Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Vývoj herní aplikace v enginu Godot

René Fiala

© 2024 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

René Fiala

Informatika

Název práce

Vývoj herní aplikace v enginu Godot

Název anglicky

Game application development in the Godot game engine

Cíle práce

Hlavním cílem bakalářské práce je představit herní engine Godot a vhodné návrhové a testovací metodiky v kontextu vývoje aplikací v tomto prostředí.

Dílčími cíli práce jsou:
- Vyvinutí výstupní 3D akční herní aplikace
- Provedení uživatelského testování a zpracování jeho výsledků
- Návrh a implementace znovu použitelného systému dynamické hudby v rámci výstupní aplikace

Metodika

V teoretické části budou popsány návrhové a testovací metodiky objektově orientovaného programování a jejich aplikace v herním vývoji. Dále bude představen herní engine Godot, jeho struktura a funkce.

V praktické části budou poznatky z teoretické části aplikovány k navržení a vývoji výsledné 3D herní aplikace pro desktopové platformy. Bude navrženo a provedeno uživatelské testování a na základě získaných výsledků bude aplikace upravena.

Doporučený rozsah práce

35-40 stran

Klíčová slova

Godot, herní engine, herní vývoj, programování, aplikace, hra, 3D

Doporučené zdroje informací

GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISIDES, 2003. Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů. Praha: Grada. Moderní programování. ISBN 80-247-0302-5.

LINIETSKY, Juan, Ariel MANZUR a Godot Engine community, 2014. Godot Engine 4.0 documentation in English [online]. Revision a42d4323. ©2014- [cit. 2023-05-09]. Dostupné z: https://docs.godotengine.org/en/4.0/

NYSTROM, Robert, 2014. Game Programming Patterns. Genever Benning. ISBN 0990582906.

SCHELL, Jesse, 2008. The Art of Game Design: A Book of Lenses. Florida: CRC Press. ISBN 0123694965.

SCHULTZ, Charles P. a Robert Denton BRYANT, 2016. Game Testing: All in One. 3rd Edition. Virginia: Mercury Learning and Information. ISBN 1942270763.

VAN VERTH, James M. a Lars M. BISHOP, 2015. Essential Mathematics for Games and Interactive Applications. 3rd Edition. Florida: CRC Press. ISBN 1482250926.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Tomáš Benda

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 28. 11. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 9. 2. 2024

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 28. 02. 2024

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci „Vývoj herní aplikace v enginu Godot“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15. března 2024 ______________________________

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Tomáši Bendovi za pomoc, věcné připomínky a rady, vstřícnost a také trpělivost při zpracování práce.

Věnováno Štrúdlovi.

Přišel jsi i odešel, když jsem tuto práci tvořil.

Nevím, kam jsi zmizel, ale snad se tam máš dobře.

Vývoj herní aplikace v enginu Godot

Abstrakt

Tato bakalářská práce se zabývá vývojem akční 3D herní aplikace v open-source herním enginu Godot.

V teoretické části práce jsou popsány návrhové a vývojové postupy objektově orientovaného programování, které jsou aplikovatelné při vývoji aplikací v enginu Godot, následované metodami testování kódu a uživatelského testování. Dále je představen samotný engine Godot, jeho struktura a důležité součásti. Následující kapitoly se zabývají technikami pro adaptivní hudbu ve hrách, serializaci dat za účelem ukládání a načítání hry a principem akčních her obecně.

V praktické části práce byly poznatky z teoretické části aplikovány k návrhu a vývoji výstupní herní aplikace. Na základě rešerše byl také implementován systém pro načítání a ukládání hry a opětovně použitelný systém pro adaptivní hudbu. Následně bylo navrženo a provedeno uživatelské testování hry za účelem objevení programových chyb a celkového dojmu ze hry. Poznatky získané z uživatelského testování byly použity k opravě chyb a vylepšení hry. Hra *En attendant ilkeätä noitaa: Čekání na černokněžníka* byla testery hodnocena pozitivně.

Klíčová slova: Godot, herní engine, herní vývoj, objektově orientované programování, aplikace, videohra, hra, 3D, adaptivní hudba

Game application development in the Godot game engine

Abstract

The focus of this bachelor’s thesis is development of a 3D action video game in the open-source game engine Godot.

In the theoretical part, design and development methods of object-oriented programming applicable to Godot application development are described first, followed by code and user testing methods. Afterwards, the Godot engine itself, its structure, and important features are described. The following chapters focus on adaptive music techniques in video games, data serialisation for saving and loading game state, and the principles of action video games in general.

In the practical part, the knowledge gathered from the theoretical part was applied to design and develop the output game application. Additionally, a save/load system and a reusable adaptive music system were implemented based on the preceding research. Following this, user testing of the game was planned and executed to discover the application's programming errors and overall impression of the game. Results from the user testing were used to fix found errors and to improve the game. The game *En attendant ilkeätä noitaa: Čekání na černokněžníka* (“Waiting for the Witch”) was well received by the testing group.

Keywords: Godot, game engine, game development, object-oriented programming, application, videogame, game, 3D, adaptive music

Obsah

1 Úvod 12

2 Cíl práce a metodika 13

2.1 Cíl práce 13

2.2 Metodika 13

3 Teoretická východiska 14

3.1 Principy a metodika objektově orientovaného programování 14

3.1.1 Dědění a kompozice 14

3.1.2 Pravidla SOLID 16

3.1.3 Návrhové vzory Gang of Four 18

3.1.4 Typová kontrola 22

3.1.5 Diagramy UML 24

3.1.6 Antivzory a pachy kódu 27

3.2 Testování 27

3.2.1 Testování kódu 28

3.2.2 Uživatelské testování 28

3.3 Herní engine 30

3.3.1 Rozdělení 30

3.4 Engine Godot 30

3.4.1 Struktura hry 31

3.4.2 Skriptování 33

3.4.3 Vybrané uzly a objekty 35

3.4.4 Datové formáty 36

3.5 Adaptivní hudba ve hrách 37

3.5.1 Neinteraktivní hudba 37

3.5.2 Horizontální přeskládání (horizontal resequencing) 37

3.5.3 Vertikální míchání (vertical remixing) 38

3.5.4 Pokročilé metody 39

3.6 Serializace dat 39

3.6.1 Nativní postupy v Godotu 39

3.6.2 Serializace referencí a vnořených objektů 40

3.6.3 Zpětná kompatibilita 41

3.7 Akční hra 41

4 Praktická část práce ... 43
4.1 Návrh hry ... 43
4.1.1 Herní assety ... 44
4.2 Struktura hlavní scény ... 45
4.3 Programování hry ... 46
4.3.1 Komponentová tvorba a signálová komunikace ... 46
4.3.2 Využití rozhraní ... 47
4.3.3 Použití návrhových vzorů ... 48
4.4 Systém adaptivní hudby ... 56
4.4.1 Návrh ... 56
4.4.2 Implementace ... 57
4.4.3 Použití ... 58
4.5 Serializace dat ... 59
4.5.1 Formát dat ... 59
4.5.2 Rozhraní a implementace ... 60
4.5.3 Zajištění bezpečnosti ... 61
4.6 Uživatelské testování ... 61
4.6.1 Průběh testování ... 61
4.6.2 Zjištěné technické chyby a jejich oprava ... 61
5 Výsledky a diskuse ... 64
6 Závěr ... 66
7 Seznam použitých zdrojů ... 68
8 Seznam obrázků, tabulek, grafů a zkratek ... 71
8.1 Seznam obrázků ... 71
8.2 Seznam zkratek ... 71
Přílohy ... 72

1 Úvod

Vývoj interaktivních a herních aplikací je populárním tématem, které zahrnuje nejen tvorbu videoher, ale také softwarové produkty určené pro vzdělávání a kulturu. Samotný vývoj takové aplikace je však složitý a od tvůrce vyžaduje znalosti programování a softwarového návrhářství, i při použití dostupných tzv. herních enginů, které vývoj značně usnadňují. Mezi ně patří open-source projekt Godot, určený pro tvorbu 2D a 3D her a aplikací. Teoretická část této práce se tak bude zabývat jako první obecnými vývojovými metodikami objektově orientovaného programování a návrhu a posléze v ní bude představen herní engine Godot a jeho funkce. Na základě těchto poznatků bude v rámci praktické části práce vytvořena výstupní herní aplikace.

Důležitou součástí her je také jejich hudební podkres za účelem navození požadované atmosféry a interaktivita her k tomuto potenciálu značně přispívá. Součástí teoretické části práce tak bude představení metod pro adaptivní hudbu ve hrách, na základě čehož bude v praktické části navržen a vyvinut opětovně použitelný systém pro adaptivní hudbu, jenž bude využit ve výstupní herní aplikaci.

Jakožto software používaný lidmi, i hry těží z uživatelského testování, které umožňuje nalézt chyby a nedostatky v programu a pomoci je tak opravit. Proto budou v teoretické části popsány metody uživatelského testování, které bude v praktické části následně provedeno na výstupní herní aplikaci.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této bakalářské práce je vyvinutí 3D akční herní aplikace v herním enginu Godot pro zjištění aplikovatelnosti vhodných návrhových a testovacích metodik objektově orientovaného programování v tomto prostředí.

Dílčími cíli práce jsou návrh a implementace opětovně použitelného systému adaptivní hudby v rámci aplikace a provedení uživatelského testování a zpracování jeho výsledků do finální verze aplikace.

2.2 Metodika

V teoretické části jsou popsány návrhové a testovací metodiky objektově orientovaného programování, jmenovitě návrhové vzory a zásady SOLID. Jsou zde popsány postupy platné pro všeobecný vývoj a uživatelské testování aplikací. Je zde představen herní engine Godot, jeho struktura a funkce a jejich souvislost s předešlými návrhovými vzory. Na konec jsou popsány techniky pro adaptivní hudbu ve hrách, serializaci dat v enginu Godot a principy akčních her obecně.

V praktické části jsou poznatky z teoretické části aplikovány k navržení a vývoji výsledné 3D akční herní aplikace pro platformu Microsoft Windows včetně opětovně použitelného systému adaptivní hudby, jenž je ve hře využit, a systému pro načítání a ukládání herních pozic. Následně je realizováno uživatelské testování metodou ad-hoc za účelem objevení programových chyb. Na základě výsledků testování je aplikace upravena.

3 Teoretická východiska

3.1 Principy a metodika objektově orientovaného programování

Moderní herní enginy v hojné míře využívají objektově orientovaných programovacích (OOP) a skriptovacích jazyků: Unity disponuje jazykem C#, Unreal Engine využívá C++, Godot nabízí jak vlastní GDScript, tak C#. Znalost principů a vhodných metod OOP je proto důležitá a tyto postupy jsou aplikovatelné napříč objektově orientovanými jazyky.

Základem kvalitního softwaru je jeho návrh: Ten se může zpočátku práce na projektu jevit jako zdlouhavost, která zdržuje před „skutečným“ vývojem, pro projekty jiného než drobného rozsahu je však důležitý – vhodný návrh umožňuje snadnější přidávání programových funkcí a údržbu programu za cenu zprvu pomalejšího vývoje (Fowler, 2007).

Fowler (2019) dále popisuje pojem „technologický dluh“ (*„technical debt“*), kterým rozděluje komplexitu softwarového systému na tu nezbytně nutnou a tu, která je nadbytečná a činí systém obtížný na pochopení a která se časem akumuluje (*„cruft*“, volně přeloženo jako šrot). Metaforickým úrokem je zde čas, potřebný k implementaci nové funkce, který se s narůstajícím „šrotem“ prodlužuje. Náprava je v takovém stádiu nákladná, ne-li prakticky nemožná. Není-li tedy tvořen pouze krátkodobý projekt, je důležité aplikovat vhodné metody návrhu již od začátku.

3.1.1 Dědění a kompozice

Dědění a kompozice jsou dvěma základními metodami pro opětovné použití komponent v objektovém programování – právě opětovná použitelnost, flexibilita a modularita komponent jsou při navrhování stěžejní. Existují debaty o vhodnosti obou postupů, především zda není dědění nevhodně nadužívané. Podle Gammy et al. (1994) návrháři často využívají dědění tam, kde by kompozice umožnila větší flexibilitu a jednoduchost, Tempero et al. (2013) naopak zjistili, že ačkoliv je dědění hojně využívané, je jeho použití většinou oprávněné.

Děděním jsou vytvářeny nové třídy, kopírující obsah jiné třídy, s přidanými a modifikovanými daty a chováním. Obsah rodičovské třídy je takto obvykle pro dědící třídu viditelný, proto je tento přístup občas nazýván „*white-box reuse*“ (Gamma et al., 1994).

Při kompozice jsou objekty skládány pospolu jakožto členové nově vytvářené třídy. S těmito objekty je komunikováno stejným způsobem, jako s cizími, a jejich interní struktura

není viditelná. Abstrakce a zapouzdření jsou tím lépe a snadněji zachovány. V kontrastu k dědění je proto kompozice nazývána „*black-box reuse*“ – znovupoužití á la černá skříňka. Členské objekty musí mít vhodně definovaná rozhraní, navazující na principy objektově orientovaného návrhu, jak je uvádí Gamma et al. (1994):

„*Programujte vůči rozhraním, ne implementacím.*[1]“ (překlad vlastní)
„*Preferujte kompozici objektů oproti dědění tříd.*[2]“ (překlad vlastní)

Podle Gammy et al. (1994) by mělo být teoreticky možné vytvořit jakoukoliv funkčnost použitím existujících komponent v kompozitech, nicméně v praxi toto obvyklé není. Dědění je pro vytváření nových komponent, vycházejících z předchozích, jednodušší. Oba přístupy proto mají svá opodstatnění.

Úzce související je zde patero principiálních zásad zvané SOLID (Martin, 2003) (více viz kap. 3.1.2), z nichž zmíněn je zde princip obrácení závislostí (DIP): Závislosti by měly vždy směřovat od konkrétní implementace k abstraktnímu rozhraní, nikoliv obráceně; zjednodušeně a extrémně Martin (2003) uvádí: „*Závisejte na abstrakcích,*[3]“ tudíž vůbec nevytvářet závislosti na ne-abstraktních třídách. Studie hierarchií tříd jedenácti open source projektů (Stevenson a Wood, 2018), však zjistila, že pouze 15 % z více než 40 tisíc zkoumaných tříd je v nich abstraktních. Při přísném dodržování DIP by tento podíl byl nejspíš větší.

Šířka a hloubka dědění

Šířka dědění označuje, kolik tříd dědí z právě jedné třídy; např. pokud z třídy `Object` dědí třídy `Player`, `NonPlayer` a `Decoration`, šířka je 3. **Hloubka** naopak označuje, kolik úrovní dědění se nachází mezi základní třídou (která ze žádné nedědí) a třídou na nejnižší úrovni; např. pokud z objektu `Object` dědí třída `Actor`, ze které dědí třída `NonPlayer`, ze které dědí třídy `Enemy` a `Friend`, hloubka je 3 (Microsoft Corporation, 2022b).

Riel (1996) uvádí, že ačkoliv by **teoreticky** hierarchie dědictví měly být hluboké, protože každá nová třída si může přivlastnit více existujících prvků a jemně přidávat další abstrakce, ukázaly se **v praxi** takové architektury být nevhodné: Jsou málo přehledné a jelikož se zde pravděpodobně dědí větší množství metod, je obtížně předpokládat jejich

[1] „*Program to an interface, not an implementation.*“
[2] „*Favor object composition over class inheritance.*“
[3] „*Depend on abstractions.*“ (v textu překlad vlastní)

chování. Na druhou stranu je zde však větší potenciál pro využití děděných metod. Jak Riel (1996), tak dokumentace k IDE Visual Studio (Microsoft Corporation, 2022b), doporučují dalece nepřekračovat hloubku šesti tříd, což dle Riela souvisí s množstvím věcí, které je běžný člověk schopen uchovat v krátkodobé paměti.

Stevenson a Wood (2018) ve své studii zjistili, že z 80 % tříd bylo děděno pouze jednou nebo dvakrát, indikující malou šířku. Každý zkoumaný projekt však obsahoval třídy, ze kterých bylo děděno pětkrát a vícekrát, většina obsahovala osmnáct a více potomků a nejvyšší nalezený počet potomků byl 172. 90 % tříd bylo děděno do hloubky jedna nebo dva, 98 % nejvýše do hloubky čtyři. Největší hloubka dědictví byla deset, nalezená však ve zdroji pro IDE Eclipse. Obecně je tak v současnosti trendem nevytvářet komplexní dědictví, pokud to je možné.

Provázanost

Spolu souvisejícími termíny jsou provázanost (*coupling*) a soudržnost (*cohesion*). Gautam (2023) je popisuje takto: **Soudržnost** ukazuje míru spolupráce prvků v rámci jednoho objektu na jednom účelu. V zásadě je chtěná vysoká míra soudržnosti, neboť značí vhodné zaměření tříd – opakem by byla třída s mnoha povinnostmi. Naopak **provázanost** ukazuje míru vzájemné závislosti mezi oddělenými objekty. Obecně je vhodné minimalizovat provázanost, čímž je umožněna nezávislost komponent a tudíž jejich snazší opětovná použitelnost. Cílem vývojových metodik je zvýšení soudržnosti a snížení provázanosti komponent.

3.1.2 Pravidla SOLID

SOLID je patero pravidel objektově orientovaného návrhu zformulovaných Robertem Martinem (2000) k zamezení tvorby obtížně modifikovatelných a křehkých (tj. náchylných na vznik chyb při modifikaci) návrhů softwaru.

Single Responsibility Principle – Princip jedné odpovědnosti

Martin (2003) definuje odpovědnost jako „důvod ke změně“ a stanovuje: „*Třída by měla mít pouze jeden důvod ke změně.*[1]“ Například třída spravující síťovou komunikaci by neměla přímo implementovat jak navázání a správu spojení a zároveň odesílání dat. Místo toho by měla být rozdělena – jinak vzniká provázanost mezi odpovědnostmi.

[1] „*A class should have only one reason to change.*“ (v textu překlad vlastní)

Jonáš (2012) vysvětluje jednodušeji: „*Každá třída by měla mít zodpovědnost za právě jednu věc, která by měla být jasně vystižena jejím názvem. Pokud nelze zodpovědnost třídy popsat jednou jednoduchou větou bez použití spojky ‚a‘, pravděpodobně porušuje princip jedné zodpovědnosti*.“

Open Closed Principle – Princip otevřenosti a uzavřenosti

Martin (2000) uvádí, že pokud je potřeba upravit funkčnost komponent kódu, neměla by být komponenta upravována, nýbrž by mělo stačit nový kód přidávat. Typickým způsobem je namísto řady podmínkových výrazů, rozhodující chování objektu dle datového typu vstupu, použití rozhraní a polymorfismu. Riel (1996) taktéž uvádí, že explicitní kontrola typu objektu je obvykle chybným postupem.

Liskov Substitution Principle - Liskovové princip zaměnitelnosti

Dědící třídy musí být schopné plnohodnotně nahradit jejich rodičovské třídy v metodách, kde je očekávána rodičovská třída (Martin, 2000). Pokud metoda, očekávající instanci nadřazené třídy, musí provádět typovou kontrolu, o jakou specifickou třídu se jedná, jedná se o pach kódu a pravděpodobné porušení principu zaměnitelnosti – podřazená třída by totiž pravděpodobně neměla být podřazenou třídou, protože de facto neplatí vztah „A je typu B“ (Metz 2009).

Porušení této zásady je často objeveno příliš pozdě a místo zdlouhavé kompletní úpravy kódu je pak obvykle v potřebných místech přidána podmínka, kontrolující vstupní typ, čímž je dodatečně porušen princip otevřenosti a uzavřenosti (Martin, 2000).

Interface Segregation Principle – Princip oddělení rozhraní

Podobně, jako by třída neměla plnit více funkcí dohromady, či se stát „božskou třídou“ (viz kap. 3.1.6), by třídy měly záviset pouze na těch rozhraních, která používají, tj. mělo by být preferováno více specifických rozhraní než jedno všeobecné (Martin, 2000). Důležité je však nepřejít v opačný extrém a nevytvářet nová rozhraní pro každou třídu (Jonáš, 2012).

Dependency Inversion Principle – Princip obrácení závislostí

Jak bylo popsáno v kap. 3.1.1, není vhodné vytvářet závislosti na konkrétních implementacích, ale na abstraktních rozhraních. Pokud je potřeba vyměnit určitou implementaci nebo přidat novou k výběru, použití rozhraní zásadně snižuje potřebnou refaktorizaci kódu. Jako příklad porušení tohoto pravidla Jonáš (2012) uvádí výkonný kód závisející na uživatelském rozhraní. Pokud by stejná funkčnost měla být adaptována třeba

do podoby webové služby či jiné formy bez UI, nastává zde problém: Výkonný kód je bez UI nepoužitelný a je potřeba jej složitě oddělovat.

3.1.3 Návrhové vzory Gang of Four

Návrhové vzory zdaleka nejsou doménou pouze objektově orientovaného programování – jejich princip byl představen v kontextu architektury a urbanistiky Alexandrem et al. (1977): Tyto vzory popisují často se objevující problémy a jejich řešení, která lze opakovaně používat v mnoha různých situacích.

