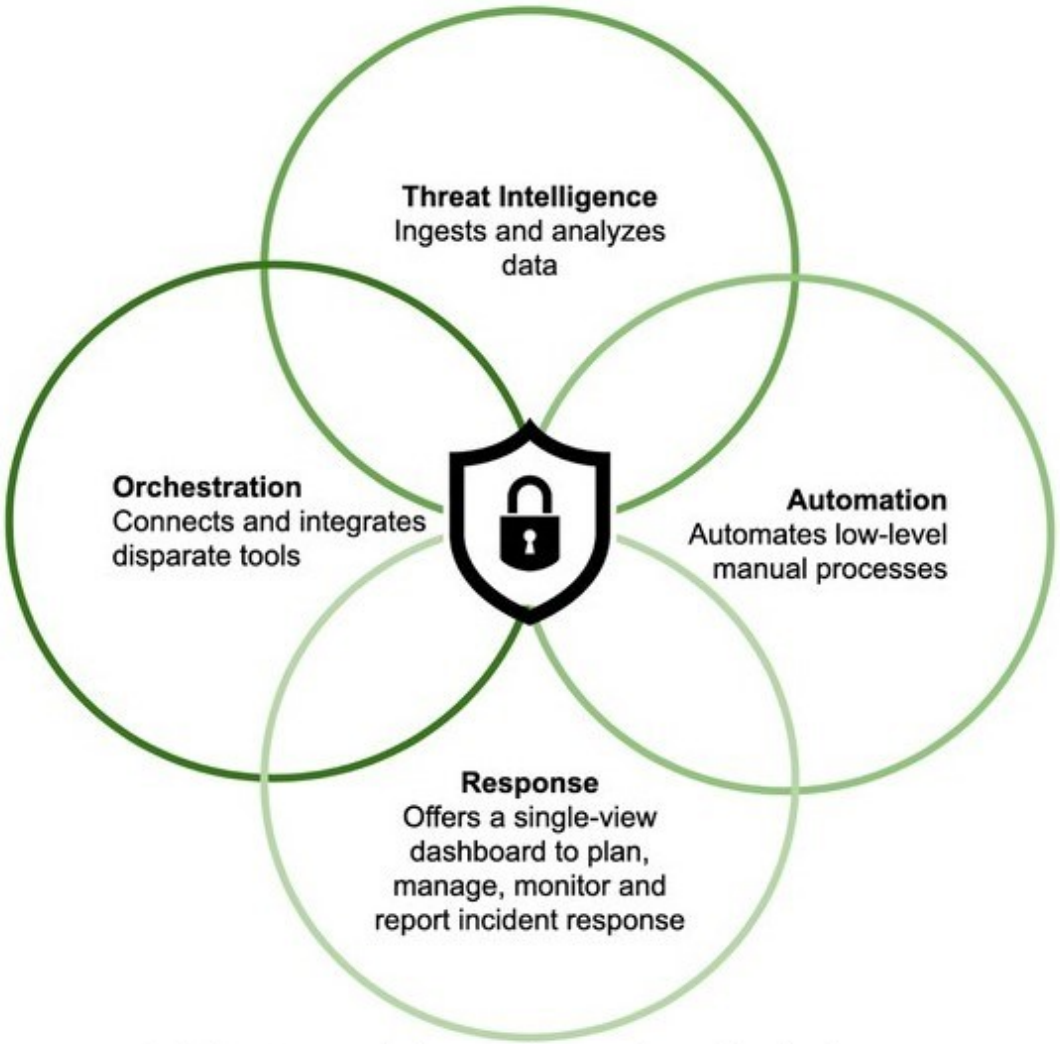
Agenti mohou být implementováni v nejrůznějších formách, od jednoduchých formalismů typu událost-odpověď nebo stavových strojů až po složité architektury UI s aspirací reprezentovat lidské chování. Roboti nemusí nutně využívat kognitivní funkce, uvažování nebo plánování, aby se chovali efektivně a působili inteligentně, ale výběr optimálních funkcí pro senzory a efektory a jednoduchých pravidel řídících odezvu a vhodná architektura umožňují implementaci robustních a efektivních robotů.

Agenti jsou složité entity, které jsou naprogramovány tak, aby pozorovaly chování uživatelů a učily se z něj. Tento proces učení jim umožňuje předvídat reakce uživatelů a případně manipulovat s jejich chováním. Zatímco roboti jsou navrženi tak, aby neškodili lidem, agenti jsou jiní. Zastupují zájmy svých vlastníků a mohou sledovat cíle, které jsou v rozporu se zájmy ostatních uživatelů. Etické chování je náročné univerzálně vynutit pomocí pravidel. Uživatelé proto musí být obezřetní, když poskytují příliš mnoho informací systémům založeným na agentech, které jsou mimo jejich kontrolu. To platí zejména pro elektronické obchodování, kde integrace systémů zpracovávajících data otevírá možnost testování a modelování chování a stavu partnerského nebo dokonce konkurenčního systému. Pokud se například zjistí, že výrobní zařízení dodavatele nejsou plně využita, je pravděpodobné, že lze vyjednat nižší ceny.

Hodnocení výkonu agenta je založeno na výhodách, které uživatel a vlastník vnímají z jeho používání. Náklady, které agentům vznikají, jsou obecně nízké na ukládání dat a výpočetní techniku, ale vysoké na komunikaci s uživatelem, ostatními agenty a okolím. Hlavním cílem agentů je snížit komunikační úsilí, které uživatel potřebuje k tomu, aby instruoval své systémy k provádění úkolů. Šířka pásma pro komunikaci mezi uživatelem a agentem by měla být výrazně nižší než šířka pásma potřebná pro přímou interakci uživatele s aplikací. Důležitým faktorem, který je třeba vzít v úvahu, je také kvalita výkonu. Kritéria pro hodnocení kvality získaných informací a rozhodnutí učiněných uživatelem zahrnují počet zásahů a chyb ve srovnání s jinými způsoby přístupu ke stejným informacím, četnost chybných rozhodnutí a nevyužitých příležitostí. Navzdory důležitosti přesných a podrobných analýz přínosů agentových aplikací jsou takové analýzy stále vzácné a měly by být začleněny jako zásadní aspekt procesu návrhu agentových systémů.

Existuje více dobrých důvodů pro vývoj distribuovaných systémů, kde je složení

velkých systémů jako souboru agentů atraktivní díky potenciálně otevřené architektuře, kdy lze podle potřeby přidávat nové agenty bez nutnosti upravovat stávající systém. Pokud není pro všechny agenty jeden vlastník, ale v jednom kontextu začne být aktivních mnoho agentů od různých vlastníků, mohou tito agenti komunikovat, vyjednávat a sdílet své zdroje (tj. spolupracovat). Předpokladem pro multiagentní systémy a kooperativní agentové systémy jsou definované komunikační jazyky a pravidla, která upravují spravedlivou a výhodnou výměnu informací mezi agenty. Jeden z příkladů multiagentního systému je SOAR (Security orchestration, automation and response) a jeho hlavní elementy lze vidět na obrázku 1.2.



Obrázek 1.2: Příklad multiagentního systému - SOAR

Výstupy agentů pro lidské uživatele se neomezují pouze na text a obrázky, ale mohou zahrnovat přirozený jazyk, řeč a animace využívající techniky virtuální reality. Jiným přístupem je využití mentalistických konceptů při návrhu funkcí agentů, jako je motivace, které napodobují komunikační chování lidí, což některým uživatelům usnadňuje porozumění zprávám.

1.1.3 Kde se dají autonomní agenti využít

Využití technologie agentů v různých aplikačních oblastech představuje zajímavé možnosti, jak rozšířit nebo dokonce nahradit přímou interakci s lidmi. Agenti vynikají v situacích, které vyžadují nepřetržité sledování datových toků, rychlé a opakované reakce na události, statistickou analýzu a detekci vzorů v obrovském množství dat a rychlý přístup k databázím.

World Wide Web a další sítě představují oblasti, které jsou pro využití agentů obzvláště zajímavé vzhledem k obrovskému množství dostupných informací, potřebě vyhledávat a vybírat data a lákavosti automatických reakcí. Přínosy agentů jsou různorodé, včetně odlehčení uživatelům od všedních úkolů a zvýšení jejich celkové výkonnosti. Výzkumné studie ukázaly, že uživatelé jsou obecně spokojeni s aktivní podporou poskytovanou agenty v situacích, kdy se cítí bezmocní. Uživatele však mohou snadno iritovat mnohomluvné a nadbytečné informace, pokud mají vyspělé znalosti o dané oblasti. Aby se předešlo takovým negativním reakcím, měli by agenti zahrnovat strategie a prahové hodnoty pro prezentaci informací vhodným způsobem.

Agenti také mohou pozorovat chování uživatelů a mají přístup ke všem systémovým informacím, interpretují požadavky uživatelů, navrhují akce a automatizují rutinní funkce na základě pozorování předchozího chování uživatelů a znalosti výchozích a standardních možností. Nevýhodou však je, že uživatelé si již nemusí být vědomi dostupných funkcí, což jim nedává možnost naučit se je používat. Pro podporu přístupu k rozsáhlým informačním katalogům, jako je trh s CD a knihami nebo vědeckými publikacemi, lze využít agenty, kteří od uživatelů přijímají pokyny v podobě klíčových slov, odkazů na relevantní podobné položky a uživatelských profilů odvozených z předchozích požadavků na aktivní vyhledávání a selektivní zobrazování informací.

Jiní agenti, jako jsou weboví pavouci využívaní vyhledávači, aktivně analyzují a indexují obrovské množství informací a tento index zpřístupňují pro dotazy, což vede k mnohem rychlejším odpovědím, než kdyby se muselo provádět úplné vyhledávání v reakci na jednotlivé dotazy.

V prostředí otevřeného elektronického obchodu mohou agenti plnit některé funkce trhu, vyjednávat ceny a podmínky a optimalizovat logistiku. Nejprve je však třeba vyřešit technické a právní otázky a stanovit vhodnou formalizaci funkcí trhu. Návrh systémů založených na agentech také podporuje rozvoj lepšího porozumění tržním mechanismům.

Pomocné funkce v uživatelských rozhraních mohou zahrnovat agenty, kteří analyzují předchozí vstupy uživatele a poskytují uživateli specifickou pomoc v kontextu jeho aplikace. Specifičtější podpora výkonu využívá databázi postupů specifických pro určitou společnost, složité technické zařízení nebo aplikační oblast. Podpora výkonnosti zahrnuje vedení uživatelů k výběru a provádění předepsaných postupů, zbavuje je nutnosti si tyto postupy zapamatovat a dříve procvičovat a také standardizuje postupy a opakovaně využívá znalosti v rámci organizace.

Důležitým trendem je zpřístupnění výuky a školení uživatelům, které jsou specifické pro jejich potřeby, někdy označované jako "učení právě včas". Archivace relevantních znalostí (knowledge management) je často nedostatečná a učící se vyžaduje

též pokyny, které ho nasměrují k nejvhodnějším informacím a umožní mu vhodně vyhodnotit jeho pokrok.

Další z oblastí, kde se dají agenti využít jsou simulátory. Operátoři odpovědní za úkoly kritické z hlediska bezpečnosti, jako je řízení elektrárny nebo pilotování, jsou intenzivně školeni na simulátorech. V situacích, kdy povaha úkolu zahrnuje spolupráci s dalšími osobami, jsou tyto osoby implementovány jako agenti, kteří představují širokou škálu lidského chování.

Nakonec se například dají autonomní agenti využít ve videohrách. Fanoušci různých her (především strategií) vytvořily autonomní agenty, kteří hrají danou hru za vás. Jiným typem využití se zabývalo studio Monolith Productions, které použilo autonomního agenta pro automatické vytváření nových postav s různorodými osobnostmi.

1.2 Typy algoritmů/systémů pro tvorbu agentů

1.2.1 Rozhodovací stromy

V oblasti rozhodovací analýzy lze vizuálního a explicitního znázornění rozhodování dosáhnout pomocí stromové struktury. Tento stromový model, trefně nazvaný rozhodovací strom[19][20][22], slouží jako nástroj pro orientaci ve složitých rozhodovacích scénářích. Teorie rozhodování, odlišná od analýzy dat, umožňuje rozhodování nezávisle na referenčních datech.

Jedna z cenných aplikací rozhodovacích stromů spočívá v jejich přizpůsobení datovým souborům, což usnadňuje interpretaci a predikci dat. Tyto stromy rozdělující data, známé také jako prediktivní modely, nacházejí uplatnění v učení pod dohledem. Při učení pod dohledem se používá soubor vstupních proměnných tzv. „prediktorů“ k předvídání hodnot jedné nebo více cílových proměnných, neboli k dosažení „výsledků“. Prediktivní model zahrnuje mapování vstupních proměnných na cílovou proměnnou.

Základním cílem je sestavit model, který dokáže předpovědět hodnotu cílové proměnné na základě extrakce jednoduchých rozhodovacích pravidel ze vstupních proměnných. K odhadu prediktivního modelu se používá soubor dat obsahující instance pozorování nebo příkladů zahrnující vstupní a cílové hodnoty. Výsledný přizpůsobený model se obvykle používá k analýze nových případů, u nichž není cílová proměnná známa.

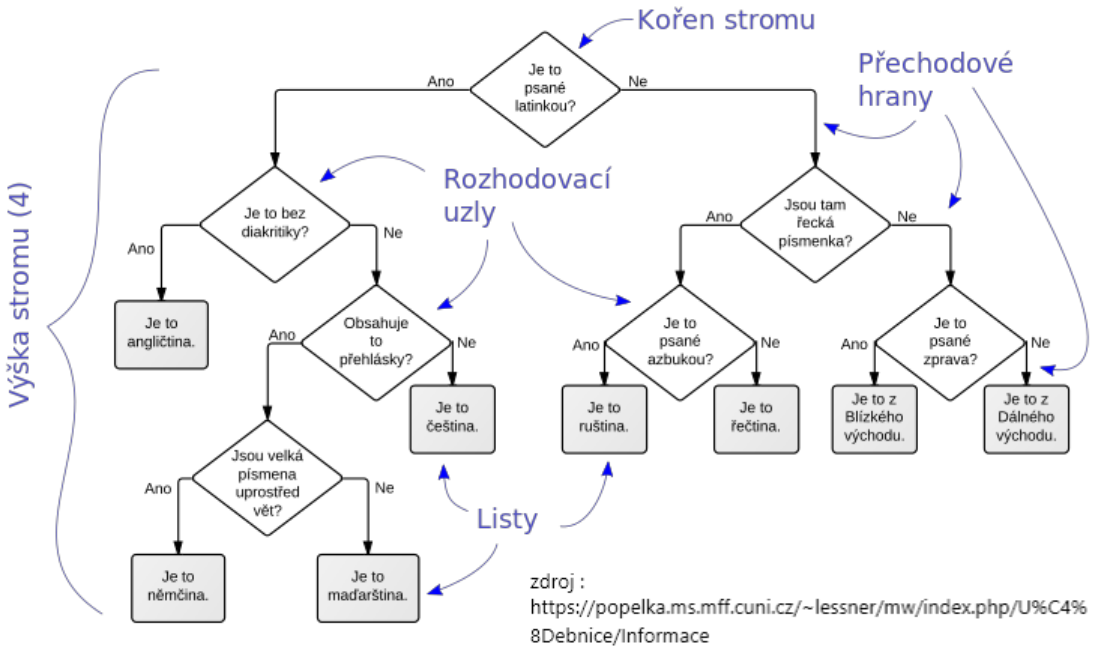
Proces konstrukce rozhodovacího stromu zahrnuje použití řady jednoduchých pravidel pro uspořádání dat do různých skupin. Každé pravidlo funguje tak, že přiřadí pozorování do skupiny na základě hodnoty jedné vstupní proměnné. Tato pravidla jsou aplikována postupně, což vede ke vzniku hierarchické struktury skládající se z vnořených skupin. Tato hierarchická struktura se označuje jako strom, přičemž každá jednotlivá skupina se označuje jako uzel.

Počáteční uzel, známý jako kořenový uzel, představuje celý soubor dat. Při aplikaci

pravidel se vytvářejí nové uzly, které tvoří větve vycházející z uzlu, který je vytvořil. Koncové uzly na konci větví se nazývají listy. V každém listu je učiněno rozhodnutí, které je následně aplikováno na všechna pozorování v rámci tohoto listu. Povaha rozhodnutí se liší v závislosti na kontextu. V oblasti učení pod dohledem odpovídá rozhodnutí předpovězené hodnotě.

Rozhodovací strom lze využít pro několik úloh, včetně klasifikace pozorování na základě hodnot nominálních, binárních nebo ordinálních cílů, předpovídání výsledků pro intervalové cíle nebo určení vhodného rozhodnutí při zadání konkrétních alternativ. Při vizualizaci strom zobrazuje počáteční rozdělení skupiny jako větve vycházející z kořenového uzlu, zatímco následná rozdělení jsou reprezentována větvemi vycházejícími z uzlů umístěných na starších větvích.

Na obrázku 1.3 je příklad rozhodovacího stromu předpovídající typ jazyka pomocí mnoha booleovských vstupů. Rozhodovací uzly obsahují různé pravidla, pomocí kterých dokáže strom predikovat typ jazyka. Listy stromu jsou konečné skupiny, nerozdělené uzly. Aby byl strom užitečný, musí být data v listu podobná vzhledem k nějakému cílovému měřítku, takže strom představuje rozdělení směsi dat do očištěných skupin.



Obrázek 1.3: Příklad rozhodovacího stromu

Rozhodovací stromy se používají pro klasifikační a regresní úlohy. Mohou pracovat s kategoriálními i spojitými predikčními proměnnými. Prostor prediktorů zahrnuje všechny kombinace proměnných. Model rozděluje prostor prediktorů do nepřekrývajících se skupin, které představují listy stromu. Rozdělování začíná kořenovým uzlem, který obsahuje všechna data, a pokračuje, dokud není splněno kritérium zastavení. V každém kroku se rodičovský uzel rozdělí na podřízené uzly pomocí primárního pravidla rozdělení, které minimalizuje variabilitu proměnné odezvy. Výběrem pravidla rozdělení se řídí měřítka jako Giniho index, entropie nebo reziduální součet

čtverců.

Stromové modely se trénují na základě známých hodnot odezvy a poté se používají k hodnocení nových dat. Klasifikační stromy klasifikují pozorování na základě nejčastější odpovědi v listech, zatímco regresní stromy předpovídají průměrnou odpověď v listech. Pravidla rozdělení, včetně primárních, náhradních a výchozích pravidel, umožňují bodování nových dat.

Sestavení rozhodovacího stromu zahrnuje zpočátku růst plného stromu, který může nadměrně vyhovovat trénovacím datům. Aby se zabránilo nadměrnému přizpůsobení, je plný strom ořezán, aby se našel menší, vyvážený podstrom. K nalezení nejlepšího podstromu se běžně používají metody prořezávání na základě nákladové složitosti a prořezávání pojmenované C4.5.

Stromově strukturované modely nabízejí výhody v oblasti interpretovatelnosti a vizualizace, zejména u malých stromů. Dobře se škálují na velká data a pomocí náhradních rozdělení zvládají chybějící hodnoty. Mají však svá omezení, například drobné změny dat mohou vést k odlišným rozštěpením, což zhoršuje interpretovatelnost modelu.

1.2.2 Neuronové sítě

Koncept „vyvíjejícího se“ nervového systému předpokládá, že mozek je plastický, což mu umožňuje přizpůsobit se okolnímu prostředí. Tato plasticita hraje klíčovou roli ve fungování lidských neuronů i umělých neuronových sítí, jejichž cílem je modelovat mozkové úlohy a funkce. Neuronové sítě[1][24] jsou ve své obecné podobě stroje určené k napodobování mozkových procesů a mohou být realizovány pomocí elektronických součástek nebo simulovány na digitálních počítačích.

Pro dosažení vysokého výkonu se neuronové sítě spoléhají na rozsáhlé propojení jednoduchých výpočetních buněk, často označovaných jako „neurony“ nebo „výpočetní jednotky“. Neuronovou síť lze v podstatě definovat jako adaptivní stroj charakterizovaný masivně paralelními distribuovanými výpočetními jednotkami schopnými získávat a uchovávat zkušenostní znalosti. Získávání znalostí probíhá prostřednictvím procesu učení, zatímco ukládání je usnadněno silou spojení mezi neurony, známou jako synaptické váhy.

Proces učení v neuronové síti je řízen učícím se algoritmem, který je zodpovědný za systematickou změnu synaptických vah k dosažení požadovaného cíle návrhu. Tato modifikace vah je v souladu s tradičními přístupy v teorii lineárních adaptivních filtrů, které byly úspěšně aplikovány v různých oblastech. Neuronové sítě mají navíc jedinečnou schopnost modifikovat svou vlastní topologii, inspirovanou skutečností, že lidský mozek prochází změnami, jako je odumírání neuronů a růst nových synaptických spojení.

Výpočetní výkon neuronových sítí[11] vyplývá z jejich paralelní distribuované struktury a schopnosti učit se a zobecňovat. To umožňuje neuronovým sítím aproximovat řešení složitých problémů, které jsou jinak neřešitelné. Složité problémy se rozkládají na jednodušší úlohy a neuronovým sítím se přidělují úkoly, které odpovídají jejich

schopnostem.

Složité problémy se rozkládají na jednodušší úlohy a neuronovým sítím se přidělují úkoly, které odpovídají jejich schopnostem. Je důležité poznamenat, že vytvoření počítačové architektury, která skutečně napodobuje lidský mozek, je značnou výzvou.

Neuronové sítě mají několik cenných vlastností a schopností:

1. Nelinearita:
    - Umělé neurony mohou být lineární nebo nelineární, a když jsou propojeny v neuronové síti, vykazují distribuovanou nelinearitu. Tato vlastnost je důležitá zejména při práci s inherentně nelineárními vstupními signály.
2. Mapování vstupů a výstupů:
    - Neuronové sítě se mohou učit prostřednictvím učení pod dohledem, kdy jsou synaptické váhy upravovány na základě označených trénovacích příkladů. Síť konstruuje vstupně-výstupní mapování pro daný problém.
3. Adaptivita:
    - Neuronové sítě mohou přizpůsobovat své synaptické váhy změnám v prostředí, což jim umožňuje pracovat v různých podmínkách a zvládat nestacionární prostředí. Dosažení rovnováhy mezi adaptabilitou a stabilitou je pro robustní výkon rozhodující.
4. Důkazová odezva:
    - Neuronové sítě mohou poskytovat informace nejen o výběru vzoru, ale také o důvěře v učiněné rozhodnutí, což zlepšuje klasifikační výkon.
5. Kontextové informace:
    - Znalosti jsou přirozeně reprezentovány ve struktuře a aktivačním stavu neuronové sítě, což umožňuje zohlednit kontextové informace.
6. Odolnost vůči poruchám:
    - Neuronové sítě vykazují za nepříznivých provozních podmínek plynulou degradaci výkonu. I při poškození jednotlivých neuronů nebo spojení nedochází k závažnému zhoršení celkové odezvy sítě.

Neuron, základní jednotka pro zpracování informací v neuronové síti, zahrnuje tři základní prvky neuronového modelu, z nichž každý přispívá k jeho funkčnosti:

1. Synapse:
    - Tyto spojovací články mají individuální váhy nebo síly. Když signál dorazí do synapse připojené k neuronu, je vynásoben synaptickou váhou. U umělých neuronů mohou synaptické váhy zahrnovat záporné i kladné hodnoty.

2. Sčítačka(Adder):

    - Jejím úkolem je sčítat vstupní signály vážené odpovídajícími synaptickými silami neuronu. Tento proces je znám jako lineární kombinace.

3. Aktivační funkce:

    - Tato funkce omezuje výstupní amplitudu neuronu, čímž účinně stlačuje signál v rámci konečného rozsahu. Působí jako omezující faktor, který kontroluje přípustný rozsah amplitudy výstupního signálu.

Struktura neuronů v neuronové síti je úzce spjata s algoritmem učení použitým pro trénování sítě. Proto můžeme učící algoritmy používané při návrhu neuronových sítí označit jako strukturované. Obecně řečeno můžeme architektury sítí rozdělit do tří různých tříd:

1. Jednovrstvé sítě s dopřednou vazbou:

    - Tyto sítě se skládají z jedné vrstvy neuronů, přičemž informace proudí jedním směrem od vstupu k výstupu. Mají jednoduchou a přímou strukturu.

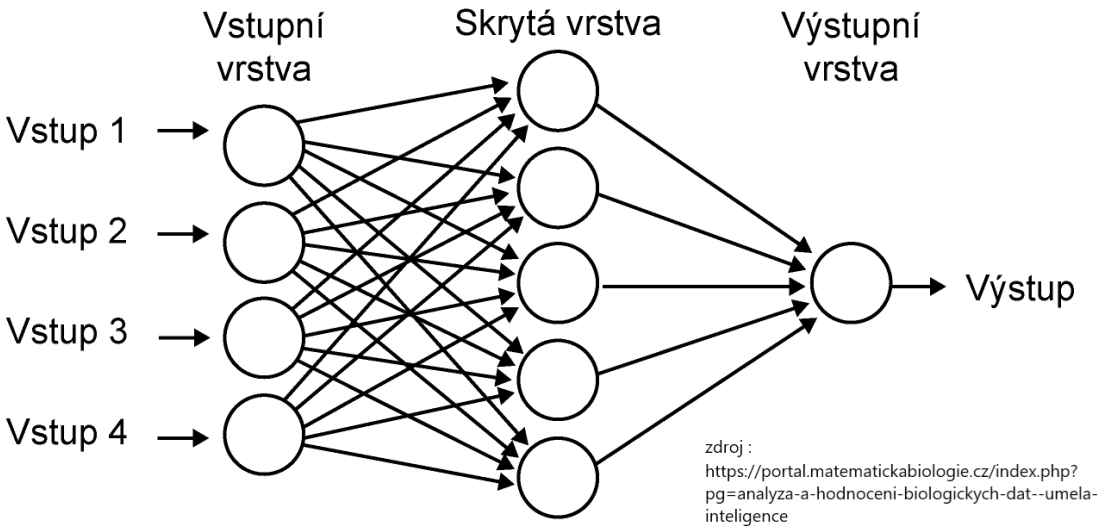
2. Vícevrstvé dopředné sítě:

    - Tyto sítě zahrnují více vrstev neuronů, které tvoří hierarchickou strukturu. Informace proudí vrstvami směrem dopředu, přičemž každá vrstva zpracovává a transformuje vstup z předchozí vrstvy.

3. Rekurentní sítě:

    - V těchto sítích existují zpětnovazební spojení, která umožňují tok informací v cyklech nebo smyčkách. Tato architektura umožňuje síti vykazovat dynamické chování a uchovávat paměť na předchozí vstupy.

Výběr architektury sítě závisí na konkrétní úloze a požadovaných schopnostech neuronové sítě. Na obrázku 1.4 je příklad vícevrstvé neuronové sítě.



Obrázek 1.4: Příklad vícevrstvé neuronové sítě

1.2.3 Genetické algoritmy

Genetické algoritmy[13][23], které čerpají inspiraci z evoluce, zahrnují rodinu výpočetních modelů. Tyto modely využívají datovou strukturu podobnou chromozomu k zakódování potenciálních řešení konkrétních problémů. Použitím rekombinačních operátorů se zachovávají kritické informace. Genetické algoritmy jsou běžně vnímány jako optimalizátory funkcí, jejich použitelnost se však rozšiřuje na širokou škálu problémů.

Implementace genetického algoritmu začíná populací „chromozomů“ („genotypů“), často náhodně generovaných. Následuje vyhodnocení těchto struktur a poté přidělení reprodukčních příležitostí (fitness funkce). Chromosomy představující lepší řešení cílového problému mají větší šanci na reprodukci, zatímco horší řešení má šanci menší. Hodnocení kvality řešení se obvykle týká stávající populace. V širším smyslu lze za genetický algoritmus považovat jakýkoli model založený na populaci, který využívá operátory selekce a rekombinace ke generování nových výběrových bodů v prohledávacím prostoru.

Ačkoli se někdy slova vyhodnocení a fitness používají zaměnitelně, je důležité rozlišovat mezi vyhodnocovací funkcí a fitness funkcí používanou v genetickém algoritmu. V tomto kontextu hodnotící funkce, známá také jako účelová funkce, měří výkonnost na základě specifických parametrů. Fitness funkce transformuje tuto míru výkonnosti na reprodukční příležitosti. Hodnocení sady parametrů reprezentované řetězcem je nezávislé na hodnocení jiných řetězců, zatímco fitness funkce je závislá na vztazích k ostatním členům aktuální populace. Fitness funkce je definována jako podíl, kde ohodnocení spojené s řetězcem je průměrným ohodnocením všech řetězců v populaci. Fitness funkce lze také přiřadit na základě pořadí řetězce v populaci nebo pomocí metod výběru vzorků, jako jsou např. „výběrové turnaje“.

Je vhodné vnímat provádění genetického algoritmu jako dvoufázový proces. Začíná

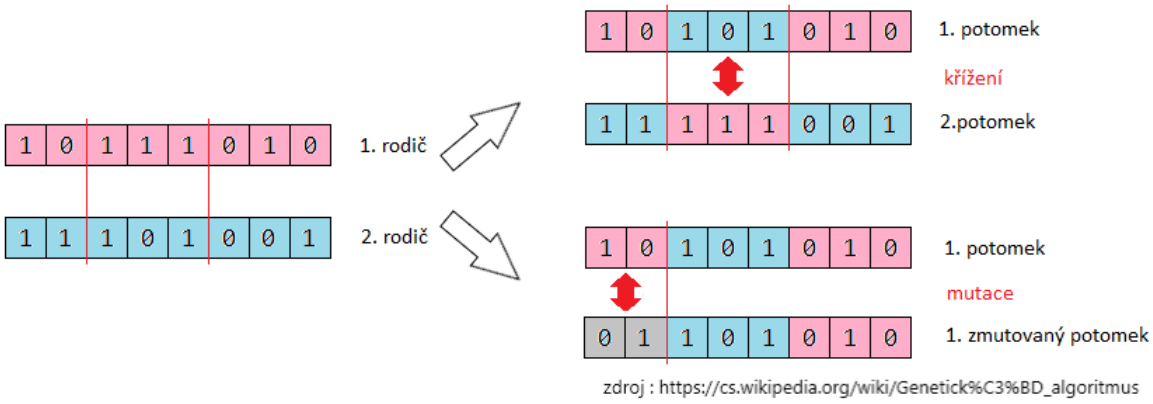
se s aktuální populací, kde se použije výběr pro vytvoření mezipopulace. Následně se na mezipopulaci aplikuje rekombinace (křížení) a mutace, čímž vznikne další populace. Tento přechod od aktuální populace k další populaci představuje generaci v rámci provádění genetického algoritmu.

Proces křížení v genetických algoritmech spočívá ve sloučení genetické informace ze dvou rodičovských řešení za účelem vytvoření potomků. Zahrnuje výběr bodu křížení v rámci rodičovských chromozomů a výměnu genetického materiálu, což vede k vytvoření dvou nebo více potomků. Cílem je spojit výhodné vlastnosti rodičů a potenciálně získat lepší řešení.

Naopak mutace vnáší do jednotlivých chromozomů náhodné změny, které umožňují prozkoumat nové oblasti v rámci prohledávaného prostoru. Jedná se o náhodné úpravy malé části chromozomu, například převrácení bitu nebo změnu hodnoty. Mutace pomáhá udržovat rozmanitost v populaci a zabraňuje tomu, aby algoritmus uvízl v lokálním optimu.

Křížení i mutace hrají v genetických algoritmech zásadní roli. Křížení usnadňuje rekombinaci slibných řešení a průzkum nových území, zatímco mutace vnáší do algoritmu náhodnost a zajišťuje, že algoritmus pokračuje v hledání potenciálně lepších řešení.

Na obrázku 1.5 je zobrazeno křížení dvou rodičů a jejich potomků a také mutace prvního potomka, aby se algoritmus nezasekl v lokálním optimu.



Obrázek 1.5: Příklad úprav chromozomů

Kapitola 2

Návrh autonomního agenta a vizualizace

2.1 Existující řešení ve hrách

2.1.1 Hraní existujících her - StarCraft II

Existuje mnoho autonomních agentů, kteří dokážou hrát hru StarCraft II. Tito agenti dokážou drtivou většinu hráčů hry StarCraft II porazit a díky tomu začali vznikat soutěže o to, kdo dokáže vytvořit nejlepšího autonomního agenta pro hraní této hry. Jeden z příkladů, který je nejvíce známý, je Alphastar[25].

Alphastar je autonomní agent, který představuje průlomový krok v oblasti umělé inteligence a her. Alphastar, vyvinutý společností DeepMind, využívá sílu neuronových sítí a posilování učení, aby vynikl ve složité oblasti hry StarCraft II.

Ve svém jádru Alphastar ztělesňuje architekturu hluboké neuronové sítě, pečlivě vycvičenou kombinací učení pod dohledem a samostatného hraní. Tento mnohostranný agent se skládá ze tří základních součástí: modulu vnímání, modulu politiky a modulu hodnot.

Modul vnímání asimiluje obrovské množství informací souvisejících s hrou, včetně nezpracovaných pixelů obrazovky a dat o stavu hry, což umožňuje agentu Alphastar efektivně vnímat a chápat své prostředí.