Stěžejním dílem je zde kniha *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., 1994)[1], ve které je popsána struktura, účel a příklady použití 23 návrhových vzorů. Nejedná se o předem použitelné části kódu, ale o koncepty a archetypy různých postupů, které již byly aplikovány dříve, nikoliv však nutně formálně popsány, a které je možno aplikovat, přizpůsobovat a kombinovat podle potřeby. Zběžně popsáno je v této práci jen několik.

Tyto vzory jsou aplikovatelné pro všeobecné řešení problémů v OOP včetně herního vývoje: Kounoukla et al. (2016) zkoumali vhodnost použití návrhových vzorů GoF k implementaci herních mechanik, neboli herních návrhových vzorů. Jejich výsledky ukázaly, že je toto možné a vhodné a autoři díky tomu mj. očekávají zvýšení udržovatelnosti herních projektů.

Návrhové vzory nejsou odpovědí na všechny problémy: Jejich implementace zvyšuje komplexitu kódu jak z hlediska výpočetního výkonu, tak z hlediska čitelnosti, jak uvádí Gamma et al. (1994) a dodávají: „*Návrhový vzor by měl být aplikován pouze v případě, kdy je jím poskytnutá flexibilita opravdu potřeba.*[2]“ (překlad vlastní)

Observer – pozorovatel

Vzor pozorovatel pomáhá zabránit provázání tříd dohromady v případě, kdy je třeba z jedné třídy něco oznámit druhé, ale není žádoucí potřeba znát konkrétní příjemce, nebo když změna jednoho objektu vyžaduje změnu předem neznámého počtu druhých. Původní implementace sestává ze dvou základních tříd (Gamma et al., 1994):

- **Pozorovatel**, který obsahuje metodu pro příjem zprávy. Zpráva obvykle obsahuje argumenty, často zapouzdřené ve vlastní třídě.

[1] Čtyři autoři, odtud Gang of Four (GoF) – „gang čtyř“.
[2] „*A design pattern should only be applied when the flexibility it affords is actually needed.*“

- **Subjekt**, který obsahuje pole pozorovatelů, privátní metodu pro odeslání zprávy všem pozorovatelům v poli a veřejné metody pro přidání a odebrání pozorovatelů z pole.

Tímto způsobem nejsou zprávy předány jednotlivým pozorovatelům, ale spíše „vysílány“. Odesílané zprávy nejsou vázáné na konkrétní pozorovatele, ti naopak obdrží všechny zprávy a mohou se zachovat podle jejich obsahu (Gamma et al., 1994).

Nystrom (2014) doporučuje nikoliv dědit konkrétní implementace subjektů ze základní třídy, nýbrž kompozicí vložit subjekt do třídy, která na něj posléze odkazuje. Dále navrhuje používat oddělené subjekty pro jednotlivé druhy událostí. Rozlišuje tak systémy s **pozorovateli** (sledující objekty vysílající různé zprávy) a systémy **událostní** (sledující jednotlivé události – events).

State – stav

Pokud se má objekt, **kontext,** nacházet v jednom z více **stavů**, které mění jeho chování, a mezi kterými lze podmíněně přecházet, lze použít tento vzor (Gamma et al., 1994); objekt se pak v podstatě chová jako konečný automat. Každý stav je vyjmut do vlastní třídy a na aktivní stav s jeho logikou a daty je odkazováno členskou proměnnou v kontextu.

Gamma et al. (1994) uvádí vzor na příkladu stavů probíhajícího síťového spojení, které může být ve stavu navázaném nebo uzavřeném, což mění chování metod pro navázání, potvrzení a ukončení spojení. Jedním možným postupem je veškerý stav řešit v jedné metodě, kdy stav (daný např. enumerací) určuje chování objektu pomocí podmínek v potřebných místech. Takový kód se však s narůstajícím počtem vyžadovaných funkcí a stavů rychle stává špatně udržovatelným. Diskrétní stavové objekty jsou přehlednějším a snadněji rozšiřitelným způsobem.

Stav je dle Nystroma (2014) velice častým vzorem v herním vývoji; typickým příkladem použití je pohyblivá postava v herním světě, disponující stavy pro stání na zemi a ve vzduchu: Pokud se postava nachází ve vzduchu, požadavek na skok by neměl být úspěšný a naopak by měla být na postavu aplikována gravitace. Další obvyklou oblastí využití stavů je umělá inteligence nehratelných postav, které přecházejí např. mezi stavy klidu, pronásledováním a hledáním hráče, nebo útočením na něj. Stavové automaty v herním AI byly popularizovány hrami jako Doom nebo Half-Life, počátky jejich využití v této oblasti však sahají až do 80. let minulého století (Thompson, 2023).

Nevýhodou automatů je podle Nystroma (2014) fakt, že se objekt může v jednu chvíli nacházet pouze v jednom stavu. Pokud je potřeba kombinovat několik typů stavů (např. vybrat ze stavů pro stání, chůzi, běh a skok postavy **a zároveň** stavů pro klidový a útočící stav), u běžných konečných automatů by musely být definovány stavy pro každou možnou kombinaci. Jako řešení je uvedeno použití více stavových referencí najednou, tzv. **souběžné** konečné automaty.

Dalšími rozšířeními jsou **hierarchické** stavové automaty, také známé jako **vnořené**, kdy podstavy (**substates**) dědí z nadstavů (**superstates**): Stavový automat tak sám funguje jako stav v rámci většího automatu (Svensson, 2021). **Zásobníkové** automaty přidávají vedle stavů také paměť: Při přechodu do nového stavu je ten přidán na zásobník; při odchodu je z něj odebrán a automat se přesouvá do předešlého stavu dle zásobníku (Nystrom, 2014).

Decorator – dekorátor

Dekorátor podle Gammy et al. (1994) umožňuje dynamicky přidělovat objektům nová chování a ta kombinovat. Jednoduchým, avšak nevhodným, postupem by bylo využít dědění a vytvořit podřízenou třídu s novým chováním. Pokud by však bylo zapotřebí stejná rozšíření přidělit více třídám nebo kombinovat více rozšíření dohromady, vzniká silně nepřehledná hierarchie se značnou duplicitou. Dekorátory umožňují tvorbu malých, samostatných objektů, kterými je možno za běhu programu hlavní objekt „obalovat“[1] a přidávat mu funkčnost.

Základní strukturu Gamma et al. (1994) popisují takto: Vzor sestává z komponenty (resp. rozhraní a implementace) a dekorátoru, který implementuje to stejné rozhraní ale zároveň obsahuje členskou proměnnou – referenci na instanci rozhraní komponenty. Potřebné metody jsou volány na dekorátoru: Ten provede svou činnost a následně zavolá stejnou metodu v instanci, kterou obsahuje.

Gamma et al. (1994) jako příklad uvádí přidání posuvníku nebo rámečku k různým prvkům grafického rozhraní, Refactoring Guru (2014) naopak vzor vysvětluje na notifikačním systému, který je možno dekorovat různými implementacemi – odesláním SMS, e-mailu apod., nebo na systému ukládání a načítání dat, který může být dekorován prvkem pro (de)šifrování či (de)kompresi dat.

[1] I proto je alternativně tento vzor nazýván *wrapper*, „obalovač“ (Gamma et al., 1994).

Bugayenko (2015) rozlišuje princip horizontálního a vertikálního dekorování. Zatímco vertikální dekorování odpovídá klasickému vzoru popsaném výše, kdy další funkcionalita vzniká vnořením objektů do sebe, horizontální dekorování využívá pole metod, skrz které je iterováno. Prvotní implementace vertikálního dekorování je jednodušší, ale se zvyšujícím se počtem vnoření může být vhodné použít druhý přístup.

Singleton – jedináček

Důvodem použití tohoto vzoru je potřeba globálního přístupu k instancované třídě, u níž je navíc potřeba možnosti přístupu právě k jedné instanci; sestává z jedné třídy, která obsahuje jednu statickou privátní instanci sebe sama a statickou veřejnou metodu, která vrací referenci právě na onu instanci (Gamma et al., 1994).

Práce se singleton objektem je podobná, jako se statickou třídou, ale s několika výhodami (Gamma et al., 1994): Programátor sám specifikuje, kdy a zda vůbec má být třída instancována; oproti tomu statické třídy jsou inicializovány při spuštění programu a toto pořadí není zpravidla možné ovlivnit. Singleton je však podle Nystroma (2014) často kritizovaným návrhovým vzorem, jelikož je jeho použití vnímáno jako hrubé řešení určitého problému, které nesnižuje (a naopak zvyšuje) provázanost komponent a ve skutečnosti neřeší problémy vyplývající z použití globálních objektů.

Godot implementuje tento vzor ve funkci Autoload, která umožňuje globální přístup k vybraným třídám (Linietsky et al., 2014).

Vzory pozorovatel a jedináček lze kombinovat v případě, že je složité zprostředkovat prvotní referenci na subjekt v pozorovateli a přidat jej jako pozorovatele, uvádí Lovato (2021): Řešením je vytvořit jedináčka, který také funguje jako subjekt – skutečný subjekt volá vysílací metody v něm a pozorovatel je schopen se snadno přidat k příjemcům. Vzniklý konstrukt Lovato nazývá **event bus** (dosl. „sběrnice událostí“). Na jednu stranu je tak dále zamezeno provázání mezi zdrojovým a cílovým objektem, nicméně hrozí vznik objektu, který se stará o veškerou komunikaci mezi objekty; doporučeno je proto tento vzor použít pouze v případě nutnosti.

Flyweight – muší váha

Vzor flyweight slouží k úspoře paměti v případě více instancí objektů stejného typu – částic v částicových simulacích či třeba znaků v grafických rozhraních. Gamma et al. (1994)

uvádí rozdělení jedné třídy na vnitřní a vnější **stav**: **Vnitřní** stav je společný pro více objektů stejného typu; **vnější** stav je pro každý objekt unikátní.

Nystrom (2014) jako příklad uvádí vykreslování lesa s mnoha stromy: Uvažujme třídu pro objekt stromu ve 3D hře: Tato třída obsahuje transformaci (pozici, rotaci, škálování) stromu spolu s 3D modelem a přidělenými texturami. Pokud je pro každý strom použit identický 3D model a textury, je zbytečné je ukládat v každém stromu zvlášť – především textury, které zabírají velké množství paměti. Tyto sdílené atributy je proto možné vyjmout do oddělené flyweight třídy, na kterou je všemi instancemi stromů odkazováno. Objekt obsahující model a textury je pak vnitřní stav, objekty odkazující na něj a obsahující unikátní transformace stromů jsou stavem vnějším.

V enginu Godot lze za flyweight považovat třídu `Resource` (zdroje), a to z hlediska jak zacházení s ní, tak jejího účelu. Vnějším stavem jsou pak v Godotu uzly (`Node`). Dle dokumentace (Linietsky et al., 2014): Při načítání souborů z úložiště funkcí `load` je kontrolováno, zda již nejsou tyto zdroje uloženy v paměti. Příkladem rozdělení stavů budiž uzel `MeshInstance3D`, objekt umístěný ve 3D prostoru herního světa, kterému je přidělen zdroj `Mesh` (3D model). Zdroje však mohou plnit role obou stavů: Zdroji `Mesh` je přidělen zdroj `Material`, jehož dílčí součásti přijímají zdroje `Texture2D`.

3.1.4 Typová kontrola

Způsoby typování a typové kontroly jsou několika rozlišujícími znaky různých programovacích jazyků. Při přechodu mezi těmito jazyky (např. spíše statickým C# a dynamickým GDScriptem) je proto třeba o tomto rozdělení mít ponětí. Křivánek (2004) uvádí: „*Typ definuje množinu hodnot, kterých může proměnná nabývat, a množinu operací, které mohou být s danou proměnnou provedeny.*“ Typová chyba pak označuje použití operace na typ, který ji nepodporuje (Siek, 2014).

Statické a dynamické typování

Pavey (2019) uvádí: Ve **staticky** typovaných jazycích je nutné pro každou proměnnou deklarovat její typ, který pak není možné za běhu programu měnit. Výhodou je vyšší bezpečnost (vůči chybám, ne zlým aktérům), jelikož špatné nakládání s typy lze odhalit již při kompilaci programu (*compile-time*). V případě **dynamicky** typovaných jazyků se proměnné při deklaraci nezavazují k určitému typu – ten se může měnit – a typová kontrola

probíhá za běhu programu (*runtime*). To umožňuje větší flexibilitu a úspornost při psaní kódu, který je také snadněji upravitelný.

Oba přístupy jsou v jazycích obvykle do určité míry kombinovány, jelikož není možné se spoléhat pouze na statickou typovou kontrolu: Jako příklad Křivánek (2004) uvádí deklaraci proměnné typu s horním limitem za použití nadlimitní konstanty: Toto statická kontrola zachytí. Přičtení hodnoty k deklarované proměnné za běhu programu, která způsobí přetečení, již však nezachytí. Siek (2014) uvádí, že není možné veškeré typové chyby zachytit jen statickou typovou kontrolou – jedná se pak o variantu známého problému zastavení (*halting problem*), který je neřešitelný. Dále popisuje další obvyklé výhody těchto přístupů: Statické typování umožňuje vyšší rychlost výsledného programu, jelikož je více kontrol prováděno již při kompilaci. Dynamické typování naopak zjednodušuje řešení situací, kdy hodnotový typ závisí na informaci známé pouze za běhu programu.

Silné a slabé typování

Typování je dále děleno na silné a slabé (Pavey, 2019): Striktně **silně** typované jazyky nedisponují implicitním převodem typů; veškeré přetypování musí být explicitně uvedeno. **Slabě** typované jazyky se pokouší podle potřeby datové typy převádět. Kupříkladu zatímco Python nedovolí implicitně provést operaci sčítání mezi číslem a řetězcem, JavaScript zde automaticky převede číslo na textový řetězec a provede konkatenaci.

Tyto přístupy mohou být taktéž kombinovány; příkladem budiž C#, který dovoluje implicitní konverzi (Microsoft Corporation, 2022a), pokud při ní nehrozí ztráta informace: Datový typ `int` (celé číslo) je možné transparentně použít v místech, které očekávají `double` (číslo s plovoucí desetinnou čárkou), obráceně toto však možné není.

Duck typing

Duck typing, doslova „*kachní typování*“, úzce souvisí s dynamickým typováním. Při zkoumání typů objektů je, podle O’Callaghana (2021), považováno za důležité pouze to, zda daný objekt implementuje požadovanou metodu, zatímco o jeho dědictví a implementovaných rozhraních není vůbec uvažováno, což je ukázáno na následujícím příkladu, zde převeden do GDScriptu:

```
var a = "aeiou"      # proměnná typu String
var b = ["a", "o"]   # proměnná typu Array
for i in [a, b]:
      print(i.find("o")) # Vypíše do konzole 3 a 1.
```

Při volání metody `find()` zde není nijak brán v potaz typ iterovaných proměnných, nýbrž pouze, zda jejich objekty danou metodu implementují.

Milojkovic et al. (2017), kteří duck typing také nazývají křížově hierarchický polymorfismus (*cross-hierarchy polymorphism*), dokonce považují rozhraní ve staticky typovaných jazycích za jeho simulaci. Zároveň upozorňují na nevýhody duck typingu: roztříštění funkcí v kódu a jeho výslednou nižší čitelnost. Při zkoumání kódu je často potřeba uvažovat o možných typech, které mohou být určitou částí kódu používány; zatímco statické typování toto usnadňuje, duck typing je zde naopak spíše přítěží.

Graduální typování

Graduální typování umožňuje využívat jak statického, tak dynamického typování dle potřeby programátora, obměňovat je a využívat obojích způsobů pospolu; jednotlivá dobrovolná značení typů proměnných se pak nazývají anotace (Siek, 2014). Deklarace proměnných z minulého příkladu by vypadala za využití graduálního typování takto:

```
var a: String        = "aeiou"                # striktně textový řetězec (string)
var b: Array[String] = ["a", "o"]             # striktně pole textových řetězců
var c: Array         = ["a", true, 1.5]       # striktně pole libovolných typů
var d               := "bflmpsvz“             # string - použita typová inference
```

3.1.5 Diagramy UML

Unified Modeling Language je sada notací, jejíž cílem je umožnit snadnou a jednotnou komunikaci při návrhu aplikací jak mezi vývojáři, tak klienty; zejména při týmovém vývoji aplikací je takový nástroj důležitý a základní znalost UML je nezbytností (Bell, 2003).

UML sestává ze čtrnácti typů diagramů, které se liší použitím, komplexností a úrovní abstrakce. UML lze použít k nastínění funkčnosti systému, což je vhodné při prvotním návrhu aplikace a při komunikaci s klienty. Při dalším vývoji je UML používán pro plánování technických struktur objektů. UML je také možno použít jako grafický programovací jazyk, který je posléze převeden na kód (Čápka, 2017).

V této práci jsou popsány a použity pouze diagramy tříd a diagramy stavů. Důvodem je zaměření této práce na projekty komparativně malého rozsahu, absence zaměření na požadavky netechnických klientů a naopak vysoká míra použití popsaných diagramů při zpracování výstupní aplikace. Jiné často používané diagramy, tj. diagramy komponent a případů použití (use-case), jsou nad rámec této práce a proto popsány nejsou.

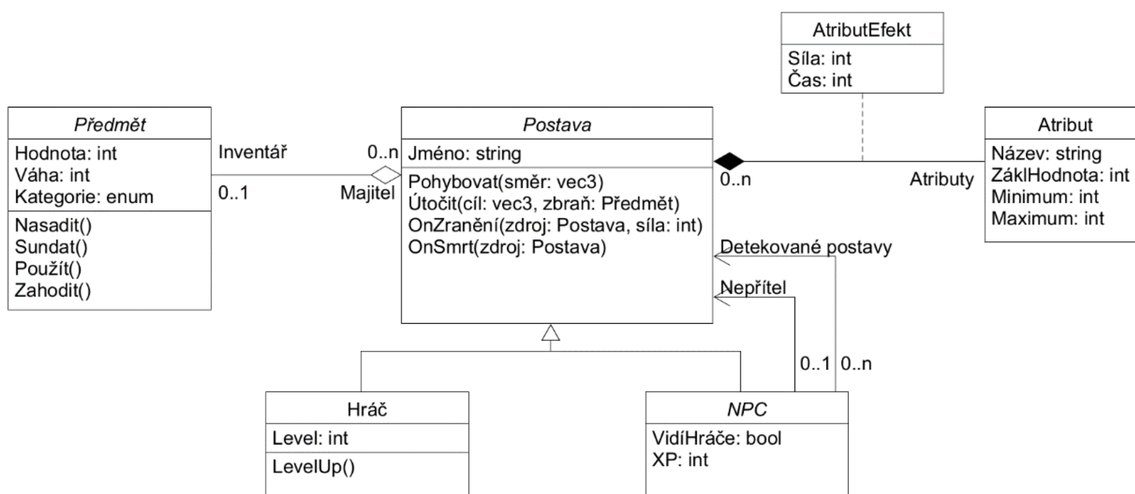
Diagramy tříd

Třídní diagramy zaznamenávají interní strukturu tříd modelovaného systému a vztahy mezi nimi; jsou tak vhodné především pro vývojáře samotné (Bell, 2004). Základním prvkem jsou zde typy (obvykle třídy nebo rozhraní), běžně vyobrazené jako obdélníky se třemi částmi vyhrazenými pro následující (Bell, 2004):

1. **Název třídy:** Může také obsahovat klasifikační klíčové slovo (naznačující např. rozhraní), tzv. *stereotyp*. Název je psán kurzívou pro naznačení abstraktní třídy. Instance tříd jsou podtržené.
2. **Atributy třídy:** Seznam proměnných vlastněných třídou ve formě `název: typ`. Referenční typy (cizí třídy) zde chybí – jsou propojeny s třídou čarou či šipkou.
3. **Operace třídy:** Seznam metod třídy s argumenty a návratovými typy.

Bell (2004) dále popisuje: Atributy a operace mohou být označeny symbolem pro označení **úrovně přístupu**: `+` pro veřejné prvky, `-` pro privátní, `#` pro chráněné (pouze pro dědící třídy) a `~` (tilda) pro prvky dostupné v rámci balíčku (volitelná sdružení souvisejících tříd). Třídy mohou být navzájem propojeny ať už ve smyslu dědění nebo ve smyslu asociace, agregace a kompozice. Toto je graficky rozlišeno. Při druze zmíněném je navíc udáván **název atributu** a **multiplicita**, označující, kolik těchto instancí může být s jedním objektem propojeno či jím vlastněno.

Na obr. 1 je vyobrazen třídní diagram abstraktní herní postavy s inventářem a atributy s efekty, jejíž konkrétními implementacemi je hráč a nehráčská postava.



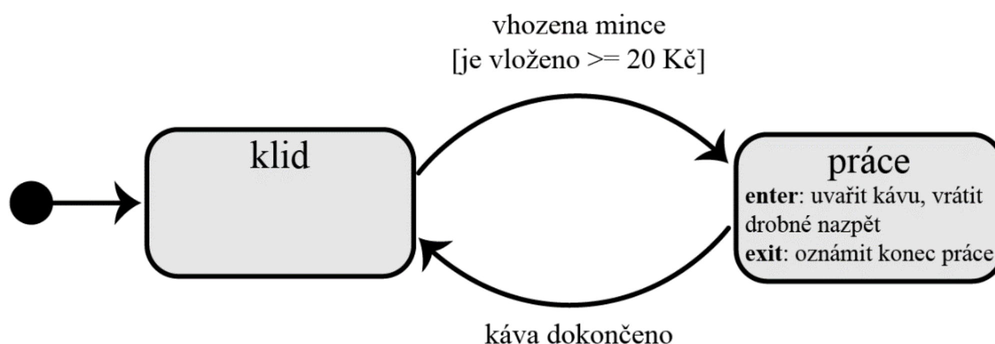
obr. 1 Příklad UML třídního diagramu (zdroj: vlastní).