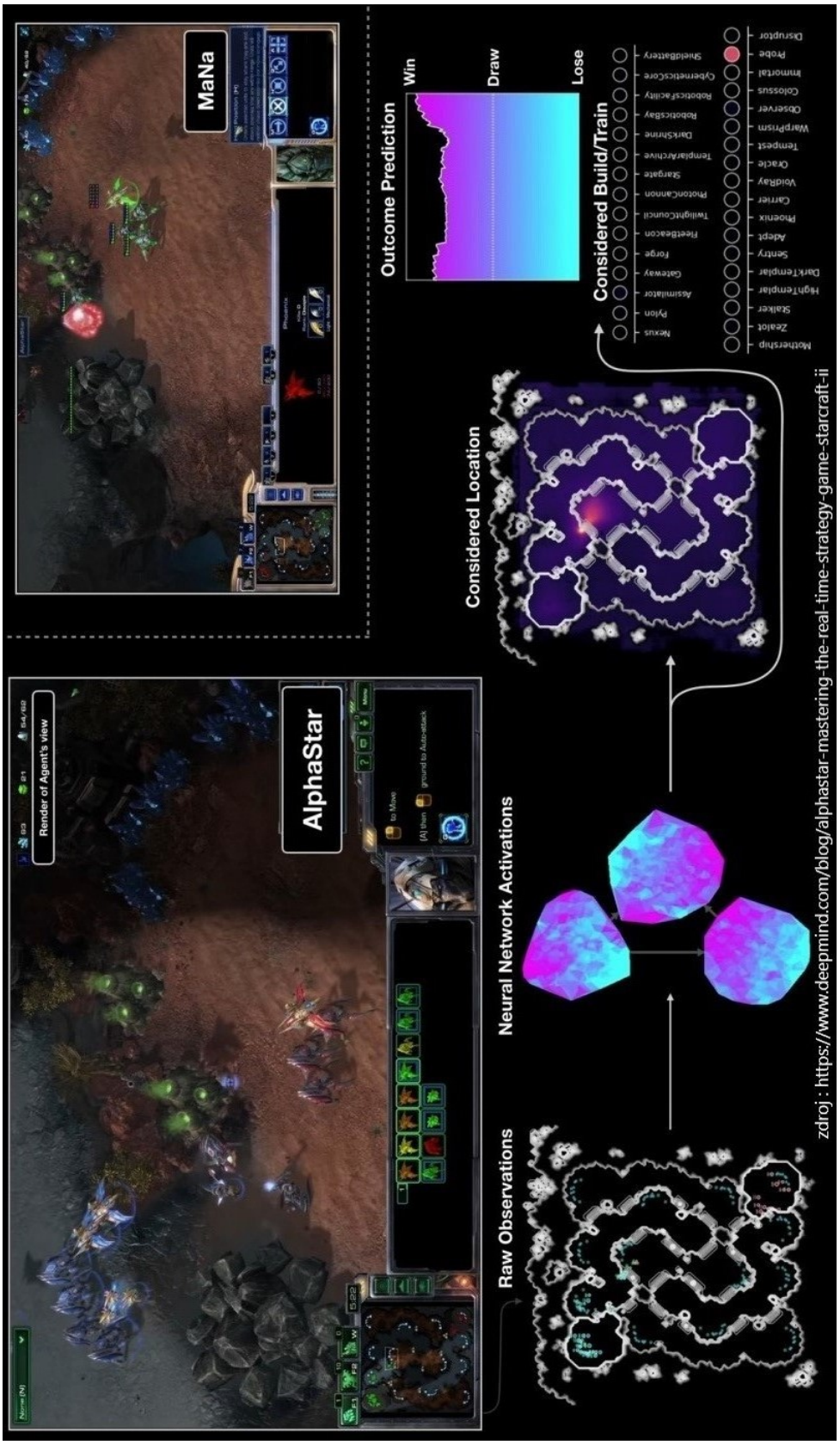
Modul politiky ztělesňuje rozhodovací schopnosti Alphastaru a umožňuje mu formulovat strategie a provádět přesné akce. Díky rozsáhlému tréninku se tento modul učí optimální politiku předpovídáním nejvýhodnějších akcí v daném herním stavu. Modul politiky Alphastar vyniká jak v rozhodování na mikroúrovni, jako je řízení jednotek a jejich umístění, tak v rozhodování na makroúrovni, které zahrnuje správu zdrojů a dlouhodobé plánování.

Modul hodnot slouží jako hodnotitel systému Alphastar a odhaduje hodnotu různých herních stavů. Vyhodnocením potenciálního výsledku budoucích akcí vede hodnotový modul systém Alphastar k přijímání informovaných rozhodnutí, čímž optima-

lizuje jeho hratelnost a přizpůsobivost.

Jedním z pozoruhodných aspektů systému Alphastar je jeho schopnost učit se zcela od nuly, bez jakýchkoli znalostí lidské domény. Prostřednictvím milionů zápasů proti různým soupeřům Alphastar zdokonaluje své strategie, objevuje nové taktiky a postupem času zlepšuje svůj výkon. Tato evoluční cesta umožňuje Alphastaru překonat lidské schopnosti a předvést své umění v jedné z nejsložitějších strategických her v reálném čase, která kdy byla vytvořena.

Pozoruhodná přizpůsobivost a všestrannost Alphastaru vynikne při jeho schopnosti orientovat se v rozsáhlém rozhodovacím prostoru hry StarCraft II. Dokáže pohotově přizpůsobovat své strategie na základě chování soupeřů a měnící se dynamiky hry. Ať už používá chytré průzkumné techniky, provádí složité manévry mikromanagementu nebo vymýšlí účinné protistrategie, hratelnost Alphastaru je důkazem síly pokročilých systémů umělé inteligence.



Obrázek 2.1: Znároznění myšlenkového pochodu autonomního agenta Alphastar

2.1.2 Upravování atributů nepřátel - Shadow of War

Middle-earth: Shadow of War[28] obsahuje druhou verzi převratné herní mechaniky známé jako Nemesis System, která je v herním světě raritou. Tento systém poskytuje četné příběhy, když např. čelíte porážce proti nepříteli z řad orků nebo se rozhodnete pro jejich povýšení. Co tento systém odlišuje od první verze je schopnost vytvářet složité příběhy a vztahy mezi potkanými orky, které vedou ke vzniku pokrevních bratrstev nebo hořké rivalitě.

Systém Nemesis je jádrem Middle-earth: Shadow of War, který hráčům nabízí možnost záměrně či nezáměrně vytvořit ultimátního orkského nepřítele, což zvyšuje náročnost a požitek z hraní. Každý orkský protivník je procedurálně generován pomocí tohoto systému a zaplňuje oblasti, které prozkoumáváte.

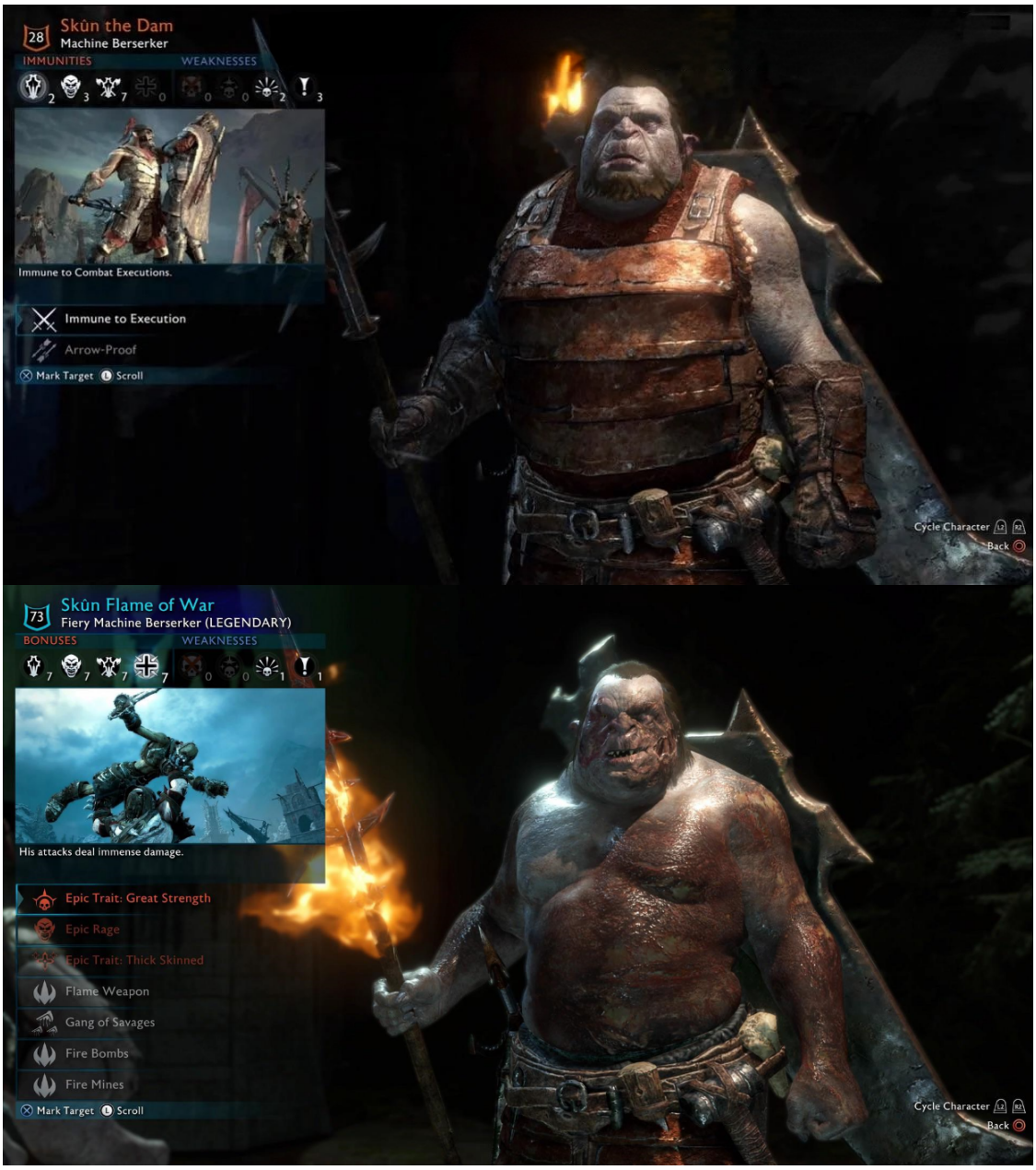
Ve hře mají orkové odlišný vzhled, třídy, vlastnosti, slabiny, silné stránky a osobnosti. Systém Nemesis navíc odhaluje složitou hierarchii orků a rozlišuje je na základní orky, kapitány, vojevůdce a vládce. Jak orkové stoupají v hodnostech tím, že porazí vás nebo své kolegy, ti, kteří dosáhnou hodnosti kapitána nebo vyšší, jsou obdařeni jmény, takže vaše střety s nimi jsou teatrálně dramatické a nebezpečné.

Když vás orkský kapitán porazí nebo vás poníží, získáte příležitost k pomstě prostřednictvím speciální mise. Orkové, kteří vás několikrát porazili, mají větší šanci, že se při vašem případném vítězství vzepřou smrti, což z nich dělá skutečné nepřátele, které se snažíte zlikvidovat. I když se vám podaří orka zabít, jeho příběh nemusí definitivně skončit, protože má schopnost obelstít smrt a navzdory zdánlivému zániku se znovu objevit a zaútočit na vás. Jejich vzkříšení prostřednictvím systému Nemesis jim však propůjčí nové vlastnosti a osobnosti, které se utvářejí na základě zranění, jež utrpěli.

Při prvním povýšení orkové obdrží jméno a sadu náhodně generovaných vlastností. Další pojmenovaní orkové, kterým se podaří vás porazit, postoupí na vyšší úroveň, čímž se zvýší jejich úroveň síly a budou pro ně představovat větší výzvu.

V celé říši Mordoru narazíte na orky známé jako červi, které poznáte podle jejich zelených ikon. Tito červi slouží jako cenný zdroj informací, který vám umožní odhalit totožnost dosud neodhalených orkských kapitánů v oblasti. Poté, co využijete červa k získání informací o orkovi, musíte jej zlikvidovat a vyhledat dalšího, pokud chcete odhalit další kapitány nebo náčelníky. Zjištění identity orků poskytuje důležité znalosti, jako je jejich jméno, úroveň, silné a slabé stránky, což značně usnadňuje jejich porážku při setkání v terénu.

Systém Nemesis také zajišťuje, že každé střetnutí s orky působí hluboce osobně a epicky, protože si vzpomínají na vaše minulá střetnutí, předchozí porážky nebo na zásah svých věrných spolubojovníků, kteří jim přispěchají na pomoc. Dynamické hlasové linky týkající se vaší společné historie výrazně zvyšují osobní vztah k orkovi.



Obrázek 2.2: Horní část je ork hned po náhodném generování a spodní část je ten samý ork, ale po několika bitvách

2.2 Použité technologie, návrh agent a jeho vizualizace

2.2.1 Použitý software k sestrojení vizualice (hry)

Hlavní software, který byl použit ke tvorbě samotné mapy a také většiny základních funkcí je Unreal Engine. Unreal Store byl použit k získání objektů k vytvoření mapy.

Unreal Engine, který je vyvíjen a spravován společností Epic Games, je rozsáhlý ekosystém pro vývoj her. Tento engine byl původně uveden na trh v roce 1998, a v průběhu let prošel několika iteracemi a vylepšeními, jejichž výsledkem je nejnovější verze Unreal Engine 5. Jeho pružnost, přizpůsobivost a robustnost zaujala nespočet vývojářů po celém světě.

Jednou z charakteristických vlastností Unreal Enginu je jeho vizuální skriptovací systém Blueprint, který tvůrcům umožňuje navrhovat herní mechaniky a logiku bez nutnosti rozsáhlých znalostí programování. To demokratizuje vývoj her a zpřístupňuje ho širšímu okruhu kreativních mozků.

Unreal Engine navíc nabízí snadno dosažitelnou grafickou věrnost díky možnostem vykreslování v reálném čase. S tímto enginem můžete snadno vytvářet realistická prostředí, modely postav, dynamické osvětlení a částicové efekty.

Unreal Engine se stal oblíbeným enginem mnoha herních vývojářů díky svému uživatelsky přívětivému rozhraní, snadnému programování a dobré grafice. Možnosti vykreslování v reálném čase v Unreal Engine zaujaly také filmaře a tvůrce obsahu v zábavním průmyslu. Umožňuje virtuální produkci, kdy mohou filmaři zachytit skutečné herce ve virtuálním prostředí, čímž se sníží náklady a čas na výrobu. Vizualizační schopnosti Unreal Engine se rozšiřují i na architektonický a produktový design. Architektům umožňuje vytvářet interaktivní a pohlcující prezentace jejich návrhů, které klientům poskytují realistický zážitek ještě před zahájením stavby. A našel si také cestu do tréninkových simulací pro různá průmyslová odvětví, jako je vojenský, lékařský a průmyslový výcvik. Schopnost enginu replikovat realistické scénáře a interakce zvyšuje efektivitu školicích programů.

Obchod Unreal Store doplňuje engine tím, že funguje jako prosperující tržiště digitálních prostředků, jako jsou 3D modely, textury, animace, zvukové efekty a další. Vedle aktiv nabízí obchod Unreal Store také množství zásuvných modulů a rozšíření, které tvůrcům umožňují snadno přidávat do svých projektů nové funkce a vlastnosti. Tyto zásuvné moduly pokrývají vše od herních mechanik až po vizuální vylepšení a systémy umělé inteligence. Obchod Unreal Store „vzkvétá“ díky příspěvkům komunity, která v něm může sdílet a prodávat své výtvory.

Unity mohl být využit k vytvoření vizualizace místo Unreal Enginu (UE), ale byl zvolen Unreal Engine ze tří hlavních důvodů:

1. Programovací jazyk C++:
    - Jazyk C++ je (kromě nadstavby Blueprint) jediný použitý jazyk v celém

Unreal Enginu. Tento jazyk je velice rychlý, což je u aplikací běžící v reálném čase potřebné. Zároveň je tento jazyk potřebný pro použití knihovny Keras2C (více v použitém softwaru na tvorbu agenta). Díky těmto požadavkům je jazyk C++ potřebný, ale zároveň mám v tomto jazyce nejvíce zkušeností.

2. Programovací nadstavba Blueprint:

   - Tato nadstavba pro programovací jazyk C++ je velice jednoduchá a lehká na používání. Usnadňuje proces tvorby programů/her a dokáže ji používat i člověk, co nikdy v C++ neprogramoval.

3. Unreal Store:

   - Za pomoci Unreal Storu nebyla potřeba vytvářet skoro žádné objekty, které byli použity ke tvorbě mapy, hráčské postavy, zbraní atd. Díky tomu mohla být hra vytvořena velice rychle.

Nakonec jsem zvolil Unreal Engine, protože byl vytvořen jako 3D nástroj pro vytváření her, filmů atd., kde Unity se spíše používá na vytváření 2D nebo 2.5D her. V Unity lze vytvářet 3D hry, ale z mého pohledu je to složitější než v Unreal Enginu.

Unreal Engine má i svoje negativa oproti Unity:

1. Menší komunita:

   - Unreal Engine má mnohem menší komunitu vývojářů než Unity, proto je někdy těžké najít řešení k problému, který může při programování nastat.

2. Náročnější nároky na výkon:

   - Kde doporučený hardware pro Unity je jenom 6 jader, 16Gb paměti a grafická karta GTX 1060, Unreal Engine požaduje alespoň 6 jader, 64Gb paměti a grafickou kartu RTX 2080 Super. Unreal Engine lze spustit na jakémkoliv hardwaru, ale vše se mnohem déle provádí a je také mnohem větší riziko spadnutí programu.

Obrázek 2.3: Ikony všech použitých softwarů a knihoven

2.2.2 Použitý software a knihovny k sestrojení autonomního agenta

Software použitý pro programování pomocí jazyka C++ je Visual Studio. Unreal Engine požaduje tento program, pokud je potřeba naprogramovat specifické funkce do Unreal Enginu. Unreal Enginu je C++, protože jeho programovací jazyk „Blueprint“ je nadstavbou jazyka C++. Jiný programovací jazyk použít nelze.

Visual Studio je integrované vývojové prostředí (IDE) vytvořené společností Microsoft a C++ je univerzální a vysokoúrovňový programovací jazyk. Je rozšířením jazyka C s přidanými funkcemi, jako je např. objektově orientované programování (OOP).

Blueprinty v Unreal Engine představují vizuální skriptovací systém, který vývojářům a návrhářům umožňuje vytvářet herní logiku, interakce a chování bez nutnosti psaní kódu. Blueprinty používají rozhraní založené na uzlech, kde uživatelé propojují uzly a definují tak tok událostí a akcí.

Dalé je knihovna, která se jmenuje Tensorflow a byla použita pro vytvoření neuronové sítě. Jako rozhraní pro použití Tensorflow byl vybrán Keras.

TensorFlow je open-source framework pro strojové učení vyvinutý společností Google. Je zejména používán pro vytváření a trénování modelů strojového učení, zejména modelů hlubokého učení. Pomocí TensorFlow můžete vytvářet neuronové sítě a další výpočetní grafy, což usnadňuje implementaci složitých algoritmů pro úlohy, jako je rozpoznávání obrazu, zpracování přirozeného jazyka atd. Nabízí rozhraní pro různé programovací jazyky, takže je přístupný vývojářům na různých platformách.

Keras je open-source vysokoúrovňové rozhraní pro neuronové sítě napsané v jazyce Python. Původně bylo vyvinuto jako nezávislý projekt, nyní je součástí TensorFlow jako oficiální vysokoúrovňové rozhraní. Keras podporuje jak konvoluční neuronové sítě (CNN), tak rekurentní neuronové sítě (RNN) pro úlohy, jako je rozpoznávání obrazu, zpracování přirozeného jazyka, modelování sekvence na sekvenci atd. Umožňuje také vytvářet vlastní vrstvy, ztrátové funkce a optimalizátory, což vývojářům poskytuje flexibilitu při experimentování a přizpůsobování jejich modelů. Modularita Kerasu umožňuje snadnou opakovanou použitelnost a přenositelnost modelů. Vývojáři mohou vytvářet složité modely sestavováním předem připravených vrstev, což podporuje rozvoj bohatého ekosystému opakovaně použitelných komponent. Keras odbourává velkou část složitosti implementací neuronových sítí, což z něj činí ideální volbu pro rychlé prototypování a experimentování.

Tensorflow Keras model byl poté přeložen do ONNX, což je formát, co požaduje plugin v Unreal Enginu jménem „Neural Network Inference“ (Odvozování neuronových sítí), pomocí kterého mohu natrénovaný model spustit v Unreal Enginu a využít.

ONNX je zkratka pro Open Neural Network Exchange. Jedná se o open-source formát a ekosystém, který má usnadnit přenos natrénovaných modelů hlubokého učení mezi různými platformami, frameworky a hardwarem bez nutnosti časově náročné konverze modelů.

Neural Network Inference (NNI) je plugin pro vyhodnocování neuronových sítí v re-

álném čase v Unreal Engine. NNI podporuje framework ONNX, který dokáže spustit jakýkoli model exportovaný do .onnx ze standardních ML tréninkových frameworků (PyTorch, TensorFlow, MXNet atd.). Plugin podporuje procesorovou inferenci na PC (Windows/Linux/Mac), konzolích (PS5/Xbox řady X) a GPU vyhodnocování v systému Windows, ale pouze při použití rozhraní DirectX 12 API.

2.2.3 Návrh autonomního agenta

Autonomní agent, který bude sestaven v této práci se bude skládat ze tří částí:

1. Základní Logika:
    - Pomocí stromů chování, ve kterém se nachází selektory, sekvence, úkoly (Tasks), dekorátory (Decorators), systém dotazování na prostředí (Environment Query System) a dalších funkcí mimo strom chování se vytvoří logika autonomního agenta v Unreal Enginu, která bude provádět nahodné chození po mapě, pronásledování a útok na hráče a když se bude autonomní agent cítit, že je v ohrožení, se bude schovávat za objekty a z jeho schované pozice střílet na uživatele na určitou dobu. Strom chování je ve své podstatě rozhodovací strom, kde selektory se opakují, dokud jeden z jeho potomků nevrátí úspěch provedení. Sekvence je přesný opak selektorů. Úkoly jsou v podstatě funkce, které se provedou, pokud je splněna podmínka, která se nachází na selektoru, sekvenci nebo na samotném úkolu. Dekorátor je už zmíněná podmínka, která musí vrátit pravdu, jinak se uzel nebo uzly pod dekorátorem neprovedou. System na dotazování se na prostředí umožňuje agentům vyhodnocovat prostředí a rozhodovat se na základě různých faktorů, jako je vzdálenost k objektům, přímá viditelnost nebo vlastní podmínky specifické pro hru.
2. Rozhodovací strom:
    - Pomocí uzlu pojmenovaného „větev“ (Branch) v jazyku Blueprint vytvořím rozhodovací strom, který se bude ptát na přítomnost zbraně, kterou v ten moment hráč používá a pokud nějakou zbraň hráč má, zeptá se na její typ a podle toho si určí zbraň pro sebe a také si určí vzdálenost, na kterou má hráče pronásledovat a ze které bude taky útočit. Tento strom má taky v sobě část, která se ptá, jestli hráč stále nosí identickou zbraň jako před např. 30 sekundami. Pokud hráč má identickou zbraň, průběh stromu se zruší a autonomní agent bude nadále používat stejnou zbraň a pronásledovat hráče ze stejné vzdálenosti.
3. Rekurentní neuronová síť:
    - Tato síť bude mít vstupní vzor typu matice s pěti řádky a šesti sloupci. Šest sloupců je pro 6 prvků ve vstupníc datech, které budou převedeny do float32. První tři proměnné jsou typu int a znamenají počet poražených nepřátel s respektivní zbraní (brokovnice, automatická puška, odstřelovací puška). Druhá skupina tří proměnných budou booly, které budou

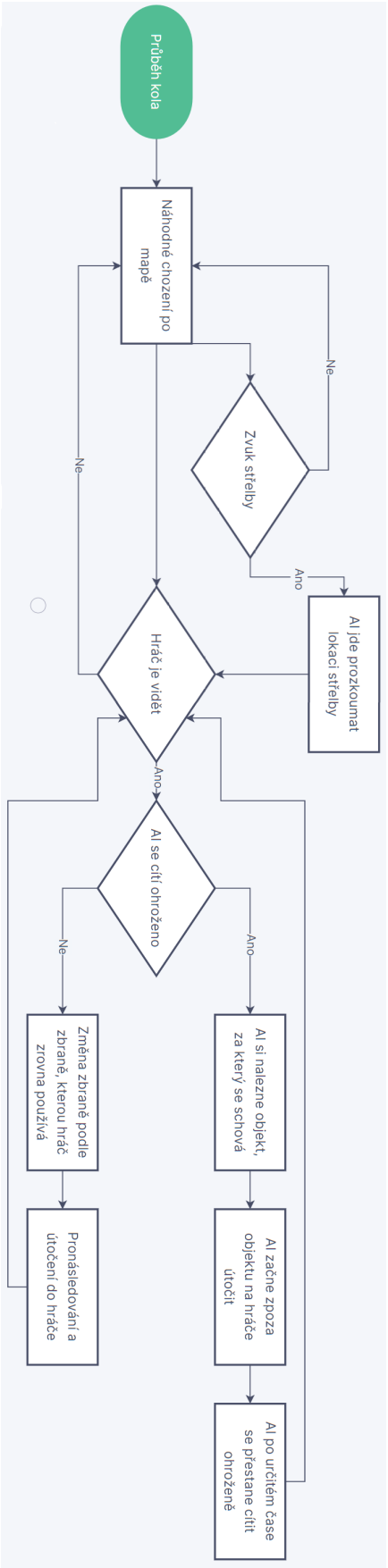
značit, který typ zbraně byl použit jako poslední na zabití posledního nepřítele před koncem kola. Takže např. pole [5 9 16 true false false] se po úpravách a normalizaci promění na pole [0.167 0.3 0.533 1 0 0]. Řádky v matici jsou časové kroky, protože se vložené vektory „recyklují“, proto rekurentní neuronová síť. Takže např. na začátku bude matice mít první řadek plný, ale další čtyři prázdné, ale při druhém projetí bude mít matice na druhém řádku ty prvky, co byli na prvním předtím a na prvním řadku bude mít prvky nové, atd. První vrstva bude rekurentní neuronová síť, kde probíhá hlavní logika. Další vrsta se jmenuje „Dense“, která vezme vstupní data z RNN a na každý vstup aplikuje váhy, následně zkreslující člen a poté použije aktivační funkci k vytvoření výstupu. Hlavním účelem vrstvy je učit se vzorům a vztahům v datech úpravou vah během procesu učení. Výstupů bude deset. Jeden výstup pro každou zbraň ve hře, když bude hráč pořád používat stejnou zbraň, což jsou 3 výstupy, krát tři, když hráč bude mít nejvíce zabití např. s brokovnicí, ale na konci kola bude mít zbraň jinou a poslední 10. výstup je když by hráč měl velice podobné množství zabití se všemi zbraněmi. Nakonec aktivační vrstva bude typu „softmax“. Po vyhodnocení a získání výstupu upravíme atributy AI postavy.

Agent s touto logikou bude moci se náhodně přemisťovat po mapě a reagovat na zvukové jevy. Když je hráč spatřen, přizpůsobí si zbraň a vzdálenost od hráče podle zbraně, kterou v ten okamžik hráč používa a začne ho pronásledovat a útočit po něm z vybrané vzdálenosti. Když se bude agent cítit ohrožen, schová se za objekt, který si on sám vybere a zpoza tohoto objektu začne na hráče útočit. Po určitém čase se ale vrátí k pronásledování hráče. Po každém kole si agent přizpůsobí svoje atributy podle počtu smrtí s každou možnou hráčskou zbraní a také podle toho, jakou zbraň hráč na konci kola držel.

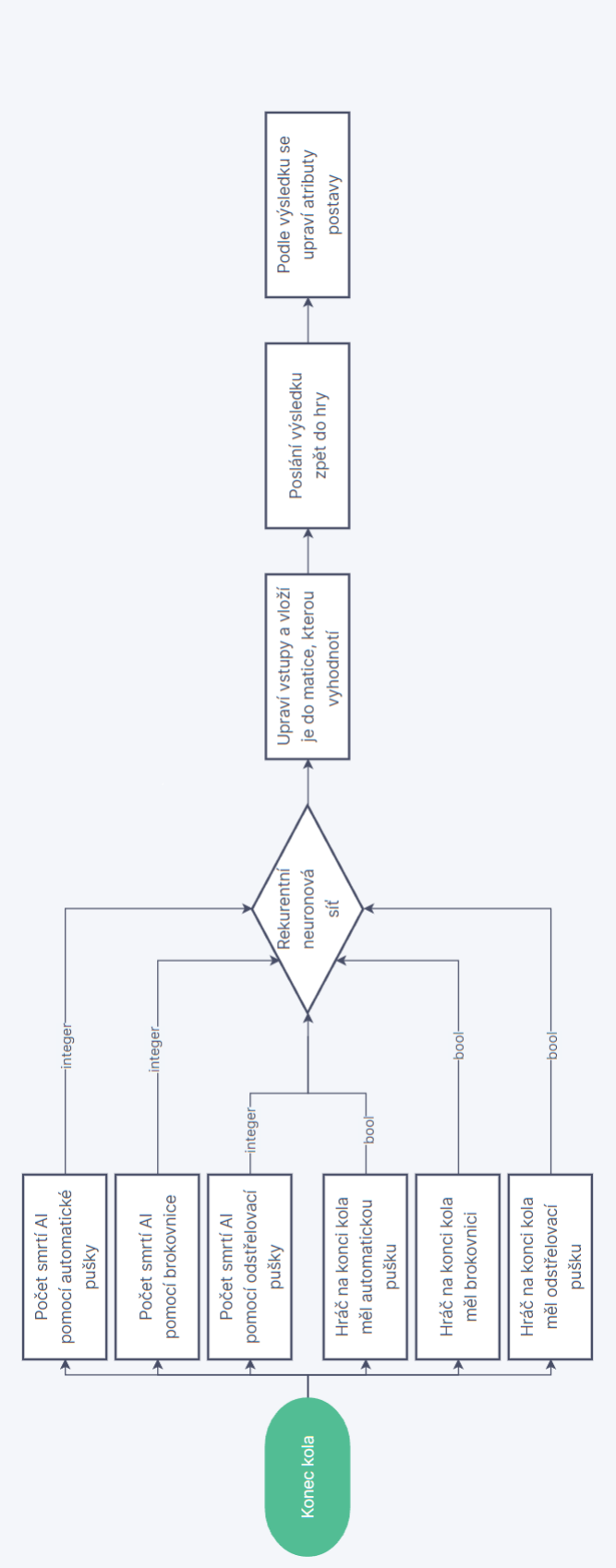
2.2.4 Návrh vizualice (hry)

Tento autonomní agent může být vizualizován mnoha metodami. K jednoduším metodám patří vytvoření projektu v Unity nebo Unreal Enginu a ke složitějším metodám patří čistý kod v rozhraní pro programování aplikací OpenGL nebo DirectX. V tomto případě bude agent vizualizován pomocí Unreal Enginu vytvořením střílečky z pohledu první osoby. Nejdříve se musí vytvořit mapa, po které může hráč a postava, která bude ovládaná autonomním agentem, chodit a bojovat. Pomocí objektů, které lze stáhnout z Unreal Storu, postavím mapu náměstí města, kde ve středu této mapy se bude nacházet park s budovou banky, ve které bude hráč zrozen. Zbraně, se kterými bude moci hráč bojovat se taky nachází v této budově. Mapa také bude obsahovat část, která je pouze obklopená budovy. Tato část mapy je více uzavřená, takže je snažší se dostat do špatné situace, oproti parku. Na postranních silnicích, které budou zablokovány ruinami budov, se budou nacházet body, ze kterých bude mít postava ovládaná umělou inteligencí možnost se zrodit. Na silních budou vraky aut, které bude moci hráč i umělá inteligence využít jako strategický bod nad svým nepřítelem. Na různých vracích nebo také na smetí se bude moci

nacházet oheň, který bude hráče a umělou inteligenci poškozovat, pokud se budou náchazet v dostatečné blízkosti. Hra bude také obsahovat sbíratelých věcí, aby si mohl hráč doplnit životy a nebo náboje. Hráčovi se bude na obrazovce zobrazovat tzv. „průhledový“ displej (HUD), pomocí kterého bude moct sledovat stav svých životů, energie, nábojů atd. S tímto displeje bude spojeno hlavní a pauzové menu. Nakonec tento projekt bude mít logiku kol. Každé kolo bude hráč muset zabít určité množství nepřátel. Když hráč všechny nepřátele zničí, dosavadní kolo se ukončí a započne kolo nové.