Diagramy stavů

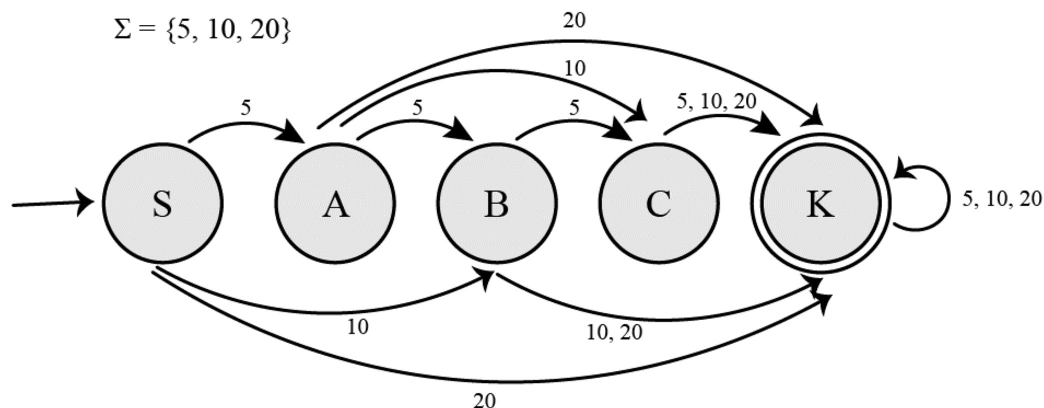
Stavový diagram UML znázorňuje stavy systému a přechody mezi nimi. Podobá se formálním diagramům konečných automatů, je však rozšířený o užitečné prvky jako podmínky přechodu či vnořené stavy s více vstupními body. Naopak zde není potřeba definovat množinu vstupní abecedy, kterou musí každý stav přijímat.

Agile Modeling (2002) popisuje jejich základní strukturu: Vstup do automatu je vyznačen černým kruhem, přechody šipkou. Stav je označen jménem a případně obsahuje akce, včetně těch při vstupu do a výstupu ze stavu. Přechody jsou slovně označeny důvodem a případně je možné v hranatých závorkách uvést podmínku přechodu. Automat je možno ukončit přechodem do kružnice s menším kruhem uvnitř.

Obr. 2 a 3 porovnávají UML stavový diagram a formální diagram konečného automatu fungování jednoduchého automatu na kávu. Diagram UML je vysokoúrovňový a umožňuje jednoduché nastínění požadované funkčnosti pomocí slovního popisu událostí a podmínek. Automat popsaný formálním diagramem byl zjednodušen; přijímá pouze mince o hodnotách 5, 10 a 20 Kč a nevrací, i tak je však oproti UML diagramu zjevně složitější.



obr. 2 UML stavový diagram jednoduchého automatu na kávu (zdroj: vlastní).



obr. 3 Formální diagram konečného automatu na kávu (zdroj: vlastní).

3.1.6 Antivzory a pachy kódu

Antivzory jsou nevhodné techniky, na které je možné narazit nebo je nechtěně při psaní kódu aplikovat a způsobit tak horší kvalitu, udržitelnost a rozšiřitelnost kódu (Codacy, 2023). Jakýkoliv programátor by se jejich psaní měl vyhnout, k tomu je však důležité je znát. Pachy kódu jsou souvisejícím fenoménem, označující případy a vzory v kódu, které nejsou nutně špatné, často však značí antivzor nebo špatně napsaný kód. Mezi obecně známé antivzory patří např. špagetový kód, který je silně provázaný a nestrukturovaný; či božský objekt, třída s mnoha odpovědnostmi a tedy přímé porušení principu jedné odpovědnosti (Parr, 2020).

Magické číslo

Použití neoznačené literální konstanty v kódu (tj. přímé použití hodnoty v prováděném výrazu) je antivzor. Její význam nemusí být zřejmý a v případě, že je potřeba danou hodnotu nahradit za jinou, je potřeba vyhledat veškeré její použití a manuálně ji upravit. Řešením je použití symbolické konstanty (Adico, 2019).

Nadužívání textových řetězců

Častou nevhodnou praktikou je použití textových řetězců (strings) tam, kde by byl vhodnější vlastní objekt nebo jednoduchá enumerace. Joshua Mathews (2022) uvádí často zbytečné množství možných hodnot v rámci stringů a nebezpečí vzniknutí chyby způsobené překlepem (případně nesoulad mezi malými/velkými písmeny). Druze zmíněný problém lze eliminovat použitím konstant, první problém tak však řešen není.

3.2 Testování

Testování softwaru je důležité pro časné odhalení a opravení chyb. Chyby jak technického rázu, tak související s uživatelskou přívětivostí, se mohou vyskytnout snadno. Schultz a Bryant (2016) rozdělují testování softwaru na black-box a white-box testování. Při *black-box* testování nemá tester žádný přístup ke zdrojovému kódu; program se tak chová jako pomyslná černá skříňka. Veškeré metody a závěry vyplývají z pokoušení různých kombinací vstupů v běžící aplikaci. Uživatelské testování spadá do této kategorie. Při *white-box* testování je naopak zdrojový kód přímo využit a je do něj zasahováno: Zdrojem akcí zde nejsou vstupy od uživatele, ale testovací kód přidaný vývojářem.

Kombinací těchto přístupů je *grey-box* testing, kdy tester nemá přímo přístup ke zdrojovému kódu, ale má částečnou znalost vnitřního fungování programu, např. jak

matematicky probíhá výpočet nějaké výstupní hodnoty, či za jakých podmínek je proveden přechod mezi stavy určitého objektu (Collins, 2023).

3.2.1 Testování kódu

Pomocí testování kódu je ověřována správná funkce jednotlivých komponent programu. Základním nástrojem je zde **aserce**, kterou zde popisuje Stotts (2000): Jedná se o kontrolu podmínky, která má být vždy pravdivá. Pokud je aserce nepravdivá, je program považován za nefunkční a je shozen s hláškou o místě vzniku. Programátorovi je tak jasně sděleno, kde se stala chyba. Kompilátor obvykle zahrnuje aserční metody pouze ve vývojovém (*debug*) sestavení programu.

S tímto souvisí návrhová metodika Design by Contract, která metodám stanovuje před-podmínky, post-podmínky a invarianty, neboli podmínky, které musí být pravdivé před vykonáním metody, po vykonání metody a kdykoliv, kdy je objekt klientovi přístupný (Stotts, 2000). DbC lze implementovat jednoduchými asercemi, ale různé jazyky mohou obsahovat funkce pro oddělení asercí od hlavní metody (např. C#/.NET) (Microsoft Corporation, 2021), nebo přímo implementují kontrakty jako základní součást jazyka (např. Eiffel) (Stotts, 2000).

3.2.2 Uživatelské testování

Uživatelské testování (black-box testing) umožňuje objevit chyby v programu stejným způsobem, jako je mohou objevit běžní uživatelé – používáním programu. Testování na uživatelích se může jevit jako nákladné, Nielsen (2000) však prokázal, že počet nově objevených problémů se snižuje se zvyšujícím se počtem testerů a stanovil ideální počet uživatelů při jednom testování na pět osob. Nicméně také podotýká, že pokud existuje několik silně odlišných skupin uživatelů, je zapotřebí více testerů, a to tři z každé kategorie. V případě her je takových skupin mnoho – Schultz a Bryant (2016) mj. definují „*běžné*“ hráče, kteří následují herní instrukce; „*hardcore*“ hráče, používající makra, klávesové zkratky a vlastní vstupní zařízení; „*achievers*“, kteří chtějí dosáhnout všech možných cílů a ocenění pokud možno nejefektivnějším způsobem; nebo „*exploiters*“, kteří se snaží hledat zkratky nebo způsoby, jak úmyslně využít programové a návrhové chyby hry k vlastnímu užitku.

Strukturované testování

Jako jednu z metod, snažící se minimalizovat množství testování při maximalizaci užitku, uvádí Schultz a Bryant (2016) **kombinační testování**: Při něm jsou nejprve stanoveny parametry (např. herní události, nastavení hry, použitý hardware) a jejich hodnoty, které jsou pro každé testování kombinovány. Rozlišovány jsou testy homogenní, kdy jsou porovnávány hodnoty ze stejné kategorie, a heterogenní, kdy jsou tyto kategorie odlišné. Oproti tomu diagramy toku testování (**test flow diagrams**) jsou sestavovány jakožto průběh toho, jak by měla hra fungovat z pohledu hráče: Jejich testování ukazuje, zda nastává očekávané chování či nenastává neočekávané. Zatímco kombinační testování je vhodné pro zkoumání vlivu několika paralelních hodnot na funkčnosti hry, TFD jsou vhodné pro zkoumání časově navazujících událostí – přechodů mezi herními stavy, opakovaných funkcí, či větvení podle voleb hráče.

Při **cleanroom testingu** jsou hledány chyby s pravděpodobností podobnou skutečnému světu a mimikovat skutečné použití produktu; v závislosti na nasbíraných datech, ukazující, jak často jsou různé funkce produktu používány koncovými uživateli, jsou stejně tak často i testovány (Schultz a Bryant, 2016).

Ad hoc testování

Ad hoc testování je nahodilejší než strukturované metody. Tento přístup využívá kuriozity a intuice, kdy hráče při testování často zajímá, co se stane, když provede určitou činnost; jde o metodu, která nejlépe umožňuje přirozeně prozkoumávat hru, nicméně je stále důležité stanovit cíle a omezení daného testování, psát zápisky, nahrávat videozáznam a v případě objevu chyby Schultz a Bryant (2016) doporučují následovat vědeckou metodu:

1. Shromáždit, co bylo před chybou prováděno a co se dělo, znovu se podívat na záznam a zápisky – provést pozorování.
2. Zpracovat informace k zjištění postupu, jak vyvolat chybu – sestavit hypotézu.
3. Předpovědět, že následování těchto kroků vede k chybě.
4. Restartovat počítač a hru a pokusit se chybu zopakovat – ověření hypotézy.
5. Případně opakovat kroky 3 a 4, dokud není hypotéza ověřena.

Playtesting

Schultz a Bryant (2016) při porovnávání playtestingu a ostatních metod srovnávají otázky „Funguje hra?“ a „Funguje hra *dobře*?“. Playtesting se od předchozích typů testování

liší tím, že nezkoumá objektivní funkčnost hry – nezaměřuje se na programové chyby, nýbrž subjektivní fungování hry. Zkoumá, zda je hra vhodně obtížná, zda jsou různé zamýšlené přístupy ke hře navzájem vyrovnané, zda je ovládání nebo průchod uživatelským rozhraní intuitivní, či zda je hra zábavná.

3.3 Herní engine

Vytvářet hru od základu je pro většinu vývojářů z důvodu enormní složitosti takového procesu nerealizovatelné. Herní engine je nástroj pro usnadnění vývoje her pomocí poskytnutí předpřipravených komponent, abstrakcí nízkoúrovňového kódu a obvykle umožnění vývoje hry pro více platforem – PC, herní konzole, mobilní OS – bez potřeby zásadního přepracování vnitřní struktury programu (Abbadi, 2016).

3.3.1 Rozdělení

Abbadi (2016) rozděluje herní enginy na nízkoúrovňové a vysokoúrovňové:

- **Nízkoúrovňové** enginy nabízejí především základní funkce jako časování programové logiky, abstrakce grafických API (Vulkan, DirectX) a správu paměti. Často jsou poskytovány čistě ve formě knihoven a frameworků. Logika hry a nové komponenty enginu jsou obvykle tvořeny běžnými programovacími jazyky.
- **Vysokoúrovňové** enginy dále snižují komplexitu vývoje her a obsahují více opětovně použitelných komponent pro různé účely, např. pokročilé renderovací systémy, vlastní skriptovací jazyky, vlastní implementace umělé inteligence, pathfindingu (hledání nejkratších cest) atd. Tyto enginy také často poskytují své vlastní editory, umožňující tvořit značnou část her za pomocí grafického rozhraní.

3.4 Engine Godot

Godot je svobodný a open-source vysokoúrovňový, multiplatformní herní engine, vyvíjen od roku 2001 a otevřený pro přispěvatele od roku 2014; současná stabilní verze je 4.2 (Linietsky et al., 2014). Major verze 4.0 přinesla mj. podporu mobilních zařízení, grafického API Vulkan (nástupce OpenGL) či nový, vlastní systém fyzikálních simulací (Linietsky et al., 2023).

3.4.1 Struktura hry

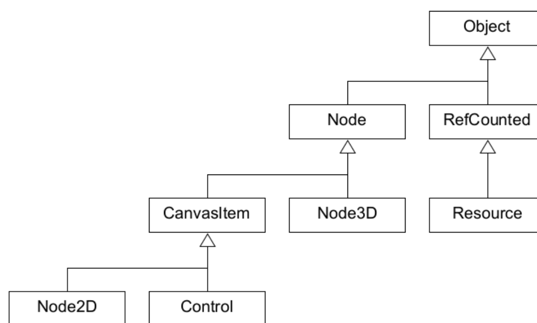
Základní struktura hry v enginu Godot je následující (Linietsky et al., 2014): „*Hra je **stromem uzlů**, které jsou shlukovány do **scén**. Tyto uzly mohou poté být pro vzájemnou komunikaci propojeny pomocí **signálů**.*[1]“ (překlad a zvýraznění vlastní)

Scény a uzly

Dle dokumentace (Linietsky et al., 2014): Uzly (třída `Node` a potomci) jsou základním stavebním kamenem aplikací v Godotu. Je možné využít velké množství předdefinovaných typů, rozdělených do tří hlavních kategorií: 2D, 3D a UI, nebo vytvořit zcela vlastní. Tyto uzly jsou ukládány v hierarchické stromové struktuře uvnitř jedné scény. Scény je možné vložit do ostatních scén, kde jsou umístěny do jejich stromových struktur jako samostatné, uzavřené uzly. Je tak dosaženo znuvupoužitelnosti a zapouzdření komponent.

Hierarchie tříd

Zjednodušená hierarchie objektů v enginu Godot je popsána třídním diagramem na obr. 4. Základní třídou je `Object`. Ke každému objektu je možno pro rozšíření funkcionality, bez potřeby definice nové třídy, přiřadit skript (Linietsky et al., 2014).



obr. 4 Základní hierarchie objektů enginu Godot (Linietsky et al., 2014).

Dokumentace (Linietsky et al., 2014) dále vysvětluje: `Node` mohou být 3D prvky (`Node3D`), 2D prvky (`CanvasItem`), dále děděné na grafická rozhraní (`Control`) a 2D herní prvky (`Node2D`), a mnoho jiných, např. časovače, doplňky editoru, nebo HTTP požadavky. `RefCounted` jsou objekty, u kterých je kontrolován počet referencí na ně v zájmu automatické správy paměti: Pokud není na instanci nijak odkazováno, je zničena. Důležitým typem je zde `Resource`, třída pro serializovatelné, sdílené flyweight objekty. Přímo z `Object` dědí i některé jiné třídy, např. nízkoúrovňové služby (zvané servery) pro správu zvuku, renderingu a fyzikálních simulací nebo komunikaci se systémem a skriptovacími jazyky.

[1] „*In Godot, a game is a tree of nodes that you group together into scenes. You can then wire these nodes so they can communicate using signals.*“

Příkladem uvedeny třídy, týkající se přehrávání zvuku (Linietsky et al., 2014):

- `AudioStream` je `Resource`, obsahující **zvuková data**, obvykle ze souboru.
- `AudioStreamPlayer` je `Node`, zprostředkovávající jednoduché rozhraní pro přehrávání zvuku – **přehrávač**.
    - `AudioStreamPlayer2D` a `3D` umožňují změnu hlasitosti a jiné efekty podle jejich pozice v herním světě.
- **Informace o právě přehrávaném zvuku** jsou uloženy v `RefCounted` třídě `AudioStreamPlayback`, obsahující současnou pozici (čas) přehrávání a zda je přehrávání pozastaveno či zdali má být opakováno.
- Na **nejnižší úrovni** leží `AudioServer`, dědící z `Object`, který vytváří jednotlivé vzorky a odesílá je do výstupního zařízení.

Pokud je požadováno přehrání zvuku, je třeba vytvořit novou instanci `AudioStreamPlayer` a vložit ji do stromu scény, její vlastnosti `stream` přidělit požadovaný `AudioStream` a zavolat metodu `play(time)`; pokud je přehrávač již původní součástí scény, není manuální tvorba instance a zavedení do stromu zapotřebí a zvuk je možno přidělit v grafickém rozhraní editoru, povolení vlastnosti `autoplay` pak umožňuje i spuštění zvuku bez metody `play` (Linietsky et al., 2014).

Skripty

Každému objektu (nejen uzlu) je možno přidělit skript, určený k rozšířený chování objektu a případnému rozšíření třídní hierarchie přidělením názvu třídy. Skripty jsou určeny k rozšíření funkčnosti určité třídy a jejích potomků, např. skript rozšiřující `Node3D` lze přidělit každému 3D uzlu, ne však 2D nebo základním uzlům (Linietsky et al., 2014).

Skupiny

Skupiny (*groups*) slouží k označení různých uzlů ve scéně. Z dokumentace (Linietsky et al., 2014): Jakémukoliv uzlu lze přidělit libovolné množství skupin, jak v editoru, tak ze skriptu. Skupiny jsou označovány textovými řetězci. Výhodou jsou zde metody dostupné ve třídě `SceneTree`, pomocí kterých lze snadno pracovat se všemi uzly ve skupině najednou, například zavolat jim přidělené metody nebo změnit hodnoty vlastností.

Resource

Již bylo zmíněno, že `Resource` je třídou k uchování a serializaci dat, na rozdíl od uzlů, které zprostředkovávají instanční logiku. Soubory načtené z úložiště včetně uložených scén

či skriptů jsou `Resources`, které jsou následně použity uzly ve scéně. Dokumentace (Linietsky et al., 2014) uvádí zvláštní typ zdroje, `PackedScene`, neboli serializovanou scénu, kterou je možno instancovat metodou `instantiate()` a vložit do stromu scény. Vlastní podtřídy zdrojů lze taktéž načítat z úložiště a ukládat. Jako běžný příklad Jennings (2022) uvádí informace o předmětech herního inventáře (název, atributy, grafika), pomocí kterých je možné udávat chování uzlů ve hře a jako zajímavější případ použití uvádí různé události, které je možno opětovně používat, přidělovat a kombinovat, např. při úmrtí vybraného nepřítele.

3.4.2 Skriptování

Godot nabízí několik jazyků pro skriptování a rozšiřování funkcionality enginu. Tím hlavním je GDScript. Podle oficiální ankety z roku 2022 jej jako hlavní používá 80 % vývojářů, využívajících Godot (Linietsky, 2022). Mezi dva další jazyky patří C# s frameworkem .NET a systém GDExtensions pro použití s vícero jazyky; tyto jazyky lze využívat společně, jeden projekt není omezený pouze na jeden (Linietsky et al., 2014).

Základní struktura GDScriptu

Dokumentace (Linietsky et al., 2014) popisuje GDScript jako „*objektově orientovaný, imperativní, graduálně typovaný jazyk. Používá syntaxi založenou na odsazeních, podobnou jazykům jako je Python.*[1]“ Jazyk podporuje mj. třídy, statické a nestatické proměnné a metody, asynchronní metody, atributy tříd, dědictví, přetypování, lambda funkce, automatickou správu paměti a správu událostí pomocí signálů. Mezi zabudované datové typy patří vektory, kvaterniony, transformace, slovníky, nebo volatelné proměnné.

Signály

Signály jsou nativní implementací návrhového vzoru pozorovatel. V základu sestává z následujících prvků (Linietsky et al., 2014):

- Deklarace signálu v třídě odpovídající subjektu:
  - `signal nazev_signalu(args ...)`
  - Název je obvykle uváděn jako sloveso v minulém pádě.
  - Argumenty a jejich množství je volitelné.
- Volaná metoda ve třídě odpovídající pozorovateli:
  - `func nazev_metody(args)`

[1] „*GDScript is a high-level, object-oriented, imperative, and gradually typed programming language built for Godot. It uses an indentation-based syntax similar to languages like Python*.“ (v textu překlad vlastní)

  - Název je obvykle uváděn ve formátu `_on_nazev_signalu`.
- Propojení signálu v subjektu a metody v pozorovateli:
  - `subjekt.nazev_signalu.connect(nazev_metody)`
- Vyslání signálu v subjektu:
  - `nazev_signalu.emit(params ...)`

Mnohé třídy obsahují předdefinované signály, např. `Area3D` (průchodná oblast v systému kolizí) při detekci vstupujícího objektu vysílá signál `body_entered(body)`. Zároveň je možné propojit subjekt a pozorovatele graficky, pokud jsou oba vytvořeny v editoru a součástí stejné scény (Linietsky et al., 2014).

Praktiky při používání signálů

Wilkes (2021) uvádí několik doporučených postupů pro práci se signály:

- Signál nemůže být přímo propojen s uzlem mimo jeho scénu. Je proto možné signály řetězit: Komponenta vyšle signál, ten je zachycen kořenovým uzlem, který poté vyšle vlastní signál.
- Při komunikaci mezi uzly v hierarchii používat přímé volání metod z nadřazeného uzlu k podřízenému a naopak z podřízeného uzlu k nadřízenému vysílat signály: *Volat dolů, signalizovat nahoru.*
  - Důvod: Pokud je jakýkoliv uzel vyjmut z jeho hierarchie do oddělené scény, měl by být stále schopen teoreticky fungovat – nic užitečného sice nedělat, ale nezpůsobit chybu a pád hry.

Ostatní jazyky

Dokumentace uvádí několik alternativ k jazyku GDScript (Linietsky et al., 2014):

- **C#** s frameworkem **.NET** pomocí **Godot API**. Od verze 4.2 platná podpora pro všechny desktopové i mobilní platformy.
  - Vyšší výkon jazyka za cenu dražší komunikace s jádrem enginu a nemožnosti okamžitého projevení změn skriptů za běhu hry.
- **GDExtensions**: Komunikace se sdílenými knihovnami za využití jazyků C, C++ aj. a to bez potřeby opětovné kompilace enginu.
- Vlastní **statické moduly** při úpravách samotného enginu, který je silně modulární.

3.4.3 Vybrané uzly a objekty

Již bylo zmíněno velké množství dostupných předdefinovaných typů uzlů. Popsány jsou zde některé nejčastěji používané v kontextu 3D hry.

Node, Node3D

Před uvedením specifických uzlů je nutné znát běžné funkce základních typů uzlů. Výběr z dokumentace (Linietsky et al., 2014): Třída `Node` obsahuje funkce a vlastnosti související se stromovou strukturou a herní smyčkou hry. Metody `get_children()`, `get_parent()` a `get_node(path)` umožňují získat reference na blízké uzly ve stromu. Virtuální metoda `_ready()` je zavolána při prvotním přidání uzlu do stromové struktury scény (resp. poté, co jsou všechny podřízené uzly taktéž přidány). Metody `_process(delta)` a `_physics_process(delta)` jsou volány neustále (co nejčastěji a v konstantním intervalu) a jsou hlavní „uvažující“ složkou uzlů. Argument `delta` udává čas od jejich posledního volání. Zda jsou tyto metody prováděny určuje enumerace `process_mode` a zda je strom scény ve stavu pozastaven (`paused`).

`Node3D`, dědící z `Node`, přidává především 3D transformaci – vlastnosti o poloze, škálování a rotaci v prostoru spolu s příslušnými metodami; důležitý je zde rozdíl mezi `global_transform`, transformací v globálním prostoru, a `transform`, která je lokální, tj. relativní vůči rodičovskému uzlu (Linietsky et al., 2014).

AnimationPlayer, AnimationTree

Uzel `AnimationPlayer` dle dokumentace (Linietsky et al., 2014) umožňuje animovat parametry ostatních uzlů pomocí předem vytvořených animací, využívajících klíčových snímků, při importu 3D modelu s animacemi je proto tento uzel vytvořen pro správu animací kostí či morfů. Jde o jednoduchý přehrávač animací, který je sám o sobě schopen v jednu chvíli přehrávat jen jednu animaci. K pokročilému míchání animací slouží uzel `AnimationTree`, který umožňuje vytvářet stromovou, stavovou, nebo kombinovanou strukturu animací.

CharacterBody3D

`CharacterBody3D` dědí z `PhysicsBody3D` spolu s `RigidBody3D` a `StaticBody3D`. Tato třída je určena pro objekty, které vyžadují specifické ovládání, a je vhodná jako základ pro objekty herních postav a projektilů, zatímco `RigidBody3D` je vhodná pro nesamohybná pevná tělesa a `StaticBody3D` pro tělesa nehybná – podlahy a stěny (Linietsky et al., 2014).

Práce s `CharacterBody3D` vyžaduje úpravu vlastnosti `velocity`, trojrozměrného vektoru určujícího rychlost, která je posléze použita nativní metodou `move_and_slide()`. Tato metoda provede pohyb tělesa podle rychlosti a při kolizi s jiným objektem ji upraví tak, aby došlo ke skluzu podle úhlu dopadu. Zároveň je bráno v potaz, pokud se objekt nachází na pohyblivé plošině. Návratovým typem je `bool`, značící, zda došlo ke kolizi. V porovnání s tímto metoda `move_and_collide(velocity)`, pocházející z `PhysicsBody3D`, vyžaduje rychlost jako argument a při kolizi objekt výchoze zastaví a navrací objekt `KinematicCollision3D` s informacemi o kolizi (Linietsky et al., 2014)[1].

Tween

Podle dokumentace (Linietsky et al., 2014): `Tween` není uzlem, ale `RefCounted` objektem, který je vytvářen v rámci skriptu uzlu metodou `create_tween()`. Jedná se o nástroj pro asynchronní animace parametrů mezi dvěma hodnotami. Pokud je potřeba hodnotu nějakého parametru během určité doby plynule změnit na jinou, je `Tween` vhodným řešením, pro pokročilejší animace o mnoha hodnotách či parametrech je však určen `AnimationPlayer`. Tweening více parametrů lze provádět jak paralelně, tak sériově, a je možný výběr z řady interpolačních funkcí.

3.4.4 Datové formáty

Pro jinou než tichou textovou hru jsou potřeba tzv. *assety* – mediální data, jako jsou textury, 3D modely, obrázky, zvuky a hudba. Ty jsou ve většině případů vytvářené v jiných programech, než v editoru herního enginu, a je proto nutné řešit problematiku datových formátů a kompatibility; posléze jsou importovány a používány (Retro Style Games, 2023).

Základními nativními formáty jsou **TSCN** (*text scene*) k ukládání scén a **TRES** (*text resource*) k ukládání `Resources` v textové, a tudíž člověkem čitelné, podobě, s binární variantou **SCN** (Linietsky et al., 2014).

3D modely a scény

Godot využívá především formát **glTF 2.0** (přípony `.glb`, `.gltf`, `.bin`). Tyto soubory je možno v editoru nahrát přímo po exportu z cizího prostředí. Možné je také vytvořit spojení s open source 3D prostředím **Blender** a nahrávat objekty v jeho nativním formátu (`.blend`). Dalšími plně podporovanými formáty jsou **COLLADA** (`.dae`) a **Wavefront OBJ**

[1] Informace o kolizi lze získat i po zavolání `move_and_slide()` pomocí metody `get_slide_collision(slide_idx)`; jelikož během jednoho volání této metody může nastat mnoho kolizí, nelze automaticky vrátit informaci o jedné kolizi (Linietsky et al., 2014).

(`.obj`, `.mtl`). Jen částečně je podporován formát **FBX**, jehož konverze vyžaduje instalaci externího programu (Linietsky et al., 2014).[1]

Audiovizuální formáty

Godot podporuje mj. běžné bitmapové formáty – **PNG**, **JPEG**, **TGA**, **DDS** a také **OpenEXR** a **Radiance HDR** jakožto formáty s podporou HDR. Z vektorových formátů je podporován **SVG**. Ze zvukových formátů Godot podporuje **Wave**, **Ogg Vorbis** a **MP3.** Jediným podporovaným formátem videa je **Ogg Theora** (Linietsky et al., 2014).

3.5 Adaptivní hudba ve hrách

Jedním z cílů práce je návrh a implementace opětovně použitelného systému pro adaptivní hudbu. Hudba je ve hrách důležitým uměleckým nástrojem a interaktivita her otevírá možnosti dynamicky měnit podobu hudebního pozadí ve hře. Adaptivní hudba je taková, která reaguje na akce hráče a jiné události v herním světě (Whitmore, 2003). Sweet (2015) klasifikuje následující metodiky skládání herní hudby:

3.5.1 Neinteraktivní hudba

Na začátku herní sekce je jednoduše spuštěna hudební smyčka, která je beze změny opakována a je ukončena při skončení dané herní sekce; tato metoda je nejstarší a nejjednodušší, ale nenabízí žádnou proměnlivost nebo odezvu na události ve hře a neodpovídá tak potřebám na adaptivní hudbu (Sweet, 2015).

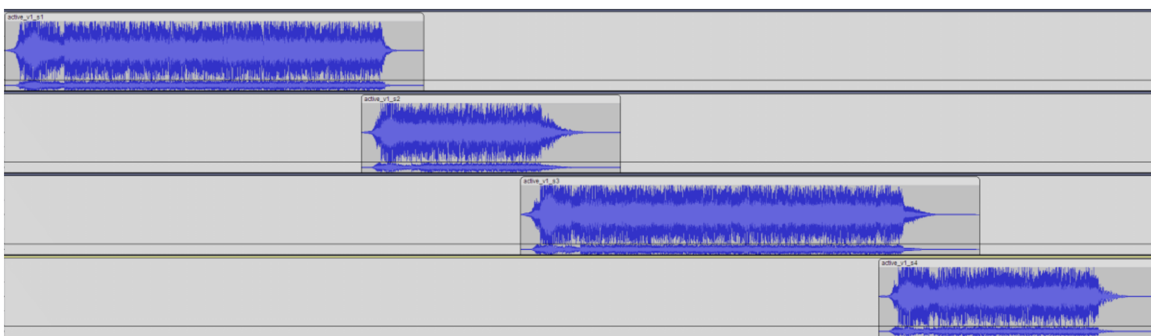
Příkladem budiž hry jako Doom nebo Commander Keen. Mírně interaktivnější variantou jsou neinteraktivní hudební stopy vyvolané herní událostí. Příkladem zde může být série Half-Life (Valve Corporation, 1998), kde je součástí skriptů úrovní jednorázové spuštění hudební stopy.

3.5.2 Horizontální přeskládání (horizontal resequencing)

Při této metodě je hudba rozdělena na krátké fráze. Na konci každé fráze je rozhodnuto, jakou další frázi přehrát, a dochází tak k větvení. Může tak docházet k plynulému přechodu mezi zcela rozdílnými typy hudby. Nevýhodou této metody je prodleva, která nastává mezi změnou herního stavu a změnou hudby, jelikož je potřeba pro změnu hudby počkat na konec fráze nebo jiný předem určený bod, např. konec doby (Sweet, 2015).

[1] Novinkou v nadcházející verzi 4.3 je zabudovaný open source konvertor FBX souborů (John 2024).

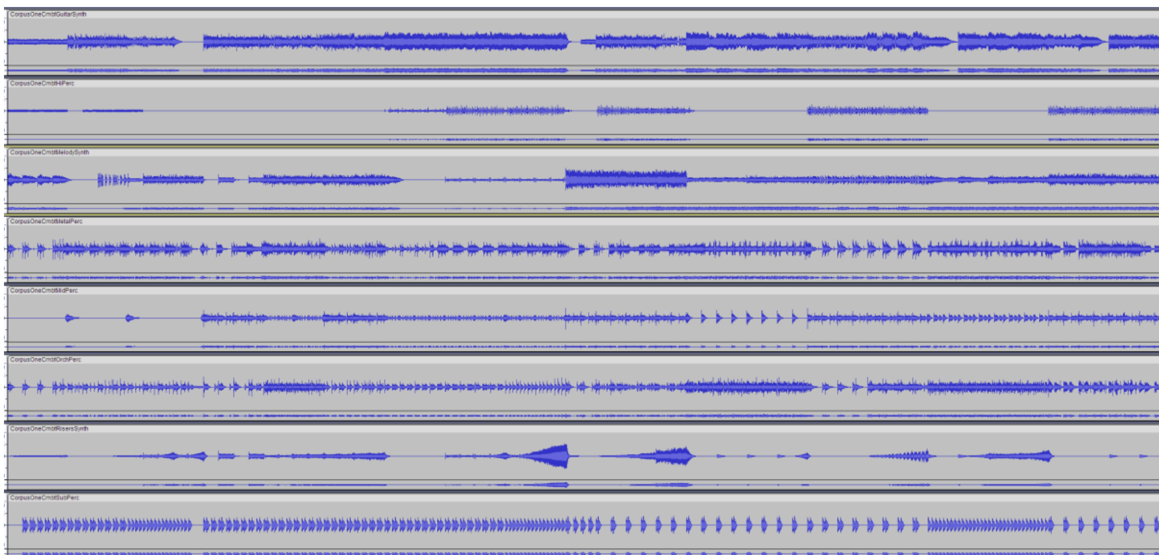
Mezi jednotlivými frázemi mohou být použity přechodné fráze nebo jednoduchá prolnutí. Pro přehlednost se v komplexních situacích využívá tzv. přechodových matic, které značí, jaká stopa má být použita pro přechod mezi jakýmikoli dvěma frázemi (Whitmore, 2003). Příkladem je zde uveden Doom (id Software, 2016), který během akčních sekvencí plynule spojuje dohromady krátké hudební fráze a reaguje na intenzitu souboje i úkony hráče. Zjednodušený příklad takového přeskládání je v audio editoru Audacity zobrazen na obr. 5.



obr. 5 Fráze horizontálního přeskládání ve hře Doom (id Software, 2016).

3.5.3 Vertikální míchání (vertical remixing)

Tato metoda rozděluje hudbu na vrstvy, určené k souběžnému přehrávání. Podle herních událostí je možné tyto vrstvy míchat a kombinovat. Zatímco přeskládání skládá v čase za sebe individuální hudební fráze, při míchání je přehráváno více audio stop najednou, jejichž hlasitost je určena hrou. Výhodou je možná okamžitá změna hudby podle potřeby, nevýhodou je nemožnost dynamicky měnit strukturu tak, jako u přeskládání (Sweet, 2015).



obr. 6 Vrstvy vertikálního míchání ve hře Warframe (Digital Extremes, 2013).

Příkladem je zde uvedena hra Warframe (Digital Extremes, 2013), ve které je v jednom typu úrovně v režimu souboje mícháno osm zvukových vrstev v závislosti na intenzitě akce. Na obr. 6 jsou tyto vrstvy zobrazeny v audio editoru Audacity.

3.5.4 Pokročilé metody

V případě některých her není hudba uložena ve formě zvukových nahrávek, nýbrž je prováděna hudební syntéza přímo za běhu programu, obvykle s využitím rozhraní MIDI. Tato metoda umožňuje nejvyšší možnou flexibilitu, kdy může být plynule, bez ztráty kvality, měněno tempo či stupnice, nicméně nevýhodou je vysoká technická náročnost na implementaci i výkon a syntézou není prakticky možné přesvědčivě mimikovat skutečné nástroje (Sweet, 2015).

Tyto metody lze kombinovat, příkladem budiž System Shock z roku 1994, který splňoval vše následující (Nightdive Studios, 2017):

- Každá úroveň hry měla tři různé oddělené skupiny skladeb: „klid“, „nebezpečí“, „souboj“. Mezi těmito skupinami bylo přepínáno podle herních událostí.
- K této hudbě byly míchány další vrstvy podle vzdálenosti hráče od různých nepřátel.
- Hudba využívala MIDI notace a syntézy hudby pomocí dedikovaných zvukových karet.

3.6 Serializace dat

Serializace znamená převod stavu objektu do datové formy, kterou je možno uložit nebo přenášet; opakem je deserializace, kdy jsou načtená data převedena zpět do podoby objektů (Microsoft Corporation, 2023). Serializace je základním principem ukládání a načítání stavu hry, aby hráč mohl po opětovném zapnutí hry pokračovat tam, kde skončil.

3.6.1 Nativní postupy v Godotu

Godot nabízí několik způsobů, jak uložit a načíst data za pomocí vestavěných funkcí.

JSON

Jedná se o všeobecně známý a člověkem čitelný formát, se kterým je možno snadno pracovat za pomocí třídy `JSON`, nepodporuje však vlastní typy a veškeré vestavěné typy; například `Vector3` je rozdělen do objektu se třemi čísly a při deserializaci je pak nutné tyto typy správně interpretovat manuálně (Linietsky et al., 2014). Další nevýhodou je potřeba odkazování na položky pomocí textových řetězců: Jejich manuální přepisování je náchylné na překlepy (Thomä, 2023c).

Binární serializace

Další metodou je binární serializace, při které jsou nativně podporovány všechny vestavěné datové typy. Možné je serializovat jednotlivé proměnné, následné načítání je však nejnáchylnější na chyby, kdy je potřeba držet v souladu pořadí ukládání a načítání proměnných. Toto lze sice řešit ukládáním proměnných do slovníkového typu (`Dictionary`) a jeho následnou binární serializaci, jelikož však slovníky využívají klíčování pomocí textových řetězců, nejsou takto řešeny všechny problémy JSONu (Thomä, 2023c).

Vlastní resources

Zdaleka nejflexibilnější je využití vlastních `Resources`: Je možno vytvořit novou dědící třídu pro ukládání, deklarovat v ní potřebné proměnné, uložit do ní požadovaná data a nativní třídou `ResourceSaver` serializovat na úložiště; po načtení jsou veškeré datové typy přímo dostupné a na proměnné je možno přímo odkazovat (Thomä, 2023c).

Thomä (2023b) varuje před rizikem neúmyslného spuštění cizího kódu při používání vlastních `Resources`, jelikož součástí těchto dat mohou být také skripty. Nebezpečí zde hrozí v případě pokusu o načtení infikované „uložené hry“ získané z cizího zdroje, například stažené z internetu. Řešením je daný `Resource` nejprve načíst jako text a validovat jej: Pokud se v něm nachází vložený skript nebo odkaz na soubor mimo interní souborový systém hry (`res://`), vůbec se jej nepokoušet načítat jako `Resource`.

ConfigFile

Pro ukládání především uživatelských nastavení (nikoliv herního stavu) je možné využít také třídu `ConfigFile`, vytvářející textové soubory ve stylu formátu INI, které jsou tvořeny sekcemi (bez vnoření), obsahující datové položky (Linietsky et al., 2014).

3.6.2 Serializace referencí a vnořených objektů

Serializace hodnotových typů je poměrně jednoduchá – data jsou uložena; data jsou posléze načtena. U referenčních typů je situace složitější: Na jednu instanci může být odkazováno z více míst a samotná referenční proměnná je jen to – reference. Řešení lze nalézt třeba ve zdrojovém kódu enginu GZDoom (ZDoom teams et al., 1998), který plnohodnotné ukládání dat nabízí jako zabudovanou funkci: Ukládání referencí na objekty jako identifikátory. Při následném načítání hry je pak deserializace prováděna ve dvou krocích: Nejprve jsou vytvořeny instance všech serializovaných objektů a ve druhém kroku

jsou přiděleny referenční proměnné podle identifikátorů. Z toho vyplývá, že každý objekt, na nějž je odkazováno ze serializovaného objektu, je proto také potřeba serializovat.

V případě uzlů ve stromu scény Godotu je také nutno předpokládat s podřízenými uzly. Dokumentace (Linietsky et al., 2014) zde varuje, že je proto objekty třeba načítat takovým způsobem, aby bylo možno na rodičovských uzlech volat `add_child()`.

3.6.3 Zpětná kompatibilita

Dalším problémem je načítání rozdílných verzí uložených souborů. Mezi některé, jež Thomä (2023a) řeší, patří:

- Změny interní souborové struktury projektu, znemožňující instancování ze souboru scény či skriptu. Pokud si je tohoto programátor vědom, může při načítání cesty upravit.
- Hra očekává proměnné, jež nejsou ve starším formátu uložené hry přítomny. V tom případě jsou načteny výchozí hodnoty a je proto vhodné je v definicích těchto tříd nastavit.

3.7 Akční hra

Hrou je možno dle Abbadiho (2016) označit libovolnou dobrovolnou aktivitu, ve které účastníci s něčím nebo někým interagují ke splnění určitého cíle za určitých podmínek a omezení – pravidel hry. Důvody ke hře mohou být různé, ať už externí (cizí doporučení věnovat se sportu) nebo interní (hráč se chce zabavit). V případě počítačových her hráči interagují s počítačovým programem, který na základě jejich vstupů vykresluje virtuální světy a stanovuje podmínky hry.

Hry vyžadují od hráče určité schopnosti, Schell (2008) je rozděluje na fyzické, mentální a sociální. Akční hry se vyznačují zaměřením na fyzické schopnosti, především koordinaci zraku a rukou a schopnosti reagovat, a do jisté míry se přidávají mentální schopnosti taktiky a strategie.

Kvintesenciální akční hrou, která značně přispěla k rozmachu žánru 3D stříleček z první osoby (Kushner, 2003), je Doom (id Software, 1993), zobrazená na obr. 7. Hra se skládá z map, uzavřených lokalit s předdefinovanými objekty. Cílem hráče je dostat se z počátku mapy na konec. Překážkami jsou útočící nepřátelé různých typů, kteří jsou schopni hráče zranit či zabít – stav hráče je vyjádřen číslem; když dosáhne nuly, umírá a mapa je načtena od začátku. Dalšími překážkami jsou zamčené dveře, vyžadující klíče či otevření na dálku, nebo samotná architektura úrovní, ve kterých je snadné zabloudit. Proti nepřátelům

hráč disponuje převážně střelnými zbraněmi, ke kterým je po mapě rozeseto omezené množství munice, spolu s předměty na obnovu zdraví hráče.

obr. 7 Ukázka hry Doom v source portu GZDoom
(id Software, 1993; ZDoom teams et al., 1998).