Obrázek 2.4: Tento graf znázorňuje myšlenkový pochod autonomního agenta v průběhu kola



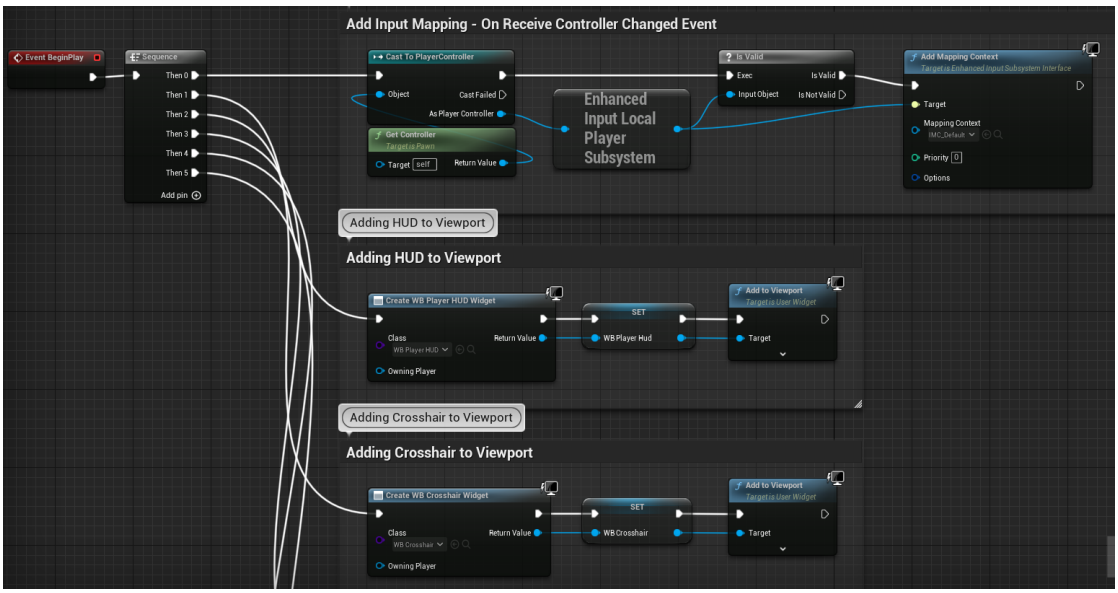
Obrázek 2.5: Tento graf znázorňuje myšlenkový pochod autonomního agenta na konci kola

Kapitola 3

Implementace hry a autonomního agenta

3.1 Základní funkce hry

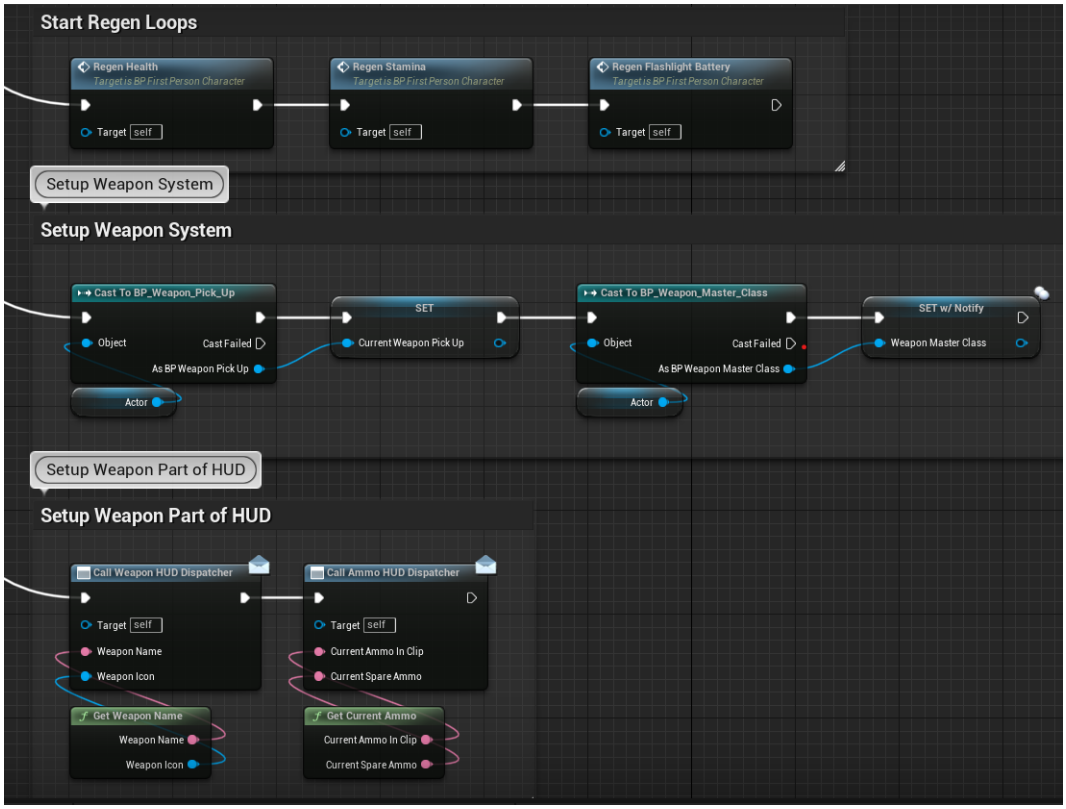
3.1.1 Hlavní postava



Obrázek 3.1:

Obrázek 3.1 ukazuje první část počáteční funkce, která se zapne, když se hra zapne. Ze sekvence vychází 6 menších funkcí. První funkce připojuje mapu vstupů kláves, kterýma ovládáte postavu. Druhá připojuje „průhledový“ displej (HUD) k postavě, aby byl vidět na obrazovce. Třetí připojuje logiku křížku, aby se pohyboval a fungoval na průhledovém displeji.

Obrázek 3.2 je pokračováním předchozího. Čtvrtá funkce zapíná funkce obnovování života, energie a baterky svítilny. Zároveň zapne část průhledového displeje, který

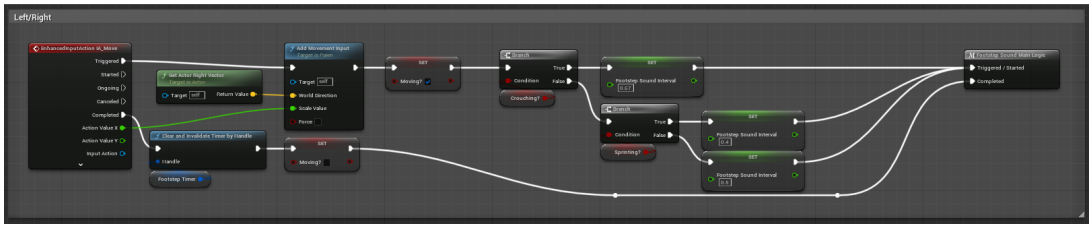


Obrázek 3.2:

se stará o zobrazení těchto údajů. Pátá funkce vytváří proměnné pro propojení blueprintu hráče s blueprintem sběrové logiky a samotné logiky zbranového blueprintu. Poslední funkce zapíná další část průhledového displeje, tentokrát pro ikonu a text podle typu zbraně, co má hráč v ruce.

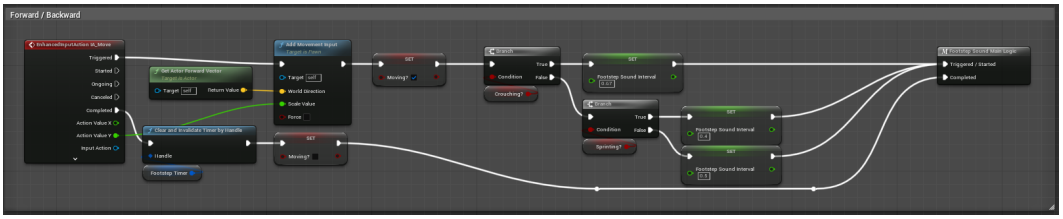
Obrázek 3.3:

Obrázek 3.3 ukazuje propojení pohybu myši s pohybem pohledu ve hře. Levá část je na propojení osy Z s pohybem myši doleva nebo doprava. Pravá část propojuje osu Y s pohybem myši nahoru a dolů.



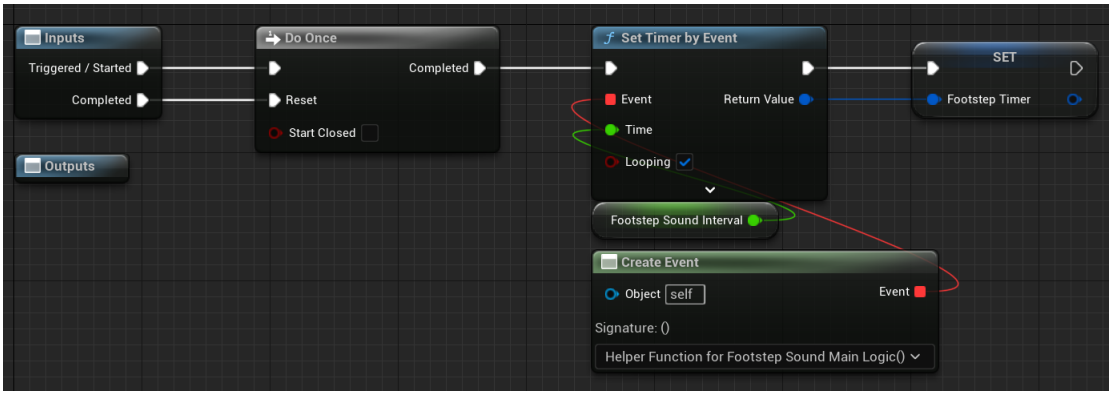
Obrázek 3.4:

Obrázek 3.4 ukazuje logiku pohybu doleva/doprava postavy po mapě. Input má float output, který se dá s vektorem směřujícím doprava do funkce pohybu, nastaví se, že se charakter pohybuje. Dalé se ptáme kvůli funkci zvuku chodiel, jestli běží, nebo je v podřepu. Když přestaneme chodit tak vypneme funkci, která zpracovává zvuky pohybu.



Obrázek 3.5:

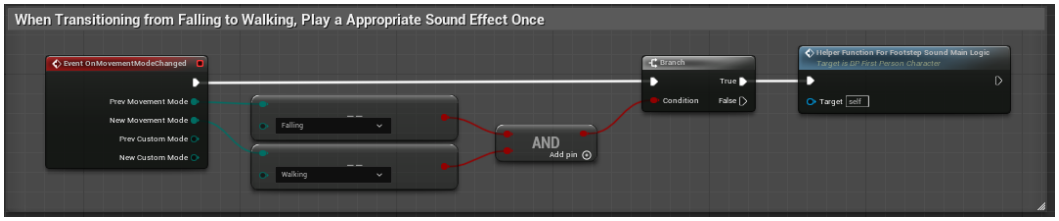
Obrázek 3.5 ukazuje logiku pohybu ve směru dopředu/dozadu postavy po mapě. Logika je identická jako u předchozího obrázku, akorát místo pravého vektoru postavy voláme vektor přední.



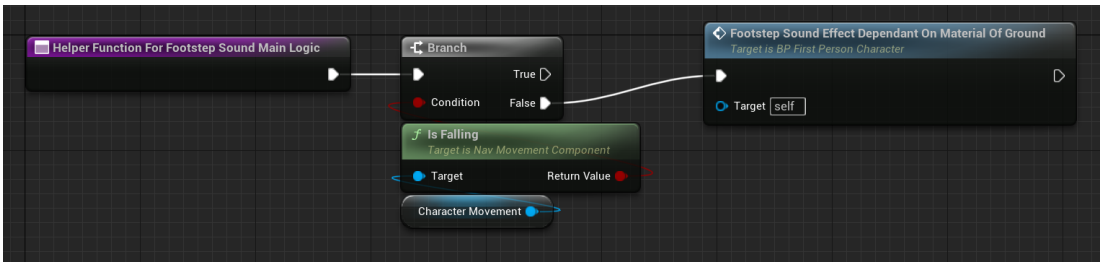
Obrázek 3.6:

Na obrázku 3.6 je ukázaná logika makra, která konstantně opakuje volání funkce, která provádí logiku zvuku pohybu.

Kód na obrázku 3.7 zavolá funkci obnovení zvuku chození, když přestaneme padat.

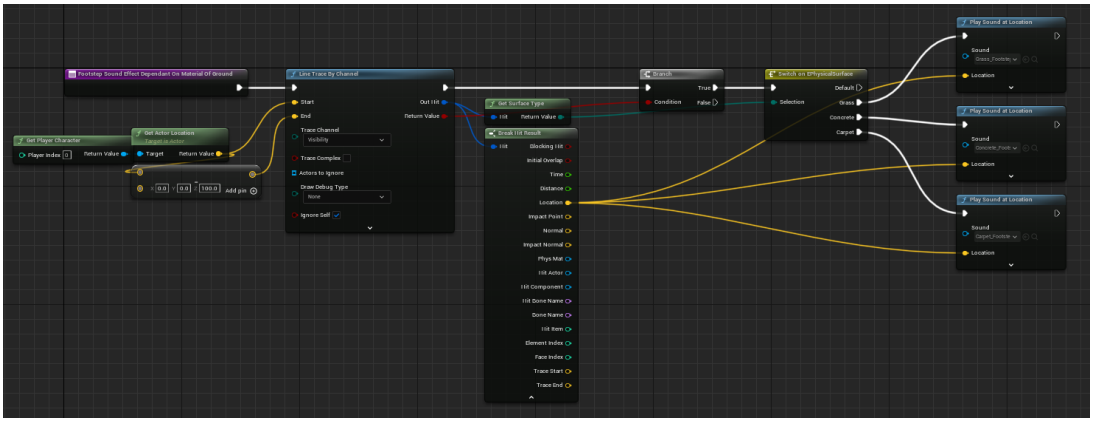


Obrázek 3.7:



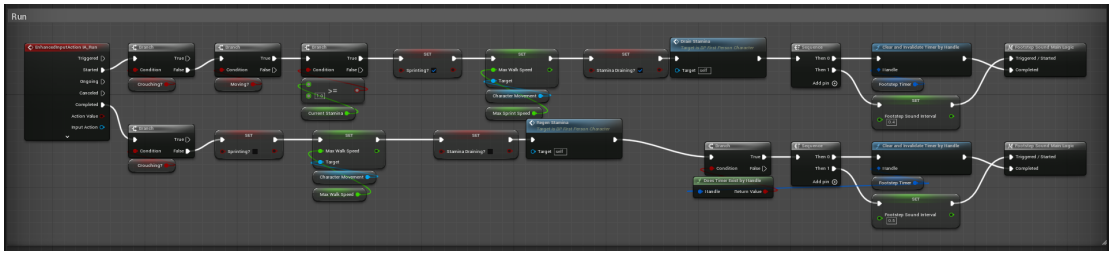
Obrázek 3.8:

Na obrázku 3.8 je kód funkce, která se volá na konci předchozího obrázku. Když hráč nepadá, spustí se zvuky chození.



Obrázek 3.9:

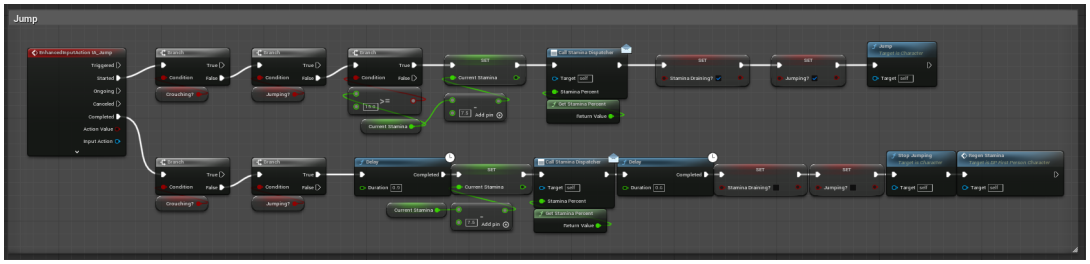
Obrázek 3.9 ukazuje hlavní kód zvuku chození. Vysíláme trasovací linku do země, abychom zjisti typ podlahy a poté pomocí switche vybereme správný zvuk chození.



Obrázek 3.10:

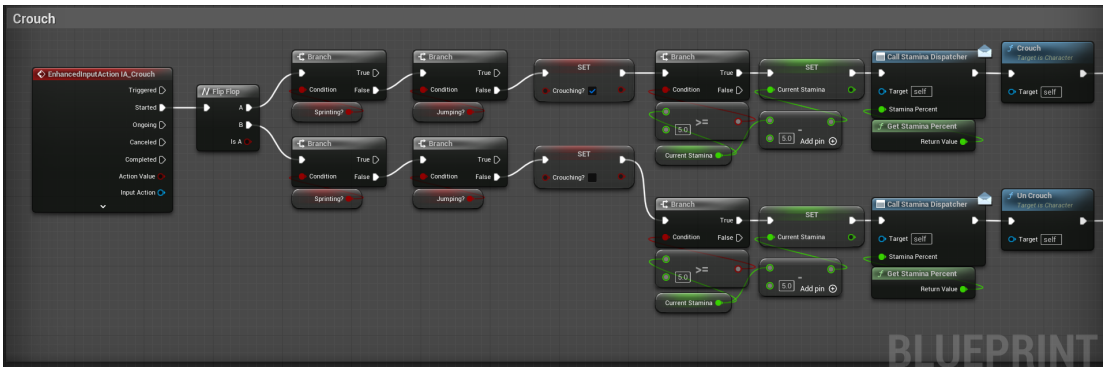
Obrázek 3.10 ukazuje celou logiku běhání. Když nejsme v podřepu, ale pohybujeme se a máme více jak 0 energie, tak začneme pomalu odebírat energii a upravíme

rychlost hraní zvuku chození. Když přestaneme běhat, začne se nám vracet energie a zpomalí se interval zvuku chození

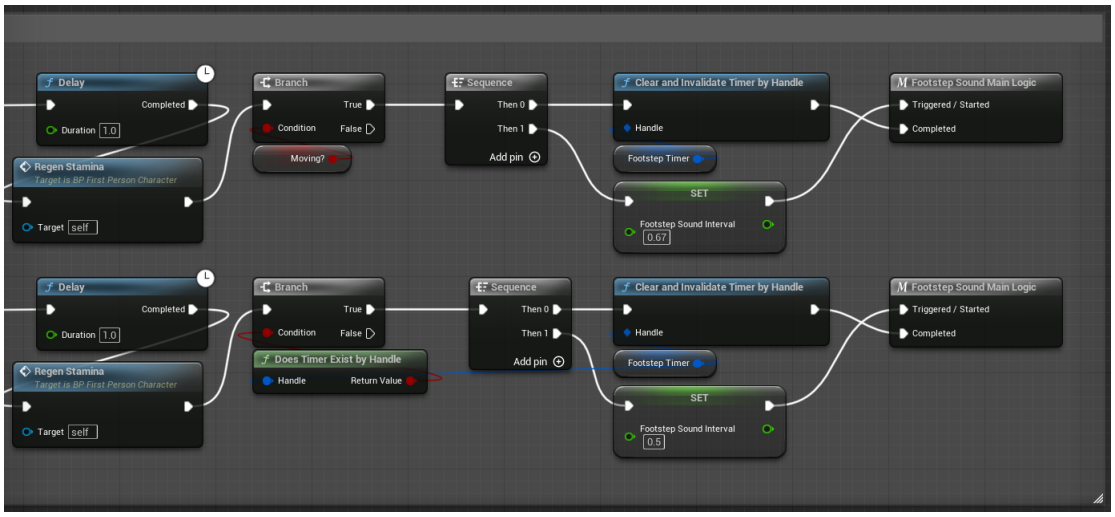


Obrázek 3.11:

Na obrázku 3.11 je logika skákaní. Když nejsme v podřepu a máme víc jak 15 energie, tak vyskočíme. Když dopadneme na zem, tak se nám po chvíli začne regenerovat energie.

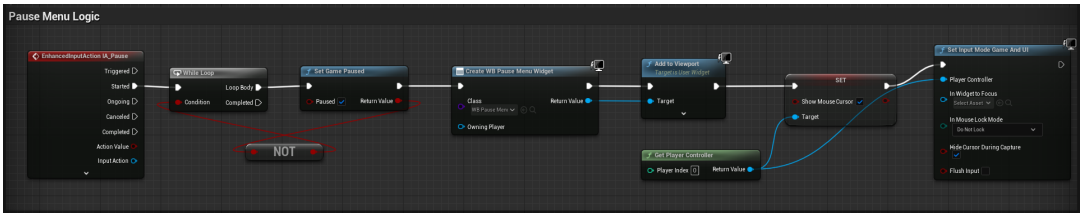


Obrázek 3.12:



Obrázek 3.13:

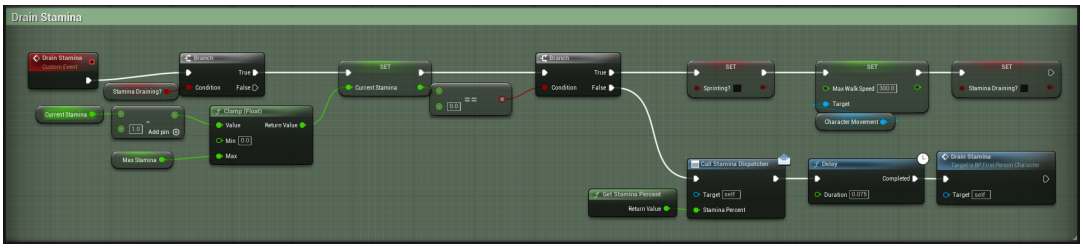
Obrázky 3.12 a 3.13 zobrazuje logiku podředu. Když neběžíme, neskáčeme a máme dostatek staminy, tak se hrác skrčí, po chvíli se začne doplňovat energie a změní



Obrázek 3.14:

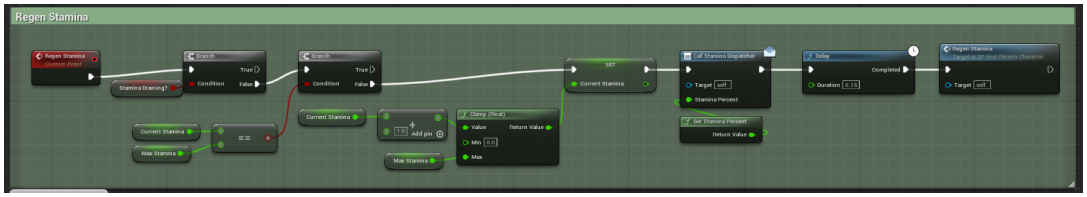
se rychlost zvuku chození. Když se zvedneme, ztratíme zase trochu energie, ale po chvíli se začne obnovovat a zase se upraví rychlost zvuku chození

Obrázek 3.14 obsahuje logiku menu pauzy. Když zmáčkneme příslušné tlačítko, tak se hra dá do pauzy a zobrazí se nám menu.



Obrázek 3.15:

Na tomto obrázku 3.15 je vyobrazena logika odebírání energie. Když je tato funkce zavolána tak každý tik odebere určité množství stamina z maxima. Když máme staminu nulovou, tak přestaneme běhat.

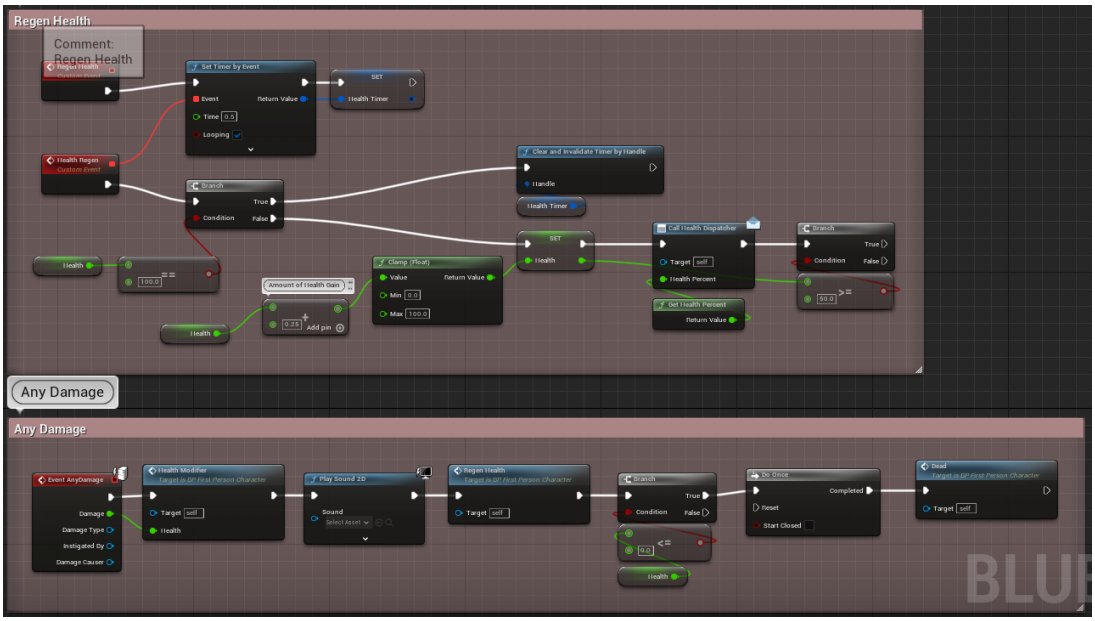


Obrázek 3.16:

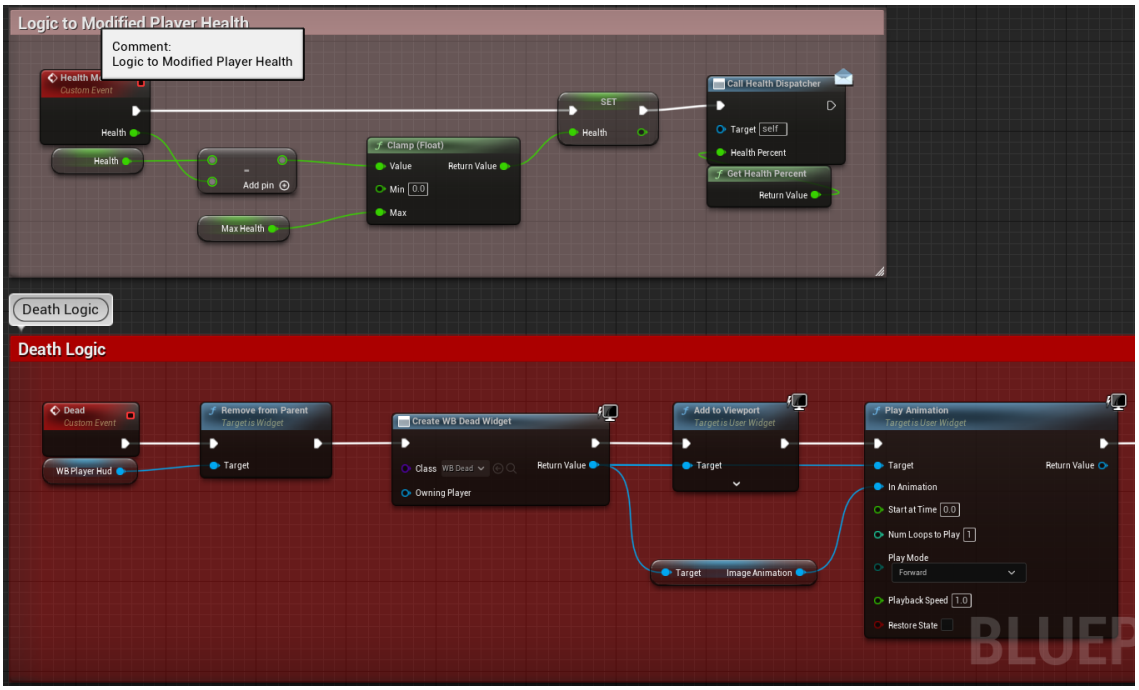
Obrázek 3.16 má logiku regenerace staminy. Funguje stejně jako logika odebírání energie, pouze místo odebírání energie energii přičítá a ještě se ptá, zda energie dosáhla maxima.

Na obrázku 3.17 jsou dvě funkce. Jedna se stará o regeneraci života, když jste ztratil životy při bitvě. Pomalu přičítá malé množství života, dokud počet života nedosáhne maxima. Druhá funkce se stará o zranění. Když dostaneme zranění, tak se nám odeberou životy, ale začnou se hned postupně doplňovat, pokud se život dostane na nulu, tak umřeme.

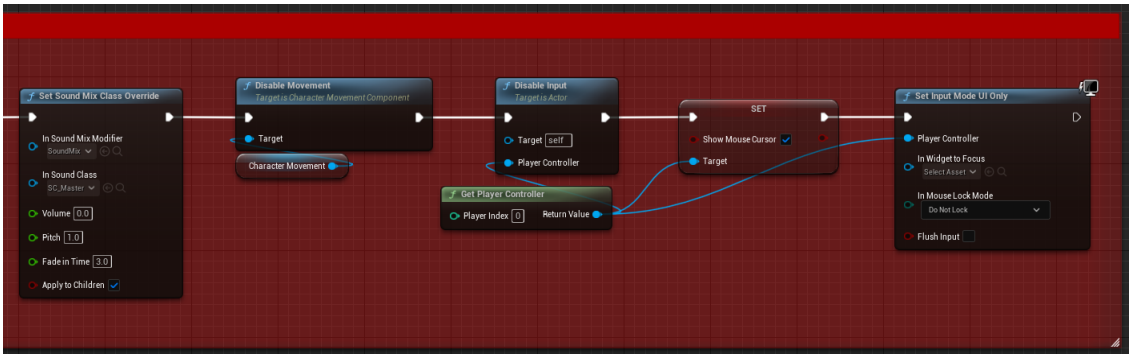
Na horní části obrázku číslo 3.18 je funkce, která odečítá životy, jak bylo řečeno u obrázku posledního. Spodní část obrázku 3.18 a obrázek 3.19 ukazuje logiku smrti. Když zemřeme, tak se zobrazí obrazovka smrti, vypnou se zvuky a přestaneme ovládat naši postavu. V menu potom můžeme hru vypnout nebo začít znova.



Obrázek 3.17:

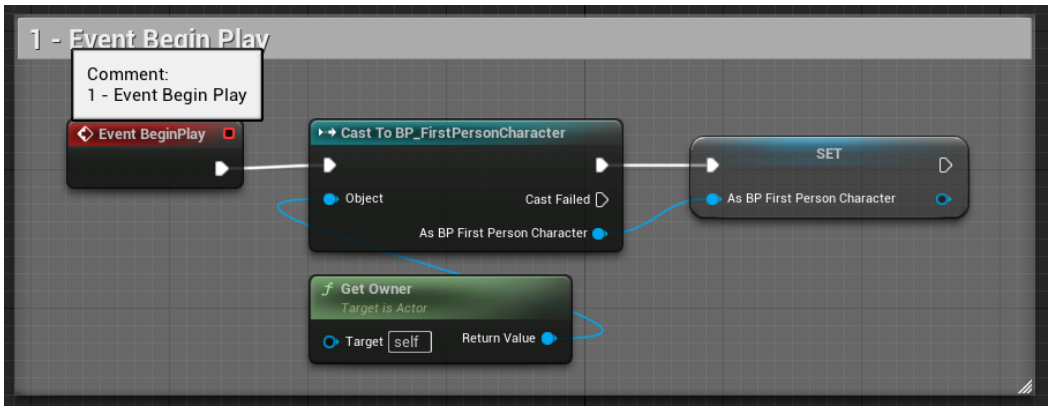


Obrázek 3.18:

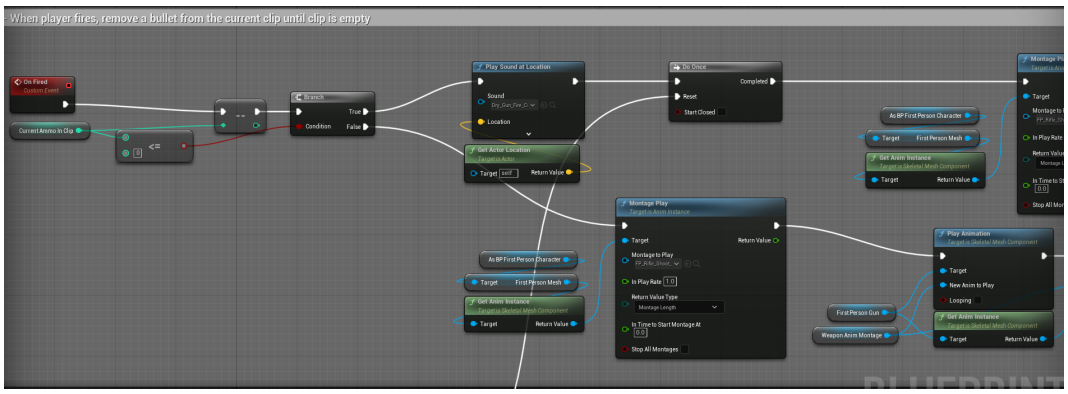

Set Sound Mix Class Override
In Sound Mix Modifier
SoundMix
In Sound Class
SC_Master
Volume 0.0
Pitch 1.0
Fade in Time 3.0
Apply to Children
Disable Movement
Target is Character Movement Component
Target
Character Movement
Disable Input
Target is Actor
Target self
Player Controller
Get Player Controller
Player Index 0
Return Value
SET
Show Mouse Cursor
Target
Set Input Mode UI Only
Player Controller
In Widget to Focus
Select Asset
In Mouse Lock Mode
Do Not Lock
Flush Input

Obrázek 3.19:

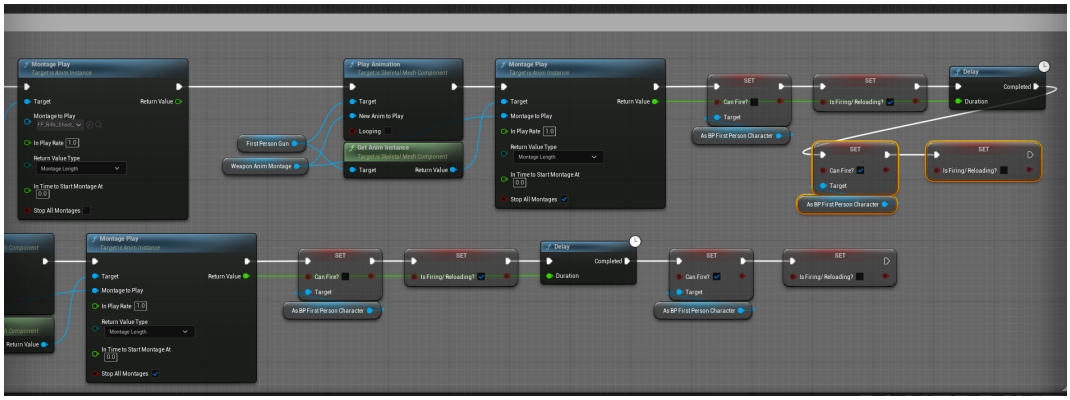
3.1.2 Zbraně a funkce s nimi spojené



Obrázek 3.20:

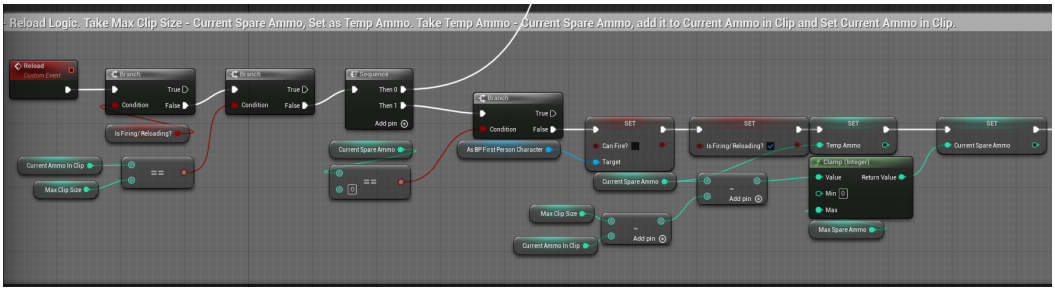


Obrázek 3.21:



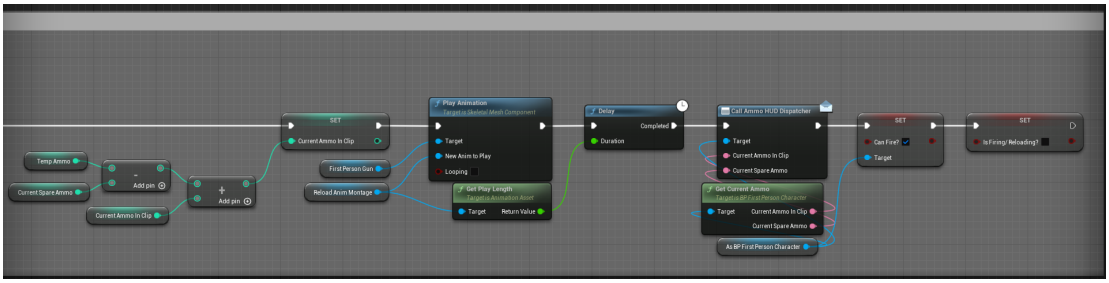
Obrázek 3.22:

Obrázek 3.20 ukazuje uložení do proměnné ukazovač na hráče. Obrázky 3.21 a 3.22 zobrazuje logiku, která se aktivuje pouze po výstřelu. Když se aktivuje, tak odebere jeden náboj ze zbraně a zahraje animace na vystřelení ze zbraně a taky animace na hráči, že vystřelil. Pokud dojdou náboje v zásobníku, tak začne zbraň vydávat zvuk cvakání.

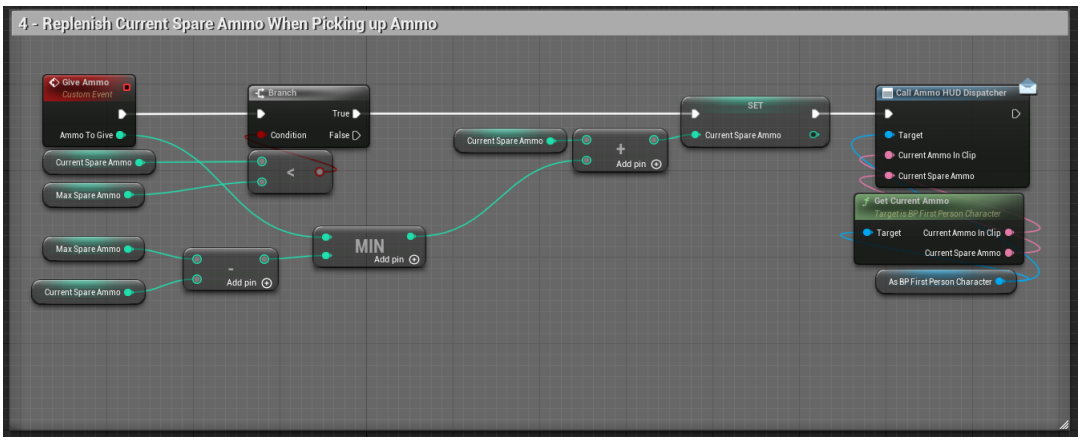


Obrázek 3.23:

Na obrázcích 3.23 a 3.24 je vyobrazena logika přebíjení. Pokud není zásobník plný, tak zjistíme, kolik nábojů chybí a ze zásoby je doplníme do zásobníku a zahrajeme animaci přebíjení.



Obrázek 3.24:

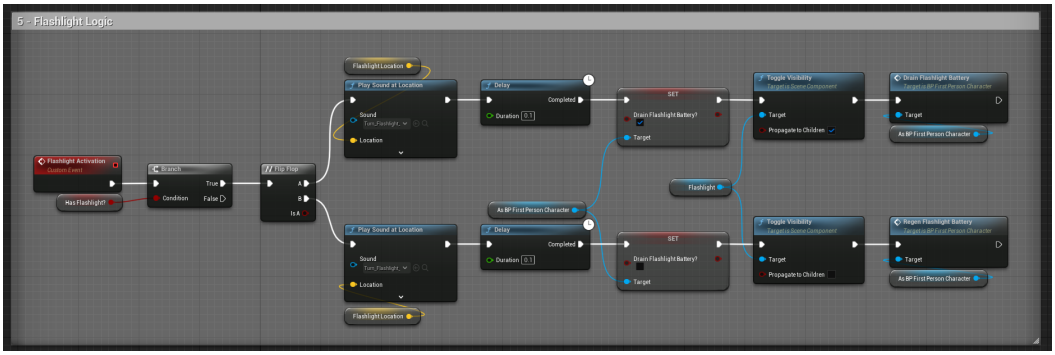


Obrázek 3.25:

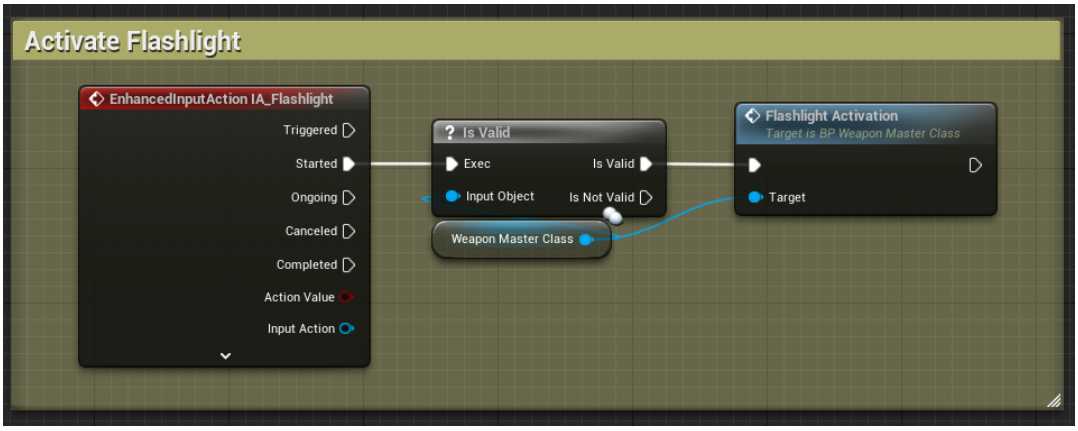
Obrázek 3.25 ukazuje funkci, kde když sebereme náboje ze země, tak zjistíme, jestli nemáme plný počet nábojů, co můžeme nosit a doplníme zásoby. Když jsou zásoby plné, tak nic nedoplníme.

Na obrázku 3.26 je logika svítilny. Pokud zmáčkneme příslušné tlačítko jednou, tak zahrajeme zvuk zapnutí svítily, svítilnu zapneme a začneme odebírat baterku. Když zmáčkneme klávesu znova, tak se vypne svítilna a začnou se baterky dobíjet.

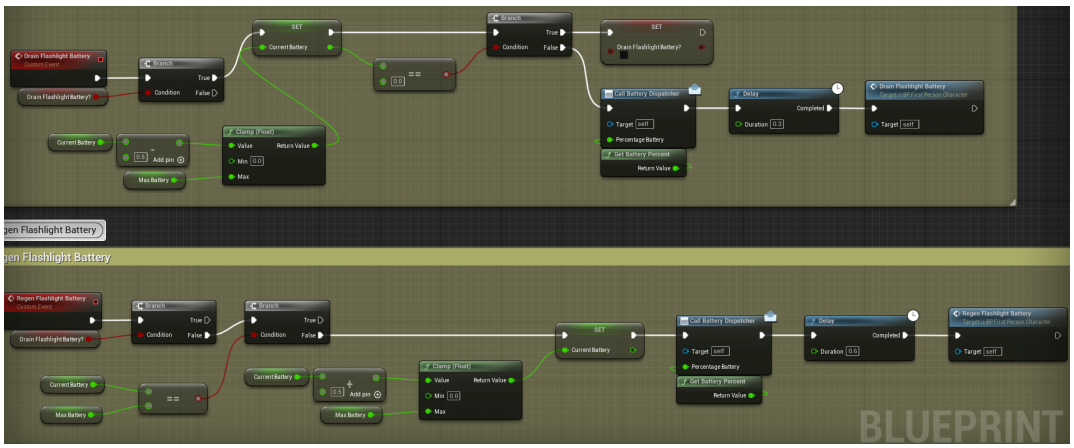
Obrázek 3.27 ukazuje zapnutí svítilny tak, že se zavolá funkce z obrázku 3.26.



Obrázek 3.26:



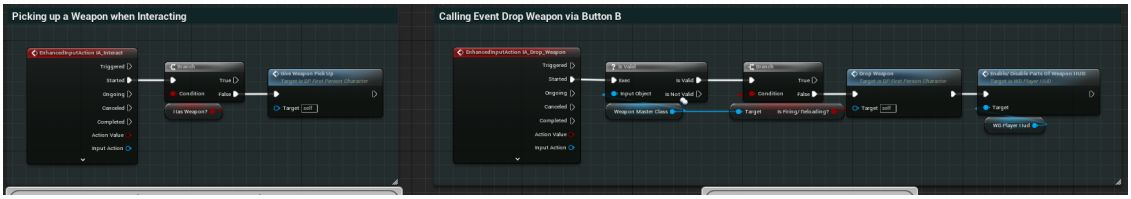
Obrázek 3.27:



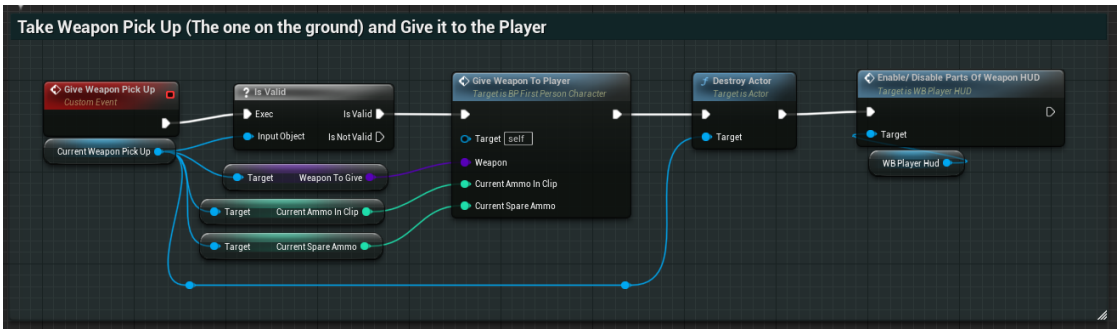
Obrázek 3.28:

V horní části obrázku 3.28 je funkce vybíjení baterky. Funguje na stejném principu jako logika života a energie. Když se zavolá, pomalu začné odebírat z hodnoty baterky za každý tik. Spodní část ukazuje naopak dobíjení baterky. Při zavolání se začne každým tikem pomalu baterka doplňovat.

Levá část obrázku 3.29 ukazuje logiku klávesy E. Při stisku se zavolá funkce z obrázku 3.30. Pravá část obrázku 3.29 naopak ukazuje logiku spojenou s klávesou B. Po zmáčknutí klávesy se zavolá funkce z obrázku 3.32.

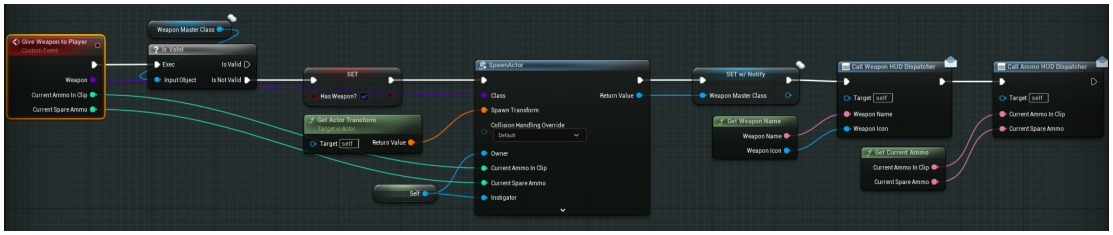


Obrázek 3.29:



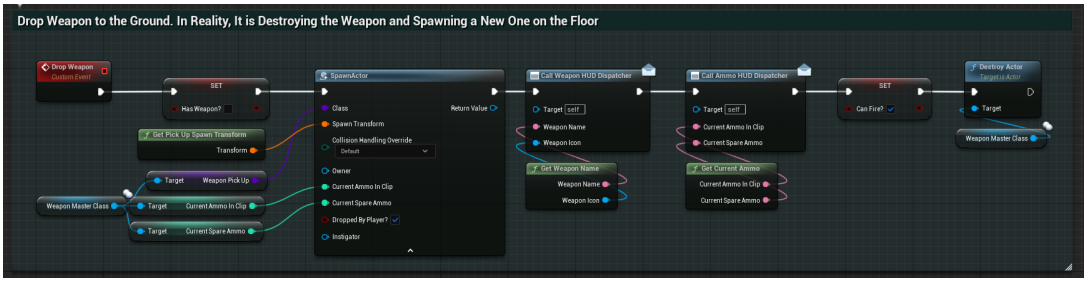
Obrázek 3.30:

Obrázek 3.30 zobrazuje funkci, která když je v okolí hráče zbraň, co lze zvednout a použít, získá z blueprintu na zvedání zbraní informace o typu zbraně (její třídu) a také kolik má zbraň nábojů v zásobníku a v zásobě. Všechny tyto informace předá funkci (obr. 3.31), která hráči dá do rukou a nakonec se zbraň na zemi zničí a zapnou se elementy na displeji.



Obrázek 3.31:

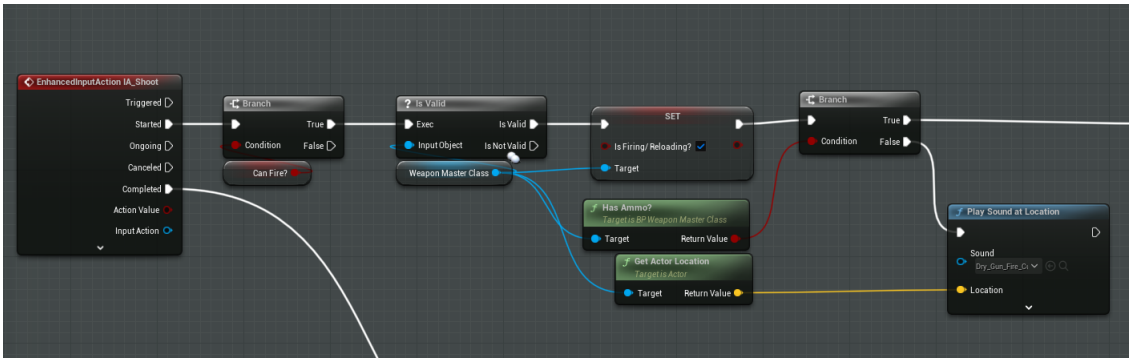
Obrázek 3.31 referencuje logiku, která dá zbraň do rukou hráče. Nastaví se rotace a lokace v rukách a „připne“ se zbraň ke hráči a on ji může začít používat.



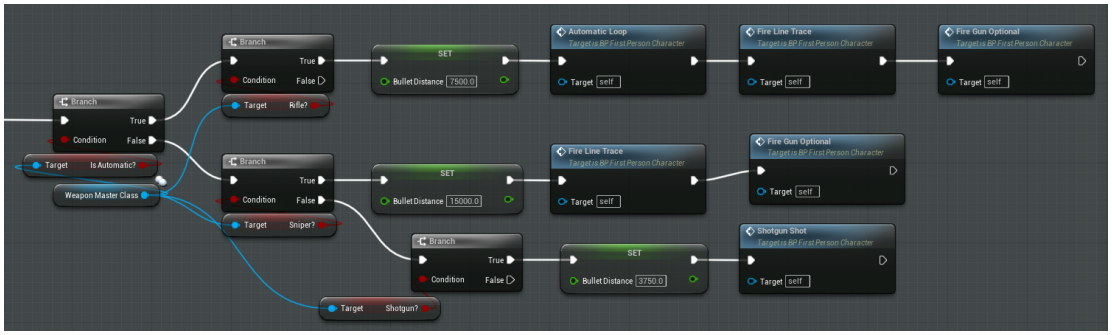
Obrázek 3.32:

Funkce na obrázku 3.32 je pro vyhození zbraně z rukou hráče. Ve skutečnosti se

zbraň v rukou hráče zničí a před hráčem se vytvoří nová zbraň, kterou lze zase zvednout.



Obrázek 3.33:

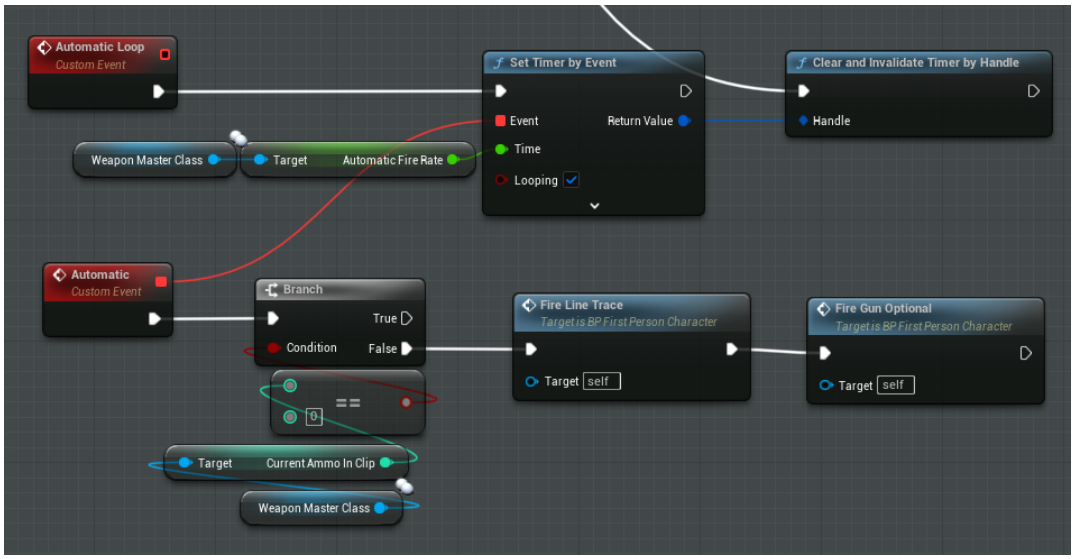


Obrázek 3.34:

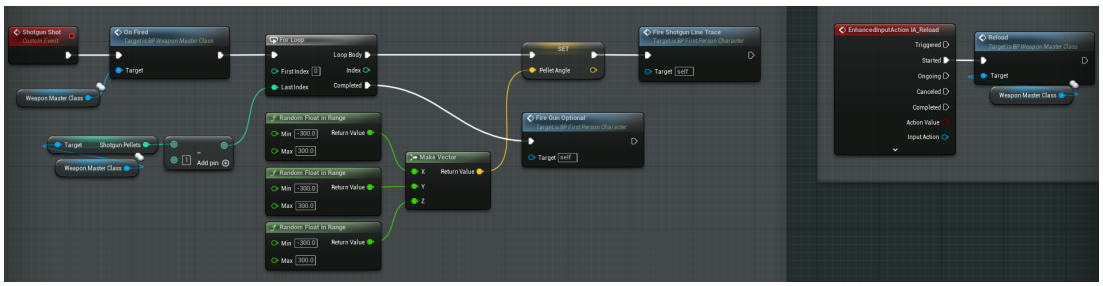
Obrázky 3.33 a 3.34 referencují začátek logiky střílení ze zbraně. Pokud můžeme střílet, tak zjistíme jaký typ zbraně máme v ruce. Pokud máme automatickou pušku, tak se zavolá funkce na obrázku 3.35, která dokud máme náboje v zásobníku bude opakovaně střílet. Když je v rukou brokovnice, tak se zavolá dodatečně funkce z obrázku 3.36, která náhodně bude měnit trajektorii střel, aby se simulovala skutečná brokovnice, a zároveň vystřelí několik střel. Pokud máme odstřelovací pušku, tak se pouze zavolá normalní funkce na střelbu.

Obrázek 3.36 také obsahuje na pravé části logiku tlačítka R, která pouze zavolá funkci z obrázků 3.23 a 3.24.

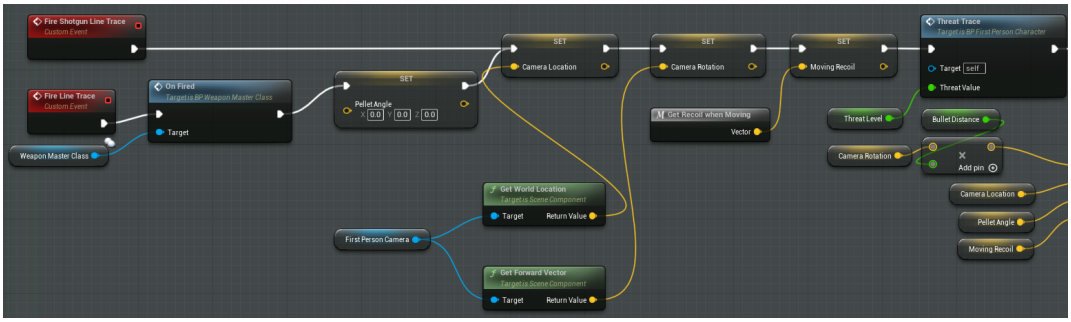
Obrázky 3.37 a 3.38 obsahují hlavní logiku střelby. Nejdříve se zavolá funkce „On Fired“ z obrázků 3.21 a 3.22. Poté co se zjistí rotate a pozice kamery hráče se ze zbraně vyšle přímka, která má šanci „ohrozit“ umělou inteligenci (logika v kapitole Základní logika). Poté se vyšle druhá přímka, která začíná v hráčově kameře a končí v určené vzdálenosti před hráčem, po upravení rotaci kvůli simulaci nepřesnosti při chození a take pokud se jedna o brokovnici tak o nahodny rozptyl střel. Po střelbě simulujeme kopnutí zbraně po výstřelu (upravíme pouze kam se kamera kouká po výstřelu). Nakonec aplikujeme poškození nepřítely, pokud jsme ho trefily a když je někde na mapě věc, se kterou se dá pohnout, tak když se trefíme do dané věci, tak jí dáme akceleraci na opačnou stranu, kde byla trefena.



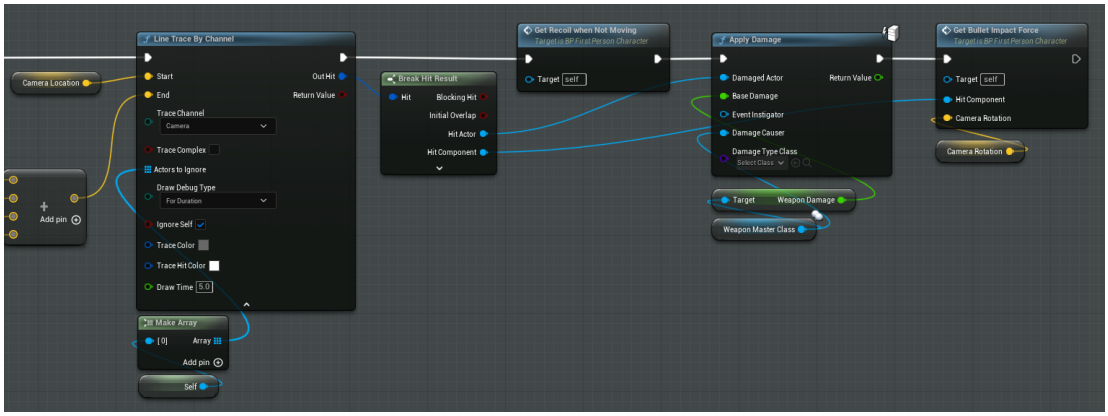
Obrázek 3.35:



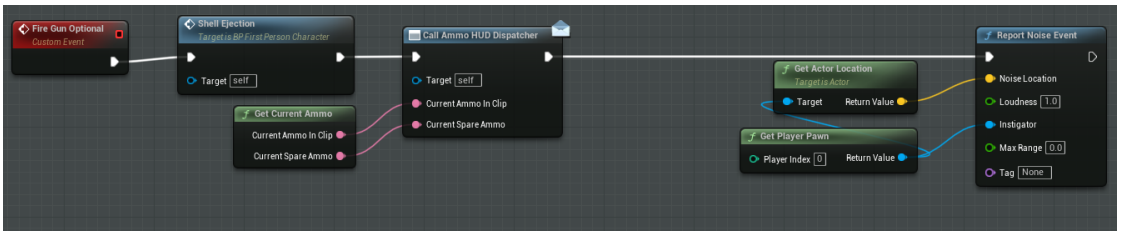
Obrázek 3.36:



Obrázek 3.37:

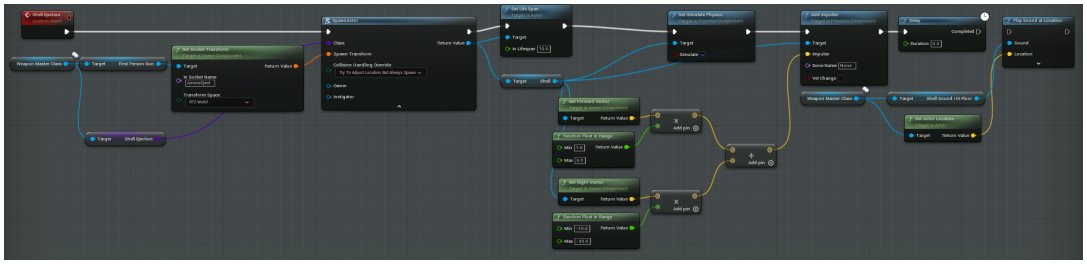


Obrázek 3.38:



Obrázek 3.39:

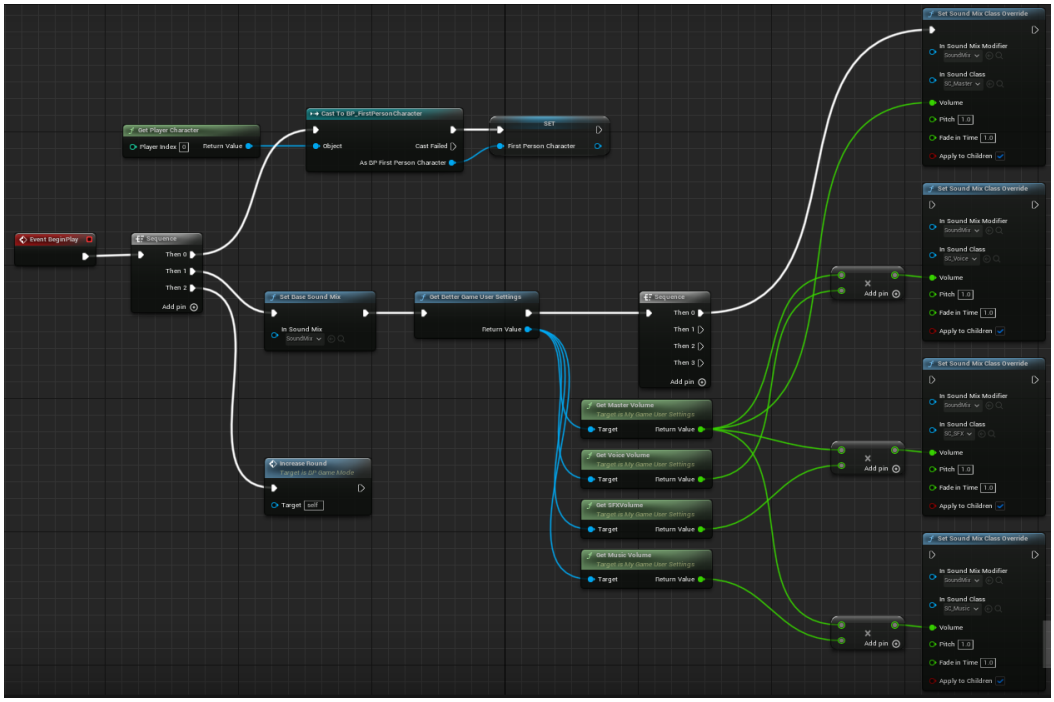
Obrázek 3.39 znázorňuje volání funkce, která se nachází na obrázku 3.40. Také vyvoláme zvukový event, na který dokáže umělá inteligence reagovat



Obrázek 3.40:

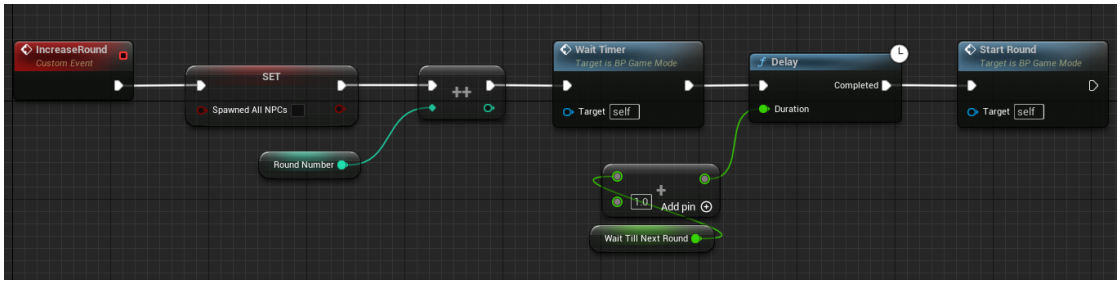
Obrázek 3.40 zobrazuje funkci, která vytváří na straně zbraně použité náboje, které s určitou silou vyhodí ze zbraně a po pár sekundách ze země zmizí.

3.1.3 Herní režim



Obrázek 3.41:

Obrázek 3.41 obsahuje nastavování proměnné, která ukazuje na hráče. Také nastavuje hlasitost, která byla naposledy uložena a nakonec volá funkci na započetí hry (obr.3.42).

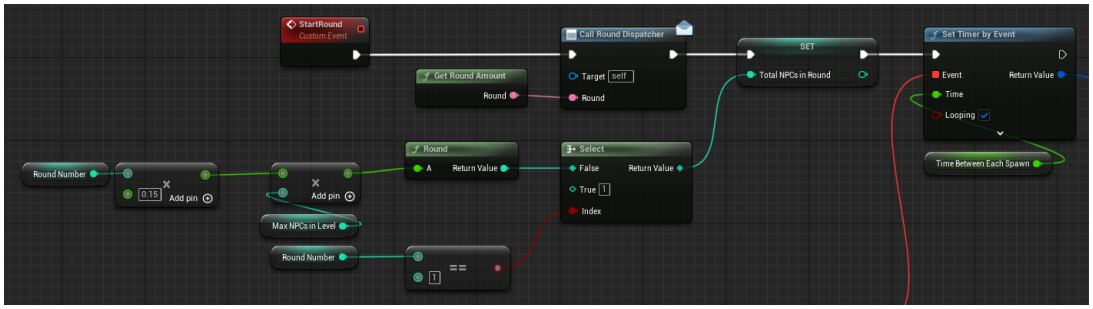


Obrázek 3.42:

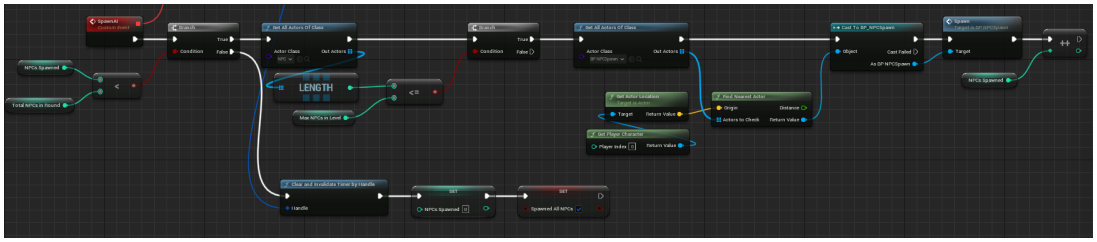
Na obrázku 3.42 je ukázána funkce, která byla zavolána na předchozím obrázku (obr.3.41). Resetujeme hodnotu „Spawned All NPCs“, inkrementujeme číslo kola a započneme časovač k začátku kola.

Obrázky 3.43 a 3.44 obsahují blueprint funkci, která započíná nové kolo. Nejdříve nastavíme počet nepřátel v tomto kole. Když máme nastavený počet, zavoláme časovač, který za daný čas zavolá funkci, která si vyhledá nejbližší bod, kde se může nepřítel objevit a poté na tomto místě nepřítele vytvoří pomocí funkce „Spawn“(obr.3.44).

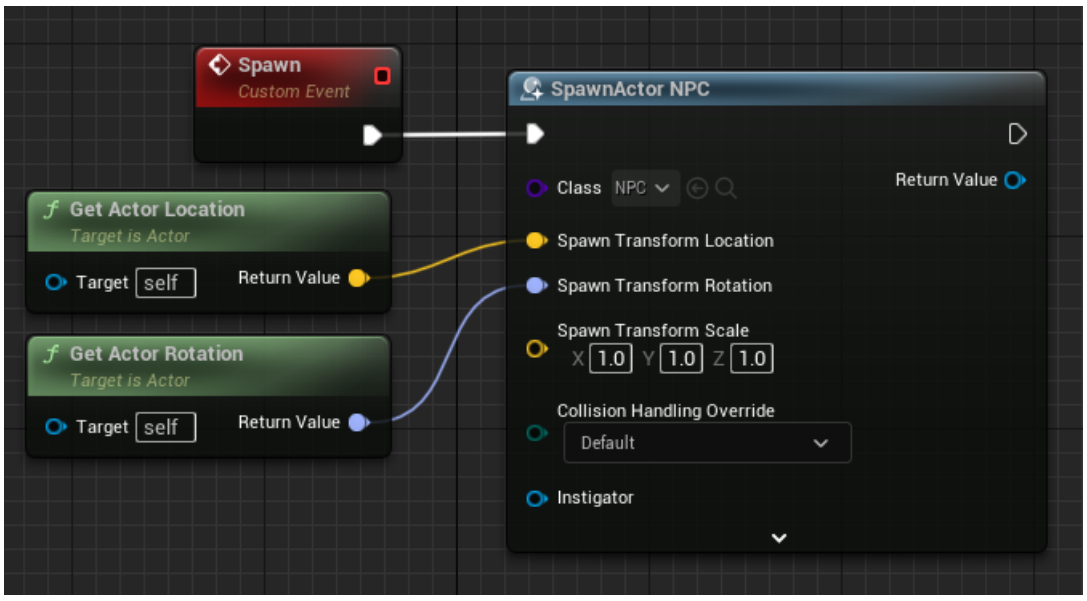
Obrázek 3.45 pouze ukazuje, co už bylo řečeno v posledním paragrafu. Zjistí rotaci a lokaci místa a poté na tomto místě se nepřítel vytvoří.