Jiné akční hry se mohou obsahem a způsobem hry značně lišit: Zatímco Doom (id Software, 1993) obsahuje minimální množství příběhu, odehrává se v bludištních měsíčních základnách s démonickými vetřelci a mechaniky jsou silně nereálné (např. postava hráče je schopna neomezeně sprintovat při nošení velkého množství zbraní nehledě na zdravotní stav), jiné hry se zaměřují na realitu, např. česká hra Vietcong se odehrává v historickém kontextu americké války ve Vietnamu a vyobrazuje vedle realističtějších schopností hráčské postavy také charakterizované a namluvené cizí postavy, a to ve spíše lineárních prostředích (Pterodon a Illusion Softworks, 2003).

4 Praktická část práce

Dílčím cílem této práce je na základě poznatků z teoretické části navrhnout a vytvořit ucelenou 3D akční herní aplikaci. Na vývoji hry jsou ukázány aplikace návrhových vzorů a využití funkcí enginu Godot. Metody pro serializaci dat a adaptivní hudbu jsou použity k jejich implementaci ve hře. Na konec je provedeno uživatelské testování. Hru, spustitelnou na platformě Microsoft Windows, lze nalézt v příloze 1.

4.1 Návrh hry

Výstupní aplikace, hra *En attendant ilkeätä noitaa: Čekání na černokněžníka* je stylem hry i estetikou inspirována hrou Doom a jinými hrami využívajícími engine id Tech 1, především Heretic a Hexen, případně pozdější engine Quake Engine (Hexen 2).

Zaměření hry je téměř výhradně akční: Cesta z počátku úrovně na konec je zahrazena především nepřátelskými příšerami, které jsou schopny hráče zranit či zabít. Hráč proti nim disponuje postupně silnějšími magickými zbraněmi: V úvodu pouze dýkou, později čarovnou knihou a nakonec mocnou kouzelnou holí. Munice je pro tyto zbraně omezená a lze ji doplnit sbíráním předmětů, obdobně jako hráčovo zdraví.

Hra obsahuje jednu úroveň obsahující několik oblastí s různě navrženými soubojovými sekcemi. Některé oblasti jsou již obydleny nepřáteli, v jiných se začnou objevovat až po vstupu hráče. Ukázky ze hry během vývoje jsou zobrazeny na obr. 8 (souboj s monstry), obr. 9 (panorama hradu) a obr. 10 (finální souboj s plným HUD).

obr. 8 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní).

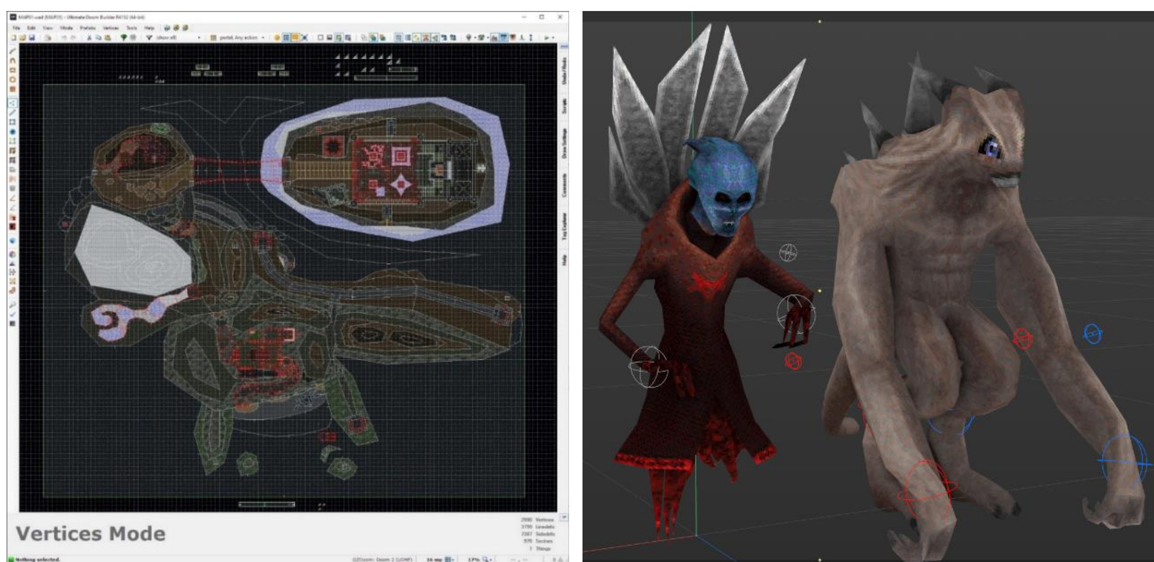
obr. 9 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní).

obr. 10 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní).

4.1.1 Herní assety

Pojem assety byl popsán v kap. 3.4.4. Z assetů třetích stran byl použit především balíček textur OTEX (Björling, 2018), určený primárně pro uživatelské mapy ve hře Doom, a dále velké množství zvuků z webu **freesound.org**, dostupné pod licencemi **Creative Commons (CC) Attribution** (s uvedením autora) a **CC Zero Attribution** (bez potřeby uvedení autora). Plné reference dle licencí lze nalézt v příloze 2. Ostatní assety: modely postav, zbraní a předmětů, animace, textury modelů, hudba a některé zvuky jsou autorské.

Statická architektura mapy byla pro požadovaný estetický styl vytvořena v editoru Ultimate Doom Builder, určeném k tvorbě map do hry Doom (obr. 11, část (a)). Po exportu do OBJ a následném re-exportu do FBX byla načtena v enginu Godot bez větších potíží. Animované modely (obr. 11, část (b)) byly také dodány ve formátu FBX, který se navzdory oficiálnímu varování slabší podpory neukázal být problematický.

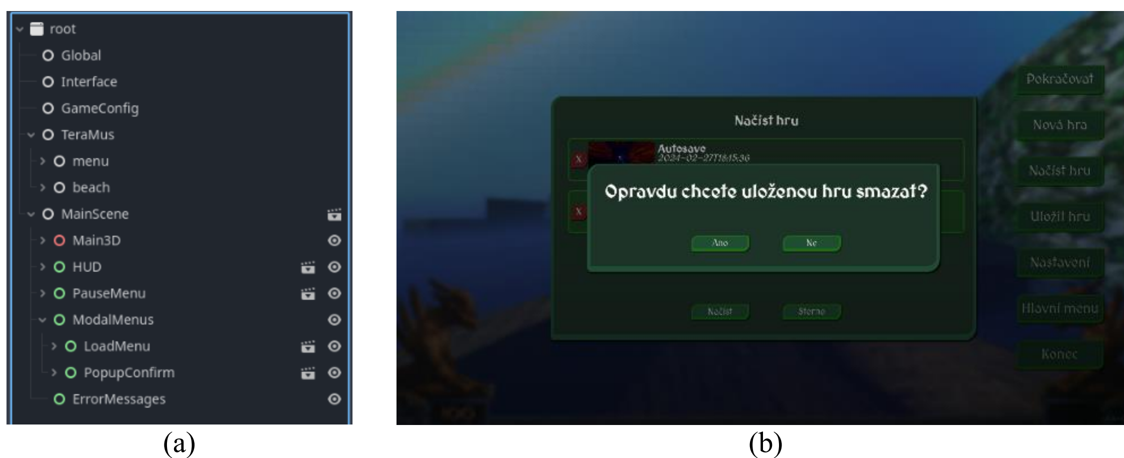


(a) (b)

obr. 11 Assety hry v editorech UDB a Cinema 4D (zdroj: vlastní).

4.2 *Struktura hlavní scény*

Hlavní scéna je v projektu Godotu ta, která je při spuštění aplikace jako první a jediná instancována. Její struktura je proto důležitá k celkové organizaci běžící aplikace, především pro pozastavení běhu hry (viz 3.4.3, část Node). Implementace Nathana Wulfa (2022) obsahuje tři pod-uzly: `Main3D`, uzel obsahující aktivní 3D svět; `Menu`, GUI uzel, který je vždy aktivní nebo při pauze; a `HUD`, GUI uzel, který je při pozastavení hry také pozastaven. Struktura hlavní scény ve výstupní aplikaci z tohoto základu vychází a rozšiřuje jej.



(a) (b)

obr. 12 Struktura hlavní scény hry (zdroj: vlastní).

Na obr. 12, části (a) je zobrazena struktura hlavní scény aplikace za běhu hry spolu s několika aktivními autoloads (singletony). Část (b) ukazuje stejnou situaci, jak ji vidí hráč.

Uzly `Main3D` a `HUD` zůstaly zachovány, uzel `Menu` však byl rozdělen na trvalé menu herní pauzy `PauseMenu` a pomocný uzel `ModalMenus`, fungující jako složka pro menu aktivní nehledě na stav herní pauzy a blokující interakci s menu pod nimi; patří sem hlavní menu, okno s konfigurací hry, vyskakovací dialogová okna aj., která jsou dle potřeby zde instancována a odsud odstraňována. Vždy aktivní, neblokující uzel `ErrorMessages` umožňuje zobrazování chybových hlášení pro vývojáře.

4.3 Programování hry

V této kapitole je popsáno použití vhodných postupů objektově orientovaného programování při vytváření jednotlivých herních objektů a komponent.

4.3.1 Komponentová tvorba a signálová komunikace

Příkladem objektu, sestávajícího – komponovaného – z mnoha komponent, je scéna hráčské postavy (obr. 13, část (a)), ve které se nachází např. komponenta `Input` (seznam jejích signálů je zobrazen v části (b)).

obr. 13 Struktura hráčské postavy (zdroj: vlastní).

`Input` má na starosti jediné: Zachytávat vstup z myši a klávesnice hráče a předat jej po minimálním zpracování dál (např. aplikování citlivosti myši z nastavení hry na vstupní pohyb myši); má jedinou zodpovědnost (viz kap. 3.1.2, část Single Responsibility Principle).

Reference na přijímající komponenty mohou být nastavené přímo ve třídě `Input`, takto však vzniká silná provázanost (popsaná v kap. 3.1.1, část Provázanost): Pokud by nějaká cílová komponenta byla ze hry odstraněna, pokus o její referenci by buď způsobil pád hry, nebo by bylo potřeba ji obalit kontrolou `null` hodnoty. V případě signálů jednoduše není odstraněná cílová komponenta „přihlášena" k poslechu zpráv ze zdrojové komponenty.

Nevýhodou je zde možnost tichého selhání: Pokud programátor zapomene přiřadit signálová spojení, hra místo pádu a hlášení chyby pracuje nesprávně[1]. Pevné přiřazení je použito např. ve třídě `Hitbox`, které musí být přidělena reference na komponentu `Health` (princip: Firebelley Games (2022)), jelikož existence prvé komponenty bez druhé nemá účel.

4.3.2 Využití rozhraní

Rozhraní jsou důležitým nástrojem objektově orientovaného programování, jak uvádí například princip obrácení závislostí (viz kap. 3.1.2, část Dependency Inversion Principle). GDScript však ve své současné podobě tuto funkci neobsahuje. Jedná se o duck-typed jazyk a jak bylo uvedeno v kap. 3.1.4, části Duck typing, pro tyto jazyky je důležité pouze, zda cílový objekt implementuje chtěnou metodu. To je možno zjistit metodou `has_method(name: String)`, tento postup však nevytváří závazek cílové třídě implementovat dané rozhraní, nehledě na použití jména požadované metody ve formě textového řetězce (popsáno jako nevhodné v kap. 3.1.6, části Nadužívání textových řetězců). Jako náhrada byla proto použita knihovna *gdscript-interfaces* (Tutemic, 2023), která napodobuje statickou kontrolu při startu hry: Pokud se nějaká třída zavazuje implementovat rozhraní, ale v plné míře to nečiní, je hra pomocí aserce s chybovou hláškou shozena.

Praktický příklad: Často používaným rozhraním je `Activatable`, určené pro herní objekty, které je možno na mapě aktivovat. Může se jednat o dveře, spínače, či abstraktní pomocné objekty aktivované skriptem. Samotné rozhraní je v autoloadu knihovny definováno takto:

```
class Activatable:
	func _trigger(source: Node3D, type: Consts.TriggerSources)²:
		pass # prázdná metoda
```

[1] Jako řešení může být při instancování scény použita aserce (viz 3.2.1), kontrolující, zda existuje alespoň jedno signálové spojení požadovaného typu, k hlasitému upozornění vývojáře na možné pochybení.

[2] Vhodnějším návrhem by zde mohla být obalující třída ActivationEvent. Případná budoucí rozšíření argumentů by tak neměla vliv na signatury metod, které by tak nebylo nutné upravovat.

Každá třída, implementující toto rozhraní, musí definovat metodu `_trigger()`. Objekt dveří například obsahuje tuto metodu:

```
var implements = Interface.Activatable # Může být pole v případě více rozhraní.
@export var player_uses := true        # @export značí nastavitelnost v editoru

func _trigger(source: Node3D, type: Consts.TriggerSources) -> void:
	if type == Consts.TriggerSources.PLAYER_USED and not player_uses:
		return
	_on_open()
```

Když jsou dveře aktivovány, mohou být otevřeny. Pokud je však jako typ akce (specifikovaný enumerací) specifikován hráč a dveře mají být pro hráče zamčené, jsou ponechány zavřené. Samotná aktivace je provedena z hráčské komponenty:

```
func _on_pressed_use() -> void:         # Přijímá signál z komp. Input
	_ray.force_raycast_update()      # Přidělený raycast uzel
	var col = _ray.get_collider()    # Získáme informace o kolizi
	if Interface.node_implements(col, Interface.Activatable):
		col._trigger(actor, Consts.TriggerSources.PLAYER_USED)
```

Rozhraní umožňují přidávání funkčnosti hierarchicky odděleným třídám: Dveře dědí ze třídy `StaticBody3D`, spínací oblasti/zóny ze třídy `Area3D`, přepínací světla z třídy `Light3D`. Pokud by byly implementace aktivovatelných objektů vytvářeny pomocí dědictví, neměly by spolu ve skutečnosti, kromě názvů metod, nic společného. Díky duck typingu toto nepůsobí zpočátku problémy. Pokud by však bylo požadováno k takovému „rozhraní“ přidat novou metodu a posléze se ujistit, že ji všechny třídy implementují, bylo by třeba manuálně kontrolovat všechny třídy, které toto rozhraní implementují. Dalším často použitým rozhraním je `Serialisable`, které bude podrobněji popsáno později.

4.3.3 Použití návrhových vzorů

Návrhové vzory byly využity podle poznatků v kapitole 3.1.3.

Observer – signály

Jednoduché použití tohoto vzoru bylo již popsáno v podobě signálů. V tradičním názvosloví by byl objekt, vysílající signál, nazýván subjektem; objekt s přijímající metodou pozorovatelem. Signály je však možno využít i jinými způsoby.

Programatické přidělení

Pokud je nějaká třída či scéna instancována za běhu hry, nelze pochopitelně vytvářet propojení signálů pomocí editoru a spojení je třeba vytvořit v rámci skriptu. Příkladem je

třída `MonsterSpawner`, která během akčních pasáží na mapě vytváří nové instance příšer. Tato třída je nastavitelná, aby mohl návrhář úrovně určit průběh souboje, a je u ní možné navolit např. maximální povolený počet v jednu chvíli existujících příšer nebo vyslání signálu v okamžik, kdy jsou všechny vytvořené příšery zničeny. Pro toto je však potřeba, aby spawner měl ponětí o sebou vytvořených entitách.

Nejprve je ve třídě `MonsterCharacter` deklarován signál, který je vyslán v okamžiku úmrtí. Ten je vyslán z metody, která je sama o sobě zavolána signálem z komponenty `Health`. Dochází tak k řetězení signálů, jak je popsáno v kap. 3.4.2, části Signály:

```
signal died(me: MonsterCharacter)

func _on_died(source: Node3D, amount: int) -> void: # aktivuje sig. Health.died()
	died.emit(self)
```

Při vytvoření příšery je propojen signál `died` nově vytvořeného objektu `MonsterCharacter` s metodou `_on_died()` ve třídě `MonsterSpawner`:

```
func _spawn(point: Transform3D, obj: PackedScene) -> void:
	var instance := Global.spawn(obj) # Global je pomocný singleton
	# nastavení a kontroly
	_monster_counter += 1 # členská proměnná
	inst.died.connect(_on_died)

func _on_died(who: MonsterCharacter) -> void:
	_monster_counter -= 1
	# počet zbylých spawn eventů (tvorba monster) a počet zbylých monster
	spawn_died.emit(spawn_events - _event_counter, _monster_counter)
	if _monster_counter == 0 and _event_counter == spawn_events:
		all_died.emit() # souboj skončil – všechna monstra poražena
```

Samotný `MonsterSpawner` obsahuje vlastní signály, které je možné použít pro skriptování úrovní, např. poskládání několika odlišných spawn-sekvencí za sebe pro proměnlivé ale s úmyslem navržené akční pasáže.

Flyweight – Resources

Informace o sbíraných předmětech (zbraně, munice, zdraví) využívají třídu `ItemInfo`, děděnou z `Resource`. Třída obsahuje mj. název předmětu, odkaz na scénu s 3D modelem a maximální sbírané množství. Z tohoto základu lze vytvářet samotné položky a ukládat je v `.tres` souborech. Ty je poté možné opět nahrát a použít pro statické informace o předmětu. Toho je využito ve scéně `ItemPickup`, která je umístitelná v mapě a která mění své chování a vzhled podle přiděleného `ItemInfo`. Při vstupu do její oblasti hráčem je daný předmět

přidán do hráčova inventáře v podobě instancovaného uzlu `Item`: Ten je měnný (vnější stav) a vedle reference na dané `ItemInfo` obsahuje proměnnou s aktuálním množstvím daného předmětu, které hráč má u sebe; `ItemInfo` je neměnný „plán" či „blueprint" (vnitřní stav).

`Resources` jsou také použity pro serializaci uložených her a systémem pro adaptivní hudbu.

Singleton - autoloads

Vzor jedináček je v enginu Godot nativně implementován v podobě funkce autoloads. Jedná se v praxi o uzly, které jsou instancovány na stejné úrovni, jako základní scéna. Přístup k nim je možný pomocí názvu třídy, jako by bylo přistupováno ke statické třídě, bez potřeby pomocné metody jako u klasického singletonu. Aplikace využívá několika jedináčků; kde to však bylo možné, byly namísto nich použity statické třídy, či bylo uvažováno, zda jsou singletony opravdu nutné, vzhledem k poznatkům z teoretické části.

Výhodou autoloads je možnost použití signálů, jelikož statické signály nejsou již z podstaty možné. Třída `GameConfig`, spravující nastavení hry (navolené grafické možnosti, hlasitost hudby a zvuku, předvolby ovládání), jich využívá pro oznámení o změně grafických nastavení objektům ve scéně:

```
#game_config.gd – singleton
signal shadows_changed()
var shadow_quality := ShadowQuality.MEDIUM: # ShadowQuality je enumerace
	set(value):
		shadow_quality = value
		_save_config()			# uloží konfigurační soubor
		shadows_changed.emit()

# shadow_config.gd – uzel podřízený typům Light3D pro změnu kvality osvětlení
func _ready() -> void:
	GameConfig.shadows_changed.connect(_on_shadows_changed)

func _on_shadows_changed() -> void:
	# změna kvality stínů a osvětlení dle GameConfig.shadow_quality
```

Event bus

Objekt může přímo vyvolat signál i v jiném objektu, nejen pouze v sobě. Toto je výhodné ve chvíli, kdy dva objekty, které nejsou spolu přímo schopny navázat spojení, potřebují komunikovat a mají přístup k jinému objektu, například singletonu.

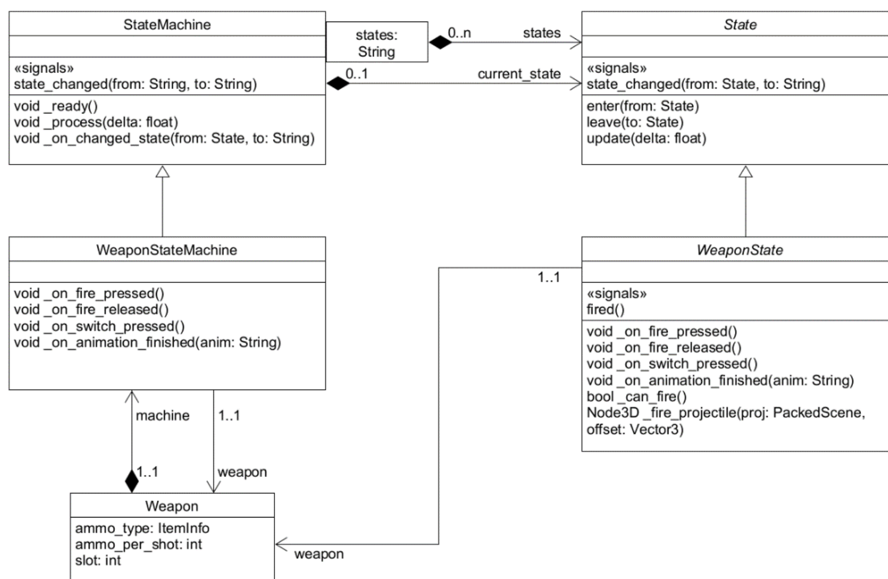
V aplikaci byl tento postup použit především k předání informací grafickému rozhraní, např. o změně hodnoty zdraví hráče:

```
# health.gd – používáno jak hráčem, tak monstry
var current_health: int:
	set(value): # settery v GDScriptu nevyžadují backing proměnnou
		# propojeno s uzlem Player ve scéně hráče pomocí editoru
		health_changed.emit(current_health, value)
		current_health = value

# player.gd
func _on_hp_changed(old_hp: int, new_hp: int) -> void:
	Global.player_hp_changed.emit(old_hp, new_hp)
	# Global je auto-load třída – singleton.
	# Tato implementace nevyžaduje volat pomocnou metodu.

# global.gd
signal player_hp_changed(old_hp: int, new_hp: int)

# hp_number.gd (rozšiřuje GUI třídu Label)
func _ready() -> void:
	Global.player_hp_changed.connect(_on_hp_changed)

func _on_hp_changed(old_hp: int, new_hp: int) -> void:
	text = "%d" % new_hp
	label_settings.font_color = (
		Color.DARK_RED if new_hp < 25 else Color.NAVAJO_WHITE )
```

Uzel `Health` vyšle signál, zachycený nadřazeným uzlem `Player`. Ten podle jeho informací vyšle signál v globálně přístupném autoloadu `Global`. K tomuto signálu je připojena metoda v uzlu `HpNumber`, ukazující hodnotu hráčova zdraví. Zde je zobrazená hodnota zdraví upravena a pokud je zdraví pod určitou hodnotou, je také nastavena odlišná barva textu.