Obrázek 3.43:



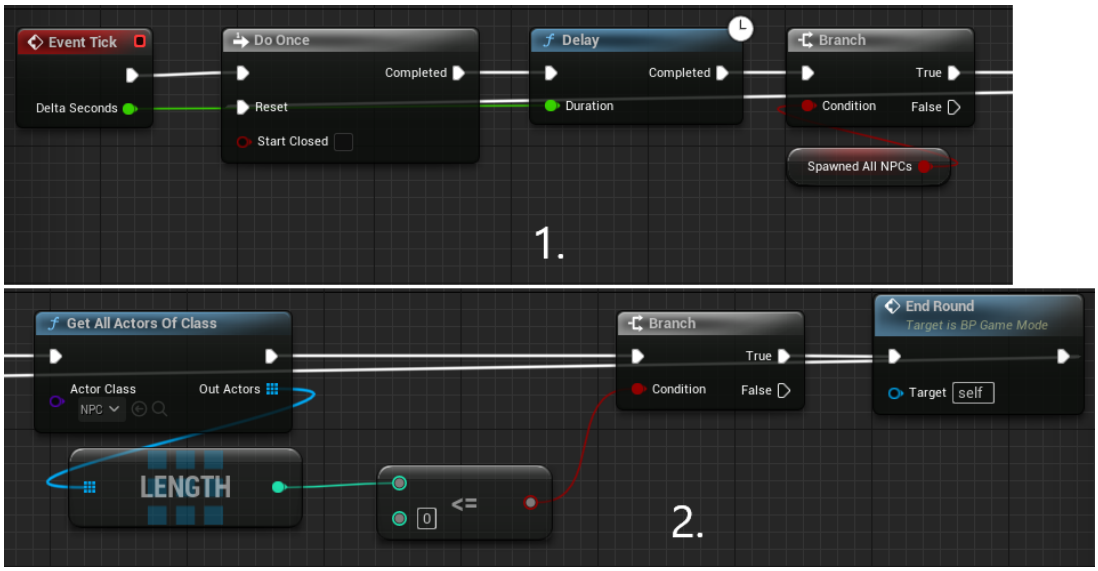
Obrázek 3.44:



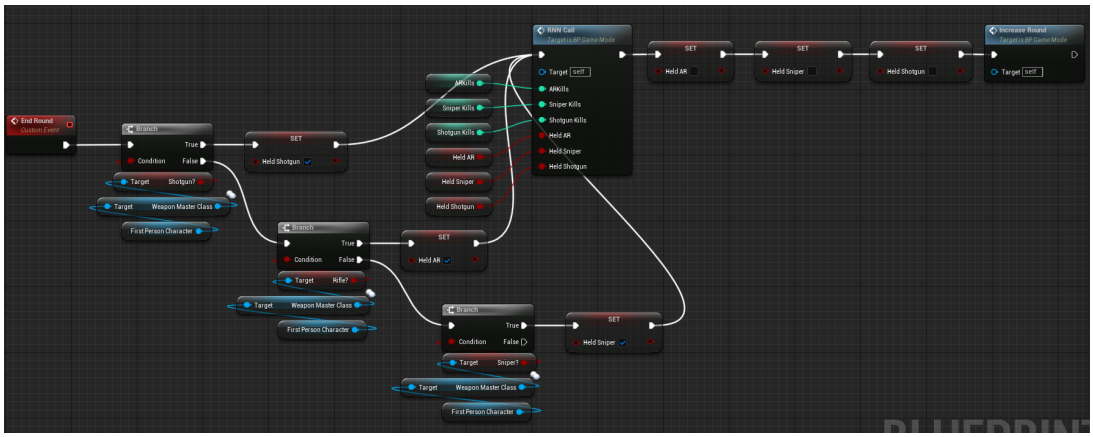
Obrázek 3.45:

Obrázek 3.46 ukazuje neustále se opakující funkci, která zkoumá, zda už všichni nepřátelé v tomto kole už zemřeli. Jestli ano, tak se zavolá konec kola.

Obrázek 3.47 zobrazuje funkci, která končí kolo. Funkce se ptá na zbraň, kterou měl (nebo stále má) hráč v ruce a také kolik je mrtvých pomocí jaké zbraně. Poté se tyto udáje pošlou do funkce „RNN Call“, ve které se volá rekurentní neuronová síť (logika v kapitole Měnění atributů umělé inteligence).

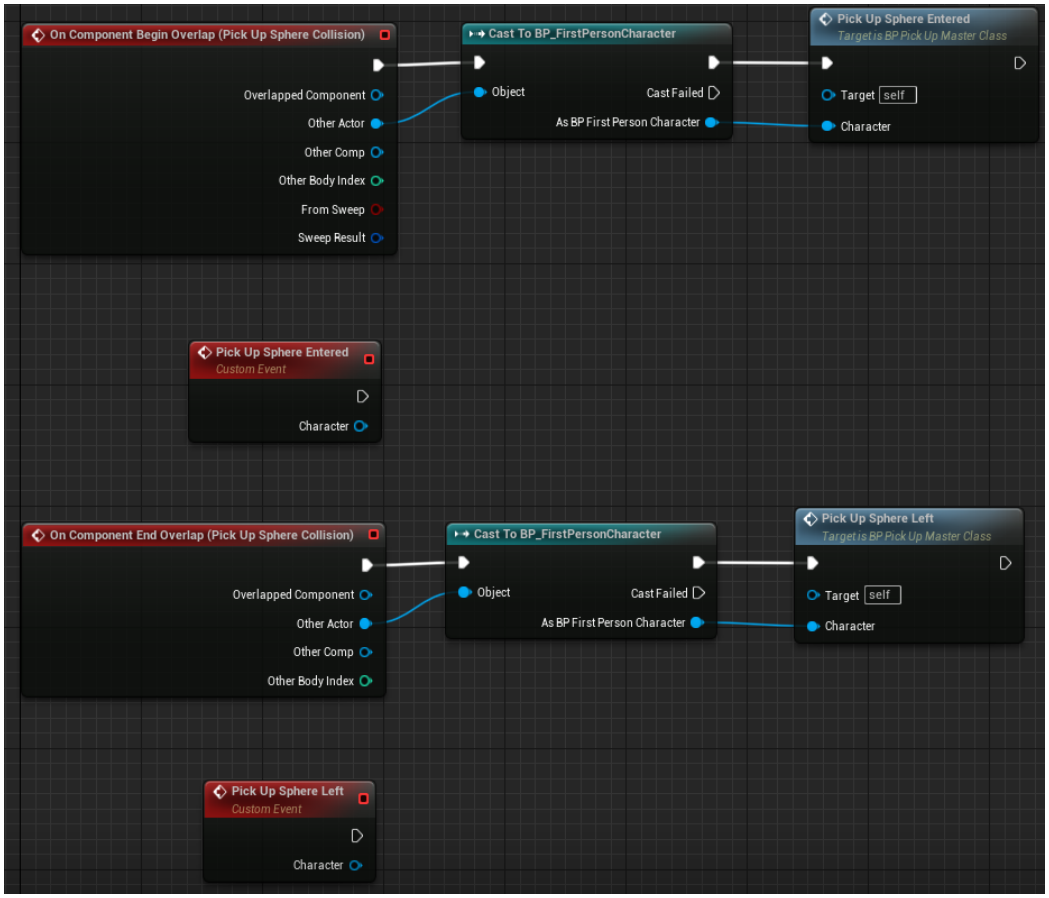


Obrázek 3.46:



Obrázek 3.47:

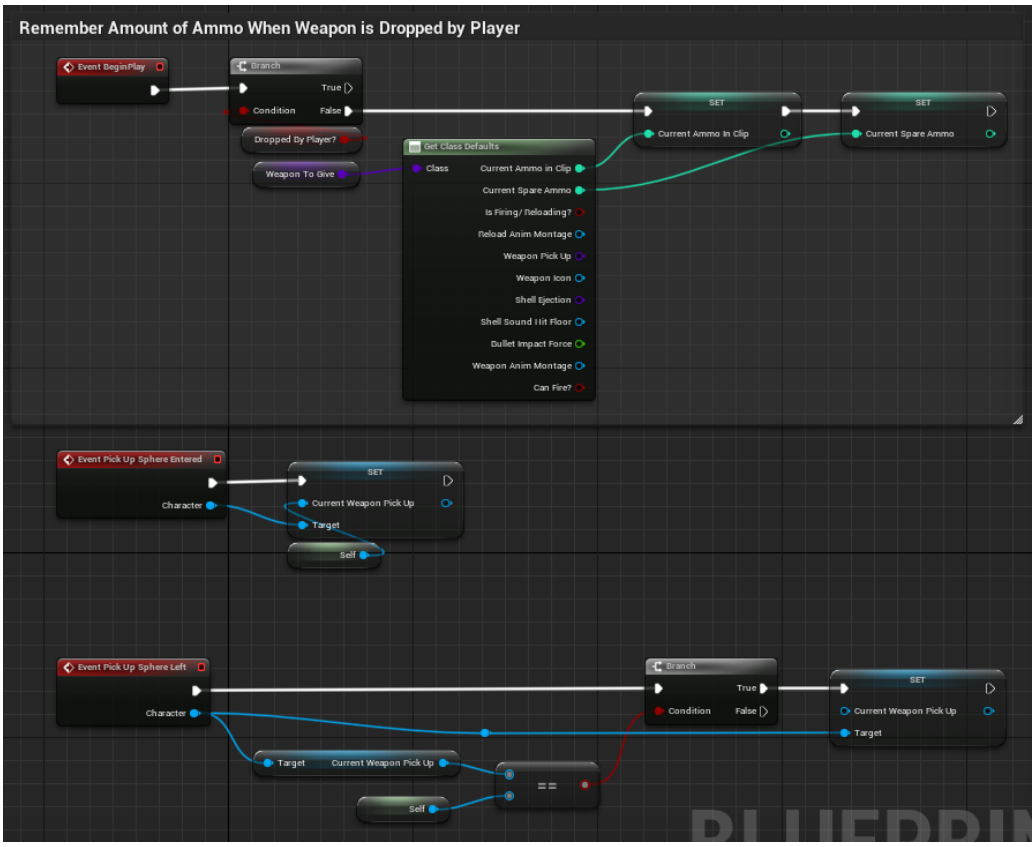
3.1.4 Logika sběru předmětů



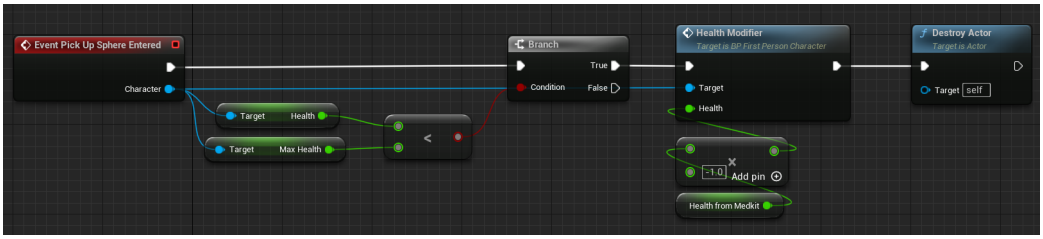
Obrázek 3.48:

Obrázek 3.48 pouze ukazuje základní logiku sběru věcí. Když hráč a sféra dotyku se dotknou, tak se zavolá funkce zvednutí věci. Když se přestane sféra a hráč dotýkat, tak se přestane volat funkce zvednutí.

Tento obrázek s číslem 3.49 obsahuje logiku sbírání zbraní. Tato sféra obsahuje informace, které využívá funkce z obrázku číslo 3.30. Pokud hráč zmáčkne klávesu, pomocí které zbraň zvedne, tak tyto informace sféra předá a tím pádem se tato sféra zničí.

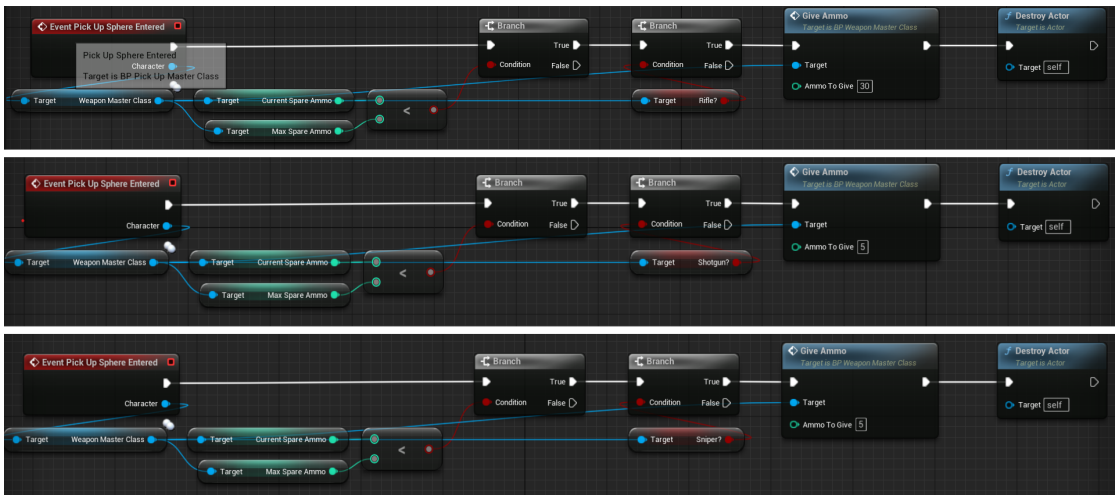

Remember Amount of Ammo When Weapon is Dropped by Player
Event BeginPlay
Branch
True
Condition
False
Dropped By Player?
Weapon To Give
Get Class Defaults
Class
Current Ammo in Clip
Current Spare Ammo
Is Firing / Reloading?
Reload Anim Montage
Weapon Pick Up
Weapon Icon
Shell Ejection
Shell Sound Hit Floor
Bullet Impact Force
Weapon Anim Montage
Can Fire?
SET
Current Ammo in Clip
SET
Current Spare Ammo
Event Pick Up Sphere Entered
Character
SET
Current Weapon Pick Up
Target
Self
Event Pick Up Sphere Left
Character
Branch
True
Condition
False
SET
Current Weapon Pick Up
Target
Target
Current Weapon Pick Up
==
Self

Obrázek 3.49:



Obrázek 3.50:

Obrázek s číslem 3.50 obsahuje jednoduchou logiku, kde když má hráč méně než 100% života, tak mu lékarna zvedne životy a vzápětí se zničí.

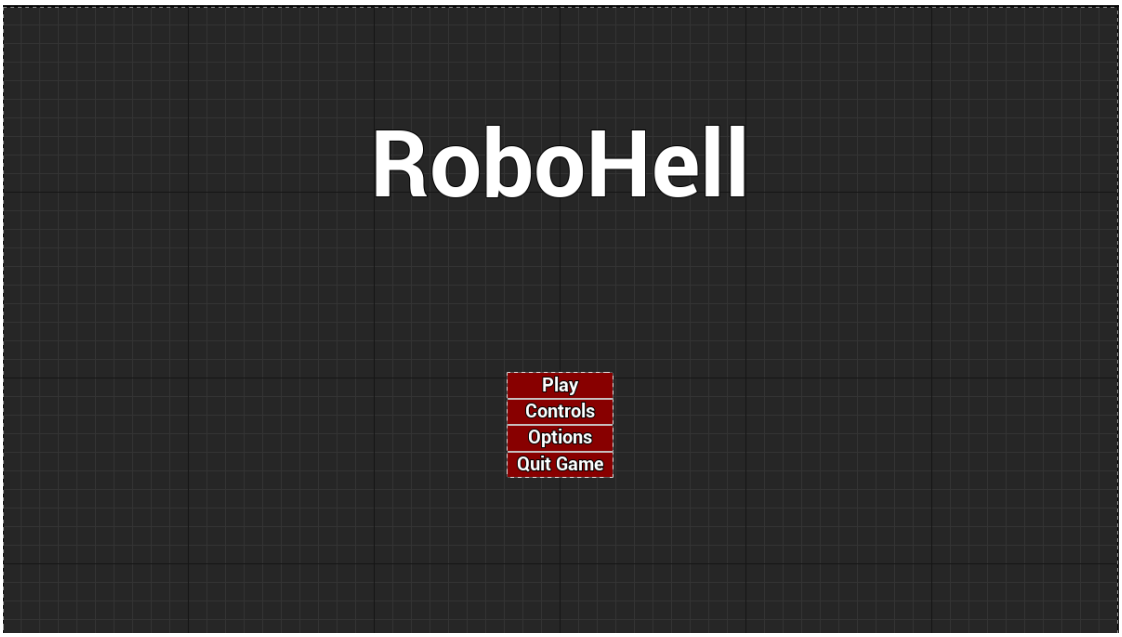


Obrázek 3.51:

Obrázek obsahuje téměř identickou logiku, která přidává náboje buď automatické pušce, brokovnici nebo odstřelovací pušce. Pokud má hráč nedostatek místa pro tyto náboje, tak zůstanou nadále ležet na zemi.

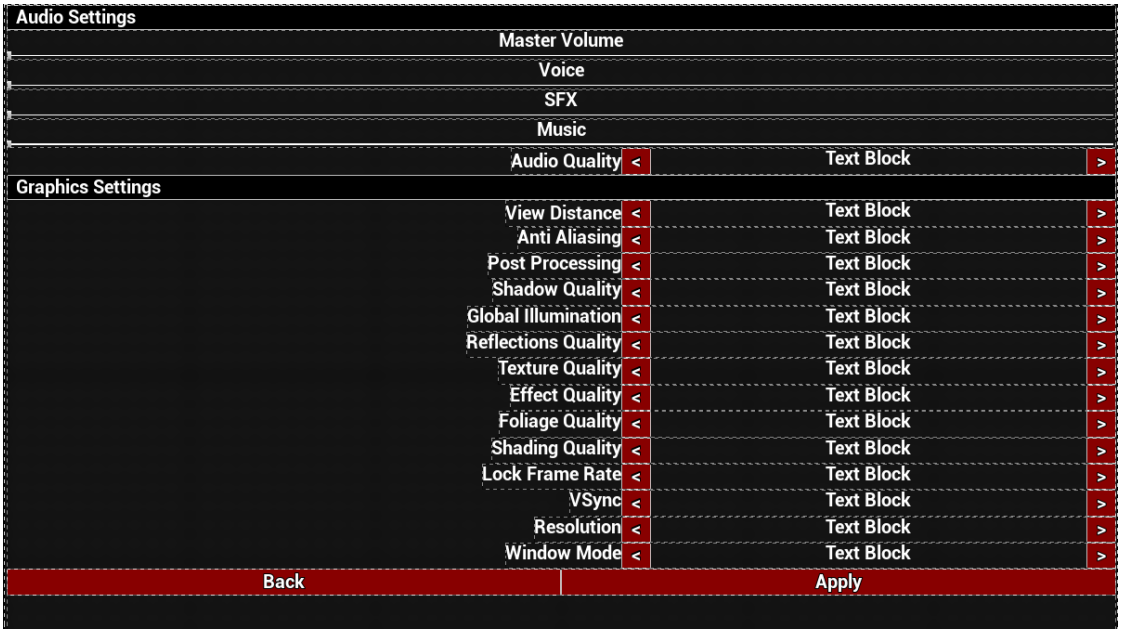
3.1.5 HUD (Heads-up display) - „průhledový“ displej

Obrázek 3.52 ukazuje hlavní menu hry, kde při stisku „Play“ se zapne hra. Když se stiskne tlačítko „Controls“, tak vás zavede do menu, kde máte vyobrazené všechny klávesy potřebné ke hraní hry (obr.3.53). Stisknutím tlačítka „Back“ se vrátíte zpět do hlavního menu. Tlačítko „Options“ vás zavede do menu (obr.3.54), kde si můžete nastavit hlasitost hry a zároveň kvalitu audia, ale také si můžete vizualně nastavit hru dle vlastních potřeb, aby hra běžela plynuleji. Stisknutím „Apply“ tyto změny aplikujete, ale stisknutím tlačítka „Back“ je zrušíte. A nakonec tlačítko „Quit Game“ hru vypne.

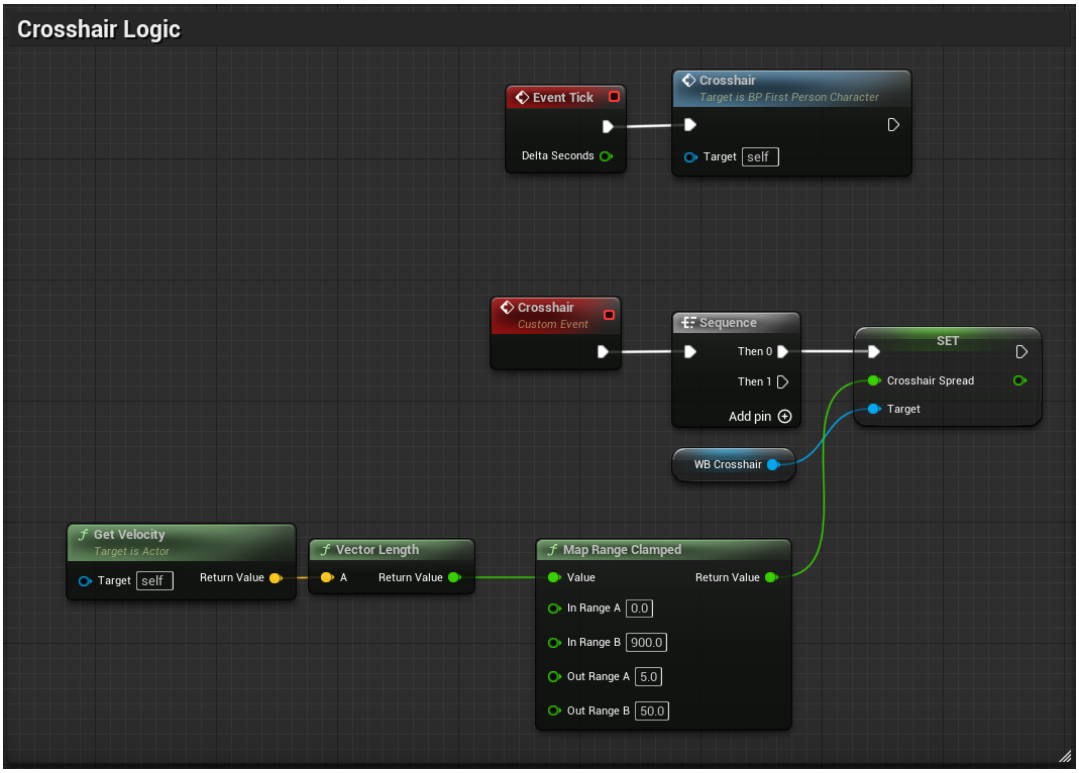


Obrázek 3.52:

Obrázek 3.53:

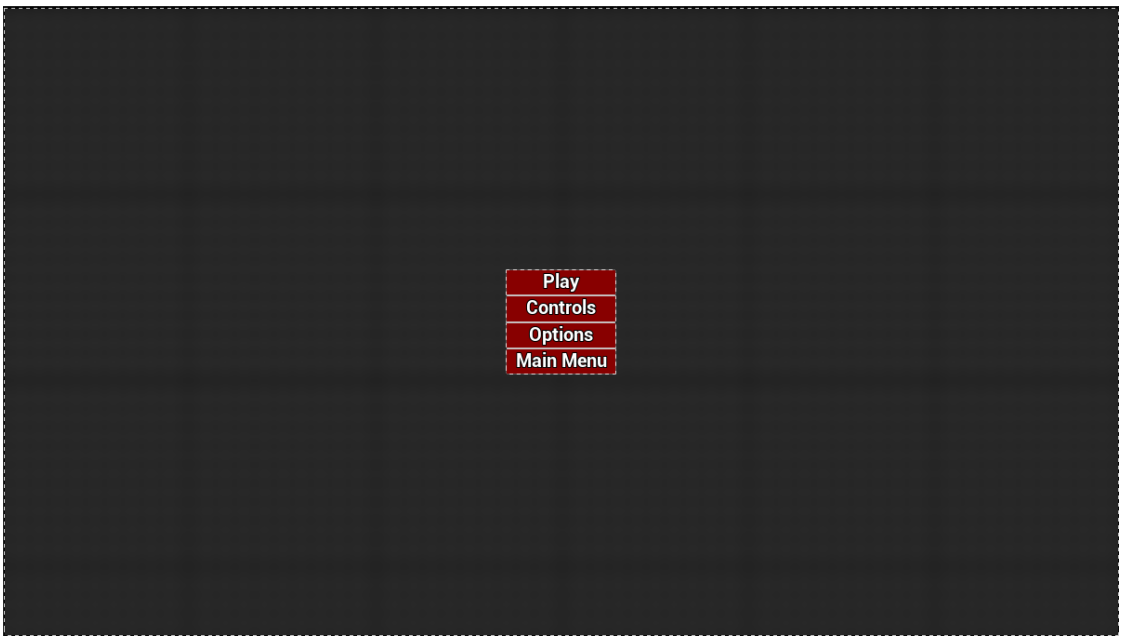


Obrázek 3.54:



Obrázek 3.55:

Obrázek 3.55 zobrazuje volání funkce křížku, kde podle rychlosti hráče poté nastavujeme velikost křížku na displeji. Křížek na displeji nám ukazuje střed displeje, abychom byli přesnější.



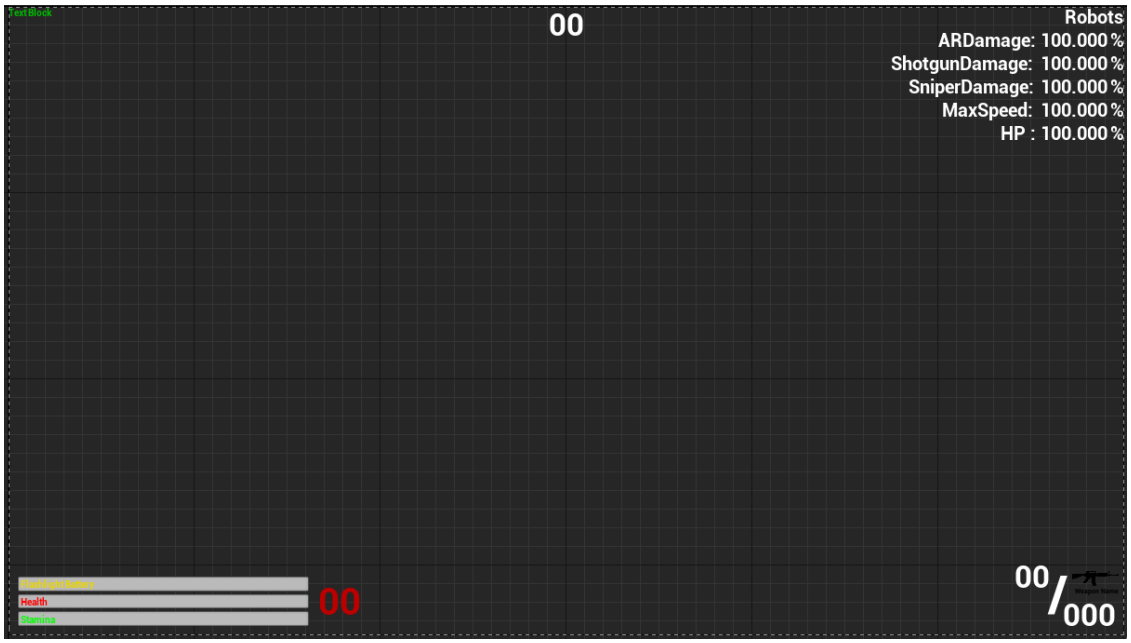
Obrázek 3.56:

Obrázek 3.56 ukazuje vzhled menu pauzy, které se da zapnout při hrání hry po stisknutí klávesy P. Tlačítka „Options“ a „Controls“ jsou identické jako ve hlavním

menu. V tomto případě tlačítko „Play“ zruší pauzu a lze pokračovat ve hře. Po stisknutí tlačítka „Main Menu“ se vrátíte zpět do hlavního menu.

Obrázek 3.57:

Obrázek 3.57 ukazuje menu, které se objeví na obrazovce po smrti. Tlačítko „Restart“ znovu zapne hru, „Quit to Main Menu“ vás vezme zpět do hlavního menu a „Quit“ vypne hru.



Obrázek 3.58:

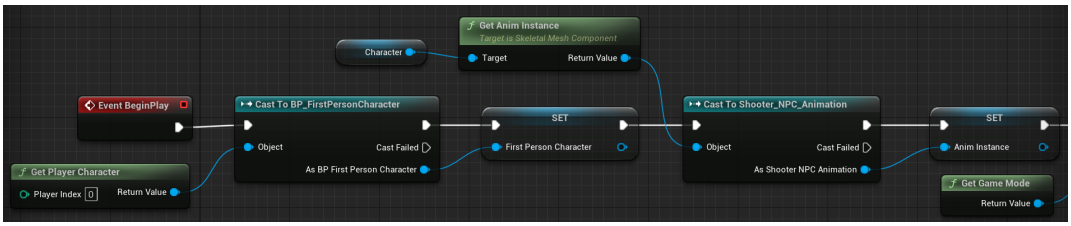
Obrázek 3.58 ukazuje HUD hráče. V horním rohu je počítadlo snímků za sekundu. Uprostřed displeje v horní části je časovač, který se odpočítává pokaždé před začátkem nového kola. Pravý roh obsahuje ukazovač atributů, který zobrazuje změněné

atributy nepřátel po každém zavolání neuronové sítě. Ve spodním levém rohu máme tři lišty. Horní lišta zobrazuje procentuální nabití baterky, která se nachází na zbrani. Prostřední lišta ukazuje život a spodní ukazuje energii. Vedle lišt se nachází počítačka kol. V pravém spodním rohu se nachází počítačka nábojů v zásobníku a také náhradních nábojů. Zároveň se v tomto rohu ukazuje jméno a typ zbraně, kterou v tu chvíli má hráč v rukou.

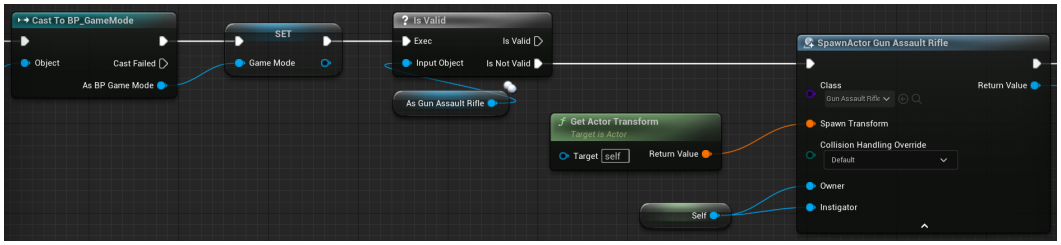
3.2 Umělá inteligence (autonomní agent)

3.2.1 Základní logika

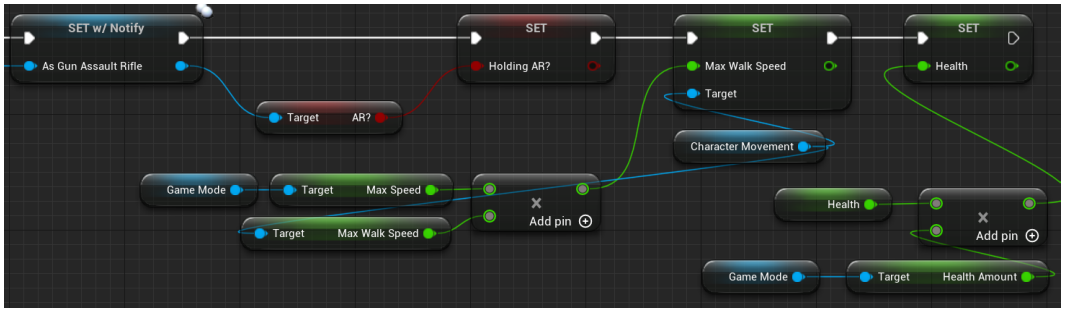
Základní funkce



Obrázek 3.59:



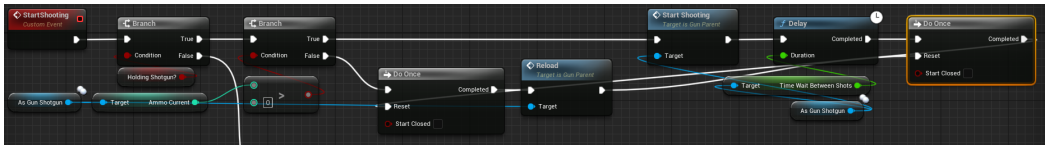
Obrázek 3.60:



Obrázek 3.61:

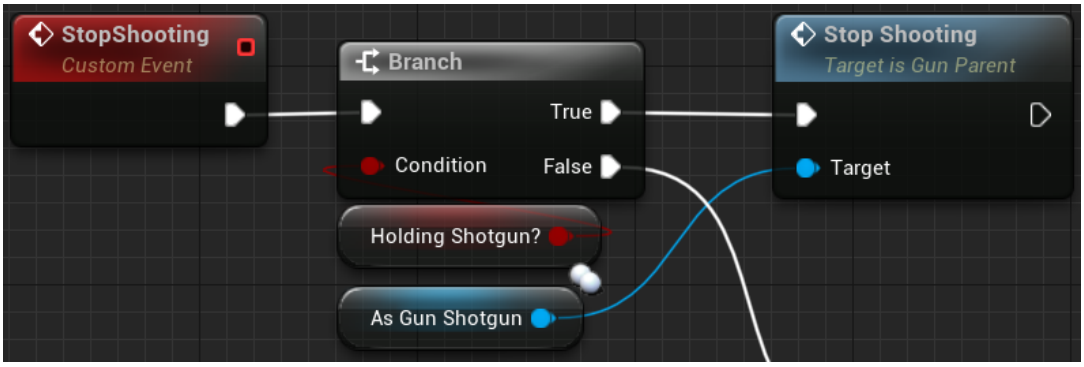
Obrázky 3.59, 3.60 a 3.61 zobrazují inicializaci postavy umělé inteligence. Na prvním obrázku lze vidět získání a následné uložení do proměnné ukazatele na hráče. Druhý ukazatel je pro animace AI postavy, který je také uložen do proměnné a poslední ukazatel, který získáme a uložíme do proměnné je pro herní mód této hry(pro měnění atributů). Dále se ptáme, zdali AI postava má v rukou útočnou pušku, pokud ne, tak tuto zbraň vložíme do rukou AI postavy. Nakonec nastavíme proměnnou, že drží postava v rukou útočnou pušku a pomocí atributů (více v kapitole Měnění atributů umělé inteligence) nastavíme maximální rychlost postavy a také počet životů.

Obrázek číslo 3.62 ukazuje jednu ze tří větví, které jsou skoro totožné (každá větev je pro jiný typ zbraně, kterou postava může používat). Nejdříve se zeptáme, jaký



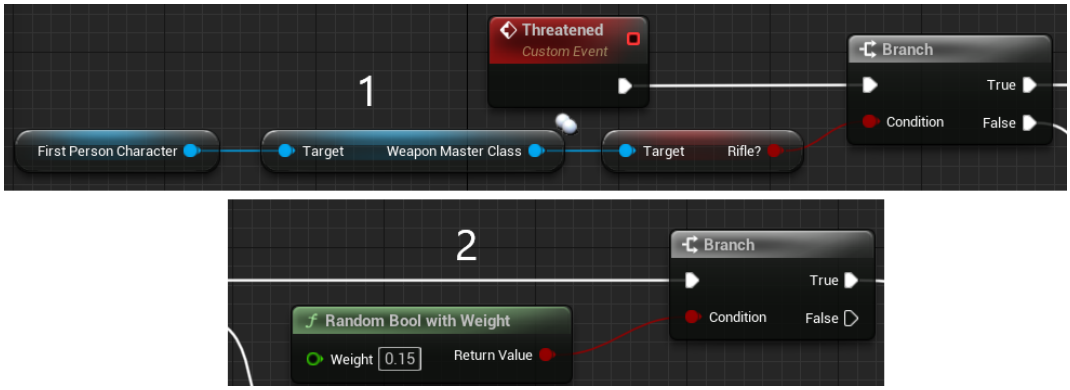
Obrázek 3.62:

typ zbraně postava má v rukou a poté se zeptáme, zdali má v zásobníku dost nábojů. Pokud nemá, zavolá se funkce pro přebíjení zbraně. Pokud má zbraň dostatek nábojů, začne umělá inteligence střílet s časovou prodlevou.



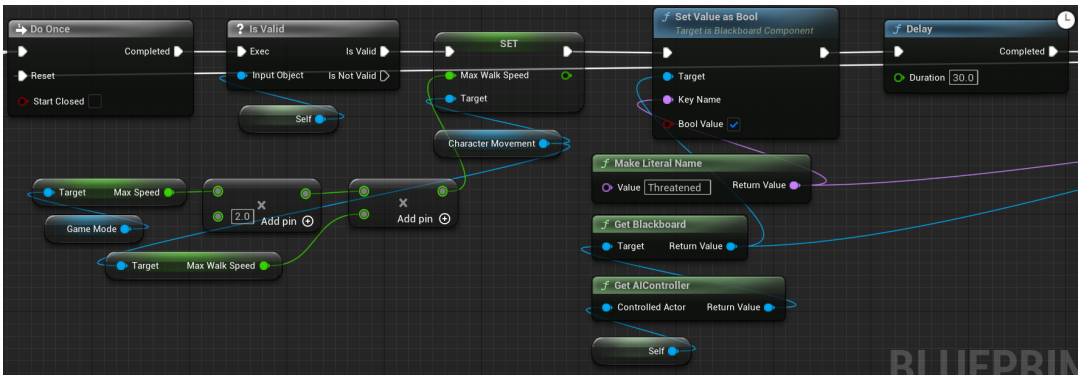
Obrázek 3.63:

Obrázek 3.63 obsahuje kratičkou funkci o třech větvích (pro každou zbraň), kde se zavolá ukončení střelby pro umělou inteligenci.

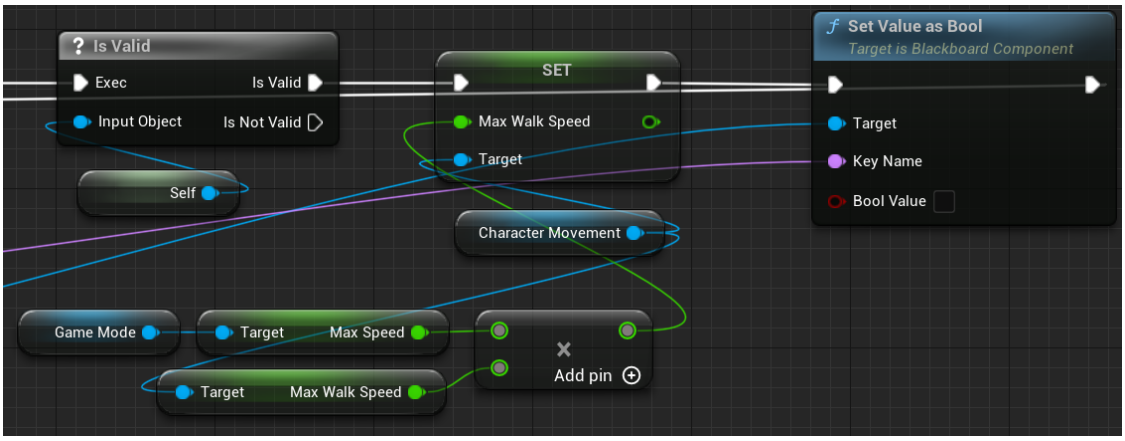


Obrázek 3.64:

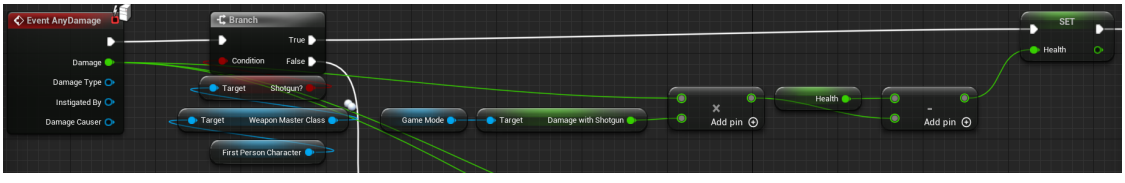
Obrázky 3.64, 3.65 a 3.66 obsahují funkci ohrožení AI postavy, První obrázek popisuje jednu ze tří větví (pro všechny zbraně), kde se ptáme, jakou zbraň má pravě v rukou hráč. Podle typu zbraně určíme šanci ohrožení AI postavy. Pokud se AI postava bude cítit ohroženě, začne běhat a nacházet si úkryt (logika v podkapitole Strom chování) po určitou dobu. Pokud AI postava do uplynutí času nebude zničena, nastaví se normální rychlost chození a AI postava se přestane cítit ohroženě.



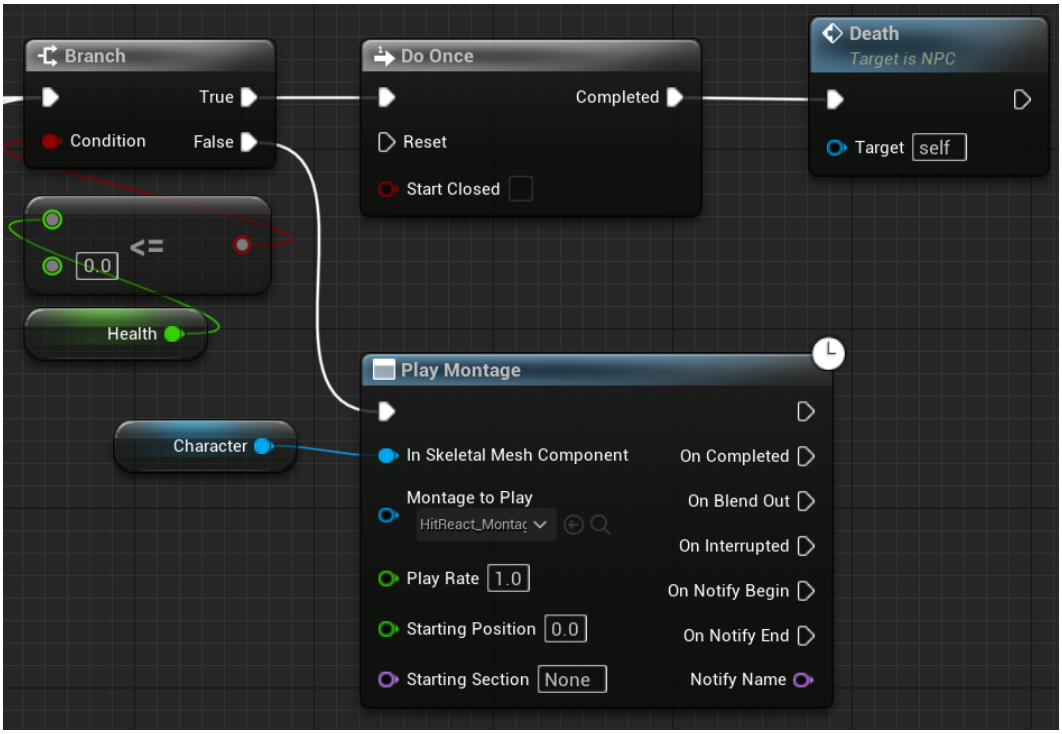
Obrázek 3.65:



Obrázek 3.66:

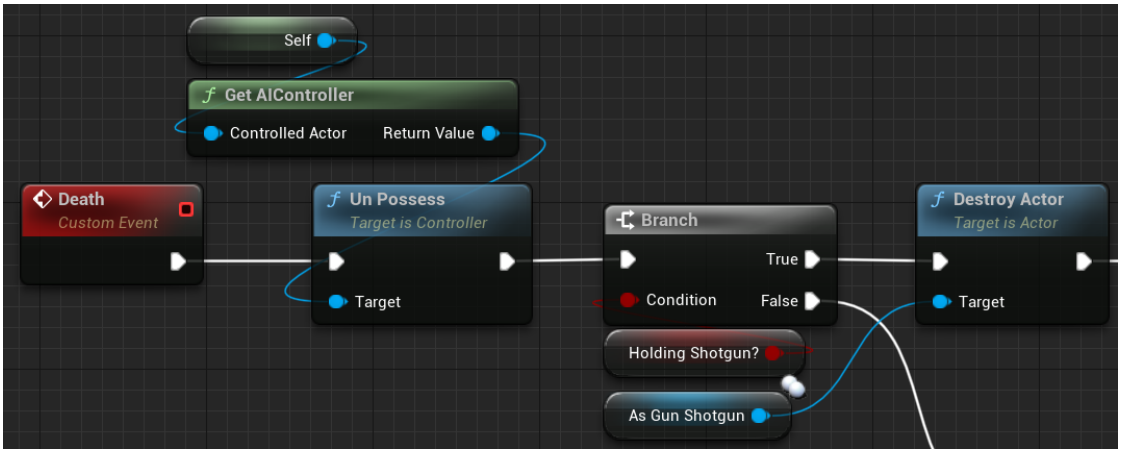


Obrázek 3.67:



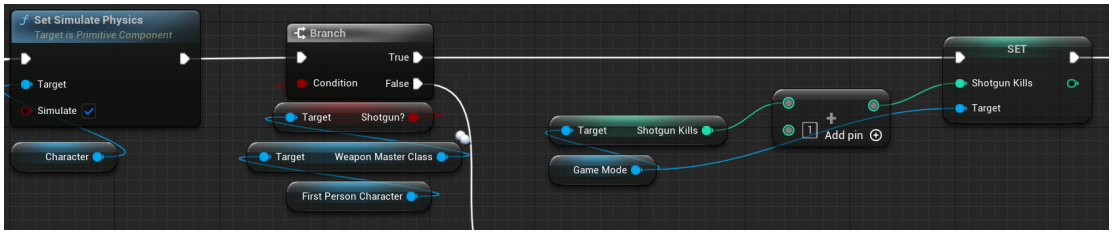
Obrázek 3.68:

Obrázky 3.67 a 3.68 obsahují funkci na poškozování AI postavy. Začátek opět má tři větve, které jsou si velice podobné, jenom se liší voláním jiné zbraně. Získáme obdržené poškození, které upravíme pomocí atributů (více v sekci Měnění atributů umělé inteligence) a následně aplikujeme do AI postavy. Pokud se život AI postavy dostane na nulu, zavoláme funkci smrt.

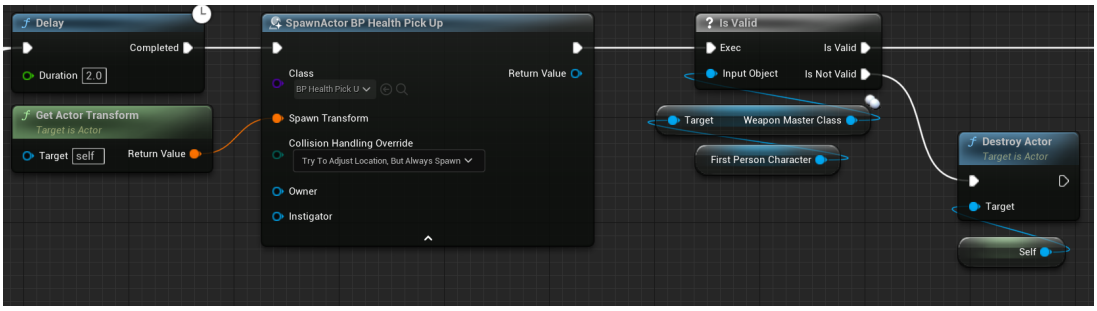


Obrázek 3.69:

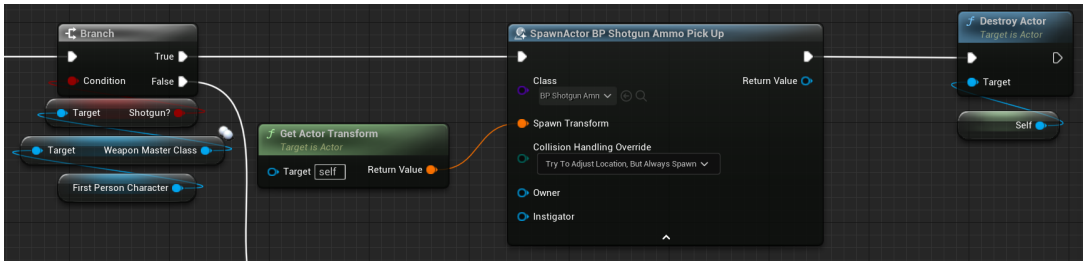
Obrázky s čísly 3.69, 3.70, 3.71 a 3.72 obsahují funkci smrt. Nejprve z AI postavy odstraníme umělou inteligenci. Po odstanění inteligence zjistíme, co má za zbraň postava v rukou a vzápětí ji zničíme. Na postavě začneme simulovat fyziku, aby postava spadla na zem. Po spadnutí se zjistí, jakou zbraní hráč tuto postavu zabil a do proměnné se přičte jednička. Po chvíli čekání spadne z postavy lékárnička a také



Obrázek 3.70:



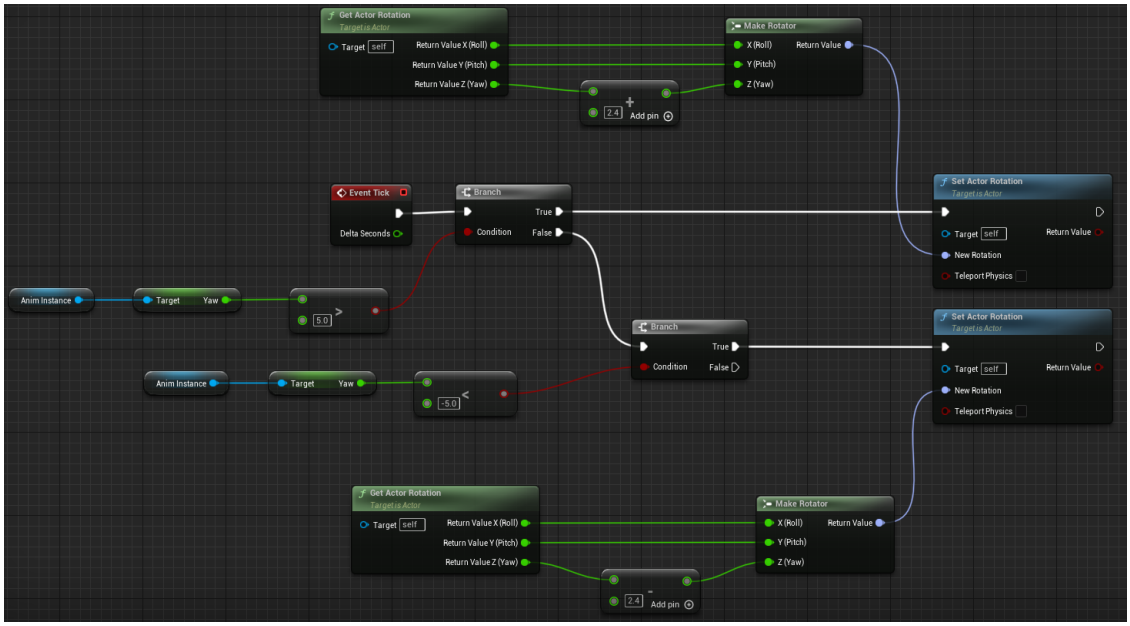
Obrázek 3.71:



Obrázek 3.72:

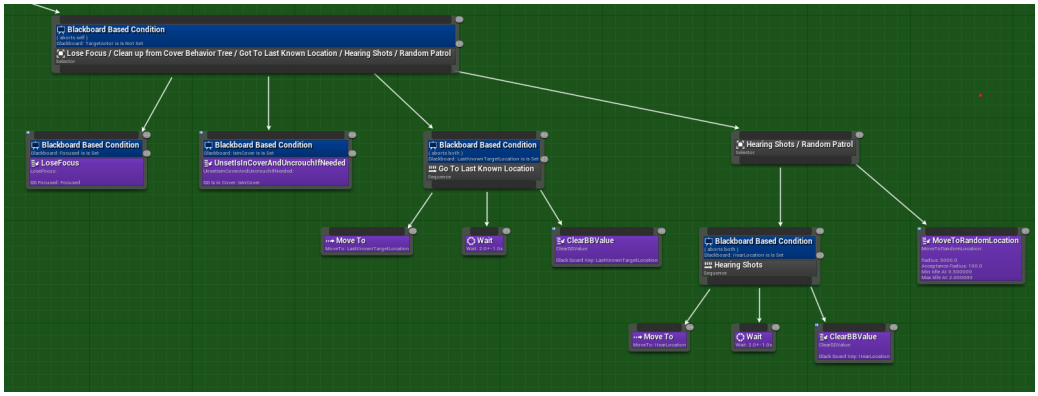
náboje podle typu zbraně, co hráč má v rukou, pokud žádnou nemá, nespdane nic. Nakonec celou postavu odstraníme.

Obrázek 3.73 ukazuje funkci, která otáčí postavu umělé inteligence, pokud je úhel dostatečně vysoký. Poté aplikujeme příslušnou rotaci postavy, aby se koukala na hráče.

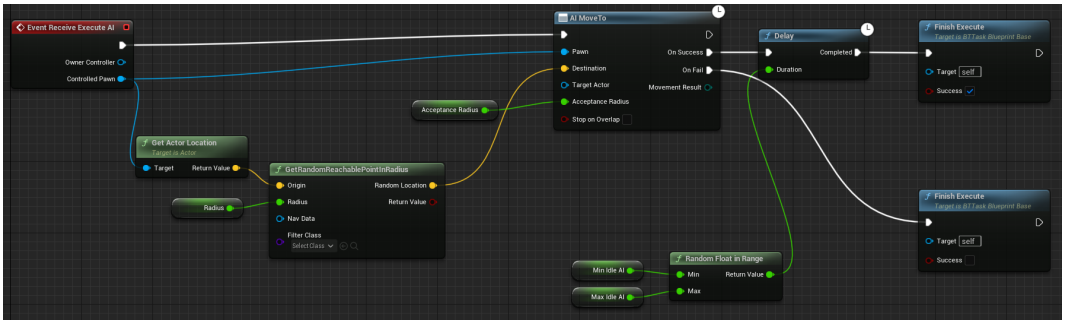

Get Actor Rotation
Target is Actor
Target self
Return Value X (Roll)
Return Value Y (Pitch)
Return Value Z (Yaw)
Make Rotator
X (Roll)
Y (Pitch)
Z (Yaw)
Return Value
2.4
Add pin
Event Tick
Delta Seconds
Branch
Condition
True
False
Set Actor Rotation
Target is Actor
Target self
New Rotation
Teleport Physics
Return Value
Anim Instance
Target
Yaw
5.0
-5.0

Obrázek 3.73:

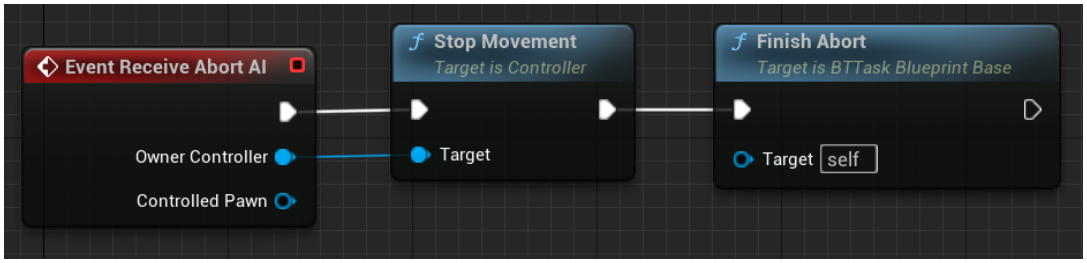
Strom chování



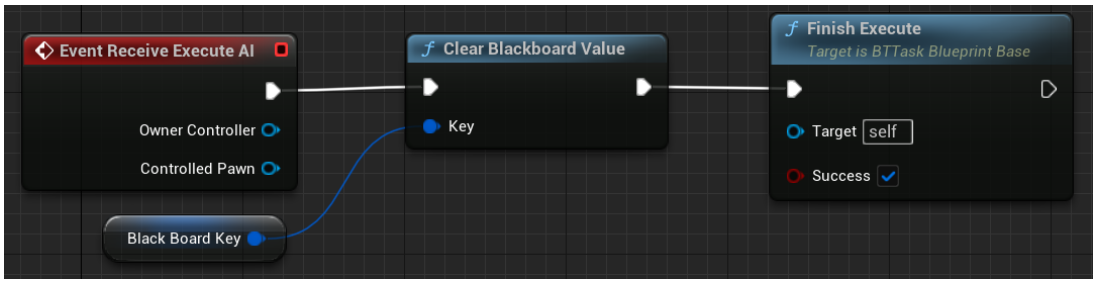
Obrázek 3.74:



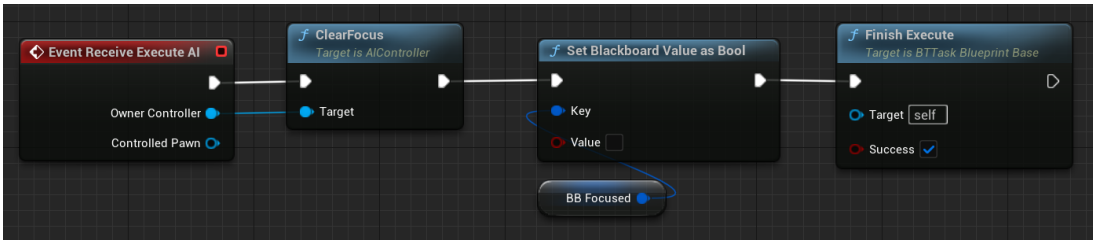
Obrázek 3.75:



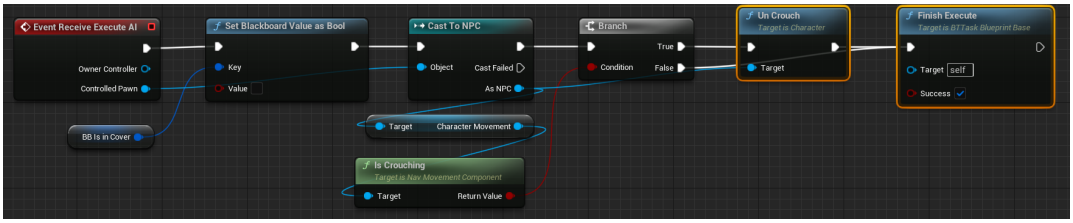
Obrázek 3.76:



Obrázek 3.77:

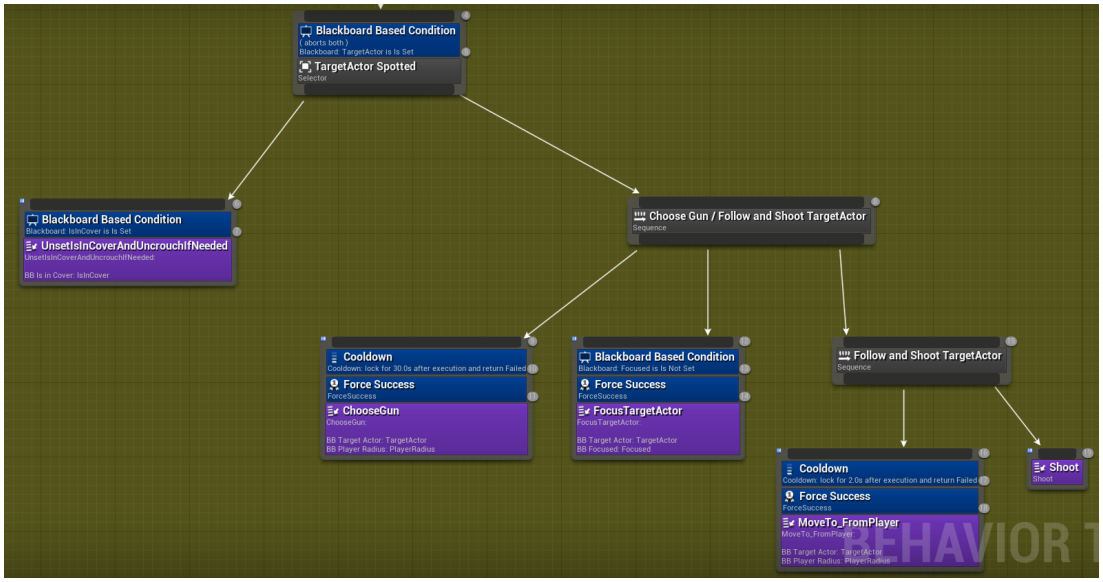


Obrázek 3.78:

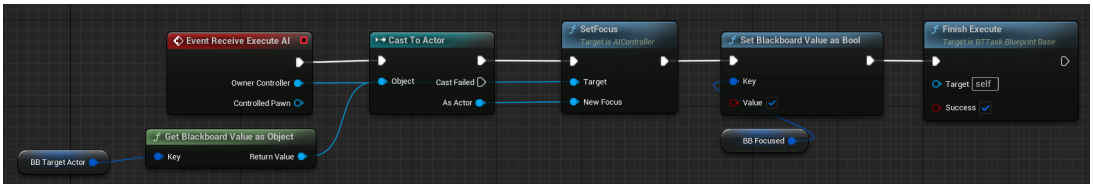


Obrázek 3.79:

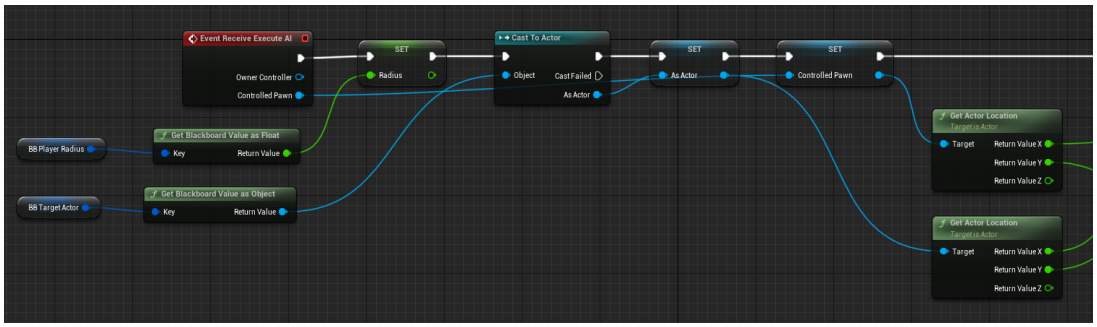
Obrázek 3.74 popisuje poslední část stromu chování, která začíná selektorem, který se aktivuje pouze tehdy, když AI postava nevidí hráče. Když postava stále pronásleduje hráče svým „pohledem“, LoseFocus funkce, viditelná na obrázku 3.78, donutí umělou funkci přestat se zaměřovat na hráče. Pokud je z druhého stromu chování stále nastavená proměnná „IsInCover“, tak se zavolá funkce viditelná na obrázku 3.79, kde hodnotu IsInCover nastavíme na false a postavu zvedneme ze dřepu. Pokud umělá inteligence zná poslední lokaci hráče, tak se přesune na danou lokaci, chvíli počká a poté odstaní hodnotu ze své černé tabule pomocí funkce viditelné na obrázku 3.77 a vrátí se k náhodnému chození po mapě. Indentické poslední zmíněné funkci, když AI postava uslyší zvuk střelby, přemístí se do této lokace, chvíli počká, smaže hodnotu z černé tabule a vrátí se k náhodné hlídce. Nakonec funkce náhodné hlídky, viditelná na obrázku 3.75 najde náhodný bod v blízkém okolí, kam se AI postava může dostat a poté dostane příkaz na tento bod jít, chvíli funkce počká a potom se vrátí do stromu. Pokud je tato funkce ukončena bez dokončení, AI postava okamžitě přestane chodit pomocí funkce na obrázku 3.76.