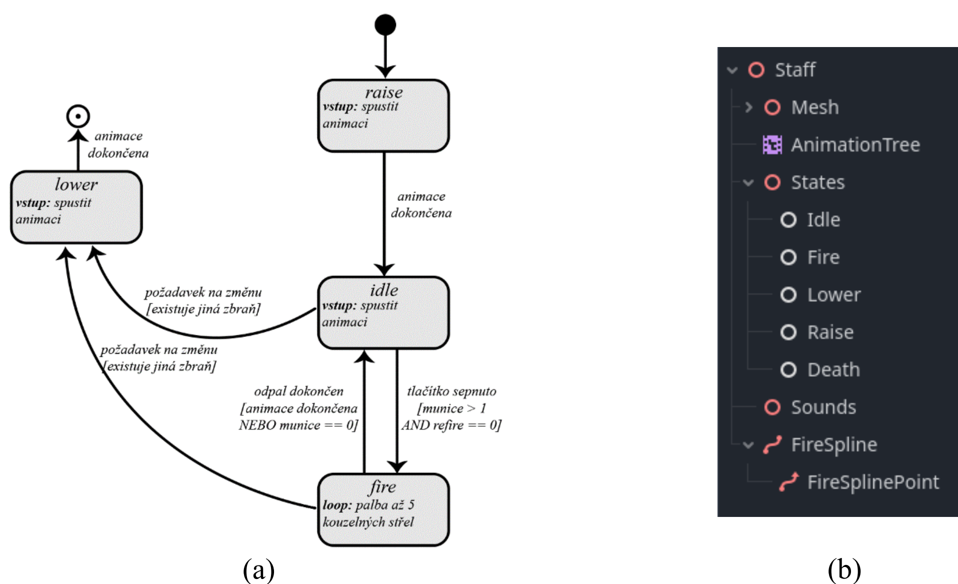
State

Stavový automat byl použit pro umělou inteligenci nepřátelských postav a ovládání hráčských zbraní. Základní struktura je tvořena nadřazeným uzlem typu `StateMachine` a podřízenými uzly typu `State`, reference na něž jsou uloženy ve slovníkovém typu. Automatem je v aktivním stavu pravidelně volána metoda `_update` a přijímající metody příchozích signálů. Změny stavů jsou taktéž vysílány signálem z automatu. Implementace v aplikaci vychází z verze od Bitlytic (2023). UML třídní diagram je zobrazen na obr. 14.



obr. 14 UML třídní diagram stavového automatu se stavy[1] (zdroj: vlastní).

Funkce jedné zbraně ve hře, kouzelné hole, je znázorněna UML stavovým diagramem (popsány v kap. 3.1.5) na obr. 15, části (a); struktura její scény je zachycena v části (b).



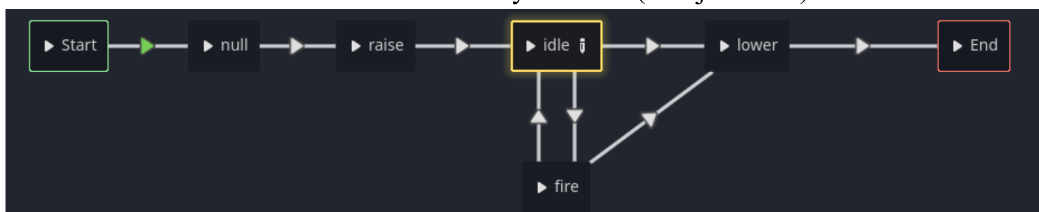
(a) (b)

obr. 15 Stavový diagram a uzlová struktura zbraně – kouzelné hole (zdroj: vlastní).

Stavové automaty a diagramy jsou v rámci enginu Godot přímo použity jako nástroj pro modelování míchání animací pomocí uzlu `AnimationTree` (popsán v kap. 3.4.3). V případě zbraní je ve výstupní aplikaci dokonce použit hierarchický stavový automat: Hlavní automat, zobrazen na obr. 16, rozlišuje pouze zda je držitel zbraně naživu či ne. Stav `alive` je superstav (struktura zobrazena na obr. 17) a obsahuje stavy pro animace pohybu hráče, střelby nebo vysunutí a zandání zbraně.

[1] Vhodnou alternativou k dědícím *Weapon** třídám by zde mohlo být rozhraní s metodami přijímající signály *(_on_*)* a referencí na zbraň, implementované v rozšiřujících skriptech (ne třídách!) pro obě třídy. Zároveň by bylo vhodnější místo textového klíčování slovníku používat enumerace.

obr. 16 Hlavní stavový automat (zdroj: vlastní).

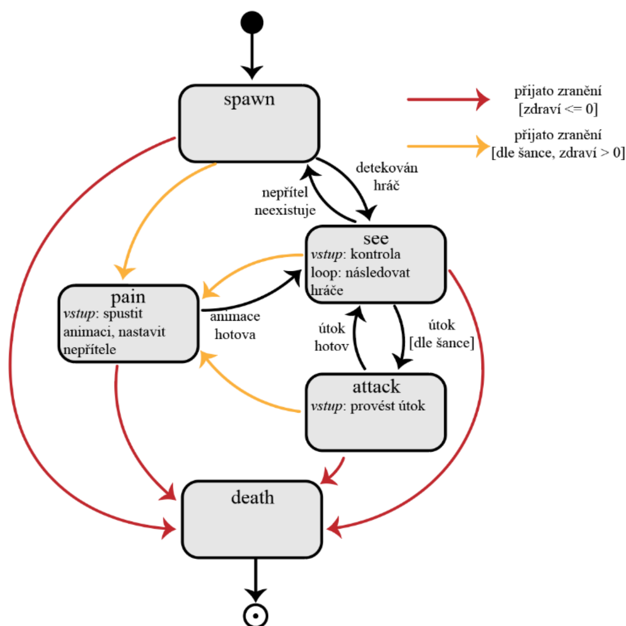


obr. 17 Superstav `alive` (zdroj: vlastní).

Kupříkladu celý klidový stav zbraně vypadá v kódu takto:

```
@export var newfire_delay: float # Prodleva po palbě, než je možno pálit znovu.
var _newfire_left: float = 0

func _enter(from: State):
	_newfire_left = 0
	if not from:
		return
	if from.name.to_lower() == FIRE_STATE: # Pokud byla zbraň právě
		_newfire_left = newfire_delay    # aktivována, možno pálit hned.

func _update(delta: float):
	_newfire_left -= delta

func _on_fire_pressed():
	if _newfire_left > 0:      # Ještě neuplynula prodleva po palbě.
		return
	if _can_fire():            # Kontrola, zda je možno střílet.
		changed_state.emit(self, FIRE_STATE)
```

Nepřátelské postavy pracují na stejném principu. Výchozí implementace využívá struktury stavů á la Doom (id Software, 1993), vyobrazené stavovým diagramem na obr. 18.



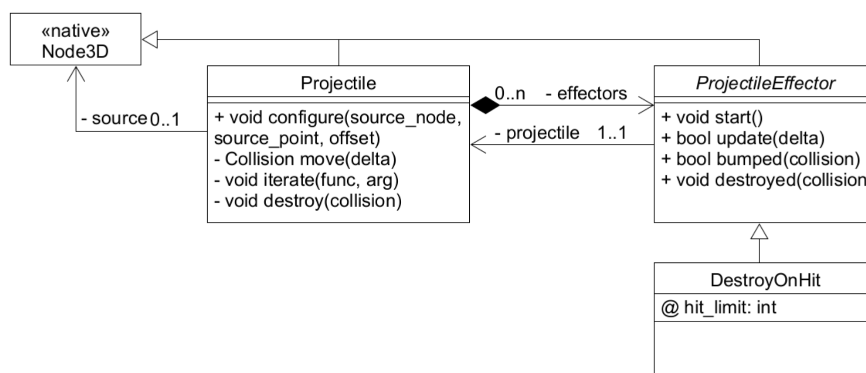
obr. 18 Stavový diagram generické nepřátelské postavy (zdroj: vlastní).

Decorator

Vzor dekorátor byl použit především pro konstrukci projektilů. Byla využita horizontální varianta, která namísto zabalování komponent a rekurzivního volání metod využívá iteraci skrz pole komponent. Dekorátory jsou zde proto raději nazývány **efektory**.

Základem je uzel `Projectile` typu `CharacterBody3D` (popsaný v kap. 3.4.3), rozšířený o skript iterující skrz potomky typu `ProjectileEffector`. Tato třída obsahuje, kromě reference na nadřazený `Projectile`, přidělenou jím samotným, tři virtuální metody: `_update()`, `_bumped()` a `_destroyed()`. Tyto metody jsou volány rodičem při provádění `_process()`, při kolizi s jiným objektem a při zničení projektilu, zažádaném předešlými dvěma metodami. Z těchto jednotlivých efektorů, které jsou na sobě zcela nezávislé, lze jednoduše sestavit kompletní chování objektu.

Tato struktura je zobrazena pomocí UML třídního diagramu na obr. 19 spolu s příkladem jednoho konkrétního efektoru.



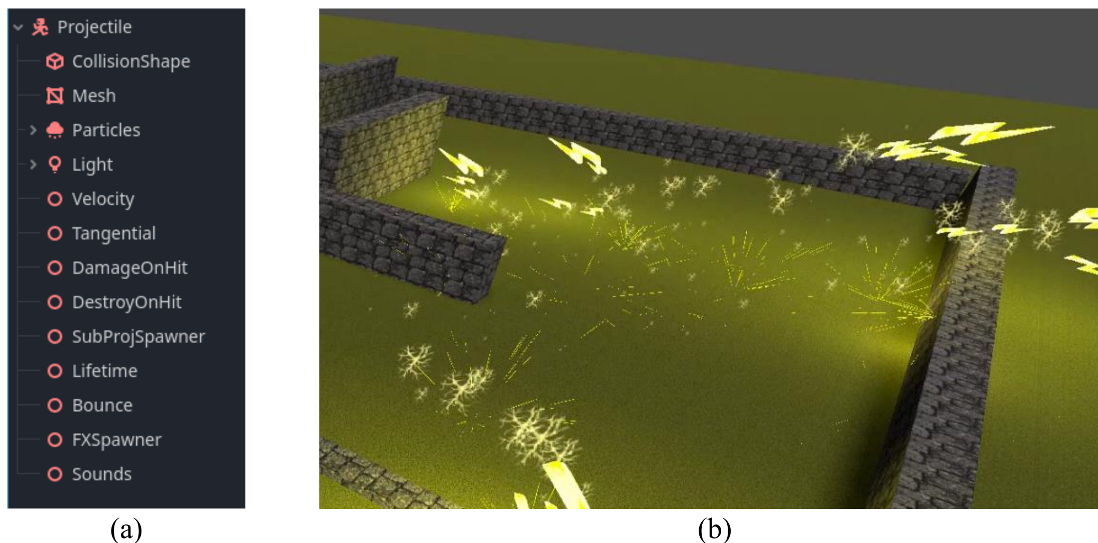
obr. 19 UML diagram tříd projektilu a abstraktního a konkrétního efektoru.[1](zdroj: vlastní).

Komplexním příkladem je zde projektil `Lightning`, používaný magickou holí. Jedná se o blesk, který během letu vytváří po stranách menší blesky a odráží se od stěn. Na obr. 20, části (a) je zachycena uzlová struktura projektilu, v části (b) projektil přímo ve hře (na testovací mapě), kde jsou patrné zmíněné menší blesky, odrážení a grafické efekty.

Význam všech použitých efektorů v části (a) je následující: Efektor `Velocity` nastavuje výchozí rychlost projektilu a případné konstantní úpravy – gravitaci, odpor vzduchu. Uzel `Tangential` udržuje rotaci projektilu ve směru jeho vektoru rychlosti. `DamageOnHit` při zásahu komponenty `Hitbox` vyvolá zranění. `DestroyOnHit` po určitém počtu kolizí zničí projektil. `SubProjSpawner` periodicky vytváří pod-projektily, v tomto

[1] @ zde značí exportní proměnnou, která je upravitelná v editoru a uložená se souborem scény; ekvivalent vytvoření instance třídy a definování hodnoty proměnné zvenčí.

případě menší blesky, které dále již další projektily netvoří. `Lifetime` limituje životnost projektilu na určitý čas. `Bounce` projektil při kolizi odráží. `FXSpawner` instancuje scény, obvykle grafické efekty, při kolizi, periodicky a zničení projektilu. `Sounds` při těchto událostech produkuje zvuky.



(a) (b)

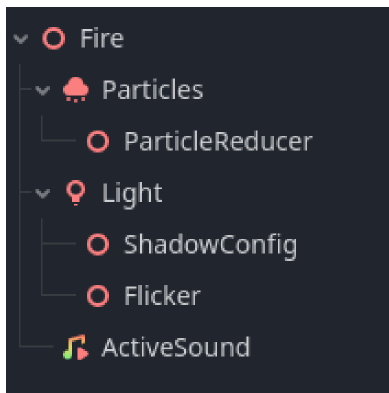
obr. 20 Struktura a ukázka projektilu blesku (zdroj: vlastní).

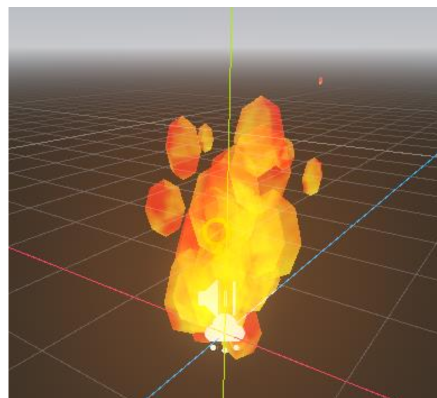
Pokud jakýkoliv efektor z metody `_update` nebo `_bumped` navrátí hodnotu `true`, má být projektil zničen a provedeny metody `_destroyed`, např. zde `DestroyOnHit` určuje počet kolizí ke zničení projektilu:

```
@export var hit_limit: int = 1

func _bumped(bump: KinematicCollision3D) -> bool:
	hit_limit -= 1
	return hit_limit < 1
```

Dekorátory byly použity také pro kombinování různých zdrojů pohupování kamery hráče, jmenovitě způsobené chůzí, skoky a dopady. Princip je v zásadě stejný: Nadřazená třída prochází potomky jednoho typu a volá určité metody. V tomto případě však tyto metody pouze navracejí hodnoty, které jsou rodičovským uzlem sečteny a aplikovány na objekt kamery.

Jistou alternativou jsou modifikátory, které svůj nadřazený uzel objevují sami. Jednoduchým příkladem je zde uveden efekt ohně (obr. 21, část (b)). V části (a) je zobrazena jeho uzlová struktura. Uzly `Particles` a `Light` obsahují podřízené uzly, které je upravují.





(a) (b)

obr. 21 Struktura a vizuály efektu ohně (zdroj: vlastní).

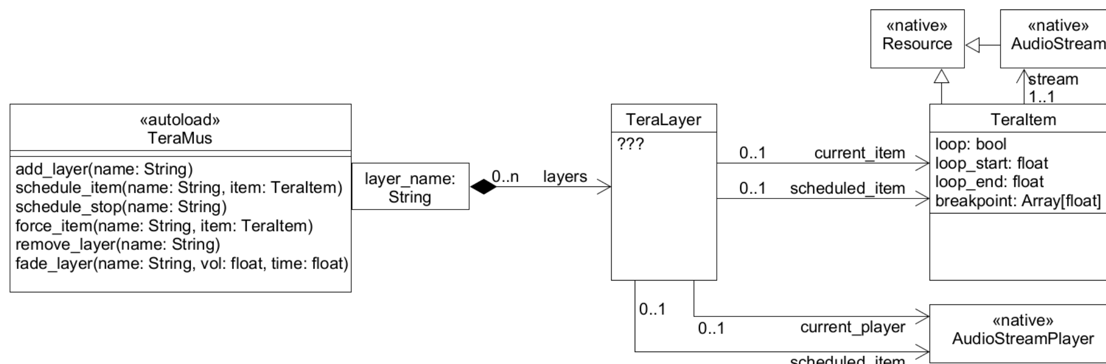
Všechny tři uzly na fungují na stejném principu: Při inicializaci zjistit svého rodiče a pokud se jedná o správný typ, pracovat s ním[1]. Tímto způsobem byly světlu přiděleny dva nezávislé modifikátory; první umožňuje světlu reagovat na předvolby grafické kvality hry a druhý provádí blikání intenzity, dosahu a barvy světla. Opět je dosaženo nezávislosti, není zatěžována třídní hierarchie a ani není zapotřebí rozšiřujícího skriptu pro upravované uzly: Kompozice je dosaženo za pomoci samotného stromu uzlů scény.

4.4 Systém adaptivní hudby

Význam a metody adaptivní hudby ve hrách byly popsány v kapitole 3.5. Systém TeraMus je určen pro jednoduchou implementaci adaptivní hudby v enginu Godot. Umožňuje hernímu designérovi snadno provést jak vertikální míchání, tak horizontální přeskládání hudebních stop.

4.4.1 Návrh

Již prvotní návrh měl požadavky na jednoduché použití, viz třídní diagram na obr. 22.



obr. 22 Nástin funkce systému TeraMus pomocí třídního diagramu UML (zdroj: vlastní).

[1] Nehledě na explicitní kontrolu typu se nejedná o porušení Liskovové principu zastupitelnosti (viz 3.1.2). Opačným případem by bylo, pokud by např. uzel `ShadowConfig` po prvotní kontrole rodiče jakožto `Light3D` dále kontroloval, zda je rodič `OmniLight3D`, `SpotLight3D` nebo `DirectionalLight3D`.

Pro tvůrce herní hudby je důležitý `Resource` `TeraItem`, který obsahuje samotnou zvukovou stopu a informace důležité pro smyčky (looping, opakování) a horizontální přeskládání. Designér herní úrovně naopak pracuje se singletonem `TeraMus`, ve kterém vytváří nové vrstvy `TeraLayer` (ukládané ve slovníku) a přiděluje jim požadované `TeraItems`, které jsou vrstvami přehrávány. Vertikálního míchání je dosaženo použitím více vrstev, horizontální přeskládání požadavkem na změnu `TeraItem` vrstvy. Na nízké úrovni pracují samotné vrstvy, které jsou zcela neveřejné.