Obrázek 3.80:

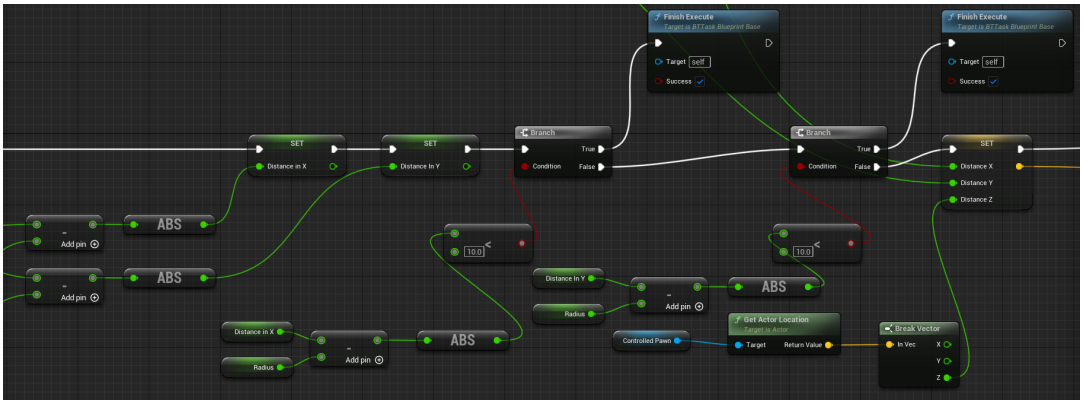


Obrázek 3.81:

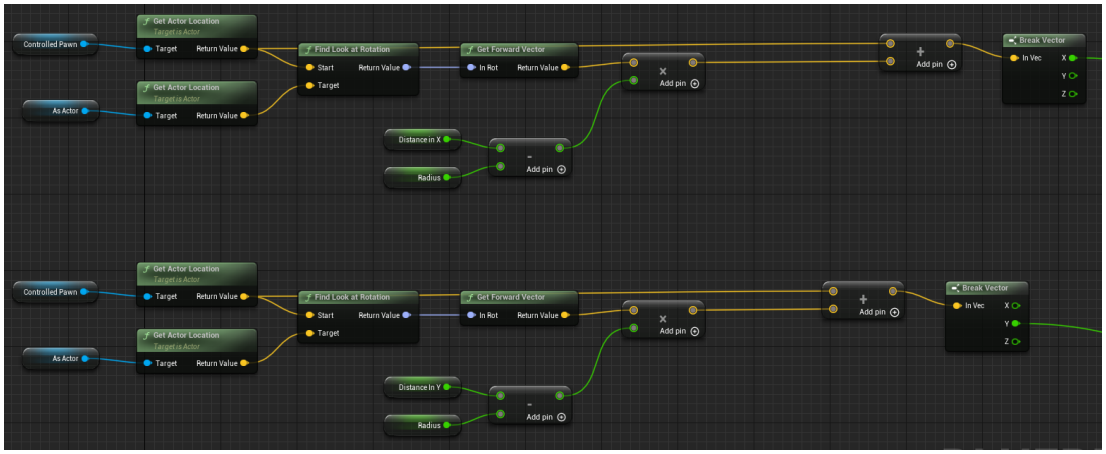


Obrázek 3.82:

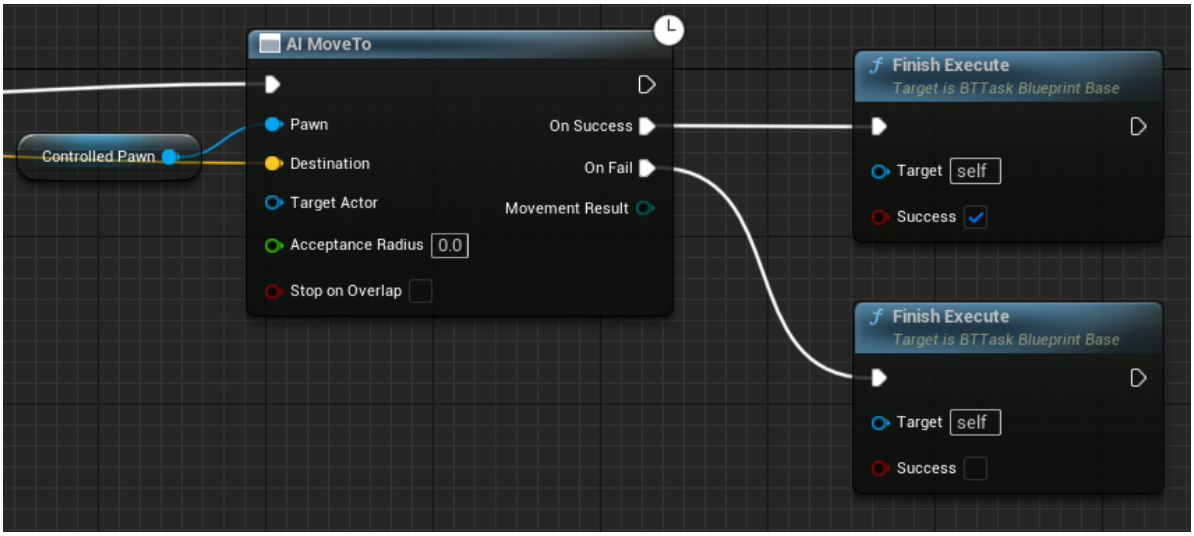
Obrázek 3.80 obsahuje druhou část stromu chování. Nejdříve, pokud je „IsInCover“ pravda, tak se zavolá identická funkce jako na obrázku 3.79. Dále se zavolá funkce, která změní zbraň AI postave podle zbraně, kterou má u sebe hráč(více v kapitole systém změny zbraně podle zbraně hráče). Pokud už se nenastavil focus dříve, funkce na obrázku 3.81. Obrázky s čísly 3.82, 3.83, 3.84, 3.85 rozvádějí funkci chození za hráčem v určité vzdálenosti. Z černé tabule získáme hráčskou proměnnou a rádius, podle které si bude udržovat vzdálenost od hráče. Nejdříve získáme lokaci hráče a NPC. Rozdíl těchto lokací nastavíme jako vzdálenost od hráče. Pokud je vzdálenost od hráče a umělé inteligence dostatečně velká, získáme přední vektor s délkou, podle kterého se určí vzdálenost, kterou AI postava na přemístnění blíž nebo dál. Když



Obrázek 3.83:

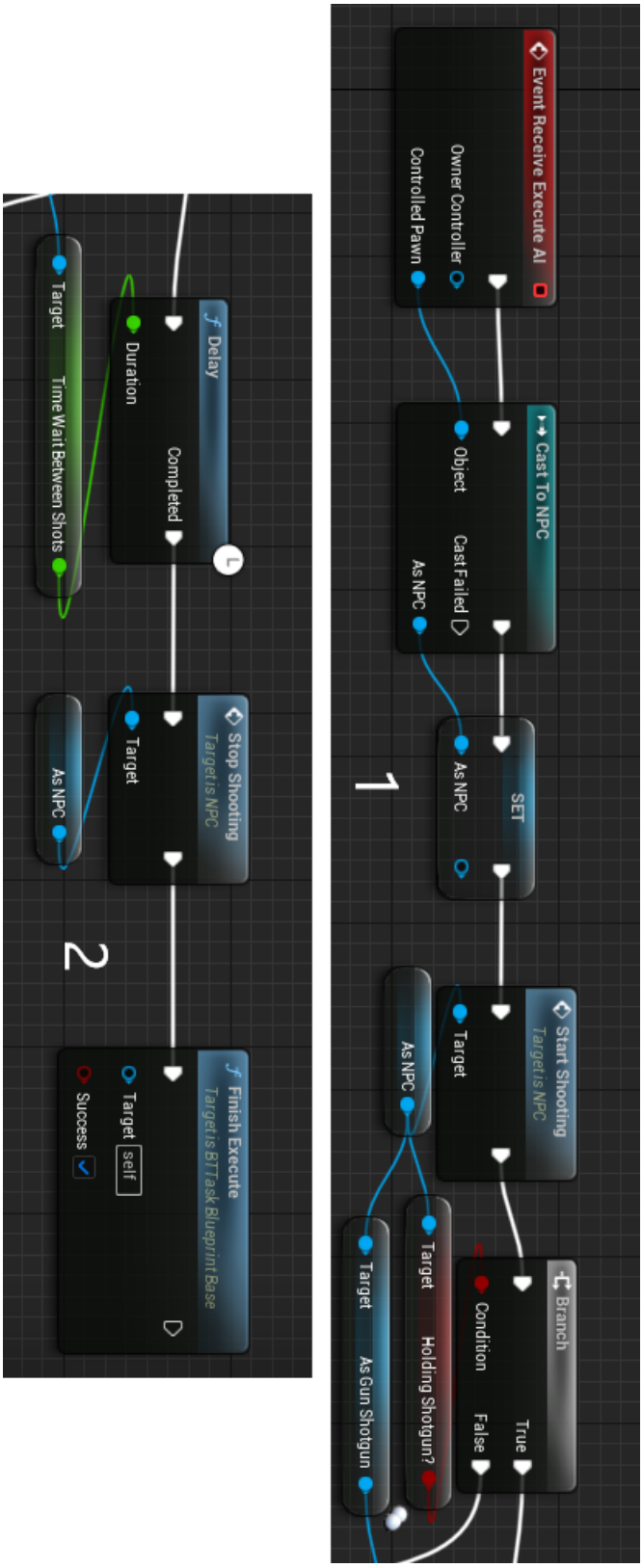


Obrázek 3.84:



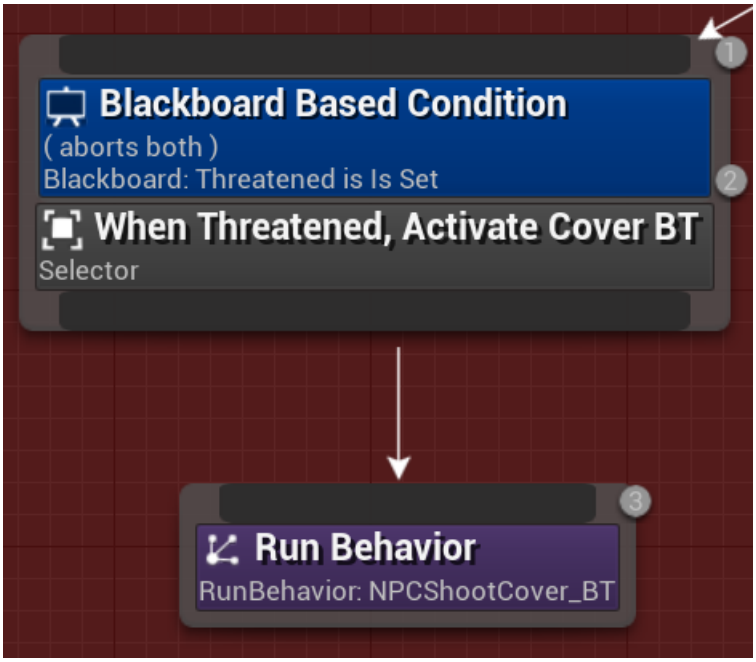
Obrázek 3.85:

se AI přemístí na tuto pozici, začne na dvě sekundy střílet na hráče pomocí funkce na obrázku 3.86. Zde se získá ukazatel na postavu umělé inteligence a uloží se do

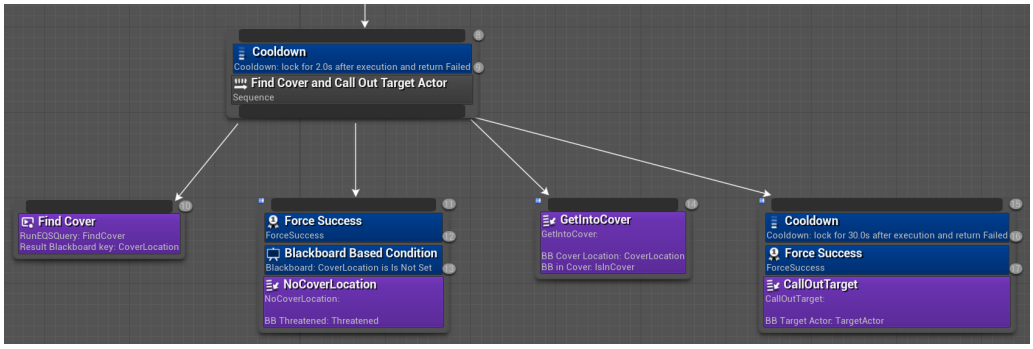


Obrázek 3.86:

proměnné, pomocí které se zavolá funkce z postavy na střílení z obrázku 3.62. Podle typu zbraně, co má u sebe AI postava, se po krátké chvíli zavolá ukončení střílení.

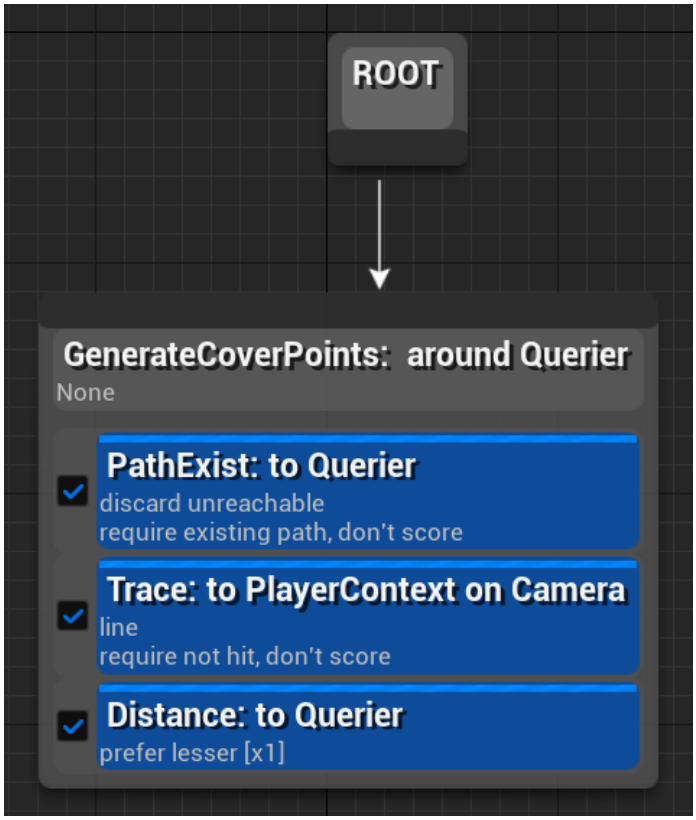


Obrázek 3.87:

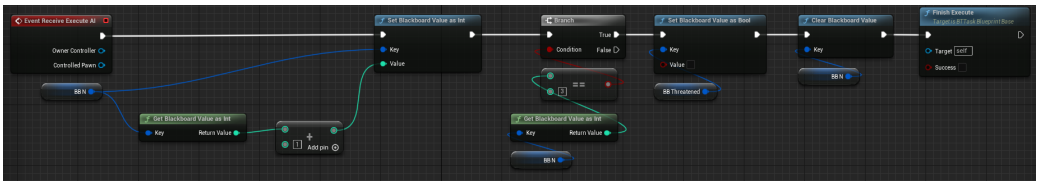


Obrázek 3.88:

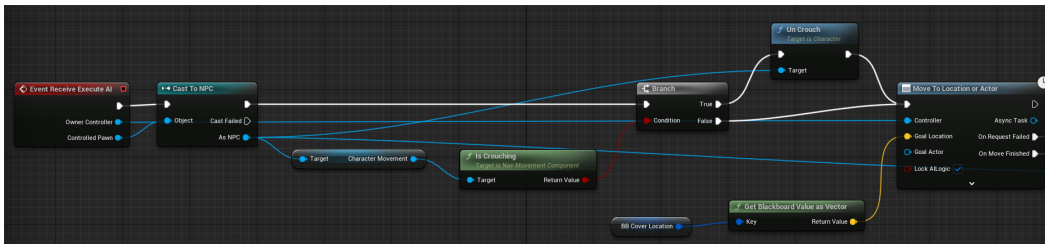
Pokud je umělá inteligence „vystrašená“, aktivuje se první část stromu chování, jak je vidět na obrázku 3.87, která volá strom na nalezení a útočení z krytu. První část, která se z tohoto stromu aktivuje je vidět na obrázku 3.88, kde se zavolá EQS funkce viditelná na obrázku 3.89, která vysílá z určité vzdálenosti do hráče přímky, která když se trefí objektu, vygeneruje na tomto místě bod. Pokud se k tomuto bodu nelze dostat, nebo není z tohoto místa vidět hráč, tak se tento bod zahazuje. AI postava bude preferovat body nejbližší ke hráči. Pokud se nenalezne žádný bod, který by tyto kritéria splňoval, zavolá se funkce z obrázku 3.90 o nenalezeném krytu. Pokud se tato funkce zavolá třikrát, AI postava přestane být ohrožená a začne vykonávat logiku z prvního stromu. Pokud se bod splňující podmínky najde, AI vykoná funkci na obrázcích 3.91 a 3.92, která zjistí, zda se umělá inteligence krčí nebo ne, pokud se krčí tak se přestane krčet a pokračuje funkce, pokud se nekrčela nebo už se nekrčí, AI přeběhne k nalezenému bodu a tam se schová. Po schování AI zavolá všechny okolní roboty pomocí funkce na obrázku 3.93. Poté se začne vykonávat levá strana stromu, která donutí umělou inteligenci sledovat hráče a střílet z krytu jak je možné



Obrázek 3.89:

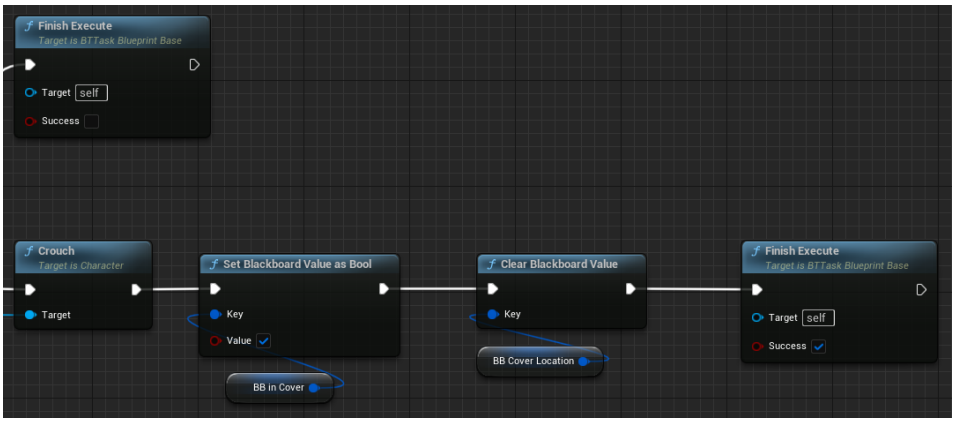


Obrázek 3.90:

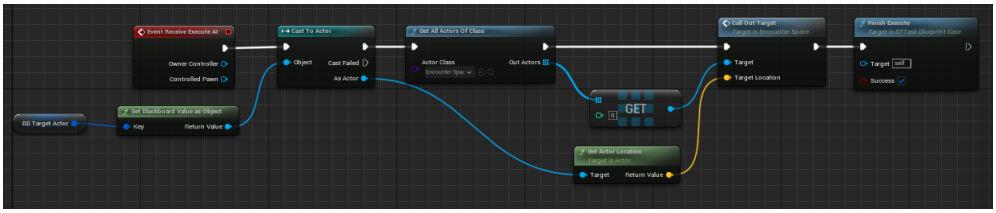


Obrázek 3.91:

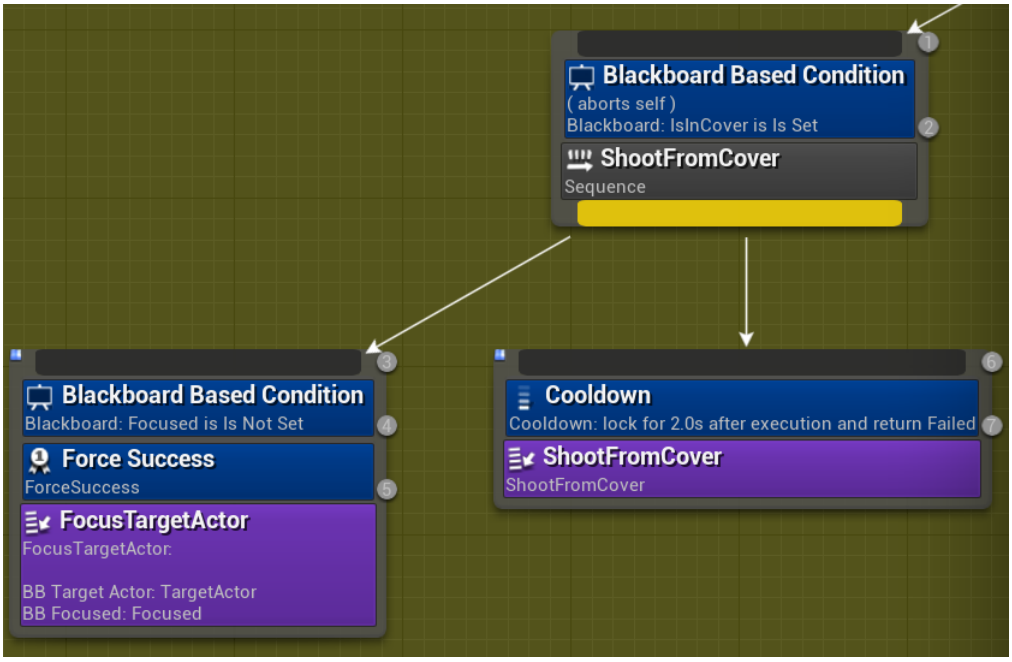
vidět na obrázku číslo 3.94. Tyto funkce jsou velice podobné předchozím zmíněným funkcím, které se jmenují FocusTargetActor a Shoot.



Obrázek 3.92:



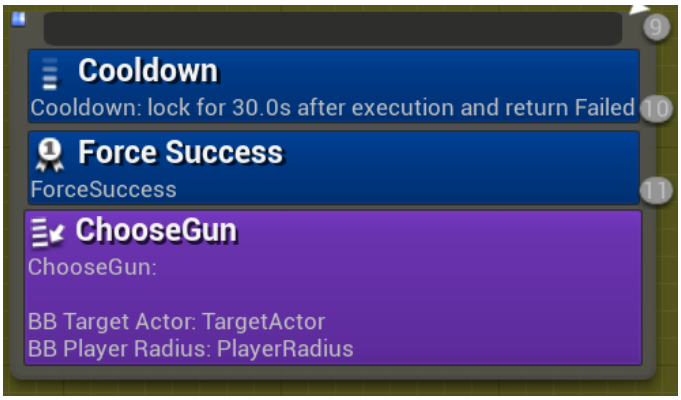
Obrázek 3.93:



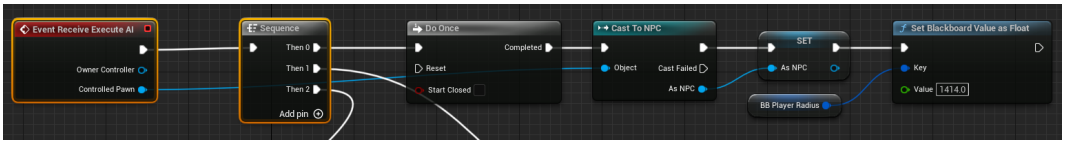
Obrázek 3.94:

3.2.2 Systém změny zbraně podle zbraně hráče

V podkapitole jménem Strom chování nebyl zmíněn uzel se jménem „ChooseGun“, neboli vyber zbraň, který je vidět na obrázku číslo 3.95. Tento uzel je jeden z prvků, který adaptuje autonomního agenta na situaci, když uvidí hráče a chce vyměnit zbraň, co má AI postava v rukách. Tento uzel se provádí každých 30 sekund.

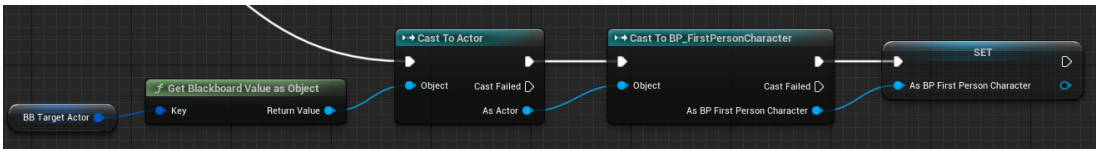


Obrázek 3.95:



Obrázek 3.96:

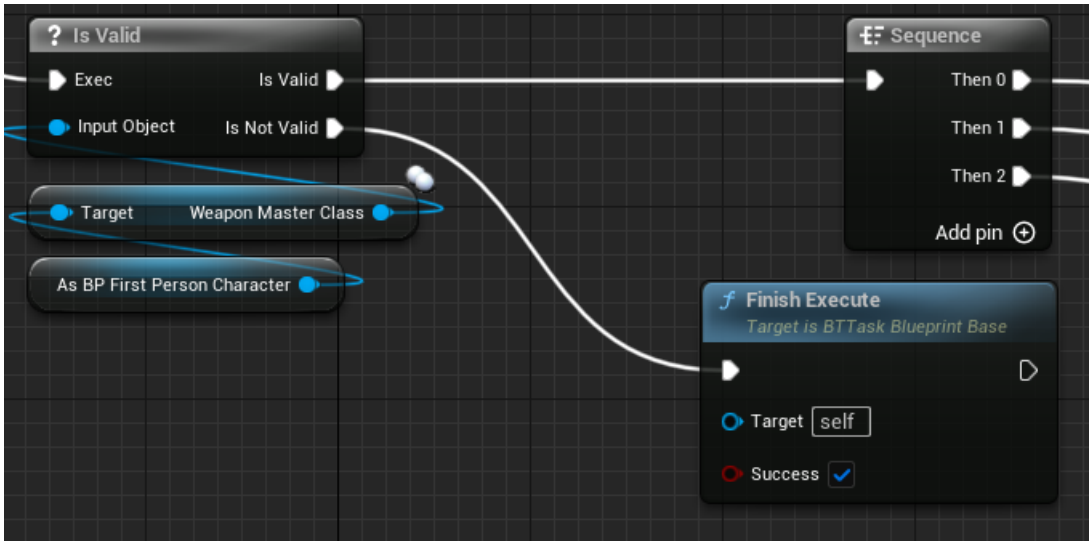
Na obrázku číslo 3.96 je vidět počátek této funkce. Pouze jednou provedeme získání ukazatele na postavu umělé inteligence a tento ukazatel přesměrujeme na blueprint této postavy a potom tuto proměnnou uložímek pozdějšímu použití. Také nastavíme vzdálenost pronásledování hráče pro automatickou pušku, protože se umělá inteligence zrodí s automatickou puškou v ruce.



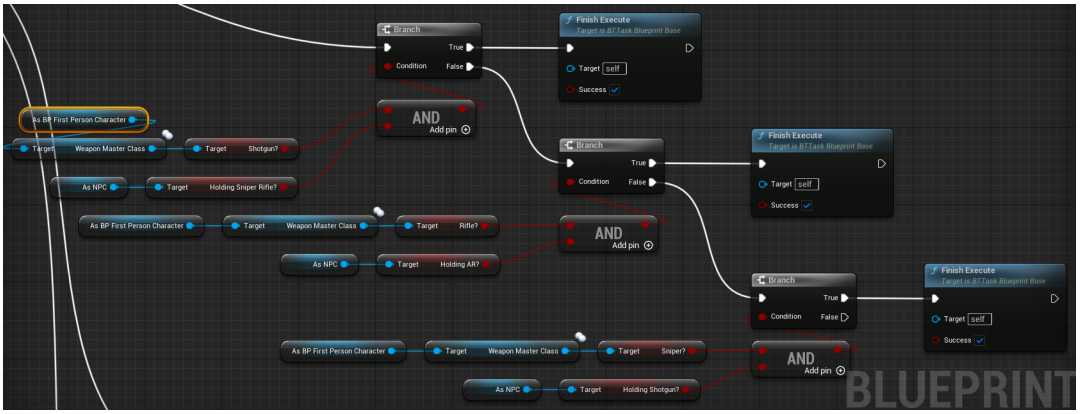
Obrázek 3.97:

Funkce na obrázku číslo 3.97 pouze ukládá hodnotu objektu z černé tabule jménem „Target Actor“, protože se v této proměnné nachází ukazatel na hráče. Tento ukazatel přesměrujeme na obecného charaktera, aby byl kompatibilní s charakterem první osoby, na který tento upraveny ukazatel znova přesměrujeme a nakonec uložíme do proměnné k pozdějšímu použití.

Po provedení akcí z obrázků 3.96 a 3.97, se na obrázku 3.98 ptáme, zda hráčská postava u sebe má zbraň. Pokud hráč zbraň u sebe nemá, tak rozhodovací strom



Obrázek 3.98:



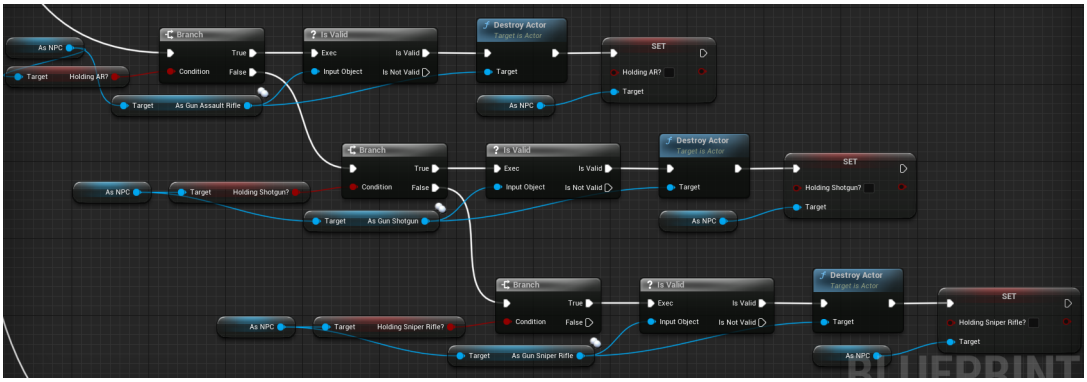
Obrázek 3.99:

končí a pokračuje zbytek logiky stromu chování, ale když se zjistí zbraň u hráče, tak logika rozhodovacího stromu pokračuje.

Obrázek 3.99 zobrazuje první část rozhodovacího stromu, který se dotazuje, zdali už postava umělé inteligence nepoužívá zbraň, která je oproti zbrani hráče neutrální nebo ve výhodě. Existují tři kombinace. Jedna z nich je když hráč má brokovnici a AI postava má odstřelovací pušku. Další je když oba používají automatické pušky a poslední kombinace je když hráč ma odstřelovací pušku a AI postava má brokovnici. Když jedna z těchto kombinací je splněná, tak končí logika rozhodovacího stromu a pokračuje zbytek logiky stromu chování. Pokud ani jedna z kombinací není pravda, tak logika rozhodovacího stromu pokračuje.

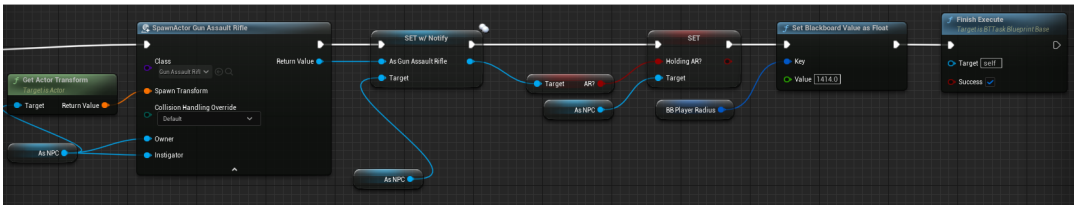
Na obrázku s číslem 3.100 je logika „ničení“ a odstranění zbraňe z ruky AI postavy, pokud předchozí větev projde bez úspěchu. Jedna ze tří větví se provede, podle zbraně co má AI postava aktuálně v ruce, a zničí se, aby se mohla do rukou použít zbraň jiná.

Na obrázcích 3.101 a 3.102, po provedení předchozí větve, „vytvoříme“ a vložíme do



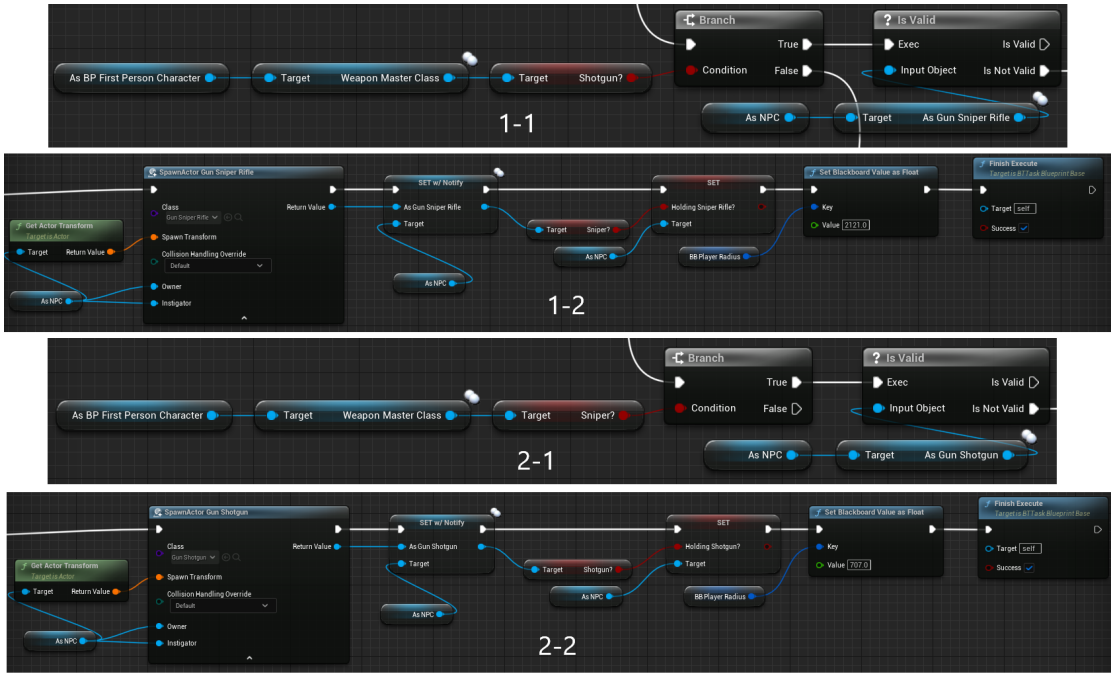
Obrázek 3.100:

Obrázek 3.101:



Obrázek 3.102:

rukou AI postavy novou zbraň podle zbraně, kterou hráč používá, aby AI postava měla výhodu oproti hráčovi. Nakonec nastavíme novou vzdálenost pronásledování hráče.



Obrázek 3.103:

Obrázek 3.103 zobrazuje téměř identickou logiku jako obrázky 3.101 a 3.102, pouze se řeší jiný typ zbraně u hráče a tím pádem se do rukou AI postavy vloží jiná zbraň.

3.2.3 Měnění atributů umělé inteligence

Atributy AI postavy budou upravovány pomocí rekurentní neuronové sítě. Jak už bylo zmíněno v návrhu autonomního agenta, tato neuronová síť bude mít vstupní vzor o velikosti 5 x 6, kde šestka značí počet prvků ve vstupních datech a pětka značí počet časových kroků ve vstupní datové sekvenci. Trénovaní proběhlo na přibližně 4150 vzorů, které byli ručně vytvořeny. Tato síť má 10 klasifikací (10 výstupů), kde:

- Výstup 0 - nejvíce zabití s útočnou puškou, vybaven útočnou puškou
- Výstup 1 - nejvíce zabití s brokovnicí, vybaven brokovnicí
- Výstup 2 - nejvíce zabití s odstřelovací puškou, vybaven odstřelovací puškou
- Výstup 3 - nejvíce zabití s útočnou puškou, vybaven brokovnicí
- Výstup 4 - nejvíce zabití s útočnou puškou, vybaven odstřelovací puškou
- Výstup 5 - nejvíce zabití s brokovnicí, vybaven útočnou puškou
- Výstup 6 - nejvíce zabití s brokovnicí, vybaven odstřelovací puškou
- Výstup 7 - nejvíce zabití s odstřelovací puškou, vybaven útočnou puškou
- Výstup 8 - nejvíce zabití s odstřelovací puškou, vybaven brokovnicí
- Výstup 9 - nerozhodný (přibližně stejný počet zabití)

```
# Potreba nainstalovat tyto knihovny
!pip install tensorflow tf2onnx onnxruntime

# Import potrebnych knihoven
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import pandas as pd
from keras.models import Sequential, load_model
from keras.layers import Dense, Activation, SimpleRNN
from keras.utils import to_categorical, plot_model
import tensorflow as tf
import tf2onnx
import onnx
import onnxruntime as ort

# Nacteni dat z~.csv pro trenink site

data = pd.read_csv('NeuralNetwork.csv',
                   dtype='float32',
                   header=None)
```

```
x_train = np.array(data.iloc[:, 1:])
y_train = np.array(list(data.iloc[:, 0]))

# Vytvori nahodny vyber pro vytvoreni testovacich vzoru

shuffler = np.random.permutation(y_train.size)

# Vypocti pocet vystupu

output_amount = len(np.unique(y_train))

# Na pozici, kde je cislo, napr. 3, se polozi vektor, ktery bude mit na
    ctvrte pozici jednicku

y_train = to_categorical(y_train)

# zmena velikosti a normalizace

# pocet casovych kroku ve vstupni datove sekvenci
time_steps = 5
# pocet prvku ve vstupnich datech
input_size = 6
# Uprava trenovacich dat
x_train = np.reshape(x_train,[-1, time_steps, input_size])
# Pretypovani dat (Pro jistotu)
x_train = x_train.astype('float32')

# Do testovacich dat vlozim zamychana trenovaci data
x_test = x_train[shuffler]
y_test = y_train[shuffler]

# Vyberu si 100 vzoru na testovani

x_test = x_test[256:356]
y_test = y_test[256:356]

# parametry site

# tvar vstupniho vzoru (matice)
input_shape = (time_steps, input_size)
# pocet vzoru na uceni v~jednom kroku (ne v~epose)
batch_size = 64
# pocet neuronu v~RNN vrstve
units = 128
# nejake nahodne vybrane neurony jsou behem treninku ignorovany
dropout = 0.2

# model je jednoducha RNN s~128 jednotkami (neuronu), vstupem je
    6-dimenzionalni pole 5 casovych kroku, vystup je vektor o~velikosti
    (-1, 10)
```

```
model = Sequential()
model.add(SimpleRNN(units=units,
                    dropout=dropout,
                    input_shape=input_shape))
model.add(Dense(output_amount))
model.add(Activation('softmax'))
model.summary()
plot_model(model,
    to_file="RNN.png",
    show_shapes=True,
    show_dtype=False,
    show_layer_names=True,
    rankdir="LR",
    expand_nested=False,
    dpi=96,
    layer_range=None,
    show_layer_activations=True,
    show_trainable=False)

# ztratova funkce pro vektor y
# pouziti optimalizatoru Adam
# presnost je dobrou metrikou pro klasifikacni ulohy
model.compile(loss='categorical_crossentropy',
              optimizer='Adam',
              metrics=['accuracy'])

# trenink neuronove site
model.fit(x_train, y_train, epochs=100, batch_size=batch_size)

# Otestovani site

loss, acc = model.evaluate(x_test, y_test, batch_size=batch_size)
print("\nTest accuracy: %.1f%%" % (100.0 * acc))
print("\nTest loss: %.1f%%" % (100.0 * loss))

# Vyzkouseni predikce

predictions = model.predict(x_test)
class_labels_keras = np.argmax(predictions, axis=-1)
# class_labels = class_labels[0]

# Vytvoreni tvaru vstupniho vektoru a konvertovani z~Keras modelu do ONNX

spec = [tf.TensorSpec((None, time_steps, input_size), tf.float32,
    name="input")]
model_proto, _ = tf2onnx.convert.from_keras(model, input_signature=spec)


#Vytvoreni relace pro onnx model
```

```
onnx_session = ort.InferenceSession(onnx_model.SerializeToString())

# Zjisteni jmena, tvaru a typu onnx modelu vstupniho vektoru

input_name = onnx_session.get_inputs()[0].name
print("input name", input_name)
input_shape = onnx_session.get_inputs()[0].shape
print("input shape", input_shape)
input_type = onnx_session.get_inputs()[0].type
print("input type", input_type)


# vystupniho vektoru

output_name = onnx_session.get_outputs()[0].name
print("output name", output_name)
output_shape = onnx_session.get_outputs()[0].shape
print("output shape", output_shape)
output_type = onnx_session.get_outputs()[0].type
print("output type", output_type)

# otestovani onnx modelu

res = onnx_session.run([output_name], {input_name: x_test})
class_labels_onnx = np.argmax(res, axis=-1)


# otestovani zachovane presnosti modelu

class_labels_success = class_labels_keras == class_labels_onnx
print(class_labels_success)

# ulozeni modelu a jeho stahnuti pro pouziti v~Unreal Enginu

onnx.save(model_proto,"/content/model.onnx")
```

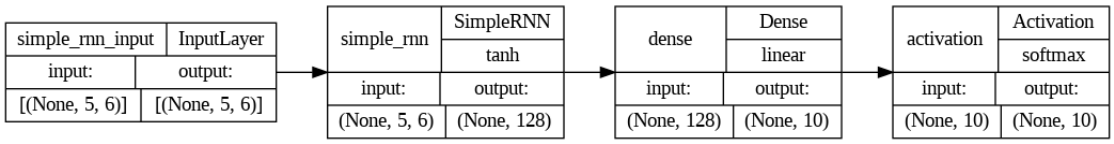
Listing 3.1: RNN_Training.ipynb

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN) (None, 128)               17280

 dense (Dense)          (None, 10)                1290

 activation (Activation) (None, 10)               0

=================================================================
```

```
Total params: 18,570
Trainable params: 18,570
Non-trainable params: 0
_________________________________________________________________
Train on 4159 samples
Epoch 1/100
4159/4159 [==============================] - 1s 153us/sample - loss:
    1.1950 - accuracy: 0.6201
Epoch 2/100
4159/4159 [==============================] - 0s 75us/sample - loss: 0.7331
    - accuracy: 0.7434
Epoch 3/100
4159/4159 [==============================] - 0s 69us/sample - loss: 0.6476
    - accuracy: 0.7685
Epoch 4/100
4159/4159 [==============================] - 0s 70us/sample - loss: 0.6003
    - accuracy: 0.7918
Epoch 5/100
4159/4159 [==============================] - 0s 70us/sample - loss: 0.6013
    - accuracy: 0.7790

...