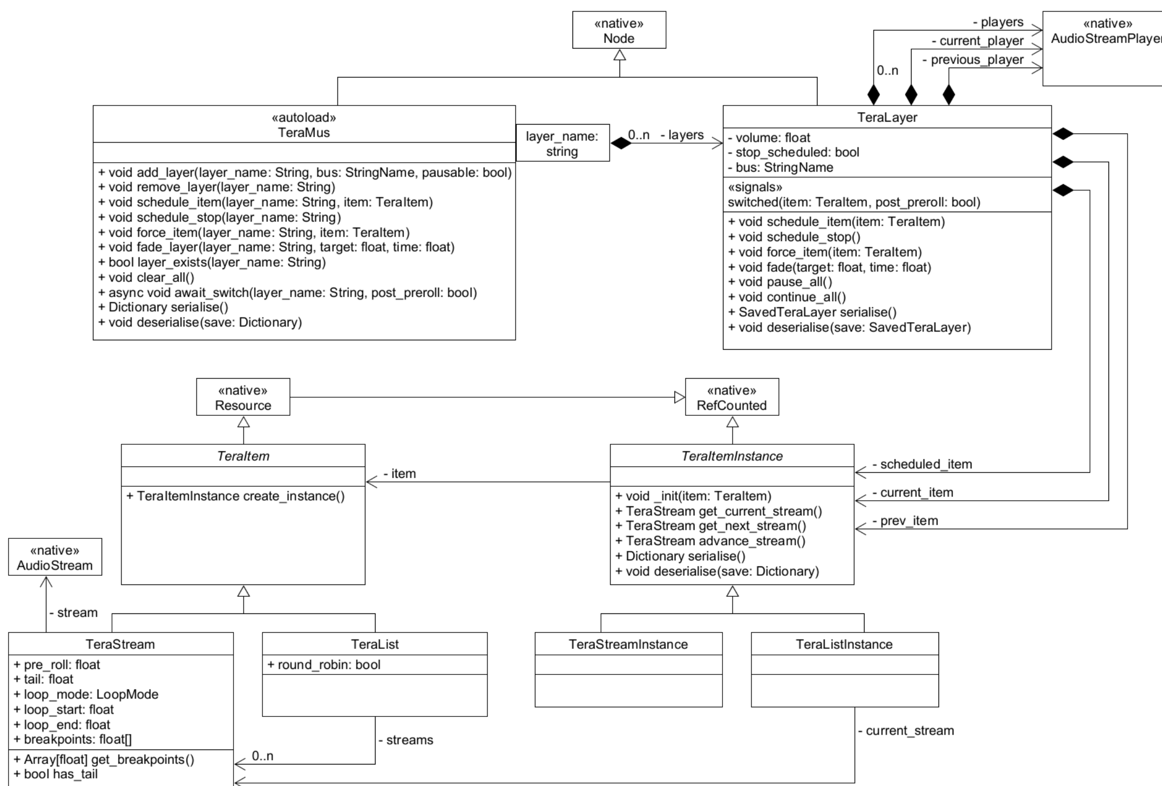
Použití textových řetězců pro přístup k vrstvám je sice možným antivzorem nadužívání stringů, nicméně je jím bezpečně dosaženo abstrakce – uživateli není umožněno snadným způsobem se dostat k samotné vrstvě jako v případě přímého přístupu. Použití enumerace by vyžadovalo zásah potenciálním vývojářem do kódu systému. Použití symbolických konstant může předejít některým problémům se stringy (viz kap. 3.1.6, část Magické číslo).

4.4.2 Implementace

Pro samotné přehrávání je použit uzel `AudioStreamPlayer` (zběžně ukázán v kap. 3.4.1, části Hierarchie tříd). Metody `get_playback_position()` a `seek(time)` umožňují kdykoliv zjistit a změnit současnou pozici přehrávání. Tato kontrola je prováděna při každém volání `_process(delta)` ve vrstvě. Pokud byl zaslán požadavek na změnu stopy a přehrávání na aktuálním přehrávači se nachází v bodu pro přechod (s tolerancí ± `delta`), je na novém přehrávači spuštěna požadovaná hudba a ten je nyní považován za aktuální. Předchozí přehrávač lze buď zahodit, nebo jeho přehrávání přesunout na pozici dozvuku.

Finální implementace (viz obr. 23, se skrytými privátními metodami) ponechává základní dělbu práce, kromě většího množství možností pro uživatele jsou zde však z hlediska vnitřního fungování následující důležité rozdíly:

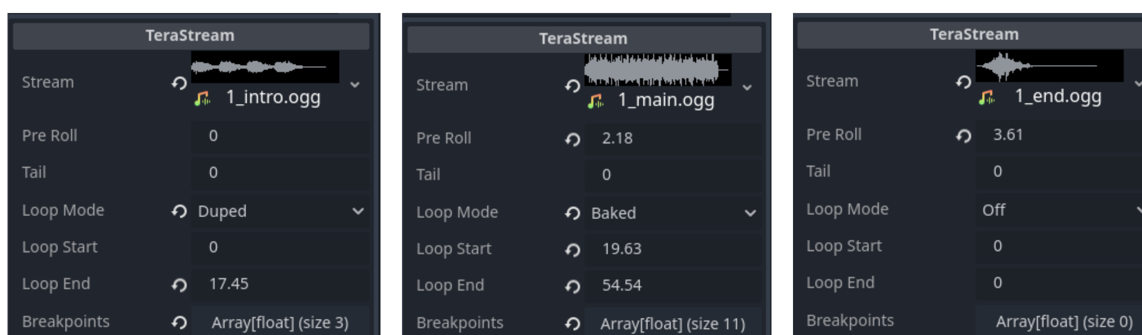
- `TeraItem` je abstraktní třídou, ze které v současné podobě dědí třídy `TeraStream` a `TeraList`. `TeraStream` obsahuje data o jednom zvukovém proudu a metadata pro přehrávání – stejně, jako v návrhu `TeraItem`. `TeraList` obsahuje pole `TeraStreams`, které je možno sekvenčně nebo náhodně vybírat.
- Při tvorbě `TeraListu` vyvstala potřeba stavu (právě vybraný stream), přičemž `Resources` by měly být bezstavové. Mezi `TeraItem` a `TeraLayer` byla proto zavedena třída `TeraItemInstance` a její dědicové.



obr. 23 UML třídní diagram současné podoby systému TeraMus. (zdroj: vlastní).

4.4.3 Použití

Veškerá hudba ve hře *Čekání na černokněžníka* je přehrávána pomocí systému *TeraMus*. Jednoduchý příklad ze hry: Úvodní sekce hry v zátoce obsahuje hudební pasáž o třech částech: úvodní smyčku, hlavní smyčku pro akční sekvenci a neopakující se závěr. `TeraStream` zdroje jsou uživatelem definovány v editoru, viz obr. 24.



obr. 24 Definice TeraStreams v editoru (zdroj: vlastní).

Enumerace `loop_mode` značí druh smyčky: `Off` je bez smyčky, `Baked` značí smyčku, kdy dozvuk po konci smyčky nemá být přehráván, protože je již přítomen i na začátku smyčky a `Duped` značí, že se má oblast po konci smyčky nechat dohrát a začít novou smyčku v novém přehrávači. Hodnota `pre_roll` značí náběhovou dobu hudby a tedy posunutí startu přehrávání dozadu. Pro dodavatele hudby je zjištění těchto hodnot samozřejmostí.

Při načtení mapy je vytvořena nová vrstva a je jí přidělena hudební stopa, která je spuštěna okamžitě, jelikož na dané vrstvě není nic přehráváno:

```
TeraMus.add_layer(Consts.MUS_BEACH, Consts.BUS_GAME_MUSIC) # string konstanty
TeraMus.schedule_item(Consts.MUS_BEACH, load("res://.../beach_intro.tres"))
```

Ve chvíli, kdy hráč aktivuje akční sekvenci, je zažádána změna hudební stopy:

```
TeraMus.schedule_item(Consts.MUS_BEACH, load("res://.../beach_main.tres"))
```

Plynulý přechod a načasování přechodu jsou vyřešeny automaticky. Po skončení souboje je požádáno o změnu na konečnou stopu a o chvíli později jsou nastaveny dvě nové vrstvy pro vertikálně míchanou hudbu v následující herní sekci:

```
TeraMus.schedule_item(Consts.MUS_BEACH, load("res://.../beach_end.tres"))
# čekací funkce
TeraMus.add_layer(Consts.MUS_CAVE_BASE, Consts.BUS_GAME_MUSIC)
TeraMus.add_layer(Consts.MUS_CAVE_OVERLAY, Consts.BUS_GAME_MUSIC)
TeraMus.fade_layer(Consts.MUS_CAVE_OVERLAY, 0.0, 0.0) # okamžitě ztlumit
TeraMus.schedule_item(Consts.MUS_CAVE_BASE, load("res://.../cave_main.tres"))
TeraMus.schedule_item(Consts.MUS_CAVE_OVERLAY, load("res://.../cave_over.tres"))
```

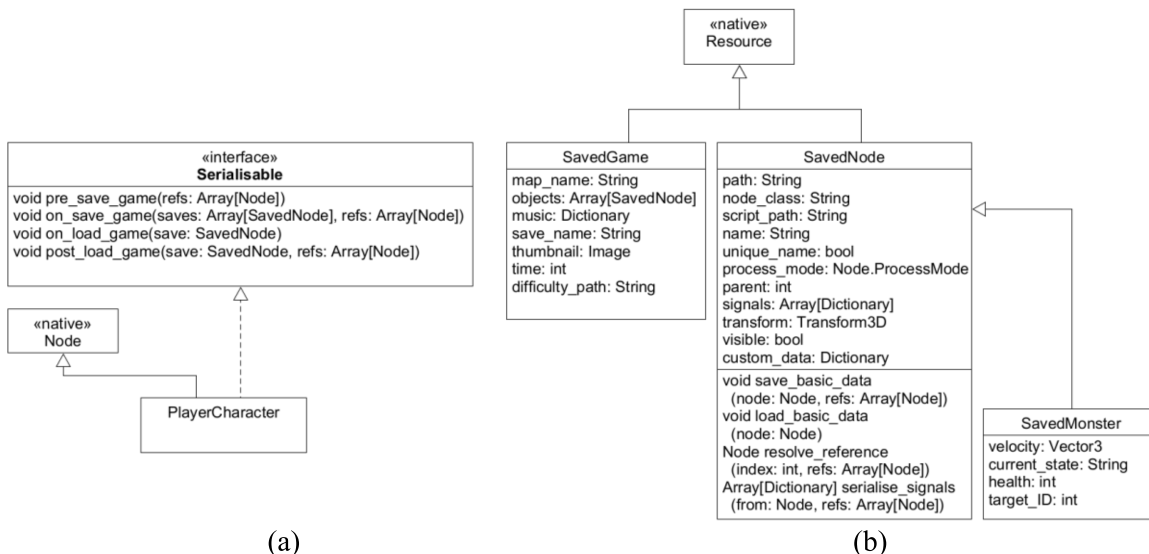
Míchání vrstev lze pak provádět voláním metody `fade_layer(layer, vol, time)`. Kdy toto provést je již v kompetenci uživatele, ne systému. Vytvořen byl například skript, který kontroluje, zda je hráč detekován nepřítelem: Pokud ano, zesílit překryvnou vrstvu; pokud ne, ztlumit ji. Metodou `await_switch(layer, post_preroll)` je taktéž umožněno čekání na změnu stopy a synchronizace herních událostí s aktivací hudební pasáže.

4.5 Serializace dat

Pro systém ukládání a načítání hry ve hře *Čekání na černokněžníka* byly aplikovány nejen postupy serializace z kapitoly 3.6, ale také byla použita rozhraní a vlastní `Resources`.

4.5.1 Formát dat

Celá uložená pozice se nachází v objektu typu `SavedGame` (dědící z `Resource`). Ten obsahuje metadata jako cestu k mapě, název uložené pozice, čas uložení, nastavenou obtížnost a především pole s objekty `SavedNode` (též `Resource`), které obsahují informace o serializovaných uzlech ve scéně: nezbytnosti k jejich instancování, informace o signálech a pozici ve světě. Informace dědících uzlů jsou uloženy jedním ze dvou způsobů: V objektu dědícím ze `SavedNode`, např. `SavedPlayer` nebo `SavedDoor`, nebo ve slovníkové proměnné `custom_data` v `SavedNode` pro jednodušší třídy. Struktura je zobrazena na obr. 25, část (b).



obr. 25 Struktura rozhraní `Serialisable` a zdrojů `SavedGame` a `SavedNode` (zdroj: vlastní).

4.5.2 Rozhraní a implementace

Využití rozhraní zde bylo důležité: Systém ukládání jako první shromáždí veškeré uzly ve scéně, implementující rozhraní `Serialisable` (viz obr. 25, část (a)), a volá na nich požadované metody.

1. `_pre_save_game` shromažďuje všechny uzly, které mají být uloženy. Ty, které ano, přidají referenci na sebe sama do pole `refs`. Příklad serializovatelného objektu, který není vždy chtěné ukládat, jsou poražení nepřátelé.
2. V `_on_save_game` probíhá samotné uložení dat do pole `Resources` typu `SavedNode`. Datové typy jsou uloženy přímo, referenční typy jsou vyhledány v poli `refs` a jejich index uložen jako číslo. Pokud není reference nalezena, jedná se o chybu.

Načtení taktéž probíhá ve více krocích, tak, jak bylo popsáno v kap. 3.6.2:

1. Je načtena mapa a veškeré serializovatelné uzly na ní jsou vymazány.
2. Podle uložených dat jsou vytvořeny nové instance uzlů. Systém ukládá tři hodnoty k určení, co a jak instancovat:
    a. Cestu k souboru scény
    b. Cestu k rozšiřujícímu skriptu uzlu, který definuje dědící třídu
    c. Název interní třídy, pokud skript nedefinuje vlastní třídu
3. První krok, popsaný v teoretické části: V metodě `_on_load_game` jsou deserializovány hodnotové typy jednotlivých instancí. Pro toto stačí pouze data v uložené hře.
4. Druhý krok: Metoda `_post_save_game` deserializuje referenční typy. K tomu musí nejprve být dostupné veškeré instance a jejich počet a pořadí musí odpovídat těm při ukládání. V rámci deserializace referencí jsou opětovně připojeny i signály.

4.5.3 Zajištění bezpečnosti

Jak bylo vysvětleno v kap. 3.6.1, části Bezpečnost vlastních Resources, neošetřené načítání `Resources` z úložiště počítače je bezpečnostní slabinou, jelikož `Resources` mohou obsahovat vložené skripty nebo odkazovat na jiné soubory v počítači. Pro potřeby bezpečného načítání uložených her byla proto použita knihovna *Safe Resource Loader* (Thomä, 2023b), jejíž nahrazující metodou `load()` je požadovaný `Resource` nejprve načten jako text a pomocí regulárních výrazů jsou v něm vyhledána možná nežádoucí data.

4.6 Uživatelské testování

Uživatelské testování bylo prováděno metodou ad-hoc (popsané v kapitole 3.2.2) na šesti uživatelích.

4.6.1 Průběh testování

Testeři byli instruováni hru dokončit dle vlastního uvážení běžným způsobem a posléze při hraní hry experimentovat. Při nalezení herní chyby měli odpovědět na následující otázky:

1) Co se stalo a kde došlo k chybě? Jaká byla očekávaná událost?
2) Je možné chybu zopakovat?
3) Stala se chyba po načtení uložené hry anebo po startu nové hry?
4) Jaký je Váš herní styl? (Běžný? Pokud možno optimální? Snažíte se najít zranitelnosti ve hře?) (viz rozdělení skupin v 3.2.2)

Vedle tohoto byli také dotázání na jejich celkový dojem ze hry a její nedostatky. Běžní hráči tvořili 4/6 testerů, ostatní dva se aktivně pokoušeli dostat mimo zamýšlené oblasti. Celkový dojem ze hry a jejího zpracování byl ode všech testerů pozitivní, vyzdviženo bylo grafické a zvukové zpracování, nicméně byly nalezeny určité nedostatky jak technického tak herně-návrhářského rázu (zde především z hlediska ovladatelnosti zbraní a obtížnosti).

4.6.2 Zjištěné technické chyby a jejich oprava

Zde jsou popsány některé nalezené technické chyby a způsob jejich opravy. Tyto úpravy byly promítnuty do finální verze hry.

Střelba po načtení hry

Při načtení hry právě držená zbraň začala střílet a pokračovala i bez držení aktivačního tlačítka do využití veškeré munice nebo do požadavku na změnu zbraně.

Zjištěný důvod

Při ukládání hry je uložen aktivní stav stavového automatu zbraně. Pokud se nachází ve stavu `fire`, je do něj také po načtení uvedena. Jednotlivé použití zbraně po načtení hry tak není neočekávané. Potíž nastává s podmíněným přechodem do klidového stavu zbraně `idle`: Ten je proveden, když hráči dojde munice, je odeslán požadavek na změnu zbraně, nebo je uvolněno tlačítko střelby. Pro tento poslední požadavek je však potřeba, aby nejprve bylo tlačítko střelby již drženo, což se v okamžiku načtení hry neděje.

Nevhodné řešení

Naskýtá se vložit kontrolu drženého tlačítka do metody `_enter` ve stavu `fire`. Tento postup je plně funkční, nicméně je jím porušena zásada jediné odpovědnosti – vstup z hráčovy myši nemá mít stavový automat zbraně na starosti.

Vhodnější řešení

Do metody `_ready()` v komponentě `PlayerInput` je přidáno vyslání signálu `released_weapon_fire`. Jak při prvotním spuštění úrovně, tak při jejím načtení je tak oznámeno ostatním komponentám, že není drženo tlačítko palby:

```
func _ready():
	released_weapon_fire.emit()
```

Svítící obloha

Hra z estetických důvodů využívá efektu volumetrické mlhy. Tento efekt reaguje na světla a stíny a lze mu přiřadit několik barev a hustotu. Během hry mohou tyto parametry být podle herního skriptování měněny. V několika případech mlha náhle trvale změnila barvu na konci úvodního souboje, při kterém se barva mlhy proměňuje.

Očekávaná událost: Na začátku souboje mlha zhoustne a začne měnit barvu z přirozeně modré na oranžovou. Na konci souboje se hustota i barva navrátí na původní hodnotu. **Chybná událost**: Na konci souboje se hustota i barva navrátily na původní hodnotu a posléze se barva náhle opět změnila na oranžovou.

Zjištěný důvod

Pro plynulý přechod mezi parametry mlhy je použita zabudovaná třída `Tween`, popsaná v kapitole 3.4.3, která umožňuje jednoduchým způsobem asynchronně prolínat mezi dvěma hodnotami. Metoda pro změnu parametrů (přijímající signál z uzlu se skriptem) vypadala takto:

```
func _on_set_fog_color(albedo: Color, time: float) -> void:
	var t := create_tween() # environment je členský Resource uzlu WorldEnv.
	t.tween_property(environment, "volumetric_fog_albedo", albedo, time)
```

Zároveň byly požadavky na změnu parametrů mlhy vysílány ze skriptu herní úrovně signálem `change_fog`:

```
# na začátku souboje
change_fog.emit(Color.hex(0xffc387ff), 30.0)
# na konci souboje
change_fog.emit(Consts.ENV_DEFAULT_ALBEDO, 6.0)
```

V případě, že hráč dokončil souboj za méně než 30 vteřin, došlo ke kolizi dvou nezávislých `Tweens`, upravující stejnou hodnotu. Technické detaily na straně enginu nejsou známy, výsledkem však byla nahlášená závada.

Řešení

K opravě byl do rozšiřujícího skriptu s přijímající metodou přidán `Tween` jakožto členská proměnná. Na počátku metody bylo zkontrolováno, zda proměnná drží referenci na již existující `Tween` a pokud ano, je na něm zavolána metoda `kill()`, která jej zastaví a invaliduje. Následně je vytvořen nový a ten aktivován:

```
var _t: Tween
func _on_set_fog_color(albedo: Color, time: float) -> void:
	if _t:
		_t.kill()
	_t = create_tween()
	_t.tween_property(environment, "volumetric_fog_albedo", albedo, time)
```

Poznámka

Skutečná metoda je mírně komplikovanější, protože zahrnuje vícero barev a hodnotu hustoty mlhy. Princip je však stejný, jen je použito více `Tweens`.

Znemožnění pohybu hráče nepřítelem

Nepřítel typu ostrorep, který na hráče útočí tím způsobem, že na něj skočí, mohl hráči přistát na hlavě. Toto znemožnilo hráči pohyb dokud nebyl nepřítel poražen, neskočil jinam, nebo hráč nezemřel.

Řešení

Jako řešení byla do komponenty pro pohyb monster přidána kontrola, zda zdola monstrum koliduje s jiným objektem typu `Actor` (hráč a monstra). Pokud ano, je mírně postrčeno vzhůru, čímž je hráči či monstru umožněn volný průchod.

5 Výsledky a diskuse

Použití vhodných návrhových metodik se na vývoji výstupní herní aplikace *En attendant ilkeätä noitaa: Čekání na černokněžníka* promítlo pozitivně: V pozdějších fázích vývoje nenastávaly komplikace plynoucí z unáhleného a nepromyšleného vývoje; naopak, jejich následováním bylo snadné implementovat požadované funkcionality podle měnících se nároků na design a prvky hry. Technické chyby, nalezené v rámci uživatelského testování, byly především menšího rázu nebo způsobené nepozorností.

Systém pro adaptivní hudbu *TeraMus* se v rámci výstupní aplikace osvědčil a je pomocí něj možné snadno přepínat mezi hudebními pasážemi a synchronizovat herní události s hudbou. Další vývoj by však měl směřovat k jeho refaktorizaci a vyjmutí práce s jednotlivými přehrávacími objekty do obalujících tříd: Bude tak usnadněno další potenciální rozšíření funkcionality systému, které by mohly zahrnout implementaci přechodových matic nebo automatickou generaci přechodových bodů podle zadaného tempa a taktu.

Osobním pohledem vývoj aplikací v enginu Godot hodnotí autor pozitivně: Editor je příjemný na práci, požadované funkce je snadné navrhnout a vytvořit, engine disponuje velkým množstvím předpřipravených funkcí a dynamická podoba GDScriptu a stromu uzlů umožňuje velice rychlé prototypování. Nativní implementace různých návrhových vzorů jsou též pozitivem. Dodatečným pozitivním kritériem pro použití enginu Godot, zejména v nekomerčních a edukačních projektech, může být fakt, že se jedná o svobodný a open-source software. Na druhou stranu měl editor tendenci při práci příležitostně padat a nabízenou funkci úpravy skriptů za běhu hry taktéž nelze považovat za stabilní.