Epoch 50/100
4159/4159 [==============================] - 0s 70us/sample - loss: 0.2717
    - accuracy: 0.8752
Epoch 51/100
4159/4159 [==============================] - 0s 67us/sample - loss: 0.2719
    - accuracy: 0.8755
Epoch 52/100
4159/4159 [==============================] - 0s 68us/sample - loss: 0.2846
    - accuracy: 0.8733
Epoch 53/100
4159/4159 [==============================] - 0s 65us/sample - loss: 0.2761
    - accuracy: 0.8805
Epoch 54/100
4159/4159 [==============================] - 0s 72us/sample - loss: 0.2743
    - accuracy: 0.8740
Epoch 55/100
4159/4159 [==============================] - 0s 68us/sample - loss: 0.2762
    - accuracy: 0.8730

...

Epoch 95/100
4159/4159 [==============================] - 0s 74us/sample - loss: 0.2625
    - accuracy: 0.8745
Epoch 96/100
4159/4159 [==============================] - 0s 72us/sample - loss: 0.2720
    - accuracy: 0.8706
```

```
Epoch 97/100
4159/4159 [==============================] - 0s 69us/sample - loss: 0.2363
    - accuracy: 0.8884
Epoch 98/100
4159/4159 [==============================] - 0s 68us/sample - loss: 0.2539
    - accuracy: 0.8769
Epoch 99/100
4159/4159 [==============================] - 0s 71us/sample - loss: 0.2465
    - accuracy: 0.8788
Epoch 100/100
4159/4159 [==============================] - 0s 66us/sample - loss: 0.2383
    - accuracy: 0.8868
```

Listing 3.2: Průběh učení Keras modelu



Obrázek 3.104: Diagram použité neuronové sítě

Po natrénování a přeložení Keras modelu do ONNX modelu se můžeme přesunout do Unreal Enginu, kde si nainstalueje plugin Neural Network Inference (NNI), který je dostupný do verze Unreal Enginu 5.2. V Unreal Enginu si vytvoříme dvě C++ třídy, kde jedna ze tříd bude mít rodičovskou třídu pojmenovanou BlueprintFunctionLibrary a druhá má rodičovskou třídu NeuralNetworkInference.NeuralNetwork.

```
#pragma once

#include "CoreMinimal.h"
#include "Kismet/BlueprintFunctionLibrary.h"
#include "MyNeuralNetwork.h"
#include "STBlueprintLibrary.generated.h"

class UNeuralNetwork;

/**
 *
 */
UCLASS()
class ROBOHELL_API USTBlueprintLibrary : public UBlueprintFunctionLibrary
{
   GENERATED_UCLASS_BODY()

   UFUNCTION(Exec, BlueprintCallable, Category = "AI")
   static int CallRNN(UNeuralNetwork* Model, float ARKills, float
       ARShotgunKills, float ARSniperKills, float ARHeld, float
       ShotgunHeld, float SniperHeld);
```

```
    static UMyNeuralNetwork* MyNetwork;
};
```

Listing 3.3: STBlueprintLibrary.h

```
#include "STBlueprintLibrary.h"
#include "MyNeuralNetwork.h"

USTBlueprintLibrary::USTBlueprintLibrary(const FObjectInitializer&
    ObjectInitializer) : Super(ObjectInitializer) { }

UMyNeuralNetwork* USTBlueprintLibrary::MyNetwork = nullptr;

int USTBlueprintLibrary::CallRNN(UNeuralNetwork* Model, float ARKills,
    float ARShotgunKills, float ARSniperKills, float ARHeld, float
    ShotgunHeld, float SniperHeld)
{
    Model->AddToRoot();
    MyNetwork = NewObject<UMyNeuralNetwork>();
    Model->SetDeviceType(ENeuralDeviceType::CPU);
    MyNetwork->Network = Model;
    int helper = MyNetwork->URunModel(ARKills, ARShotgunKills,
        ARSniperKills, ARHeld, ShotgunHeld, SniperHeld);
    return helper;
}
```

Listing 3.4: STBlueprintLibrary.cpp

Knihovna BlueprintFunctionLibrary slouží k zobrazení funkce CallRNN v Blueprintu. funkce CallRNN musí mít UFUNCTION identifikátor, kde exec hlavně znamená, že funkci lze spustit i v příkazovém řádku (např. pro otestování). BlueprintCallable jak už může z názvu vyplívat zaručuje, že funkce se zobrazí v Blueprintu a kategorie určuje, kde se bude v menu výběru nacházet. Tato funkce má následujících 7 vstupů:

- Vstup 0 - ONNX model
- Vstup 1 - počet zabití s útočnou puškou
- Vstup 2 - počet zabití s brokovnicí
- Vstup 3 - počet zabití s odstřelovací puškou
- Vstup 4 - vybaven útočnou puškou na konci kola
- Vstup 5 - vybaven brokovnicí na konci kola
- Vstup 6 - vybaven odstřelovací puškou na konci kola

a funkce CallRNN vrací proměnnou int po zjištění klasifikace zjištěním největší hodnoty ve vektoru. Dále připravuje proměnnou MyNetwork, která ukazuje na vytvořenou třídu UMyNeuralNetwork. V .cpp souboru inicializujeme třídu STBlueprintLibrary a nastavujeme deklarované proměnné MyNetwork nullptr. Ve funkci CallRNN přidáme Model do rootu, aby model nemohl být zahozen „sběračem odpadků“. Dále nastavujeme proměnnou MyNetwork nový objekt, který ukazuje na třídu UMyNeuralNetwork. U modelu nastavujeme, aby používal na inferenci sítě procesor a do proměnné MyNetwork vložíme ONNX Model. Nakonec vytvoříme int proměnnou se jménem helper, která volá funkci vytvořenou ve třídě UMyNeuralNetwork URunModel, který si vezme nevyužité vstupní proměnné. Po proběhnutí URunModel vrátíme do Blueprintu helper.

```
#pragma once

#include "CoreMinimal.h"
#include "NeuralNetwork.h"
#include <vector>
#include <algorithm>
#include "MyNeuralNetwork.generated.h"

/**
 *
 */
UCLASS()
class ROBOHELL_API UMyNeuralNetwork : public UNeuralNetwork
{
   GENERATED_BODY()

public:
   UPROPERTY(Transient)
   UNeuralNetwork* Network = nullptr;
   UMyNeuralNetwork();

   TArray<float> Tinput;
   TArray<float> Thelp;
   std::vector<float> vinput;

   int URunModel(float ARKills, float ARShotgunKills, float ARSniperKills,
       float ARHeld, float ShotgunHeld, float SniperHeld);

   TArray<float> UPreProcess(float ARKills, float ARShotgunKills, float
       ARSniperKills, float ARHeld, float ShotgunHeld, float SniperHeld);
};
```

Listing 3.5: MyNeuralNetwork.h

```
#include "MyNeuralNetwork.h"

UMyNeuralNetwork::UMyNeuralNetwork()
{
```

```
    Network = nullptr;
}

int UMyNeuralNetwork::URunModel(float ARKills, float ARShotgunKills, float
    ARSniperKills, float ARHeld, float ShotgunHeld, float SniperHeld)
{
    if (Network == nullptr || !Network->IsLoaded())
    {
        return 10;
    }
    Tinput = UPreProcess(ARKills, ARShotgunKills, ARSniperKills, ARHeld,
        ShotgunHeld, SniperHeld);
    Network->SetInputFromArrayCopy(Tinput);

    // Run UNeuralNetwork
    Network->Run();

    TArray<float> OutputTensor =
        Network->GetOutputTensor().GetArrayCopy<float>();
    std::vector<float> OutputVector;

    if (OutputVector.empty() == true)
    {
        OutputVector.resize(10, 0);
    }

    for (int i = 0; i < OutputTensor.Num(); i++) {
        OutputVector[i] = OutputTensor[i];
    }

    int maxElementIndex = std::max_element(OutputVector.begin(),
        OutputVector.end()) - OutputVector.begin();
    return maxElementIndex;
}

TArray<float> UMyNeuralNetwork::UPreProcess(float ARKills, float
    ARShotgunKills, float ARSniperKills, float ARHeld, float ShotgunHeld,
    float SniperHeld)
{
    if (vinput.empty() == true)
    {
        vinput.resize(30, 0);
    }
    std::vector<float> input = { ARKills, ARShotgunKills, ARSniperKills,
        ARHeld, ShotgunHeld, SniperHeld };

    vinput.insert(vinput.begin(), input.begin(), input.end());
    vinput.erase(vinput.end() - 6, vinput.end());

    Thelp.Reset();
```

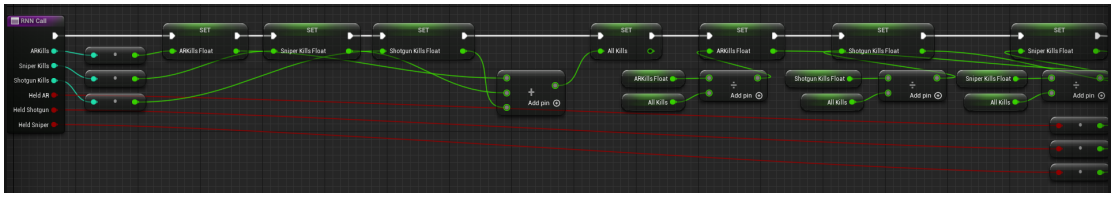
```
    Thelp.SetNumZeroed(vinput.size());

    for (int i = 0; i < vinput.size(); i++) {
        Thelp[i] = vinput[i];
    }

    return Thelp;
}
```

Listing 3.6: MyNeuralNetwork.cpp

Ve UMyNeuralNetwork třídě si v header souboru inicializujeme proměnnou typu transient („přechodná“) UNeuralNetwork pojmenovanou Network, do které prozatím vložíme nullptr. Přechodná proměnná se nijak neukládá, zůstává pouze v paměti. Dále vytváříme konstruktor pro tuto třídu. Připravíme si tři proměnné. Dvě proměnné budou typu TArray<float>. TArray je běžná kontejnerová třída v Unreal Engine a model neuronové sítě ho zde vyžaduje. Třetí proměnná je běžný float vektor. Nakonec máme dvě funkce, jedna už byla zmíněna a má jméno URunModel, kde voláme model neuronové sítě na výpočet vstupu. Druhá funkce se jménem UPreProcess vytváří vektor požadované velikosti v jedné dimenzi, který potom převedeme na TArray<float>, který můžeme vložit do sítě. Originálně požaduje Keras Model a čistý ONNX model vektor o velikost [-1, 5, 6]. Zde v této knihovně potřebujeme vytvořit jednorozměrný TArray<float>o velikosti 5 x 6, takže 30 prvků. V .cpp souboru máme konstruktor, který nastavuje Networku nullptr hodnotu, pokud nebyla dříve nastavena. V našem případě je v Network uložen náš model. Kdyby se ale nenačtl model správně, vrátíme hodnotu 10, která nic v Blueprintu nenastavuje. Dalším krokem je vytvořit si vstupní TArray v pomocné třídě UPreProcess. pokud je vinput vektor prázdný (na začátku bude vždy prázdný), tak funkce naplní vektor třiceti nulami, aby byla možnost s vektorem pracovat. Vytvoříme vector<float> s jménem input. Do tohoto vektoru vložíme všechna naše data, které se převzali ze hry, přesněji z Blueprintu. Tento vektor vložíme do vinput vektoru a na konci vektoru smažeme šest pozic. Poté resetujeme TArray pojmenovaný Thelp, aby v této proměnné nebyli zbytkové hodnoty z předchozího učení. Thelp inicializujeme vyplněním nulami na třiceti pozicích pomocí vinput.size(). Díky interoperabilitě std::vector a TArray můžeme jednoduchým for cyklem vzít hodnoty z vektoru na i pozicích a vložit je na stejné pozice v TArray. Tento TArray nakonec vrátíme zpět do hlavní funkce. Jako vstup do proměnné Network zvolíme čerstvě naplněný Tinput a poté necháme Network zapnout. Pomocí proměnné OutputTensor si z Network vybereme výstup sítě a zároveň vytvoříme nový std::vector<float>, který pojmenuje OutputVector. Tento vektor naplníme deseti nulami, aby jsme měli vektor o velikosti 10. Stejně jako ve funkci UPreProcess, zde také převedeme hodnoty z OutputTensor do OutputVector. Nakonec využijeme std::max_element z knihovny <algorithm>, pomocí kterého zjistíme pozici největší hodnoty, kterou potom vrátíme zpět do STBlueprintLibrary.ccp. Tato hodnota se vrátí do Blueprint, kde s její pomocí vybereme změnu atributů pro AI postavu.

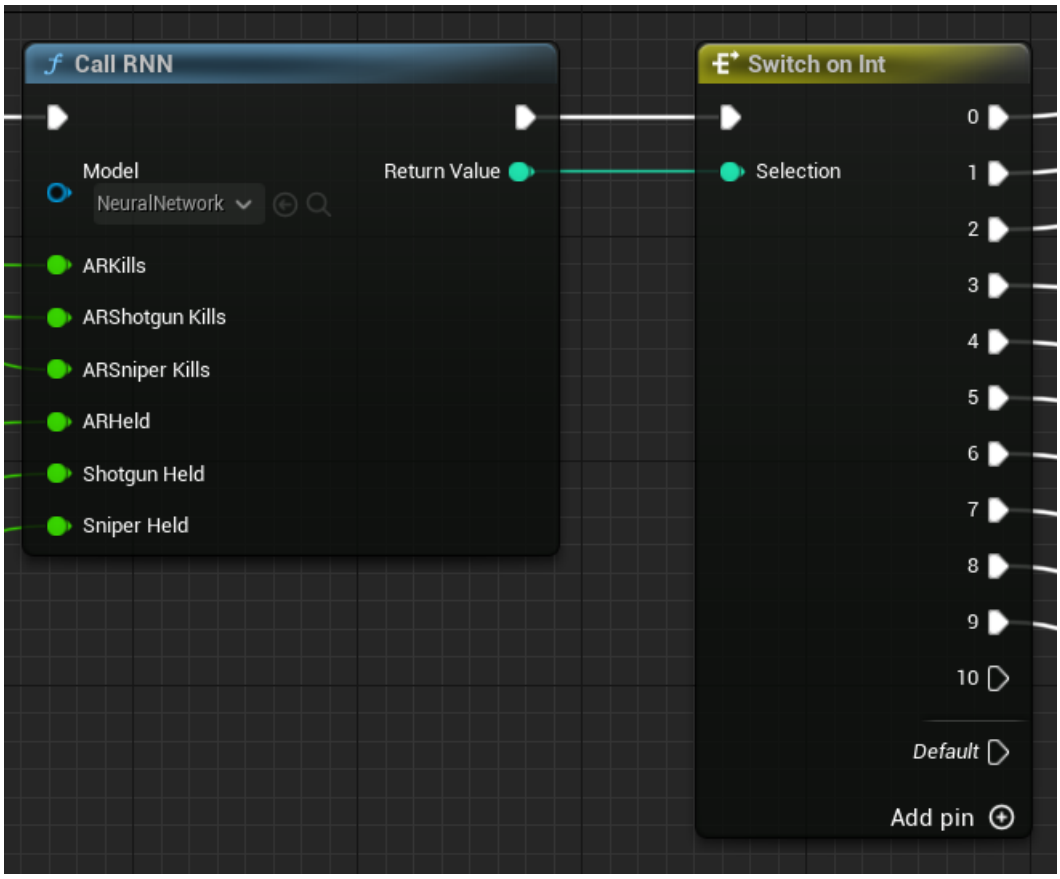
V Blueprintu máme dříve zmíněnou funkci RNN Call (obrázek číslo 3.47).

Na prvním obrázku (číslo 3.105) je vidět převod hodnot z int a bool na float, kde



Obrázek 3.105:

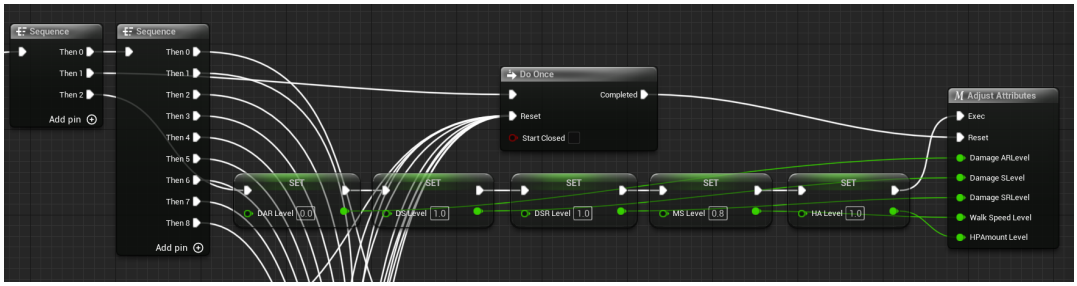
hodnoty ARKill, ShotgunKill a SniperKill jsou prvně převedeny do float a poté vyděleny součtem těchto hodnot, aby součet těchto tří floatů se rovnal 1.0f. Bool hodnoty jsou převedeny do floatu a žádné další úpravy nejsou potřeba.



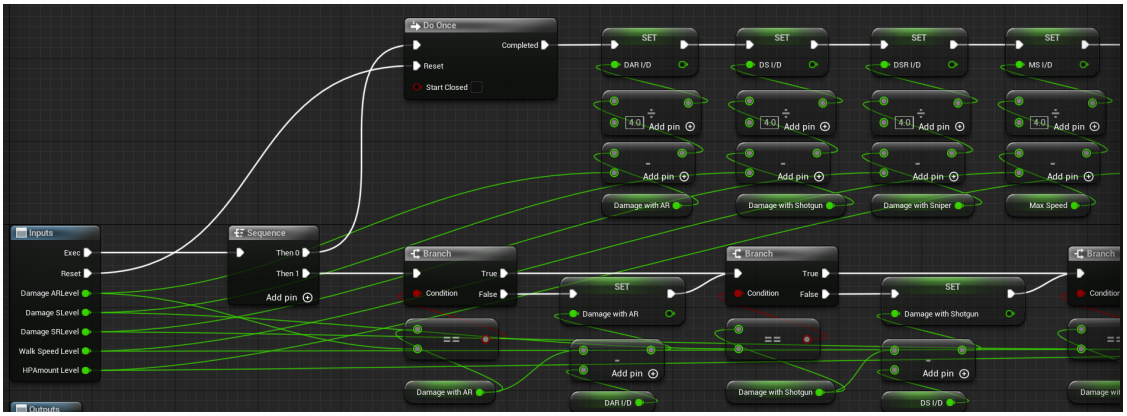
Obrázek 3.106:

Obrázek číslo 3.106 zobrazuje Blueprint funkci, kterou jsme sestrojili v e Visual studio. Vložíme do této funkce všechny potřebné vstupy, abych na druhé straně získali výstup. Tento výstup použijeme ve switchy na výběr úprav atributů.

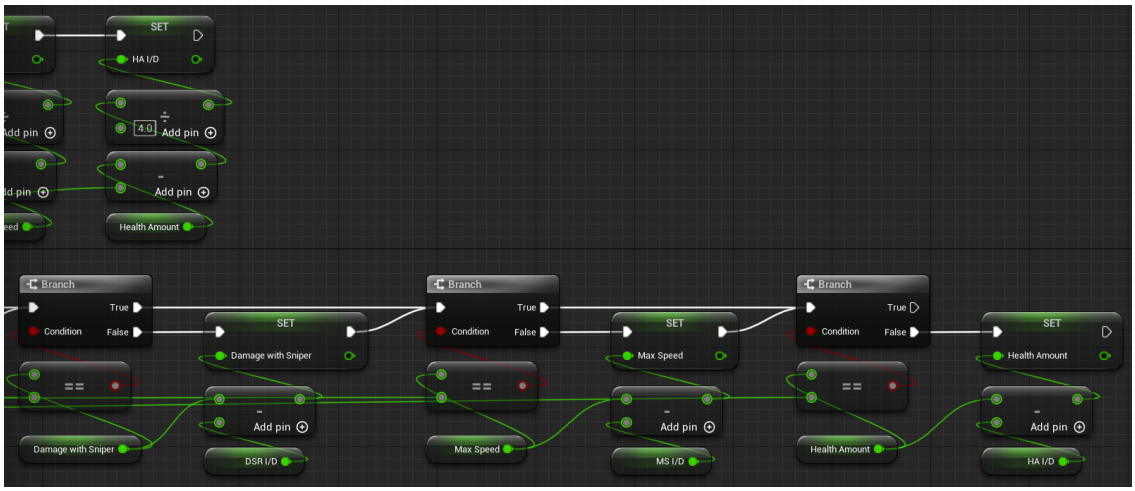
Tento obrázek (číslo 3.107) zobrazuje jeden ze switchů, který nastavuje úrovně pěti atributů, kde tři z nich nastavují poškození hráčské zbraně vůči AI postavě. Čtvrtý vstup nastavuje rychlost chůze a běhu AI postavy a poslední vstup nastavuje počet životů AI postavy. Např. na tomto obrázku je vidět, že za čtyři kola (logika na obrázcích číslo 3.108 a 3.109) bude poškození z útočné pušky nulové a AI postava bude o 20% pomalejší.



Obrázek 3.107:



Obrázek 3.108:



Obrázek 3.109:

Postupné nastavování se nachází v makru „Adjust Attributes“, který má logiku zobrazenou na obrázcích 3.108 a 3.109. Pouze jednou nastavíme rychlost snižování nebo zvyšování atributů (horní část obrázků). Tato část se resetuje pokaždé, když vznikne jiný výstup z neuronové sítě. Pokud nevznikne nový výstup ze sítě, atributy se upraví o stejné hodnoty jako poslední kolo. Upravování hodnot je ve spodní části obrázků. Funkce změny atributů se navzájem nevynulují. Hodnoty, co jsou nižší nebo vyšší, než jaké v ten moment mají být např. při jiném výstupu neuronové sítě,

se postupně upravují na hodnoty, jaké byli nastaveny neuronovou sítí.

Závěr

V této bakalářské práci bylo hlavním cílem sestrojit jednoduchého autonomního agenta, který se dokáže přizpůsobit situaci a upravovat sám sebe podle chování uživatele.

Výsledkem práce je jednoduchá střílečka z pohledu první osoby, ve které na vás po vlnách útočí roboti, kteří si dokáží vyměnit svoji zbraň podle zbraně, kterou v ten moment hráč používá. Roboti si dokáží měnit i styl boje, kde se zbraní na vzdálenost se snaží od hráče držet dál, než např. robot s brokovnicí, který se snaží ke hráči přiblížit. Nakonec si roboti dokáží upravovat svoje atributy (např. život) sběrem informací (např. počet smrtí automatickou puškou), které se předají do rekurentní neuronové sítě, ze které získáme výsledek, podle kterého potom roboti svoje atributy upraví.

Hru jsem otestoval sám, kde kromě chyb z Unreal Enginu (z důvodu nedostatečného výkonu, paměti atd.) jsem nezaznamenal žádnou chybu. Agent reaguje podle očekávání.

Jednou z možností, kterým by se dal vylepšit tento agent je plnohodnotným samostatným učením. Je to strojové učení, kdy model se učí z dat samotných, aniž by vyžadoval explicitní lidské značení. Místo toho využívá vnitřní strukturu nebo vlastnosti dat k vytvoření náhradních dozorových signálů. Tyto signály vedou model během učícího procesu, což mu umožňuje naučit se užitečné reprezentace ze vstupních dat.

Další možností je Reinforcement Learning (Učení s posilováním). Je to druh strojového učení, který se zaměřuje na trénování agenta, aby se rozhodoval a jednal v prostředí tak, aby maximalizoval získanou odměnu. V tomto přístupu není agentu předkládán tréninkový dataset s označenými příklady, ale místo toho se učí průběžně komunikovat s prostředím, získávat zpětnou vazbu v podobě odměn nebo trestů a optimalizovat své akce na základě těchto zpětných vazeb.

Literatura

[1] ANDERSON, James A. a J. P. SUTTON. A network of networks: Computation and neurobiology. *World Congress on Neural Networks, San Diego*. 1995, **1**, 561—568. ISBN 080581745X.

[2] ASPLAND, Matt. Sprinting And Stamina - Unreal Engine 5 Tutorial. *Youtube* [online]. 2005, 12. 5. 2022 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/watch?v=seGQy-GBfhY

[3] BISHOP, Christopher M. *Neural Networks for Pattern Recognition*. Oxford: Clarendon Press, 1995. ISBN 9780080512617.

[4] BOHL, Evans. How to Animate Characters in UE5. *Youtube* [online]. 2005, 20. 5. 2022 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/watch?v=w9mijf-gKOg

[5] BROOKS, Rodney A. Intelligence without representation. *Artificial Intelligence* [online]. 1991, **47**(1-3), 139-159 [cit. 2023-03-26]. ISSN 0004-3702. Dostupné z: doi:10.1016/0004-3702(91)90053-M

[6] DEVMINUTE. Crouching System In 3 Minutes ! Smooth with Animations | UE4 & 5 Tutorial. *Youtube* [online]. 2005, 12. 8. 2021 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/watch?v=71HcMAr-mKE

[7] DEVOP, Unreal. Unreal Engine - Making your character look in the camera's direction (My Simple Method). *Youtube* [online]. 2005, 8. 2. 2022 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/watch?v=3hy7Oh7r64E

[8] ESTERLE, Lukas. Deep learning in multiagent systems. In: IOSIFIDIS, Alexandros a Anastasios TEFAS, ed. *Deep Learning for Robot Perception and Cognition*. Amsterdam: Elsevier, 2022, s. 435-460. ISBN 978-0-323-85787-1. Dostupné z: doi:10.1016/B978-0-32-385787-1.00022-1

[9] FRAMES, Weird. Bringing Deep Learning to Unreal Engine 5 - Pt. 1. In: *Medium* [online]. 2012 [cit. 2023-08-01]. Dostupné z: https://medium.com/@weirdframes/bringing-deep-learning-to-unreal-engine-5-pt-1-aa84c8c05ffa

[10] FRAMES, Weird. Bringing Deep Learning to Unreal Engine 5 - Pt. 2. In: *Medium* [online]. 2012 [cit. 2023-08-01]. Dostupné z: https://medium.com/@weirdframes/bringing-deep-learning-to-unreal-engine-5-pt-2-51c1a2a2c3

[11] GAZZANIGA, Michael S. On Neural Circuits and Cognition. *Neural Computation*. 1995, **7**(1), 1-12. ISSN 0899-7667. Dostupné z: doi:10.1162/neco.1995.7.1.1

[12] GENESERETH, Michael R. a Steven P. KETCHPEL. Software agents. *Communications of the ACM* [online]. 1994, **37**(7), 48-53 [cit. 2023-03-26]. ISSN 1557-7317. Dostupné z: doi:10.1145/176789.176794

[13] GREFENSTETTE, John J. a James E. BAKER. How Genetic Algorithms Work: A Critical Look at Implicit Parallelism. *Proceedings of the Third International Conference on Genetic Algorithms* [online]. 1989, **3**, 20-27 [cit. 2023-05-21]. ISBN 1558600063. Dostupné z: doi:10.5555/93126.93136

[14] LALEY, Ryan. Shooter AI. In: *Youtube* [online]. 2005, 2019 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/playlist?list=PLZc8ZAEBk3AOpIEZgKtpS9n66-fEff3xR

[15] LIEBERMAN, Henry. Autonomous interface agents. In: PEMBERTON, Steven. *CHI '97: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. New York: Association for Computing Machinery, 1997, s. 67-74. ISBN 978-0-89791-802-2. Dostupné z: doi:10.1145/258549.258592

[16] MICROSOFT. OnnxRuntime-UnrealEngine. In: *GitHub* [online]. 2008 [cit. 2023-08-01]. Dostupné z: https://github.com/microsoft/OnnxRuntime-UnrealEngine

[17] ORMSTAD, Ben. UE5 Tutorial | Create Movable Player Character From Scratch (Blueprints). *Youtube* [online]. 2005, 1. 12. 2021 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/watch?v=Tf4rpJfOy54

[18] PARSONS, Simon a Michael WOOLDRIDGE. Game Theory and Decision Theory in Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems* [online]. 2002, **5**(3), 243-254 [cit. 2023-03-21]. Dostupné z: doi:10.1023/A:1015575522401

[19] PATEL, Harsh a Purvi PRAJAPATI. Study and Analysis of Decision Tree Based Classification Algorithms. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING* [online]. 2018, **6**(10), 74-78 [cit. 2023-05-21]. ISSN 2347-2693. Dostupné z: doi:10.26438/ijcse/v6i10.7478

[20] QUINLAN, J. Ross. Simplifying decision trees. *International Journal of Man-Machine Studies* [online]. 1987, **27**(3), 221-234 [cit. 2023-05-21]. ISSN 0020-7373. Dostupné z: doi:10.1016/S0020-7373(87)80053-6

[21] RUNTIME, ONNX. Use ONNX Runtime and OpenCV with Unreal Engine 5 New Beta Plugins. *Youtube* [online]. 2005 [cit. 2023-08-01]. Dostupné z: https://www.youtube.com/watch?v=LX1w_etaftY

[22] SAXENA, Sharad. *Tree-Based Machine Learning Methods in SAS Viya* [online]. North Carolina: SAS Institute, 2022 [cit. 2023-05-21]. ISBN 9781954846647. Dostupné z: https://ebookcentral.proquest.com/lib/cvut/detail.action?docID=6893594

[23] SYSWERDA, Gilbert. A Study of Reproduction in Generational and Steady-State Genetic Algorithms. *Foundations of Genetic Algorithms* [online]. 1991, **1**, 94-101 [cit. 2023-05-21]. ISSN 1081-6593. Dostupné z: doi:10.1016/B978-0-08-050684-5.50009-4

[24] TAYLOR, John G. *The Promise of Neural Networks*. London: Springer London, 1993. ISBN 978-1-4471-0395-0.

[25] VINYALS, Oriol, Igor BABUSCHKIN, Wojciech Marian CZARNECKI a David SILVER. *Grandmaster level in StarCraft II using multi-agent reinforcement learning: AlphaStar* [online]. November 2019, 23 [cit. 2023-05-21]. Dostupné z: doi:10.1038/s41586-019-1724-z

[26] *Unreal Engine a Marketplace* [online]. USA: Epic Games, 2023 [cit. 2023-07-26]. Dostupné z: www.unrealengine.com

[27] UNREAL BY YOURSELF. First Person Series. In: *Youtube* [online]. 2005, 2023 [cit. 2023-07-28]. Dostupné z: https://www.youtube.com/playlist?list=PLZc8ZAEBk3AOpIEZgKtpS9n66-fEff3xR

[28] WEBB, Jack. Middle-Earth: Shadow Of War - Nemesis System Complete Guide. *TheGamer* [online]. Canada, 2023 [cit. 2023-05-21]. Dostupné z: https://www.thegamer.com/middle-earth-shadow-of-war-orc-nemesis-system-complete-guide/