Za zmínku však stojí současná úroveň jazyka GDScript ve smyslu přímého použití metodik popsaných v této práci: Absence rozhraní je problematická, jelikož duck-typingové metody nejsou dostačující náhradou za skutečné závazky vyplývající z použití rozhraní a použitá knihovna *gdscript-interfaces* je poměrně hrubým řešením. Absence rozdělení proměnných a metod na privátní a veřejné též není vhodná pro větší projekty. Pro budoucí projekty je proto vhodné zvážit použití jazyka C#, který svými funkcemi může být pro udržitelný a správně navržený kód vhodnější a je možné jej používat paralelně s GDScriptem. Potřeba bližšího než povrchního obeznámení se strukturou enginu z něj však

činí jazyk spíše pro pokročilejší uživatele. Samotná možnost výběru (a jejich kombinace) mezi jazykem pro začátečníky a pokročilejší je však určitě výhodou.

Další poznámkou k použití enginu Godot je kvalita dostupných informací týkající se vývoje v něm. Oficiální dokumentace je poměrně kvalitní a obsahuje jak referenční informace o třídách a metodách, tak návody pro začátečníky. Neoficiální zdroje a především různé návody již jsou méně kvalitní, resp. velice často popisují pouhé základy dané problematiky. Příkladem je serializace dat, kdy specifika ukládání a načítání referenčních typů je zdroji v naprosté většině případů zcela opomenuta.

Poslední připomínkou jsou některé funkce enginu, které zdánlivě přímo vyzývají k nevhodným praktikám, jmenovitě skupiny (popsané v kap. 3.4.1, části Skupiny) či již zmíněná metoda `has_method()`, a to z důvodu použití textových řetězců tam, kde se jeví zcela nevhodné.

Možná navazující témata na tuto práci zahrnují porovnání použití jazyků GDScript a C# v enginu Godot; porovnání tohoto enginu s jinými, např. Unity; nebo použití enginu Godot k vývoji herních aplikací pro více hráčů (multiplayer), aplikací pro virtuální realitu, či neherních aplikací, a to se zaměřením nejen na technický návrh, ale i herní design a použitelnost.

6 Závěr

Tato bakalářská práce se věnovala vývoji 3D akční herní aplikace v enginu Godot za využití vhodných softwarově-návrhových praktik.

Teoretická část se nejprve věnovala obecným postupům při návrhu a vývoji aplikací za pomocí objektově orientovaných jazyků, jmenovitě návrhovým vzorům, nejčastěji používaným diagramům UML, zásadám SOLID a také základním principům jako dědictví a kompozice. Část práce se věnovala představení a popisu samotného enginu Godot, jeho funkcí a nativních implementací návrhových vzorů. Tyto kapitoly spolu souvisely mj. nativními implementacemi návrhových vzorů v podobě funkcí enginu Godot.

Mezi další témata patřily postupy testování kódu a na uživatelích, které pomáhají nalezení programových chyb; metody pro adaptivní hudbu ve hrách, umožňující navození požadované atmosféry v kombinaci s interaktivitou; a metody serializace a deserializace dat, které jsou klíčové pro ukládání a načítání uložené herní pozice.

Praktická část práce zahrnovala především tvorbu výstupní akční herní aplikace *En attendant ilkeätä noitaa: Čekání na černokněžníka* zužitkováním poznatků z části teoretické: Obecně uplatnitelné postupy softwarového návrhu byly aplikovány spolu se znalostí funkcí a obecné struktury enginu Godot. Tyto postupy byly vysvětleny na případech v reálném vývoji.

Vysvětlené metody pro adaptivní hudbu byly implementovány v podobě herně nezávislého systému *TeraMus*, který byl ve hře využit pro veškeré přehrávání hudby. Metody serializace dat byly použity k umožnění uložení a načtení herního stavu pro možný opětovný návrat ke hře hráčem.

V neposlední řadě bylo provedeno uživatelské testování metodou ad-hoc s šesti uživateli. Cílem testování bylo odhalit technické chyby výstupní aplikace a celkový výsledný dojem. Ačkoliv hra jako celek byla hodnocena pozitivně, zejména z hlediska uměleckého zpracování, bylo testery nalezeno několik chyb. Díky použití vhodných metod se však nejednalo o zásadní architekturní nedostatky a byly snadno opravitelné, jak bylo provedeno v závěru praktické části.

V rámci diskuse bylo zhodnocena práce s enginem Godot obecně, která, přes menší nedostatky, byla autorem, jakožto člověkem na počátku zpracování práce zcela neznalým

s náležitostmi tohoto prostředí, hodnocena pozitivně. Zároveň byly navrženy možné další oblasti výzkumu, na které je možné se v souvislosti s enginem Godot v budoucnu zaměřit.

Výsledkem práce bylo obeznámení čtenáře s enginem Godot a použitím vhodných programově-návrhových postupů při vývoji v něm, které se ukázaly být aplikovatelné. Poznatky, zjištěné v této práci, mohou být čtenářem využity pro vývoj vlastních následných aplikací v enginu Godot nebo při rozhodování o volbě herního enginu pro budoucí projekty.

7 Seznam použitých zdrojů

ABBADI, Mohamed, 2016. Taxonomy of Game Development Approaches. In: *International GI-Dagstuhl Seminar on Entertainment Computing and Serious Games* [online]. Dostupné z: doi:10.1007/978-3-319-46152-6_6

ADICO, Samuel, 2019. Programming and Magic Numbers. *Medium.com* [online] [vid. 2024-02-11]. Dostupné z: https://medium.com/@samaddico/programming-and-magic-numbers-f766e0cd1369

AGILE MODELING, 2002. UML State Machine Diagrams: Diagramming Guidelines. *AgileModeling.com* [online] [vid. 2024-01-06]. Dostupné z: https://agilemodeling.com/style/stateChartDiagram.htm

ALEXANDER, Christopher, Sara ISHIKAWA a Murray SILVERSTEIN, 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford: Oxford University Press. ISBN 0-19-501919-9.

BELL, Donald, 2003. An introduction to the Unified Modeling Language. *IBM Developer* [online] [vid. 2023-10-17]. Dostupné z: https://developer.ibm.com/articles/an-introduction-to-uml/

BELL, Donald, 2004. The UML 2 class diagram. *IBM Developer* [online] [vid. 2023-09-17]. Dostupné z: https://developer.ibm.com/articles/the-class-diagram/

BITLYTIC, 2023. Finite State Machines in Godot 4 in Under 10 Minutes. *YouTube* [online] [vid. 2024-01-06]. Dostupné z: https://www.youtube.com/watch?v=ow_Lum-Agbs

BJÖRLING, Ola (ukiro), 2018. *OTEX* [online]. 2018. Dostupné z: https://doom.ukiro.com/otex-downloads/

BUGAYENKO, Yegor, 2015. Vertical and Horizontal Decorating. *yegor256.com* [online] [vid. 2024-01-06]. Dostupné z: https://www.yegor256.com/2015/10/01/vertical-horizontal-decorating.html

ČÁPKA, David, 2017. UML - Online kurz. *itnetwork.cz* [online] [vid. 2023-09-17]. Dostupné z: https://www.itnetwork.cz/navrh/uml

CODACY, 2023. Code Smells and Anti-Patterns: Signs You Need to Improve Code Quality. *Codacy.com* [online] [vid. 2024-02-14]. Dostupné z: https://blog.codacy.com/code-smells-and-anti-patterns

COLLINS, Tom, 2023. What is Grey Box Testing? (Techniques & Example). *BrowserStack* [online] [vid. 2024-01-23]. Dostupné z: https://www.browserstack.com/guide/grey-box-testing

DIGITAL EXTREMES, 2013. *Warframe*. 2013. London (Kanada): Digital Extremes.

FIREBELLEY GAMES, 2022. Using Composition to Make More Scalable Games in Godot. *YouTube* [online] [vid. 2024-01-05]. Dostupné z: https://www.youtube.com/watch?v=rCu8vQrdDDI

FOWLER, Martin, 2007. Design Stamina Hypothesis. *martinfowler.com* [online] [vid. 2024-01-03]. Dostupné z: https://martinfowler.com/bliki/DesignStaminaHypothesis.html

FOWLER, Martin, 2019. Technical Debt. *martinfowler.com* [online] [vid. 2024-01-22]. Dostupné z: https://martinfowler.com/bliki/TechnicalDebt.html

GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley. ISBN 0-201-63361-2.

GAUTAM, Shubham, 2023. Cohesion and Coupling in Object Oriented Programming (OOPS). *Enjoy Algorithms* [online] [vid. 2024-01-07]. Dostupné z: https://www.enjoyalgorithms.com/blog/cohesion-and-coupling-in-oops

ID SOFTWARE, 1993. *DOOM*. 1993.

ID SOFTWARE, 2016. *DOOM*. 2016. Rockville: Bethesda Softworks.

JENNINGS, Samuel, 2022. Custom Resources - A Godot Workflow GAME CHANGER. *YouTube* [online] [vid. 2024-01-07]. Dostupné z: https://www.youtube.com/watch?v=vzRZjM9MTGw

JOHN, Clay, 2024. Dev snapshot: Godot 4.3 dev 4. *Godot* [online] [vid. 2024-03-06]. Dostupné z: https://godotengine.org/article/dev-snapshot-godot-4-3-dev-4/

JONÁŠ, Martin, 2012. Návrhové principy: SOLID. *Zdroják.cz* [online] [vid. 2024-01-06]. Dostupné z: https://zdrojak.cz/clanky/navrhove-principy-solid/

KOUNOUKLA, Xeni-Christina, Apostolos AMPATZOGLOU a Konstantinos ANAGNOSTOPOULOS, 2016. Implementing Game Mechanics with GoF Design Patterns. In: *20th Pan-Hellenic Conference on Informatics (PCI)* [online]. Dostupné z: doi:10.1145/3003733.3003779

KŘIVÁNEK, Pavel, 2004. Statická vs. dynamická typová kontrola. *Root.cz* [online] [vid. 2023-11-15]. Dostupné z: https://www.root.cz/clanky/staticka-dynamicka-typova-kontrola/

KUSHNER, David, 2003. *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture*. New York City: Random House. ISBN 0-375-50524-5.

LINIETSKY, Juan, 2022. Godot 4.0 will discontinue VisualScript. *Godot* [online]. Dostupné z: https://godotengine.org/article/godot-4-will-discontinue-visual-scripting/

LINIETSKY, Juan, Ariel MANZUR a GODOT ENGINE COMMUNITY, 2014. Godot Engine 4.2 documentation in English. *Godot* [online] [vid. 2023-12-14]. Dostupné z: https://docs.godotengine.org/en/4.2/

LINIETSKY, Juan, Ariel MANZUR a GODOT ENGINE COMMUNITY, 2023. Godot 4.0 sets sail: All aboard for new horizons. *Godot* [online] [vid. 2023-07-12]. Dostupné z: https://godotengine.org/article/godot-4-0-sets-sail/

LOVATO, Nathan, 2021. The Events Bus Singleton. *GDQuest* [online] [vid. 2024-01-05]. Dostupné z: https://www.gdquest.com/tutorial/godot/design-patterns/event-bus-singleton/

MARTIN, Robert C., 2000. Design Principles and Design Patterns. *ObjectMentor.com* [online] [vid. 2024-01-06]. Dostupné z: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

MARTIN, Robert C., 2003. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River: Pearson Education Inc. ISBN 0-13-597444-5.

MATHEWS, Joshua F., 2022. Stop abusing strings. *Medium.com* [online] [vid. 2024-02-10]. Dostupné z: https://medium.com/@joshfeltonm/stop-abusing-strings-991f9538d9be

METZ, Sandi, 2009. SOLID Object-Oriented Design. In: *GoRuCo* [online]. Dostupné z: https://www.youtube.com/watch?v=v-2yFMzxqwU

MICROSOFT CORPORATION, 2021. Code contracts (.NET Framework). *.NET Framework Documentation* [online] [vid. 2024-01-22]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts

MICROSOFT CORPORATION, 2022a. Casting and type conversions. *C# Programming Guide* [online] [vid. 2023-10-15]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions

MICROSOFT CORPORATION, 2022b. Code metrics - Depth of inheritance. *Visual Studio Documentation* [online] [vid. 2023-09-16]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-depth-of-inheritance

MICROSOFT CORPORATION, 2023. Serialization in .NET. *.NET Fundamentals* [online] [vid. 2024-01-20].Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/serialization/

MILOJKOVIC, Nevena, Mohammad GHAFARI a Oscar NIERSTRASZ, 2017. It's Duck (Typing) Season! In: *IEEE International Conference on Program Comprehension* [online]. s. 312 – 315. Dostupné z: doi:10.1109/ICPC.2017.10

NIELSEN, Jakob, 2000. *Why You Only Need to Test with 5 Users* [online] [vid. 2023-10-19]. Dostupné z: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

NIGHTDIVE STUDIOS, 2017. May Update: Dev Diary Audio Edition. *Kickstarter* [online]. Dostupné z: https://www.kickstarter.com/projects/1598858095/system-shock/posts/1889009?

NYSTROM, Robert, 2014. *Game Programming Patterns*. B.m.: Genever Benning. ISBN 978-0-9905829-2-2.

O'CALLAGHAN, Eamon, 2021. What is duck typing in JavaScript? *Medium.com* [online] [vid. 2022-10-15]. Dostupné z: https://medium.com/@eamonocallaghan/what-is-duck-typing-in-javascript-f3eb10853361

PARR, Kealan, 2020. Anti-patterns You Should Avoid in Your Code. *freeCodeCamp* [online] [vid. 2024-02-12]. Dostupné z: https://www.freecodecamp.org/news/antipatterns-to-avoid-in-code/

PAVEY, Cameron, 2019. Understanding Types; Static vs Dynamic, & Strong vs Weak. *Medium.com* [online] [vid. 2022-10-15]. Dostupné z: https://medium.com/@cpave3/understanding-types-static-vs-dynamic-strong-vs-weak-88a4e1f0ed5f

PTERODON a ILLUSION SOFTWORKS, 2003. *Vietcong*. 2003. New York City: Gathering of Developers.

REFACTORING GURU, 2014. *Decorator* [online] [vid. 2024-01-06]. Dostupné z: https://refactoring.guru/design-patterns/decorator

RETRO STYLE GAMES, 2023. What Are Assets in Game Design? *RetroStyleGames.com* [online] [vid. 2024-02-08]. Dostupné z: https://retrostylegames.com/blog/what-are-assets-in-game-design/

RIEL, Arthur J., 1996. *Object-Oriented Design Heuristics*. 1st edition. Boston: Addison-Wesley Professional. ISBN 978-0321774965.

SCHELL, Jesse, 2008. *The Art of Game Design: A Book of Lenses*. Burlington: Morgan Kaufmann.

SCHULTZ, Charles P. a Robert Denton BRYANT, 2016. *Game Testing: All in One*. 3rd Edition. Herndon: Mercury Learning and Information. ISBN 1942270763.

SIEK, Jeremy, 2014. What is Gradual Typing. *Indiana University* [online] [vid. 2023-10-15]. Dostupné z: https://wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/

STEVENSON, Jamie a Murray I. WOOD, 2018. Inheritance Usage Patterns in Open-Source Systems. In: *40th ACM/IEEE International Conference on Software Engineering (ICSE)* [online]. Dostupné z: doi:10.1145/3180155.3180168

STOTTS, David, 2000. Design By Contract. *University of North Carolina* [online] [vid. 2024-01-23]. Dostupné z: https://www.cs.unc.edu/~stotts/COMP145/CRC/DesByContract.html

SVENSSON, Johan, 2021. Hierarchical Finite State Machine. *Medium.com* [online] [vid. 2024-02-27]. Dostupné z: https://medium.com/dotcrossdot/hierarchical-finite-state-machine-c9e3f4ce0d9e

SWEET, Michael, 2015. *Writing Interactive Music for Video Games: A Composer's Guide*. Boston: Addison-Wesley. ISBN 978-0-321-96158-7.

TEMPERO, Ewan, Hong Yul YANG a James NOBLE, 2013. What programmers do with inheritance in Java. In: *Proceedings of the 27th European Conference on Object-Oriented Programming* [online]. Berlin, Heidelberg: Springer-Verlag, s. 577–601. ECOOP'13. ISBN 9783642390371. Dostupné z: doi:10.1007/978-3-642-39038-8_24

THOMÄ, Jan, 2023a. From Past to Present: Ensuring Saved Game Compatibility. *Godotneers* [online]. Dostupné z: https://raw.githubusercontent.com/godotneers/saving-loading-video/main/godotneers-ensuring-saved-game-compatibility.pdf

THOMÄ, Jan, 2023b. Godot Safe Resource Loader. *GitHub* [online] [vid. 2024-01-19]. Dostupné z: https://github.com/derkork/godot-safe-resource-loader/tree/master

THOMÄ, Jan, 2023c. Saving and loading games with Godot. *YouTube / Godotneers* [online] [vid. 2024-01-20].Dostupné z: https://www.youtube.com/watch?v=43BZsLZheA4

THOMPSON, Tommy, 2023. The Origins of Half-Life's Finite State Machines. *AI and Games* [online] [vid. 2024-01-22]. Dostupné z: https://www.aiandgames.com/p/history2-halflife

TUTEMIC, 2023. *gdscript-interfaces* [online]. 2023. B.m.: GitHub. Dostupné z: https://github.com/tutemic/gdscript-interfaces

VALVE CORPORATION, 1998. *Half-Life*. 1998. Oakhurst: Sierra Online.

WHITMORE, Guy, 2003. Design With Music In Mind: A Guide to Adaptive Audio for Game Designers. *Game Developer* [online] [vid. 2023-07-08]. Dostupné z: https://www.gamedeveloper.com/audio/design-with-music-in-mind-a-guide-to-adaptive-audio-for-game-designers

WILKES, Andrew, 2021. Signals. *GDScript.com* [online] [vid. 2024-01-23]. Dostupné z: https://gdscript.com/solutions/signals-godot/

WULF, Nathan, 2022. Godot Tutorial: How do I structure my project scenes? Main scene + global autoload. *YouTube* [online] [vid. 2023-12-27]. Dostupné z: https://www.youtube.com/watch?v=a0UQ-t-vuzY

ZDOOM TEAMS, GZDOOM TEAMS a PŘISPĚVATELÉ, 1998. GZDoom repository. *GitHub* [online] [vid. 2024-01-20]. Dostupné z: https://github.com/ZDoom/gzdoom

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

obr. 1 Příklad UML třídního diagramu (zdroj: vlastní). 25
obr. 2 UML stavový diagram jednoduchého automatu na kávu (zdroj: vlastní). 26
obr. 3 Formální diagram konečného automatu na kávu (zdroj: vlastní). 26
obr. 4 Základní hierarchie objektů enginu Godot (Linietsky et al. 2014). 31
obr. 5 Fráze horizontálního přeskládání ve hře Doom (id Software 2016). 38
obr. 6 Vrstvy vertikálního míchání ve hře Warframe (Digital Extremes 2013). 38
obr. 7 Ukázka hry Doom v source portu GZDoom (id Software, 1993; ZDoom teams et al., 1998). 42
obr. 8 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní). 43
obr. 9 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní). 44
obr. 10 Snímek obrazovky z Čekání na černokněžníka (zdroj: vlastní). 44
obr. 11 Assety hry v editorech UDB a Cinema 4D (zdroj: vlastní). 45
obr. 12 Struktura hlavní scény hry (zdroj: vlastní). 45
obr. 13 Struktura hráčské postavy (zdroj: vlastní). 46
obr. 14 UML třídní diagram stavového automatu se stavy (zdroj: vlastní). 52
obr. 15 Stavový diagram a uzlová struktura zbraně – kouzelné hole (zdroj: vlastní). 52
obr. 16 Hlavní stavový automat (zdroj: vlastní). 53
obr. 17 Superstav alive (zdroj: vlastní). 53
obr. 18 Stavový diagram generické nepřátelské postavy (zdroj: vlastní). 53
obr. 19 UML diagram tříd projektilu a abstraktního a konkrétního efektoru.(zdroj: vlastní). 54
obr. 20 Struktura a ukázka projektilu blesku (zdroj: vlastní). 55
obr. 21 Struktura a vizuály efektu ohně (zdroj: vlastní). 56
obr. 22 Nástin funkce systému TeraMus pomocí třídního diagramu UML (zdroj: vlastní). 56
obr. 23 UML třídní diagram současné podoby systému TeraMus. (zdroj: vlastní). 58
obr. 24 Definice TeraStreams v editoru (zdroj: vlastní). 58
obr. 25 Struktura rozhraní Serialisable a zdrojů SavedGame a SavedNode (zdroj: vlastní). 60

8.2 Seznam zkratek

API Application Programming Interface

CC Creative Commons

DIP Dependency Inversion Principle

GoF Gang of Four

GUI Graphical User Interface

HUD Heads-Up Display

IDE Integrated Development Environment

OOP objektově orientované programování

OS operační systém

TFD Test Flow Diagram

UI User Interface

UML Unified Modeling Language

Přílohy

Příloha 1: Hotová verze hry *En attendant ilkeätä noitaa: Čekání na černokněžníka*, též dostupná ke stažení on-line, s možnými budoucími aktualizacemi oproti přiložené verzi: https://drive.google.com/file/d/1OxHKZCyleVF3DE3NhxAmlcZ4ocq_s9Pk

Příloha 2: Plný seznam ve hře použitých zdrojových kódů, fontů a zvukových efektů se jmény autorů, odkazy a licencemi